A unified approach to the study of asynchronous communication mechanisms in real time systems.

Ian G. Clark

Submitted for the degree of

Doctor of Philosophy

London University

King's College

May 2000

Abstract

This thesis presents a new application of analyzing Asynchronous Communication Mechanisms (ACMs) using Petri nets. This technique facilitates the testing of essential ACM operating properties: data coherence (concurrent reading and writing of data at the same location should not happen), data freshness (not reading out of date data) and data sequencing (not reading data in a new-old-new order). The technique allows for analysis under metastable conditions which cannot be avoided in an asynchronous environment, but have usually been omitted in the analysis of published ACM algorithms. The modelling techniques are described, along with the analysis methods and optimizations which allow the ACM models to be as compact as possible without omitting necessary detail. The method allows for fast automated analysis of ACMs therefore allowing design changes in the algorithms to be quickly analyzed, without the need to perform long formal proofs. The use of a common analysis method allows comparisons to be made between existing ACM algorithms and several results and comparisons are published.

To my parents

Acknowledgements

This work would not have come to fruition without the invaluable help received throughout the duration of this course of study.

I would like to express my deep gratitude to my supervisor Professor Tony Davies for his invaluable insight, encouragement and support throughout this work.

Words cannot express my gratitude to my friend, colleague and co-conspirator Fei Xia, whose endless enthusiasm, ideas and brilliant mind benefited much of the work reported here.

Many thanks must go to David Fraser, Sergio Velastin and Alan Kent of King's College London for their invaluable help and support. Also Hugo Simpson, Eric Campbell, Roy Bavister and Len Griffiths of British Aerospace for their support and many hours spent discussing technical issues and collaboration on various projects. Thanks also to Alex Yakovlev, Albert Koelmans and David Kinniment from Newcastle University for help and guidance during the joint COMFORT research project which is based on the work presented in this thesis, and that presented by Fei Xia in his PhD. thesis.

Thanks go to the EPSRC whose support of the COMFORT project (GR/L92471) has allowed me time to focus and complete this work.

I would like to thank all the people who have passed through the 10B lab at King's, including Rod Hiorns, Anthony Magrath, Julian Bean, Victor Bocharov, Ian Johnson, Ben Leslie, Chris Landone, Mohamed Aswat, Marc Price, Simon Kershaw, Michael Walters, Mark Townsend and Jonathon Macensie for making it a stimulating place to work and study. Also thanks to colleagues in the computing center at King's, Andy Harper and Marion Rosenberg, for their friendship and technical support.

Thanks must also go to Peter King, Talat Malik, Mustaq Mohammed and Nick Nicola in the Electrical Engineering Department at King's College London, for their continued technical support.

I am deeply grateful to my parents for their continual support and encouragement, and to many friends who have kept me sane. Thanks also to my girlfriend Penny who has put up with, helped and supported me during the long hours it took to complete this thesis.

Contents

1 Introduction **19**

1.1 Motivation 19

1.2 Publications and original contributions 20

1.3 Joint material 22

1.4 Thesis organization 22

2 Background **25**

2.1 Real-time systems 25

2.2 Existing and traditional forms of interprocess communication 28

2.2.1 Data communication 30

2.3 Fully asynchronous interprocess communication 33

2.3.1 Mascot and IDAs 36

2.3.2 Database operations 38

2.4 Metastability 40

2.5 Petri nets 46

2.5.1 A classical concurrent problem 48

2.5.2 Structural and behavioral analysis 50

3 Modelling of latches shared between concurrent processes **59**

3.1 Introduction 59

3.1.1 Processes and processors 60

3.1.2 Latches 62

3.1.3 Microprocessors 63

3.1.4 Atomicity 64

3.1.5 Registers 64

3.1.6 Wait-freedom and mutual exclusion 65

3.2 Methods used for determining the minimum accurate model 66

3.3 Two or three level modelling 67

3.4 One, two or three stage modelling 68

3.5 The modelling approach 72

3.6 Modelling the different variables 79

3.6.1 One shared variable 79

3.6.2 Two shared variables 83

3.7 Novelty and application 92

3.8 Conclusion 93

4 Modelling of Simple Asynchronous Communication Mechanisms 95

4.1 Introduction to ACMs and wait free communications 95

4.1.1 General ACM concepts 99

4.1.2 Open and closed access 101

4.2 Modelling techniques 102

4.2.1 Cyclic models 102

4.2.2 Safe Petri nets 104

4.2.3 Abstract models or individual instruction models 104

4.2.4 Control variables 106

4.2.5 Assignment and logic operators 108

4.2.6 The 'enable' place and atomic actions 111

4.2.7 A state and its complement 112

4.2.8 'Bad' states 113
4.2.9 OR arcs 114
4.2.10 Inhibitor arcs, safe Petri nets, modelling a Turing machine and priority 114
4.2.11 Modelling assumptions 115
4.3 The analysis program - 'Reach' 117
4.3.1 Convention for Petri Net models in this thesis 119
4.3.2 Memory optimization during reachability analysis with two or more concurrent processes 120
4.4 Asynchronous communication mechanism properties 128
4.4.1 Role models 128
4.4.2 Data coherence 130
4.4.3 Data freshness 131
4.4.4 Data sequencing 137
4.4.5 Checking for full asynchronism 139
4.5 Timing constraints for conditionally asynchronous algorithms 139
4.6 One slot mechanism 142
4.7 Two slot mechanism 144
4.8 Three slot mechanism 152
4.9 Conclusion 165

5 Analysis and comparison of single writer / single reader atomic variables 166

5.1 Introduction 166
5.1.1 Properties used in the comparison 166
5.2 Peterson's algorithm 170
5.3 Kirousis's $4b + 39$ algorithm 175

5.4 Tromp's algorithms 182
5.4.1 Tromp's $4b + 14$ algorithm 182
5.5 Anderson's algorithm 187
5.5.1 Anderson $4b + 9$ under metastable operation 192
5.6 Simpson's algorithms 193
5.6.1 Simpson's $4b + 7$ algorithm 193
5.6.2 Simpson's $4b + 6$ algorithm 199
5.7 Other algorithms 205
5.8 Conclusion 206

6 ACM design rules and future work **208**
6.1 ACM design rules 208
6.1.1 Use variable fan-out only in synchronous circuits 210
6.1.2 Use only 'copy', 'invert' and logical functions 211
6.1.3 Use different logical sequences in different processes - if possible 212
6.1.4 Use only single bit signals 213
6.1.5 Use local copies of shared variables 214
6.1.6 Use different variables for data slot index variables 215
6.2 Which algorithm fits these rules 215
6.3 Future work 216
6.3.1 Multiple writer / multiple reader mechanisms 219
6.4 Conclusion 219

7 Conclusion **220**
7.1 Positive results 220
7.1.1 Identification of how to accurately describe ACMs and metastability using Petri nets 221
7.1.2 A method of testing algorithms 222

7.1.3 Algorithm review . . . 222

7.1.4 Algorithm design criteria . . . 223

A The analysis program - *Reach* **224**

A.1 *Reach* run time options . . . 224

A.2 Model checking . . . 230

A.3 Reachability tree output from *reach* . . . 235

A.4 How to check if a model reduction is safe . . . 236

B Complete Petri net models of ACMs **241**

B.1 Peterson '$5b + 9$' Petri net models . . . 241

B.1.1 Model for testing coherence . . . 241

B.2 Kirousis '$4b + 39$' Petri net models . . . 243

B.2.1 Model for testing coherence . . . 243

B.2.2 Model for testing freshness . . . 251

B.2.3 Model for testing sequencing . . . 259

B.3 Tromp '$4b + 14$' Petri net models . . . 269

B.3.1 Model for testing coherence . . . 269

B.3.2 Model for testing freshness . . . 272

B.3.3 Model for testing sequencing . . . 277

B.4 Anderson '$4b + 9$' Petri net models . . . 284

B.4.1 Model for testing coherence . . . 284

B.4.2 Model for testing freshness . . . 285

B.4.3 Model for testing sequence . . . 288

B.5 Simpson '$4b + 7$' Petri net models . . . 293

B.5.1 Model for testing coherence . . . 293

B.5.2 Model for testing freshness . . . 294

B.5.3 Model for testing sequence 296

B.6 Simpson '$4b + 6$' Petri net models 300

B.6.1 Model for testing coherence 300

B.6.2 Model for testing freshness 301

B.6.3 Model for testing modified freshness 303

B.6.4 Model for testing sequence 306

List of Figures

1.1 Thesis structure. 24

2.1 Two and four phase hand shaking. 32

2.2 Mutual exclusion on a train track introduces waiting. 32

2.3 A generic ACM. 35

2.4 An example MASCOT network. 37

2.5 Set-up time (choose) and propagation delay (indicate) definition. . . 41

2.6 Metastability modelled with a hill and ball. 42

2.7 Comparison of latch critical input to output timing. 42

2.8 The different forms of metastability. 43

2.9 CMOS equivalent of Seitz's metastability filter. 45

2.10 The four core Petri net elements. 47

2.11 Petri net connection. 47

2.12 Examples of Petri net firings. 48

2.13 Two ways of modelling a single philosopher. 49

2.14 Three philosophers both picking up both chopsticks together. 50

2.15 Three philosophers picking up chopsticks separately. 51

2.16 An example Petri-net. 52

2.17 Self-loops in Petri-nets. 53

2.18 A variable assignment statement modelled in Petri-nets. 54

2.19 Petri-net example with an extra place. 57

2.20 Part of the infinite reachability tree for the Petri-net in Figure 2.16. 58

2.21 The minimum reachability tree for the Petri-net in Figure 2.16. . . . 58

3.1 Operator implementation in hardware and software. 61

3.2 Three level Petri net model of the asynchronous shared variable output. 69

3.3 Illustration of need for two-stage modelling for two-level asynchronous systems. 70

3.4 Petri net models showing single and two stage models for two level outputs. . . . 71

3.5 Illustration of need for three-stage modelling for three-level asynchronous systems. . . . 73

3.6 Petri net model showing two stage model for three level outputs. . . 74

3.7 Petri net model showing three stage model for three level outputs. . 75

3.8 Initial model of $\{1 : X := Y\}\&\{2 : Y := X\}$ using two stage modelling. 77

3.9 Different instruction interleaving possibilities depend upon shared variable paths. . . . 78

3.10 The time line representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on a single processor. . . . 80

3.11 The time lines representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on two processors synchronously. . . . 81

3.12 The time lines representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on two processors asynchronously. . . . 82

3.13 The time lines representing $\{1 : A :=?; x := B\}$ $\{2 : B := A\}$ executing synchronously. . . . 84

3.14 The time lines representing $\{1 : A :=?; x := B\}$ $\{2 : B := A\}$ executing asynchronously. . . . 85

3.15 Parallel time lines for $\{1 : A := B\}$ & $\{2 : B := A\}$ executing synchronously. . . . 86

3.16 Parallel time lines for $\{1 : A := B\}$ & $\{2 : B := A\}$ executing asynchronously. . . . 87

3.17 Two and three level state spaces for all versions of $\{1 : A := B\}$ & $\{2 : B := A\}$. . . . 89

3.18 Parallel time lines for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$ executing synchronously. . . . 90

3.19 Parallel time lines for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$ executing asynchronously. . . . 91

3.20 The state graph for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$. . . . 92

3.21 The reachable states of $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$. . . . 93

4.1 Different write and read types [Sim94]. . . . 96

4.2 Destructive write asynchronous taxonomy. . . . 100

4.3 Time line representation of open and closed access. . . . 102

4.4 A cyclic Petri net model of processing and communication. . . . 103

4.5 Complex cyclic Petri net model with processing element removed. . . 103

4.6 Instruction firing without changing state, except the control variable. 107

4.7 The control loop will always have one and only one token. 107

4.8 Places repeated for presentation clarity. 108

4.9 The assignment operator requires four transitions. 109

4.10 Logic operators modelled in Petri nets. 110

4.11 Textual model for '$a = b$ OR c'. 110

4.12 Example of how an 'enable' place is inserted to facilitate atomic actions. 112

4.13 Example textual Petri net with OR arcs 114

4.14 A priority shared channel implemented with an inhibitor arc. 115

4.15 A priority shared channel implemented with complementary places. . 116

4.16 A bounded priority shared channel implemented with complementary places. 116

4.17 An example 'reach' input file. 119

4.18 A small single process cyclic model. 121

4.19 A simple linked list used for visited list storage. 122

4.20 A second attempt using a tree of depth related to the model size, and generated by asking which places are marked or unmarked. 123

4.21 First attempt at splitting the problem into processes. 125

4.22 Second attempt at splitting the problem into processes, but with the full markings stored in a 2D array. 126

4.23 Format of a role transition. 129

4.24 'rab' role transition. 129

4.25 Petri net model for checking data coherence. 130

4.26 Simpson's first algorithm data freshness role transitions. 132

4.27 Simpson's second algorithm data freshness role transitions. 133

4.28 Modified first data freshness role transitions. 135

4.29 Modified second data freshness role transitions. 135

4.30 Time line showing the modified first data freshness model states. . . 136

4.31 Data sequencing role transitions. 138

4.32 A time line showing the data sequencing test. 138

4.33 Timing constraint, instruction A (or C) cannot start until instruction B has finished. 140

4.34 Petri net model of constraint for instruction A (or C) cannot start until instruction B has finished. 140

4.35 Timing constraint, instruction A (or C) cannot finish until instruction B has finished. 141

4.36 Petri net model of constraint for instruction A (or C) cannot finish until instruction B has finished. 141
4.37 Timing constraint, instruction A (or C) cannot start or finish until instruction B has finished. 141
4.38 Petri net model of constraint for instruction A (or C) cannot start or finish until instruction B has finished. 142
4.39 The one slot algorithm 143
4.40 One slot algorithm in algebraic form. 143
4.41 The one slot Petri net model 143
4.42 The one slot graphical Petri net model 143
4.43 One slot output from 'reach' 144
4.44 A two slot mechanism with no control variables 144
4.45 The two slot algorithm 145
4.46 Two slot algorithm in algebraic form. 145
4.47 The two slot writer Petri net model 146
4.48 Two slot $w0$ Petri net model 147
4.49 Two slot $writing$ Petri net model 147
4.50 Two slot $w1$ Petri net model 147
4.51 The two slot reader Petri net model 148
4.52 Two slot $r0$ Petri net model 148
4.53 Two slot rd Petri net model 149
4.54 Two slot coherence Petri net model 149
4.55 Two slot reachability output 150
4.56 Two slot model producing all initial conditions 151
4.57 Two slot timing constraint shown as a time line 152
4.58 Two slot writer model with timing constraint 153
4.59 Two slot reader model with timing constraint 154
4.60 Two slot data freshness writer model with timing constraint 155
4.61 Two slot data freshness reader model with timing constraint 156
4.62 Two slot data sequencing writer model with timing constraint 157
4.63 Two slot data sequencing reader model with timing constraint 158
4.64 The three slot algorithm 159
4.65 Three slot algorithm in algebraic form. 159
4.66 The three slot writer Petri net model 160

4.67 The three slot reader Petri net model 161
4.68 The three slot Petri net model of $w0$ 162
4.69 The three slot Petri net model of wr 162
4.70 The three slot Petri net model of $w1$ 162
4.71 Three slot bad states 164

5.1 Graph comparing the number of bits used by each algorithm with differing sized data slots. 169
5.2 Peterson's original algorithm. 171
5.3 Peterson's $5b + 9$ algorithm re-written. 172
5.4 Peterson's $5b + 9$ algorithm time line. 173
5.5 Peterson's $5b + 9$ data coherence test time line. 174
5.6 Kirousis $4b + 39$ algorithm (with modulo 3 arithmetic). 176
5.7 Time line showing Kirousis's $4b + 39$ algorithm. 176
5.8 Kirousis's $4b + 39$ algorithm rewritten in a PASCAL like form. 177
5.9 Time line showing Kirousis's algorithm under the data freshness test. 180
5.10 Time lines show the instruction interleavings on a bad route with data sequencing and data freshness roles highlighted. 181
5.11 Time line showing Kirousis's algorithm under the data sequence test. 181
5.12 Tromp's $4b + 14$ algorithm. 183
5.13 Tromp's $4b + 14$ algorithm rewritten. 184
5.14 Tromp's $4b + 14$ algorithm in algebraic form. 185
5.15 Tromp's $4b + 14$ algorithm time line. 186
5.16 Tromp's $4b + 14$ algorithm time line for data freshness testing. . . . 186
5.17 Tromp's $4b + 14$ algorithm time line for data sequence testing. . . . 186
5.18 Anderson's algorithm. 188
5.19 Anderson's $4b + 9$ algorithm re-written. 189
5.20 Anderson's algorithm as a time line. 190
5.21 Anderson's algorithm in an algebraic format. 191
5.22 Data freshness test for Anderson's algorithm. 191
5.23 Data sequening test for Anderson's algorithm. 191
5.24 Simpson's $4b + 7$ algorithm (in software form this algorithm is $4b + 8$ - see text). 195
5.25 Simpson's $4b + 7$ algorithm in algebraic form. 195
5.26 Simpson's 4 slot $4b + 7$ in hardware [Sim90b]. 196

5.27 Simpson's $4b + 7$ statement '$w0$' in Petri nets. 197

5.28 Simpson's $4b + 7$ statement 'wr' in Petri nets. 197

5.29 Simpson's $4b + 7$ statement '$r0$' in a two stage Petri net model. . . . 197

5.30 Simpson's $4b + 6$ algorithm. 200

5.31 Simpson's $4b + 6$ algorithm in algebraic form. 200

5.32 Simpson's $4b + 6$ four slot timing constraint for open access, with the start and end of the data access being merged with the end and start of the control part of the algorithm. 203

5.33 Simpson's $4b + 6$ four slot timing constraint re-written for closed access. The data accesses are not merged with the control parts of the algorithm. 203

5.34 Petri net of the Simpson $4b + 6$ timing constraint. 204

6.1 Avoid variable fan-out. 210

6.2 Hardware 'IF' statement. 211

6.3 Variable setting. 212

6.4 Be careful with variable interleaving. 213

6.5 A simple algorithm by Fraser illustrating the use of shared variables for the data slot index variables, which is not safe. 214

6.6 Anderson's algorithm simplified to $4b + 7$. 216

A.1 Core flowchart of 'reach'. 225

A.2 *reach* run time options. 226

A.3 A sample single ACM algorithm instruction. 230

A.4 The Petri net model in graphical form for the instruction given in Figure A.3. 231

A.5 The Petri net model in textual form for the instruction given in Figures A.3 and A.4. 231

A.6 Place label appearance count from the coverage output file. 232

A.7 Control variable loop check from the coverage output file. 233

A.8 Transition reference, remove and add place information from the coverage output file. 234

A.9 Place reference, remove and add place information from the coverage output file. 234

A.10 Transition firing and enabled information from the coverage output file. 235

A.11 Place label appearance in the complete reachability tree information from the coverage output file. 235

A.12 An example state space tree listing. 236

A.13 Example routes through a state space. 237
A.14 An example difference output listing. 238
A.15 An example difference output listing listed by common parent to common children. 239
A.16 An example difference output listing showing a missing route. 240

List of Tables

4.1 Comparison of storage efficiency for $Marked/\overline{Marked}$ trees with a standard linked list. 124

4.2 Comparisons between each algorithm presented, its memory usage and number of MALLOC calls. 127

4.3 Relationship between events and instructions. 129

4.4 An example execution sequence showing an invalid read (of non-fresh data) being masked. (Slot '10' has been omitted as it does not have any roles in this example run). 134

5.1 Comparison of ACM algorithms (P = Pass, F = Fail, X = Untested). 167

5.2 Key for Table 5.1. 167

5.3 Variables used within the Peterson $5b + 9$ algorithm. 173

5.4 Multi-value variables used within the Kirousis algorithm. 178

5.5 Variables used within the Tromp $4b + 14$ algorithm. 185

5.6 Variables used within the Anderson $4b + 9$ algorithm. 190

5.7 Shared variable metastability within the Anderson $4b + 9$ algorithm. 193

5.8 Variables used within the Simpson $4b + 7$ algorithm. 194

5.9 Shared variable metastability within the Simpson $4b + 7$ algorithm. . 199

5.10 Variables used within the Simpson $4b + 6$ algorithm. 201

5.11 Shared variable metastability within the Simpson $4b + 6$ algorithm. . 205

Chapter 1

Introduction

1.1 Motivation

This thesis proposes automated methods which improve the design and analysis of Asynchronous Communication Mechanisms (ACMs). Existing published ACM algorithms offer no explanations as to how the algorithms were designed, and the analysis methods seem to be unique to each author. Most of these published methods only offer a partial insight into the operation of each algorithm. Simpson's analysis method [Sim92a], using role models, is state space based and offers innovative insights into mechanism operation and failure. These role model techniques have been generalized into Petri nets by this author and Xia [XC00] and used as a common analysis method for existing ACM solutions.

This work enables all existing ACM algorithms to be analyzed with the same automated method. The results from this analysis can be directly compared, which has not previously been possible. Because the analysis method is executed on a computer, design changes are simple and quick to analyze: consequently their impact is quickly highlighted, improving the design cycle of ACMs. The majority of the existing techniques require formal proofs, which can be time consuming to develop, difficult to understand and hence open to human error.

This method is aimed to be a popularisation of the ACM design cycle. The existing formal methods are extremely difficult to follow without a strong background in formal methods, logic and mathematics. This method uses Petri nets to model system's states and conditions which are analyzed in the state space. From this state space analysis any failures in ACM operating properties are highlighted, with the conditions of the failure, and the way each condition arose from the initial state of the system.

Petri nets were introduced in the 1960's by Carl Petri. Petri nets have evolved today into a comprehensive theory which is widely supported by much published research material. Recently this research has extended to the field of system design, hardware design and more recently still, to the design and synthesis of asynchronous electronics.

With recent research [Cor] the ACM algorithms which can be designed, modelled and analyzed using Petri nets can now be synthesized into hardware using publicly available software packages.

With many ACM algorithms being compared it is possible to see design patterns, and design "do's and dont's" can be spotted. Software instructions which cause operating difficulties in an asynchronous environment can be highlighted. Many of the existing ACM solutions have been published by computer scientists and use software instructions which can be problematic in asynchronous environments, and most of the algorithms are presented with no thought given to hardware implementation. Also the possibility of metastability is often omitted from the analysis of such algorithms, and this can be a dangerous omission.

The technique presented in this thesis is used to give a new comparison of selected ACM algorithms. The published ACM algorithms are too numerous to permit the presentation of an exhaustive comparison. The technique has been used here to compare only single writer / single reader algorithms, but could easily be extended to include multiple writer / multiple reader algorithms.

1.2 Publications and original contributions

Some of the material presented in chapters 3 to 5 has been published as technical reports, conference / workshop proceedings, and one part has been included in a book published by Kluwer Academic Publishers on hardware design and Petri nets.

- The work in Chapter 3 on modelling metastable variables in Petri nets is an original contribution and has been published [CXYD98, XCYD97, XCYD98]. This material includes the concepts behind two stage modelling to avoid losing information from the state space during reachability analysis. An online bibliography of papers on the subject of metastability has also been published [Clab].

- The notational extension of Petri nets by the use of 'OR' arcs is an original unpublished contribution and is introduced in Chapter 4.

- The memory optimization techniques used for the reachability searching algorithm described in Chapter 4 have been published [Cla99].

- The work on analyzing Simpson's first four-slot algorithm and also the one, two and three slot mechanisms (Chapters 4 and 5) have been published [XC95, XCD97].

- An online bibliography of ACM algorithms has been published [Claa]: from this bibliography the algorithms analyzed in Chapter 5 were taken.

- Analysis of the Simpson two slot algorithm operating under a timing constraint has confirmed that the algorithm can maintain data freshness and sequencing. This is an original unpublished contribution.

- In depth analysis of Simpson's three slot algorithm (chapter 4) which prompted work at Newcastle University for the design of a self-timed version of the algorithm was published in [XC99]. The analysis of this self-timed algorithm was performed by the author of this thesis.

- Work on the modelling of the ACM data freshness property in Petri nets has been published [XC98a, XC98b, XYC99]. Work in this thesis also modifies the existing published data freshness roles [Sim92a, Sim97c].

- The data freshness material from Chapter 4 has also been formalized and published by Kluwer Academic in [XC00].

- The results from the comparison of ACM algorithms from Chapter 5 present the analysis of some algorithms previously untested in the state space. This has been accepted for presentation at the Asynchronous Interfaces (AINT'2000) Workshop in Delft, The Netherlands, 19-20th July 2000.

The results presented in chapter 5 from the initial studies of Simpson's 4 slot algorithm [Sim90b, Sim92a], are supported by a recent publication from Jiffry [Jif98]

1.3 Joint material

The initial work which started this line of research was performed jointly with Dr Fei Xia who has also presented the joint material in his own thesis [Xia00]. This material was jointly published in a Departmental Research Report [XC95] and also at a UK asynchronous forum [XCD97]. The joint work presented in this thesis is the Petri net models for data coherence of the one, two and three slot mechanisms (Sections 4.6, 4.7, 4.8) and also the Petri net models for Simpson's $4b + 7$ algorithm for data coherence and freshness (Section 5.6.1).

Owing to the co-operative nature of this work, in most parts it is not possible to completely separate the results and assign responsibilities to each participant and still make a meaningful presentation. Attempts have been made to concentrate on the contributions of the author in this thesis. However it has been impossible to completely avoid using material originally contributed by Xia.

1.4 Thesis organization

Chapter 2 introduces the background material for this thesis, and provides enough information for the thesis to be read without reference to other material. Different forms of interprocess communication are introduced in both synchronous and asynchronous modes. MASCOT [MAS87] (Modular Approach to Software Construction Operation and Test) is introduced. It is a method for designing systems based around the data flow within the system, how processing is performed and communication between different processing elements is facilitated, and its different types. Atomic operations and actions are introduced in the context of database operations. Metastability is introduced, with a discussion of how the phenomenon has been previously reported and some of the existing solutions to overcome it from the literature. Petri nets are introduced, which are the modelling medium used for the results presented in this thesis. The classical concurrent computer science problem of the dining philosophers is used as a modelling example to show how Petri nets can be used for such problems. Different types of Petri net analysis (structural and behavioral) are discussed and it is shown why most types are unsuitable for analysis of ACM models.

Chapter 3 introduces the modelling of latches shared between concurrent processes. Many algorithms may be incorrectly analyzed if the normal atomic assumption is made during the modelling process (i.e. each instruction is modelled as an atomic event). Examples of the problems caused and the details lost by this assumption are presented. The modelling of the metastable state is introduced. This is later employed in the testing of ACM algorithms in Chapter 5.

Chapter 4 presents the modelling techniques used in this thesis for the analysis of the ACM algorithms. This includes the modelling of timing constraints in classical Petri nets which do not include any time representation, 'OR' arcs are also introduced by the author to reduce model sizes. Memory optimization techniques used in the computer analysis are presented, along with an introduction to the analysis program written and used by the author. (A detailed presentation of this software is made in Appendix A). Three essential ACM operating properties of data coherence, data freshness and data sequencing are introduced in detail, with examples of how to test for them. Some simple algorithms are presented, modelled and analyzed. Their failure modes are discussed.

Chapter 5 introduces several of the published ACM algorithms, and shows how these can be modelled and analyzed. The analysis method presented in this thesis allows for algorithm comparisons to be made and these are presented at the beginning of the chapter. Each algorithm is analyzed for the three ACM operating properties. The results from these tests are presented with any timing constraints which may be required. For those algorithms which could be practically implemented in hardware, analysis is included taking into account the effects of metastability.

Chapter 6 presents future directions for this research. A section of ACM design "do's and dont's" is included which have been obtained during the analysis of the failure modes of some ACM algorithms.

Appendix A presents a more detailed insight into the analysis program (*reach*) which was written by the author and used in obtaining the results presented in this thesis.

Appendix B lists the full textual Petri net models of the algorithms compared in Chapter 5. The algorithms were not included in the main thesis text to retain focus on the core issues of the thesis.

A bibliography is included at the end of the thesis, and can be supplemented by two online bibliographies (Metastability [Clab] (approx. 210 entries) and Atomic communication [Claa] (approx. 275 entries)) which are published and maintained by the author.

This layout is shown graphically in Figure 1.1.

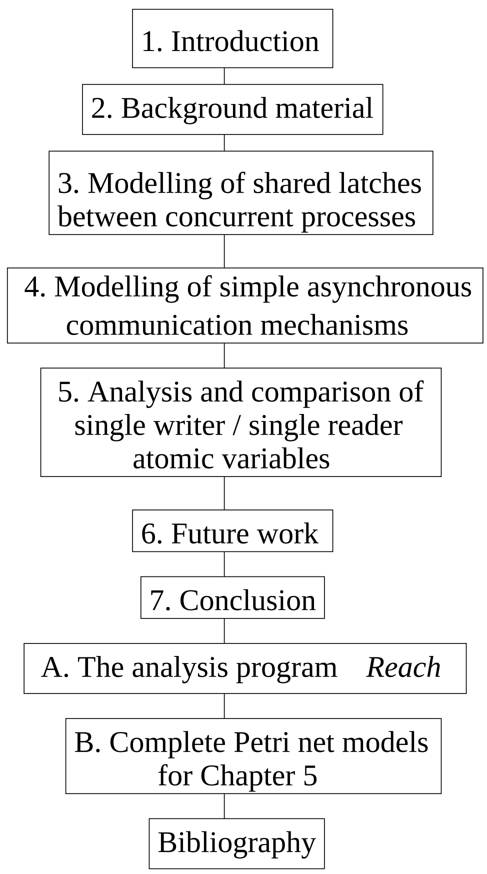


Figure 1.1: Thesis structure.

Chapter 2

Background

This chapter introduces the background material required within this thesis. The concepts of real time system design are introduced together with an introduction to using parallel processing units to enable concurrency. Existing traditional forms of interprocess communications are discussed and the ideas of fully asynchronous communication introduced. The MASCOT real time system development method is discussed which incorporates fully asynchronous communication mechanisms into its designs. Atomicity with in the area of database operations is described, and also the unavoidable phenomenon of metastability within unsynchronized hardware. Finally Petri nets are described along with numerous analysis methods, and a discussion as to why reachability searching was used for the results presented in later chapters of this thesis.

2.1 Real-time systems

A system is designed to perform a specified set of tasks, and in the case of a real-time system that task must be completed within a specified time. An example of a real time application is the steering of a missile control flap. A missile is an aerodynamically unstable object, so as soon as the missile deviates from the intended course, the flaps must be operated to correct the directional errors. If the guidance system and control system do not operate within a specified period then the guidance system will be unable to guide the missile's flight correctly. Unlike normal software which is executed once, produces a result and terminates, such systems are required to run continuously, normally in an endless loop structure. Therefore termination is not expected, and demonstrating termination in a finite time is not part of a proof of correct behaviour.

Once a real time application has been defined with its timing requirements, the process of design can commence. At the initial high level of design the functionality is of prime importance. Once the functionality is correct, implementation can proceed. Implementation may be split between hardware and software. Once an implementation has been defined its timing and performance characteristics can be investigated to make sure that any hard real time requirements have been met. (The term *hard real time* (HRT) is used here to designate that a system has timing requirements such that the system will fail if the timing requirements are not met).

As real time systems become more complex they require more processing power (operations per second) to maintain specified operating and timing constraints. There are two logical ways forward in meeting future more demanding specifications. Either to run the system faster or to introduce parallelism. However, both have limitations. The system speed can be increased as the silicon technology process size gets smaller, but this affects the ratio of gate delay to interconnect delay, making the interconnect delay a significant part of the propagation delay [JA89, Anc82]. The practice of distributing a common system clock around a whole device becomes impractical because the clock propagation delay across a large device means one corner of the device may operate a significant time behind the other. This will not only affect the distribution of the clock pulse but also of data. Note: no technology will be able to exceed the physical limit of the speed of light ($29.9cm/ns$). The interconnect and the dielectric material will affect the propagation delay (for example the speed of a signal in copper is approximately $20cm/ns$ [Tan90]). The clock signal is designed to make the whole system operate together in synchronization. This forces the whole system to be switching at the clock frequency which concentrates radiation at this fixed frequency (and its multiples) which leads to problems with electro magnetic compatibility (EMC). Power consumption is another topic causing concern in synchronous applications. Most modern digital hardware is constructed using CMOS (Complementary Metal Oxide Silicon) technology. CMOS logic gates consume most power when they are switching, compared to their normal rest or stable state. In a synchronous system those gates that are connected to the clock are repeatedly switching, even when not doing any useful work.

For the second option of parallelism to be implemented the problem must include (or enable) partitioning for concurrent execution, since no speed up would be achieved by a multiple processor architecture for a system without running concurrently. Typically the system being implemented has to be split into smaller areas of (local) activity, which can be individually implemented either in software (running on general purpose hardware elements) or in dedicated hardware (if higher performance is required). However, doubling the number of processing elements does not produce a doubling of performance. There is a data communication overhead between the processing elements. Care has to be taken over the type of communication used, as both processes may end up effectively running as a single sequential thread of execution if each is regularly waiting for data from the other to continue. For example suppose that process 1 produces some data that is required by process 2, process 2 waits for this and then continues while process 1 waits for data from process 2 before continuing. This, in practice, would result in worse performance than a single process, because of the added communication overhead.

"While the performance of computers and the speed of data networks continues to improve steadily, our ability to utilize these resources effectively does not. Data communications software has been a bottleneck in many high performance systems; it is often much slower than the hardware permits and it can be hard to establish its logical consistency. It is notoriously difficult to write software for a distributed system. It is even harder to prove rigorously the correctness of such software. In its simplest form, the problem is to design methods that allow asynchronously executing communicating machines to exchange information quickly and reliably and to prove that these methods, or protocols have certain desirable properties ...most protocols are designed in an ad-hoc manner. Little is known as to why some work and others fail. A designer needs to know how a correct protocol can be constructed from scratch and how that design can be matched to specific design and correctness criteria. The techniques that can be used to prove that a new protocol design is correct have long been considered too esoteric for real day-to-day use." [Hol91].

As these systems become larger and more complex, correctness and proof of correctness becomes a problem. Over a decade ago large software systems hit the problem of how to perform correctness analysis. Many large software projects were running into trouble with incorrect functionality. This was when software projects were typically up to one million lines of functional code.

Today the same problem has arisen with hardware because of its increased complexity. Typically new processors contain several millions of transistors, such large and complex designs can contain errors. An example is the well publicized Pentium floating point '*FDIV*' bug [Int94]. Despite extensive testing, the floating point arithmetic unit contained a design error which resulted in an incorrect numerical answer being generated for a few particular operands in a floating point division operation (e.g. '$1/824633702441 * 824633702441$' did not result in the expected answer of '1'). The problem of implementation correctness has been well reported for complex software but this is also a problem for hardware as devices use tens of millions of transistors. The correctness problem is also extended from software into hardware when devices use lookup tables, which are designed in software and then implemented in hardware (as with the Pentium '*FDIV*' bug). If the second option of introducing parallelism is used then this proof becomes even more complex because of concurrency between the individual processing elements, and their communication.

2.2 Existing and traditional forms of interprocess communication

Most existing multi-processor systems run synchronously. The synchronization can be either system wide (the whole system runs using a common distributed clock pulse), or each local processing element has its own local clock and the synchronization is performed during communication.

Synchronous communication is used for the same reason that most computer hardware is synchronous. In the design of digital circuits and digital computers, synchronous operation helps to facilitate structured design methods because everything operates in step with a known clock with fixed discrete intervals. The control of sequential operation is easier than parallel operation, which is inherent in all analogue circuitry. (Interestingly, early logic designs were asynchronous/parallel because their designers were familiar with analogue design. Later this was abandoned in favour of 'easier' synchronous/sequential designs). Sequential operation allows complex problems to be broken up into smaller and more manageable sub-problems. In the past an important factor was that physical hardware was expensive to build, and it was cheaper to make a single clock and use it globally over the whole design. For the same reasons, synchronous communication was easier to design and control. As early systems ran synchronously then logically it was desired that any communication also be synchronous and use existing clock signals. However, any input from the external world cannot be synchronous because nature is not clocked.

The correctness proof of a globally synchronous system is a complex problem because the whole system must be taken into account at the same time. Each element of the system must be included. If, however, the communications mechanisms are chosen to give a temporal break between the processes, then any time calculations are made much simpler because it can be assumed that data acquisition is being handled autonomously elsewhere in the system. Therefore only the time calculations for the current processing element need to be determined. Then a total delay of all the elements in the data flow path give an overall indication of the systems performance. (A previously reported method of testing parallel systems was to run them using controlled or logical clocks which affect the system being monitored [CZTS99], this is not acceptable for real-time systems).

The simplest solution is for the two communicating processes to synchronize using some form of handshaking. There is no buffer implemented between the two processes. This means that one of the processes will wait for the other process to service its request for data communication. Such solutions are well documented in the literature.

The existing synchronous data communication solutions which use some form of buffer can cause blocking, or mutually exclusive access to data being communicated. This inherently causes waiting which may be unacceptable. For example, if a FIFO (First In, First Out) buffer is used as a communication channel between two processes, when the channel is empty the reading process is denied access and forced to wait. When the channel is full, the writing process is forced to wait. (An exception to this is the non-blocking FIFO buffer [YKXK98]).

If, however, communication between processes is to have no access restrictions on the data, then asynchronous communication may be required.

2.2.1 Data communication

Asynchronous communication between one (or more) processes implies a lack of regularity or periodicity (e.g. not synchronized with a common clock) and no blocking of reading or writing access to the data being communicated. Many existing solutions claim to be asynchronous but are not fully asynchronous. Therefore the existing definition of asynchronous communication is refined. The reason for introducing this new classification which differs from the established one is that the mechanisms investigated in this thesis require no form of synchronization to perform data communication.

Dictionary definitions of whether a particular type of communication is asynchronous or synchronous are inappropriate. Several factors need to be included in the definition, including blocking, waiting and mutual exclusion. The definition of 'asynchronous' according to 'The Chambers Dictionary' is 'lack of synchronism or correspondence in time', and the definition of 'asynchronous transmission' is 'transmission of electronic data along communication lines, upon which time intervals between characters vary and data is transmitted when available', whereas 'synchronous transmission' is defined as 'transmission of data along communication lines at a fixed rate and in a continuous string'. For example, the RS232 standard is defined as asynchronous, but in fact the data within an RS232 packet is sent synchronously, i.e. each bit is sent with a uniform spacing, and start and stop bits are sent to facilitate synchronization between the communicating processes. It can be seen from the Chambers definition that RS232 is defined as asynchronous because it only transmits data when it is available or the receiver is ready for it, so there is not necessarily a continuous stream of data. Waiting is possible if the receiver is not ready for data and the sender is waiting to send, or the receiver wants data but the sender has none to send.

ATM (Asynchronous Transfer Mode) [Sta00] is another example of a communication method that is labelled as asynchronous. ATM is also known as isochronous which means it handles time critical data, but it is not truly asynchronous (isochronous is defined as "something which happens at regular intervals"). When data is to be transmitted a path is negotiated between the two communicating processes, which must involve waiting while this is being performed. Also both processes must be synchronized while the data is being transmitted.

Using these definitions any communication which cannot provide a continuous stream of data would be classed as asynchronous, as RS232 was classified. A more representative definition of synchronous and asynchronous communication, should be based on the way the actual data is transmitted, and how the two communicating processes interact with each other. This should include a class called semi-synchronous or semi-asynchronous. Any form of communication which requires the two communicating parties to synchronize with each other should be labelled as synchronous communication. Forms of communication including *hand shaking* and *critical sections* (mutual exclusion) should be classed as semi-asynchronous, as neither communicating process is required to synchronize with the other, but waiting is involved, whether for an acknowledge signal or access to the communicating resource.

Hand shaking is the method of communication, where, for each piece of data sent, an acknowledge signal is returned. There are two forms of hand shaking, two phase and four phase. Both two phase and four phase hand shaking use two lines to control the communication. The two phase protocol uses pulse edges on one line to indicate 'start of event' and pulse edges on the second line to indicate 'event done'. Both rising and falling edges are used with the two phase protocol. In the four phase protocol, the rising edge of the request line is used to indicate 'start of event', the rising edge of acknowledge is used to indicate 'event done', then the falling edge of request is not used and the falling edge of acknowledge is used to indicate that the system is ready for the next event. Both these are shown in Figure 2.1. There are variations on these definitions but the principles remain the same.

A *critical section* is a section of code in a concurrent system that is controlled in such a way that only one process can enter and execute it at a time. A critical section is the software equivalent of mutual exclusion in hardware. Within the critical section will be access (reading or writing) to the data which is being communicated.

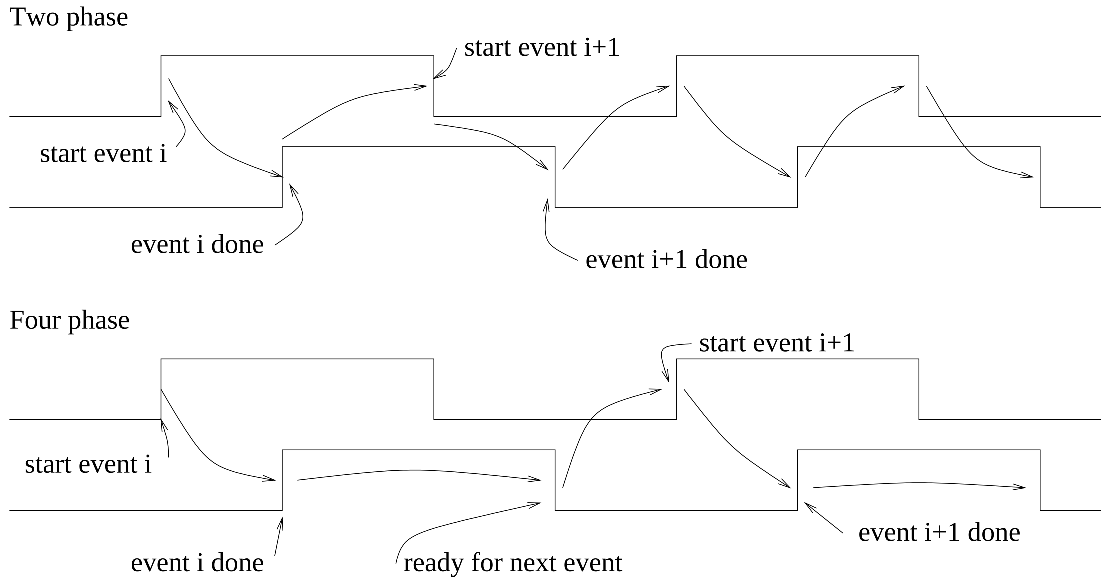
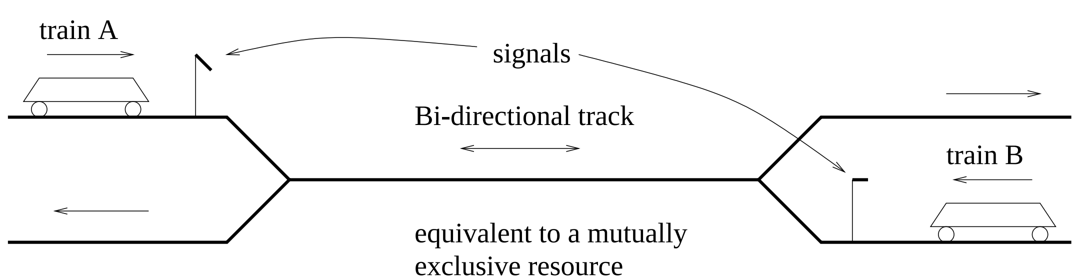


Figure 2.1: Two and four phase hand shaking.

Both critical sections and mutual exclusion implement a blocking type of communication, i.e. when one process is accessing the exclusive area or code, then the other process can be forced to wait, so its access is blocked, see Figure 2.2. The idea that 'hardware and software are logically equivalent' [Tan90] implies that an algorithm implemented using software can also be implemented in hardware if required. In this case the equivalent of a critical section in software is called mutual exclusion if implemented in hardware, and both can perform the same logical function. The choice of implementation medium is dependent on location, cost, speed, reliability and the frequency of system changes and updates.



Train A and Train B are travelling in opposite directions. The shared piece of track can only be used by one train at a time, so the mutually exclusive access to this track means one train must wait for the other.

Figure 2.2: Mutual exclusion on a train track introduces waiting.

A FIFO (First In First Out) buffer, can be conditionally asynchronous. While the buffer can be operated so that it is never full or empty then asynchronous operation can be maintained with the writing process always able to write in new data, and the reading process always able to obtain data as required. However if the buffer becomes full or empty, one process is forced to wait for the other process to access the buffer. A variation on this is the asynchronous FIFO buffer [YKXK98]. This asynchronous FIFO can always be written to, so data can be lost by over-writing, and the buffer never becomes empty because the last piece of data is never removed, so data read from the FIFO can be read repeatedly. (Note: there will always be a delay to the data signal caused by the writing of the data into the buffer).

These examples either induce some form of waiting, mutual exclusion / critical section, or blocking, under certain operating conditions (except for new work on a FIFO buffer with a non-blocking interface [YKXK98]). If communication is required in a real time environment or a safety critical environment then waiting or blocking may be unacceptable. In any system where waiting or blocking is unacceptable, then data loss and duplication has to be tolerated. This is a required property of applications which involve reference data. (An example of reference data is data which is regularly sampled, but is required by different processes at differing times and rates. Only the latest sample is required, not every sample).

2.3 Fully asynchronous interprocess communication

With asynchronous systems many of the above mentioned problems with synchronous systems and communication can be avoided. This section introduces fully asynchronous communication which aims to achieve communication without any form of blocking, waiting, mutual exclusion or timing interference between the communicating processes.

Fully asynchronous communication allows the writing and reading processes access to respectively write or read data at any time. Lynch [Lyn96] describes this as "reading and writing are permanently enabled". The writing process uses a destructive writing algorithm so that new data can always be written. The reading process uses a non-destructive reading algorithm so that data can be re-read if nothing new has been written. A fully asynchronous algorithm should always attempt to read the most recently written data, even if previously written data has not been read. Fully asynchronous communication is unidirectional i.e. one process will only ever be a writer and another will only ever be a reader, no process can ever be both a writer and a reader to one algorithm. Extensions to this simple one writer one reader algorithm have been presented in the literature with multiple readers, multiple writers, or both, but each is still a separate process.

If fully asynchronous communication is required there are side effects to be considered. Fully asynchronous communication implies that there is no temporal connection between the two communicating processes (i.e. a temporal firewall). In a large real time system, it will often be necessary to predict how quickly internal elements operate. With such a temporal firewall between the processing elements, this prediction is much simpler because waiting for data can be omitted; as with most asynchronous implementations there are overheads (hardware cost and size). For example, with hand shaking multiple wires are required for the communication of the control signals, usually two wires, one for communication in either direction. This is a significant overhead for serial data which requires only two wires for data transmission. For fully asynchronous communication the overhead comes with the requirement of multiple data areas to facilitate non blocking access to data for both the writing and reading processes, there is also the overhead of control or steering logic to control which data area each process accesses. The purpose of the control or steering algorithm is to maintain the operating requirements of data coherence, data freshness and data sequencing, which are defined precisely later in this thesis.

It is important to realize that fully asynchronous communication cannot employ mutual exclusion, critical sections, hand shaking, semaphores, waiting or blocking in the transmission of data. As already mentioned multiple data areas are required, which are referred to in the existing literature as slots, buffers or tracks (in this thesis 'slots' is the preferred term). Multiple slots are required because only one process can access a slot at any one time. With only one slot, both processes cannot have access to data at the same instant in time. The different algorithms use various ways to transmit the data. Some algorithms only write or read a single slot in each cycle of the algorithm, others write the same data in several slots, and then read from several slots and the controlling algorithm defines which copy of the data is used. By allowing only one process to access a single slot at any one time, the risk of metastability[1] is removed from the data path of the algorithm. However, metastability cannot be completely removed from the algorithm. It is shifted to the controlling algorithm because both accessing processes set and test common shared variables. To reduce the effect of metastability on this interaction the shared control variables should ideally be binary bits, which are the smallest available control variables. Some published algorithms do not use binary control variables and, as the results presented in this thesis will show, some operating problems are introduced. With the possible presence of metastability in the control variables the algorithm should ideally be restricted to simple logical elements, but some algorithms make use of more complex operators which can introduce operating abnormalities. The algorithms for fully asynchronous communication are usually implemented in hardware because the communicating processes are often executing on separate processors, for which no physical location is specified. When implemented in hardware these algorithms are referred to as Asynchronous Communication Mechanisms (ACMs). (A generic ACM is presented in Figure 2.3).

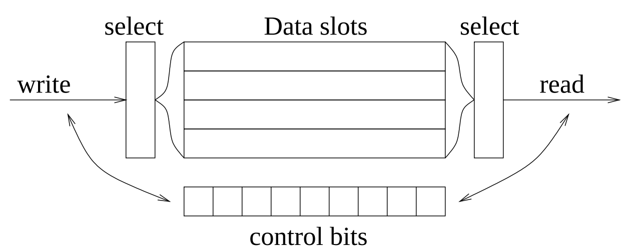


Figure 2.3: A generic ACM.

A subset of fully asynchronous communication is conditionally asynchronous communication. During the design and testing of an asynchronous communication algorithm, it may be discovered that under certain operating conditions the algorithm may not maintain one of the operating requirements. If this is the case when a certain set of operating conditions is true, then the algorithm can be claimed to be conditionally asynchronous if the operating condition causing the failure can be excluded from the operating environment. These operating constraints relate to the duration of concurrent algorithm instructions.

The proof of correctness of existing published ACM solutions has been performed

[1]Metastability is defined and discussed later in this chapter

using distinct methods, unique to each author. These solutions are therefore difficult to compare, and considerable time is required to understand the nuances of each algorithm described using a different approach. The single method of this thesis is a consistent approach using Petri nets. By using a consistent method, ready comparison between algorithms becomes easy.

2.3.1 Mascot and IDAs

A high level design approach (MASCOT) is introduced in this section. MASCOT can use ACMs for communication among its parts.

"MASCOT is a Modular Approach to Software Construction Operation and Test" [MAS87]. A key feature of MASCOT is that the design is expressed as a network of independent components. Each individual component can be developed independently of the rest, and the components can then be brought together. The core of MASCOT is the flow of data around a network, controlled by the concurrent execution of sequential processing elements [Sim86]. The two core MASCOT elements are the 'activity' and the 'IDA'. The 'activity' (≡ a concurrent process / single thread of execution) is where all processing of data is performed, and the 'IDA' or 'Intercommunication Data Area', where data is stored or communicated through the system and between the activities. At a higher level, 'subsystems' are used to facilitate hierarchy. A subsystem can represent a functional block at a high level of abstraction, but at the lower levels, the subsystem is broken down into its component activities and IDAs.

The whole network is contained within an element called a 'system'. Data travels through a network using 'paths', graphically represented by arrows which indicate the direction of travel. The data paths connect to IDAs and activities through 'windows' and 'ports'. There must always be an IDA between two activities, but two IDAs may be directly connected together indicating that data is just being communicated without any processing. There are also other components called 'servers' and 'devices', which represent the interface between a MASCOT network and its surroundings. Devices represent the external objects, and servers are used to control and transfer data to and from a particular device. A simple MASCOT network containing the core MASCOT elements is shown in Figure 2.4. A full description of the MASCOT method is beyond the scope of this thesis, and so for details the reader is referred to the MASCOT Handbook [MAS87].

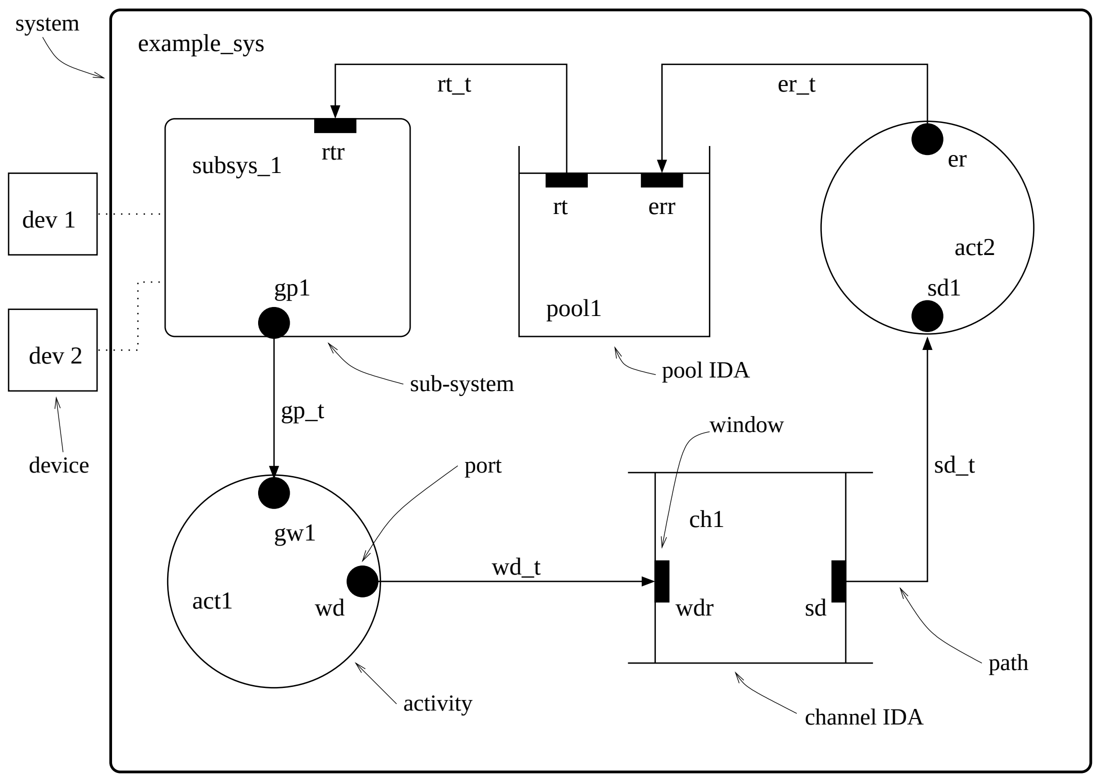


Figure 2.4: An example MASCOT network.

The MASCOT IDA has several different forms depending on the required properties of the communication. Two types of IDA are the ‘pool’ and ‘channel’. The channel is characterized by a destructive read operation and a non-destructive write operation. Therefore a channel can become empty of data, and because it will have a finite capacity it can also become full. This is usually implemented using a form of FIFO (First In First Out) buffer. The second core IDA, the pool, is characterized by having a non-destructive read operation and a destructive write operation. A pool is suitable for reference type data which an activity may wish to periodically consult or update. A pool will always contain data because it cannot be emptied, and it is never full because a new write will overwrite existing data within the pool, updating the reference information. The MASCOT pool is implemented using a buffer with multiple data areas of which the writing to and reading from is controlled by a control algorithm which ensures that certain operating properties are maintained (these operating properties are defined later in the thesis).

A MASCOT network does not define the implementation of the system it describes. Therefore two activities may be executed concurrently on a single CPU

(Central Processing Unit) by using some form of multi-tasking, or on two independent CPUs. More importantly, if the activities are executed using the latter option there is no specification of the location of the individual CPUs, local or remote, and also it specifies nothing about the clocking operations of the individual CPUs. Therefore the CPUs may be internally synchronous or asynchronous, and also the communication between them maybe synchronous or asynchronous. This is important so that MASCOT can be used for the design and development of HRT systems, where the temporal progress of individual activities must be measurable [Cam92]. With the pool type IDA, because data is always available, and writing can always take place, there is communication with no temporal connection between two activities. Therefore the prediction of temporal properties of small areas of a MASCOT network can be established without temporal information from other connected parts of the network.

A multiprocessor architecture called the Data Interchange Architecture (DIA) has been developed for the implementation of MASCOT systems [Sim90a, Sim92b]. The DIA is a 'locally synchronous, globally asynchronous' architecture, which means each processor is synchronous, but communication between the processors can be asynchronous [Cha84]. A demonstrator using this architecture called the DIADEM is reported in the literature [Cam91], which uses physical implementations of the IDAs described here. The MASCOT method is used for the overall system design and is then migrated onto the DIA architecture using the DORIS tool set [Nap93, Tho91]. Another such architecture was reported in [Pea99].

The subject of this thesis is primarily related to the pool IDA, and specifically with implementations of its fully asynchronous form.

2.3.2 Database operations

The terms 'atomic action' or 'atomicity' are used heavily in database theory and terminology [Bac92, Bel92, CDJ84]. These are the database equivalent to wait free communications and are introduced here. These terms are used in reference to operations performed on multi-user databases to indicate that any operations on the database do not affect its integrity, i.e. no operation on data within a database may be allowed to corrupt another and therefore risk corruption of the data within the database itself. The 'atomic' label conveys the concept that no matter what operations are performed on the database, their effect can be broken down into individual and separate sequential operations on the database.

An example of this type of operation is in the maintenance of a bank account. If two operations were performed on a single account at the same time, corruption of the amount of money in the account could occur. If for example you pay a sum of money into your account at the same time as another bank is cashing a cheque you wrote last week to your Uncle Fred. Suppose that at the start of the operation you have 1000 units of currency in your account. You are paying in 250 units into your account, and Uncle Fred's cheque is for 400 units. The correct amount in your account after these transactions is 850 units. (In this case the order of the two operations is not important, because you cannot go overdrawn with the large initial amount). However, assume that the two operations occur concurrently without exclusive access to your account. Both processes take a copy of your account details, one adds the 250 units to its copy, the other removes the 400 units from its copy. Both then 'write' their new calculated value back to your account. When you go to check your balance, you will end up with either 600 units (the debit was last to write) - the banks advantage, or 1250 units (the credit was last to write) - your advantage. An alternative operation would involve the two processes operating slightly staggered with regard to each other. One operation has occurred and the new account value is being written back into the account with some digits not yet updated. At this instant the second operation takes a copy of the value in the account. The value being read might not be coherent. If the withdrawal had been first the range of possible values include 0000 and 1600, if the deposit had occurred first the possible range includes 1050 and 1200. This is an application where mutually exclusive access, or some form of atomic action on the data is required. With atomic actions performed on the account the correct final value would be maintained, implying that a single writer is essential, but multiple readers are acceptable.

2.4 Metastability

Two stable states are required to implement the effect of memory in digital logic. Metastability is a phenomenon which can occur when a latch's input timing requirements are not respected, and there is insufficient energy to change properly from one state to the other. ACMs inherently use latches between two unsynchronized processes, and therefore metastability is possible within the mechanisms and is introduced in this section.

The first thing to state about metastability is that it is unavoidable. Any dynamic system which has two or more stable states will have unstable operating regions. This is because each stable state is a region of attraction and regions of attraction will abut, but can never overlap. Even if the gain of the active devices being used is infinitely large, there will still be a ridge, no matter how small, between the regions of attraction and at some point on the ridge is an unstable equilibrium point. This is crudely shown in Figure 2.6 as a ball rolling over a hill between two valleys. The two valleys represent the two stable states of a system, and the ball must be given enough energy to make it past the peak of the hill to the other region of attraction or stable state.

When latches are used in any circuit, the system designer is required to make the latches operate maintaining their set-up times to guarantee correct well behaved output operation. This is usually manageable in most synchronous systems, because the controlling clock can be set up to act at a time dictated by the slowest element in the clock's domain. However when using latches in an asynchronous environment this set-up time cannot be guaranteed. The set up time can be defined from the minimum period needed for the internal circuitry of the latches to store the energy required during a change of state.

Metastability can also be caused by rogue clock pulses (e.g. short transient 'spikes') being supplied to the latches. These rogue pulses may not have enough energy to clock the device properly, causing unpredictable results.

Metastability is reviewed in [CVH91, COL72, CM73, Cha83, Dav89, Dav90, Dav99, FD79, HC87, HEC89, Jac85, JA85, JA89, KD90, KYG97, Mar77, Mar81, Sei80, She92, Wak87, Wel96], and other significant material is listed in [Clab] (approx. 210

entries). A short description is included here.

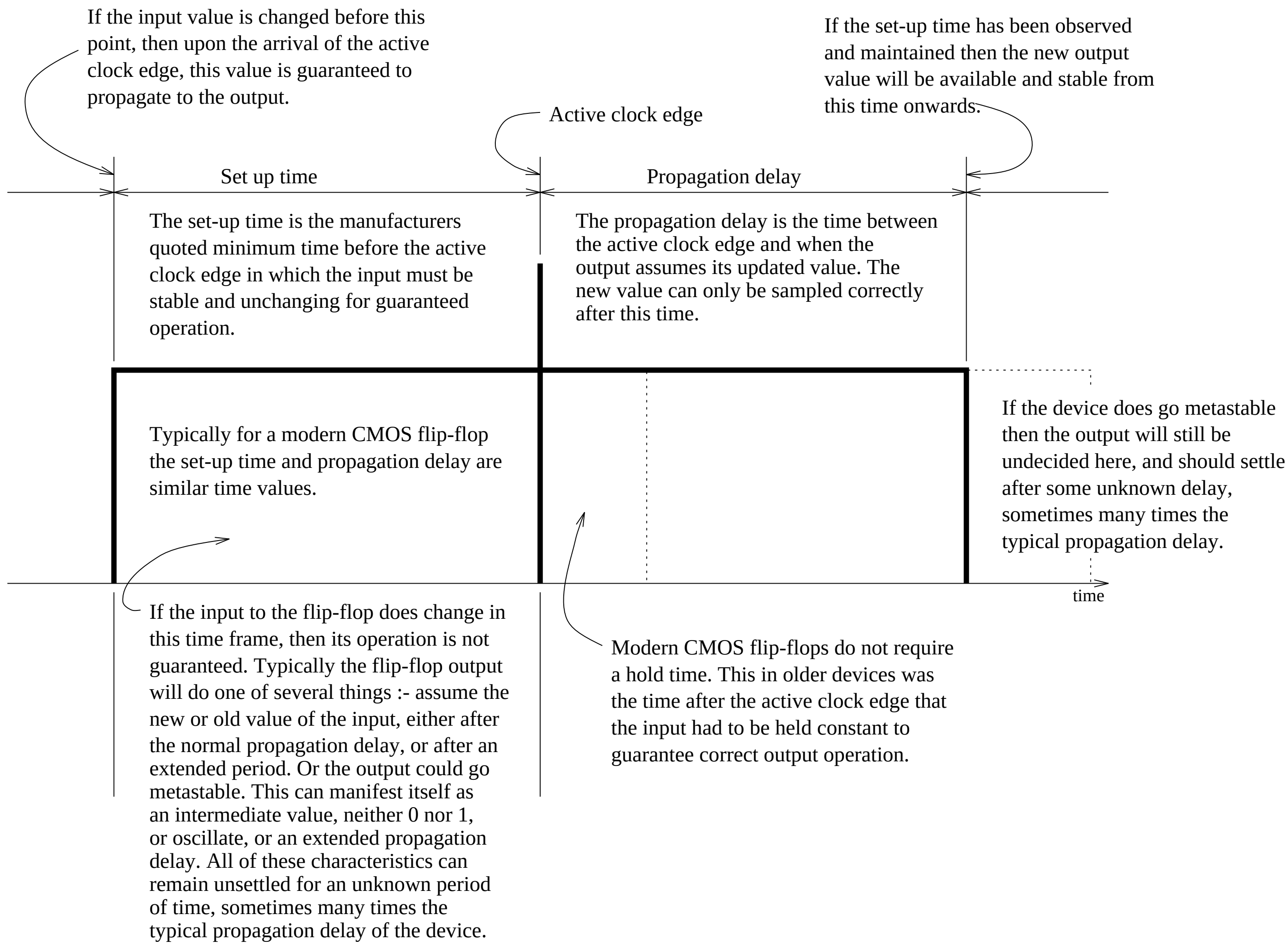


Figure 2.5: Set-up time (choose) and propagation delay (indicate) definition.

A latch has a timing characteristic as part of its specification, shown in Figure 2.5. This defines the set-up and hold times of the device and the transition delay associated with the output stage of the device. The set-up and hold times given in data books are the worst case values obtained by testing random samples of devices from several different production runs. The specified values being the worst case, means that even if these limits are broken the device may operate in a well behaved manner. Jackson [JA89] published a graph showing the relationship between latch input timing and output timing, see Figure 2.7. This graph shows that, depending on the allowed propagation delay of a system the input timing specification can be modified. There are only a very small number of situations where the device will actually go metastable, but no matter how small the probability, it is not zero. The output state space of the latch has two stable states and also an area where if the change energy in the device is not sufficient then the output may linger at this third state for an undefined period of time, see Figure 2.6 [Yar96]. ([Kin00] claims that in practice metastability will usually have settled within 5ns in digital logic fabricated using modern CMOS processes). In practice this third state lies in the region in the middle of the specified levels for logic levels '0' and '1'. Metastability can present itself in several different forms. These forms are shown in Figure 2.8. The forms are a mid way level between the defined logic levels '0' and '1' (b), output oscillations (c), and delayed switching (d). A normal change of state is shown in (a). All of these manifestations can last for an unknown time, but are usually resolved within the offending device due to circuit and thermal noise internal to the device, although [CW75] claims that any thermal noise trying to make a device settle is counteracted by noise in the opposite polarity and can therefore be omitted from any settling calculation. Simpson [Sim90b] calls (b) dither and (c) flicker. Oscillations or flicker cause the possible reading of a variable as a new-old-new sequence.

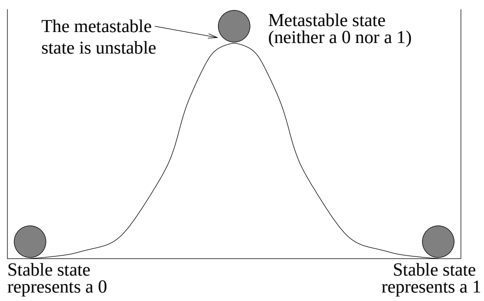


Figure 2.6: Metastability modelled with a hill and ball.

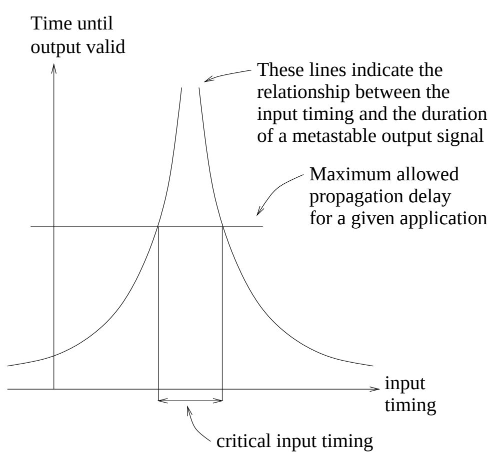


Figure 2.7: Comparison of latch critical input to output timing.

When metastability is experienced within circuits, system failures can occur. Such failures are hard to track down because it is difficult to ascertain if the failure was

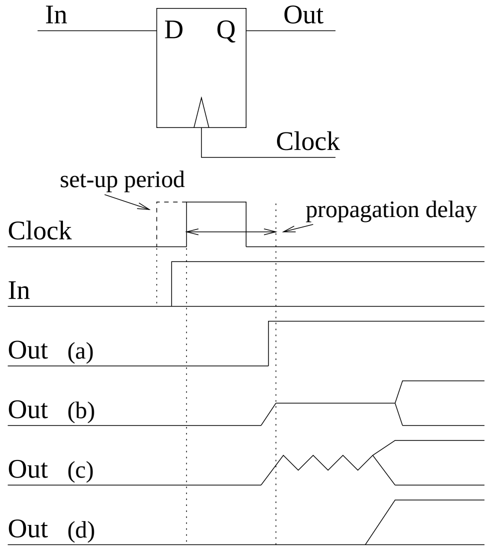


Figure 2.8: The different forms of metastability.

caused by metastability or something else. In the case of safety critical systems the effects of metastability can be dangerous. Metastability is an analog effect, and the effect is due to the analogue system having two stable states separated by an unstable region, which must be traversed when changing from one stable state to the other. When the devices output is in this unstable region for a longer period than for a normal transition between states the device is considered to be metastable. Analogue electronic systems themselves do not have metastability because they have no specified levels, just a continuous range between 0 volts and the supply rail (if the supply is $\pm 12V$, say, the range is $-12V$ to $+12V$).

Several authors have published methods and results trying to overcome the problems and effects of metastability. Some of these solutions are discussed here.

One proposed solution [KC86] was to use three latches in parallel in place of one latch. Each of the three latches was designed to have differing set-up times. It is suggested that because of three differing delays, not all devices will go metastable in any one cycle of operation. The result from the three latches is then combined in a voting arrangement to decide the new value. This would reduce the effect of metastability slightly, but it does not cure the problem. If during one run of the proposed solution one latch goes high, one latch goes low, and the third goes metastable, then the outcome of the voting is again relying on and waiting for a metastable device to settle. This outcome can occur with any solution using an odd number of devices. The solution could be modified to use an even number of devices, but this can be even worse. If no devices go metastable, but half of them go high and half go low, then there is no decision. Therefore we can state that using multiple devices does not guarantee a solution.

It has been argued [Sim90b] that the value obtained from a metastable bit variable is never wrong, it is either old or new, once it has settled. This can be accepted to be true for single bit variables, but not so with multi-bit variables. This still leaves the question of unknown and variable propagation delays from the metastable device.

CMOS latches fabricated using the present technology have two stable states which are used for the memory effect and also cause the metastable phenomenon. These devices consume very little power when in either of the stable states, and consume much more power during a change from one stable state to the other. One theoretical solution to the metastability problem is to have a device with only one stable state and make the other state non-stable and require permanent drive to be in the second state. In theory this might work, if it were not for the amount of power required to sustain many of these devices in the driven state. If however only one or two devices were required for interfacing purposes only then this extra power requirement would be manageable. Another solution is to use the power consumption as a metastable state indicator since the power consumption is much higher while the latch is metastable. Completion detection has been proposed before for asynchronous self-timed circuits [BG96, DDH94, IST90]. Each latch operating with an asynchronous input could have a built in power consumption indicator. This would then indicate when a latch was switching and the reader of the value would know that the value is not stable until the power consumption has reduced to its stable state value. This does mean that waiting has been included, and the worse case is that the reader could wait forever for the device to settle, although this is unlikely [Kin00] (Mitchell claims that because instructions are software, the delay between the instructions will "almost surely" allow metastability to settle [MG96]). It should also be noted that the completion detection circuitry can be a considerable hardware overhead in a large circuit. In [LC83] an extra output called 'M' is discussed which is active when the latch is in a metastable state. However the discussion is at a high level of abstraction only and no hardware implementation is proposed.

Metastability filters have also been proposed in differing forms. One form of

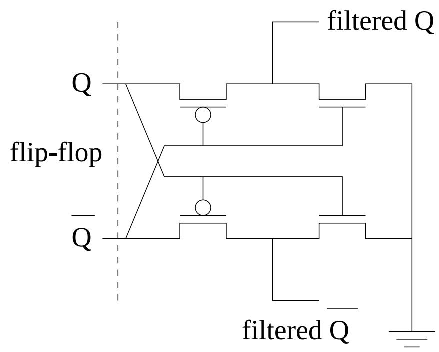


Figure 2.9: CMOS equivalent of Seitz's metastability filter.

metastability can cause the output signal to remain at a level half way between the two specified logic levels, i.e. 50%. If a filter is connected to the device output which has a threshold level of either 33% or 66%, then a 50% output value will be blocked from passing and will appear at the output as either a definite '1' or '0'. In its simplest form this filter can be a three input NOR gate, which has a threshold of 33%. One of its more complex forms was published by Seitz [Sei79], a CMOS variant is shown in Figure 2.9. These filters have the effect of removing one unknown quantity. A metastable device will typically have an unknown output level which will settle at an unknown time. The filter does not allow the unknown level output to pass to the output, so the metastable state is now just a deferred switching event (which can be delayed for an indeterminate time).

Numerous solutions have proposed passing multiple bits instead of just a single bit and then determining the result from a function of the bits sent. Another solution would be to send the bit using a Hamming code [Tan90], where the bit is sent and its parity is sent as another bit. Hamming codes use added parity bits to perform simple error correction or detection (two bits (including data) can only provide one bit error detection, three bits (including data) are needed for one bit error correction). In the case of metastability, there would be one data bit and one parity bit, preferably sent separately. If one were to go metastable then the parity result would not be correct. However it would be unknown which bit went metastable. It is unlikely that both bits would go metastable and the error result be cancelled out, but the probability is still not zero. Adding extra bits to try and improve the reliability is a form of redundancy: there is of course a hardware overhead to pay. Lamport claims that to implement an atomic bit two way communication is required [Lam86], taking the form of two bits owned by the writer and one bit owned by the reader.

Many ACM solutions, including some presented later in this thesis, do not acknowledge metastability, and analysis is performed with atomic assumptions. Several authors acknowledge the possibility of metastability but proceed to omit it from their analysis. In one paper metastability is acknowledged as a physical characteristic of the low level hardware devices, but then the author adds a footnote claiming that analyzing the effects of such a physical phenomenon was “too radical” for the journal in which the paper was published, and proceeded to omit it from any further analysis. Many of the solutions found have been analyzed using formal methods, including various logics. These solutions do not include metastable operation in their analysis, perhaps because the methods used cannot support this type of analysis. Simpson [Sim90b] is one author who acknowledges metastability and whose analysis partly includes it.

2.5 Petri nets

Petri nets are introduced here as they are the primary modelling and analysis tool used in this thesis.

Petri nets [Pet81b, Rei85] are a simple yet powerful means of modelling discrete event systems. Since their introduction by Carl Petri in the 1960s, they have been applied to many varied applications [Bib87]. A Petri net is a development of a directed graph.

Only classical Petri nets are used in this thesis because they are sufficient for this analysis problem. By the term ‘classical Petri nets’ the author refers to the fact that only four original core elements are used. These elements are ‘places’, ‘transitions’, ‘tokens’ and ‘arcs’. These are shown in Figure 2.10.

The basic rules for modelling with Petri nets lies in the function of each of the core elements. The ‘place’ is a data storage area or state indicator. The ‘transition’ is where work is done. The ‘token’ is used to indicate the presence of data or to indicate that a state is active. Finally the ‘arc’ is used to connect places and transitions together and it indicates the direction of data flow along each connection. Places are never connected directly to places and transitions are never connected directly to transitions. Places are connected to transitions by arcs and vice versa. Places connected to transitions via arcs with the direction towards the transition are called input places, whereas, if the arc is directed from the transition to the place then it is called an output place. See Figure 2.11.

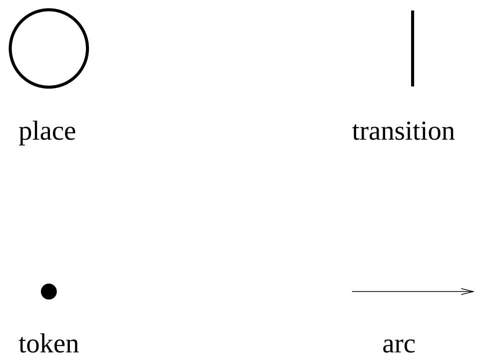


Figure 2.10: The four core Petri net elements.

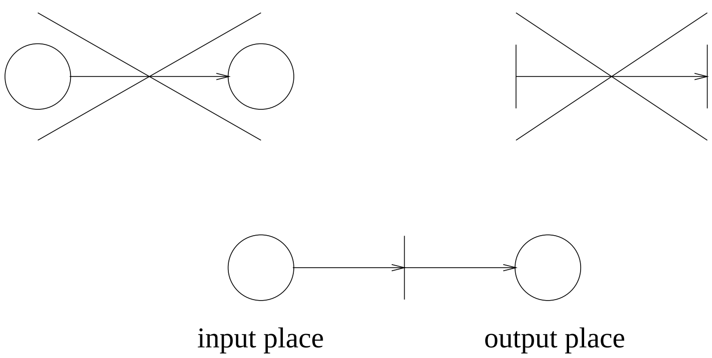


Figure 2.11: Petri net connection.

To indicate a state or the presence of data, a place must contain a token, this is known as being marked. For example if a place was labelled 'A=1' or 'running', this state is only true while the place is marked. For a transition to fire, each of its input places must be marked. When a transition fires it removes 1 token from each of its input places and puts 1 token in each of its output places. Several examples are shown in Figure 2.12.

Inhibitor arcs have been included in the Petri nets used in this thesis, see Section 4.2.10. One notational short cut has also been included called an OR arc (Section 4.2.9), which to the knowledge of the author has not been previously published.

The reason for choosing Petri nets over other available analysis methods such as finite state machines (FSM), binary decision diagrams (BDD), CSP (Communicating

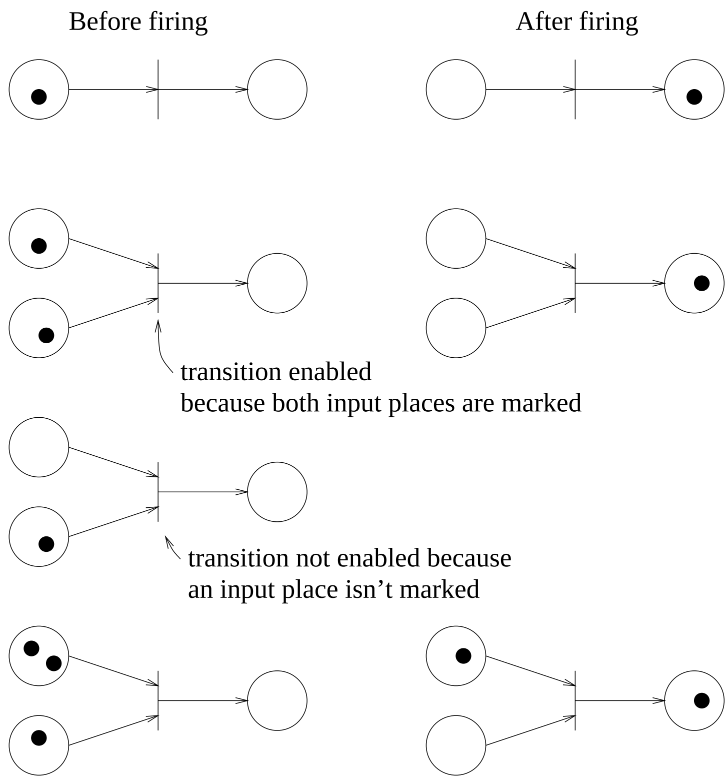


Figure 2.12: Examples of Petri net firings.

Sequential Processes) and CCS (Calculus of Communicating Systems), was that from Petri nets hardware can be directly synthesized [CKK+96, Cor]. Petri nets provide a formal syntax for modelling the ACMs being analyzed, (the models created were FSM in style), which is suitable for the purposes of this work. Petri nets also offer a simple modelling language which is comprehendable by a wide audience. CSP [Hoa78, Hoa85], does not allow for true asynchronous operation between processes, because any communication is synchronized by the "act" instruction. CCS [Mil80], is similarly not suitable because of synchronized communication.

2.5.1 A classical concurrent problem

Probably the best way to introduce the Petri net method is by way of a well known classical computer science problem, called the dining philosophers, introduced by Dijkstra [Dij68]. The fable clearly demonstrates how deadlock in concurrent processes can arise and also shows how Petri nets can be used to check for such a condition. The problem is as follows: there is any odd number, greater than or equal to three, of philosophers sitting around a table. Each philosopher is either meditating or eating and performs these actions alternately for an unlimited time. Between each philosopher and his neighbour is a single chopstick. For a philosopher to eat he must pick up two chopsticks, one chopstick from his left and one from his right. There are two modelling options here, according to whether the philosopher picks up one chopstick at a time or both together. Both models for a single philosopher are shown in Figure 2.13.

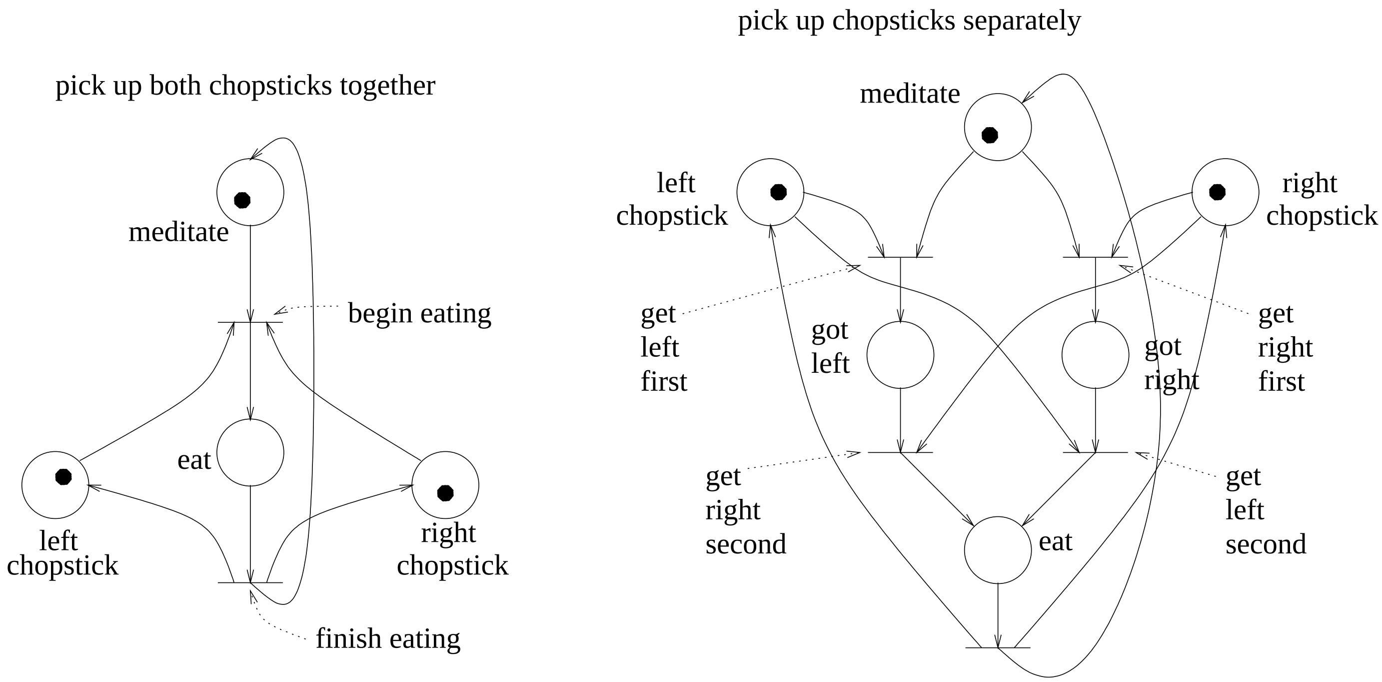


Figure 2.13: Two ways of modelling a single philosopher.

The full model for a three philosopher system is shown in Figure 2.14, with each philosopher trying to pick up both chopsticks together. It can be exercised by following the Petri net game and firing transitions according to which transitions are enabled, i.e. their input places are marked. If this is done, it can be confirmed that only one philosopher can eat at any time because when he has two chopsticks, the others cannot. This solution is not complete. A philosopher cannot eat forever, so eventually they will all get a chance to eat because deadlock is not represented. If however the philosophers in this model are changed to ones who pick up one chopstick at a time, deadlock may occur. (Deadlock is a state where the system cannot progress because no transitions are enabled, e.g. several processes have got some way through execution of a given function and along the way block the progress of the other processes). If each philosopher picks up the chopstick to his left, then the system cannot do anything else. It is stuck. No philosopher can pick up his second

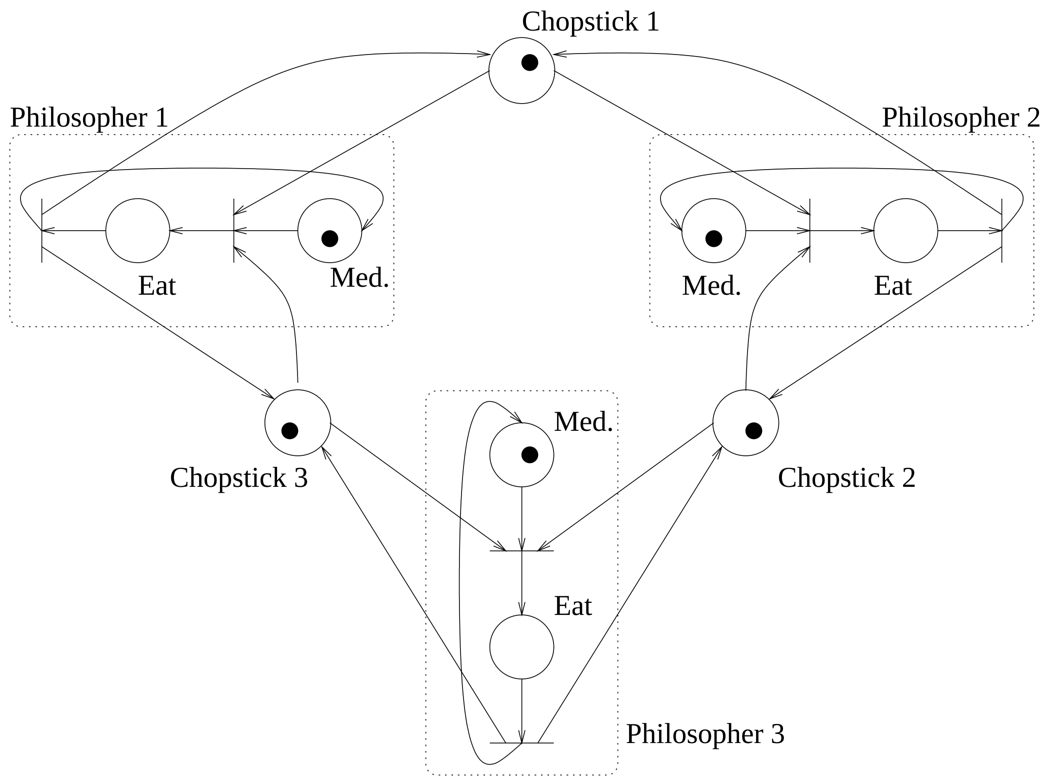


Figure 2.14: Three philosophers both picking up both chopsticks together.

chopstick. This is deadlock. The same condition can also arise if they all pick up their right chopstick. See Figure 2.15. Discovering whether a system may deadlock is an important part of the Petri net analysis.

2.5.2 Structural and behavioral analysis

There are two types of Petri Net (PN) analysis, structural and behavioral. Structural analysis is used to define properties of the model such as whether a particular transition is live, or the maximum number of tokens in certain areas of the net. Behavioral analysis allows investigation into whether a certain state or condition is reachable. There are many published techniques for PN analysis, most of which are not suitable for the analysis of the mechanisms being presented in this thesis. In the following sections several of these techniques are introduced with an explanation of whether they are suitable.

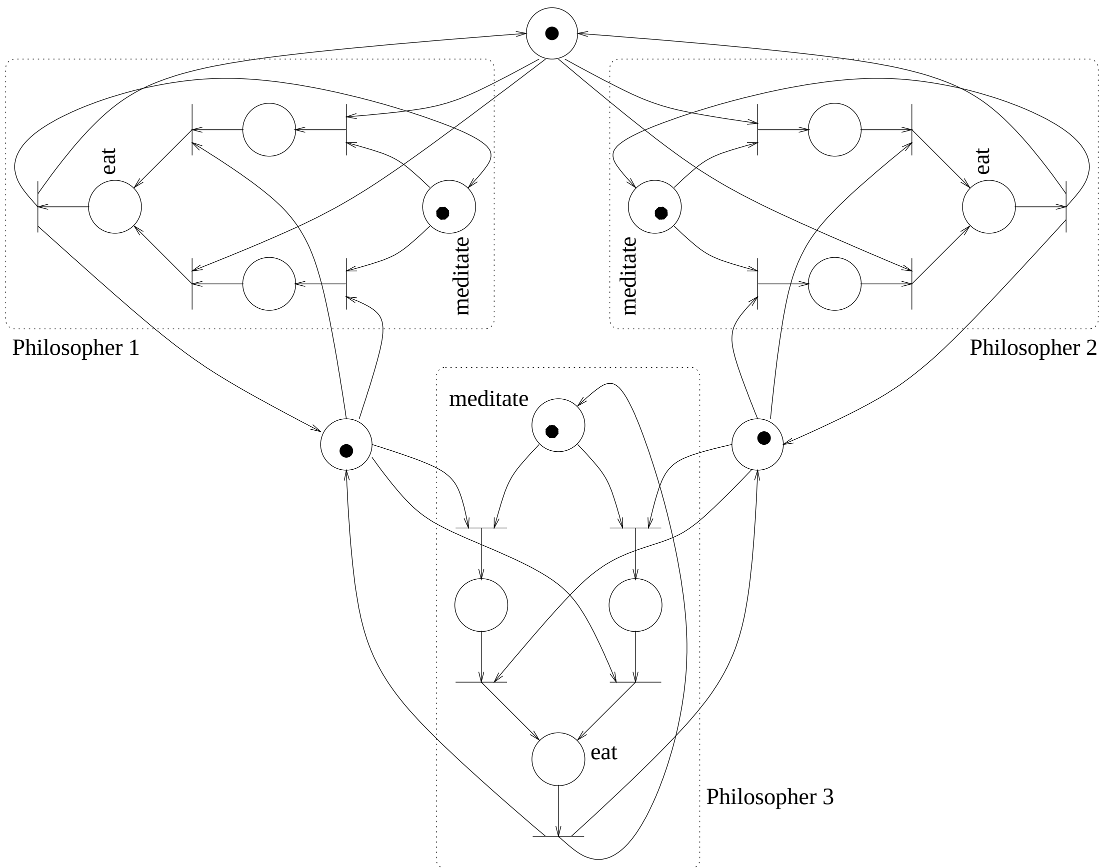


Figure 2.15: Three philosophers picking up chopsticks separately.

The incidence matrix

Petri-nets can be represented using matrices, most importantly the incidence matrix, which enables Petri-nets to be analyzed using linear programming techniques.

The incidence matrix is constructed by combining the two matrices D^- and D^+ which represent the input and output functions of the Petri-net. Each matrix has m rows (one for each place) by n columns (one for each transition)[2]. The D^- input function matrix has 1's corresponding to arcs pointing from places to transitions, i.e. the transition inputs. The D^+ output function matrix has 1's corresponding to arcs pointing from transitions to places. Using the Petri-net in Figure 2.16, the input and

[2]There are many texts using these matrices but because of the lack of a standard convention, some interchange row and column definitions, and so use the transpose of these matrices in developing the theory and results. The convention used in this thesis represents markings as column vectors.

output function matrices are

$$D^{-} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad D^{+} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

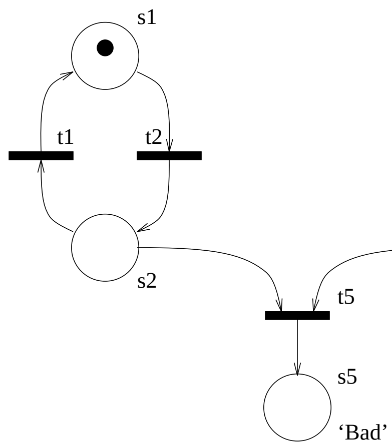


Figure 2.16: An example Petri-net.

The incidence matrix D is constructed using

$$D = -D^{-} + D^{+}$$

which for the example is

$$-\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Using the incidence matrix D, a sequence of Petri-net markings can be calculated using

$$D.x + m_0 = m_f$$

where x is a transition firing vector (the number of times each transition is fired), m_0 is the initial marking of the Petri net and m_f is the final marking. An example for this net gives

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

This example shows that $s5$ can be marked, but notice that there is no firing order for the transitions. Re-arranging the equation to

$$D.x = m_f - m_0$$

shows that given a known m_f, if there exists a solution for x, then m_f is reachable from m_0. Taking this a stage further, in the communication mechanism models presented in this thesis, the marking m_f may not be completely known. The result required in this case is for any solution where the 'bad' place (Section 4.2.8) in m_f is marked. This gives the problem

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

where there are two sets of unknowns. This can be solved using the linear programming technique called the 'Simplex' method. However there is a problem. In the main models, self-loops are used to allow variables to be referenced without being altered. A self-loop has two arcs between a place and transition, and the arcs point in opposite directions, see Figure 2.17. The rightmost net in Figure 2.17 is the more common representation for a self-loop using a single double headed arc. These self-loops cancel out during the construction of the incidence matrix D. The result of this cancelling is that invalid markings can be shown as reachable.

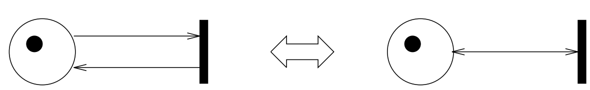


Figure 2.17: Self-loops in Petri-nets.

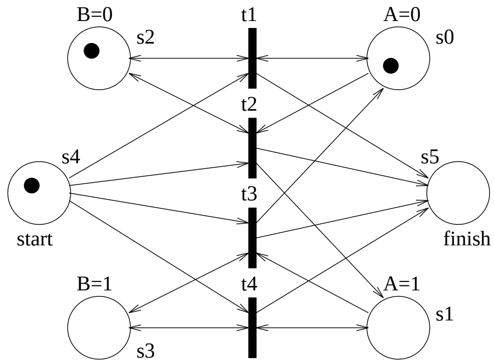


Figure 2.18: A variable assignment statement modelled in Petri-nets.

A variable assignment net which is commonly used in the main models is used here to highlight the problem. The variable assignment net for $A := B$ is shown in Figure 2.18. The incidence matrix for this net is

$$D = \begin{bmatrix} 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

It can be seen that the six self-loops in the net have cancelled themselves out in the incidence matrix. Two of these self-loops between the transitions and $s0$ (A=0) and $s1$ (A=1) have no effect when cancelled from the incidence matrix, but the other four self-loops, between the places $s2$ (B=0), $s3$ (B=1) and the transitions have a major effect. The effect of these second four cancellations means that the value of B is never referenced, therefore $A :=?$. For example

$$\begin{bmatrix} 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

indicates that it is possible to fire transition $t3$, giving a result where $A = 1$ and $B = 0$, which is clearly wrong and shows that $B = 0$ was never referenced.

Therefore this technique cannot be used safely in the analysis of the communication mechanisms.

Invariant analysis

The Chambers science dictionary definition of *invariant* is "Any characteristic of a system which is unchanged by transformations of the type under consideration". In Petri-net terms this indicates a structural property which does not change during any firing of the net. There are two types of invariants associated with Petri-nets; S-invariants[3]/P-invariants (place invariants) and T-invariants (transition invariants). Invariants help in the analysis of the liveness (the absence of deadlock) properties of a net, and also in investigating and quantifying operational facts about a model.

A place-invariant is a vector which indicates a subset of the places in a Petri-net which always contain the same sum of tokens. An example application of place-invariants would be in the analysis of a FIFO buffer with n registers. In the Petri-net model there would probably be two places representing *registers full* and *registers empty*, which would be used to stop a full buffer being written to, and an empty buffer from being read. The sum of the tokens in these two places must always equal n, otherwise there is a error in the model. The place-invariant representing a correct implementation would be a vector with its number of elements equal to the number of places in the model, and the two places representing *registers full* and *registers empty* would both be marked with n. This confirms that the firing of any transition in the model does not affect the sum of tokens in these places, and therefore that this aspect of the buffer is functionally correct, assuming it has been modelled correctly.

The communication mechanisms presented in this thesis are cyclic in nature and use binary control variables. Place-invariants can be used to check that a binary variable always has only one token, or that a process control loop also always has only one token. The latter proves that the process does not disable itself by removing the token from the control loop. These properties are structural in nature, and without the use of place-invariants a proof about behavioral properties would require an exhaustive state space search, followed by further analysis.

[3] S representing Stellen - German for 'place'

The main requirements of these mechanisms are to maintain data coherence (Section 4.4.2), data freshness (Section 4.4.3) and data sequencing (Section 4.4.4) as described in Chapter 4. All three of these involve testing to see if a place labelled 'bad' ever receives a token, i.e. is marked. (A place labelled 'bad' is used to indicate that an operating property has not been maintained) All these requirements are behavioral in nature because they are dependent on the reachable states of the model during execution.

Place-invariants only give a necessary condition for reachability, not a sufficient one. This means that if a place can be shown never to be marked with invariants, then that place will never be marked in any of the reachable states. The opposite condition however does not hold. A place-invariant may include a place as being marked, but in practice this place may never become marked. This can be shown using a slightly modified version of the Petri-net example introduced in Section 2.5.2, see Figure 2.19. This Petri-net implements the 'bad' place used on each slot of the four-slot model, and is used to check for overlapped reading and writing of the same slot, i.e. checking data coherence. However this Petri-net cannot ever place a token in the 'bad' place ($s5$) because a token removed earlier by $t2$ is required. The place-invariants of this Petri-net are found to be

$$(1, 1, 0, 0, 1, 0)$$

and

$$(0, 0, 1, 1, 1, 0)$$

These show that the token sum remains always at 1 in places $s0$, $s1$ and $s5$, also $s3$, $s4$ and $s5$, therefore both invariants show $s5$ as possibly being reachable. These place-invariants remain unchanged when place $s6$ is removed (see Figure 2.16) from the net thus allowing $s5$ to be marked. Each slot in the four-slot model produces similar place-invariants, therefore this technique does not indicate the non-reachability result required. This is because the behavior of the net is not considered.

A transition-invariant is a vector which indicates how often, starting from some marking, each transition has to fire to reproduce the same marking. Using the example described above (Figure 2.19), it is found that there is only one transition-invariant,

$$(0, 0, 1, 1, 0)$$

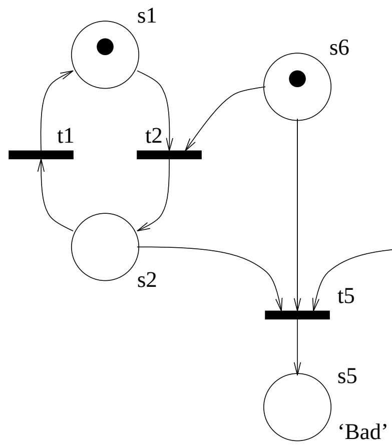


Figure 2.19: Petri-net example with an extra place.

This invariant indicates that from some marking m, if we fire transitions $t3$ and $t4$ (order not specified by the invariant) we can obtain the original marking again. This however does not take into account the fact that one of the transitions could be disabled. This is confirmed if place $s6$ is removed from the example net. With $s6$ removed, see Figure 2.16, the place $s5$ can be marked, removing the tokens from both loops. However the transition-invariant remains the same, and another transition-invariant is found for the other loop, $(t1, t2)$.

Reachability searching

A reachable state is defined as a state which is obtained by firing a valid sequence of transitions, starting from an initial marking. During an exhaustive reachability search, a reachability tree is constructed which has the initial marking as the tree root, and the branches from this root are the transitions which are enabled by the initial marking. The leaves on the tree are the reachable states of the Petri-net. Reachability searching is invariance analysis of the state space [Hol91]. As stated earlier, the models used for the communication mechanisms are cyclic in nature, which produce infinite reachability trees. An example of this is the reachability tree for the Petri-net in Figure 2.16, which is shown in Figure 2.20.

The solution to this problem is to maintain a visited states list. When a new state is generated it is checked against this list, if it is already on this list then the state is discarded, if it is not on the list it is added to a list of states still to be investigated. This reduces the reachability tree to its minimum, and every valid state is covered.

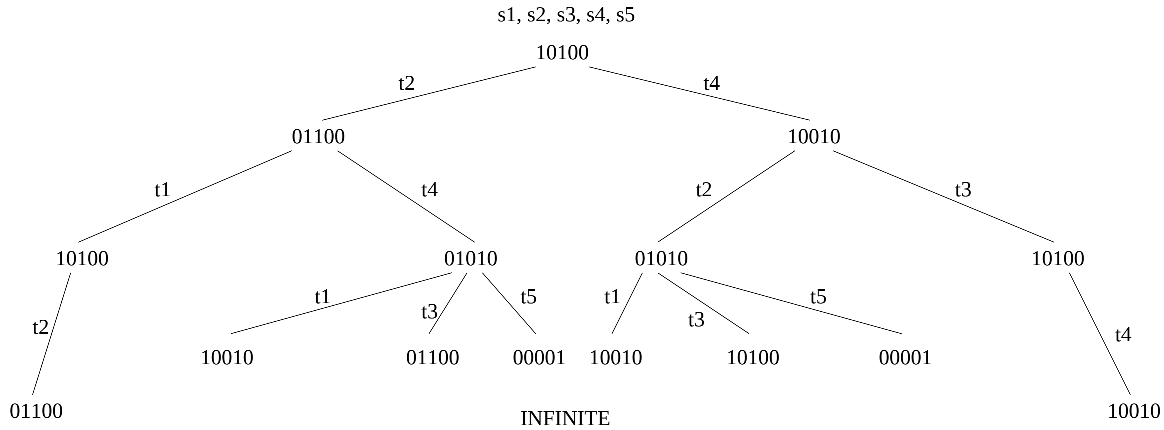


Figure 2.20: Part of the infinite reachability tree for the Petri-net in Figure 2.16.

The minimum reachability tree for the net in Figure 2.16 is shown in Figure 2.21.

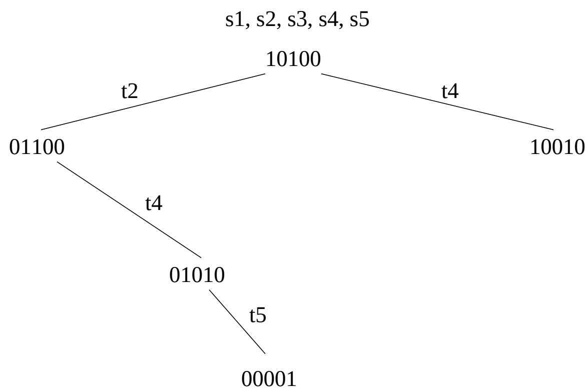


Figure 2.21: The minimum reachability tree for the Petri-net in Figure 2.16.

Breadth first searching was chosen over depth first for this application, because when a ‘bad’ marking is found, the route of transitions fired to get to this marking is stored. This ‘bad’ marking route can then be investigated showing how the mechanism being modelled has failed a particular property. By using breadth first searching any ‘bad’ marking routes given are the shortest possible.

This reachability analysis is the main method used in the analysis of the mechanisms in this thesis. At the time of the initial studies for this work a search was made for suitable tools, and unfortunately it was found that all the available tools were unable to handle the cyclic models because they tried to produce infinite reachability trees. A program was written to perform the reachability analysis with a visited list (see Section 4.3).

Chapter 3

Modelling of latches shared between concurrent processes

3.1 Introduction

In this chapter the modelling requirements of latches shared between multiple processes is considered. This chapter concentrates on single one bit latches which are written by one process and read by another. This provides the simplest form of persistent one way communication, which is a very limited form of communication between processes, but as later chapters will demonstrate, it is the building block upon which true asynchronous communications mechanisms are implemented. With multiple processes three execution possibilities are investigated, and the unavoidable possibility of a metastable state is also considered.

No matter at what abstraction level a function is implemented conceptually, eventually it will be physically executed at the hardware level, normally involving the use of latches for the purpose of variable storage. At all abstraction levels the designer will use some form of assignment operator for storage and at the higher levels will probably assume an atomic characteristic for the operation. However, looking at the hardware level, whether dedicated or micro-processor based, these operations are not atomic, being made up of several different stages to perform the required operation. It is shown here under which circumstances assuming atomicity is inaccurate, and it is shown where assumptions can be safely made to avoid unnecessary detail by breaking down every operation into its elemental parts. ("Results are only as good as the model being analyzed" [Sei71]). These assumptions help to reduce the size and complexity of the analysis.

3.1.1 Processes and processors

A *process* is defined here as the sequential execution of defined instructions, collectively performing a desired task. In many cases the instructions of a process form an iteratively executing loop (which is unlike non real-time software for which termination in a finite time is usually a requirement of correctness). This *process* is mapped onto some fundamental (lower) level device which physically executes the defined instructions - the *processor*. The processors assumed throughout this work are synchronous by design, being operated either with a common synchronizing clock, or individual unsynchronized clocks.

A single process must execute on a single processor element, but a single processor element may execute one or more processes, either concurrently (e.g. in some interleaved, time-multiplexed way) or sequentially. (A single processor element is typically a single processor, but may be several processors operating together - this is referred to here as a processor.) The execution of multiple-processes provides a choice of three forms of operation, these being sequential, synchronized concurrent and asynchronous concurrent.

Sequential operation means the execution of one or more processes on a single processor, whether the processes are executed one after another or concurrently with their instructions interleaved. With multiple processor systems, the number of active processor elements should always be less than or equal to the number of processes, i.e. a single process will not normally execute on more than one processor element.

Synchronized concurrent operation means the execution of processes on separate processors, when the processors share a common synchronizing clock. In this case multiple instructions are being executed simultaneously. This means that instructions can be interleaved but that they all execute together synchronized to the common clock.

Asynchronous concurrent operation means the execution of processes on separate processors, but this time each has its own independent clock running at its own rate. In this case, as well, multiple instructions are being executed but all possible interleaving combinations have to be considered. This type of operation is called 'globally asynchronous, locally synchronous'.

The term 'executing processor' does not necessarily dictate a microprocessor implementation. Dedicated hardware implementation or microprogrammed state machine are also possible. If an operator, '$a := x$' for example, were being executed on a microprocessor, then '$a := x$' would be a line of program code, where x is set somewhere earlier in the program. In dedicated hardware '$a := x$' would be implemented using latches, the number of which would depend on the data type and range of a and x, see Figure 3.1. The latch(es) holding the value of x would be clocked at one clock instant, then the latch(es) to hold a's value would be clocked at a later time.

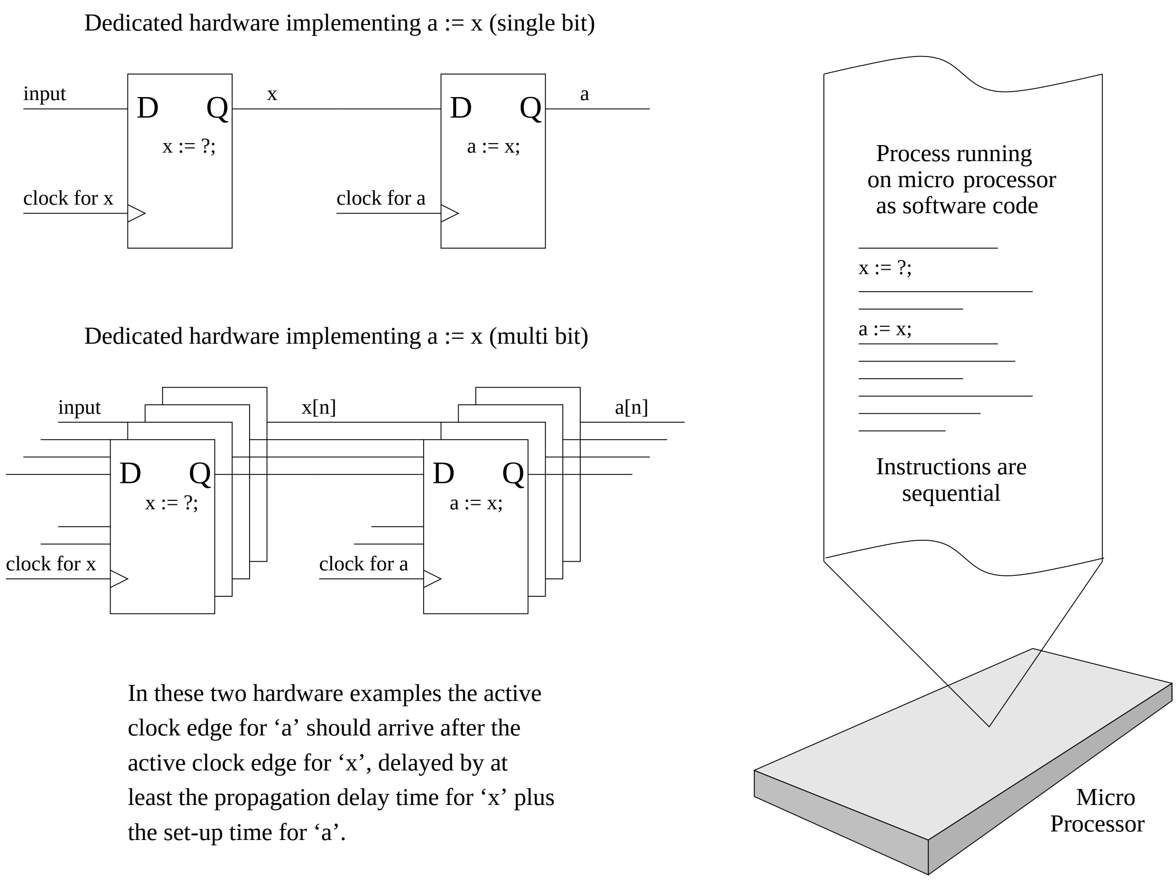


Figure 3.1: Operator implementation in hardware and software.

3.1.2 Latches

Regardless of data type, variables are represented using binary data at the hardware level, and this is typically stored using registers constructed from latches, usually of the 'D' type. Edge triggered 'D' type latches have three timing requirements for correct operation based around the active clocking edge of the device[1].

There is a period of time before the active clock edge called the set-up time, during which the input signal to the latch should be maintained constant. After the active clock edge there is a period called the hold time, during which the constant input signal should continue to be maintained (typically - depending on design - modern CMOS latches do not require a hold time). Also, after the active clock edge there is a period called the propagation delay, which is the time taken for any new output value to propagate through the device and appear at the output, see Figure 2.5. To guarantee correct device operation the manufacturer's specifications of the set-up and hold times must be complied with. These specifications correspond to worst case values obtained through testing random samples of latches from several different production runs. Failure to uphold these worst case values does not automatically mean operational failure, but operation is unpredictable. For a discussion of metastability see Section 2.4. It should be noted that at power-up, a latch may settle into either of its stable states, and not necessarily '0'.

Simpson [Sim90b], points out that when single bit latches sample asynchronous inputs this results in the latch obtaining the input's old or new value, never a wrong value (after allowing any metastable state to settle). This is based on the assumption that the latch samples a value on its input either before, during or after the input has changed. If the input is changing from 0 to 1, then sampling before the change will result in the old value 0 being obtained. If sampling is after the change it gets the new value 1. If it samples during the change, there is a chance that the latch will go into a metastable state. The metastable state will eventually settle into one of the stable states, 0 or 1 and therefore to the old or the new value.

[1] Some latches can be level sensitive rather than edge sensitive.

3.1.3 Microprocessors

The term microprocessor encompasses many different devices, however they all have defined instruction sets, which can be used to construct software programs. Microprocessors are usually operated synchronized to a fixed frequency clock signal. Each individual instruction, at the machine code level, has to be 'fetched' and placed in the decode unit of the microprocessor, the instruction is decoded, and the relevant operation is performed in the arithmetic and logical unit (ALU). Each of these steps although usually performed within one external clock cycle, has several distinct internal steps each taking time. Take for example the fetch part of the cycle. The instruction pointer points to the the location of the next instruction, this instruction then has to be retrieved from memory, and stored in the instruction register of the decode unit, this alone is several simpler internal steps that are sequenced using an internal multi-phase clock signal. It is found by looking at the instruction sheet for a typical CISC microprocessor that many instructions take greater than one clock cycle to execute. (RISC microprocessors depend on same length instructions).

This lowest level of program is written in machine code (a defined instruction set which is microprocessor specific). Programs written in machine code by human operators are usually more efficient in terms of size and speed, compared to the automated compilation of high level languages. However programming in machine code is usually the most expensive in terms of program construction time, and also the most error prone. Most programs today are written using high level languages, which are compiled. The compiling process takes each high level instruction and translates it into the lower level machine code instructions. The advantage of using high level languages is their ability to describe complex problems simply, rather than getting bogged down writing low level machine instructions. High level language instructions are not performed as atomic actions, but are made up of simpler lower level stages. Even at the machine code level, simple instructions are not atomic. The instruction '$a := b;$' may be implemented in a microprocessor as '$acc := b; a := acc;$'

There is one advantage with this type of operation if external asynchronous hardware is being controlled. If each high level instruction typically takes several clock cycles to complete, then the probability of a metastable variable settling to one of the stable states before the next instruction increases.

3.1.4 Atomicity

From the description of a latch and the microprocessor it can be seen that the view of variables changing atomically is a high level simplification and not what is physically happening. When executing a process on a single processor machine, this simplification is safe because of the sequential nature that is dictated, see Section 3.6.1. If two or more processes are time shared on a single processor there may be a need to share data (variables) between them. This, if run on a single processor, can safely be modelled using the atomic approach, remembering instruction interleaving between the processes.

However when more processing power is required processes may be scheduled to run on multi-processor systems. Now the simplification of shared variables changing atomically is inaccurate, and the model needs to include the more fundamental steps mentioned above (set-up and propagation). If the processors are run asynchronously, with their own local clocks, then the communication is asynchronous, which inherently allows operation of the latches in their undefined timing regions due to the possible violation of the set-up and hold times, and therefore makes the possibility of metastable operation inevitable.

3.1.5 Registers

Registers are built up from multiple latches and are capable of holding variables with a range greater than two. Lamport [Lam86] defines three types of register, the safe register, the regular register and the atomic register. A *safe* register is the weakest possibility: it is assumed that a read not concurrent with a write will obtain the correct value. No assumption is made about the value obtained if a read overlaps a write, except that it must obtain one of the possible values of the register. Therefore if a safe register is updated during a change from 1 to 2, it may obtain either the old value 1, the new value 2, or any value which is not 1 or 2, but which can be stored in the register.

A regular register may only obtain either the old value or the new value. In our example above this would mean a regular register would only obtain 1 or 2.

An atomic register obtains its values in some definite order, i.e. there is a way of ordering the reads and writes so that no overlapping was apparent.

If a basic register made from multiple latches is used for communication between two asynchronous processes where one writes and the other reads, then the register would be only safe. This is because the multiple bits of the register may be written at slightly different times, and the reader may sample at a particular moment and catch the changing value. When a register has a new value written into it, maybe not all the bits are changed, therefore some bits from the old value are sampled and some from the new value are sampled. This is especially the case if sampling is near to the instant when the change occurs because latch transition delays are commonly different for rising and falling values.

It will be shown in the next chapter how atomic registers can be constructed from this basic safe register, but this requires multiple registers, and some form of control logic, unless waiting or mutual exclusion is allowed. The rest of this chapter concentrates on single one-bit latches, which are used to make up the registers, and the control logic in atomic registers.

3.1.6 Wait-freedom and mutual exclusion

Wait-freedom is defined as the ability to allow multiple processes immediate access to a shared resource at any time, even if concurrent accesses are performed. (A resource is something connected to the system, e.g. a printer on a network). Mutual exclusion only allows multiple processes exclusive access to a specified shared resource. If mutual exclusion is implemented as an access characteristic, then one process may be forced to wait while the other accesses the resource. In many applications this waiting is acceptable, however in real-time applications this may cause unwanted timing interference between processes. If the application is safety critical and real-time, then waiting may not be acceptable because if a process stalls during its access to the resource, other processes may be frozen in an endless wait loop. A possible solution to this problem is to incorporate a time-out period to prevent lock out.

With wait-freedom a single multi-bit register can only maintain 'safe register' operation, but with mutual exclusion 'regular register' accesses are possible. If however wait-freedom and regular accesses are required together then multiple multi-bit registers are required with some control/steering logic. As will be shown in the next chapter, such mechanisms are well documented, but many solutions to the problem make restrictive assumptions. With a correct implementation of such a mechanism, mutual exclusion is inherent in the solution, but it does not cause process waiting because there is a choice among multiple registers.

There are certain characteristics and side effects related to such mechanisms. The first of these is that the communication they provide has to be persistent, i.e. they hold a copy of the data being communicated, they are not transient in that the mechanism should not return to its original state when the data has finished being written. The second is that these mechanisms provide a reference data type which can be written to or read from at any time, therefore data duplication from multiple reads between two writes, and data loss from multiple writes between two reads are possible. This is acceptable in some situations but not all.

3.2 Methods used for determining the minimum accurate model

To do a state space analysis the smallest model possible should be used in order to keep the state space as small as possible. If the size of the final state space is not of importance then all shared variable accesses can be modelled using two stage models. However, any extra stage in a model has the potential to double the state space size, and therefore any reduction possible by acceptable simplification of the model is beneficial.

Interleaving graphs and reachable-state analysis will be used to help visualize any states omitted by the minimization process, and these are backed up by time line visualization for the benefit of clarity. The need for two methods will become apparent later, when it is shown that each method can fail to illustrate important properties.

The first method is an interleaving graph. This has the initial state drawn at the top left point of the graph, and each process executes in a different dimension. In the cases discussed here there are only two processes: the two dimensions can therefore be represented as horizontal and vertical. An example of this type of graph is shown in Figure 3.3. Each horizontal and vertical arrow represents a Petri net transition at firing. The diagonal line represents the firing if both could be fired together. The final output states are drawn at the bottom right of the graph. If the output state for the diagonal line is different from a route reachable using the horizontal and vertical lines then analysis of the model being used may omit states.

The two graphs shown in Figure 3.3, show firings taking one or two arrows per dimension. This is elaborated in a later section on one and two stage modelling.

The second method is simply a reachable state analysis. All the possible states are drawn in a circle, and then each state is taken in turn, working out what the next state would be if process one fired, process two fired or both processes fire together. Some examples of this reachable state graph are shown in Figure 3.17. The plain lines show the normal firings, and the dashed lines indicate where metastable states could settle - this will be explained in Section 3.3 on two and three level modelling. In the completed state graph if any state is reachable only via an arc representing a concurrent firing, and is not reachable via an arc representing a sequential firing, then a two stage model is required.

The final method used here is the parallel time line representation. This is used here as a graphical indication of the actual instruction interleaving in the time domain. An example of parallel time lines is shown in Figure 3.10. This gives the designer an insight into the physical reality rather than a state space view during the analysis, and is included here to increase clarity. The left side of the time lines indicates the physical interleaving, while the right side indicates the minimum model with its number of stages. Note that the right side is representing the single transition firing at any one time, therefore the representation cannot have any lines in parallel.

3.3 Two or three level modelling

Working in the digital domain two signal levels, '0' (off) and '1' (on), are used to represent data, known as binary. This is over simplified because intermediate levels must occur (other states) at the transitions between the two, e.g. an instantaneous jump is not physically possible.

If the shared variable being modelled is in a process being executed on a single processor (machine) or on a synchronous multiple processor arrangement then these two levels are adequate because values will only be sampled at regular clock-controlled intervals, and assuming the system is well designed then the values being sampled will have settled. There are established exceptions to two level modelling, for example multi-value logic packages in VHDL model extra states such as 'X' for unknown and 'Z' for high impedance [LSU89] (this is labelled as MVL4 (Multi-Value Logic) which uses 0, 1, X and Z. The IEEE standard is MVL9 which has nine levels). However, these packages perform time domain simulation, and not state space analysis.

For modelling shared variables in processes running on an asynchronous multi-processor arrangement, the two levels of '0' and '1' alone are inadequate. Physical implementation requires the use of latches, and using them asynchronously means they may not comply with the defined set-up and hold times, and therefore metastable operation is possible. In the two level models, metastability cannot be modelled. For more details on this metastability and its different manifestations see Section 2.4.

Here the case of metastability is generalized into the state 'M', and thus shared variables can be modelled with a three level model using '0', 'M' and '1'. The output net of a shared variable is shown in Figure 3.2. The transitions from 'M' to '0' and '1' allow the variable to settle from the metastable state to either value at some undefined time, just as a metastable state would settle in a real device.

However, two level models have been used [Sim92a] to analyze an asynchronous communication mechanism. This was justified by claiming that practical measures [KW76] can be used to make this hazard (metastability) vanishingly small (i.e. commensurate with other unavoidable failure modes). Two level analysis in this case is investigated further to show whether possible inaccuracies can arise.

3.4 One, two or three stage modelling

The single stage model is adequate for modelling variables in single process or single processor synchronous systems (see section 3.6.1). However when multiple processor

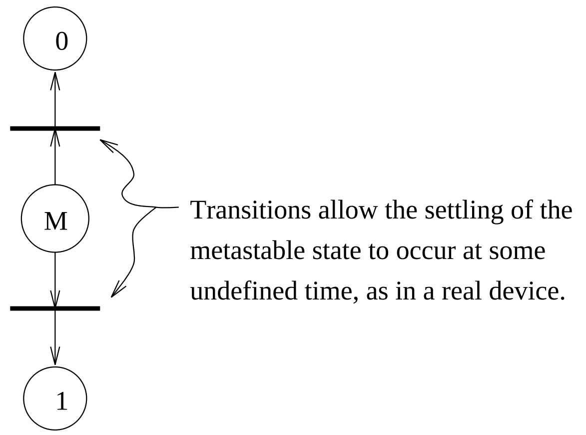


Figure 3.2: Three level Petri net model of the asynchronous shared variable output.

concurrency is introduced into the system, in some shared variable cases single stage modelling is inaccurate.

Consider two level models, where it is found that two stage models are needed in some cases, (Figure 3.3). The graph used here indicates whether an extra stage is required in a model. It can be seen in Figure 3.3 that the end point of the diagonal line where processes 1 & 2 fire together, cannot be reached by firing the processes separately. This indicates that single stage modelling is inaccurate here, and one of the equality operators needs to be modelled using a two stage model. This two stage model gives us a sequential equivalent to the concurrent operation. This is necessary here as Petri nets can fire only one transition at any one time. (There is one solution to this problem - the model incorporates transitions representing the two instructions executing at the same instant). Lamport [LS89] gives another complementary explanation of this behaviour. The diagonal line indicates that in the hardware both are firing at exactly the same instant: if one fires slightly before or after the other, this is equivalent to them firing singly one before or after the other - therefore in any graph there should only be one diagonal line (multiple firing) in each section. The single and two stage Petri net models are shown for two level systems in Figure 3.4. As there is no way of representing a metastable state even if it occurs in the two level model then there is no need to test for its possible initiating conditions, and therefore the one and two stage models cover all modelling requirements.

Three level models can represent a metastable state, so the minimum model is the two stage model. Because there is a possibility of going into a metastable state if the set-up time is violated, it is necessary to monitor the model inputs twice, at

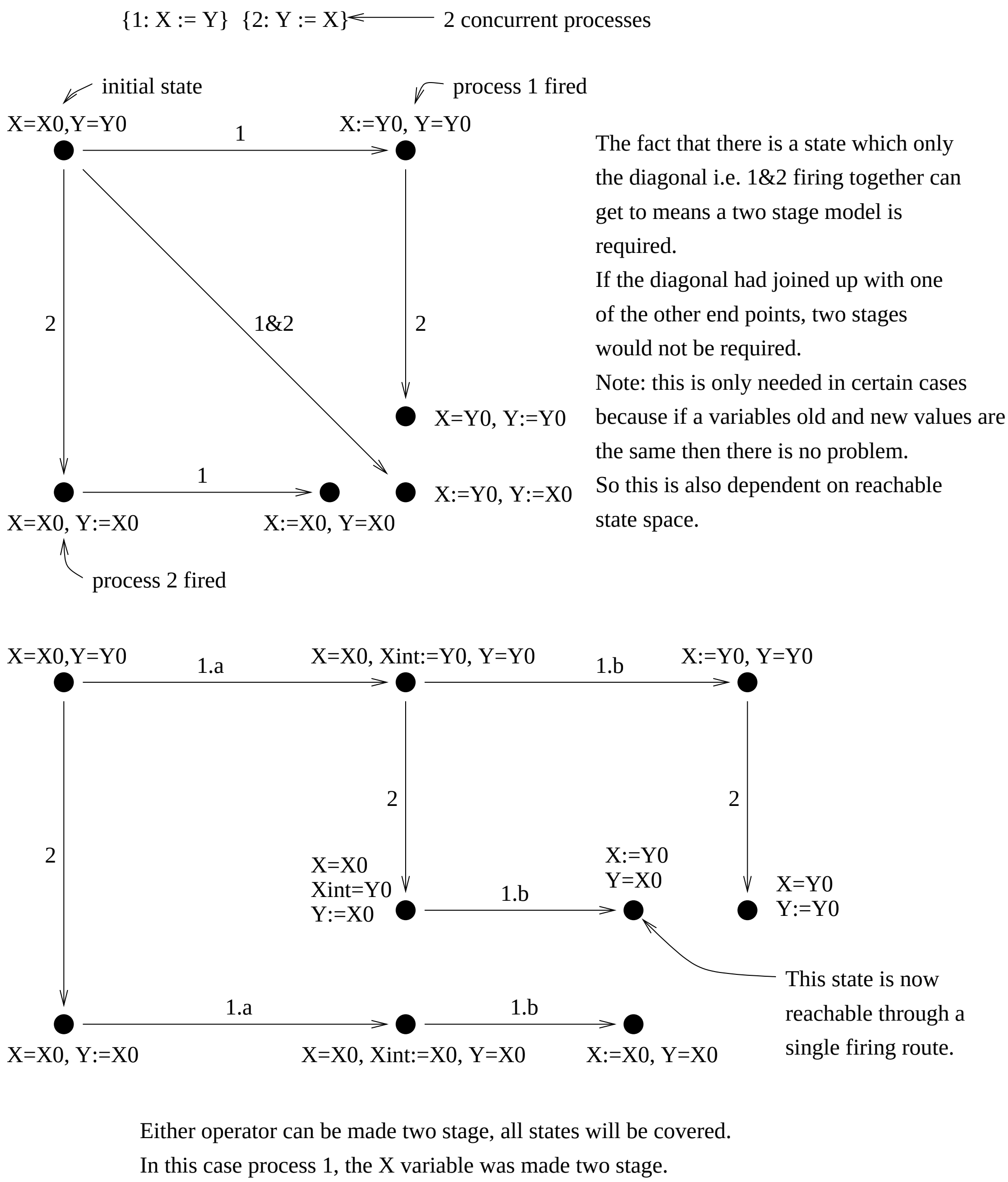


Figure 3.3: Illustration of need for two-stage modelling for two-level asynchronous systems.

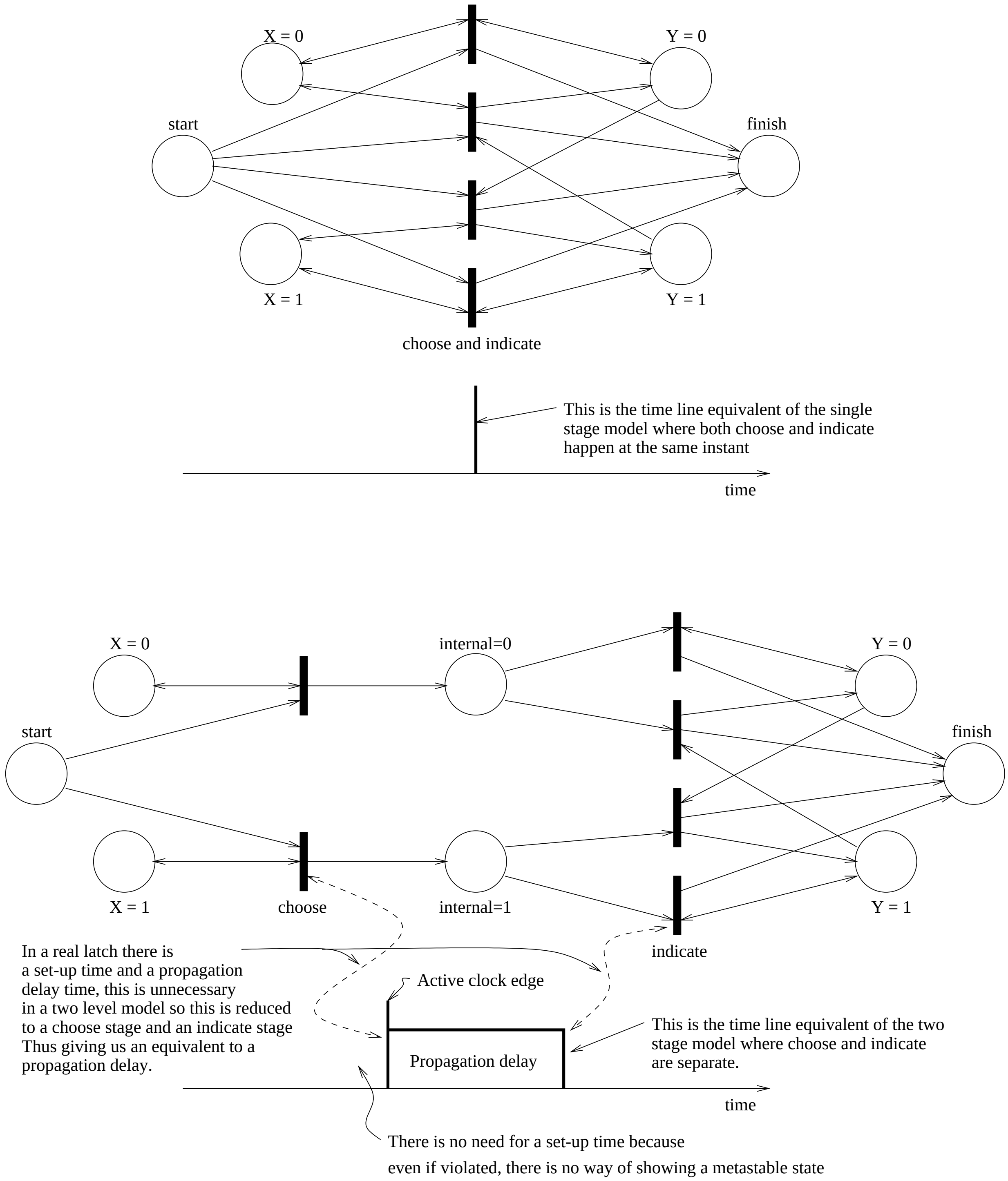


Figure 3.4: Petri net models showing single and two stage models for two level outputs.

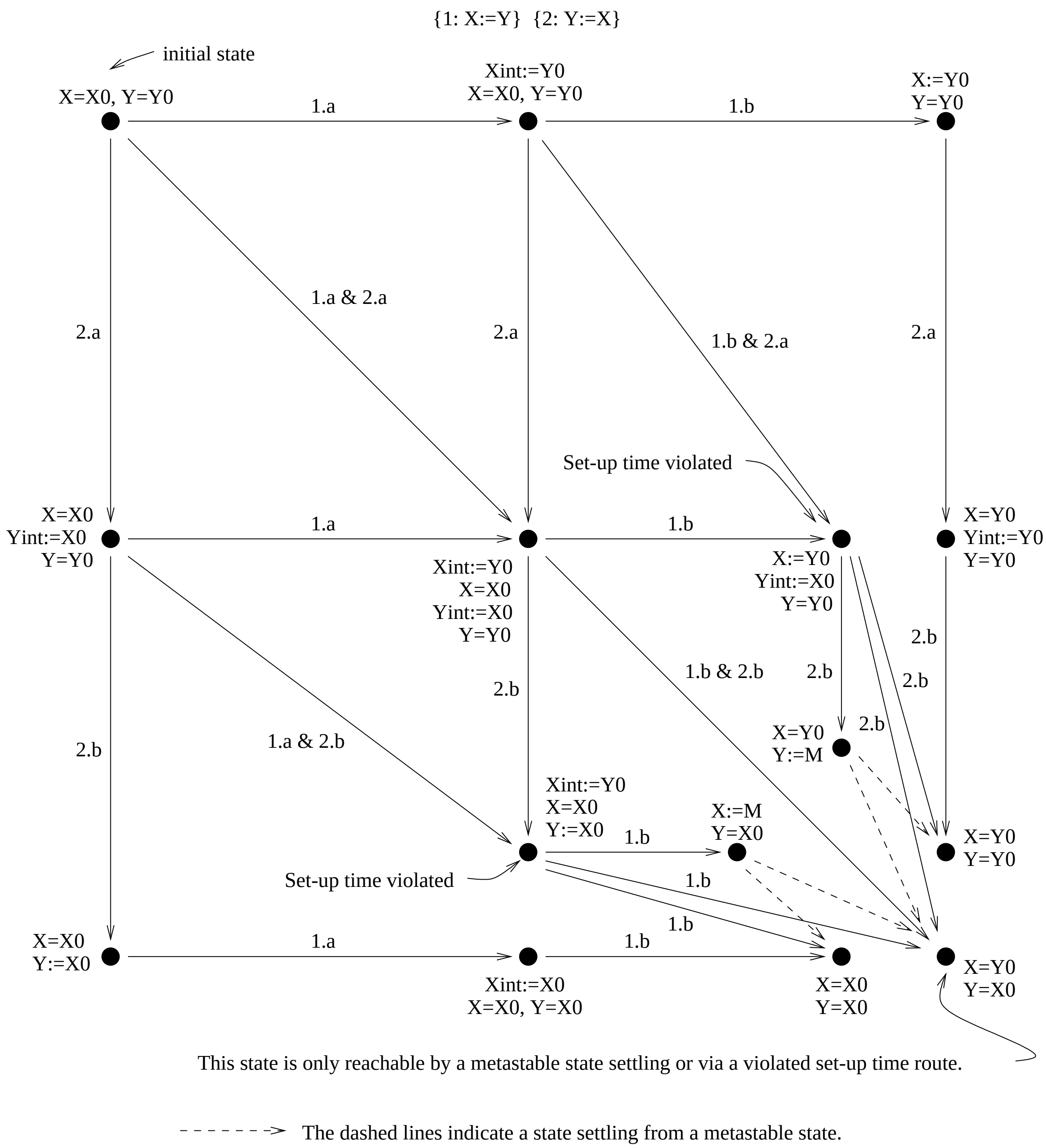
the start and end of the set-up time, and determine if the input has changed during this period. This is a valid test because the timing of changes on the device's input cannot be guaranteed due to the asynchronous operation of the other process. The other possibility for inducing metastable operation is if the device is supplied with a clock pulse which is insufficient to trigger the device properly. This is assumed unlikely as the processor controlling the device is synchronous with it.

It can be seen that in some cases a three stage model could be used for accuracy, so that the propagation delay is modelled as in the two level two stage model, see Figure 3.5. This graph shows that all the states are reachable without the three stage model, however this involves relying on the random settling of a metastable state from a violated set-up time for complete coverage. In reachability analysis this is not a problem because all possible states are searched. The two and three stage models for three level modelling are shown in Figures 3.6 and 3.7.

3.5 The modelling approach

In this section the three operating styles are discussed in relation to the minimum model required for accurate analysis. The minimum model is required to keep the size of the analysis state space as manageable as possible, thereby making any errors more obvious and results easier to obtain.

If the system being modelled has only one processor, capable of executing one instruction at a time, then at all times two level, single stage modelling is accurate. The number of processes, whether they are executed completely one after another or with their instructions interleaved makes no difference to the model. It is assumed here that the processor used is synchronous with a controlling clock pulse, and that the processor is well designed. This means the processor is being run at a rate at which all its internal elements have time to complete their operations and settle before the start of the next cycle. With this assumption it is clear that instructions will either be executed before or after one another, and all results can be assumed to have settled before any value is sampled (in the next cycle). It is because of this synchronous sequential operation that sampling unsettled values will not happen and therefore that no metastable operation is possible.



In real hardware, any of the above routes are useable, however when modelling in Petri nets we can only execute one transition at a time, therefore only routes with single numbers on can be executed. All diagonals in this graph are reachable from single firing arcs as well. This means two stage modelling is always adequate. However it should be noted that the state at the bottom right of this graph is only reachable by relying on a metastable state settling to it, or a route with a violated set-up time. But this doesn't matter because being state space analysis, all possible routes are investigated.

In this three level case the two stage modelling is always required because of the detection of violated set-up time. As in the two level case, not all states may be required, but no reductions can be used here.

Figure 3.5: Illustration of need for three-stage modelling for three-level asynchronous systems.

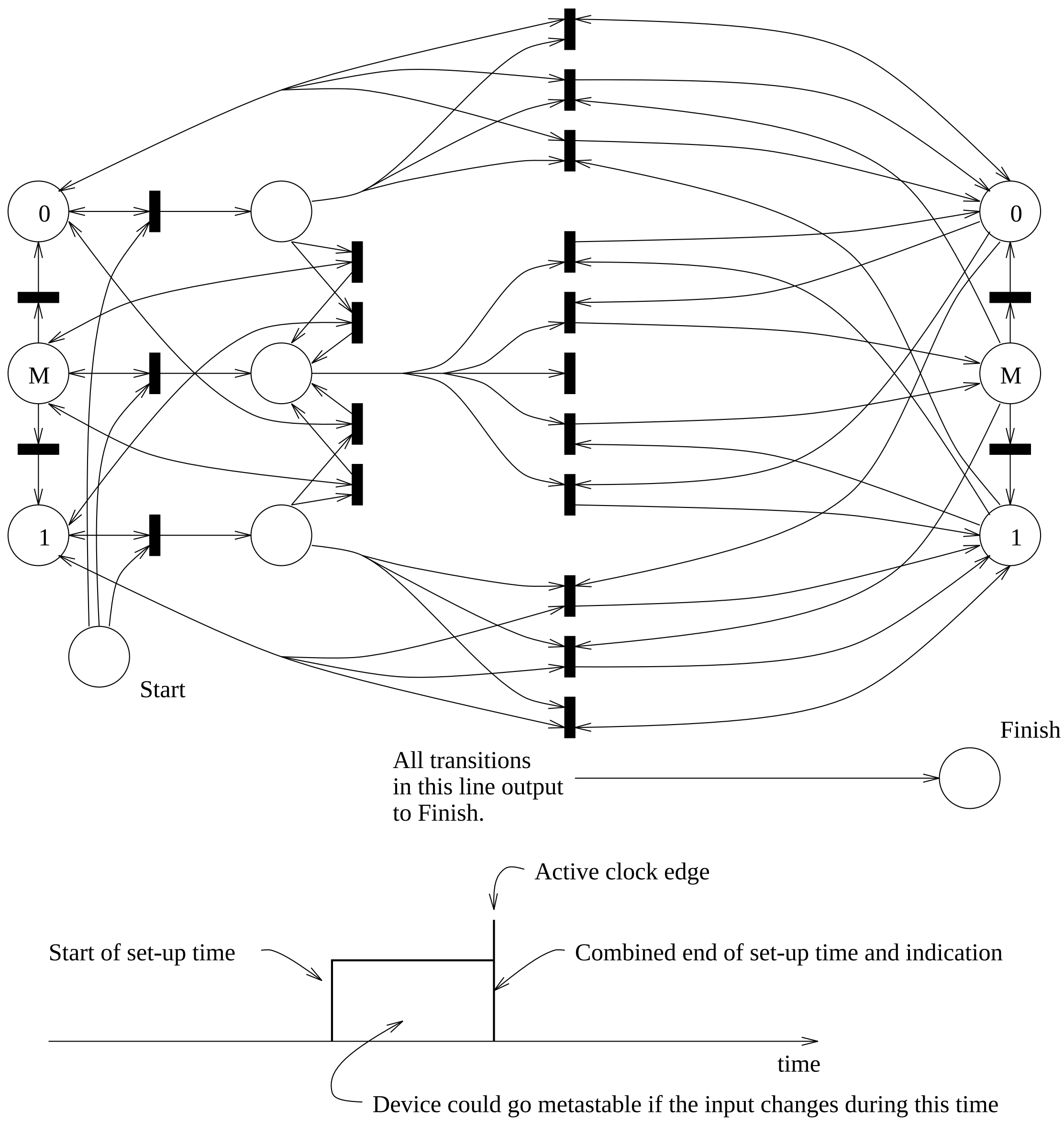


Figure 3.6: Petri net model showing two stage model for three level outputs.

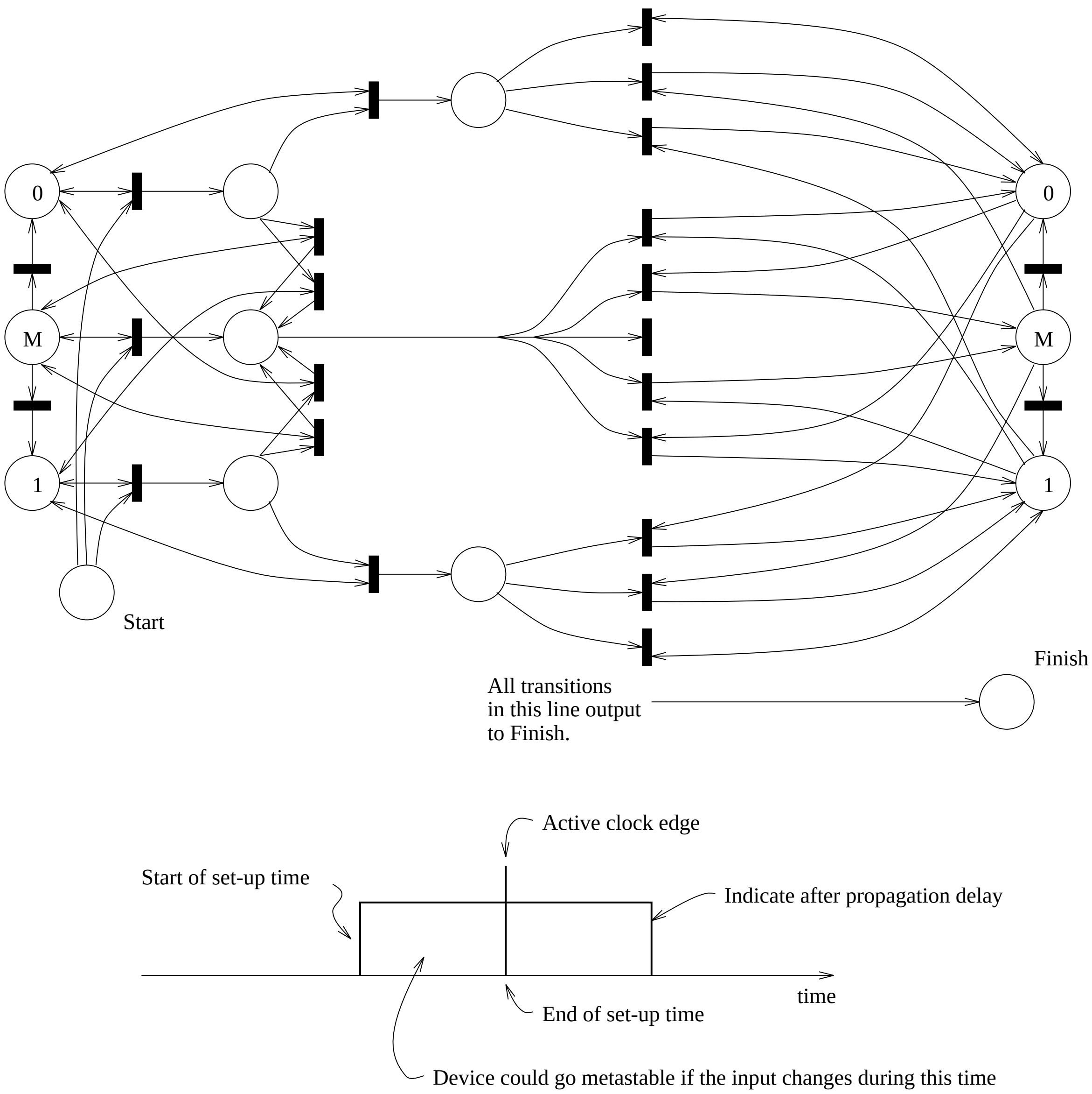


Figure 3.7: Petri net model showing three stage model for three level outputs.

Therefore two level modelling e.g. '0' and '1' is accurate. The reason for requiring a two stage model is described in the next few paragraphs, and it will be shown that this is only necessary when concurrent operation is implemented.

The two other types of processor configuration, synchronous concurrent and asynchronous concurrent, are not so straightforward to model. For the first of these, synchronous and concurrent implies that two or more (depending of the number of processors) instructions can be executed at the same time. This implies there are three possible interleavings, instructions executed one before another, one after another, and overlapping in time. It is the two instructions executing at the same time which causes the problem with the inadequacy of single stage modelling. This is highlighted in Figure 3.3 and an example in [LS89]. There are two processes $\{1 : X := Y\}$ & $\{2 : Y := X\}$. The brackets '$\{\ldots\}$' are used to show that the operator is executed within a process. The number '1:' is used to distinguish between processes, i.e. process number 1, 2 etc. It can be seen from the top diagram in Figure 3.3 that the final state is different if both processes are executed together rather than one after the other. Therefore if this situation were modelled using single stage models, then states would be missed out from the state space analysis. If however one of the processes instructions is modelled as a two stage model (the lower diagram in Figure 3.3), then it can be seen that all states are reachable using a sequential firing route. The route to determining that only one instruction needs to be modelled using two stages should include a check using both instructions modelled using two stages for completeness. This is shown in Figure 3.8. It can be seen from this that the final states match those in Figure 3.3, therefore clarifying that a model with one instruction modelled as a one stage model and the other instruction modelled as a two stage model is accurate.

Two level modelling is still accurate in this case. This is because all the processors are run synchronously, and therefore the set-up and hold times of all latches can be guaranteed to be satisfied, so that metastable operation does not occur.

This form of construction is possible if the processors are close together, and the connection distances between them is kept to a minimum. If however the processors are widely separated, then the transmission of a global synchronizing clock signal is difficult. This is exacerbated if one of the communicating elements is moving, e.g. on a plane or in a car. To try and overcome this problem processes can be run asyn-

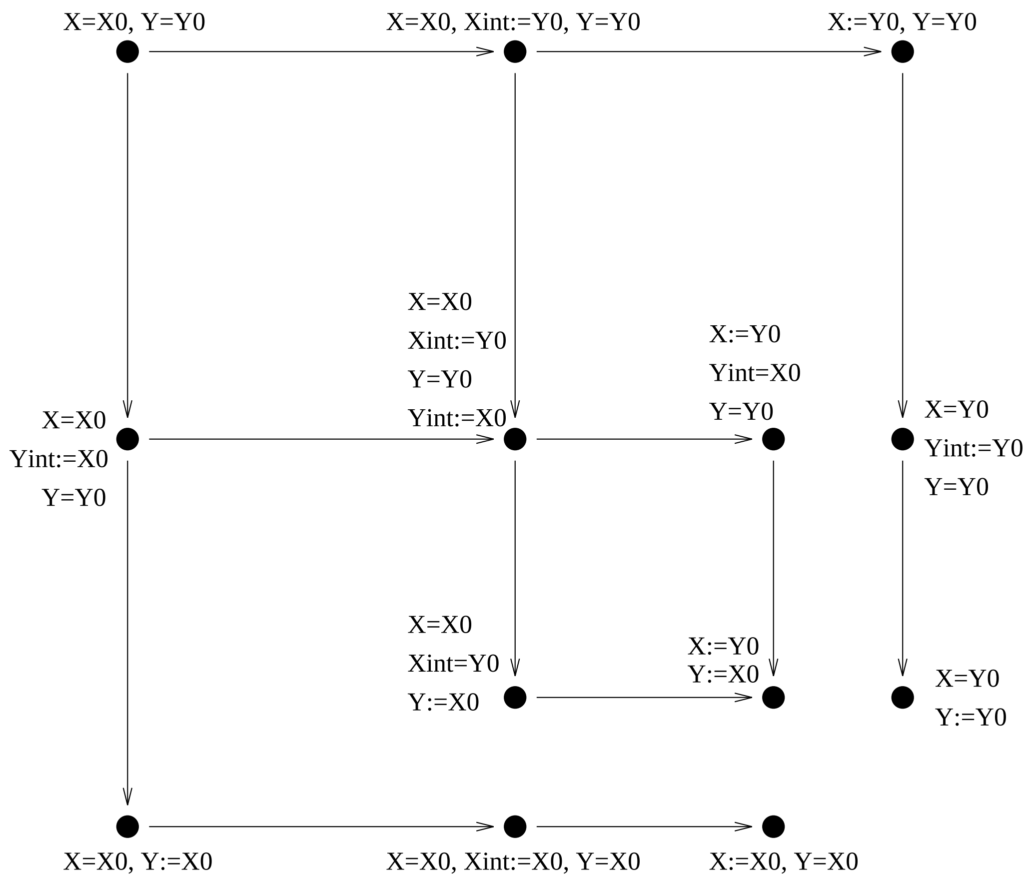


Figure 3.8: Initial model of $\{1 : X := Y\}\&\{2 : Y := X\}$ using two stage modelling.

chronously and concurrently. This is an arrangement known as globally asynchronous, locally synchronous. Locally synchronous refers to the fact that synchronous sequential processors are used locally, but they communicate and operate asynchronously with respect to remote processors, i.e. each has its own local clock which can be run at any independent speed.

Running a distributed system using local independent clocks has advantages from the safety critical view point. If a whole system is relying on a global clock signal coming from one location, and that location fails, then the remote locations will stop. However this advantage brings with it the disadvantage that running with independent clock signals means that communicating device set-up and hold times may be violated, and so metastable operation is possible.

With two processes using independent clocks there are five possible interleavings between each processes instructions, (i) one before another, (ii) one after another, (iii) together, (iv) one starts slightly before the other and indicates its new value during the others set-up and hold time, (v) and one starts slightly after the other and has its set-up and hold time violated. The possible interleavings depend on the operations being modelled, (Figure 3.9). For accurate modelling and analysis three level models, i.e. '0', 'M' and '1', are required. It was described above that the minimum model for three level modelling uses two stages, however there are certain cases where three stages could be used. Note 'could' and not 'must'. This is because state space analysis is being performed and all reachable states will be exhaustively examined. It can be seen from Figure 3.5 that to reach the state where $X = Y0$ and $Y = X0$ (bottom right of the Figure) the route relies on the settling from a metastable state.

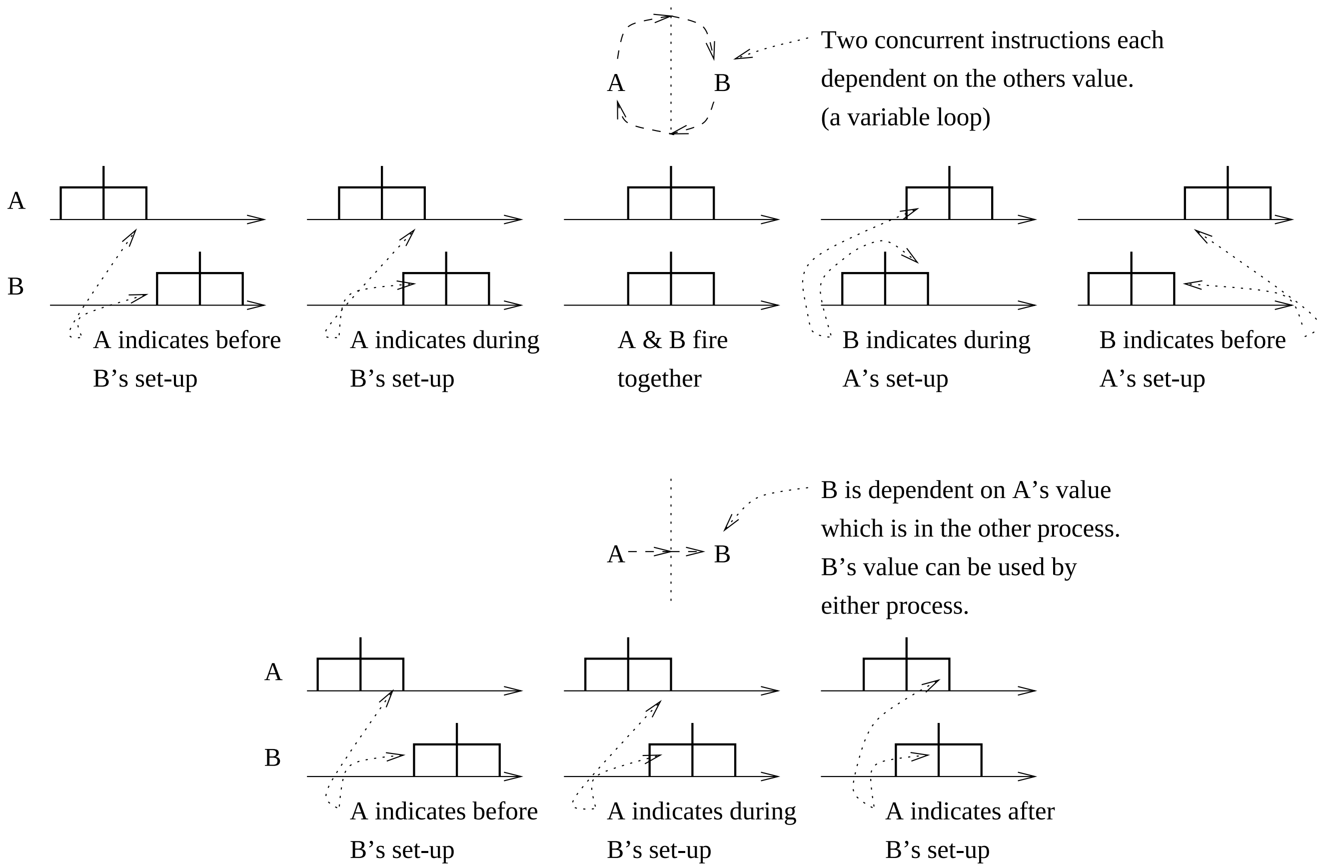


Figure 3.9: Different instruction interleaving possibilities depend upon shared variable paths.

In the two concurrent cases above, it is only where variables are shared that the extra modelling is required, and this is also dependent on the states reachable in a particular arrangement. In the next sections several different configurations are investigated to highlight where special modelling considerations are required. The included configurations are not an exhaustive collection.

3.6 Modelling the different variables

In this section the models for different concurrent accesses to shared variables are presented. The description of single processor systems has shown that all instructions may be modelled atomically using the single stage and two level approach, with instructions being interleaved so they can happen before or after one another but never concurrently. Here the emphasis is on concurrent interleaving, both synchronous and asynchronous.

The convention used is that lower case characters represent local variables, upper case characters represent shared variables and '?' represents a local variable which was set to a value earlier in the process. Any variable may only be set (written) by one process, but may be referenced (read) by one or more. This is because the writing process is destructive, and two concurrent writes cannot be handled, however reading is non-destructive and therefore concurrency is possible.

3.6.1 One shared variable

$\{1 : A :=?\}$ $\{2 : x := A\}$

In this example there is one shared variable A, which is written in process 1 and read in process 2. The first model of these processes supports the claim that atomic modelling is accurate if multiple processes are run sequentially on a single processor. Figure 3.10 shows the single time line representing the execution of the single processor. It can be seen, that because the processor is running synchronously from a constant fixed frequency clock the instruction in process 1 has indicated its new value before the set-up period of process 2's instruction begins. Therefore at the sampling time the indicated output value will be the same as the internal state of a multi-stage model. Therefore the two stages can be safely combined. This atomic execution is valid for any analysis on single processor systems, and therefore the remainder of this chapter concentrates on concurrent analysis.

Examining concurrent execution, the writing of A can be modelled atomically, i.e. using a single stage model. This is because it samples a value which is set locally, therefore sequentially, and thus it will be sampling a settled value. Variable x is set

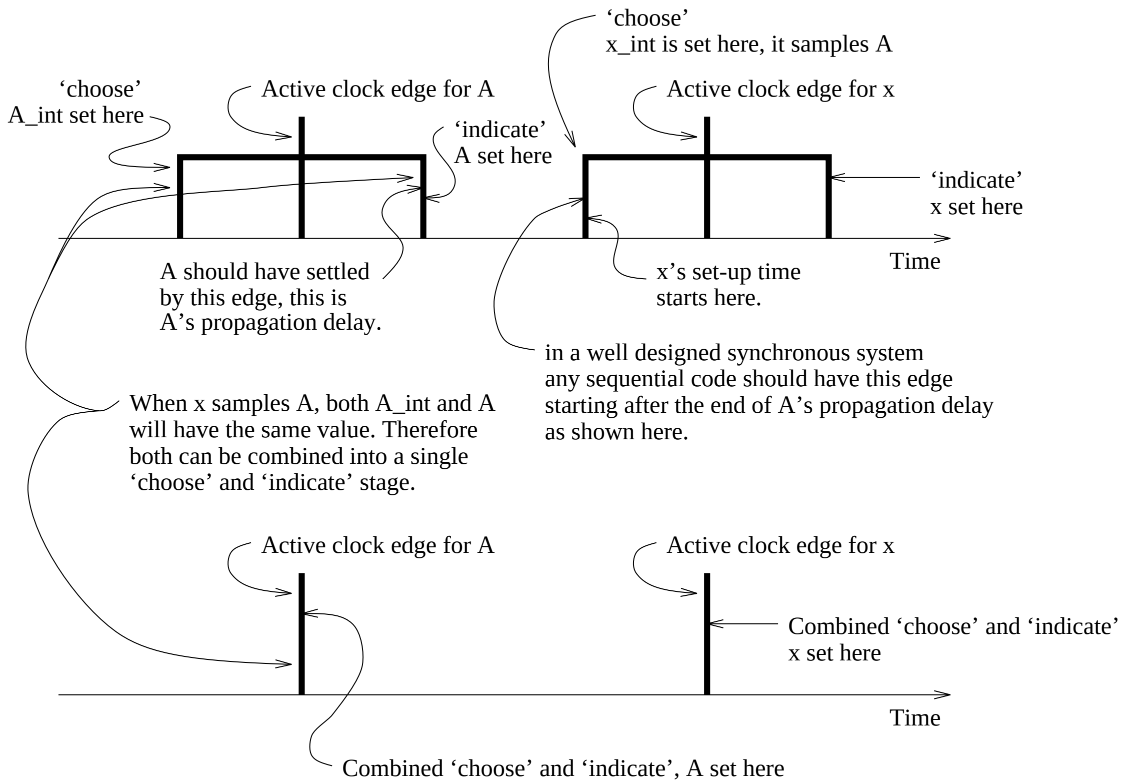


Figure 3.10: The time line representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on a single processor.

in process 2, its value obtained by sampling A. The possible instruction interleaving is dependent on whether the system is synchronous or asynchronous. If the system is synchronous then x may sample A either before or after A indicates a new value, see Figure 3.11. This means x can also be modelled using an atomic model with two levels because entering a metastable state is prohibited by the synchonization. If however the system is asynchronous, then x may sample A at any time and this includes sampling A as it indicates a new value, which could violate the set-up time of x and possibly cause a metastable state. The possible interleavings are (i) x samples A before it indicates a new value, (ii) after it indicates, or (iii) during its indication. See Figure 3.12. This means x should be modelled using a three level model to enable the metastable state to be represented.

$\{1 : A := \overline{A}\}$ $\{2 : x := A\}$

This example has the same modelling requirements as the above example. Process 1 sets A to be an alternating value by sampling its own last value. This must be done synchronously with itself, thus entering a metastable state is prohibited and an

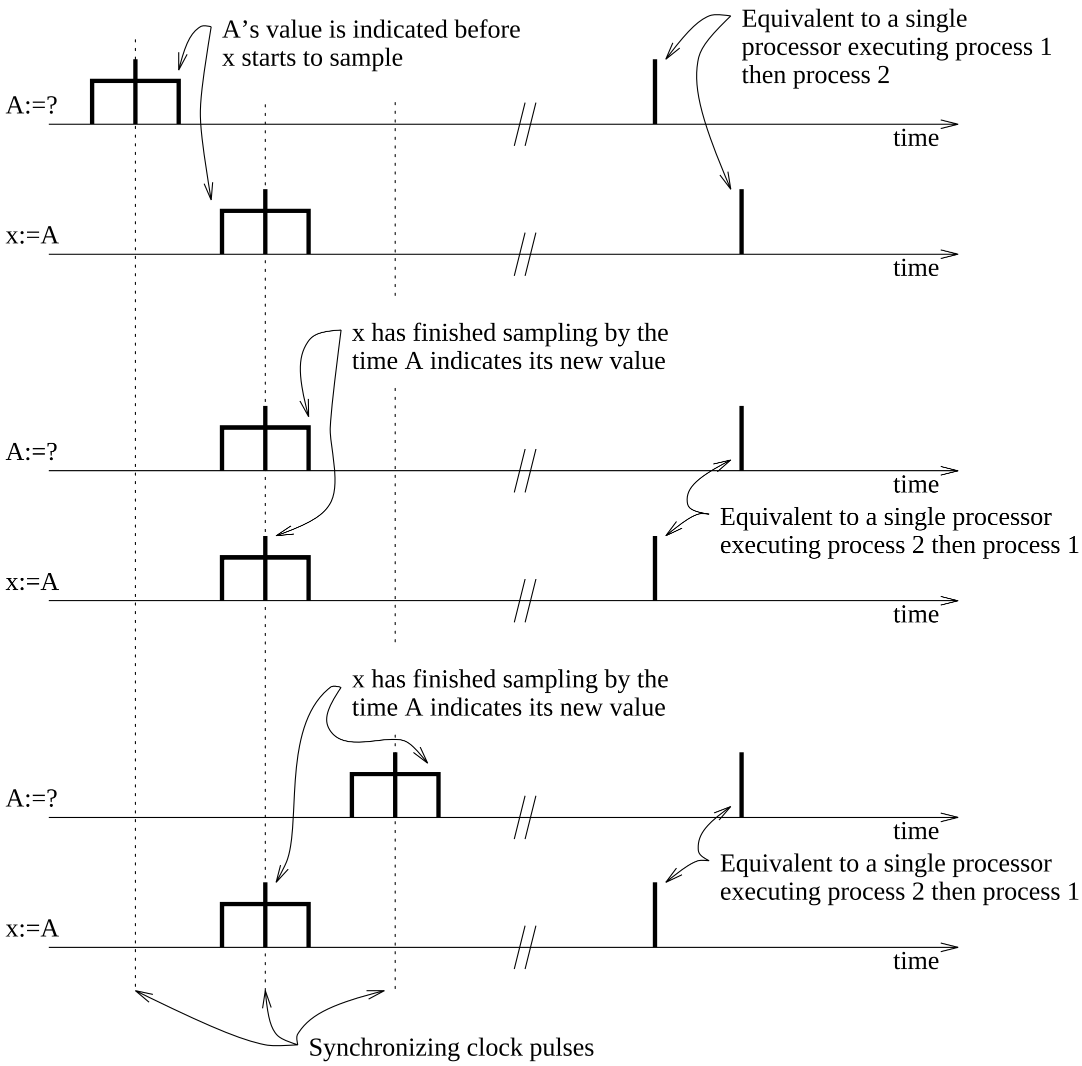


Figure 3.11: The time lines representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on two processors synchronously.

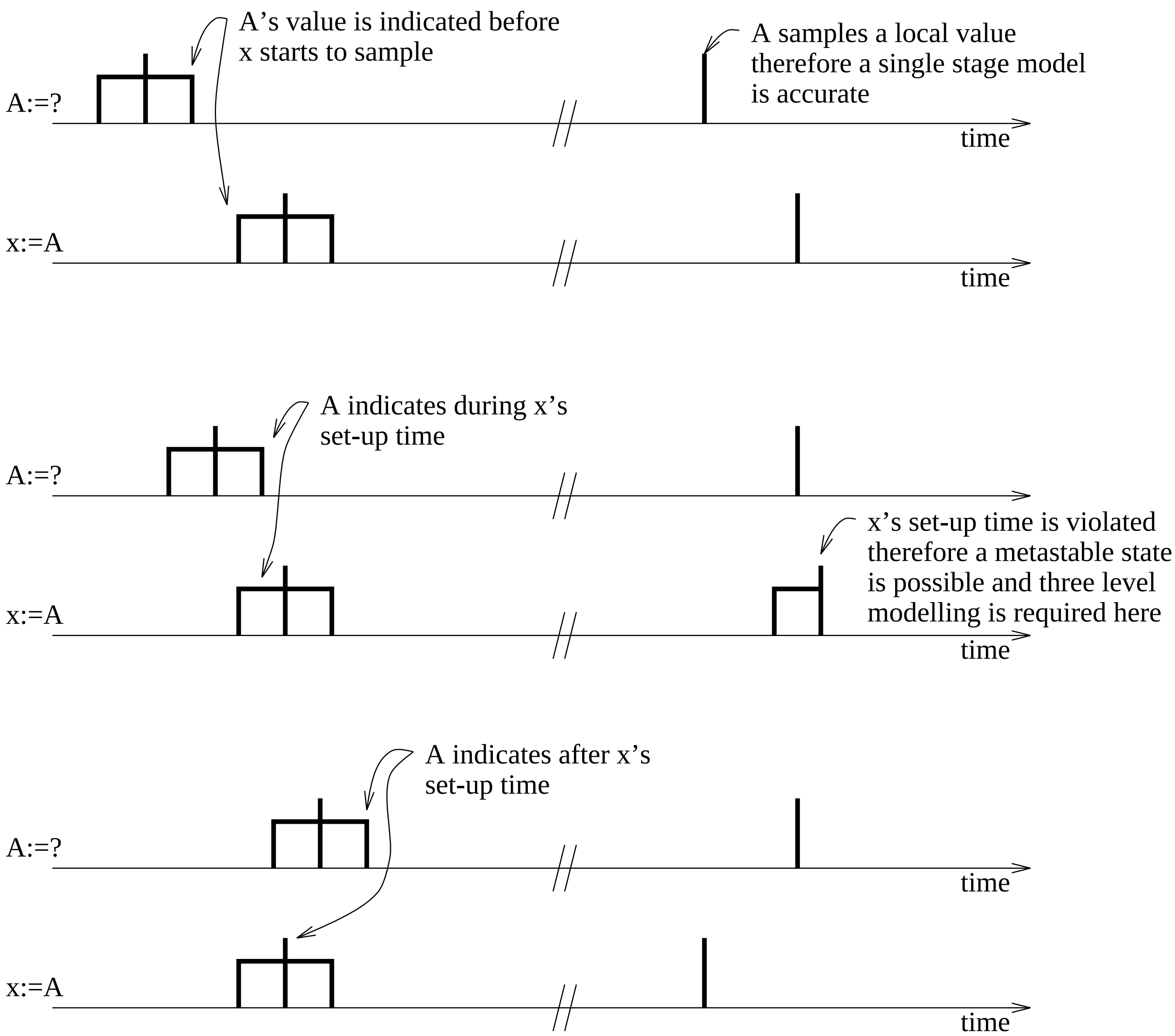


Figure 3.12: The time lines representing $\{1 : A :=?\}$ $\{2 : x := A\}$ executing on two processors asynchronously.

atomic model of A is accurate using two levels. Process 2 is the same as in the last example.

3.6.2 Two shared variables

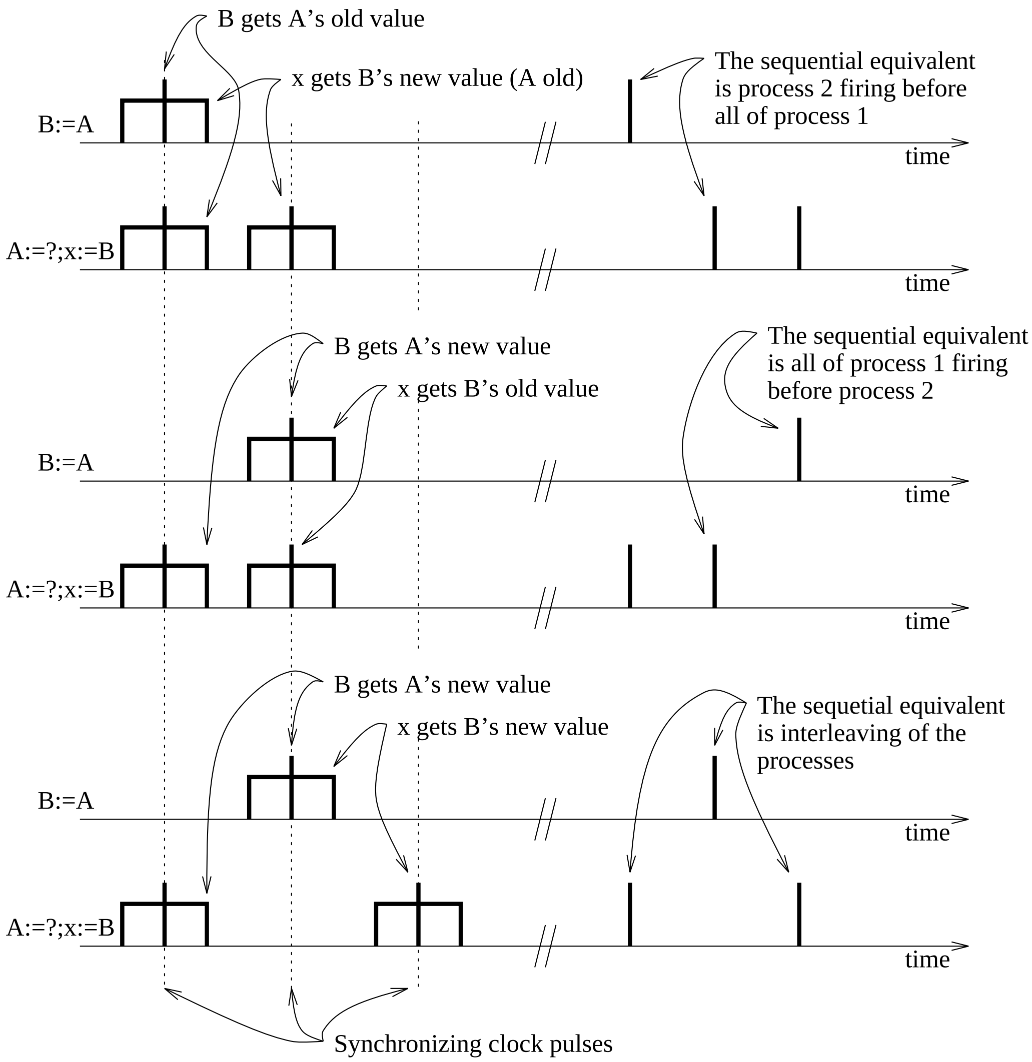
$\{1 : A :=?; x := B\}$ $\{2 : B := A\}$

Here process 1 first sets the shared variable A to a local value which was set earlier, then it samples B to obtain x's new value. In process 2, there are two accesses to two shared variables in one instruction. B samples the value of A. '$A :=?$' in process one can be modelled atomically because A is sampling a locally set value. The models for '$x := B$' and '$B := A$' depend on whether the execution is synchronous or asynchronous. In the synchronous case neither x or B can go metastable. The possible interleavings are shown in Figure 3.13. If the execution is asynchronous then both x and B may go metastable due to violated set-up times, therefore they must both be modelled using three level models. The possible interleavings are shown in Figure 3.14.

None of the examples so far have required reachable state analysis. This is because only open loop configurations have been shown. Open loop indicates the dependence on the value of a variable somewhere else in the system. The following two examples are closed loop in that they rely only on values set within the shown instructions, and are independent of external values.

$\{1 : A := B\}$ $\{2 : B := A\}$

In this example there are two variables which sample up to date values of each other. Typically, to be of use, one of the equality operators would sample the inverse of the other variable, but a latch normally produces its output and an inverse output at the same time, so either can be analysed. This problem is used as an example to show why two stage modelling is required in the synchronous case, see Figure 3.3, and why three stage modelling may be required in the asynchronous case, see Figure 3.5. The time lines for the synchronous case are shown in Figure 3.15, and for the asynchronous case in Figure 3.16. It can be seen from the middle time line in both Figures that



This final case is not legal in true programming standards because there is a gap in the execution of the instructions. If however this is being run on a micro-processor, then the number of clock cycles the instruction 'A:=?' should be checked. If it only takes one clock cycle then this interleaving is not possible and could lead to erroneous results. To cure this, the instructions 'A:=?' and 'x:=B' should be fired in parallel. This allows the top two interleaving patterns to be modelled, but does not allow the bottom one.

Figure 3.13: The time lines representing $\{1 : A :=?; x := B\}$ $\{2 : B := A\}$ executing synchronously.

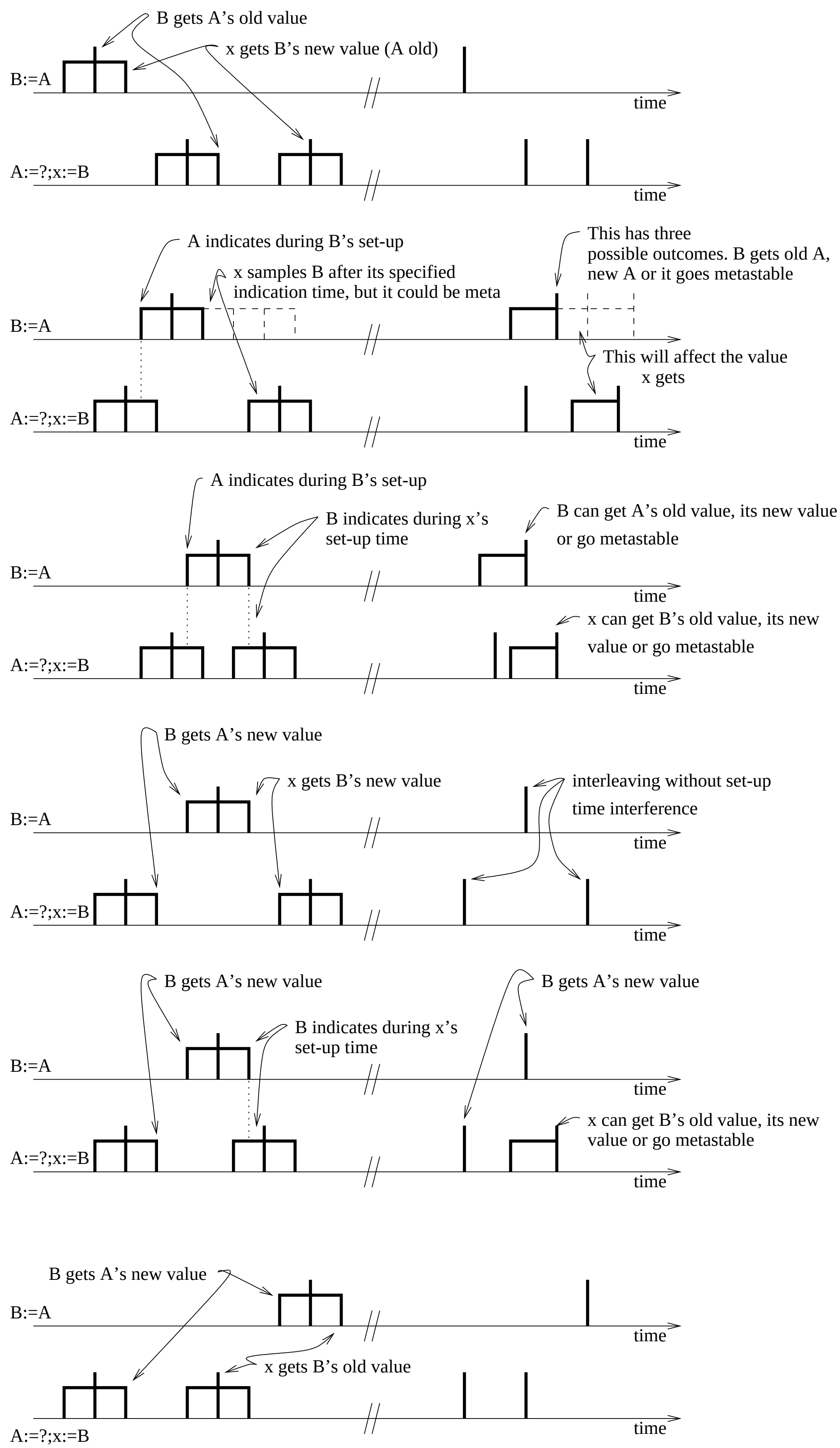


Figure 3.14: The time lines representing $\{1 : A :=?; x := B\}$ $\{2 : B := A\}$ executing asynchronously.

the set-up requirements are not violated. In both cases an extra stage in the model should be used. In the synchronous case this is vital, but in the asynchronous case, it is optional as the model is already two stage, and the state coverage is already complete, (see the earlier section on one, two and three stage modelling).

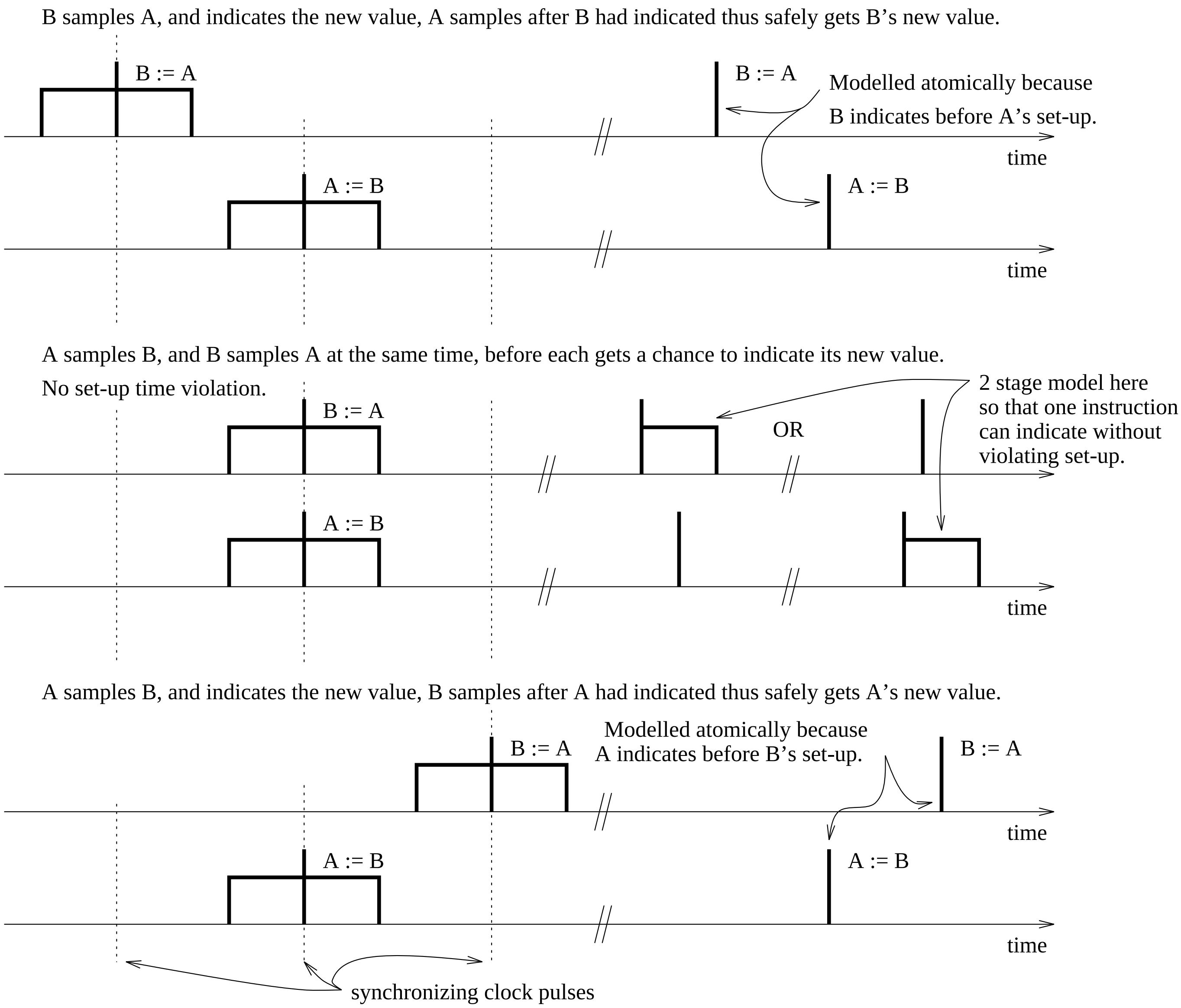


Figure 3.15: Parallel time lines for $\{1 : A := B\}$ & $\{2 : B := A\}$ executing synchronously.

There are four possible versions of these processes :- $\{A := B\}\{B := A\}$, $\{A := \overline{B}\}\{B := A\}$, $\{A := B\}\{B := \overline{A}\}$ and $\{A := \overline{B}\}\{B := \overline{A}\}$. The two and three level state spaces of these are shown in Figure 3.17. From these state spaces it can seen that all possible initial states should be modelled, otherwise in the two level cases starting at all 0's, some states are not reachable. Consider, for example, at $\{A := B\}\{B := A\}$. If a state space search for reachable states starting at 00 is performed, then it is found

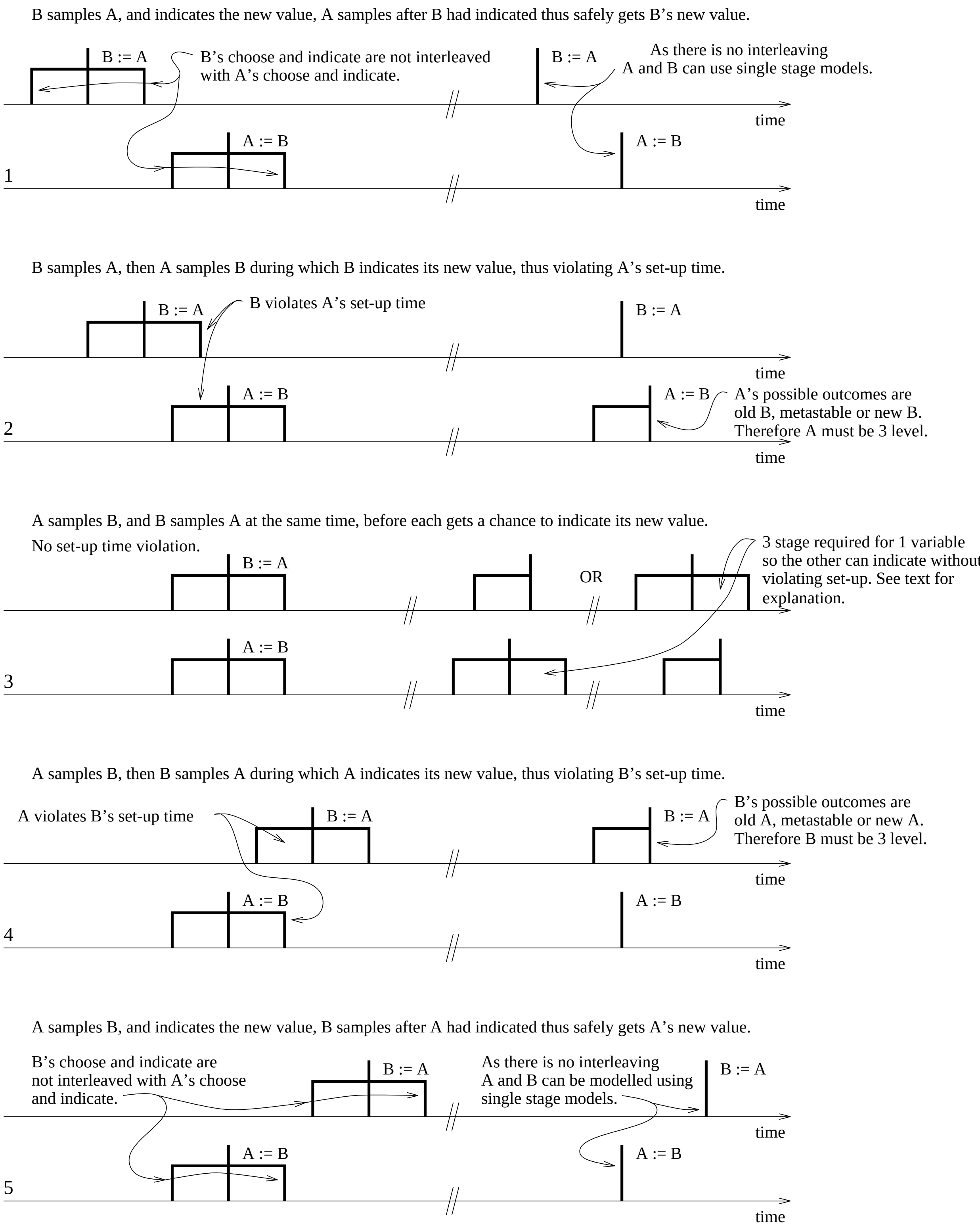


Figure 3.16: Parallel time lines for $\{1 : A := B\}$ & $\{2 : B := A\}$ executing asynchronously.

that no other state is reachable. In the three level case with metastable states, the effect of settling covers all states including the normally unreachable ones. It can be seen here that for the $\{A := \overline{B}\}\{B := A\}$ and $\{A := B\}\{B := \overline{A}\}$ cases, single stage modelling sufficient because all states can be reached from single firing routes (this is because all the double firing routes are also mirrored using single firing routes).

This degree of care in modeling can allow removal of unecessary stages. This allows a reduction in the generated state space size as every model stage has the potential of doubling the state space size. This will help during the analysis of results due to the smaller state spaces. This becomes more apparent later when more complex algorithms are modelled and analysed.

$\{1 : A := c \| c := B\}\ \{2 : B := A\}$

Here there are three instructions in a loop, with two of the assignments happening in parallel ($\|$). Note that to enable instructions to be executed in parallel, they must be modelled together, i.e. the firing of one transition must change states in both variables. A should never go metastable as it is sampling c (synchronously), which was set in the last cycle (and presumably settled by now). This is a closed loop system like the last example except that there is a temporal break in the loop caused by c being sampled and set concurrently. The time lines for synchronous operation are shown in Figure 3.18, and for asynchronous operation in Figure 3.19.

Both Figure 3.18 and 3.20 show that for synchronous operation (i.e. two level), B should be modelled using a two stage model. When a reachable state analysis is performed, it is found that the two states '100' and '011' are the two states requiring the extra stage. If a reachable state search is performed from the starting state of '000' then these states are not reachable; this is not a valid reason to drop them because a latch at power-up may start from any settled state, however, no other state in the system can reach this state while running synchronously, and even if the system does start in this state it can get out of it. If the system does start in this state however, the two stage model does allow a route between the two states if both processes fire together, which is certainly possible, see Figure 3.21. If at power-up the system was supplied with a reset pulse (to start all latches at '0'), then these states would never be reachable, and single stage atomic modelling would be safe.

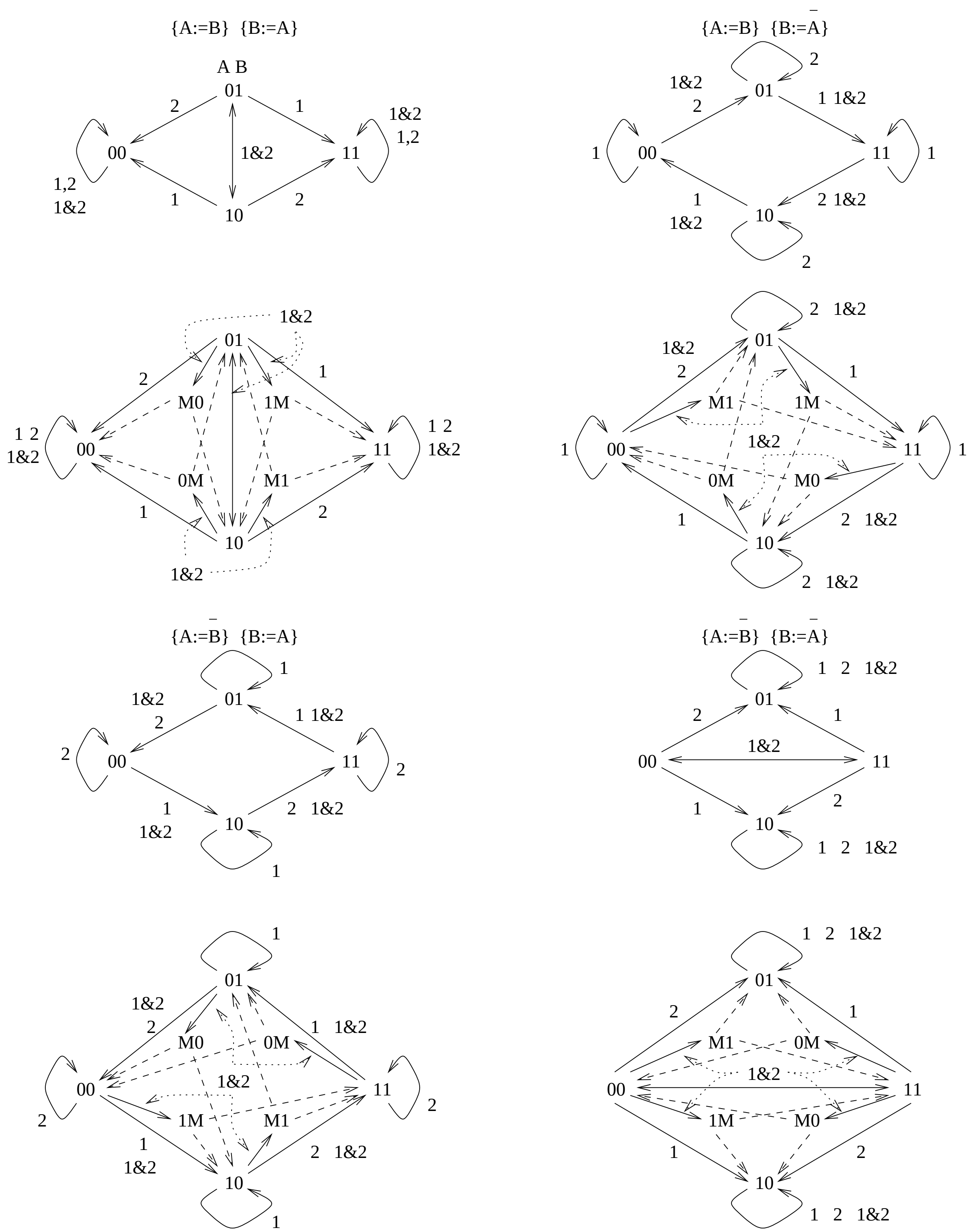


Figure 3.17: Two and three level state spaces for all versions of $\{1 : A := B\}$ & $\{2 : B := A\}$.

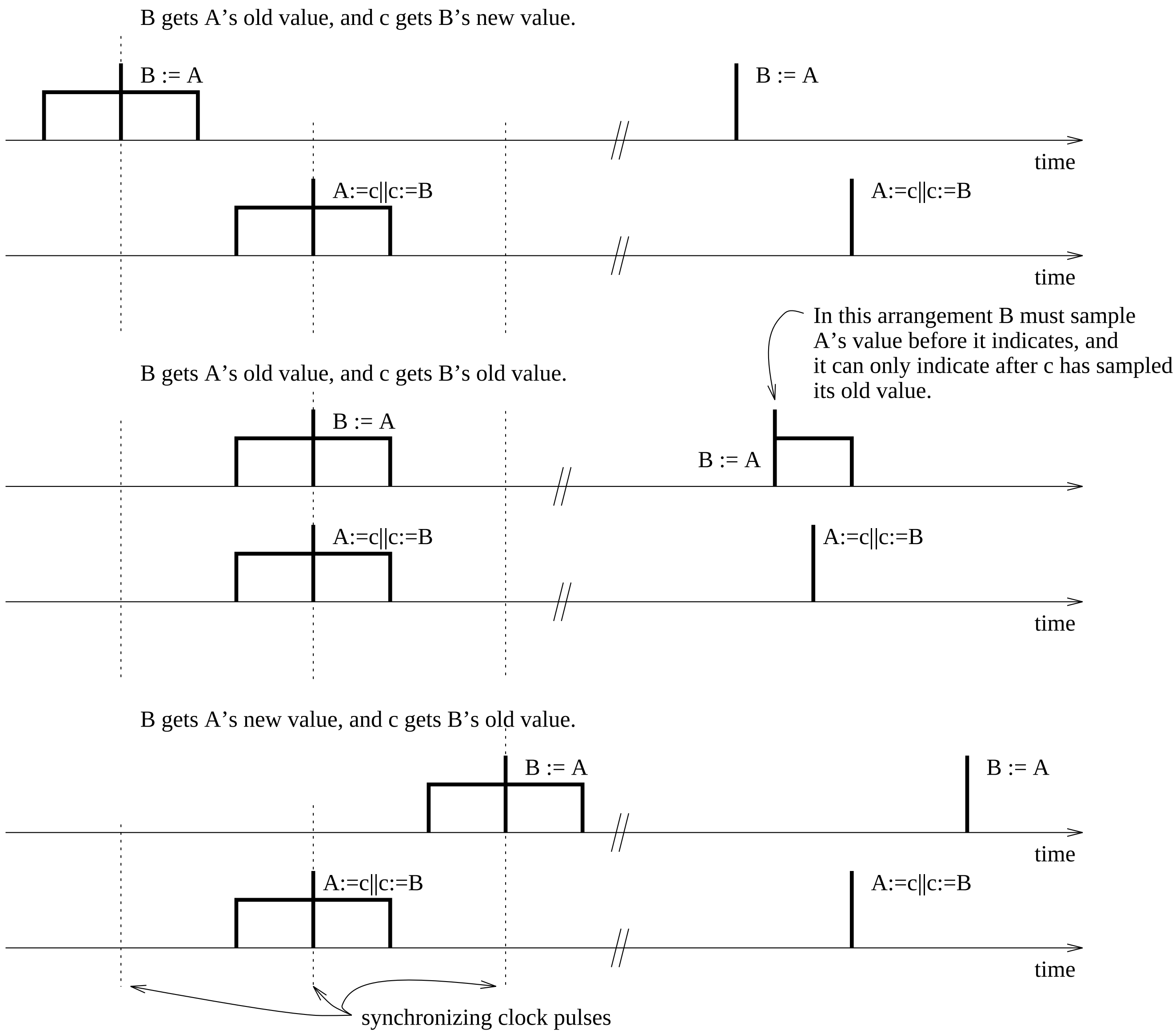


Figure 3.18: Parallel time lines for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$ executing synchronously.

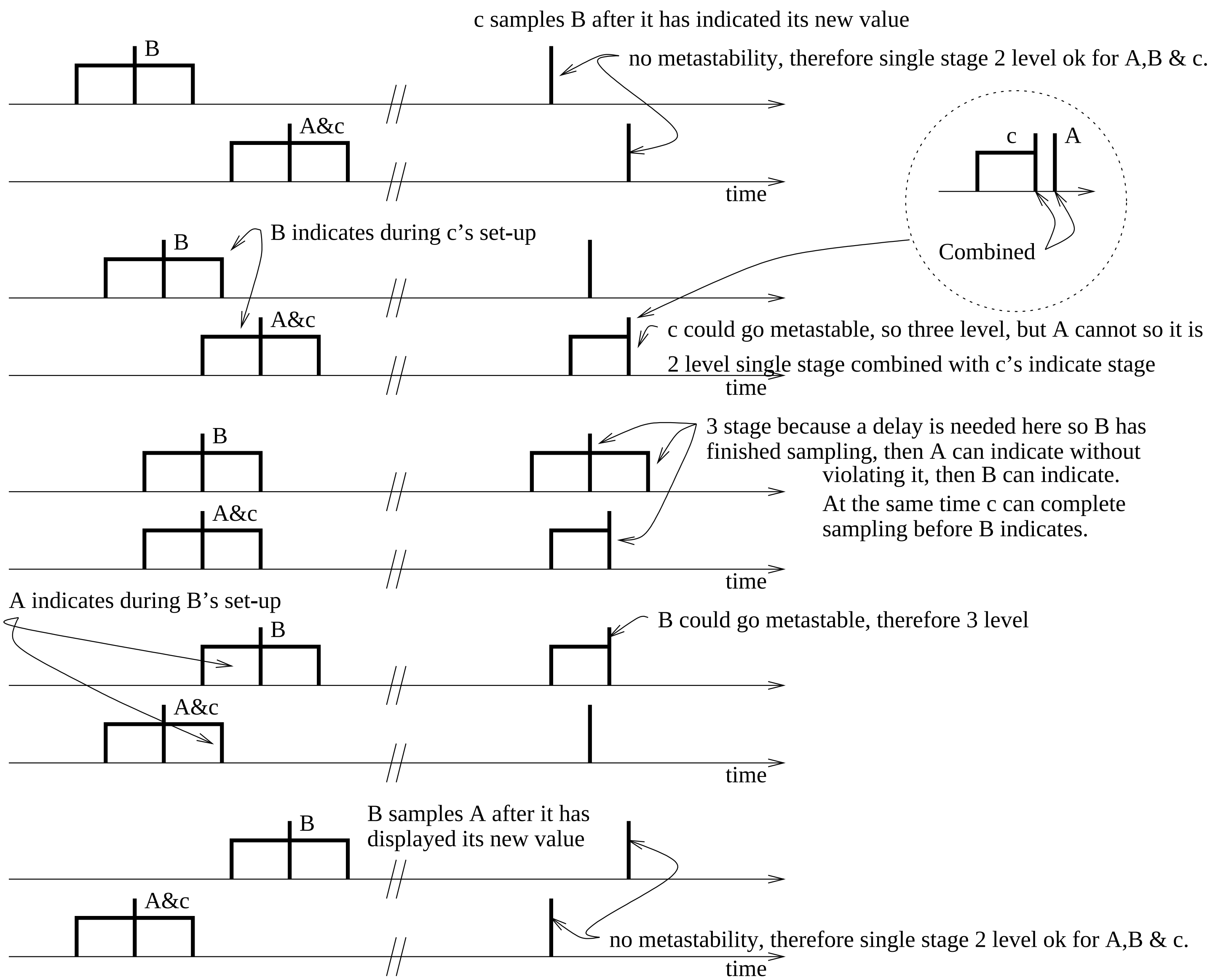


Figure 3.19: Parallel time lines for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$ executing asynchronously.

Turning to asynchronous execution, these states are certainly reachable because a metastable state could settle to it. The two states are automatically covered under three level modelling anyway.

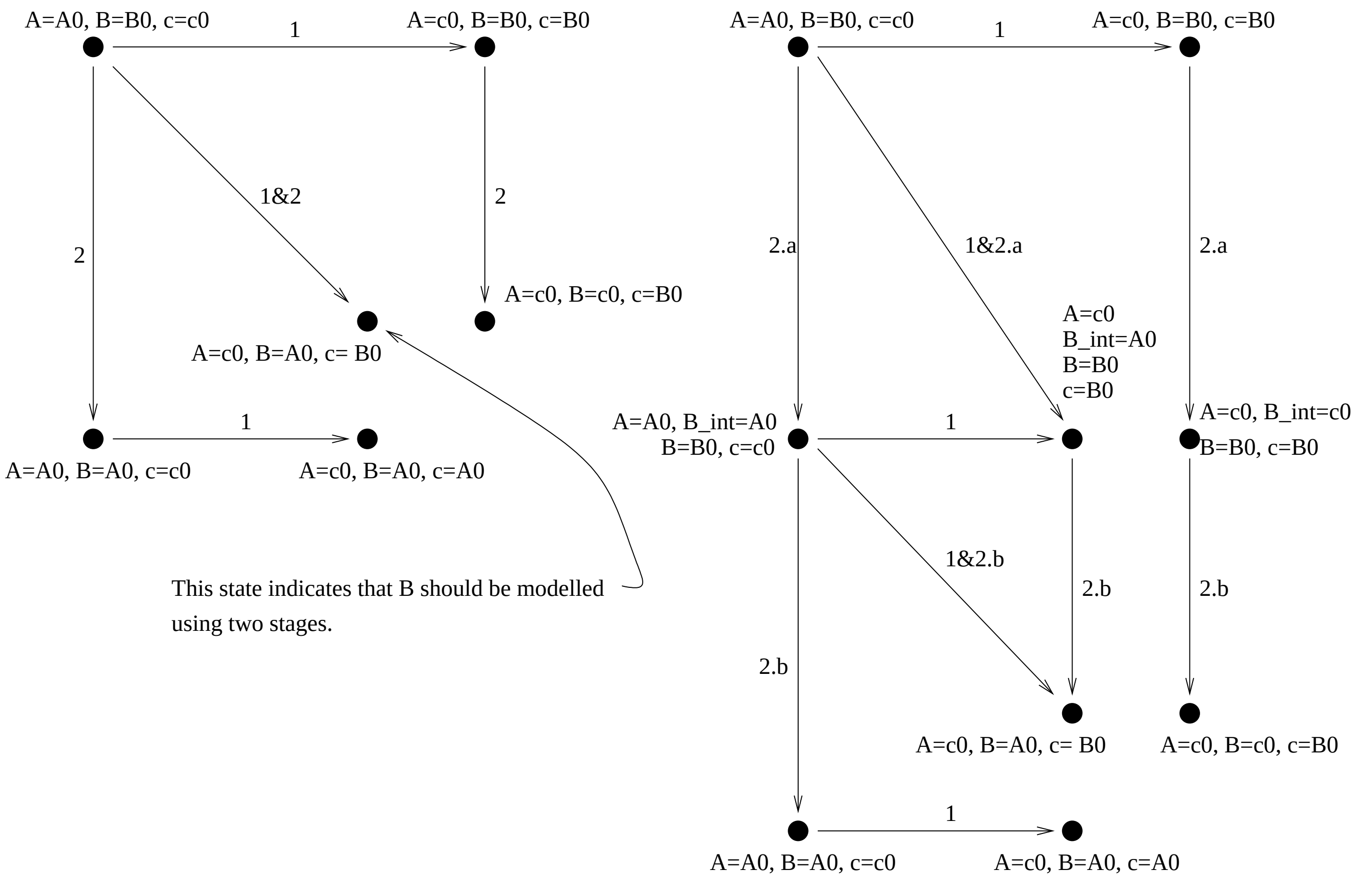


Figure 3.20: The state graph for $\{1 : A := c \| c := B\}$ & $\{2 : B := A\}$.

3.7 Novelty and application

The novelty of this chapter is the modelling of the metastable state as an extra state 'M' [CXYD98, XCYD97, XCYD98] for use in state space analysis of asynchronous systems. Petri net models of latches with three levels permit more accurate analysis under asynchronous operation.

Latches are used as the basic building blocks of more complex asynchronous communication mechanisms which are presented in the next chapter. Some descriptions of these mechanisms in the literature make assumptions about the settling of this metastable state, and others are limited to running synchronized multiple processors. Most however, rely on atomic switching for their variables, which has been shown to

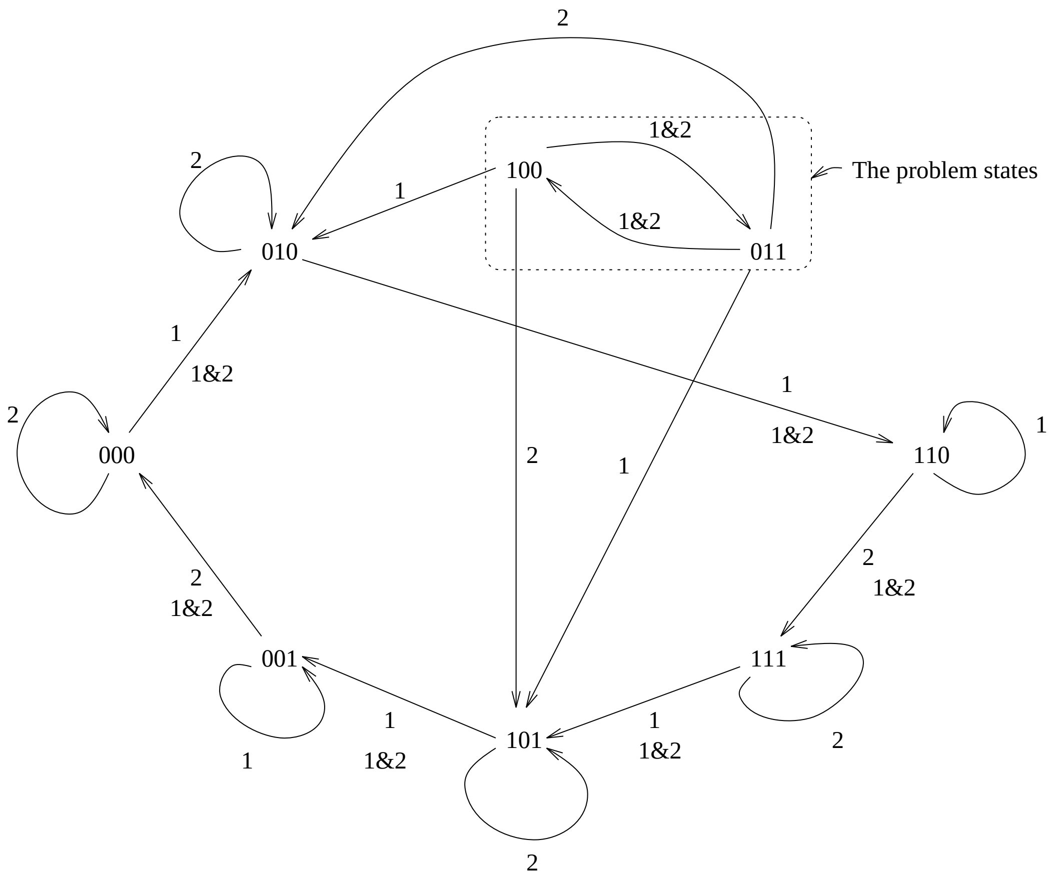


Figure 3.21: The reachable states of $\{1 : A := c || c := B\}$ & $\{2 : B := A\}$.

be insufficient.

The reachable state analysis described enables the models to be proved exhaustively complete. This aids later analysis when the latches are used within systems and communications mechanisms.

3.8 Conclusion

In this chapter new latch models were presented, including the ability to model the unavoidable metastable state when operating asynchronously. Model reductions were also presented which inherently reduce the reachable state space size and therefore aid the results analysis.

Two stage models ('choose' and 'indicate') have been used [Sim92a] in a role model analysis of a four slot mechanism. Using the complementary two level model analysis presented here, the number of system states can be reduced thus aiding the analysis.

It is also apparent from the examples in this chapter that in certain cases, an assumption of atomic switching is inaccurate because of omitted states. These examples represent a small number of the possible configurations, which were chosen to highlight the modelling problems.

Chapter 4

Modelling of Simple Asynchronous Communication Mechanisms

This chapter introduces the Petri net modelling techniques used in the modelling of ACMs. The ACM operating properties of data coherence, freshness and sequencing are defined and models introduced which are used to test each algorithm to see if it can maintain the properties. The author's reachability program *reach* is also introduced, it is this program which was used to analyze the models in this thesis. The last part of this chapter describes algorithms for ACMs with one, two and three slots. Each algorithm is tested and the failure of each algorithm under unconstrained operation is given.

4.1 Introduction to ACMs and wait free communications

All communication requires a protocol. Generally, a 'protocol' is defined as "a set of rules governing the transmission of data between two computers which cannot communicate directly" [Cha99]. Here we will be more specific and define a 'protocol' as - 'a set of rules governing the transmission of data between two processing elements through a communication element'. Communication between two processes can be categorized according to whether the communicating process performs destructive or non-destructive access [Sim94, MAS87] (Figure 4.1). The more well known types in Figure 4.1 are implemented with non-destructive writing: the channel, a FIFO (First In First Out) buffer, and the constant, which is set up at the beginning of any operations and remains during the whole execution of the system. The signal "allows event data to be passed from one process to another. This data can be overwritten at any time by the writer, but can only be actioned once by the reader. The signal is important in real-time systems as it avoids back propagation of temporal interaction effects (i.e. the actions (or inaction) of the reader have no direct effect on the timing of the writer)" [Sim94].

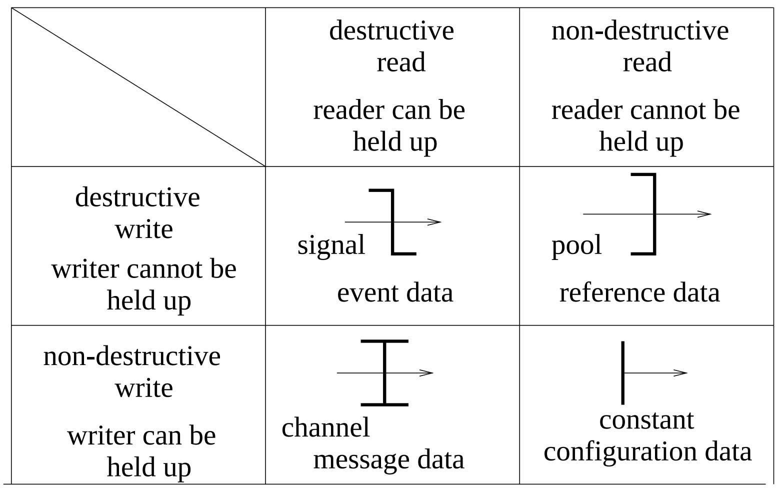


Figure 4.1: Different write and read types [Sim94].

The pool is the only type which facilitates fully asynchronous communication. A pool is realized by an ACM (Asynchronous Communication Mechanism), a wait free algorithm which enables non-blocking access to reference data. Concurrent processes can read or write with no timing constraints made on their access to the data being transmitted, except in the case of conditionally asynchronous mechanisms. A property of non-blocking or wait free communication is to provide temporal independence or temporal firewalls between communicating processes. This can be of vital importance in the construction of real time and safety critical systems [KK90], and can aid in execution time calculations of such systems. An ACM is a buffer or interface between two concurrent processes, with the properties that data can always be written into the mechanism and data can always be read from the mechanism, i.e. one process does not wait for the other, and more importantly one process cannot block or lockout the other process. This is only possible because the construction and operation of an ACM does not use semaphores, monitors, rendezvous, critical sections or mutual exclusion, which inevitably introduce temporal interference as the processes synchronize their communications operations, together with a failure mode associated with the implicit waiting and need for restart controls [Sim97b].

The possible applications of ACMs include on and off chip communications. As more and more processing power is required the two options are to increase the system speed or introduce parallelism. Currently processors run at 850MHz and this frequency is continually on the increase, but eventually the physical limit of the speed of light, or more practically the speed of a signal in copper wire will be the limiting factor. The figure of 20cm/ns in copper wire is given in [Tan90] for an unknown wire size, but this becomes slower as feature size and track size become smaller. Gradually as system speeds increase the ratio of wire delay to gate delay will become more significant [JA89, Anc82]. This will mean that as devices become larger and process technology gets smaller one corner of a chip will be a significant period of time behind its opposite corner. From [Fur] - "A circuit can only operate synchronously if all parts of it see the clock at the same time, at least to a reasonable approximation". "If the delay to a particular part of the circuit takes a significant part of a clock cycle-time, that part of the circuit cannot be viewed as being in step with other parts". This failure of synchronization is considerably worse if parts of the system are in relative motion e.g. moving on a plane or train and are coupled to stationary parts on the ground. Optical fibers have been suggested as a solution for the on chip clock distribution problem [Unk98], however, this simply postpones the point at which physical limits must be considered.

However the speed and distribution of the clock pulse are not the only problems facing synchronous operations. Electro Magnetic Compatibility, EMC, and low power operation also need review. With a system running at a fixed frequency, the system will radiate E.M. radiation at the system frequency and its multiples which is often unacceptable. For low power operation it is inefficient for all elements connected to the clock to be switching and consuming power even when not doing any useful work. Asynchronous operation does not suffer from these side effects and enables lower power operation. ACMs also enable the communication of on chip asynchronous data.

If parallelism is introduced to increase processing power then ACMs can be used for the communication of data between concurrently operating processes or off-chip communication. Parallelism could also be introduced for remote processing in a distributed system, but communication in such circumstances can be problematic and ACMs have advantages to offer in such circumstances.

ACMs are however limited by the type of communication that they offer. Their role is the communication of reference data. The non blocking nature of these mechanisms implies that multiple data areas are required, these being called slots, buffers or tracks in the literature. Multiple slots are required because only one reading or writing process can safely access a physical shared memory area or slot at any one instant in time. Unlike a channel or FIFO which can require waiting when it is full to the writing process, and when it is empty to the reading process, an ACM cannot cause waiting. A FIFO buffer has a non-destructive writing process and a destructive reading process, whereas an ACM has a destructive writing process and a non-destructive reading process. This means that the ACM which is usually full, can always be written to, and because reading does not destroy data it can always be re-read. This has the side effect that data can be lost or duplicated. If the writing process is not running as fast as the reading process then data will be duplicated. If the reading process is not running as fast as the writing process, then data will be lost. This is acceptable when communicating reference type data, where the latest data is required. Typically if three pieces of data are written into an ACM and then a piece is read from the ACM, the latest written piece of data is read, and the previous two would be lost. An exception is the arbitrating or overwriting FIFO [YKXK98]. (Holzmann [Hol91] states that data insertion and deletion from a stream of data is the result of lost synchronization).

As mentioned above ACMs provide a temporal break or firewall in real time systems, in which timing constraints can be imposed by HRT applications such as a control system in an aircraft flight control system. This type of application is usually safety critical. The temporal break allows a concurrent digital system to mimic an analogue system: there is no waiting for data, and can therefore be thought of as continuous data which has been sampled.

From a programming perspective an ACM can provide the equivalent of a global variable in a sequential program, but in a parallel asynchronous environment. However care must be taken because from atomicity theory, a parallel program is equivalent to a number of processes running sequentially on one processor, where the order of instruction interleaving between the processes is not specified. If by atomicity theory the access to the global variable is made sequential, there are two possible sequences. The writing process sets the variable before the other process reads its value, or the reading process obtains its value before a new value is written and therefore obtains a previous value. This is equivalent in a sequential program to changing the order of program statements, so care must be taken. If this is unacceptable for the application then an ACM is not a safe solution, and some form of synchronizing communication must be employed.

4.1.1 General ACM concepts

As mentioned in the introduction above, multiple data slots are required for fully asynchronous operation. It is assumed that the data or variable being communicated is at least a multiple bit construction. It does not matter whether this data is sent serially or in parallel, it is unsafe to allow a reading process access to a slot which is currently being written. Concurrent access to the same slot is tested for by 'data coherence' later in this chapter. A problem arises when a reading process accesses data which has not been completely written, because the data obtained could be half old data and half new data, as well as parts of the data signal being affected by metastability in individual bits within the data. By implementing multiple slots the ACM algorithm can allow only one process to access one slot at any one time while allowing simultaneous writing and reading. For this to operate correctly, steering logic or a steering algorithm is required. It is in this steering algorithm that the two communicating processes communicate directly. Metastability can affect control variables within the algorithms because no restrictions can be put on algorithm access in a fully asynchronous design. This is because at some level, the problem of concurrent access to a shared variable must be faced.

Ideally the variables in the steering algorithm should be the smallest type possible i.e. the binary bit. The binary bit can be affected by metastability but it is not affected as badly as setting multi-bit variables in an asynchronous environment. Lamport [Lam86] referred to the simplest register variable as 'safe' because the value obtained is only correct if not concurrent with any write. The term register refers to a multiple bit variable, therefore several bits could change at writing time. However, from this 'safe' register, Lamport then constructs a 'regular' register, which has the same properties as the safe register, with the added condition that a read which was concurrent with a write would only get the new or old value. A single binary bit has this property [Sim90b], if you allow a period of time long enough for the metastability to settle. Hence, a single binary bit is better in an asynchronous environment than a register. This is simply shown because if a register is multiple bit, for example '101', and the value '010' is being written to it, a read during the write could obtain '000', '001', '011', '110', '100' or '111' because the bits may change at different instants. Lamport takes this work further by constructing an 'atomic' register from regular registers. An 'atomic' register, is safe, and reads and writes to it behave as if they were sequential. "In other words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by reads are the same as if the operations had been performed in that order, with no overlapping." [Lam86]. This is another definition for an ACM. In his work, Lamport showed that any construction better than 'safe' required two way communication between the two communicating processes.

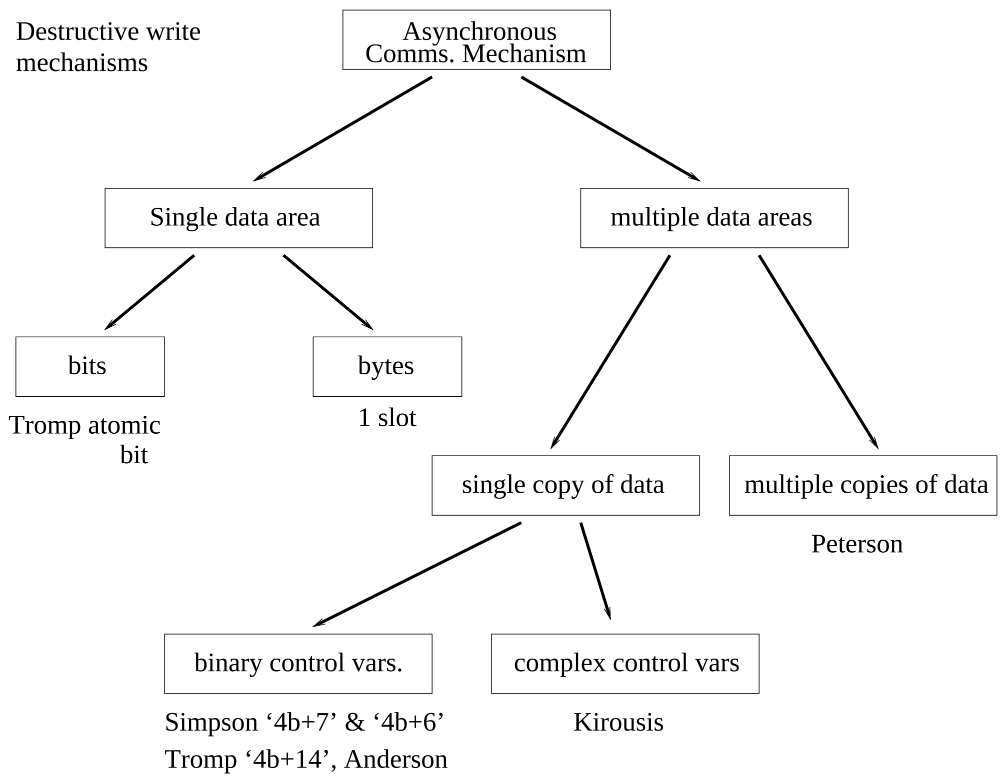


Figure 4.2: Destructive write asynchronous taxonomy.

So far, the requirements for ACM construction are destructive write operations, multiple data slots and a control or steering algorithm. Figure 4.2 shows how ACMs can be roughly classified. Mechanisms which use only one data slot (the left branch of the tree), are only of use in constructing single bit variables such as Tromp's atomic bit [Tro89]. The 1 slot (a 'slot' is used to decribe a register large enough to hold one complete piece of data which is being transmitted) is mentioned here for completeness but as will be shown later in this chapter, it fails very simply. The right branch of the tree is used to show ACMs with multiple data slots. Some keep single copies of data i.e. only write the data into one slot, others keep multiple copies i.e. they write the same data into several slots, the reader then reads the same slots and decides which copy of the data to use. Taking the breakdown one stage further, multiple slot ACMs require a control or steering algorithm, which will require the use of control variables. These can be binary, in the case of Simpson, or complex as in the case of Kirousis.

The next sections define how the ACM algorithms under test are modelled in Petri nets and analyzed, what operating requirements are tested and analyzed, and an explanation of why 4 slots seem to be required, and why 3 slots is not enough.

4.1.2 Open and closed access

Access to an ACM refers to how data is written and read to and from an ACM. Open and closed refers to two different access timing methods. The two communicating process using the ACM will be producing and consuming the data they communicate on top of the actual communication. The cycle of the writing process is *produce data*, *communicate data*, *produce more data* etc. There are two options for the writer here. As the data is produced it can write the data straight to the mechanism, this will take the same period of time that it takes to produce the data. Alternatively the writer could keep a local copy of the data being produced and then write the data into the ACM in one go. These options are called *open access* and *closed access* respectively. *Open access* because the mechanism is left open while the data is produced, and *closed access* because the mechanism is kept closed while the data is produced and only opened when the complete data item is ready to be written. This is shown in Figure 4.3.

When modelling ACMs, if closed access is assumed, then both open and closed

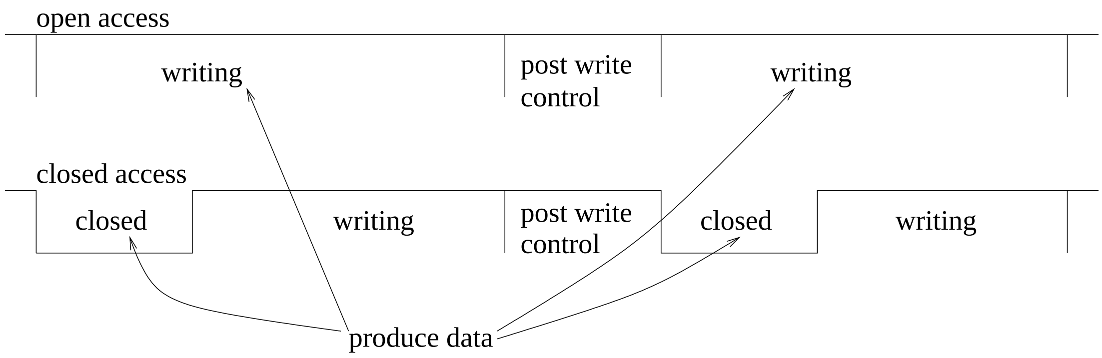


Figure 4.3: Time line representation of open and closed access.

access are checked because open access is equivalent to closed access with a closed period of 0. However modelling open access access does not cover the closed access case.

4.2 Modelling techniques

It is sensible to make systems which are small enough to understand / analyze, then build these up into larger systems [Ran75].

In this section the concepts and ideas behind the Petri net models used for the analysis in this work is introduced.

4.2.1 Cyclic models

The ACMs being modelled for this work allow two single-thread processing elements to communicate asynchronously. These processing elements usually execute the same code many times (i.e. they are non-terminating) and will also communicate many times: therefore we can model the system in Petri nets as a cycle, see Figure 4.4. The ACM models being cyclic in nature, and with no timing assumptions made about the interleaving of instructions from the accessing processes, require the use of reference arcs (self loops - Figure 2.17) in the Petri net models. If tokens were produced and consumed, then the model may inadvertently block part of the model from executing correctly, and therefore giving false results. By using reference arcs, shared variable values can be interrogated at any time, as with variables in hardware.

As the Petri net model is extended to model more complex ACM algorithms

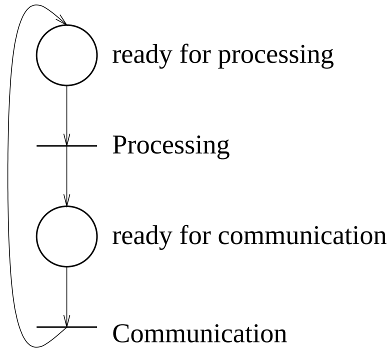


Figure 4.4: A cyclic Petri net model of processing and communication.

which include multiple instructions, the single 'communication' transition is replaced by more transitions and places. It is also found that the modelling of the processing part of the processing element is not necessary. The processing does not interact with any of the ACM internal variables, and can therefore be removed from the model, reducing the number of system states. This is shown in Figure 4.5.

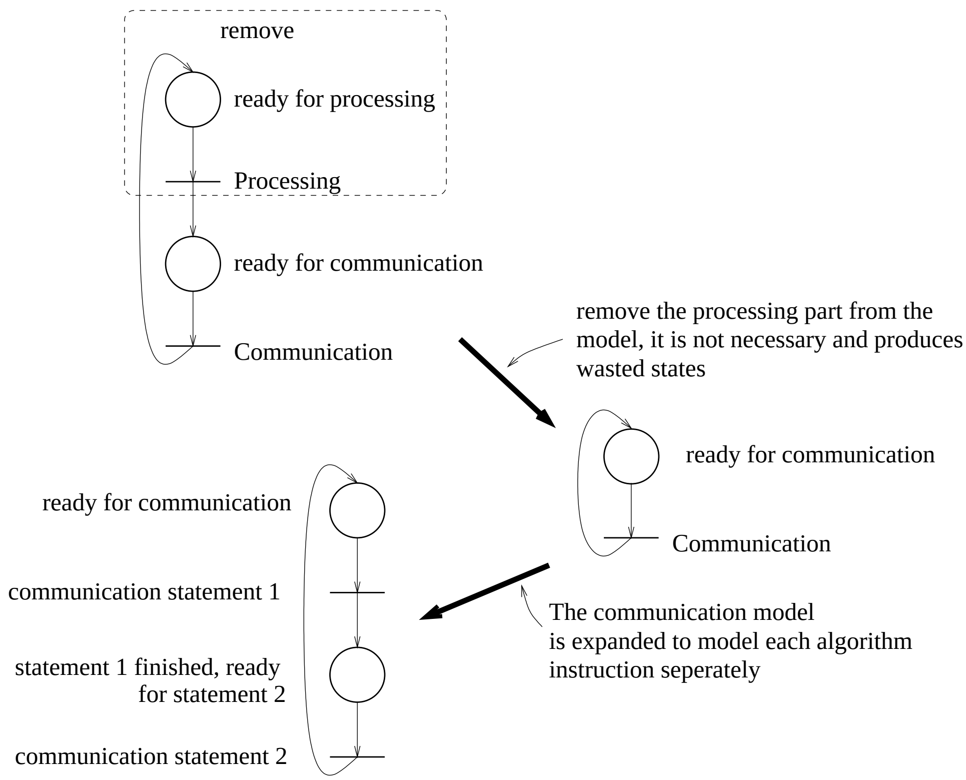


Figure 4.5: Complex cyclic Petri net model with processing element removed.

The Petri net places in Figure 4.5 are used to model control variables, which are discussed later in this section.

4.2.2 Safe Petri nets

In classical Petri nets a place may hold zero or any positive integer number of tokens. A safe Petri net is a net in which places can only hold either 0 or 1 token. Only safe Petri nets are used in this thesis, because the states being modelled are in complementary loops. For example in modelling a binary variable, there are two states, '$variable = 0$' or '$variable = 1$', which are sometimes referred to as '$variable$' or '$\overline{variable}$'. This is the same for the status of a part of the ACM under analysis, such as '$slot\_00\_writing$' or '$\overline{slot\_00\_writing}$', which are the only two valid states for this. This aids in fault finding because the two places concerned will never contain 0 or 2 tokens, only ever 1 between them. The same technique is also used for the control variables which are discussed later in this section, where only one token will ever be present in the control variable loop of a communicating process.

The models being constructed and analyzed are operating between two concurrent asynchronous processes, with no time conditions enforced upon operation. This implies that any variable which is shared between these communicating processes must always be available for reference. Therefore production and consumption of such variables or tokens is not acceptable as this could incur waiting in the model which is not an accurate representation of the real mechanism in physical hardware. Reference arcs are used for variable referencing, and are introduced later in this section.

4.2.3 Abstract models or individual instruction models

The ACMs under analysis comprise a collection of individual instructions which when executed sequentially by the appropriate process, allow the communication of reference data. The problem is how best to approach the production of models and what should actually be modelled? The available options are 'abstract' or 'individual instruction'.

In each algorithm there will be one or more instructions which carry out a specific function, for example a group of instructions may be required to perform a defined task such as 'indicate a specific condition'. Two different models can be constructed, one represents this as a single event 'indicate condition', the second models each individual instruction as separate events. Both methods are equally valid if correct models are constructed.

The analysis in this thesis is focused on ACM algorithms published by several different authors, each has been designed and analyzed using different methods. It was therefore not practical to construct abstract models of these different mechanisms and then to use these models for comparison purposes. Abstract models although smaller and simpler are not always obvious and can require extended time to construct accurately. Also if a designer comes back to an old design the time taken to relate the model to the algorithm can be longer. If each individual instruction is directly mapped onto a model then the model may be more complex and produce a larger state space, but the relationship between algorithm and model is clearer and simpler to see and understand. Therefore individual instruction models have been used in this thesis.

Simpson's role models and analysis [Sim92a] are an abstract modelling method. The method relates the algorithm to higher level events of the system, i.e. 'choose slot' and 'indicate slot'. In the analysis of operating properties of ACMs such abstractions can aid the analysis, in giving a higher level and often clearer picture of the test. This is certainly the case in the analysis of data freshness, where the abstract model results in a more compact description, which is described later in this chapter. Role modelling is similar to state space searching with the addition of pattern matching to reduce the state space. The pattern matching means models do not have to model each slot individually but can match any slot with a particular role.

However the lower level individual instruction models allow for the rapid analysis of metastability of individual variables, stuck at faults in control logic and instruction order errors. Instruction level models also help in the validation of model correctness. Each individual instruction can be checked to verify that it is an accurate representation of the operator it models. Then as each checked instruction model is put together into the larger model of the algorithm under test, its analysis can accurately represent the instruction interleaving between the processes.

The trade off between abstract models and individual instruction models is that of computability, i.e. state space size, human readability and human error.

4.2.4 Control variables

The Petri net models constructed for ACM analysis use control variables which ensure sequential operation of the individual instructions. Each communicating process has a sequential list of instructions which are used to perform correct access to the mechanism. Each instruction within the process is labeled with a unique label, for example in Simpson's first algorithm [Sim90b] the writing process has three instructions which are subsequently labeled WR, W0 and W1. The label WR refers to the writing of data into the mechanism, W0 and W1 refer to the post write sequence of indicating where the last piece of data was written. Whenever enabled, Petri net transitions may fire, therefore several statements could execute when enabled but not necessarily in the desired order. Every transition related to the input of each instruction will have an arc from the relevant control variable, so that the transition is only enabled when the instruction is sequenced to execute. This implies that writing to and reading from the mechanism only happens at defined times, and therefore the ACM operating properties are checked at valid times. Likewise each output transition has an arc from itself to the control variable of the next instruction of the process. Control variables are also required because under certain conditions the firing of a transition may not change the state of the mechanism, therefore the reachability analysis would not continue to investigate that branch of the state space because it would have already been seen. This may not be the case, the models are of concurrent processes and therefore each variable must only be referenced, not produced and consumed which would imply waiting in a truly asynchronous environment. An example of this is an initial condition of all 0's, where an instruction may make one variable equal to another which would result in no change of state, see Figure 4.6.

Each process has a loop of control variables of which one and only one is marked. One exception to this is where an instruction is modelled using a two stage model, in which case an internal control variable to the instruction is marked, but this still forms part of the overall process control variable loop, see Figure 4.7.

Later in the thesis, Petri net models of individual instructions are presented. Each instruction has a starting control variable and a finishing control variable which enables the next statement. The control variables are modelled using Petri net places which are repeated for presentation clarity. For example, the model for W0 would

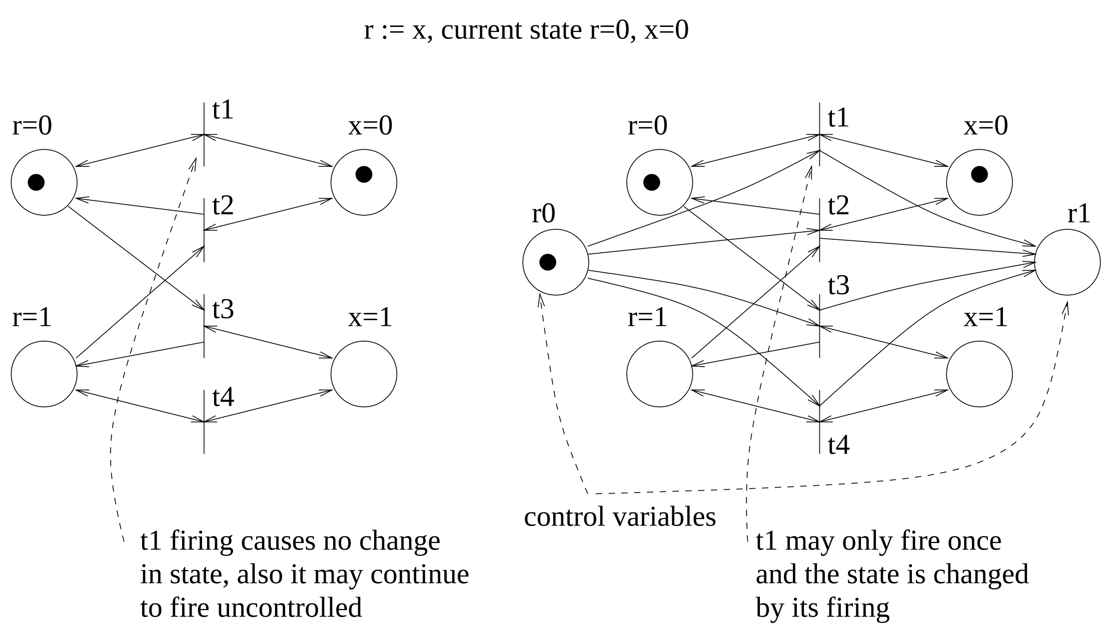


Figure 4.6: Instruction firing without changing state, except the control variable.

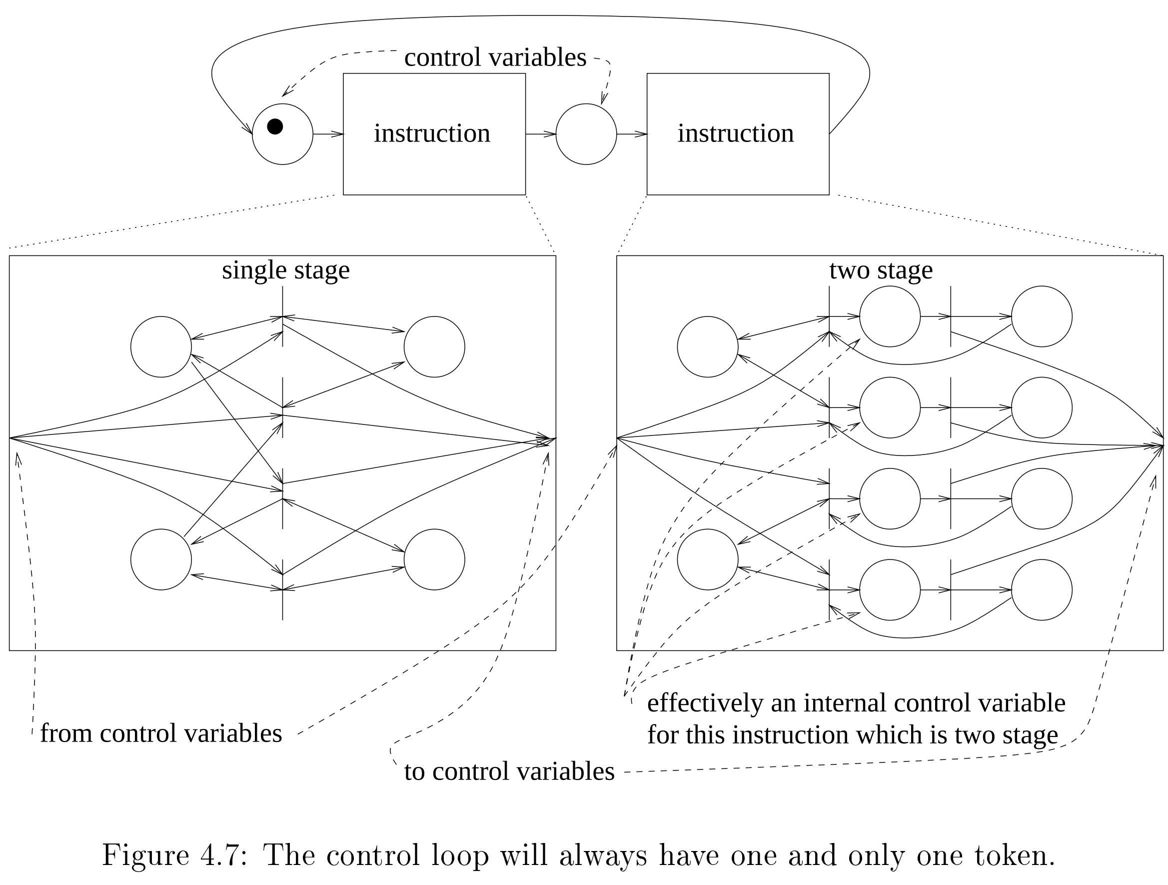


Figure 4.7: The control loop will always have one and only one token.

have the start control variable of W0 and the finishing control variable of W1. For simplicity of presentation on paper, the graphical model of W0 shows both W0 and W1, then the graphical model of W1 also shows W1 as the starting control place. This is the equivalent of a jigsaw puzzle with two matching pieces both having a peg and neither having a receptacle, see Figure 4.8.

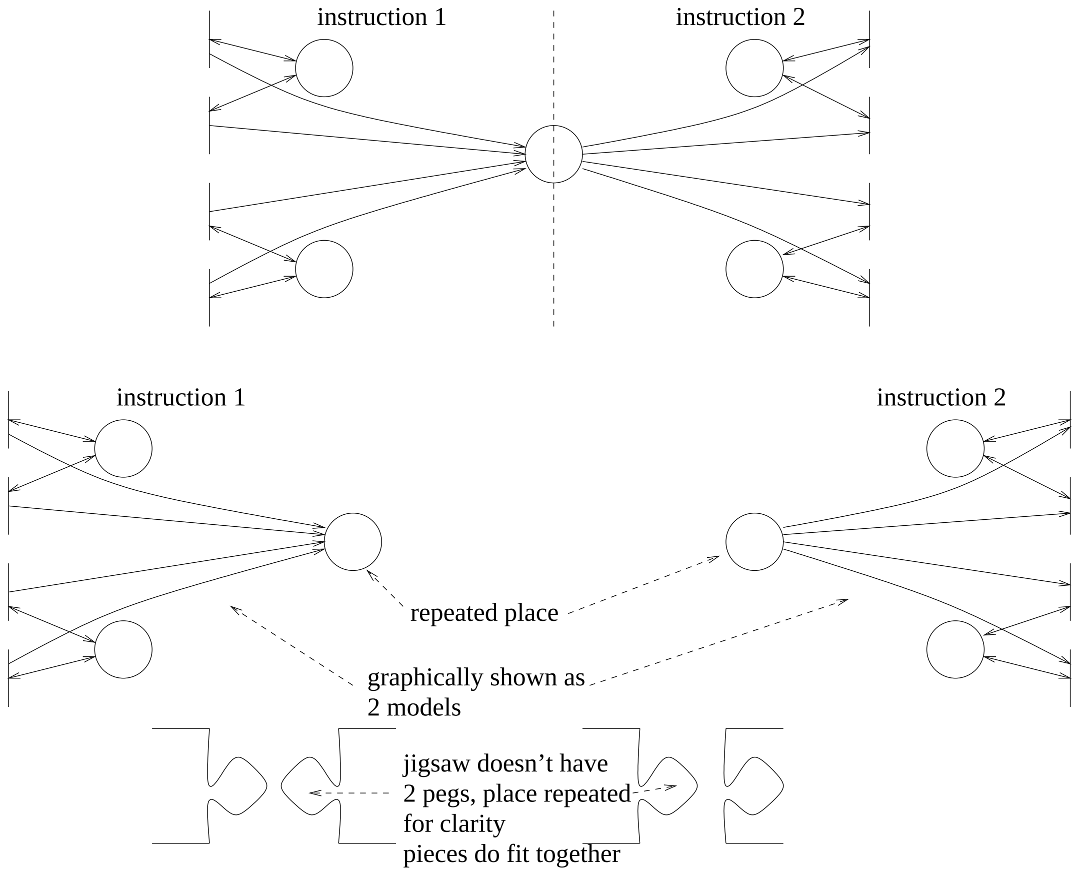


Figure 4.8: Places repeated for presentation clarity.

4.2.5 Assignment and logic operators

The assignment operator ($a = b$) is the simplest instruction used within an algorithm, and is used to make a copy of another variable or keep a result from another operation. The variable being copied may be local or remote (see modelling requirements for shared/remote variables in Chapter 3, and Section 2.4 on metastability). The ACM models are cyclic and no assumptions are made about different processes instruction interleaving, therefore reference arcs are used in the Petri net models so that the value of a variable being copied is not consumed and can be used by other instructions. Control variables, (subsection 4.2.4), also affect the way in which the assignment operation is modelled. For example the operation '$a := b$' requires four Petri net transitions, even though there are only two possible changes of state. The two extra transitions are required in the cases where '$b := 0$' and a is already equal to 0, and vice versa with '$b := 1$'. The control place token still needs to be advanced to enable the next instruction, see Figure 4.9.

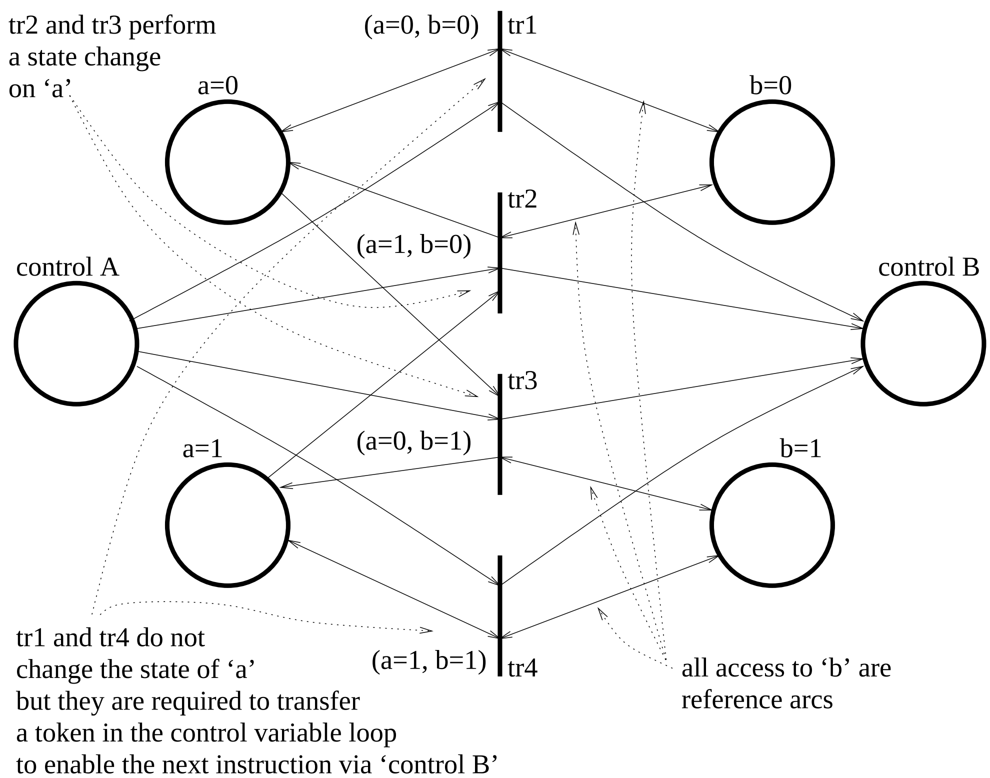


Figure 4.9: The assignment operator requires four transitions.

The NOT logical operator ($\overline{b}$) with an assignment is similar to Figure 4.9 except that the 'b' place labels are reversed, providing '$a := \overline{b}$'. However $b := \overline{b}$ only requires two transitions because there is no previous state of 'a' to be considered, and the current condition of 'b' is consumed. For the operators AND, OR and XOR, similar models are used but with eight transitions. '$a := b \wedge c$' requires eight transitions because the previous state of 'a' has to be consumed, (OR arcs, Section 4.2.9, would reduce this to four). For examples of these logical operators, see Figure 4.10. Textual models are good for logical operator models, as can be seen by comparing the graphical Petri net of Figure 4.10 with the textual Petri net of Figure 4.11.

It should be noted that Petri net models of these operators all seem to balance. Balance is achieved whenever the number of times a place label appears in the textual model is a power of two, i.e. $2, 4, 8$ etc. The textual model for Figure 4.9 for example

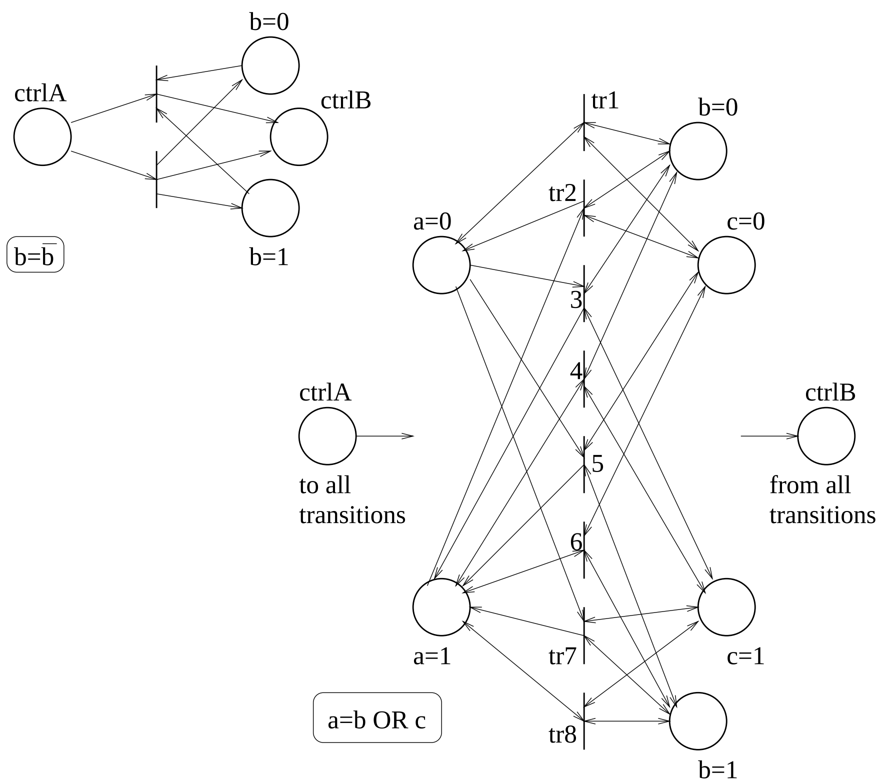


Figure 4.10: Logic operators modelled in Petri nets.

```
tr1,[ctrlA,a=0,b=0,c=0],[ctrlB,a=0,b=0,c=0]
tr2,[ctrlA,a=1,b=0,c=0],[ctrlB,a=0,b=0,c=0]
tr3,[ctrlA,a=0,b=0,c=1],[ctrlB,a=1,b=0,c=1]
tr4,[ctrlA,a=1,b=0,c=1],[ctrlB,a=1,b=0,c=1]
tr5,[ctrlA,a=0,b=1,c=0],[ctrlB,a=1,b=1,c=0]
tr6,[ctrlA,a=1,b=1,c=0],[ctrlB,a=1,b=1,c=0]
tr7,[ctrlA,a=0,b=1,c=1],[ctrlB,a=1,b=1,c=1]
tr8,[ctrlA,a=1,b=1,c=1],[ctrlB,a=1,b=1,c=1]
```

Figure 4.11: Textual model for '$a = b$ OR c'.

has four '*control A*', four '*control B*', four '$a := 0$', four '$a := 1$', four '$b := 0$' and four '$b := 1$'. This can form the basis of a check to see if a model has a simple typing error in it, although this is not complete as two similar typos could cancel each other out. This balancing can also be seen with ternary variables.

4.2.6 The 'enable' place and atomic actions

In the Petri net models of ACMs it has already been stated that all possible instruction interleavings should be investigated for accurate analysis, but in the case of checking for operating properties, there are some extra states which need to be included in the models. For example in testing for data freshness (Section 4.4.3), extra states are added at the same instant as another instruction begins or finishes. This can be included in the model by changing the existing model to include such operations (which has the effect of making the model far more complex and more difficult to follow) or by the inclusion of an 'enable' place, with extra states added as transitions. The 'enable' place is marked with one token in the initial condition, and all existing model transitions should be made to reference it. Then, where an extra part is to added into the model, the 'enable' place can be temporarily removed, and then replaced when the operation is complete. The operations which have then been inserted look like an atomic operation to the other process, and therefore do not affect the analysis of the model by changing the interleaving characteristics. Figure 4.12 shows the insertion of an 'enable' place between two processes.

Making more than one instruction seem atomic to the other process is safe. "When considering communication and synchronization properties of concurrent systems, the only operation executions that are of interest are the ones that involve interprocess communication - for example, the operations of sending a message or reading a shared variable. Internal 'calculation' steps can be ignored" [Lam86]. The 'enable' place provides for a mutual exclusion for time in the model, and uses a similar model to parallel mutual exclusion [DA95].

When the enable place is removed by a process to allow multiple instructions to become atomic, the other process is not made to wait or synchronize, it is simply paused. The interleaving characteristics of the algorithm under test are not affected.

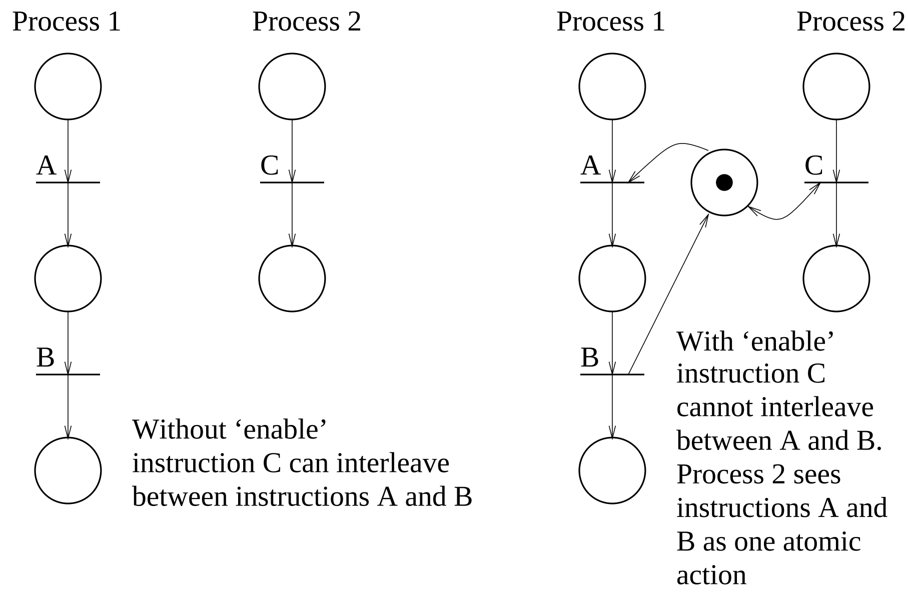


Figure 4.12: Example of how an 'enable' place is inserted to facilitate atomic actions.

The 'enable' place can also aid memory usage in the storage of the state space, see Section 4.3.2. The internal actions within an atomic instruction are not of interest to the analysis, therefore only states which contain the enable place need be stored.

4.2.7 A state and its complement

The models in this thesis use Petri net places in complementary pairs. They are used to represent a state or variable value and its complement. In the case of variables this can be binary variables, i.e. 'a' and '$\overline{a}$', or for ternary variables or set-based variables, not just a value and its complement, but one value in its range or set of possible values. In the case of a state it can only be its value or its complement, because the state is either happening or it is not happening, e.g. a process can either be $writing$ or $\overline{writing}$.

The presence of data within an ACM mechanism is not modelled, just the action of writing data into the mechanism. If a data channel (Section 2.3.1) were being modelled then the presence of data would need to be modelled, because a reader cannot read from an empty channel and the model must reflect this property. In the case of ACMs, writing and reading data is always possible, so the data itself is not relevant for the model. The modelling of a state and its complement or one value from its range of values, again, leads to loops within the models (Section 4.2.1). This however permits another safety check, because at all times each loop should contain one and only one token. If a loop contains no tokens or more than one then an error has occurred.

An ACM algorithm can also be tested under partial failure conditions. This method can be used to introduce a 'stuck-at' fault (a stuck-at fault implies a hardware failure where a latch is permanently stuck in one state, either logic 0 or 1) in any control variable and analysis of the ACM algorithm will then indicate which of the ACM operating properties (Sections 4.4.2, 4.4.3 and 4.4.4) are affected in such operating circumstances. Such a failure condition should only be applied to control variables, and is modelled by only using one state from the variable state and its complement (e.g. $a = 0$ and $a = 1$).

4.2.8 'Bad' states

'Bad' states are states which indicate that an operating property being testing for has not been maintained. There is no need to have a bad state, because the states which cause the bad states to arise could be checked for, instead, however highlighting this bad state, makes it easier to spot in large state spaces. A typical example of a bad state is from the data coherence property 4.4.2, where a Petri net place is labelled 'coherence_lost'. This place is marked when the state of a single slot is concurrently being written to and read from. The bad states are not included in the 'visited list' of the program, and none of its children are found or investigated (i.e. it is not put on the 'to be visited list'). In the state space this would double the state space size because it would just be a copy of the existing state space with the added 'coherence_lost' place marked. This means that bad states are extra states on the side of the full state space. Although the bad state is not stored on the 'visited list', or the 'to be visited list', its parent which caused the bad state is stored and fully investigated, so all bad states are passed through to see if the failure causes other operational problems, or whether the mechanism will correct its operation during the following cycle of the algorithm. If a complete state space contains no markings with the bad place marked then the algorithm under test maintains the operating property which has been modelled. This is actually a checking of system invariants in the state space, where a property is invariably true under all possible executions of the system.

4.2.9 OR arcs

'OR' arcs have not been found in any of the published Petri net descriptions and are an extension to standard Petri nets invented by the author of this thesis. It was noticed in the construction of safe, cyclic Petri net models with reference arcs, multiple transitions were required to remove a token from a variables previous state. For example, in a model with a variable called 'just-written', (with possible values '1', '2', '3' & '4'), an instruction to make 'just-written' equal to '1' would require four transitions, one each for the four possible previous states of 'just-written'. An 'OR' reduces this to one transition by requiring a token from one and only one of the previous states. In textual Petri net representation an 'OR' arc is indicated by the use of '$+n$' preceding a place name, ('n' being an integer indicating the grouping of the set of 'OR' arcs), (see the example statement in Figure 4.13). Graphically 'OR' arcs are shown as Petri net arcs with a dashed line rather than the standard solid line, and linked using another dashed line indicating the set, (see Figure 4.70). Multiple sets of 'OR' arcs can be used, which is when 'OR' arcs provide the best model reduction, and this can reduce human error in the construction of large, complex models.

```
w1111,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=1],[w0,latest=1]
w1212,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=2],[w0,latest=2]
w1313,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=3],[w0,latest=3]
```

Figure 4.13: Example textual Petri net with OR arcs

4.2.10 Inhibitor arcs, safe Petri nets, modelling a Turing machine and priority

It has been noted by several authors [Hol91, Pet81a] that Petri nets are incomplete. They lack the ability to stop an enabled transition, (a place which is marked stopping the firing of a transition). This implies that Petri nets in their classical form cannot model a Turing machine [Tur36]. If a modelling and analysis technique is unable to model a Turing machine then it is unable to accurately model computer hardware. Inhibitor arcs have been included in 'reach' (Section 4.3), but are unnecessary because safe Petri nets using complementary places can model a Turing machine without inhibitor arcs [EN94]. This is only true for safe Petri nets and not for finite Petri nets.

Priority is another phenomenon classical Petri nets cannot model [AF73, Kos73, Pet81a] and is therefore considered a limitation. Agerwala and Flynn [AF73], also Kosaraju [Kos73] defined a solution to the priority channel problem to highlight this limitation of basic Petri nets. In Figure 4.14 the channel $b1$ has priority over $b2$. The equivalent Petri net model using complementary places is given in Figure 4.15. It should be noted that they are only equivalent if the channel is to handle single tokens i.e. only one piece of data. The inhibitor arc version is capable of handling multiple tokens whereas the complementary place version is not. The complementary place model can be altered to handle multiple tokens but the solution is not elegant, see Figure 4.16. This solution is also bounded, where the inhibitor arc model is not, this may however not be a problem as hardware is physically bounded. (Bounded means there is a limit to the number of tokens a place can hold, or areas in hardware where data can be stored).

Both these examples highlight that safe Petri nets are capable of accurately modelling computer hardware and priority.

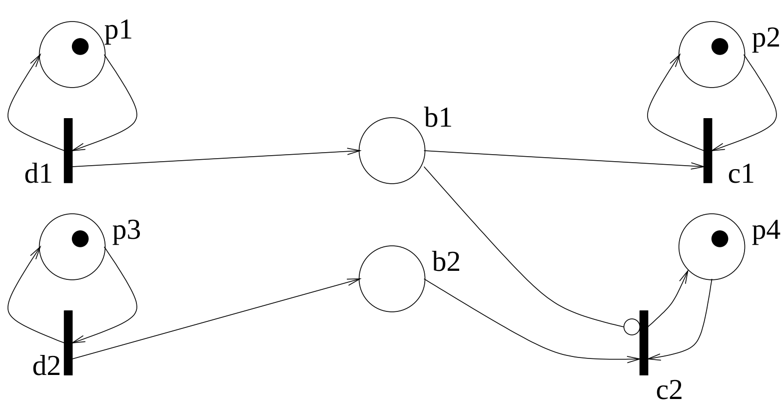


Figure 4.14: A priority shared channel implemented with an inhibitor arc.

4.2.11 Modelling assumptions

The models used in this thesis are based on some assumptions about their environment. These are discussed here.

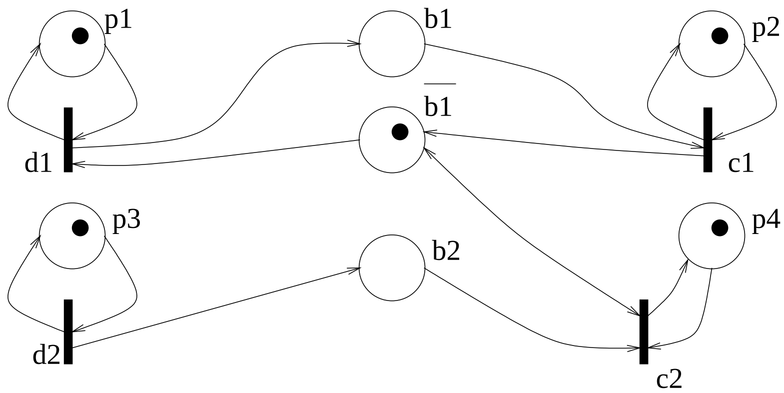


Figure 4.15: A priority shared channel implemented with complementary places.

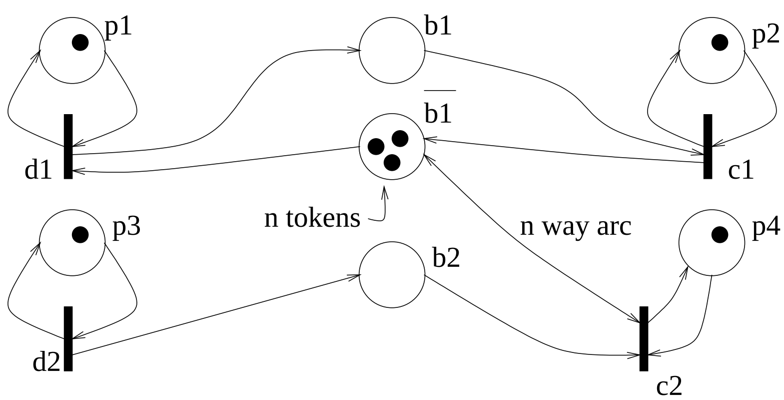


Figure 4.16: A bounded priority shared channel implemented with complementary places.

It is assumed that the operating environment produces and supplies the ACM with correct control pulses. This means that the pulses are of a size that causes proper triggering of the circuitry, i.e. they are not too short, and also their level is sufficient (this includes no metastable control pulses). The models can be extended to include such characteristics if necessary during the testing, but for the initial models which are used to test the basic operating properties, this is not included. It is also assumed that the control pulses are supplied in the correct order, and not in a reversed sequence. During the course of testing the model can be modified to test the algorithm with incorrect control pulses if required.

The models included in Appendix B show the initial conditions as specified in the literature by the publishing author. If no initial conditions are specified then a random initial condition is chosen. For complete analysis an ACM algorithm should be tested under all possible starting conditions, where necessary this can be achieved by implementing a small initial Petri net which sets up the variables, and every possible initial condition combination can be checked.

4.3 The analysis program - 'Reach'

Reach is a Petri net reachability search program written by the author of this thesis which has been used in obtaining all the results presented in this thesis. When this work was initiated the cyclic nature of the ACM models either caused the available packages to stay in infinite loops trying to generate the state space, or the input to the analysis package was a basic graphical package which was very laborious and difficult when inputing large cyclic models. Graphical entry does not easily allow for partial model reuse, whereas textual input models as used by *reach* can easily be reused. The contextual arcs or reference arcs also caused problems to the packages which used matrices during their analyses, as already shown in subsection 2.5.2. More recent Petri net packages which are available are capable of analyzing these ACM models, but *reach* has many options which have been specifically written to highlight ACM state space features. A full list of *reach* run time options are described in Appendix A.

Reach has been optimized for the analysis of two process models, although other models can be analyzed, and expanding the memory usage optimizations (subsection 4.3.2) to models with more than two processes is trivial. The model input files used by *reach* are described in subsection 4.3.1, which follows the syntax used in Peterson [Pet81a].

Reach performs a breadth first search of the input model's state space. This is performed by generating all the child states from the state currently at the head of the 'to be visited' list: these new states are then appended to the end of this list, but only if they have not been seen before. The 'to be visited' list starts with one state, the initial state. When all the children have been generated for the current head of the list, that state is discarded. Another list called the 'visited' list is also kept to ensure that no state is investigated twice, and would not generate the minimum reachability tree, see subsection 2.5.2.

During the reachability search each generated state is checked to see if it is '*bad*', i.e. the state has the bad place marked with a token, if it is bad then an operating property has failed. In this condition the bad state needs to be indicated and also a Petri net transition firing list produced showing how to get to the bad state from the initial state. This is performed using the 'visited' list which stores the states already investigated and each state's parent(s).

During the construction of *reach* many test models were run which were checked manually to increase confidence that states were not being omitted from any state spaces. Independently Wlodek Zuberek [Zub96] had constructed a Petri net reachability package, and through a request on the Petri net email list [Pet], the author sent several models to Zuberek who independently analyzed them using his package. The results were then compared with the results obtained from *reach*, both agreed, thereby increasing the confidence in the results. *Reach* has also been used by users of the Petri net email list [Cla95].

Appendix A presents an indepth description of *reach* and basic Petri net model completeness checking.

4.3.1 Convention for Petri Net models in this thesis

The input file for 'reach', the reachability program described in section 4.3, is a textual description of the Petri net. A sample input file is shown in Figure 4.17. The beginning of the model file is the header, and contains between three and five lines. The first line of a *reach* input file is a model description, purely for identification. The second line of the input file always describes the place labels which are considered bad, i.e. they indicate that the Petri net has reached a state, the occurrence of which is considered of interest or importance for the investigation of properties of the model being analyzed. Each set of bad places, is listed within square brackets, each place label being comma separated, and each set being comma separated, with no white spaces allowed, and the line must begin with 'bad='. The first two lines are always the model label and the bad place list. An optional third line is the 'control' line. This line lists all the control variables (subsection 4.2.4) within the model so that the flow or sequence of the model can be checked for errors. The line starts with 'control=' and then lists each control variable enclosed within square brackets and comma separated. The optional fourth line is the 'mask' line. If memory optimization is to be used for the storage of the visited list generated by *reach* then a place mask is required as described in subsection 4.3.2. The line begins with 'mask=' and then contains a single list of place labels to be used for the mask, enclosed in square brackets and comma separated, again with no white spaces. The last line of the header is the initial marking for the Petri net. The line starts with 'initial=' and then lists the places which will be marked with one token for the initial condition. This set of places is contained within square brackets and is comma delimited, and again no white spaces.

```
1 slot software model
bad=[coh_lost]
initial=[nowrite,noread]
COM comments here after 'COM'
startw,[nowrite],[write]
endw,[write],[nowrite]
startr,[noread],[read]
endr,[read],[noread]
coh_lost,[write,read],[write,read,coh_lost]
END
Anything can go here after the END tag.
```

Figure 4.17: An example 'reach' input file.

The model itself is listed transition by transition. Each line has a transition label ending in a comma, then the input marking of the transition, which is a list of place labels comma separated, and enclosed in square brackets, and also ending with a comma. There can be white space(s) between the comma of the transition label and the start of the input marking. (White spaces can be *space* or *tab* characters). Lastly, is the output marking, again a list of place labels comma separated and enclosed in square brackets. This is similar in syntax to the models published in Peterson [Pet81a]. There are two additions to the normal Petri net place labels listed in *reach* format, they are to indicate 'OR' arcs and 'inhibitor' arcs. 'OR' arcs were introduced in subsection 4.2.9, and are written as place labels preceeded by '$+n$', the '+' indicating the 'OR' function and n indicating which 'OR' set it is with. For example in transitions input may be listed as '$[+0place1, +0place2, +1place3, +1place4]$', this would mean there are two OR sets $\{0, 1\}$ and would imply that for this transition to fire a token would be required in either $place1$ or $place2$, and also a token would be required in $place3$ or $place4$. The second addition is for 'inhibitor arcs' (subsection 4.2.10), the notation is for a place label to be preceeded by a '$-$' symbol, i.e. '$[place1, -place2]$'. Firing would require a token in $place1$ and no token in $place2$. If there were a token in $place2$ then the transition is inhibited from firing.

Comments can be included in the model part of the file by using 'COM' as the transition label. At the end of the label 'END' can be used, and then any extra notes may be included here. This means 'COM' and 'END' are reserved transition labels not to be used in any model analyzed using this program.

This textual format is used throughout the rest of this thesis to present the Petri net models used in the ACM analysis.

4.3.2 Memory optimization during reachability analysis with two or more concurrent processes

The state space explosion problem is well known in Petri Net reachability analysis [Pet81b] and several authors have published solutions to alleviate this problem by using non-exhaustive techniques with reasoning about the results. This however does not provide the exhaustive results which are required in many applications. The number of unique states a system has is directly related to the number of atomic stages in the model, the number of atomic stages which are concurrent, and the number of unique states at each stage. Therefore the size of the system being modelled and analyzed is limited by the memory resources available if the state space is to be stored. Requirements to model systems often exceed the resources available on even modern computers.

When using traditional state space exploration techniques with cyclic models, there is a need to store the entire state space for use as an 'already visited' list. This list can then be referenced to see which newly generated states have previously been explored and which need investigating further. This prevents the state space appearing to become infinite because of the generation of duplicate states.

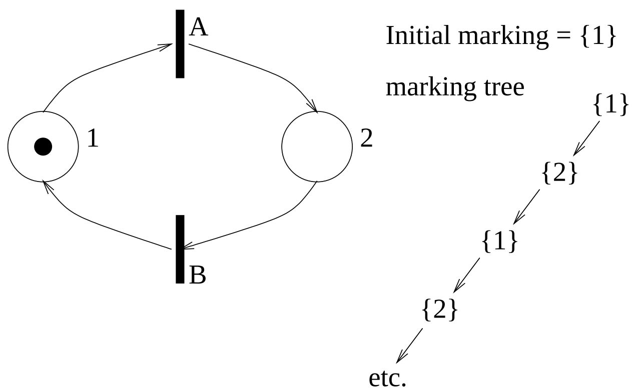


Figure 4.18: A small single process cyclic model.

An example of this is the cyclic single process model shown in Figure 4.18. The model contains only two transitions $\{A, B\}$, and two places $\{1, 2\}$. The initial marking, place $\{1\}$, has a token, and it can be seen from the marking tree that this model could produce an infinite state space due to duplicate states. If a visited list had been used then the tree would only have two leaves, the first two. When the second state $\{1\}$ had been produced it would already be on the visited list and therefore not need further investigation.

The model used here for comparison purposes is based on a model of an ACM which is presented later in this thesis. This model consists of two processes, a writing process and a reading process. Internally each of these processes is sequential, externally they are concurrent. The nature of these processes dictates that the models used for their analysis are cyclic.

The models used are safe, i.e. each Petri net place can only be marked with zero or one token. (A state in a Petri Net model is sometimes called a marking). However the techniques presented here could be extended to include non-safe models. The use of safe models allows the storage of individual states in binary form. A binary number of n bits, where n is equal to or greater than the number of places in the Petri Net model, is used for each state. Each place is allocated a bit position in this number and a '0' indicates no token, a '1' indicates a token is present and therefore the place is marked. The analysis method used during this work was reachability searching [Pet81b], which generates the exhaustive state space, while looking for states of interest.

The next few pages review program developments with regard to memory usage for the storage of state space lists.

The program used to perform the reachability search was written in C [KP84, KR88]. To perform run time variable size memory allocation the function 'MALLOC' [KP84] is required. It should be noted that each call of the MALLOC function carries with it a small but significant memory overhead. This overhead includes the size of the memory being allocated, its location plus other required internal details. It was found by investigation that the size of this overhead averages between 6 - 8 bytes in size. This is significant when the block of memory being allocated is small in size. However this information is required by the 'FREE' [KP84] function, which releases previously allocated memory.

Single linked list

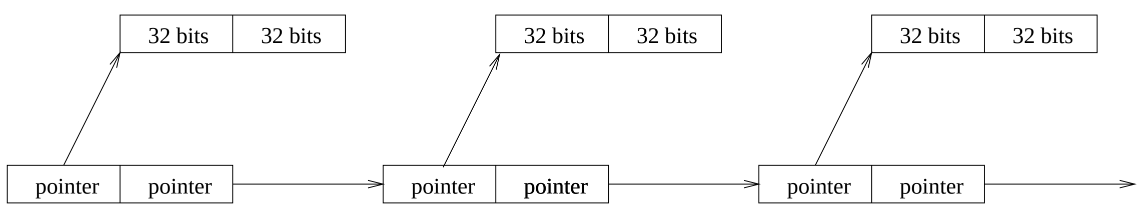


Figure 4.19: A simple linked list used for visited list storage.

The first attempt at storing the visited list was a simple linked list, the main elements containing two pointers, one to the marking, and the other to the next element of the list, see Figure 4.19. Each state requires two MALLOC calls, one for the main list element, and the other for the marking itself, which has its size dictated by the size of model being analyzed, and multiples of the executing computer word size, typically 32 bits. The example used here for comparison had 54 transitions and 39 places. The model produced 1472 unique states. The results used here for comparison (Table 4.2) are,

Memory used for data in the list	24KB
Number of MALLOCs	2944
Total memory used including MALLOC overhead	75KB

This example is small in size with only three statements in each of the processes. However when larger models are analyzed the optimization of state space storage can be vital.

The n depth, marked/not-marked tree

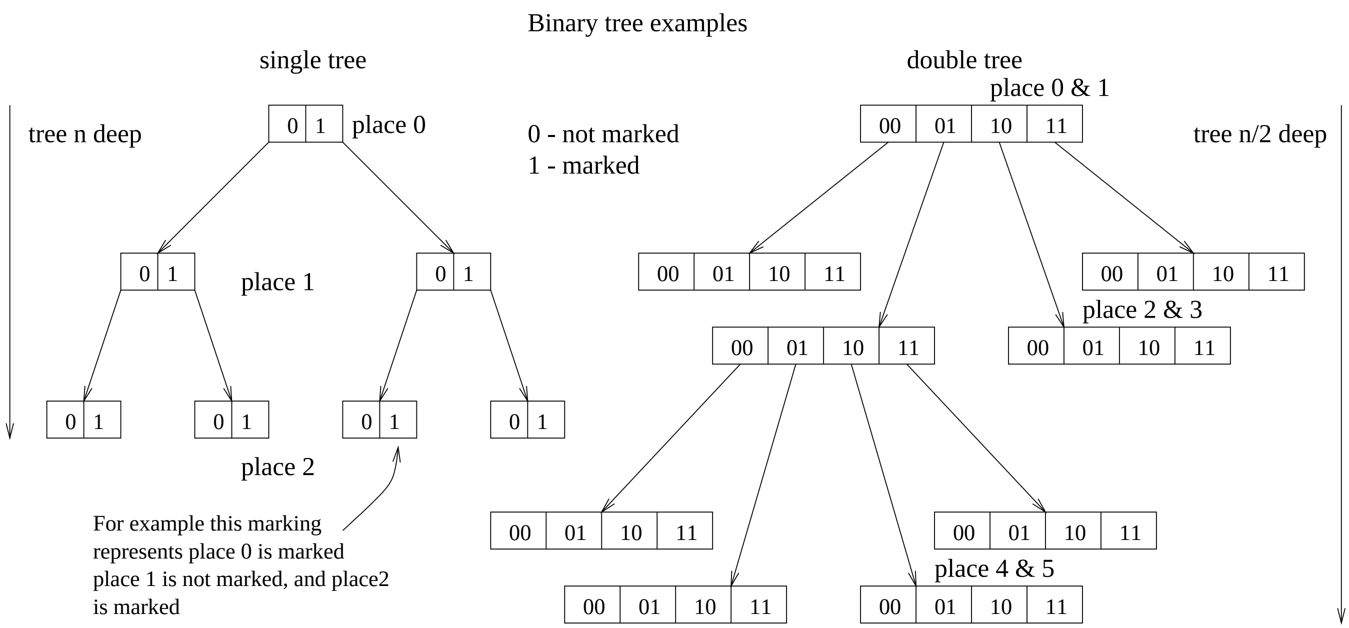


Figure 4.20: A second attempt using a tree of depth related to the model size, and generated by asking which places are marked or unmarked.

The second approach to this storage problem, and an attempt to improve memory usage, was a tree with depth equal to the number of places in the Petri Net, with each leaf of the tree directing the marking in two ways, marked or unmarked. This tree is made up from simple leaves of two pointers. The tree is read by looking at the root of the tree and determining if the first place is marked or unmarked, then following the tree down looking at each individual leaf with its corresponding Petri Net place. If the problem had been to store the complete state space, then memory improvements would have been seen in models with more than 96 places. Complete state space in this context means that a model with 8 places uses all $2^8 = 256$ states. However the state spaces here are not complete, typically they are only a small percentage of complete. Another attempt was to group places into pairs and ask "if places n and $n+1$ are marked?" etc, under complete state space conditions would have given improvements for models with only 1 place or more. But this was not the case in this application. Another optimization method ordered the places lists, either alphabetically or reverse-alphabetically. This provided some improvements, but the results were still worse than the single list presented above. Examples of both trees are shown in Figure 4.20 and results are given in Table 4.2. It can be seen from the results that this method gives considerably worse results for this type of problem, and has no compensatory advantages.

Interestingly, using double place leaves gives on average better savings than treble place leaves and single place leaves. Table 4.1 gives the ranges of possible memory savings compared to the standard linked list presented above. The left column indicates the number of places in the model and the remaining columns are for bits per leaf. The data indicates how efficient each leaf type is. For example a 90 place model using 2 bits per leaf would be between 1.5 and 3 times more efficient (less memory) than the single linked list if the complete state space were being stored.

Places	1 bit leaf	2 bit leaf	3 bit leaf
0 - 31	0.5	1.5 - 1.75	0.43 - 1.75
32 - 63	0.75	1.125 - 2.25	0.65 - 2.62
64 - 95	1.0	1.5 - 3	0.875 - 3.5
96 - 127	1.25	1.875 - 3.75	1.09 - 4.375
128 - 159	1.5	2.25 - 4.5	1.31 - 5.25
160 - 191	1.75	2.625 - 5.25	1.53 - 6.125
192 - 223	2.0	3 - 6	1.75 - 7

Table 4.1: Comparison of storage efficiency for $Marked/\overline{Marked}$ trees with a standard linked list.

Splitting the problem up into processes

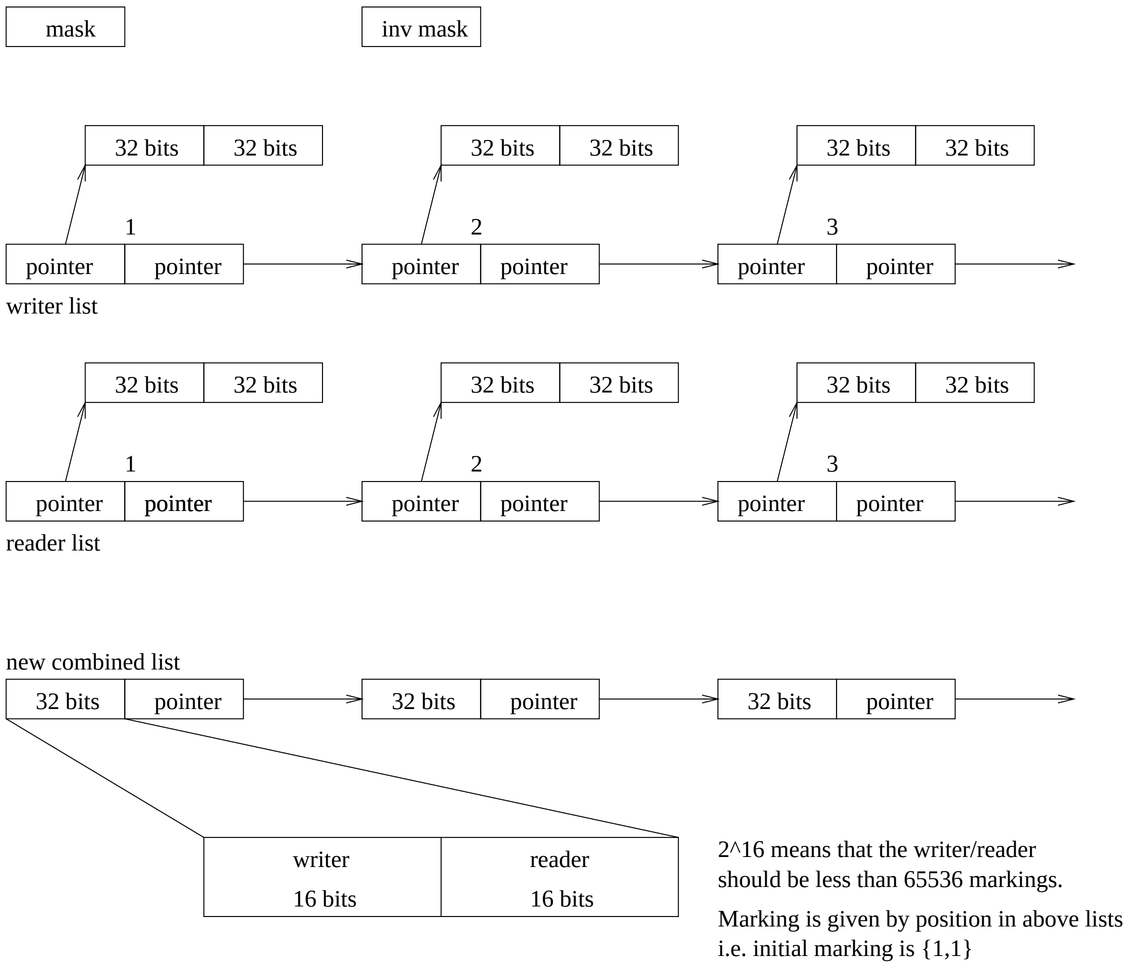


Figure 4.21: First attempt at splitting the problem into processes.

The first breakthrough in reducing storage came when the visited list was split into individual processes. The visited list is first split into two individual linked lists, one for the writing process and one for the reading process. This is done using a mask which contains the places which are controlled by one process, the inverse of this mask is then used for the other process. Again only unique markings are stored on the list. The example here produced 64 unique writer states and 48 unique reader states. The overall marking is then stored by using the marking position in each of the two lists. For example if the marking to be stored matched the 3rd element in the writing list and the 5th element in the reading list then this would be $\{3, 5\}$, see Figure 4.21. For small models two 16 bit words can be used, giving a range of 65536 states for each of the writer and the reader. If this is exceeded then 32 bits can be used, which will not be so easily exhausted.

This method gave a reasonable saving for this problem.

Memory used for data in the list	14KB
Number of MALLOCs	1698
Total memory used including MALLOC overhead	56KB

The memory usage saving for the list was 10KB, and a saving of 19KB with the MALLOC overhead included (Table 4.2).

The split list with 2D array for the full marking

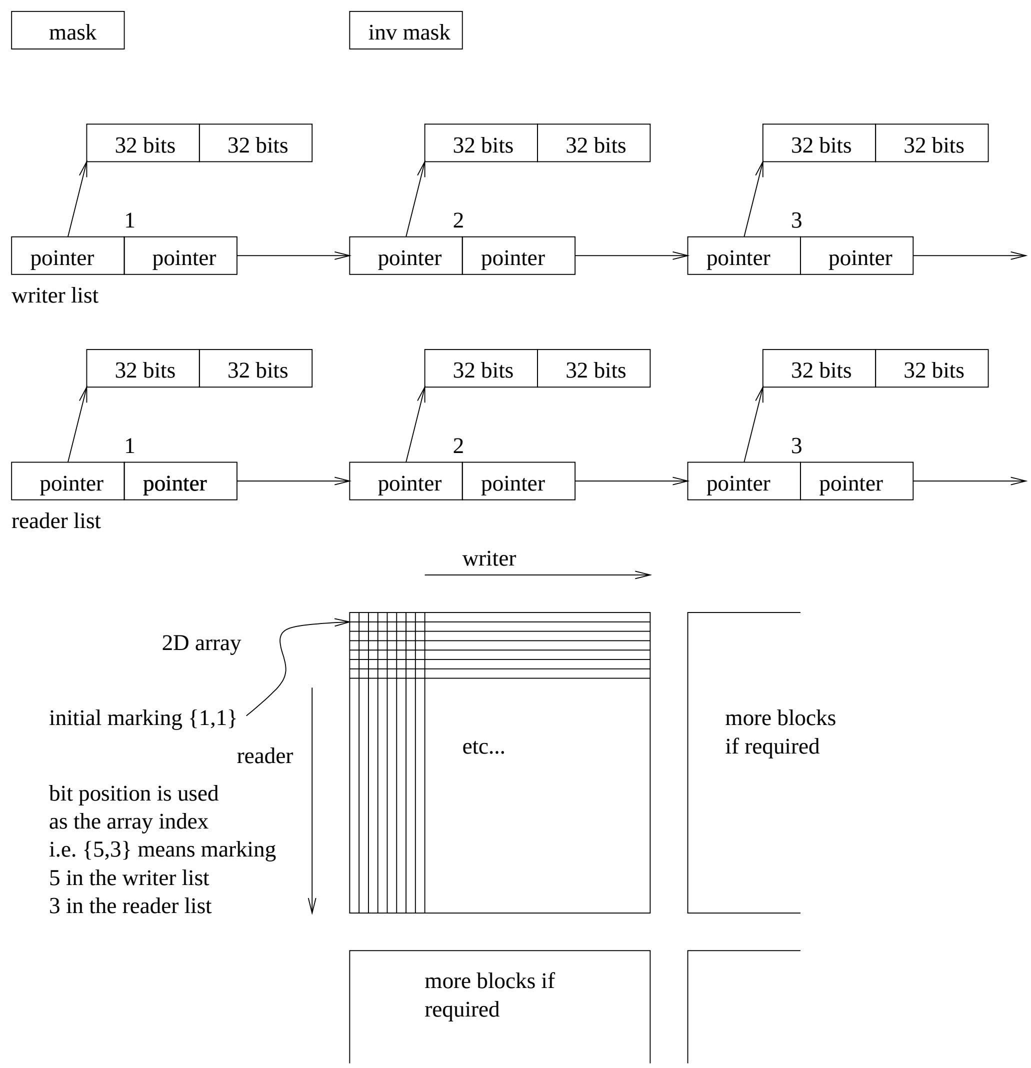


Figure 4.22: Second attempt at splitting the problem into processes, but with the full markings stored in a 2D array.

The most memory efficient method found used the split process lists from the above method, but rather than storing the full marking in a linked list, a simple two dimensional binary array was used. Again bit position in the process lists were used but this time for the array indices. The array starts off as 1 block, which is a 32 x 32 bit array, and as more states are found blocks are added in either the x or y directions depending on whether the new states are related to the writer or the reader. This is shown in Figure 4.22.

The memory used by this method was

Memory used for data in the list	3KB
Number of MALLOCs	230
Total memory used including MALLOC overhead	33KB

with the array being 2 x 2 blocks, or 64 x 64 bits.

A comparison of the different methods is presented in Table 4.2. The third column indicates the memory used by the visited list, not including the MALLOC overheads. The fourth column indicates the actual memory used at run time for the complete program including the visited list. The significance of the MALLOC overhead can be clearly seen from these Figures.

Algorithm	MALLOCs	State space	Total memory
Original	2944	24KB	75KB
$Marked/\overline{Marked}$ tree, single leaf (alpha)	14754	116KB	260KB
$Marked/\overline{Marked}$ tree, single leaf (reverse)	18405	144KB	317KB
$Marked/\overline{Marked}$ tree, double lf. (alpha)	8307	130KB	224KB
$Marked/\overline{Marked}$ tree, double lf. (reverse)	10092	158KB	266KB
Process split with linked list	1698	14KB	56KB
Process split with 2D array	230	3KB	33KB

Table 4.2: Comparisons between each algorithm presented, its memory usage and number of MALLOC calls.

The example used here was also safe and constructed of only two processes. The methods presented can easily be extended to non-safe models, i.e. ones with more than one token per Petri Net place, and also to models with greater than two processes. A more detailed account of this work has been published in a Technical Report associated with the 'COMFORT EPSRC Project' [Cla99].

4.4 Asynchronous communication mechanism properties

4.4.1 Role models

Role models were invented by Simpson for the purpose of analyzing his own ACM mechanisms [Sim90b, Sim97b], which are discussed later in Sections 5.6.1 and 5.6.2. Role models are a state space analysis technique, but at a higher level of abstraction than the hardware control variables used in the algorithms. Role models consist of three parts. Role model 'events' are used to signify when something happens and each one consists of three lower case letters. The first letter is 'r' for reader or 'w' for writer. The second letter is either 'a' for access, 'i' for indicate or 'c' for choose. The third letter depends on the second letter. If the second indicates an access then the third letter can be 'b' for beginning or 'e' for end. If the second letter indicated choose or indicate then the third letter specifies 's' for slot control variable or 'p' for pair control variable.

Role model 'roles' define what each slot is doing at a period of time. 'W' is used to show that a slot is written to, 'R' is used to show that a slot is being read from. '0' indicates that no role is currently allocated to a slot, or pair. A '*' indicates that any role can be present. The full list of defined roles can be found in Simpson [Sim90b, Sim97b].

Role model 'transitions' define how the model changes its 'roles' at defined 'events'. The layout of a role model transition for a system with two pairs of two bits each is shown in Figure 4.23. The role transitions for the algorithms are not reproduced here but their relationship to the individual algorithm instructions is given in Table 4.3 by using the event labels. The instructions 'wr' & 'rd' are the write and read access instructions, '$w0$' & '$w1$' are the post-write sequence, '$r0$' & '$r1$' are the pre-read sequence. Both sequences reference and/or update the control variables. It can be seen from Table 4.3 that in the first algorithm '$w0$' & '$r1$' operate on the slot-control variables, while '$w1$' & '$r0$' operate on the pair-control variables. In the second algorithm this is reversed with '$w1$' & '$r0$' operating on the slot-control variables, while '$w0$' & '$r1$' operate on the pair-control variables. An example role transition is given in Figure 4.24, showing the transition for '*rab*' which is for '*read access begin*'. It shows that the pair and slot which have been chosen for reading, get the 'R' indicating reading, and all 'C' roles are to be cleared.

The 4 slots are represented by b, c, e and f.

current state

next state

label: [{(a),b,c},{(d),e,f}] -> [{(a'),b',c'},{(d'),e',f'}]

a and d contain the roles for the slot pairs
(i.e. a could be for slots 00 and 01, or 10 and 11)

b and c contain the roles for individual slots
(i.e. b could be for slot 00 or 10)

Figure 4.23: Format of a role transition.

Inst.	First Algorithm	Second Algorithm
$wr:$	wab, wae	
$w0:$	wis	wip
$w1:$	wip, wcp	wcs, wis
$r0:$	rcp, rip	rcs, ris
$r1:$	rcs	rcp
$rd:$	rab, rae	

Table 4.3: Relationship between events and instructions.

$$rab : [\{(C), C, *\}, \{*\}] \rightarrow [\{(\overline{C}), \overline{C}R, *\}, \{(*), \overline{C}, \overline{C}\}]$$

Figure 4.24: '*rab*' role transition.

Once a complete set of 'events', 'roles' and 'transitions' have been defined for the model being analyzed, ACM operating properties (subsections 4.4.2, 4.4.3, and 4.4.4), need to be defined, modelled and then complete state spaces can be constructed looking for any states which show that the properties have not been maintained.

4.4.2 Data coherence

Data coherence is the first of several operating properties which an ACM is required to satisfy. The test for data coherence ensures that a writing process never accesses a slot at the same instant in time as a reading process. The specification of any ACM does not dictate details about the size of any slot within the mechanism, therefore it could range from a single byte to several megabytes (in typical applications this ranges from 1 byte to 4KB). The size of the slots is unimportant to the analysis of the ACM algorithm, what does matter is that the data written by the writer is read by the reader as a single block of data. If the writing process and reading process are both accessing the same slot at the same time, then the reading process may obtain some parts of the old message mixed with some parts of the new message, in any proportions. For example, if a sequence of time values were being communicated, such as ... $07:52$, $07:58$, $08:01$, $08:06$, $08:12$, ... then reading values such as $08:58$ and $07:06$ would correspond to incoherent data, and could be more seriously wrong than getting 'late' data such as $07:58$ when the time was really $08:01$. The accessing processes themselves should not be physically affected (blocked or short circuited) by such an operation if correct dual port memory is used within each slot.

The Petri net models for the writing and the reading process access to each slot are based upon the 1 slot model shown in Figure 4.42, but typically with reference made to shared and control variables for steering purposes in multi-slot algorithms. To check for data coherence, one transition per slot is required to test if at any time both the writing and reading places for a single slot are marked at the same time. If they are, then data coherence has been lost. The Petri net model for this is shown in Figure 4.25.

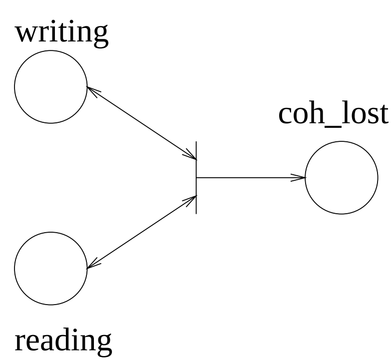


Figure 4.25: Petri net model for checking data coherence.

Care must be taken when checking for data coherence. The test described to check for data coherence only detects one slot at a time. However, in the case where the ACM algorithm uses shared variables as slot index variables the read or write data process may be moved mid access (see Section 6.1.5). The algorithm under test can be analyzed if it uses shared variables instead of local copies of the slot index variables by checking to make sure that any data access begins and end the access in the same data slot, as well as checking for concurrent access to the same data slot.

Care must also be taken when analyzing ACM algorithms which employ multiple copies of the same data to be transmitted, such as the algorithm from Peterson [Pet83]. This algorithm does not steer the writer away from the reader, but instead tries to sense when the writer and reader have accessed the same data slot, therefore possibly corrupting that copy of the data, and forces the reader process to use another copy of the data. In this case simply detecting that the writer and reader have accessed the same slot is insufficient, instead the Petri net model needs to sense when this happens but also to test that this corrupted copy is not the one which is returned from the reader process. This test is described in detail in Section 5.2.

4.4.3 Data freshness

Data freshness is another required operating property of ACMs, but is more complex than data coherence, in that it requires several extra states to be included in the model under test. Data freshness was first introduced by Simpson [Sim92a], and was modelled using *role models*, (subsection 4.4.1). An ACM could maintain data coherence by never passing any data between the two communication processes. Data freshness will flag up such faulty behaviour. The principle of role models and of modelling data freshness has been implemented in Petri nets [XC95, XCYD98, Xia00, XC00]. Here the definition of data freshness is first explained, using Simpson's role model terminology, followed by this author's modification to the method used in testing. Later the implementation in Petri nets is included in Chapter 5 where it is used in ACM analysis.

In [Sim92a], Simpson defines the loss of data freshness by posing a question: "Are there any circumstances in which stale data is obtained from the mechanism? Here staleness means that data obtained by a read operation is not the latest supplied by a write operation". Using the example of communicating time values, unfresh data would be any value returned which was not the last written and indicated value. For example the sequence of time values ... 07 : 52, 07 : 58, 08 : 01, 08 : 06, 08 : 12, written into the mechanism and inidicated, followed by a read access should return 08 : 12 any previous value is unfresh. The analysis for checking that data freshness is maintained requires the addition of extra roles 'F' (freshest), 'P' (previous freshest) & 'V' (valid). The 'F' & 'V' roles are assigned to the latest written data, but its location is not indicated until the post-write sequence ('$w0$' & '$w1$') updates the control variables, thus the 'P' role is assigned during this period to the slot previously carrying the 'F' role. At the end of the read access any slot not carrying the 'F' or 'P' roles (if 'P' exists) have the 'V' role cleared. Data freshness is maintained if the reader always reads data from a slot carrying the 'V' role. The role transitions for this first data freshness role-model (from [Sim92a]) are reproduced in Figure 4.26, (the *events* for the two algorithms were shown in Table 4.3). These freshness role transitions were updated for the analysis of the second communication algorithm [Sim97c], as shown in Figure 4.27. The updated version copies '$F \rightarrow f$', '$P \rightarrow p$', and clears all the 'V' roles at the beginning of the pre-read control sequence. Reading from 'p' is valid only if an overlapped pre-read and post-write sequence has occurred, reading from 'f' is always valid, and reading from a slot carrying the 'V' role is regarded as "better than fresh" ('f').

$$
\begin{array}{rlcl}
wae: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),P,*\},\{*\}] \\
 & [\{(*),W,*\},\{*\}] & \rightarrow & [\{(*),VF,\overline{F}\},\{(*),\overline{F},\overline{F}\}] \\
wab: & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}] \\
rae: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),V,\overline{V}\},\{(*),\overline{V},\overline{V}\}] \\
 & [\{(*),P,*\},\{*\}] & \rightarrow & [\{(*),V,*\},\{*\}]
\end{array}
$$

Figure 4.26: Simpson's first algorithm data freshness role transitions.

The role transitions given in Figures 4.26 and 4.27 were originally published to analyze the mechanism under open access operation, (subsection 4.1.2). This is indicated from the event labels used by Simpson. For example in Figure 4.26 the role transitions which clear all 'V' roles except for slots carrying 'F' and 'P' are given at 'rae' (read access end). If closed access were being analyzed the event label would have been 'rcp' which is at the beginning of the pre-read sequence, thus

$$
\begin{array}{rlcl}
wip: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),P,*\},\{*\}] \\
 & [\{(*),W,*\},\{*\}] & \rightarrow & [\{(*),VF,\overline{F}\},\{(*),\overline{F},\overline{F}\}] \\
wis: & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}] \\
rcs: & [\{(*),P,*\},\{*\}] & \rightarrow & [\{(*),p,\overline{p}\},\{(*),\overline{p},\overline{p}\}] \\
 & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),f,\overline{f}\},\{(*),\overline{f},\overline{f}\}] \\
 & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{V},\overline{V}\},\{(*),\overline{V},\overline{V}\}]
\end{array}
$$

Figure 4.27: Simpson's second algorithm data freshness role transitions.

allowing time for the previous data to be consumed. If closed mechanism access is modelled, and all possible instruction interleavings are investigated, then the case of open access is also included. This is because closed access with zero time for data production/consumption is equivalent in the state space to the open access case.

At first sight, since data freshness is defined as reading the latest available data, subsequent reading of earlier data should not be possible, and therefore it would appear that freshness implies correct data sequencing. This is however not the case, because the data is written into the mechanism and is then indicated to the other accessing process by a later instruction. Interleaving, destructive writing and a delayed data indication can cause a mechanism that maintains data freshness to fail data sequencing. It is easier to see that correct data sequencing does not imply freshness. Both versions of the algorithm have been claimed to maintain data freshness, but it has been reported [Sim97c] that the second fails data sequencing.

This anomaly appears to have arisen due to the freshness definition when the pre-read sequence starts during a post write sequence. From [Sim97c] fresh data "(in the case of a read pre sequence which starts during a write post sequence) was the previous freshest at the start of the read pre sequence". Also Simpson has reported that "there is some variability in the precise moment at which new data becomes available" [Sim97c], which allows fresh data to be read before the end of the post write sequence. The combination of this variability and a pre-read sequence starting during a post write sequence can cause a stale (non-fresh) data read to be masked. The published role transitions (Figures 4.26 & 4.27) do not to take this variability into account. If the reading process accesses a slot carrying the 'F' role during the post-write sequence, then the slot carrying the 'P' role becomes stale (non-fresh).

This is highlighted in Table 4.4, which shows an example execution of the second communication algorithm.

Instruction	pst-wr pre-rd		Slot roles 00	01	11
$wr : ip = 0, w[0] = 0$			$W\{FV\}[FV]$		
$wip : w0 : ip = 1$	w		$\{FV\}[FV]$		
$wcs, wis : w1 : w[1] = 1$	w		$\{FV\}[FV]$		
$rcs : r0a : r'[0] = 0, r'[1] = 1$		r	$\{FV\}[Ff]$		
$ris : r0b : r[0] = 0, r[1] = 1$		r	$\{FV\}[Ff]$		
$wr : ip = 1, w[1] = 1$		r	$\{PV\}[Pf]$		$W\{FV\}[FV]$
$wip : w0 : ip = 0$	w	r	$\{PV\}[Pf]$		$\{FV\}[FV]$
$rcp : r1 : op = 1$	w	r	$\{PV\}[Pf]$		$\{FV\}[FV]$
$rd : op = 1, r[1] = 1$	w		$\{(P)V\}[(P)f]$		$R\{FV\}[FV]$
$rcs : r0a : r'[0] = 0, r'[1] = 1$	w	r	$\{(PV)\}[(Pp)]$		$\{FV\}[Ff]$
$ris : r0b : r[0] = 0, r[1] = 1$	w	r	$\{(PV)\}[(Pp)]$		$\{FV\}[Ff]$
$wcs, wis : w1 : w[0] = 1$	w	r	$\{(PV)\}[(Pp)]$		$\{FV\}[Ff]$
$wr : ip = 0, w[0] = 1$		r	$\{(V)\}[(p)]$	$W\{FV\}[FV]$	$\{PV\}[Pf]$
$wip : w0 : ip = 1$	w	r	$\{(V)\}[(p)]$	$\{FV\}[FV]$	$\{PV\}[Pf]$
$rcp : r1 : op = 0$	w	r	$\{(V)\}[(p)]$	$\{FV\}[FV]$	$\{PV\}[Pf]$
$rd : op = 0, r[0] = 0$	w		$R\{(V)\}[(p)]$	$\{FV\}[FV]$	$\{PV\}[Pf]$

Table 4.4: An example execution sequence showing an invalid read (of non-fresh data) being masked. (Slot '10' has been omitted as it does not have any roles in this example run).

The first column in Table 4.4 is an ordered list of role model events and the instructions executed for this example, starting from an initial condition of all control variables set to logic '0' at the top. The first column also lists the control variable values when they are updated or referenced. The instruction '$r0$' has two operations '$r0a$' & '$r0b$', which mimic the 'choose' and 'indicate' events. This is the only instruction which requires a non-atomic assignment for all possible states to be investigated (see Chapter 3). The variable '$r'[0/1]$' is an internal temporary variable, which holds the next value of '$r[0/1]$' during the 'choose'/'indicate' period. The second column indicates the periods when the algorithm is in a post-write ('w') or a pre-read ('r') control sequence. The next block of three columns titled "Slot roles" indicate the mechanisms slots and the location of the freshness roles (slot '10' has been omitted as it is not used in this example). The roles for writing and reading are 'W' & 'R'. Roles from the first freshness model ('F', 'P' & 'V') are given in '{...}' brackets, and the roles from the second freshness model ('F', 'P', 'V', 'f' & 'p') are given in '[...]' brackets. Any roles inside '(...)' brackets indicate where the published role transitions leave the stale 'P' role and the assignment of extra roles in error. It can be seen from Table 4.4 that the first read (of slot 11) happens during a post-write sequence and it accesses the slot carrying the 'F' role. This is then followed by the start of a pre-read sequence ('$r0a$'), which re-assigns 'V' (assigns 'p') to the slot carrying the stale 'P' role (slot 00). This stale 'P' role causes the masking of the second read (of slot 00) from being highlighted as non-fresh (and out of sequence).

To solve this problem the role transitions need to be modified to clear the 'P' role (if 'P' exists) if the freshest data ('F') is read ('R'). Clearing the 'P' role would result in the '(...)' bracketed roles in Table 4.4 becoming empty, thus leaving the second read 'R' (slot 00) not carrying the 'V' ('p') role, and thus highlighting a non-fresh, out of sequence, data read. This modification adds an extra role transition to the existing transitions. The modified role transitions are given in Figures 4.28 & 4.29, and are slightly modified to analyze closed mechanism access. The time lines in Figure 4.30 show the modified role transitions more clearly. Analysis using the modified role transitions involves checking the existing roles for data freshness ('V' - first algorithm or 'p', 'f' & 'V' - second algorithm).

$$
\begin{array}{rlcl}
wae: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),P,*\},\{*\}] \\
 & [\{(*),W,*\},\{*\}] & \rightarrow & [\{(*),VF,\overline{F}\},\{(*),\overline{F},\overline{F}\}] \\
wab: & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}] \\
rab: & [\{(*),RF,*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}] \\
rcp: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),V,\overline{V}\},\{(*),\overline{V},\overline{V}\}] \\
 & [\{(*),P,*\},\{*\}] & \rightarrow & [\{(*),V,*\},\{*\}]
\end{array}
$$

Figure 4.28: Modified first data freshness role transitions.

$$
\begin{array}{rlcl}
wip: & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),P,*\},\{*\}] \\
 & [\{(*),W,*\},\{*\}] & \rightarrow & [\{(*),VF,\overline{F}\},\{(*),\overline{F},\overline{F}\}] \\
wis: & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}] \\
rcs: & [\{(*),P,*\},\{*\}] & \rightarrow & [\{(*),p,\overline{p}\},\{(*),\overline{p},\overline{p}\}] \\
 & [\{(*),F,*\},\{*\}] & \rightarrow & [\{(*),f,\overline{f}\},\{(*),\overline{f},\overline{f}\}] \\
 & [\{*\},\{*\}] & \rightarrow & [\{(*),\overline{V},\overline{V}\},\{(*),\overline{V},\overline{V}\}] \\
rcp: & [\{(*),RF,*\},\{*\}] & \rightarrow & [\{(*),\overline{P},\overline{P}\},\{(*),\overline{P},\overline{P}\}]
\end{array}
$$

Figure 4.29: Modified second data freshness role transitions.

According to an unpublished communication between Simpson and this author, Simpson suggests that this modification means that the freshness test now checks for data freshest-ness, and is not a valid test for data freshness. This is because the data

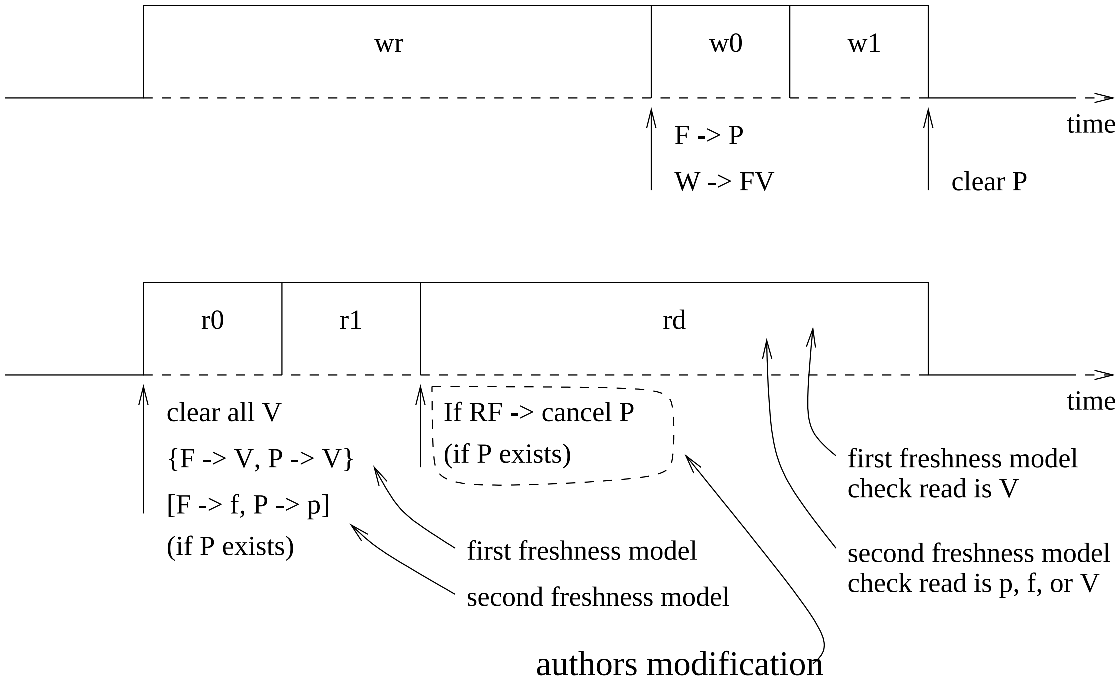


Figure 4.30: Time line showing the modified first data freshness model states.

being read has not yet been indicated therefore the previously written and indicated data is still classed as fresh and therefore valid by Simpson.

The implementation of this test in Petri nets will be presented in Chapter 5 where it is used for the analysis of ACM data freshness. As with any test which is appended onto an existing model it is necessary to check that the operation of the original underlying model under test is unaffected.

One minor modification to observe during the modelling process is that any read before a complete write (a write and indicate where the write occurred) will produce an invalid freshness violation. The Petri net model can be modified to only test for freshness after one complete write has been executed.

This test for data freshness will need modification if the ACM algorithm under test uses multiple copies of the data to be transmitted, such as the algorithm published by Peterson [Pet83] (Section 5.2). In this case all writes of the same current data would be regarded as fresh.

4.4.4 Data sequencing

Data sequencing is concerned with the order of the data which is read from the ACM under test. Data re-reading i.e. reading the same piece of data two or more times is acceptable, and data loss, where data is written but superseded and never read is also acceptable behaviour. The behaviour which is not acceptable is reading a new piece of data, and then reading an older piece of data. The roles in Figure 4.31 perform a data sequence test.

An error in data sequencing or as it is sometimes known the '$new - old - new$' phenomenon, can occur when a new piece of data has been read before it has been properly indicated by the post-write sequence. With variables being set to indicate the pair and slot in which the new data has been written, the precise moment at which the new data is indicated to the writing process can vary depending on which slot has been written to. Using the time values example again, reading the values $08:01$, $07:58$, $08:01$, $08:12$, would correspond to out of sequence data. Simpson [Sim97c] introduced a role model for checking data sequencing using the added roles of '1', '2', '3' and '4' to indicate the order in which the data in the ACM has been written, '1' being the most recently written. The role 'X' is used to indicate that a slot contains old data, i.e. newer data has already been read by the reading process. The role 'x' highlights when '$new - old - new$' has occurred. The data sequencing role transitions are given in Figure 4.31. The first three wip transitions allocate 1 to the latest written and increment all the other numbers. The fourth transition makes sure any new write is not highlighted as old. The first transition on rcp highlights a $new - old - new$ read has occurred. The remaining transitions highlight older data depending on which data has been chosen for the next read access. The transition at rcs clears any highlighted $new - old - new$ accesses. This is shown in Figure 4.32.

Simpson [Sim97c] used these roles to prove that his $4b + 6$ algorithm had a data sequencing problem, and this is confirmed later in Section 5.6.2.

This test for data sequencing requires minor modification similar to the freshness test above. This is to make sure data has been written into the mechanism before any reads are tested, and also for algorithms which use multiple copies of the same data such as that by Peterson [Pet83] (Section 5.2).

$$
\begin{array}{rlcl}
wip: & [W2,1,*,*] & \rightarrow & [1\overline{2},2\overline{1},*,*] \\
 & [W3,1,2,*] & \rightarrow & [1\overline{3},2\overline{1},3\overline{2},*] \\
 & [W4,1,2,3] & \rightarrow & [1\overline{4},2\overline{1},3\overline{2},4\overline{3}] \\
 & [WX,*,*,*] & \rightarrow & [\overline{X},*,*,*] \\
rcp: & [XR,*,*,*] & \rightarrow & [x,*,*,*] \\
 & [R1,*,*,*] & \rightarrow & [\overline{X},X,X,X] \\
 & [R2,1,*,*] & \rightarrow & [\overline{X},\overline{X},X,X] \\
 & [R3,1,2,*] & \rightarrow & [\overline{X},\overline{X},\overline{X},X] \\
 & [R4,1,2,3] & \rightarrow & [\overline{X},\overline{X},\overline{X},\overline{X}] \\
rcs: & [*,*,*,*] & \rightarrow & [\overline{x},\overline{x},\overline{x},\overline{x}]
\end{array}
$$

Figure 4.31: Data sequencing role transitions.

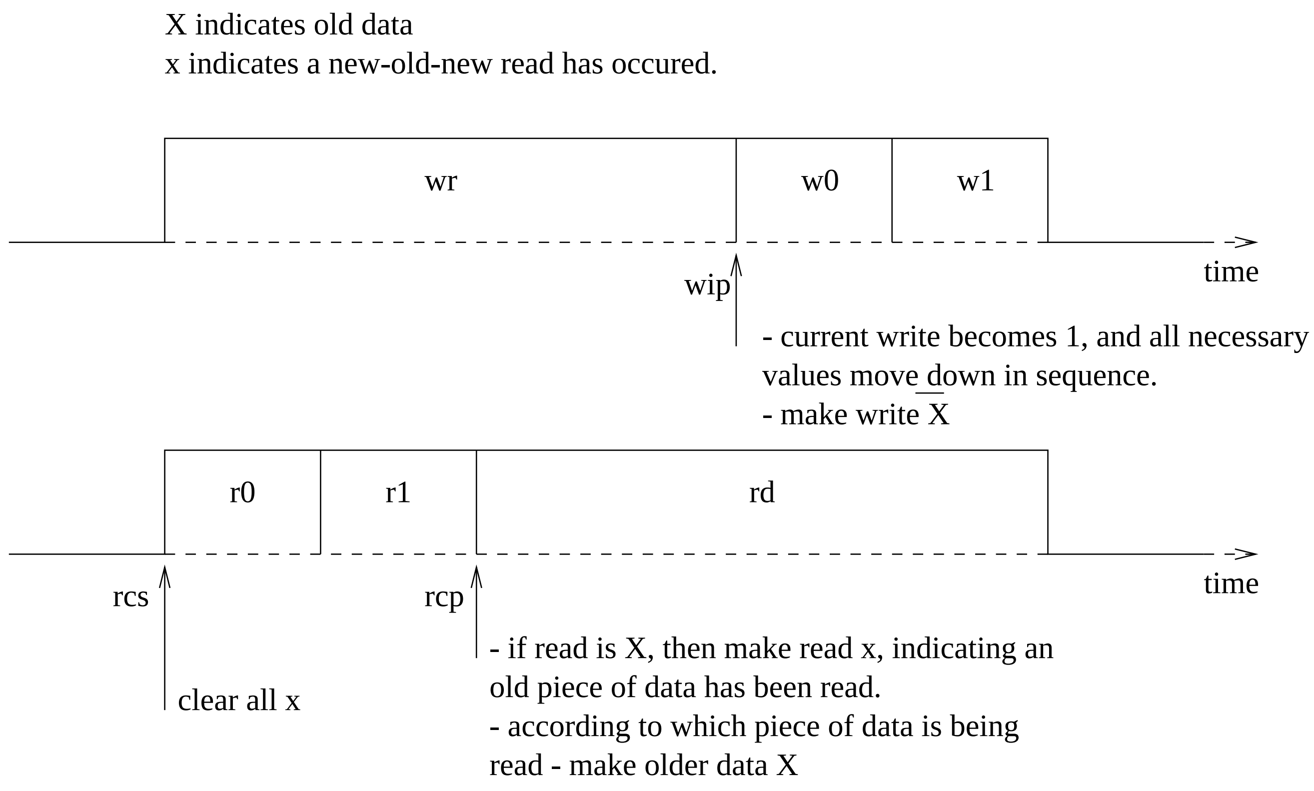


Figure 4.32: A time line showing the data sequencing test.

4.4.5 Checking for full asynchronism

If a mechanism / algorithm can maintain all the previously mentioned properties, it still does not prove that the mechanism is fully asynchronous. This full asynchronism test is however already being produced when creating a reachability tree. In a fully asynchronous environment, every state must have a number of child states equal to the total number of processes in the system. If there are not two child states for any state then some form of blocking is affecting the other process. Therefore in the examples presented in this thesis with one writer and one reader process, each state should have two child states. The only exception to this is 'atomic' instructions which use the 'enable' place, (Section 4.2.6).

4.5 Timing constraints for conditionally asynchronous algorithms

Timing constraints allow an algorithm to be analyzed for an environment where some set of known conditions will not occur (this is equivalent to introducing waiting in a fully asynchronous environment). This relaxes the stringent operating requirements of fully asynchronous operation, and is defined as conditionally asynchronous operation. Timing constraints can also be used to pin-point an erroneous area of operation of an algorithm under test. If during analysis a mechanism fails one of the defined operating properties, it may be observed that the failure only occurs within certain operating conditions, for example when an instruction from one process is interleaved by certain interacting instructions from an other process. To confirm this a timing constraint may be included in the model under test, which will not allow that condition to occur during the analysis. This result either highlights the problem area of the algorithm and changes can be made to the algorithm design, or the result confirms that the algorithm is only conditionally asynchronous.

Classical Petri nets cannot represent actual time, however they represent time relative to the other process. Therefore if a timing constraint is given in terms of "instruction A must be longer then instruction B", then in Petri nets this would be represented by instruction B being restricted from starting or finishing if instruction

A is being executed (depending on the actual constraint). Three example timing constraints are discussed here.

The first timing constraint restricts an instruction starting (A or C) until the other instruction has finished (B). This is shown in Figure 4.33 and the Petri net for this constraint is given in Figure 4.34.

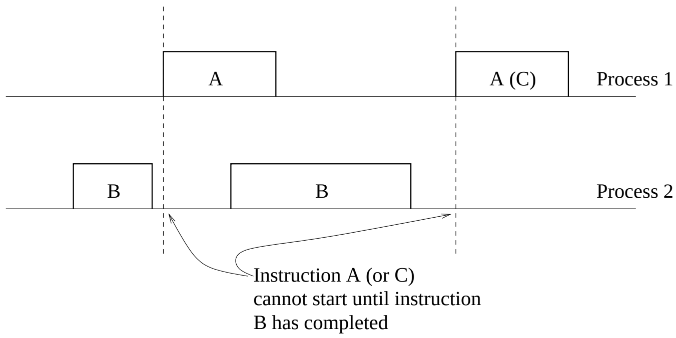


Figure 4.33: Timing constraint, instruction A (or C) cannot start until instruction B has finished.

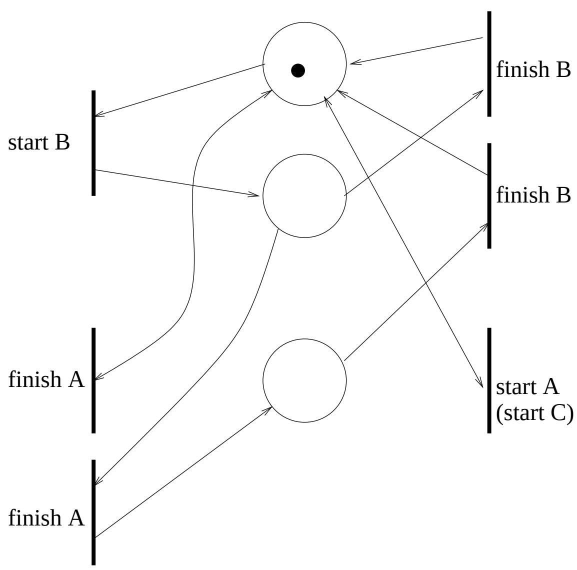


Figure 4.34: Petri net model of constraint for instruction A (or C) cannot start until instruction B has finished.

The second constraint restricts instruction A (or C) finishing until instruction B has finished. This timing constraint is given in Figure 4.35 and the Petri net for this constraint is given in Figure 4.36.

The third constraint restricts instruction A (or C) starting or finishing until instruction B has finished. This timing constraint is given in Figure 4.37 and the Petri net for this constraint is given in Figure 4.38.

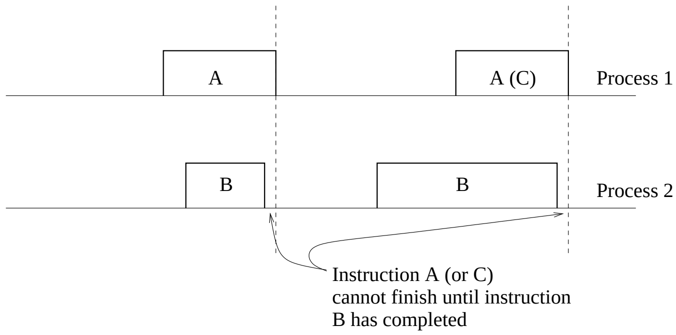


Figure 4.35: Timing constraint, instruction A (or C) cannot finish until instruction B has finished.

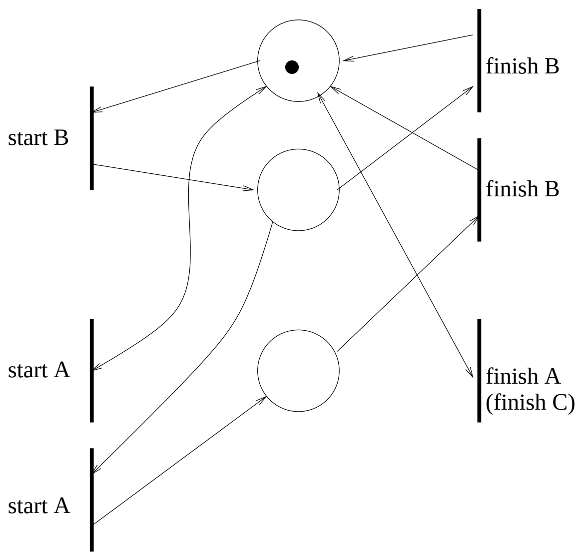


Figure 4.36: Petri net model of constraint for instruction A (or C) cannot finish until instruction B has finished.

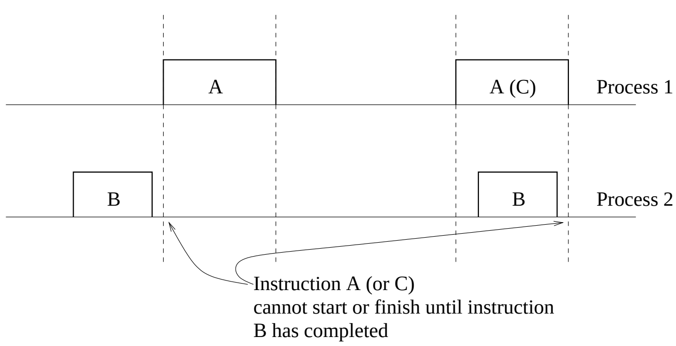


Figure 4.37: Timing constraint, instruction A (or C) cannot start or finish until instruction B has finished.

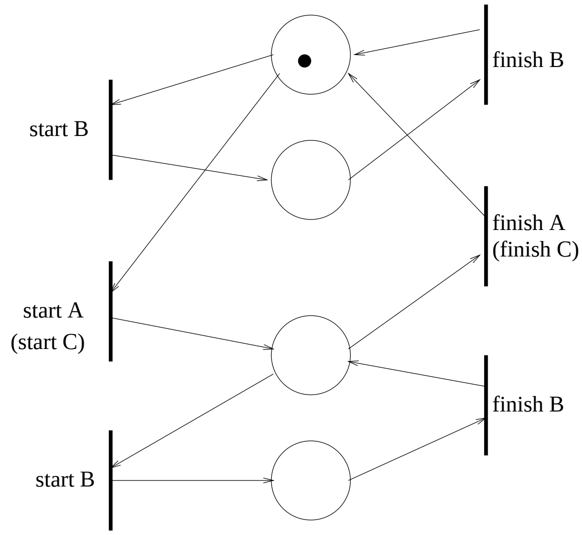


Figure 4.38: Petri net model of constraint for instruction A (or C) cannot start or finish until instruction B has finished.

These three examples show that Petri nets can accurately model timing constraints, and these constraints will be used in Chapter 5.

4.6 One slot mechanism

The one slot mechanism is the simplest form of buffered communication. It is just a single multi-bit register in between the two communicating processes. There are no control variables because no steering of the communicating processes is possible. As there is only a single slot, inclusion of control variables would only introduce process waiting which is unacceptable for true asynchronous communication.

The one slot algorithm [Sim90b] is shown in Figure 4.39 (and in algebraic form in Figure 4.40), the textual Petri model for *reach* is shown in Figure 4.41 and the graphic representation is shown in Figure 4.42.

Figure 4.42 does not show the *coh_lost* transition, which checks to see that the places labelled *write* and *read* are not marked at the same instant in time. This is shown in Figure 4.25.

This model has 5 transitions and 5 places. From reachability searching there are 4 unique non-bad markings, and 1 bad marking, although the bad marking has more than one route. It is trivial to see that because we make no timing implications on

mechanism *one slot*;

var *data* : **data** := *null*;

procedure *write* (*item* : **data**);
begin *data* := *item* **end**;

function *read* : **data**;
begin *read* := *data* **end**;

end.

Figure 4.39: The one slot algorithm

Writer	Reader
$wr : d := input$	$rd : output := d$

Figure 4.40: One slot algorithm in algebraic form.

```
1 slot software model from Simpson's first paper
bad=[coh_lost]
initial=[nowrite,noread]
COM: writer
startw,   [nowrite],    [write]
endw,     [write],      [nowrite]
COM: reader
startr,   [noread],     [read]
endr,     [read],       [noread]
COM: coherence check
coh_lost, [write,read], [write,read,coh_lost]
```

Figure 4.41: The one slot Petri net model

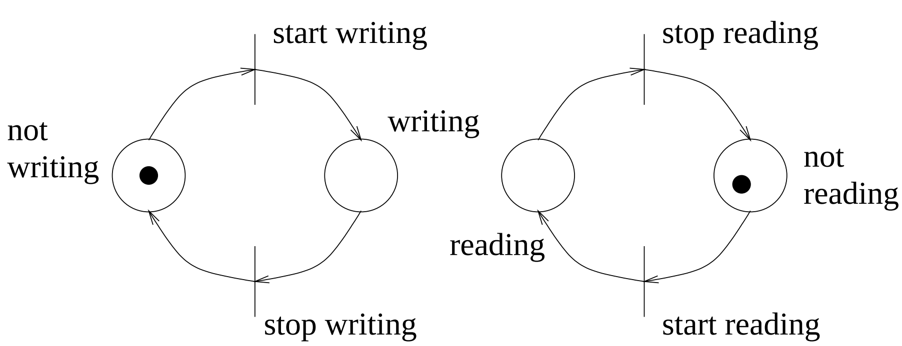


Figure 4.42: The one slot graphical Petri net model

the writing and reading of the mechanism both processes can access the one slot at the same time. The example bad route given by *reach* is shown in Figure 4.43.

```
Reachability model input file = '1slot_coh'. Path = /home/ian/petri/1slot
Number of transitions in model = 5, number of places in model = 5.

bad marking=[write,read,coh_lost] route=[coh_lost,startr,startw]

Number of markings found (not including bad markings) = 4.
Number of bad markings found = 1.
```

Figure 4.43: One slot output from 'reach'

This naturally leads to the design modification of adding another slot to make room for the two processes to access the mechanism at the same time, with corresponding re-design of the algorithm. This is known as the two slot mechanism.

4.7 Two slot mechanism

By adding a second slot but no control variables as shown in Figure 4.44, the data coherence property can be maintained, but no data is ever passed through the mechanism (data sequence is also maintained because there is no data to get out of sequence). This failure would be picked up by the ACM freshness property.

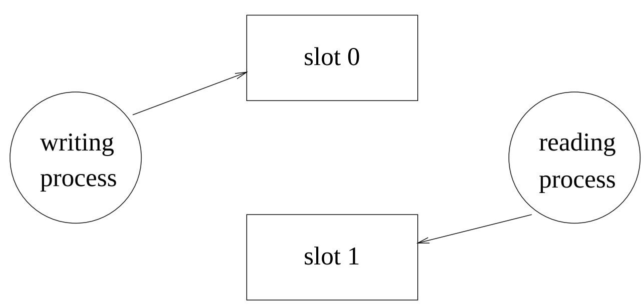


Figure 4.44: A two slot mechanism with no control variables

Several two slot solutions have been previously published [Pat79, Sim90b]. The algorithm used here is from Simpson [Sim90b], with the introduction of control/steering variables, see Figure 4.45. There is one global shared variable 'latest' and each process has its own local variable 'index'. The writer process simply alternates between data slots so as not to overwrite the last piece of data written, and the reading process chases the writer via the global variable 'latest' which indicates which was the last slot to be written.

```
mechanism two slot;

        var data : array[bit] of data := (null, null);
            latest : bit := 0;

        procedure write (item : data);
        var index : bit;
        begin
            index := not latest;            {w0}
            data[index] := item;            {wr}
            latest := index;                {w1}
        end;

        function read : data;
        var index := bit;
        begin
            index := latest;                {r0}
            read := data[index];            {rd}
        end;
end.
```

Figure 4.45: The two slot algorithm

Writer	Reader
$wr : d[\bar{l}] := input$	$r0 : r := l$
$w0 : l := \bar{l}$	$rd : output := d[r]$

Figure 4.46: Two slot algorithm in algebraic form.

Although the 2 slot algorithm operates correctly only in the case of restrictions in the relative timing between writer and reader, it is simple and therefore enables a detailed explanation to be given of the analysis method using a reachability search of Petri net models. Such a detailed description would take up too much space in the case of more complex algorithms.

The algorithm in Figure 4.45 is modelled in Petri nets by breaking it down into individual statements and then controlling the sequential flow of each process using control variables. For this algorithm the writer is broken down into $w0$, *writing* (wr) and $w1$. The writer part of the textual model used for the analysis in 'reach' is given in Figure 4.47, and the three model parts are given graphically in Figure 4.48 for $w0$,

```
2 slot software version 2 done broken down into statements as with 4 slot.
bad=[coh_lost]
initial=[w0,nowrite0,nowrite1,r0,noread0,noread1,latest=0,windex=0,rindex=0]
COM writer
COM
COM w0 index := not latest;
COM
w0001,[w0,latest=0,windex=0],[wr,latest=0,windex=1]
w0011,[w0,latest=0,windex=1],[wr,latest=0,windex=1]
w0110,[w0,latest=1,windex=1],[wr,latest=1,windex=0]
w0100,[w0,latest=1,windex=0],[wr,latest=1,windex=0]
COM
COM start write - data[index] := item;
COM
strtwri0,[wr,windex=0,nowrite0],[windex=0,write0]
strtwri1,[wr,windex=1,nowrite1],[windex=1,write1]
COM
COM end write
COM
endwri0,[write0],[w1,nowrite0]
endwri1,[write1],[w1,nowrite1]
COM
COM w1 latest := index;
COM
w1000,[w1,latest=0,windex=0],[w0,latest=0,windex=0]
w1010,[w1,latest=1,windex=0],[w0,latest=0,windex=0]
w1101,[w1,latest=0,windex=1],[w0,latest=1,windex=1]
w1111,[w1,latest=1,windex=1],[w0,latest=1,windex=1]
```

Figure 4.47: The two slot writer Petri net model

Figure 4.49 for the *writing* part, and Figure 4.50 for $w1$. The Petri net model initial condition is also given in the textual model. The Petri net models for $w0$ and $w1$ are both atomic because in both cases the copying or setting of shared variables is only one way in the single instruction. The algorithm given in Figure 4.45 is rewritten in algebraic form in Figure 4.46.

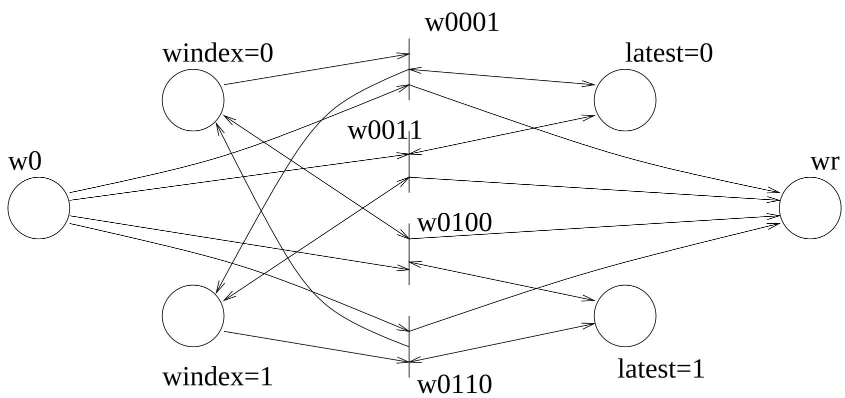


Figure 4.48: Two slot $w0$ Petri net model

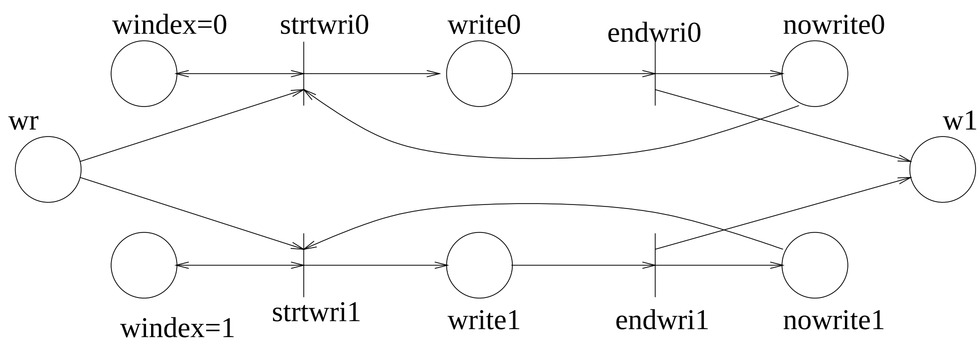


Figure 4.49: Two slot *writing* Petri net model

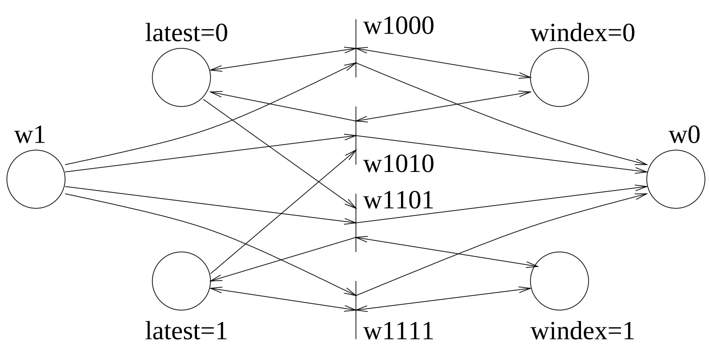


Figure 4.50: Two slot $w1$ Petri net model

The two slot reader textual model is given in Figure 4.51, this is made up from two instructions $r0$ and *reading*, which are shown in Figures 4.52, and 4.53 respectively. The model used for checking coherence is shown in Figure 4.54

```
COM
COM reader
COM
COM r0 index := latest;
COM
r0000,[latest=0,r0,rindex=0],[latest=0,rd,rindex=0]
r0010,[latest=0,r0,rindex=1],[latest=0,rd,rindex=0]
r0101,[latest=1,r0,rindex=0],[latest=1,rd,rindex=1]
r0111,[latest=1,r0,rindex=1],[latest=1,rd,rindex=1]
COM
COM start read - read := data[index];
COM
strtrd0,[rd,rindex=0,noread0],[rindex=0,read0]
strtrd1,[rd,rindex=1,noread1],[rindex=1,read1]
COM
COM end read
COM
endrd0,[read0],[r0,noread0]
endrd1,[read1],[r0,noread1]
COM
COM check coherence
COM
coh_lost0,[write0,read0],[write0,read0,coh_lost]
coh_lost1,[write1,read1],[write1,read1,coh_lost]
END.
```

Figure 4.51: The two slot reader Petri net model

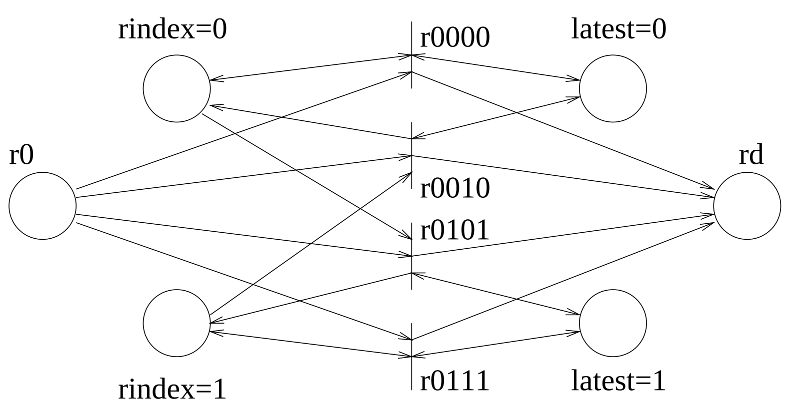


Figure 4.52: Two slot $r0$ Petri net model

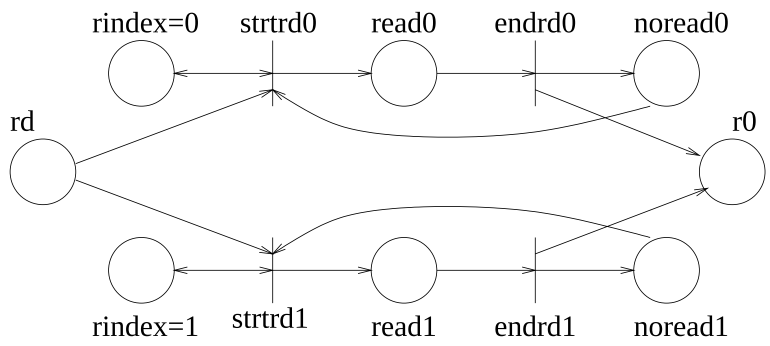


Figure 4.53: Two slot *rd* Petri net model

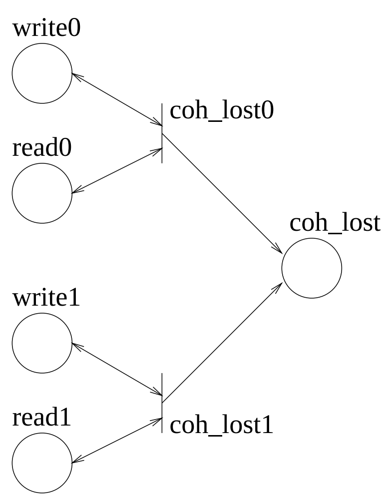


Figure 4.54: Two slot coherence Petri net model

The two slot model has 22 transitions and 20 places. When analysed by 'reach' 48 unique markings plus 2 bad markings are found, 1 for each slot, which indicates that data coherence is not maintained in a fully asynchronous environment, confirming the results from Simpson [Sim90b]. The output showing this is given in Figure 4.55. The two bad markings show the shortest routes available from the initial condition to the coherence being lost on each slot. Note that the routes are listed in reverse order, so the first transition listed is the one causing the bad marking, a bad marking being any marking which contains a place listed as bad in the model, in this case *coh_lost*.

```
Reachability model input file = '2slot_coh_single'. Path = /home/ian/petri/2slot
Number of transitions in model = 22, number of places in model = 20.

Note :- Bad marking routes are given in reverse order.

bad marking=[coh_lost,latest=1,noread1,nowrite1,read0,rindex=0,windex=0,write0]
route=[coh_lost0,strtwri0,w0110,w1101,endwri1,strtwri1,w0001,strtrd0,r0000]

bad marking=[coh_lost,latest=0,noread0,nowrite0,read1,rindex=1,windex=1,write1]
route=[coh_lost1,strtwri1,w0001,w1010,endwri0,strtwri0,w0110,strtrd1,r0101,w1101
,endwri1,strtwri1,w0001]

Number of markings found (not including bad markings) = 48.
Number of bad markings found = 2.
```

Figure 4.55: Two slot reachability output

The first bad route given in Figure 4.55 shows that the mechanism operation starts with $r0$, and then starts to *read slot 0*. The writer then executes $w0$, *writes to slot 1*, $w1$, $w0$, and then starts *writing to slot 0*. Both processes are now accessing the same slot and data coherence is lost. The second bad route, starts with *writes to slot 1*, then the reader starts with *read slot 1*. The writer continues on, *writes to slot 0*, and then comes back again to start *writing in slot 1*, so again both processes are accessing the same slot and therefore there is a loss of data coherence.

This first model of the two slot mechanism uses an initial condition of all variables set to binary 0. To verify that the initial condition does not affect this result all initial conditions must be checked. By changing the initial condition to that shown in Figure 4.56, adding the transitions shown in the same Figure, and making all existing transitions reference the place labelled 'run', all initial conditions are checked.

The model used for checking all initial conditions has 28 transitions and 24 places,

```
2 slot software version 2 done broken down into statements as with 4 slot.
test for all initial conditions
bad=[coh_lost]
initial=[start,w0,nowrite0,nowrite1,r0,noread0,noread1]
COM
COM Initial condition set-up
COM
latest_0,[start],[latest=0,start2]
latest_1,[start],[latest=1,start2]
windex_0,[start2],[windex=0,start3]
windex_1,[start2],[windex=1,start3]
rindex_0,[start3],[rindex=0,run]
rindex_1,[start3],[rindex=1,run]
```

Figure 4.56: Two slot model producing all initial conditions

the number of unique markings is 63, plus there are 2 bad markings, implying that data coherence is not maintained. The 2 bad marking routes found were different from the initial condition of all 0's above, and each initial condition has been tried separately, each giving 2 bad markings, 1 for each slot confirming that the 2 slot mechanism cannot maintain data coherence in a fully asynchronous environment. This is not surprising if one thinks of the operation of each process. The writing process will never write to the same slot twice in succession so 2 slots are required for the writing process alone. The reading process seems to only require one slot, therefore 3 slots would seem to provide enough space for fully asynchronous communication.

By using run time options during the execution of the analysis package *reach* it is possible to search for possible timing constraints (Section 4.5) that could be added to the mechanism to maintain data coherence. This analysis indicates that a timing constraint on the 2 slot will allow it to maintain coherence. The timing constraint given by Simpson [Sim92a] implies that the duration of one cycle of the read access (rab to rae) should be less than the interval between two write acceses (wab to wae), (see Figure 4.57). Using Simpson's role model notation this is $(wab - wae) > (rae - rab)$ which can be rewritten as $(wab - rae) > (wae - rab)$. It is not stated by Simpson but this timing constraint only allows the open access version to maintain data coherence and not the closed access version. For closed access the timing constraint needs to cover the timing of the whole mechanism access including pre-read or post-write control, rather than just the data access part of the algorithm. This new timing constraint can be written as "the time taken by a single read process cycle must be less than the period between two write cycles". A new two slot Petri net model with the timing constraint was constructed having 30 transitions and 23 places. This new model uses the additional timing constraint Petri net found in Section 4.5, using the start and end of cycle rather than start and end of data access for the transitions. The model for this is given in Figures 4.58 and 4.59. This produced a state space with 36 states plus 0 bad states. This result confirms Simpson's result [Sim92a] that the two slot mechanism can maintain data coherence with the addition of a timing constraint.

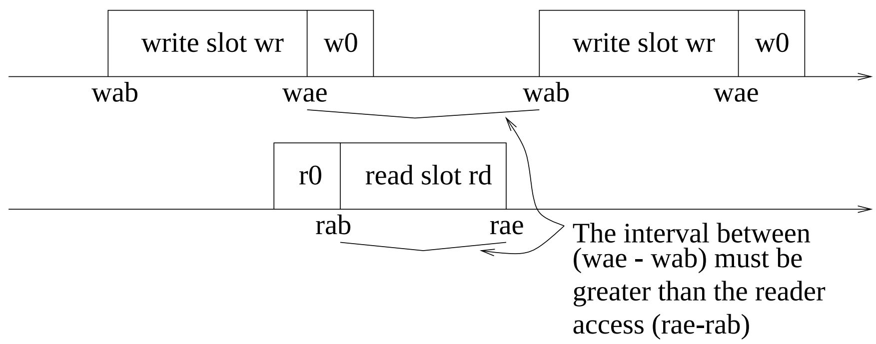


Figure 4.57: Two slot timing constraint shown as a time line

However, now the two slot can maintain data coherence with a timing constraint, it needs to be checked for data freshness and data sequencing (this has not previously been published). The two slot model for checking data freshness with a timing constraint has 64 transitions and 51 places. The model is given in Figures 4.60 and 4.61. This produces a state space with 299 states plus 0 bad states. This means that the two slot algorithm with a timing constraint can maintain data freshness.

The two slot model to test for data sequencing with the timing constraint has 71 transitions and 59 places. The model is given in Figures 4.62 and 4.63. This produces a state space with 343 states plus 0 bad states. This means the two slot algorithm with a timing constraint can maintain data sequencing.

4.8 Three slot mechanism

Because the two slot does not provide enough room for the two processes to access the mechanism asynchronously without a timing constraint the addition of another slot is necessary, making a three slot mechanism. Several authors have published 3

```
2 slot software version 2. Timing constraint on whole read/write time,
not just wr and rd from Simpson which is for open access.
bad=[coh_lost]
initial=[time1,w0,nowrite0,nowrite1,r0,noread0,noread1,latest=0,
windex=0,rindex=0]
COM writer
COM
COM w0 index := not latest;
COM
w0001,[w0,latest=0,windex=0,time1],[wr,latest=0,windex=1,time1]
w0011,[w0,latest=0,windex=1,time1],[wr,latest=0,windex=1,time1]
w0110,[w0,latest=1,windex=1,time1],[wr,latest=1,windex=0,time1]
w0100,[w0,latest=1,windex=0,time1],[wr,latest=1,windex=0,time1]
COM
COM start write - data[index] := item;
COM
strtwri0,[wr,windex=0,nowrite0],[windex=0,write0]
strtwri1,[wr,windex=1,nowrite1],[windex=1,write1]
COM
COM end write
COM
endwri0,[write0],[w1,nowrite0]
endwri1,[write1],[w1,nowrite1]
endwri0a,[write0],[w1,nowrite0]
endwri1a,[write1],[w1,nowrite1]
COM
COM w1 latest := index;
COM
w1000,[w1,latest=0,windex=0,time1],[w0,latest=0,windex=0,time1]
w1010,[w1,latest=1,windex=0,time1],[w0,latest=0,windex=0,time1]
w1101,[w1,latest=0,windex=1,time1],[w0,latest=1,windex=1,time1]
w1111,[w1,latest=1,windex=1,time1],[w0,latest=1,windex=1,time1]
w1000a,[w1,latest=0,windex=0,time2],[w0,latest=0,windex=0,time3]
w1010a,[w1,latest=1,windex=0,time2],[w0,latest=0,windex=0,time3]
w1101a,[w1,latest=0,windex=1,time2],[w0,latest=1,windex=1,time3]
w1111a,[w1,latest=1,windex=1,time2],[w0,latest=1,windex=1,time3]
```

Figure 4.58: Two slot writer model with timing constraint

```
COM
COM reader
COM
COM r0 index := latest;
COM
r0000,[latest=0,r0,rindex=0,time1],[latest=0,rd,rindex=0,time2]
r0010,[latest=0,r0,rindex=1,time1],[latest=0,rd,rindex=0,time2]
r0101,[latest=1,r0,rindex=0,time1],[latest=1,rd,rindex=1,time2]
r0111,[latest=1,r0,rindex=1,time1],[latest=1,rd,rindex=1,time2]
COM
COM start read - read := data[index];
COM
strtrd0,[rd,rindex=0,noread0],[rindex=0,read0]
strtrd1,[rd,rindex=1,noread1],[rindex=1,read1]
COM
COM end read
COM
endrd0,[read0,time2],[r0,noread0,time1]
endrd1,[read1,time2],[r0,noread1,time1]
endrd0a,[read0,time3],[r0,noread0,time1]
endrd1a,[read1,time3],[r0,noread1,time1]
COM
COM check coherence
COM
coh_lost0,[write0,read0],[write0,read0,coh_lost]
coh_lost1,[write1,read1],[write1,read1,coh_lost]
END.
```

Figure 4.59: Two slot reader model with timing constraint

```
2 slot software version 2. Timing constraint on whole read/write time,
not just wr and rd from Simpson which is for open access. Freshness.
bad=[fre_lost]
initial=[no_w1,no_r0,enable,time1,w0,nowrite0,nowrite1,clear_valid,
noread0,noread1,latest=0,windex=0,rindex=0,no_fresh,no_prev,not_valid_0,not_valid_1]
COM writer
COM
COM w0 index := not latest;
COM
w0001,[enable,w0,latest=0,windex=0,time1],[enable,wr,latest=0,windex=1,time1]
w0011,[enable,w0,latest=0,windex=1,time1],[enable,wr,latest=0,windex=1,time1]
w0110,[enable,w0,latest=1,windex=1,time1],[enable,wr,latest=1,windex=0,time1]
w0100,[enable,w0,latest=1,windex=0,time1],[enable,wr,latest=1,windex=0,time1]
COM
COM start write - data[index] := item;
COM
strtwri0,[enable,wr,windex=0,nowrite0],[enable,windex=0,write0]
strtwri1,[enable,wr,windex=1,nowrite1],[enable,windex=1,write1]
COM
COM end write
COM
endwri0,[enable,write0],[wrote0,f_to_p,nowrite0]
endwri1,[enable,write1],[wrote1,f_to_p,nowrite1]
endwri0a,[enable,write0],[wrote0,f_to_p,nowrite0]
endwri1a,[enable,write1],[wrote1,f_to_p,nowrite1]
nftop,[f_to_p,no_fresh],[w_to_f,no_fresh]
ftop0,[fresh_0,no_prev,f_to_p],[no_fresh,prev_0,w_to_f]
ftop1,[fresh_1,no_prev,f_to_p],[no_fresh,prev_1,w_to_f]
wtofv0b,[wrote0,valid_0,w_to_f,no_fresh],[w1,enable,fresh_0,valid_0]
wtofv0c,[wrote0,not_valid_0,w_to_f,no_fresh],[w1,enable,fresh_0,valid_0]
wtofv1b,[wrote1,valid_1,w_to_f,no_fresh],[w1,enable,fresh_1,valid_1]
wtofv1c,[wrote1,not_valid_1,w_to_f,no_fresh],[w1,enable,fresh_1,valid_1]
COM
COM w1 latest := index;
COM
w1000,[enable,w1,latest=0,windex=0,time1],[w1_post,latest=0,windex=0,time1]
w1010,[enable,w1,latest=1,windex=0,time1],[w1_post,latest=0,windex=0,time1]
w1101,[enable,w1,latest=0,windex=1,time1],[w1_post,latest=1,windex=1,time1]
w1111,[enable,w1,latest=1,windex=1,time1],[w1_post,latest=1,windex=1,time1]
w1000a,[enable,w1,latest=0,windex=0,time2],[w1_post,latest=0,windex=0,time3]
w1010a,[enable,w1,latest=1,windex=0,time2],[w1_post,latest=0,windex=0,time3]
w1101a,[enable,w1,latest=0,windex=1,time2],[w1_post,latest=1,windex=1,time3]
w1111a,[enable,w1,latest=1,windex=1,time2],[w1_post,latest=1,windex=1,time3]
w1_initial,[w1_post,no_w1],[clear_prev,w1+]
w1_normal,[w1_post,w1+],[clear_prev,w1+]
ncp,[no_prev,clear_prev],[w0,enable,no_prev]
cp0,[prev_0,clear_prev],[w0,enable,no_prev]
cp1,[prev_1,clear_prev],[w0,enable,no_prev]
```

Figure 4.60: Two slot data freshness writer model with timing constraint

```
COM
COM reader
COM
COM r0 index := latest;
COM
clear_valid_00a,[not_valid_0,clear_valid,enable],[not_valid_0,clear_valid2]
clear_valid_00b,[valid_0,clear_valid,enable],[not_valid_0,clear_valid2]
clear_valid_01a,[not_valid_1,clear_valid2],[not_valid_1,f_to_v]
clear_valid_01b,[valid_1,clear_valid2],[not_valid_1,f_to_v]
nftov,[f_to_v,no_fresh,not_valid_0,not_valid_1],[no_fresh,valid_0,valid_1,r0_pre]
ftov00,[f_to_v,fresh_0,not_valid_0],[fresh_0,valid_0,p_to_v]
ftov01,[f_to_v,fresh_1,not_valid_1],[fresh_1,valid_1,p_to_v]
nptov,[no_prev,p_to_v],[no_prev,r0_pre]
ptov00,[prev_0,not_valid_0,p_to_v],[prev_0,valid_0,r0_pre]
ptov01,[prev_1,not_valid_1,p_to_v],[prev_1,valid_1,r0_pre]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0000,[latest=0,r0,rindex=0,time1],[enable,latest=0,pre_read_ctrl,rindex=0,time2]
r0010,[latest=0,r0,rindex=1,time1],[enable,latest=0,pre_read_ctrl,rindex=0,time2]
r0101,[latest=1,r0,rindex=0,time1],[enable,latest=1,pre_read_ctrl,rindex=1,time2]
r0111,[latest=1,r0,rindex=1,time1],[enable,latest=1,pre_read_ctrl,rindex=1,time2]
COM
COM start read - read := data[index];
COM
pre_read_00a,[pre_read_ctrl,enable,rindex=0,no_fresh],[no_fresh,rd,rindex=0]
pre_read_00b,[pre_read_ctrl,enable,rindex=0,fresh_0],[fresh_0,clear_prev_by_r,
rindex=0]
pre_read_00c,[pre_read_ctrl,enable,rindex=0,fresh_1],[fresh_1,rd,rindex=0]
pre_read_01a,[pre_read_ctrl,enable,rindex=1,no_fresh],[no_fresh,rd,rindex=1]
pre_read_01b,[pre_read_ctrl,enable,rindex=1,fresh_0],[fresh_0,rd,rindex=1]
pre_read_01c,[pre_read_ctrl,enable,rindex=1,fresh_1],[fresh_1,clear_prev_by_r,
rindex=1]
ncp_r,[no_prev,clear_prev_by_r],[rd,no_prev]
cp00_r,[prev_0,clear_prev_by_r],[rd,no_prev]
cp01_r,[prev_1,clear_prev_by_r],[rd,no_prev]
strtrd0,[rd,rindex=0,noread0],[enable,rindex=0,read0]
strtrd1,[rd,rindex=1,noread1],[enable,rindex=1,read1]
COM
COM end read
COM
endrd0,[enable,read0,time2],[enable,clear_valid,noread0,time1]
endrd1,[enable,read1,time2],[enable,clear_valid,noread1,time1]
endrd0a,[enable,read0,time3],[enable,clear_valid,noread0,time1]
endrd1a,[enable,read1,time3],[enable,clear_valid,noread1,time1]
COM
COM check freshness
COM
not_fresh_00,[enable,read0,not_valid_0,r0+],[enable,read0,not_valid_0,fre_lost,r0+]
not_fresh_01,[enable,read1,not_valid_1,r0+],[enable,read1,not_valid_1,fre_lost,r0+]
END.
```

Figure 4.61: Two slot data freshness reader model with timing constraint

```
2 slot software version 2. Timing constraint on whole read/write time,
not just wr and rd from Simpson which is for open access. data sequencing.
bad=[seq_lost]
initial=[no_w1,no_r0,enable,time1,w0,nowrite0,nowrite1,clear_x,noread0,
noread1,latest=0,windex=0,rindex=0,Xb_0,Xb_1,x_0,x_1,2_0,2_1]
COM writer
COM w0 index := not latest;
w0001,[w0,latest=0,windex=0,time1,enable],[wr,latest=0,windex=1,time1,enable]
w0011,[w0,latest=0,windex=1,time1,enable],[wr,latest=0,windex=1,time1,enable]
w0110,[w0,latest=1,windex=1,time1,enable],[wr,latest=1,windex=0,time1,enable]
w0100,[w0,latest=1,windex=0,time1,enable],[wr,latest=1,windex=0,time1,enable]
COM
COM start write - data[index] := item;
strtwri0,[wr,windex=0,nowrite0,enable],[windex=0,write0,enable]
strtwri1,[wr,windex=1,nowrite1,enable],[windex=1,write1,enable]
COM
COM end write
endwri0,[write0,enable],[wip_roles0,nowrite0]
endwri1,[write1,enable],[wip_roles1,nowrite1]
endwri0a,[write0,enable],[wip_roles0,nowrite0]
endwri1a,[write1,enable],[wip_roles1,nowrite1]
COM make current write Xb
wip1a,[wip_roles0,X_0],[wip_role20,Xb_0]
wip1b,[wip_roles0,Xb_0],[wip_role20,Xb_0]
wip1c,[wip_roles1,X_1],[wip_role21,Xb_1]
wip1d,[wip_roles1,Xb_1],[wip_role21,Xb_1]
COM make current write 1 and make the other 2
wip2a,[wip_role20,1_0],[1_0,w1,enable]
wip2b,[wip_role21,1_1],[1_1,w1,enable]
wip2c,[wip_role20,2_0],[1_0,wip_roles30]
wip2d,[wip_role21,2_1],[1_1,wip_roles31]
wip3a,[wip_roles30,1_1],[2_1,w1,enable]
wip3b,[wip_roles31,1_0],[2_0,w1,enable]
wip3c,[wip_roles30,2_1],[2_1,w1,enable]
wip3d,[wip_roles31,2_0],[2_0,w1,enable]
COM
COM w1 latest := index;
w1000,[w1,latest=0,windex=0,time1,enable],[w1_post,latest=0,windex=0,time1]
w1010,[w1,latest=1,windex=0,time1,enable],[w1_post,latest=0,windex=0,time1]
w1101,[w1,latest=0,windex=1,time1,enable],[w1_post,latest=1,windex=1,time1]
w1111,[w1,latest=1,windex=1,time1,enable],[w1_post,latest=1,windex=1,time1]
w1000a,[w1,latest=0,windex=0,time2,enable],[w1_post,latest=0,windex=0,time3]
w1010a,[w1,latest=1,windex=0,time2,enable],[w1_post,latest=0,windex=0,time3]
w1101a,[w1,latest=0,windex=1,time2,enable],[w1_post,latest=1,windex=1,time3]
w1111a,[w1,latest=1,windex=1,time2,enable],[w1_post,latest=1,windex=1,time3]
w1_initial,[w1_post,no_w1],[w0,w1+,enable]
w1_normal,[w1_post,w1+],[w0,w1+,enable]
```

Figure 4.62: Two slot data sequencing writer model with timing constraint

```
COM reader
COM r0 index := latest;
clr_x00,[enable,clear_x,x_0],[clear_x2,xb_0]
clr_x00a,[enable,clear_x,xb_0],[clear_x2,xb_0]
clr_x01,[clear_x2,x_1],[r0_pre,xb_1]
clr_x01a,[clear_x2,xb_1],[r0_pre,xb_1]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0000,[latest=0,r0,rindex=0,time1],[latest=0,rcp_roles,rindex=0,time2,enable]
r0010,[latest=0,r0,rindex=1,time1],[latest=0,rcp_roles,rindex=0,time2,enable]
r0101,[latest=1,r0,rindex=0,time1],[latest=1,rcp_roles,rindex=1,time2,enable]
r0111,[latest=1,r0,rindex=1,time1],[latest=1,rcp_roles,rindex=1,time2,enable]
COM
COM start read - read := data[index];
x_r00a,[enable,rcp_roles,rindex=0,X_0,xb_0],[rindex=0,X_0,x_0,rcp_roles2_0]
x_r00b,[enable,rcp_roles,rindex=0,Xb_0],[rindex=0,Xb_0,rcp_roles2_0]
x_r01a,[enable,rcp_roles,rindex=1,X_1,xb_1],[rindex=1,X_1,x_1,rcp_roles2_1]
x_r01b,[enable,rcp_roles,rindex=1,Xb_1],[rindex=1,Xb_1,rcp_roles2_1]
X_r00a,[rcp_roles2_0,1_0],[1_0,rcp_Xb_0,rcp_X_1,rcp_role3]
X_r00b,[rcp_roles2_0,2_0],[2_0,rcp_Xb_0,rcp_Xb_1,rcp_role3]
X_r01a,[rcp_roles2_1,1_1],[1_1,rcp_Xb_1,rcp_X_0,rcp_role3]
X_r01b,[rcp_roles2_1,2_1],[2_1,rcp_Xb_1,rcp_Xb_0,rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_0,X_0],[Xb_0,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_0,Xb_0],[Xb_0,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_1,X_1],[Xb_1,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_1,Xb_1],[Xb_1,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_1,X_1],[X_1,rd]
rcp_4b,[rcp_role4,rcp_X_1,Xb_1],[X_1,rd]
rcp_4c,[rcp_role4,rcp_X_0,Xb_0],[X_0,rd]
rcp_4d,[rcp_role4,rcp_X_0,X_0],[X_0,rd]
rcp_4e,[rcp_role4,rcp_Xb_0,X_0],[Xb_0,rd]
rcp_4f,[rcp_role4,rcp_Xb_0,Xb_0],[Xb_0,rd]
rcp_4g,[rcp_role4,rcp_Xb_1,X_1],[Xb_1,rd]
rcp_4h,[rcp_role4,rcp_Xb_1,Xb_1],[Xb_1,rd]
strtrd0,[rd,rindex=0,noread0],[rindex=0,read0,enable]
strtrd1,[rd,rindex=1,noread1],[rindex=1,read1,enable]
COM
COM end read
endrd0,[read0,time2,enable],[clear_x,noread0,time1,enable]
endrd1,[read1,time2,enable],[clear_x,noread1,time1,enable]
endrd0a,[read0,time3,enable],[clear_x,noread0,time1,enable]
endrd1a,[read1,time3,enable],[clear_x,noread1,time1,enable]
COM
COM check sequencing
order_lost_00,[enable,read0,x_0,r0+],[enable,read0,x_0,seq_lost,r0+]
order_lost_01,[enable,read1,x_1,r0+],[enable,read1,x_1,seq_lost,r0+]
END.
```

Figure 4.63: Two slot data sequencing reader model with timing constraint

slot solutions [OH76, Sim90b], and the one used here is taken from Simpson [Sim90b], see Figure 4.64. The problem with three slot solutions is that three is not a multiple of two, and therefore the construction of control variables is a problem. Simpsons solution assumes atomic variables with three possible values. A new function called *differ* is used, which given two ternary values, returns a value which is not equal to either of the supplied variables.

```
mechanism three slot;

        var data : array[1..3] of data := (null, null, null);
            latest, reading : 1..3 := 1, 1;

        procedure write (item : data);
        const differ = ((2, 3, 2), (3, 3, 1), (2, 1, 1));
        var index : 1..3;
        begin
            index := differ [latest, reading];
            data[index] := item;
            latest := index
        end;

        function read : data;
        var index := 1..3;
        begin
            index := latest;
            reading := index;
            read := data[index]
        end;
end.
```

Figure 4.64: The three slot algorithm

Writer	Reader
$wr : d[n] := input$	$r0 : r := l$
$w0 : l := n$	$rd : output := d[r]$
$w1 : n := \neg(l \vee r)$	

Figure 4.65: Three slot algorithm in algebraic form.

The algorithm given in Figure 4.64 is re-written in algebraic form for modelling and analysis, see Figure 4.65. The Petri net model for this algorithm is given textually in 'reach' input format in Figures 4.66 and 4.67. The writer process is split up into statements $w0$, *writing* and $w1$, the reading process is split up for $r0$, $r1$ and *reading*. Several of the equality operators use the same basic model so only new implementations are shown graphically. The *differ* function $w0$ is shown in Figure 4.68, *writing* is given in Figure 4.69, $w1$ which uses 'OR' arcs for compactness is given in Figure 4.70. The Petri net model of statement $w1$ includes the use of 'OR' arcs, which were introduced in section 4.2.9. If 'OR' arcs had not been used the model would have required nine transitions, but with them it requires only three. The dashed arcs used to represent 'OR' arcs require that only one token is required from one of the *latest* transitions. The models for the reader process statements are similar to that used within the writer, and the coherence checking model is similar to the one used on the 2 slot analysis, with the addition of 1 extra slot.

```
3 slot version 2 from Simpsons Jan 1990 paper.
bad=[coh_lost]
initial=[w0,latest=1,reading=1,not_writing_1,not_writing_2,not_writing_3,
not_reading_1,not_reading_2,not_reading_3,r0]
COM
COM index := differ[latest, reading] {condt differ = ((2,3,2),(3,3,1),(2,1,1))}
COM
w0321,[w0,latest=2,reading=3],[wr,latest=2,reading=3,w_index=1]
w0231,[w0,latest=3,reading=2],[wr,latest=3,reading=2,w_index=1]
w0331,[w0,latest=3,reading=3],[wr,latest=3,reading=3,w_index=1]
w0112,[w0,latest=1,reading=1],[wr,latest=1,reading=1,w_index=2]
w0312,[w0,latest=1,reading=3],[wr,latest=1,reading=3,w_index=2]
w0132,[w0,latest=3,reading=1],[wr,latest=3,reading=1,w_index=2]
w0213,[w0,latest=1,reading=2],[wr,latest=1,reading=2,w_index=3]
w0123,[w0,latest=2,reading=1],[wr,latest=2,reading=1,w_index=3]
w0223,[w0,latest=2,reading=2],[wr,latest=2,reading=2,w_index=3]
COM
COM data[index]:=item; start write
COM
start_writing_1,[wr,w_index=1,not_writing_1],[w_index=1,writing_1]
start_writing_2,[wr,w_index=2,not_writing_2],[w_index=2,writing_2]
start_writing_3,[wr,w_index=3,not_writing_3],[w_index=3,writing_3]
COM
COM data[index]:=item; end write
COM
stop_writing_1,[writing_1],[w1,not_writing_1]
stop_writing_2,[writing_2],[w1,not_writing_2]
stop_writing_3,[writing_3],[w1,not_writing_3]
COM
COM latest := index;
COM
w1111,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=1],[w0,latest=1]
w1212,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=2],[w0,latest=2]
w1313,[w1,+0latest=1,+0latest=2,+0latest=3,w_index=3],[w0,latest=3]
```

Figure 4.66: The three slot writer Petri net model

```
COM
COM index:=latest;
COM
r011,[latest=1,r0],[latest=1,r_index=1,r1]
r022,[latest=2,r0],[latest=2,r_index=2,r1]
r033,[latest=3,r0],[latest=3,r_index=3,r1]
COM
COM reading := index;
COM
r11,[r1,+0reading=1,+0reading=2,+0reading=3,r_index=1],[rd,reading=1,r_index=1]
r12,[r1,+0reading=1,+0reading=2,+0reading=3,r_index=2],[rd,reading=2,r_index=2]
r13,[r1,+0reading=1,+0reading=2,+0reading=3,r_index=3],[rd,reading=3,r_index=3]
COM
COM read:=data[index]; start read
COM
start_reading_1,[r_index=1,not_reading_1,rd],[reading_1]
start_reading_2,[r_index=2,not_reading_2,rd],[reading_2]
start_reading_3,[r_index=3,not_reading_3,rd],[reading_3]
COM
COM read:=data[index]; end read
COM
stop_reading_1,[reading_1],[not_reading_1,r0]
stop_reading_2,[reading_2],[not_reading_2,r0]
stop_reading_3,[reading_3],[not_reading_3,r0]
COM
COM check coherence
COM
coh_lost1,[writing_1,reading_1],[writing_1,reading_1,coh_lost]
coh_lost2,[writing_2,reading_2],[writing_2,reading_2,coh_lost]
coh_lost3,[writing_3,reading_3],[writing_3,reading_3,coh_lost]
END
```

Figure 4.67: The three slot reader Petri net model

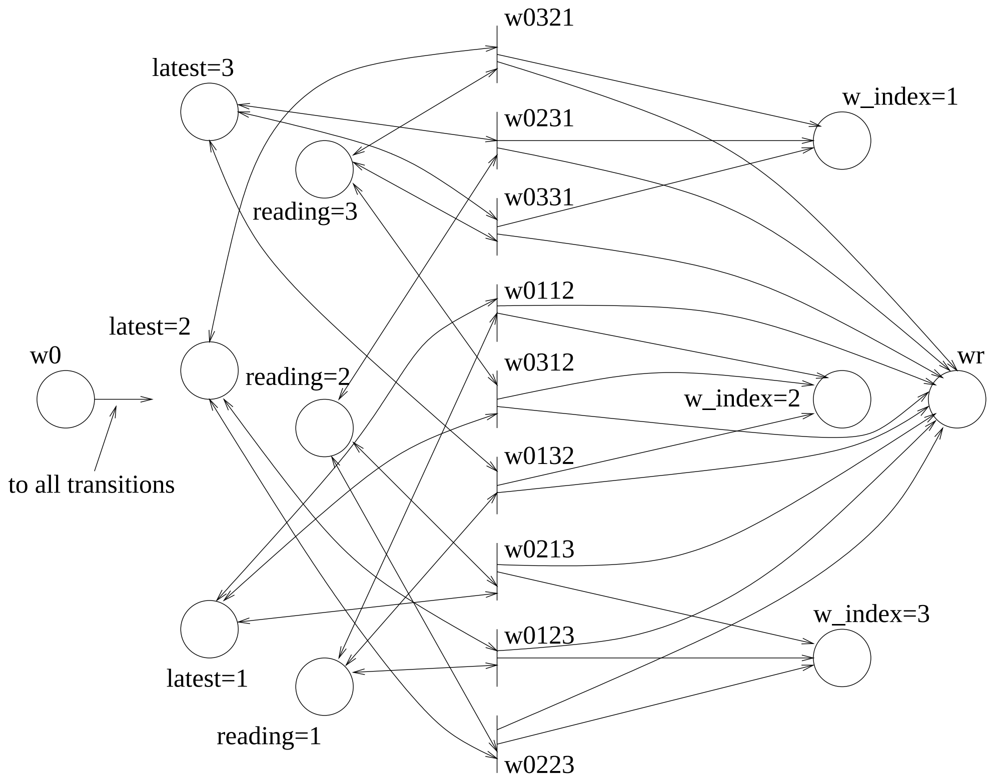


Figure 4.68: The three slot Petri net model of $w0$

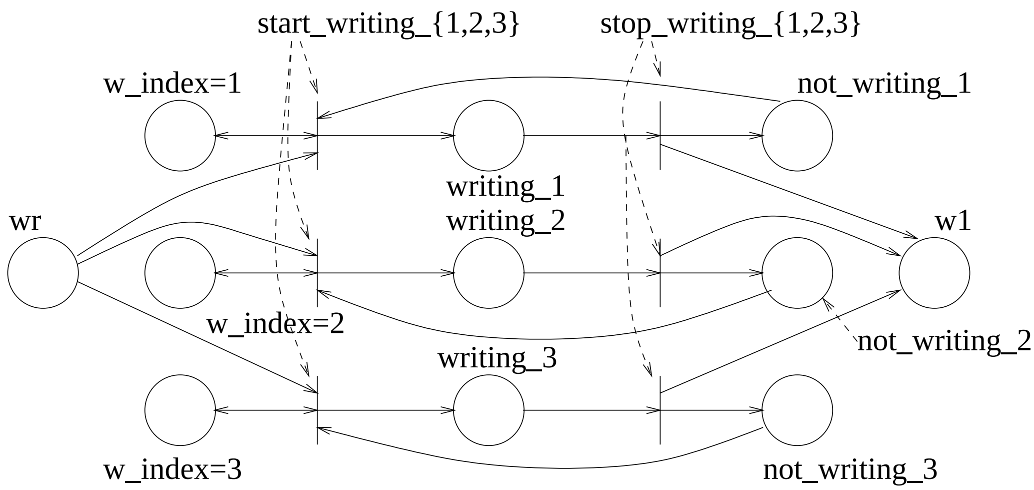


Figure 4.69: The three slot Petri net model of wr

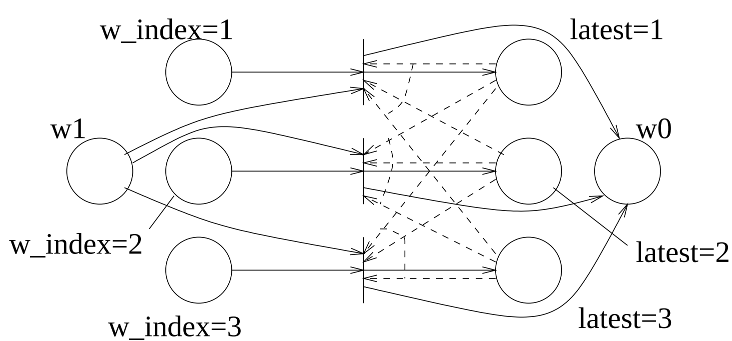


Figure 4.70: The three slot Petri net model of $w1$

This three slot model has 33 transitions, and 31 places. When analysed by *reach* it is found that the state space has 324 unique states plus 6 bad states, which means that the three slot mechanism does not maintain data coherence. This is surprising because it would appear at first sight that three slots should provide enough space for asynchronous access to the mechanism. The 6 bad states are given in Figure 4.71.

By tracing the route of each bad state, it can be seen that these results confirm those given by Simpson [Sim90b]. The interleaving of the control variables for choosing and indicating the next reading and writing slots causes the failure. However Simpson states that three slots do not provide sufficient manoeuvering space for fully asynchronous operation, but this author feels that three slots can provide enough space, it is this algorithm and its control variable switching which causes the failures.

The three slot can have a timing constraint applied to it, (Simpson [Sim92a] gives one for open access as $(wcs - wis) > (ris - rcs)$ or "if wis occurs in the interval between rcs and ris then wcs does not"). No timing constraints are applied in this thesis because the author feels that using ternary value variables is dangerous, (because of sampling changing values). A solution to this problem of ternary value variables may be with the use of ternary logic circuits which have three stable states [TS98], these are however untested with regard to metastability. Xia [XYS$^+$00, XC99] has produced designs for a self timed version of the three slot algorithm, (which were confirmed to maintain data coherence, freshness and sequencing by the author of this thesis), the concurrent access to ternary value variables is resolved by using mutex elements (mutual exclusion elements) on each variable. This implies that if two processes try to access the same variable at the same time then one is forced to wait until the other has finished its access, which is not ideal. Self timed designs produce a 'done' signal which is used to indicate to the calling environment that it has finished its cycle. To check for this signal requires either some form of 'IF' statement or a 'wait' statement, both of which can cause problems.

These results indicate that three slots is sufficient to support conditionally asynchronous communication, i.e. with either a timing constraint or mutually exclusive access to control variables. If an algorithm for a three slot were later found that did not require such constraints, then it would probably still not be as robust as solutions employing four slots.

```
Reachability model input file = '3slot_coh'. Path = /home/ian/petri/3slot
Number of transitions in model = 33, number of places in model = 31.

Note :- Bad marking routes are given in reverse order.

bad marking=[coh_lost,latest=3,not_reading_1,not_reading_3,not_writing_1,
not_writing_3,reading=2,reading_2,w_index=2,writing_2] route=[coh_lost2,
start_writing_2,start_reading_2,r12,w0132,w1313,stop_writing_3,
start_writing_3,w0123,r022,w1212,stop_writing_2,start_writing_2,w0112]

bad marking=[coh_lost,latest=2,not_reading_1,not_reading_2,not_writing_1,
not_writing_2,reading=3,reading_3,w_index=3,writing_3] route=[coh_lost3,
start_writing_3,start_reading_3,r13,w0123,w1212,stop_writing_2,
start_writing_2,w0132,r033,w1313,stop_writing_3,start_writing_3,w0123,
w1212,stop_writing_2,start_writing_2,w0112]

bad marking=[coh_lost,latest=1,not_reading_1,not_reading_2,not_writing_1,
not_writing_2,reading=3,reading_3,w_index=3,writing_3] route=[coh_lost3,
start_writing_3,start_reading_3,r13,w0213,w1111,stop_writing_1,
start_writing_1,w0231,r033,w1313,stop_writing_3,start_writing_3,w0223,
stop_reading_2,start_reading_2,r12,r022,w1212,stop_writing_2,
start_writing_2,w0112]

bad marking=[coh_lost,latest=3,not_reading_2,not_reading_3,not_writing_2,
not_writing_3,reading=1,reading_1,w_index=1,writing_1] route=[coh_lost1,
start_writing_1,start_reading_1,r11,w0231,w1313,stop_writing_3,
start_writing_3,w0213,r011,w1111,stop_writing_1,start_writing_1,w0231,
w1313,stop_writing_3,start_writing_3,w0223,stop_reading_2,start_reading_2,
r12,r022,w1212,stop_writing_2,start_writing_2,w0112]

bad marking=[coh_lost,latest=2,not_reading_2,not_reading_3,not_writing_2,
not_writing_3,reading=1,reading_1,w_index=1,writing_1] route=[coh_lost1,
start_writing_1,start_reading_1,r11,w0321,w1212,stop_writing_2,
start_writing_2,w0312,r011,w1111,stop_writing_1,start_writing_1,w0331,
stop_reading_3,start_reading_3,r13,r033,w1313,stop_writing_3,
start_writing_3,w0123,w1212,stop_writing_2,start_writing_2,w0112]

bad marking=[coh_lost,latest=1,not_reading_1,not_reading_3,not_writing_1,
not_writing_3,reading=2,reading_2,w_index=2,writing_2] route=[coh_lost2,
start_writing_2,start_reading_2,r12,w0312,w1111,stop_writing_1,
start_writing_1,w0321,r022,w1212,stop_writing_2,start_writing_2,
stop_reading_3,start_reading_3,r13,w0132,r033,w1313,stop_writing_3,
start_writing_3,w0123,w1212,stop_writing_2,start_writing_2,w0112]

Number of markings found (not including bad markings) = 324.
Number of bad markings found = 6.
```

Figure 4.71: Three slot bad states

4.9 Conclusion

In this chapter the Petri net modelling techniques have been introduced which are required for accurate and compact modelling of ACM algorithms, and these techniques have been applied to algorithms which employ one, two and three slots for data communication. The analysis program has been introduced: for a more complete description of this package the reader is referred to Appendix A. The Petri net extension of 'OR' arcs has been introduced, along with how each ACM algorithm is broken down and modelled by each instruction rather than abstract models. The ACM operating properties of data coherence, data freshness and data sequencing have been defined, and these have been used to analyze the one, two and three slot mechanisms. All the techniques described in this chapter are utilized in the next chapter which is concerned with the analysis of fully asynchronous ACMs which have a single writing process and a single reading process.

Chapter 5

Analysis and comparison of single writer / single reader atomic variables

5.1 Introduction

This chapter presents and compares ACM algorithms and the analysis results for single writer, single reader mechanisms using four slots (and one algorithm with five slots). Figure 5.1 shows a table for the comparison of the different algorithms (including the self timed three slot for comparison). The key for the comparison is given in Figure 5.2. Traditionally the correctness of each algorithm has been proved by its author using a method preferred by that author. This work analyses each algorithm using a common Petri net technique, reviews the algorithm, and tabulates the analysis under the operating properties of data coherence (Section 4.4.2), data freshness (Section 4.4.3) and data sequencing (Section 4.4.4).

5.1.1 Properties used in the comparison

To assess full correctness and to compare algorithms, a number of properties have to be investigated. The main properties are the operating properties of data coherence, data freshness and data sequencing (Sections 4.4.2, 4.4.3 and 4.4.4). If an algo-

	Xia3	Pe9	Kr39	Tr14	An9	Si7	Si6
No. of slots	3	5	4	4	4	4	4
in pairs	No	No	No	Yes	Yes	Yes	Yes
weighting	No	No	‘min’	No	No	No	No
waiting	Yes	No	No	No	No	No	No
data copies	1	2 or 3	1	1	1	1	1
control vars.	mod. 3	bin.	mod. 3	bin.	bin.	bin.	bin.
complex ops.	No	Yes	Yes	Yes	No	No	No
var. fanout	No	Yes	Yes	No	No	No	No
redundant inst.	No	No	Yes	No	No	No	No
bits used	$3b+6$	$5b+9$	$4b+39$	$4b+14$	$4b+9$	$4b+7$	$4b+6$
vars. shared	2	4	6	8	4	4	5
bits shared	4	4	18	8	4	4	5
HW/SW/both	both	SW	SW	SW	SW	both	both
WR instruct.	3	8	10	9	6	3	3
RD instruct.	2	11	18	9	4	3	3
WR $\rightarrow$ ind.	1	-	3	1	2	2	2
Coh/Fre/Seq.	P/P/P	F/X/X	P/P/P	P/P/P	P/P/P	P/P/P	P/P?/F

Table 5.1: Comparison of ACM algorithms (P = Pass, F = Fail, X = Untested).

Algorithm	Key	Section
Xia self timed 3 slot [XYS$^+$00]	Xia3	4.8
Peterson ‘$5b+9$’ [Pet83]	Pe9	5.2
Kirousis ‘$4b+39$’ [KKV88]	Kr39	5.3
Tromp ‘$4b+14$’ [Tro89]	Tr14	5.4.1
Anderson ‘$4b+9$’ [AG92]	An9	5.5
Simpson ‘$4b+7$’ [Sim90b]	Si7	5.6.1
Simpson ‘$4b+6$’ [Sim97b]	Si6	5.6.2

Table 5.2: Key for Table 5.1.

rithm/mechanism cannot maintain these then it cannot be classed as an ACM. The remaining properties are used to indicate physical characteristics, such as the size of the control variables etc. Some are described here.

'No. of slots' indicates the number of data slots within the algorithm; only 3 and 4 slot mechanisms are presented here.

'In pairs' refers to how the slots are arranged: some algorithms with four slots arrange them in two pairs of slots to enable binary variable access.

'Weighting' indicates whether the slots have equal priority; algorithms can either try to steer away from the reading process, others try to choose the lowest valued available slot (Kirousis uses a 'minimum' function).

'Data copies' indicates how many copies of the data to be transmitted are written into the mechanism. Many algorithms write only one, but several have been found in the literature which write multiple copies of the data, then do some form of processing within the reader process to determine which copy to use.

'Redundant inst.' indicates if any instructions within the algorithm have been found to be redundant and not used.

The 'control vars' entry is used to compare the range of the control variables within the algorithm. Ideally binary variables should be used, but it has been found that some authors have opted for more complicated variables, such as ternary ('$\{1, 2, 3\}$' or '$\{0, 1, 2\}$') listed as 'mod 3' in the table.

'Complex ops.' indicates whether complex operators have been used, rather than sticking to simple operations such as assignment and use of logical operators. Kirousis (Section 5.3) uses a 'minimum' function to determine the lowest available slot number.

'Var. fanout' indicates whether the control variables are used only once by the other process or their value is copied multiple times, and therefore fanned out. This can cause problems if metastability occurs because during settling of a metastable transient different copies may take different values.

'Bits used' indicates the number of binary bits required to implement the whole algorithm. The format of this entry is based on '$nb + m$' [Tro89], where 'n' indicates the number of slots within the algorithm, 'b' indicates the number of bits used to make a single slot (this is application dependent) and 'm' indicates the number of bits used for the control algorithm. A graph comparing the number of bits used by each algorithm according to the size of each data slot is given in Figure 5.1.

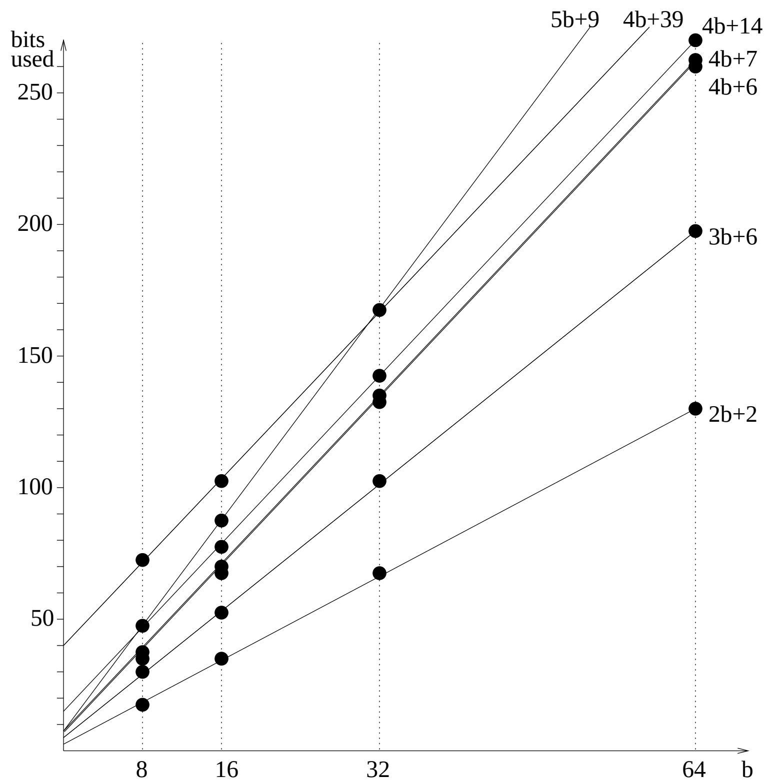


Figure 5.1: Graph comparing the number of bits used by each algorithm with differing sized data slots.

'Vars. shared' indicates the number of variables which are shared between the writing and the reading processes.

'Bits shared' indicates the number of bits shared between the writing and the reading processes. This value will be the same as 'vars. shared' if the algorithm uses binary control variables.

'HW/SW/both' indicates whether the algorithm was designed for implementation in hardware, software or both.

'WR instruct.' indicates the number of instructions used to make the writer process.

'RD instruct.' indicates the number of instructions used to make the reader process.

'WR $\rightarrow$ ind.' indicates how many instructions are in between the writer finishing writing the data into the mechanism and this fact being indicated to the reading process. (The Peterson algorithm does not indicate a specific write because multiple copies of the data are used, in the table a '-' is entered to indicate this fact).

'Coh/Fre/Seq.' gives the results from the operating property tests. A 'P' indicates that the algorithm has passed the test and therefore maintains the operating property. A 'F' indicates a fail, and the property is not maintained. A 'P?' indicates that the algorithm can only maintain the modified property when a timing constraint is enforced. An 'X' indicates the algorithm was not tested for this operating property (probably because the algorithm does not maintain data coherence).

Each algorithm presented here is reproduced in its original form as published by each author. Each algorithm is also presented re-written in a Pascal like format for easier reading, which also aids the comparison process. This Pascal like format is not designed for direct implementation in software as some instructions need to be broken down into machine instructions. These re-written algorithms present one way of implementing the algorithms, there may be alternatives. Some algorithms are also presented in an algebraic format which is better suited to hardware implementation, again, alternatives may be possible.

Each algorithm compared in this chapter is accompanied with a brief description of its operation, and the variables it employs. For a full description of each algorithm the reader is referred to the algorithms original publication.

5.2 Peterson's algorithm

Peterson [Pet83] was one of the first authors to propose the problem of concurrent reading while writing without the need for waiting by either process. Peterson presents several solutions which involve waiting by one of the processes if the other process is accessing the data. This is implemented by Peterson using loops which run continuously until access is gained. These solutions are not suitable in real-time applications because the waiting periods cannot be defined. Peterson's main solution (solution 3) does not involve any process waiting, but as the author admits, the "deficiencies of Solution 3 is that the writer, during a single pass, may have to make n copies (of the data) for the readers". This solution is quoted for the multiple readers case, and the variable n refers to the number of readers the algorithm is servicing. The algorithm uses multiple copies of the same data to be communicated. The writer produces $n+2$ copies of the data, which in the case of a single reader application is therefore three copies of the data. Peterson's algorithm is reproduced here as published in Figure 5.2, and for the single reader case uses $5b+9$ bits, where b refers to the number of bits required to store the data in each slot/buffer. There are five slots within the algorithm for a single reader, but only three of them are shared. Two of the slots are local to the reader for keeping two local copies of the data stored at $buff1$ and $buff2$. Local copies of the data need to be taken by the reading process because the algorithm does not steer the writing and reading processes away from each other, rather it senses concurrent access to a single slot, and then uses a backup copy of the data. Because this chapter is only concerned with single reader ACM algorithms, the algorithm has been re-written by this author to remove the loop for making the multiple copies of the data for the readers, it has also been put into a more standard format. This re-written algorithm is given in Figure 5.3, and is also shown in time line format in Figure 5.4. The variables used within this re-written version are given in Table 5.3.

Writer	ith Reader
$wflag$:= **true**;	$reading[i]$:= **not** $writing[i]$;
write $buff1$	$sflag$:= $wflag$;
$switch$:= **not** $switch$;	$sswitch$:= $switch$;
$wflag$:= **false**;	*read* $buff1$;
for j := 1 **step** 1 **until** n **do**	$sflag2$:= $wflag$;
if $reading[j] \neq writing[j]$ **then**	$sswitch2$:= $switch$;
begin	*read* $buff2$;
write $copybuff[j]$;	$bad1$:= $sswitch \neq sswitch2$
$writing[j]$:= $reading[j]$;	**or** $sflag$ **or** $sflag2$;
end;	**if** $reading[i] = writing[i]$ **then**
write $buff2$;	**return** $copybuff[i]$ *value*;
	elseif $bad1$ **then return** $buff2$ *value*;
	else return $buff1$ *value*;

Figure 5.2: Peterson's original algorithm.

Peterson's algorithm writes multiple copies of the data to be transmitted in one cycle of the algorithm. Rather than steer each process to a specific slot, each slot is written in the same order in each cycle. The algorithm detects when a read and write have occured on the same slot, and another copy of the data is used. The

```
mechanism peterson algorithm;

          var buff1, buff2, copybuff : data := null, null, null;
              switch,wflag, writing, reading : bit := 0, 0, 0, 0;

          procedure write (item : data);
          begin
              wflag := 1;                                        {w0}
              buff1 := item;                                     {w1}
              switch := not switch;                              {w2}
              wflag := 0;                                        {w3}
              if reading = writing then                          {w4}
                     copybuff := item;                           {w5}
                     writing := reading;                         {w6}
              endif;                                             {w7}
              buff2 := item;                                     {w8}
          end;

          function read : data;
          var buff1_copy, buff2_copy : data;
              bad1, sswitch, sswitch2, sflag, sflag2 : bit := 0, 0, 0, 0, 0;
          begin
              reading := not writing;                            {r0}
              sflag := wflag;                                    {r1}
              sswitch := switch;                                 {r2}
              buff1_copy := buff1;                               {r3}
              sflag2 := wflag;                                   {r4}
              sswitch2 := switch;                                {r5}
              buff2_copy := buff2;                               {r6}
              bad1 := sswitch ≠ (sswitch2 OR sflag OR sflag2);   {r7}
              if reading = writing then                          {r8}
                     output := copybuff;                         {r9}
              elseif bad1 then output := buff2_copy;             {r10}
              else output := buff1_copy;                         {r11}
          end;
end.
```

Figure 5.3: Peterson's $5b + 9$ algorithm re-written.

Variable name	Purpose
Shared reader variables	
$reading$	used with $writing$ to protect $buff1$
Shared writer variables	
$switch$	alternates at the end of each $buff1$ write
$wflag$	indicate $buff1$ being written
$writing$	used with $reading$ to protect $buff1$
Local reader variables	
$bad1$	indicates if $buff1$ was written to while being read
$sswitch$	readers first copy of $switch$, used to determine $bad1$
$sswitch2$	readers second copy of $switch$, used to determine $bad1$
$sflag$	readers first copy of $wflag$, used to determine $bad1$
$sflag2$	readers second copy of $wflag$, used to determine $bad1$
Local writer variable	
none	the writer has no local variables

Table 5.3: Variables used within the Peterson $5b + 9$ algorithm.

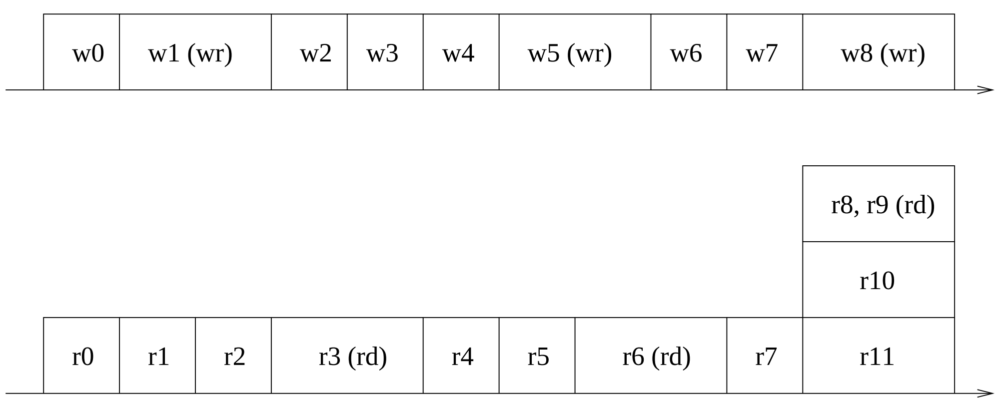


Figure 5.4: Peterson's $5b + 9$ algorithm time line.

writing algorithm writes three slots, called $buff1$, $buff2$ and $copybuff$, these slots are shared with the reader. The reader keeps two local copies of $buff1$ and $buff2$. The setting and copying of the control variables is used to determine if a read and write have occured at the same time on $buff1$. This information is then used to set the reader variable $bad1$ which along with the variables $reading$ and $writing$ is used to determine which copy of the data is returned by the reader process.

The traditional test for data coherence (Section 4.4.2) is not sufficient in this case. The algorithm does not steer each process, instead it senses if a concurrent access to a slot has occured and uses a different backup copy of the data. For this algorithm the data coherence test is modified to that shown on the time line in Figure 5.5. This test senses if there is a concurrent access to $buff1$ ($w1$ and $r3$) and flags it, and also to $buff2$ ($w8$ and $r6$). The standard data coherence check is used on $copybuff$. If instruction $r10$ is executed, returning the value of $buff2$, and $buff2$ is marked with a concurrent access, then coherence has not been maintained. Likewise if $r11$ is executed and $buff1$ is marked with a concurrent access.

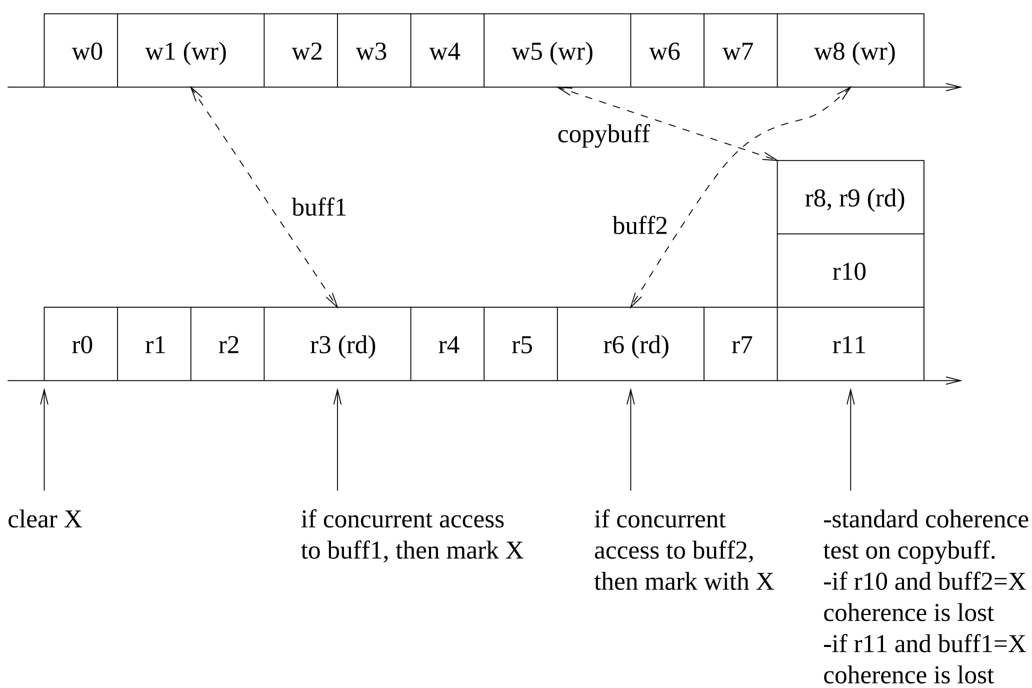


Figure 5.5: Peterson's $5b + 9$ data coherence test time line.

The Petri net model of Peterson's $5b + 9$ algorithm with the modified data coherence test is given in Appendix B.1.1. It has 53 transitions and 53 places which produces a state space of 54460 states plus 1280 bad markings, indicating that this algorithm cannot maintain the data coherence property. Investigation of these bad states shows that coherence is lost on all slots in the algorithm, and that there are no specific interleaving patterns which cause the failure. Many of the failures are caused by interleavings within a single execution cycle of each accessing process. Because no interleaving patterns emerged from the analysis, no timing constraints have been applied to this algorithm and it has not been tested for data freshness or data sequencing.

5.3 Kirousis's $4b+39$ algorithm

The algorithm by Kirousis, Kranakis and Vitanyi [KKV87b, KKV87a, KKV88], is by far the most complex algorithm included in this comparison. The complexity of such large algorithms hampers efficient implementation in hardware. Several solution algorithms are presented by the authors, starting with an algorithm which uses "finitely many layers", then gradually improving the algorithm to one which requires only three layers, and therefore uses modulo three variables, which require multiple bits in hardware (layers are used within the algorithm to pass control information between the two processes). The authors claim that "the problem of simultaneous memory access needs to be solved without recourse to synchronization primitives ... solution involves no waiting ... making no assumptions about the relative durations of the readings and writings." The authors claim that "the behaviour of the building units (flip-flops) under simultaneous access is irrelevant for the final output." Then they give a footnote which gives the reader a very brief understanding of metastable operation, but go on to state - "We do not consider this level of physical detail here, but consider idealized bistables." Throughout the text describing their analysis method they give several statements which relate to data coherence and data sequencing, but no mention of a test like that of data freshness. Their published description and understanding of ACM algorithm properties is thus inadequate. Of the various solutions they propose, only the final, one writer, one reader algorithm/protocol is analyzed here, (see Figure 5.6 and Figure 5.7 for the time line). The algorithm is also presented in Figure 5.8 in a software form. It should be noted that two extra local variables have been added in this software form. The two extra variables are 'rq_{wl}' and 'a_{rl}' which store local copies of the shared variables 'RQ_{wl}' and 'A_{rl}' (These extra variables make the algorithm larger than the quoted $4b+39$). These copies are taken for safe mechanism operation because only local copies of shared variables should be used (Section 6.1.5). The algebraic form of Kirousis's algorithm is identical to that presented in Figure 5.8.

The writer writes a word v:
1. read RQ_{wl};
2. **if** $RQ_{wl} = P$ **then do** $wl := wl + 1; vb := \{wt, wt^-\}$ **od**;
3. $wt^- := wt; wt := min(\{1, 2, 3, 4\} - (\{wt^- \cup vb\}))$;
4. write the bits of v onto the track with number wt;
5. $wtg := wtg + 1$; write E on A_{wl+1};
   /*the writer clears one layer ahead before writing*/
   write(wtg, wt) on A_{wl};

The reader returns a word:
1. $rl := rl + 1$;
2. read A_{rl};

3a. **if** $A_{rl} = E$ **then do** $rl := rl - 1$; read A_{rl};
   if ($A_{rl} \neq E$ and tag of $A_{rl} = rtg + 1$ **then**
   store value of A_{rl} into (rtg, rt) **od**

3b. **else do**
   store the value of A_{rl} into s;
   write E on RQ_{rl+1};
   /*the reader clears ahead one layer before printing a P*/
   write P on RQ_{rl}; read A_{rl};
   if $A_{rl} \neq E$ **then** store the value of A_{rl} into (rtg, rt) **else**
   (rtg, rt):= s **od**;

4. read the bits of track with number rt and return the word thus obtained.

Figure 5.6: Kirousis $4b + 39$ algorithm (with modulo 3 arithmetic).

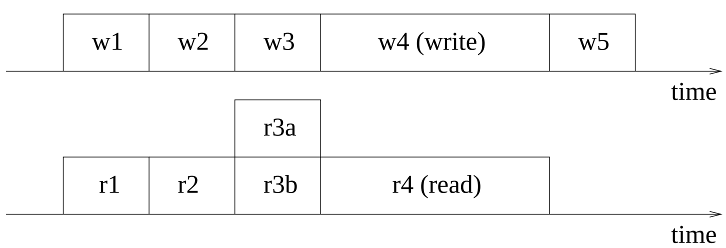


Figure 5.7: Time line showing Kirousis's $4b + 39$ algorithm.

The algorithm uses thirty nine bits for the control algorithm, which is higher than the number of variables in the algorithm because they are multi-value variables and not binary. The variables are listed in Table 5.4 (vb[1]). The variables 'A' and 's' hold two variable values, the first variable is the 'tag' value ({0,1,2}, which is modulo 3) and the second is the 'track' value ({1,2,3,4}). The 'tag' value is used to convey

[1]vb - from verboten, which is German for forbidden

```
mechanism kirousis four track;
    var TRACK : array[1..4] of data := (null, null, null, null);
        RQ : array[0..2] of {P, E} := (E, E, E);
        A : array[0..2] of {(E), (0, 1), (0, 2), (0, 3), (0, 4), (1, 1), (1, 2), (1, 3), (1, 4),
                                  (2, 1), (2, 2), (2, 3), (2, 4)} := (E, E, E);
    procedure write (item : data);
    var wl, wtg : {0,1,2} := 0,0;
        wt, wt⁻ : {1,2,3,4};
        vb : array[bit] of {E,1,2,3,4} := (E, E);
    begin
        rq_wl := RQ_wl;
        if rq_wl = P then
          wl := wl + 1;
          vb := {wt, wt⁻}; endif
        wt⁻ := wt;
        wt := min({1,2,3,4} - ({wt⁻} ∪ vb));
        TRACK[wt] := item;
        wtg := wtg + 1;
        A_wl+1 := E;
        A_wl := (wtg, wt);
    end;

    function read : data;
    var s : {(E),(0,1),(0,2),(0,3),(0,4),(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4)};
        rt : {1,2,3,4};
        rl : {-1,0,1,2} := -1;
        rtg : {0,1,2} := 0;
    begin
        rl := rl + 1;
        a_rl := A_rl;
        if a_rl = E then
          rl := rl - 1;
          a_rl := A_rl;
          if(a_rl ≠ E) and (a_rl^tag = rtg + 1) then
            (rtg, rt) := a_rl; endif;
        else
          s := a_rl;
          RQ_rl+1 := E;
          RQ_rl := P;
          a_rl := A_rl;
          if a_rl ≠ E then
            (rtg, rt) := a_rl;
          else
            (rtg, rt) := s; endif
        endif
        read := TRACK[rt];
    end;
end.
```

Figure 5.8: Kirousis's $4b + 39$ algorithm rewritten in a PASCAL like form.

which layer {0,1,2} the valid control variables are located, and the 'track' variable indicates which track contains the freshest written and indicated data. The variables wl and wtg are both modulo three, so the statement $wl := wl + 1$ will run in the sequence '$0 \to 1 \to 2 \to 0$ etc.'. The variable rl is also modulo three, but it has an initial condition of '-1'. This means the instruction '$rl := rl + 1$' has the sequence '$-1 \to 0 \to 1 \to 2 \to 0$ etc.', and the instruction '$rl := rl - 1$' has the sequence '$2 \to 1 \to 0 \to 2$ etc.'.

Variable	range	initial	Purpose
Shared reader			
RQ_0	$\{P, E\}$	E	P (Please), a request to
RQ_1	$\{P, E\}$	E	the writer to write its track
RQ_2	$\{P, E\}$	E	number to A, or E (Empty)
Shared writer			
A_0	$\{E \cup \{(0,1,2)\text{X}(1,2,3,4)\}\}$	E	indicate writer's current
A_1	$\{E \cup \{(0,1,2)\text{X}(1,2,3,4)\}\}$	E	tag and track
A_2	$\{E \cup \{(0,1,2)\text{X}(1,2,3,4)\}\}$	E	
Local reader			
rt	$\{1,2,3,4\}$	none	reader's track
rl	$\{-1,0,1,2\}$	-1	reader's layer
rtg	$\{0,1,2\}$	0	reader's tag
s	$\{E \cup \{(0,1,2)\text{X}(1,2,3,4)\}\}$	none	reader's copy of A
Local writer			
wl	$\{0,1,2\}$	0	writer's layer
vb	$2\text{X}\{E,1,2,3,4\}$	$\{E, E\}$	tracks not to be written to
wt	$\{1,2,3,4\}$	none	writer's track
wt^-	$\{1,2,3,4\}$	none	last written track
wtg	$\{0,1,2\}$	0	writer's tag

Table 5.4: Multi-value variables used within the Kirousis algorithm.

Using multi-value variables in a concurrent environment is not good design practice, because the individual bits which make up the variable may change at slightly different times, and therefore an intermediate, or erroneous value may be obtained by the sampling process.

The Petri net models for this Kirousis algorithm are given in Appendix B.2. Specifically the Petri net model to test coherence is given in Section B.2.1. For this high level test, it has been assumed that multi-bit variables do not cause intermediate value problems. This model had 293 transitions, and 149 places. Each instruction has been modelled using an atomic event, this is safe because no instruction references and sets shared variables at the same time. This model produces a state space which requires 32MB to store during the reachability search, and is therefore quite large (this value includes all the route information for the entire state space). The state space has 334263 states (markings with 'enable', markings without 'enable' were 1201211), of which 0 were *bad*. Therefore at this high level it can be seen that this algorithm maintains the data coherence property. Care must however be taken because if a slightly different initial marking is used then for the first few cycles of the algorithm coherence is lost (part of the initial marking changes from $[v1 = 1, v2 = 1, wt^{-} = 1, wt = 1]$ to $[v1 = 4, v2 = 4, wt^{-} = 4, wt = 4]$). This changes the state space to 334263 (1203320 without enable) plus 4 bad states.

The Kirousis model for checking the data freshness property is given in Section B.2.2. The model is shown as a time line in Figure 5.9. This model has 336 transitions and 176 places, and produces a state space with 412085 (1679863 without enable) states plus 35 states which are bad. This indicates that the algorithm does not maintain the data freshness operating property (using the amended model for data freshness). As a check, Simpson's original freshness definition was also used, producing a model with 328 transitions with 175 places, giving a state space containing 412085 (1656063 without enable) states plus 35 bad states. This indicates that the algorithm does not maintain the unmodified definition of data freshness either.

These results were compared and analyzed. The 35 *bad* states in each case have almost identical routes, the only difference being the addition of the modification to the freshness testing algorithm. All 35 states fail freshness on slot 1, and all failures are during the first read access to the ACM. This indicates that this is a problem with the initialization of the algorithm under certain startup/setup interleaving conditions. No other freshness failures occur, therefore it can be concluded that the algorithm can fail during the first reader process cycle, which might not be serious since once the algorithm is up and running, freshness is maintained.

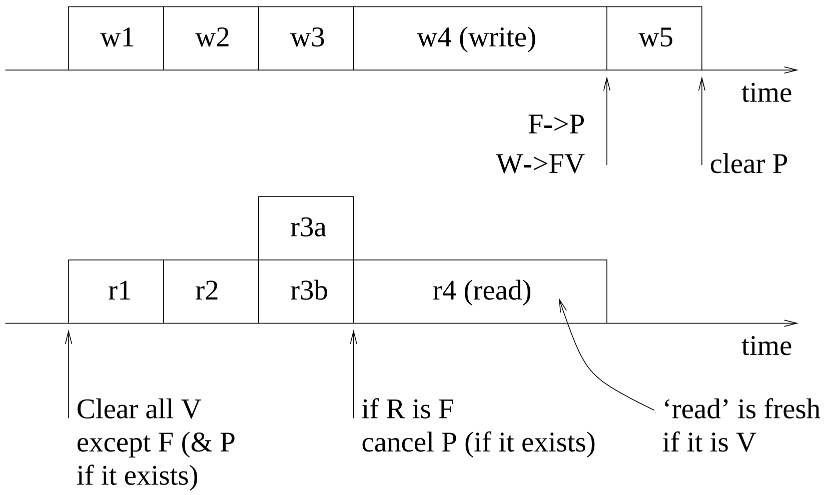


Figure 5.9: Time line showing Kirousis's algorithm under the data freshness test.

The Kirousis model for checking the data sequence property is given in Section B.2.3. This model is shown as a time line in Figure 5.11. This model has 441 transitions with 234 places. This produces a state space with 377352 (1986349 without enable) states with 0 bad states. This indicates that the algorithm does maintain the data sequencing property. However care must be taken with the initial condition again. If the initial condition of $[v1 = 4, v2 = 4, wt^{-} = 4, wt = 4]$ is used then the state space produces 378996 states (1993295 without enable) plus 105 bad states.

The analysis of this result shows that the Kirousis algorithm has problems with the interleaving of the instructions which indicate new data, choosing new data to be read and data being over written in the first few cycles of operation if the initial conditions are not correct. One bad route from this result is shown on a time line in Figure 5.10. These time lines show the specific instructions interleavings for this bad route. The top set of time lines in Figure 5.10 show the roles for data sequencing and show that the re-writing of slot one causes the failure. The lower set of time lines show the same time lines but with the data freshness roles indicated. This shows why it is necessary to check for both data sequencing and data freshness. The problem here lies in the fact that the reader has chosen slot one, but before reading slot two is written and then slot one is overwritten. Therefore the reader obtains the new data in slot one, before it is indicated properly, and the writer is still indicating slot two as the latest data. The reader then chooses slot two, the writer writes to slot three, and the read process then tries to read slot two. This causes the data sequencing error. The *reach* coverage data (Section A.2) shows that slot two is the only slot to lose data sequencing, this is due to the *min* function used to choose the next available slot for the writing process.

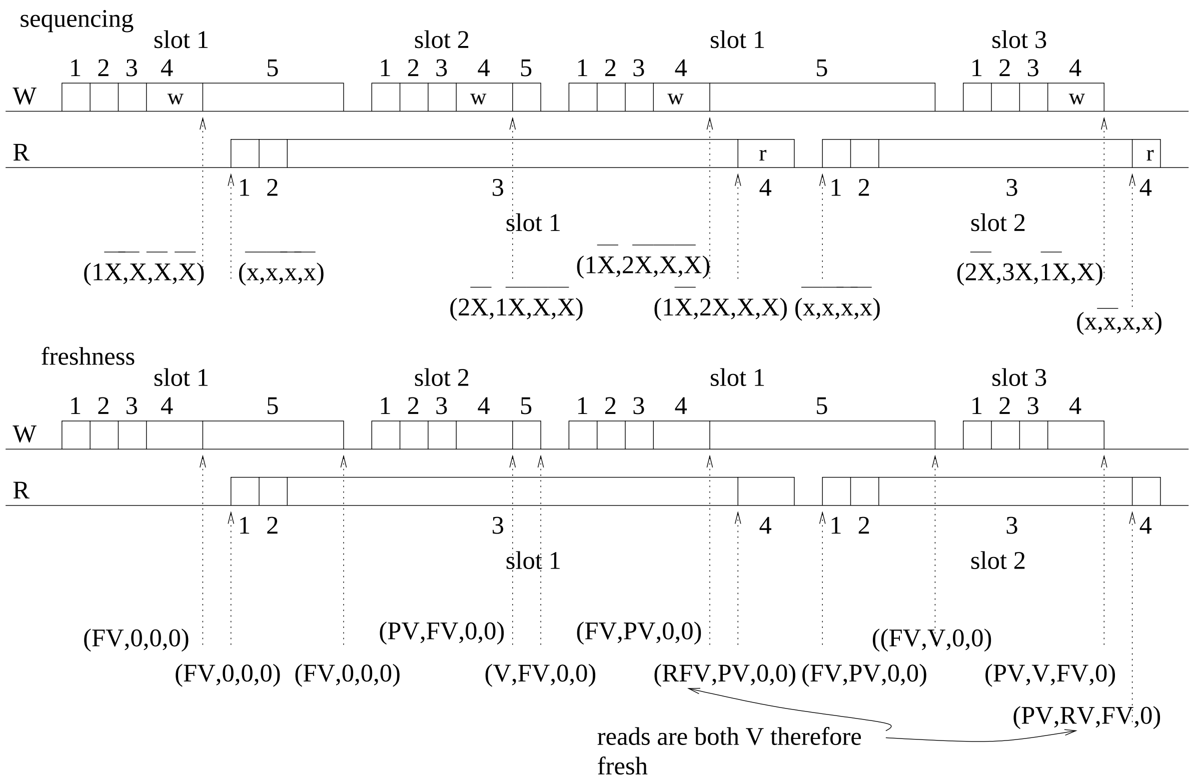


Figure 5.10: Time lines show the instruction interleavings on a bad route with data sequencing and data freshness roles highlighted.

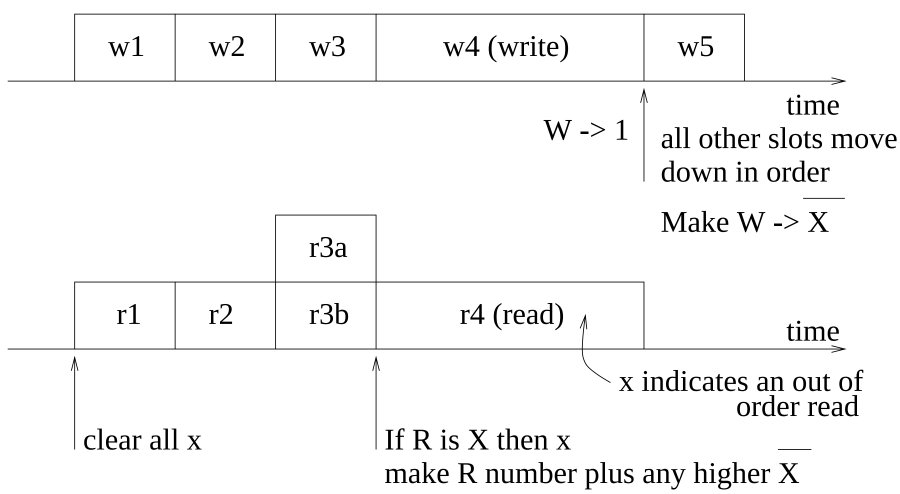


Figure 5.11: Time line showing Kirousis's algorithm under the data sequence test.

The coverage data of the Kirousis Petri net model highlights that some parts of the algorithm are redundant or never executed. (Coverage data is output from *reach* and shows how many times each transition has fired during the construction of the complete state space, Section A.2). In the case of the Kirousis algorithm it highlights that within the reader instruction '$3a$' $A_r l$ is always equal to 'E' therefore the second 'IF' statement is never true. Also within reader instruction '$3b$', at the last 'IF' statement $A_r l$ is never equal to 'E' therefore the 'IF' statement is redundant.

No metastable operating analysis has been investigated because the size of the algorithm is excessively large for implementation in hardware (compared to other algorithms analyzed in this thesis).

5.4 Tromp's algorithms

Tromp's work [Tro89] starts with the construction of an atomic bit from three safe bits (or latches), which he then incorporates into an ACM algorithm which uses four of his atomic bits for the shared control variable communication between processes. This ACM algorithm used '$4b + 17$' bits, although Tromp claims it uses '$4b + 12$' (Tromp's size claim lists only the bits for the slots and the shared control variables and not the total local variables). During his analysis he claims that some internal parts of the atomic bits are redundant and produces a final algorithm which does not use atomic bits in its construction. This final algorithm which uses '$4b + 14$' (Tromp claims '$4b + 8$') is presented here.

5.4.1 Tromp's $4b + 14$ algorithm

Tromp's main construction has eight shared bits, six controlled by the writer and two controlled by the reader. These are shown in Table 5.5, along with the local variables. The algorithm uses no complex operators but unfortunately does use 'IF' statements. An 'IF' statement can introduce fatal errors (i.e. in the statement "if A then B else C", metastability of A leads to uncertainty over whether the process executes B or C, see Section 6.1.2). Tromp's main algorithm is given in Figure 5.12. This algorithm is also given in Figure 5.13 rewritten in a more standard format, and in algebraic form in Figure 5.14. The control variables are binary therefore $w := 1 - w$ is simply $w := \overline{w}$. *Change* is Tromp's high level *not* operator for array variables. The <> operator means ‘not equal’. The operator $\oplus$ is ‘exclusive-OR’. Each instruction in the model can safely be modelled using atomic events because no instructions reference and set shared variables at the same time. This algorithm is shown in time line format in Figure 5.15.

Writer protocol

if $R[1-w] == W[1-w]$ then
  $w := 1 - w$
  write track $T[w, x[w]]$
  change $W[1-w]$
else
  $x[w] := 1 - x[w]$
  write track $T[w, x[w]]$
  change $D[w, 1 - x[w]]$
endif

Reader protocol

if $W[r] <> R[r]$ then
  change $R[r]$
  $r := 1 - r$
  read $d[0] := D[r, 0]$
  read $d[1] := D[r, 1]$
else
  read $d[d[0] \oplus d[1]] := D[r, d[0] \oplus d[1]]$
endif
read track $T[r, d[0] \oplus d[1]]$

Figure 5.12: Tromp’s $4b + 14$ algorithm.

The Petri net model for testing data coherence is given in Section B.3.1. This model has 96 transitions and 53 places, producing a state space with 13712 states plus 0 bad states. This means the Tromp $4b + 14$ algorithm can maintain data coherence.

The Petri net model for testing data freshness is given in Section B.3.2. This model has 212 transitions and 119 places, producing a state space with 303748 states plus 0 bad states. This indicates that the Tromp $4b+14$ algorithm can maintain data freshness.

The Petri net model for testing data sequence is given in Section B.3.3. This model has 329 transitions and 173 places, and produces a state space with 146751 states plus 0 bad states. Indicating that the Tromp $4b + 14$ algorithm can maintain

```
mechanism tromp four slot;

        var T : array[bit, bit] of data := ((null, null),(null, null));
            D : array[bit, bit] of bit := (0, 0);
            R, W : array[bit] of bit := 1, 1;

        procedure write (item : data);
        var x : array[bit] of bit;
            w : bit;
        begin
            if R[not w] = R[not w] then
              w := not w;
              T[w, x[w]] := item;
              W[not w] := not W[not w];
            else
              x[w] := not x[w];
              T[w, x[w]] := item;
              D[w, not x[w]] := not D[w, not x[w]];
            endif;
        end;

        function read : data;
        var d : array[bit] of bit;
            r : bit;
        begin
            if W[r] <> R[r] then
              R[r] := not R[r];
              r := not r;
              d[0] := D[r,0];
              d[1] := D[r,1];
            else
              d[d[0] ⊕ d[1]] := D[r,d[0] ⊕ d[1]];
            endif
            read := T[r, d[0] ⊕ d[1]];
        end;
end.
```

Figure 5.13: Tromp's $4b + 14$ algorithm rewritten.

Writer	Reader
$w1$: if $R[\overline{w}] == W[\overline{w}]$ then	$r1$: if $W[r] \neq R[r]$ then
$w2$: $\quad w := \overline{w}$	$r2$: $\quad R[r] := \overline{R[r]}$
$w3$: $\quad T[w, x[w]] := input$	$r3$: $\quad r := \overline{r}$
$w4$: $\quad W[\overline{w}] := \overline{W[\overline{w}]}$	$r4$: $\quad d[0] := D[r, 0]$
$w5$: else	$r5$: $\quad d[1] := D[r, 1]$
$w6$: $\quad x[w] := \overline{x[w]}$	$r6$: else
$w7$: $\quad T[w, x[w]] := input$	$r7$: $\quad d[d[0] \oplus d[1]] := D[r, d[0] \oplus d[1]]$
$w8$: $\quad D[w, \overline{x[w]}] := \overline{D[w, \overline{x[w]}]}$	$r8$: endif
$w9$: endif	$r9$: $output := T[r, d[0] \oplus d[1]]$

Figure 5.14: Tromp's $4b + 14$ algorithm in algebraic form.

Variable name	Purpose
Shared reader variables	
R_0, R_1	Indicate the reader's group
Shared writer variables	
$D_{0,0}$, $D_{0,1}$ $D_{1,0}$, $D_{1,1}$	Group 0 track display Group 1 track display
W_0, W_1	Indicate the writer's group
Local reader variables	
r $d[0]$, $d[1]$	reader's local group reader's local track variables
Local writer variables	
w $x[0]$, $x[1]$	writer's local group writer's local track variables

Table 5.5: Variables used within the Tromp $4b + 14$ algorithm.

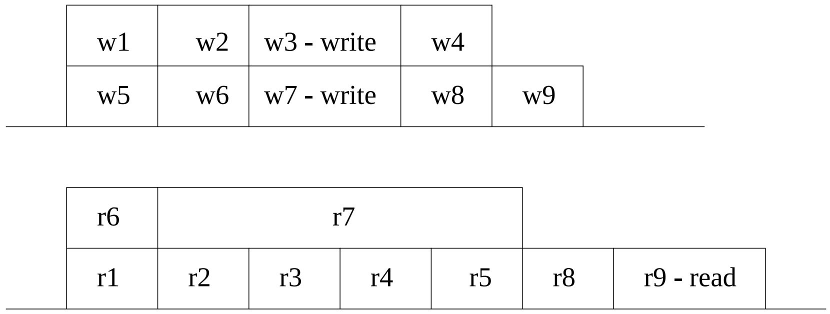


Figure 5.15: Tromp's $4b + 14$ algorithm time line.

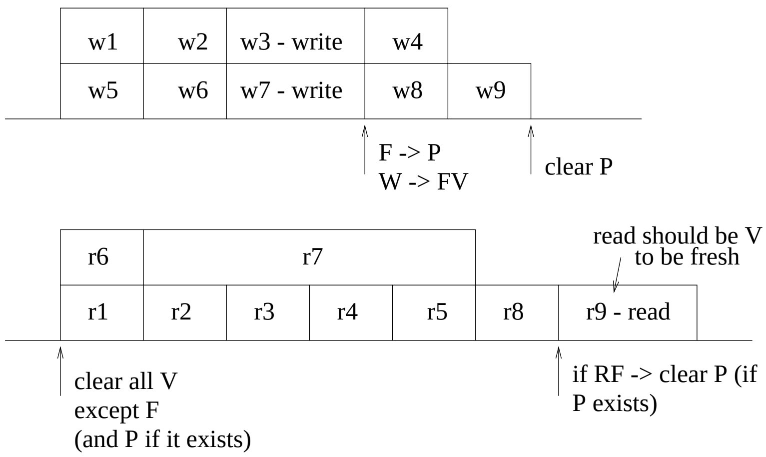


Figure 5.16: Tromp's $4b + 14$ algorithm time line for data freshness testing.

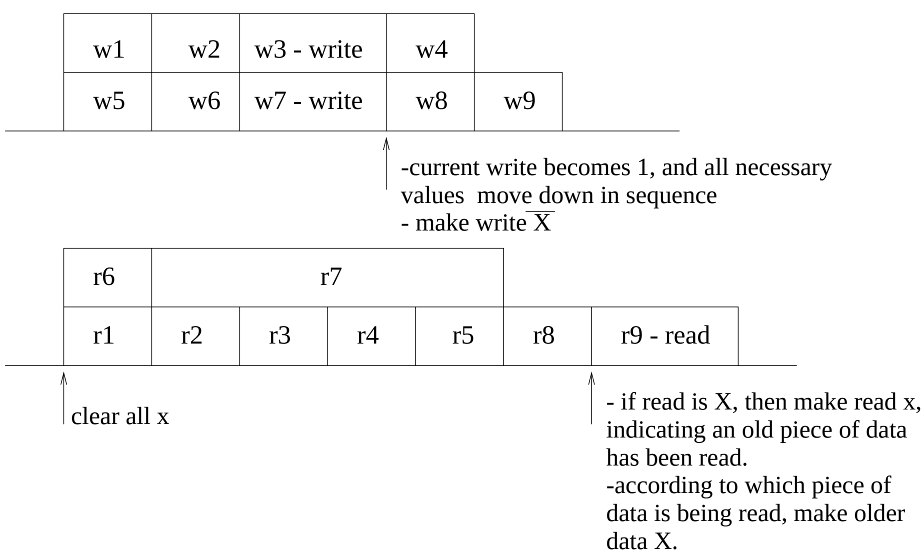


Figure 5.17: Tromp's $4b + 14$ algorithm time line for data sequence testing.

data sequence.

5.5 Anderson's algorithm

Anderson [AG92] presents his "criterion for atomicity" which includes a list of six properties for the design of an ACM (these are discussed in Section 6.1). From these design properties some example programs which comply with these criterion are presented along with a solution ACM algorithm. This algorithm is reproduced here in Figure 5.18. The algorithm seems from an initial inspection to be quite long and complex, however it is not. The instructions inside the 'do' loops of each process are concerned with encoding the data to be transmitted inside a data slot. This encoding uses two bits in a slot to encode one data bit in the message being transmitted. Two consecutive bits encoded as '(1,0)' are used to represent a logic 0, the bits '(1,1)' are used to represent a logic 1, and the bits '(0,-)' are used to represent the end of a string. This data encoding is not required for this comparative study, and therefore the algorithm can be re-written to use more standard write and read data slot instructions. This re-writing of the algorithm without data encoding simplifies the algorithm to that shown in Figure 5.19, which is written in a more standard form. The re-written form of the algorithm can be simplified by merging some of the instructions, but following Anderson's "atomicity criterion" (Section 6.1), a single instruction should not read and write a shared variable, therefore these reductions have not been implemented here. The symbols used within the original algorithm and Figure 5.18 do not follow standard meanings. The symbol '$\oplus$' usually represents the XOR (exclusive OR) function, however, in this algorithm the '$\oplus$' symbol is used to represent a modulus two addition, which is standard one bit binary addition, and simply performs a logical 'not' function in this algorithm. The algorithm employs four slots for the data transmission, and Anderson claims that the algorithm can handle a slot size up to infinity (although this author cannot see the point of transmitting a piece of data which is infinite in size). In its re-written form, Anderson's algorithm uses 9 variables, making the algorithm $4b + 9$ in size. The variables used are listed in Table 5.6 with their definition.

This algorithm follows a simple plan of steering the writing process away from

```
type Y type = array[0..2L] of 0..1;
shared var Y : array[0..1][0..1] of Y type;
           Z : array[0..1] of 0..1;
           WP, RP : 0..1;

initialization WP = RP ∧ (∀i, j : 0 ≤ i ≤ 1 ∧ 0 ≤ j ≤ 1 : (∃k : k ≥ 0 : Y[i, j][2k] = 0));

procedure Writer (val : valtype)
private var d, wp, walt : 0..1;
           n : 1..L;
           k : 0..L;
initialization d = wp = WP;
begin
           read d := RP;
           wp, walt, n := d ⊕ 1, walt ⊕ 1, number of bits in val;
           write Y[wp, walt], [2n] := 0;
           k := 0;
           do k < n →
              write Y[wp, walt], [2k] := 1;
              write Y[wp, walt], [2k + 1] := val[k];
              k := k + 1;
           od;
           write Z[wp] := walt;
           write WP := wp;
end;

procedure Reader returns valtype
private var val : valtype;
           ralt, rp, x, y : 0..1;
           k : 0..L + 1;
initialization rp = RP;
begin
           read rp := WP;
           write RP := rp;
           read ralt := Z[rp];
           val, k := 0, 0;
           do k ≤ L →
              read x := Y[rp, ralt], [2k];
              if x = 0 → exit |x ≠ 0 → skip fi;
              read y := Y[rp, ralt], [2k + 1];
              val, k := val + y · 2^k, k + 1;
           od;
           return(val);
end;
```

Figure 5.18: Anderson's algorithm.

```
mechanism anderson four slot;
          var Y = array[bit, bit] of data := ((null, null),(null, null));
              Z : array[bit] of bit := (0, 0);
              WP, RP : bit := 0, 0;

          procedure write (item : data)
          var d, wp, walt : bit := 0, 0, 0;
          begin
              d := RP;                                          {w0}
              wp := not d;                                      {w1}
              walt := not walt;                                 {w2}
              Y[wp, walt] := item;                              {w3}
              Z[wp] := walt;                                    {w4}
              WP := wp;                                         {w5}
          end;

          function read : data
          var ralt, rp : bit := 0, 0;
          begin
              rp := WP;                                         {r0}
              RP := rp;                                         {r1}
              ralt := Z[rp];                                    {r2}
              read := Y[rp, ralt];                              {r3}
          end;
end.
```

Figure 5.19: Anderson's $4b + 9$ algorithm re-written.

Variable name	Purpose
Shared reader variable	
RP	shared reader pair pointer, used to steer writer away from reader
Shared writer variables	
WP	shared writer pair pointer
$Z[0]$, $Z[1]$	shared writer slot pointer
Local reader variables	
rp	local pair pointer
$ralt$	local slot pointer
Local writer variables	
wp	local pair pointer
$walt$	local slot pointer
d	writers initial local copy of RP

Table 5.6: Variables used within the Anderson $4b + 9$ algorithm.

the reading process, and the reading process chases the writing process to get the lastest data. The algorithm is given in time line format in Figure 5.20 and also in an algebraic form in Figure 5.21 which is more suited to implementation in hardware. In Figure 5.21 the symbol '$\|$' is used to mean the instructions can be executed concurrently.

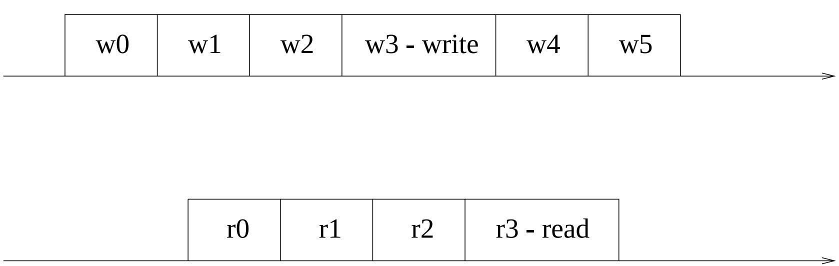


Figure 5.20: Anderson's algorithm as a time line.

The textual Petri net models for Anderson's $4b + 9$ algorithm are given in Appendix B.4. More specifically the model for testing data coherence is given in Section B.4.1 and has 40 transitions and 37 places, producing a state space of 1517 states plus 0 bad states. Therefore the Anderson $4b + 9$ algorithm can maintain the data

Writer	Reader
$w0 : d := RP$	$r0 : rp := WP$
$w1 : wp := \overline{d} \;\|\; walt := \overline{walt}$	$r1 : RP := rp$
$w2 : Y[wp, walt] := input$	$r2 : ralt := Z[rp]$
$w3 : Z[wp] := walt$	$r3 : output := Y[rp, ralt]$
$w4 : WP := wp$	

Figure 5.21: Anderson's algorithm in an algebraic format.

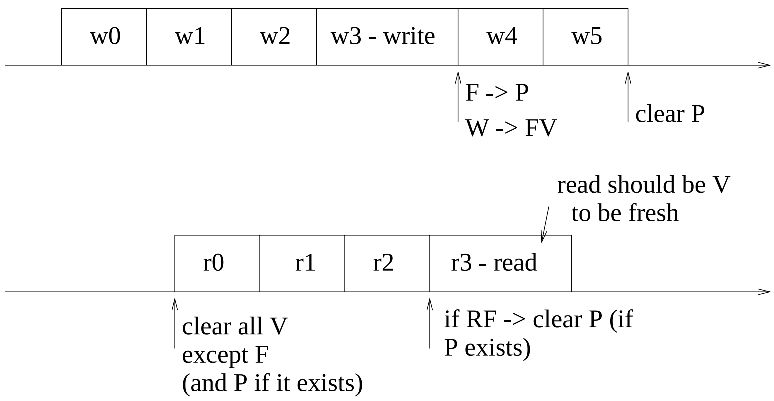


Figure 5.22: Data freshness test for Anderson's algorithm.

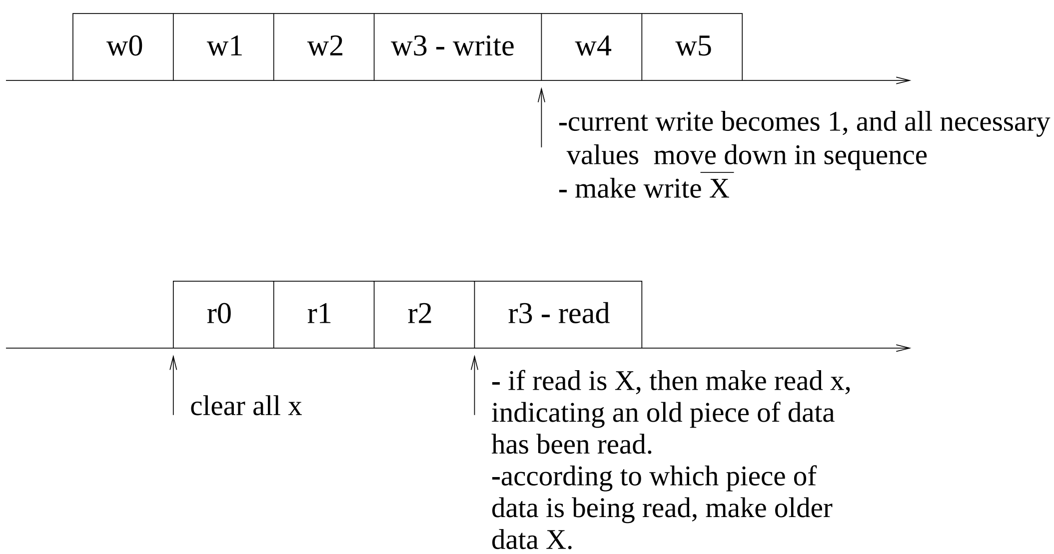


Figure 5.23: Data sequening test for Anderson's algorithm.

coherence operating property.

The test for data freshness is shown on a time line in Figure 5.22 and is given textually in Section B.4.2. The model has 116 transitions and 95 places, which produces a state space with 31885 states plus 0 bad markings. Therefore this algorithm maintains the data freshness property.

The test for data sequencing is shown on a time line in Figure 5.23 and is given textually in Section B.4.3. The model has 200 transitions and 132 places, which produces a state space with 16561 states plus 0 bad markings. Therefore this algorithm maintains the data sequencing property.

5.5.1 Anderson $4b + 9$ under metastable operation

The models of Anderson's $4b+9$ algorithm which are given in Appendix B.4 have been extended with the models given in Figures 3.6 and 3.7. This allows for the analysis of the algorithm under metastable operation. The models have been slightly modified in that only one metastable event is allowed per test, because metastability is a fairly rare event. It has been assumed that each metastable event will have settled by the time the variable is used.

The variables which are shared in this algorithm are RP, WP, $Z[0]$ and $Z[1]$, therefore the variables which sample these variables (d, rp and $ralt$) could become metastable because their input signals are supplied from the other process which is not running synchronized to the same clock. Three models were constructed, one for each variable which may become metastable. The metastable outcome is tested for by the model which checks to see if the signals input has changed during its set-up time. Each of the three models is then tested for data coherence, data freshness and data sequence. The models are not included here, as they have only minor changes from those already presented in Appendix B.4. The results from this analysis are presented in Table 5.7

From these results it can be seen that a single metastable event on a variable which samples a shared variable does not affect the operating properties of the mechanism.

Metastable Variable	Input Variable	Coherence	Freshness	Sequence
d ($w0$)	RP ($r1$)	Pass	Pass	Pass
rp ($r0$)	WP ($w5$)	Pass	Pass	Pass
$ralt$ ($r2$)	$Z[rp]$ ($w4$)	Pass	Pass	Pass

Table 5.7: Shared variable metastability within the Anderson $4b + 9$ algorithm.

5.6 Simpson's algorithms

5.6.1 Simpson's $4b + 7$ algorithm

Simpson's '$4b + 7$' algorithm [Sim90b], is not the smallest algorithm analyzed here, but it is the smallest to maintain the operating properties without the need for any timing requirements, or simplification of the published algorithm. The software algorithm from [Sim90b] is reproduced in Figure 5.24. In software form this algorithm is $4b + 8$, which is simplified for implementation in hardware [Sim92a]. The changes to the algorithm are that the reader variable *pair* is omitted and is combined with the variable *reader*, the reader variable *index* is a two bit array in hardware. In the writer process the *index* variable is removed and $\overline{pair[index]}$ is used. The simplified hardware algorithm is reproduced in Figure 5.25, and a possible hardware implementation is given in Figure 5.26. The simplified algorithm has only three instructions per communicating process, this being achieved on the writer side by making two instructions operate in parallel in instruction '$w1$', this is indicated by the '$\|$' symbol. The algorithm variables are given in Table 5.8.

The order of the instructions within the algorithm is important, even though instructions $r0$ and $r1$ could be interchanged without any effect on the mechanisms normal operation. The writer sets the shared variables to indicate slot then pair, however the reader samples pair then slot, so this order is reversed, thus reducing the possibility of more then one variable going metastable in any one cycle of the algorithm.

The control of the mechanism is via two wires. In hardware this can be reduced to a single wire and the instructions made sensitive to alternative clock edges. For example $w0$ might be active on a rising edge and $w1$ active on a falling edge, and therefore the number of physical connections for control can be reduced.

Another important feature of this algorithm is that each latch only drives one other latch in the other process: there is no variable fan out. This avoids a possible problem under metastable operation, because if two latches sample the metastable output of a latch, they may both settle to opposite values which could cause erroneous behaviour.

The algorithm has four slots arranged in two pairs. The writing algorithm tries to ensure it is always in the pair that the reader is not using, and the reader is always chasing the writer to ensure it is reading the newest data. The slots are not weighted in any way. The writing process never writes to the same slot in succession, it alternates, and this allows the reader access to the freshest data during a succession of writes.

The algorithm uses only simple equality and logical operators. There are no complex operators, and no 'IF' statements.

Variable name	Purpose
Shared reader variable	
r	Indicates to the writer which pair the reader is currently occupying
Shared writer variables	
l	Indicates to the reader which pair the writer wrote the latest data in
$s[0], s[1]$	Each indicates which slot within each pair was last written to
Local reader variables	
$v[0], v[1]$	The readers local copy of $s[0]$ and $s[1]$
Local writer variable	
n	This indicates which pair the writer will next use

Table 5.8: Variables used within the Simpson $4b + 7$ algorithm.

```
mechanism four slot;

        var data : array[bit, bit] of data := ((null, null),(null, null));
            slot : array[bit] of bit := (0, 0);
            latest, reading : bit := 1, 1;

        procedure write (item : data);
        var pair, index : bit;
        begin
            pair := not reading;
            index := not slot[pair];
            data[pair, index] := item;
            slot[pair] := index
            latest := pair;
        end;

        function read : data;
        var pair, index : bit;
        begin
            pair := latest;
            reading := pair;
            index := slot[pair];
            read := data[pair, index];
        end;
end.
```

Figure 5.24: Simpson's $4b + 7$ algorithm (in software form this algorithm is $4b + 8$ - see text).

Writer	Reader
$wr : d[n, \overline{s[n]}] := input$	$r0 : r := l$
$w0 : s[n] := \overline{s[n]}$	$r1 : v := s$
$w1 : l := n \ \| \ n := \overline{r}$	$rd : output := d[r, v[r]]$

Figure 5.25: Simpson's $4b + 7$ algorithm in algebraic form.

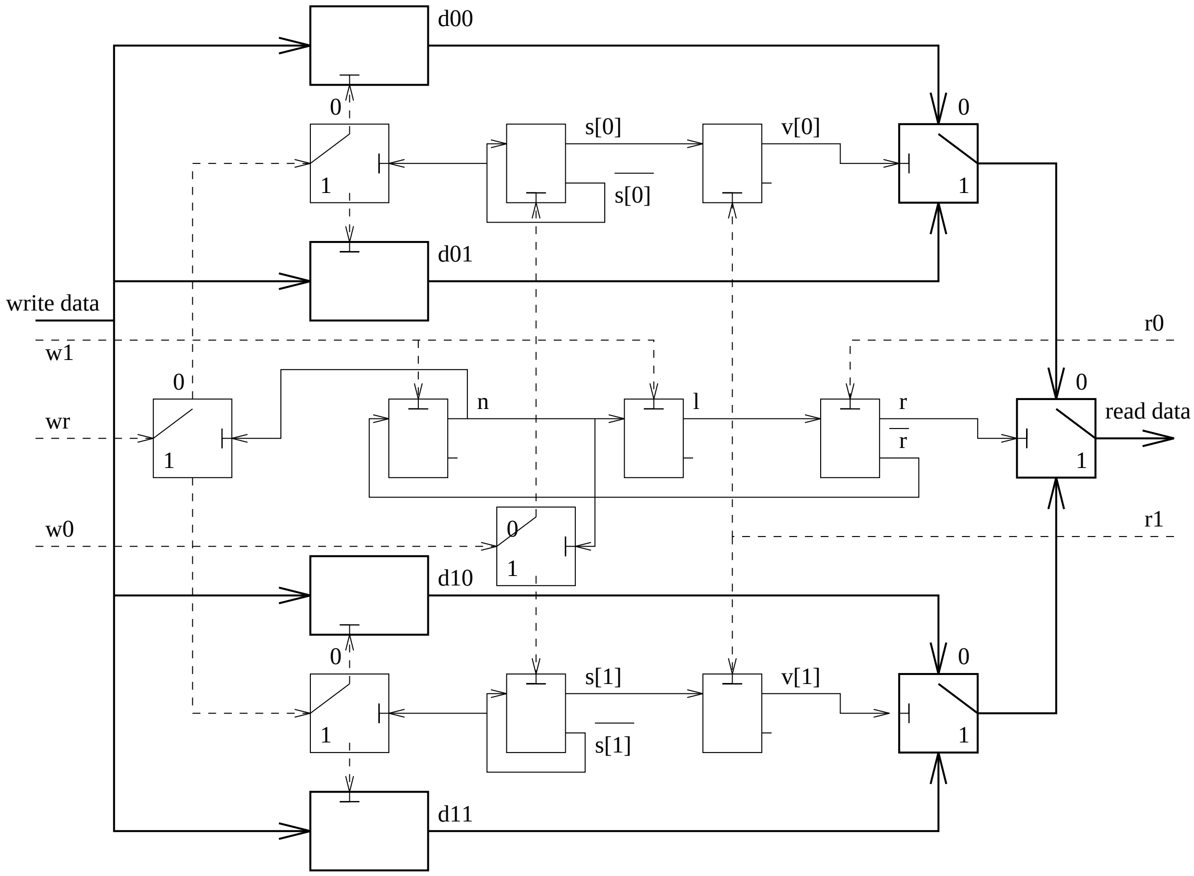


Figure 5.26: Simpson's 4 slot $4b + 7$ in hardware [Sim90b].

The Petri net models for instructions $w0$ and wr are given in Figures 5.27 and 5.28 respectively. These are fairly standard models as used within other models in this thesis. Care, however, must be taken when modelling the interaction between instruction $w1$ and $r0$, as both instructions sample shared variables and also change shared variables. If both these instructions are modelled using atomic actions then the state space will be incomplete. This situation was discussed in Section 3.6.2. From this earlier work it was shown that one of the two instructions needs to be modelled using a two stage model. This was implemented for instruction $r0$ and this is shown in Figure 5.29. The textual Petri net models are given in Appendix B.5.

The Petri net model for testing data coherence for this algorithm is given in Section B.5.1. This model has 54 transitions and 39 places. There are no occurences of a bad marking within the full state space, which has 1472 states, and therefore this algorithm maintains data coherence under normal operation. The results given by Simpson [Sim92a] are thereby confirmed.

The Petri net model for testing data freshness for this algorithm is given in Sec-

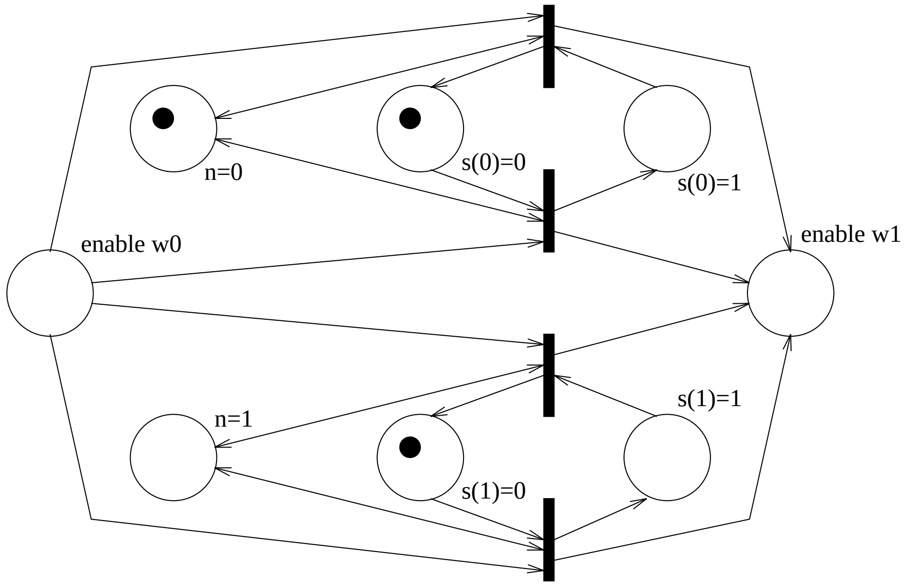


Figure 5.27: Simpson's $4b + 7$ statement '$w0$' in Petri nets.

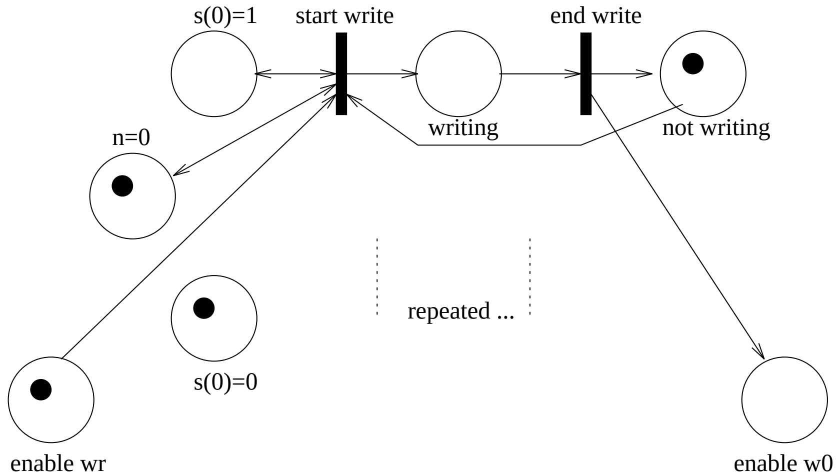


Figure 5.28: Simpson's $4b + 7$ statement 'wr' in Petri nets.

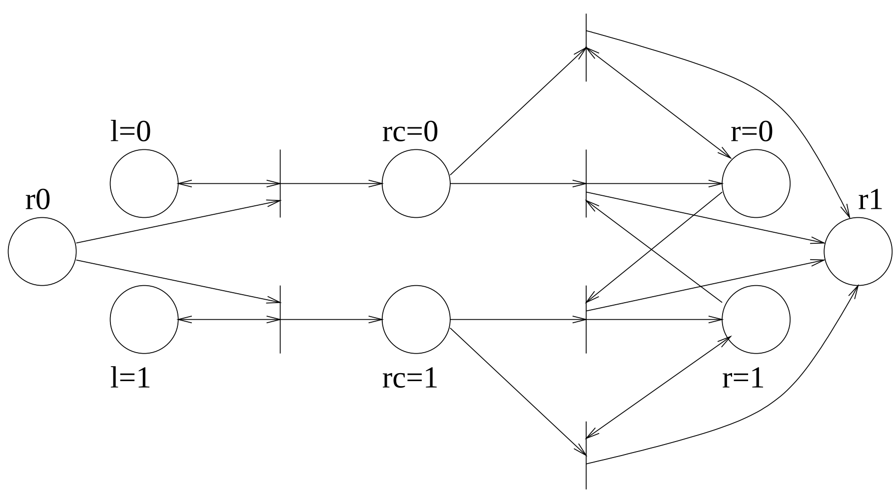


Figure 5.29: Simpson's $4b + 7$ statement '$r0$' in a two stage Petri net model.

tion B.5.2. This model uses the modifed form of data freshness which cancels the erroneous 'p' role (Section 4.4.3). This model has 124 transitions and 80 places. There are 0 bad markings in the state space which has 9076 states. Therefore the algorithm maintains the data freshness property under normal operation. This result confirms the results given by Simpson [Sim92a], even with the extra constraint.

The Petri net model for testing data sequencing for this algorithm is given in Section B.5.3. This model uses the roles from Simpson's '$4b + 6$' algorithm for testing the $new - old - new$ phenomenon [Sim97c]. The Petri net model has 214 transitions and 134 places. The state space has no states which are marked *bad*. The total state space has 13586 states. Therefore the algorithm maintains data sequencing under normal operation. This result has not previously been published.

Simpson $4b + 7$ under metastable operation

The models of Simpson's $4b + 7$ algorithm which are given in Section B.5 have been extended with the models given in Figures 3.6 and 3.7. This allows for the analysis of the algorithm under metastable operation. The models have been slightly modified in that only one metastable event is allowed per test, because metastability is a fairly rare event. Simpson [Sim90b] states that because the algorithm will be controlled by software instructions, the delay between any two instructions should be long enough for any metastable signal to have settled to a stable state. This assumption has been included in the models used here.

The variables which sample shared variables in this algorithm are n, r, $v0$ and $v1$, therefore these are the variables which may become metastable because their input signals are supplied from the other process which is not running to the same clock. Four models were constructed, one for each variable which may become metastable. The metastable outcome is tested for by the model which checks to see if the signals input has changed during its set-up time. Each of the four models is then tested for data coherence, data freshness and data sequence. The models are not included here, as they have only minor changes from those already presented in Appendix B.5. The results from this analysis is presented in Table 5.9.

From these results it can be seen that a single metastable event on the variables

Metastable Variable	Input Variable	Coherence	Freshness	Sequence
n ($w1$)	r ($r0$)	Fail	Fail	Fail
r ($r0$)	l ($w1$)	Pass	Pass	Pass
$v0$ ($r1$)	$s0$ ($w0$)	Pass	Pass	Pass
$v1$ ($r1$)	$s1$ ($w0$)	Pass	Pass	Pass

Table 5.9: Shared variable metastability within the Simpson $4b + 7$ algorithm.

r, $v0$ or $v1$ does not affect the operating properties of the mechanism. However a metastable event on the variable n does affect the algorithm, and causes the algorithm to fail all operating property tests. In the writing process the variable n is used to determine both slot array index variables ($wr : d[n, \overline{s[n]}] := input$), therefore a metastable value on this variable causes the old pair and old slot to be used for writing. From this result it can be seen that the writer process is susceptible to metastable events and the reader process is not.

5.6.2 Simpson's $4b + 6$ algorithm

In a later publication [Sim97b], Simpson made modifications to his original algorithm. The newer algorithm uses one less control bit, but it shares one more variable between processes. A significant consequence is that the writing process has no private local variables (Figure 5.10). This algorithm is reproduced in Figure 5.30 and in algebraic form in Figure 5.31. This algorithm uses a similar statement order reversal in the setting and indicating of pairs and slots as Simpson's '$4b$+7' algorithm. The difference is that the writer now indicates pair first then slot, a reversal from the '$4b + 7$' algorithm. The construction is similar to that of Simpson's '$4b + 7$' algorithm in that is has four slots, in two pairs, and they are not weighted. There is no variable fan out, each latch (variable) only drives one other in the other process.

The Petri net models for this algorithm are given in Appendix B.6. Again care needs to be taken when modelling instructions $w1$ and $r0$, because each samples a shared variable and changes a shared variable: this is similar to Simpson's '$4b + 7$' algorithm. One of the two instructions needs to be modelled using a two stage model (Section 3.6.2).

```
mechanism pool;

        var data : array [0..1, 0..1] of data := ((init, init),(init, init));
            r,w : array [0..1] of 0..1;
            ip : 0..1;

        procedure write (item : data);
        begin
            data[ip, w[ip]] := item;
            ip := not ip;
            w[ip] := not r[ip];
        end;

        function read : data;
        var op : 0..1;
        begin
            r := w;
            op := not ip;
            read := data[op, r[op]]
        end;
end.
```

Figure 5.30: Simpson's $4b + 6$ algorithm.

Writer	Reader
$wr : d[ip, w[ip]] := input$	$r0 : r := w$
$w0 : ip := \overline{ip}$	$r1 : op := \overline{ip}$
$w1 : w[ip] := \overline{r[ip]}$	$rd : output := d[op, r[op]]$

Figure 5.31: Simpson's $4b + 6$ algorithm in algebraic form.

Variable name	Purpose
Shared reader variables	
$r[0], r[1]$	Indicates to the writer which slot the reader is currently occupying
Shared writer variables	
$w[0], w[1]$	Indicates to the reader which slot the writer is currently using
ip	Indicates which pair contains the latest written data
Local reader variable	
op	The readers local copy of ip

Table 5.10: Variables used within the Simpson $4b + 6$ algorithm.

The Petri net model for data coherence is given in Section B.6.1. This model has 54 transitions and 40 places. The full state space has 1350 states and none are marked *bad*, so the algorithm maintains coherence under normal operation, which confirms the published results.

There are two Petri net models for testing data freshness. Both use Simpson's newer freshness roles [Sim97c] (Section 4.4.3). The first model is given in Section B.6.2. This version uses the unmodified roles and has 109 transitions and 97 places. The complete state space has 29117 states, and no '*bad*' states. This implies that the algorithm maintains data freshness, which confirms the published result.

There is however a problem with the definition of data freshness. This problem was discussed in Section 4.4.3, and an example run of the algorithm under the data freshness test is given in Figure 4.4. It can be seen from this route that the '*p*' role masks a problem. A modified freshness algorithm was proposed in Section 4.4.3. A new Petri net model for this algorithm is given in Section B.6.3, and the '*p*' role is removed if the current read cycle is fresh (Figure 4.30). This model has 134 transitions and 99 places. This state space has 33487 states plus 88 bad markings. Therefore this algorithm does not maintain the modified version of the freshness check.

The model for the data sequencing test is given in Section B.6.4. This model has

214 transitions and 135 places. The state space has 20399 states plus 88 bad markings demonstrating that the algorithm does not maintain data sequencing under normal operating conditions with no timing constraints.

Timing constraints for Simpson's $4b+6$ algorithm

Simpson, having found that the new algorithm did not maintain the data sequencing property, published [Sim97c] a timing constraint under which operation of the algorithm would maintain this property. The proposed timing constraint was "$(wip - wis) > (rcp - rcs)$" (using the role model events from Figure 4.3), or re-phrased "the write data operation should always be longer than the read control sequence". Ideally an ACM should have no timing constraints to allow fully asynchronous communication, however in practice the size of a slot should be several bytes, and therefore this operating constraint is often satisfied in practice.

This timing constraint is shown in Figures 5.32 and 5.33. Figure 5.32 shows the timing constraint on an open access algorithm where the start and end of the data write is merged with the end and start of the control part of the algorithm. Figure 5.33 shows the timing constraint on the closed access version of the algorithm where the begin and end of the data accesses are not merged with the end and start of the control part of the algorithm. Therefore the timing constraint should be re-written as $(wre - wrb) > (rcp - rcs)$. If the timing constraint was used in its open access form the timing constraint would have included the prepare data and write data periods together. The Petri net model of the timing constraint is given in Figure 5.34.

The Petri net model of the algorithm with the timing constraint applied for testing the modified data freshness has 142 transitions and 103 places producing a state space with 9465 markings plus 0 bad markings.

The Petri net model for this algorithm under the data sequencing test with the timing constraint has 222 transitions and 139 places. This produces a state space with 13116 states plus 0 bad states. It can therefore be concluded that this timing constraint is sufficient to enable the algorithm to maintain the required operating properties.

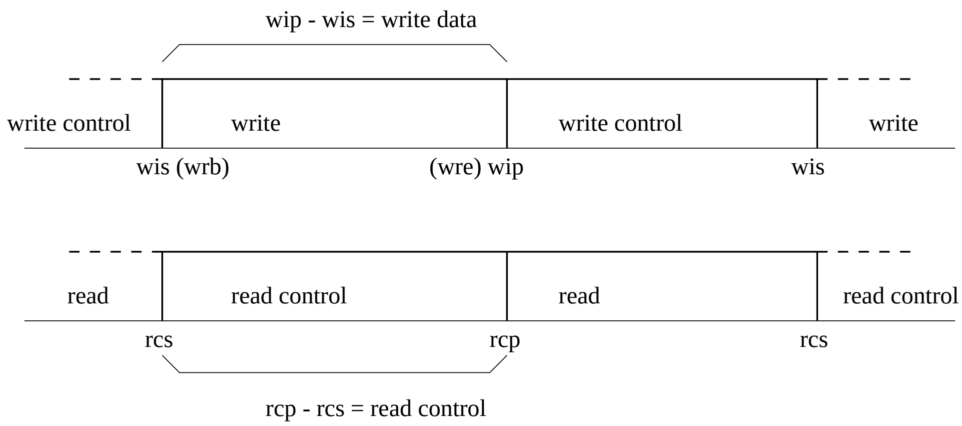


Figure 5.32: Simpson's $4b+6$ four slot timing constraint for open access, with the start and end of the data access being merged with the end and start of the control part of the algorithm.

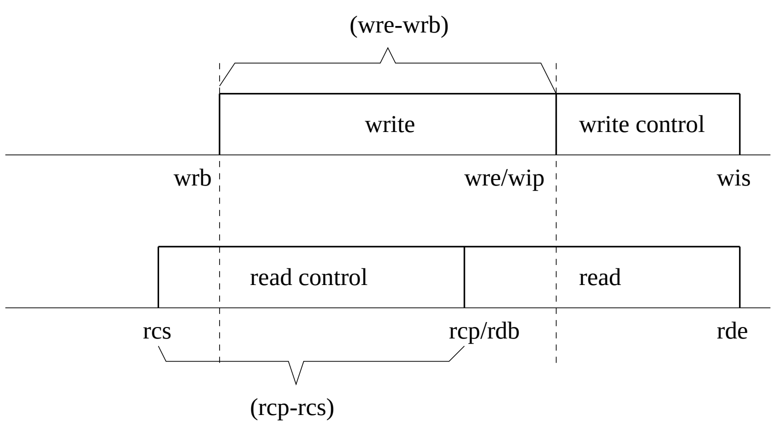


Figure 5.33: Simpson's $4b+6$ four slot timing constraint re-written for closed access. The data accesses are not merged with the control parts of the algorithm.

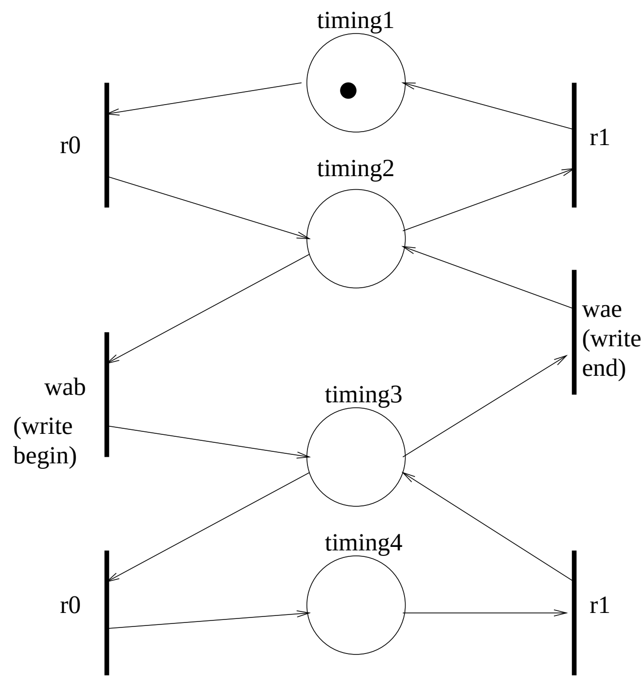


Figure 5.34: Petri net of the Simpson $4b + 6$ timing constraint.

Simpson $4b + 6$ under metastable operation

The models of Simpson's $4b+6$ algorithm which are given in Appendix B.6 have been extended with the models given in Figures 3.6 and 3.7. This allows for the analysis of the algorithm under metastable operation. The models have been slightly modified in that only one metastable event is allowed per test: this is because metastability is a fairly rare event. Simpson [Sim90b] states that because the algorithm will be controlled by software instructions, the delay between any two instructions should be long enough for any metastable signal to have settled to a stable state. This assumption has been included in the models used here.

The variables which sample shared variables in this algorithm are op, $r0$, $r1$, $w0$ and $w1$, therefore these are the variables which may become metastable because their input signals are supplied from the other process which is not running synchronized to the same clock. Five models were constructed, one for each variable which may become metastable. The metastable outcome is tested by the model which checks whether the signal's input has changed during its set-up time. Each of the five models is then tested for data coherence, data freshness and data sequence. The models are not included here, as they have only minor changes from those already presented in Appendix B.6. The results from this analysis are presented in Table 5.11

Metastable Variable	Input Variable	Coherence	Freshness	Sequence
op ($r1$)	ip ($w0$)	Pass	Pass	Pass
$r0$ ($r1$)	$w0$ ($w1$)	Pass	Pass	Pass
$r1$ ($r1$)	$w1$ ($w1$)	Pass	Pass	Pass
$w0$ ($w1$)	$r0$ ($r0$)	Pass	Pass	Pass
$w1$ ($w1$)	$r1$ ($r0$)	Pass	Pass	Pass

Table 5.11: Shared variable metastability within the Simpson $4b + 6$ algorithm.

From these results it can be seen that a single metastable event on any sampled shared variable does not affect the operating properties of the mechanism. This may at first sight seem a strange result, however the algorithm with the timing constraint applied is partially limited so that not all instruction interleavings are possible. This timing constraint therefore helps to protect the mechanism from failing the operating properties, this is helped because with limited instruction interleaving the sampling variable can only get a new or old value and not an erroneous value (Section 6.1.4).

5.7 Other algorithms

There have been several other ACM design solutions published in the literature [Claa]. A selection of some which have not been analyzed here is given below. The techniques presented in this thesis can be used to analyze these other algorithms. Some of the published algorithms inherently include waiting, and so have not been analyzed in this thesis because only wait-free algorithms are of interest here. The solutions listed here do not include any metastable analysis, and most employ software constructs such as 'FOR' loops and 'IF' statements, and therefore do not limit the operators they use to those with predictable asynchronous operation. (Predictable asynchronous operation means that an operator under metastable conditions has a known characteristic, i.e. a metastable bit can hover between the two specified logic levels, however the operation of some software operators and other complex operators when implemented in hardware have unknown characteristics under metastable operation).

Lamport's [Lam86] publication is widely cited by many authors as the first core paper on the subject of asynchronous communications and he has published many of the definitions upon which other authors base their results. Lamport's two core papers list the formalisms of interprocess communication and algorithms. Lamport was the first to define different types of register as $safe$, $regular$ and $atomic$ and he states that there is no solution for an atomic register with only one process writing the control variables, this implying that two way communication is required. One concern already voiced is that Lamport acknowledges the existence of metastability but he does not include it in any of his analysis. Lamport's main construction unfortunately uses 'IF' statements.

Haldar and Vidyasankar [HVwn] proposed a minimal solution to the ACM using four slots and five control bits $(4b + 5)$. This has been referenced from [HV96] but as yet has not been found by the author of this thesis. However another publication by the same authors [HV95], gives a construction which writes the same data to two slots and then upon reading decides which should be used. This construction uses an 'IF' statement which is not desirable in a concurrent environment (Section 6.1). The multi-reader constructions given in [HV95] are both large in size, the first having 11 instructions for the writing process, and 23 instructions for each reading process. The second has 13 for the writer and 22 for each reader process. However all unfortunately use 'IF' statements.

Singh, Anderson and Gouda [SAG86, SAG87, SAG89, SAG94] give several constructions which warrant future investigation, but an initial observation is that the constructions use software type constructs such as 'FOR' loops and 'IF' statements. The proofs of the given constructions are very long and the methods presented here would provide an easier to follow proof.

Li, Tromp and Vitanyi [LTV96] give several interesting constructions, but the physically constructable versions (i.e. those with bounded arrays) use software constructs like 'IF' which is not ideal in asynchronous hardware. Most of the constructions are however designed for multi-reader ACMs.

5.8 Conclusion

This chapter has presented several ACM algorithms published in the literature. Most of the algorithms have been modelled and analyzed using the same Petri net technique, therefore allowing a direct comparison of the operating properties of each algorithm which has not previously been possible. The results of this analysis were summarized in Table 5.1 with the key given in Table 5.2. The technique used here allows for quick analysis of such algorithms without the need for long and complex formal proofs, which are often difficult to follow, and are unique to the author who published each algorithm. This analysis has also introduced the metastable phenomenon which is omitted by most authors. Operation under metastable conditions has been investigated on the smaller algorithms as these would produce the best solutions in hardware, and are therefore the most likely to be used in actual hardware.

Chapter 6

ACM design rules and future work

The purpose of this chapter is two fold. Firstly, a set of design ideas for ACM algorithms is presented highlighting why some operators should be avoided, and the failures they can cause. Secondly, the future directions of this research is discussed, including the extension of this work to the analysis of multiple writer / multiple reader ACM algorithms.

6.1 ACM design rules

From the algorithms presented in this thesis and others studied by the author it has been noticed that some rules should be kept in mind if new algorithms are to be designed. Many of the algorithms studied are published by computer scientists who use software operators in their algorithms with no thought to hardware implementation. Many of the authors do not acknowledge the phenomenon of metastability and omit it from their analysis.

The most basic of design rules is to use the smallest possible number of variables, share as few of these as is possible and keep the algorithm as simple and short as possible. Any unnecessary complexity and size should always be avoided. By keeping the number of shared variables to a minimum the possibility of metastable signals is reduced. Also keeping the algorithm simple and short, the algorithm can be more easily understood and analyzed. Unnecessary complexity may hide avoidable conditions under metastable operation.

Anderson [AG92] published his "atomicity criterion" as a list of six requirements for an atomic variable, or ACM as it has been called in this thesis. The six requirements are :-

1. Each shared variable is a single bit.
2. Each shared variable is read by at most one process.
3. Each assignment statement reads at most one shared variable and does so at most once.
4. Each shared variable is written by at most one process.
5. Each assignment statement writes at most one shared variable.
6. Each assignment statement does not both read and write a shared variable.

Item one is discussed in Section 6.1.4, and item four was discussed in Section 2.3.2 which uses the running of a bank account as an illustrative example.

This Author finds the remainder of this list contentious.

Item two cannot be realized in multiple reader mechanisms, and in the case of single reader mechanisms Tromp's $4b+14$ algorithm succeeds in using multiple copies of a shared variable.

Item three cannot be realized if an array of shared variables is copied. Simpson's $4b+7$ algorithm manages this safely with instruction $r1$ in which $v := s$ where 's' and 'v' are both two bit arrays. The ACM designer should ensure that the shared variables being copied are not changed within the same instructions of the controlling process. Operation of the copying instruction can then in the worst case obtain one changing variable, which may cause a metastable event. This is no worse than copying a single shared variable. This relates to Item five.

Item five is sensible but limiting. The ACM designer should ensure that if multiple shared variables are to be written by a single instruction, then the shared variables must be read by different instructions in the other process. Therefore if an instruction were to change three shared variables, then these variables should be read by three different instructions in the sampling process, therefore limiting the chances of sampling a changing value.

Item six was discussed in Section 3.6.2. The point Anderson attempts to make is that if a shared variable is sampled while it is changing then a metastable event may occur, and if this event is then transmitted back via a shared variable written by the same instruction then this is unacceptable. However, metastability will normally settle in the period of time between two instructions and therefore this operation is acceptable. (Anderson does not deal with metastability in his paper). Simpson's $4b + 7$ algorithm performs such an operation safely.

This author recommends the following as good ACM design practices.

6.1.1 Use variable fan-out only in synchronous circuits

If a variable is shared between two or more processes, ideally a single local copy of that variable should be made, and then any further references to its value should be made to the local copy. If a shared variable is read more than once during an access cycle the possibility of reading two different values exists. This is made worse in algorithms which use multiple bit variables because this can lead to a worse scenario in which two completely wrong values could be obtained, because of sampling the value while it is being updated.

A variation on this problem is shown in Figure 6.1, where process 2 takes two copies of a variable from process 1. In this situation if this single sampling period coincides with the variable in process 1 being updated then both A and/or B could go metastable, and eventually could settle into opposite states (one latch settles to logic '0' and the other settles to logic '1').

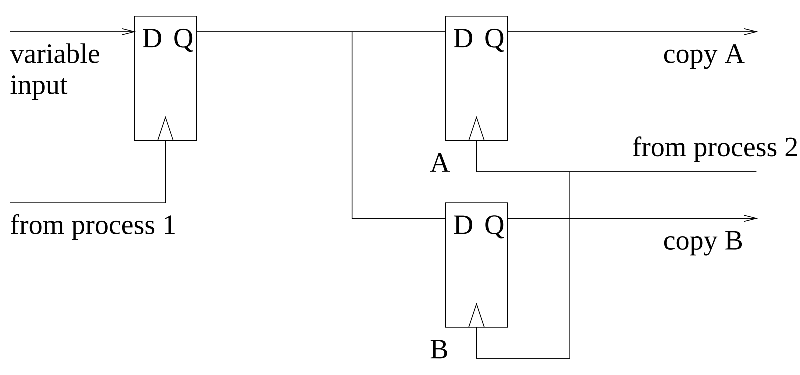


Figure 6.1: Avoid variable fan-out.

6.1.2 Use only 'copy', 'invert' and logical functions

Functions which are used in software environments can be complex to implement in hardware, or can have unknown characteristics under metastable operation. For example Kirousis's algorithm uses a function called 'min' to obtain the smallest of a set of numbers (which are used to identify the slots within the mechanism). This is not simple to implement in hardware. Ideally ACM algorithms should only use simple logical devices and latches.

The software 'IF' statement should ideally be avoided in ACM algorithms but many authors use it. The 'IF' function can be easily implemented in hardware using an exclusive-OR gate. However care must be taken if a shared variable is used as one of the 'IF' inputs. Two possible implementations are shown in Figure 6.2. If the implementation does not buffer the inputs as the top diagram in Figure 6.2 shows, then if the result of the 'IF' statement could be fed back as it is changed and cause self canceling, i.e. at the start of the statement the 'IF' is true, but as the action of the 'IF' is performed the 'IF' becomes false (or vice-versa). If the inputs are buffered then the inputs to the 'IF' statement could become metastable and can cause a dangerous failure. If this metastable input settles during the following action then half of one action and half of another action could be performed.

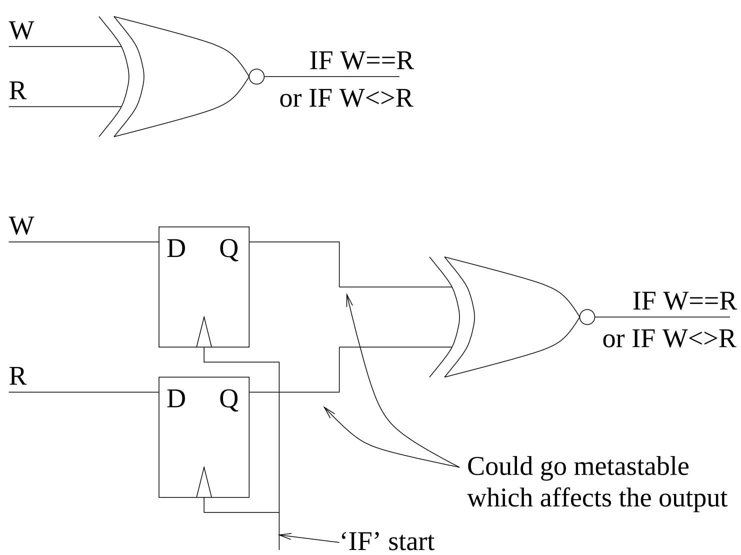


Figure 6.2: Hardware 'IF' statement.

Variable transitions should be kept to a minimum in order to reduce the probability of triggering a metastable event. Variables can be changed by connecting Q bar to D and only clocking the latch when a change is required. Some algorithms only change variables when a certain condition arises (in order to reduce the number of variable transitions), this however, may require an 'IF' statement to test for the condition when a transition should occur.

6.1.3 Use different logical sequences in different processes - if possible

When two algorithms are communicating a sequence of two or more (shared) variables, then the order in which they are indicated and chosen by the other process can improve the solution. Simpson's $4b+7$ algorithm [Sim90b] exchanges shared variables between its $w0$, $w1$, $r0$ and $r1$ instructions. It should be noted from the algorithm that the order of the indication and choosing of the values has been reversed so that they do not occur in the same order. In the Simpson case the writer process sets and indicates the value of S and then the value of l. On the reader side this is reversed and l is sampled before the value of S. This reduces the chances that both sampling variables go metastable in the same cycle of the algorithm. This is shown in Figure 6.3.

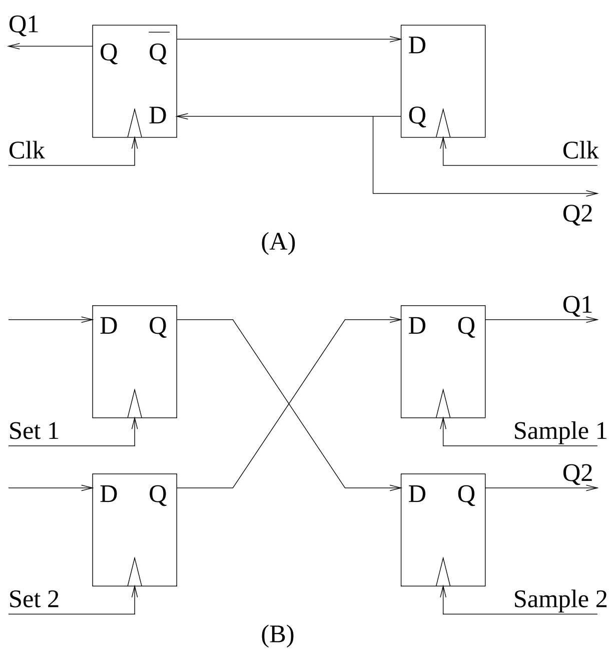


Figure 6.3: Variable setting.

6.1.4 Use only single bit signals

Variables should not employ multiple bits in asynchronous environments. If they do, they may sample incorrect values from a changing variable. However this can also happen to a binary variable, but because there are only two stable states, the result is not necessarily so dangerous. This can happen because of a metastable state, but can also be caused by runt pulses within the logic of an algorithm. An example is given in Figure 6.4. It can be seen from Figure 6.4 that the value of A is changing as B samples it. This in itself is not dangerous, but it causes B to produce a short runt pulse (which settles back to A's old value). At this point C samples this pulse from B and assumes the wrong value. Now process one could use variable C's value to indicate the current location of process two, and use this value to steer itself away from process two (one bit is not enough for indicating a processes exact location, the single bit may indicate the processes location as a pair or slot within a pair). An immediate solution to this problem is to make sure that clock 3 occurs before clock 1, in this case the value of C is obtained before the value of A is changed. Or make sure there are other instructions between changing A's value and C sampling B, therefore any metastability on B should have settled by the time it is sampled.

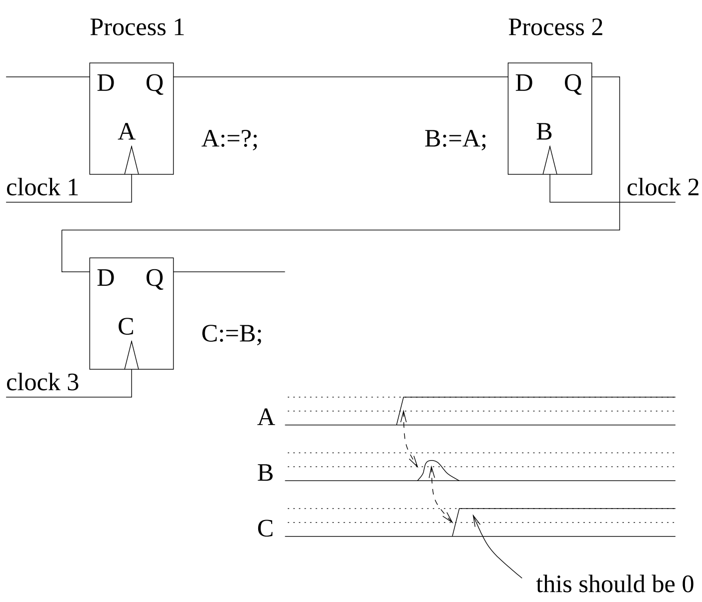


Figure 6.4: Be careful with variable interleaving.

Another solution to this problem of obtaining incorrect values has been published by constructing atomic bits from safe bits. Lamport [Lam86] reported that 2 way communication is required for atomicity using only safe registers. Tromp [Tro89] also published a solution using this two way communication. Both these methods use three shared variables, but both omit the metastable phenomenon from their analysis, and both algorithms use 'IF' statements which are not predictable in such conditions.

6.1.5 Use local copies of shared variables

The use of shared variables should be restricted so that only local copies of shared variables are used by each process. This is necessary for correct algorithm operation. The problematic cases of using shared variables are part of the equality operator in an 'IF' statement (Section 6.1.2), as an array index variable, or as a slot index for the writing or reading process. The problem with using shared variables is that because the value of the variable is controlled by a process which is not synchronized and any instruction interleaving is possible, then the variable may be changed during the operation which is using it. Therefore in the case of using a shared variable as a slot index variable, the writing or reading process may be moved mid access, which leaves two slots containing illegal data in the case of a write, or in the case of a read, the data obtained will be some combination of two slots, and therefore illegal. A simple algorithm by Fraser [Fra00] given in Figure 6.5 shows this problem.

Writer	Reader
$w0 : b[a] := \overline{b[a]}$	$r0 : last := \overline{a}$
$w1 : d[b[0], b[1]] := input$	$r1 : output := d[b[0], b[1]]$
$w2 : a := last$	

Figure 6.5: A simple algorithm by Fraser illustrating the use of shared variables for the data slot index variables, which is not safe.

In Fraser's algorithm the slots are depicted by the variable 'd' which is a two by two array, therefore four slots. The slot indices are the variables '$b[0]$' and '$b[1]$'. These indices are controlled by the writer process, but they are used by the reader process when it accesses the slots. Therefore if a read process starts and the writer instruction $w0$ executes then the indices will be changed mid-read, which is unsafe operation.

If a control variable uses a shared index variable during an assignment statement the effects will be less pronounced because at worst the variable can become metastable (assuming the control variable is binary, as recommended).

6.1.6 Use different variables for data slot index variables

Several of the ACM algorithms presented in Chapter 5 arrange four slots into two pairs of slots and then access each slot using a two dimensional array, such as $a, [b, c]$, where a is the group of all four slots. As the results in Chapter 5 have shown this can be a very effective way of organising the data slots. However, the algorithm should use two independent variables as the index variables (b and c in this example). Some algorithms do not use two independent variables, such as Simpson's $4b + 7$ algorithm which uses '$wr : d[n, \overline{s[n]}] := input$'. It can be seen that this instruction from Simpson's algorithm uses the variable n in both index variables. Under non-metastable operation this is safe, however under metastable operation (which is unavoidable in an asynchronous environment) it can cause the algorithm to not maintain the ACM operating properties of data coherence, freshness and sequencing. If the variable n were generated locally to the process in which it is used then there is no problem, but in the case if Simpson's algorithm its value is sampled from a variable in the other communicating process.

6.2 Which algorithm fits these rules

Given these design recommendations and the results presented in Chapter 5 two algorithms are chosen as the most suitable solutions to the problem of asynchronous communication. Simpson's $4b+6$ algorithm which is the smallest solution analysed requires the use of a timing constraint, and is therefore not a solution for unconstrained fully asynchronous communication. The two algorithms which are suitable are Anderson's $4b + 9$ ACM algorithm and Simpson's $4b + 7$ ACM algorithm. Simpson's algorithm maintains the required operating properties with no metastable events, however it fails all three operating property tests when one if its variables (n) becomes metastable (due to using the affected variable in both slot index variables, Section 6.1.6). Anderson's solution uses more bits than that of Simpson's, however, if Anderson's "atomicity criterion" is relaxed (Section 6.1), then Anderson's algorithm can be simplified. This simplified algorithm is given in Figure 6.6 and is similar in size to Simpson's algorithm using $4b + 7$ bits. This reduction is achieved by allowing an instruction to both read and write a shared variable, which therefore renders the variables d and rp redundant. This simplified algorithm maintains all operating properties, without the need for a timing constraint, and also with a single metastable event. It also adheres to all the above mentioned recommended design rules.

Writer	Reader
$w0 : wp := \overline{RP} \parallel walt := \overline{walt}$	$r0 : RP := WP$
$w1 : Y[wp, walt] := input$	$r1 : ralt := Z[RP]$
$w2 : Z[wp] := walt$	$r2 : output := Y[RP, ralt]$
$w3 : WP := wp$	

Figure 6.6: Anderson's algorithm simplified to $4b + 7$.

6.3 Future work

The work described in this thesis has laid the foundation for an investigation of other ACMs and for the development of improved rules for the design of ACMs. Further work is currently in progress as part of the EPSRC supported 'COMFORT'[1] project at King's College London.

The models presented in this thesis were all constructed manually. Although this is not a long process it is open to human errors, like the simple typo. The coverage output from *reach* helps to highlight such errors, but unfortunately odd errors creep in, so care needs to be employed. The best test against this is to test each instruction against its Petri net model to make sure that the model is an accurate representation. A set of rules is followed in the modelling process, so it should be feasible to automate the process. Such a system would enable an analysis program to take any ACM algorithm as input (depending on the syntax used), and construct the Petri net model automatically.

With insight of the ACM operating properties, and given certain algorithm properties, direct algorithm synthesis should be possible. Existing search techniques (like that used for noise shaper filter design in [CHD93]) could be employed to hunt for all possible solutions with a set of given characteristics. The search algorithm could be given a set of possible operators, a maximum number of variables and instructions, and the ACM operating properties would be used to evaluate all solutions found.

[1]The COMFORT project - COMmunications FOr Real Time systems

Exhaustive search methods may prove unproductive in such circumstances, in which case a directed search method could be employed. Such search methods may uncover suitable three slot solutions to the ACM problem, but it is expected that any solutions found using three slots would not be as robust as those using four slots.

As discussed, (Section 6.1.2) software instructions such as 'FOR' and 'IF' should ideally be avoided in ACM designs which will be implemented in hardware due to the unpredictable nature of these instructions under metastable events. It may be possible to rewrite some of the ACM algorithms analysed in this thesis to remove these instructions. Any rewriting must ensure that the underlying algorithm operation is unaffected. (The term algorithm operation is used here to mean the function of steering the writing and reading operations to the individual data slots).

During this work, memory optimisation techniques (Section 4.3.2) had to be introduced for the analysis of larger algorithms. Simpson's role models [Sim92a, Sim97c] have proved quite compact because of their use of pattern matching. It may be possible to use pattern matching when using Petri nets to facilitate more efficient analysis. This would introduce a new set of questions into the analysis problem.

The analysis method used in this thesis is reachability searching, which is a simple state space searching technique. Other methods of analysis have been shown to be unable to handle the cyclic models used in ACM analysis. However a method called unfolding, first introduced by McMillan [McM92], involves the transformation of a Petri-net into an occurrence net with an aim of reducing the size of the state space of the model being investigated.

The unfolding of a net produces another net, usually infinite but with a simpler structure, called the occurence net. McMillan's algorithm constructs a finite initial part of the unfolding which contains full information about the reachable states. This finite initial part is called a finite complete prefix. Esparza et al [ERV95] reported that McMillan's algorithm although "simple and elegant", sometimes generated prefixes much larger than necessary. McMillan's algorithm in some cases generates a prefix of size $O(2^n)$ from a net where the minimal complete prefix is $O(n)$, where n is the number of nodes in the original net. The minimal complete prefix is defined by Esparza and improved cut conditions are defined which result in an algorithm which, it is claimed, will always generate a prefix no larger than the prefix generated with

McMillans algorithm.

Semenov and Yakovlev [SY97] have investigated the four-slot algorithm presented in [Sim90b] using the unfolding technique. They confirm that unfolding can support self-loops but the combination of concurrency and conflict cause the technique to lose its efficiency. Their solution to this efficiency problem was to introduce contextual-nets [BP96]. This change resulted in an unfolding of 1299 transition instances for the contextual net, in contrast to the Petri-net unfolding which could not be built due to size (> 3000 transition instances). Their results claim confirmation that the four-slot maintains data coherence. The data freshness property was not tackled, but the author of this thesis expects that the unfolding of the data freshness net would also be prohibitively large (estimated to be at least four times larger).

Currently the output data from *reach* is textual, (appart from some model sections which can be drawn using at AT&T's "Graphviz" package [GbATR]). A great improvement would be the output of 'bad' marking routes in timeline format. Currently these have to be drawn by hand from the textual output routes.

During discussions and presentation of this research both Dr Xia and this author were asked why the work is based on Petri nets and not Finite State Machines (FSMs). Because this work is directed towards designs that are likely to be implemented in hardware, existing Petri net packages (like Petrify [Cor]) can be employed in producing hardware designs automatically. It would however be interesting to examine the FSM method of analysis to see if it offers any advantages over the existing Petri net based solution. It has been suggested in a private communication between the author and Jordi Cortadella (the author of Petrify) that FSM methods can be directly analysed using Binary Decision Diagram (BDD) techniques. Holzmann [Hol91] claims that PNs are relatively weak for modelling compared to FSMs, but the context of this comparison is not indicated. Recently a number of FSM analysis packages have been made available ([VHHP95, Arn97, DES99]) and these need to be investigated and verified for all the properties which can be confirmed by *reach*. It may prove that there is a direct route into FSM with the ACM algorithm without need for translation into Petri nets. This would remove one stage from the analysis.

6.3.1 Multiple writer / multiple reader mechanisms

The results presented in Chapter 5 were based on ACM algorithms with a single writer and a single reader. The techniques used here can be extended to include more than two processes and therefore could be used for algorithms with multiple writers and/or multiple readers. A number of algorithms have been published by Simpson [Sim97a], Bloom [Blo86, Blo88], Israeli [IS92], Vitanyi and Awerbuch [VA86], Awerbuch et al. [AKKV87] and Peterson and Burns [PB86] all of which warrant future investigation.

Any investigation should aim to identify a pattern or rule relating the number of extra slots and variables required for each extra writer or reader process. The analysis process should be generalised to n processes, especially in the memory optimisation of the state space search. The ACM operating properties also need extending for algorithms with more than two processes. Data coherence would remain unchanged, but the properties of freshness and sequencing would require substantial change.

6.4 Conclusion

Some design guide lines for ACM algorithm construction are identified, and some of the operators which should be avoided are highlighted with the failures that each can produce. The future direction of this research has also been described, including the extension of this method to the analysis of ACMs with more than two processes, and automation of some stages currently carried out manually.

Chapter 7

Conclusion

7.1 Positive results

The work presented in this thesis, describes an analysis method for ACM algorithms using Petri nets. ACM algorithms provide a necessary communication method in parallel processing networks, and also form a core part of MASCOT, which is used to design complex systems based on the flow of data throughout a system and can be implemented using concurrent hardware. These algorithms refine the current definitions of asynchronous communication, and facilitate temporal breaks in multi-process systems.

Overall this method enables the analysis of ACM algorithms using the accessible method of Petri net modelling and analysis. This form of ACM analysis enables fast analysis of the algorithms and tests their operating properties against defined requirements. This enables the design of new algorithms to be performed more quickly as amendments and changes to a design are quickly analyzed using these techniques. These techniques also provide insight into how an algorithm has failed and where its weaknesses lie. The four main results of this work are discussed in the following Sections.

7.1.1 Identification of how to accurately describe ACMs and metastability using Petri nets

Petri nets have been shown to be a very capable medium for modelling and analysis of ACM algorithms, and with the availability of asynchronous hardware synthesis packages (like Petrify) the algorithms can be directly implemented in hardware. The methods and techniques for modelling ACM algorithms have been described, and the reasons set out for modelling each instruction individually and not using abstract modelling methods. The use of cyclic Petri net (or complementary place) models has been described, and how cyclic models cause many of the existing Petri net analysis techniques to fail. Reachability searching was found adequate for the purpose of this work, and memory optimization techniques have been presented to allow multiple process models to be analyzed using the least amount of memory possible, therefore allowing larger and more complex models to be analyzed.

In this thesis textual models were preferred over graphical models because the construction of large cyclic models by graphical means is difficult and easily affected by human error. Larger models are also extremely difficult to follow in their entirety unless multiple place instantiation is allowed, which none of the tested packages support. The textual approach allows individual instructions to be modelled using a self contained block of transition statements. This also allows for model reuse, and during the design of the algorithm any changes are more easily made and helped by easy searching and sorting.

The methods for modelling shared variables between concurrent processes has been described in detail and example cases of where two stage models are required to ensure the hardware implementation of such algorithms is accurately modelled. The technique of modelling the metastable phenomenon in Petri nets has been introduced which has not previously been published or used in the analysis of such mechanisms. The 'OR' arc extension to Petri nets has been proposed and used within this research, this extension does not change the functionality of the Petri nets but reduces the size of the models required for some instructions.

7.1.2 A method of testing algorithms

The ACM operating properties of data coherence, data freshness and data sequencing have been described in detail and their implementation in Petri nets has been demonstrated, and used throughout the results presented in this thesis. The modelling methods have been demonstrated and built upon throughout the thesis starting with the model of the basic one slot mechanism and building up to the fully asynchronous mechanisms which require four data slots to operate. Timing constraints have been described, which facilitate the inclusion of basic relational timing information in this method which includes no physical time information. The inclusion of timing constraints allows some algorithms which fail fully asynchronous operation to be analyzed for use in environments where certain timing or interleaving characteristics can be guaranteed not to occur. The analysis of the three slot mechanism showed that the presented algorithm did not maintain data coherence under fully asynchronous operation, however work at Newcastle University has shown that by making the shared variable access mutually exclusive the three slots are sufficient. This does not provide a robust solution in a safety critical environment because of the possible lock out which can be caused by the failure of one process. It is therefore concluded that at present four slots are required for unconstrained fully asynchronous communication. But the possibility of a three slot solution being devised in the future is not ruled out completely.

7.1.3 Algorithm review

The results presented in Chapter 5 compare several single writer / single reader ACM algorithms using the same technique and operating requirements. This has highlighted previously unpublished operating failures in some of the algorithms. The algorithms presented range widely in size and complexity. No comparison of ACM algorithms has previously been published and the methods presented in this thesis enable direct comparisons to be made between different authors algorithms. The algorithms being compared have all been modelled and analyzed using a common Petri net technique, therefore the comparison is based on the individual algorithm and not each authors style or analysis method. The method also allows any algorithm failures to be directly compared. The future work of extending this work to include more than two processes (multiple writers and/or multiple readers) has also been introduced.

7.1.4 Algorithm design criteria

During the course of this work many algorithms have been analyzed and compared. The success and failure of these ACM algorithms has provided a valuable insight into ACM design guidelines, which if followed improve the robustness of ACM solutions, especially under metastable conditions. A list of ACM design guidelines has been presented in the thesis, and the algorithms which follow this list the closest and maintain the operating properties have been indicated.

Appendix A

The analysis program - *Reach*

This appendix describes in more detail the analysis program *reach* which was introduced in Section 4.3. This program performs a breadth first reachability search on Petri net models and uses a visited list to keep a record of the states it has previously investigated. This ensures that only the minimal reachability tree is produced and ensures that the analysis terminates. The cyclic nature of the ACM models causes many other reachability programs to get stuck in an infinite loop. A flowchart showing the core of *reach* is given in Figure A.1.

Reach has many run time options which are listed below.

A.1 *Reach* run time options

Reach has many run time options, the list is given in Figure A.2. Each option is explained here.

c - output the model in sections in GraphViz format (GraphViz cannot handle node names starting with numbers or nodes containing some ASCII chars - i.e. = (equals sign)). The input petri net model is broken down into individual instructions and outputs in a format reach uses for visualization in *GraphViz* [GbATR]. This helps to spot simple model errors, and also to visualize the function that an instruction is performing.

r - reverse state ordering (default alphabetical). The list of Petri net place labels

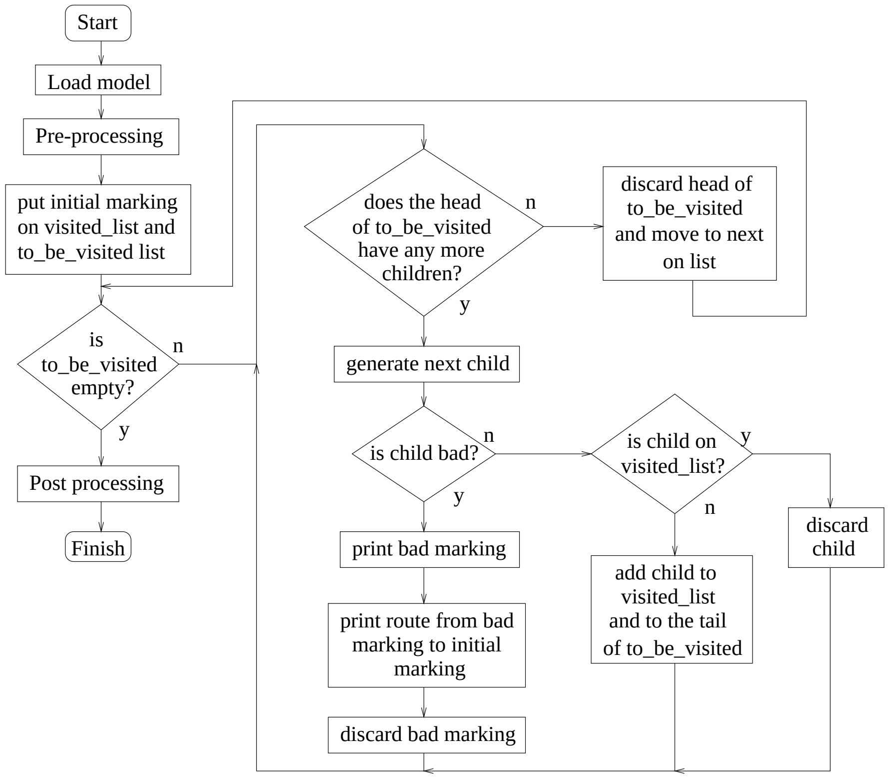


Figure A.1: Core flowchart of 'reach'.

is ordered within *reach* alphabetically: this option allows this ordering to be reverse alphabetical, which sometimes helps state searching using *grep*.

A - store visited list in experimental tree (1 bit). Change the visited list storage from the default single linked list (Section 4.3.2, to a 1 bit $marked/\overline{marked}$ tree (Section 4.3.2).

B - store visited list in experimental tree (2 bit). As above except a 2 bit tree.

C - store visited list in 2 trees (wr/rd) (Max $2^{16} = 65536$ tree). The next three options optimize the visited list storage to systems with two processes (Section 4.3.2, this option using a 16 bit final tree, so limiting the number of states in each process to 65536.

D - store visited list in 2 trees (wr/rd) (Max 2^{32} tree). Same as the last option except the final tree uses 32 bits so much larger state spaces can be stored.

E - store visited list in 2 trees (wr/rd) then in 32 bit block arrays. Same optimization for two processes as the last two options, but the final marking is stored in an array (Section 4.3.2).

```
"reach_7e.c" : Usage : reach [-options] <input model file> [<tree file>]

options

  c output the model in sections in GraphViz format
(GraphViz cant handle node names starting with numbers or
nodes containing some ASCII chars - i.e. = (equals sign))
  r reverse state ordering (default alphabetical)
  A store visited list in experimental tree (1 bit)
  B store visited list in experimental tree (2 bit)
  C store visited list in 2 trees (wr/rd) (Max 2^16 = 65536 tree)
  D store visited list in 2 trees (wr/rd) (Max 2^32 tree)
  E store visited list in 2 trees (wr/rd) then in 32 bit block arrays
  F store only markings with 'enable' marked on visited list
  M debug the memory usage by splitting it up into known areas
  R run without ROUTE information (to save memory)
  o output all visited states (in <input model file>.out)
  t generate textual reachability tree
(in <input model file>.tree)
  w same as -t option but tree produced is in HTML format
NOTE if F is used with t or w then there will be nodes missed.
  f text tree produced with only reading/writing nodes
as fathers and children
  p probability calculation between reading/writing nodes
  x output state space in xpm format
  j output statistical data from reachability tree
(note can't do probability and statistics together)
  a produce all bad routes (default is first only)
  b change the bad_example_file state output (default)
to {ref / remove / add} output
  k change the bad_example_file to split writing / reading trans.
  d tree differences between the current model and <tree file>
note: the <tree file> MUST be subset of <input model file>
  e same as d except lists head non-common nodes to common
children, plus trial matching <tree file> routes
  s make the difference output (d) contain full state details
  i make the difference output (d) contain {ref / remove / add}
  v output tree in HTML with the common/non common coloured/ref'd.
  z difference output - node numbers in a separate file
  g output tree in GraphViz format highlighting common/non common
  m same as g except common routes omitted to save space.
  G print debugging information.
  X print debugging information - in binary.
  T enable testing section of code.
  Z enable testing2 section of code.
  H enable debug(3) output, ie routes to specific transitions.
  I Interactive model running.
  y output writer/reader trees if alt_visited = C/D/E (above).
  h display usage information (i.e. this).
```

Figure A.2: *reach* run time options.

F - store only markings with 'enable' marked on visited list. Another memory saving option, only stored states on the visited list which have the enable place marked. This saves storing intermediate states in atomic operations (Section 4.2.6).

M - debug the memory usage by splitting it up into known areas. *Reach* allows memory usage monitoring during run time. This option splits the memory usage output up into its constituent parts to allow closer monitoring. This is a debugging option.

R - run without ROUTE information (to save memory). When a *bad* state is found the transition firing order is required to show how, from the initial condition, this *bad* state was obtained. The information required to produce this route is however quite memory hungry, so this option turns the route storage information off.

o - output all visited states (in <input model file>.out). In some cases a complete list of all systems states may be required. This option outputs all obtained states in a file, with one state per line.

t - generate textual reachability tree (in <input model file>.tree). This generates an output file which lists all generated states, followed by a list of all that states children. Each state is given an identifying integer number. In a system with two processes each state should have two children, no more and no less, unless the 'enable' place has been used to create atomic actions within the model.

w - same as -*t* option but tree produced is in HTML format NOTE if *F* is used with *t* or *w* then there will be nodes missed. Same as last option except the tree is generated in HTML format which can be viewed in any web browser. Each state is linked with its children and its parents so the tree can be traversed using *URL* links.

f - text tree produced with only reading/writing nodes as fathers and children. Experimental output which listed states only containing a reading or writing state and producing routes to all other reading or writing states.

p - probability calculation between reading/writing nodes. This is an experimental output which is not reported in this thesis. Each instruction is given a time duration, and probabilities were calculated from this. Eventually dropped as results revealed nothing.

x - output state space in xpm format. Output the state space in *xpm* format which is a form of bitmap syntax. Initial results were published by Xia [XL98].

j - output statistical data from reachability tree (note: can't do probability and statistics together). This options outputs the complete state space to a file, and with each state is produced its route from the initial state. However in the route the only transitions included are the ones which have been asked for during a run time input section. This is used in the determination of timing constraints on conditionally asynchronous mechanisms. For example if an algorithm is thought to only fail when one statement is interleaved by two others, the routes can be produced to only include the three relating blocks of Petri net transitions. Making analysis easier than looking through complete routes.

a - produce all bad routes (default is first only). One bad state is normally output with its complete route broken down and illustrated with which places were marked at each stage during the route from the initial marking. The a option produces all bad markings this than just one.

b - change the bad_example_file state output (default) to ref / remove / add output. Rather than list all places at each stage during the bad markings route, only list those which were $referenced$, *removed* or *added*.

k - change the bad_example_file to split writing / reading transitions. Rather than list all route transitions in one column, they are separated into writing process and reading process. Visualization of complex routes is aided.

d - tree differences between the current model and <tree file> note: the <tree file> MUST be subset of <input model file>. When trying to optimize the number of stages each model requires to represent an algorithm accurately, this option can be used to highlight differences between two models state space trees (Section 3.4).

e - same as d except lists head non-common nodes to common children, plus trial matching <tree file> routes. The previous option lists all non-common nodes, i.e. all states which are different from one tree to the other being compared. In practice, these non-common nodes tend to group together in branches. This option finds the starting or head non-common nodes and only lists those. It also tries to match routes in each tree to show how the non-common node affects the internal structure of the state space tree.

s - make the difference output (d) contain full state details. Same as above with full state details being listed.

i - make the difference output (d) contain ref / remove / add. Same as above but listing places which are $referenced$, *removed* or *added*.

v - output tree in HTML with the common/non common coloured/ref'd. A similar output to that of w above, except being coloured where the differences between the two trees under comparison are found.

z - difference output - node numbers in a separate file. The two tree being compared have all their nodes identified using integers. These integers are paired together to show which states are common between the two trees. This option outputs these paired state numbers in a separate file rather than included in the main difference output files.

g - output tree in GraphViz format highlighting common/non common. A combined state space is output in a form which can be visualized using GraphViz. The non-common routes are indicated using dashed branch lines.

m - same as g except common routes omitted to save space. Because larger models can produce huge state space trees, this option allows only the non-common routes to be output for visualization.

G - print debugging information. Option used during the construction and testing of the code.

X - print debugging information - in binary. Option used during the construction and testing of the code.

T - enable testing section of code. Option used during the construction and testing of the code.

Z - enable testing(2) section of code. Option used during the construction and testing of the code.

H - enable debug(3) output, i.e. routes to specific transitions. Option used during the construction and testing of the code.

I - Interactive model running. This option allows the user to run the Petri net transition by transition steering the route from a given state manually. Used when confirming bad state routes.

y - output writer/reader trees if alt_visited = $C/D/E$ (above). The memory optimization options above generate trees which contain all the writer specific states and a separate tree which contains all the reader specific states. This option outputs these trees to file.

h - display usage information (i.e. this). This displays the information given in Figure A.2, and explained in this section.

A.2 Model checking

The analysis performed on any model is only as accurate as the model itself. If a model is not an accurate representation of the thing it describes then any analysis on this will be inaccurate. The work in chapter 3 demonstrated how some statements in a concurrent environment cannot be accurately modelled using atomic actions, but require a two stage model. Once this has been determined, the Petri net model for each instruction needs to be implemented. In Section 4.2.3 it was shown that models based on each individual instruction were preferred over abstract models of the ACM algorithms. With a Petri net model based on each individual instruction it is easier to try and validate the model because each part of the algorithm directly corresponds to a part of the model, and each part of the model can be validated bit by bit rather than as a whole. There is no extra layer of abstraction between the algorithm and the model. (The downside to this is that abstract models can be much smaller in size, but the size trade off is regarded as unimportant compared to that of accuracy). An example of this is given in Figures A.3, A.4 and A.5, which show a single ACM algorithm instruction, its Petri net model in graphical form, and in textual form respectively.

$$wr : d[n, \overline{s[n]}] := input;$$

Figure A.3: A sample single ACM algorithm instruction.

Within *reach* extra code has been included to perform checks for model completeness, but no formal tests have been included. Within the input file format described in Section 4.3.1 includes an optional line labelled 'control=', this line contains a comma

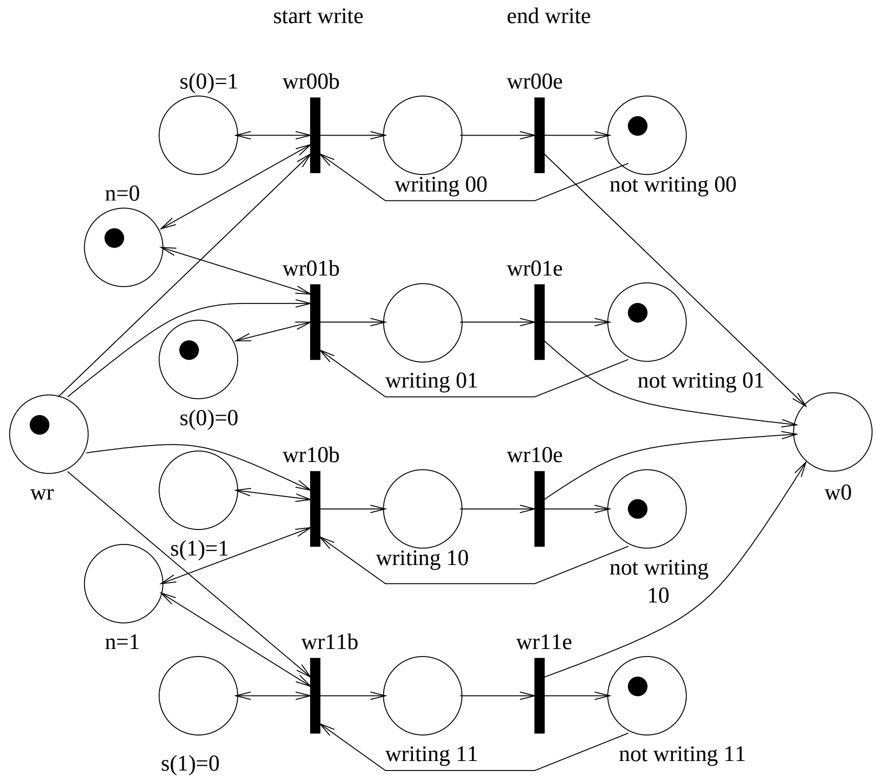


Figure A.4: The Petri net model in graphical form for the instruction given in Figure A.3.

```
wr00b,[wr,n=0,s0=1,not_writing_00],[n=0,s0=1,writing_00]
wr00e,[writing_00],[w0,not_writing_00]
wr01b,[wr,n=0,s0=0,not_writing_01],[n=0,s0=0,writing_01]
wr01e,[writing_01],[w0,not_writing_01]
wr10b,[wr,n=1,s1=1,not_writing_10],[n=1,s1=1,writing_10]
wr10e,[writing_10],[w0,not_writing_10]
wr11b,[wr,n=1,s1=0,not_writing_11],[n=1,s1=0,writing_11]
wr11e,[writing_11],[w0,not_writing_11]
```

Figure A.5: The Petri net model in textual form for the instruction given in Figures A.3 and A.4.

separated list of the control variables described in Section 4.2.4. This line is read in by *reach* and is checked to make sure it is a complete loop, with no breaks. The output from this check and others is in the form of a 'coverage' output file. This file has six sections. The first section lists each Petri net place label included in the model, with the number of input and output markings it appears in. An example of this is shown in Figure A.6. Because the models are cyclic in nature, each place label should appear an even number of times. If any place label only appears once, then the brackets are highlighted with stars (*) to indicate a possible typo in the place label. Two similar typos would cancel each other out, but this list of places being alphabetical can easily be scanned for obvious errors. In the example given the place 'coh_lost' is never used as transition input place, but it is used by four transitions in its output marking. (If the labels used in the models all differ by more than one symbol, then typos are less likely to be hidden)

```
Place label appearance count  [label    i/p    o/p] :-
 [coh_lost                 0       4]
 [l=0                      5       5]
 [l=1                      5       5]
 [n=0                      8       8]
 [n=1                      8       8]
 [not_reading_00           1       1]
 [not_reading_01           1       1]
 [not_reading_10           1       1]
 [not_reading_11           1       1]
 [not_writing_00           1       1]
 [not_writing_01           1       1]
 [not_writing_10           1       1]
```

Figure A.6: Place label appearance count from the coverage output file.

The second section of the coverage output file is the control loop check. An example of this is given in Figure A.7. This sample of a control variable loop check lists each of the control variables and then lists each one's next control variable. This gives the flow of control through the algorithm and highlights where any control problems may occur. Control places which are highlighted with a star indicate only one control route. From this example it can be seen that the control variable 'wr' splits into four different control variables 'writing_00', 'writing_01', 'writing_10' and 'writing_11'. This shows that when the writing statement 'wr' is enabled it passes control to the actual writing statements. In this case only one of the four will ever be enabled at any one time, but the route is there to all four. Each of these writing process control variables is highlighted as having only one control route, this is correct, implying there is only one entry route into an actual slot writing activity, which is correct in this example.

```
Next Control place list :-
[wr -> * writing_00 (1)*, * writing_01 (1)*, * writing_10 (1)*,
* writing_11 (1)*]
[writing_00 -> * w0 (1)*]
[writing_01 -> * w0 (1)*]
[writing_10 -> * w0 (1)*]
[writing_11 -> * w0 (1)*]
[w0 -> w1 (4)]
[w1 -> wr (8)]
[r0 -> * rc=0 (1)*, * rc=1 (1)*]
[rc=0 -> r1 (2)]
[rc=1 -> r1 (2)]
[r1 -> rd (16)]
[rd -> * reading_00 (1)*, * reading_01 (1)*, * reading_10 (1)*,
* reading_11 (1)*]
[reading_00 -> * r0 (1)*]
[reading_01 -> * r0 (1)*]
[reading_10 -> * r0 (1)*]
```

Figure A.7: Control variable loop check from the coverage output file.

The third section of the coverage output lists each transition within the model and lists which places it refers, removes and adds from the current state to the next state after it has fired. An example of this is given in Figure A.8. This example is taken from the writing section of the Simpson $4b + 7$ algorithm, and it shows how the start of each slots writing (transitions 'wr00b', 'wr01b', 'wr10b' and 'wr11b') each refer to the 'n' and '$s0$' or '$s1$' variables to determine which slot to write to, as the actual algorithm which states 'wr: slot[n, $\overline{s[n]}$] := input;'. each begin write statement also removes the token from 'wr' disabling the start write command, and also changes the status of the writing places from 'not_writing_??' to 'writing_??' (? implies either 0 or 1, but the slot number must be the same in both cases). This provides an extra check to make sure that a transition does not interfere with data it should not because of a mistake or a typo.

The fourth section in the coverage output file lists each place label and which transitions refer to it, remove it and add it to a state marking. An example of this is given in Figure A.9. In this example the places listed are involved with indicating that each slot is not being read or written to. It can be seen that each state is only

```
trans : wr00b :  {ref: n=0,s0=1} {remove: not_writing_00,wr} {add: writing_00}
trans : wr00e :  {ref: } {remove: writing_00} {add: not_writing_00,w0}
trans : wr01b :  {ref: n=0,s0=0} {remove: not_writing_01,wr} {add: writing_01}
trans : wr01e :  {ref: } {remove: writing_01} {add: not_writing_01,w0}
trans : wr10b :  {ref: n=1,s1=1} {remove: not_writing_10,wr} {add: writing_10}
trans : wr10e :  {ref: } {remove: writing_10} {add: not_writing_10,w0}
trans : wr11b :  {ref: n=1,s1=0} {remove: not_writing_11,wr} {add: writing_11}
trans : wr11e :  {ref: } {remove: writing_11} {add: not_writing_11,w0}
```

Figure A.8: Transition reference, remove and add place information from the coverage output file.

removed by the corresponding start 'rd' or start 'wr', and each slot is returned to this state by the end 'rd' or end 'wr' transition.

```
place : not_reading_00 : {ref: } {remove: rd00b } {add: rd00e }
place : not_reading_01 : {ref: } {remove: rd01b } {add: rd01e }
place : not_reading_10 : {ref: } {remove: rd10b } {add: rd10e }
place : not_reading_11 : {ref: } {remove: rd11b } {add: rd11e }
place : not_writing_00 : {ref: } {remove: wr00b } {add: wr00e }
place : not_writing_01 : {ref: } {remove: wr01b } {add: wr01e }
place : not_writing_10 : {ref: } {remove: wr10b } {add: wr10e }
place : not_writing_11 : {ref: } {remove: wr11b } {add: wr11e }
```

Figure A.9: Place reference, remove and add place information from the coverage output file.

The information presented so far is all constructed as the model is read into the reachability program. The last two sections of the coverage output file are constructed after the reachability analysis has been performed. Therefore the first four sections indicate if any errors have occurred within the model itself (syntactic errors) and the next two sections indicate errors in the actual algorithm (semantic errors). The fifth section contains a list of transitions and how many times each has been fired and also how many times each has been enabled. An example of this output is given in Figure A.10. This example shows the transition firing and enable coverage for some transitions from Simpson's $4b + 7$ algorithm, specifically the 'wr' writing command. Each transition is listed followed by the number of times it was actually fired, in brackets '{' and '}' is listed the number of times it was enabled to fire (but was not fired because the next state was already on the visited list and therefore did not need investigating again). Lastly a percentage figure shows how the ratio of how many times each transition was fired compared to the number of transitions fired to construct the complete state space. If any transition is never fired or enabled then the model must be thoroughly checked, as this indicates that this part of the algorithm is redundant and can be removed (an example of this was shown in Kirousis's $4b+39$ algorithm).

```
Transition coverage :-
[wr00b           55  {      88}     3.74%]
[wr00e           55  {      88}     3.74%]
[wr01b           42  {      88}     2.86%]
[wr01e           42  {      88}     2.86%]
[wr10b           51  {      88}     3.47%]
[wr10e           51  {      88}     3.47%]
[wr11b           41  {      88}     2.79%]
[wr11e           41  {      88}     2.79%]
```

Figure A.10: Transition firing and enabled information from the coverage output file.

The last part of the reachability output lists each place label and how many states it appears marked within. An example of this is given in Figure A.11. Each place label is listed with the number of states it was marked in, and a percentage compared to the total number of states. Therefore it can be seen that slot_00 was not being read from in 95.92% of the total states.

```
[not_reading_00       1412    95.92%]
[not_reading_01       1412    95.92%]
[not_reading_10       1412    95.92%]
[not_reading_11       1412    95.92%]
[not_writing_00       1384    94.02%]
[not_writing_01       1384    94.02%]
[not_writing_10       1384    94.02%]
[not_writing_11       1384    94.02%]
```

Figure A.11: Place label appearance in the complete reachability tree information from the coverage output file.

A.3 Reachability tree output from *reach*

As an extra aid in the process of understanding each algorithm *reach* can output the reachability tree as a list of states with each states next and previous states. This tree can be output in plain ASCII or in HTML format which links each next and previous state with its state in the list, and the user can click on these links and navigate through the state space tree. An example tree is given in Figure A.12. The first listed state is the initial state and this is listed as state 0. It can be seen from this example that the *bad* state is not listed, but the state that causes the bad state is listed at node number 3 with the single slot being read from and written to in the same state, and therefore at the same time.

```
Initial_node: 0 :[noread,nowrite]
next node: 1 :(startw) {ref: } {remove: nowrite} {add: write}
next node: 2 :(startr) {ref: } {remove: noread} {add: read}
prev node: 1 :(endw) {ref: } {remove: write} {add: nowrite}
prev node: 2 :(endr) {ref: } {remove: read} {add: noread}

node: 1 :[noread,write]
next node: 0 :(endw) {ref: } {remove: write} {add: nowrite}
next node: 3 :(startr) {ref: } {remove: noread} {add: read}
prev node: 3 :(endr) {ref: } {remove: read} {add: noread}
prev node: 0 :(startw) {ref: } {remove: nowrite} {add: write}

node: 2 :[nowrite,read]
next node: 3 :(startw) {ref: } {remove: nowrite} {add: write}
next node: 0 :(endr) {ref: } {remove: read} {add: noread}
prev node: 3 :(endw) {ref: } {remove: write} {add: nowrite}
prev node: 0 :(startr) {ref: } {remove: noread} {add: read}

node: 3 :[read,write]
next node: 2 :(endw) {ref: } {remove: write} {add: nowrite}
next node: 1 :(endr) {ref: } {remove: read} {add: noread}
prev node: 1 :(startr) {ref: } {remove: noread} {add: read}
prev node: 2 :(startw) {ref: } {remove: nowrite} {add: write}

END
```

Figure A.12: An example state space tree listing.

A.4 How to check if a model reduction is safe

The results presented in Chapter 3 showed that two stage modelling is required in some circumstances, otherwise states can be omitted from the state space and this can give inaccurate results when checking for the ACM operating properties. Omitting states also leads to missing route information within the state space. Figure A.13 shows the same route within a state space, one modelled by an atomic model and the other a two stage model. This relates to the hardware process of choosing and indicating the next state, or these two actions being combined into an atomic action. It can be seen that the atomic route ends up at a different final state to that of the two stage model, therefore showing that the atomic reduction is not accurate in this case.

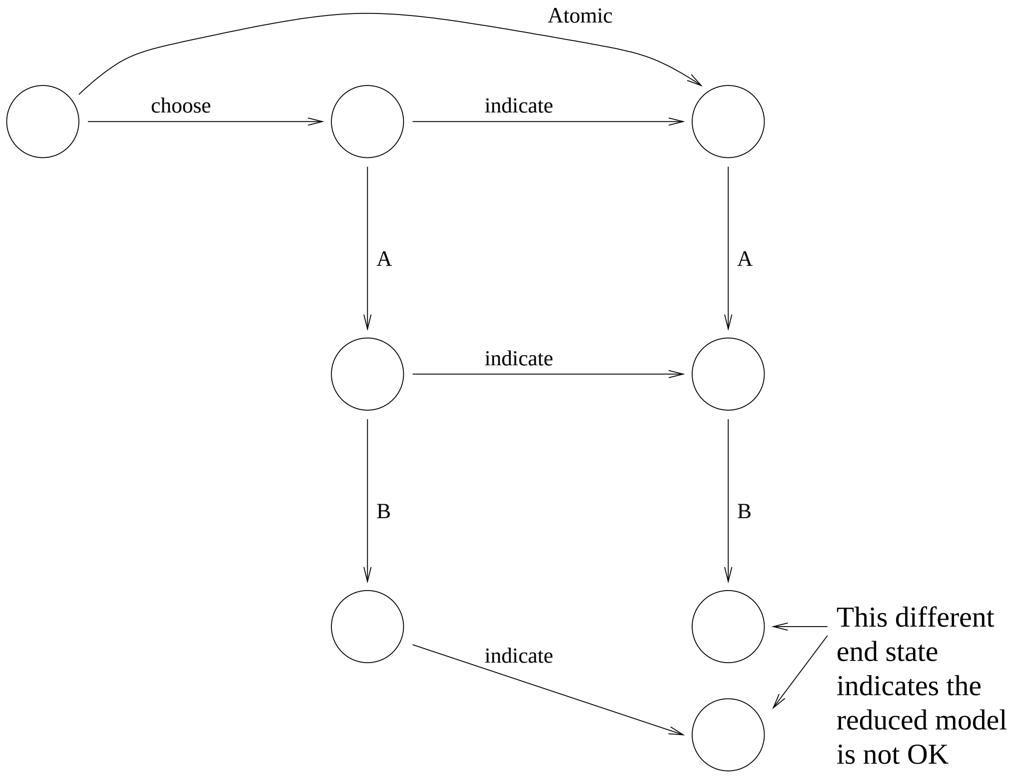


Figure A.13: Example routes through a state space.

Reach can help testing for these cases of non matching routes. One model is run through *reach* and the tree is output to file. Then the second model is run through *reach* and then compared to the tree file from the first run. Several output options are possible.

The default difference output from *reach* compares all the states from the two state spaces being compared, and then tries to match routes between the state spaces. Any routes which are different are highlighted because they will either be much longer in length, or it will go via completely different states. An example difference output is given in Figure A.14. In this example the first route uses a two stage model and the second an atomic model.

Other difference output options allow the user to concentrate on certain states without the total number of states in the way. The output trees can be output in a format which can be input to GraphViz [GbATR], which then draws the output trees for graphical inspection. This method is often more pleasing to the eye but can be extremely large, and difficult to inspect. Another textual output lists each common

```
node : 2 : [l=0,n=0,not_reading_00,not_reading_01,not_reading_10,not_reading_11,
not_writing_00,not_writing_01,not_writing_10,not_writing_11,r=0,rc=0,s0=0,s1=0,v
0=0,v1=0,wr]

4slot2_hard_coh : 0 : r0c0 : * 2 : r0r00i : 5

4slot2_hard_coh2.tree : 0 : r0r00 : 2
```

Figure A.14: An example difference output listing.

parent for a non-common parent, and then lists all that parent's common children and gives routes between the two using both state spaces. Again any differences are highlighted by routes using very different states or very long routes. An example of this output is given in Figure A.15.

It must be stressed that this test must be performed with all the operating property tests. The two examples already given in Figures A.14 and A.15 were taken from the model of Simpson's $4b+7$ algorithm with the instruction $r0$ as an atomic model and a two stage model. Under the test for data coherence the two stage model seems not to make any routing difference, but under the data freshness test, differences are found. One such difference is listed in Figure A.16. The model called 4slot2_hard_fre has ro modelled using a two stage model (the top route), and the second model is called 4slot2_hard_fre2 which has $r0$ modelled using an atomic instruction. The atomic instruction route is much longer than the two stage model route, which indicates that routes are omitted from the atomic instruction model and therefore this model is not accurate.

```
NOTE: routes are listed in reverse order!

------------------------------------------------
Head non-common node 2, common parent 0{0}.

next common 5{2} : r0r00.
common route 2 : r0r00 : 0.

next common 8{4} : r0r00 : 4 : wr01b.
common route 4 : r0r00 : 1 : wr01b : 0.

next common 12{7} : r0r00 : 7 : wr01e : 4 : wr01b.
common route 7 : r0r00 : 3 : wr01e : 1 : wr01b : 0.

next common 17{11} : r0r00 : 11 : w0s01 : 7 : wr01e : 4 : wr01b.
common route 11 : r0r00 : 6 : w0s01 : 3 : wr01e : 1 : wr01b : 0.

next common 22{15} : r0r00 : 16 : w1000 : 11 : w0s01 : 7 : wr01e : 4 : wr01b.
common route 15 : r0r00 : 10 : w1000 : 6 : w0s01 : 3 : wr01e : 1 : wr01b : 0.

next common 28{20} : r0r00 : 21 : wr11b : 16 : w1000 : 11 : w0s01 : 7 : wr01e :
4 : wr01b.
common route 20 : r0r00 : 14 : wr11b : 10 : w1000 : 6 : w0s01 : 3 : wr01e : 1 :
wr01b : 0.

next common 35{26} : r0r00 : 27 : wr11e : 21 : wr11b : 16 : w1000 : 11 : w0s01 :
 7 : wr01e : 4 : wr01b.
common route 26 : r0r00 : 19 : wr11e : 14 : wr11b : 10 : w1000 : 6 : w0s01 : 3 :
 wr01e : 1 : wr01b : 0.

next common 44{33} : r0r00 : 34 : w0s11 : 27 : wr11e : 21 : wr11b : 16 : w1000 :
 11 : w0s01 : 7 : wr01e : 4 : wr01b.
common route 33 : r0r00 : 25 : w0s11 : 19 : wr11e : 14 : wr11b : 10 : w1000 : 6
: w0s01 : 3 : wr01e : 1 : wr01b : 0.

etc.
```

Figure A.15: An example difference output listing listed by common parent to common children.

```
node : 871 : [enable,fresh_10,l=1,n=1,no_prev,not_reading_00,not_reading_01,not_
reading_10,not_reading_11,not_valid_00,not_valid_01,not_writing_00,not_writing_0
1,not_writing_10,not_writing_11,r0+,r1,r=1,s0=1,s1=0,v0=0,v1=0,valid_10,valid_11
,w1+,wr]

4slot2_hard_fre : 555 : rc1 : 613 : w1110 : # 666 : w1_normal : # 728 : cp11 : 8
01 : r0r01 : * 871 : wr11b : 933 : wr11e : # 1004 : ftop10 : # 1085 : wtofv11b :
 1159

4slot2_hard_fre2.tree : # 549 : r0r01 : 597 : w1111 : # 653 : w1_normal : # 720
: cp11 : 783 : wr00b : 836 : wr00e : # 897 : ftop10 : # 970 : wtofv00c : 1038 :
w0s00 : 1096 : r10000 : 1164 : pre_read_10b : # 1246 : rd10b : 1321 : rd10e : 13
84 : w1101 : # 1456 : w1_normal : # 1539 : cp10 : 1615 : clear_valid_00b : # 167
9 : clear_valid_01a : # 1748 : clear_valid_10b : # 1825 : clear_valid_11b : # 18
88 : ftov00 : # 1942 : nptov : # 2003 : r0_normal : # 2069 : r0r10 : 2133 : wr01
b : 2198 : wr01e : # 2272 : ftop00 : # 2354 : wtofv01c : 2429 : r10000 : 2498 :
w0s01 : 2577 : w1000 : # 2668 : w1_normal : # 2753 : cp00 : 2832 : wr11b : 2924
: wr11e : # 3021 : ftop01 : # 3101 : wtofv11c : 3173 : w0s11 : 3262 : pre_read_0
0e : # 3363 : rd00b : 3452 : w1010 : # 3535 : w1_normal : # 3634 : cp01 : 3736 :
 wr10b : 3819 : wr10e : # 3890 : ftop11 : # 3977 : wtofv10c : 4073 : w0s10 : 415
0 : w1110 : # 4227 : w1_normal : # 4327 : cp11 : 4430 : rd00e : 4514 : wr11b : 4
596 : clear_valid_00b : # 4695 : clear_valid_01b : # 4795 : clear_valid_10b : #
4877 : clear_valid_11b : # 4953 : ftov10 : # 5050 : nptov : # 5152 : r0_normal :
 # 781 : r0r01 : 834 : wr11e : # 894 : ftop10 : # 966 : wtofv11c : 1034
```

Figure A.16: An example difference output listing showing a missing route.

Appendix B

Complete Petri net models of ACMs

B.1 Peterson '$5b + 9$' Petri net models

B.1.1 Model for testing coherence

```
Peterson 1983 model for coherence
bad=[coherence_lost]
initial=[w0,r0,writing=0,reading=0,switch=0,sswitch=0,sswitch2=0,wflag=0,
sflag=0,sflag2=0,bad1=0,not_wr_buff1,not_wr_buff2,not_wr_copybuff,
not_rd_buff1,not_rd_buff2,not_rd_copybuff,buff1=xb,buff2=xb]
COM
COM writer
COM
COM w0: wflag := true;
COM
w0,[w0,wflag=0],[w1,wflag=1]
COM
COM w1: write buff1;
COM
st_wr_buff1,[w1,not_wr_buff1],[writing_buff1]
end_wr_buff1,[writing_buff1],[w2,not_wr_buff1]
COM
COM w2: switch := not switch;
COM
w2a,[w2,switch=0],[w3,switch=1]
w2b,[w2,switch=1],[w3,switch=0]
COM
COM w3: wflag := false;
COM
w3,[w3,wflag=1],[w4,wflag=0]
COM
COM w4: if reading = writing then
COM
w4a,[w4,reading=0,writing=0],[w5,reading=0,writing=0]
w4b,[w4,reading=1,writing=1],[w5,reading=1,writing=1]
w4c,[w4,reading=0,writing=1],[w8,reading=0,writing=1]
w4d,[w4,reading=1,writing=0],[w8,reading=1,writing=0]
COM
```

```
COM w5: write copybuff;
COM
st_wr_copybuff,[w5,not_wr_copybuff],[writing_copybuff]
end_wr_copybuff,[writing_copybuff],[w6,not_wr_copybuff]
COM
COM w6: writing := reading;
COM
w6a,[w6,reading=0,+0writing=0,+0writing=1],[w8,reading=0,writing=0]
w6b,[w6,reading=1,+0writing=0,+0writing=1],[w8,reading=1,writing=1]
COM
COM w7: endif;
COM
COM w8: write buff2;
COM
st_wr_buff2,[w8,not_wr_buff2],[writing_buff2]
end_wr_buff2,[writing_buff2],[w0,not_wr_buff2]
COM
COM reader
COM
COM Clear X
COM
clear_buff,[r0,+0buff1=xb,+0buff1=x,+1buff2=xb,+1buff2=x],[r0a,buff1=xb,
buff2=xb]
COM
COM r0: reading := not writing;
COM
r0a,[r0a,writing=1,+0reading=0,+0reading=1],[r1,writing=1,reading=0]
r0b,[r0a,writing=0,+0reading=0,+0reading=1],[r1,writing=0,reading=1]
COM
COM r1: sflag := wflag;
COM
r1a,[r1,wflag=0,+0sflag=0,+0sflag=1],[r2,wflag=0,sflag=0]
r1b,[r1,wflag=1,+0sflag=0,+0sflag=1],[r2,wflag=1,sflag=1]
COM
COM r2: sswitch := switch;
COM
r2a,[r2,switch=0,+0sswitch=0,+0sswitch=1],[r3,switch=0,sswitch=0]
r2b,[r2,switch=1,+0sswitch=0,+0sswitch=1],[r3,switch=1,sswitch=1]
COM
COM r3: read buff1;
COM
st_rd_buff1,[r3,not_rd_buff1],[reading_buff1]
end_rd_buff1,[reading_buff1],[r4,not_rd_buff1]
COM
COM r4: sflag2 := wflag;
COM
r4a,[r4,wflag=0,+0sflag2=0,+0sflag2=1],[r5,wflag=0,sflag2=0]
r4b,[r4,wflag=1,+0sflag2=0,+0sflag2=1],[r5,wflag=1,sflag2=1]
COM
COM r5: sswitch2 := switch;
COM
r5a,[r5,switch=0,+0sswitch2=0,+0sswitch2=1],[r6,switch=0,sswitch2=0]
r5b,[r5,switch=1,+0sswitch2=0,+0sswitch2=1],[r6,switch=1,sswitch2=1]
COM
COM r6: read buff2;
COM
st_rd_buff2,[r6,not_rd_buff2],[reading_buff2]
end_rd_buff2,[reading_buff2],[r7,not_rd_buff2]
COM
COM r7: bad1 := sswitch /= (sswitch2 or sflag or sflag2);
COM
r7a,[r7,+0bad1=0,+0bad1=1,sswitch=0,sswitch2=0,sflag=0,sflag2=0],[r9,bad1=0,
sswitch=0,sswitch2=0,sflag=0,sflag2=0]
r7b,[r7,+0bad1=0,+0bad1=1,sswitch=1,sswitch2=1,sflag=1,sflag2=1],[r9,bad1=0,
sswitch=1,sswitch2=1,sflag=1,sflag2=1]
r7c,[r7,+0bad1=0,+0bad1=1,sswitch=0,sswitch2=1],[r9,bad1=1,sswitch=0,
```

```
sswitch2=1]
r7d,[r7,+0bad1=0,+0bad1=1,sswitch=1,sswitch2=0],[r9,bad1=1,sswitch=1,
sswitch2=0]
r7e,[r7,+0bad1=0,+0bad1=1,sswitch=0,sflag=1],[r9,bad1=1,sswitch=0,sflag=1]
r7f,[r7,+0bad1=0,+0bad1=1,sswitch=1,sflag=0],[r9,bad1=1,sswitch=1,sflag=0]
r7g,[r7,+0bad1=0,+0bad1=1,sswitch=0,sflag2=1],[r9,bad1=1,sswitch=0,sflag2=1]
r7h,[r7,+0bad1=0,+0bad1=1,sswitch=1,sflag2=0],[r9,bad1=1,sswitch=1,sflag2=0]
COM
COM r8: if reading = writing then
COM
COM r9: return copybuff1 value;
COM
st_rd_copybuffa,[r9,reading=0,writing=0,not_rd_copybuff],[reading=0,
writing=0,reading_copybuff]
st_rd_copybuffb,[r9,reading=1,writing=1,not_rd_copybuff],[reading=1,
writing=1,reading_copybuff]
end_rd_copybuff,[reading_copybuff],[r0,not_rd_copybuff]
COM
COM r10: else if bad1 then return buff2 value;
COM
r10a,[r9,reading=0,writing=1,bad1=1,buff2=xb],[r0,reading=0,writing=1,
bad1=1,buff2=xb]
r10b,[r9,reading=1,writing=0,bad1=1,buff2=xb],[r0,reading=1,writing=0,
bad1=1,buff2=xb]
r10c,[r9,reading=0,writing=1,bad1=1,buff2=x],[r0,reading=0,writing=1,
bad1=1,buff2=x,coherence_lost]
r10d,[r9,reading=1,writing=0,bad1=1,buff2=x],[r0,reading=1,writing=0,
bad1=1,buff2=x,coherence_lost]
COM
COM r11: else return buff1 value;
COM
r11a,[r9,reading=0,writing=1,bad1=0,buff1=xb],[r0,reading=0,writing=1,
bad1=0,buff1=xb]
r11b,[r9,reading=1,writing=0,bad1=0,buff1=xb],[r0,reading=1,writing=0,
bad1=0,buff1=xb]
r11c,[r9,reading=0,writing=1,bad1=0,buff1=x],[r0,reading=0,writing=1,
bad1=0,buff1=x,coherence_lost]
r11d,[r9,reading=1,writing=0,bad1=0,buff1=x],[r0,reading=1,writing=0,
bad1=0,buff1=x,coherence_lost]
COM
COM check coherence
COM
coh1,[writing_copybuff,reading_copybuff],[coherence_lost]
coh2,[writing_buff1,reading_buff1,buff1=xb],[writing_buff1,reading_buff1,
buff1=x]
coh3,[writing_buff2,reading_buff2,buff2=xb],[writing_buff2,reading_buff2,
buff2=x]
END
```

B.2 Kirousis '$4b + 39$' Petri net models

B.2.1 Model for testing coherence

```
Kirousis 4 track software model
bad=[coherence_lost]
mask=[w1,w2,w3,w2_temp1,w3_temp1,w3_temp1d,w3_temp2,w3_temp4,w4,w5,w5_temp1,
w5_temp2,A0=E,A0=01,A0=02,A0=03,A0=04,A0=11,A0=12,A0=13,A0=14,A0=21,A0=22,
A0=23,A0=24,A1=E,A1=01,A1=02,A1=03,A1=04,A1=11,A1=12,A1=13,A1=14,A1=21,
A1=22,A1=23,A1=24,A2=E,A2=01,A2=02,A2=03,A2=04,A2=11,A2=12,A2=13,A2=14,
A2=21,A2=22,A2=23,A2=24,wl=0,wl=1,wl=2,v1=1,v1=2,v1=3,v1=4,v2=1,v2=2,v2=3,
v2=4,wt-=1,wt-=2,wt-=3,wt-=4,wt=1,wt=2,wt=3,wt=4,wtg=0,wtg=1,wtg=2,temp1,
```

```
temp1b,temp2,temp2b,temp3,temp3b,temp4,temp4b,wRQ=P,wRQ=E,writing_1,
writing_2,writing_3,writing_4]
initial=[enable,w1,r1,RQ0=E,RQ1=E,RQ2=E,A0=E,A1=E,A2=E,wl=0,v1=1,v2=1,wt-=1,
wt=1,wtg=0,rt=1,rl=-1,rtg=0,temp1,temp2,temp3,temp4]
COM read RQwl
w1a,[w1,wl=0,RQ0=P,enable],[w2,wl=0,RQ0=P,wRQ=P]
w1b,[w1,wl=0,RQ0=E,enable],[w2,wl=0,RQ0=E,wRQ=E]
w1c,[w1,wl=1,RQ1=P,enable],[w2,wl=1,RQ1=P,wRQ=P]
w1d,[w1,wl=1,RQ1=E,enable],[w2,wl=1,RQ1=E,wRQ=E]
w1e,[w1,wl=2,RQ2=P,enable],[w2,wl=2,RQ2=P,wRQ=P]
w1f,[w1,wl=2,RQ2=E,enable],[w2,wl=2,RQ2=E,wRQ=E]
COM if RQwl=E do nothing
w2a,[w2,wRQ=E],[w3]
COM if RQwl=P then wl=wl+1 and
w2b,[w2,wl=0,wRQ=P],[w2_temp1,wl=1]
w2c,[w2,wl=1,wRQ=P],[w2_temp1,wl=2]
w2d,[w2,wl=2,wRQ=P],[w2_temp1,wl=0]
COM vb:={wt,wt-}; (v1=wt) (v2=wt-)
w2fa,[w2_temp1,wt=1,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=1,v1=1,v2=1]
w2fb,[w2_temp1,wt=1,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=2,v1=1,v2=2]
w2fc,[w2_temp1,wt=1,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=3,v1=1,v2=3]
w2fd,[w2_temp1,wt=1,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=4,v1=1,v2=4]
w2ga,[w2_temp1,wt=2,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=1,v1=2,v2=1]
w2gb,[w2_temp1,wt=2,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=2,v1=2,v2=2]
w2gc,[w2_temp1,wt=2,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=3,v1=2,v2=3]
w2gd,[w2_temp1,wt=2,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=4,v1=2,v2=4]
w2ha,[w2_temp1,wt=3,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=1,v1=3,v2=1]
w2hb,[w2_temp1,wt=3,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=2,v1=3,v2=2]
w2hc,[w2_temp1,wt=3,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=3,v1=3,v2=3]
w2hd,[w2_temp1,wt=3,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=4,v1=3,v2=4]
w2ia,[w2_temp1,wt=4,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=1,v1=4,v2=1]
w2ib,[w2_temp1,wt=4,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=2,v1=4,v2=2]
w2ic,[w2_temp1,wt=4,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=3,v1=4,v2=3]
w2id,[w2_temp1,wt=4,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=4,v1=4,v2=4]
COM wt-:=wt;
w3a,[w3,wt=1,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=1]
w3b,[w3,wt=2,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=2]
w3c,[w3,wt=3,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=3]
w3d,[w3,wt=4,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=4]
COM wt:=min({1,2,3,4}-({wt-} U vb)); (setup {1,2,3,4})
w3aa,[w3_temp1,+0temp1,+0temp1b,+1temp2,+1temp2b,+2temp3,+2temp3b,+3temp4,
+3temp4b],[temp1,temp2,temp3,temp4,w3_temp1d]
COM (remove wt-)
w3ba,[w3_temp1d,wt-=1,temp1],[w3_temp2,wt-=1,temp1b]
w3bb,[w3_temp1d,wt-=2,temp2],[w3_temp2,wt-=2,temp2b]
w3bc,[w3_temp1d,wt-=3,temp3],[w3_temp2,wt-=3,temp3b]
w3bd,[w3_temp1d,wt-=4,temp4],[w3_temp2,wt-=4,temp4b]
COM (remove v1)
w3ca,[w3_temp2,v1=1,+0temp1,+0temp1b],[w3_temp3,v1=1,temp1b]
w3cc,[w3_temp2,v1=2,+0temp2,+0temp2b],[w3_temp3,v1=2,temp2b]
```

```
w3ce,[w3_temp2,v1=3,+0temp3,+0temp3b],[w3_temp3,v1=3,temp3b]
w3cg,[w3_temp2,v1=4,+0temp4,+0temp4b],[w3_temp3,v1=4,temp4b]
COM (remove v2)
w3da,[w3_temp3,v2=1,+0temp1,+0temp1b],[w3_temp4,v2=1,temp1b]
w3dc,[w3_temp3,v2=2,+0temp2,+0temp2b],[w3_temp4,v2=2,temp2b]
w3de,[w3_temp3,v2=3,+0temp3,+0temp3b],[w3_temp4,v2=3,temp3b]
w3dg,[w3_temp3,v2=4,+0temp4,+0temp4b],[w3_temp4,v2=4,temp4b]
COM (assign wt)
w3ea,[w3_temp4,temp1],[w4,temp1,wt=1,enable]
w3eb,[w3_temp4,temp1b,temp2],[w4,temp1b,temp2,wt=2,enable]
w3ec,[w3_temp4,temp1b,temp2b,temp3],[w4,temp1b,temp2b,temp3,wt=3,enable]
w3ed,[w3_temp4,temp1b,temp2b,temp3b,temp4],[w4,temp1b,temp2b,temp3b,temp4,
wt=4,enable]
COM write to track wt
w4a,[w4,wt=1,enable],[wt=1,writing_1,enable]
w4b,[writing_1,enable],[w5,enable]
w4c,[w4,wt=2,enable],[wt=2,writing_2,enable]
w4d,[writing_2,enable],[w5,enable]
w4e,[w4,wt=3,enable],[wt=3,writing_3,enable]
w4f,[writing_3,enable],[w5,enable]
w4g,[w4,wt=4,enable],[wt=4,writing_4,enable]
w4h,[writing_4,enable],[w5,enable]
COM wtg := wtg + 1;
w5aa,[w5,wtg=0,enable],[w5_temp1,wtg=1]
w5ab,[w5,wtg=1,enable],[w5_temp1,wtg=2]
w5ac,[w5,wtg=2,enable],[w5_temp1,wtg=0]
COM write E on Awl+1;
w5bb,[w5_temp1,wl=0,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24],[w5_temp2,wl=0,A1=E,enable]
w5bd,[w5_temp1,wl=1,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24],[w5_temp2,wl=1,A2=E,enable]
w5bf,[w5_temp1,wl=2,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24],[w5_temp2,wl=2,A0=E,enable]
COM write (wtg,wt) on Awl;
w5ca,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=1,enable],[w1,wl=0,
wtg=0,wt=1,A0=01,enable]
w5cb,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=2,enable],[w1,wl=0,
wtg=0,wt=2,A0=02,enable]
w5cc,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=3,enable],[w1,wl=0,
wtg=0,wt=3,A0=03,enable]
w5cd,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=4,enable],[w1,wl=0,
wtg=0,wt=4,A0=04,enable]
w5ce,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=1,enable],[w1,wl=0,
wtg=1,wt=1,A0=11,enable]
w5cf,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=2,enable],[w1,wl=0,
wtg=1,wt=2,A0=12,enable]
w5cg,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=3,enable],[w1,wl=0,
wtg=1,wt=3,A0=13,enable]
w5ch,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=4,enable],[w1,wl=0,
wtg=1,wt=4,A0=14,enable]
w5ci,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=1,enable],[w1,wl=0,
wtg=2,wt=1,A0=21,enable]
w5cj,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=2,enable],[w1,wl=0,
wtg=2,wt=2,A0=22,enable]
w5ck,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=3,enable],[w1,wl=0,
```

```
wtg=2,wt=3,A0=23,enable]
w5cl,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=4,enable],[w1,wl=0,
wtg=2,wt=4,A0=24,enable]
w5da,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=1,enable],[w1,wl=1,
wtg=0,wt=1,A1=01,enable]
w5db,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=2,enable],[w1,wl=1,
wtg=0,wt=2,A1=02,enable]
w5dc,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=3,enable],[w1,wl=1,
wtg=0,wt=3,A1=03,enable]
w5dd,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=4,enable],[w1,wl=1,
wtg=0,wt=4,A1=04,enable]
w5de,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=1,enable],[w1,wl=1,
wtg=1,wt=1,A1=11,enable]
w5df,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=2,enable],[w1,wl=1,
wtg=1,wt=2,A1=12,enable]
w5dg,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=3,enable],[w1,wl=1,
wtg=1,wt=3,A1=13,enable]
w5dh,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=4,enable],[w1,wl=1,
wtg=1,wt=4,A1=14,enable]
w5di,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=1,enable],[w1,wl=1,
wtg=2,wt=1,A1=21,enable]
w5dj,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=2,enable],[w1,wl=1,
wtg=2,wt=2,A1=22,enable]
w5dk,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=3,enable],[w1,wl=1,
wtg=2,wt=3,A1=23,enable]
w5dl,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=4,enable],[w1,wl=1,
wtg=2,wt=4,A1=24,enable]
w5ea,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=1,enable],[w1,wl=2,
wtg=0,wt=1,A2=01,enable]
w5eb,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=2,enable],[w1,wl=2,
wtg=0,wt=2,A2=02,enable]
w5ec,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=3,enable],[w1,wl=2,
wtg=0,wt=3,A2=03,enable]
w5ed,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=4,enable],[w1,wl=2,
wtg=0,wt=4,A2=04,enable]
w5ee,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=1,enable],[w1,wl=2,
wtg=1,wt=1,A2=11,enable]
w5ef,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=2,enable],[w1,wl=2,
wtg=1,wt=2,A2=12,enable]
w5eg,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=3,enable],[w1,wl=2,
wtg=1,wt=3,A2=13,enable]
w5eh,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=4,enable],[w1,wl=2,
wtg=1,wt=4,A2=14,enable]
w5ei,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=1,enable],[w1,wl=2,
```

```
wtg=2,wt=1,A2=21,enable]
w5ej,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=2,enable],[w1,wl=2,
wtg=2,wt=2,A2=22,enable]
w5ek,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=3,enable],[w1,wl=2,
wtg=2,wt=3,A2=23,enable]
w5el,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=4,enable],[w1,wl=2,
wtg=2,wt=4,A2=24,enable]
COM - The READER - rl := rl+1;
r1a,[r1,rl=-1,enable],[r2,rl=0]
r1b,[r1,rl=0,enable],[r2,rl=1]
r1c,[r1,rl=1,enable],[r2,rl=2]
r1d,[r1,rl=2,enable],[r2,rl=0]
COM read Arl;
r2aa,[r2,rl=0,A0=E],[r3,rl=0,A0=E,Arl=E]
r2ab,[r2,rl=0,A0=01],[r3,rl=0,A0=01,Arl=01,Arln=E]
r2ac,[r2,rl=0,A0=02],[r3,rl=0,A0=02,Arl=02,Arln=E]
r2ad,[r2,rl=0,A0=03],[r3,rl=0,A0=03,Arl=03,Arln=E]
r2ae,[r2,rl=0,A0=04],[r3,rl=0,A0=04,Arl=04,Arln=E]
r2af,[r2,rl=0,A0=11],[r3,rl=0,A0=11,Arl=11,Arln=E]
r2ag,[r2,rl=0,A0=12],[r3,rl=0,A0=12,Arl=12,Arln=E]
r2ah,[r2,rl=0,A0=13],[r3,rl=0,A0=13,Arl=13,Arln=E]
r2ai,[r2,rl=0,A0=14],[r3,rl=0,A0=14,Arl=14,Arln=E]
r2aj,[r2,rl=0,A0=21],[r3,rl=0,A0=21,Arl=21,Arln=E]
r2ak,[r2,rl=0,A0=22],[r3,rl=0,A0=22,Arl=22,Arln=E]
r2al,[r2,rl=0,A0=23],[r3,rl=0,A0=23,Arl=23,Arln=E]
r2am,[r2,rl=0,A0=24],[r3,rl=0,A0=24,Arl=24,Arln=E]
r2ba,[r2,rl=1,A1=E],[r3,rl=1,A1=E,Arl=E]
r2bb,[r2,rl=1,A1=01],[r3,rl=1,A1=01,Arl=01,Arln=E]
r2bc,[r2,rl=1,A1=02],[r3,rl=1,A1=02,Arl=02,Arln=E]
r2bd,[r2,rl=1,A1=03],[r3,rl=1,A1=03,Arl=03,Arln=E]
r2be,[r2,rl=1,A1=04],[r3,rl=1,A1=04,Arl=04,Arln=E]
r2bf,[r2,rl=1,A1=11],[r3,rl=1,A1=11,Arl=11,Arln=E]
r2bg,[r2,rl=1,A1=12],[r3,rl=1,A1=12,Arl=12,Arln=E]
r2bh,[r2,rl=1,A1=13],[r3,rl=1,A1=13,Arl=13,Arln=E]
r2bi,[r2,rl=1,A1=14],[r3,rl=1,A1=14,Arl=14,Arln=E]
r2bj,[r2,rl=1,A1=21],[r3,rl=1,A1=21,Arl=21,Arln=E]
r2bk,[r2,rl=1,A1=22],[r3,rl=1,A1=22,Arl=22,Arln=E]
r2bl,[r2,rl=1,A1=23],[r3,rl=1,A1=23,Arl=23,Arln=E]
r2bm,[r2,rl=1,A1=24],[r3,rl=1,A1=24,Arl=24,Arln=E]
r2ca,[r2,rl=2,A2=E],[r3,rl=2,A2=E,Arl=E]
r2cb,[r2,rl=2,A2=01],[r3,rl=2,A2=01,Arl=01,Arln=E]
r2cc,[r2,rl=2,A2=02],[r3,rl=2,A2=02,Arl=02,Arln=E]
r2cd,[r2,rl=2,A2=03],[r3,rl=2,A2=03,Arl=03,Arln=E]
r2ce,[r2,rl=2,A2=04],[r3,rl=2,A2=04,Arl=04,Arln=E]
r2cf,[r2,rl=2,A2=11],[r3,rl=2,A2=11,Arl=11,Arln=E]
r2cg,[r2,rl=2,A2=12],[r3,rl=2,A2=12,Arl=12,Arln=E]
r2ch,[r2,rl=2,A2=13],[r3,rl=2,A2=13,Arl=13,Arln=E]
r2ci,[r2,rl=2,A2=14],[r3,rl=2,A2=14,Arl=14,Arln=E]
r2cj,[r2,rl=2,A2=21],[r3,rl=2,A2=21,Arl=21,Arln=E]
r2ck,[r2,rl=2,A2=22],[r3,rl=2,A2=22,Arl=22,Arln=E]
r2cl,[r2,rl=2,A2=23],[r3,rl=2,A2=23,Arl=23,Arln=E]
r2cm,[r2,rl=2,A2=24],[r3,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl=E then do rl=rl-1; reread Arl;
r3aa,[r3,Arl=E,rl=0],[r3a_temp1,rl=2,enable]
r3ab,[r3,Arl=E,rl=1],[r3a_temp1,rl=0,enable]
r3ac,[r3,Arl=E,rl=2],[r3a_temp1,rl=1,enable]
COM (reread Arl)
r3aaa,[r3a_temp1,rl=0,A0=E,enable],[r3a_temp2,rl=0,A0=E,Arl=E]
r3aab,[r3a_temp1,rl=0,A0=01,enable],[r3a_temp2,rl=0,A0=01,Arl=01,Arln=E]
r3aac,[r3a_temp1,rl=0,A0=02,enable],[r3a_temp2,rl=0,A0=02,Arl=02,Arln=E]
r3aad,[r3a_temp1,rl=0,A0=03,enable],[r3a_temp2,rl=0,A0=03,Arl=03,Arln=E]
r3aae,[r3a_temp1,rl=0,A0=04,enable],[r3a_temp2,rl=0,A0=04,Arl=04,Arln=E]
r3aaf,[r3a_temp1,rl=0,A0=11,enable],[r3a_temp2,rl=0,A0=11,Arl=11,Arln=E]
```

```
r3aag,[r3a_temp1,rl=0,A0=12,enable],[r3a_temp2,rl=0,A0=12,Arl=12,Arln=E]
r3aah,[r3a_temp1,rl=0,A0=13,enable],[r3a_temp2,rl=0,A0=13,Arl=13,Arln=E]
r3aai,[r3a_temp1,rl=0,A0=14,enable],[r3a_temp2,rl=0,A0=14,Arl=14,Arln=E]
r3aaj,[r3a_temp1,rl=0,A0=21,enable],[r3a_temp2,rl=0,A0=21,Arl=21,Arln=E]
r3aak,[r3a_temp1,rl=0,A0=22,enable],[r3a_temp2,rl=0,A0=22,Arl=22,Arln=E]
r3aal,[r3a_temp1,rl=0,A0=23,enable],[r3a_temp2,rl=0,A0=23,Arl=23,Arln=E]
r3aam,[r3a_temp1,rl=0,A0=24,enable],[r3a_temp2,rl=0,A0=24,Arl=24,Arln=E]
r3aba,[r3a_temp1,rl=1,A1=E,enable],[r3a_temp2,rl=1,A1=E,Arl=E]
r3abb,[r3a_temp1,rl=1,A1=01,enable],[r3a_temp2,rl=1,A1=01,Arl=01,Arln=E]
r3abc,[r3a_temp1,rl=1,A1=02,enable],[r3a_temp2,rl=1,A1=02,Arl=02,Arln=E]
r3abd,[r3a_temp1,rl=1,A1=03,enable],[r3a_temp2,rl=1,A1=03,Arl=03,Arln=E]
r3abe,[r3a_temp1,rl=1,A1=04,enable],[r3a_temp2,rl=1,A1=04,Arl=04,Arln=E]
r3abf,[r3a_temp1,rl=1,A1=11,enable],[r3a_temp2,rl=1,A1=11,Arl=11,Arln=E]
r3abg,[r3a_temp1,rl=1,A1=12,enable],[r3a_temp2,rl=1,A1=12,Arl=12,Arln=E]
r3abh,[r3a_temp1,rl=1,A1=13,enable],[r3a_temp2,rl=1,A1=13,Arl=13,Arln=E]
r3abi,[r3a_temp1,rl=1,A1=14,enable],[r3a_temp2,rl=1,A1=14,Arl=14,Arln=E]
r3abj,[r3a_temp1,rl=1,A1=21,enable],[r3a_temp2,rl=1,A1=21,Arl=21,Arln=E]
r3abk,[r3a_temp1,rl=1,A1=22,enable],[r3a_temp2,rl=1,A1=22,Arl=22,Arln=E]
r3abl,[r3a_temp1,rl=1,A1=23,enable],[r3a_temp2,rl=1,A1=23,Arl=23,Arln=E]
r3abm,[r3a_temp1,rl=1,A1=24,enable],[r3a_temp2,rl=1,A1=24,Arl=24,Arln=E]
r3aca,[r3a_temp1,rl=2,A2=E,enable],[r3a_temp2,rl=2,A2=E,Arl=E]
r3acb,[r3a_temp1,rl=2,A2=01,enable],[r3a_temp2,rl=2,A2=01,Arl=01,Arln=E]
r3acc,[r3a_temp1,rl=2,A2=02,enable],[r3a_temp2,rl=2,A2=02,Arl=02,Arln=E]
r3acd,[r3a_temp1,rl=2,A2=03,enable],[r3a_temp2,rl=2,A2=03,Arl=03,Arln=E]
r3ace,[r3a_temp1,rl=2,A2=04,enable],[r3a_temp2,rl=2,A2=04,Arl=04,Arln=E]
r3acf,[r3a_temp1,rl=2,A2=11,enable],[r3a_temp2,rl=2,A2=11,Arl=11,Arln=E]
r3acg,[r3a_temp1,rl=2,A2=12,enable],[r3a_temp2,rl=2,A2=12,Arl=12,Arln=E]
r3ach,[r3a_temp1,rl=2,A2=13,enable],[r3a_temp2,rl=2,A2=13,Arl=13,Arln=E]
r3aci,[r3a_temp1,rl=2,A2=14,enable],[r3a_temp2,rl=2,A2=14,Arl=14,Arln=E]
r3acj,[r3a_temp1,rl=2,A2=21,enable],[r3a_temp2,rl=2,A2=21,Arl=21,Arln=E]
r3ack,[r3a_temp1,rl=2,A2=22,enable],[r3a_temp2,rl=2,A2=22,Arl=22,Arln=E]
r3acl,[r3a_temp1,rl=2,A2=23,enable],[r3a_temp2,rl=2,A2=23,Arl=23,Arln=E]
r3acm,[r3a_temp1,rl=2,A2=24,enable],[r3a_temp2,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl n= E and tag of Arl=rtg+1 then store value of Arl into (rtg,rt) od
COM tag = 1st digit of Arl = (rtg,rt)
r3ada,[r3a_temp2,Arln=E,rtg=2,Arl=01,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=1,enable]
r3adb,[r3a_temp2,Arln=E,rtg=2,Arl=02,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=2,enable]
r3adc,[r3a_temp2,Arln=E,rtg=2,Arl=03,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=3,enable]
r3add,[r3a_temp2,Arln=E,rtg=2,Arl=04,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=4,enable]
r3ade,[r3a_temp2,Arln=E,rtg=0,Arl=11,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=1,enable]
r3adf,[r3a_temp2,Arln=E,rtg=0,Arl=12,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=2,enable]
r3adg,[r3a_temp2,Arln=E,rtg=0,Arl=13,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=3,enable]
r3adh,[r3a_temp2,Arln=E,rtg=0,Arl=14,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=4,enable]
r3adi,[r3a_temp2,Arln=E,rtg=1,Arl=21,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=1,enable]
r3adj,[r3a_temp2,Arln=E,rtg=1,Arl=22,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=2,enable]
r3adk,[r3a_temp2,Arln=E,rtg=1,Arl=23,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=3,enable]
r3adl,[r3a_temp2,Arln=E,rtg=1,Arl=24,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=4,enable]
COM if Arl = E then move on to r4 read
r3ads,[r3a_temp2,Arl=E],[r4,enable]
COM if Arl n= E, but tag of Arl n= rtg + 1
r3adt,[r3a_temp2,Arln=E,rtg=2,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=21,
+0Arl=22,+0Arl=23,+0Arl=24],[r4,rtg=2,enable]
r3adu,[r3a_temp2,Arln=E,rtg=0,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=21,
+0Arl=22,+0Arl=23,+0Arl=24],[r4,rtg=0,enable]
```

```
r3adv,[r3a_temp2,Arln=E,rtg=1,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=01,
+0Arl=02,+0Arl=03,+0Arl=04],[r4,rtg=1,enable]
COM else do - store value of Arl into s;
r3baa,[r3,Arln=E,Arl=01],[r3b_temp1,s=01,enable]
r3bab,[r3,Arln=E,Arl=02],[r3b_temp1,s=02,enable]
r3bac,[r3,Arln=E,Arl=03],[r3b_temp1,s=03,enable]
r3bad,[r3,Arln=E,Arl=04],[r3b_temp1,s=04,enable]
r3bae,[r3,Arln=E,Arl=11],[r3b_temp1,s=11,enable]
r3baf,[r3,Arln=E,Arl=12],[r3b_temp1,s=12,enable]
r3bag,[r3,Arln=E,Arl=13],[r3b_temp1,s=13,enable]
r3bah,[r3,Arln=E,Arl=14],[r3b_temp1,s=14,enable]
r3bai,[r3,Arln=E,Arl=21],[r3b_temp1,s=21,enable]
r3baj,[r3,Arln=E,Arl=22],[r3b_temp1,s=22,enable]
r3bak,[r3,Arln=E,Arl=23],[r3b_temp1,s=23,enable]
r3bal,[r3,Arln=E,Arl=24],[r3b_temp1,s=24,enable]
COM write E onto RQrl+1
r3bba,[r3b_temp1,rl=0,+0RQ1=E,+0RQ1=P,enable],[r3b_temp2,rl=0,RQ1=E,enable]
r3bbb,[r3b_temp1,rl=1,+0RQ2=E,+0RQ2=P,enable],[r3b_temp2,rl=1,RQ2=E,enable]
r3bbc,[r3b_temp1,rl=2,+0RQ0=E,+0RQ0=P,enable],[r3b_temp2,rl=2,RQ0=E,enable]
COM write P into RQrl
r3bca,[r3b_temp2,rl=0,+0RQ0=E,+0RQ0=P,enable],[r3b_temp3,rl=0,RQ0=P,enable]
r3bcb,[r3b_temp2,rl=1,+0RQ1=E,+0RQ1=P,enable],[r3b_temp3,rl=1,RQ1=P,enable]
r3bcc,[r3b_temp2,rl=2,+0RQ2=E,+0RQ2=P,enable],[r3b_temp3,rl=2,RQ2=P,enable]
COM read Arl
r3bda,[r3b_temp3,rl=0,A0=E,enable],[r3b_temp4,rl=0,A0=E,Arl=E]
r3bdb,[r3b_temp3,rl=0,A0=01,enable],[r3b_temp4,rl=0,A0=01,Arl=01,Arln=E]
r3bdc,[r3b_temp3,rl=0,A0=02,enable],[r3b_temp4,rl=0,A0=02,Arl=02,Arln=E]
r3bdd,[r3b_temp3,rl=0,A0=03,enable],[r3b_temp4,rl=0,A0=03,Arl=03,Arln=E]
r3bde,[r3b_temp3,rl=0,A0=04,enable],[r3b_temp4,rl=0,A0=04,Arl=04,Arln=E]
r3bdf,[r3b_temp3,rl=0,A0=11,enable],[r3b_temp4,rl=0,A0=11,Arl=11,Arln=E]
r3bdg,[r3b_temp3,rl=0,A0=12,enable],[r3b_temp4,rl=0,A0=12,Arl=12,Arln=E]
r3bdh,[r3b_temp3,rl=0,A0=13,enable],[r3b_temp4,rl=0,A0=13,Arl=13,Arln=E]
r3bdi,[r3b_temp3,rl=0,A0=14,enable],[r3b_temp4,rl=0,A0=14,Arl=14,Arln=E]
r3bdj,[r3b_temp3,rl=0,A0=21,enable],[r3b_temp4,rl=0,A0=21,Arl=21,Arln=E]
r3bdk,[r3b_temp3,rl=0,A0=22,enable],[r3b_temp4,rl=0,A0=22,Arl=22,Arln=E]
r3bdl,[r3b_temp3,rl=0,A0=23,enable],[r3b_temp4,rl=0,A0=23,Arl=23,Arln=E]
r3bdm,[r3b_temp3,rl=0,A0=24,enable],[r3b_temp4,rl=0,A0=24,Arl=24,Arln=E]
r3bea,[r3b_temp3,rl=1,A1=E,enable],[r3b_temp4,rl=1,A1=E,Arl=E]
r3beb,[r3b_temp3,rl=1,A1=01,enable],[r3b_temp4,rl=1,A1=01,Arl=01,Arln=E]
r3bec,[r3b_temp3,rl=1,A1=02,enable],[r3b_temp4,rl=1,A1=02,Arl=02,Arln=E]
r3bed,[r3b_temp3,rl=1,A1=03,enable],[r3b_temp4,rl=1,A1=03,Arl=03,Arln=E]
r3bee,[r3b_temp3,rl=1,A1=04,enable],[r3b_temp4,rl=1,A1=04,Arl=04,Arln=E]
r3bef,[r3b_temp3,rl=1,A1=11,enable],[r3b_temp4,rl=1,A1=11,Arl=11,Arln=E]
r3beg,[r3b_temp3,rl=1,A1=12,enable],[r3b_temp4,rl=1,A1=12,Arl=12,Arln=E]
r3beh,[r3b_temp3,rl=1,A1=13,enable],[r3b_temp4,rl=1,A1=13,Arl=13,Arln=E]
r3bei,[r3b_temp3,rl=1,A1=14,enable],[r3b_temp4,rl=1,A1=14,Arl=14,Arln=E]
r3bej,[r3b_temp3,rl=1,A1=21,enable],[r3b_temp4,rl=1,A1=21,Arl=21,Arln=E]
r3bek,[r3b_temp3,rl=1,A1=22,enable],[r3b_temp4,rl=1,A1=22,Arl=22,Arln=E]
r3bel,[r3b_temp3,rl=1,A1=23,enable],[r3b_temp4,rl=1,A1=23,Arl=23,Arln=E]
r3bem,[r3b_temp3,rl=1,A1=24,enable],[r3b_temp4,rl=1,A1=24,Arl=24,Arln=E]
r3bfa,[r3b_temp3,rl=2,A2=E,enable],[r3b_temp4,rl=2,A2=E,Arl=E]
r3bfb,[r3b_temp3,rl=2,A2=01,enable],[r3b_temp4,rl=2,A2=01,Arl=01,Arln=E]
r3bfc,[r3b_temp3,rl=2,A2=02,enable],[r3b_temp4,rl=2,A2=02,Arl=02,Arln=E]
r3bfd,[r3b_temp3,rl=2,A2=03,enable],[r3b_temp4,rl=2,A2=03,Arl=03,Arln=E]
r3bfe,[r3b_temp3,rl=2,A2=04,enable],[r3b_temp4,rl=2,A2=04,Arl=04,Arln=E]
r3bff,[r3b_temp3,rl=2,A2=11,enable],[r3b_temp4,rl=2,A2=11,Arl=11,Arln=E]
r3bfg,[r3b_temp3,rl=2,A2=12,enable],[r3b_temp4,rl=2,A2=12,Arl=12,Arln=E]
r3bfh,[r3b_temp3,rl=2,A2=13,enable],[r3b_temp4,rl=2,A2=13,Arl=13,Arln=E]
r3bfi,[r3b_temp3,rl=2,A2=14,enable],[r3b_temp4,rl=2,A2=14,Arl=14,Arln=E]
r3bfj,[r3b_temp3,rl=2,A2=21,enable],[r3b_temp4,rl=2,A2=21,Arl=21,Arln=E]
r3bfk,[r3b_temp3,rl=2,A2=22,enable],[r3b_temp4,rl=2,A2=22,Arl=22,Arln=E]
r3bfl,[r3b_temp3,rl=2,A2=23,enable],[r3b_temp4,rl=2,A2=23,Arl=23,Arln=E]
r3bfm,[r3b_temp3,rl=2,A2=24,enable],[r3b_temp4,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl n= E then store value of Arl into (rtg,rt) else (rtg,rt:=s) od
r3bga,[r3b_temp4,Arln=E,Arl=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=1]
```

```
r3bgb,[r3b_temp4,Arln=E,Arl=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=2]
r3bgc,[r3b_temp4,Arln=E,Arl=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=3]
r3bgd,[r3b_temp4,Arln=E,Arl=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=4]
r3bge,[r3b_temp4,Arln=E,Arl=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=1]
r3bgf,[r3b_temp4,Arln=E,Arl=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=2]
r3bgg,[r3b_temp4,Arln=E,Arl=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=3]
r3bgh,[r3b_temp4,Arln=E,Arl=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=4]
r3bgi,[r3b_temp4,Arln=E,Arl=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=1]
r3bgj,[r3b_temp4,Arln=E,Arl=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=2]
r3bgk,[r3b_temp4,Arln=E,Arl=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=3]
r3bgl,[r3b_temp4,Arln=E,Arl=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=4]
r3bha,[r3b_temp5,+0s=01,+0s=02,+0s=03,+0s=04,+0s=11,+0s=12,+0s=13,+0s=14,
+0s=21,+0s=22,+0s=23,+0s=24],[r4,enable]
r3bia,[r3b_temp4,Arl=E,s=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=1]
r3bib,[r3b_temp4,Arl=E,s=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=2]
r3bic,[r3b_temp4,Arl=E,s=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=3]
r3bid,[r3b_temp4,Arl=E,s=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=4]
r3bie,[r3b_temp4,Arl=E,s=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=1]
r3bif,[r3b_temp4,Arl=E,s=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=2]
r3big,[r3b_temp4,Arl=E,s=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=3]
r3bih,[r3b_temp4,Arl=E,s=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=4]
r3bii,[r3b_temp4,Arl=E,s=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=1]
r3bij,[r3b_temp4,Arl=E,s=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=2]
r3bik,[r3b_temp4,Arl=E,s=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=3]
r3bil,[r3b_temp4,Arl=E,s=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=4]
r3bja,[r3b_temp6,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=11,+0Arl=12,
+0Arl=13,+0Arl=14,+0Arl=21,+0Arl=22,+0Arl=23,+0Arl=24],[r4,enable]
COM read track rt;
r4a,[r4,rt=1,enable],[rt=1,reading_1,enable]
r4b,[reading_1,enable],[r1,enable]
r4c,[r4,rt=2,enable],[rt=2,reading_2,enable]
r4d,[reading_2,enable],[r1,enable]
r4e,[r4,rt=3,enable],[rt=3,reading_3,enable]
r4f,[reading_3,enable],[r1,enable]
r4g,[r4,rt=4,enable],[rt=4,reading_4,enable]
r4h,[reading_4,enable],[r1,enable]
COM check coherence
coh1,[writing_1,reading_1,enable],[coherence_lost,enable]
coh2,[writing_2,reading_2,enable],[coherence_lost,enable]
coh3,[writing_3,reading_3,enable],[coherence_lost,enable]
coh4,[writing_4,reading_4,enable],[coherence_lost,enable]
END
```

```
Kirousis model 1 writer 1 reader modulo3 arithmatic.

4b+39 registers

Shared vars
RQ0 {P,(E)} - Request?
RQ1 {P,(E)}
RQ2 {P,(E)}
A0 {(E),01,02,03,04,11,12,13,14,21,22,23,24} (rtg,rt/wtg,wt) - Acknowledge?
A1 {(E),01,02,03,04,11,12,13,14,21,22,23,24} (rtg,rt/wtg,wt)
A2 {(E),01,02,03,04,11,12,13,14,21,22,23,24} (rtg,rt/wtg,wt)

Writer local
wl {(0),1,2} - writer layer
vb - v1,v2 {(E),1,2,3,4} - verboden, not allowed track numbers
wt {1,2,3,4} - write track - current
wt- {1,2,3,4} - writer track - last
wtg {(0),1,2} - writer tag

Reader local
rt {1,2,3,4} - reader track
rl {(-1),0,1,2} - reader layer
rtg {(0),1,2} - reader tag
s {(E),01,02,03,04,11,12,13,14,21,22,23,24} - store?
```

B.2.2 Model for testing freshness

```
Kirousis 4 track software model incl. Freshness
bad=[coherence_lost],[freshness_lost]
mask=[w1,w2,w3,w2_temp1,w3_temp1,w3_temp1d,w3_temp2,w3_temp4,w4,w5,w5_temp1,
w5_temp2,A0=E,A0=01,A0=02,A0=03,A0=04,A0=11,A0=12,A0=13,A0=14,A0=21,A0=22,
A0=23,A0=24,A1=E,A1=01,A1=02,A1=03,A1=04,A1=11,A1=12,A1=13,A1=14,A1=21,A1=22,
A1=23,A1=24,A2=E,A2=01,A2=02,A2=03,A2=04,A2=11,A2=12,A2=13,A2=14,A2=21,A2=22,
A2=23,A2=24,wl=0,wl=1,wl=2,v1=1,v1=2,v1=3,v1=4,v2=1,v2=2,v2=3,v2=4,wt-=1,
wt-=2,wt-=3,wt-=4,wt=1,wt=2,wt=3,wt=4,wtg=0,wtg=1,wtg=2,temp1,temp1b,temp2,
temp2b,temp3,temp3b,temp4,temp4b,wRQ=P,wRQ=E,writing_1,writing_2,writing_3,
writing_4]
initial=[enable,w1,r1,RQ0=E,RQ1=E,RQ2=E,A0=E,A1=E,A2=E,wl=0,v1=1,v2=1,wt-=1,
wt=1,wtg=0,rt=1,rl=-1,rtg=0,temp1,temp2,temp3,temp4,no_fresh,no_prev,
not_valid_1,not_valid_2,not_valid_3,not_valid_4]
COM read RQwl
w1a,[w1,wl=0,RQ0=P,enable],[w2,wl=0,RQ0=P,wRQ=P]
w1b,[w1,wl=0,RQ0=E,enable],[w2,wl=0,RQ0=E,wRQ=E]
w1c,[w1,wl=1,RQ1=P,enable],[w2,wl=1,RQ1=P,wRQ=P]
w1d,[w1,wl=1,RQ1=E,enable],[w2,wl=1,RQ1=E,wRQ=E]
w1e,[w1,wl=2,RQ2=P,enable],[w2,wl=2,RQ2=P,wRQ=P]
w1f,[w1,wl=2,RQ2=E,enable],[w2,wl=2,RQ2=E,wRQ=E]
COM if RQwl=E do nothing
w2a,[w2,wRQ=E],[w3]
COM if RQwl=P then wl=wl+1 and
w2b,[w2,wl=0,wRQ=P],[w2_temp1,wl=1]
w2c,[w2,wl=1,wRQ=P],[w2_temp1,wl=2]
w2d,[w2,wl=2,wRQ=P],[w2_temp1,wl=0]
COM vb:={wt,wt-}; (v1=wt) (v2=wt-)
w2fa,[w2_temp1,wt=1,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=1,v1=1,v2=1]
w2fb,[w2_temp1,wt=1,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=2,v1=1,v2=2]
w2fc,[w2_temp1,wt=1,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=3,v1=1,v2=3]
w2fd,[w2_temp1,wt=1,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=4,v1=1,v2=4]
w2ga,[w2_temp1,wt=2,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=1,v1=2,v2=1]
```

```
w2gb,[w2_temp1,wt=2,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=2,v1=2,v2=2]
w2gc,[w2_temp1,wt=2,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=3,v1=2,v2=3]
w2gd,[w2_temp1,wt=2,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=4,v1=2,v2=4]
w2ha,[w2_temp1,wt=3,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=1,v1=3,v2=1]
w2hb,[w2_temp1,wt=3,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=2,v1=3,v2=2]
w2hc,[w2_temp1,wt=3,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=3,v1=3,v2=3]
w2hd,[w2_temp1,wt=3,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=4,v1=3,v2=4]
w2ia,[w2_temp1,wt=4,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=1,v1=4,v2=1]
w2ib,[w2_temp1,wt=4,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=2,v1=4,v2=2]
w2ic,[w2_temp1,wt=4,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=3,v1=4,v2=3]
w2id,[w2_temp1,wt=4,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=4,v1=4,v2=4]
COM wt-:=wt;
w3a,[w3,wt=1,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=1]
w3b,[w3,wt=2,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=2]
w3c,[w3,wt=3,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=3]
w3d,[w3,wt=4,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=4]
COM wt:=min({1,2,3,4}-({wt-} U vb)); (setup {1,2,3,4})
w3aa,[w3_temp1,+0temp1,+0temp1b,+1temp2,+1temp2b,+2temp3,+2temp3b,+3temp4,
+3temp4b],[temp1,temp2,temp3,temp4,w3_temp1d]
COM (remove wt-)
w3ba,[w3_temp1d,wt-=1,temp1],[w3_temp2,wt-=1,temp1b]
w3bb,[w3_temp1d,wt-=2,temp2],[w3_temp2,wt-=2,temp2b]
w3bc,[w3_temp1d,wt-=3,temp3],[w3_temp2,wt-=3,temp3b]
w3bd,[w3_temp1d,wt-=4,temp4],[w3_temp2,wt-=4,temp4b]
COM (remove v1)
w3ca,[w3_temp2,v1=1,+0temp1,+0temp1b],[w3_temp3,v1=1,temp1b]
w3cc,[w3_temp2,v1=2,+0temp2,+0temp2b],[w3_temp3,v1=2,temp2b]
w3ce,[w3_temp2,v1=3,+0temp3,+0temp3b],[w3_temp3,v1=3,temp3b]
w3cg,[w3_temp2,v1=4,+0temp4,+0temp4b],[w3_temp3,v1=4,temp4b]
COM (remove v2)
w3da,[w3_temp3,v2=1,+0temp1,+0temp1b],[w3_temp4,v2=1,temp1b]
w3dc,[w3_temp3,v2=2,+0temp2,+0temp2b],[w3_temp4,v2=2,temp2b]
w3de,[w3_temp3,v2=3,+0temp3,+0temp3b],[w3_temp4,v2=3,temp3b]
w3dg,[w3_temp3,v2=4,+0temp4,+0temp4b],[w3_temp4,v2=4,temp4b]
COM (assign wt)
w3ea,[w3_temp4,temp1],[w4,temp1,wt=1,enable]
w3eb,[w3_temp4,temp1b,temp2],[w4,temp1b,temp2,wt=2,enable]
w3ec,[w3_temp4,temp1b,temp2b,temp3],[w4,temp1b,temp2b,temp3,wt=3,enable]
w3ed,[w3_temp4,temp1b,temp2b,temp3b,temp4],[w4,temp1b,temp2b,temp3b,temp4,
wt=4,enable]
COM write to track wt
w4a,[w4,wt=1,enable],[wt=1,writing_1,enable]
w4b,[writing_1,enable],[w4_written_1]
w4c,[w4,wt=2,enable],[wt=2,writing_2,enable]
w4d,[writing_2,enable],[w4_written_2]
w4e,[w4,wt=3,enable],[wt=3,writing_3,enable]
w4f,[writing_3,enable],[w4_written_3]
w4g,[w4,wt=4,enable],[wt=4,writing_4,enable]
w4h,[writing_4,enable],[w4_written_4]
COM F -> P and W -> FV
w4_fre_a1,[w4_written_1,no_fresh,+0valid_1,+0not_valid_1],[fresh_1,valid_1,
w5,enable]
w4_fre_a2,[w4_written_1,fresh_1,no_prev,+0valid_1,+0not_valid_1],[prev_1,
fresh_1,valid_1,w5,enable]
w4_fre_a3,[w4_written_1,fresh_2,no_prev,+0valid_1,+0not_valid_1],[prev_2,
```

```
fresh_1,valid_1,w5,enable]
w4_fre_a4,[w4_written_1,fresh_3,no_prev,+0valid_1,+0not_valid_1],[prev_3,
fresh_1,valid_1,w5,enable]
w4_fre_a5,[w4_written_1,fresh_4,no_prev,+0valid_1,+0not_valid_1],[prev_4,
fresh_1,valid_1,w5,enable]
w4_fre_a6,[w4_written_2,no_fresh,+0valid_2,+0not_valid_2],[fresh_2,valid_2,
w5,enable]
w4_fre_a7,[w4_written_2,fresh_1,no_prev,+0valid_2,+0not_valid_2],[prev_1,
fresh_2,valid_2,w5,enable]
w4_fre_a8,[w4_written_2,fresh_2,no_prev,+0valid_2,+0not_valid_2],[prev_2,
fresh_2,valid_2,w5,enable]
w4_fre_a9,[w4_written_2,fresh_3,no_prev,+0valid_2,+0not_valid_2],[prev_3,
fresh_2,valid_2,w5,enable]
w4_fre_a10,[w4_written_2,fresh_4,no_prev,+0valid_2,+0not_valid_2],[prev_4,
fresh_2,valid_2,w5,enable]
w4_fre_a11,[w4_written_3,no_fresh,+0valid_3,+0not_valid_3],[fresh_3,valid_3,
w5,enable]
w4_fre_a12,[w4_written_3,fresh_1,no_prev,+0valid_3,+0not_valid_3],[prev_1,
fresh_3,valid_3,w5,enable]
w4_fre_a13,[w4_written_3,fresh_2,no_prev,+0valid_3,+0not_valid_3],[prev_2,
fresh_3,valid_3,w5,enable]
w4_fre_a14,[w4_written_3,fresh_3,no_prev,+0valid_3,+0not_valid_3],[prev_3,
fresh_3,valid_3,w5,enable]
w4_fre_a15,[w4_written_3,fresh_4,no_prev,+0valid_3,+0not_valid_3],[prev_4,
fresh_3,valid_3,w5,enable]
w4_fre_a16,[w4_written_4,no_fresh,+0valid_4,+0not_valid_4],[fresh_4,valid_4,
w5,enable]
w4_fre_a17,[w4_written_4,fresh_1,no_prev,+0valid_4,+0not_valid_4],[prev_1,
fresh_4,valid_4,w5,enable]
w4_fre_a18,[w4_written_4,fresh_2,no_prev,+0valid_4,+0not_valid_4],[prev_2,
fresh_4,valid_4,w5,enable]
w4_fre_a19,[w4_written_4,fresh_3,no_prev,+0valid_4,+0not_valid_4],[prev_3,
fresh_4,valid_4,w5,enable]
w4_fre_a20,[w4_written_4,fresh_4,no_prev,+0valid_4,+0not_valid_4],[prev_4,
fresh_4,valid_4,w5,enable]
COM wtg := wtg + 1;
w5aa,[w5,wtg=0,enable],[w5_temp1,wtg=1]
w5ab,[w5,wtg=1,enable],[w5_temp1,wtg=2]
w5ac,[w5,wtg=2,enable],[w5_temp1,wtg=0]
COM write E on Awl+1;
w5bb,[w5_temp1,wl=0,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24],[w5_temp2,wl=0,A1=E,enable]
w5bd,[w5_temp1,wl=1,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24],[w5_temp2,wl=1,A2=E,enable]
w5bf,[w5_temp1,wl=2,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24],[w5_temp2,wl=2,A0=E,enable]
COM write (wtg,wt) on Awl;
w5ca,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=1,enable],[w5_temp3,
wl=0,wtg=0,wt=1,A0=01]
w5cb,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=2,enable],[w5_temp3,
wl=0,wtg=0,wt=2,A0=02]
w5cc,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=3,enable],[w5_temp3,
wl=0,wtg=0,wt=3,A0=03]
w5cd,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=4,enable],[w5_temp3,
wl=0,wtg=0,wt=4,A0=04]
w5ce,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=1,enable],[w5_temp3,
wl=0,wtg=1,wt=1,A0=11]
w5cf,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=2,enable],[w5_temp3,
wl=0,wtg=1,wt=2,A0=12]
w5cg,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
```

```
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=3,enable],[w5_temp3,
wl=0,wtg=1,wt=3,A0=13]
w5ch,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=4,enable],[w5_temp3,
wl=0,wtg=1,wt=4,A0=14]
w5ci,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=1,enable],[w5_temp3,
wl=0,wtg=2,wt=1,A0=21]
w5cj,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=2,enable],[w5_temp3,
wl=0,wtg=2,wt=2,A0=22]
w5ck,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=3,enable],[w5_temp3,
wl=0,wtg=2,wt=3,A0=23]
w5cl,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=4,enable],[w5_temp3,
wl=0,wtg=2,wt=4,A0=24]
w5da,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=1,enable],[w5_temp3,
wl=1,wtg=0,wt=1,A1=01]
w5db,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=2,enable],[w5_temp3,
wl=1,wtg=0,wt=2,A1=02]
w5dc,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=3,enable],[w5_temp3,
wl=1,wtg=0,wt=3,A1=03]
w5dd,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=4,enable],[w5_temp3,
wl=1,wtg=0,wt=4,A1=04]
w5de,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=1,enable],[w5_temp3,
wl=1,wtg=1,wt=1,A1=11]
w5df,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=2,enable],[w5_temp3,
wl=1,wtg=1,wt=2,A1=12]
w5dg,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=3,enable],[w5_temp3,
wl=1,wtg=1,wt=3,A1=13]
w5dh,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=4,enable],[w5_temp3,
wl=1,wtg=1,wt=4,A1=14]
w5di,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=1,enable],[w5_temp3,
wl=1,wtg=2,wt=1,A1=21]
w5dj,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=2,enable],[w5_temp3,
wl=1,wtg=2,wt=2,A1=22]
w5dk,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=3,enable],[w5_temp3,
wl=1,wtg=2,wt=3,A1=23]
w5dl,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=4,enable],[w5_temp3,
wl=1,wtg=2,wt=4,A1=24]
w5ea,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=1,enable],[w5_temp3,
wl=2,wtg=0,wt=1,A2=01]
w5eb,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=2,enable],[w5_temp3,
wl=2,wtg=0,wt=2,A2=02]
w5ec,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=3,enable],[w5_temp3,
wl=2,wtg=0,wt=3,A2=03]
w5ed,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=4,enable],[w5_temp3,
wl=2,wtg=0,wt=4,A2=04]
w5ee,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
```

```
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=1,enable],[w5_temp3,
wl=2,wtg=1,wt=1,A2=11]
w5ef,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=2,enable],[w5_temp3,
wl=2,wtg=1,wt=2,A2=12]
w5eg,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=3,enable],[w5_temp3,
wl=2,wtg=1,wt=3,A2=13]
w5eh,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=4,enable],[w5_temp3,
wl=2,wtg=1,wt=4,A2=14]
w5ei,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=1,enable],[w5_temp3,
wl=2,wtg=2,wt=1,A2=21]
w5ej,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=2,enable],[w5_temp3,
wl=2,wtg=2,wt=2,A2=22]
w5ek,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=3,enable],[w5_temp3,
wl=2,wtg=2,wt=3,A2=23]
w5el,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=4,enable],[w5_temp3,
wl=2,wtg=2,wt=4,A2=24]
COM clear P
w5_clear_p,[w5_temp3,+0no_prev,+0prev_1,+0prev_2,+0prev_3,+0prev_4],[w1,
enable,no_prev]
COM - The READER -
COM Clear all Valid except F, and P if it exists
r1_pre_a,[r1,enable,+0not_valid_1,+0valid_1,+1not_valid_2,+1valid_2,
+2not_valid_3,+2valid_3,+3not_valid_4,+3valid_4,no_fresh],[r1a,no_fresh,
not_valid_1,not_valid_2,not_valid_3,not_valid_4]
r1_pre_b,[r1,enable,+0not_valid_1,+0valid_1,+1not_valid_2,+1valid_2,
+2not_valid_3,+2valid_3,+3not_valid_4,+3valid_4,fresh_1],[r1a,fresh_1,
valid_1,not_valid_2,not_valid_3,not_valid_4]
r1_pre_c,[r1,enable,+0not_valid_1,+0valid_1,+1not_valid_2,+1valid_2,
+2not_valid_3,+2valid_3,+3not_valid_4,+3valid_4,fresh_2],[r1a,fresh_2,
not_valid_1,valid_2,not_valid_3,not_valid_4]
r1_pre_d,[r1,enable,+0not_valid_1,+0valid_1,+1not_valid_2,+1valid_2,
+2not_valid_3,+2valid_3,+3not_valid_4,+3valid_4,fresh_3],[r1a,fresh_3,
not_valid_1,not_valid_2,valid_3,not_valid_4]
r1_pre_e,[r1,enable,+0not_valid_1,+0valid_1,+1not_valid_2,+1valid_2,
+2not_valid_3,+2valid_3,+3not_valid_4,+3valid_4,fresh_4],[r1a,fresh_4,
not_valid_1,not_valid_2,not_valid_3,valid_4]
r1_pre_f,[r1a,no_prev],[r1b,no_prev]
r1_pre_g,[r1a,prev_1,not_valid_1],[r1b,prev_1,valid_1]
r1_pre_h,[r1a,prev_2,not_valid_2],[r1b,prev_2,valid_2]
r1_pre_i,[r1a,prev_3,not_valid_3],[r1b,prev_3,valid_3]
r1_pre_j,[r1a,prev_4,not_valid_4],[r1b,prev_4,valid_4]
COM rl := rl+1;
r1a,[r1b,rl=-1],[r2,rl=0]
r1b,[r1b,rl=0],[r2,rl=1]
r1c,[r1b,rl=1],[r2,rl=2]
r1d,[r1b,rl=2],[r2,rl=0]
COM read Arl;
r2aa,[r2,rl=0,A0=E],[r3,rl=0,A0=E,Arl=E]
r2ab,[r2,rl=0,A0=01],[r3,rl=0,A0=01,Arl=01,Arln=E]
r2ac,[r2,rl=0,A0=02],[r3,rl=0,A0=02,Arl=02,Arln=E]
r2ad,[r2,rl=0,A0=03],[r3,rl=0,A0=03,Arl=03,Arln=E]
r2ae,[r2,rl=0,A0=04],[r3,rl=0,A0=04,Arl=04,Arln=E]
r2af,[r2,rl=0,A0=11],[r3,rl=0,A0=11,Arl=11,Arln=E]
r2ag,[r2,rl=0,A0=12],[r3,rl=0,A0=12,Arl=12,Arln=E]
r2ah,[r2,rl=0,A0=13],[r3,rl=0,A0=13,Arl=13,Arln=E]
r2ai,[r2,rl=0,A0=14],[r3,rl=0,A0=14,Arl=14,Arln=E]
r2aj,[r2,rl=0,A0=21],[r3,rl=0,A0=21,Arl=21,Arln=E]
r2ak,[r2,rl=0,A0=22],[r3,rl=0,A0=22,Arl=22,Arln=E]
r2al,[r2,rl=0,A0=23],[r3,rl=0,A0=23,Arl=23,Arln=E]
```

```
r2am,[r2,rl=0,A0=24],[r3,rl=0,A0=24,Arl=24,Arln=E]
r2ba,[r2,rl=1,A1=E],[r3,rl=1,A1=E,Arl=E]
r2bb,[r2,rl=1,A1=01],[r3,rl=1,A1=01,Arl=01,Arln=E]
r2bc,[r2,rl=1,A1=02],[r3,rl=1,A1=02,Arl=02,Arln=E]
r2bd,[r2,rl=1,A1=03],[r3,rl=1,A1=03,Arl=03,Arln=E]
r2be,[r2,rl=1,A1=04],[r3,rl=1,A1=04,Arl=04,Arln=E]
r2bf,[r2,rl=1,A1=11],[r3,rl=1,A1=11,Arl=11,Arln=E]
r2bg,[r2,rl=1,A1=12],[r3,rl=1,A1=12,Arl=12,Arln=E]
r2bh,[r2,rl=1,A1=13],[r3,rl=1,A1=13,Arl=13,Arln=E]
r2bi,[r2,rl=1,A1=14],[r3,rl=1,A1=14,Arl=14,Arln=E]
r2bj,[r2,rl=1,A1=21],[r3,rl=1,A1=21,Arl=21,Arln=E]
r2bk,[r2,rl=1,A1=22],[r3,rl=1,A1=22,Arl=22,Arln=E]
r2bl,[r2,rl=1,A1=23],[r3,rl=1,A1=23,Arl=23,Arln=E]
r2bm,[r2,rl=1,A1=24],[r3,rl=1,A1=24,Arl=24,Arln=E]
r2ca,[r2,rl=2,A2=E],[r3,rl=2,A2=E,Arl=E]
r2cb,[r2,rl=2,A2=01],[r3,rl=2,A2=01,Arl=01,Arln=E]
r2cc,[r2,rl=2,A2=02],[r3,rl=2,A2=02,Arl=02,Arln=E]
r2cd,[r2,rl=2,A2=03],[r3,rl=2,A2=03,Arl=03,Arln=E]
r2ce,[r2,rl=2,A2=04],[r3,rl=2,A2=04,Arl=04,Arln=E]
r2cf,[r2,rl=2,A2=11],[r3,rl=2,A2=11,Arl=11,Arln=E]
r2cg,[r2,rl=2,A2=12],[r3,rl=2,A2=12,Arl=12,Arln=E]
r2ch,[r2,rl=2,A2=13],[r3,rl=2,A2=13,Arl=13,Arln=E]
r2ci,[r2,rl=2,A2=14],[r3,rl=2,A2=14,Arl=14,Arln=E]
r2cj,[r2,rl=2,A2=21],[r3,rl=2,A2=21,Arl=21,Arln=E]
r2ck,[r2,rl=2,A2=22],[r3,rl=2,A2=22,Arl=22,Arln=E]
r2cl,[r2,rl=2,A2=23],[r3,rl=2,A2=23,Arl=23,Arln=E]
r2cm,[r2,rl=2,A2=24],[r3,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl=E then do rl=rl-1; reread Arl;
r3aa,[r3,Arl=E,rl=0],[r3a_temp1,rl=2,enable]
r3ab,[r3,Arl=E,rl=1],[r3a_temp1,rl=0,enable]
r3ac,[r3,Arl=E,rl=2],[r3a_temp1,rl=1,enable]
COM (reread Arl)
r3aaa,[r3a_temp1,rl=0,A0=E,enable],[r3a_temp2,rl=0,A0=E,Arl=E]
r3aab,[r3a_temp1,rl=0,A0=01,enable],[r3a_temp2,rl=0,A0=01,Arl=01,Arln=E]
r3aac,[r3a_temp1,rl=0,A0=02,enable],[r3a_temp2,rl=0,A0=02,Arl=02,Arln=E]
r3aad,[r3a_temp1,rl=0,A0=03,enable],[r3a_temp2,rl=0,A0=03,Arl=03,Arln=E]
r3aae,[r3a_temp1,rl=0,A0=04,enable],[r3a_temp2,rl=0,A0=04,Arl=04,Arln=E]
r3aaf,[r3a_temp1,rl=0,A0=11,enable],[r3a_temp2,rl=0,A0=11,Arl=11,Arln=E]
r3aag,[r3a_temp1,rl=0,A0=12,enable],[r3a_temp2,rl=0,A0=12,Arl=12,Arln=E]
r3aah,[r3a_temp1,rl=0,A0=13,enable],[r3a_temp2,rl=0,A0=13,Arl=13,Arln=E]
r3aai,[r3a_temp1,rl=0,A0=14,enable],[r3a_temp2,rl=0,A0=14,Arl=14,Arln=E]
r3aaj,[r3a_temp1,rl=0,A0=21,enable],[r3a_temp2,rl=0,A0=21,Arl=21,Arln=E]
r3aak,[r3a_temp1,rl=0,A0=22,enable],[r3a_temp2,rl=0,A0=22,Arl=22,Arln=E]
r3aal,[r3a_temp1,rl=0,A0=23,enable],[r3a_temp2,rl=0,A0=23,Arl=23,Arln=E]
r3aam,[r3a_temp1,rl=0,A0=24,enable],[r3a_temp2,rl=0,A0=24,Arl=24,Arln=E]
r3aba,[r3a_temp1,rl=1,A1=E,enable],[r3a_temp2,rl=1,A1=E,Arl=E]
r3abb,[r3a_temp1,rl=1,A1=01,enable],[r3a_temp2,rl=1,A1=01,Arl=01,Arln=E]
r3abc,[r3a_temp1,rl=1,A1=02,enable],[r3a_temp2,rl=1,A1=02,Arl=02,Arln=E]
r3abd,[r3a_temp1,rl=1,A1=03,enable],[r3a_temp2,rl=1,A1=03,Arl=03,Arln=E]
r3abe,[r3a_temp1,rl=1,A1=04,enable],[r3a_temp2,rl=1,A1=04,Arl=04,Arln=E]
r3abf,[r3a_temp1,rl=1,A1=11,enable],[r3a_temp2,rl=1,A1=11,Arl=11,Arln=E]
r3abg,[r3a_temp1,rl=1,A1=12,enable],[r3a_temp2,rl=1,A1=12,Arl=12,Arln=E]
r3abh,[r3a_temp1,rl=1,A1=13,enable],[r3a_temp2,rl=1,A1=13,Arl=13,Arln=E]
r3abi,[r3a_temp1,rl=1,A1=14,enable],[r3a_temp2,rl=1,A1=14,Arl=14,Arln=E]
r3abj,[r3a_temp1,rl=1,A1=21,enable],[r3a_temp2,rl=1,A1=21,Arl=21,Arln=E]
r3abk,[r3a_temp1,rl=1,A1=22,enable],[r3a_temp2,rl=1,A1=22,Arl=22,Arln=E]
r3abl,[r3a_temp1,rl=1,A1=23,enable],[r3a_temp2,rl=1,A1=23,Arl=23,Arln=E]
r3abm,[r3a_temp1,rl=1,A1=24,enable],[r3a_temp2,rl=1,A1=24,Arl=24,Arln=E]
r3aca,[r3a_temp1,rl=2,A2=E,enable],[r3a_temp2,rl=2,A2=E,Arl=E]
r3acb,[r3a_temp1,rl=2,A2=01,enable],[r3a_temp2,rl=2,A2=01,Arl=01,Arln=E]
r3acc,[r3a_temp1,rl=2,A2=02,enable],[r3a_temp2,rl=2,A2=02,Arl=02,Arln=E]
r3acd,[r3a_temp1,rl=2,A2=03,enable],[r3a_temp2,rl=2,A2=03,Arl=03,Arln=E]
r3ace,[r3a_temp1,rl=2,A2=04,enable],[r3a_temp2,rl=2,A2=04,Arl=04,Arln=E]
r3acf,[r3a_temp1,rl=2,A2=11,enable],[r3a_temp2,rl=2,A2=11,Arl=11,Arln=E]
r3acg,[r3a_temp1,rl=2,A2=12,enable],[r3a_temp2,rl=2,A2=12,Arl=12,Arln=E]
r3ach,[r3a_temp1,rl=2,A2=13,enable],[r3a_temp2,rl=2,A2=13,Arl=13,Arln=E]
```

```
r3aci,[r3a_temp1,rl=2,A2=14,enable],[r3a_temp2,rl=2,A2=14,Arl=14,Arln=E]
r3acj,[r3a_temp1,rl=2,A2=21,enable],[r3a_temp2,rl=2,A2=21,Arl=21,Arln=E]
r3ack,[r3a_temp1,rl=2,A2=22,enable],[r3a_temp2,rl=2,A2=22,Arl=22,Arln=E]
r3acl,[r3a_temp1,rl=2,A2=23,enable],[r3a_temp2,rl=2,A2=23,Arl=23,Arln=E]
r3acm,[r3a_temp1,rl=2,A2=24,enable],[r3a_temp2,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl n= E and tag of Arl=rtg+1 then store value of Arl into (rtg,rt) od
COM tag = 1st digit of Arl
r3ada,[r3a_temp2,Arln=E,rtg=2,Arl=01,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=1,enable]
r3adb,[r3a_temp2,Arln=E,rtg=2,Arl=02,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=2,enable]
r3adc,[r3a_temp2,Arln=E,rtg=2,Arl=03,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=3,enable]
r3add,[r3a_temp2,Arln=E,rtg=2,Arl=04,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=0,
rt=4,enable]
r3ade,[r3a_temp2,Arln=E,rtg=0,Arl=11,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=1,enable]
r3adf,[r3a_temp2,Arln=E,rtg=0,Arl=12,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=2,enable]
r3adg,[r3a_temp2,Arln=E,rtg=0,Arl=13,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=3,enable]
r3adh,[r3a_temp2,Arln=E,rtg=0,Arl=14,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=1,
rt=4,enable]
r3adi,[r3a_temp2,Arln=E,rtg=1,Arl=21,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=1,enable]
r3adj,[r3a_temp2,Arln=E,rtg=1,Arl=22,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=2,enable]
r3adk,[r3a_temp2,Arln=E,rtg=1,Arl=23,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=3,enable]
r3adl,[r3a_temp2,Arln=E,rtg=1,Arl=24,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[r4,rtg=2,
rt=4,enable]
r3ads,[r3a_temp2,Arl=E],[r4,enable]
r3adt,[r3a_temp2,Arln=E,rtg=2,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=21,
+0Arl=22,+0Arl=23,+0Arl=24],[r4,rtg=2,enable]
r3adu,[r3a_temp2,Arln=E,rtg=0,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=21,
+0Arl=22,+0Arl=23,+0Arl=24],[r4,rtg=0,enable]
r3adv,[r3a_temp2,Arln=E,rtg=1,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=01,
+0Arl=02,+0Arl=03,+0Arl=04],[r4,rtg=1,enable]
COM else do - store value of Arl into s;
r3baa,[r3,Arln=E,Arl=01],[r3b_temp1,s=01,enable]
r3bab,[r3,Arln=E,Arl=02],[r3b_temp1,s=02,enable]
r3bac,[r3,Arln=E,Arl=03],[r3b_temp1,s=03,enable]
r3bad,[r3,Arln=E,Arl=04],[r3b_temp1,s=04,enable]
r3bae,[r3,Arln=E,Arl=11],[r3b_temp1,s=11,enable]
r3baf,[r3,Arln=E,Arl=12],[r3b_temp1,s=12,enable]
r3bag,[r3,Arln=E,Arl=13],[r3b_temp1,s=13,enable]
r3bah,[r3,Arln=E,Arl=14],[r3b_temp1,s=14,enable]
r3bai,[r3,Arln=E,Arl=21],[r3b_temp1,s=21,enable]
r3baj,[r3,Arln=E,Arl=22],[r3b_temp1,s=22,enable]
r3bak,[r3,Arln=E,Arl=23],[r3b_temp1,s=23,enable]
r3bal,[r3,Arln=E,Arl=24],[r3b_temp1,s=24,enable]
COM write E ontp RQrl+1
r3bba,[r3b_temp1,rl=0,+0RQ1=E,+0RQ1=P,enable],[r3b_temp2,rl=0,RQ1=E,enable]
r3bbb,[r3b_temp1,rl=1,+0RQ2=E,+0RQ2=P,enable],[r3b_temp2,rl=1,RQ2=E,enable]
r3bbc,[r3b_temp1,rl=2,+0RQ0=E,+0RQ0=P,enable],[r3b_temp2,rl=2,RQ0=E,enable]
COM write P into RQrl
r3bca,[r3b_temp2,rl=0,+0RQ0=E,+0RQ0=P,enable],[r3b_temp3,rl=0,RQ0=P,enable]
r3bcb,[r3b_temp2,rl=1,+0RQ1=E,+0RQ1=P,enable],[r3b_temp3,rl=1,RQ1=P,enable]
r3bcc,[r3b_temp2,rl=2,+0RQ2=E,+0RQ2=P,enable],[r3b_temp3,rl=2,RQ2=P,enable]
COM read Arl
r3bda,[r3b_temp3,rl=0,A0=E,enable],[r3b_temp4,rl=0,A0=E,Arl=E]
r3bdb,[r3b_temp3,rl=0,A0=01,enable],[r3b_temp4,rl=0,A0=01,Arl=01,Arln=E]
r3bdc,[r3b_temp3,rl=0,A0=02,enable],[r3b_temp4,rl=0,A0=02,Arl=02,Arln=E]
r3bdd,[r3b_temp3,rl=0,A0=03,enable],[r3b_temp4,rl=0,A0=03,Arl=03,Arln=E]
r3bde,[r3b_temp3,rl=0,A0=04,enable],[r3b_temp4,rl=0,A0=04,Arl=04,Arln=E]
r3bdf,[r3b_temp3,rl=0,A0=11,enable],[r3b_temp4,rl=0,A0=11,Arl=11,Arln=E]
```

```
r3bdg,[r3b_temp3,rl=0,A0=12,enable],[r3b_temp4,rl=0,A0=12,Arl=12,Arln=E]
r3bdh,[r3b_temp3,rl=0,A0=13,enable],[r3b_temp4,rl=0,A0=13,Arl=13,Arln=E]
r3bdi,[r3b_temp3,rl=0,A0=14,enable],[r3b_temp4,rl=0,A0=14,Arl=14,Arln=E]
r3bdj,[r3b_temp3,rl=0,A0=21,enable],[r3b_temp4,rl=0,A0=21,Arl=21,Arln=E]
r3bdk,[r3b_temp3,rl=0,A0=22,enable],[r3b_temp4,rl=0,A0=22,Arl=22,Arln=E]
r3bdl,[r3b_temp3,rl=0,A0=23,enable],[r3b_temp4,rl=0,A0=23,Arl=23,Arln=E]
r3bdm,[r3b_temp3,rl=0,A0=24,enable],[r3b_temp4,rl=0,A0=24,Arl=24,Arln=E]
r3bea,[r3b_temp3,rl=1,A1=E,enable],[r3b_temp4,rl=1,A1=E,Arl=E]
r3beb,[r3b_temp3,rl=1,A1=01,enable],[r3b_temp4,rl=1,A1=01,Arl=01,Arln=E]
r3bec,[r3b_temp3,rl=1,A1=02,enable],[r3b_temp4,rl=1,A1=02,Arl=02,Arln=E]
r3bed,[r3b_temp3,rl=1,A1=03,enable],[r3b_temp4,rl=1,A1=03,Arl=03,Arln=E]
r3bee,[r3b_temp3,rl=1,A1=04,enable],[r3b_temp4,rl=1,A1=04,Arl=04,Arln=E]
r3bef,[r3b_temp3,rl=1,A1=11,enable],[r3b_temp4,rl=1,A1=11,Arl=11,Arln=E]
r3beg,[r3b_temp3,rl=1,A1=12,enable],[r3b_temp4,rl=1,A1=12,Arl=12,Arln=E]
r3beh,[r3b_temp3,rl=1,A1=13,enable],[r3b_temp4,rl=1,A1=13,Arl=13,Arln=E]
r3bei,[r3b_temp3,rl=1,A1=14,enable],[r3b_temp4,rl=1,A1=14,Arl=14,Arln=E]
r3bej,[r3b_temp3,rl=1,A1=21,enable],[r3b_temp4,rl=1,A1=21,Arl=21,Arln=E]
r3bek,[r3b_temp3,rl=1,A1=22,enable],[r3b_temp4,rl=1,A1=22,Arl=22,Arln=E]
r3bel,[r3b_temp3,rl=1,A1=23,enable],[r3b_temp4,rl=1,A1=23,Arl=23,Arln=E]
r3bem,[r3b_temp3,rl=1,A1=24,enable],[r3b_temp4,rl=1,A1=24,Arl=24,Arln=E]
r3bfa,[r3b_temp3,rl=2,A2=E,enable],[r3b_temp4,rl=2,A2=E,Arl=E]
r3bfb,[r3b_temp3,rl=2,A2=01,enable],[r3b_temp4,rl=2,A2=01,Arl=01,Arln=E]
r3bfc,[r3b_temp3,rl=2,A2=02,enable],[r3b_temp4,rl=2,A2=02,Arl=02,Arln=E]
r3bfd,[r3b_temp3,rl=2,A2=03,enable],[r3b_temp4,rl=2,A2=03,Arl=03,Arln=E]
r3bfe,[r3b_temp3,rl=2,A2=04,enable],[r3b_temp4,rl=2,A2=04,Arl=04,Arln=E]
r3bff,[r3b_temp3,rl=2,A2=11,enable],[r3b_temp4,rl=2,A2=11,Arl=11,Arln=E]
r3bfg,[r3b_temp3,rl=2,A2=12,enable],[r3b_temp4,rl=2,A2=12,Arl=12,Arln=E]
r3bfh,[r3b_temp3,rl=2,A2=13,enable],[r3b_temp4,rl=2,A2=13,Arl=13,Arln=E]
r3bfi,[r3b_temp3,rl=2,A2=14,enable],[r3b_temp4,rl=2,A2=14,Arl=14,Arln=E]
r3bfj,[r3b_temp3,rl=2,A2=21,enable],[r3b_temp4,rl=2,A2=21,Arl=21,Arln=E]
r3bfk,[r3b_temp3,rl=2,A2=22,enable],[r3b_temp4,rl=2,A2=22,Arl=22,Arln=E]
r3bfl,[r3b_temp3,rl=2,A2=23,enable],[r3b_temp4,rl=2,A2=23,Arl=23,Arln=E]
r3bfm,[r3b_temp3,rl=2,A2=24,enable],[r3b_temp4,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl n= E then store value of Arl into (rtg,rt) else (rtg,rt:=s) od
r3bga,[r3b_temp4,Arln=E,Arl=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=1]
r3bgb,[r3b_temp4,Arln=E,Arl=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=2]
r3bgc,[r3b_temp4,Arln=E,Arl=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=3]
r3bgd,[r3b_temp4,Arln=E,Arl=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=4]
r3bge,[r3b_temp4,Arln=E,Arl=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=1]
r3bgf,[r3b_temp4,Arln=E,Arl=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=2]
r3bgg,[r3b_temp4,Arln=E,Arl=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=3]
r3bgh,[r3b_temp4,Arln=E,Arl=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=4]
r3bgi,[r3b_temp4,Arln=E,Arl=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=1]
r3bgj,[r3b_temp4,Arln=E,Arl=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=2]
r3bgk,[r3b_temp4,Arln=E,Arl=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=3]
r3bgl,[r3b_temp4,Arln=E,Arl=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=4]
r3bha,[r3b_temp5,+0s=01,+0s=02,+0s=03,+0s=04,+0s=11,+0s=12,+0s=13,+0s=14,
+0s=21,+0s=22,+0s=23,+0s=24],[r4,enable]
r3bia,[r3b_temp4,Arl=E,s=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=1]
r3bib,[r3b_temp4,Arl=E,s=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=2]
r3bic,[r3b_temp4,Arl=E,s=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=3]
```

```
r3bid,[r3b_temp4,Arl=E,s=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=4]
r3bie,[r3b_temp4,Arl=E,s=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=1]
r3bif,[r3b_temp4,Arl=E,s=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=2]
r3big,[r3b_temp4,Arl=E,s=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=3]
r3bih,[r3b_temp4,Arl=E,s=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=4]
r3bii,[r3b_temp4,Arl=E,s=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=1]
r3bij,[r3b_temp4,Arl=E,s=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=2]
r3bik,[r3b_temp4,Arl=E,s=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=3]
r3bil,[r3b_temp4,Arl=E,s=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=4]
r3bja,[r3b_temp6,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=11,+0Arl=12,
+0Arl=13,+0Arl=14,+0Arl=21,+0Arl=22,+0Arl=23,+0Arl=24],[r4,enable]
COM if Read is F then cancel P if it exists
r4_pre_a,[r4,enable,rt=1,fresh_1,+0no_prev,+0prev_1,+0prev_2,+0prev_3,
+0prev_4],[r4a,rt=1,fresh_1,no_prev]
r4_pre_b,[r4,enable,rt=2,fresh_2,+0no_prev,+0prev_1,+0prev_2,+0prev_3,
+0prev_4],[r4a,rt=2,fresh_2,no_prev]
r4_pre_c,[r4,enable,rt=3,fresh_3,+0no_prev,+0prev_1,+0prev_2,+0prev_3,
+0prev_4],[r4a,rt=3,fresh_3,no_prev]
r4_pre_d,[r4,enable,rt=4,fresh_4,+0no_prev,+0prev_1,+0prev_2,+0prev_3,
+0prev_4],[r4a,rt=4,fresh_4,no_prev]
r4_pre_e,[r4,enable,rt=1,-fresh_1],[r4a,rt=1]
r4_pre_f,[r4,enable,rt=2,-fresh_2],[r4a,rt=2]
r4_pre_g,[r4,enable,rt=3,-fresh_3],[r4a,rt=3]
r4_pre_h,[r4,enable,rt=4,-fresh_4],[r4a,rt=4]
COM read track rt;
r4a,[r4a,rt=1],[rt=1,reading_1,enable]
r4b,[reading_1,enable],[r1,enable]
r4c,[r4a,rt=2],[rt=2,reading_2,enable]
r4d,[reading_2,enable],[r1,enable]
r4e,[r4a,rt=3],[rt=3,reading_3,enable]
r4f,[reading_3,enable],[r1,enable]
r4g,[r4a,rt=4],[rt=4,reading_4,enable]
r4h,[reading_4,enable],[r1,enable]
COM check coherence
coh1,[writing_1,reading_1,enable],[coherence_lost,enable]
coh2,[writing_2,reading_2,enable],[coherence_lost,enable]
coh3,[writing_3,reading_3,enable],[coherence_lost,enable]
coh4,[writing_4,reading_4,enable],[coherence_lost,enable]
COM check freshness (-no_fresh is included to stop false result ie before
no writes)
fre1,[reading_1,not_valid_1,-no_fresh,enable],[freshness_lost,enable]
fre2,[reading_2,not_valid_2,-no_fresh,enable],[freshness_lost,enable]
fre3,[reading_3,not_valid_3,-no_fresh,enable],[freshness_lost,enable]
fre4,[reading_4,not_valid_4,-no_fresh,enable],[freshness_lost,enable]
END
```

B.2.3 Model for testing sequencing

```
Kirousis 4 track software model, data sequencing.
bad=[order_lost]
mask=[w1,w2,w3,w2_temp1,w3_temp1,w3_temp1d,w3_temp2,w3_temp4,w4,w5,w5_temp1,
w5_temp2,A0=E,A0=01,A0=02,A0=03,A0=04,A0=11,A0=12,A0=13,A0=14,A0=21,A0=22,
A0=23,A0=24,A1=E,A1=01,A1=02,A1=03,A1=04,A1=11,A1=12,A1=13,A1=14,A1=21,
A1=22,A1=23,A1=24,A2=E,A2=01,A2=02,A2=03,A2=04,A2=11,A2=12,A2=13,A2=14,
A2=21,A2=22,A2=23,A2=24,wl=0,wl=1,wl=2,v1=1,v1=2,v1=3,v1=4,v2=1,v2=2,v2=3,
```

```
v2=4,wt-=1,wt-=2,wt-=3,wt-=4,wt=1,wt=2,wt=3,wt=4,wtg=0,wtg=1,wtg=2,temp1,
temp1b,temp2,temp2b,temp3,temp3b,temp4,temp4b,wRQ=P,wRQ=E,writing_1,
writing_2,writing_3,writing_4]
initial=[enable,w1,clear_x,RQ0=E,RQ1=E,RQ2=E,A0=E,A1=E,A2=E,wl=0,v1=1,v2=1,
wt-=1,wt=1,wtg=0,rt=1,rl=-1,rtg=0,temp1,temp2,temp3,temp4,Xb_00,Xb_01,Xb_10,
Xb_11,x_00,x_01,x_10,x_11,4_00,4_01,4_10,4_11,not1,not2,not3]
COM read RQwl
w1a,[w1,wl=0,RQ0=P,enable],[w2,wl=0,RQ0=P,wRQ=P]
w1b,[w1,wl=0,RQ0=E,enable],[w2,wl=0,RQ0=E,wRQ=E]
w1c,[w1,wl=1,RQ1=P,enable],[w2,wl=1,RQ1=P,wRQ=P]
w1d,[w1,wl=1,RQ1=E,enable],[w2,wl=1,RQ1=E,wRQ=E]
w1e,[w1,wl=2,RQ2=P,enable],[w2,wl=2,RQ2=P,wRQ=P]
w1f,[w1,wl=2,RQ2=E,enable],[w2,wl=2,RQ2=E,wRQ=E]
COM if RQwl=E do nothing
w2a,[w2,wRQ=E],[w3]
COM if RQwl=P then wl=wl+1 and
w2b,[w2,wl=0,wRQ=P],[w2_temp1,wl=1]
w2c,[w2,wl=1,wRQ=P],[w2_temp1,wl=2]
w2d,[w2,wl=2,wRQ=P],[w2_temp1,wl=0]
COM vb:={wt,wt-}; (v1=wt) (v2=wt-)
w2fa,[w2_temp1,wt=1,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=1,v1=1,v2=1]
w2fb,[w2_temp1,wt=1,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=2,v1=1,v2=2]
w2fc,[w2_temp1,wt=1,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=3,v1=1,v2=3]
w2fd,[w2_temp1,wt=1,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=1,wt-=4,v1=1,v2=4]
w2ga,[w2_temp1,wt=2,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=1,v1=2,v2=1]
w2gb,[w2_temp1,wt=2,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=2,v1=2,v2=2]
w2gc,[w2_temp1,wt=2,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=3,v1=2,v2=3]
w2gd,[w2_temp1,wt=2,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=2,wt-=4,v1=2,v2=4]
w2ha,[w2_temp1,wt=3,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=1,v1=3,v2=1]
w2hb,[w2_temp1,wt=3,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=2,v1=3,v2=2]
w2hc,[w2_temp1,wt=3,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=3,v1=3,v2=3]
w2hd,[w2_temp1,wt=3,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=3,wt-=4,v1=3,v2=4]
w2ia,[w2_temp1,wt=4,wt-=1,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=1,v1=4,v2=1]
w2ib,[w2_temp1,wt=4,wt-=2,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=2,v1=4,v2=2]
w2ic,[w2_temp1,wt=4,wt-=3,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=3,v1=4,v2=3]
w2id,[w2_temp1,wt=4,wt-=4,+0v1=1,+0v1=2,+0v1=3,+0v1=4,+1v2=1,+1v2=2,+1v2=3,
+1v2=4],[w3,wt=4,wt-=4,v1=4,v2=4]
COM wt-:=wt;
w3a,[w3,wt=1,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=1]
w3b,[w3,wt=2,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=2]
w3c,[w3,wt=3,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=3]
w3d,[w3,wt=4,+0wt-=1,+0wt-=2,+0wt-=3,+0wt-=4],[w3_temp1,wt-=4]
COM wt:=min({1,2,3,4}-({wt-} U vb)); (setup {1,2,3,4})
w3aa,[w3_temp1,+0temp1,+0temp1b,+1temp2,+1temp2b,+2temp3,+2temp3b,+3temp4,
+3temp4b],[temp1,temp2,temp3,temp4,w3_temp1d]
COM (remove wt-)
w3ba,[w3_temp1d,wt-=1,temp1],[w3_temp2,wt-=1,temp1b]
w3bb,[w3_temp1d,wt-=2,temp2],[w3_temp2,wt-=2,temp2b]
w3bc,[w3_temp1d,wt-=3,temp3],[w3_temp2,wt-=3,temp3b]
w3bd,[w3_temp1d,wt-=4,temp4],[w3_temp2,wt-=4,temp4b]
COM (remove v1)
```

```
w3ca,[w3_temp2,v1=1,+0temp1,+0temp1b],[w3_temp3,v1=1,temp1b]
w3cc,[w3_temp2,v1=2,+0temp2,+0temp2b],[w3_temp3,v1=2,temp2b]
w3ce,[w3_temp2,v1=3,+0temp3,+0temp3b],[w3_temp3,v1=3,temp3b]
w3cg,[w3_temp2,v1=4,+0temp4,+0temp4b],[w3_temp3,v1=4,temp4b]
COM (remove v2)
w3da,[w3_temp3,v2=1,+0temp1,+0temp1b],[w3_temp4,v2=1,temp1b]
w3dc,[w3_temp3,v2=2,+0temp2,+0temp2b],[w3_temp4,v2=2,temp2b]
w3de,[w3_temp3,v2=3,+0temp3,+0temp3b],[w3_temp4,v2=3,temp3b]
w3dg,[w3_temp3,v2=4,+0temp4,+0temp4b],[w3_temp4,v2=4,temp4b]
COM (assign wt)
w3ea,[w3_temp4,temp1],[w4,temp1,wt=1,enable]
w3eb,[w3_temp4,temp1b,temp2],[w4,temp1b,temp2,wt=2,enable]
w3ec,[w3_temp4,temp1b,temp2b,temp3],[w4,temp1b,temp2b,temp3,wt=3,enable]
w3ed,[w3_temp4,temp1b,temp2b,temp3b,temp4],[w4,temp1b,temp2b,temp3b,temp4,
wt=4,enable]
COM write to track wt
w4a,[w4,wt=1,enable],[wt=1,writing_1,enable]
w4b,[writing_1,enable],[wip_roles00]
w4c,[w4,wt=2,enable],[wt=2,writing_2,enable]
w4d,[writing_2,enable],[wip_roles01]
w4e,[w4,wt=3,enable],[wt=3,writing_3,enable]
w4f,[writing_3,enable],[wip_roles10]
w4g,[w4,wt=4,enable],[wt=4,writing_4,enable]
w4h,[writing_4,enable],[wip_roles11]
COM wip roles for sequencing
wip1a,[wip_roles00,X_00],[wip_role200,Xb_00]
wip1b,[wip_roles00,Xb_00],[wip_role200,Xb_00]
wip1c,[wip_roles01,X_01],[wip_role201,Xb_01]
wip1d,[wip_roles01,Xb_01],[wip_role201,Xb_01]
wip1e,[wip_roles10,X_10],[wip_role210,Xb_10]
wip1f,[wip_roles10,Xb_10],[wip_role210,Xb_10]
wip1g,[wip_roles11,X_11],[wip_role211,Xb_11]
wip1h,[wip_roles11,Xb_11],[wip_role211,Xb_11]
wip2a,[wip_role200,1_00],[1_00,w5,enable]
wip2b,[wip_role201,1_01],[1_01,w5,enable]
wip2c,[wip_role210,1_10],[1_10,w5,enable]
wip2d,[wip_role211,1_11],[1_11,w5,enable]
wip2e,[wip_role200,2_00],[w2b1_00,11b2,2_00]
wip2f,[wip_role201,2_01],[w2b1_01,11b2,2_01]
wip2g,[wip_role210,2_10],[w2b1_10,11b2,2_10]
wip2h,[wip_role211,2_11],[w2b1_11,11b2,2_11]
wip2i,[wip_role200,3_00],[w3b1_00,22b3,3_00]
wip2j,[wip_role201,3_01],[w3b1_01,22b3,3_01]
wip2k,[wip_role210,3_10],[w3b1_10,22b3,3_10]
wip2l,[wip_role211,3_11],[w3b1_11,22b3,3_11]
wip2m,[wip_role200,4_00],[w4b1_00,33b4,4_00]
wip2n,[wip_role201,4_01],[w4b1_01,33b4,4_01]
wip2o,[wip_role210,4_10],[w4b1_10,33b4,4_10]
wip2p,[wip_role211,4_11],[w4b1_11,33b4,4_11]
wip3a,[33b4,3_00],[22b3,4_00]
wip3b,[33b4,3_01],[22b3,4_01]
wip3c,[33b4,3_10],[22b3,4_10]
wip3d,[33b4,3_11],[22b3,4_11]
wip3e,[33b4,not3],[22b3,not3]
wip4a,[22b3,2_00,not3],[11b2,3_00,not3b]
wip4aa,[22b3,2_00,not3b],[11b2,3_00,not3b]
wip4b,[22b3,2_01,not3],[11b2,3_01,not3b]
wip4ba,[22b3,2_01,not3b],[11b2,3_01,not3b]
wip4c,[22b3,2_10,not3],[11b2,3_10,not3b]
wip4ca,[22b3,2_10,not3b],[11b2,3_10,not3b]
wip4d,[22b3,2_11,not3],[11b2,3_11,not3b]
wip4da,[22b3,2_11,not3b],[11b2,3_11,not3b]
wip4e,[22b3,not2],[11b2,not2]
wip5a,[11b2,1_00,not2],[11b2_fin,2_00,not2b]
wip5aa,[11b2,1_00,not2b],[11b2_fin,2_00,not2b]
wip5b,[11b2,1_01,not2],[11b2_fin,2_01,not2b]
```

```
wip5ba,[11b2,1_01,not2b],[11b2_fin,2_01,not2b]
wip5c,[11b2,1_10,not2],[11b2_fin,2_10,not2b]
wip5ca,[11b2,1_10,not2b],[11b2_fin,2_10,not2b]
wip5d,[11b2,1_11,not2],[11b2_fin,2_11,not2b]
wip5da,[11b2,1_11,not2b],[11b2_fin,2_11,not2b]
wip5e,[11b2,not1],[11b2_fin,not1]
wip6a,[11b2_fin,w2b1_00,2_00],[1_00,w5,enable]
wip6b,[11b2_fin,w2b1_01,2_01],[1_01,w5,enable]
wip6c,[11b2_fin,w2b1_10,2_10],[1_10,w5,enable]
wip6d,[11b2_fin,w2b1_11,2_11],[1_11,w5,enable]
wip6e,[11b2_fin,w3b1_00,3_00],[1_00,w5,enable]
wip6f,[11b2_fin,w3b1_01,3_01],[1_01,w5,enable]
wip6g,[11b2_fin,w3b1_10,3_10],[1_10,w5,enable]
wip6h,[11b2_fin,w3b1_11,3_11],[1_11,w5,enable]
wip6i,[11b2_fin,w4b1_00,4_00,not1],[1_00,w5,enable,not1b]
wip6ia,[11b2_fin,w4b1_00,4_00,not1b],[1_00,w5,enable,not1b]
wip6j,[11b2_fin,w4b1_01,4_01,not1],[1_01,w5,enable,not1b]
wip6ja,[11b2_fin,w4b1_01,4_01,not1b],[1_01,w5,enable,not1b]
wip6k,[11b2_fin,w4b1_10,4_10,not1],[1_10,w5,enable,not1b]
wip6ka,[11b2_fin,w4b1_10,4_10,not1b],[1_10,w5,enable,not1b]
wip6l,[11b2_fin,w4b1_11,4_11,not1],[1_11,w5,enable,not1b]
wip6la,[11b2_fin,w4b1_11,4_11,not1b],[1_11,w5,enable,not1b]
COM wtg := wtg + 1;
w5aa,[w5,wtg=0,enable],[w5_temp1,wtg=1]
w5ab,[w5,wtg=1,enable],[w5_temp1,wtg=2]
w5ac,[w5,wtg=2,enable],[w5_temp1,wtg=0]
COM write E on Awl+1;
w5bb,[w5_temp1,wl=0,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24],[w5_temp2,wl=0,A1=E,enable]
w5bd,[w5_temp1,wl=1,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24],[w5_temp2,wl=1,A2=E,enable]
w5bf,[w5_temp1,wl=2,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24],[w5_temp2,wl=2,A0=E,enable]
COM write (wtg,wt) on Awl;
w5ca,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=1,enable],[w1,wl=0,
wtg=0,wt=1,A0=01,enable]
w5cb,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=2,enable],[w1,wl=0,
wtg=0,wt=2,A0=02,enable]
w5cc,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=3,enable],[w1,wl=0,
wtg=0,wt=3,A0=03,enable]
w5cd,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=0,wt=4,enable],[w1,wl=0,
wtg=0,wt=4,A0=04,enable]
w5ce,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=1,enable],[w1,wl=0,
wtg=1,wt=1,A0=11,enable]
w5cf,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=2,enable],[w1,wl=0,
wtg=1,wt=2,A0=12,enable]
w5cg,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=3,enable],[w1,wl=0,
wtg=1,wt=3,A0=13,enable]
w5ch,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=1,wt=4,enable],[w1,wl=0,
wtg=1,wt=4,A0=14,enable]
w5ci,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=1,enable],[w1,wl=0,
wtg=2,wt=1,A0=21,enable]
w5cj,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=2,enable],[w1,wl=0,
wtg=2,wt=2,A0=22,enable]
w5ck,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=3,enable],[w1,wl=0,
```

wtg=2,wt=3,A0=23,enable]
w5cl,[w5_temp2,wl=0,+0A0=E,+0A0=01,+0A0=02,+0A0=03,+0A0=04,+0A0=11,+0A0=12,
+0A0=13,+0A0=14,+0A0=21,+0A0=22,+0A0=23,+0A0=24,wtg=2,wt=4,enable],[w1,wl=0,
wtg=2,wt=4,A0=24,enable]
w5da,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=1,enable],[w1,wl=1,
wtg=0,wt=1,A1=01,enable]
w5db,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=2,enable],[w1,wl=1,
wtg=0,wt=2,A1=02,enable]
w5dc,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=3,enable],[w1,wl=1,
wtg=0,wt=3,A1=03,enable]
w5dd,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=0,wt=4,enable],[w1,wl=1,
wtg=0,wt=4,A1=04,enable]
w5de,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=1,enable],[w1,wl=1,
wtg=1,wt=1,A1=11,enable]
w5df,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=2,enable],[w1,wl=1,
wtg=1,wt=2,A1=12,enable]
w5dg,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=3,enable],[w1,wl=1,
wtg=1,wt=3,A1=13,enable]
w5dh,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=1,wt=4,enable],[w1,wl=1,
wtg=1,wt=4,A1=14,enable]
w5di,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=1,enable],[w1,wl=1,
wtg=2,wt=1,A1=21,enable]
w5dj,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=2,enable],[w1,wl=1,
wtg=2,wt=2,A1=22,enable]
w5dk,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=3,enable],[w1,wl=1,
wtg=2,wt=3,A1=23,enable]
w5dl,[w5_temp2,wl=1,+0A1=E,+0A1=01,+0A1=02,+0A1=03,+0A1=04,+0A1=11,+0A1=12,
+0A1=13,+0A1=14,+0A1=21,+0A1=22,+0A1=23,+0A1=24,wtg=2,wt=4,enable],[w1,wl=1,
wtg=2,wt=4,A1=24,enable]
w5ea,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=1,enable],[w1,wl=2,
wtg=0,wt=1,A2=01,enable]
w5eb,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=2,enable],[w1,wl=2,
wtg=0,wt=2,A2=02,enable]
w5ec,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=3,enable],[w1,wl=2,
wtg=0,wt=3,A2=03,enable]
w5ed,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=0,wt=4,enable],[w1,wl=2,
wtg=0,wt=4,A2=04,enable]
w5ee,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=1,enable],[w1,wl=2,
wtg=1,wt=1,A2=11,enable]
w5ef,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=2,enable],[w1,wl=2,
wtg=1,wt=2,A2=12,enable]
w5eg,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=3,enable],[w1,wl=2,
wtg=1,wt=3,A2=13,enable]
w5eh,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=1,wt=4,enable],[w1,wl=2,
wtg=1,wt=4,A2=14,enable]
w5ei,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=1,enable],[w1,wl=2,

```
wtg=2,wt=1,A2=21,enable]
w5ej,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=2,enable],[w1,wl=2,
wtg=2,wt=2,A2=22,enable]
w5ek,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=3,enable],[w1,wl=2,
wtg=2,wt=3,A2=23,enable]
w5el,[w5_temp2,wl=2,+0A2=E,+0A2=01,+0A2=02,+0A2=03,+0A2=04,+0A2=11,+0A2=12,
+0A2=13,+0A2=14,+0A2=21,+0A2=22,+0A2=23,+0A2=24,wtg=2,wt=4,enable],[w1,wl=2,
wtg=2,wt=4,A2=24,enable]
COM Clear x
clr_x00,[enable,clear_x,+0x_00,+0xb_00,+1x_01,+1xb_01,+2x_10,+2xb_10,+3x_11,
+3xb_11],[r1,xb_00,xb_01,xb_10,xb_11]
COM - The READER - rl := rl+1;
r1a,[r1,rl=-1],[r2,rl=0]
r1b,[r1,rl=0],[r2,rl=1]
r1c,[r1,rl=1],[r2,rl=2]
r1d,[r1,rl=2],[r2,rl=0]
COM read Arl;
r2aa,[r2,rl=0,A0=E],[r3,rl=0,A0=E,Arl=E]
r2ab,[r2,rl=0,A0=01],[r3,rl=0,A0=01,Arl=01,Arln=E]
r2ac,[r2,rl=0,A0=02],[r3,rl=0,A0=02,Arl=02,Arln=E]
r2ad,[r2,rl=0,A0=03],[r3,rl=0,A0=03,Arl=03,Arln=E]
r2ae,[r2,rl=0,A0=04],[r3,rl=0,A0=04,Arl=04,Arln=E]
r2af,[r2,rl=0,A0=11],[r3,rl=0,A0=11,Arl=11,Arln=E]
r2ag,[r2,rl=0,A0=12],[r3,rl=0,A0=12,Arl=12,Arln=E]
r2ah,[r2,rl=0,A0=13],[r3,rl=0,A0=13,Arl=13,Arln=E]
r2ai,[r2,rl=0,A0=14],[r3,rl=0,A0=14,Arl=14,Arln=E]
r2aj,[r2,rl=0,A0=21],[r3,rl=0,A0=21,Arl=21,Arln=E]
r2ak,[r2,rl=0,A0=22],[r3,rl=0,A0=22,Arl=22,Arln=E]
r2al,[r2,rl=0,A0=23],[r3,rl=0,A0=23,Arl=23,Arln=E]
r2am,[r2,rl=0,A0=24],[r3,rl=0,A0=24,Arl=24,Arln=E]
r2ba,[r2,rl=1,A1=E],[r3,rl=1,A1=E,Arl=E]
r2bb,[r2,rl=1,A1=01],[r3,rl=1,A1=01,Arl=01,Arln=E]
r2bc,[r2,rl=1,A1=02],[r3,rl=1,A1=02,Arl=02,Arln=E]
r2bd,[r2,rl=1,A1=03],[r3,rl=1,A1=03,Arl=03,Arln=E]
r2be,[r2,rl=1,A1=04],[r3,rl=1,A1=04,Arl=04,Arln=E]
r2bf,[r2,rl=1,A1=11],[r3,rl=1,A1=11,Arl=11,Arln=E]
r2bg,[r2,rl=1,A1=12],[r3,rl=1,A1=12,Arl=12,Arln=E]
r2bh,[r2,rl=1,A1=13],[r3,rl=1,A1=13,Arl=13,Arln=E]
r2bi,[r2,rl=1,A1=14],[r3,rl=1,A1=14,Arl=14,Arln=E]
r2bj,[r2,rl=1,A1=21],[r3,rl=1,A1=21,Arl=21,Arln=E]
r2bk,[r2,rl=1,A1=22],[r3,rl=1,A1=22,Arl=22,Arln=E]
r2bl,[r2,rl=1,A1=23],[r3,rl=1,A1=23,Arl=23,Arln=E]
r2bm,[r2,rl=1,A1=24],[r3,rl=1,A1=24,Arl=24,Arln=E]
r2ca,[r2,rl=2,A2=E],[r3,rl=2,A2=E,Arl=E]
r2cb,[r2,rl=2,A2=01],[r3,rl=2,A2=01,Arl=01,Arln=E]
r2cc,[r2,rl=2,A2=02],[r3,rl=2,A2=02,Arl=02,Arln=E]
r2cd,[r2,rl=2,A2=03],[r3,rl=2,A2=03,Arl=03,Arln=E]
r2ce,[r2,rl=2,A2=04],[r3,rl=2,A2=04,Arl=04,Arln=E]
r2cf,[r2,rl=2,A2=11],[r3,rl=2,A2=11,Arl=11,Arln=E]
r2cg,[r2,rl=2,A2=12],[r3,rl=2,A2=12,Arl=12,Arln=E]
r2ch,[r2,rl=2,A2=13],[r3,rl=2,A2=13,Arl=13,Arln=E]
r2ci,[r2,rl=2,A2=14],[r3,rl=2,A2=14,Arl=14,Arln=E]
r2cj,[r2,rl=2,A2=21],[r3,rl=2,A2=21,Arl=21,Arln=E]
r2ck,[r2,rl=2,A2=22],[r3,rl=2,A2=22,Arl=22,Arln=E]
r2cl,[r2,rl=2,A2=23],[r3,rl=2,A2=23,Arl=23,Arln=E]
r2cm,[r2,rl=2,A2=24],[r3,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl=E then do rl=rl-1; reread Arl;
r3aa,[r3,Arl=E,rl=0],[r3a_temp1,rl=2,enable]
r3ab,[r3,Arl=E,rl=1],[r3a_temp1,rl=0,enable]
r3ac,[r3,Arl=E,rl=2],[r3a_temp1,rl=1,enable]
COM (reread Arl)
r3aaa,[r3a_temp1,rl=0,A0=E,enable],[r3a_temp2,rl=0,A0=E,Arl=E]
r3aab,[r3a_temp1,rl=0,A0=01,enable],[r3a_temp2,rl=0,A0=01,Arl=01,Arln=E]
r3aac,[r3a_temp1,rl=0,A0=02,enable],[r3a_temp2,rl=0,A0=02,Arl=02,Arln=E]
```

```
r3aad,[r3a_temp1,rl=0,A0=03,enable],[r3a_temp2,rl=0,A0=03,Arl=03,Arln=E]
r3aae,[r3a_temp1,rl=0,A0=04,enable],[r3a_temp2,rl=0,A0=04,Arl=04,Arln=E]
r3aaf,[r3a_temp1,rl=0,A0=11,enable],[r3a_temp2,rl=0,A0=11,Arl=11,Arln=E]
r3aag,[r3a_temp1,rl=0,A0=12,enable],[r3a_temp2,rl=0,A0=12,Arl=12,Arln=E]
r3aah,[r3a_temp1,rl=0,A0=13,enable],[r3a_temp2,rl=0,A0=13,Arl=13,Arln=E]
r3aai,[r3a_temp1,rl=0,A0=14,enable],[r3a_temp2,rl=0,A0=14,Arl=14,Arln=E]
r3aaj,[r3a_temp1,rl=0,A0=21,enable],[r3a_temp2,rl=0,A0=21,Arl=21,Arln=E]
r3aak,[r3a_temp1,rl=0,A0=22,enable],[r3a_temp2,rl=0,A0=22,Arl=22,Arln=E]
r3aal,[r3a_temp1,rl=0,A0=23,enable],[r3a_temp2,rl=0,A0=23,Arl=23,Arln=E]
r3aam,[r3a_temp1,rl=0,A0=24,enable],[r3a_temp2,rl=0,A0=24,Arl=24,Arln=E]
r3aba,[r3a_temp1,rl=1,A1=E,enable],[r3a_temp2,rl=1,A1=E,Arl=E]
r3abb,[r3a_temp1,rl=1,A1=01,enable],[r3a_temp2,rl=1,A1=01,Arl=01,Arln=E]
r3abc,[r3a_temp1,rl=1,A1=02,enable],[r3a_temp2,rl=1,A1=02,Arl=02,Arln=E]
r3abd,[r3a_temp1,rl=1,A1=03,enable],[r3a_temp2,rl=1,A1=03,Arl=03,Arln=E]
r3abe,[r3a_temp1,rl=1,A1=04,enable],[r3a_temp2,rl=1,A1=04,Arl=04,Arln=E]
r3abf,[r3a_temp1,rl=1,A1=11,enable],[r3a_temp2,rl=1,A1=11,Arl=11,Arln=E]
r3abg,[r3a_temp1,rl=1,A1=12,enable],[r3a_temp2,rl=1,A1=12,Arl=12,Arln=E]
r3abh,[r3a_temp1,rl=1,A1=13,enable],[r3a_temp2,rl=1,A1=13,Arl=13,Arln=E]
r3abi,[r3a_temp1,rl=1,A1=14,enable],[r3a_temp2,rl=1,A1=14,Arl=14,Arln=E]
r3abj,[r3a_temp1,rl=1,A1=21,enable],[r3a_temp2,rl=1,A1=21,Arl=21,Arln=E]
r3abk,[r3a_temp1,rl=1,A1=22,enable],[r3a_temp2,rl=1,A1=22,Arl=22,Arln=E]
r3abl,[r3a_temp1,rl=1,A1=23,enable],[r3a_temp2,rl=1,A1=23,Arl=23,Arln=E]
r3abm,[r3a_temp1,rl=1,A1=24,enable],[r3a_temp2,rl=1,A1=24,Arl=24,Arln=E]
r3aca,[r3a_temp1,rl=2,A2=E,enable],[r3a_temp2,rl=2,A2=E,Arl=E]
r3acb,[r3a_temp1,rl=2,A2=01,enable],[r3a_temp2,rl=2,A2=01,Arl=01,Arln=E]
r3acc,[r3a_temp1,rl=2,A2=02,enable],[r3a_temp2,rl=2,A2=02,Arl=02,Arln=E]
r3acd,[r3a_temp1,rl=2,A2=03,enable],[r3a_temp2,rl=2,A2=03,Arl=03,Arln=E]
r3ace,[r3a_temp1,rl=2,A2=04,enable],[r3a_temp2,rl=2,A2=04,Arl=04,Arln=E]
r3acf,[r3a_temp1,rl=2,A2=11,enable],[r3a_temp2,rl=2,A2=11,Arl=11,Arln=E]
r3acg,[r3a_temp1,rl=2,A2=12,enable],[r3a_temp2,rl=2,A2=12,Arl=12,Arln=E]
r3ach,[r3a_temp1,rl=2,A2=13,enable],[r3a_temp2,rl=2,A2=13,Arl=13,Arln=E]
r3aci,[r3a_temp1,rl=2,A2=14,enable],[r3a_temp2,rl=2,A2=14,Arl=14,Arln=E]
r3acj,[r3a_temp1,rl=2,A2=21,enable],[r3a_temp2,rl=2,A2=21,Arl=21,Arln=E]
r3ack,[r3a_temp1,rl=2,A2=22,enable],[r3a_temp2,rl=2,A2=22,Arl=22,Arln=E]
r3acl,[r3a_temp1,rl=2,A2=23,enable],[r3a_temp2,rl=2,A2=23,Arl=23,Arln=E]
r3acm,[r3a_temp1,rl=2,A2=24,enable],[r3a_temp2,rl=2,A2=24,Arl=24,Arln=E]
COM if Arl n= E and tag of Arl=rtg+1 then store value of Arl into (rtg,rt) od
COM tag = 1st digit of Arl = (rtg,rt)
r3ada,[r3a_temp2,Arln=E,rtg=2,Arl=01,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=0,rt=1,enable]
r3adb,[r3a_temp2,Arln=E,rtg=2,Arl=02,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=0,rt=2,enable]
r3adc,[r3a_temp2,Arln=E,rtg=2,Arl=03,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=0,rt=3,enable]
r3add,[r3a_temp2,Arln=E,rtg=2,Arl=04,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=0,rt=4,enable]
r3ade,[r3a_temp2,Arln=E,rtg=0,Arl=11,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=1,rt=1,enable]
r3adf,[r3a_temp2,Arln=E,rtg=0,Arl=12,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=1,rt=2,enable]
r3adg,[r3a_temp2,Arln=E,rtg=0,Arl=13,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=1,rt=3,enable]
r3adh,[r3a_temp2,Arln=E,rtg=0,Arl=14,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=1,rt=4,enable]
r3adi,[r3a_temp2,Arln=E,rtg=1,Arl=21,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=2,rt=1,enable]
r3adj,[r3a_temp2,Arln=E,rtg=1,Arl=22,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=2,rt=2,enable]
r3adk,[r3a_temp2,Arln=E,rtg=1,Arl=23,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=2,rt=3,enable]
r3adl,[r3a_temp2,Arln=E,rtg=1,Arl=24,+0rt=1,+0rt=2,+0rt=3,+0rt=4],[rcp_roles,
rtg=2,rt=4,enable]
COM if Arl = E then move on to r4 read
r3ads,[r3a_temp2,Arl=E],[rcp_roles,enable]
COM if Arl n= E, but tag of Arl n= rtg + 1
r3adt,[r3a_temp2,Arln=E,rtg=2,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=21,
```

```
+0Arl=22,+0Arl=23,+0Arl=24],[rcp_roles,rtg=2,enable]
r3adu,[r3a_temp2,Arln=E,rtg=0,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=21,
+0Arl=22,+0Arl=23,+0Arl=24],[rcp_roles,rtg=0,enable]
r3adv,[r3a_temp2,Arln=E,rtg=1,+0Arl=11,+0Arl=12,+0Arl=13,+0Arl=14,+0Arl=01,
+0Arl=02,+0Arl=03,+0Arl=04],[rcp_roles,rtg=1,enable]
COM else do - store value of Arl into s;
r3baa,[r3,Arln=E,Arl=01],[r3b_temp1,s=01,enable]
r3bab,[r3,Arln=E,Arl=02],[r3b_temp1,s=02,enable]
r3bac,[r3,Arln=E,Arl=03],[r3b_temp1,s=03,enable]
r3bad,[r3,Arln=E,Arl=04],[r3b_temp1,s=04,enable]
r3bae,[r3,Arln=E,Arl=11],[r3b_temp1,s=11,enable]
r3baf,[r3,Arln=E,Arl=12],[r3b_temp1,s=12,enable]
r3bag,[r3,Arln=E,Arl=13],[r3b_temp1,s=13,enable]
r3bah,[r3,Arln=E,Arl=14],[r3b_temp1,s=14,enable]
r3bai,[r3,Arln=E,Arl=21],[r3b_temp1,s=21,enable]
r3baj,[r3,Arln=E,Arl=22],[r3b_temp1,s=22,enable]
r3bak,[r3,Arln=E,Arl=23],[r3b_temp1,s=23,enable]
r3bal,[r3,Arln=E,Arl=24],[r3b_temp1,s=24,enable]
COM write E onto RQrl+1
r3bba,[r3b_temp1,rl=0,+0RQ1=E,+0RQ1=P,enable],[r3b_temp2,rl=0,RQ1=E,enable]
r3bbb,[r3b_temp1,rl=1,+0RQ2=E,+0RQ2=P,enable],[r3b_temp2,rl=1,RQ2=E,enable]
r3bbc,[r3b_temp1,rl=2,+0RQ0=E,+0RQ0=P,enable],[r3b_temp2,rl=2,RQ0=E,enable]
COM write P into RQrl
r3bca,[r3b_temp2,rl=0,+0RQ0=E,+0RQ0=P,enable],[r3b_temp3,rl=0,RQ0=P,enable]
r3bcb,[r3b_temp2,rl=1,+0RQ1=E,+0RQ1=P,enable],[r3b_temp3,rl=1,RQ1=P,enable]
r3bcc,[r3b_temp2,rl=2,+0RQ2=E,+0RQ2=P,enable],[r3b_temp3,rl=2,RQ2=P,enable]
COM read Arl
r3bda,[r3b_temp3,rl=0,A0=E,enable],[r3b_temp4,rl=0,A0=E,Arl=E]
r3bdb,[r3b_temp3,rl=0,A0=01,enable],[r3b_temp4,rl=0,A0=01,Arl=01,Arln=E]
r3bdc,[r3b_temp3,rl=0,A0=02,enable],[r3b_temp4,rl=0,A0=02,Arl=02,Arln=E]
r3bdd,[r3b_temp3,rl=0,A0=03,enable],[r3b_temp4,rl=0,A0=03,Arl=03,Arln=E]
r3bde,[r3b_temp3,rl=0,A0=04,enable],[r3b_temp4,rl=0,A0=04,Arl=04,Arln=E]
r3bdf,[r3b_temp3,rl=0,A0=11,enable],[r3b_temp4,rl=0,A0=11,Arl=11,Arln=E]
r3bdg,[r3b_temp3,rl=0,A0=12,enable],[r3b_temp4,rl=0,A0=12,Arl=12,Arln=E]
r3bdh,[r3b_temp3,rl=0,A0=13,enable],[r3b_temp4,rl=0,A0=13,Arl=13,Arln=E]
r3bdi,[r3b_temp3,rl=0,A0=14,enable],[r3b_temp4,rl=0,A0=14,Arl=14,Arln=E]
r3bdj,[r3b_temp3,rl=0,A0=21,enable],[r3b_temp4,rl=0,A0=21,Arl=21,Arln=E]
r3bdk,[r3b_temp3,rl=0,A0=22,enable],[r3b_temp4,rl=0,A0=22,Arl=22,Arln=E]
r3bdl,[r3b_temp3,rl=0,A0=23,enable],[r3b_temp4,rl=0,A0=23,Arl=23,Arln=E]
r3bdm,[r3b_temp3,rl=0,A0=24,enable],[r3b_temp4,rl=0,A0=24,Arl=24,Arln=E]
r3bea,[r3b_temp3,rl=1,A1=E,enable],[r3b_temp4,rl=1,A1=E,Arl=E]
r3beb,[r3b_temp3,rl=1,A1=01,enable],[r3b_temp4,rl=1,A1=01,Arl=01,Arln=E]
r3bec,[r3b_temp3,rl=1,A1=02,enable],[r3b_temp4,rl=1,A1=02,Arl=02,Arln=E]
r3bed,[r3b_temp3,rl=1,A1=03,enable],[r3b_temp4,rl=1,A1=03,Arl=03,Arln=E]
r3bee,[r3b_temp3,rl=1,A1=04,enable],[r3b_temp4,rl=1,A1=04,Arl=04,Arln=E]
r3bef,[r3b_temp3,rl=1,A1=11,enable],[r3b_temp4,rl=1,A1=11,Arl=11,Arln=E]
r3beg,[r3b_temp3,rl=1,A1=12,enable],[r3b_temp4,rl=1,A1=12,Arl=12,Arln=E]
r3beh,[r3b_temp3,rl=1,A1=13,enable],[r3b_temp4,rl=1,A1=13,Arl=13,Arln=E]
r3bei,[r3b_temp3,rl=1,A1=14,enable],[r3b_temp4,rl=1,A1=14,Arl=14,Arln=E]
r3bej,[r3b_temp3,rl=1,A1=21,enable],[r3b_temp4,rl=1,A1=21,Arl=21,Arln=E]
r3bek,[r3b_temp3,rl=1,A1=22,enable],[r3b_temp4,rl=1,A1=22,Arl=22,Arln=E]
r3bel,[r3b_temp3,rl=1,A1=23,enable],[r3b_temp4,rl=1,A1=23,Arl=23,Arln=E]
r3bem,[r3b_temp3,rl=1,A1=24,enable],[r3b_temp4,rl=1,A1=24,Arl=24,Arln=E]
r3bfa,[r3b_temp3,rl=2,A2=E,enable],[r3b_temp4,rl=2,A2=E,Arl=E]
r3bfb,[r3b_temp3,rl=2,A2=01,enable],[r3b_temp4,rl=2,A2=01,Arl=01,Arln=E]
r3bfc,[r3b_temp3,rl=2,A2=02,enable],[r3b_temp4,rl=2,A2=02,Arl=02,Arln=E]
r3bfd,[r3b_temp3,rl=2,A2=03,enable],[r3b_temp4,rl=2,A2=03,Arl=03,Arln=E]
r3bfe,[r3b_temp3,rl=2,A2=04,enable],[r3b_temp4,rl=2,A2=04,Arl=04,Arln=E]
r3bff,[r3b_temp3,rl=2,A2=11,enable],[r3b_temp4,rl=2,A2=11,Arl=11,Arln=E]
r3bfg,[r3b_temp3,rl=2,A2=12,enable],[r3b_temp4,rl=2,A2=12,Arl=12,Arln=E]
r3bfh,[r3b_temp3,rl=2,A2=13,enable],[r3b_temp4,rl=2,A2=13,Arl=13,Arln=E]
r3bfi,[r3b_temp3,rl=2,A2=14,enable],[r3b_temp4,rl=2,A2=14,Arl=14,Arln=E]
r3bfj,[r3b_temp3,rl=2,A2=21,enable],[r3b_temp4,rl=2,A2=21,Arl=21,Arln=E]
r3bfk,[r3b_temp3,rl=2,A2=22,enable],[r3b_temp4,rl=2,A2=22,Arl=22,Arln=E]
r3bfl,[r3b_temp3,rl=2,A2=23,enable],[r3b_temp4,rl=2,A2=23,Arl=23,Arln=E]
r3bfm,[r3b_temp3,rl=2,A2=24,enable],[r3b_temp4,rl=2,A2=24,Arl=24,Arln=E]
```

```
COM if Arl n= E then store value of Arl into (rtg,rt) else (rtg,rt:=s) od
r3bga,[r3b_temp4,Arln=E,Arl=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=1]
r3bgb,[r3b_temp4,Arln=E,Arl=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=2]
r3bgc,[r3b_temp4,Arln=E,Arl=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=3]
r3bgd,[r3b_temp4,Arln=E,Arl=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=0,rt=4]
r3bge,[r3b_temp4,Arln=E,Arl=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=1]
r3bgf,[r3b_temp4,Arln=E,Arl=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=2]
r3bgg,[r3b_temp4,Arln=E,Arl=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=3]
r3bgh,[r3b_temp4,Arln=E,Arl=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=1,rt=4]
r3bgi,[r3b_temp4,Arln=E,Arl=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=1]
r3bgj,[r3b_temp4,Arln=E,Arl=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=2]
r3bgk,[r3b_temp4,Arln=E,Arl=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=3]
r3bgl,[r3b_temp4,Arln=E,Arl=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp5,rtg=2,rt=4]
r3bha,[r3b_temp5,+0s=01,+0s=02,+0s=03,+0s=04,+0s=11,+0s=12,+0s=13,+0s=14,
+0s=21,+0s=22,+0s=23,+0s=24],[rcp_roles,enable]
r3bia,[r3b_temp4,Arl=E,s=01,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=1]
r3bib,[r3b_temp4,Arl=E,s=02,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=2]
r3bic,[r3b_temp4,Arl=E,s=03,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=3]
r3bid,[r3b_temp4,Arl=E,s=04,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=0,rt=4]
r3bie,[r3b_temp4,Arl=E,s=11,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=1]
r3bif,[r3b_temp4,Arl=E,s=12,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=2]
r3big,[r3b_temp4,Arl=E,s=13,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=3]
r3bih,[r3b_temp4,Arl=E,s=14,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=1,rt=4]
r3bii,[r3b_temp4,Arl=E,s=21,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=1]
r3bij,[r3b_temp4,Arl=E,s=22,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=2]
r3bik,[r3b_temp4,Arl=E,s=23,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=3]
r3bil,[r3b_temp4,Arl=E,s=24,+0rtg=0,+0rtg=1,+0rtg=2,+1rt=1,+1rt=2,+1rt=3,
+1rt=4],[r3b_temp6,rtg=2,rt=4]
r3bja,[r3b_temp6,+0Arl=01,+0Arl=02,+0Arl=03,+0Arl=04,+0Arl=11,+0Arl=12,
+0Arl=13,+0Arl=14,+0Arl=21,+0Arl=22,+0Arl=23,+0Arl=24],[rcp_roles,enable]
COM rcp roles
x_r00a,[enable,rcp_roles,rt=1,X_00,xb_00],[rt=1,X_00,x_00,rcp_roles2_00]
x_r00b,[enable,rcp_roles,rt=1,Xb_00],[rt=1,Xb_00,rcp_roles2_00]
x_r01a,[enable,rcp_roles,rt=2,X_01,xb_01],[rt=2,X_01,x_01,rcp_roles2_01]
x_r01b,[enable,rcp_roles,rt=2,Xb_01],[rt=2,Xb_01,rcp_roles2_01]
x_r10a,[enable,rcp_roles,rt=3,X_10,xb_10],[rt=3,X_10,x_10,rcp_roles2_10]
x_r10b,[enable,rcp_roles,rt=3,Xb_10],[rt=3,Xb_10,rcp_roles2_10]
x_r11a,[enable,rcp_roles,rt=4,X_11,xb_11],[rt=4,X_11,x_11,rcp_roles2_11]
x_r11b,[enable,rcp_roles,rt=4,Xb_11],[rt=4,Xb_11,rcp_roles2_11]
X_r00a,[rcp_roles2_00,1_00],[1_00,rcp_Xb_00,rcp_X_01,rcp_X_10,rcp_X_11,
rcp_role3]
X_r00b,[rcp_roles2_00,2_00],[2_00,rcp_Xb_00,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
```

```
X_r00c,[rcp_roles2_00,3_00],[3_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r00d,[rcp_roles2_00,4_00],[4_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r01a,[rcp_roles2_01,1_01],[1_01,rcp_Xb_01,rcp_X_00,rcp_X_10,rcp_X_11,
rcp_role3]
X_r01b,[rcp_roles2_01,2_01],[2_01,rcp_Xb_01,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r01c,[rcp_roles2_01,3_01],[3_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r01d,[rcp_roles2_01,4_01],[4_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r10a,[rcp_roles2_10,1_10],[1_10,rcp_Xb_10,rcp_X_01,rcp_X_00,rcp_X_11,
rcp_role3]
X_r10b,[rcp_roles2_10,2_10],[2_10,rcp_Xb_10,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r10c,[rcp_roles2_10,3_10],[3_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r10d,[rcp_roles2_10,4_10],[4_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r11a,[rcp_roles2_11,1_11],[1_11,rcp_Xb_11,rcp_X_01,rcp_X_10,rcp_X_00,
rcp_role3]
X_r11b,[rcp_roles2_11,2_11],[2_11,rcp_Xb_11,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r11c,[rcp_roles2_11,3_11],[3_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r11d,[rcp_roles2_11,4_11],[4_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_00,X_00],[Xb_00,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_00,Xb_00],[Xb_00,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_01,X_01],[Xb_01,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_01,Xb_01],[Xb_01,rcp_role4]
rcp_3e,[rcp_role3,rcp_Xb_10,X_10],[Xb_10,rcp_role4]
rcp_3f,[rcp_role3,rcp_Xb_10,Xb_10],[Xb_10,rcp_role4]
rcp_3g,[rcp_role3,rcp_Xb_11,X_11],[Xb_11,rcp_role4]
rcp_3h,[rcp_role3,rcp_Xb_11,Xb_11],[Xb_11,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_01,X_01],[X_01,rcp_role5]
rcp_4b,[rcp_role4,rcp_X_01,Xb_01],[X_01,rcp_role5]
rcp_4c,[rcp_role4,rcp_X_00,Xb_00],[X_00,rcp_role5]
rcp_4d,[rcp_role4,rcp_X_00,X_00],[X_00,rcp_role5]
rcp_4e,[rcp_role4,rcp_Xb_1,1_00,X_00],[1_00,Xb_00,rcp_role5]
rcp_4f,[rcp_role4,rcp_Xb_1,1_00,Xb_00],[1_00,Xb_00,rcp_role5]
rcp_4g,[rcp_role4,rcp_Xb_1,1_01,X_01],[1_01,Xb_01,rcp_role5]
rcp_4h,[rcp_role4,rcp_Xb_1,1_01,Xb_01],[1_01,Xb_01,rcp_role5]
rcp_4i,[rcp_role4,rcp_Xb_1,1_10,X_10],[1_10,Xb_10,rcp_role5]
rcp_4j,[rcp_role4,rcp_Xb_1,1_10,Xb_10],[1_10,Xb_10,rcp_role5]
rcp_4k,[rcp_role4,rcp_Xb_1,1_11,X_11],[1_11,Xb_11,rcp_role5]
rcp_4l,[rcp_role4,rcp_Xb_1,1_11,Xb_11],[1_11,Xb_11,rcp_role5]
rcp_5a,[rcp_role5,rcp_X_10,X_10],[X_10,rcp_role6]
rcp_5b,[rcp_role5,rcp_X_10,Xb_10],[X_10,rcp_role6]
rcp_5c,[rcp_role5,rcp_X_00,X_00],[X_00,rcp_role6]
rcp_5d,[rcp_role5,rcp_X_00,Xb_00],[X_00,rcp_role6]
rcp_5e,[rcp_role5,rcp_Xb_2,2_00,X_00],[2_00,Xb_00,rcp_role6]
rcp_5f,[rcp_role5,rcp_Xb_2,2_00,Xb_00],[2_00,Xb_00,rcp_role6]
rcp_5g,[rcp_role5,rcp_Xb_2,2_01,X_01],[2_01,Xb_01,rcp_role6]
rcp_5h,[rcp_role5,rcp_Xb_2,2_01,Xb_01],[2_01,Xb_01,rcp_role6]
rcp_5i,[rcp_role5,rcp_Xb_2,2_10,X_10],[2_10,Xb_10,rcp_role6]
rcp_5j,[rcp_role5,rcp_Xb_2,2_10,Xb_10],[2_10,Xb_10,rcp_role6]
rcp_5k,[rcp_role5,rcp_Xb_2,2_11,X_11],[2_11,Xb_11,rcp_role6]
rcp_5l,[rcp_role5,rcp_Xb_2,2_11,Xb_11],[2_11,Xb_11,rcp_role6]
rcp_5m,[rcp_role5,rcp_X_3,3_00,X_00],[3_00,X_00,rcp_role6]
rcp_5n,[rcp_role5,rcp_X_3,3_00,Xb_00],[3_00,X_00,rcp_role6]
rcp_5o,[rcp_role5,rcp_X_3,3_01,X_01],[3_01,X_01,rcp_role6]
rcp_5p,[rcp_role5,rcp_X_3,3_01,Xb_01],[3_01,X_01,rcp_role6]
rcp_5q,[rcp_role5,rcp_X_3,3_10,X_10],[3_10,X_10,rcp_role6]
rcp_5r,[rcp_role5,rcp_X_3,3_10,Xb_10],[3_10,X_10,rcp_role6]
```

```
rcp_5s,[rcp_role5,rcp_X_3,3_11,X_11],[3_11,X_11,rcp_role6]
rcp_5t,[rcp_role5,rcp_X_3,3_11,Xb_11],[3_11,X_11,rcp_role6]
rcp_6a,[rcp_role6,rcp_X_11,X_11],[X_11,r4]
rcp_6b,[rcp_role6,rcp_X_11,Xb_11],[X_11,r4]
rcp_6c,[rcp_role6,rcp_X_00,X_00],[X_00,r4]
rcp_6d,[rcp_role6,rcp_X_00,Xb_00],[X_00,r4]
rcp_6e,[rcp_role6,rcp_Xb_3,3_00,X_00],[3_00,Xb_00,r4]
rcp_6f,[rcp_role6,rcp_Xb_3,3_00,Xb_00],[3_00,Xb_00,r4]
rcp_6g,[rcp_role6,rcp_Xb_3,3_01,X_01],[3_01,Xb_01,r4]
rcp_6h,[rcp_role6,rcp_Xb_3,3_01,Xb_01],[3_01,Xb_01,r4]
rcp_6i,[rcp_role6,rcp_Xb_3,3_10,X_10],[3_10,Xb_10,r4]
rcp_6j,[rcp_role6,rcp_Xb_3,3_10,Xb_10],[3_10,Xb_10,r4]
rcp_6k,[rcp_role6,rcp_Xb_3,3_11,X_11],[3_11,Xb_11,r4]
rcp_6l,[rcp_role6,rcp_Xb_3,3_11,Xb_11],[3_11,Xb_11,r4]
rcp_6m,[rcp_role6,rcp_X_4,4_00,X_00],[4_00,X_00,r4]
rcp_6n,[rcp_role6,rcp_X_4,4_00,Xb_00],[4_00,X_00,r4]
rcp_6o,[rcp_role6,rcp_X_4,4_01,X_01],[4_01,X_01,r4]
rcp_6p,[rcp_role6,rcp_X_4,4_01,Xb_01],[4_01,X_01,r4]
rcp_6q,[rcp_role6,rcp_X_4,4_10,X_10],[4_10,X_10,r4]
rcp_6r,[rcp_role6,rcp_X_4,4_10,Xb_10],[4_10,X_10,r4]
rcp_6s,[rcp_role6,rcp_X_4,4_11,X_11],[4_11,X_11,r4]
rcp_6t,[rcp_role6,rcp_X_4,4_11,Xb_11],[4_11,X_11,r4]
COM read track rt;
r4a,[r4,rt=1],[rt=1,reading_1,enable]
r4b,[reading_1,enable],[clear_x,enable]
r4c,[r4,rt=2],[rt=2,reading_2,enable]
r4d,[reading_2,enable],[clear_x,enable]
r4e,[r4,rt=3],[rt=3,reading_3,enable]
r4f,[reading_3,enable],[clear_x,enable]
r4g,[r4,rt=4],[rt=4,reading_4,enable]
r4h,[reading_4,enable],[clear_x,enable]
COM check sequence
order_lost_00,[enable,reading_1,x_00],[enable,reading_1,x_00,order_lost]
order_lost_01,[enable,reading_2,x_01],[enable,reading_2,x_01,order_lost]
order_lost_10,[enable,reading_3,x_10],[enable,reading_3,x_10,order_lost]
order_lost_11,[enable,reading_4,x_11],[enable,reading_4,x_11,order_lost]
END
```

B.3 Tromp '$4b + 14$' Petri net models

B.3.1 Model for testing coherence

```
Tromp 4b+8 4 track
bad=[coherence_lost]
initial=[w1,r1,w=0,r=0,R0=0,R1=0,W0=0,W1=0,x0=0,x1=0,D00=0,D01=0,D10=0,
D11=0,d0=0,d1=0]
COM
COM writer
COM
COM 1: if R[1-w] == W[1-w] then ( if R[not w] == W[not w] then )
COM
w1a,[w1,w=0,R1=0,W1=0],[w2,w=0,R1=0,W1=0]
w1b,[w1,w=0,R1=1,W1=1],[w2,w=0,R1=1,W1=1]
w1c,[w1,w=1,R0=0,W0=0],[w2,w=1,R0=0,W0=0]
w1d,[w1,w=1,R0=1,W0=1],[w2,w=1,R0=1,W0=1]
w1e,[w1,w=0,R1=0,W1=1],[w6,w=0,R1=0,W1=1]
w1f,[w1,w=0,R1=1,W1=0],[w6,w=0,R1=1,W1=0]
w1g,[w1,w=1,R0=0,W0=1],[w6,w=1,R0=0,W0=1]
w1h,[w1,w=1,R0=1,W0=0],[w6,w=1,R0=1,W0=0]
COM
COM 2: w := 1-w (w := not w)
COM
```

```
w2a,[w2,w=0],[w3,w=1]
w2b,[w2,w=1],[w3,w=0]
COM
COM 3: write track T[w,x[w]]
COM
w3_strt_wr_T00, [w3,w=0,x0=0],[writing_00,w=0,x0=0,a]
w3_end_wr_T00,  [writing_00,a],[w4]
w3_strt_wr_T01, [w3,w=0,x0=1],[writing_01,w=0,x0=1,a]
w3_end_wr_T01,  [writing_01,a],[w4]
w3_strt_wr_T10, [w3,w=1,x1=0],[writing_10,w=1,x1=0,a]
w3_end_wr_T10,  [writing_10,a],[w4]
w3_strt_wr_T11, [w3,w=1,x1=1],[writing_11,w=1,x1=1,a]
w3_end_wr_T11,  [writing_11,a],[w4]
COM
COM 4: change W[1-w] (change W[not w])
COM
w4a,[w4,w=0,W1=0],[w1,w=0,W1=1]
w4b,[w4,w=0,W1=1],[w1,w=0,W1=0]
w4c,[w4,w=1,W0=0],[w1,w=1,W0=1]
w4d,[w4,w=1,W0=1],[w1,w=1,W0=0]
COM
COM 5: else
COM
COM 6: x[w] := 1 - x[w] (x[w] := not x[w])
COM
w6a,[w6,w=0,x0=0],[w7,w=0,x0=1]
w6b,[w6,w=0,x0=1],[w7,w=0,x0=0]
w6c,[w6,w=1,x1=0],[w7,w=1,x1=1]
w6d,[w6,w=1,x1=1],[w7,w=1,x1=0]
COM
COM 7: write track T[w,x[w]]
COM
w7_strt_wr_T00, [w7,w=0,x0=0],[writing_00,w=0,x0=0,b]
w7_end_wr_T00,  [writing_00,b],[w8]
w7_strt_wr_T01, [w7,w=0,x0=1],[writing_01,w=0,x0=1,b]
w7_end_wr_T01,  [writing_01,b],[w8]
w7_strt_wr_T10, [w7,w=1,x1=0],[writing_10,w=1,x1=0,b]
w7_end_wr_T10,  [writing_10,b],[w8]
w7_strt_wr_T11, [w7,w=1,x1=1],[writing_11,w=1,x1=1,b]
w7_end_wr_T11,  [writing_11,b],[w8]
COM
COM 8: change D[w,1-x[w]] (change D[w, not x[w]])
COM
w8a,[w8,w=0,x0=1,D00=0],[w1,w=0,x0=1,D00=1]
w8b,[w8,w=0,x0=1,D00=1],[w1,w=0,x0=1,D00=0]
w8c,[w8,w=0,x0=0,D01=0],[w1,w=0,x0=0,D01=1]
w8d,[w8,w=0,x0=0,D01=1],[w1,w=0,x0=0,D01=0]
w8e,[w8,w=1,x1=1,D10=0],[w1,w=1,x1=1,D10=1]
w8f,[w8,w=1,x1=1,D10=1],[w1,w=1,x1=1,D10=0]
w8g,[w8,w=1,x1=0,D11=0],[w1,w=1,x1=0,D11=1]
w8h,[w8,w=1,x1=0,D11=1],[w1,w=1,x1=0,D11=0]
COM
COM 9: end if
COM
COM reader
COM
COM 1: if W[r] <> R[r] then
COM
r1a,[r1,r=0,W0=0,R0=1],[r2,r=0,W0=0,R0=1]
r1b,[r1,r=0,W0=1,R0=0],[r2,r=0,W0=1,R0=0]
r1c,[r1,r=0,W0=0,R0=0],[r7,r=0,W0=0,R0=0]
r1d,[r1,r=0,W0=1,R0=1],[r7,r=0,W0=1,R0=1]
r1e,[r1,r=1,W1=0,R1=1],[r2,r=1,W1=0,R1=1]
r1f,[r1,r=1,W1=1,R1=0],[r2,r=1,W1=1,R1=0]
r1g,[r1,r=1,W1=0,R1=0],[r7,r=1,W1=0,R1=0]
r1h,[r1,r=1,W1=1,R1=1],[r7,r=1,W1=1,R1=1]
```

```
COM
COM 2: change R[r]
COM
r2a,[r2,r=0,R0=0],[r3,r=0,R0=1]
r2b,[r2,r=0,R0=1],[r3,r=0,R0=0]
r2c,[r2,r=1,R1=0],[r3,r=1,R1=1]
r2d,[r2,r=1,R1=1],[r3,r=1,R1=0]
COM
COM 3: r := 1-r (r := not r)
COM
r3a,[r3,r=0],[r4,r=1]
r3b,[r3,r=1],[r4,r=0]
COM
COM 4: read d[0] := D[r,0]
COM
r4a,[r4,r=0,D00=0,+0d0=0,+0d0=1],[r5,r=0,D00=0,d0=0]
r4b,[r4,r=0,D00=1,+0d0=0,+0d0=1],[r5,r=0,D00=1,d0=1]
r4c,[r4,r=1,D10=0,+0d0=0,+0d0=1],[r5,r=1,D10=0,d0=0]
r4d,[r4,r=1,D10=1,+0d0=0,+0d0=1],[r5,r=1,D10=1,d0=1]
COM
COM 5: read d[1] := D[r,1]
COM
r5a,[r5,r=0,D01=0,+0d1=0,+0d1=1],[r9,r=0,D01=0,d1=0]
r5b,[r5,r=0,D01=1,+0d1=0,+0d1=1],[r9,r=0,D01=1,d1=1]
r5c,[r5,r=1,D11=0,+0d1=0,+0d1=1],[r9,r=1,D11=0,d1=0]
r5d,[r5,r=1,D11=1,+0d1=0,+0d1=1],[r9,r=1,D11=1,d1=1]
COM
COM 6: else
COM
COM 7: read d[d[0] (xor) d[1]] := D[r,d[0] (xor) d[1]]
COM
r7a,[r7,d0=0,d1=0,r=0,D00=0],[r9,d0=0,d1=0,r=0,D00=0]
r7b,[r7,d0=0,d1=0,r=0,D00=1],[r9,d0=1,d1=0,r=0,D00=1]
r7c,[r7,d0=0,d1=0,r=1,D10=0],[r9,d0=0,d1=0,r=1,D10=0]
r7d,[r7,d0=0,d1=0,r=1,D10=1],[r9,d0=1,d1=0,r=1,D10=1]
r7e,[r7,d0=1,d1=1,r=0,D00=0],[r9,d0=0,d1=1,r=0,D00=0]
r7f,[r7,d0=1,d1=1,r=0,D00=1],[r9,d0=1,d1=1,r=0,D00=1]
r7g,[r7,d0=1,d1=1,r=1,D10=0],[r9,d0=0,d1=1,r=1,D10=0]
r7h,[r7,d0=1,d1=1,r=1,D10=1],[r9,d0=1,d1=1,r=1,D10=1]
r7i,[r7,d0=0,d1=1,r=0,D01=0],[r9,d0=0,d1=0,r=0,D01=0]
r7j,[r7,d0=0,d1=1,r=0,D01=1],[r9,d0=0,d1=1,r=0,D01=1]
r7k,[r7,d0=0,d1=1,r=1,D11=0],[r9,d0=0,d1=0,r=1,D11=0]
r7l,[r7,d0=0,d1=1,r=1,D11=1],[r9,d0=0,d1=1,r=1,D11=1]
r7m,[r7,d0=1,d1=0,r=0,D01=0],[r9,d0=1,d1=0,r=0,D01=0]
r7n,[r7,d0=1,d1=0,r=0,D01=1],[r9,d0=1,d1=1,r=0,D01=1]
r7o,[r7,d0=1,d1=0,r=1,D11=0],[r9,d0=1,d1=0,r=1,D11=0]
r7p,[r7,d0=1,d1=0,r=1,D11=1],[r9,d0=1,d1=1,r=1,D11=1]
COM
COM 8: endif
COM
COM 9: read track T[r,d[0] (xor) d[1]]
COM
r9_strt_rd_T00a,[r9,r=0,d0=0,d1=0],[reading_00,r=0,d0=0,d1=0]
r9_strt_rd_T00b,[r9,r=0,d0=1,d1=1],[reading_00,r=0,d0=1,d1=1]
r9_end_rd_T00,[reading_00],[r1]
r9_strt_rd_T01a,[r9,r=0,d0=0,d1=1],[reading_01,r=0,d0=0,d1=1]
r9_strt_rd_T01b,[r9,r=0,d0=1,d1=0],[reading_01,r=0,d0=1,d1=0]
r9_end_rd_T01,[reading_01],[r1]
r9_strt_rd_T10a,[r9,r=1,d0=0,d1=0],[reading_10,r=1,d0=0,d1=0]
r9_strt_rd_T10b,[r9,r=1,d0=1,d1=1],[reading_10,r=1,d0=1,d1=1]
r9_end_rd_T10,[reading_10],[r1]
r9_strt_rd_T11a,[r9,r=1,d0=0,d1=1],[reading_11,r=1,d0=0,d1=1]
r9_strt_rd_T11b,[r9,r=1,d0=1,d1=0],[reading_11,r=1,d0=1,d1=0]
r9_end_rd_T11,[reading_11],[r1]
COM
COM check coherence
```

```
COM
coherence_00,[writing_00,reading_00],[coherence_lost]
coherence_01,[writing_01,reading_01],[coherence_lost]
coherence_10,[writing_10,reading_10],[coherence_lost]
coherence_11,[writing_11,reading_11],[coherence_lost]
END
```

B.3.2 Model for testing freshness

```
Tromp 4b+8 4 track freshness.
bad=[freshness_lost]
initial=[enable,w1,w=0,r=0,R0=0,R1=0,W0=0,W1=0,x0=0,x1=0,D00=0,D01=0,D10=0,
D11=0,d0=0,d1=0,clear_valid,no_w1,no_r0,no_w_fresh,no_w_prev,not_r_fresh_00,
not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,not_r_prev_00,not_r_prev_01,
not_r_prev_10,not_r_prev_11,not_valid_00,not_valid_01,not_valid_10,
not_valid_11]
COM
COM writer
COM
COM 1: if R[1-w] == W[1-w] then ( if R[not w] == W[not w] then )
COM
w1a,[w1,w=0,R1=0,W1=0,enable],[w2,w=0,R1=0,W1=0,enable]
w1b,[w1,w=0,R1=1,W1=1,enable],[w2,w=0,R1=1,W1=1,enable]
w1c,[w1,w=1,R0=0,W0=0,enable],[w2,w=1,R0=0,W0=0,enable]
w1d,[w1,w=1,R0=1,W0=1,enable],[w2,w=1,R0=1,W0=1,enable]
w1e,[w1,w=0,R1=0,W1=1,enable],[w6,w=0,R1=0,W1=1,enable]
w1f,[w1,w=0,R1=1,W1=0,enable],[w6,w=0,R1=1,W1=0,enable]
w1g,[w1,w=1,R0=0,W0=1,enable],[w6,w=1,R0=0,W0=1,enable]
w1h,[w1,w=1,R0=1,W0=0,enable],[w6,w=1,R0=1,W0=0,enable]
COM
COM 2: w := 1-w (w := not w)
COM
w2a,[w2,w=0,enable],[w3,w=1,enable]
w2b,[w2,w=1,enable],[w3,w=0,enable]
COM
COM 3: write track T[w,x[w]]
COM
w3_strt_wr_T00, [w3,w=0,x0=0,enable],[writing_00,w=0,x0=0,a,enable]
w3_end_wr_T00,  [writing_00,a,enable],[f_to_pa,wrote_00a]
w3_strt_wr_T01, [w3,w=0,x0=1,enable],[writing_01,w=0,x0=1,a,enable]
w3_end_wr_T01,  [writing_01,a,enable],[f_to_pa,wrote_01a]
w3_strt_wr_T10, [w3,w=1,x1=0,enable],[writing_10,w=1,x1=0,a,enable]
w3_end_wr_T10,  [writing_10,a,enable],[f_to_pa,wrote_10a]
w3_strt_wr_T11, [w3,w=1,x1=1,enable],[writing_11,w=1,x1=1,a,enable]
w3_end_wr_T11,  [writing_11,a,enable],[f_to_pa,wrote_11a]
nftopa,[f_to_pa,no_w_fresh],[w_to_fa,no_w_fresh]
ftop00a,[w_fresh_00,no_w_prev,f_to_pa],[no_w_fresh,w_prev_00,w_to_fa]
ftop01a,[w_fresh_01,no_w_prev,f_to_pa],[no_w_fresh,w_prev_01,w_to_fa]
ftop10a,[w_fresh_10,no_w_prev,f_to_pa],[no_w_fresh,w_prev_10,w_to_fa]
ftop11a,[w_fresh_11,no_w_prev,f_to_pa],[no_w_fresh,w_prev_11,w_to_fa]
wtofv00ba,[wrote_00a,valid_00,w_to_fa,no_w_fresh],[w4,enable,w_fresh_00,
valid_00]
wtofv00ca,[wrote_00a,not_valid_00,w_to_fa,no_w_fresh],[w4,enable,
w_fresh_00,valid_00]
wtofv01ba,[wrote_01a,valid_01,w_to_fa,no_w_fresh],[w4,enable,w_fresh_01,
valid_01]
wtofv01ca,[wrote_01a,not_valid_01,w_to_fa,no_w_fresh],[w4,enable,
w_fresh_01,valid_01]
wtofv10ba,[wrote_10a,valid_10,w_to_fa,no_w_fresh],[w4,enable,w_fresh_10,
valid_10]
wtofv10ca,[wrote_10a,not_valid_10,w_to_fa,no_w_fresh],[w4,enable,w_fresh_10,
valid_10]
wtofv11ba,[wrote_11a,valid_11,w_to_fa,no_w_fresh],[w4,enable,w_fresh_11,
```

```
valid_11]
wtofv11ca,[wrote_11a,not_valid_11,w_to_fa,no_w_fresh],[w4,enable,w_fresh_11,
valid_11]
COM
COM 4: change W[1-w] (change W[not w])
COM
w4a,[w4,w=0,W1=0,enable],[w4_post,w=0,W1=1]
w4b,[w4,w=0,W1=1,enable],[w4_post,w=0,W1=0]
w4c,[w4,w=1,W0=0,enable],[w4_post,w=1,W0=1]
w4d,[w4,w=1,W0=1,enable],[w4_post,w=1,W0=0]
w1_initiala,[w4_post,no_w1],[clear_w_preva,w1+]
w1_normala,[w4_post,w1+],[clear_w_preva,w1+]
ncpa,[no_w_prev,clear_w_preva],[w1,enable,no_w_prev]
cp00a,[w_prev_00,clear_w_preva],[w1,enable,no_w_prev]
cp01a,[w_prev_01,clear_w_preva],[w1,enable,no_w_prev]
cp10a,[w_prev_10,clear_w_preva],[w1,enable,no_w_prev]
cp11a,[w_prev_11,clear_w_preva],[w1,enable,no_w_prev]
COM
COM 5: else
COM
COM 6: x[w] := 1 - x[w] (x[w] := not x[w])
COM
w6a,[w6,w=0,x0=0,enable],[w7,w=0,x0=1,enable]
w6b,[w6,w=0,x0=1,enable],[w7,w=0,x0=0,enable]
w6c,[w6,w=1,x1=0,enable],[w7,w=1,x1=1,enable]
w6d,[w6,w=1,x1=1,enable],[w7,w=1,x1=0,enable]
COM
COM 7: write track T[w,x[w]]
COM
w7_strt_wr_T00, [w7,w=0,x0=0,enable],[writing_00,w=0,x0=0,b,enable]
w7_end_wr_T00,  [writing_00,b,enable],[f_to_pb,wrote_00b]
w7_strt_wr_T01, [w7,w=0,x0=1,enable],[writing_01,w=0,x0=1,b,enable]
w7_end_wr_T01,  [writing_01,b,enable],[f_to_pb,wrote_01b]
w7_strt_wr_T10, [w7,w=1,x1=0,enable],[writing_10,w=1,x1=0,b,enable]
w7_end_wr_T10,  [writing_10,b,enable],[f_to_pb,wrote_10b]
w7_strt_wr_T11, [w7,w=1,x1=1,enable],[writing_11,w=1,x1=1,b,enable]
w7_end_wr_T11,  [writing_11,b,enable],[f_to_pb,wrote_11b]
nftop,[f_to_pb,no_w_fresh],[w_to_fb,no_w_fresh]
ftop00,[w_fresh_00,no_w_prev,f_to_pb],[no_w_fresh,w_prev_00,w_to_fb]
ftop01,[w_fresh_01,no_w_prev,f_to_pb],[no_w_fresh,w_prev_01,w_to_fb]
ftop10,[w_fresh_10,no_w_prev,f_to_pb],[no_w_fresh,w_prev_10,w_to_fb]
ftop11,[w_fresh_11,no_w_prev,f_to_pb],[no_w_fresh,w_prev_11,w_to_fb]
wtofv00b,[wrote_00b,valid_00,w_to_fb,no_w_fresh],[w8,enable,w_fresh_00,
valid_00]
wtofv00c,[wrote_00b,not_valid_00,w_to_fb,no_w_fresh],[w8,enable,w_fresh_00,
valid_00]
wtofv01b,[wrote_01b,valid_01,w_to_fb,no_w_fresh],[w8,enable,w_fresh_01,
valid_01]
wtofv01c,[wrote_01b,not_valid_01,w_to_fb,no_w_fresh],[w8,enable,w_fresh_01,
valid_01]
wtofv10b,[wrote_10b,valid_10,w_to_fb,no_w_fresh],[w8,enable,w_fresh_10,
valid_10]
wtofv10c,[wrote_10b,not_valid_10,w_to_fb,no_w_fresh],[w8,enable,w_fresh_10,
valid_10]
wtofv11b,[wrote_11b,valid_11,w_to_fb,no_w_fresh],[w8,enable,w_fresh_11,
valid_11]
wtofv11c,[wrote_11b,not_valid_11,w_to_fb,no_w_fresh],[w8,enable,w_fresh_11,
valid_11]
COM
COM 8: change D[w,1-x[w]] (change D[w, not x[w]])
COM
w8a,[w8,w=0,x0=1,D00=0,enable],[w8_post,w=0,x0=1,D00=1]
w8b,[w8,w=0,x0=1,D00=1,enable],[w8_post,w=0,x0=1,D00=0]
w8c,[w8,w=0,x0=0,D01=0,enable],[w8_post,w=0,x0=0,D01=1]
w8d,[w8,w=0,x0=0,D01=1,enable],[w8_post,w=0,x0=0,D01=0]
w8e,[w8,w=1,x1=1,D10=0,enable],[w8_post,w=1,x1=1,D10=1]
```

```
w8f,[w8,w=1,x1=1,D10=1,enable],[w8_post,w=1,x1=1,D10=0]
w8g,[w8,w=1,x1=0,D11=0,enable],[w8_post,w=1,x1=0,D11=1]
w8h,[w8,w=1,x1=0,D11=1,enable],[w8_post,w=1,x1=0,D11=0]
w1_initialb,[w8_post,no_w1],[clear_w_prevb,w1+]
w1_normalb,[w8_post,w1+],[clear_w_prevb,w1+]
ncpb,[no_w_prev,clear_w_prevb],[w1,enable,no_w_prev]
cp00b,[w_prev_00,clear_w_prevb],[w1,enable,no_w_prev]
cp01b,[w_prev_01,clear_w_prevb],[w1,enable,no_w_prev]
cp10b,[w_prev_10,clear_w_prevb],[w1,enable,no_w_prev]
cp11b,[w_prev_11,clear_w_prevb],[w1,enable,no_w_prev]
COM
COM 9: end if
COM
COM reader
COM
COM 1: if W[r] <> R[r] then
COM
clear_valid_00a,[not_valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_00b,[valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_01a,[not_valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_01b,[valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_10a,[not_valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_10b,[valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_11a,[not_valid_11,clear_valid4],[not_valid_11,clear_r_f]
clear_valid_11b,[valid_11,clear_valid4],[not_valid_11,clear_r_f]
no_rf,[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,clear_r_f],
[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,wf_to_rf]
clear_rf00,[r_fresh_00,clear_r_f],[not_r_fresh_00,wf_to_rf]
clear_rf01,[r_fresh_01,clear_r_f],[not_r_fresh_01,wf_to_rf]
clear_rf10,[r_fresh_10,clear_r_f],[not_r_fresh_10,wf_to_rf]
clear_rf11,[r_fresh_11,clear_r_f],[not_r_fresh_11,wf_to_rf]
no_wf,[no_w_fresh,wf_to_rf,not_valid_00,not_valid_01,not_valid_10,
not_valid_11],[no_w_fresh,r0_pre,valid_00,valid_01,valid_10,valid_11]
wf_to_rf_00,[w_fresh_00,not_r_fresh_00,wf_to_rf],[w_fresh_00,r_fresh_00,
clear_r_p]
wf_to_rf_01,[w_fresh_01,not_r_fresh_01,wf_to_rf],[w_fresh_01,r_fresh_01,
clear_r_p]
wf_to_rf_10,[w_fresh_10,not_r_fresh_10,wf_to_rf],[w_fresh_10,r_fresh_10,
clear_r_p]
wf_to_rf_11,[w_fresh_11,not_r_fresh_11,wf_to_rf],[w_fresh_11,r_fresh_11,
clear_r_p]
no_rp,[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,clear_r_p],
[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,wp_to_rp]
clear_rp00,[r_prev_00,clear_r_p],[not_r_prev_00,wp_to_rp]
clear_rp01,[r_prev_01,clear_r_p],[not_r_prev_01,wp_to_rp]
clear_rp10,[r_prev_10,clear_r_p],[not_r_prev_10,wp_to_rp]
clear_rp11,[r_prev_11,clear_r_p],[not_r_prev_11,wp_to_rp]
no_wp,[no_w_prev,wp_to_rp],[no_w_prev,r0_pre]
wp_to_rp_00,[w_prev_00,not_r_prev_00,wp_to_rp],[w_prev_00,r_prev_00,r0_pre]
wp_to_rp_01,[w_prev_01,not_r_prev_01,wp_to_rp],[w_prev_01,r_prev_01,r0_pre]
wp_to_rp_10,[w_prev_10,not_r_prev_10,wp_to_rp],[w_prev_10,r_prev_10,r0_pre]
wp_to_rp_11,[w_prev_11,not_r_prev_11,wp_to_rp],[w_prev_11,r_prev_11,r0_pre]
r0_init0,[r0_pre,no_w1],[r1,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r1,w1+,r0+]
r0_normal,[r0_pre,r0+],[r1,r0+]
r1a,[r1,r=0,W0=0,R0=1],[r2,r=0,W0=0,R0=1,enable]
r1b,[r1,r=0,W0=1,R0=0],[r2,r=0,W0=1,R0=0,enable]
r1c,[r1,r=0,W0=0,R0=0],[r7,r=0,W0=0,R0=0,enable]
r1d,[r1,r=0,W0=1,R0=1],[r7,r=0,W0=1,R0=1,enable]
r1e,[r1,r=1,W1=0,R1=1],[r2,r=1,W1=0,R1=1,enable]
r1f,[r1,r=1,W1=1,R1=0],[r2,r=1,W1=1,R1=0,enable]
r1g,[r1,r=1,W1=0,R1=0],[r7,r=1,W1=0,R1=0,enable]
r1h,[r1,r=1,W1=1,R1=1],[r7,r=1,W1=1,R1=1,enable]
COM
COM 2: change R[r]
COM
```

```
r2a,[r2,r=0,R0=0,enable],[r3,r=0,R0=1,enable]
r2b,[r2,r=0,R0=1,enable],[r3,r=0,R0=0,enable]
r2c,[r2,r=1,R1=0,enable],[r3,r=1,R1=1,enable]
r2d,[r2,r=1,R1=1,enable],[r3,r=1,R1=0,enable]
COM
COM 3: r := 1-r (r := not r)
COM
r3a,[r3,r=0,enable],[r4,r=1,enable]
r3b,[r3,r=1,enable],[r4,r=0,enable]
COM
COM 4: read d[0] := D[r,0]
COM
r4a,[r4,r=0,D00=0,+0d0=0,+0d0=1,enable],[r5,r=0,D00=0,d0=0,enable]
r4b,[r4,r=0,D00=1,+0d0=0,+0d0=1,enable],[r5,r=0,D00=1,d0=1,enable]
r4c,[r4,r=1,D10=0,+0d0=0,+0d0=1,enable],[r5,r=1,D10=0,d0=0,enable]
r4d,[r4,r=1,D10=1,+0d0=0,+0d0=1,enable],[r5,r=1,D10=1,d0=1,enable]
COM
COM 5: read d[1] := D[r,1]
COM
r5a,[r5,r=0,D01=0,+0d1=0,+0d1=1,enable],[pre_read_ctrl,r=0,D01=0,d1=0,enable]
r5b,[r5,r=0,D01=1,+0d1=0,+0d1=1,enable],[pre_read_ctrl,r=0,D01=1,d1=1,enable]
r5c,[r5,r=1,D11=0,+0d1=0,+0d1=1,enable],[pre_read_ctrl,r=1,D11=0,d1=0,enable]
r5d,[r5,r=1,D11=1,+0d1=0,+0d1=1,enable],[pre_read_ctrl,r=1,D11=1,d1=1,enable]
COM
COM 6: else
COM
COM 7: read d[d[0] (xor) d[1]] := D[r,d[0] (xor) d[1]]
COM
r7a,[r7,d0=0,d1=0,r=0,D00=0,enable],[pre_read_ctrl,d0=0,d1=0,r=0,D00=0,enable]
r7b,[r7,d0=0,d1=0,r=0,D00=1,enable],[pre_read_ctrl,d0=1,d1=0,r=0,D00=1,enable]
r7c,[r7,d0=0,d1=0,r=1,D10=0,enable],[pre_read_ctrl,d0=0,d1=0,r=1,D10=0,enable]
r7d,[r7,d0=0,d1=0,r=1,D10=1,enable],[pre_read_ctrl,d0=1,d1=0,r=1,D10=1,enable]
r7e,[r7,d0=1,d1=1,r=0,D00=0,enable],[pre_read_ctrl,d0=0,d1=1,r=0,D00=0,enable]
r7f,[r7,d0=1,d1=1,r=0,D00=1,enable],[pre_read_ctrl,d0=1,d1=1,r=0,D00=1,enable]
r7g,[r7,d0=1,d1=1,r=1,D10=0,enable],[pre_read_ctrl,d0=0,d1=1,r=1,D10=0,enable]
r7h,[r7,d0=1,d1=1,r=1,D10=1,enable],[pre_read_ctrl,d0=1,d1=1,r=1,D10=1,enable]
r7i,[r7,d0=0,d1=1,r=0,D01=0,enable],[pre_read_ctrl,d0=0,d1=0,r=0,D01=0,enable]
r7j,[r7,d0=0,d1=1,r=0,D01=1,enable],[pre_read_ctrl,d0=0,d1=1,r=0,D01=1,enable]
r7k,[r7,d0=0,d1=1,r=1,D11=0,enable],[pre_read_ctrl,d0=0,d1=0,r=1,D11=0,enable]
r7l,[r7,d0=0,d1=1,r=1,D11=1,enable],[pre_read_ctrl,d0=0,d1=1,r=1,D11=1,enable]
r7m,[r7,d0=1,d1=0,r=0,D01=0,enable],[pre_read_ctrl,d0=1,d1=0,r=0,D01=0,enable]
r7n,[r7,d0=1,d1=0,r=0,D01=1,enable],[pre_read_ctrl,d0=1,d1=1,r=0,D01=1,enable]
r7o,[r7,d0=1,d1=0,r=1,D11=0,enable],[pre_read_ctrl,d0=1,d1=0,r=1,D11=0,enable]
r7p,[r7,d0=1,d1=0,r=1,D11=1,enable],[pre_read_ctrl,d0=1,d1=1,r=1,D11=1,enable]
COM
COM 8: endif
COM
COM 9: read track T[r,d[0] (xor) d[1]]
COM
pre_read_00a,[pre_read_ctrl,enable,r=0,d0=0,d1=0,no_w_fresh],[no_w_fresh,
r9,r=0,d0=0,d1=0]
pre_read_00b,[pre_read_ctrl,enable,r=0,d0=0,d1=0,w_fresh_00],[w_fresh_00,
clear_prev_by_r,r=0,d0=0,d1=0]
pre_read_00c,[pre_read_ctrl,enable,r=0,d0=0,d1=0,w_fresh_01],[w_fresh_01,
r9,r=0,d0=0,d1=0]
pre_read_00d,[pre_read_ctrl,enable,r=0,d0=0,d1=0,w_fresh_10],[w_fresh_10,r9,
r=0,d0=0,d1=0]
pre_read_00e,[pre_read_ctrl,enable,r=0,d0=0,d1=0,w_fresh_11],[w_fresh_11,r9,
r=0,d0=0,d1=0]
pre_read_00f,[pre_read_ctrl,enable,r=0,d0=1,d1=1,no_w_fresh],[no_w_fresh,r9,
r=0,d0=1,d1=1]
pre_read_00g,[pre_read_ctrl,enable,r=0,d0=1,d1=1,w_fresh_00],[w_fresh_00,
clear_prev_by_r,r=0,d0=1,d1=1]
pre_read_00h,[pre_read_ctrl,enable,r=0,d0=1,d1=1,w_fresh_01],[w_fresh_01,r9,
r=0,d0=1,d1=1]
pre_read_00i,[pre_read_ctrl,enable,r=0,d0=1,d1=1,w_fresh_10],[w_fresh_10,r9,
```

```
r=0,d0=1,d1=1]
pre_read_00j,[pre_read_ctrl,enable,r=0,d0=1,d1=1,w_fresh_11],[w_fresh_11,r9,
r=0,d0=1,d1=1]
pre_read_01a,[pre_read_ctrl,enable,r=0,d0=0,d1=1,no_w_fresh],[no_w_fresh,r9,
r=0,d0=0,d1=1]
pre_read_01b,[pre_read_ctrl,enable,r=0,d0=0,d1=1,w_fresh_00],[w_fresh_00,r9,
r=0,d0=0,d1=1]
pre_read_01c,[pre_read_ctrl,enable,r=0,d0=0,d1=1,w_fresh_01],[w_fresh_01,
clear_prev_by_r,r=0,d0=0,d1=1]
pre_read_01d,[pre_read_ctrl,enable,r=0,d0=0,d1=1,w_fresh_10],[w_fresh_10,r9,
r=0,d0=0,d1=1]
pre_read_01e,[pre_read_ctrl,enable,r=0,d0=0,d1=1,w_fresh_11],[w_fresh_11,r9,
r=0,d0=0,d1=1]
pre_read_01f,[pre_read_ctrl,enable,r=0,d0=1,d1=0,no_w_fresh],[no_w_fresh,r9,
r=0,d0=1,d1=0]
pre_read_01g,[pre_read_ctrl,enable,r=0,d0=1,d1=0,w_fresh_00],[w_fresh_00,r9,
r=0,d0=1,d1=0]
pre_read_01h,[pre_read_ctrl,enable,r=0,d0=1,d1=0,w_fresh_01],[w_fresh_01,
clear_prev_by_r,r=0,d0=1,d1=0]
pre_read_01i,[pre_read_ctrl,enable,r=0,d0=1,d1=0,w_fresh_10],[w_fresh_10,r9,
r=0,d0=1,d1=0]
pre_read_01j,[pre_read_ctrl,enable,r=0,d0=1,d1=0,w_fresh_11],[w_fresh_11,r9,
r=0,d0=1,d1=0]
pre_read_10a,[pre_read_ctrl,enable,r=1,d0=0,d1=0,no_w_fresh],[no_w_fresh,r9,
r=1,d0=0,d1=0]
pre_read_10b,[pre_read_ctrl,enable,r=1,d0=0,d1=0,w_fresh_00],[w_fresh_00,r9,
r=1,d0=0,d1=0]
pre_read_10c,[pre_read_ctrl,enable,r=1,d0=0,d1=0,w_fresh_01],[w_fresh_01,r9,
r=1,d0=0,d1=0]
pre_read_10d,[pre_read_ctrl,enable,r=1,d0=0,d1=0,w_fresh_10],[w_fresh_10,
clear_prev_by_r,r=1,d0=0,d1=0]
pre_read_10e,[pre_read_ctrl,enable,r=1,d0=0,d1=0,w_fresh_11],[w_fresh_11,r9,
r=1,d0=0,d1=0]
pre_read_10f,[pre_read_ctrl,enable,r=1,d0=1,d1=1,no_w_fresh],[no_w_fresh,r9,
r=1,d0=1,d1=1]
pre_read_10g,[pre_read_ctrl,enable,r=1,d0=1,d1=1,w_fresh_00],[w_fresh_00,r9,
r=1,d0=1,d1=1]
pre_read_10h,[pre_read_ctrl,enable,r=1,d0=1,d1=1,w_fresh_01],[w_fresh_01,r9,
r=1,d0=1,d1=1]
pre_read_10i,[pre_read_ctrl,enable,r=1,d0=1,d1=1,w_fresh_10],[w_fresh_10,
clear_prev_by_r,r=1,d0=1,d1=1]
pre_read_10j,[pre_read_ctrl,enable,r=1,d0=1,d1=1,w_fresh_11],[w_fresh_11,r9,
r=1,d0=1,d1=1]
pre_read_11a,[pre_read_ctrl,enable,r=1,d0=0,d1=1,no_w_fresh],[no_w_fresh,r9,
r=1,d0=0,d1=1]
pre_read_11b,[pre_read_ctrl,enable,r=1,d0=0,d1=1,w_fresh_00],[w_fresh_00,r9,
r=1,d0=0,d1=1]
pre_read_11c,[pre_read_ctrl,enable,r=1,d0=0,d1=1,w_fresh_01],[w_fresh_01,r9,
r=1,d0=0,d1=1]
pre_read_11d,[pre_read_ctrl,enable,r=1,d0=0,d1=1,w_fresh_10],[w_fresh_10,r9,
r=1,d0=0,d1=1]
pre_read_11e,[pre_read_ctrl,enable,r=1,d0=0,d1=1,w_fresh_11],[w_fresh_11,
clear_prev_by_r,r=1,d0=0,d1=1]
pre_read_11f,[pre_read_ctrl,enable,r=1,d0=1,d1=0,no_w_fresh],[no_w_fresh,r9,
r=1,d0=1,d1=0]
pre_read_11g,[pre_read_ctrl,enable,r=1,d0=1,d1=0,w_fresh_00],[w_fresh_00,r9,
r=1,d0=1,d1=0]
pre_read_11h,[pre_read_ctrl,enable,r=1,d0=1,d1=0,w_fresh_01],[w_fresh_01,r9,
r=1,d0=1,d1=0]
pre_read_11i,[pre_read_ctrl,enable,r=1,d0=1,d1=0,w_fresh_10],[w_fresh_10,r9,
r=1,d0=1,d1=0]
pre_read_11j,[pre_read_ctrl,enable,r=1,d0=1,d1=0,w_fresh_11],[w_fresh_11,
clear_prev_by_r,r=1,d0=1,d1=0]
ncp_r,[no_w_prev,clear_prev_by_r],[r9,no_w_prev]
cp00_r,[w_prev_00,clear_prev_by_r],[r9,no_w_prev]
cp01_r,[w_prev_01,clear_prev_by_r],[r9,no_w_prev]
```

```
cp10_r,[w_prev_10,clear_prev_by_r],[r9,no_w_prev]
cp11_r,[w_prev_11,clear_prev_by_r],[r9,no_w_prev]
r9_strt_rd_T00a,[r9,r=0,d0=0,d1=0],[reading_00,r=0,d0=0,d1=0,enable]
r9_strt_rd_T00b,[r9,r=0,d0=1,d1=1],[reading_00,r=0,d0=1,d1=1,enable]
r9_end_rd_T00,[reading_00,enable],[clear_valid,enable]
r9_strt_rd_T01a,[r9,r=0,d0=0,d1=1],[reading_01,r=0,d0=0,d1=1,enable]
r9_strt_rd_T01b,[r9,r=0,d0=1,d1=0],[reading_01,r=0,d0=1,d1=0,enable]
r9_end_rd_T01,[reading_01,enable],[clear_valid,enable]
r9_strt_rd_T10a,[r9,r=1,d0=0,d1=0],[reading_10,r=1,d0=0,d1=0,enable]
r9_strt_rd_T10b,[r9,r=1,d0=1,d1=1],[reading_10,r=1,d0=1,d1=1,enable]
r9_end_rd_T10,[reading_10,enable],[clear_valid,enable]
r9_strt_rd_T11a,[r9,r=1,d0=0,d1=1],[reading_11,r=1,d0=0,d1=1,enable]
r9_strt_rd_T11b,[r9,r=1,d0=1,d1=0],[reading_11,r=1,d0=1,d1=0,enable]
r9_end_rd_T11,[reading_11,enable],[clear_valid,enable]
COM
COM check freshness
COM
not_fresh_00,[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,
r0+],[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,
freshness_lost,r0+]
not_fresh_01,[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,
r0+],[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,
freshness_lost,r0+]
not_fresh_10,[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,
r0+],[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,
freshness_lost,r0+]
not_fresh_11,[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,
r0+],[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,
freshness_lost,r0+]
END
```

B.3.3 Model for testing sequencing

```
Tromp 4b+8 4 track, sequence.
bad=[order_lost]
initial=[enable,w1,w=0,r=0,R0=0,R1=0,W0=0,W1=0,x0=0,x1=0,D00=0,D01=0,D10=0,
D11=0,d0=0,d1=0,clear_x,no_w1,no_r0,Xb_00,Xb_01,Xb_10,Xb_11,x_00,x_01,x_10,
x_11,4_00,4_01,4_10,4_11,not1,not2,not3]
COM
COM writer
COM
COM 1: if R[1-w] == W[1-w] then ( if R[not w] == W[not w] then )
COM
w1a,[w1,w=0,R1=0,W1=0,enable],[w2,w=0,R1=0,W1=0,enable]
w1b,[w1,w=0,R1=1,W1=1,enable],[w2,w=0,R1=1,W1=1,enable]
w1c,[w1,w=1,R0=0,W0=0,enable],[w2,w=1,R0=0,W0=0,enable]
w1d,[w1,w=1,R0=1,W0=1,enable],[w2,w=1,R0=1,W0=1,enable]
w1e,[w1,w=0,R1=0,W1=1,enable],[w6,w=0,R1=0,W1=1,enable]
w1f,[w1,w=0,R1=1,W1=0,enable],[w6,w=0,R1=1,W1=0,enable]
w1g,[w1,w=1,R0=0,W0=1,enable],[w6,w=1,R0=0,W0=1,enable]
w1h,[w1,w=1,R0=1,W0=0,enable],[w6,w=1,R0=1,W0=0,enable]
COM
COM 2: w := 1-w (w := not w)
COM
w2a,[w2,w=0,enable],[w3,w=1,enable]
w2b,[w2,w=1,enable],[w3,w=0,enable]
COM
COM 3: write track T[w,x[w]]
COM
w3_strt_wr_T00, [w3,w=0,x0=0,enable],[writing_00,w=0,x0=0,a,enable]
w3_end_wr_T00,  [writing_00,a,enable],[wip_roles00a]
w3_strt_wr_T01, [w3,w=0,x0=1,enable],[writing_01,w=0,x0=1,a,enable]
w3_end_wr_T01,  [writing_01,a,enable],[wip_roles01a]
w3_strt_wr_T10, [w3,w=1,x1=0,enable],[writing_10,w=1,x1=0,a,enable]
```

```
w3_end_wr_T10,  [writing_10,a,enable],[wip_roles10a]
w3_strt_wr_T11, [w3,w=1,x1=1,enable],[writing_11,w=1,x1=1,a,enable]
w3_end_wr_T11,  [writing_11,a,enable],[wip_roles11a]
wip1a,[wip_roles00a,X_00],[wip_role200a,Xb_00]
wip1b,[wip_roles00a,Xb_00],[wip_role200a,Xb_00]
wip1c,[wip_roles01a,X_01],[wip_role201a,Xb_01]
wip1d,[wip_roles01a,Xb_01],[wip_role201a,Xb_01]
wip1e,[wip_roles10a,X_10],[wip_role210a,Xb_10]
wip1f,[wip_roles10a,Xb_10],[wip_role210a,Xb_10]
wip1g,[wip_roles11a,X_11],[wip_role211a,Xb_11]
wip1h,[wip_roles11a,Xb_11],[wip_role211a,Xb_11]
wip2a,[wip_role200a,1_00],[1_00,w4,enable]
wip2b,[wip_role201a,1_01],[1_01,w4,enable]
wip2c,[wip_role210a,1_10],[1_10,w4,enable]
wip2d,[wip_role211a,1_11],[1_11,w4,enable]
wip2e,[wip_role200a,2_00],[w2b1_00a,11b2a,2_00]
wip2f,[wip_role201a,2_01],[w2b1_01a,11b2a,2_01]
wip2g,[wip_role210a,2_10],[w2b1_10a,11b2a,2_10]
wip2h,[wip_role211a,2_11],[w2b1_11a,11b2a,2_11]
wip2i,[wip_role200a,3_00],[w3b1_00a,22b3a,3_00]
wip2j,[wip_role201a,3_01],[w3b1_01a,22b3a,3_01]
wip2k,[wip_role210a,3_10],[w3b1_10a,22b3a,3_10]
wip2l,[wip_role211a,3_11],[w3b1_11a,22b3a,3_11]
wip2m,[wip_role200a,4_00],[w4b1_00a,33b4a,4_00]
wip2n,[wip_role201a,4_01],[w4b1_01a,33b4a,4_01]
wip2o,[wip_role210a,4_10],[w4b1_10a,33b4a,4_10]
wip2p,[wip_role211a,4_11],[w4b1_11a,33b4a,4_11]
wip3a,[33b4a,3_00],[22b3a,4_00]
wip3b,[33b4a,3_01],[22b3a,4_01]
wip3c,[33b4a,3_10],[22b3a,4_10]
wip3d,[33b4a,3_11],[22b3a,4_11]
wip3e,[33b4a,not3],[22b3a,not3]
wip4a,[22b3a,2_00,not3],[11b2a,3_00,not3b]
wip4aa,[22b3a,2_00,not3b],[11b2a,3_00,not3b]
wip4b,[22b3a,2_01,not3],[11b2a,3_01,not3b]
wip4ba,[22b3a,2_01,not3b],[11b2a,3_01,not3b]
wip4c,[22b3a,2_10,not3],[11b2a,3_10,not3b]
wip4ca,[22b3a,2_10,not3b],[11b2a,3_10,not3b]
wip4d,[22b3a,2_11,not3],[11b2a,3_11,not3b]
wip4da,[22b3a,2_11,not3b],[11b2a,3_11,not3b]
wip4e,[22b3a,not2],[11b2a,not2]
wip5a,[11b2a,1_00,not2],[11b2_fina,2_00,not2b]
wip5aa,[11b2a,1_00,not2b],[11b2_fina,2_00,not2b]
wip5b,[11b2a,1_01,not2],[11b2_fina,2_01,not2b]
wip5ba,[11b2a,1_01,not2b],[11b2_fina,2_01,not2b]
wip5c,[11b2a,1_10,not2],[11b2_fina,2_10,not2b]
wip5ca,[11b2a,1_10,not2b],[11b2_fina,2_10,not2b]
wip5d,[11b2a,1_11,not2],[11b2_fina,2_11,not2b]
wip5da,[11b2a,1_11,not2b],[11b2_fina,2_11,not2b]
wip5e,[11b2a,not1],[11b2_fina,not1]
wip6a,[11b2_fina,w2b1_00a,2_00],[1_00,w4,enable]
wip6b,[11b2_fina,w2b1_01a,2_01],[1_01,w4,enable]
wip6c,[11b2_fina,w2b1_10a,2_10],[1_10,w4,enable]
wip6d,[11b2_fina,w2b1_11a,2_11],[1_11,w4,enable]
wip6e,[11b2_fina,w3b1_00a,3_00],[1_00,w4,enable]
wip6f,[11b2_fina,w3b1_01a,3_01],[1_01,w4,enable]
wip6g,[11b2_fina,w3b1_10a,3_10],[1_10,w4,enable]
wip6h,[11b2_fina,w3b1_11a,3_11],[1_11,w4,enable]
wip6i,[11b2_fina,w4b1_00a,4_00,not1],[1_00,w4,enable,not1b]
wip6ia,[11b2_fina,w4b1_00a,4_00,not1b],[1_00,w4,enable,not1b]
wip6j,[11b2_fina,w4b1_01a,4_01,not1],[1_01,w4,enable,not1b]
wip6ja,[11b2_fina,w4b1_01a,4_01,not1b],[1_01,w4,enable,not1b]
wip6k,[11b2_fina,w4b1_10a,4_10,not1],[1_10,w4,enable,not1b]
wip6ka,[11b2_fina,w4b1_10a,4_10,not1b],[1_10,w4,enable,not1b]
wip6l,[11b2_fina,w4b1_11a,4_11,not1],[1_11,w4,enable,not1b]
wip6la,[11b2_fina,w4b1_11a,4_11,not1b],[1_11,w4,enable,not1b]
```

```
COM
COM 4: change W[1-w] (change W[not w])
COM
w4a,[w4,w=0,W1=0,enable],[w4_post,w=0,W1=1]
w4b,[w4,w=0,W1=1,enable],[w4_post,w=0,W1=0]
w4c,[w4,w=1,W0=0,enable],[w4_post,w=1,W0=1]
w4d,[w4,w=1,W0=1,enable],[w4_post,w=1,W0=0]
w1_initiala,[w4_post,no_w1],[w1,w1+,enable]
w1_normala,[w4_post,w1+],[w1,w1+,enable]
COM
COM 5: else
COM
COM 6: x[w] := 1 - x[w] (x[w] := not x[w])
COM
w6a,[w6,w=0,x0=0,enable],[w7,w=0,x0=1,enable]
w6b,[w6,w=0,x0=1,enable],[w7,w=0,x0=0,enable]
w6c,[w6,w=1,x1=0,enable],[w7,w=1,x1=1,enable]
w6d,[w6,w=1,x1=1,enable],[w7,w=1,x1=0,enable]
COM
COM 7: write track T[w,x[w]]
COM
w7_strt_wr_T00, [w7,w=0,x0=0,enable],[writing_00,w=0,x0=0,b,enable]
w7_end_wr_T00,  [writing_00,b,enable],[wip_roles00b]
w7_strt_wr_T01, [w7,w=0,x0=1,enable],[writing_01,w=0,x0=1,b,enable]
w7_end_wr_T01,  [writing_01,b,enable],[wip_roles01b]
w7_strt_wr_T10, [w7,w=1,x1=0,enable],[writing_10,w=1,x1=0,b,enable]
w7_end_wr_T10,  [writing_10,b,enable],[wip_roles10b]
w7_strt_wr_T11, [w7,w=1,x1=1,enable],[writing_11,w=1,x1=1,b,enable]
w7_end_wr_T11,  [writing_11,b,enable],[wip_roles11b]
wip1ab,[wip_roles00b,X_00],[wip_role200b,Xb_00]
wip1bb,[wip_roles00b,Xb_00],[wip_role200b,Xb_00]
wip1cb,[wip_roles01b,X_01],[wip_role201b,Xb_01]
wip1db,[wip_roles01b,Xb_01],[wip_role201b,Xb_01]
wip1eb,[wip_roles10b,X_10],[wip_role210b,Xb_10]
wip1fb,[wip_roles10b,Xb_10],[wip_role210b,Xb_10]
wip1gb,[wip_roles11b,X_11],[wip_role211b,Xb_11]
wip1hb,[wip_roles11b,Xb_11],[wip_role211b,Xb_11]
wip2ab,[wip_role200b,1_00],[1_00,w8,enable]
wip2bb,[wip_role201b,1_01],[1_01,w8,enable]
wip2cb,[wip_role210b,1_10],[1_10,w8,enable]
wip2db,[wip_role211b,1_11],[1_11,w8,enable]
wip2eb,[wip_role200b,2_00],[w2b1_00b,11b2b,2_00]
wip2fb,[wip_role201b,2_01],[w2b1_01b,11b2b,2_01]
wip2gb,[wip_role210b,2_10],[w2b1_10b,11b2b,2_10]
wip2hb,[wip_role211b,2_11],[w2b1_11b,11b2b,2_11]
wip2ib,[wip_role200b,3_00],[w3b1_00b,22b3b,3_00]
wip2jb,[wip_role201b,3_01],[w3b1_01b,22b3b,3_01]
wip2kb,[wip_role210b,3_10],[w3b1_10b,22b3b,3_10]
wip2lb,[wip_role211b,3_11],[w3b1_11b,22b3b,3_11]
wip2mb,[wip_role200b,4_00],[w4b1_00b,33b4b,4_00]
wip2nb,[wip_role201b,4_01],[w4b1_01b,33b4b,4_01]
wip2ob,[wip_role210b,4_10],[w4b1_10b,33b4b,4_10]
wip2pb,[wip_role211b,4_11],[w4b1_11b,33b4b,4_11]
wip3ab,[33b4b,3_00],[22b3b,4_00]
wip3bb,[33b4b,3_01],[22b3b,4_01]
wip3cb,[33b4b,3_10],[22b3b,4_10]
wip3db,[33b4b,3_11],[22b3b,4_11]
wip3eb,[33b4b,not3],[22b3b,not3]
wip4ab,[22b3b,2_00,not3],[11b2b,3_00,not3b]
wip4aab,[22b3b,2_00,not3b],[11b2b,3_00,not3b]
wip4bb,[22b3b,2_01,not3],[11b2b,3_01,not3b]
wip4bab,[22b3b,2_01,not3b],[11b2b,3_01,not3b]
wip4cb,[22b3b,2_10,not3],[11b2b,3_10,not3b]
wip4cab,[22b3b,2_10,not3b],[11b2b,3_10,not3b]
wip4db,[22b3b,2_11,not3],[11b2b,3_11,not3b]
wip4dab,[22b3b,2_11,not3b],[11b2b,3_11,not3b]
```

```
wip4eb,[22b3b,not2],[11b2b,not2]
wip5ab,[11b2b,1_00,not2],[11b2_finb,2_00,not2b]
wip5aab,[11b2b,1_00,not2b],[11b2_finb,2_00,not2b]
wip5bb,[11b2b,1_01,not2],[11b2_finb,2_01,not2b]
wip5bab,[11b2b,1_01,not2b],[11b2_finb,2_01,not2b]
wip5cb,[11b2b,1_10,not2],[11b2_finb,2_10,not2b]
wip5cab,[11b2b,1_10,not2b],[11b2_finb,2_10,not2b]
wip5db,[11b2b,1_11,not2],[11b2_finb,2_11,not2b]
wip5dab,[11b2b,1_11,not2b],[11b2_finb,2_11,not2b]
wip5eb,[11b2b,not1],[11b2_finb,not1]
wip6ab,[11b2_finb,w2b1_00b,2_00],[1_00,w8,enable]
wip6bb,[11b2_finb,w2b1_01b,2_01],[1_01,w8,enable]
wip6cb,[11b2_finb,w2b1_10b,2_10],[1_10,w8,enable]
wip6db,[11b2_finb,w2b1_11b,2_11],[1_11,w8,enable]
wip6eb,[11b2_finb,w3b1_00b,3_00],[1_00,w8,enable]
wip6fb,[11b2_finb,w3b1_01b,3_01],[1_01,w8,enable]
wip6gb,[11b2_finb,w3b1_10b,3_10],[1_10,w8,enable]
wip6hb,[11b2_finb,w3b1_11b,3_11],[1_11,w8,enable]
wip6ib,[11b2_finb,w4b1_00b,4_00,not1],[1_00,w8,enable,not1b]
wip6iab,[11b2_finb,w4b1_00b,4_00,not1b],[1_00,w8,enable,not1b]
wip6jb,[11b2_finb,w4b1_01b,4_01,not1],[1_01,w8,enable,not1b]
wip6jab,[11b2_finb,w4b1_01b,4_01,not1b],[1_01,w8,enable,not1b]
wip6kb,[11b2_finb,w4b1_10b,4_10,not1],[1_10,w8,enable,not1b]
wip6kab,[11b2_finb,w4b1_10b,4_10,not1b],[1_10,w8,enable,not1b]
wip6lb,[11b2_finb,w4b1_11b,4_11,not1],[1_11,w8,enable,not1b]
wip6lab,[11b2_finb,w4b1_11b,4_11,not1b],[1_11,w8,enable,not1b]
COM
COM 8: change D[w,1-x[w]] (change D[w, not x[w]])
COM
w8a,[w8,w=0,x0=1,D00=0,enable],[w8_post,w=0,x0=1,D00=1]
w8b,[w8,w=0,x0=1,D00=1,enable],[w8_post,w=0,x0=1,D00=0]
w8c,[w8,w=0,x0=0,D01=0,enable],[w8_post,w=0,x0=0,D01=1]
w8d,[w8,w=0,x0=0,D01=1,enable],[w8_post,w=0,x0=0,D01=0]
w8e,[w8,w=1,x1=1,D10=0,enable],[w8_post,w=1,x1=1,D10=1]
w8f,[w8,w=1,x1=1,D10=1,enable],[w8_post,w=1,x1=1,D10=0]
w8g,[w8,w=1,x1=0,D11=0,enable],[w8_post,w=1,x1=0,D11=1]
w8h,[w8,w=1,x1=0,D11=1,enable],[w8_post,w=1,x1=0,D11=0]
w1_initialb,[w8_post,no_w1],[w1,w1+,enable]
w1_normalb,[w8_post,w1+],[w1,w1+,enable]
COM
COM 9: end if
COM
COM reader
COM
COM 1: if W[r] <> R[r] then
COM
clr_x00,[enable,clear_x,x_00],[clear_x2,xb_00]
clr_x00a,[enable,clear_x,xb_00],[clear_x2,xb_00]
clr_x01,[clear_x2,x_01],[clear_x3,xb_01]
clr_x01a,[clear_x2,xb_01],[clear_x3,xb_01]
clr_x02,[clear_x3,x_10],[clear_x4,xb_10]
clr_x02a,[clear_x3,xb_10],[clear_x4,xb_10]
clr_x03,[clear_x4,x_11],[r0_pre,xb_11]
clr_x03a,[clear_x4,xb_11],[r0_pre,xb_11]
r0_init0,[r0_pre,no_w1],[r1,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r1,w1+,r0+]
r0_normal,[r0_pre,r0+],[r1,r0+]
r1a,[r1,r=0,W0=0,R0=1],[r2,r=0,W0=0,R0=1,enable]
r1b,[r1,r=0,W0=1,R0=0],[r2,r=0,W0=1,R0=0,enable]
r1c,[r1,r=0,W0=0,R0=0],[r7,r=0,W0=0,R0=0,enable]
r1d,[r1,r=0,W0=1,R0=1],[r7,r=0,W0=1,R0=1,enable]
r1e,[r1,r=1,W1=0,R1=1],[r2,r=1,W1=0,R1=1,enable]
r1f,[r1,r=1,W1=1,R1=0],[r2,r=1,W1=1,R1=0,enable]
r1g,[r1,r=1,W1=0,R1=0],[r7,r=1,W1=0,R1=0,enable]
r1h,[r1,r=1,W1=1,R1=1],[r7,r=1,W1=1,R1=1,enable]
COM
```

```
COM 2: change R[r]
COM
r2a,[r2,r=0,R0=0,enable],[r3,r=0,R0=1,enable]
r2b,[r2,r=0,R0=1,enable],[r3,r=0,R0=0,enable]
r2c,[r2,r=1,R1=0,enable],[r3,r=1,R1=1,enable]
r2d,[r2,r=1,R1=1,enable],[r3,r=1,R1=0,enable]
COM
COM 3: r := 1-r (r := not r)
COM
r3a,[r3,r=0,enable],[r4,r=1,enable]
r3b,[r3,r=1,enable],[r4,r=0,enable]
COM
COM 4: read d[0] := D[r,0]
COM
r4a,[r4,r=0,D00=0,+0d0=0,+0d0=1,enable],[r5,r=0,D00=0,d0=0,enable]
r4b,[r4,r=0,D00=1,+0d0=0,+0d0=1,enable],[r5,r=0,D00=1,d0=1,enable]
r4c,[r4,r=1,D10=0,+0d0=0,+0d0=1,enable],[r5,r=1,D10=0,d0=0,enable]
r4d,[r4,r=1,D10=1,+0d0=0,+0d0=1,enable],[r5,r=1,D10=1,d0=1,enable]
COM
COM 5: read d[1] := D[r,1]
COM
r5a,[r5,r=0,D01=0,+0d1=0,+0d1=1,enable],[rcp_roles,r=0,D01=0,d1=0,enable]
r5b,[r5,r=0,D01=1,+0d1=0,+0d1=1,enable],[rcp_roles,r=0,D01=1,d1=1,enable]
r5c,[r5,r=1,D11=0,+0d1=0,+0d1=1,enable],[rcp_roles,r=1,D11=0,d1=0,enable]
r5d,[r5,r=1,D11=1,+0d1=0,+0d1=1,enable],[rcp_roles,r=1,D11=1,d1=1,enable]
COM
COM 6: else
COM
COM 7: read d[d[0] (xor) d[1]] := D[r,d[0] (xor) d[1]]
COM
r7a,[r7,d0=0,d1=0,r=0,D00=0,enable],[rcp_roles,d0=0,d1=0,r=0,D00=0,enable]
r7b,[r7,d0=0,d1=0,r=0,D00=1,enable],[rcp_roles,d0=1,d1=0,r=0,D00=1,enable]
r7c,[r7,d0=0,d1=0,r=1,D10=0,enable],[rcp_roles,d0=0,d1=0,r=1,D10=0,enable]
r7d,[r7,d0=0,d1=0,r=1,D10=1,enable],[rcp_roles,d0=1,d1=0,r=1,D10=1,enable]
r7e,[r7,d0=1,d1=1,r=0,D00=0,enable],[rcp_roles,d0=0,d1=1,r=0,D00=0,enable]
r7f,[r7,d0=1,d1=1,r=0,D00=1,enable],[rcp_roles,d0=1,d1=1,r=0,D00=1,enable]
r7g,[r7,d0=1,d1=1,r=1,D10=0,enable],[rcp_roles,d0=0,d1=1,r=1,D10=0,enable]
r7h,[r7,d0=1,d1=1,r=1,D10=1,enable],[rcp_roles,d0=1,d1=1,r=1,D10=1,enable]
r7i,[r7,d0=0,d1=1,r=0,D01=0,enable],[rcp_roles,d0=0,d1=0,r=0,D01=0,enable]
r7j,[r7,d0=0,d1=1,r=0,D01=1,enable],[rcp_roles,d0=0,d1=1,r=0,D01=1,enable]
r7k,[r7,d0=0,d1=1,r=1,D11=0,enable],[rcp_roles,d0=0,d1=0,r=1,D11=0,enable]
r7l,[r7,d0=0,d1=1,r=1,D11=1,enable],[rcp_roles,d0=0,d1=1,r=1,D11=1,enable]
r7m,[r7,d0=1,d1=0,r=0,D01=0,enable],[rcp_roles,d0=1,d1=0,r=0,D01=0,enable]
r7n,[r7,d0=1,d1=0,r=0,D01=1,enable],[rcp_roles,d0=1,d1=1,r=0,D01=1,enable]
r7o,[r7,d0=1,d1=0,r=1,D11=0,enable],[rcp_roles,d0=1,d1=0,r=1,D11=0,enable]
r7p,[r7,d0=1,d1=0,r=1,D11=1,enable],[rcp_roles,d0=1,d1=1,r=1,D11=1,enable]
COM
COM 8: endif
COM
COM 9: read track T[r,d[0] (xor) d[1]]
COM
x_r00a,[enable,rcp_roles,r=0,d0=0,d1=0,X_00,xb_00],[r=0,d0=0,d1=0,X_00,x_00,
rcp_roles2_00]
x_r00b,[enable,rcp_roles,r=0,d0=0,d1=0,Xb_00],[r=0,d0=0,d1=0,Xb_00,
rcp_roles2_00]
x_r00c,[enable,rcp_roles,r=0,d0=1,d1=1,X_00,xb_00],[r=0,d0=1,d1=1,X_00,x_00,
rcp_roles2_00]
x_r00d,[enable,rcp_roles,r=0,d0=1,d1=1,Xb_00],[r=0,d0=1,d1=1,Xb_00,
rcp_roles2_00]
x_r01a,[enable,rcp_roles,r=0,d0=0,d1=1,X_01,xb_01],[r=0,d0=0,d1=1,X_01,x_01,
rcp_roles2_01]
x_r01b,[enable,rcp_roles,r=0,d0=0,d1=1,Xb_01],[r=0,d0=0,d1=1,Xb_01,
rcp_roles2_01]
x_r01c,[enable,rcp_roles,r=0,d0=1,d1=0,X_01,xb_01],[r=0,d0=1,d1=0,X_01,x_01,
rcp_roles2_01]
x_r01d,[enable,rcp_roles,r=0,d0=1,d1=0,Xb_01],[r=0,d0=1,d1=0,Xb_01,
```

```
rcp_roles2_01]
x_r10a,[enable,rcp_roles,r=1,d0=0,d1=0,X_10,xb_10],[r=1,d0=0,d1=0,X_10,x_10,
rcp_roles2_10]
x_r10b,[enable,rcp_roles,r=1,d0=0,d1=0,Xb_10],[r=1,d0=0,d1=0,Xb_10,
rcp_roles2_10]
x_r10c,[enable,rcp_roles,r=1,d0=1,d1=1,X_10,xb_10],[r=1,d0=1,d1=1,X_10,x_10,
rcp_roles2_10]
x_r10d,[enable,rcp_roles,r=1,d0=1,d1=1,Xb_10],[r=1,d0=1,d1=1,Xb_10,
rcp_roles2_10]
x_r11a,[enable,rcp_roles,r=1,d0=0,d1=1,X_11,xb_11],[r=1,d0=0,d1=1,X_11,x_11,
rcp_roles2_11]
x_r11b,[enable,rcp_roles,r=1,d0=0,d1=1,Xb_11],[r=1,d0=0,d1=1,Xb_11,
rcp_roles2_11]
x_r11c,[enable,rcp_roles,r=1,d0=1,d1=0,X_11,xb_11],[r=1,d0=1,d1=0,X_11,x_11,
rcp_roles2_11]
x_r11d,[enable,rcp_roles,r=1,d0=1,d1=0,Xb_11],[r=1,d0=1,d1=0,Xb_11,
rcp_roles2_11]
X_r00a,[rcp_roles2_00,1_00],[1_00,rcp_Xb_00,rcp_X_01,rcp_X_10,rcp_X_11,
rcp_role3]
X_r00b,[rcp_roles2_00,2_00],[2_00,rcp_Xb_00,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r00c,[rcp_roles2_00,3_00],[3_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r00d,[rcp_roles2_00,4_00],[4_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r01a,[rcp_roles2_01,1_01],[1_01,rcp_Xb_01,rcp_X_00,rcp_X_10,rcp_X_11,
rcp_role3]
X_r01b,[rcp_roles2_01,2_01],[2_01,rcp_Xb_01,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r01c,[rcp_roles2_01,3_01],[3_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r01d,[rcp_roles2_01,4_01],[4_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r10a,[rcp_roles2_10,1_10],[1_10,rcp_Xb_10,rcp_X_01,rcp_X_00,rcp_X_11,
rcp_role3]
X_r10b,[rcp_roles2_10,2_10],[2_10,rcp_Xb_10,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r10c,[rcp_roles2_10,3_10],[3_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r10d,[rcp_roles2_10,4_10],[4_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r11a,[rcp_roles2_11,1_11],[1_11,rcp_Xb_11,rcp_X_01,rcp_X_10,rcp_X_00,
rcp_role3]
X_r11b,[rcp_roles2_11,2_11],[2_11,rcp_Xb_11,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r11c,[rcp_roles2_11,3_11],[3_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r11d,[rcp_roles2_11,4_11],[4_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_00,X_00],[Xb_00,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_00,Xb_00],[Xb_00,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_01,X_01],[Xb_01,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_01,Xb_01],[Xb_01,rcp_role4]
rcp_3e,[rcp_role3,rcp_Xb_10,X_10],[Xb_10,rcp_role4]
rcp_3f,[rcp_role3,rcp_Xb_10,Xb_10],[Xb_10,rcp_role4]
rcp_3g,[rcp_role3,rcp_Xb_11,X_11],[Xb_11,rcp_role4]
rcp_3h,[rcp_role3,rcp_Xb_11,Xb_11],[Xb_11,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_01,X_01],[X_01,rcp_role5]
rcp_4b,[rcp_role4,rcp_X_01,Xb_01],[X_01,rcp_role5]
rcp_4c,[rcp_role4,rcp_X_00,Xb_00],[X_00,rcp_role5]
rcp_4d,[rcp_role4,rcp_X_00,X_00],[X_00,rcp_role5]
rcp_4e,[rcp_role4,rcp_Xb_1,1_00,X_00],[1_00,Xb_00,rcp_role5]
rcp_4f,[rcp_role4,rcp_Xb_1,1_00,Xb_00],[1_00,Xb_00,rcp_role5]
rcp_4g,[rcp_role4,rcp_Xb_1,1_01,X_01],[1_01,Xb_01,rcp_role5]
rcp_4h,[rcp_role4,rcp_Xb_1,1_01,Xb_01],[1_01,Xb_01,rcp_role5]
rcp_4i,[rcp_role4,rcp_Xb_1,1_10,X_10],[1_10,Xb_10,rcp_role5]
```

```
rcp_4j,[rcp_role4,rcp_Xb_1,1_10,Xb_10],[1_10,Xb_10,rcp_role5]
rcp_4k,[rcp_role4,rcp_Xb_1,1_11,X_11],[1_11,Xb_11,rcp_role5]
rcp_4l,[rcp_role4,rcp_Xb_1,1_11,Xb_11],[1_11,Xb_11,rcp_role5]
rcp_5a,[rcp_role5,rcp_X_10,X_10],[X_10,rcp_role6]
rcp_5b,[rcp_role5,rcp_X_10,Xb_10],[X_10,rcp_role6]
rcp_5c,[rcp_role5,rcp_X_00,X_00],[X_00,rcp_role6]
rcp_5d,[rcp_role5,rcp_X_00,Xb_00],[X_00,rcp_role6]
rcp_5e,[rcp_role5,rcp_Xb_2,2_00,X_00],[2_00,Xb_00,rcp_role6]
rcp_5f,[rcp_role5,rcp_Xb_2,2_00,Xb_00],[2_00,Xb_00,rcp_role6]
rcp_5g,[rcp_role5,rcp_Xb_2,2_01,X_01],[2_01,Xb_01,rcp_role6]
rcp_5h,[rcp_role5,rcp_Xb_2,2_01,Xb_01],[2_01,Xb_01,rcp_role6]
rcp_5i,[rcp_role5,rcp_Xb_2,2_10,X_10],[2_10,Xb_10,rcp_role6]
rcp_5j,[rcp_role5,rcp_Xb_2,2_10,Xb_10],[2_10,Xb_10,rcp_role6]
rcp_5k,[rcp_role5,rcp_Xb_2,2_11,X_11],[2_11,Xb_11,rcp_role6]
rcp_5l,[rcp_role5,rcp_Xb_2,2_11,Xb_11],[2_11,Xb_11,rcp_role6]
rcp_5m,[rcp_role5,rcp_X_3,3_00,X_00],[3_00,X_00,rcp_role6]
rcp_5n,[rcp_role5,rcp_X_3,3_00,Xb_00],[3_00,X_00,rcp_role6]
rcp_5o,[rcp_role5,rcp_X_3,3_01,X_01],[3_01,X_01,rcp_role6]
rcp_5p,[rcp_role5,rcp_X_3,3_01,Xb_01],[3_01,X_01,rcp_role6]
rcp_5q,[rcp_role5,rcp_X_3,3_10,X_10],[3_10,X_10,rcp_role6]
rcp_5r,[rcp_role5,rcp_X_3,3_10,Xb_10],[3_10,X_10,rcp_role6]
rcp_5s,[rcp_role5,rcp_X_3,3_11,X_11],[3_11,X_11,rcp_role6]
rcp_5t,[rcp_role5,rcp_X_3,3_11,Xb_11],[3_11,X_11,rcp_role6]
rcp_6a,[rcp_role6,rcp_X_11,X_11],[X_11,r9]
rcp_6b,[rcp_role6,rcp_X_11,Xb_11],[X_11,r9]
rcp_6c,[rcp_role6,rcp_X_00,X_00],[X_00,r9]
rcp_6d,[rcp_role6,rcp_X_00,Xb_00],[X_00,r9]
rcp_6e,[rcp_role6,rcp_Xb_3,3_00,X_00],[3_00,Xb_00,r9]
rcp_6f,[rcp_role6,rcp_Xb_3,3_00,Xb_00],[3_00,Xb_00,r9]
rcp_6g,[rcp_role6,rcp_Xb_3,3_01,X_01],[3_01,Xb_01,r9]
rcp_6h,[rcp_role6,rcp_Xb_3,3_01,Xb_01],[3_01,Xb_01,r9]
rcp_6i,[rcp_role6,rcp_Xb_3,3_10,X_10],[3_10,Xb_10,r9]
rcp_6j,[rcp_role6,rcp_Xb_3,3_10,Xb_10],[3_10,Xb_10,r9]
rcp_6k,[rcp_role6,rcp_Xb_3,3_11,X_11],[3_11,Xb_11,r9]
rcp_6l,[rcp_role6,rcp_Xb_3,3_11,Xb_11],[3_11,Xb_11,r9]
rcp_6m,[rcp_role6,rcp_X_4,4_00,X_00],[4_00,X_00,r9]
rcp_6n,[rcp_role6,rcp_X_4,4_00,Xb_00],[4_00,X_00,r9]
rcp_6o,[rcp_role6,rcp_X_4,4_01,X_01],[4_01,X_01,r9]
rcp_6p,[rcp_role6,rcp_X_4,4_01,Xb_01],[4_01,X_01,r9]
rcp_6q,[rcp_role6,rcp_X_4,4_10,X_10],[4_10,X_10,r9]
rcp_6r,[rcp_role6,rcp_X_4,4_10,Xb_10],[4_10,X_10,r9]
rcp_6s,[rcp_role6,rcp_X_4,4_11,X_11],[4_11,X_11,r9]
rcp_6t,[rcp_role6,rcp_X_4,4_11,Xb_11],[4_11,X_11,r9]
r9_strt_rd_T00a,[r9,r=0,d0=0,d1=0],[reading_00,r=0,d0=0,d1=0,enable]
r9_strt_rd_T00b,[r9,r=0,d0=1,d1=1],[reading_00,r=0,d0=1,d1=1,enable]
r9_end_rd_T00,[reading_00,enable],[clear_x,enable]
r9_strt_rd_T01a,[r9,r=0,d0=0,d1=1],[reading_01,r=0,d0=0,d1=1,enable]
r9_strt_rd_T01b,[r9,r=0,d0=1,d1=0],[reading_01,r=0,d0=1,d1=0,enable]
r9_end_rd_T01,[reading_01,enable],[clear_x,enable]
r9_strt_rd_T10a,[r9,r=1,d0=0,d1=0],[reading_10,r=1,d0=0,d1=0,enable]
r9_strt_rd_T10b,[r9,r=1,d0=1,d1=1],[reading_10,r=1,d0=1,d1=1,enable]
r9_end_rd_T10,[reading_10,enable],[clear_x,enable]
r9_strt_rd_T11a,[r9,r=1,d0=0,d1=1],[reading_11,r=1,d0=0,d1=1,enable]
r9_strt_rd_T11b,[r9,r=1,d0=1,d1=0],[reading_11,r=1,d0=1,d1=0,enable]
r9_end_rd_T11,[reading_11,enable],[clear_x,enable]
COM
COM check coherence
COM
order_lost_00,[enable,reading_00,x_00,r0+],[enable,reading_00,x_00,
order_lost,r0+]
order_lost_01,[enable,reading_01,x_01,r0+],[enable,reading_01,x_01,
order_lost,r0+]
order_lost_10,[enable,reading_10,x_10,r0+],[enable,reading_10,x_10,
order_lost,r0+]
order_lost_11,[enable,reading_11,x_11,r0+],[enable,reading_11,x_11,
order_lost,r0+]
```

```
END
```

B.4 Anderson '$4b + 9$' Petri net models

B.4.1 Model for testing coherence

```
Anderson's 4b+9 algorithm. Coherence test.
bad=[coherence_lost]
initial=[w0,r0,d=0,RP=0,wp=0,walt=0,Z0=0,Z1=0,WP=0,ralt=0,rp=0]
COM
COM writer
COM
COM w0: d := RP;
COM
w0a,[w0,+0d=0,+0d=1,RP=0],[w1,d=0,RP=0]
w0b,[w0,+0d=0,+0d=1,RP=1],[w1,d=1,RP=1]
COM
COM w1: wp := not d;
COM
w1a,[w1,+0wp=0,+0wp=1,d=0],[w2,wp=1,d=0]
w1b,[w1,+0wp=0,+0wp=1,d=1],[w2,wp=0,d=1]
COM
COM w2: walt := not walt;
COM
w2a,[w2,walt=0],[w3,walt=1]
w2b,[w2,walt=1],[w3,walt=0]
COM
COM w3: Y[wp,walt] := input;
COM
st_wr_00,[w3,wp=0,walt=0],[writing_00,wp=0,walt=0]
end_wr_00,[writing_00],[w4]
st_wr_01,[w3,wp=0,walt=1],[writing_01,wp=0,walt=1]
end_wr_01,[writing_01],[w4]
st_wr_10,[w3,wp=1,walt=0],[writing_10,wp=1,walt=0]
end_wr_10,[writing_10],[w4]
st_wr_11,[w3,wp=1,walt=1],[writing_11,wp=1,walt=1]
end_wr_11,[writing_11],[w4]
COM
COM w4: Z[wp] := walt;
COM
w4a,[w4,wp=0,+0Z0=0,+0Z0=1,walt=0],[w5,wp=0,Z0=0,walt=0]
w4b,[w4,wp=0,+0Z0=0,+0Z0=1,walt=1],[w5,wp=0,Z0=1,walt=1]
w4c,[w4,wp=1,+0Z1=0,+0Z1=1,walt=0],[w5,wp=1,Z1=0,walt=0]
w4d,[w4,wp=1,+0Z1=0,+0Z1=1,walt=1],[w5,wp=1,Z1=1,walt=1]
COM
COM w5: WP := wp;
COM
w5a,[w5,+0WP=0,+0WP=1,wp=0],[w0,WP=0,wp=0]
w5b,[w5,+0WP=0,+0WP=1,wp=1],[w0,WP=1,wp=1]
COM
COM reader
COM
COM r0: rp := WP;
COM
r0a,[r0,+0rp=0,+0rp=1,WP=0],[r1,rp=0,WP=0]
r0b,[r0,+0rp=0,+0rp=1,WP=1],[r1,rp=1,WP=1]
COM
COM r1: RP := rp;
COM
r1a,[r1,+0RP=0,+0RP=1,rp=0],[r2,RP=0,rp=0]
r1b,[r1,+0RP=0,+0RP=1,rp=1],[r2,RP=1,rp=1]
```

```
COM
COM r2: ralt := Z[rp];
COM
r2a,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=0],[r3,ralt=0,rp=0,Z0=0]
r2b,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=1],[r3,ralt=1,rp=0,Z0=1]
r2c,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=0],[r3,ralt=0,rp=1,Z1=0]
r2d,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=1],[r3,ralt=1,rp=1,Z1=1]
COM
COM r3: output := Y[rp,ralt];
COM
st_rd_00,[r3,rp=0,ralt=0],[reading_00,rp=0,ralt=0]
end_rd_00,[reading_00],[r0]
st_rd_01,[r3,rp=0,ralt=1],[reading_01,rp=0,ralt=1]
end_rd_01,[reading_01],[r0]
st_rd_10,[r3,rp=1,ralt=0],[reading_10,rp=1,ralt=0]
end_rd_10,[reading_10],[r0]
st_rd_11,[r3,rp=1,ralt=1],[reading_11,rp=1,ralt=1]
end_rd_11,[reading_11],[r0]
COM
COM cherence check
COM
coh1,[writing_00,reading_00],[coherence_lost]
coh2,[writing_01,reading_01],[coherence_lost]
coh3,[writing_10,reading_10],[coherence_lost]
coh4,[writing_11,reading_11],[coherence_lost]
END.
```

B.4.2 Model for testing freshness

```
Anderson's 4b+9 algorithm. freshness test (with clear p by reader).
bad=[not_fresh]
initial=[enable,w0,d=0,RP=0,wp=0,walt=0,Z0=0,Z1=0,WP=0,ralt=0,rp=0,clear_valid,
no_w1,no_r0,no_w_fresh,no_w_prev,not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,
not_r_fresh_11,not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,
not_valid_00,not_valid_01,not_valid_10,not_valid_11]
COM
COM writer
COM
COM w0: d := RP;
COM
w0a,[w0,+0d=0,+0d=1,RP=0,enable],[w1,d=0,RP=0,enable]
w0b,[w0,+0d=0,+0d=1,RP=1,enable],[w1,d=1,RP=1,enable]
COM
COM w1: wp := not d;
COM
w1a,[w1,+0wp=0,+0wp=1,d=0,enable],[w2,wp=1,d=0,enable]
w1b,[w1,+0wp=0,+0wp=1,d=1,enable],[w2,wp=0,d=1,enable]
COM
COM w2: walt := not walt;
COM
w2a,[w2,walt=0,enable],[w3,walt=1,enable]
w2b,[w2,walt=1,enable],[w3,walt=0,enable]
COM
COM w3: Y[wp,walt] := input;
COM
st_wr_00,[w3,wp=0,walt=0,enable],[writing_00,wp=0,walt=0,enable]
end_wr_00,[writing_00,enable],[f_to_p,wrote_00]
st_wr_01,[w3,wp=0,walt=1,enable],[writing_01,wp=0,walt=1,enable]
end_wr_01,[writing_01,enable],[f_to_p,wrote_01]
st_wr_10,[w3,wp=1,walt=0,enable],[writing_10,wp=1,walt=0,enable]
end_wr_10,[writing_10,enable],[f_to_p,wrote_10]
st_wr_11,[w3,wp=1,walt=1,enable],[writing_11,wp=1,walt=1,enable]
end_wr_11,[writing_11,enable],[f_to_p,wrote_11]
COM
```

```
nftop,[f_to_p,no_w_fresh],[w_to_f,no_w_fresh]
ftop00,[w_fresh_00,no_w_prev,f_to_p],[no_w_fresh,w_prev_00,w_to_f]
ftop01,[w_fresh_01,no_w_prev,f_to_p],[no_w_fresh,w_prev_01,w_to_f]
ftop10,[w_fresh_10,no_w_prev,f_to_p],[no_w_fresh,w_prev_10,w_to_f]
ftop11,[w_fresh_11,no_w_prev,f_to_p],[no_w_fresh,w_prev_11,w_to_f]
wtofv00b,[wrote_00,valid_00,w_to_f,no_w_fresh],[w4,enable,w_fresh_00,
valid_00]
wtofv00c,[wrote_00,not_valid_00,w_to_f,no_w_fresh],[w4,enable,w_fresh_00,
valid_00]
wtofv01b,[wrote_01,valid_01,w_to_f,no_w_fresh],[w4,enable,w_fresh_01,
valid_01]
wtofv01c,[wrote_01,not_valid_01,w_to_f,no_w_fresh],[w4,enable,w_fresh_01,
valid_01]
wtofv10b,[wrote_10,valid_10,w_to_f,no_w_fresh],[w4,enable,w_fresh_10,
valid_10]
wtofv10c,[wrote_10,not_valid_10,w_to_f,no_w_fresh],[w4,enable,w_fresh_10,
valid_10]
wtofv11b,[wrote_11,valid_11,w_to_f,no_w_fresh],[w4,enable,w_fresh_11,
valid_11]
wtofv11c,[wrote_11,not_valid_11,w_to_f,no_w_fresh],[w4,enable,w_fresh_11,
valid_11]
COM
COM w4: Z[wp] := walt;
COM
w4a,[w4,wp=0,+0Z0=0,+0Z0=1,walt=0,enable],[w5,wp=0,Z0=0,walt=0,enable]
w4b,[w4,wp=0,+0Z0=0,+0Z0=1,walt=1,enable],[w5,wp=0,Z0=1,walt=1,enable]
w4c,[w4,wp=1,+0Z1=0,+0Z1=1,walt=0,enable],[w5,wp=1,Z1=0,walt=0,enable]
w4d,[w4,wp=1,+0Z1=0,+0Z1=1,walt=1,enable],[w5,wp=1,Z1=1,walt=1,enable]
COM
COM w5: WP := wp;
COM
w5a,[w5,+0WP=0,+0WP=1,wp=0,enable],[w1_post,WP=0,wp=0]
w5b,[w5,+0WP=0,+0WP=1,wp=1,enable],[w1_post,WP=1,wp=1]
COM
w1_initial,[w1_post,no_w1],[clear_w_prev,w1+]
w1_normal,[w1_post,w1+],[clear_w_prev,w1+]
ncp,[no_w_prev,clear_w_prev],[w0,enable,no_w_prev]
cp00,[w_prev_00,clear_w_prev],[w0,enable,no_w_prev]
cp01,[w_prev_01,clear_w_prev],[w0,enable,no_w_prev]
cp10,[w_prev_10,clear_w_prev],[w0,enable,no_w_prev]
cp11,[w_prev_11,clear_w_prev],[w0,enable,no_w_prev]
COM
COM reader
COM
COM r0: rp := WP;
COM
clear_valid_00a,[not_valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_00b,[valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_01a,[not_valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_01b,[valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_10a,[not_valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_10b,[valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_11a,[not_valid_11,clear_valid4],[not_valid_11,clear_r_f]
clear_valid_11b,[valid_11,clear_valid4],[not_valid_11,clear_r_f]
no_rf,[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,
clear_r_f],[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,
wf_to_rf]
clear_rf00,[r_fresh_00,clear_r_f],[not_r_fresh_00,wf_to_rf]
clear_rf01,[r_fresh_01,clear_r_f],[not_r_fresh_01,wf_to_rf]
clear_rf10,[r_fresh_10,clear_r_f],[not_r_fresh_10,wf_to_rf]
clear_rf11,[r_fresh_11,clear_r_f],[not_r_fresh_11,wf_to_rf]
no_wf,[no_w_fresh,wf_to_rf,not_valid_00,not_valid_01,not_valid_10,
not_valid_11],[no_w_fresh,r0_pre,valid_00,valid_01,valid_10,valid_11]
wf_to_rf_00,[w_fresh_00,not_r_fresh_00,wf_to_rf],[w_fresh_00,r_fresh_00,
clear_r_p]
wf_to_rf_01,[w_fresh_01,not_r_fresh_01,wf_to_rf],[w_fresh_01,r_fresh_01,
```

```
clear_r_p]
wf_to_rf_10,[w_fresh_10,not_r_fresh_10,wf_to_rf],[w_fresh_10,r_fresh_10,
clear_r_p]
wf_to_rf_11,[w_fresh_11,not_r_fresh_11,wf_to_rf],[w_fresh_11,r_fresh_11,
clear_r_p]
no_rp,[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,clear_r_p],
[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,wp_to_rp]
clear_rp00,[r_prev_00,clear_r_p],[not_r_prev_00,wp_to_rp]
clear_rp01,[r_prev_01,clear_r_p],[not_r_prev_01,wp_to_rp]
clear_rp10,[r_prev_10,clear_r_p],[not_r_prev_10,wp_to_rp]
clear_rp11,[r_prev_11,clear_r_p],[not_r_prev_11,wp_to_rp]
no_wp,[no_w_prev,wp_to_rp],[no_w_prev,r0_pre]
wp_to_rp_00,[w_prev_00,not_r_prev_00,wp_to_rp],[w_prev_00,r_prev_00,r0_pre]
wp_to_rp_01,[w_prev_01,not_r_prev_01,wp_to_rp],[w_prev_01,r_prev_01,r0_pre]
wp_to_rp_10,[w_prev_10,not_r_prev_10,wp_to_rp],[w_prev_10,r_prev_10,r0_pre]
wp_to_rp_11,[w_prev_11,not_r_prev_11,wp_to_rp],[w_prev_11,r_prev_11,r0_pre]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
COM
r0a,[r0,+0rp=0,+0rp=1,WP=0],[r1,rp=0,WP=0,enable]
r0b,[r0,+0rp=0,+0rp=1,WP=1],[r1,rp=1,WP=1,enable]
COM
COM r1: RP := rp;
COM
r1a,[r1,+0RP=0,+0RP=1,rp=0,enable],[r2,RP=0,rp=0,enable]
r1b,[r1,+0RP=0,+0RP=1,rp=1,enable],[r2,RP=1,rp=1,enable]
COM
COM r2: ralt := Z[rp];
COM
r2a,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=0,enable],[pre_read_ctrl,ralt=0,rp=0,
Z0=0,enable]
r2b,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=1,enable],[pre_read_ctrl,ralt=1,rp=0,
Z0=1,enable]
r2c,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=0,enable],[pre_read_ctrl,ralt=0,rp=1,
Z1=0,enable]
r2d,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=1,enable],[pre_read_ctrl,ralt=1,rp=1,
Z1=1,enable]
COM
COM r3: output := Y[rp,ralt];
COM
pre_read_00a,[pre_read_ctrl,enable,rp=0,ralt=0,no_w_fresh],[no_w_fresh,r3,
rp=0,ralt=0]
pre_read_00b,[pre_read_ctrl,enable,rp=0,ralt=0,w_fresh_00],[w_fresh_00,
clear_prev_by_r,rp=0,ralt=0]
pre_read_00c,[pre_read_ctrl,enable,rp=0,ralt=0,w_fresh_01],[w_fresh_01,r3,
rp=0,ralt=0]
pre_read_00d,[pre_read_ctrl,enable,rp=0,ralt=0,w_fresh_10],[w_fresh_10,r3,
rp=0,ralt=0]
pre_read_00e,[pre_read_ctrl,enable,rp=0,ralt=0,w_fresh_11],[w_fresh_11,r3,
rp=0,ralt=0]
pre_read_01a,[pre_read_ctrl,enable,rp=0,ralt=1,no_w_fresh],[no_w_fresh,r3,
rp=0,ralt=1]
pre_read_01b,[pre_read_ctrl,enable,rp=0,ralt=1,w_fresh_00],[w_fresh_00,r3,
rp=0,ralt=1]
pre_read_01c,[pre_read_ctrl,enable,rp=0,ralt=1,w_fresh_01],[w_fresh_01,
clear_prev_by_r,rp=0,ralt=1]
pre_read_01d,[pre_read_ctrl,enable,rp=0,ralt=1,w_fresh_10],[w_fresh_10,r3,
rp=0,ralt=1]
pre_read_01e,[pre_read_ctrl,enable,rp=0,ralt=1,w_fresh_11],[w_fresh_11,r3,
rp=0,ralt=1]
pre_read_10a,[pre_read_ctrl,enable,rp=1,ralt=0,no_w_fresh],[no_w_fresh,r3,
rp=1,ralt=0]
pre_read_10b,[pre_read_ctrl,enable,rp=1,ralt=0,w_fresh_00],[w_fresh_00,r3,
rp=1,ralt=0]
pre_read_10c,[pre_read_ctrl,enable,rp=1,ralt=0,w_fresh_01],[w_fresh_01,r3,
```

```
rp=1,ralt=0]
pre_read_10d,[pre_read_ctrl,enable,rp=1,ralt=0,w_fresh_10],[w_fresh_10,
clear_prev_by_r,rp=1,ralt=0]
pre_read_10e,[pre_read_ctrl,enable,rp=1,ralt=0,w_fresh_11],[w_fresh_11,r3,
rp=1,ralt=0]
pre_read_11a,[pre_read_ctrl,enable,rp=1,ralt=1,no_w_fresh],[no_w_fresh,r3,
rp=1,ralt=1]
pre_read_11b,[pre_read_ctrl,enable,rp=1,ralt=1,w_fresh_00],[w_fresh_00,r3,
rp=1,ralt=1]
pre_read_11c,[pre_read_ctrl,enable,rp=1,ralt=1,w_fresh_01],[w_fresh_01,r3,
rp=1,ralt=1]
pre_read_11d,[pre_read_ctrl,enable,rp=1,ralt=1,w_fresh_10],[w_fresh_10,r3,
rp=1,ralt=1]
pre_read_11e,[pre_read_ctrl,enable,rp=1,ralt=1,w_fresh_11],[w_fresh_11,
clear_prev_by_r,rp=1,ralt=1]
ncp_r,[no_w_prev,clear_prev_by_r],[r3,no_w_prev]
cp00_r,[w_prev_00,clear_prev_by_r],[r3,no_w_prev]
cp01_r,[w_prev_01,clear_prev_by_r],[r3,no_w_prev]
cp10_r,[w_prev_10,clear_prev_by_r],[r3,no_w_prev]
cp11_r,[w_prev_11,clear_prev_by_r],[r3,no_w_prev]
st_rd_00,[r3,rp=0,ralt=0],[reading_00,rp=0,ralt=0,enable]
end_rd_00,[reading_00,enable],[clear_valid,enable]
st_rd_01,[r3,rp=0,ralt=1],[reading_01,rp=0,ralt=1,enable]
end_rd_01,[reading_01,enable],[clear_valid,enable]
st_rd_10,[r3,rp=1,ralt=0],[reading_10,rp=1,ralt=0,enable]
end_rd_10,[reading_10,enable],[clear_valid,enable]
st_rd_11,[r3,rp=1,ralt=1],[reading_11,rp=1,ralt=1,enable]
end_rd_11,[reading_11,enable],[clear_valid,enable]
COM
COM freshness check
COM
not_fresh_00,[enable,reading_00,not_valid_00,not_r_fresh_00,
not_r_prev_00,r0+],[enable,reading_00,not_valid_00,not_r_fresh_00,
not_r_prev_00,not_fresh,r0+]
not_fresh_01,[enable,reading_01,not_valid_01,not_r_fresh_01,
not_r_prev_01,r0+],[enable,reading_01,not_valid_01,not_r_fresh_01,
not_r_prev_01,not_fresh,r0+]
not_fresh_10,[enable,reading_10,not_valid_10,not_r_fresh_10,
not_r_prev_10,r0+],[enable,reading_10,not_valid_10,not_r_fresh_10,
not_r_prev_10,not_fresh,r0+]
not_fresh_11,[enable,reading_11,not_valid_11,not_r_fresh_11,
not_r_prev_11,r0+],[enable,reading_11,not_valid_11,not_r_fresh_11,
not_r_prev_11,not_fresh,r0+]
END.
```

B.4.3 Model for testing sequence

```
Anderson's 4b+9 algorithm. Sequence test.
bad=[order_lost]
initial=[enable,w0,d=0,RP=0,wp=0,walt=0,Z0=0,Z1=0,WP=0,ralt=0,rp=0,
clear_x,no_w1,no_r0,Xb_00,Xb_01,Xb_10,Xb_11,x_00,x_01,x_10,x_11,4_00,
4_01,4_10,4_11,not1,not2,not3]
COM
COM writer
COM
COM w0: d := RP;
COM
w0a,[w0,+0d=0,+0d=1,RP=0,enable],[w1,d=0,RP=0,enable]
w0b,[w0,+0d=0,+0d=1,RP=1,enable],[w1,d=1,RP=1,enable]
COM
COM w1: wp := not d;
COM
w1a,[w1,+0wp=0,+0wp=1,d=0,enable],[w2,wp=1,d=0,enable]
w1b,[w1,+0wp=0,+0wp=1,d=1,enable],[w2,wp=0,d=1,enable]
```

```
COM
COM w2: walt := not walt;
COM
w2a,[w2,walt=0,enable],[w3,walt=1,enable]
w2b,[w2,walt=1,enable],[w3,walt=0,enable]
COM
COM w3: Y[wp,walt] := input;
COM
st_wr_00,[w3,wp=0,walt=0,enable],[writing_00,wp=0,walt=0,enable]
end_wr_00,[writing_00,enable],[wip_roles00]
st_wr_01,[w3,wp=0,walt=1,enable],[writing_01,wp=0,walt=1,enable]
end_wr_01,[writing_01,enable],[wip_roles01]
st_wr_10,[w3,wp=1,walt=0,enable],[writing_10,wp=1,walt=0,enable]
end_wr_10,[writing_10,enable],[wip_roles10]
st_wr_11,[w3,wp=1,walt=1,enable],[writing_11,wp=1,walt=1,enable]
end_wr_11,[writing_11,enable],[wip_roles11]
COM
wip1a,[wip_roles00,X_00],[wip_role200,Xb_00]
wip1b,[wip_roles00,Xb_00],[wip_role200,Xb_00]
wip1c,[wip_roles01,X_01],[wip_role201,Xb_01]
wip1d,[wip_roles01,Xb_01],[wip_role201,Xb_01]
wip1e,[wip_roles10,X_10],[wip_role210,Xb_10]
wip1f,[wip_roles10,Xb_10],[wip_role210,Xb_10]
wip1g,[wip_roles11,X_11],[wip_role211,Xb_11]
wip1h,[wip_roles11,Xb_11],[wip_role211,Xb_11]
wip2a,[wip_role200,1_00],[1_00,w4,enable]
wip2b,[wip_role201,1_01],[1_01,w4,enable]
wip2c,[wip_role210,1_10],[1_10,w4,enable]
wip2d,[wip_role211,1_11],[1_11,w4,enable]
wip2e,[wip_role200,2_00],[w2b1_00,11b2,2_00]
wip2f,[wip_role201,2_01],[w2b1_01,11b2,2_01]
wip2g,[wip_role210,2_10],[w2b1_10,11b2,2_10]
wip2h,[wip_role211,2_11],[w2b1_11,11b2,2_11]
wip2i,[wip_role200,3_00],[w3b1_00,22b3,3_00]
wip2j,[wip_role201,3_01],[w3b1_01,22b3,3_01]
wip2k,[wip_role210,3_10],[w3b1_10,22b3,3_10]
wip2l,[wip_role211,3_11],[w3b1_11,22b3,3_11]
wip2m,[wip_role200,4_00],[w4b1_00,33b4,4_00]
wip2n,[wip_role201,4_01],[w4b1_01,33b4,4_01]
wip2o,[wip_role210,4_10],[w4b1_10,33b4,4_10]
wip2p,[wip_role211,4_11],[w4b1_11,33b4,4_11]
wip3a,[33b4,3_00],[22b3,4_00]
wip3b,[33b4,3_01],[22b3,4_01]
wip3c,[33b4,3_10],[22b3,4_10]
wip3d,[33b4,3_11],[22b3,4_11]
wip3e,[33b4,not3],[22b3,not3]
wip4a,[22b3,2_00,not3],[11b2,3_00,not3b]
wip4aa,[22b3,2_00,not3b],[11b2,3_00,not3b]
wip4b,[22b3,2_01,not3],[11b2,3_01,not3b]
wip4ba,[22b3,2_01,not3b],[11b2,3_01,not3b]
wip4c,[22b3,2_10,not3],[11b2,3_10,not3b]
wip4ca,[22b3,2_10,not3b],[11b2,3_10,not3b]
wip4d,[22b3,2_11,not3],[11b2,3_11,not3b]
wip4da,[22b3,2_11,not3b],[11b2,3_11,not3b]
wip4e,[22b3,not2],[11b2,not2]
wip5a,[11b2,1_00,not2],[11b2_fin,2_00,not2b]
wip5aa,[11b2,1_00,not2b],[11b2_fin,2_00,not2b]
wip5b,[11b2,1_01,not2],[11b2_fin,2_01,not2b]
wip5ba,[11b2,1_01,not2b],[11b2_fin,2_01,not2b]
wip5c,[11b2,1_10,not2],[11b2_fin,2_10,not2b]
wip5ca,[11b2,1_10,not2b],[11b2_fin,2_10,not2b]
wip5d,[11b2,1_11,not2],[11b2_fin,2_11,not2b]
wip5da,[11b2,1_11,not2b],[11b2_fin,2_11,not2b]
wip5e,[11b2,not1],[11b2_fin,not1]
wip6a,[11b2_fin,w2b1_00,2_00],[1_00,w4,enable]
wip6b,[11b2_fin,w2b1_01,2_01],[1_01,w4,enable]
```

```
wip6c,[11b2_fin,w2b1_10,2_10],[1_10,w4,enable]
wip6d,[11b2_fin,w2b1_11,2_11],[1_11,w4,enable]
wip6e,[11b2_fin,w3b1_00,3_00],[1_00,w4,enable]
wip6f,[11b2_fin,w3b1_01,3_01],[1_01,w4,enable]
wip6g,[11b2_fin,w3b1_10,3_10],[1_10,w4,enable]
wip6h,[11b2_fin,w3b1_11,3_11],[1_11,w4,enable]
wip6i,[11b2_fin,w4b1_00,4_00,not1],[1_00,w4,enable,not1b]
wip6ia,[11b2_fin,w4b1_00,4_00,not1b],[1_00,w4,enable,not1b]
wip6j,[11b2_fin,w4b1_01,4_01,not1],[1_01,w4,enable,not1b]
wip6ja,[11b2_fin,w4b1_01,4_01,not1b],[1_01,w4,enable,not1b]
wip6k,[11b2_fin,w4b1_10,4_10,not1],[1_10,w4,enable,not1b]
wip6ka,[11b2_fin,w4b1_10,4_10,not1b],[1_10,w4,enable,not1b]
wip6l,[11b2_fin,w4b1_11,4_11,not1],[1_11,w4,enable,not1b]
wip6la,[11b2_fin,w4b1_11,4_11,not1b],[1_11,w4,enable,not1b]
COM
COM w4: Z[wp] := walt;
COM
w4a,[w4,wp=0,+0Z0=0,+0Z0=1,walt=0,enable],[w5,wp=0,Z0=0,walt=0,enable]
w4b,[w4,wp=0,+0Z0=0,+0Z0=1,walt=1,enable],[w5,wp=0,Z0=1,walt=1,enable]
w4c,[w4,wp=1,+0Z1=0,+0Z1=1,walt=0,enable],[w5,wp=1,Z1=0,walt=0,enable]
w4d,[w4,wp=1,+0Z1=0,+0Z1=1,walt=1,enable],[w5,wp=1,Z1=1,walt=1,enable]
COM
COM w5: WP := wp;
COM
w5a,[w5,+0WP=0,+0WP=1,wp=0,enable],[w1_post,WP=0,wp=0]
w5b,[w5,+0WP=0,+0WP=1,wp=1,enable],[w1_post,WP=1,wp=1]
COM
w1_initial,[w1_post,no_w1],[w0,w1+,enable]
w1_normal,[w1_post,w1+],[w0,w1+,enable]
COM
COM reader
COM
COM r0: rp := WP;
COM
clr_x00,[enable,clear_x,x_00],[clear_x2,xb_00]
clr_x00a,[enable,clear_x,xb_00],[clear_x2,xb_00]
clr_x01,[clear_x2,x_01],[clear_x3,xb_01]
clr_x01a,[clear_x2,xb_01],[clear_x3,xb_01]
clr_x02,[clear_x3,x_10],[clear_x4,xb_10]
clr_x02a,[clear_x3,xb_10],[clear_x4,xb_10]
clr_x03,[clear_x4,x_11],[r0_pre,xb_11]
clr_x03a,[clear_x4,xb_11],[r0_pre,xb_11]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
COM
r0a,[r0,+0rp=0,+0rp=1,WP=0],[r1,rp=0,WP=0,enable]
r0b,[r0,+0rp=0,+0rp=1,WP=1],[r1,rp=1,WP=1,enable]
COM
COM r1: RP := rp;
COM
r1a,[r1,+0RP=0,+0RP=1,rp=0,enable],[r2,RP=0,rp=0,enable]
r1b,[r1,+0RP=0,+0RP=1,rp=1,enable],[r2,RP=1,rp=1,enable]
COM
COM r2: ralt := Z[rp];
COM
r2a,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=0,enable],[rcp_roles,ralt=0,rp=0,
Z0=0,enable]
r2b,[r2,+0ralt=0,+0ralt=1,rp=0,Z0=1,enable],[rcp_roles,ralt=1,rp=0,
Z0=1,enable]
r2c,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=0,enable],[rcp_roles,ralt=0,rp=1,
Z1=0,enable]
r2d,[r2,+0ralt=0,+0ralt=1,rp=1,Z1=1,enable],[rcp_roles,ralt=1,rp=1,
Z1=1,enable]
COM
COM r3: output := Y[rp,ralt];
```

```
COM
x_r00a,[enable,rcp_roles,rp=0,ralt=0,X_00,xb_00],[rp=0,ralt=0,
X_00,x_00,rcp_roles2_00]
x_r00b,[enable,rcp_roles,rp=0,ralt=0,Xb_00],[rp=0,ralt=0,Xb_00,
rcp_roles2_00]
x_r01a,[enable,rcp_roles,rp=0,ralt=1,X_01,xb_01],[rp=0,ralt=1,
X_01,x_01,rcp_roles2_01]
x_r01b,[enable,rcp_roles,rp=0,ralt=1,Xb_01],[rp=0,ralt=1,Xb_01,
rcp_roles2_01]
x_r10a,[enable,rcp_roles,rp=1,ralt=0,X_10,xb_10],[rp=1,ralt=0,
X_10,x_10,rcp_roles2_10]
x_r10b,[enable,rcp_roles,rp=1,ralt=0,Xb_10],[rp=1,ralt=0,Xb_10,
rcp_roles2_10]
x_r11a,[enable,rcp_roles,rp=1,ralt=1,X_11,xb_11],[rp=1,ralt=1,
X_11,x_11,rcp_roles2_11]
x_r11b,[enable,rcp_roles,rp=1,ralt=1,Xb_11],[rp=1,ralt=1,Xb_11,
rcp_roles2_11]
X_r00a,[rcp_roles2_00,1_00],[1_00,rcp_Xb_00,rcp_X_01,rcp_X_10,
rcp_X_11,rcp_role3]
X_r00b,[rcp_roles2_00,2_00],[2_00,rcp_Xb_00,rcp_Xb_1,rcp_X_3,
rcp_X_4,rcp_role3]
X_r00c,[rcp_roles2_00,3_00],[3_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,
rcp_X_4,rcp_role3]
X_r00d,[rcp_roles2_00,4_00],[4_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,
rcp_Xb_3,rcp_role3]
X_r01a,[rcp_roles2_01,1_01],[1_01,rcp_Xb_01,rcp_X_00,rcp_X_10,
rcp_X_11,rcp_role3]
X_r01b,[rcp_roles2_01,2_01],[2_01,rcp_Xb_01,rcp_Xb_1,rcp_X_3,
rcp_X_4,rcp_role3]
X_r01c,[rcp_roles2_01,3_01],[3_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,
rcp_X_4,rcp_role3]
X_r01d,[rcp_roles2_01,4_01],[4_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,
rcp_Xb_3,rcp_role3]
X_r10a,[rcp_roles2_10,1_10],[1_10,rcp_Xb_10,rcp_X_01,rcp_X_00,
rcp_X_11,rcp_role3]
X_r10b,[rcp_roles2_10,2_10],[2_10,rcp_Xb_10,rcp_Xb_1,rcp_X_3,
rcp_X_4,rcp_role3]
X_r10c,[rcp_roles2_10,3_10],[3_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,
rcp_X_4,rcp_role3]
X_r10d,[rcp_roles2_10,4_10],[4_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,
rcp_Xb_3,rcp_role3]
X_r11a,[rcp_roles2_11,1_11],[1_11,rcp_Xb_11,rcp_X_01,rcp_X_10,
rcp_X_00,rcp_role3]
X_r11b,[rcp_roles2_11,2_11],[2_11,rcp_Xb_11,rcp_Xb_1,rcp_X_3,
rcp_X_4,rcp_role3]
X_r11c,[rcp_roles2_11,3_11],[3_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,
rcp_X_4,rcp_role3]
X_r11d,[rcp_roles2_11,4_11],[4_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,
rcp_Xb_3,rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_00,X_00],[Xb_00,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_00,Xb_00],[Xb_00,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_01,X_01],[Xb_01,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_01,Xb_01],[Xb_01,rcp_role4]
rcp_3e,[rcp_role3,rcp_Xb_10,X_10],[Xb_10,rcp_role4]
rcp_3f,[rcp_role3,rcp_Xb_10,Xb_10],[Xb_10,rcp_role4]
rcp_3g,[rcp_role3,rcp_Xb_11,X_11],[Xb_11,rcp_role4]
rcp_3h,[rcp_role3,rcp_Xb_11,Xb_11],[Xb_11,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_01,X_01],[X_01,rcp_role5]
rcp_4b,[rcp_role4,rcp_X_01,Xb_01],[X_01,rcp_role5]
rcp_4c,[rcp_role4,rcp_X_00,Xb_00],[X_00,rcp_role5]
rcp_4d,[rcp_role4,rcp_X_00,X_00],[X_00,rcp_role5]
rcp_4e,[rcp_role4,rcp_Xb_1,1_00,X_00],[1_00,Xb_00,rcp_role5]
rcp_4f,[rcp_role4,rcp_Xb_1,1_00,Xb_00],[1_00,Xb_00,rcp_role5]
rcp_4g,[rcp_role4,rcp_Xb_1,1_01,X_01],[1_01,Xb_01,rcp_role5]
rcp_4h,[rcp_role4,rcp_Xb_1,1_01,Xb_01],[1_01,Xb_01,rcp_role5]
rcp_4i,[rcp_role4,rcp_Xb_1,1_10,X_10],[1_10,Xb_10,rcp_role5]
```

```
rcp_4j,[rcp_role4,rcp_Xb_1,1_10,Xb_10],[1_10,Xb_10,rcp_role5]
rcp_4k,[rcp_role4,rcp_Xb_1,1_11,X_11],[1_11,Xb_11,rcp_role5]
rcp_4l,[rcp_role4,rcp_Xb_1,1_11,Xb_11],[1_11,Xb_11,rcp_role5]
rcp_5a,[rcp_role5,rcp_X_10,X_10],[X_10,rcp_role6]
rcp_5b,[rcp_role5,rcp_X_10,Xb_10],[X_10,rcp_role6]
rcp_5c,[rcp_role5,rcp_X_00,X_00],[X_00,rcp_role6]
rcp_5d,[rcp_role5,rcp_X_00,Xb_00],[X_00,rcp_role6]
rcp_5e,[rcp_role5,rcp_Xb_2,2_00,X_00],[2_00,Xb_00,rcp_role6]
rcp_5f,[rcp_role5,rcp_Xb_2,2_00,Xb_00],[2_00,Xb_00,rcp_role6]
rcp_5g,[rcp_role5,rcp_Xb_2,2_01,X_01],[2_01,Xb_01,rcp_role6]
rcp_5h,[rcp_role5,rcp_Xb_2,2_01,Xb_01],[2_01,Xb_01,rcp_role6]
rcp_5i,[rcp_role5,rcp_Xb_2,2_10,X_10],[2_10,Xb_10,rcp_role6]
rcp_5j,[rcp_role5,rcp_Xb_2,2_10,Xb_10],[2_10,Xb_10,rcp_role6]
rcp_5k,[rcp_role5,rcp_Xb_2,2_11,X_11],[2_11,Xb_11,rcp_role6]
rcp_5l,[rcp_role5,rcp_Xb_2,2_11,Xb_11],[2_11,Xb_11,rcp_role6]
rcp_5m,[rcp_role5,rcp_X_3,3_00,X_00],[3_00,X_00,rcp_role6]
rcp_5n,[rcp_role5,rcp_X_3,3_00,Xb_00],[3_00,X_00,rcp_role6]
rcp_5o,[rcp_role5,rcp_X_3,3_01,X_01],[3_01,X_01,rcp_role6]
rcp_5p,[rcp_role5,rcp_X_3,3_01,Xb_01],[3_01,X_01,rcp_role6]
rcp_5q,[rcp_role5,rcp_X_3,3_10,X_10],[3_10,X_10,rcp_role6]
rcp_5r,[rcp_role5,rcp_X_3,3_10,Xb_10],[3_10,X_10,rcp_role6]
rcp_5s,[rcp_role5,rcp_X_3,3_11,X_11],[3_11,X_11,rcp_role6]
rcp_5t,[rcp_role5,rcp_X_3,3_11,Xb_11],[3_11,X_11,rcp_role6]
rcp_6a,[rcp_role6,rcp_X_11,X_11],[X_11,r3]
rcp_6b,[rcp_role6,rcp_X_11,Xb_11],[X_11,r3]
rcp_6c,[rcp_role6,rcp_X_00,X_00],[X_00,r3]
rcp_6d,[rcp_role6,rcp_X_00,Xb_00],[X_00,r3]
rcp_6e,[rcp_role6,rcp_Xb_3,3_00,X_00],[3_00,Xb_00,r3]
rcp_6f,[rcp_role6,rcp_Xb_3,3_00,Xb_00],[3_00,Xb_00,r3]
rcp_6g,[rcp_role6,rcp_Xb_3,3_01,X_01],[3_01,Xb_01,r3]
rcp_6h,[rcp_role6,rcp_Xb_3,3_01,Xb_01],[3_01,Xb_01,r3]
rcp_6i,[rcp_role6,rcp_Xb_3,3_10,X_10],[3_10,Xb_10,r3]
rcp_6j,[rcp_role6,rcp_Xb_3,3_10,Xb_10],[3_10,Xb_10,r3]
rcp_6k,[rcp_role6,rcp_Xb_3,3_11,X_11],[3_11,Xb_11,r3]
rcp_6l,[rcp_role6,rcp_Xb_3,3_11,Xb_11],[3_11,Xb_11,r3]
rcp_6m,[rcp_role6,rcp_X_4,4_00,X_00],[4_00,X_00,r3]
rcp_6n,[rcp_role6,rcp_X_4,4_00,Xb_00],[4_00,X_00,r3]
rcp_6o,[rcp_role6,rcp_X_4,4_01,X_01],[4_01,X_01,r3]
rcp_6p,[rcp_role6,rcp_X_4,4_01,Xb_01],[4_01,X_01,r3]
rcp_6q,[rcp_role6,rcp_X_4,4_10,X_10],[4_10,X_10,r3]
rcp_6r,[rcp_role6,rcp_X_4,4_10,Xb_10],[4_10,X_10,r3]
rcp_6s,[rcp_role6,rcp_X_4,4_11,X_11],[4_11,X_11,r3]
rcp_6t,[rcp_role6,rcp_X_4,4_11,Xb_11],[4_11,X_11,r3]
COM
st_rd_00,[r3,rp=0,ralt=0],[reading_00,rp=0,ralt=0,enable]
end_rd_00,[reading_00,enable],[clear_x,enable]
st_rd_01,[r3,rp=0,ralt=1],[reading_01,rp=0,ralt=1,enable]
end_rd_01,[reading_01,enable],[clear_x,enable]
st_rd_10,[r3,rp=1,ralt=0],[reading_10,rp=1,ralt=0,enable]
end_rd_10,[reading_10,enable],[clear_x,enable]
st_rd_11,[r3,rp=1,ralt=1],[reading_11,rp=1,ralt=1,enable]
end_rd_11,[reading_11,enable],[clear_x,enable]
COM
COM sequence check
COM
order_lost_00,[enable,reading_00,x_00,r0+],[enable,
reading_00,x_00,order_lost,r0+]
order_lost_01,[enable,reading_01,x_01,r0+],[enable,
reading_01,x_01,order_lost,r0+]
order_lost_10,[enable,reading_10,x_10,r0+],[enable,
reading_10,x_10,order_lost,r0+]
order_lost_11,[enable,reading_11,x_11,r0+],[enable,
reading_11,x_11,order_lost,r0+]
END.
```

B.5 Simpson '$4b+7$' Petri net models

B.5.1 Model for testing coherence

```
4 slot v2 (hardware algebraic) improved reader, two stage model on r
bad=[coh_lost]
control=[wr],[writing_00],[writing_01],[writing_10],[writing_11],[w0],[w1],
[r0],[rc=0],[rc=1],[r1],[rd],[reading_00],[reading_01],[reading_10],
[reading_11]
initial=[wr,r0,n=0,s0=0,s1=0,not_writing_00,not_writing_01,not_writing_10,
not_writing_11,l=0,r=0,v0=0,v1=0,not_reading_00,not_reading_01,
not_reading_10,not_reading_11]
wr00b,[wr,n=0,s0=1,not_writing_00],[n=0,s0=1,writing_00]
wr00e,[writing_00],[w0,not_writing_00]
wr01b,[wr,n=0,s0=0,not_writing_01],[n=0,s0=0,writing_01]
wr01e,[writing_01],[w0,not_writing_01]
wr10b,[wr,n=1,s1=1,not_writing_10],[n=1,s1=1,writing_10]
wr10e,[writing_10],[w0,not_writing_10]
wr11b,[wr,n=1,s1=0,not_writing_11],[n=1,s1=0,writing_11]
wr11e,[writing_11],[w0,not_writing_11]
w0s00,[w0,n=0,s0=1],[w1,n=0,s0=0]
w0s01,[w0,n=0,s0=0],[w1,n=0,s0=1]
w0s10,[w0,n=1,s1=1],[w1,n=1,s1=0]
w0s11,[w0,n=1,s1=0],[w1,n=1,s1=1]
w1000,[w1,n=0,l=0,r=0],[wr,n=1,l=0,r=0]
w1001,[w1,n=0,l=0,r=1],[wr,n=0,l=0,r=1]
w1010,[w1,n=1,l=0,r=0],[wr,n=1,l=1,r=0]
w1011,[w1,n=1,l=0,r=1],[wr,n=0,l=1,r=1]
w1100,[w1,n=0,l=1,r=0],[wr,n=1,l=0,r=0]
w1101,[w1,n=0,l=1,r=1],[wr,n=0,l=0,r=1]
w1110,[w1,n=1,l=1,r=0],[wr,n=1,l=1,r=0]
w1111,[w1,n=1,l=1,r=1],[wr,n=0,l=1,r=1]
r0c0,[r0,l=0],[l=0,rc=0]
r0c1,[r0,l=1],[l=1,rc=1]
r0r00,[rc=0,r=0],[r1,r=0]
r0r10,[rc=0,r=1],[r1,r=0]
r0r01,[rc=1,r=0],[r1,r=1]
r0r11,[rc=1,r=1],[r1,r=1]
r10000,[r1,s0=0,s1=0,v0=0,v1=0],[rd,s0=0,s1=0,v0=0,v1=0]
r10001,[r1,s0=0,s1=0,v0=0,v1=1],[rd,s0=0,s1=0,v0=0,v1=0]
r10010,[r1,s0=0,s1=0,v0=1,v1=0],[rd,s0=0,s1=0,v0=0,v1=0]
r10011,[r1,s0=0,s1=0,v0=1,v1=1],[rd,s0=0,s1=0,v0=0,v1=0]
r10100,[r1,s0=0,s1=1,v0=0,v1=0],[rd,s0=0,s1=1,v0=0,v1=1]
r10101,[r1,s0=0,s1=1,v0=0,v1=1],[rd,s0=0,s1=1,v0=0,v1=1]
r10110,[r1,s0=0,s1=1,v0=1,v1=0],[rd,s0=0,s1=1,v0=0,v1=1]
r10111,[r1,s0=0,s1=1,v0=1,v1=1],[rd,s0=0,s1=1,v0=0,v1=1]
r11000,[r1,s0=1,s1=0,v0=0,v1=0],[rd,s0=1,s1=0,v0=1,v1=0]
r11001,[r1,s0=1,s1=0,v0=0,v1=1],[rd,s0=1,s1=0,v0=1,v1=0]
r11010,[r1,s0=1,s1=0,v0=1,v1=0],[rd,s0=1,s1=0,v0=1,v1=0]
r11011,[r1,s0=1,s1=0,v0=1,v1=1],[rd,s0=1,s1=0,v0=1,v1=0]
r11100,[r1,s0=1,s1=1,v0=0,v1=0],[rd,s0=1,s1=1,v0=1,v1=1]
r11101,[r1,s0=1,s1=1,v0=0,v1=1],[rd,s0=1,s1=1,v0=1,v1=1]
r11110,[r1,s0=1,s1=1,v0=1,v1=0],[rd,s0=1,s1=1,v0=1,v1=1]
r11111,[r1,s0=1,s1=1,v0=1,v1=1],[rd,s0=1,s1=1,v0=1,v1=1]
rd00b,[rd,r=0,v0=0,not_reading_00],[r=0,v0=0,reading_00]
rd00e,[reading_00],[r0,not_reading_00]
rd01b,[rd,r=0,v0=1,not_reading_01],[r=0,v0=1,reading_01]
rd01e,[reading_01],[r0,not_reading_01]
rd10b,[rd,r=1,v1=0,not_reading_10],[r=1,v1=0,reading_10]
rd10e,[reading_10],[r0,not_reading_10]
rd11b,[rd,r=1,v1=1,not_reading_11],[r=1,v1=1,reading_11]
rd11e,[reading_11],[r0,not_reading_11]
coh_lost_00,[writing_00,reading_00],[writing_00,reading_00,coh_lost]
coh_lost_01,[writing_01,reading_01],[writing_01,reading_01,coh_lost]
```

```
coh_lost_10,[writing_10,reading_10],[writing_10,reading_10,coh_lost]
coh_lost_11,[writing_11,reading_11],[writing_11,reading_11,coh_lost]
```

B.5.2 Model for testing freshness

```
4 slot v2 (hardware algebraic) freshness, two stage r, new freshness + p
error
bad=[coh_lost],[not_fresh]
control=[wr],[writing_00],[writing_01],[writing_10],[writing_11],[f_to_p],
[w_to_f],[w0],[w1],[w1_post],[clear_prev],[clear_valid],[clear_valid2],
[clear_valid3],[clear_valid4],[f_to_v],[p_to_v],[r0_pre],[r0],[rc=0],[rc=1],
[r1],[pre_read_ctrl],[clear_prev_by_r],[rd],[reading_00],[reading_01],[reading_10],[reading_11]
initial=[wr,clear_valid,no_w1,no_r0,n=0,s0=0,s1=0,not_writing_00,
not_writing_01,not_writing_10,not_writing_11,l=0,r=0,v0=0,v1=0,
not_reading_00,not_reading_01,not_reading_10,not_reading_11,enable,
no_fresh,no_prev,not_valid_00,not_valid_01,not_valid_10,not_valid_11]
wr00b,[wr,n=0,s0=1,not_writing_00,enable],[n=0,s0=1,writing_00,enable]
wr00e,[writing_00,enable],[not_writing_00,f_to_p,wrote_00]
wr01b,[wr,n=0,s0=0,not_writing_01,enable],[n=0,s0=0,writing_01,enable]
wr01e,[writing_01,enable],[not_writing_01,f_to_p,wrote_01]
wr10b,[wr,n=1,s1=1,not_writing_10,enable],[n=1,s1=1,writing_10,enable]
wr10e,[writing_10,enable],[not_writing_10,f_to_p,wrote_10]
wr11b,[wr,n=1,s1=0,not_writing_11,enable],[n=1,s1=0,writing_11,enable]
wr11e,[writing_11,enable],[not_writing_11,f_to_p,wrote_11]
nftop,[f_to_p,no_fresh],[w_to_f,no_fresh]
ftop00,[fresh_00,no_prev,f_to_p],[no_fresh,prev_00,w_to_f]
ftop01,[fresh_01,no_prev,f_to_p],[no_fresh,prev_01,w_to_f]
ftop10,[fresh_10,no_prev,f_to_p],[no_fresh,prev_10,w_to_f]
ftop11,[fresh_11,no_prev,f_to_p],[no_fresh,prev_11,w_to_f]
wtofv00b,[wrote_00,valid_00,w_to_f,no_fresh],[w0,enable,fresh_00,valid_00]
wtofv00c,[wrote_00,not_valid_00,w_to_f,no_fresh],[w0,enable,fresh_00,valid_00]
wtofv01b,[wrote_01,valid_01,w_to_f,no_fresh],[w0,enable,fresh_01,valid_01]
wtofv01c,[wrote_01,not_valid_01,w_to_f,no_fresh],[w0,enable,fresh_01,valid_01]
wtofv10b,[wrote_10,valid_10,w_to_f,no_fresh],[w0,enable,fresh_10,valid_10]
wtofv10c,[wrote_10,not_valid_10,w_to_f,no_fresh],[w0,enable,fresh_10,valid_10]
wtofv11b,[wrote_11,valid_11,w_to_f,no_fresh],[w0,enable,fresh_11,valid_11]
wtofv11c,[wrote_11,not_valid_11,w_to_f,no_fresh],[w0,enable,fresh_11,valid_11]
w0s00,[w0,n=0,s0=1,enable],[w1,n=0,s0=0,enable]
w0s01,[w0,n=0,s0=0,enable],[w1,n=0,s0=1,enable]
w0s10,[w0,n=1,s1=1,enable],[w1,n=1,s1=0,enable]
w0s11,[w0,n=1,s1=0,enable],[w1,n=1,s1=1,enable]
w1000,[w1,n=0,l=0,r=0,enable],[n=1,l=0,r=0,w1_post]
w1001,[w1,n=0,l=0,r=1,enable],[n=0,l=0,r=1,w1_post]
w1010,[w1,n=1,l=0,r=0,enable],[n=1,l=1,r=0,w1_post]
w1011,[w1,n=1,l=0,r=1,enable],[n=0,l=1,r=1,w1_post]
w1100,[w1,n=0,l=1,r=0,enable],[n=1,l=0,r=0,w1_post]
w1101,[w1,n=0,l=1,r=1,enable],[n=0,l=0,r=1,w1_post]
w1110,[w1,n=1,l=1,r=0,enable],[n=1,l=1,r=0,w1_post]
w1111,[w1,n=1,l=1,r=1,enable],[n=0,l=1,r=1,w1_post]
w1_initial,[w1_post,no_w1],[clear_prev,w1+]
w1_normal,[w1_post,w1+],[clear_prev,w1+]
ncp,[no_prev,clear_prev],[wr,enable,no_prev]
cp00,[prev_00,clear_prev],[wr,enable,no_prev]
cp01,[prev_01,clear_prev],[wr,enable,no_prev]
cp10,[prev_10,clear_prev],[wr,enable,no_prev]
cp11,[prev_11,clear_prev],[wr,enable,no_prev]
clear_valid_00a,[not_valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_00b,[valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_01a,[not_valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_01b,[valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_10a,[not_valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_10b,[valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_11a,[not_valid_11,clear_valid4],[not_valid_11,f_to_v]
clear_valid_11b,[valid_11,clear_valid4],[not_valid_11,f_to_v]
```

```
nftov,[f_to_v,no_fresh,not_valid_00,not_valid_01,not_valid_10,not_valid_11],
[no_fresh,valid_00,valid_01,valid_10,valid_11,r0_pre]
ftov00,[f_to_v,fresh_00,not_valid_00],[fresh_00,valid_00,p_to_v]
ftov01,[f_to_v,fresh_01,not_valid_01],[fresh_01,valid_01,p_to_v]
ftov10,[f_to_v,fresh_10,not_valid_10],[fresh_10,valid_10,p_to_v]
ftov11,[f_to_v,fresh_11,not_valid_11],[fresh_11,valid_11,p_to_v]
nptov,[no_prev,p_to_v],[no_prev,r0_pre]
ptov00,[prev_00,not_valid_00,p_to_v],[prev_00,valid_00,r0_pre]
ptov01,[prev_01,not_valid_01,p_to_v],[prev_01,valid_01,r0_pre]
ptov10,[prev_10,not_valid_10,p_to_v],[prev_10,valid_10,r0_pre]
ptov11,[prev_11,not_valid_11,p_to_v],[prev_11,valid_11,r0_pre]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
rc0,[r0,l=0],[l=0,rc=0,enable]
rc1,[r0,l=1],[l=1,rc=1,enable]
r0r00,[rc=0,r=0,enable],[r1,r=0,enable]
r0r10,[rc=0,r=1,enable],[r1,r=0,enable]
r0r01,[rc=1,r=0,enable],[r1,r=1,enable]
r0r11,[rc=1,r=1,enable],[r1,r=1,enable]
r10000,[r1,s0=0,s1=0,v0=0,v1=0,enable],[pre_read_ctrl,s0=0,s1=0,v0=0,v1=0,
enable]
r10001,[r1,s0=0,s1=0,v0=0,v1=1,enable],[pre_read_ctrl,s0=0,s1=0,v0=0,v1=0,
enable]
r10010,[r1,s0=0,s1=0,v0=1,v1=0,enable],[pre_read_ctrl,s0=0,s1=0,v0=0,v1=0,
enable]
r10011,[r1,s0=0,s1=0,v0=1,v1=1,enable],[pre_read_ctrl,s0=0,s1=0,v0=0,v1=0,
enable]
r10100,[r1,s0=0,s1=1,v0=0,v1=0,enable],[pre_read_ctrl,s0=0,s1=1,v0=0,v1=1,
enable]
r10101,[r1,s0=0,s1=1,v0=0,v1=1,enable],[pre_read_ctrl,s0=0,s1=1,v0=0,v1=1,
enable]
r10110,[r1,s0=0,s1=1,v0=1,v1=0,enable],[pre_read_ctrl,s0=0,s1=1,v0=0,v1=1,
enable]
r10111,[r1,s0=0,s1=1,v0=1,v1=1,enable],[pre_read_ctrl,s0=0,s1=1,v0=0,v1=1,
enable]
r11000,[r1,s0=1,s1=0,v0=0,v1=0,enable],[pre_read_ctrl,s0=1,s1=0,v0=1,v1=0,
enable]
r11001,[r1,s0=1,s1=0,v0=0,v1=1,enable],[pre_read_ctrl,s0=1,s1=0,v0=1,v1=0,
enable]
r11010,[r1,s0=1,s1=0,v0=1,v1=0,enable],[pre_read_ctrl,s0=1,s1=0,v0=1,v1=0,
enable]
r11011,[r1,s0=1,s1=0,v0=1,v1=1,enable],[pre_read_ctrl,s0=1,s1=0,v0=1,v1=0,
enable]
r11100,[r1,s0=1,s1=1,v0=0,v1=0,enable],[pre_read_ctrl,s0=1,s1=1,v0=1,v1=1,
enable]
r11101,[r1,s0=1,s1=1,v0=0,v1=1,enable],[pre_read_ctrl,s0=1,s1=1,v0=1,v1=1,
enable]
r11110,[r1,s0=1,s1=1,v0=1,v1=0,enable],[pre_read_ctrl,s0=1,s1=1,v0=1,v1=1,
enable]
r11111,[r1,s0=1,s1=1,v0=1,v1=1,enable],[pre_read_ctrl,s0=1,s1=1,v0=1,v1=1,
enable]
pre_read_00a,[pre_read_ctrl,enable,r=0,v0=0,no_fresh],[no_fresh,rd,r=0,v0=0]
pre_read_00b,[pre_read_ctrl,enable,r=0,v0=0,fresh_00],[fresh_00,
clear_prev_by_r,r=0,v0=0]
pre_read_00c,[pre_read_ctrl,enable,r=0,v0=0,fresh_01],[fresh_01,rd,r=0,v0=0]
pre_read_00d,[pre_read_ctrl,enable,r=0,v0=0,fresh_10],[fresh_10,rd,r=0,v0=0]
pre_read_00e,[pre_read_ctrl,enable,r=0,v0=0,fresh_11],[fresh_11,rd,r=0,v0=0]
pre_read_01a,[pre_read_ctrl,enable,r=0,v0=1,no_fresh],[no_fresh,rd,r=0,v0=1]
pre_read_01b,[pre_read_ctrl,enable,r=0,v0=1,fresh_00],[fresh_00,rd,r=0,v0=1]
pre_read_01c,[pre_read_ctrl,enable,r=0,v0=1,fresh_01],[fresh_01,
clear_prev_by_r,r=0,v0=1]
pre_read_01d,[pre_read_ctrl,enable,r=0,v0=1,fresh_10],[fresh_10,rd,r=0,v0=1]
pre_read_01e,[pre_read_ctrl,enable,r=0,v0=1,fresh_11],[fresh_11,rd,r=0,v0=1]
pre_read_10a,[pre_read_ctrl,enable,r=1,v1=0,no_fresh],[no_fresh,rd,r=1,v1=0]
pre_read_10b,[pre_read_ctrl,enable,r=1,v1=0,fresh_00],[fresh_00,rd,r=1,v1=0]
```

```
pre_read_10c,[pre_read_ctrl,enable,r=1,v1=0,fresh_01],[fresh_01,rd,r=1,v1=0]
pre_read_10d,[pre_read_ctrl,enable,r=1,v1=0,fresh_10],[fresh_10,
clear_prev_by_r,r=1,v1=0]
pre_read_10e,[pre_read_ctrl,enable,r=1,v1=0,fresh_11],[fresh_11,rd,r=1,v1=0]
pre_read_11a,[pre_read_ctrl,enable,r=1,v1=1,no_fresh],[no_fresh,rd,r=1,v1=1]
pre_read_11b,[pre_read_ctrl,enable,r=1,v1=1,fresh_00],[fresh_00,rd,r=1,v1=1]
pre_read_11c,[pre_read_ctrl,enable,r=1,v1=1,fresh_01],[fresh_01,rd,r=1,v1=1]
pre_read_11d,[pre_read_ctrl,enable,r=1,v1=1,fresh_10],[fresh_10,rd,r=1,v1=1]
pre_read_11e,[pre_read_ctrl,enable,r=1,v1=1,fresh_11],[fresh_11,
clear_prev_by_r,r=1,v1=1]
ncp_r,[no_prev,clear_prev_by_r],[rd,no_prev]
cp00_r,[prev_00,clear_prev_by_r],[rd,no_prev]
cp01_r,[prev_01,clear_prev_by_r],[rd,no_prev]
cp10_r,[prev_10,clear_prev_by_r],[rd,no_prev]
cp11_r,[prev_11,clear_prev_by_r],[rd,no_prev]
rd00b,[rd,r=0,v0=0,not_reading_00],[r=0,v0=0,reading_00,enable]
rd00e,[reading_00,enable],[not_reading_00,clear_valid,enable]
rd01b,[rd,r=0,v0=1,not_reading_01],[r=0,v0=1,reading_01,enable]
rd01e,[reading_01,enable],[not_reading_01,clear_valid,enable]
rd10b,[rd,r=1,v1=0,not_reading_10],[r=1,v1=0,reading_10,enable]
rd10e,[reading_10,enable],[not_reading_10,clear_valid,enable]
rd11b,[rd,r=1,v1=1,not_reading_11],[r=1,v1=1,reading_11,enable]
rd11e,[reading_11,enable],[not_reading_11,clear_valid,enable]
coh_lost_00,[writing_00,reading_00],[writing_00,reading_00,coh_lost]
coh_lost_01,[writing_01,reading_01],[writing_01,reading_01,coh_lost]
coh_lost_10,[writing_10,reading_10],[writing_10,reading_10,coh_lost]
coh_lost_11,[writing_11,reading_11],[writing_11,reading_11,coh_lost]
not_fresh_00,[enable,reading_00,not_valid_00,r0+],[enable,reading_00,
not_valid_00,not_fresh,r0+]
not_fresh_01,[enable,reading_01,not_valid_01,r0+],[enable,reading_01,
not_valid_01,not_fresh,r0+]
not_fresh_10,[enable,reading_10,not_valid_10,r0+],[enable,reading_10,
not_valid_10,not_fresh,r0+]
not_fresh_11,[enable,reading_11,not_valid_11,r0+],[enable,reading_11,
not_valid_11,not_fresh,r0+]
```

B.5.3 Model for testing sequence

```
4 slot (Simpsons old version) checking new-old-new
bad=[order_lost]
initial=[enable,wr,clear_x,no_w1,no_r0,n=0,s0=0,s1=0,not_writing_00,
not_writing_01,not_writing_10,not_writing_11,l=0,r=0,v0=0,v1=0,
not_reading_00,not_reading_01,not_reading_10,not_reading_11,Xb_00,Xb_01,
Xb_10,Xb_11,x_00,x_01,x_10,x_11,4_00,4_01,4_10,4_11,not1,not2,not3]
wr00b,[enable,wr,n=0,s0=1,not_writing_00],[enable,n=0,s0=1,writing_00]
wr00e,[enable,writing_00],[wip_roles00,not_writing_00]
wr01b,[enable,wr,n=0,s0=0,not_writing_01],[enable,n=0,s0=0,writing_01]
wr01e,[enable,writing_01],[wip_roles01,not_writing_01]
wr10b,[enable,wr,n=1,s1=1,not_writing_10],[enable,n=1,s1=1,writing_10]
wr10e,[enable,writing_10],[wip_roles10,not_writing_10]
wr11b,[enable,wr,n=1,s1=0,not_writing_11],[enable,n=1,s1=0,writing_11]
wr11e,[enable,writing_11],[wip_roles11,not_writing_11]
wip1a,[wip_roles00,X_00],[wip_role200,Xb_00]
wip1b,[wip_roles00,Xb_00],[wip_role200,Xb_00]
wip1c,[wip_roles01,X_01],[wip_role201,Xb_01]
wip1d,[wip_roles01,Xb_01],[wip_role201,Xb_01]
wip1e,[wip_roles10,X_10],[wip_role210,Xb_10]
wip1f,[wip_roles10,Xb_10],[wip_role210,Xb_10]
wip1g,[wip_roles11,X_11],[wip_role211,Xb_11]
wip1h,[wip_roles11,Xb_11],[wip_role211,Xb_11]
wip2a,[wip_role200,1_00],[1_00,w0,enable]
wip2b,[wip_role201,1_01],[1_01,w0,enable]
wip2c,[wip_role210,1_10],[1_10,w0,enable]
wip2d,[wip_role211,1_11],[1_11,w0,enable]
```

```
wip2e,[wip_role200,2_00],[w2b1_00,11b2,2_00]
wip2f,[wip_role201,2_01],[w2b1_01,11b2,2_01]
wip2g,[wip_role210,2_10],[w2b1_10,11b2,2_10]
wip2h,[wip_role211,2_11],[w2b1_11,11b2,2_11]
wip2i,[wip_role200,3_00],[w3b1_00,22b3,3_00]
wip2j,[wip_role201,3_01],[w3b1_01,22b3,3_01]
wip2k,[wip_role210,3_10],[w3b1_10,22b3,3_10]
wip2l,[wip_role211,3_11],[w3b1_11,22b3,3_11]
wip2m,[wip_role200,4_00],[w4b1_00,33b4,4_00]
wip2n,[wip_role201,4_01],[w4b1_01,33b4,4_01]
wip2o,[wip_role210,4_10],[w4b1_10,33b4,4_10]
wip2p,[wip_role211,4_11],[w4b1_11,33b4,4_11]
wip3a,[33b4,3_00],[22b3,4_00]
wip3b,[33b4,3_01],[22b3,4_01]
wip3c,[33b4,3_10],[22b3,4_10]
wip3d,[33b4,3_11],[22b3,4_11]
wip3e,[33b4,not3],[22b3,not3]
wip4a,[22b3,2_00,not3],[11b2,3_00,not3b]
wip4aa,[22b3,2_00,not3b],[11b2,3_00,not3b]
wip4b,[22b3,2_01,not3],[11b2,3_01,not3b]
wip4ba,[22b3,2_01,not3b],[11b2,3_01,not3b]
wip4c,[22b3,2_10,not3],[11b2,3_10,not3b]
wip4ca,[22b3,2_10,not3b],[11b2,3_10,not3b]
wip4d,[22b3,2_11,not3],[11b2,3_11,not3b]
wip4da,[22b3,2_11,not3b],[11b2,3_11,not3b]
wip4e,[22b3,not2],[11b2,not2]
wip5a,[11b2,1_00,not2],[11b2_fin,2_00,not2b]
wip5aa,[11b2,1_00,not2b],[11b2_fin,2_00,not2b]
wip5b,[11b2,1_01,not2],[11b2_fin,2_01,not2b]
wip5ba,[11b2,1_01,not2b],[11b2_fin,2_01,not2b]
wip5c,[11b2,1_10,not2],[11b2_fin,2_10,not2b]
wip5ca,[11b2,1_10,not2b],[11b2_fin,2_10,not2b]
wip5d,[11b2,1_11,not2],[11b2_fin,2_11,not2b]
wip5da,[11b2,1_11,not2b],[11b2_fin,2_11,not2b]
wip5e,[11b2,not1],[11b2_fin,not1]
wip6a,[11b2_fin,w2b1_00,2_00],[1_00,w0,enable]
wip6b,[11b2_fin,w2b1_01,2_01],[1_01,w0,enable]
wip6c,[11b2_fin,w2b1_10,2_10],[1_10,w0,enable]
wip6d,[11b2_fin,w2b1_11,2_11],[1_11,w0,enable]
wip6e,[11b2_fin,w3b1_00,3_00],[1_00,w0,enable]
wip6f,[11b2_fin,w3b1_01,3_01],[1_01,w0,enable]
wip6g,[11b2_fin,w3b1_10,3_10],[1_10,w0,enable]
wip6h,[11b2_fin,w3b1_11,3_11],[1_11,w0,enable]
wip6i,[11b2_fin,w4b1_00,4_00,not1],[1_00,w0,enable,not1b]
wip6ia,[11b2_fin,w4b1_00,4_00,not1b],[1_00,w0,enable,not1b]
wip6j,[11b2_fin,w4b1_01,4_01,not1],[1_01,w0,enable,not1b]
wip6ja,[11b2_fin,w4b1_01,4_01,not1b],[1_01,w0,enable,not1b]
wip6k,[11b2_fin,w4b1_10,4_10,not1],[1_10,w0,enable,not1b]
wip6ka,[11b2_fin,w4b1_10,4_10,not1b],[1_10,w0,enable,not1b]
wip6l,[11b2_fin,w4b1_11,4_11,not1],[1_11,w0,enable,not1b]
wip6la,[11b2_fin,w4b1_11,4_11,not1b],[1_11,w0,enable,not1b]
w0s00,[enable,w0,n=0,s0=1],[enable,w1,n=0,s0=0]
w0s01,[enable,w0,n=0,s0=0],[enable,w1,n=0,s0=1]
w0s10,[enable,w0,n=1,s1=1],[enable,w1,n=1,s1=0]
w0s11,[enable,w0,n=1,s1=0],[enable,w1,n=1,s1=1]
w1000,[enable,w1,n=0,l=0,r=0],[w1_post,n=1,l=0,r=0]
w1001,[enable,w1,n=0,l=0,r=1],[w1_post,n=0,l=0,r=1]
w1010,[enable,w1,n=1,l=0,r=0],[w1_post,n=1,l=1,r=0]
w1011,[enable,w1,n=1,l=0,r=1],[w1_post,n=0,l=1,r=1]
w1100,[enable,w1,n=0,l=1,r=0],[w1_post,n=1,l=0,r=0]
w1101,[enable,w1,n=0,l=1,r=1],[w1_post,n=0,l=0,r=1]
w1110,[enable,w1,n=1,l=1,r=0],[w1_post,n=1,l=1,r=0]
w1111,[enable,w1,n=1,l=1,r=1],[w1_post,n=0,l=1,r=1]
w1_initial,[w1_post,no_w1],[wr,w1+,enable]
w1_normal,[w1_post,w1+],[wr,w1+,enable]
clr_x00,[enable,clear_x,x_00],[clear_x2,xb_00]
```

```
clr_x00a,[enable,clear_x,xb_00],[clear_x2,xb_00]
clr_x01,[clear_x2,x_01],[clear_x3,xb_01]
clr_x01a,[clear_x2,xb_01],[clear_x3,xb_01]
clr_x02,[clear_x3,x_10],[clear_x4,xb_10]
clr_x02a,[clear_x3,xb_10],[clear_x4,xb_10]
clr_x03,[clear_x4,x_11],[r0_pre,xb_11]
clr_x03a,[clear_x4,xb_11],[r0_pre,xb_11]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0c0,[r0,l=0],[enable,l=0,rc=0]
r0c1,[r0,l=1],[enable,l=1,rc=1]
r0r00,[enable,rc=0,r=0],[enable,r1,r=0]
r0r10,[enable,rc=0,r=1],[enable,r1,r=0]
r0r01,[enable,rc=1,r=0],[enable,r1,r=1]
r0r11,[enable,rc=1,r=1],[enable,r1,r=1]
r10000,[enable,r1,s0=0,s1=0,v0=0,v1=0],[enable,rcp_roles,s0=0,s1=0,v0=0,v1=0]
r10001,[enable,r1,s0=0,s1=0,v0=0,v1=1],[enable,rcp_roles,s0=0,s1=0,v0=0,v1=0]
r10010,[enable,r1,s0=0,s1=0,v0=1,v1=0],[enable,rcp_roles,s0=0,s1=0,v0=0,v1=0]
r10011,[enable,r1,s0=0,s1=0,v0=1,v1=1],[enable,rcp_roles,s0=0,s1=0,v0=0,v1=0]
r10100,[enable,r1,s0=0,s1=1,v0=0,v1=0],[enable,rcp_roles,s0=0,s1=1,v0=0,v1=1]
r10101,[enable,r1,s0=0,s1=1,v0=0,v1=1],[enable,rcp_roles,s0=0,s1=1,v0=0,v1=1]
r10110,[enable,r1,s0=0,s1=1,v0=1,v1=0],[enable,rcp_roles,s0=0,s1=1,v0=0,v1=1]
r10111,[enable,r1,s0=0,s1=1,v0=1,v1=1],[enable,rcp_roles,s0=0,s1=1,v0=0,v1=1]
r11000,[enable,r1,s0=1,s1=0,v0=0,v1=0],[enable,rcp_roles,s0=1,s1=0,v0=1,v1=0]
r11001,[enable,r1,s0=1,s1=0,v0=0,v1=1],[enable,rcp_roles,s0=1,s1=0,v0=1,v1=0]
r11010,[enable,r1,s0=1,s1=0,v0=1,v1=0],[enable,rcp_roles,s0=1,s1=0,v0=1,v1=0]
r11011,[enable,r1,s0=1,s1=0,v0=1,v1=1],[enable,rcp_roles,s0=1,s1=0,v0=1,v1=0]
r11100,[enable,r1,s0=1,s1=1,v0=0,v1=0],[enable,rcp_roles,s0=1,s1=1,v0=1,v1=1]
r11101,[enable,r1,s0=1,s1=1,v0=0,v1=1],[enable,rcp_roles,s0=1,s1=1,v0=1,v1=1]
r11110,[enable,r1,s0=1,s1=1,v0=1,v1=0],[enable,rcp_roles,s0=1,s1=1,v0=1,v1=1]
r11111,[enable,r1,s0=1,s1=1,v0=1,v1=1],[enable,rcp_roles,s0=1,s1=1,v0=1,v1=1]
x_r00a,[enable,rcp_roles,v0=0,r=0,X_00,xb_00],[v0=0,r=0,X_00,x_00,
rcp_roles2_00]
x_r00b,[enable,rcp_roles,v0=0,r=0,Xb_00],[v0=0,r=0,Xb_00,rcp_roles2_00]
x_r01a,[enable,rcp_roles,v0=1,r=0,X_01,xb_01],[v0=1,r=0,X_01,x_01,
rcp_roles2_01]
x_r01b,[enable,rcp_roles,v0=1,r=0,Xb_01],[v0=1,r=0,Xb_01,rcp_roles2_01]
x_r10a,[enable,rcp_roles,v1=0,r=1,X_10,xb_10],[v1=0,r=1,X_10,x_10,
rcp_roles2_10]
x_r10b,[enable,rcp_roles,v1=0,r=1,Xb_10],[v1=0,r=1,Xb_10,rcp_roles2_10]
x_r11a,[enable,rcp_roles,v1=1,r=1,X_11,xb_11],[v1=1,r=1,X_11,x_11,
rcp_roles2_11]
x_r11b,[enable,rcp_roles,v1=1,r=1,Xb_11],[v1=1,r=1,Xb_11,rcp_roles2_11]
X_r00a,[rcp_roles2_00,1_00],[1_00,rcp_Xb_00,rcp_X_01,rcp_X_10,rcp_X_11,
rcp_role3]
X_r00b,[rcp_roles2_00,2_00],[2_00,rcp_Xb_00,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r00c,[rcp_roles2_00,3_00],[3_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r00d,[rcp_roles2_00,4_00],[4_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r01a,[rcp_roles2_01,1_01],[1_01,rcp_Xb_01,rcp_X_00,rcp_X_10,rcp_X_11,
rcp_role3]
X_r01b,[rcp_roles2_01,2_01],[2_01,rcp_Xb_01,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r01c,[rcp_roles2_01,3_01],[3_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r01d,[rcp_roles2_01,4_01],[4_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r10a,[rcp_roles2_10,1_10],[1_10,rcp_Xb_10,rcp_X_01,rcp_X_00,rcp_X_11,
rcp_role3]
X_r10b,[rcp_roles2_10,2_10],[2_10,rcp_Xb_10,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r10c,[rcp_roles2_10,3_10],[3_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
```

```
X_r10d,[rcp_roles2_10,4_10],[4_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r11a,[rcp_roles2_11,1_11],[1_11,rcp_Xb_11,rcp_X_01,rcp_X_10,rcp_X_00,
rcp_role3]
X_r11b,[rcp_roles2_11,2_11],[2_11,rcp_Xb_11,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r11c,[rcp_roles2_11,3_11],[3_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r11d,[rcp_roles2_11,4_11],[4_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_00,X_00],[Xb_00,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_00,Xb_00],[Xb_00,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_01,X_01],[Xb_01,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_01,Xb_01],[Xb_01,rcp_role4]
rcp_3e,[rcp_role3,rcp_Xb_10,X_10],[Xb_10,rcp_role4]
rcp_3f,[rcp_role3,rcp_Xb_10,Xb_10],[Xb_10,rcp_role4]
rcp_3g,[rcp_role3,rcp_Xb_11,X_11],[Xb_11,rcp_role4]
rcp_3h,[rcp_role3,rcp_Xb_11,Xb_11],[Xb_11,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_01,X_01],[X_01,rcp_role5]
rcp_4b,[rcp_role4,rcp_X_01,Xb_01],[X_01,rcp_role5]
rcp_4c,[rcp_role4,rcp_X_00,Xb_00],[X_00,rcp_role5]
rcp_4d,[rcp_role4,rcp_X_00,X_00],[X_00,rcp_role5]
rcp_4e,[rcp_role4,rcp_Xb_1,1_00,X_00],[1_00,Xb_00,rcp_role5]
rcp_4f,[rcp_role4,rcp_Xb_1,1_00,Xb_00],[1_00,Xb_00,rcp_role5]
rcp_4g,[rcp_role4,rcp_Xb_1,1_01,X_01],[1_01,Xb_01,rcp_role5]
rcp_4h,[rcp_role4,rcp_Xb_1,1_01,Xb_01],[1_01,Xb_01,rcp_role5]
rcp_4i,[rcp_role4,rcp_Xb_1,1_10,X_10],[1_10,Xb_10,rcp_role5]
rcp_4j,[rcp_role4,rcp_Xb_1,1_10,Xb_10],[1_10,Xb_10,rcp_role5]
rcp_4k,[rcp_role4,rcp_Xb_1,1_11,X_11],[1_11,Xb_11,rcp_role5]
rcp_4l,[rcp_role4,rcp_Xb_1,1_11,Xb_11],[1_11,Xb_11,rcp_role5]
rcp_5a,[rcp_role5,rcp_X_10,X_10],[X_10,rcp_role6]
rcp_5b,[rcp_role5,rcp_X_10,Xb_10],[X_10,rcp_role6]
rcp_5c,[rcp_role5,rcp_X_00,X_00],[X_00,rcp_role6]
rcp_5d,[rcp_role5,rcp_X_00,Xb_00],[X_00,rcp_role6]
rcp_5e,[rcp_role5,rcp_Xb_2,2_00,X_00],[2_00,Xb_00,rcp_role6]
rcp_5f,[rcp_role5,rcp_Xb_2,2_00,Xb_00],[2_00,Xb_00,rcp_role6]
rcp_5g,[rcp_role5,rcp_Xb_2,2_01,X_01],[2_01,Xb_01,rcp_role6]
rcp_5h,[rcp_role5,rcp_Xb_2,2_01,Xb_01],[2_01,Xb_01,rcp_role6]
rcp_5i,[rcp_role5,rcp_Xb_2,2_10,X_10],[2_10,Xb_10,rcp_role6]
rcp_5j,[rcp_role5,rcp_Xb_2,2_10,Xb_10],[2_10,Xb_10,rcp_role6]
rcp_5k,[rcp_role5,rcp_Xb_2,2_11,X_11],[2_11,Xb_11,rcp_role6]
rcp_5l,[rcp_role5,rcp_Xb_2,2_11,Xb_11],[2_11,Xb_11,rcp_role6]
rcp_5m,[rcp_role5,rcp_X_3,3_00,X_00],[3_00,X_00,rcp_role6]
rcp_5n,[rcp_role5,rcp_X_3,3_00,Xb_00],[3_00,X_00,rcp_role6]
rcp_5o,[rcp_role5,rcp_X_3,3_01,X_01],[3_01,X_01,rcp_role6]
rcp_5p,[rcp_role5,rcp_X_3,3_01,Xb_01],[3_01,X_01,rcp_role6]
rcp_5q,[rcp_role5,rcp_X_3,3_10,X_10],[3_10,X_10,rcp_role6]
rcp_5r,[rcp_role5,rcp_X_3,3_10,Xb_10],[3_10,X_10,rcp_role6]
rcp_5s,[rcp_role5,rcp_X_3,3_11,X_11],[3_11,X_11,rcp_role6]
rcp_5t,[rcp_role5,rcp_X_3,3_11,Xb_11],[3_11,X_11,rcp_role6]
rcp_6a,[rcp_role6,rcp_X_11,X_11],[X_11,rd]
rcp_6b,[rcp_role6,rcp_X_11,Xb_11],[X_11,rd]
rcp_6c,[rcp_role6,rcp_X_00,X_00],[X_00,rd]
rcp_6d,[rcp_role6,rcp_X_00,Xb_00],[X_00,rd]
rcp_6e,[rcp_role6,rcp_Xb_3,3_00,X_00],[3_00,Xb_00,rd]
rcp_6f,[rcp_role6,rcp_Xb_3,3_00,Xb_00],[3_00,Xb_00,rd]
rcp_6g,[rcp_role6,rcp_Xb_3,3_01,X_01],[3_01,Xb_01,rd]
rcp_6h,[rcp_role6,rcp_Xb_3,3_01,Xb_01],[3_01,Xb_01,rd]
rcp_6i,[rcp_role6,rcp_Xb_3,3_10,X_10],[3_10,Xb_10,rd]
rcp_6j,[rcp_role6,rcp_Xb_3,3_10,Xb_10],[3_10,Xb_10,rd]
rcp_6k,[rcp_role6,rcp_Xb_3,3_11,X_11],[3_11,Xb_11,rd]
rcp_6l,[rcp_role6,rcp_Xb_3,3_11,Xb_11],[3_11,Xb_11,rd]
rcp_6m,[rcp_role6,rcp_X_4,4_00,X_00],[4_00,X_00,rd]
rcp_6n,[rcp_role6,rcp_X_4,4_00,Xb_00],[4_00,X_00,rd]
rcp_6o,[rcp_role6,rcp_X_4,4_01,X_01],[4_01,X_01,rd]
rcp_6p,[rcp_role6,rcp_X_4,4_01,Xb_01],[4_01,X_01,rd]
```

```
rcp_6q,[rcp_role6,rcp_X_4,4_10,X_10],[4_10,X_10,rd]
rcp_6r,[rcp_role6,rcp_X_4,4_10,Xb_10],[4_10,X_10,rd]
rcp_6s,[rcp_role6,rcp_X_4,4_11,X_11],[4_11,X_11,rd]
rcp_6t,[rcp_role6,rcp_X_4,4_11,Xb_11],[4_11,X_11,rd]
rd00b,[rd,r=0,v0=0,not_reading_00],[enable,r=0,v0=0,reading_00]
rd00e,[enable,reading_00],[enable,clear_x,not_reading_00]
rd01b,[rd,r=0,v0=1,not_reading_01],[enable,r=0,v0=1,reading_01]
rd01e,[enable,reading_01],[enable,clear_x,not_reading_01]
rd10b,[rd,r=1,v1=0,not_reading_10],[enable,r=1,v1=0,reading_10]
rd10e,[enable,reading_10],[enable,clear_x,not_reading_10]
rd11b,[rd,r=1,v1=1,not_reading_11],[enable,r=1,v1=1,reading_11]
rd11e,[enable,reading_11],[enable,clear_x,not_reading_11]
order_lost_00,[enable,reading_00,x_00,r0+],[enable,reading_00,x_00,
order_lost,r0+]
order_lost_01,[enable,reading_01,x_01,r0+],[enable,reading_01,x_01,
order_lost,r0+]
order_lost_10,[enable,reading_10,x_10,r0+],[enable,reading_10,x_10,
order_lost,r0+]
order_lost_11,[enable,reading_11,x_11,r0+],[enable,reading_11,x_11,
order_lost,r0+]
```

B.6 Simpson '$4b + 6$' Petri net models

B.6.1 Model for testing coherence

```
4 slot (Simpsons new version) improved reader model, 2 stage r0
bad=[coh_lost]
control=[wr],[writing_00],[writing_01],[writing_10],[writing_11],[w0],
[w1],[r0],[r0a],[r1],[rd],[reading_00],[reading_01],[reading_10],[reading_11]
initial=[wr,r0,ip=0,w0=0,w1=0,not_writing_00,not_writing_01,not_writing_10,
not_writing_11,r0=0,r1=0,op=0,not_reading_00,not_reading_01,not_reading_10,
not_reading_11]
wr00b,[wr,ip=0,w0=0,not_writing_00],[ip=0,w0=0,writing_00]
wr00e,[writing_00],[w0,not_writing_00]
wr01b,[wr,ip=0,w0=1,not_writing_01],[ip=0,w0=1,writing_01]
wr01e,[writing_01],[w0,not_writing_01]
wr10b,[wr,ip=1,w1=0,not_writing_10],[ip=1,w1=0,writing_10]
wr10e,[writing_10],[w0,not_writing_10]
wr11b,[wr,ip=1,w1=1,not_writing_11],[ip=1,w1=1,writing_11]
wr11e,[writing_11],[w0,not_writing_11]
w0ip0,[w0,ip=1],[w1,ip=0]
w0ip1,[w0,ip=0],[w1,ip=1]
w10001,[w1,ip=0,w0=0,r0=1],[wr,ip=0,w0=0,r0=1]
w10101,[w1,ip=0,w0=1,r0=1],[wr,ip=0,w0=0,r0=1]
w10010,[w1,ip=0,w0=0,r0=0],[wr,ip=0,w0=1,r0=0]
w10110,[w1,ip=0,w0=1,r0=0],[wr,ip=0,w0=1,r0=0]
w11001,[w1,ip=1,w1=0,r1=1],[wr,ip=1,w1=0,r1=1]
w11101,[w1,ip=1,w1=1,r1=1],[wr,ip=1,w1=0,r1=1]
w11010,[w1,ip=1,w1=0,r1=0],[wr,ip=1,w1=1,r1=0]
w11110,[w1,ip=1,w1=1,r1=0],[wr,ip=1,w1=1,r1=0]
r0a1,[r0,w0=0,w1=0],[r0a,w0=0,w1=0,r0int=0,r1int=0]
r0a2,[r0,w0=0,w1=1],[r0a,w0=0,w1=1,r0int=0,r1int=1]
r0a3,[r0,w0=1,w1=0],[r0a,w0=1,w1=0,r0int=1,r1int=0]
r0a4,[r0,w0=1,w1=1],[r0a,w0=1,w1=1,r0int=1,r1int=1]
r00000b,[r0a,r0int=0,r1int=0,r0=0,r1=0],[r1,r0=0,r1=0]
r00001b,[r0a,r0int=0,r1int=0,r0=0,r1=1],[r1,r0=0,r1=0]
r00010b,[r0a,r0int=0,r1int=0,r0=1,r1=0],[r1,r0=0,r1=0]
r00011b,[r0a,r0int=0,r1int=0,r0=1,r1=1],[r1,r0=0,r1=0]
r00100b,[r0a,r0int=0,r1int=1,r0=0,r1=0],[r1,r0=0,r1=1]
r00101b,[r0a,r0int=0,r1int=1,r0=0,r1=1],[r1,r0=0,r1=1]
r00110b,[r0a,r0int=0,r1int=1,r0=1,r1=0],[r1,r0=0,r1=1]
r00111b,[r0a,r0int=0,r1int=1,r0=1,r1=1],[r1,r0=0,r1=1]
```

```
r01000b,[r0a,r0int=1,r1int=0,r0=0,r1=0],[r1,r0=1,r1=0]
r01001b,[r0a,r0int=1,r1int=0,r0=0,r1=1],[r1,r0=1,r1=0]
r01010b,[r0a,r0int=1,r1int=0,r0=1,r1=0],[r1,r0=1,r1=0]
r01011b,[r0a,r0int=1,r1int=0,r0=1,r1=1],[r1,r0=1,r1=0]
r01100b,[r0a,r0int=1,r1int=1,r0=0,r1=0],[r1,r0=1,r1=1]
r01101b,[r0a,r0int=1,r1int=1,r0=0,r1=1],[r1,r0=1,r1=1]
r01110b,[r0a,r0int=1,r1int=1,r0=1,r1=0],[r1,r0=1,r1=1]
r01111b,[r0a,r0int=1,r1int=1,r0=1,r1=1],[r1,r0=1,r1=1]
r1001,[r1,ip=1,op=0],[rd,ip=1,op=0]
r1101,[r1,ip=1,op=1],[rd,ip=1,op=0]
r1010,[r1,ip=0,op=0],[rd,ip=0,op=1]
r1110,[r1,ip=0,op=1],[rd,ip=0,op=1]
rd00b,[rd,r0=0,op=0,not_reading_00],[r0=0,op=0,reading_00]
rd00e,[reading_00],[r0,not_reading_00]
rd01b,[rd,r0=1,op=0,not_reading_01],[r0=1,op=0,reading_01]
rd01e,[reading_01],[r0,not_reading_01]
rd10b,[rd,r1=0,op=1,not_reading_10],[r1=0,op=1,reading_10]
rd10e,[reading_10],[r0,not_reading_10]
rd11b,[rd,r1=1,op=1,not_reading_11],[r1=1,op=1,reading_11]
rd11e,[reading_11],[r0,not_reading_11]
coh_lost_00,[writing_00,reading_00],[writing_00,reading_00,coh_lost]
coh_lost_01,[writing_01,reading_01],[writing_01,reading_01,coh_lost]
coh_lost_10,[writing_10,reading_10],[writing_10,reading_10,coh_lost]
coh_lost_11,[writing_11,reading_11],[writing_11,reading_11,coh_lost]
```

B.6.2 Model for testing freshness

```
4 slot (Simpsons new version) - freshness, Simpson's new freshness roles
bad=[coh_lost],[not_fresh]
control=[wr],[writing_00],[writing_01],[writing_10],[writing_11],[f_to_p],
[w_to_f],[w0],[w1],[w1_post],[clear_w_prev],[clear_valid],[clear_valid2],
[clear_valid3],[clear_valid4],[clear_r_f],[wf_to_rf],[clear_r_p],[wp_to_rp],
[r0_pre],[r0],[r0a],[r1],[rd],[reading_00],[reading_01],[reading_10],
[reading_11]
initial=[wr,clear_valid,no_w1,no_r0,ip=0,w0=0,w1=0,not_writing_00,
not_writing_01,not_writing_10,not_writing_11,r0=0,r1=0,op=0,not_reading_00,
not_reading_01,not_reading_10,not_reading_11,enable,no_w_fresh,no_w_prev,
not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,not_r_prev_00,
not_r_prev_01,not_r_prev_10,not_r_prev_11,not_valid_00,not_valid_01,
not_valid_10,not_valid_11]
wr00b,[wr,ip=0,w0=0,not_writing_00,enable],[ip=0,w0=0,writing_00,enable]
wr00e,[writing_00,enable],[not_writing_00,f_to_p,wrote_00]
wr01b,[wr,ip=0,w0=1,not_writing_01,enable],[ip=0,w0=1,writing_01,enable]
wr01e,[writing_01,enable],[not_writing_01,f_to_p,wrote_01]
wr10b,[wr,ip=1,w1=0,not_writing_10,enable],[ip=1,w1=0,writing_10,enable]
wr10e,[writing_10,enable],[not_writing_10,f_to_p,wrote_10]
wr11b,[wr,ip=1,w1=1,not_writing_11,enable],[ip=1,w1=1,writing_11,enable]
wr11e,[writing_11,enable],[not_writing_11,f_to_p,wrote_11]
nftop,[f_to_p,no_w_fresh],[w_to_f,no_w_fresh]
ftop00,[w_fresh_00,no_w_prev,f_to_p],[no_w_fresh,w_prev_00,w_to_f]
ftop01,[w_fresh_01,no_w_prev,f_to_p],[no_w_fresh,w_prev_01,w_to_f]
ftop10,[w_fresh_10,no_w_prev,f_to_p],[no_w_fresh,w_prev_10,w_to_f]
ftop11,[w_fresh_11,no_w_prev,f_to_p],[no_w_fresh,w_prev_11,w_to_f]
wtofv00b,[wrote_00,valid_00,w_to_f,no_w_fresh],[w0,enable,w_fresh_00,valid_00]
wtofv00c,[wrote_00,not_valid_00,w_to_f,no_w_fresh],[w0,enable,w_fresh_00,
valid_00]
wtofv01b,[wrote_01,valid_01,w_to_f,no_w_fresh],[w0,enable,w_fresh_01,valid_01]
wtofv01c,[wrote_01,not_valid_01,w_to_f,no_w_fresh],[w0,enable,w_fresh_01,
valid_01]
wtofv10b,[wrote_10,valid_10,w_to_f,no_w_fresh],[w0,enable,w_fresh_10,valid_10]
wtofv10c,[wrote_10,not_valid_10,w_to_f,no_w_fresh],[w0,enable,w_fresh_10,
valid_10]
wtofv11b,[wrote_11,valid_11,w_to_f,no_w_fresh],[w0,enable,w_fresh_11,valid_11]
wtofv11c,[wrote_11,not_valid_11,w_to_f,no_w_fresh],[w0,enable,w_fresh_11,
```

```
valid_11]
w0ip0,[w0,ip=1,enable],[w1,ip=0,enable]
w0ip1,[w0,ip=0,enable],[w1,ip=1,enable]
w10001,[w1,ip=0,w0=0,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10101,[w1,ip=0,w0=1,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10010,[w1,ip=0,w0=0,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w10110,[w1,ip=0,w0=1,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w11001,[w1,ip=1,w1=0,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11101,[w1,ip=1,w1=1,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11010,[w1,ip=1,w1=0,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w11110,[w1,ip=1,w1=1,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w1_initial,[w1_post,no_w1],[clear_w_prev,w1+]
w1_normal,[w1_post,w1+],[clear_w_prev,w1+]
ncp,[no_w_prev,clear_w_prev],[wr,enable,no_w_prev]
cp00,[w_prev_00,clear_w_prev],[wr,enable,no_w_prev]
cp01,[w_prev_01,clear_w_prev],[wr,enable,no_w_prev]
cp10,[w_prev_10,clear_w_prev],[wr,enable,no_w_prev]
cp11,[w_prev_11,clear_w_prev],[wr,enable,no_w_prev]
clear_valid_00a,[not_valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_00b,[valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_01a,[not_valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_01b,[valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_10a,[not_valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_10b,[valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_11a,[not_valid_11,clear_valid4],[not_valid_11,clear_r_f]
clear_valid_11b,[valid_11,clear_valid4],[not_valid_11,clear_r_f]
no_rf,[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,clear_r_f],
[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,wf_to_rf]
clear_rf00,[r_fresh_00,clear_r_f],[not_r_fresh_00,wf_to_rf]
clear_rf01,[r_fresh_01,clear_r_f],[not_r_fresh_01,wf_to_rf]
clear_rf10,[r_fresh_10,clear_r_f],[not_r_fresh_10,wf_to_rf]
clear_rf11,[r_fresh_11,clear_r_f],[not_r_fresh_11,wf_to_rf]
no_wf,[no_w_fresh,wf_to_rf,not_valid_00,not_valid_01,not_valid_10,
not_valid_11],[no_w_fresh,r0_pre,valid_00,valid_01,valid_10,valid_11]
wf_to_rf_00,[w_fresh_00,not_r_fresh_00,wf_to_rf],[w_fresh_00,r_fresh_00,
clear_r_p]
wf_to_rf_01,[w_fresh_01,not_r_fresh_01,wf_to_rf],[w_fresh_01,r_fresh_01,
clear_r_p]
wf_to_rf_10,[w_fresh_10,not_r_fresh_10,wf_to_rf],[w_fresh_10,r_fresh_10,
clear_r_p]
wf_to_rf_11,[w_fresh_11,not_r_fresh_11,wf_to_rf],[w_fresh_11,r_fresh_11,
clear_r_p]
no_rp,[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,clear_r_p],
[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,wp_to_rp]
clear_rp00,[r_prev_00,clear_r_p],[not_r_prev_00,wp_to_rp]
clear_rp01,[r_prev_01,clear_r_p],[not_r_prev_01,wp_to_rp]
clear_rp10,[r_prev_10,clear_r_p],[not_r_prev_10,wp_to_rp]
clear_rp11,[r_prev_11,clear_r_p],[not_r_prev_11,wp_to_rp]
no_wp,[no_w_prev,wp_to_rp],[no_w_prev,r0_pre]
wp_to_rp_00,[w_prev_00,not_r_prev_00,wp_to_rp],[w_prev_00,r_prev_00,r0_pre]
wp_to_rp_01,[w_prev_01,not_r_prev_01,wp_to_rp],[w_prev_01,r_prev_01,r0_pre]
wp_to_rp_10,[w_prev_10,not_r_prev_10,wp_to_rp],[w_prev_10,r_prev_10,r0_pre]
wp_to_rp_11,[w_prev_11,not_r_prev_11,wp_to_rp],[w_prev_11,r_prev_11,r0_pre]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0a1,[r0,w0=0,w1=0],[r0a,w0=0,w1=0,r0int=0,r1int=0,enable]
r0a2,[r0,w0=0,w1=1],[r0a,w0=0,w1=1,r0int=0,r1int=1,enable]
r0a3,[r0,w0=1,w1=0],[r0a,w0=1,w1=0,r0int=1,r1int=0,enable]
r0a4,[r0,w0=1,w1=1],[r0a,w0=1,w1=1,r0int=1,r1int=1,enable]
r00000b,[r0a,r0int=0,r1int=0,r0=0,r1=0,enable],[r1,r0=0,r1=0,enable]
r00001b,[r0a,r0int=0,r1int=0,r0=0,r1=1,enable],[r1,r0=0,r1=0,enable]
r00010b,[r0a,r0int=0,r1int=0,r0=1,r1=0,enable],[r1,r0=0,r1=0,enable]
r00011b,[r0a,r0int=0,r1int=0,r0=1,r1=1,enable],[r1,r0=0,r1=0,enable]
r00100b,[r0a,r0int=0,r1int=1,r0=0,r1=0,enable],[r1,r0=0,r1=1,enable]
r00101b,[r0a,r0int=0,r1int=1,r0=0,r1=1,enable],[r1,r0=0,r1=1,enable]
```

```
r00110b,[r0a,r0int=0,r1int=1,r0=1,r1=0,enable],[r1,r0=0,r1=1,enable]
r00111b,[r0a,r0int=0,r1int=1,r0=1,r1=1,enable],[r1,r0=0,r1=1,enable]
r01000b,[r0a,r0int=1,r1int=0,r0=0,r1=0,enable],[r1,r0=1,r1=0,enable]
r01001b,[r0a,r0int=1,r1int=0,r0=0,r1=1,enable],[r1,r0=1,r1=0,enable]
r01010b,[r0a,r0int=1,r1int=0,r0=1,r1=0,enable],[r1,r0=1,r1=0,enable]
r01011b,[r0a,r0int=1,r1int=0,r0=1,r1=1,enable],[r1,r0=1,r1=0,enable]
r01100b,[r0a,r0int=1,r1int=1,r0=0,r1=0,enable],[r1,r0=1,r1=1,enable]
r01101b,[r0a,r0int=1,r1int=1,r0=0,r1=1,enable],[r1,r0=1,r1=1,enable]
r01110b,[r0a,r0int=1,r1int=1,r0=1,r1=0,enable],[r1,r0=1,r1=1,enable]
r01111b,[r0a,r0int=1,r1int=1,r0=1,r1=1,enable],[r1,r0=1,r1=1,enable]
r1001,[r1,ip=1,op=0,enable],[rd,ip=1,op=0,enable]
r1101,[r1,ip=1,op=1,enable],[rd,ip=1,op=0,enable]
r1010,[r1,ip=0,op=0,enable],[rd,ip=0,op=1,enable]
r1110,[r1,ip=0,op=1,enable],[rd,ip=0,op=1,enable]
rd00b,[enable,rd,r0=0,op=0,not_reading_00],[r0=0,op=0,reading_00,enable]
rd00e,[reading_00,enable],[not_reading_00,clear_valid,enable]
rd01b,[enable,rd,r0=1,op=0,not_reading_01],[r0=1,op=0,reading_01,enable]
rd01e,[reading_01,enable],[not_reading_01,clear_valid,enable]
rd10b,[enable,rd,r1=0,op=1,not_reading_10],[r1=0,op=1,reading_10,enable]
rd10e,[reading_10,enable],[not_reading_10,clear_valid,enable]
rd11b,[enable,rd,r1=1,op=1,not_reading_11],[r1=1,op=1,reading_11,enable]
rd11e,[reading_11,enable],[not_reading_11,clear_valid,enable]
coh_lost_00,[writing_00,reading_00],[writing_00,reading_00,coh_lost]
coh_lost_01,[writing_01,reading_01],[writing_01,reading_01,coh_lost]
coh_lost_10,[writing_10,reading_10],[writing_10,reading_10,coh_lost]
coh_lost_11,[writing_11,reading_11],[writing_11,reading_11,coh_lost]
not_fresh_00,[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,
r0+],[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,not_fresh,
r0+]
not_fresh_01,[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,
r0+],[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,not_fresh,
r0+]
not_fresh_10,[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,
r0+],[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,not_fresh,
r0+]
not_fresh_11,[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,
r0+],[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,not_fresh,
r0+]
```

B.6.3 Model for testing modified freshness

```
4 slot (Simpsons new version) - freshness, Simpson's newer freshness roles,
with clear p by reader.
bad=[coh_lost],[not_fresh]
control=[wr],[writing_00],[writing_01],[writing_10],[writing_11],[f_to_p],
[w_to_f],[w0],[w1],[w1_post],[clear_w_prev],[clear_valid],[clear_valid2],
[clear_valid3],[clear_valid4],[clear_r_f],[wf_to_rf],[clear_r_p],[wp_to_rp],
[r0_pre],[r0],[r0a],[r1],[pre_read_ctrl],[clear_prev_by_r],[rd],[reading_00],
[reading_01],[reading_10],[reading_11]
initial=[wr,clear_valid,no_w1,no_r0,ip=0,w0=0,w1=0,not_writing_00,
not_writing_01,not_writing_10,not_writing_11,r0=0,r1=0,op=0,not_reading_00,
not_reading_01,not_reading_10,not_reading_11,enable,no_w_fresh,no_w_prev,
not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,not_r_prev_00,
not_r_prev_01,not_r_prev_10,not_r_prev_11,not_valid_00,not_valid_01,
not_valid_10,not_valid_11]
wr00b,[wr,ip=0,w0=0,not_writing_00,enable],[ip=0,w0=0,writing_00,enable]
wr00e,[writing_00,enable],[not_writing_00,f_to_p,wrote_00]
wr01b,[wr,ip=0,w0=1,not_writing_01,enable],[ip=0,w0=1,writing_01,enable]
wr01e,[writing_01,enable],[not_writing_01,f_to_p,wrote_01]
wr10b,[wr,ip=1,w1=0,not_writing_10,enable],[ip=1,w1=0,writing_10,enable]
wr10e,[writing_10,enable],[not_writing_10,f_to_p,wrote_10]
wr11b,[wr,ip=1,w1=1,not_writing_11,enable],[ip=1,w1=1,writing_11,enable]
wr11e,[writing_11,enable],[not_writing_11,f_to_p,wrote_11]
nftop,[f_to_p,no_w_fresh],[w_to_f,no_w_fresh]
```

```
ftop00,[w_fresh_00,no_w_prev,f_to_p],[no_w_fresh,w_prev_00,w_to_f]
ftop01,[w_fresh_01,no_w_prev,f_to_p],[no_w_fresh,w_prev_01,w_to_f]
ftop10,[w_fresh_10,no_w_prev,f_to_p],[no_w_fresh,w_prev_10,w_to_f]
ftop11,[w_fresh_11,no_w_prev,f_to_p],[no_w_fresh,w_prev_11,w_to_f]
wtofv00b,[wrote_00,valid_00,w_to_f,no_w_fresh],[w0,enable,w_fresh_00,valid_00]
wtofv00c,[wrote_00,not_valid_00,w_to_f,no_w_fresh],[w0,enable,w_fresh_00,
valid_00]
wtofv01b,[wrote_01,valid_01,w_to_f,no_w_fresh],[w0,enable,w_fresh_01,valid_01]
wtofv01c,[wrote_01,not_valid_01,w_to_f,no_w_fresh],[w0,enable,w_fresh_01,
valid_01]
wtofv10b,[wrote_10,valid_10,w_to_f,no_w_fresh],[w0,enable,w_fresh_10,valid_10]
wtofv10c,[wrote_10,not_valid_10,w_to_f,no_w_fresh],[w0,enable,w_fresh_10,
valid_10]
wtofv11b,[wrote_11,valid_11,w_to_f,no_w_fresh],[w0,enable,w_fresh_11,valid_11]
wtofv11c,[wrote_11,not_valid_11,w_to_f,no_w_fresh],[w0,enable,w_fresh_11,
valid_11]
w0ip0,[w0,ip=1,enable],[w1,ip=0,enable]
w0ip1,[w0,ip=0,enable],[w1,ip=1,enable]
w10001,[w1,ip=0,w0=0,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10101,[w1,ip=0,w0=1,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10010,[w1,ip=0,w0=0,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w10110,[w1,ip=0,w0=1,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w11001,[w1,ip=1,w1=0,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11101,[w1,ip=1,w1=1,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11010,[w1,ip=1,w1=0,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w11110,[w1,ip=1,w1=1,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w1_initial,[w1_post,no_w1],[clear_w_prev,w1+]
w1_normal,[w1_post,w1+],[clear_w_prev,w1+]
ncp,[no_w_prev,clear_w_prev],[wr,enable,no_w_prev]
cp00,[w_prev_00,clear_w_prev],[wr,enable,no_w_prev]
cp01,[w_prev_01,clear_w_prev],[wr,enable,no_w_prev]
cp10,[w_prev_10,clear_w_prev],[wr,enable,no_w_prev]
cp11,[w_prev_11,clear_w_prev],[wr,enable,no_w_prev]
clear_valid_00a,[not_valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_00b,[valid_00,clear_valid,enable],[not_valid_00,clear_valid2]
clear_valid_01a,[not_valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_01b,[valid_01,clear_valid2],[not_valid_01,clear_valid3]
clear_valid_10a,[not_valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_10b,[valid_10,clear_valid3],[not_valid_10,clear_valid4]
clear_valid_11a,[not_valid_11,clear_valid4],[not_valid_11,clear_r_f]
clear_valid_11b,[valid_11,clear_valid4],[not_valid_11,clear_r_f]
no_rf,[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,clear_r_f],
[not_r_fresh_00,not_r_fresh_01,not_r_fresh_10,not_r_fresh_11,wf_to_rf]
clear_rf00,[r_fresh_00,clear_r_f],[not_r_fresh_00,wf_to_rf]
clear_rf01,[r_fresh_01,clear_r_f],[not_r_fresh_01,wf_to_rf]
clear_rf10,[r_fresh_10,clear_r_f],[not_r_fresh_10,wf_to_rf]
clear_rf11,[r_fresh_11,clear_r_f],[not_r_fresh_11,wf_to_rf]
no_wf,[no_w_fresh,wf_to_rf,not_valid_00,not_valid_01,not_valid_10,
not_valid_11],[no_w_fresh,r0_pre,valid_00,valid_01,valid_10,valid_11]
wf_to_rf_00,[w_fresh_00,not_r_fresh_00,wf_to_rf],[w_fresh_00,r_fresh_00,
clear_r_p]
wf_to_rf_01,[w_fresh_01,not_r_fresh_01,wf_to_rf],[w_fresh_01,r_fresh_01,
clear_r_p]
wf_to_rf_10,[w_fresh_10,not_r_fresh_10,wf_to_rf],[w_fresh_10,r_fresh_10,
clear_r_p]
wf_to_rf_11,[w_fresh_11,not_r_fresh_11,wf_to_rf],[w_fresh_11,r_fresh_11,
clear_r_p]
no_rp,[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,clear_r_p],
[not_r_prev_00,not_r_prev_01,not_r_prev_10,not_r_prev_11,wp_to_rp]
clear_rp00,[r_prev_00,clear_r_p],[not_r_prev_00,wp_to_rp]
clear_rp01,[r_prev_01,clear_r_p],[not_r_prev_01,wp_to_rp]
clear_rp10,[r_prev_10,clear_r_p],[not_r_prev_10,wp_to_rp]
clear_rp11,[r_prev_11,clear_r_p],[not_r_prev_11,wp_to_rp]
no_wp,[no_w_prev,wp_to_rp],[no_w_prev,r0_pre]
wp_to_rp_00,[w_prev_00,not_r_prev_00,wp_to_rp],[w_prev_00,r_prev_00,r0_pre]
wp_to_rp_01,[w_prev_01,not_r_prev_01,wp_to_rp],[w_prev_01,r_prev_01,r0_pre]
```

```
wp_to_rp_10,[w_prev_10,not_r_prev_10,wp_to_rp],[w_prev_10,r_prev_10,r0_pre]
wp_to_rp_11,[w_prev_11,not_r_prev_11,wp_to_rp],[w_prev_11,r_prev_11,r0_pre]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0a1,[r0,w0=0,w1=0],[r0a,w0=0,w1=0,r0int=0,r1int=0,enable]
r0a2,[r0,w0=0,w1=1],[r0a,w0=0,w1=1,r0int=0,r1int=1,enable]
r0a3,[r0,w0=1,w1=0],[r0a,w0=1,w1=0,r0int=1,r1int=0,enable]
r0a4,[r0,w0=1,w1=1],[r0a,w0=1,w1=1,r0int=1,r1int=1,enable]
r00000b,[r0a,r0int=0,r1int=0,r0=0,r1=0,enable],[r1,r0=0,r1=0,enable]
r00001b,[r0a,r0int=0,r1int=0,r0=0,r1=1,enable],[r1,r0=0,r1=0,enable]
r00010b,[r0a,r0int=0,r1int=0,r0=1,r1=0,enable],[r1,r0=0,r1=0,enable]
r00011b,[r0a,r0int=0,r1int=0,r0=1,r1=1,enable],[r1,r0=0,r1=0,enable]
r00100b,[r0a,r0int=0,r1int=1,r0=0,r1=0,enable],[r1,r0=0,r1=1,enable]
r00101b,[r0a,r0int=0,r1int=1,r0=0,r1=1,enable],[r1,r0=0,r1=1,enable]
r00110b,[r0a,r0int=0,r1int=1,r0=1,r1=0,enable],[r1,r0=0,r1=1,enable]
r00111b,[r0a,r0int=0,r1int=1,r0=1,r1=1,enable],[r1,r0=0,r1=1,enable]
r01000b,[r0a,r0int=1,r1int=0,r0=0,r1=0,enable],[r1,r0=1,r1=0,enable]
r01001b,[r0a,r0int=1,r1int=0,r0=0,r1=1,enable],[r1,r0=1,r1=0,enable]
r01010b,[r0a,r0int=1,r1int=0,r0=1,r1=0,enable],[r1,r0=1,r1=0,enable]
r01011b,[r0a,r0int=1,r1int=0,r0=1,r1=1,enable],[r1,r0=1,r1=0,enable]
r01100b,[r0a,r0int=1,r1int=1,r0=0,r1=0,enable],[r1,r0=1,r1=1,enable]
r01101b,[r0a,r0int=1,r1int=1,r0=0,r1=1,enable],[r1,r0=1,r1=1,enable]
r01110b,[r0a,r0int=1,r1int=1,r0=1,r1=0,enable],[r1,r0=1,r1=1,enable]
r01111b,[r0a,r0int=1,r1int=1,r0=1,r1=1,enable],[r1,r0=1,r1=1,enable]
r1001,[r1,ip=1,op=0,enable],[pre_read_ctrl,ip=1,op=0,enable]
r1101,[r1,ip=1,op=1,enable],[pre_read_ctrl,ip=1,op=0,enable]
r1010,[r1,ip=0,op=0,enable],[pre_read_ctrl,ip=0,op=1,enable]
r1110,[r1,ip=0,op=1,enable],[pre_read_ctrl,ip=0,op=1,enable]
pre_read_00a,[pre_read_ctrl,enable,op=0,r0=0,no_w_fresh],[no_w_fresh,rd,
op=0,r0=0]
pre_read_00b,[pre_read_ctrl,enable,op=0,r0=0,w_fresh_00],[w_fresh_00,
clear_prev_by_r,op=0,r0=0]
pre_read_00c,[pre_read_ctrl,enable,op=0,r0=0,w_fresh_01],[w_fresh_01,rd,
op=0,r0=0]
pre_read_00d,[pre_read_ctrl,enable,op=0,r0=0,w_fresh_10],[w_fresh_10,rd,
op=0,r0=0]
pre_read_00e,[pre_read_ctrl,enable,op=0,r0=0,w_fresh_11],[w_fresh_11,rd,
op=0,r0=0]
pre_read_01a,[pre_read_ctrl,enable,op=0,r0=1,no_w_fresh],[no_w_fresh,rd,
op=0,r0=1]
pre_read_01b,[pre_read_ctrl,enable,op=0,r0=1,w_fresh_00],[w_fresh_00,rd,
op=0,r0=1]
pre_read_01c,[pre_read_ctrl,enable,op=0,r0=1,w_fresh_01],[w_fresh_01,
clear_prev_by_r,op=0,r0=1]
pre_read_01d,[pre_read_ctrl,enable,op=0,r0=1,w_fresh_10],[w_fresh_10,
rd,op=0,r0=1]
pre_read_01e,[pre_read_ctrl,enable,op=0,r0=1,w_fresh_11],[w_fresh_11,
rd,op=0,r0=1]
pre_read_10a,[pre_read_ctrl,enable,op=1,r1=0,no_w_fresh],[no_w_fresh,
rd,op=1,r1=0]
pre_read_10b,[pre_read_ctrl,enable,op=1,r1=0,w_fresh_00],[w_fresh_00,
rd,op=1,r1=0]
pre_read_10c,[pre_read_ctrl,enable,op=1,r1=0,w_fresh_01],[w_fresh_01,
rd,op=1,r1=0]
pre_read_10d,[pre_read_ctrl,enable,op=1,r1=0,w_fresh_10],[w_fresh_10,
clear_prev_by_r,op=1,r1=0]
pre_read_10e,[pre_read_ctrl,enable,op=1,r1=0,w_fresh_11],[w_fresh_11,
rd,op=1,r1=0]
pre_read_11a,[pre_read_ctrl,enable,op=1,r1=1,no_w_fresh],[no_w_fresh,
rd,op=1,r1=1]
pre_read_11b,[pre_read_ctrl,enable,op=1,r1=1,w_fresh_00],[w_fresh_00,
rd,op=1,r1=1]
pre_read_11c,[pre_read_ctrl,enable,op=1,r1=1,w_fresh_01],[w_fresh_01,
rd,op=1,r1=1]
pre_read_11d,[pre_read_ctrl,enable,op=1,r1=1,w_fresh_10],[w_fresh_10,
```

```
rd,op=1,r1=1]
pre_read_11e,[pre_read_ctrl,enable,op=1,r1=1,w_fresh_11],[w_fresh_11,
clear_prev_by_r,op=1,r1=1]
ncp_r,[no_w_prev,clear_prev_by_r],[rd,no_w_prev]
cp00_r,[w_prev_00,clear_prev_by_r],[rd,no_w_prev]
cp01_r,[w_prev_01,clear_prev_by_r],[rd,no_w_prev]
cp10_r,[w_prev_10,clear_prev_by_r],[rd,no_w_prev]
cp11_r,[w_prev_11,clear_prev_by_r],[rd,no_w_prev]
rd00b,[rd,r0=0,op=0,not_reading_00],[r0=0,op=0,reading_00,enable]
rd00e,[reading_00,enable],[not_reading_00,clear_valid,enable]
rd01b,[rd,r0=1,op=0,not_reading_01],[r0=1,op=0,reading_01,enable]
rd01e,[reading_01,enable],[not_reading_01,clear_valid,enable]
rd10b,[rd,r1=0,op=1,not_reading_10],[r1=0,op=1,reading_10,enable]
rd10e,[reading_10,enable],[not_reading_10,clear_valid,enable]
rd11b,[rd,r1=1,op=1,not_reading_11],[r1=1,op=1,reading_11,enable]
rd11e,[reading_11,enable],[not_reading_11,clear_valid,enable]
coh_lost_00,[writing_00,reading_00],[writing_00,reading_00,coh_lost]
coh_lost_01,[writing_01,reading_01],[writing_01,reading_01,coh_lost]
coh_lost_10,[writing_10,reading_10],[writing_10,reading_10,coh_lost]
coh_lost_11,[writing_11,reading_11],[writing_11,reading_11,coh_lost]
not_fresh_00,[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,
r0+],[enable,reading_00,not_valid_00,not_r_fresh_00,not_r_prev_00,not_fresh,
r0+]
not_fresh_01,[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,
r0+],[enable,reading_01,not_valid_01,not_r_fresh_01,not_r_prev_01,not_fresh,
r0+]
not_fresh_10,[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,
r0+],[enable,reading_10,not_valid_10,not_r_fresh_10,not_r_prev_10,not_fresh,
r0+]
not_fresh_11,[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,
r0+],[enable,reading_11,not_valid_11,not_r_fresh_11,not_r_prev_11,not_fresh,
r0+]
```

B.6.4 Model for testing sequence

```
4 slot (Simpsons new version) checking new-old-new, version b.
bad=[order_lost]
initial=[enable,wr,clear_x,no_w1,no_r0,ip=0,w0=0,w1=0,not_writing_00,
not_writing_01,not_writing_10,not_writing_11,r0=0,r1=0,op=0,not_reading_00,
not_reading_01,not_reading_10,not_reading_11,Xb_00,Xb_01,Xb_10,Xb_11,x_00,
x_01,x_10,x_11,4_00,4_01,4_10,4_11,not1,not2,not3]
wr00b,[enable,wr,ip=0,w0=0,not_writing_00],[enable,ip=0,w0=0,writing_00]
wr00e,[enable,writing_00],[wip_roles00,not_writing_00]
wr01b,[enable,wr,ip=0,w0=1,not_writing_01],[enable,ip=0,w0=1,writing_01]
wr01e,[enable,writing_01],[wip_roles01,not_writing_01]
wr10b,[enable,wr,ip=1,w1=0,not_writing_10],[enable,ip=1,w1=0,writing_10]
wr10e,[enable,writing_10],[wip_roles10,not_writing_10]
wr11b,[enable,wr,ip=1,w1=1,not_writing_11],[enable,ip=1,w1=1,writing_11]
wr11e,[enable,writing_11],[wip_roles11,not_writing_11]
wip1a,[wip_roles00,X_00],[wip_role200,Xb_00]
wip1b,[wip_roles00,Xb_00],[wip_role200,Xb_00]
wip1c,[wip_roles01,X_01],[wip_role201,Xb_01]
wip1d,[wip_roles01,Xb_01],[wip_role201,Xb_01]
wip1e,[wip_roles10,X_10],[wip_role210,Xb_10]
wip1f,[wip_roles10,Xb_10],[wip_role210,Xb_10]
wip1g,[wip_roles11,X_11],[wip_role211,Xb_11]
wip1h,[wip_roles11,Xb_11],[wip_role211,Xb_11]
wip2a,[wip_role200,1_00],[1_00,w0,enable]
wip2b,[wip_role201,1_01],[1_01,w0,enable]
wip2c,[wip_role210,1_10],[1_10,w0,enable]
wip2d,[wip_role211,1_11],[1_11,w0,enable]
wip2e,[wip_role200,2_00],[w2b1_00,11b2,2_00]
wip2f,[wip_role201,2_01],[w2b1_01,11b2,2_01]
wip2g,[wip_role210,2_10],[w2b1_10,11b2,2_10]
```

```
wip2h,[wip_role211,2_11],[w2b1_11,11b2,2_11]
wip2i,[wip_role200,3_00],[w3b1_00,22b3,3_00]
wip2j,[wip_role201,3_01],[w3b1_01,22b3,3_01]
wip2k,[wip_role210,3_10],[w3b1_10,22b3,3_10]
wip2l,[wip_role211,3_11],[w3b1_11,22b3,3_11]
wip2m,[wip_role200,4_00],[w4b1_00,33b4,4_00]
wip2n,[wip_role201,4_01],[w4b1_01,33b4,4_01]
wip2o,[wip_role210,4_10],[w4b1_10,33b4,4_10]
wip2p,[wip_role211,4_11],[w4b1_11,33b4,4_11]
wip3a,[33b4,3_00],[22b3,4_00]
wip3b,[33b4,3_01],[22b3,4_01]
wip3c,[33b4,3_10],[22b3,4_10]
wip3d,[33b4,3_11],[22b3,4_11]
wip3e,[33b4,not3],[22b3,not3]
wip4a,[22b3,2_00,not3],[11b2,3_00,not3b]
wip4aa,[22b3,2_00,not3b],[11b2,3_00,not3b]
wip4b,[22b3,2_01,not3],[11b2,3_01,not3b]
wip4ba,[22b3,2_01,not3b],[11b2,3_01,not3b]
wip4c,[22b3,2_10,not3],[11b2,3_10,not3b]
wip4ca,[22b3,2_10,not3b],[11b2,3_10,not3b]
wip4d,[22b3,2_11,not3],[11b2,3_11,not3b]
wip4da,[22b3,2_11,not3b],[11b2,3_11,not3b]
wip4e,[22b3,not2],[11b2,not2]
wip5a,[11b2,1_00,not2],[11b2_fin,2_00,not2b]
wip5aa,[11b2,1_00,not2b],[11b2_fin,2_00,not2b]
wip5b,[11b2,1_01,not2],[11b2_fin,2_01,not2b]
wip5ba,[11b2,1_01,not2b],[11b2_fin,2_01,not2b]
wip5c,[11b2,1_10,not2],[11b2_fin,2_10,not2b]
wip5ca,[11b2,1_10,not2b],[11b2_fin,2_10,not2b]
wip5d,[11b2,1_11,not2],[11b2_fin,2_11,not2b]
wip5da,[11b2,1_11,not2b],[11b2_fin,2_11,not2b]
wip5e,[11b2,not1],[11b2_fin,not1]
wip6a,[11b2_fin,w2b1_00,2_00],[1_00,w0,enable]
wip6b,[11b2_fin,w2b1_01,2_01],[1_01,w0,enable]
wip6c,[11b2_fin,w2b1_10,2_10],[1_10,w0,enable]
wip6d,[11b2_fin,w2b1_11,2_11],[1_11,w0,enable]
wip6e,[11b2_fin,w3b1_00,3_00],[1_00,w0,enable]
wip6f,[11b2_fin,w3b1_01,3_01],[1_01,w0,enable]
wip6g,[11b2_fin,w3b1_10,3_10],[1_10,w0,enable]
wip6h,[11b2_fin,w3b1_11,3_11],[1_11,w0,enable]
wip6i,[11b2_fin,w4b1_00,4_00,not1],[1_00,w0,enable,not1b]
wip6ia,[11b2_fin,w4b1_00,4_00,not1b],[1_00,w0,enable,not1b]
wip6j,[11b2_fin,w4b1_01,4_01,not1],[1_01,w0,enable,not1b]
wip6ja,[11b2_fin,w4b1_01,4_01,not1b],[1_01,w0,enable,not1b]
wip6k,[11b2_fin,w4b1_10,4_10,not1],[1_10,w0,enable,not1b]
wip6ka,[11b2_fin,w4b1_10,4_10,not1b],[1_10,w0,enable,not1b]
wip6l,[11b2_fin,w4b1_11,4_11,not1],[1_11,w0,enable,not1b]
wip6la,[11b2_fin,w4b1_11,4_11,not1b],[1_11,w0,enable,not1b]
w0ip0,[enable,w0,ip=1],[enable,w1,ip=0]
w0ip1,[enable,w0,ip=0],[enable,w1,ip=1]
w10001,[w1,ip=0,w0=0,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10101,[w1,ip=0,w0=1,r0=1,enable],[ip=0,w0=0,r0=1,w1_post]
w10010,[w1,ip=0,w0=0,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w10110,[w1,ip=0,w0=1,r0=0,enable],[ip=0,w0=1,r0=0,w1_post]
w11001,[w1,ip=1,w1=0,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11101,[w1,ip=1,w1=1,r1=1,enable],[ip=1,w1=0,r1=1,w1_post]
w11010,[w1,ip=1,w1=0,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w11110,[w1,ip=1,w1=1,r1=0,enable],[ip=1,w1=1,r1=0,w1_post]
w1_initial,[w1_post,no_w1],[wr,w1+,enable]
w1_normal,[w1_post,w1+],[wr,w1+,enable]
clr_x00,[enable,clear_x,x_00],[clear_x2,xb_00]
clr_x00a,[enable,clear_x,xb_00],[clear_x2,xb_00]
clr_x01,[clear_x2,x_01],[clear_x3,xb_01]
clr_x01a,[clear_x2,xb_01],[clear_x3,xb_01]
clr_x02,[clear_x3,x_10],[clear_x4,xb_10]
clr_x02a,[clear_x3,xb_10],[clear_x4,xb_10]
```

```
clr_x03,[clear_x4,x_11],[r0_pre,xb_11]
clr_x03a,[clear_x4,xb_11],[r0_pre,xb_11]
r0_init0,[r0_pre,no_w1],[r0,no_w1]
r0_init1,[r0_pre,w1+,no_r0],[r0,w1+,r0+]
r0_normal,[r0_pre,r0+],[r0,r0+]
r0a1,[r0,w0=0,w1=0],[enable,r0a,w0=0,w1=0,r0int=0,r1int=0]
r0a2,[r0,w0=0,w1=1],[enable,r0a,w0=0,w1=1,r0int=0,r1int=1]
r0a3,[r0,w0=1,w1=0],[enable,r0a,w0=1,w1=0,r0int=1,r1int=0]
r0a4,[r0,w0=1,w1=1],[enable,r0a,w0=1,w1=1,r0int=1,r1int=1]
r00000b,[enable,r0a,r0int=0,r1int=0,r0=0,r1=0],[enable,r1,r0=0,r1=0]
r00001b,[enable,r0a,r0int=0,r1int=0,r0=0,r1=1],[enable,r1,r0=0,r1=0]
r00010b,[enable,r0a,r0int=0,r1int=0,r0=1,r1=0],[enable,r1,r0=0,r1=0]
r00011b,[enable,r0a,r0int=0,r1int=0,r0=1,r1=1],[enable,r1,r0=0,r1=0]
r00100b,[enable,r0a,r0int=0,r1int=1,r0=0,r1=0],[enable,r1,r0=0,r1=1]
r00101b,[enable,r0a,r0int=0,r1int=1,r0=0,r1=1],[enable,r1,r0=0,r1=1]
r00110b,[enable,r0a,r0int=0,r1int=1,r0=1,r1=0],[enable,r1,r0=0,r1=1]
r00111b,[enable,r0a,r0int=0,r1int=1,r0=1,r1=1],[enable,r1,r0=0,r1=1]
r01000b,[enable,r0a,r0int=1,r1int=0,r0=0,r1=0],[enable,r1,r0=1,r1=0]
r01001b,[enable,r0a,r0int=1,r1int=0,r0=0,r1=1],[enable,r1,r0=1,r1=0]
r01010b,[enable,r0a,r0int=1,r1int=0,r0=1,r1=0],[enable,r1,r0=1,r1=0]
r01011b,[enable,r0a,r0int=1,r1int=0,r0=1,r1=1],[enable,r1,r0=1,r1=0]
r01100b,[enable,r0a,r0int=1,r1int=1,r0=0,r1=0],[enable,r1,r0=1,r1=1]
r01101b,[enable,r0a,r0int=1,r1int=1,r0=0,r1=1],[enable,r1,r0=1,r1=1]
r01110b,[enable,r0a,r0int=1,r1int=1,r0=1,r1=0],[enable,r1,r0=1,r1=1]
r01111b,[enable,r0a,r0int=1,r1int=1,r0=1,r1=1],[enable,r1,r0=1,r1=1]
r1001,[enable,r1,ip=1,op=0],[enable,rcp_roles,ip=1,op=0]
r1101,[enable,r1,ip=1,op=1],[enable,rcp_roles,ip=1,op=0]
r1010,[enable,r1,ip=0,op=0],[enable,rcp_roles,ip=0,op=1]
r1110,[enable,r1,ip=0,op=1],[enable,rcp_roles,ip=0,op=1]
x_r00a,[enable,rcp_roles,r0=0,op=0,X_00,xb_00],[r0=0,op=0,X_00,x_00,
rcp_roles2_00]
x_r00b,[enable,rcp_roles,r0=0,op=0,Xb_00],[r0=0,op=0,Xb_00,rcp_roles2_00]
x_r01a,[enable,rcp_roles,r0=1,op=0,X_01,xb_01],[r0=1,op=0,X_01,x_01,
rcp_roles2_01]
x_r01b,[enable,rcp_roles,r0=1,op=0,Xb_01],[r0=1,op=0,Xb_01,rcp_roles2_01]
x_r10a,[enable,rcp_roles,r1=0,op=1,X_10,xb_10],[r1=0,op=1,X_10,x_10,
rcp_roles2_10]
x_r10b,[enable,rcp_roles,r1=0,op=1,Xb_10],[r1=0,op=1,Xb_10,rcp_roles2_10]
x_r11a,[enable,rcp_roles,r1=1,op=1,X_11,xb_11],[r1=1,op=1,X_11,x_11,
rcp_roles2_11]
x_r11b,[enable,rcp_roles,r1=1,op=1,Xb_11],[r1=1,op=1,Xb_11,rcp_roles2_11]
X_r00a,[rcp_roles2_00,1_00],[1_00,rcp_Xb_00,rcp_X_01,rcp_X_10,rcp_X_11,
rcp_role3]
X_r00b,[rcp_roles2_00,2_00],[2_00,rcp_Xb_00,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r00c,[rcp_roles2_00,3_00],[3_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r00d,[rcp_roles2_00,4_00],[4_00,rcp_Xb_00,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r01a,[rcp_roles2_01,1_01],[1_01,rcp_Xb_01,rcp_X_00,rcp_X_10,rcp_X_11,
rcp_role3]
X_r01b,[rcp_roles2_01,2_01],[2_01,rcp_Xb_01,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r01c,[rcp_roles2_01,3_01],[3_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r01d,[rcp_roles2_01,4_01],[4_01,rcp_Xb_01,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r10a,[rcp_roles2_10,1_10],[1_10,rcp_Xb_10,rcp_X_01,rcp_X_00,rcp_X_11,
rcp_role3]
X_r10b,[rcp_roles2_10,2_10],[2_10,rcp_Xb_10,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r10c,[rcp_roles2_10,3_10],[3_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r10d,[rcp_roles2_10,4_10],[4_10,rcp_Xb_10,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
X_r11a,[rcp_roles2_11,1_11],[1_11,rcp_Xb_11,rcp_X_01,rcp_X_10,rcp_X_00,
```

```
rcp_role3]
X_r11b,[rcp_roles2_11,2_11],[2_11,rcp_Xb_11,rcp_Xb_1,rcp_X_3,rcp_X_4,
rcp_role3]
X_r11c,[rcp_roles2_11,3_11],[3_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_X_4,
rcp_role3]
X_r11d,[rcp_roles2_11,4_11],[4_11,rcp_Xb_11,rcp_Xb_1,rcp_Xb_2,rcp_Xb_3,
rcp_role3]
rcp_3a,[rcp_role3,rcp_Xb_00,X_00],[Xb_00,rcp_role4]
rcp_3b,[rcp_role3,rcp_Xb_00,Xb_00],[Xb_00,rcp_role4]
rcp_3c,[rcp_role3,rcp_Xb_01,X_01],[Xb_01,rcp_role4]
rcp_3d,[rcp_role3,rcp_Xb_01,Xb_01],[Xb_01,rcp_role4]
rcp_3e,[rcp_role3,rcp_Xb_10,X_10],[Xb_10,rcp_role4]
rcp_3f,[rcp_role3,rcp_Xb_10,Xb_10],[Xb_10,rcp_role4]
rcp_3g,[rcp_role3,rcp_Xb_11,X_11],[Xb_11,rcp_role4]
rcp_3h,[rcp_role3,rcp_Xb_11,Xb_11],[Xb_11,rcp_role4]
rcp_4a,[rcp_role4,rcp_X_01,X_01],[X_01,rcp_role5]
rcp_4b,[rcp_role4,rcp_X_01,Xb_01],[X_01,rcp_role5]
rcp_4c,[rcp_role4,rcp_X_00,Xb_00],[X_00,rcp_role5]
rcp_4d,[rcp_role4,rcp_X_00,X_00],[X_00,rcp_role5]
rcp_4e,[rcp_role4,rcp_Xb_1,1_00,X_00],[1_00,Xb_00,rcp_role5]
rcp_4f,[rcp_role4,rcp_Xb_1,1_00,Xb_00],[1_00,Xb_00,rcp_role5]
rcp_4g,[rcp_role4,rcp_Xb_1,1_01,X_01],[1_01,Xb_01,rcp_role5]
rcp_4h,[rcp_role4,rcp_Xb_1,1_01,Xb_01],[1_01,Xb_01,rcp_role5]
rcp_4i,[rcp_role4,rcp_Xb_1,1_10,X_10],[1_10,Xb_10,rcp_role5]
rcp_4j,[rcp_role4,rcp_Xb_1,1_10,Xb_10],[1_10,Xb_10,rcp_role5]
rcp_4k,[rcp_role4,rcp_Xb_1,1_11,X_11],[1_11,Xb_11,rcp_role5]
rcp_4l,[rcp_role4,rcp_Xb_1,1_11,Xb_11],[1_11,Xb_11,rcp_role5]
rcp_5a,[rcp_role5,rcp_X_10,X_10],[X_10,rcp_role6]
rcp_5b,[rcp_role5,rcp_X_10,Xb_10],[X_10,rcp_role6]
rcp_5c,[rcp_role5,rcp_X_00,X_00],[X_00,rcp_role6]
rcp_5d,[rcp_role5,rcp_X_00,Xb_00],[X_00,rcp_role6]
rcp_5e,[rcp_role5,rcp_Xb_2,2_00,X_00],[2_00,Xb_00,rcp_role6]
rcp_5f,[rcp_role5,rcp_Xb_2,2_00,Xb_00],[2_00,Xb_00,rcp_role6]
rcp_5g,[rcp_role5,rcp_Xb_2,2_01,X_01],[2_01,Xb_01,rcp_role6]
rcp_5h,[rcp_role5,rcp_Xb_2,2_01,Xb_01],[2_01,Xb_01,rcp_role6]
rcp_5i,[rcp_role5,rcp_Xb_2,2_10,X_10],[2_10,Xb_10,rcp_role6]
rcp_5j,[rcp_role5,rcp_Xb_2,2_10,Xb_10],[2_10,Xb_10,rcp_role6]
rcp_5k,[rcp_role5,rcp_Xb_2,2_11,X_11],[2_11,Xb_11,rcp_role6]
rcp_5l,[rcp_role5,rcp_Xb_2,2_11,Xb_11],[2_11,Xb_11,rcp_role6]
rcp_5m,[rcp_role5,rcp_X_3,3_00,X_00],[3_00,X_00,rcp_role6]
rcp_5n,[rcp_role5,rcp_X_3,3_00,Xb_00],[3_00,X_00,rcp_role6]
rcp_5o,[rcp_role5,rcp_X_3,3_01,X_01],[3_01,X_01,rcp_role6]
rcp_5p,[rcp_role5,rcp_X_3,3_01,Xb_01],[3_01,X_01,rcp_role6]
rcp_5q,[rcp_role5,rcp_X_3,3_10,X_10],[3_10,X_10,rcp_role6]
rcp_5r,[rcp_role5,rcp_X_3,3_10,Xb_10],[3_10,X_10,rcp_role6]
rcp_5s,[rcp_role5,rcp_X_3,3_11,X_11],[3_11,X_11,rcp_role6]
rcp_5t,[rcp_role5,rcp_X_3,3_11,Xb_11],[3_11,X_11,rcp_role6]
rcp_6a,[rcp_role6,rcp_X_11,X_11],[X_11,rd]
rcp_6b,[rcp_role6,rcp_X_11,Xb_11],[X_11,rd]
rcp_6c,[rcp_role6,rcp_X_00,X_00],[X_00,rd]
rcp_6d,[rcp_role6,rcp_X_00,Xb_00],[X_00,rd]
rcp_6e,[rcp_role6,rcp_Xb_3,3_00,X_00],[3_00,Xb_00,rd]
rcp_6f,[rcp_role6,rcp_Xb_3,3_00,Xb_00],[3_00,Xb_00,rd]
rcp_6g,[rcp_role6,rcp_Xb_3,3_01,X_01],[3_01,Xb_01,rd]
rcp_6h,[rcp_role6,rcp_Xb_3,3_01,Xb_01],[3_01,Xb_01,rd]
rcp_6i,[rcp_role6,rcp_Xb_3,3_10,X_10],[3_10,Xb_10,rd]
rcp_6j,[rcp_role6,rcp_Xb_3,3_10,Xb_10],[3_10,Xb_10,rd]
rcp_6k,[rcp_role6,rcp_Xb_3,3_11,X_11],[3_11,Xb_11,rd]
rcp_6l,[rcp_role6,rcp_Xb_3,3_11,Xb_11],[3_11,Xb_11,rd]
rcp_6m,[rcp_role6,rcp_X_4,4_00,X_00],[4_00,X_00,rd]
rcp_6n,[rcp_role6,rcp_X_4,4_00,Xb_00],[4_00,X_00,rd]
rcp_6o,[rcp_role6,rcp_X_4,4_01,X_01],[4_01,X_01,rd]
rcp_6p,[rcp_role6,rcp_X_4,4_01,Xb_01],[4_01,X_01,rd]
rcp_6q,[rcp_role6,rcp_X_4,4_10,X_10],[4_10,X_10,rd]
rcp_6r,[rcp_role6,rcp_X_4,4_10,Xb_10],[4_10,X_10,rd]
rcp_6s,[rcp_role6,rcp_X_4,4_11,X_11],[4_11,X_11,rd]
```

```
rcp_6t,[rcp_role6,rcp_X_4,4_11,Xb_11],[4_11,X_11,rd]
rd00b,[rd,r0=0,op=0,not_reading_00],[enable,r0=0,op=0,reading_00]
rd00e,[enable,reading_00],[enable,clear_x,not_reading_00]
rd01b,[rd,r0=1,op=0,not_reading_01],[enable,r0=1,op=0,reading_01]
rd01e,[enable,reading_01],[enable,clear_x,not_reading_01]
rd10b,[rd,r1=0,op=1,not_reading_10],[enable,r1=0,op=1,reading_10]
rd10e,[enable,reading_10],[enable,clear_x,not_reading_10]
rd11b,[rd,r1=1,op=1,not_reading_11],[enable,r1=1,op=1,reading_11]
rd11e,[enable,reading_11],[enable,clear_x,not_reading_11]
order_lost_00,[enable,reading_00,x_00,r0+],[enable,reading_00,x_00,
order_lost,r0+]
order_lost_01,[enable,reading_01,x_01,r0+],[enable,reading_01,x_01,
order_lost,r0+]
order_lost_10,[enable,reading_10,x_10,r0+],[enable,reading_10,x_10,
order_lost,r0+]
order_lost_11,[enable,reading_11,x_11,r0+],[enable,reading_11,x_11,
order_lost,r0+]
```

Bibliography

[AF73] T. Agerwala and M. Flynn. Comments on capabilities, limitations and 'correctness' of Petri nets. Computer Science Program Research Report 26, John Hopkins University, Baltimore, Maryland, July 1973. 58 pages.

[AG92] J.H. Anderson and M.G. Gouda. A criterion for atomicity. *Formal Aspects of Computing*, (4):273–298, 1992.

[AKKV87] B. Awerbuch, L.M. Kirousis, E. Kranakis, and P.M.B. Vitanyi. On proving register atomicity. Technical Report CS-R8707, CWI - Centrum voor Wiskunde en Informatica, Amsterdam, February 1987.

[Anc82] F. Anceau. A synchronous apporach for clocking VLSI systems. *IEEE Journal on Solid State Circuits*, SC-17:51–56, February 1982.

[Arn97] C. Arnault. A finite state machine toolkit: FSM. LAL - Orsay, France, 1997.

[Bac92] J. Bacon. *Concurrent Systems: an integrated approach to operating systems, database and distributed systems.* Addison Wesley, 1992.

[Bel92] D. Bell. *Distributed Database Systems.* International Computer Science Series. Addison Wesley, 1992. ISBN 0-201-54400-8.

[BG96] V. A. Bartlett and E. Grass. Comparison of completion detction techniques for asynchronous circuits. In Furber S. Arvind D.K., editor, *Proceedings of the 1st UK Asynchronous Forum*, Department of Computer Science, University of Edinburgh, December 1996.

[Bib87] Bibliography of Petri nets. LNCS Vol.266, 1987. pp 309–451.

[Blo86] B. Bloom. Constructing two writer atomic registers. MIT Press, June 1986.

[Blo88] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, December 1988.

[BP96] N. Busi and G.M. Pinna. Non-sequential semantics for contextual P/T nets. In *Proceedings of 17th Int. Conf. on Applications and Theory of Petri Nets*, pages 113–132, June 1996. Osaka.

[Cam91] E. Campbell. DIADEM hardware description. British Aerospace Defence Dynamics Ltd., 1991.

[Cam92] E. Campbell. DIA temporal characteristics and their experimental verification. British Aerospace Defence Ltd. Univ. of York, Admiral Management Services, 1992.

[CDJ84] F.B. Chambers, D.A. Duce, and G.P. Jones, editors. *Distributed Computing*. APIC studies in Data Processing No. 20. Academic Press, 1984. ISBN 0-12-167350-2.

[Cha83] T.J. Chaney. Measured flip-flop responses to marginal triggering. *IEEE Transactions on Computers*, C-32(12):1207–1209, December 1983.

[Cha84] D.M. Chapiro. *Globally asynchronous locally synchronous systems*. PhD thesis, Stanford University, October 1984. STAN-CS-84-1026.

[Cha99] The Chambers Dictionary. Chambers, 1999.

[CHD93] I.G. Clark, R.E. Hiorns, and A.C. Davies. VHDL implementations of integer noise shapers with floating point accuracy. In *94th AES Convention, Berlin*, pages Pre–print 3568(D4–7), March 1993.

[CKK+96] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[Claa] I.G. Clark. Atomic and asynchronous communication bibliography. Online at http://www.eee.kcl.ac.uk/˜ ianc/.

[Clab] I.G. Clark. Metastability bibliography. Online at http://www.eee.kcl.ac.uk/˜ ianc/.

[Cla95] I.G. Clark. Reach - a reachability search program for cyclic Petri nets. Petri net email list URL:http://www.daimi.au.dk/PetriNets/, 1995.

[Cla99] I.G. Clark. Memory optimisation for multi-process state spaces. Technical Report CS-TR-689, University of Newcastle upon Tyne, Computing Science, Newcastle, December 1999.

[CM73] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[COL72] T.J. Chaney, S.M. Ornstein, and W.M. Littlefield. Beware the synchronizer. In *IEEE Compcon.*, September 1972.

[Cor] J. Cortadella. Petrify: a tool for synthesis of Petri nets and asynchronous circuits. Online at http://www.lsi.upc.es/j̃ordic/petrify/petrify.html.

[CVH91] J. Calvo, M. Valencia, and J.L. Huertas. Metastable operation in RS flip flops. *International Journal of Electronics*, 70(6):1073–1091, 1991.

[CW75] G.R. Couranz and D.F. Wann. Theoretical and experimental behavior of synchronizers operating in metastable region. *IEEE Transactions on Computers*, C-24(6):604–616, June 1975.

[CXYD98] I.G. Clark, F. Xia, A.V. Yakovlev, and A.C. Davies. Petri net models of latch metastability. *IEE Electronics Letters*, (7):635–636, April 1998. IEE online number 19980502.

[CZTS99] W. Cai, K. Zhang, S.J. Turner, and C. Sun. Interlock avoidance in transparent and dynamic parallel program instrumentation using logical clocks. *Parallel Computing*, 25(5):569–591, May 1999.

[DA95] A.A. Desrochers and R.Y. Aljaar. *Application of Petri nets in manufacturing systems: modelling, control and performance analysis.* IEEE, New York, 1995. ISBN 0-87942-295-5.

[Dav89] A.C. Davies. Analysis of the metastable dynamics of bistable flip-flops. In *6th Int. Symp. on Networks, Systems and Signal Processing*, pages 379–382, June 1989.

[Dav90] A.C. Davies. Modelling and simulation of the metastability of flip-flops. In *4th Tagung Schaltkreisentwarf*, February 1990.

[Dav99] A.C. Davies. Metastability in latches, arbiters and data-convertors. In *IEEE International symposium on Signals, Circuits and Systems SCS'99*, pages 1–4. IEEE, July 6-7 1999. Invited paper.

[DDH94] M.E. Dean, D.L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (cscd). *Journal of VLSI Signal Processing*, 7(1):7–16, 1994.

[DES99] Discrete Event Systems Group, UMDES software library. University of Michigan, 1999.

[Dij68] E. Dijkstra. Cooperating Sequential Processes. In Genuys F., editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[EN94] J. Esparza and M. Neilson. Decidability issues for Petri nets. Petri net newsletter No. 47, 1994. pp 5–23.

[ERV95] J. Esparza, S. Romer, and W. Vogler. An improvement of McMillan's unfolding algorithm. Technical Report TUM-I9599, Technical University of Munchen, August 1995.

[FD79] W. Fleischhammer and O. Dortok. The anomalous behavior of flip-flops in synchronizer circuits. *IEEE Transactions on Computers*, C-28(3):273–276, March 1979.

[Fra00] D. Fraser. Private communication, April 2000.

[Fur] S. Furber. Comments on asynchronous operation from the authors personal website.

[GbATR] GraphViz by AT & T Research. on the internet at http://www.research.att.com/sw/tools/.

[HC87] J. Horstmann and R. Coates. Metastable behaviour of LSI logic flip-flops, 1.5u technology. Technical report, LSI Logic Corp., Palo Alto, CA, September 1987.

[HEC89] J.U. Horstmann, H.W. Eichel, and R.L. Coates. Metastability behavior of CMOS ASIC flip-flops in theory and test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, February 1989.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol91] G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1991. ISBN 0-13-539834-7.

[HV95] S. Haldar and K. Vidyasankar. Constructing 1-writer multi-reader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, 1995.

[HV96] S. Halder and K. Vidyasankar. Space-efficient construction of buffer-optimal 1-writer 1-reader multivalued atomic variable. *Journal of the ACM*, 1996.

[HVwn] S. Haldar and K. Vidyasankar. A space-optimum 1-buffer read/write construction of 1-writer 1-reader multivalued atomic variable. Technical report, Year unknown.

[Int94] White paper on Pentium FDIV flaw. Online at http://support.intel.com/support/processors/pentium/fdiv/wp/, November 1994.

[IS92] A. Israeli and A. Shaham. Optimal multi-writer multi-reader atomic register. In *Proc. Annual (ACM) symposium on Principles of Distributed Computing*, number 11th annual, pages 71–82, August 1992.

[IST90] O.A. Izosimov, I.I. Shagurin, and V.V. Tsylyov. Physical approach to CMOS self-timing. *IEE Electronics Letters*, 26:1835–1836, 1990.

[JA85] T.A. Jackson and A. Albicki. Modelling and simulation of NMOS flip-flops in metastable region. In *Proc. 12th Int. Conf. Applied Simulation and Modelling*, July 1985.

[JA89] T.A. Jackson and A. Albicki. Analysis of metastable operation in D latches. *IEEE Transactions on Circuits and Systems*, 36(11):1392–1404, November 1989.

[Jac85] T.A. Jackson. *Metastable operation of nMOS flip-flops*. PhD thesis, Elec. Eng. Dept. Univ of Rochester, NY, 1985.

[Jif98] M.A. Jiffry. Petri net analysis of the MASCOT pool IDA commuication mechanisms. In *Proceedings Design CPN Workshop*, 1998.

[KC86] L. Kleeman and A. Cantoni. Can redundancy and masking improve the performance of synchronizers. *IEEE Transactions on Computers*, 35:643–646, July 1986.

[KD90] L.S. Kim and R.W. Dutton. Metastability of CMOS latch/flip-flop. *IEEE Journal of Solid-State Circuits*, 25(4):942–951, August 1990.

[Kin00] D. Kinniment. Measurements on a high speed arbiter. Technical Report TR677, Dept. of Computing Science, Newcastle University, Feb. 2000.

[KK90] H. Kopetz and K. Kim. Real-time temporal uncertainties in interactions among real-time objects. In *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, pages 165–174, Huntsville, 1990.

[KKV87a] L.M. Kirousis, E. Kranakis, and P.M.B. Vitanyi. Atomic multireader register. Technical Report CS-8722, CWI - Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

[KKV87b] L.M. Kirousis, E. Kranakis, and P.M.B. Vitanyi. Atomic multireader register (detailed abstract). Technical Report CS-8704, CWI - Centrum voor Wiskunde en Informatica, Amsterdam, January 1987.

[KKV88] L.M. Kirousis, E. Kranakis, and P.M.B. Vitanyi. Atomic multireader register. In *Proc. International Workshop on Distributed Computing*, number 312 in Lecture Notes in Computer Science, pages 278–296, July 1988.

[Kos73] S.R. Kosaraju. Limitations of Dijkstra's semaphore primitives and Petri nets. Technical Report report 25, John Hopkins University, Baltimore, Maryland, May 1973. 20 pages.

[KP84] A. Kelley and I. Pohl. *A Book on C*. Benjamin Cummings, 1984.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988. ISBN 0-13-110362-8.

[KW76] D.J. Kinniment and J.V. Woods. Synchronisation and arbitration circuits in digital systems. *IEE Proceedings*, 123(10):961–966, October 1976.

[KYG97] D. Kinniment, A.V. Yakovlev, and B. Gao. Metastable behavior and system performance. In *Proc. 2nd UK Asynchronous Forum, Newcastle Uni.*, July 1997.

[Lam86] L. Lamport. On interprocess communication parts I (Basic Formalism) and II (Algorithms). *Distributed Computing*, 1:77–101, 1986.

[LC83] Y.W. Lim and J.R. Cox. Clocks and the performance of synchronisers. *IEE Proceedings, Part E, Computers and Digital Techniques*, 130(2):57–64, March 1983.

[LS89] L. Lamport and F.B. Schnieder. Pretending atomicity. Technical Report 44, SRC Digital Equipment Corp., 1989.

[LSU89] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer, 1989.

[LTV96] M. Li, J. Tromp, and P.M.B. Vitanyi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, July 1996.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Data Management Systems. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.

[Mar77] L.R. Marino. The effect of asynchronous inputs on sequential network reliability. *IEEE Transactions on Computers*, 26(11):1082–1090, November 1977.

[Mar81] L.R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[MAS87] *The Official Handbook of MASCOT*. Computing Division, N Building, Royal Signals and Radar Establishment, St Andrews Road, Malvern, Worcestershire, WR14 3PS, UK. (Now available from the Defence Research Information Centre, DERA, Kentigern House, 65 Bron Street, Glasgow, G2 8EX.), 3.1 edition, June 1987.

[McM92] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[MG96] I. Mitchell and M. Greenstreet. Proving newtonian arbiters correct, almost surely. Master's thesis, University of British Colombia, 1996.

[Mil80] R. Milner. *A calculus of communicating systems*. Springer-Verlag, Berlin, 1980. ISBN 3-540-10235-3.

[Nap93] R. Napthine. DORIS - an integrated approach to real time systems design. In *In Proc. IEE Int. Workshop on Systems Eng. for Real Tiem Applications*, pages 55–60, Royal Agricultural College, Cirencester, Spetember 1993. IEE.

[OH76] M. O'Neill and J.N. Hanlon. A data storage arrangement for buffering asynchronous input and output data streams. European patent, January 1976. Patent No. 1-487-706.

[Pat79] Asynchronous buffer. European patent, October 1979. Patent No. 2-039-102.

[PB86] G.L. Peterson and J.E. Burns. Concurrent reading while writing II: The multi-writer case (preliminary report). Technical Report GIT-ICS-86/26, December 1986.

[Pea99] D. Powell and et al. GUARDS: A generic upgradable architecture for real-time dependable systems. In *IEEE Trans. on Parallel and Distributed Systems*, volume 10, pages 580–599, June 1999.

[Pet] Petri net email list, online at http://www.daimi.au.dk/PetriNets/, and email PetriNets@daimi.au.dk.

[Pet81a] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[Pet81b] J.L. Peterson. *Petri Net theory and the modelling of systems.* Prentice Hall, 1981.

[Pet83] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Langauges and Systems*, 5(1):46–55, 1983.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, 1975.

[Rei85] W. Reisig. *Petri Nets: an Introduction.* EATCS Monographs on theoretical computer science. Springer Verlag, 1985.

[SAG86] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. Technical Report 86.30, December 1986.

[SAG87] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proc. Annual (ACM) symposium on Principles of Distributed Computing*, number 6th annual, pages 206–221, August 1987.

[SAG89] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register (updated version). Technical Report TR-89-14, University of Texas at Austin, May 1989.

[SAG94] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, March 1994.

[Sei71] C.L. Seitz. *Graph representation of logical machines.* PhD thesis, MIT, 1971.

[Sei79] C.L. Seitz. Self-times VLSI systems. In Seitz C.L., editor, *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, pages 345–355, C.S. Dept., Pasadena., January 1979.

[Sei80] C.L. Seitz. Ideas about arbiters. *Lambda, 1 (1, first quarter)*, pages 10–14, 1980.

[She92] D. Shear. Exorcise metastability from your design. *EDN*, 37:58–64, December 10 1992.

[Sim86] H.R. Simpson. The MASCOT method. *Software Engineering Journal*, 1(3):103–120, 1986.

[Sim90a] H.R. Simpson. A data interchange architecture (DIA) for real time embedded multi processor systems. In *Proc. RAe Conf. on Computing Techniques in Guided Flight*, April 1990.

[Sim90b] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E, Computers and Digital Techniques*, 137(1):17–30, January 1990.

[Sim92a] H.R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings, Part E, Computers and Digital Techniques*, 139(1):35–49, January 1992.

[Sim92b] H.R. Simpson. Real time networks in configurable distributed systems. In *Int. Workshop on Configurable Distributed Systems*, Imperial College, London., March 1992.

[Sim94] H.R. Simpson. Architecture for computer based systems. In *Proc. of tutorial and workshop on systems engineering of computer based systems*, pages 70–82, May 1994. Stockholm, Sweden.

[Sim97a] H.R. Simpson. Multireader and multiwriter asynchronous communication mechanisms. *IEE Proceedings, Part E, Computers and Digital Techniques*, 144(4):241–243, July 1997.

[Sim97b] H.R. Simpson. New algorithms for asynchronous communication. *IEE Proceedings, Part E, Computers and Digital Techniques*, 144(4):227–231, July 1997.

[Sim97c] H.R. Simpson. Role model analysis of an asynchronous communication mechanism. *IEE Proceedings, Part E, Computers and Digital Techniques*, 144(4):232–240, July 1997.

[Sta00] W. Stallings. *Data and Computer Communications*. Prentice Hall, 6th edition, 2000. ISBN 0-13-084370-9.

[SY97] A. Semenov and A. Yakovlev. Contextual net unfolding and asynchronous system verification. Technical Report TR-572, Computing Science, University of Newcastle upon Tyne, April 1997.

[Tan90] A.S. Tanenbaum. *Structured computer organization*. Prentice Hall, 1990. pp47–48.

[Tho91] C.M. Thomas. Definition of the DORIS design language. British Aerospace Defence Dynamics Ltd., 1991.

[Tro89] J. Tromp. How to construct an atomic variable (extended abstract). In *Proc. International workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 292–302, 1989.

[TS98] F. Toto and R. Saletti. CMOS dynamic ternary circuit with full logic swing and zero-static power consumption. *Electronics Letters*, 34(11):1083–1084, May 1998.

[Tur36] A. Turing. On computable numbers, with an application to the Entscheiduags Problem. In *Proc. London Math Society*, volume 42 of *2*, pages 230–265, 1936. Correction in Vol. 43 pp 544–546.

[Unk98] Unknown. Optical fibres for on chip clock distribution. *Electronic Design*, page 27, 22nd October 1998.

[VA86] P.M.B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. Technical Report MIT/LCS/TM 314, MIT Press, 1986.

[VHHP95] K. Varpaaniemi, K. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Digital Systems Lab. Espoo, Finland, August 1995.

[Wak87] J.F. Wakerly. Designers guide to synchronisers and metastability - parts I & II. *Stanford University*, 1(Part I No 1, part II No 2), Part I September, part II October 1987. Microprocessor Report.

[Wel96] C. Wellheuser. Metastability performance of clocked FIFOs. Technical Report SCZA004A, Texas Instruments, 1996.

[XC95] F. Xia and I.G. Clark. Petri net models of a class of asynchronous communication mechanisms. Technical report, Department of Electronic and Electrical Engineering, King's College London, 1995.

[XC98a] F. Xia and I. Clark. Complementing role models with Petri nets in studying asynchronous data communications. In *International Conference on Applications and Theory of Petri nets*, pages 66–85, 1998.

[XC98b] F. Xia and I.G. Clark. Complementing the role model method with Petri net techniques in studying issues of data freshness of the four-slot mechanism. Technical Report CS-TR-654, University of Newcastle upon Tyne, Computing Science, Newcastle, October 1998.

[XC99] F. Xia and I.G. Clark. Studying the three-slot asynchronous communication mechanism. In *Proceedings of the 7th UK Asynchronous Forum*, Newcastle upon Tyne, UK, December 1999.

[XC00] F. Xia and I. Clark. *Hardware Design and Petri Nets (Eds. A. Yakovlev, L. Gomes, L. Lavagno) Chapter 1: Complementing role models with Petri nets in studying asynchronous data communications*, pages 33–50. Kluwer Academic Publishers, March 2000. ISBN 0-7923-7791-5.

[XCD97] F. Xia, I.G. Clark, and A.C. Davies. Petri-net based investigation of synchronisation-free interprocess communication in shared-memory real-time systems. In Yakovlev A.V. Furber S.B., editor, *Second UK Asynchronous Forum*, July 1997.

[XCYD97] F. Xia, I.G. Clark, A.V. Yakovlev, and A.C. Davies. Petri net models of metastable operations in latch circuits. In Furber S. Arvind D.K., editor, *Third UK Forum on Asynchronous Systems*, December 1997.

[XCYD98] F. Xia, I.G. Clark, A.V. Yakovlev, and A.C. Davies. Petri net models of metastable operations in latch circuits (full paper). Technical Report TR627, Dept. of Computing Science, University of Newcastle upon Tyne, January 1998.

[Xia00] F. Xia. *Supporting the MASCOT method with Petri net techniques for real-time systems development.* PhD thesis, King's College London, 2000.

[XL98] F. Xia and B. Liu. Quantative investigation of certain properties of slot machanisms. In *5th UK Asynchronous Forum*, Computer Lab. Cambridge, 17–18 December 1998.

[XYC99] F. Xia, A. Yakovlev, and I.G. Clark. Testing the data freshness properties of asynchronous communication mechanisms. In *Proceedings of the 7th UK Asynchronous Forum*, Newcastle upon Tyne, UK, December 1999.

[XYS+00] F. Xia, A. Yakovlev, D. Shang, A. Bystrov, A. Koelmans, and D.J. Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *Async 2000, Proceedings of the sixth international symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000. Eilat, Israel.

[Yar96] J.M. Yarbrough. *Digital Logic Applications and Design*, chapter 5, pages 268–271. publisher unknown, 1996. ISBN 0-314-06675-6, Section 5.3.3.

[YKXK98] A. Yakovlev, D.J. Kinniment, F. Xia, and A.M. Koelmans. A FIFO buffer with non-blocking interface. *IEEE Computer Society TCVLSI Technical Bulletin*, pages 11–14, Fall 1998.

[Zub96] W.M. Zuberek. Modelling using timed Petri nets - model description and representation. Technical Report 960, Dept. of Computer Science, Memorial University, St. John's, NF, Canada. A1C 5S7, August 1996.