Asynchronous Computing
(Electronic Designer’s perspective)

Alex Yakovlev
Newcastle University
Newcastle upon Tyne, U.K.
http://async.org.uk
http://workcraft.org

First School on Reaction Systems, Toruń, Poland, 3-5 June 2019
• Simple intro examples on whiteboard
• Six Asynchronous Design Principles:
  – Asynchronous Handshaking
  – Delay-insensitive Encoding
  – Completion Detection
  – Causal Acknowledgement
  – Full Indication and Early Evaluation
  – Time Comparison
• Pros and Cons
• (Some of the) Models, Techniques and Tools for Asynchronous Design
• Asynchronous Design from Signal Transition Graphs
Asynchronous Behaviour

• Synchronous vs Asynchronous behaviour in general terms, examples:
  – Orchestra playing with vs without a conductor
  – Party of people having a set menu vs a la carte
• Synchronous means all parts of the system acting globally in tact, even if some or all part ‘do nothing’
• Asynchronous means parts of the system act on demand rather than on global clock tick
• Acting in computation and communication is, generally, changing the system state
• Synchrony and Asynchrony can be found in CPUs, Memory, Communications, SoCs, NoCs etc.
Introduction/Motivation: Simple example

David MÜLLER 1959

Question: WHEN can we see data valid?

Answer - WITH or without clock.

\[ \begin{align*}
  y_0 & := x_1 \land x_2 \\
  x_1 & := \neg y \\
  x_2 & := \neg y
\end{align*} \]

Delays are everywhere:
- gates
- wires
Key Principles of Asynchronous Design

- Asynchronous handshaking
- Delay-insensitive encoding
- Completion detection
- Causal acknowledgment (indicatability)
- Strong and weak causality (full indication and early evaluation)
- “Time comparison” (synchronisation, arbitration)
Why and what is handshaking?

Mutual Synchronisation is via Handshake
Synchronous clocking

How we think

What we design

(a)

(b)
Asynchronous handshaking

What we design

How we think

“Channel” or “Link”
Handshake latch
Handshake CL
Handshake Signalling Protocols

Level Signalling (RTZ or 4-phase)

Transition Signalling (NRZ or 2-phase)
Data Token = (Data Value, Validity Flag)
Bundled Data

Data
req
ack

Return to Zero:

Data
req
ack

One cycle

Non-Return-to-Zero

Data
req
ack

One cycle
One cycle
Delay-Insensitive encoding (Dual-Rail)

RTZ:

- Data.0
- Data.1
- ack

NRZ:

- Data.0
- Data.1
- ack

NULL (spacer)

NRZ coding leads to complex logic implementation; special ways to track odd and even phases and logic values are usually needed.
DI codes (1-of-n and m-of-n)

• 1-of-4:
  – 0001=> 00, 0010=>01, 0100=>10, 1000=>11

• 2-of-4:
  – 1100, 1010, 1001, 0110, 0101, 0011 – total 6 combinations
    (cf. 2-bit dual-rail – 4 comb.)

• 3-of-6:
  – 111000, 110100, ..., 000111 – total 20 combinations (can encode 4 bits + 4 control tokens)

• 2-of-7:
  – 1100000, 1010000, ..., 0000011 – total 21 combinations (4 bits + 5 control tokens)
Why and what is completion detection?

Signalling that the Transients are over
Bundled-data logic blocks

Completion is implicit: by done signal

The delay must scale with the worst case delay path, So … not really self-timed

Conventional logic + matched delay
True completion detection

Dual-rail logic

Completion detection for one dual-rail bit

Multi-input C-element

done

Completion detection tree
The Muller C element

\[ Z_{\text{next}} = AB + Z(A+B) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Static Logic Implementation

[van Berkel 91]
Why and what is causal acknowledgment?

Every signal event must be acknowledged by another event
Causal acknowledgment

C-element implementation using simple gates

Unack’ed transitions $x_2$- and $x_3$- may cause a hazard on output $c$

However, under Fundamental Mode (slow environment) the circuit is safe
Principle of causal acknowledgement

Each transition is causally ack’ed, hence no hazards can appear.
Why and what are strong and weak causality?

Degree of necessity of precedence of some events for other events
Strong Causality

- Petri net transitions synchronising as rendez-vous

- Logic circuits: Muller C-element (in 0-1 and 1-0 transitions), AND gate (in 0-1 transitions), OR gate (in 1-0 transitions)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Weak Causality

- Petri net transitions communicating via places

- Logic circuits: AND gate (in 1-0 transitions), OR gate (in 0-1 transitions)
Full indication versus Early Evaluation

Dual-rail AND gate with full input acknowledgement

Dual-rail AND gate with “early propagation”

Allows outputs to be produced from NULL to Codeword only when some (required) inputs have transitioned from NULL to Codeword (similar for Codeword to NULL)
Why and what is timing comparison?

Telling if some event happened before another event
Synchronizers and arbiters

- Synchronizer
  Decides which clock cycle to use for the input data

- Asynchronous arbiter
  Decides the order of inputs
Metastability is not being able to decide...

\[ \Delta t_{\text{in}} \rightarrow 0 \]
Typical responses

• We assume all starting points are equally probable
• Most are a long way from the “balance point”
• A few are very close and take a long time to resolve
Two-way arbiter (Mutual exclusion element)

Basic arbitration element: Mutex (due to Seitz, 1979)

An asynchronous data latch with metastability resolver can be built similarly
Pros...

- People have always been excited by asynchronous design, and motivated by:
  - **Higher performance** (work on average not worst case delays)
  - **Lower power consumption** (automatic fine-grain “clock” gating; automatic instantaneous stand-by at arbitrary granularity in time and function; distributed localized control; more architectural options/freedom; more freedom to scale the supply voltage)
  - **Modularity** (Timing is at interfaces)
  - **Lower EMI and smoother $I_{dd}$** (the local “clocks” tend to tick at random points in time)
  - **Low sensitivity to PVT variations** (timing based on matched delays or even *delay insensitive*)
  - **Secure chips** (white noise current spectrum)
  - **Plus, ... a lot of scope and fun for research** (there are many unexplored paths in this forest!)
So why have async designers been often “crucified” in the past?

- **Overhead** (area, speed, power)
  - Control and handshaking
  - Dual-rail and completion detection costs

- **Hard to design**
  - yes and no, ... It’s different — there are very many styles and variants to go and one can easily get confused which is better

- **Very few **practical** CAD tools** (but many academic tools)
  - Tools are quite specific to particular design styles and design niches; hence don’t cover the whole spectrum
  - Complexity of timing and performance models
  - Difficulty with sign-off (for particular frequency requirements)
  - ... But the situation is improving

- **Hard to Test**
  - Possible, but not as mature as sync
Models and techniques for design
Models and techniques for asynchronous design

• **Models:**
  – Delay model (inertial, pure, gate delay, wire delay, bounded and unbounded delays)
  – Models of environment (fundamental mode, input-output)
  – Models of switching behaviour (state-based, event-based, hybrid)

• **RTL level:**
  – Data and control paths separate (data flow graphs, FSMs, Signal Transition Graphs, Synchronised Transitions)
  – Pipeline based (Combinational logic plus registers and latch controllers, e.g. micropipelines, gate-level pipelining)
  – Process-based (CSP-like, Balsa, Haste, Communicating Hardware Processes)

• **High-level models**
  – Flow graphs (Marked graphs, extended MGs), Petri nets, Markov Chains
  – Behavioural HDLs (C, SystemC)
Gate vs wire delay models

- Gate delay model: delays in gates, no delays in wires

- Wire delay model: delays in gates and wires
Delay models for async. circuits

- **Bounded delays (BD):** realistic for gates and wires.
  - Technology mapping is easy, verification is difficult
- **Speed independent (SI):** Unbounded (pessimistic) delays for gates and “negligible” (optimistic) delays for wires.
  - Technology mapping is more difficult, verification is easy
- **Delay insensitive (DI):** Unbounded (pessimistic) delays for gates and wires.
  - DI class (built out of basic gates) is almost empty
- **Quasi-delay insensitive (QDI):** Delay insensitive except for critical wire forks (*isochronic forks*).
  - In practice it is the same as speed independent
Control Logic

• Control specification based on Petri nets (Signal Transition graphs)
Control specification

Signal Transition Graph (STG)

Timing Diagram

A input
B output
Control specification
Control specification

A+ → A
B- → B
A- → A
B+ → B

A → B

A

B
Control specification
VME bus example using Petri nets

Bus

Data Transceiver

VME Bus Controller

Device

DSr

LDS

LDTACK

D

DTACK

Read Cycle
STG for the READ cycle

LDS+ → LDTACK+ → D+ → DTACK+ → DSr- → D- → LDTACK- → LDS- → DSr+ → DTACK-
Choice: Read and Write cycles

\[
\begin{align*}
\text{DSr+} & \rightarrow \text{LDS+} \rightarrow \text{LDTACK+} \rightarrow \text{DTACK-} \rightarrow \text{DSr-} \rightarrow \text{D-} \\
\text{D+} & \rightarrow \text{LDTACK-} \rightarrow \text{DTACK+} \rightarrow \text{LDS-} \\
\text{DSw+} & \rightarrow \text{D+} \rightarrow \text{LDS+} \rightarrow \text{LDTACK+} \rightarrow \text{DTACK-} \rightarrow \text{DSw-}
\end{align*}
\]
Choice: Read and Write cycles
Workcraft tool: workcraft.org

- Framework for interpreted graph models
  - Circuits, STGs, state graphs, dataflow structures, ...
  - Interoperability between different abstraction levels
  - Consistency for users; convenience for developers
- Elaborate graphical user interface
  - Visual editing, analysis, and simulation
  - Easy access to common operations
  - Possibility to script specialised actions
- Interface to back-end tools for synthesis and verification
  - Reuse of established theory and tools (PETRIFY, MPSAT, PUNF)
Logic synthesis: xyz-example

Signal Transition Graph (STG)
Token flow

\[
\begin{align*}
x & \\
y & \\
z & \\
x^+ & \rightarrow y^+ & \rightarrow z^- \\
z^+ & \rightarrow x^- \\
y^- & \\
\end{align*}
\]
State graph
Next-state functions

\[ x = \overline{z} \cdot (x + \overline{y}) \]

\[ y = z + x \]

\[ z = x + \overline{y} \cdot z \]
1) Truth Table

<table>
<thead>
<tr>
<th>Previous state</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0<em>0</em></td>
<td>1 1 1</td>
</tr>
<tr>
<td>0 1*0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 0*</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0 0*1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1<em>0</em>1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1*</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1*1 1</td>
<td>0 1 1</td>
</tr>
</tbody>
</table>

2) Boolean Minimization

<table>
<thead>
<tr>
<th>xy z</th>
<th>00</th>
<th>10</th>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ x = \overline{z} \cdot (x + \overline{y}) \]

Observations in this example:
1) All combinations are used as states
2) All states appear uniquely
Generally, this is not always the case!
Complex Gate netlist

\[ x = \overline{z} \cdot (x + \overline{y}) \]

\[ y = z + x \]

\[ z = x + \overline{y} \cdot z \]
Circuit synthesis

• Goal:
  – Derive a hazard-free circuit under a given delay model and mode of operation
Speed independence

- **Delay model**
  - Unbounded gate / environment delays
  - Certain wire delays shorter than certain paths in the circuit

- **Conditions for implementability:**
  - Consistency
  - Complete State Coding
  - Persistency
Implementability conditions

- **Consistency**
  - Rising and falling transitions of each signal alternate in any trace

- **Complete state coding (CSC)**
  - Next-state functions correctly defined

- **Persistency**
  - No event can be disabled by another event (unless they are both inputs)
Implementability conditions

• Consistency + CSC + persistency

• There exists a speed-independent circuit that implements the behavior of the STG

(under the assumption that any Boolean function can be implemented with one complex gate)
Persistency

100 \rightarrow a^- \rightarrow 000 \rightarrow c^+ \rightarrow 001

\downarrow b^+ \downarrow b^+

is this a pulse?

Speed independence \Rightarrow \text{glitch-free output behavior under any delay}
Conclusion

• Asynchronous design is based on rigorous principles of causality and concurrency

• It allows electronic circuits to operate without centralised time constraints and adapt to any structural or behavioural variations and noise

• Nature is largely asynchronous as it is typically structured around energy supplies; synchronisation takes place in a distributed way

• Research question: Are reaction systems asynchronous? And in what way they are or can be asynchronous?

• Attend the workshop on “Bringing Asynchrony to Reaction Systems”!

• I encourage development of a Reaction Systems modelling plug-in under Workcraft.org