Crossing the Synchronous-Asynchronous Divide

Mika Nyström (*Caltech*)
Alain J. Martin (*Caltech*)
Outline of Talk

• Where we’re coming from: quasi delay-insensitive, “slack-elastic” asynchronous VLSI.

• Synchronous and asynchronous models of computation and the friction between them.

• Two “typical” interfacing problems: the synchronizer and the clock-event generator.
Synchronous Design Style

All units are “combinational logic” without internal synchronization.

Advantage:
- Simple circuits.

Disadvantages:
- Enforces that all units have the same delay.
- Clock period is the maximum delay over all units over all data.
- Lots of unnecessary synchronization.
- Breaks modularity.
“Traditional” Asynchronous Style

Delay lines are used to match the delays of each unit, allowing each unit to compute at its own rate.

Advantages:
- Modular.
- Units that aren’t used use no power.

Disadvantages:
- Complicated—lots of delay lines to tune.
- Difficult to make each unit *internally* have data-dependent delays.
Delay-insensitive Asynchronous Style

Originally explored by D. Muller in the 1950s. “Quasi” delay-insensitive (Martin 1990) style makes data values themselves encode the timing.

Advantages:
• No clock!
• Low energy: units that aren’t used use no power.
• Self-adjusting w.r.t. environmental conditions.
• No “margins.”
• No global synchronization: easy to compose.

Disadvantage:
• Handshake-based communication uses more wires.
Communicating Hardware Processes

- Hardware design as distributed programming.
- The language used is a descendant of Hoare’s CSP.
- Sequential processes that communicate on FIFO channels.
"MIPS RISC Architecture"

informal translation

Sequential CHP

process decomposition

Parallel CHP

process decomposition

Parallel CHP

compilation

HSE, PRS (circuits)

layout

Layout

Fab
Deterministic Asynchronous Processes

- Each hardware process is deterministic w.r.t. its inputs.
- Only the sequence of values on the channels matters ("slack elasticity"—related to the dataflow model (Dennis 1976, 1980).
- Unrelated processes are not synchronized with each other, but all values sent on channels can be determined from the program, input, and initial state.
- Timing-independent behavior.
Synchronous-Asynchronous Divide
Synchronous-Asynchronous Divide
Delay-Insensitivity

At lowest level, production rule set:

\[ G_{x,c}(\Sigma) \rightarrow x := c \]

where \( c \in \{\text{true, false}\} \) and \( \Sigma \) represents the state of the system (value of all variables).

Quasi delay-insensitivity:

- Each PR is stable and non-interfering.
- QDI achieved by using four-phase signaling.

- Allows us to build deterministic systems.
DI Communication Protocols

HSE requires:

- Data as delay-insensitive code, dual-rail, other 1-of-\( n \).
  One-bit channel \( \mathbf{Z} \equiv (z_0, z_1, ze) \) (Sims 1966)

- Example: 4-phase dual-rail transfer:

  \[
  \begin{align*}
  &\text{Send zero} : z_0 \uparrow; \ [\neg ze]; \ z_0 \downarrow; \ [ze] \\
  &\text{Send one} : z_1 \uparrow; \ [\neg ze]; \ z_1 \downarrow; \ [ze] \\
  &\text{Receive} : [z_0 \lor z_1]; \ ze \downarrow; \ [\neg z_0 \land \neg z_1]; \ ze \uparrow
  \end{align*}
  \]

![Diagram of data transfer](image)

Alternative: 2-phase protocol:

\[
\begin{align*}
&\text{Send zero} : z_0 \uparrow; \ [\neg ze]; \ldots; \ z_0 \downarrow; \ [ze] \\
&\text{Send one} : z_1 \uparrow; \ [\neg ze]; \ldots; \ z_1 \downarrow; \ [ze]
\end{align*}
\]

- Much more difficult to decode (CMOS is level-sensitive).
Some Designs Done This Way

- Distributed mutual exclusion element (1985)
  - 200 T
- First asynchronous microprocessor (1989):
  - 23 kT
  - 2-µm CMOS; 18 MIPS in 1.6-µm CMOS
  - Works from 0.4 V to 12 V & runs on a potato as power supply
  - Successfully remapped to GaAs
- DSP filter (1995)
  - 500 kT
- MiniMIPS microprocessor (1998)
  - 2 MT
    - Almost-complete implementation of MIPS-1 ISA
    - 180 MIPS/3–7 W in 0.6-µm CMOS
    - Layout error led to ≈20% speed loss
- “Lutonium” 8051 microcontroller
  - Work in progress 2002
  - Design goal: very low power and high performance
MiniMIPS Block Diagram

Synchronous-Asynchronous Divide
Asynchronous Pulse Logic

• Variation on theme: introduce easy-to-satisfy timing constraints without destroying modularity. (Nyström 2001, Nyström & Martin 2002)

Use “pulse”—two-phase timing with four-phase signals:

\[ \text{Send zero} : [ze]; \; \cdots ; \; z0\uparrow; \; z0\downarrow \]
\[ \text{Send one} : [ze]; \; \cdots ; \; z1\uparrow; \; z1\downarrow \]
\[ \text{Receive} : [z0 \lor z1]; \; ze\downarrow; \; \cdots ; \; ze\uparrow \]
The Divide

\[ G_{x,c}(\Sigma) \rightarrow x := c \]

Key properties that make it possible to design QDI circuits:

- **Stability**
  - \( G_{x,c}(\Sigma) \) holds at least until \( x := c \) is complete

- **Noninterference**
  - \( x := \text{true} \) and \( x := \text{false} \) are never enabled at the same time

- Imply partial-ordering constraints on signal transitions.
Synchronous Model

The QDI model is *fundamentally incompatible* with the standard synchronous model:

- **QDI model** doesn’t include timing information (it would defeat the purpose to include it).
Crossing Over

How can we build systems that combine synchronous and asynchronous elements?

• Common answer: don’t. (In fact, most common answer is: “avoid asynchronous.”)

• Use “synchronous synchronizers”: synchronize all inputs to the synchronous subsystems. (Problem: metastability can cause synchronization failure.)

• Pausable clocks. (Standard GALS.)

• This work: let the clocked subsystems run as usual and use “asynchronous synchronizers” on the asynchronous side.
First Problem: The Synchronizer

How do we “inspect” a variable that the environment may change the variable at any time?

- $x$ can change at any time.
- Perform a four-phase handshake on $r$ indicating the value of $x$ when $re$ was asserted.

Either:

$$r0\uparrow; \ [\neg re]; \ r0\downarrow; \ [re]$$

or

$$r1\uparrow; \ [\neg re]; \ r1\downarrow; \ [re]$$
Possibilities
A Clearer Specification

We settle on:

• If $x$ is stable within some finite time interval (the confusion interval) of when $re$ went true, we want to produce the value of $x$ on $r$.

• If $x$ changed during the confusion interval, we produce either zero or one (but not both!)

• Must involve metastability: delay of synchronizing cannot be bounded.
When do we Need it?

When is it necessary to use a synchronizer? Usually demanded by some permissive specification:

- Interrupts that need not be acknowledged. (MIPS specification, Martin et al. 1997)
- Networking protocols that need to be implemented robustly. (Marshall et al. 1994)
Why is it Hard?

The difficulty is that the input is *unstable*: a direct implementation is vulnerable to the following sequence:

- *re* arrives when $x$ is *false* and the circuit begins to enter the output-zero state
- the output switches far enough to disable the output-one state
- $x$ changes to *true* before the output-zero state has been properly entered
Solution

The solution we have proposed is to introduce a new, intermediate pair of variables $a_0$ and $a_1$ that “integrate” $x$ and $\neg x$ and are monotonic until they are acknowledged (and therefore stable):

- Once $re$ goes true we’re guaranteed that at least one of $a_0$ and $a_1$ will eventually go true and stay true.
Finishing the Synchronizer

With the new guarantee, it’s relatively easy to complete the synchronizer.

\[
SEL \equiv *\left[ [a_0 \rightarrow r_0\uparrow; [\neg a_0 \land \neg a_1]; r_0\downarrow \right.
\left. [a_1 \rightarrow r_1\uparrow; [\neg a_1 \land \neg a_0]; r_1\downarrow \right]
\]

Circuit implementation:
Analysis

- Very similar to a standard mutual-exclusion element (arbiter): only two more transistors to check for the \( \neg a_0 \land \neg a_1 \) state.

- Can enter a metastable state when \( a_0 \) and \( a_1 \) are both true at the same time (as expected).
An Asynchronous Clock Circuit

We don’t want to use a clock, but we still want to keep track of the time of day.

Idea:

- Convert a clock signal into a sequence of asynchronous handshakes.
- Count the handshakes: if we have seen $n$ handshakes, we know that the time of day is at least $n$, and at most $n + s$ where $s$ is the slack of the system.
An Asynchronous Clock Circuit

At first, the clock signal seems to have all the properties of the variable $x$ in the synchronizer example: it is unstable w.r.t. the actions of the asynchronous system, but:

- We know that it will change again.
- We don’t care about the value of the clock at any particular time.

Solution:

- Integrate the clock signal as we did in the synchronizer.
- Wait until both $a0$ and $a1$ have appeared; perform the handshake.
- Reset $a0$ and $a1$ and repeat.
Schmitt triggers rule allow us to bound the rise and fall times of the signals $qt$ and $qf_-$ since their inputs are monotonic (Greenstreet 1999).
Summary

- Overview of asynchronous design style and quasi delay-insensitive design.

- A good way to exploit the asynchrony: “slack elastic” design.

- Reading a variable at an arbitrary time: the synchronizer.

- Counting pulses asynchronously: the asynchronous clock circuit.