Advanced Digital Design
Static Data-flow Structures

by A. Steininger and J. Lechner
Vienna University of Technology
Outline

- Recap: Handshake Protocols
- Data-flow Abstraction
  - from handshakes to tokens
  - latches & pipelines
  - flow control elements
  - function blocks
- Implementation
Asynchronous Circuits

local handshake protocol

recall
Handshake Protocols

- **Handshake style**
  - 4-phase / Return-To-Zero
  - 2-phase / Non-Return-To-Zero

- **Timing model**
  - Bundled data
  - Delay-Insensitive

*Choice of protocol affects speed, area, power*
## Comparing the Styles

<table>
<thead>
<tr>
<th>Handshake Style</th>
<th>4-phase (RTZ)</th>
<th>2-phase (NRZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Coding</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Rail</td>
<td>Bundled Data</td>
<td></td>
</tr>
<tr>
<td>Multirail</td>
<td>NCL</td>
<td>LEDR</td>
</tr>
<tr>
<td><strong>Signaling</strong></td>
<td>Level</td>
<td>Transition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delay Model</th>
<th>Single Rail</th>
<th>Multirail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded</td>
<td>QDI</td>
<td></td>
</tr>
<tr>
<td><strong>ACK</strong></td>
<td>Explicit Handshake</td>
<td></td>
</tr>
<tr>
<td><strong>REQ</strong></td>
<td>Explicit</td>
<td>Completeness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Determination</td>
</tr>
</tbody>
</table>
Data-flow Abstraction

- Asynchronous data-flow circuits
  - *components* communicate via *channels*
  - *channels* may implement any protocol/coding

- Building blocks (components):
  - Latches
  - Flow control elements: Join, Fork, Merge, MUX, DEMUX
  - Functional blocks

RTL-like modeling of asyn. circuits
Latches – The Concept

- the ONLY components with active part in handshaking:
  - process requests from “upstream” latches => overwrite old data
  - issue requests to “downstream” latches => provide new data

Abstraction: Tokens and Bubbles

- **Tokens**: new data to be captured, move downstream
- **Bubbles**: “old” data to be overwritten, move upstream
Latches - Symbols

- Tokens marked with circles
- E.g., 4-phase protocol

\[
\begin{align*}
E & \quad \text{empty codeword (NULL)} \\
V & \quad \text{value codeword (DATA)}
\end{align*}
\]
Latches – Switching Rule

- Latch L1 may store data iff:
  - all direct successor latches (downstream) have consumed (stored and acknowledged) L1’s current value
  - L1 thus stores a bubble
  - all direct predecessor latches (upstream) provide new data
  - L1’s predecessor(s) store a token
  - then the token moves on to L1
Latches – Pipelines
Elastic Pipeline

Recall...
Elastic Pipeline

initial state

\[ R_{IN} \rightarrow C \rightarrow C \rightarrow C \rightarrow A_{OUT} \]

\[ C \leftrightarrow C \leftrightarrow C \]

\[ E \rightarrow E \rightarrow E \rightarrow E \]
Elastic Pipeline

request (rising edge) / token

C C C

R_{IN} A_{OUT} R_{OUT}

V E E E
Elastic Pipeline

request/token passes stage 1
Elastic Pipeline

request/token passes stage 2
Elastic Pipeline

request/token passes stage 3 => output
Elastic Pipeline

Further request/token
Elastic Pipeline

new request/token passes stage 1
new request/token passes stage 2

Elastic Pipeline

R_{IN} \quad C \quad C \quad C \quad A_{OUT} \quad R_{OUT}

E \quad E \quad V
Elastic Pipeline

new request/token blocked for stage 3
Elastic Pipeline

one more request/token ...

![Diagram of Elastic Pipeline](image)
Elastic Pipeline

... passes stage 1 only
Elastic Pipeline

ACK consumes the rightmost token
Elastic Pipeline

bubble moves upstream ...

Diagram showing a pipeline with labels labeled as follows:
- $C$: Execution stage
- $V$: Virtualization stage
- $E$: Execution stage

Input signals:
- $R_{IN}$
- $R_{OUT}$

Output signals:
- $A_{OUT}$
Elastic Pipeline

... up to stage 1
Pipeline - Rings

- Cycles for iterative computations
- Initialization of registers:
  - At least one bubble
  - At least one value and one empty token
  - Wrong initialization causes a deadlock
Control Flow Elements

- Transparent for handshaking
- Unconditional: Join, Fork, Merge
- Conditional: MUX, DEMUX

Join
Fork
MERGE
MUX
DEMUX
Control Flow - Join

- Join synchronizes inputs
- Requests are passed on when all inputs are available

Missing input

All inputs ready
Control Flow - Join

- Join synchronizes inputs
- Requests are passed on when all inputs are available

Missing input  
Join blocks execution

All inputs ready  
Tokens passed on
Control Flow - Fork

- Requests are passed on to multiple successor stages
- Requests are passed on to multiple successor stages
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed

Token arrives at input 0
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed

Value token is passed on
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed

Value token is consumed, receiver sends ack and demands empty token
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed

Empty token arrives
Control Flow - Merge

- Passes input tokens to output
- Mutually exclusive inputs!
- Full handshake routine needs to be performed

Empty token is forwarded
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected

Input 0 is selected
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected

Value token is passed on
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected

Value token is consumed, receiver sends ack and demands empty tokens
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected

Empty tokens arrive
Control Flow - MUX

- Passes selected input to output
- The complete handshake routine needs to be performed before another input can be selected

Empty token is forwarded
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected

Output 0 is selected
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected

Value token is passed on to output 0
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected

Value token is consumed, receiver sends ack and demands empty tokens
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected

Empty tokens arrive
Control Flow - DEMUX

- Passes input to selected output
- The complete handshake routine needs to be performed before another output can be selected

Empty token is transferred to output 0
Functional Blocks

- Combinational logic
- Perform computation
  1. Block waits for tokens on inputs
  2. Performs function
  3. Issues output tokens containing the results

Implicit Join/Fork

Explicit Join/Fork
Functional Blocks

- Transparent for handshaking
- Relay handshaking signals unmodified
- For every input request exactly one request must be produced at output

The function block is transparent for the latches in terms of handshaking
Functional Blocks

- **Strongly indicating**: function block waits for tokens on *all* inputs before producing output tokens.
  Worst-case latency: slowest input determines speed.

- **Weakly indicating**: function block may produce some output tokens as soon as they can be computed (based on some subset of inputs).
  Actual-case latency.
Implementation

- Many different implementation styles
- Implementation depends on used handshake protocol
  - 4-phase/2-phase
  - Bundled data/DI encoding
- Different area complexity
  - 4-phase ≤ 2-phase
  - Bundled data ≤ DI encoding
- Protocols may be mixed
  - Optimization of performance/area/power
Implementation

- Function blocks
  - Example: Full adder

- Control Flow
  - MUX
  - DEMUX
  - Fork
  - Join
Adder - Bundled Data

- Single rail implementation
- Matched delay
  - E.g., inverter chain
  - Asymmetric delays for 4-phase
Adder – Dual Rail I

- Delay-insensitive Minterm Synthesis (DIMS)
- For dual-rail 4-phase functions
- DNF built with Muller C-gates

Source: [Sparso 06]
Adder – Dual Rail II

- NCL threshold gates with hysteresis
  - Proposed by Theseus Logic, Inc.
  - High when m-of-n inputs high
  - Low when all inputs low

Source: [Sparso 06]
Join - Bundled Data

4-phase join

Source: [Sparso 06]
Fork - Bundled Data

4-phase fork

Source: [Sparso 06]
Join - Dual Rail

4-phase join

x-ack ←

y-ack ←

x.t ←

x.f →

y.t →

y.f →

z.ack

z0.t

z0.f

z1.t

z1.f

Source: [Sparso 06]
Fork - Dual Rail

4-phase fork

Source: [Sparso 06]
MUX - Bundled Data

4-phase MUX

Source: [Sparso 06]
DEMUX - Bundled Data

4-phase DEMUX

Source: [Sparso 06]