Asynchronous Dataflow: MOUSETRAP Pipelines

Montek Singh
UNC Chapel Hill

ASYNC 2022 Summer School
Week 2: June 13
MOUSETRAP Pipelines

[Singh/Nowick, ICCD 2001 & TVLSI 2007]

Simple asynchronous implementation style, uses...

- transparent \textit{D-latches} + standard combinational function logic
- \textit{simple control}: 1 gate/pipeline stage

Uses a “capture protocol”: Latches are ...

- normally \textit{transparent} while waiting for data
- become \textit{opaque} after data arrives

Control Signaling: transition-signaling $= 2$-phase

Goals:

- fast cycle time
- simple inter-stage communication
- standard cell implementation
MOUSETRAP: A Basic FIFO

Stages communicate using *transition-signaling*:

1 transition per data item

Data Latch

Latch Controller

Stage \( N-1 \)

\( \text{req}_N \)

\( \text{done}_N \)

\( \text{ack}_N \)

Stage \( N \)

Stage \( N+1 \)

Data in

Data out

2\textsuperscript{nd} data item flowing through the pipeline
MOUSETRAP: A Basic FIFO (contd.)

Latch controller (XNOR) acts as "phase converter":

- 2 distinct transitions (up or down) → pulsed latch enable

Latch is disabled when current stage is "done"
MOUSETRAP: FIFO Cycle Time

Cycle Time: $T_{LATCH} + T_{XNOR}$

Fast self-loop: $N$ disables itself

$N$ re-enabled to compute $N+1$ computes itself

Stage $N$: Data Latch

Data in $\rightarrow$ Stage $N-1$ $\rightarrow$ Data Latch $\rightarrow$ Stage $N+1$ $\rightarrow$ Data out

$ack_{N-1}$ $\rightarrow$ req$_N$ $\rightarrow$ Latch Controller

$done_N$ $\rightarrow$ $ack_N$ $\rightarrow$ req$_{N+1}$
MOUSETRAP: Pipeline With Logic

Simple Extension to FIFO:
insert *logic block* + *matching delay* in each stage

"Bundled Data" Requirement:
- each "req" must arrive *after* data inputs valid and stable
A fork stage has two (or more) successors

- same data and req sent to all
- wait for ack from all
Fork

In Ben's talk: **COPY, n-way link**

*[In?x; Out₀!x, ..., Outₙ₋₁!x]*
Join

* A join stage has two (or more) predecessors
  - wait for data from all
  - same ack sent to all
Join

*\[ \text{In}_0?\text{arg}_0, \text{In}_1?\text{arg}_1, \ldots, \text{In}_{n-1}?\text{arg}_{n-1}; \]
*\[ \text{Out}!\text{func}(\text{arg}_0, \text{arg}_1, \ldots, \text{arg}_{n-1}) \]

In Ben’s talk: FUNCTION, OPERATOR
Arbitration Stage

★ Two input channels, two output channels
★ Only one input read
  ● whichever arrives first
  ● ... goes out on the corresponding output

★ Seitz’ Mutex is the core
  ● [from Brunvand’s thesis]
  ● surrounding logic adapts it to transition signals
Merge without (or after) Arbitration

A merge stage has two (or more) predecessors
- data is taken from whichever input channel has a new request

Assumption:
- no arbitration needed
- input channels are mutually exclusive
Merge without (or after) Arbitration

* Datapath

- mux controlled by change detectors on input channels

In Ben’s talk: MERGE, MIXER
Conditional Select (or event mux)

* Two data inputs and one select
  - first read select
  - then:
    - based on select, read one of the inputs
    - do not read the other input

* datapath
  - mux + latch
Conditional Select (or event mux)

In Ben's talk: MUX, CONTROLLED MERGE

*[C?c;
  [  c=0  ->  In_0?x
    []  c=1  ->  In_1?x
  ];
  Out!x
]*
Conditional Split (or router)

* One data input and one select
  * first read data + select
  * then:
    - based on select, send data along one output channel

In Ben’s talk: DEMUX, SPLIT
Conditional Split (or router)

**In Ben’s talk:** DEMUX, SPLIT

```
* [In?x, C?c;
  [ c=0 -> Out₀!x
  [] c=1 -> Out₁!x
 ]
]*
```
Timing Analysis

Setup constraint: matched delay

Hold Time constraint:

Data must be safely “captured” by Stage N before new inputs arrive from Stage N-1

- Stage N’s “self-loop” faster than entire path through previous stage
Example: Greatest Common Divider
Euclid’s GCD algorithm

gcd(a, b)
  while (b != 0)
    if(a>b)
      a = a - b
    else
      b = b - a
  return a

Example
  * gcd(42, 28)
  * (14, 28)
  * (14, 14)
  * (14, 0)
  * 14
Euclid’s GCD algorithm

gcd(a, b)

while (b != 0)
    if (a > b)
        a = a - b
    else
        b = b - a

return a
gcd(a, b)

while (b != 0)

\[
s = a - b \\
\text{if}(s < 0) \\
\text{swap}(a, b) \\
\text{else} \\
a = s
\]

return a

* Example
  * gcd(42, 28)
  * (14, 28)
  * (28, 14)
  * (14, 14)
  * (0, 14)
  * (14, 0)
  * → 14
while (b != 0)
    s = a - b
    if (s < 0)
        swap(a, b)
    else
        a = s
return a
GCD implementation

while (b != 0)
    s = a - b
    if (s < 0)
        swap(a, b)
    else
        a = s
return a
GCD implementation

```
while (b != 0)
    s = a - b
    if(s<0)
        swap(a,b)
    else
        a = s
return a
```

Unroll 8 times!

Pipeline subtract into 8 stages

Pipeline swap into 4 stage
GCD implementation

while (b != 0)
    s = a - b
    if (s < 0)
        swap(a, b)
    else
        a = s
return a
GCD chip

* Layout and fab:
  - 0.13um, standard cell
GCD chip

* Testing:
  - Full scan for latches
  - Combinational test patterns generated (Shi 2005) gave 98% coverage
  - Functional tests for timing violations (Gill 2006)

* Evaluation:
GCD chip: Results

Throughput (Mega operations/second)

Occupancy (# data items in ring)

2069 MHz

1010 MHz
GCD chip: Results

* Operates correctly over a wide voltage range
GCD chip: Results

* Impact of voltage variation on power consumption

![Graph](image-url)

- **2.1V**
- **1.8V**
- **1.5V**
- **1.2V**
- **0.9V**
GCD chip: Results

* Impact of temperature performance

![Graph showing the impact of temperature on Giga Iteration Completions per Second. The graph indicates a downward trend as temperature increases.](image-url)
References


