Asynchronous FIFOs

Honors Discussion #14
EECS150 Spring 2010
Chris W. Fletcher
Big Picture

• Many weeks ago + 1: **Synchronous** pipelines & data transactions

• Many weeks ago: **Asynchronous** pipelines & data transactions

• **This week:** \{Synchronous, Asynchronous*\} FIFOs

* JohnW covered Synchronous FIFOs, so we’ll stick to two+ clock domains
Motivation

• We want to pass data across clock domains
• Well, why not use a data/hold register?

footnote:
... with as high a throughput as possible

• Applications
  • Rate matching video interfaces
  • Communicating to off-chip components
  • Bulk data transfer/DMA across a chip
Why is this hard?

• Metastability
  – The ball getting stuck at the top of the hill

• Incorrect synchronizer outputs
  – The ball falling down the wrong side of the hill

  * Keep this case in mind throughout the hour

• Determining full/empty signals on time

Recall:

\[
\begin{align*}
\text{Ready} & \quad \text{We want} \\
\text{Valid} & \quad \text{We get} \\
\text{Clock} & \quad \text{Synchronizer delay}
\end{align*}
\]

“We almost full” and “Almost empty” are used to fix this problem
Full & Empty

- **Disclaimer:** This is the hardest part of Async FIFO design!

- **Out loud:** Why doesn’t the synchronous FIFO counter work?

- **First-draft solution:**
  
  *Keep 2 counters and synchronize across clock boundaries*
  
  (we’ll see what this looks like in several slides)

- **Caveat:** leads to “pessimistic” full/empty
Pessimistic State Signals

• Full goes high exactly when the FIFO fills
  ... but doesn’t learn that the FIFO gets read until several cycles after the fact (Synchronizer latency)

• Same story for the empty signal

• The good
  – This guarantees no {over, under} flow
  – Works well when we burst data
    (when the FIFO is between full and empty)

• The bad
  – Works badly when the FIFO is in the full/empty state most of the time
    **Why? Every time the FIFO goes full/empty, we impose the synchronizer delay**
Proposal #1

- Pulse based inc/dec
- Resources
  - $2n$ counter FFs
  - $2n$ pointer FFs
  - 4 synchronizers FFs

- Does this design work?
  → No!
Proposal #2

- Binary pointers
  - Direct comparison

- Resources
  - $2n$ pointer FFs
  - $2n + 4$ synchronizer FFs

- Does this design work?

→ In Theory, but can we do better?
Gray Code (GC) Primer

• 1 bit edit distance between adjacent words
• Most useful gray codes are powers of 2 long
  – Even gray code sequences are possible, but typically require more resources to decode
  – Odd gray code sequences are not possible. Why?
• (Right) An efficient mirror-image gray code scheme
  – Quadrants are colored
  – Notice that the MSBs show the counter’s “quadrant”
    – Can be used to generate { , almost} {full, empty}
  • Con: gray code schemes usually require GC←→Binary conversions

UCB EECS150 Spring 2010, Honors #14
Proposal #3

- Gray code pointers
  - Direct comparison
  - Requires GC$\leftrightarrow$Bin
- Resources
  - 2n pointer FFs
  - 4n synchronizer FFs
  - GC$\leftrightarrow$Bin converters

Does this design work?

→ In Theory
Binary vs. Gray code (#2 vs. #3)

• #2 can pass arbitrary values over the clock boundary
  – #3 is limited to increments/decrements
• #2 allows for arbitrary FIFO depth
  – #3 is best suited to powers of 2
• #2 can calculate arbitrary “almost {full, empty}”
• #3 can efficiently calculate some “almost {full, empty}” thresholds (based on counter quadrant)

... but ...

• #2 imposes a handshake latency through using data synchronizers
  (this is a serious problem for throughput!)
Acknowledgements & Contributors

Slides developed by Chris Fletcher (4/2010).

This work is partially based on ideas from:
(1) “Simulation and Synthesis Techniques for Asynchronous FIFO Design”
(2) “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparison”

This work has been used by the following courses:
– UC Berkeley CS150 (Spring 2010): Components and Design Techniques for Digital Systems