Synchronous \{Pipelines, dataflow\}

Honors Discussion #4
EECS 150 Spring 2010
Chris W. Fletcher
Today

- SCORE commentary
- Synchronous pipelines
- Throughput case study
Big Picture

• This week: **Synchronous** pipelines & data transactions

• Next week: **Asynchronous** pipelines & data transactions

• After that: ... putting it all together...

  \{Synchronous, Asynchronous\} FIFOs
Synchronous Pipelines

• Where do we start?

With the simple case...

This circuit is called a **shift register or (de)serializer**

• If you put logic in between each stage...

You get a pipeline!
Synchronous Pipelines

• When/how does this work?
  – System puts data into the pipeline
    ... knowing that it will come out at a known time

– Considerations
  • Must know each module’s “latency” (doesn’t have to == 1)
  • Make sure the output is ready by the time it sees the data!
Synchronous Pipelines

• What happens when you want to stop?

(i.e. output isn’t ready when the data arrives)

Train wreck!

How do we fix this?

Add control logic
Synchronous Pipelines

• Monolithic/lock-step approach:
  1. Controller-timed
     • Controller keeps time
     • Handles 1/2+ cycle stages
  1. Self-timed
     • Each module keeps time
     • Handles 1/2+ cycle stages
Synchronous Pipelines

• Monolithic/lock-step pitfall
  – Each can handle multi-stage operations
  – But what does this do to performance?
  – Well, NO data moves in any stage...
  – until the slowest stage is done.
  – Consider (try to find the bottleneck assuming lock-step):
Synchronous Pipelines

• “Decoupled” approach

- No central controller
- Latency Insensitive
- Each module keeps track of its own time
- Data moves at the rate of each module, not the rate of the slowest piece
- Sound familiar?
Throughput Considerations

• A look under the covers
  – What is this register?
    \{E,R,S\} match the Virtex-5
  – What events can occur?

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Clear</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>???</td>
</tr>
</tbody>
</table>

What are the implications of this state being unreachable?
Throughput Considerations

• What we want: full throughput when...

  Sender is constantly sending (InValid = 1 always)
  Receiver is constantly receiving (OutReady = 1 always)

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Clear</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>???</td>
</tr>
</tbody>
</table>

But can we ever get ideal throughput?
Homework

• Thought problem
  – Fix the throughput issue
    (allow for ideal throughput)

• (More) reading will be posted
Acknowledgements & Contributors

Slides developed by Chris Fletcher (2/2010).

This work is based in part on slides by:
Krste Asanovic, John Wawrzynek, and John Lazzaro

And is based on ideas by:
Greg Gibeling

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