EE108A

Lecture 11: Pipelining

Announcements

• Lab 5: easier to implement than lab 4, but harder to debug

• Quiz 2 will be on 11/14

• Metastability demonstration at next week’s lecture (11/7)
  – When good flip-flops go bad
  – This is cool because no one every demonstrates this except Bill!
Some Comments on Labs 4 and 5

- Complete your design before you start coding
  - Understand your interfaces
  - Prepare a timing table (remember it takes a cycle to read a RAM/ROM)
  - Partition functions
- Keep it simple
  - Address each feature in exactly one place
  - Don’t change interfaces
- Incremental debugging
  - Get one piece working at a time
  - Find out where the signal stops
- Think about “event flow”
  - What event causes things to happen, then what event
    - e.g., new_frame, then next_note, etc...

System Design – a process

- **Design (25% of the time)**
  - Specification
    - Understand what you need to build
  - Divide and conquer
    - Break it down into manageable pieces
  - Define interfaces (this takes practice to get it right)
    - Clearly specify every signal between pieces
  - Hide implementation
  - Choose representations
  - Timing and sequencing
    - Overall timing – use a table
    - Timing of each interface – use a simple convention (e.g., valid – ready)
  - Add parallelism as needed (pipeline or duplicate units)
  - Timing and sequencing (of parallel structures)
  - Design each module
- **Code (25% of the time)**
- **Verify (50+% of the time)** <= Note the time here. Plan for it.

Iterate back to the top at any step as needed.
Making things Faster

• **Question:**
  - Why do people design circuits for computation?
  - Why not just use a programmable processor and a compiler?
  - Verilog is so much harder to program and debug...what’s the benefit?

• **Answer:**
  - Performance (parallelism)
  - Efficiency (just use what you need)

• You can tailor the circuit to be just the right tradeoff for your design
• Exactly as much computation power as you need
• **Result:** Much higher (100x or more) efficiency than a generic processor
  but...much higher (1000x or more) cost to develop
  (when does this tradeoff make sense?)

---

Two ways to make things faster: pipeline and parallel

A → B → C → D

Master

A B C D

A B C D
Pipelines

More Like an Assembly Line
Pipelines

- Like an assembly line – each pipeline stage does part of the work and passes the 'workpiece' to the next stage

- Example 1: Pipelined 32b Adder

Split into 4 8-bit adders
Split into stages
4 problems ‘in process’ at once

Pipeline Register
(Stores intermediate results)

Pipeline Diagram
Illustrates pipeline timing

<table>
<thead>
<tr>
<th>Cycle</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>1</td>
<td>15.8</td>
<td>15.8</td>
<td>15.8</td>
<td>15.8</td>
<td>15.8</td>
</tr>
<tr>
<td>2</td>
<td>23.16</td>
<td>23.16</td>
<td>23.16</td>
<td>23.16</td>
<td>23.16</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cycle 2

- $b[31:24]$  
- $a[31:24]$  
- $b[23:16]$  
- $a[23:16]$  
- $b[15:8]$  
- $a[15:8]$  
- $b[7:0]$  
- $a[7:0]$  
- $c0$  

Cycle 3

- $b[31:24]$  
- $a[31:24]$  
- $b[23:16]$  
- $a[23:16]$  
- $b[15:8]$  
- $a[15:8]$  
- $b[7:0]$  
- $a[7:0]$  
- $c0$
Cycle 3

Pipeline Diagram
Illustrates pipeline timing

• Takes 3 cycles to fill the pipeline
• Then 1 output per cycle
• Takes 3 cycles to drain the pipeline
Latency and Throughput of a Pipeline

- Suppose before pipelining the delay of our 32b adder is 3200ps (100ps per bit) and this adder can do one problem each 3200ps for a throughput of 1/3200ps = 312Mops

- What is the delay (latency) and throughput of the adder with pipelining?

Suppose $t_{\text{DCQ}} = 100\text{ps}$, $t_s = 50\text{ps}$, $t_k = 50\text{ps}$ (200ps Overhead for each FF)

- $t_{\text{pipe}} = n(t_{\text{stage}} + t_{\text{DCQ}} + t_s + t_k) = 4000\text{ps}$
- $\Theta = n/t_{\text{pipe}} = 1/(t_{\text{stage}} + t_{\text{DCQ}} + t_s + t_k) = 0.0004$ Gops

- So it now takes 4000ps for the first result, but we get one every 1000ps
- Big win if you’re doing lots of adds back-to-back…
  …loss if you only do them every 4 cycles or less.

Example 2: Processor Pipeline
Example 3: Graphics rendering pipeline

- Triangle pipeline:
  - Xform
  - Clip
  - Light
  - Rasterize

- Fragment pipeline:
  - Shade
  - Composite
  - Textures
  - Z-buffer

Example 4 – Packet Processing Pipeline

- Framer
- Policing
- Route Lookup
- Switch Scheduling
- Queue Mgt
- Output Scheduler
- Framer

And each of these modules is internally pipelined

You get the idea. Lots of systems are organized this way.
Pipelines: Key Point

- **Slower for a single problem** (overhead from FFs and partitioning)
  - If we can input one new problem per cycle we can get an output every cycle
  - By splitting up the logic the cycles get shorter
  - So, if we can keep the pipeline full, we can get dramatically better throughput.
  - But it’s rarely so easy...

Issues with pipelines
(all deal with time per stage)

- **Load balance (across stages)**
  - one stage takes longer to process each input than the others – becomes a ‘bottleneck’
  - Example
    - Rasterizing an ‘average’ triangle in a graphics pipeline takes more time than ‘lighting’ its vertices.
- **Variable load (across data)**
  - A given stage takes more time on some inputs than others
  - Example
    - The the time needed to rasterize a triangle is proportional to the number of fragments in the triangle. The average triangle may contain 20 fragments, but triangles range from 0 to over 1M
- **Long latency**
  - A stage may require a long latency operation (e.g., texture access)

- **Rigid Pipeline**: each stage takes the same time, so all stages must take the maximum time. (Inefficient if some are waiting for others.)
Load Balancing Pipelines

- Suppose transform takes 2 cycles and clip 4 cycles
- Clip is a ‘bottleneck’ pipeline stage
- Xform unit is busy only half the time

Load Balancing Solutions
1 – Parallel copies of slow unit
Load Balancing Solutions
2 – Split slow pipeline stage

When is it better to split? To copy?
Throughput and latency are the same
Variable load
Stage A always takes 10 cycles.
Stage B takes 5 or 15 cycles – averages 10 cycles
Pipeline averages ____ cycles per element

Stalling a rigid pipeline
A stall in any stage halts all stages upstream of the stall point instantly (on the next clock)

What if we stopped all stages, not just upstream stages?
How does the delay of this structure scale with the number of stages?
Double Buffer
Add an extra buffer to each stage that is filled during the first cycle of a stall.

Input goes to stage when ready, or to buffer when stalled.

What is the logic equation for “next_full”? “next_buf”? “mux_sel”?
Double Buffer Timing

Double Buffer Alternate Timing
(how do you make this happen?)
Elastic Pipelines
A FIFO between stages decouples timing
Allows stages to operate at their ‘average’ speed

Resource Sharing
Suppose two pipeline stages need to access the same memory

How would you set the priority on the arbiter?
Pipeline overview

- Divide large problem into stages assembly-line style
- Divide evenly or load imbalance will occur
  - Fix by splitting or copying bottleneck stage
- Rigid pipelines have no extra storage between stages
  - A stall on any stage halts all upstream stages
  - Hard to stop 100 stages at once
    - Make this scalable with double-buffering
- Variable load results in stalls and idle cycles on a rigid pipeline
  - Make pipeline elastic by adding FIFOs between key stages