Review: Single cycle datapath

* 5 steps to design a processor
  1. Analyze instruction set ⇒ datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that affects the register transfer.
  5. Assemble the control logic

*Control is the hard part
*MIPS makes that easier
  • Instructions same size
  • Source registers always in same place
  • Immediates same size, location
  • Operations always on registers/immediates

Processor Performance

- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  1. 1 cycle per instruction
  2. Is the most demanding instruction.
  3. Assume approximate delays for major pieces of the datapath:
     - Instr. Mem, ALU, Data Mem : 2ns each, refcycle 1ns
     - Instruction execution requires: 2 + 1 + 2 + 1 = 8ns
  4. => 125 MHz
- What can we do to improve clock rate?
- Will this improve performance as well?
- We want increases in clock rate to result in programs executing quicker.

Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads

Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

- **Latency**: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency
- **Throughput**: amount of work that can be done over a period of time

Steps in Executing MIPS

1) **IFtch**: Instruction Fetch, Increment PC
2) **Dcd**: Instruction Decode, Read Registers
3) **Exec**: Mem-ref: Calculate Address Arith-log: Perform Operation
4) **Mem**: Load: Read Data from Memory Store: Write Data to Memory
5) **WB**: Write Data Back to Register

Pipelined Execution Representation

```
Time
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
```

- Every instruction must take same number of steps, also called pipeline “stages”, so some will go idle sometimes
Review: Datapath for MIPS

- Use datapath figure to represent pipeline

Graphical Pipeline Representation

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode/ Register Read</th>
<th>Execute</th>
<th>Memory</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Dc</td>
<td>Ex</td>
<td>M</td>
<td>WB</td>
</tr>
</tbody>
</table>

Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instr rate

- Nonpipelined Execution:
  - lw : IF + Read Reg + ALU + Memory + Write Reg = 2 + 1 + 2 + 2 + 1 = 8 ns
  - add: IF + Read Reg + ALU + Write Reg = 2 + 1 + 2 + 1 = 6 ns (recall 8ns for single-cycle processor)

- Pipelined Execution:
  - Max(IF,Read Reg,ALU,Memory,Write Reg) = 2 ns

Pipeline Hazard: Matching socks in later load

A depends on D; stall since folder tied up

Administrivia

- We know the readers are behind, a fire has been lit, and they say they’re on it
- No lecture next Friday
  - Those who have lab on Friday should come to any lab in Thursday and do it
  - Worst case, you can do it at home and get it checked off next week

Problems for Pipelining CPUs

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support some combination of instructions (single person to fold and put clothes away)
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- These might result in pipeline stalls or “bubbles” in the pipeline.
**Structural Hazard #1: Single Memory (1/2)**

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Order</th>
<th>Load</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TS Reg DS</td>
<td>TS Reg DS</td>
<td>TS Reg DS</td>
<td>TS Reg DS</td>
</tr>
</tbody>
</table>

Read same memory twice in same clock cycle

**Solution:**
- Infeasible and inefficient to create second memory
- (We’ll learn about this more next week)
- So simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
- Have both an L1 Instruction Cache and an L1 Data Cache
- Need more complex hardware to control when both caches miss

**Structural Hazard #2: Registers (1/2)**

Can we read and write to registers simultaneously?

**Structural Hazard #2: Registers (2/2)**

- Two different solutions have been used:
  1) RegFile access is VERY fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

**Peer Instruction**

A. Thanks to pipelining, I have reduced the time it took me to wash my shirt.
B. Longer pipelines are always a win (since less work per stage & a faster clock).
C. We can rely on compilers to help us avoid data hazards by reordering instrs.

**Things to Remember**

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute far more quickly.

- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.