Recall: Performance Evaluation

- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI, for type</th>
<th>Frequency</th>
<th>CPI, x freq,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Average CPI: 4.1

Can we get CPI < 4.1?

- Seems to be lots of “idle” hardware
  - Why not overlap instructions???

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
  - Processor
  - Control
  - Memory
  - Input
  - Output

- Next Topics:
  - Pipelining by Analogy
  - Pipeline hazards
Pipelining is Natural!

- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences
The Five Stages of Load

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Register Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file

Note: These 5 stages were there all along

```
IR <= MEM[PC]
PC <= PC + 4
```

**Pipelining**

- **Improve performance by increasing throughput**

```
Ideal speedup is number of stages in the pipeline.
Do we achieve this?
```

**Basic Idea**

```
* What do we need to add to split the datapath into stages?*
```
Graphically Representing Pipelines

° Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Conventional Pipelined Execution Representation

Why Pipeline?

° Suppose we execute 100 instructions
° Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
° Multicycle Machine
  - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
° Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
Why pipeline (cont.)?

Can pipelining get us into trouble?

- **Yes: Pipeline Hazards**
  - **Structural Hazards**: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - **Control Hazards**: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in branch instructions
  - **Data Hazards**: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - Instruction depends on result of prior instruction still in the pipeline

- Can always resolve hazards by waiting
  - Pipeline control must detect the hazard
  - Take action (or delay action) to resolve hazards

---

Structural Hazards limit performance

- **Example**: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - Average CPI $> 1.3$
  - Otherwise resource is more than 100% utilized
Control Hazard Solution #1: Stall

- **Stall**: wait until decision is clear
- **Impact**: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
- **Move decision to end of decode**
  - save 1 cycle per branch

Control Hazard Solution #2: Predict

- **Predict**: guess one direction then back up if wrong
- **Impact**: 0 lost cycles per branch instruction if right, 1 if wrong (right ~ 50% of time)
  - Need to “Squash” and restart following instruction if wrong
  - Produce CPI on branch of (1 * .5 + 2 * .5) = 1.5
  - Total CPI might then be: 1.5 * .2 + 1 * .8 = 1.1 (20% branch)
- **More dynamic scheme**: history of 1 branch (~ 90%)

Control Hazard Solution #3: Delayed Branch

- **Delayed Branch**: Redefine branch behavior (takes place after next instruction)
- **Impact**: 0 clock cycles per branch instruction if can find instruction to put in “slot” (~ 50% of time)
- **As launch more instruction per clock cycle, less useful**

Data Hazard on r1

- `add r1, r2, r3`
- `sub r4, r1, r3`
- `and r6, r1, r7`
- `or r8, r1, r9`
- `xor r10, r1, r11`
Data Hazard on r1:

• Dependencies backwards in time are hazards

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

Data Hazard Solution:

• “Forward” result from one stage to another

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

Forwarding (or Bypassing): What about Loads?

• Dependencies backwards in time are hazards

```
lw r1, 0(r2)
sub r4, r1, r3
```

• Can’t solve with forwarding:
  • Must delay/stall instruction dependent on loads

Forwarding (or Bypassing): What about Loads

```
lw r1, 0(r2)
```

• Can’t solve with forwarding:
  • Must delay/stall instruction dependent on loads
Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

Summary: Pipelining

- Reduce CPI by overlapping many instructions
  - Average throughput of approximately 1 CPI with fast clock
- Utilize capabilities of the Datapath
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction