Pipelining the MIPS Datapath

This week & next week - Pipelining

Bring handout back
Today’s lecture

- Pipeline implementation
  - Single-cycle Datapath
  - Pipelining performance
  - Pipelined datapath
  - Example
We have built a **single-cycle implementation** of a subset of the MIPS-based instruction set

- We have assumed that instructions execute in the same amount of time; this determines the clock cycle time.
- We have implemented the datapath and the control unit.
Refresher: The Full Single-cycle Datapath
We will use a simplified implementation of MIPS to create a pipelined version

Arithmetic:    add    sub    and    or    slt

Data           lw      sw

Transfer:      

Control:       beq
lw $t0, -4($sp)
beq \$at, \$0, offset

A) ALUSrc=0  
B) ALUSrc=1
add $1, $2, $3

A) ALUSrc=0
   RegDst=0

B) ALUSrc=0
   RegDst=1

C) ALUSrc=1
   RegDst=0

D) ALUSrc=1
   RegDst=1
Worst-case delay from register-read to register-write determines clock speed

What is the worst-case delay for an instruction?

a) 6 ns  
b) 7 ns  
c) 8 ns  
d) 10 ns  
e) 11 ns
Break datapath into 5 stages

Clock cycle:

<table>
<thead>
<tr>
<th>lw $t0, 4($sp)</th>
<th>lw $t1, 8($sp)</th>
<th>lw $t2, 12($sp)</th>
<th>lw $t3, 16($sp)</th>
<th>lw $t4, 20($sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>
Ideal pipeline performance is **time to fill the pipeline + one cycle per instruction**

- How long for N instructions on pipelined architecture?
  \[(4 + N) \cdot 2\text{ns} = 8 + 2N \text{ ns}\]

- How long for N instructions on single-cycle (8ns clock period)?
  \[N \cdot 8\text{ns} = 8N \text{ ns}\]

- How much faster is pipelining for N=1000?
  \[
  \frac{8 + 2N}{8N} \approx \frac{2000}{8000} \approx \frac{1}{4}
  \]
Pipelining improves throughput at the cost of increased latency.
Some instructions do not require all five stages, can we skip stages?

- Example: R-type instructions only require 4 stages: IF, ID, EX, and WB
  - We don’t need the MEM stage
- What happens if we try to pipeline loads with R-type instructions?
Trying to use the single stage for multiple instructions creates a structural hazard

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File’s Write Port during its 5th stage
  - R-type uses Register File’s Write Port during its 4th stage

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw  $t0, 4($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or   $s0, $s1, $s2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw  $t1, 8($sp)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Insert NOP stages to avoid structural hazards

- All instructions take 5 cycles with the same stages in the same order
  - Some stages will do nothing for some instructions

```
<table>
<thead>
<tr>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>add $sp, $sp, -4</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
</tr>
</tbody>
</table>
```

- Stores and Branches have NOP stages, too...

```
| store     | IF | ID | EX | MEM | NOP |
| branch    | IF | ID | EX | NOP | NOP |
```
Single-cycle datapath rearranged to align with pipeline stages
Add pipeline registers in between stages

- There’s a lot of information to save, however. We’ll simplify our diagrams by drawing just one big **pipeline register** between each stage.
- The registers are named for the stages they connect.

  IF/ID   ID/EX   EX/MEM   MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done.
Paths from register-read to register-write are now shorter
Data values required in later stages must be propagated forward through the pipeline registers.

- Example
  - Destination register ($rd$) is determined during the first stage (IF)
  - We store into the destination register during the fifth stage (WB)
  - $rd$ must be passed through all of the pipeline stages
Note – We cannot keep values like destination register in the “instruction register”
Control signals are generated in the decode stage and are propagated across stages.
Categorize control signals by the pipeline stage that uses them

<table>
<thead>
<tr>
<th>Stage</th>
<th>Control signals needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>ALUSrc ALUOp RegDst PCSrc</td>
</tr>
<tr>
<td>MEM</td>
<td>MemRead MemWrite</td>
</tr>
<tr>
<td>WB</td>
<td>RegWrite MemToReg</td>
</tr>
</tbody>
</table>
The pipeline registers and program counter update every clock cycle, so they do not have write enable controls.