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 Transfer:  lw   sw

Control:   beq
lw $t0, -4($sp)
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

\[ \max(2, 4, 6, 8) = 8 \text{ ns} \]
How fast can we clock a pipelined datapath?

- **IF** - instruction fetch
- **ID** - instruction decode
- **EXE** - execute
- **MEM** - memory
- **WB** - write back

- **Read instruction address**
- **Instruction memory**
- **RegWrite**
- **Read register 1**
- **Read data 1**
- **Write register**
- **Read data 2**
- **RegDst**
- **Write data**
- **ALU**
- **Zero Result**
- **ALUOp**
- **ALUSrc**
- **Read address**
- **Read data**
- **Write address**
- **Write data**
- **Data memory**
- **MemWrite**
- **MemRead**
- **MemToReg**

**Clock times:**
- IF: 2ns
- ID: 1ns
- EXE: 2ns
- MEM: 2ns
- WB: 1ns
Break datapath into 5 stages

<table>
<thead>
<tr>
<th>Instruction</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><strong>lw $t0, 4($sp)</strong></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><strong>lw $t1, 8($sp)</strong></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><strong>lw $t2, 12($sp)</strong></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><strong>lw $t3, 16($sp)</strong></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><strong>lw $t4, 20($sp)</strong></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>
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?

```
Clock cycle

1  2  3  4  5  6  7  8  9

| add $sp, $sp, -4 | IF | ID | EX |  | WB |
| sub $v0, $a0, $a1 | IF | ID | EX |  | WB |
| lw  $t0, 4($sp)   | IF | ID | EX | MEM | WB |
| or  $s0, $s1, $s2 | IF | ID | EX |  | WB |
| lw  $t1, 8($sp)   | IF | ID | EX | MEM | WB |
```
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.

<table>
<thead>
<tr>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9</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>
# 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

## R-type

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $sp, $sp, -4</td>
<td>1</td>
</tr>
<tr>
<td>sub $v0, $a0, $a1</td>
<td>2</td>
</tr>
<tr>
<td>lw $t0, 4($sp)</td>
<td>3</td>
</tr>
<tr>
<td>or $s0, $s1, $s2</td>
<td>4</td>
</tr>
<tr>
<td>lw $t1, 8($sp)</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td>IF</td>
</tr>
<tr>
<td>branch</td>
<td>IF</td>
</tr>
</tbody>
</table>
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

Group control signals based on the stages they are used in
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</td>
</tr>
<tr>
<td>MEM</td>
<td>MemRead</td>
</tr>
<tr>
<td>WB</td>
<td>RegWrite</td>
</tr>
</tbody>
</table>
The pipeline registers and program counter update every clock cycle, so they do not have write enable controls.
An example execution sequence

addresses in decimal

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>lw</td>
<td>$8, 4($29)</td>
</tr>
<tr>
<td>1004</td>
<td>sub</td>
<td>$2, $4, $5</td>
</tr>
<tr>
<td>1008</td>
<td>and</td>
<td>$9, $10, $11</td>
</tr>
<tr>
<td>1012</td>
<td>or</td>
<td>$16, $17, $18</td>
</tr>
<tr>
<td>1016</td>
<td>add</td>
<td>$13, $14, $0</td>
</tr>
</tbody>
</table>

- **ASSUMPTIONS**

- **CONVENTIONS**
  - X indicates values that are not important, Example: Imm16 for R-type.
  - Question marks ??? indicate values we do not know, usually resulting from instructions coming before and after the ones in our example.
Cycle 1 (filling)

IF: lw $8, 4($29):

ID: ???

EX: ???

MEM: ???

WB: ???

Flowchart details:
- IF/ID: lw $8, 4($29)
- ID/EX: ???
- EX/MEM: ???
- MEM/WB: ???
Cycle 2

IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???

Read instruction address [31-0]
Instruction memory

Read register 1
Read data 1
Write register
Write data

Shift left 2

ALU Zero

MemToReg (?)
MemWrite (?)
MemRead (?)

RegWrite (?)
Imm extend
Sign extend

RegDst (?)
ALUOp (?)
ALUSrc (?)

MemWrite (?)
MemRead (?)
MemToReg (?)

**Cycle 3**

IF: and \$9, \$10, \$11

ID: sub \$2, \$4, \$5

EX: lw \$8, 4(\$29)

MEM: ???

WB: ???
Cycle 4

IF: or $16, $17, $18

ID: and $9, $10, $11

EX: sub $2, $4, $5

MEM: lw $8, 4($29)

WB: ???
Cycle 5 (full)

IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)
Cycle 6 (emptying)

IF: ???
ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

PCSrc

Read address [31-0]
Instruction memory

Instruction

Add

Add

Shift left 2

ALUSrc

ALUOp (or)

RegWrite

RegWrite

Write register

Write data

Write register

Write data

MemRead (0)

MemWrite (0)

ALU Zero

Result

Add

Add

RegDst

RegDst

Sign extend

MemRead (0)

MemWrite (0)

Data memory

Cycle 6 (emptying)

MemWrite (0)
Cycle 7

IF: ???
ID: ???
EX: add $13, $14, $0
MEM: or $16, $17, $18
WB: and $9, $10, $11
Cycle 8

IF: ???

ID: ???

EX: ???

MEM: add $13, $14, $0

WB: or $16, $17, $18
Cycle 9

IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0

Read address [31-0]

Instruction memory

Add

PCSrc

Shift left 2

RegWrite (1)

Read register 1
Read register 2
Write register
Write data

RegDst (?)

Sign extend

MemToReg (0)

MemWrite (?)

MemRead (?)

ALU Zero

ALUSrc (?)

ALUOp (???)

ALU

Write data

Data memory

Address

Result

EX/ID

ID/EX

EX/MEM

MEM/WB

WB

P C
Things to notice from the last nine slides

- Instruction executions overlap
- Each functional unit is used by a *different* instruction in each cycle.
- In clock cycle 5, all of the hardware units are used (the pipeline is full). This is the ideal situation, and what makes pipelined processors so fast
- Similar example in the book available at the end of Section 6.3.
MIPs ISA makes pipelining “easy”

- Instruction formats are the same length and uniform
- Addressing modes are simple
- Each instruction takes only one cycle
Note how everything goes left to right, except …

Next time: We will discuss Data Hazards
Cycle 6 (emptying)

IF: ???
ID: add $13, $14, $0
EX: or $16, $17, $18
MEM: and $9, $10, $11
WB: sub $2, $4, $5

Instruction
memory

Instruction
address [31-0]

RegWrite (1)

Read register 1
Read register 2
Write register
Write data

ALU Zero
Result

ALUSrc (0)

Add

EX/MEM

MEM/WB

MemWrite (0)

MemRead (0)

Sign extend

RegDst (1)

MemToReg

October 30, 2017
Pipelined datapath and control