Designing a Pipelined CPU
Quiz next Wednesday (2/20)

• No class on Monday 2/18
• Quiz will be full hour (less time pressure?)
• How do you set control lines?
• What would happen if some line was stuck on (or off)
• How would you add such-and-such instruction
• Why is such-and-such needed
• Where do you insert no-ops or stalls to execute ...
Review -- Single Cycle CPU

![Diagram of a single cycle CPU](image-url)
Review -- Multiple Cycle CPU
Review -- Instruction Latencies

Single-Cycle CPU

Load: Ifetch, Reg/Dec, Exec, Mem, Wr

Multiple Cycle CPU

Load: Ifetch, Reg/Dec, Exec, Mem, Wr

Add: Ifetch, Reg/Dec, Exec, Wr
A Pipelined Datapath

IF: Instruction fetch
ID: Instruction decode and register fetch
EX: Execution and effective address calculation
MEM: Memory access
WB: Write back
Pipelined Datapath

**Warning – “write register” line is incorrect in this figure!**
Execution in a Pipelined Datapath

CSE 141

steady state

Dean Tullsen
Instruction Latencies and Throughput

**Single-Cycle CPU**

- Cycle 1
  - Load: IF, Dec, EX, Mem, WB

**Multiple Cycle CPU**

- Cycle 1 to Cycle 5
  - Load: IF, Dec, EX, Mem, WB

**Pipelined CPU**

- Cycle 1 to Cycle 8
  - Load: IF, Dec, EX, Mem, WB
Pipelining Advantages

• Higher maximum throughput
• Higher utilization of CPU resources

• But, more hardware needed, perhaps complex control
Mixed Instructions in the Pipeline

lw
IM → Reg → ALU → DM → Reg

add
IM → Reg → ALU → Reg
Pipeline Principles

• All instructions that share a pipeline must have the same stages in the same order.
  – therefore, *add* does nothing during Mem stage
  – *sw* does nothing during WB stage

• All intermediate values must be latched each cycle.

• There is no functional block reuse
  – example: we need 2 adders and ALU (like in single-cycle)
Pipelined Datapath

CSE 141
The Pipeline in Execution

**add $10, $1, $2**

- **Instruction Decode/Register Fetch**
- **Execute/Address Calculation**
- **Memory Access**
- **Write Back**

---

**IF/ID**
- Address
- Instruction memory

**ID/EX**
- Add
- Read register 1
- Read register 2
- Write register
- Write data

**EX/MEM**
- Add
- Add result
- Shift left 2
- Zero ALU result

**MEM/WB**
- Address memory
- Read data
- Write data

---

CSE 141

Dean Tullsen
The Pipeline in Execution

lw $12, 1000($4)  add $10, $1, $2

Execute/
Address Calculation

Memory Access  Write Back

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

0 1
Mux

Add

4

PC

Instruction memory

Instruction

Read

Read

Read

Read

Write

Write

Write

Write

Register 1

Register 2

Data 1

Data 2

Data

Zero

ALU

Shift left 2

Add

Add result

0

Mux

1

Mux

0

1

Mux

16

Sign extend

32

Address

Data

memory

Write

data

Write

data

Write

data

Write

data

CSE 141  Dean Tullsen
The Pipeline in Execution

\[ \text{sub $15, $4, $1} \quad \text{lw $12, 1000($4)} \quad \text{add $10, $1, $2} \]

Memory Access \quad Write Back

![Pipeline Diagram]

CSE 141

Dean Tullsen
The Pipeline in Execution

Instruction Fetch

sub $15, $4, $1
lw $12, 1000($4)
add $10, $1, $2

Write Back

CSE 141

Dean Tullsen
The Pipeline in Execution

Instruction Fetch

Instruction Decode/ Register Fetch

\[
\text{sub } \$15, \$4, \$1 \quad \text{lw } \$12, 1000(\$4) \quad \text{add } \$10, \$1, \$2
\]

IF/ID

ID/EX

EX/MEM

MEM/WB

Mux

Add

Shift left 2

Zero

ALU result

Write data

Write register

Read register 1

Read register 2

Read data 1

Read data 2

PC

Address

Instruction memory

Instruction

Register Fetch

Write

Read

Zero extend

16

32

Data memory

Address

Read data

Write data

1 Mux

0 Mux

0 Mux

1 Mux
The Pipeline in Execution

sub $15, $4, $1  lw $12, 1000($4)

CSE 141  Dean Tullsen
(This figure has correct Write register bus)
Pipelined Control

• FSM isn’t really appropriate
• Combinational Logic (like single-cycle design)!
  – signals generated once in ID stage
  – follow instruction through the pipeline
  – get used in whatever stage they are needed
Pipelined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>RegWrite</th>
<th>MemtoReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
Is it really that easy?

- Suppose initially, register i holds the number 2i
- What happens when...
  - add $3, $10, $11 - this should add 20 + 22, putting result 42 into r3
  - lw $8, 50($3) - this should load r8 from memory location 42+50 = 92
  - sub $11, $8, $7 - this should subtract 14 from that just-loaded value
The Pipeline in Execution

lw $8, 50($3)  add $3, $10, $11  Execute/
Address Calculation

Memory Access  Write Back

Instruction memory

PC

Add

Address

0 Mix 1

IF/ID

ID/EX

EX/MEM

MEM/WB

alu

Add

Add result

Shift left 2

ALU

Zero

al

Add

data

Write

register 1

Read register 1

Read data 1

Write data

Read register 2

Read data 2

Registers

Write register

Instruction

Address

16

Sign extend

32

Data memory

Write data

Read data

Address

Write data

1 Mix 0

CSE 141  Dean Tullsen
The Pipeline in Execution

sub $11, $8, $7
lw $8, 50($3)
add $3, $10, $11

Memory Access
Write Back

Ooops! This should have been “42”!
But register 3 didn’t get updated yet.
The Pipeline in Execution

Add $10, $1, $2  
Sub $11, $8, $7  
LW $8, 50($3)  
Add $3, $10, $11  
Write Back

And this should be value from memory (which hasn't even been loaded yet).

Recall: this should have been “92”
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```