CSE 141: Introduction to Computer Architecture

Pipelines
First things first: **Pipelines are the coolest.**

- Seriously, this idea is everywhere
THE key idea of pipelining

• Throughput >>> latency
• Computers are very useful because they do a lot of things well
  – It is much less important how well any one thing is done

• Which is faster?
  – A machine with average CPI of 2.0 running at 48 MHz
  – A machine with average CPI of 10.0 running at 4 GHz
Review -- Single Cycle CPU
(not quite) Review -- Multiple Cycle CPU
Review -- Instruction Latencies

Single-Cycle CPU

- Ifetch
- Reg/Dec
- Exec
- Mem
- Wr

Multiple Cycle CPU

- Ifetch
- Reg/Dec
- Exec
- Mem
- Wr
- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9

Load
Add
Instruction Latencies and Throughput

Single-Cycle CPU

Multiple Cycle CPU

Pipelined CPU
Instruction Latencies and Throughput

**Single-Cycle CPU**

Cycle 1

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

**Multiple Cycle CPU**

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

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

**Pipelined CPU**

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8

Load: Ifetch, Reg/Dec, Exec, Mem, Wr
Instruction Latencies and Throughput

**Single-Cycle CPU**

- **Cycle 1**
- **Cycle 2**
- **Cycle 3**
- **Cycle 4**
- **Cycle 5**

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

**Multiple Cycle CPU**

- **Cycle 1**
- **Cycle 2**
- **Cycle 3**
- **Cycle 4**
- **Cycle 5**

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

**Pipelined CPU**

- **Cycle 1**
- **Cycle 2**
- **Cycle 3**
- **Cycle 4**
- **Cycle 5**
- **Cycle 6**
- **Cycle 7**
- **Cycle 8**

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

Load: Ifetch, Reg/Dec, Exec, Mem, Wr
Instruction Latencies and Throughput

**Single-Cycle CPU**

- **Cycle 1:** Ifetch
- **Cycle 2:** Reg/Dec
- **Cycle 3:** Exec
- **Cycle 4:** Mem
- **Cycle 5:** Wr

**Multiple Cycle CPU**

- **Cycle 1:** Ifetch
- **Cycle 2:** Reg/Dec
- **Cycle 3:** Exec
- **Cycle 4:** Mem
- **Cycle 5:** Wr

**Pipelined CPU**

- **Cycle 1:** Ifetch
- **Cycle 2:** Reg/Dec
- **Cycle 3:** Exec
- **Cycle 4:** Mem
- **Cycle 5:** Wr
- **Cycle 6:** Ifetch
- **Cycle 7:** Reg/Dec
- **Cycle 8:** Exec
Pipelining Advantages

- Higher *maximum* throughput
- Higher *utilization* of CPU resources

- But, more complicated *datapath*, more complex control(?)
Poll Q: What affects throughput? **Peak throughput** depends on...

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Multi-Cycle</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Longest Instruction</td>
<td>Cycle Time</td>
<td>Average Instruction</td>
</tr>
<tr>
<td>B</td>
<td>Longest Instruction</td>
<td>Cycle Time</td>
<td>Longest Instruction</td>
</tr>
<tr>
<td>C</td>
<td>Longest Instruction</td>
<td>Average Instruction</td>
<td>Cycle Time</td>
</tr>
<tr>
<td>D</td>
<td>Average Instruction</td>
<td>Longest Instruction</td>
<td>Cycle Time</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Poll Q: What affects throughput?
Peak throughput depends on...

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Multi-Cycle</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Longest Instruction</td>
<td>Average Instruction</td>
<td>Cycle Time</td>
</tr>
</tbody>
</table>

Throughput is useful work over time – one measure: insts / sec

\[ ET = \text{Inst} \times CPI \times CT \]

Single Cycle: \[ ET = \text{Inst} \times 1 \times \text{BIG} \]
Multi Cycle: \[ ET = \text{Inst} \times [3 \ldots 5] \times CT \]
Pipeline: \[ ET = \text{Inst} \times 1 \times CT \]
Pipelining in Modern CPUs

- CPU Datapath
- Arithmetic Units
- System Buses
- Software (at multiple levels)
- etc...
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 (roughly)

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back

IF
ID
EX
MEM
WB

Instruction Fetch
Instruction Decode
Operand Fetch
Execute
Result Store
Next Instruction
Execution in a Pipelined Datapath
Execution in a Pipelined Datapath

- **CC1**: IF (Instruction Fetch)
- **CC2**: ID (Instruction Decode)
- **CC3**: EX (Execution)
- **CC4**: MEM (Memory Access)
- **CC5**: WB (Write Back)
- **CC6**: Reg
- **CC7**: Reg
- **CC8**: Reg
- **CC9**: Reg

**lw** (Load Word)

- IF: IM (Instruction Memory)
- ID: Reg
- EX: ALU
- MEM: DM (Data Memory)
- WB: Reg

**steady state**
Mixed Instructions in the Pipeline

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mixed Instructions in the Pipeline

lw

add
Mixed Instructions in the Pipeline

lw

add
Mixed Instructions in the Pipeline

lw

add
This is called a **structural hazard** – too many instructions want to use the same resource.

In our pipeline, we can make this hazard disappear (next slide).

In more complex pipelines, structural hazards are again possible.
Pipeline Principles

• All instructions that share a pipeline should 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.
Pipeline stages

• What is the performance implication of making every instruction go through all 5 stages? (e.g., instead of 4 for add, 3 for beq, etc.)

<table>
<thead>
<tr>
<th>(Choose BEST answer)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Decreases peak throughput by 20%</td>
</tr>
<tr>
<td>B</td>
<td>Increases program latency by 20%</td>
</tr>
<tr>
<td>C</td>
<td>No significant impact on peak throughput or program latency</td>
</tr>
<tr>
<td>D</td>
<td>Depends on how many R-type instructions, beq, etc.</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Pipelined Datapath

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

IF | ID | EX | MEM | WB

Instruction memory | Registers | ALU result | Write memory | Write data

Add | Shift left 2 | ALU | Data | Write data

PC | 0 Mux | 16 | 32 | 16

Address | Instruction | Read register 1 | Read register 2 | Read data 1 | Read data 2
Pipelined Datapath

Instruction Fetch

Instruction Decode/Register Fetch

Execute/Address Calculation

Memory Access

Write Back

IF

ID

EX

MEM

WB

registers!
Poll Q: How many D flip flops are in this pipeline?

A 4
B 12
C 128
D 352
E Other
The Pipeline in Execution

add $10, $1, $2

Instruction Decode/Register Fetch

Execute/Address Calculation

Memory Access

Write Back

Diagram showing the pipeline stages of instruction execution with labels for each stage and a flowchart representing the process.
The Pipeline in Execution

lw $12, 1000($4)

\textbf{lw} $12, 1000($4)

\textbf{Add} $10, $1, $2

\textbf{Add} $10, $1, $2

\textbf{Execute/Address Calculation}

Memory Access

Write Back
The Pipeline in Execution

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

Memory Access  Write Back
The Pipeline in Execution

**Instruction Fetch**

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

**Write Back**
The Pipeline in Execution

Instruction Fetch  Instruction Decode/Register Fetch

\( \text{sub } \$15, \$4, \$1 \quad \text{lw } \$12, 1000(\$4) \quad \text{add } \$10, \$1, \$2 \)
The Pipeline in Execution

Instruction Fetch

Instruction Decode/ Register Fetch

Execute/ Address Calculation

sub $15, $4, $1  lw $12, 1000($4)
The Pipeline in Execution

Instruction Fetch  Instruction Decode/ Register Fetch  Execute/ Address Calculation  Memory Access

sub $15, $4, $1
### Review: When executing only R-type instructions...

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Multi-Cycle</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># cycles to exec 1 inst</td>
<td>CPI for 1M insts</td>
<td># cycles to exec 1 inst</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table Values:
- **A**: 1, 1, 4, 4, 5, 5
- **B**: 4, 4, 5, 1, 1, 5
- **C**: 4, 4, 5, 5, 4, 1
- **D**: 1, 1, 4, 4, 5, 1
- **E**: None of the above

### Notes:
- **CPI**: Cycle per Instruction
- **# cycles to exec 1 inst**: Number of cycles required to execute one instruction
The Pipeline, now with controls....
I told you multicycle control was messy. We would expect pipelined control to be messier.
Pipelined Control

• I told you multicycle control was messy. We would expect pipelined control to be messier.
  – Why?
Pipelined Control

- I told you multicycle control was messy. We would expect pipelined control to be messier.
  - Why?
- But it turns out we can do it with just...
Pipelined Control

- I told you multicycle control was messy. We would expect pipelined control to be messier.
  - Why?
- But it turns out we can do it with just...
- **Combinational logic!**
  - Signals generated **once**
  - Follow instruction through the pipeline
Recall: Control signals in the single-cycle machine
Pipelined Control
Pipelined Control

So, really it is combinational logic and some registers to propagate the signals to the right stage.
The Pipeline with Control Logic
### Pipelined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Stage Control Lines</th>
<th>Memory Stage Control Lines</th>
<th>Write Back Stage Control Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RegDst</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Pipeplined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Stage Control Lines</th>
<th>Memory Stage Control Lines</th>
<th>Write Back Stage Control Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RegDst</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let’s just do one.
The Pipeline with Control Logic

You Choose:
A. R-format
B. lw
C. sw
D. beq
Is it really that easy?

- What happens when...
  - `add $3, $10, $11`
  - `lw $8, 1000($3)`
  - `sub $11, $8, $7`
The Pipeline in Execution

lw $8, 1000($3)  add $3, $10, $11

Execute/
Address Calculation

Memory
Access

Write Back

```
lw $8, 1000($3)
add $3, $10, $11
```
The Pipeline in Execution

```
sub $11, $8, $7
lw $8, 1000($3)
add $3, $10, $11
```
The Pipeline in Execution

\begin{align*}
\text{add } & $10, $1, $2 & \text{sub } & $11, $8, $7 & \text{lw } & $8, 1000($3) & \text{add } & $3, $10, $11 \quad \text{Write Back}
\end{align*}
Data Hazards

When a result is needed in the pipeline before it is available, a data hazard occurs. What can we do?

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

- Data Hazards are caused by data dependences
- Not all data dependences result in data hazards
- A data hazard results when there is a data dependence between two instructions that appear too close together in the pipeline

- We will define a data hazard as any data dependence that requires either the software or hardware to take special action to get correct
Dealing With Data Hazards – What can we do...

• ...in Software?
  –

• ...in Hardware?
  –

Data Hazards are caused by *instruction dependences*. For example, the add is data-dependent on the subtract:

```
subi  $5, $4, #45
add   $8, $5, $2
```
Dealing with Data Hazards in Software

sub $2, $1, $3

and $12, $2, $5
Dealing with Data Hazards in Software

- sub $2, $1, $3
- nop
- nop
- nop
- and $12, $2, $5
How Many No-ops?

sub $2, $1, $3
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
Are No-ops Really Necessary?

\[ \text{sub } 2, 1, 3 \]
\[ \text{and } 4, 2, 5 \]
\[ \text{or } 8, 3, 6 \]
\[ \text{add } 9, 2, 8 \]
\[ \text{slt } 1, 6, 7 \]
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls

- \text{sub} \, 2, \, 1, \, 3
- \text{and} \, 12, \, 2, \, 5
- \text{or} \, 13, \, 6, \, 2
- \text{add} \, 14, \, 2, \, 2
- \text{sw} \, 15, \, 100(2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
### Dealing with Data Hazards in Hardware
#### Part II - Pipeline Stalls (alt. View)

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

- sub $2, \$, 1, \$, 3
- and $12, \$, 2, \$, 5
- or $13, \$, 6, \$, 2
- add $14, \$, 2, \$, 2
- sw $15, 100(\$, 2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Dealing with Data Hazards in Hardware
Part II-Pipeline Stalls (alt. View)

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Poll Q: Try it yourself

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

How many bubbles?

A 5
B 6
C 7
D 8
E None of the above
Working this example...

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>$2, $1, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$12, $3, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>$13, $6, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>$14, $12, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$14, 100($2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Poll Q: How to actually implement this in hardware?

Once you detect the hazard in ID – what must you do to insert the nop and “stall”?

1. Flush all instructions in the pipeline (set control signals to 0).
2. Set all control signals going to ID/EX register to zero.
3. Set PCWrite to zero.
4. Set IF/ID register write to zero.

<table>
<thead>
<tr>
<th>Selection</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>B</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>C</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Pipeline Stalls

- To ensure proper pipeline execution in light of register dependences, we must:
  - detect the hazard
  - stall the pipeline
Knowing When to Stall

- 6 types of data hazards
  - two reg reads * 3 reg writes
Knowing When to Stall

- 6 types of data hazards
  - two reg reads * 3 reg writes
The Pipeline

• What comparisons tell us when to stall?
Stalling the Pipeline

Once we detect a hazard, then we have to be able to stall the pipeline (insert a *bubble*).

Stalling the pipeline is accomplished by

- (1) preventing the IF and ID stages from making progress
  - the ID stage because it cannot proceed until the dependent instruction completes
  - the IF stage because we do not want to lose any instructions.
- (2) essentially, inserting “*nops*” in hardware
Stalling the Pipeline

- Preventing the IF and ID stages from proceeding
  - don’t write the PC (PCWrite = 0)
  - don’t rewrite IF/ID register (IF/IDWrite = 0)
- Inserting “nops”
  - set all control signals propagating to EX/MEM/WB to zero
Can we do better? How else might we deal with (some?) data hazards?
Reducing Data Hazards Through Forwarding

add $2, $3, $4

add $5, $3, $2
Reducing Data Hazards Through Forwarding
Reducing Data Hazards Through Forwarding

**EX Hazard:** (similar for the MEM stage)

if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
then ForwardA = 10

if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
then ForwardB = 10
Data Forwarding

- The Previous Data Path handles two types of data hazards
  - EX hazard
  - MEM hazard
- The register file handles the third (WB hazard)
  - if the register file is asked to read and write the same register in the same cycle, the register file has internal forwarding logic that allows the write data to be forwarded to the output
  - This is still forwarding (even if you don’t “see” the lines b/c internal)!
Eliminating Data Hazards via Forwarding

sub $2, $1, $3
and $6, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Forwarding in Action

\[
\begin{align*}
\text{add} & \quad \text{sub} & \quad \text{add} \\
$1, $12, $3 & \quad $12, $3, $4 & \quad $3, $10, $11
\end{align*}
\]

Memory Access  Write Back
Forwarding in Action

Instruction Fetch  
\[\text{add } 1, 12, 3\]  \[\text{sub } 12, 3, 4\]  \[\text{add } 3, 10, 11\]  Write Back
Forwarding in Action

Instruction Fetch  Instruction Decode

- add $1, $12, $3
- sub $12, $3, $4
- add $3, $10, $11
Eliminating Every Data Hazard via Forwarding?

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
### Eliminating Data Hazards via Forwarding and stalling

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
</table>

```assembly
lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
```
Eliminating Data Hazards via Forwarding and stalling

1w $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Eliminating Data Hazards via Forwarding and stalling

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Eliminating Data Hazards via Forwarding and stalling

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Eliminating Data Hazards via Forwarding and stalling

Just to be clear, let’s review what we mean by “bubble” particularly in the context of this pipeline!
Eliminating Data Hazards via Forwarding and stalling

lw $2, 10($1)

and $12, $2, $5

What is really happening during the bubble (for this particular pipeline)?
Eliminating Data Hazards via Forwarding and stalling

lw $2, 10($1)
and $12, $2, $5

What is really happening during the bubble (for this particular pipeline)?

- While *lw* moves to the Mem stage in CC4, the *and* instruction repeats the ID stage (important because the values the *and* reads in CC4 are the ones it will carry forward).
Eliminating Data Hazards via Forwarding and stalling

1w $2, 10($1)
and $12, $2, $5

What is really happening during the bubble (for this particular pipeline)?

- While *lw* moves to the Mem stage in CC4, the *and* instruction repeats the ID stage (important because the values the *and* reads in CC4 are the ones it will carry forward).
- There is now *no instruction* in the EX stage. So we better make sure that whatever is in the EX stage is *safe*. 
Eliminating Data Hazards via Forwarding and stalling

What is really happening during the bubble (for this particular pipeline)?

- While `lw` moves to the Mem stage in CC4, the `and` instruction repeats the ID stage (important because the values the `and` reads in CC4 are the ones it will carry forward).
- There is now no instruction in the EX stage. So we better make sure that whatever is in the EX stage is safe.
  - Safe = no state changes (PC, reg, memory), now or as it moves through the pipeline.
Poll Q: Stalls & Forwards

- How many stalls occur and how many values require hardware forwarding support to avoid stalling for our MIPS 5-stage pipeline?

```
add $3, $2, $1
lw $4, 100($3)
and $6, $4, $3
sub $7, $6, $2
add $9, $3, $6
```

<table>
<thead>
<tr>
<th></th>
<th>Stalls</th>
<th>Forwarded values</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
</tr>
</tbody>
</table>
Try this one...

• Show bubbles and forwarding for this code

  add $3, $2, $1
  lw $4, 100($3)
  and $6, $4, $3
  sub $7, $6, $2
  add $9, $3, $6
Another one...

- Show bubbles and forwarding for this code

\[
\begin{align*}
\text{l}w & \quad $9, 100($6) & \text{IF} & \quad \text{ID} & \quad \text{EX} & \quad \text{M} & \quad \text{WB} \\
\text{add}i & \quad $6, $9, #26 \\
\text{sub} & \quad $7, $6, $9 \\
\text{add} & \quad $6, $3, $6 \\
\text{add} & \quad $3, $2, $6
\end{align*}
\]
Poll Q: How many stalls?

- Suppose EX is the longest (in time) pipeline stage
- To reduce CT, we split it in half. Given the following (new) pipeline:
  \[\text{IF ID EX1 EX2 M WB}\]
  Assume the input data must be available at the start of EX1 and the output is available after EX2
- How many hardware stalls would be required in the following code (assuming hardware forwarding wherever possible)?

\[
\begin{align*}
\text{add r1, r2, r3} \\
\text{add r4, r1, r3}
\end{align*}
\]
Poll Q: How many stalls?

- Suppose EX is the longest (in time) pipeline stage
- To reduce CT, we split it in half. Given the following (new) pipeline:
  
  IF ID EX1 EX2 M WB

  Assume the input data must be available at the start of EX1 and the output is available after EX2
- **How many hardware stalls** would be required in the following code (assuming hardware forwarding wherever possible)?

  \[
  \text{lw} \ r1, 0(r3) \\
  \text{add} \ r2, r1, r3
  \]
Datapath with Hazard-Detection

if (ID/EX.MemRead and 
((ID/EX.RegisterRt = IF/ID.RegisterRs) or 
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
then stall the pipeline
Hazard Detection

and $4, $2, $5 \text{ lw } $2, 20($1)$
Hazard Detection

and $4, $2, $5 \text{  nop (bubble)  lw }$2, 20($1)$
What other hazards might we have to watch out for?

- Data hazards are when the result of one computation is used in a later computation
- Is there other re-use?
Control Dependence

• Just as an instruction will be dependent on other instructions to provide its operands (data dependence), it will also be dependent on other instructions to determine whether it gets executed or not (control dependence, aka, branch dependence).

• Control dependences are particularly critical with conditional branches.

```assembly
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...
somewhere: or $10, $5, $2
add $12, $11, $9
...```

CC BY-NC-ND Pat Pannuto – Many slides adapted from Leo Porter, Dean Tullsen, and the UCSD faculty
Branch Hazards

- Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline)
  - (sound familiar?)
Stalling the pipeline

Given our current pipeline, let’s assume we stall until we know the branch outcome (i.e., until the PC is known to be correct). How many cycles will we lose per branch?

<table>
<thead>
<tr>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>
Branch Hazards

beq $2, $1, here

add ...

sub ...

lw ...

here: lw ...
Dealing With Branch Hazards

• Ideas??
Dealing With Branch Hazards

• Hardware
  – stall until you know which direction
  – reduce hazard through earlier computation of branch direction
  – guess which direction
    • assume not taken (easiest)
    • more educated guess based on history
      – (requires that you know it is a branch before it is even decoded!)
Dealing With Branch Hazards

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history
      - (requires that you know it is a branch before it is even decoded!)

- Hardware/Software
  - nops
  - instructions that get executed either way (delayed branch).
beq $4, $0, there
and $12, $2, $5
or ... 
add ...
sw ...
Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.
Assume Branch \textbf{Not Taken}

- works pretty well when you’re right!

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...
```
Assume Branch Not Taken

- *same performance as stalling* when you’re wrong

```assembly
beq $4, $0, there
and $12, $2, $5
or ...
add ...
there: sub $12, $4, $2
```
Assume Branch Not Taken

- Performance depends on percentage of time you guess right
- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC)
  - sounds a lot like a bubble...
  - But notice that we need to be able to **insert** those bubbles **later** in the pipeline
Branch Hazards – What if we predict taken instead?

-beq $2, $1, here

here: lw

**Required** information to predict Taken:

1. Whether an instruction is a branch (before decode)
2. The target of the branch
3. The outcome of the branch condition

<table>
<thead>
<tr>
<th></th>
<th>Required knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2, 3</td>
</tr>
<tr>
<td>B</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>C</td>
<td>1, 2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Branch Target Buffer

*a, how to know it’s a branch before you know it’s a branch*

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target
Reducing the Branch Delay

• Can we change anything in the pipeline to make branch delay less bad?
Reducing the Branch Delay

First, let’s try to get to 2-cycle stall
Stalling for Branch Hazards

beq $4, $0, there

and $12, $2, $5

or ...

add ...

sw ...
Reducing the Branch Delay More??

Harder… but possible to get to a 1-cycle stall?
Stalling for Branch Hazards

beq $4, $0, there

and $12, $2, $5

or ...

add ...

sw ...
Quick Flashback:
Part I said, we only need “beq”, no “bgt”, “blt” in MIPS...

What if we want to condition the control flow? Branches.
do { ... ; a++ ; } while (a < 100);

- beq and bne are the only branches you need
  - beq r1, r2, addr => if (r1 == r2): goto addr
- But other operations can be combined...
  - slt $1, $2, $3 => if ($2 < $3) $1 = 1; else $1 = 0
- beq, bne, slt, and $zero, can implement all fundamental conditions
  - Always, never, !=, =, =, >, <=, >=, <, > (unsigned), <= (unsigned), ...

    if (i < j)
    w = w + 1;
    else
    w = 5;
The Pipeline with flushing for taken branches

- Notice the IF/ID flush line added.
Eliminating the Branch Stall

A cute idea, but not one used by modern cores

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!
Branch Delay Slot

beq $4, $0, there

and $12, $2, $5

there: or ...

add ...

sw ...

Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.
**Filling the branch delay slot**

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a `nop` to ensure correctness.
- Where do we find instructions to fill the branch delay slot?
  - 
  - 
  -
Filling the branch delay slot

1. add $5, $3, $7
2. add $9, $1, $3
3. sub $6, $1, $4
4. and $7, $8, $2
5. beq $6, $7, there
   
   nop /* branch delay slot */

6. add $9, $1, $4
7. sub $2, $9, $5

... there:
8. mult $2, $10, $11

...
Branch Delay Slots

• This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
• What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle??
• What about the Pentium 4, which has a 21-cycle branch penalty and executes up to 3 instructions per cycle??
Early resolution of branch + branch delay slot

- Worked well for MIPS R2000 (the 5-stage pipeline MIPS)
- Early resolution doesn’t scale well to modern architectures
  - Better to always have execute happen in execute
  - Forwarding into branch instruction?
- Branch delay slot
  - Doesn’t solve the problem in modern pipelines
  - Still in ISA, so have to make it work even though it doesn’t provide any significant advantage.
  - Violates important general principal – (unless you really only want a single generation of your product) do not expose current technology limitations to the ISA.
Okay, then...

- What do we do in modern architectures???
Branch Prediction

- Always assuming a branch is not taken is a crude form of branch prediction.
- What about loops that are taken 95% of the time?
  - we would like the option of assuming not taken for some branches, and assuming taken for others, depending on ???
Branch Prediction

• Historically, two broad classes of branch predictors:
  
• Static predictors – for branch B, always make the same prediction.
  
• Dynamic predictors – for branch B, make a new prediction every time the branch is fetched.
  
• Tradeoffs?
  
• Modern CPUs all have sophisticated dynamic branch prediction.
Dynamic Branch Prediction

- What information is available to make an intelligent prediction?
Branch Prediction: Simplest 1-bit predictor

for (i=0; i<10; i++) {
    ...
    ...
}

add $i, $i, #1
beq $i, #10, loop

PC-based Predictor Table

Multiple predictors, so that we can answer “what has this branch done lately”
Two-bit predictors give better loop prediction

This state machine also referred to as a saturating counter. It counts down (on not taken) to 00 or up (on taken) to 11, but does not wrap around.

```
for (i=0; i<10; i++) {
    ...
    ...
}
...  
add  $i, $i, #1  
beq $i, #10, loop
```
Branch History Table
first introduced by the “[2-bit] bimodal predictor”

- has limited size
- 2 bits by N (e.g. 4K)
- uses low bits of branch address to choose entry

- what about even/odd branch?

Multiple 2-bit predictors, so that we can answer “what has this branch done lately”
2-bit bimodal predictor

• For the following loop, what will be the prediction accuracy of the bimodal predictor for the conditional branch that closes the loop?

```
for (i=0; i< 2; i++)
  //two iterations per loop
  {  z = ...  }
```
2-bit bimodal misprediction rates

Is this good enough?
Can We Do Better?

- Can we get more information dynamically than just the recent bias of this branch?
Can We Do Better?
Yes: 2-level local predictor

- Can we get more information dynamically than just the recent bias of this branch?
- We can look at patterns (2-level local predictor) for a particular branch.
  - last eight branches 00100100, then it is a good guess that the next one is “1” (taken)
Can We Do Better?
Yes: 2-level local predictor

• “2-level” → Two tables
• Pattern History Table (PHT)
  – Indexed by PC (branch address)
  – Width ~ Pattern Complexity
• Branch History Table (BHT)
  – Indexed by \textit{pattern}
  – Same structure as used in the 2-bit bimodal, but different meaning!
  – No longer “what is this branch likely to do next”, now, “what is likely to come next in this pattern”
Can We Do Better?
Yes: 2-level local predictor

- even / odd branch?
Can We Do Better?

- Can we get more information dynamically than just the recent bias of this branch?
Can We Do Better?
Yes: Correlating Predictor

- Can we get more information dynamically than just the recent bias of this branch?
- **Correlating Branch Predictors** also look at other branches for clues
  ```java
  if (i == 0)
  ...
  if (i > 7)
  ...
  ```
- Typically use two indexes
  - Global history register --> history of last m branches (e.g., 0100011)
  - branch address
Correlating Branch Predictors

- The *global history register (ghr)* is a shift register that records the last $n$ branches (of any address) encountered by the processor.
  - "What does the pattern of recent branching done tell me?"
Two-level correlating branch predictors

- Can use both the PC address and the GHR

Most common – \textit{gshare}: use xor as the combining function.
Are we happy yet????

- Combining branch predictors use multiple schemes and a voter to decide which one typically does better for that branch.
Compaq/Digital Alpha 21264

Local Predictor

Global Predictor

Chooser

Branch Prediction

PC

10

3

2

GHR

12

2
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?
Aliasing in Branch Predictors

- Branch predictors will always be of finite size, while code size is relatively unlimited.
- What happens when (in the common case) there are more branches than entries in the branch predictor?
- We call these conflicts *aliasing*.
- We can have negative aliasing (when biases are different) or neutral aliasing (biases same). Positive aliasing is unlikely.
Bimodal aliasing

branch address

PHT

00
Local Predictor Aliasing
Gshare aliasing

2-bit predictors

PC -> xor -> ghr -> 00, 01, 00, 11
Branch Prediction

• Latest branch predictors significantly more sophisticated, using more advanced correlating techniques, larger structures, and soon possibly using AI techniques.
• Remember from earlier....
  – Presupposes what two pieces of information are available at fetch time?
    •
    •
  – Branch Target Buffer supplies this information.
OKAY. So how many of these crazy branch predictor variations do I need to memorize for CSE 141??

- What I want you to know about branch predictors:
  - Why they are useful (why do we put *so much work* into making good ones)?
  - What info do predictors need to operate, and where do they get this info?
  - How the simpler ones work, specifically...
    - 1-bit predictor
    - 2-bit bimodal predictor
    - 2-level local predictor
  - What some of the ‘additional tricks’ are, specifically...
    - What is a “Global History Register”?
    - What does a “combining function” do?
  - What problems can arise that confound prediction?
  - *Given a description*, how to analyze novel branch predictor performance
Defining CSE141 “standard parameters”
(And one more performance example while we’re at it)

```assembly
loop: lw $15, 1000($2)
    add $16, $15, $12
lw $18, 1004($2)
    add $19, $18, $12
    beq $19, $0, loop
    nop
```

What is the steady-state CPI of this code?

Assume branch taken many times.
Assume 5-stage pipeline, forwarding, early branch resolution, branch delay slot

*Always assume this architecture if not given the details*
Putting it all together.

For a given program on our 5-stage MIPS pipeline processor:

- 20% of insts are loads, 50% of instructions following a load are arithmetic instructions depending on the load.
- 20% of instructions are branches.
- We manage to fill 80% of the branch delay slots with useful instructions.

- What is the CPI of your program?

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.76</td>
</tr>
<tr>
<td>B</td>
<td>0.9</td>
</tr>
<tr>
<td>C</td>
<td>1.0</td>
</tr>
<tr>
<td>D</td>
<td>1.1</td>
</tr>
<tr>
<td>E</td>
<td>1.14</td>
</tr>
</tbody>
</table>
Given our 5-stage MIPS pipeline...

What is the steady state CPI for the following code?

Loop:   lw  r1, 0 (r2)
        add r2, r3, r4
        sub r5, r1, r2
        beq r5, $zero, Loop
        nop

<table>
<thead>
<tr>
<th>Selection</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1.25</td>
</tr>
<tr>
<td>C</td>
<td>1.5</td>
</tr>
<tr>
<td>D</td>
<td>1.75</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
That was a lot.

- Seriously!
- Loosely, we just covered ~30 years of processor design in 4 weeks
  - (The good ideas are always more obvious in hindsight...)
Pipelining Key Points

- ET = IC * CPI * CT
- Achieve high **throughput** without reducing instruction **latency**
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles by different instructions).
- Pipelining uses combinational logic to generate (and registers to propagate) control signals.
- Pipelining creates potential hazards.
Data Hazard Key Points

• Pipelining provides high throughput, but does not handle data dependences easily.
• Data dependences cause \textit{data hazards}.
• Data hazards can be solved by:
  – software (nops)
  – hardware stalling
  – hardware forwarding
• Our processor, and indeed all modern processors, use a combination of forwarding and stalling.
• $ET = IC \times CPI \times CT$
Control Hazard Key Points

- Control (branch) hazards arise because we must fetch the next instruction before we know:
  - if we are branching
  - where we are branching
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - *branch prediction*
  - branch delay slots
Branch Prediction Key Points

- Branch mispredicts are *expensive*, especially in deeper pipelines
- Predictors must answer three things correctly to avoid misprediction:
  1. Is the instruction at this address a branch?
  2. If so, are we likely to take this branch?
  3. If so, where is it going to take us?
- The best predictions combine multiple sources of information