CS2100 Computer Organisation

Pipelining
(AY2016/2017) Semester 2
Road Map: Part II

- Performance
- Assembly Language
- Processor: Datapath
- Processor: Control
- Pipelining
- Cache

- **Pipelining**
  - MIPS Pipeline
  - Pipeline datapath & control
  - Pipeline hazards
  - Branch prediction
Pipelining: Overview

- MIPS Pipeline Stages

- Pipelined Datapath and Control

- Pipeline Hazards
  - Structural
  - Data
  - Control

- Forwarding

- Branch prediction
Inspiration: Assembly Line

Simpler station tasks → more cars per hour. Simple tasks take less time, clock is faster.
The Laundry Example

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold and stash

- **Washer** takes 30 minutes

- **Dryer** takes 30 minutes

- “Folder” takes 30 minutes

- “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
- Steady state: 1 load every 2 hours
- If they learned pipelining, how long would laundry take?
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
- Steady state: 1 load every 30 min
- Potential speedup = $2 \text{ hr}/30 \text{ min} = 4$ (# stages)
- Time to fill pipeline takes 2 hours $\rightarrow$ speedup ↓
What IF: Slow Dryer

- Pipelined laundry now takes 5.5 hours!
- Steady state: One load every 1 hr (dryer speed)
- Pipeline rate is limited by the slowest stage
What IF: Dependency

- Brian is using the laundry for the first time; he wants to see the outcome of one wash + dry cycle first before putting in his clothes
- Pipelined laundry now takes 4 hours
Pipelining Lessons

- Pipelining doesn’t help latency of single task:
  - It helps the throughput of entire workload

- Multiple tasks operating simultaneously using different resources

- Possible delays:
  - Pipeline rate limited by slowest pipeline stage
  - Stall for dependencies (more on this later)
Datapath and Control

MIPS PIPELINE
MIPS Pipeline Stages (1/2)

- **Five Execution Stages**
  - **IF**: Instruction Fetch
  - **ID**: Instruction Decode and Register Read
  - **EX**: Execute an operation or calculate an address
  - **MEM**: Access an operand in data memory
  - **WB**: Write back the result into a register

- **Idea:**
  - Each execution stage takes 1 clock cycle
  - General flow of data is from one stage to the next

- **Exceptions:**
  - Update of PC and write back of register file – more about this later…
MIPS Pipeline Stages (2/2)

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

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

Several instructions are in the pipeline simultaneously!
MIPS Pipeline: **Datapath** (1/3)

- **Single-cycle implementation:**
  - Update all state elements (**PC**, register file, data memory) at the end of a clock cycle

- **Pipelined implementation:**
  - One cycle per pipeline stage
  - Data required for each stage needs to be stored separately (why?)
MIPS Pipeline: Datapath (2/3)

- Data used by *subsequent instructions*:
  - Store in programmer-visible state elements: PC, register file and memory

- Data used by *same instruction* in later pipeline stages:
  - Additional registers in datapath called *pipeline registers*
    - **IF/ID**: register between IF and ID
    - **ID/EX**: register between ID and EX
    - **EX/MEM**: register between EX and MEM
    - **MEM/WB**: register between MEM and WB

- Why no register at the end of WB stage?
MIPS Pipeline: Datapath (3/3)

Coloured rectangles are pipeline registers
Pipeline Datapath: **IF Stage**

- At the end of a cycle, **IF/ID** receives (stores):
  - Instruction read from DataMemory[ PC ]
  - PC + 4
- PC + 4
  - Also connected to one of the MUX's inputs (another coming later)
### Pipeline Datapath: ID Stage

**At the beginning of a cycle**

**IF/ID register supplies:**
- Register numbers for reading two registers
- 16-bit offset to be sign-extended to 32-bit

**At the end of a cycle**

**ID/EX receives:**
- Data values read from register file
- 32-bit immediate value
- PC + 4
### Pipeline Datapath: EX Stage

**At the beginning of a cycle**

<table>
<thead>
<tr>
<th>ID/EX register supplies:</th>
</tr>
</thead>
<tbody>
<tr>
<td>❖ Data values read from register file</td>
</tr>
<tr>
<td>❖ 32-bit immediate value</td>
</tr>
<tr>
<td>❖ PC + 4</td>
</tr>
</tbody>
</table>

**At the end of a cycle**

<table>
<thead>
<tr>
<th>EX/MEM receives:</th>
</tr>
</thead>
<tbody>
<tr>
<td>❖ (PC + 4) + (Immediates x 4)</td>
</tr>
<tr>
<td>❖ ALU result</td>
</tr>
<tr>
<td>❖ isZero? signal</td>
</tr>
<tr>
<td>❖ Data Read 2 from register file</td>
</tr>
</tbody>
</table>
**Pipeline Datapath: MEM Stage**

At the beginning of a cycle, **EX/MEM** register supplies:

- \((PC + 4) + (\text{Immediates} \times 4)\)
- ALU result
- \(\text{isZero?} \) signal
- Data Read 2 from register file

At the end of a cycle, **MEM/WB** receives:

- ALU result
- Memory read data
Pipeline Datapath: **WB Stage**

At the beginning of a cycle:

**MEM/WB** register supplies:
- ALU result
- Memory read data

At the end of a cycle:
- Result is written back to register file (if applicable)
- **There is a bug here**
Corrected Datapath (1/2)

- Observe the “Write register” number
  - Supplied by the IF/ID pipeline register
  - It is NOT the correct write register for the instruction now in WB stage!

- Solution:
  - Pass “Write register” number from ID/EX through EX/MEM to MEM/WB pipeline register for use in WB stage
  - i.e. let the "Write register" number follows the instruction through the pipeline until it is needed in WB stage
Corrected Datapath (2/2)

Instruction memory

Address

PC

Add

4

Add

result

Shift

left 2

Instruction

ID/EX

EX/MEM

MEM/WB

Read register 1
Read register 2
Read data 1
Read data 2

Write register
Write data

Add

Add result

Shift left 2

Zero

ALU result

ALU

Data memory

Address

Read data

Write data

0 Mux

0 Mux

0 Mux

1 Mux
Pipeline Control: Main Idea

- **Same control signals as single-cycle datapath**
- **Difference:** Each control signal belongs to a particular pipeline stage
Pipeline Control: Try it!

Try associating each control signal with the correct pipeline stage.
Pipeline Control: Grouping

- Group control signals according to pipeline stage

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemTo Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>op1</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EX Stage</th>
<th>MEM Stage</th>
<th>WB Stage</th>
<th></th>
<th>ALUop</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>MemTo Reg</th>
<th>Reg Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td>op1</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td>0 1</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td>1 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS2100 Pipelining
Pipeline Control: Implementation

Signal generated in ID stage

Signal used in EX stage and onwards

Signal propagated along until utilized

Instruction
MIPS Pipeline: Datapath and Control

Pipelining
PIPELINE PERFORMANCE
Single Cycle Processor: Performance

- Cycle time:
  - \( CT_{seq} = \sum_{k=1}^{N} T_k \)
  - \( T_k \) = Time for operation in stage \( k \)
  - \( N \) = Number of stages

- Total Execution Time for \( I \) instructions:
  - \( Time_{seq} = \text{Cycles} \times \text{CycleTime} \)
    \( = I \times CT_{seq} = I \times \sum_{k=1}^{N} T_k \)
Single Cycle Processor: Example

- Cycle Time
  - Choose the longest total time = 8ns
- To execute 100 instructions:
  - 100 x 8ns = 800ns

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inst Mem</th>
<th>Reg read</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Reg write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Multi-Cycle Processor: Performance

- **Cycle time:**
  - \( CT_{multi} = \max(T_k) \)
  - \( \max(T_k) = \) longest stage duration among the N stages

- **Total Execution Time for I instructions:**
  - \( Time_{multi} = \text{Cycles} \times Cycle\text{Time} \)
    \[ = I \times Average\ CPI \times CT_{multi} \]
  - Average CPI is needed because each instruction take different number of cycles to finish
## Multi-Cycle Processor: Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inst Mem</th>
<th>Reg read</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Reg write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

- **Cycle Time:**
  - Choose the longest stage time = 2ns

- **To execute 100 instructions**, with a given average CPI of 4.6

- **100 x 4.6 x 2ns = 920ns**
Pipeline Processor: Performance

- **Cycle Time:**
  - $CT_{pipe} = \max(T_k) + T_d$
  - $\max(T_k)$ = longest time among the N stages
  - $T_d$ = Overhead for pipelining, e.g. pipeline register

- **Cycles needed for I instructions:**
  - $I + N - 1$
  - $N - 1$ is the cycles wasted in filling up the pipeline

- **Total Time needed for I instructions:**
  - $Time_{pipe} = Cycle \times CT_{pipe}$
  - $= (I + N - 1) \times (\max(T_k) + T_d)$
Pipeline Processor: Example

- **Cycle Time**
  - assume pipeline register delay of **0.5ns**
  - longest stage time + overhead = $2 + 0.5 = 2.5\text{ns}$

- **To execute 100 instructions:**
  - $(100 + 5 - 1) \times 2.5\text{ns} = 260\text{ns}$

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inst Mem</th>
<th>Reg read</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Reg write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Pipeline Processor: Ideal Speedup

- Assumptions for ideal case:
  - Every stage takes the same amount of time:
    \[ \sum_{k=1}^{N} T_k = N \times T_k \]
  - No pipeline overhead \( T_d = 0 \)
  - Number of instructions \( I \), is much larger than number of stages, \( N \)

- Note: The above also shows how pipeline processor loses performance
Pipeline Processor: Ideal Speedup

\[ \text{Speedup}_{\text{pipe}} = \frac{\text{Time}_{\text{seq}}}{\text{Time}_{\text{pipe}}} \]

\[ = \frac{\sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k) + T_d)} \]

\[ = \frac{I \times N \times T_k}{(I+N-1) \times T_k} \]

\[ \approx \frac{I \times N \times T_k}{I \times T_k} \]

\[ \approx N \]

**Conclusion:**
Pipeline processor can gain \( N \) times speedup, where \( N \) is the number of pipeline stages.
Summary for different implementations

**Single-Cycle**

- Cycle 1
- Cycle 2

**Multi-Cycle**

- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9
- Cycle 10

**Pipeline**

- Load
- Store
- R-type

CS2100 Pipelining
Review Question

Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

a) How many cycles will it take to execute the code on a single-cycle datapath?

b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?

c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?

d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?
Grinding to a halt.....

PIPELINE HAZARDS
Pipeline Hazards

- **Speedup from pipeline implementation:**
  - Based on the assumption that a new instruction can be "pumped" into pipeline every cycle

- **However, there are pipeline hazards**
  - Problems that prevent next instruction from immediately following previous instruction
    - **Structural hazards:**
      - Simultaneous use of a hardware resource
    - **Data hazards:**
      - Data dependencies between instructions
    - **Control hazards:**
      - Change in program flow
Graphical Notation for Pipeline

- Horizontal = the stages of an instruction
- Vertical = the instructions in different pipeline stages

Program execution order (in instructions):
- `lw $10, 20($1)`
- `sub $11, $2, $3`

```
lw $10, 20($1)
sub $11, $2, $3
```
Structure Hazard: Example

- If there is only a **single memory module**:

  ![Diagram of instruction order and memory access]

  *Load and Inst 3 access memory in the same cycle!*

  **Instruction Order**
  - Load
  - Inst 1
  - Inst 2
  - Inst 3

  **Time (clock cycles)**
Solution 1: Stall the Pipeline

Instruction Order:
- Load
- Inst 1
- Inst 2
- Stall
- Inst 3

Time (clock cycles)

Delay (Stall) Inst 3 for 1 cycle
Solution 2: Separate Memory

- Split memory into:
  - **Data** and **Instruction** memory

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
</tr>
<tr>
<td>Inst 1</td>
</tr>
<tr>
<td>Inst 2</td>
</tr>
<tr>
<td>Inst 3</td>
</tr>
</tbody>
</table>

**Load** uses Data Memory

**Inst 3** uses Instr. Memory

```
<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Mem</th>
<th>Reg</th>
<th>ALU</th>
<th>Mem</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst 1</td>
<td>Mem</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>Inst 2</td>
<td>Mem</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>Inst 3</td>
<td>Mem</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
</tbody>
</table>
```
Quiz: Is there another conflict?

Instruction Order:

- **Inst 0**: Mem, Reg, ALU, Mem, Reg
- **Inst 1**: Mem, Reg, ALU, Mem, Reg
- **Inst 2**: Mem, Reg, ALU, Mem, Reg
- **Inst 3**: Mem, Reg, ALU, Mem, Reg

Time (clock cycles) vs. Instruction Order
**Instruction Dependencies**

- Instructions can have relationships that prevent pipeline execution:
  - Although a partial overlap maybe possible in some cases

- When different instructions accesses (read/write) the same register:
  - Register contention is the cause of dependency
  - Known as **data dependency**

- When the execution of an instruction depends on another instruction:
  - Control flow is the cause of dependency
  - Known as **control dependency**

- Failure to handle dependencies can affect **program correctness!**
Data Dependency: RAW

"Read-After-Write" Definition:
- Occurs when a later instruction reads from the destination register written by an earlier instruction
- Also known as true data dependency

```
\[
\begin{align*}
i1 & : \text{add } \$1, \$2, \$3 \quad \text{#writes to } \$1 \\
i2 & : \text{sub } \$4, \$1, \$5 \quad \text{#reads from } \$1 \\
\end{align*}
\]
```

Effect of incorrect execution:
- If \( i2 \) reads register \( \$1 \) before \( i1 \) can write back the result, \( i2 \) will get a stale result (old result)
Other Data Dependencies

- Similarly, we have:
  - **WAR**: Write-after-Read dependency
  - **WAW**: Write-after-Write dependency

- Fortunately, these dependencies do not cause any pipeline hazards

- They affects the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor
RAW Dependency: Hazards?

Suppose we are executing the following code fragment:

```
sub $2, $1, $3    #i1
and $12, $2, $5  #i2
or $13, $6, $2   #i3
add $14, $2, $2  #i4
sw $15, 100($2)  #i5
```

- Note the multiple uses of register $2
- Question:
  - Which are the instructions require special handling?
RAW Data Hazards: Illustration

- Value from prior instruction is needed before write back

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>Value of register $2$:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC 1</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

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

Questions:

- When is the result from `sub` instruction actually produced?
  - End of EX stage for `sub` or clock cycle 3
- When is the data actually needed by `and`?
  - Beginning of `and`’s EX stage or clock cycle 4
- When is the data actually needed by `or`?
  - Beginning of `or`’s EX stage or clock cycle 5

Solution:

- **Forward** the result to any trailing (later) instructions before it is reflected in register file
- **Bypass (replace)** the data read from register file
### Solution: Forwarding

**Time (in clock cycles)**

<table>
<thead>
<tr>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$ :</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/−20</td>
<td>−20</td>
<td>−20</td>
<td>−20</td>
<td>−20</td>
</tr>
<tr>
<td>Value of EX/MEM : $X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$−20$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
</tr>
<tr>
<td>Value of MEM/WB : $X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$−20$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
</tr>
</tbody>
</table>

**Sub $2$, $1$, $3$**

**Add $12$, $2$, $5$**

**Or $13$, $6$, $2$**

**Add $14$, $2$, $2$**

**Sw $15$, $100($2$)**

- **Forward** results from one stage to another
- **Bypass** data read from register file

---

CS2100

Pipelining
Data Hazard: **LOAD** Instruction

- **lw $2, 20($1)**
  - IM → Reg
  - DM → Reg

- **and $4, $2, $5**
  - IM → Reg
  - DM → Reg

- **or $8, $2, $6**
  - IM → Reg
  - DM → Reg

- **add $9, $4, $2**
  - IM → Reg
  - DM → Reg

- **slt $1, $6, $7**
  - IM → Reg
  - DM → Reg

**Cannot** solve with forwarding!
Data is needed **before** it is actually produced!
Data Hazard: **LOAD** Instruction Solution

Program execution order (in instructions)

- lw $2, 20($1)
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2
- slt $1, $6, $7

Time (in clock cycles)

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>OC 1</th>
<th>OC 2</th>
<th>OC 3</th>
<th>OC 4</th>
<th>OC 5</th>
<th>OC 6</th>
<th>OC 7</th>
<th>OC 8</th>
<th>OC 9</th>
<th>OC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $2, 20($1)</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>and $4, $2, $5</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>or $8, $2, $6</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>add $9, $4, $2</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>slt $1, $6, $7</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
</tr>
</tbody>
</table>

**Stall** the pipeline!
Exercise #2: Without Forwarding

How many cycles will it take to execute the following code on a 5-stage pipeline without forwarding?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Exercise #2: Without Forwarding

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise #3: Without Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline **without** forwarding?

  ```
  lw $2, 20 ($3)
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100 ($2)
  ```
Exercise #3: Without Forwarding

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise #4: With Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Exercise #4: With Forwarding

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise #5: With Forwarding

- How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding?

```
lw    $2,  20($3)
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```
Exercise #5: With Forwarding

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CONTROL DEPENDENCY
Control Dependency

Definition:
- An instruction $j$ is control dependent on $i$ if $i$ controls whether or not $j$ executes
- Typically $i$ would be a branch instruction

Example:

```
  i1: beq $3, $5, label  # Branch
  i2: add $1, $2, $4     # depends on i1
  ...  ...  ...  ...
```

Effect of incorrect execution:
- If $i3$ is allowed to execute before $i2$ is determined, register $\$1$ maybe incorrectly changed!
Control Dependency: Example

Let us turn to a code fragment with a conditional branch:

```
40  beq $1, $3, 7
44  and $12, $2, $5
48  or $13, $6, $2
52  add $14, $2, $2
    ............
72  lw $4, 5($7)
```

How does the code affect a pipeline processor?
Pipeline Execution: **IF** Stage

- Read instruction from memory using the address in PC and put it in **IF/ID** register
- PC address is incremented by 4 and then written back to the PC for next instruction
Control Dependency: Why?

Decision is made in MEM stage: Too late!
Control Dependency: Example

Program execution order (in instructions)

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9

40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)

Wrong Execution!
Control Hazards: Stall Pipeline?

- Wait until the branch outcome is known and then fetch the correct instructions
  → Introduces 3 clock cycles delay
Control Hazards: Reducing the Penalty

- Branching is very common in code:
  - A 3 cycles stall penalty is too heavy!

- Many techniques invented to reduce the control hazard penalty:
  - Move branch decision calculation to earlier pipeline stage
    - Early Branch Resolution
  - Guess the outcome before it is produced
    - Branch Prediction
  - Do something useful while waiting for the outcome
    - Delayed Branching
Reduce Stalls: Early Branch (1/3)

- Make decision in **ID** stage instead of **MEM**
  - Move branch target address calculation
  - Move register comparison → cannot use ALU for register comparison any more
Reduce Stalls: Early Branch (2/3)

Register comparison moved to ID stage
Reduce Stalls: Early Branch (3/3)

- Wait until the branch decision is known:
  - Then fetch the correct instructions
- Reduced to 1 clock cycle delay
Early Branch: Problems (1/3)

However, if the register(s) involved in the comparison is produced by preceding instruction:

- Further stall is still needed!

```
add $s0, $s1, $s2
beq $s0, $s3, Exit
```

$s0$ is needed before it is produced!
Early Branch: **Problems** (2/3)

- **Solution:**
  - Add forwarding path from ALU to ID stage
  - **One clock cycle delay** is still needed

![Diagram showing pipeline stages](image)
Early Branch: **Problems (3/3)**

- **Problem is worse with load followed by branch**
- **Solution:**
  - MEM to ID forwarding and 2 more stall cycles!
  - In this case, we ended up with 3 total stall cycles → no improvement!

```
lw $s0, 0($s1)
beq $s0, $s3, Exit
```
Reduce Stalls: Branch Prediction

- There are many branch prediction schemes
  - We only cover the simplest in this course 😊

- Simple prediction:
  - All branches are assumed to be **not taken**
    - Fetch the successor instruction and start pumping it through the pipeline stages

- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly ➔ No pipeline stall
  - **Taken**: Guessed wrongly ➔ Wrong instructions in the pipeline ➔ **Flush** successor instruction from the pipeline
Branch Prediction: Correct Prediction

Program execution order (in instructions)

40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2

Branch is known to be not taken in cycle 3 ➔ no stall needed!
Branch Prediction: **Wrong Prediction**

Branch is known to be **taken** in cycle 3

- "and" instruction should not be executed
- Flushed from pipeline
Exercise #6: No Branch Prediction

- How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding but no branch prediction?
  - Decision making moved to ID stage

<table>
<thead>
<tr>
<th>Loop:</th>
<th>addi</th>
<th>$s0, $zero, 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addi</td>
<td>$s0, $s0, -1</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>$s0, $zero, Loop</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>$t0, $t1, $t2</td>
</tr>
</tbody>
</table>

- Total instructions = 1 + 10×2 + 1 = 22
- **Ideal** pipeline = 4 + 22×1 = 26 cycles
### Exercise #6: No Branch Prediction

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addi</strong></td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addi</strong></td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>bne</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addi</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelining
Exercise #7: Branch Prediction

How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding and branch prediction?

Total instructions = 1 + 10\times2 + 1 = 22

Ideal pipeline = 4 + 22\times1 = 26 cycles
Exercise #7: **Branch Prediction**

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- **CS2100**
- **Pipelining**
Reduce Stalls: **Delayed Branch**

- **Observation:**
  - Branch outcome takes \( X \) number of cycles to be known
  - \( X \) cycles stall

- **Idea:**
  - Move **non-control dependent instructions** into the \( X \) slots following a branch
    - Known as the **branch-delay slot**
  - These instructions are executed **regardless of the branch outcome**

- **In our MIPS processor:**
  - Branch-Delay slot = 1 (with the early branch)
Delayed Branch: Example

**Nondelayed branch**

- `or $8, $9, $10`
- `add $1, $2, $3`
- `sub $4, $5, $6`
- `beq $1, $4, Exit`
- `xor $10, $1, $11`

**Delayed branch**

- `add $1, $2, $3`
- `sub $4, $5, $6`
- `beq $1, $4, Exit`
- `or $8, $9, $10`
- `xor $10, $1, $11`

Exit:

- The "or" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome
  - Same behavior as the original code!
Delayed Branch: Observation

- **Best case scenario**
  - There is an instruction *preceding the branch* which *can be moved* into delayed slot
  - Program correctness must be preserved!

- **Worst case scenario**
  - Such instruction cannot be found
    - Add a no-op (nop) instruction in the branch-delay slot
  - Re-ordering instructions is a common method of program optimization
    - Compiler must be smart enough to do this
    - Usually can find such an instruction at least 50% of the time
For your reference

EXPLORATION
Multiple Issue Processors (1/2)

- Multiple Issue processors
  - **Multiple instructions** in every pipeline stage
  - 4 washer, 4 dryer...

- **Static multiple issue:**
  - EPIC (Explicitly Parallel Instruction Computer) or VLIW (Very Long Instruction Word), e.g. IA64
  - Compiler specifies the set of instructions that execute together in a given clock cycle
  - Simple hardware, complex compiler

- **Dynamic multiple issue:**
  - Superscalar processor: Dominant design of modern processors
  - Hardware decides which instructions to execute together
  - Complex hardware, simpler compiler
Multiple Issue Processors (2/2)

- A 2-wide superscalar pipeline:
  - By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed.

![Diagram of pipeline stages with instructions being processed in parallel]
Summary

- Pipelining is a fundamental concept in computer systems!
  - Multiple instructions in flight
  - Limited by length of the longest stage
  - Hazards create trouble by stalling pipeline

- Pentium 4 has 22 pipeline stages!
Reading Assignment

- 3rd edition
  - Sections 6.1 – 6.3
  - Sections 6.4 – 6.6 (data hazards and control hazards in details; read for interest; not in syllabus)

- 4th edition
  - Sections 4.5 – 4.6
  - Sections 4.7 – 4.8 (data hazards and control hazards in details; read for interest; not in syllabus)