Lecture 2

Processor Pipeline
Overview

- Instruction Set Architecture
  - Overview of MIPS

- Quantifying Processor Performance
  - Review of commonly used formula and terms

- Simple Processor Architectures:
  - Single-Cycle Architecture
  - Multi-Cycle Architecture
  - Pipeline Architecture
    - Hazards detection and resolution
INSTRUCTION SET
ARCHITECTURE
Design: Storage Architecture

- Major design consideration for an ISA:
  - Decide where to store intermediate computation result internally in the processor
  - Influence the format of the instructions

- Dominant storage architecture:
  - General Purpose Register (GPR)
  - Internal storage = a number of addressible registers

- Alternatives, less common storage architectures:
  - Stack, Accumulator, Memory-Only
Design Philosophy

- Two major approaches in Instruction Set Architecture (ISA) design:
  - **Complex Instruction Set Computer (CISC)**
    - Provide as many instructions as possible to the programmer
    - Possible for a single instruction to accomplish complex operation, e.g. matrix multiplication
  - **Reduced Instruction Set Computer (RISC)**
    - Provide the minimum number of instructions to the programmer
    - Complicated operations are coded from the simpler counterparts
CISC Instruction Set Architecture

- Instructions carry different amount of information
  - Varying length of instruction

- Example CISC Microprocessor:
  - The Intel IA32 Architecture:
    - Basic instructions from earlier x86 series are still maintained
    - Each generation adds on more instructions
    - Single Instruction Multiple Data (SIMD) instructions from the Multimedia eXtensions
    - Even more SIMD instructions from the 3DNow! extensions
RISC Instruction Set Architecture

Observation:
- Over 70% of CISC instructions not used often.
  - Shrink instruction set to the most common 30%

Characteristics:
- Instructions are of fixed length
- Very simple processor designed to be very fast
- Almost all instructions limited to registers only
  - Memory operations limited to load and store only
- Rely on compiler to optimize code

Example of RISC machine:
- MIPS CPUs
The Current State of CISC ISA

- RISC has essentially won the "RISC vs CISC" debate:
  - Complex instruction set processor is much harder to design / validate / optimize

- Intel IA32/x86-64 processors use the following approach:
  - Each CISC instruction is broken into a series of simpler RISC-like operation (μops) internally
  - The processing core executes the μops instead of the CISC instruction directly
Example ISA: MIPS Basics (1/2)

- We will use MIPS as an example ISA to illustrate pipeline processor designs

- Typical GPR ISA:
  - 32 registers; register 0 always has the value 0
  - We'll denote them as $0, $1, \ldots, $31 (or $r0… $r30)

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Register-to-Register</td>
<td>ADD rd, rs, rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rd ← rs + rt</td>
</tr>
<tr>
<td></td>
<td>Immediate Value</td>
<td>ADDI rt, rs, IMV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rt ← rs + IMV</td>
</tr>
<tr>
<td></td>
<td>Unsigned</td>
<td>ADDU rd, rs, rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rd ← rs + rt</td>
</tr>
</tbody>
</table>

[CS5222 Adv. Comp. Arch. AY1415S2]
## MIPS Basics (2/2)

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory</strong></td>
<td>Load from memory</td>
<td>LW ( rt, \text{offset}(rs) ) ( rt \leftarrow \text{mem}[rs + \text{offset}] )</td>
</tr>
<tr>
<td></td>
<td>Store to memory</td>
<td>SW ( rt, \text{offset}(rs) ) ( \text{mem}[rs + \text{offset}] \leftarrow rt )</td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>Conditional</td>
<td>BEQ ( rs, rt, \text{label} ) ( \text{Jump to label if } rs == rt, \text{otherwise continue to next instruction} )</td>
</tr>
<tr>
<td></td>
<td>Unconditional (Jump)</td>
<td>J ( \text{label} ) ( \text{Jump to label} )</td>
</tr>
</tbody>
</table>
MIPS Assembly Code: An Example

// Iterative Fibonacci in C
// Assume n is non-negative

int cur = 1, p1 = 1, p2 = 1, j;
// p1 = n-1 fib term
// p2 = n-2 fib term

if( n > 2 )
    for ( j=3; j<=n; j++ ) {
        cur = p1 + p2;
        p2 = p1;
        p1 = cur;
    }
// cur is the nth fib term

#$r1 = cur, $r2 = p1, $r3 = p2
#$r4 = j, $r13 = n
li $r1, 1
li $r2, 1
li $r3, 1
subi $r10, $r13, 2
ble $r10, $r0, done
li $r4, 3
loop: add $r1, $r2, $r3
move $r3, $r2
move $r2, $r1
addi $r4, $r4, 1
ble $r4, $r13, loop
done:
PROCESSOR PERFORMANCE
Execution Time: Clock Cycles

- **Clock cycles** is the basic time unit in machine

\[
\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}
\]

- **Cycle time** (or cycle period or clock period)
  - Time between two consecutive rising edges, measured in seconds.

- **Clock rate** (or clock frequency)
  - \(= \frac{1}{\text{cycle-time}}\)
  - \(= \frac{\text{number-of-cycles}}{\text{second}}\)
  - Unit is in Hz, \(1\ \text{HZ} == 1\ \text{cycle} / \text{second}\)
Execution Time: Introducing CPI

- A given program will require

  Some number of instructions (machine instructions) × Average Cycle per Instruction (CPI)

  Some number of cycles × cycle time

  Some number of seconds

- We use the average of CPI as different instruction take different number of cycles to finish
Execution Time: Version 2.0

- **Average Cycle Per Instruction (CPI)**
  
  \[ \text{CPI} = \frac{(\text{CPU time} \times \text{Clock rate})}{\text{Instruction count}} = \frac{\text{Clock cycles}}{\text{Instruction count}} \]

  - **CPU time** = \( \frac{\text{Seconds}}{\text{Program}} \)
  - **Instructions** = \( \frac{\text{Instructions}}{\text{Program}} \)
  - **Cycles** = \( \frac{\text{Cycles}}{\text{Instruction}} \)
  - **Seconds** = \( \frac{\text{Seconds}}{\text{Cycle}} \)

  \[ \text{CPI} = \sum_{k=1}^{n} \text{CPI}_k \times F_k \text{ where } F_k = \frac{I_k}{\text{Instruction count}} \]

  \[ I_k = \text{Instruction frequency} \]
SIMPLE PROCESSOR ARCHITECTURE
Computer Organization: Recap

- Major components of a computer system:
  - Processor, Memory and Devices
  - Buses for transporting data between component

- Von Neumann’s **stored-memory concept**:
  - Data and program are stored in memory
## 5-Stage MIPS Instruction Execution

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Fetch</th>
<th>Decode &amp; Operand Fetch</th>
<th>ALU</th>
<th>Memory Access</th>
<th>Result Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add $3, $1, $2</td>
<td>Read inst. at [PC]</td>
<td>oRead [$1] as opr1</td>
<td>Result = opr1 + opr2</td>
<td>Use MemAddr to read from memory</td>
<td>Result stored in $3</td>
</tr>
<tr>
<td></td>
<td>lw $3, 20( $1 )</td>
<td>Read inst. at [PC]</td>
<td>oRead [$1] as opr1</td>
<td>MemAddr = opr1 + opr2</td>
<td></td>
<td>Memory data stored in $3</td>
</tr>
<tr>
<td></td>
<td>beq $1, $2, label</td>
<td>Read inst. at [PC]</td>
<td>oRead [$1] as opr1</td>
<td>Taken = (opr1 == opr2)? Target = PC + Label*</td>
<td></td>
<td>if (Taken) PC = Target</td>
</tr>
</tbody>
</table>

```plaintext
add $3, $1, $2
lw $3, 20( $1 )
beq $1, $2, label
```
**Single-Cycle Implementation (2/2)**

- **Characteristics:**
  - Each instruction takes 1 long clock cycle
  - Updating the visible states (PC, data memory, and registers) at the end of the clock cycle
  - No intermediate storage during execution

- **Drawbacks:**
  - Inefficient if instructions require different amount of work
Multi-Cycle Implementation (1/2)
Multi-cycle implementation (2/2)

- **Characteristics:**
  - Each instruction takes different number of cycles to execute

- **Average Cycle Per Instruction (CPI):**
  - **Assumptions:**
    - 4 cycles for store and branches
    - 5 cycles for ALU and load
  - **Instruction Mix:**
    - 40% ALU, 20% Branch, 20% load, 20% store
  - **CPI:**
    - \((20\% + 20\%) \times 4 + (40\% + 20\%) \times 5 = 4.6\)
  - Assume 1ns clock period (1 GHz processor)
    - Average instruction execution time 4.6ns
Processor Pipelining (1/4)

- A technique to improve processor performance by overlapping execution of multiple instructions

- **Basic Idea:**
  - Split the instruction execution into multiple stages
    - Similar to multi-cycle implementation
  - Start a new instruction in each clock cycle
  - Different pipeline stages are completing different parts of different instructions in parallel
Processor Pipelining (2/4)
Processor Pipelining (3/4)
Pipeline Registers / Latches
- Added as buffers between two adjacent pipeline stages
- Passes data and control signal to the next pipeline stage
- Content is copied to the next pipeline latch until it is no longer useful
- Introduces delay/overhead
Processor Pipelining

- **Characteristics:**
  - Each pipeline stage corresponds to a **clock cycle**
  - Each instruction effectively need to travel the whole pipeline depth during execution (why?)
    - i.e. Each instruction takes 5 clock cycles in this example

- **Advantage:**
  - Pipelining helps **throughput** (number of instructions executed in unit time)
  - Pipelining **does not** improve **latency** of a single instruction (amount of time taken for execution)
Ideal Pipeline

- **Assumptions:**
  - Perfectly balanced pipeline stages
    - Each stage takes exactly the same amount of time
  - Instruction utilize the same hardware resource in every pipeline stage
    - Good utilization of hardware resource
  - There are no stalls for dependencies
    - Instructions can always be executed in lock-step fashion

- **Optimal Behavior:**
  - Complete one instruction every cycle
    - \( \text{CPI} = 1 \)
  - Speedup is equal to number of pipeline stages
Real Pipeline

**Reality:**
- Stages will not be perfectly balanced
- Pipelining involves overhead ($T_d$)

**Let**
- $T_i =$ Execution time of the $i^{th}$ pipeline stage
- $N =$ Total number of pipeline stages
- Time without pipelining = $\sum_{i=1}^{N} T_i$
- Time with pipelining = $\max(T_i) + T_d$

**Speedup = $N$** only if $T_d=0$ and $T_i = T_j$ for all $i, j$
Limitation Pipeline Depth

- Pipeline overheads
  - Pipeline register delay: setup time etc.
  - **Clock skew**: maximum delay between the clock arrival times at any two registers

- Practical limits on the pipeline depth
  - Once clock cycle is as small as sum of clock skew and latch overhead, no further pipelining is useful
ILLUSTRATIVE EXAMPLE
Single-Cycle, Multi-Cycle, Pipeline (1/3)

- Given the following timing information:

<table>
<thead>
<tr>
<th>Inst</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></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>

- Single-cycle implementation
  - All instructions take as much time as the slowest one
  - Cycle time = 8ns

- Multi-cycle and pipelined implementation
  - Cycle time = 2ns
Single-Cycle, Multi-Cycle, Pipeline (2/3)

Single-Cycle Implementation:

Load Store Waste

Cycle 1 Cycle 2

Clk

Multi-Cycle Implementation:

Load Store R-type

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

Clk

Pipeline Implementation:

Load IF ID EX MEM WB

Store IF ID EX MEM WB

R-type IF ID EX MEM WB

[CS5222 Adv. Comp. Arch. AY1415S2]
Suppose we execute 100 instructions

**Single-Cycle Machine**
- $8 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 800 \text{ ns}$

**Multi-cycle Machine**
- $2 \text{ ns/cycle} \times 4.6 \text{ CPI} \times 100 \text{ inst} = 920 \text{ ns}$

**Ideal pipelined machine**
- $2 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle pipeline fill}) = 208 \text{ ns}$
PIPELINE HAZARDS
Pipeline Performance: Amdahl’s Law

- Performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

\[
\text{Speedup} = \frac{\text{Time}_{\text{old}}}{\text{Time}_{\text{new}}}
\]

\[
\text{Time}_{\text{new}} = \text{Time}_{\text{old}} \times \left( (1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)
\]

\[
\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]
Corollaries

- **Corollary 1:**
  - If the enhancement is only applicable for a fraction of a task, we cannot speed up the task by more than
  \[
  \frac{1}{1 - \text{Fraction}_{\text{enhanced}}}
  \]

- **Corollary 2:**
  - Make the common case first (In making a design trade-off, favor the frequent case over the infrequent case)
Pipeline performance (1/2)

\[
\text{Speedup} = \frac{\text{Avg instr time unpipelined}}{\text{Avg instr time pipelined}}
\]

\[
= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{clock period unpipelined}}{\text{clock period pipelined}}
\]

- If we assume perfectly balanced pipeline

\[
\text{Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}
\]

- If all instructions take equal number of cycles in multi-cycle implementation

\[
\text{CPI unpipelined} = \text{Pipeline depth}
\]
Pipeline performance (2/2)

\[ \text{CPI pipelined} = \text{Ideal CPI} + \text{stall cycles per instr} \]
\[ = 1 + \text{stall cycles per instr} \]

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{stall cycles per instr}} \]

- If there are no pipeline stalls, pipelining can improve performance by the depth of the pipeline
Pipeline Hazards

There are a number of pipeline hazards which prevent safe parallel execution of instructions:

- **Structural hazards**
  - Simultaneous use of a hardware resource

- **Data hazards**
  - Data dependencies between instructions

- **Control hazards**
  - Change in program flow
Structural Hazard: Example

- Suppose we have a single memory module:

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Load</th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
</tbody>
</table>

Load and Inst 3 access memory in the same cycle!
Structural Hazard: Stall as Solution

Instruction Order

Load
Inst 1
Inst 2
Stall
Inst 3

Time (clock cycles)

Delay (Stall)
Inst 3 for 1 cycle

[CS5222 Adv. Comp. Arch. AY1415S2]
Structural Hazard: Alternative Solution

- Better Solution:
  - Split cache memory into: **Data and Code Cache**

```
<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 1</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Reg</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Inst 2</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Reg</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Inst 3</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Reg</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>Mem</td>
</tr>
<tr>
<td>Load</td>
</tr>
<tr>
<td>Load uses Data Cache</td>
</tr>
</tbody>
</table>
```

- Inst 3 uses Code Cache

[CS5222 Adv. Comp. Arch. AY1415S2]
Quiz: What about this?

- Identify another resource conflict and its solution:
Pipeline Hazards

INSTRUCTION DEPENDENCY
Instruction Dependency

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

- When different instructions read or write from 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**!
Program Correctness

- If a program consisting of a sequence $i_0, i_1, i_2, \ldots i_{n-1}$ instructions is executed sequentially
  - Then, any instruction $i_t$ is not executed until $i_{t-1}$ has executed to completion

- If several instructions are executed concurrently (e.g. in a pipeline), the outcome must be exactly the same as the case of sequential execution
  - A different outcome is **WRONG**!

- Let us find out how different types of dependency affect the program correctness
Data Dependency: **Read-After-Write**

**Definition:**
- Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction.
- aka **true dependency** or RAW dependency.

**Example:**

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

**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 affect the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor
Dependencies: **Detect and Resolution**

- True data dependency and control dependency are unavoidable:
  - A pipeline design must be equipped to handle these hazards properly

- The main tasks are:
  - **Detection:**
    - Monitor the instructions in the pipeline to detect any dependency
  - **Resolution:**
    - Look for ways to ensure program correctness and reduce overhead
Data Hazards: Example

Observe 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**

<table>
<thead>
<tr>
<th>CC</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>[$2]</td>
<td>10</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>
</tbody>
</table>

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Quiz: How to detect RAW hazard?

- Recall that:
  - Each latch stores information to be used in execution until it is needed
- Give the "formula/condition" to check for RAW hazard by referring to fields in the latches
Data Hazards: Resolution

- We now know how to detect RAW hazard:
  - What is the solution?

- Simplest solution:
  - Just stall the offending instruction until the condition turned false
    - Undesirable due to the penalty
  - Is there more efficient solution?

- Hint: When is the result produced and when is the result needed in RAW hazard?
Data Forwarding: Illustration

<table>
<thead>
<tr>
<th>CC</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>[$2]</td>
<td>10</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>EX/MEM</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>
<td>X</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>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`
- `and $12, $2, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`
Data Hazards: Example 2

Let us try another code fragment:

\[
\begin{array}{ll}
\text{l} \text{w} & \$2, 20(\$1) \quad \#i1 \\
\text{a} \text{n} \text{d} & \$4, \$2, \$5 \quad \#i2 \\
\text{o} \text{r} & \$8, \$6, \$2 \quad \#i3 \\
\text{a} \text{d} \text{d} & \$9, \$4, \$2 \quad \#i4
\end{array}
\]

Note the multiple uses of register \$2

Question:

With the forwarding paths discussed, can we execute this code without stalling?
Data Hazards: Problem with $lw$

- Forwarding fails to solve the problem!

<table>
<thead>
<tr>
<th>CC</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>
</table>

lw $2, 20($1)

and $4, $2, $5$

or $8, $2, $6$

add $9, $4, $2$

slt $1, $6, $7$
Data Hazards: Solution for LW

- We have to stall the pipeline for 1 cycle

```
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```
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: blt there, $0, $1 ; Branch
  i2: add $0, $1, $2 ; depends on i1
  ... ... ...
```

Effect of incorrect execution:
- If $i2$ is allowed to execute before $i1$ is determined, register $0$ maybe incorrectly changed!
Control Hazard: Example

- How does the code affect a pipeline processor?
Control Hazard: Problem

- Branch outcome is known only in MEM stage:
  - **Too late**: subsequent instructions already went into the pipeline!

[CS5222 Adv. Comp. Arch. AY1415S2]
Control Hazards: Example

beq $1, $3, 7
and $12, $2, $5
or $13, $2, $6
add $14, $2, $2
lw $4, 5($7)

[CS5222 Adv. Comp. Arch. AY1415S2]
Control Hazards: Stall Pipeline

- Wait for the branch outcome and then fetch the correct instructions
  - Introduces **3 clock cycles delay**
Control Hazard: **Branch Prediction**

- Instead of waiting for the branch outcome
  - Predict (Guess) the outcome right away

- Simple Prediction:
  - Predict all branches are **not taken**
    - Fetch and execute the fall through path

- When outcome is known:
  - **Correct prediction**: No need to do anything
  - **Wrong prediction**: Flush wrong instructions from the pipeline
Branch Prediction: Correct Prediction

<table>
<thead>
<tr>
<th>CC</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>
</table>

```
beq $1, $3, 7
and $12, $2, $5
or $13, $2, $6
add $14, $2, $2

beq not taken: Fall through path already in pipeline!
```
Branch Prediction: **Wrong Prediction**

- **beq $1, $3, 7**
- **and $12, $2, $5**
- **or $13, $2, $6**
- **add $14, $2, $2**
- **lw $4, 5($7)**

- **beq** taken: Need to **flush** subsequent instructions and then fetch from the right location
Summary

- Pipelining is the fundamental principle of modern processor architecture design

- Major topics:
  - Ideal pipeline
  - Hazards:
    - Data dependency
    - Control dependency
  - Hazard detection and resolution