Midterm Review

Hung-Wei Tseng
By loading different programs into memory, your computer can perform different functions.
How does a processor execute instructions?

- Instruction Fetch (IF)
  - Fetch the **instruction** pointed by **PC** from **memory**
- Instruction Decode (ID)
  - Decode the instruction for the desired operation and operands
  - Reading source **register** values
- Execution (EX)
  - ALU instructions: Perform **ALU** operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (MEM) — Read/write **data memory**
- Write Back (WB) — Present ALU result/read value in the target **register**
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4
Instruction Set Architecture (ISA)
Instruction Set Architecture (ISA)

- The **contract** between the hardware and software
- Provide an **abstraction** of the underlying processor
  - Defines the set of operations that a computer/processor can execute
  - Defines the **memory** space that a program can use
- Programs are combinations of these instructions
- The hardware implements these instructions in any way it choose.
  - Directly in hardware circuit. e.g. CPU
  - Software virtual machine. e.g. VirtualPC
  - Simulator/Emulator. e.g. DeSmuME
  - Trained monkey with pen and paper

We’re abstracting a von Neumann machine!
What ISA includes?

- Instructions: what programmers want processors to do?
  - Math: add, subtract, multiply, divide, bitwise operations
  - Control: if, jump, function call
  - Data access: load and store
- Architectural states: the current execution result of a program
  - Registers: a few named data storage that instructions can work on
  - Memory: a much larger data storage array that is available for storing data
  - Program Counter (PC): the number/address of the current instruction
The “abstraction”
Assembly language

- The human-readable representation of “instructions”/“machine language”
- Has a direct mapping between assembly code and instructions
  - Assembly may contain “pseudo instructions” for programmer to use
  - Each pseudo instruction still has its own mapping to a set of real machine instructions

```
add $v0, $a1, $a2
```

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amount</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110</td>
<td>000000</td>
<td>0010</td>
<td>0010</td>
<td>00000</td>
<td>0010</td>
</tr>
<tr>
<td>00101</td>
<td>000000</td>
<td>0010</td>
<td>0010</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
An Overview of MIPS ISA
The abstracted "MIPS" machine

Registers:
- $zero
- $at
- $v0
- $v1
- $a0
- $a1
- $a2
- $a3
- $t0
- $t1
- $t2
- $t3
- $t4
- $t5
- $t6
- $t7
- $s0
- $s1
- $s2
- $s3
- $s4
- $s5
- $s6
- $s7
- $t8
- $t9
- $k0
- $k1
- $gp
- $sp
- $fp
- $ra

Program Counter: 0x0000000000000004

ALU:
- add
- addi
- and
- andi
- ori
- xori
- beq
- blt
- jal
- jr

Memory:
- 0x00000000
- 0x00000008
- 0x00000010
- 0x00000018
- 0x00000020
- 0x00000028
- 0x00000030
- 0x00000038
- 0xFFFFFFC0
- 0xFFFFFFC8
- 0xFFFFFFD0
- 0xFFFFFFD8
- 0xFFFFFFE0
- 0xFFFFFFE8
- 0xFFFFFFF0
- 0xFFFFFFF8

Byte Addressing — every byte of data/instruction has its own address

Program Counter:
- 0x00000000
- 0x00000008
- 0x00000010
- 0x00000018
- 0x00000020
- 0x00000028
- 0x00000030
- 0x00000038

Memory:
- 0x00000000
- 0x00000008
- 0x00000010
- 0x00000018
- 0x00000020
- 0x00000028
- 0x00000030
- 0x00000038
- 0xFFFFFFC0
- 0xFFFFFFC8
- 0xFFFFFFD0
- 0xFFFFFFD8
- 0xFFFFFFE0
- 0xFFFFFFE8
- 0xFFFFFFF0
- 0xFFFFFFF8
MIPS ISA

- All instructions are **32** bits
- **32** 32-bit registers
  - All registers are the same
  - $zero$ is always 0
- **50** opcodes
  - Arithmetic/Logic operations
  - Load/store operations
  - Branch/jump operations
- **3** instruction formats
  - R-type: all operands are registers
  - I-type: one of the operands is an immediate value
  - J-type: non-conditional, non-relative branches

<table>
<thead>
<tr>
<th>Reg. Name</th>
<th>Reg. Num</th>
<th>Usage</th>
<th>Saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporarily</td>
<td>no</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>OS kernel</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
<tr>
<td>Category</td>
<td>Instruction</td>
<td>Usage</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1, $s2, 20</td>
<td>$s1 = $s2 + 20</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>andi</td>
<td>andi $s1, $s2, 20</td>
<td>$s1 = $s2 &amp; 20</td>
</tr>
<tr>
<td></td>
<td>sll</td>
<td>sll $s1, $s2, 10</td>
<td>$s1 = $s2 * 2^10</td>
</tr>
<tr>
<td></td>
<td>srl</td>
<td>srl $s1, $s2, 10</td>
<td>$s1 = $s2 / 2^10</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>lw</td>
<td>lw $s1, 4($s2)</td>
<td>$s1 = mem[$s2+4]</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 4($s2)</td>
<td>mem[$s2+4] = $s1</td>
</tr>
<tr>
<td>Branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if($s1 == $s2), PC = PC + 100</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if($s1 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td>Jump</td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>
Practice

• Translate the C code into assembly:

```c
int A[100];
int sum=0,i;

for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

Assume int is 32 bits
$s0 = &A[0]$
$v0 = sum;
$t0 = i;$
$\text{and } t0, t0, $zero \# \text{let } i = 0$
$\text{addi } t1, $zero, 100 \# \text{temp} = 100$
$\text{lw } t3, 0($s0) \# \text{temp1} = A[i]$
$\text{add } v0, v0, t3 \# \text{sum += temp1}$
$\text{addi } s0, s0, 4 \# \text{addr of A[i+1]}$
$\text{addi } t0, t0, 1 \# i = i+1$
$\text{bne } t1, t0, LOOP \# \text{if } i < 100$

1. Initialization (if i = 0, it must < 100)
2. Load A[i] from memory to register
3. Add the value of A[i] to sum
4. Increase by 1
5. Check if i still < 100

There are many ways to translate the C code. But efficiency may be differ among translations
Number of instructions being executed?

• For the following C code snippet and its corresponding MIPS translation on the right hand side, how many operations will the processor execute in total to complete all loop iterations?

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```mips
and $t0, $t0, $zero #let i = 0
addi $t1, $zero, 100 #temp = 100
LOOP: lw $t3, 0($s0) #temp1 = A[i]
add $v0, $v0, $t3 #sum += temp1
addi $s0, $s0, 4 #addr of A[i+1]
addi $t0, $t0, 1 #i = i+1
bne $t1, $t0, LOOP #if i < 100
```

A. 7  **Static instructions** — how many instructions in the compiled program
B. 402
C. 502  **Dynamic instructions** — how many instructions in the executed program
D. 407
E. 507
Why doesn’t the program work?

• For the current implementation, the program won’t output the expected solution because?
  ① The program cannot reach the callee, sum
  ② The program cannot pass the arguments correctly
  □ The program cannot maintain local variables correctly during the expected scope
  ⑤ The return value of function call will be incorrect
  □ The program cannot return to its caller

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

main: ...... 
...... 
...... 
...... 
add $a0, $v0, $zero 
jal sum
PC1: add $a1, $zero, $v0 
li $a0, 4 
syscall 
...... 
...... 
...... 
...... 
sum: addi $a0, $a0, -1 
addi $v0, $zero, 1 
beq $a0, $zero, return 
jal sum 
add $v0, $v0, $a0 
addi $v0, $v0, 1 
return: jr $ra
What happens when we execute the code

Say we input 2!

main:  
       ... 
       ... 
       ... 
       add $a0, $v0, $zero 
       jal sum 
       li $a0, 4 
       syscall 
       ... 
       ... 
       sum:  
       addi $a0, $a0, -1 
       addi $v0, $zero, 1 
       beq $a0, $zero, return 
       jal sum 
       add $v0, $v0, $a0 
       addi $v0, $v0, 1 
       return: lw $a0, 0($sp) 
                lw $ra, 4($sp) 

PC1:  
      add $a1, $zero, $v0 
      syscall 
      ... 
      ... 
      return: lw $a0, 0($sp) 
               lw $ra, 4($sp) 

Mem:  
      addi $sp, $sp, -8 
      sw $a0, 0($sp) 
      sw $ra, 4($sp) 
      addi $sp, $sp, -8 
      sw $a0, 0($sp) 
      sw $ra, 4($sp) 
      lw $a0, 0($sp) 
      lw $ra, 4($sp) 
      addi $sp, $sp, 8 
      jr $ra 
      lw $a0, 0($sp) 
      lw $ra, 4($sp) 
      addi $sp, $sp, 8 
      jr $ra 

 Registers 
$zero 3 
$at 6 
$v0 3 
$v1 2 
$a0 4 
$a1 3 
$a2 2 
$a3 1 
$t0 0 
$t1 1 
$t2 2 
$t3 3 
$t4 4 
$t5 5 
$t6 6 
$t7 7 
$k0 8 
$k1 9 
$gp 10 
$sp 11 
$fp 12 
$ra 13 

Memory 
PC1 2 
sum+20 1 
sum+20 1 
sum+20 2 
sum+20 3 

Other ISAs
The abstracted x86 machine

CPU

Registers

RAX
RBX
RCX
RDX
RSP
RBP
RSI
RDI
R8
R9
R10
R11
R12
R13
R14
R15
RIP
FLAGS

CS
SS
DS
ES
FS
GS

ADD
SUB
IMUL
AND
OR
XOR

JMP
JE
CALL
RET

MOV

ALU

Memory

0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8

0xFFFFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFFFF8

64-bit

2^{64} Bytes

MOV

18
x86 ISA

- The most widely used ISA
- A poorly-designed ISA
  - It breaks almost every rule of a good ISA
    - variable length of instructions
    - the work of each instruction is not equal
    - makes the hardware become very complex
  - It’s popular != It’s good
- You don’t have to know how to write it, but you need to be able to read them and compare x86 with other ISAs
- Reference
MOV and addressing modes

- MOV instruction can perform load/store as in MIPS
- MOV instruction has many address modes
  - an example of non-uniformity

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $6, %eax</td>
<td>R[eax] = 0x6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl .L0, %eax</td>
<td>R[eax] = .L0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl %ebx, %eax</td>
<td>R[ebx] = R[eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl -4(%ebp), %ebx</td>
<td>R[ebx] = mem[R[ebp]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl (%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>movl -4(%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4-4]</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>movl %ebx, -4(%ebp)</td>
<td>mem[R[ebp]-4] = R[ebx]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl $6, -4(%ebp)</td>
<td>mem[R[ebp]-4] = 0x6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Addressing and accessing the data structure

- Memory allocation
  - Each object/instance of the data structure occupies consecutive memory locations that can accommodate all members in this object/instance
  - The starting address of each object/instance must be aligned with the multiple of their “width”

- Memory access:
  - The base address register points to the beginning of the accessing object/instance
  - The offset points to the member — one of the reason why we have an offset field
Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What’s the output of `printf("%lu\n", sizeof(struct student));`?

A. 20  
B. 28  
C. 32  
D. 36  
E. 40
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td>base+offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
<tr>
<td>ISA type</td>
<td>Reduced Instruction Set Computers (RISC)</td>
<td>Complex Instruction Set Computers (CISC)</td>
</tr>
</tbody>
</table>
How many operations: CISC v.s. RISC

- CISC (Complex Instruction Set Computing)
  - Examples: x86, Motorola 68K
  - Provide many powerful/complex instructions
    - Many: more than 1503 instructions since 2016
    - Powerful/complex: an instruction can perform both ALU and memory operations

- RISC (Reduced Instruction Set Computer)
  - Examples: ARMv8, RISC-V, MIPS (the first RISC instruction, invented by the authors of our textbook)
  - Each instruction only performs simple tasks
  - Easy to decode
Performance
**CPU Performance Equation**

\[
\begin{align*}
\text{Performance} &= \frac{1}{\text{Execution Time}} \\
\text{Execution Time} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \\
ET &= IC \times CPI \times CT
\end{align*}
\]

\[
1\text{GHz} = 10^9\text{Hz} = \frac{1}{10^9}\text{sec per cycle} = 1\text{ ns per cycle}
\]

Frequency (*i.e.*, clock rate)
Execution Time

- The simplest kind of performance
- Shorter execution time means better performance
- Usually measured in seconds

![Diagram of a processor with memory and instructions](image)

**Instructions Program**

**How many of these?**

**How long is it take to execution each of these?**

**Cycles**

- Instruction time

**Seconds**

- Cycle time

**Execution Time**

<table>
<thead>
<tr>
<th>Address</th>
<th>Op Code</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>120007a30</td>
<td>0f00bb27</td>
<td>ldah gp,15(t12)</td>
</tr>
<tr>
<td>120007a34</td>
<td>509cbd23</td>
<td>lda gp,-25520(gp)</td>
</tr>
<tr>
<td>120007a38</td>
<td>00005d24</td>
<td>ldah t1,0(gp)</td>
</tr>
<tr>
<td>120007a3c</td>
<td>0000bd24</td>
<td>ldah t1,0(gp)</td>
</tr>
<tr>
<td>120007a40</td>
<td>2ca422a0</td>
<td>ldl t0,-23508(t1)</td>
</tr>
<tr>
<td>120007a44</td>
<td>130020e4</td>
<td>beq t0,120007a94</td>
</tr>
<tr>
<td>120007a48</td>
<td>00003d24</td>
<td>ldah t0,0(gp)</td>
</tr>
<tr>
<td>120007a4c</td>
<td>2ca4e2b3</td>
<td>stl zero,-23508(t1)</td>
</tr>
<tr>
<td>120007a50</td>
<td>0004ff47</td>
<td>clr v0</td>
</tr>
<tr>
<td>120007a54</td>
<td>28a4e5b3</td>
<td>stl zero,-23512(t4)</td>
</tr>
<tr>
<td>120007a58</td>
<td>20a421a4</td>
<td>ldq t0,-23520(t0)</td>
</tr>
<tr>
<td>120007a5c</td>
<td>0e0020e4</td>
<td>beq t0,120007a98</td>
</tr>
<tr>
<td>120007a60</td>
<td>0204e147</td>
<td>mov t0,t1</td>
</tr>
<tr>
<td>120007a64</td>
<td>0304ff47</td>
<td>clr t2</td>
</tr>
<tr>
<td>120007a68</td>
<td>05000c3</td>
<td>br 120007a80</td>
</tr>
</tbody>
</table>
Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle. If the processor runs at 2 GHz, how long is the execution time?

A. 500000 ns
B. 1000000 ns
C. 2000000 ns
D. 3500000 ns
E. None of the above

\[ ET = IC \times CPI \times CT \]

\[ ET = (5 \times 10^5) \times (20\% \times 6 + 80\% \times 1) \times \frac{1}{2 \times 10^{-9}} \text{ sec} = 5 \times 10^5 \text{ ns} \]
The relative performance between two machines, X and Y. Y is $n$ times faster than X

$$n = \frac{\text{Execution Time}_X}{\text{Execution Time}_Y}$$

The speedup of Y over X

$$\text{Speedup} = \frac{\text{Execution Time}_X}{\text{Execution Time}_Y}$$
Speedup of Y over X

Consider the same program on the following two machines, X and Y. By how much Y is faster than X?

<table>
<thead>
<tr>
<th>Clock Rate</th>
<th>Instructions</th>
<th>Percentage of Type-A Insts.</th>
<th>CPI of Type-A Insts.</th>
<th>Percentage of Type-B Insts.</th>
<th>CPI of Type-B Insts.</th>
<th>Percentage of Type-C Insts.</th>
<th>CPI of Type-C Insts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine X</td>
<td>3 GHz</td>
<td>500000</td>
<td>20%</td>
<td>8</td>
<td>20%</td>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>Machine Y</td>
<td>5 GHz</td>
<td>500000</td>
<td>20%</td>
<td>13</td>
<td>20%</td>
<td>4</td>
<td>60%</td>
</tr>
</tbody>
</table>

A. 0.2

\[ ET_Y = (5 \times 10^9) \times (20\% \times 13 + 20\% \times 4 + 60\% \times 1) \times \frac{1}{5 \times 10^{-9}} \text{sec} = 4 \]

B. 0.25  \[ \text{Speedup} = \frac{\text{Execution Time}_X}{\text{Execution Time}_Y} \]

C. 0.8

\[ = \frac{5}{4} = 1.25 \]

D. 1.25

E. No changes
What makes the difference?
Identify the limiting factor

Why does an Intel Core i7 @ 3.5 GHz usually perform better than an Intel Core i5 @ 3.5 GHz or AMD FX-8350@4GHz?

- A. Because the instruction count of the program are different
- B. Because the clock rate of AMD FX is higher
- C. Because the CPI of Core i7 is better
- D. Because the clock rate of AMD FX is higher and CPI of Core i7 is better
- E. None of the above

\[ ET = IC \times CPI \times CT \]
Programmer’s impact

- By adding the “sort” in the following code snippet, what the programmer changes in the performance equation to achieve better performance?

```cpp
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize*1000; ++c) {
    if (data[c%arraySize] >= INT_MAX/2)
        sum ++;
}
```

A. CPI  
B. IC — we increased IC, suppose to make the  
C. CT performance worse  
D. IC & CPI
Demo — programmer & performance

```c
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

```c
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

**Complexity**

- $O(n^2)$
  - Same

**Instruction Count?**

- Same

**Clock Rate**

- Same

**CPI**

- ???
Use “performance counters” to figure out!

- Modern processors provide performance counters
  - Instruction counts
  - Cache accesses/misses
  - Branch instructions/mis-predictions

- How to get their values?
  - You may use “perf stat” in Linux
  - You may use Instruments —> Time Profiler on a Mac
  - Intel’s vtune — only works on Windows w/ Intel processors
  - You can also create your own functions to obtain counter values
Demo — programmer & performance

A

```c
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

B

```c
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Complexity</th>
<th>Instruction Count?</th>
<th>Clock Rate</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Same</td>
<td>Same</td>
<td>Same</td>
<td>Better</td>
</tr>
<tr>
<td>B</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>Same</td>
<td>Worse</td>
</tr>
</tbody>
</table>
• How many of the following make(s) the performance of one better than the other between version A & version B?

① IC
② CPI
③ CT

A. 0
B. 1
C. 2
D. 3
Programmers can also set the cycle time

https://software.intel.com/sites/default/files/comment/1716807/how-to-change-frequency-on-linux-pub.txt

If the OS is Linux, you can manually control the CPU speed by reading and writing some virtual files in the '/proc' directory. If the 'directory' '/sys/devices/system/cpu/cpu0/cpufreq/max_freq' exists, speed is controllable. If it does not exist, you may need to go to the BIOS and turn on KINI and any other C and P state control and via:

```
2.] What speed is the box set to now?
Do the following:
$ cd /sys/devices/system/cpu
$ cat /cpu0/cpufreq/cpuinfo_max_freq
3153000
$ cat /cpu0/cpufreq/cpuinfo_min_freq
1296000

3.] What speeds can I set to?
Do:
$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_available_frequencies
It will list highest settable to lowest; example from my INN 'Snakeover' DX880 HD6T board, I see:
3153000 3152000 3125000 2926000 293000 260000 2527000 2394000 2201000 2128000 1995000 1862000 1712000 1596000
You can choose from among those numbers to set the 'high water' mark and 'low water' mark for speed. If you set 'h

4.] Show me how to set all to highest settable speed!
Use the following little sh/ksh/bash script:
$ cd /sys/devices/system/cpu; # a virtual directory made visible by device drivers
$newSpeedTop=`awk '{print $1}' /cpu0/cpufreq/scaling_available_frequencies`
$newSpeedLow=$newSpeedTop; # make them the same in this example
$ for c in (/cpu[0-9]*); do
  echo $newSpeedTop > $(cat /cpufreq/scaling_max_freq
  echo $newSpeedLow > $(cat /cpufreq/scaling_min_freq
  done

5.] How do I return to the default - i.e. allow machine to vary from highest to lowest?
Edit line # 3 of the script above, and re-run it. Change the lines:
$newSpeedLow=$newSpeedTop; # make them the same in this example

Subject: setting CPU speed on running linux system
How programmer affects performance?

- Performance equation consists of the following three factors:
  - IC
  - CPI
  - CT

How many can a programmer affect?

A. 0
B. 1
C. 2
D. 3
• Which of the following programming language needs to highest instruction count to print “Hello, world!” on screen?

A. C
B. C++
C. Java
D. Perl
E. Python
### Programming languages

- How many instructions are there in “Hello, world!”

<table>
<thead>
<tr>
<th>Language</th>
<th>Instruction count</th>
<th>LOC</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>600k</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>C++</td>
<td>3M</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Java</td>
<td>~210M</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Perl</td>
<td>10M</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Python</td>
<td>~30M</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Recap: How my “C code” becomes a “program”
Recap: How my “Java code” becomes a “program”

One Time Cost!
Everytime when we run it!
Recap: How my “Python code” becomes a “program”

Everytime when we run it!
How programming languages affect performance

• Performance equation consists of the following three factors
  ① IC
  ② CPI
  ③ CT

How many can the **programming language** affect?

A. 0
B. 1
C. 2
D. 3
How compilers affect performance

- Performance equation consists of the following three factors:
  ① IC
  ② CPI
  ③ CT

How many can the compiler affect?

A. 0  
B. 1  
C. 2  
D. 3  

Correct answer: C. 2
Revisited the demo with compiler optimizations!

- gcc has different optimization levels.
  - -O0 — no optimizations
  - -O3 — typically the best-performing optimization

```c
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

```c
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```
Demo revisited — compiler optimization

- Compiler can reduce the instruction count, change CPI — with “limited scope”
- Compiler CANNOT help improving “crummy” source code

```
if (option)
    std::sort(data, data + arraySize);
```

```
for (unsigned c = 0; c < arraySize*1000; ++c) {
    if (data[c%arraySize] >= INT_MAX/2)
        sum ++;
}
```

Compiler can never add this — only the programmer can!
How about “computational complexity”

- Algorithm complexity provides a good estimate on the performance if —
  - Every instruction takes exactly the same amount of time
  - Every operation takes exactly the same amount of instructions

These are unlikely to be true
Summary of CPU Performance Equation

\[ Performance = \frac{1}{\text{Execution Time}} \]
\[ \text{Execution Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \]

\[ ET = IC \times CPI \times CT \]

- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, programmer
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, programmer
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, programmer
Amdahl’s Law
Amdahl’s Law

\[ \text{Speedup}_{\text{enhanced}}(f, s) = \frac{1}{(1-f) + \frac{f}{s}} \]

- \( f \) — The fraction of time in the original program
- \( s \) — The speedup we can achieve on \( f \)

Execution Time_{baseline} = 1

Execution Time_{enhanced} = (1-f) + f/s

\[ \text{Speedup}_{\text{enhanced}} = \frac{\text{Execution Time}_{\text{baseline}}}{\text{Execution Time}_{\text{enhanced}}} = \frac{1}{(1-f) + \frac{f}{s}} \]
Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time. By how much can we speed up the map loading process?

A. ~7x
B. ~10x
C. ~17x
D. ~29x
E. ~100x

\[
\text{Speedup}_{\text{enhanced}}(95\%,100) = \frac{1}{(1 - 95\%) + \frac{95\%}{100}} = 16.81 \times
\]
We can apply Amdahl’s law for multiple optimizations

These optimizations must be dis-joint!

- If optimization #1 and optimization #2 are dis-joint:

\[
\text{Speedup}\_\text{enhanced}(f_{\text{Opt1}}, f_{\text{Opt2}}, s_{\text{Opt1}}, s_{\text{Opt2}}) = \frac{1}{(1 - f_{\text{Opt1}} - f_{\text{Opt2}}) + \frac{f_{\text{Opt1}}}{s_{\text{Opt1}}} + \frac{f_{\text{Opt2}}}{s_{\text{Opt2}}}}
\]

- If optimization #1 and optimization #2 are not dis-joint:

\[
\text{Speedup}\_\text{enhanced}(f_{\text{OnlyOpt1}}, f_{\text{OnlyOpt2}}, f_{\text{BothOpt1Opt2}}, s_{\text{OnlyOpt1}}, s_{\text{OnlyOpt2}}, s_{\text{BothOpt1Opt2}}) = \frac{1}{(1 - f_{\text{OnlyOpt1}} - f_{\text{OnlyOpt2}} - f_{\text{BothOpt1Opt2}}) + \frac{f_{\text{BothOpt1Opt2}}}{s_{\text{BothOpt1Opt2}}} + \frac{f_{\text{OnlyOpt1}}}{s_{\text{OnlyOpt1}}} + \frac{f_{\text{OnlyOpt2}}}{s_{\text{OnlyOpt2}}}}
\]
Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?

A. ~7x  
B. ~10x  
C. ~17x  
D. ~29x  
E. ~100x

\[
\text{Speedup}_{\text{enhanced}}(95\%, 5\%, 100, 2) = \frac{1}{(1 - 95\% - 5\%) + \frac{95\%}{100} + \frac{5\%}{2}} = 28.98 \times
\]
Amdahl’s Law Corollary #1

- The maximum speedup is bounded by

\[
\text{Speedup}_{\text{max}}(f, \infty) = \frac{1}{(1 - f) + \frac{f}{\infty}}
\]

\[
\text{Speedup}_{\text{max}}(f, \infty) = \frac{1}{1 - f}
\]
With the latest flash memory technologies, the system spends 16% of time on accessing the flash, and the software overhead is now 84%. If we want to adopt a new memory technology to replace flash to achieve 2x speedup on loading maps, how much faster the new technology needs to be?

A. ~5x  
B. ~10x  
C. ~20x  
D. ~100x  
E. None of the above

\[ \text{Speedup}_{max}(16\%, \infty) = \frac{1}{(1 - 16\%)} = 1.19 \]

2x is not possible
Corollary #1 on Multiple Optimizations

- If we can pick just one thing to work on/optimize

\[
\begin{align*}
\text{Speedup}_{\text{max}}(f_1, \infty) &= \frac{1}{(1 - f_1)} \\
\text{Speedup}_{\text{max}}(f_2, \infty) &= \frac{1}{(1 - f_2)} \\
\text{Speedup}_{\text{max}}(f_3, \infty) &= \frac{1}{(1 - f_3)} \\
\text{Speedup}_{\text{max}}(f_4, \infty) &= \frac{1}{(1 - f_4)}
\end{align*}
\]

The biggest \( f_x \) would lead to the largest \( \text{Speedup}_{\text{max}} \)!
Corollary #2 — make the common case fast!

• When f is small, optimizations will have little effect.
• Common == most time consuming not necessarily the most frequent
• The uncommon case doesn’t make much difference
• The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.
Identify the most time consuming part

- Compile your program with -pg flag
- Run the program
  - It will generate a gmon.out
  - gprof your_program gmon.out > your_program.prof
- It will give you the profiled result in your_program.prof
With optimization, the common becomes uncommon.

An uncommon case will (hopefully) become the new common case.

Now you have a new target for optimization.

— You have to revisit “Amdahl’s Law” every time you applied some optimization.
Don't hurt non-common part too mach

• If the program spend 90% in A, 10% in B. Assume that an optimization can accelerate A by 9x, by hurts B by 10x...

• Assume the original execution time is $T$. The new execution time

$$ET_{new} = \frac{ET_{old} \times 90\%}{9} + ET_{old} \times 10\% \times 10$$

$$ET_{new} = 1.1 \times ET_{old}$$

$$\text{Speedup} = \frac{ET_{old}}{ET_{new}} = \frac{ET_{old}}{1.1 \times ET_{old}} = 0.91 \times \quad \text{......slowdown!}$$

You may not use Amdahl’s Law for this case as Amdahl’s Law does NOT
(1) consider overhead
(2) bound to slowdown
Corollary #3, Corollary #4 & Corollary #5

\[
\text{Speedup}_{\text{parallel}}(f_{\text{parallelizable}}, \infty) = \frac{1}{(1 - f_{\text{parallelizable}}) + \frac{f_{\text{parallelizable}}}{\infty}}
\]

\[
\text{Speedup}_{\text{parallel}}(f_{\text{parallelizable}}, \infty) = \frac{1}{(1 - f_{\text{parallelizable}})}
\]

- Single-core performance still matters — it will eventually dominate the performance
- Finding more “parallelizable” parts is also important
- If we can build a processor with unlimited parallelism — the complexity doesn’t matter as long as the algorithm can utilize all parallelism — that’s why bitonic sort works!
“Fair” Comparisons
TFLOPS (Tera FLoating-point Operations Per Second)

- TFLOPS does not include instruction count!
  - Cannot compare different ISA/compiler
  - Different CPI of applications, for example, I/O bound or computation bound
  - If new architecture has more IC but also lower CPI?

<table>
<thead>
<tr>
<th></th>
<th>TFLOPS</th>
<th>clock rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBOX One X</td>
<td>6</td>
<td>1.75 GHz</td>
</tr>
<tr>
<td>PS4 Pro</td>
<td>4</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>GeForce GTX 1080</td>
<td>8.228</td>
<td>3.5 GHz</td>
</tr>
</tbody>
</table>
Is TFLOPS (Tera FLoating-point Operations Per Second) a good metric?

\[
TFLOPS = \frac{\text{\# of floating point instructions} \times 10^{-12}}{\text{Execution Time}}
\]

\[
= \frac{IC \times \% \text{ of floating point instructions} \times 10^{-12}}{IC \times CPI \times CT}
\]

\[
= \frac{\% \text{ of floating point instructions} \times 10^{-12}}{CPI \times CT}
\]

IC is gone!

- Cannot compare different ISA/compiler
  - What if the compiler can generate code with fewer instructions?
  - What if new architecture has more IC but also lower CPI?
- Does not make sense if the application is not floating point intensive
Choose the right metric — Latency v.s. Throughput/Bandwidth
Latency v.s. Bandwidth/Throughput

- Latency — the amount of time to finish an operation
  - access time
  - response time
- Throughput — the amount of work can be done within a given period of time
  - bandwidth (MB/Sec, GB/Sec, Mbps, Gbps)
  - IOPs
  - MFLOPs
Latency/Delay v.s. Throughput

**Toyota Prius**
- 100 miles (161 km) from UCSD
- 75 MPH on highway!
- Max load: 374 kg = 2,770 hard drives (2TB per drive)

**100 Gb Network**
- 100 miles (161 km) from UCSD
- Lightspeed! — $3 \times 10^8$ m/sec
- Max load: 4 lanes operating at 25GHz

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Toyota Prius</th>
<th>100 Gb Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>290 GB/sec</td>
<td>100 Gb/s or 12.5 GB/sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency</th>
<th>Toyota Prius</th>
<th>100 Gb Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.5 hours</td>
<td>2 Peta-byte over 167772 seconds = 1.94 Days</td>
</tr>
</tbody>
</table>

You see nothing in the first 3.5 hours

You can start watching the movie as soon as you get a frame!
Processor Design
The “life” of a dynamic instruction?

- Instruction Fetch (IF)
  - Fetch the instruction pointed by PC from memory
- Instruction Decode (ID)
  - Decode the instruction for the desired operation and operands
  - Reading source register values
- Execution (EX)
  - ALU instructions: Perform ALU operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (MEM) — Read/write data memory
- Write Back (WB) — Present ALU result/read value in the target register
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4
Performance of a single-cycle processor

How many of the following statements about a single-cycle processor is correct?

① The CPI of a single-cycle processor is always 1
② If the single-cycle implements MIPS ISA, the memory instruction will determine the cycle time
③ Hardware elements are mostly idle during a cycle
④ We can always reduce the cycle time of a single-cycle processor by supporting fewer instructions  — Only if this instruction is the most time-critical one

A. 0
B. 1
C. 2
D. 3
E. 4
Pipelining
Pipelining

- Different parts of the processor works on different instructions simultaneously
- A clock signal controls and synchronize the beginning and the end of each part of the work
- A pipeline register between different parts of the processor to keep intermediate results necessary for the upcoming work
After this point, we are completing an instruction each cycle!

Cycles
\[
\begin{array}{c}
\text{Instruction} \\
= 1
\end{array}
\]

All hardware parts are in use.

Inst1 Inst2 Inst3 Inst4 Inst5 Inst6 Inst7 Inst8 Inst9 Inst10 Inst11
Cycle time of a pipeline processor

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance.
- Ideally, all paths are about the same length.
Performance of pipelining

- The following diagram shows the latency in each part of a single-cycle processor:

If we can make each part as a “pipeline stage”, what’s the maximum speedup we can achieve? (choose the closest one)

A. 3.33
B. 4
C. 5
D. 6.67
E. 10

\[
\text{Speedup} = \frac{\text{\# of insts} \times 1 \times 10\text{ns}}{\text{\# of insts} \times 1 \times 3\text{ns}} = 3.33
\]
How many of the following descriptions about pipelining is correct?

① You can always divide stages into short stages with latches to improve performance
   - Only if this stage is the most time-critical one
② Pipeline registers incur overhead for each pipeline stage
③ The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
   - You have pipeline registers and each stage needs to be equally long
④ The throughput of a pipeline processor is usually better than a single-cycle processor

A. 0
B. 1
C. 2
D. 3
E. 4
Performance of pipelining

• The following diagram shows the latency in each part of a single-cycle processor:

If we can make each part as a “pipeline stage”, what’s the maximum speedup we can achieve? (choose the closest one)

A. 3.33
B. 4
C. 5
D. 6.67
E. 10

\[
\text{Speedup} = \frac{\#_{\text{of\_insts}} \times 1 \times 10\text{ns}}{\#_{\text{of\_insts}} \times 1 \times 3\text{ns}} = 3.33
\]

— The cycle time is 3ns
— Each instruction now takes “15ns” to leave the pipeline!
— But even “hello, world” has at lease 500K instructions, who cares?
The 5-stage MIPS Pipeline Processor
Single-cycle processor
Pipelined processor

Instruction Memory
Instruction Address [31:0] Instruction [31:0]

Register File
Read Reg 1 Read Data 1
Read Reg 2
Read Data 2
Write Reg
Write Data

Control
RegDst
MemToReg
Branch
MemRead
MemoryWrite
ALUOp
ALUSrc
RegWrite

ALU
Is zero?
ALU Ctrl.

Data Memory
Data Address [31:0]
Write Data[31:0]

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

PC
Instruction
Memory
Instruction
[25-21]
Instruction
[20-16]
Instruction
[15-11]
Instruction
[15-0]
Instruction
[5-0]

m x

m x

m x

m x
Pipelined processor

Instruction Memory

Instruction Address [31:0] Instruction [31:0]

Instruction Memory Read Reg 1 Read Data 1


Register File Read Reg 1 Read Data 1 Read Reg 2 Read Data 2

Write Reg Write Data

sign-extend

ALU

4

control

RegDst MemToReg Branch MemoryRead MemoryWrite ALUOp ALUSrc RegWrite


Instruction

IF/ID

ID/EX

EX/MEM

MEM/WB

ALUSrc

MemoryWrite

Branch

MemToReg

RegWrite

Control

Instruction [31-0]

sign-extend

ALU

Control

Instruction [5-0]

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Tips of drawing a pipeline diagram

• Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order — only valid if it’s single-issue, MIPS 5-stage pipeline

• An instruction can enter the next pipeline stage in the next cycle if
  • No other instruction is occupying the next stage
  • This instruction has completed its own work in the current stage
  • The next stage has all its inputs ready and it can retrieve those inputs

• Fetch a new instruction only if
  • We know the next PC to fetch
  • We can predict the next PC
  • Flush an instruction if the branch resolution says it’s mis-predicted.
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
xor $13,$14,$15
and $16,$17,$18
add $19,$20,$21
sub $22,$23,$24
lw  $25, 4($26)
sw $27, 0($28)

After this point, we are completing an instruction each cycle!
Both version A and B produces the same output. Without compiler optimization, which version of code would have significantly better performance?

A. Version A  
B. Version B  
C. They are about the same (less than 10% difference) — Because we have pipelined instructions, the CPI of one instruction doesn’t matter as long as we can keep the pipeline busy
Can we get them right?

Given a simple pipelined MIPS processor that we discussed so far, how many of the following code snippets can be executed with expected outcome?

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b</td>
<td>lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $9, $1, $10</td>
<td>sub $6, $7, $8</td>
</tr>
<tr>
<td>d</td>
<td>sub $9, $10, $11</td>
<td>sub $9, $10, $11</td>
<td>sub $9, $10, $11</td>
<td>sub $9, $10, $11</td>
</tr>
<tr>
<td>e</td>
<td>sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Draw the pipeline diagrams

The desired value of $x_1$ is not ready yet

Both instructions need $x_1$

Doesn’t know what to fetch at this moment

90
Can we get them right?

• Given a simple pipelined MIPS processor that we discussed so far, how many of the following code snippets can be executed with expected outcome?

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b</td>
<td>lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
</tr>
<tr>
<td>d</td>
<td>sub $9, $10, $11</td>
<td>bne $0, $7, L</td>
<td>sub $9, $10, $11</td>
<td>sub $9, $10, $11</td>
</tr>
<tr>
<td>e</td>
<td>sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

A. 0
B. 1
C. 2
D. 3
E. 4

b cannot get x1 produced by a before WB
both a and d are accessing x1 at the 5th cycle
We don’t know if d & e will be executed or not until c finishes
Pipeline hazards
Three pipeline hazards

- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that’s not yet generated or propagated when the instruction needs that
Given a simple pipelined MIPS processor that we discussed so far, how many of the following code snippets can be executed with expected outcome?

Can we get them right?

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b</td>
<td>lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c</td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $9, $1, $10</td>
<td>sub $9, $10, $11</td>
</tr>
<tr>
<td>d</td>
<td>sub $9,$10,$11</td>
<td>sub $9, $1, $10</td>
<td>sub $11, 0($12)</td>
<td>sub $11, 0($12)</td>
</tr>
<tr>
<td>e</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

- A. 0  
  - b cannot get x1 produced by a before WB

- B. 1  
  - both a and d are accessing x1 at the 5th cycle
  - We don’t know if d & e will be executed or not until c finishes

- C. 2  
  - Data Hazard

- D. 3  
  - Structural Hazard

- E. 4  
  - Control Hazard
Structural Hazards
Dealing with the conflicts between ID/WB

- The same register cannot be read/written at the same cycle
- Solution: stall the later instruction, allowing the write to present the change in the register and the later can get the desired value
- Drawback: slow

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9, $1, $10
sw  $11, 0($12)
```
Dealing with the conflicts between ID/WB

• The same register cannot be read/written at the same cycle
• Better solution: write early, read late
  • Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  • This leaves enough time for outputs to settle for reads
  • The revised register file is the default one from now!

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9, $1, $10
sw  $11, 0($12)
```
What pair of instructions will be problematic if we allow ALU instructions to skip the “MEM” stage?

A. a & b
B. a & c
C. b & e
D. c & e
E. None

Structural Hazards

IF | ID | EX | MEM | WB
---|----|----|------|----
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB
IF | ID | EX | MEM | WB

a: lw $1, 0($2)  
b: add $3, $4, $5  
c: sub $6, $7, $8  
d: sub $9, $10, $11  
e: sw $1, 0($12)
Structural Hazards

• Stall can address the issue — but slow
• Compiler optimization — what if the hardware design changes?
• Improve the pipeline unit design to allow parallel execution
Data hazards
Data hazards

• An instruction currently in the pipeline cannot receive the "logically" correct value for execution

• Data dependencies
  • The output of an instruction is the input of a later instruction
  • May result in data hazard if the later instruction that consumes the result is still in the pipeline
How many dependencies do we have?

- How many pairs of data dependences are there in the following MIPS instructions?

```
lw   $t0,0($a0)
add  $t0,$t0, $t2
sw   $t0,0($a0)
addi $a0,$a0, 4
bne  $a0,$t1, LOOP
```

A. 1
B. 2
C. 3
D. 4
E. 5
Solution 1: Let’s try “stall” again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.
How many pairs of instructions in the following MIPS instructions will result in data hazards/stalls in a basic 5-stage MIPS pipeline?

lw $t0,0($a0)
add $t0,$t0, $t2
sw $t0,0($a0)
addi $a0,$a0, 4
bne $a0,$t1, LOOP

A. 1
B. 2
C. 3
D. 4
E. 5
Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions
- In our five stage pipeline — if the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file
Do we still have to stall?

• How many pairs of instructions in the following MIPS instructions will result in data hazards/stalls in a basic 5-stage MIPS pipeline with “full” data forwarding?

```
lw   $t0, 0($a0)  IF  ID  EX  MEM  WB
add  $t0, $t0, $t2  IF  ID  ID  EX  MEM  WB
sw   $t0, 0($a0)  IF  IF  ID  EX  MEM  WB
addi $a0, $a0, 4  IF  ID  EX  MEM  WB
bne  $a0, $t1, LOOP
```

A. 0
B. 1
C. 2
D. 3
E. 4
Pipelined processor with Data Forwarding

- Instruction Memory
  - Instruction Address [31:0]
  - Instruction [31:0]
- Register File
  - Read Reg 1
  - Read Data 1
  - Read Reg 2
  - Read Data 2
  - Write Reg
  - Write Data
- Control
  - Instruction [31-26]
  - Instruction [25-21]
  - Instruction [20-16]
  - Instruction [15-11]
  - Instruction [15-0]
  - Instruction [5-0]
- Hazard Detection
- Data Memory
  - Data [31:0]
  - Write
  - Data [31:0]
- ALU
  - ALU Op
  - Memory Write
  - Memory Read
  - Forwarding
  - Forwarding
  - ALU Ctrl.
Control Hazards
The impact of control hazards

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. What’s the average CPI if we execute this program on the 5-stage MIPS pipeline?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A. 1</td>
<td>add x1, x2, x3</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>B. 1.2</td>
<td>ld x4, 0(x5)</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>C. 1.4</td>
<td>bne x0, x7, L</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>D. 1.6</td>
<td>add x0, x0, x0</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>E. 1.8</td>
<td>add x0, x0, x0</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

\[ C_{avg} = 1 + 0.20 \times 2 = 1.4 \]
Why can’t we proceed without stalls/no-ops?

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor

  ① The target address when branch is taken is not available for instruction fetch stage of the next cycle
  ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
  ③ The branch outcome cannot be decided until the comparison result of ALU is not out
  ④ The next instruction needs the branch instruction to write back its result

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Recap: Static Not-Taken Predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

\[ 1 + 75\% \times (20\% \times 2) = 1.3 \]

For branch Y, almost 0%
For branch X, only 50%
A basic dynamic branch predictor

![Diagram of a branch predictor](image)

**Branch Target Buffer**

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>
Recap: 2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on actual outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>

Predict Taken
Recap: 2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```c
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
    a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%  
E. ~75%  

For branch Y, almost 100%,  
For branch X, only 50%
Two-level global predictor

What's the overall branch prediction (include both branches) accuracy for this nested for loop?

\[
i = 0;
\text{do } \{ 
    \text{if( } i \% 2 \neq 0) \text{ // Branch X, taken if } i \% 2 = 0 \\
    \ a[i] *= 2; \\
    \ a[i] += i;
\} \text{ while ( ++i < 100) // Branch Y}
\]

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

This pattern repeats all the time!

For branch Y, almost 100%, For branch X, only 50%
Global history (GH) predictor

Global History Register

Branch PC | Target PC
--- | ---
0x400048 | 0x400032
0x400080 | 0x400068
0x401080 | 0x401100
0x4000F8 | 0x400100

Predict Taken = (NT, T, NT, NT)

States associated with history

00
01
10
11
10
11
10
00
00
00
00
11
10
01
00
Performance of GH predictor

i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y

Near perfect after this
Better predictor?

Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?

A. 0
B. 1
C. 2
D. 3
E. 4

1. `i = 0; do { if (i % 10 != 0) a[i] *= 2; a[i] += i; } while (++i < 100);`
2. `i = 0; do { a[i] += i; } while (++i < 100);`
3. `i = 0; do { j = 0; do { sum += A[i*2+j]; } while (++j < 2); } while (++i < 100);`
4. `i = 0; do { if (rand() %2 == 0) a[i] *= 2; a[i] += i; } while (++i < 100)`

L could be better
Putting it altogether — Performance programming on modern pipeline processors
Four implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

C

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4
    }; while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

D

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4
    };
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
If we can unroll the loop...

Consider the following dynamic instructions:

1. \texttt{lw} \hspace{0.5cm} \$t0,0($a0)
2. \texttt{add} \hspace{0.5cm} \$t0,$t0, $t2
3. \texttt{sw} \hspace{0.5cm} \$t0,0($a0)
4. \texttt{lw} \hspace{0.5cm} \$t3,4($a0)
5. \texttt{add} \hspace{0.5cm} \$t3,$t3, $t2
6. \texttt{sw} \hspace{0.5cm} \$t3,4($a0)
7. \texttt{addi} \hspace{0.5cm} $a0,$a0, 8
8. \texttt{bne} \hspace{0.5cm} $a0,$t1, LOOP

Which of the following pair can we reorder without affecting the correctness if the loop is unrolled twice?

A. (2) and (3)
B. (3) and (4)
C. (5) and (6)
D. (6) and (7)
E. (7) and (8)
If we can unroll the loop...

(1) lw $t0,0($a0)  
(2) add $t0,$t0, $t2  
(3) sw $t0,0($a0)  
(4) lw $t3,4($a0)  
(5) add $t3,$t3, $t2  
(6) sw $t3,4($a0)  
(7) addi $a0,$a0, 8  
(8) bne $a0,$t1, LOOP
Why is B better than A?

**A**

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

**B**

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

**4*n instructions**

```c
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
bne x1, x0, LOOP
```  

**13*(n/4) = 3.25*n instructions**

- Only one branch for four iterations in A
- 125
Loop unrolling eliminates all branches!

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
For midterm

- No cheating allowed
- Make sure you have stable internet
- Make sure you have a webcam
- Make sure you show up 5 minute before the exam starts — joining by 1:55p
- We will lock after 1:55p
- Format
  - Multiple choice question * 20
  - Free answer / filling blanks * 4
Sample Midterm
• Assume that we have an application composed with a total of 5000000000 instructions, in which 20% of them are “Type-A” instructions with an average CPI of 8 cycles, 20% of them are “Type-B” instructions with an average CPI of 4 cycles and the rest instructions are “Type-C” instructions with average CPI of 1 cycle. If the processor runs at 3 GHz, how long is the execution time?

A. 3.67 sec
B. 5 sec
C. 6.67 sec
D. 15 sec
E. 45 sec
Identify the limiting factor

Why does an Intel Core i7 @ 3.5 GHz usually perform better than an Intel Core i5 @ 3.5 GHz or AMD FX-8350@4GHz?

A. Because the instruction count of the program are different
B. Because the clock rate of AMD FX is higher
C. Because the CPI of Core i7 is better
D. Because the clock rate of AMD FX is higher and CPI of Core i7 is better
E. None of the above
Recap: Speedup

• Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle when using a 2GHz processor.
  • If we double the CPU clock rate to 4GHz that helps to accelerate all instructions by 2x except that load/store instruction cannot be improved — their CPI will become 12 cycles. What’s the performance improvement after this change?
    • No change
    • 1.25
    • 1.5
    • 2
    • None of the above
Practicing Amdahl’s Law (2)

• Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?

A. ~7x
B. ~10x
C. ~17x
D. ~29x
E. ~100x
Speedup further!

• With the latest flash memory technologies, the system spends 16% of time on accessing the flash, and the software overhead is now 84%. If we want to adopt a new memory technology to replace flash to achieve 2x speedup on loading maps, how much faster the new technology needs to be?

A. ~5x
B. ~10x
C. ~20x
D. ~100x
E. None of the above
Amdahl’s Law on Multicore Architectures

Regarding Amdahl’s Law on multicore architectures, how many of the following statements is/are correct?

① If we have unlimited parallelism, the performance of each parallel piece does not matter as long as the performance slowdown in each piece is bounded

② With unlimited amount of parallel hardware units, single-core performance does not matter anymore

③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts

④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
By adding the “sort” in the following code snippet, what the programmer changes in the performance equation to achieve better performance?

```cpp
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize*1000; ++c) {
    if (data[c%arraySize] >= INT_MAX/2)
        sum ++;
}
```

A. CPI  
B. IC  
C. CT  
D. IC & CPI
Comparing x86 and MIPS ISAs, how many of the following statements is/are “generally” correct?

① x86 provides more instructions than MIPS
② x86 usually needs more instructions to express the same program
③ An x86 instruction may access memory for 3 times
④ An x86 instruction may be shorter than a MIPS instruction
⑤ An x86 instruction may be longer than a MIPS instruction

A. 1
B. 2
C. 3
D. 4
E. 5
How many dependencies do we have?

- How many pairs of data dependences are there in the following MIPS instructions?

```
lw    $t0,0($a0)
add   $t0,$t0, $t2
sw    $t0,0($a0)
addi  $a0,$a0, 4
bne   $a0,$t1, LOOP
```

A. 1
B. 2
C. 3
D. 4
E. 5
Why the performance is better when option is not “0”

1. The amount of dynamic instructions needs to execute is a lot smaller
2. The amount of branch instructions to execute is smaller
3. The amount of branch mis-predictions is smaller
4. The amount of data accesses is smaller

A. 0
   if (option)
       std::sort(data, data + arraySize);
B. 1
   for (unsigned i = 0; i < 100000; ++i) {
C. 2
       int threshold = std::rand();
       for (unsigned i = 0; i < arraySize; ++i) {
           if (data[i] >= threshold)
               sum ++;
       }
D. 3
E. 4
}
Why is B better than A?

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  ① B has lower dynamic instruction count than A
  ② B has significantly lower branch mis-prediction rate than A
  ③ B has significantly fewer branch instructions than A
  ④ B can incur fewer data hazards

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while (x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```
Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  1. C has lower dynamic instruction count than B
  2. C has significantly lower branch mis-prediction rate than B
  3. C has significantly fewer branch instructions than B
  4. C can incur fewer data hazards

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```
Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  ① D has lower dynamic instruction count than C
  ② D has significantly lower branch mis-prediction rate than C
  ③ D has significantly fewer branch instructions than C
  ④ D can incur fewer data hazards than C

A. 0
B. 1
C. 2
D. 3
E. 4
The result of `sizeof(struct student)`

- Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What's the output of

`printf("%lu\n", sizeof(struct student))`?

A. 20
B. 28
C. 32
D. 36
E. 40
What address should I store to?

- We have the following C code:
  ```c
  int i, A[512];
  A[5] = 0; // sw  $zero, 0($t0)
  ```

  if the `sw` instruction implements `A[5]=0` and `&A[0]` is 32'd1000. What value should $t0$ contain in decimal?
  
  A. 1020  
  B. 1023  
  C. 1005  
  D. 1005  
  E. 1000
Why doesn’t the program work?

• For the current implementation, the program won’t output the expected solution because?
  ① The program cannot reach the callee, sum
  ② The program cannot pass the arguments correctly
  ③ The program cannot maintain local variables correctly during the expected scope
  ④ The return value of function call will be incorrect
  ⑤ The program cannot return to its caller

A. 0
B. 1
C. 2
D. 3
E. 4
Performance of a single-cycle processor

• How many of the following statements about a single-cycle processor is correct?

  ① The CPI of a single-cycle processor is always 1
  ② If the single-cycle implements MIPS ISA, the memory instruction will determine the cycle time
  ③ Hardware elements are mostly idle during a cycle
  ④ We can always reduce the cycle time of a single-cycle processor by supporting fewer instructions

A. 0
B. 1
C. 2
D. 3
E. 4
Limitations of pipelining

- How many of the following descriptions about pipelining is correct?
  1. You can always divide stages into short stages with latches to improve performance
  2. Pipeline registers incur overhead for each pipeline stage
  3. The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
  4. The throughput of a pipeline processor is usually better than a single-cycle processor

A. 0
B. 1
C. 2
D. 3
E. 4
• Both version A and B produces the same output. Without compiler optimization, which version of code would have significantly better performance?
  A. Version A
  B. Version B
  C. They are about the same (less than 10% difference)
Both version A and B produce the same output. Without compiler optimization, which version of code would have significantly better performance?

A. Version A
B. Version B
C. They are about the same (less than 10% difference)
Amdahl’s Law for multicore processors

- Assume that we have an application, in which 50% of the application can be fully parallelized with 2 processors. Assuming 80% of the parallelized part can be further parallelized with 4 processors, what’s the speed up of the application running on a 4-core processor?
Example

• Draw the pipeline execution diagram
  LOOP: lw $t1, 0($a0)
        lw $a0, 0($t1)
        addi $a1, $a1, -1
        bne $a1, $zero, LOOP
        add $v0, $zero, $a1

• Assume that we have no data forwarding and no branch prediction
• Assume that we have full data forwarding and always predict taken.
• Assume that we split the MEM stage into M1 and M2, and the memory data is ready after M2. The processor still has full forwarding and always predict taken
Dynamic branch prediction

- Consider the following code, which branch predictor (2-bit local, 2-bit global history with 4-bit GHR) works the best?

```c
for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
        sum+=a[i][j]
    }
}
```
Other things to think ...

• What is performance equation? What affects each term in the equation?
• What is Amdahl’s law? What’s the implication of Amdahl’s law?
• What is instruction set architecture?
• What is process of generating a binary from C source files?
• What are the architectural states of a program?
• What are the differences between MIPS and x86?
• What’s the functionality of gcc, gprof, perf? When should we use them?
Other things to think ...

- Why TFLOPS (Tera FLoating-Point Operations Per Second) is not a proper performance metric in most cases?
- What are the drawbacks of a single cycle processor?
- What are the advantages of pipelining?
- What is clocking methodology?
- What are the basic steps of executing an instruction?
- What are pipeline hazards? Please explain and give examples
- How to solve the pipeline hazards?
- Code optimization demoed in class
Announcements

• Reading quiz — due next Monday
• Resources
  • Ask questions — piazza
  • Reading quizzes, turning in assignments — Canvas
  • Slides, schedule, assignment questions — Check our website
  • Video archive — Prof. Usagi’s Youtube channel