Midterm Review

Hung-Wei Tseng
Von Neumann architecture

CPU is a dominant factor of performance since we heavily rely on it to execute programs.

By pointing “PC” to different part of your memory, we can perform different functions!
How CPU handle instructions

- **Instruction fetch**: where?
  - program counter & instruction memory
- **Decode**:
  - What’s the instruction?
  - Where are the operands?
- **Execute**
- **Memory access**
  - Where is my data?
- **Write back**
  - Where to put the result
- **Determine the next PC**
Instruction Set Architecture (ISA)

- The **contract** between the hardware and software
- Defines the set of operations that a computer/processor can execute
- Programs are combinations of these instructions
  - Abstraction to programmers/compilers
- 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
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
Instruction Set Architecture (ISA)
From C/C++ to Machine Code

- Intermediate Representation
  - compiler frontend (e.g. gcc/llvm)

- Object
  - compiler backend assembler/optimizer

- Executable
  - linker (e.g. ld)
  - Library

- Machine code/binary

- OS loader

One time cost
From Java to Machine Code

Java byte-code

Intermediate Representation

compiler frontend (e.g. javac)

compiler backend

.JClass

one time cost

JVM

Machine code

.JClass
From Script Languages to Machine Code

Python

Perl

Intermediate Representation

interpreter (python, perl)

compiler

binary

machine code

compiler

executable

binary

runtime
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>name</th>
<th>number</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>temporaries</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>
</tbody>
</table>
"Abstracted" MIPS Architecture

CPU

Program Counter
0x00000004

Registers

Registers

Add
Sub
And
Or
Bne
Beq
Jal

lw
sw

ALU

Memory

0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C
0xFFFFFFE0
0xFFFFFFE4
0xFFFFFFE8
0xFFFFFFEC
0xFFFFFFF0
0xFFFFFFF4
0xFFFFFFF8
0xFFFFFFFC

Memory

64-bit

32-bit

2^32 Bytes

$zero
$at
$5
$v0
$v1
$s0
$s1
$s2
$s3
$s4
$s5
$s6
$s7
$t0
$t1
$t2
$t3
$t4
$t5
$t6
$t7
$k0
$k1
$gp
$sp
$fp
$ra
## Frequently used MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></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><strong>Logical</strong></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><strong>Data Transfer</strong></td>
<td>lw</td>
<td>lw $s1, 4($s2)</td>
<td>$s1 = mem[$s2+4]</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>lw $s1, 4($s2)</td>
<td>mem[$s2+4] = $s1</td>
</tr>
<tr>
<td><strong>Branch</strong></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 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>
• op $rd, $rs, $rt
  • 3 regs.: add, addu, and, nor, or, sltu, sub, subu
  • 2 regs.: sll, srl
  • 1 reg.: jr

• 1 arithmetic operation, 1 I-memory access
• Example:
    opcode = 0x0, funct = 0x20
  • sll $t0, $t1, 8: R[8] = R[9] << 8
    opcode = 0x0, shamt = 0x8, funct = 0x0
**l-type**

- **op $rt, $rs, immediate**
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- **op $rt, offset($rs)**
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - `lw $s0, 4($s2)`:  
    \[ R[16] = \text{mem}[R[18]+4] \]
  - `add $s2, $s2, $s1`
  - `lw $s0, 0($s2)`

- **opcode** 6 bits  
- **rs** 5 bits  
- **rt** 5 bits  
- **immediate / offset** 16 bits

*only two addressing modes*
Data transfer instructions

• The ONLY type of instructions that can interact with memory in MIPS

• Two big categories
  • Load (e.g., lw): copy data from memory to a register
  • Store (e.g., sw): copy data from a register to memory

• Two parts of operands
  • A source or destination register
  • Target memory address = base address + offset
    • Register contains the “base address”
    • Constant as the “offset”
    • $8(\$s0) = (the content in \$s0) + 8$
l-type (cont.)

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory
  and 1 D-memory access
- Example:
  - beq $t0, $t1, -40
    
    if (R[8] == R[9]) PC = PC + 4 + 4*(-40)
J-type

- op immediate
  - j, jal
- 1 instruction memory access, 1 arithmetic op
- Example:
  - jal quicksort
    R[31] = PC + 4
    PC = quicksort
Translate the C code into assembly:

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

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

Assume
- int is 32 bits
- $s0 = &A[0]$  
- $v0 = sum;$  
- $t0 = i;$  

There are many ways to translate the C code. But efficiency may be differ among translations.
How to manage the memory space?

- Function A
  - Register values

- Function B
  - Register values

- Function C
  - Register values

Memory

stack pointer

stack pointer

stack pointer

stack pointer
Manage registers

• Sharing registers
  • A called function will modified registers
  • The caller may use these values later

• Using memory stack
  • The stack provides local storage for function calls
  • FILO (first-in-last-out)
  • For historical reasons, the stack grows from high memory address to low memory address
  • The stack pointer ($sp) should point to the top of the stack
Function calls

Caller

addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

Callee

hanoi: addi $sp, $sp, -8
sw $ra, 0($sp)
sw $a0, 4($sp)

hanoi_0: addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

hanoi_1: jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1

return: lw $a0, 4(sp)
lw $ra, 0(sp)
addi $sp, $sp, 8
jr $ra

save shared registers to the stack, maintain the stack pointer
restore shared registers from the stack, maintain the stack pointer
Recursive calls

**Caller**

- jal hanoi
- addi $a0, $zero, 2
- addi $a0, $t1, $t0

**Callee**

```
hanoi:  addi $sp, $sp, -8
         sw  $ra, 0($sp)
         sw  $a0, 4($sp)

hanoi_0:addi $a0, $a0, -1
         bne $a0, $zero, hanoi_1
         addi $v0, $zero, 1
         j    return

hanoi_1:jal hanoi
         sll  $v0, $v0, 1
         addi $v0, $v0, 1

return: lw  $a0, 4(sp)
         lw  $ra, 0(sp)
         addi $sp, $sp, 8
         jr  $ra
```
The overhead of function calls

The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead

Does not work if it’s called using a function pointer
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
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
The abstracted x86 machine architecture

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

CPU
- ADD
- SUB
- IMUL
- AND
- OR
- XOR
- JMP
- JE
- CALL
- RET

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

64-bit

2^4 Bytes

MOV
### Registers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16bit</td>
<td>32bit</td>
<td>64bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
<td>The accumulator register</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
<td>The base register</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
<td>The counter</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
<td>The data register</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
<td>Pointer to the base of stack frame</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>RnD</td>
<td></td>
<td>General purpose registers (8-15)</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
<td>Source index for string operations</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
<td>Destination index for string</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>EIP</td>
<td>RIP</td>
<td>Instruction pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>FLAGS</strong></td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ISA type</strong></td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td><strong>instruction width</strong></td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td><strong>code size</strong></td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td><strong>registers</strong></td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td><strong>addressing modes</strong></td>
<td>reg+offset</td>
<td>base+offset base+index scaled+index scaled+index+offset</td>
</tr>
<tr>
<td><strong>hardware</strong></td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Performance
Execution Time = \( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \)

- How many instructions are executed?
- How long does it take to execute each instruction?

- ET = IC * CPI * CT
- IC (Instruction Count)
- CPI (Cycles Per Instruction)
- CT (Seconds Per Cycle)
  - 1 Hz = 1 second per cycle; 1 GHz = 1 ns per cycle
dynamic v.s. static instructions

- Static instructions — number of instructions in the “compiled” code
- Dynamic instruction — number of instances of executing instructions when running the program

If the loop is executed 100 times, the dynamic instruction count will be 10+100*10+10

static instructions: 30
Speedup

• Compare the relative performance of the baseline system and the improved system
• Definition

\[
\text{Speedup} = \frac{\text{Execution time}_{\text{baseline}}}{\text{Execution time}_{\text{improved system}}}
\]
Performance Example

- 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 we double the CPU clock rate to 4GHz but keep using the same memory module, the average CPI for load/store instruction will become 12 cycles. What’s the performance improvement after this change?

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

\[
ET_{\text{new}} = 500000 \times (0.8\times1+0.2\times12) \times 0.25 \text{ ns} = 400000 \text{ ns}
\]

\[
\text{Speedup} = \frac{ET_{\text{old}}}{ET_{\text{new}}} = \frac{500000}{400000} = 1.25
\]
How many instructions are there in “Hello, world!”

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Instruction count</th>
<th>LOC</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>480k</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>C++</td>
<td>2.8M</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Java</td>
<td>166M</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Perl</td>
<td>9M</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Python</td>
<td>30M</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary: Performance Equation

Execution Time = \( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \)

- ET = IC * CPI * Cycle Time
- 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

\[
\text{Speedup} = \frac{1}{\left(\frac{x}{S}\right) + (1-x)}
\]

- \(x\): the fraction of “execution time” that we can speed up in the target application
- \(S\): by how many times we can speed up \(x\)

![Diagram](image.png)

The total execution time is 1.
Corollaries of Amdahl’s Law

- Maximum possible speedup $S_{\text{max}}$
  
  $$S_{\text{max}} = \frac{1}{(1-x)}$$

- Make the common case fast (i.e., $x$ should be large)
  - Common most time consuming not necessarily the most frequent
  - Use profiling tools to figure out

- Estimate the potential of parallel processing
  
  $$S_{\text{par}} = \frac{1}{\frac{x}{S} + (1-x)}$$

- Estimate the effect of multiple optimizations
  
  $$S = \frac{1}{(1 - X_{\text{Opt1Only}} - X_{\text{Opt2Only}} - X_{\text{Opt1&Opt2}}) + \frac{X_{\text{Opt1}}}{S_{\text{Opt1Only}}} + \frac{X_{\text{Opt2}}}{S_{\text{Opt2Only}}} + \frac{X_{\text{Opt1&Opt2}}}{S_{\text{Opt1&Opt2}}}}$$

Amdahl’s Law can help you in making the right decision!
Power

- Dynamic power: $P = aCV^2f$
  - $a$: switches per cycle
  - $C$: capacitance
  - $V$: voltage
  - $f$: frequency, usually linear with $V$
  - Doubling the clock rate consumes more power than a quad-core processor!
- Static/Leakage power becomes the dominant factor in the most advanced process technologies.
- Power is the direct contributor of “heat”
  - Packaging of the chip
  - Heat dissipation cost
Dynamic voltage/frequency scaling

- Dynamically trade-off power for performance
  - Change the voltage and frequency at runtime
  - Under control of operating system — that’s why updating iOS may slow down an old iPhone
- Recall: $P_{\text{dynamic}} \sim aC V^2 f N$
  - Because frequency $\sim$ to $V$
  - $P_{\text{dynamic}} \sim$ to $V^3$
- Reduce both $V$ and $f$ linearly
  - Cubic decrease in dynamic power
  - Linear decrease in performance (actually sub-linear)
    - Thus, only about quadratic in energy
  - Linear decrease in static power
    - Thus, only modest static energy improvement
- Newer chips can do this on a per-core basis
  - `cat /proc/cpuinfo` in Linux
Energy

• Energy = P * ET
• The electricity bill and battery life is related to energy!
• Lower power does not necessary means better battery life if the processor slow down the application too much
What happens if power doesn’t scale with process technologies?

• If we are able to cram more transistors within the same chip area (Moore’s law continues), but the power consumption per transistor remains the same. Right now, if we power the chip with the same power consumption but put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  ① The power consumption per chip will increase
  ② The power density of the chip will increase
  ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

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

- \[ \text{P}_{\text{Leakage}} \sim N^*V^*e^{-Vt} \]
  - \( N \): number of transistors
  - \( V \): voltage
  - \( Vt \): threshold voltage where transistor conducts (begins to switch)
- Your power consumption goes up as the number of transistors goes up
  - You have to turn off/slow down some transistors completely to reduce leakage power
  - Intel TurboBoost: dynamically turn off/slow down some cores to allow a single core achieve the maximum frequency
  - big.LITTLE cores: Qualcomm Snapdragon 835 has 4 cores can achieve more than 2GHz but 4 other cores can only achieve up to 1.9GHz
Bandwidth

- The amount of work (or data) during a period of time
  - Network/Disks: MB/sec, GB/sec, Gbps, Mbps
  - Game/Video: Frames per second
- Also called “throughput”
- “Work done” / “execution time”
Response time and BW trade-off

- Increase bandwidth can hurt the response time of a single task
- If you want to transfer a 2 Peta-Byte video from UCLA
  - 125 miles (201.25 km) from UCSD
  - Assume that you have a 100Gbps ethernet
    - 2 Peta-byte over 167772 seconds = 1.94 Days
    - 22.5TB in 30 minutes
    - Bandwidth: 100 Gbps
Toyota Prius
- 25 miles from UNC
- 75 MPH on highway!
- 50 MPG
- Max load: 374 kg = 2,770 hard drives (2TB per drive)

<table>
<thead>
<tr>
<th></th>
<th>10Gb Ethernet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100 Gb/s or 12.5GB/sec</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bandwidth</strong></td>
<td>290GB/sec</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>4 hours</td>
</tr>
<tr>
<td><strong>response time</strong></td>
<td>You see nothing in the first 4 hours</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>2 Peta-byte over 167772 seconds = 1.94 Days</td>
</tr>
<tr>
<td><strong>response time</strong></td>
<td>You can start watching the movie as soon as you get a frame!</td>
</tr>
</tbody>
</table>
TFLOPS (Tera FLoating-point Operations Per Second)
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</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 (Giga FLoating-point Operations Per Second) a good metric?

- 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

\[
TFLOPS = \frac{\text{# of floating point instructions} / 10^9}{\text{Execution Time}}
\]

\[
= \frac{\text{IC} \times \% \text{ of floating point instructions}}{\text{IC} \times \text{CPI} \times \text{CycleTime} \times 10^9}
= \frac{\text{Clock Rate} \times \% \text{ FP ins.}}{\text{CPI} \times 10^9}
\]
Processor Design
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

- Break up the logic with “pipeline registers” into pipeline stages
  - These registers only changes their output at the triggered edge cycle
- Each stage can act on different instruction/data
- States/Control signals of instructions are hold in pipeline registers
After the 5th cycle, the processor can do 5 instructions in parallel
Pipelining

The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!

But you only need to do this amount of things for each instruction in each cycle.
Single cycle processor
5-stage pipeline processor
5-stage pipeline processor
5-stage pipeline processor

- **add $1, $2, $3**
- **lw $4, 0($5)**
- **sub $6, $7, $8**
- **sub $9,$10,$11**
- **sw $1, 0($12)**
5-stage pipeline processor

Instruction memory

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw $1, 0($12)
5-stage pipeline processor

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw $1, 0($12)
5-stage pipeline processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
5-stage pipeline processor

Instruction memory

Instruction:
- $add\ $1, \ $2, \ $3$
- $lw\ $4, \ 0($5$)$
- $sub\ $6, \ $7, \ $8$
- $sub\ $9, \ $10, \ $11$
- $sw\ $1, \ 0($12$)$
5-stage pipeline processor

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

- Use symbols to represent the physical resources with the abbreviations for pipeline stages.
  - IF, ID, EXE, MEM, WB
- The horizontal axis represents the timeline, and the vertical axis represents the instruction stream
- Example:

  ```
  add $1, $2, $3
  lw $4, 0($5)
  sub $6, $7, $8
  sub $9,$10,$11
  sw $1, 0($12)
  ```
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

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

- The cycle time is 3ns
- Each instruction now takes “15ns” to leave the pipeline!
Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

Pipeline hazards:
- Structural hazard
  - The hardware does not allow two pipeline stages to work concurrently
- Data hazard
  - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
- Control hazard
  - The processor is not clear about what’s the next instruction to fetch
Can we get the right result?

- Given the current 5-stage pipeline,

how many of the following MIPS code can work correctly?

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>b:</td>
<td>lw  $4, 0($1)</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>bne $0, $7, L</td>
</tr>
<tr>
<td>d:</td>
<td>sub $9,$10,$11</td>
<td>sub $9, $1, $10</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>sub $9,$10,$11</td>
</tr>
</tbody>
</table>

b cannot get $1 produced by a before WB
both a and d are accessing $1 at 5th cycle
We don’t know if d & e will be executed or not

Data hazard  Structural hazard  Control hazard
Structural hazard
The hardware cannot support the combination of instructions that we want to execute at the same cycle.

The original pipeline incurs structural hazard when two instructions competing the same register.

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!
Structural hazard

- What pair of instructions will be problematic if we allow R-type instructions to skip the “MEM” stage?

A: a & b
B: a & c
C: b & e
D: c & e
E: None
Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order

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

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.
Data hazard
Sol. of data hazard I: Stall

- When the source operand of an instruction is not ready, stall the pipeline
  - Suspend the instruction and the following instruction
  - Allow the previous instructions to proceed
  - This introduces a pipeline bubble: a bubble does nothing, propagate through the pipeline like a nop instruction

- How to stall the pipeline?
  - Disable the PC update
  - Disable the pipeline registers on the earlier pipeline stages
  - When the stall is over, re-enable the pipeline registers, PC updates
Performance of stall

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)

add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
Sol. of data hazard II: Forwarding

- The result is available after EXE and MEM stage, but publicized in WB!
- The data is already there, we should use it right away!
- Also called bypassing

```
add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
```
Sol. of data hazard II: Forwarding

- Take the values, where ever they are!

```assembly
add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
```

10 cycles! CPI == 2 (Not optimal, but much better!)
5-stage pipeline processor
There is still a case that we have to stall...

- Revisit the following code:

```assembly
add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
```

If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!
5-stage pipeline processor
Control hazard
Consider the following code and the pipeline we designed

```
LOOP: lw $t3, 0($s0)
     addi $t0, $t0, 1
     add $v0, $v0, $t3
     addi $s0, $s0, 4
     bne $t1, $t0, LOOP
     sw $v0, 0($s1)
```

How many cycles does the processor need to stall before we figure out the next instruction after “bne”? 

A. 0
B. 1
C. 2
D. 3
E. 4
Why do we need to stall for branch instructions

- 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
Control hazard

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards, what’s the average CPI if we execute this program on the 5-stage MIPS pipeline?
  A. 1
  B. 1.2
  C. 1.4
  D. 1.6
  E. 1.8
Branch prediction to reduce the overhead of control hazards
Tips of drawing pipeline diagram

Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order.

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

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.

Assume full data forwarding, predict always taken

```
addi $a1, $zero, 2
lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
addi $a1, $zero, 2
lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
lw $t1, 0($a0)
lw $a0, 0($t1)
add $v0, $zero, $a1
```

Addi $a1, $zero, 2
Loop: lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
addi $a1, $zero, 2
lw $t1, 0($a0)
lw $a0, 0($t1)
addi $a1, $a1, -1
bne $a1, $zero, LOOP
For midterm

• No cheat sheet allowed
• No cheating allowed
• We will use scantron
• We will have some problems require you to write
• You may bring a calculator
• You should bring pen/pencil/eraser
Sample midterm
MIPS v.s. x86

• Which of the following is NOT correct about these two ISAs?
  A. x86 provides more instructions than MIPS
  B. x86 usually needs more instructions to express a program
  C. An x86 instruction may access memory for 3 times
  D. An x86 instruction may be shorter than a MIPS instruction
  E. An x86 instruction may be longer than a MIPS instruction
Identify the performance bottleneck

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
Performance of a single-cycle processor

• How many of the following statements about a single-cycle processor is correct?
  • The CPI of a single-cycle processor is always 1
  • If the single-cycle implements lw, sw, beq, and add instructions, the sw instruction determines the cycle time
  • Hardware elements are mostly idle during a cycle
  • 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?
  • You can always divide stages into short stages with latches
  • Pipeline registers incur overhead for each pipeline stage
  • The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
  • The throughput of a pipeline processor is usually better than a single-cycle processor
  • Pipelining a stage can always improve cycle time

A. 1
B. 2
C. 3
D. 4
E. 5
How many pairs of data dependences are there in the following code?

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

A. 1
B. 2
C. 3
D. 4
E. 5
Branch predictions

• How many of the following about static branch prediction method are correct?
  • Comparing with stalls, branch prediction mechanisms are never doing worse in our current MIPS 5-stage pipeline
  • The dynamic 2-bit branch prediction mechanism never changes the prediction result during program execution
  • “Flush” occurs only after the processor detects an incorrect branch prediction
  • The branch predictor cannot fetch a taken instruction during the ID stage of the branch instruction without the help of BTB
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4
Fair comparison

• How many of the following comparisons are fair?

① Comparing the frame rates of Halo 5 on AMD RyZen 1600X and civilization on Intel Core i7 7700X
② Using bit torrent to compare the network throughput on two machines
③ Comparing the frame rates of Halo 5 using medium settings on AMD RyZen 1600X and low settings on Intel Core i7 7700X
④ Using the peak floating point performance to judge the gaming performance of machines using AMD RyZen 1600X and Intel Core i7 7700X

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
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 1GHz, how long is the execution time?
Example of Amdahl’s Law

- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine, and spends 20% of this time in integer instructions.
- How much faster must you make the integer unit to make the map loading 1 minute faster?
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 are the uniformity of MIPS?
• Why power consumption is an important issue in computer system design?
Other things to think ...

- Why GFLOPS (Giga 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
Thank you!