What is computer architecture about?

- Computer architecture is the study of building computer systems.

- CSE378 is roughly split into three parts.
  - The first third discusses instruction set architectures—the bridge between hardware and software.
  - Next, we introduce more advanced processor implementations. The focus is on pipelining, which is one of the most important ways to improve performance.
  - Finally, we talk about memory systems, I/O, and how to connect it all together.
Why should you care?

- It is interesting.
  - You will learn how a processor actually works!

- It will help you be a better programmer.
  - Understanding how your program is translated to assembly code lets you reason about correctness and performance.
  - Demystify the seemingly arbitrary (e.g., bus errors, segmentation faults)

- Many cool jobs require an understanding of computer architecture.
  - The cutting edge is often pushing computers to their limits.
  - Supercomputing, games, portable devices, etc.

- Computer architecture illustrates many fundamental ideas in computer science
  - Abstraction, caching, and indirection are CS staples

CSE 370 vs. CSE 378

- This class expands upon the computer architecture material from the last few weeks of CSE370, and we rely on many other ideas from CS370.
  - Understanding binary, hexadecimal and two's-complement numbers is still important.
  - Devices like multiplexers, registers and ALUs appear frequently. You should know what they do, but not necessarily how they work.
  - Finite state machines and sequential circuits will appear again.

- We do not spend time with logic design topics like Karnaugh maps, Boolean algebra, latches and flip-flops.
Who we are

- **Instructor:**
  Luis Ceze, luisceze@cs, **Office: CSE 576**

- **Teaching Assistants:**
  Jacob Nelson nelson@cs
  Aaron Miller ajmiller@cs

- **Communications**
  - discussion board
  - mailing list (mostly for announcements from course staff)

Who is Luis?

PhD in architecture, multiprocessors, parallelism, compilers.
Who are you?

- 59 students (wow!)
- Who has written programs in assembly before?
- Anyone designed HW before?
- Written a threaded program before?

Administrative - The Course

The textbook provides the most comprehensive coverage (it’s a beautiful textbook, easy to read & use)

- Computer Organization and Design, Patterson and Hennessy, 4th Edition

Lectures will present course material

Sections, you signed up for one; here’s how they work

- We have CSE 003 Lab (2:30-5:30) for “lab work”
- We’ll use another room (tba) for “classroom work” as needed
- Use lab time wisely, because they won’t usually be around at other times
- Don’t expect to finish lab projects during your official lab time – start immediately and plan on outside time
Administrivia - The Grading

Grading
- Lab assignments: 25%
- Homeworks: 15%
- Midterm: 20%
- Final: 35%
- Participation: 5%

Midterm: May 5, in class

Final: Trying to change. More later.

Instruction set architectures

- Interface between hardware and software
  — abstraction: hide HW complexity from the software through a set of simple operations and devices

  add, mul, and, lw, ...
MIPS

- In this class, we'll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
  - Of course, the concepts are not MIPS-specific
  - MIPS is just convenient because it is real, yet simple (unlike x86)

- The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2

---

From C to Machine Language

High-level language (C)

\[ a = b + c; \]

Compiler

Add $16, $17, $18

Assembler

Binary Machine Language (MIPS)

01010111010101101...
What you will need to learn soon

- You must become "fluent" in MIPS assembly:
  - Translate from C to MIPS and MIPS to C

- Example problem: Write a recursive function

Here is a function pow that takes two arguments (n and m, both 32-bit numbers) and returns \( n^m \) (i.e., \( n \) raised to the \( m \)th power):

```c
int
pow(int n, int m) {
  if (m == 1)
    return n;
  return n * pow(n, m-1);
}
```

Translate this into a MIPS assembly language function.

Instruction Execution Engines

Computers are instruction execution engines that endlessly run the fetch/execute cycle

![Instruction Execution Diagram]

This course explains in detail this logical process and how it is implemented in hardware.
MIPS: register-to-register, three address

- MIPS is a register-to-register, or load/store, architecture.
  - The destination and sources must all be registers.
  - Special instructions, which we’ll see soon, are needed to access main memory.

- MIPS uses three-address instructions for data manipulation.
  - Each ALU instruction contains a destination and two sources.
  - For example, an addition instruction \((a = b + c)\) has the form:

```
operation   operands
add         a, b, c
```

MIPS register file

- MIPS processors have 32 registers, each of which holds a 32-bit value.
  - Register addresses are 5 bits long.
  - The data inputs and outputs are 32-bits wide.
- More registers might seem better, but there is a limit to the goodness.
  - It's more expensive, because of both the registers themselves as well as the decoders and muxes needed to select individual registers.
  - Instruction lengths may be affected, as we'll see in the future.
MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - By number:
    
    \[
    \begin{align*}
    \$0 & \quad \$1 & \quad \$2 & \quad \ldots & \quad \$31 \\
    \end{align*}
    \]
  - By (mostly) two-character names, such as:
    
    \[
    \begin{align*}
    \$a0-\$a3 & \quad \$s0-\$s7 & \quad \$t0-\$t9 & \quad \$sp & \quad \$ra \\
    \end{align*}
    \]
- Not all of the registers are equivalent:
  - E.g., register \$0 or $zero always contains the value 0
    (go ahead, try to change it)
  - E.g., register $sp is used to hold the "stack pointer"
- Other registers have special uses, by convention:
  - E.g., register $t0 is used to hold the "stack pointer"

You have to be a little careful in picking registers for your programs.
— More about this later

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:
  
  \[
  \begin{align*}
  \text{add} & \quad \text{sub} & \quad \text{mul} & \quad \text{div} \\
  \end{align*}
  \]
- And here are a few logical operations:
  
  \[
  \begin{align*}
  \text{and} & \quad \text{or} & \quad \text{xor} \\
  \end{align*}
  \]
- Remember that these all require three register operands; for example:
  
  \[
  \begin{align*}
  \text{add} & \quad \$t0, \$t1, \$t2 \quad \# \ \$t0 = \$t1 + \$t2 \\
  \text{mul} & \quad \$s1, \$s1, \$a0 \quad \# \ \$s1 = \$s1 \times \$a0 \\
  \text{mul} & \quad \$t0, \$t1, \$t2 \\
  \text{mul} & \quad \$t2, \$t3, \$t4 \\
  \text{add} & \quad \$t0, \$t0, \$t2 \\
  \end{align*}
  \]
Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction set level.

\[ t_0 = (t_1 + t_2) \times (t_3 - t_4) \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$t_0$, $t_1$, $t_2$</td>
<td>$t_0$ contains $t_1 + t_2$</td>
</tr>
<tr>
<td>sub</td>
<td>$s_0$, $t_3$, $t_4$</td>
<td>Temporary value $s_0 = t_3 - t_4$</td>
</tr>
<tr>
<td>mul</td>
<td>$t_0$, $s_0$, $s_0$</td>
<td>$t_0$ contains the final product</td>
</tr>
</tbody>
</table>

- Temporary registers may be necessary, since each MIPS instruction can access only two source registers and one destination.
  - In this example, we could re-use $t_3$ instead of introducing $s_0$.
  - But be careful not to modify registers that are needed again later.

Immediate operands

- The ALU instructions we’ve seen so far expect register operands. How do you get data into registers in the first place?
  - Some MIPS instructions allow you to specify a signed constant, or “immediate” value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

\[ \text{addi } t_0, t_1, 4 \quad \text{# } t_0 = t_1 + 4 \]

- Immediate operands can be used in conjunction with the $s_0$ zero register to write constants into registers:

\[ \text{addi } t_0, s_0, 4 \quad \text{# } t_0 = 4 \]

- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.
We need more space!

- Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide.
  - That's not enough to hold data structures like large arrays.
  - We also can't access data elements that are wider than 32 bits.
- We need to add some main memory to the system!
  - RAM is cheaper and denser than registers, so we can add lots of it.
  - But memory is also significantly slower, so registers should be used whenever possible.
- In the past, using registers wisely was the programmer's job.
  - For example, C has a keyword “register” to mark commonly-used variables which should be kept in the register file if possible.
  - However, modern compilers do a good job of using registers intelligently and minimizing RAM accesses.

How to Succeed in CSE 378

- **Remember the big picture.**
  What are we trying to accomplish, and why?

- **Read the textbook.**
  It's clear, well-organized, and well-written. The diagrams can be complex, but are worth studying. Work through the examples and try some exercises on your own. Read the “Real Stuff” and “Historical Perspective” sections.

- **Talk to each other.**
  You can learn a lot from other CSE378 students, both by asking and answering questions. Find some good partners for the homeworks/labs (but make sure you all understand what's going on).

- **Help us help you.**
  Come to lectures, sections and office hours. Use the discussion board & Wiki. Ask lots of questions! Check out the web pages.