Welcome minute: Shanti!
The words means “Peace”

I love Playing Badminton and Table Tennis

Hate Animal cruelty. I eat lots of veggies, yay!

I miss my dog back home, so I babysit this cutie here

I love dancing and trekking along “short” forest trails :P

🎵 Yellow Submarine, The Beatles 🎵
Announcements

• Send your welcome slide!
• Homework 1 is posted
  – Logistical updates & questions
    • Posted on Gradescope – is it helpful to create an “assignment” on Canvas as well?
    • “Call for Consistency”: Homework every week, assigned Thursday, due Thursday
• Reminder: First participation quiz will go live today
  – Due: Tuesday (midnight)
  – You have “something 141” every day (M/W/F lecture; Tu Mini-quiz; Th HW)
“Control Flow” describes how programs execute

- Jumps
- Procedure call (jump subroutine)
- Conditional Branch
  - Used to implement, for example, if-then-else logic, loops, etc.

- Control flow must specify two things
  - Condition under which the jump or branch is taken
  - If take, the location to read the next instruction from ("target")
Jumps are unconditional control flow. What do they look like in MIPS?

- need to be able to jump to an absolute address sometimes
- need to be able to do procedure calls and returns

- **Jump**
  - `j 10000 => PC = 10000`
- **Jump and Link**
  - `jal 20000 => $31 = PC + 4` and `PC = 20000`
  - used for procedure calls
- **Jump register**
  - `jr $31 => PC = $31`
  - used for returns, but can be useful for lots of other things
  - Q: how to encode `jr` instruction?

**Warning:** Some ISAs call jumps “unconditional branches” – useful not to for MIPS
What if we want to condition the control flow? Branches.

do { ... ; a++; } while (a < 100);

• `beq` and `bne` are the only branches you need
  – `beq r1, r2, addr` => if (r1 == r2): goto addr

• But other operations can be combined...
  – `slt $1, $2, $3` => if ($2 < $3) $1 = 1; else $1 = 0

• `beq`, `bne`, `slt`, and `$zero`, can implement all fundamental conditions
  – Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```c
if (i < j)
    w = w + 1;
else
    w = 5;
```
Re-working this example

1. Need to do the comparison
   - Use “store less than”, \texttt{slt $\text{temp}, \text{i, j}}
   - This writes 1 in \text{temp} \text{ \textit{when the condition is true}}

2. Need to decide whether to branch, \textit{using only registers}
   - Only have \texttt{$\text{zero}} \text{ available to compare with}
   - The question is “should we jump over the if body”
   - Want to jump to \texttt{else\_body} \text{ when } \text{temp} \text{ is 0}
   - So we conceptually we are asking \texttt{if !(i<j) \text{ [confusing!]}}
   - \texttt{beq $\text{temp, zero, else\_body}}
   - This says goto the else body \textit{when the slt was not true}

3. Need to jump over the else body
   - Don’t do both the \texttt{if} and the \texttt{else} on accident!
   - Use “unconditional jump”
   - \texttt{\text{j after\_else}}

4. Finally, fill in the bodies
   \begin{verbatim}
   if (i<j)
       \text{if\_body:}
       $w = w+1$
   \text{else}
       \text{else\_body:}
       \text{w = 5;}
       \text{after\_else:}
   \end{verbatim}
   \begin{verbatim}
   \text{slt $\text{temp, i, j}}
   \text{beq $\text{temp, zero, else\_body}}
   \text{if\_body:}
   \text{addi $w, w, 1}
   \text{j after\_else}
   \text{else\_body:}
   \text{addi $w, \text{zero, 5}}
   \text{after\_else:}
   \end{verbatim}
FAQs / Extras

1. Could we have used a `bne` instead?
   - Yes, if you get the value 1 into a register

   ```
   if (i<j)
   if_body:
       w = w+1;
   else
   else_body:
       w = 5;
   after_else:
   ```

   - But this is inefficient
     - Extra instruction
     - **Register pressure**
FAQs / Extras

1. Could we have used a bne with no more instructions?
   
   – Yes... if you flip the body and “put the else first”

   ```asm
   if (i<j)
   if_body:
     w = w+1;
   else
   else_body:
     w = 5;
   after_else:
   slt $temp, $i, $j
   bne $temp, $zero, if_body
   else_body:
   addi $w, $zero, 5
   j  after
   if_body:
   addi $w, $w, 1
   after:
   ```

   – Real compilers do this sometimes
How do you specify the destination of a branch/jump?

• Unconditional jumps may go long distances
  – Function calls, returns, ...

• Studies show that almost all conditional branches go short distances from the current program counter
  – loops, if-then-else, ...

• A relative address requires (many) fewer bits than an absolute address
  – e.g., beq $1, $2, 100 => if ($1 == $2): PC = (PC+4) + 100 * 4
MIPS Branch and Jump Addressing Modes

• Branches (e.g., beq) use PC-relative addressing mode
  – uses fewer bits since address typically close
  – Aka: base+displacement mode, with the PC being the base

• Jumps use pseudo-direct addressing mode
  – Recall opcode is 6 bits...
    • How many bits are available for displacement? How far can you jump?
  – 26 bits of the address is in the instruction, the rest is taken from the PC.
### MIPS operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory[0], Memory[4], ...</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^{16}</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 * 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 * 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Review — Instruction Execution in a CPU

Registers\textsubscript{10} 
\begin{align*} 
\text{R0} & \quad 0 \\
\text{R1} & \quad \textcolor{red}{60000} \\
\text{R2} & \quad 45 \\
\text{R3} & \quad 198 \\
\text{R4} & \quad \textcolor{red}{20000} \\
\text{R5} & \quad \textcolor{red}{10001} \\
\text{...} & \quad \text{...} \\
\end{align*}

Program Counter\textsubscript{10} 
\begin{align*} 
20000 \\
20004 \\
80000 \\
\end{align*}

Instruction Buffer

Address\textsubscript{10} 
\begin{align*} 
20000 & \quad 1000110001000110100111000100000 \\
20004 & \quad 000000000100001010000001000000 \\
80000 & \quad 00000000000000000000000000111001 \\
\end{align*}

Memory

Load/Store Unit

ALU

CPU
Poll Q: Work an Example

• Can we figure out the code?

```c
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

**swap:**
- `mul` $2, $5, 4
- `add` $2, $4, $2
- `lw` $15, 0($2)
- `lw` $16, 4($2)
- `sw` $16, 0($2)
- `sw` $15, 4($2)
- `jr` $31

**Where is k?**

<table>
<thead>
<tr>
<th></th>
<th>Where is k?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$4</td>
</tr>
<tr>
<td>B</td>
<td>$5</td>
</tr>
<tr>
<td>C</td>
<td>$15</td>
</tr>
<tr>
<td>D</td>
<td>$16</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
# MIPS ISA Tradeoffs

The MIPS Instruction Set Architecture (ISA) includes several tradeoffs that allow for efficient execution of instructions. Here are some key tradeoffs:

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td><strong>OP</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>rd</strong></td>
<td><strong>sa</strong></td>
<td><strong>funct</strong></td>
</tr>
<tr>
<td>I-type</td>
<td><strong>OP</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>immediate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td><strong>OP</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>target</strong></td>
<td></td>
</tr>
</tbody>
</table>

### What if?
- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. $Y = AX + B$)
RISC Architectures

• MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
  – fixed instruction length
  – few instruction formats
  – load/store architecture

• RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.
Alternative Architectures

• Design alternative:
  – provide more powerful operations
  – goal is to reduce number of instructions executed
  – danger is a slower cycle time and/or a higher CPI (cycles per instruction)

• Sometimes referred to as “RISC vs. CISC”
  – CISC = Complex Instruction Set Computer (as alt to RISC)
  – virtually all new instruction sets since 1982 have been RISC
  – VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!

• We’ll look (briefly!) at PowerPC and 80x86
• What is ARM?
PowerPC

• Indexed addressing
  – example: \texttt{lw \$t1, \$a0 + \$s3} \hspace{1em} # \$t1=\text{Memory[\$a0 + \$s3]}
  – What do we have to do in MIPS?

• Update addressing
  – update a register as part of load (for marching through arrays)
  – example: \texttt{lwu \$t0, 4(\$s3)} \hspace{1em} # \$t0=\text{Memory[\$s3 + 4]; \$s3=\$s3+4}
  – What do we have to do in MIPS?

• Others:
  – load multiple/store multiple
  – a special counter register “\texttt{bc Loop}”
  \hspace{1em} \textit{decrement counter, if not 0 goto loop}
1978: The Intel 8086 is announced (16 bit architecture)
1980: The 8087 floating point coprocessor is added
1982: The 80286 increases address space to 24 bits, +instructions
1985: The 80386 extends to 32 bits, new addressing modes
1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
1997: MMX is added
1999: Pentium III (same architecture)
2001: Pentium 4 (144 new multimedia instructions), simultaneous multithreading (hyperthreading)
2005: dual core Pentium processors
2006: quad core (sort of) Pentium processors
2009: Nehalem – eight-core multithreaded processors
2015: Skylake – 4-core, multithreaded, added hw security features, transactional memory...
80x86

• Complexity:
  – Instructions from 1 to 17 bytes long
  – one operand must act as both a source and destination
  – one operand can come from memory
  – complex addressing modes
e.g., “base or scaled index with 8 or 32 bit displacement”

• Saving grace:
  – the most frequently used instructions are not too difficult to build
  – compilers avoid the portions of the architecture that are slow
Key Points

• MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
• MIPS is optimized for fast pipelined performance, not for low instruction count
• Historic architectures favored code size over parallelism.
• MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.