MIPS assembly programming:
Today’s lecture

- Exam 3 Structure
- Review the Datapath
  - Trace a couple of instructions
- Assembly programming
  - Register names
  - How is it implemented?
- Branches
  - Loops
  - If/then/else
  - How implemented?
Exam 3 Structure

- OH WOW! In 6 weeks you have learned how to build a computer!

- For exam 3, you will add components and control signals to the datapath to implement a new instruction for MIPS
  - Create an instruction that creates world peace
**CORE INSTRUCTION SET**

<table>
<thead>
<tr>
<th>NAME</th>
<th>MNEMONIC</th>
<th>MAT</th>
<th>OPERATION (in Verilog)</th>
<th>OPCODE/FUNCT (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Word</td>
<td><strong>Lw</strong></td>
<td><strong>I</strong></td>
<td>( R[rt] = M[R[rs] + \text{SignExtImm}] )</td>
<td>23\text{_{hex}}</td>
</tr>
</tbody>
</table>

**LOAD**

**l**w **$5, 1024($2)**

**ALU**

**alu_op[2:0]**

**write_enable**

**itype**

**except**

**opcode[5:0]**

**funct[5:0]**
CORE INSTRUCTION SET

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FOR-OPERATION (in Verilog)</th>
<th>OPCODE/FUNCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Upper Imm</td>
<td>lui I R[rt] = {imm, 16,b0}</td>
<td>f_{hex}</td>
</tr>
<tr>
<td>CORE INSTRUCTION SET</td>
<td>FOR-</td>
<td>OPERATION (in Verilog)</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----</td>
<td>------------------------</td>
</tr>
<tr>
<td>NAME,</td>
<td>MNEMONIC</td>
<td>MAT</td>
</tr>
<tr>
<td>Add</td>
<td>add</td>
<td>R</td>
</tr>
</tbody>
</table>
**CORE INSTRUCTION SET**

<table>
<thead>
<tr>
<th>NAME, OPERATION (in Verilog)</th>
<th>FOR-MAT</th>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Word</td>
<td>I</td>
<td>sw</td>
<td>R[rt] = R[rs] + R[rt]</td>
</tr>
</tbody>
</table>

**OPCODE/FUNCT (Hex)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw $4, 8($4)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>x</td>
</tr>
</tbody>
</table>
What you need for exams 4 & 5

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

Example problem from a 233 mid-term:

Question 3: Write a recursive function (30 points)

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.
We give MIPS registers meaningful names to help when writing software

- In hardware, all the registers are equivalent:
  - Except register $0$, which is always zero

- For temporary values, we’ll use the $t$ registers
  
  $t0$-$t9$

- If you have no reason for picking another register, then you should probably be using a $t$ register.
Replace register numbers with names

$\rightarrow t0 = (t1 + t2) \times (t3 - t4)$

$8 = (9 + 10) \times (11 - 12)$

add $t0, t1, t2$ # $t0$ contains $t1 + t2$
sub $t5, t3, t4$ # Temporary value $t5 = t3 - t4$
mul $t0, t0, t5$ # $t0$ contains the final product
How do we perform calculations on data in main memory?

```c
char A[4] = {1, 2, 3, 4};
int result;
void main(){
}
```
Computing on data in main memory generally requires load->compute->store

- Steps
  1. Load the data from memory into the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.
Global data is allocated in the .data segment

- Allocated to memory addresses at compile time.
- Amount of space allocated is based on variable type.

```
.data        // indicates the beginning of data segment
.word        // allocates space for 4-byte variable
.byte        // allocates space for 1-byte variable
.asciiz      // allocates space for an ASCII string
.space       // allocates a defined amount of space.
```

```
" hi (CS 233)! 
```

```
.space 64
```
Use either byte or word operations based on datatype

\[ \text{lb} \text{ and } \text{sb} \quad \text{lw} \text{ and } \text{sw} \]

- Transfer 1 byte of data between regs and mem
- Datatypes: char
- Note: Use least significant bits from registers

- Transfer 1 word (4 bytes) of data between regs and mem
- Datatypes: integers, float, addresses/pointers
- Note: must be word-aligned
Word alignment: 32-bit words must start at an address that is divisible by 4.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.
Arrays are groups of variables contiguous in memory

- Contiguous = laid out one after another in memory.

```
char name[8] = {'s', 't', 'r', 'i', 'n', 'g', '!' , '!' };

int pair[2] = {4, 5};
```

Which cell contains the bits 00000101?
An array of words

- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
  - The first array element is at address 2000.
  - The second word is at address 2004, not 2001.
- Revisiting the earlier example, if $a0 contains 2000, then
  \[
  \text{lw } \$t0, 0(\$a0) \quad A[0]
  \]
  accesses the 0th word of the array, but
  \[
  \text{lw } \$t0, 8(\$a0) \quad A[2]
  \]
  would access the 2nd word of the array, at address 2008.
Pseudo-instructions give programmers useful instructions that are not part of the MIPS architecture.

<table>
<thead>
<tr>
<th>Pseudo instructions</th>
<th>Real instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>li $a0, 20</code></td>
<td><code>addi $a0, $0, 20</code></td>
</tr>
<tr>
<td><code>move $a1, $t0</code></td>
<td><code>add $a1, $t0, $0</code></td>
</tr>
</tbody>
</table>

- A complete list of instructions is given in [Appendix A](#) of the text.
char A[4] = {1, 2, 3, 4};
int result;

void main(){
}

.data
A: .byte 1, 2, 3, 4

.text
main:
    la $t0, A
    lb $t1, 0($t0)
    lb $t2, 1($t0)
    add $t3, $t1, $t2
    add $t4, $t3, $t3
    add $t5, $t4, $t3
    la $t6, result
    sw $t5, $t6($t6)
Coding Example

```c
int A[4] = {1,2,3,4};
int result;

If we changed the datatype of the array? Which part of the load instruction would need to change?

Orig: `lb  $t2, 1($t0)  # loads A[1]
      a)  b)  c)  d)  e)More than one
      part needs to be changed`
Pseudo-branches (motivation)

- Need a slt and a beq.... (or was it a bne?)

```assembly
slti $t0, $t4, 10  # slt immediate version
___  $t0, $zero, skip_if_body  a)eq
             b)ne
```

This is extremely error prone; bad design for humans
Assemblers provide 4 pseudo-branches to make our lives easier

```
blt   $t0, $t1, L1  # Branch if $t0 < $t1
ble   $t0, $t1, L2  # Branch if $t0 <= $t1
bgt   $t0, $t1, L3  # Branch if $t0 > $t1
bge   $t0, $t1, L4  # Branch if $t0 >= $t1
```

There are also immediate versions of these branches, where the second source is a constant instead of a register.
Pseudo-branches assemble down to $\text{slt}$ and either $\text{beq}$ or $\text{bne}$

\[
\text{blt} \ $a0, \ $a1, \ \text{Label} \\
\text{Assembles into} \\
\text{slt} \ $at, \ $a0, \ $a1 \ \# \ $at = 1 \text{ if } $a0 < $a1 \\
\text{bne} \ $at, \ $0, \ \text{Label} \ \# \ \text{Branch if } $at \neq 0
\]

$at$ is the “assembler temporary” register ($$1$$)
if-then-else statements require branches and jumps

- If there is an else clause, it is the target of the conditional branch
- And the then clause needs a jump over the else clause

```
if (v0 < 0)
    v0 --;
else
    v0 ++;
```

```
E:
    addi $v0, $v0, 1
L:
    move $v1, $v0
```