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]+SignExtImm]</td>
<td>23&lt;sub&gt;hex&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
**CORE INSTRUCTION SET**

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FOR-MAT</th>
<th>OPERATION (in Verilog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Upper Imm</td>
<td>lui</td>
<td>( R[rt] = {imm, 16,b0} )</td>
</tr>
</tbody>
</table>

**_OPCODE/FUNCT (Hex)**

\( f_{hex} \)
### CORE INSTRUCTION SET

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FOR-MAT</th>
<th>OPERATION (in Verilog)</th>
<th>OPCODE/FUNCT (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>add</td>
<td>R</td>
<td>M[R[rs] + SignExtimm] – R[rt]</td>
</tr>
</tbody>
</table>

**Verilog Code:**

```verilog
reg [31:0] A; // Destination register
reg [31:0] B; // Source register
reg [31:0] C; // Source register

A = B + C;
```

**Diagram:**

The diagram shows the flow of data through the ALU and memory components, illustrating the process of adding registers. The graphical representation includes control signals such as reset, clk, and enable, along with the connections between registers and memory. The specific instructions and their operands are highlighted in the diagram to emphasize the flow of data and the logical operations performed by each component.
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>Store Word</td>
<td>sw</td>
<td>I</td>
<td>R[rt] = R[rs] + R[rt]</td>
<td>0 / $20_{hex}$</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

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

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

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
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 the 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.
```

```
.space 64
```

" hi CS 233 !"
Use either byte or word operations based on datatype

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

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

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

- 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.

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

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

Address

<table>
<thead>
<tr>
<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>Address 4B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>8-bit data</td>
<td>0x00000064</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
  \[ \text{lw } \$t0, 0(\$a0) \]
  accesses the 0th word of the array, but
  \[ \text{lw } \$t0, 8(\$a0) \]
  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.

- A complete list of instructions is given in Appendix A of the text.
Coding Example

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

void main(){
}
```
Coding Example

```cpp
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) **i**-**clicker**

```markdown
if (x < 10) {
    ... 
}
```

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

```markdown
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

\begin{verbatim}
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
\end{verbatim}

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

\begin{align*}
\texttt{blt} \ $a0, \ $a1, \ \text{Label} \\
\text{Assembles into} \\
\texttt{slt} \ $at, \ $a0, \ $a1 \ # \ $at = 1 \text{ if } $a0 < $a1 \\
\texttt{bne} \ $at, \ $0, \ \text{Label} \ # \ \text{Branch if } $at \neq 0
\end{align*}

\$at\text{ is the “assembler temporary” register (}$\$1$\text{)}
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

```c
if (v0 < 0)
    v0 --;
else
    v0 ++;
v1 = v0;
```
```assembly
bge $v0, $0, E
subi $v0, $v0, 1
j    L
E: addi $v0, $v0, 1
L: move $v1, $v0
```