I wonder where this goes?
“Motivating Force”

or

“Inciting Incident”

This is the point in the course where the PLOT actually begins. We are now ready to build a computer.

The ingredients are all in place, now it is time to build a legitimate computer. One that executes instructions, much the way any other desktop, PDA, or other computer does.
## Review: The MIPS ISA

### Instruction classes distinguished by types:

1. **3-operands ALU**
2. **ALU w/immediate**
3. **Loads/Stores**
4. **Branches**
5. **Jumps**

### The MIPS instruction set as seen from a Hardware Perspective

<table>
<thead>
<tr>
<th>OP</th>
<th>r_s</th>
<th>r_t</th>
<th>r_d</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### R-type: ALU with Register operands

\[
\text{Reg}[rd] \leftarrow \text{Reg}[rs] \text{ op Reg}[rt] 
\]

<table>
<thead>
<tr>
<th>001XXX</th>
<th>r_s</th>
<th>r_t</th>
<th>immediate</th>
</tr>
</thead>
</table>

#### I-type: ALU with constant operand

\[
\text{Reg}[rt] \leftarrow \text{Reg}[rs] \text{ op SEXT(Immediate)} 
\]

<table>
<thead>
<tr>
<th>10XXXX</th>
<th>r_s</th>
<th>r_t</th>
<th>immediate</th>
</tr>
</thead>
</table>

#### L-type: Load and Store

\[
\begin{align*}
\text{Reg}[rt] & \leftarrow \text{Mem}[\text{Reg}[rs] + \text{SEXT(Immediate)}] \\
\text{Mem}[\text{Reg}[rs] + \text{SEXT(Immediate)]]} & \leftarrow \text{Reg}[rt]
\end{align*}
\]

<table>
<thead>
<tr>
<th>0001XX</th>
<th>r_s</th>
<th>r_t</th>
<th>immediate</th>
</tr>
</thead>
</table>

#### I-type: Branch Instructions

\[
\begin{align*}
\text{if (Reg[rs] == Reg[rt]) PC} & \leftarrow \text{PC + 4 + 4*SEXT(Immediate)} \\
\text{if (Reg[rs] != Reg[rt]) PC} & \leftarrow \text{PC + 4 + 4*SEXT(Immediate)}
\end{align*}
\]

<table>
<thead>
<tr>
<th>00001X</th>
<th>r_s</th>
<th>r_t</th>
<th>immediate</th>
</tr>
</thead>
</table>

#### J-type: jump

\[
\text{PC} \leftarrow (\text{PC} \& 0xf0000000) \lor 4*(\text{Immediate}) 
\]

<table>
<thead>
<tr>
<th>000001X</th>
<th>26-bit constant</th>
</tr>
</thead>
</table>

---

*Comp 411*
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We’ll try implementing data paths for each class individually, and merge them (using MUXes, etc).
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We’ll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We’ll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions

Our Bag of Components:

Registers
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We'll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions

Our Bag of Components:
- Registers
- Muxes
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We'll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
3. Load & Store Instructions
4. Jump & Branch instructions
5. Exceptions

Our Bag of Components:
- Registers
- Muxes
- ALU & adders
Design Approach

Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We’ll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
3. Load & Store Instructions
4. Jump & Branch instructions
5. Exceptions

Our Bag of Components:
- Registers
- MUXes
- ALU & adders
- Memories

- RA1, RA2
- WA, WE, WD
- Register File (3-port)
- Instruction Memory
- Data Memory
A Few ALU Tweaks

Let's review the ALU that we built a few lectures ago. (With a few minor additions)
A Few ALU Tweaks

Let's review the ALU that we built a few lectures ago.

(With a few minor additions)
Instruction Fetch/Decode

- Use a counter to FETCH the next instruction:
  PROGRAM COUNTER (PC)

- use PC as memory address
- add 4 to PC, load new value at end of cycle
- fetch instruction from memory
  - use some instruction fields directly (register numbers, 16-bit constant)
  - use bits <31:26> and <5:0> to generate controls
3-Operand ALU Data Path

R-type: ALU with Register operands

Reg[rd] ← Reg[rs] op Reg[rt]

PC +4

Instruction Memory

Control Logic

Register File

ALU
Shift Instructions

R-type: ALU with Register operands
sll: Reg[rd] ← Reg[rt] (shift) shamt
sltv: Reg[rd] ← Reg[rt] (shift) Reg[rs]

- Rs: <25:21>
- Rd: <15:11>
- Rt: <20:16>
- ALU with Register operands

PC +4

Instruction Memory

Control Logic

Register File

ALU

ALUFN
WERF
ASEL

shamt: <10:6>

ASEL!
ALU with Immediate

I-type: ALU with constant operand
Reg[rt] ← Reg[rs] op SEXT(immediate)

Instruction Memory

ALU with Immediate

001XXX | rs | rt | immediate

Control Logic

BSEL

ALU

SEXT

BSEL

ALUFN

WERF

ASEL

Register File

Rs: <25:21>  Rt: <20:16>

Rd: <15:11>  Rt: <20:16>

imm: <15:0>

Rd: <15:11>  Rs: <25:21>

Shamt: <10:6>

SEXT

SEXT

SEXT

Figure 15-1: ALU with Immediate

Comp 411
Load Instruction

I-type: Load

\[
\text{Reg}[rt] \leftarrow \text{Mem}[\text{Reg}[rs] + \text{SEXT}(\text{immediate})]
\]
Store Instruction

I-type: Store
Mem[Reg[rs] + SEXT(immediate)] ← Reg[rt]

Control Logic

ALU

Data Memory

Register File

PC

Instruction Memory

+4
JMP Instructions

**00001X**

26-bit constant

J-type:

- **j**: $PC \leftarrow (PC \& 0xf0000000) \mid 4 \ast (\text{immediate})$
- **jal**: $PC \leftarrow (PC \& 0xf0000000) \mid 4 \ast (\text{immediate});$ \hspace{1cm} $\text{Reg}[31] \leftarrow PC + 4$

**PC<31:29>:J<25:0>:00**

PCSEL → 6 5 4 3 2 1 0

Instruction Memory

Control Logic

Register File

ALU

Data Memory

**Rt: <20:16>**

**Rs: <25:21>**

**Imm: <15:0>**

**J<25:0>**

**WASEL**

**PCSEL**

**WAS**

**SEXT**

**ALUFN**

**Wr**

**WERF**

**BSEL**

**ASEL**

**WR**

**RD**

**Addr**

**32**

**PC+4**

**00001X**

**26-bit constant**

- **j**: $PC \leftarrow (PC \& 0xf0000000) \mid 4 \ast (\text{immediate})$
- **jal**: $PC \leftarrow (PC \& 0xf0000000) \mid 4 \ast (\text{immediate});$ \hspace{1cm} $\text{Reg}[31] \leftarrow PC + 4$
BEQ/BNE Instructions

<table>
<thead>
<tr>
<th>10X011</th>
<th>( r_s )</th>
<th>( r_t )</th>
<th>immediate</th>
</tr>
</thead>
</table>

**R-type:** Branch Instructions
- if (Reg[rs] == Reg[rt]) \( PC \leftarrow PC + 4 + 4 \cdot \text{SEXT}(\text{immediate}) \)
- if (Reg[rs] != Reg[rt]) \( PC \leftarrow PC + 4 + 4 \cdot \text{SEXT}(\text{immediate}) \)

That "x4" unit is trivial. I'll just wire the input shifted over 2-bit positions.

Why add, another adder? Couldn't we reuse the one in the ALU? Nope, it needs to do a subtraction.
Jump Indirect Instructions

**R-type:** Jump Indirect, Jump and Link Indirect

- **jr:** PC ← Reg[rs]
- **jalr:** PC ← Reg[rs], Reg[rd] ← PC + 4

**Instruction Format:**

```
000000  r_s  r_t  r_d  00000  00100X
```

- **Rs:** <25:21>
- **Rt:** <20:16>
- **Imm:** <15:0>
- **Rd:** <15:11>
- **Rt:** <20:16>

**Control Logic**

- **PCSEL**
- **WASEL**
- **SEXT**
- **BSEL**
- **WDSEL**
- **ALUFN**
- **Wr**
- **WERF**
- **ASEL**

**Register File**

- **RA1**
- **WD**
- **WE**
- **WEF**

**ALU**

- **A SEL**
- **B SEL**
- **SHAMT:** <10:6>
- **Z**

**Data Memory**

- **Addr**
- **RD**

**Diagram Details:**

- **Jump Indirect:**
  - Instruction: PC ← Reg[rs]
  - Execution: PC ← PC + 4

- **Jump and Link Indirect:**
  - Instruction: PC ← Reg[rs], Reg[rd] ← PC + 4
Loose Ends

\[
\begin{array}{cccc}
001XXX & r_s & r_t & \text{immediate} \\
\end{array}
\]

1-type: set on less than & set on less than unsigned immediate

\[
sli: \quad \text{if } (\text{Reg}[r_s] < \text{SEXT}(\text{imm})) \text{ Reg}[r_t] \leftarrow 1; \text{ else } \text{Reg}[r_t] \leftarrow 0
\]

\[
sliu: \quad \text{if } (\text{Reg}[r_s] < \text{SEXT}(\text{imm})) \text{ Reg}[r_t] \leftarrow 1; \text{ else } \text{Reg}[r_t] \leftarrow 0
\]

Reminder:
To evaluate \((A < B)\) we first compute \(A-B\) and look at the flags.

\[
LT = N \oplus V
\]

\[
LTU = C
\]
I-type: Load upper immediate

lui: Reg[rt] ← Immediate << 16
Reset, Interrupts, and Exceptions

FIRST, we need some way to get our machine into a known initial state. This doesn’t mean that all registers will be initialized, just that we’ll know where to fetch the first instruction. We’ll call this control input, RESET.

We’d also like recoverable interrupts for

- Faults (e.g., Illegal Instruction)
  - CPU or system generated [synchronous]

- Traps & system calls (e.g., read-a-character)
  - CPU generated [synchronous]

  (Implemented as an “agreed upon” illegal instruction)

- I/O events (e.g., key struck)
  - Externally generated [asynchronous]

Exception goal: Interrupt running program, invoke exception handler, return to continue execution.

These are “Software” notions of synchrony and asynchrony.
Exceptions

Reset:  PC ← 0x80000000
Bad Opcode:  Reg[27] ← PC+4; PC ← 0x80000040
IRQ:  Reg[27] ← PC+4; PC ← 0x80000080
MIPS: Our Final Version

This is a complete 32-bit processor. It executes the majority of the MIPS R2000 instruction set.

- Executes one instruction per clock
- All that's left is the control logic design
MIPS Control

The control unit can be built as a large ROM

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RESET</th>
<th>IRQ</th>
<th>Z</th>
<th>N</th>
<th>V</th>
<th>C</th>
<th>PSEL</th>
<th>SELECT</th>
<th>WSEL</th>
<th>WDSEL</th>
<th>ALUFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>slt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>andi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</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>