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

- The MIPS instruction set as seen from a Hardware Perspective

<table>
<thead>
<tr>
<th>OP</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>func</td>
<td></td>
</tr>
</tbody>
</table>

**R-type:** ALU with Register operands
Reg[rd] ← Reg[rs] op Reg[rt]

<table>
<thead>
<tr>
<th>001XXX</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>10XXXX</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>0001XX</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

**I-type:** Branch Instructions
if (Reg[rs] == Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)
if (Reg[rs] != Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)

<table>
<thead>
<tr>
<th>00001X</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

**J-type:** jump
PC ← (PC & 0xf0000000) | 4*(immediate)
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
3. Load & Store Instructions
4. Jump & Branch instructions
5. Exceptions

Our Bag of Components:

- Registers
- Muxes

Incremental Featurism
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
Let's review the ALU that we built a few lectures ago. (With a few minor additions)

A Few ALU Tweaks

Flags
V, C

N Flag

R

Z Flag
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]
Shift Instructions

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

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

<table>
<thead>
<tr>
<th>100011</th>
<th>( r_s )</th>
<th>( r_t )</th>
<th>immediate</th>
</tr>
</thead>
</table>

l-type: Load

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

1

10X011 | \( r_s \) | \( r_t \) | immediate

I-type: Store

\[ \text{Mem}[\text{Reg}[rs] + \text{SEXT}(\text{immediate})] \leftarrow \text{Reg}[rt] \]
**JMP Instructions**

**J-type:**
- **j:** \( \text{PC} \leftarrow (\text{PC} \& 0x00000000) \mid 4*(\text{immediate}) \)
- **jal:** \( \text{PC} \leftarrow (\text{PC} \& 0x00000000) \mid 4*(\text{immediate}); \quad \text{Reg}[31] \leftarrow \text{PC} + 4 \)

**26-bit constant**

- **00001X**

---

**Control Logic**

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

**ALU**

- **A**
- **B**

**ALUFN**

- **Wr**
- **RD**

**Data Memory**

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

**Register File**

- **RA1**
- **WA**
- **RD1**

- **RA2**
- **WD**
- **WE**

**Instruction Memory**

- **PCSEL**
- **+4**

---

**Diagram Notes:**

- **PC<31:29>: J<25:0>: 00**
- **J: \( \text{PC} \leftarrow (\text{PC} \& 0x00000000) \mid 4*(\text{immediate}) \)**
- **jal: \( \text{PC} \leftarrow (\text{PC} \& 0x00000000) \mid 4*(\text{immediate}); \quad \text{Reg}[31] \leftarrow \text{PC} + 4 \)**

---

**Context:**

- **Comp 411**
- **L18- Building a Computer 18**
BEQ/BNE Instructions

R-type: Branch Instructions

if (Reg[rs] == Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)
if (Reg[rs] != Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)

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
Loose Ends

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

\[
\text{slt}: \quad \text{if } (\text{Reg}[rs] < \text{SEXT}(\text{imm})) \text{ Reg}[rt] \leftarrow 1; \text{ else } \text{Reg}[rt] \leftarrow 0
\]

\[
\text{sltiu}: \quad \text{if } (\text{Reg}[rs] < \text{SEXT}(\text{imm})) \text{ Reg}[rt] \leftarrow 1; \text{ else } \text{Reg}[rt] \leftarrow 0
\]

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

\[
\text{LT} = N \oplus V
\]
\[
\text{LTU} = C
\]
LUI

I-type: Load upper immediate
\[ \text{lui: } \text{Reg}[rt] \leftarrow \text{Immediate} \ll 16 \]

Instruction Memory

Register File

Control Logic

Data Memory
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 (eg, Illegal Instruction)
  - CPU or SYSTEM generated \([\text{[synchronous]}]\)
- TRAPS & system calls (eg, read-a-character)
  - CPU generated \([\text{[synchronous]}]\)

  (Implemented as an “agreed upon” Illegal instruction)

- I/O events (eg, key struck)
  - externally generated \([\text{[asynchronous]}]\)

EXCEPTION GOAL: Interrupt running program, invoke exception handler, return to continue execution.
Exceptions

Reset: \( \text{PC} \leftarrow 0x80000000 \)

Bad Opcode: \( \text{Reg}[27] \leftarrow \text{PC}+4; \ \text{PC} \leftarrow 0x80000040 \)

IRQ: \( \text{Reg}[27] \leftarrow \text{PC}+4; \ \text{PC} \leftarrow 0x80000080 \)
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

| Instruction | R | E | S | T | P | C | S | E | X | T | W | A | S | E | W | D | S | E | X | T | W | A | S | E | L | B | S | E | L |
| X           | 1 | X | X | X | X | X | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X           | 0 | 1 | X | X | X | X | 6 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| add         | 0 | 0 | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sll         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| andi        |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| lw          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| sw          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| beq         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |