Building a Computer

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

Instruction classes distinguished by types:
1) 3-operands ALU
2) ALU w/immediate
3) Loads/Stores
4) Branches
5) Jumps

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

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

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

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)

J-type: jump
PC ← (PC & 0xffffffff) | 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
3. Load & Store Instructions
4. Jump & Branch instructions
5. 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
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]
Shift Instructions

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

000000 rs rt rd shamt 000XXX

PC: 00
+4

Instruction Memory

ALU

Control Logic

Register File

Rs: <25:21>
Rd: <15:11>
Rt: <20:16>

ALUFN
WERF
ASEL

ASEL!
ALU with Immediate

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

Control Logic

ALU

Register File

001XX X  r_s  r_t  immediate

PC

+4

Instruction Memory

BSEL

Rs: <25:21>

Rt: <20:16>

Rd: <15:11>

Rt: <20:16>

imm: <15:0>

SEXT

shamt: <10:6>

ASEL

BSEL

WERF

ALUFN

Comp 411
**Load Instruction**

<table>
<thead>
<tr>
<th>100011</th>
<th>( r_s )</th>
<th>( r_t )</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I-type:</strong> Load</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{Reg}[rt] \leftarrow \text{Mem}[\text{Reg}[rs] + \text{SEXT}(\text{immediate})] )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of a computer architecture with instructions for load operation]
Store Instruction

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

00001X  26-bit constant

J-type:

j:  \( PC \leftarrow (PC \& 0xf0000000) | 4*(\text{immediate}) \)
jal: \( PC \leftarrow (PC \& 0xf0000000) | 4*(\text{immediate}); \)
\( \text{Reg}[31] \leftarrow PC + 4 \)

PC<31:29>:J<25:0>:00
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 \leftarrow \text{Reg}[rs]
jalr: $PC \leftarrow \text{Reg}[rs], \text{Reg}[rd] \leftarrow PC + 4
Loose Ends

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

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

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

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

\[
lui: \quad \text{Reg}[rt] \leftarrow \text{Immediate} \ll 16
\]
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: \( PC \leftarrow 0x80000000 \)

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

IRQ: \( \text{Reg}[27] \leftarrow \text{PC+4}; \ PC \leftarrow 0x80000080 \)
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
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>PCSEL</th>
<th>SEXT</th>
<th>WASL</th>
<th>WDSL</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>0</td>
<td>1</td>
</tr>
<tr>
<td>sll</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>