Building a Computer

I wonder where this goes?

Quiz #2 on 10/31, open book and notes
(This is the last lecture covered)
THIS IS IT!

“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.
# The MIPS ISA

- **The MIPS instruction set as seen from a Hardware Perspective**

## Instruction classes distinguished by types:

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

### R-type: ALU with Register operands

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

### I-type: ALU with constant operand

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

### I-type: Load and Store

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

### I-type: Branch Instructions

- \(\text{if (Reg}[rs] == \text{Reg}[rt])\) \(\text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{SEXT}(\text{immediate})\)
- \(\text{if (Reg}[rs] != \text{Reg}[rt])\) \(\text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{SEXT}(\text{immediate})\)

### J-type: Jump

\[
\text{PC} \leftarrow (\text{PC} \& 0xf0000000) \lor 4 \times (\text{immediate})
\]
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).

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

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)

Flags
V, C
N
R
Z
Flag

5-bit ALUFN

<table>
<thead>
<tr>
<th>Sub</th>
<th>Bool</th>
<th>Shift</th>
<th>Math</th>
<th>OP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XX</td>
<td>0</td>
<td>1</td>
<td>A+B</td>
</tr>
<tr>
<td>1</td>
<td>XX</td>
<td>0</td>
<td>1</td>
<td>A-B</td>
</tr>
<tr>
<td>X</td>
<td>X0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>B&lt;&lt;A</td>
</tr>
<tr>
<td>X</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>X</td>
<td>X1</td>
<td>1</td>
<td>0</td>
<td>B&gt;&gt;&gt;A</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>A ^ B</td>
</tr>
<tr>
<td>X</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>
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

[Diagram showing flow of instruction processing]
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
sllv: Reg[rd] ← Reg[rt] (shift) Reg[rs]
I-type: ALU with constant operand
Reg[rt] ← Reg[rs] op SEXT(immediate)
Load Instruction

I-type: Load

Reg[rt] ← Mem[Reg[rs] + SEXT(immediate)]

| 100011 | r_s | r_t | immediate |

Control Logic

 ALU

Data Memory

Register File

Instruction Memory

Adder

Control Logic

SEXT

ALUFN

BSEL

WDSEL

Wr

WERF

ASEL

SEXT

shamt:<10:6>

 WD

R/W

Wr

Addr

RD

32

32

01

Rd:<15:11>

Rt:<20:16>

Rs: <25:21>

Rt: <20:16>

Imm: <15:0>

Rd: <15:11>

Rt: <20:16>

BSEL

WDSEL

ALUFN

W

R/W

32

01

00
Store Instruction

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

Comp 411 – Fall 2007
10/24/07
**JMP Instructions**

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

- **00001X**
  - 26-bit constant

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

That “x4” unit is trivial. I'll just wire the input shifted over 2-bit positions.
Jump Indirect Instructions

- **R-type:** Jump Indirect, Jump and Link Indirect
  - **jr:** \( PC \leftarrow \text{Reg}[rs] \)
  - **jalr:** \( PC \leftarrow \text{Reg}[rs], \text{Reg}[rd] \leftarrow PC + 4 \)

**Instruction Format:**

| 000000 | \( r_s \) | \( r_t \) | \( r_d \) | 00000 | 00100X |

**Logic Diagram:**

- **Control Logic:**
  - JT
  - BT
  - Z

- **Register File:**
  - RA1
  - RA2
  - RD1
  - RD2

- **ALU:**
  - A
  - B
  - ALUFN
  - SEXT
  - Jx4
  - BSEL
  - ASEL
  - Wr

- **Data Memory:**
  - Addr
  - RD
  - Wr

- **Control Signals:**
  - PCSEL
  - WASEL
  - SEXT
  - BSEL
  - WDSEL
  - ALUFN
  - WERF
  - ASEL

**PC Calculation:**

- **Jump Indirect:**
  - \( \text{PC} + 4 \)

- **Jump and Link Indirect:**
  - \( \text{PC} \leftarrow \text{Reg}[rs], \text{Reg}[rd] \leftarrow \text{PC} + 4 \)
Loose Ends

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

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

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

\[
\text{Reminder: To evaluate (A < B) we first compute A-B and look at the flags.}
\]
\[
\text{LT = N \oplus V}
\]
\[
\text{LTU = C}
\]
More Loose Ends

R-type: set on less than & set on less than unsigned

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

\begin{align*}
\text{slt}: & \quad \text{if } (\text{Reg}[rs] < \text{Reg}[rt]) \text{Reg}[rd] \leftarrow 1; \text{else } \text{Reg}[rd] \leftarrow 0 \\
\text{sltu}: & \quad \text{if } (\text{Reg}[rs] < \text{Reg}[rt]) \text{Reg}[rd] \leftarrow 1; \text{else } \text{Reg}[rd] \leftarrow 0
\end{align*}
LUI Ends

Instruction Memory

Data Memory

Register File

Control Logic

ALU

PCSEL

WASEL

SEXT

BSEL

WDSEL

ALUFN

Wr

WERF

ASEL

001XXX 00000  \( r_t \)  Immediate

I-type: Load upper immediate

lui: \[ \text{Reg}[rt] \leftarrow \text{Immediate} \ll 16 \]
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
    - [synchronous]

- **TRAPS & system calls** (eg, read-a-character)
  - CPU generated
    - [synchronous]

  (Implemented as an “agreed” upon Illegal instruction)

- **I/O events** (eg, key struck)
  - externally generated
    - [asynchronous]

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

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

These inputs should probably be registered a few times to avoid metastability problems
MIPS: Our Final Version

This is a complete 32-bit processor. Although designed in one class lecture, 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>WASEL</th>
<th>WDSL</th>
<th>ALUFN</th>
<th>WR</th>
<th>WER</th>
<th>ASE</th>
<th>BSEL</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>00</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>00</td>
<td>0</td>
<td>0</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>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>
<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>
<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>
<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>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>