Topic 8: Instruction Formats & CPU Details

- Simple Datapaths
- R/I Instruction Formats
- Shared R/I Datapath
- Integrated Processor View
- Control
- Conditional Branches
- J Format
- Using the textbook to encode/decode
The Problem

- Need to encode many different instructions

**MIPS Limitations:**
- All instructions exactly 32 bits in size
- Want to maximize commonality of instructions
R Format Datapath

- "R format" (reg-reg):
  - 2 input registers, one output register
R Format Datapath

- R format instructions:
  - 2 input registers, one output register

Question:

How many instructions can we think of, which fit into the R-format style?
R Format Datapath

- R format instructions:
  - 2 input registers, one output register

- Example Instructions:
  - add/sub
  - or/and/xor/nor
  - slt

(We'll discover a few more later on)
Simplicity is good

• Ideally, the register control fields (3x5 bits) should always come from the **same bits** in the instruction
Simplicity is good

- Ideally, ALU operation should be easy to determine.
Simplicity is good

- All R-format instructions end up writing to a register.
R-Format Datapath
Simplicity is good

Instruction (32 bits)

Need to get the three 5-bit values, for the two input register numbers, and one output register number.

Remember, connecting some bits to another place is as simple as just running wires to another place.

Need to decode what operation is happening.
I Format Datapath

• “I format” (immediate values):
  – 16 bit immediate field
I Format Datapath

- I format instructions:
  - 16 bit immediate values
  - Lots of variety on the details

Question:
How many instructions can we think of, which might use a 16-bit immediate value as part of the calculation?
I Format Datapath

• I format instructions:
  - 16 bit immediate values
  - Lots of variety on the details

• Example Instructions:
  
  addi
  ori/andi/xori/nori
  lw/sw
  beq/bne
I Format Datapath

addi
I Format Datapath

- `addi`, `xori`, etc.
  - Read from a register

![Diagram of I Format Datapath]
I Format Datapath

- **addi, xori, etc.**
  - Other input is 16-bit immediate field, sign extended
I Format Datapath

- `addi`, `xori`, etc.
  - ALU output written back to the registers
I Format Datapath

- `addi`, `xori`, etc.
  - 2\textsuperscript{nd} input register ignored
I Format Datapath

- addi, xori, etc.
  - Input reg 2 doesn't matter.
addi Datapath
I Format Datapath

lw
I Format Datapath

- lw
  - ALU output is address
I Format Datapath

- $\text{l}_w$
  - Disable writes, **enable** read
I Format Datapath

- lw
  - Read from memory, store to register
I Format Datapath

- LW
  - Instruction needs to feed write register, input reg 1 to register file.
I Format Datapath

- $lw$
  - Input reg 2 doesn't matter.
lw Datapath
I Format Datapath

SW
I Format Datapath

- $SW$
  - ALU output is address
I Format Datapath

- $SW$
  - Disable read, enable **write**
I Format Datapath

- $SW$
  - 2\textsuperscript{nd} input register is written to memory
I Format Datapath

- **SW**
  - Disable RegWrite
SW Datapath
I Format Datapath

lw vs. Sw
Group Exercise:

Identify all of the differences between \( lw \) and \( sw \) in this datapath.
• \texttt{lw} vs. \texttt{sw}
  
  – 2\textsuperscript{nd} register input or output?
    
    • Is a register changed?
    
    • Is the 2\textsuperscript{nd} read register ignored or not?
  
  – Mem read or write?
• \texttt{lw} vs. \texttt{sw}

  - Still only need 2 register fields
    • Input / output
    • Input / input
lw, sw Datapath
I format (conceptual)

Instruction (32 bits)

- Need to decode what operation is happening.
- Only need two 5-bit fields.
- Need a 16-bit immediate field.
R/I Instruction Formats

- Problems:
  - Need to encode most-complex instruction in 32 bits
  - Need to be able to tell R-format from I-format instructions

Think/Pair/Share:

How many bits (minimum) do we need for R-format instructions? How about I-format?

(Ignore opcode for now.)
R/I Instruction Formats

- Problems:
  - Need to encode most-complex instruction in 32 bits
  - Need to be able to tell R-format from I-format instructions

- R-format:
  - Minimum 15 bits (3x5)

- I-format
  - Minimum 26 bits (16 + 2x5)
## I-Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td>16 bit number</td>
<td>5+5+6</td>
</tr>
</tbody>
</table>

- 16 bits for immediate field
- 2x5 bits for registers
- 6 bits left over for opcode
- $rs$ is the first input register
- Tied directly to input register 1
- Value goes directly to ALU input 1
The image shows a block diagram and a binary format for instructions.

### Binary Format

- **31**
- **I**
- `op` (6 bits)
- `rs` (5 bits)
- `rt` (5 bits)
- **16 bit number**

### Block Diagram

- **Instruction**
  - Read register 1
  - Read data 1
  - Write register
  - Write data
- **Registers**
  - Read register 2
  - Read data 2
- **ALU**
  - Zero
  - ALU result
  - ALU operation
- **Data memory**
  - Address
  - Read data
  - Write data
- **Sign extend**
  - `offset_value`
  - `extend`
  - `RegWrite`
  - MemRead
  - MemWrite
<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>op</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>16 bit number</td>
<td>10</td>
</tr>
</tbody>
</table>

- **rt** is the second register
- Sometimes used as 2\textsuperscript{nd} input, sometimes used as destination register
<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>bits</td>
<td>5</td>
<td>bits</td>
<td></td>
</tr>
</tbody>
</table>

Instruction

- Read register 1
- Read register 2
- Write register

ALU

- ALU result
- Zero
- Operation

Data Memory

- Read data
- Address
- Write data

RegWrite

Sign extend

MemRead

MemWrite
- **imm** is the immediate field
- Sign extended

  **NOTE:** Some instructions zero-extend. We'll ignore those for simplicity.

- Passed to 2\(^{nd}\) ALU input
- **opcode** tells CPU what instruction is running

- **Need unique opcodes for each of:**
  - `addi`
  - `lw`
  - `lb`  
    - Not the same opcode!
  - `sw`
  - Many others

- **Also need a way to tell I-format from R-format!**
R-Format

- **opcode** is in **same position** as I-format
  - That's how you tell them apart!
- $rs, rt$ in same position as in I-format
- Connect to same inputs as I-format

- Not technically necessary, but...
  - Simpler (fewer wires)
  - Faster (less logic means less delay)
• $rd$ is destination register
### Instruction Format

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

### ALU Operation Diagram

- **Instruction** flows into the diagram from the top left.
- **Registers** section includes:
  - Read register 1
  - Read register 2
  - Write register
  - Read data 1
  - Read data 2
  - Write data
- **RegWrite** flows from the bottom right of the registers to the top right of the ALU.
- **ALU operation** flows from the top right of the ALU to the zero result.
- **ALU result** flows from the zero result to the bottom right of the ALU.
- **shamt** is shift amount (only used in shift instructions)

- We'll ignore this (assume it's zero) in our simple CPU implementation
• **funct** is “function”

• Basically, a 2\textsuperscript{nd} opcode
  - Only used for R-format
  - Allow us to do **many** different instructions using a single R-format opcode
  - Most R-format instructions: opcode 0 (a few 1)
Oops! We overlooked some complexity...

**Question:**

Where does a `lw` instruction encode its **destination register**?

Where does an R-format instruction encode it?
• Oops! We overlooked some complexity...
  - `lw` destination register is `rt`
  - R-format destination register is `rd`
Shared Hardware

- R/I format use **different fields** to encode the destination register
- Need a **MUX** to choose between the two
  - Won't know the proper value to choose until we have inspected the opcode

**Group Exercise:**

On the next slide, compare R/I datapaths. What is common? What is different? (Ignore control bits.)

Where could we add MUXes to choose?
Common

$rs$, $rt$ fields always connect to same place: Read Registers 1, 2.
Read data 1 always connects to input 1 of the ALU.
Common
Read data 2 goes different places, but that's just connecting wires. It's the same source.

No MUX required.
Different
ALU input 2 can come from two different sources.

We'll use a MUX to choose between them.
Again, the ALU result goes to two different places, but with a single, common source.
Different

The data to write (to a register) can come from two different possible sources: ALU or memory.
Different

The number of the register to write might come from \( rt \) or \( rd \).
The Integrated CPU

• What does the CPU look like, when R-format and I-format instructions are integrated into a single datapath?
Lots of pieces in common.
MUXes make choices at key locations.
Now that we've seen the instruction formats, we see that certain wires of the instruction are connected to various inputs.
Control

• We have lots of control bits
  - ALU op
  - bInvert
  - Various MUXes we've added to the datapath
  - Some more that we'll see later...

• How do we choose the correct control bits?
  - Inspect opcode
  - R-format: also inspect funct
The **Control** component reads **opcode** (the top 6 bits).

It basically is a bunch of sum-of-product expressions, one for each output line.

Each of these lines is 1 bit, except for ALUOp.
The **ALU Control** component reads \texttt{funct} (bottom 6 bits), plus the 2-bit \texttt{ALUop} from the Control.

It basically just figures out the correct 3-bit control for the ALU.

We'll more or less consider it part of the **Control**, and ignore it.
PC+4

- Every clock cycle, our Program Counter must increment by one **word**.
- This requires a 32-bit adder
- Not the ALU (it's busy doing other things)
• Program Counter contains the address of the next instruction
• Automatically incremented by 4 bytes each clock.
• To **branch**, we use a MUX.

• But how does this work?
An Important Side Note...

beq $s1,$s2, LABEL

Important Note:

We never store labels in machine code.

Instead, the assembler translates labels to addresses during instruction encoding.
beq $s1,$s2, LABEL

Think/Pair/Share:

What are the values that need to be encoded for this instruction?

Addresses are 32-bits. Can we encode a 32-bit address in a single instruction?

Does this fit into either of the 2 formats we've seen before? Do we need a new one?
Branches

beq $s1,$s2, LABEL

Fields:
- 2x5 bits for register numbers
- Address to jump to (details coming)
Two Options for Addressing

Two common ways to encode addresses:

Absolute Addressing
  - Give the actual address

PC-Relative Addressing
  - Add an offset relative to the current program location
Absolute Addressing

• In theory, Absolute Addressing should allow you to jump anywhere in memory.

• MIPS compromises, and allows you to jump almost anywhere.
  – Details later (j / jal instructions)
PC-Relative Addressing

- PC-Relative addressing adds an offset (positive or negative) to the current program counter.

- (In MIPS) All instructions are word-aligned, so the offset is always a multiple of 4.
PC-Relative Addressing

- PC-Relative Addressing commonly used for loops and `if()`
  - Almost all branches are relatively short

- MIPS compromise:
  - Use 16-bit offset only (I-format)
  - But encode in **words not bytes** to maximize range
Conditional Branches

`beq $s1,$s2, LABEL`

- I-format instruction, but...
  - Need to read 2 registers, pass both to ALU
  - Compare using ALU (Zero output)
  - Immediate has to go elsewhere!
• *beq*
  - Reads two registers, feeds both into ALU
  - **So ALU src** used like an R-format!
- **beq**
  - Zero output from ALU indicates if registers were equal
- `beq`
  - Take the immediate field, and shift left by 2
  - Thus, we're adding **words** not bytes!
- **beq**
  - New adder, for calculating branch destination
  - Now there are 3 adders!
- `beq` - MUX to choose between PC+4 and branch destination
- **beq**
  - Branch if correct opcode AND Zero.
Conditional Branch Summary

- I-format
- Read $rs, rt$, pass both to ALU
- Control sets $Branch=1$ (based on opcode)

- if Zero AND $Branch$:
  
  $PC = PC+4 + 4*imm$

- else:
  
  $PC = PC+4$
Problem:

- Want to allow maximal-range, absolute addressing jumps.
- But instruction words are same size as address (32 bits)

Think/Pair/Share:

Pretend that you are designing the MIPS instruction set for the first time.

How would you go about solving this problem?
The MIPS solution: J-format instructions

- 6 bits for opcode (just like R/I formats)
- Everything else for address
• Still not perfect
  - Shift field left by two (words not bytes)
  - Total address size: 28 bits

• Solution
  - Top 4 bits copied from PC
  - Usually OK, few programs need more than 256 MB of code!
The bottom 26 bits of the instruction are shifted left, to make a 28-bit address.

Shifting left by 2 is just connecting wires! The bottom two wires are always zero...
The top 4 bits come from the PC.
We have a new MUX, and a new control bit, which allows us to select this operation.
If you absolutely need a 32-bit address field, use the \texttt{jr} instruction

- Step 1: Set a register (as many instructions as it takes)
- Step 2: \texttt{jr}

\textbf{Think/Pair/Share:}

What format should we use to encode a \texttt{jr} instruction?

How might it be implemented?
jr

- \textit{jr} only has a single register as input, could have been R/I format.
- MIPS chose R-format

- To implement, add another input to the Jump MUX – reading from the register.
Finding the Proper Opcode

• You will be encoding/decoding instructions in homework

• How do we find the proper opcode, funct for various instructions (or vice-versa)?
Addition (with overflow)

```
add rd, rs, rt  0   rs   rt   rd   0   0x20
  6   5   5   5   5   6
```

Addition (without overflow)

```
addu rd, rs, rt  0   rs   rt   rd   0   0x21
  6   5   5   5   5   6
```

Put the sum of registers $rs$ and $rt$ into register $rd$.

- Pages A-51 through A-81 of your textbook
  - List all instructions (more than we will use)
  - Show the proper values for each field.
### Addition (with overflow)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>0x20</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

```
add rd, rs, rt
```

### Addition (without overflow)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>0x21</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

```
addu rd, rs, rt
```

Put the sum of registers `rs` and `rt` into register `rd`.

- Shows the assembly language instruction
Finding the Proper Opcode

Addition (with overflow)

```
add rd, rs, rt
```

```
0  rs  rt  rd  0  0x20
6  5  5  5  5  6
```

Addition (without overflow)

```
addu rd, rs, rt
```

```
0  rs  rt  rd  0  0x21
6  5  5  5  5  6
```

Put the sum of registers rs and rt into register rd.

- Shows the arrangement of registers in the fields
Finding the Proper Opcode

Addition (with overflow)

```
add rd, rs, rt
```

Addition (without overflow)

```
addu rd, rs, rt
```

Put the sum of registers $rs$ and $rt$ into register $rd$.

- Gives required values
Addition (with overflow)

\[
\text{add rd, rs, rt}
\]

\[
\begin{array}{ccccccc}
0 & rs & rt & rd & 0 & 0x20 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Addition (without overflow)

\[
\text{addu rd, rs, rt}
\]

\[
\begin{array}{ccccccc}
0 & rs & rt & rd & 0 & 0x21 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Put the sum of registers \(rs\) and \(rt\) into register \(rd\).

- Reminds you about field sizes
Decoding Opcodes

• What if you are going from binary to assembly?

• See the table on page A-50
- Left column is opcode
• Opcodes are given in both decimal and hex
• I/J format instructions will be listed in this column
• R-format instructions link to a 2\textsuperscript{nd} column, with \texttt{func}t field values.

• For some reason, these are only in decimal.
<table>
<thead>
<tr>
<th>Register name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0</td>
<td>2</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$v1</td>
<td>3</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$a0</td>
<td>4</td>
<td>argument 1</td>
</tr>
<tr>
<td>$a1</td>
<td>5</td>
<td>argument 2</td>
</tr>
<tr>
<td>$a2</td>
<td>6</td>
<td>argument 3</td>
</tr>
<tr>
<td>$a3</td>
<td>7</td>
<td>argument 4</td>
</tr>
<tr>
<td>$t0</td>
<td>8</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t1</td>
<td>9</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t2</td>
<td>10</td>
<td>temporary (not preserved across call)</td>
</tr>
</tbody>
</table>

• Page A-24 has a handy table of registers (same as in our slides)
Using these two tables, you can either **decode** or **encode**.

Expect to have **only** these tables on a test.