Review

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use $\text{lw}$ and $\text{sw}$).
- Computer actually stores programs as a series of these 32-bit numbers.
- MIPS Machine Language Instruction: 32 bits representing a single instruction

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>
I-Format Problem (1/3)

Problem:

- Chances are that `addi, lw and sw` will use immediates small enough to fit in the immediate field.

- ...but what if it’s too big? For example, what is the MIPS assembly code to load this 32-bit constant into register $s0:
  
  \[
  \begin{align*}
  0000 & 0000 0011 1101 0000 1001 0000 0000 \\
  \end{align*}
  \]

- We need a way to deal with a 32-bit immediate in any I-format instruction.
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  
  ```
lui rt, constant
  ```
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lhi $s0, 61
0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304
0000 0000 0111 1101 0000 0000 0000 0000 1001 0000 0000 0000
```
Solution to Problem:
- Handle it in software + new instruction
- Don’t change the current instructions: instead, add a new instruction to help out

New instruction:
\[
\text{lui \ register, immediate}
\]
- stands for Load Upper Immediate
- takes 16-bit immediate and puts these bits in the upper half (high order half) of the register
- sets lower half to 0s
I-Format Problems (3/3)

- Solution to Problem (continued):
  - So how does \texttt{lui} help us?
  - Example:
    \begin{verbatim}
    addiu $t0,$t0, 0xABABCDCD
    \end{verbatim}
    ...becomes
    \begin{verbatim}
    lui $at 0xABAB
    ori $at, $at, 0xCDCD
    addu $t0,$t0,$at
    \end{verbatim}
  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn’t it be nice if the assembler would do this for us automatically? (later)
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;

- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;

- `j L1`
  - unconditional jump to instruction labeled L1
More Conditional Operations

• Set result to 1 if a condition is true
  • Otherwise, set to 0
• `slt rd, rs, rt`
  • if (rs < rt) rd = 1; else rd = 0;
• `slti rt, rs, constant`
  • if (rs < constant) rt = 1; else rt = 0;

• Use in combination with `beq, bne`
  `slt $t0, $s1, $s2  # if ($s1 < $s2)`
  `bne $t0, $zero, L  # branch to L`
Compiling If Statements

• C code:
  
  ```
  if (i==j) f = g+h;
  else f = g-h;
  ```

  • f, g, ... in $s0, $s1, ...

• Compiled MIPS code:

  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

Assembler calculates addresses
Compiling Loop Statements

• C code:

```c
while (save[i] == k) {
    i += 1;
}
```

• i in $s3$, k in $s5$, address of save in $s6$
Compiled MIPS code:

Loop:  sll  $t1, $s3, 2  
      add  $t1, $t1, $s6  
      lw   $t0, 0($t1)  
      bne  $t0, $s5, Exit  
      addi $s3, $s3, 1  
      j    Loop  

Exit:  ...
Branches: PC-Relative Addressing (1/5)

- Use I-Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

- `opcode` specifies `beq` versus `bne`
- `rs` and `rt` specify registers to compare
- What can `immediate` specify?
  - `immediate` is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory
  - So `immediate` cannot specify entire address to branch to.
Branches: PC-Relative Addressing (2/5)

- How do we typically use branches?
  - **Answer:** `if-else, while, for`
  - Loops are generally small: usually up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.

- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount
Solution to branches in a 32-bit instruction: PC-Relative Addressing

Let the 16-bit immediate field be a signed two's complement integer to be added to the PC if we take the branch.

Now we can branch ± 2^{15} bytes from the PC, which should be enough to cover almost any loop.

Any ideas to further optimize this?
Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they’re word aligned (byte address is always a multiple of 4, which means it ends with 000 in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the immediate in words.

- Now, we can branch $\pm 2^{15}$ words from the PC (or $\pm 2^{17}$ bytes), so we can handle loops 4 times as large.
Branch Calculation:

- If we don’t take the branch:
  \[ \text{PC} = \text{PC} + 4 = \text{byte address of next instruction} \]

- If we do take the branch:
  \[ \text{PC} = (\text{PC} + 4) + (\text{immediate} \times 4) \]

Observations

- \text{Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.}
- \text{Immediate field can be positive or negative.}
- \text{Due to hardware, add \text{immediate} to (PC+4), not to PC; will be clearer why later in course}
Branch Example (1/3)

- **MIPS Code:**
  
  ```mips
  Loop: beq $9, $0, End
  addu $8, $8, $10
  addiu $9, $9, -1
  j Loop
  
  End:
  ```

- **beq branch is I-Format:**
  
  - opcode = 4 (look up in table)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - immediate = ???

Dr. Dan Gracia
Branch Example (2/3)

- **MIPS Code:**
  
  Loop: beq $9,$0,End
  addu $8,$8,$10
  addiu $9,$9,−1
  j Loop

  End:

- **immediate Field:**
  
  ▫ Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.
  
  ▫ In beq case, immediate = 3
Branch Example (3/3)

- **MIPS Code:**

  Loop: `beq $9, $0, End`
  `addu $8, $8, $10`
  `addiu $9, $9, -1`
  `j Loop`

  End:

  decimal representation:

<table>
<thead>
<tr>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
</table>

  binary representation:

  | 000100 | 01001 | 000000 | 0000000000000000011 |
Questions on PC-addressing

 Does the value in branch field change if we move the code?

 What do we do if destination is > $2^{15}$ instructions away from branch?
Branching Far Away

• If branch target is too far to encode with 16-bit offset, assembler rewrites the code

• Example

```
beq $s0,$s1, L1
↓
bne $s0,$s1, L2
j L1
L2: ...
```
For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.

For general jumps ($j$ and $jal$), we may jump to anywhere in memory.

Ideally, we could specify a 32-bit memory address to jump to.

Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.
J-Format Instructions (2/5)

- Define two “fields” of these bit widths:
  
  | 6 bits | 26 bits |
  
- As usual, each field has a name:
  
  | opcode | target address |
  
- Key Concepts
  
  - Keep `opcode` field identical to R-format and I-format for consistency.
  
  - Collapse all other fields to make room for large target address.
J-Format Instructions (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let’s just take this for granted and not even specify them.
Now specify 28 bits of a 32-bit address

Where do we get the other 4 bits?

- By definition, take the 4 highest order bits from the PC.
- Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999…% of the time, since programs aren’t that long
  - only if straddle a 256 MB boundary
- If we absolutely need to specify a 32-bit address, we can always put it in a register and use the \texttt{jr} instruction.
Summary:
- New PC = \{ PC[31..28], target address, 00 \}

Understand where each part came from!

Note: \{ , , \} means concatenation
\{ 4 \text{ bits}, 26 \text{ bits}, 2 \text{ bits} \} = 32 \text{ bit address}
- \{ 1010, 11111111111111111111111111, 00 \} = \begin{align*}
10101111111111111111111111111111111100
\end{align*}

Note: Book uses \|\
When combining two C files into one executable, recall we can compile them independently & then merge them together.

1) **Jump** insts don’t require any changes.
2) **Branch** insts don’t require any changes.

Dr. Dan Gracia
Branch Addressing

• Branch instructions specify
  • Opcode, two registers, target address

• Most branch targets are near branch
  • Forward or backward

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **PC-relative addressing**
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = PC\_31...28 \times (address \times 4)
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop:  sll  $t1, $s3, 2  80000
      add  $t1, $t1, $s6  80004
      lw   $t0, 0($t1)    80008
      bne  $t0, $s5, Exit 80012
      addi $s3, $s3, 1    80016
      j    Loop            80020
Exit:  ...              80024
```
### Solution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Addressing Mode Summary

1. Immediate addressing
   \[
   \begin{array}{c c c c}
   \text{op} & \text{rs} & \text{rt} & \text{Immediate}
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{c c c c c}
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{funct}
   \end{array}
   \]
   Registers
   Register

3. Base addressing
   \[
   \begin{array}{c c c c}
   \text{op} & \text{rs} & \text{rt} & \text{Address}
   \end{array}
   \]
   Register
   +
   Memory
   Byte | Halfword | Word

4. PC-relative addressing
   \[
   \begin{array}{c c c c}
   \text{op} & \text{rs} & \text{rt} & \text{Address}
   \end{array}
   \]
   PC
   +
   Memory
   Word

5. Pseudodirect addressing
   \[
   \begin{array}{c c}
   \text{op} & \text{Address}
   \end{array}
   \]
   PC
   :
   Memory
   Word