Lecture 9:
Instruction Representation II
• **Problem:**

  • Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field.
  
  • …but what if it’s too big?
  
  • We need a way to deal with a 32-bit immediate in any I-format instruction.
I-Format Problems (2/3)

• Solution:
  • 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 specified register
  • sets lower half to 0s
Solution (continued):

- So how does \texttt{lui} help us?

Example:

\begin{verbatim}
addi $t0,$t0, 0xABABCDCD
\end{verbatim}

becomes:

\begin{verbatim}
lui $at, 0xABAB
ori $at, $at, 0xCD
add $t0,$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? Yes!
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 v. 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 usually use branches?
  - Answer: if-else, while, for
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (j and jal), not branches.

- Conclusion: may want to jump 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 00 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.
Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don’t take the branch:
    \[ PC = PC + 4 \]
    \[ PC + 4 = \text{byte address of next instruction} \]
  - If we do take the branch:
    \[ PC = (PC + 4) + (\text{immediate} \times 4) \]

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

• MIPS Code:

    Loop:   beq   $9,$0,End
            add   $8,$8,$10
            addi  $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 = ???
Branch Example (2/3)

• MIPS Code:

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

decimal representation:

```
4  9  0  3
```

binary representation:

```
000100 01001 00000 00000000000000011
```
• 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 “fields” of the following number of bits each:

| 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.

  • Combine all other fields to make room for large target address.
• 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.
J-Format Instructions (4/5)

• Now specify 28 bits of a 32-bit address

• Where do we get the other 4 bits?
  • Use the 4 highest order bits from the PC.
  • Technically, this means that we cannot jump to *anywhere* in memory, but it’s adequate most 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 *jr* instruction.
• Summary:
  • New PC = { PC[31..28], target address, 00 }

• Understand where each part came from!

• Note: { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  • { 1010, 11111111111111111111111111111111, 00 }
    = 101011111111111111111111111111100
In conclusion...

- **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>
</table>

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

<table>
<thead>
<tr>
<th>J</th>
<th>opcode</th>
<th>target address</th>
</tr>
</thead>
</table>

- Branches use PC-relative addressing, Jumps use absolute addressing.

- Disassembly is simple and starts by decoding opcode field.