Alternatives to Immediates

° Are immediates really needed?
° Is it possible to invent an ISA (instruction set architecture) without immediates?
° If so, what would be the consequences?
I-Format Problem (1/3)

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 Problem (2/3)

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:

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
I-Format Problems (3/3)

Solution to Problem (continued):

• So how does \textit{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? (later)

Branches: PC-Relative Addressing (1/5)

• Use I-Format

\begin{table}[h]
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{opcode} & \textbf{rs} & \textbf{rt} & \textbf{immediate} \\
\hline
\end{tabular}
\end{table}

• \textit{opcode} specifies \textit{beq} versus \textit{bne}

• \textit{rs} and \textit{rt} specify registers to compare

• What can \textit{immediate} specify?

  • Immediate is only 16 bits

  • \textbf{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

Branches: PC-Relative Addressing (3/5)

- 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:

\[
\text{Loop: } \begin{align*}
\text{beq} & \quad 9, 0, \text{End} \\
\text{add} & \quad 8, 8, 10 \\
\text{addi} & \quad 9, 9, -1 \\
\text{j} & \quad \text{Loop}
\end{align*}
\]

\[
\text{End:}
\]

\[
\text{beq branch is I-Format:}
\]

\[
\text{opcode} = 4 \text{ (look up in table)} \\
\text{rs} = 9 \text{ (first operand)} \\
\text{rt} = 0 \text{ (second operand)} \\
\text{immediate} = ???
\]

Branch Example (2/3)

MIPS Code:

\[
\text{Loop: } \begin{align*}
\text{beq} & \quad 9, 0, \text{End} \\
\text{add} & \quad 8, 8, 10 \\
\text{addi} & \quad 9, 9, -1 \\
\text{j} & \quad \text{Loop}
\end{align*}
\]

\[
\text{End:}
\]

\[
\text{Immediate Field:}
\]

- 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:

\[
\begin{align*}
\text{Loop:} & \quad \text{beq} \quad $9,$0,End \\
& \quad \text{addi} \quad $8,$8,$10 \\
& \quad \text{addi} \quad $9,$9,-1 \\
& \quad j \quad \text{Loop} \\
\text{End:}
\end{align*}
\]

decimal representation:

\[
\begin{array}{cccc}
4 & 9 & 0 & 3
\end{array}
\]

binary representation:

\[
\begin{array}{cccccccccccccccc}
000100 & 01001 & 0000 & 000000000000011
\end{array}
\]

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?

° Why do we need different addressing modes (different ways of forming a memory address)? Why not just one?
J-Format Instructions (1/5)

° 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.
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.

J-Format Instructions (4/5)

° 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 jr instruction.
J-Format Instructions (5/5)

- **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
  
  \begin{align*}
  \{ 1010, 11111111111111111111111111, 00 \} \\
  = 10101111111111111111111111111100 \\
  \end{align*}

- **Note:** Book uses II

Quiz

(for A,B) When combining two C files into one executable, recall we can compile them independently & then merge them together.

- A. Jump insts don’t require any changes.
- B. Branch insts don’t require any changes.
- C. You now have all the tools to be able to “decompile” a stream of 1s and 0s into

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>FF</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>TT</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>TF</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>TT</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>TF</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>TT</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>TT</td>
</tr>
</tbody>
</table>
In conclusion...

- **MIPS Machine Language Instruction:** 32 bits representing a single instruction

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>J</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- Disassembly is simple and starts by decoding opcode field. (more in a week)