Lecture 8: Instruction Representation
Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).

- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs
Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: “Program Counter” (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name
Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintosh and IBM PC
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today
Instructions as Numbers

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - \texttt{lw} and \texttt{sw} both access memory one word at a time.

- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so “add $t0, $0, $0” is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words too.
Instructions as Numbers

- One word is 32 bits, so divide instruction word into “fields”
- Each field tells computer something about instruction
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format
Instruction Formats

- **I-format**: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- **J-format**: used for `j` and `jal`
- **R-format**: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way
R–Format Instructions

- Define “fields” of the following number of bits each: \(6 + 5 + 5 + 5 + 5 + 6 = 32\)

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

- For simplicity, each field has a name:

<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>
</table>

- **Important**: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.

  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.
What do these field integer values tell us?

- **opcode**: partially specifies what instruction it is
  - Note: This number is equal to 0 for all R-Format instructions.
- **funct**: combined with opcode, this number exactly specifies the instruction
- Question: Why aren’t **opcode** and **funct** a single 12-bit field?
  - Answer: We’ll answer this later
R–Format Instructions

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

- **More fields:**
  - **rs** *(Source Register):* *generally* used to specify register containing first operand
  - **rt** *(Target Register):* *generally* used to specify register containing second operand (note that name is misleading)
  - **rd** *(Destination Register):* *generally* used to specify register which will receive result of computation
R–Format Instructions

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word “generally” was used because there are exceptions that we’ll see later. E.g.,
    - `mult` and `div` have nothing important in the `rd` field since the dest registers are `hi` and `lo`
    - `mfhi` and `mflo` have nothing important in the `rs` and `rt` fields since the source is determined by the instruction
R–Format Instructions

- Final field:
  - **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is redundant, so this field is only 5 bits (so it can represent the numbers 0-31).
  - This field is set to 0 in all but the shift instructions.

- For a detailed description of field usage for each instruction, see back inside cover of P&H textbook.
R-Format Example

- MIPS Instruction:
  \[
  \text{add} \quad \$8, \$9, \$10
  \]

  opcode = 0 (look up in table in book)
  funct = 32 (look up in table in book)
  rs = 9 (first \textit{operand})
  rt = 10 (second \textit{operand})
  rd = 8 (destination)
  shamt = 0 (not a shift)
R–Format Example

- **MIPS Instruction:**
  \[
  \text{add} \quad \$8, \$9, \$10
  \]
  Decimal number per field representation:
  \[
  \begin{array}{ccccccc}
    0 & 9 & 10 & 8 & 0 & 32 \\
  \end{array}
  \]
  Binary number per field representation:
  \[
  \begin{array}{cccccccc}
    000000 & 01001 & 01010 & 01000 & 00000 & 100000
  \end{array}
  \]
  hex representation: \(012A\ 4020_{\text{hex}}\)
  decimal representation: \(19,546,144_{\text{ten}}\)

- Called a **Machine Language Instruction**
I–Format Instructions

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise

- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.
I–Format Instructions

- Define “fields” of the following number of bits each: $6 + 5 + 5 + 16 = 32$ bits

| 6 | 5 | 5 | 16 |

- Again, each field has a name:

| opcode | rs | rt | immediate |

- **Key Concept**: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.
I–Format Instructions

- What do these fields mean?
  - **opcode**: same as before except that, since there’s no **funct** field, opcode uniquely specifies an instruction in I-format.
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent with other formats.
I–Format Instructions

More fields:

- **rs**: specifies the *only* register operand (if there is one)
- **rt**: specifies register which will receive result of computation (this is why it’s called the *target* register “rt”)

The Immediate Field:

- `addi, slti, sltiu, the immediate is sign-extended to 32 bits. Thus, it’s treated as a signed integer.
- 16 bits can be used to represent immediate up to $2^{16}$ different values
- This is large enough to handle the offset in a typical `lw` or `sw`, plus a vast majority of values that will be used in the `slti` instruction.
I–Format Example

- MIPS Instruction:
  \[
  \text{addi} \quad $21, $22, -50
  \]

  \[
  \begin{align*}
  \text{opcode} & = 8 \text{ (look up in table in book)} \\
  \text{rs} & = 22 \text{ (register containing operand)} \\
  \text{rt} & = 21 \text{ (target register)} \\
  \text{immediate} & = -50 \text{ (by default, this is decimal)}
  \end{align*}
  \]
I-Format Example

- **MIPS Instruction:**
  ```
  addi $21, $22, -50
  ```

  **Decimal/field representation:**
  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>22</td>
<td>21</td>
<td>-50</td>
</tr>
</tbody>
</table>

  **Binary/field representation:**
  
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  | 001000 | 10110 | 10101 | 111111111111001110 |

  **hexadecimal representation:** 22D5 FFCE\text{hex}

  **decimal representation:** 584,449,998\text{ten}
I–Format Problems

- Problem 1:
  - Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field.
  - What if too big?
    - We need a way to deal with a 32-bit immediate in *any* I-format instruction.
I–Format Problems

- Solution to Problem 1:
  - Handle it in software + new instruction
  - Don’t change the current instructions: instead, add a new instruction to help out

- New instruction:
  - \texttt{lui register, immediate}
  - stands for \texttt{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 to Problem 1 (continued):

- So how does `lui` help us?

Example:

```assembly
addi $t0,$t0, 0xABABCD
```

becomes:

```assembly
lui $at, 0xABAB
ori $at, $at, 0xCD
add $t0,$t0,$at
```

- Now each I-format instruction has only a 16-bit immediate.

- Wouldn’t it be nice if the assembler would this for us automatically? (later)
Branches: PC–Relative Addressing

- 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
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 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

- 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

- 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 +/- $2^{15}$ words from the PC (or +/- $2^{17}$ bytes), so we can handle loops 4 times as large.
Branches: PC–Relative Addressing

- **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.
Branch Example

- **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

- **MIPS Code:**
  - Loop: beq $9, $0, End
    - add $8, $8, $10
    - addi $9, $9, -1
  - j Loop

- **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

- **MIPS Code:**
  - Loop:  `beq $9, $0, End`
    `add $8, $8, $10`
    `addi $9, $9, -1`
    `j Loop`
  - End:

  **decimal representation:**
  
  | 4 | 9 | 0 | 3 |

  **binary representation:**
  
  000100 01001 00000 000000000000000011
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?
- Since its limited to $\pm 2^{15}$ instructions, doesn’t this generate lots of extra MIPS instructions?
- Why do we need all these addressing modes? Why not just one?
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

- 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.
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

- **Summary:**
  - New PC = PC[31..28] || target address || 00

- **Note:** || means concatenation
  - 4 bits || 26 bits || 2 bits = 32 bit address
    - 1010 || 11111111111111111111111111 || 00 = 10101111111111111111111111111100
Conclusion

- Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `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></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>