Overview – Instruction Representation

° Big idea: stored program
  • consequences of stored program
° Instructions as numbers
° Instruction encoding
° MIPS instruction format for Add instructions
° MIPS instruction format for Immediate, Data transfer instructions
Big Idea: Stored-Program Concept

° Computers built on 2 key principles:

  1) Instructions are represented as bit patterns - can think of these as numbers.

  2) Therefore, entire programs can be stored in memory to be read or written just like 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, 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 Macintoshs and PCs
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backward compatible” 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 (1/2)**

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - `lw` and `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 (2/2)

° One word is 32 bits, so divide instruction word into “fields”.

° Each field tells processor 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 (1/5)

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

| 6 | 5 | 5 | 5 | 5 | 6 |

° For simplicity, each field has a name:

| opcode | rs | rt | rd | shamt | funct |

° 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.

R-Format Instructions (2/5)

° 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 (3/5)

° 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 (4/5)

° 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 (p. 264 P&H)
R-Format Instructions (5/5)

Final field:
- **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, 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 green insert in COD 3/e
- (You can bring with you to all exams)

R-Format Example (1/2)

MIPS Instruction:
```
add $8, $9, $10
```

- **opcode** = 0 (look up in table in book)
- **funct** = 32 (look up in table in book)
- **rd** = 8 (destination)
- **rs** = 9 (first operand)
- **rt** = 10 (second operand)
- **shamt** = 0 (not a shift)
R-Format Example (2/2)

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

Decimal number per field representation:

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>10</th>
<th>8</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</table>

Binary number per field representation:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

hex representation: \[ 012A \ 4020_{\text{hex}} \]

decimal representation: \[ 19,546,144_{\text{ten}} \]

• Called a Machine Language Instruction

Administrivia
I-Format Instructions (1/4)

° 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 (2/4)

° 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 (3/4)**

- **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 as possible with other formats while leaving as much space as possible for immediate field.
  - **rs**: specifies a 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”) or other operand for some instructions.

**I-Format Instructions (4/4)**

- **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.
  - We’ll see what to do when the number is too big in our next lecture...
I-Format Example (1/2)

° MIPS Instruction:
addi $21,$22,-50

opcode = 8 (look up in table in book)
rs = 22 (register containing operand)
rt = 21 (target register)
immediate = -50 (by default, this is decimal)

I-Format Example (2/2)

° MIPS Instruction:
addi $21,$22,-50

Decimal/field representation:

<table>
<thead>
<tr>
<th>8</th>
<th>22</th>
<th>21</th>
<th>-50</th>
</tr>
</thead>
</table>

Binary/field representation:

| 001000 | 10110 | 10101 | 1111111111001110 |

hexadecimal representation: 22D5 FFCE hex
decimal representation: 584,449,998 ten
Quiz

Which instruction has same representation as $35_{ten}$?

1. add $0$, $0$, $0$
2. subu $s0$, $s0$, $s0$
3. lw $0$, 0($0)$
4. addi $0$, $0$, 35
5. subu $0$, $0$, $0$
6. Trick question!

Instructions are not numbers

Registers numbers and names:
0: $s0$, .. 8: $s10$, 9:$s11$, ..15: $s17$, 16: $s1$, 17: $s1$, .. 23: $s7$

Opcodes and function fields (if necessary)

add: opcode = 0, funct = 32
subu: opcode = 0, funct = 35
addi: opcode = 8
lw: opcode = 35

In conclusion...

° Simplifying MIPS: 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