
CS 430 Computer Architecture

Instruction Representation

William J. Taffe

using slides of

David Patterson

Review

- ° Instructions:

add, addi, sub, lw, sw

beq, bne, j

slt, slti, sltu, sltiu

- ° C Decisions are made using **conditional statements**, MIPS Decision making instructions are the **conditional branches**: beq and bne.

- ° For inequalities, we introduce a single instruction: “Set on Less Than” called slt, slti, sltu, sltiu

Overview

- **Instructions as Numbers**
- **R-Format**
- **I-Format**

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...**

Instructions as Numbers (2/2)

- **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

- **J-format: used for `j` and `jal`**
- **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)**
- **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
---	---	---	---	---	---

° For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

° **Important:** 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, such as:
 - `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 (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 back cover of textbook.

R-Format Example (1/2)

° MIPS Instruction:

`add $8, $9, $10`

`opcode = 0` (look up in table)

`funct = 32` (look up in table)

`rs = 9` (first *operand*)

`rt = 10` (second *operand*)

`rd = 8` (destination)

`shamt = 0` (not a shift)

R-Format Example (2/2)

° MIPS Instruction:

add \$8, \$9, \$10

decimal representation:

0	9	10	8	0	32
---	---	----	---	---	----

binary representation:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

Called a Machine Language Instruction

I-Format Instructions (1/5)

- **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 (2/5)

- Define “fields” of the following number of bits each:

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 (3/5)

◦ What do these fields mean?

- **opcode**: same as before except that, since there's no `funct` field, `opcode` uniquely specifies an I-format instruction
- 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 (4/5)

◦ 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")

I-Format Instructions (5/5)

◦ The Immediate Field:

- `addi`, `slti`, `slltu`, 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 (1/2)

° MIPS Instruction:

`addi $21, $22, -50`

`opcode = 8` (look up in table)

`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 representation:

8	22	21	-50
---	----	----	-----

binary representation:

001000	10110	10101	11111111111001110
--------	-------	-------	-------------------

I-Format Problems (1/3)

° Problem 1:

- Chances are that `addi`, `lw`, `sw` and `slli` 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 (2/3)

◦ Solution to Problem 1:

- Handle it in software
- 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 1 (continued):

- So how does `lui` help us?

- Example:

```
addi    $t0, $t0, 0xABABCDCD
```

becomes:

```
lui     $at, 0xABAB
ori     $at, $at, 0xCDCD
add     $t0, $t0, $at
```

- Now each I-format instruction has only a 16-bit immediate.
- An instruction that must be broken up is called a **pseudoinstruction**. (Note that `$at` was used in this code.)

Branches: PC-Relative Addressing (1/5)

◦ Use I-Format

opcode	rs	rt	immediate
--------	----	----	-----------

- opcode specifies beq v. bne
- Rs and Rt specify registers to compare
- What can immediate specify?
 - Immediate is only 16 bits
 - PC is 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 the branches.**
- **Conclusion: Though we may want to branch to anywhere in memory, a single branch will generally change the **PC** by a very small amount.**

Branches: PC-Relative Addressing (3/5)

- **Solution: 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 $\pm 2^{15}$ bytes from the PC, which should be enough to cover 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)

° Final Calculation:

- If we don't take the branch:

$$PC = PC + 4$$

- If we do take the branch:

$$PC = (PC + 4) + (\text{immediate} * 4)$$

• 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:
```

° 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 this case, `immediate = -4`

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	-4
---	---	---	----

binary representation:

000100	01001	00000	111111111111111100
--------	-------	-------	--------------------

Big Idea: 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, which is better

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 II); could still run program from 1981 PC today**

Things to Remember

- **Simplifying MIPS: Define instructions to be same size as data (one word) so that they can use the same memory (can use lw and sw).**
- **Machine Language Instruction: 32 bits representing a single instruction**

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

- **Computer actually stores programs as a series of these.**