

Lecture #9: MIPS Instruction Format



2005-07-05

Andy Carle



Big Idea: Stored-Program Concept

Computers built on 2 key principles:

- 1) Instructions are represented as data.
- 2) Therefore, entire programs can be stored in memory to be read or written just like data.



Consequence: Everything Addressed

- Everything has a memory address: instructions, data words
- 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
 - Computer "brain" executes the instruction at PC
 - Jumps and branches modify PC



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 computer something about instruction.
- 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



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.

5-bit fields → 0-31, 6-bit fields → 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 for R-Format instructions



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`

- `mflhi` and `mfllo` 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)

`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 number per field representation:

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

Binary number per field representation:

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

hex representation: 012A 4020_{hex}

decimal representation: 19,546,144_{ten}

- Called a [Machine Language Instruction](#)



I-Format Instructions (1/4)

- What about instructions with immediates (e.g. addi and lw)?

- 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:

- Notice that, if instruction has an 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 with other formats.

- `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 (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.



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:

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

- Binary/field representation:

001000	10110	10101	111111111001110
--------	-------	-------	-----------------

hexadecimal representation: 22D5 FFCE_{hex}

decimal representation: 584,449,998_{ten}



CS 61C L09 Instruction Format (19)

A Carls, Summer 2005 © UCB

I-Format Problems (0/3)

- Problem 0: Unsigned # sign-extended?

- addiu, sltiu, **sign-extends** immediates to 32 bits. Thus, # is a “signed” integer.

- Rationale

- addiu so that can add w/out overflow
 - See K&R pp. 230, 305
- sltiu suffers so that we can have ez HW
 - Does this mean we'll get wrong answers?
 - Nope, it means assembler has to handle any unsigned immediate $2^{15} \leq n < 2^{16}$ (i.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large. \Rightarrow



CS 61C L09 Instruction Format (20)

A Carls, Summer 2005 © UCB

I-Format Problems (1/3)

- Problem 1:

- 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.



CS 61C L09 Instruction Format (21)

A Carls, Summer 2005 © UCB

I-Format Problems (2/3)

- 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:

```
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



CS 61C L09 Instruction Format (22)

A Carls, Summer 2005 © UCB

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.
- Wouldn't it be nice if the assembler would this for us automatically? (later)



CS 61C L09 Instruction Format (23)

A Carls, Summer 2005 © UCB

J-Format Instructions (0/5)

Jumps modify the PC:

“j <label>”

means

“Set the next PC = the address of the instruction pointed to by <label>”



CS 61C L09 Instruction Format (24)

A Carls, Summer 2005 © UCB

J-Format Instructions (1/5)

Jumps modify the PC:

- j and jal jump to labels
- but a label is just a name for an address!
- so, the ML equivalents of j and jal use addresses
 - 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:



CS 61C L09 Instruction Format (26)

A Carls, Summer 2005 © UCB

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.



CS 61C L09 Instruction Format (26)

A Carls, Summer 2005 © UCB

J-Format Instructions (3/5)

- target has 26 bits of the 32-bit bit address.

• Optimization:

- jumps will only jump to word aligned addresses,
 - so last two bits of address are always 00 (in binary).
 - let's just take this for granted and not even specify them.



CS 61C L09 Instruction Format (27)

A Carls, Summer 2005 © UCB

J-Format Instructions (4/5)

- Now : we have 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 jump straddles 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.



CS 61C L09 Instruction Format (28)

A Carls, Summer 2005 © UCB

J-Format Instructions (5/5)

• Summary:

- Next 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, 111111111111111111111111111111, 00 }
= 1010111111111111111111111111111100

- Note: Book uses ||, Verilog uses { , , }

- We won't actually be learning Verilog, but it is useful to know a little of its notation



CS 61C L09 Instruction Format (29)

A Carls, Summer 2005 © UCB

Other Jumps and Branches

- We have j and jal

- What about jr?

- J-format won't work (no reg field)
- So, use R-format and ignore other regs:

opcode	rs	rt	rd	shamt	funct
0	\$reg	0	0	0	8

- What about beq and bne?

- Tight fit: 2 regs and an immediate (address)



CS 61C L09 Instruction Format (30)

A Carls, Summer 2005 © UCB

Branches: PC-Relative Addressing (1/4)

- 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
 - Using word-align trick, we can get 18 bits
 - Still not enough!
 - Would have to use jr if straddling a 256KB.



Branches: PC-Relative Addressing (2/4)

- 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 (3/4)

- 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 $\pm 2^{15}$ words from the PC, which should be enough to cover almost any loop.



Branches: PC-Relative Addressing (4/4)

- Branch Calculation:

- If we **don't** take the branch:
 $next\ PC = PC + 4$
PC+4 = byte address of next instruction
- If we **do** take the branch:
 $next\ PC = (PC + 4) + (immediate * 4)$
- 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: sub $2,$3,$4
```

- 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: sub $2,$3,$4
```

- Immediate Field:

- Number of instructions to add to (or subtract from) the PC, starting at the instruction **following** the branch (“+4”).
- In beq case, immediate = 3



