

CSE 31

Computer Organization

Lecture 13 – Instruction Format (2)



Announcement

- ▶ Lab #6 this week
 - Due next week
- ▶ HW #4 out this Friday (from zyBooks)
 - Due Monday (10/22)
- ▶ Project #1
 - Due Monday (10/22)
 - Don't start late, you won't have time!
- ▶ Reading assignment
 - Chapter 4.1-4.9 of zyBooks (Reading Assignment #4)
 - Make sure to do the Participation Activities
 - Due Monday (10/15)

Review

- ▶ 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



R-Format Example (1/2)

- ▶ MIPS Instruction:

add \$t0, \$t1, \$t2

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

opcode = 0 (look up in table)

funct = 32 (look up in table)

rd = 8 (destination)

rs = 9 (first operand)

rt = 10 (second operand)

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

hex

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 later

I-Format Example (1/2)

- ▶ MIPS Instruction:

addi **\$s5, \$s6, -50**

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	1111111111001110
--------	-------	-------	------------------

hexadecimal representation: `22D5 FFCEhex`

decimal representation: `584,449,998ten`

Quiz

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

- a) add \$0, \$0, \$0
- b) subu \$s0,\$s0,\$s0
- c) lw \$0, 0(\$0)
- d) addi \$0, \$0, 35
- e) subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Registers numbers and names:

0: \$0, .. 8: \$t0, 9: \$t1, ..15: \$t7, 16: \$s0, 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

Quiz

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

- a) add \$0, \$0, \$0
- b) subu \$s0,\$s0,\$s0
- c) lw \$0, 0(\$0)
- d) addi \$0, \$0, 35
- e) subu \$0, \$0, \$0**

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Registers numbers and names:

0: \$0, .. 8: \$t0, 9: \$t1, ..15: \$t7, 16: \$s0, 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

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 easy 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.

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?
 - `addi $s0, $s1, 40000`
 - 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 **L**oad **U**pper **I**mmEDIATE
- takes 16-bit immediate and puts these bits in the upper half (high order half) of the register
- sets lower half to 0s

I-Format Problems (3/3)

► Solution to Problem (continued):

- So how does `lui` help us?
- Example:

```
addiu $t0, $t0, 0xABABCD
```

...becomes

```
lui $at 0xABAB  
ori $at, $at, 0xCD  
addu $t0, $t0, $at
```

- 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

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

- ▶ `opcode` specifies `beq` versus `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.

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 $\pm 2^{15}$ 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:

$PC = PC + 4 = \text{byte address of next instruction}$

- 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  
      addu   $8, $8, $10  
      addiu  $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 (2/3)

▶ MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $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 **beq** case, **immediate** = 3

Branch Example (3/3)

► MIPS Code:

```
Loop:  beq    $9, $0, End
        addu   $8, $8, $10
        addiu  $9, $9, -1
        j      Loop
```

End:

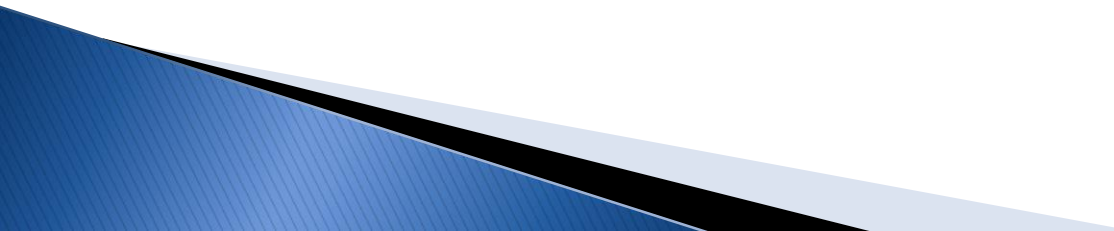
decimal representation:

4	9	0	3
---	---	---	---

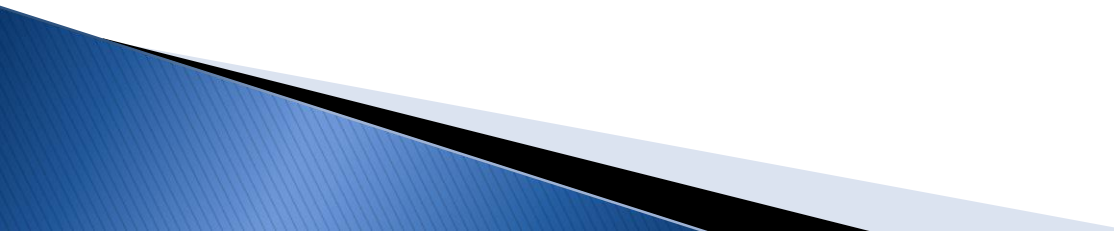
binary representation:

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?
 - ▶ 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 two “fields” of these bit widths:

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.
 - Collapse 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
 - What 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
 - { 1010, 11111111111111111111111111111111, 00 } =
10101111111111111111111111111111111100

Quiz

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

- 1) **Jump** insts don't require any changes.
- 2) **Branch** insts don't require any changes.

	12
a)	FF
b)	FT
c)	TF
d)	TT
e)	dunno

Quiz

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

- 1) **Jump** insts don't require any changes.
- 2) **Branch** insts don't require any changes.

	12
a)	FF
b)	FT
c)	TF
d)	TT
e)	dunno

Summary

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

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- ▶ Branches use **PC-relative addressing**, Jumps use **absolute addressing**.
- ▶ Disassembly is simple and starts by decoding `opcode` field. (more in a week)

Decoding Machine Language

- ▶ How do we convert 1s and 0s to assembly language and to C code?

Machine language \Rightarrow assembly \Rightarrow C?

- ▶ For each 32 bits:
 1. Look at `opcode` to distinguish between R-Format, J-Format, and I-Format.
 2. Use instruction format to determine which fields exist.
 3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
 4. Logically convert this MIPS code into valid C code.
- ▶ Always possible? Unique?

Decoding Example (1/7)

- ▶ Here are six machine language instructions in hexadecimal:

00001025_{hex}

0005402A_{hex}

11000003_{hex}

00441020_{hex}

20A5FFFF_{hex}

08100001_{hex}

- ▶ Let the first instruction be at address 4,194,304_{ten} (0x00400000_{hex}).
- ▶ Next step: convert hex to binary

Decoding Example (2/7)

- ▶ The six machine language instructions in binary:

```
00000000000000000000000010000000100101
000000000000000010101000000000101010
00010001000000000000000000000000000011
00000000001000100000010000001000000100000
00100000010100101111111111111111111111
00001000000010000000000000000000000001
```

- ▶ Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4–62	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/7)

- ▶ Select the opcode (first 6 bits) to determine the format:

Format:

R	000000	000000	000000	00010	000000	100101
R	000000	000000	00101	01000	000000	101010
I	000100	01000	00000	0000000000000000	0000000000000000	0000000000000000
R	000000	00010	00100	00010	000000	100000
I	001000	00101	00101	1111111111111111	1111111111111111	1111111111111111
J	000010	000000	100000	0000000000000000	0000000000000000	0000000000000000

- ▶ Look at opcode:
0 means R-Format,
2 or 3 mean J-Format,
otherwise I-Format.
- ▶ Next step: separation of fields

Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

Next step: translate (“disassemble”) to MIPS assembly instructions

Decoding Example (5/7)

- ▶ MIPS Assembly (Part 1):

Address:	Assembly instructions:
0x00400000	or \$2, \$0, \$0
0x00400004	slt \$8, \$0, \$5
0x00400008	beq \$8, \$0, 3
0x0040000c	add \$2, \$2, \$4
0x00400010	addi \$5, \$5, -1
0x00400014	j 0x100001

Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

Decoding Example (6/7)

► MIPS Assembly (Part 2):

```

                or      $v0, $0, $0
Loop:          slt     $t0, $0, $a1
                beq     $t0, $0, Exit
                add     $v0, $v0, $a0
                addi    $a1, $a1, -1
                j       Loop
Exit:
```

Next step: translate to C code (must be creative!)

Decoding Example (7/7)

Before Hex:

00001025_{hex}
0005402A_{hex}
11000003_{hex}
00441020_{hex}
20A5FFFF_{hex}
08100001_{hex}

After C code (Mapping below)

```
$v0: product  
$a0: multiplicand  
$a1: multiplier  
product = 0;  
while (multiplier > 0) {  
    product += multiplicand;  
    multiplier -= 1;  
}
```

Idea: Instructions are just numbers, code is treated like data

```
or      $v0, $0, $0  
Loop:  slt      $t0, $0, $a1  
        beq      $t0, $0, Exit  
        add      $v0, $v0, $a0  
        addi     $a1, $a1, -1  
        j        Loop  
Exit:
```