inst.eecs.berkeley.edu/~cs61c/su05

CS61C: Machine Structures

Lecture #9: MIPS Instruction Format



2005-07-05

Andy Carle

Carle. Summer 2005 ©

Big Idea: Stored-Program Concept

Computers built on 2 key principles:

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



A Carle, Summer 2005 © UC

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



tion Format (3) A Carle

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
 - Each register is a word.
 - •1w 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



CS 61C L09 Instruction Format (4)

A Carlo Summer 2005 © I

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



A Carle, Summer 2005 © UCB

Instruction Formats

- I-format: used for instructions with immediates, 1w 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



CS 61C L09 Instruction Format (6)

A Carle, Summer 2005 © UCB

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:

 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 \rightarrow 0-31, 6-bit fields \rightarrow 0-63.



Carle Summer 2

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.
 - <u>funct</u>: combined with opcode, this number exactly specifies the instruction for R-Format instructions



A Carlo Summer 2005 © HCI

R-Format Instructions (3/5)

- More fields:
 - •<u>rs</u> (Source Register): *generally* used to specify register containing first operand
 - <u>rt</u> (Target Register): generally used to specify register containing second operand (note that name is misleading)
 - •<u>rd</u> (Destination Register): <u>generally</u> used to specify register which will receive result of computation



A Carle Summer 2005 © LICE

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)



CS 61C I 09 Instruction Format (1)

A Carlo Summer 2005 ©

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)



A Carle, Summer 2005 © UCB

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)



S 61C L09 Instruction Format (12)

A Carle, Summer 2005 © UCE

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: decimal representation: 012A 4020_{hex} 19,546,144_{ten}

• Called a Machine Language Instruction



A Carle, Summer 2005 ©

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.



(14) A Carle, Summer 2005 © UC

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.



A Carle, Summer 2005 © UCB

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
 - •<u>rs</u>: specifies the *only* register operand (if there is one)
 - <u>rt</u>: specifies register which will receive result of computation (this is why it's called the *target* register "rt")



Called till

A Carlo Summer 2005 ©

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 2¹⁶ 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.



A Carle, Summer 2005 © UCB

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)



A Carle, Summer 2005 © UCB

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



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 $2^{15} \le n < 2^{16}$ (l.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



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.



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

I-Format Problems (3/3)

- Solution to Problem 1 (continued):
 - So how does lui help us?
 - Example:

addi \$t0,\$t0, 0xABABCDCD

becomes:

\$at, 0xABAB \$at, \$at, 0xCDCD \$t0,\$t0,\$at ori add

- Now each I-format instruction has only a 16bit immediate.
- · Wouldn't it be nice if the assembler would this for us automatically? (later)



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>"



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:



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.



A Carle, Summer 2005 © UC

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.



A Carle, Summer 2005 © U

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 I 09 Instruction Format (28)

A Carla Summer 2005 ©

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, 11111111111111111111111111, 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



A Carle, 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 beg and bne?
 - Tight fit: 2 regs and an immediate (address)

CS 61C L09 Instruction Format (30)

A Carle, Summer 2005 © UC

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 ± 2¹⁵ words from the PC, which should be enough to cover álmost 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:

\$9,\$0,<u>End</u> \$8,\$8,\$10 Loop: beq add addi \$9,\$9,-1 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 \$2,\$3,\$4 End: sub

- 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



Branch Example (3/3)

• MIPS Code:

Loop: beq \$9,\$0,End addi \$8,\$8,\$10 addi \$9,\$9,-1 j Loop End: sub \$2,\$3,\$4

decimal representation:

4	9	0	3			
hinary representation:						

binary representation:

000100 01001 00000 000000000000011



MIPS So Far:

• MIPS Machine Language Instruction: 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
ı	opcode	rs	rt	immediate		:e
J	opcode	target address				

 Branches use PC-relative addressing, Jumps use PC-absolute addressing.



A Carlo Summer 2005

Decoding Machine Language

Questions on PC-addressing

if we move the code?

Does the value in branch field change

• What do we do if destination is > 2¹⁵ instructions away from branch?

- How do we convert 1s and 0s to C code?
 Machine language ⇒ C?
- For each 32 bits:
 - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
 - Use instruction type to determine which fields exist.
 - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
- 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}

- \bullet Let the first instruction be at address 4,194,304 $_{\rm ten}$ (0x00400000 $_{\rm hex}$).
- Next step: convert hex to binary



A Carle, Summer 2005 © UCB

Decoding Example (2/7)

• The six machine language instructions in binary:

· Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
1	1, 4-31	rs	rt	immediate		te
J	2 or 3	target address				

struction Format (42)

A Carle, Summer 2005 © UCB

Decoding Example (3/7)

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

Format:

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
1	4	8	0		+3	
R	0	2	4	2	0	32
1	8	5	5		-1	
J	2		1	,048,5	77	

 Next step: translate ("disassemble") to MIPS assembly instructions



A Carle, Summer 2005 © UCB

Decoding Example (5/7)

• MIPS Assembly (Part 1):

Address: Assembly instructions:

0×00400000	or	\$2,\$0,\$0
0×00400004	slt	\$8,\$0,\$5
0×00400008	beq	\$8,\$0,3
0x0040000c	add	\$2,\$2,\$4
0×00400010	addi	\$5,\$5,-1
0×00400014	i	0x100001

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



inii o 7 toooniibiy (i airo

• MIPS Assembly (Part 2):

Decoding Example (6/7)

```
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 (be creative!)



CS 61C L09 Instruction Format (46)

A Carle Summer 2005 ©

Decoding Example (7/7)

```
Before Hex:

    After C code (Mapping below)

                        $v0: product
00001025<sub>hex</sub>
                        $a0: multiplicand
0005402A<sub>hex</sub>
                        $a1: multiplier
11000003<sub>hex</sub>
                   product = 0;
00441020<sub>hex</sub>
                   while (multiplier > 0) {
20A5FFFF<sub>hex</sub>
                        product += multiplicand;
multiplier -= 1;
08100001<sub>hex</sub>
              $v0,$0,$0
        or
```

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

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data

A Carle, Summer 2005 © UCB

Peer Instruction Question

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

- A. Jump insts don't require any changes.
- B. Branch insts don't require any changes.
- C. You now have all the tools to be able to "decompile" a stream of 1s and 0s into C!



A Carle, Summer 2005 © UC