Machine Code

-and-

How the Assembler Works

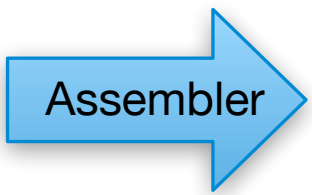
Mar 8–13, 2013

Outline

Assembly language vs. machine code

Assembler translates assembly code to machine code

```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```




```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

Assembly program (text file)
source code

Machine code (binary)
object code

What is machine code?

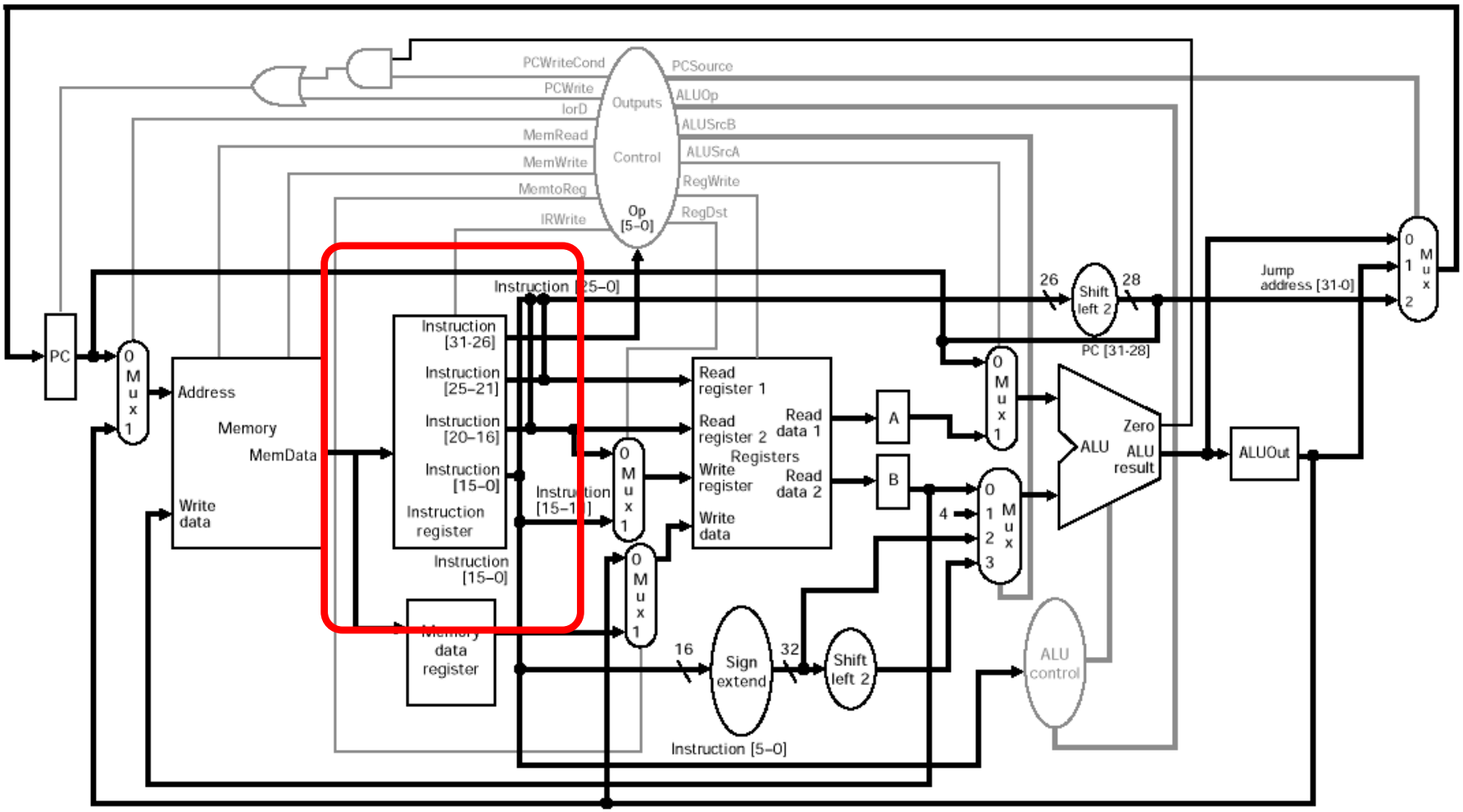
Machine code is the **interface** between software and hardware

The processor is "hardwired" to implement machine code

- the bits of a machine instruction are *direct inputs* to the components of the processor

This is only true for RISC architectures!

Decoding an instruction (RISC)



What about CISC?

Main difference between RISC and CISC

- RISC – machine code implemented directly by hardware
- CISC – processor implements an even lower-level instruction set called **microcode**

Translation from machine code to microcode is “hardwired”

- written by an architecture designer
- never visible at the software level

RISC vs. CISC

Advantages of CISC

- an extra layer of abstraction from the hardware
- easy to add new instructions
- can change underlying hardware without changing the machine code interface

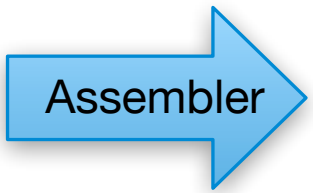
Advantages of RISC

- easier to understand and teach :-)
- regular structure make it easier to pipeline
- no machine code to microcode translation step

No clear winner . . . which is why we still have both!

How does the assembler assemble?

```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```




```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

Assembly program (text file)

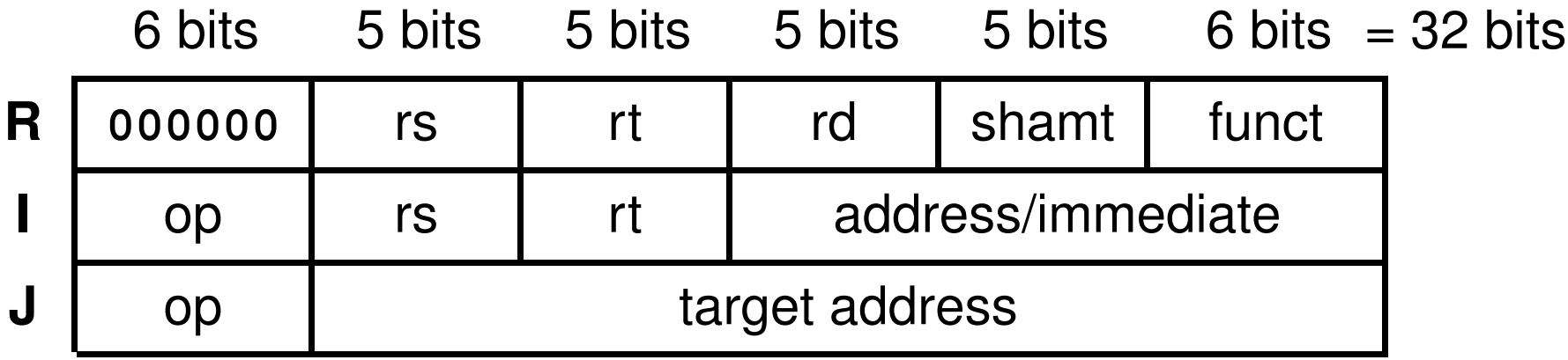
source code

Machine code (binary)

object code

MIPS instruction formats

Every assembly language instruction is translated into a machine code instruction in one of three **formats**



- **R**egister-type
- **I**mmediate-type
- **J**ump-type

Example instructions for each format

Register-type instructions

```
# arithmetic and logic
add   $t1, $t2, $t3
or    $t1, $t2, $t3
slt   $t1, $t2, $t3

# mult and div
mult  $t2, $t3
div   $t2, $t3

# move from/to
mfhi  $t1
mflo  $t1

# jump register
jr    $ra
```

Immediate-type instructions

```
# immediate arith and logic
addi  $t1, $t2, 345
ori   $t1, $t2, 345
slti  $t1, $t2, 345

# branch and branch-zero
beq   $t2, $t3, label
bne   $t2, $t3, label
bgtz  $t2, label

# load/store
lw    $t1, 345($t2)
sw    $t2, 345($t1)
lb    $t1, 345($t2)
sb    $t2, 345($t1)
```

Jump-type instructions

```
# unconditional jump
j     label
```

```
# jump and link
jal   label
```

Outline

Components of an instruction

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	000000	rs	rt	rd	shamt	funct
I	op	rs	rt	address/immediate		
J	op	target address				

Component	Description
op, funct	codes that determine operation to perform
rs, rt, rd	register numbers for args and destination
shamt, imm, addr	values embedded in the instruction

Assembling instructions

Assemble: translate from assembly to machine code

- for our purposes: translate to a hex representation of the machine code

How to assemble a single instruction

1. decide which instruction format it is (R, I, J)
2. determine value of each component
3. convert to binary
4. convert to hexadecimal

Determining the value of register components

Number	Name	Usage	Preserved?
`$0`	`$zero`	constant `0x00000000`	N/A
`$1`	`$at`	assembler temporary	N/A
`$2–$3`	`$v0–$v1`	function return values	✗
`$4–$7`	`$a0–$a3`	function arguments	✗
`$8–$15`	`$t0–$t7`	temporaries	✗
`$16–$23`	`$s0–$s7`	saved temporaries	✓
`$24–$25`	`$t8–$t9`	more temporaries	✗
`$26–$27`	`$k0–$k1`	reserved for OS kernel	N/A
`$28`	`$gp`	global pointer	✓
`$29`	`$sp`	stack pointer	✓
`$30`	`$fp`	frame pointer	✓
`$31`	`$ra`	return address	N/A

Components of an R-type instruction

R-type instruction

• op	6 bits	always zero!
• rs	5 bits	1st argument register
• rt	5 bits	2nd argument register
• rd	5 bits	destination register
• shamt	5 bits	used in shift instructions (for us, always 0s)
• funct	6 bits	code for the operation to perform
	32 bits	

Note that the destination register is third in the machine code!

Assembling an R-type instruction

```
add $t1, $t2, $t3
```

000000	rs	rt	rd	shamt	funct

rs = 10 ($t2 = $10)
rt = 11 ($t3 = $11)
rd = 9 ($t1 = $9)
funct = 32 (look up function code for add)
shamt = 0 (not a shift instruction)

000000	10	11	9	0	32

000000	01010	01011	01001	00000	100000

0000 0001 0100 1011 0100 1000 0010 0000

0x014B4820

Exercises

Assemble the following instructions:

- `sub $s0, $s1, $s2`
- `mult $a0, $a1`
- `jr $ra`

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Instr	fn
`add`	32
`sub`	34
`mult`	24
`div`	26
`jr`	8

Components of an I-type instruction

I-type instruction

• op	6 bits	code for the operation to perform
• rs	5 bits	1st argument register
• rt	5 bits	destination or 2nd argument register
• imm	16 bits	constant value embedded in instruction
	32 bits	

Note the destination register is second in the machine code!

Assembling an I-type instruction

```
addi $t4, $t5, 67
```

op	rs	rt	address/immediate

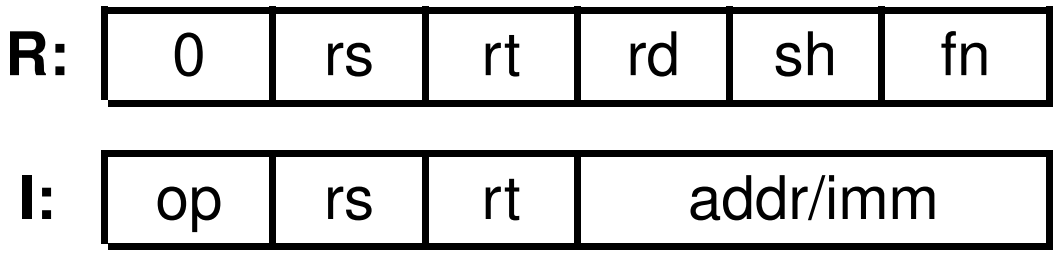
op = 8 (look up op code for `addi`)
rs = 13 (`$t5` = `$13`)
rt = 12 (`$t4` = `$12`)
imm = 67 (constant value)

8	13	12	67
`001000`	`01101`	`01100`	`0000 0000 0100 0011`

`0010 0001 1010 1100 0000 0000 0100 0011`

`0x21AC0043`

Exercises



Assemble the following instructions:

- `or  $s0, $t6, $t7`
- `ori $t8, $t9, 0xFF`

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Instr	op/fn
`and`	36
`andi`	12
`or`	37
`ori`	13

Conditional branch instructions

```
beq $t0, $t1, label
```

I-type instruction

- op 6 bits code for the comparison to perform
- rs 5 bits 1st argument register
- rt 5 bits 2nd argument register
- imm 16 bits **jump offset** embedded in instruction

32 bits

Calculating the jump offset

Jump offset

Number of instructions from the **next instruction**

(`nop` is an instruction that does nothing)

```
        beq  $t0, $t1, skip
        nop  # 0 (start here)
        nop  # 1
        nop  # 2
skip:   nop  # 3!
        ...
```

offset = 3

```
loop:   nop  # -5
        nop  # -4
        nop  # -3
        nop  # -2
        beq  $t0, $t1, loop
        nop  # 0 (start here)
```

offset = -5

Assembling a conditional branch instruction

```
        beq  $t0, $t1, label
        nop
        nop
label:  nop
```

op	rs	rt	address/immediate

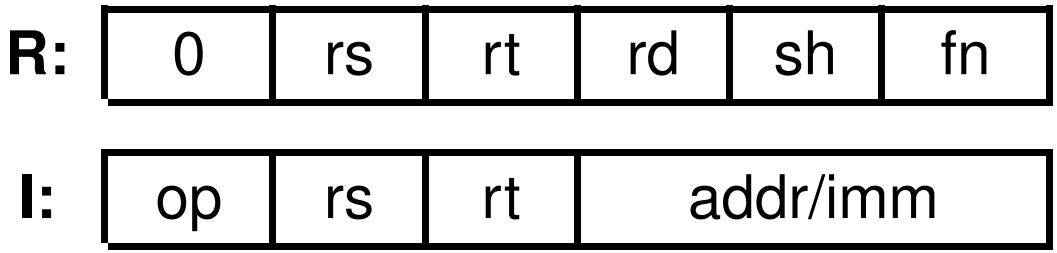
op = 4 (look up op code for `beq`)
rs = 8 (`$t0` = `$8`)
rt = 9 (`$t1` = `$9`)
imm = 2 (jump offset)

4	8	9	2
000100	01000	01001	0000 0000 0000 0010

0001 0001 0000 1001 0000 0000 0000 0010

0x11090002

Exercises



Assemble the following program:

```
# Pseudocode:
#   do {
#     i++
#   } while (i != j);
loop:  addi  $s0, $s0, 1
       bne   $s0, $s1, loop
```

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Instr	op/fn
`add`	32
`addi`	8
`beq`	4
`bne`	5

J-type instructions

J:	op	target address

Only two that we care about: `j` and `jal`

- remember, `jr` is an R-type instruction!

Relative vs. absolute addressing

Branch instructions – offset is relative:

$$PC = PC + 4 + offset \times 4$$

Jump instructions – address is absolute:

$$PC = (PC \ \& \ 0xF0000000) \mid (address \times 4)$$

"Absolute" relative to a 256Mb region of memory

(MARS demo: AbsVsRel.asm)

Determining the address of a jump

```
0x4000000          j     label
...                ...
0x40000A4  label:  nop
...                ...
0x404C100          j     label
```

Address component of jump instruction

1. Get address at label in hex — `0x40000A4`
2. Drop the first hex digit — `0x 0000A4` = `0xA4`
3. Convert to binary — `10100100`
4. Drop the last two bits — `101001`

Assembling a jump instruction

```
0x4000000 |          j     label
...       |          ...
0x40000A4 | label:   nop
...       |          ...
0x404C100 |          j     label
```

op	target address

op = 2 (look up opcode for `j`)
addr = `101001` (from previous slide)

2	`101001`

`0000 10`	`00 0000 0000 0000 0000 0010 1001`

`0x08000029`

Comparison of jump/branch instructions

Conditional branches – `beq`, `bne`

- offset is 16 bits
  - effectively 18 bits, since $\times 4$
- range: 2^{18} = PC $\pm$ 128kb

Unconditional jumps – `j`, `jal`

- address is 26 bits
  - effectively 28 bits, since $\times 4$
- range: any address in current 256Mb block

Jump register – `jr`

- address is 32 bits (in register)
- range: any addressable memory location (4GB)

Outline

Basic instructions vs. macro instructions

Basic assembly instruction

- has a corresponding machine code instruction
  - can find the name in the op/funct table
- always assembles into one machine code instruction
- part of the MIPS instruction set
  - will work with any assembler

Basic instructions vs. macro instructions

Macro assembly instruction

- does **not** have a corresponding machine code instruction
  - can't find the name in the op/funct table
- may assemble into multiple machine code instructions
  - can use the `$at` register as a temp to support this
- may be assembler specific!

Examples of macro instructions we've been using all quarter:

- `la`, `li`, `move`, `mul`, `blt`, `bgt`, `ble`, `bge`

Some example macro instructions

```
mul   $t0, $t1, $t2      ⇒     mult $t1, $t2
                               mflo $t0

li    $t0, 0xABCD        ⇒     ori  $t0, $zero, 0xABCD

li    $t0, 0x1234ABCD    ⇒     lui  $at, 0x1234
                               ori  $t0, $at, 0xABCD
```