

Chapter 5:

Machine language

Topics:

Instruction operations

Instruction format/encoding

Assembly and disassembly

Linking and loading

Reading: Patterson and Hennessy

2.5, 2.6, 2.10, 2.12

Appendix B B-49 to B-71

Instruction operations:
what operations (+, - etc) are available

Instruction format/encoding:
how an instruction is represented in binary
in memory

One machine language instruction -> 32-bit
word

Recall: some MIPS assembly instructions are
pseudoinstructions

Major difference between assembly language
and machine language:
Machine language has no labels (!)

(Only raw numeric addresses are used.)

Arithmetic/logic instructions

all operands in registers

General format:

op rd, rs, rt

means $rd = rs \text{ op } rt$

add rd, rs, rt

sub rd, rs, rt

addu rd, rs, rt

subu rd, rs, rt

u: unsigned

addu, subu like add, sub, but overflow ignored

mult rs, rt

64-bit result \leftarrow rs * rt

Two extra 32-bit registers: LO and HI

LO = low 32 bits of 64-bit result

HI = high 32 bits of 64-bit result

(or, HI || LO = 64-bit result)



multu rs, rt

same as mult, but overflow ignored

Use special move instructions to get mult result from HI/LO into \$1 to \$31.

(f: from, t: to)

mfhi rd means $rd = HI$ # move from hi

mflo rd means $rd = LO$ # move from lo

mtli rs means $HI = rs$ # move to hi

mtlo rs means $LO = rs$ # move to lo

assembly language: `mul $23, $22, $21` // int x, y, z;
// $y = x * z$;

machine language: `mult $22, $21`
 `mflo $23`

[assume only care about 32-bit result]

spim 8.0 and later:

`mul $23, $22, $21`

div rs, rt

LO = rs/rt

HI = rs % rt

divu rs, rt same as div, but ignore overflow

assembly language: div \$23, \$22, \$21

machine language: div \$22, \$21
 mflo \$23

MIPS assembly: rem \$23, \$22, \$21

machine lang: div \$22, \$21
 mfhi \$23

Bitwise logical operators (same as assembly):

and rd, rs, rt

or rd, rs, rt

nor rd, rs, rt

xor rd, rs, rt

Shifts:

sllv rd, rt, rs

rd = rt shift left logical by rs bits

srlv rd, rt, rs

rd = rt shift right logical by rs bits

sraav rd, rt, rs

rd = rt shift right arithmetic by rs bits

Instruction format for arithmetic/logic instrs,
all operands in registers

Example:

main: add \$23, \$22, \$13

main: 0x400000

register

R-format:

000000	rs	rt	rd	extended opcode or FUNC
6	5	5	5	11
opcode				

add \$23, \$22, \$13 is represented as:

000000	10110	01101	10111	00000 100000
6	5	5	5	11

main: add \$23, \$22, \$13

Given: &main is 0x400000

0x400000 0x02cdb820

List of arithmetic/logic instructions, register operands only (R-format):

0000 00ss ssst tttt dddd d000 0010 0000	add rd,rs,rt
0000 00ss ssst tttt dddd d000 0010 0010	sub rd,rs,rt
0000 00ss ssst tttt 0000 0000 0001 1000	mult rs,rt
0000 00ss ssst tttt 0000 0000 0001 1010	div rs,rt
0000 00ss ssst tttt dddd d000 0010 0001	addu rd,rs,rt
0000 00ss ssst tttt dddd d000 0010 0011	subu rd,rs,rt
0000 00ss ssst tttt 0000 0000 0001 1001	multu rs,rt
0000 00ss ssst tttt 0000 0000 0001 1011	divu rs,rt
0000 0000 0000 0000 dddd d000 0001 0000	mfhi rd
0000 00ss sss0 0000 0000 0000 0001 0001	mthi rs
0000 0000 0000 0000 dddd d000 0001 0010	mflo rd
0000 00ss sss0 0000 0000 0000 0001 0011	mtlo rs
0000 00ss ssst tttt dddd d000 0010 0100	and rd,rs,rt
0000 00ss ssst tttt dddd d000 0010 0111	nor rd,rs,rt
0000 00ss ssst tttt dddd d000 0010 0101	or rd,rs,rt
0000 00ss ssst tttt dddd d000 0010 0110	xor rd,rs,rt
0000 00ss ssst tttt dddd d000 0000 0100	sllv rd,rt,rs
0000 00ss ssst tttt dddd d000 0000 0110	srlv rd,rt,rs
0000 00ss ssst tttt dddd d000 0000 0111	srav rd,rt,rs

Arithmetic/logic instructions with immediate (constant) operand

addi rt, rs, I
[I is a 16-bit immediate]

$rt = rs + [I \text{ sign-extended to 32 bits}]$

andi rt, rs, I

$Rt = 0^{16} \parallel ([Rs]_{15..0} \text{ AND } I_{15..0})$

(16 0s) low 16 bits of Rs AND 16-bit I
concatenate

ori rt, rs, I

$Rt = [Rs]_{31..16} \parallel ([Rs]_{15..0} \text{ OR } I_{15..0})$

high 16 bits of Rs low 16 bits of Rs OR 16-bit I

xori rt, rs, I

$Rt = [Rs]_{31..16} \parallel ([Rs]_{15..0} \text{ XOR } I_{15..0})$

high 16 bits of Rs low 16 bits of Rs XOR 16-bit I

Instruction format for arithmetic/logic instrs, with a constant operand:

I-format:

opcode	rs	rt	16-bit I
6	5	5	16

Example: ori \$23, \$13, 0x9876

001101	01101	10111	1001 1000 0111 0110
6	5	5	16

If constants that are greater than 16-bit are needed, must construct them 16 bits at a time in a temporary register.

Use load upper immediate instruction (lui):

lui rt,I

$$rt = I_{15..0} \parallel 0^{16}$$

Example: lui \$23, 0x9876

$$\$23 = 0x9876\ 0000$$

Translate:

Assembly language: add \$13, \$23, 0x12345678

Machine language:

lui \$1, 0x1234 # \$1 = 0x1234 0000

ori \$1, \$1, 0x5678 # \$1 = 0x1234 5678

add \$13, \$23, \$1

I-format is also used for loads and stores.

Machine language load/stores have no labels!
Must use numerical addresses.

lw rt, I(rb)

ADDR = contents of rb + (I sign-ext. to 32 bits)

rt = 32-bit word at ADDR

100011	rb	rt	16-bit I
6	5	5	16

I-format for load/stores: 4 = 0000 0000 0000 0100
-4 = 1111 1111 1111 1100

Example: *lw \$s0, -4(\$sp)*

100011	11101	10000	1111 1111 1111 1100
6	5	5	16

Remember that MIPS load/stores have three ways of specifying memory address.

Option 3: `lw rt, constant(rb)`

same as basic machine language format, if constant fits in 16 bits.

(If constant does not fit in 16 bits?)

use `lui`, construct 32-bit constant in `$1`, use `$1`

Option 2: `lw rt, (rb)`

Machine language equivalent:

`lw rt, 0(rb)`

What if memory address is specified with a label?

When system loads MIPS assembly program, addresses are computed for all labels.

```
        .data
x:      .word
y:      .word    0:3
z:      .word
```

Given: address of x = 0x10010008

Assembly language: lw \$23, x

Machine language:

1) construct address of x in a register

```
lui    $1, 0x1001
ori    $1, $1, 8    # $1 = 0x1001 0008
```

2) load word

```
lw     $23, 0($1)
```

More efficient:

```
        lui    $1, 0x1001
        lw     $23, 8($1)
```


List of I-format instructions (arith/logic with immediates, load/stores):

0010 00ss ssst tttt iiii iiii iiii iiii	addi rt,rs,I
0010 01ss ssst tttt iiii iiii iiii iiii	addiu rt,rs,I
0011 00ss ssst tttt iiii iiii iiii iiii	andi rt,rs,I
0011 1100 000t tttt iiii iiii iiii iiii	lui rt,I
0011 01ss ssst tttt iiii iiii iiii iiii	ori rt,rs,I
0011 10ss ssst tttt iiii iiii iiii iiii	xori rt,rs,I
0000 0000 000t tttt dddd diii ii00 0000	sll rd,rt,I
0000 0000 000t tttt dddd diii ii00 0010	srl rd,rt,I
0000 0000 000t tttt dddd diii ii00 0011	sra rd,rt,I
1000 11bb bbbt tttt iiii iiii iiii iiii	lw rt,I(rb)
1000 00bb bbbt tttt iiii iiii iiii iiii	lb rt,I(rb)
1001 00bb bbbt tttt iiii iiii iiii iiii	lbu rt,I(rb)
1010 11bb bbbt tttt iiii iiii iiii iiii	sw rt,I(rb)
1010 00bb bbbt tttt iiii iiii iiii iiii	sb rt,I(rb)

Conditional branches

I-format is used.

Six machine language conditional branches:

beq rs,rt,I

bne rs,rt,I

bltz rs,I

blez rs,I

bgtz rs,I

bgez rs,I

16-bit immediate I gives information on branch target address (explained later).

Translate assembly branches to machine language branches:

Assembly:

beqz rs, target

bnez rs, target

bltz rs, target

blez rs, target

bgtz rs, target

bgez rs, target

beq rs, rt, target

bne rs, rt, target

Machine Language:

`beq rs, $0, target`

`bne rs, $0, target`

same as assembly

For other conditions, must use *set-less-than (slt)* instruction.

slt rd,rs,rt

if (rs < rt) rd = 1
else rd = 0;

slti rt,rs,I

if (rs < (I sign-ext to 32 bits)) rt = 1
else rt = 0;

Assembly Language: blt \$13, \$17, ?

Machine Language: slt \$1, \$13, \$17 # \$1 = \$13 < \$17
bne \$1, \$0, ?

Assembly Language: blt \$13, 10, ?

Machine Language: slti \$1, \$13, 10 # \$1 = \$13 < 10
bne \$1, \$0, ?

\$13 >= \$17? !(\$13 < \$17)?

Assembly Language: bge \$13, \$17, ?

Machine Language: slt \$1, \$13, \$17
beq \$1, \$0, ?

\$13 <= \$17? !(\$13 > \$17)?

Assembly Language: ble \$13, \$17, ?

Machine Language: slt \$1, \$17, \$13
beq \$1, \$0, ?

Branch target address (BTA): address of instruction to jump to if condition is true

$$\text{BTA} = \text{address of branch instruction} + 4$$
$$+ (\text{I shift left 2 bits, sign-ext to 32 bits})$$

Example: given

here: bne \$s1, \$s2, ??

label	address	contents
here:	0x400018	000101 10001 10010 11...101

Where is bne jumping to?

$$\text{BTA} = 0x400018 + 4 + \text{offset}$$

offset = 1111 1111 1111 1111 1111 0100

= 0xffff fff4

$$\text{BTA} = 0x40001c + 0xffff fff4 = 0x400010$$

Example: given addr of here = 0x40002c,
addr of there = 0x400080

here: beq \$t0, \$t1, there

there: [other instruction]

Show contents of word at 0x40002c.

here:
0x40002c

000100 01000 01001 0000 0000 0001 0100
--

$BTA = \&beq + 4 + \text{offset}$

$0x400080 = 0x400030 + \text{offset}$

$\text{offset} = 0x400080 - 0x400030 = 0x50$

$\text{offset} = 0\dots0 \quad 0101 \overset{\uparrow}{0000}$

$I = 0000 \ 0000 \ 0001 \ 0100$

Consider: BTA = addr of branch + 4 + offset

here: b??[registers] there

there: [other instruction]

distance between *here* and *there* is
determined by offset

What is the furthest we can branch with a
16-bit I?

biggest positive I = $2^{15}-1$

biggest positive offset = approx. 2^{17}

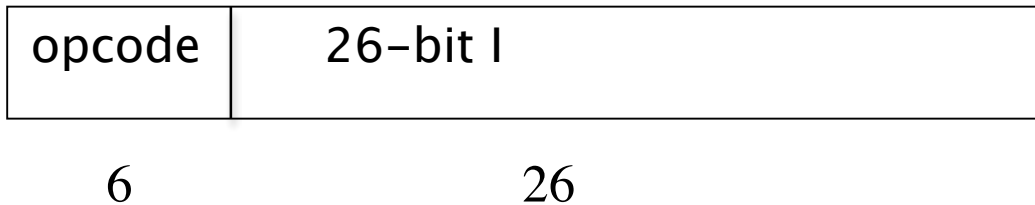
no. of instructions that we can jump
= approx. $2^{15} = 32k$

What if *there* is very far from *here*? beq [], there
Rewrite:

```
        bne [], skip
        j  there
skip:
```

J-format instructions

For jump (*j I*) and jump and link (*jal I*)
(I is a 26-bit constant in J-format)



top 4 bits of
PC

$$PC = [PC]_{31..28} \parallel I_{25..0} \parallel 0^2$$

26-bit I

Example: given address of here = 0x400104
here: j there

label	address	contents
here:	0x400104	000010 0...0 0101 0010

Target address =

0000 0000 0000 0000 0000 0001 0100 1000
0x148

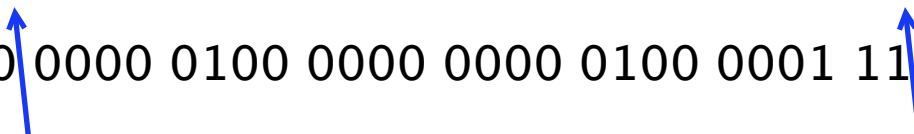
Example: given address of here = 0x400104
addr of there = 0x40041c
here: j there

Show contents of word at 0x400104

here:
0x400104

0000 1000 0001 0000 0000 0001 0000 0111

target addr = 0000 0000 0100 0000 0000 0100 0001 1100



With 26-bit I, max number of instructions
that we can jump = 2^{26}

What if we need to jump further?

construct address in 2 steps using lui, -> \$1

jr \$1

More pseudoinstruction translation:

Given: address of $x = 0x10010008$

MIPS pseudoinstruction: *la \$13, x*

Machine language: `lui $1, 0x1001`
`ori $13, $1, 8`

MIPS pseudoinstruction: *li \$13, 5*

Machine language: `addi $13, $0, 5`

MIPS pseudoinstruction: *move \$23, \$13*

`add $23, $13, $0`

List of conditional branch instructions:

0000 01ss sss0 0000 iiii iiii iiii iiii	bltz rs,I
0000 01ss sss0 0001 iiii iiii iiii iiii	bgez rs,I
0001 10ss sss0 0000 iiii iiii iiii iiii	blez rs,I
0001 11ss sss0 0000 iiii iiii iiii iiii	bgtz rs,I
0001 00ss ssst tttt iiii iiii iiii iiii	beq rs,rt,I
0001 01ss ssst tttt iiii iiii iiii iiii	bne rs,rt,I

List of set less than instructions:

0000 00ss ssst tttt dddd d000 0010 1010	slt rd,rs,rt
0010 10ss ssst tttt iiii iiii iiii iiii	slti rt,rs,I

List of jump instructions:

0000 10ii iiii iiii iiii iiii iiii iiii	j I
0000 11ii iiii iiii iiii iiii iiii iiii	jal I
0000 00ss sss0 0000 0000 0000 0000 0000	jr rs

Special instructions:

0000 0000 0000 0000 0000 0000 0000 1100	syscall
---	---------

Assembly and disassembly

Compilation:

High-level language source code is translated into machine language (or assembly language)

To generate assembly language from C/C++:

```
g++ -S
```

Assembly:

assembly language code is translated into machine code

Disassembly:

machine code (binary) is translated into assembly language

Assembly: translate assembly language program to machine language (binary)

Example:

```

                .data
x:              .word    0:4
y:              .word    3

                .text
main:
                lw        $23,y
                la        $16,x
loop:           sw        $23,($16)
                add       $16,$16,4
                ble       $16,12,loop
```

Given:

address of x = 0x10010000

address of main = 0x400020

address of y = 0x1001 0010

Machine language version:

```
main:    lui    $1, 0x1001      # lw $23, y
         ori    $1, $1, 0x10
         lw     $23, 0($1)
         lui    $1, 0x1001      # la $16, x
         ori    $16, $1, 0

loop:    sw     $23, 0($16)
         addi   $16, $16, 4
         slti   $1, $16, 13
         bne    $1, $0, loop
```

Machine language binary:

0x400020	001111	00000	00001	0001 0000 0000 0001
0x400024	001101	00001	00001	0000 0000 0001 0000
0x400028 lw	100011	00001	10111	0000 0000 0000 0000
0x40002c lui	001101	00001	10000	0000 0000 0000 0000
0x400030 ori	001101	00001	10000	0000 0000 0000 0000
0x400034 sw	101011	10000	10111	0000 0000 0000 0000
0x400038 addi	001000			
0x40003c slti				
0x400040 bne	000101	00001	00000	1111 1111 1111 1100

Calculating branch offset:

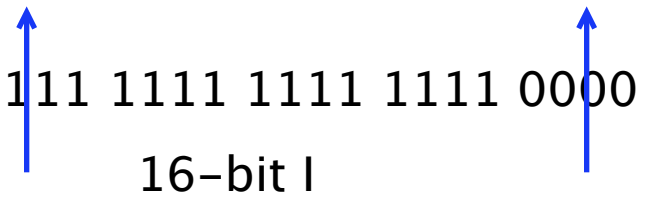
$$\text{BTA} = \&\text{bne} + 4 + \text{offset}$$

$$0x400034 = 0x400044 + \text{offset}$$

$$\text{offset} = 0x400034 - 0x400044 = 0xffffff0$$

$$\text{offset} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000$$

16-bit I



Disassembly example:

0x400000	0011	0100	0001	0000	0000	0000	0000	0001
0x400004	0011	1100	0001	0001	0001	0000	0000	0001
0x400008	1010	1110	0011	0000	0000	0000	0000	0000
0x40000c	0010	0010	0001	0000	0000	0000	0000	0001
0x400010	0010	0010	0011	0001	0000	0000	0000	0100
0x400014	0010	1010	0000	0001	0000	0000	0000	0101
0x400018	0001	0100	0010	0000	1111	1111	1111	1011

Given:

address of main = 0x400000

address of loop = 0x400008

0x400000 ori \$16, \$0, 1	0011 0100 0001 0000 0000 0000 0000 0001
0x400004 lui \$17, 0x1001	0011 1100 0001 0001 0001 0001 0000 0001
0x400008 sw \$16, 0(\$17)	1010 1110 0011 0000 0000 0000 0000 0000
0x40000c addi \$16, \$16, 1	0010 0010 0001 0000 0000 0000 0000 0001
0x400010	0010 0010 0011 0001 0000 0000 0000 0100
0x400014 slti \$1, \$16, 5	0010 1010 0000 0001 0000 0000 0000 0101
0x400018 bne \$1, \$0, loop [0x400008]	0001 0100 0010 0000 1111 1111 1111 1011

Linking and loading

Simple case: main and all functions are in same file, no calls to library functions. Assume functions follow main.

Once address of main is fixed, addresses of all instructions can be fixed.

What if main and user functions are in separate files?

Compile to object files

Link together to form executable

Note: object files do not have all necessary addresses!

Code in object files may:

- reference variables with unresolved addresses
- call functions declared in other files (again, with unresolved addresses)

Example:

[file 1 contains main]

[file 2 contains:

function A

declaration for global variable X]

.data

x: .word ?

more allocations not shown

.text

A: lw \$a0, ?? # load X

code not shown

jr \$31

[file 3 contains:

function B
declaration for global variable Y]

```
.data
y: .word      ?
   # more allocations not shown
.text
B: sw $a1, ??   # store Y
               # code not shown
jr $31
```

An object file contains

Header

Text segment (code with missing addresses)

Data segment (data allocations)

Relocation information

Symbol table

Object files containing A and B: P&H p. 143

What linker does:

1. layout data and code in memory
2. determine addresses of all labels
3. fill in all unresolved addresses/references

Object files are concatenated to form an executable file (*static linking*):

P&H p. 144

Loader copies executable file into memory, starts execution.

Static linking is fine for user code. But libraries can be large! Executables will become too large.

Most compilers by default use *dynamic linking* instead. (This may be tricky because of issues with 64-bit libraries...)

```
libra% gcc ref.c -o ref
libra% ls -l ref
-rwx----- 1 whsu  f1      5873 Jan  3 15:53 ref
libra% gcc ref.c -static -o ref
libra% ls -l ref
-rwx----- 1 whsu  f1  367179 Jan  3 15:53 ref
libra%
```

Disadvantages of static linking:

- executables are large
(include both user code and libraries)
- executable always uses old version of libraries

In dynamic linking:

- only user functions are linked at compile time (library functions remain unresolved)
- at run time, libraries are linked with executable
- executable (user code + libraries) then loaded into memory, start execution

With simple dynamic linking, executables are smaller (include user code only). But entire libraries are still loaded into memory at run time.

Refinement (lazy procedure linkage): a library routine is linked only after it is called.

(More in P&H 2.12)

Summary

Topics covered in this chapter:

MIPS machine language instructions

MIPS binary format

- Arithmetic/logic R-type instructions

- Arithmetic/logic I-format instructions

- Loads and stores

- Conditional branches and jumps

Assemble a MIPS assembly language program

Disassemble a MIPS binary program

Basic concepts of linking and loading