

Instruction Sets - Part 2

Ref: Chapter 3

MIPS Instructions (so far)

- We've seen 2 arithmetic ops: **add** & **sub**
 - 3 operands – all registers.
- 2 Data transfer instructions: **lw**, **sw**
 - base/index addressing
- Two machine language instruction formats:
 - R-Type (3 registers)
 - I-Type (2 registers and offset)

Jumping Around

- There are instructions that change the sequence of instructions fed to the processor, that *jump* to a new part of the program.
- Jumping is also called *branching*.
- Sometimes we want to jump only when some condition is true (or false).

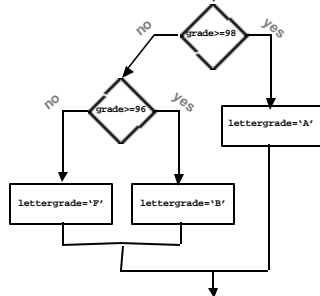
C Program that requires *jumping*

```
if (grade >= 98)
    lettergrade = 'A';
else if (grade >= 96)
    lettergrade = 'B';
else
    lettergrade = 'F';
```

CompOrg Fall 2001 - Instruction Sets (part 2)

4

The *flow* of the program

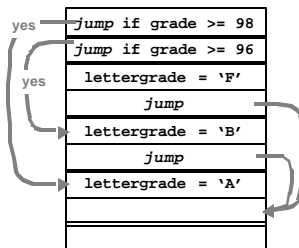


CompOrg Fall 2001 - Instruction Sets (part 2)

5

Possible layout in memory

```
if (grade >= 98)
    lettergrade = 'A';
else if (grade >= 96)
    lettergrade = 'B';
else
    lettergrade = 'F';
```



CompOrg Fall 2001 - Instruction Sets (part 2)

6

MIPS instructions for jumping

beq reg1, reg2, address

Branch if Equal: if the contents of register **reg1** are equal to the contents of register **reg2**, then jump to **address**.

If the registers are not equal, don't do anything special (continue on to the next instruction).

CompOrg Fall 2001 - Instruction Sets (part 2)

7

bne reg1, reg2, address

Branch if Not Equal: if the contents of register **reg1** is not equal to the contents of register **reg2**, then jump to **address**.

If the registers are equal, don't do anything special (continue on to the next instruction).

CompOrg Fall 2001 - Instruction Sets (part 2)

8

beq r1, r2, address: What is **address**?

- In assembly language we create *labels* in the program that can be used as the address (example next slide).
- In machine language the address is an *offset* from the location of the current instruction.

CompOrg Fall 2001 - Instruction Sets (part 2)

9

Example:

```

    if (a==b) c=c+d;
    a=b+b;

```

$\$s0$ $\$s1$ $\$s2$ $\$s3$
 ↙ ↓ ↓ ↓

```

    bne $s0,$s1,L1 # go to L1 if a!=b
    add $s2,$s2,$s3 # c=c+d
L1: add $s0,$s1,$s1 # a=b+b

```

↙ label

CompOrg Fall 2001 - Instruction Sets (part 2) 10

Machine Code for **bne** and **beq**

Instruction Format: *I-Type*:

6 bits 5 bits 5 bits 16 bits
 op rs rt address
 32 bits

CompOrg Fall 2001 - Instruction Sets (part 2) 11

The **address** Field of I-Type Instructions

6 bits 5 bits 5 bits 16 bits
 op rs rt address

MIPS supports 32 bit addresses.
 If we only have a 16 bit address we can't have very large programs!
 The **address** field is *relative to the address of the current instruction*.
 – this is just base/index addressing with the PC as the base register.

CompOrg Fall 2001 - Instruction Sets (part 2) 12

PC: The Program Counter

- There is a special register called the *program counter* that holds the address of the current instruction.
- Normally, this register is incremented by 4 each instruction (MIPS instructions are 4 bytes each).
- When a branch happens – the **address** is added to the **PC** register.

CompOrg Fall 2001 - Instruction Sets (part 2)

13

The value of **PC** during an instruction.

- During the *execution* of an instruction, the processor always adds 4 to the **PC** register.
- This happens *very early* in the instruction.
- As far as we are concerned the PC always holds the address of the *next* instruction.

CompOrg Fall 2001 - Instruction Sets (part 2)

14

Instruction Alignment

- Since MIPS instructions are always stored in memory at an address on a *word boundary* (divisible by 4), the offset specified is actually a *word* offset (not a byte offset).
- A value of 1 means “add 4 to PC”.
- A value of 100 means “add 400 to PC”.

CompOrg Fall 2001 - Instruction Sets (part 2)

15

Assembly vs. Machine Code

```
    bne $s0,$s1,L1    # go to L1 if a!=b
    add $s2,$s2,$s3   # c=c+d
L1:  add $s0,$s1,$s1   # a=b+b
```

- The assembler calculates the difference between the address of the instruction following the **bne** instruction and the instruction labeled L1.
- This difference is used in the address field of the machine code for the **bne** instruction.
- In this case the difference is 1 (1 instruction).

CompOrg Fall 2001 - Instruction Sets (part 2)

16

bne, beq limitations

- The offset from the PC is actually a *signed* integer value.
 - we can jump backwards or forwards.
- The maximum offset is:
 2^{15} instructions = 2^{17} bytes
- The MIPS memory is 2^{32} bytes !

CompOrg Fall 2001 - Instruction Sets (part 2)

17

Unconditional Jump

- MIPS includes an instruction that always jumps:
j address
- In assembly language we just use a label again.

```
j L1
```

CompOrg Fall 2001 - Instruction Sets (part 2)

18

Loops in Assembly

```

    while (a!=b)
        a=a+i;

```

\$s0 \$s1
\$s2

```

Loop:  beq $s0,$s1,Eol
        add $s0,$s0,$s2
        j   Loop
Eol:

```

```

while (a[i] == k)
    i=i+j;

```

\$s3 \$s0 \$s2 \$s1

```

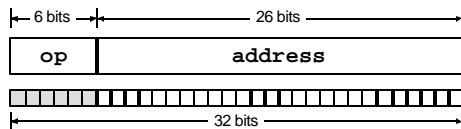
L1:  add $t0,$s0,$s0    # $t0=i*2
      add $t0,$t0,$t0    # $t0=i*4
      add $t0,$t0,$s3    # $t0 = addr of a[i]
      lw  $t1,0($t0)     # $t1 = a[i]
      bne $t1,$s1,L2     # if a[i]!=k goto L2
      add $s0,$s0,$s2    # i=i+j
      j   L1             # goto L1

```

L2:

j instruction format

New Instruction Format: *J-Type*:



J-Type **address** field

- The address field is treated as an instruction address (not a byte address).
- The *rightmost* 28 bits of the PC are replaced with this address.
- It's **not** relative to the PC!
- We have to build very large programs (with more than 2^{26} instructions) very carefully!

↑
Actually the compiler/assembler/linker takes care of this for us!

What about < and > ?

- We often want to compare numbers and jump if one number is less-than or greater than another number.
- MIPS does not include a conditional jump instruction that does this, instead there are some instructions that compare numbers and *store the result in a register*.
- We can then use **beq**, **bne** with the result of the comparison.

Set if Less Than: **slt**

slt *dstreg*, *reg1*, *reg2*

dstreg is set to a 1 if ***reg1*** is less than ***reg2***
(if it's true that ***reg1* < *reg2***)

dstreg is set to a 0 if ***reg1*** is not less than ***reg2***

\$s0
\$s2
 ↙ ↘
if (a<b)

```

slt $t0,$s0,$s2 # $t0 <- a<b
beq $t0,$zero,L1 # jump if a>=b
    
```

↖
\$zero is a MIPS register
 that always holds the value 0!

branch if (a<b)==0, that is, if a<b is false

CompOrg Fall 2001 - Instruction Sets (part 2) 25

The **blt** instruction

(actually there is a **blt** pseudoinstruction)

← 6 bits →	← 5 bits →	← 5 bits →	← 5 bits →	← 5 bits →	← 6 bits →
op	bc	lt	tm	brd	mayo

bc is the amount of bacon
lt is the type of lettuce.
tm is the amount of tomato
brd is wheat, white or rye.
mayo is no, yes, or “pour it on”

CompOrg Fall 2001 - Instruction Sets (part 2) 26

Another unconditional jump

jr reg

- “Jump Register”
 - put the contents of the register in to the PC register.
- The book describes using this with a *jump table* to build a C switch statement.

CompOrg Fall 2001 - Instruction Sets (part 2) 27

Instruction Summary (so far)

- Arithmetic: **add sub**
- Data Movement: **lw sw**
- Jumping around: **bne beq slt j jr**

Byte operations

- In C programs we often deal with strings of characters (ASCII characters).
- Each character is 1 byte.
- We need instructions that can deal with 1 byte at a time.

Load Byte: **lb**

lb destreg, const(addrreg)

- Moves a single byte from memory to the rightmost (least significant) 8 bits of **destreg**.
- The other 24 bits of **destreg** are set to 0
 - actually the byte is *sign extended* (more on this when we talk about arithmetic).
- Base/Index addressing (just like **lw**).

Store Byte: **sb**

sb srcreg, const(addrreg)

- Moves a single byte from the rightmost 8 bits of **destreg** to memory.
- Base/Index addressing (just like **sw**).

CompOrg Fall 2001 - Instruction Sets (part 2)

31

Copying a C string.

- C strings are terminated with a 0.
 - the last byte in the string has the value $00000000_2 = 0_{10}$
- Strings are like arrays of characters.
 - now each array item is 1 byte only!

CompOrg Fall 2001 - Instruction Sets (part 2)

32

Assembly for **strcpy(str1, str2)**

- We aren't yet worried about making this a real subroutine, we just want the code that can do the copying.
- Assume register **\$s1** holds the address of **str1**, and **\$s2** holds the address of **str2**
- We need a loop that copies from the address in **\$s2** to the address in **\$s1**
 - increments **\$s2** and **\$s1** each time.

CompOrg Fall 2001 - Instruction Sets (part 2)

33

A start at **strcpy**

```
Loop:  lb  $t0,0($s2)    # $t0 = *str2
      sb  $t0,0($s1)    # *str1 = $t0
```

need to increment \$s1,\$s2

```
      bne $t0,$zero,Loop
```

Incrementing a register

- We could assume some register has the value 1 in it.
 - it would have to get there some how!
- New instruction: *Add Immediate*

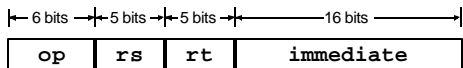
Add Immediate

```
addi destreg, reg1, const
```

Adds a constant to **reg1** and puts the sum in **destreg**.

The term “immediate” means the value (the constant) is already available to the processor (it’s part of the instruction).

addi is an *I-Type* instruction



rt is the destination register
rs is a source operand.
immediate is the constant.

16 bit constant!

CompOrg Fall 2001 - Instruction Sets (part 2)

37

Incrementing a register

- To add 1 to the register **\$s0**:
`addi $s0,$s0,1`
- To add 1234 to the register **\$t3**:
`addi $t3,$t3,1234`
- To add 1,000,000 to the register **\$s2**: ???

CompOrg Fall 2001 - Instruction Sets (part 2)

38

Finishing **strcpy**

```
Loop:  lb  $t0,0($s2)    # $t0 = *str2
      sb  $t0,0($s1)    # *str1 = $t0
      addi $s2,$s2,1    # str2++
      addi $s1,$s1,1    # str1++
      bne $t0,$zero,Loop #
```

CompOrg Fall 2001 - Instruction Sets (part 2)

39

32 bit constants

- Sometimes we need to deal with 32 bit constants!
 - not often, but it happens...
- We know how to load the lower 16 bits with any constant value:
`addi $s0,$zero,const`
- We need some way to put 16 bits in to the left half of a register.

CompOrg Fall 2001 - Instruction Sets (part 2)

40

Load Upper Immediate: `lui`

`lui destreg, const`

- `const` is a 16 bit immediate value.
- The lower 16 bits of `destreg` are all set to 0! (you have to load the upper half first!)

CompOrg Fall 2001 - Instruction Sets (part 2)

41

Immediates are fun!

- There is also a version of `slt` that uses an immediate value:
`slti destreg, reg1,const`
- Will set `destreg` to 1 if `reg1` is less than the 16 bit constant.

CompOrg Fall 2001 - Instruction Sets (part 2)

42

Exercise: Write this in MIPS Assembly Language

```
for (i=0;i<10;i++) {  
    s[i] = s[i+1];  
}
```

s is an array of **char**!

CompOrg Fall 2001 - Instruction Sets (part 2)

43

One Solution

(assumes the address of **s** is in **\$s1**)

```
addi $s0,$zero,$zero    # i=0  
L1:  slti $t1,$s0,10     # i<10?  
     beq  $t1,$zero,L2   # no-jump  
     add  $t2,$s0,$s1    # t2 is addr of s[i]  
     lb  $t3,0($t2)      # t3 is s[i]  
     addi $t2,$t2,1      # t2 is addr of s[i+1]  
     sb  $t3,0($t2)      # s[i+1] = t3  
     addi $s0,$s0,1      # i++  
     j   L1              # go to L1  
L2:
```

There are other solutions! (less code?)

CompOrg Fall 2001 - Instruction Sets (part 2)

44

More Fun

Write the MIPS code to clear an array of
integers (4 bytes each) using as few
instructions as possible.

- what is the IC if the array size is n ?
- Already forgot what IC is?
 - Instruction Count!

CompOrg Fall 2001 - Instruction Sets (part 2)

45

Yes, there is a **mult** instruction,
but first...

Write the MIPS code to calculate the product
of 2 integers (without using any multiply
instructions).

- for now, assume the two integers are at most 16
bits, so don't worry about overflow...

What's Next

- Instructions for procedures (subroutines).
- Complete Assembly Language programs.
- Other Instruction Sets
 - 80x86 CISC
 - Redcode: CoreWars!
- SPIM, SPIM, SPIM
- Computer Arithmetic
 - representation: signed integers, floating point.
 - Building an Arithmetic Logic Unit (ALU)
 - more MIPS arithmetic and logical instructions
