


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface



EECS 2021

Computer Organization Fall 2015


*The slides are based on the publisher slides
and contribution from Profs Amir Asif and
Peter Lian*
*The slides will be modified, annotated,
explained on the board, and sometimes
corrected in the class*

Based on slides by the author and prof.
Mary Jane Irwin of PSU.

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- beq rs, rt, L1
 - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
 - if (rs != rt) branch to instruction labeled L1;
- j L1
 - unconditional jump to instruction labeled L1

§2.7 Instructions for Making Decisions



Chapter 2 — Instructions: Language of the Computer — 2

Conditional Operations

- `beq $s0, $s1, L1` How to specify L1
- `bne $s0, $s1, L1`
- Instruction format

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits
5	16	17	L1
4	16	17	L1

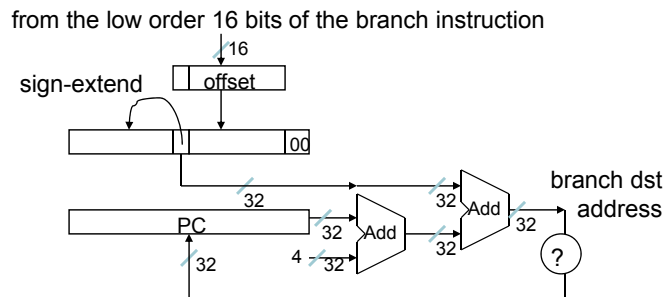


Specifying Branch Destination

- We could specify the memory location, but that will require 32 bits ???
- Can use a base register, the base register is PC
- Limits jumps to $-2^{15} \rightarrow 2^{15} - 1$
- In reality, 00 is appended to the immediate thus instructions (words not bytes)



Branch destination



Jump instruction

- J Label #go to label

op	26-bit address
6 bits	

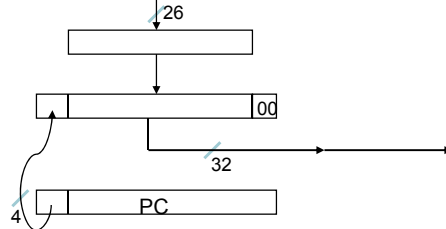
2	xxxx..xx
---	----------

- Again, concatenating 00 increase the effective number to 28 + the left-most 4 bits of the PC (added to the PC)



Jump instruction

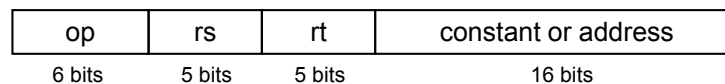
from the low order 26 bits of the jump instruction



Chapter 2 — Instructions: Language of the Computer — 7

Branch Addressing

- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward



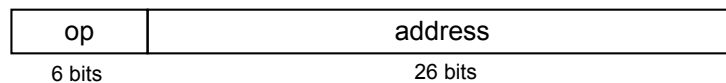
- PC-relative addressing
 - Target address = PC + offset × 4
 - PC already incremented by 4 by this time



Chapter 2 — Instructions: Language of the Computer — 8

Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
 - Encode full address in instruction



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31..28} : (\text{address} \times 4)$



Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

Loop: sll \$t1, \$s3, 2	80000	0	0	19	9	4	0
add \$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw \$t0, 0(\$t1)	80008	35	9	8			0
bne \$t0, \$s5, Exit	80012	5	8	21			2
addi \$s3, \$s3, 1	80016	8	19	19			1
j Loop	80020	2					20000
Exit: ...	80024						



Compiling If Statements

- C code:

```
if (i == j) f = g+h;
else f = g-h;
```

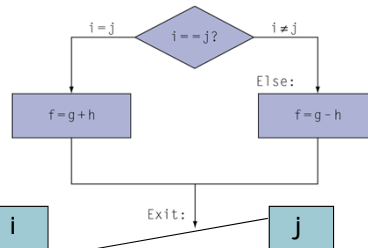
- f, g, ... in \$s0, \$s1, ...

- Compiled MIPS code:

```

        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   ...

```



Assembler calculates addresses



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

```

Loop:  sll $t1, $s3, 2
        add $t1, $t1, $s6
        lw  $t0, 0($t1)
        bne $t0, $s5, Exit
        addi $s3, $s3, 1
        j   Loop
Exit:  ...

```

Multiply i by 4

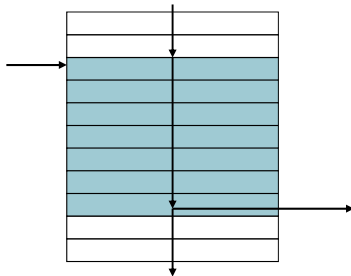
Address of save[i]

save[i]



Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



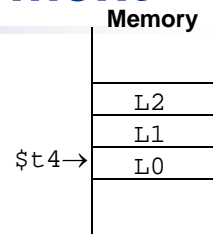
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks



Compiling Case Statement

```
switch (k) {
  case 0: h=i+j; break; /*k=0*/
  case 1: h=i+h; break; /*k=1*/
  case 2: h=i-j; break; /*k=2*/
}
```

- Assuming three sequential words in memory starting at the address in \$t4 have the addresses of the labels L0, L1, and L2 and **k** is in \$s2



```

      add    $t1, $s2, $s2      # $t1 = 2*k
      add    $t1, $t1, $t1      # $t1 = 4*k
      add    $t1, $t1, $t4      # $t1 = addr of JumpT[k]
      lw     $t0, 0($t1)        # $t0 = JumpT[k]
      jr     $t0                # jump based on $t0
L0:    add    $s3, $s0, $s1      # k=0 so h=i+j
      j     Exit
L1:    add    $s3, $s0, $s3      # k=1 so h=i+h
      j     Exit
L2:    sub    $s3, $s0, $s1      # k=2 so h=i-j
Exit:  . . .
```



Example

- Assemble the following machine code

```

        beq    $s0, $s1, Else
        add    $s3, $s0, $s1
        j      Exit
Else:    sub    $s3, $s0, $s1
Exit:    ...
    
```



Example

0x00400020	4	16	17	2	
0x00400024	0	16	17	19	0 0x20
0x00400028	2	0000	0100	0 ... 0	0011 00 ₂
		jmp dst = (0x0) 0x040003 00 ₂ (00 ₂) = 0x00400030			
0x0040002c	0	16	17	19	0 0x22
0x00400030	...				



More Conditional Operations

- Set `dest` to 1 if a condition is true
 - Otherwise, set to 0
- `slt rd, rs, rt`
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;
- `slti rt, rs, constant`
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L # branch to L
```



Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise



Signed vs. Unsigned

- Signed comparison: `sl t, sl ti`
- Unsigned comparison: `sl tu, sl tui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `sl t $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sl tu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

