Math 230
Assembly Programming
(*AKA Computer Organization*)
Spring 2008

MIPS Intro III – Branch Instructions

Feb 19, 2008

Lect 11

Adapted from slides developed for:

Mary J. Irwin PSU CSE331

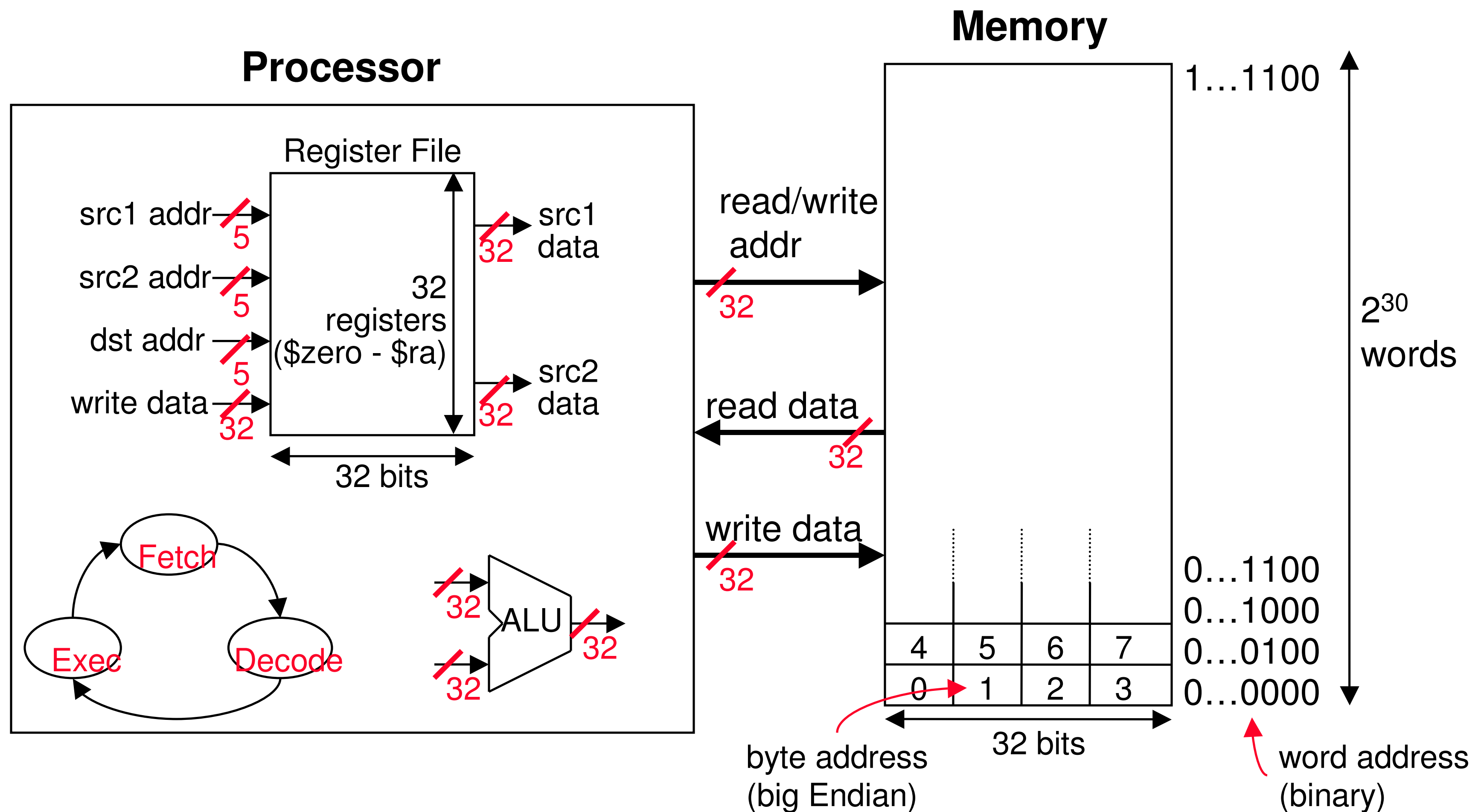
Dave Patterson's UCB CS152

MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	**add**	0 and 32	**add** $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	**sub** $s1, $s2, $s3	$s1 = $s2 - $s3
Data transfer (I format)	**load word**	35	**lw** $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	43	**sw** $s1, 100($s2)	Memory($s2+100) = $s1

Review: MIPS Organization

- Arithmetic instructions – to/from the register file
- Load/store word and byte instructions – from/to memory



Assembling Code

- Remember the assembler code we compiled for the C statement

```
A[8] = A[2] - b
```

```
lw   $t0, 8($s3)     #load A[2] into $t0
sub  $t0, $t0, $s2   #subtract b from A[2]
sw   $t0, 32($s3)    #store result in A[8]
```

Assemble the MIPS code for these three instructions

Assembling Code

- Remember the assembler code we compiled for the C statement

```
A[8] = A[2] - b
```

```
lw   $t0, 8($s3)      #load A[2] into $t0
sub  $t0, $t0, $s2    #subtract b from A[2]
sw   $t0, 32($s3)     #store result in A[8]
```

Assemble the MIPS code for these three instructions

Assembling Code

- Remember the assembler code we compiled for the C statement

```
A[8] = A[2] - b
```

```
lw    $t0, 8($s3)      #load A[2] into $t0
sub   $t0, $t0, $s2    #subtract b from A[2]
sw    $t0, 32($s3)     #store result in A[8]
```

Assemble the MIPS code for these three instructions

lw	35	19	8	8		
sub	0	8	18	8	0	34

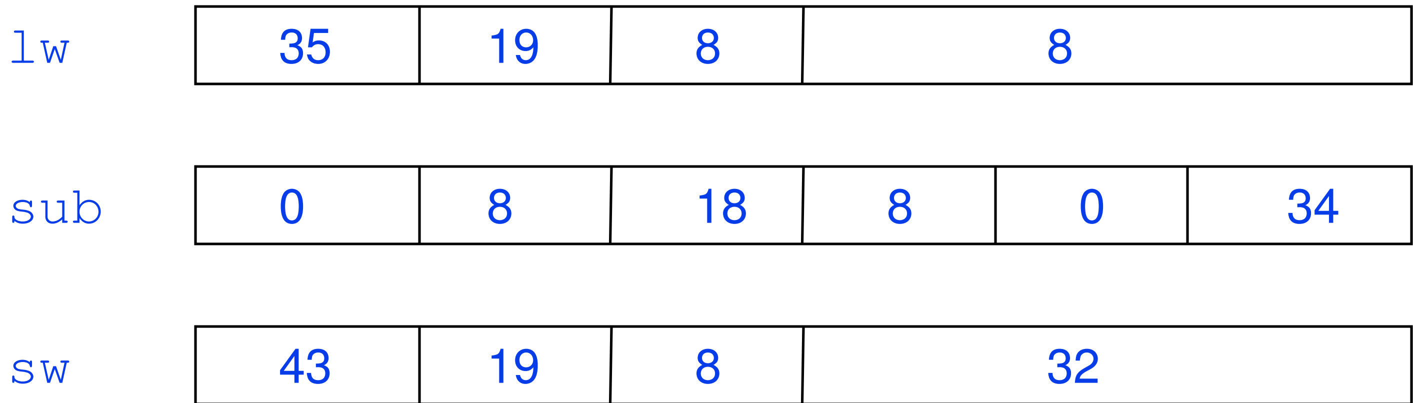
Assembling Code

- Remember the assembler code we compiled for the C statement

```
A[8] = A[2] - b
```

```
lw    $t0, 8($s3)      #load A[2] into $t0
sub   $t0, $t0, $s2    #subtract b from A[2]
sw    $t0, 32($s3)     #store result in A[8]
```

Assemble the MIPS code for these three instructions



Review: MIPS Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

4 bits is a nibble
8 bits is a byte
16 bits is a half-word
32 bits is a word
64 bits is a double-word

Character:

ASCII 7 bit code

Decimal:

digits 0-9 encoded as 0000b thru 1001b
two decimal digits packed per 8 bit byte

Integers: 2's complement

Floating Point

Beyond Numbers

- Most computers use 8-bit bytes to represent characters with the American Std Code for Info Interchange (ASCII)

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
0	Null	32	space	48	0	64	@	96	`	112	p
1		33	!	49	1	65	A	97	a	113	q
2		34	“	50	2	66	B	98	b	114	r
3		35	#	51	3	67	C	99	c	115	s
4	EOT	36	$	52	4	68	D	100	d	116	t
5		37	%	53	5	69	E	101	e	117	u
6	ACK	38	&	54	6	70	F	102	f	118	v
7		39	‘	55	7	71	G	103	g	119	w
8	bksp	40	(	56	8	72	H	104	h	120	x
9	tab	41	)	57	9	73	I	105	i	121	y
10	LF	42	*	58	:	74	J	106	j	122	z
11		43	+	59	;	75	K	107	k	123	{
12	FF	44	,	60	<	76	L	108	l	124	|
15		47	/	63	?	79	O	111	o	127	DEL

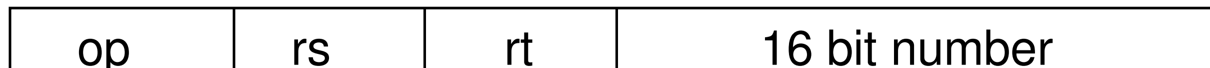
- So, we need instructions to move bytes around

Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)   #load byte from memory

sb    $t0, 6($s3)   #store byte to  memory
```



- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory

Example of Loading and Storing Bytes

- Given following code sequence and memory state (contents are given in hexadecimal), what is the state of the memory after executing the code?

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

Memory (Data)	Word Address (Decimal)
0 0 0 0 0 0 0 0	24
0 0 0 0 0 0 0 0	20
0 0 0 0 0 0 0 0	16
1 0 0 0 0 0 1 0	12
0 1 0 0 0 4 0 2	8
F F F F F F F F	4
0 0 9 0 1 2 A 0	0

- What value is left in $t0?
- What if the machine were little Endian?

Example of Loading and Storing Bytes

- Given following code sequence and memory state (contents are given in hexadecimal), what is the state of the memory after executing the code?

```
add  $s3, $zero, $zero
lb   $t0, 1($s3)
sb   $t0, 6($s3)
```

Memory (Data)	Word Address (Decimal)
0 0 0 0 0 0 0 0	24
0 0 0 0 0 0 0 0	20
0 0 0 0 0 0 0 0	16
1 0 0 0 0 0 1 0	12
0 1 0 0 0 4 0 2	8
F F F F F F F F	4
0 0 9 0 1 2 A 0	0

- What value is left in $t0?

  $t0 = 0x00000090

- What if the machine were little Endian?

Example of Loading and Storing Bytes

- Given following code sequence and memory state (contents are given in hexadecimal), what is the state of the memory after executing the code?

```
add  $s3, $zero, $zero
lb   $t0, 1($s3)
sb   $t0, 6($s3)
```

mem(4) = 0xFFFF90FF

Memory (Data)	Word Address (Decimal)
0 0 0 0 0 0 0 0	24
0 0 0 0 0 0 0 0	20
0 0 0 0 0 0 0 0	16
1 0 0 0 0 0 1 0	12
0 1 0 0 0 4 0 2	8
F F F F F F F F	4
0 0 9 0 1 2 A 0	0

- What value is left in $t0?

  $t0 = 0x00000090

- What if the machine were little Endian?

Example of Loading and Storing Bytes

- Given following code sequence and memory state (contents are given in hexadecimal), what is the state of the memory after executing the code?

```
add  $s3, $zero, $zero
lb   $t0, 1($s3)
sb   $t0, 6($s3)
```

mem(4) = 0xFFFF90FF

Memory (Data)	Word Address (Decimal)
0 0 0 0 0 0 0 0	24
0 0 0 0 0 0 0 0	20
0 0 0 0 0 0 0 0	16
1 0 0 0 0 0 1 0	12
0 1 0 0 0 4 0 2	8
F F F F F F F F	4
0 0 9 0 1 2 A 0	0

- What value is left in $t0?

  $t0 = 0x00000090

- What if the machine were little Endian?

  mem(4) = 0xFF12FFFF
  $t0 = 0x00000012

MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
Data transfer (I format)	load word	35	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	43	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	32	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	40	sb $s1, 101($s2)	Memory($s2+101) = $s1

Instructions for Making Decisions

- Decision making instructions
  - alter the control flow
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $s0, $s1, Label     #go to Label if $s0≠$s1
beq $s0, $s1, Label     #go to Label if $s0=$s1
```

- Example: `if (i==j) h = i + j;`

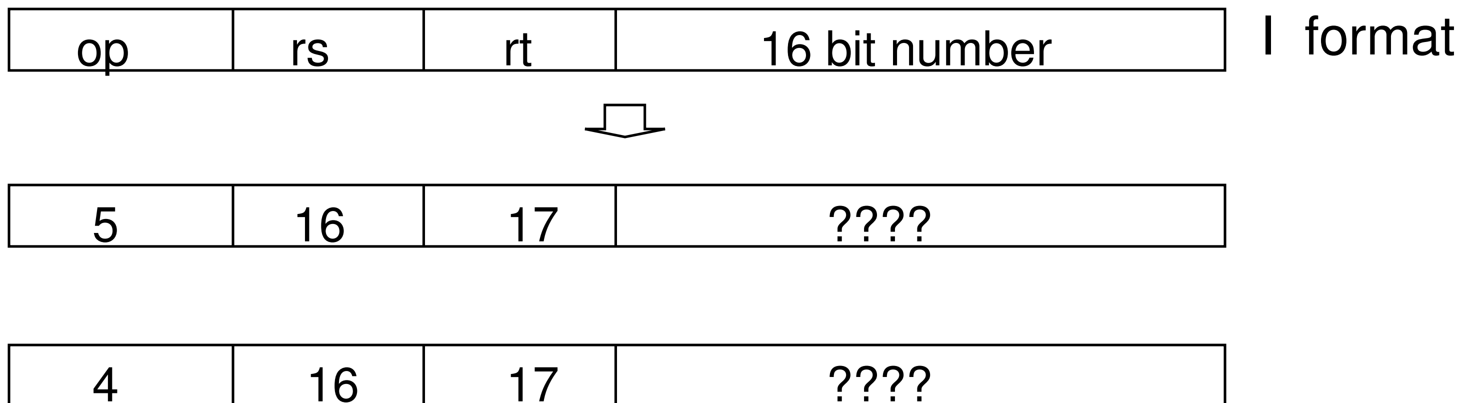
```
            bne $s0, $s1, Lab1
            add $s3, $s0, $s1
Lab1: ...
```

Assembling Branches

- Instructions:

```
bne $s0, $s1, Label   #go to Label if $s0≠$s1
beq $s0, $s1, Label   #go to Label if $s0=$s1
```

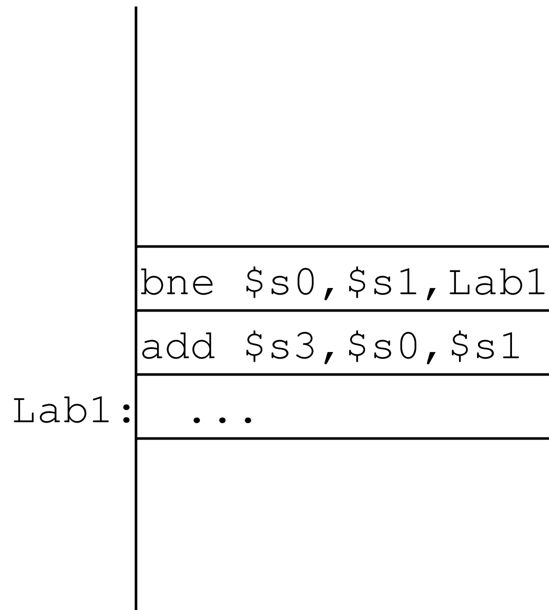
- Machine Formats:



- How is the branch destination address specified?

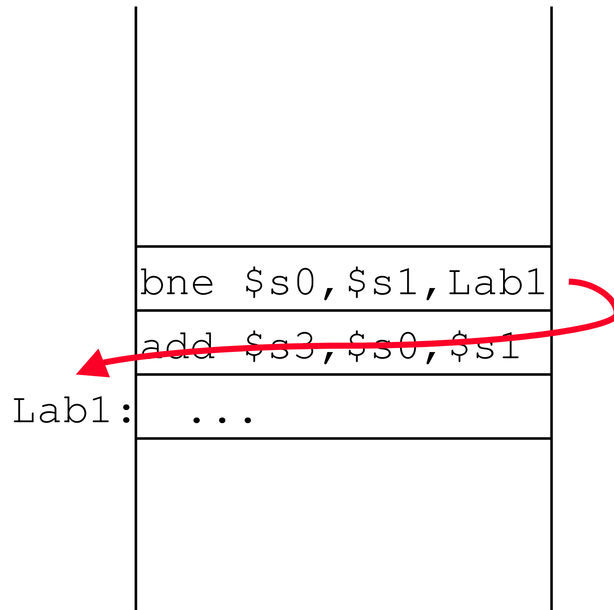
Specifying Branch Destinations

- Could specify the memory address - but that would require a 32 bit field



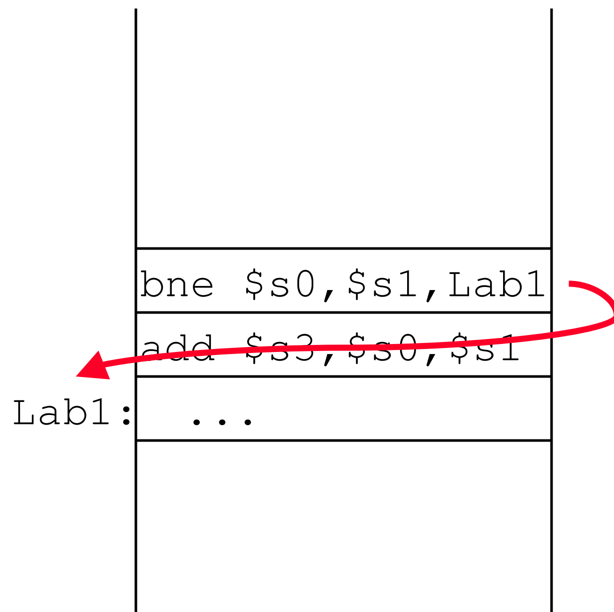
Specifying Branch Destinations

- Could specify the memory address - but that would require a 32 bit field



Specifying Branch Destinations

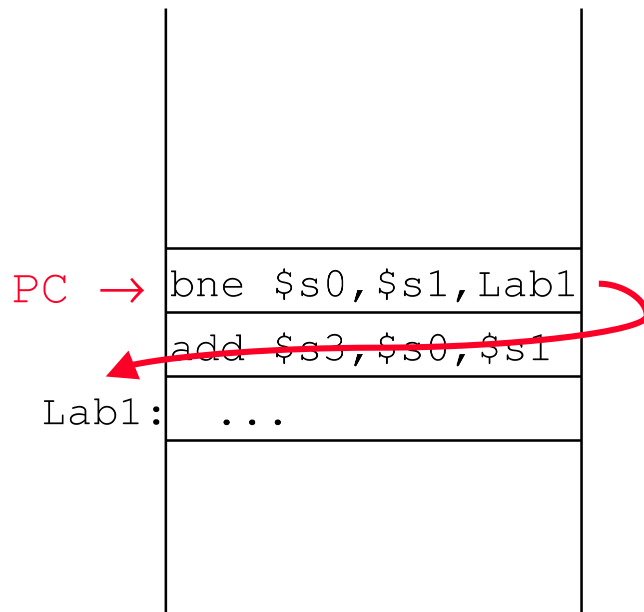
- Could specify the memory address - but that would require a 32 bit field



- Could use a register (like lw and sw) and add to it the 16-bit offset
  - which register?
    - Instruction Address Register (PC = program counter)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but
    - most branches are local anyway (principle of locality)

Specifying Branch Destinations

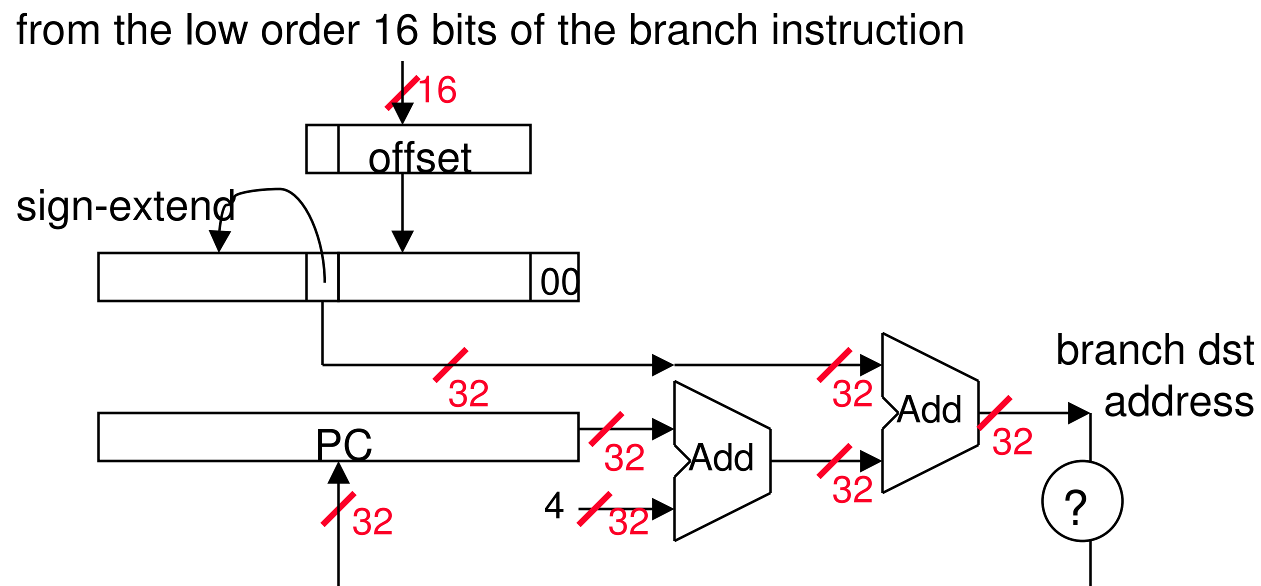
- Could specify the memory address - but that would require a 32 bit field



- Could use a register (like lw and sw) and add to it the 16-bit offset
  - which register?
    - Instruction Address Register (PC = program counter)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but
    - most branches are local anyway (principle of locality)

Disassembling Branch Destinations

- The contents of the updated PC (PC+4) is added to the low order 16 bits of the branch instruction which is converted into a 32 bit value by
  - concatenated two low-order zeros to create an 18 bit number
  - sign-extending those 18 bits
- The result is written into the PC if the branch condition is true prior to the next Fetch cycle

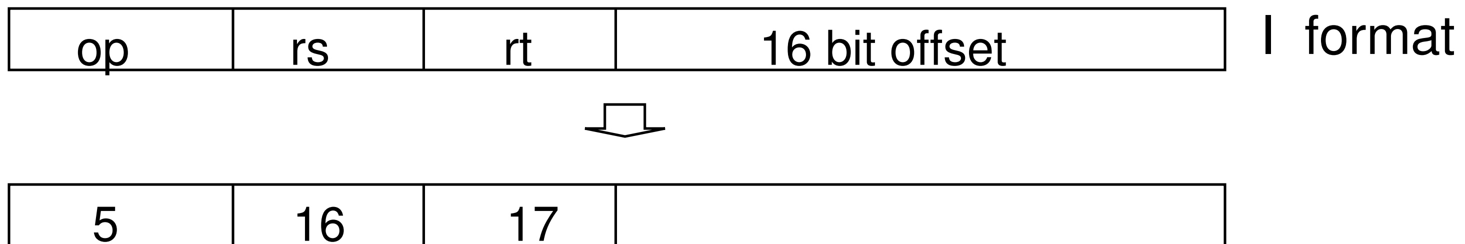


Assembling Branches Example

- Assembly code

```
            bne $s0, $s1, Lab1
            add $s3, $s0, $s1
Lab1:       ...
```

- Machine Format of `bne`:

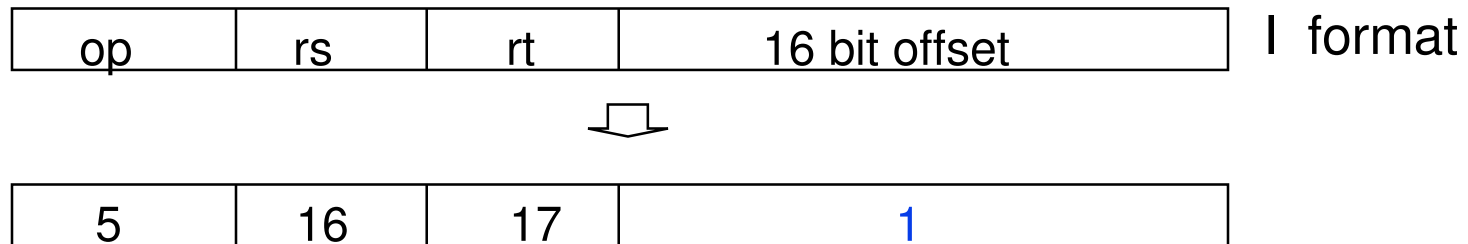


Assembling Branches Example

- Assembly code

```
            bne $s0, $s1, Lab1
            add $s3, $s0, $s1
Lab1:       ...
```

- Machine Format of `bne`:

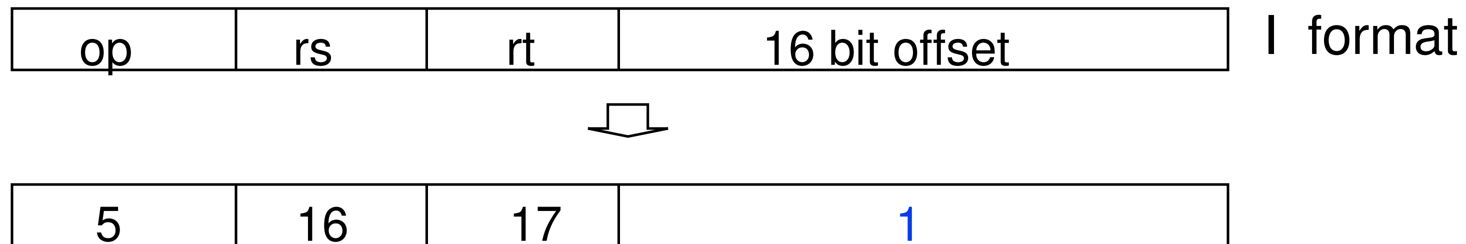


Assembling Branches Example

- Assembly code

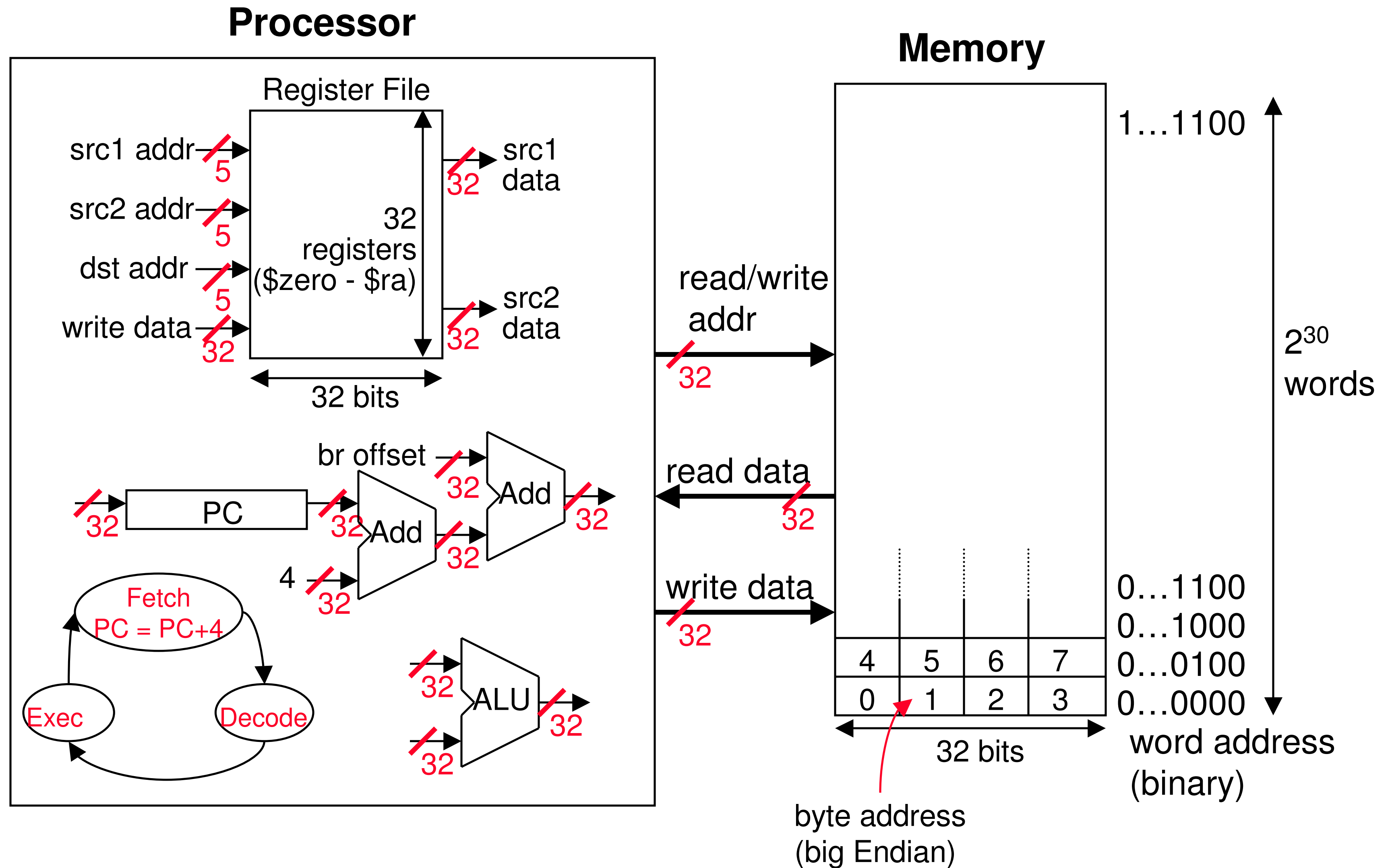
```
            bne $s0, $s1, Lab1
            add $s3, $s0, $s1
Lab1:       ...
```

- Machine Format of `bne`:



- Remember
  - After the `bne` instruction is fetched, the PC is updated to address the `add` instruction (PC = PC + 4).
  - Two low-order zeros are concatenated to the offset number and that value, sign-extended, is added to the (updated) PC

MIPS Organization



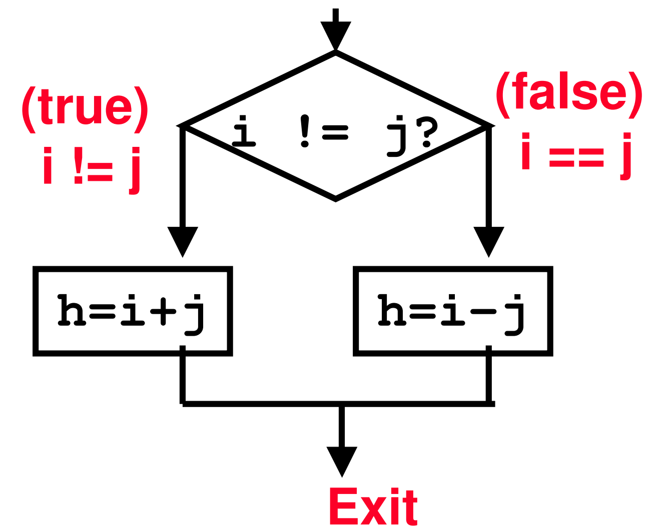
Another Instruction for Changing Flow

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label         #go to label
```

- Example:

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



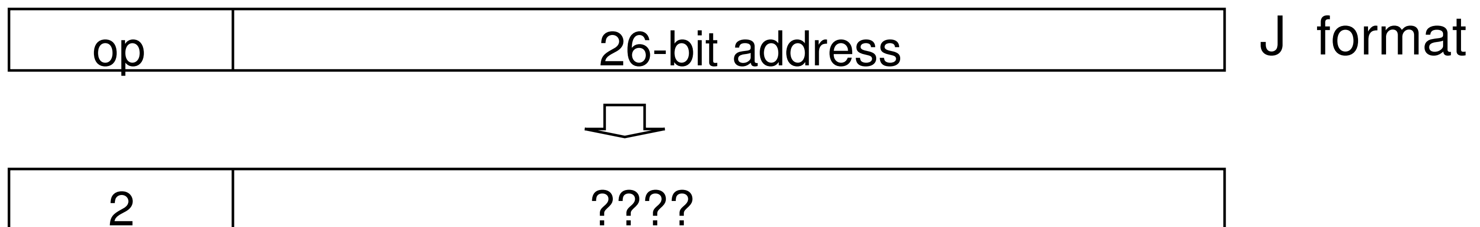

```
        beq   $s0, $s1, Lab1
        add   $s3, $s0, $s1
        j     Lab2
Lab1:   sub   $s3, $s0, $s1
Lab2:   ...
```

Assembling Jumps

- Instruction:

```
j  label        #go to label
```

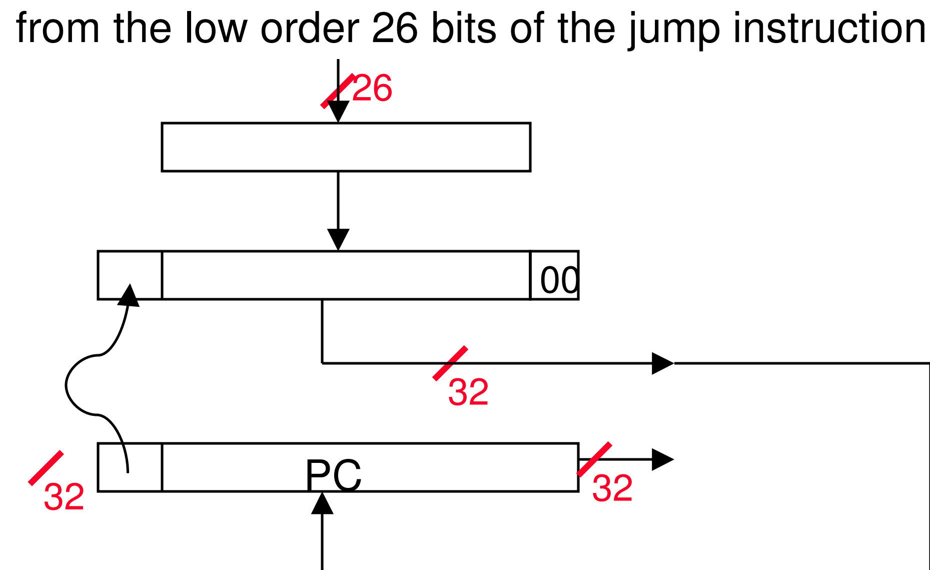
- Machine Format:



- How is the jump destination address specified?
  - As an absolute address formed by
    - concatenating the upper 4 bits of the current PC (now PC+4) to the 26-bit address and
    - concatenating 00 as the 2 low-order bits

Disassembling Jump Destinations

- The low order 26 bits of the jump instruction is converted into a 32 bit jump instruction destination address by
  - concatenated two low-order zeros to create a 28 bit (word) address and
  - concatenating the upper 4 bits of the current PC (now PC+4)
- to create a 32 bit instruction address that is placed into the PC prior to the next Fetch cycle



Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
            beq   $s0, $s1, Lab1
            add   $s3, $s0, $s1
            j     Lab2
Lab1:       sub   $s3, $s0, $s1
Lab2:       ...
```

Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
            beq    $s0, $s1, Lab1
            add    $s3, $s0, $s1
            j      Lab2
Lab1:       sub    $s3, $s0, $s1
Lab2:       ...
```

```
0x00400020          4     16     17          2
```

Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
        beq   $s0, $s1, Lab1
        add   $s3, $s0, $s1
        j     Lab2
Lab1:   sub   $s3, $s0, $s1
Lab2:   ...
```

0x00400020	4	16	17		2	
0x00400024	0	16	17	19	0	32

Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
            beq    $s0, $s1, Lab1
            add    $s3, $s0, $s1
            j      Lab2
Lab1:       sub    $s3, $s0, $s1
Lab2:       ...
```

```
0x00400020        4     16     17             2
0x00400024        0     16     17     19      0     32
0x00400028        2     0000 0100 0 ... 0 0011 00₂
```

Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
          beq   $s0, $s1, Lab1
          add   $s3, $s0, $s1
          j     Lab2
Lab1:     sub   $s3, $s0, $s1
Lab2:     ...
```

0x00400020	4	16	17		2	
0x00400024	0	16	17	19	0	32
0x00400028	2	0000 0100 0 ... 0 0011 00_2				

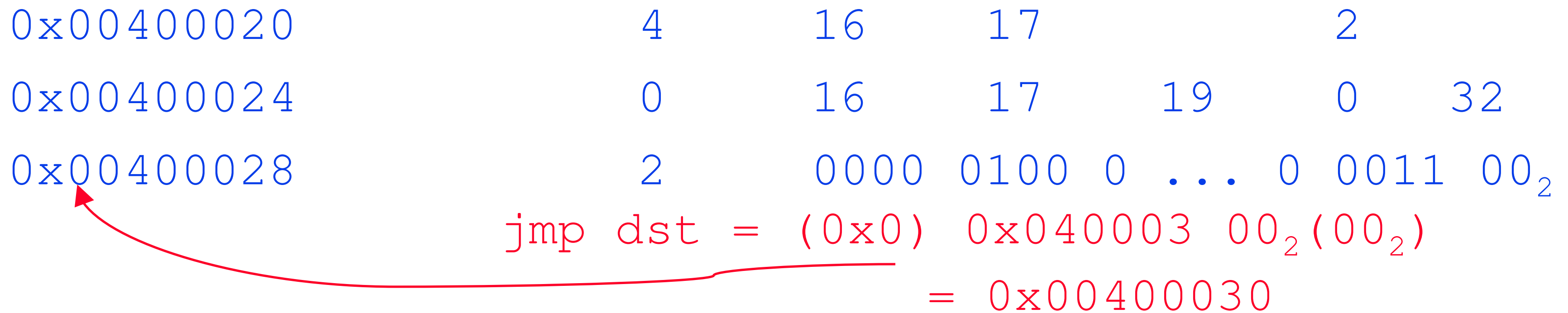
jmp dst = (0x0) 0x040003 00_2(00_2)

= 0x00400030

Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

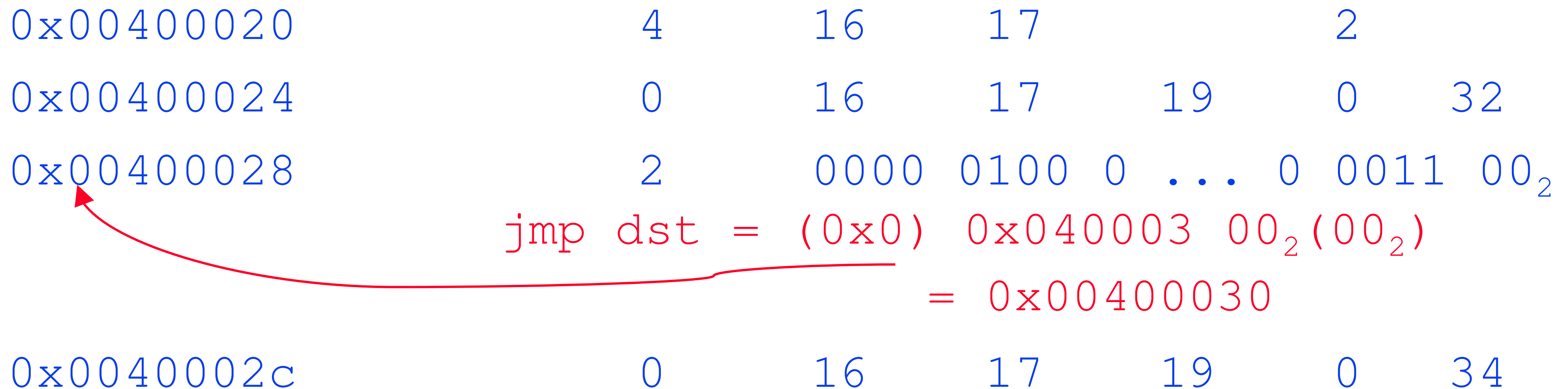
```
            beq   $s0, $s1, Lab1
            add   $s3, $s0, $s1
            j     Lab2
Lab1:       sub   $s3, $s0, $s1
Lab2:       ...
```



Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

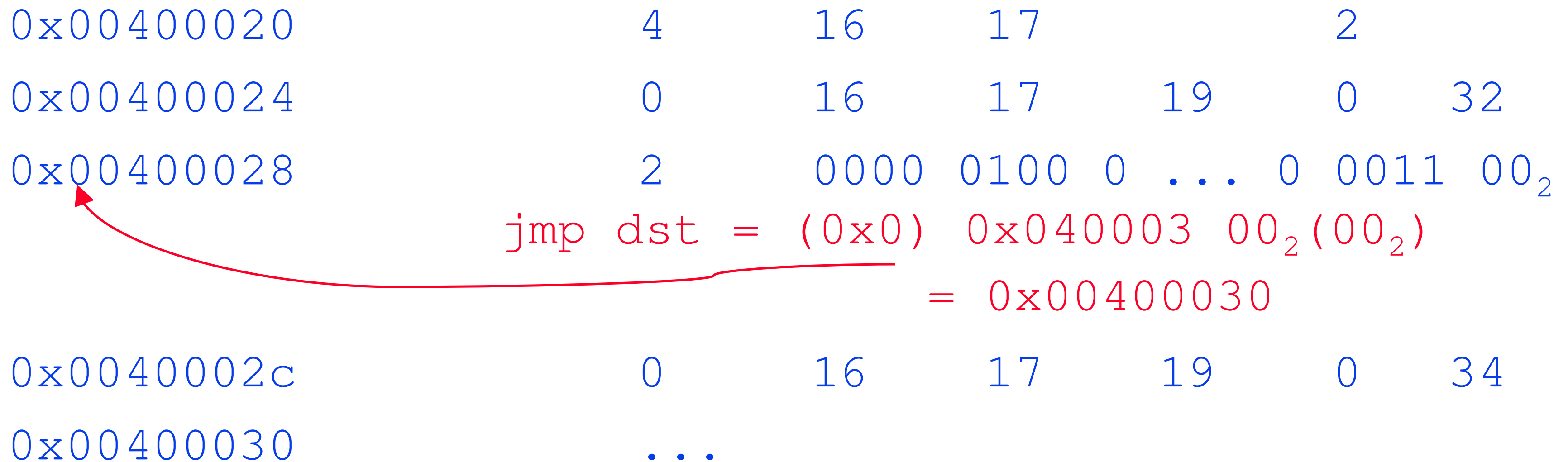
```
            beq   $s0, $s1, Lab1
            add   $s3, $s0, $s1
            j     Lab2
Lab1:       sub   $s3, $s0, $s1
Lab2:       ...
```



Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the address of the `beq` instruction is 0x00400020 (hex address)

```
              beq   $s0, $s1, Lab1
              add   $s3, $s0, $s1
              j     Lab2
Lab1:         sub   $s3, $s0, $s1
Lab2:         ...
```



Compiling While Loops

- Compile the assembly code for the C while loop where i is in `$s0`, j is in `$s1`, and k is in `$s2`

```
while (i!=k)
     i=i+j;
```

Compiling While Loops

- Compile the assembly code for the C while loop where i is in `$s0`, j is in `$s1`, and k is in `$s2`

```
while (i!=k)
      i=i+j;
```

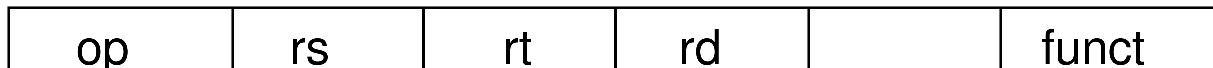
```
Loop:   beq  $s0, $s2, Exit
        add  $s0, $s0, $s1
        j    Loop
Exit:   . . .
```

More Instructions for Making Decisions

- We have `beq`, `bne`, but what about branch-if-less-than?
- New instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1
                      #     then
                      # $t0 = 1
                      #     else
                      # $t0 = 0
```

- Machine format:



op	rs	rt	rd		funct

0	16	17	8	0	42 = 0x2a

Other Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create all relative conditions
  - less than `blt $s1, $s2, Label`
  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`
- As pseudo instructions - recognized (and expanded) by the assembler
- The assembler needs a reserved register (`$at`)
  - there are policy of use conventions for registers

Other Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create all relative conditions
  - less than `blt $s1, $s2, Label`

```
slt  $t0, $s1, $s2        #$t0 set to 1 if
bne  $t0, $zero, Label    #  $s1 < $s2
```

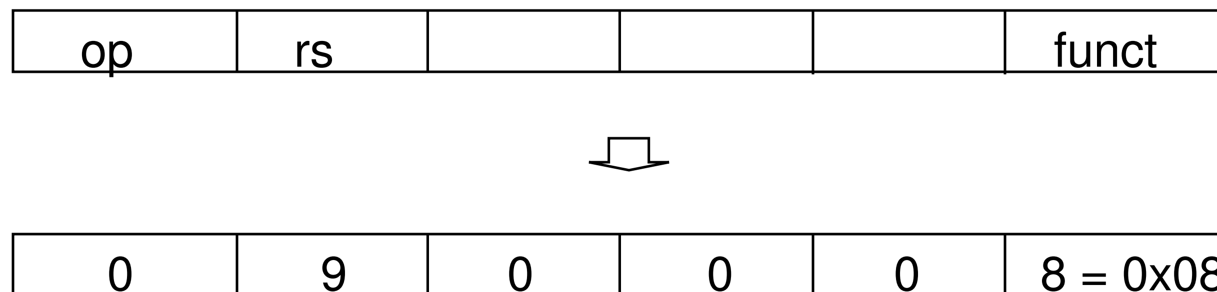
  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`
- As pseudo instructions - recognized (and expanded) by the assembler
- The assembler needs a reserved register (`$at`)
  - there are policy of use conventions for registers

Another Instruction for Changing Flow

- Most higher level languages have `case` or `switch` statements allowing the code to select one of many alternatives depending on a single value.
- Instruction:

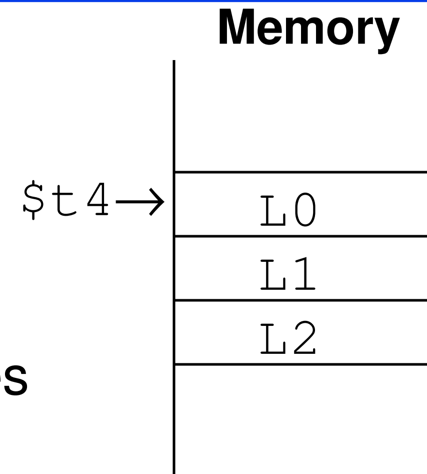
```
jr   $t1          #go to address in $t1
```

- Machine format:



Compiling a Case (Switch) 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 JT[k]
        lw      $t0, 0($t1)           #$t0 = JT[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:   . . .
```