Chapter 2

Instructions: Language of the Computer

§2.9 Communicating with People

Byte/Halfword Operations

- MIPS byte/halfword load/store are common for string processing
- `lb $t0, 0($sp)   //load with sign extension`
  - Load rs+offset address into rt
  - Sign extend to 32 bits in rt

$t0=00000000000000000000000000000000

$sp

00000000000000000000000110000000

$t0=? After lb $t0 0($sp)

11111111111111111111111110000000

- `sb $t0, 0($sp)`
  - Store just rightmost byte

$t0=

00000000000000000000000110000000

$sp

11111111111111111111111110000000

Byte/Halfword Operations

- `lbu $t0, 0($sp)`   //load without sign extension
  - Load rs+offset address into rt
  - Sign bit is not extended in rt

$sp

$t0=?

00000000000000000000000110000000   00000000000000000000000010000000

No store byte unsigned instruction (Sbu)

Other similar instruction

- `lh rt, offset(rs)    ; load half word`
  - Load rs+offset address into rt
  - Sign extend to 32 bits in rt

- `lhu rt, offset(rs) ;load half word unsigned`

- `sh rt, offset(rs) ; store halfword`
  - Store just rightmost halfword

- No store halfword unsigned instruction

32-bit Constants

- Most constants are small (16 bit range is $-2^{16}\sim 2^{16}-1$)
- Sometimes we need 32-bit constant, but a instruction can't have 32-bit constant (no space for op code)
- => combine lui and ori instruction to achieve this
- lui rt, constant
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

Question: Steps to set $s0 to 4,000,000

4000000_{10}=0000 0000 0011 1101 0000 1001 0000 0000_2

```
lui $s0, 61
```

0000 0000 0011 1101 0000 0000 0000 0000

```
ori $s0,$s0,2304
```

0000 0000 0000 0000 0000 1001 0000 0000

Finally, $s0 =

0000 0000 0011 1101 0000 1001 0000 0000

Branch Addressing (for beq, bne)

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

Addr.	Inst.	
8 (01000)	beq $t0 $t1	□
12(01100)	....	
16(10000)	.....	
20(10100)	.....	

- PC-relative addressing
  - Address is always a multiple of 4 => offset/4 is stored in the instruction
  - Target address = PC + (Address × 4)
  - PC is already incremented by 4

op	rs	rt	Address (=Offset/4)
6 bits	5 bits	5 bits	16 bits

If the above beq go to address 20 when $s0==$t1, □ =? 2

000100	01000	01001	0000 0000 0000 0010
beq	$t0	$t1	

Example

- Suppose $s0=$t1,
- (1) find target address of beq instruction
- (2) the next instruction to be executed

Addr.	Inst.
4(00100)	beq $s0 $t1 2
8(01000)	Inst1......
12(01100)	Inst2.....
16(10000)	Inst3....
20(10100)	Inst4....

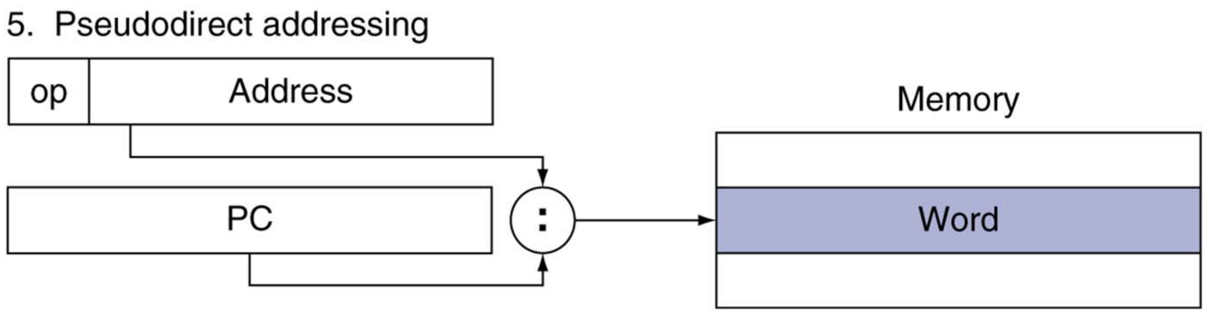
- Target address= 8+ 2*4=16
- Next instruction to be executed: inst3

Jump Addressing

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

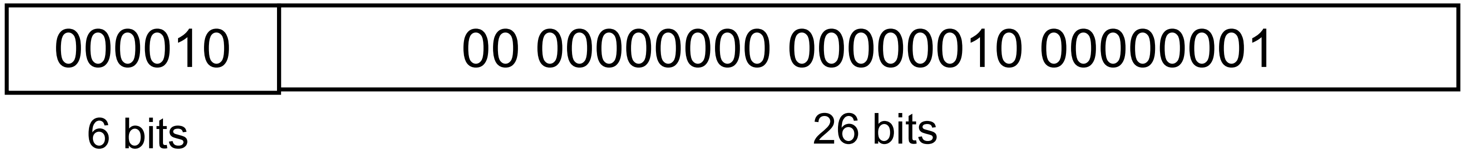
op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)



Jump example

- Assume PC=40000000_{16}, what is the target address of the jump instruction?



000010	00 00000000 00000010 00000001
6 bits	26 bits

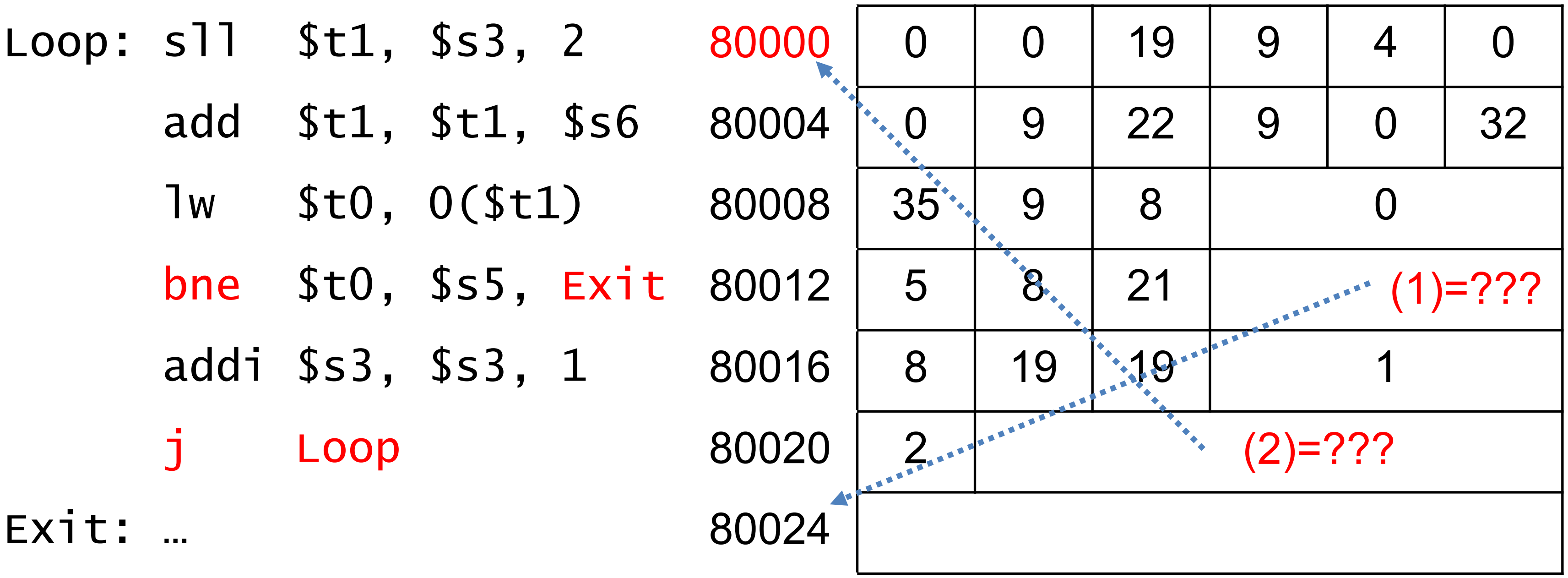
Address in the instruction= 0x0000201

Target Address= PC[31:28]+0021_{16}*4= 0x40000804

Hidden

Target Addressing Example

- Loop code from earlier example (assume PC[31:28]=0000), what is the value of (1) and (2)
  - Assume Loop at location 80000



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

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	(1)=???		
80016	8	19	19	1		
80020	2	(2)=???				
80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler insert an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.
- Example

```
beq $s0,$s1, L1
```

⇨ L1 can only be 16bit address

↓

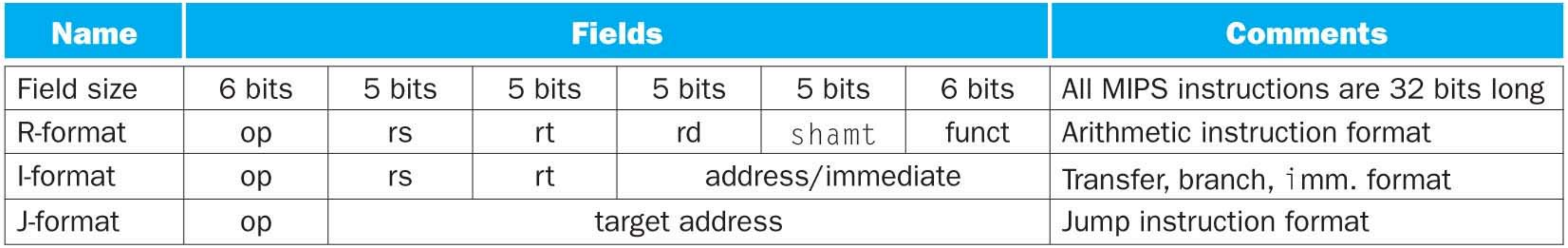
```
bne $s0,$s1, L2
        j L1
L2:     …
```

⇨ L2 can only be 26bit

j can jump farer than beq

Summary: Instruction format

- R-format: add, and, or …
- I-format: beq, bneq, addi, …
- J-format: j, jal

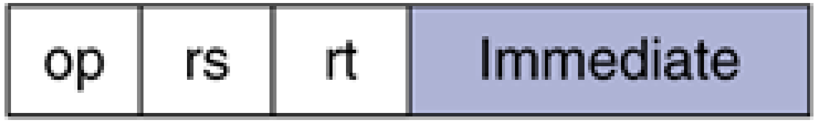


Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Summary: Addressing Mode

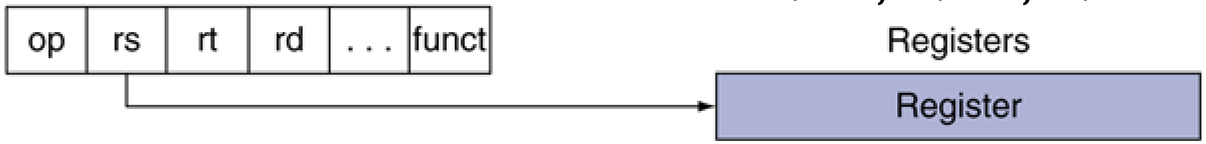
- Immediate addressing: operand is a constant within the instruction (e.g. addi)

```
addi $s1, $s0, 1   # s1 = s0+1
```



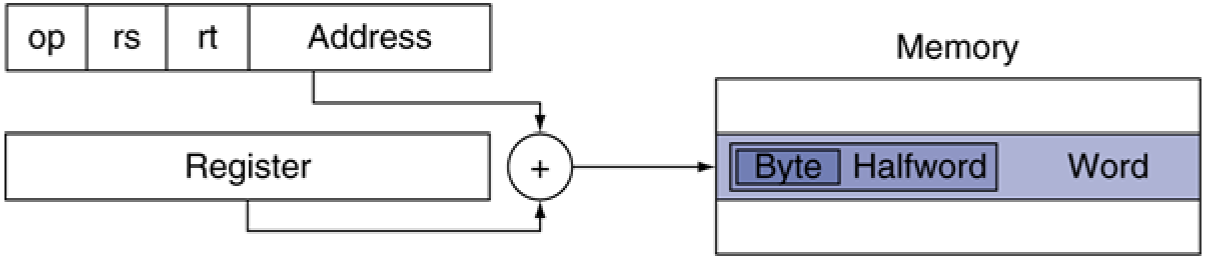
- Register addressing: operand is a register (e.g. add, nor)

```
add $s1, $s0, $s2
```



- Base or displacement addressing: operand is at the memory location (e.g. lw, sw)

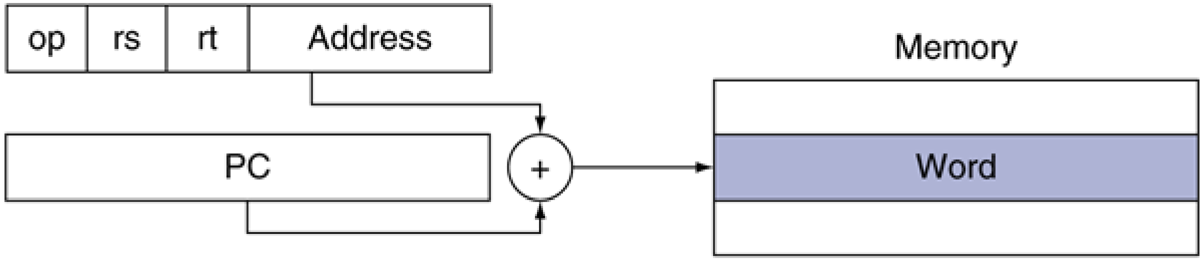
```
lw $t0, 32($s3)
```



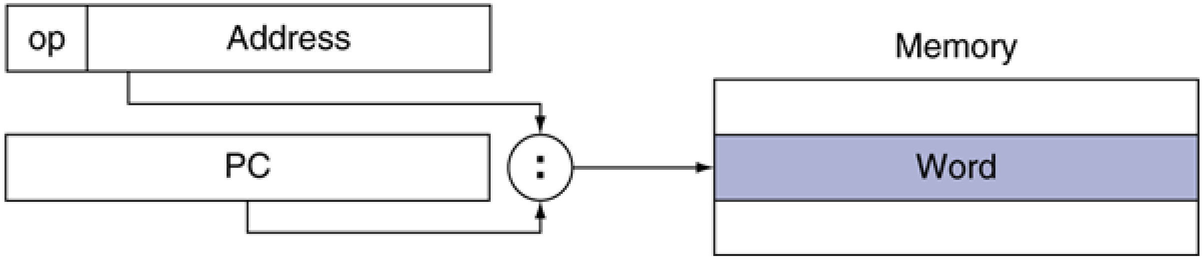
Summary: Addressing Mode (2)

- PC-relative addressing: branch address is the sum of PC and constant (e.g. beq)

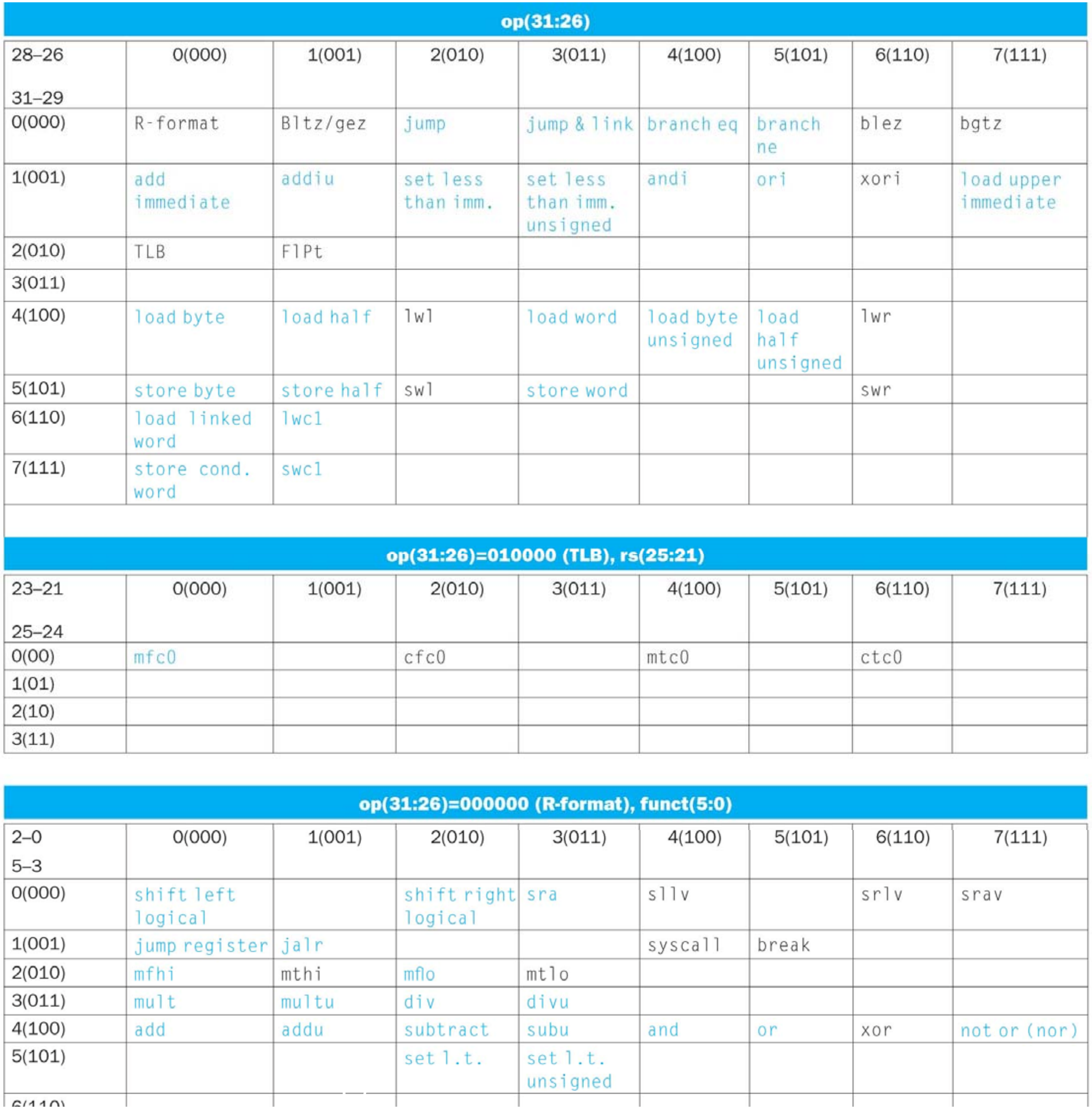
beq $s0,$s1, L1



- (Pseudo)direct addressing: jump address is 26 bit of instruction + PC (e.g. j)



Decoding MIPS instruction



op(31:26)

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=010000 (TLB), rs(25:21)

23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

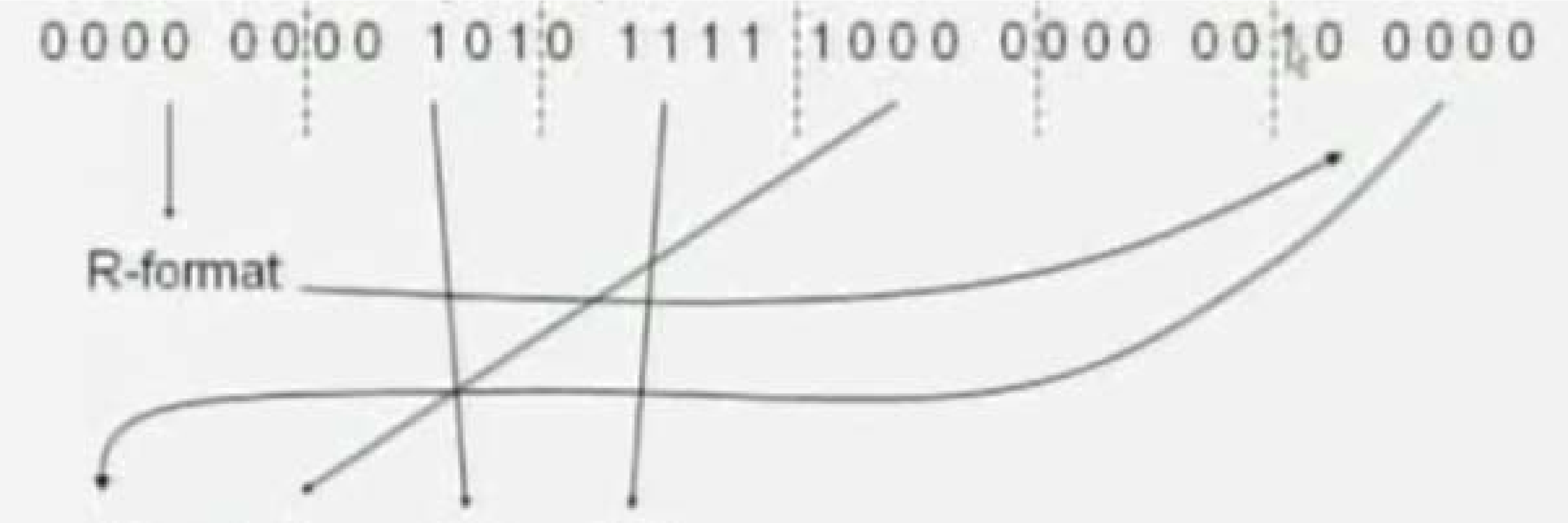
op(31:26)=000000 (R-format), funct(5:0)

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								

Decoding Machine Code

- What’s the assembly code represent?

  00af8020 (hex)



add $s0 $a1 $t7

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- However: some useful instructions may be missing
  - Can be achieved using by other instructions
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1      →  add $t0, $zero, $t1
blt $t0, $t1, L     →  slt $at, $t0, $t1
                       bne $at, $zero, L
```

  - $at (register 1) is reserved for assembler

More pseudoinstructions in MIPS

- blt (branch less than), bgt (branch greater than), ble (branch less than and equal to), bge (branch great than and equal to)
- neg: changes the mathematical sign of the number
- not: bitwise logical negation
- li: loads an immediate value into a register

```
li $t0, 0x3BF20    →    lui $t0, 0x0003
                        ori $t0, $t0, 0xBF20
```

- sge (set greater than and equal to), sgt (set great than). See references for more details

Exercise

- What is the range of addresses for conditional branches in MIPS (K=1024)?

1. Address between 0 and 64K -1
2. Address between 0 and 256K -1
3. Address up to about 32K before the branch to about 32K after
4. Addresses up to about 128K before the branch to about 128K after

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

Answer: 4 $2^{16}*4=2^{18}$ = about -128K ~ 128K

Exercise -2

- What is the range of addresses for jump and jump and link in MIPS (K=1024)?

1. Address between 0 and 64M -1
2. Address between 0 and 256M -1
3. Address up to about 32M before the branch to about 32M after
4. Addresses up to about 128M before the branch to about 128M after
5. Anywhere within a block of 64M addresses where the PC supplies the upper 6 bits
6. Anywhere within a block of 256M addresses where the PC supplies the upper 4bits

000010	00 00000000 00000010 00000001
6 bits	26 bits

Answer: $2^{26}*4=2^{28}$ =256MB. The answer is 6

C Sort Example

§2.13 A C Sort Example to Put It All Together

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Translate Swap procedure assuming
v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

v in $a0,
k in $a1,
temp in $t0

```
swap: sll $t1, $a1, 2   # $t1 = k * 4 (k is in $a1)
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
          void sort (int v[], int n)
          {
            int i, j;
for1tst ⇒   for (i = 0; i < n; i += 1) {
for2tst ⇒     for (j = i – 1;
                   j >= 0 && v[j] > v[j + 1];
                   j -= 1) {
                swap(v,j);
              }
            }
          }
```

Translate sort procedure assuming
v in $a0, n in $a1, i in $s0, j in $s1

The Sort Procedure

```
sort:    addi $sp,$sp, -20      # make room on stack for 5 registers
         sw $ra, 16($sp)        # save $ra on stack
         sw $s3,12($sp)         # save $s3 on stack
         sw $s2, 8($sp)         # save $s2 on stack
         sw $s1, 4($sp)         # save $s1 on stack
         sw $s0, 0($sp)         # save $s0 on stack
         …                      # procedure body
         …
  exit1: lw $s0, 0($sp)         # restore $s0 from stack
         lw $s1, 4($sp)         # restore $s1 from stack
         lw $s2, 8($sp)         # restore $s2 from stack
         lw $s3,12($sp)         # restore $s3 from stack
         lw $ra,16($sp)         # restore $ra from stack
         addi $sp,$sp, 20       # restore stack pointer
         jr $ra                 # return to calling routine
```

The Procedure Body

```
         move $s2, $a0          # save $a0 into $s2
         move $s3, $a1          # save $a1 into $s3
         move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2       # $t1 = j * 4
         add  $t2, $s2, $t1     # $t2 = v + (j * 4)
         lw   $t3, 0($t2)       # $t3 = v[j]
         lw   $t4, 4($t2)       # $t4 = v[j + 1]
         slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
         move $a0, $s2          # 1st param of swap is v (old $a0)
         move $a1, $s1          # 2nd param of swap is j
         jal  swap              # call swap procedure
         addi $s1, $s1, –1      # j –= 1
         j    for2tst           # jump to test of inner loop
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```

Move params

Outer loop

Inner loop

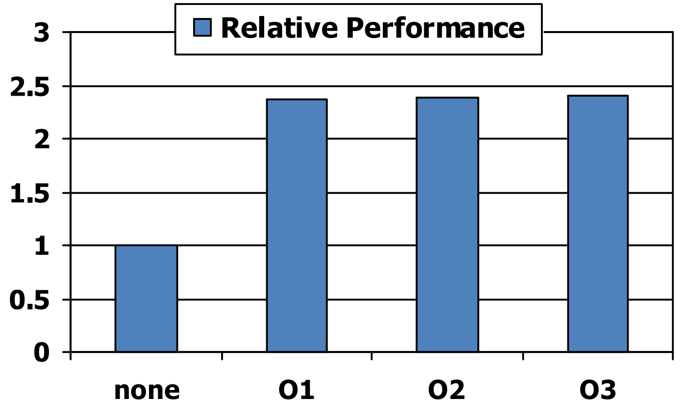
Pass params & call

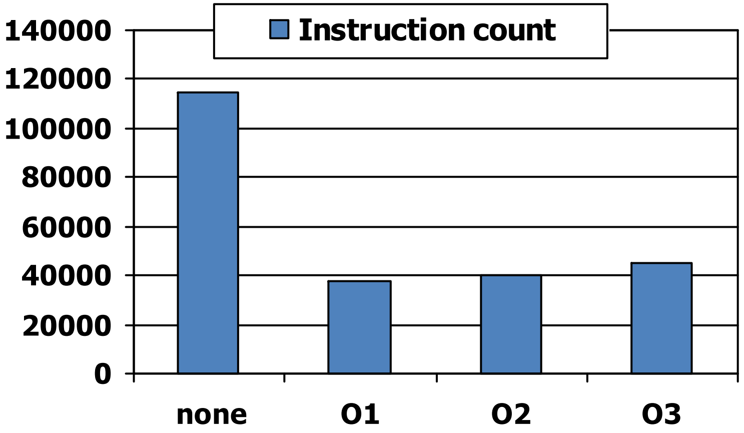
Inner loop

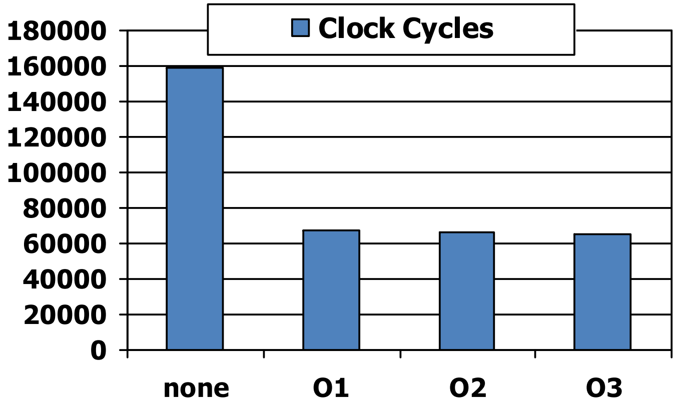
Outer loop

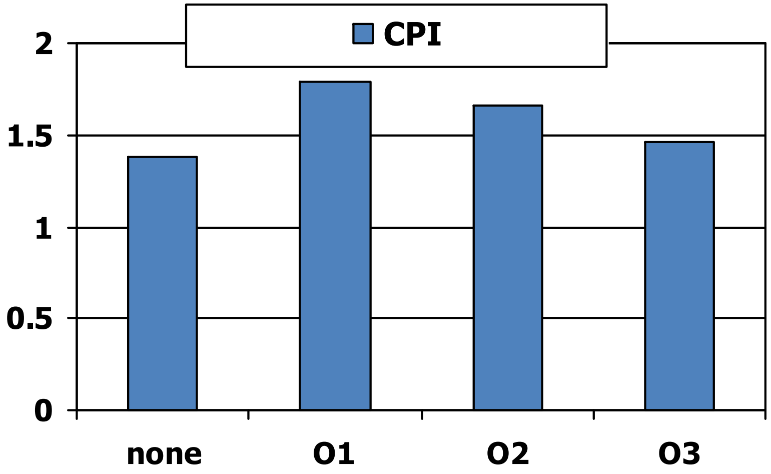
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux









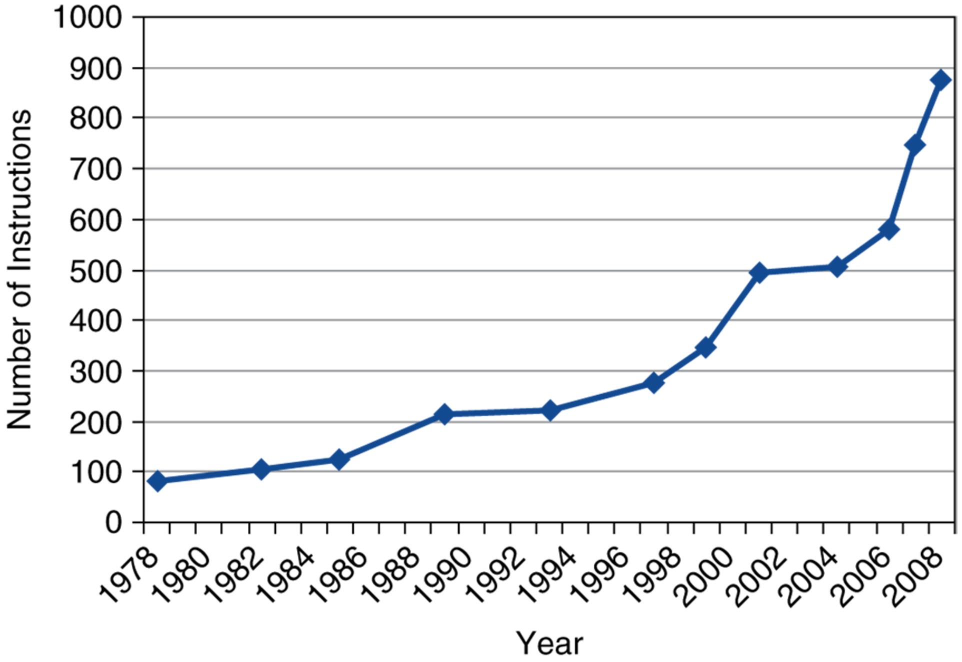
Instruction count and CPI are not good performance indicators in isolation

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
  - – But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses=>increment by 4, not by 1!

```
g = h + A[8];
```

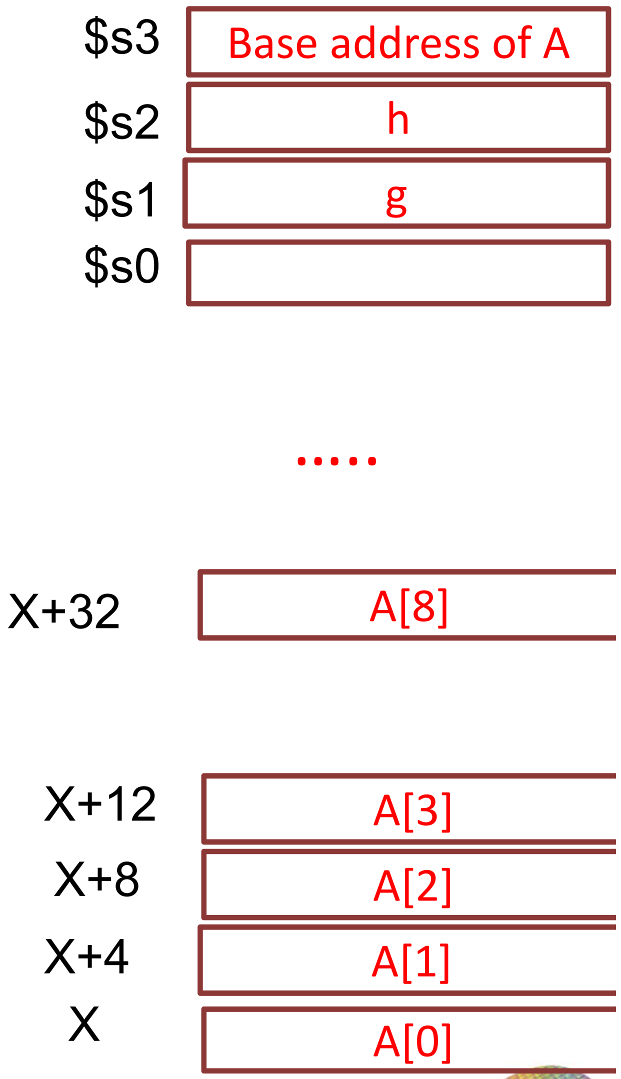
  - *g in $s1, h in $s2, base address of A in $s3*

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw $t0, 32($s3) #load word
add $s1, s2, $t0
```

offset

base register



$s3	Base address of A
$s2	h
$s1	g
$s0	

.....

X+32	A[8]
X+12	A[3]
X+8	A[2]
X+4	A[1]
X	A[0]

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Exercise

Backup slides