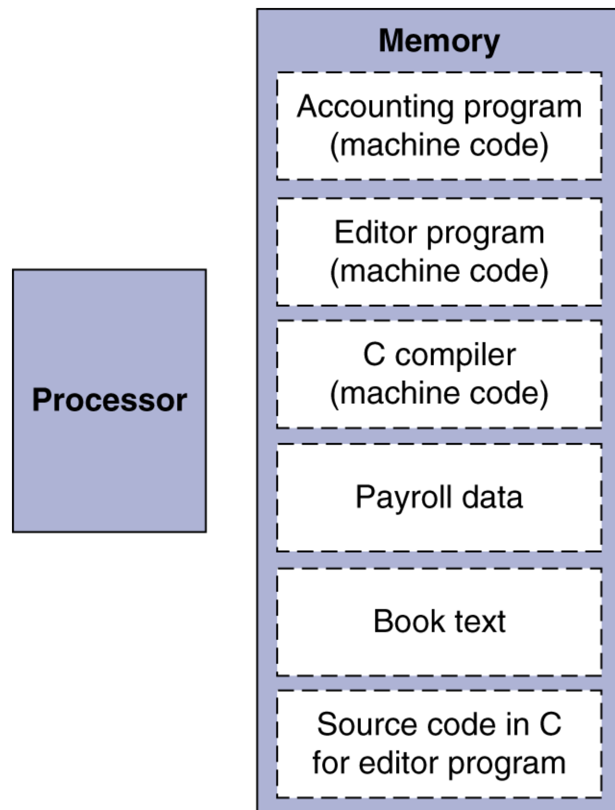
Chapter 2

Instructions: Language of the Computer

Chapter 2 (Continue)

- Representing Instructions (Instruction Format)

Stored Program Computers

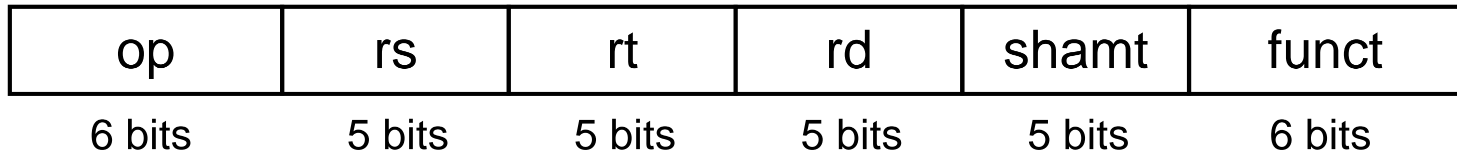


- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg’s 8 – 15
  - $t8 – $t9 are reg’s 24 – 25
  - $s0 – $s7 are reg’s 16 – 23

1) MIPS R-format Instructions



- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example -Arithmetic

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

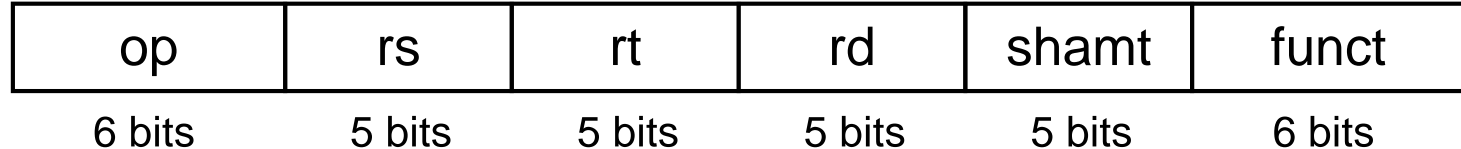
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

R-format Example -Shift Operations



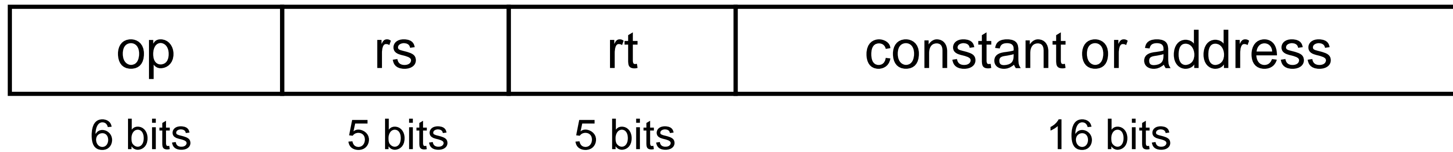
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i

R-format Example -Shift Operations

sll $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

2) MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: **offset** added to **base address** in **rs**
- ***Design Principle 4: Good design demands good compromises***
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

MIPS Instruction Encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a." (not applicable) means this field does not appear in this format.

Example on Representation

If $t1 has the base of the array A and $s2 corresponds to h, the assignment statement

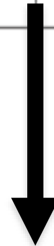
A[300] = h + A[300]; is compiled into

```
lw $t0,1200($t1)
add $t0,$s2,$t0
sw $t0,1200($t1)
```

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Example on Representation cont.

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

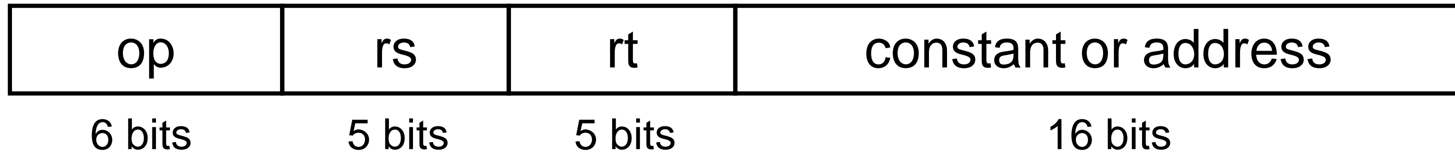


100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS Machine Language so far

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

32-bit Constants



- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rt, constant`

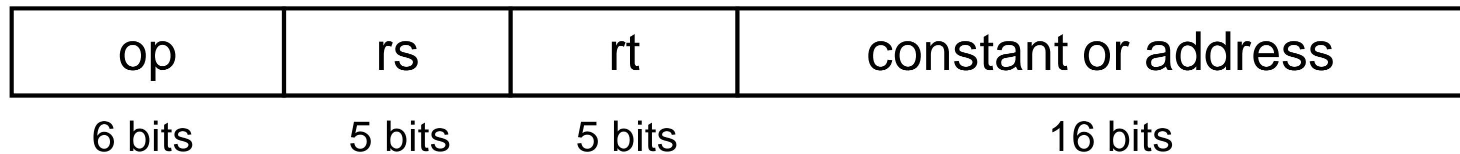
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

Branch Addressing

Branch on equal : beq r1, r2, L
Branch on not equal: bne r1,r2,L

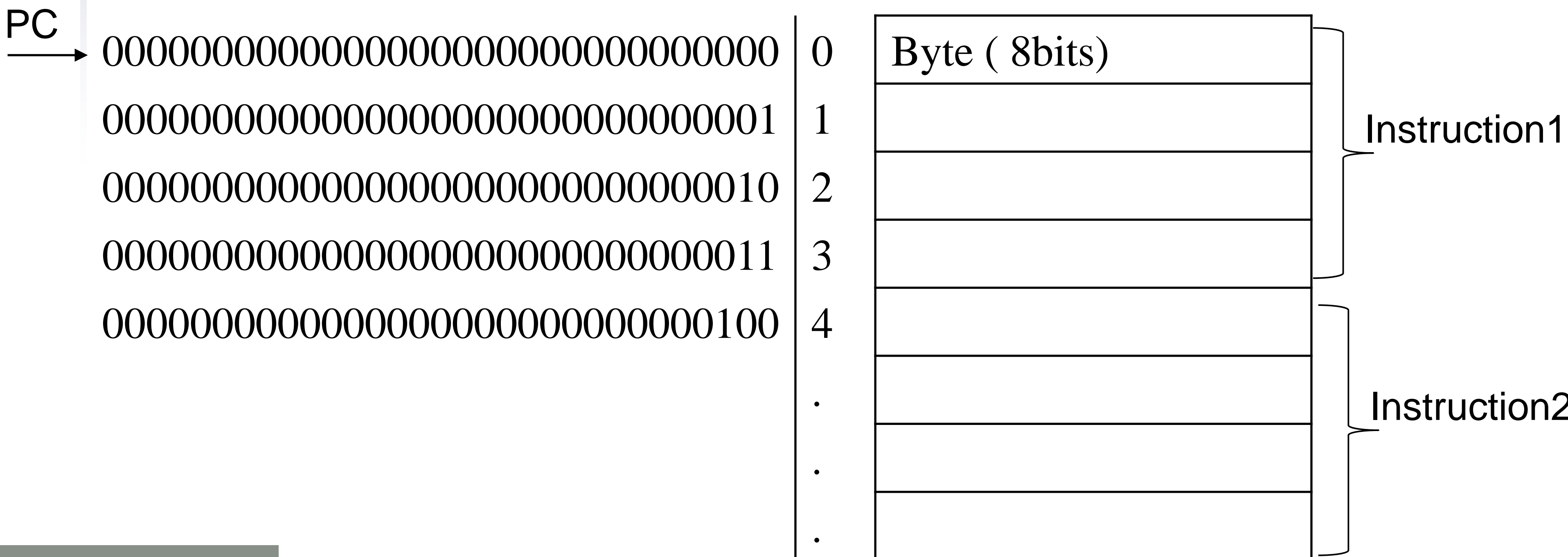
PC relative brach



- PC-relative addressing
  - Target address = PC+4 + address × 4

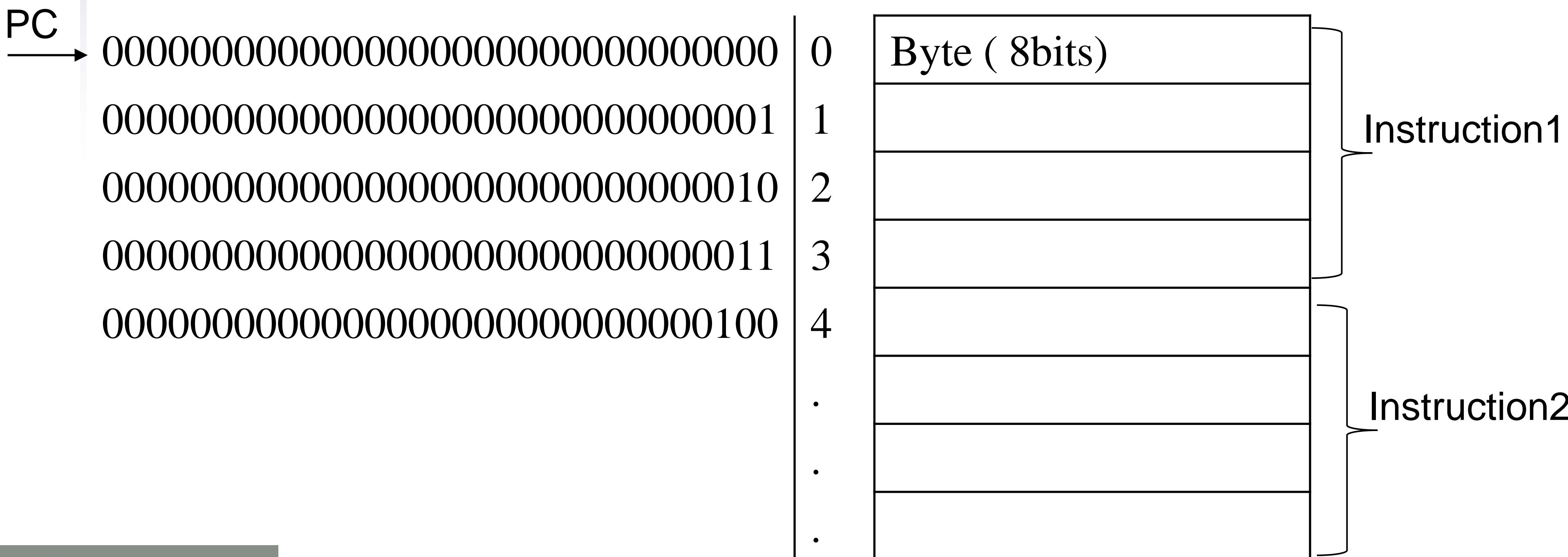
Program Counter and control flow

- Every machine has a program counter (called PC) that points to the next instruction to be executed.



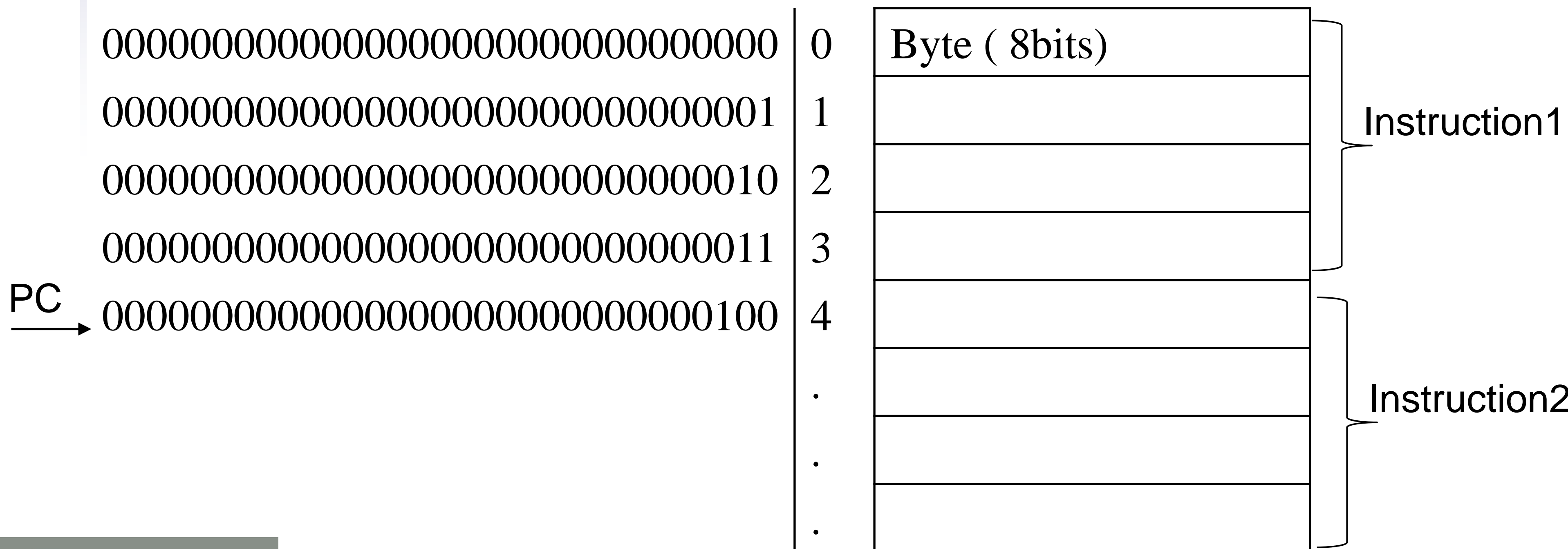
Program Counter and control flow

- Ordinarily, PC is incremented by 4 after each instruction is executed. A branch instruction alters the flow of control by modifying the PC.



Program Counter and control flow

- Ordinarily, PC is incremented by 4 after each instruction is executed. A branch instruction alters the flow of control by modifying the PC.



Program Counter and control flow

- ```
  bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
  Else: sub $s0, $s1, $s2
  Exit: …
  ```

PC label	Address (32bit)		Instruction Memory
PC →	00000000000000000000000000000000	0	Instruction 1: **bne**
PC+4 →	00000000000000000000000000000100	4	Instruction 2 **add**
	00000000000000000000000000001000	8	Instruction 3 **j**
	00000000000000000000000000001100	12	Instruction 4 Else: **sub**
	00000000000000000000000000010000	16	Instruction 5
		.	
  ```

Program Counter and control flow

- ```
  bne $s3, $s4, 2
         add $s0, $s1, $s2
         j   Exit
  Else:  sub $s0, $s1, $s2
  Exit:  …
  ```

	Address (32bit)		Instruction Memory
PC →	00000000000000000000000000000000	0	Instruction 1: **bne**
PC+4 →	00000000000000000000000000000100	4	Instruction 2 **add**
	00000000000000000000000000001000	8	Instruction 3 **j**
	00000000000000000000000000001100	12	Instruction 4 Else: **sub**
	00000000000000000000000000010000	16	Instruction 5
		.	

MK MORGAN KAUFMANN
  ```

Program Counter and control flow

- `bne $s3, $s4, 2`

op	rs	rt	address
5	19	20	2

```
      j    Exit
Else: sub $s0, $s1, $s2
Exit: …
```

PC label	Address (32bit)		Instruction Memory
PC →	00000000000000000000000000000000	0	Instruction 1: **bne**
PC+4 →	00000000000000000000000000000100	4	Instruction 2 **add**
	00000000000000000000000000001000	8	Instruction 3 **j**
	00000000000000000000000000001100	12	Instruction 4 Else: **sub**
	00000000000000000000000000010000	16	Instruction 5
		.	

MK MORGAN KAUFMANN

Program Counter and control flow

- `bne $s3, $s4, 2`

op	rs	rt	address
5	19	20	2

- Target address = PC+4 + address × 4
  = 4 + 2 ×4 =12

PC	Address (32bit)		Instruction Memory
PC →	00000000000000000000000000000000	0	Instruction 1: **bne**
PC+4 →	00000000000000000000000000000100	4	Instruction 2 **add**
	00000000000000000000000000001000	8	Instruction 3 **j**
	00000000000000000000000000001100	12	Instruction 4 Else: **sub**
	00000000000000000000000000010000	16	Instruction 5
		.	

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

$-2^{15} \rightarrow +2^{15}-1$

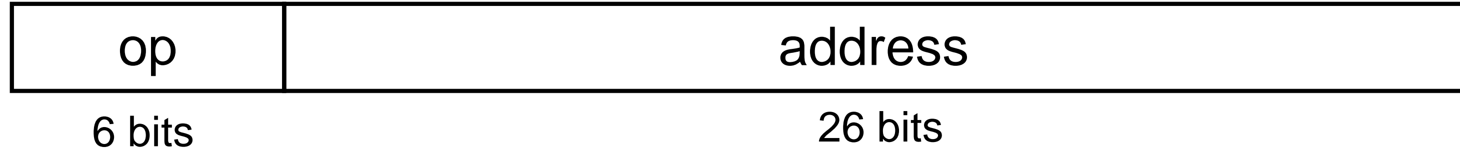
- Example

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

MK MORGAN KAUFMANN

3) MIPS J-format Instructions

- Jump (`j` and `jal`) targets could be anywhere in text segment



- Direct jump addressing
  - Target address = address × 4
- j 10000
  - op = 2, go to address = 10000 × 4

Target Addressing Example

- Loop code from earlier example
  - 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: …
```

Address						
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Jump Addressing

Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

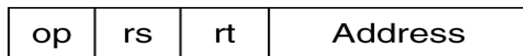
MK MORGAN KAUFMANN

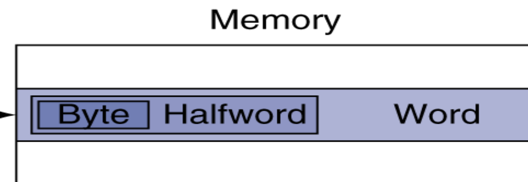
Addressing Mode Summary

1. Immediate addressing

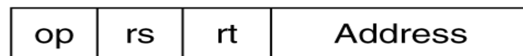
2. Register addressing

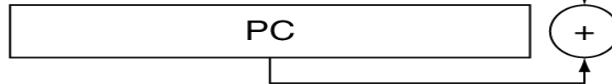
3. Base addressing

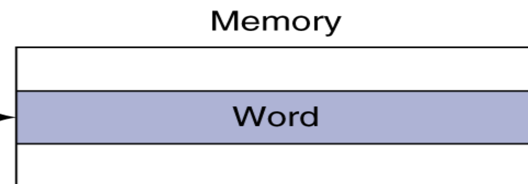




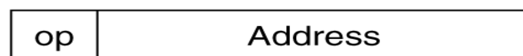
4. PC-relative addressing

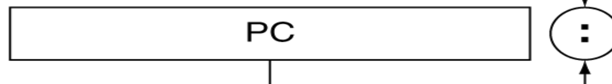


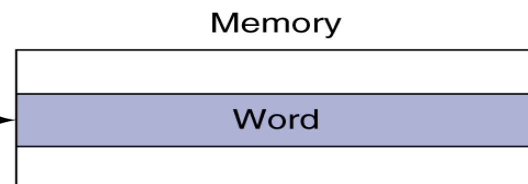




5. Pseudodirect addressing







Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- 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): assembler temporary

MK MORGAN KAUFMANN

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
- Instruction categories:
  - Arithmetic/logical (equations)
  - Data transfer (memory data structures)
  - Conditional branch (if statement and while loops)
  - Unconditinoal jump (procedure/fn call and return)

Problems to solve

- 2.14 to 2.17, 2.21, 2.25, 2.29 to 2.41, 2.47

MORGAN KAUFMANN