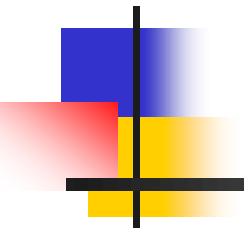


Chapter2 – Part II

Instructions: Language of the Computer

臺大電機系　吳安宇教授

2025/03/03 v1

Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer (Week2)
- 2.6 Logical Operations (Week3)
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

Logical Operations

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Example:

  0000 0000 0000 0000 0000 0000 0000 $1001_2 = 9_{10}$

  Shift left by 4 → 0000 0000 0000 0000 0000 0000 1001 $0000_2 = 144_{10}$

- sll (shift left logical) (c.f. Shift right logical, srl)

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

op	rs	rt	rd	shamt	funct
0	unused	16	10	4	0
000000	00000	10000	01010	00100	000000

Logical Operations

- Example: **($t2 as mask)**

  assume $t2 = 0000 0000 0000 0000 **1**000 **1111** 0000 0000_2

  $t1 = 0000 0000 0000 0000 0011 1100 0000 0000_2

(1) **AND** operation (set to 0 OR mask of 1)

and $t0 , $t1, $t2 # $t0 = $t1 & $t2

0	9	10	8	0	36

→ $t0 = 0000 0000 0000 0000 0000 1100 0000 0000_2

(2) **OR** operation (set to 1)

or $t0 , $t1, $t2 # $t0 = $t1 | $t2

0	9	10	8	0	37

→ $t0 = 0000 0000 0000 0000 1011 1111 0000 0000_2

Logical Operations

- Example:

  assume $t2 = 0000 0000 0000 0000 0000 1101 0000 0000_2

  $t1 = 0000 0000 0000 0000 0000 0000 0000_2

(3) **NOR** operation (R-type)

Example:

nor $t0 , $t1, $t2 # $t0 = ~ ($t1 | $t2)

➔ $t0 = 1111 1111 1111 1111 1100 0011 1111 1111_2

0	9	10	8	0	39

(4) Pseudo Inst.: Perform **NOT operation using NOR Instruction**

A **NOR** 0 = **NOT** (A **OR** 0) = **NOT (A)**

nor $t0 , $t1, $0 # $t0 = ~ ($t1 | 0) = ~$t1

MIPS Instructions

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory

Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Representing Instructions in the Computer
- 2.5 Logical Operations
- 2.6 Instructions for Making Decisions
- 2.7 Supporting Procedures in Computer Hardware
- 2.8 Communicating with People
- 2.9 MIPS Addressing for 32-bit Immediates and Addresses
- 2.10 Translating and Starting a Program

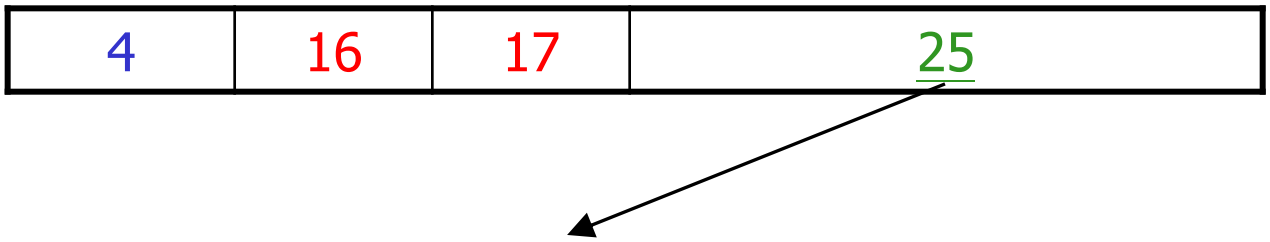
Instructions for Making Decisions

- (a) beq reg1 reg2 L1 ***beq = "branch equal"***

  # IF (reg1 value) == (reg2 value)
  then GOTO statement labled L1

  - Example:

    beq $s1, $s2, L1 # if ($s1==$s2) goto location (PC+4)+100



Branch by 25 **Instructions**
→ New location of Instruction in memory:
Next Program Counter (PC+4) + 25 x 4 (=100)

Instructions for Making Decisions

- (b) bne reg1 reg2 L1 ***bne = "branch not equal"***

  # IF (reg1 value) != (reg2 value)
  then GOTO statement labled L1

  - Example:
    bne $s1, $s2, 100 # if ($16 != $17) goto location (PC+4)+100

5	16	17	25

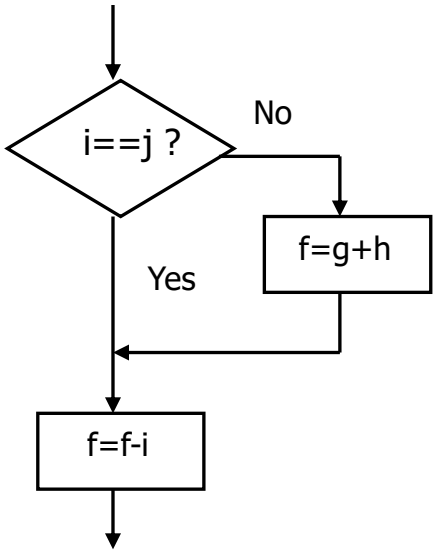
- The two instructions are traditionally called **conditional branches.**

Instructions for Making Decisions

- Example:

  In C language :

```
        If ( i == j ) goto L1
        f = g + h;
L1:     f = f – i;
```



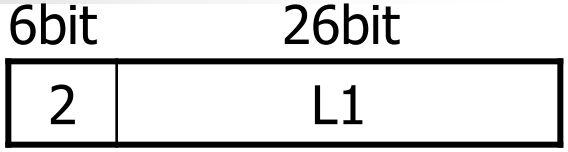
In MIPS: assume i=$s1, j =$s2, f=$s3, g=$s4, h=$s5

PC (Program Counter)

```
100   beq $s1, $s2, L1     # if (i==j) goto L1
104   add $s3, $s4, $s5    # f=g+h (skipped if i==j)
L1    sub $s3, $s3, $s1    # f=f-i (always executed)
```

L1 = address of the subtract instruction ***(L1 =108)***

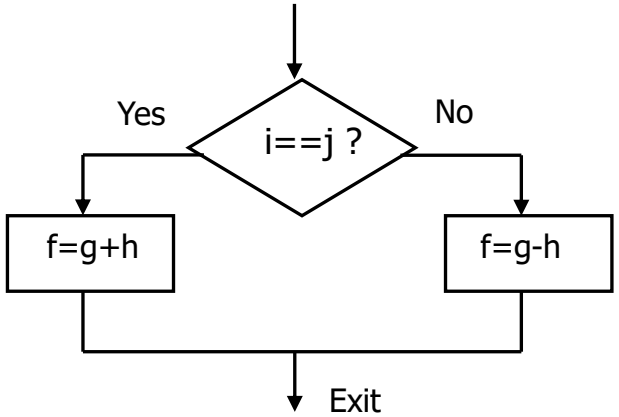
Instructions for Making Decisions



- (c) jump (unconditional branch) : j L1
  - Example:

In C language :

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



In MIPS: assume i=$3, j =$s4, f=$s0, g=$s1, h=$s2

```
100     bne $s3, $s4, Else      #  if (i!=j) goto Else (+2)
104     add $s0, $s1, $s2       #  f=g+h
108     J  Exit                 #  jump Exit
Else:   sub $s0, $s1, $s2       #  f=g-h,   (Else=112)
Exit:   ..................      #  (Exit = 116)
        ..................
```

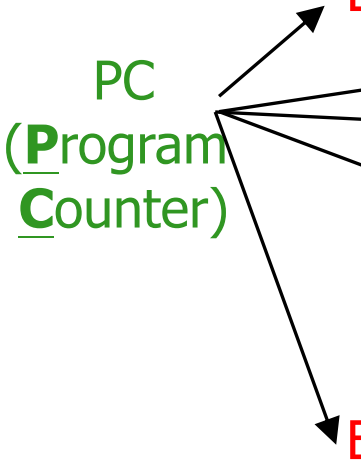
Instructions for Making Decisions

- Loops: **In C language :**

```
while ( A[i] == k )
        i+=1;     # counting continuous k value in A[]
```

In MIPS: assume i=$s3, k=$s5,

and the base register of the array **A** is in $s6




```
Loop:   sll   $t1, $s3, 2     # temp $t1 = 4* i
        add   $t1, $t1, $s6   # $t1 = address of A[i]
        lw    $t0, 0($t1)     # temp $t0 = A[i]
        bne   $t0, $s5, Exit  # if (A[i] != k),  go to Exit
        add    $s3, $s3, 1    # i = i+1
        j Loop                # go to Loop
Exit:   ……..                  # Loop=1000, Exit =??
```

1024 or +5 instructions

Instructions for Making Decisions (machine code)

- Addressing in branches and jumps
  - J-type : consists of 6 bits for the operation field and the rest of the bits for the address.

  - Example: j 10000 # go to the location 10000

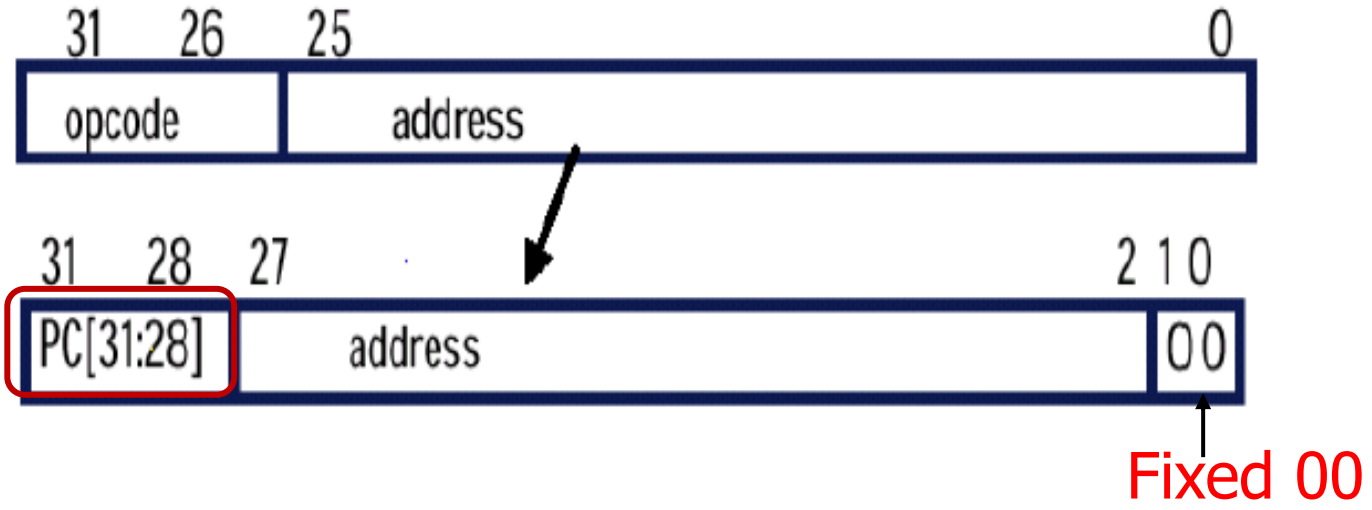
  - Unlike the jump instruction, the conditional branch instruction must specify 2 operands in addition to the branch address.

bne $s0, $s1, Exit # if ($s0!=$s1) go to Exit (+/- 100)

Program Counter (PC) = (PC + 4, next inst.) + branch address x4

"Jump" operation

- "Jump" operation: (opcode = 000010)
  - Replace a portion of the PC(bit 27-0) with the lower 26 bits of the instruction shifted left by 2 bits.
  - The shift operation is accomplished by simple concatenating "00" to the jump offest.



Showing Branch Offset in Machine Language

The *while* loop on page 107–108 was compiled into this MIPS assembler code:

```
80000  Loop: sll    $t1,$s3,2   # Temp reg $t1 = 4 * i
80004        add  $t1,$t1,$s6   # $t1 = address of save[i]
80008        lw   $t0,0($t1)    # Temp reg $t0 = save[i]
80012        bne  $t0,$s5,Exit  # go to Exit if save[i] ≠ k
80016        addi $s3,$s3,1     # i = i + 1
80020        j    Loop          # go to Loop
80024  Exit:
```

Location address vs Machine code!

The assembled instructions and their addresses are:

Starting address of the program

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	...					

(24-16)/4

Loop=80000 (/4)

MIPS machine language

MIPS machine language

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
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Remark

- In Assembly program, you only need to write Labels (in text format) for BEQ, J, BNE. Assembler (PC-SPIM or Qt-SPIM) will convert them into binary address numbers implicitly.

- The starting address of your program is decided when this program is loaded into Main Memory. So it need two-pass procedure to decide the addresses of the labels and offsets.

Branching Far Away (Limitation by op field)

Given a branch on register `$s0` being equal to register `$s1`,

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

Limited range due to 16-bit field

replace it by a pair of instructions that offers a much greater branching distance.

These instructions replace the short-address conditional branch:

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

Far range by using 26-bit field

Instructions for Making Decisions: Slt

- Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

- slt reg3, reg1, reg2 **slt : set ON less than *(R-type)***
  - ➔ Compare two registers and set
    if (reg1 < reg2) reg3=1; Otherwise, reg3=0;
  - Example:
    slt $s1, $s2, $s3 #if ($s2<$s3) set $s1=1, else $s1=0.

0	18	19	17	0	42

- "Set on less than" immediate (slti) – *I-type (page p.A-666)*
  - Example: slti $t0, $s2, 10
    # if ($s2 < 10), $t0 =1; Otherwise, $t0 =0.

Set ON Less Than (slt)

- Signed integer (+/-) : Normal numbers (usually)
- Unsigned integer (+) : Memory addressing

→ **slt (slti) : set ON less than, signed integer (immediate)**
sltu (sltiu) : set ON less than, unsigned integer (immediate)

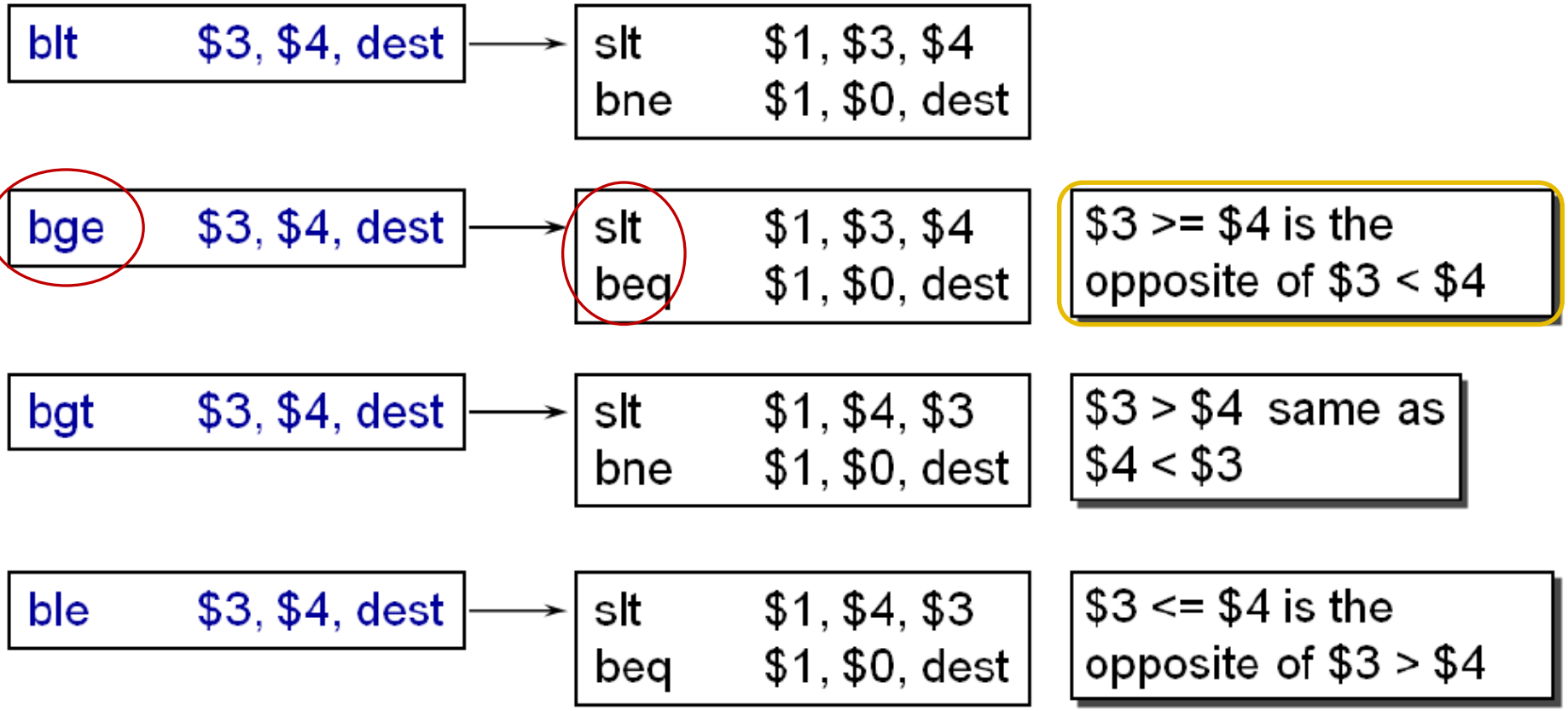
- Example

$s1 = 1111 1111 …… 1111_2 (**= -1 for signed or $(2^{32}-1)$ for unsigned**)
$s2 = 0000 0000 …… 0001_2 (**= +1 for signed and for unsigned**)

slt $t0, $s1, $s2 # signed comparison → $t0 =1
sltu $t0, $s1, $s2 # unsigned comparison → $t0=0

Inst	Example	Meaning (unsigned comparison)
sltu	sltu $s1,$s2,$s3	If($s2<$s3),$s1=1; else $s1=0
sltiu	sltiu $s1,$s2,100	If($s2<100),$s1=1; else $s1=0

Build Pseudo-Instructions



Summary of MIPS Instructions

MIPS operands

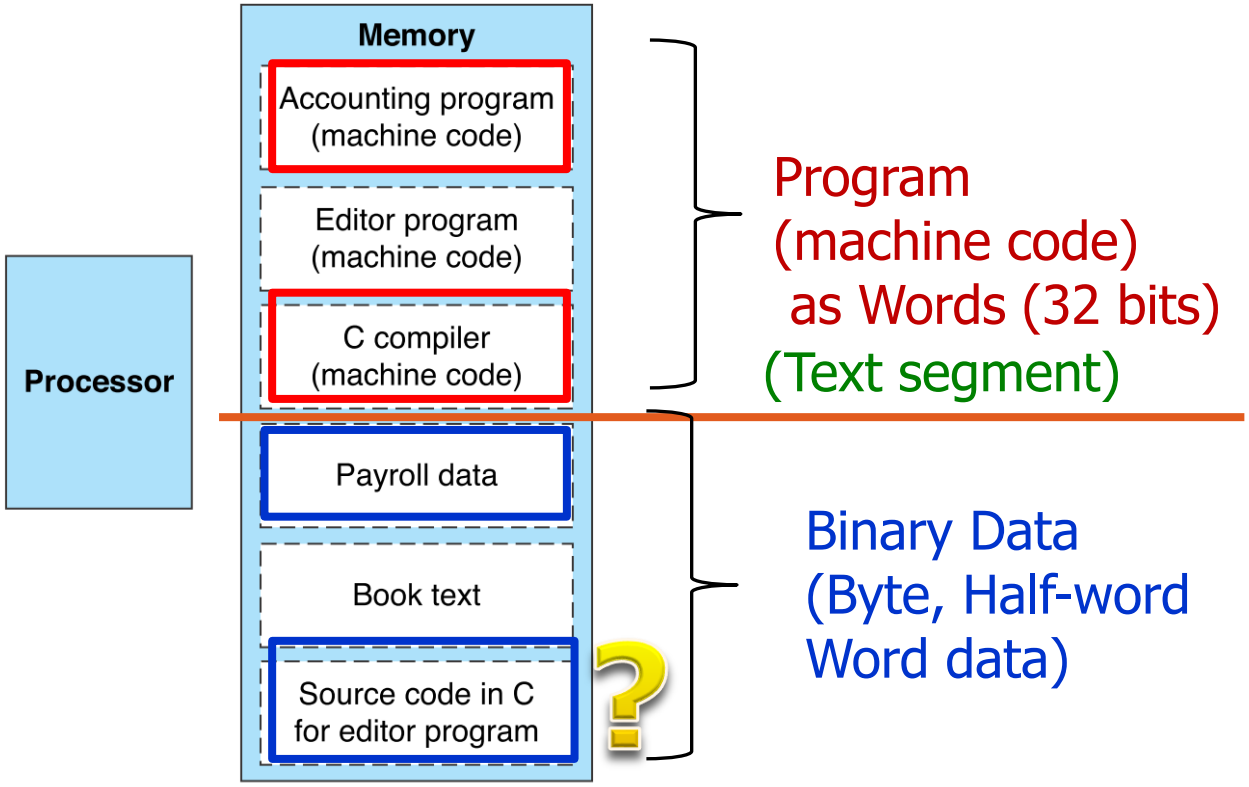
Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Word from memory to register
	store word	`sw $s1,20($s2)`	`Memory[$s2 + 20] = $s1`	Word from register to memory
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1 = ~ ($s2 | $s3)`	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1 = $s2 & 20`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1 = $s2 | 20`	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; for `beq`, `bne`
	set less than immediate	`slti $s1,$s2,20`	if (`$s2 < 20`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
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

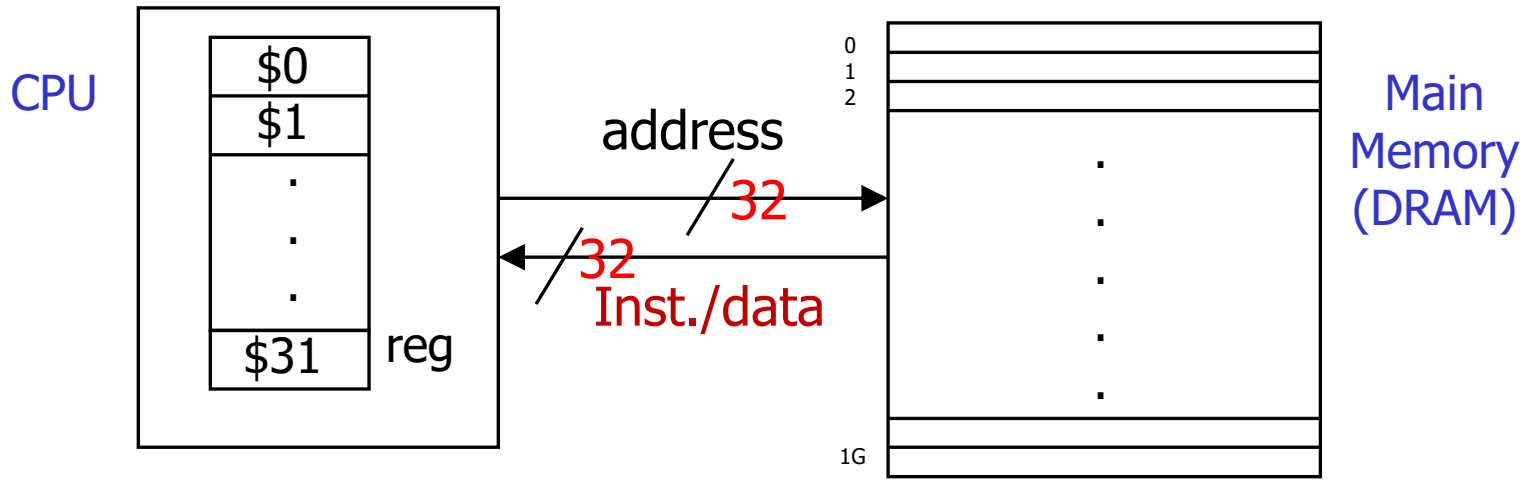
Review of "Stored-program" Concept

- Instructions are represented as (32-bit) numbers!!
- Programs can be stored in memory to be read or written just like numbers.
- Most important concept in computer history (*von Neumann architecture*)



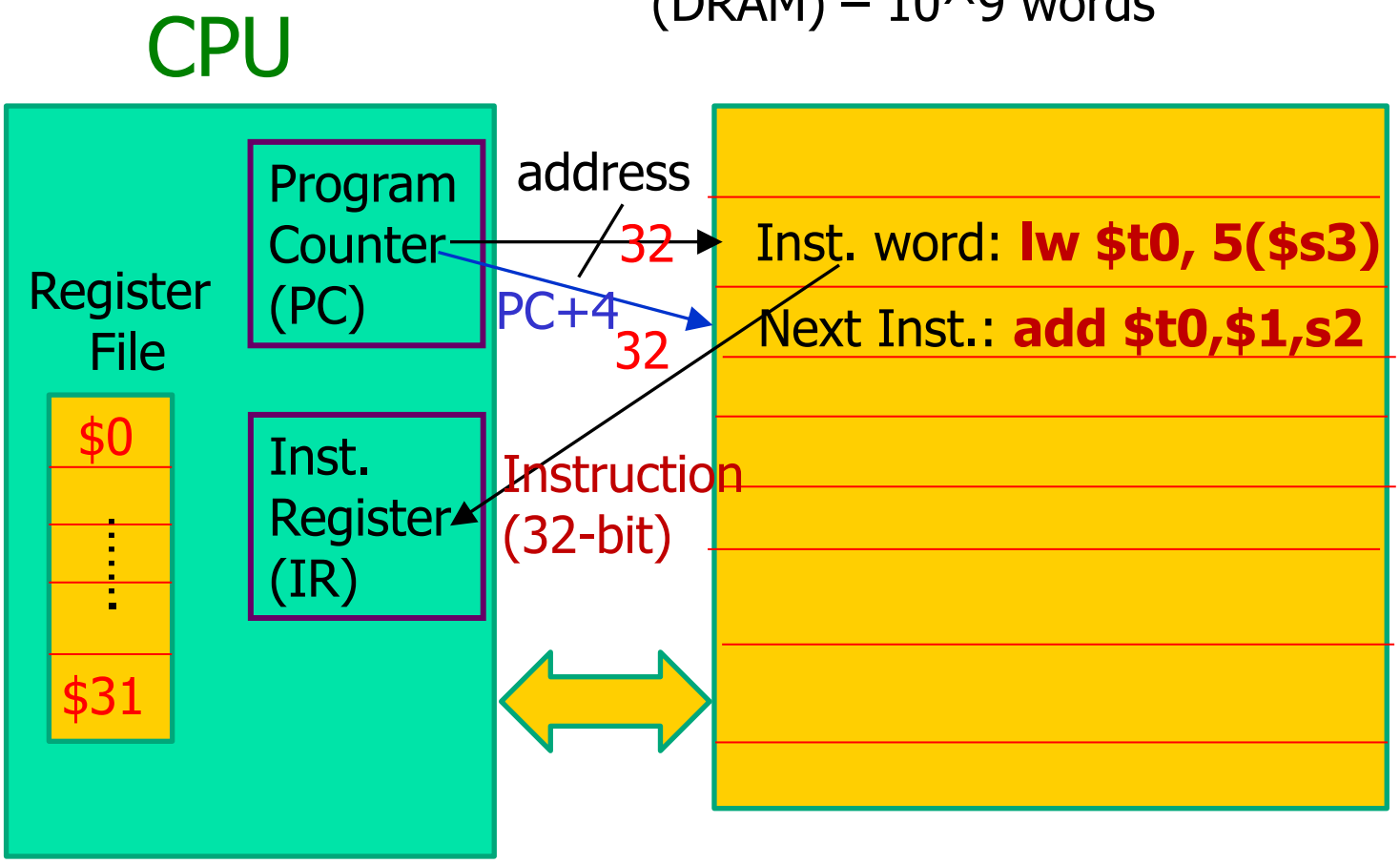
Registers

- MIPS has 32 registers, using the notation $0, $1, ……, $31 to represent them.

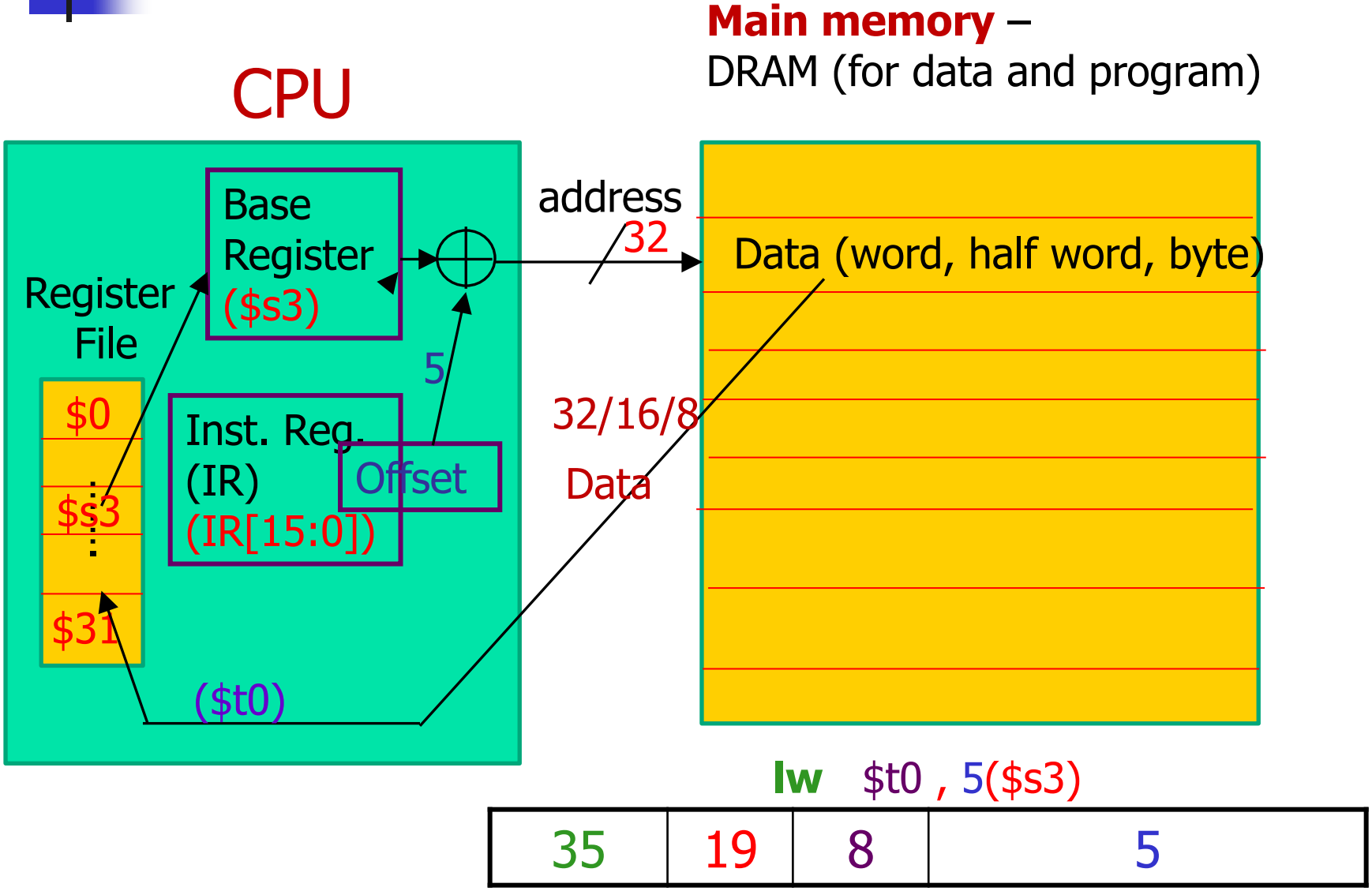


Addressing mode of "Inst. Fetch (IF)"

Main memory – (for data and program) (DRAM) – 10^9 words



Addressing mode of "Data fetch"



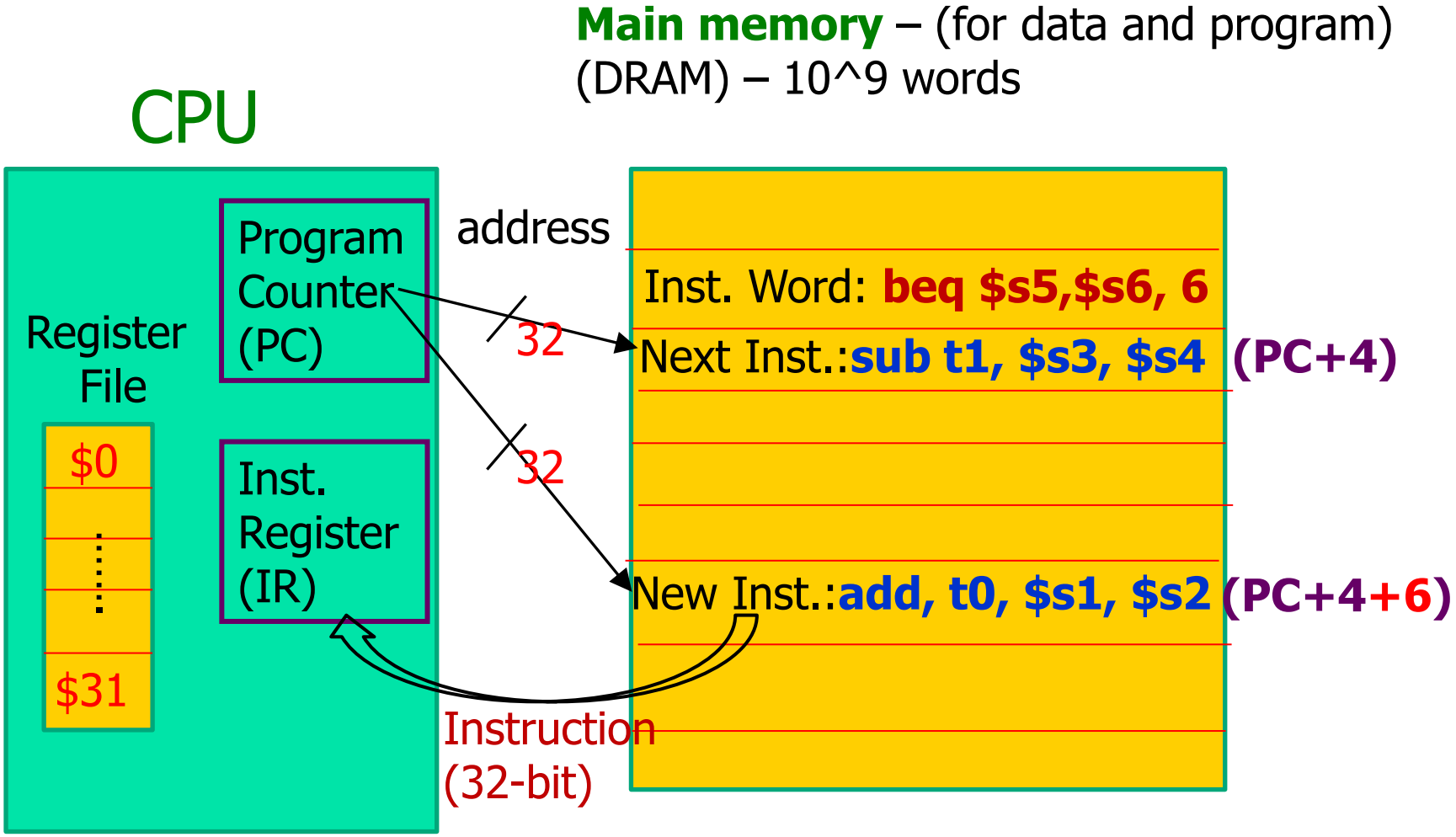
Branch Instructions (recall)

- (ex) beq $t1, $t2, offset

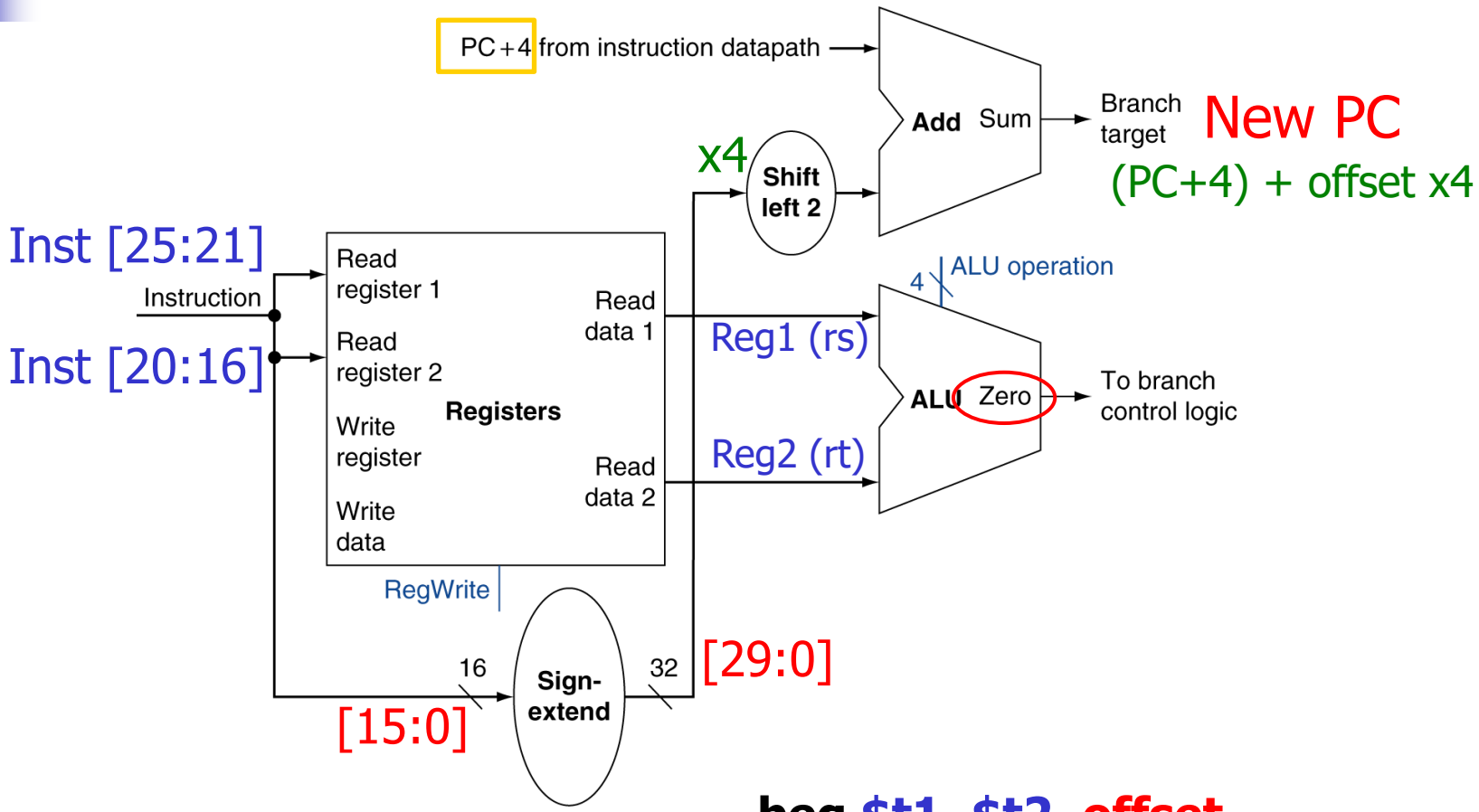
```
# if ($t1==$t2)
        goto  (PC+4 + offset)       // PC ← PC+4+offset
  else
        execute next instruction  //  PC ← PC+4
```

- Note:
  (1) The offset field is shifted left 2 bits so that it's a "word offset".
  (2) Branch is taken (taken branch): when the condition is true, the branch target address becomes the new PC.
  (3) Branch isn't taken (untaken branch): the incremented PC (PC+4) replaces the current PC, just as for normal instruction.
- Operations:
  (1) Compute the branch target address.
  (2) Compare the contents of the two registers.

Addressing mode of Branch (Beq)



Datapath for "beq" Instructions

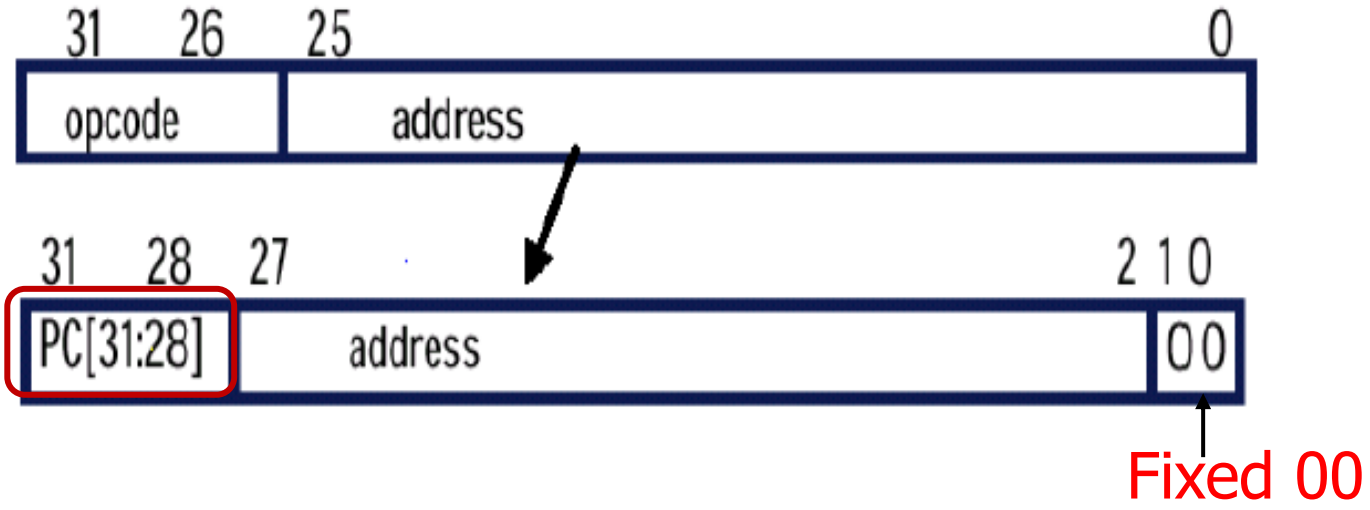


- **beq $t1, $t2, offset**

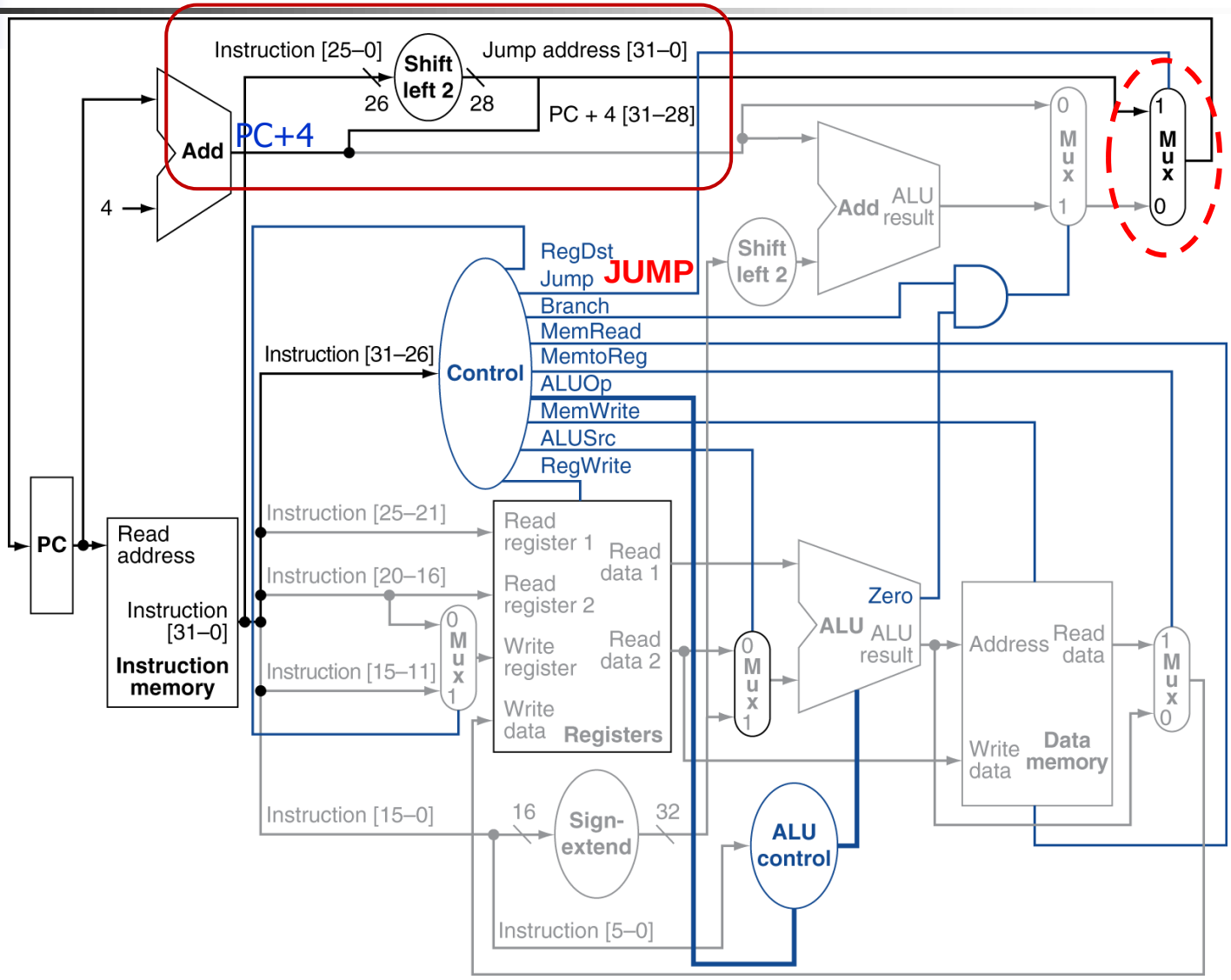
I:	op	rs	rt	address / immediate
		[25:21]	[20:16]	[15:0]

Datapath for "Jump"

- "Jump" operation: (opcode = 000010)
  - Replace a portion of the PC(bit 27-0) with the lower 26 bits of the instruction shifted left by 2 bits.
  - The shift operation is accomplished by simple concatenating "00" to the jump offest.



Implementing “Jumps”



Outline

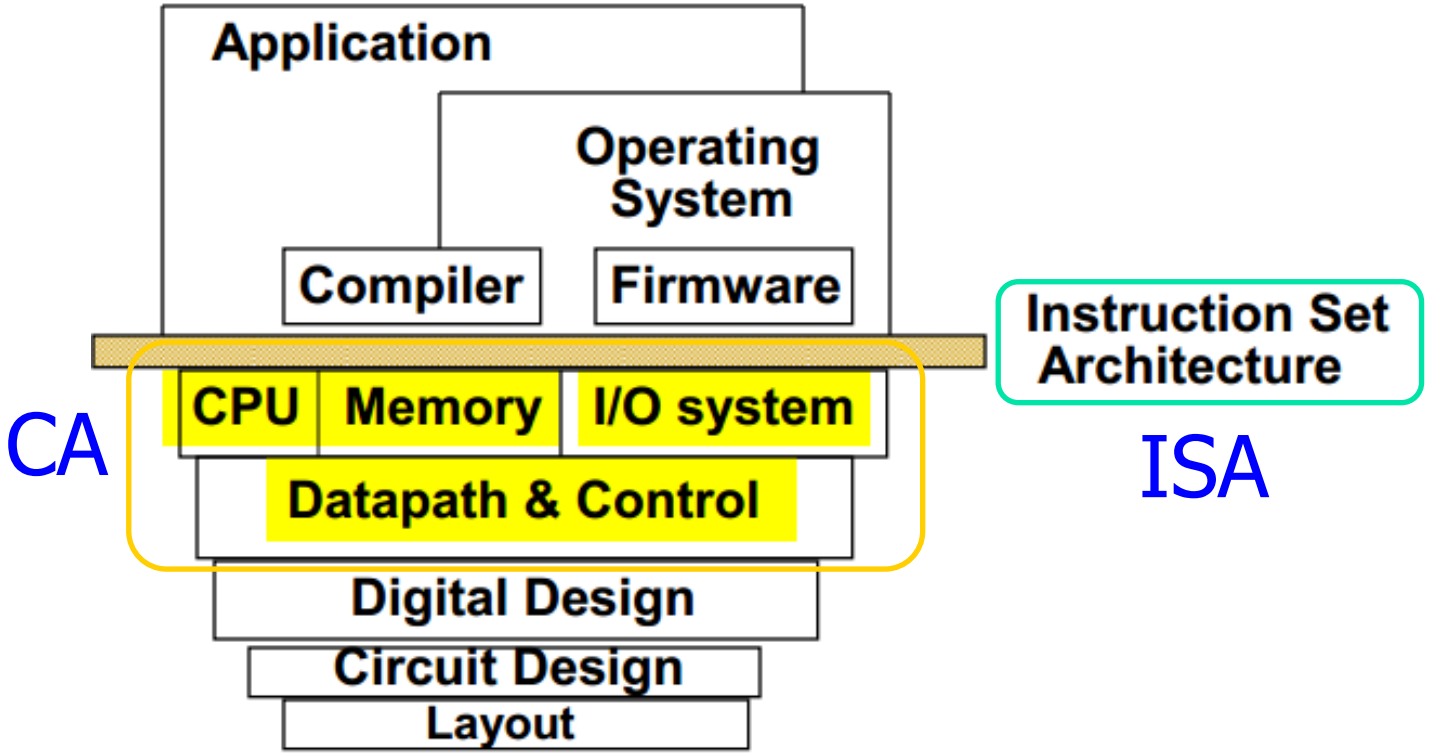
- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

Review: MIPS Instruction Category

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

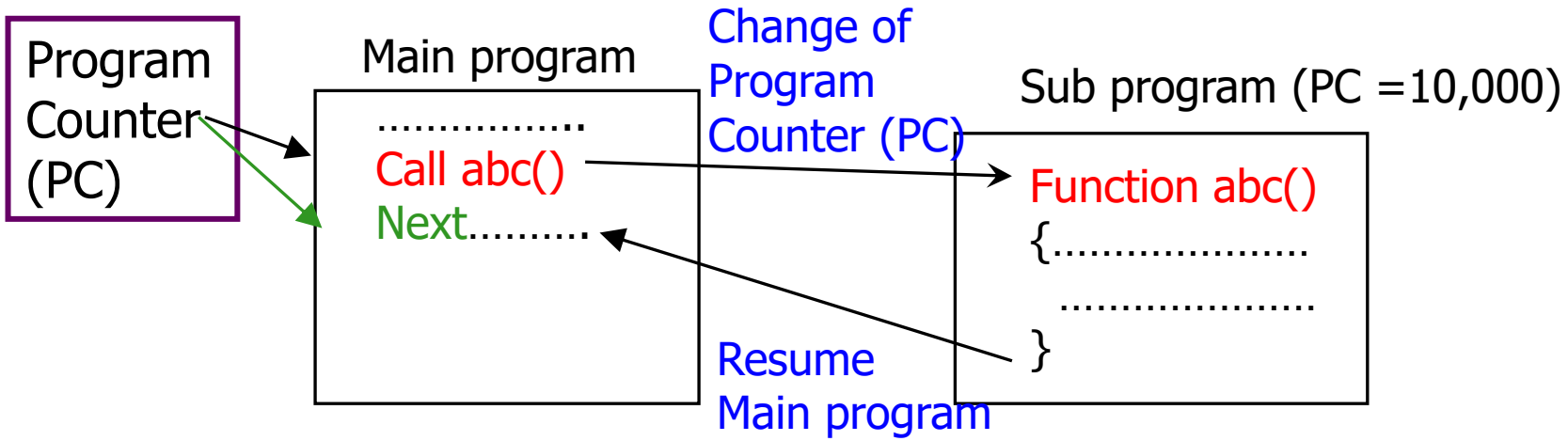
Role of ISA in CPU Design



http://gitqwerty777.github.io/computer-architecture1/

Procedure/Subroutines

- **Procedure/subroutines**: Used in structured programming to make programs easier to understand and allow codes to be "reused" once or for several times.
  - We need instructions:
    - ***Jump to procedure***: *Change PC: jal; $ra ← PC +4;*
    - ***Return to the instruction after the calling point:*** *jr $ra*



Supporting Procedures in Computer Hardware

- In the execution of a procedure, the program must follow these 6 steps:
  1. **Place parameters in a place** where the procedure can access them.
  2. **Transfer control** to the procedure.
  3. **Acquire the storage resources** needed for the procedure.
  4. **Perform the desired task.**
  5. **Place the result value in a place** where the calling program can access it.
  6. **Return control** to point of origin.
- Note: A procedure can be called from ***several points*** in a program (e.g., *y=sin(x)*)

Some related Terms

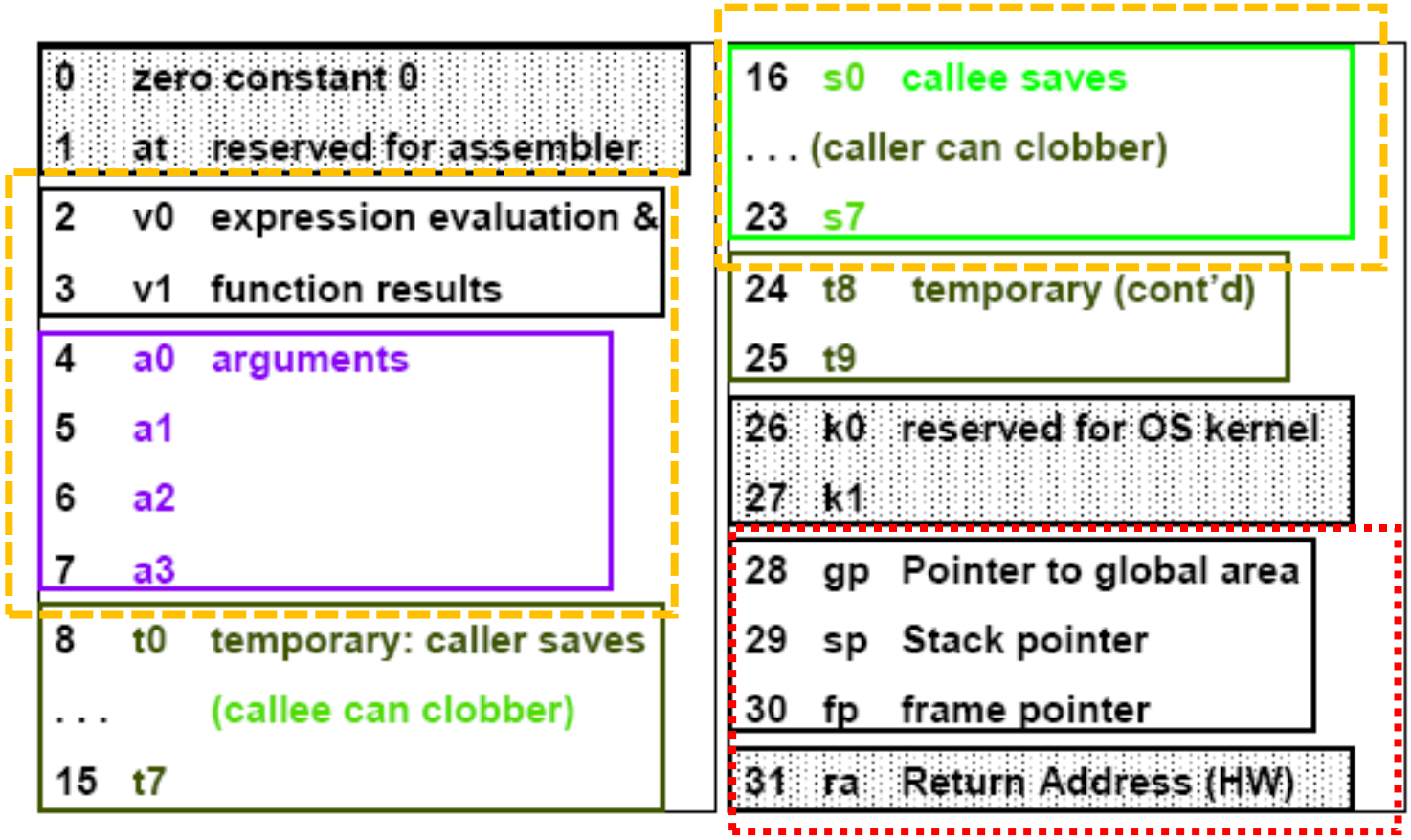
- Caller: The program that instigates a procedure and provides the necessary parameter values.
- Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

- Stack (in MEM (DRAM)): A data structure for spilling registers organized as a last-in-first-out queue.
- Stack pointer ($sp) : A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.
- Push: place data into the stack
- Pop: remove data from the stack

Registers in Procedures

- MIPS software follows the following convention in allocating its 32 registers for procedure calling:
  - $a0 - $a3 ($4 ~ $7): Four **argument registers** in which to pass parameters.
  - $v0 - $v1 ($2 ~ $3) : Two **value registers** in which to return values.
  - $ra ($31): One **return address register** to return to the point of origin.

MIPS Register Convention



Summary of MIPS Registers

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

http://gitqwerty777.github.io/computer-architecture1/

Supporting Procedures: **jal & jr**

- ***jal ProcedureAddress*** ***jal : jump and link***

  ➔ Jump to an address and simultaneously saves the address of the following instruction (PC+4) in Reg $ra.

  (1) Return address (link) is stored in Reg $ra ($31)

  (2) Returning to main program is performed as below.

  jr $ra # jump to the address stored in $ra

  jr (Jump Register)

- Program counter (PC): The register containing the address of the instruction in the program being executed.
- Return address: A link to the calling site that allows a procedure to return to the proper address **($ra ← PC+4);**

Example of Procedures

- Example: f = (g + h) – (i + j) – from old example
  - Assignment: (by compiler)

    f = $s0 (returned value)

    g = $s1, h = $s2, i = $s3, j = $s4 (parameters)
  - Compiler → assembly language

```
add $t0, $s1, $s2    # t0 = g+h
add $t1, $s3 $s4     # t1= i+j
sub $s0, $t0, $t1    # f = (g+h)-(i+j)
```

Example of Procedures (Callee save)

In C program: int leaf_example (int g, int h, int i, int j)

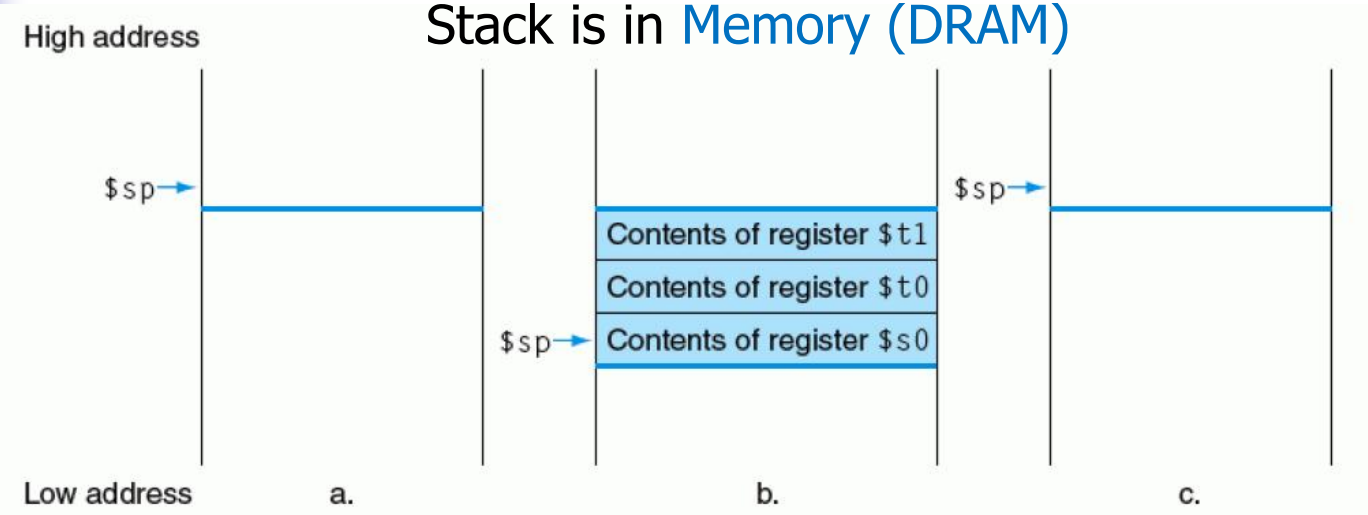
```
{
        int f;
        f = ( g + h ) – ( i + j);
        return f;
}
```

- In MIPS: assume g=$a0, h=$a1, i=$a2, j=$a3, **f=$s0.**
- **Need to save 3 register: $s0, $t0, $t1 (as they will be overwritten).** We "**push**" the old values onto the stack by creating space for three words on the stack and then store them.

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw   $t1,  8($sp)     # save $t1 for use afterwards
sw   $t0,  4($sp)     # save $t0 for use afterwards
sw   $s0,  0($sp)     # save $s0 for use afterwards
```

Example of Procedures (cont)

Stack is in Memory (DRAM)



The next three instructions correspond to the **body** of the procedure.

```
add $t0, $a0, $a1        # $t0 = g + h
add $t1, $a2, $a3        # $t1 = i + j
sub $s0, $t0, $t1        # f = $t0 - $t1 = (g + h) - (i + j)
```

To return the value of f, we copy it into a return value register.

```
add $v0, $s0, $zero      # return f ( $v0 = $s0 + 0 )
                         #  Move $s0  to $v0!!
```

Example of Procedures (cont)

- Before retuning, we restore the three old values of the registers we saved by "**popping**" them from the stack.

```
lw $s0, 0($sp)          # restore $s0 for caller
lw $t0, 4($sp)          # restore $t0 for caller
lw $t1, 8($sp)          # restore $t1 for caller
addi  $sp, $sp, 12      # adjust stack to delete 3 items
```

- The procedure ends with a jump register using the return address.

```
jr $ra                  #  jump back to calling routine
```

Supporting Procedures in Computer Hardware

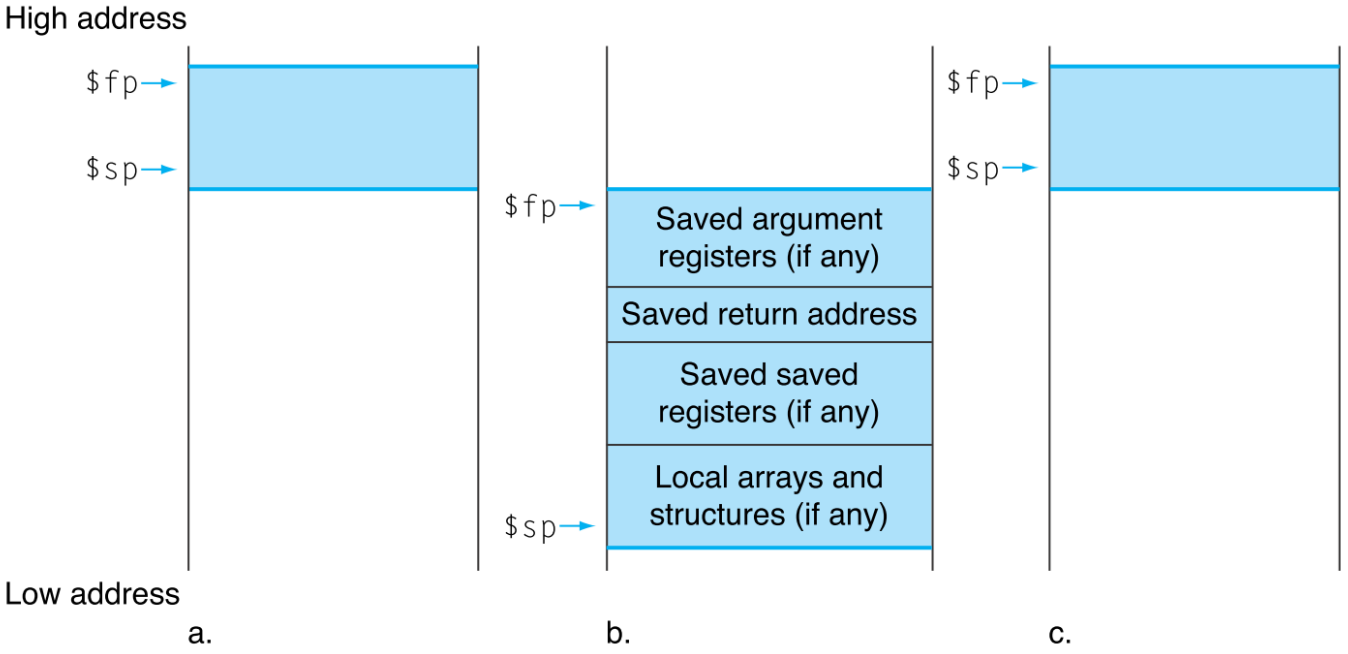
- MIPS software separates 18 of the registers into two groups:
  - $t0 - $t9: 10 temporary registers that are **NOT** preserved by the *callee (called procedure)* on a procedure call.
  - $s0 - $s7: 8 saved registers **that MUST be preserved by the callee** on a procedure call (if used, the callee saves and restores them)
  - Can help to reduce "register spiling"

- What is and what is not preserved across a procedure call:

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

Allocating space for new data on the Stack

- Procedure frame : also called activation record. The segment of the stack containing a *procedure's saved registers and local (temp) variables.*
- *Stack pointer ($sp)* : May change during the procedure.
- *Frame pointer ($fp)* : Pointer points to the 1st word of the frame of a procedure (main program).



Allocating space for new data on the Heap

- C programmers need space in memory for static variables and for dynamic data structure.
  - Stack starts in the high end of memory and grows down.
  - The first part of low end of memory is reserved, followed by the "machine codes" – text segment.
  - Above the code is the "static data segment," which is the place for constants and other static variables (int A[100]).
  - Data structure like "linked list (via *malloc() and free()*)" tends to grow and shrink during execution – the segment is called "heap."
  - Forget to free memory (*free()*) leads to "memory leak"

Allocating space for new data on the Heap

- text segment: The segment of a Unix object file that contains the machine language code for routines in the source file.
- global pointer ($gp = 1000_8000 at initialization): The register that is reserved to access static data ranging from 1000_0000_hex to 1000_ffff_hex.

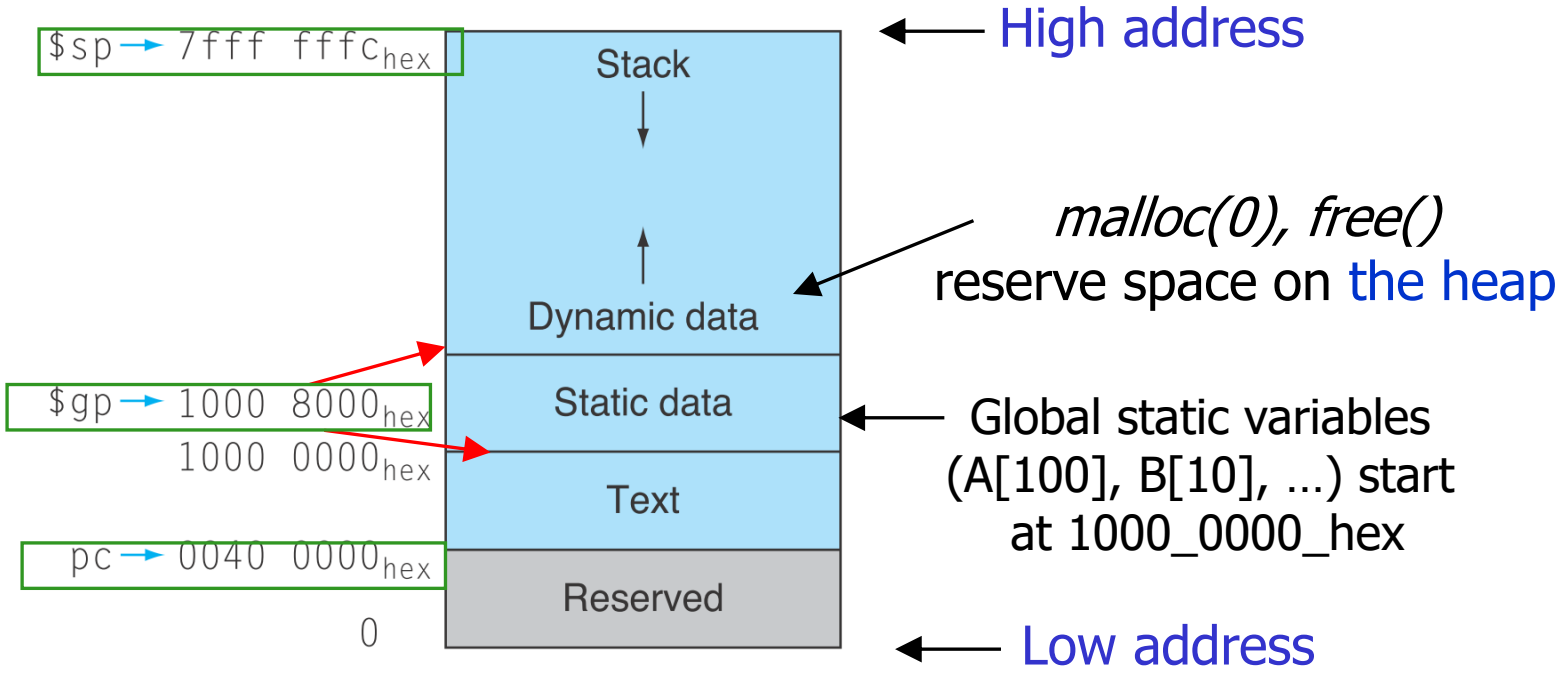


Fig. 2.13 MIPS memory allocation for program and data
(memory leak: forget to free space on the heap)

Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People (skip)
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

Communicating with People

- American Standard Code for Information Interchange (ASCII)

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

- Move bytes:
  - load byte: lb $t0 , 0 ($sp) # read byte from source
  - store byte: sb $t0 , 0 ($gp) # write byte to destination
  - load byte unsigned **lbu $t0 , 0 ($sp)** # read ASCII from source

- Move halfwords:
  - load halfword: lh $t0 , 0 ($sp) # read halfword (16 bits) from source
  - store halfword: sh $t0 , 0 ($gp) # write halfword (16 bits) to destination

Example of "Copy String"

- **End of a string**: C program terminates a string with a byte of value 0 (named *null* in ASCII)
- E.g., **"Cal"** is represented by 4 bytes, shown as decimal numbers: **67, 97, 108, 0 (Null, '\0')**
- *Example:*

Compiling a String Copy Procedure, Showing How to Use C Strings

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
     i += 1;
}
```

What is the MIPS assembly code?

Copy String Example

- Assume base of addresses of Array X[i] and Y[i] are in *$a0* and *$a1*, respectively. ***i*** is in *$s0.*

```
strcpy:
    addi   $sp,$sp,-4    # adjust stack for 1 more item
    sw     $s0, 0($sp)   # save $s0
```

(push $s0)

To initialize `i` to 0, the next instruction sets `$s0` to 0 by adding 0 to 0 and placing that sum in `$s0`:

```
    add    $s0,$zero,$zero # i = 0 + 0
```

This is the beginning of the loop. The address of `y[i]` is first formed by adding `i` to `y[]`:

```
L1: add    $t1,$s0,$a1   # address of y[i] in $t1
```

(y[0] at 1st time)

Note that we don't have to multiply `i` by 4 since `y` is an array of *bytes* and not of words, as in prior examples.

Copy String Example

To load the character in y[i], we use load byte unsigned, which puts the character into $t2:

```
lbu    $t2, 0($t1)   # $t2 = y[i]
```

A similar address calculation puts the address of x[i] in $t3, and then the character in $t2 is stored at that address.

```
add    $t3,$s0,$a0   # address of x[i] in $t3
sb     $t2, 0($t3)   # x[i] = y[i]
```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
beq    $t2,$zero,L2 # if y[i] == 0, go to L2
```

If not, we increment i and loop back:

```
addi   $s0, $s0,1    # i = i + 1
j      L1            # go to L1
```

Copy String Example

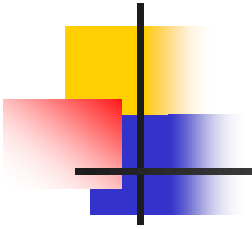
- *End (pop $s0) and Return control to Caller*

If we don't loop back, it was the last character of the string; we restore `$s0` and the stack pointer, and then return.

```
L2: lw    $s0, 0($sp) # y[i] == 0: end of string. Re-
store old $s0

    addi  $sp,$sp,4   # pop 1 word off stack
    jr    $ra         # return
```

(pop $s0)



Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing Modes
- 2.11 Translating and Starting a Program

32-bit immediate operands

- **load upper immediate (lui) – for 32-bit constant**

>> lui $t0, 255 # $t0 = 255_{10} << 16 bits

001111	00000	01000	0000000011111111 (255)

The contents of the register $t0 becomes

0000 0000 1111 1111	0000 0000 0000 0000

>> ori $t0, $t0, 1010101010101111

The contents of the register $t0 ($8) becomes

0000 0000 1111 1111	1010 1010 1010 1111

32-bit immediate operands

- Example:

  (Question)

  What is the MIPS assembly code to load this 32-bit constant into register $s0?

  (Answer)

  lui $s0, 61 # 61_{10} = 0000 0000 0011 1101_2

  → $s0 = 0000 0000 0011 1101 0000 0000 0000 0000

  ori $s0, $s0, 2304 # 2304_{10} = 0000 1001 $0000\ 0000_2$

  → $s0 = 0000 0000 0011 1101 0000 1001 0000 0000

  → $s0 = 61 << 16 (x 2^16) + 2304 (answer)

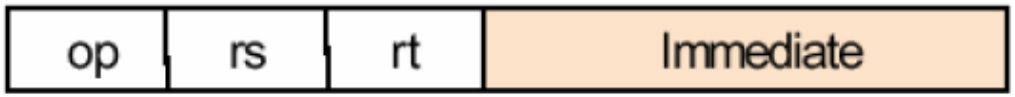
MIPS Addressing Modes

How to get the operands in instructions?

- Immediate addressing, where the operand is a constant within the instruction itself.
- Register addressing, where the operand is a register.
- Base or displacement addressing, where the operand is at the memory location whose **address is the sum of a register and a constant in the instruction.**
- PC-relative addressing, where the address is **the sum of the PC and a constant in the instruction.**
- Pseudo addressing, where the jump address is the 26 bits of the instruction **concatenated with the upper bits of the PC.**

MIPS addressing mode (I-type, Immediate)

- (1) Immediate addressing (I-type)
  -- get data immediately from Instruction

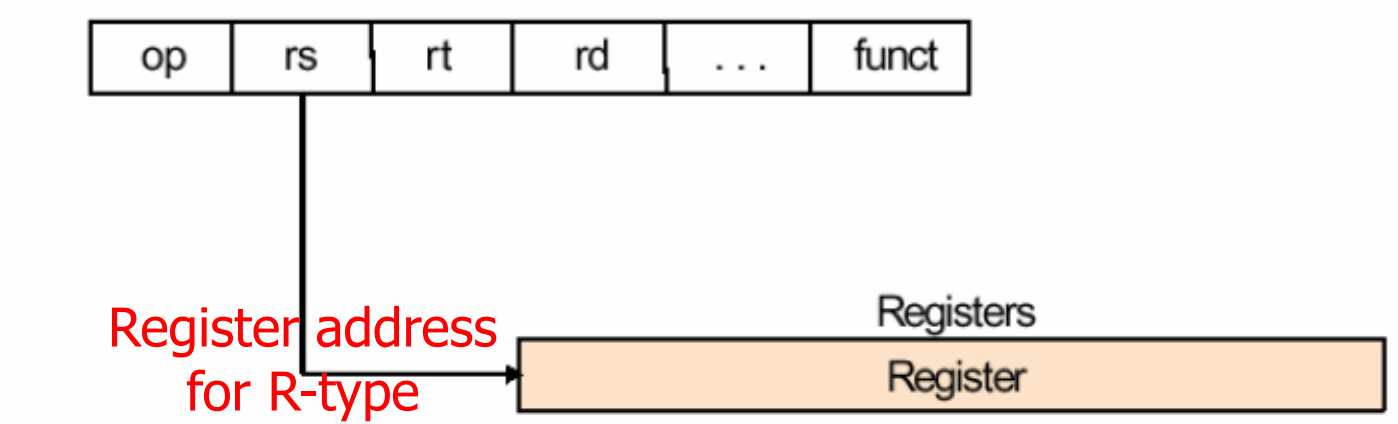


- Example:

```
addi R1, R0, 10
```

MIPS addressing mode (R-type)

- (2) Register addressing (R-type)
  -- get data from register file



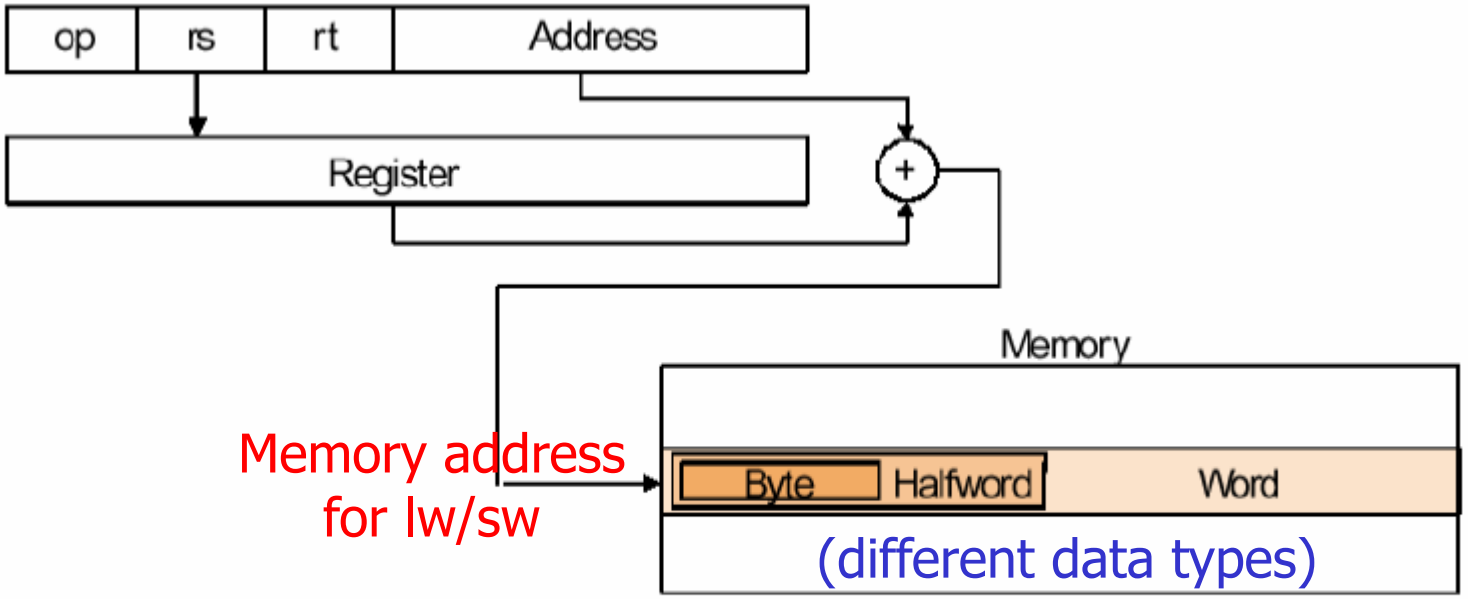
- Example:

```
add R2, R0, R1
```

0	0	1	2	0	32

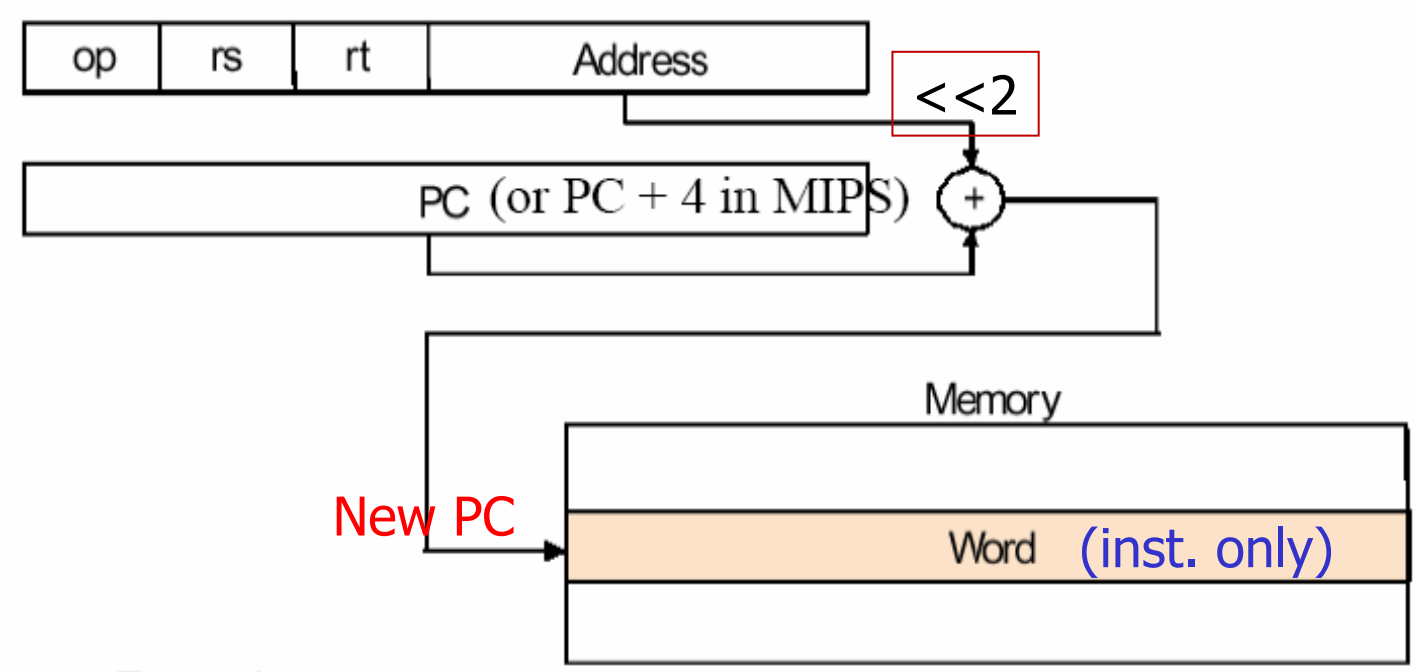
MIPS addressing mode (I-type)

- (3) Base or displacement addressing (I-type)
  – get data from memory (lw), or store data to mem (sw)



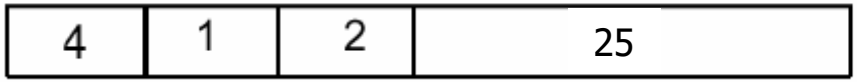
MIPS addressing mode (PC-relative)

- (4) PC-relative addressing (I-type) -- to get next Instruction from Compare-and-Branch (beq, bne)



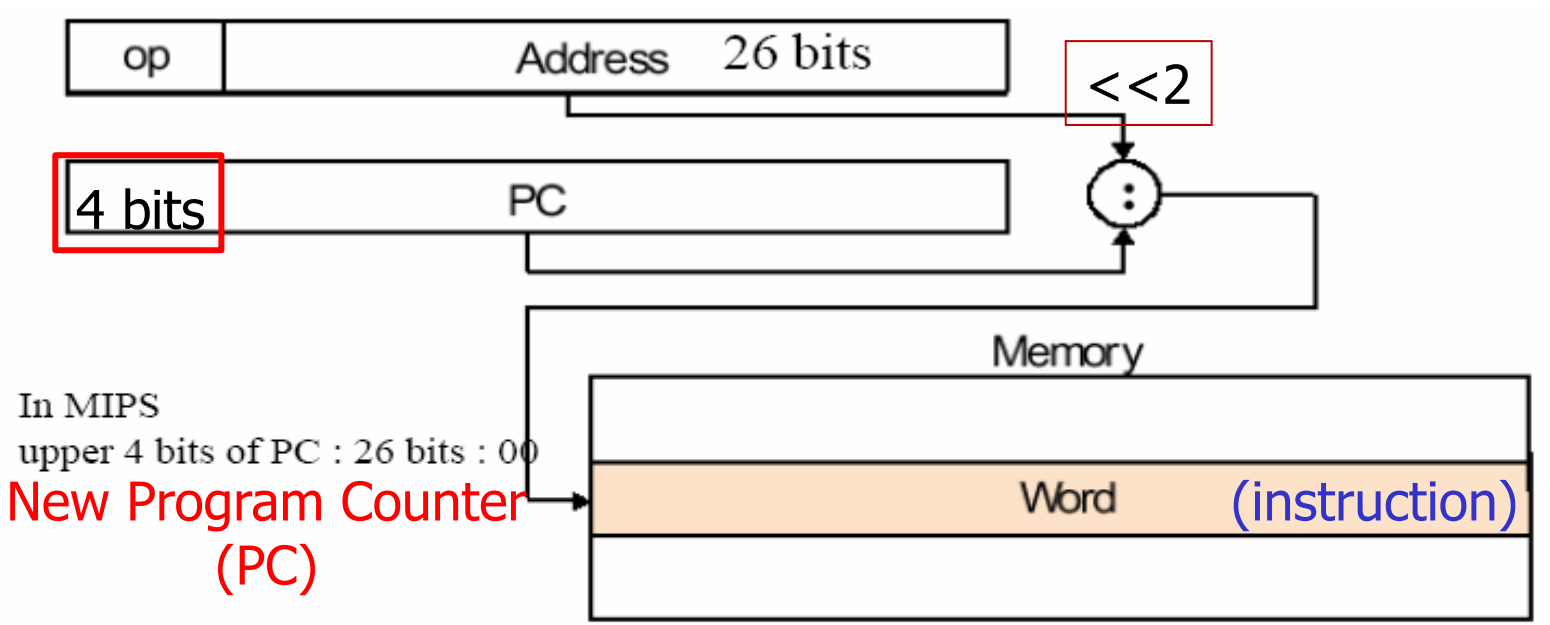
▶ Example:

```
beq R1, R2, 100
```



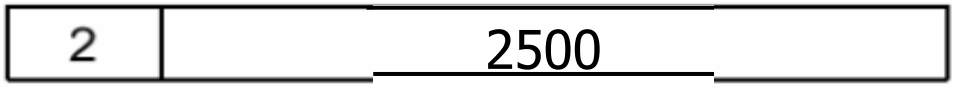
MIPS addressing modes

(5) Pseudodirect addressing (J-type) -- to get next Inst. (change PC)

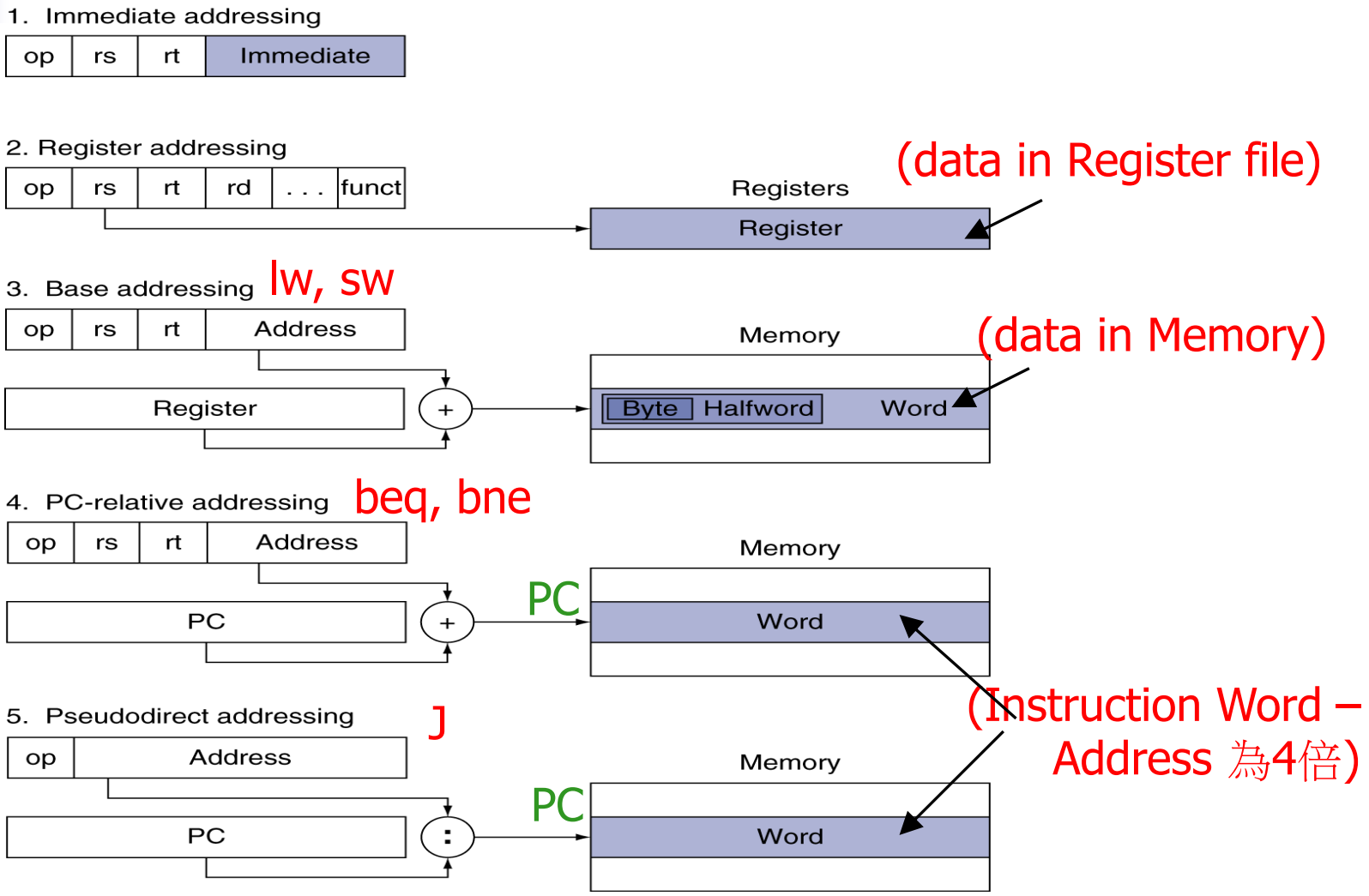


- Example:

```
j 10000
```



Summary of MIPS Addressing Modes



Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

Decoding a Binary Instruction

- Question: What is the assembly language statement corresponding to this machine instruction? **$00af8020_{hex}$**
- Answer:
  - Convert it into bit format:

    $00af8020_{hex}$ = 0000 0000 1010 1111 1000 0000 0010 0000 $_2$

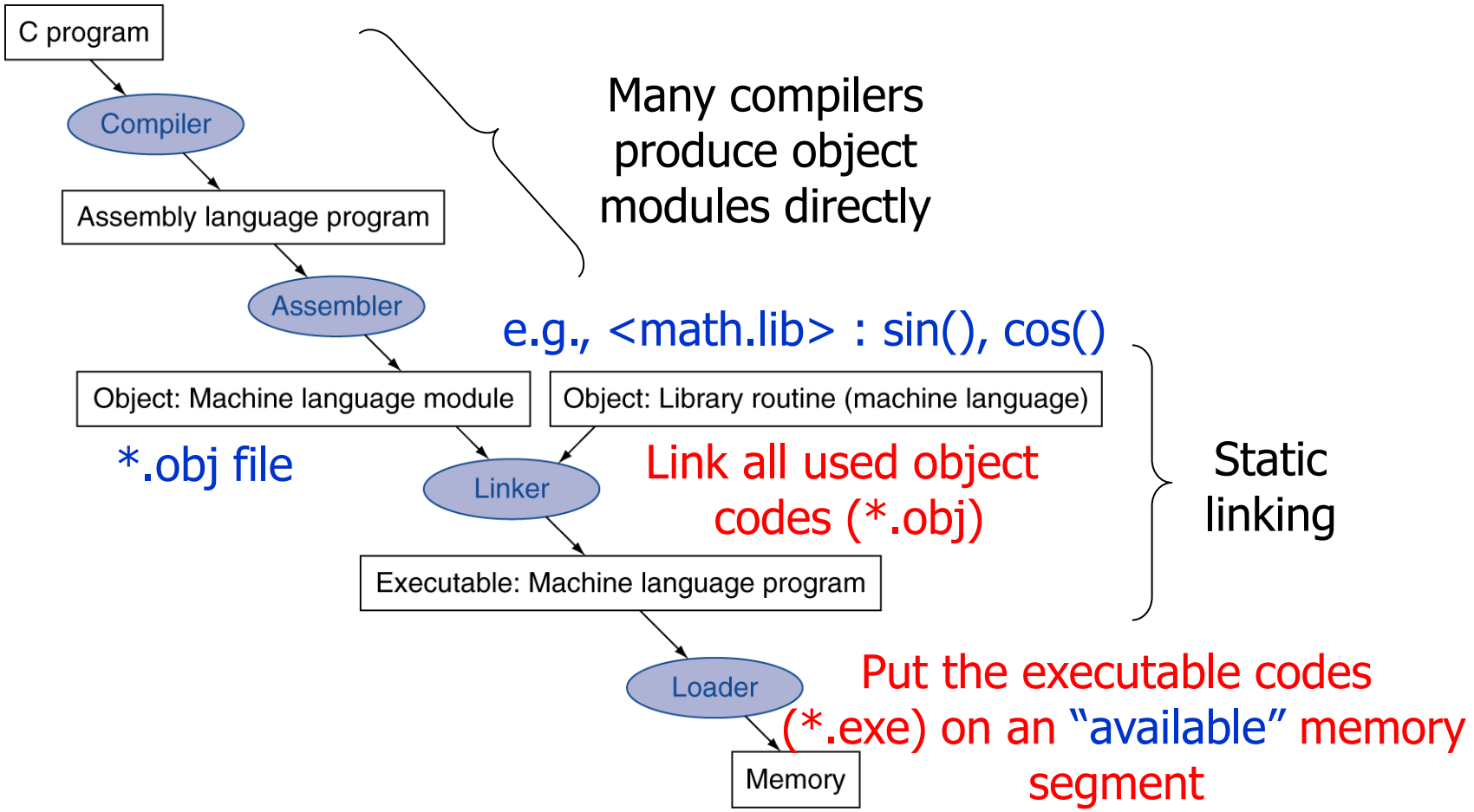
  - From the op field (bit31-26), we know it is R-type.

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

  - From the function field (bit5-0), we know it represents an add instruction.

  - Decode it ➔ add $s0, $a1, $t7

A translation hierarchy for C program



SWAP in Bubble Sorting Program

- When translating from C language to assembly language by hand, we follow the general steps:
  - (1) Allocate registers to program variables
  - (2) Produce code for the body of the procedure
  - (3) Preserve registers across the procedure invocation

- In C language: the procedure swap

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

SWAP in Bubble Sorting Program

- Recall that the memory address for MIPS refers to the byte address and so words are really 4 bytes apart.

  assume **v=$a0 and k=$a1.**

  get the address of v[k] by multiplying k by 4.

```
sll   $t1, $a1, 2        # $t1 = k * 4
add $t1, $a0, $t1        # $t1 = v + ( k * 4 )
                         # $t1 has the address of v[k]
```

  load v[k] using $t1, and then v[k] by adding 4 to $t1.

```
lw   $t0, 0($t1)         # $t0 (temp) = v[k]
lw   $t2, 4($t1)         # $t2 = v[k+1]
                         # refers to the next element of v
```

  store $t0 and $t2 to the swapped address.

```
sw  $t2, 0($t1)          # v[k] = $t2
sw  $t0, 4($t1)          # v[k+1] = $t0 (temp)
```

SWAP in Bubble Sorting Program

- MIPS assembly code of the procedure swap

Procedure body

```
swap:  sll    $t1, $a1, 2          # reg $t1 = k * 4
       add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                   # reg $t1 has the address of v[k]
       lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
       lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                   # refers to next element of v
       sw     $t2, 0($t1)          # v[k] = reg $t2
       sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
       jr     $ra                  # return to calling routine
```

Sorting an Array in C Program

- Whole Sorting program on Fig. 2.27 on Page 145

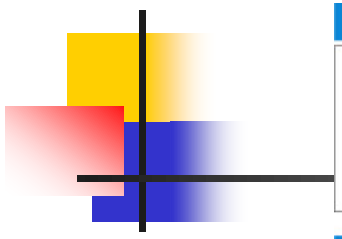
```
void sort (int v[], int n)
{
    int i, j;

    for (i=0;  i < n; i += 1) {
        for ( j = i-1;  j >=0  &&  v[j] > v [j+1];  j -=1 ) {
        swap (v, j );
        }
    }
}
```

Sorting an Array in C Program

- MIPS assembly code of the procedure **sort**

Procedure body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 Š $s3 (i Š n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 Š $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 Š $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop



Saving registers

```
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

Move parameters		move	$s2, $a0 # copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1 # copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero# i = 0
	for1tst:	slt	$t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)
		beq	$t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi	$s1, $s0, -1# j = i - 1
	for2tst:	slti	$t0, $s1,0 #reg$t0=1if$s1<0(j<0)
		bne	$t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2# reg $t1 = j * 4
		add	$t2, $s2, $t1# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)# reg $t3 = v[j]
		lw	$t4, 4($t2)# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3
		beq	$t0, $zero, exit2# go to exit2 if $t4 Š $t3
Pass parameters and call		move	$a0, $s2 # 1st parameter of swap is v (old $a0)
		move	$a1, $s1 # 2nd parameter of swap is j
		jal	swap # swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1# j -= 1
		j	for2tst # jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1 # i += 1
		j	for1tst # jump to test of outer loop

Restoring registers

```
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
```

Procedure return

```
        jr      $ra                 # return to calling routine
```

FIGURE 2.27 MIPS assembly version of procedure sort in Figure 2.26.

Summary: MIPS assembly language formats

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

MIPS machine language

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

Summary: MIPS Instruction Category

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

Summary of Instruction Format

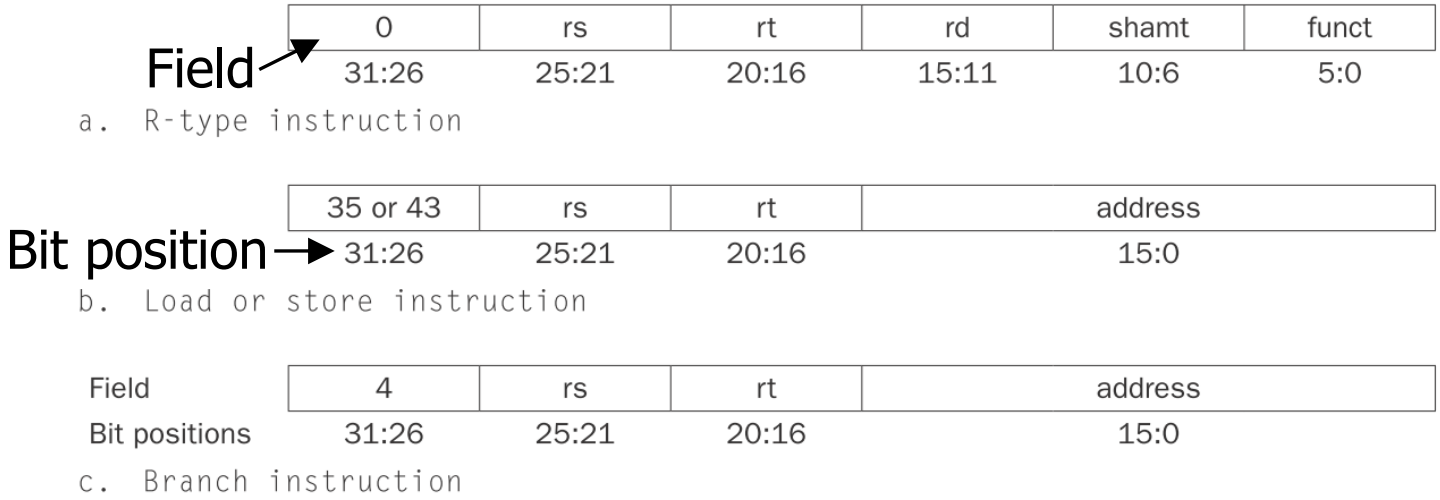
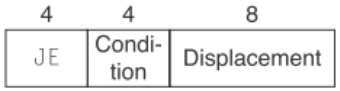


FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `AND`, `OR`, and `slt`. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC+4 to compute the branch target address.

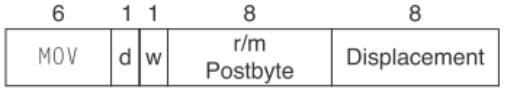
Typical Intel x86 Inst. Format (Fig.2.41)

a. JE EIP + displacement

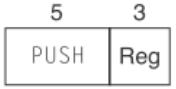


b. CALL

c. MOV EBX, [EDI + 45]

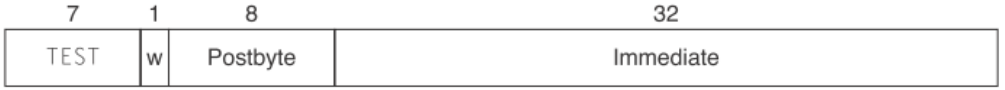


d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42



Intel x86 → 286 → 386 → 486, Pentium

Growth of x86 Instruction set over time

- More than one inst. per month over past 30 years!

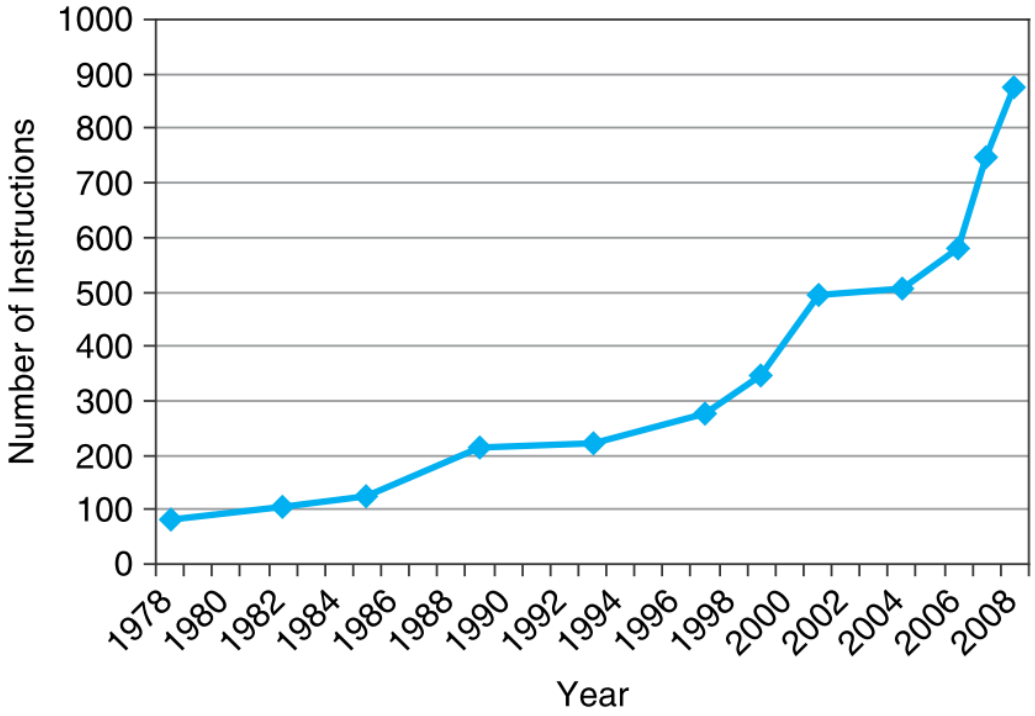


FIGURE 2.43 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

Summary

- *Simplicity favors regularity:* Regularity favors many features of the MIPS instruction set.
- *Smaller is faster:* MIPS has only 32 registers (for smaller size of field, too.)
- *Make the common case fast:* **combine** PC-relative addressing for conditional branches with immediate addressing for large constant operands.
- *Good design demands good compromises:* Balance among fields (reg. size v.s. field length in 32-bit wordlength)

- Write and simulate your first Assembly language: -- Assigned as Lab1