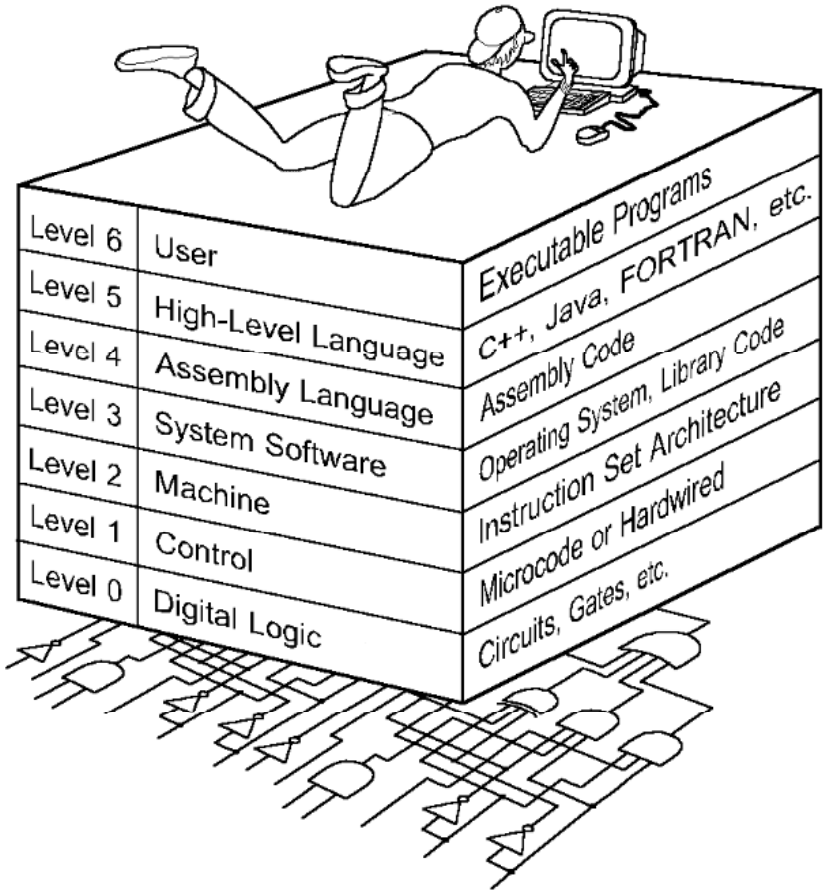
Instructions: Language of the Computer

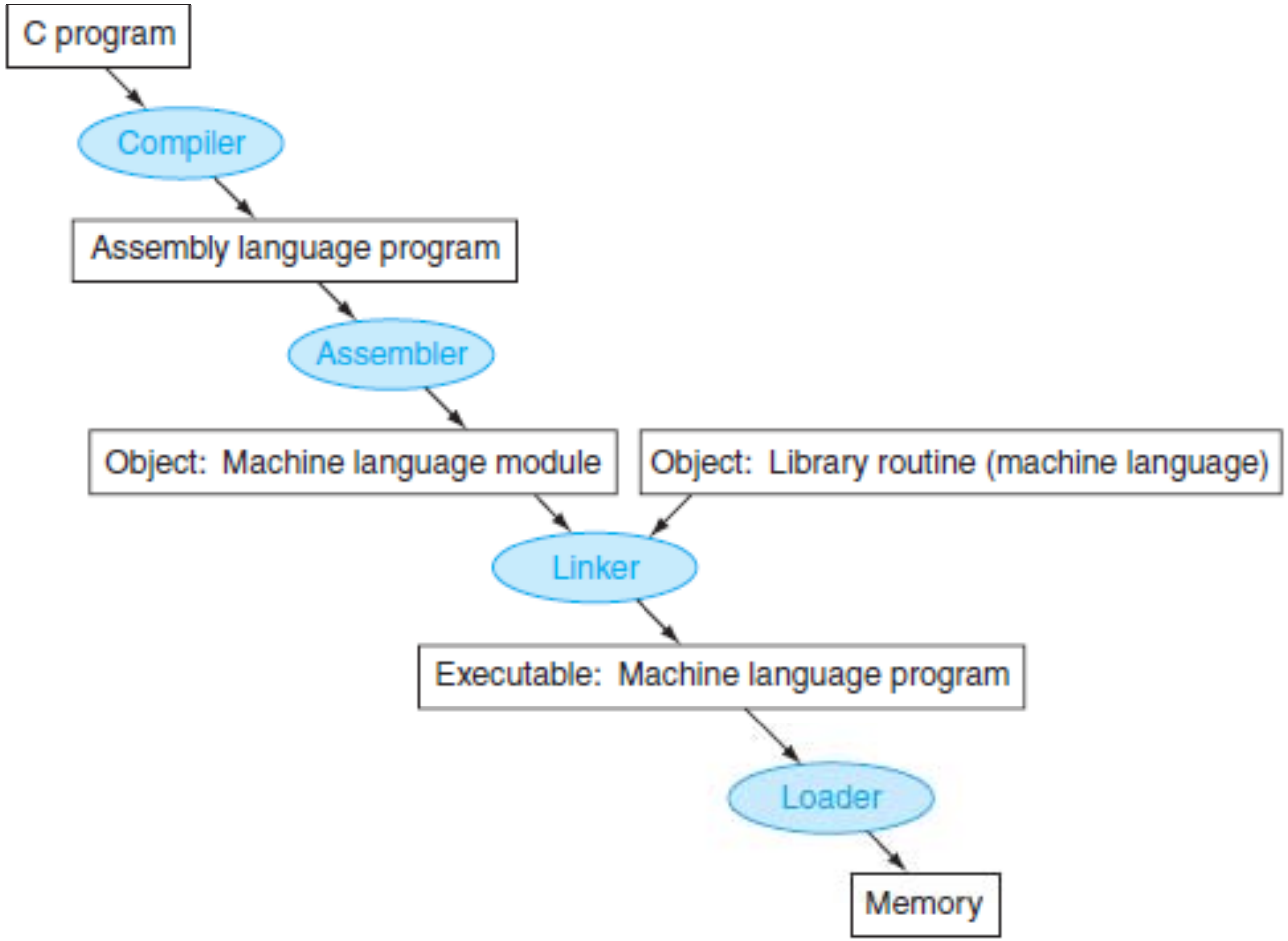
Rui Wang, Assistant professor
Dept. of Information and Communication
Tongji University

Email: ruiwang@tongji.edu.cn

Computer Hierarchy Levels



- Language understood by the computer's hardware: machine language(0101..)
- Usually discussed in terms of assembly language


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

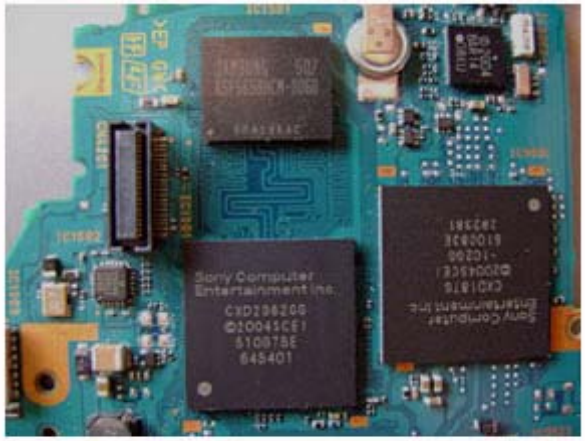
Some concepts

- Instruction set: collection of instructions that a processor can execute
- A *compiler* translates a high level language, which is architecture independent, into assembly language, which is architecture dependent.
- An *assembler* translates assembly language programs into executable binary codes.

.

MIPS assembly language

- **MIPS** is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies
- The early MIPS architectures were 32-bit, and later versions were 64-bit
- Used areas: embedded system, supercomputer



Portable Game Device with Two MIPS Chips

2.2 Operations of the computer hardware

Every computer must be able to perform arithmetic. The MIPS assembly language notation

```
add a, b, c
```

instructs a computer to add the two variables b and c and to put their sum in a.

```
add a, b, c     # The sum of b and c is placed in a.
add a, a, d     # The sum of b, c, and d is now in a.
add a, a, e     # The sum of b, c, d, and e is now in a.
```

place the sum of variables b, c, d, and e into variable a.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add a,b,c	a = b + c	Always three operands
	subtract	sub a,b,c	a = b - c	Always three operands

EXAMPLE

Compiling Two C Assignment Statements into MIPS

This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;
d = a - e;
```

The translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.

ANSWER

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c
sub d, a, e
```

EXAMPLE

A somewhat complex statement contains the five variables f, g, h, i, and j:

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

What might a C compiler produce?

ANSWER

The compiler must break this statement into several assembly instructions since only one operation is performed per MIPS instruction. The first MIPS instruction calculates the sum of g and h. We must place the result somewhere, so the compiler creates a temporary variable, called t0:

```
add t0,g,h # temporary variable t0 contains g + h
```

```
add t1,i,j  # temporary variable t1 contains i + j
```

2.3 Operands of the computer hardware

- Operands must be from a limited number of *Registers*;
- Register: a special location built directly in hardware.
  - With a size of 32 bits = length of a word
  - Byte = 8 bits
  - MIPS has 32 registers

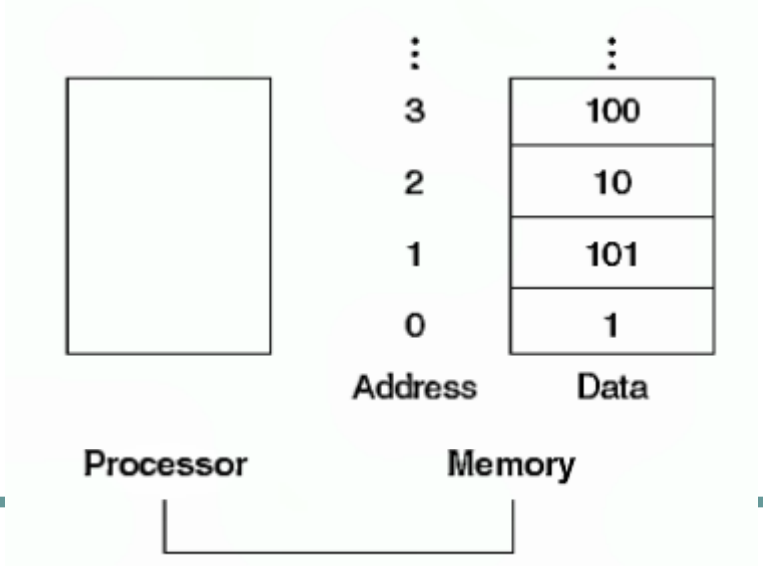
Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

2.3 Operands of the computer hardware

- Memory operands
  - Only a small amount of data is kept in registers
  - Large data structures (like array) are kept in the memory
- *Data transfer function*: A command that moves data between memory and register

2.3 Operands of the computer hardware

- To access a word in memory, the instruction must supply the memory address
  - Address: a value used to denote the location of a specific data elements in a memory array.



Load a word

EXAMPLE

Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2 as before. Let's also assume that the starting address, or *base address*, of the array is in $s3. Compile this C assignment statement:

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

ANSWER

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer A[8] to a register. The address of this array element is the sum of the base of the array A, found in register $s3, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
lw   $t0,8($s3) # Temporary reg $t0 gets A[8]
```

- 8 is a offset
- $s3 is a base register

```
add  $s1,$s2,$t0 # g = h + A[8]
```

- Alignment restriction: in MIPS, words must start at addresses that are multiples of 4.

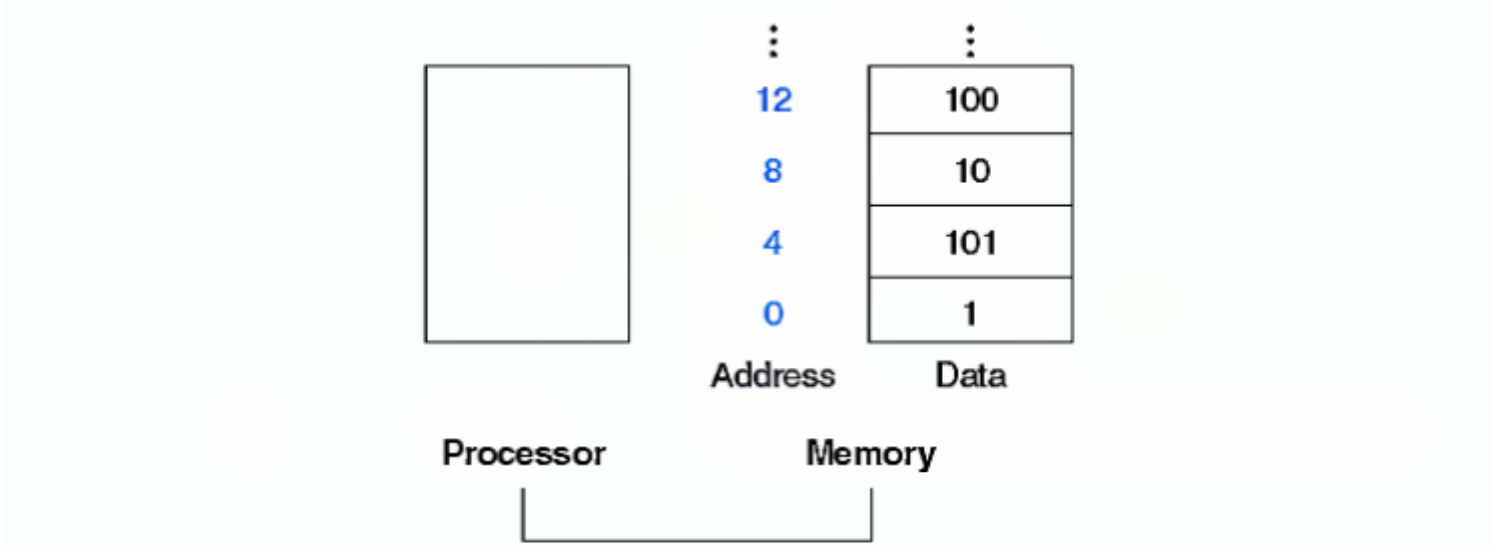
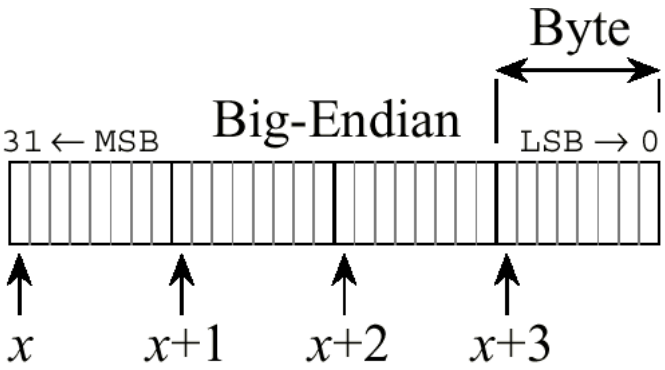
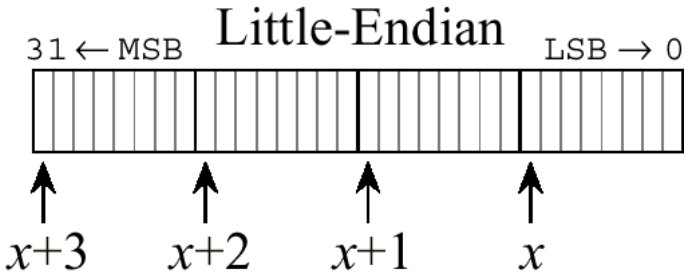


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of four: there are four bytes in a word.

Big-Endian and Little-Endian Formats

- In a byte-addressable machine, two choices for the order in which the bytes are stored in memory: most significant byte at lowest address, referred to as *big-endian*, or least significant byte stored at lowest address, referred to as *little-endian*



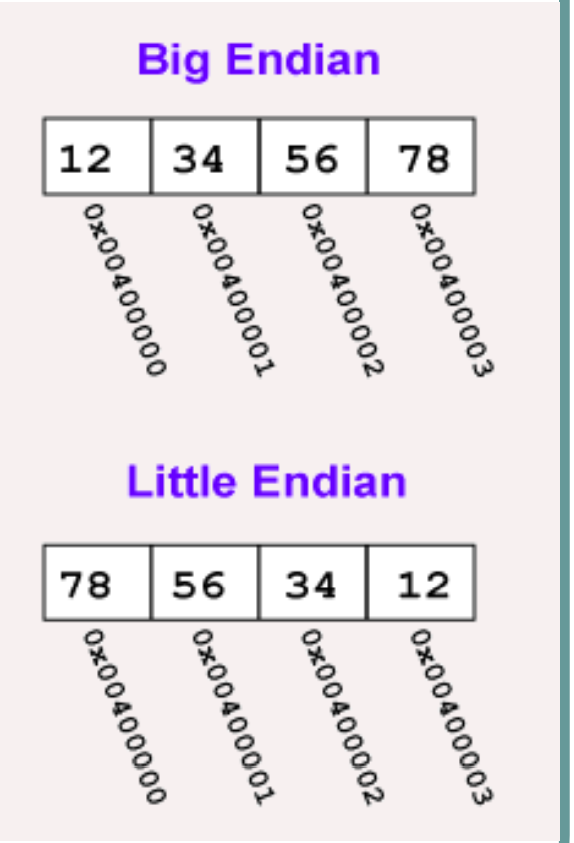


Word address is x for both big-endian and little-endian formats.

Byte #			
0	1	2	3

Byte #			
3	2	1	0

- The "most significant" byte is the one for the largest powers of two: 2^{31}, ..., 2^{24}. The "least significant" byte is the one for the smallest powers of two: 2^{7}, ..., 2^{0}.
- 32-bit pattern 0x12345678 is at address 0x00400000. The most significant byte is 0x12; the least significant is 0x78.



Store a word

EXAMPLE

Compiling Using Load and Store

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select A[8], and the add instruction places the sum in $t0:

```
lw   $t0,32($s3)   # Temporary reg $t0 gets A[8]

add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 as the offset and register $s3 as the base register.

```
sw   $t0,48($s3)   # Stores h + A[8] back into A[12]
```

Constant or Immediate operands

```
addi    $s3,$s3,4          # $s3 = $s3 + 4
```

- Add instruction with only one operand is called *add immediate* or *addi*.

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $t0, $t1, . . .	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Constant or Immediate operands

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

2.4 Representing instruction in the computer

- Decimal number: base 10 number
- Binary number: base 2 number
- Hexadecimal number: base 16 number

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

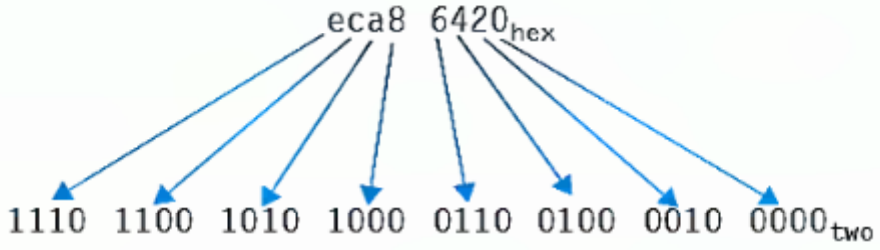
Binary-to-Hexadecimal and Back

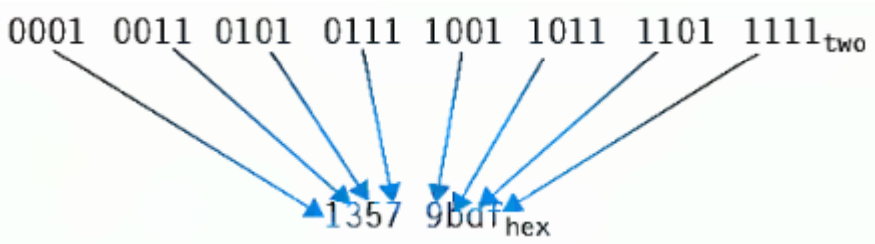
Convert the following hexadecimal and binary numbers into the other base:

$eca8\ 6420_{hex}$

$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$

Just a table lookup one way:





2.4 Representing instruction in the computer

In MIPS assembly language

- Register $s0 - $s7 = Register 16 – 23
- Register $t0 - $t7 = Register 8 – 15

Translating a MIPS Assembly Instruction into a Machine Instruction

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

The decimal representation is

0	17	18	8	0	32

- Field: each of these segment
- 0 and 32 (two fields in combination) tell the computer perform addition
- 17 = $s1, 18 = $s2, 8 = $t0
- The fifth field is unused

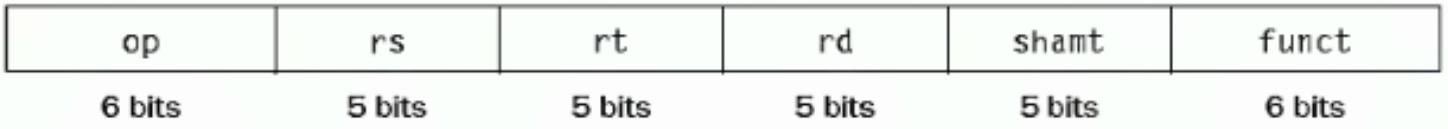
This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- MIPS instruction is 32 bit long
- Equal to the length of word

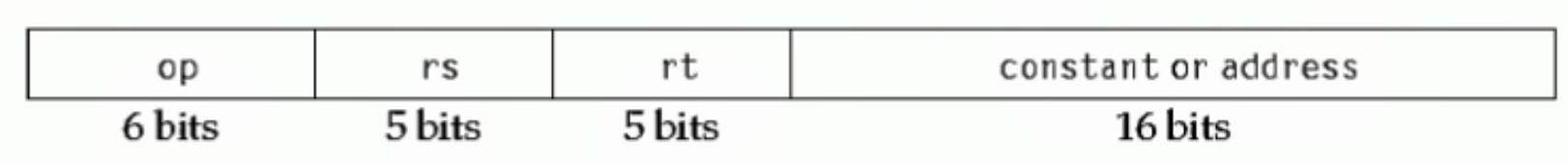
MIPS Fields *R-type* or *R-format.*

MIPS fields are given names to make them easier to discuss:



- *op:* Basic operation of the instruction, traditionally called the opcode.
- *rs:* The first register source operand.
- *rt:* The second register source operand.
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct:* Function. This field selects the specific variant of the operation in the op field and is sometimes called the *function code.*

- I-type or I-format: used for data transfer



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

EXAMPLE

We can now take an example all the way from what the programmer writes to what the computer executes. 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) # Temporary reg $t0 gets A[300]

add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]

sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

ANSWER

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

The `lw` instruction is identified by 35 (see Figure 2.6) in the first field (op). The base register 9 (`$t1`) is specified in the second field (rs), and the destination register 8 (`$t0`) is specified in the third field (rt). The offset to select `A[300]` ($1200 = 300 \times 4$) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

2.5 Logic operations

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

- Sll: shift left logical, Srl: shift right logical
- Shift: move all the bits in a word to left or right, filling the emptied bits with 0s.
- Shifting left by i bits gives the same result as multiplying by 2^i

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

- Machine language version

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

- Shamt = 4: stand for the shift amount
- 0 in both op and funct fileds
- rd = $t2
- rt = $s0
- rs = 0 is unused

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

- AND: bit-by-bit operation that leaves 1 in the result if both bits of the operands are 1.

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{two}$$

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$$

$$= \quad 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{two}$$

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

- or: bit-by-bit operation that leaves 1 in the result if either operand is 1.

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{two}$$

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$$

$$= \quad 0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000_{two}$$

- NOT: a logic bit-by-bit operation with one operand that inverts the bits, i.e., it replace every 1 with 0, and every 0 with 1;
- NOR: A logic bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

$$1111\,1111\,1111\,1111\,1100\,0011\,1111\,1111_{two}$$

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
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 $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	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 $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory

2.6 Instruction for making decisions

```
beq register1, register2, L1
```

- Go to statement label L1 if the value in register 1 *equals* the value in register 2;

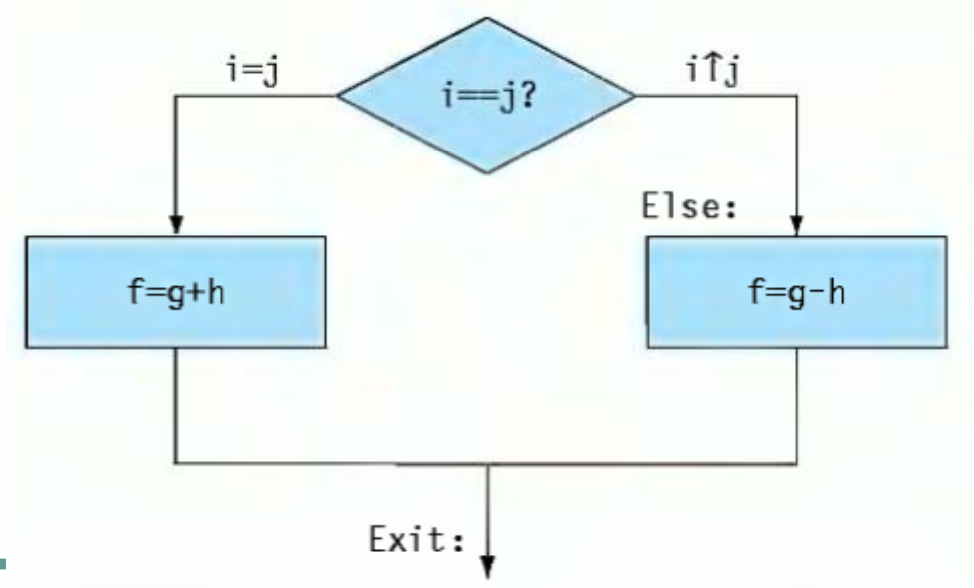
```
bne register1, register2, L1
```

- Go to statement label L if the value in register 1 *does not equal* the value in register 2;
- Two instructions are called conditional branches

EXAMPLE

In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

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



ANSWER

```
bne $s3,$s4,Else    # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2     # f = g + h (skipped if i ≠j)
```

```
 j Exit     # go to Exit
```

```
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

Loops

EXAMPLE

Here is a traditional loop in C:

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

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

The first step is to load save[i] into a temporary register. Before we can load save[i] into a temporary register, we need to have its address. Before we can add i to the base of array save to form the address, we must multiply the index i by 4 due to the byte addressing problem. Fortunately, we can use shift left logical since shifting left by 2 bits multiplies by 4 (see page 69 in Section 2.5). We need to add the label Loop to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2     # Temp reg $t1 = 4 * i
```

To get the address of save[i], we need to add $t1 and the base of save in $s6:

```
      add $t1,$t1,$s6    # $t1 = address of save[i]
```

Now we can use that address to load save[i] into a temporary register:

```
      lw  $t0,0($t1)     # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if save[i] ≠ k:

```
bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to i:

```
add  $s3,$s3,1      # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the Exit label after it, and we're done:

```
    j    Loop           # go to Loop
Exit:
```

```
slt     $t0, $s3, $s4
```

- Register $t0 is set to 1 if the value in $s3 is less than the value in $s4

```
slti    $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

- $t0 =1 if the value in $2 is less than 10

Unconditional jump instructions

- j
  - Unconditionally jumps to the instruction at the specified address
  - j exit # jump to the statement labeled exit
- jr
  - Jumps to the address contained in the specified register
  - jr $s # goto address $s

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data 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 $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	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,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

2.7 Supporting Procedures in computer hardware

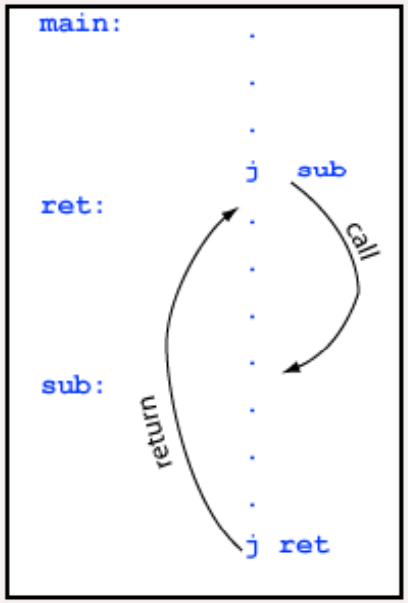
- Procedure/function: a stored subroutine that performs a specific task based on the parameters with which it is provided

- Jal (jump-and-link instruction) : An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register ($ra in MIPS)

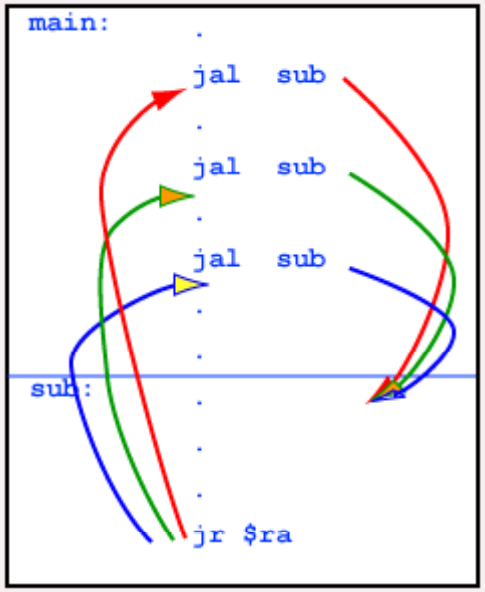
```
jal ProcedureAddress
```

2.7 Supporting Procedures in computer hardware

- Caller: the program that instigates a procedure and provides 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



Subroutine Called Once



Correct Subroutine Linkage

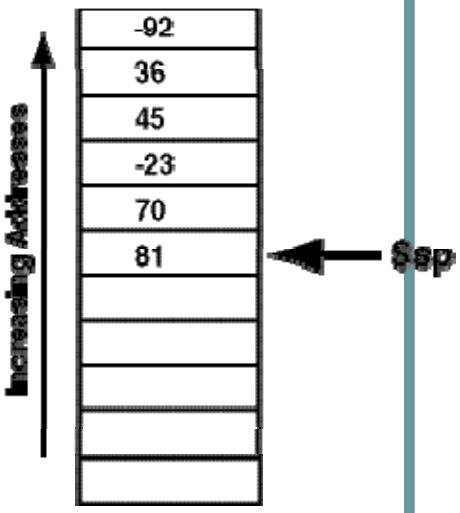
- Stack: a data structure for spilling registers organized as a last-in-first-out queue
- Stack pointer: a value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found

Using more registers

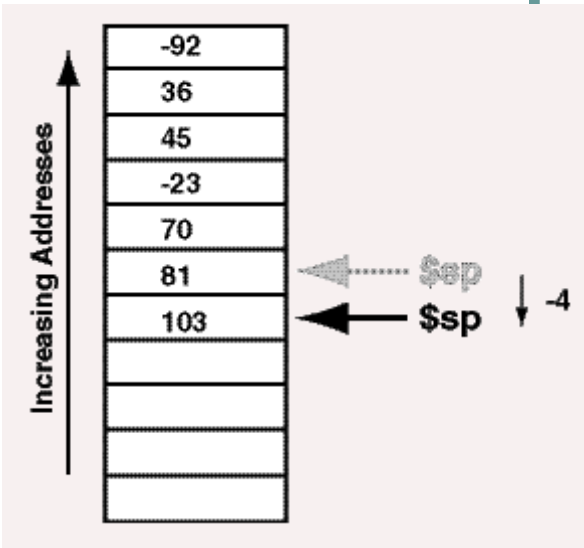
- $a0-$a3: 4 registers to pass parameters
- $v0-$v1: 2 registers for return values
- $ra: 1 return address register to return to the point of origin
- $pc (program counter): hold the address of current instruction
- $sp: stack pointer
- $gp (global pointer): point to static data

Stack

- Stack-like behavior is sometimes called "LIFO" for Last In First Out.
- The top item of the stack is 81. The bottom of the stack contains the integer -92
- $sp always points to the top of the stack.

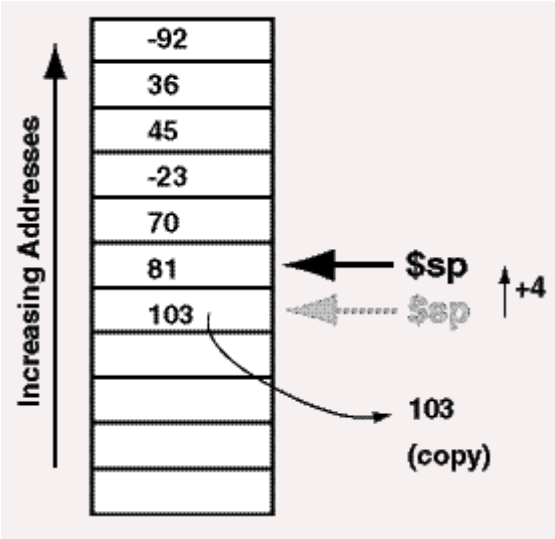


- To **push** an item onto the stack, first **subtract** 4 from the stack pointer, then store the item at the address in the stack pointer

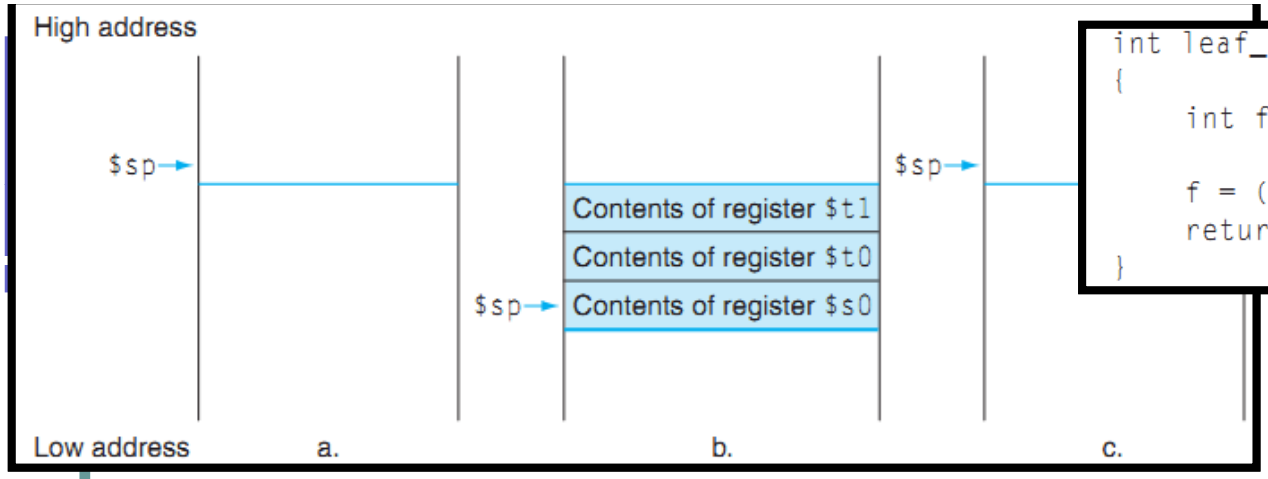



```
#  Push the item in $t0
Addi $sp, $sp, -4    # point to the place for the new item
 sw $t0, 0($sp)      # store the contents of $t0 as the new top
```

- To **pop** the top item from a stack, copy the item pointed at by the stack pointer, then **add** 4 to the stack pointer.




```
# Pop the item into $t0
 lw $t0, 0($sp)       # copy the top item to $t0
Addi $sp, $sp, 4      # point to the new place
```




```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

g,h,i,j → $a0-$a3
f → $s0

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

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

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)

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

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

- $t0–$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 on a procedure call (if used, the callee saves and restores them)

Jal: jump-and-link

• Jumps to an address and simultaneously saves the address of the following instruction in register $ra

• Saves PC+4 in register $ra to link to the following instruction to set up the procedure return

- Read and write byte and halfword

```
lb $t0,0($sp)              # Read byte from source
sb $t0,0($gp)              # Write byte to destination
```

```
lh $t0,0($sp)  # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

EXAMPLE

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?

For C language, a string is terminated with a byte whose value is 0

ANSWER

Assume that base addresses for arrays x and y are found in $a0 and $a1, while i is in $s0

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

```
lb     $t2, 0($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, 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:

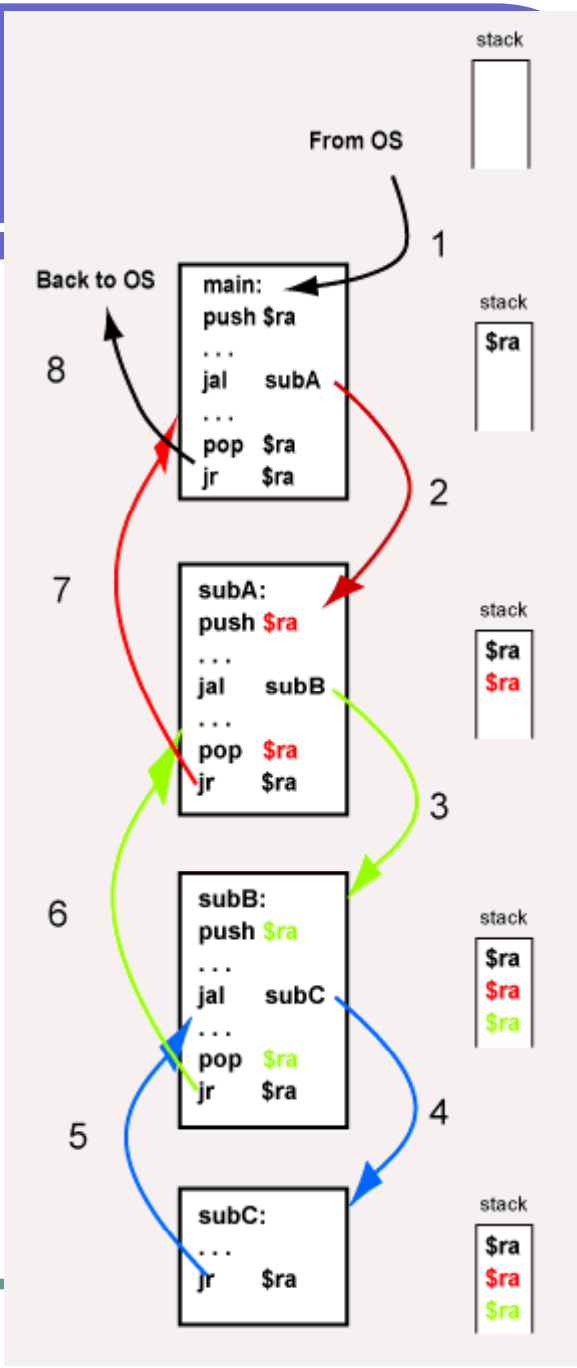
```
addi   $s0, $s0,1   # i = i + 1
j      L1           # go to L1
```

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;
                       # restore old $s0
    addi  $sp,$sp,4    # pop 1 word off stack
    jr    $ra          # return
```

Nested procedure

- Leaf procedures: procedures that do not call others
- Nested procedure: e.g. procedure A calls procedure B



Subroutine Call (done by the caller):

1. Push onto the stack any registers `$t0-$t9` that contain values that must be saved. The subroutine might change these registers.
2. Put argument values into `$a0-$a3`.
3. Call the subroutine using `jal`.

Subroutine Prolog (done by the subroutine at its beginning):

4. If this subroutine might call other subroutines, push `$ra` onto the stack.
5. Push onto the stack any registers `$s0-$s7` that this subroutine might alter.

Subroutine Body:

6. The subroutine may alter any T or A register, or any S register that it saved in the prolog (step 5).
7. If the subroutine calls another subroutine, then it does so by following these rules.

Subroutine Epilog (done by the subroutine just before it returns to the caller):

8. Put returned values in `$v0-$v1`
9. Pop from the stack (in reverse order) any registers `$s0-$s7` that were pushed in the prolog (step 5).
10. If it was pushed in the prolog (step 4), pop the return address from the stack into `$ra`.
11. Return to the caller using `jr $ra`.

Regaining Control from a subroutine (done by the caller):

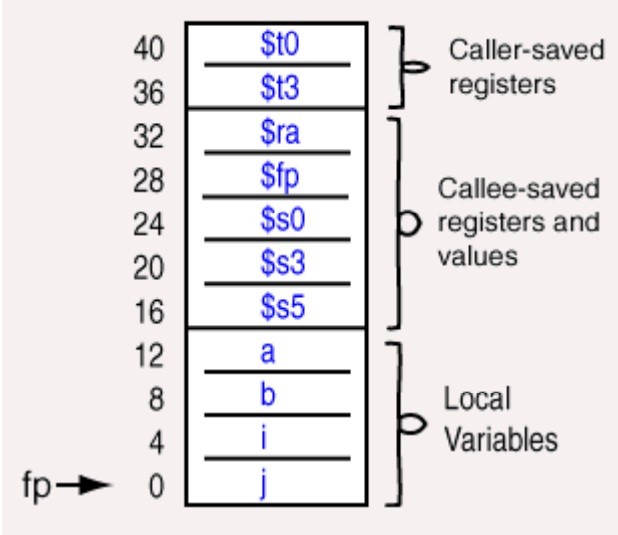
12. Pop from the stack (in reverse order) any registers `$t0-$t9` that were previously pushed (step 1).

Preserved registers across a 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

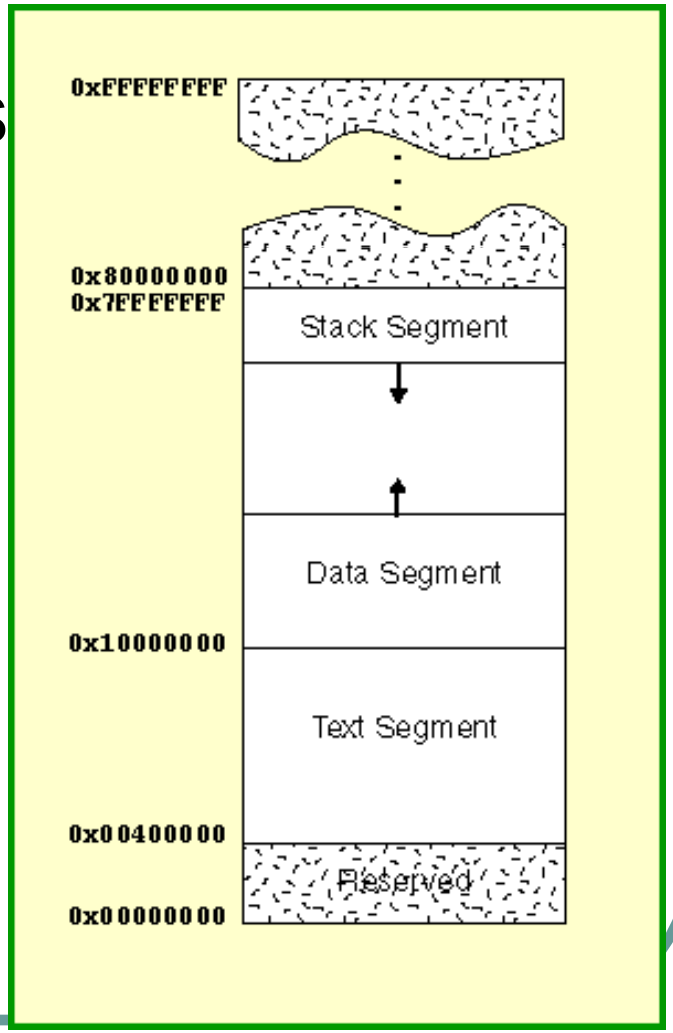
- Stack can be used to store local arrays or structures local to the procedure
- Procedure frame / activation record: the segment of the stack containing a procedure's saved registers and local variables
- Some MIPS software uses a frame pointer $fp to point to the first word of the frame



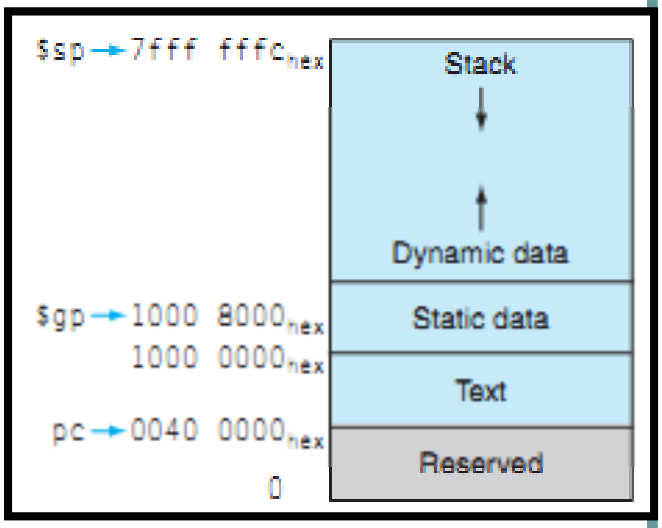

```
lw $t0,8($fp)       # get b
lw $t1,4($fp)       # get i
lw $t2,0($fp)       # get j
addu $t3,$t0,$t1    # b + i
addu $t3,$t3,$t2    # b + i + j
sw $t3,12($fp)      # a =b+i+j
```

Allocating Space for New Data on the Heap

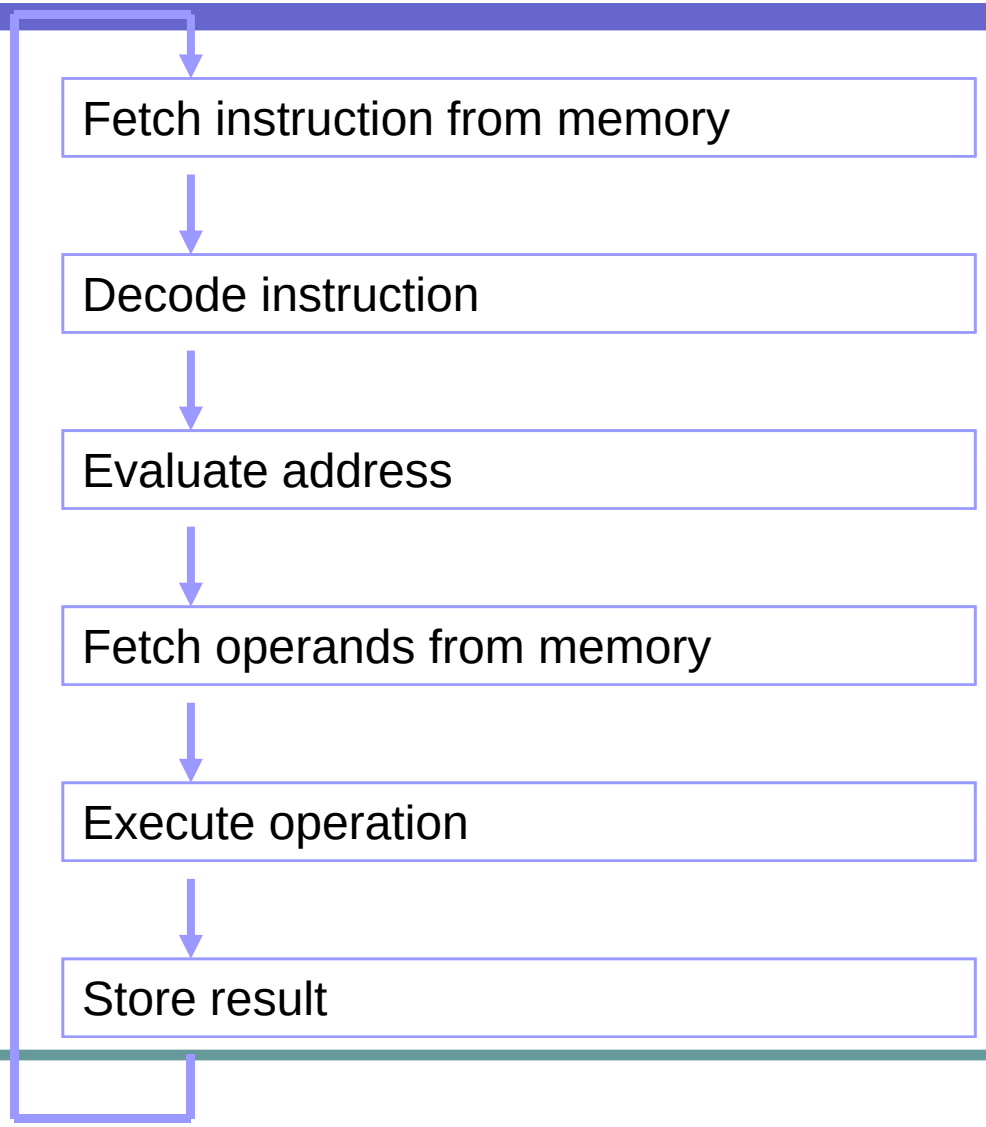
- Space for static variables and dynamic data structure is needed
- The portion of memory above the data segment that has been allocated to data structures is called the **heap**



- $sp is initialized to 0x7ffffffc
- Program code starts at 0x00400000 (PC)
- $gp (global pointer, access to static data)initialized to 0x10008000
- $gp can access the data range: 0x1000000-0x1000ffff



Instruction Processing



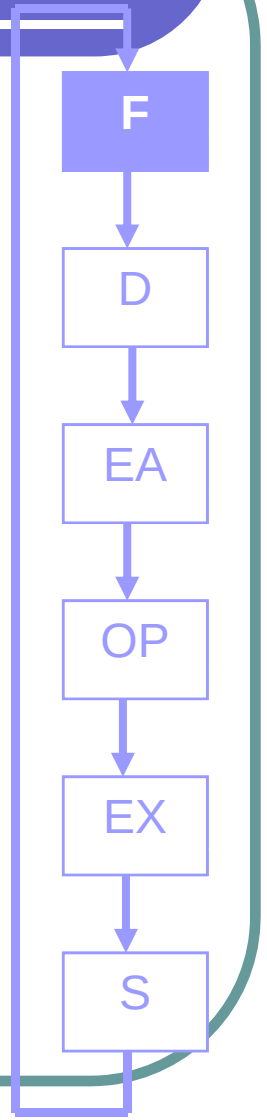
Instruction

- The instruction is the fundamental unit of work.
- Specifies two things:
  - *opcode*: operation to be performed
  - *operands*: data/locations to be used for operation

- An instruction is encoded as a sequence of bits.
  (Just like data!)
  - Often, but not always, instructions have a fixed length, such as 32 or 64 bits.
  - Control unit interprets instruction:
    generates sequence of control signals to carry out operation.
  - Operation is either executed completely, or not at all.
- A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*.

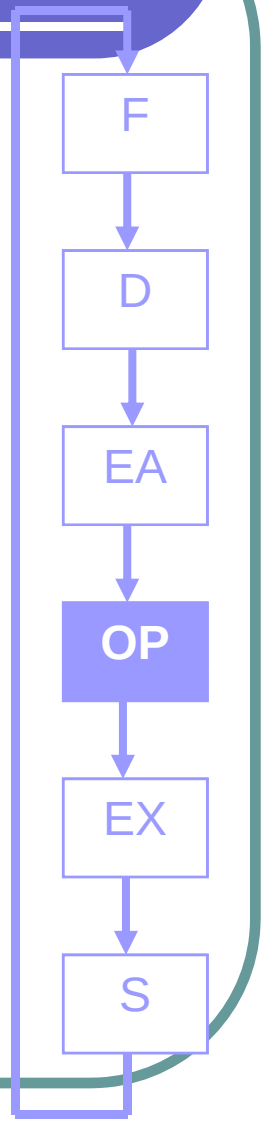
Instruction Processing: FETCH

- Load next instruction (at address stored in PC) from memory into Instruction Register (IR).
- Then increment PC, so that it points to the next instruction in sequence.
  - PC becomes PC+1.



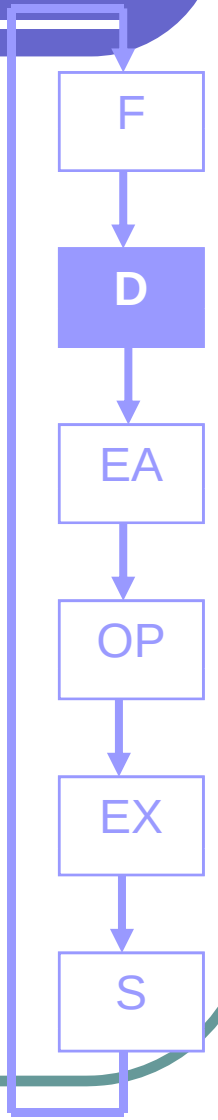
Instruction Processing: FETCH OPERANDS

- Obtain source operands needed to perform operation.
- Examples:
  - load data from memory
  - read data from register file



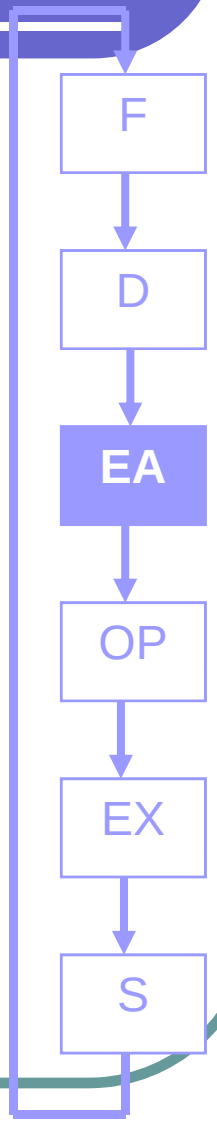
Instruction Processing: DECODE

- First identify the opcode.
- Depending on opcode, identify other operands from the remaining bits.



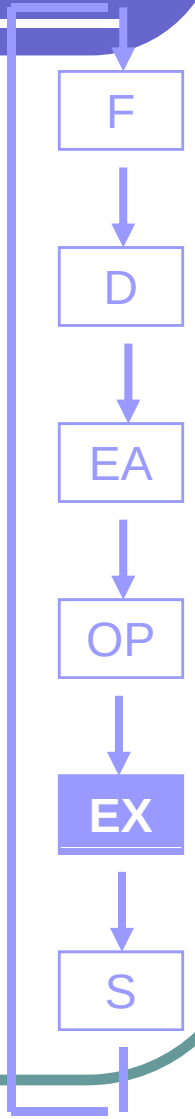
Instruction Processing: EVALUATE ADDRESS

- For instructions that require memory Access, compute address used for access.



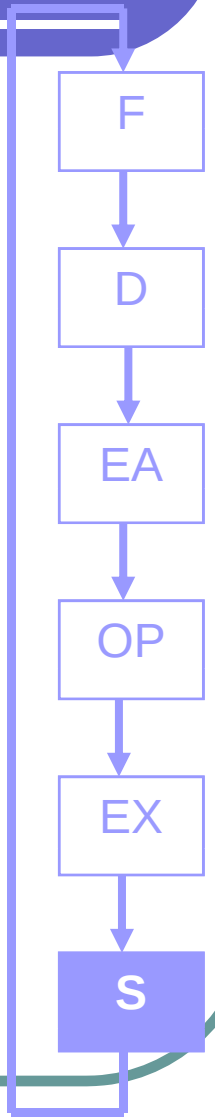
Instruction Processing: EXECUTE

- Perform the operation, using the source operands.



Instruction Processing: STORE

- Write results to destination. (register or memory)



Instruction Processing Summary

- Instructions look just like data -- it's all interpretation.
- Three basic kinds of instructions:
  - computational instructions (*like* +,-)
  - data movement instructions (*like* a=b)
  - control instructions (*like* if…then)
- Six basic phases of instruction processing:

  F → D → EA → OP → EX → S

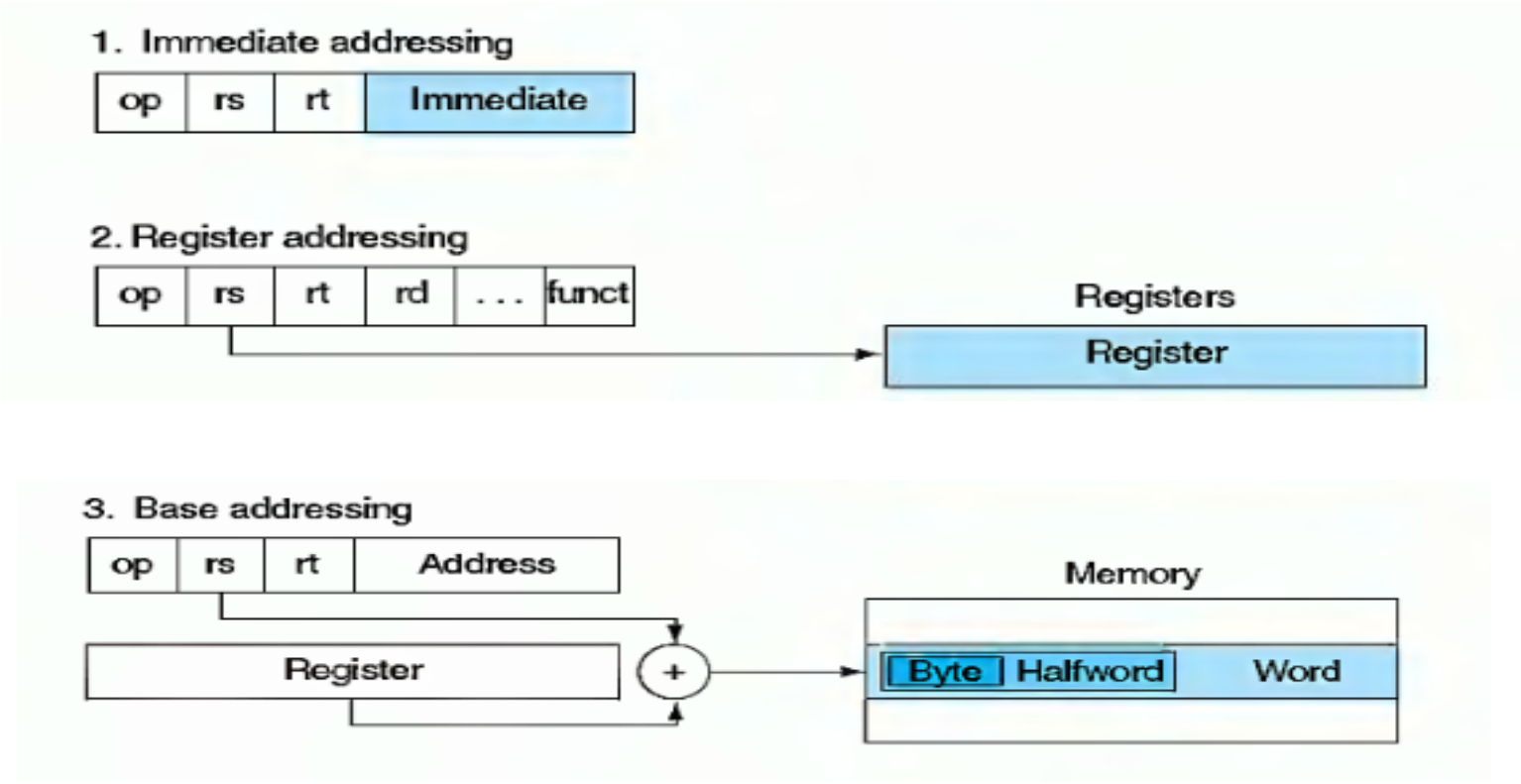
  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles

MIPS addressing modes

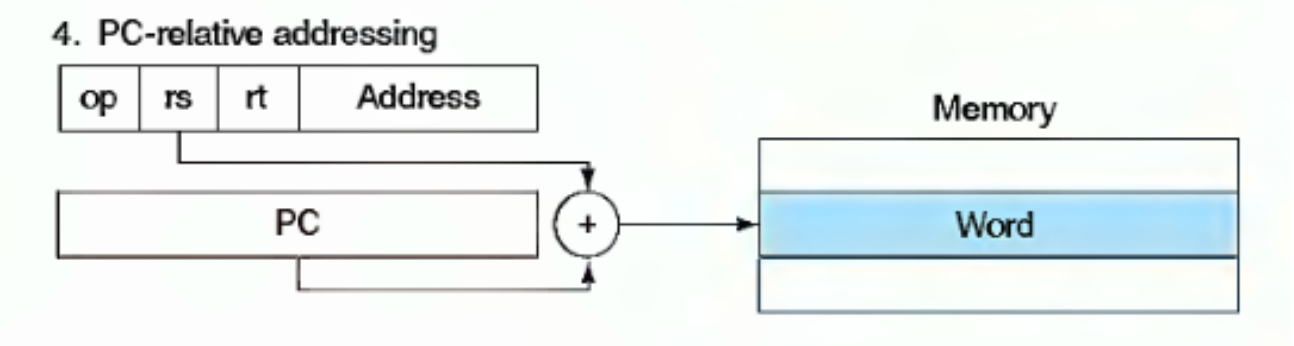
1. **Register addressing**, where the operand is a register
2. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
3. **Immediate addressing**, where the operand is a constant within the instruction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

Mode 4: Add a 16-bit address shifted left 2 bits to the PC
Mode 5: Concatenate a 26-bit address shifted left 2 with the 4 upper bits of the PC

Addressing examples



Addressing examples




```
bne  $s0,$s1,Exit   # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:


```
bne  $s0,$s1,Exit   # go to Exit if $s0 ≠ $s1
```

only 16 bits for the branch address

Addressing examples

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16}, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

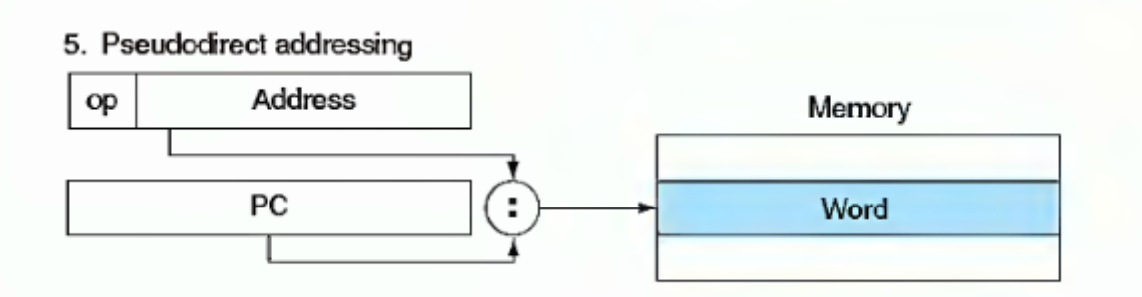
Program counter = Register + Branch address

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. The question is then, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC2000 benchmarks go to locations less than 16 instructions away. Since the program counter (PC) contains the address of the current instruction, we can

Program counter = Register + Branch address
branch within $\pm 2^{15}$ words of the current instruction

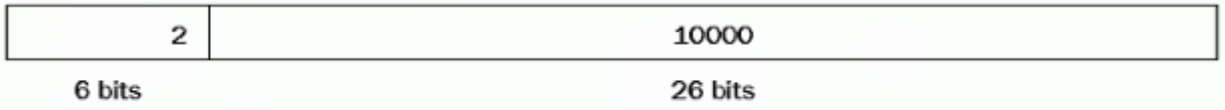
Addressing examples



The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j    10000    # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see on the next page):



Addressing examples

- lui: set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `$s0` afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to add the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register `$s0` is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Register addressing

Operand is in register

add $s1, $s2, $s3 means $s1 ← $s2 + $s3

Base addressing

Operand is in memory.

The address is the sum of a register and a constant.

lw $s1, 32($s3) means $s1 ← M[s3 + 32]

As special cases, you can implement

Direct addressing $s1 ← M[32]

Indirect addressing $s1 ← M[s3]

Which helps implement pointers.

Immediate addressing

The operand is a constant.

How can you execute $s1 ← 7?

addi $s1, $zero, 7 means $s1 ← 0 + 7

(add immediate, uses the I-type format)

PC-relative addressing

The operand address = PC + an offset

Implements position-independent codes. A small offset is adequate for short loops.

C Sort Example

```
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);
      }
   }
}
```

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

assign v and k t0 $a0 and $a1, respectively

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
```

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 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ 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 ≥ $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $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 2.34
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

assign v and n t0 $a0 and $a1, respectively
assign I and j t0 $s0 and $s1, respectively

Restoring registers

```
exitl:  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
```

Arrays versus pointers

- Clear a sequence of words in memory using arrays and pointers

```
clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
       array[i] = 0;
}
```

```
clear2(int *array, int size)
{
  int *p;
  for (p = &array[0]; p < &array[size]; p = p + 1)
       *p = 0;
}
```

```
      move  $t0,$zero     # i = 0
loop1:sll   $t1,$t0,2     # $t1 = i * 4
      add   $t2,$a0,$t1   # $t2 = &array[i]
      sw    $zero, 0($t2) # array[i] = 0
      addi  $t0,$t0,1     # i = i + 1
      slt   $t3,$t0,$a1   # $t3 = (i < size)
      bne   $t3,$zero,loop1# if () go to loop1
```

```
      move  $t0,$a0       # p = & array[0]
      sll   $t1,$a1,2     # $t1 = size * 4
      add   $t2,$a0,$t1   # $t2 = &array[size]
loop2:sw    $zero,0($t0)  # Memory[p] = 0
      addi  $t0,$t0,4     # p = p + 4
      slt   $t3,$t0,$t2   # $t3=(p<&array[size])
      bne   $t3,$zero,loop2# if () go to loop2
```

- "Arrays"must have the "multiply" and add inside the loop because i is incremented and each address must be recalculated from the new index;
- "Pointer" increments the pointer p directly. The pointer version reduces the instructions executed per iteration from 7 to 4

Thank you!