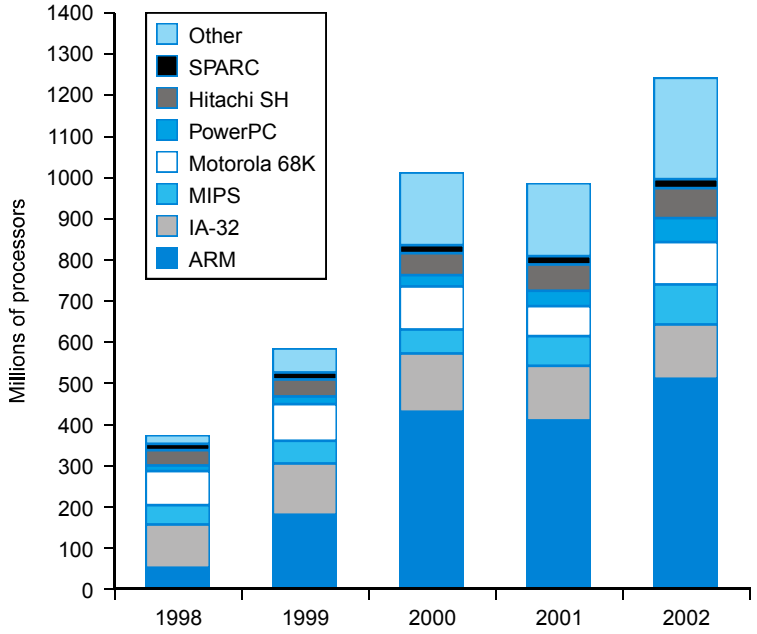
Chapter 2
Instructions: Language of the Computer

2.1 Introduction

- Language of the Machine
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



2.2 Operations of the Computer Hardware

- **All instructions have 3 operands**
- **Operand order is fixed (destination first)**

Example:

C code: `a = b + c`

MIPS 'code': `add a, b, c` `#the sum of b and c is placed in a.`

(we'll talk about registers in a bit)

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

MIPS arithmetic

- **Design Principle: simplicity favors regularity.**
- **Of course this complicates some things...**

C code:

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

MIPS code:

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

Example:

C code:

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

MIPS code:

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

Example:

C code: `f = (g + h) – (i + j);`

MIPS code:

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

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

2.3 Operands of the Computer Hardware

- **Operands of arithmetic instructions must be registers, only 32 registers provided**
- **Each register contains 32 bits**

- **Design Principle: smaller is faster. Why?**

Example: Compiling a C Assignment using registers

```
f = (g + h) – (i + j)
```

variables f, g, h, I and j are assigned to registers: $s0, $s1, $s2, $s3, and $s4 respectively

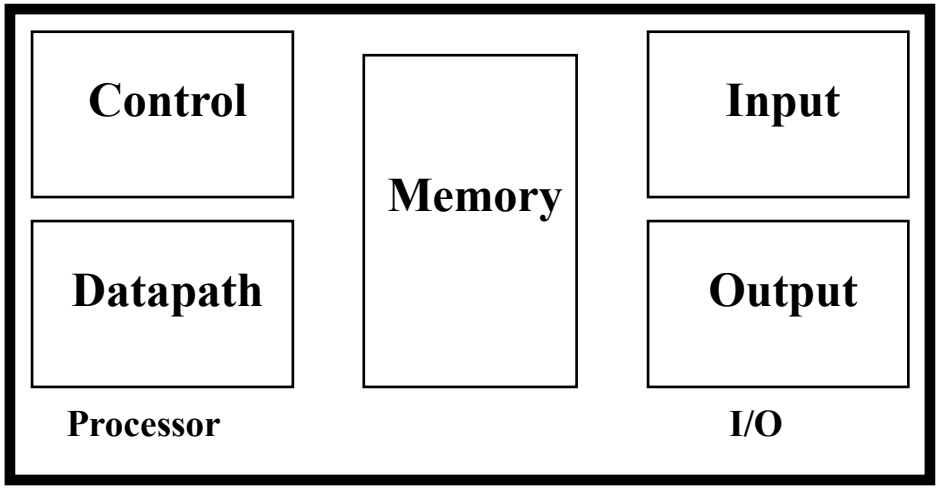
MIPS Code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

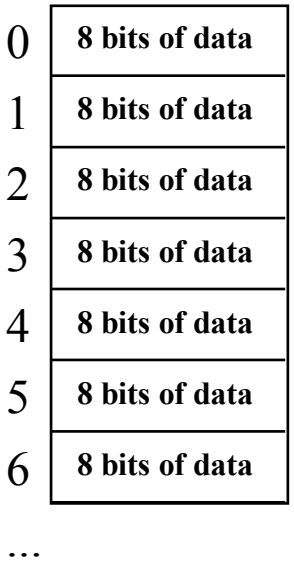
Data Transfer instructions

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Example:

g = h + A[8];

Where

g → $s1 h → $s2 base address of A → $s3

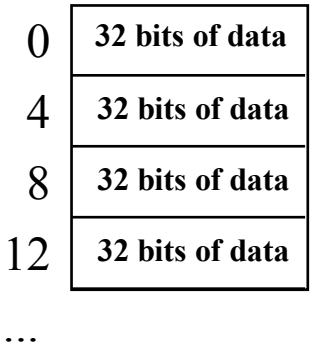
MIPS Code:

```
lw  $t0, 8($s3) # error
add $s1, $s2, $t0
```

Hardware Software Interface

Alignment restriction

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions

Example:

C code: `A[12] = h + A[8];`
where: h → $s2 base address of A → $s3

MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: ~~`add 48($s3), $s2, 32($s3)`~~

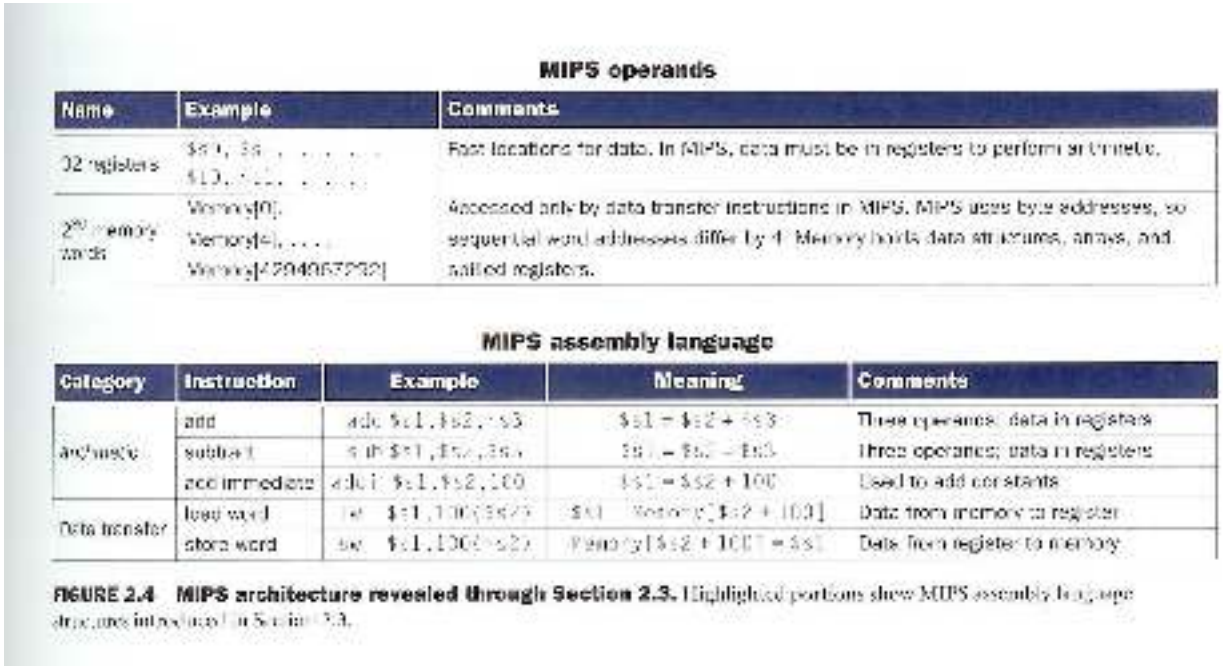
Constant or Immediate Operands

Constant in memory:

```
lw   $t0, AddrConstant4($s1)   #$t0 = constant 4
add  $s3, $s3, $t0             #$s3 = $s3 + $t0
```

Constant operand (add immediate):

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



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.

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

FIGURE 2.4 MIPS architecture revealed through Section 2.3. Highlighted portions show MIPS assembly language structures introduced in Section 2.3.

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** **Meaning**

```
add $s1, $s2, $s3        $s1 = $s2 + $s3
sub $s1, $s2, $s3        $s1 = $s2 - $s3
lw $s1, 100($s2)         $s1 = Memory[$s2+100]
sw $s1, 100($s2)         Memory[$s2+100] = $s1
```

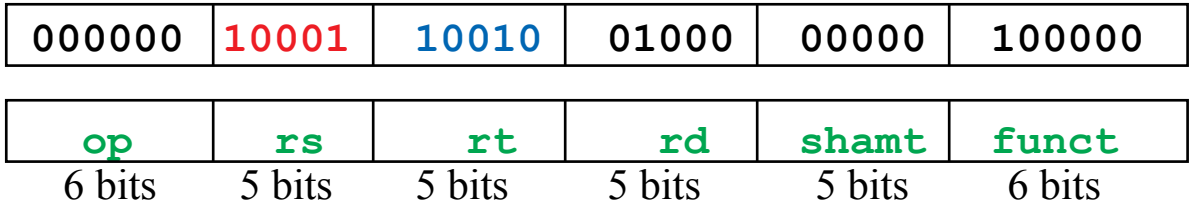
2.4 Representing Instructions in the Computer

- **Instructions, like registers and words of data, are also 32 bits long**

Example: `add $t0, $s1, $s2`

- **registers have numbers,** `$t0=8, $s1=17, $s2=18`

Instruction Format (MIPS Fields):



op: **opcode.**

rs: **first register source**

rt: **second register source**

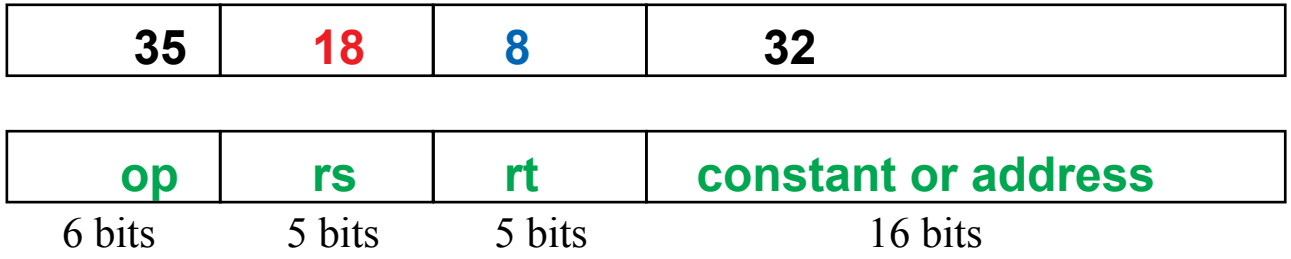
rd: **register destination**

shamt: **shift amount**

funct: **function code**

R-format and *I-format*

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`



- Where's the compromise?

Translating MIPS Assembly into Machine Language

Example:

```
    A[300] = h + A[300];
             $s2  $t1
```

is compiled into:

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

t0 → 8 t1 → 9 s2 → 18

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

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

So far we've learned:

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, ..., $s7 $t0, $t1, ..., $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15.
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.

MIPS assembly language

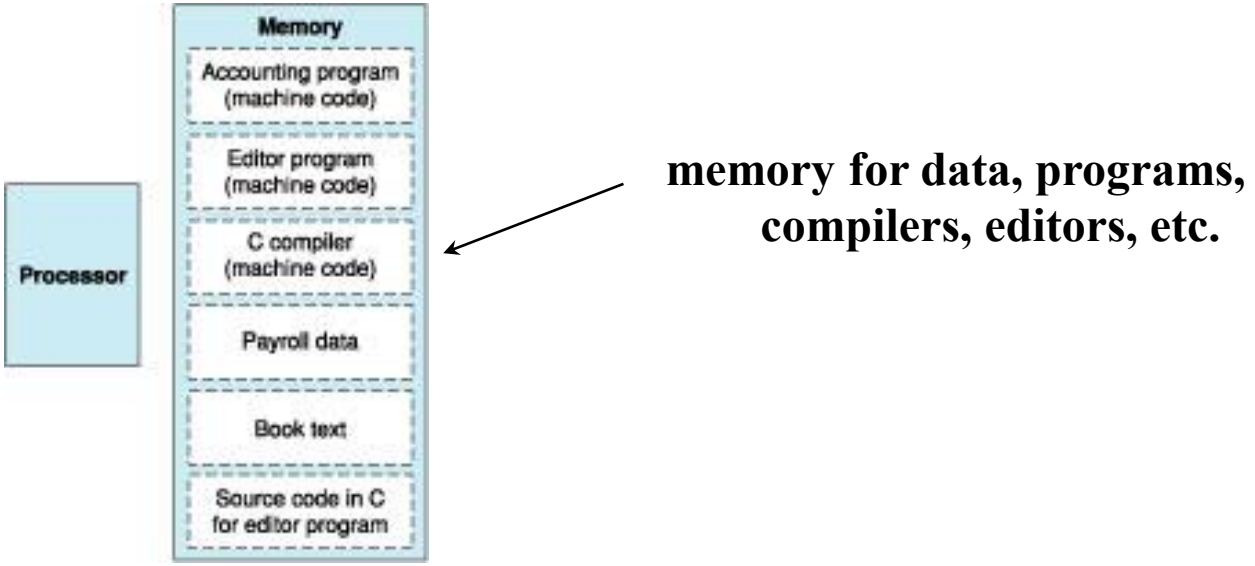
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store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

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 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

2.5 Logical 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

FIGURE 2.9 C and Java logical operators and their corresponding MIPS instructions.

Example:

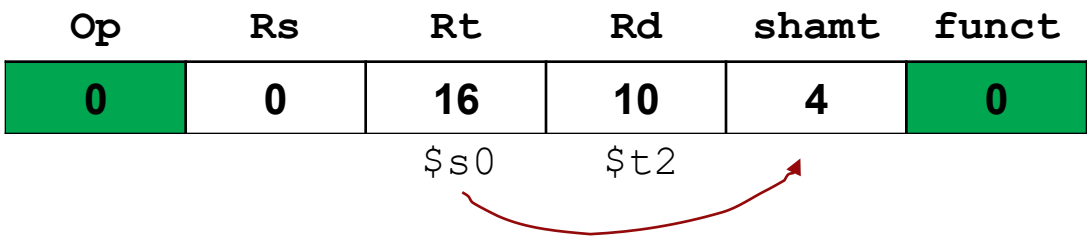
```
sll   $t2, $s0, 4    # reg $t2 = reg $s0 << 4 bits
```

`$s0` contained:

```
0000 0000 0000 0000 0000 0000 0000 1001
```

After executions:

```
0000 0000 0000 0000 0000 0000 1001 0000
```



Logical opeartions: and or nor

$t1:	0000 0000 0000 0000 0011 1100 0000 0000
$t2:	0000 0000 0000 0000 0000 1101 0000 0000
$t1 & $t2	0000 0000 0000 0000 0000 1100 0000 0000
$t1 | $t2	0000 0000 0000 0000 0011 1101 0000 0000

$t1:	0000 0000 0000 0000 0011 1100 0000 0000
$t3:	0000 0000 0000 0000 0000 0000 0000 0000
~($t1 | $t3)	1111 1111 1111 1111 1100 0011 1111 1111

```
and $t0, $t1, $t2      # $t0 = $t1 & $t2
or  $t0, $t1, $t2      # $t0 = $t1 | $t2
nor $t0, $t1, $t3      # $t0 = ~($t1 | $t3)
```

- andi: and immediate
- ori : or immediate
- No immediate version for NOR (its main use is to invert the bits of a single operand.

So far we've learned:

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, ..., $s7 $t0, $t1, ..., $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15.
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 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

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

- **MIPS conditional branch instructions:**

```
beq $t0, $t1, Label
bne $t0, $t1, Label
```

- **Example:**

```
if (i==j) h = i + j;
i→$s0   i→$s1   h→$s3

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

If-else

- **MIPS unconditional branch instructions:**

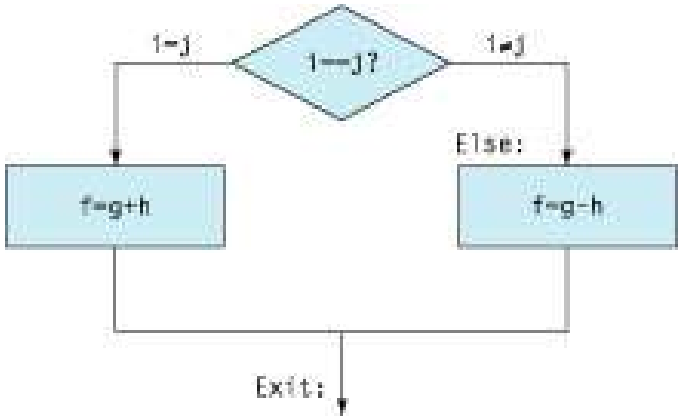
```
j  label
```

Example:

f ≡s0 g ≡s1 h ≡s2 i ≡s3 and j ≡s4

```
if (i==j)                bne $s3, $s4, Else
    f=g+h;               add $s0, $s1, $s2
else                     j Exit
    f=g-h;               Else:  sub $s0, $s1, $s2
                         Exit:  ...
```

- ***Can you build a simple for loop?***



loops

Example: Compiling a while loop in C:

```
while (save[i] == k)
       i += 1;
Assume: i ≡s3   k ≡s5  base-of A[]≡s6

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

So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 - $s3
lw $s1,100($s2)   $s1 = Memory[$s2+100]
sw $s1,100($s2)   Memory[$s2+100] = $s1
bne $s4,$s5,L     Next instr. is at Label if $s4 ≠ $s5
beq $s4,$s5,L     Next instr. is at Label if $s4 = $s5
j Label           Next instr. is at Label
```

- **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Continue

***set on less than* instruction:**

```
slt  $t0, $s3, $s4
slti $t0, $s2, 10
```

Means:

```
if($s3<$s4) $t0=1;
else $t0=0;
```

2.7 Supporting Procedures in Computer Hardware

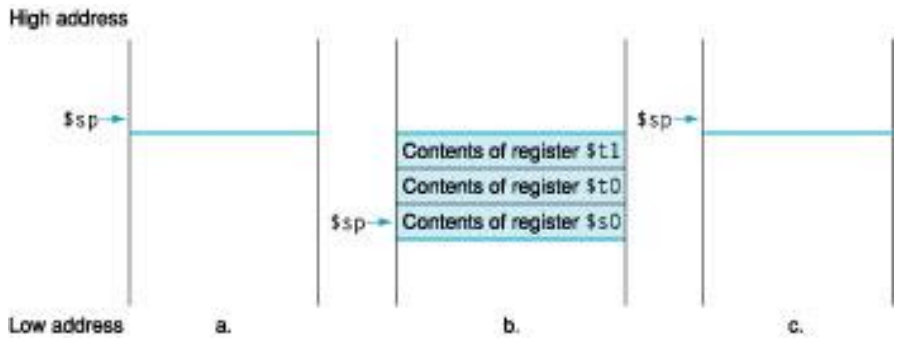
- `$a0-$a3`: four arguments
- `$v0-Sv1`: two value registers in which to return values
- `$ra`: one return address register
- `jal`: jump and save return address in `$ra`.
- `jr $ra`: jump to the address stored in register `$ra`.

Using More Registers

- Spill registers to memory
- Stack
- `$sp`

Example: Leaf procedure

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

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




```
leaf_example:
addi $sp, $sp, -12
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)

add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero

lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12

jr $ra
```

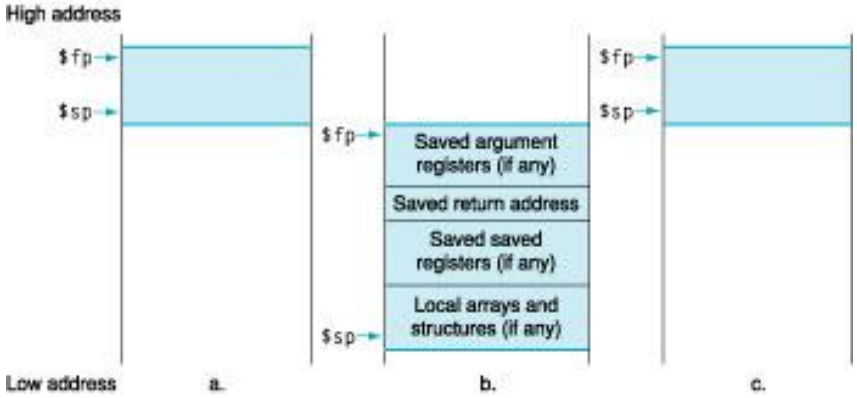
Nested Procedures

```
int fact (int n)
{
   if(n<1) return (1);
   else return (n*fact(n-1));
}
```

```
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)

slti $t0, $a0, 1
beq $t0, $zero, L1

addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra

L1: addi $a0, $a0, -1
    jal fact

lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8

mul $v0, $a0, $v0
jr $ra
```

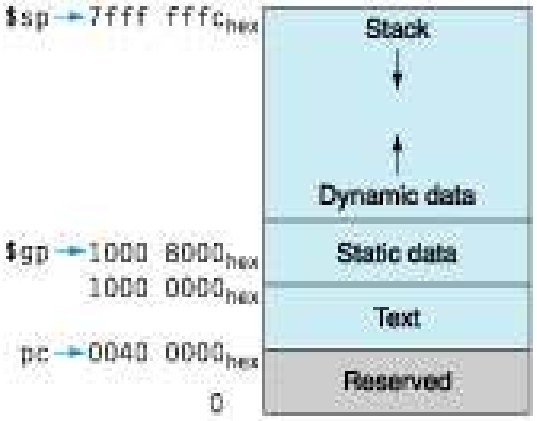
Allocating Space for New Data on the Stack

- Stack is also used to store Local variables that do not fit in registers (arrays or structures)
- Procedure frame (activation record)
- Frame Pointer



Allocating Space for New Data on the Heap

- **Need space for static variables and dynamic data structures (heap).**



Policy of Use Conventions

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

Register 1 ($at) reserved for assembler, 26-27 for operating system

2.8 Communicating with People

- For text process, MIPS provides instructions to move bytes (8 bits ASCII):

```
lb $t0, 0($sp)
```

Load a byte from memory, placing it in the rightmost 8 bits of a register.

```
sb $t0, 0($gp)
```

Store byte (rightmost 8 bits of a register) in to memory.

- Java uses Unicode for characters: 16 bits to represent characters:

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

Example:

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

   i=0;
   while((x[i]=y[i])!='\0')
        i+=1;
}
```

```
strcpy:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $s0, $zero, $zero
L1:add  $t1, $s0, $a1
   lb   $t2, 0($t1)
   add  $t3, $s0, $a0
   sb   $t2, 0($t3)
   beq  $t2, $zero, L2
   addi $s0, $s0, 1
   j    L1
L2:lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

2.9 MIPS Addressing for 32-Bit Immediates and Address

32-Bit Immediate Operands

- **load upper immediate `lui` to set the upper 16 bits of a constant in a register.**

```
lui $t0, 255 # $t0 is register 8:
```

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing: `lui $t0, 255`

0000 0000 1111 1111	0000 0000 0000 0000

Example: loading a 32-Bit Constant

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

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

```
lui $s0, 61
```

The value of `$s0` afterward is:

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

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

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

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

Addressing in Branches and Jumps

- **jump addressing**

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

2	10000
6 bits	26 bits

- **Unlike the jump, the conditional branch must specify two operands:**

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

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

- **No program could be bigger than $2^{16,}$ which is far too small**
- **Solution: PC-relative addressing**

Program counter = PC + Branch address

PC points to the next instruction to be executed.

Examples:

Showing Branch Offset in Machine Language

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

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

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	.....					

Example

Branching Far Away

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

```
beq $S0, $s1, L1
```

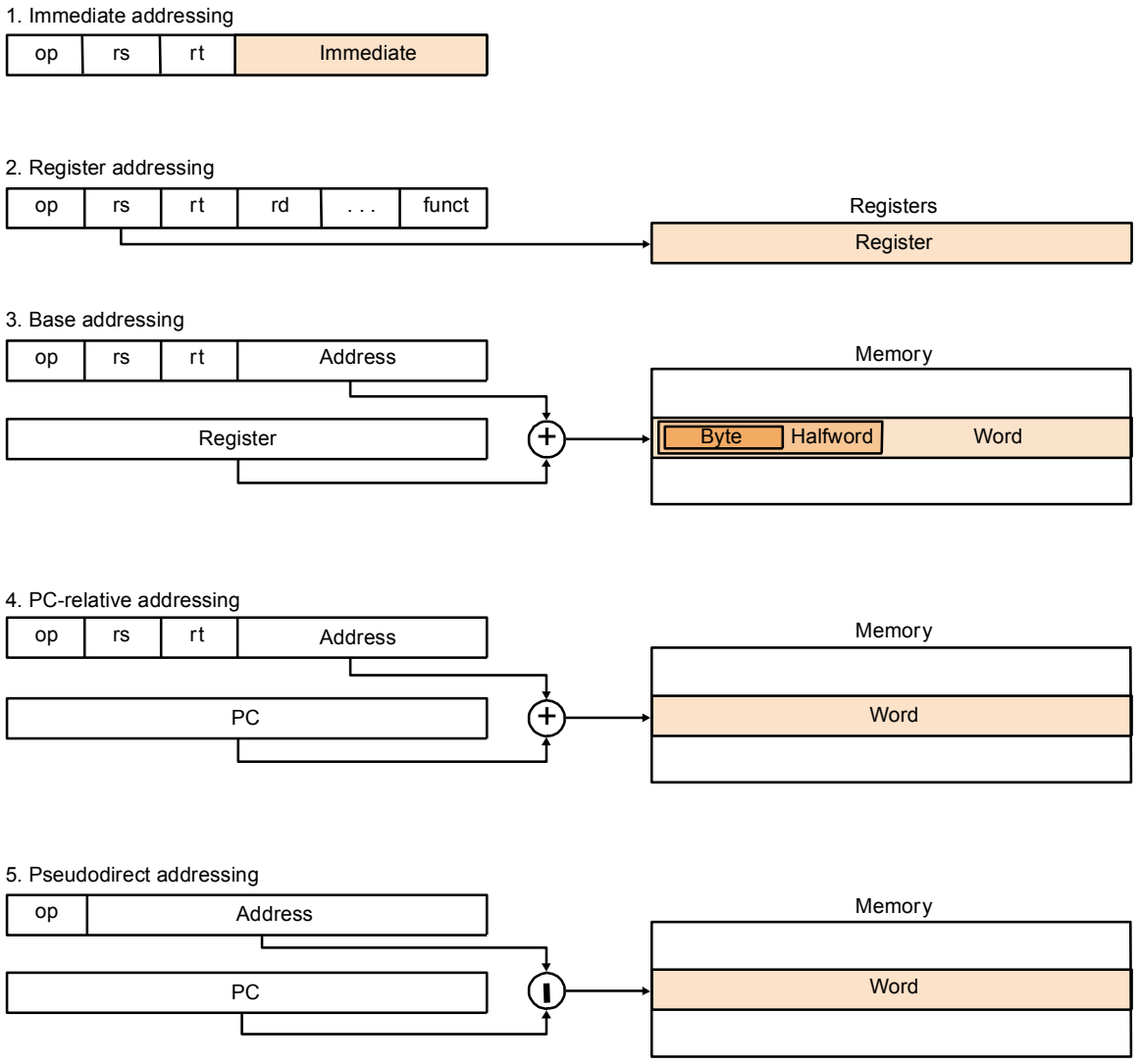
replace it by a pair of instructions that offers a much greater branching distance.

These instructions replace the short-address conditional branch:

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

MIPS Addressing Modes Summary

1. Register addressing
2. Base or displacement addressing
3. Immediate addressing
4. PC-relative addressing
5. Pseudodirect addressing: 26 bits concatenated with the upper bits of the PC.



Decoding Machine Language

What is the assembly language corresponding to this machine code:

$00af8020_{hex}$

```
0000 0000 1010 1111 1000 0000 0010 0000
```

From fig 2.25, it is an R-format instruction

```
000000   00101   01111   10000   00000   100000
```

from fig 2.25, Represent an add instruction

```
add $s0, $a1, $t7
```