Lecture 3
Machine Instructions

School of Computer Science and Engineering
Soongsil University

2. Instruction: Language of the Compute

2.5 Representing Instructions in the Computer

- **Instruction format**
  - Layout of the instruction
  - A form of representation of an instruction composed of fields of binary numbers
- **R(Register)-type instruction format**

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

  - op: Basic operation of the instruction (opcode)
  - rs, rt: The first and second register source operands
  - rd: The register destination operand
    It gets the result of the operation.
  - shamt: Shift amount (Explained in §2.6)
  - funct(function code): This field selects the specific variant of the operation in the op field. (cf) opcode extension

Example: Machine instruction

- **Translating assembly instruction to machine instruction**

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

[Answer]

0	**17**	**18**	**8**	**0**	**32**

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

- First and last fields: addition operation
- Second field: register number of the first source operand ($s1)
- Third field: register number of the other source operand ($s2)
- Fourth field: number of the destination register ($t0)
- Fifth field: unused in this instruction (set to 0)

I-type Instruction Format

Design Principle 3 :

Good design demands good compromises.

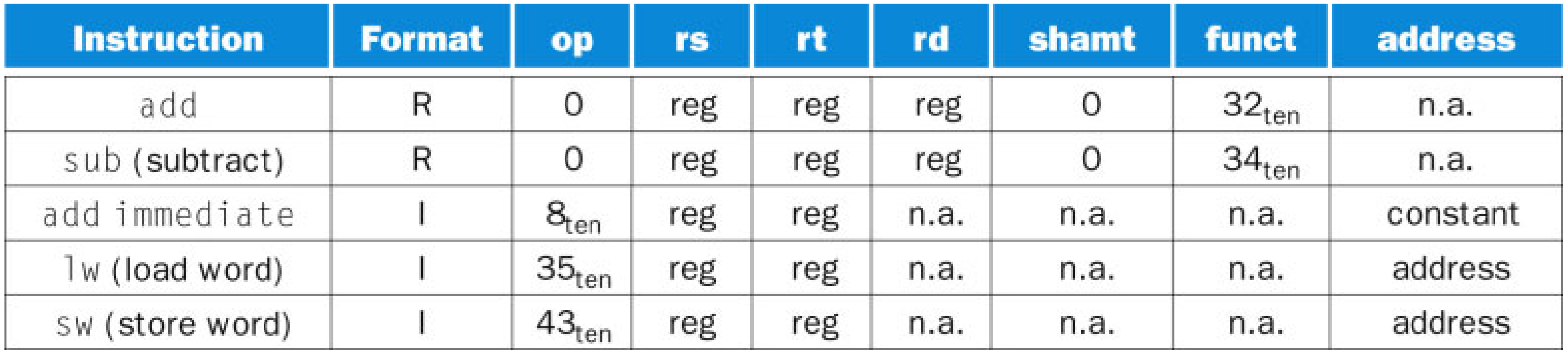
[Example] MIPS instructions

- Same length but different formats

- **I(Immediate)-type instruction format**

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

Figure 2.5



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: A[300]=h+A[300];

```
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]
```

[Answer]

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

The BIG Picture

- Instructions are represented as numbers.
- Programs are stored in memory to be read or written, just like data.

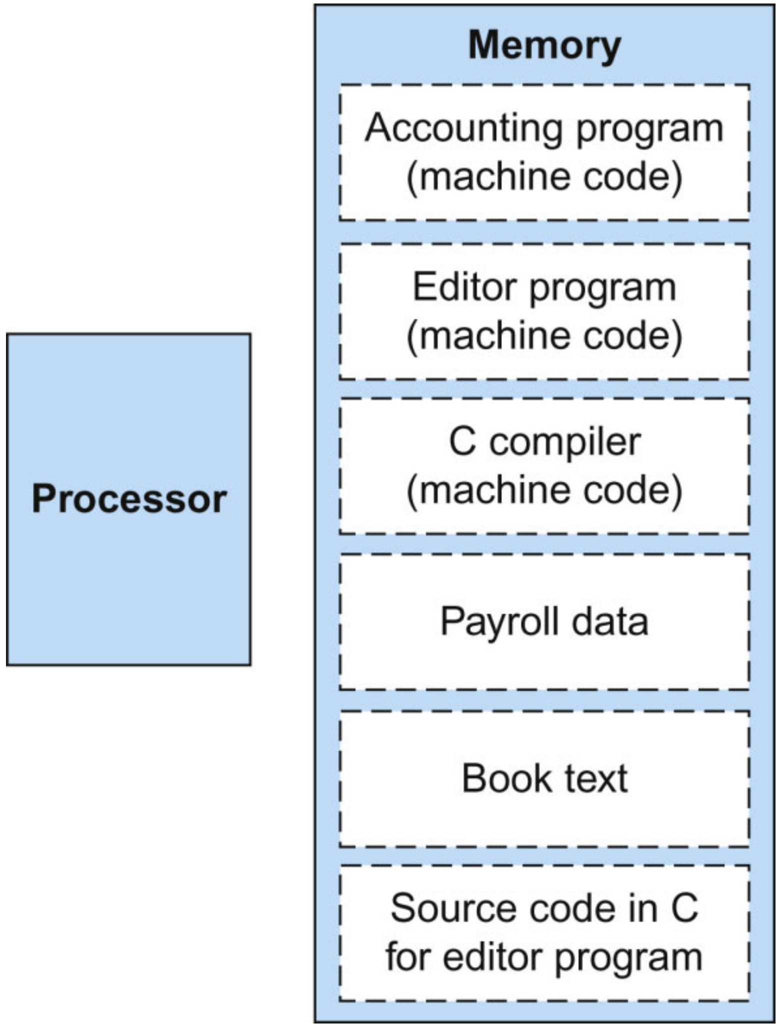
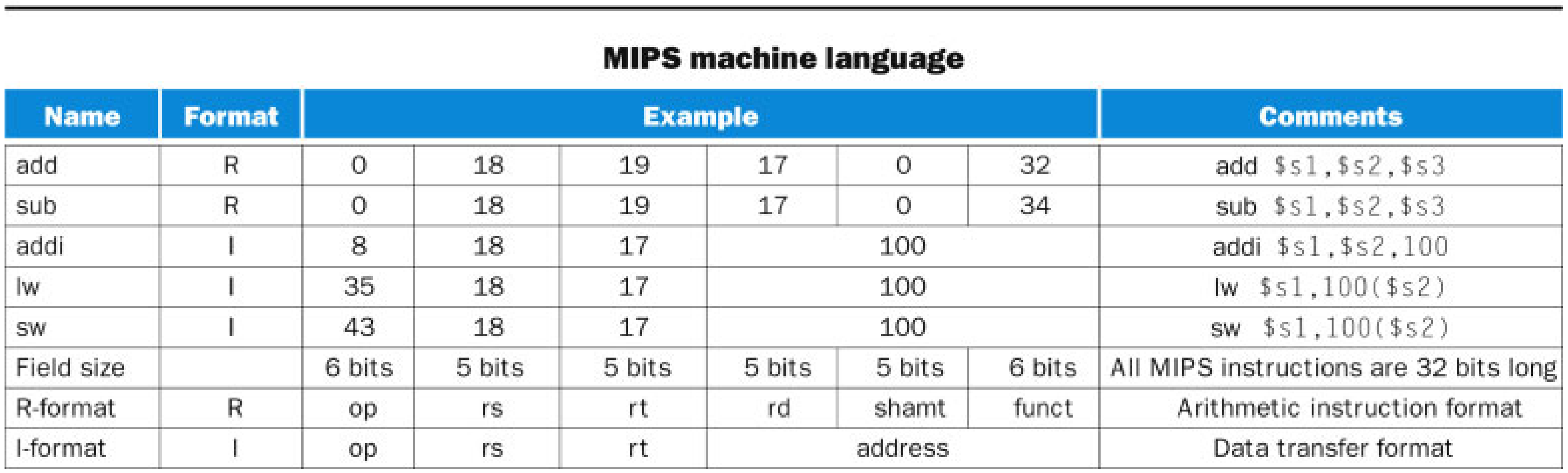

Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

Figure 2.7 Stored-program concept

MIPS Machine Language

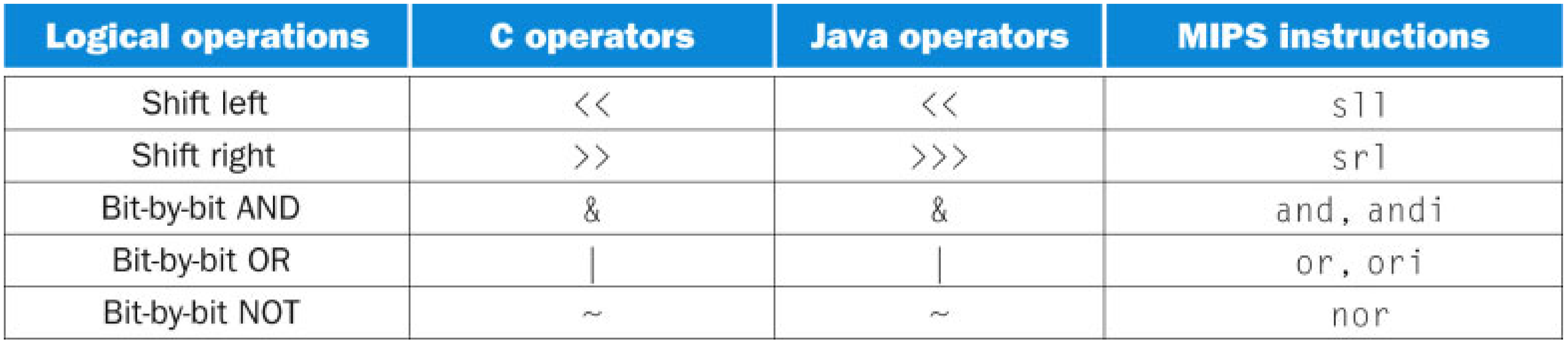


MIPS machine language

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`
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

Figure 2.6

2.6 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`

Fig. 2.8

Shift Operations

$$X = x_{31}x_{30} \cdots x_0$$

1. Logical shift

- Logical shift right (X) = $0x_{31}x_{30} \cdots x_1$
- Logical shift left (X) = $x_{30} \cdots x_0 0$

2. Arithmetic shift (for 2's complement)

- Arithmetic shift right (X) = $x_{31}x_{31}x_{30} \cdots x_1$ (cf) $\div 2$
- Arithmetic shift left (X) = $x_{30} \cdots x_0 0$ (cf) $*2$

3. Circular shift (= rotate)

- Circular shift right (X) = $x_0 x_{31}x_{30} \cdots x_1$
- Circular shift left (X) = $x_{30} \cdots x_0 x_{31}$

MIPS Shift Instructions

- `sll (shift left logical); funct = 000 000`
- `srl (shift right logical); funct = 000 010`
- `sra (shift right arithmetic); funct = 000 011`

- **Instruction format**
  - `sll $t2, $s0, 4   # reg $t2 = reg $s0 << 4 bits`

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

- `sllv (shift left logical variable), srlv, srav`
  - `sllv $t2, $s0, $s1`

MIPS Rotate Instructions

- `rotr (rotate word right); funct = 000 010 (SRL)`

op	rs	rt	rd	shamt	funct
000000	00001				000010

- `rotrv (rotate word right variable)`
  `; funct = 000 110 (SRLV)`

op	rs	rt	rd	shamt	funct
000000				00001	000110

MIPS Logical Instructions

- `and   $t0,$t1,$t2`
- `or    $t0,$t1,$t2`
- `xor   $t0,$t1,$t2`
- `nor   $t0,$t1,$t2`

- `andi $s1,$s1,100`
- `ori  $s1,$s1,100`
- `xori $s1,$s1,100`
- `lui  $s1,100`

2.7 Instructions for Making Decisions

- Similar to an *if* statement with a *go to* statement
  - `beq register1, register2, L1`
  - `bne register1, register2, L1`
- Example

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

[Answer]

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

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

[Answer]

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

slt(set on less than) Instruction

- `slt  $t0,$s3,$s4   # if($s3<$s4) then $t0=1`
  `                   #                  else $t0=0`
- `slti $t0,$s2,10    # if $s2<10 $t0=1; else $t0=0`
- **Pseudo instruction**

  `blt  $s0,$s1,Less   # branch on less than`
- **Why no `blt` instruction in MIPS architecture?**
  - It would stretch the clock cycle time,

    or it would take extra clock cycles per instruction.

Hardware/Software Interface

- All comparisons are possible with slt, slti, beq, bne with $zero.

Other Jump Instructions

- `jr` (jump register) instruction

```
jr  $t0       # jump to the address specified
              # in a register($t0)
              # i.e. PC ← $t0
```

- `jal` (jump-and-link) instruction

```
jal  ProcedureAddress   # $ra ← PC + 4,
                        # PC ← ProcedureAddress
```

- Return from procedure

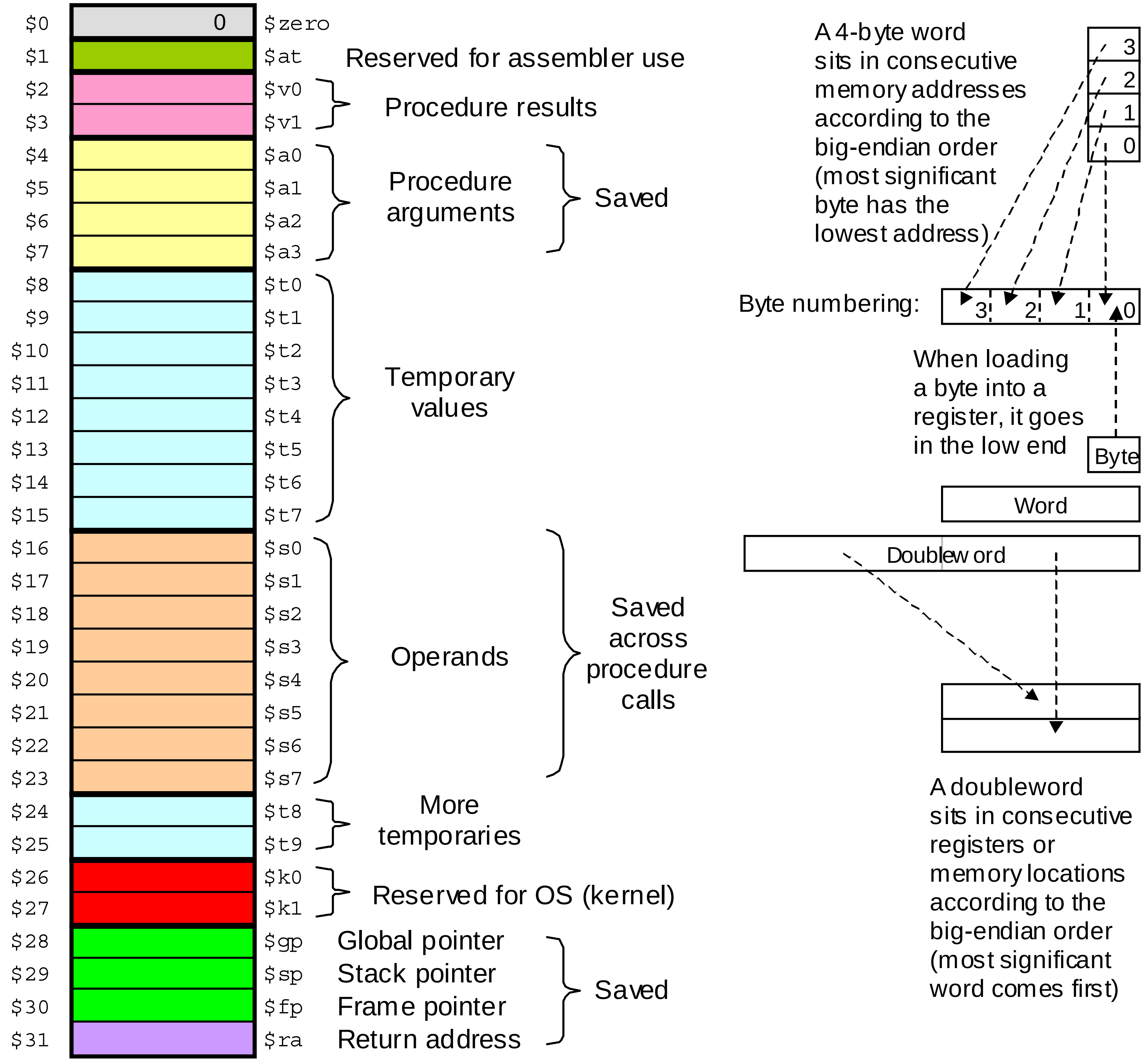
```
jr   $ra
```

MIPS Register Convention

Name	Register Number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for assembler
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t9`	8-15, 24-25	temporaries
`$s0-$s7`	16-23	saved
`$k0-$k1`	26-27	reserved for operating system
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Figure 2.18

MIPS Registers



$0 0 $zero
$1 $at Reserved for assembler use
$2 $v0
$3 $v1 Procedure results
$4 $a0
$5 $a1
$6 $a2
$7 $a3 Procedure arguments Saved
$8 $t0
$9 $t1
$10 $t2
$11 $t3
$12 $t4
$13 $t5
$14 $t6
$15 $t7 Temporary values
$16 $s0
$17 $s1
$18 $s2
$19 $s3
$20 $s4
$21 $s5
$22 $s6
$23 $s7 Operands Saved across procedure calls
$24 $t8
$25 $t9 More temporaries
$26 $k0
$27 $k1 Reserved for OS (kernel)
$28 $gp Global pointer
$29 $sp Stack pointer
$30 $fp Frame pointer
$31 $ra Return address Saved
A 4-byte word sits in consecutive memory addresses according to the big-endian order (most significant byte has the lowest address)
3
2
1
0
Byte numbering: 3 2 1 0
When loading a byte into a register, it goes in the low end
Byte
Word
Doubleword
A doubleword sits in consecutive registers or memory locations according to the big-endian order (most significant word comes first)

Figure 5.1 of Parhami

Summary

- **Instruction formats**
  - R-type and I-type
- **Shift instructions**
  - **`sll, srl, sra, sllv, srlv, srav, rotr, rotrv`**
- **Logical instructions**
  - **`and, or, xor, nor, andi, ori, xori, lui`**
- **Branch and jump instructions**
  - **`beq, bne, j, jal, jr`**
  - **`slt, slti, sltu, sltiu`**
- **Data transfer instructions**
  - **`lw`** and **`sw`**
  - **`lh`** and **`sh`**
  - **`lb`** and **`sb`**