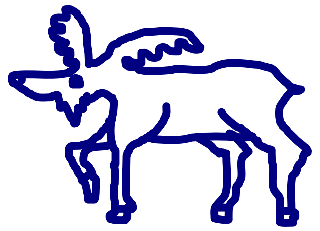
# Lecture 4
# Addressing Modes

Byung-gi Kim

School of Computer Science and Engineering

Soongsil University



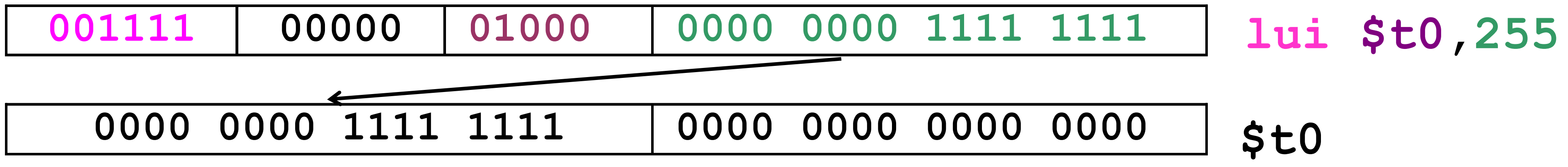
# 2. Instruction: Language of the Computer (3rd edition)

## 2.9 MIPS Addressing for 32-Bit Immediates and Addresses

- **Addressing modes** (cf) Wikipedia
  - defines how machine language instructions in that architecture identify the operand (or operands) of each instruction
  - specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere
- **Number of addressing modes**
  - MIPS: 6 modes
  - ARM: 9 modes
  - IA-32: 12 modes

# 32-Bit Immediate Operands

- Load upper immediate(lui) Instruction



- **Example**

  $t0 <= 0000 0000 1111 1111 0000 1001 0000 0000

  ```
  li $t0, 0b 0000 0000 0011 1101 0000 1001 0000 0000
  ```

  **[Answer]**

  ```
  lui   $t0,0b 0000 0000 1111 1111
  ori   $t0,$t0,0b 0000 1001 0000 0000
  ```

# Addressing in Branches and Jumps

- **J-type instruction format**

| opcode | (word) address |
|---|---|
| 6 bits | 26 bits |

- **Direct addressing mode**
  - `j   10000       # branch address = 10000`
- **Word addressing**
  - Addressing in jump instruction
  - Alignment restriction → instruction address = 4n → LS 2 bits = 00

    | **xxxx xxxx xxxx xxxx xxxx xxxx xxxx xx00** |
    |---|

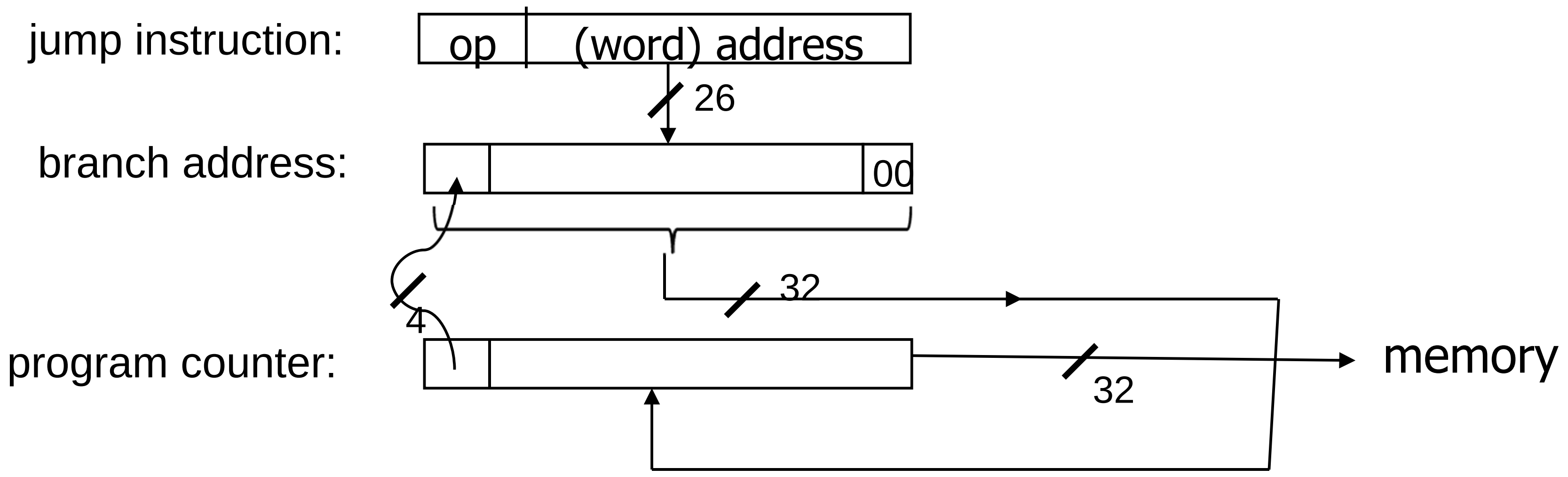
  - Word address
    - Branch address = address * 4
  - `j   10000       # branch address = 40000`

# Pseudo-Direct Addressing Mode

- Upper 4 bits of PC are unchanged.
- Address boundary of $2^{28}$ B = 256 MB
- Jump address

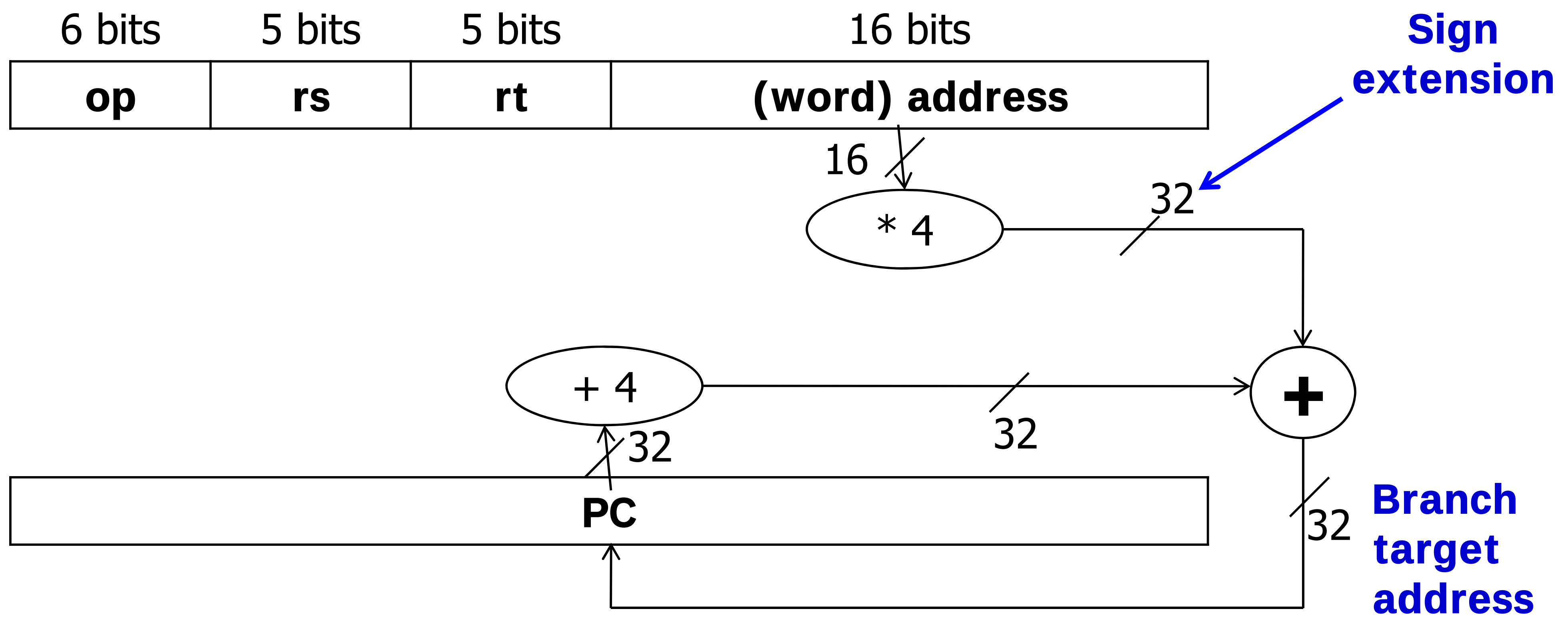
| from PC | from instruction | 00 |
|---|---|---|
| 4 bits | 26 bits | 2 bits |

from the low order 26 bits of the jump instruction



# PC-relative Addressing Mode

- Branch target address = (PC+4) + Branch offset
- New PC ≈ (PC+4) $\pm$ $2^{15}$ words = (PC+4) $\pm$ $2^{17}$ bytes
- PC - 131068 $\leq$ new PC (=branch target address) $\leq$ PC + 131072



## Example: Showing Branch Offset

```
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:
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 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** | | | | |

## Example: Branching Far Away

- `beq $s0,$s1,L1`

  But `L1` is too far.

**[Answer]**

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

# MIPS Addressing Modes

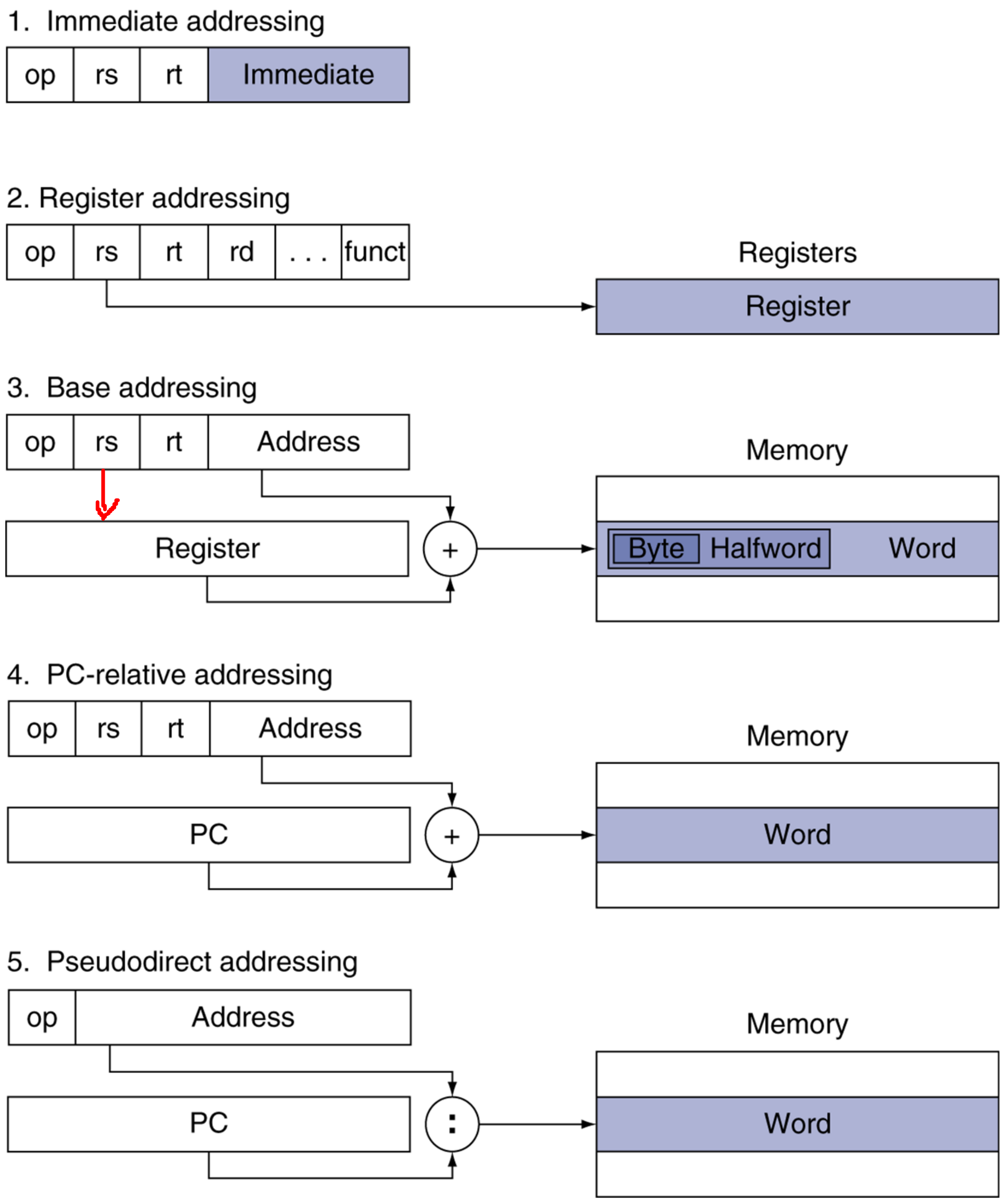


Figure 2.24

## Summary

- **Instruction / Meaning**

| 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 |

- **Instruction formats**

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

# MIPS Instructions

| 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 $s1,$s2,100` | `$s1 = $s2 + 100` | Used to add constants |
| 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 |
| | load half | `lh $s1,100($s2)` | `$s1 = Memory[$s2 + 100]` | Halfword memory to register |
| | store half | `sh $s1,100($s2)` | `Memory[$s2 + 100] = $s1` | Halfword register to memory |
| | load byte | `lb $s1,100($s2)` | `$s1 = Memory[$s2 + 100]` | Byte from memory to register |
| | store byte | `sb $s1,100($s2)` | `Memory[$s2 + 100] = $s1` | Byte from register to memory |
| | load upper immed. | `lui $s1,100` | `$s1` $= 100 * 2^{16}$ | 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 $s1,$s2,100` | `$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 $s1,$s2,25` | 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 $s1,$s2,100` | if (`$s2 < 100`) `$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 |

## Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions
  - Figments of the assembler's imagination
  - \$at (register 1): assembler temporary

```
move $t0, $t1       ⇨   add $t0, $zero, $t1

blt $t0, $t1, L     ⇨   slt $at, $t0, $t1
                        bne $at, $zero, L
```

# Decoding Machine Language

- **Example**

  $00AF8020_{hex}$

**[Answer]**

0000 0000 1010 1111 1000 0000 0010 0000

Opcode=000000 -> R-type

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

Funct = 100000 -> `add` (See Fig. 2.25)

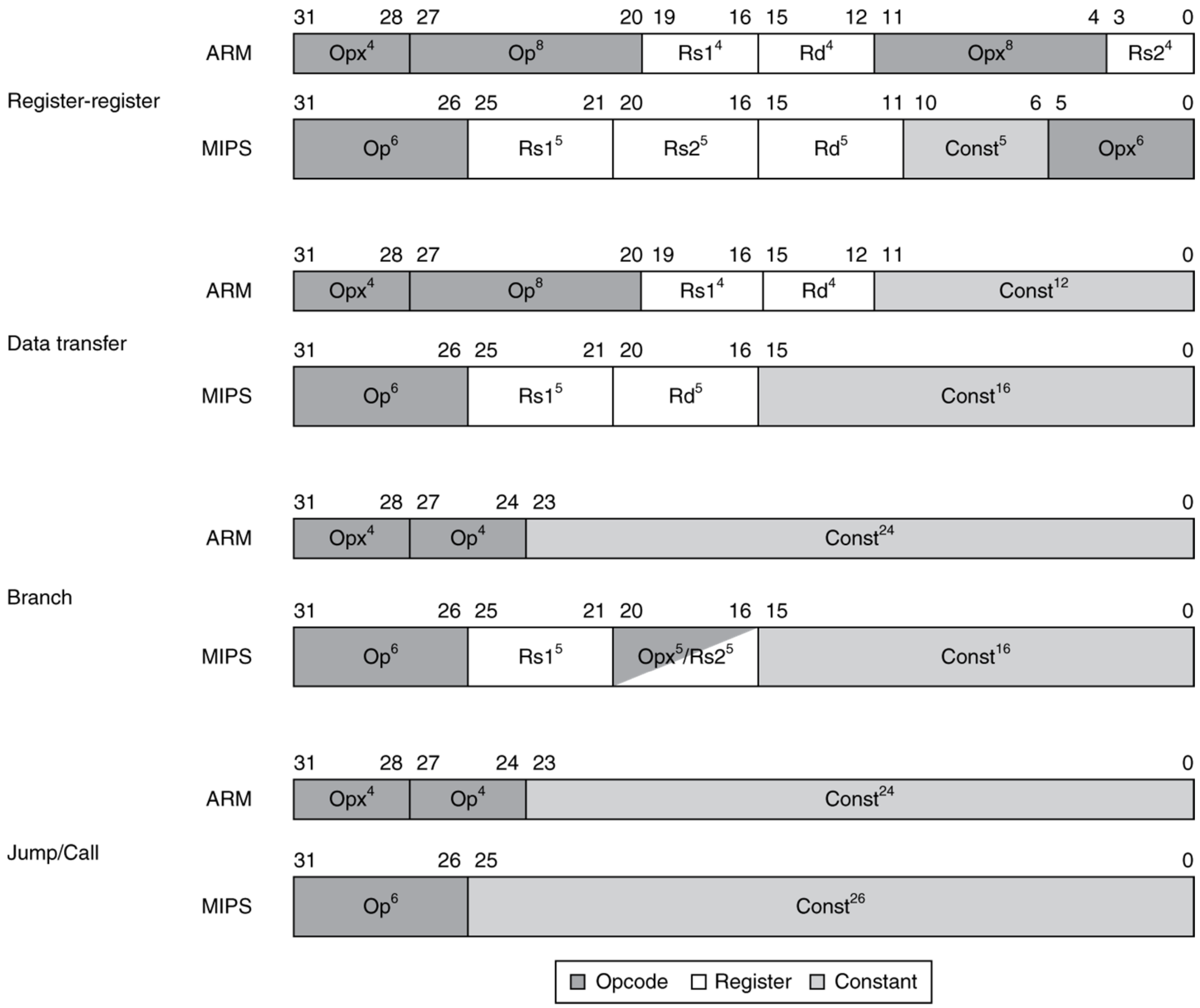
`$5 = $a1, $15 = $t7, $16 = $s0` (See Fig. 2.18)

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

# ARM & MIPS Similarities

| | ARM | MIPS |
|---|---|---|
| Date announced | 1985 | 1985 |
| Instruction size | 32 bits | 32 bits |
| Address space | 32-bit flat | 32-bit flat |
| Data alignment | Aligned | Aligned |
| Data addressing modes | 9 | 3 |
| Registers | 15 × 32-bit | 31 × 32-bit |
| Input/output | Memory mapped | Memory mapped |

# Instruction Encoding



# 2.18 Concluding Remarks

1. **Simplicity favors regularity.**
   - fixed size instructions
   - small number of instruction formats
   - opcode always the first 6 bits
2. **Smaller is faster.**
   - limited instruction set
   - limited number of registers in register file
   - limited number of addressing modes
3. **Make the common case fast.**
   - arithmetic operands from the register file (load-store machine)
   - allow instructions to contain immediate operands
4. **Good design demands good compromises.**
   - three instruction formats

# Instruction Categories

| Instruction class | MIPS example | Frequency | |
|---|---|---|---|
| | | Integer | FP |
| **Arithmetic** | `add,sub,addi` | 24% | 48% |
| **Data transfer** | `lw,sw,lb,sb,lui` | 36% | 39% |
| **Logical** | `and,or,nor,andi,` `ori,sll,srl` | 18% | 4% |
| **Conditional branch** | `beq,bne,slt,slti` | 18% | 6% |
| **Jump** | `j,jr,jal` | 3% | 0% |

Figure 2.48

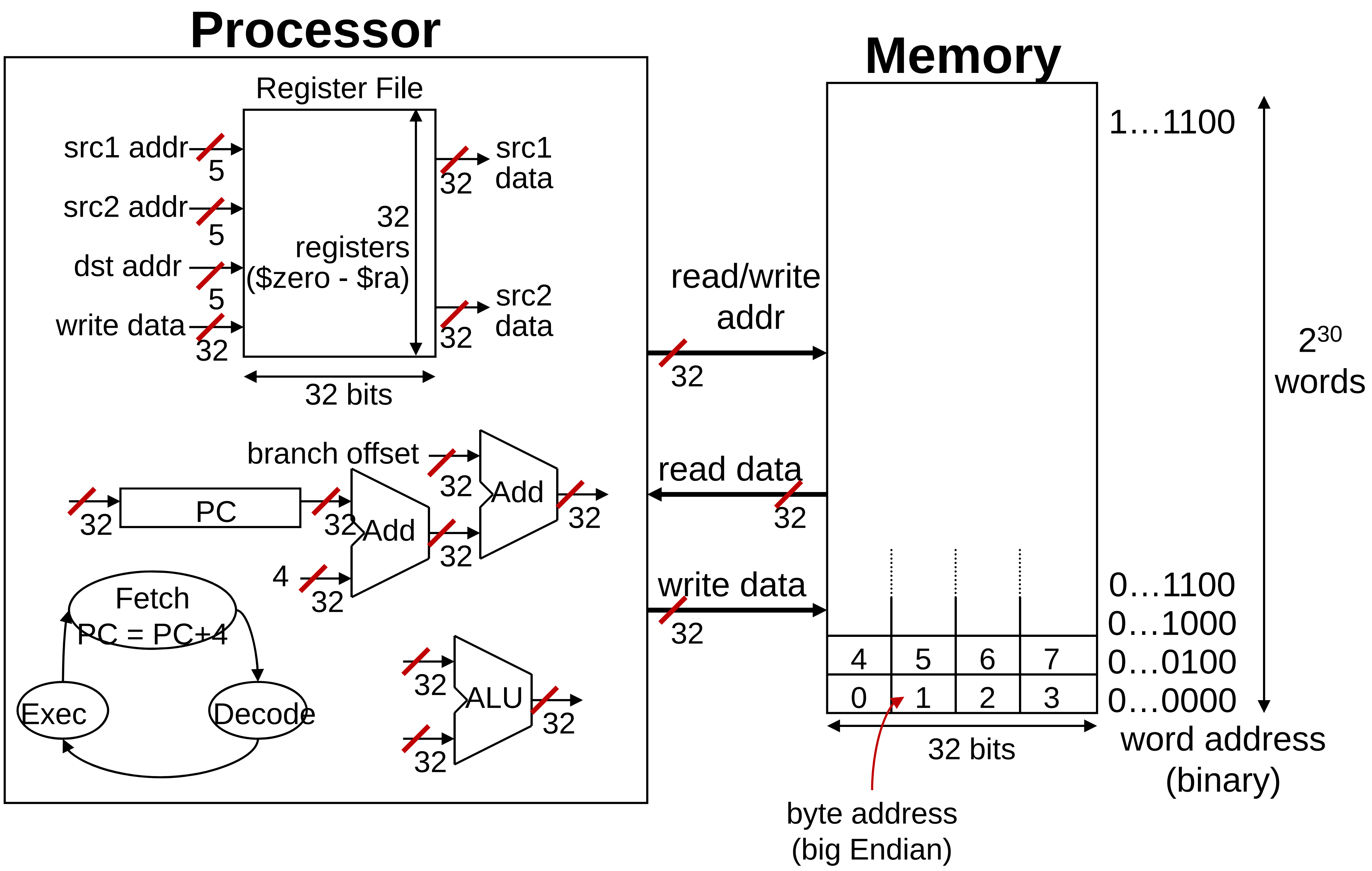
## MIPS Instructions

| MIPS instruction | Name | Format |
|---|---|---|
| add | add | R |
| subtract | sub | R |
| add immediate | addi | I |
| load word | lw | I |
| store word | sw | I |
| load half | lh | I |
| store half | sh | I |
| load byte | lb | I |
| store byte | sb | I |
| load upper immediate | lui | I |
| and | and | R |
| or | or | R |
| nor | nor | R |
| and immediate | andi | I |
| or immediate | ori | I |
| shift left logical | sll | R |
| shift right logical | srl | R |
| branch on equal | beq | I |
| branch on not equal | bne | I |
| set on less than | slt | R |
| set on less than immediate | slti | I |
| jump | j | J |
| jump register | jr | R |
| jump and link | jal | J |

| Pseudo MIPS | Name | Format |
|---|---|---|
| move | move | R |
| multiply | mult | R |
| multiply immediate | multi | I |
| load immediate | li | I |
| branch on less than | blt | I |
| branch on less than or equal to | ble | I |
| branch on greater than | bgt | I |
| branch on greater than or equal to | bge | I |

Figure 2.47

# MIPS Organization So Far



Processor
Register File
src1 addr
5
src2 addr
5
dst addr
5
write data
32
32 registers ($zero - $ra)
src1 data
32
src2 data
32
32 bits
branch offset
32
Add
32
PC
32
32
Add
4
32
32
Fetch PC = PC+4
Decode
Exec
32
32
ALU
32
read/write addr
32
read data
32
write data
32
Memory
1…1100
$2^{30}$ words
0…1100
0…1000
0…0100
0…0000
4
5
6
7
0
1
2
3
32 bits
word address (binary)
byte address (big Endian)