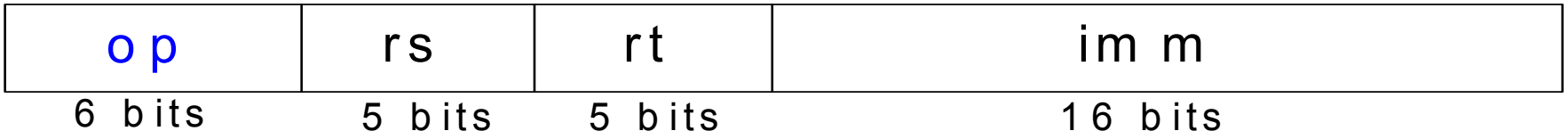
# COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Lecture 9

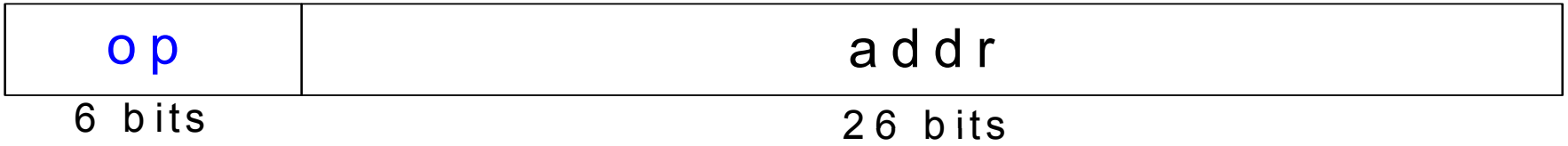
# Instruction Formats

## R-Type


## I-Type



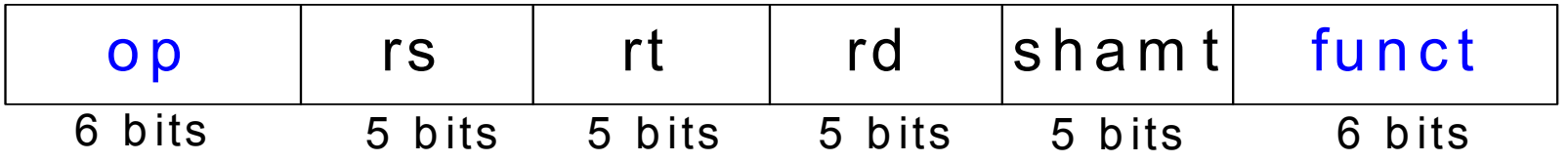

## J-Type



# R-Type

- *Register-type*
- 3 register operands:
  - rs, rt: source registers
  - rd: destination register
- Other fields:
  - op: the *operation code* or *opcode* (0 for R-type instructions)
  - funct: the *function*
    together, the opcode and function tell the computer what operation to be performed
  - shamt: the *shift amount* for shift instructions, otherwise it's 0

**R-Type**



# R-Type Examples

## Assembly Code

```
ld $s0, $s1, $s2

b $t0, $t3, $t5
```

## Field Values

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Order of registers in the assembly code:

add rd, rs, rt

Instructions are 32 bits long

registers have numbers 0 .. 31,

e.g., $t0=8, $t1=9, $s0=16, $s1=17 etc.

# R-Type Examples

## Assembly Code

## Field Values

| Assembly Code | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| ld $s0, $s1, $s2 | 0 | 17 | 18 | 16 | 0 | 32 |
| b $t0, $t3, $t5 | 0 | 11 | 13 | 8 | 0 | 34 |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Order of registers in the assembly code:

add rd, rs, rt

Instructions are 32 bits long

registers have numbers 0 .. 31,

e.g., $t0=8, $t1=9, $s0=16, $s1=17 etc.

# -Type Examples

## Assembly Code

```
dd  $s0,  $s1,  $s2
ub  $t0,  $t3,  $t5
```

## Field Values

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 16 | 0 | 32 |
| 0 | 11 | 13 | 8 | 0 | 34 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op | rs | rt | rd | shamt | funct | |
|---|---|---|---|---|---|---|
| 000000 | 10001 | 10010 | 10000 | 00000 | 100000 | (0x02328020) |
| 000000 | 01011 | 01101 | 01000 | 00000 | 100010 | (0x016D4022) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | |

Order of registers in the assembly code:

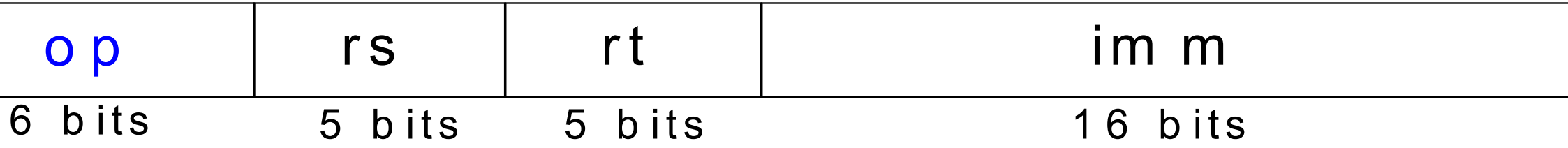
add rd, rs, rt

# I-Type

Immediate-type

- 3 operands:
  - rs, rt: register operands
  - imm: 16-bit two's complement immediate
- Other fields:
  - op: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

**I-Type**



# -Type Examples

| Assembly Code | op | rs | rt | imm |
|---|---|---|---|---|
| `addi $s0, $s1, 5` | 8 | 17 | 16 | 5 |
| `addi $t0, $s3, -12` | 8 | 19 | 8 | -12 |
| `lw $t2, 32($0)` | 35 | 0 | 10 | 32 |
| `sw $s1, 4($t1)` | 43 | 9 | 17 | 4 |
| | 6 bits | 5 bits | 5 bits | 16 bits |

Field Values

Differing order of registers in the assembly and machine codes:

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

# -Type Examples

## Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

## Field Values

| op | rs | rt | imm |
|---|---|---|---|
| 8 | 17 | 16 | 5 |
| 8 | 19 | 8 | -12 |
| 35 | 0 | 10 | 32 |
| 43 | 9 | 17 | 4 |
| 6 bits | 5 bits | 5 bits | 16 bits |

fering order of registers in the
sembly and machine codes:

di rt, rs, imm

rt, imm(rs)

rt, imm(rs)

## Machine Code

| op | rs | rt | imm | |
|---|---|---|---|---|
| 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x2230000 |
| 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF |
| 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A002 |
| 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD3100( |
| 6 bits | 5 bits | 5 bits | 16 bits | |

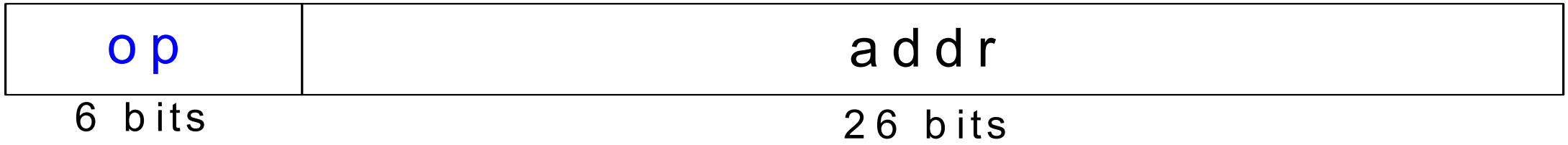
2's complement of (-12)=1111 1111 1111 0100

# Machine Language: J-Type

*Jump-type*

- 26-bit address operand (addr)
- Used for jump instructions (j)

**J-Type**



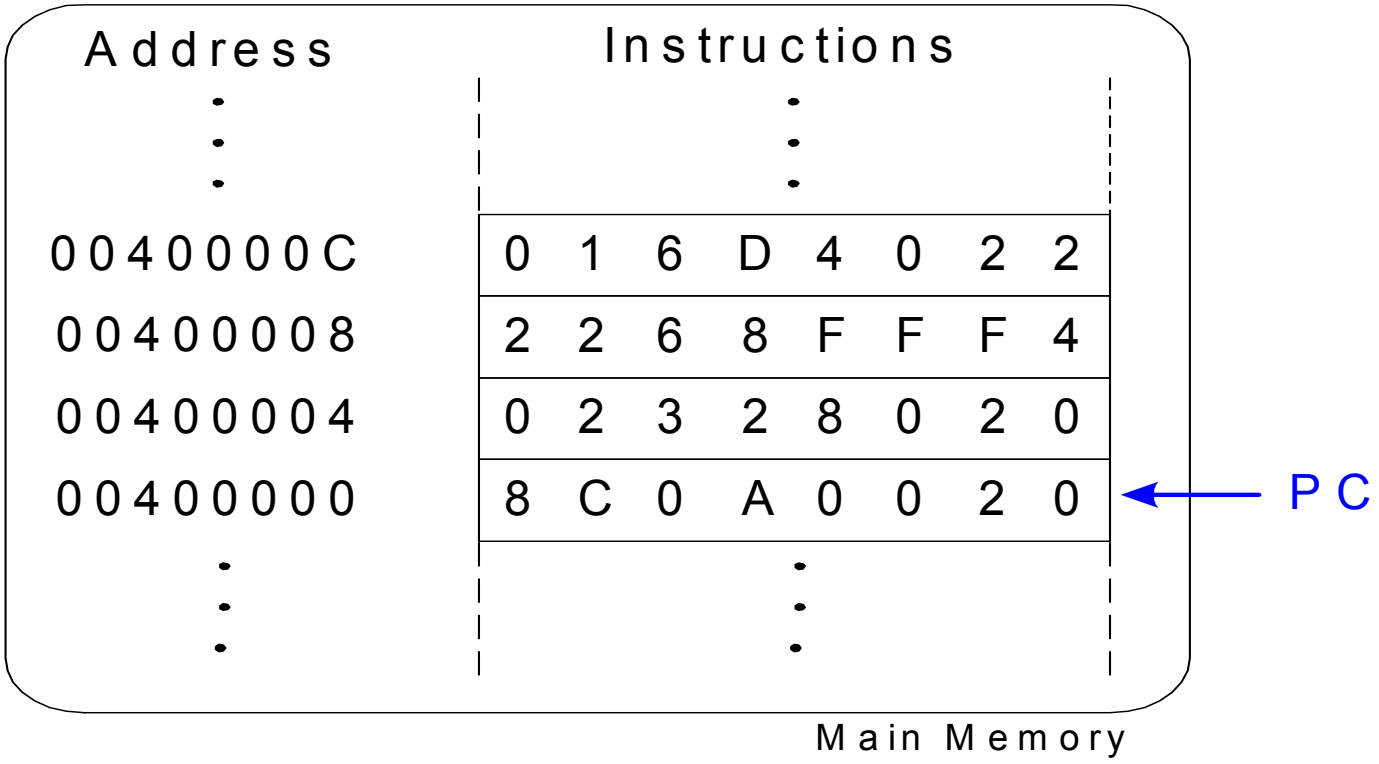
# Power of Stored Program

- 32-bit instructions and data stored in memory
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Processor hardware executes the program:
  - *fetches* (reads) the instructions from memory in sequence
  - performs the specified operation
- Program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000

# Stored Program

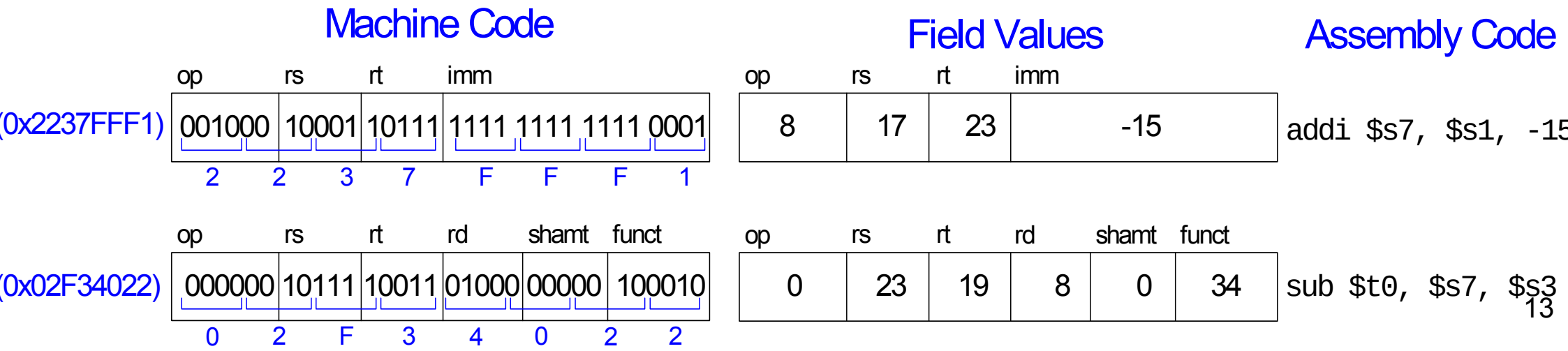
| Assembly Code | Machine Code |
|---|---|
| `lw $t2, 32($0)` | `0x8C0A0020` |
| `add $s0, $s1, $s2` | `0x02328020` |
| `addi $t0, $s3, -12` | `0x2268FFF4` |
| `sub $t0, $t3, $t5` | `0x016D4022` |

Stored Program



# nterpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is



# Logical Instructions

- **and, or, xor, nor**
  - **and: useful for masking bits**
    - **Masking all but the least significant byte of a value:**

      **0xF234012F AND 0x000000FF = 0x0000002F**
  - **or: useful for combining bit fields**
    - **Combine 0xF2340000 with 0x000012BC:**

      **0xF2340000 OR 0x000012BC = 0xF23412BC**
  - **nor: useful for inverting bits:**
    - **A NOR $0 = NOT A**
- **andi, ori, xori**
  - **16-bit immediate is zero-extended (not sign-extended)**
  - **nori not needed**

# Logical Instruction Examples

Source Registers

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| $s2 | 0100 | 0110 | 1010 | 0001 | 1111 | 0000 | 1011 | 0111 |

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s3 | | | | | | | | |
| $s4 | | | | | | | | |
| $s5 | | | | | | | | |
| $s6 | | | | | | | | |

# ogical Instruction Examples

Source Registers

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| $s2 | 0100 | 0110 | 1010 | 0001 | 1111 | 0000 | 1011 | 0111 |

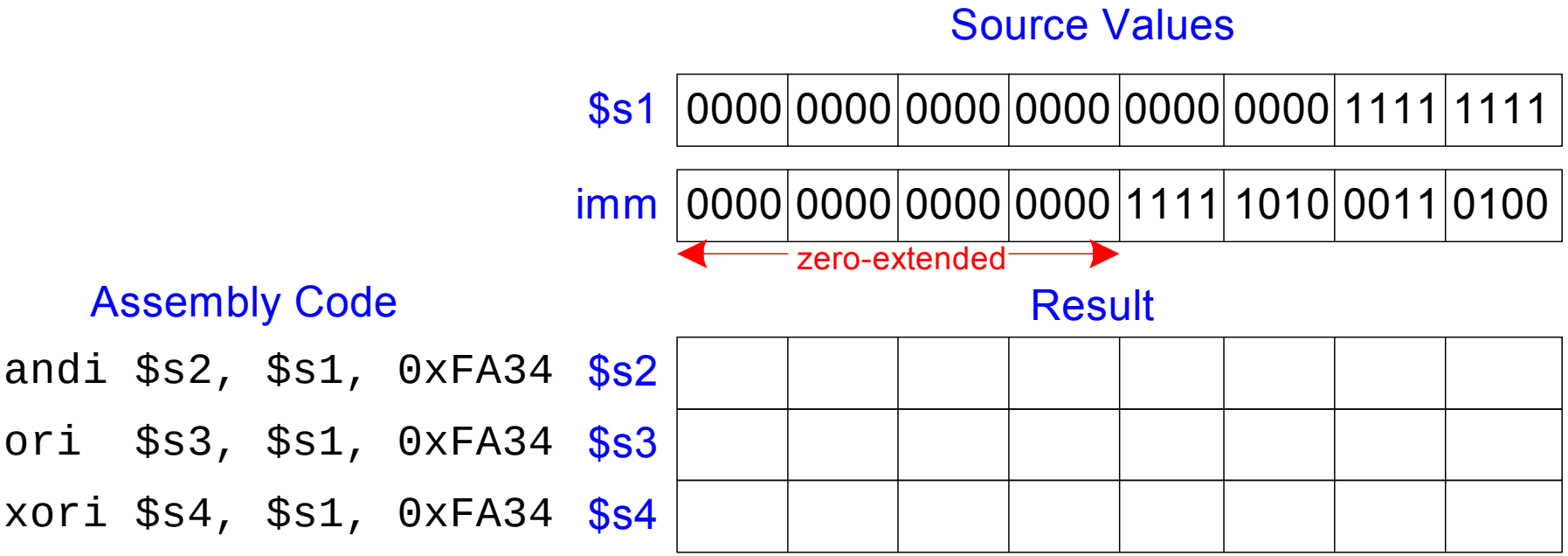
Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

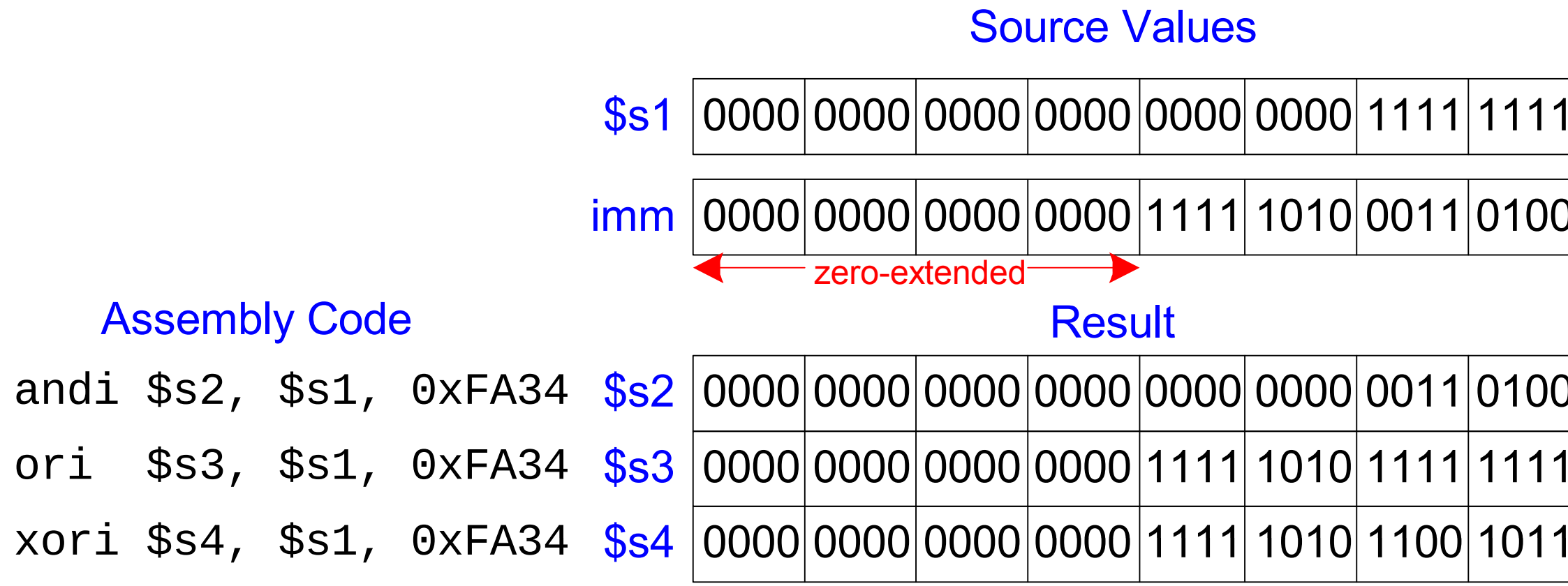
Result

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s3 | 0100 | 0110 | 1010 | 0001 | 0000 | 0000 | 0000 | 0000 |
| $s4 | 1111 | 1111 | 1111 | 1111 | 1111 | 0000 | 1011 | 0111 |
| $s5 | 1011 | 1001 | 0101 | 1110 | 1111 | 0000 | 1011 | 0111 |
| $s6 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 0100 | 1000 |

# Logical Instruction Examples



# Logical Instruction Examples



# Shift Instructions

- sll: shift left logical
  - **Example:** sll $t0, $t1, 5 # $t0 <= $t1 << 5
- srl: shift right logical
  - **Example:** srl $t0, $t1, 5 # $t0 <= $t1 >> 5
- sra: shift right arithmetic
  - **Example:** sra $t0, $t1, 5 # $t0 <= $t1 >>> 5

Variable shift instructions:

- sllv: shift left logical variable
  - **Example:** sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2
- srlv: shift right logical variable
  - **Example:** srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2
- srav: shift right arithmetic variable
  - **Example**: srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2

# Shift Instructions

## Assembly Code

## Field Values

| | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| `sll $t0, $s1, 2` | 0 | 0 | 17 | 8 | 2 | 0 |
| `srl $s2, $s1, 2` | 0 | 0 | 17 | 18 | 2 | 2 |
| `sra $s3, $s1, 2` | 0 | 0 | 17 | 19 | 2 | 3 |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op | rs | rt | rd | shamt | funct | |
|---|---|---|---|---|---|---|
| 000000 | 00000 | 10001 | 01000 | 00010 | 000000 | (0x00114080) |
| 000000 | 00000 | 10001 | 10010 | 00010 | 000010 | (0x00119082) |
| 000000 | 00000 | 10001 | 10011 | 00010 | 000011 | (0x00119883) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | |

# Shift Instructions

Source Values

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 | 1000 |

| | |
|---|---|
| shamt | 00100 |

Assembly Code

```
ll $t0, $s1, 4
rl $s2, $s1, 4
ra $s3, $s1, 4
```

Result

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $t0 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 | 1000 | 0000 |
| $s2 | 0000 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 |
| $s3 | 1111 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 |

# Shift Instructions

## Assembly Code

```
sllv  $s3,  $s1,  $s2
srlv  $s4,  $s1,  $s2
srav  $s5,  $s1,  $s2
```

## Field Values

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 18 | 17 | 19 | 0 | 4 |
| 0 | 18 | 17 | 20 | 0 | 6 |
| 0 | 18 | 17 | 21 | 0 | 7 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op | rs | rt | rd | shamt | funct | |
|---|---|---|---|---|---|---|
| 000000 | 10010 | 10001 | 10011 | 00000 | 000100 | (0x02519804) |
| 000000 | 10010 | 10001 | 10100 | 00000 | 000110 | (0x0251A006) |
| 000000 | 10010 | 10001 | 10101 | 00000 | 000111 | (0x0251A807) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | |

# Shift Instructions

Source Values

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 | 1010 | 1000 |
| $s2 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | **1000** |

Assembly Code

```
sllv $s3, $s1, $s2
srlv $s4, $s1, $s2
srav $s5, $s1, $s2
```

Result

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s3 | 0000 | 0100 | 0000 | 0010 | 1010 | 1000 | 0000 | 0000 |
| $s4 | 0000 | 0000 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 |
| $s5 | 1111 | 1111 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 |

# Generating Constants

- 16-bit constants using addi:

**High-level code**

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

**MIPS assembly code**

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (lui) and ori:

  (lui loads the 16-bit immediate into the upper half of the register and set the lower half to 0.)

**High-level code**

```
int a = 0xFEDC8765;
```

**MIPS assembly code**

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

# Multiplication, Division

- Special registers: lo, hi
- 32 × 32 multiplication, 64 bit result
  - mult $s0, $s1
  - Result in {hi, lo}

- 32-bit division, 32-bit quotient, 32-bit remainder
  - div $s0, $s1
  - Quotient in lo
  - Remainder in hi

- Moves from lo/hi special registers
  - mflo $s2
  - mfhi $s3

# Branching

- Allows a program to execute instructions out of sequence.

- Types of branches:
  - Conditional branches
    - branch if equal (beq)
    - branch if not equal (bne)

  - Unconditional branches
    - jump (j)
    - jump register (jr)
    - jump and link (jal)

# Conditional Branching (beq)

```
# MIPS assembly
addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
                                                  {0000 0000 0000 0001
beq  $s0, $s1, target       # branch is taken     0000 0000 0000 0100}
addi $s1, $s1, 1            # not executed
sub  $s1, $s1, $s0          # not executed

target:                     # label
add  $s1, $s1, $s0          # $s1 = 4 + 4 = 8
```

**Labels** indicate instruction locations in a program. They cannot use reserve words and must be followed by a colon (:).

# Branch Not Taken (bne)

```
# MIPS assembly
  addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne    $s0, $s1, target    # branch not taken
  addi   $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub    $s1, $s1, $s0       # $s1 = 5 – 4 = 1

target:
  add    $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

# Unconditional Branching / Jumping (j)

```
# MIPS assembly
  addi $s0, $0, 4              # $s0 = 4
  addi $s1, $0, 1              # $s1 = 1
  j        target              # jump to target
  sra      $s1, $s1, 2         # not executed
  addi     $s1, $s1, 1         # not executed
  sub      $s1, $s1, $s0       # not executed

  target:
  add      $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

# Unconditional Branching (jr)

**# MIPS assembly**

**0x00002000** addi $s0, $0, 0x2010

**0x00002004** jr $s0

**0x00002008** addi $s1, $0, 1

**0x0000200C** sra $s1, $s1, 2

**0x00002010** lw $s3, 44($s1)

# Unconditional Branching (jr)

**# MIPS assembly**

**0x00002000** addi $s0, $0, 0x2010

**0x00002004** jr $s0

**0x00002008** addi $s1, $0, 1

**0x0000200C** sra $s1, $s1, 2

**0x00002010** lw $s3, 44($s1)

# Conditional Statement

- **if statements**
- **if/else statements**
- **while loops**
- **for loops**

# If Statement

**High-level code**

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

f = f – i;
```

**MIPS assembly code**

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

bne $s3, $s4, L1
 add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

# If / Else Statement

High-level code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

# For Loops

General form of a ‘for loop’ is:

for (initialization; condition; loop operation)

loop body

- initialization: executes before the loop begins
- condition: is tested at the beginning of each iteration
- loop operation: executes at the end of each iteration
- loop body: executes each time the condition is met

# For Loops

## High-level code

```
// add the numbers from 0 to 9
  int sum = 0;
  int i;

  for (i=0; i!=10; i = i+1)
{
  sum = sum + i;
}
```

## MIPS assembly code

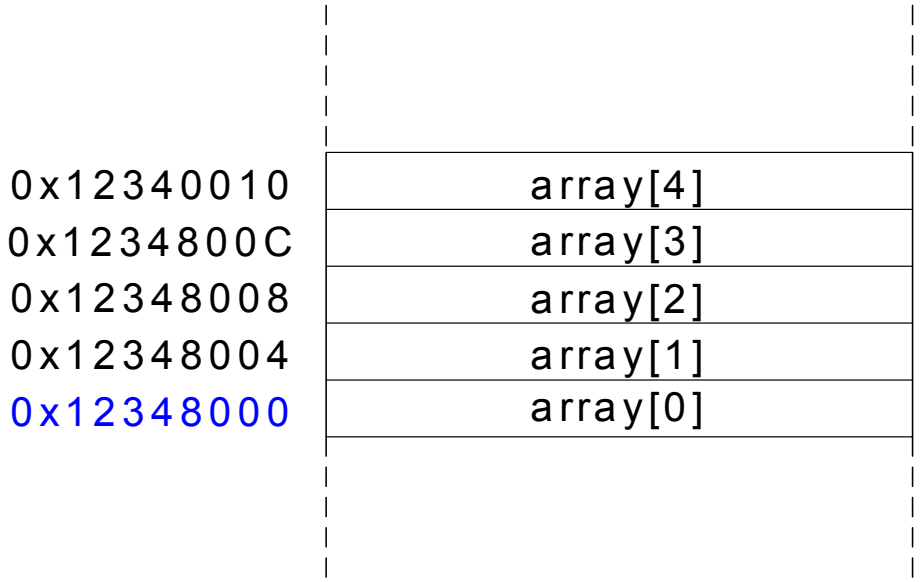
```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

# Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by *index*
- Array *size:* number of elements in the array

# Arrays

- 5-element array
- Base address = 0x12348000 (address of the first array element, array[0])
- First step in accessing an array: load base address into a register



# ccessing Arrays

**High-Level Code**

nt array [5];

rray [0]=array[0] * 8;

rray[1]=array[1]*8;

**MIPS Assembly Code**

```
# $s0=base address of array
lui $s0, 0x1000   #s0=0x1000 0000
ori $s0, $s0, 0x7000  #$s0=0x1000 7000

lw $t1, 0($s0)          #$t1=array[0];
sll $t1,$t1,3           #$t1=$t1<<3=$t1*8
sw $t1,0($s0)          #array[0]=$t1

lw  $t1, 4($s0)        #$t1=array[1]
sll $t1,$t1,3          #$t1=$t1<<3=$t1*8
sw $t1,4($s0)          # array[1]=$t1
```

# Manipulation of Characters of Data

load byte and store byte instructions to manipulate bytes or characters of data

load byte unsigned (lbu)

load byte (lb)

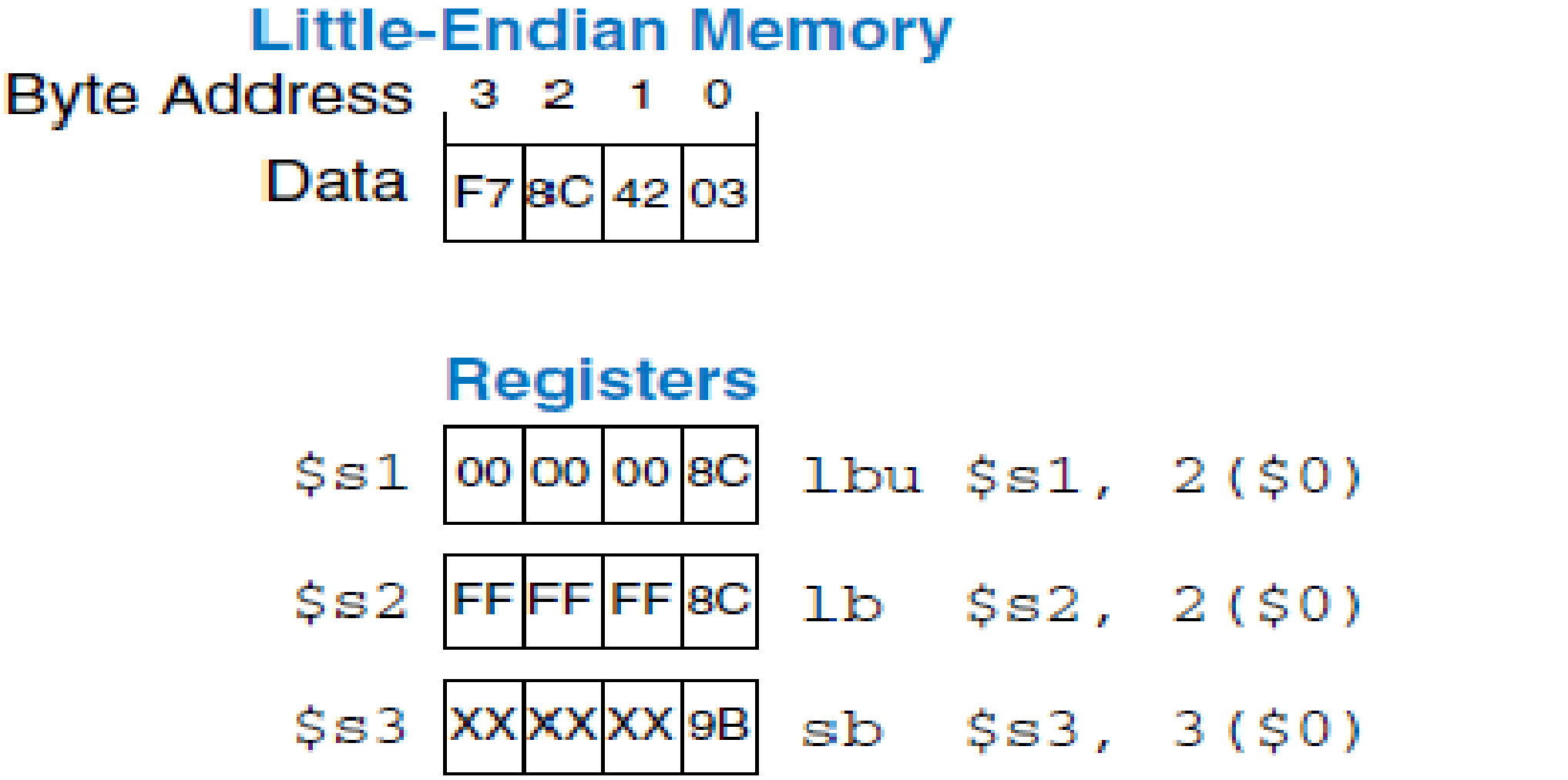
store byte (sb)

Load byte unsigned (lbu) zero-extends the byte

load byte (lb) sign-extends the byte to fill the entire 32-bit register

Store byte (sb) stores the least significant byte of the 32-bit register into the specified byte address in memory

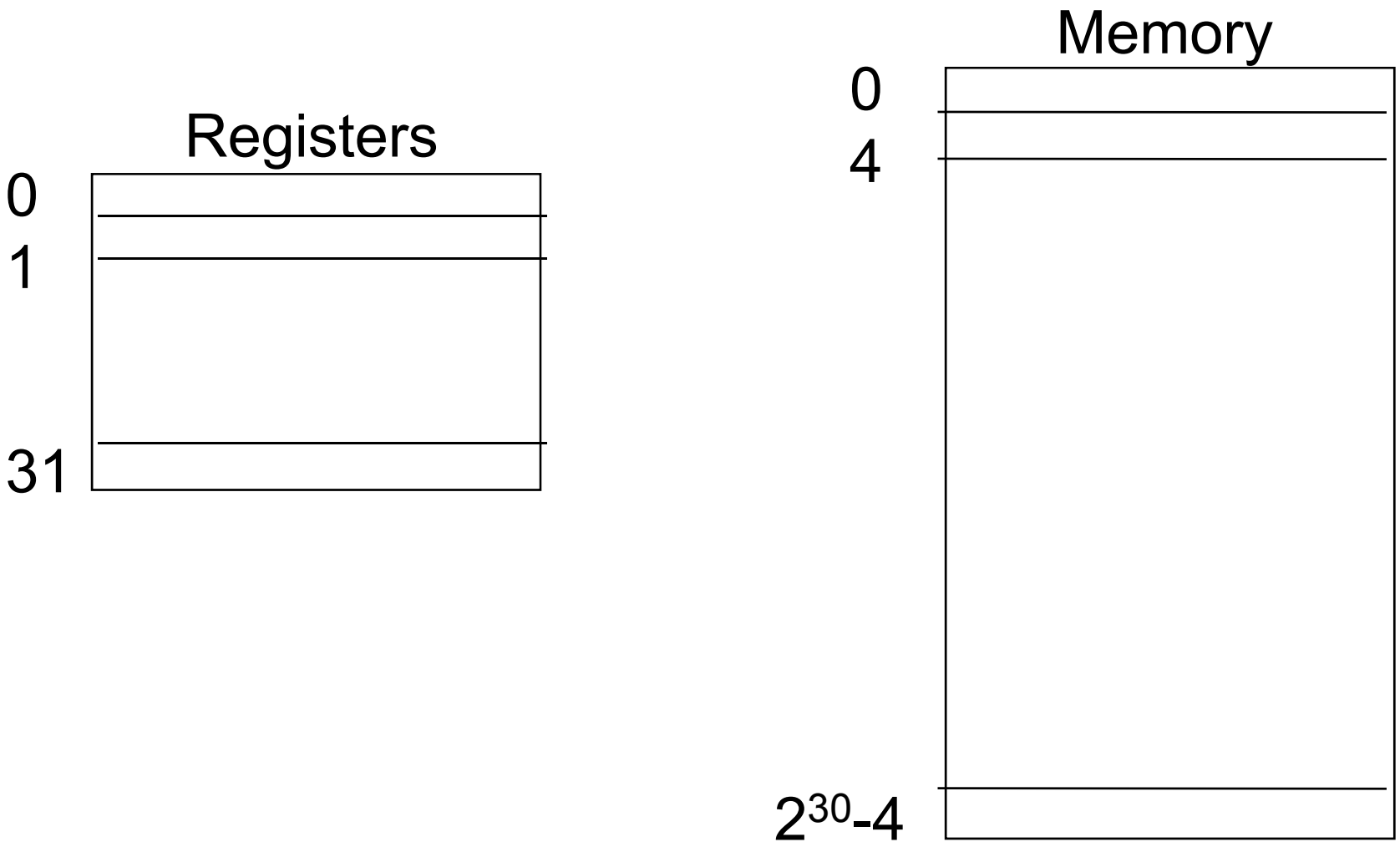
# Manipulation of Characters of Data




3; It replaces 0xF7 with 0x9B

| 9B | 8C | 42 | 03 |
|---|---|---|---|

# MIPS Storage



# Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

# Addressing Modes

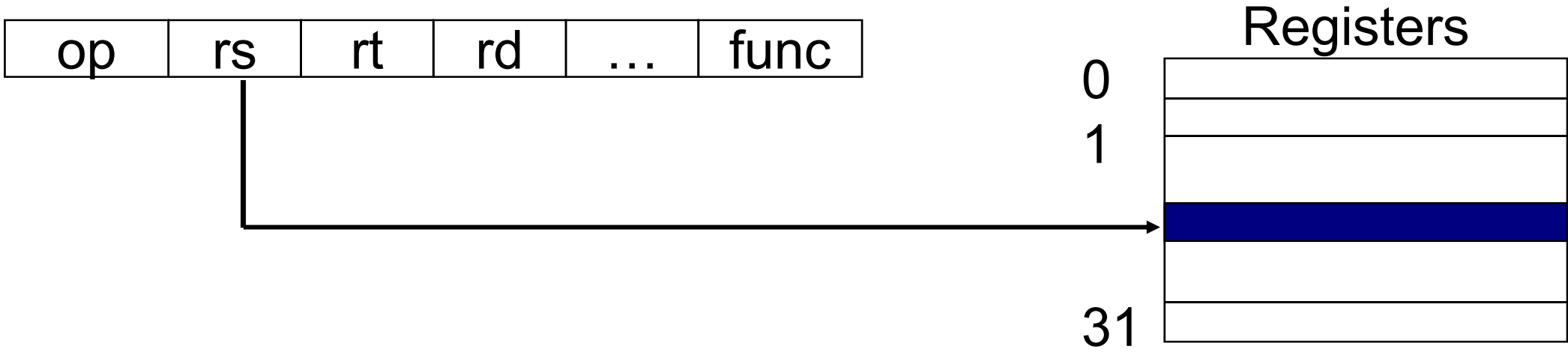
Immediate addressing


16-bit immediate (constant) used as an operand

- Example: addi $s4, $t5, -73
- Example: ori $t3, $t7, 0xFF

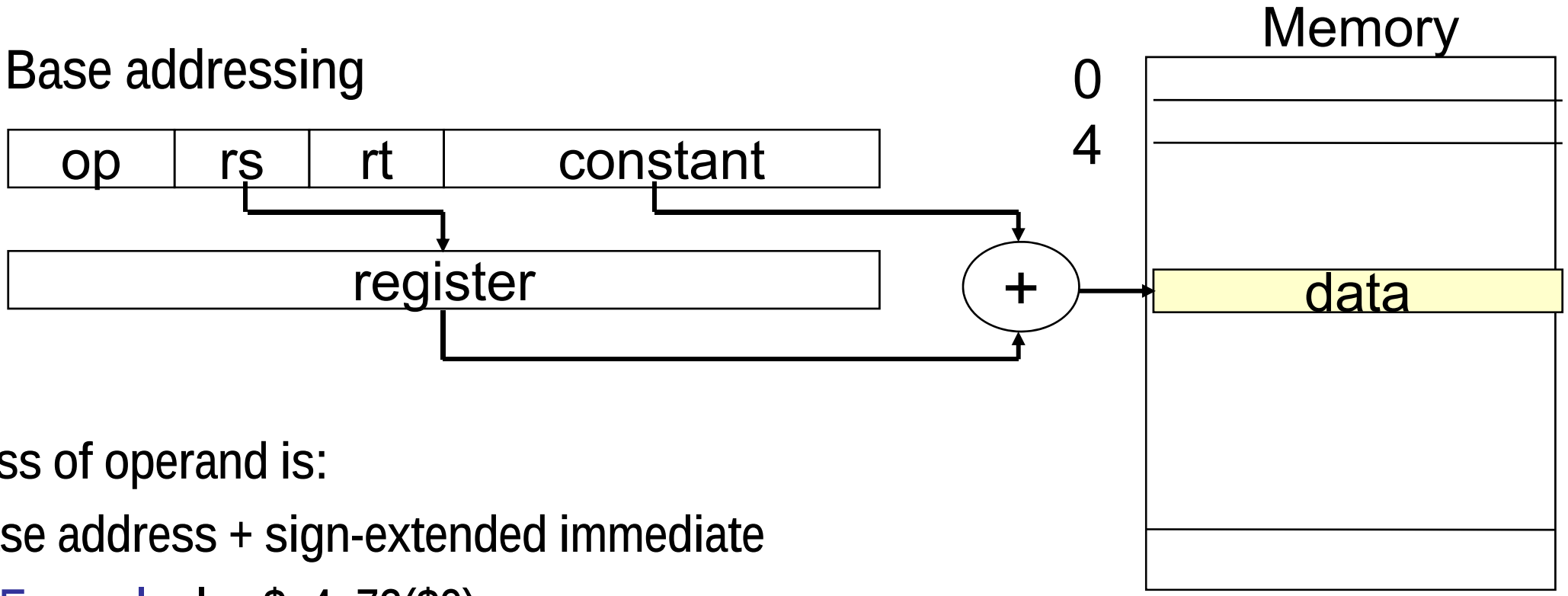
# Addressing Modes

Register addressing




- **Operands found in registers**
  - **Example: add $s0, $t2, $t3**
  - **Example: sub $t8, $s1, $0**

# Addressing Modes

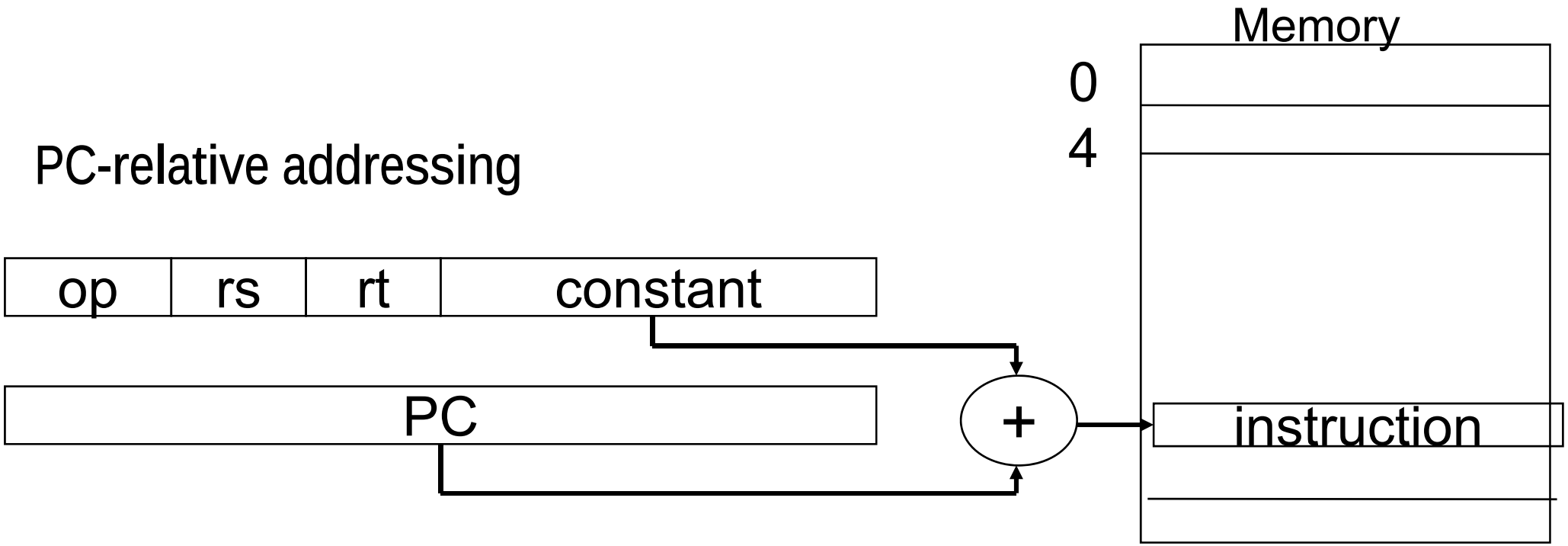



ddress of operand is:

base address + sign-extended immediate

- Example: lw $s4, 72($0)
  - Address = $0 + 72
- Example: sw $t2, -25($t1)
  - Address = $t1 - 25

# Addressing Modes



# Addressing Modes

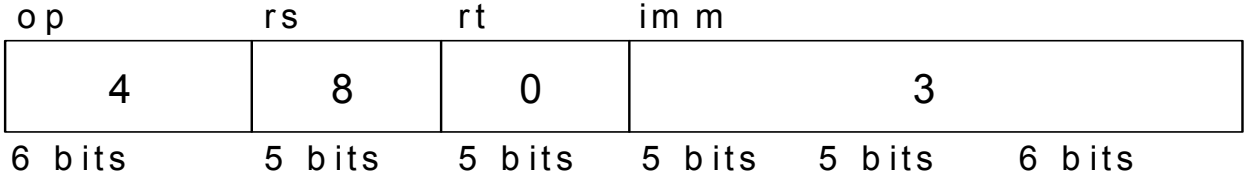
## PC-Relative Addressing

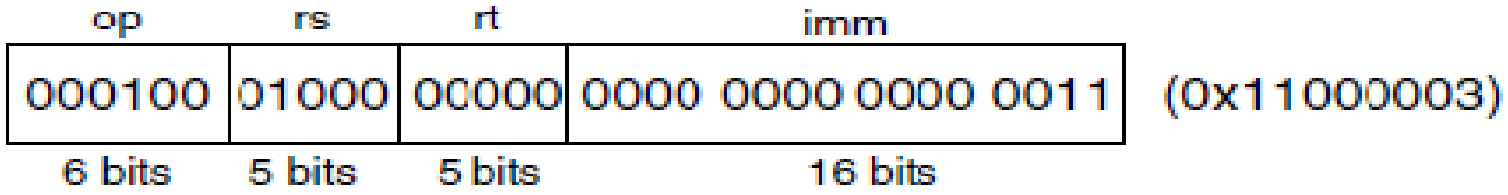
```
0x10          beq   $t0, $0, else
0x14          addi  $v0, $0, 1
0x18          addi  $sp, $sp, i
0x1C          jr    $ra
0x20    else: addi  $a0, $a0, -1
0x24          jal   factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Field Values

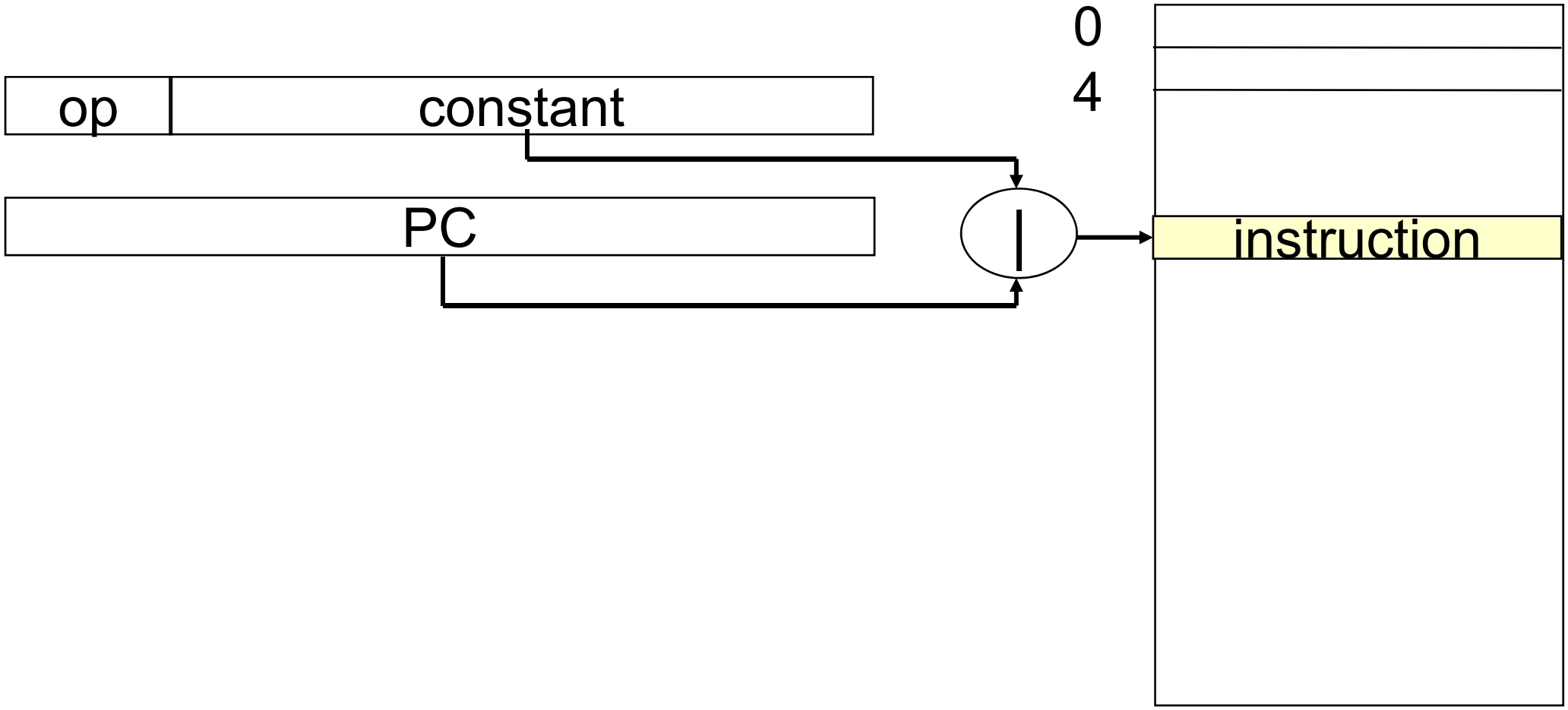



Address of else=PC+4+(4*3)=0x14+0x12=0x20

# Addressing Modes

(Pseudo) Direct addressing



# Addressing Modes

## Pseudo-direct Addressing

(jump target address=JTA)

0x0040005C jal sum

...

0x004000A0 sum: add $v0, $a0, $a1

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

Field Values

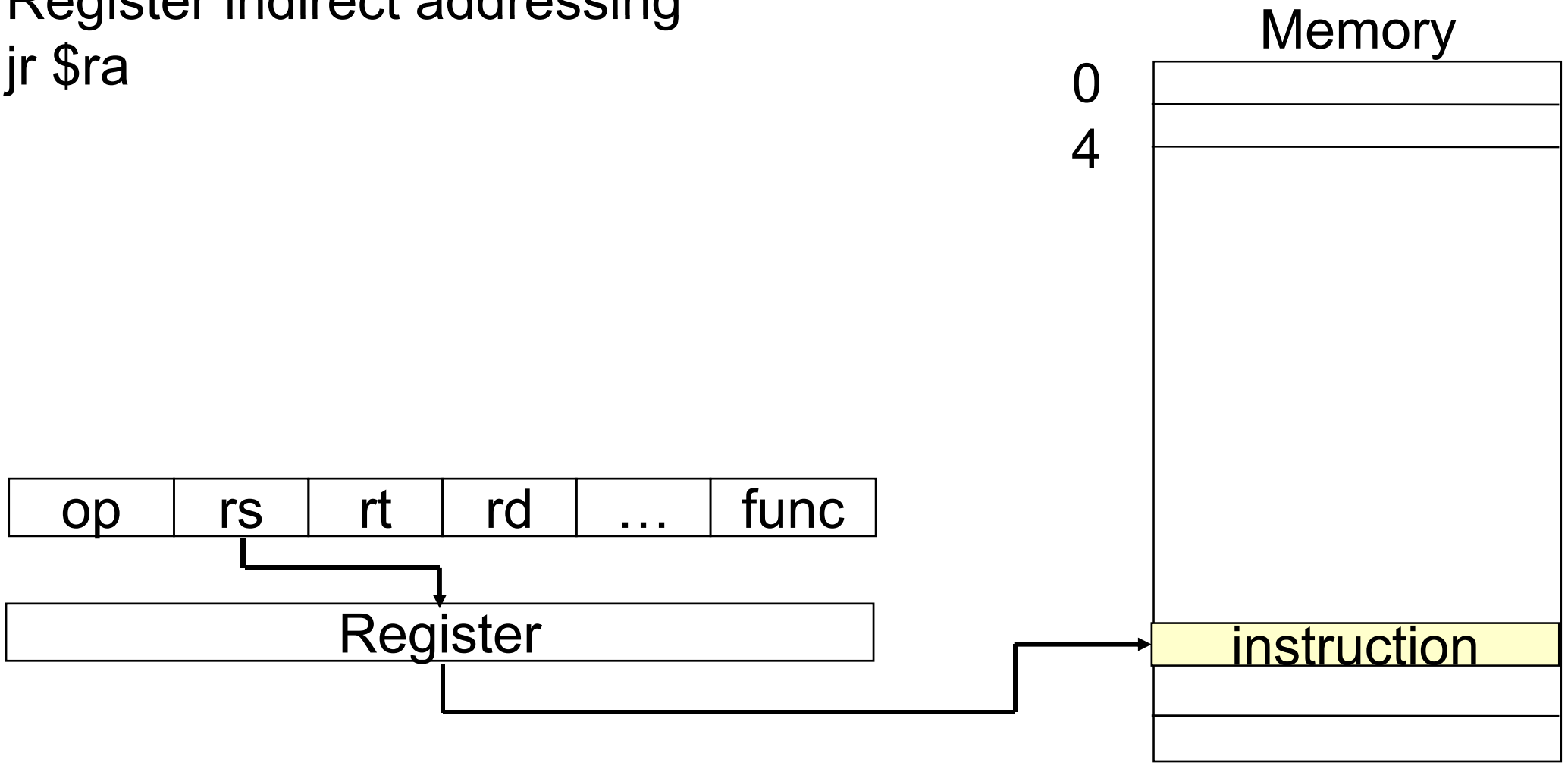
| op | imm |
|---|---|
| 3 | 0x0100028 |
| 6 bits | 26 bits |

Machine Code

| op | addr | |
|---|---|---|
| 000011 | 00 0001 0000 0000 0000 0010 1000 | (0x0C100028) |
| 6 bits | 26 bits | |

# Addressing Modes

Register indirect addressing
jr $ra



# CPU Performance

# CPU Performance: Introduction

- Usually, processor execute instructions in synchronous manner using a clock that runs at a constant clock rate or frequency *f*.
- Clock cycle time is the reciprocal of clock rate *f*. Clock cycle time is often termed as clock period, which is the reciprocal of frequency, that is *1/f*.
- Clock rate *f* depends on two factors:

a) Implementation technology used.

b) CPU organization used

An instruction typically consists of a number of elementary micro operations that vary in number and complexity depending on the instruction and the CPU organization used.

# CPU Performance: Introduction

- Micro-operation is an elementary HW operation that can be carried out in a clock cycle.
- Register Transfer operations, arithmetic and logic operation etc
- Thus a single machine instruction may take one or more CPU cycles to complete.
- We can characterize an instruction by Cycles Per Instruction (CPI)

**Average CPI of a program**

- Average CPI of all instructions executed in the program on a given processor
- Different instructions can have different CPIs

# Defining Performance in Terms of Time

- Time is the ultimate measure of computer performance
- A computer exhibits higher performance, if it executes the program faster.
- Time (response time, execution time, latency)
- Response time is amount of time system takes to process a request after it has received one.
- Time between the start and completion of a task is referred to as execution time.

# Defining Performance in Terms of Time

- Latency is a measure of time delay experienced in a system.
- Latency refers to the time delay, normally measured in milliseconds (1/1000 sec.), between initial input and an output.
- Latency is sometimes also called transport delay.
- Throughput (tasks per unit time)
- Throughput (tasks per unit time) :- Total amount of work done in specified time is ‘Throughput’.

# Defining Performance in Terms of Time

- Obviously, execution time will be measured whenever we measuring performance. Faster is the execution time, the performance is better.

- For system management point of view, throughput is important .

# Performance in Terms of Time

- For a given program compiled to run on a specific computer, e can define the following parameters:

a) Total number of instructions executed or instruction count (IC).

b) Average number of cycles per instruction (CPI)

c) Clock cycle time (C) of the computer

Total execution time can be computed as

Execution Time XT= IC x CPI x C

# Performance

- Performance is expressed as unit of things-per-second
- If we are primarily concerned with execution time
  - Performance (x) = $\frac{1}{\text{Execution\_Time}(x)}$

# Performance

- One of the easiest methods to make the comparison
- We measure the execution times of a program on two computers (A & B), as $XT_A$ and $XT_B$.
- Performance can be defined as the recipal of execution time:

  $Perf_A = 1/XT_A$

  $Perf_B = 1/XT_A$
- We can estimate the speedup of machine A over machine B as

  $Speedup=Perf_A / Perf_B = XT_B / XT_A$

# Performance Definitions

- " X is n times faster than Y" means

$$n = \frac{\text{performance}(X)}{\text{performance}(Y)} = \frac{\text{execution\_time}(Y)}{\text{execution\_time}(X)}$$

- Problem: Compare performances of A and B
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds

  Answer: A is (25/20) times faster than B

# Understanding Performance

- If a Pentium-IV runs a program in 8 seconds and a PowerPC runs the same program in 10 seconds, how many times faster is the Pentium-IV?

n = 10 / 8 = 1.25 times faster (or 25% faster)

- Why might someone choose to buy Power-PC in this case?

# Computing CPU time

- Time to execute a given program can be computed as

  CPU time = CPU clock cycles x clock cycle time  or

  CPU time = CPU clock cycles / clock rate

- CPU clock cycles = (# of instruction/program) x (# of clock cycles/instruction)

  = Instruction count x CPI

which gives

  CPU time = Instruction count x CPI x clock cycle time

  CPU time = Instruction count x CPI / clock rate

- Units for this are

$$\text{seconds} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

# Example of Computing CPU time

- If clock rate = 50 MHz, find execute time for a program with 1,000 instructions, if the CPI for the program = 3.5?
- CPU time = instruction count x CPI / clock rate
- CPU time = 1000 x 3.5 / (50 x $10^6$) sec = 70 $\mu$s
- If clock rate increases from 200 MHz to 250 MHz and the other factors remain the same, how many times faster will the computer be?

$$\frac{\text{CPU time old}}{\text{CPU time new}} = \frac{\text{clock rate new}}{\text{clock rate old}} = \frac{250\ \text{MHz}}{200\ \text{MHz}} = 1.25$$

# Computing Overall CPI

- **CPI = avg. no. of cycles per instruction.**

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i$$

| Op | $F_i$ | $CPI_i$ | $CPI_i$x $F_i$ | % Time |
|---|---|---|---|---|
| ALU | 50% | 1 | 0.5 | 23% |
| Load | 20% | 5 | 1.0 | 45% |
| Store | 10% | 3 | 0.3 | 14% |
| Branch | 20% | 2 | 0.4 | 18% |
| Total | 100% | | 2.2 | 100% |

(0.5/2.2)*100=23%

# Example

- A program runs in 10 sec on computer A which has a speed of 400 MHz clock.
- A designer wants to build a new machine B, that will run this program in 6 sec.
- He can use new (more expensive!) technology to increase the clock rate. This will affect the rest of the CPU design, causing B to require 1.2 times as many clock cycles as A for the same program.

- What clock rate should we tell the designer to target?"

[cpu time=(instruction count x cpi)/clock rate]
N→instruction count]

$10=(NxCPI_A)/fa),$

$6=NxCPI_B)/fb, \qquad CPI_B=1.2xCPI_A$

# MIPS

MIPS (Million Instructions per second):

=(Instruction Count)/(Execution time x $10^6$)

MFLOPS = Million floating point operations per second

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Exec. Time} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$$

# Performance

- Performance is determined by execution time
- Other parameters used to measure performance are as follows
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?

# CPI Example

- Two implementations (A and B) of the same instruction set architecture (ISA).
  For some program,
  A has clock cycle time = 10 ns, CPI = 2.0
  B has clock cycle time = 20 ns, CPI = 1.2

- Which machine is faster for this program, and by how much?
- CPU Time =#of Instruction x CPI X Clock cycle time
- (# of instruction is same for both)
- If two machines have the same ISA, which of the quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?

  == # of Instructions

# # of Instructions Example

A compiler designer is to decide between two code sequences.

- First code sequence :- 2 of A, 1 of B, 2 of C
- Second code sequence:- 4 of A, 1 of B, 1 of C

3 classes of instructions:

- A : 1 cycle, B : 2 cycles, C : 3 cycles

- Which sequence will be faster?
  - How much?
  - CPI for each?

First code=(2*1 +1*2+2*3)=10 cycles, 2$^{ND}$ code=(4*1+1*2+1*3)=9 cycles

# of Instructions in 1$^{st}$ code=1(A)+1(B)+2(c)=5,

in 2$^{nd}$ code =4(A)+1(B)+1(c) =6, CPI (1$^{st}$ code)=10 cycles/5=2 CPI,

CPI(2$^{nd}$ code)=9cycles/6=1.5CPI

# MIPS Example

- Two compilers being tested for 100 MHz machine with 3 classes of instructions:
  - A (1 cycle), B (2 cycles), and C (3 cycles)
  - Compiler 1: 5M A, 1M B, 1M C instructions
  - Compiler 2: 10M A, 1M B, 1M C instructions
- Which sequence has higher MIPS?
- Which sequence has lower execution time?

Cycles for Compiler 1=5Mx1 +1M*2+1M*3=10 M cycles.

Cycles for Compiler 1=10Mx1 +1M*2+1M*3=15 M cycles.

Execution Time in Compiler=10M cycles/f=10 Mcycles/100Mhz=1/10 sec.=.10

Execution Time in Compiler=15M cycles/f=10 Mcycles/100Mhz=15/100 sec=.15 sec.

Instruction in compiler 1= 5M+1M+1M=7M,

Instruction in Compiler-II=10+1+1 =12M

MIPS(compiler-1)=Instruction count/(exe.time*$10^6$)=7*$10^6$ /0.1*$10^6$=70 MIPS

MIPS(compiler-2)=Instruction count/(exe.time*$10^6$)=12*$10^6$ /0.15*$10^6$=80 MIPS

# Performance Example

- M1 and M2 are two impl of same ISA.
- M1 clock = 50 MHz, M2 clock = 75 MHz.
- M1 CPI = 2.8, M2 CPI = 3.2 for a program.
- How many times faster is M2 than M1?

$$\frac{\text{ExTime}_{M1}}{\text{ExTime}_{M2}} = \frac{IC_{M1} \times CPI_{M1} / \text{Clock Rate}_{M1}}{IC_{M2} \times CPI_{M2} / \text{Clock Rate}_{M2}} = \frac{2.8/50}{3.2/75} = 1.31$$

- What clock rate of M1 will give same execution time?

Exe. Time (M1)=2.8/*fm1*

Exe. Time (M1)=3.28/*75*

2.8/*fm1*=3.28/*75*, *fm1=75*2.8/3.2=525/8=65.63 Mhz*

# Programs to test Performance

Performance is determined by running a real application

–Typical class of applications
e.g., compilers/editors, scientific applications, graphics, etc.

# Computer Benchmarks

- Benchmark = program (s) used to evaluate computer performance.
- Benchmarks can vary greatly in terms of their complexity and their usefulness.

# Sources of Benchmarks

- Small benchmarks can be easily written
  - nice for architects and designers
  - easy to standardize
- SPEC (Standard Performance Evaluation Corporation) http://www.spec.org
  - companies have agreed on a set of real program and inputs
  - valuable indicator of  performance (HW+SW)

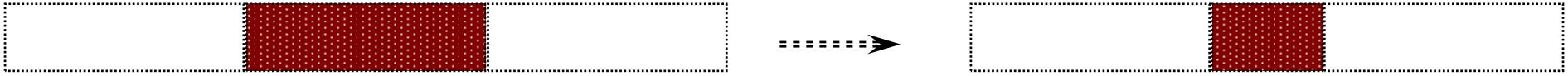
# SPEC ’95

| Benchmark | Description |
| --- | --- |
| go | AI; plays the game of Go |
| m88ksim | Motorola 88k chip simulator |
| gcc | Gnu C compiler |
| compress | Compresses and decompresses file |
| li | Lisp interpreter |
| ijpeg | Graphics compression/ decompression |
| perl | Manipulates strings and primes |
| vortex | A database program |
| tomcatv | Mesh generation program |

# Amdahl's Law

$$\text{Speedup} = \frac{\text{ExTime old}}{\text{ExTime new}} = \frac{\text{Performance new}}{\text{Performance old}}$$

- Use to find the maximum expected improvement to an overall system when only part of the system is improved.



$T_{old}$ $T_{new}$

- Improvement from Told to Tnew, the performance of whole program will be speeded by a factor.

# Amdahl's Law

Execution Time After Improvement = time unaffected + time affected / improvement

Example: A program runs in 100 seconds on a machine, with multiply responsible for 80%. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

How about making it 5 times faster?

Exe. Time for multiply= 80%, Rest=100-80=20%, New exe.time=100/4=25 sec.

80 secs/n+20 secs=25, n=80/5=16 times.

100/5=80/n+20sec.

# Example

- Suppose, we want to make all floating-point instructions to run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

  Exe. Time for float point=10/2 = 5 sec.

  After speed up floating point takes 5/5 sec=1 sec.

  Other instructions take (10-5) or 5 sec.

  After speed up, total time required=5+1 =6sec.

  Speed up=10sec/6=1.678

# Example

- We are looking for processor with new floating-point unit of 5 times faster. We are considering one benchmark to run for 100 seconds with the old floating-point hardware. How much execution time would the floating-point instructions take in this program in order to yield a speedup of 3 on this benchmark?
- Say, floating point instruction takes x sec. now we will make all floating instructions to run 5 times faster.
- Rest instructions take (100-5x)
- After enhancement, time required=100-5x+x
- =100-4x
- Speed up=100/(100-4x)=3, x=200/12=16.667 sec.

# Example

- A processor currently requires 10 seconds to run a program and processor performance improves by 50 % per year. By what factor does processor performance improve in 5 years?

  $(1 + 0.5)^5 = 7.59$

- Exec time after 5 yrs? 10/7.59 = 1.32 s

- What assumptions are made in the above problem?