

Chapter 2

Instructions: Language of the Computer

Computer Architecture and Organization

School of CSEE

1. Introduction to Instruction Set Architecture

2. Binary Number Basic

- In this section, we will discuss the following issues.
 - What is instruction set architecture ?
 - What does instruction format consists of ?
 - How to represent an operation with instructions supporting different number of operands ?
 - What is the addressing mode of operands?
 - What types of operations are there to support instruction set completeness ?

1. Introduction to Instruction Set Architecture

- **Basics**

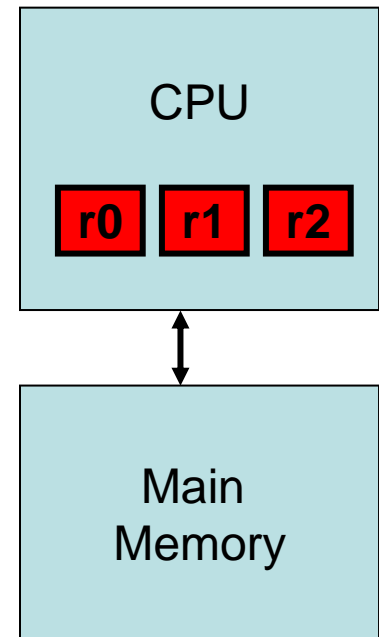
- The number of operands

- Addressing modes

- Instruction Set Completeness

- Machine Instruction is a native language of processor.
- A machine instruction (macro instruction) is a group of bits that tells the computer to perform a specific operation.
- Program : A finite sequence of instructions that performs a specific task (on the computer with microprocessor)

- Reside in CPU
- Temporary storage of data/instruction
- Faster than main memory
- Expensive
- Types of Registers
 - General Purpose Register : for computation purpose (r0, r1, ...)
 - Special Purpose Register : PC (Program Counter), AC (Accumulator), IR (Instruction Register) *Assembly language* (직접 접근)
 - Some Registers are not accessible to programmers



Special Purpose Registers

- **PC** (Program Counter) : Special register to hold address of next instruction
- **AC** (Accumulator) : Special register which holds the result of the computation temporarily.
- **IR** (Instruction register) : Special register which holds the instruction fetched from the memory.



ISA

Instruction Set Architecture

- ISA (Instruction Set Architecture) is essentially a programmer's view of processors
 - information needed to interact with processor, but not the ~~details~~ of how it is designed and implemented

(2+4+8) 아님 아나

- ISA includes:

- Processor's instruction set

- the set of assembly language instructions

- Programmer accessible register within processor

- Size of each programmer-accessible registers
- Instructions that can use each register

접근가능한 레지스터의
크기와 Instruction

- Information necessary to interact with memory

- ✱ • Memory alignment (메모리의 중요성을 알고 있어야 함)

- How processor reacts to interrupt from the programming view point

Exercise

ISA

(Q1) What are Not included in Instruction Set Architecture definition ?

(1) Programmer accessible registers and their size

~~(2) Number of address bits~~

(3) Machine Instruction Sets

~~(4) Physical actions required to recognize that an external interrupt has occurred~~

(5) How to react to interrupt from the programming point of view

~~(6) How to design and implement the processor~~

(7) Memory alignment for data

~~(8) The number of clock cycles required for each instruction execution~~

(Q2) If two machines A and B have the same Instruction Set Architecture, which one is true ?

☒ (1) A machine object program running on A can run on B, too.

(2) Performance of A = Performance of B

(3) Design of A = Design of B

(4) All the internal registers in two microprocessors are same

Levels of Programming Languages

- **High Level Languages**
 - Machine independent
- **Assembly Languages**
 - Each processor has its own assembly language.
 - Each statement corresponds to a machine instruction
 - Directly manipulate data stored in a processor's internal components (registers)
- **Machine Languages**
 - Binary code causing processor to perform certain operation
 - Native language of processor

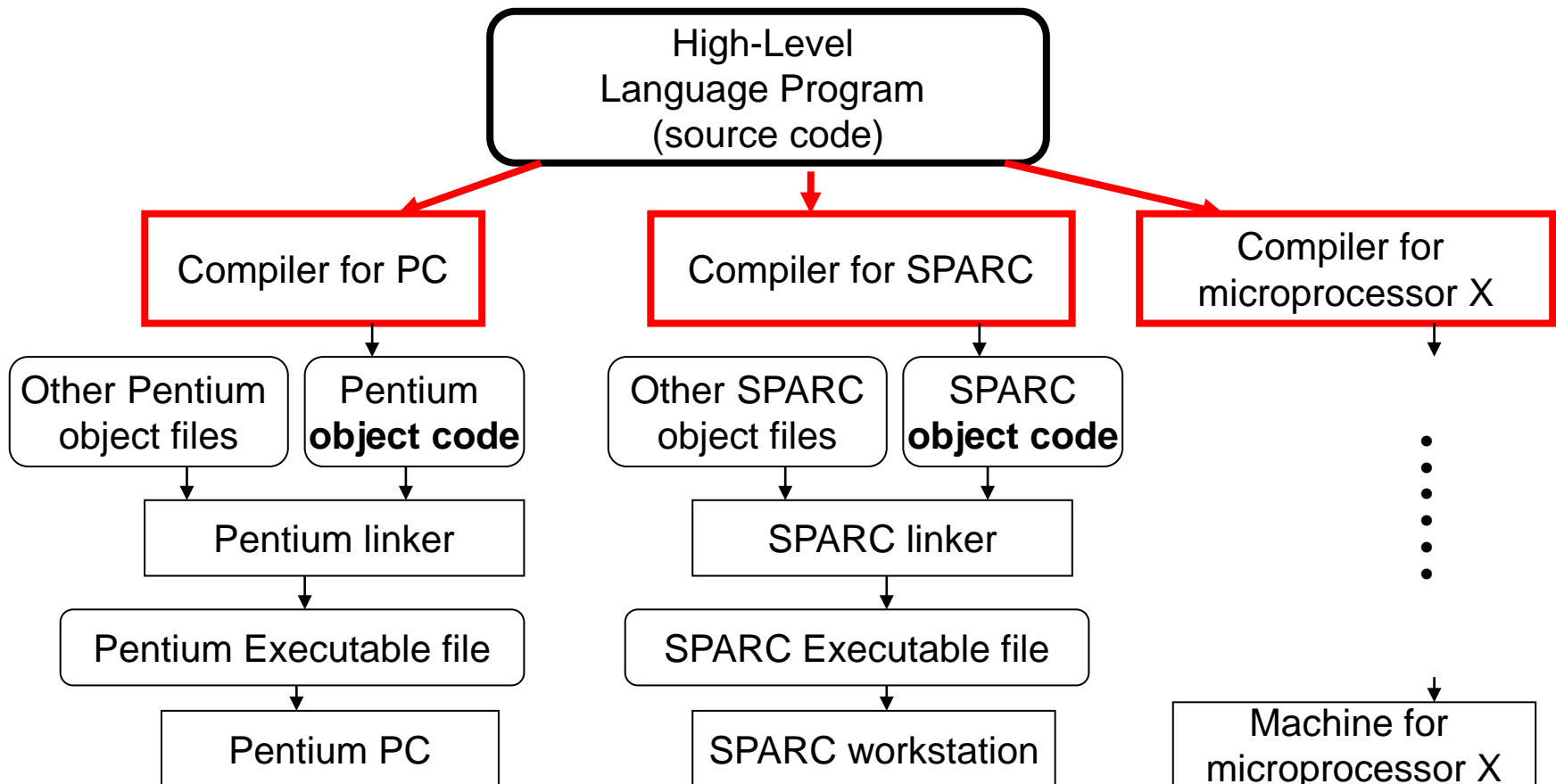
Highest level



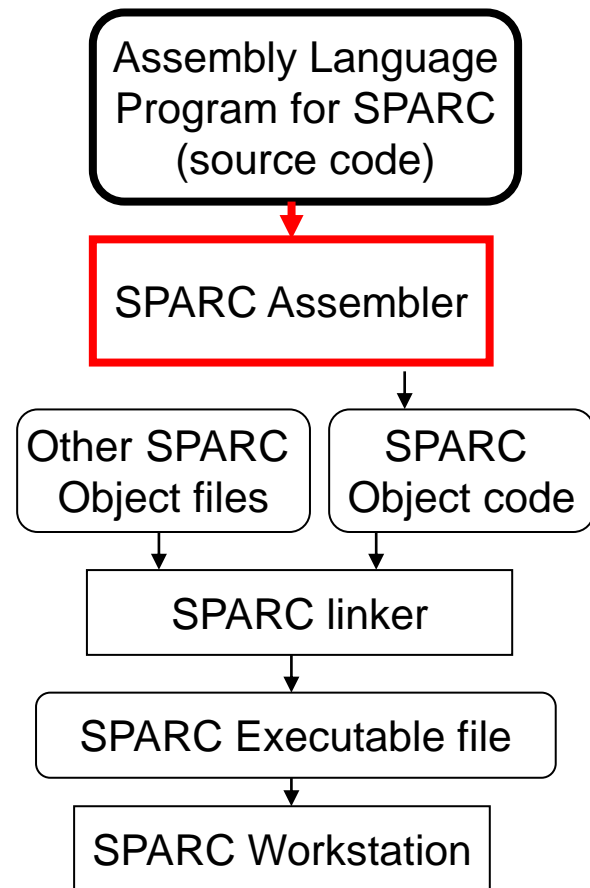
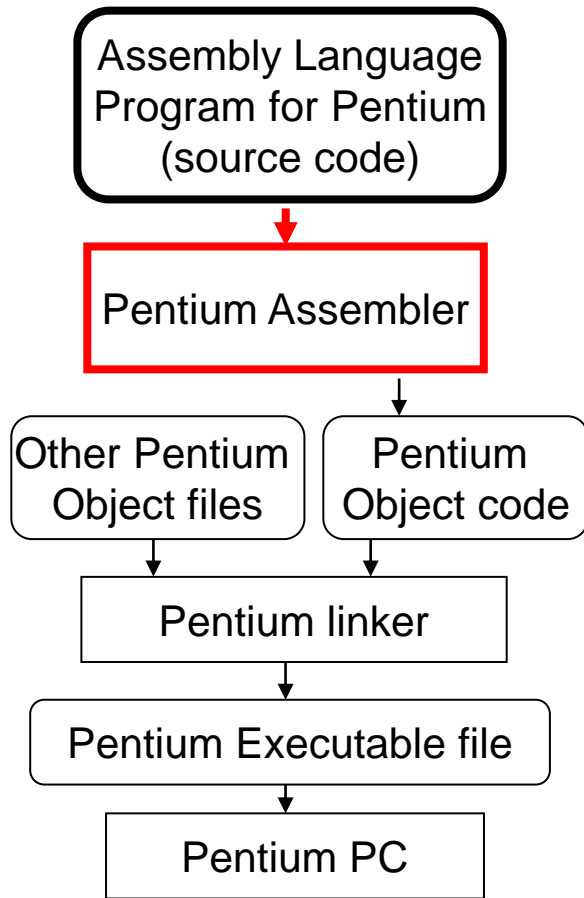
Lowest level

Compiling

- The same High-Level Language program can be compiled to run on different processors.



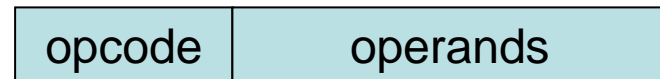
- Each processor has its own Assembly Language





Instruction Format

- Assembly language is converted to its equivalent machine code represented as binary number in specific format
- Instruction length ; 8 bits, 16 bits, 32bits, 64 bits...
 - (1) fixed size
 - (2) variable size
- Typical instructions consist of two parts:
 - Opcode: operation to be performed
 - Operands (address): indicate where data are
 - Number of Operands
 - Addressing Mode

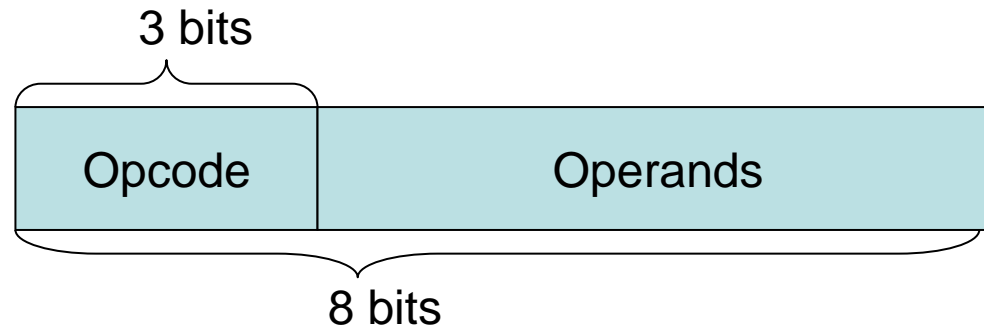




2

Instruction Format

- If op code is k bits long, there could be up to 2^k instructions.
- If address part is m bits long, 2^m different memory locations can be accessed by address part.
- The computer architect should decide the length of op code and operand.



1. Introduction to Instruction Set Architecture

- Basics
- **The number of operands**
- Addressing modes
- Instruction Set Completeness

Number of Operands

- **Three Operands**

- **ADD A, B, C**

Destination : A

Sources : B and C

($A \leftarrow B + C$)

opcode	operand #1	operand #2	operand #3
--------	---------------	---------------	---------------

- **Two Operands**

- **ADD A, B**

Destination : A

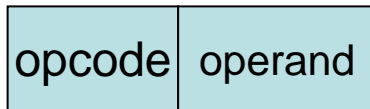
Sources : A and B

($A \leftarrow A + B$)

opcode	operand #1	operand #2
--------	---------------	---------------

- **One Operands**

- Assume a Special Register “AC” (Accumulator) to hold data value



- ADD A
Destination : AC
Sources : A and AC
 $(AC \leftarrow AC + A)$

- LD A
Destination : AC
Source : A
 $AC \leftarrow A$

- Q: Is Zero Operand Instruction Possible ?

Number of Operands

Stack machine

- **Zero Operand Instruction**

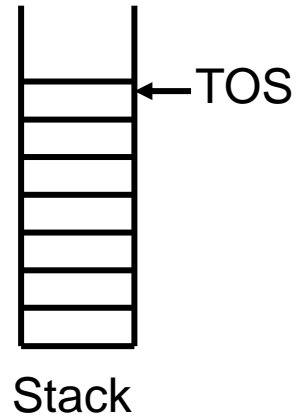
- Assume a Stack (First In Last Out memory)
 - All Operands are placed at TOS (Top Of Stack)
 - Data Move by PUSH and POP Operation

PUSH A ($TOS \leftarrow A$)

PUSH B ($TOS \leftarrow B$)

ADD ($TOS \leftarrow A+B$)

POP C ($C \leftarrow TOS$)



opcode

- Above is zero operand instruction since operation type instruction (ADD, OR, etc) do not need an address field.



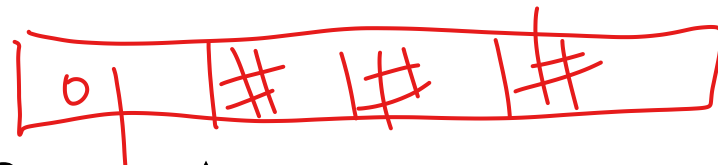
The number of address field

Example) Suppose op code is 5 bits long and each address fields is 12 bits long

3 addr. inst. mach. : $5+3 \times 12=41$ bits

2 addr. inst. mach. : $5+2 \times 12=29$ bits

1 addr. inst. mach. : $5+1 \times 12=17$ bits



**wider bus &
less # of instructions**

Exercise

(Q) Implement the following expression with three-, two-, one-, and zero-operand instruction sequences. Assume we have ADD and DIV Instructions.

$$Z = (A+B)/C$$

1. Introduction to Instruction Set Architecture

- Basics
- The number of operands
- **Addressing modes**
- Instruction Set Completeness

- Different ways to specify the place where an operand resides in memory or register.
- Advantages
 - It gives programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, etc.
 - It may reduce the number of bits in the addressing field of the instruction.
- Effective Address (E.A.)
 - Actual address of the location containing the referenced operand.

ex) In ARM

```
LDR r1, [r2, #20]
```

1. Implied addressing mode : No explicit address

(ex) instructions for accumulator and stack

Ex) RTS : Return From Subroutine

PUSH A

ADD A

2. Immediate addressing mode : Operand field contains the actual operand value

Ex) In ARM,

ADD r3, r3, #4



3. Register addressing mode : Selected Register contains the operand

- E.A. = Selected Register

Ex) In ARM,

ADD r0, r1, r2

4. Register Indirect addressing mode : Selected Register contains the **address of operands**

- E.A. = Contents of Selected Register

Ex) In Motorola 68000,

MOVE.W (A1), D1



Addressing Modes

5. Direct Addressing mode : Effective address is equal to **the address part of the instruction** and operands reside in memory

- E.A. = Address Field of Instruction

Ex) In Motorola 68000,

MOVE.W 10000, D1

6. Indirect Addressing mode : The Address field gives **the address where the effective address is stored.**

- E.A. = Memory [Address Field of Instruction]
- Two Memory References (Slowest)

Ex) In hypothetical machine,

ADD (1000)



Direct vs. Indirect

Strength) Address field can be extended to word length compared to direct addressing

Shortcoming) It needs two memory references : slower

- **Multilevel of indirection**
- **How to identify data / address?**



Direct vs. Register

Register addressing Mode vs. direct addressing mode

- Register addressing mode is faster
- Addressing space of memory is larger than register

7. (PC) Relative Address mode : Value in PC is added to the address part of instruction to obtain the effective address (branch type instructions)

- E.A. = PC + Offset in Address Field of Instruction

ex) In MIPS,

100	bne \$s0, \$s1, Exit	← 실행중
104	add	
108	sub	
112	lw	
116	Exit:	

PC → 104
3*4

Exit = 3

E.A.
= 104

Assembler translate 'Exit' as 3.

When the program is executed,

$$\text{E.A.} = (\text{PC}) + \text{Exit} = 104 + 3 \times 4 = 116$$

Branching less than...

Programmer write the assembly language program.

```
slt $t0, $s0, $s1 # t0 = 1 if $s0 < $s1
```

```
beq $t0 $zero, Less
```

```
add $s2, $s0,$s1
```

```
j Exit
```

```
Less : sub $s2,$s0,$s1
```

```
Exit:
```

Assembler convert the above program into machine code.

```
000000 10001 10010 01000 00000 101010    /* slt
```

```
000100 01000 00000 00000 00000 000010    /* beq
```

```
000000 .....                      /* add
```

```
000010 .....                      /* j
```

```
000000 .....                      /* sub
```



Branching less than...

Machine code is loaded into main memory.

Addr.	Machine Code	
200	000000 10001 10010 01000 00000 101010	/* slt
204	000100 01000 00000 00000 00000 000010	/* beq
208	000000	/* add
212	000010	/* j
216	000000	/* sub

When 'beq' instruction is executed, PC contains 208. If \$t0 contains 0, the condition is 'TRUE' and the branch should be taken. The CPU computes the memory address of next instruction to be executed as follows.

$$\text{Address of next instruction} = \text{PC} + 2 * 4 = 216$$

8. Base Register Addressing mode : Value in Base Register

(base address) is added to the address part of instruction (offset) to obtain the effective address (to facilitate the relocation of programs in memory)

- E.A. = Base Register + Offset in the Address Field of Instruction

ex) In ARM,

LDR r5, [r3, #32]

E.A. = (r3) + 32

In MIPS,

lw \$t1, 1000(\$s1)

E.A. = (s1) + 1000

1. Introduction to Instruction Set Architecture

- Basics
- The number of operands
- Addressing modes
- **Instruction Set Completeness**

- **Completeness** : A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable
 - Categories of Instructions that should be covered
 - Arithmetic/Logical
 - Data Transfer (including I/O)
 - Control Transfer
- **Instruction Orthogonality** : Instructions are orthogonal if they do not overlap, or perform same function

- **Arithmetic/Logic Instructions (Data Operations)**
 - Modify Data Values
- **Data Transfer Instructions**
 - Copy Data from one Place to another
- **Control Transfer Instructions (Program Control)**
 - Jump or Branch
 - (1) conditional branch ex) BEQ A, B, C
 - (2) unconditional branch ex) Jump A

[1] Instruction Types (Arithmetic/Logic)

- **Arithmetic/Logic Instructions (Data Operations)**
 - Integer arithmetic operations
 - ADD/Subtract
 - Multiply
 - Divide
 - Increment/Decrement
 - Logical operations
 - Bitwise AND
 - Bitwise OR
 - Complement
 - Shift instructions
 - Logic shift
 - Arithmetic shift
 - Rotate
 - Floating point arithmetic operations

- Arithmetic Instructions

- ADD, SUB, MUL, DIV : smallest set we need is () and

()

- Integer Arithmetic and Floating Point Arithmetic Operations :

() Operations can be emulated by () Operations

- Logical Instructions

- AND, OR, NOT, etc : smallest set is () or ().

CISC vs. RISC

Increasing functionality – CISC approach - reduce the **instruction count**, but need more op code bits and hardware. Suppose we have complex instruction set:

- ➔ more hardware to realize this instruction set
- ➔ possibly more propagation delay resulting less clock frequency or need more clock cycles to execute the instruction i.e. more **CPI (clock cycles per instruction)**

* $\text{CPU clock cycles} = \text{instruction count} \times \text{CPI}$



[2] Instruction Types (Data Transfer)

- **Data Transfer Instructions**
 - **Load Data from Memory into processor**
 - **Memory to register**
 - **Store Data from processor into Memory**
 - **Register to memory**
 - **Move data within processor**
 - **Copy from register to register**
 - **Special Instructions**
 - **Between I/O Device and processor**
 - **(Block transfer of data)**
 - **Between Memory and memory**
 - **Between Memory and IO device**



memory mapped I/O

To give a command to an I/O device, the processor must be able to address the device and to supply one or more command words. Two methods are used to address the device: memory-mapped I/O and special I/O instructions. In memory-mapped I/O, portions of the address space are assigned to I/O devices. Read and writes to those addresses are interpreted as commands to the I/O device.



[3] Instruction Types (Control Transfer)

- **Control Transfer Instructions**
 - **Conditional Branch instructions**
 - **Unconditional Branch instructions**
 - **Subroutine Calls and returns instructions**
 - **Software interrupt instructions**
 - **(Hardware Interrupts)**
 - **Halt instructions to stop**



Exercise

- **Which issue is related with the question?**

Does the Instruction Set have all of Instructions needed to perform its required tasks?

(1) Completeness (2) Orthogonality

- **Which issue is related with the question ?**

Does the Instruction Set has any redundant Instruction ?

(1) Completeness (2) Orthogonality

- ISA is a **Programmer's** View of processor
- There may be many valid ISA for a specific requirement
- Machine Instruction Consists of “Opcode” + “Operands”
- Instruction design should consider
 - Number of Operands of each Instruction
 - Addressing Mode for Operands of each Instruction
 - Instructions Set should be Complete
 - Arithmetic/Logical Operations
 - Data Transfer
 - Control Transfer

1. Introduction to Instruction Set Architecture

2. Binary Number Basic



Introduction

- The goal of this section is to repeat the binary number basics.
- It includes binary number representation, binary number arithmetic, and overflow.
- Since the computer is designed based on binary number, it is very important to know about binary number basics.



- Bits are just bits (no inherent meaning)
 - conventions define relationship between bits and numbers
- Binary numbers (base 2) : Unsigned
 - 0000 0001 0010 0011 0100 0101 0110 0111 1000...
 - decimal: $0 \dots 2^n - 1$
- Of course it gets more complicated:
 - numbers are finite (overflow)
 - fractions and real numbers
 - negative numbers
 - e.g., no MIPS subi instruction; addi can add a negative number
- How do we represent negative numbers?
 - i.e., which bit patterns will represent which numbers?

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

Negative Number Representations

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $$\begin{aligned} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

- Using 32 bits

- $$-2,147,483,648 \text{ to } +2,147,483,647$$

- 32 bit signed numbers:

0000 0000 0000 0000 0000 0000 0000 0000	$_{\text{two}}$	=	0_{ten}	
0000 0000 0000 0000 0000 0000 0000 0001	$_{\text{two}}$	=	$+ 1_{\text{ten}}$	
0000 0000 0000 0000 0000 0000 0000 0010	$_{\text{two}}$	=	$+ 2_{\text{ten}}$	
...				
0111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}}$	=	$+ 2,147,483,646_{\text{ten}}$	
0111 1111 1111 1111 1111 1111 1111 1111	$_{\text{two}}$	=	$+ 2,147,483,647_{\text{ten}}$	<i>maxint</i>
1000 0000 0000 0000 0000 0000 0000 0000	$_{\text{two}}$	=	$- 2,147,483,648_{\text{ten}}$	
1000 0000 0000 0000 0000 0000 0000 0001	$_{\text{two}}$	=	$- 2,147,483,647_{\text{ten}}$	<i>minint</i>
1000 0000 0000 0000 0000 0000 0000 0010	$_{\text{two}}$	=	$- 2,147,483,646_{\text{ten}}$	
...				
1111 1111 1111 1111 1111 1111 1111 1101	$_{\text{two}}$	=	$- 3_{\text{ten}}$	
1111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}}$	=	$- 2_{\text{ten}}$	
1111 1111 1111 1111 1111 1111 1111 1111	$_{\text{two}}$	=	$- 1_{\text{ten}}$	



2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \ \dots \ 0010_2$
 - $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$
 $= 1111 \ 1111 \ \dots \ 1110_2$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- In ARM instruction set
 - LDRSB, LDRSH: extend loaded byte/halfword
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- Cf) “zero extension” for logical instruction



Exercise

1. Represent -11 in signed magnitude, one's complement, and two's complement notation.

1. Introduction to Instruction Set Architecture
2. Binary Number Basic
- 3. MIPS Design Principles**
4. MIPS Instruction Set
5. MIPS Instruction Format & Addressing Modes
6. Supporting Procedure Call

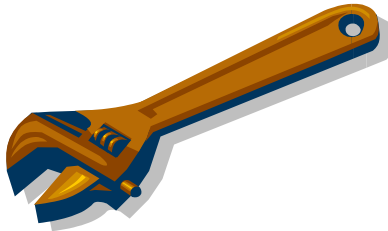
- In this and following modules, we will discuss about the MIPS **Instruction Set Architecture**.
- We will briefly look at MIPS register and memory structure.
- MIPS ISA is based on four **Design Principles** which is important to general RISC design.



MIPS ISA Key Idea

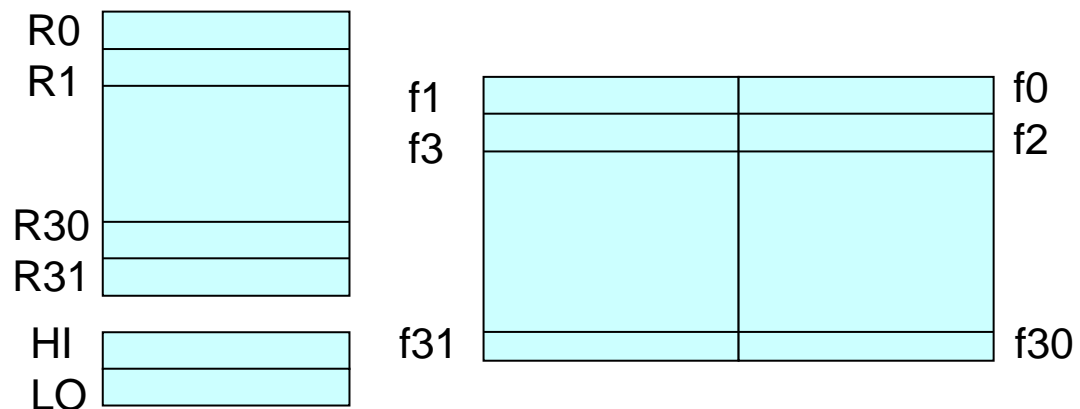
- **Goals of instruction set design for MIPS**
: *maximize performance and minimize cost, and reduce design time (of compiler and hardware)*

By the simplicity of Hardware!



MIPS Register Model

- **32 x 32 bits General Purpose Registers**
 - R0 always contains 0
- **32 x 32 bits Floating Point Registers**
 - FP registers can be paired as 16 x 64bits FPRs for double precision by specifying even register which holds less-significant word
- **HI (32 bits) and LO (32 bits)**
 - for multiply and divide
- **PC : invisible to programmers except branch and procedure call**



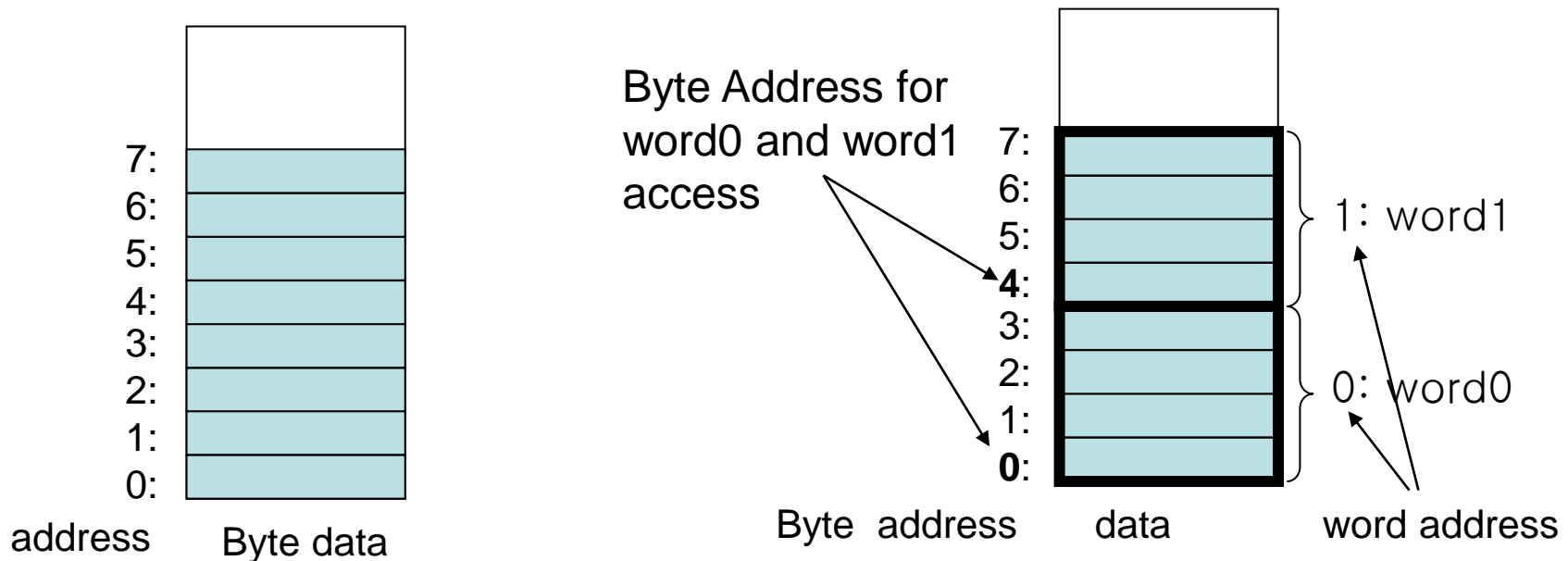
Register Naming Convention

0	\$zero constant 0	16	\$s0 permanent
1	\$at reserved for assembler	...	
2	\$v0 return values	23	\$s7
3	\$v1	24	\$t8 temporary
4	\$a0 arguments	25	\$t9
5	\$a1	26	\$k0 OS kernel (reserved)
6	\$a2	27	\$k1
7	\$a3	28	\$gp global pointer
8	\$t0 temporary	29	\$sp stack pointer
...		30	\$fp frame pointer
15	\$t7	31	\$ra return address



Memory Structure

- **Memory is just a large 1-dim. Array, with address acting as index to the array.**
- **Memory address indicates bytes.**
- **In some cases, memory is accessed in words (4 bytes)**
 - **EX) When array element is 4 bytes integer, $A[1]$ will look up $A+1*4$ byte address**



Memory Model

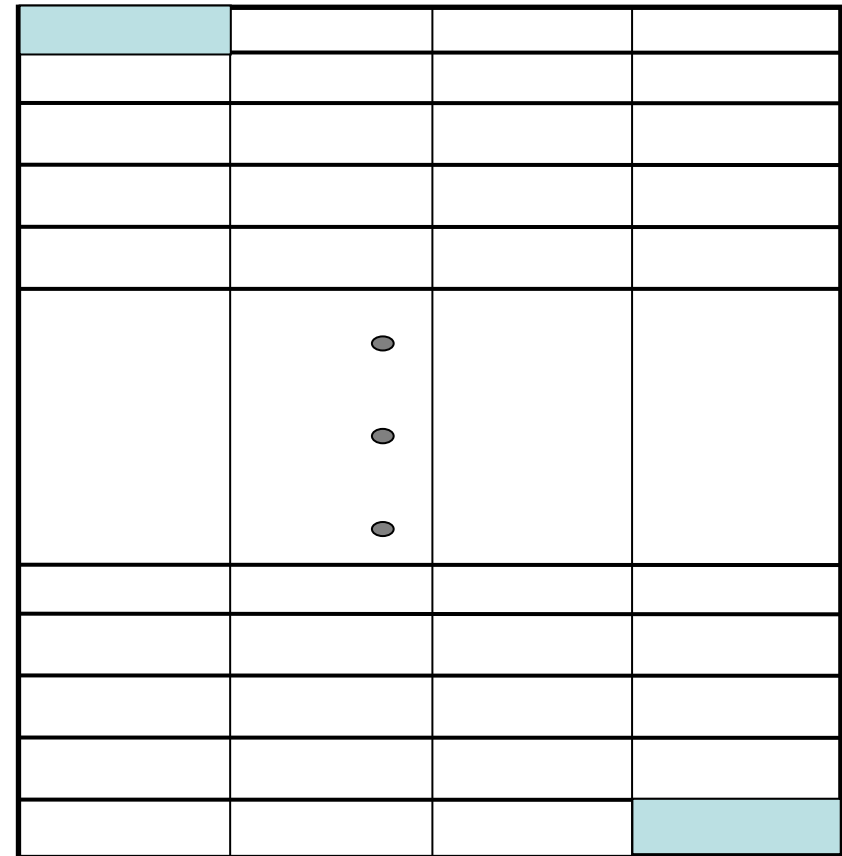
- MIPS has 32 bits address.

High addr: 0xffff ffff

$$2^{32} \text{ bytes} = 2^{30} \text{ words}$$

Bytes are nice, but most data items use larger "words".

Low addr: 0x0000 0000



1 word = 4 bytes



MIPS Design Principles

- Simplicity favors regularity.
- Smaller is faster.
- Make common case fast.
- Good design demands a compromise.

Design Principle 1

- Most of arithmetic/logic instructions have 3 operands
 - Order is fixed (destination first)

- Example:

C code : $C = A + B$

MIPS : ADD C, A, B

C code $C = A - B$

MIPS : SUB C, A, B

- Requiring many instruction to have exactly three operands




Design Principle 1 : Simplicity favors Regularity

- Operands of MIPS arithmetic/logic instructions are restricted to special locations built in hardware called “Registers”. (register addressing mode)

- Example:

C code: $A = B + C$
MIPS code: `add $s0,$s1,$s2`



- Compiler associates the variables with the registers.

- MIPS provide only 32 registers available to programmers.
- Limited number of registers
 - Question: Why only 32 registers?
 - Answer: Our underlying design principles of hardware technology

A very large number of registers may increase clock cycle time (because it takes more time for electrical signals to traverse farther).



Design Principle 2 : Smaller is Faster



Design Principle 3

- Constants are used quite frequently as operands (> 50% of MIPS arithmetic instructions)
 - EX) $A = A + 1$
 $B = A - 5$
- (Solution 1 for constants)
 - Put typical values in memory and load them from memory
 $\text{LW } \$t0, \text{AddrOfConstant1}(\$s0) \# \$t0 \leftarrow 1$
 $\text{ADD } \$s3, \$s3, \$t0 \# A = A + 1$
 - Create hard-wired register for specific constant value like R0
(R0 = \$zero = have constant value 0)

Design Principle 3

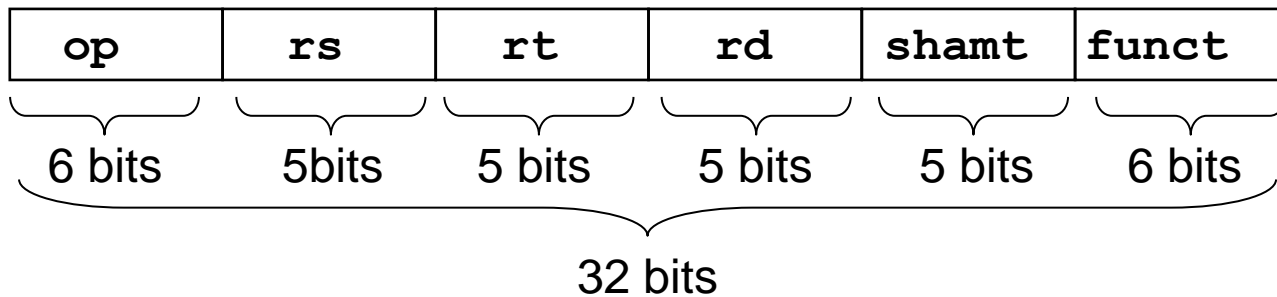
- (Solution 2 for constants) Make constants part of arithmetic instructions (much faster than loaded from memory)
 - C code : $A = A + 1$
 - MIPS code: `ADDI $s3,$s3,1`



Design Principle 3 : Make the Common case Fast

Design Principle 4

- Assembly code should be represented in a binary code (Machine Language)
- Machine language structure:
 - Length : 32 bits (Design Principle 1)
 - Fields :



Design principle 4

- There are 3 types of instruction formats.

R-type format

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

ADD \$t0, \$s1, \$s2

\$t0=R8, \$s1=R17, \$s2=R18

Decimal:

0	17	18	8	0	32
---	----	----	---	---	----

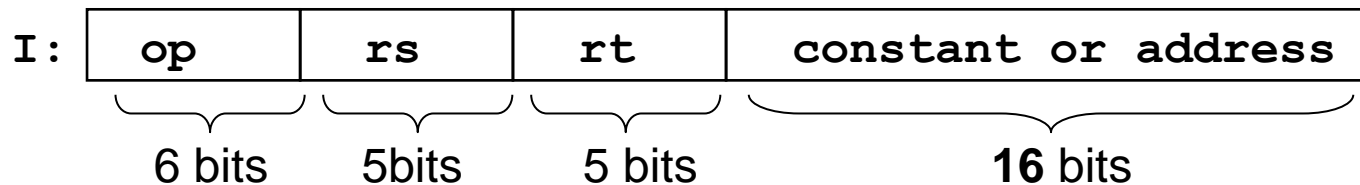


Design Principle 4

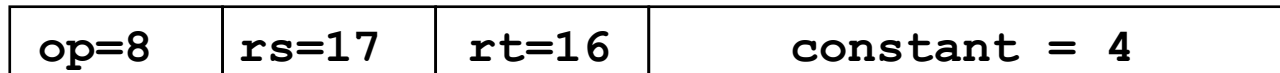
- **Problems with Instructions that need longer fields for constant value (LW, SW, ADDI).**
 - **5 bits field is too small to be useful as an array index.**
ex) **ADDI \$s3,\$s3,1000**
 - **Hence, conflict between desire to have single instruction format and keep instructions same length.**

Design Principle 4

I-type format : for Immediate value



ADDI \$s0, \$s1, 4



Design Principle 4 :
Good Design demands good Compromises.



Exercise

- Explain the difference between byte-addressable memory and word-addressable memory.

Exercise

- **Connect Each Principles to one of the Examples.**

- **Principles:**

- **Design Principle 1 : Simplicity favors Regularity**
- **Design Principle 2 : Smaller is Faster**
- **Design Principle 3 : Make Common Case Fast**
- **Design Principle 4 : Good Design demands good Compromises**

- **Examples:**

1. **Limited Number of registers,**

2. **All Arithmetic Instructions have Three Operands**

3. **Support Immediate value Operands**

4. **Different types of Instructions with same length rather than different Lengths of Instructions of single type**



1. Introduction to Instruction Set Architecture
2. Binary Number Basic
3. MIPS Design Principles
- 4. MIPS Instruction Set**
5. MIPS Instruction Format & Addressing Modes
6. Supporting Procedure Call



Introduction

- In this module, we will discuss about the MIPS **Instruction Set Architecture**
- MIPS satisfies instruction set completeness.
 - Arithmetic / Logic
 - Data Transfer
 - Control



(1) Arithmetic Instructions

- **ADD \$s0, \$s1, \$s2 # \$s0 \leftarrow \$s1 + \$s2**
- **ADDI \$s0, \$s1, 100 # \$s0 \leftarrow \$s1 + 100**
- **SUB \$s0, \$s1, \$s2 # \$s0 \leftarrow \$s1 - \$s2**
- **MULT \$s1, \$s2 # Hi, Lo \leftarrow \$s1 * \$s2**
- **DIV \$s1, \$s2 # Hi, Lo \leftarrow \$s1 / \$s2**
- **MFHI \$s1 # \$s1 \leftarrow Hi**
- **MFLO \$s1 # \$s1 \leftarrow Lo**



Logical Instructions

- **AND \$s0, \$s1, \$s2 # \$s0 ← \$s1 bitwise-AND \$s2**
- **OR \$s0, \$s1, \$s2 # \$s0 ← \$s1 bitwise-AND \$s2**
- **ANDI \$s0, \$s1, 31 # \$s0 ← \$s1 bitwise-AND 31**
- **ORI \$s0, \$s1, 32 # \$s0 ← \$s1 bitwise-OR 32**
- **NOR \$s0, \$s1, \$s2 # \$s0 ← \$s1 bitwise-NOR \$s2**
- **SLL \$s0, \$s1, 10 # \$s0 ← \$s1 << 10 (shift left logical)**
- **SRL \$s0, \$s1, 10 # \$s0 ← \$s1 >> 10 (shift right logical)**

- **(Question) How to Implement bit-wise NOT operation with above instructions ? (How \$s1 ← Complement(\$s2) ?)**



Exercise

- (Question) What might a C Compiler produce for the code ?

$$f = (g+h) - (i+j)$$

- Assume variables g, h, i, j, and f are assigned to the registers s0,s1,s2,s3,and s4
- Compiler allocates variables with registers

ANSWER:

(2) Data transfer instructions

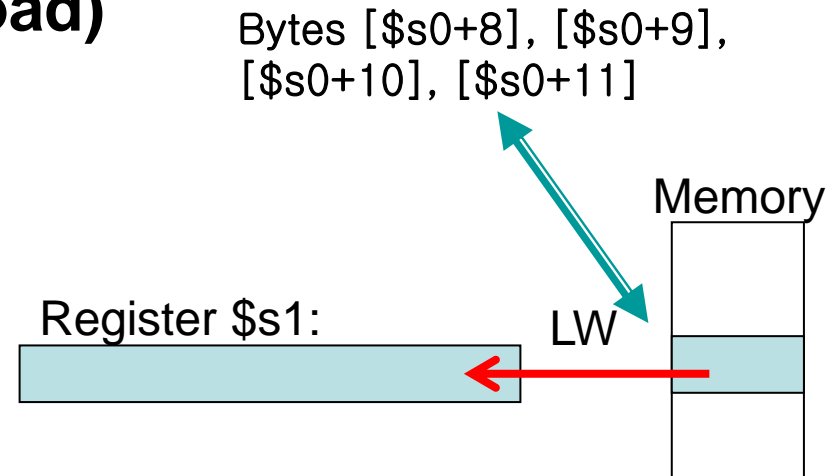
- **Why data transfer instructions ?**
 - Data structures are kept in memory.
 - BUT, arithmetic/logic operations occur only on registers.
 - So, we need data transfer instructions moving data between Memory and Registers
 - We might also need to transfer data between registers.
- **MIPS has only two instructions for Memory Access**
(← RISC architecture)

lw sw

- Read Data From Memory (Load)

- LW \$s1, 8 (\$s0)

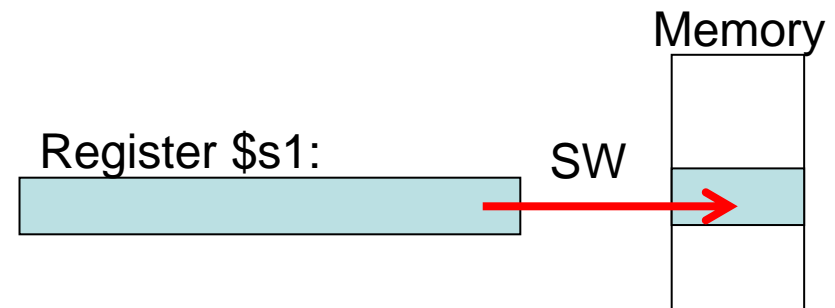
- : $\$s1 \leftarrow \text{Memory}[\$s0 + 8]$



- Write Data To Memory (Store)

- SW \$s1, 12(\$s0)

- : $\text{Memory}[\$s0 + 12] \leftarrow \$s1$



Data transfer Instructions

- Example:

C code: $A[8] = h + A[8];$

MIPS code:
 lw $\$t0, 32(\$s3)$ $\$s3$: Address of A
 add $\$t0, \$s2, \$t0$ $\$s2$: h
 sw $\$t0, 32(\$s3)$ $32 = 8 * 4 (= \text{word size})$

- Source comes first in store instruction
- Base Register Addressing** (Address: $[R] + \text{Offset}$)

Exercise

- **Convert the C Program into MIPS assembly language program.**

`A[1] = A[3] + 3;`

Associate the base address of A with the register \$s0. Assume every data is integer (32-bits).

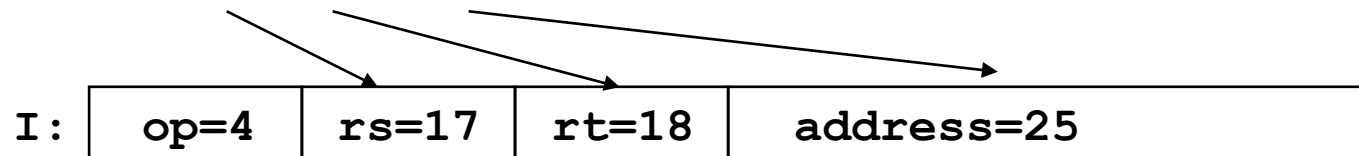


(3) Control Instructions

- MIPS has conditional and unconditional branch instructions.
 1. Conditional branch ~ BEQ, BNE, SLT, SLTI
 2. Unconditional branch ~ J, JR, JAL

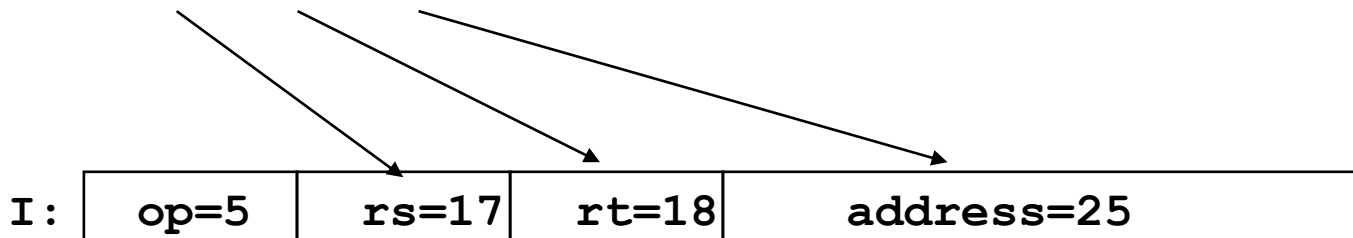
- Branch if Equal

- BEQ \$s1,\$s2, label : if (\$s1 == \$s2) PC ← TargetAddress



- Branch if Not Equal

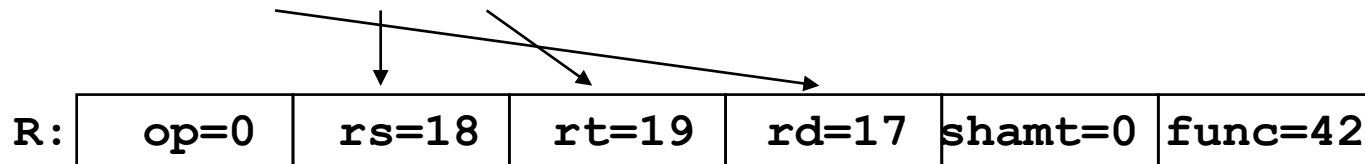
- BNE \$s1,\$s2, label : if(\$s1 ≠ \$s2) PC ← TargetAddress



Branching less than...

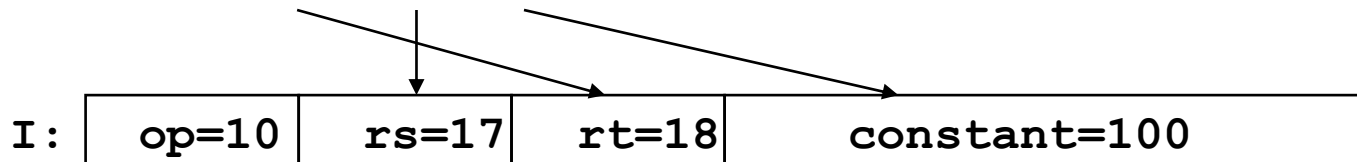
- Set Less Than

- SLT \$s1,\$s2,\$s3 : if (\$s2 < \$s3) \$s1 ← 1 else \$s1 ← 0



- Set Less Than immediate

- SLTI \$s1,\$s2, 100 : if (\$s2 < 100) \$s1 ← 1 else \$s1 ← 0



Branching less than...

if $x < y$ then
 $z = x + y$;
else
 $z = x - y$;

$\$s0 : x$
 $\$s1 : y$
 $\$s2 : z$

$\text{slt } \$t0, \$s0, \$s1$ # $t0 = 1$ if $\$s0 < \$s1$
 $\text{beq } \$t0, \$zero, \text{Less}$
 $\text{add } \$s2, \$s0, \$s1$
 j Exit

Less : $\text{sub } \$s2, \$s0, \$s1$
Exit:

'slti' instruction

Ex) $\text{slti } \$t0, \$s2, 10$ # $\$t0 = 1$ if $\$s2 < 10$

- Convert the C code into MIPS instructions

- C code:

- if (i == j)

- goto L0;



- if (i >= j)

- F = G+H;

- else

- F=G-H;



Exercise

- Convert C Loop statement code into MIPS code.

```
i = 0;  
while (A[i] == 0) {  
    i++;  
}
```

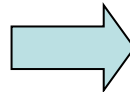


```
for (S=0, i=0; i<100; i++) {  
    S = S + i ;  
}
```



- **J *target***
 - Jump to a *target* address
by setting PC as given new value
 - EX) J label → PC = TargetAddress
 - (C code)

```
Label1: A = B + C
....
goto Label1
```



- (MIPS code)

```
Label1: ADD $s0,$s1,$s3
....
J Label1
```

Unconditional Branch Instructions

- **Jump**

- **J 1000 : PC \leftarrow 1000**

J:

op=2	target=250 (word address)
------	---------------------------

- **Jump Register**

- **JR \$ra : PC \leftarrow \$ra**

R:

op=0	rs=31	rt=0	rd=0	shamt=0	func=8
------	-------	------	------	---------	--------

- **Jump and Link**

- **JAL 1000 : \$ra \leftarrow PC; PC \leftarrow 1000**

J:

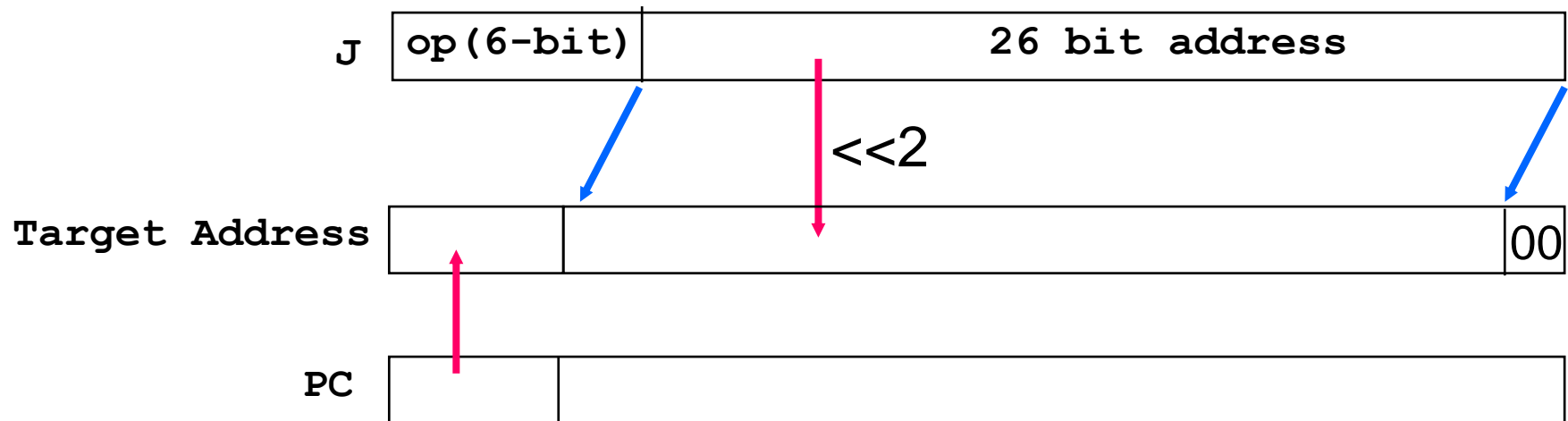
op=3	target=250 (word address)
------	---------------------------

What is
the use
of JAL
and JR?



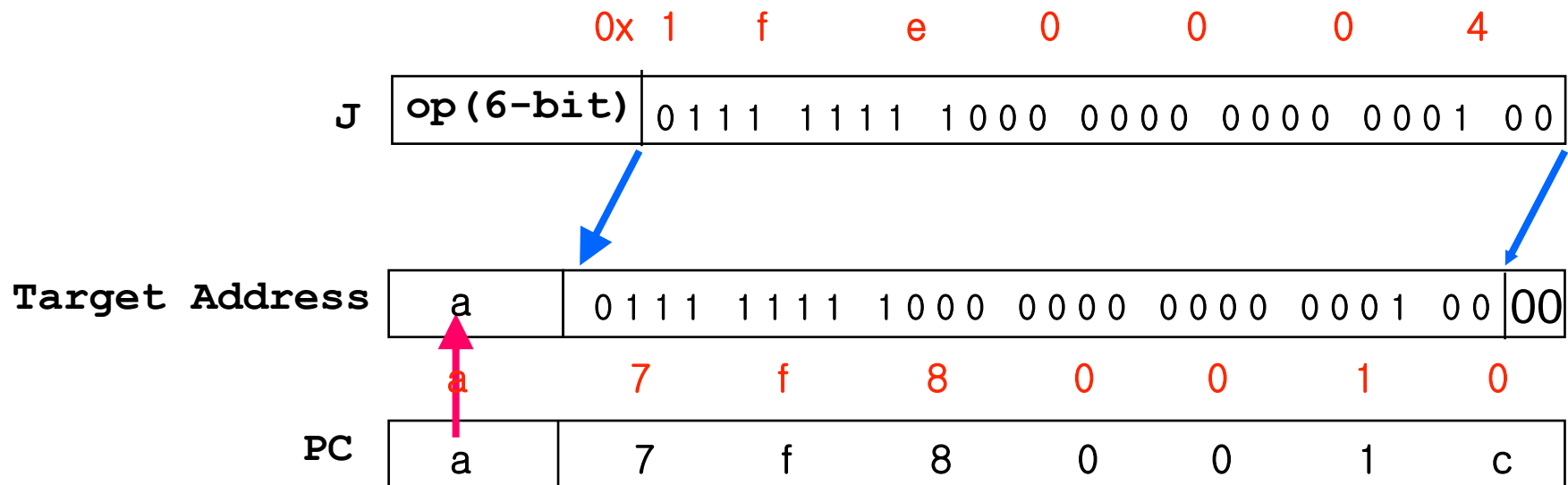
Addresses in Branches - Unconditional

- Jump instruction (j Label) just use high order bits of PC
 - Jump address boundaries of 256 MB ($2^{26+2}=2^8\text{MB}$)



Deciding target address

- TargetAddress = 0xa7f80010
- PC = 0xa7f8001c



- Beq / Bne \$t0, \$t1, 16-bit Offset

PC (Target Address) \leftarrow PC + Offset*4

- J / Jal 26-bit address

PC (Target Address) \leftarrow PC(31:28) && (address << 2)

- Jr \$t0

PC (Target Address) \leftarrow \$t0

Exercise

- **BEQ \$s0,\$s1,L**

The Range of L :

$-2^{15} \leq L \leq +2^{15}-1$ because *L* is 16-bits long

(Question) What if *L* is out of the range ?

(Answer) ➔

- **J target**

The target is *anywhere within a block of 256M (2^{28})* address where PC supplies the upper 4 bits.

(Question) What if target is out of 256M block of current PC points?

(Answer) ➔

Exercise

- (Question) What is the difference between `&&` (Logical AND) and `&` (Bit-wise AND) in C and how are they converted into MIPS machine language?



**AND instruction in MIPS corresponds to `&` in C.
Sequence of Conditional Branch instructions in MIPS corresponds to `&&` in C.**



Exercise

- What is the range of addresses for conditional branches when $PC=0x00200000$?
- What is the range of addresses for J and JAL instructions when $PC=0x80FEDCBA$?

Exercise

```
i = 0;  
while (save[i] == k)    i++;
```

\$s3 : i
\$s5 : k
\$s6 : base address of
the array 'save'



Instruction Summary

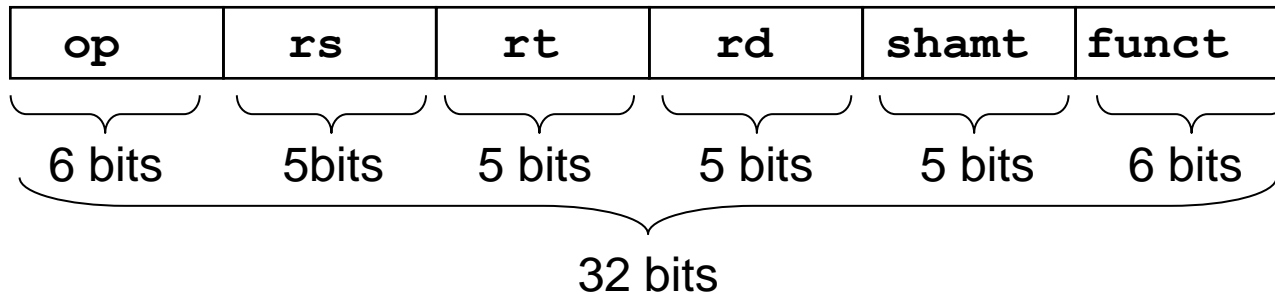
Category	Format	Instruction	Format	Meaning
Arithmetic	R	ADD	ADD rd,rs,rt	$rd \leftarrow rs + rt$
	R	SUB	SUB rd,rs,rt	$rd \leftarrow rs - rt$
	I	ADDI	ADDI rt,rs,imm16	$rt \leftarrow rs + se(imm16)$
	R	SLT	SLT rd,rs,rt	$rd \leftarrow (rs < rt) ? 1 : 0$
	I	SLTI	SLTI rt,rs, imm16	$rt \leftarrow (rs < se(imm16)) ? 1 : 0$
Logical	R	AND	AND rd,rs,rt	
	R	OR	OR rd,rs,rt	
	I	ANDI	ANDI rt,rs,imm16	$rt \leftarrow rs \& ze(imm16)$
	R	NOR	NOR rd, rs, rt	
	R	SLL	SLL rd, rs, shamt	
	R	SRL	SRL rd, rs, shamt	
Data Transfer	I	LW	LW rt, imm16(rs)	$rt \leftarrow M[rs+se(imm16)]$
	I	SW	SW rt, imm16(rs)	$M[rs+se(imm16)] \leftarrow rt$
Control Transfer	I	BEQ	BEQ rs,rt, imm16	if (rs == rt) $PC \leftarrow PC+se(imm16)*4$
	I	BNE	BNE rs,rt, imm16	If (rs != rt) $PC \leftarrow PC+se(imm16)*4$
	R	JR	JR rs	$PC \leftarrow rs$
	J	JAL	JAL imm26	$\$ra \leftarrow PC+4; PC \leftarrow imm26*4$
	J	J	J imm26	$PC \leftarrow imm26*4$

1. Introduction to Instruction Set Architecture
2. Binary Number Basic
3. MIPS Design Principles
4. MIPS Instruction Set
- 5. MIPS Instruction Format & Addressing Modes**
6. Supporting Procedure Call

- In this section, we will continue to discuss about the MIPS **Instruction Set Architecture**
- MIPS has three **instruction formats**.
- We will also discuss ‘**stored program concept**’ which is the basis of modern conventional computer design.
- We will see the instructions for large constants and addresses.
- MIPS has pseudo-instructions.
- We will summarize MIPS addressing modes.

Representing Instruction

- Assembly code should be represented in a binary code (Machine Language)
- Machine language structure:
 - Length : 32 bits (Design Principle 1)
 - Formats : 3 types
 - Fields :



(1) R-type Instruction Format

- R-type for arithmetic and logic instructions

R:	op	rs	rt	rd	shamt	funct
	6 bits	5bits	5 bits	5 bits	5 bits	6 bits
	Operation (opcode)	first source operand	second source operand	destination operand	shift amount	function code

ADD \$t0, \$s1, \$s2 # \$t0=R8, \$s1=R17, \$s2=R18

Decimal:

0	17	18	8	0	32
---	----	----	---	---	----

Binary:

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

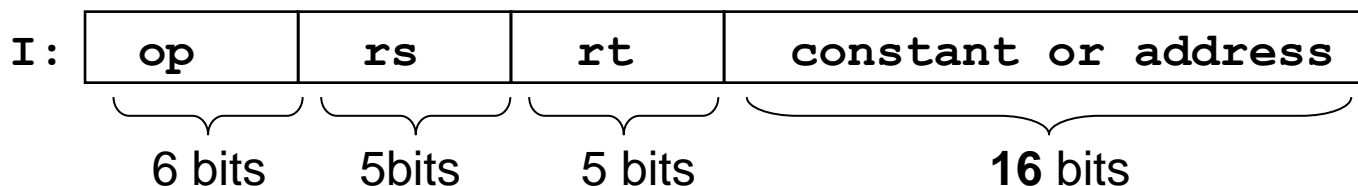
Hexadecimal:

0 2 3 2 4 0 2 0

→ 0x02324020

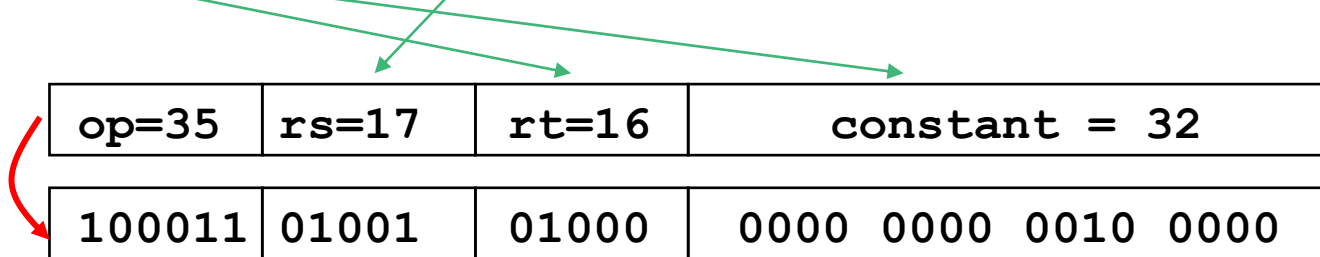
(2) I-type Instruction Format

- Problems with instructions that need longer fields for constant value (LW, SW, ADDI).
 - 5 bits field is too small to be useful as an array index.
 - We will assign 16bit for constant or index.
- I-type format : for immediate value

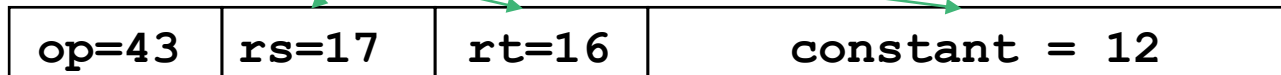


I-type Instruction Format

- **LW \$s0, 32 (\$s1)**



- **SW \$s0, 12 (\$s1)**

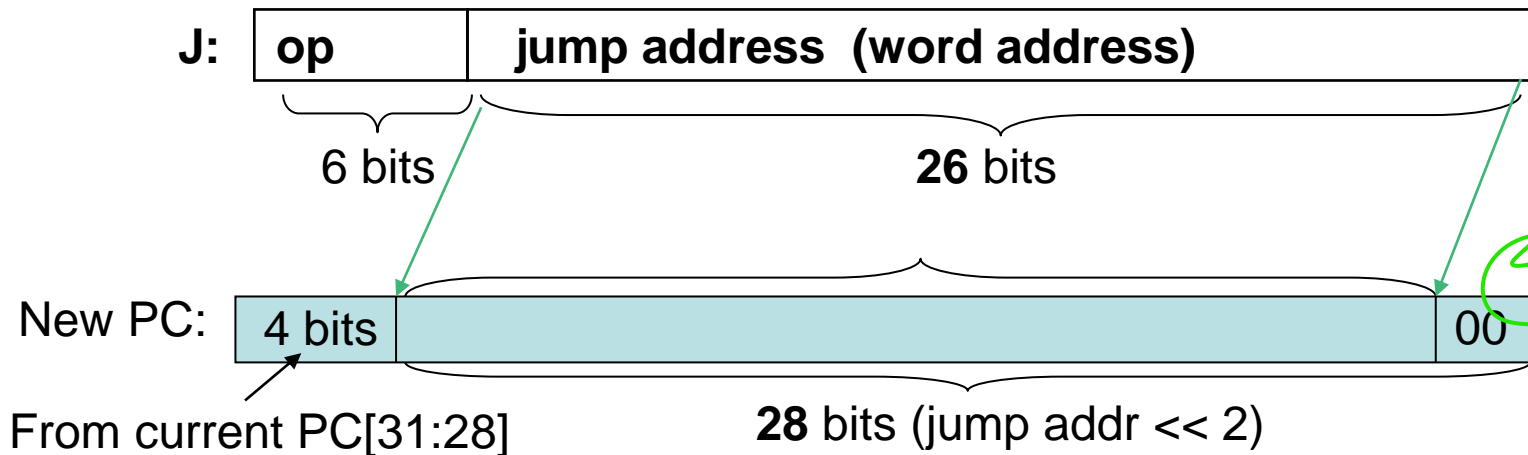


- **ADDI \$s0, \$s1, 4**



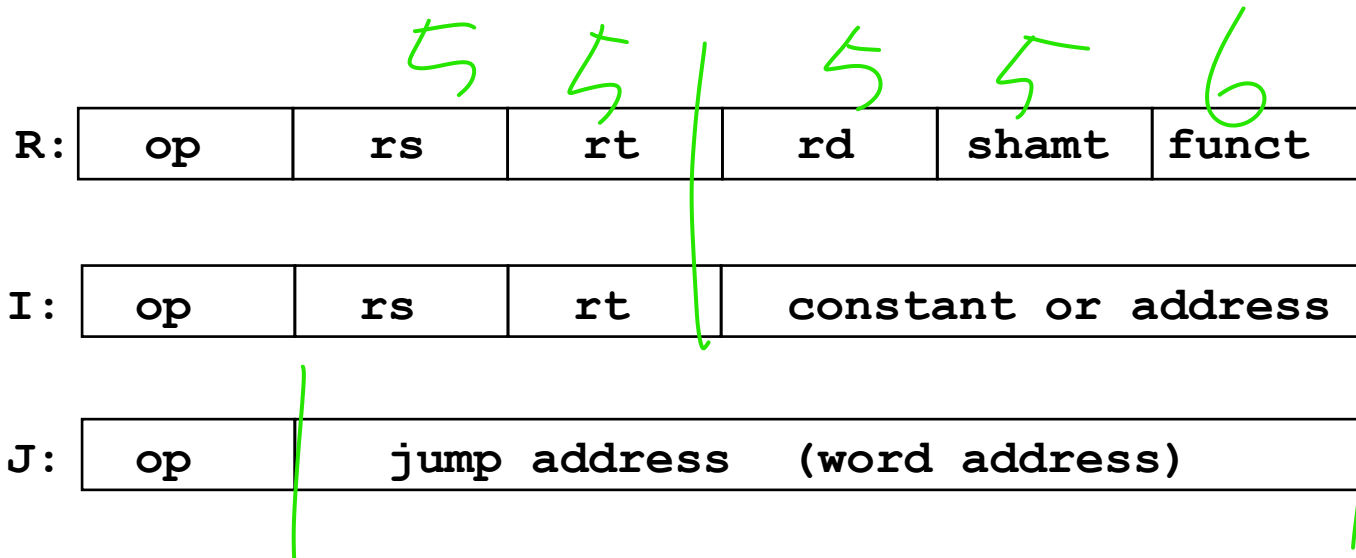
(3) J-type Instruction Format

- Jump address needs more than 16-bits
- J-type Instruction : **Jump instruction & JAL**



Instruction format Summary

- Three MIPS Instruction Formats:



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Fig. 2.17

op(31:26)								
28-26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31-29								
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						
op(31:26)=010000 (TLB), rs(25:21)								
23-21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
25-24								
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								
op(31:26)=000000 (R-format), funct(5:0)								
2-0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
5-3								
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jair			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Exercise

- Find machine code for the assembly code program.

```
Loop : add $t1,$s3,$s3
      add $t1,$t1,$t1
      add $t1,$t1,$s6
      lw $t0,0($t1)
      bne $t0,$s5, Exit
      add $s3, $s3, $s4
      j Loop
```

Exit :

\$s3 = \$19	\$t0=\$8
\$s4 = \$20	\$t1=\$9
\$s5 = \$21	\$s6=\$22

Exercise

- Convert the MIPS assembly language into MIPS machine language. And show them in hexadecimal numbers.

ADD \$s1,\$s2,\$s3

LW \$s1, 2(\$s2)

ADDI \$s1, \$s2,1

J 400

SW \$s1,4(\$zero)

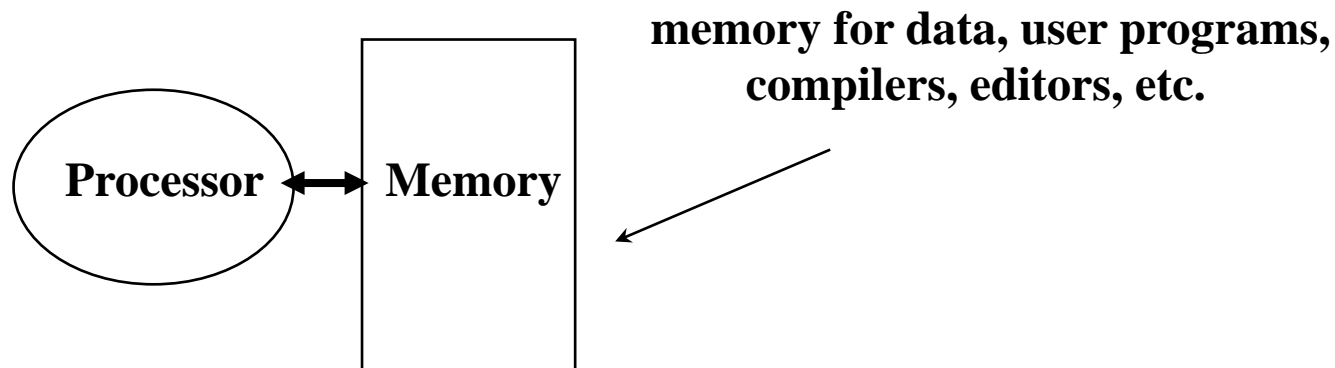
6-bits (opcode)

<div></div>	<div></div>
<div></div>	<div></div>
<div></div>	<div></div>
<div></div>	<div></div>
<div></div>	<div></div>

Stored Program Concept

- **Two Principles**

- Instructions are represented as numbers (indistinguishable from number *or* data)
- 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

- Control Transfer Instructions

- Alter the control flow,
- By changing the "next" instruction to be executed.

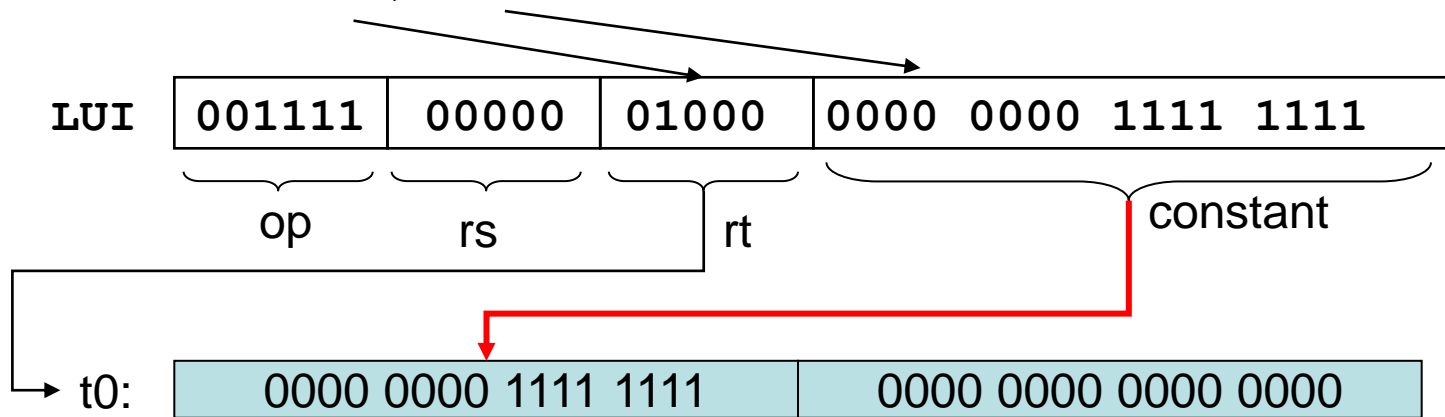
- PC register points to the memory location where the “next” Instruction resides.

Loading 32-bits Constants

- Keeping MIPS instructions 32-bits long is good for hardware simplicity. *But*, we need full 32-bits constants or address sometimes. → Conflict ?
- Either compiler or assembler must *break large constants into pieces* and then reassemble them into a register.
- MIPS provides LUI for loading upper 16-bits of a constant in a register.

Loading 32-bits Constants

- **LUI \$t0, 255** # \$t0 is R8



- **What is MIPS assembly code for 32-bit constant “0x003D 0900” into \$s0?**

(Ans) LUI \$t0, 0x003D → t0: 003D 0000

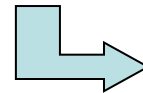
ORI \$s0,\$t0, 0x0900 → s0: 003D 0900

- Sign Extension
- Zero Extension



Pseudo Instructions

- **Pseudo instruction:** A common variation of original assembly language instructions often treated as if it were an instruction in its own right
- **COPY** from one register to another
 - Move \$t0, \$t1
→ add \$t0, \$t1, \$zero
- **Branch if Less Than**
 - BLT \$t0,\$t1,L # if (\$t0 < \$t1) goto L
 - Use \$at register as temporary
- **Branch if Greater Than**
 - BGT \$t0, \$t1, L # if (\$t0 > \$t1) goto L
- **Branch if Greater (Less) Than or Equal to**
 - BGE \$t0, \$t1, L # if(\$t0 >= \$t1) goto L



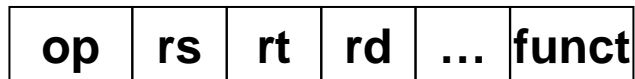
```
SLT $at, $t0, $t1  
BNE $at, $zero, L
```

MIPS Addressing Modes

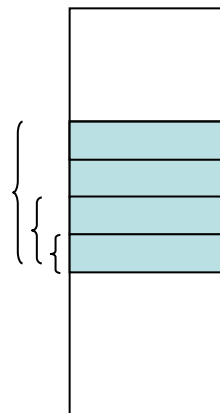
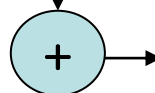
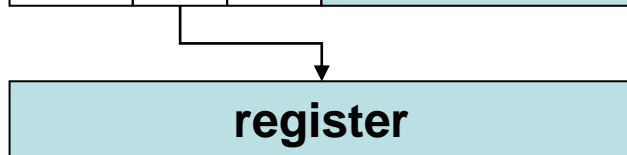
Immediate addressing



Register addressing



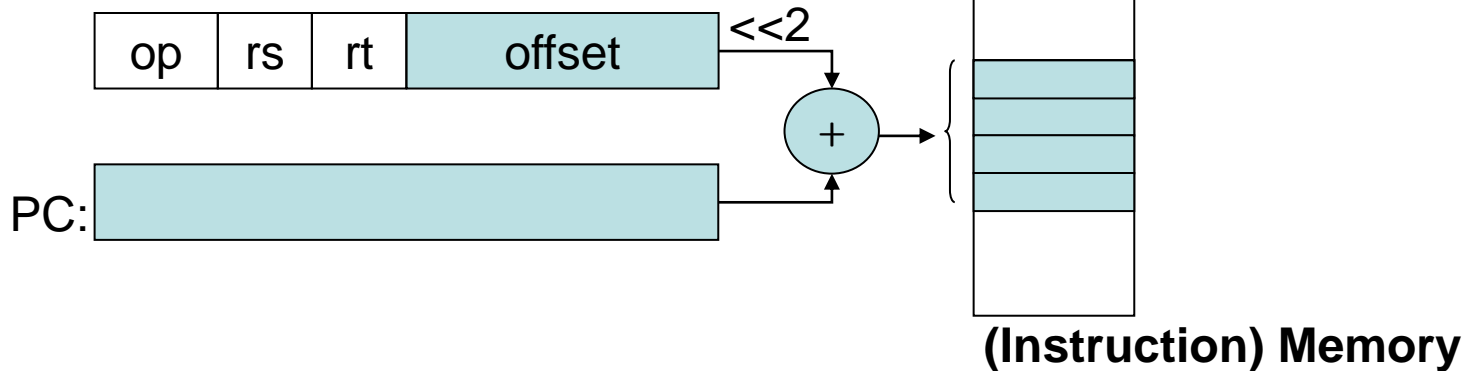
Base addressing



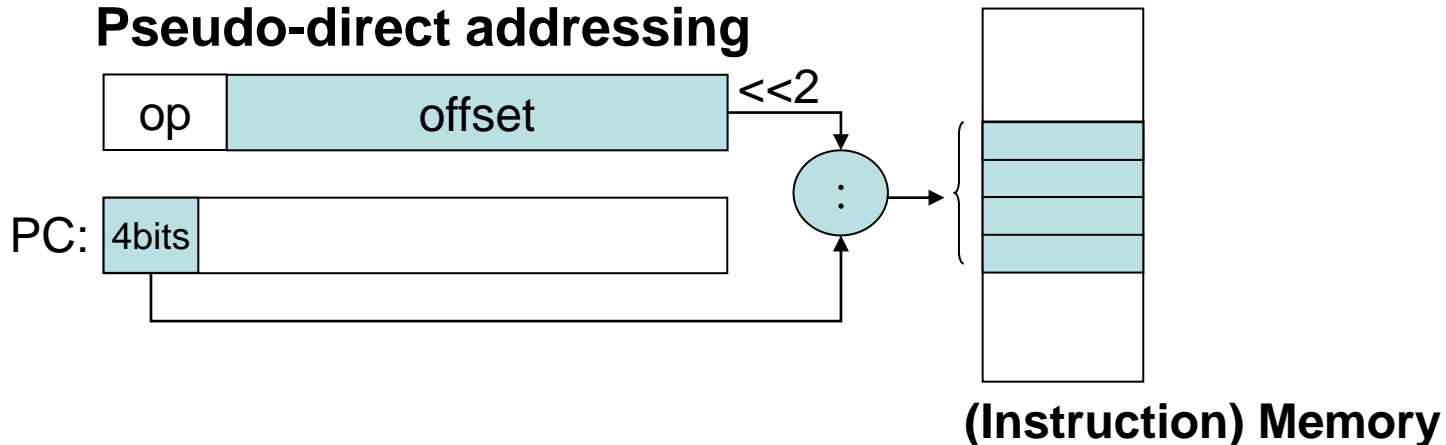
(data) Memory

MIPS Addressing Modes

PC-Relative addressing



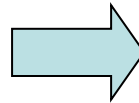
Pseudo-direct addressing



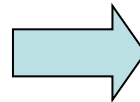
Exercise

- Show the MIPS Assembly Instruction sequence for the C code.

```
if (A > B)
    A = B;
else
    A = A + 1;
```



```
if (B <= 2048)
    A = A + 1024*10;
else
    A = 1024 * 12 + 32 - 1;
```



.text

main:

addi \$s0, \$zero, 0

addi \$t0, \$zero, 0

addi \$t1, \$zero, 10

loop:

slt \$t8, \$t0, \$t1

beq \$t8, \$zero, exit

addi \$t0, \$t0, 1

add \$s0, \$s0, \$t0

j loop

exit:

.end



SPIM

.data

digit:

.word 10, 12, 23, 28

str:

.asciiz " \n"

.text

main:

la \$s1, digit

lw \$t0, 0(\$s1)

lw \$t1, 4(\$s1)

lw \$t2, 8(\$s1)

lw \$t3, 12(\$s1)

addi \$t4, \$t0, 5

sw \$t4, 20(\$s1)

lw \$t0, 20(\$s1)

li \$v0, 1

add \$a0, \$t0, \$t1

syscall

li \$v0, 4

la \$a0, str

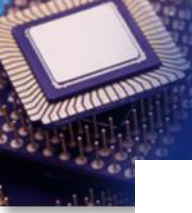
syscall

li \$v0, 1

sub \$a0, \$t2, \$t3

syscall

.end



SPIM System Call

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$a0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

FIGURE A.9.1 System services.

1. Introduction to Instruction Set Architecture
2. Binary Number Basic
3. MIPS Design Principles
4. MIPS Instruction Set
5. MIPS Instruction Format & Addressing Modes
- 6. Supporting Procedure Call**



Procedure Call Steps

1. **Place Parameters in a place where the procedure can access them** → Where?
2. **Transfer control to the procedure** → What Instruction ?
3. **Acquire the storage resources needed for the procedure** → How to get and access ?
4. **Perform the desired task**
5. **Place the result value in a place where the calling program can access it** → Where ?
6. **Return control to the point of origin** → What Instruction ?

- **MIPS Software Convention**
 - **\$a0 ~ \$a3** : four registers for arguments to pass parameters
 - **\$v0 ~ \$v1** : two registers for return values
 - **\$ra** : one return address

MIPS Instruction Pair for Procedures

- Procedure Call

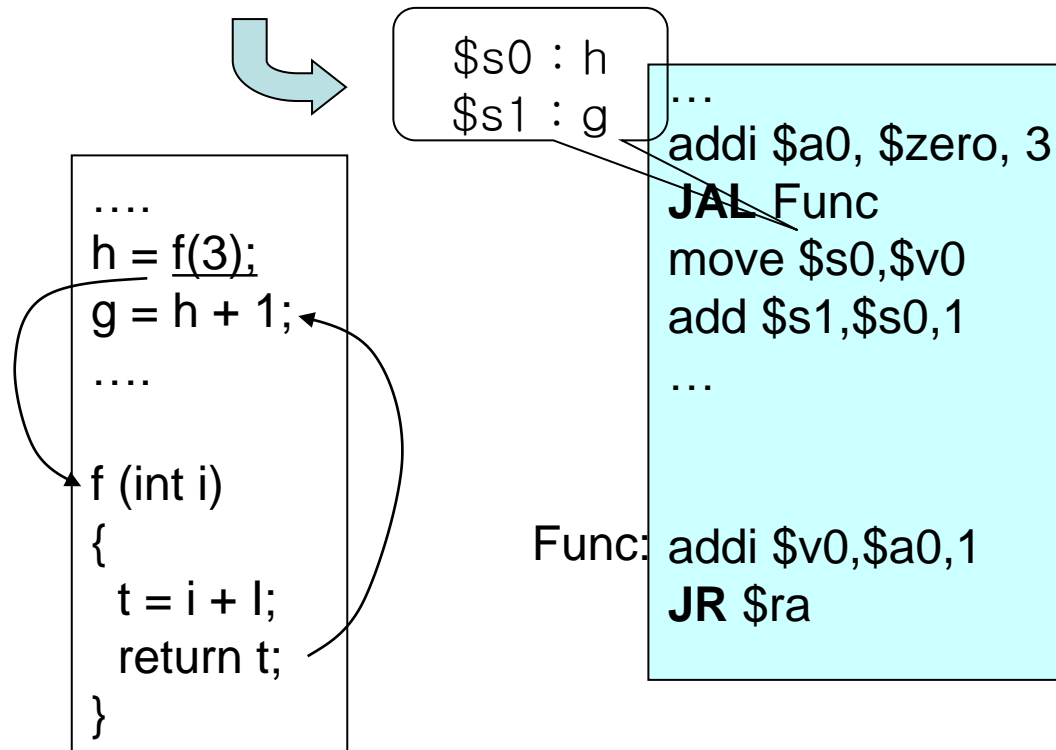
- JAL ProcedureAddr

PC \leftarrow ProcedureAddr

\$ra \leftarrow next instruction address (PC or PC+4)

- JR \$ra

PC \leftarrow \$ra





Procedures use More Registers by Stack

- Suppose a procedure *needs more registers* than four arguments registers and two return value registers?
 - Note any registers used by called should be reserved and restored
 - Stack (Last-In-First-Out) is an ideal place for spilling register
 - Stack Pointer (\$SP register in MIPS) points to most recently allocated places in the stack
 - PUSH and POP operations for get and put data in stack

Procedures use More Registers by Stack

```
int leaf_example(int g, int h, int i, int j) {  
    int f;  
    f = (g+h) - (i+j);  
    return f ;  
}
```

1. Save registers in Stack

{ Parameters are in \$a0 - \$a3
Return address is in \$ra

- Register \$s0 is assigned to local variable f.
- Use \$t0 and \$t1 for (g+h) and (i+j) value

2. Compute f=(g+h)-(i+j)

{ Reserve \$s0, \$t0, \$t1 in stack

3. Restore register values

4. Place the Return values in \$v0

{ Restores \$s0, \$t0, \$t1 from stack

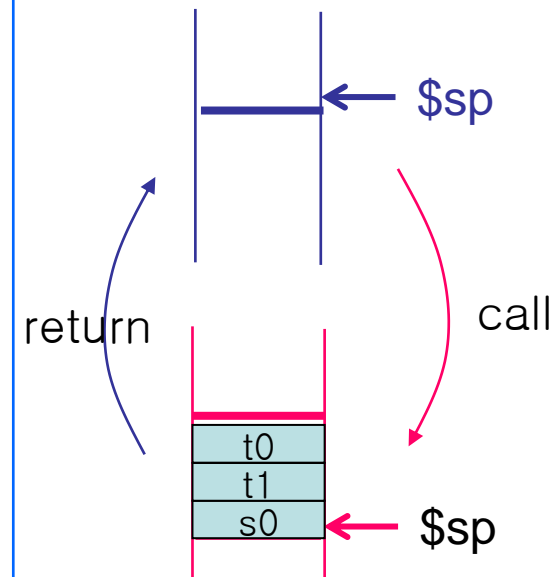
5. Return control to caller by JR \$ra

Compiling C Functions: Stack

```
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 # adjust stack to make room for 3 items  
sw $t0, 8($sp)   # save $t0 at MEM[sp+8]  
sw $t1, 4($sp)   # save $t1 at MEM[sp+4]  
sw $s0, 0($sp)   # save $s0 at MEM[sp]  
add $t0, $a0,$a1  # $s1 = g + h  
add $t1,$a2,$a3   # $s2 = i + j  
sub $s0,$t0,$t1   # $s0=$s1-$s2=(g+h)-(i+j)  
add $v0, $s0,$zero  
lw $s0, 0 ($sp)   # restore $s0 for caller  
lw $t1, 4 ($sp)   # restore $t1 for caller  
lw $t0, 8 ($sp)   # restore $t0 for caller  
addi $sp, $sp, 12 # pop 3 items  
jr $ra
```

\$a0 : g
\$a1 : h
\$a2 : i
\$a3 : j
\$ra : return address



MIPS Software Convention on Registers

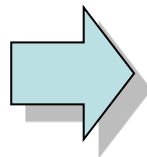
- **To Reduce the Register Spilling, MIPS Software Convention:**
 - **\$t0 - \$t9** : registers that are *not preserved by the callee* on a procedure call
 - **\$s0 - \$s7** : registers that *must be preserved* on a procedure call
- ➔ **The caller does not expect the \$t0 - \$t9 to be preserved across procedure call.**

Exercise :Reducing Register Spilling

- **Modify the code of leaf_example() to save time**

leaf_example :

```
addi $sp,$sp,-12 # adjust stack
sw $t0, 8($sp)    # M[$sp+8] ← $t0
sw $t1, 4($sp)    # M[$sp+4] ← $t1
sw $s0, 0($sp)    # M[$sp] ← $s0
add $t0, $a0,$a1  # $s1 = g + h
add $t1,$a2,$a3   # $s2 = i + j
sub $s0,$t0,$t1   # $s0=$s1-$s2
add $v0, $s0,$zero
lw $s0, 0 ($sp)   # $s0 ← M[$sp]
lw $t1, 4 ($sp)   # $t1 ← M[$sp+4]
lw $t0, 8 ($sp)   # $t0 ← M[$sp+8]
addi $sp, $sp, 12 # pop 3 items
jr $ra
```



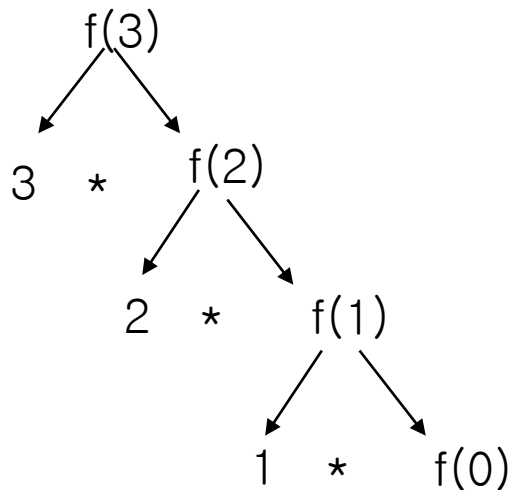


Nested Procedure Call

- **Caller saves (pushes onto the stack):**
 - Any argument register (\$a0-\$a3) that are needed after call
 - Any temporary registers (\$t0 - \$t9) that are needed after call
- **Callee saves (pushes onto the stack):**
 - return address (\$ra)
 - Any saved register (\$s0 - \$s7) used by the callee

Nested Procedure Call

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



fact :

```
addi $sp,$sp,-8 # adjust stack for 2 items
sw $ra, 4($sp) # saves return address
sw $a0, 0($sp) # saves argument n
slti $t0, $a0,1 # test if n < 1
beq $t0,$zero, L1 # if n >= 1 goto L1
addi $v0,$zero,1 # return 1
addi $sp,$sp,8 # pop 2 items
jr $ra # return to after jal
```

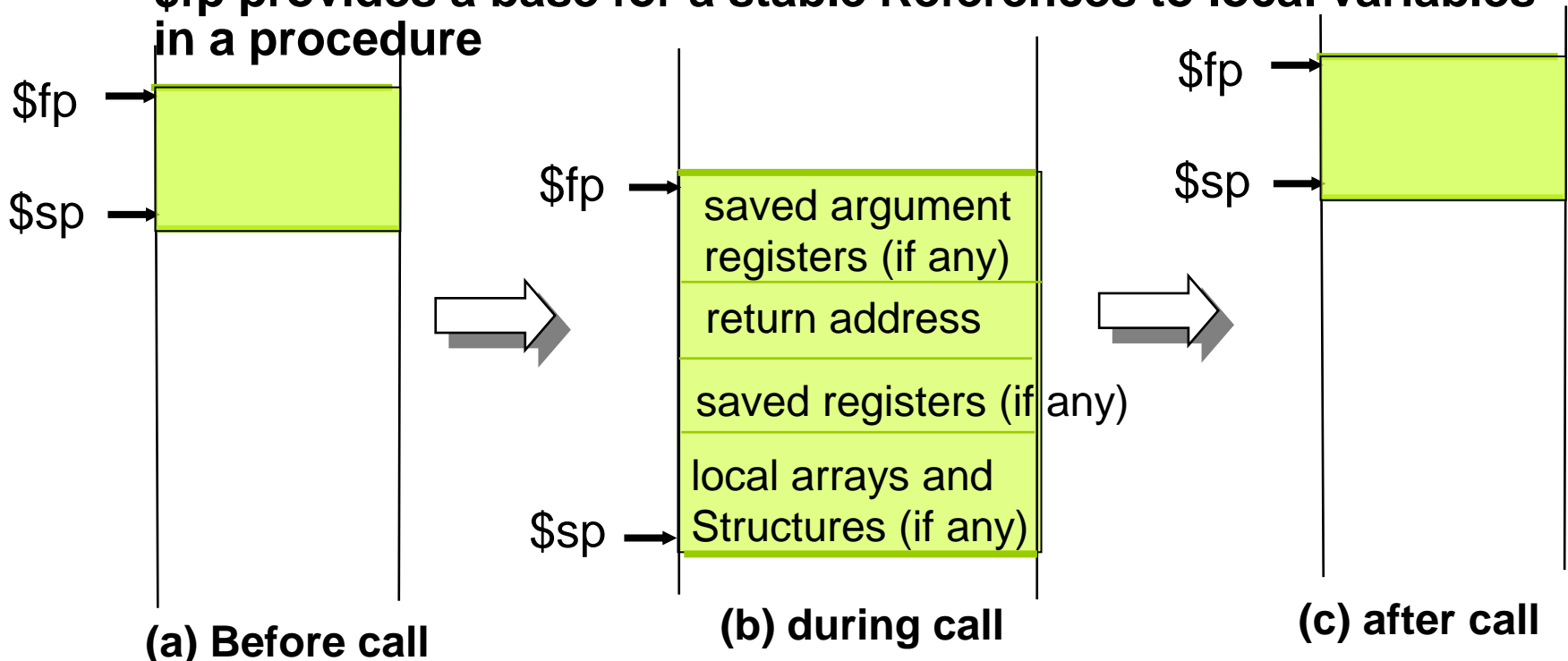
L1: addi \$a0,\$a0, -1 # n=n-1

```
jal fact # recursive call with (n-1)
lw $a0, 0 ($sp) # return from jal : restore n
lw $ra, 4 ($sp) # restore address
addi $sp,$sp,8 # adjust stack for 2 items
mul $v0, $a0,$v0 # return n * fact(n-1)
jr $ra # return to the caller
```

Allocating Space for New Data On the Stack

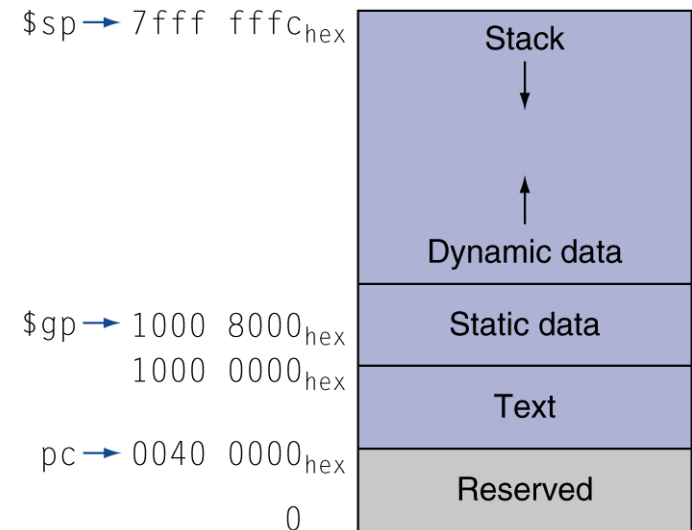
- **Procedure frame (Activation record)**

- A segment for stack containing a procedure's saved registers and local variables.
- Frame pointer (\$fp register) points to the first word of the procedure frame .
- \$fp provides a base for a stable References to local variables in a procedure



Memory Layout

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - \$gp initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage



- **MIP ISA Features Supporting Procedure Call**
 - **\$a0 - \$a3 : arguments**
 - **\$v0 - \$v1 : return value**
 - **\$ra : return address**
 - **Procedure Call instruction : JAL**
 - **Return from procedure call : JR \$ra**
- **Stacks are used for reserving registers and place for variables local to procedure**
 - **\$sp : stack pointer**
 - **\$fp : frame pointer**

Exercise : Procedure Call

- Convert the C program into MIPS assembly language program.

```
int f1 (int j, int k)
{
    int i; // $s0
    i = j+k+2;
    i = f2 (i);
    i = i << 4;
    return i;
}
```

```
int f2(int k)
{
    return (k<<1) - 2;
}
```