Computer Architecture

Instruction Set Architecture

Lynn Choi

Korea University

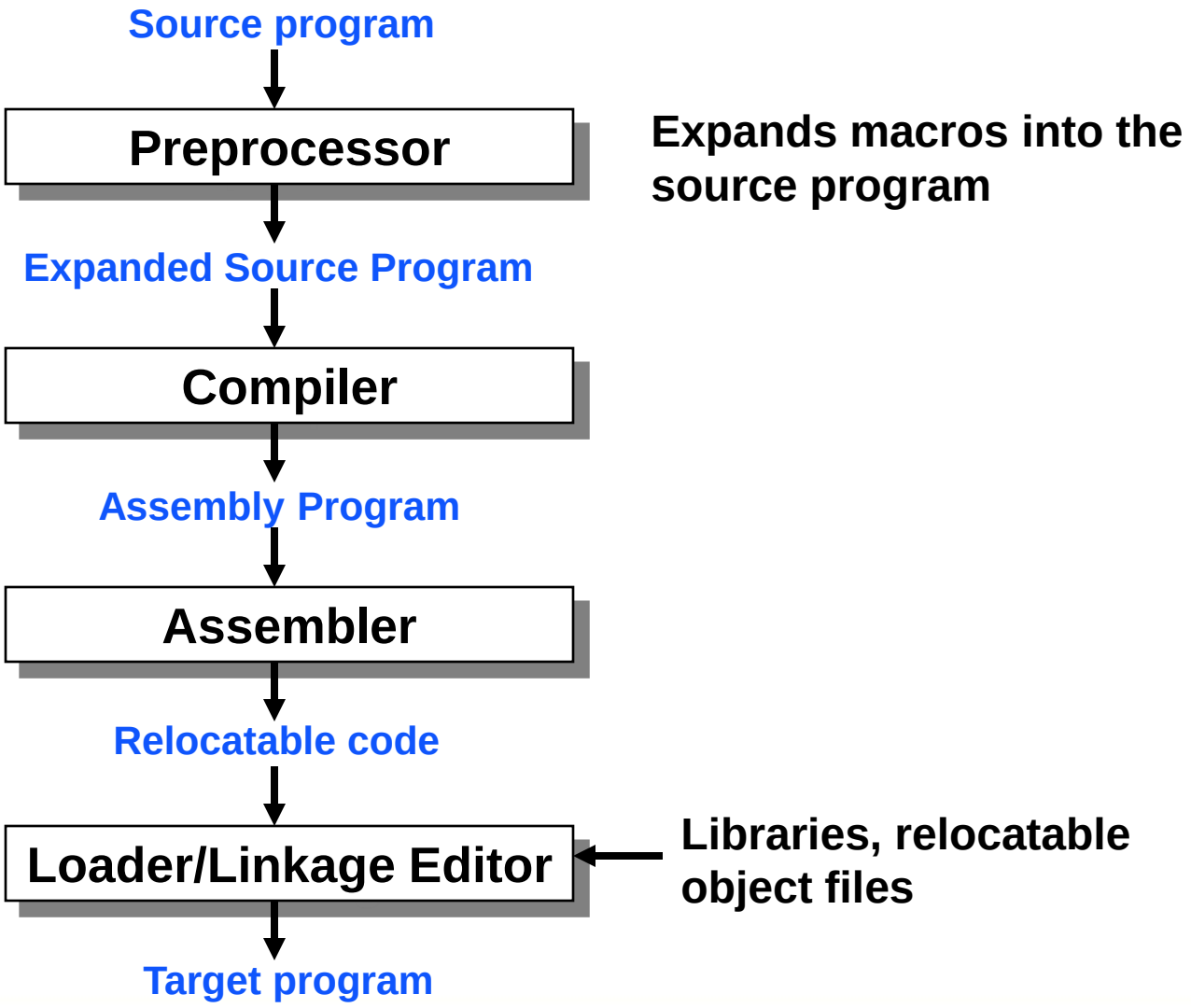
高麗大學校

Computer System Laboratory

Machine Language

- **Programming language**
  - High-level programming languages
    - Procedural languages: C, PASCAL, FORTRAN
    - Object-oriented languages: C++, Objective-C, Java
    - Functional languages: Lisp, Scheme
  - Assembly programming languages: symbolic machine languages
  - Machine languages: binary codes (1's and 0's)
- **Translator**
  - Compiler
    - Translates high-level language programs into machine language programs
    - Assembler: a part of a compiler
      - Translates assembly language programs into machine language programs
  - Interpreter
    - Translates and executes programs directly
    - Examples: JVM(Java virtual machine): translate/execute Java bytecode to native machine instructions

Compilation Process

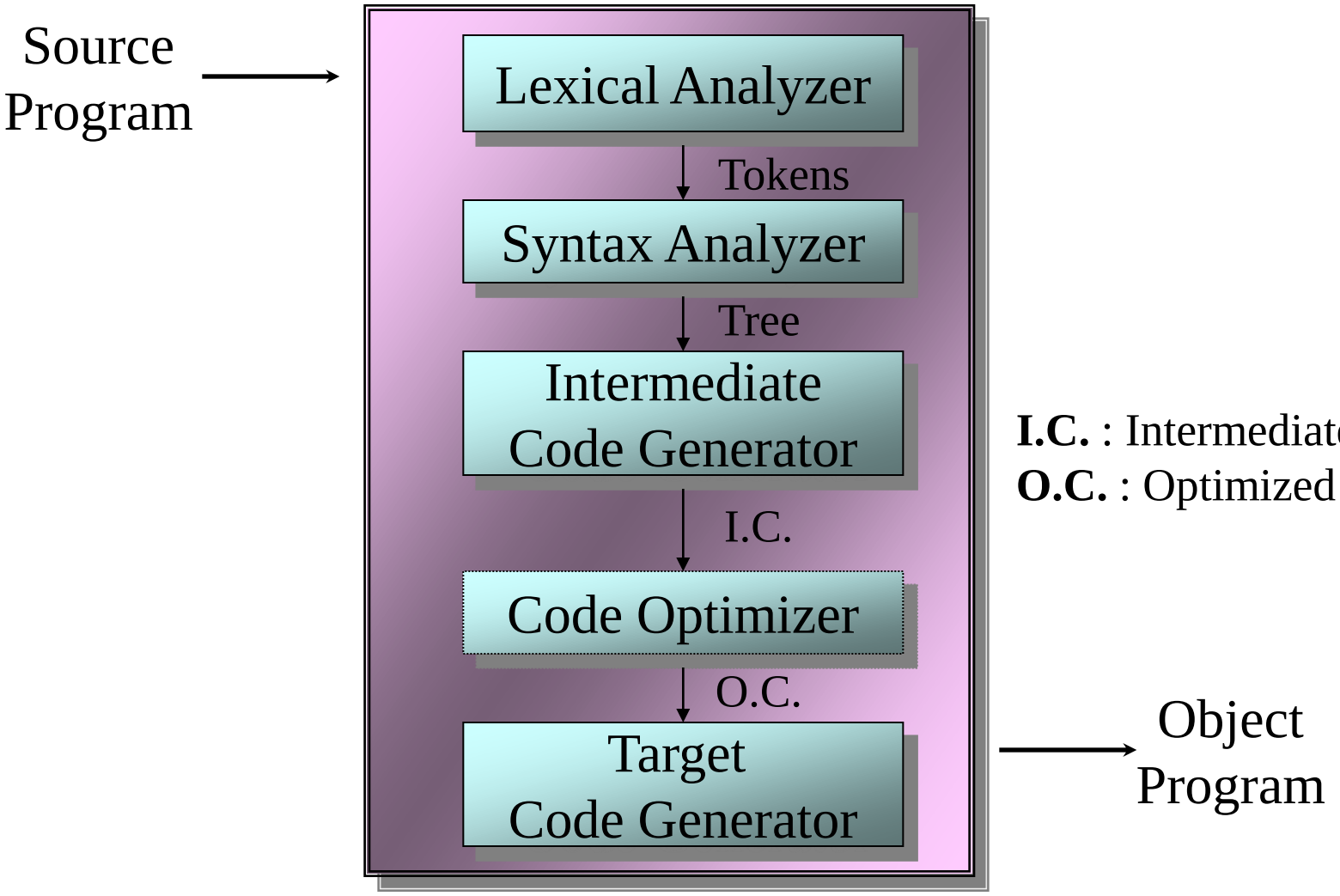


Compiler

- **Compiler**
  - A program that translates a source program (written in language A) into an equivalent target program (written in language B)

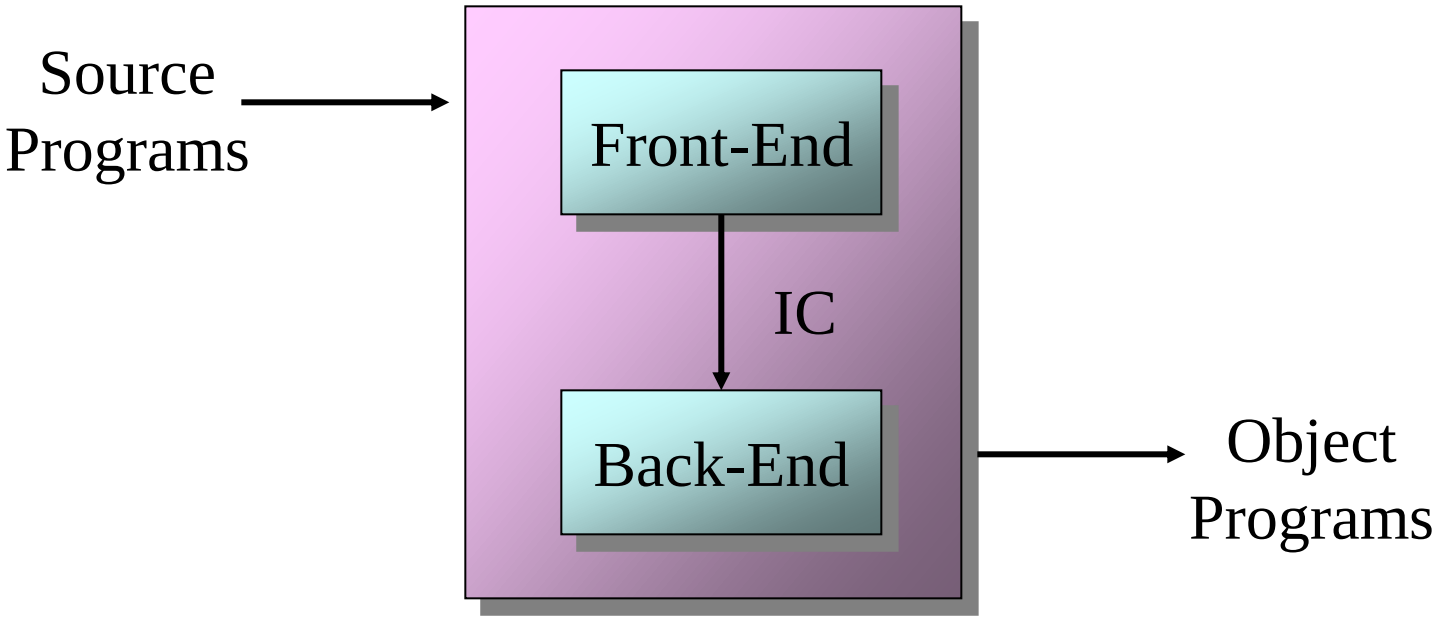
- **Source program**
  - Usually written in high-level programming languages (called ***source language***) such as C, C++, Java, FORTRAN
- **Target program**
  - Usually written in machine languages (called ***target language***) such as x86, Alpha, MIPS, SPARC, or ARM instructions
- **What qualities do you want in a compiler?**
  - Generate correct code
  - Target code runs fast
  - Compiler runs fast
  - Support for separate compilation, good diagnostics for errors

Compiler Phases



I.C. : Intermediate Code
O.C. : Optimized Code

Compiler Structure



Front-End : language dependent part
Back-End : machine dependent part

Machine State

- **ISA defines machine states and instructions**
- **Registers**
  - CPU internal storage to *store data* fetched from memory
    - Can be read or written in a single cycle
    - Arithmetic and logic operations are usually performed on registers
    - MIPS ISA has 32 32-bit registers: Each register consists of 32 flip-flops
  - Top level of the memory hierarchy
    - Registers <-> caches <-> memory <-> hard disk
      - Registers are visible to programmers and maintained by programmers
      - Caches are invisible to programmers and maintained by HW
- **Memory**
  - A large, single dimensional array, starting at address 0
    - To access a data item in memory, an instruction must supply an address.
  - Store programs (which contains both instructions and data)
  - To transfer data, use load (memory to register) and store (register to memory) instructions

Data Size & Alignment

- **Data size**
  - *Word* : the basic unit of data transferred between register and memory
    - 32b for 32b ISA, 64b for 64b ISA
  - *Double word*: 64b data, *Half word*: 16b data, *Byte*: 8b data
    - Load/store instructions can designate data sizes transferred: ldw, lddw, ldhw, ldb
- **Byte addressability**
  - Each byte has an address
- **Alignment**
  - Objects must start at addresses that are multiple of their size

Object addressed	Aligned addresses	Misaligned addresses
Byte	0, 1, 2, 3, 4, 5, 6, 7	Never
Half Word	0, 2, 4, 6	1, 3, 5, 7
Word	0, 4	1, 2, 3, 5, 6, 7
Double Word	0	1, 2, 3, 4, 5, 6, 7

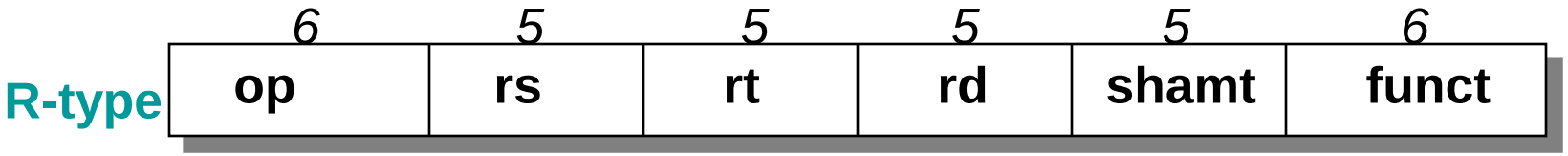
Machine Instruction

- **Opcode : specifies the operation to be performed**
  - EX) ADD, MULT, LOAD, STORE, JUMP
- **Operands : specifies the location of data**
  - Source operands (input data)
  - Destination operands (output data)
  - The location can be
    - Memory specified by a memory address : EX) 8(R2), x1004F
    - Register specified by a register number : R1

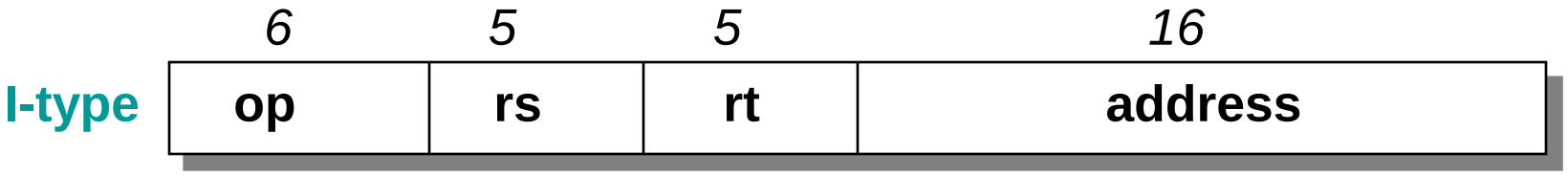
Instruction Types

- **Arithmetic and logic instructions**
  - Performs actual computation on operands
  - EX) ADD, MULT, SHIFT, FDIVIDE, FADD
- **Data transfer instructions (memory instructions)**
  - Move data from/to memory to/from registers
  - EX) LOAD, STORE
  - Input/Output instructions are usually implemented by memory instructions (memory-mapped IO)
    - IO devices are mapped to memory address space
- **Control transfer instructions (branch instructions)**
  - Change the program control flow
    - Specifies the next instruction to be fetched
  - Unconditional jumps and conditional branches
  - EX) JUMP, CALL, RETURN, BEQ

Instruction Format



- Op: Opcode, basic operation of the instruction
- Rs: 1st source register
- Rt: 2nd source register
- Rd: destination register
- shamt: shift amount
- funct: Function code, the specific variant of the opcode
- Used for arithmetic/logic instructions



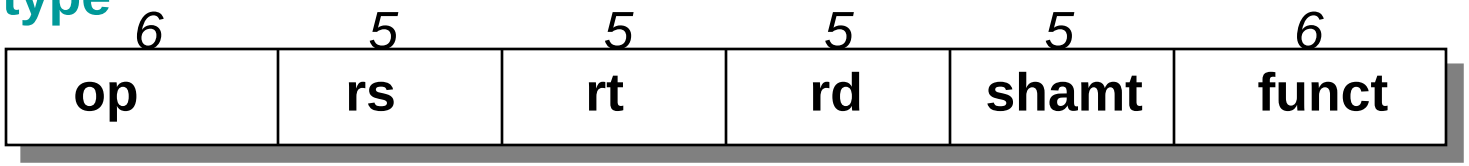
- Rs: *base register*
- Address: +/- 2^{15} bytes *offset* (or also called *displacement*)
- Used for loads/stores and conditional branches

MIPS Addressing Modes

- **Register addressing**
  - Address is in a register
  - Jr $ra
- **Base addressing**
  - Address is the sum of a register and a constant
  - Ldw $s0, 100($s1)
- **Immediate addressing**
  - For constant operand
  - Add $t1, $t2, 3
- **PC-relative addressing**
  - Address is the sum of PC and a constant (*offset*)
  - Beq $s0, $s1, L1
- **Pseudodirect addressing**
  - Address is the 26 bit offset concatenated with the upper bits of PC
  - J L1

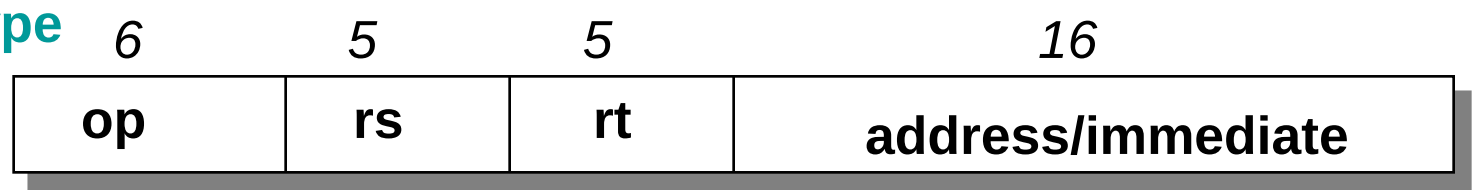
MIPS Instruction formats

R-type



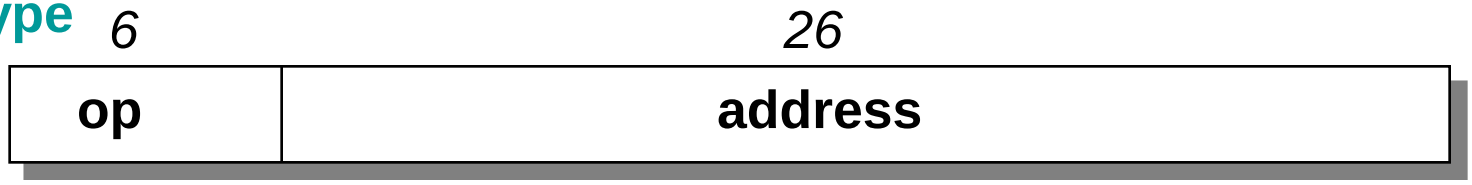
- Arithmetic instructions

I-type



- Data transfer, conditional branch, immediate format instructions

J-type



- Jump instructions

MIPS Instruction Example: R-format

- MIPS Instruction:
  - add $8,$9,$10

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

hex representation: 012A 4020$_{hex}$

decimal representation: 19,546,144$_{ten}$

- Called a Machine Language Instruction

MIPS Instruction Opcode Table

op(31:26)								
28–26 / 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	lh	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	sh	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26)=000000 (R-format), funct(5:0)								
2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	sll		srl	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	nor
5(101)			set l.t.	sltu				
6(110)								
7(111)								

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $1,$2,$3	$1 = $2 + $3	3 operands; data in registers
	subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; data in registers
Data Transfer	load word	lw $1,100($2)	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,100($2)	Memory[$2+100] = $1	Data from register to memory
Conditional Branch	branch on equal	beq $1,$2,L	if ($1 == $2) go to L	Equal test and branch
	branch on not eq.	bne $1,$2,L	if ($1 != $2) go to L	Not equal test and branch
	set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0	Compare less than; for beq,bne
Unconditional Jump	jump	j 10000	go to 10000	Jump to target address
	jump register	jr $31	go to $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4; go to 10000	For procedure call

MIPS machine language

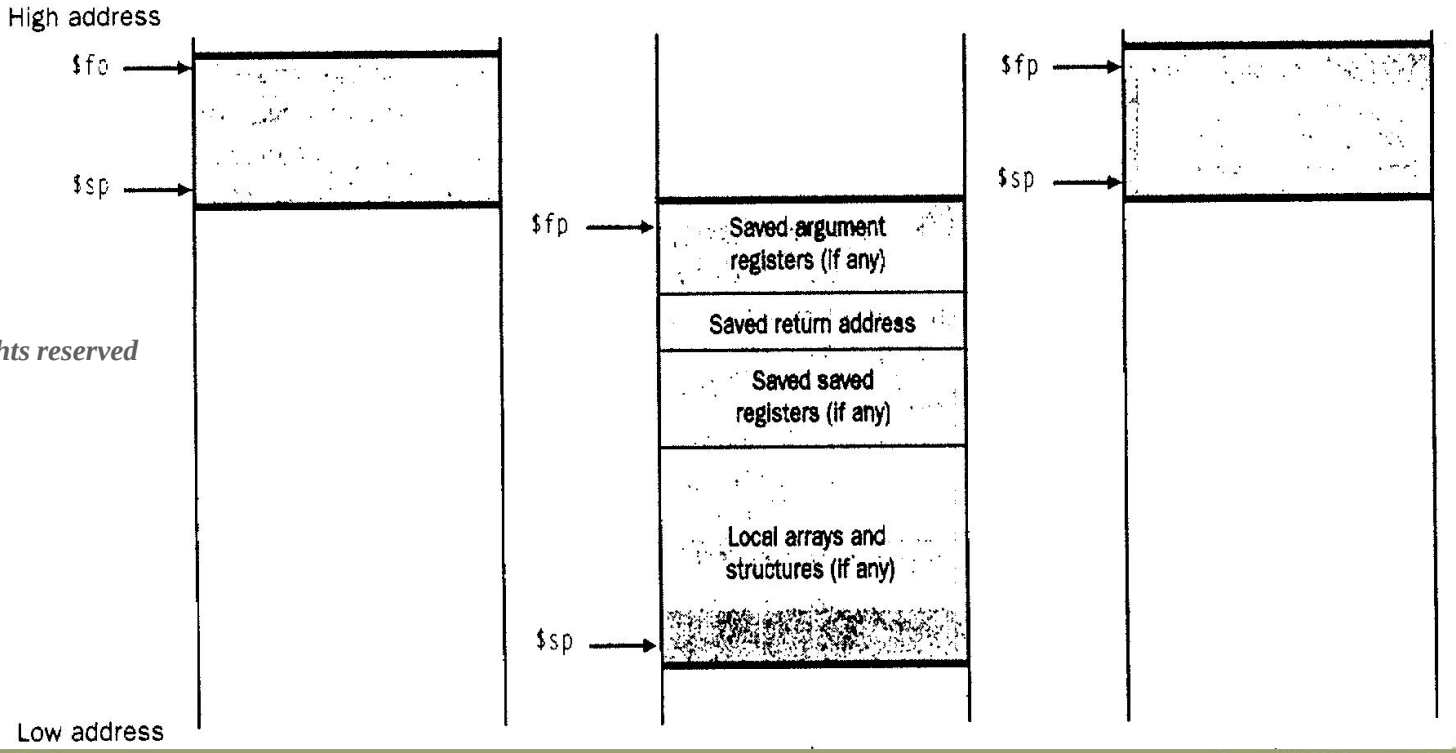
Name	Format	Example						Comments
add	R	0	2	3	1	0	32	add $1,$2,$3
sub	R	0	2	3	1	0	34	sub $1,$2,$3
lw	I	35	2	1	100			lw $1,100($2)
sw	I	43	2	1	100			sw $1,100($2)
beq	I	4	1	2	100			beq $1,$2,100
bne	I	5	1	2	100			bne $1,$2,100
slt	R	0	2	3	1	0	42	slt $1,$2,$3
j	J	2	10000					j 10000 (see section 3.7)
jr	R	0	31	0	0	0	8	jr $1
jal	J	3	10000					jal 10000 (see section 3.7)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
Format R	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
Format I	I	op	rs	rt	address			Data transfer, branch format

Procedure Call & Return

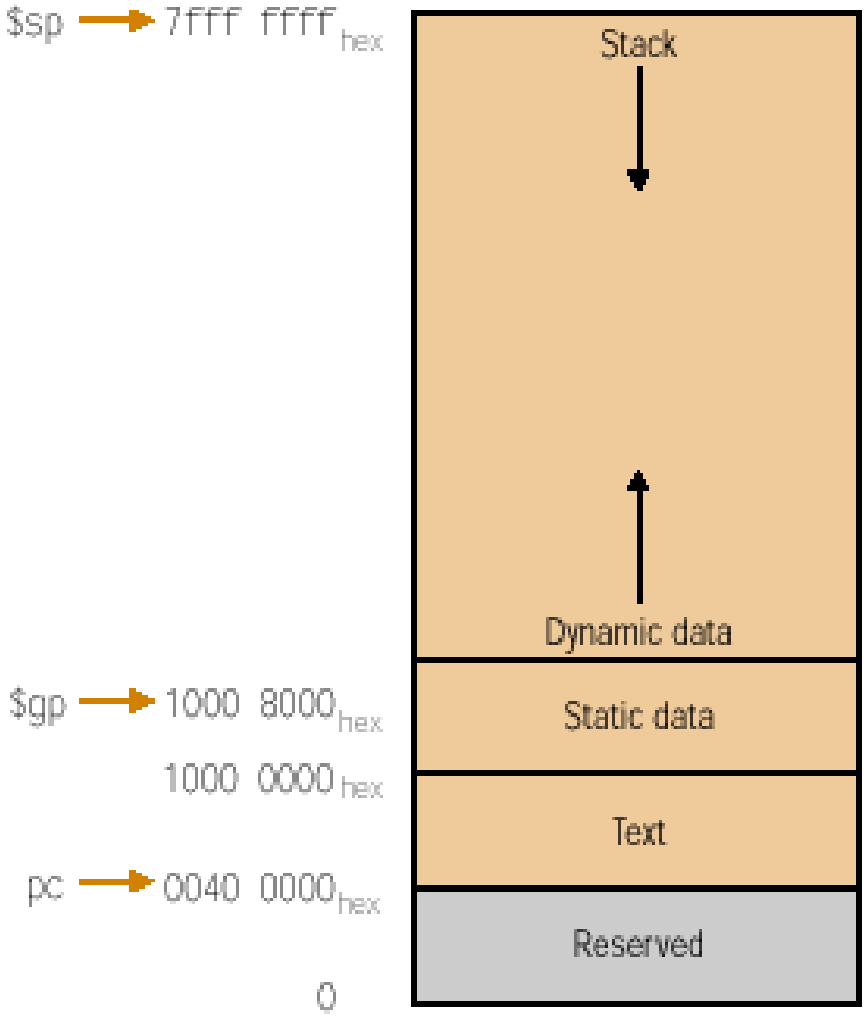
- **Steps of procedure call & return**
  - Place parameters in a place where the callee can access
    - $a0 - $a3: four argument registers
  - Transfer control to the callee
    - Jal callee_address : Jump and link instruction
      - put return address (PC+4) in $ra and jump to the callee
  - Acquire the storage needed for the callee
  - Perform the desired task
  - Place the result value in a place where the caller can access
    - $v0 - $v1: two value registers to return values
  - Return control to the caller
    - Jr $ra

Stack

- **Stack frame (activation record) of a procedure**
  - Store variables local to a procedure
    - Procedure's saved registers (arguments, return address, saved registers, local variables)
    - Stack pointer : points to the top of the stack



MIPS Memory Allocation



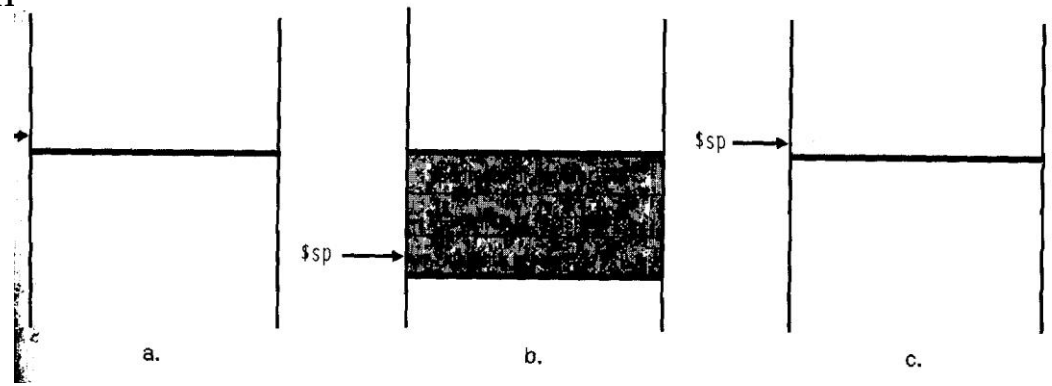
MIPS Register Convention

Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0–$v1	2–3	values for results and expression evaluation	no
$a0–$a3	4–7	arguments	yes
$t0–$t7	8–15	temporaries	no
$s0–$s7	16–23	saved	yes
$t8–$t9	24–25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

FIGURE 3.13 MIPS register convention. Register 1, called $at, is reserved for the assembler (see section 3.9), and registers 26–27, called $k0–$k1, are reserved for the operating system.

MIPS Example : Procedure

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



- **Assembly code**

```
leaf_example:
      sub $sp, $sp, 8
      sw $t1, 4($sp)          # save register $t1, $t0 onto stack
      sw $t0, 0($sp)
      add $t0, $a0, $a1       # $t0 = g + h
      add $t1, $a2, $a3       # $t1 = i + j
      sub $v0, $t0, $t1       # $v0 = (g + h) – (i + j)
      lw $t0, 0($sp)    # restore $t0, $t1 for caller
      lw $t1, 4($sp)
      add $sp, $sp, 8
      jr $ra
```

MIPS Example : Recursion

```
Int   fact  (int n)
{      if (n <2) return 1;
      else return n * fact (n – 1); }
```

- **Assembly code**

```
fact:  addi $sp, $sp, -8 # adjust stack pointer for 2 items
       sw $ra, 4($sp)                  # save return address and argument n
       sw $a0, 0($sp)
       slt $t0, $a0, 2     # if n < 2, then $t0 = 1
       beq $t0, $zero, L1              # if n >=2, go to L1
       addi $v0, $zero, 1              # return 1
       addi $sp, $sp, 8  # pop 2 items off stack
       jr $ra
L1:    addi $a0, $a0, -1               # $a0 = n - 1
       jal fact                        # call fact(n – 1)
       lw $a0, 0($sp)                  # pop argument n and return address
       lw $ra, 4($sp)
       addi $sp, $sp, 8                #
       mul $v0, $a0, $v0      # return n * fact(n – 1)
       jr $ra
```

Homework 2

- **Read Chapter 7 (from Computer Systems textbook)**
- **Exercise**
  - 2.2
  - 2.4
  - 2.5
  - 2.8
  - 2.12
  - 2.19
  - 2.27