Instruction Set Architecture

Outline

- Instruction set architecture (taking MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

What Is Computer Architecture?

Computer Architecture =
Instruction Set Architecture
+ Machine Organization

- "... the attributes of a [computing] system as seen by the [__**assembly**____ language] programmer, *i.e.* the conceptual structure and functional behavior …"

What are specified?

Recall in C Language

- Operators: `+`, `-`, `*`, `/`, `%` (mod), ...
  - `7/4==1`, `7%4==3`
- Operands:
  - Variables: `lower`, `upper`, `fahr`, `celsius`
  - Constants: `0`, `1000`, `-17`, `15.4`
- Assignment statement:

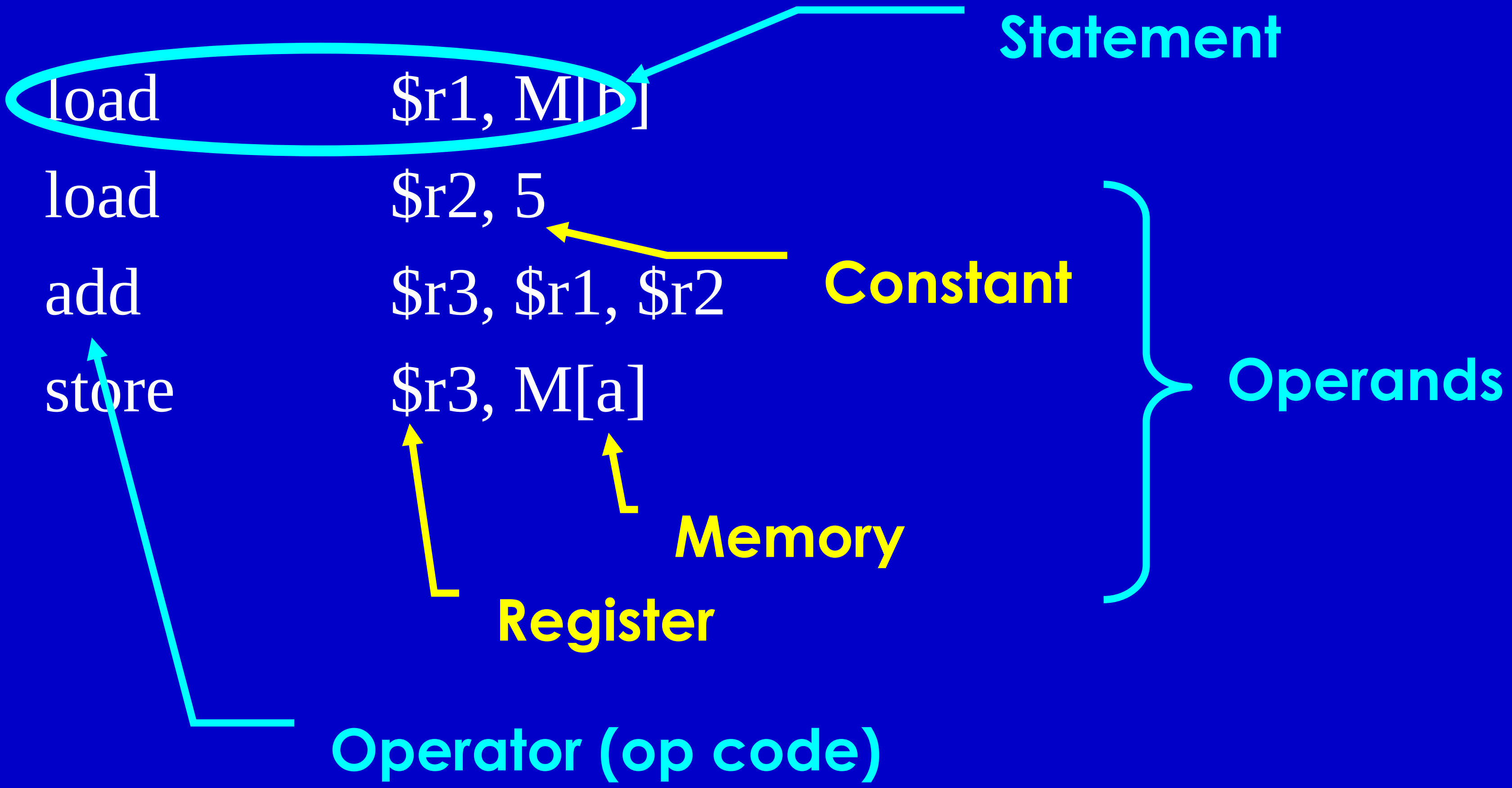
  variable = expression

  - Expressions consist of operators operating on operands, e.g.,

```
celsius = 5*(fahr-32)/9;
a = b+c+d-e;
```

When Translating to Assembly ...

a = b + 5;



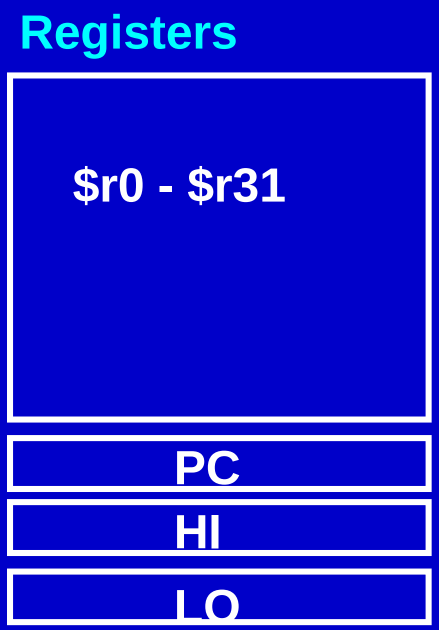
```
load        $r1, M[b]
load        $r2, 5
add         $r3, $r1, $r2
store       $r3, M[a]
```

Components of an ISA

- Organization of programmable storage
  – registers
  – memory: flat, segmented
  – Modes of addressing and accessing data items and instructions
- Data types and data structures
  – encoding and representation (next chapter)
- Instruction formats
- Instruction set (or operation code)
  – ALU, control transfer, exceptional handling

MIPS ISA as an Example

- Instruction categories:
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide

OP	$rs	$rt	$rd	sa	funct

OP	$rs	$rt	immediate

OP	jump target

Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

Operands and Registers

- Unlike high-level language, assembly don’t use variables => assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations are performed on these
- Benefits:
  - Registers in hardware => faster than memory
  - Registers are easier for a compiler to use
    - e.g., as a place for temporary storage
  - Registers can hold variables to reduce memory traffic and improve code density (since register named with fewer bits than memory location)

MIPS Registers

- 32 registers, each is 32 bits wide
  - Why 32? smaller is faster
  - Groups of 32 bits called a *word* in MIPS
  - Registers are numbered from 0 to 31
  - Each can be referred to by number or name
  - Number references:

    `$0, $1, $2, … $30, $31`
  - By convention, each register also has a name to make it easier to code, e.g.,

    `$16 - $22` ➔ `$s0 - $s7` (C variables)

    `$8 - $15` ➔ `$t0 - $t7` (temporary)
- 32 x 32-bit FP registers (paired DP)
- Others: HI, LO, PC

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Fig. 2.18

MIPS R2000 Organization

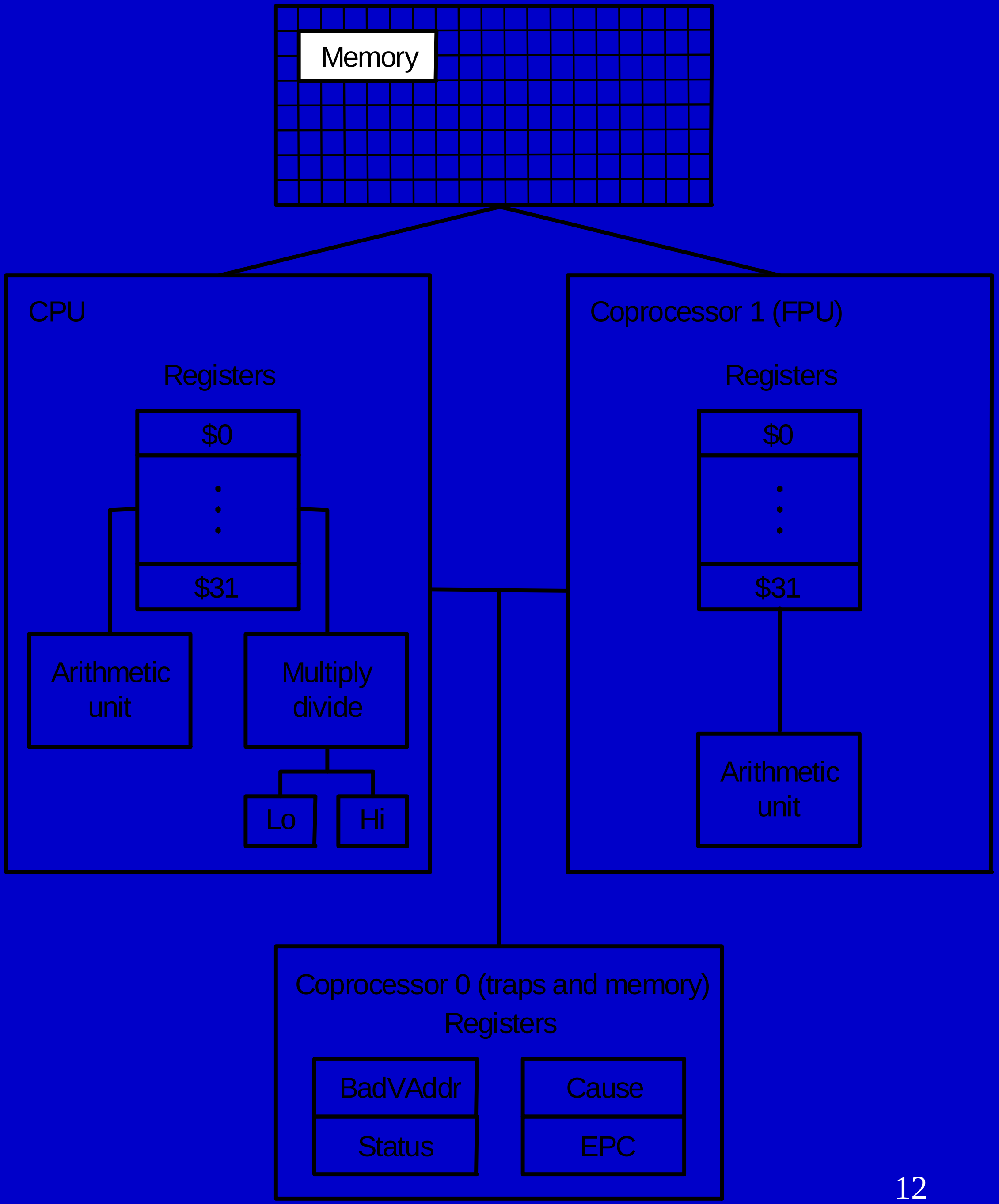


Fig. A.10.1

Operations of Hardware

- Syntax of basic MIPS arithmetic/logic instructions:

```
    1    2     3     4
add $s0,$s1,$s2       # f = g + h
```

1) operation by name

2) operand getting result ("destination")

3) 1st operand for operation ("source1")

4) 2nd operand for operation ("source2")

- Each instruction is 32 bits
- Syntax is rigid: 1 operator, 3 operands
  – Why? Keep hardware simple via regularity

Example

- How to do the following C statement?

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

use intermediate temporary register `t0`

```
add $s0,$s1,$s2# f = g + h
add $t0,$s3,$s4# t0 = i + j
sub $s0,$s0,$t0# f=(g+h)-(i+j)
```

Register Architecture

- Accumulator (1 register):

1 address:	add A	//acc ← acc + mem[A]
1+x address:	addx A	//acc ← acc + mem[A+x]

- Stack:

0 address:	add	//tos ← tos + next

- General Purpose Register:

2 address:	add A,B	//EA(A) ← EA(A) + EA(B)
3 address:	add A,B,C	//EA(A) ← EA(B) + EA(C)

- Load/Store: (a special case of GPR)

3 address:	add $ra,$rb,$rc	//$ra ← $rb + $rc
	load $ra,$rb	//$ra ← mem[$rb]
	store $ra,$rb	//mem[$rb] ← $ra

Register Organization Affects Programming

Code for C = A + B for four register organizations:

Stack	Accumulator	Register (reg-mem)	Register (load-store)
Push A	Load A	Load $r1,A	Load $r1,A
Push B	Add B	Add $r1,B	Load $r2,B
Add	Store C	Store C,$r1	Add $r3,$r1,$r2
Pop C			Store C,$r3

=> Register organization is an attribute of ISA!

Comparison: Byte per instruction? Number of instructions? Cycles per instruction?

Since 1975 all machines use GPRs

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

Memory Operands

- C variables map onto registers; what about large data structures like arrays?
  - Memory contains such data structures
- But MIPS arithmetic instructions operate on registers, not directly on memory
  - Data transfer instructions (lw, sw, ...) to transfer between memory and register
  - A way to address memory operands

Data Transfer: Memory to Register (1/2)

- To transfer a word of data, need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a 1D array
    - Address it by supplying a pointer to a memory address
    - Offset (in bytes) from this pointer
    - The desired memory address is the sum of these two values, e.g., `8($t0)`
    - Specifies the memory address pointed to by the value in `$t0`, plus 8 bytes (why “bytes”, not “words”?)
    - Each address is 32 bits

Data Transfer: Memory to Register (2/2)

- Load Instruction Syntax:

```
 1   2   3   4
lw $t0,12($s0)
```

  1) operation name

  2) register that will receive value

  3) numerical offset in bytes

  4) register containing pointer to memory

- Example: `lw $t0,12($s0)`
  - lw (Load Word, so a word (32 bits) is loaded at a time)
  - Take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`
- Notes:
  - `$s0` is called the *base register*, 12 is called the *offset*
  - Offset is generally used in accessing elements of array: base register points to the beginning of the array

Data Transfer: Register to Memory

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Example: `sw $t0,12($s0)`
  - sw (meaning Store Word, so 32 bits or one word are loaded at a time)
  - This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into the memory address pointed to by the calculated sum

Compilation with Memory

- Compile by hand using registers:
  `$s1`:g, `$s2`:h, `$s3`:base address of A

```
        g = h + A[8];
```

- What offset in `lw` to select an array element `A[8]` in a C program?
  - 4x8=32 bytes to select `A[8]`
  - 1st transfer from memory to register:

```
  lw   $t0,32($s3)      # $t0 gets A[8]
```

  - Add 32 to `$s3` to select `A[8]`, put into `$t0`

- Next add it to h and place in g

```
  add $s1,$s2,$t0      # $s1 = h+A[8]
```

Addressing: Byte versus Word

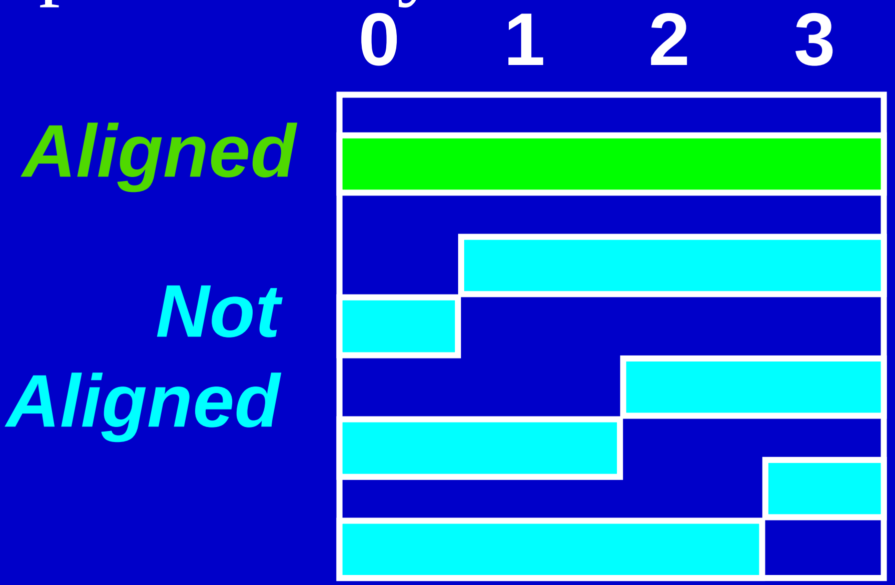
- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], …

Called the "**address**" of a word

- Computers need to access 8-bit bytes as well as words (4 bytes/word)
- Today, machines address memory as bytes, hence word addresses differ by 4
  - `Memory[0]`, `Memory[4]`, `Memory[8]`, …
  - This is also why lw and sw use bytes in offset

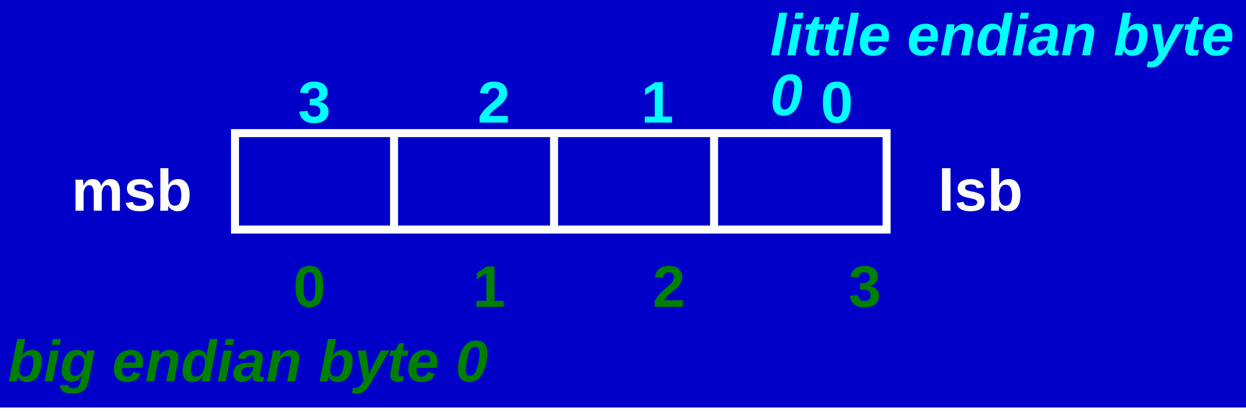
A Note about Memory: Alignment

- MIPS requires that all words start at addresses that are multiples of 4 bytes



Another Note: Endianess

- Byte order: numbering of bytes within a word
- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (00xx = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

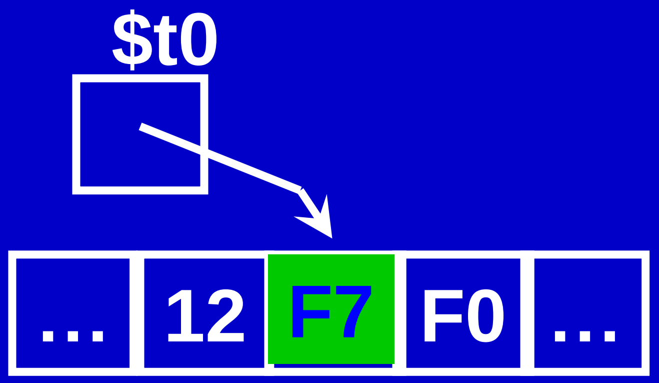


MIPS Data Transfer Instructions

Instruction		Comment
sw	$t3,500($t4)	Store word
sh	$t3,502($t2)	Store half
sb	$t2,41($t3)	Store byte
lw	$t1, 30($t2)	Load word
lh	$t1, 40($t3)	Load halfword
lhu	$t1, 40($t3)	Load halfword unsigned
lb	$t1, 40($t3)	Load byte
lbu	$t1, 40($t3)	Load byte unsigned
lui	$t1, 40	Load Upper Immediate (16 bits shifted left by 16)

What does it mean?

Load Byte Signed/Unsigned



lb $t1, 0($t0)

$t1

FFFFFF F7

Sign-extended

lbu $t2, 0($t0)

$t2

000000F7

Zero-extended

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
  - Writes less common variables to memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster:
    registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction
    - MIPS data transfers only read or write 1 operand per instruction, and no operation

Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer, and addressing
  - Immediate operands (Sec 2.3)
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

Constants

- Small constants used frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them
  - create hard-wired registers (like $zero) for constants
- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- Design Principle: <u>Make the common case fast</u> Which format?

Immediate Operands

- Immediate: numerical *constants*
  - Often appear in code, so there are special instructions for them
  - Add Immediate:

```
        f = g + 10                (in C)
        addi $s0,$s1,10           (in MIPS)
```

    where `$s0,$s1` are associated with `f,g`
  - Syntax similar to `add` instruction, except that last argument is a number instead of a register
  - One particular immediate, the number zero (0), appears very often in code; so we define register zero (`$0` or `$zero`) to always 0
  - This is defined in hardware, so an instruction like `addi $0,$0,5` will not do anything

Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format (Sec. 2.4.~2.9)
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

Instructions as Numbers

- Currently we only work with words (32-bit blocks):
  - Each register is a word
  - `lw` and `sw` both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "`add $t0,$0,$0`" is meaningless to hardware
  - MIPS wants simplicity: since data is in words, make instructions be words…

MIPS Instruction Format

- One instruction is 32 bits
  => divide instruction word into “fields”
  - Each field tells computer something about instruction
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - *R-format*: for register
  - *I-format*: for immediate, and `lw` and `sw` (since the offset counts as an immediate)
  - *J-format*: for jump

R-Format Instructions (1/2)

- Define the following "fields":

6	5	5	5	5	6
`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`

  - `opcode`: partially specifies what instruction it is (Note: 0 for all R-Format instructions)
  - `funct`: combined with `opcode` to specify the instruction

    Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

R-Format Instructions (2/2)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
- Final field:
  - `shamt`: contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
  - This field is set to 0 in all but the shift instructions

R-Format Example

- MIPS Instruction:

```
add    $8,$9,$10
```

  - opcode = 0 (look up in table)
  - funct = 32 (look up in table)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - rd = 8 (destination)
  - shamt = 0 (not a shift)

binary representation:

000000	**01001**	**01010**	**01000**	**00000**	**100000**

called a Machine Language Instruction

I-Format Instructions

- Define the following "fields":

6	5	5	16
`opcode`	`rs`	`rt`	`immediate`

  - `opcode`: uniquely specifies an I-format instruction
  - `rs`: specifies the *only* register operand
  - `rt`: specifies register which will receive result of computation (*target register*)
  - `addi`, `slti`, immediate is sign-extended to 32 bits, and treated as a signed integer
  - 16 bits ➔ can be used to represent immediate up to 2^{16} different values
- Key concept: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location

I-Format Example 1

- MIPS Instruction:

```
addi    $21,$22,-50
```

  - opcode = 8 (look up in table)
  - rs = 22 (register containing operand)
  - rt = 21 (target register)
  - immediate = -50 (by default, this is decimal)

decimal representation:

8	22	21	-50

binary representation:

001000	10110	10101	1111111111001110

I-Format Example 2

- MIPS Instruction:

```
lw    $t0,1200($t1)
```

  - opcode = 35 (look up in table)
  - rs = 9 (base register)
  - rt = 8 (destination register)
  - immediate = 1200 (offset)

decimal representation:

35	9	8	1200

binary representation:

100011	01001	01000	0000010010110000

I-Format Problem

What if immediate is too big to fit in immediate field?

- Load Upper Immediate:

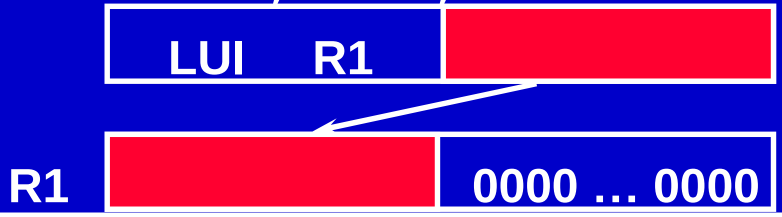
```
lui   register, immediate
```

  - puts 16-bit immediate in upper half (high order half) of the specified register, and sets lower half to 0s

```
addi   $t0,$t0, 0xABABCDCD
```

becomes:

```
lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```



Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Thus, entire programs can be stored in memory to be read or written just like numbers (data)
- One consequence: everything addressed
  - Everything has a memory address: instructions, data
    - both branches and jumps use these
  - One register keeps address of the instruction being executed: "Program Counter" (PC)
    - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better
  - A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), etc.

Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer, and addressing
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical (Sec 2.5)
  - Decision making and branches
  - Jumps for procedures

MIPS Arithmetic Instructions

Instruction	*Example*	*Meaning*	*Comments*
add	**add \$1,\$2,\$3**	**\$1 = \$2 + \$3**	**3 operands;**
subtract	**sub \$1,\$2,\$3**	**\$1 = \$2 - \$3**	**3 operands;**
add immediate	**addi \$1,\$2,100**	**\$1 = \$2 + 100**	**+ constant;**

Bitwise Operations

- Up until now, we've done arithmetic (`add`, `sub`, `addi`) and memory access (`lw` and `sw`)
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New perspective: View contents of register as 32 bits rather than as a single 32-bit number
- Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions

Logical Operators

- Logical instruction syntax:

```
1      2    3    4
or     $t0, $t1, $t2
```

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) second operand (register) or immediate (numerical constant)
- Instruction names:
  - `and`, `or`: expect the third argument to be a register
  - `andi`, `ori`: expect the third argument to be immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

Use for Logical Operator And

- and operator can be used to set certain portions of a bit-string to 0s, while leaving the rest alone => mask

- Example:

Mask: 1011 0110 1010 0100 0011 $\boxed{\text{1101 1001 1010}}$

0000 0000 0000 0000 0000 1111 1111 1111

- The result of anding these two is:

0000 0000 0000 0000 0000 $\boxed{\text{1101 1001 1010}}$

- In MIPS assembly: `andi $t0,$t0,0xFFF`

Use for Logical Operator Or

- or operator can be used to force certain bits of a string to 1s

- For example,

  `$t0` = `0x12345678`, then after

  ```
          ori $t0, $t0, 0xFFFF
  ```

  `$t0` = `0x1234FFFF`

  (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are set to 1s)

Shift Instructions (1/3)

- Shift Instruction Syntax:

```
 1      2   3   4
sll    $t2,$s0,4
```

  1) operation name

  2) register that will receive value

  3) first operand (register)

  4) shift amount (constant)

- MIPS has three shift instructions:
  - `sll` (shift left logical): shifts left, fills empties with 0s
  - `srl` (shift right logical): shifts right, fills empties with 0s
  - `sra` (shift right arithmetic): shifts right, fills empties by sign extending

Shift Instructions (2/3)

- Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Example: shift right by 8 bits

$\boxed{\text{0001 0010 0011 0100 0101 0110}}$ 0111 1000

0000 0000 $\boxed{\text{0001 0010 0011 0100 0101 0110}}$

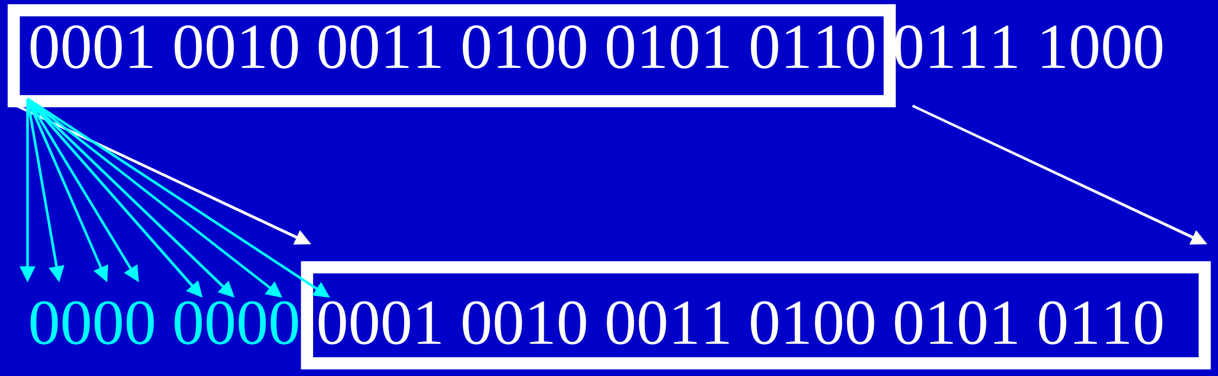
- Example: shift left by 8 bits

0001 0010 $\boxed{\text{0011 0100 0101 0110 0111 1000}}$

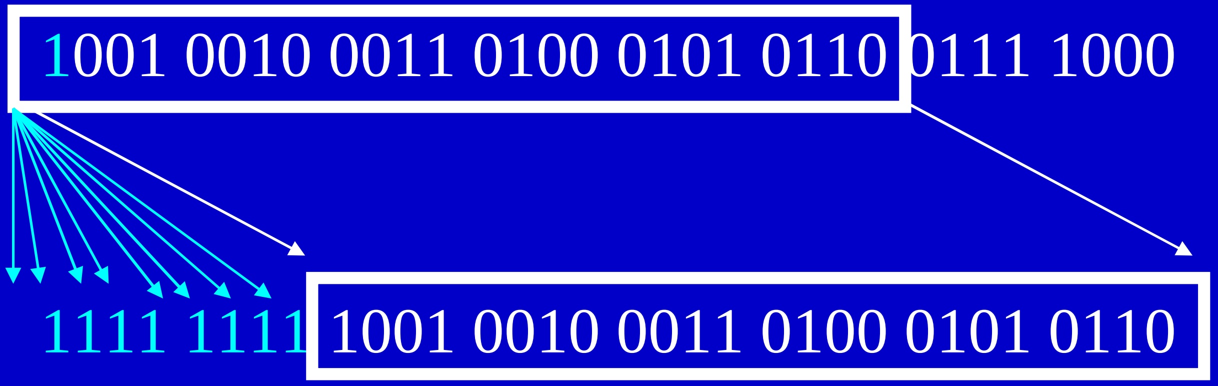
$\boxed{\text{0011 0100 0101 0110 0111 1000}}$ 0000 0000

Shift Instructions (3/3)

- Example: shift right arithmetic by 8 bits



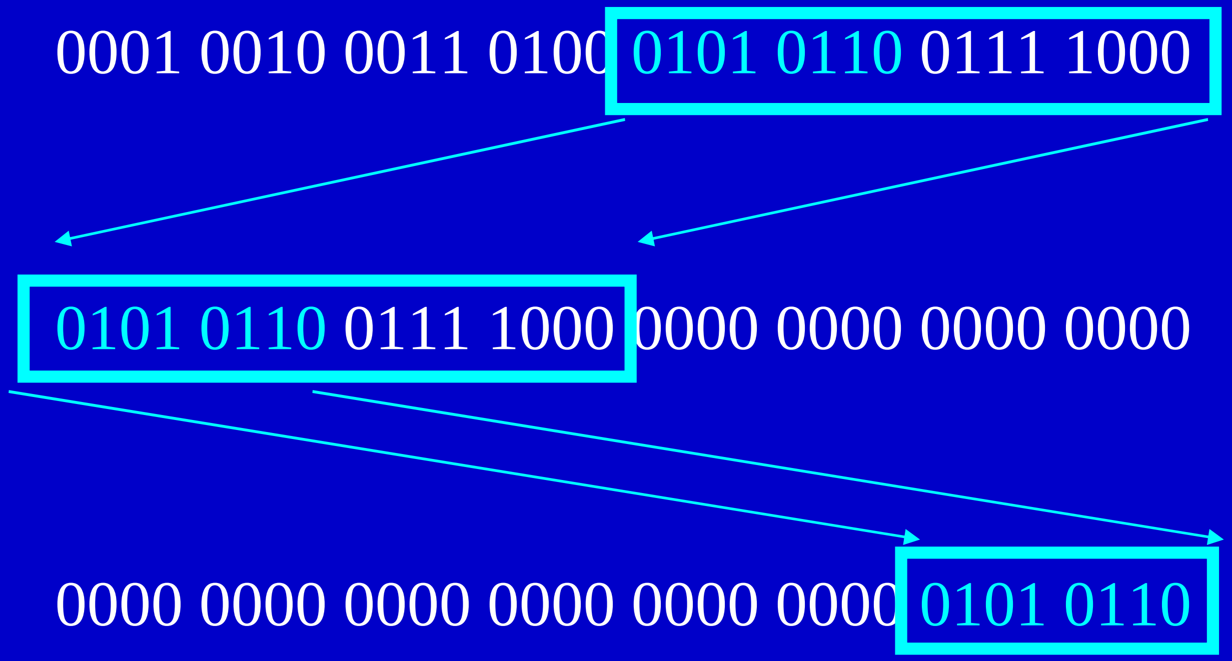
- Example: shift right arithmetic by 8 bits



Uses for Shift Instructions (1/2)

- Suppose we want to get byte 1 (bit 15 to bit 8) of a word in `$t0`. We can use:

```
    sll   $t0,$t0,16
    srl   $t0,$t0,24
```



0001 0010 0011 0100 0101 0110 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0101 0110

Uses for Shift Instructions (2/2)

- Shift for multiplication: in binary
  - Multiplying by 4 is same as shifting left by 2:
    - 11_2 x 100_2 = 1100_2
    - 1010_2 x 100_2 = 101000_2
  - Multiplying by 2^n is same as shifting left by n
- Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

```
a *= 8;
```
(in C)

would compile to:

```
sll   $s0,$s0,3
```
(in MIPS)

MIPS Logical Instructions

Instruction	*Example*	*Meaning*	*Comment*
and	**and $1,$2,$3**	**$1 = $2 & $3**	**3 reg. operands; Logical AND**
or	**or $1,$2,$3**	**$1 = $2 | $3**	**3 reg. operands; Logical OR**
nor	**nor $1,$2,$3**	**$1 = ~($2 |$3)**	**3 reg. operands; Logical NOR**
and immediate	**andi $1,$2,10**	**$1 = $2 & 10**	**Logical AND reg, zero exten.**
or immediate	**ori $1,$2,10**	**$1 = $2 | 10**	**Logical OR reg, zero exten.**
shift left logical	**sll $1,$2,10**	**$1 = $2 << 10**	**Shift left by constant**
shift right logical	**srl $1,$2,10**	**$1 = $2 >> 10**	**Shift right by constant**
shift right arithm.	**sra $1,$2,10**	**$1 = $2 >> 10**	**Shift right (sign extend)**

So Far...

- All instructions have allowed us to manipulate data.
- So we’ve built a calculator.
- In order to build a computer, we need ability to make decisions…

Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands
  – Register operands and their organization
  – Memory operands, data transfer, and addressing
  – Immediate operands
- Instruction format
- Operations
  – Arithmetic and logical
  – Decision making and branches (Sec. 2.6, 2.9)
  – Jumps for procedures

Decision Making: Branches

Decision making: *if* statement, sometimes combined with *goto* and *labels*

beq register1, register2, L1(beq: Branch if equal)

Go to the statement labeled L1 if the value in register1 equals the value in register2

bne register1, register2, L1(bne: Branch if not equal)

Go to the statement labeled L1 if the value in register1 does not equal the value in register2

beq and bne are termed Conditional branches

What instruction format is beq and bne?

MIPS Decision Instructions

```
beq   register1, register2, L1
```

- Decision instruction in MIPS:

```
beq   register1, register2, L1
```

  "Branch if (registers are) equal"

  meaning :

```
 if  (register1==register2) goto L1
```

- Complementary MIPS decision instruction

```
bne   register1, register2, L1
```

  "Branch if (registers are) not equal"

  meaning :

```
 if  (register1!=register2) goto L1
```

- These are called conditional branches

MIPS Goto Instruction

```
j    label
```

- MIPS has an unconditional branch:

```
j     label
```

  - Called a Jump Instruction: jump directly to the given label without testing any condition
  - meaning :

```
goto label
```

- Technically, it’s the same as:

```
beq      $0,$0,label
```

  since it always satisfies the condition

- It has the j-type instruction format

Compiling an If statement

```
        If (i == j) go to L1;
        f = g + h;
L1:     f = f-i;
```

f, g, h, i, and j correspond to five registers $s0 through $s4.

```
        beq $s3, $s4, L1        #go to L1 if i equals j
        add $s0, $s1, $s2       # f = g+h (skipped if i equals j)
L1:     sub $s0, $s0, $s3       # f = f –i (always executed)
```

Instructions must have memory addresses

Label L1 corresponds to address of sub instruction

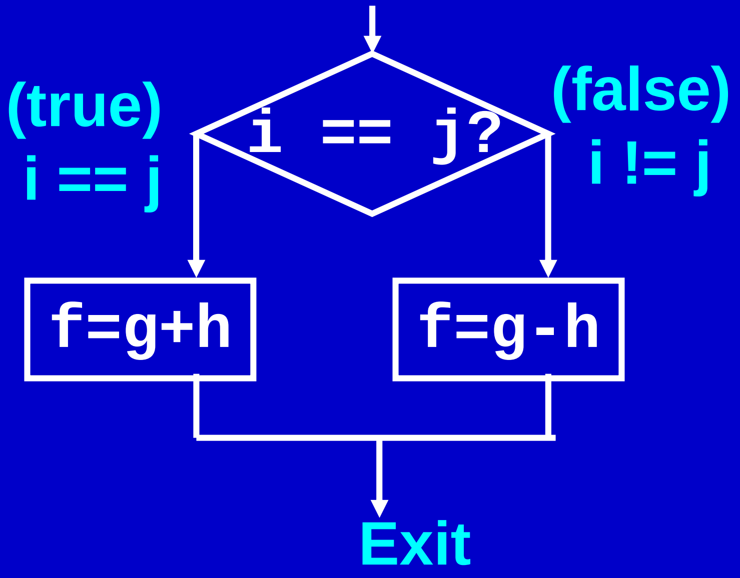
Compiling an if-then-else

- Compile by hand

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

- Use this mapping:

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



- Final compiled MIPS code:

```
          beq   $s3,$s4,True      # branch i==j
          sub   $s0,$s1,$s2       # f=g-h(false)
          j     Fin               # go to Fin
True:     add   $s0,$s1,$s2       # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches) appropriately

Inequalities in MIPS

- Until now, we've only tested equalities (== and != in C), but general programs need to test < and >
- Set on Less Than:

```
        slt  reg1,reg2,reg3
```

meaning :

```
         if (reg2 < reg3)
           reg1 = 1;                # set
         else reg1 = 0;             # reset
```

- Compile by hand: `if (g < h) goto Less;`
  Let `g: $s0, h: $s1`

```
slt $t0,$s0,$s1    # $t0 = 1 if g<h
bne $t0,$0,Less    # goto Less if $t0!=0
```

MIPS has no "branch on less than" => too complex

Immediate in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`

```
     if (g >= 1) goto Loop
C Loop:     . . .
```

M I P S

```
slti $t0,$s0,1     # $t0 = 1 if $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop if $t0==0
```

- Unsigned inequality: `sltu`, `sltiu`

  $s0 = FFFF FFFA$_{hex}$, $s1 = 0000 FFFA$_{hex}$

```
slt  $t0, $s0, $s1     => $t0 = ?
sltu $t1, $s0, $s1     => $t1 = ?
```

Branches: Instruction Format

- Use I-format:

`opcode`	`rs`	`rt`	`immediate`

  - `opcode` specifies `beq` or `bne`
  - `rs` and `rt` specify registers to compare
- What can *`immediate`* specify? PC-relative addressing
  - *`Immediate`* is only 16 bits, but PC is 32-bit
    => *`immediate`* cannot specify entire address
  - Loops are generally small: < 50 instructions
    - Though we want to branch to anywhere in memory, a single branch only need to change PC by a small amount
  - How to use PC-relative addressing
    - 16-bit *`immediate`* as a signed two's complement integer to be *added* to the PC if branch taken
    - Now we can branch +/- 2^{15} bytes from the PC ?

Branches: Instruction Format

- *`Immediate`* specifies word address
  - Instructions are word aligned (byte address is always a multiple of 4, i.e., it ends with 00 in binary)
    - The number of bytes to add to the PC will always be a multiple of 4
  - Specify the *`immediate`* in words (confusing?)
  - Now, we can branch +/- 2^{15} words from the PC (or +/- 2^{17} bytes), handle loops 4 times as large

- *`Immediate`* specifies PC + 4
  - Due to hardware, add `immediate` to (PC+4), not to PC
  - If branch not taken: PC = PC + 4
  - If branch taken: PC = (PC+4) + (`immediate*4`)

Branch Example

- MIPS Code:

```
Loop:  beq   $9,$0,End
       add   $8,$8,$10
       addi  $9,$9,-1
       j     Loop
End:
```

- Branch is I-Format:

opcode	rs	rt	immediate

`opcode` = 4 (look up in table)

`rs` = 9 (first operand)

`rt` = 0 (second operand)

`immediate` = ???

– Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch
=> `immediate` = 3

Branch Example

- MIPS Code:

```
Loop: beq   $9,$0,End
      add   $8,$8,$10
      addi  $9,$9,-1
      j     Loop
 End:
```

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

J-Format Instructions (1/3)

- For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.
- For general jumps (`j` and `jal`), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

J-Format Instructions (2/3)

- Define "fields" of the following number of bits each:

6 bits	26 bits

- As usual, each field has a name:

`opcode`	`target address`

- Key concepts:
  - Keep `opcode` field identical to R-format and I-format for consistency
  - Combine other fields to make room for target address
- Optimization:
  - Jumps only jump to word aligned addresses
    - last two bits are always 00 (in binary)
    - specify 28 bits of the 32-bit bit address

J-Format Instructions (3/3)

- Where do we get the other 4 bits?
  - Take the 4 highest order bits from the PC
  - Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999…% of the time, since programs aren’t that long
  - Linker and loader avoid placing a program across an address boundary of 256 MB
- Summary:
  - New PC = PC[31..28] || target address (26 bits) || 00
  - Note: II means concatenation
    4 bits || 26 bits || 2 bits = 32-bit address
- If we absolutely need to specify a 32-bit address:
  - Use *jr $ra* # jump to the address specified by $ra

MIPS Jump, Branch, Compare

Instruction	*Example*	*Meaning*
branch on equal	**beq \$1,\$2,25**	**if (\$1 == \$2) go to PC+4+100** ***Equal test; PC relative branch***
branch on not eq.	**bne \$1,\$2,25**	**if (\$1!= \$2) go to PC+4+100** ***Not equal test; PC relative***
set on less than	**slt \$1,\$2,\$3**	**if (\$2 < \$3) \$1=1; else \$1=0** ***Compare less than; 2's comp.***
set less than imm.	**slti \$1,\$2,100**	**if (\$2 < 100) \$1=1; else \$1=0** ***Compare < constant; 2's comp..***
jump	**j 10000**	**go to 10000** ***26-bit+4-bit of PC***

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Immediate operands
  - Memory operands, data transfer, and addressing
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures (Sec. 2.7)

Procedures

- Procedure/Subroutine

  A set of instructions stored in memory which perform a set of operations based on the values of parameters passed to it and returns one or more values

- Steps for execution of a procedure or subroutine

  ➢The program (caller) places parameters in places where the procedure (callee) can access them

  ➢The program transfers control to the procedure

  ➢The procedure gets storage needed to carry out the task

  ➢The procedure carries out the task, generating values

  ➢The procedure (callee) places values in places where the program (caller) can access them

  ➢The procedure transfers control to the program (caller)

Procedures

- int f1 (inti, intj, intk, intg)
```
{ ::::
        return 1;                callee
}
```
- int f2 (ints1, ints2)
```
{
::::::
add $3,$4, $3
i = f1 (3,4,5, 6);               caller
add $2, $3, $3
::::
}
```
- How to pass parameters & results?
- How to preserve caller register values?
- How to alter control? (i.e., go to callee, return from callee)

Procedures

- How to pass parameters & results
  - $a0-$a3: four argument registers. What if # of parameters is larger than 4? – push to the stack
  - $v0-$v1: two value registers in which to return values
- How to preserve caller register values?
  - Caller saved register
  - Callee saved register
  - Use stack
- How to switch control?
  - How to go to the callee
    - jal procedure_address(jump and link)
      - Store the the return address (PC +4) at $ra
      - set PC = procedure_addres
- How to return from the callee
  - Callee exectues **jr $ra**

Procedure calling/return

•Studies of programs show that a large portions of procedures have a few parameters passed to them and return a very few, often one value to the caller

•Parameter values can be passed in registers

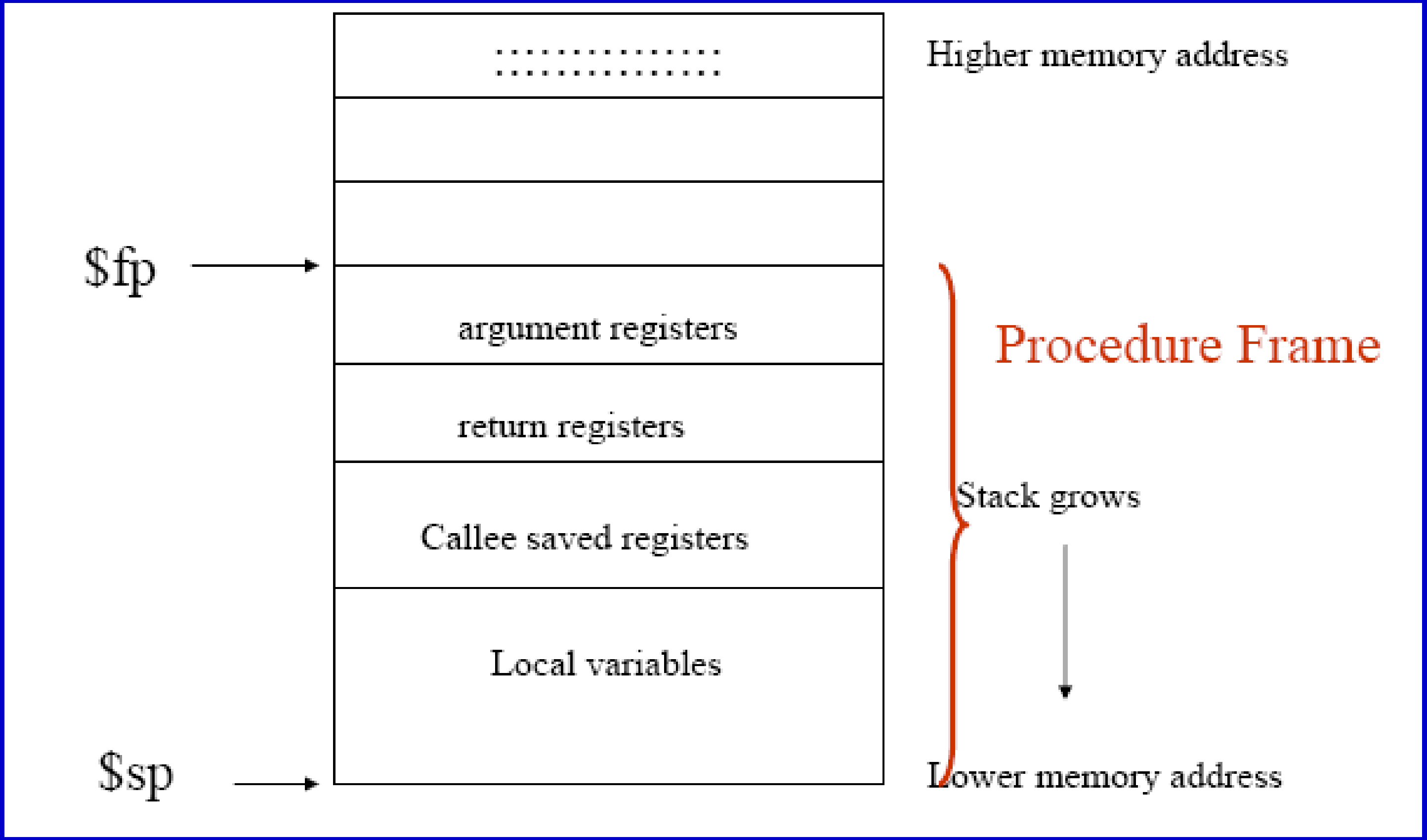
•MIPS allocates various registers to facilitate use of procedures

•$a0-$a3	four argument registers in which to pass parameters
•$v0-$v1	two value registers in which to return values
•$ra	one return address register to return to point of origin

•jump-and-link instruction   jal ProcedureAddress

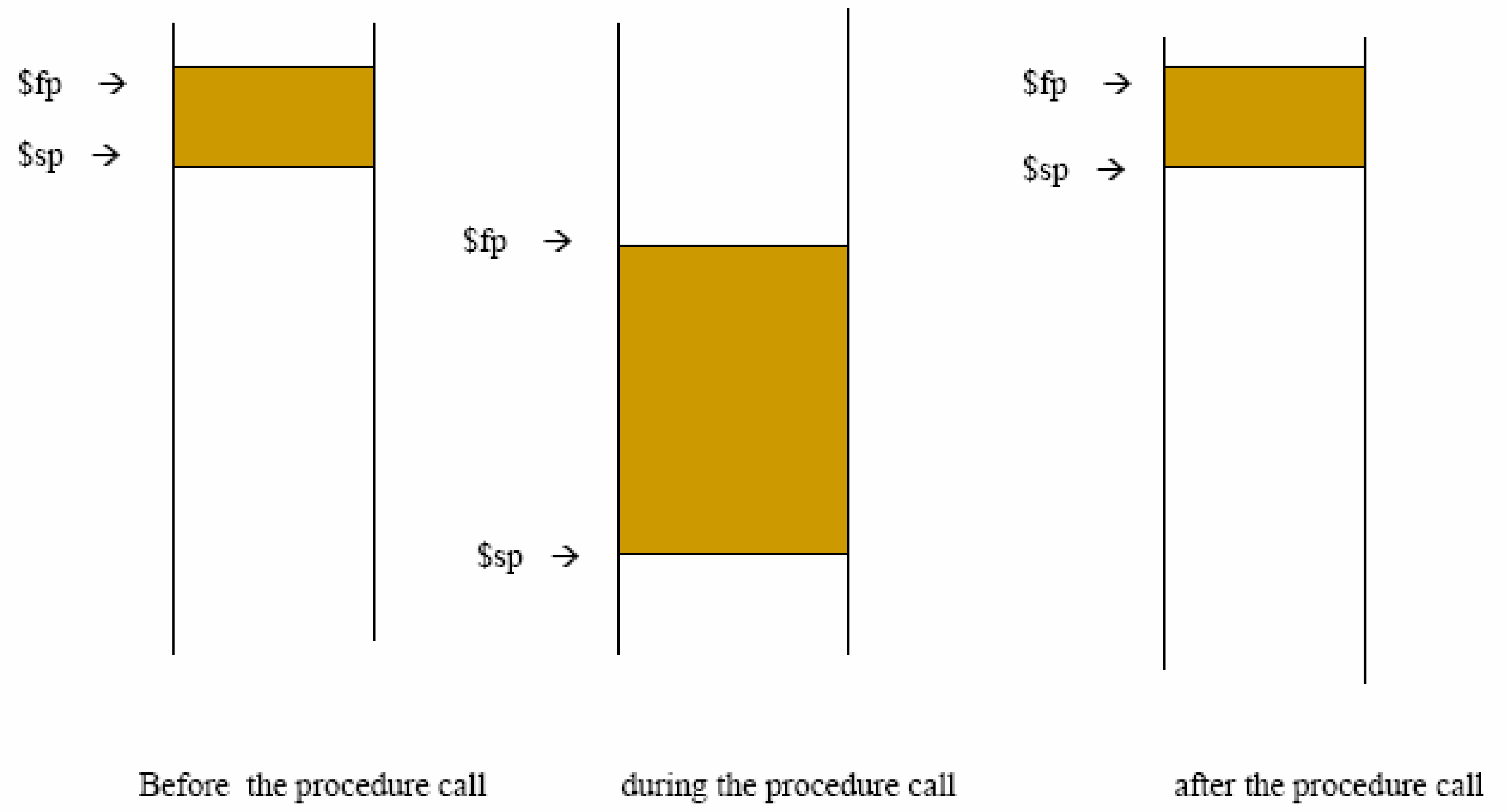
- ➢Jump to an address and simultaneously save the address of the following instruction in register $ra (What is the address of the following instruction?)
- ➢jal is a J-format instruction, with 26 bits relative word address. Pseudodirect addressing applies in this case.

Procedure Call Stack (Frame)



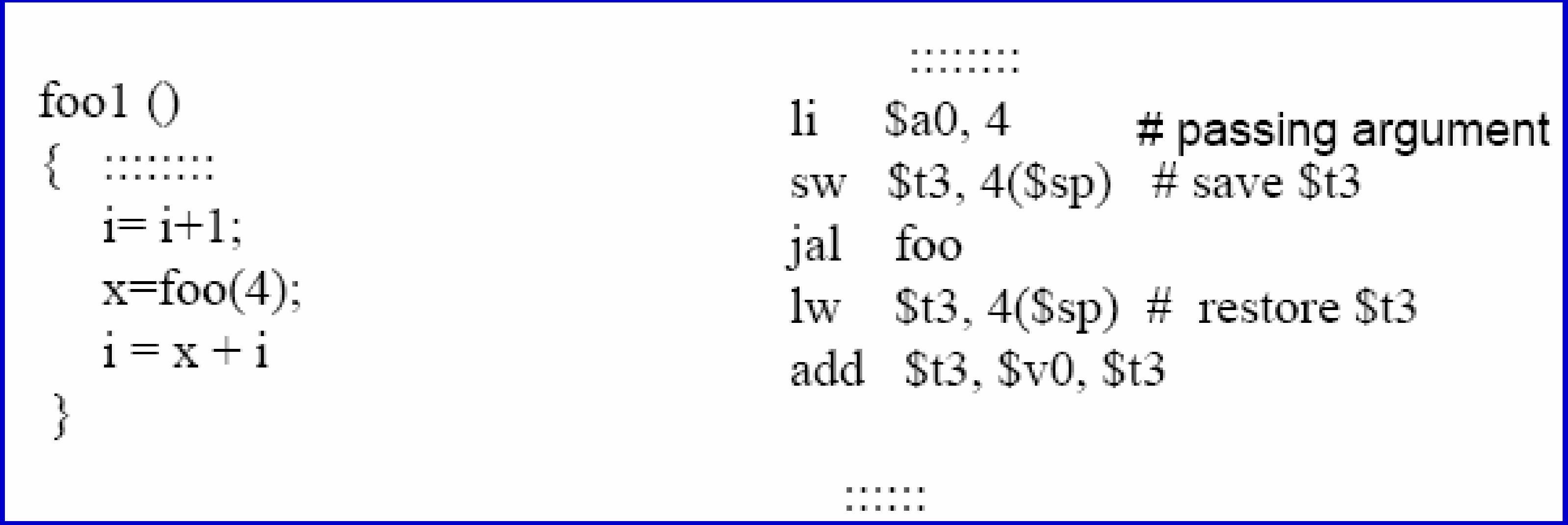
Frame pointer points to the first word of the procedure frame

Procedure Call Stack (Frame)



Procedure Calling Convention

- Calling Procedure
  - Step-1: pass the argument
  - Step-2: save caller-saved registers
  - Step-3: Execute a jal instruction

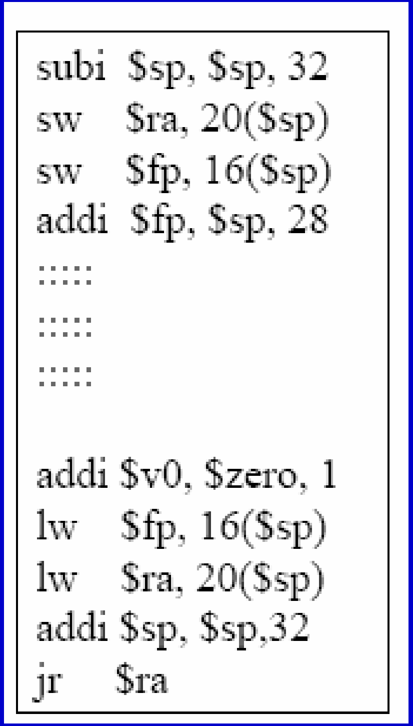


```
foo1 ()
{   ::::::::
    i= i+1;
    x=foo(4);
    i = x + i
}
```

```
          ::::::::
li     $a0, 4           # passing argument
sw     $t3, 4($sp)   # save $t3
jal    foo
lw     $t3, 4($sp)  #  restore $t3
add    $t3, $v0, $t3

       ::::::
```

Procedure Calling Convention

- Called Procedure
  - Step-1: establish stack frame
    - subi $sp, $sp <frame-size>
  - Step-2: saved callee saved registers
    - $ra, $fp,$s0-$s7
  - Step-3: establish frame pointer
    - add $fp, $sp, <frame-size>-4
- On return from a call
  - Step-1: put returned values in
    - register $v0, [$v1].
  - Step-2: restore callee-saved registers
  - Step-3: pop the stack
  - Step-4: return: jr $ra

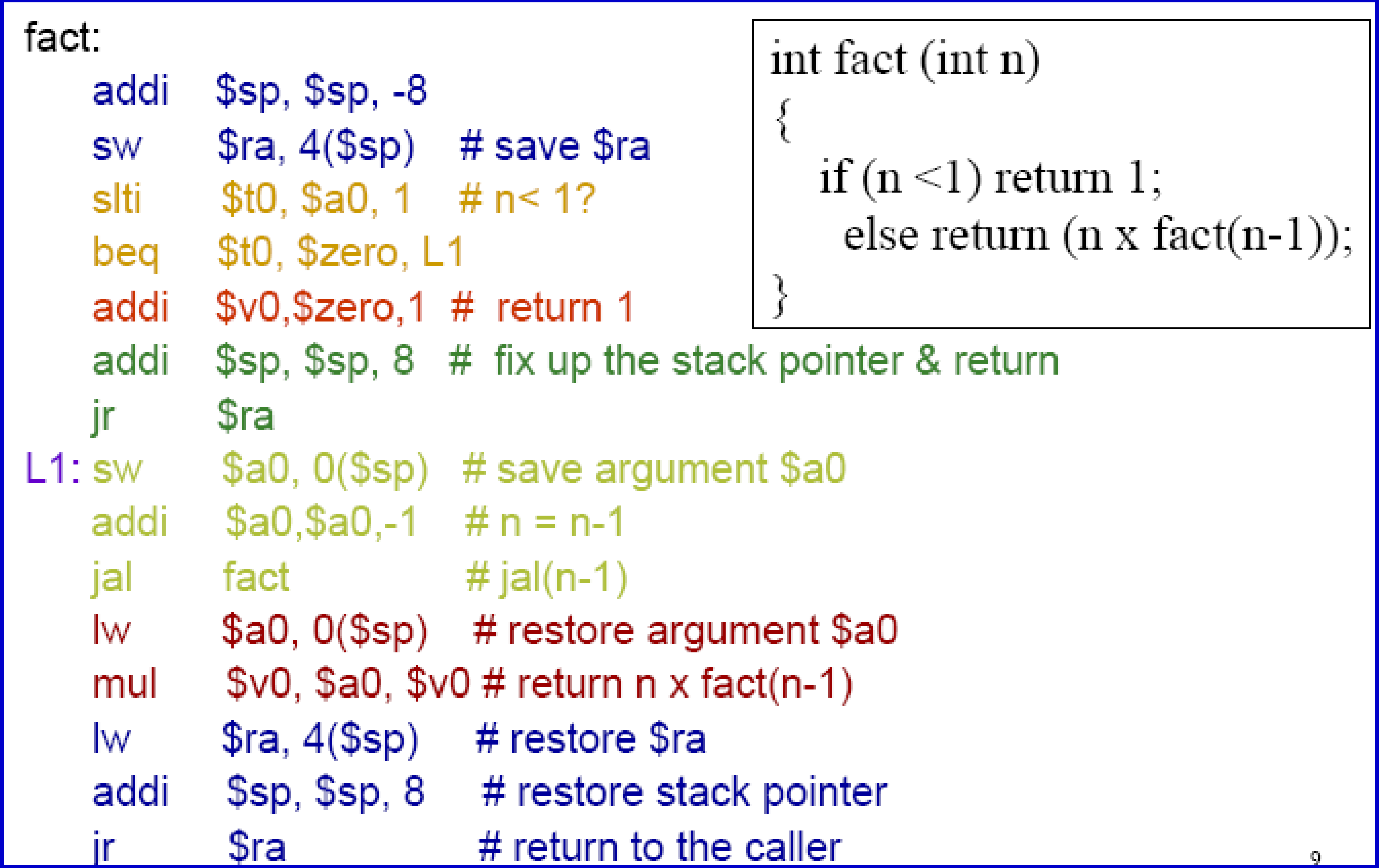


```
subi  $sp, $sp, 32
sw    $ra, 20($sp)
sw    $fp, 16($sp)
addi  $fp, $sp, 28
:::::
:::::
:::::

addi $v0, $zero, 1
lw    $fp, 16($sp)
lw    $ra, 20($sp)
addi $sp, $sp,32
jr    $ra
```

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Nested Procedures



```
fact:
     addi    $sp, $sp, -8
     sw      $ra, 4($sp)    # save $ra
     slti    $t0, $a0, 1    # n< 1?
     beq     $t0, $zero, L1
     addi    $v0,$zero,1  #  return 1
     addi    $sp, $sp, 8   #  fix up the stack pointer & return
     jr      $ra
L1:  sw      $a0, 0($sp)   # save argument $a0
     addi    $a0,$a0,-1    # n = n-1
     jal     fact          # jal(n-1)
     lw      $a0, 0($sp)   # restore argument $a0
     mul     $v0, $a0, $v0 # return n x fact(n-1)
     lw      $ra, 4($sp)    # restore $ra
     addi    $sp, $sp, 8    # restore stack pointer
     jr      $ra            # return to the caller
```

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

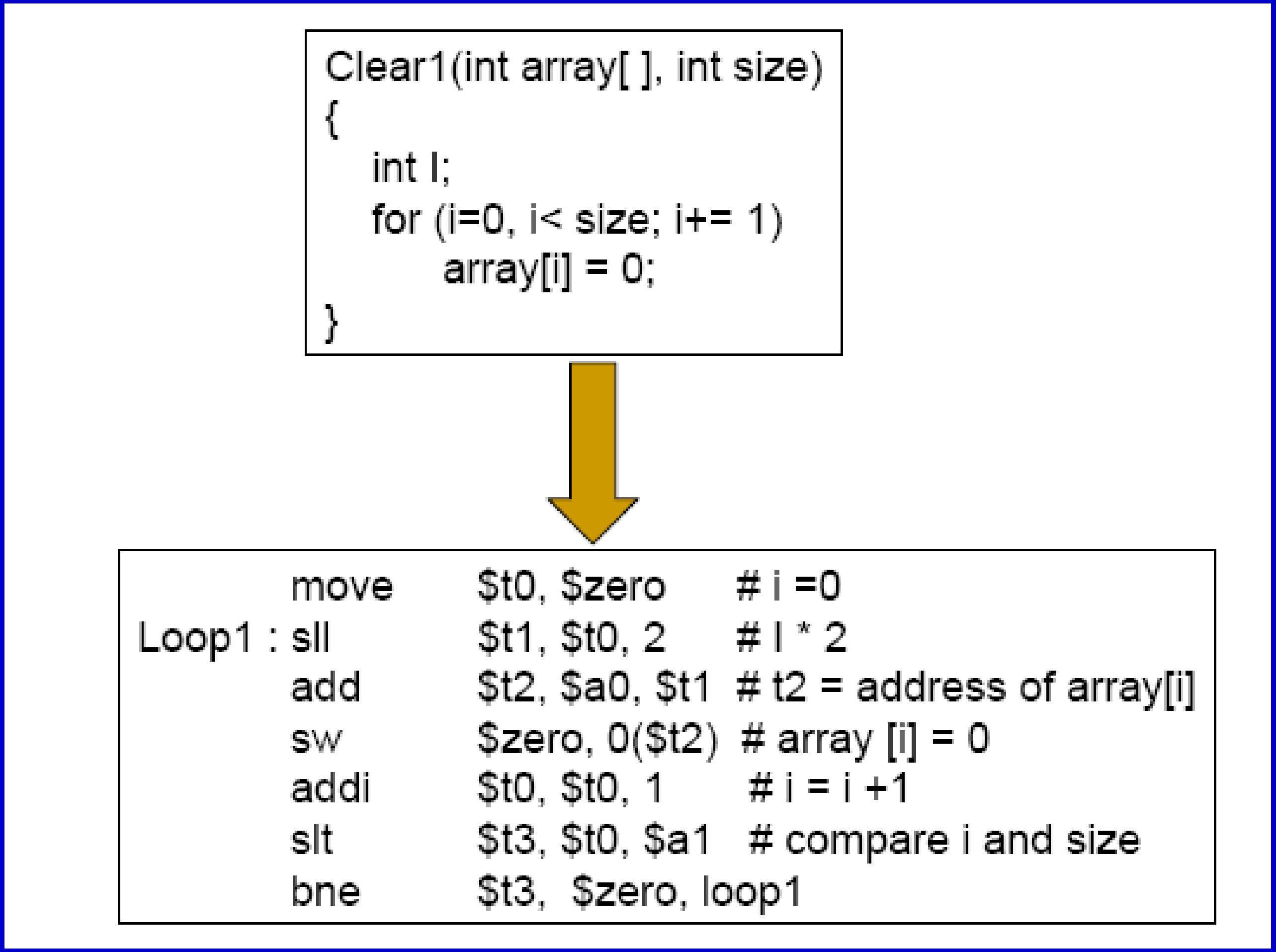
9

String Copy Procedure in C

```
void strcpy (char x[ ], char y[ ]) {
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
     i+=1;
}
```

```
strcpy:
     addi   $sp, $sp, -4              # adjust stack for 1 more item
     sw     $s0, 0($sp)               # save $s0
     add    $s0, $zero, $zero         # i = 0
L1:  add    $t1, $s0,  $a1            # address of y[i] in $t1
     lb     $t2, 0($t1)               # t2 = y[i]
     add    $t3, $s0, $a0             # address of x[i] in $t3
     sb     $t2, 0($t3)               # x[i] = y[i]
     beq    $t2, $zero, L2            #  if y[i]==0, go to L2
     addi   $s0, $s0, 1               #  i= i+1
     j      L1                        #  go to L1
L2:  lw     $s0, 0($sp)               #  y[i] ==0; end of string, restore old
                                      #  $s0
     addi   $sp, $sp, 4               #pop 1 word off stack
     jr     $ra                       #return
```

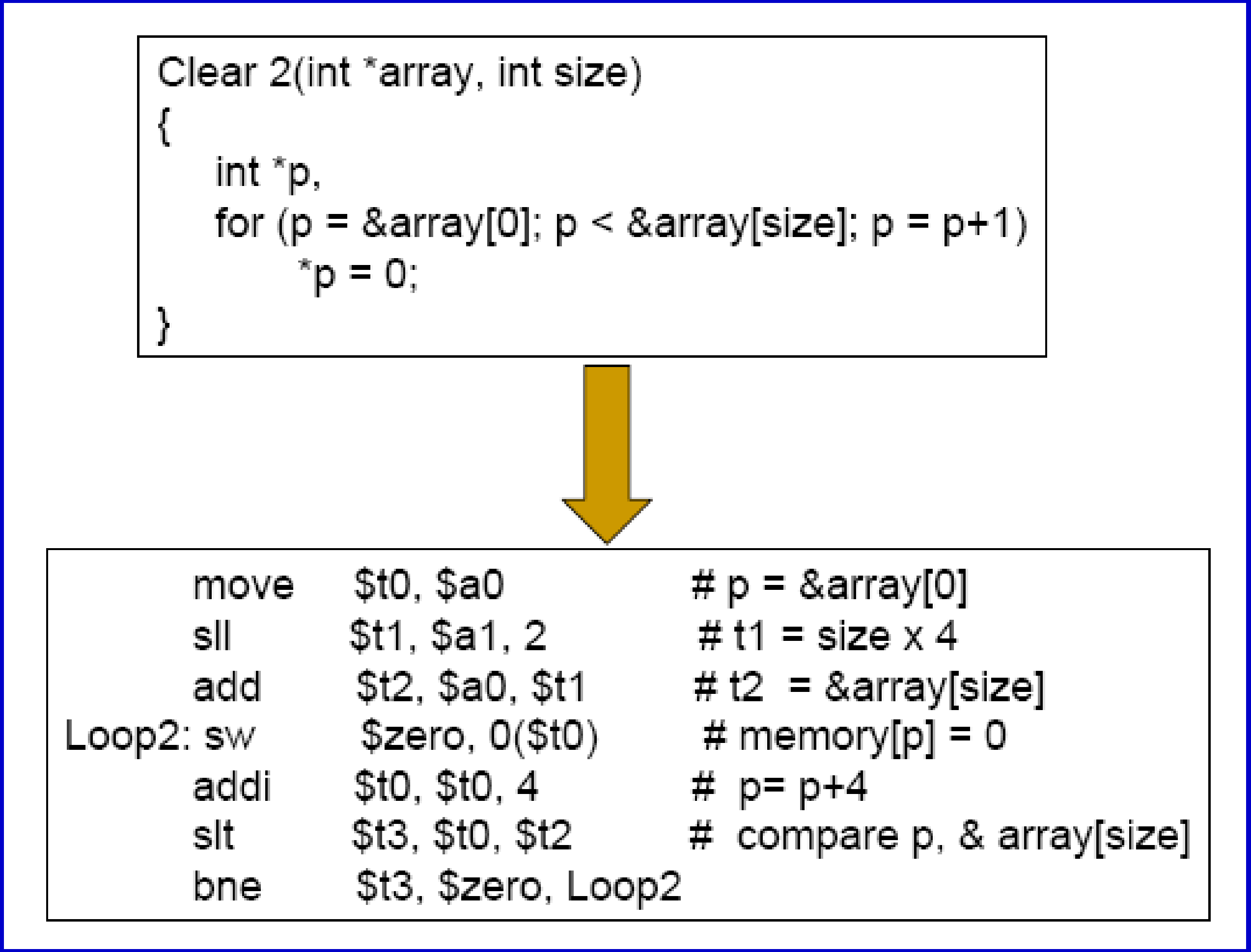
Array vs. Pointer



```
Clear1(int array[ ], int size)
{
    int I;
    for (i=0, i< size; i+= 1)
        array[i] = 0;
}
```

```
            move     $t0, $zero       # i =0
Loop1 : sll          $t1, $t0, 2      # I * 2
            add      $t2, $a0, $t1   # t2 = address of array[i]
            sw       $zero, 0($t2)  # array [i] = 0
            addi     $t0, $t0, 1       # i = i +1
            slt      $t3, $t0, $a1    # compare i and size
            bne      $t3,  $zero, loop1
```

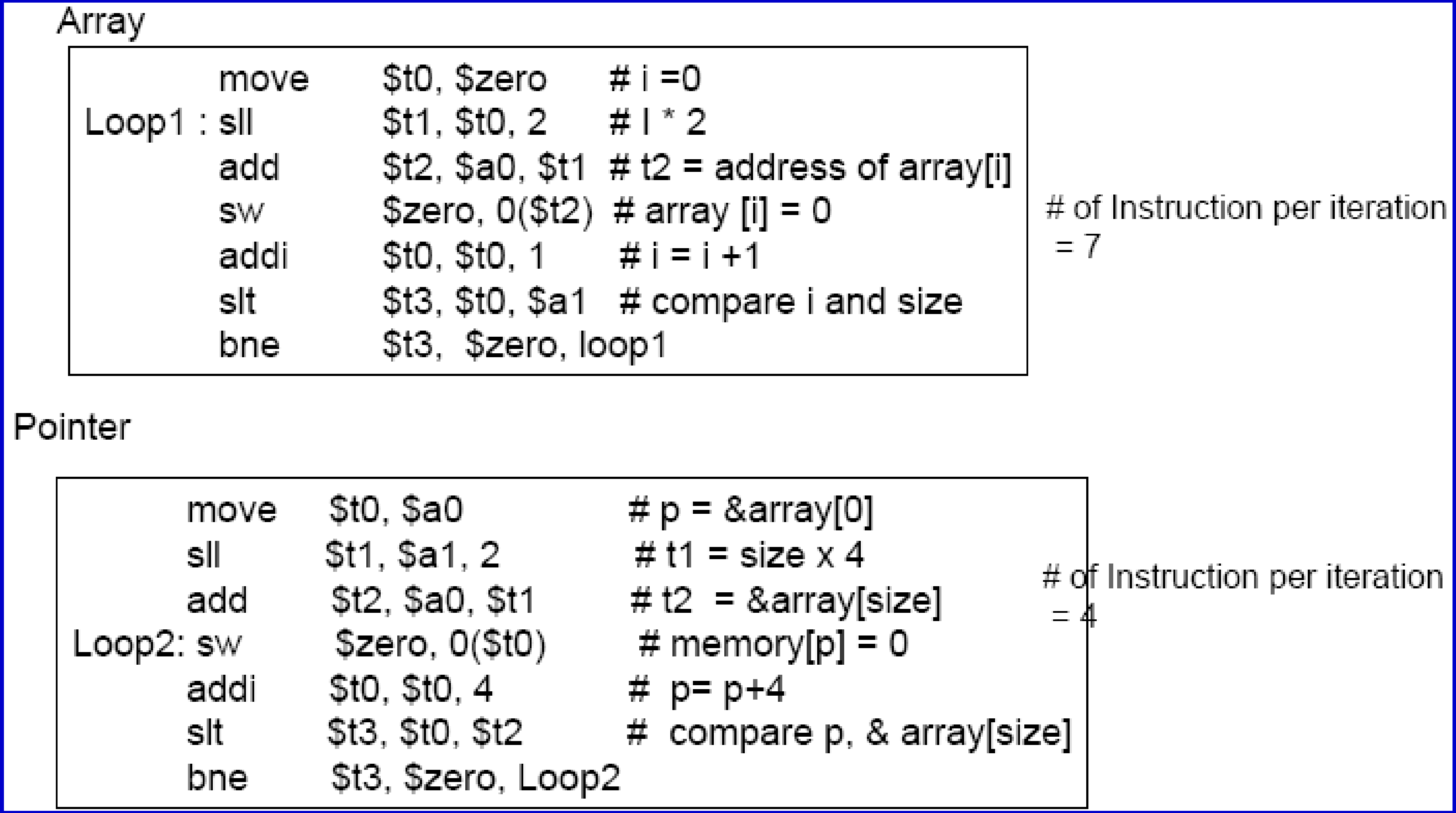
Array vs. Pointer



```
Clear 2(int *array, int size)
{
    int *p,
    for (p = &array[0]; p < &array[size]; p = p+1)
        *p = 0;
}
```

```
        move    $t0, $a0            # p = &array[0]
        sll     $t1, $a1, 2         # t1 = size x 4
        add     $t2, $a0, $t1       # t2  = &array[size]
Loop2:  sw      $zero, 0($t0)       # memory[p] = 0
        addi    $t0, $t0, 4         #  p= p+4
        slt     $t3, $t0, $t2       #  compare p, & array[size]
        bne     $t3, $zero, Loop2
```

Array vs. Pointer



Array

```
         move     $t0, $zero     # i =0
Loop1 : sll       $t1, $t0, 2    # I * 2
         add      $t2, $a0, $t1  # t2 = address of array[i]
         sw       $zero, 0($t2)  # array [i] = 0
         addi     $t0, $t0, 1     # i = i +1
         slt      $t3, $t0, $a1   # compare i and size
         bne      $t3,  $zero, loop1
```

# of Instruction per iteration
= 7

Pointer

```
        move    $t0, $a0              # p = &array[0]
        sll     $t1, $a1, 2            # t1 = size x 4
        add      $t2, $a0, $t1        # t2  = &array[size]
Loop2: sw        $zero, 0($t0)         # memory[p] = 0
        addi     $t0, $t0, 4          #  p= p+4
        slt     $t3, $t0, $t2         #  compare p, & array[size]
        bne      $t3, $zero, Loop2
```

# of Instruction per iteration
= 4

Procedure calling/return

- How to do the return jump?
  - Use a jr instruction   jr $ra

- Refined MIPS steps for execution of a procedure
  - ➢Caller puts parameter values in $a0-$a3
  - ➢Caller uses a jal X to jump to procedure X (callee)
  - ➢Callee performs calculations
  - ➢Callee place results in $v0-$v1
  - ➢Callee returns control to the caller using jr $ra

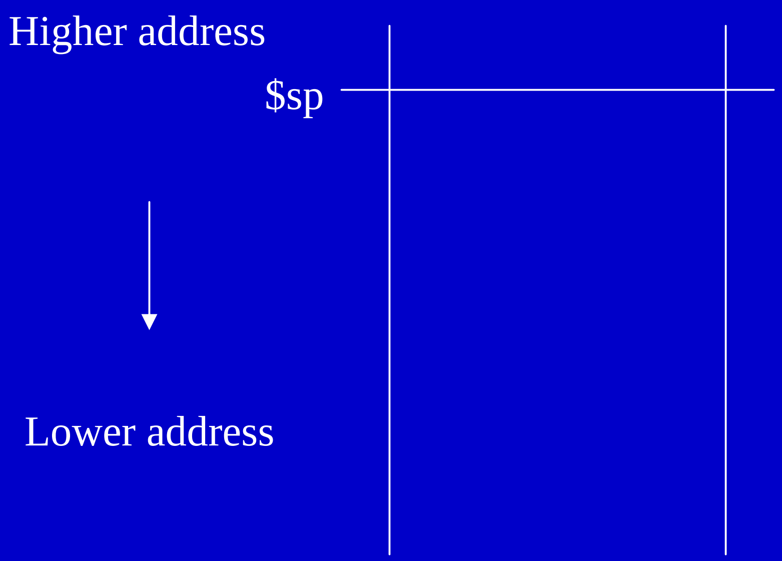
More Registers??

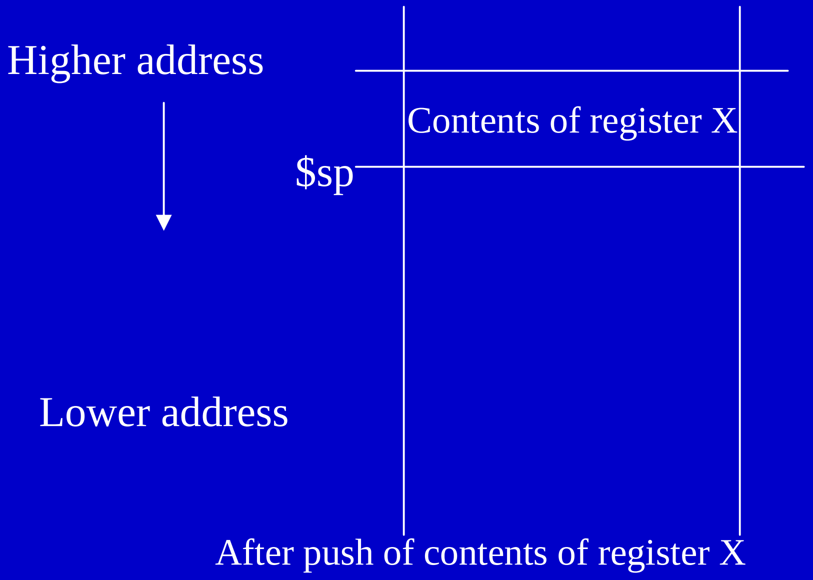
- What happens when the compiler needs more registers than 4 argument and 2 return value registers?
    - ➢Can we use \$t0-\$t7, \$s0-\$s7 in callee or does caller need values in these registers??
    - ➢\$t0-\$t9: 10 temporary registers that are not preserved by the callee on a procedure call
    - ➢\$s0-\$s7: 8 saved registers that must be preserved on a procedure call if used
- Any registers needed by the caller must be restored to the values they contained before the procedure was invoked
- How?
    - ➢Spill registers to memory
    - ➢use the registers in callee
    - ➢restore contents from memory
- We need a stack (LIFO data structure) (Why?)

➢Placing data onto stack	push
➢Removing data from stack	pop

Stack and Stack Pointer

- A pointer is needed to the stack top , to know where the next procedure should place the registers to be spilled or where old register values can be found (stack pointer)
- $sp is the stack pointer
- Stacks grow from higher addresses to lower addresses
    - What does a push/pop means in terms of operations on the stack pointer (+/-)?





Simple Example 1/2

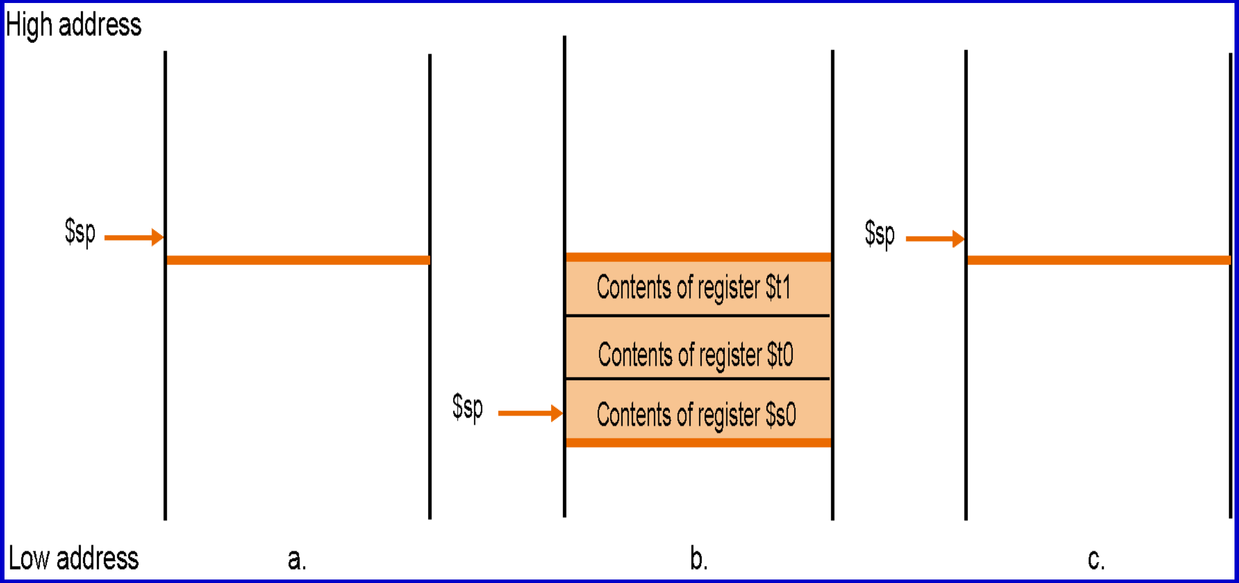
```
int leaf_example (int g, int h, int i, int j)
{
        int f;
        f = (g+h) – (i+j);
        return f;
}
```

What is the generated MIPS assembly code?

- g,h, i, and j correspond to $a0 through $a3
- Local variable f corresponds to $s0. Hence, we need to save $s0 before actually using it for local variable f (maybe caller needs it)
- Return value will be in $v0
- Textbook assumes that $t0, $t1 need to be saved for caller (page 135)

```
Leaf_example:          #procedure label
subi $sp,$sp,4         #make room for 1 item
sw $s0, 0 ($sp)        #store register $s0 for use later

add $t0, $a2, $a1      # $t0 ← g+h
add $t1,$a2,$a3        # $t1 ← i+j
sub $s0,$t0,$t1        #f ← $t0-$t1

add $v0,$s0,$zero      # set up return value in $v0

lw $s0, 0($sp)         # restore register $s0 for caller
addi $sp,$sp,4         #adjust stack to delete 1 item

jr $ra                 #jump back to caller
```

Simple Example 2/2



```
subi $sp,$sp,12      # adjust stack to make room for 3 items
sw $t1, 8($sp)       # save register $t1 for later use
sw $t0 ,4($sp)       # save register $t0 for later use
sw $s0,0($sp)        # save register $s0 for later use
```

Real Picture: It is not that Simple 1/2

How about if a procedure invokes another procedure?

- main calls procedure A with one argument
- A calls procedure B with one argument
- If precautions not taken
  - $a0 would be overwritten when B is called and value of parameter passed to A would be lost
  - When B is called using a jal instruction, $ra is overwritten

- How about if caller needs the values in temporary registers $t0-$t9?
- More than 4 arguments?
- Local variables that do not fit in registers defined in procedures? (such as?)
- We need to store the register contents and allocate the local variables somewhere?
- We already saw a solution when we saved $s0 before using it in the previous example

Real Picture: It is not that Simple 2/2

Solution

- Use segment of stack to save register contents and hold local variables (procedure frame or activation record)
- If $sp changes during procedure execution, that means that accessing a local variable in memory might use different offsets depending on their position in the procedure
- Some MIPS software uses a frame pointer $fp to point to first word procedure frame
- $fp provides a stable base register within a procedure for local memory references
- $sp points to the top of the stack, or the last word in the current procedure frame
- An activation record appears on the stack even if $fp is not used.

Procedure Call details 1/3

Caller

- Passes arguments
  - ➢The first 4 in registers $a0-$a3
  - ➢The remainder of arguments in the stack (push onto stack)
    - ✓Load other arguments into memory in the frame
    - ✓$sp points to last argument
- Save the caller-saved registers ($a0-$a3 and $t0-$t9) if and only if the caller needs the contents intact after call return
- Execute a jal instruction which saves the return address in $ra and jumps to the procedure

Procedure Call details 2/3

Callee

•Allocates memory on the stack for its frame by subtracting the frame's size from the stack pointer ($sp ← $sp – frame size)

•Save callee-saved registers in the frame ($s0-$s7, $fp, and $ra) before altering them since the caller expects to find these registers unchanged after the call

- ➢$fp is saved by every procedure that allocates a new stack frame (we will not worry about this issue in our examples)
- ➢$ra only needs to be saved if the callee itself makes a call

•Establish its frame pointer (we will not worry about this issue in our examples)

•The callee ends by

- •Return the value if a function in $v0
- •Restore all callee-saved registers that were saved upon procedure entry
- •Pop the stack frame by adding the frame size to $sp
- •Return by jumping to the address in register $ra   (jr $ra)

Procedure Call details 3/3

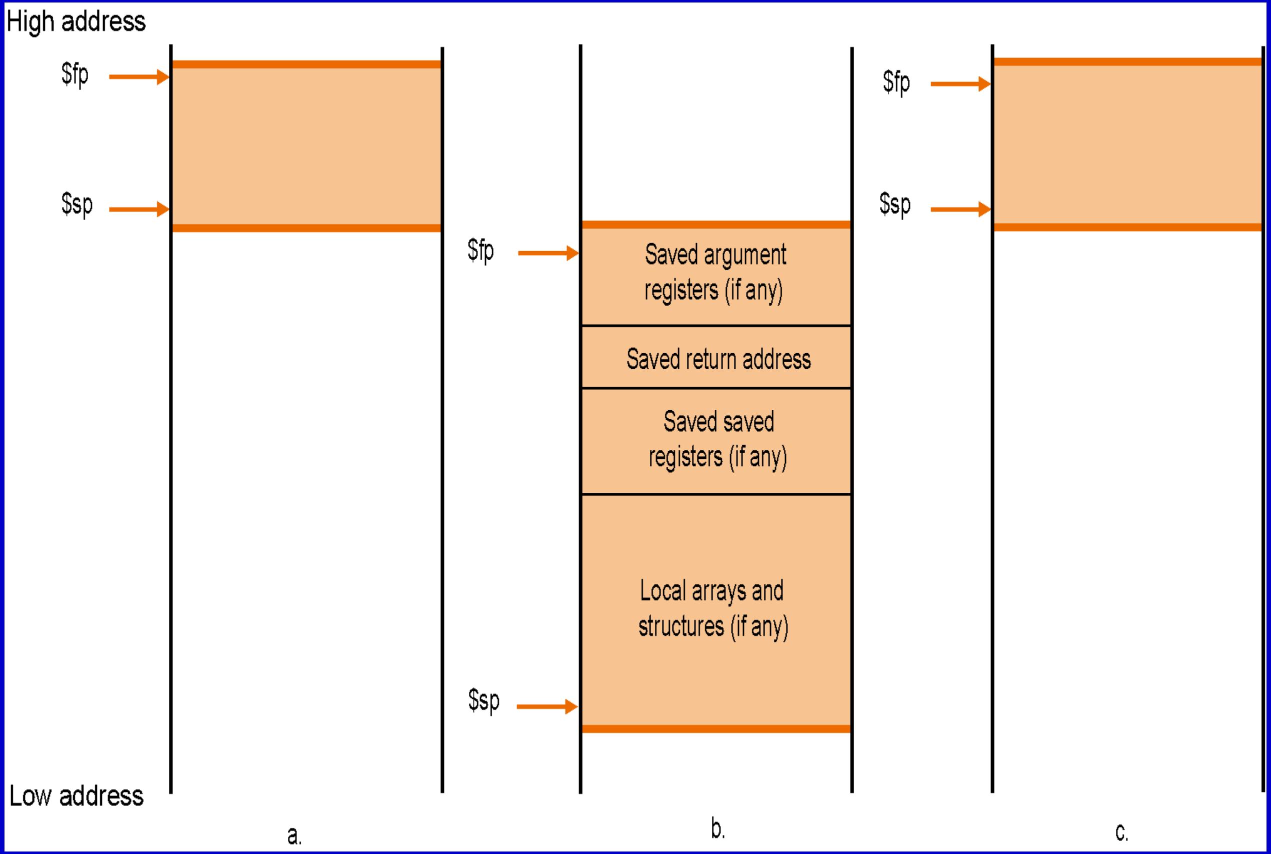

High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

Figure 3.12 page 139

Example: Swap array Elements

```
void swap (int v[], int k)
{
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

What is the generated MIPS assembly code?

- v and k correspond to $a0 and $a1
- What is actually passed as v?
  - The base address of the array
- Local variable temp corresponds to $t0. (Why we can use $t0 and not use $s0 as explained before?)
  - ➢This is a leaf procedure
  - ➢$t0 does not have to be saved by callee
- No registers need to be saved
- No return value

```
swap:                 #procedure label
add $t1, $a1, $a1     # $t1 ← k *2
add $t1,$t1,$t1       # $t1 ← k *4
add $t1,$a0,$t1       #$t1 ← base + (k*4)

lw $t0, 0($t1)        # temp ← v[k]
lw $t2, 4($t1)        # $t2 ← v[k+1]

sw $t2,0($t1)         #v[k] ← $t2 (which is v[k+1])
sw $t0,4($t1)         # v[k+1] ← v[k] (temp)

jr $ra                #jump back to caller
```

Example: A Recursive Procedure

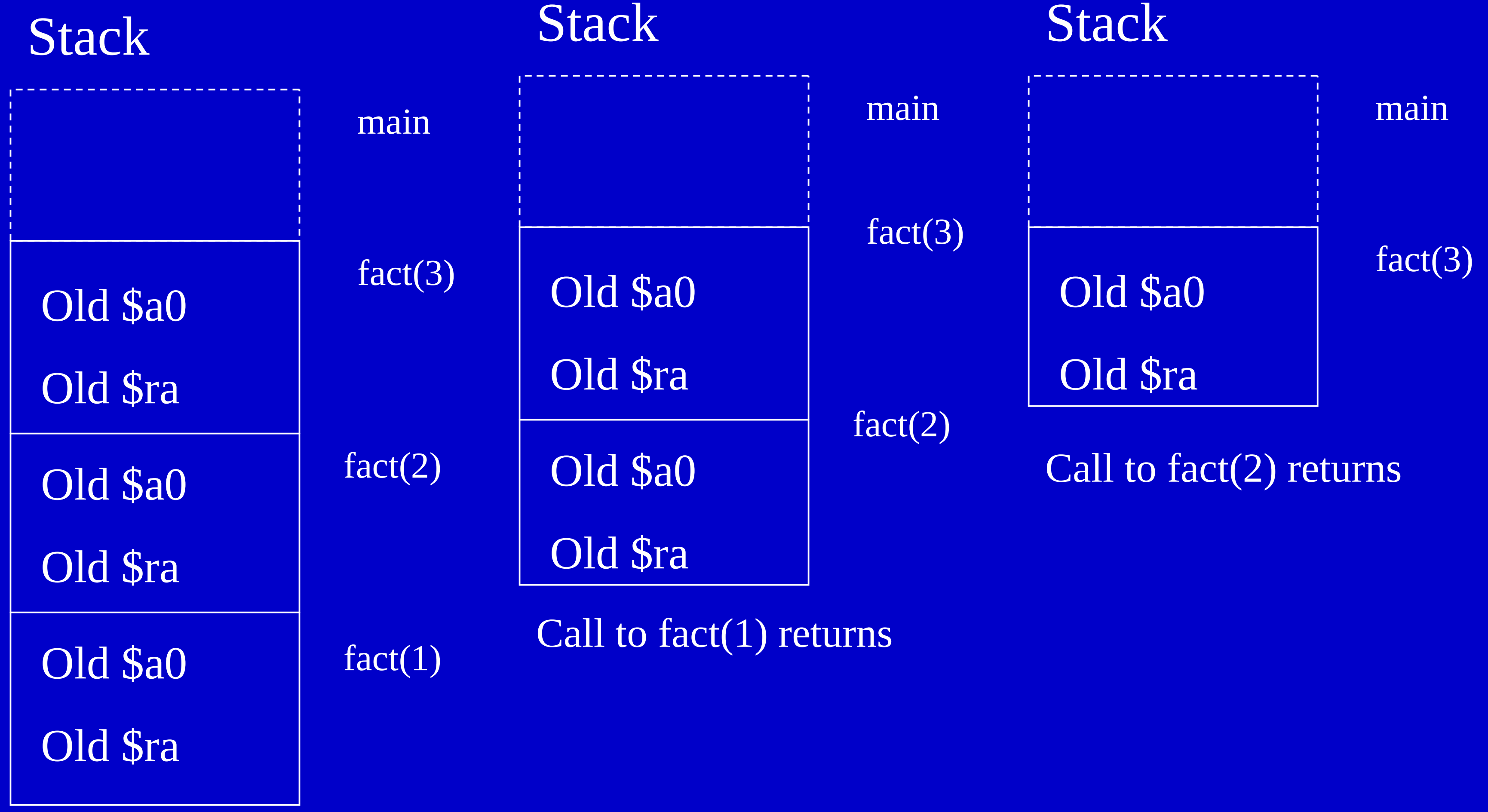
```
int fact (int n)
{
        if ( n < 1)
                return 1;
        else
                return (n * fact(n-1));
}
```

What is the generated MIPS assembly code?

- Parameter n corresponds to $a0
- This procedure makes recursive calls which means $a0 will be overwritten, and so does $ra when executing jal instruction (Why?). Implications?
- Return value will be in $v0

```
fact:                           #procedure label
        addi $sp,$sp,-8         #make room for 2 items
        sw $ra, 04($sp)         #store register $ra
        sw $a0,0($sp)           # store register $a0
        slti $t0,$a0, 1          # test if n < 1
        beq $t0, $zero,L1        # if n >= 1, go to L1
        addi $v0, $zero, 1        # return 1
        addi $sp,$sp,8           # pop 2 items off the stack
        jr $ra                   # return to caller
L1:     addi $a0,$a0,-1         # next argument is n-1
        jal fact                # call fact with argument n-1
        lw $a0,0($sp)           # restore argument n
        lw $ra,4($sp)           # restore $ra
        addi $sp,$sp,8          # adjust stack pointer
        mul $v0,$a0,$v0        # return n *fact (n-1)
        jr $ra                 #return to caller
```

Stack Frames: A call to fact(3)



Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Fig. 2.18

JAL and JR

- Single instruction to jump and save return address: jump and link (`jal`)
  - Replace:
    ```
    1008 addi $ra,$zero,1016   #$ra=1016
    1012 j sum                 #go to sum
    ```
    with:
    ```
    1012 jal sum          # $ra=1016,go to sum
    ```
  - Step 1 (link): Save address of *next* instruction into $ra
  - Step 2 (jump): Jump to the given label
  - Why have a `jal`? Make the common case fast: functions are very common
- jump register: `jr register`
  - `jr` provides a register that contains an address to jump to; usually used for procedure return

MIPS Jump, Branch, Compare

Instruction	*Example*	*Meaning*
branch on equal	**beq \$1,\$2,25**	**if (\$1 == \$2) go to PC+4+100** ***Equal test; PC relative branch***
branch on not eq.	**bne \$1,\$2,25**	**if (\$1!= \$2) go to PC+4+100** ***Not equal test; PC relative***
set on less than	**slt \$1,\$2,\$3**	**if (\$2 < \$3) \$1=1; else \$1=0** ***Compare less than; 2's comp.***
set less than imm.	**slti \$1,\$2,100**	**if (\$2 < 100) \$1=1; else \$1=0** ***Compare < constant; 2's comp..***
jump	**j 10000**	**go to 10000 *26-bit+4-bit of PC***
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***

Why Procedure Conventions?

- Definitions
  - Caller: function making the call, using jal
  - Callee: function being called
- Procedure conventions as a contract between the Caller and the Callee
- If both the Caller and Callee obey the procedure conventions, there are significant benefits
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

Caller’s Rights, Callee’s Rights

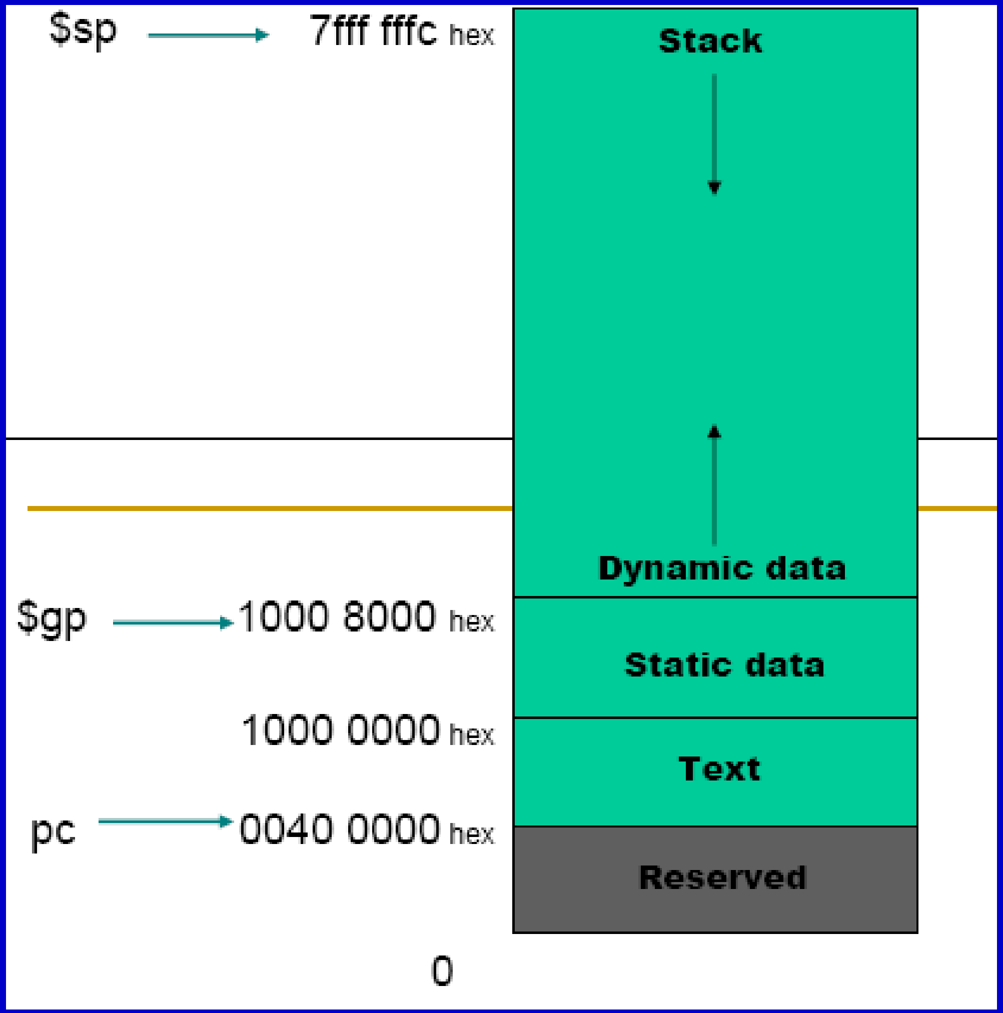
- Callees’ rights:
  - Right to use VAT registers freely
  - Right to assume arguments are passed correctly
- To ensure callees’s right, caller saves registers:

– Return address	$ra
– Arguments	$a0, $a1, $a2, $a3
– Return value	$v0, $v1
– $t Registers	$t0 - $t9

- Callers’ rights:
  - Right to use S registers without fear of being overwritten by callee
  - Right to assume return value will be returned correctly
- To ensure caller’s right, callee saves registers:

– $s Registers	$s0 - $s7

Memory Allocation for Program and Data



Representation of Characters

- ASCII (American Standard Code for Information Interchange)
  - Uses 8 bits to represent a character
  - MIPS provides instructions to move bytes:
    - lb $t0, 0($sp)#Read byte from source
    - sb $t0, 0($gp)#Write byte to destination
- Unicode
  - Uses 16 bits to represent a character
  - MIPS provides instructions to move 16 bits:
    - lh $t0, 0($sp) #Read halfwordfrom source
    - sh $t0, 0($gp) #Write halfwordto destination

2.9 MIPS Addressing for 32-Bit Immediates and Addresses

32-Bit Immediate Operands

- If constants are bigger than 16-bit, e.g., 0xABABCDCD

```
lui     $S0, 0xABAB
ori     $S0, $S0, 0xCDCD
```

Addressing in Branches and Jumps

- J-type

6 bits	26 bits

- I-type

6 bits	5 bits	5 bits	16 bits

  – Program counter = Register + Branch address
    - PC-relative addressing
      – We can branch within $\pm 2^{15}$ words of the current instruction.

  – Conditional branches are found in loops and in if statements, so they tend to branch to a nearby instruction.

J-type

- 26-bit field is sufficient to represent 32-bit address?
  - PC is 32 bits
    - The lower 28 bits of the PC come from the 26-bit field
      - The field is a word address
      - It represents a 28-bit byte address
    - The higher 4 bits
      - Come from the original PC content
- An address boundary of 256 MB (64 million instructions)

Branching Far Away

- If we need branch farther than can be represented in the 16 bits of the conditional branch instruction
  - Ex: beq $s0, $s1, L1
    - L1 with 16 bits is not sufficient
    - The new instructions replace the short-address conditional branch:

```
        bne $S0, $S1, L2
        j     L1
    L2:
```

Addressing Modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4 ← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct / Absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]

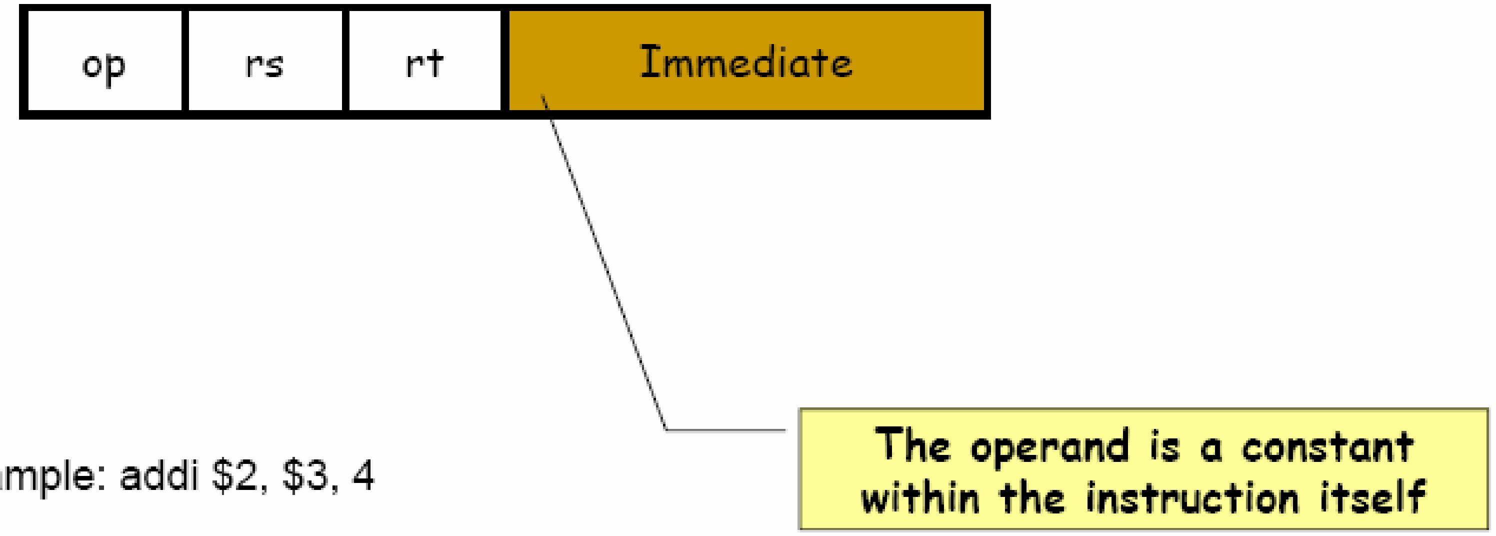
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]
		R2 ← R2+d

Auto-decrement	Add R1,-(R2)	R2 ← R2-d
		R1 ← R1+Mem[R2]

Scaled	Add R1,100(R2)[R3]	R1 ← R1+ Mem[100+R2+R3*d]

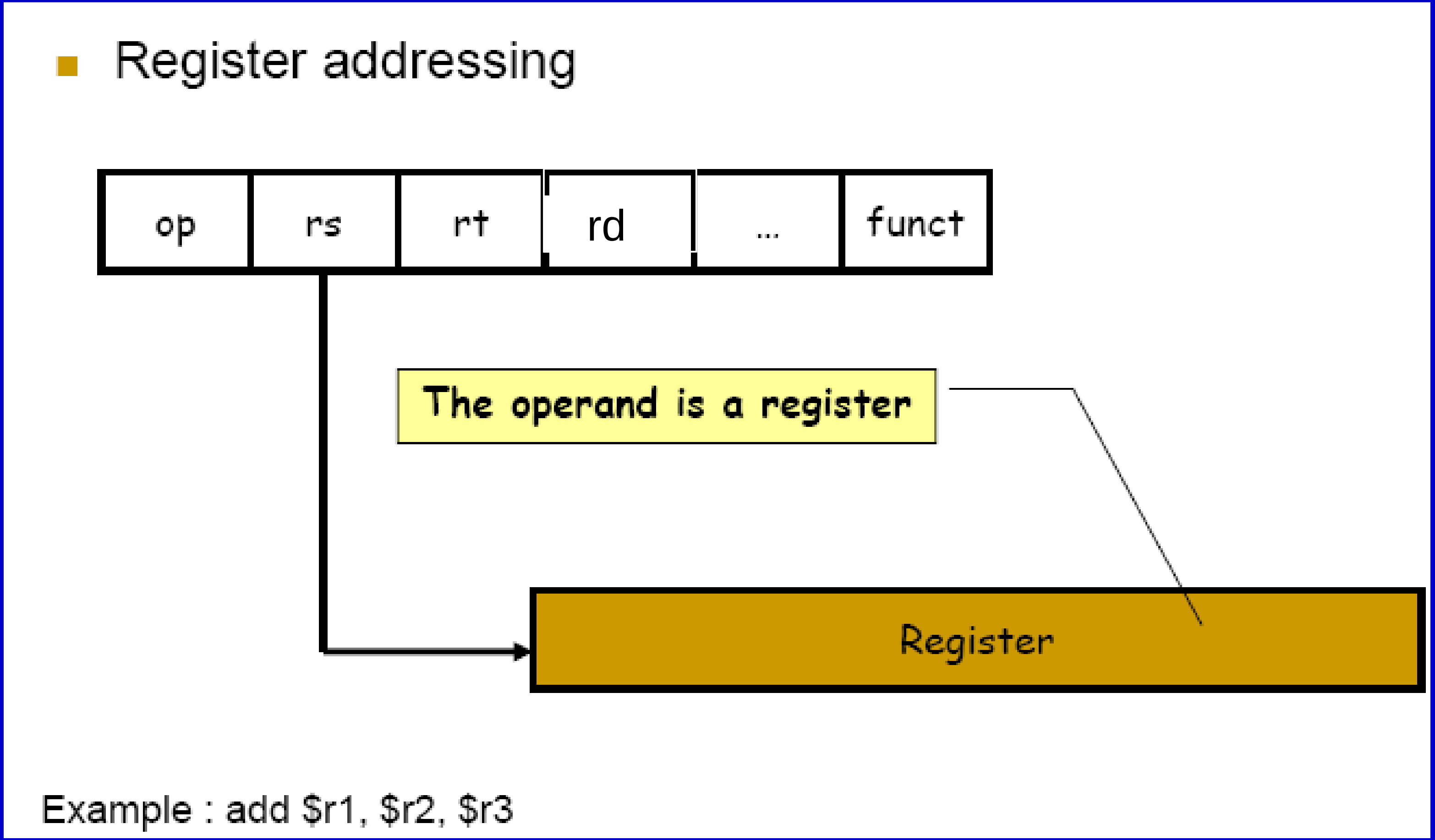
MIPS Addressing Mode (1)

- Immediate addressing



Example: addi $2, $3, 4

MIPS Addressing Mode (2)

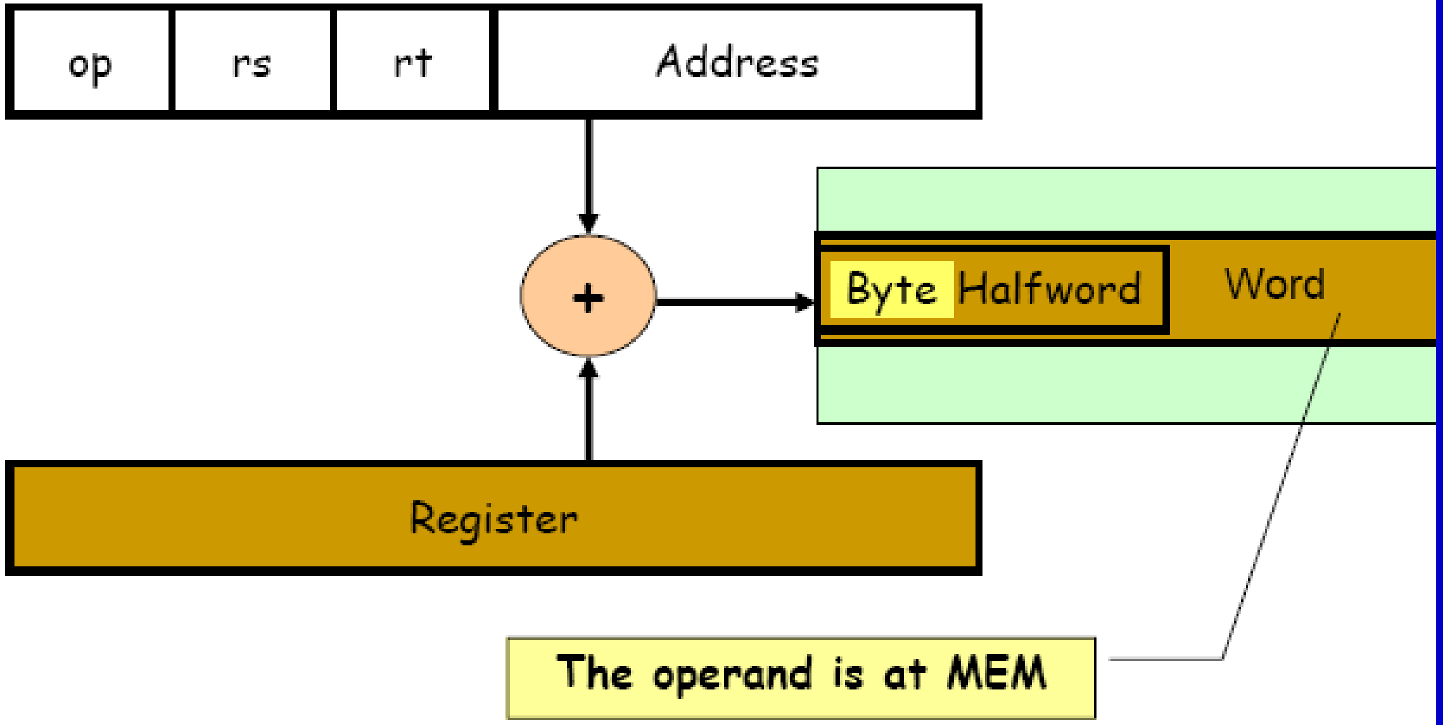


- Register addressing

Example : add $r1, $r2, $r3

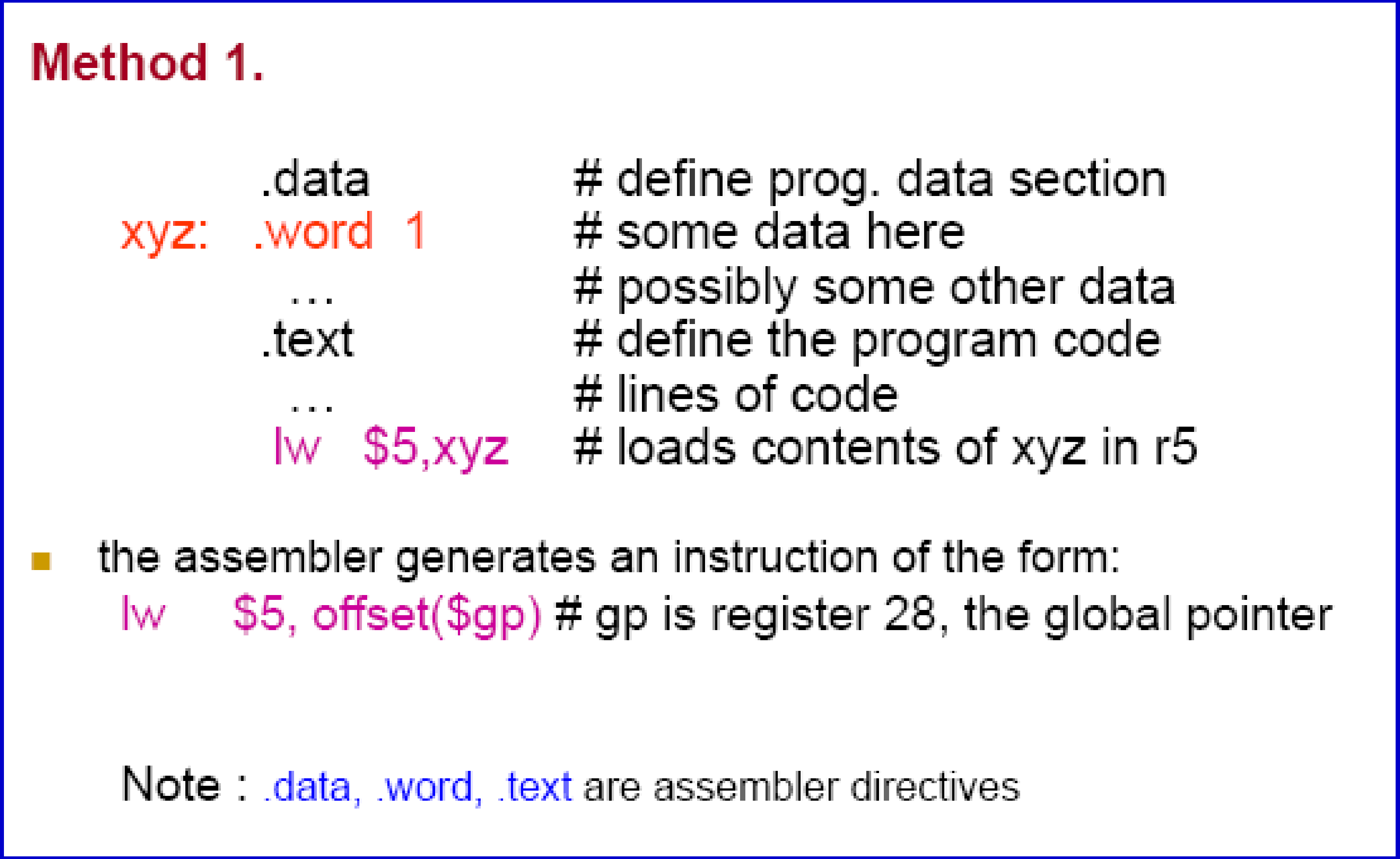
MIPS Addressing Mode (3)

- Base addressing



Example : lw $2, 100($3)

How to Get the Base Address in the Base Register



Method 1.

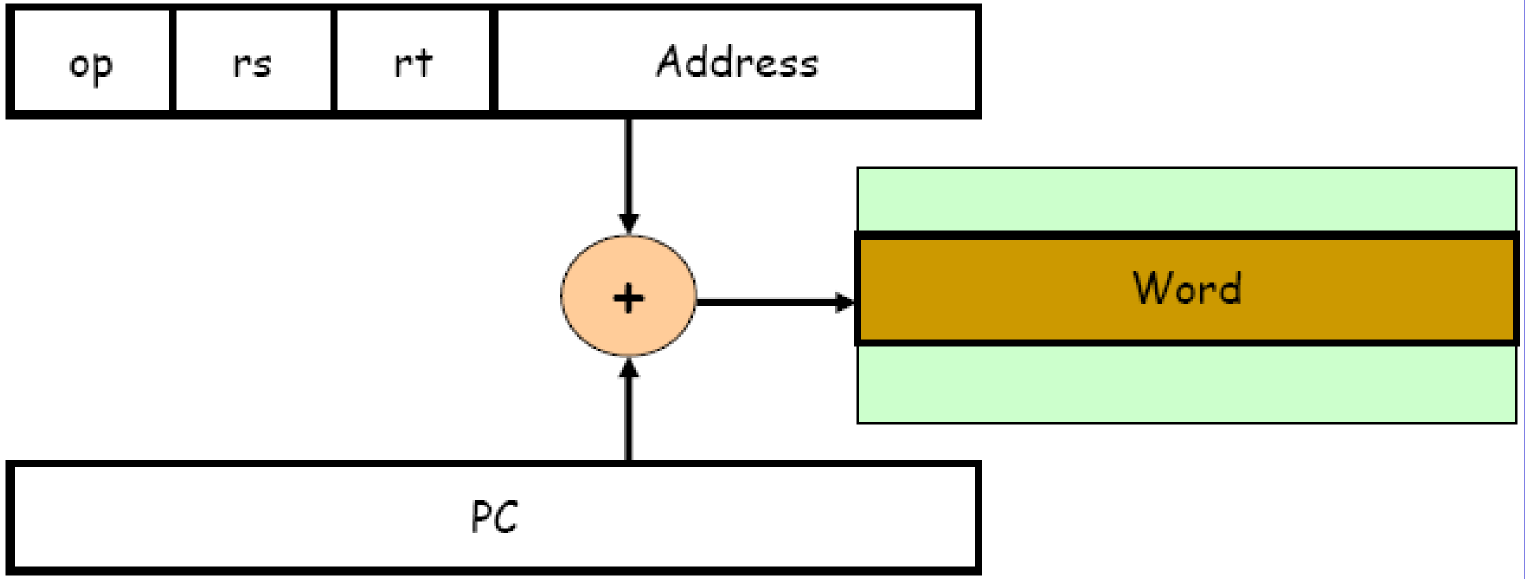
```
            .data           # define prog. data section
     xyz:   .word  1        # some data here
             ...            # possibly some other data
            .text           # define the program code
             ...            # lines of code
             lw   $5,xyz    # loads contents of xyz in r5
```

- the assembler generates an instruction of the form:
  `lw     $5, offset($gp)` # gp is register 28, the global pointer

Note : .data, .word, .text are assembler directives

MIPS Addressing Mode (4)

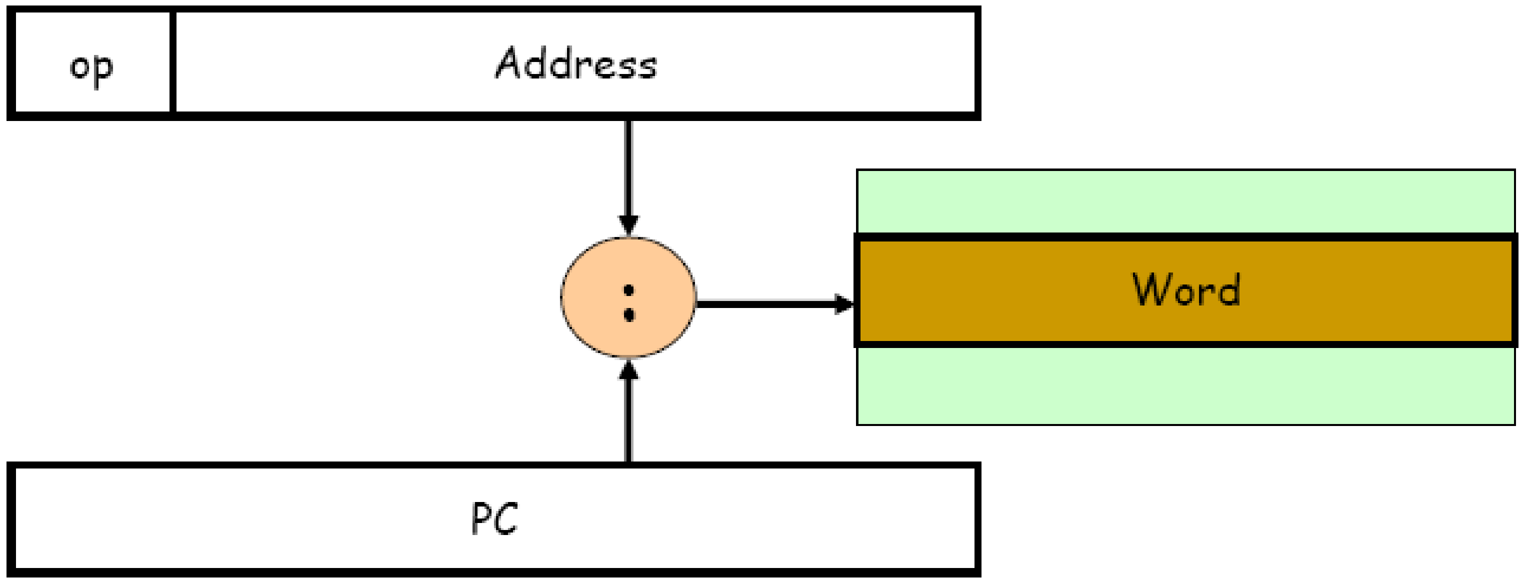
- PC-relative addressing



Example : beq $2, $3, 100

MPIS Addressing Mode (5)

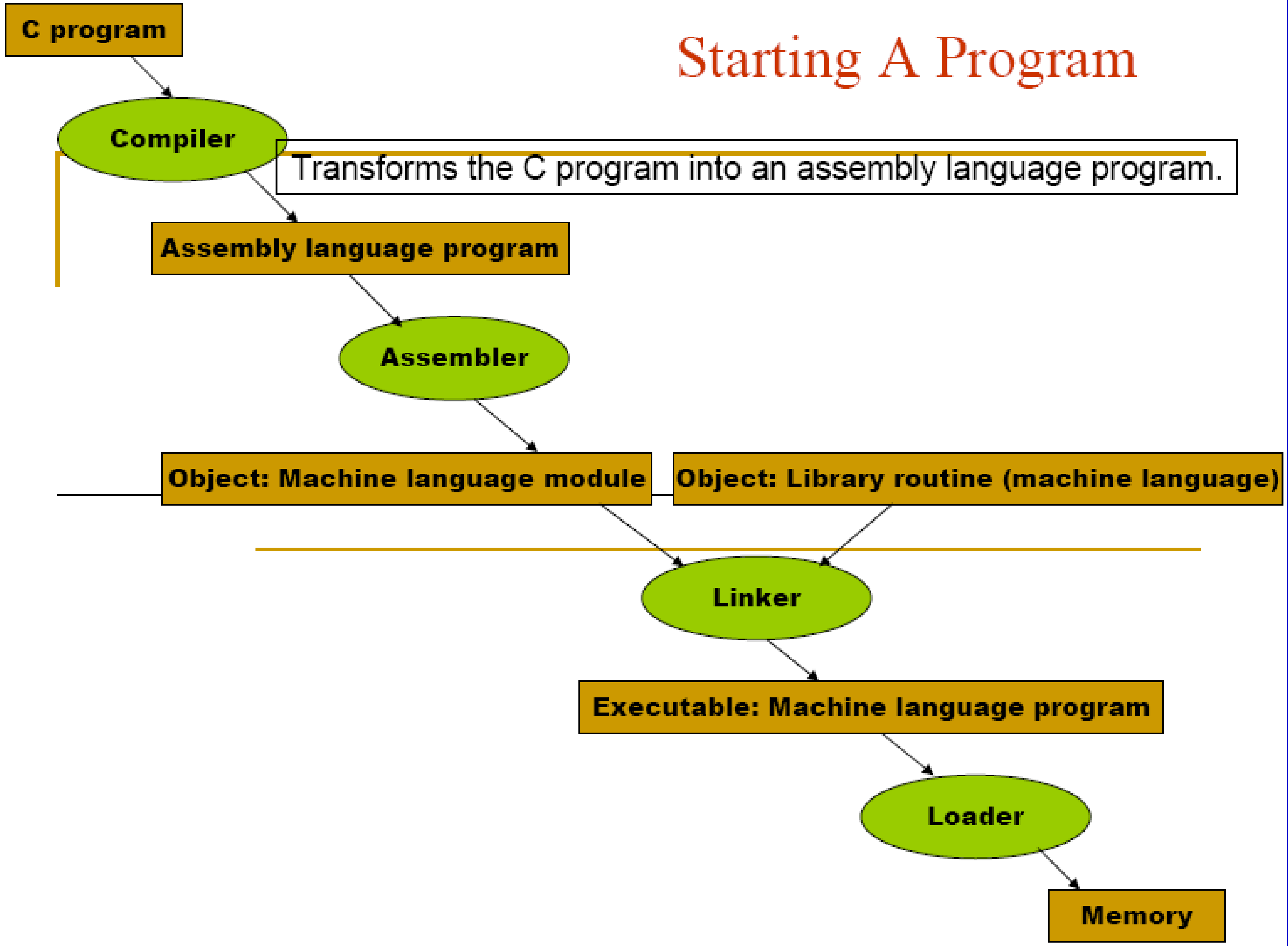
- Pseudodirect addressing



Example : j 100

2.10 Translating and Starting a Program

Starting A Program

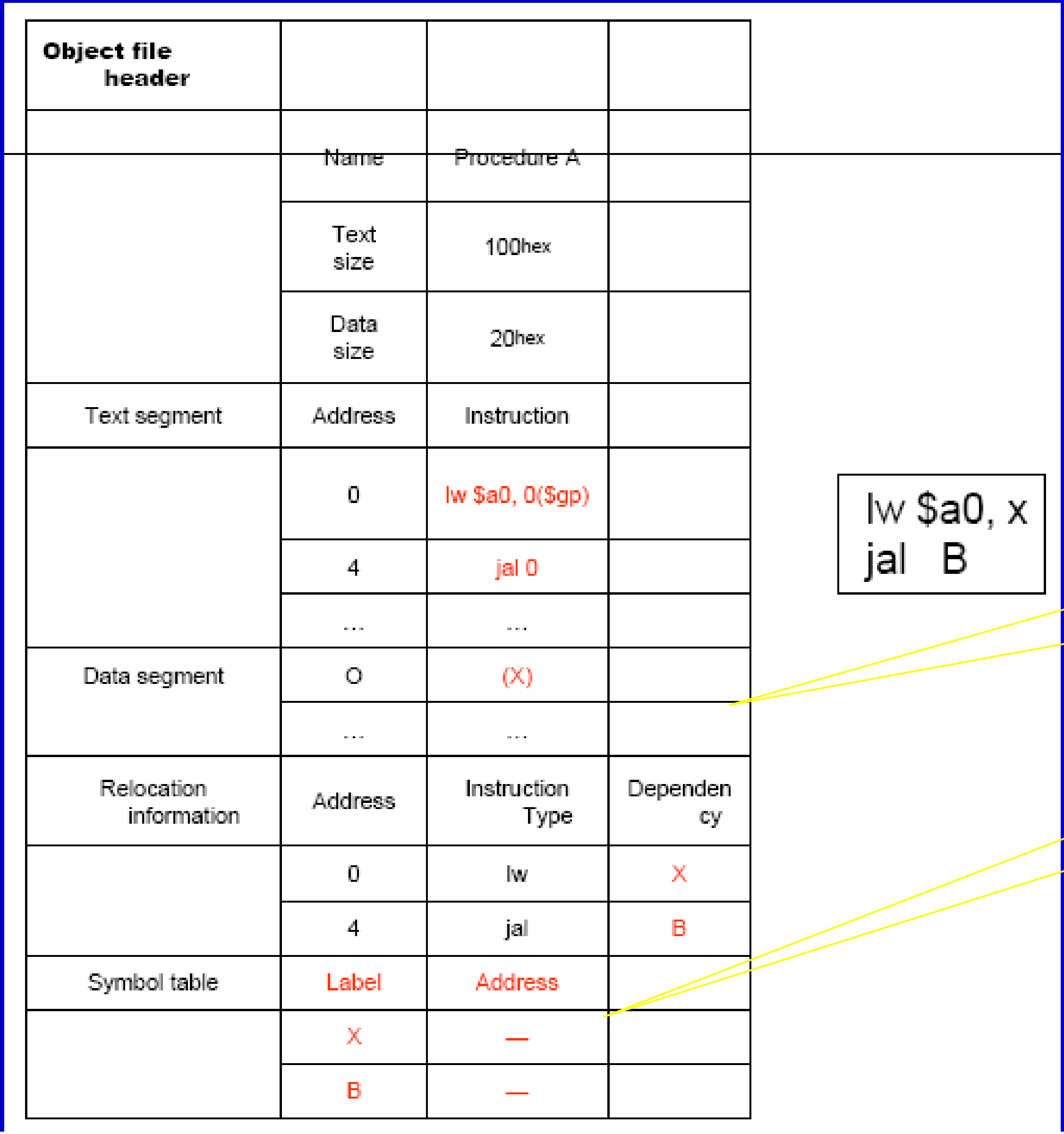

C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Transforms the C program into an assembly language program.

Assembler

- Assembler
  - The assembler turns the assembly language program (pseudoinstructions) into an object file.
    - An object file contains
      - machine language instructions
      - Data
      - ..
  - Symbol table: A table that matches names of labels to the addresses of the memory words that instruction occupy.
  - In MIPS
    - Register $at is reserved for use by the assembler.

An Object File for UNIX Systems



Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction Type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	

```
lw $a0, x
jal   B
```

static data segment

external references

Linker (Link editor)

- Linker takes all the independently assembled machine language programs and “stitches” them together to produce an executable file that can be run on a computer.
- There are three steps for the linker:
  - – 1.Place code and data modules symbolically in memory.
  - – 2.Determine the addresses of data and instruction labels.
  - – 3.Patch both the internal and external references.

Linker

- The linker use the relocation information and symbol table in each object module to resolve all undefined labels.
- If all external references are resolved, the linker next determines the memory locations each modules will occupy.

Example

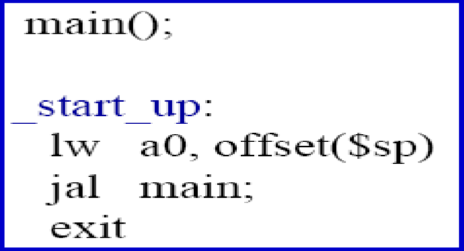
```
% gcc -c main.cc
% gcc -c a.c
% gcc -c b.c
% gcc -o hello_world main.o a.o b.o
```

The first 3 commands have each taken one source file, and compiled it into something called "object file", with the same names, but with a '.o' suffix. The object file contains the code for the source file in machine language, but with some unresolved symbols.

The 4th command links the 3 object files into one program. The linker (which is invoked by the compiler now) takes all the symbols from the 3 object files, and links them together.

Loader

- Read the executables file header to determine the size of the text and data segments
- Creates an address space large enough for the text and data
- Copies the instructions and data from the executable file into memory
- Copies the parameters (if any) to the main program onto the stack
- Initializes the machine registers and sets the stack pointer the first free location
- Jump to a start-up routine which copies the parameters into the argument registers



```
main();

_start_up:
  lw   a0, offset($sp)
  jal  main;
  exit
```

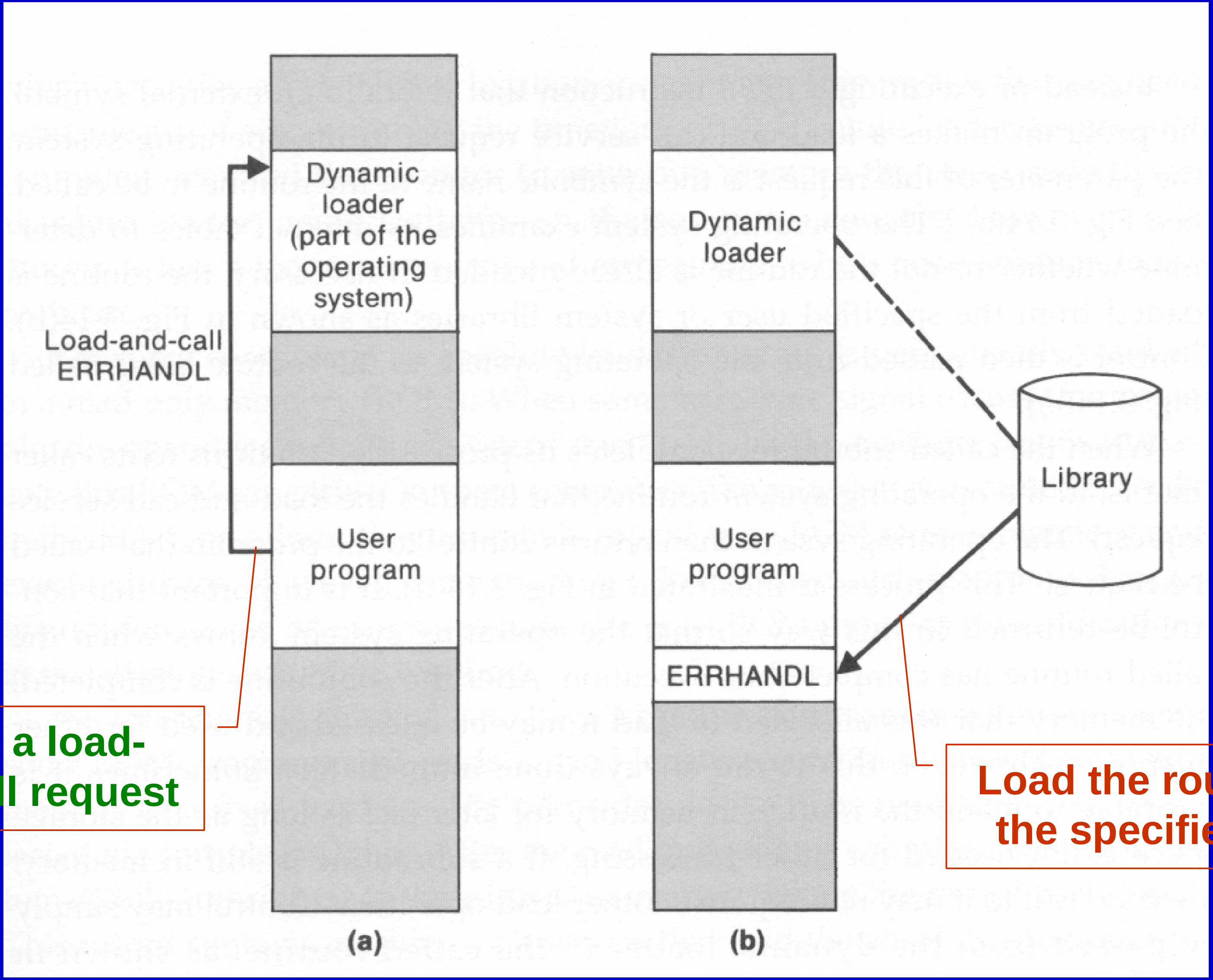
Dynamically Linked Libraries (DLL)

- Disadvantages with traditional statically linked library
  - Library updates
  - Loading the whole library even if all of the library is not used
    - The standard C library is 2.5 MB.
- Dynamically linked library
  - The libraries are not linked and loaded until the program is run.
  - Lazy procedure linkage
    - Each routine is linked only after it is called.

Dynamic Linking

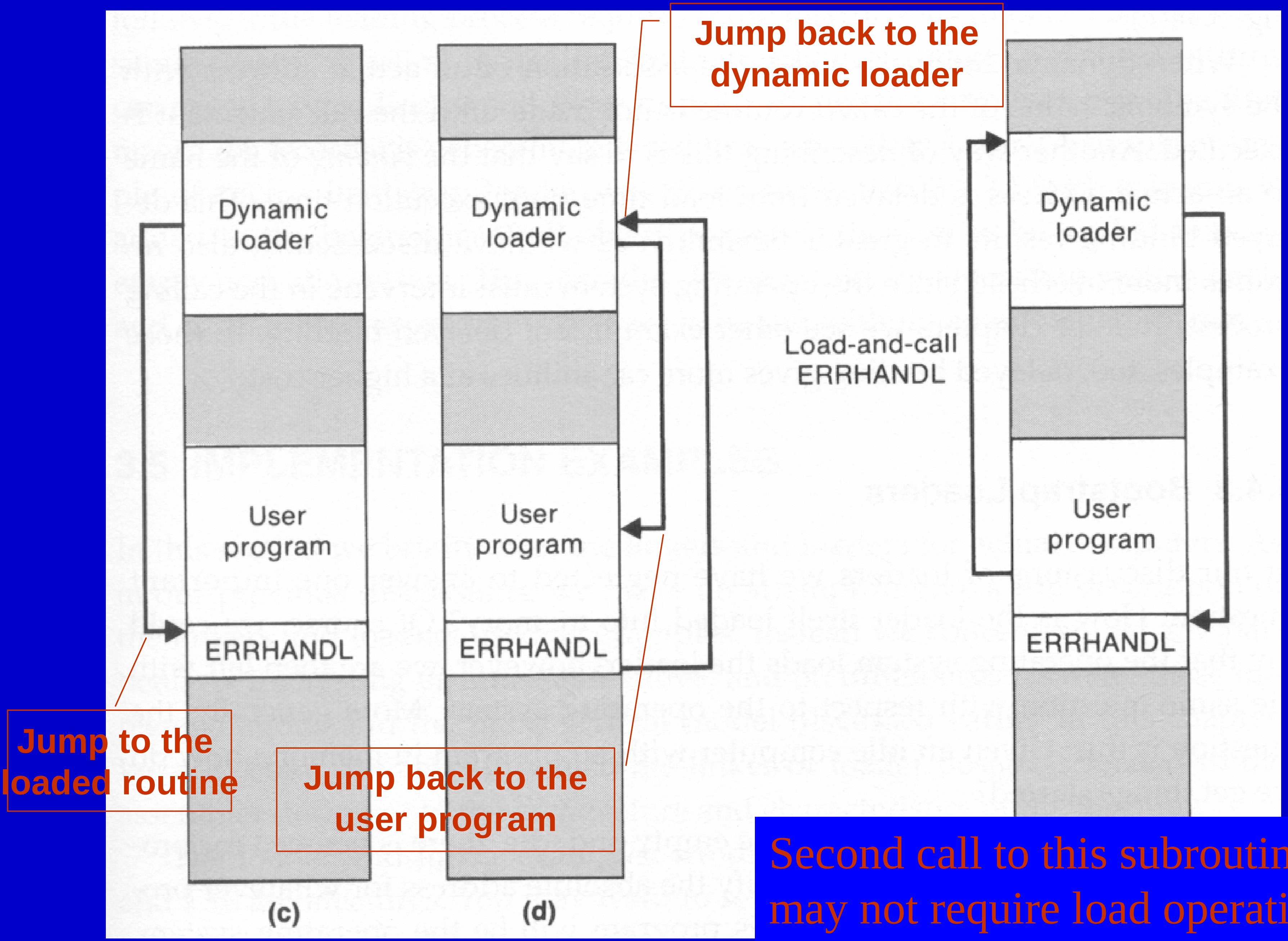
- O.S. services request of dynamic linking
  - Dynamic loader is one part of the OS
  - Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the OS
- Example
  - When call a routine, pass routine name as parameter to O.S. (a)
  - If routine is not loaded, O.S. loads it from library and pass the control to the routine (b and c)
  - When the called routine completes it processing, it returns to the caller (O.S.) (d)
  - When call a routine and the routine is still in memory, O.S. simply passes the control to the routine (e)

Example of Dynamic Linking



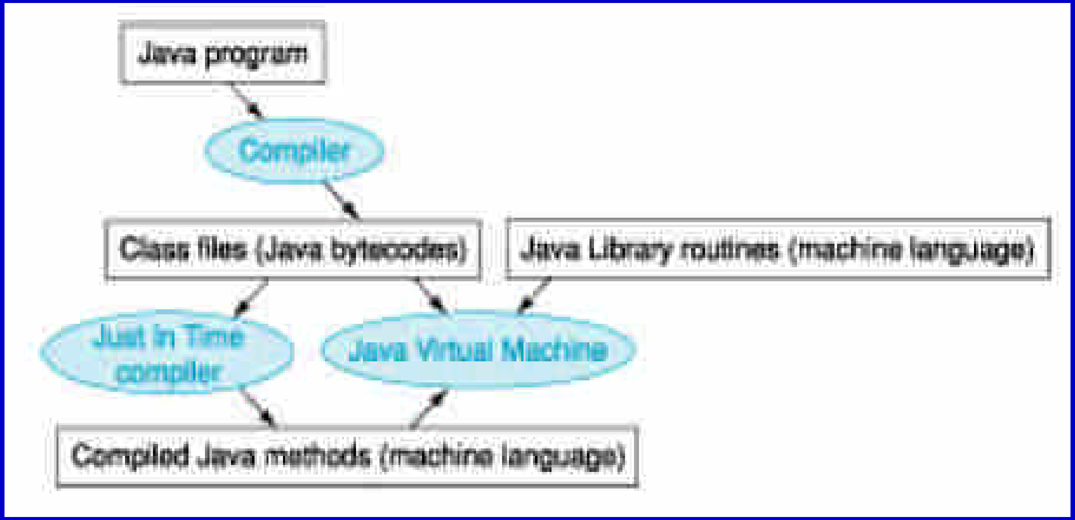
Issue a load-and-call request

Load the routine from the specified library



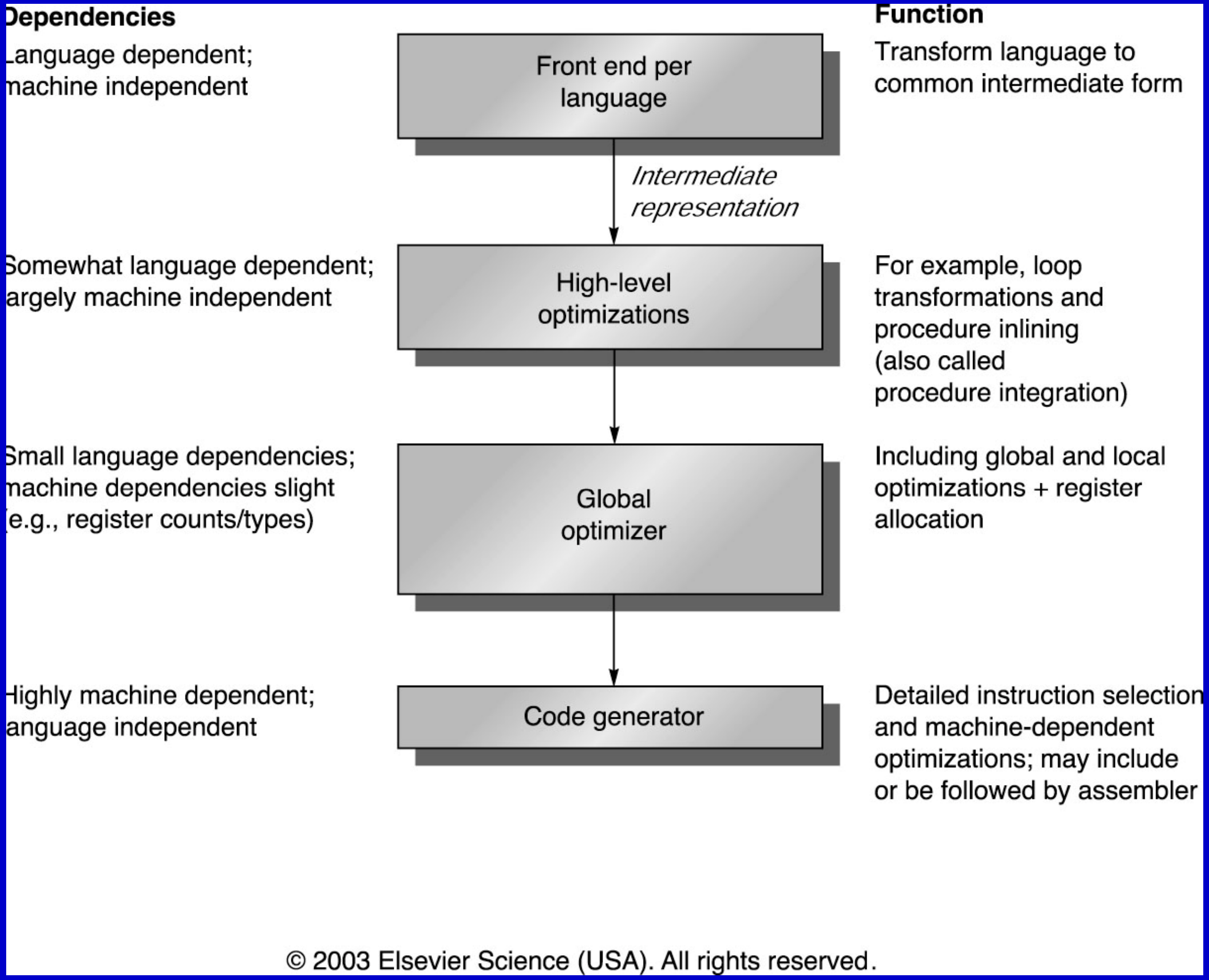
Second call to this subroutine may not require load operation.

Starting a Java Program


Java program
Compiler
Class files (Java bytecodes)
Java Library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

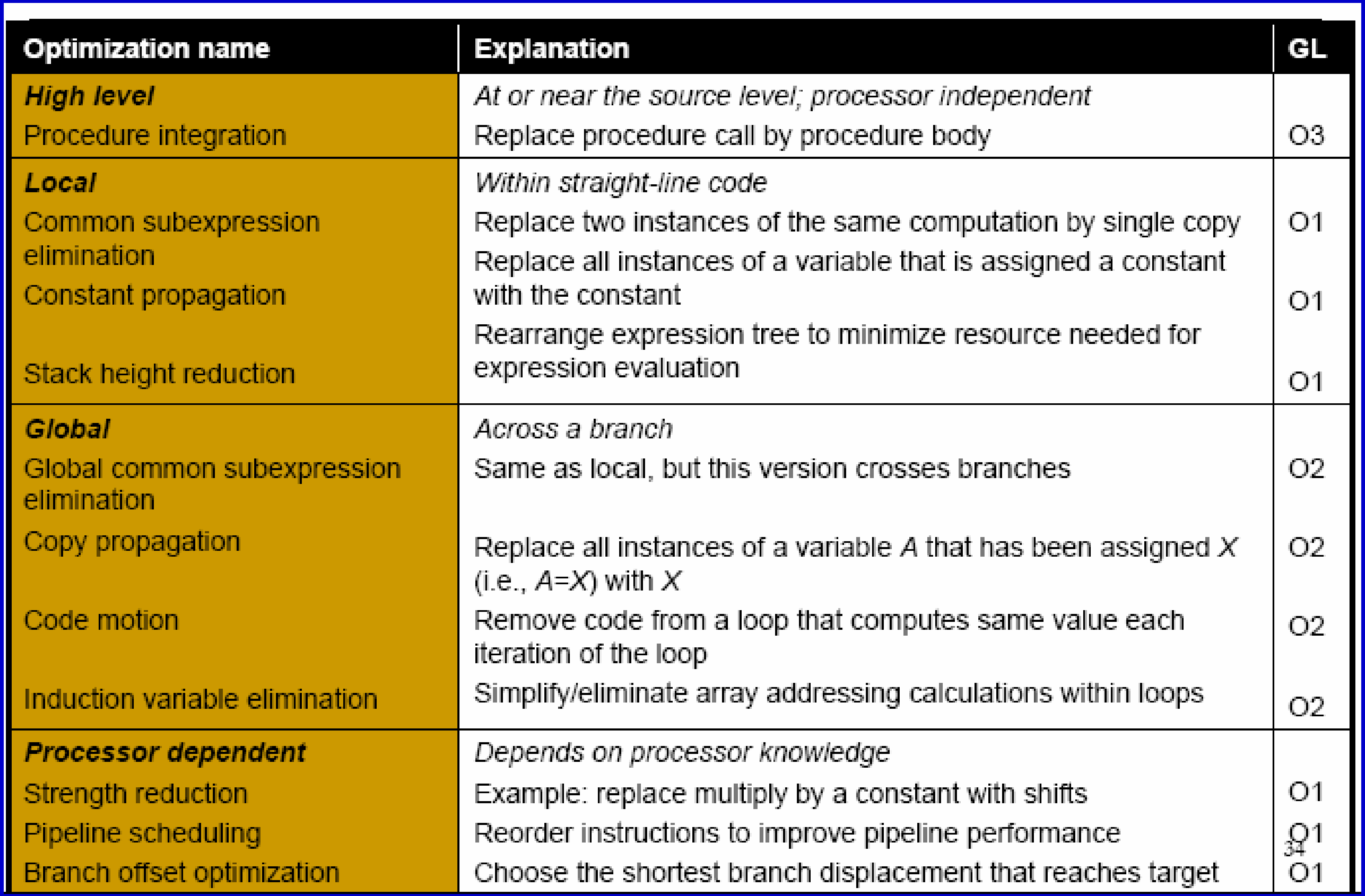
- **Java Virtual Machine (JVM):** The program that interprets Java bytecodes
  - Low performance
- **Just In Time Compiler (JIT):** profile the running program to find where the hot methods are, and then compile them into the native instruction set on which the virtual machine is running.
  - The program can run faster each time it is run.

How Compilers Optimize



Dependencies
Language dependent; machine independent
Front end per language
Transform language to common intermediate form
Intermediate representation
Somewhat language dependent; largely machine independent
High-level optimizations
For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Global optimizer
Including global and local optimizations + register allocation
Highly machine dependent; language independent
Code generator
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler
Function

Compiler Optimization Summary



Optimization name	Explanation	GL
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resource needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A*=*X*) with *X*	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Example: replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

34

To Summarize

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
branch	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to $ra	For switch, procedure return
tional jump	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Summary: MIPS ISA (1/2)

- 32-bit fixed format instructions (3 formats)
- 32 32-bit GPR (R0 = zero), 32 FP registers, (and HI LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instructions
- Memory is byte-addressable with a single addressing mode: base+displacement
  - 16-bit immediate plus LUI
- Decision making with conditional branches: beq, bne
  - Often compare against zero or two registers for =
  - To help decisions with inequalities, use: "Set on Less Than"called slt, slti, sltu, sltui
- Jump and link puts return address PC+4 into link register $ra (R31)
- Branches and Jumps were optimized to address to words, for greater branch distance

Summary: MIPS ISA (2/2)

- Immediates are extended as follows:
  - logical immediate: zero-extended to 32 bits
  - arithmetic immediate: sign-extended to 32 bits
  - Data loaded by lb and lh are similarly extended:
    lbu, lhu are zero extended; lb, lh are sign extended
- Simplifying MIPS: Define instructions to be same size as data (one word), so they can use same memory
- Stored Program Concept: Both data and actual code (instructions) are stored in the same memory
- Instructions formats are kept as similar as possible

R	`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
I	`opcode`	`rs`	`rt`	`immediate`		
J	`opcode`	`target address`				

Alternative Architectures

- Design alternative:
  - to provide more powerful operations
  - to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
    - ***"The path toward operation complexity is thus fraught with peril.***
      To avoid these problems, designers have moved toward simpler instructions"
- Let's look (briefly) at Intel IA-32

IA-32

- 1978: Intel 8086 is announced (16 bit architecture)
- 1980: 8087 floating point coprocessor is added
- 1982: 80286 increases address space to 24 bits, +instructions
- 1985: 80386 extends to 32 bits, new addressing modes
- 1989-1995: 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new “MMX” instructions are added, Pentium II
- 1999: Pentium III added another 70 instructions for streaming SIMD extension (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

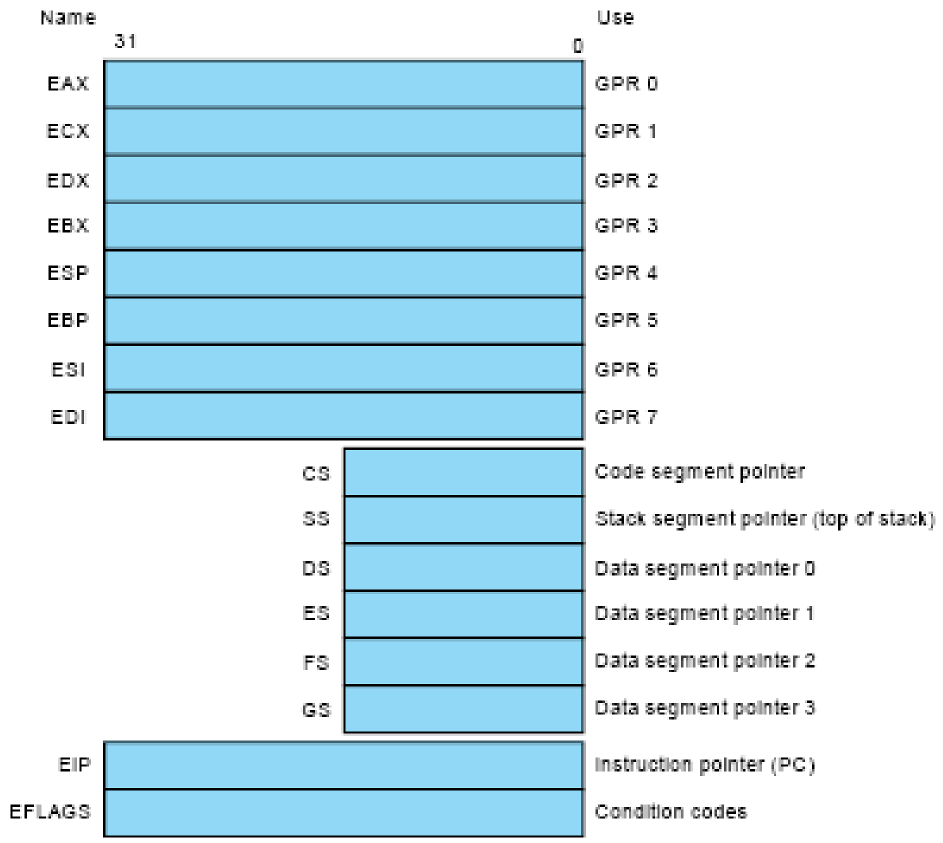
“an architecture that is difficult to explain and impossible to love”

IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

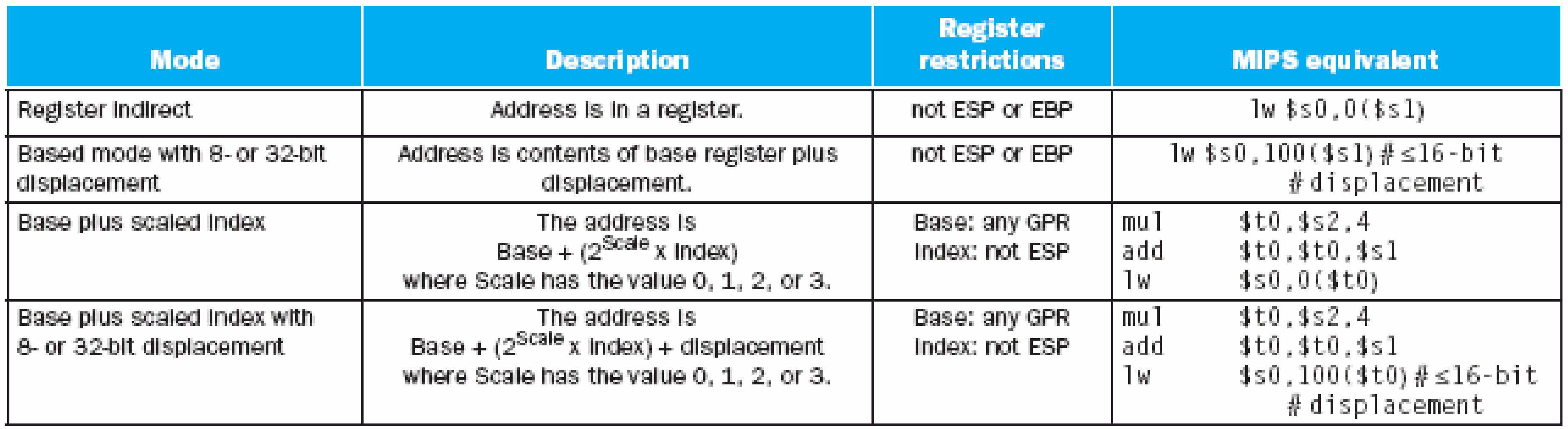
IA-32 Registers



- Fewer registers than MIPS

IA-32 Addressing Mode

- Registers are not "general purpose" – note the restrictions below



Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`

Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)#≤16-bit`
			`#displacement`

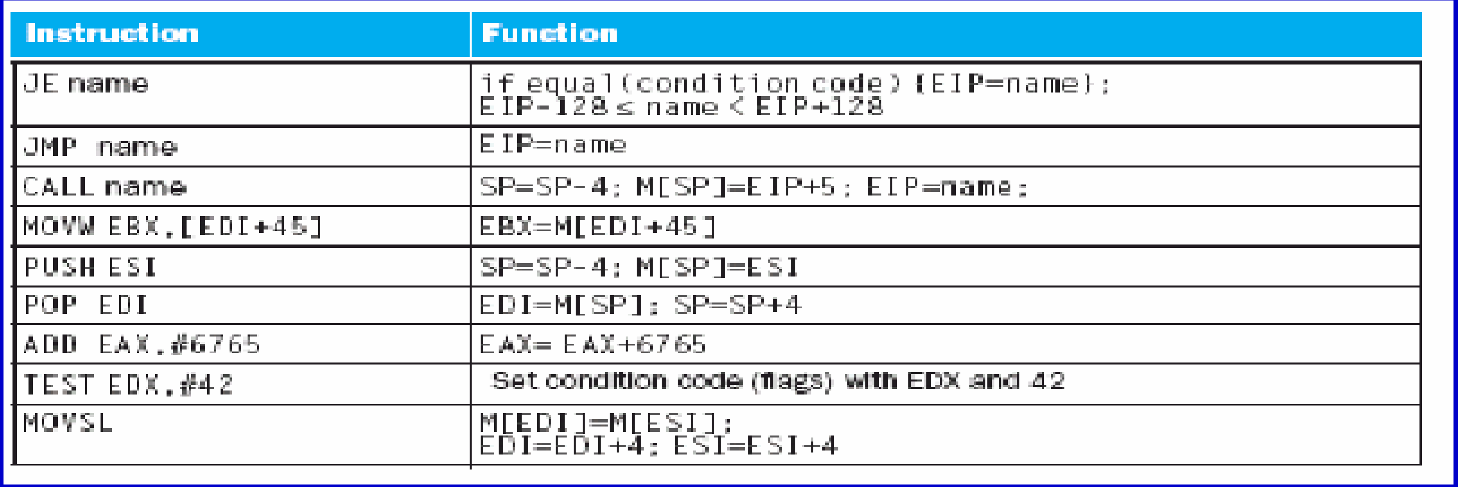
Base plus scaled index	The address is
	Base + $(2^{Scale} \times \text{Index})$
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is
	Base + $(2^{Scale} \times \text{Index})$ + displacement
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)#≤16-bit`
			`#displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typical Instructions

- Four major types of integer instructions:
  – Data movement including move, push, pop
  – Arithmetic and logical (destination register or memory)
  – Control flow (use of condition codes / flags)
  – String instructions, including string move and compare



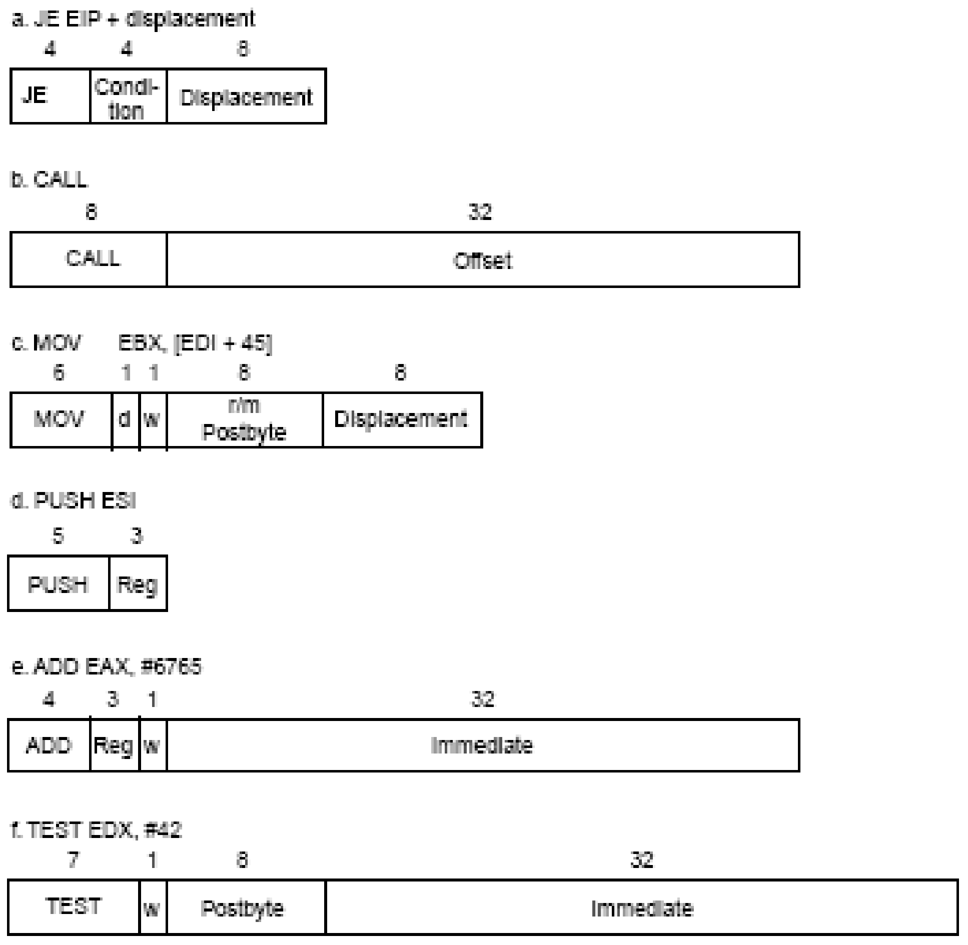
Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42

MOVSL	M[EDI]=M[ESI];
	EDI=EDI+4; ESI=ESI+4

Fig. 2.43

1.IA-32: Two-operand operation vs. MIPS: three-operand operation
2.IA-32: Register-memory vs. MIPS: register-register

IA-32 instruction Formats



IA-32 variable-length encoding vs. MIPS fixed-length encoding

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!