Computer Organization & Architecture

MIPS Assembly Programming

Lecture Contents

- Introduction to MIPS instructions set
- Arithmetic/Logical Instructions like add, sub
- MIPS logical operations
- Memory Instructions
- Decision Instructions
- More MIPS instructions

  Procedure call/return etc…

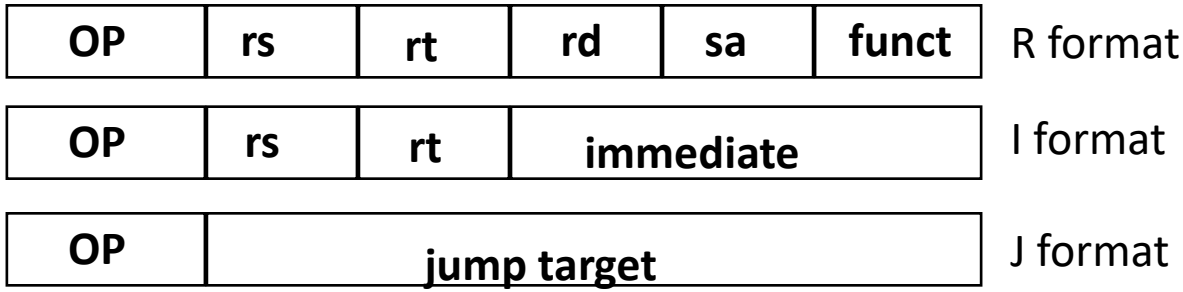
MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in detail in this class
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs
  - NEC, Silicon Graphics, Nintendo, Sony etc.

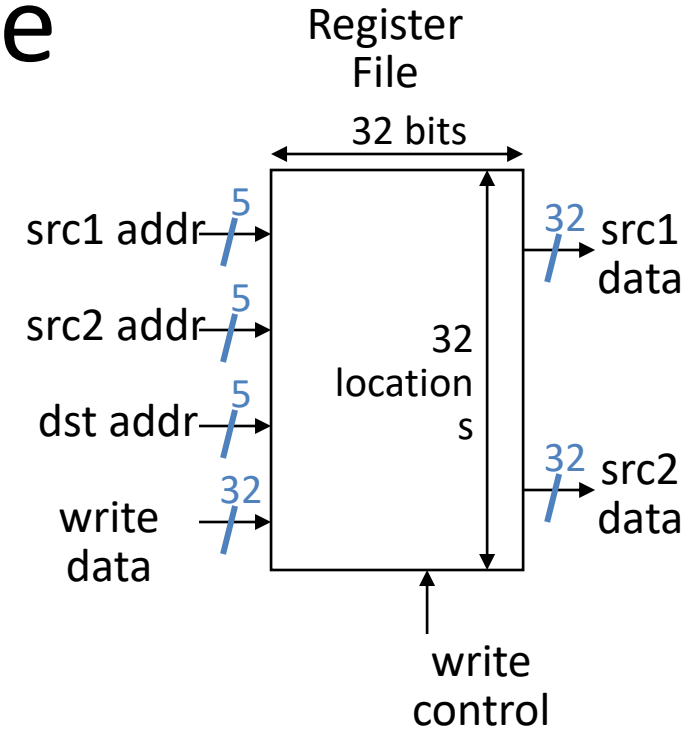
MIPS R3000 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

3 Instruction Formats: all 32 bits wide



MIPS-Register File



- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port

MIPS Registers and Numbers

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS – Programmer's Model

- 32 registers
- Size of registers = 32 bit
- Size of the word = 32 bits
- Size of the data bus = 32 bits
- Memory size = 2^{30} words

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast
  (faster than 1 billionth of a second)

Assembly Variables: Registers

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Assembly Variables: Registers

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:

```
$0, $1, $2, … $30, $31
```

Assembly Variables: Registers

- By convention, each register also has a name to make it easier to code
- For now:

  `$16 - $23` ➔ `$s0 - $s7`

  (correspond to C variables)

  `$8 - $15` ➔ `$t0 - $t7`

  (correspond to temporary variables)

  Later will explain other 16 register names
- In general, use names to make your code more readable

C, Java variables vs. registers

- In C (and most High-Level Languages) variables declared first and given a type
  - Example:

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C `//`
- Note: Different from C.
  - C comments have format
    `/* comment */`
    so they can span many lines

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High-Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java

MIPS Addition and Subtraction

- Syntax of Instructions:
  - 1 2,3,4

  where:

  1) operation by name

  2) operand getting result (“destination”)

  3) 1st operand for operation (“source1”)

  4) 2nd operand for operation (“source2”)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers

- Addition in Assembly
  - Example: `add $s0,$s1,$s2` (in MIPS)

    Equivalent to: `a = b + c` (in C)

    where MIPS registers `$s0,$s1,$s2` are associated with C variables `a, b, c`
- Subtraction in Assembly
  - Example: `sub $s3,$s4,$s5` (in MIPS)

    Equivalent to: `d = e - f` (in C)

    where MIPS registers `$s3,$s4,$s5` are associated with C variables `d, e, f`

Addition and Subtraction of Integers

- How do the following C statement?
  - `a = b + c + d - e;`
- Break into multiple instructions
  - `add $t0, $s1, $s2 # temp = b + c`
  - `add $t0, $t0, $s3 # temp = temp + d`
  - `sub $s0, $t0, $s4 # a = temp - e`
- Notice: A single line of C may break up into several lines of MIPS.
- Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers

- How do we do this?

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

- Use intermediate temporary register

```
   add $t0,$s1,$s2   # temp = g + h
   add $t1,$s3,$s4   # temp = i + j
   sub $s0,$t0,$t1
```

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero (`$0` or `$zero`) to always have the value 0; eg
  - `add $s0,$s1,$zero` (in MIPS)
  - `f = g` (in C)
  - where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- defined in hardware, so an instruction
  - `add $zero,$zero,$s0`
  - will not do anything!

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - `addi $s0,$s1,10` (in MIPS)
  - `f = g + 10` (in C)
  - where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- Syntax similar to `add` instruction, except that last argument is a number instead of a register.

Your Call Now

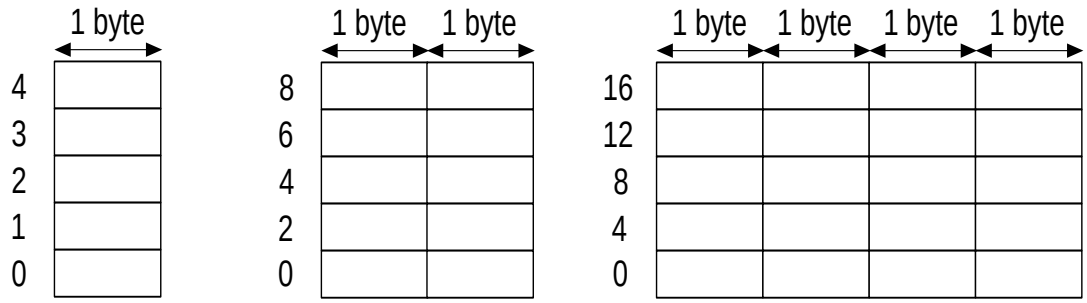
- There is no Subtract Immediate in MIPS: Why?

Addressing Modes

- What is the location of an operand?
- Three basic possibilities
  - Register: operand is in a register
    - Register number encoded in the instruction
  - Immediate: operand is a constant
    - Constant encoded in the instruction
  - Memory: operand is in memory
    - Many address modes possibilities

MIPS Memory

- MIPS memory organized as 32-bit word
- Byte Addressing
- **Big Endian**



Load from Memory Instruction

- lw register, constant (register)
- Memory address = constant + register
- Memory address = offset + Base Register
- **`lw $s1, 4($s2)`**
- C code: g = h + A[8];
- MIPs code:

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

- The base address of the array A[] is stored in the Base Register **`$s3`** with an offset of **`32`** and the effective address will be stored in **`$t0`** where as the contents of h are in register **`$s2`** and the contents of h are in register **`$s1`**

Store to Memory Instruction

- sw register, constant (register)
- Memory address = constant + register
- Memory address = offset + Base Register
- **`sw $s1, 4($s2)`**
- C code: A[12] = h + A[8];
- MIPs code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

Pointers v. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) `int`, an `unsigned int`, a pointer (memory address), and so on
- If you write `add $t2,$t1,$t0`

  then `$t0` and `$t1` better contain values
- If you write `lw $t2,0($t0)`

  then `$t0` better contain a pointer
- Don't mix these up!

Control instructions

- Involve decision making: if a certain condition is satisfied then do something else do something else.
- MIPS assembly language includes two decision making instructions **(conditional branches)**
  - beq register1, register2, L1 #go to L1 if contents of register1 and 2 are equal
  - bne register1, register2, L1 #go to L1 if contents of register1 and 2 are not equal

Branch instructions

• beqz $s0, label	if $s0==0	goto label
• bnez $s0, label	if $s0!=0	goto label
• bge $s0, $s1, label	if $s0>=$s1	goto label
• ble $s0, $s1, label	if $s0<=$s1	goto label
• blt $s0, $s1, label	if $s0<$s1	goto label
• beq $s0, $s1, label	if $s0==$s1	goto label
• bne $s0, $s1, label	if $s0!=$s1	goto label
• bgez $s0, label	if $s0>=0	goto label

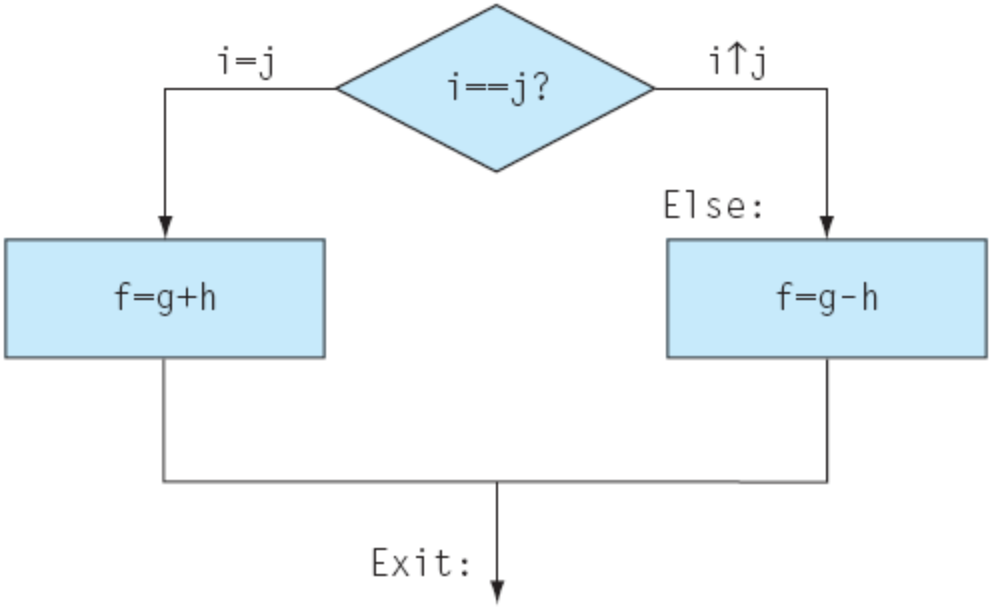
Control instructions

Convert to assembly:

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

Assembly code

```
bne   $s3, $s4, Else
add   $s0, $s1, $s2
j       Exit
Else:   sub   $s0, $s1, $s2
 Exit:
```



Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll   $t1, $s3, 2
       add   $t1, $t1, $s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Exit
       addi  $s3, $s3, 1
       j     Loop
Exit:
```

Unconditional branch

Example

Convert to assembly:

For (i=0; i=<50; i = i +1)
Z[i] = X[i] +Y[i];

initial & final values of i are in s0 & s4 respectively while base addresses of arrays X, Y & Z are in s1, s2 & s3 respectively.

```
addi    $s0, $s0, 0          # initial index value
addi    $s4, $s4, 50         # final index value
Loop:   sll   $t0, $s0, 2    # index multiplied by 4
        add   $t1, $t0, $s1  # base address of X in s1
        add   $t2, $t0, $s2  # base address of Y in s2
        add   $t3, $t0, $s3  # base address of Z in s3
        lw    $t4, 0($t1)    # load at effective address of X
        lw    $t5, 0($t2)    # load at effective address of Y
        add   $t6, $t4, $t5  # addition of two arrays X+Y
        sw    $t6, 0($t3)    # store at effective address of Z
        addi  $s0, $s0,1     # increment in index value
        bne   $s0, $s4, Loop # check whether i=<50
```

More control instructions

- To check equality is important
- What if we want to check if a certain variable is less than or greater than another variable
  - slt $t0, $S3, $S4
  - slti $t0, $S2, 10 # $t0 = 1 if $s2 < 10

Summary of MIPS assembly language instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $$s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,L`	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	`slt $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	`j L`	go to L	Jump to target address

Summary of MIPS machine language instructions

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Procedures

- A stored subroutine that performs a specific task based on the parameters with which it is provided
- Each procedure (function, subroutine) maintains a scratchpad of register values – when another procedure is called (the callee), the new procedure takes over the scratchpad – values may have to be saved so we can safely return to the caller

- In the execution of the procedure, the program must follow these six steps:

1. parameters (arguments) are placed where the callee can see them
2. control is transferred to the callee
3. acquire storage resources for callee
4. execute the procedure
5. place result value where caller can access it
6. return control to caller

Registers

- $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 the point of origin

Procedures: Jump and link instruction

- A special register (storage not part of the register file) maintains the address of the instruction currently being executed – this is the *program counter* (PC)

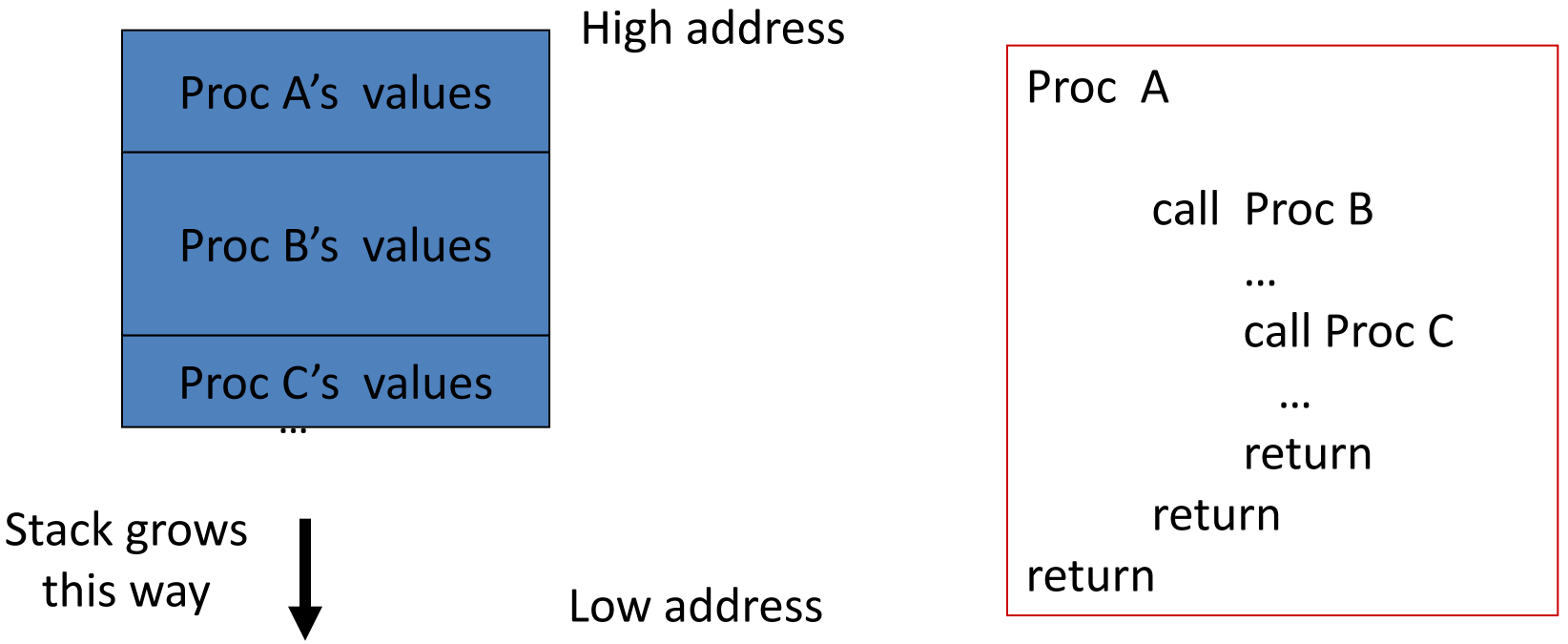
- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure's address (the PC is accordingly set to this address)
- jal NewProcedureAddress

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction

- How do we return control back to the caller after completing the callee procedure?

Procedures: Jump and link instruction, Stack

- What if more registers are required than the four parameter registers ($a0 - $a3) and two value registers ($v0 - $v1)
- The register scratchpad for a procedure seems volatile – it seems to disappear every time we switch procedures – a procedure's values are therefore backed up in memory on a stack



Procedures: Jump and link instruction, Stack, Storage management on call/return

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into its own stack space
- Arguments are copied into $a0-$a3; the jal is executed
- After the callee creates stack space, it updates the value of $sp
- Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

Example 1

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

- If g, h, i, j correspond to parameter register $a0 - $a3 and f corresponds to $s0 what will be MIPS assembly code?

Example 1

```
leaf_example
addi    $sp, $sp, -12       #making room for 3 registers $t0, $t1, $s0
sw      $t1, 8($sp)         #save $t1 for use afterwards
sw      $t0, 4($sp)         #save $t0 for use afterwards
sw      $s0, 0($sp)         #save $s0 for use afterwards
add     $t0, $a0, $a1       # $t0 = g+h
add     $t1, $a2, $a3       # $t1 = i+j
sub     $s0, $t0, $t1       # f = $t0 - $t1
add     $v0, $s0, $zero     # return f
lw      $s0, 0($sp)         # restore register $s0 for caller
lw      $t0, 4($sp)         # restore register $t0 for caller
lw      $t1, 8($sp)         # restore register $t1 for caller
addi    $sp, $sp, 12        # adjust stack to delete three items
jr      $ra                 # jump back to calling routine
```

Example 2

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

What is MIPS assembly code?

Example 2

```
fact:
    addi    $sp, $sp, -8          # adjust stack for two items
    sw      $ra, 4($sp)           # save the return address
    sw      $a0, 0($sp)           # save the argument n
    slti    $t0, $a0, 1           # test for n<1
    beq     $t0, $zero, L1        #if n>=1 go to L1
    addi   $v0, $zero, 1          # return 1
    addi   $sp, $sp, 8            # pop two items off stack
     jr     $ra                   # return to after jal
L1:
    addi    $a0, $a0, -1          #n>=1 argument gets (n-1)
    jal     fact                  # call fact with n-1
    lw      $a0, 0($sp)           # return from jal:restore argument n
    lw      $ra, 4($sp)           # restore the return address
    addi    $sp, $sp, 8           # adjust stack pointer to pop 2 items
    mul     $v0, $a0, $v0         # return n*fact(n-1)
    jr      $ra                   # return to the caller
```

Dealing with characters

- Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh
- These data types are most useful when dealing with characters, pixel values, etc.
- C employs ASCII formats to represent characters – each character is represented with 8 bits and a string ends in the null character (corresponding to the 8-bit number 0)

ASCII code of characters

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Example

The procedure strcpy copies string y to string x using the null byte termination convention of C:

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

Assume that base addresses for arrays x and y are found in $a0 and $a1 while i is in $s0.

Example

```
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
 addi    $sp, $sp, 4           # pop 1 word off stack
 jr      $ra                   # return
```