# CSE 340 (Chap -2)

## Supplementary Slides for: Function/ Procedure/ Subsection

Prepared by Fairoz Nower Khan

# Function

```
Main (){
int x=0;
int y=9;
int z = addition(x,y);
.
. }

int addition(int a, int b){
int c = a+b;
return c;
}
```

Main function is the Caller function, where a function is called, and the parameters are provided for the called function

Parameters for function can be passed using 4 Registers, that are $a0, $a1, $a2, $a3 (argument Registers)

Return address from function to origin is stored is $ra

Value can be returned from function using 2 registers: $v0 and $v1.

# Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

# Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

```
Main (){
int f=0;
f = f + 1;

int z = addition(x,y);
Int c = x-y;
}

int addition(int a, int b){
int f = a+b;
return f;
}
```

```
add $s0, $zero, $zero
addi $s0, $s0, 1
```

f □ $s0

```
jal addition
```

#jumps to addition function and

#creates link with where the function is called from, by storing the address in $ra

Save the register values used by the procedure, to recover any value that can get lost

# Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

# Leaf Procedure Example

```
Main (){
int f=0;
f= f + 1;
int z = leaf_example (1,2,4,3);
int y = f + z;
.
. }
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

jal leaf_example

- Arguments:

g □ $a0

h □ $a1

i □ $a2

j □ $a3

- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

Saved value registers ($s0 - $s7) that are used by the function are stored in Stack, to recover any value that can get lost

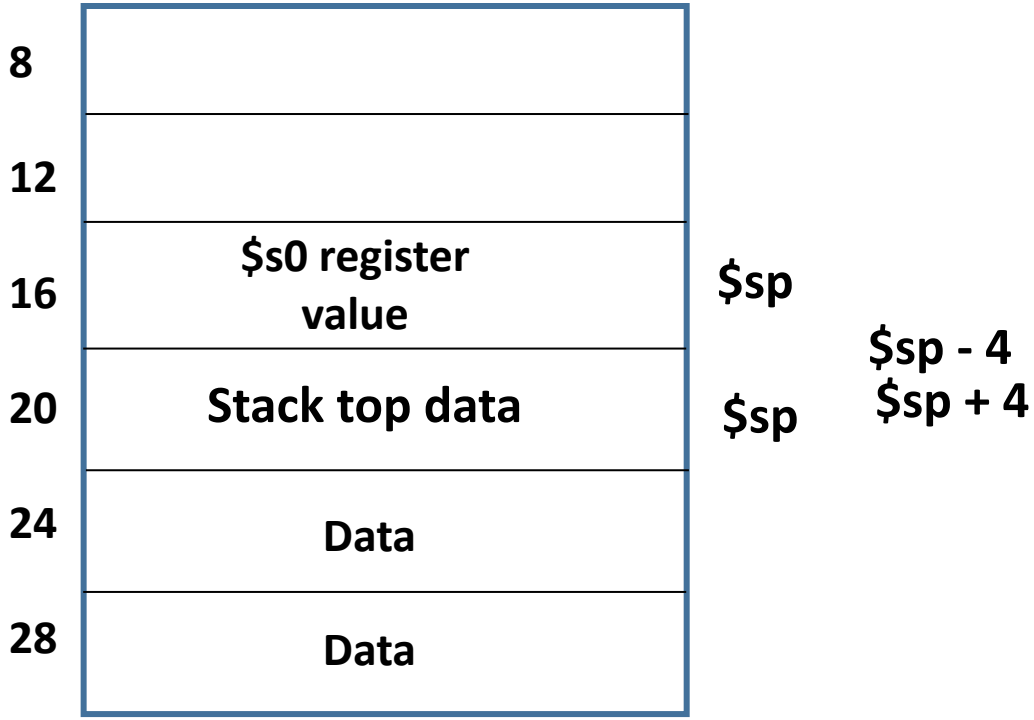
Storing data in stack

$s0 register value

```
addi $sp, $sp, -4
sw  $s0,  0($sp)
```

Retrieving data from stack

```
lw  $s0,  0($sp)
addi $sp, $sp, 4
```




When a value is inserted in stack, the stack pointer register moves 4 slots upwards,
hence $sp = $sp - 4

- Arguments:

g □ $a0, h □ $a1, i □ $a2, j □ $a3

- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

**C Code:**

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

jal leaf_example

MIPS Code:

```
leaf_example:
    addi $sp, $sp, -4
    sw  $s0,   0($sp)
    add   $t0, $a0,$a1
    add   $t1, $a2,$a3
    sub   $s0, $t0, $t1
    add   $v0, $s0, $zero
    lw   $s0,   0($sp)
    addi $sp, $sp, 4
    jr      $ra
```

# Leaf Procedure Example

```
Main (){
int f, x=0;
f= f + 1;
int z = leaf_example (1,2,4,3);
int y = f + z + x;
.
. }
```

```
int leaf_example (int g, h, i, j)
{ int f, x;
  f = (g + h) - (i + j);
  x= f+1;
  return f;
}
```

jal leaf_example

- Arguments:

g □ $a0

h □ $a1

i □ $a2

j □ $a3

- f in $s0, x in $s1 (hence, need to save $s0, $s1 on stack)
- Result in $v0

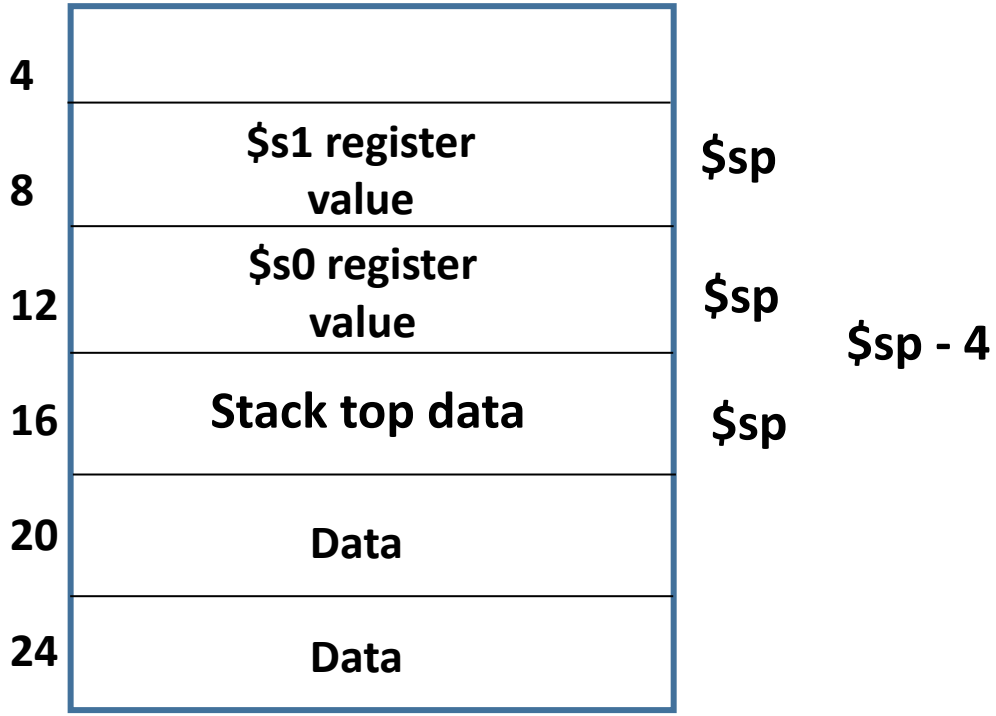
Saved value registers ($s0 - $s7) are stored in Stack used by the function, to recover any value that can get lost

Storing data in stack

$s0 register value

$s1 register value

```
addi $sp, $sp, -4
sw $s0, 0($sp)
addi $sp, $sp, -4
sw $s1, 0($sp)
```




Retreving data from stack

Last In First Out

```
lw $s1, 0($sp)
addi $sp, $sp, 4
lw $s0, 0($sp)
addi $sp, $sp, 4
```

When a value is inserted in stack, the stack pointer register moves 4 slots upwards, hence $sp = $sp - 4

- Arguments:

g □ $a0, h □ $a1, i □ $a2, j □ $a3

- f in $s0, x in $s1 (hence, need to save $s0, $s1 on stack)
- Result in $v0

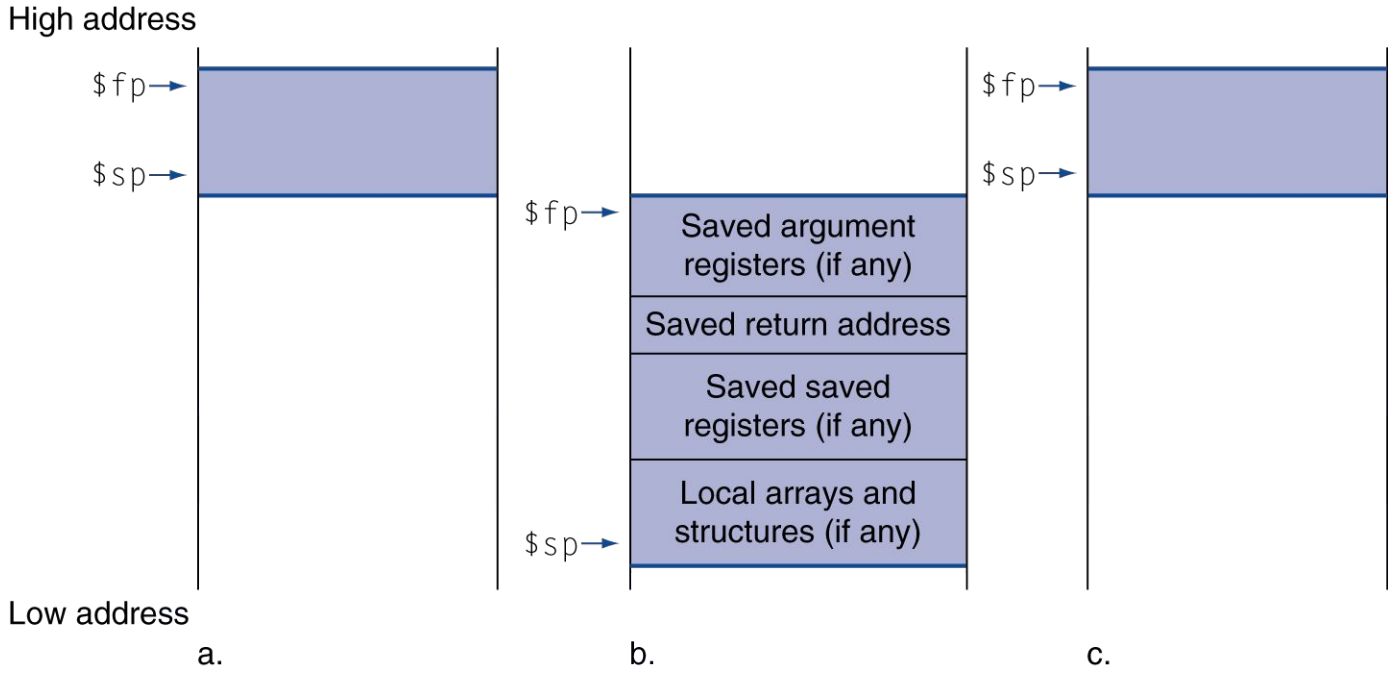
C Code:

```
int leaf_example (int g, h, i, j)
{ int f, x;
  f = (g + h) - (i + j);
  x= f+1;
  return x;
}
```

MIPS Code:

```
leaf_example:
        addi $sp, $sp, -4
        sw   $s0,   0($sp)
        addi $sp, $sp, -4
        sw   $s1,   0($sp)
        add   $t0, $a0, $a1
        add   $t1, $a2, $a3
        sub   $s0, $t0, $t1
        addi $s1, $s0, 1
        add   $v0, $s1, $zero
        lw   $s1,   0($sp)
        addi $sp, $sp, 4
        lw   $s0,   0($sp)
        addi $sp, $sp, 4
        jr     $ra
```

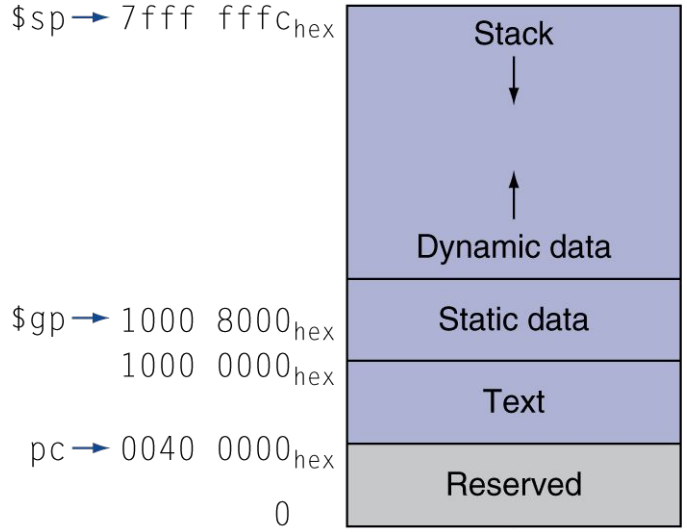
# Local Data on the Stack




- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

# Memory Layout

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



# Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

# Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

Copy Array y in Array x

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

- Arguments:

x (base address) → $a0, y (base address) → $a1, i → $s0

```
strcpy:
addi $sp, $sp, -4
sw $s0, 0($sp)
add $s0, $zero, $zero
Loop:
add $t0, $s0, $a1
lbu $t1, 0($t0)
add $t2, $s0, $a0
sb $t1, 0($t2)
beq $t1, $zero, L1
addi $s0, $s0, 1
j loop

L1:
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
```

As each data is 8 bit, no need to multiply the index with 4