Assembly Language and Computer Architecture Lab

Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

Week 9: Stack

- The stack data structure
- The purpose of a program stack, and how to implement it.
- How to implement reentrant programs.
- How to store local variables on the stack
- What recursive subprograms are, and how to implement them.

Reentrant Subprograms

- One method to call a subprogram is the **jal** operand, and then the **jr $ra** instruction was used as the equivalent of a return statement.

- To implement subprograms which do not call other subprograms, this definition of how to call and return from a subprogram was sufficient.

- However, this limitation on subprograms that they cannot be reentrant is far too restrictive to be of use for in real programs.

Stack data structure

- A stack is a Last-In-First-Out data structure.

- In a computer a stack is implemented as an array on which there are two operations, push and pop.

  - The **push** operation places an item on the top of the stack, and so places the item at the next available spot in the array and adds 1 to the array index to the next available spot.

  - A **pop** operation removes the top most item, so returns the item on top of stack, and subtracts 1 from the size of the stack.

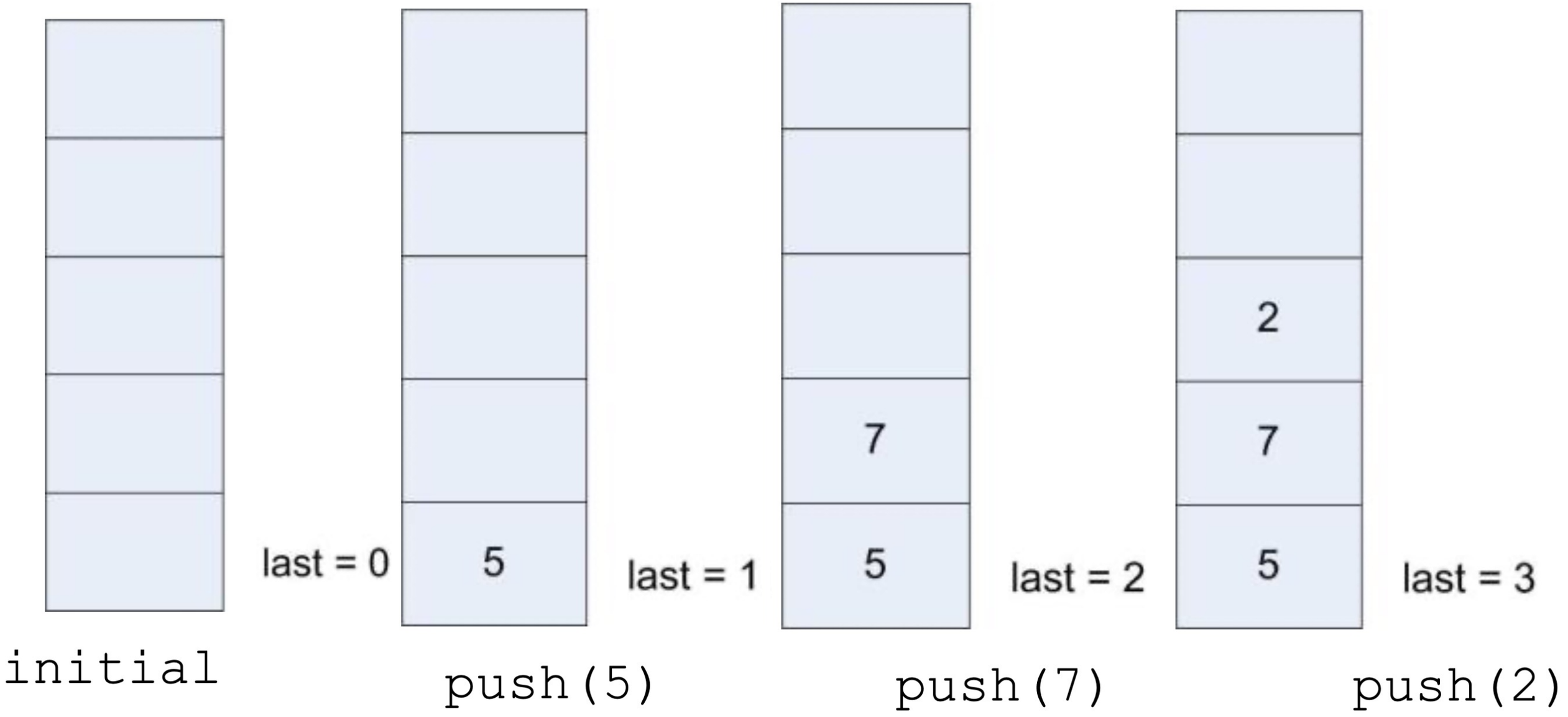
Stack data structure

- The example begins by creating a data structure for a stack.

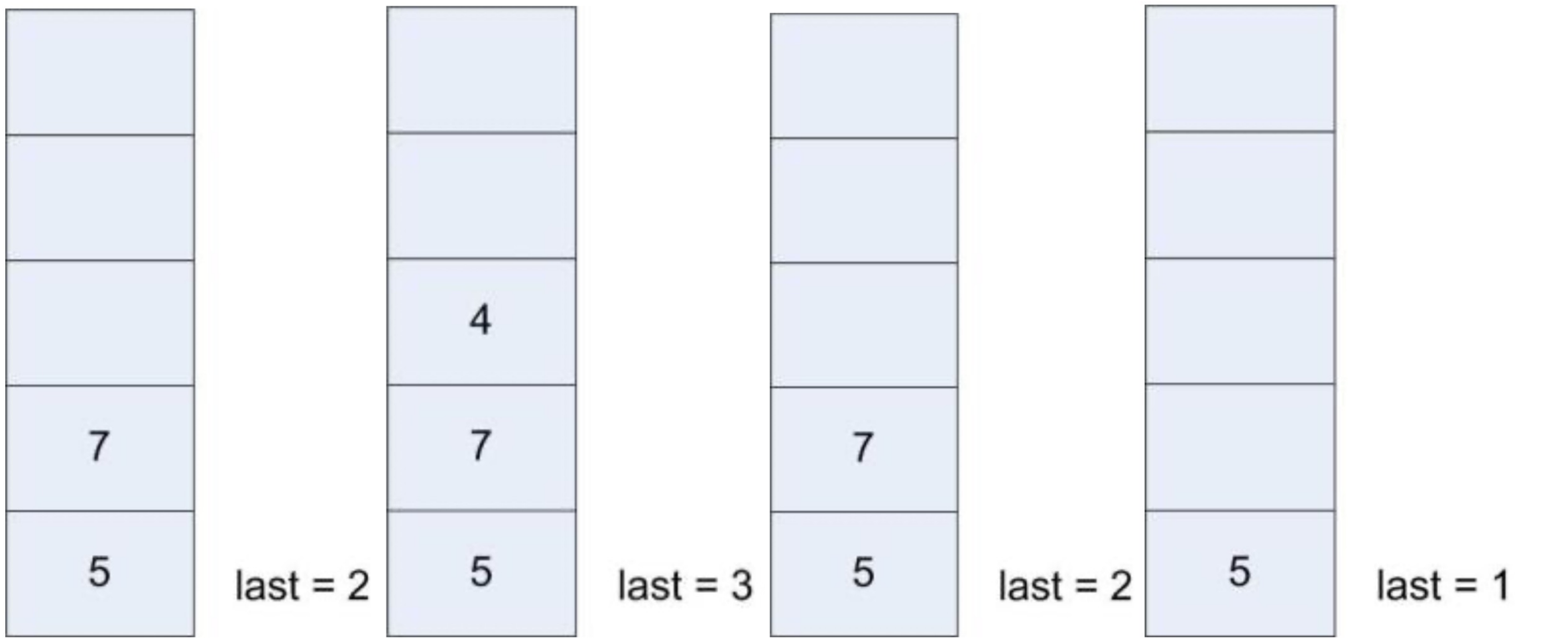
```
class Stack
{
        int SIZE=100;
        int[SIZE] elements;
        int last = 0;
        push(int newElement)
        {
            elements[last] = newElement;
            last = last + 1
        }
        int pop()
        {
            last = last - 1;
            return element[last];
        }
}
```

```
push(5)
push(7)
push(2)
print(pop())
push(4)
print(pop())
print(pop())
```

Stack data structure



Stack data structure



- Output: 2, 4, 7

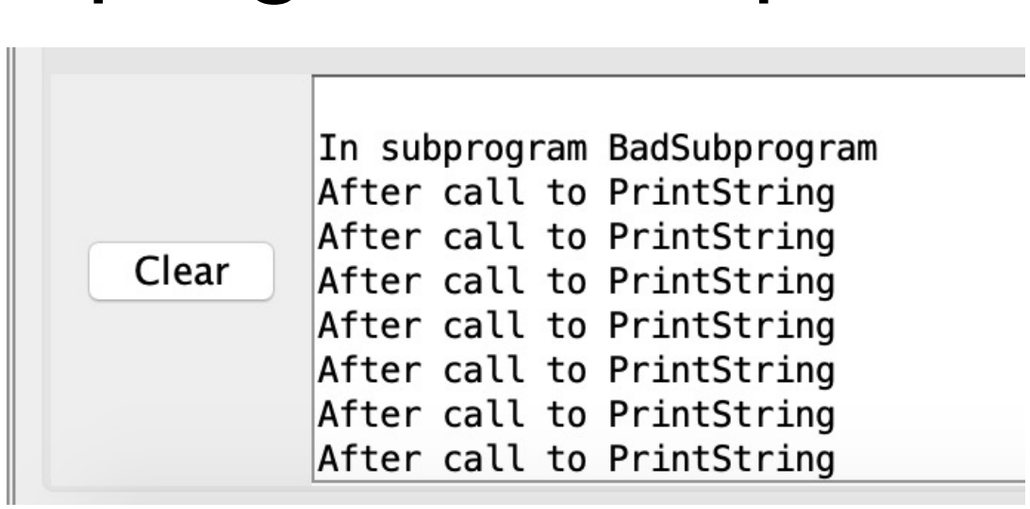
The non-reentrant subprogram problem

```
1   .text
2   .globl main
3   main:
4               jal BadSubprogram
5               la $a0, string3
6               jal PrintString
7   jal Exit
8
9   BadSubprogram:
10              la $a0, string1
11              jal PrintString
12              li $v0, 4
13              la $a0, string2
14              syscall
15              jr $ra
16  .data
17  string1: .asciiz "\nIn subprogram BadSubprogram\n"
18  string2: .asciiz "After call to PrintString\n"
19  string3: .asciiz "After call to BadSubprogram\n"
20  .include "utils.asm"
```

- The expected output is:

```
In subprogram BadSubprogram
After call to PrintString
After call to example
```

- The program's output is:



The non-reentrant subprogram problem

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x0c100005	jal 0x00400014	4: jal BadSubprogram
	0x00400004	0x3c011001	lui $1,0x00001001	5: la $a0, string3
	0x00400008	0x34240039	ori $4,$1,0x00000039	
	0x0040000c	0x0c10001e	jal 0x00400078	6: jal PrintString
	0x00400010	0x0c100021	jal 0x00400084	7: jal Exit
	0x00400014	0x3c011001	lui $1,0x00001001	9: la $a0, string1
	0x00400018	0x34240000	ori $4,$1,0x00000000	
✓	0x0040001c	0x0c10001e	jal 0x00400078	10: jal PrintString
✓	0x00400020	0x24020004	addiu $2,$0,0x00000004	11: li $v0, 4

Labels

Label	Address ▲
(global)	
main	0x00400000
mips8-3.asm	
BadSubprogram	0x00400014
PrintNewLine	0x00400034
PrintInt	0x00400048
PromptInt	0x00400060
PrintString	0x00400078

✓ Data ✓ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x206e490a	0x70627573	0x72676f72	0x42206d61	0x75536461	0x6f727062	0x6d617267	0x6641000a
0x10010020	0x20726574	0x6c6c6163	0x206f7420	0x6e697250	0x72745374	0x0a676e69	0x74664100	0x63207265
0x10010040	0x206c6c61	0x42206f74	0x75536461	0x6f727062	0x6d617267	0x000a000a	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ASCII

Mars Messages Run I/O

Clear

```
Assemble: assembling /Users/ngantt/Dropbox/Baigiang/IT3280/mips8-3.asm

Assemble: operation completed successfully.

Go: running mips8-3.asm

Go: execution paused at breakpoint: mips8-3.asm
```

Registers Coproc 1 Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400004
pc		0x0040001c
hi		0x00000000
lo		0x00000000

The non-reentrant subprogram problem

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040000c	0x0c10001e	jal 0x00400078	6: jal PrintString
	0x00400010	0x0c100021	jal 0x00400084	7: jal Exit
	0x00400014	0x3c011001	lui $1,0x00001001	9: la $a0, string1
	0x00400018	0x34240000	ori $4,$1,0x00000000	
✓	0x0040001c	0x0c10001e	jal 0x00400078	10: jal PrintString
✓	0x00400020	0x24020004	addiu $2,$0,0x00000004	11: li $v0, 4
	0x00400024	0x3c011001	lui $1,0x00001001	12: la $a0, string2
	0x00400028	0x3424001e	ori $4.$1.0x0000001e	

Labels

Label	Address ▲
(global)	
main	0x00400000
mips8-3.asm	
BadSubprogram	0x00400014
PrintNewLine	0x00400034
PrintInt	0x00400048
PromptInt	0x00400060

✓ Data ✓ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x206e490a	0x70627573	0x72676f72	0x42206d61	0x75536461	0x6f727062	0x6d617267	0x6641000a
0x10010020	0x20726574	0x6c6c6163	0x206f7420	0x6e697250	0x72745374	0x0a676e69	0x74664100	0x63207265
0x10010040	0x206c6c61	0x42206f74	0x75536461	0x6f727062	0x6d617267	0x000a000a	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ☐ ASCII

Registers Coproc 1 Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000004
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400020
pc		0x00400020
hi		0x00000000
lo		0x00000000

Mars Messages Run I/O

```
Assemble: assembling /Users/ngantt/Dropbox/Baig
Assemble: operation completed successfully.
Go: running mips8-3.asm
Go: execution paused at breakpoint: mips8-3.asm
Go: running mips8-3.asm
Go: execution paused at breakpoint: mips8-3.as
```

Clear

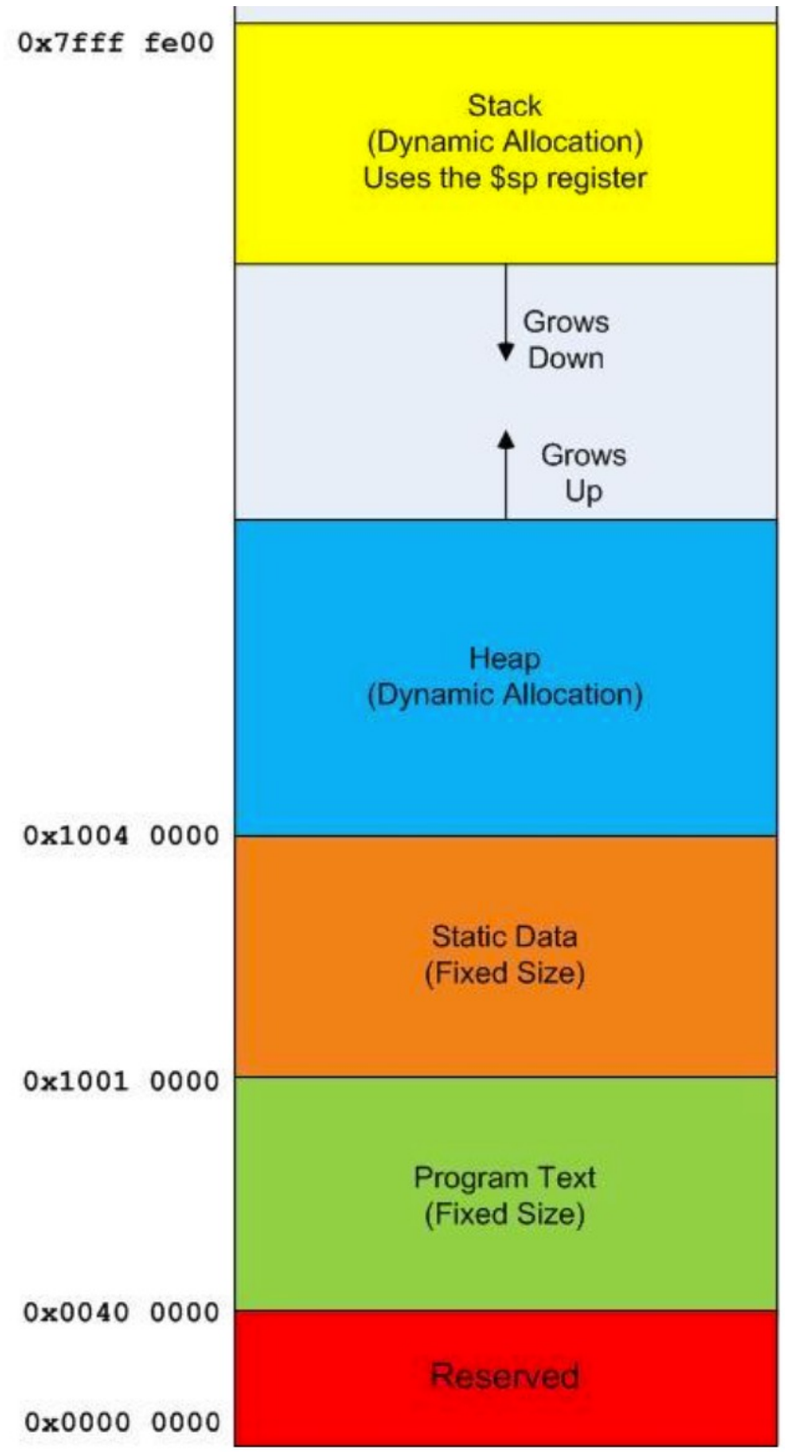
The $ra was overwritten → the link back to the main program was lost.

The $ra needs to be stored when the subprogram is entered and restored just before the program leaves.

Making subprograms re-entrant

- The problem with the **$ra** is also a problem with any registers that the program uses, as well as any variables that are defined in the subprogram.

- Space is needed in memory to store these variables.

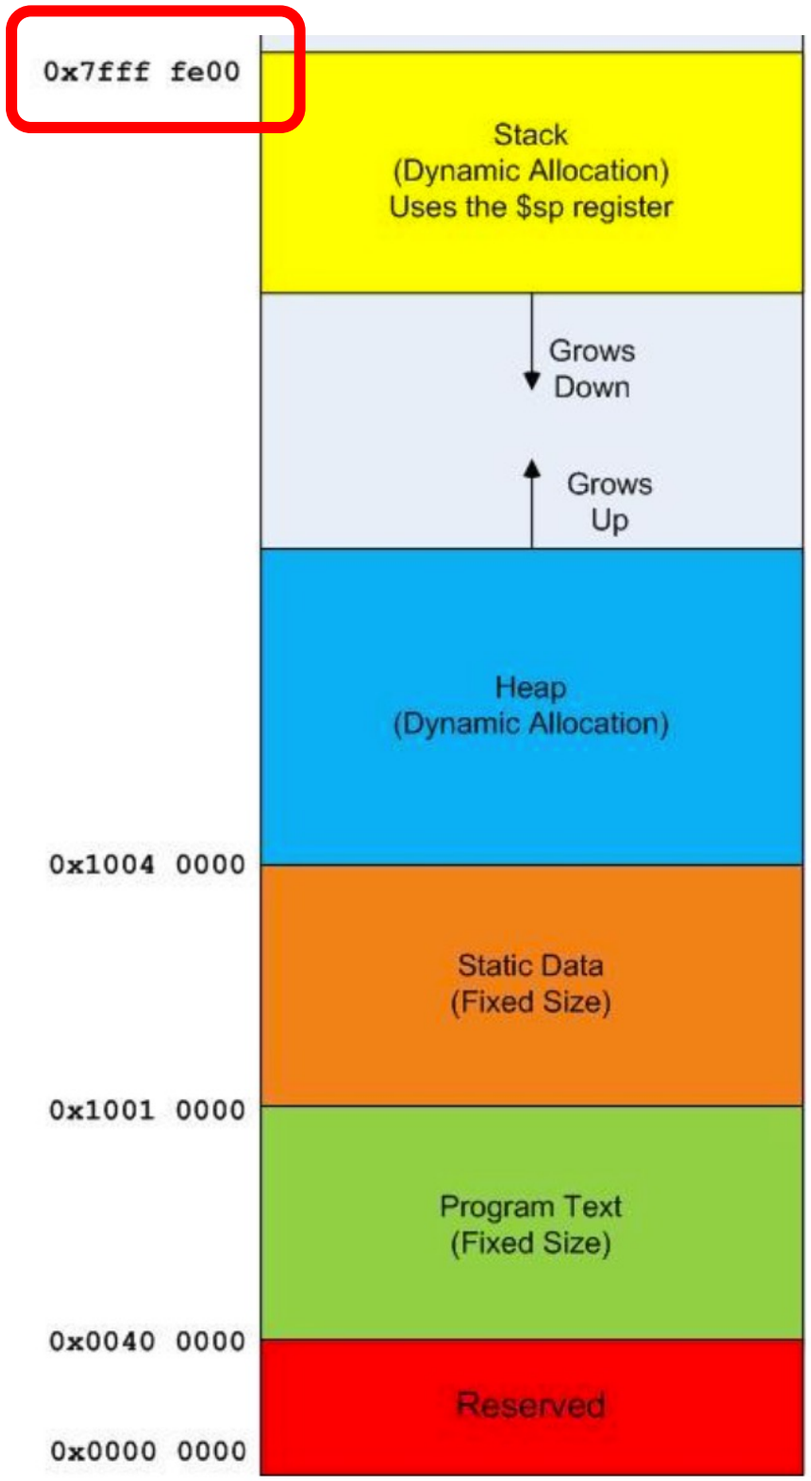


Making subprograms re-entrant

- The space to store variables and registers which need to be saved for a subprogram is called a stack.

- When the program begins to run, memory at a high address, is allocated to store the stack.

- The stack then grows downward in memory.



Making subprograms re-entrant

- When a subprogram is entered, it pushes (or allocates) space on the stack for any registers it needs to save, and any local variables it might need to store.

- When the subprogram is exited, it pops this memory off of the stack, freeing any memory that it might have allocated, and restoring the stack to the state it was in before the subprogram was called.

Making subprograms re-entrant

```
1   .text
2   .globl main
3   main:
4           jal GoodSubprogram
5           la $a0, string3
6           jal PrintString
7           jal Exit
8
9   GoodSubprogram:
10          addi $sp, $sp, -4       # save space on the stack (push) for the $ra
11          sw $ra, 0($sp)          # save $ra
12          la $a0, string1
13          jal PrintString
14
15          li $v0, 4
16          la $a0, string2
17          syscall
18
19          lw $ra, 0($sp)          # restore $ra
20          addi $sp, $sp, 4        # return the space on the stack (pop)
21          jr $ra
22  .data
23  string1: .asciiz "\nIn subprogram GoodExample\n"
24  string2: .asciiz "After call to PrintString\n"
25  string3: .asciiz "After call to GoodExample\n"
26  .include "utils.asm"
```

This example shows how the **$ra** register is stored while the subprogram is running and then be restored just before it is used to return from the subprogram.

Making subprograms re-entrant

- Consider the following piece of code:

```
addi $sp, $sp, -4      # save space on the stack (push) for the $ra
sw $ra, 0($sp)         # save $ra
```

- When the subprogram is entered, the stack pointer (**$sp**) register points to the current end of the stack.

$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400004

Making subprograms re-entrant

- Consider the following piece of code:

```
addi $sp, $sp, -4      # save space on the stack (push) for the $ra
sw $ra, 0($sp)         # save $ra
```

- All subprograms before have incremented the **$sp** to allocate space for their automatic variables, so all previous subprograms have stack frames on the stack for their execution.

- All space above the stack pointer is taken, but the space below the **$sp** is open, and this is where this subprogram allocates its space to place it variables.

Making subprograms re-entrant

- Consider the following piece of code:

```
addi $sp, $sp, -4      # save space on the stack (push) for the $ra
sw $ra, 0($sp)         # save $ra
```

- The stack grows downward, which is why 4 is subtracted from **$sp** when the space is allocated.

$sp	29	0x7fffeff8
$fp	30	0x00000000
$ra	31	0x00400004

- The allocation of 4 bytes is the amount needed to store the **$ra**.

Making subprograms re-entrant

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400004	0x3c011001	lui $1,0x00001001	5: la $a0, string3
	0x00400008	0x34240037	ori $4,$1,0x00000037	
	0x0040000c	0x0c100022	jal 0x00400088	6: jal PrintString
	0x00400010	0x0c100025	jal 0x00400094	7: jal Exit
	0x00400014	0x23bdfffc	addi $29,$29,0xfffffffc	10: addi $sp, $sp, -4 # save spac...
	0x00400018	0xafbf0000	sw $31,0x00000000($29)	11: sw $ra, 0($sp) # save $ra
✓	0x0040001c	0x3c011001	lui $1,0x00001001	12: la $a0, string1

Labels

Label	Address ▲
(global)	
main	0x00400000
mips8-4.asm	
GoodSubprogram	0x00400014
PrintNewLine	0x00400044
PrintInt	0x00400058
PromptInt	0x00400070

✓ Data ✓ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00400004	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

current $sp ✓ Hexadecimal Addresses ✓ Hexadecimal Values ☐ ASCII

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff8
$fp	30	0x00000000
$ra	31	0x00400004
pc		0x0040001c
hi		0x00000000
lo		0x00000000

Mars Messages | Run I/O

Clear

Select box for the memory to view is "**current $sp**", or stack. Looking at the value of **$sp** of 0x7fefff8, it can be seen that the correct return address from GoodSubprogram has been saved.

Making subprograms re-entrant

```
1   .text
2   .globl main
3   main:
4           jal GoodSubprogram
5           la $a0, string3
6           jal PrintString
7           jal Exit
8
9   GoodSubprogram:
10          addi $sp, $sp, -4       # save space on the stack (push) for the $ra
11          sw $ra, 0($sp)          # save $ra
12          la $a0, string1
13          jal PrintString
14
15          li $v0, 4
16          la $a0, string2
17          syscall
18
19          lw $ra, 0($sp)          # restore $ra
20          addi $sp, $sp, 4        # return the space on the stack (pop)
21          jr $ra
22  .data
23  string1: .asciiz "\nIn subprogram GoodExample\n"
24  string2: .asciiz "After call to PrintString\n"
25  string3: .asciiz "After call to GoodExample\n"
26  .include "utils.asm"
```

This example shows how the **$ra** register is stored while the subprogram is running and then be restored just before the it is used to return from the subprogram.

Making subprograms re-entrant

- The second piece of code, shown below, is be placed just before the last statement in a subprogram.

```
lw $ra, 0($sp)      # restore $ra
addi $sp, $sp, 4    # return the space on the stack (pop)
```

- The **$ra** is restored to the value that it contained when the subprogram was called, so the subprogram can return to the correct line in the calling program.

Making subprograms re-entrant

- The second piece of code:

```
lw $ra, 0($sp)      # restore $ra
addi $sp, $sp, 4    # return the space on the stack (pop)
```

- The stack frame for this subprogram is then ***popped*** by adding the amount of memory used back to the stack.

- In this case, 4 bytes are added to the **$sp**

$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400004

Recursion

- Recursion is a mechanism to a way to divide a problem up into successively smaller instances of the same problem until some stopping condition (or base case) is reached.

- The individual solutions to all of the smaller problems are then gathered together and an overall solution to the problem is obtained.

Exercise 9.1 - Print an integer array

- Print an integer array pseudo code:

```
Subprogram PrintIntArray(array, size)
{
    print("[")
    for (int i = 0; i < size; i++)
    {
        print("," + array[i])
    }
    print("]")
}
```

Exercise 9.1 - Print an integer array

- Please input the following program, assemble, run, observe and explain the change in $sp:

```
1  .text
2  .globl main
3  main:
4      la $a0, array_base
5      lw $a1, array_size
6      jal PrintIntArray
7      jal Exit
8
9  .data
10     array_size: .word 5
11     array_base:
12                 .word 12
13                 .word 7
14                 .word 3
15                 .word 5
16                 .word 11
17
18 .text
19 # Subprogram: PrintIntArray
20 # Purpose: print an array of ints
21 # inputs: $a0 - the base address of the array
22 # $a1 - the size of the array
23 #
24 PrintIntArray:
25     addi $sp, $sp, -16  # Stack record
26     sw $ra, 0($sp)
27     sw $s0, 4($sp)
28     sw $s1, 8($sp)
29     sw $s2, 12($sp)
30
31     move $s0, $a0               # save the base of the array to $s0
32
33
34     # initialization for counter loop
35     # $s1 is the ending index of the loop
36     # $s2 is the loop counter
37     move $s1, $a1
38     move $s2, $zero
39
40     la $a0 open_bracket         # print open bracket
41     jal PrintString
42 loop:
43     # check ending condition
44     sge $t0, $s2, $s1
45     bnez $t0, end_loop
46
47         sll $t0, $s2, 2         # Multiply the loop counter by
48                                 # by 4 to get offset (each element
49                                 # is 4 big)
50         add $t0, $t0, $s0       # address of next array element
51         lw $a1, 0($t0)          # Next array element
52         la $a0, comma
53         jal PrintInt            # print the integer from array
54
55         addi $s2, $s2, 1        #increment $s0
56         b loop
57 end_loop:
58     li $v0, 4
59     la $a0, close_bracket       # print close bracket
60     syscall
61
62     lw $ra, 0($sp)
63     lw $s0, 4($sp)
64     lw $s1, 8($sp)
65     lw $s2, 12($sp)             # restore stack and return
66     addi $sp, $sp, 16
67     jr $ra
68
69 .data
70     open_bracket: .asciiz "["
71     close_bracket: .asciiz "]"
72     comma: .asciiz ","
73 .include "utils.asm"
```

Exercise 9.2 - Recursive multiply

- Multiplication can be defined as adding the multiplier (m) to itself the number of times in the multiplicand (n) times.
- Thus a recursive definition of multiplication is the following:

$$M(m,n) = m \text{ (when } n = 1)$$

$$\text{else} = m + M(m,n-1)$$

Exercise 9.2 - Recursive multiply

- Implemented pseudo code is shown below:

```
subprogram global main()
{
    register int multiplicand
    register int multiplier
    register int answer
    m = prompt("Enter the multiplicand")
    n = prompt("Enter the multiplier")
    answer = Multiply(m, n)
    print("The answer is: " + answer)
}
subprogram int multiply(int m, int n)
{
    if (n == 1)
        return m;
    return m + multiply(m,n-1)
}
```

Exercise 9.2 - Recursive multiply

- Please input the following program, assemble, run, observe and explain the change in registers:

```
1   .text
2   .globl main
3   main:
4       # register conventions
5       # $s0 - m
6       # $s1 - n
7       # $s2 - answer
8       la $a0, prompt1     # Get the multiplicand
9       jal PromptInt
10      move $s0, $v0
11
12      la $a0, prompt2     # Get the multiplier
13      jal PromptInt
14      move $s1, $v0
15
16      move $a0, $s0
17      move $a1, $s1
18
19      jal Multiply        # Do multiplication
20      move $s2, $v0
21
22      la $a0, result      #Print the answer
23      move $a1, $s2
24      jal PrintInt
25
26      jal Exit
27
```

```
28  Multiply:
29      addi $sp, $sp -8    # push the stack
30      sw $a0, 4($sp)      # save $a0
31      sw $ra, 0($sp)      # save the $ra
32
33      seq $t0, $a1, $zero # if (n == 0) return
34      addi, $v0, $zero, 0 # set return value
35      bnez $t0, Return
36
37      addi $a1, $a1, -1   # set n = n-1
38      jal Multiply        # recurse
39      lw $a0, 4($sp)      # retrieve m
40      add $v0, $a0, $v0   # return m +
41                          #  multiply(m,n-1)
42      Return:
43      lw $ra, 0($sp)      #pop the stack
44      addi $sp, $sp, 8
45      jr $ra
46
47  .data
48  prompt1: .asciiz "Enter the multiplicand: "
49  prompt2: .asciiz "Enter the multiplier: "
50  result: .ascii "The answer is: "
51  .include "utils.asm"
```

Exercise 9.3

- The following program uses **abs** subprogram to find the absolute value in an integer.

- This program uses two registers, **$a0** for input argument and **$v0** for result.

- Please read this example carefully, pay attention to how to write and invoke a subprogram.

Exercise 9.3

```
.text
main:   li $a0,-45                  #load input parameter
        jal abs                     #jump and link to abs subprogram
        nop
        add $s0, $zero, $v0
        jal Exit                    #terminate
endmain:
#---------------------------------------------------------------
# Subprogram:   abs
# input:        $a1 the interger need to be gained the absolute value
# output:       $v0 absolute value
#---------------------------------------------------------------
abs:    sub $v0,$zero,$a1           #put -(a0) in v0; in case (a0)<0
                                    #if (a0)<0 then done
        bltz $a1,done               #else put (a0) in v0
        nop
done:   add $v0,$a1,$zero
        jr $ra

.include "utils.asm"
```

Exercise 9.3

- Create a new project to implement the above program.
- Compile and upload to simulator.
- Change input parameters and observe the memory when run the program step by step.
- Pay attention to register $pc, $ra to clarify invoking subprogram process.

Exercise 9.4

- The following program has the function **max** to find the largest elements in three of integers.

- These intergers are passed to **max** subprogram through **$a0**, **$a1** and **$a2** registers.

- Read this example carefully and try to explain each line of code.

Exercise 9.4

```
.text
main:   li $a0,2                    #load test input
        li $a1,6
        li $a2,9
        jal max                     #call max subprogram
        nop
endmain:
#-------------------------------------------------------
#Subprogram:    max
#Purpose:       find the largest of three integers
#input:         $a0
#               $a1
#               $a2
#output:        $v0 the largest value
#-------------------------------------------------------
max:    add $v0,$a0,$zero           #copy (a0) in v0; largest so far
        sub $t0,$a1,$v0             #compute (a1)-(v0)
        bltz $t0,okay               #if (a1)-(v0)<0 then no change
        nop
        add $v0,$a1,$zero           #else (a1) is largest thus far
okay:   sub $t0,$a2,$v0             #compute (a2)-(v0)
        bltz $t0,done               #if (a2)-(v0)<0 then no change
        nop
        add $v0,$a2,$zero           #else (a2) is largest overall
done:   jr $ra                      #return to calling program
```

Exercise 9.4

- Create a new project to implement the above program.
- Compile and upload to simulator.
- Change input parameters (register a0, a1, a2) and observe the memory when run the program step by step.
- Pay attention to register $pc, $ra to clarify invoking subprogram process.

Exercise 9.5

- The following program demonstrates the push and pop operations with stack.
- The value of two register **$s0** and **$s1** will be swapped using stack.

Exercise 9.5

- Read this example carefully, pay attention to adjust operation and the order of push and pop (**sw** and **lw** instructions).

```
.text
push:   addi $sp,$sp,-8
        sw $s0,4($sp)
        sw $s1,0($sp)
work:   nop
        nop
        nop
pop:    lw $s0,0($sp)
        lw $s1,4($sp)
        addi $sp,$sp,8
```

Exercise 9.5

- Create a new project to implement the above program.
- Compile and upload to simulator.
- Pass test value to registers **$s0** and **$s1**, observe run process, pay attention to stack pointer.
- Goto memory space that pointed by **$sp** register to view **push** and **pop** operations in detail.

Parameters

In this section, the following unresolved questions relating to subprogram are raised:

- How to pass more than four input parameters to a subprogram or receive more than two results from it?

- Where does a subprogram save its own parameters and immediate results when calling another subprogram (nested calls)?

Parameters

- The stack is used in both cases.
- Before a subprogram call, the calling program pushed the contents of any register that need to be saved onto the top of the stack and follow these with any additional arguments for the subprogram.
- The subprogram can access these arguments in the stack.

Parameters

- After the subprogram terminates, the calling subprogram expects to find the stack pointer undisturbed, thus allowing it to restore the save registers to their original states and proceed with its own computations.

- Thus, a procedure that uses the stack by modifying the stack pointer must save the content of the stack pointer at the outset and at the end, restore $sp to its original state.

- This is done by copying the stack pointer into the frame pointer register **$fp**.

Parameters

- Before doing this, however, the old contents of the frame pointer must be saved.

- Hence, while a subprogram is executing, **$fp** may hold the content of **$sp** right before it was called.

- The **$fp** and **$sp** registers together “frame” the area of the stack that is in use by the current subprogram.

The stack frames

- The subprogram infrastructure given here will have a limitation.

- Only 4 input parameters to the subprogram, and 2 return values from the subprogram, will be allowed.

- This limitation is used to simplify the concept of a stack.

- In many cases in MIPS assembly subprograms are implemented using a Frame Pointer (**$fp**) to keep track of parameters and return values.

The stack frames

- Every time a subroutine is called, a unique stack frame is created for that instance of the subroutine call.

- In the case of a recursive subroutine call, multiple stack frames are created, one for each instance of the call.

- The organization of the stack frame is important for two reasons.

The stack frames

- One, it forms a contract between the caller and the callee that defines:

  - how arguments are passed

  - how the result of a function is passed

  - how registers are shared

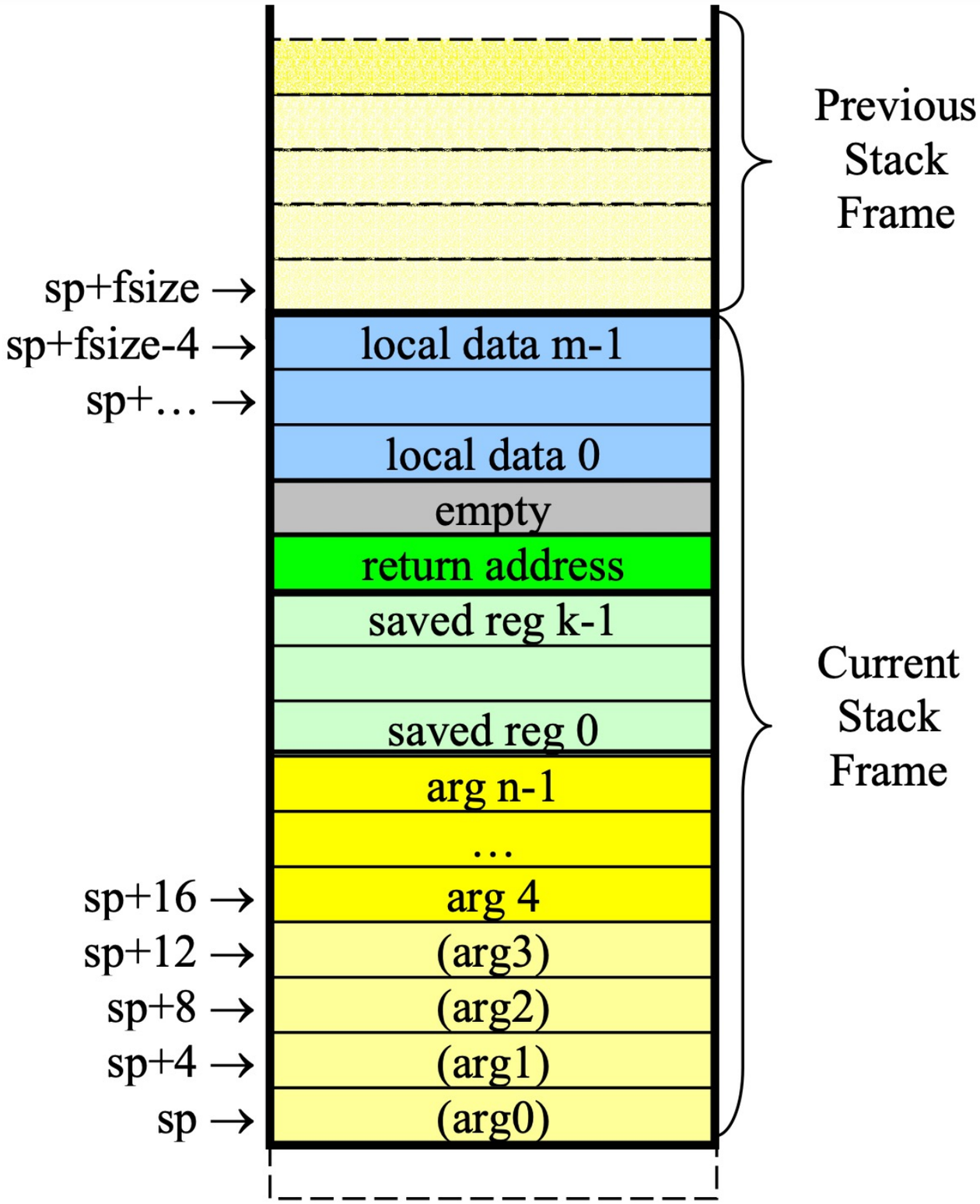
- Second, it defines the organization of storage local to the callee within its stack frame.

The stack frames

In general, the stack frame for a subroutine may contain space to contain the following:

- Space to store arguments passed to subroutines that this subroutine calls.
- A place to store the values of saved registers (**$s0** to **$s7**).
- A place to store the subroutine's return address (**$ra**).
- A place for local data storage.

The stack frames organization



Note that in this figure the stack is growing in a *downward* direction. I.e., the *top* of the stack is at the *bottom* of the picture.

The stack frames organization

The stack frame is divided into five regions:

- The Argument Section
- The Saved Registers Section
- The Return Address Section
- The Pad
- The Local Data Storage Section

The Argument Section

- Contains the space to store the arguments that are passed to any subroutine that are called by the current subroutine (i.e., the subroutine whose stack frame currently on top of the stack.)

- The first four words of this section are never used by the current subroutine; they are reserved for use by any subroutine called by the current subroutine.

- If there are more than four arguments, the current subroutine stores them on the stack, at addresses sp+16, sp+20, sp+24, etc.

The Saved Registers Section

- Contains space to save the values of any of the saved registers (**$s0** to **$s7**) that the current subroutine wants to use.
- On entry into the current subroutine, the subroutine copies the values of any of the saved registers, **$s0** to **$s7**, whose values it might change during the course of the execution of the subroutine into this section of the stack frame.

The Saved Registers Section

- Just before the subroutine returns, it copies these values from the Saved Registers Section back into the original saved registers.

- In between, the current subroutine is free to change the value of the saved registers at will.

- However, the caller of the current subroutine will see the same values in these registers after the subroutine call as it saw before the subroutine call.

The Return Address Section

- Be used to store the value of the return address register, **$ra**.
- This value is copied onto the stack at the start of execution of the current subroutine and copied back into the **$ra** register just before the current subroutine returns.

The Pad

- Be inserted into the stack frame to make sure that total size of the stack frame is always a multiple of 8.
- It is inserted here to ensure that the local data storage area starts on a double word boundary.

The Local Data Storage Section

- Be used for local variable storage.
- The current subroutine must reserve enough words of storage in this area for all of its local data, including space to store the value of any temporary registers ($t0 to $t9) that it needs to preserve across subroutine calls.
- The local data storage area must also be padded so that its size is always a multiple of 8 words.

The stack frames: additional rules

- The value of the stack pointer is required to be multiple of 8 at all times.
- This ensures that a 64-bit data object can be pushed on the stack without generating an address alignment error at run-time.
- This implies the size of every stack frame must be a multiple of 8;

The stack frames: additional rules

- The values of the argument registers **$a0-$a3** are not required to be preserved across subroutine calls.
- Thus, a subroutine is allowed to change the values of any of the argument registers without saving/restoring them.

The stack frames: additional rules

- The first four words of the Argument Section of a stack frame are known as *argument slots* -- memory locations reserved to store the four arguments **$a0**-**$a3**.

- It is important to remember that a subroutine does *not* store anything in the first four argument slots, the actual arguments are passed in **$a0** to **$a3**.

- However, the called subroutine may choose to copy the values of **$a0** to **$a3** into the argument slots if it wants to save these values.

- If it does so, then it can then treat all its arguments as a 1-dimensional array in memory.

The stack frames: additional rules

- The argument slots are allocated by the caller but are used by the callee!

- All four slots are *required*, even if the caller knows it is passing fewer than four arguments. Thus, on entry a subroutine may legally store *all* of the argument registers into the argument slots if desired.

- The caller must allocate space on its stack frame for the maximum number of arguments for *any* subroutine that it calls (or the minimum of four arguments, if all the subroutines that it calls have fewer than four arguments.)

Leaf vs Nonleaf Subroutines

- Not every subroutine will need every section described above in its stack frame.

- The general rule is that if the subroutine does not need a section, then it may *omit* that section from its call frame. To help make this clear, 3 different classes of subroutines are introduced:

  - *Simple Leaf*

  - *Leaf with Data*

  - *Nonleaf*

Simple Leaf

- *Simple Leaf* subroutines:
  - Do not call any other subroutines,
  - Do not use any memory space on the stack (either for local variables or to save the values of saved registers).
- Such subroutines do not require a stack frame, consequently never need to change **$sp**.

A Simple Leaf Function

- Consider the simple function

```
int g( int x, int y ) {
   return (x + y);
}
```

- This function does not call any other function, so it does not need to save **$ra**.

- Also, it does not require any temporary storage. Thus, it can be written with no stack manipulation at all.

```
g:      add   $v0,$a0,$a1 # result is sum of args
        jr    $ra # return
```

Leaf with Data

- Leaf subroutines (i.e. do not call any other subroutines) that require stack space, either for local variables or to save the values of saved registers.

- Such subroutines push a stack frame (the size of which should be a multiple of 8 as discussed above).

- However, **$ra** is not saved in the stack frame.

A leaf function with data

- Now let's make **g** a little more complicated:

```
int g( int x, int y ) {
  int a[32];
  ... (calculate using x, y, a);
  return a[0];
}
```

- This function does not call any other function, so it does not need to save **$ra**, but, it does require space for the array **a**.

A leaf function with data

- Thus, it must push a stack frame.

```
g:      # start of prologue
        addiu $sp,$sp,(-128) # push stack frame
        # end of prologue

        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 0($sp) to 124($sp)

        lw    $v0,0($sp)     # result is a[0]

        # start of epilogue
        addiu $sp,$sp,128    # pop stack frame
        # end of epilogue

        jr    $ra            # return
```

A leaf function with data

- Because this is a leaf function, there is no need to save/restore **$ra** and no need to leave space for argument slots.
- This stack frame only needs space for local data storage. Its size is 32 words or 128 bytes.

A leaf function with data

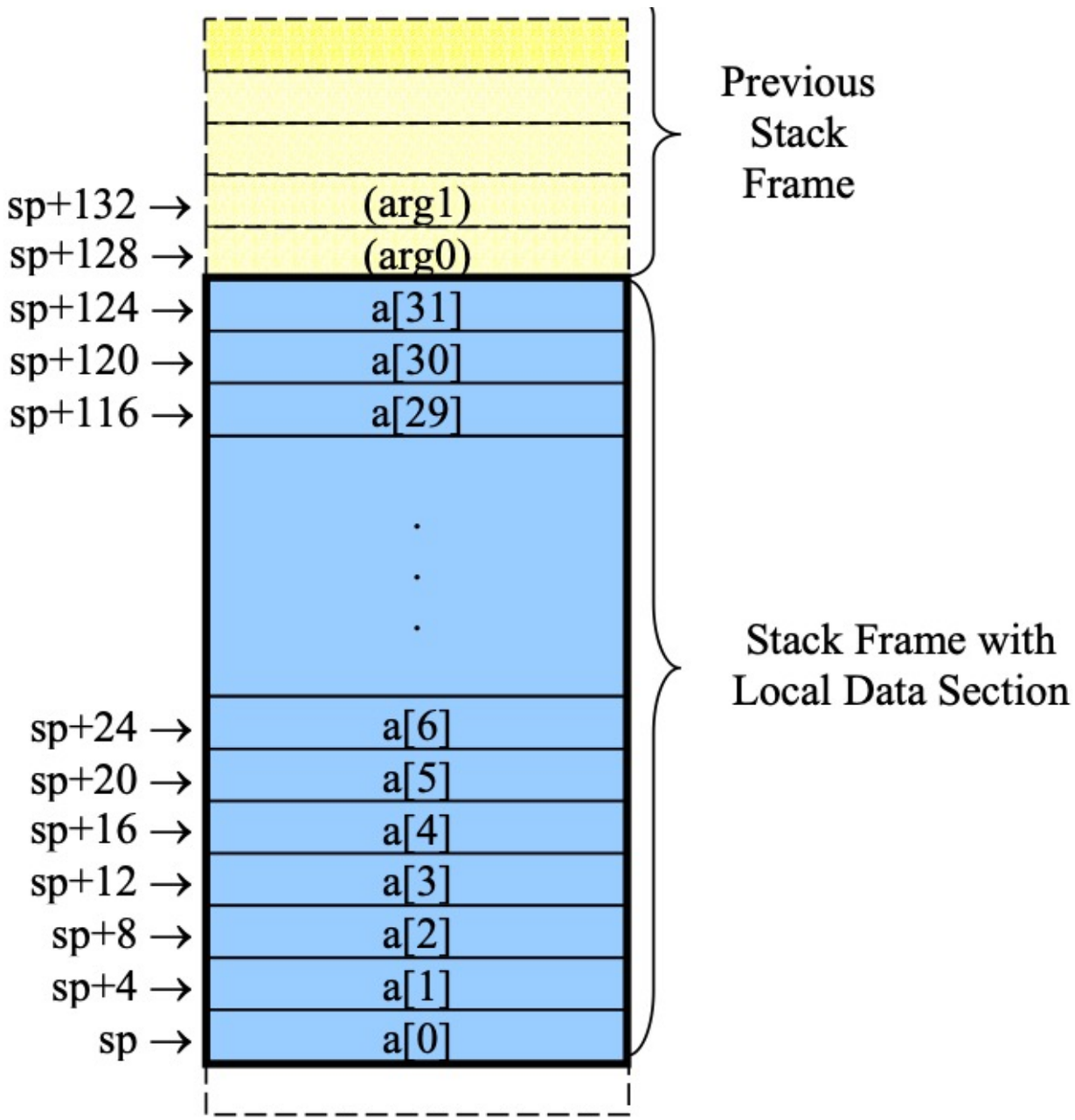
- Also, we always require that the local data storage block be double word aligned, but in this case since the value of **$sp** is always double word aligned, no padding is necessary.

- The array a is stored at addresses 0(**$sp**) to 124(**$sp**).

- Note that, in this example, the code for the calculations (...) is *not* allowed to change the values of any of the registers **$s0** to **$s7**.

A leaf function with data

- The stack frame for this example follows.



Nonleaf

- Subroutines are those that call other subroutines.
- The stack frame of a nonleaf subroutine will probably have most if not all the sections.

A Leaf Function With Data and Saved Registers

- Suppose that the code for the calculations (...) *does* change the value of some of the saved registers, **$s0**, **$s1**, and **$s3**.
- We would have to change code for **g** as follows.

A Leaf Function With Data and Saved Registers

```
g:      # start of prologue
        addiu $sp,$sp,(-144) # push stack frame

        sw $s0,0($sp)        # save value of $s0
        sw $s1,4($sp)        # save value of $s1
        sw $s3,8($sp)        # save value of $s3
        # end of prologue

        # start of body
        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 16($sp) to 140($sp)

        lw $v0,16($sp)       # result is a[0]
        # end of body

        # start of epilogue
        lw $s0,0($sp)        # restore value of $s0
        lw $s1,4($sp)        # restore value of $s1
        lw $s3,8($sp)        # restore value of $s3
        addiu $sp,$sp,144    # pop stack frame
        # end of epilogue

        jr    $ra            # return
```

A Leaf Function With Data and Saved Registers

- In this case, we had to add a Saved Register Section and Pad to the stack frame.

- The Saved Register Section consists of three words, at addresses 0(sp), 4(sp), and 8(sp), that are used to save the value of **$s0**, **$s1**, and **$s3**.

- The Pad, at address 12(sp) is used to pad the stack frame so that its size is a multiple of 8 and start to the Local Data Section at an address that is also a multiple of 8.

A Leaf Function With Data and Saved Registers

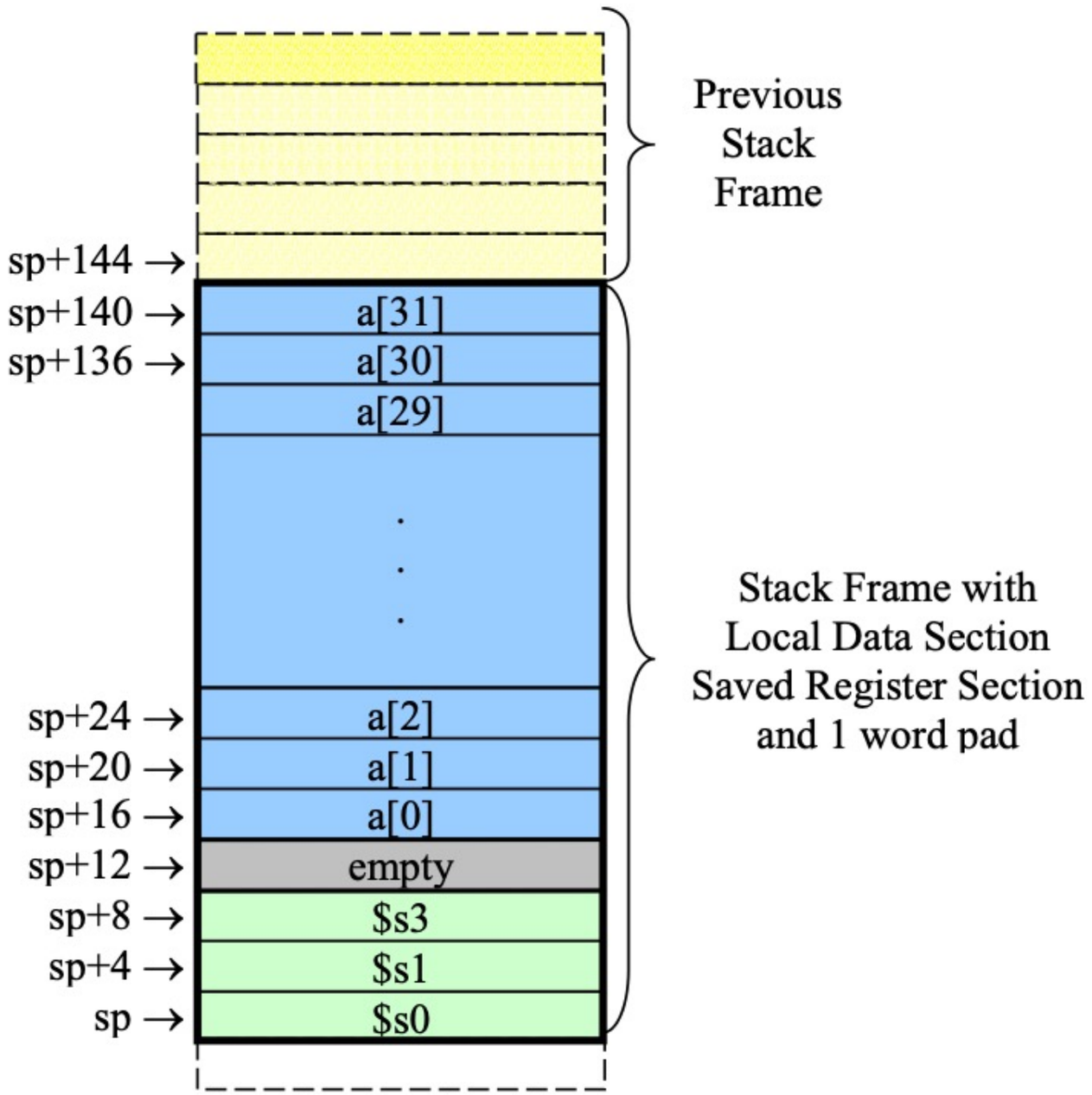
- The total size of this stack frame is 36 (3+1+32) words or 144 (12+4+128) bytes.

- Instructions to save the saved registers have been added to the prologue and instructions to restore the saved registers have been added to the epilogue.

- Note that the addresses used to store the array a increased by 16 as a result of adding this new section, so the array is now stored at addresses 16($sp) to 140($sp).

A Leaf Function With Data and Saved Registers

- The stack frame for this example follows.



Assigment 9.4

- The following program is a recursive procedure for computing n!

- Read this program carefully and pay attention to register **$sp** and **$fp** at the start and the end of each recursive step.

Excercise 9.6

```
.text
main:   jal     Warp

print:  add     $a1, $v0, $zero         # $a0 = result from N!
        li      $v0, 56
        la      $a0, Message
        syscall
        jal Exit                        #terminate
endmain:

#-----------------------------------------------------------------
#Procedure Warp: assign value and call Fact
#-----------------------------------------------------------------
Warp:   sw      $fp,-4($sp)     #save frame pointer (1)
        addi    $fp,$sp,0       #new frame pointer point to the top (2)
        addi    $sp,$sp,-8      #adjust stack pointer (3)
        sw      $ra,0($sp)      #save return address (4)

        li      $a0,6           #load test input N
        jal     Fact            #call fact procedure
        nop

        lw      $ra,0($sp)      #restore return address (5)
        addi    $sp,$fp,0       #return stack pointer (6)
        lw      $fp,-4($sp)     #return frame pointer (7)
        jr      $ra
wrap_end:
```

Excercise 9.6

```
#-----------------------------------------
#Subprogram:     Fact
#Purpose:        compute N!
#input:          $a0: integer N
#output:         $v0: the largest value
#-----------------------------------------
Fact:   sw      $fp,-4($sp)             #save frame pointer
        addi    $fp,$sp,0               #new frame pointer point to stack's top
        addi    $sp,$sp,-12             #allocate space for $fp,$ra,$a0 in stack
        sw      $ra,4($sp)              #save return address
        sw      $a0,0($sp)              #save $a0 register
        slti    $t0,$a0,2               #if input argument N < 2
        beq     $t0,$zero,recursive     #if it is false ((a0 = N) >=2)
        nop
        li      $v0,1                   #return the result N!=1
        j       done
        nop
recursive:
        addi    $a0,$a0,-1              #adjust input argument
        jal     Fact                    #recursive call
        nop
        lw      $v1,0($sp)              #load a0
        mult    $v1,$v0                 #compute the result
        mflo    $v0
done:   lw      $ra,4($sp)              #restore return address
        lw      $a0,0($sp)              #restore a0
        addi    $sp,$fp,0               #restore stack pointer
        lw      $fp,-4($sp)             #restore frame pointer
        jr      $ra                     #jump to calling
```

```
.data
Message: .asciiz "Result of recursive computing: "

.include "utils.asm"
```

Excercise 9.6

- Create a new project to implement the program.
- Compile and upload to simulator.
- Pass test input through register **$a0**, run this program and test result in register **$v0**.
- Run this program in step by step mode, observe the changing of register **$pc**, **$ra**, **$sp** and **$fp**.
- Draw the stack through this recursive program in case of n=3 (compute 3!).

Assigment 9.1

- Implement a subprogram which takes 4 numbers in the argument registers $a0...$a3, and returns the largest value and the average in $v0 and $v1 to the calling program. The program must be structured as follows:

```
Subprogram largestAndAverage($a1, $a2, $a3, $a4)
{
    int var0 = $a0, var1 = $a1, var2 = $a2, var3 = $a3;
    $s0 = getLarger($a1, $a2);
    $s0 = getLarger($s0, $a3);
    $v0 = getLarger(s0, $a4); // Largest is in $v0

    $v1 = (var0 + var1 + var2 + var3)/ 4; // Aversge is in $v1
    return;
}

Subprogram getLarger($a0, $a1) {
    $v0 = $a0
    if ($a1 > $a0)
        $v0 = $a1
    return;
}
```

Assigment 9.2

- In the utils.asm file, fix the PrintInt subprogram so that it can call the PrintNewLine subprogram to print a new line character.

Assigment 9.3

- Implement a subprogram that prompt the user for 3 numbers, finds the median (middle value) of the 3, and returns that value to the calling program.

Assigment 9.4

- Write a procedure to find the largest, the smallest and their positions in a list of 8 elements that are stored in regsiters $s0 through $s7.

- For example:
  - Largest: 9,3 -> The largest element is stored in $s3, largest value is 9
  - Smallest: -3,6 -> The smallest element is stored in $s6, smallest value is -3

- Tips: using stack to pass arguments and return results.

Assigment 9.5

- Implement a subprogram that prompts a user to enter values from 0..100 until a sentinel value of -1 is entered. Return the average of the numbers to the calling program.

Assigment 9.6

- Implement a recursive program that takes in a number and finds the square of that number through addition.
- For example if the number 3 is entered, you would add 3+3+3=9.
- If 4 is entered you would add 4+4+4+4=16.
- This program must be implemented using recursion to add the numbers together.

Conclusions

Before passing the laboratory exercises, think about the questions below:

- What registers that the Caller need to save by convention?
- What registers that the Callee need to save by convention?
- In push label of Home Assignment 3, could we change the order of adjust stack and store word operations? If yes, what should we have to modify?
- What is stack pointer?
- What is frame pointer?

End of week 9