

What you will learn.

In this chapter you will learn:

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

Chapter 8 Reentrant Subprograms

In Chapter 5 the concept of a subprogram was introduced. At the time the `jal` operand was introduced to call a subprogram, and the `jr $ra` instruction was introduced 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 in real programs. This chapter will implement an infrastructure for subprogram dispatching that removes the non-reentrant problem from subprograms.¹

Chapter 8.1 Stacks

This section will cover the concept of a program stack. To do so, first the data structure of a stack will be implemented.

Chapter 8.1.1 Stack data structure: definition

This section will cover what a stack is and how it works.

A **stack is a Last-In-First-Out data structure**. The most used metaphor for a stack is trays in a lunchroom. When a tray is returned to the stack, it is placed on top, and each subsequent tray placed on top of the previous tray. When a patron wants a tray, they take the first one off the top of the stack. Hence the last tray returned is the first tray used. To better understand this, note that some trays will be heavily used (those on top), and some, such as the bottom most tray, might never be used.

In a computer, a stack is implemented as an array on which there are two operations: push and pop. Push places an item on the top of the stack and adds 1 to an array index. A pop operation removes the topmost item and subtracts 1 from the size of the stack.

¹ 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 MIPS assembly subprograms are implemented using a Frame Pointer (`$fp`) to keep track of parameters and return values. Check the additional resources for this Module for the MIPS Calling Convention document.

The following is an example of how a stack works. The example begins by creating a data structure for a stack, which is shown below. Note that there is no error checking.

```
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];
    }
}
```

Program 8-1: Stack class definition

To see how the array works, the following example illustrates these operations:

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

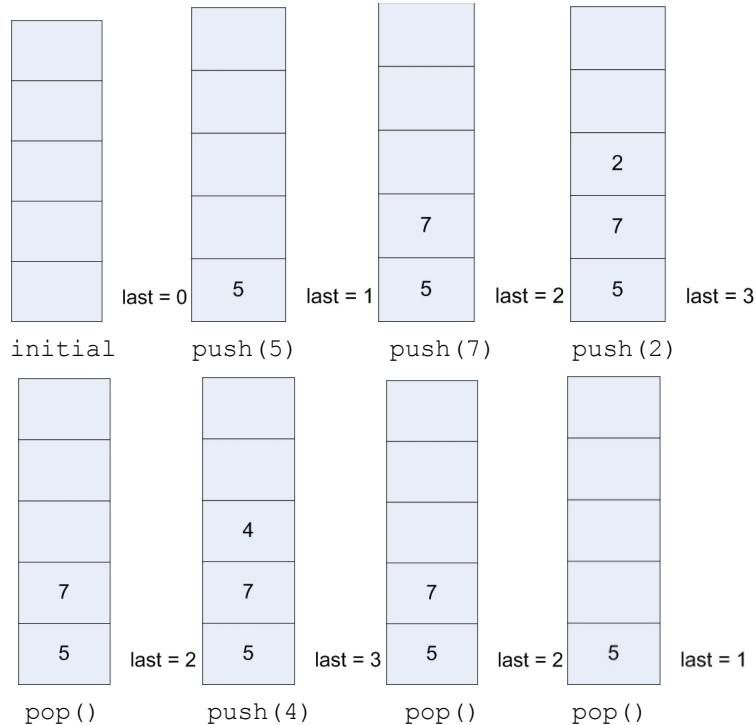


Figure 8-1: Push/Pop Example 1

The output of the program would be 2, 4, 7.

Chapter 8.1.2 Another stack implementation

There is another stack implementation that is better suited to our needs in describing the program stack. This stack implementation is still a LIFO data structure, but it will ‘grow down’ in memory and the items on it will not be fixed sized.

In this implementation, the stack will store groups of characters (strings) of varying sizes. The storage for the stack will use an array of size 7 elements. Each character will be stored in an element. Because the stack grows downward in memory, the first item in the string will be allocated at `elements[size-1]`, and each time a new string is desired, it will be stored in the next lowest available place in memory. Because the strings are not all the same size, the size of each string must be stored in the array with the characters. This means that with each string a number must be stored which gives the size of the string.

In the above paragraph, it says that the strings are not all the same size, but it purposefully does not say that the strings are not all fixed size. The strings must have a known, fixed size when the push operation is executed. This is an important point which will have an impact when discussing the program stack. The data structure for this new `stringStack` class is defined below.

```
class stringStack
{
    int SIZE=7;
    int elements[SIZE]; # Note that characters are stored as int
    int last = SIZE-1;

    push(String s) {
        last = (last - s.length())-1;
        elements[last] = s.length();
        int i = last + 1;
        for (char c in s)
        {
            elements[i] = c;
            i = i + 1;
        }
    }

    String pop()
    {
        int i = elements[last];
        int j = last + 1;
        last = last + i;
        for ( ; j < last; j++) {
            s = s + elements[j];
        }
        return s;
    }
}
```

Program 8-2: String class stack definition

Note that in the `stringStack` class the size of the string is stored with the string on the stack and the push and pop operations use the size to determine the amount of memory to be allocated/deallocated.

Chapter 8.2 The Program Stack

This section will cover the reason why non-reentrant subprograms are a problem. A special type of stack, the program stack, will be introduced to resolve this problem. This stack will allow memory for a subprogram to be allocated when the subprogram is entered and freed when the program is exited.

Chapter 8.2.1 The non-reentrant subprogram problem

The subprograms presented in Chapter 3 had a limitation that they could not call other subprograms. The problem is illustrated in the following example.

```
.text
.globl main
main:
    jal BadSubprogram

    la $a0, string3
    jal PrintString

    jal Exit

BadSubprogram:
    la $a0, string1
    jal PrintString

    li $v0, 4
    la $a0, string2
    syscall
    jr $ra


.data
string1: .asciiz "\nIn subprogram BadSubprogram\n"
string2: .asciiz "After call to PrintString\n"
string3: .asciiz "After call to BadSubprogram\n"
.include "utils.asm"
```

To the programmer who wrote this it appears that the `jal` and `jr` operators act like subprogram call and return statements. Therefore the expected output is:

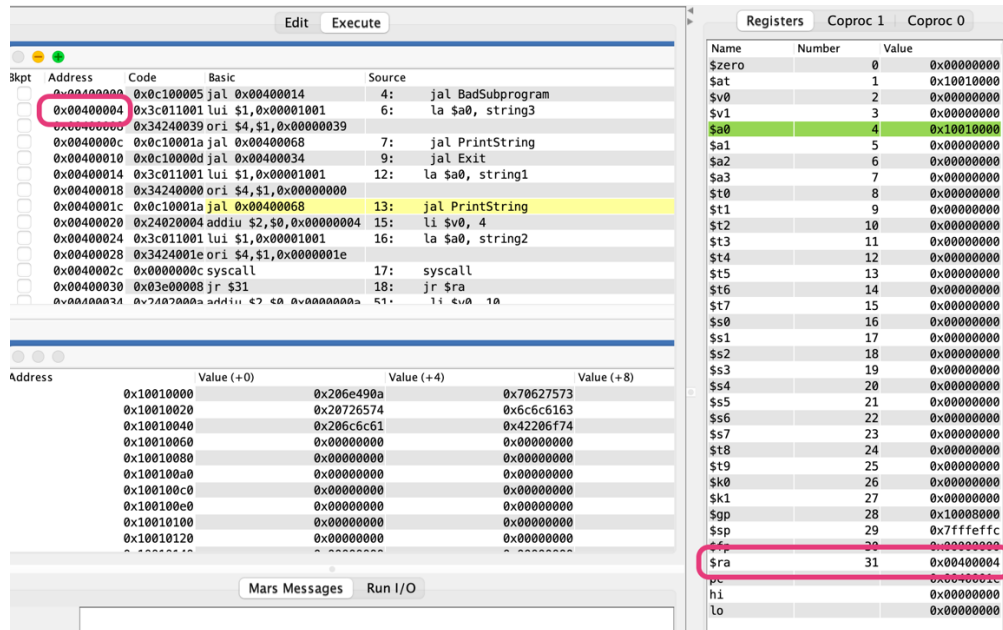
```
In subprogram BadSubprogram
After call to PrintString
After call to BadSubprogram
```

However, when this program runs, what appears to be an infinite loop appears, and the output is:

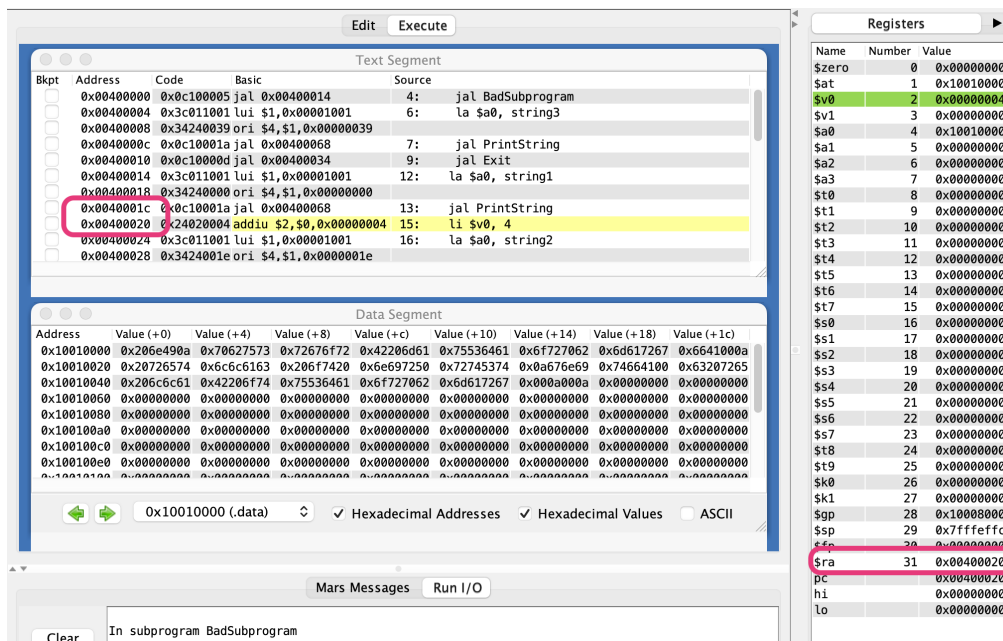
```
In subprogram BadSubprogram
After call to PrintString
After call to PrintString
...
```

In the example subprogram *BadSubprogram* above, the subprogram is making a call to another subprogram, *PrintString*. To see what is happening, we can step through the execution of the program command by command using the 'Run one step at a time' button:  1

You should get a MARS screen that looks like the following image.



From this MARS screen image, note that before the call to *PrintString* the value of the *\$ra* register is address 0x00400004. This is the address which was set when the subprogram *BadSubprogram* was called and is the address the subprogram *BadSubprogram* should return to when it completes execution. Now click the green arrow key to advance the program a few more commands and stop at the statement after the "jal Subprogram" statement, as shown below.



Note that now the `$ra` register points to the current statement address, `0x00400020`. The `PrintString` subprogram needed to have a return address, and so when the `"jal PrintString"` instruction was executed, it wrote over the address in the `$ra` register. When this register was overwritten, the subprogram `BadSubprogram` lost its link back to the main subprogram. Now when `"jr $ra"` instruction runs to return to main, the `$ra` is incorrect, and the program keeps going back to the same spot in the middle of `BadSubprogram`. This is an infinite loop, though it was achieved through a strange mechanism. So, the `jal` and `jr` operators cannot be thought of as call and return statements. These two operators simply transfer control of the program and a call and return mechanism is more complicated to implement than these simple operators in assembly.

Chapter 8.2.2 Making subprograms re-entrant

About the only good thing about the `BadSubprogram` example is that it identifies the problem with the subprogram calling mechanisms in assembly, that the `$ra` needs to be stored when the subprogram is entered and restored just before the program leaves. But the problem with the `$ra` is also a problem with any register that the program uses, as well as any variables that are defined in the subprogram. Space is needed in memory to store these variables.

The space to store these variables for a subprogram is a stack. When the program begins to run, memory at a high address, in this case `0x7ffffe00`, is allocated to store the stack. The stack then grows downward in memory. Generally, the area allocated to the stack is sufficient for any properly executing program, though it is common for incorrect programs to reach the limit of the stack memory segment. If a properly running program reaches the limit of the stack memory segment, it can always allocate larger segments of memory.

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 the stack, freeing memory that it might have allocated, and restoring the stack to the state it was before the subprogram was called. The following program, using the subprogram `GoodSubprogram`, highlights the `$ra` register while the subprogram is running.

```
.text
.globl main
main:
    jal GoodSubprogram

    la $a0, string3
    jal PrintString

    jal Exit

GoodSubprogram:
    addi $sp, $sp, -4    # save space on the stack (push) for the $ra
    sw $ra, 0($sp)      # save $ra
    la $a0, string1
    jal PrintString
```

```
li $v0, 4
la $a0, string2
syscall
```

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

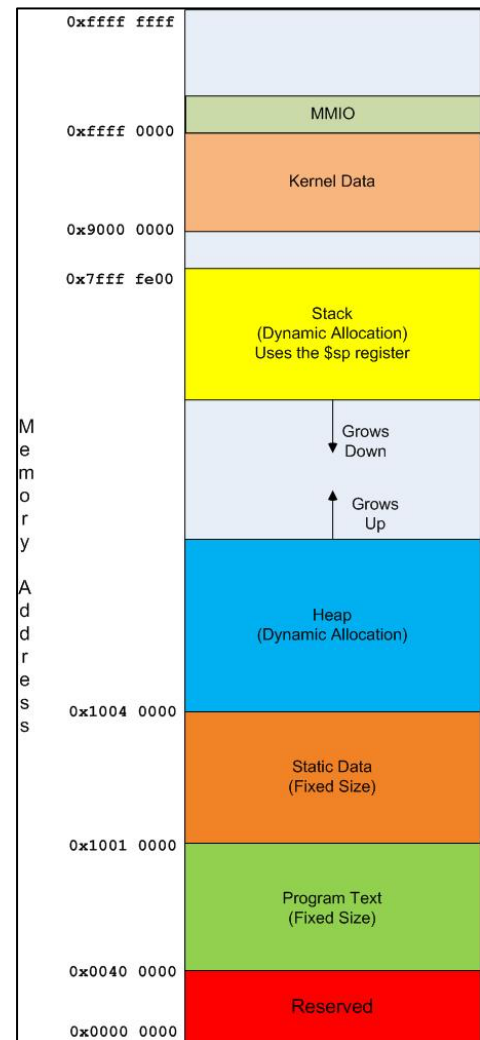
```
.data
string1: .asciiz "\nIn subprogram GoodExample\n"
string2: .asciiz "After call to PrintString\n"
string3: .asciiz "After call to GoodExample\n"
.include "utils.asm"
```

This program will work as expected and the infinite loop is gone. To show why this program works, the highlighted lines in the program are explained. The first set of highlighted lines are the following.

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

This is an example of the type of code that should be placed at the start of a reentrant subprogram. When the subprogram is entered, the stack pointer (\$sp) register points to the current end of the stack. All subprograms before this have incremented the \$sp to allocate space for their automatic variables, so all previous subprograms have stack frames (or activation records) on the stack for their execution. So, 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 its variables.

Remember that the stack grows downward, which is why 4 is subtracted from \$sp when the space is allocated. The allocation of 4 bytes is the amount needed to store the \$ra. When the GoodSubprogram is run, and execution stopped immediately after the \$ra is written to the stack, the MARS screen would look as follows.



The screenshot shows a MIPS assembly editor with three main sections: Text Segment, Registers, and Data Segment.

Text Segment: The assembly code is shown with addresses from 0x00400000 to 0x00400050. A red box highlights the following instructions:

```

0x00400013: 0xafbf0000 sw $31, 0x00000000($29) 13: sw $ra, 0($sp)      # save $ra
0x0040001c: 0x3c011001 lui $1, 0x00001001      14: la $a0, string1

```

Registers: The register file is shown on the right. The register \$sp is highlighted in green and circled, showing its value as 0x7ffffeff8.

Data Segment: The data segment is shown at the bottom. A red box highlights the value 0x00400004 at address 0x7ffffeff8. A red arrow points from the register \$sp to this value.

The value stored in the register `$sp` is (0x7ffffeff8) as shown in green and circled to the right. This is the address location in the stack of the value for our return address, `$ra`. Look at the top row in the Data Segment section. The address on the far left of this row is 0x7ffffefe0. We are offset from this address by 0x18 (the circled area in red). If we add 0x00000018 to 0x7ffffefe0 we get 0x7ffffeff8. That is the address in the stack where our `$ra` has been stored (0x00400004).

The second set of highlighted code, shown below, is an example of the type of code which should be placed just before the last statement in a subprogram. In this code, 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. The stack frame for this subprogram is then *popped* by adding the amount of memory used back to the stack.

```

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

```

Note that when using a HLL compiler, the compiler will decide if the overhead of a stack is needed or not and will handle any of the mechanics to handle the program stack. In assembly, this is up to the programmer. If the program does not need to share any data or transfer control outside of the subprogram, the allocation of the stack frame can be avoided.

Chapter 8.3 Recursion

Recursion is a way to divide a problem up into successively smaller instances of the same problem until some stopping condition (or base case). The individual solutions to all the smaller problems are then gathered and an overall solution to the problem is obtained.

Unfortunately, the types of problems that easily lend themselves to recursive solutions are more complex than can be covered in an introductory programming text such as this. Thus, the example problems which are presented are more easily solved using other means like iteration.

Chapter 8.3.1 Recursive multiply in a HLL

To implement recursion, both the current state of the solution as well as the path that has led to this state must be maintained. The current state allows the problem to be further subdivided as the recursion proceeds towards the base case. The path to the current state allows the results to be gathered back together to achieve the results. This is a perfect situation for a stack.

An example is a recursive definition of a multiply operation. Multiplication ($m \times n$) can be defined as adding the multiplier (m) to itself the number of times indicated by the multiplicand (n). Thus, a recursive definition of multiplication is the following:

$$M(m, n) = \begin{cases} m & (\text{when } n = 1) \\ m + M(m, n-1) & \text{else} \end{cases} \quad \# \ m \times n$$

This is implemented in pseudocode below.

```
subprogram global main()
{
    register int multiplicand
    register int multiplier
    register int answer
    m = prompt("Enter the multiplicand n")
    n = prompt("Enter the multiplier m")
    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)
}
```

The following MIPS assembly language program implements the above pseudocode program. Note that at the start of each call to multiply, the `$ra` is stored. In all cases but the first call to the subprogram, the `$ra` will contain the same address. The stack records storing the `$ra` are just a way to count how far into the stack the program has gone, so it can return the correct number of times.

```

.text
.globl main
main:
    # register conventions
    # $s0 - m
    # $s1 - n
    # $s2 - answer

    la $a0, prompt1      # get the multiplicand
    jal PromptInt
    move $s0, $v0

    la $a0, prompt2      # get the multiplier
    jal PromptInt
    move $s1, $v0

    move $a0, $s0
    move $a1, $s1

    jal Multiply          # do multiplication
    move $s2, $v0

    la $a0, result        # print the answer
    move $a1, $s2
    jal PrintInt

    jal Exit

Multiply:
    addi $sp, $sp, -8     # push the stack
    sw $a0, 4($sp)        # save $a0
    sw $ra, 0($sp)        # save the $ra

    seq $t0, $a1, $zero    # if (n == 0) return
    addi $v0, $zero, 0     # set return value
    bnez $t0, Return

    addi $a1, $a1, -1      # set n = n-1
    jal Multiply          # recurse
    lw $a0, 4($sp)        # retrieve m
    add $v0, $a0, $v0      # return m+multiply(m, n-1)

Return:
    lw $ra, 0($sp)        # pop the stack
    addi $sp, $sp, 8
    jr $ra

.data
prompt1: .asciiz "Enter the multiplicand: "
prompt2: .asciiz "Enter the multiplier: "
result:  .ascii  "The answer is: "
.include "utils.asm"

```

Program 8-3: Recursive multiplication

Chapter 8.4 Exercises

- 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; // Average is in $v1
    return;
}

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

Note the use of the variables `var0...var3`. Because the values of $\$a0$ and $\$a1$ (at least) are changed on the call to `getLarger`, they will not be available when they are needed to calculate the average and must be stored on the stack. To do this problem correctly, you must calculate the maximum value using the `getLarger` subprogram shown here, and it must be called before the average is calculated. This implies that at a minimum $\$a0$ and $\$a1$ must be stored on the stack, though I would suggest all four be stack variables as shown here.

It is possible to create a solution which does not require the use of the stack variables, for example by simply calculating the average first. Such solutions do not answer the issue of how to handle variables that change using the stack and are thus incorrect.