Computer Architecture - Lab 6

The Runtime Stack

May 2, 2019

1 Instruction

This lab topics:

- Stacks
- The stack pointer register `$sp`
- Push and Pop stack operations
- The MIPS runtime stack

Stack

A **stack** is a way of organizing data in memory. Data items are visualized as behaving like a stack of physical items. Often a stack is visualized as behaving like a stack of dinner plates. Data are added and removed from the stack data structure in a way analogous to placing plates on the stack of plates and removing them.

Normally with a stack of dinner plates all operations take place at the top of the stack. If you need a plate, you take the one at the top of the stack. If you add a plate to the stack, you place it on top of the stack.

Upside-down MIPS Stack

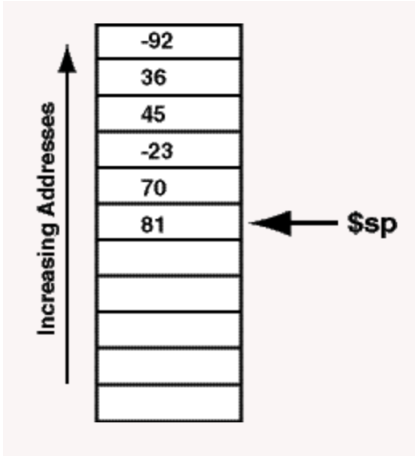
Stack-like behavior is sometimes called "LIFO" for Last In First Out.

The data elements in our stacks are 32-bit words. In general, stacks can be used for all types of data. But in these chapters, stacks contain only 32-bit MIPS full words.

The picture shows a stack of MIPS full words. The stack pointer register `$sp` by convention points at the top item of the stack. The stack pointer is register `$29`. The mnemonic register name `$sp` is used by the extended assembler.

In the usual way of drawing memory the stack is upside-down. In the picture, the top item of the stack is 81. The bottom of the stack contains the integer -92.

Before the operating system starts your program it ensures that there is a range of memory for a stack and puts a suitable address into `$sp`.



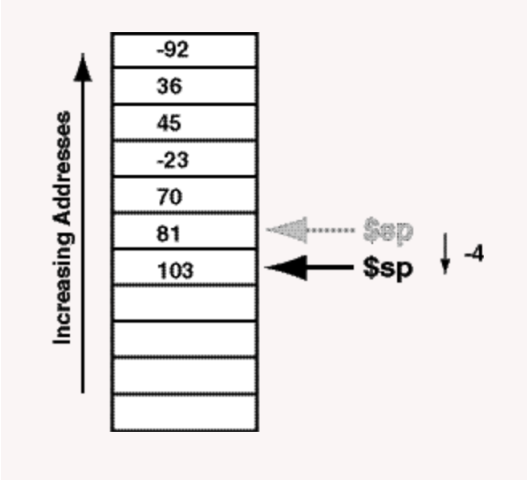
Push

By software convention, `$sp` always points to the top of the stack. Also by convention, the stack grows downward (in terms of memory addresses). So, for our stack of 4-byte (full word) data, adding an item means subtracting four from `$sp` and storing the item in that address. This operation is called a **push** operation.

To **push** an item onto the stack, first **subtract** 4 from the stack pointer, then store the item at the address in the stack pointer.

Here is what that looks like in code. Say that the value to push on the stack is in register `$t0`:

```
                        # PUSH the item in $t0:
 subu $sp,$sp,4         #   point to the place for the new item,
 sw   $t0, 0($sp)       #   store the contents of $t0 as the new top.
```



Pop

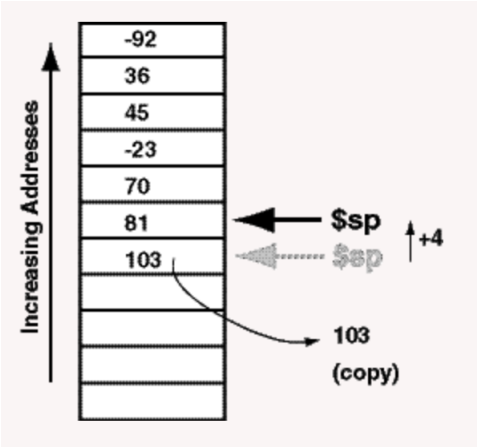
Removing an item from a stack is called a **pop** operation. In the real-world analogy an item is actually removed: a dish is physically moved from the stack. In a software stack, "removal" of an item means it is copied to another location and the stack pointer is adjusted.

The picture shows a pop operation. The data is first copied from the top of stack to the new location and then the stack pointer is increased by four.

To **pop** the top item from a stack, copy the item pointed at by the stack pointer, then **add 4** to the stack pointer.

Here is what that looks like in code. Say that we want the value to be popped into `$t0`:

```
                    # POP the item into $t0:
lw   $t0,($sp)      #   Copy top item to $t0.
addu $sp,$sp,4      #   Point to the item beneath the old top.
```



Example 1
The stack is often used to hold temporary values when most registers are already in use. An example of this is how a compiler translates an arithmetic expression into machine codes that uses a stack. Say that the arithmetic expression is $ab - 12a + 18b - 7$.

Say that only `$t0` and `$t1` are available. Perhaps only two registers are available because the compiler has already output code that uses all the others.

Before SPIM starts running a program it initializes the stack pointer `$sp` appropriately. On a computer with a full operating system, the stack pointer is initialized by the operating system before control is passed to a user program.

Terms of the expression are pushed onto the stack as they are evaluated. Then the sum is initialized to -7 and the terms on the stack are popped one by one and added to the sum.

Here is the finished program. If you had plenty of registers available you would use them to hold the terms of the expressions, and not use the stack. But in a large program, with many registers in use, you might do it this way.

```
# Evaluate the expression ab - 12a + 18b - 7
#
# Settings: Load delays OFF; Branch delays OFF,
#           Trap file    ON; Pseudoinstructions ON
```

```
        .globl  main

main:
        lw      $t0,a           # get a
        lw      $t1,bb          # get b
        mul     $t0,$t0,$t1     # a*b
        subu    $sp,$sp,4       # push a*b onto stack
        sw      $t0,0($sp)

        lw      $t0,a           # get a
        li      $t1,-12         #
        mul     $t0,$t0,$t1     # -12a
        subu    $sp,$sp,4       # push -12a onto stack
        sw      $t0,0($sp)

        lw      $t0,bb          # get b
        li      $t1,18          #
        mul     $t0,$t0,$t1     # 18b
        subu    $sp,$sp,4       # push 18b onto stack
        sw      $t0,0($sp)

        li      $t1,-7          # init sum to -7
        lw      $t0,($sp)       # pop 18b
        addu    $sp,$sp,4
        addu    $t1,$t1,$t0     # 18b -7

        lw      $t0,0($sp)       # pop -12a
        addu    $sp,$sp,4
        addu    $t1,$t1,$t0     # -12a + 18b -7

        lw      $t0,0($sp)       # pop ab
        addu    $sp,$sp,4
        addu    $t1,$t1,$t0     # ab - 12a + 18b -7

done:   li      $v0,1           # print sum
        move    $a0,$t1
        syscall
        li      $v0,10          # exit
        syscall

        .data
a:      .word  0
bb:     .word  10
```

Runtime Stack

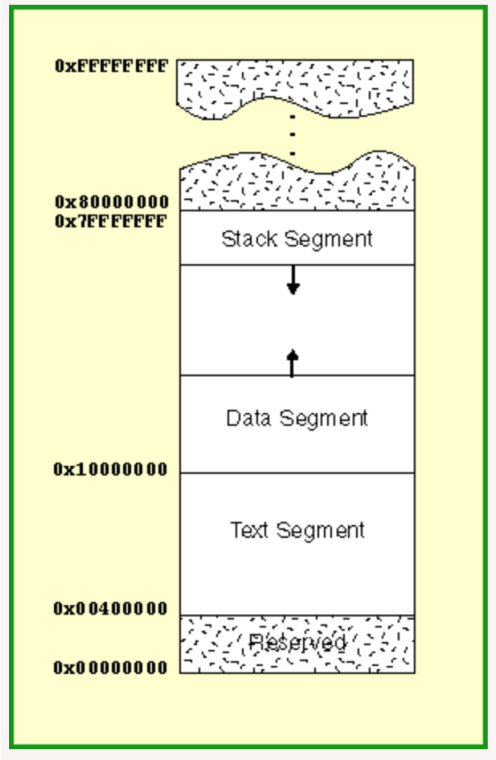
There is a finite amount of memory, even in the best computer systems. So it is possible to push more words than there are words of memory. Usually this would be the result of an infinite loop because when a program is first entered the operating system gives it space for a very large stack.

The picture shows how a typical operating system arranges memory when a program starts. There are four gigabytes of (virtual) memory available in total. The section of memory from 0x10000000 to 0x7FFFFFFF is available for the data segment and the stack segment. This is 1.8 Gigabytes of space.

When the program is started the stack pointer (`$sp`) is initialized to 0x7FFFFFFC (the address of the last fullword of user memory). As the program runs, the stack grows downward into the available space. The data segment grows upward as the program runs.

In a dynamic program, the segments grow and shrink. If the combined size of the segments exceeds the available space their boundaries will meet somewhere in the middle of the range. When this happens there is no memory left.

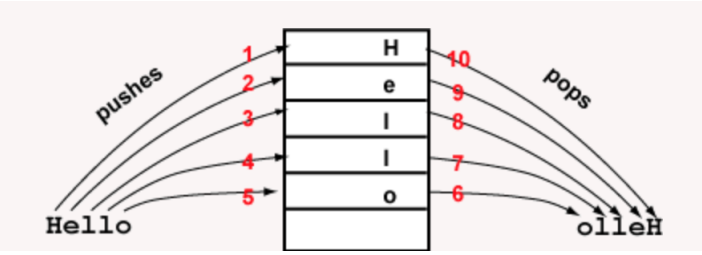
Another (inferior) way of arranging memory might be to divide the space above the text segment so the stack gets half and the data segment gets half. But now the stack could grow too large for its allocated space, even if there was a tiny data segment that used little of its space.



Example 2

Here is a classic example of how a stack is used. The user enters a string, which is stored as a null-terminated string in a character buffer. The program then reverses the order of the characters in the buffer, and then writes out the reversed buffer. To understand how the program works inspect the following diagram. The string "Hello" is pushed onto the stack, from the buffer, character by character, starting with the 'H'. Then the characters are then popped from the stack back into the original string buffer. The last character pushed is the first one out. This reverses the order of the characters.

We will always push and pop full words (four bytes). Each character on the stack will be contained in the low order byte of a fullword.



The first section of the program reads a line from the terminal in the usual way. To shorten the program, there is no user prompt.

Next, the stack is initialized. The program itself does not initialize the stack pointer because this is done by the operating system when it before it passes control to the program. (In our case, the stack pointer is initialized by SPIM.) Null is pushed onto the stack. Later on, the stack will be popped until this null is encountered.

In the next stage, characters from the character buffer are pushed one by one onto the stack. The first instruction (at `pushl:`) uses indexed addressing to load the current character from the buffer (`str:`) into the least significant byte of `$t0`.

Next, the current character is tested. If it is null (zero) then control branches out of the loop. Otherwise the character is pushed onto the stack. Then the process is repeated.

When the null byte of the null-terminated input string is encountered, the first loop exits and the next loop begins. This next loop pops characters (contained in full words) off of the stack until the null at the bottom of the stack is encountered. Each character popped off the stack is placed into the string buffer, overwriting the character originally there.

The null at the end of the input string is not overwritten. It will remain there as part of the null-terminated result string.

The last phase of the program prints out the result string. There is nothing new here. If you want to see the complete program, copy and paste the several above sections into a text file.

```
# Reverse and output a user-supplied string
#
# Settings: Load delays OFF; Branch delays OFF,
#           Trap file    ON; Pseudoinstructions ON
#
# $t0 --- character pushed or popped
# $t1 --- index into string buffer str

        .text
        .globl  main

main:   #input the string
        li      $v0,8           # service code
        la      $a0,str         # address of buffer
        li      $a1,128         # buffer length
        syscall
```

```
        li      $t0,0           # push a null
        subu    $sp,$sp,4       # onto the stack
        sw      $t0,0($sp)       # to signal its bottom
        li      $t1,0           # index of first char in str buffer

        # push each character onto the stack
pushl:
        lbu     $t0,str($t1)    # get current char into
                                # a full word
        beqz    $t0,stend       # null byte: end of string

        subu    $sp,$sp,4       # push the full word
        sw      $t0,0($sp)       # holding the char

        addu    $t1,1           # inc the index
        j       pushl           # loop

        # pop chars from stack back into the buffer
stend:  li      $t1,0           # index of first byte of str buffer
popl:
        lw      $t0,0($sp)       # pop a char off the stack
        addu    $sp,$sp,4
        beqz    $t0,done        # null means empty stack

        sb      $t0,str($t1)    # store at string[$t1]
        addu    $t1,1           # inc the index
        j       popl            # loop

        # print the reversed string
done:   li      $v0,4           # service code
        la      $a1,str         # address of string
        syscall
        li      $v0,10          # exit
        syscall

        .data
str:    .space  128             # character buffer
```

2 Exercises

1. **Exercise 1 - Arithmetic Evaluation (stack-based) (40pts)**
   Write a program to evaluate $3ab - 2bc - 5a + 20ac - 16$

   Prompt the user for the values a, b, and c. Try to use a small number of registers. Use the stack to hold intermediate values. Write the final value to the monitor.

2. **Exercise 2 - String Reversal (stack-based) (40pts)**
   Write a program that asks the user for a string. Read the string into a buffer, then reverse the string using the stack. However, unlike the example above, don't push a null character

on the stack. Keep track of the number of characters on the stack by incrementing a count each time a character is pushed, and decrementing it each time a character is popped. Write out the reversed string.

3. **Exercise 3 - Vowel Removal (stack-based) (20pts)**
   Write a program that asks the user for a string. Read the string into a buffer. Now scan the string from right to left starting with the right-most character (this is the one just before the null termination.) Push each non-vowel character onto the stack. Skip over vowels.

   Now pop the stack character by character back into the buffer. Put characters into the buffer from right to left. End the string with a null byte. The buffer will now contain the string, in the correct order, without vowels.

   Write out the final string.

3 References

1. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-25/ass25_1.html