

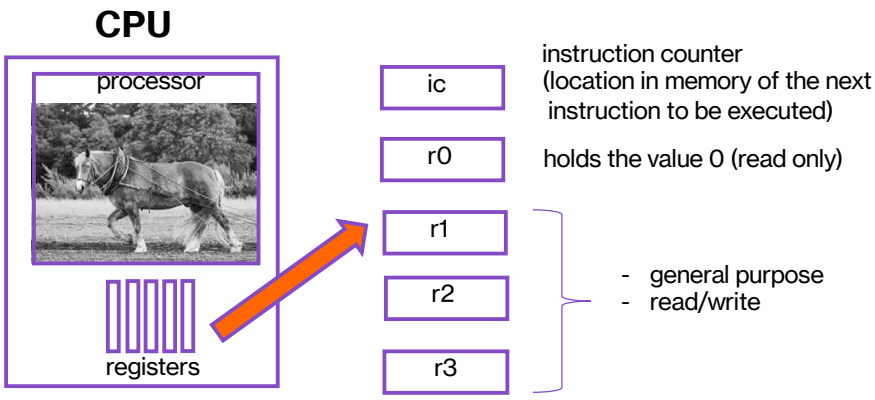
For the last lecture of the Computer Systems unit, we will talk about how calling functions works in assembly-like languages like the CS51 machine assembly we started last time.

Examples from this lecture

http://www.cs.pomona.edu/classes/cs51/cs51machine/

Remember, as with the last class meeting, you can find all the examples we will encounter today here along with a jar file that is the cs51machine program.

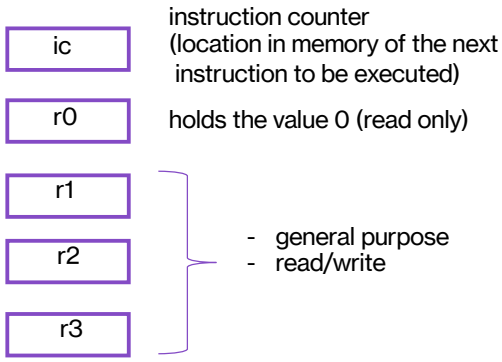
CS51 machine (processor)



Last time, we saw how in the CS51 machine the CPU works with five registers to execute programs written in its special flavor of assembly. The first register, ic, stands for instruction counter and holds the location in memory of the next instruction in memory to be executed (it will make sense in a bit, I promise). The other four registers hold data values on which the machine operates. Register r0 is special in that it always contains the value zero; any attempt to change it is ignored. The other registers, r1, r2, and r3, are general-purpose registers whose values can be read and modified by program instructions.

Program execution in the CS51 Machine

- When executing a program, the CS51 machine loops over the follow:
  - Fetch the value from `mem[ic]` for use as an instruction
  - Increment `ic` by 2
  - Decode the instruction and then execute it



As a refresher, when executing a program, the CS51 machine performs repeatedly the following steps:
1) First, it goes to the memory location indicated by the register ic and it fetches the value for use as an instruction to be executed. It then increments ic by 2 (why 2? because the word size is 2 bytes!). It proceeds with decoding and executing the instruction. And then moves to the next cycle of doing the same.

Basic structure of CS51 program

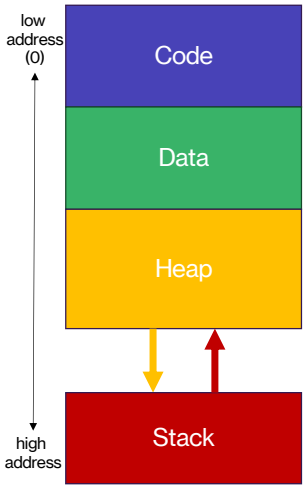
```
; great comments at the top!
;
        instruction1            ; comment
        instruction2            ; comment
        ...
label1
        instruction3            ; comment
        instruction4            ; comment
label2
        ...
        hlt
```



- whitespace before operations/instructions
- labels go here

We also saw what the basic structure of a CS51 machine looks like. Comments are signified with semicolon (;) and ignored. Instructions need to have whitespace before them. The instruction hlt terminates the program. You may also want to label chunks of code. For example, label1 might be useful if you want to start executing the code at instruction3

The organization of memory



- Memory in modern computers is organized in four basic areas:
  - **Code**: Where the instructions of our program are stored.
  - **Data**: Where the global data are stored and live for the entire duration of our program. Think of global data as variables that are created outside a function. The size of this memory region is fixed.
  - **Heap**: Where dynamic user-defined objects, lists, dictionaries, etc are created. The size is automatically adjusted in Python.
    - We will see such data structures later in this course and you will also see how to work with objects (in Java) in CS62.
  - **Stack**: Devoted to managing subprogram calls and stores their local variables and return address. Depending on the language, subprograms are also called *functions*, *methods,* or *procedures*. In Python, subprograms come in the form of functions (and methods).

6

When a program gets run it's loaded in the main memory. Memory in modern computers is organized in four basic areas. code: where the instructions of the program are stored, data, where global variables outside a function live, heap, where dynamic user-defined objects such as lists and dictionaries are created, and stack which is devoted to manage subprogram calls, that is calls to functions and the information they need (]parameters, local variables, and return address). Subprograms are known as functions, methods, or procedures.



The stack abstract data type

- A stack is a data structure (a way of organizing and storing data) that maintains a collection of elements.
- A stack is like a stack of dishes or cafeteria trays.
  - Dishes may be placed on the top of the stack or removed from the top of the stack, not the middle.
- If a stack is not empty, we may **pop** the top element.
- To add an element to the stack, we **push** it to the top.
- The element popped is the one most recently pushed. This is known as last-in-first-out or **LIFO** principle.

7

As we just saw, to manage functions we need a stack. In general, a stack is a data structure that maintains a collection of elements like a stack of dishes or cafeteria trays. If a stack is not empty, we may pop, that is remove, the top element. If the stack had been empty, we would have a stack underflow. When adding an element, we push it to the top. If we are working with a certain capacity, adding elements beyond the capacity would result to stack overflow. The key property of stacks is that they follow the LIFO, last in, first out principle. The last element we added to the stack is the first one to be removed.

Think

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```

What does this program do?

Bringing this abstract data type in the context of executing programs, we see that memory has a part called stack or call stack to take care of functions and the info they need to do their work. Let's see what the stack looks like without talking yet about functions. In the context of CS51 machine, we will assume that push takes a value from the specified register and pushes it to the top of the stack and pop removes a value from the top of the stack **and** puts it in the register. With these two operations in mind, what do you think will happen as this program gets executed instruction by instruction?

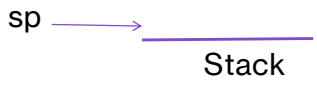
Think

r3	8

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```



add r3 r0 8 will set r3 to 8 (r3=0+8=8). We also have a register (unclear which one for this program) that points to the top of the stack.

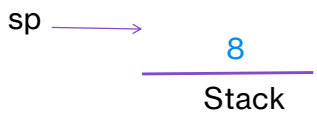
Think

r3	8

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```



psh r3 takes the contents of register r3 and pushes them to the top of the stack. This moves the stack pointer one address up (remember, the stack grows from high addresses to low addresses, that is reversely than every other part of memory)

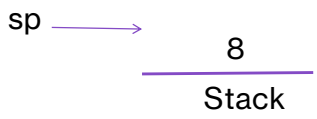
Think

r3	0

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```



add r3 r0 0 is equivalent to setting r3=0+0=0

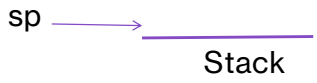
Think

r3	8

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```



pop r3 removes the value from the top of the stack, i.e. 8 and puts it in the register r3. The stack pointer goes down accordingly.

Think

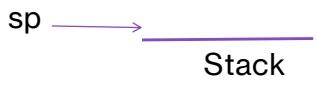
r3	8

Two operations

- `psh R`: push the value of the register R to the top of the stack
- `pop R`: pop the value from the top of the stack and put it in the register R

```
add r3 r0 8
psh r3
add r3 r0 0
pop r3
sto r3 r0
```

prints 8



As a last step, sto r3 r0 prints to the user the contents of r3, that is it prints 8

Stack frame

- When our program calls a function, the system reserves memory space for the function needs in a **stack frame**.
- The newly-created stack frame is pushed onto the top of the **call stack** and becomes the active stack frame.
- Stack frames keep track of the:
  - return address (where to go when we're done executing)
  - parameters
  - local variables

So far our programs have been simple instructions and some control flow. Now we will see how we can work with functions. When a program calls a function, the system reserves memory space for the function needs in what is known as the stack frame. That newly created stack frame is pushed onto the top of the call stack and becomes the active stack frame. Stack frames keep track of the return address, that is where to go when we're done executing that piece of code, the parameters and local variables that the function needs.

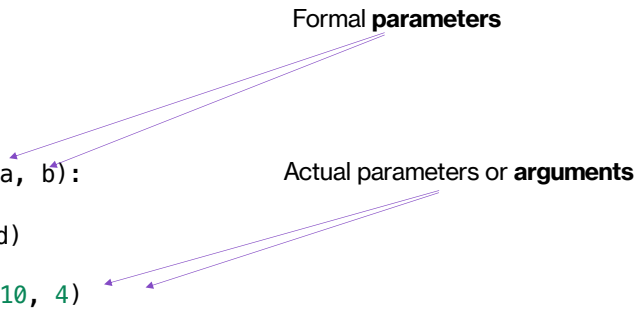
Look at the following Python program

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```



15

Look at the following Python program. We have the declaration of three functions. From the shell, we call add_then_double, which first calls add and then pass its result to double and which returns it. As a point of vocabulary here, a and b would be formal parameters. When invoking a function and passing actual parameters like -10 and 4, we call these arguments. OK, what should this program return?

The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```

When calling a function, a stack frames gets created and pushed into the call stack. The stack frame keep track of:

- return address (where to go when we're done executing)
- parameters
- local variables

```
add_then_double()
          a: -10
          b: 4
   return addr: shell
```

16

The program begins with add_then_double function when called with arguments -10 and 4. A new stack frame for add_then_double with the arguments -10 and 4 for the associated parameters a and b is being added to the top of the stack. The add_then_double stack frame becomes the active frame and remembers who called it (shell) so that it can return to it

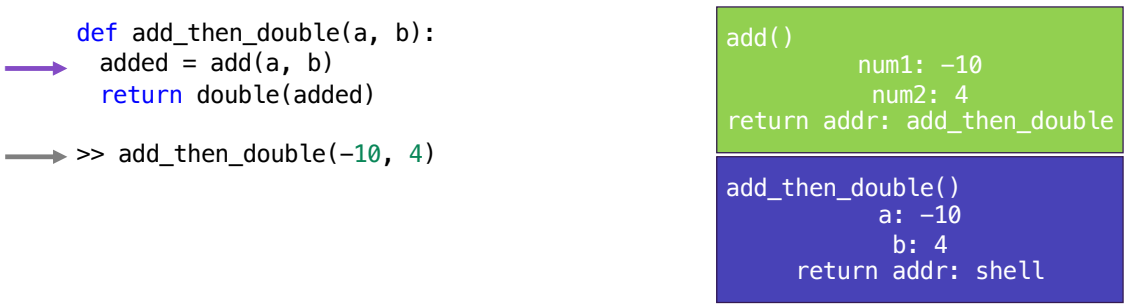
The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```



17

In the first line of the body of the add_then_double function, we see that it calls the function add so a new stack frame with the arguments -10 and 4 associated with the parameters num1 and num2 is being created for the add function and pushed to the top of the call stack.

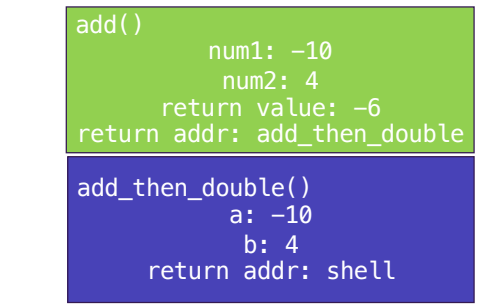
The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```



18

Within the body of add we see that it returns num1+num2. add then completes its work (adding -10 and 4) and returns -6 to the function that called it, i.e. add_then_double.

The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```

```
add_then_double()
        a: -10
        b: 4
    added: -6
return addr: shell
```

19

That results in the stack frame for add being popped from the top of the stack. Now the top of the stack is add_then_double stack frame and the local variable added is initialized to -6. We then move to the next line return double(added).

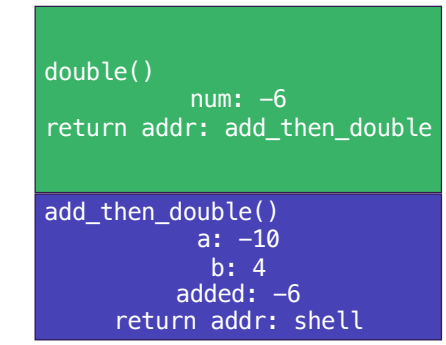
The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```



20

The last line of add_then_double calls the double function which results in a new stack frame being pushed to the top of the call stack for double, associating the parameter num with the argument -6.

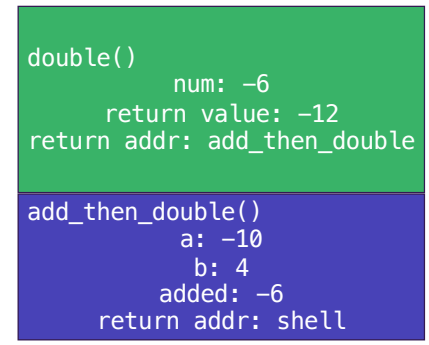
The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```



21

double returns 2*(-6)=12

The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```

```
add_then_double()
          a: -10
          b: 4
      added: -6
 return value: -12
 return addr: shell
```

22

Its stack frame is popped and that results in add_then_double returning -12.

The call stack

```
def add(num1, num2):
  return num1 + num2

def double(num):
  return 2 * num

def add_then_double(a, b):
  added = add(a, b)
  return double(added)

>> add_then_double(-10, 4)
```

"-12" is returned

23

The stack frame for add_then_double is popped and the shell receives the answer -12. The call stack is empty and the program ends.

Function calls in assembly

For high-level languages, the call stack is managed for you

In assembly, **we will manage the stack!**

Stack



In high-level languages like Python, the call stack is managed for us but in low-level languages like the assembly, we are responsible for managing the stack!

CS51 machine call stack conventions

r1 is reserved for the stack pointer, pointing to the top of the stack address

r2 contains the return address (a memory address in the code portion of where we should come back to when the function is done)

r3 contains the first parameter for the function in the active call stack frame

additional parameters go on the stack (more on this in a bit)

the result after the function completes its work should go in **r3**

Let's agree on a few things about how we will manage the call stack in the CS51 machine. The register r1 is reserved for the stack pointer which as we saw points to the top of the stack. When data are pushed to the top of the stack, the call stack pointer's address decreases (goes up). When things get popped, its address increases (goes down).
Register r2 contains the return address, that is the memory address in the code portion of where we should come back to when the function work is done.
Register r3 contains the first parameter of the function. If we have additional parameters, we would save them on the stack (we will see how later). The result of the work that the function has completed goes to register r3.

Structure of a *single* parameter function

```
function_name
       psh r2              ; save return address on stack
       ...                 ; do work using r3 as argument
                           ; put result in r3
       pop r2              ; restore return address from stack
       jmp r2              ; return to caller
```

`jmp` "jumps" to the line of code at r2. Really: sets ic = r2

- r2 has the return address
- argument is in r3
- r1 is off-limits since it's used for the stack pointer
- return value goes in r3

This is what a CS51 machine program that contains a single parameter function looks like. We start with labeling the code with the name of our function. First instruction always pushes the return address of the code that called this function to the stack so that we know where to go back. As we said, r3 is reserved for passing the argument to our single parameter. After we complete the work, we need to put the result of the function back into r3. pop r2 will allow us to restore the return address from stack. jmp jumps to the line of code that r2 had, that is it sets ic=r2 so that it can return the execution to whomever called this address

Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2             ; save the return address on the stack
22   add r3 r3 1        ; add 1 to the input parameter
24   pop r2             ; get the return address from stack
28   jmp r2             ; go back to where we were called from
```

Let's look at this CS51 machine and our first function call. You will notice that on the left side of the instructions I have noted down the addresses for each instruction. We will assume that we start at address 6 (there is some stuff before it we won't see yet). Each instruction typically takes 2 bytes (our word size in CS51 machine), this is why we go from 6 to 8. Instructions like lcw, psh, and pop take 2 words, or 4 bytes, that's why we jump from address 8 to 12 instead of 10.

Our first function call

```
6   loa r3 r0           ; get input from user for input parameter
8   lcw r2 increment    ; call increment
12  cal r2 r2

14  sto r3 r0           ; write result,
16  hlt                 ;    and halt

increment
18   psh r2             ; save the return address on the stack
22   add r3 r3 1        ; add 1 to the input parameter
24   pop r2             ; get the return address from stack
28   jmp r2             ; go back to where we were called from
```

ic	6
r1	130
r2	0
r3	0



Our program starts and we are in address 6, that is ic ==6. r1 is reserved for the stack pointer. We will assume it starts at address 130. Right now r2 and r3 are empty (set to 0). loa r3 r0 will prompt the user for input and store it in r3.

Our first function call

```
6   loa r3 r0           ; get input from user for input parameter
8   lcw r2 increment    ; call increment
12  cal r2 r2

14  sto r3 r0           ; write result,
16  hlt                 ;    and halt

increment
18   psh r2             ; save the return address on the stack
22   add r3 r3 1        ; add 1 to the input parameter
24   pop r2             ; get the return address from stack
28   jmp r2             ; go back to where we were called from
```

ic	6
r1	130
r2	0
r3	10



Let's say the user entered 10. This is stored in r3. This is the argument that will be associated with the first parameter of our function.

Our first function call

`lcw`: put the memory address of the label into the register r2

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	8
r1	130
r2	0
r3	10



ic is increased by 2. lcw (load constant word) is a new instruction. It is followed always by a register and a label, in this case r2 and increment respectively. What it does is that it puts the memory address of the first instruction right after the label into the register r2.

Our first function call

`lcw`: put the memory address of the label into the register r2

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;     and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	8
r1	130
r2	18
r3	10



That means that it will put the address of first instruction after the label increment, that is instruction psh r2, which is 18, into r2

Our first function call

`cal r2 r2`: call a function
`rY`: set ic to address in rY
`rX`: save the current ic to rX, i.e. next instruction that would have been executed

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	12
r1	130
r2	18
r3	10



ic advances by 4 (because lcw takes 4 bytes instead of 2). The next instruction, cal, is also new to us. It is followed by two registers rX and rY (in this case both are r2). rY captures which function to call by setting the ic (instruction counter) to the address in rY and rX where should the return address go when the execution of the called function is over. that is it saves the current ic to rX before it jumps to executing the function.

Our first function call

`cal r2 r2`: call a function
`rY`: set ic to address in rY
`rX`: save the current ic to rX, i.e. next instruction that would have been executed

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;     and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	18
r1	130
r2	14
r3	10



This results in the ic being updated to the address in r2, that is in address 18 and r2 now stores the address of the next instruction had we not called the function increment, that is the address of the instruction sto r3 r0 which is 14

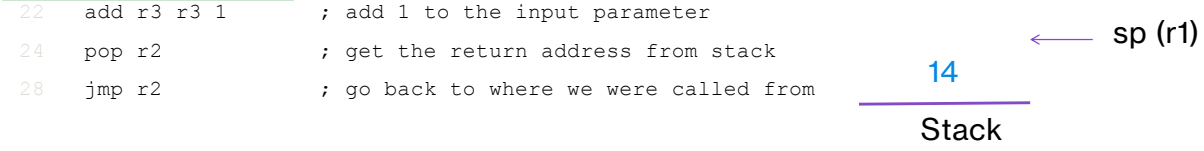
Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	18
r1	128
r2	14
r3	10



push r2 saves the return address (contents of r2) into the stack, that is it pushes 14 into the call stack and moves the stack pointer up by a word (reduces its address from 130 to 128)

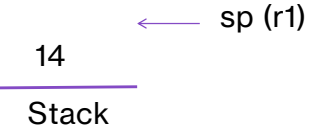
Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	22
r1	128
r2	14
r3	11



Next, we go to the next instruction which advances the ic by 4 bytes (psh in one of these instructions that take 2 words). The instruction we have to execute is add r3 r3 1 which is equal to r3=r3+1=10+1=11

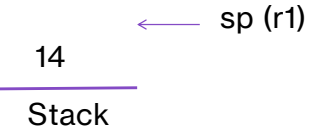
Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;     and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	24
r1	128
r2	14
r3	11



ic increments by 2. Next instruction is pop r2 which is going to pop from the stack the return address of where we should go back (14) and store it in r2.

Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	24
r1	130
r2	14
r3	11



This results in r2 being updated to 14 (I know, here it didn't change, but it could have) and increasing r1 by 2 bytes. Notice that our stack is now empty. The function increment has completed its work, its result is in r3, and it's ready to return back to whomever called it

Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;    and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	28
r1	130
r2	14
r3	11



Because pop took 4 bytes, we increment ic to 28. Finally, we have jmp r2 which tells us to update the instruction counter ic to the address stored in r2, that is to jump the execution of the program to address 14.

Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;     and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

11 ☺

ic	14
r1	130
r2	14
r3	11



So now we are done with our call to increment, we have popped its stack frame, and the result of the work of the function increment (i.e. incrementing the number 10 by 1 to 11) is in r3. The next instruction sto r3 r0 will now print to the user the answer 11!

Our first function call

```
6   loa r3 r0          ; get input from user for input parameter
8   lcw r2 increment   ; call increment
12  cal r2 r2

14  sto r3 r0          ; write result,
16  hlt                ;     and halt

increment
18   psh r2            ; save the return address on the stack
22   add r3 r3 1       ; add 1 to the input parameter
24   pop r2            ; get the return address from stack
28   jmp r2            ; go back to where we were called from
```

ic	16
r1	130
r2	14
r3	11



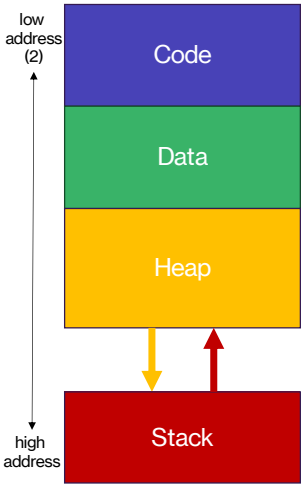
And our last program instruction is to halt the program, that is terminate here.

Real structure of CS51 program

```
; great comments at the top!
;
        lcw r1 stack

        instruction1            ; comment
        instruction2            ; comment
        ...
        hlt

;
; stack area: 50 words
;
        dat 100
stack
```



So seeing a simplified version of our memory diagram, our code starts at address 2 (0 is reserved for I/O) and then our stack grows reversely.

To the simulator!



look at increment.a51 code

Let's have a live demo of increment.a51 in the cs51machine jar.

Functions with multiple parameters

When we call a function:

ic	instruction counter
r0	holds the value 0 (read only)
r1	holds the address of the top of the stack
r2	return address
r3	first input to function

OK now that we saw how a function with a single parameter works, we need to consider what we would do if we had a function with more than one parameters. Remember that when we call a function we have the following conventions. ic is our instruction counter to keep track of where we are in execution of the program; it's off limits. r0 is also off limits because it always holds the value 0. r1, by convention, holds the address of the top of the stack. r2 is the return address so when we are done with the work of the function we know where to return. So how do we go about working with more than 1 parameters if we are out of registers?!

Functions with multiple arguments

```
function_name
        psh r2              ; save return address on stack
        loa r2 r1 4         ; load the second parameter into r2
        ...                 ; do work using r3 and r2 as arguments
                            ; put result in r3
        pop r2              ; restore return address from stack
        jmp r2              ; return to caller
```

loa R_a R_b: R_a = mem[R_b]

What does this operation do? What is the 4?

Let's look into this program. We see the label function_name and as before we have psh r2 which will save the return address to the stack. The next instruction is loa r2 r1 r4. What does this instruction do? What is the 4? We have seen before loa with two registers.

Functions with multiple arguments

```
function_name
        psh r2              ; save return address on stack
        loa r2 r1 4         ; load the second parameter into r2
        ...                 ; do work using r3 and r2 as arguments
                            ; put result in r3
        pop r2              ; restore return address from stack
        jmp r2              ; return to caller
```

loa R_a R_b: R_a = mem[R_b]

loa R_a R_b S: R_a = mem[R_b + S]

What does this operation do? What is the 4?

This version sets to the first register the address in Rb + the constant signed number, in this case 4.

Functions with multiple arguments

```
function_name
        psh r2              ; save return address on stack
        loa r2 r1 4         ; load the second parameter into r2
        ...                 ; do work using r3 and r2 as arguments
                            ; put result in r3
        pop r2              ; restore return address from stack
        jmp r2              ; return to caller
```

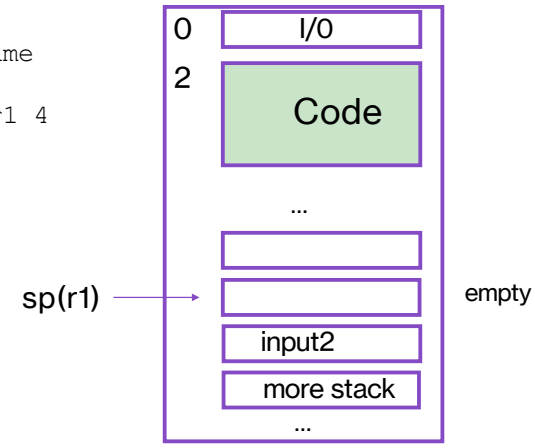
loa R_a R_b S: R_a = mem[R_b + S]

- r1 is the stack pointer and points at the top (next) slot
- stacks grow towards smaller memory values

Remember, r1 is the stack pointer and points at the top of the stack that is the next available memory slot. Since the stack grows upward, it grows towards smaller memory values.

Functions with multiple arguments

```
function_name
   psh r2
   loa r2 r1 4
```

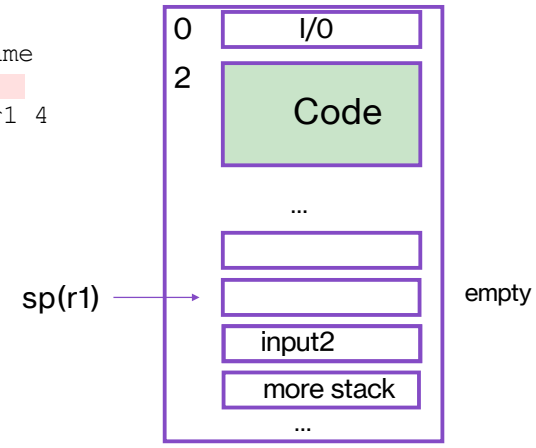


When we call a function, input1 is stored in r3, additional arguments on top of stack

As we said, when we call a function, the first argument is stored in r3 but any subsequent one is stored on the stack. Let's assume that now the function_name function is called.

Functions with multiple arguments

```
function_name
   psh r2
   loa r2 r1 4
```

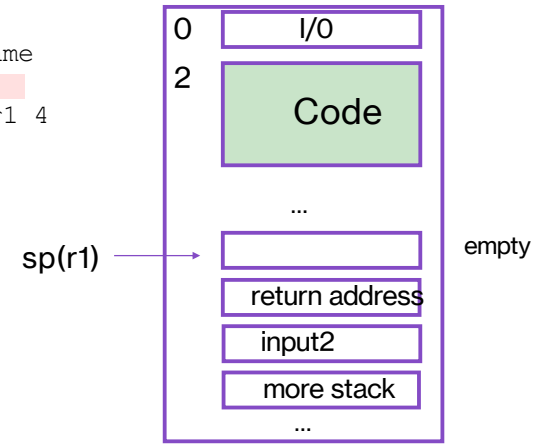


First thing we do in a function is save the return address

The first instruction is psh r2 which saves the return address stored in r2 into the stack.

Functions with multiple arguments

```
function_name
   psh r2
   loa r2 r1 4
```

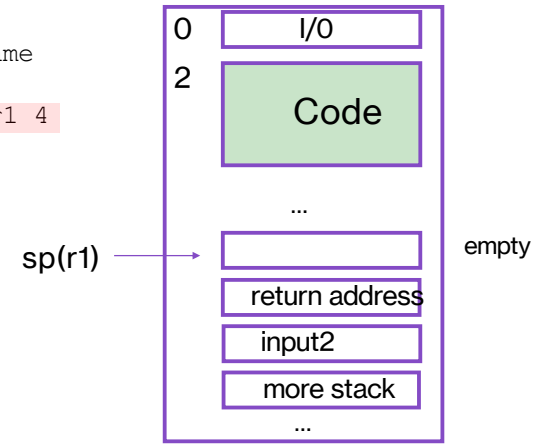


First thing we do in a function is save the return address

That results in the return address being at the top of the stack and the stack pointer decreasing (going up)

Functions with multiple arguments

```
function_name
   psh r2
   loa r2 r1 4
```



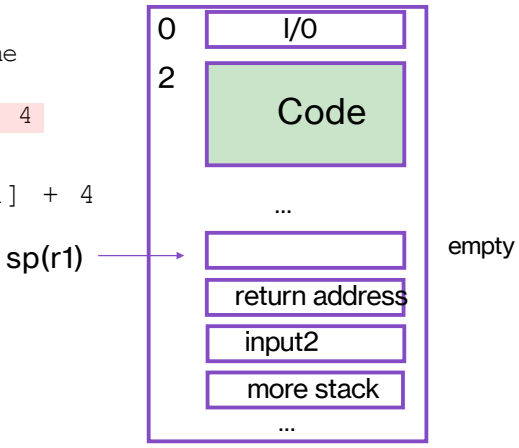
We've freed up r2. *How do we get input2 into r2?*

Now that r2's value (the return address) has been saved up in the stack, we are free to use r2. The question is how do we get input2 into r2 to use it. This will captured through loa r2 r1 r4

Functions with multiple arguments

```
function_name
   psh r2
   loa r2 r1 4

r2 = mem[r1] + 4
```



Want to look ahead two “words” from where the stack pointer is

We want to look ahead two words from where the stack pointer is, that is 2x2 bytes which results to the + 4 part.

Think: What does this function do?

```
mystery
    psh r2                  ; save the return address
    loa r2 r1 4             ; get the second argument
    add r3 r3 r2
    pop r2                  ; get the return address
    jmp r2                  ; return (answer is in r3)
```

- conventions:
  - r2 has the return address
  - input1 is in r3
  - input2 is on top of stack
  - return value goes in r3

Let's see if all that made sense by looking into this mystery function. What does it do? Remember the conventions!

- r2 has the return address
- input1 is in r3
- input2 is on top of stack
- return value goes in r3

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add_then_double
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

notice the symmetry of psh/pop

This function adds together the two arguments it receives and returns their sum. This is what it looks like as part of the larger program add.a51. Notice the symmetry of psh pop, we always do them in pairs.

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	2
r1	142
r2	0
r3	0

Let's walk through this program together. ic starts at 2. Our first instruction, lcw r1 stack, sets up the stack pointer r1 to the top of the stack. Let's say this address is calculated to be 142

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	6
r1	142
r2	0
r3	10



ic is incremented by 4 bytes based on the size of lcw. Next, we prompt the user for their first argument, say it's 10. This is stored in r3

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	8
r1	142
r2	4
r3	10



Second argument prompt and say we got 4 from the user and stored it in r2.

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

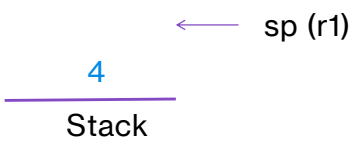
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	10
r1	140
r2	4
r3	10



The point of psh r2 is to save the second argument onto the stack. That results in r1 decreasing as sp moves up, decreasing to 140

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	14
r1	140
r2	28
r3	10



psh was also 4 bytes, so ic is advanced to 14. lcw r2 add results in the address of the first instruction under the label add being stored in r2. The first instruction is psh r2 and its address is 28, so 28 is saved in r2. Notice that we would have erased the second argument if we hadn't stored it in the stack.

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

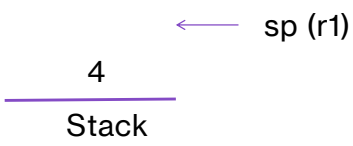
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	18
r1	140
r2	28
r3	10



cal r2 r2 does two things. It sets ic to the address in r2 and then sets r2 in the next instruction we should return to, that is to pop r0 which has an address of 20.

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

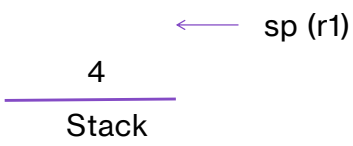
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	28
r1	140
r2	20
r3	10



That results in the ic jumping to 28 and r2 having r2. Next we push the contents of r2 (i.e. 20 which is the return address) into the stack

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

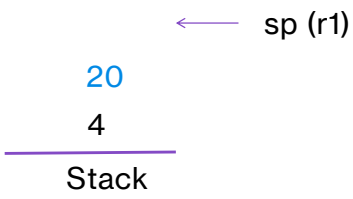
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	28
r1	138
r2	20
r3	10



So now we know where to return to when we are done with the add function because the return address is in the stack. Don't forget that as we pushed 20 into the stack, r1 decreased by 2 bytes,

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

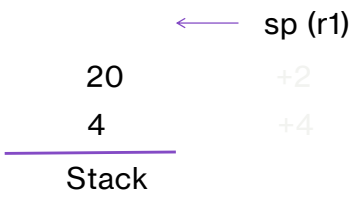
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	32
r1	138
r2	20
r3	10



ic increases by 4. loa r2 r1 4 is next and it loads into r2 the second argument by moving two words (4 bytes) from where the sp r1 is

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

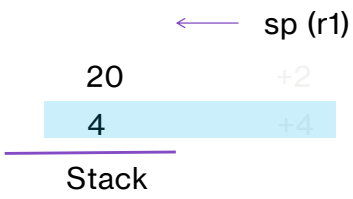
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	32
r1	138
r2	4
r3	10



That results in 4 being saved in r2, so now we can use the second argument.

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

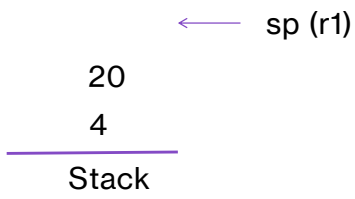
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	34
r1	138
r2	4
r3	14



add r3 r3 r2 results in r3+=r2=10+4=14. This is the result of what we wanted to calculate and as we had promised, it's in r3!

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

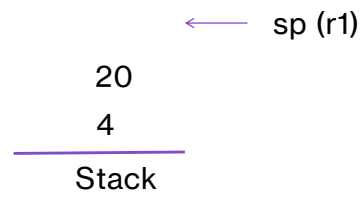
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	36
r1	138
r2	4
r3	14



Next, we pop r2 to get the return address from the stack into r2

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

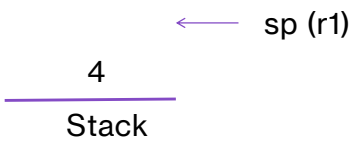
10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

ic	36
r1	140
r2	20
r3	14



This results in the stack frame removed, in r2 having 20, and sp r1 increases by 2 bytes/1 word

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

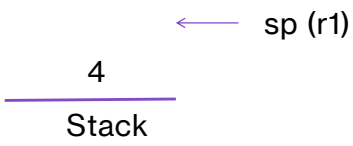
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	40
r1	140
r2	20
r3	14



so now we can jump the execution of our code back to address 20, which is the next instruction that follows our call to add.

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

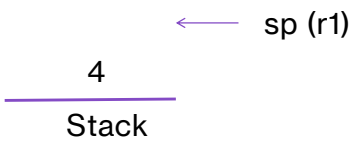
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	20
r1	140
r2	20
r3	14



That is, the instruction pop r0.

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

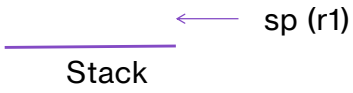
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	20
r1	142
r2	20
r3	14



That is, the instruction pop r0. pop r0 just throws the top of the stack away so that we clean it up completely. r1 is back to where it started in address 142

add (add.a51)

```
2    lcw r1 stack            ; setup the stack
6    loa r3 r0               ; get input1
8    loa r2 r0               ; get input2

10   psh r2                  ; save input2 on the stack
14   lcw r2 add              ; setup call to add
18   cal r2 r2               ; call add
20   pop r0                  ; remove input2 from stack

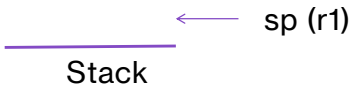
24   sto r3 r0               ; print out the result
26   hlt

add
28   psh r2                  ; save the return address
32   loa r2 r1 4             ; get the second argument
34   add r3 r3 r2
36   pop r2                  ; get the return address
40   jmp r2                  ; return (answer is in r3)

42     dat 100
stack
```

14 ☺

ic	24
r1	142
r2	20
r3	14



Next, we print to the user the result of adding its two arguments 10+4 which we have stored in r3 as 14.

add (add.a51)

```
2   lcw r1 stack            ; setup the stack
6   loa r3 r0               ; get input1
8   loa r2 r0               ; get input2

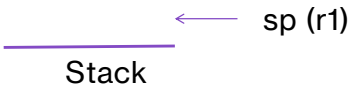
10  psh r2                  ; save input2 on the stack
14  lcw r2 add              ; setup call to add
18  cal r2 r2               ; call add
20  pop r0                  ; remove input2 from stack

24  sto r3 r0               ; print out the result
26  hlt

add
28  psh r2                  ; save the return address
32  loa r2 r1 4             ; get the second argument
34  add r3 r3 r2
36  pop r2                  ; get the return address
40  jmp r2                  ; return (answer is in r3)

42    dat 100
stack
```

ic	26
r1	142
r2	20
r3	14



And we conclude by halting the program.

Call stack in action

```
def add(num1, num2):
        return num1 + num2

def double(num):
        return 2 * num

def add_then_double(a, b):
        added = add(a, b)
        return double(added)
```



```
 lcw r1 stack    ; setup the stack
    loa r3 r0       ; get input1
    loa r2 r0       ; get input2

    psh r2          ; save input2 on the stack
    lcw r2 add_then_double ; setup call to add_then_double
    cal r2 r2       ; call add_then_double
    pop r0          ; remove input2 from stack

    sto r3 r0       ; print out the result
    hlt

add
    psh r2          ; save the return address
    loa r2 r1 4     ; get the second argument
    add r3 r3 r2    ; add r3 = arg1 + arg2
    pop r2          ; get the return address
    jmp r2          ; return

double
    psh r2          ; save the return address
    add r3 r3 r3    ; double the input2
    pop r2          ; get the return address
    jmp r2          ; return

add_then_double
    psh r2          ; save the return address
    loa r2 r1 4     ; get the second argument

    psh r2          ; setup call for add
    lcw r2 add      ; r3 has input one, push input2 on stack
    cal r2 r2       ; call add
    pop r0          ; remove input2 from stack

    lcw r2 double   ; result of add is in r3
    cal r2 r2       ; call double

    pop r2          ; get the return address
    jmp r2          ; return

    dat 100         ; setup the stack
stack
```

Next, we will see how a high-level program written in a language like Python maps to the lower-level assembly.

add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

Let's look into add_then_double.a51. The first few lines of the program include these instructions. We'll go through them instruction by instruction

add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

ic	2
r1	186
r2	0
r3	0



lcw r1 stack sets up the stack pointer r1 to the top of the stack (the instructions for how much space we allocate for the stack are further down and are not shown here). This results in r1 set to hold memory 186 where the top of the stack is.

add_then_double.a51

```
 2   lcw r1 stack     ; setup the stack
 6   loa r3 r0        ; get input1
 8   loa r2 r0        ; get input2

10   psh r2           ; save input2 on the stack
14   lcw r2 add_then_double ; setup call to add_then_double
18   cal r2 r2        ; call add_then_double
20   pop r0           ; remove input2 from stack

24   sto r3 r0        ; print out the result
26   hlt
```

ic	8
r1	186
r2	4
r3	-10



The next two instructions prompt the user for the two inputs. Let's assume that the user gave -10 and 4 which we store in r3 and r2 respectively. This is the exact same values passed to the Python add_then_double function when we saw how things would unfold in Python. So r3 has the first argument and r2 has, temporarily, the second argument.

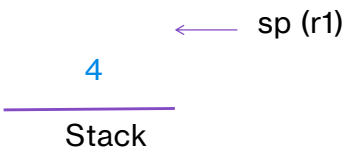
add_then_double.a51

```
2   lcw r1 stack     ; setup the stack
6   loa r3 r0        ; get input1
8   loa r2 r0        ; get input2

10  psh r2           ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2        ; call add_then_double
20  pop r0           ; remove input2 from stack

24  sto r3 r0        ; print out the result
26  hlt
```

ic	10
r1	184
r2	4
r3	-10



Since we don't have enough space to do everything we need and hold in our limited registers both inputs, we will save the second argument on the stack. That will result in 4 being pushed on the stack and r1 decreasing by 2 bytes (stack pointer going up)

add_then_double.a51

ic	14
r1	184
r2	4
r3	-10

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt

..

add_then_double
54 psh r2
```



lcw r2 add_then_double sets us up for calling the add_then_double function. The add_then_double function is further down in the file and its first instruction appears in address 54

add_then_double.a51

ic	14
r1	184
r2	54
r3	-10

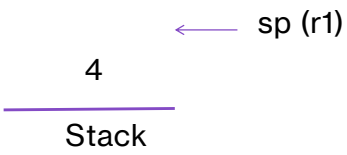
```
2   lcw r1 stack     ; setup the stack
6   loa r3 r0        ; get input1
8   loa r2 r0        ; get input2

10  psh r2           ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2        ; call add_then_double
20  pop r0           ; remove input2 from stack

24  sto r3 r0        ; print out the result
26  hlt

..

add_then_double
54 psh r2
```



so we're going to store in r2 the address 54 so that we know where the function add_then_double is.

add_then_double.a51

```
2   lcw r1 stack     ; setup the stack
6   loa r3 r0        ; get input1
8   loa r2 r0        ; get input2

10  psh r2           ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2        ; call add_then_double
20  pop r0           ; remove input2 from stack

24  sto r3 r0        ; print out the result
26  hlt

..

add_then_double
54 psh r2
```

ic	14
r1	184
r2	54
r3	-10



Next, we will do the two things: jump execution to line 54 and store in r2 the address of the next instruction had we not jumped into add_then_double, that is the address of the instruction pop r0.

add_then_double.a51

```
2   lcw r1 stack     ; setup the stack
6   loa r3 r0        ; get input1
8   loa r2 r0        ; get input2

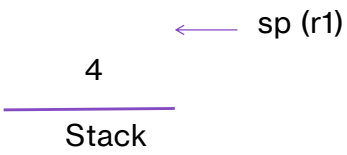
10  psh r2           ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2        ; call add_then_double
20  pop r0           ; remove input2 from stack

24  sto r3 r0        ; print out the result
26  hlt

..

add_then_double
54  psh r2
```

ic	54
r1	184
r2	20
r3	-10



That results in ic being 54 and r2 having the return address 20.

add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

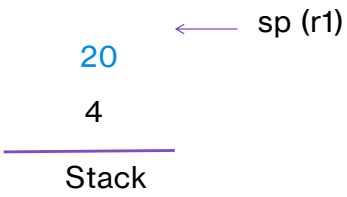
10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt

..

add_then_double
54  psh r2
```

ic	54
r1	182
r2	20
r3	-10



Next, psh r2 saves the return address into the stack, or equivalently, it pushes the stack frame for add_then_double into the call stack (and also moves sp by reducing r1 by 2 bytes)

add_then_double.a51

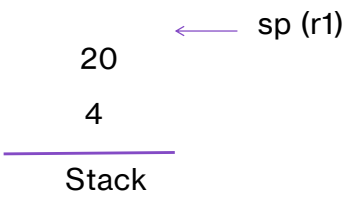
```
add_then_double
54 psh r2           ; save the return address
60 loa r2 r1 4      ; get the second argument

62 psh r2           ; setup call for add
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	54
r1	182
r2	20
r3	-10



Let's zoom into the instructions in the add_then_double function. We just completed pushing the return address from r2 into the stack.

add_then_double.a51

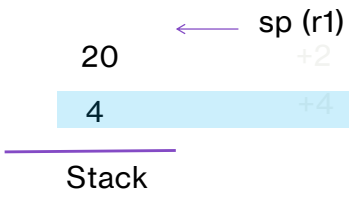
```
add_then_double
54 psh r2          ; save the return address
60 loa r2 r1 4     ; get the second argument

62 psh r2          ; setup call for add
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	60
r1	182
r2	4
r3	-10



Next, we will store into r2 the second input the user provided. Again, we will have to go 4 bytes (2 words) below the stack pointer r1. That leads us to the number 4 which we load into r2.

add_then_double.a51

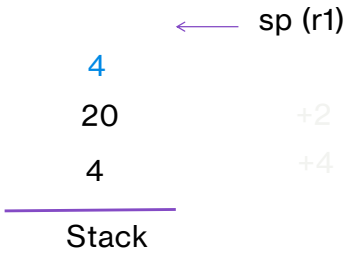
ic	62
r1	180
r2	4
r3	-10

```
add_then_double
54 psh r2          ; save the return address
60 loa r2 r1 4     ; get the second argument

62 psh r2          ; setup call for add
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```



psh r2 will store the second argument into the stack (moving sp up, reducing r1 by 2) so that the add function can use it as an input

add_then_double.a51

```
add
28    psh r2          ; save the return address
…

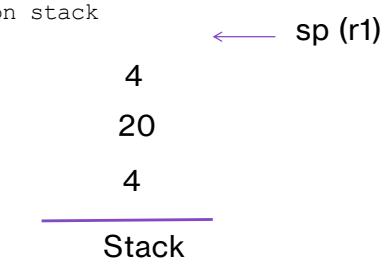
add_then_double
54 psh r2           ; save the return address
60 loa r2 r1 4      ; get the second argument

62 psh r2           ; setup call for add
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	66
r1	180
r2	28
r3	-10



Next, we are ready to make the call to add but we need to store the address of the first instruction in add into r2. add as a function is actually defined earlier so the first instruction, psh r2 has a lower memory location, 28.

add_then_double.a51

```
add
28    psh r2          ; save the return address
…

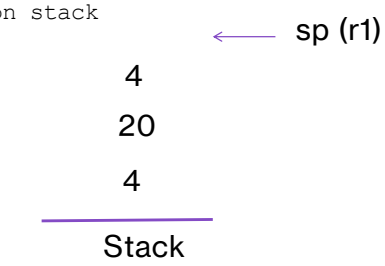
add_then_double
54 psh r2           ; save the return address
60 loa r2 r1 4      ; get the second argument

62 psh r2           ; setup call for add
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	70
r1	180
r2	72
r3	-10



cal r2 r2 sets the ic to where r2 is, that is 28, and stores in r2 the return address of the current instruction, that is the address of pop r0 which is 72

add_then_double.a51

```
add
28    psh r2          ; save the return address
…

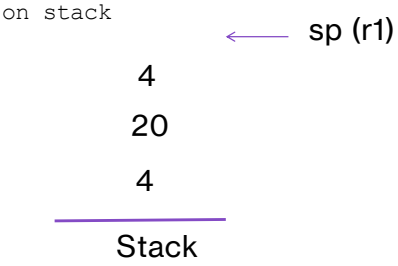
add_then_double
54 psh r2           ; save the return address
60 loa r2 r1 4      ; get the second argument

62 psh r2           ; setup call for add
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	28
r1	180
r2	72
r3	-10

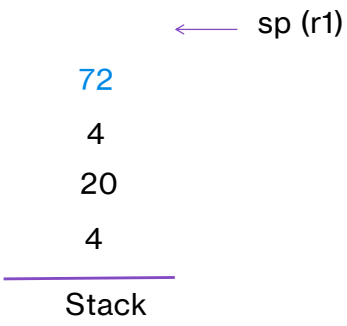


So we are now in the first line of add and r2 remembers that we need to go back to address 72 when we are done..

add_then_double.a51

```
add
28      psh r2          ; save the return address
32      loa r2 r1 4     ; get the second argument
34      add r3 r3 r2    ; add r3 = arg1 + arg2
36      pop r2          ; get the return address
40      jmp r2          ; return
```

ic	28
r1	178
r2	72
r3	-10

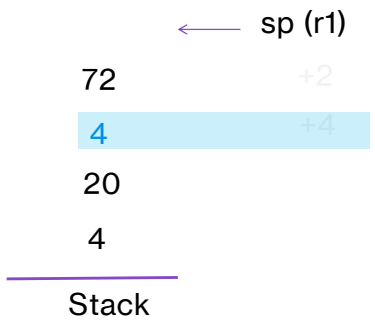


Next we have to push this return address to the stack so that we can free up r2.

add_then_double.a51

```
add
28      psh r2          ; save the return address
32      loa r2 r1 4     ; get the second argument
34      add r3 r3 r2    ; add r3 = arg1 + arg2
36      pop r2          ; get the return address
40      jmp r2          ; return
```

ic	32
r1	178
r2	4
r3	-10

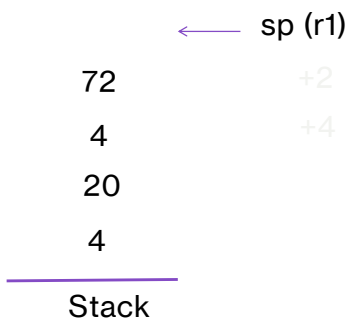


Next, we retrieve the second argument that the user provided by going 4 bytes or 2 words below the stack pointer. This will result in loading the number 4 into r2

add_then_double.a51

```
add
28      psh r2          ; save the return address
32      loa r2 r1 4     ; get the second argument
34      add r3 r3 r2    ; add r3 = arg1 + arg2
36      pop r2          ; get the return address
40      jmp r2          ; return
```

ic	34
r1	178
r2	4
r3	-6

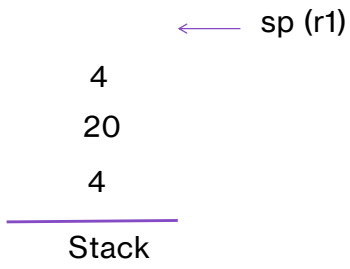


The next instruction adds the two arguments together through what r2 and r3 hold and stores the result we should return into r3.

add_then_double.a51

```
add
28      psh r2          ; save the return address
32      loa r2 r1 4     ; get the second argument
34      add r3 r3 r2    ; add r3 = arg1 + arg2
36      pop r2          ; get the return address
40      jmp r2          ; return
```

ic	34
r1	180
r2	72
r3	-6



pop r2 gets the return address 72 out of the stack and into r2. This results in r1 increasing since sp is going down

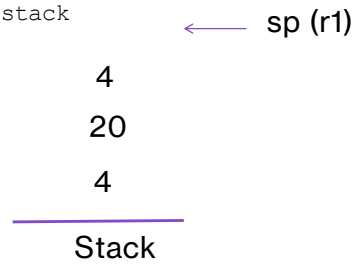
add_then_double.a51

```
add
28    psh r2           ; save the return address
32    loa r2 r1 4      ; get the second argument
34    add r3 r3 r2     ; add r3 = arg1 + arg2
36    pop r2           ; get the return address
40    jmp r2           ; return
…
add_then_double
…
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	40
r1	180
r2	72
r3	-6



Next, jmp r2 will move the ic into the return address 72

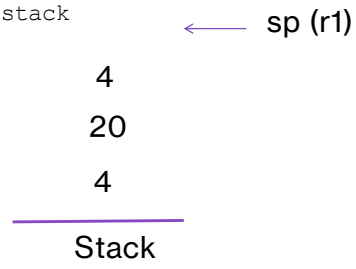
add_then_double.a51

```
add
28    psh r2          ; save the return address
32    loa r2 r1 4     ; get the second argument
34    add r3 r3 r2    ; add r3 = arg1 + arg2
36    pop r2          ; get the return address
40    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	72
r1	180
r2	72
r3	-6



So now we have popped the stack frame for add, we have the work of add(-10,4) in r3. We are ready to move into pop r0 which will remove the second argument we had stored in the stack

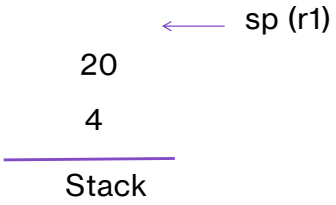
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	76
r1	182
r2	42
r3	-6



This will result in r1 increasing to 186 since the sp moved down. Next, we are ready to call the function double. We need to save the address of its first instruction in r2, that is we store the address of psh r2, 42, in r2.

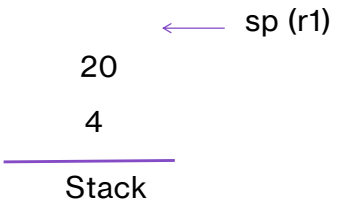
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	80
r1	182
r2	42
r3	-6



Next, cal r2 r2, will jump the execution to line 42 and update r2 to be the address of the next instruction, that is of instruction pop r2 which is 82.

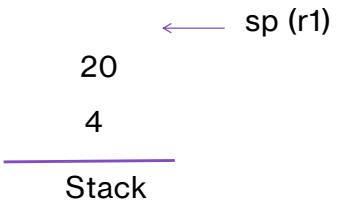
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	42
r1	182
r2	82
r3	-6



We are in double! Time to push its stack frame by pushing the return address into the stack.

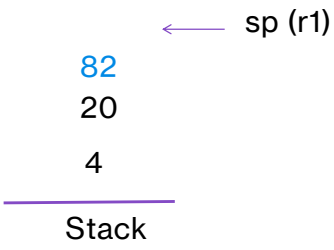
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	42
r1	180
r2	82
r3	-6



This will result in 82 being at the top of the stack and stack pointer is going up by a word which decreases r1 by 2 bytes,

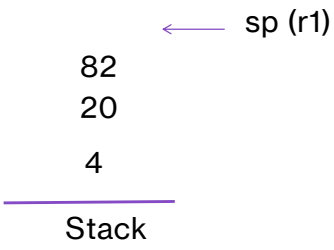
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	46
r1	180
r2	82
r3	-12



Next, add r3 r3 r3 is equivalent to r3*=2, which is -6*2=12. So the result of double is saved in r3

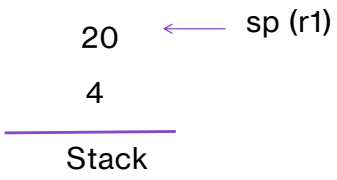
add_then_double.a51

```
double
42     psh r2           ; save the return address
46     add r3 r3 r3     ; double the input2
48     pop r2           ; get the return address
52     jmp r2           ; return
…
add_then_double
…
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	48
r1	182
r2	82
r3	-12



We next pop the return address and store it into r2. That doesn't really change things, r2 remains 82 but the stack frame has been popped.

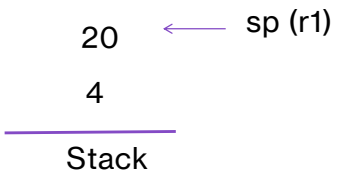
add_then_double.a51

```
double
42     psh r2           ; save the return address
46     add r3 r3 r3     ; double the input2
48     pop r2           ; get the return address
52     jmp r2           ; return
…
add_then_double
…
66 lcw r2 add       ; r3 has input one, push input2 on stack
70 cal r2 r2        ; call add
72 pop r0           ; remove input2 from stack

76 lcw r2 double    ; result of add is in r3
80 cal r2 r2        ; call double

82 pop r2           ; get the return address
86 jmp r2           ; return
```

ic	52
r1	182
r2	82
r3	-12



Finally jmp r2 will take us back to add_then_double which called double.

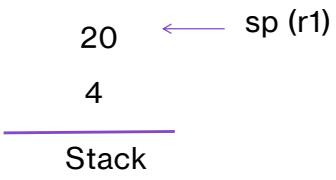
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	82
r1	182
r2	82
r3	-12



We get rid of 20 which is at the top of the stack and put it in r2

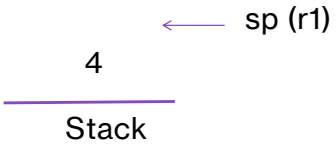
add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	82
r1	184
r2	20
r3	-12



This results in the sp going down and r1 increases by 2.

add_then_double.a51

```
double
42    psh r2          ; save the return address
46    add r3 r3 r3    ; double the input2
48    pop r2          ; get the return address
52    jmp r2          ; return
…
add_then_double
…
66 lcw r2 add      ; r3 has input one, push input2 on stack
70 cal r2 r2       ; call add
72 pop r0          ; remove input2 from stack

76 lcw r2 double   ; result of add is in r3
80 cal r2 r2       ; call double

82 pop r2          ; get the return address
86 jmp r2          ; return
```

ic	86
r1	184
r2	20
r3	-12



Next, we have jmp r2 which returns us to the address 20

add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

ic	20
r1	184
r2	20
r3	-12



This was the instruction right after we called add_then_double.

add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

ic	20
r1	186
r2	20
r3	-12



We then discard the number 4 from the stack so that we keep it clean. sp gets reset back to where it started, that is address 186

add_then_double.a51

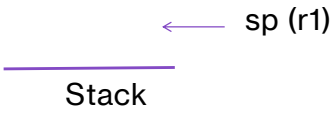
```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

ic	24
r1	186
r2	20
r3	-12

-12 ☺



We are then ready to print the result -12 to the user

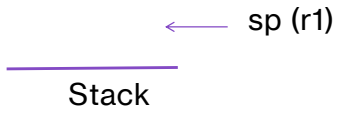
add_then_double.a51

```
2   lcw r1 stack    ; setup the stack
6   loa r3 r0       ; get input1
8   loa r2 r0       ; get input2

10  psh r2          ; save input2 on the stack
14  lcw r2 add_then_double ; setup call to add_then_double
18  cal r2 r2       ; call add_then_double
20  pop r0          ; remove input2 from stack

24  sto r3 r0       ; print out the result
26  hlt
```

ic	26
r1	186
r2	20
r3	-12



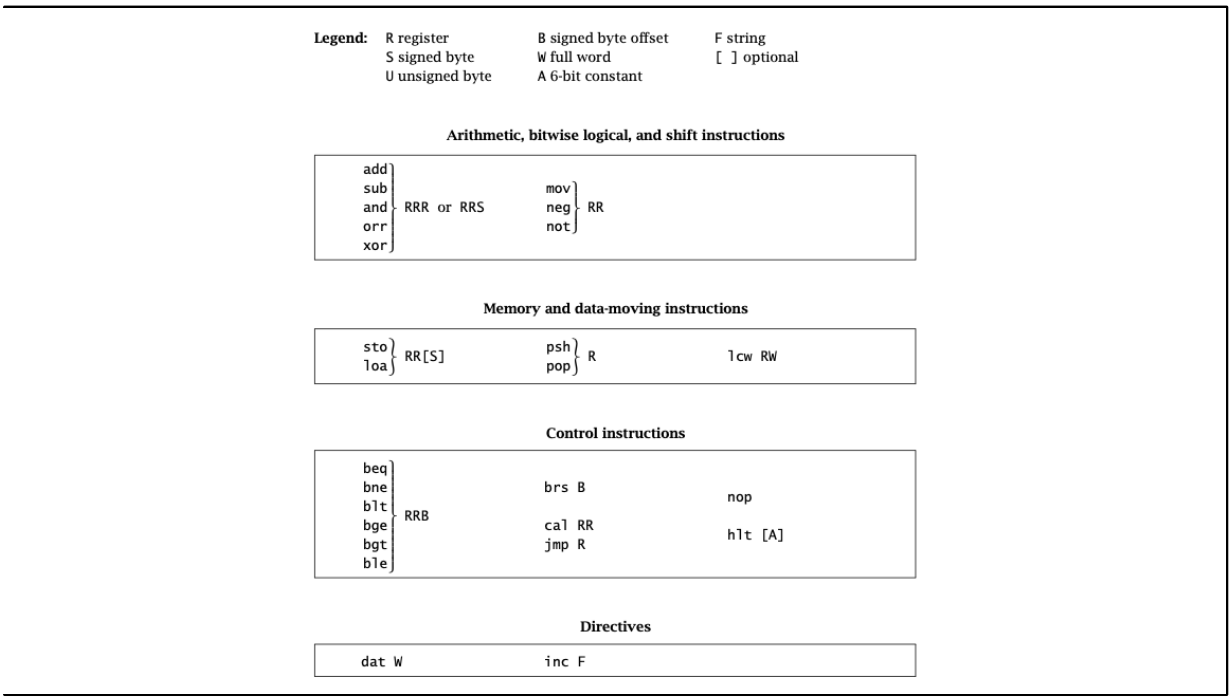
and terminate. So all in all, we saw all the steps that happen when someone calls add_then_double(-10,4) -> =12

To the simulator!



look at add_then_double.a51 code

We can next see what that looks like in practice on the simulator



Here's a summary of all the instructions we learned about for the CS51machine assembly. You can see that there a few more instructions beyond the scope of what we will cover in this class.