CS51 MACHINE: CALLING FUNCTIONS

David Kauchak
CS 51 – Spring 2026

Admin

Assignment 4 due today

Assignment 5 out early next week (CS51 machine)

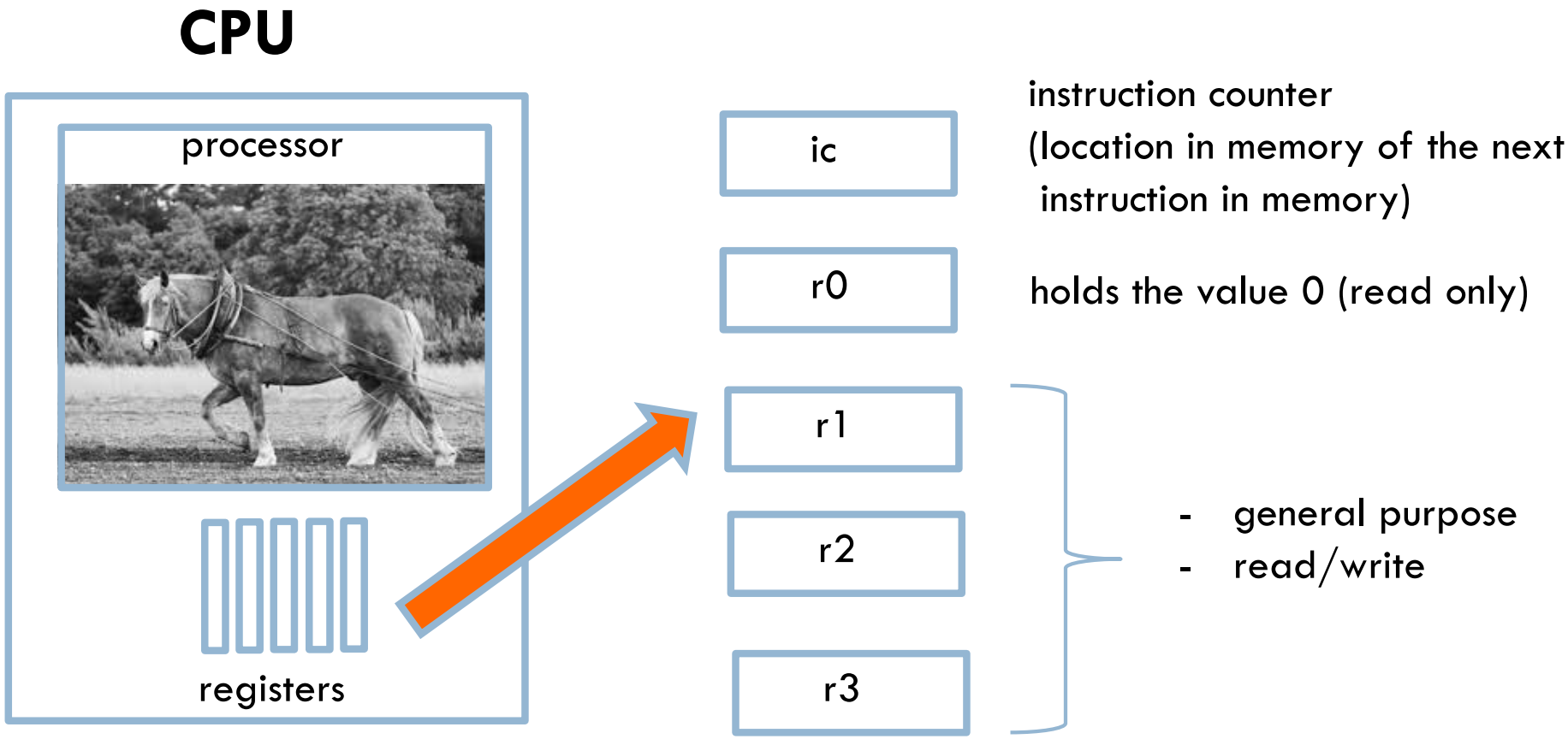
Checkpoint 1 on Tuesday

Lab tomorrow: review – attendance optional

Examples from this lecture

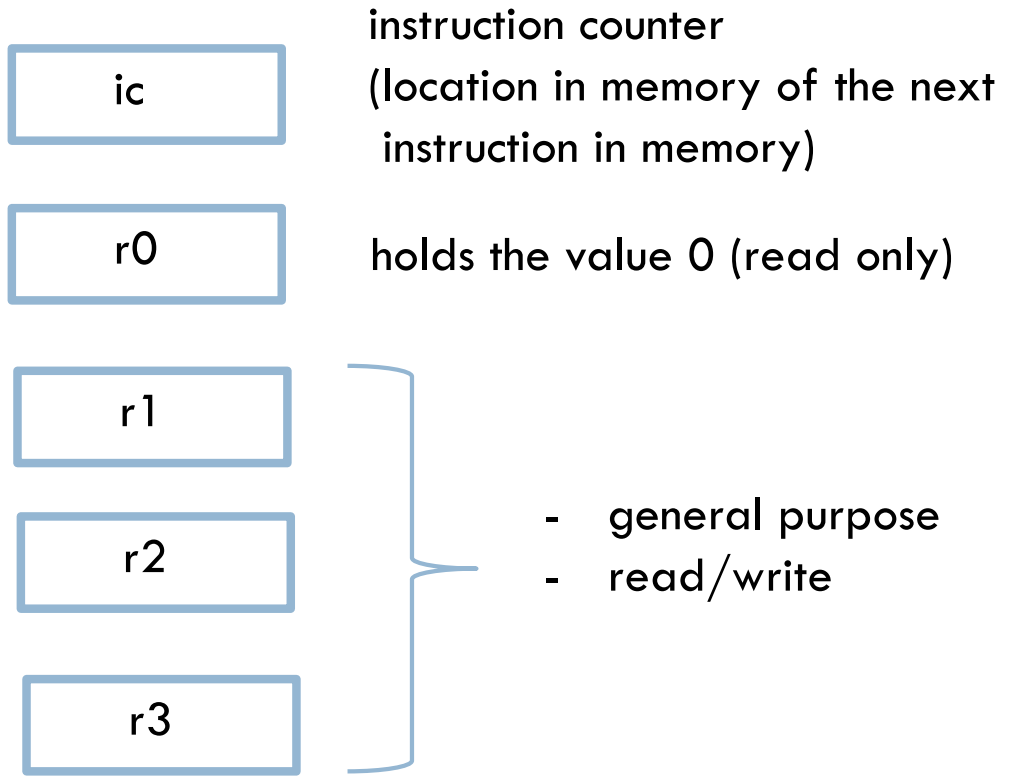
http://www.cs.pomona.edu/classes/cs51/cs51machine/

CS51 machine



When executing a program, the CS51 machine loops over the follow:

- Fetch the value from $\mathrm{mem}[\mathrm{ic}]$ for use as an instruction
- Increment ic by 2
- Decode the instruction and then execute it

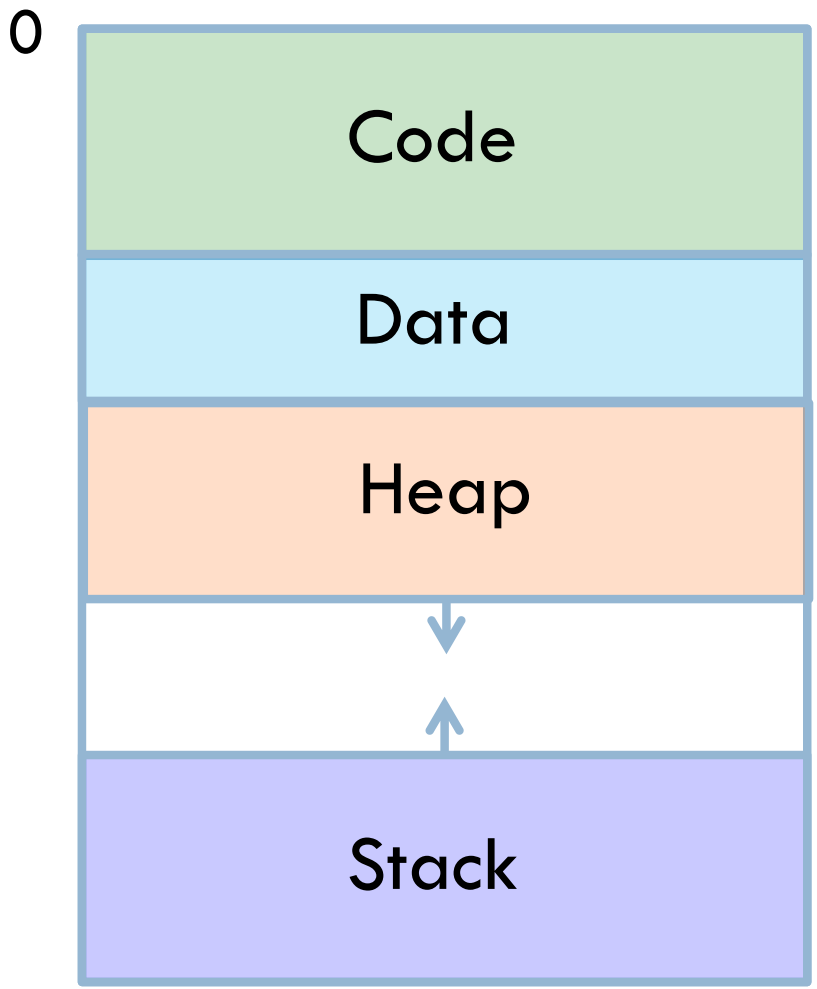


Basic structure of CS51 program

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

- whitespace before operations/instructions
- labels go here

Memory layout



Global data, e.g., variables that are created outside a function

Where dynamically allocated program data is stored

Where program/function execution information is stored, parameters, and local variables

Stacks

Two operations

- push: add a value in the register to the top of the stack
- pop: remove a value from the top of the stack and put it in the register

---

For example:

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

What will be printed out?

Stacks

Two operations

- push: add a value in the register to the top of the stack
- pop: remove a value from the top of the stack and put it in the register

For example:

```
add r3 r0 8          ; r3 = 8
psh r3               ; push r3 (8) onto the stack
add r3 r0 0          ; r3 = 0
pop r3               ; r3 get top value of stack (8)
sto r3 r0            ; print out 8
```

Stack frame

Key unit for keeping track of a function call

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

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

What is add_then_double(-10, 4)?

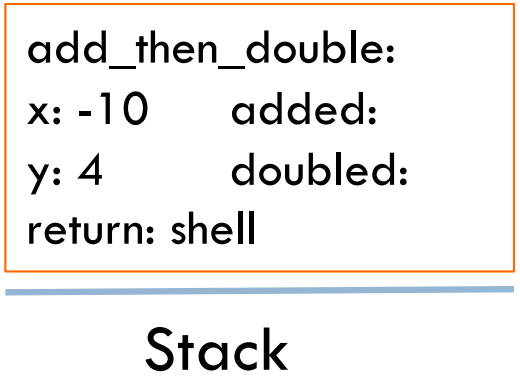
Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

When you call a function a new stack frame is created:

- return address (where should we go when the function finishes)
- parameters
- any local variables

add_then_double(-10, 4)



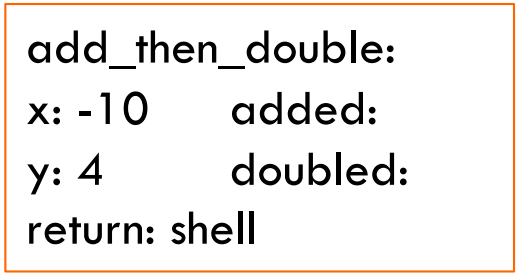
Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

How do we evaluate this?

add_then_double(-10, 4)



Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

Make another function call

add_then_double(-10, 4)

add_then_double:
x: -10 added:
y: 4 doubled:
return: shell

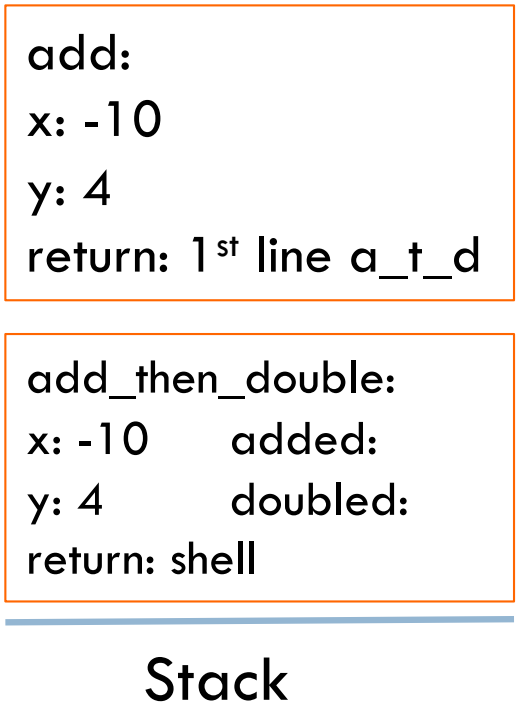
Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

add(-10, 4)

add_then_double(-10, 4)



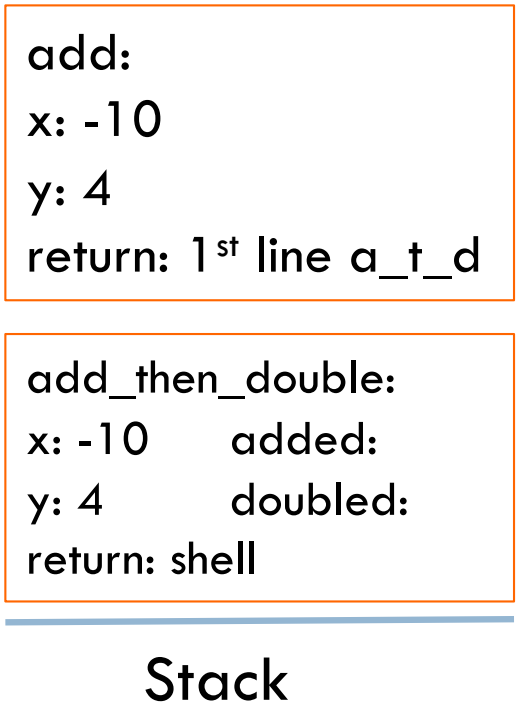
Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

How do we evaluate this?

add(-10, 4)

add_then_double(-10, 4)



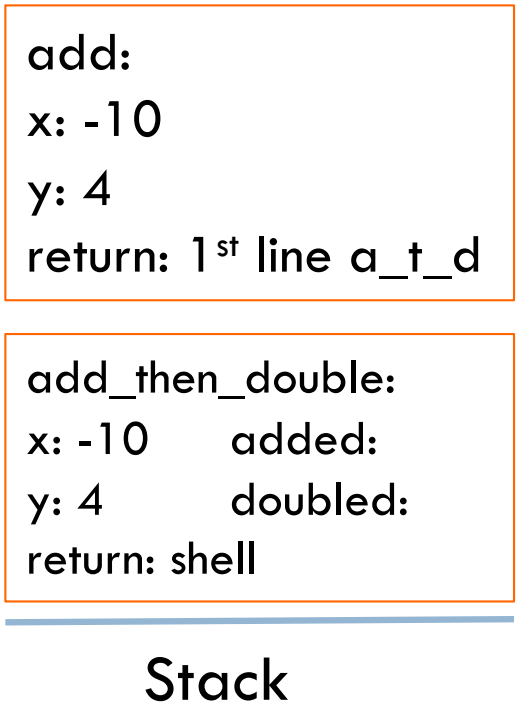
Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

Evaluate arguments and make another function call

add(-10, 4)

add_then_double(-10, 4)



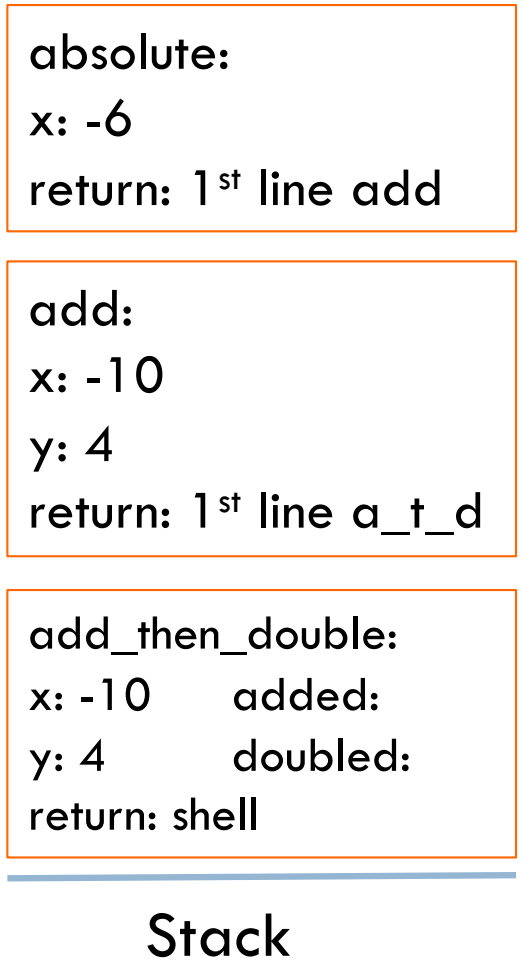
Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

absolute(-6)

add(-10, 4)

add_then_double(-10, 4)



Stack frames

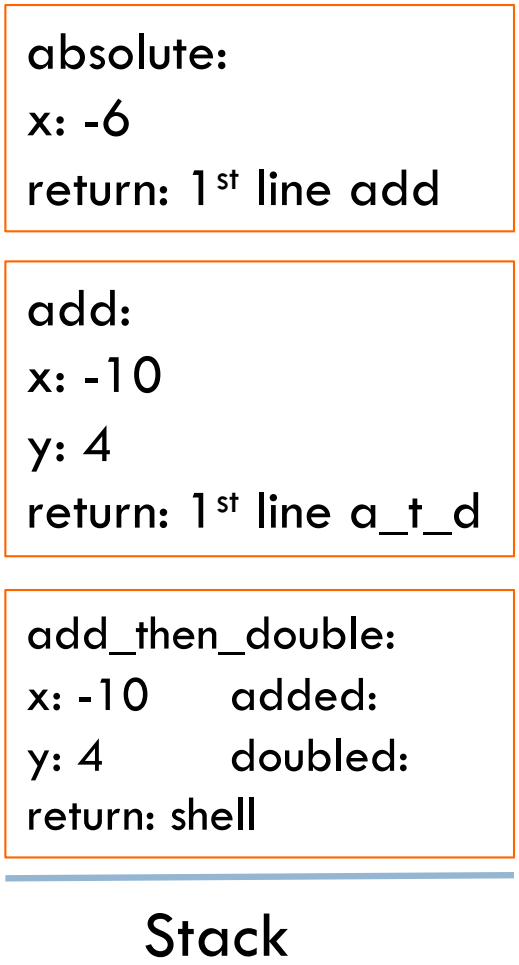
```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

What now?

absolute(-6)

add(-10, 4)

add_then_double(-10, 4)



Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x
```

6

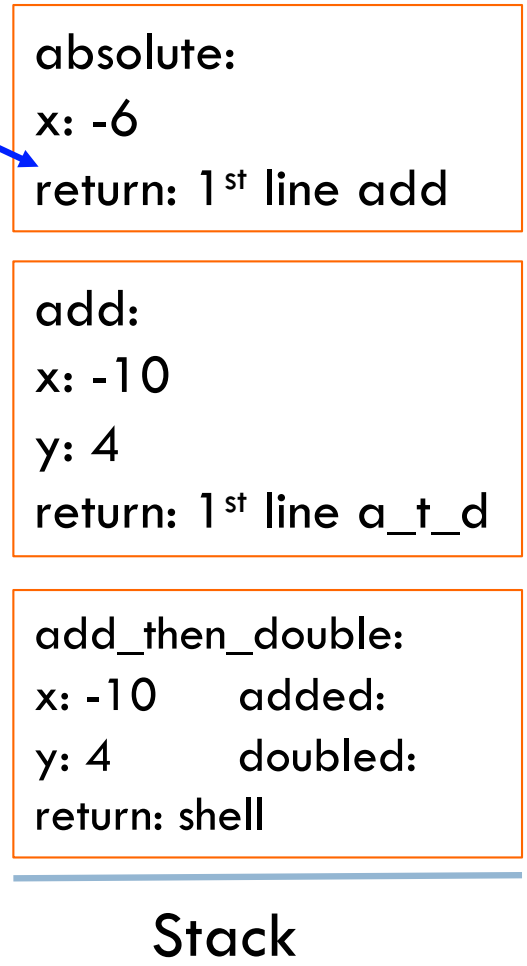
When a function finishes:
return to where it was called from (return address)

```
def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

absolute(-6)

add(-10, 4)

add_then_double(-10, 4)



Stack frames

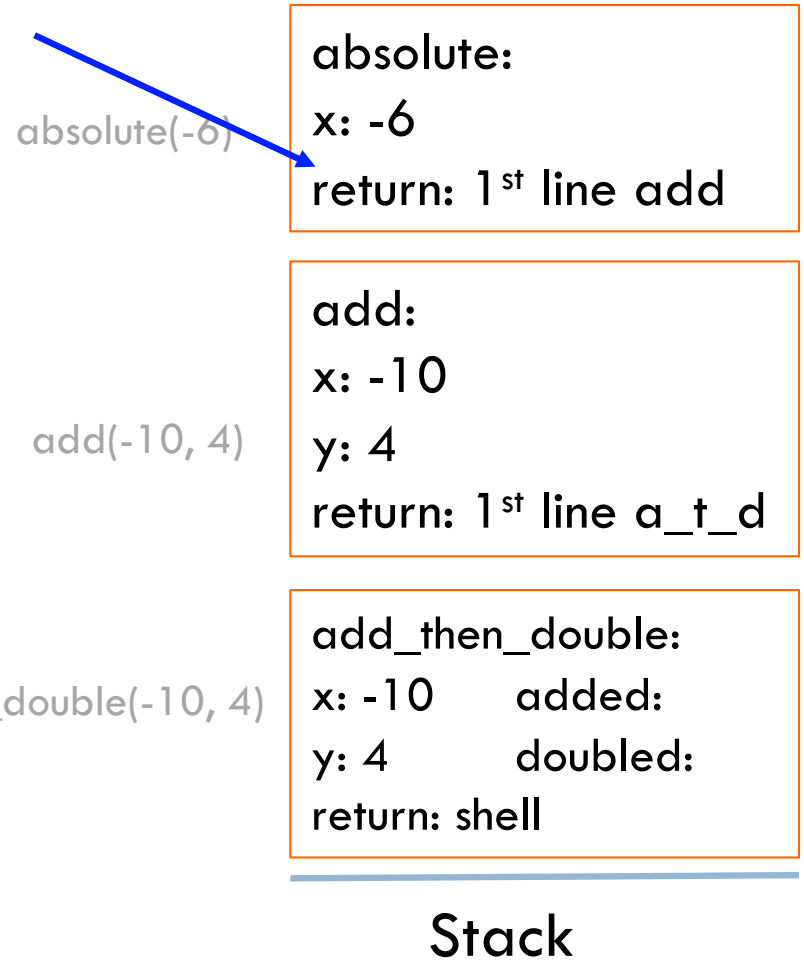
```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

We remove (pop) the stack frame from the call stack

absolute(-6)

add(-10, 4)

add_then_double(-10, 4)



Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

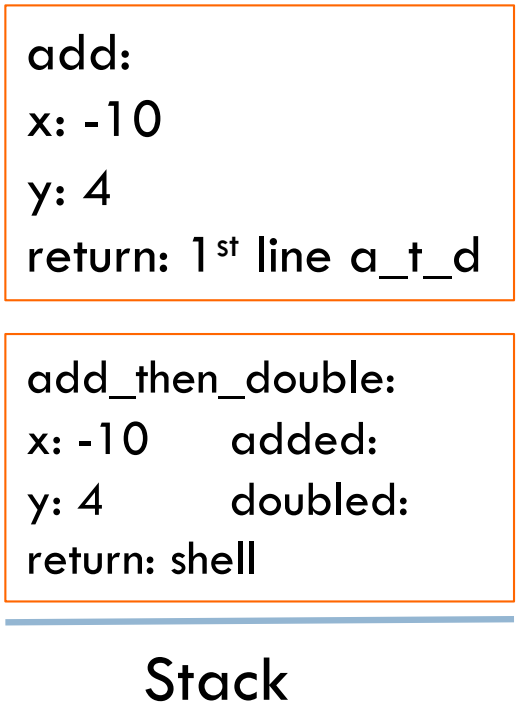
We remove (pop) the stack frame from the call stack

Continue executing where the function was called

6

add(-10, 4)

add_then_double(-10, 4)



Stack frames

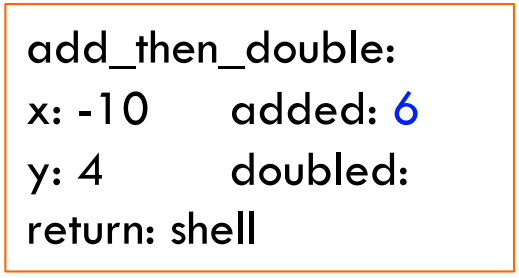
```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

We remove (pop) the stack frame from the call stack

Continue executing where the function was called

6

add_then_double(-10, 4)

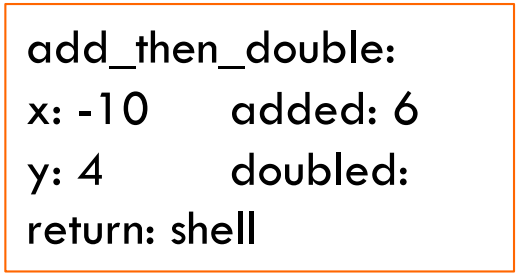


Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

add_then_double(-10, 4)



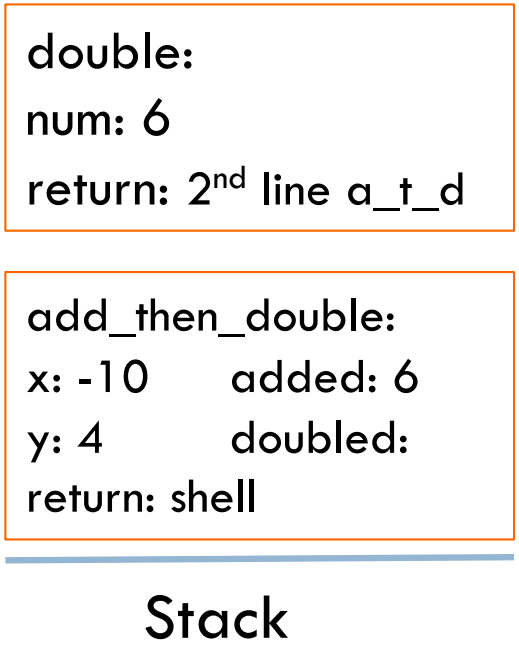
Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

double(6)

add_then_double(-10, 4)



Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

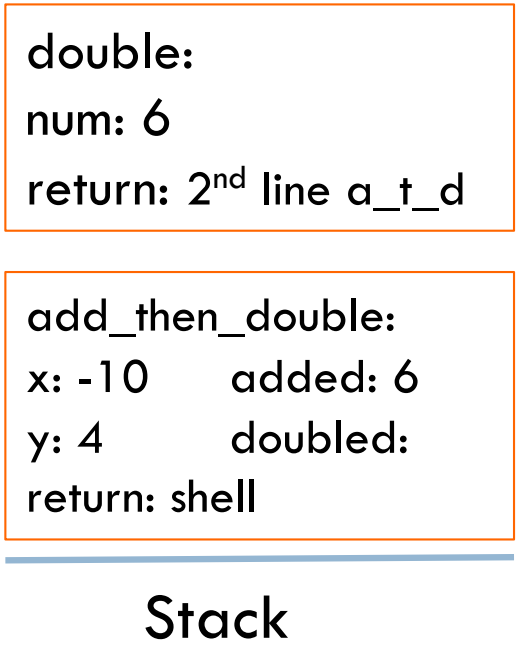
We remove (pop) the stack frame from the call stack

Continue executing where the function was called

12

double(6)

add_then_double(-10, 4)



Stack frames

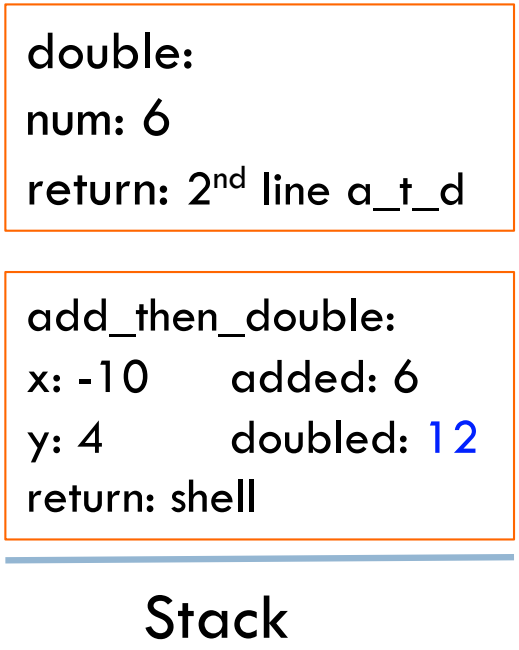
```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

We remove (pop) the stack frame from the call stack

Continue executing where the function was called

add_then_double(-10, 4)

12



Stack frames

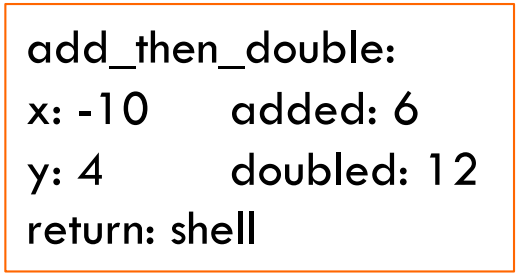
```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

12

We remove (pop) the stack frame from the call stack

Continue executing where the function was called

add_then_double(-10, 4)



Stack

Stack frames

```
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```

We remove (pop) the stack frame from the call stack

Continue executing where the function was called

Answer: 12

Stack

Function calls in assembly

For high-level languages the stack is managed for you

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

Stack

CS51 function call conventions

r1 is reserved for the stack pointer

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

additional parameters go on the stack (more on this)

the result should go in **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
```

What do you think jmp does?

conventions:

- 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

Structure of a single parameter function

```
fname
        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
```

"Jumps" to the line of code at r2

Really: sets ic = r2

conventions:

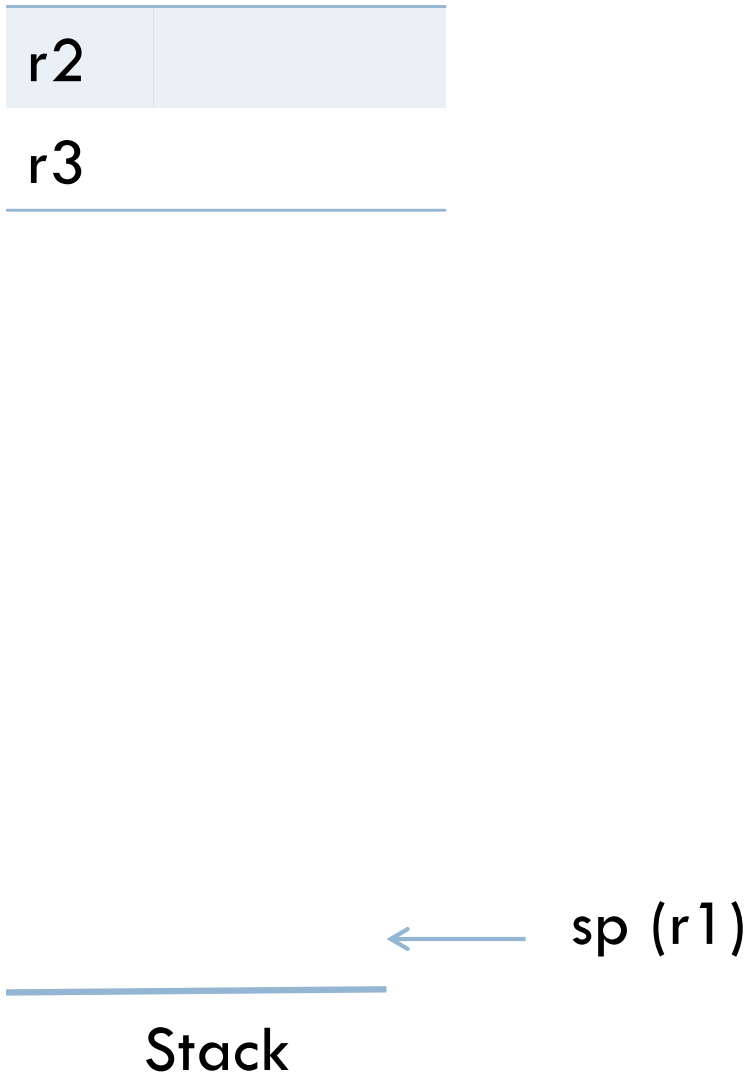
- r2 has the return address
- argument is in r3
- r1 is off-limits since it's used for the stack pointer
- return value goes in r3

Our first function call

```
     loa r3 r0          ; get input from user for input parameter

     lcw r2 increment   ; call increment
     cal r2 r2

     sto r3 r0          ; write result,
     hlt                ;    and halt

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

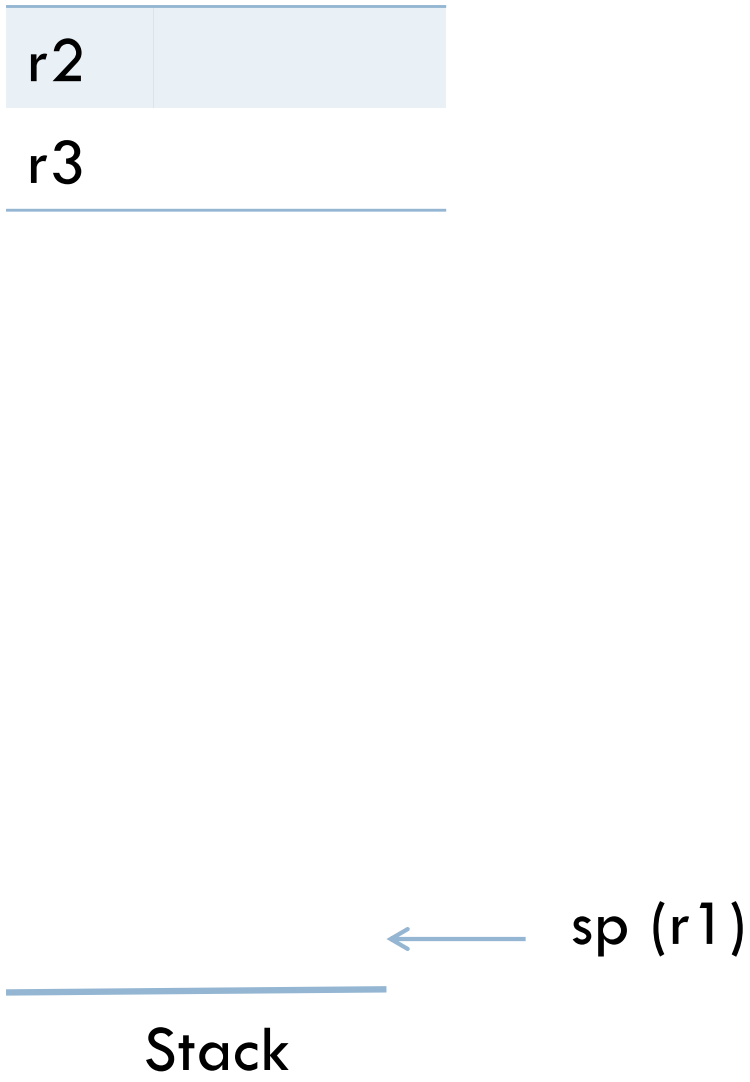
Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```



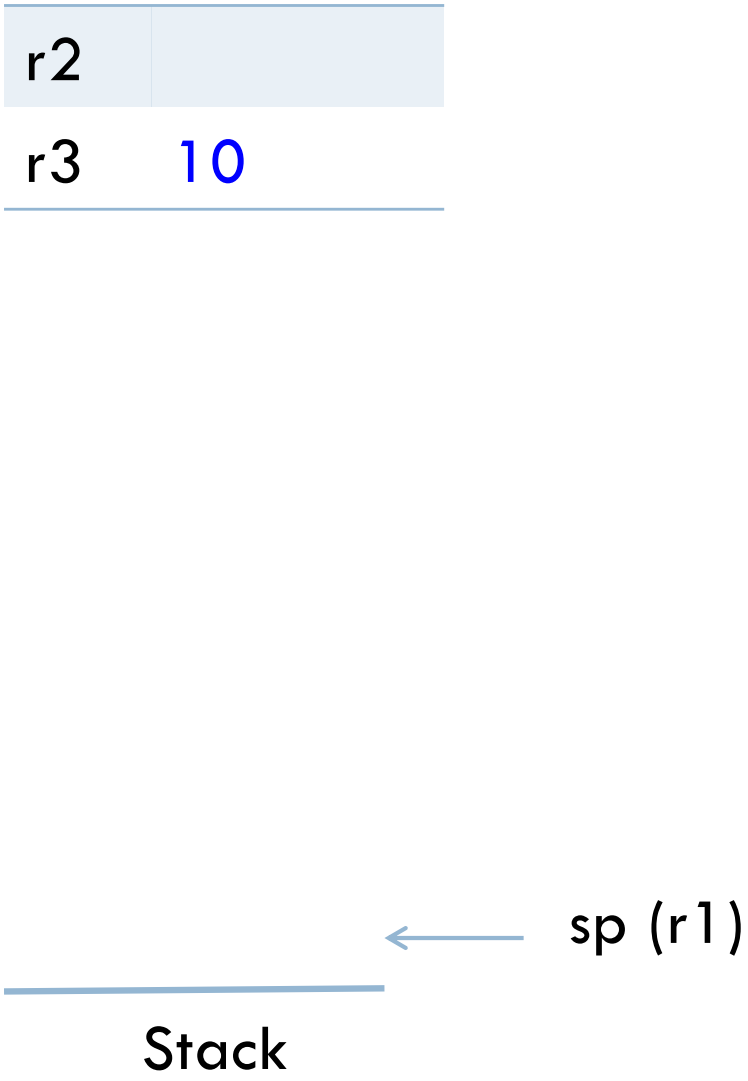
Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

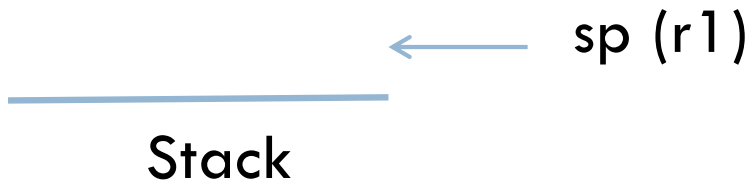


Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r3 r0
    hlt

increment
    psh r2
    add r3 r3 1
    pop r2
    jmp r2
```

r2	
r3	10

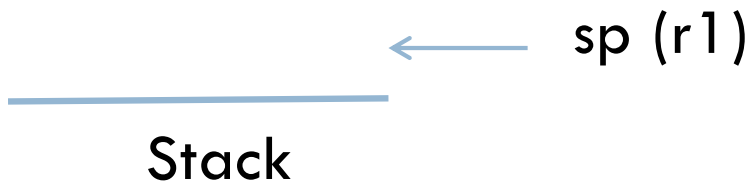
lcw: put the memory address of the label into the register



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

r2	increment
r3	10



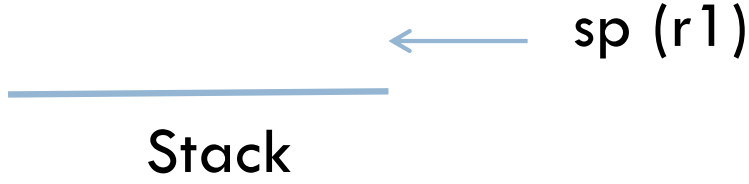
Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

r2	increment
r3	10

cal: call a function

- which function to call
- where should the return address go



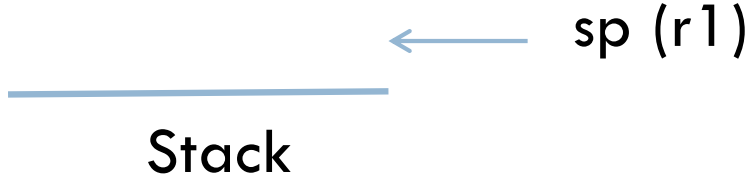
Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

r2	increment
r3	10

cal:

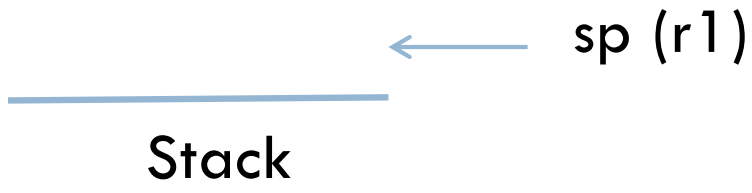
1. Go to instruction address in r2 (2nd r2)
2. Save current ic into r2 (i.e. the address of the ***next*** instruction that would have been executed)



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

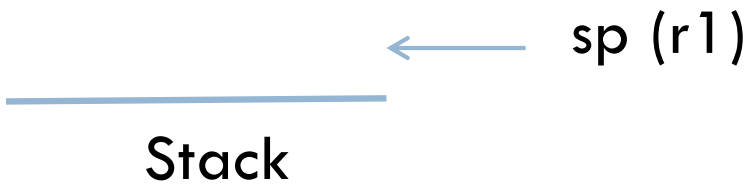
r2	loc: sto
r3	10



Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r3 r0
    hlt

increment
    psh r2
    add r3 r3 1
    pop r2
    jmp r2
```

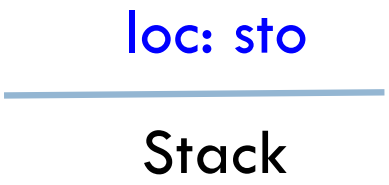
r2	loc: sto
r3	10



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

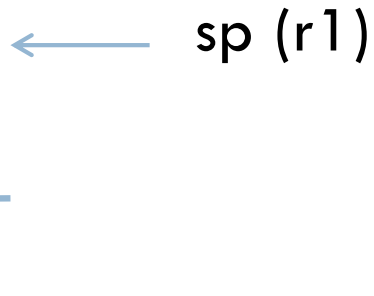
r2	loc: sto
r3	10



Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

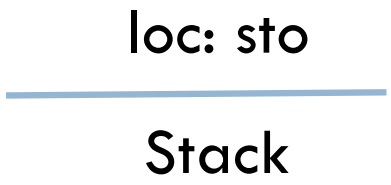
r2	loc: sto
r3	10



Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

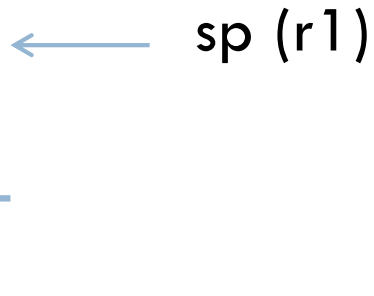
r2	loc: sto
r3	11



Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

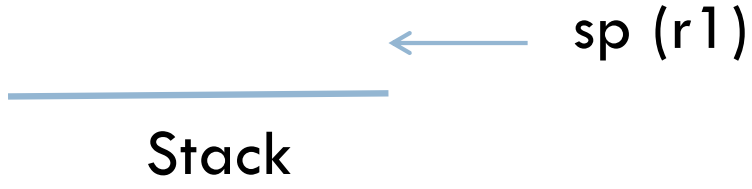
r2	loc: sto
r3	11



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

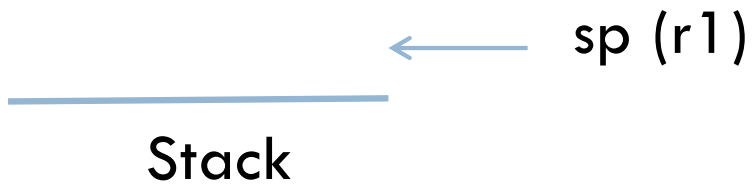
r2	loc: sto
r3	11



Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

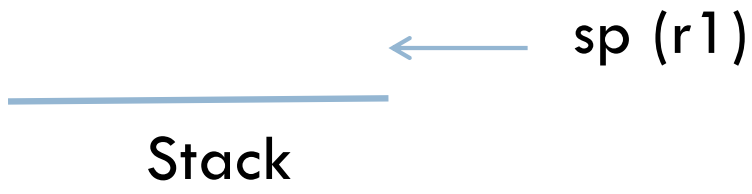
r2	loc: sto
r3	11



Our first function call

```
        loa r3 r0

        lcw r2 increment
        cal r2 r2

        sto r3 r0
        hlt

increment
        psh r2
        add r3 r3 1
        pop r2
        jmp r2
```

r2	loc: sto
r3	11

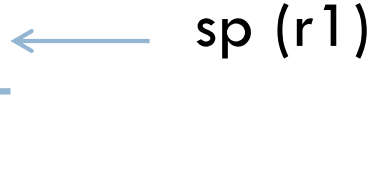


Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

r2	loc: sto
r3	11

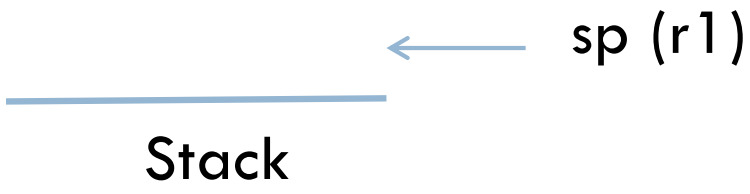
11 ☺



Our first function call

```
      loa r3 r0

      lcw r2 increment
      cal r2 r2

      sto r3 r0
      hlt

increment
      psh r2
      add r3 r3 1
      pop r2
      jmp r2
```

r2	loc: sto
r3	11



Real structure of CS51 program

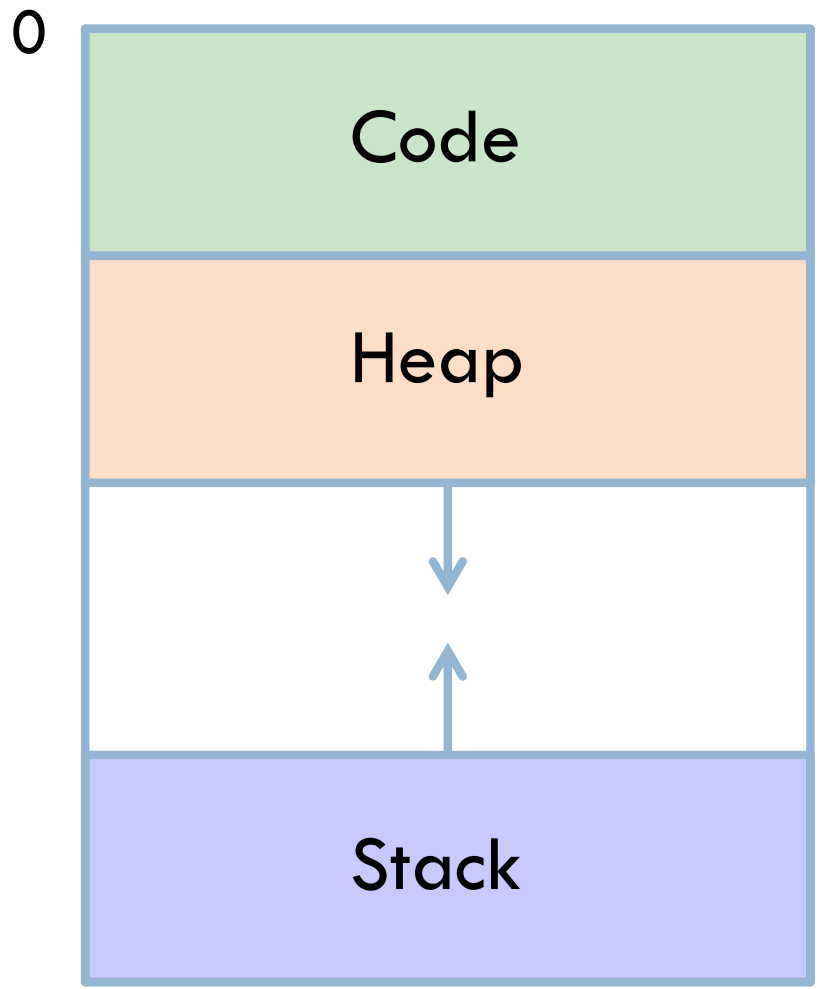
```
; great comments at the top!
;
        lcw r1 stack

        instruction1          ; comment
        instruction2          ; comment
        ...
        hlt


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

Save address of highest end (highest address) of the stack in r1

Reserve 50 words for the stack

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
```



To the simulator!

look at increment.a51 code

Functions with multiple arguments

When 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

What if we want to have functions with multiple parameters?

Functions with multiple arguments

```
fname
        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
```

conventions:

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

Functions with multiple arguments

```
fname
        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*?

Functions with multiple arguments

```
fname
        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
```

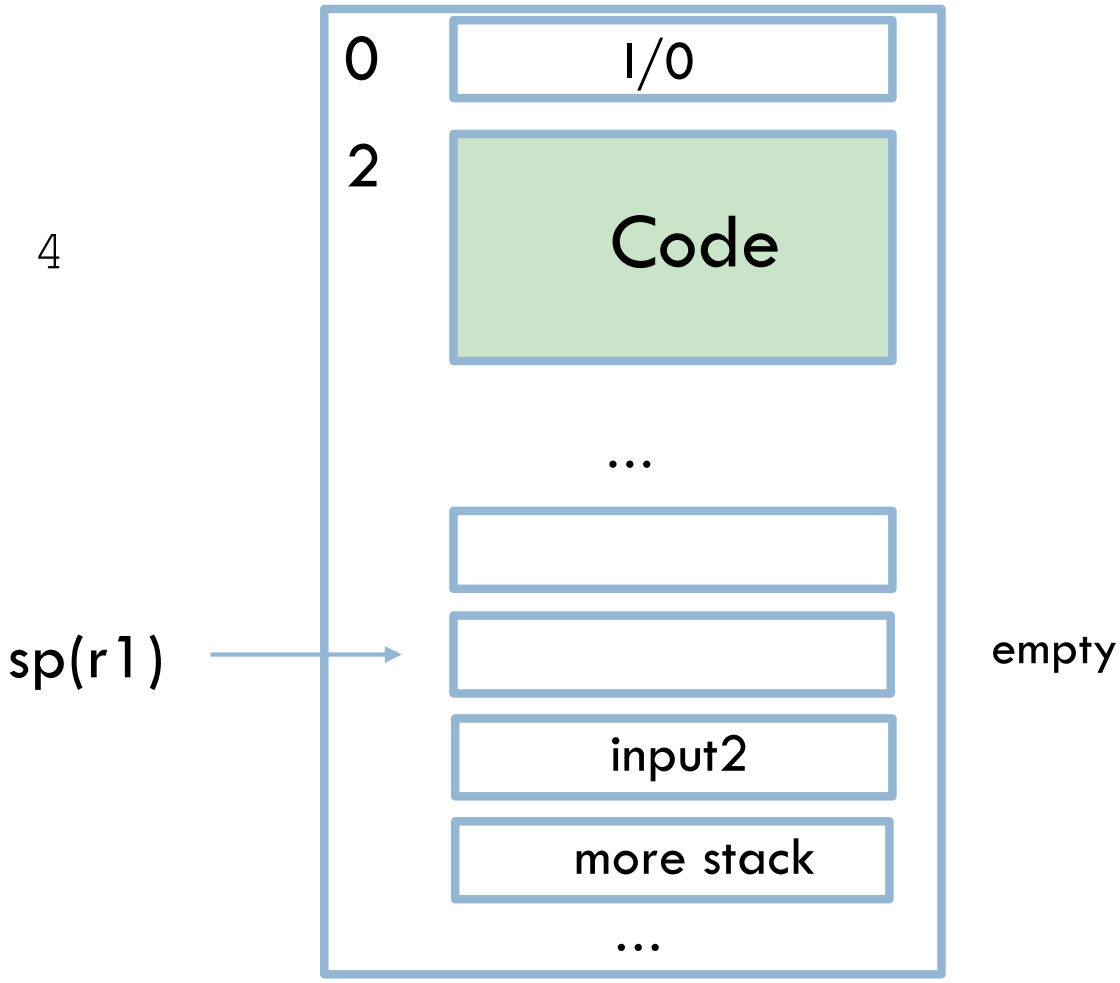
loa R_a R_b: $R_a = mem[R_b]$

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

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

Functions with multiple arguments

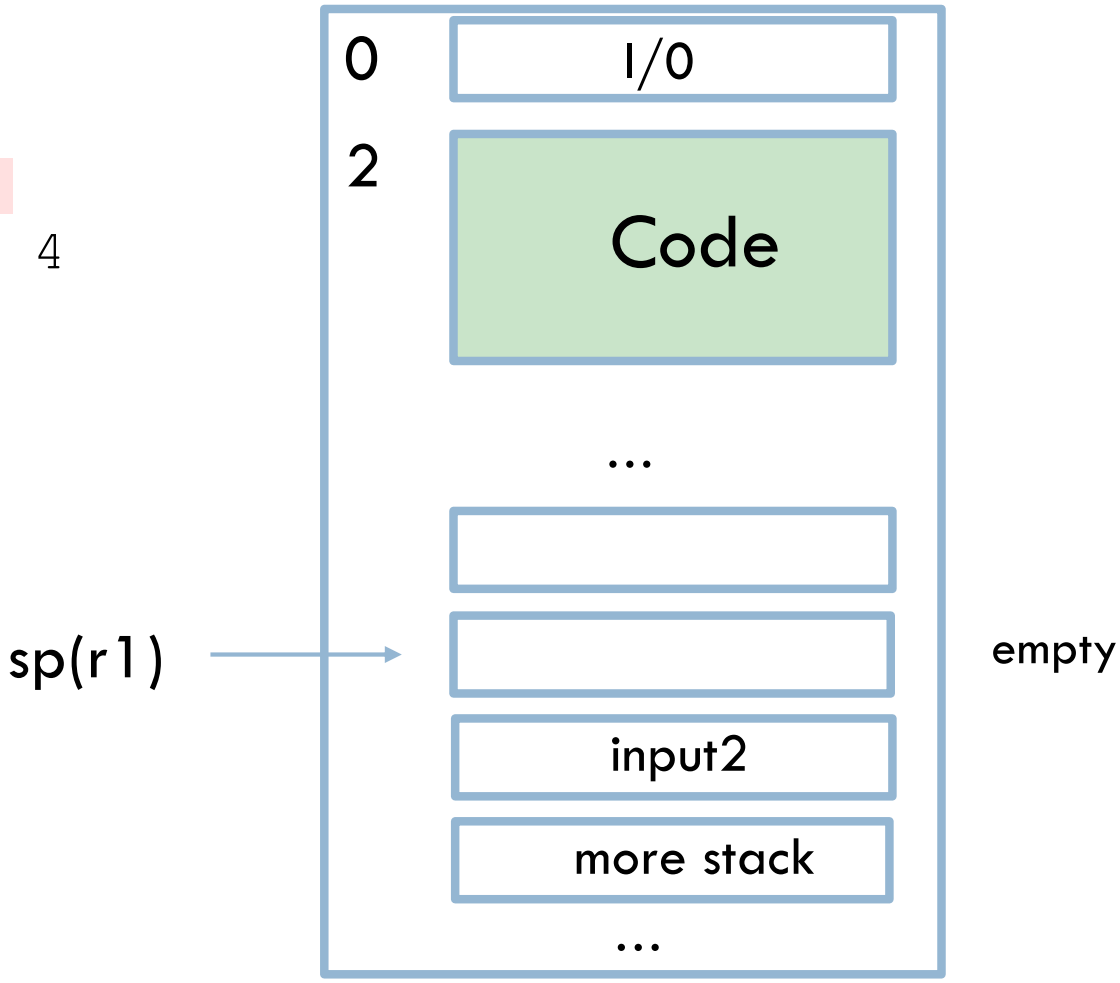
```
fname
    psh r2
    loa r2 r1 4
```



When we call a function, input1 = r3, additional arguments on top of stack

Functions with multiple arguments

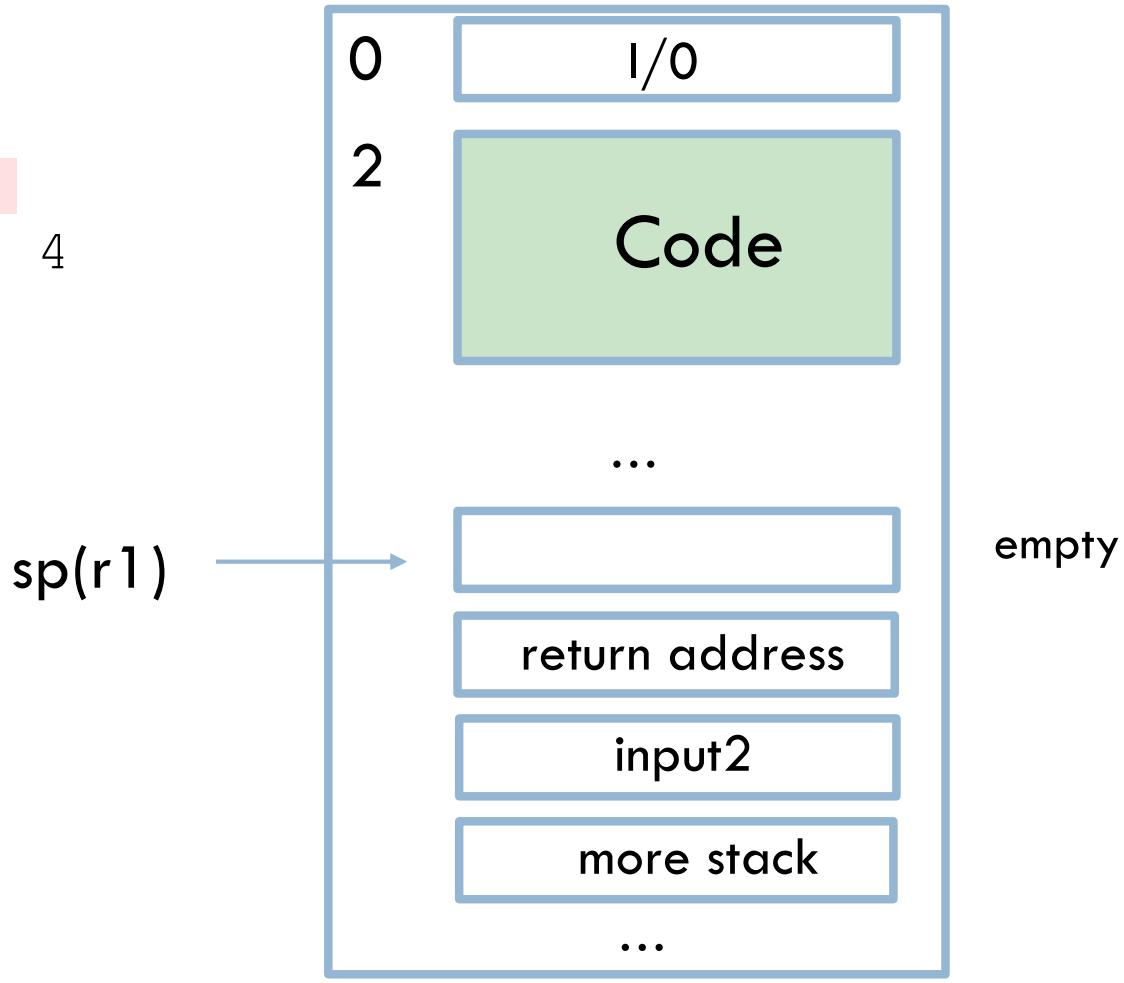
```
fname
    psh r2
    loa r2 r1 4
```



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

Functions with multiple arguments

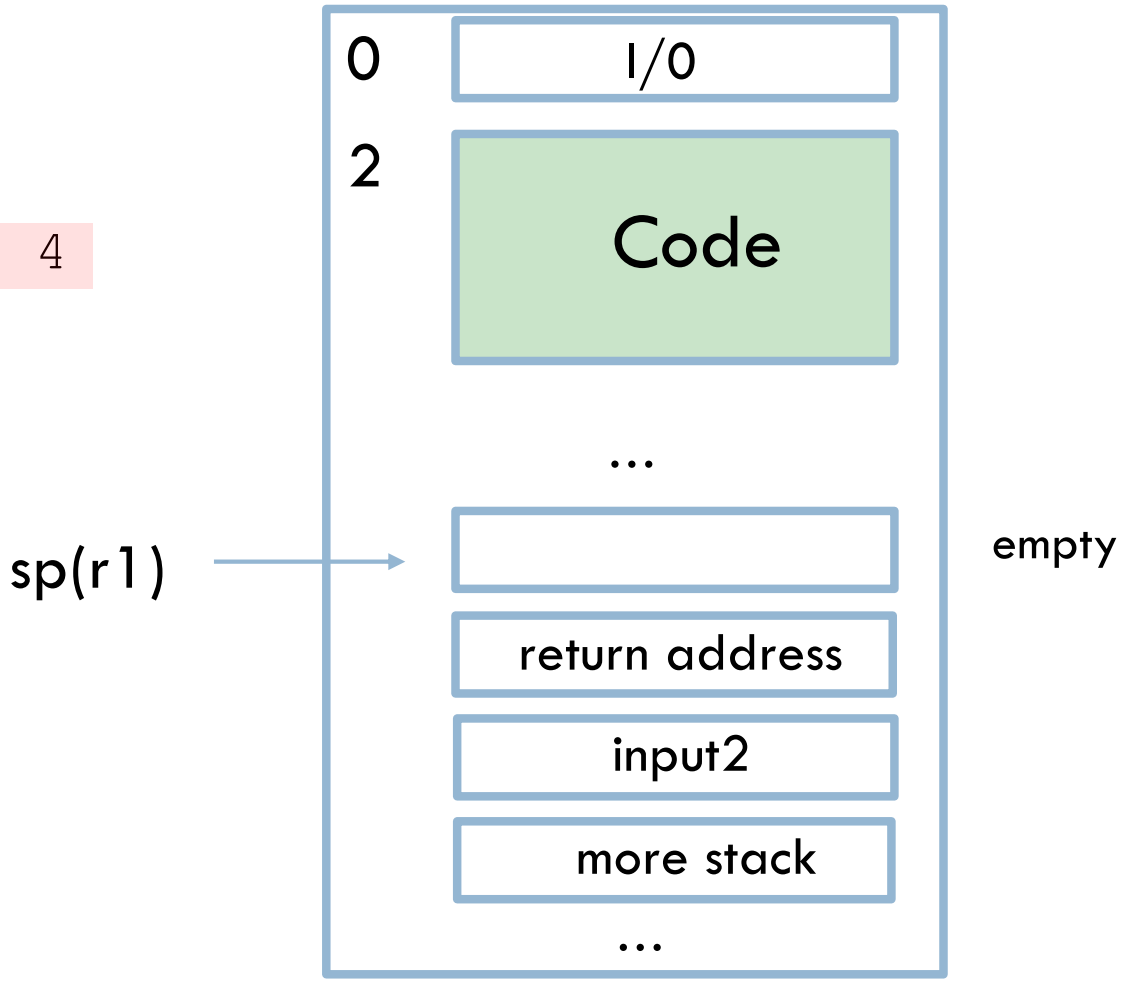
```
fname
    psh r2
    loa r2 r1 4
```



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

Functions with multiple arguments

```
fname
    psh r2
    loa r2 r1 4
```

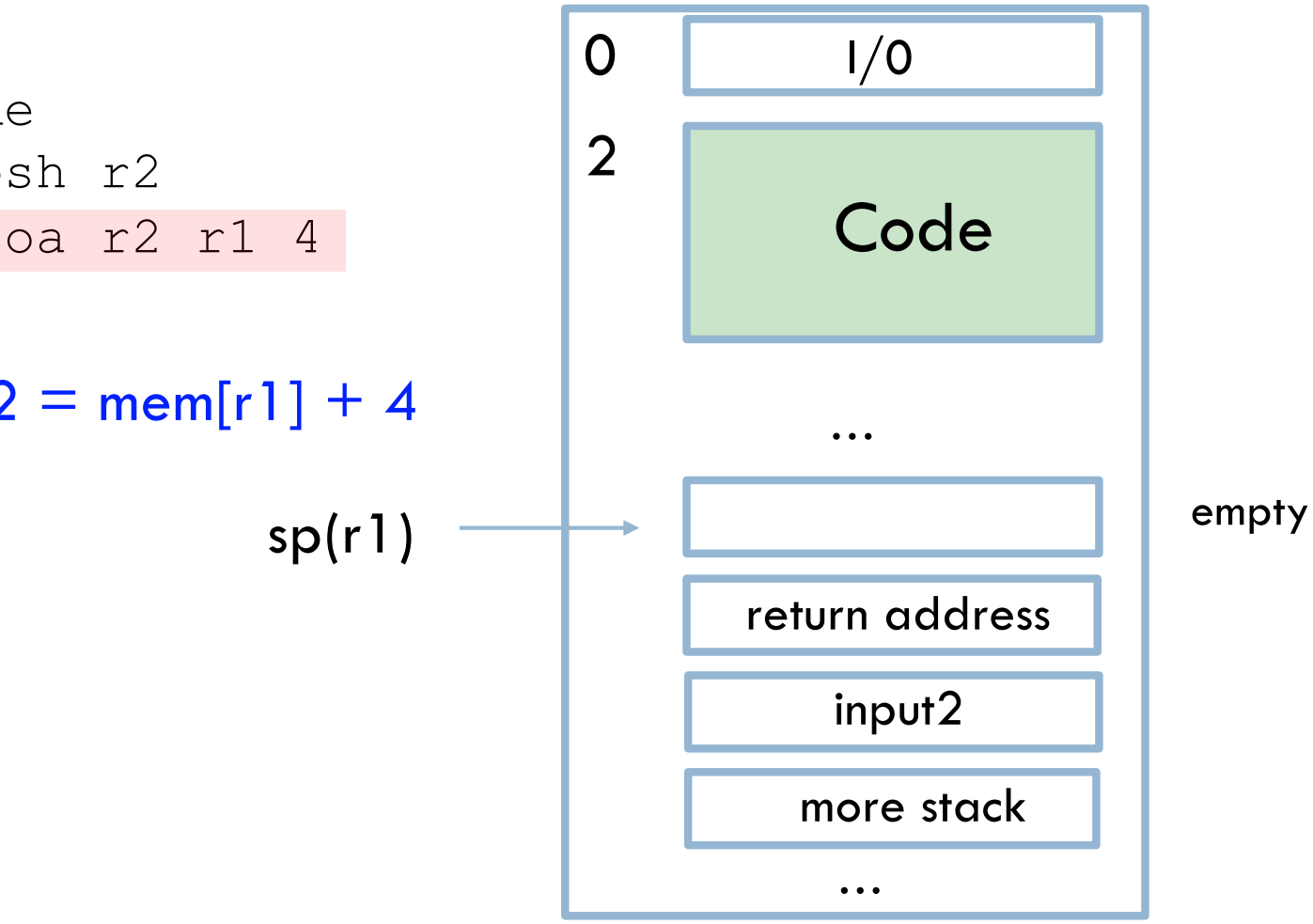


We've freed up r2. How do we get input2 into r2?

Functions with multiple arguments

```
fname
    psh r2
    loa r2 r1 4
```

r2 = mem[r1] + 4



Want to look ahead two "words" from where the stack point is

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

add (add.a51)

notice the symmetry of psh/pop

```
    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              ; setup call to add_then_double
    cal r2 r2               ; call absolute
    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
    pop r2                  ; get the return address
    jmp r2                  ; return (answer is in r3)

    dat 100
stack
```

add (add.a51)

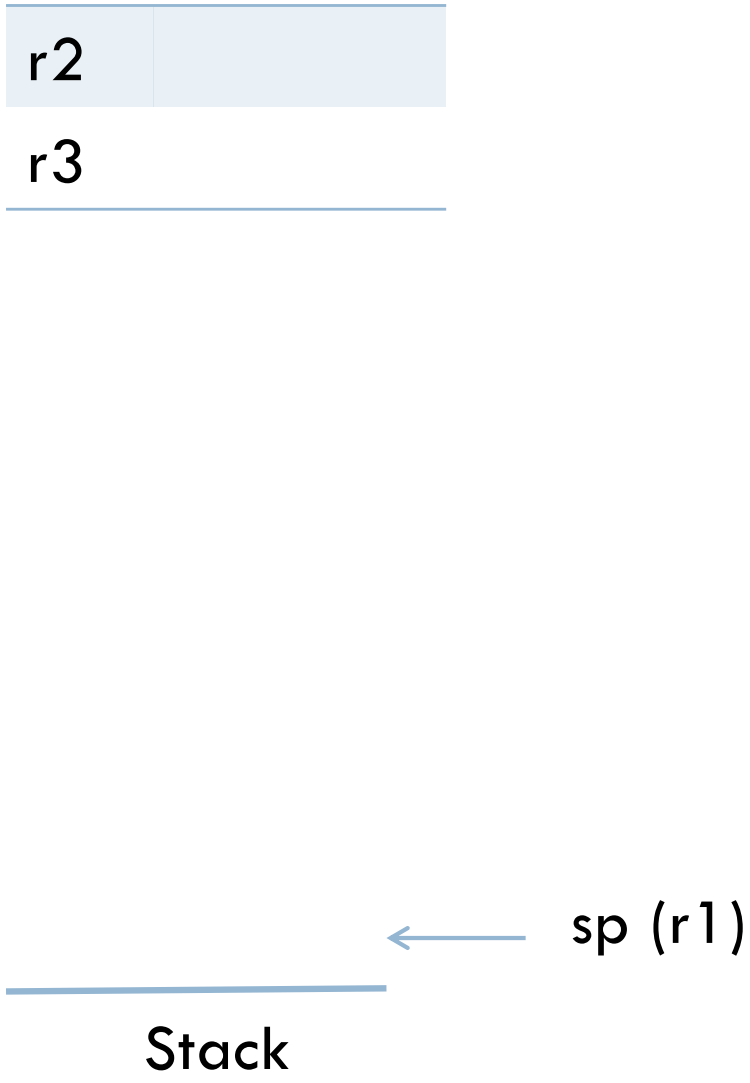
```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

r2	
r3	

add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

Setup the stack

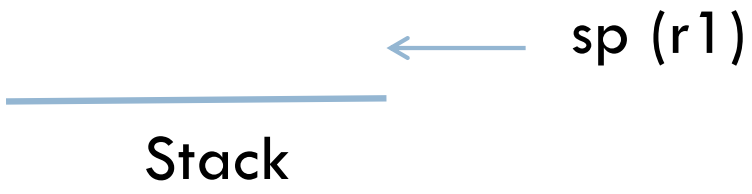


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

Get two inputs

r2	4
r3	10

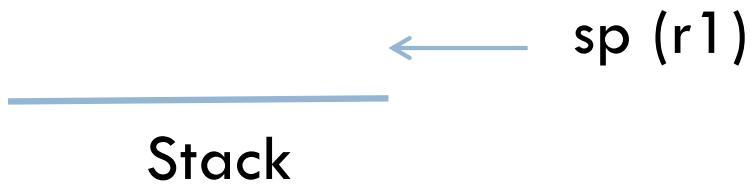


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

Second argument goes on stack

r2	4
r3	10

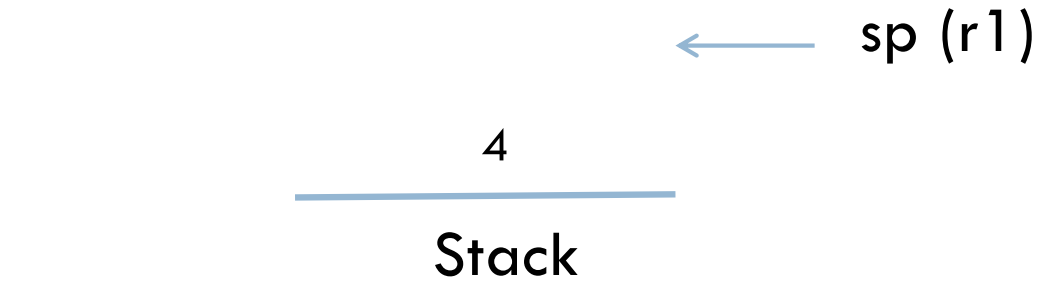


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

Second argument goes on stack

r2	4
r3	10



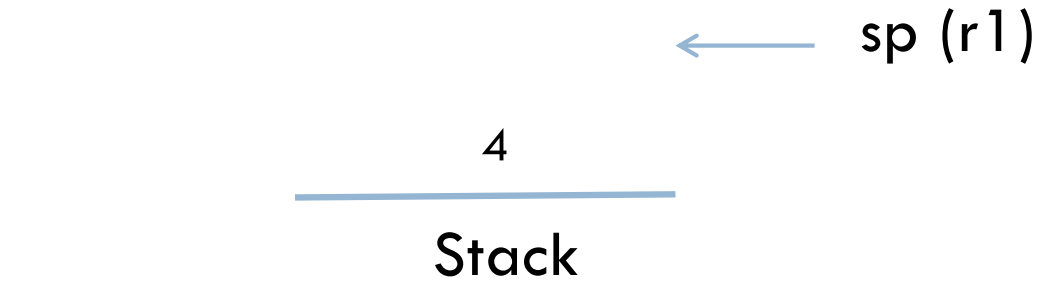
add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

setup call to add

r2	loc: add
r3	10

We would have lost our second input if we didn't use the stack!

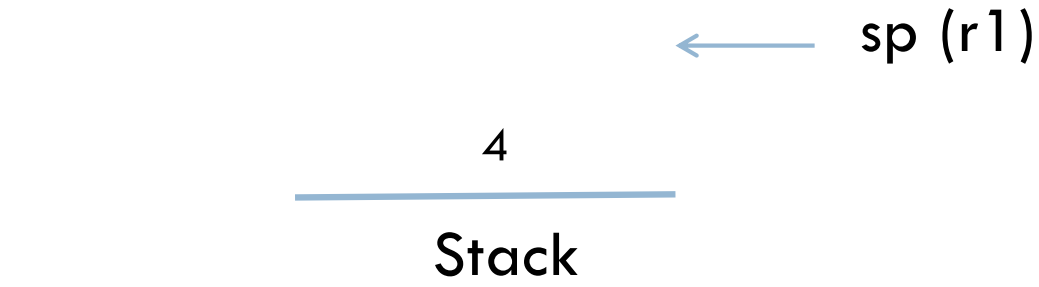


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

r2	loc: add
r3	10

call the function
- jump to location currently in r2
- then, save return address (ic) into r2

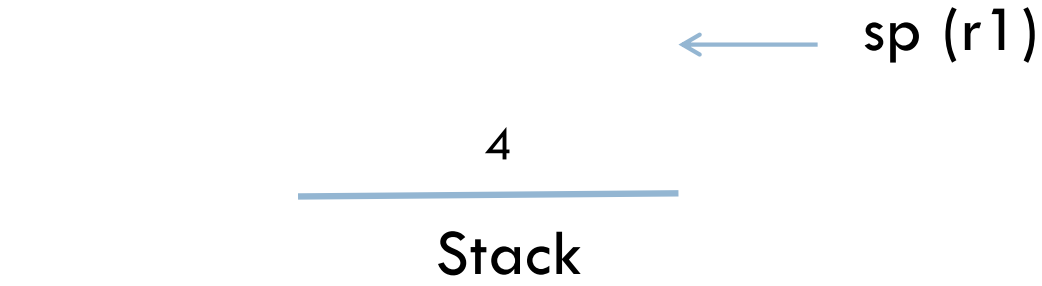


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

r2	loc: pop r0
r3	10

call the function
- jump to location currently in r2
- then, save return address (ic) into r2

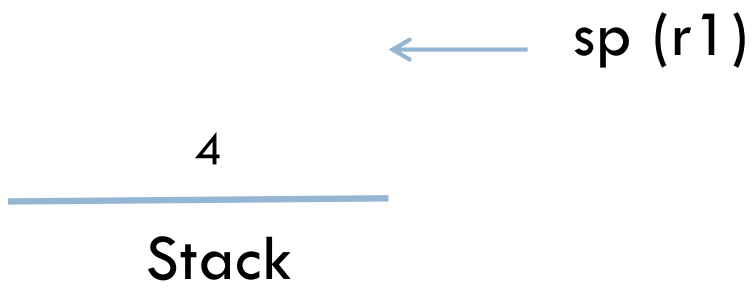


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

save the return address

r2	loc: pop r0
r3	10

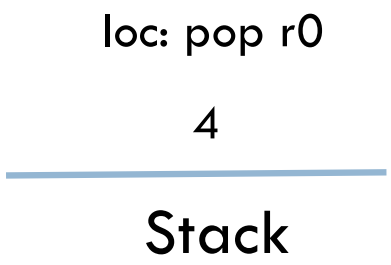


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

save the return address

r2	loc: pop r0
r3	10

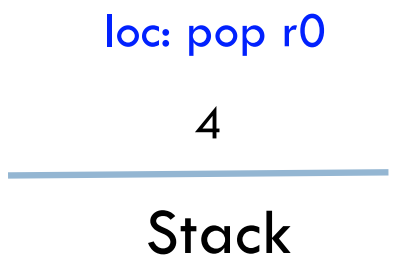


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

save the return address

r2	loc: pop r0
r3	10

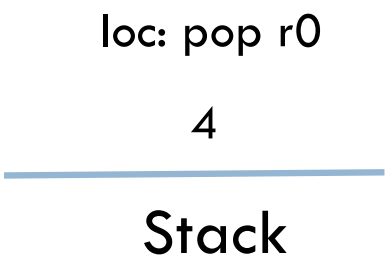


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

get the second input

r2	loc: pop r0
r3	10



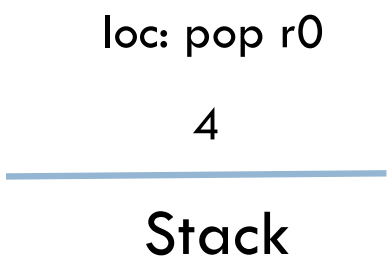
add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

get the second input

r2	4
r3	10

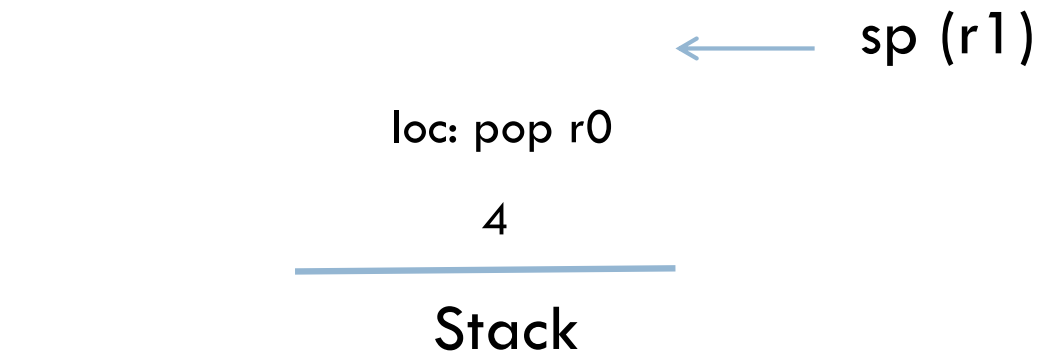
We needed to save the return address to the stack to free up r2



add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

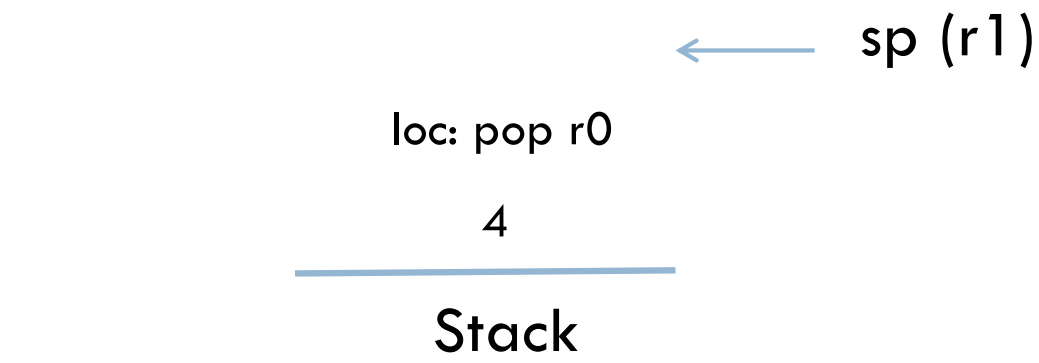
r2	4
r3	10



add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

r2	4
r3	14



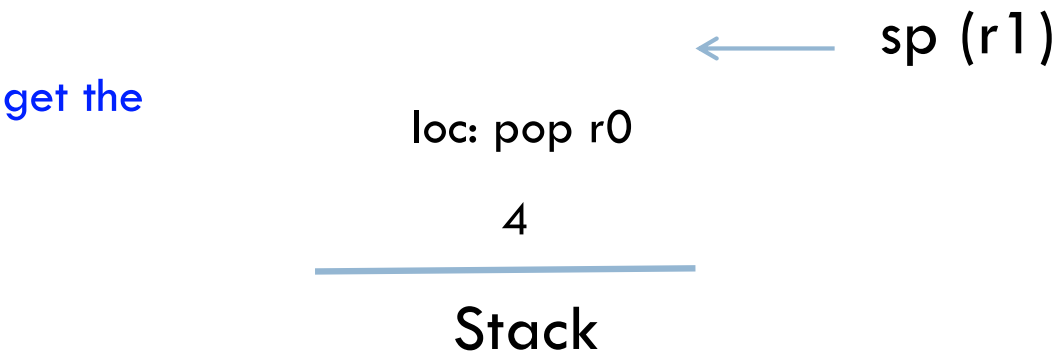
add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

ready to return: get the return address

r2	4
r3	14

Note that the return value is in r3



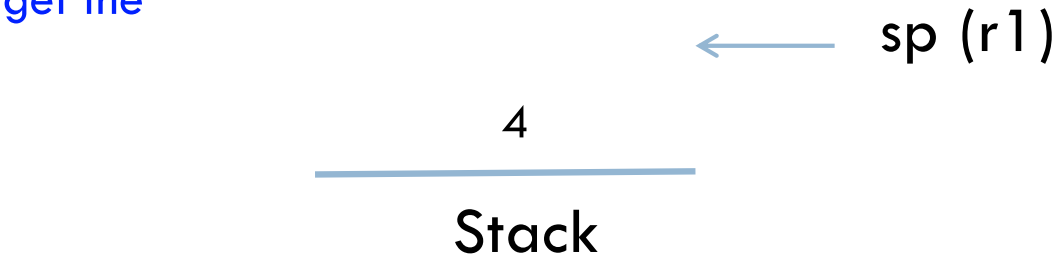
add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

ready to return: get the return address

r2	loc: pop r0
r3	14

Note that the return value is in r3



add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

return

r2	loc: pop r0
r3	14

Note that the return value is in r3

sp (r1)

4

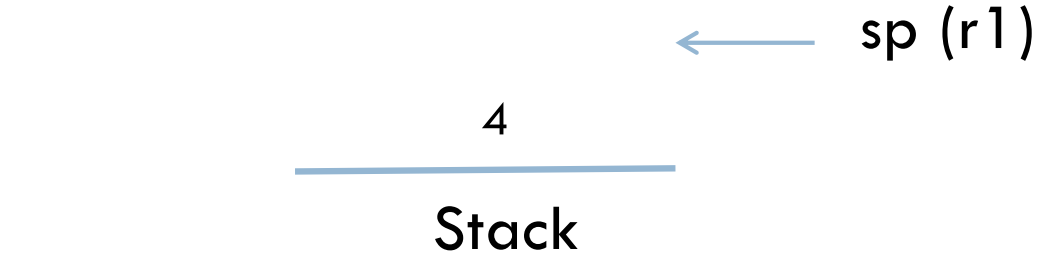
Stack

add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

keep the stack consistent (pop r0 throws the value away)

r2	loc: pop r0
r3	14

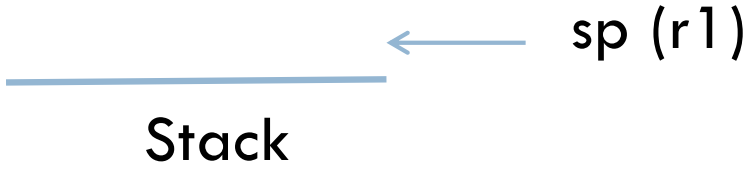


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

keep the stack consistent (pop r0 throws the value away)

r2	loc: pop r0
r3	14

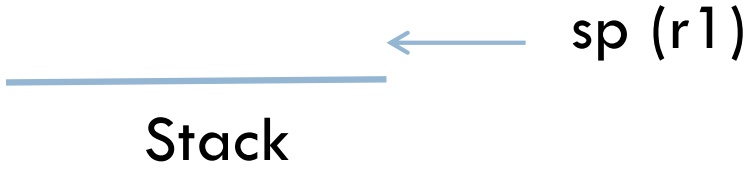


add (add.a51)

```
    lcw r1 stack
    loa r3 r0
    loa r2 r0

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    sto r3 r0
    hlt

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2
    pop r2
    jmp r2

    dat 100
stack
```

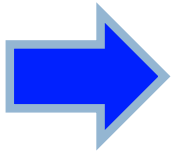
print out 14!

r2	loc: pop r0
r3	14



Call stack in action

```python
def absolute(x):
    if x < 0:
        x = -x

    return x

def add(x, y):
    return absolute(x + y)

def double(num):
    return 2 * num

def add_then_double(x, y):
    added = add(x, y)
    doubled = double(added)
    return doubled
```



```
absolute
    psh r2                  ; save the return address
    bge r3 r0 else          ; check if input is positive
    sub r3 r0 r3            ; negative, so flip the sign
else
    pop r2                  ; get the return address
    jmp r2                  ; return

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

    lcw r2 absolute         ; setup call for absolute (r3 already has input)
    cal r2 r2               ; call absolute

    pop r2                  ; get the return address
    jmp r2                  ; return (answer is already in r3)

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                  ; push input2 on the stack (r3 already has input1)
    lcw r2 add              ; setup call to add
    cal r2 r2               ; call add
    pop r0                  ; remove input2 from stack (and throw it away)

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

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

add_then_double.a51

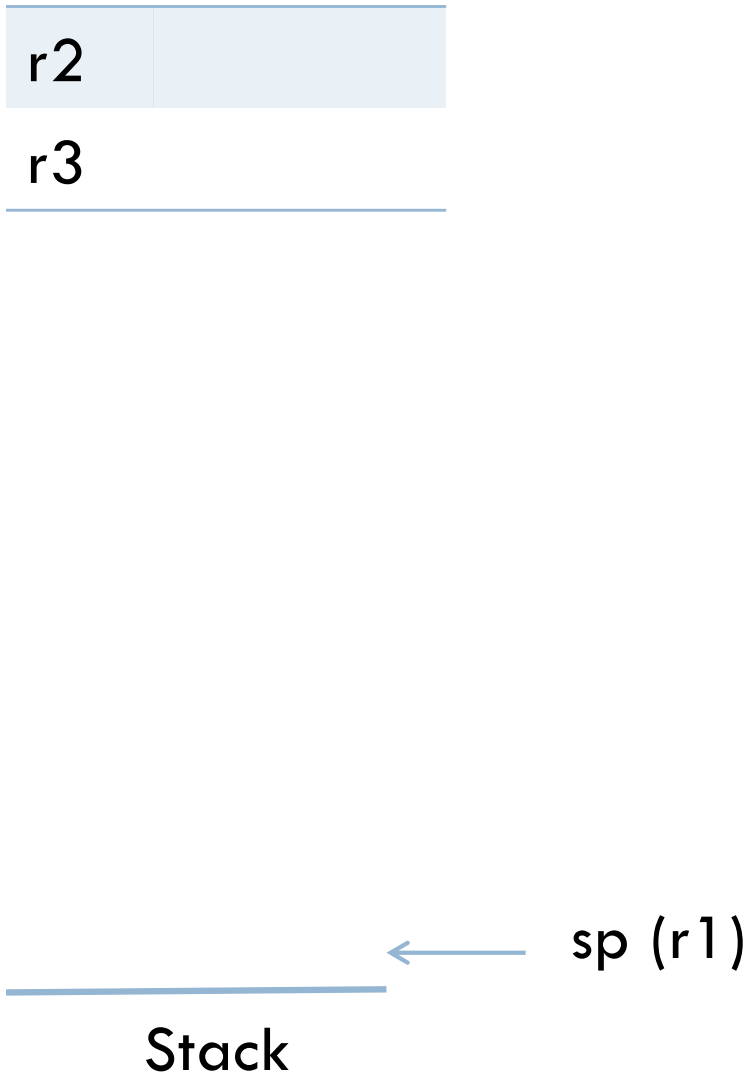
```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt
```

r2	
r3	

add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

setup the stack

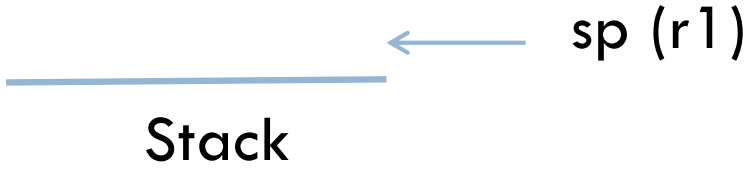


add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

get two values

r2	4
r3	-10

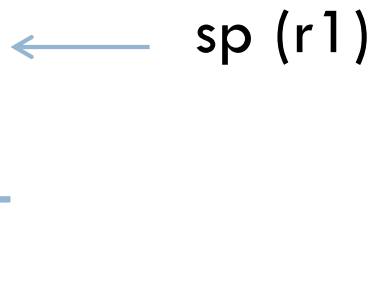


add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

save second argument onto stack

r2	4
r3	-10

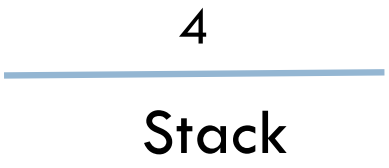


add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

setup the function call

r2	loc: a_t_d
r3	-10

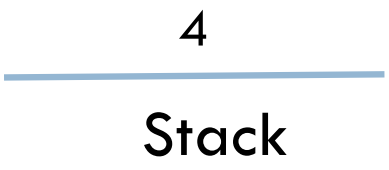


add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

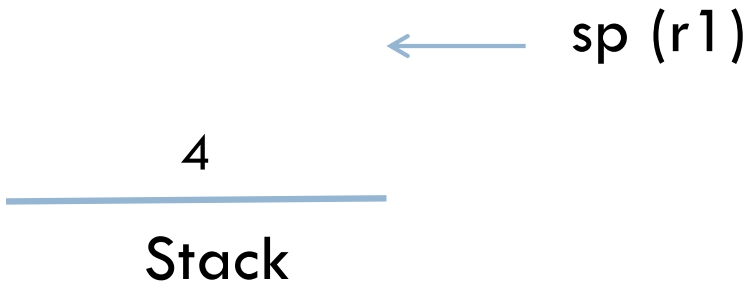
call the function
- go to location in r2
- save return address (ic)

r2	loc: a_t_d
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

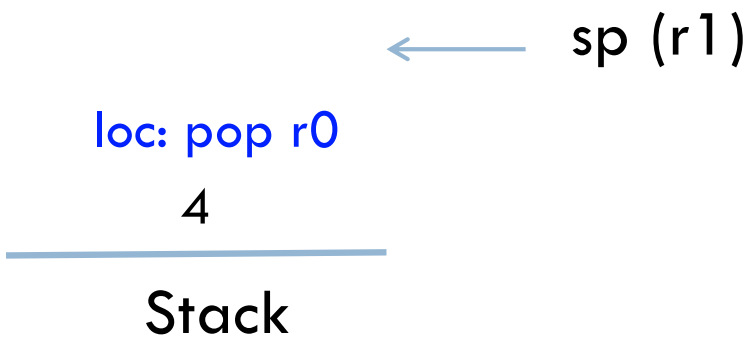
r2	loc: pop r0
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

save return address

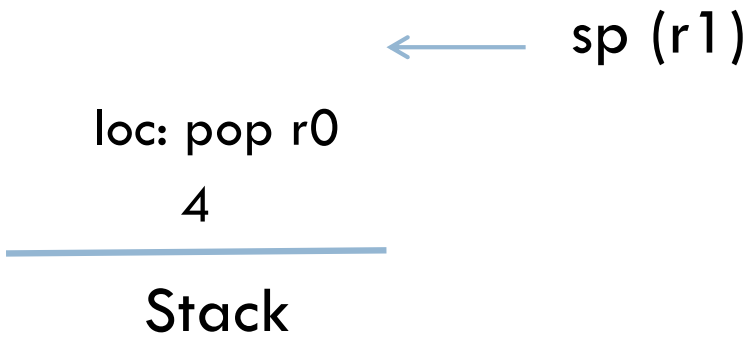
r2	loc: pop r0
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

get the second argument

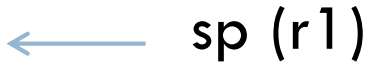
r2	4
r3	-10

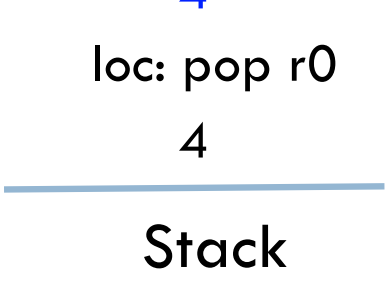


```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	4
r3	-10

add second argument for call to add

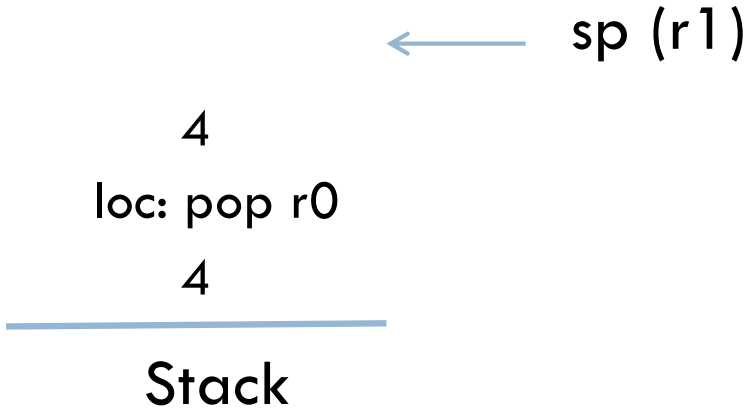




```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

setup call to add

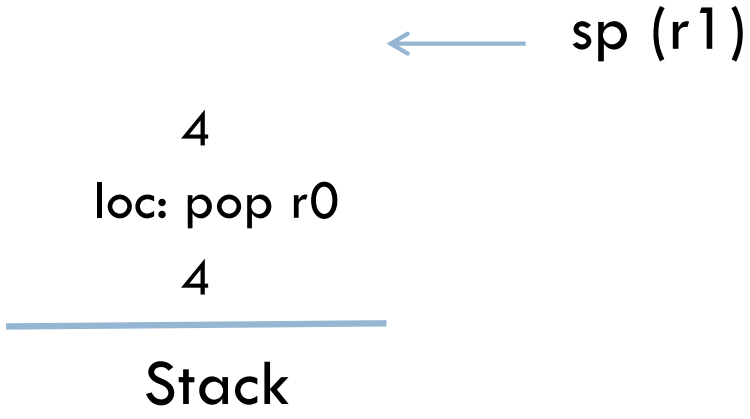
r2	loc: add
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

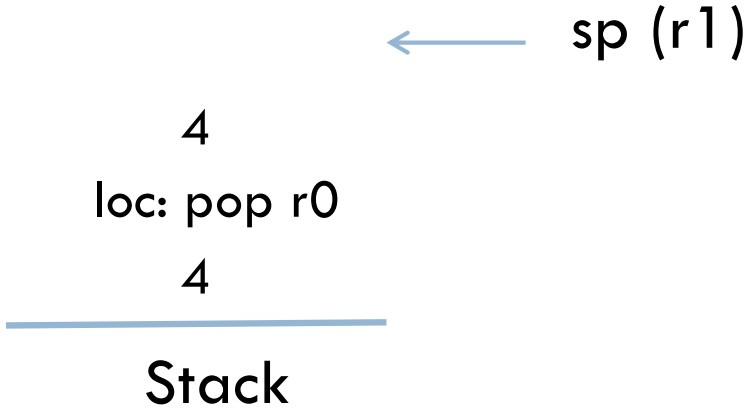
call add

r2	loc: add
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

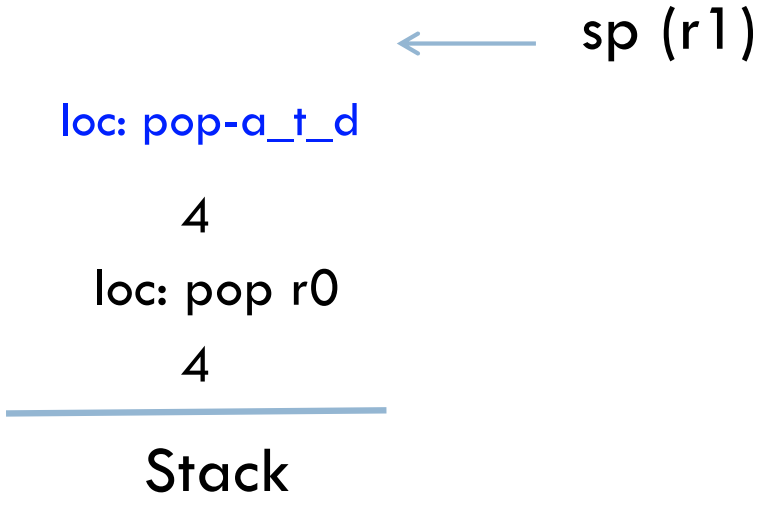
r2	loc: pop-a_t_d
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

save return address

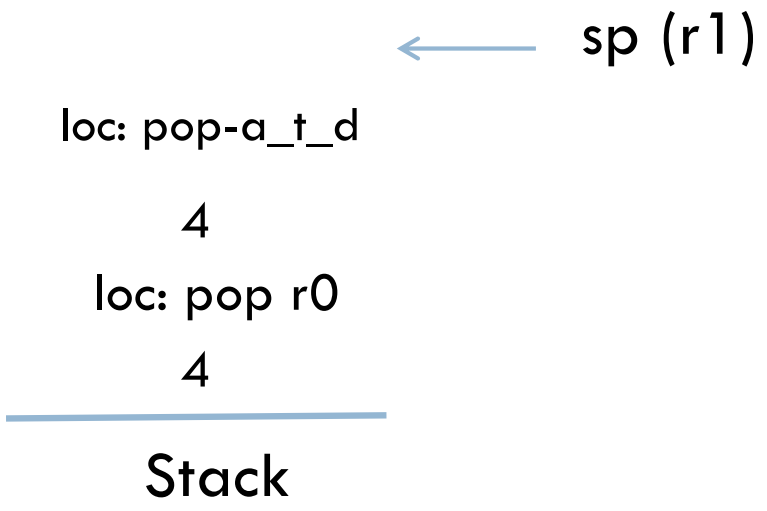
r2	loc: pop-a_t_d
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

get the second argument

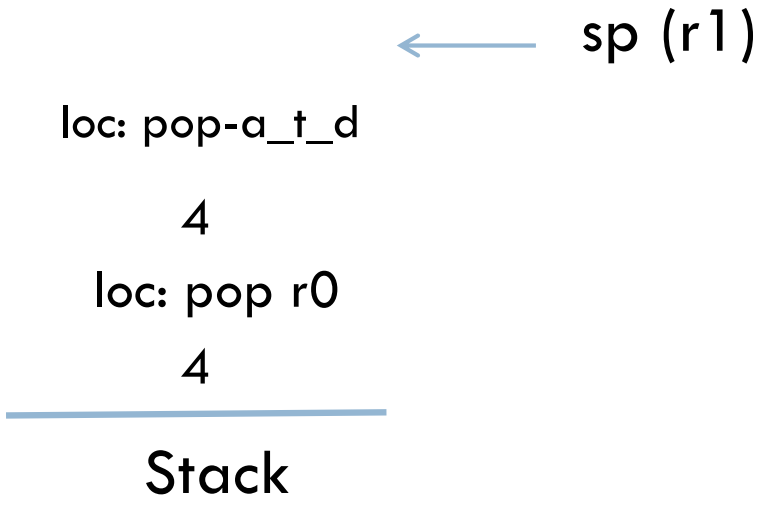
r2	4
r3	-10



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

add the two arguments

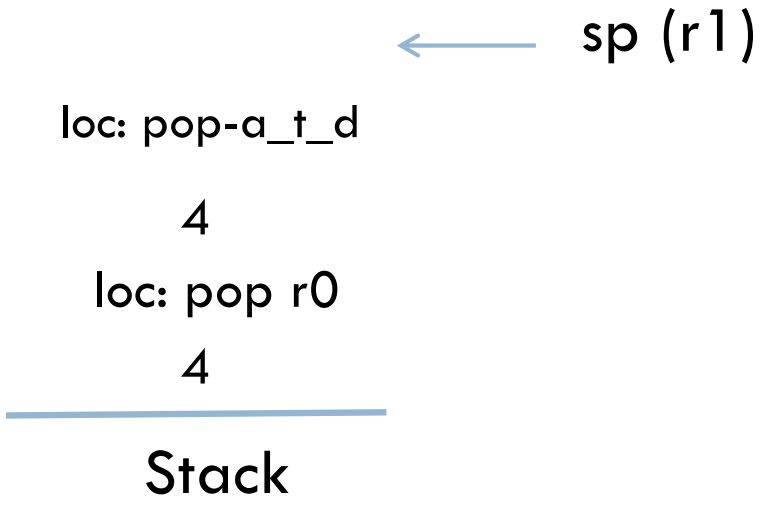
r2	4
r3	-6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

setup for call to absolute
(argument is already in r3)

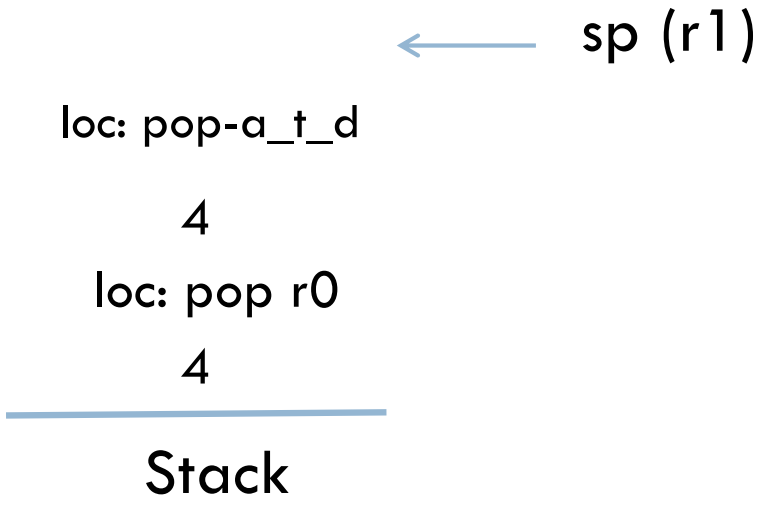
r2	loc: absolute
r3	-6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

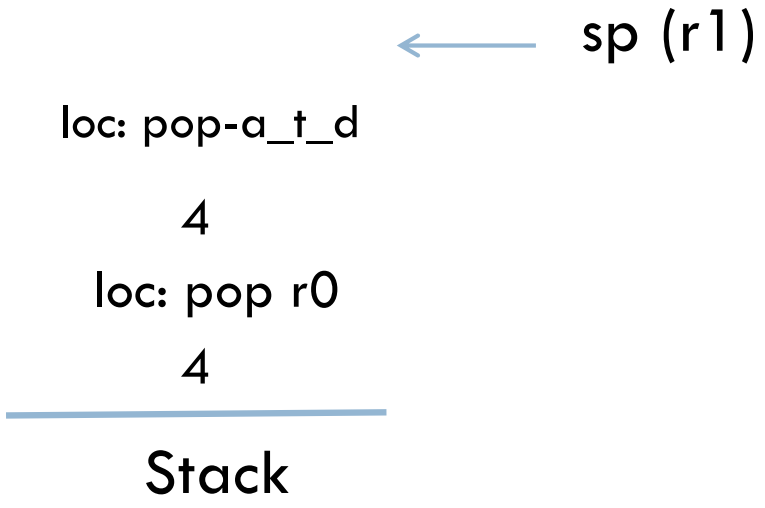
call absolute

r2	loc: absolute
r3	-6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

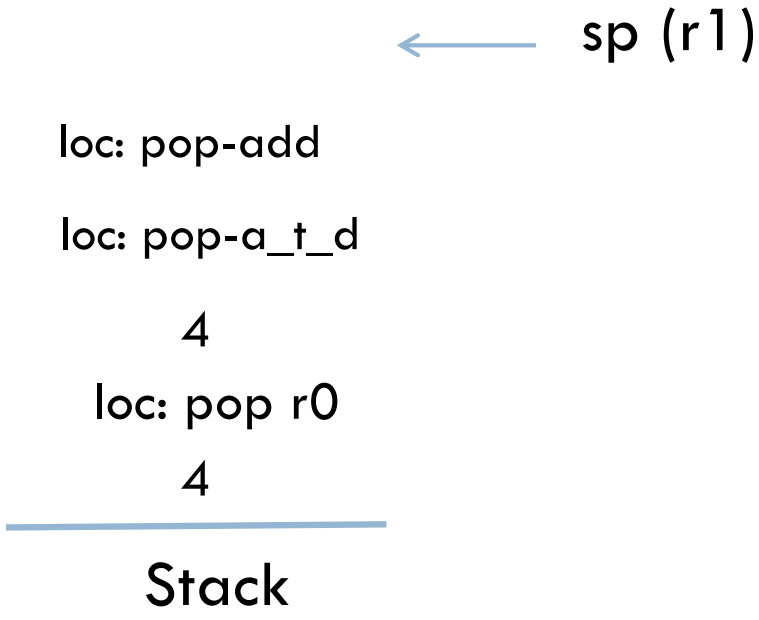
r2	loc: pop-add
r3	-6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

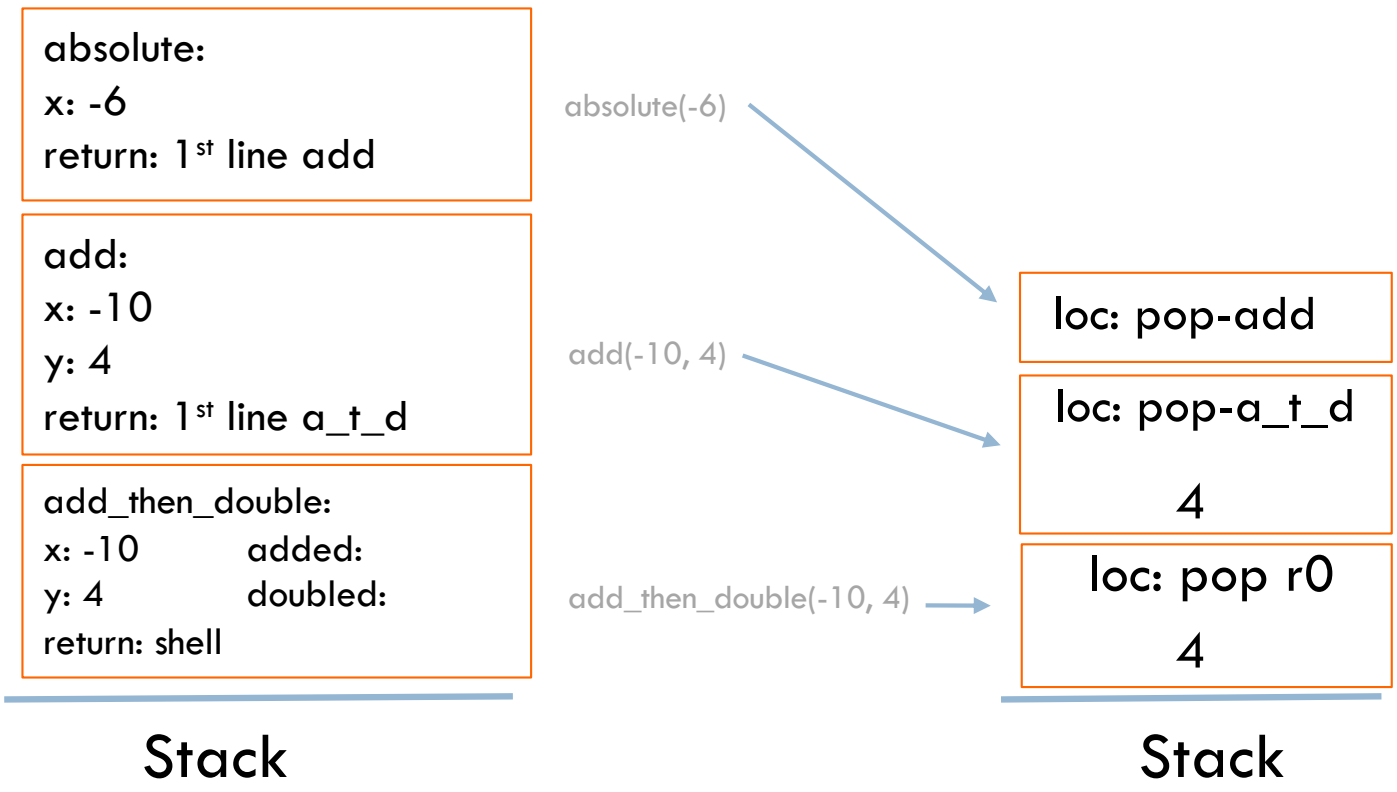
save the return address

r2	loc: pop-add
r3	-6



r2	loc: pop-add
r3	-6

Stack frames on the stack!



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

if r3 < 0

r2	loc: pop-add
r3	-6

sp (r1)

loc: pop-add

loc: pop-a_t_d

4

loc: pop r0

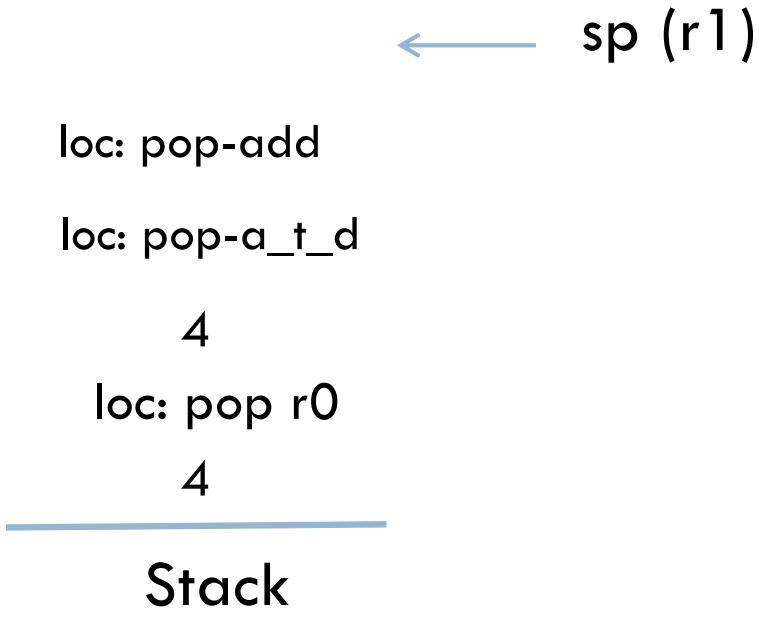
4

Stack

```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r3 = -r3

r2	loc: pop-add
r3	6

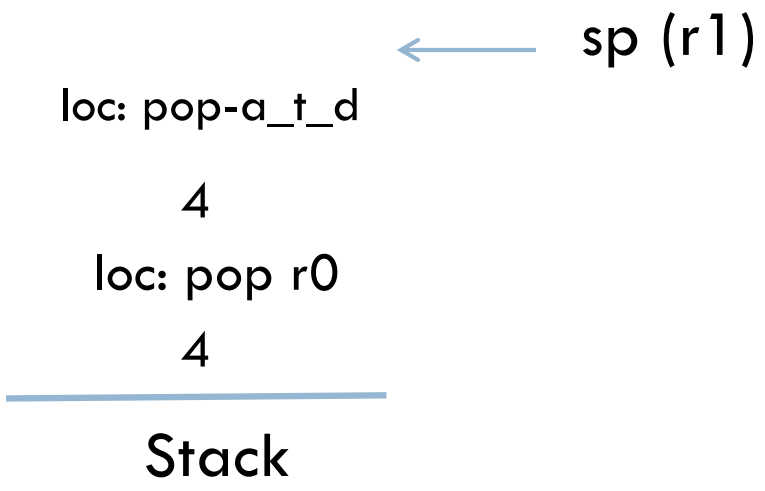


```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

get the return address

r2	loc: pop-add
r3	6

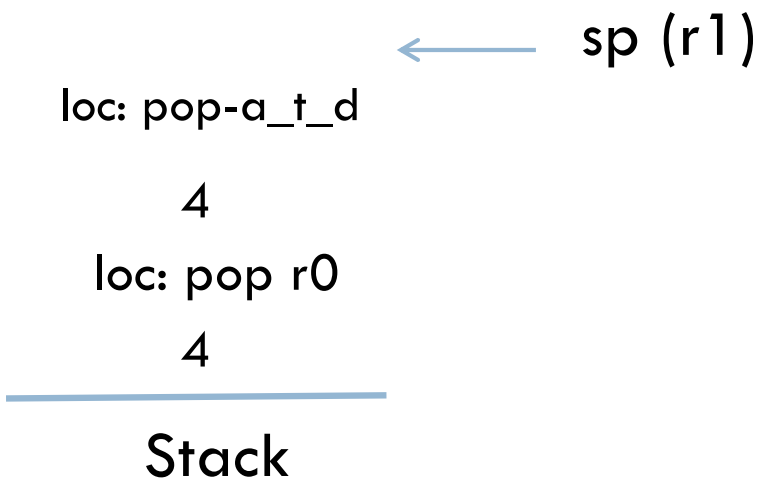
Note: since we didn't actually use r2 in absolute, we didn't actually need to save it to the stack



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

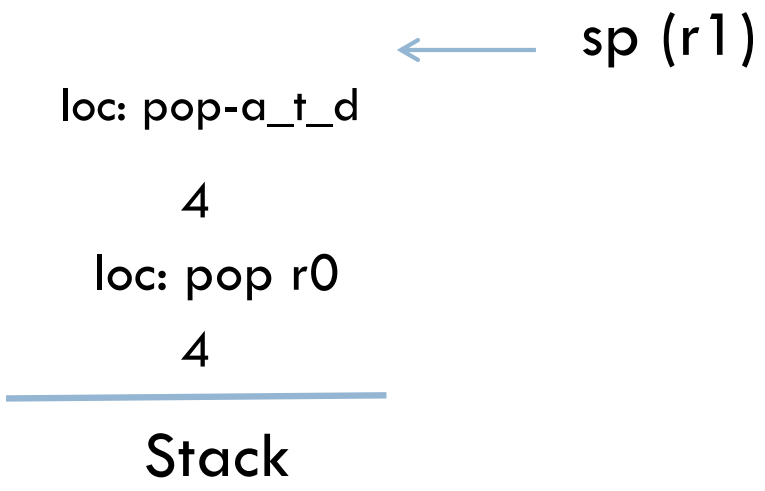
return (result is in r3)

r2	loc: pop-add
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

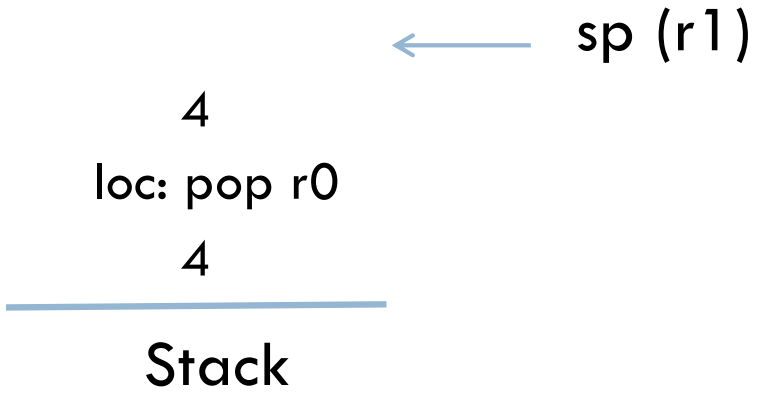
r2	loc: pop-add
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

get return address

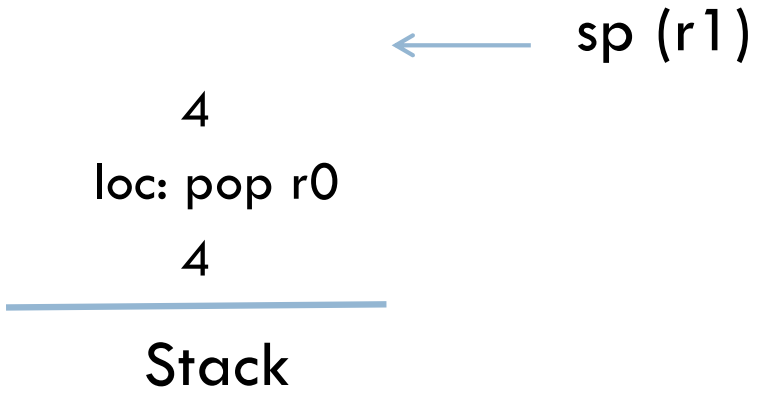
r2	loc: pop-a_t_d
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

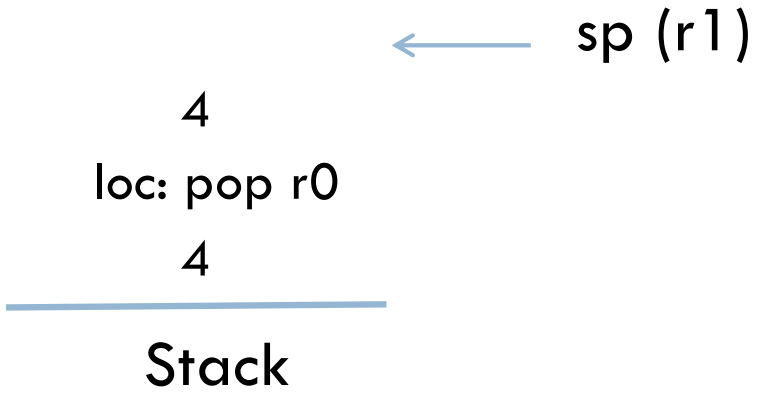
return (results is in r3)

r2	loc: pop-a_t_d
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

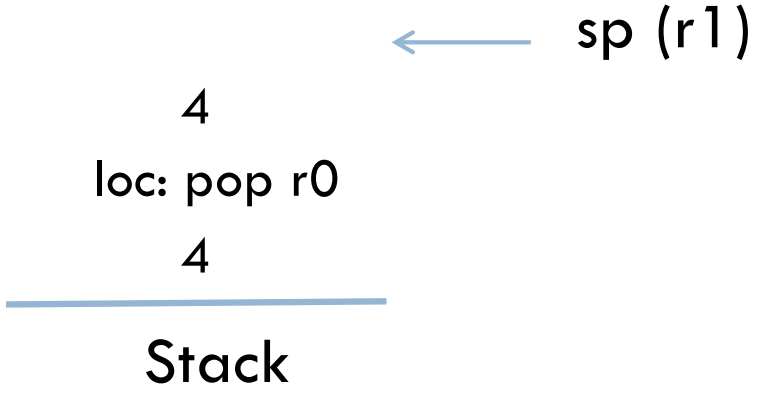
r2	loc: pop-a_t_d
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	loc: pop-a_t_d
r3	6

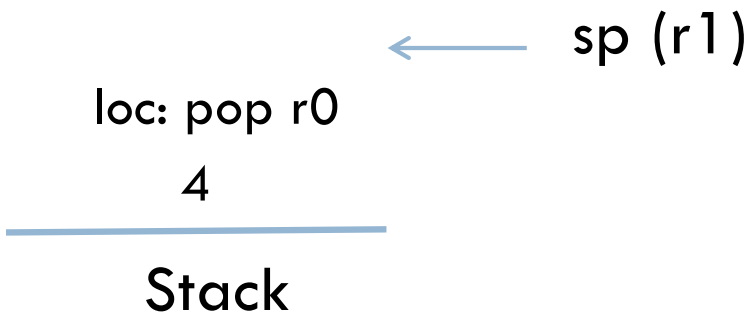
cleanup second argument from stack



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	loc: pop-a_t_d
r3	6

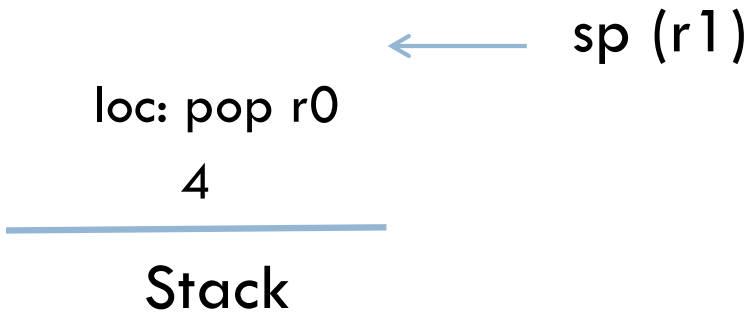
cleanup second argument from stack



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	loc: double
r3	6

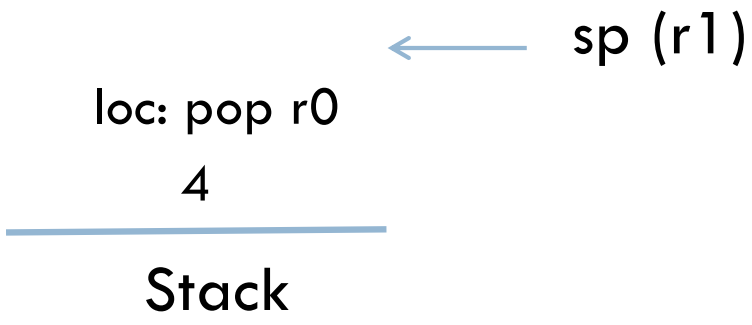
setup call to double (argument is already in r3)



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

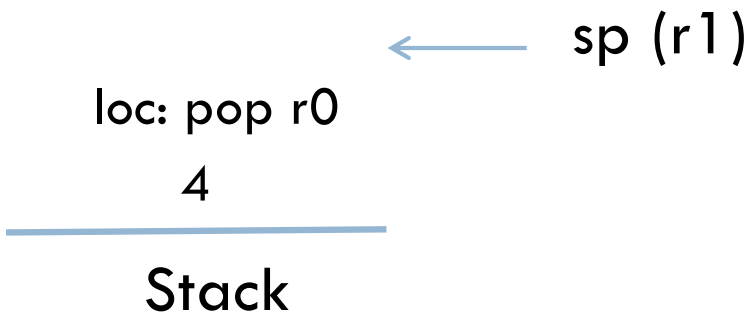
call double

r2	loc: double
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

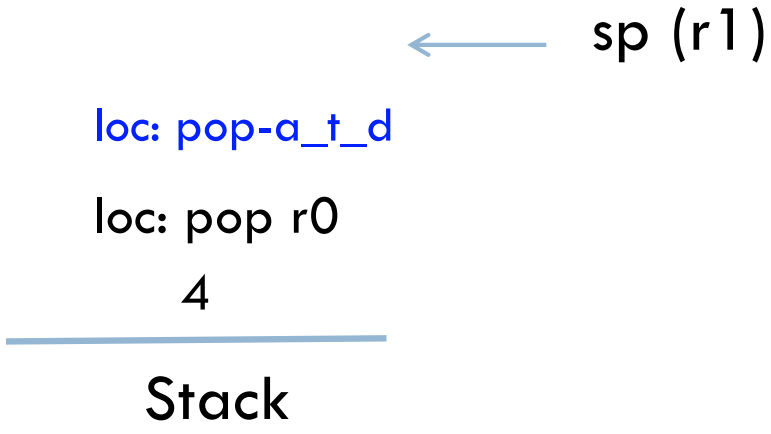
r2	loc: pop-a_t_d
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

save the return address

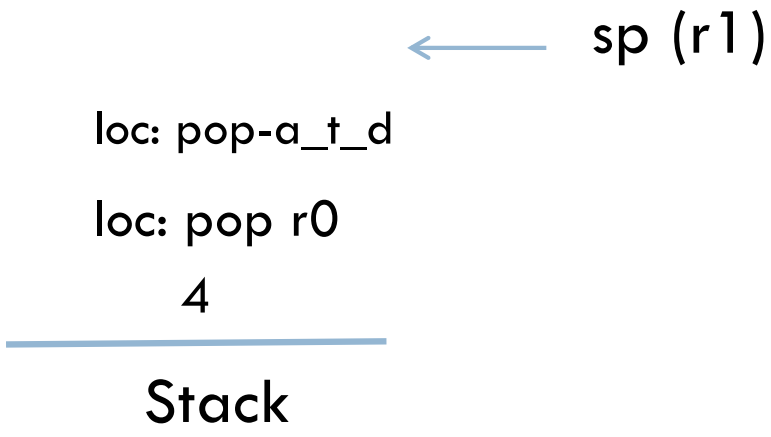
r2	loc: pop-a_t_d
r3	6



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

double r3 (r3 = r3 + r3)

r2	loc: pop-a_t_d
r3	12

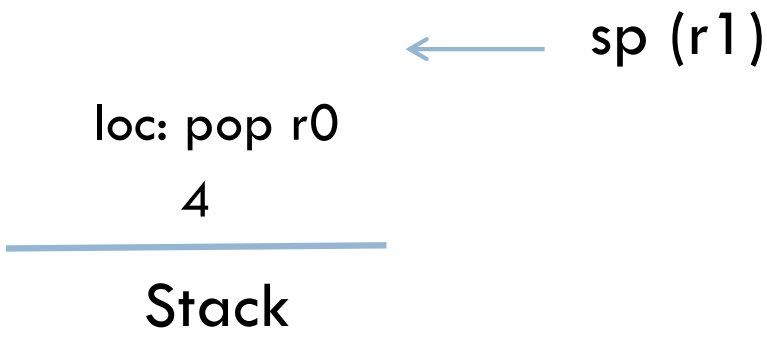


```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	loc: pop-a_t_d
r3	12

Again: since we didn't actually use r2 in absolute, we didn't actually need to save it to the stack

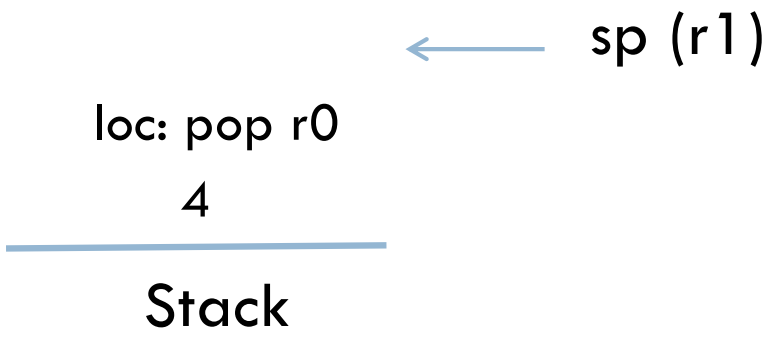
get the return address (answer is already in r3)



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

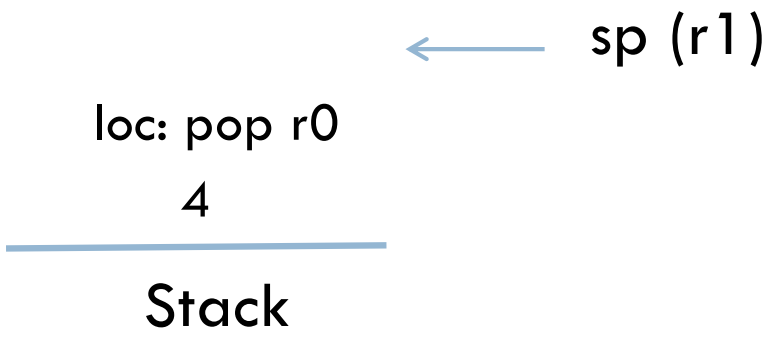
return

r2	loc: pop-a_t_d
r3	12



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

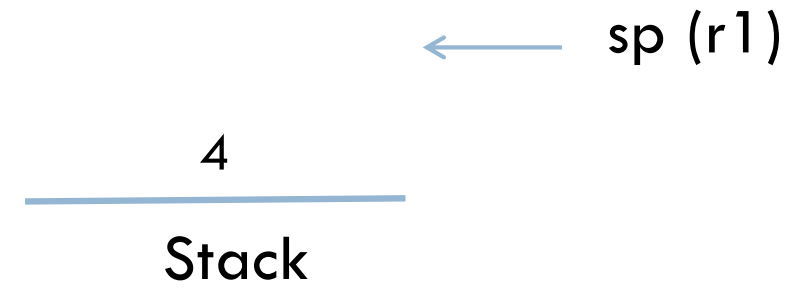
r2	loc: pop-a_t_d
r3	12



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

r2	loc: pop r0
r3	12

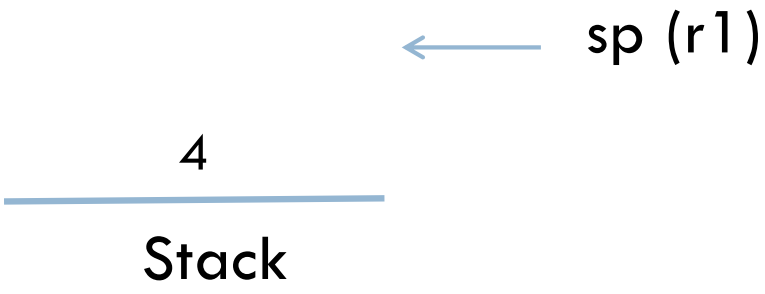
get the return address
(answer is already in r3)



```
absolute
    psh r2
    bge r3 r0 else
    sub r3 r0 r3
else
    pop r2
    jmp r2

add
    psh r2
    loa r2 r1 4
    add r3 r3 r2

    lcw r2 absolute
    cal r2 r2

    pop r2
    jmp r2

double
    psh r2
    add r3 r3 r3
    pop r2
    jmp r2

add_then_double
    psh r2
    loa r2 r1 4

    psh r2
    lcw r2 add
    cal r2 r2
    pop r0

    lcw r2 double
    cal r2 r2

    pop r2
    jmp r2
```

return

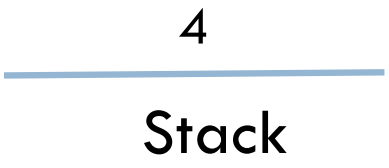
r2	loc: pop r0
r3	12



add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

r2	loc: pop r0
r3	12

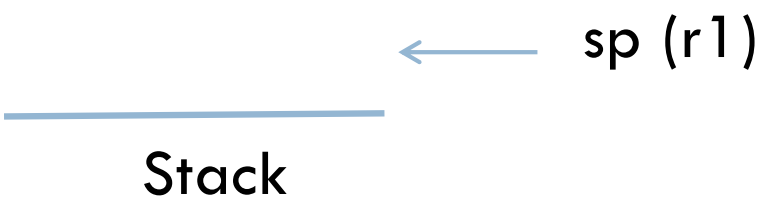


add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

clean up the stack
(pop r0 throws the value away)

r2	loc: pop r0
r3	12



add_then_double.a51

```
lcw r1 stack
loa r3 r0
loa r2 r0

psh r2
lcw r2 add_then_double
cal r2 r2
pop r0

sto r3 r0
hlt

...

   dat 100
stack
```

print 12

r2	loc: pop r0
r3	12

sp (r1)

Stack

To the simulator!

look at increment.a51 code

Legend:

R register	B signed byte offset	F string
S signed byte	W full word	[ ] optional
U unsigned byte	A 6-bit constant	

Arithmetic, bitwise logical, and shift instructions

```
add ┐
sub │
and ├ RRR or RRS        mov ┐
orr │                   neg ├ RR
xor ┘                   not ┘
```

Memory and data-moving instructions

```
sto ┐
loa ├ RR[S]             psh ┐
                        pop ├ R          lcw RW
```

Control instructions

```
beq ┐
bne │
blt │
bge ├ RRB               brs B            nop
bgt │
ble ┘                   cal RR           hlt [A]
                        jmp R
```

Directives

```
dat W                   inc F
```