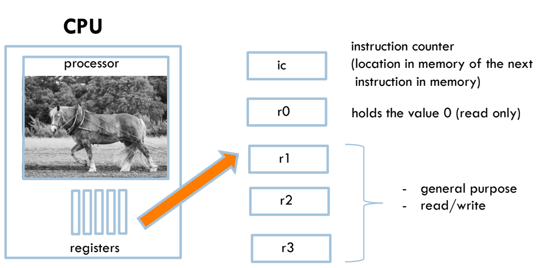
CS52 CALLING FUNCTIONS

David Kauchak
CS 52 – Spring 2017

Examples from this lecture

http://www.cs.pomona.edu/~dkauchak/classes/cs52/examples/cs52machine/

CS52 machine



CS52 machine execution

A *program* is simply a sequence of instructions stored in a block of contiguous words in the machine's memory. In executing a program, the CS52 Machine follows a simple loop:

- The machine fetches the value at `mem[ic]` for use as an instruction.
- The machine increments the value in `ic` by 2.
- The machine decodes and carries out the instruction.

Basic structure of CS52 program

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

- whitespace before operations/instructions
- labels go here

More CS52 examples

Look at max_simple.a52

- Get two values from the user
- Compare them
- Use a branch to distinguish between the two cases
  - Goal is to get largest value in r3
- print largest value

What does this code do?

```
       bge r3 r0 elif
       add r2 r0 -1
       brs endif
elif
       beq r3 r0 else
       add r2 r0 1
       brs endif
else
       add r2 r0 0
endif
       sto r2 r0
       hlt
```

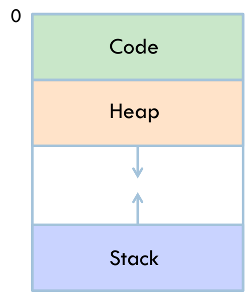
What does this code do?

```
       bge r3 r0 elif        if( r3 < 0 ){
       add r2 r0 -1            r2 = -1
       brs endif
elif
       beq r3 r0 else        }else if( r3 != 0 ){
       add r2 r0 1             r2 = 1
       brs endif
else                         }else{
       add r2 r0 0             r2 = 0
endif                        }
       sto r2 r0
       hlt
```

What does this code do?

```
      bge r3 r0 elif       ; if r3 >= 0 go to elif
      add r2 r0 -1         ; r3 < 0: r2 = -1
      brs endif            ; jump to end of if/elif/else
elif
      beq r3 r0 else       ; if r3 = 0 go to else
      add r2 r0 1          ; r3 > 0: r2 = 1
      brs endif            ; jump to end of if/elif/else
else
      add r2 r0 0          ; r3 = 0: r2 = 0
endif
      sto r2 r0            ; print out r2
      hlt
```

Memory layout



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

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

What is sum 2?

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

- sum 2

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

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

How do we evaluate this?

- sum 2

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

Make another function call

- sum 2

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

How do we evaluate this?

- sum 2

sum 1

sum:
x = 1
return: sum (2nd line)

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

- sum 2

Make another function call

sum 1

sum:
x = 1
return: sum (2nd line)

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

- sum 2

sum 0

sum:
x = 0
return: sum (2nd line)

What now?

sum 1

sum:
x = 1
return: sum (2nd line)

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

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

- sum 2

When a function finishes:
- return to where it was called from (return address)
- if substitute the function call with the return value
- pop the stack frame off the stack

sum 0	sum: x = 0 return: sum (2nd line)
sum 1	sum: x = 1 return: sum (2nd line)
sum 2	sum: x = 2 return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

0

- sum 2

What now?

When a function finishes:
- return to where it was called from (return address)
- if substitute the function call with the return value
- pop the stack frame off the stack

sum 1	sum: x = 1 return: sum (2nd line)
sum 2	sum: x = 2 return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

0

- sum 2

When a function finishes:
- return to where it was called from (return address)
- if substitute the function call with the return value
- pop the stack frame off the stack

sum 1	sum: x = 1 return: sum (2nd line)
sum 2	sum: x = 2 return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

1

- sum 2

What now?

When a function finishes:
- return to where it was called from (return address)
- if substitute the function call with the return value
- pop the stack frame off the stack

sum 2	sum: x = 2 return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

1

- sum 2

When a function finishes:
- return to where it was called from (return address)
- if substitute the function call with the return value
- pop the stack frame off the stack

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

2

- sum 2

Where do we return to?

sum 2

sum:
x = 2
return: shell

Stack

Stack frames

```
fun sum 0 = 0
  | sum x = x + sum (x-1);
```

2

- sum 2
val it = 3 : int

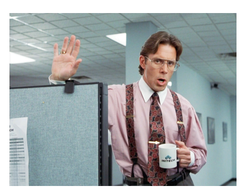
Stack

Function calls in assembly

For high-level languages the stack is managed for you

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

Stack



CS52 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

```
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
```

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
```

r2

r3

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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 (2^{nd} r2)
2. Save current ic into r2 (i.e. the address of the *next* instruction that would have been executed)

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

loc: sto

Stack

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

sp (r1)

loc: sto

Stack

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

sp (r1)

loc: sto

Stack

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

sp (r1)

loc: sto

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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

sp (r1)

Stack

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 ☺

sp (r1)

Stack

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

sp (r1)

Stack

To the simulator!

look at increment.a52 code

Sum revisited

```
fun sum x =
  if x <= 0 then
      0
  else
      x + sum (x-1);
```

Note to future Dave from past Dave: write the function up on the board ☺

```
sum
    psh r2              ; save the return address on the stack
    bgt r3 r0 recurse   ; check base case
    add r3 r0 0         ; if x <= 0, result is 0
    brs done

recurse
    psh r3              ; save n on the stack
    sub r3 r3 1         ; x = x-1

    lcw r2 sum          ; make recursive call
    cal r2 r2           ; sum (x-1), answer should be in r3

    pop r2              ; get n into r2
    add r3 r3 r2        ; r3 = n + sum (x-1)
done
    pop r2              ; get the return address
    jmp r2              ; go back to where we were called from
```

Function startup

base cases

recursive call

recursive case

answer calculation

Function cleanup and return

```
sum
    psh r2              ; save the return address on the stack
    bgt r3 r0 recurse   ; check base case
    add r3 r0 0         ; if n <= 0, result is 0
    brs done

recurse
    psh r3              ; save n on the stack
    sub r3 r3 1         ; n = n-1

    lcw r2 sum          ; make recursive call
    cal r2 r2           ; sum(n-1), answer should be in r3

    pop r2              ; get n into r2
    add r3 r3 r2        ; r3 = n + sum(n-1)
done
    pop r2              ; get the return address
    jmp r2              ; go back to where we were called from
```

Notice symmetry of psh and pop

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	
r3	

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	
r3	2

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	
r3	2

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:sum
r3	2

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:sum
r3	2

sp (r1)

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

sp (r1)

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

sp (r1)

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

Why psh r3?

← sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

- We're about to make a function call
- The result of that call will go into r3 so we'll lose what's in there if we don't save it!
- **x + sum (x-1)**

← sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	2

← sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	1

← sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	1

Make a recursive call:
sum 1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:sum
r3	0

Make a recursive call: sum 0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

sp (r1)

loc:cal1

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

sp (r1)

loc:cal1

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

sp (r1)

loc:cal1

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:call
r3	0

sp (r1)

loc:call
1
loc:call
2
loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:call
r3	0

sp (r1)

loc:call
1
loc:call
2
loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:call
r3	0

sp (r1)

loc:call
1
loc:call
2
loc:cal0

Stack

r2	loc:call
r3	0

Stack frames!

sum 0 — sum: x = 0, return: sum (2nd line)

sum 1 — sum: x = 1, return: sum (2nd line)

sum 2 — sum: x = 2, return: shell

sp (r1)

loc:call — sum 0
1, loc:call — sum 1
2, loc:cal0 — sum 2

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

loc:cal1

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

← sp (r1)

1

loc:cal1

2

loc:cal0

Returning with answer in r3

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

Why are we doing this?

sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	0

- Need to calculate x + sum (x-1)
- Saved x on the stack

sp (r1)

1

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	1
r3	0

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	1
r3	0

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	1
r3	1

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	1
r3	1

sp (r1)

loc:cal1

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

2

loc:cal0

Stack

Returning with answer in r3

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal1
r3	1

sp (r1)

2

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	2
r3	1

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done
recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	2
r3	1

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	2
r3	3

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	2
r3	3

sp (r1)

loc:cal0

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	3

sp (r1)

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	3

sp (r1)

Stack

```
sum
        psh r2
        bgt r3 r0 recurse
        add r3 r0 0
        brs done

recurse
        psh r3
        sub r3 r3 1

        lcw r2 sum
        cal r2 r2

        pop r2
        add r3 r3 r2
done
        pop r2
        jmp r2
```

r2	loc:cal0
r3	3

sp (r1)

Returning with answer in r3

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:cal0
r3	3

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:cal0
r3	3

Print the answer: 3!

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:cal0
r3	3

sp (r1)

Stack

Calling sum

```
loa r3 r0

lcw r2 sum
cal r2 r2

sto r3 r0
hlt
```

r2	loc:cal0
r3	3

Notice that when we're all done, the stack is empty

sp (r1)

Stack

Real structure of CS52 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

Time permitting

Bitwise operators:
- and
- orr
- xor

Admin

Midterm 1
- Average: 36.4 (83%)
- Q1: 32.75 (74%)
- Q2 (median): 37.5 (85%)
- Q3: 40.5 (92%)

Assignment 4

Examples from this lecture

http://www.cs.pomona.edu/~dkauchak/classes/cs52/examples/cs52machine/

max_simple.a52: max (repeated from last time)
increment.a52: increment example
sum.a52: sum example