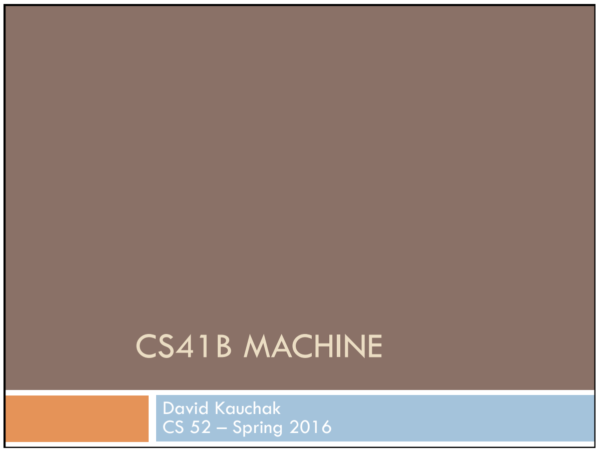
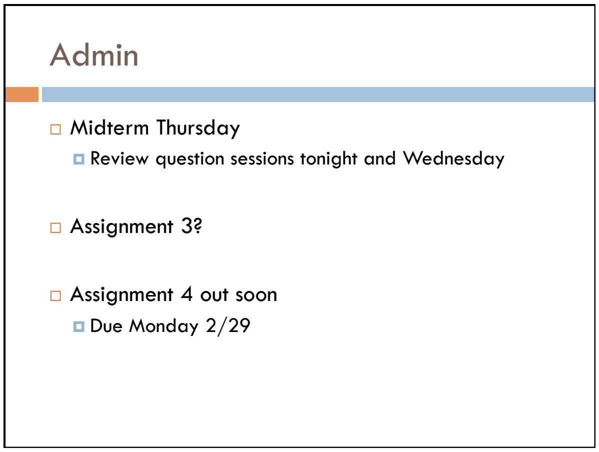


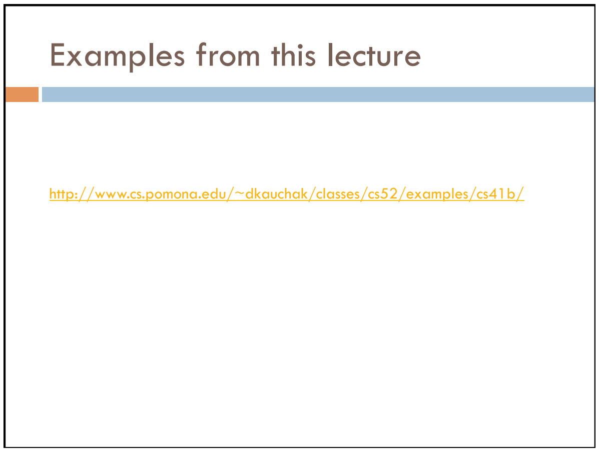
CS41B MACHINE

David Kauchak
CS 52 – Spring 2016



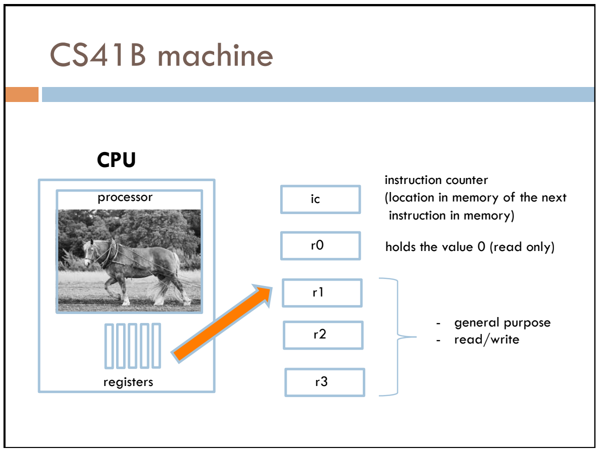
Admin

- Midterm Thursday
  - Review question sessions tonight and Wednesday
- Assignment 3?
- Assignment 4 out soon
  - Due Monday 2/29



Examples from this lecture

http://www.cs.pomona.edu/~dkauchak/classes/cs52/examples/cs41b/



CS41B machine

CPU

processor

registers

ic

instruction counter
(location in memory of the next
instruction in memory)

r0

holds the value 0 (read only)

r1

r2

r3

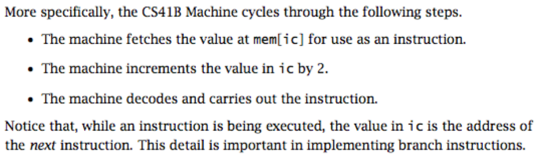
- general purpose
- read/write

CS41B code

Four main types of operations

1. math
2. branch/conditionals
3. memory
4. control the machine (e.g. stop it)

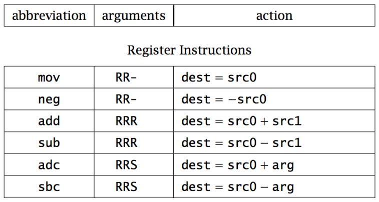
CS41B execution



More specifically, the CS41B Machine cycles through the following steps.

- 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.

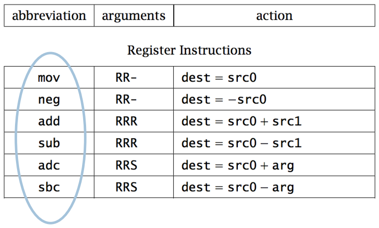
Notice that, while an instruction is being executed, the value in `ic` is the address of the *next* instruction. This detail is important in implementing branch instructions.



abbreviation	arguments	action

Register Instructions

mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg

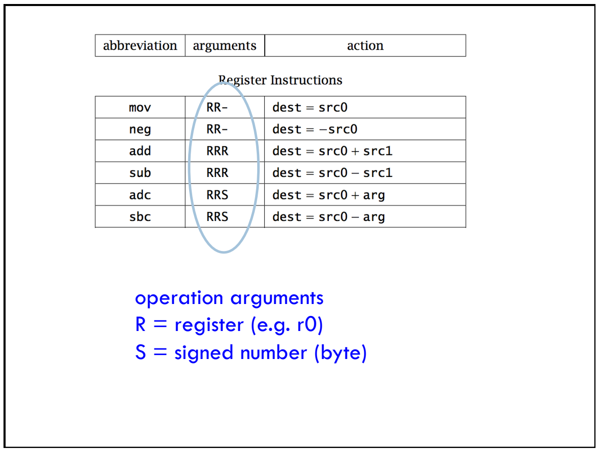


abbreviation	arguments	action

Register Instructions

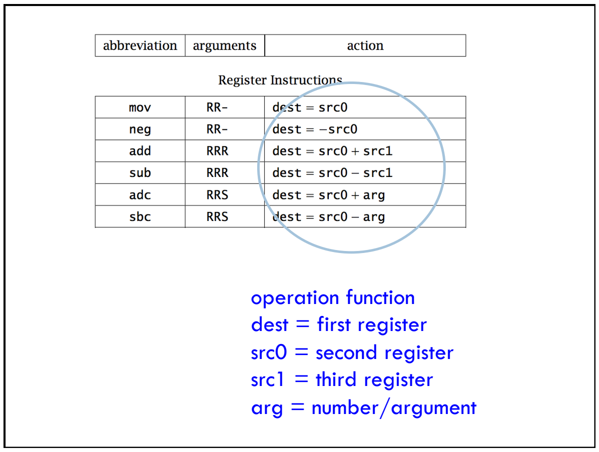
mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg

operation name
(always three characters)



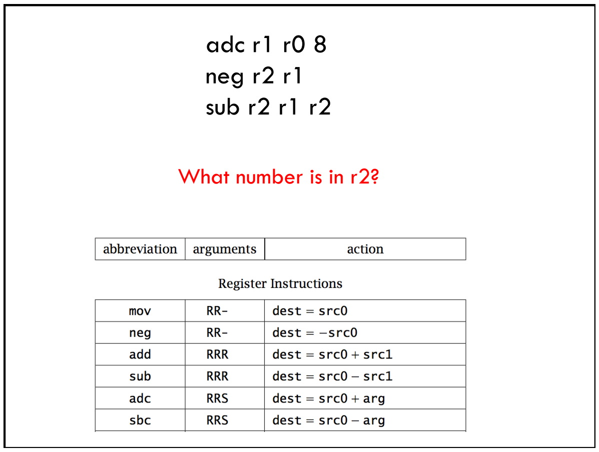
abbreviation	arguments	action
	Register Instructions	
mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg

operation arguments
R = register (e.g. r0)
S = signed number (byte)



abbreviation	arguments	action
	Register Instructions	
mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg

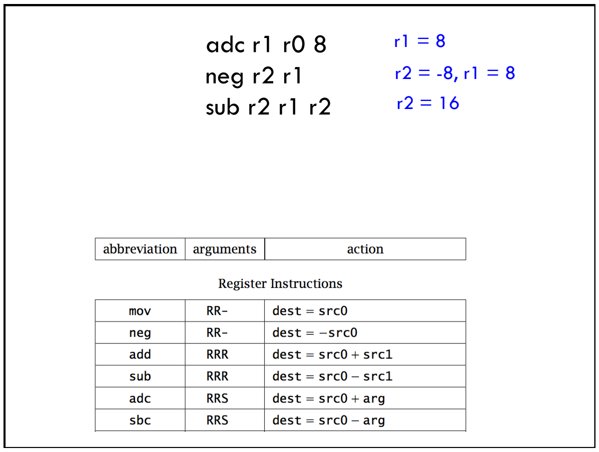
operation function
dest = first register
src0 = second register
src1 = third register
arg = number/argument



```
adc r1 r0 8
neg r2 r1
sub r2 r1 r2
```

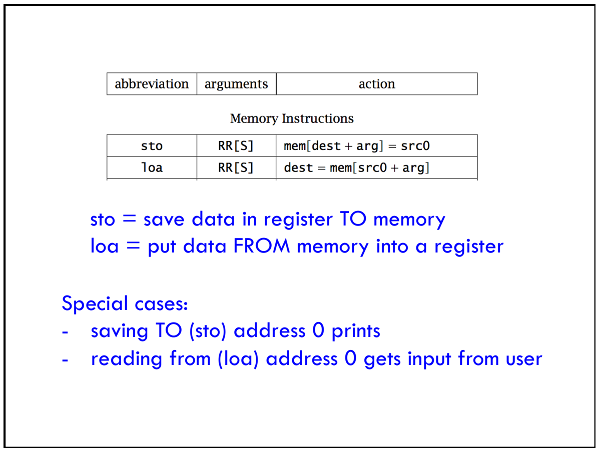
What number is in r2?

abbreviation	arguments	action
	Register Instructions	
mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg



```
adc r1 r0 8      r1 = 8
neg r2 r1        r2 = -8, r1 = 8
sub r2 r1 r2     r2 = 16
```

abbreviation	arguments	action
	Register Instructions	
mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg

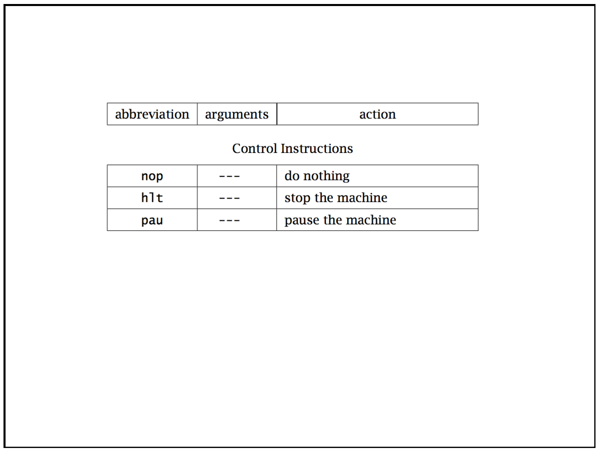


abbreviation	arguments	action
	Memory Instructions	
`sto`	`RR[S]`	`mem[dest + arg] = src0`
`loa`	`RR[S]`	`dest = mem[src0 + arg]`

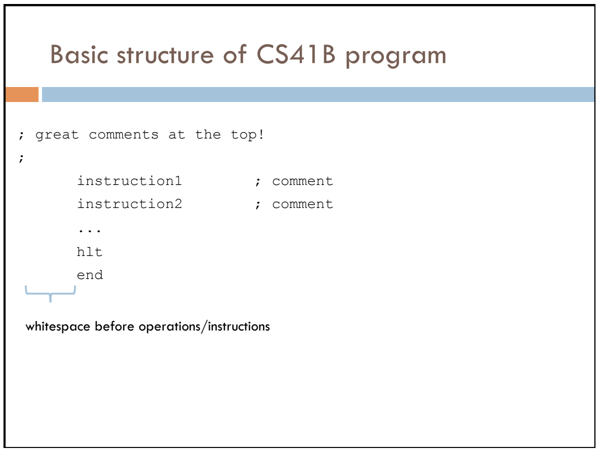
sto = save data in register TO memory
loa = put data FROM memory into a register

Special cases:

- saving TO (sto) address 0 prints
- reading from (loa) address 0 gets input from user



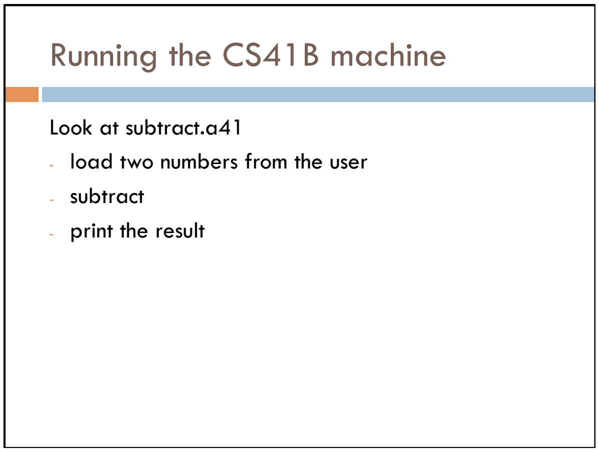
abbreviation	arguments	action
	Control Instructions	
`nop`	---	do nothing
`hlt`	---	stop the machine
`pau`	---	pause the machine



Basic structure of CS41B program

```
; great comments at the top!
;
       instruction1        ; comment
       instruction2        ; comment
       ...
       hlt
       end
```

whitespace before operations/instructions



Running the CS41B machine

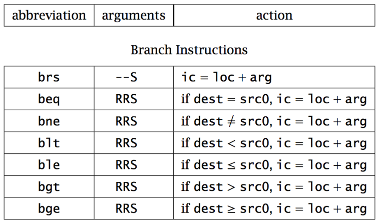
Look at subtract.a41

- load two numbers from the user
- subtract
- print the result

CS41B simulater

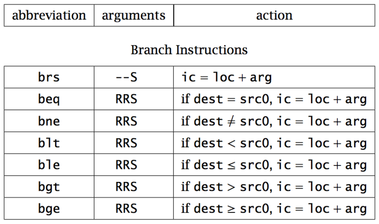
Different windows

- Memory (left)
- Instruction execution (right)
- Registers
- I/O and running program



abbreviation	arguments	action
	Branch Instructions	
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

What do these operations do?

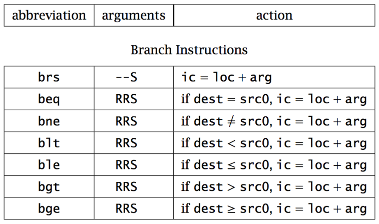


abbreviation	arguments	action
	Branch Instructions	
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

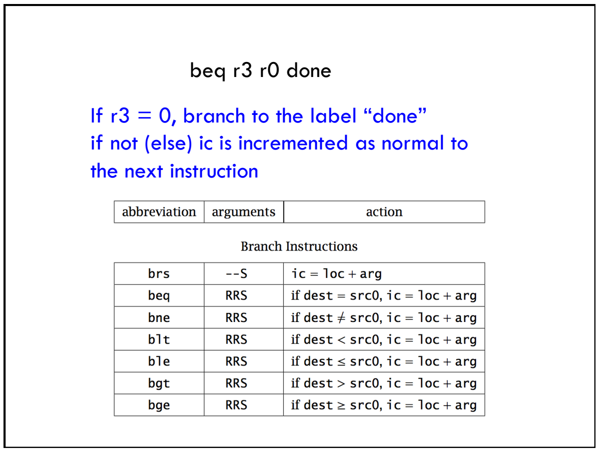
Modify ic, the instruction counter…
which changes the flow of the program!

beq r3 r0 done

What does this do?



abbreviation	arguments	action
	Branch Instructions	
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

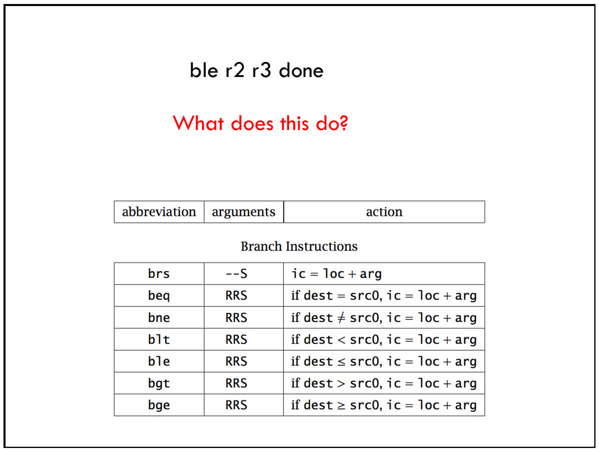


beq r3 r0 done

If r3 = 0, branch to the label "done"
if not (else) ic is incremented as normal to the next instruction

Branch Instructions

abbreviation	arguments	action
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

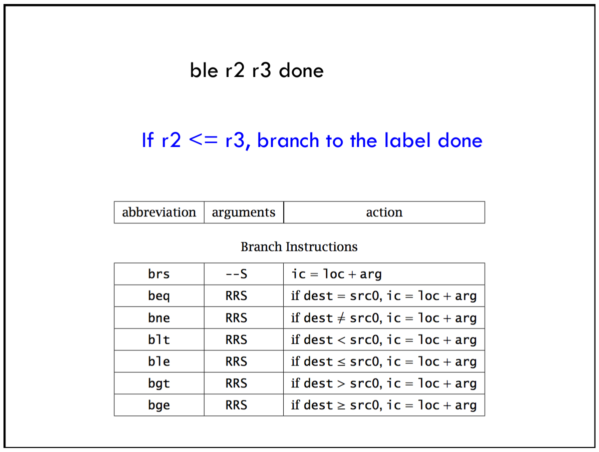


ble r2 r3 done

What does this do?

Branch Instructions

abbreviation	arguments	action
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

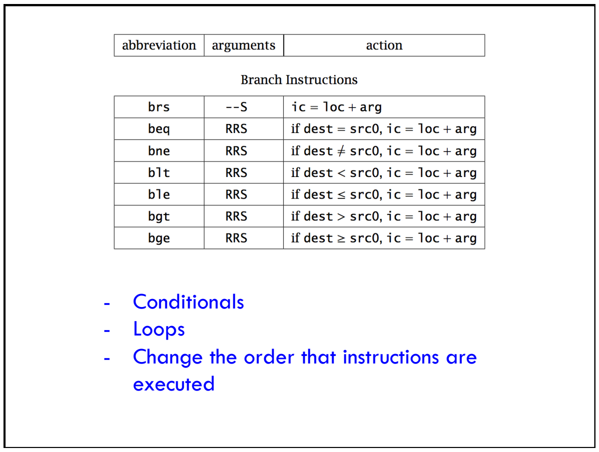


ble r2 r3 done

If r2 <= r3, branch to the label done

Branch Instructions

abbreviation	arguments	action
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg



Branch Instructions

abbreviation	arguments	action
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

- Conditionals
- Loops
- Change the order that instructions are executed

Basic structure of CS41B program

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

- whitespace before operations/instructions
- labels go here

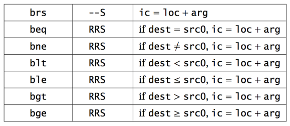
More CS41B examples

Look at max_simple.a41

- 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?

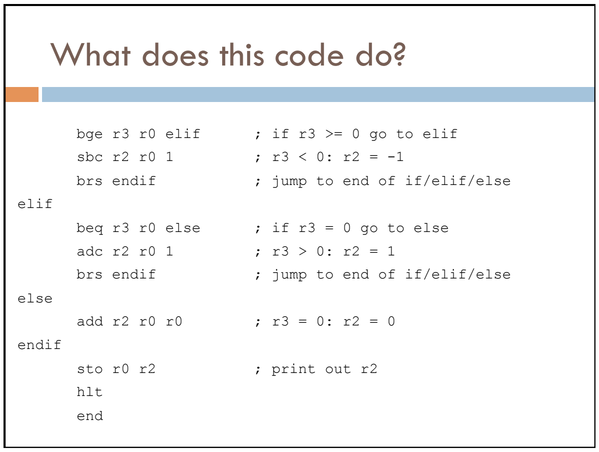
```
       bge r3 r0 elif
       sbc r2 r0 1
       brs endif
elif
       beq r3 r0 else
       adc r2 r0 1
       brs endif
else
       add r2 r0 r0
endif
       sto r0 r2
       hlt
       end
```



brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

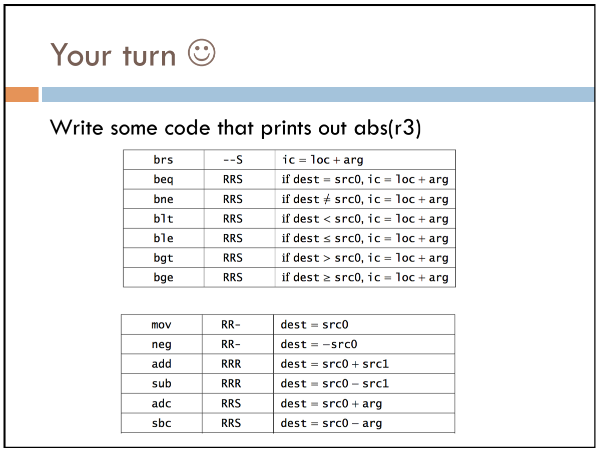
What does this code do?

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



What does this code do?

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

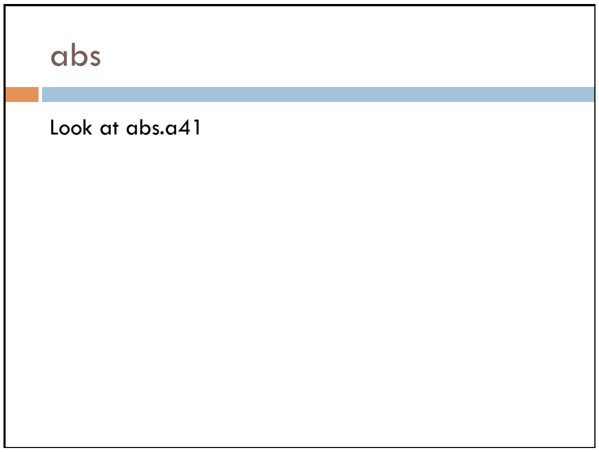


Your turn ☺

Write some code that prints out abs(r3)

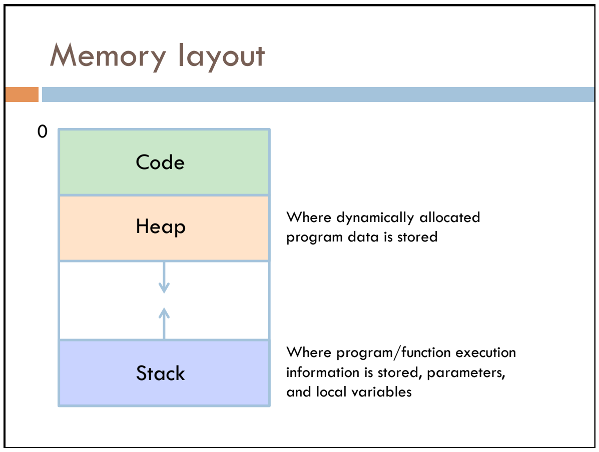
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg

mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg



abs

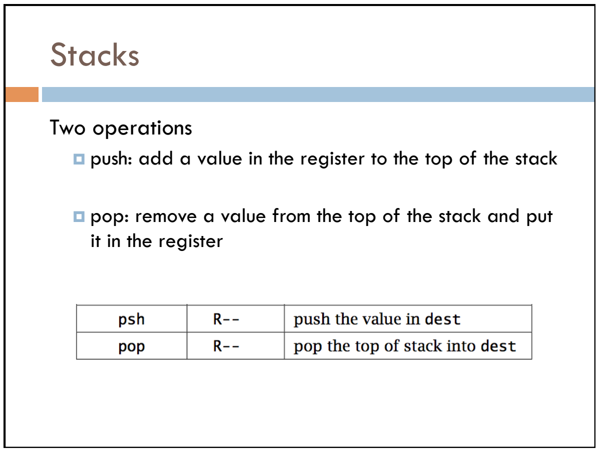
Look at abs.a41



Memory layout

Where dynamically allocated program data is stored

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

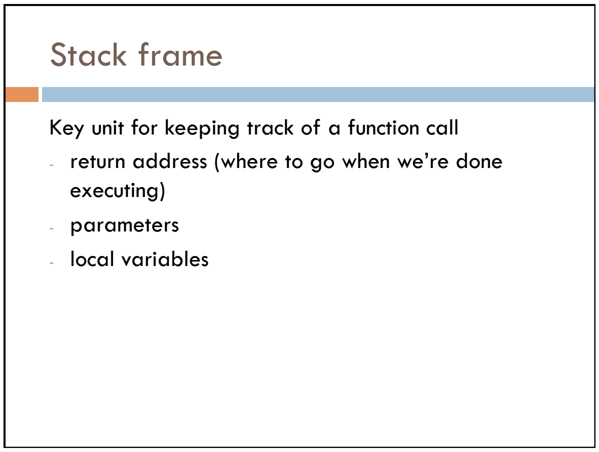


Stacks

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

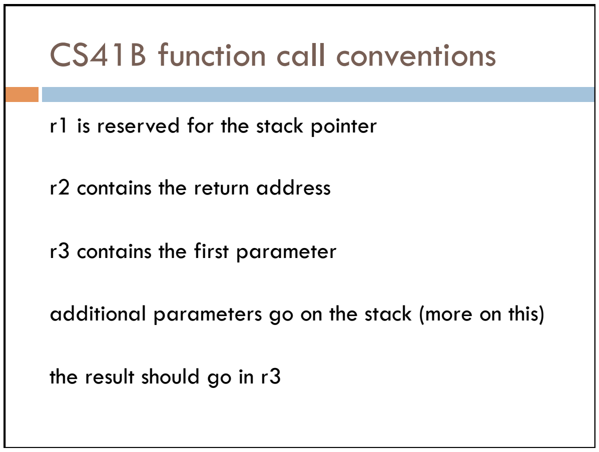
psh	R--	push the value in dest
pop	R--	pop the top of stack into dest



Stack frame

Key unit for keeping track of a function call

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



CS41B function call conventions

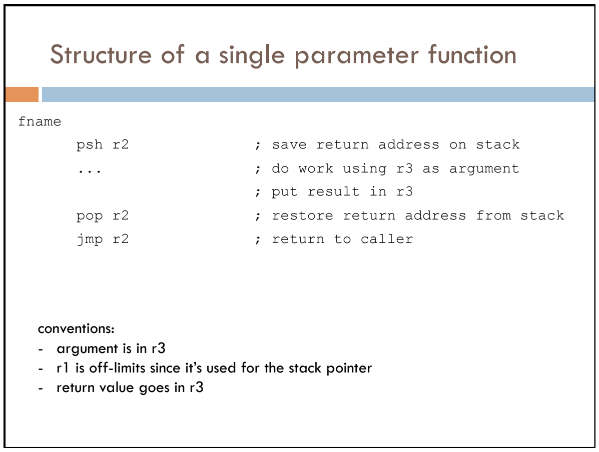
r1 is reserved for the stack pointer

r2 contains the return address

r3 contains the first parameter

additional parameters go on the stack (more on this)

the result should go in r3

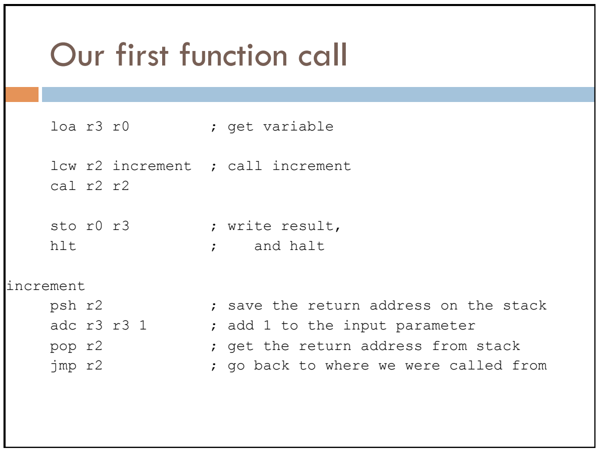


Structure of a single parameter function

```
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
```

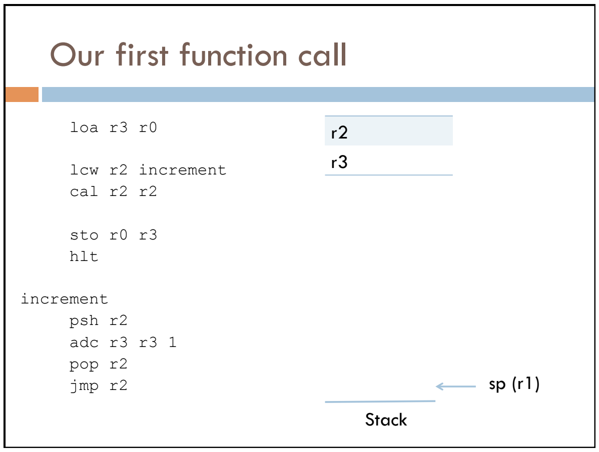
conventions:

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



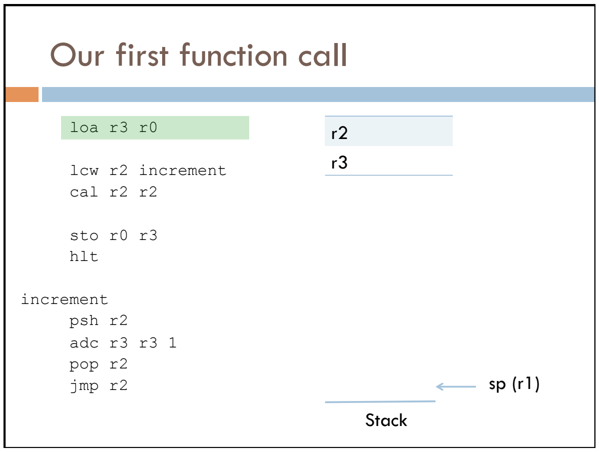
Our first function call

```
    loa r3 r0          ; get variable

    lcw r2 increment   ; call increment
    cal r2 r2

    sto r0 r3          ; write result,
    hlt                ;    and halt

increment
    psh r2             ; save the return address on the stack
    adc 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
```



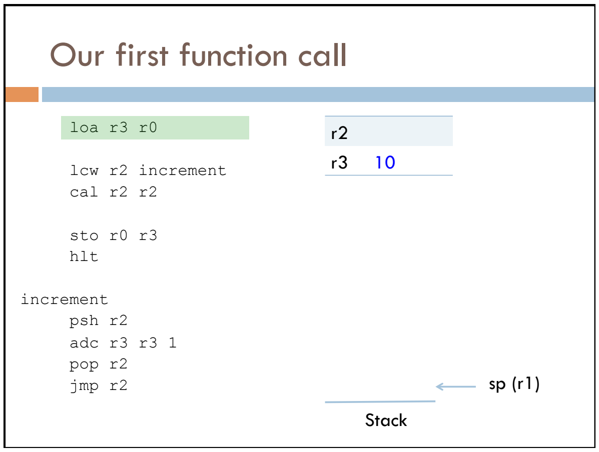
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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



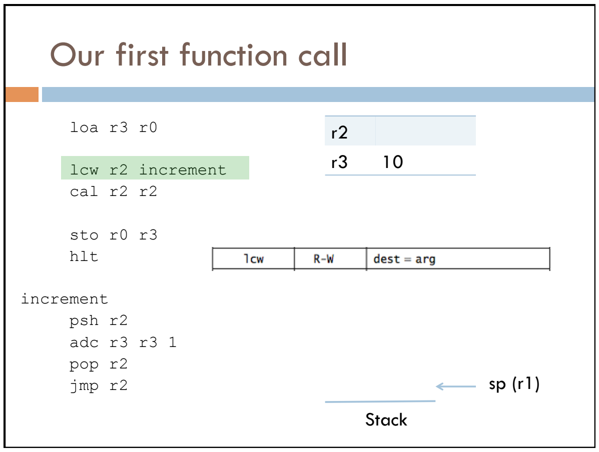
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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



Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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



Our first function call

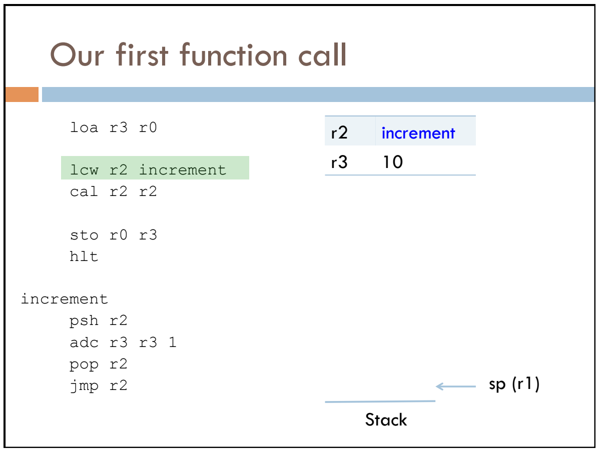
```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	
r3	10

lcw	R-W	dest = arg

sp (r1)

Stack



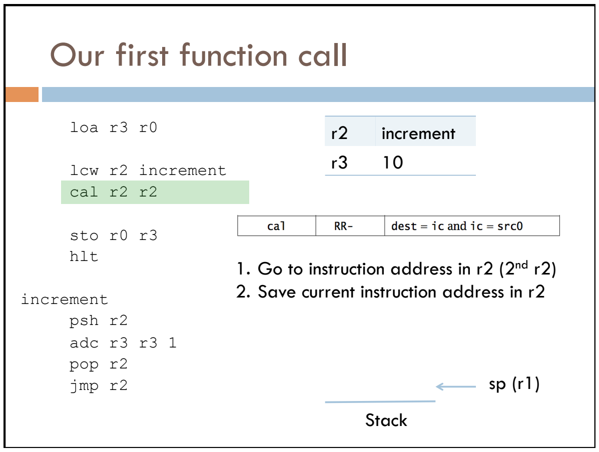
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

increment
    psh r2
    adc 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 r0 r3
    hlt

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

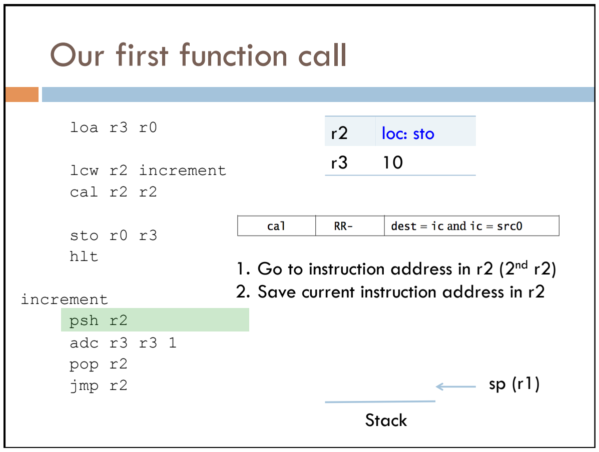
r2	increment
r3	10

cal	RR-	dest = ic and ic = src0

1. Go to instruction address in r2 (2nd r2)
2. Save current instruction address in r2

sp (r1)

Stack



Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

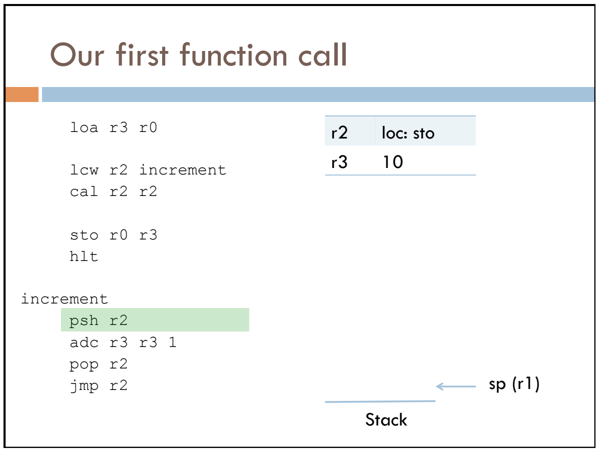
r2	loc: sto
r3	10

cal	RR-	dest = ic and ic = src0

1. Go to instruction address in r2 (2nd r2)
2. Save current instruction address in r2

sp (r1)

Stack



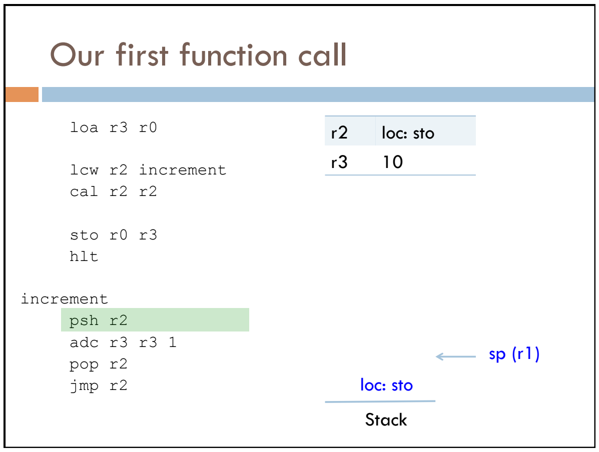
Our first function call

```
     loa r3 r0

     lcw r2 increment
     cal r2 r2

     sto r0 r3
     hlt

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

r2	loc: sto
r3	10

sp (r1)

Stack



Our first function call

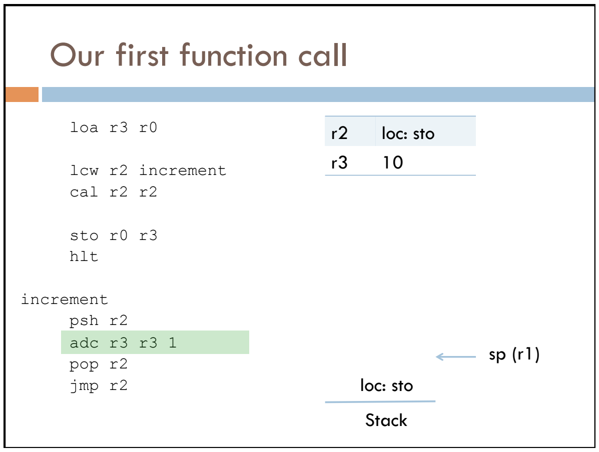
```
     loa r3 r0

     lcw r2 increment
     cal r2 r2

     sto r0 r3
     hlt

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

r2	loc: sto
r3	10

sp (r1)

loc: sto

Stack



Our first function call

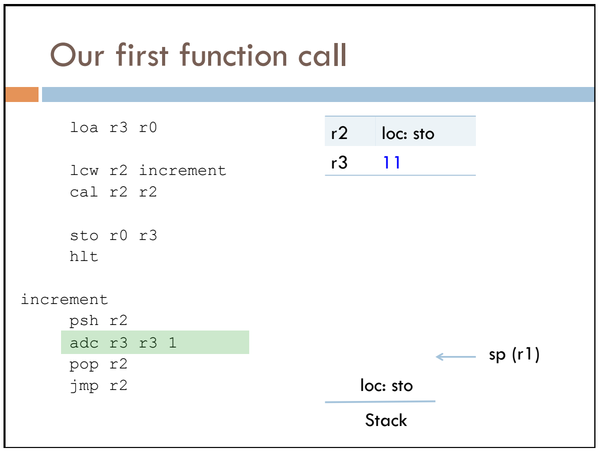
```
     loa r3 r0

     lcw r2 increment
     cal r2 r2

     sto r0 r3
     hlt

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

r2	loc: sto
r3	10

sp (r1)

loc: sto

Stack



Our first function call

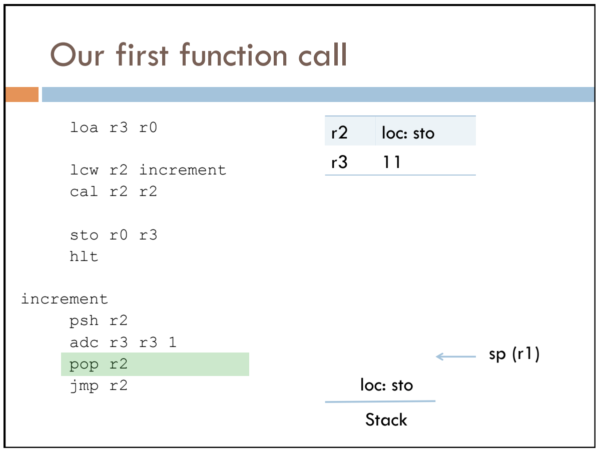
```
     loa r3 r0

     lcw r2 increment
     cal r2 r2

     sto r0 r3
     hlt

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

r2	loc: sto
r3	11

sp (r1)

loc: sto

Stack



Our first function call

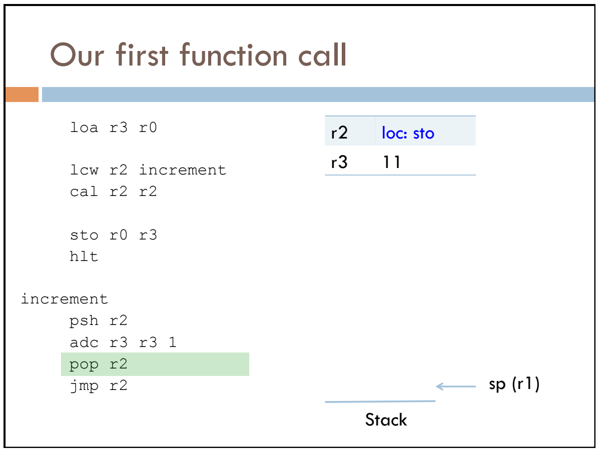
```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	loc: sto
r3	11

sp (r1)

loc: sto

Stack



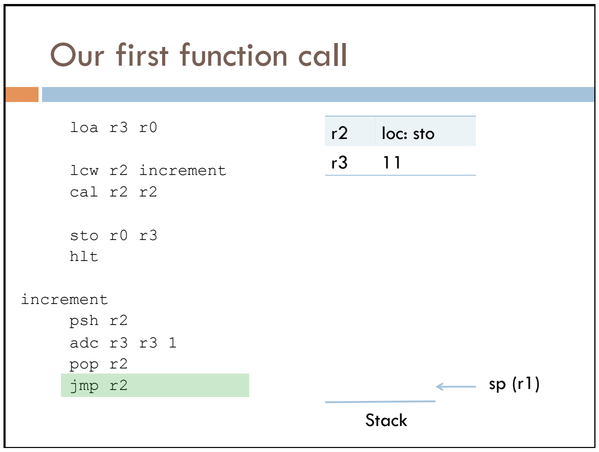
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	loc: sto
r3	11

sp (r1)

Stack



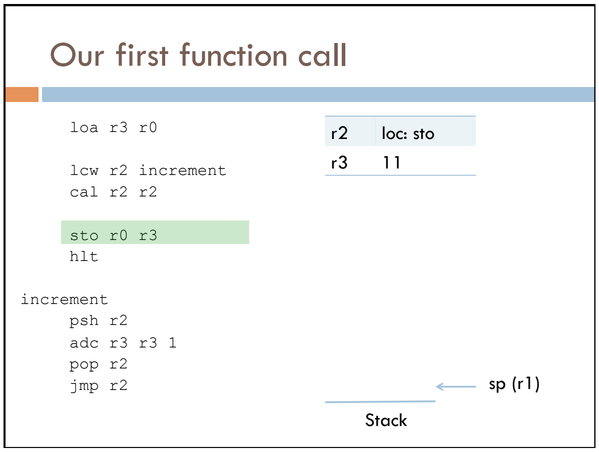
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	loc: sto
r3	11

sp (r1)

Stack



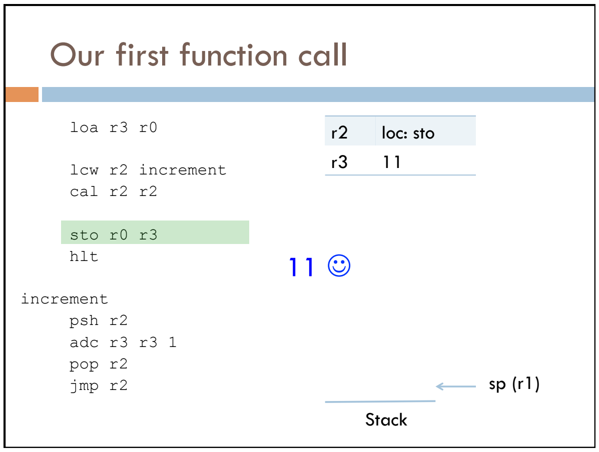
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	loc: sto
r3	11

sp (r1)

Stack



Our first function call

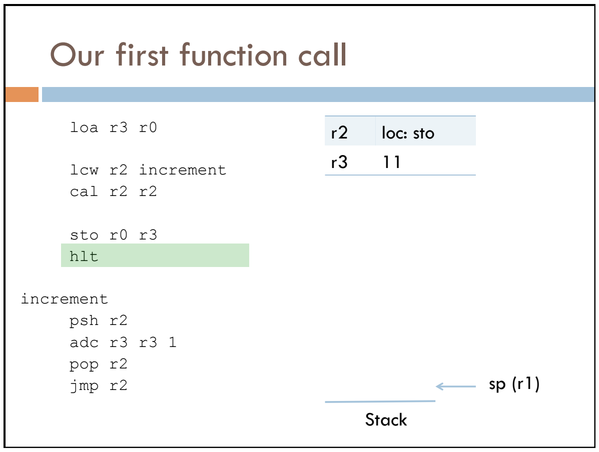
```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

r2	loc: sto
r3	11

11 ☺

sp (r1)

Stack



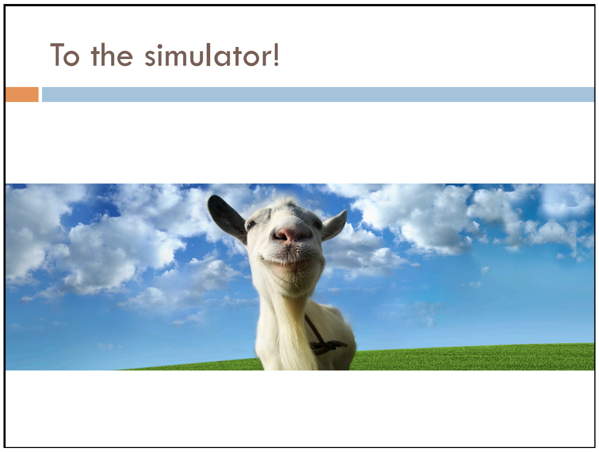
Our first function call

```
    loa r3 r0

    lcw r2 increment
    cal r2 r2

    sto r0 r3
    hlt

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

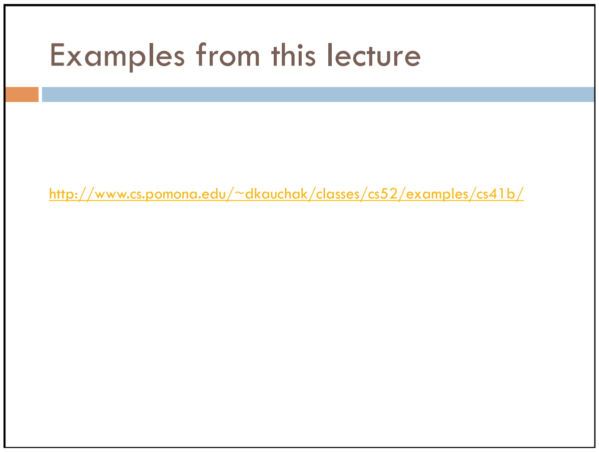
r2	loc: sto
r3	11

sp (r1)

Stack



To the simulator!



Examples from this lecture

http://www.cs.pomona.edu/~dkauchak/classes/cs52/examples/cs41b/