CS52 MACHINE: CALLING FUNCTIONS

David Kauchak
CS 54 – Fall 2022

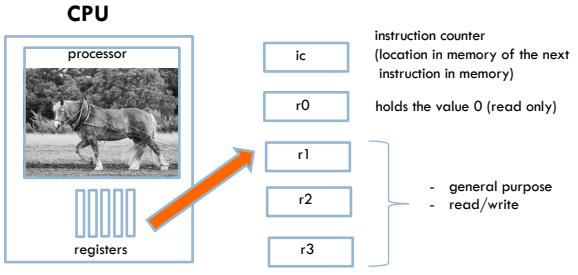
1

Examples from this lecture

https://cs.pomona.edu/classes/cs54/examples/cs52machine/

2

CS52 machine



3

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.

4

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

5

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

6

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
```

7

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
```

```
if( r3 < 0 ){
  r2 = -1

}else if( r3 != 0 ){
  r2 = 1

}else{
  r2 = 0
}
```

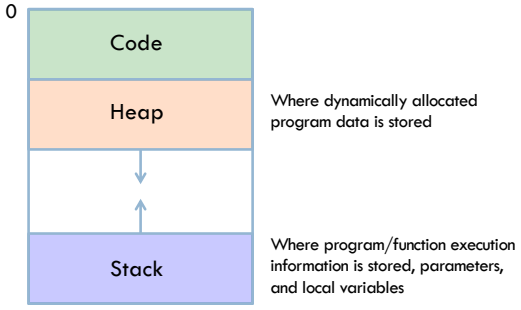
8

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
```

9

Memory layout



10

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?

11

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
```

12

Stack frame

Key unit for keeping track of a function call

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

13

Stack frames

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

What is sum 2?

Stack

14

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

15

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

16

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

17

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

18

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

19

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?

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

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

sum 2 — sum: x = 2, return: shell

Stack

20

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)
- substitute the function call with the return value
- pop the stack frame off the stack

sum 0

sum 1

sum 2

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

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

sum:
x = 2
return: shell

Stack

21

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)
- substitute the function call with the return value
- pop the stack frame off the stack

sum 1

sum 2

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

sum:
x = 2
return: shell

Stack

22

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)
- substitute the function call with the return value
- pop the stack frame off the stack

sum 1

sum 2

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

sum:
x = 2
return: shell

Stack

23

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)
- substitute the function call with the return value
- pop the stack frame off the stack

sum 2

sum:
x = 2
return: shell

Stack

24

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)
- substitute the function call with the return value
- pop the stack frame off the stack

sum 2

sum:
x = 2
return: shell

Stack

25

Stack frames

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

2 1

```
- sum 2
```

Where do we return to?

sum 2

sum:
x = 2
return: shell

Stack

26

Stack frames

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

- sum 2
val it = 3 : int
```

Stack

27

Function calls in assembly

For high-level languages the stack is managed for you

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

Stack

28

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

29

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

30

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

31

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
```

32

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

33

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

34

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

35

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

36

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

37

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

38

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)

sp (r1) ←

Stack

39

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

40

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

41

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

42

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

43

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

44

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: loc: sto

45

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

46

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

47

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

48

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

49

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

50

To the simulator!

look at increment.a52 code

51

Midterm 1 stats

Mean: 39 (88.7%)

Quartile 1: 42 (95.5%)
Quartile 2: 40.3 (91%)
Quartile 3: 38.1 (87%)

52