CS52 MACHINE: CALLING FUNCTIONS

Examples from this lecture

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

CS52 machine

CPU

- instruction counter (location in memory of the next instruction in memory)
- holds the value 0 (read only)
- general purpose register read/write

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[i c] for use as an instruction.
- The machine increments the value in i c 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
add r2 r0 -1
bres endif
elif
beq r3 r0 else
add r2 r0 1
bres 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
}

What does this code do?

```
\[ \begin{array}{l}
\text{bge } r3 \text{ } r0 \text{ } \text{elif} \\
\text{add } r2 \text{ } r0 \text{ } -1 \\
\text{brs} \\
\text{endif} \\
\text{elif} \\
\text{beq } r3 \text{ } r0 \text{ } \text{else} \\
\text{add } r2 \text{ } r0 \text{ } 1 \\
\text{brs} \\
\text{endif} \\
\text{else} \\
\text{add } r2 \text{ } r0 \text{ } 0 \\
\text{endif} \\
\text{sto } r2 \text{ } r0 \\
\text{hlt}
\end{array} \]
```

Memory layout

- **Code**: Where dynamically allocated program data is stored
- **Heap**: Where program/function execution information is stored, parameters, and local variables
- **Stack**: Where dynamically allocated program data is stored

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
  - push r3
  - add r3 r0 0
  - pop r3
  - store r3 r0

  **What will be printed out?**

  - add r3 r0 8
  - push r3
  - add r3 r0 8
  - pop r3
  - store r3 r0

  **What will be printed out?**
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 frames

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

How do we evaluate this?

Stack frames

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

How do we evaluate this?

Stack
Stack frames

10/3/22

Stack frames

fun sum 0 = 0
  | sum x = x + sum (x-1);  
  Make another function call

- sum 2

  sum 2
  x = 2
  return: shell

  Stack

17

Stack frames

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

- sum 2

  How do we evaluate this?

  sum 1
  x = 1
  return: sum (2^{th} line)

  sum 2
  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
  x = 1
  return: sum (2^{th} line)

  sum 2
  x = 2
  return: shell

  Stack

19

Stack frames

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

- sum 2

  sum 0
  x = 0
  return: sum (2^{th} line)

  sum 1
  x = 1
  return: sum (2^{th} line)

  sum 2
  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
- substitute the function call with the return value
- pop the stack frame off the stack

Stack

Stack frames

fun sum 0 = 0
| sum x = x + sum (x-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

Stack

Stack frames

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

- sum 1

Stack

Stack frames

fun sum 0 = 0
| sum x = x + sum (x-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

Stack
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

Where do we return to?

For high-level languages the stack is managed for you

In assembly we will manage the 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

```assembly
fname
  psh r2 ; save return address on stack
  ... ; do work using r3 as argument
  pop r2 ; put result in r3
  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

Our first function call

```assembly
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

```assembly
fname
  psh r2         ; save return address on the stack
  add r3 r3 l    ; add l 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

1. `loa r3 r0`  
2. `lcw r2 increment`  
3. `cal r2 r2`  
4. `sto r3 r0`  
5. `hlt`  

Increment
- `psh r2`  
- `add r3 r3 1`  
- `pop r2`  
- `jmp r2`

Stack: `sp (r1)`

--  

33

Our first function call

1. `loa r3 r0`  
2. `lcw r2 increment`  
3. `cal r2 r2`  
4. `sto r3 r0`  
5. `hlt`  

Increment
- `psh r2`  
- `add r3 r3 1`  
- `pop r2`  
- `jmp r2`

Stack: `sp (r1)`

--  

34

Our first function call

1. `loa r3 r0`  
2. `lcw r2 increment`  
3. `cal r2 r2`  
4. `sto r3 r0`  
5. `hlt`  

Increment
- `psh r2`  
- `add r3 r3 1`  
- `pop r2`  
- `jmp r2`

Stack: `sp (r1)`

--  

35

Our first function call

1. `loa r3 r0`  
2. `lcw r2 increment`  
3. `cal r2 r2`  
4. `sto r3 r0`  
5. `hlt`  

Increment
- `psh r2`  
- `add r3 r3 1`  
- `pop r2`  
- `jmp r2`

Stack: `sp (r1)`

--  

36
Our first function call

```
loa r3 r0
lcw r2 increment
sta r3 r0

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

Stack

sp (r1)

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)

"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

loa r3 r0
loc sto
r2
r3 11

loa r3 r0
psh r2
add r3 r3 1
pop r2
jmp r2

stack
sp (r1)
loc sto
r3 11

stack
sp (r1)
loc sto
r3 11
Our first function call

```
loa r3 r0
lcw r2 increment
cal r2 r2
sto r3 r0
hltr2 increment
pshr2
add r3 r3 1
pop r2
jmp r2
```

Stack

```
11
```

```
sto r3 11
```

Stack

To the simulator!

Midterm 1 stats

- Mean: 39 (88.7%)
- Quartile 1: 42 (95.5%)
- Quartile 2: 40.3 (91%)
- Quartile 3: 38.1 (87%)