CS52 CALLING FUNCTIONS

Examples from this lecture

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

CS52 machine

CPU

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

- holds the value 0 (read only)

- general purpose
- 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(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 ...

- 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 z2 r0 -1
bra endif
elif
beq r3 r0 else
add z2 r0 1
bra endif
else
add z2 r0 0
endif
sto r2 r0
hlt

What does this code do?

bge r3 r0 elif
add r2 r0 -1
bra endif
elif
beq r3 r0 else
add r2 r0 1
bra endif
else
add r2 r0 0
endif
sto r2 r0
hlt
if( r3 < 0 ){
    r2 = -1
} elsif
    r2 = 1
else{
    r2 = 0
}
What does this code do?

```
bge r3 r0 0; 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

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Where dynamically allocated program data is stored</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Where program/function execution information is stored, parameters, and local variables</td>
</tr>
</tbody>
</table>
```

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 (\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

Stack

```
sum: x = 2
return: shell
```

Stack frames

```
fun sum (\x. x + sum (x-1));
```

- `sum 2`

How do we evaluate this?

Stack

```
sum: x = 2
return: shell
```
Stack frames

<table>
<thead>
<tr>
<th>Stack frames</th>
</tr>
</thead>
</table>
| **fun** $\text{sum } 0 = 0$
| $\text{ 1 sum } x = x + \text{ sum } (x-1);$
| Make another function call
| - sum 2
| - sum 2
| sum 2
| sum 1
| $x = 1$
| return: sum (2nd line)
| sum 2
| $x = 2$
| return: shell
| Stack |

Stack frames

<table>
<thead>
<tr>
<th>Stack frames</th>
</tr>
</thead>
</table>
| **fun** $\text{sum } 0 = 0$
| $\text{ 1 sum } x = x + \text{ sum } (x-1);$
| Make another function call
| - sum 2
| - sum 2
| sum 2
| sum 1
| $x = 1$
| return: sum (2nd line)
| sum 2
| $x = 2$
| return: shell
| Stack |

How do we evaluate this?

- sum 2

What now?

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

Stack

sum 0
sum 1
sum 2

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

Stack

sum 0
sum 1
sum 2

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

Stack

sum 0
sum 1
sum 2

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

Stack

sum 0
sum 1
sum 2

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

Stack

sum 0
sum 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

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

```plaintext
fname

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

```plaintext
fname

psh r2 ; save return address on stack
...
; do work using r3 as argument
pop r2 ; put result in r3
jep r2 ; restore return address from stack
jmp r2 ; return to caller
```

Our first function call

```plaintext
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

```plaintext
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

Stack

loa r3 r0
lcw r2 increment
cal r2 r2
sto r3 r0
hlt

increment
psh r2
add r3 r3 1
pop r2
jmp r2

Stack

lcw: put the memory address of the label into the register
Our first function call

```
loa r3 r0
lcw r2 increment
sto r3 r0
hlt
increment
psh r2
add r3 r3 1
pop r2
jmp r2
```

Stack: sp [r1]

---

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
```

Stack: sp [r1]

---

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
```

Stack: sp [r1]

---

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
```

Stack: sp [r1]

---

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
```

Stack: sp [r1]
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
Stack
```
Our first function call

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

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

Stack:
- `sp [r1]`
- `loc: sto`

Our first function call

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

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

Stack:
- `sp [r1]`
- `loc: sto`

Our first function call

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

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

Stack:
- `sp [r1]`
- `loc: sto`

Our first function call

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

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

Stack:
- `sp [r1]`
- `loc: sto`
Our first function call

10a r3 r0
1cw r2 increment
cal r2 r2
sto r3 r0
hit
increment
psh r2
add r3 r3 1
pop r2
jmp r2

Stack

Our first function call

10a r3 r0
1cw r2 increment
cal r2 r2
sto r3 r0
hit
increment
psh r2
add r3 r3 1
pop r2
jmp r2

Stack

To the simulator!

look at increment.a32 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
  bns 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

Function startup
  base cases
  recursive case
  answer calculation
Function cleanup
  and return

Notice symmetry of psh and pop
Why \texttt{psh r3}? 

We're about to make a function call 
- The result of that call will go into \texttt{r3} 
- so we'll lose what's in there if we don't save it!

\texttt{x + sum (x-1)}
```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```

```asm
sum:
    psh r2
    bgt r3 r0 recurse
    add r3 r0 0
    brs done
recurse:
    psh r3
    sub r3 r3 1
    lcw r2 sum
    ca1 r2 r2
    pop r2
    add r3 r3 r2
done:
    pop r2
    jmp r2

Stack
```
sum  
  pah r2  
  bgt r3 r0 recurse  
  add r3 r0 0  
  brs done  

recurse  
  pah r3  
  sub r3 r3 1  
  lcw r2 sum  
  r2 r2  
  pop r2  
  add r3 r3 r2  

done  
  pop r2  
  jmp r2  

Stack

r2  locsum  
  r3  0

Make a recursive call:  
sum 0

l2  locsum  
  r3  0

sum  
  pah r2  
  bgt r3 r0 recurse  
  add r3 r0 0  
  brs done  

recurse  
  pah r3  
  sub r3 r3 1  
  lcw r2 sum  
  r2 r2  
  pop r2  
  add r3 r3 r2  

done  
  pop r2  
  jmp r2  

Stack

r2  locsum  
  r3  0

l2  locsum  
  r3  0

sum  
  pah r2  
  bgt r3 r0 recurse  
  add r3 r0 0  
  brs done  

recurse  
  pah r3  
  sub r3 r3 1  
  lcw r2 sum  
  r2 r2  
  pop r2  
  add r3 r3 r2  

done  
  pop r2  
  jmp r2  

Stack

r2  loccal1  
  r3  0
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Registers</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pah r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgt r3 r0</code></td>
<td>r2, r3, r0</td>
<td></td>
</tr>
<tr>
<td><code>recursion</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pah r3</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub r3 r3 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lcw r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sum</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>cal r2 r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pop r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add r3 r3 r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>done</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pop r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jmp r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Stack</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Return
- Returning with answer in r3

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Registers</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pah r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgt r3 r0</code></td>
<td>r2, r3, r0</td>
<td></td>
</tr>
<tr>
<td><code>recursion</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pah r3</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub r3 r3 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lcw r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sum</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>cal r2 r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pop r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add r3 r3 r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>done</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pop r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>jmp r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Stack</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Return
- Returning with answer in r3
```

---
Why are we doing this?

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

Stack
sum  
psh r2  
bgt r3 r0  
recurse  
add r3 r0  
brs done  

recurse  
psh r3  
sub r3 r3  
local  
lcw r2 sum  
cal r2 r2  
pop r2  
add r3 r3 2  
done  
pop r2  
jmp r2  

Stack

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

recurse  
psh r3  
sub r3 r3  
lcw r2 sum  
cal r2 r2  
pop r2  
add r3 r3 2  
done  
pop r2  
jmp r2  

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

Returning with answer in r3

Calling sum
  loa r3 r0
  lcw r2 sum
cal r2 r2
  sto r3 r0
  hit

Print the answer: 3!

Calling sum
  loa r3 r0
  lcw r2 sum
cal r2 r2
  sto r3 r0
  hit

Print the answer: 3!
Calling sum

```
loa r3 r0
lwc r2 sum
cal r2 r2
sto r3 r0
hlt
```

Calling sum

```
r2  local0
r3  3
```

Real structure of CS52 program

```
; great comments at the top!
; Save address of highest end
; (highest address) of the stack in r1
instruction1 ; comment
instruction2 ; comment
...
hlt
;
; stack area: 50 words
;
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/~dkaudhuk/classes/cs52/examples/cs52machine/

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