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

Instruction counter  
( location in memory of the next  
instruction in memory)  
holds the value 0 (read only)

- general purpose  
- read/write

processor

registers

r0  
r1  
r2  
r3
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[1c] for use as an instruction.
- The machine increments the value in 1c by 2.
- The machine decodes and carries out the instruction.

Notice that, while an instruction is being executed, the value in 1c is the address of the next instruction. This detail is important in implementing branch instructions.

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>arguments</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>RR-</td>
<td>dest = src0</td>
</tr>
<tr>
<td>neg</td>
<td>RR-</td>
<td>dest = src0</td>
</tr>
<tr>
<td>add</td>
<td>RRR</td>
<td>dest = src0 + src1</td>
</tr>
<tr>
<td>sub</td>
<td>RRR</td>
<td>dest = src0 - src1</td>
</tr>
<tr>
<td>adc</td>
<td>RRS</td>
<td>dest = src0 + arg</td>
</tr>
<tr>
<td>sbc</td>
<td>RRS</td>
<td>dest = src0 - arg</td>
</tr>
</tbody>
</table>
operation arguments
R = register (e.g. r0)
S = signed number (byte)

table:

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>arguments</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>RR-</td>
<td>dest = src0</td>
</tr>
<tr>
<td>neg</td>
<td>RR-</td>
<td>dest = -src0</td>
</tr>
<tr>
<td>add</td>
<td>RRR</td>
<td>dest = src0 - src1</td>
</tr>
<tr>
<td>sub</td>
<td>RRR</td>
<td>dest = src0 - src1</td>
</tr>
<tr>
<td>adc</td>
<td>RRS</td>
<td>dest = src0 + arg</td>
</tr>
<tr>
<td>sbc</td>
<td>RRS</td>
<td>dest = src0 - arg</td>
</tr>
</tbody>
</table>

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

table:

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>arguments</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>RR-</td>
<td>dest = src0</td>
</tr>
<tr>
<td>neg</td>
<td>RR-</td>
<td>dest = -src0</td>
</tr>
<tr>
<td>add</td>
<td>RRR</td>
<td>dest = src0 - src1</td>
</tr>
<tr>
<td>sub</td>
<td>RRR</td>
<td>dest = src0 - src1</td>
</tr>
<tr>
<td>adc</td>
<td>RRS</td>
<td>dest = src0 + arg</td>
</tr>
<tr>
<td>sbc</td>
<td>RRS</td>
<td>dest = src0 - arg</td>
</tr>
</tbody>
</table>

adc r1 r0 8
neg r2 r1
sub r2 r1 r2

What number is in r2?

adc r1 r0 8
neg r2 r1
sub r2 r1 r2

r1 = 8
r2 = -8, r1 = 8
r2 = 16
<table>
<thead>
<tr>
<th>abbreviation</th>
<th>arguments</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sto</strong></td>
<td>RR[5]</td>
<td>mem(dest + arg) - src0</td>
</tr>
<tr>
<td><strong>loa</strong></td>
<td>RR[5]</td>
<td>dest = mem(src0 + arg)</td>
</tr>
</tbody>
</table>

**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

**Basic structure of CS41B program**

```plaintext
; great comments at the top!

; instruction1 ; comment
instruction2 ; comment
...

hlt
end
```

**Running the CS41B machine**

Look at subtract.a41
- load two numbers from the user
- subtract
- print the result
Different windows
- Memory (left)
- Instruction execution (right)
- Registers
- I/O and running program

Modify \( ic \), the instruction counter... which changes the flow of the program!

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>arguments</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{brs}</td>
<td>\textit{--5 \ $tc = loc + arg$}</td>
<td>\textit{Branch Instructions}</td>
</tr>
<tr>
<td>\textit{beq}</td>
<td>\textit{RRS \ if dest = src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
<tr>
<td>\textit{bne}</td>
<td>\textit{RRS \ if dest # src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
<tr>
<td>\textit{blt}</td>
<td>\textit{RRS \ if dest &lt; src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
<tr>
<td>\textit{ble}</td>
<td>\textit{RRS \ if dest \leq src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
<tr>
<td>\textit{bgt}</td>
<td>\textit{RRS \ if dest &gt; src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
<tr>
<td>\textit{bge}</td>
<td>\textit{RRS \ if dest \geq src0, \ $tc = loc + arg$}</td>
<td></td>
</tr>
</tbody>
</table>

What do these operations do?

\textit{beq \ r3 \ r0 \ done}

What does this do?
beq r3 r0 done
If r3 = 0, branch to the label “done”
if not (else) ic is incremented as normal to
the next instruction

ble r2 r3 done
If r2 <= r3, branch to the label done

- 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?

```assembly
bge r3 r0
bge r3 r0
beq r3 r0
beq r3 r0
add r2 r0
add r2 r0
add r2 r0
add r2 r0
```

```assembly
else
else
else
```

```assembly
if( r3 < 0 ) {
  r2 = -1
} else if( r3 != 0 ) {
  r2 = 1
} else {
  r2 = 0
}
```

```assembly
add r2 r0
add r2 r0
add r2 r0
add r2 r0
```

```assembly
sto r0 r2
sto r0 r2
```

```assembly
hlt
hlt
```

end

end
What does this code do?

```
bge r3 r0 ifif ; if r3 >= 0 go to elseif
sb r2 r0 1 ; r3 < 0; r2 = -1
br endif ; jump to end of if/elseif/else
elseif
beq r3 r0 else ; if r3 = 0 go to elseif
adc r2 r0 1 ; r3 > 0; r2 = 1
br endif ; jump to end of if/elseif/else
else
add r2 r0 r0 ; r3 = 0; r2 = 0
endif
sto r0 r2 ; print out r2
hit
end
```

Your turn 😊

Write some code that prints out abs(r3)

```
mov RR- dest = ssrc0
neg RR- dest = -ssrc0
add RR- dest = src0 + src1
sub RR- dest = src0 - src1
adc RR- dest = src0 + arg
sbc RR- dest = src0 - arg
```

abs

Look at abs.a41

Memory layout

- **Code**: Where dynamically allocated program data is stored.
- **Heap**: 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>R--</td>
</tr>
<tr>
<td>pop</td>
<td>R--</td>
</tr>
</tbody>
</table>

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

```assembly
frame
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
push 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
    push 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
    push 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
    push r2
    adc r3 r3 1
    pop r2
    jmp r2
```

Stack

```
sp (r1)
```

r2

r3

10

r2

r3

sp (r1)
1. Go to instruction address in r2 (2nd r2)
2. Save current instruction address in r2
### Our first function call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>loa r3 r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cal r2 r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sto r0 r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increment:
- push r2
- adc r3 r3 1
- pop r2
- jmp r2

- sp [r1]

**Stack**

### Our first function call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>loa r3 r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cal r2 r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sto r0 r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increment:
- push r2
- adc r3 r3 1
- pop r2
- jmp r2

- sp [r1]

**Stack**

### Our first function call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>loa r3 r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cal r2 r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sto r0 r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increment:
- push r2
- adc r3 r3 1
- pop r2
- jmp r2

- sp [r1]

**Stack**

### Our first function call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>loa r3 r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cal r2 r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sto r0 r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increment:
- push r2
- adc r3 r3 1
- pop r2
- jmp r2

- sp [r1]

**Stack**
Our first function call

loa r3 r0
loa r2 increment
cal r2 r2
sto r0 r3
hlt
increment
psh r2
adc r3 r3 1
pop r2
jmp r2
loc. sto
Stack

r2
loc. sto
r3 11

Stack

Our first function call

loa r3 r0
loa r2 increment
cal r2 r2
sto r0 r3
hlt
increment
psh r2
adc r3 r3 1
pop r2
jmp r2
loc. sto
Stack

r2
loc. sto
r3 11

Stack
Our first function call

```
load r3 r0
load r2 increment
call r1 r2
store r0 r3
halt
increment
push r2
add r3 r3 1
pop r2
jump r2
```

Stack

`sp (r1)`

```
load r3 r0
load r2 increment
call r1 r2
store r0 r3
halt
increment
push r2
add r3 r3 1
pop r2
jump r2
```

Stack

`sp (r1)`

To the simulator!

Examples from this lecture

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