Last time

- Introduced assembly code
    - Spent a fair bit of time on labels and how they work.
    - Looked at writing "if" statements
    - Started in on functions.
- Discussed how assembly code is translated to machine code.
    - How could write a program with just ".data" instruction (if needed, it wouldn't be fun).

Today

- Loops
- Function calls
- Discuss midterm
- Start on I/O devices
- *Maybe* get to 2D arrays ("arrays of arrays")

E100 assembly

- Describe how you would write a loop in assembly.

- Let's write a program that sums the first "N" integers (so 1 to N) using a loop.
    - In C
    - In e100 assembly

Function calls

Functions are fundamental to programming.

- They provide modularity
- The allow code reuse

Machine code ***must*** support functions. After all, our high-level languages use them all the time and those languages all get translated into machine code.

Let's take our "diff" code from before and write a function which returns the difference of two numbers (always positive as before).

We are going to make a few changes now.

1. Because all labels have "global scope," we need to make sure each label is unique. So we follow a *convention* of having all labels in a function start with the function name and then an underscore. So instead of "x" we will have "diff_x" in a function named "diff"
2. Rather than ending in a halt, we will end with a "ret" instruction.
3. We need to know where to pass values in and where to get the return value(s).

Let's convert the code "diff" into the function "diff"

```
         blt less x y
         sub result x y
         be end 0 0
less     sub result y x
end      halt
result   .data 0
x        .data 0
y        .data 0
```

Now, how would we call it? Say we want the diff of two values: "M" and "N" and get the result into "P".

Question: Do functions help performance?

Assembly function writing—practice with bits!

Write a function which returns the “Nth” bit of a number. So if our number is 010001 the rightmost bit is the “0th” bit (an so is a 1) while the “1st” bit is the next rightmost (so is a 0). The function should be named “bs” (bit select) and it takes two arguments: bs_X (the number) and bs_N (the bit number). Its return value is to be in “bs_result”.

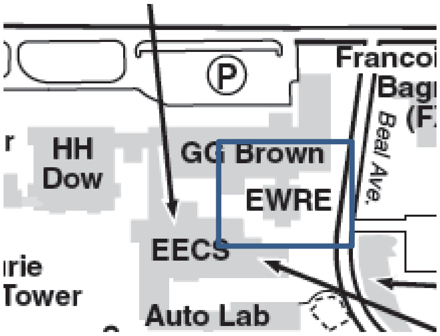
Let’s talk about how to do this from an algorithmic viewpoint first.

- Probably the easiest way is to right shift the value “N” bits and then “AND” it with 1.
    - So if I want the 4th right-most bit of 010001 (where we start counting at 0…) we could right shift by 4 (so 000001) and then AND it with 1 getting 000001. If we went for the 2nd bit, what would happen?

So write that code!

Administration

- This week’s lab is all about implementing the missing instructions from the e100.
    - You would *really* like to have time to start on lab 7. So try to be done with something you think works before your lab starts.
- Saturday I will hold a review/Q&A session
    - 5-6:30pm in 2311 EECS
- Sunday I will hold another one
    - 3-4:30 in 2311 EECS
- Those that are meeting me on Sunday will need to call to get let up to my office
    - 734 764 0525 is my office number.
- Monday’s lecture will be split by the two instructors
    - Making up for the snow day…
- Wednesday is the exam
    - EWRE 185 during our normal class time.
    - Be sure you can find it!



- Next week’s lab is the hard one.
    - Be ready for it. Read and try to get something up and running first. ASE100 works, or come to office hours!

I/O devices

Speaking to I/O devices isn't something we worry too much about when programming a computer. You might be writing to the screen with "cout" and reading from the keyboard with "cin". And maybe you are doing some file I/O (reading and writing files). But that's generally about all you'd do.

But there is a *lot* of "magic" (opaque level of abstraction if you prefer…) that is going on. So we'll explore how this works on the e100. As you'll see, the e100 has some fairly sophisticated I/O capabilities.

Memory-mapped I/O

This is a bit of a weird notion, so hang on. We need a way to talk to our I/O devices. The way it is generally done is "memory-mapped I/O". What this means is that certain memory locations are reserved for talking to certain I/O devices. We'll talk about two different types of devices, the "trivial" ones and the standard ones.

"Trivial" I/O devices

There are a number of devices you can just read or write to directly from the e100. They include the switches, LEDs, and HEX displays.

Address	Allowed access	Definition	Use
0x80000000	read	bits 17-0: SW[17:0]	binary input
0x80000001	write	bits 17-0: LED_RED[17:0]	binary output
0x80000002	write	bits 7-0: LED_GREEN[7:0]	binary output
0x80000003	write	bits 15-0: HEX3-HEX0	hexadecimal output
0x80000004	write	bits 15-0: HEX7-HEX4	hexadecimal output
0x80000005	read	bits 31-0: real-time clock	measure time

If you want to turn on LED_RED[0] and turn the rest off, you just write a "1" to location 0x80000001.

Practice: Write e100 assembly code that turns on LED_RED[4] and LED_RED[5] turning the rest off.

Other I/O devices (text from lab 7)

At first glance, it seems easy to send data to/from an I/O device. For example, one could send a sample to the speaker simply by copying data to the speaker_sample device register. However, this is not quite enough to send a **sequence** of values. One problem is that the speaker controller doesn't know when the program has sent the next value to thespeaker_sample device register. Another problem is that the program doesn't know when the speaker controller has read the last sample and is ready to receive the next sample. These problems are addressed by an I/O device protocol.

A protocol is used to guide the interaction between two parties. In the context of I/O devices, an I/O protocol is used to guide the interaction between an E100 program and an I/O device. A protocol defines the steps involved in the interaction and includes how each party knows when the current step is complete. We will use a protocol to send commands to an I/O device and receive the response from that device.

The part of an E100 program that implements the E100's side of an I/O protocol is called a **device driver**. The I/O protocol uses four types of signals to allow an E100 program to send commands to a device and receive the response from that device.

- **command parameters**: These signals carry the data that the E100 program wants to send to the device as part of the command. These signals are set by the E100 program.
- **command**: The value of this signal is set by the E100 program. When it is 1, it tells the device that the E100 program is done setting the command parameters and is ready for the device to carry out the command.
- **response parameters**: These signals carry the data that the device wants to send to the E100 program as part of its response. These signals are set by the device.
- **response**: The value of this signal is set by the device. When it is 1, it tells the E100 program that the device has executed the command and is sending its response to that command.

The steps involved in sending data to an output device are:

command	**response**	**Description**
0	0	System is idle.
1	0	E100 program sets the **command parameters** to describe the desired command, then sets **command** to 1 to ask the device to execute the command. After setting **command** to 1, the E100 program waits for device to execute the command.
1	1	After the device executes the command, it sets the **response parameters** for the command, then sets **response** to 1 to tell the E100 program that it has executed the command and is sending back the response. After setting **response** to 1, the device waits for the E100 program to set **command** to 0.
0	1	E100 program sets **command** to 0 to tell the device that the program has seen the device's response. After this state, the device sets **response** back to 0, and the system returns to the Idle state.

List of devices and their interfaces

Address	Allowed access	Definition	Use
0x80000010	write	bit 0: lcd_command	LCD display
0x80000011	read	bit 0: lcd_response	
0x80000012	write	bits 3-0: lcd_x[3:0]	
0x80000013	write	bit 0: lcd_y	
0x80000014	write	bits 7-0: lcd_ascii[7:0]	
0x80000020	write	bit 0: ps2_command	PS/2 keyboard
0x80000021	read	bit 0: ps2_response	
0x80000022	read	bit 0: ps2_pressed	
0x80000023	read	bits 7-0: ps2_ascii[7:0]	
0x80000030	write	bit 0: sdram_command	SDRAM memory
0x80000031	read	bit 0: sdram_response	
0x80000032	write	bit 0: sdram_write	
0x80000033	write	bits 24-0: sdram_address[24:0]	
0x80000034	write	bits 31-0: sdram_data_write	
0x80000035	read	bits 31-0: sdram_data_read	
0x80000040	write	bit 0: speaker_command	speaker
0x80000041	read	bit 0: speaker_response	
0x80000042	write	bits 31-0: speaker_sample	
0x80000050	write	bit 0: microphone_command	microphone
0x80000051	read	bit 0: microphone_response	
0x80000052	read	bits 31-0: microphone_sample	
0x80000060	write	bit 0: vga_command	VGA monitor
0x80000061	read	bit 0: vga_response	
0x80000062	write	bit 0: vga_write	
0x80000063	write	bits 9-0: vga_x1[9:0]	
0x80000064	write	bits 9-0: vga_y1[9:0]	
0x80000065	write	bits 9-0: vga_x2[9:0]	
0x80000066	write	bits 9-0: vga_y2[9:0]	
0x80000067	write	bits 14-0: vga_color_write[14:0]	
0x80000068	read	bits 14-0: vga_color_read[14:0]	

0x80000070	write	bit 0: mouse_command	USB mouse/touchscreen
0x80000071	read	bit 0: mouse_response	
0x80000072	read	bits 31-0: mouse_deltax	
0x80000073	read	bits 31-0: mouse_deltay	
0x80000074	read	bit 0: mouse_button1	
0x80000075	read	bit 0: mouse_button2	
0x80000076	read	bit 0: mouse_button3	
0x80000080	write	bit 0: sd_command	SD card
0x80000081	read	bit 0: sd_response	
0x80000082	write	bits 0: sd_write	
0x80000083	write	bits 29-0: sd_address[29:0]	
0x80000084	write	bits 31-0: sd_data_write	
0x80000085	read	bits 31-0: sd_data_read	
0x80000090	write	bit 0: serial_receive_command	serial communication (wired and wireless)
0x80000091	read	bit 0: serial_receive_response	
0x80000092	read	bits 7-0: serial_receive_data[7:0]	
0x800000a0	write	bit 0: serial_send_command	
0x800000a1	read	bit 0: serial_send_response	
0x800000a2	write	bits 7-0: serial_send_data[7:0]	
0x800000b0	write	bit 0: camera_command	camera
0x800000b1	read	bit 0: camera_response	
0x800000b2	write	bits 9-0: camera_x[9:0]	
0x800000b3	write	bits 9-0: camera_y[9:0]	
0x800000b4	write	bits 1-0: camera_scale[1:0]	
0x800000b5	write	bit 0: camera_mirror	
0x800000c0	write	bit 0: fft_send_command	Fast Fourier Transform
0x800000c1	read	bit 0: fft_send_response	
0x800000c2	write	bits 31-0: fft_send_real	
0x800000c3	write	bits 31-0: fft_send_imaginary	
0x800000c4	write	bits 0: fft_send_inverse	
0x800000c5	write	bit 0: fft_send_end	
0x800000d0	write	bit 0: fft_receive_command	
0x800000d1	read	bit 0: fft_receive_response	
0x800000d2	read	bits 31-0: fft_receive_real	
0x800000d3	read	bits 31-0: fft_receive_imaginary	

Keyboard example

From lab 7:

> ps2_command and ps2_response implement the standard I/O protocol. There are no command parameters. The response parameters are ps2_pressed and ps2_ascii[7:0]. The response parameters represent a **keyboard event**, describing which key was acted on (ps2_ascii) and whether the action was a key press or key release (ps2_pressed). Ifps2_pressed is 1, the event was a key press. If ps2_pressed is 0, the event was a key release. ps2_ascii contains the ASCII value for the key that was pressed or released.
>
> ase100 simulates the PS/2 keyboard controller accurately enough to test your device driver and to run assembly-language programs. ase100 sees keyboard events when the mouse is in the VGA window.

Consider the following code:

```
LEDTEST

        cp 0x80000020 one
wait    bne wait 0x80000021 one
        cp 0x80000001 one
        halt

one .data 1
```

What happens when you run this?