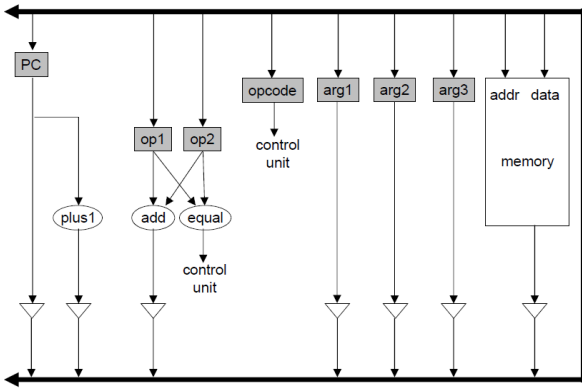
Last time

- Looked again at the datapath for the e100.
- Started in on the state machine
  - Finished the "fetch" part of the state machine
  - Designed the "add" instruction
  - Designed the "be" instruction
- Started on the "Max" assembly function
  - Mike finished up in lab



state	next_state	pc_drive	plus1_drive	add_drive	arg1_drive	arg2_drive	arg3_drive	mem_drive	pc_write	op1_write	op2_write	opcode_write	arg1_write	arg2_write	arg3_write	address_write	mem_write
reset	fetch1																
fetch1	fetch2																
fetch2	fetch3																
fetch3	fetch4																
fetch4	fetch5																
fetch5	fetch6																
fetch6	fetch7																
fetch7	fetch8																
fetch8	decode																

Review

Write a program which finds the absolute value of the data in memory location 100 and halts once that data is in memory location 100. (yes, we've done this before—it's review!)

mem[0]	
mem[1]	
mem[2]	
mem[3]	
mem[4]	
mem[5]	
mem[6]	
mem[7]	
mem[8]	
mem[9]	
mem[10]	
mem[11]	

mem[12]	
mem[13]	
mem[14]	
mem[15]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	

Assembly

What we wrote above is machine code. It is data intended to be directly run by the computer. However, it's really annoying to work with. In the real world, we generally program in high-level languages that get translated to machine code. But sometimes, we want to have complete control of what the computer does, but don't want to write in machine code. So, as you saw in lab, we use assembly.

Let's consider the following program written in assembly:

```
        blt 12 21 22
        sub 20 21 22
        be 16 0 0       (unconditional branch)
        sub 20 22 21
        halt
```

Question: What does that code do? How would you describe the algorithm?

That is clearly a lot better than writing things in memory. We are using the instruction name and writing the whole instruction on one line. But the numbers get old real fast. We could instead do something like this:

```
          blt less 21 22
          sub result 21 22
          be end 0 0            (unconditional branch)
less      sub result 22 21
end       halt
```

Using "less" in place of the number "12" is really helpful. Not only does it make it a lot more readable, but it also makes it easier to make changes. Before, if we added an instruction between the blt and the sub, we'd need to adjust the value of the location we were branching to (to a 16). But now, we just branch to "less". We can also do something similar for the data.

```
          blt less x y
          sub result x y
          be end 0 0            (unconditional branch)
less      sub result y x
end       halt
result    .data 0
x         .data 0
y         .data 0
```

Now we can refer to the data locations by name also. And adding more instructions or data is easy.

Formal description

The e100 assembly language format is as follows:

```
[label]     opcode  arg1    arg2    arg3
```

Fields are separated by white space (spaces or tabs).

- **`label`** associates a name with the (first) address for this line of code
  - Labels are optional.
  - Labels must start in column 0 (left justified)
  - If there is no label the line should start with white space otherwise the opcode will look like a label.
- **`opcode`** is one of: halt, add, sub, and, or, not, cp, sl, sr, mult, cpfa, cpta, be, bne, blt, call, ret
- **`arg1, arg2, arg3`** can each be a decimal number, a hexadecimal number (prefixed with 0x), or a label.
- Comments are marked by // (rest of line ignored)
- Blank lines are ignored
- unspecified locations initialized with 0
- We can initialize a location by using the ".data" directive.
  - .data followed by a number puts that number in the location.

Question: Translate the following (nonsense) program into machine code (in decimal)

```
          be bob x y
          mult tmp x y
bob       halt
result    .data 156
x         .data 0x12
```

mem[0]	
mem[1]	
mem[2]	
mem[3]	
mem[4]	
mem[5]	
mem[6]	
mem[7]	

mem[8]	
mem[9]	
mem[10]	
mem[11]	
mem[ ]	
mem[ ]	
mem[ ]	
mem[ ]	

Review questions/problems

- So what is a program?
    - A program is just how memory is initialized!
- A program that converts assembly to machine code is called an "assembler". Describe the process of writing an assembler.

- Describe how you would write an "if" statement in assembly.

- Describe how you would write a loop in assembly.

Assembly programming practice

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

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 write the function "diff"

Now, how would we call it?

Question: Do functions help performance?

Assembly function writing

Write a function which returns the "N^{th}" bit of a number. So if our number is 100010001 the rightmost bit is the "0^{th}" bit (an so is a 1) while the "1^{st}" 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".