Last time

- Introduced and explained the "MAX" circuit.
- Introduced the idea of a computer
    - "If you had one wish you'd wish for more wishes. If you can have only one hardware design, you'd want one that can solve any problem."
        - A hardware algorithm that can implement any other algorithm.
        - That's what a computer is!
- Started on a specific computer—the E100.
    - Looked at three instructions so far: add, multiply and halt. Mentioned that other similar ones exist.

Instruction name	Opcode	Effect

halt	0	`PC = PC+4`
		`stop executing instructions`

add	1	`PC = PC+4`
		`memory[arg1] = memory[arg2] + memory[arg3]`

sub	2	`PC = PC+4`
		`memory[arg1] = memory[arg2] - memory[arg3]`

mult	3	`PC = PC+4`
		`memory[arg1] = memory[arg2] * memory[arg3]`

div	4	`PC = PC+4`
		`memory[arg1] = memory[arg2] / memory[arg3]`

    - Instructions laid out in 4 memory addresses.
        - First contains the opcode.
        - Next three contain arg1, arg2 and arg3.
- Also discussed resource for projects.
    - Monitor, keyboard, speaker, mouse, etc.

Review questions

- Consider the following chunk of memory. What would memory look like after this code completes?

mem[0]	1
mem[1]	100
mem[2]	101
mem[3]	102
mem[4]	0 (HALT opcode)
mem[5]	0
mem[6]	0
mem[7]	0
mem[100]	0
mem[101]	22
mem[102]	33

- What are the distinquishing characteristics that make a computer a computer (as opposed to, say, a "max" circuit or a "rot13" circuit?

More instructions in the e100 instruction set

There is more to a computer than arithmetic. In this section we'll discuss the rest of the instruction set.

Logical operations (and copy)

Instruction name	Opcode	Effect

cp	5	`PC = PC+4`
		`memory[arg1] = memory[arg2]`

and	6	`PC = PC+4`
		`memory[arg1] = memory[arg2] & memory[arg3]`

or	7	`PC = PC+4`
		`memory[arg1] = memory[arg2] | memory[arg3]`

not	8	`PC = PC+4`
		`memory[arg1] = ~memory[arg2]`

The above instructions allow us to do logical operations (and, or, not) on the data or to copy it.

- "cp" stands for copy (cp is the standard Unix/Linux command to copy a file) and it just move data from one location to another.
- "and", "or" and "not" each do a bitwise logical operation on the data. So for example, the not of 0000111100110101 is 1111000011001010. Likewise, "and" is a bitwise and where each bit of the two arguments are ANDed together (so 1100 & 1010 is 1000) and "or" is a bitwise or (so 1100 | 1010 is 1110).

Aside: Most computers don't have a "cp" instruction. What else could they use instead? Assume there is a memory location known to hold a zero.

Shift operations

Instruction name	Opcode	Effect

sl	9	`PC = PC+4`
		`memory[arg1] = memory[arg2] << memory[arg3]`

sr	10	`PC = PC+4`
		`memory[arg1] = memory[arg2] >> memory[arg3]`

The shift operation is just another type of logical operation. But I've separated it out because sometimes it causes people grief. A shift left just shifts the bits in the number over one way or the other and fill in with zeros where new values are needed. So if you shift 0100 to the right by 1 you get 0010. If you shift it to the left by one you get 1000.

- *<u>What do you get if you shift 1001001 to the right by 2? If you shift it to the left by 2?</u>*

Branches

So what's missing? Are their programs you can't write? Think about writing a program that computes absolute value. What do we need?

We also are lacking loops—the ability to go back and do things a number of times.

First, let's talk about the PC. So far, all of our instructions have incremented it by 4. That should make sense because what we want to do is get the PC to point to the next instruction after the one we are doing. Each instruction is 4 words, so we want the PC to increment to end up pointing to the next instruction. Think of the PC as where you are reading in a recipe for something you are baking. If the recipe says "if you have already boiled the water you can skip to step 4" that direction is telling you where to go next for the next instruction. Normally, you just go to the instruction below the one you just did (so step 1, step 2, step 3, etc.) The PC just tells us which "step" we are on.

Branches give us a way to change to some other step of the recipe. And in fact they give us a *conditional* way to do that. So "be" (branch equal) says we change the PC to arg1 if memory[arg2]==memory[arg3]. It's a conditional goto!

Instruction name	Opcode	Effect

be	13	`if (memory[arg2] == memory[arg3]) {`
		`    PC = arg1`
		`} else {`
		`    PC = PC+4`
		`}`

bne	14	`if (memory[arg2] != memory[arg3]) {`
		`    PC = arg1`
		`} else {`
		`    PC = PC+4`
		`}`

blt	15	`if (memory[arg2] < memory[arg3]) {`
		`    PC = arg1`
		`} else {`
		`    PC = PC+4`
		`}`
		
		Comparisons take into account the sign of the number. E.g., 32'hffffffff (-1) is less than 32'h00000000 (0).

So what we've got are "conditional goto" statements. Each says to jump to a specific part of memory to get the next instruction if a certain condition is true. Otherwise just do the next instruction (PC=PC+4)

- *What do you suppose bne and blt stand for?*
- *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.*

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[ ]	

Arrays

There is really only one thing we lack—we don't have a good way to deal with arrays. All of the instructions we've used only read from fixed memory locations and only write to a fixed memory location. Writing something like "max" is just not very reasonable. So we need a way to index into an array.

Instruction name	Opcode	Effect

cpfa	11	`PC = PC+4`
		`memory[arg1] = memory[arg2 + memory[arg3]]`

cpta	12	`PC = PC+4`
		`memory[arg2 + memory[arg3]] = memory[arg1]`

Do you think you could find the maximum value in an array from memory location 200 to 215? How would you do it (at a high level)? We won't have time to finish this in class, but it's a good exercise to try to work through.

Hardware Algorithms

Thus far, we have implemented two hardware algorithms. MAX (in class) and rot13 (in lab).

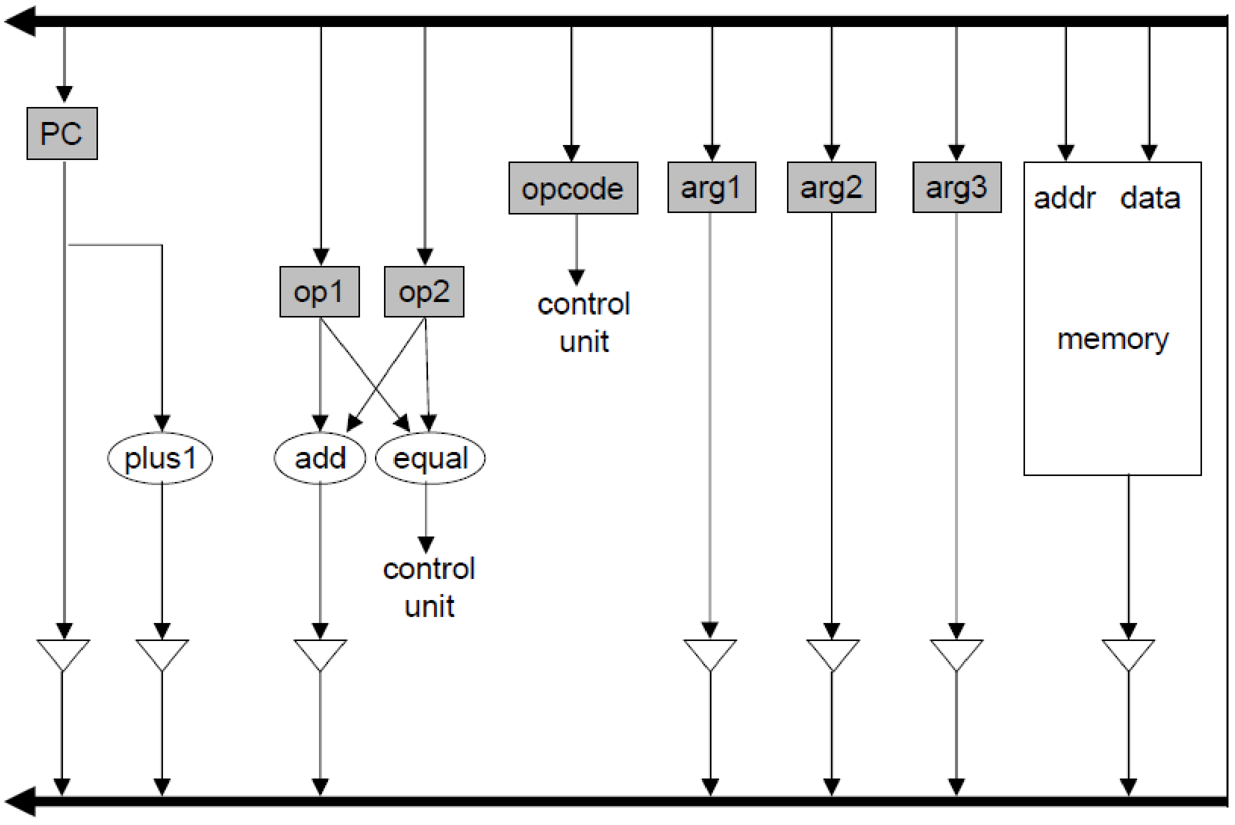
// max task (algorithm described as C++ program)

```
max = 0;  /* assume no negative numbers */
i = 0;
while (i != 16) {
    if (memory[i] > max) {
        max = memory[i];
    }
    i++;
}
```

```
// rot13 task (algorithm described as C++ program)
    i = 0;
    while (1) {
        element = memory[i]
        if (element == 0) {
            break;
        }
        if (element < 110) {
            element = element + 13;
        } else {
            element = element - 13;
        }
        memory[i] = element;
        i++;
    }
```

Let's use those two as a starting point and consider a new data path.

E100 Data path and control



```
while (1) {
        opcode = memory[PC];
        arg1 = memory[PC+1];
        arg2 = memory[PC+2];
        arg3 = memory[PC+3];

        if (opcode == 0) {
            PC = PC + 4;
            break;

        } else if (opcode == 1) {
            PC = PC + 4;
            memory[arg1] = memory[arg2] + memory[arg3];

        ...

        } else if (opcode == 13) {
            if (memory[arg2] == memory[arg3]) {
                PC = arg1;
            } else {
                PC = PC + 4;
            }
        }

        ...

    }
```

With a bit of work, we can rewrite this as:

```
while (1) {                          | keep executing next instruction
                                     | (until you execute a halt
                                     | instruction)

    opcode = memory[PC]; PC++        |
    arg1 = memory[PC]; PC++          | This is called "fetch"
    arg2 = memory[PC]; PC++          |
    arg3 = memory[PC]; PC++          |

    if (opcode == 0) {
        break;

    } else if (opcode == 1) {            | This is called "decode"

        memory[arg1] = memory[arg2] + memory[arg3];  | This is called
                                                     | "execute"

    ...

    } else if (opcode == 13) {
        if (memory[arg2] == memory[arg3]) {
            PC = arg1;
        }
    }

    ...

}
```

Can you create a table that implements our "fetch"?

state	next_state	pc_write	pc_drive	plus1_drive	opcode_write	arg1_write	arg2_write	arg3_write	address_write	mem_drive
reset	fetch1									
fetch1	fetch2									
fetch2	fetch3									
fetch3	fetch4									
fetch4	fetch5									
fetch5	fetch6									
fetch6	fetch7									
fetch7	fetch8									
fetch8	decode									

See http://www.eecs.umich.edu/courses/eng100/lab5/control.pdf for an answer...