Last time

- Introduced most of the e100 assembly language (all but call and ret).

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]
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]
sl	9	PC = PC+4 memory[arg1] = memory[arg2] << memory[arg3]
sr	10	PC = PC+4 memory[arg1] = memory[arg2] >> memory[arg3]
cpfa	11	PC = PC+4 memory[arg1] = memory[arg2 + memory[arg3]]
cpta	12	PC = PC+4 memory[arg2 + memory[arg3]] = memory[arg1]

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
		}

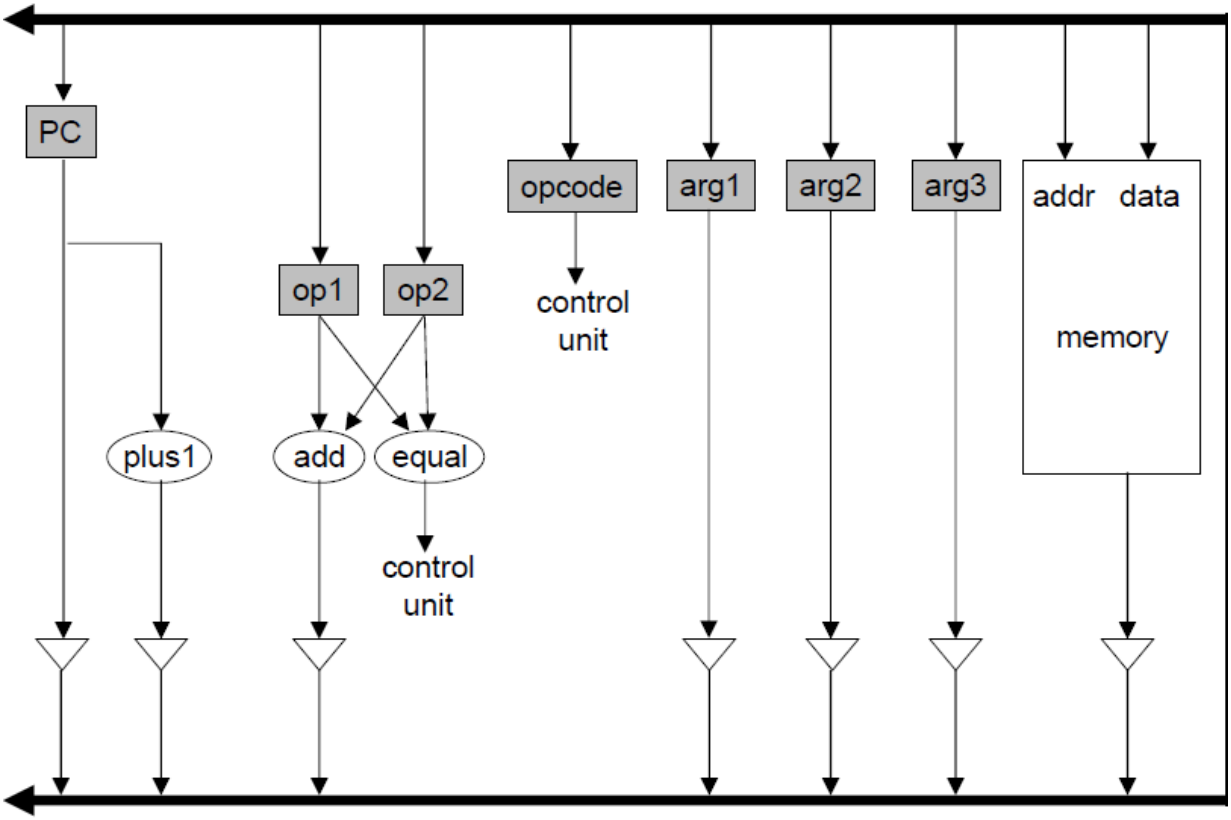
blt	15	if (memory[arg2] < memory[arg3]) {
		  PC = arg1
		} else {
		  PC = PC+4
		}
		
		Comparisons take into account the sign of the number. E.g., 32'hffffffff (-1) is less than 32'h00000000 (0).

call	16	memory[arg2] = PC+4 PC = arg1
ret	17	PC = memory[arg1]

- Wrote “absolute value” in machine code.

- Started in on how to *implement* the e100 processor. Looked at the data path and started on fetch.

E100 Data path and control



We didn't quite finish this last time, so let's try to get fetch done now.

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																

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

Decode and execute

So we've managed to "fetch" the instruction. We need to now figure out what instruction we've got and *for each instruction* do whatever that instruction needs to have done. That sounds like a lot. And it is. For now, let's just look at "add".

state	opcode_out	equal_out	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
decode			add1																

And finally, let's do "be"

state	opcode_out	equal_out	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
decode			be1																

Context time—levels of abstraction

We have done a lot in class so far.

- Learned basic combinational logic (and, or, not, etc.)
  - And not-so-basic combinational logic (tri-states)
- Learned basic sequential logic
  - Registers, memories
- Learned basic finite state machines with a data path (that's what we've been doing)
- Learned Verilog
- Learned how to build a device in hardware (max, rot13)
- Learned a "simple" instruction set for a computer (e100)
- *Started* to learn how to build a processor (e100 again)

One useful question a few of you have asked is "how does all this fit together". As I mentioned at the start of class, there is an idea of "levels of abstraction" in engineering—we work at one level and largely ignore the rest. What levels do we have and how do they interact? Let's start at the "top" and work down (more-or-less the opposite way we've learned things)

Assembly/machine language.

- We can write code in assembly to implement an algorithm. So far the only ones we've done are:
  - Find a^2+b^2
  - Find the absolute value.

Hardware design

- We can create hardware algorithms (max, rot13).
- We are working on building a computer—a generic hardware algorithm.
  - We will be able to run our machine language code on this hardware.

Combinational logic and sequential logic

- When we write Verilog code to implement the above hardware designs, we are specifying gates, registers and memories.
  - All of our combinational logic blocks (say equal16) are just a bit truth table. We specify the truth table in Verilog. And the tools create them and create them on the FPGA.
  - We more clearly request registers and memory (in your top.v files) by just instantiating a register or memory.

When we work at one level, we can largely focus on that level and may be worrying a bit about the level above and below. But we shouldn't need to be worrying about things more than one level away at all! So when writing assembly we really don't need to worry *how* it is implemented (how our hardware design was done) and we certainly don't need to worry about what gates/registers/memory were used.

Programming the E100

We have two instructions left to examine:

Instruction name	Opcode	Effect
call	16	`memory[arg2] = PC+4` `PC = arg1`
ret	17	`PC = memory[arg1]`

Call is used to call a function. It just jumps to the location of arg1 (wherever the function is) and it keeps track of what the next instruction would have been. That's so when the function it called finishes, it can return to the next instruction (just like a C++ program would…)

Return is how we return from the function. It just has arg1 be equal to whatever arg2 was for the call. So if the call wrote its return address in location 100, return should use that same location.

Mem[20] call 500 100 0
Mem[24] //next instruction

Mem[100] 0

Mem[500] //start of function

Mem[???] ret 100

Don't worry too much about the call and return for now. There is quite a bit involved in getting a function call to work in assembly (how do I pass arguments? How do I return values?) We'll tackle all that next time. I just wanted to finish off the instruction set before proceeding.

Example code

Let's write "Max". That is, let's write a program in assembly that finds the largest value in an array of 16 elements. For this program, let's assume

- The array starts at location 500 (so where is the last entry?)
- The "max" value should end up in location 499.

Go…