Meeting 5 SP23

Building the LC3 Microprocessor

@nebu

Outline

Plan

- I assume you've seen a processor before, and written at least 30 lines of assembly.
- Cover basic digital logic in Verilog.
- Build up the LC3 processor datapath and control with Verilog.
- Hopefully, get you started on building a simple processor of your own, simulating it, and extending it.
- ~~SIGARCH projects?~~

Section 1

Digital Logic Fundamentals

Combinational Logic

- AND
- OR
- NOT
- Together, these are logically complete:
- Any Boolean function can be represented in terms of AND, OR, and NOT.

In Verilog

```
// Continuous assignment
assign out = (a & b) | c;

// always_comb block
always_comb begin
    out = 1'b0;
    if (c == 1'b1) begin
        out = 1'b1;
    end
    if ((a == 1'b1) && (b == 1'b1)) begin
        out = 1'b1;
    end
end
```

Clocked Logic

- A clock is a square wave with (usually) a 50% duty cycle.
- D flip-flops pick up input data at the positive edge of the clock, and holds it as output data until the next positive edge.
- Usually has a reset signal to clear its value at startup.

In Verilog

```
always_ff @(posedge clk) begin
    if (rst == 1'b1) dff_out <= 1'b0;
    else dff_out <= dff_in;
end
```

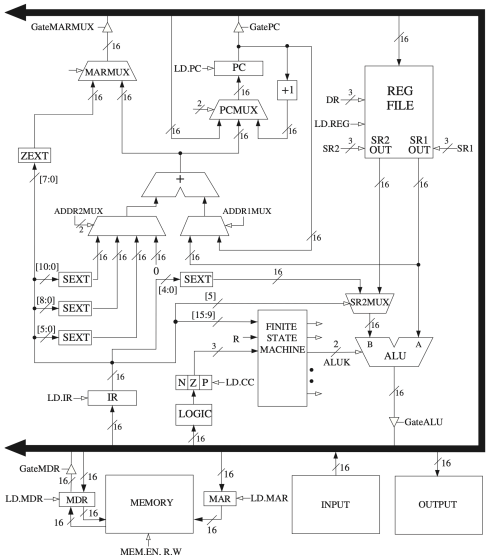
Datapath and Control

- Lots of digital systems can be logically thought of as:
- A datapath with combinational and clocked components with various signals that can be set to make it do certain things: to make data “flow” in a certain way.
- A “control” FSM that sets these signals sequentially to make the datapath do useful things.

Section 2

The LC3 Datapath

Datapath Image

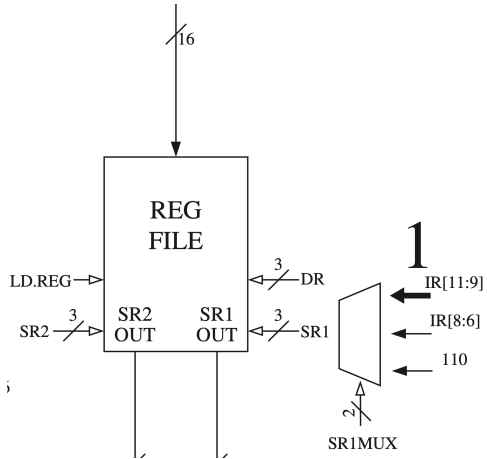


Looks Complex...

- Let's build it part-by-part.
- Start easy...

Register File

- Holds the processor's registers, that hold the values it'll operate on.



In Verilog: Regfile Loading

```
always @(posedge clk or negedge rst) begin
    if (rst == 1'b0) begin
        for (i = 0; i < 8; i = i + 1)
            registers[i] <= {16{1'b0}};
    end else if (ld_reg) begin
        priority case (drmux)
            2'b00: registers[ir[11:9]] <= data_bus;
            2'b01: registers[3'b111] <= data_bus;
            2'b10: registers[3'b110] <= data_bus;
        endcase
    end
end
```

Regfile Output

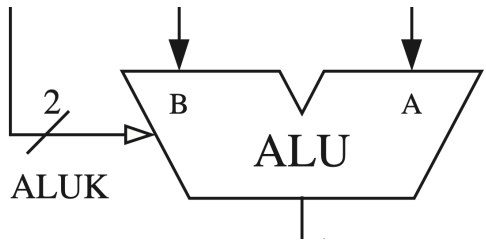
```
always_comb begin
    priority case (sr1mux)
        2'b00: sr1out = registers[ir[11:9]];
        2'b01: sr1out = registers[ir[8:6]];
        2'b10: sr1out = registers[3'b110];
    endcase
end
```

From Now On...

- Since you know how to build MUXes, I will skip some of the MUXes in the datapath, they're obvious to build.

Another Easy Part: ALU

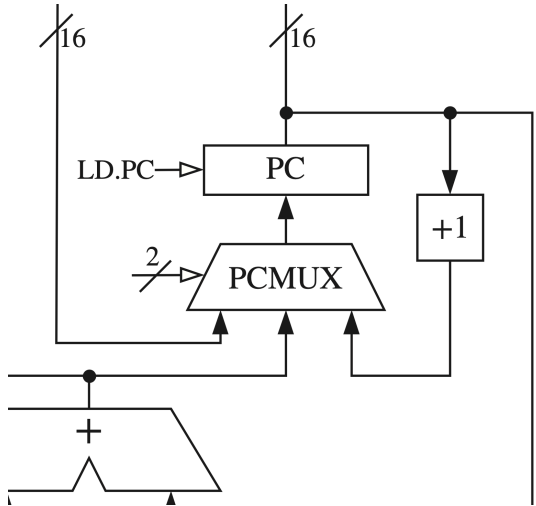
- Combinational, simply computes a Boolean function on the inputs.



ALU in Verilog

```
always_comb begin
    case (aluk)
        2'b00: alu_comp = op_a + op_b;
        2'b01: alu_comp = op_a & op_b;
        2'b10: alu_comp = ~op_a;
        2'b11: alu_comp = op_a;
    endcase
end
```

More Detailed Part: Program Counter



In Verilog

```
always @(posedge clk or negedge rst) begin
    if (rst == 1'b0) begin
        pc <= 16'h0200; // OS start.
    end else if (ld_pc) begin
        priority case (pcmux)
            2'b00: pc <= pc + 1'b1;
            2'b01: pc <= data_bus;
            2'b10: pc <= addr_out;
        endcase
    end
end
```

You do the Rest!

- The rest of the datapath is MUXes, adders, and registers, all of which you know how to do!
- Hooking these separate parts up requires some understanding of Verilog modules, which you can see in the provided code.

Section 3

LC3 Control

Setting Signals

- There were a bunch of inputs the datapath components took:
  - ALUK
  - MUX inputs
  - etc.
- These are set by the finite state machine that decodes instructions and figures out how to set the “control signals” in the datapath to execute it.

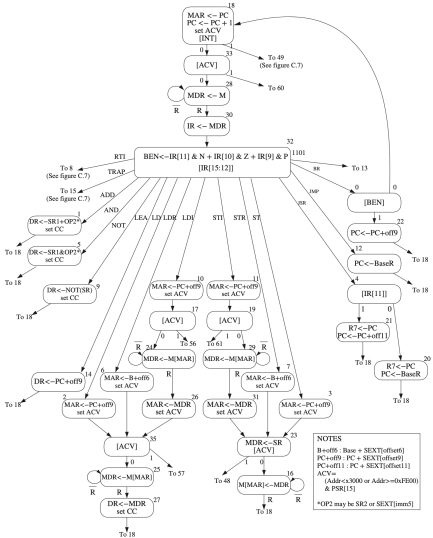
The FSM is Fundamentally

- A set of states.
- An initial state.
- A *state transition* function.
- An output function.

In Digital Logic

- A register for holding the current state.
- A combinational state transition function that outputs the next state based on the current state and the input.
- A combinational output function that computes the FSM output based on the current state.[1]

The LC3 FSM is Large...



The LC3 FSM is Large...

- So we “microcode it”.
- Store the state transition data and the outputs in a ROM.
- Index into it using the current state.
- The contents of the ROM tell us:
  - Outputs at the current state.
  - The next state to go to.

In Verilog, First Infer a ROM

```
// Infer a ROM for control store
initial begin
    $readmemb("microcode.bin", control_store);
end
```

Use the ROM for Control Signal Output:

```
// Index into it with the current state.
// Lower 25 bits are control signals.
assign control_bits = control_store[state][24:0];
```

Transition Function is a Bit More Complex...

```
always @(posedge clk or negedge rst) begin
    if (rst == 1'b0) begin
        state <= 6'b010010; // start at state 18
    end
```

Transition Function

```
else begin
    // Decode state
    if (control_store[state][25] == 1'b1) begin
        state <= {2'b00, ir[15:12]};
    end else begin
        case (control_store[state][28:26])
            // state <= the next state.
            // Logic omitted for brevity.
        endcase
    end
end
```

Where Did We Get the Contents of the ROM?

- Good exercise!
- Look at the datapath and the current state, and think about which signals to set to get the job done for that state.
- FSM and datapath will be in distributed ZIP file.

Section 4

Conclusion

What Next?

- Build your own small processor!
- Try: LC3, RISC-V, MIPS.
- Do not try: x86.
- For reference, the provided Verilog implements LC3.
- The run script requires that you have `iverilog` installed, an open source Verilog simulator.