Backend Control Logic Design the MBT way

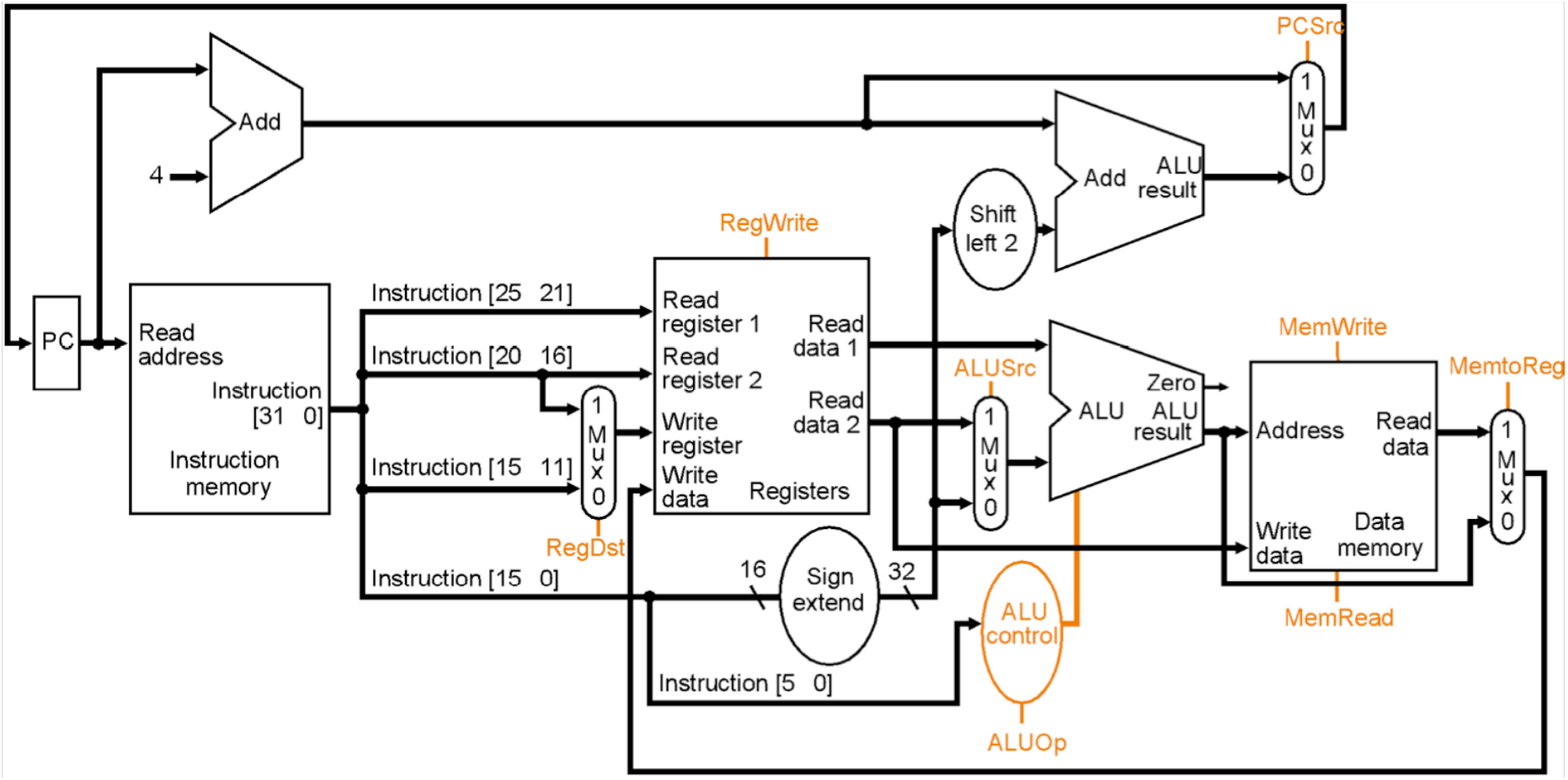
- Shown here for a simple MIPS-like design.
- Your design has a few differences; for instance it has stalls/restarts from the front end, has memory stalls, different ISA, etc, etc.
- Further pipelining of your backend (beyond the 2-way front-end/back-end split) is possible – but get the basic non-pipelined backend working first.

Recap: The Processor Design Algorithm

- Once you have an ISA…
- Design/Draw the datapath
  - Identify and instantiate the hardware for your architectural state
  - Foreach instruction
    - Simulate the instruction
    - Add and connect the datapath elements it requires
    - Is it workable? If not, fix it.
- Design the control (single-cycle machine)
  - Foreach instruction
    - Simulate the instruction
    - What control lines do you need?
    - How will you compute their value?
    - Modify control accordingly
    - Is it workable? If not, fix it.

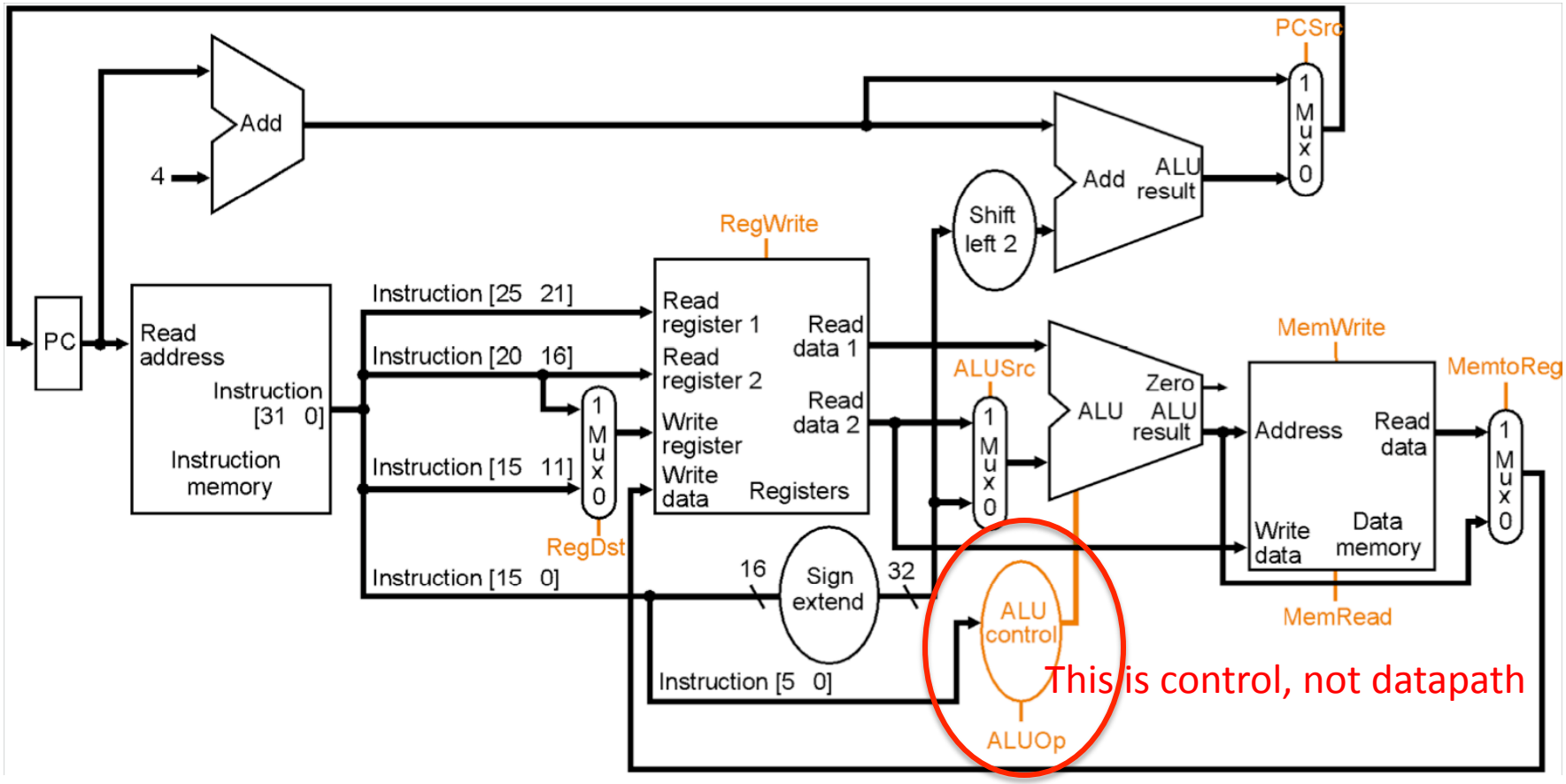
Derive datapath

(You would create this on a blackboard or whiteboard and powerpoint it up.)



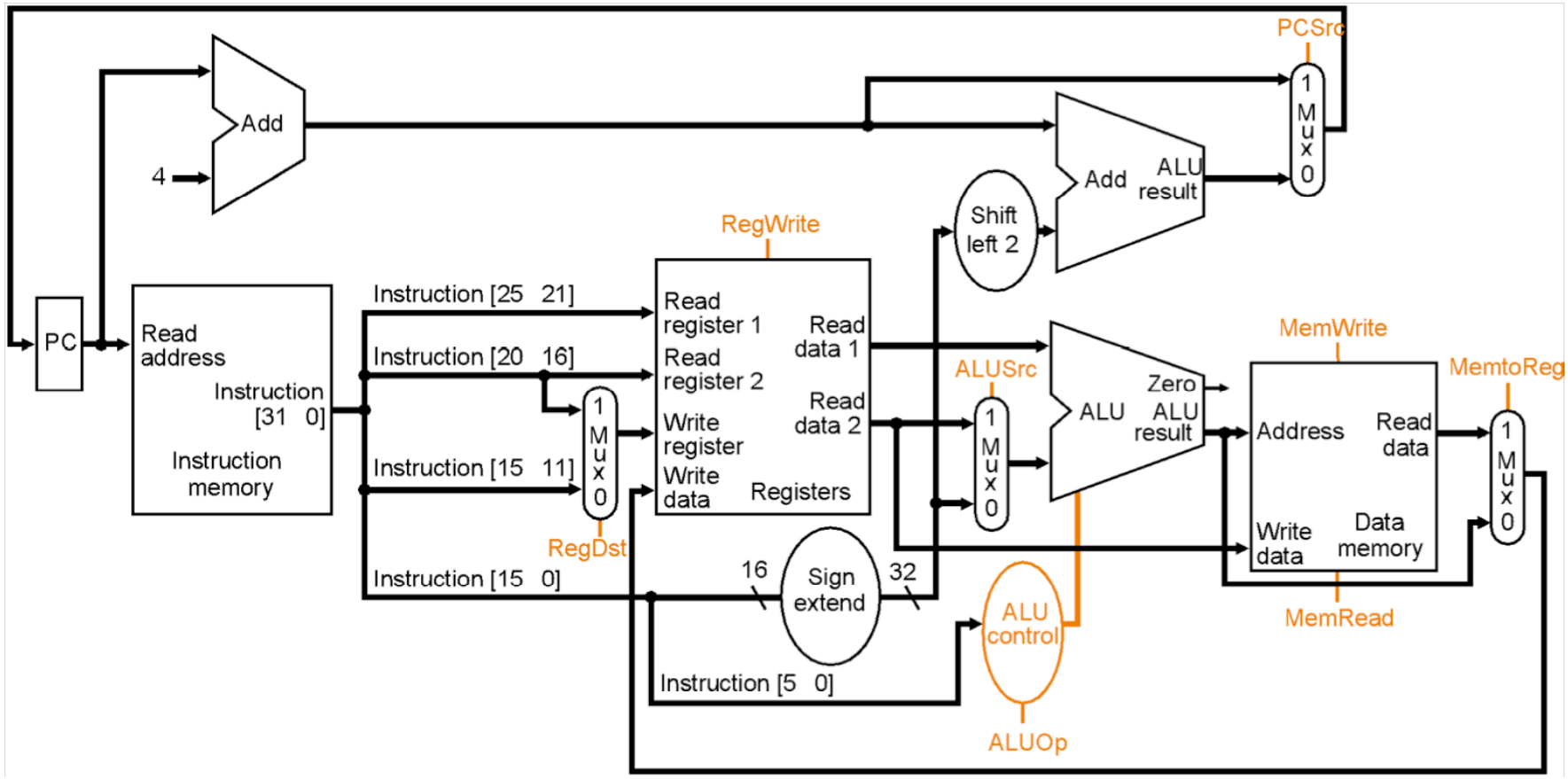
Define and Name Control Signals

Signal	== 0	== 1
RegDst	Write to rd	Write to rt
RegWrite	Register writes suppressed	Register writes occur
ALUSrc	2nd ALU input is R[rd]	2nd ALU input is the immediate
ALUop	Multiple bits; value determines the operation the ALU will perform.	



Define and Name Control Signals

Signal	== 0	== 1
PCSrc	PC <= PC + 4	PC <= PC + 4 + immediate
MemRead	Do not read data memory	Perform read at address
MemWrite	Do not write data memory	Perform write at address
MemtoReg	Present ALU result to register file for write	Present ALU result to register file for write.



Writing Control Logic

- hard part is to not make careless mistakes
- important to structure the code to avoid mistakes

```
`define LW    32'b100011_?????_?????_?????_?????_??????
`define SW    32'b101011_?????_?????_?????_?????_??????
`define ADDIU 32'b001001_?????_?????_?????_?????_??????
`define BNE   32'b000101_?????_?????_?????_?????_??????
```

```
reg writes_rf_c;  // aka RegWrite;

always_comb
  unique casez (IR)
      `LW:    writes_rf_c = 1'b1;
      `SW:    writes_rf_c = 1'b0;
      `BNE:   writes_rf_c = 1'b0;
      `ADDIU: writes_rf_c = 1'b1;
      …
      default: writes_rf_c = 1'b0;
  endcase
```

```
reg writes_dmem_c; // aka MemWrite;

always_comb
  unique casez (IR)
      `SW:    writes_dmem_c = 1'b1;
       …
      default: writes_dmem_c = 1b'0;
  endcase
```

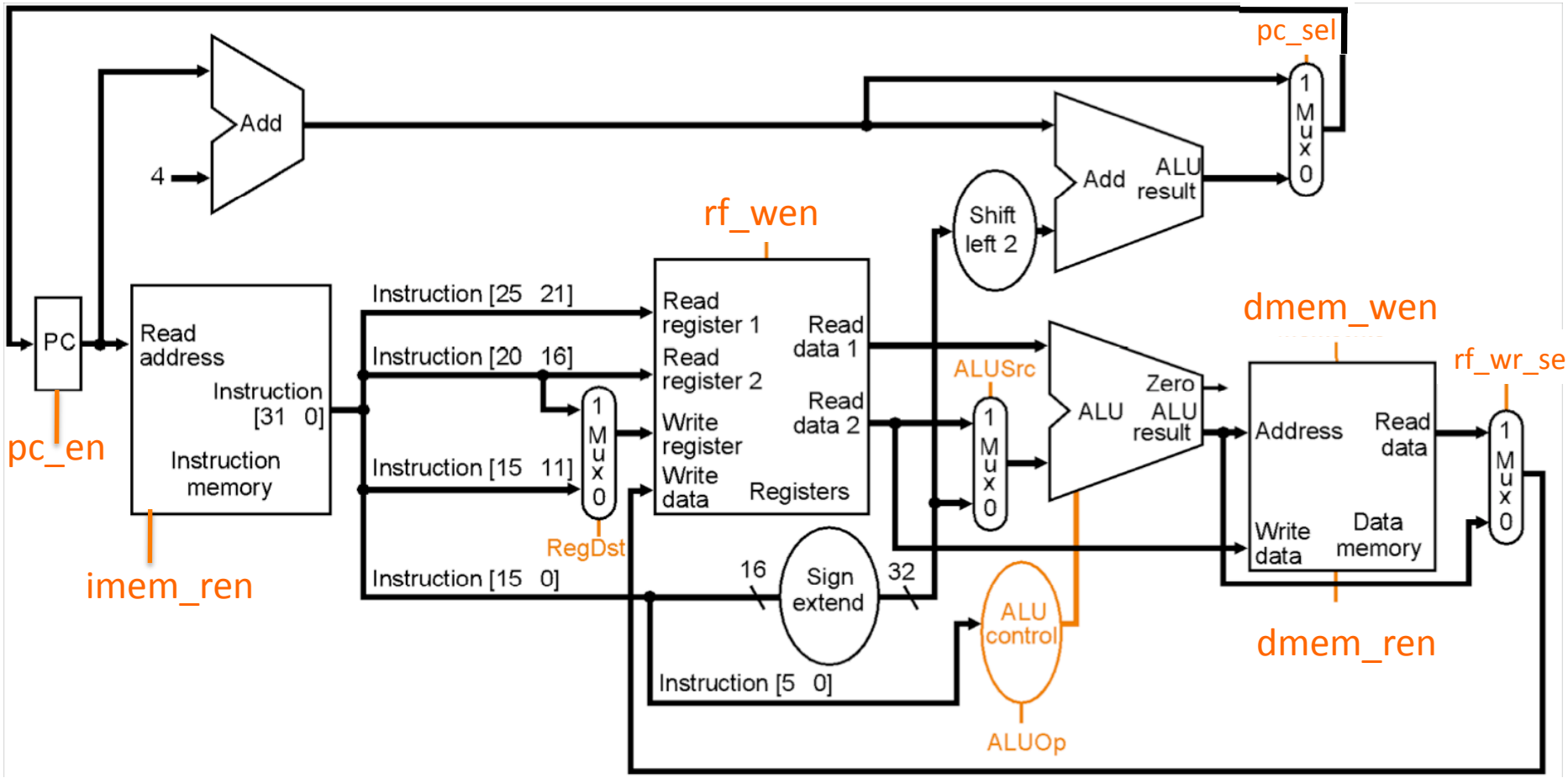
Single-Cycle Pipeline Is Often Not Ideal

- Your data memory, and possibly other structures can only be read on the clock edge: thus, you need at least two cycles, one for fetch, and another for doing loads.
- Instructions may require multiple states – perhaps they read the RAM multiple times, or require too many RF ports
- Your design needs to respect the refuse signal, which may affect things.
- Long cycle time. Timing Analyzer will identify excessively long paths. Increase clock rate by breaking paths up with registers and adding states. To improve performance, have instructions skip states they don't need.

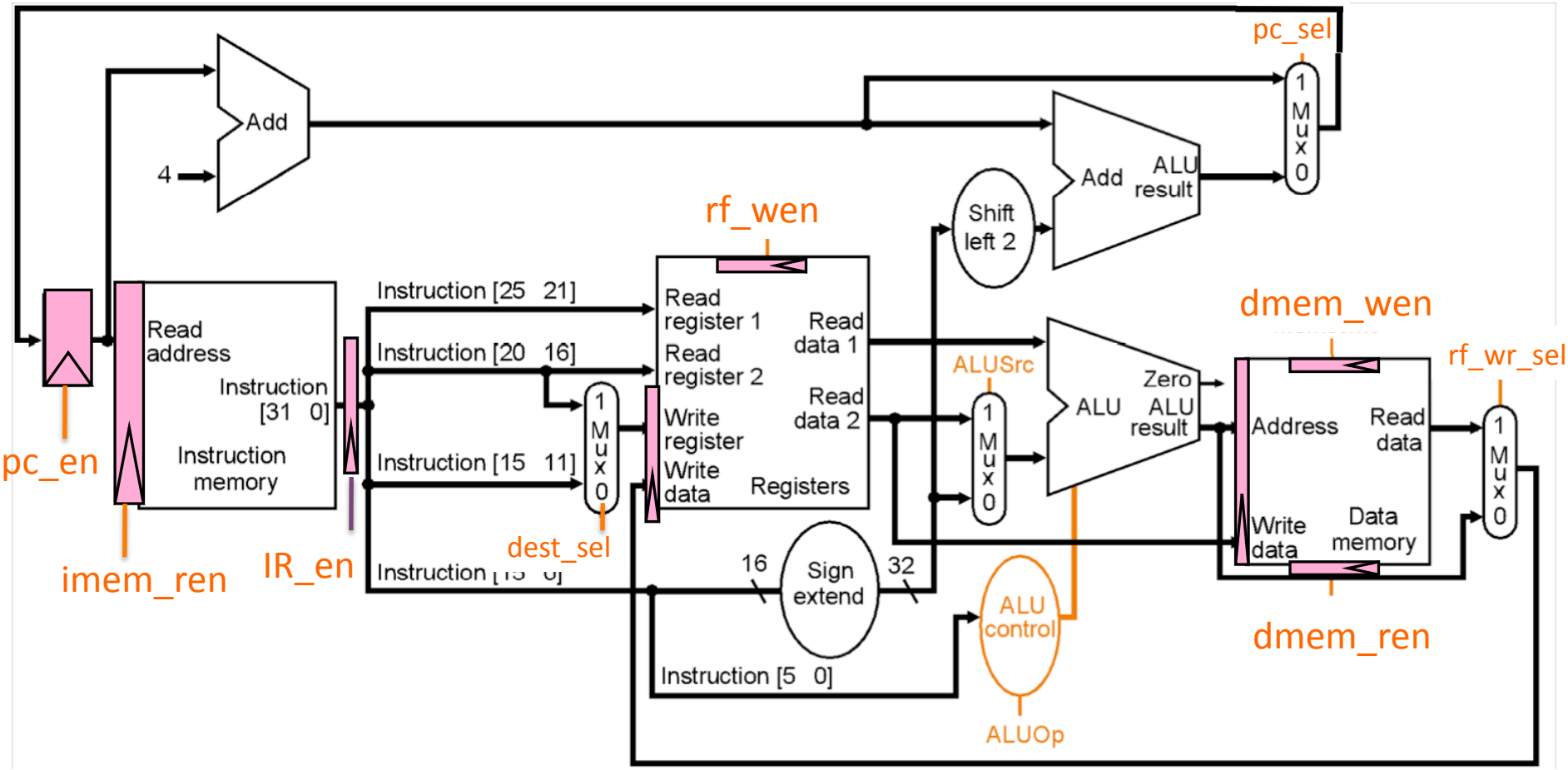
Solution: Convert to Multicycle Data Path

- Update your datapath with new registers, as needed.
- Draw Finite State Machine
- Mentally simulate on your datapath to ensure correctness. Do you wait enough cycles in your state machine such that all inputs have arrived at each datapath element?
- Code Enum to define states
- Use `always_ff` to create state registers.
- Code next state logic.
- Code Logic that determines control signals at each state
  - many signals, like muxes, stay steady through all states in the instruction
    - unless you reuse a resource more than once in an instruction's execution
  - write enable signals usually are state dependent – if you write too earlier, you will corrupt state that current instruction needs – usually wait until last cycle
  - read enables are often state dependent to save power.

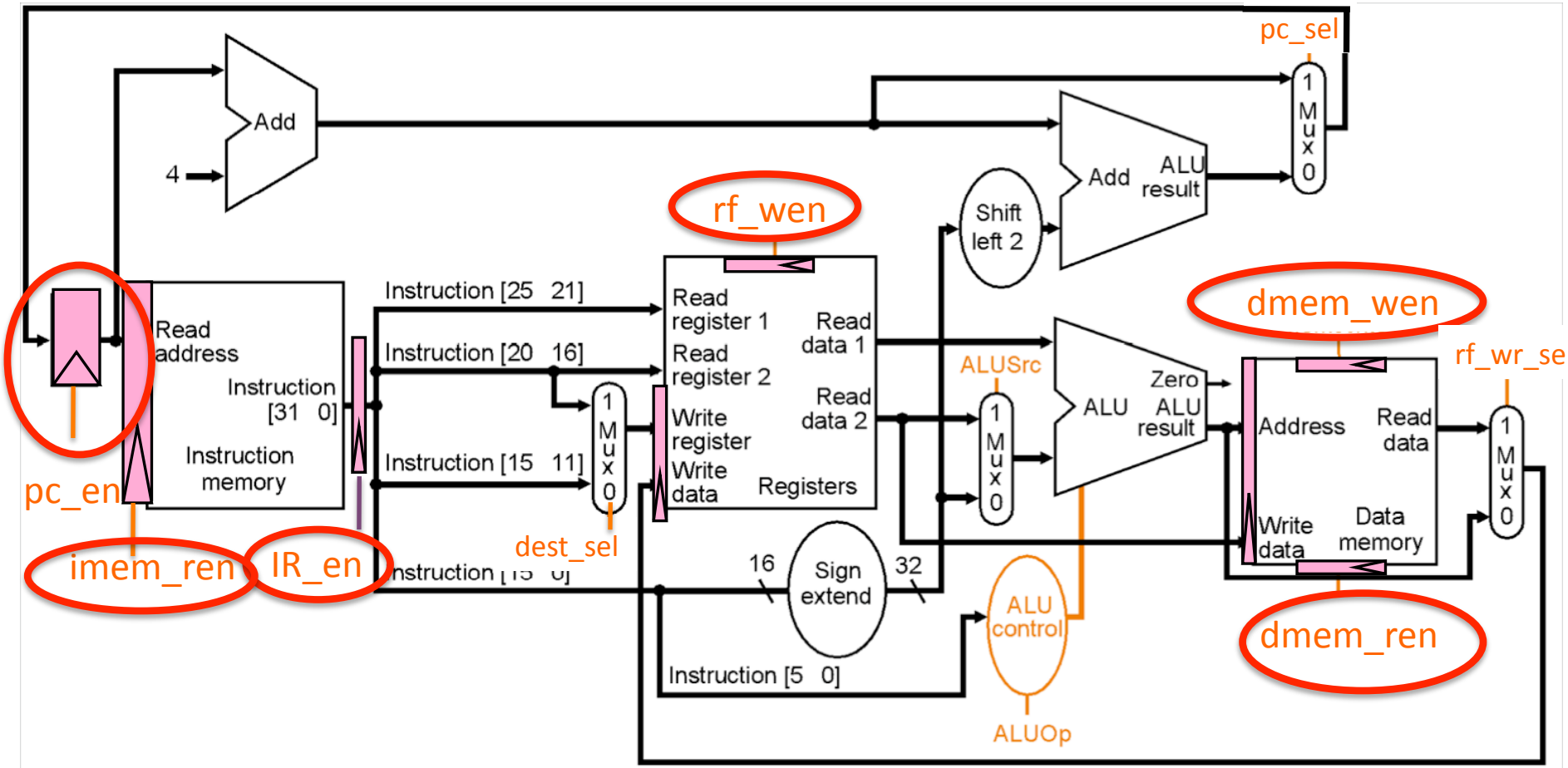
Insert Necessary or Existant Registers



Insert Necessary or Existant Registers

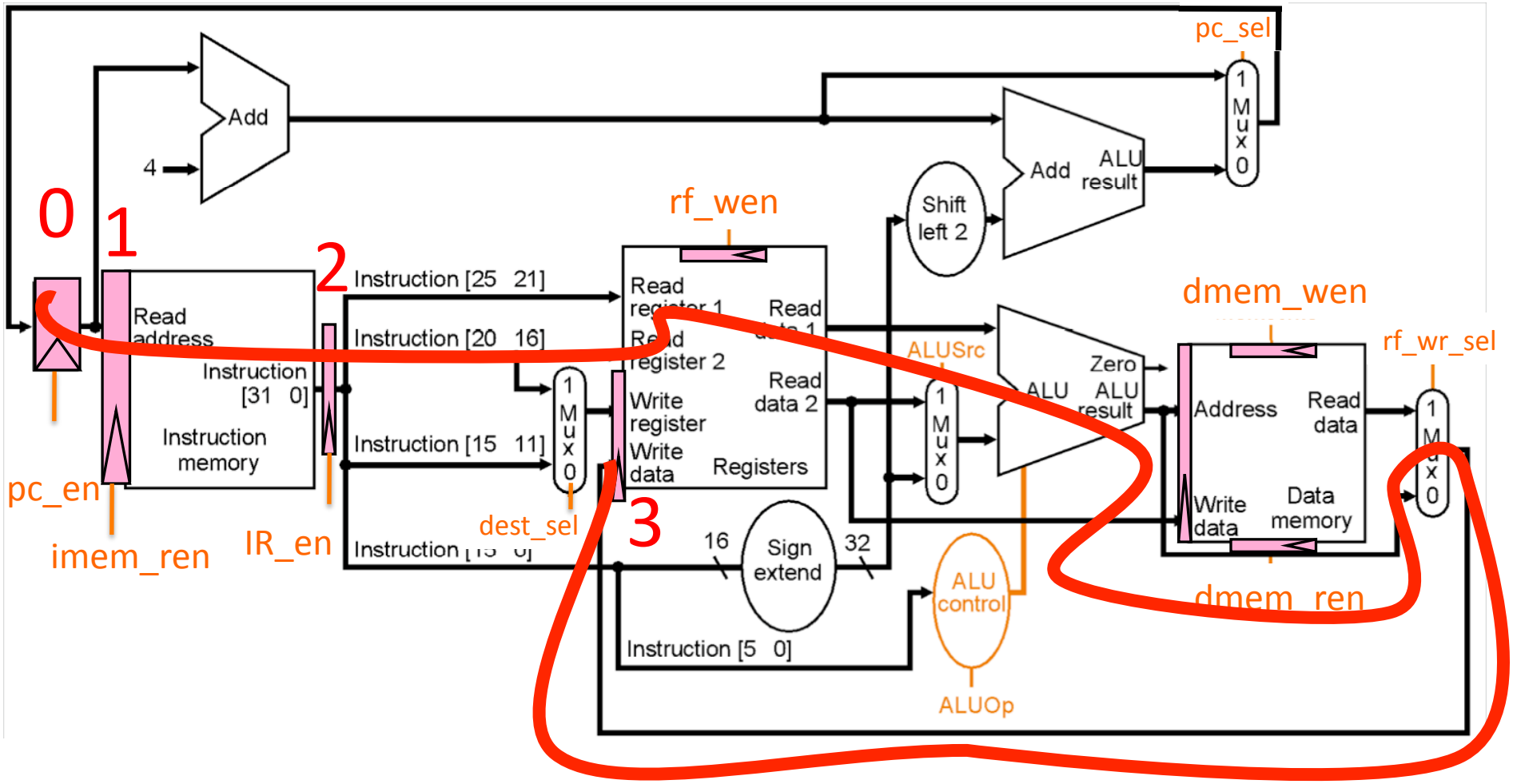


Identify Write/Read Enables



For each instruction's datapath components, identify longest latency (in cycles) inputs. This is the cycle it "fires" on.

NewPC Fetch ⟵ Execute ⟶ Memory

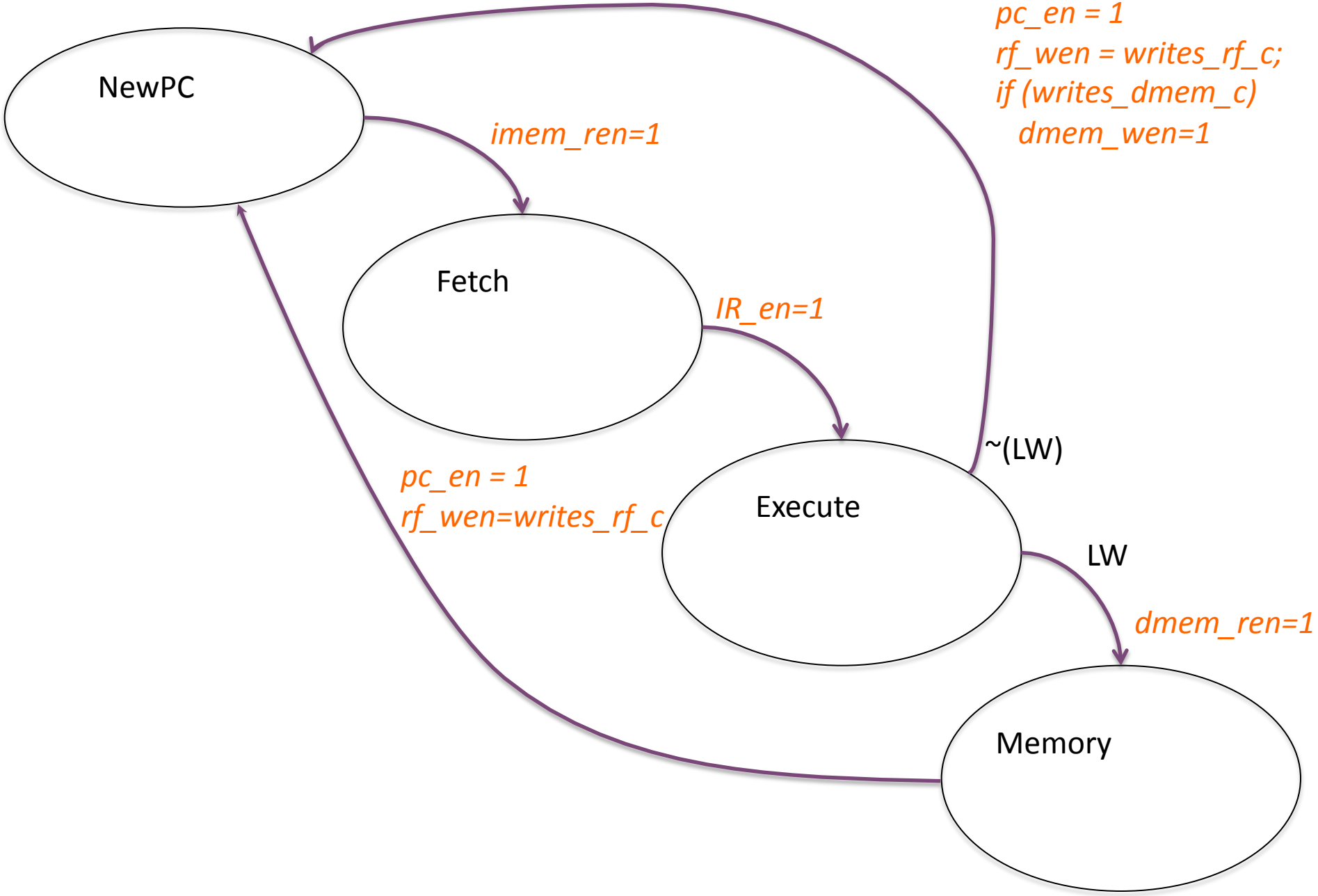


Then, code up datapath in structural verilog

...

and move onto control.

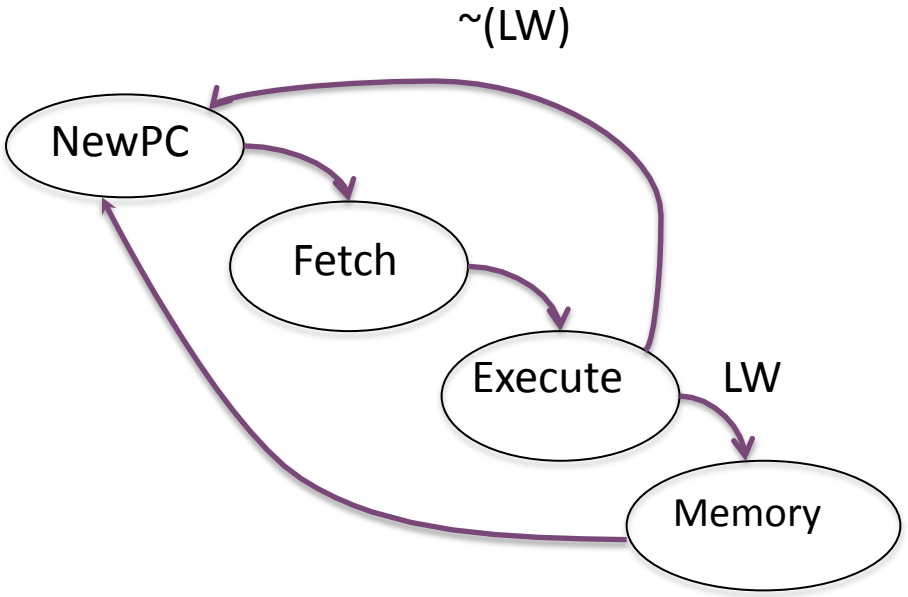
Draw / Debug State machine



Coding Control: Next State Logic

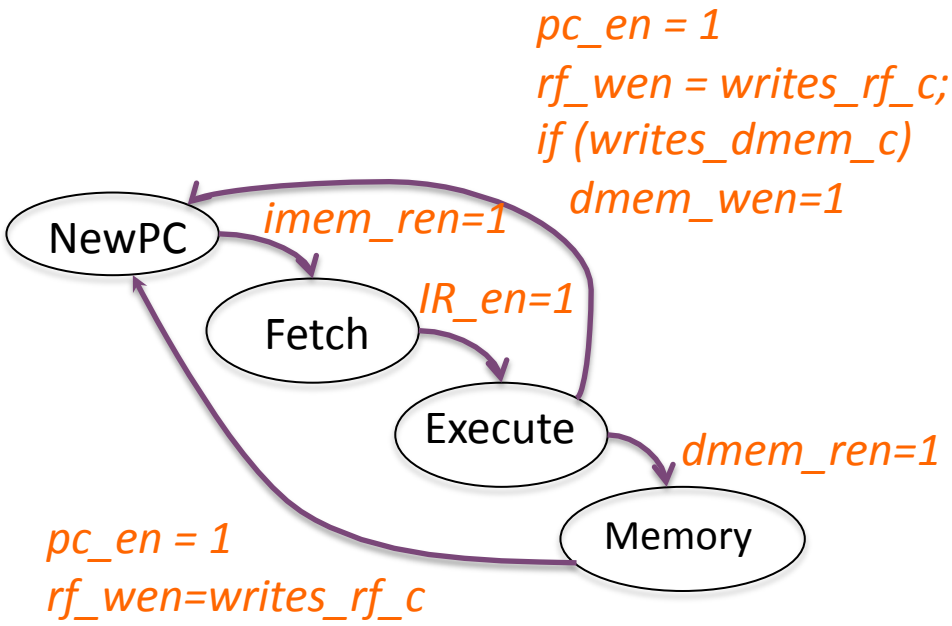
(with a little Synthesizable SystemVerilog; remember macros for LW etc from earlier)

```
typedef enum { sNewPC, sFetch, sExecute, sMemory } state_s;
state_s substate_r, substate_n;

always_ff @(posedge clk)
   substate_r <= reset ? sNewPC : substate_n;

always_comb
unique case (substate_r)
      sNewPC:     substate_n = sFetch;
      sFetch:     substate_n = sExecute;
      sExecute:
                  unique casez (IR_r)
                        `LW: substate_n = sMemory;
                     default:   substate_n = sNewPC;
                  endcase
     sMemory: substate_n = sNewPC;
        default: substate_n = sNewPC;
 endcase
```



Coding Control: Enables

```
always_comb
begin
    pc_en        = 1'b0;
    imem_en      = 1'b0;
    IR_en        = 1'b0;
    dmem_wen     = 1'b0;
    dmem_ren     = 1'b0;
    rf_wen       = 1'b0;

  unique case (substate_n)
     sNewPC:
      begin
        pc_en   = 1;
        rf_wen  = writes_rf_c;
        dmem_wen = writes_dmem_c;
      end
     sFetch:   imem_ren = 1;
     sExecute: IR_en = 1;
     sMemory:  dmem_ren = 1;
  endcase

end
```



Coding Control: Combinationals

- combinational decode logic often does not require state-sensitivity

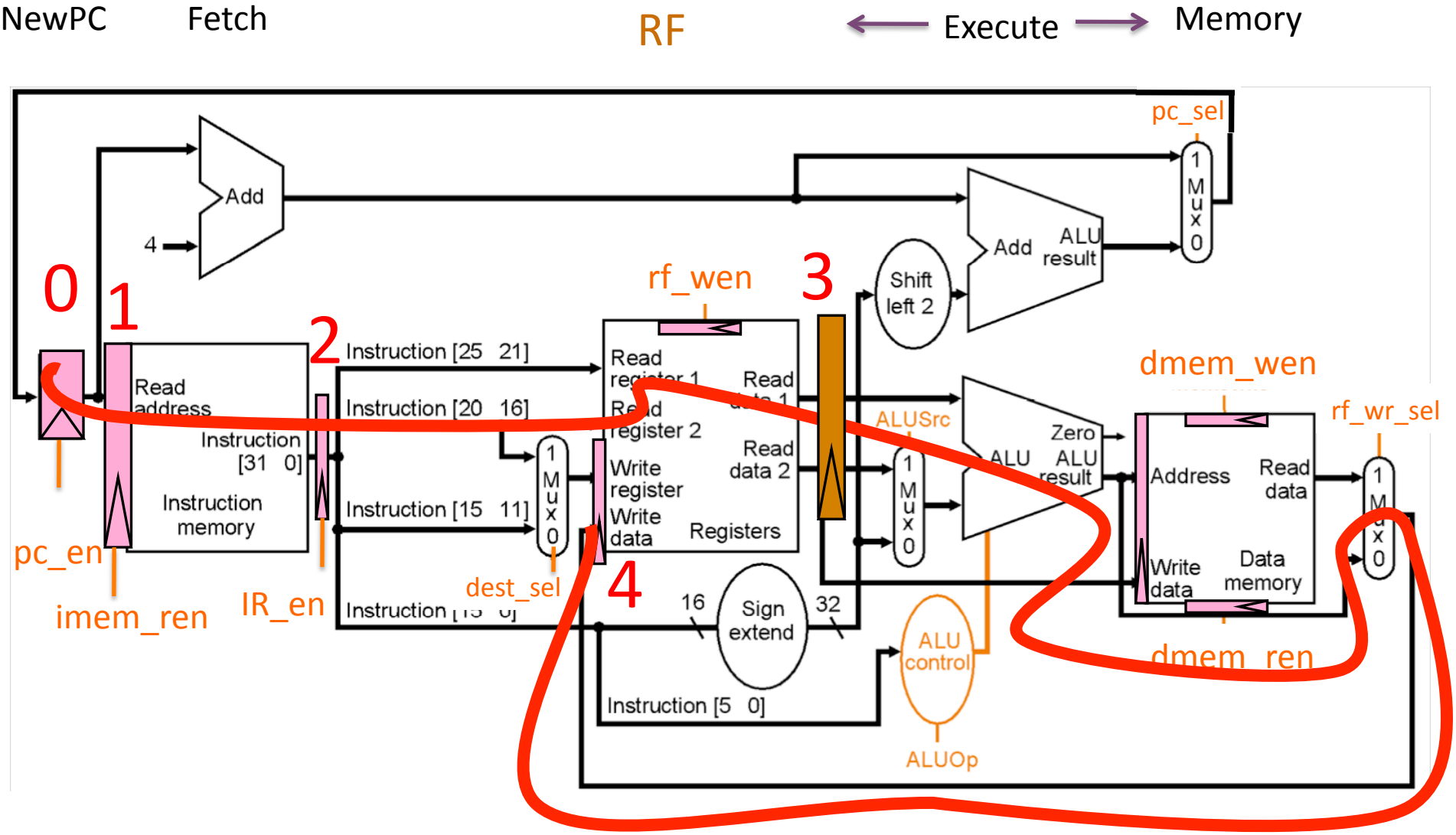
```
reg rf_wr_sel; // aka WriteToReg;

always_comb
  unique casez (IR_r)
      'LW:     rf_wr_sel = 1'b1;
       …
     default: rf_wr_sel = 1b'0;
  endcase
```

```
reg writes_rf_c;  // aka RegWrite;

always_comb
  unique casez (IR_r)
      `LW:     writes_rf_c = 1'b1;
      'SW:     writes_rf_c = 1'b0;
      `BNE:    writes_rf_c = 1'b0;
      `ADDIU:  writes_rf_c = 1'b1;
      …
     default: writes_rf_c = 1'b0;
  endcase
```

remaining:
- dest_sel
- ALUSrc
- ALUOp
- pc_sel
- etc…

```
reg writes_dmem_c; // aka MemWrite;

always_comb
  unique casez (IR_r)
      'SW:     writes_dmem_c = 1'b1;
       …
     default: writes_dmem_c = 1b'0;
  endcase
```

Fixing Critical Paths:
Add registers on slow paths and fix state machine logic accordingly.



Additional Complexities