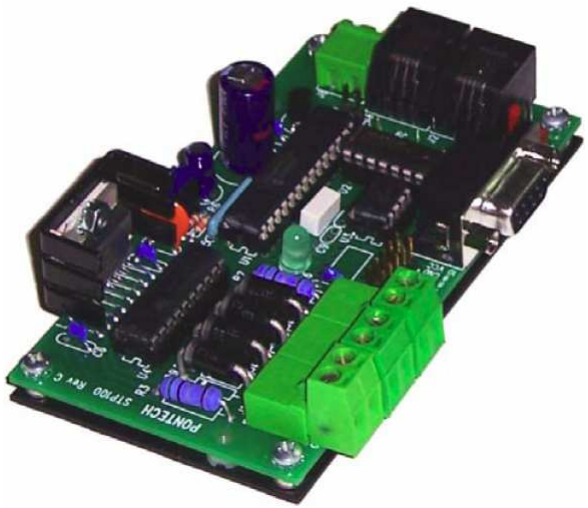
Stepper Motor ASIP

- We will build a simple ASIP that has instruction set specifically aimed at controlling stepper motors
- Instruction set
  - Move motor by a number of (half-)steps
  - Pause between moves
  - Add/subtract, branch, set registers, move data between registers
    - To support loops
  - Move motor to an absolute position (bonus)

Real-World Example

- Such controllers are actually sold as real products



- STP100 motor controller from Pontech
- Contains motor drivers (H bridges)
  - We will build ours on the protoboard

ASIP Requirements

- We want to build a microprocessor whose instruction set directly supports stepper motor control
  - Move motor clockwise/counter-clockwise by a number of steps/half-steps (all combinations)
    - Automatically pause between moves to accommodate the motor's inertia
  - Keep track of motor position
  - Move motor to a desired position (optional)
  - Explicit "pause" instruction
  - Other common instructions to support basic loops

ASIP Architecture

- Very simple
  - 8-bit instructions and data
  - Instruction ROM
  - Register file with 4 registers
    - 2 general purpose (R0 and R1)
    - 2 special purpose
      - R2 – stores motor's position (**in half steps**)
      - R3 – determines delay between steps/half-steps
  - Stepper ROM
    - Stores the basic sequence of signals used to drive the motor
  - ALU
    - Performs basic computation
  - Delay logic
    - Implements delay between steps
  - PC, decoder, branch logic, multiplexers

ASIP Instruction Set

- `MOVR reg`
  - Move the motor by a number of full steps specified in the register
  - The number in the register is **signed**, so the motor can move in either direction
  - After each step, wait for the amount of time specified in the `delay` register before proceeding
- `MOVRHS reg`
  - Move the motor by a number of half-steps specified in a register
  - All other requirements are the same as for `MOVR`

ASIP Instruction Set…

- `PAUSE`
  - Wait for the amount of time specified inside the delay register
- `CLR reg`
  - Clear (i.e. set to 0) the specified register
- `ADDI reg, imm3`
  - Add the 3-bit **unsigned** immediate operand to the specified register; store the result into the register
- `SUBI reg, imm3`
  - Subtract the 3-bit **unsigned** immediate operand from the specified register; store the result into the register

ASIP Instruction Set…

- `SR0 imm4`
  - Set (fill) 4 lower bits of register 0 with the 4-bit immediate operand
- `SRH0 imm4`
  - Set (fill) 4 higher bits of register 0 with the 4-bit immediate operand
- `MOV regd, regs`
  - Move the content of the `regs` (source) register into `regd` (destination) register

ASIP Instruction Set…

- `BR imm5`
  - Branch (unconditionally) to `PC+imm5`, where PC is current program counter and `imm5` is **signed** 5-bit immediate
- `BRZ imm5`
  - If register R0 is zero, branch to `PC+imm5`, where PC is current program counter and `imm5` is **signed** 5-bit immediate

ASIP Instruction Set…

- `MOVA reg`
  - Move the motor to an absolute position specified in the register
  - If the number in the register is different from the current position (R2) move the motor in an appropriate direction until they are the same
    - May involve a combination of steps and half-steps
  - After each step (or half-step), wait for the amount of time specified in the `delay` register before proceeding
  - BONUS
    - If you implement this instruction, you get 1% bonus
    - We will not include it in the class discussion

Instruction Encoding

- Instructions are encoded as binary numbers
- I = immediate
- R = register
- R_S = source register
- R_D = destination register

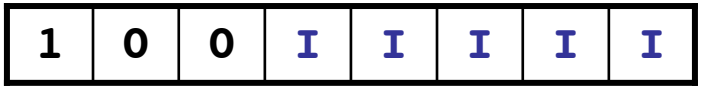
1	0	0	I	I	I	I	I	BR imm5
1	0	1	I	I	I	I	I	BRZ imm5
0	0	0	I	I	I	R	R	ADDI reg, imm3
0	0	1	I	I	I	R	R	SUBI reg, imm3
0	1	0	0	I	I	I	I	SR0 imm4
0	1	0	1	I	I	I	I	SRH0 imm4
0	1	1	0	0	0	R	R	CLR reg
0	1	1	1	R_D	R_D	R_S	R_S	MOV regd, regs
1	1	0	0	0	0	R	R	MOVA reg **(BONUS)**
1	1	0	0	0	1	R	R	MOVR reg
1	1	0	0	1	0	R	R	MOVRHS reg
1	1	1	1	1	1	1	1	PAUSE

Instruction Encoding Example

- ADDI R1,5

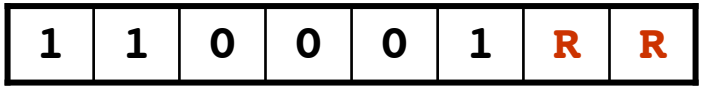
- R1 → 01
- 5 → 101
- ADDI R1,5 → 00010101 = 0x13

- BR -3



- -3 → 11101
- BR -3 → 10011101 = 0x9D

- MOVR R0

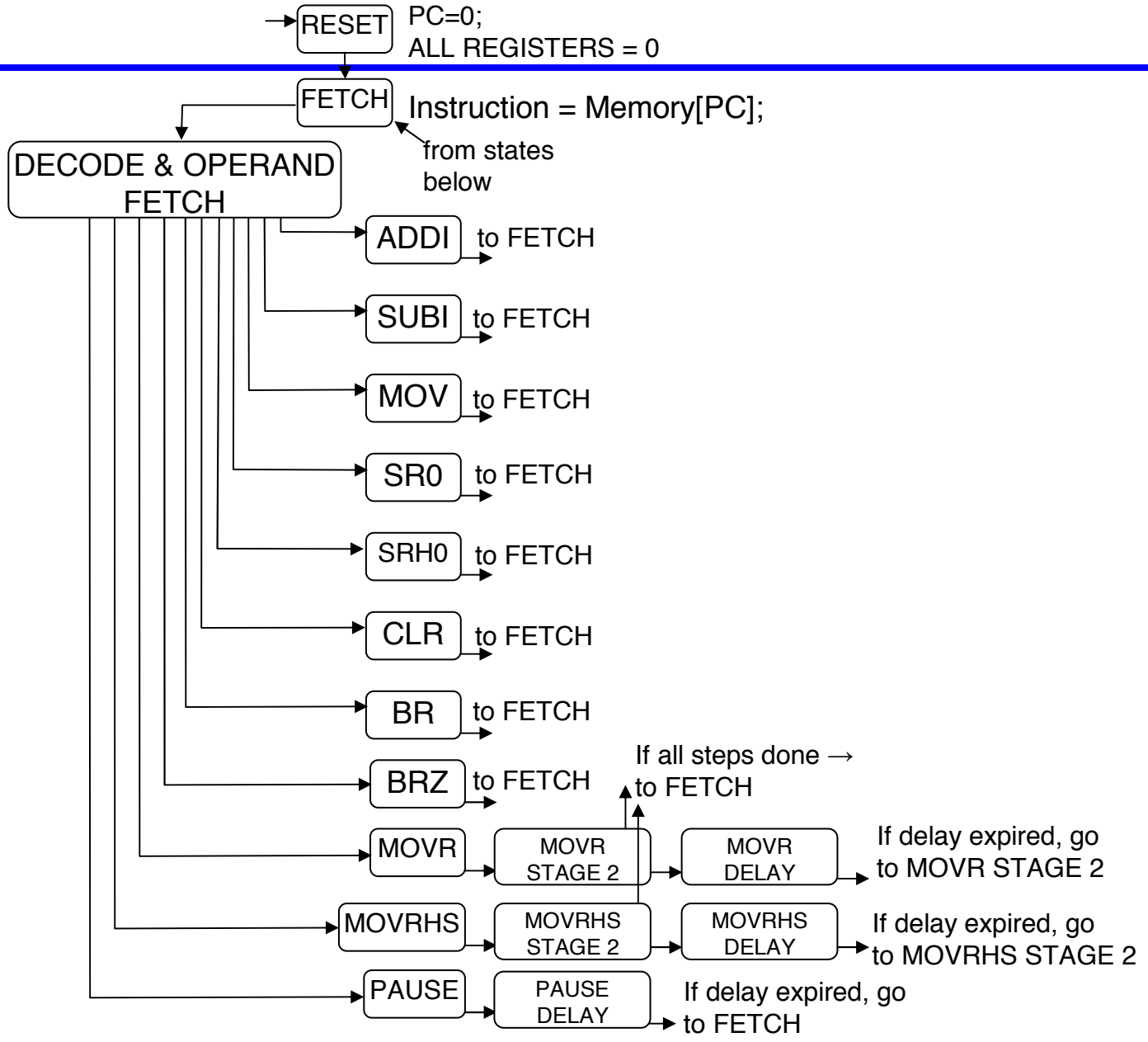


- R0 → 00
- MOVR R0 → 11000100 = 0xC4

Implementing the ASIP

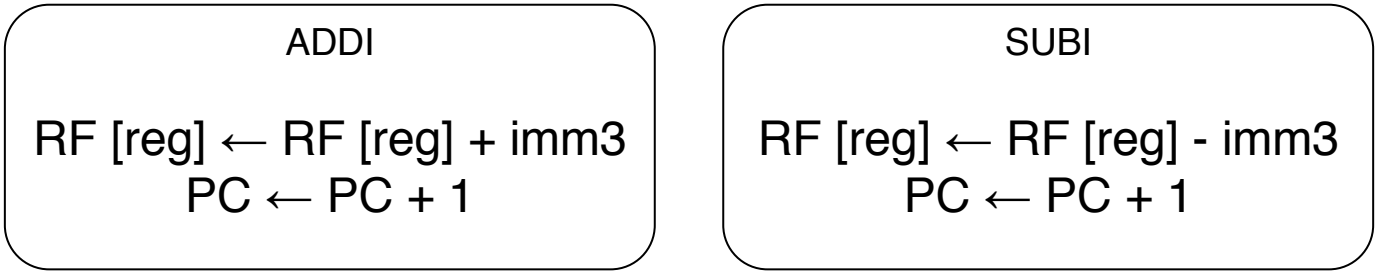
- Create FSMD
- Draw datapath
  - Write Verilog
- Create FSM for the controller
  - Write Verilog
- Put them together
  - Write Verilog
- Test and debug
- You will be given quite a bit of skeleton code, to save you time on unnecessary typing

Creating FSMD



FSMD Operations

- Need to specify the actions taken in each stage of the FSMD
  - Based on previous description of the instructions
- Examples
- `ADDI reg, imm3` `SUBI reg, imm3`



- `RF[reg]` means register `reg` in the register file (RF)

FSMD Operations…

- `MOV regd, regs`

MOV

RF [regd] ← RF [regs]
PC ← PC + 1

- `SRH0 imm4`

SRH0

RF [0] ← {imm4,RF $[0]_{3:0}$}
PC ← PC + 1

- `BR imm5`

BR

PC ← PC + sext(imm5)

`SR0 imm4`

SR0

RF [0] ← {RF $[0]_{7:4}$,imm4}
PC ← PC + 1

`CLR reg`

CLR

RF [reg] ← 0
PC ← PC + 1

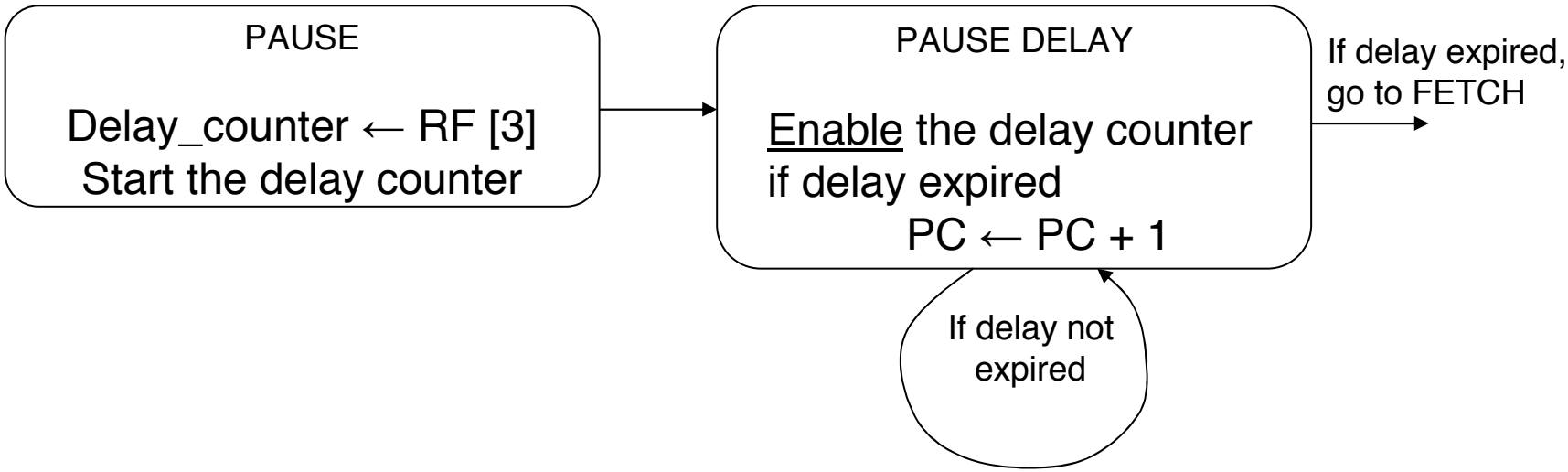
`BRZ imm5`

BRZ

if (RF[0] == 0)
    PC ← PC + sext(imm5)
else
    PC ← PC + 1

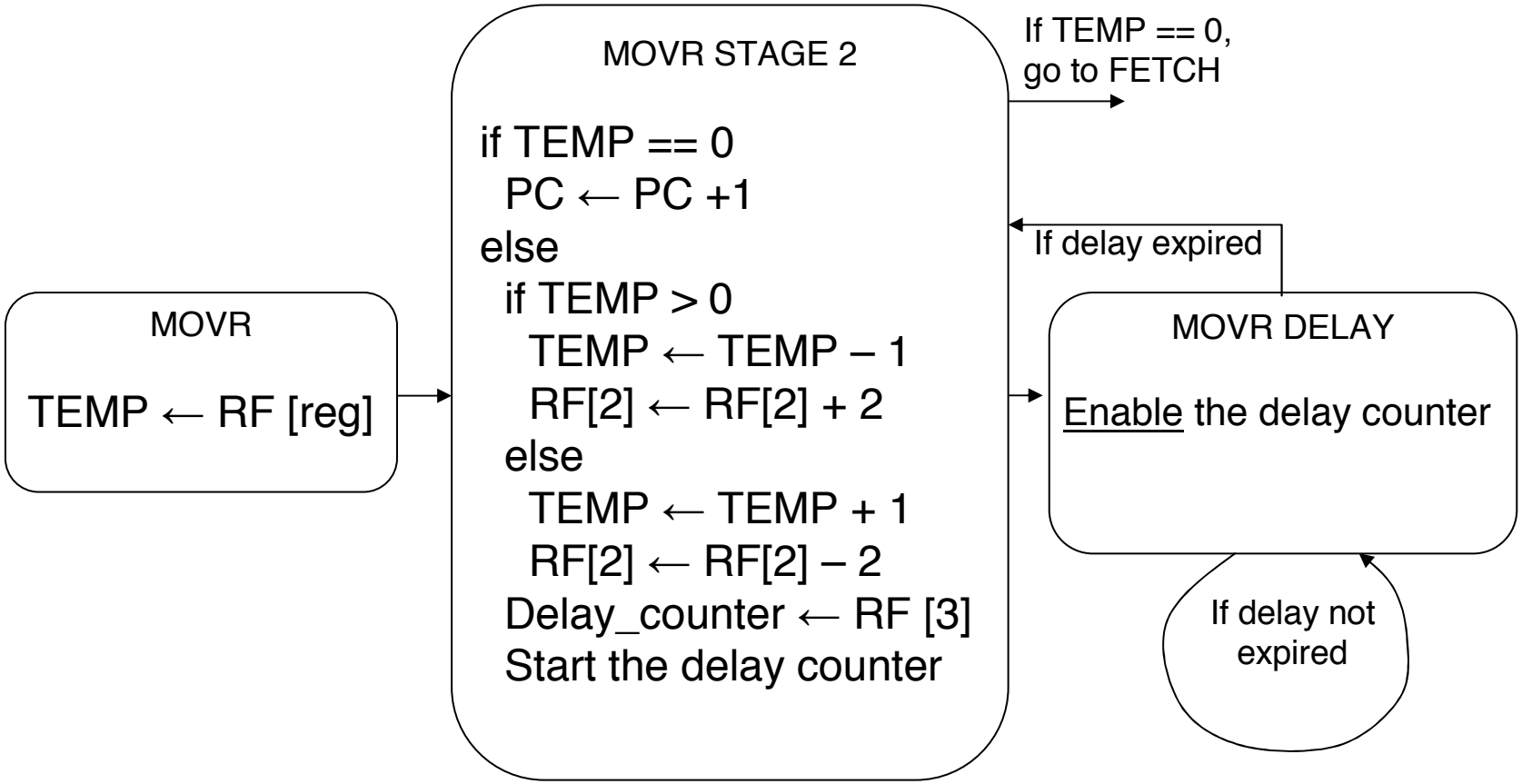
FSMD Operations…

- PAUSE



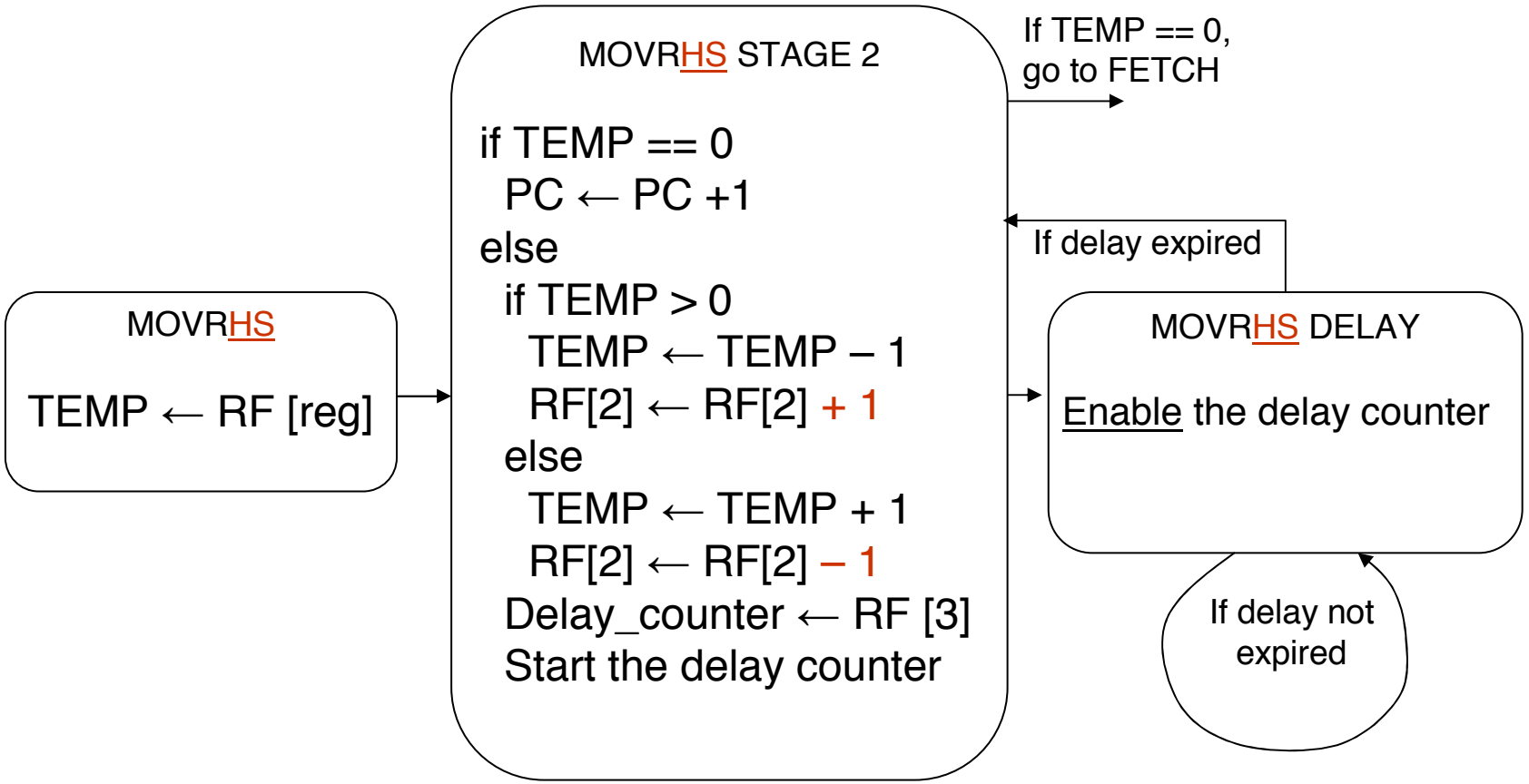
FSMD Operations…

- `MOVR reg`



FSMD Operations…

- `MOVRHS reg`

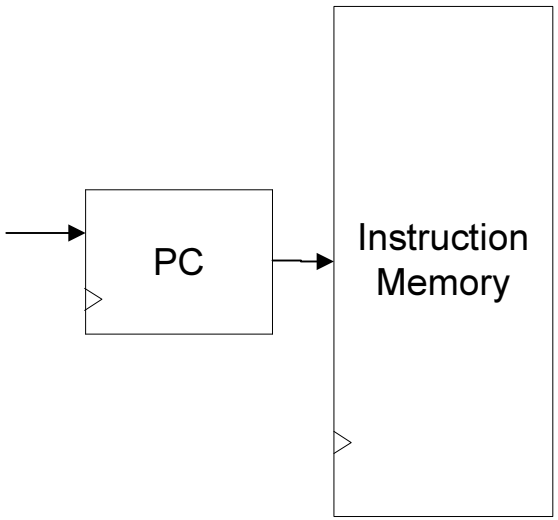


Datapath

- FSMD is complete
- We need to develop datapath that can implement this FSMD
- We could follow the procedure provided in Vahid
  - Many circuit components
  - We will partly follow this procedure, but will simplify things where possible

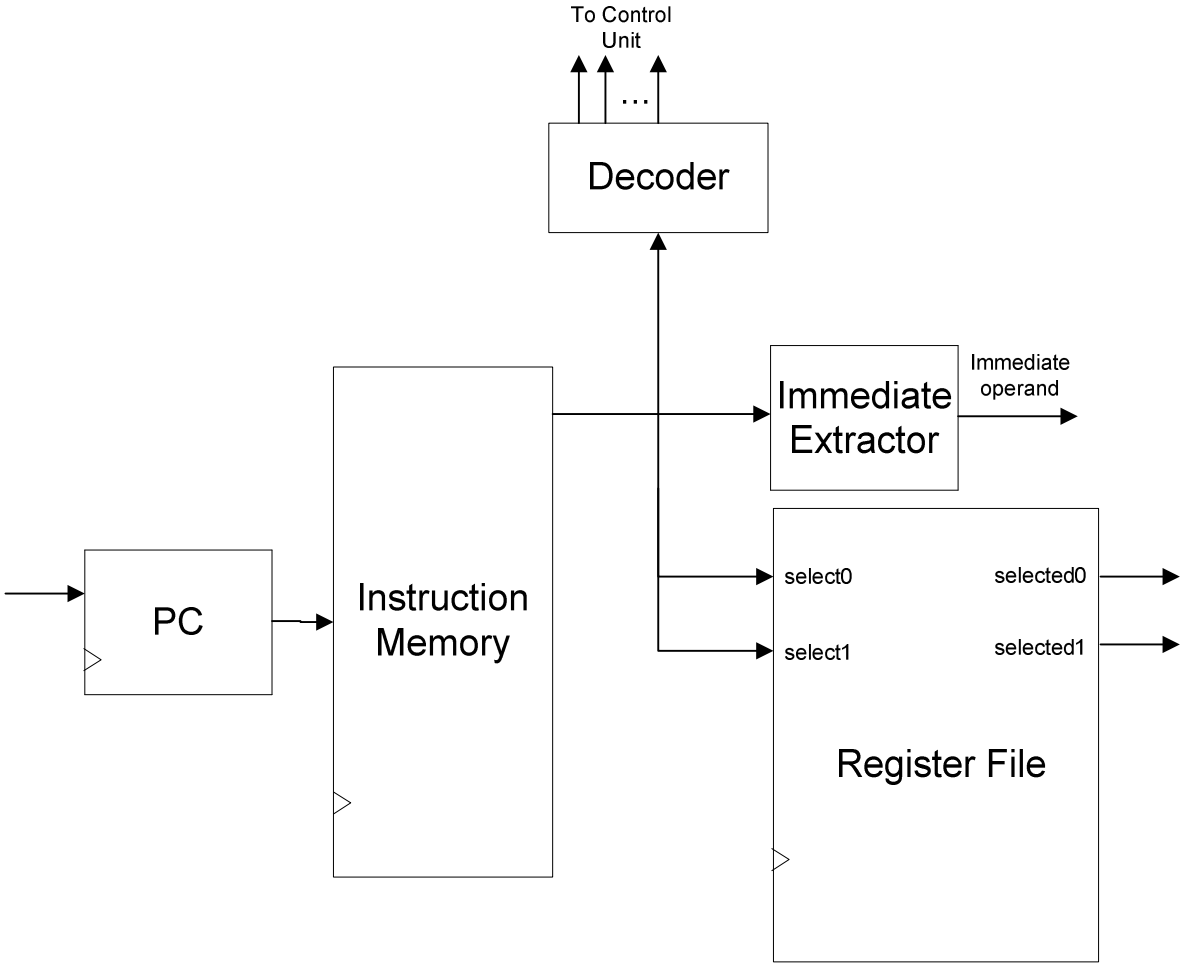
Datapath: PC

- Fetching instructions
  - Need a program counter and instruction memory
  - Program counter should be able to count by 1 (be able to increment itself) and be loaded with a new value (on branch)



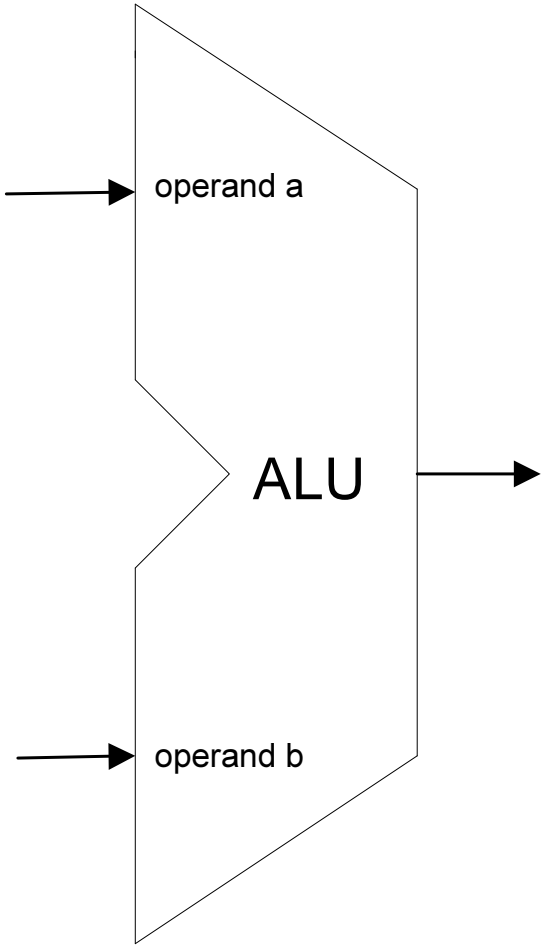
Datapath: Decode and Operand Fetch

- Once the instruction is fetched, it should be <u>decoded</u> and operands should be <u>fetched</u>



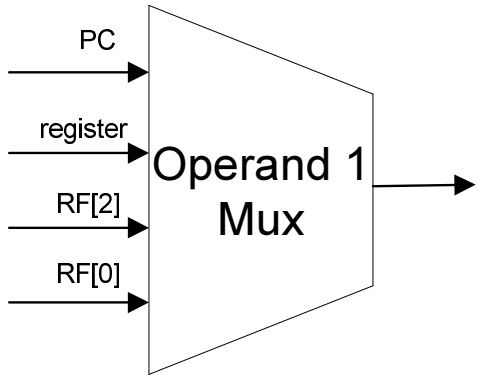
Datapath: Computation

- We will have one ALU for most of computing tasks
  - ALU stands for Arithmetic and Logic Unit
  - Our unit should be able to
    - Add (for ADDI, BR, BRZ, MOVR, MOVRHS)
    - Subtract (for SUBI, MOVR, MOVRHS)
    - Set lower 4 bits of a register (for SR0)
    - Set higher 4 bits of a register (for SRH0)



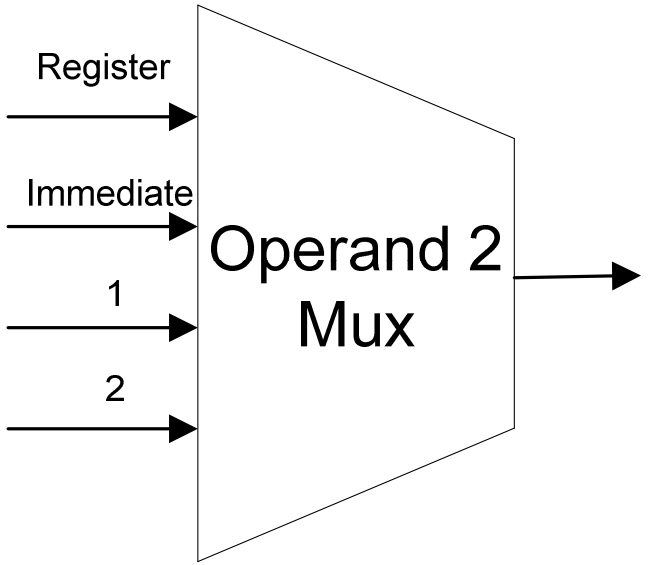
Datapath: Operand Selection

- ALU computes many different operations
  - Also on different operands
  - Need multiplexers to choose one of several possible operands
- *Operand a* can be
  - PC (for BR)
  - Register specified by the instruction (for ADDI, SUBI,…)
  - Special register
    - R2 (i.e. RF[2], which contains position, for MOVR, MOVRHS)
    - R0 (for SR0, SRH0)

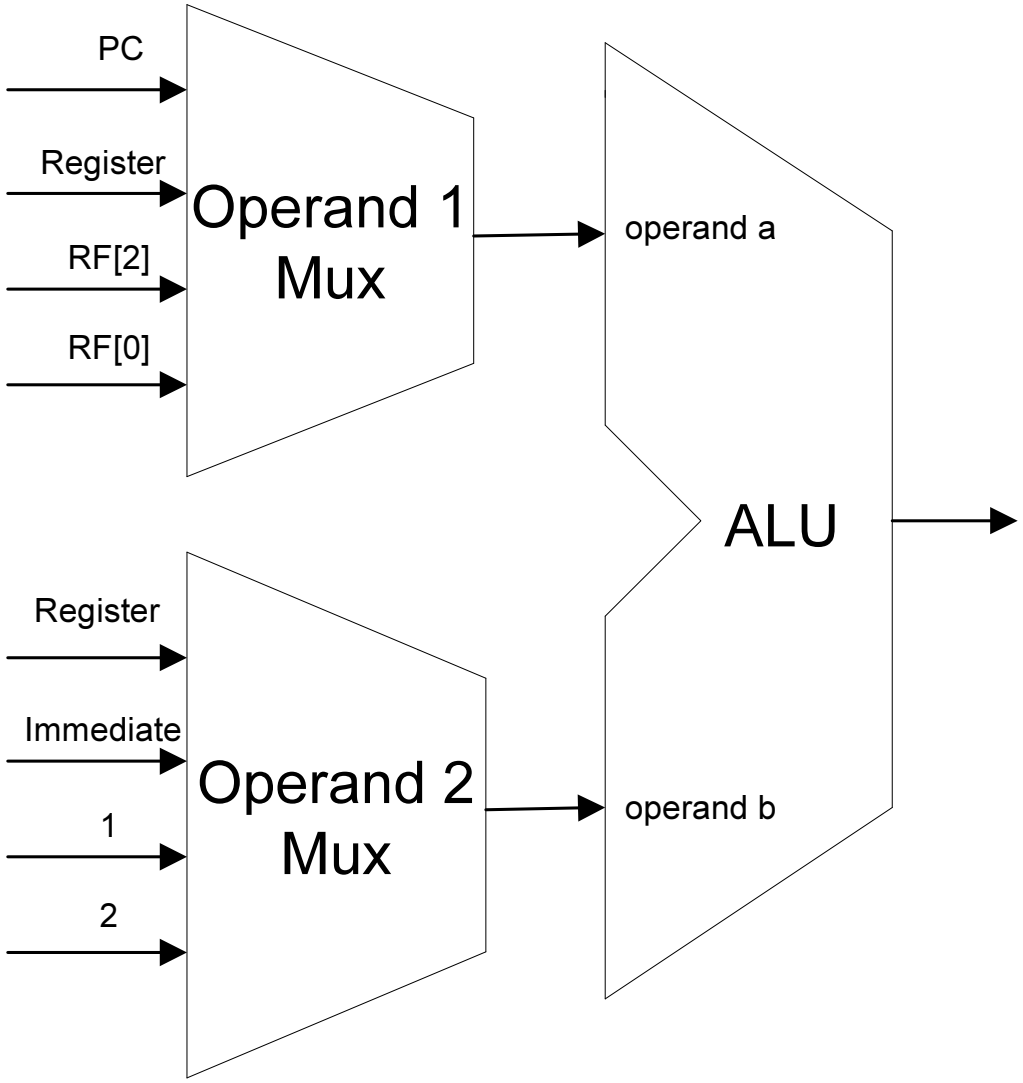


Datapath: Operand Selection…

- *Operand b* can be
  - Register specified by the instruction (for MOVA – bonus)
  - Immediate (for ADDI, SUBI,…)
  - Constant value 1 (for MOVRHS)
  - Constant value 2 (for MOVR)



Datapath: ALU + Muxes

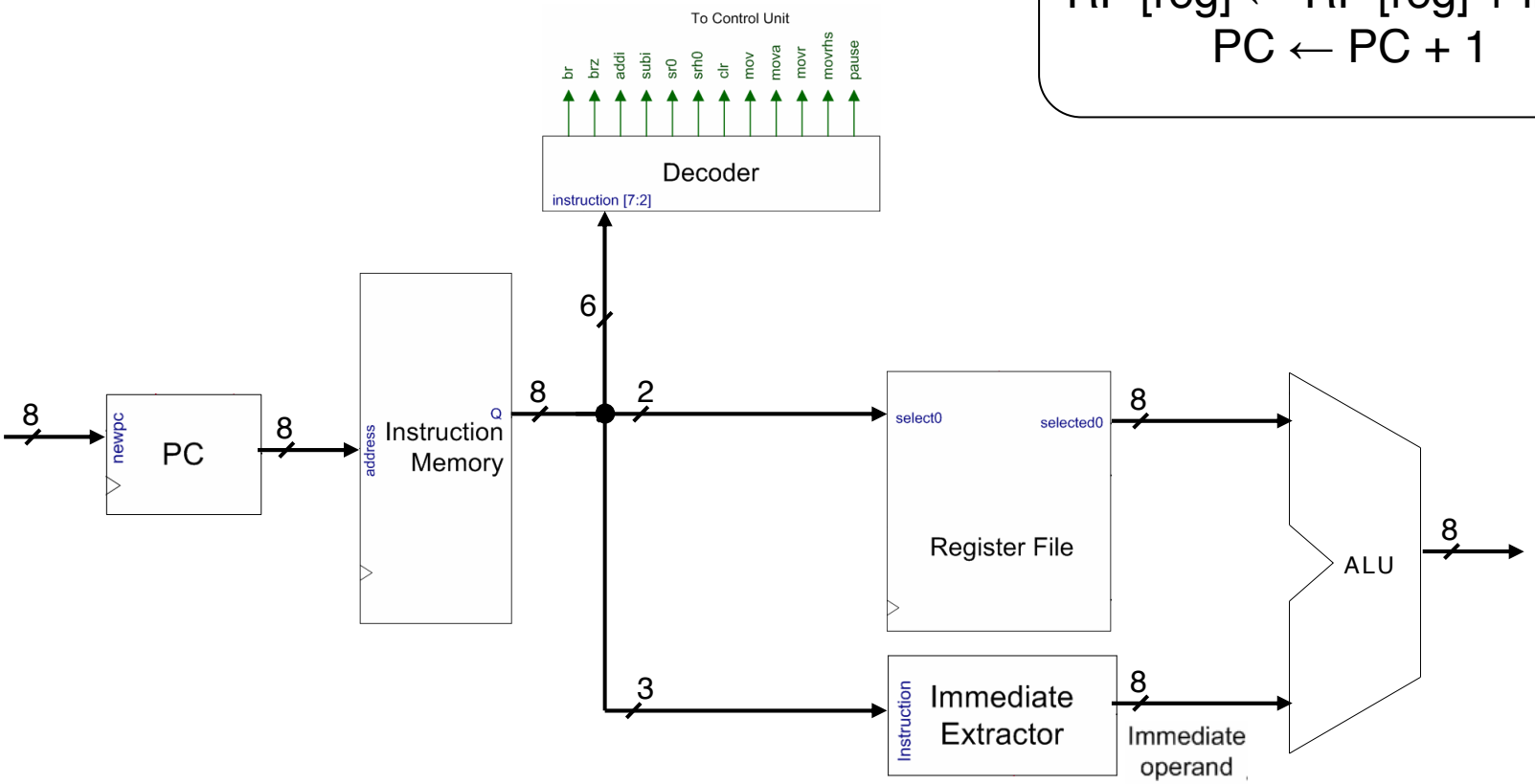


Datapath: Computation

- One ALU will perform all functions
  - `ADDI reg, imm3`
  - For ADDI it will perform addition

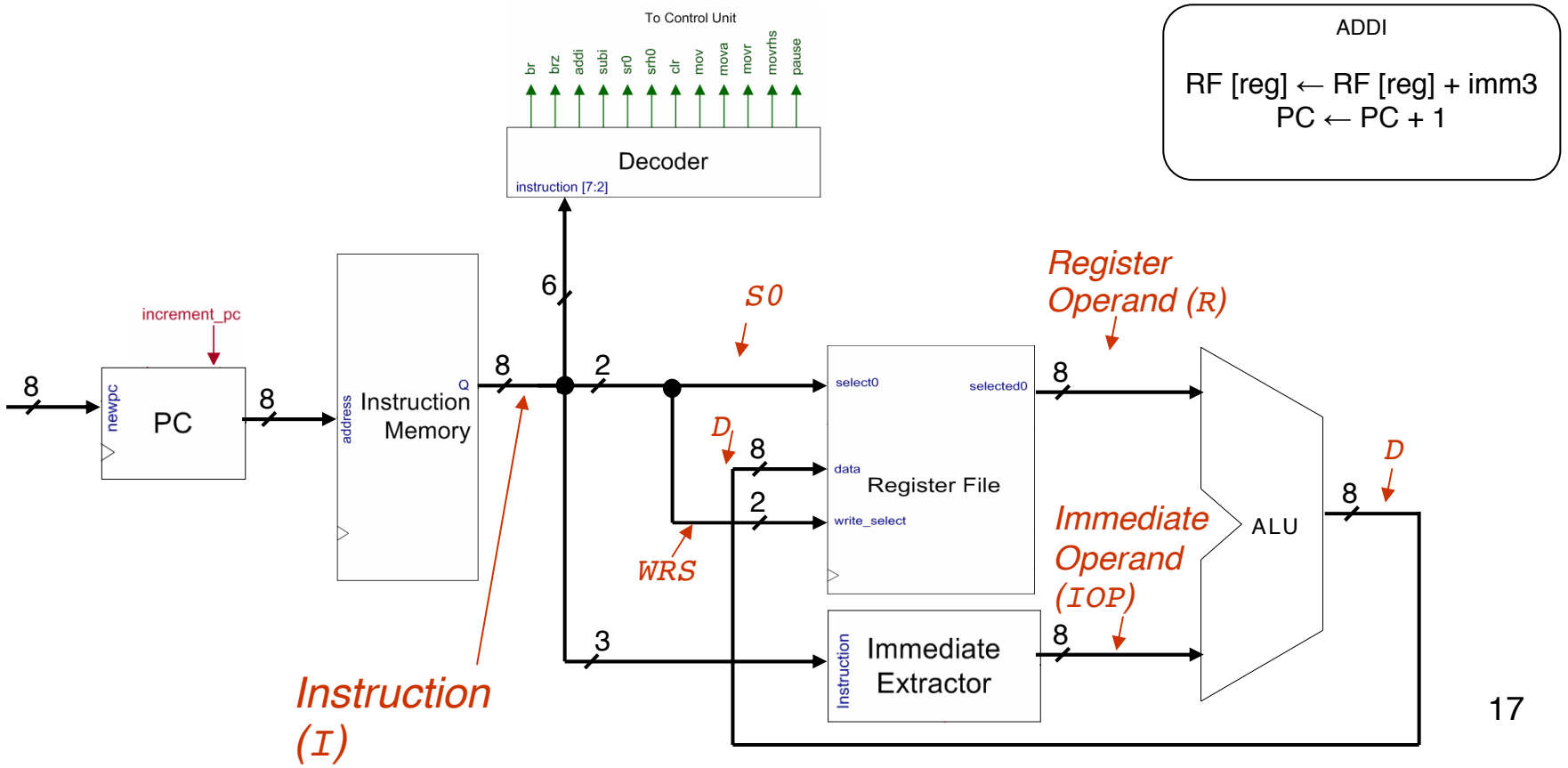
ADDI

RF [reg] ← RF [reg] + imm3
PC ← PC + 1



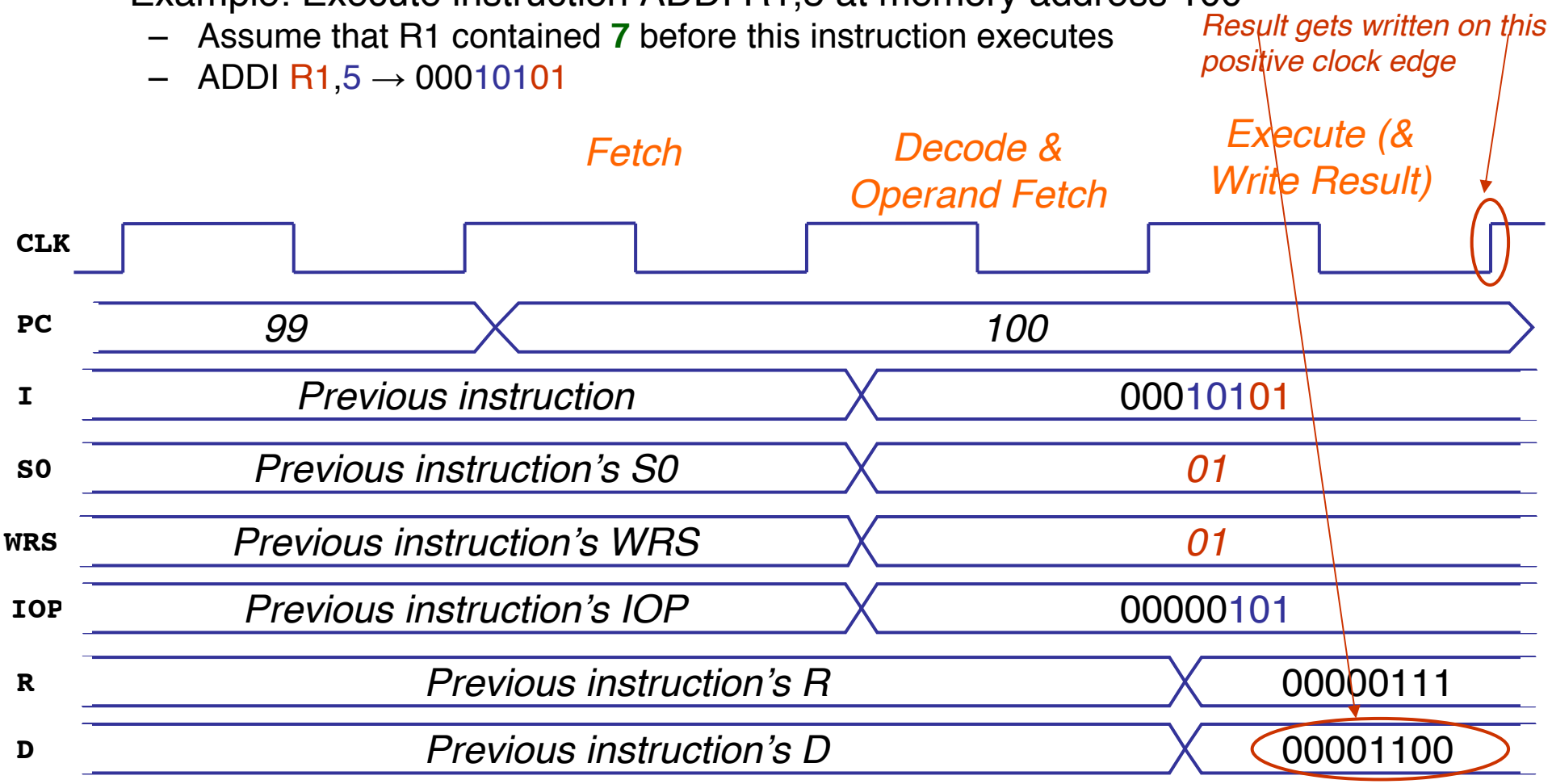
Datapath: Writing the Result

- `ADDI reg, imm3`
- The result is written back to the same register



ADDI: Cycle by Cycle

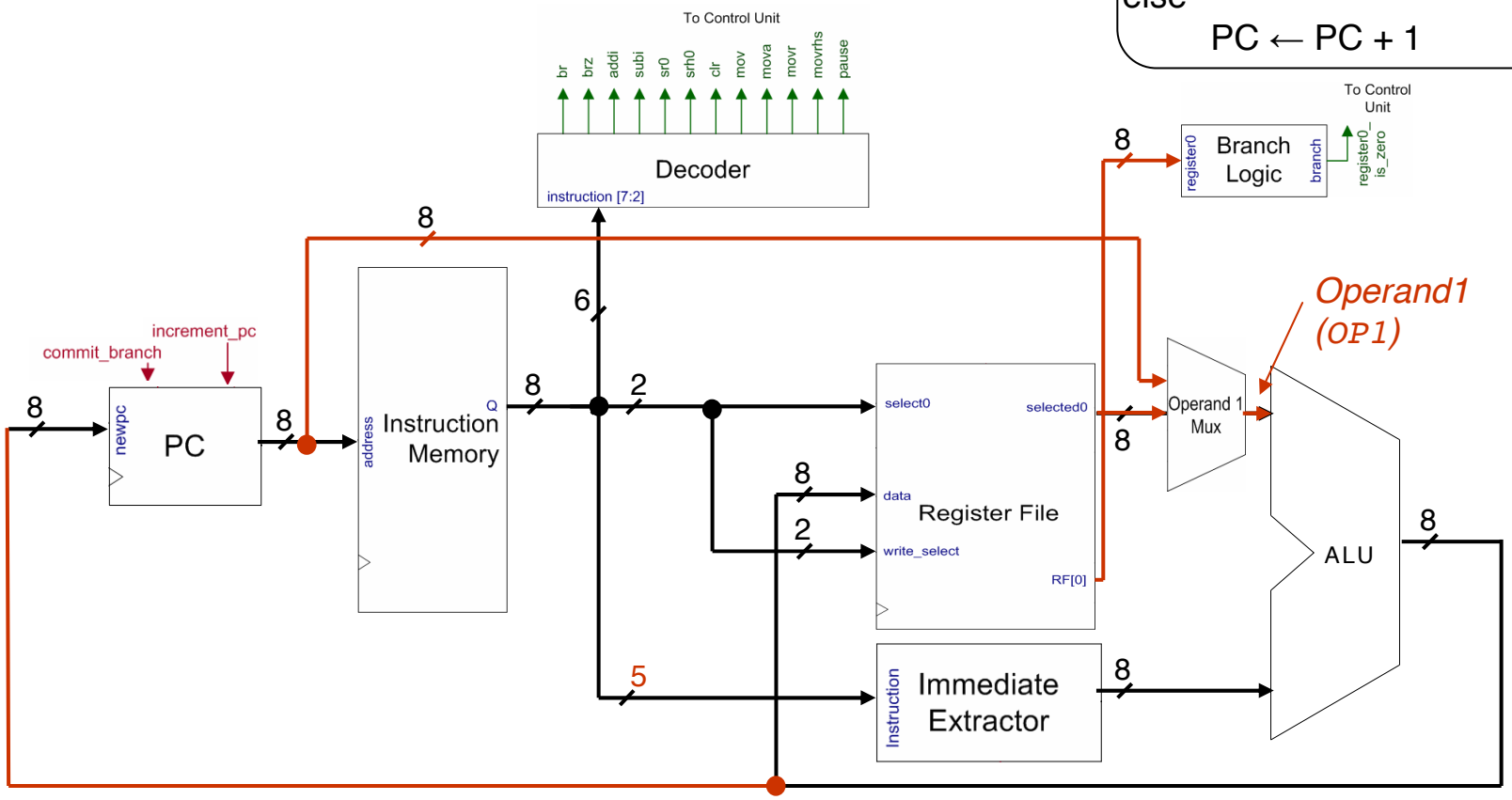
- Example: Execute instruction ADDI R1,5 at memory address 100
  - Assume that R1 contained **7** before this instruction executes
  - ADDI R1,5 → 00010101



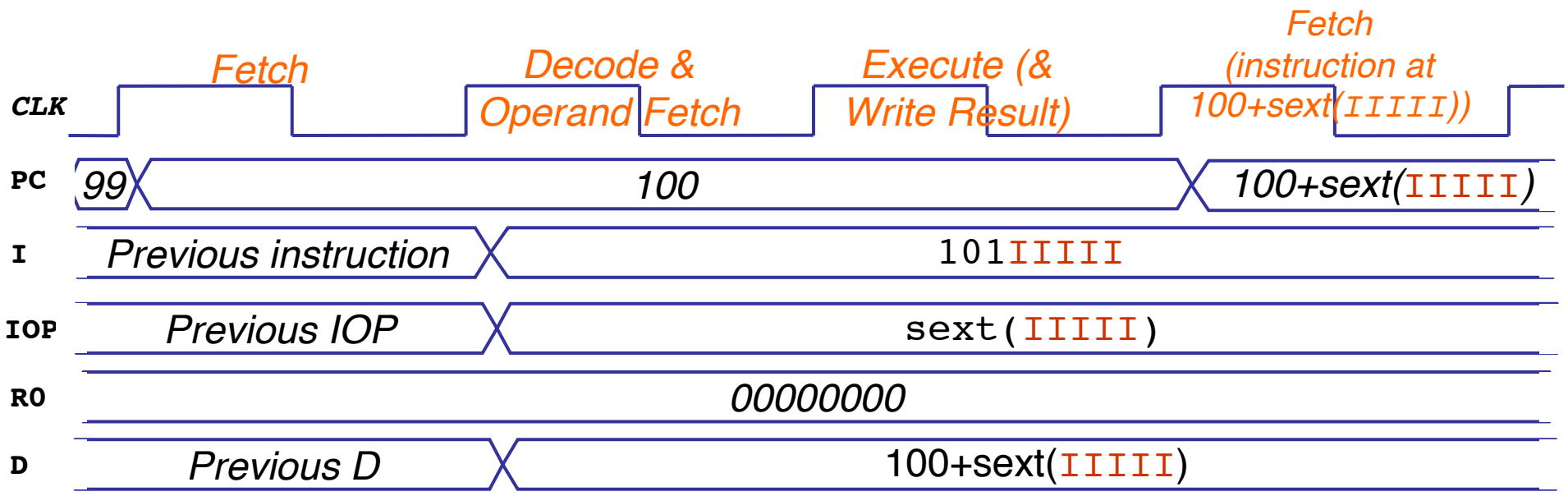
Datapath: BRZ

- What do we need to add to the datapath to implement the BRZ instruction
  - BRZ imm5

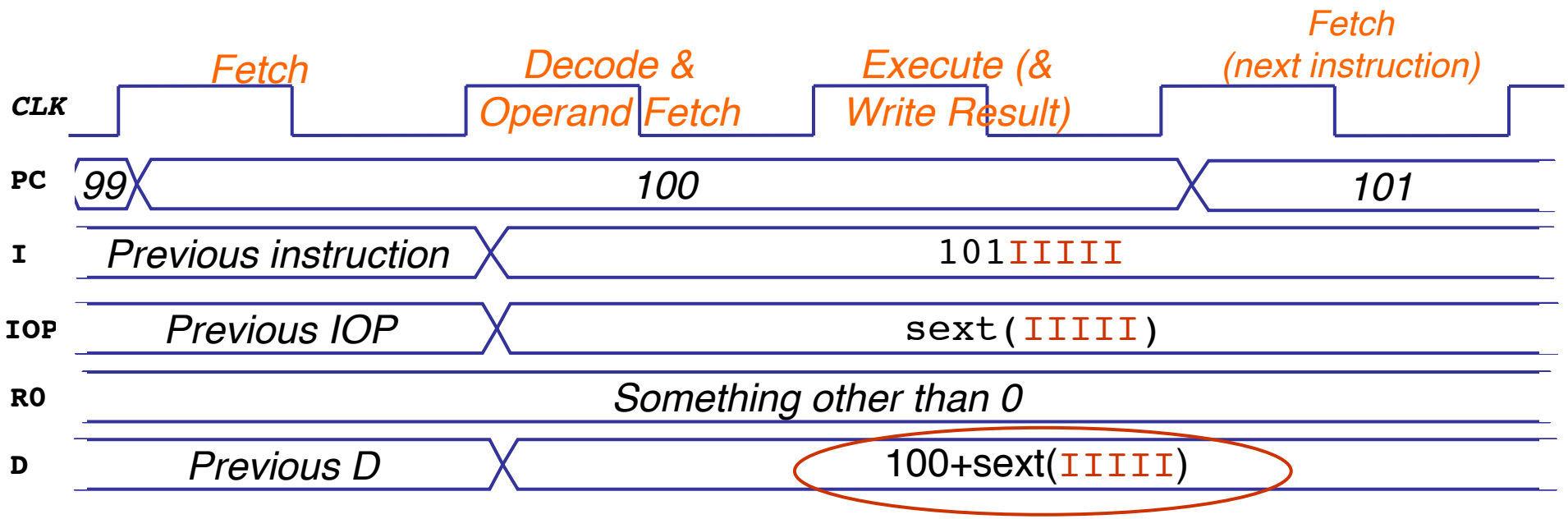
```
BRZ
if (RF[0] == 0)
        PC ← PC + sext(imm5)
else
        PC ← PC + 1
```



BRZ: Cycle by Cycle (Branch Taken)



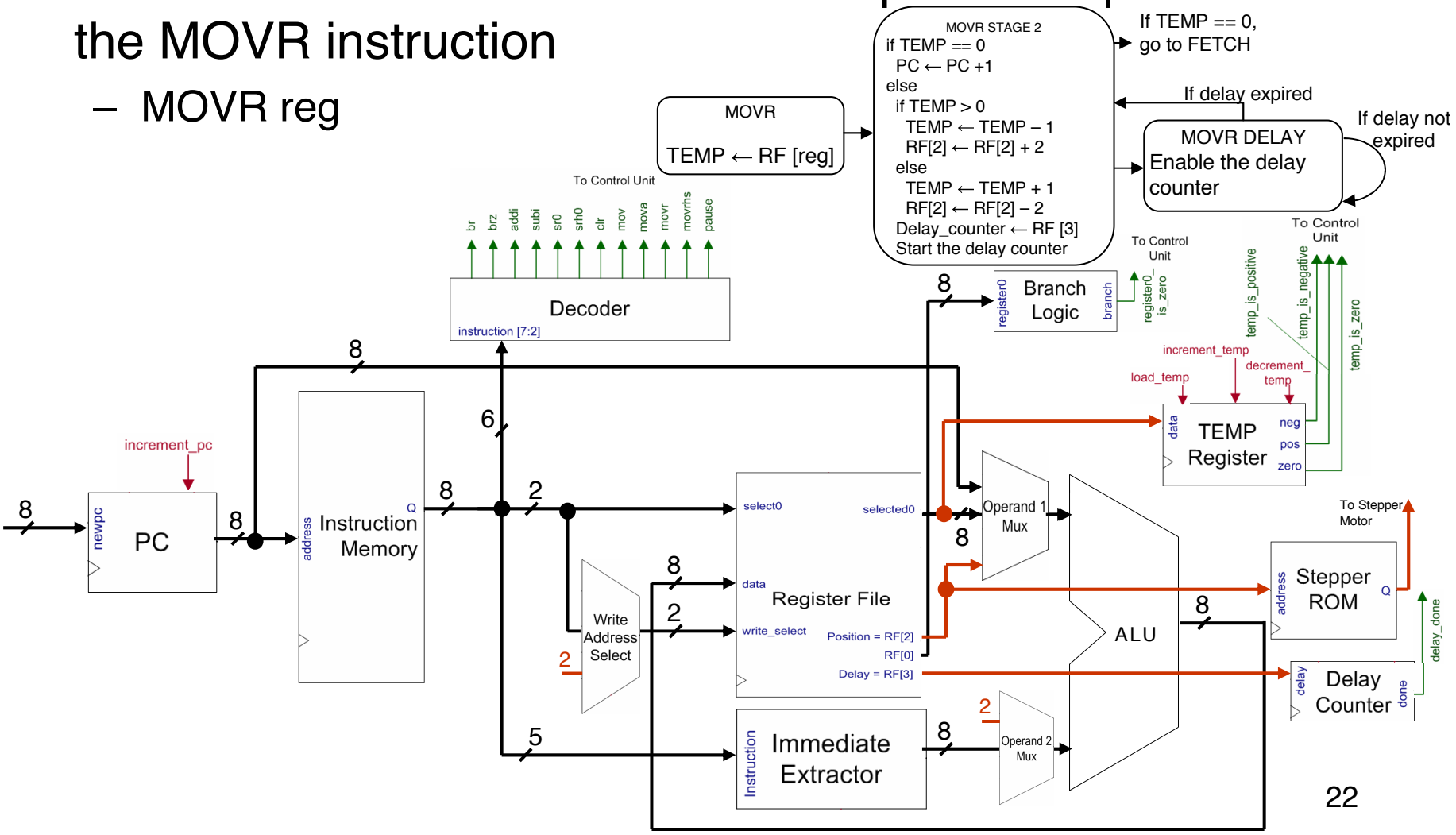
BRZ: Cycle by Cycle (Branch Not Taken)



May or may not compute this:
It doesn't matter because it's not used!

Datapath: MOVR

- What do we need to add to the datapath to implement the MOVR instruction
  - MOVR reg



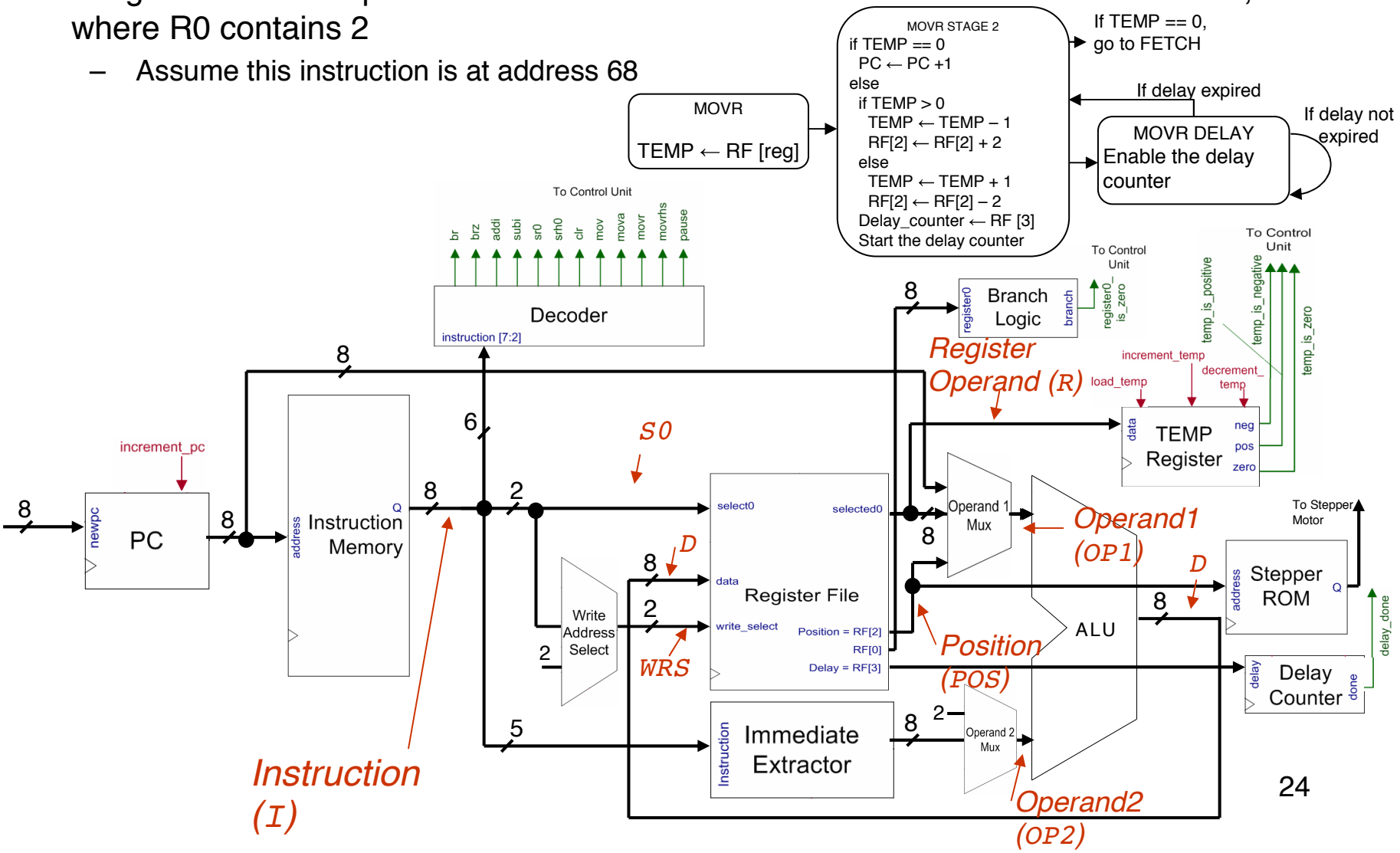
Stepper ROM

- How does stepper ROM work?
  - It contains eight 4-bit combinations
    - Correspond to the table of signals that need to be driven to a stepper motor to perform half-stepping
  - Lowest three bits of the position register are connected to the address port of this ROM
  - Consider what happens when the position is 6 and gets increased by 1 or 2?

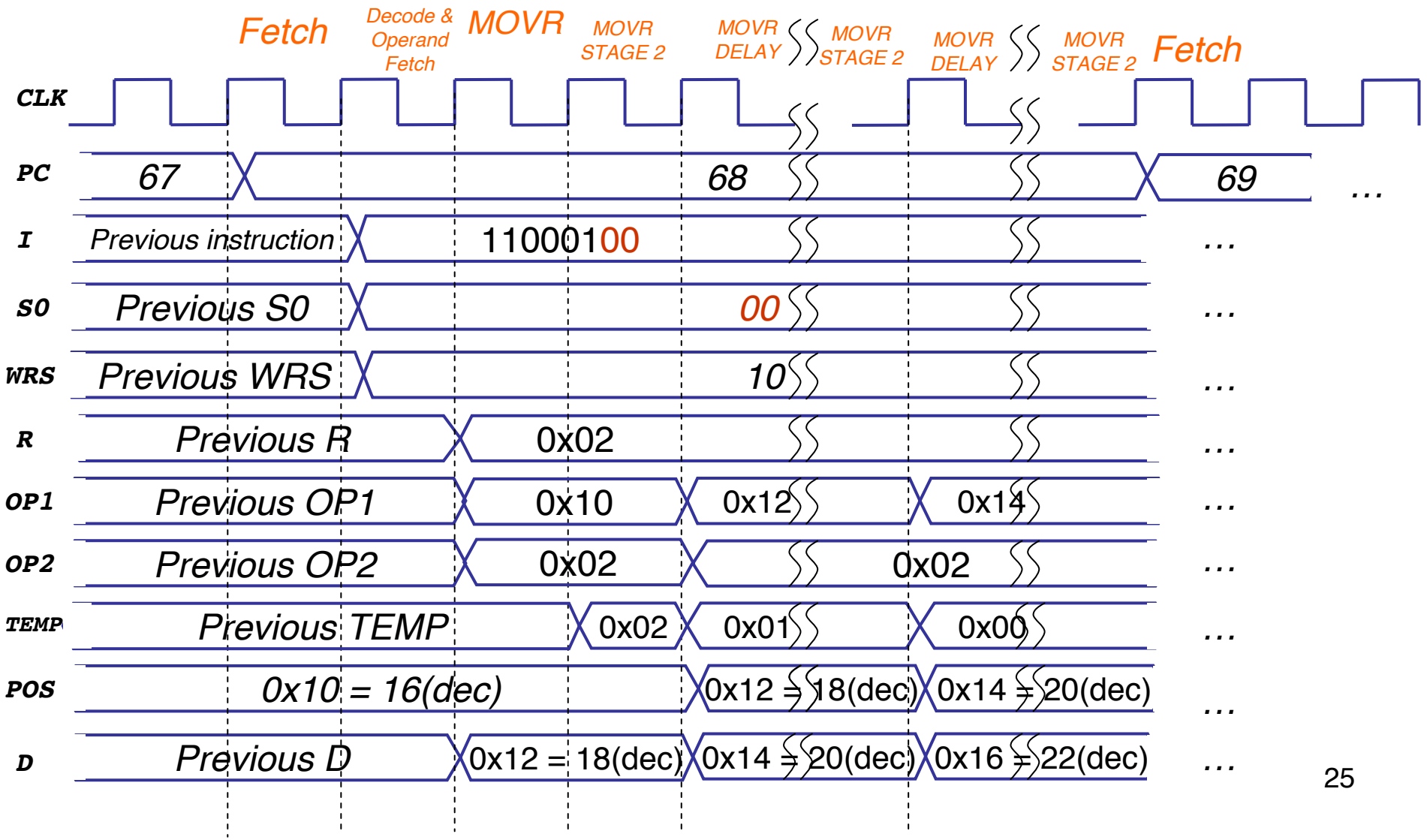
Location	b_3	b_2	b_1	b_0
000	–	–	+	–
001	–	+	+	–
010	–	+	–	–
011	–	+	–	+
100	–	–	–	+
101	+	–	–	+
110	+	–	–	–
111	+	–	+	–

MOVR: Cycle By Cycle

- Imagine the current position is 16 and the instruction to be executed is MOVR R0, where R0 contains 2
  - Assume this instruction is at address 68

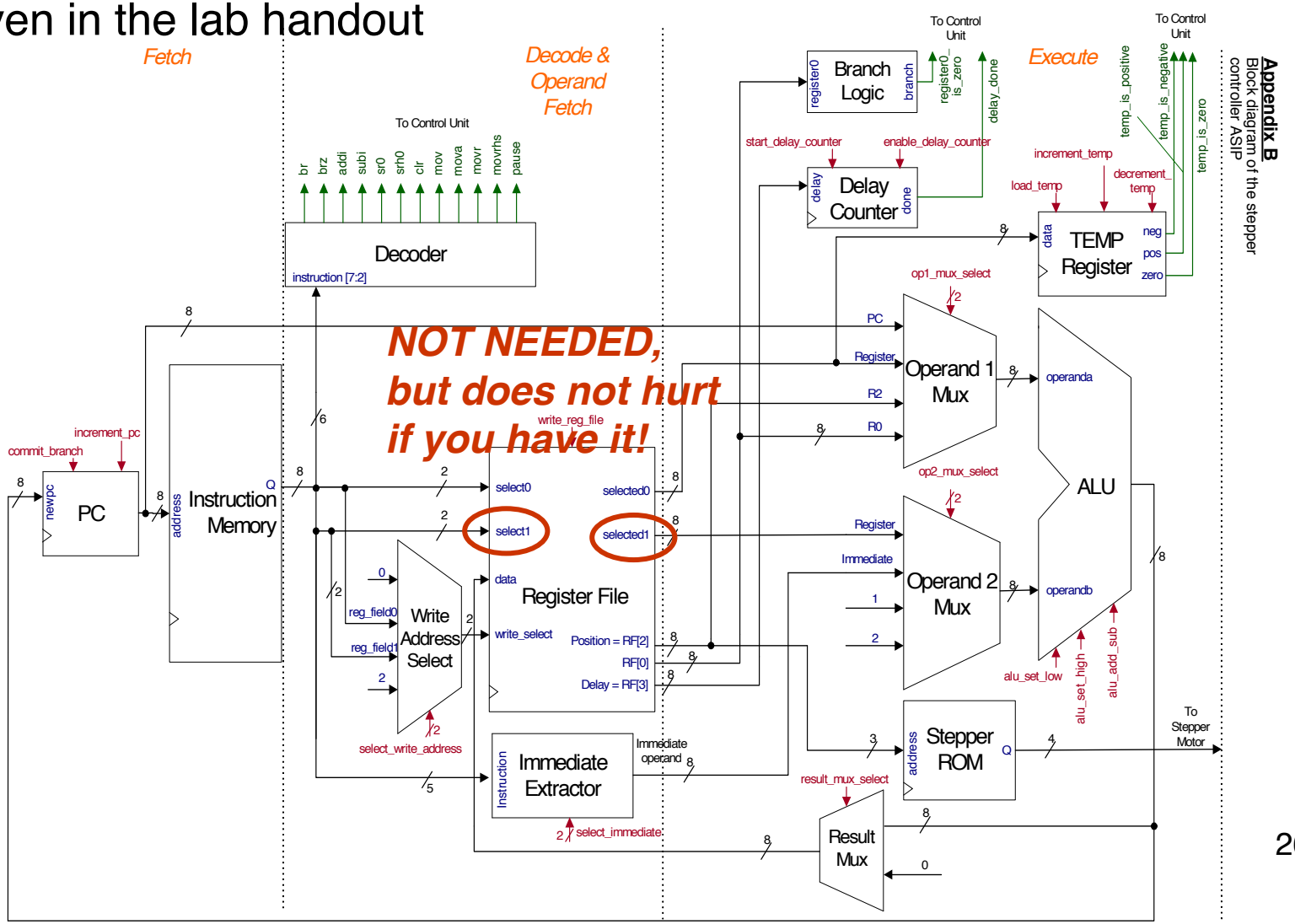


MOVR: Cycle by Cycle



Finishing the Datapath

- Continuing this procedure for all instructions, gets us to the datapath given in the lab handout



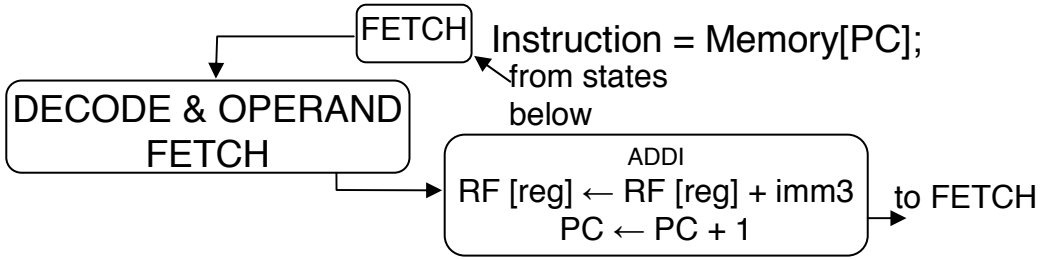
Control Signals

- The datapath includes many control signals, which need to be set appropriately to have the datapath perform the operations we want them to perform
- Analogy
  - Like a pipe system with many valves
  - We need to open correct valves to direct the fluid flow

Determining Control Signals

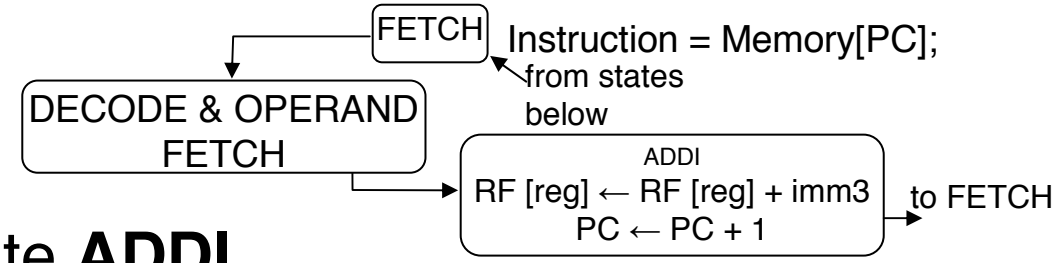
- To determine how the control signals should be set, we just read the FSMD and look at the datapath
  - Ask yourself: Which “valves” should be open to get the data to flow through the datapath to implement operations specified in the FSMD?
  - This can be done one state at a time
    - No need to worry how states interact; we took care of that when we were designing the FSMD

Control Signals: ADDI



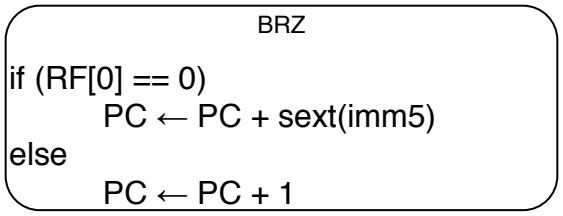
- State: **Fetch**
  - No special considerations (for any instruction)!
- **Decode and operand fetch**
  - No special considerations (for any instruction)!
    - You could set the select_immediate signal if it makes you happy ☺
    - Not needed until the next state, so not necessary to select it here because immediate extractor is combinational!

Control Signals: ADDI…



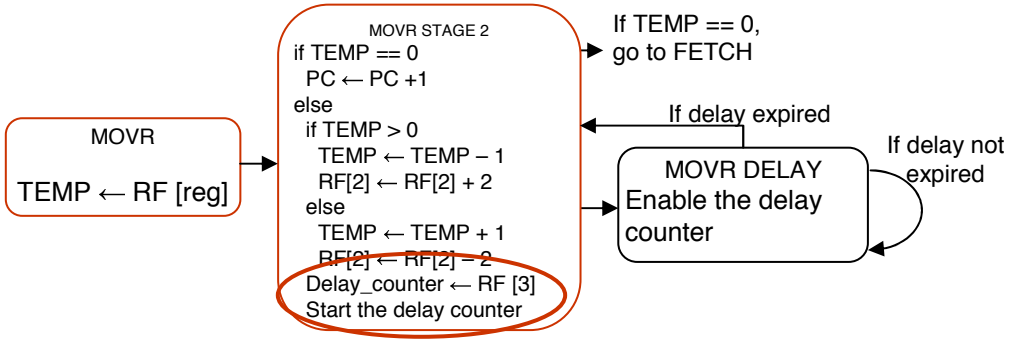
- State **ADDI**
  - Select the two operands:
  - Operand 1 is RF[reg] → op1_mux_select = 2'b01
  - Operand 2 is immediate → op2_mux_select = 2'b01
  - Immediate should be selected to be the 3-bit immediate → select_immediate = whichever_code_you_assigned
  - Operation should be selected to be addition → alu_add_sub = 1'b0
  - Result should come from ALU → result_mux_select = 1'b1 (providing that is how you implemented result mux)
  - Result should be written to RF[reg] → select_write_address = 2'b01
  - Results should indeed be written → write_reg_file = 1'b1
  - PC should be incremented → increment_pc = 1'b1
  - All other signals should be INACTIVE

Control Signals: BRZ



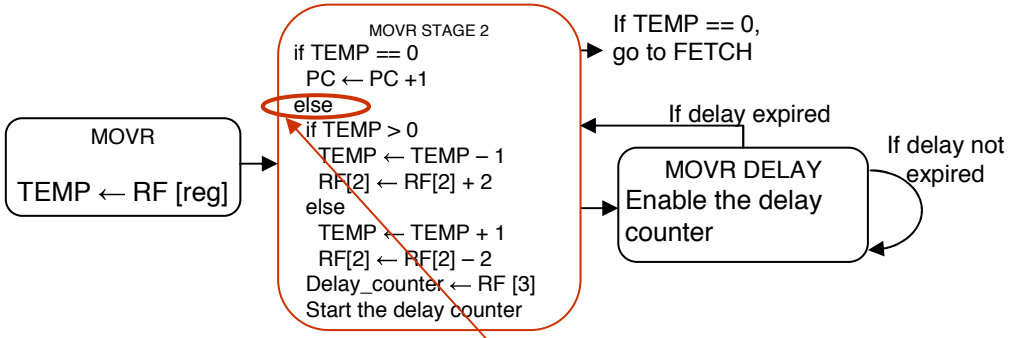
- State **BRZ**
  - if register0_is_zero
    - Select the two operands
    - Operand 1 is PC → op1_mux_select = 2'b00
    - Operand 2 is immediate → op2_mux_select = 2'b01
    - Immediate should be selected to be the 5-bit immediate → select_immediate = whichever_code_you_assigned
    - Operation should be selected to be addition → alu_add_sub = 1'b0
    - Result should be written to PC → commit_branch = 1'b1
  - else
    - PC should be incremented → increment_pc = 1'b1
  - All other signals should be INACTIVE

Control Signals: MOVR



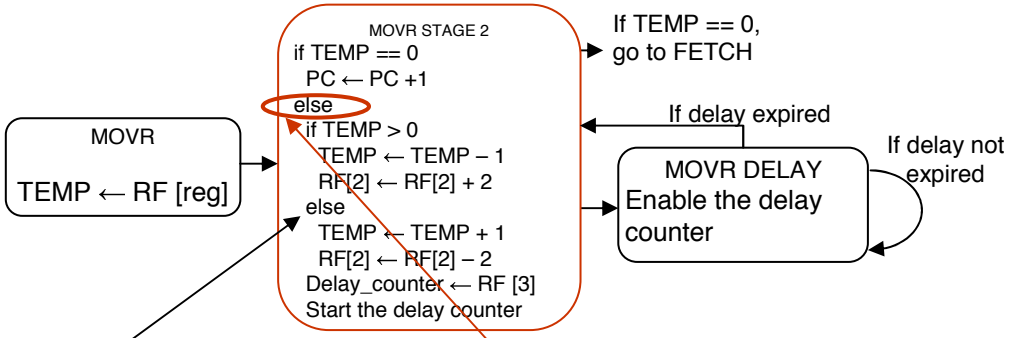
- State **MOVR**
  - RF[reg] is already connected directly to the TEMP register
  - Only need to enable the TEMP register → load_temp = 1'b1
- State **MOVR STAGE 2**
  - if temp_is_zero
    - increment_pc = 1'b1
  - else
    - Delay counter is already connected to RF[3]
    - Only need to enable it → start_delay_counter = 1'b1

Control Signals: MOVR…



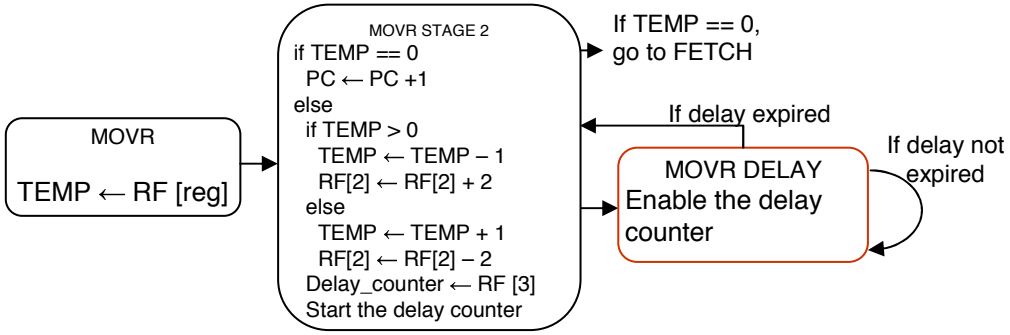
- State **MOVR STAGE 2 (continued)**
- Everything below is under the 'else' branch from the previous slide
  - if temp_is_positive
    - decrement_temp = 1'b1
    - Select the two operands for RF[2] ← RF[2] + 2
    - Operand 1 is RF[2] → op1_mux_select = 2'b10
    - Operand 2 is constant 2 → op2_mux_select = 2'b11
    - Operation should be selected to be addition → alu_add_sub = 1'b0
    - Result should come from ALU → result_mux_select = 1'b1 (providing that is how you implemented result mux)
    - Result should be written to RF[2] → select_write_address = 2'b11
    - Results should indeed be written → write_reg_file = 1'b1

Control Signals: MOVR…



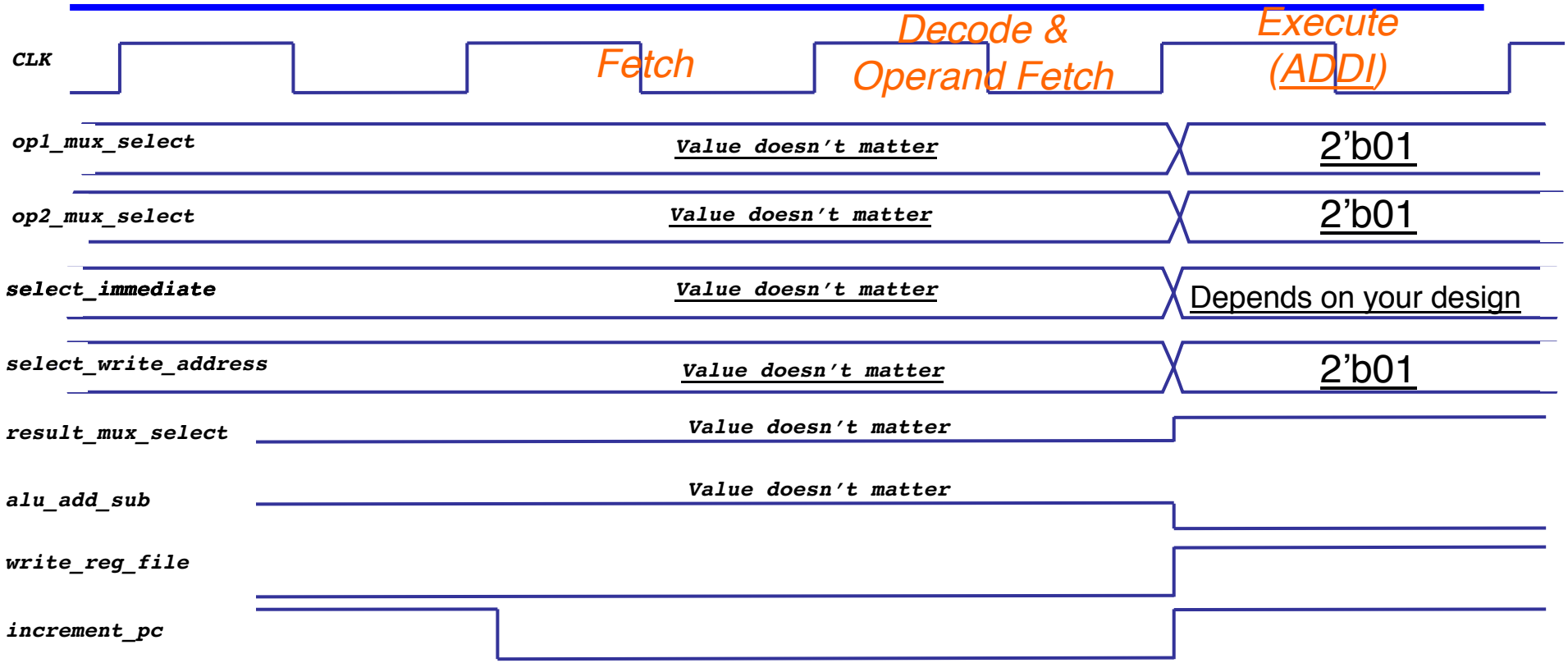
- State **MOVR STAGE 2 (continued)**
- Everything below is under the 'else' branch from the previous slide
  - else (i.e. if temp_is_negative)
    - increment_temp = 1'b1
    - Select the two operands for RF[2] ← RF[2] – 2
    - Operand 1 is RF[2] → op1_mux_select = 2'b10
    - Operand 2 is constant 2 → op2_mux_select = 2'b11
    - Operation should be selected to be addition → alu_add_sub = 1'b1
    - Result should come from ALU → result_mux_select = 1'b1 (providing that is how you implemented result mux)
    - Result should be written to RF[2] → select_write_address = 2'b11
    - Results should indeed be written → write_reg_file = 1'b1

Control Signals: MOVR…



- State **MOVR DELAY**
  - We only need to enable the delay counter
  - enable_delay_counter = 1'b1
  - All other signals should be INACTIVE
  - For enable-type signals (e.g. load, write, …), "inactive" means not active
  - For other signals (such as select), inactive means that we do not really care what their values are
    - e.g. if nothing is being written to the register file, it doesn't matter which address the Write Address Select multiplexer is selecting

Simulating the Datapath



- Assert signals according to the scheme we have developed
  - You will have to put some instructions into the instruction memory to be able to simulate this

Simulating the Conditionals

- When we were devising the control signals to be asserted, we had conditions
  - e.g. State **BRZ:**
    - if register0_is_zero
      ...
    - else
      ...
- Since you will have full control of the processor, you will know whether the value in the register R0 is 0 or not
  - Therefore, you will know which signals to assert
  - You should test both cases, to make sure that datapath performs correct operation in both branches of the if statement

Simulating the Delay

- As stated in the lab handout, the delay in 1/100's of a second takes long to simulate
  - You should make your delay circuit parameterizable, so that its fundamental delay period can be reduced and the simulation does not take long
    - We will see shortly how this can be done
  - If you are not sure when the delay circuit will produce the delay_done signal, simulate it and see, then set other signals that should depend on it appropriately

Verilog: Combinational Logic

- Multiplexers should be easy to implement by now
- ALU and immediate extractor are simply some logic that computes something, followed by a multiplexer
  - One thing that might confuse you is the ALU, because it has more than 2 lines to encode 4 operations

ALU in Verilog

- Imagine you are implementing an ALU, which should compute 4 operations
  - a+b
  - a-b
  - b-a
  - a*b
- Imagine there are 3 control signals
  - multiply
  - compute_b_minus_a
  - add_sub
- Meaning of these control signals is as follows
  - If multiply is high, compute a*b
  - If compute_b_minus_a is high, compute b-a
  - If neither of the above is true
    - If add_sub is 0
      - Compute a+b
    - Else
      - Compute a-b

ALU in Verilog…

- In Verilog, we could describe this as follows

```
always @(*)
begin
   if (multiply == 1'b1)
       result = a * b;
   else if (compute_b_minus_a == 1'b1)
       result = b - a;
   else if (add_sub == 1'b0)
       // Add
       result = a + b;
   else
       // Subtract
       result = a - b;
end
```

Counter with Load

- We already know how to implement a counter (Lecture 10/11)

```
module counter_n_bit (input clk, input reset, output
   reg [N-1:0] count);
parameter N=8;

always @(posedge clk, posedge reset)
begin
   if (reset)
       count <= {N{1'b0}};
   else
       count <= count + 1'b1;
end

endmodule
```

Caution: This reset is active high!

- What does this counter count?
  - Number of positive clock edges
- We want to add to it the ability to enable/disable the counting

Counter with Count Enable

- We already know how to implement a counter (Lecture 10/11)

```
module counter_n_bit (input clk, input reset, input cnt_en,
   output reg [N-1:0] count);
parameter N=8;

always @(posedge clk, posedge reset)
begin
   if (reset)
        count <= {N{1'b0}};
   else
   begin
        if (cnt_en == 1'b1)
                count <= count + 1'b1;
   end
end

endmodule
```

- We want to add to it the ability to load an arbitrary value into the counter

Counter with Count Enable and Load

- We already know how to implement a counter (Lecture 10/11)

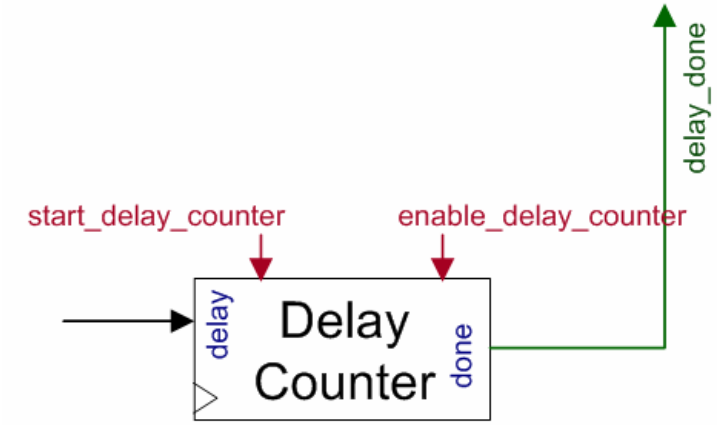
```
module counter_n_bit (input clk, input reset, input load, input [N-1:0]
    load_data, output reg [N-1:0] count);
parameter N=8;

always @(posedge clk, posedge reset)
begin
    if (reset)
          count <= {N{1'b0}};
    else
    begin
          if (load_data == 1'b1)
          begin
                    count <= load_data;
          end
          else
          begin
                    if (cnt_en == 1'b1)
                              count <= count + 1'b1;
          end
    end
end

endmodule
```

Delay Counter

- Delay Counter is the only somewhat tricky part of the datapath



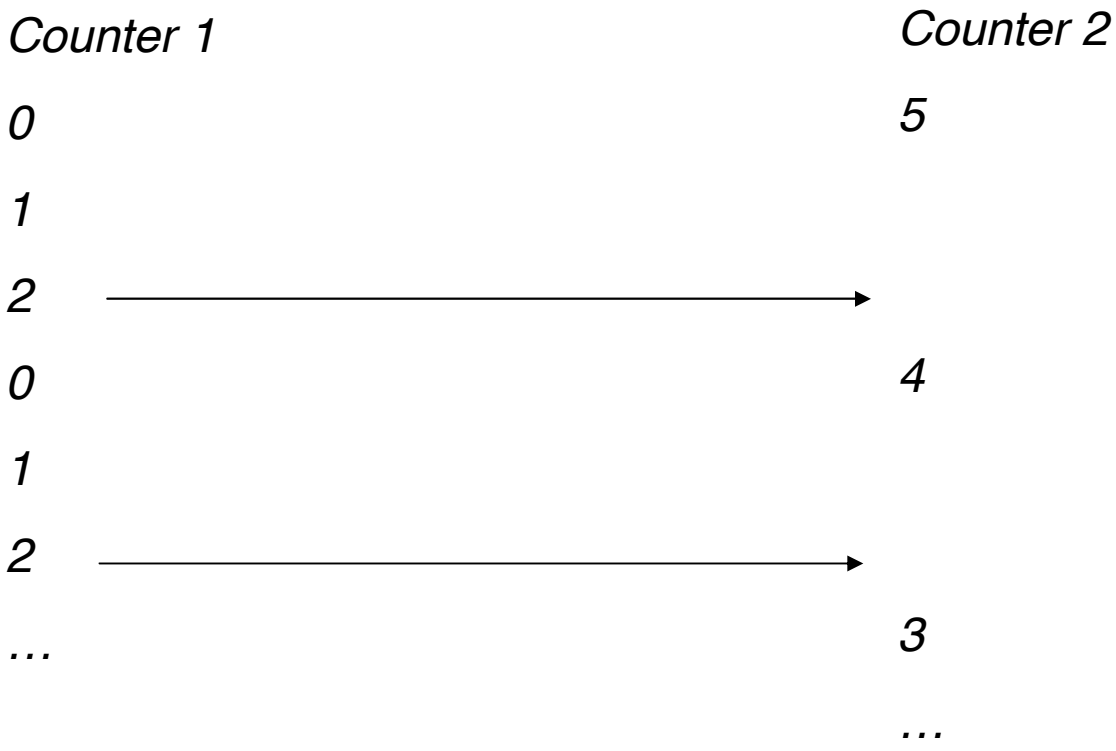
- Delay counter loads the amount of delay when the start_delay_counter is high
- When the enable_delay_counter is high, the counter counts down, until it reaches 0
  - It asserts delay_done signal once it reaches 0

Delay Counter Design

- Since counters count clock edges, and our clock's frequency is 50 MHz, we cannot directly count the time intervals specified in 1/100's of a second
- Instead, you should apply the technique that you used when programming AVR's timer
- Employ 2 counters
  - Have one counter that tells you each time a 1/100 of a second has passed
  - The other counter can be loaded with the desired delay, and decremented (i.e. decreased by 1) every time the first counter indicates that a 1/100 of a second has passed
    - Once this other counter reaches 0, the desired interval has passed

Delay Counter Example

- For simplicity of the example, imagine the first counter has period of 3 clock cycles



Modulo-N counter

- We know that binary counters are normally modulo 2^n, where n is the number of bits
- What if we want a counter that counts modulo arbitrary number N, which is not a power of 2?

Modulo-312 counter

- Count: 0, 1, 2, …, 310, 311, 0, 1, 2, …
- 312 can be stored in 9 bits

```
always @(posedge clk)
begin
   if (start == 1'b1)
       count <= 9'b0;
   else if ((cnt_enable == 1'b1) && (count < 9'd311))
        count <= count + 1'b1;
   else if ((cnt_enable == 1'b1) && (count == 9'd311))
        count <= 9'b0;
end
```