Verilog Notes - 161 Lab

Jose M. Rodriguez
jrodr050@ucr.edu
Chung 464

Embedded System Lab
UC Riverside

May 1, 2017

FMS Mealy Vs Moore

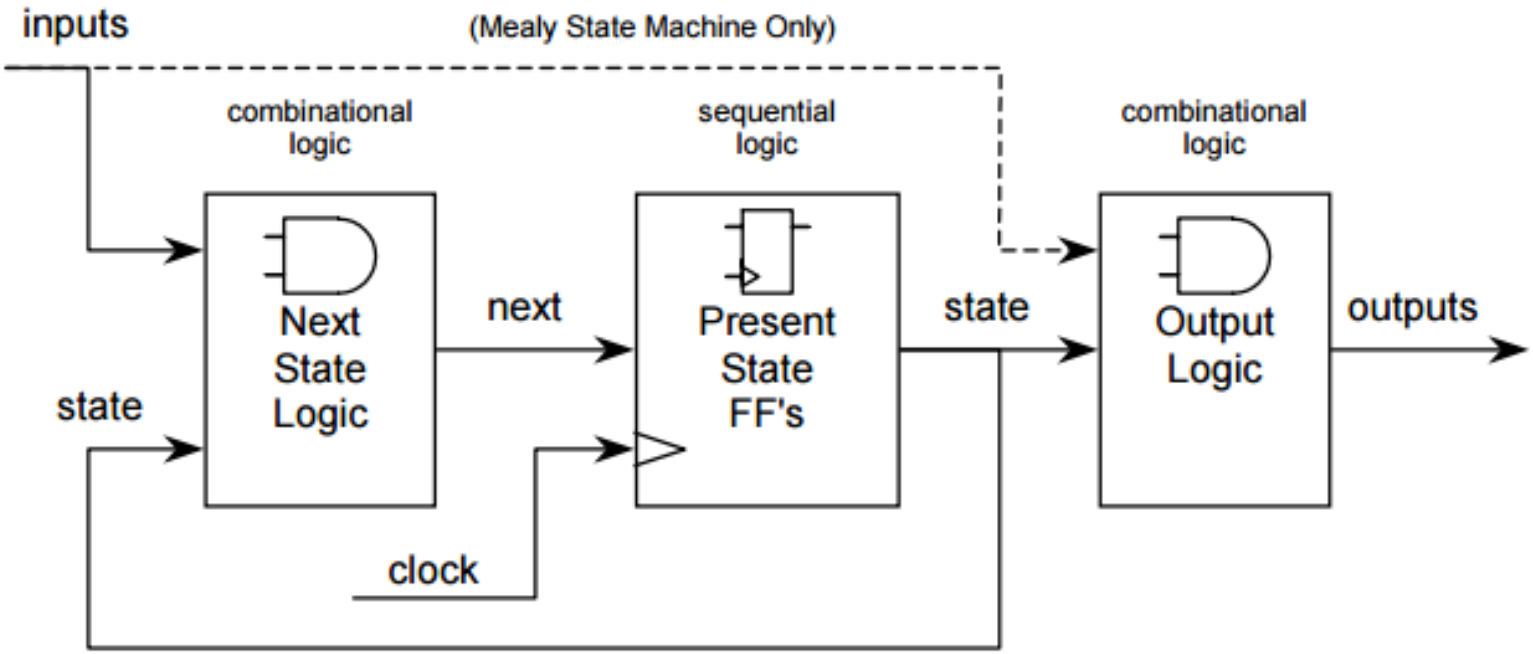

Figure 1 - FSM Block Diagram

Source : "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs." by Cummings, C.E.

FMS State Diagram

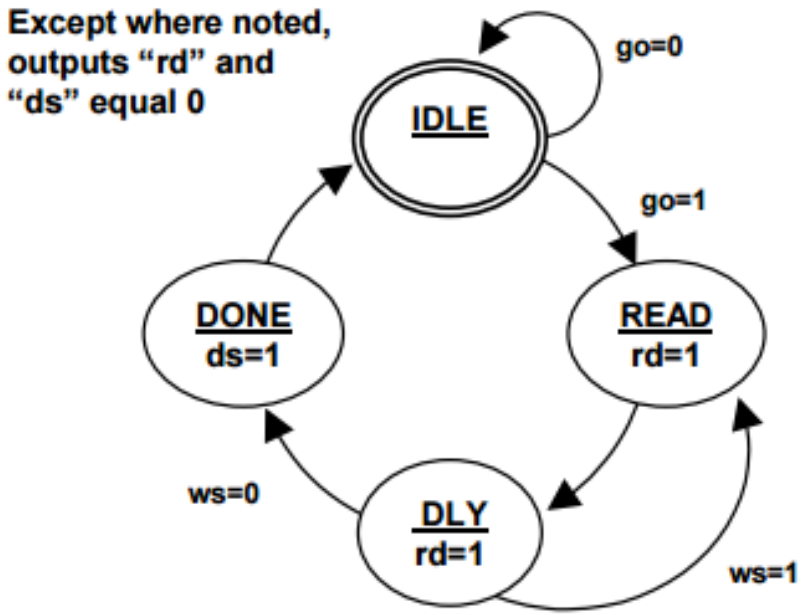

Figure 2 - FSM1 State Diagram

FMS Coding - Right Code

```
module fsm1a (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;

  parameter [1:0] IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or go or ws) begin
    next = 2'bx;
    case (state)
      IDLE: if (go) next = READ;
            else    next = IDLE;

      READ:         next = DLY;

      DLY:  if (ws) next = READ;
            else    next = DONE;

      DONE:         next = IDLE;

    endcase
  end

  assign rd = (state==READ || state==DLY);
  assign ds = (state==DONE);
endmodule
```

State register, sequential always block

Next state, combinational always block

Continuous assignment outputs

Example 1 - FSM Coding Style - Two-always blocks with continuous assignment outputs

FMS Coding - Right Code

```
module fsm1 (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;
  reg    ds, rd;

  parameter [1:0] IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or go or ws) begin
    next = 2'bx;
    ds = 1'b0;
    rd = 1'b0;
    case (state)
      IDLE: if (go)    next = READ;
            else       next = IDLE;

      READ: begin      rd   = 1'b1;
                       next = DLY;
            end

      DLY:  begin      rd   = 1'b1;
              if (ws)  next = READ;
              else     next = DONE;
            end

      DONE: begin      ds   = 1'b1;
                       next = IDLE;
            end

    endcase
  end
endmodule
```

State register, sequential always block

Next state & outputs, combinational always block

Multi-Stage Coding (Pipelines)

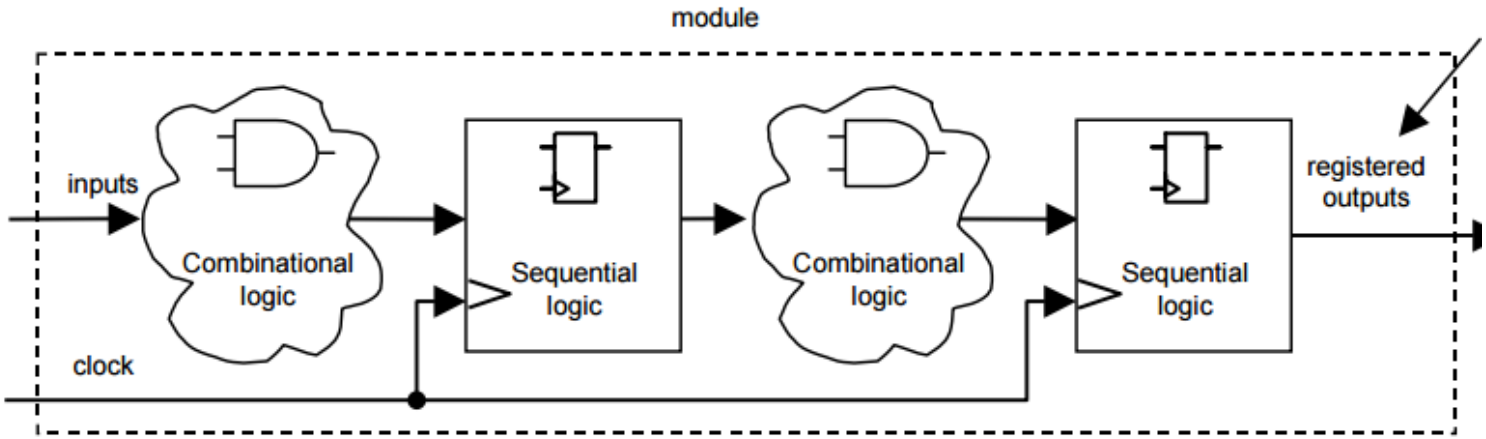

Figure 4 - Multi-stage module partition with registered outputs

Multi-Stage Coding (Pipelines)

Example : Compute in hardware the following function using a multi-stage design (Taylor Expansion)

$$\sqrt{1+x} \approx 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^5}{128} + \frac{7x^5}{256} \text{ for } |x| < 1$$

Proposed solution :

- Compute each factor in one stage .
- Register the output of each stage.
- Add the results as the computation progress.
- Reuse the previous computations i.e. reuse x^{n-1} to compute x^n.

Multi-Stage Coding (Pipelines)

First, we design and implement one stage.

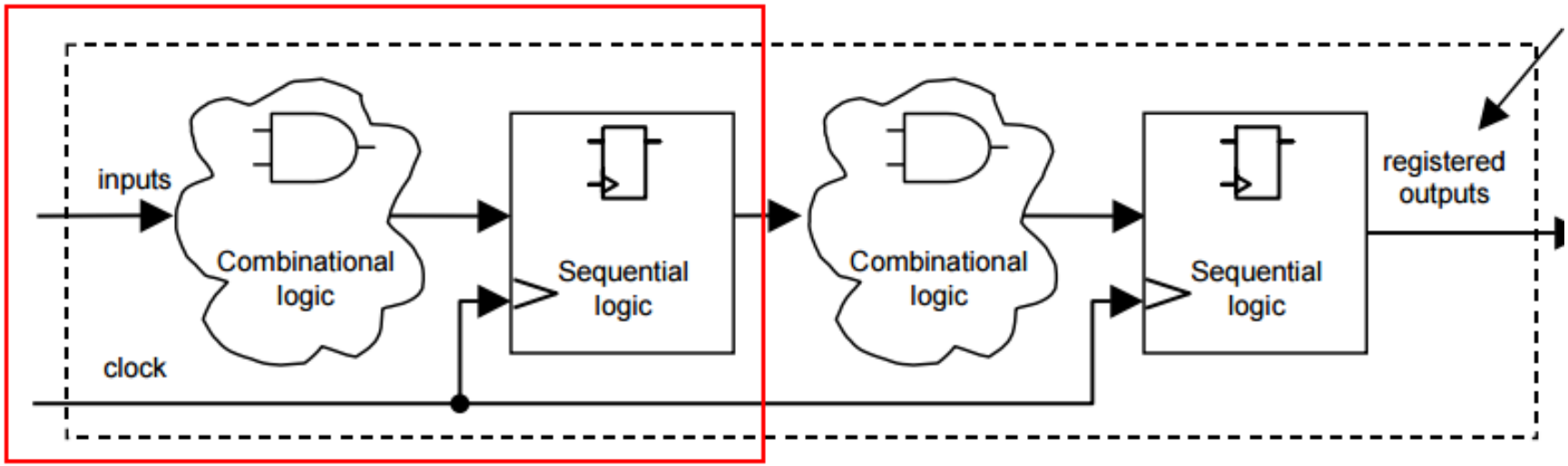

Figure 4 - Multi-stage module partition with registered outputs

Multi-Stage Coding (Pipelines)

- Inputs
- Clock
- Enable
- Add flag
- X
- X^{n-1}
- Multiplication and Division Factor
- Running Result
- Parameters

- Outputs
- Valid
- New result
- X^n

Multi-Stage Coding (Pipelines)

First, we design and implement one stage.

```
 4
 5 module fixpbinomialstep  #( parameter    NBITS = 16, parameter Q = 8 )
 6 (
 7 input wire clk,
 8 input wire enable,
 9 input wire add ,
10 input wire [NBITS-1:0 ] xn,
11 input wire [NBITS-1:0 ] xnacum ,
12 input wire [NBITS-1:0 ] mfactor ,
13 input wire [NBITS-1:0 ] dfactor ,
14 input wire [NBITS-1:0 ] oresult ,
15 output reg valid ,
16 output reg [NBITS-1:0 ] xnacumn ,
17 output reg [NBITS-1:0 ] result
18 ) ;
19
20 wire [NBITS-1 :0 ] t1, t2, t3 ;
21
22 fixpmult #( .NBITS (NBITS), .Q(Q) ) m1 ( .a( xn ),  .b( xnacum ), .r(t1)   ) ;
23 fixpmult #( .NBITS (NBITS), .Q(Q) ) m2 ( .a(mfactor), .b(t1),     .r(t2) ) ;
24
25 assign t3 =  (add == 1'b1) ?  $signed(oresult) +  $signed(t2 >>> dfactor) :
26                              $signed(oresult) -  $signed(t2 >>> dfactor) ;
27
28 always @( posedge clk )  begin
29
30     valid   <= enable ;
31     xnacumn <= t1 ;
32     result  <=  t3 ;
33
34 end
35 endmodule
36
```

Multi-Stage Coding (Pipelines)

Now, all stages together

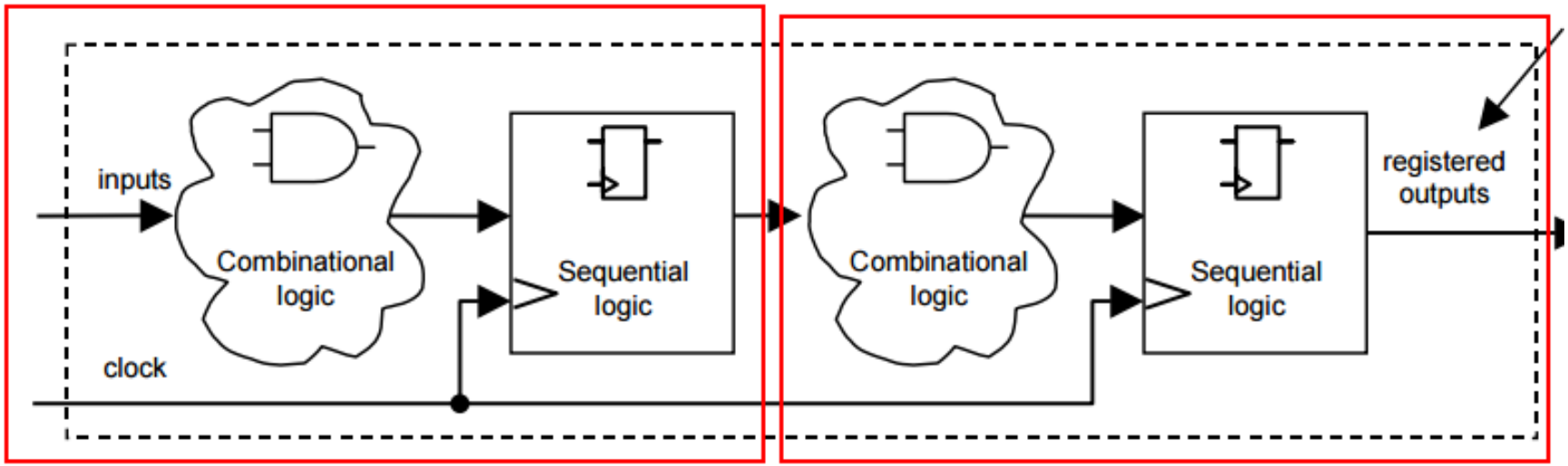

Figure 4 - Multi-stage module partition with registered outputs

Multi-Stage Coding (Pipelines)

Next, all stages together

```
35 // ----------------------------------------------------
36 // First Iteration  r = 1 + x/2
37 // ----------------------------------------------------
38
39 wire [NBITS-1 :0 ] ONE ;
40 assign ONE = { {NBITS-Q-1{1'b0}}, {1'b1} , {Q{1'b0}} } ;
41
42 assign r0 = $signed(ONE) + $signed( x0 >>> 1 )  ;
43
44 // ----------------------------------------------------
45 // Second Iteration 2 Cycle  rn = r - (1/8) x ^ 2
46 // ----------------------------------------------------
47
48 fixpbinomialstepaux #( .NBITS (NBITS) , .Q(Q) )
49    c1 (
50    .clk(clk),
51    .enable(iter0),
52    .add(1'b0) ,
53    .xn(x0),
54    .xnacum(x0),
55    .dfactor(32'd3) ,
56    .oresult(r0) ,
57    .valid(iter1),
58    .xnacumn ( xnacum1),
59    .result  ( r1 )  ) ;
60
```

```
61 // ----------------------------------------------------
62 // Third Iteration 3 Cycle  rn = r + (1/16) x ^ 3
63 // ----------------------------------------------------
64
65 fixpbinomialstepaux #( .NBITS (NBITS) , .Q(Q) )
66    c2 (
67      .clk(clk),
68      .enable(iter1),
69      .add(1'b1),
70      .xn(x1),
71      .xnacum(xnacum1),
72      .dfactor(32'd4),
73      .oresult(r1) ,
74      .valid(iter2),
75      .xnacumn(xnacum2) ,
76      .result (r2) ) ;
77
78 // ----------------------------------------------------
79 // Four Iteration 3 Cycle   rn = r - (1/128) x ^ 4
80 // ----------------------------------------------------
81
82 wire [NBITS-1 :0 ] FIVE ;
83 assign FIVE = { {NBITS-Q-3{1'b0}}, {3'b101} , {Q{1'b0}} }
84
85 fixpbinomialstep #( .NBITS (NBITS) , .Q(Q) )
86    c3 (
87        .clk(clk),
88        .enable(iter2),
89        .add(1'b0) ,
90        .xn(x2),
91        .xnacum(xnacum2),
92        .mfactor( FIVE ),
93        .dfactor(32'd7) ,
94        .oresult(r2) ,
95        .valid(iter3),
96        .xnacumn(xnacum3),
97        .result (r3) ) ;
98
```

Review

- Good coding styles for FSM.
- FSMs implemented in two or three blocks (One sequential the others combinatorial)
- Introduced the idea of digital pipelines.
- Pipelines are make of combinatorial and sequential components.
- The divide and conquer approach at work.

Questions ...

Lab Notes

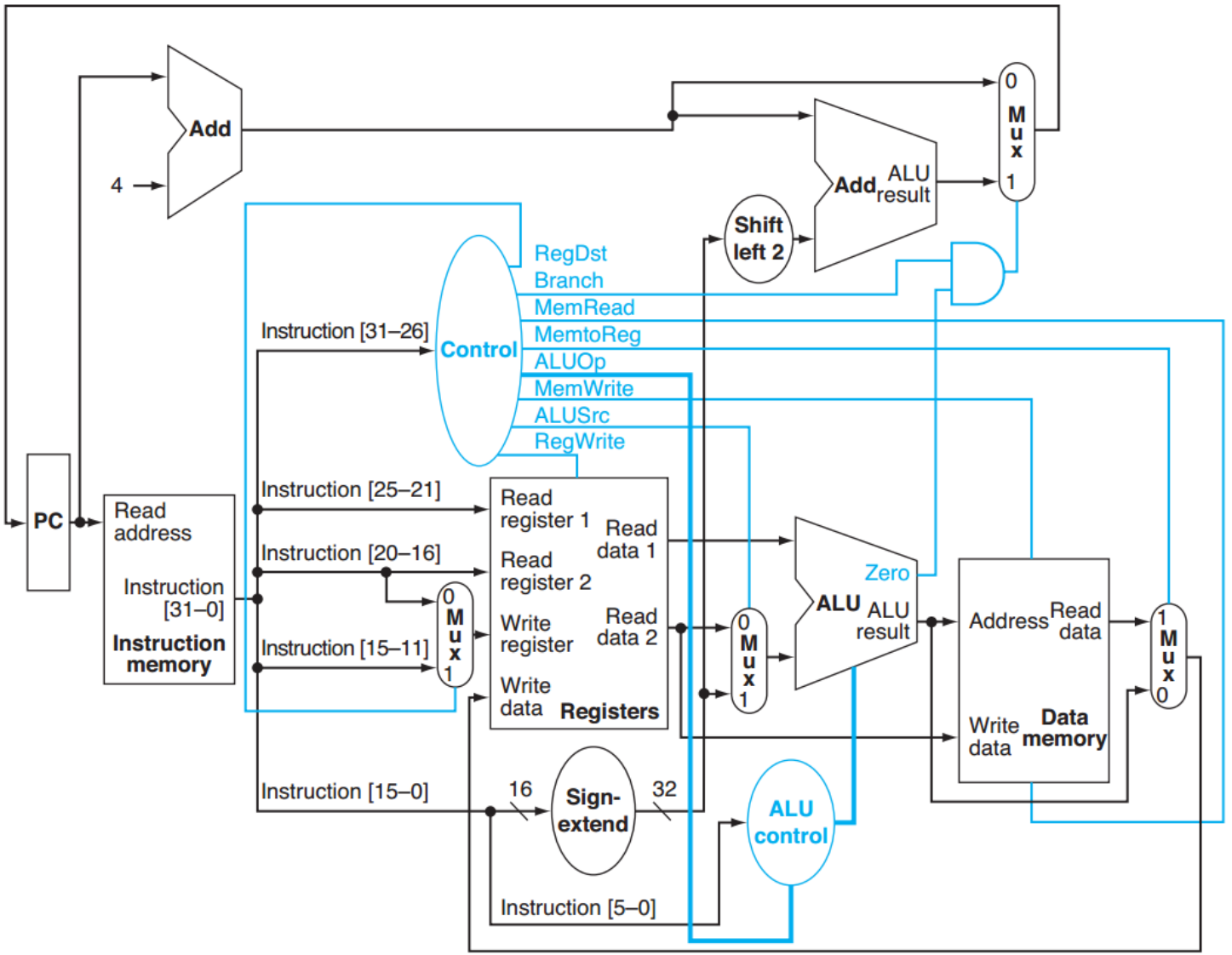

FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction.