Lecture 9 Numerical Hardware Design and Integration

Outline

9.1 Design Template for Numerical Hardware

9.2 Register Files

9.3 Design Example: FP Matrix-Matrix Adder

9.4 Design Example: FP AXPY Calculation

9.5 Design Example: Basic Design on FP DDOT

Outline

9.1 Design Template for Numerical Hardware

- A. Design Architecture
  - Datapath:
    - Two memory blocks, ``mem0'' and ``mem1'', are utilized to buffer the input and output data frames.
    - Data processing unit: constructed using multiple FP operations through parallel and pipeline structures and may result in a long data path.
  - FSM for the functions of timing control.
    - 1) Generates control signals; 2) Communicate with the datapath to trigger FP operations; and 3) Monitor the status of the datapath.

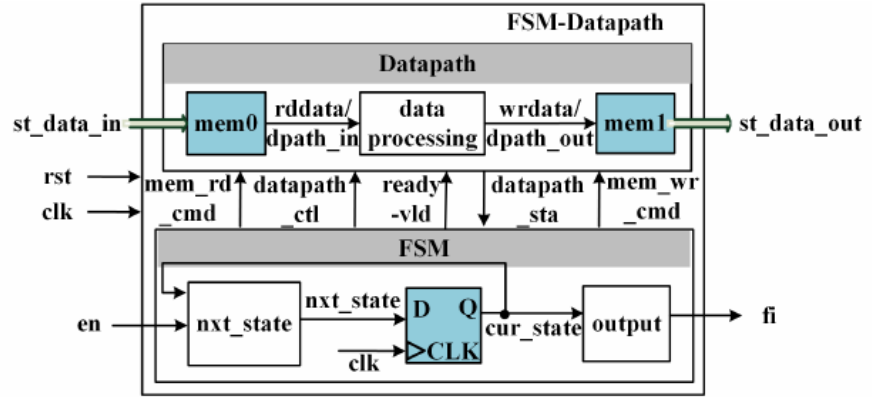


FIGURE 9.1
Design Structure of Numerical Hardware

9.1 Design Template for Numerical Hardware

- B. Timing Diagram
  - ``Mem0 Read & Datapath In'' :
    - Performs two tasks: reading data from ``mem0'' and inputting the same data into the data processing unit.
  - ``Mem1 Write & Datapath Out'':
    - Output data (with one cycle delay) and write the same data into ``mem1''.
  - The FSM controller
    - Generates and updates the read and write commands
    - Generates the indicator signals such as the data valid indicator ``vld'' and the data processing end indicator ``fi''

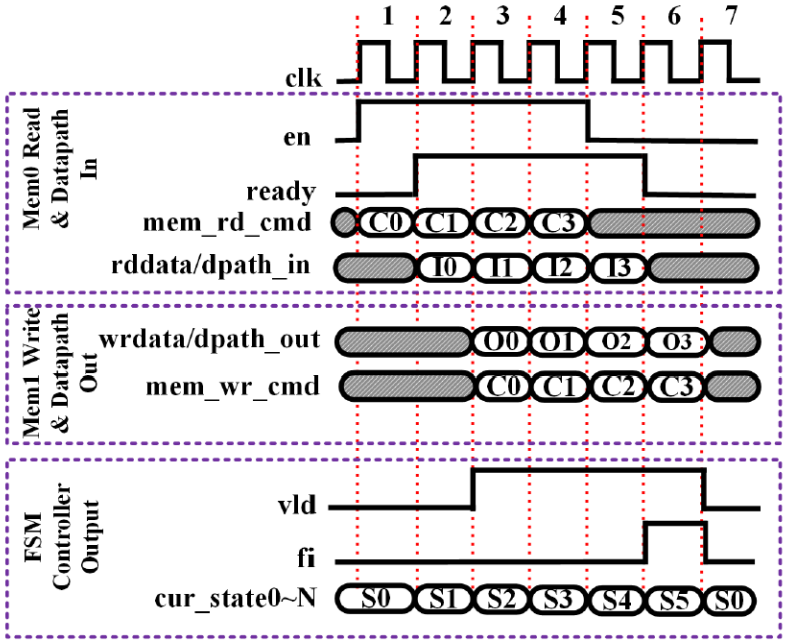


FIGURE 9.2
Timing Diagram of Numerical Hardware

Outline

- 9.2.1 RTL Design on Register Files
  - Custom register files hold a significant position in RTL design.
    - They function both as *simulation models* for random access memories (RAMs) and as *synthesizable implementations of register arrays*.
    - ASICs:
      - RF: High cost in terms of silicon real estate and are not intended for storing large data sets on ASICs.
      - Once a fabrication technology is established, it becomes feasible to replace large-sized register files with synthesizable RAM blocks.
    - FPGAs:
      - RF: Presents challenges because of constraints on available resources like flip-flops and registers within the FPGA device.
      - FPGA vendors offer pre-designed Block RAM or Ultra RAM IP cores, which can be integrated into SoC designs, simplifying the development process and ensuring efficient utilization of available FPGA resources.

- 9.2.2 Single-Port Register File
  - A. Design Specification
    - Memory write/read enable signal ``en'':
      - Enables the write (binary one) or read (binary zero) operation of the register array.
    - Write enable signal ``we'':
      - The corresponding data byte within the write data bus (multi-byte) is written into the register array.
    - Write/read address ``addr'':
      - The address at which the data is to be written or read.
    - Data input buses ``din''
      - The input data to be written into the register array.
    - Data output buses ``dout'':
      - The output data read from the specified address.

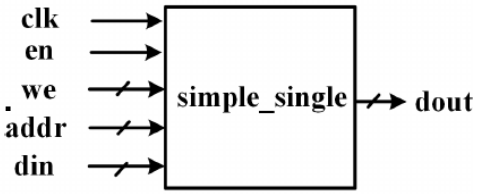


FIGURE 9.3
Block Diagram of Single-Port Register File

- 9.2.2 Single-Port Register File:
  - A. Design Specification
    - (a) Simultaneous writing of the entire data word into the designated register.
    - (b) The lower three bytes of the write data bus are writable into the register corresponding to the address.
    - (c) The lower two bytes of the write data bus.
    - (d) The least-significant byte can be written into the register at the corresponding address.

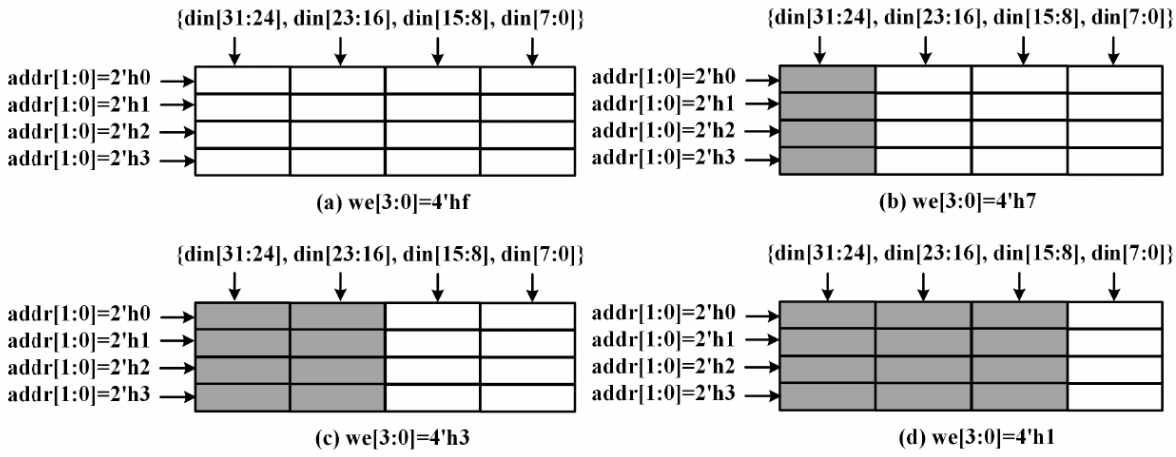
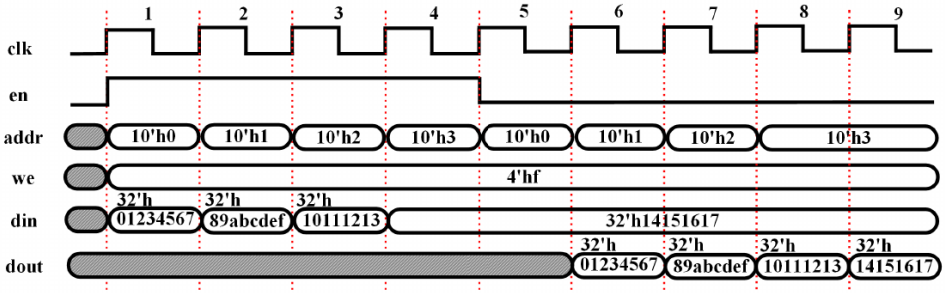
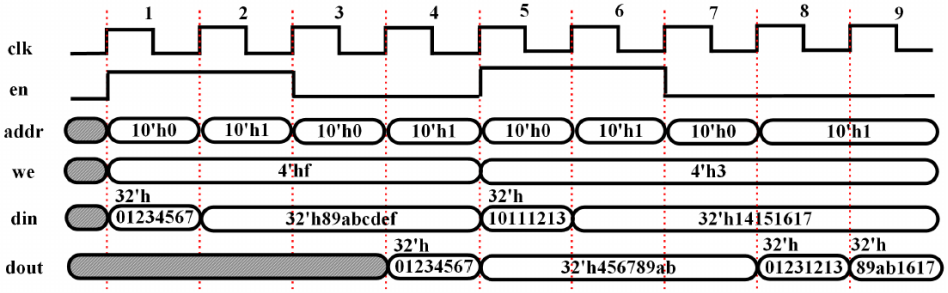


FIGURE 9.4
Register File Access Utilizing Write Enable Signal (Register Bytes in White Boxes Indicate Data Writing Enabling)

- 9.2.2 Single-Port Register File:
  - B. Timing Diagram
    - (a) Write and read the entire word
    - (b) Demonstrates write operations performed in words and half-words to showcase the difference.



(a) Timing Diagram of Register File Access with "we=4'hf"



(b) Timing Diagram of Register File Access with "we=4'hf" and "we=4'h3"

FIGURE 9.5
Timing Diagram of Register File Access Utilizing Write Enable Controlling

9.2 Register Files

- 9.2.2 Single-Port Register File:
  - C. Verilog Design

```
1  `define WIDTH 32
2  `define DEPTH 1024
3  `define ADDR_WIDTH 10
4  `define WE_WIDTH 4
5  module simple_single(input                          clk   ,
6                       input                          en    ,
7                       input       [`WE_WIDTH-1:0]    we    ,
8                       input       [`ADDR_WIDTH-1:0]  addr  ,
9                       input       [`WIDTH-1:0]       din   ,
10                      output reg  [`WIDTH-1:0]       dout);
11 reg [`WIDTH-1:0] ram[0:`DEPTH-1];
12
13 always @(posedge clk) begin
14   if (en) begin
15     if(we[`WE_WIDTH-1]) begin
16       ram[addr][`WIDTH-1:`WIDTH-8*1]<=
17                                 din[`WIDTH-1:`WIDTH-8*1];
18     end
19     if (we[`WE_WIDTH-2]) begin
20       ram[addr][`WIDTH-8*1-1:`WIDTH-8*2]<=
21                               din[`WIDTH-8*1-1:`WIDTH-8*2];
22     end
23     if (we[`WE_WIDTH-3]) begin
24       ram[addr][`WIDTH-8*2-1:`WIDTH-8*3]<=
25                               din[`WIDTH-8*2-1:`WIDTH-8*3];
26     end
27     if (we[`WE_WIDTH-4]) begin
28       ram[addr][`WIDTH-8*3-1:`WIDTH-8*4]<=
29                               din[`WIDTH-8*3-1:`WIDTH-8*4];
30     end
31   end else begin
32     dout <= ram[addr];
33   end
34 end
35 endmodule
```

- 9.2.3 Dual-Port Register File
  - A. Design Specification
    - Memory write enable signal ``ena'':
      - Enables the write operation of the register array.
    - Write enable signal ``wea'':
      - The corresponding data byte within the write data bus (multi-byte) is written into the register array.
    - Write address ``addra'':
      - The address at which the data is to be written.
    - Data input buses ``dina'':
      - The input data to be written into the register array.
    - Memory read enable signal ``enb'':
      - The read operation of the register array.
    - Write/read address ``addrb'':
      - The address from which the data is to be read.
    - Data output buses ``doutb'':
      - The output data read from the specified address.

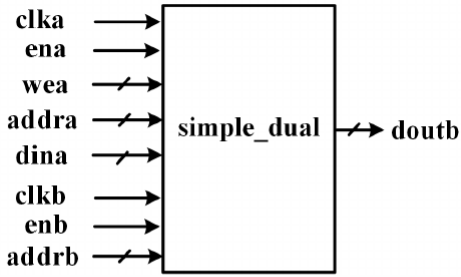


FIGURE 9.6
Block Diagram of Dual-Port Register File

- 9.2.3 Dual-Port Register File:
  - B. Timing Diagram
    - Different clocks: ``clka" and ``clkb" signals
    - Overlapping: The read operations can be performed in parallel with the ongoing write operations.
    - Read after Write: the read operation for address 2'h0 occurs two clock cycles after the write operation to the same address.
    - Address Collision Issues: it is crucial to highlight that simultaneous write and read operations *to the same address within the same clock cycle* are not permitted in a dual-port register array.

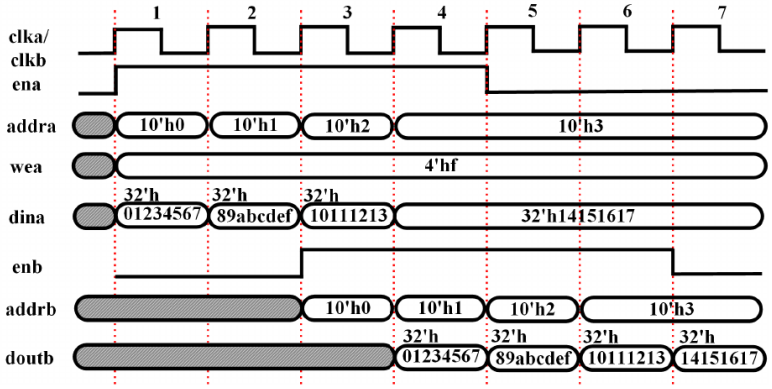


FIGURE 9.7
Timing Diagram of Dual-Port Register File Access

9.2 Register Files

- 9.2.3 Dual-Port Register File:
  - C. Verilog Design

```
1  `define WIDTH 32
2  `define DEPTH 1024
3  `define ADDR_WIDTH 10
4  `define WEA_WIDTH 4
5  module simple_dual(input                          clka ,
6                     input                          ena  ,
7                     input      [`WEA_WIDTH-1:0]    wea  ,
8                     input      [`ADDR_WIDTH-1:0]   addra,
9                     input      [`WIDTH-1:0]        dina ,
10                    input                          clkb ,
11                    input                          enb  ,
12                    input      [`ADDR_WIDTH-1:0]   addrb,
13                    output reg [`WIDTH-1:0]        doutb);
14 reg [`WIDTH-1:0] ram[0:`DEPTH-1];
15
16 always @(posedge clka) begin
17   if (ena) begin
18     if(wea[`WEA_WIDTH-1]) begin
19       ram[addra][`WIDTH-1:`WIDTH-8*1]<=
20                              dina[`WIDTH-1:`WIDTH-8*1];
21     end
22     if (wea[`WEA_WIDTH-2]) begin
23       ram[addra][`WIDTH-8*1-1:`WIDTH-8*2]<=
24                            dina[`WIDTH-8*1-1:`WIDTH-8*2];
25     end
26     if (wea[`WEA_WIDTH-3]) begin
27       ram[addra][`WIDTH-8*2-1:`WIDTH-8*3]<=
28                            dina[`WIDTH-8*2-1:`WIDTH-8*3];
29     end
30     if (wea[`WEA_WIDTH-4]) begin
31       ram[addra][`WIDTH-8*3-1:`WIDTH-8*4]<=
32                            dina[`WIDTH-8*3-1:`WIDTH-8*4];
33     end
34   end
35 end
36
37 always @(posedge clkb) begin
38   if (enb) begin
39       doutb <= ram[addrb];
40   end
41 end
42 endmodule
```

- 9.2.4 Ping-Pong Buffer
  - A. Design Specification
    - Comprise two single-port memories.
      - The control signals, namely ``u0_wr'' and ``u0_rd'', are responsible for overseeing write and read access to the ``u0_mem'' memory.
      - The signals ``u1_wr'' and ``u1_rd'' govern write and read access to the ``u1_mem'' memory.
      - When ``u0_rd'' is asserted, indicating active reading or data processing in ``u0_mem'', it facilitates concurrent execution of ``u1_wr'' to update data in ``u1_mem''.
      - When ``u1_rd'' is asserted, signaling a read operation and data processing in ``u1_mem'', it enables simultaneous data feeding into ``u0_mem'' with an asserted ``u0_wr''.

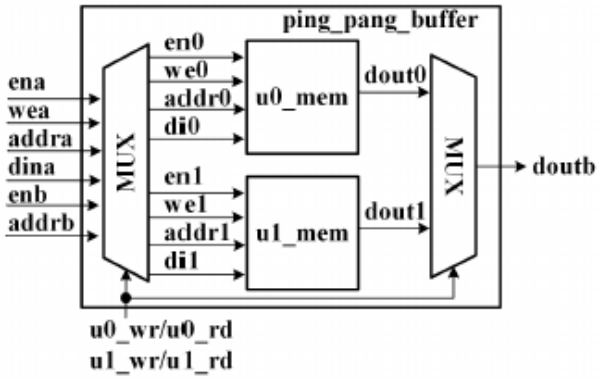


FIGURE 9.8
Block Diagram of Ping-Pong Buffer

```
1  `include "simple_single.v"
2  module ping_pong_buffer (input                       clk   ,
3                           input                       u0_wr ,
4                           input                       u1_wr ,
5                           input                       u0_rd ,
6                           input                       u1_rd ,
7                           input                       ena   ,
8                           input  [`WE_WIDTH-1:0]      wea   ,
9                           input  [`ADDR_WIDTH-1:0]    addra ,
10                          input  [`WIDTH-1:0]         dina  ,
11                          input                       enb   ,
12                          input  [`ADDR_WIDTH-1:0]    addrb ,
13                          output [`WIDTH-1:0]         doutb);
14 wire en0 = (u0_wr&ena) | ~(u0_rd&enb) ;
15 wire en1 = (u1_wr&ena) | ~(u1_rd&enb) ;
16 wire [`WE_WIDTH-1:0] we0 = u0_wr ? wea : `WEA4;
17 wire [`WE_WIDTH-1:0] we1 = u1_wr ? wea : `WEA4;
18
19 wire [`ADDR_WIDTH-1:0] addr0 = u0_wr ? addra :
20                                 u0_rd ? addrb : `ADDR_WIDTH'b0;
21 wire [`ADDR_WIDTH-1:0] addr1 = u1_wr ? addra :
22                                 u1_rd ? addrb : `ADDR_WIDTH'b0;
23 wire [`WIDTH-1:0]      din0 = u0_wr ? dina  : `WIDTH'b0    ;
24 wire [`WIDTH-1:0]      din1 = u1_wr ? dina  : `WIDTH'b0    ;
25
26 wire [`WIDTH-1:0]      dout0 ;
27 wire [`WIDTH-1:0]      dout1 ;
28 assign doutb = u0_rd ? dout0 :
29                       u1_rd ? dout1 : `ADDR_WIDTH'h0;
30
31 simple_single u0_mem (.clk (clk  ),
32                       .en  (en0  ),
33                       .we  (we0  ),
34                       .addr(addr0),
35                       .din (din0 ),
36                       .dout(dout0));
37 simple_single u1_mem (.clk (clk  ),
38                       .en  (en1  ),
39                       .we  (we1  ),
40                       .addr(addr1),
41                       .din (din1 ),
42                       .dout(dout1));
43 endmodule
```

9.2 Register Files

- 9.2.4 Ping-Pong Buffer
  - B. Verilog Design
    - The first-stage multiplexer (described in lines 14-24)
    - The second-stage multiplexer (Lines 26-29)

Outline

9.3 Design Example: FP Matrix-Matrix Adder

- Design Example: FP Matrix-Matrix Adder
  - The FP matrix addition can be performed with multiple FP adders in parallel.
    - The mathematical equation for a 4×4 matrix addition can be written as

$$\begin{pmatrix} z_{00} & z_{01} & z_{02} & z_{03} \\ z_{10} & z_{11} & z_{12} & z_{13} \\ z_{20} & z_{21} & z_{22} & z_{23} \\ z_{30} & z_{31} & z_{32} & z_{33} \end{pmatrix} = \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{pmatrix} + \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} \tag{9.1}$$

$$= \begin{pmatrix} x_{00}+y_{00} & x_{01}+y_{01} & x_{02}+y_{02} & x_{03}+y_{03} \\ x_{10}+y_{10} & x_{11}+y_{11} & x_{12}+y_{12} & x_{13}+y_{13} \\ x_{20}+y_{20} & x_{21}+y_{21} & x_{22}+y_{22} & x_{23}+y_{23} \\ x_{30}+y_{30} & x_{31}+y_{31} & x_{32}+y_{32} & x_{33}+y_{33} \end{pmatrix}$$

  - Three register arrays are designed to store the matrices.
    - As two arrays are used only for reading and one is used only for writing, single-port memory blocks are utilized.

9.3 Design Example: FP Matrix-Matrix Adder

- 9.3.1 Design Structure of FP Matrix-Matrix Adder
  - Datapath:
    - "mem0" and "mem1" and "mem2":
      - 128 bits data bus
      - The columns of each matrix are stored in memory row-wise
    - Parallel 4× 32-bit FP Adders
  - Timing controller
    - Reading four rows from memory;
    - Performing four rounds of 128-bit FP additions;
    - Writing the outcomes back into memory.

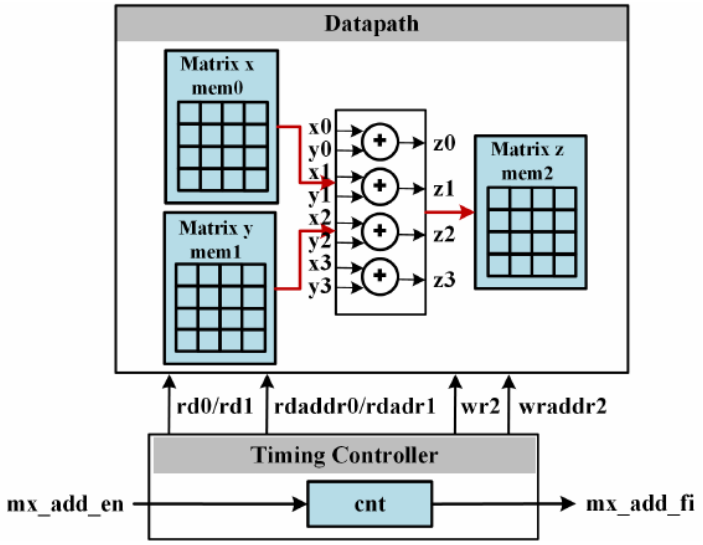


FIGURE 9.9
Design Structure of FP Matrix-Matrix Adder

9.3 Design Example: FP Matrix-Matrix Adder

- 9.3.2 Timing Diagram of FP Matrix-Matrix Adder
  - A. Assumption for Test Cases:
    - The four word-sized data in ``rddata0[127:96]'', ``rddata0[95:64]'', ``rddata0[63:32]'', and ``rddata0[31:0]'' are all identical, as are the four word-sized data in ``rddata1[127:96]'', ``rddata1[95:64]'', ``rddata1[63:32]'', and ``rddata1[31:0]''. Consequently, the data on the ``x0-x3'' buses for all four adders are identical, and the data on the ``y0-y3'' buses are also identical.
    - This results in the summations ``z0-z3'' being the same as well. Therefore, all four word-sized data on ``wrdata2[127:96]'', ``wrdata2[95:64]'', ``wrdata2[63:32]'', and ``wrdata2[31:0]'' will also be identical.
    - By making this assumption, we can simplify the timing diagram to display only one set of data buses: ``rddata0[127:96]/x0'', ``rddata1[127:96]/y0'', and ``wrdata2[127:96]/z0''.

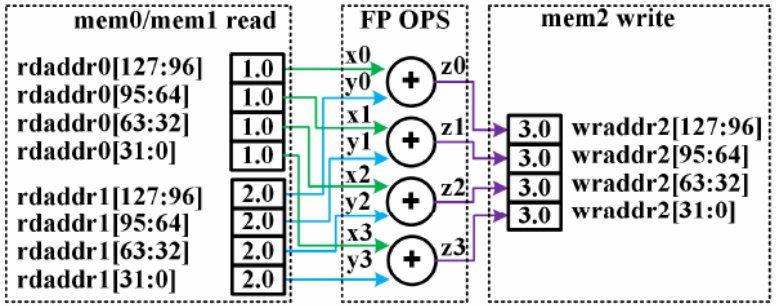


FIGURE 9.10
Data Processing Example of FP Matrix-Matrix Adder

9.3 Design Example: FP Matrix-Matrix Adder

- 9.3.2 Timing Diagram of FP Matrix-Matrix Adder
  - B. Timing Diagram of FP Matrix-Matrix Adder
    - Mem0&1 Read:
      - Four rows are read from ``mem0" and ``mem1" over four clock cycles.
    - Addition & Mem2 Write
      - In cycles 3-6, The summations are written in ``mem2''.
    - FSM Controller
      - In the last cycle, sets the ``mx_add_fi" indicator to signal the completion of the matrix addition.

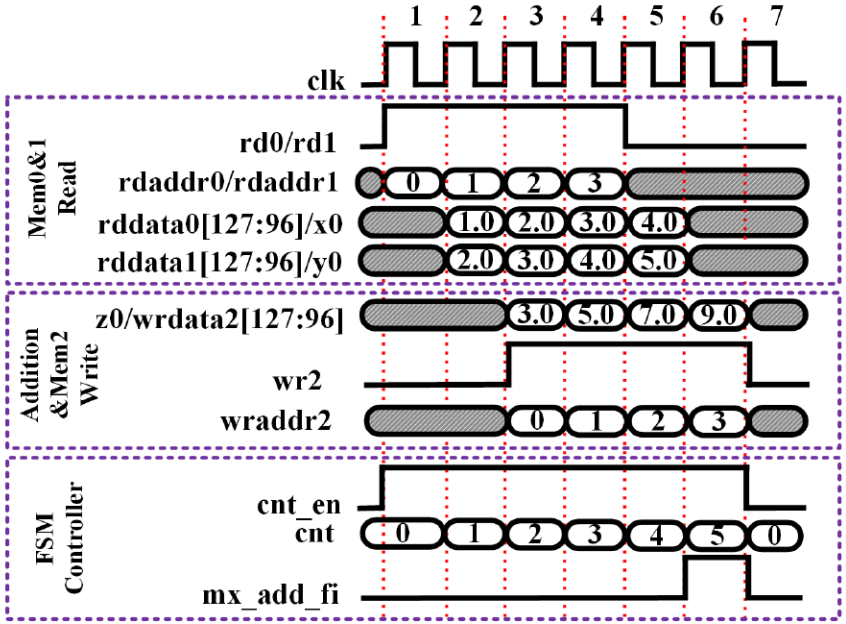
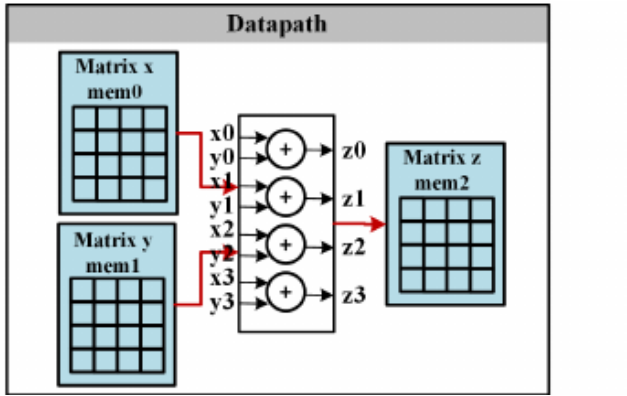


FIGURE 9.11
Timing Diagram of FP Matrix-Matrix Adder

9.3 Design Example: FP Matrix-Matrix Adder

- 9.4.3 Verilog Code for FP Matrix-Matrix Adder
  - Lines 15-38 contain the instantiation of four 32-bit FP adders.

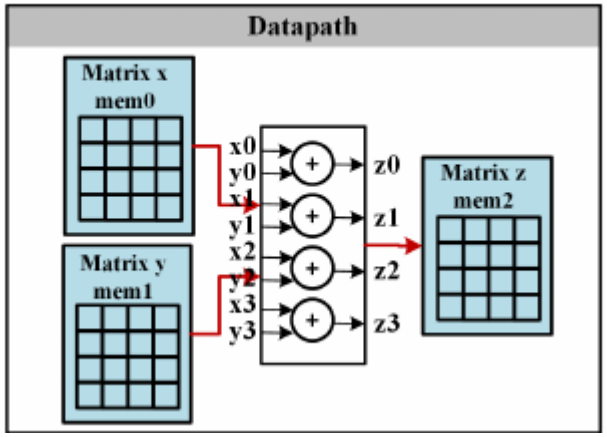
```
1  module mx_add (input          clk        ,
2                 input          rst        ,
3                 input          mx_add_en  ,
4                 output         mx_add_fi  );
5  //*********************************
6  //****    Datapath Design *********
7  //*********************************
8  wire [31:0] x0, x1, x2, x3;
9  wire [31:0] y0, y1, y2, y3;
10 wire [31:0] z0, z1, z2, z3;
11
12 wire        wr2, rd0, rd1;
13 reg  [1:0]  wraddr2, rdaddr0, rdaddr1;
```




```
14
15 // FP adders instantiation
16 FP_adder u0_fp_add(.clock   (clk  ),
17                    .reset   (rst  ),
18                    .io_in_a (x0   ),
19                    .io_in_b (y0   ),
20                    .io_out_s(z0   ));
21
22 FP_adder u1_fp_add(.clock   (clk  ),
23                    .reset   (rst  ),
24                    .io_in_a (x1   ),
25                    .io_in_b (y1   ),
26                    .io_out_s(z1   ));
27
28 FP_adder u2_fp_add(.clock   (clk  ),
29                    .reset   (rst  ),
30                    .io_in_a (x2   ),
31                    .io_in_b (y2   ),
32                    .io_out_s(z2   ));
33
34 FP_adder u3_fp_add(.clock   (clk  ),
35                    .reset   (rst  ),
36                    .io_in_a (x3   ),
37                    .io_in_b (y3   ),
38                    .io_out_s(z3   ));
```

9.3 Design Example: FP Matrix-Matrix Adder

- 9.4.3 Verilog Code for FP Matrix-Matrix Adder
  - In lines 40-60, three memory blocks are used to store the matrices.
  - Each 128-bit data read from ``u0/1_mem'' is divided into four words which serve as inputs ``x0-x3'' and ``y0-y3'' for the four FP adders.
  - The outputs of the four FP adders, ``z0-z3'', are combined to form the 128-bit input for ``u2_mem2''.

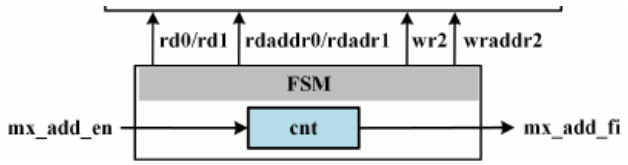



```
40  // memory instantiation
41  simple_single u0_mem(.clk  (clk          ) ,
42                       .en   (rd0          ) ,
43                       .we   (`WEA4        ) ,
44                       .addr (rdaddr0      ) ,
45                       .di   (128'h0       ) ,
46                       .dout ({x0,x1,x2,x3}));
47
48  simple_single u1_mem(.clk  (clk          ) ,
49                       .en   (rd1          ) ,
50                       .we   (`WEA4        ) ,
51                       .addr (rdaddr1      ) ,
52                       .di   (128'h0       ) ,
53                       .dout ({y0,y1,y2,y3}));
54
55  simple_single u2_mem(.clk  (clk          ) ,
56                       .en   (wr2          ) ,
57                       .we   (`WEA4        ) ,
58                       .addr (wraddr2      ) ,
59                       .di   ({z0,z1,z2,z3}) ,
60                       .dout (             ));
```

9.3 Design Example: FP Matrix-Matrix Adder

- 9.4.3 Verilog Code for FP Matrix-Matrix Adder
  - The timing controller design is illustrated in lines 62-101.
  - Memory read and write operations are explained in lines 77-99.

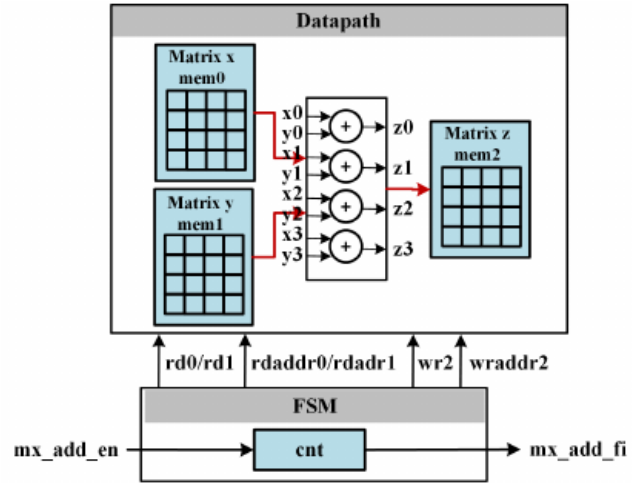
```
62  //*************************************
63  //****     FSM Controller Design *********
64  //*************************************
65  reg  [2:0]  cnt;
66  wire cnt_en = mx_add_en | |cnt;
67  wire [2:0] nxt_cnt=(cnt==3'd5) ? 3'd0 :
68                                   cnt_en ? (cnt+3'd1) : cnt;
69  always @(posedge clk) begin
70    if(rst) begin
71      cnt<=3'd0    ;
72    end else begin
73      cnt<=nxt_cnt;
74    end
75  end
76
77  assign rd0 = mx_add_en ;
78  assign rd1 = mx_add_en ;
79  wire [1:0] nxt_rdaddr0 = rd0 ? rdaddr0+2'h1 : rdaddr0;
80  wire [1:0] nxt_rdaddr1 = rd1 ? rdaddr1+2'h1 : rdaddr1;
81  always @(posedge clk) begin
82    if(rst) begin
83      rdaddr0<=2'h0    ;
84      rdaddr1<=2'h0    ;
85    end else begin
86      rdaddr0<=nxt_rdaddr0 ;
87      rdaddr1<=nxt_rdaddr1 ;
88    end
89  end
```



(a) Design Structure of Matrix Adder

9.3 Design Example: FP Matrix-Matrix Adder

- 9.4.3 Verilog Code for FP Matrix-Matrix Adder
  - The completion of the matrix addition is indicated by the ``mx_add_fi'' output in line 101.



(a) Design Structure of Matrix Adder

```
91  assign wr2 = cnt>=3'd2 & cnt<=3'd5 ;
92  wire [1:0] nxt_wraddr2 = wr2 ? wraddr2+2'h1 : 2'h0;
93  always @(posedge clk) begin
94    if(rst) begin
95      wraddr2<=2'h0          ;
96    end else begin
97      wraddr2<=nxt_wraddr2 ;
98    end
99  end
100
101 assign mx_add_fi = &wraddr2;
102 endmodule
```

Outline

9.4 Design Example: FP AXPY Calculation

- Design Example: FP AXPY Calculation
  - The mathematical expression *y* = *a* * *x* + *y* is described in hardware utilizing multiple single-precision FP adders and multipliers.
    - In this equation, *x* and *y* are 4×4 FP matrices, while a is a FP constant.

$$\begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} = a \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{pmatrix} + \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} \tag{9.2}$$

$$= \begin{pmatrix} ax_{00} + y_{00} & ax_{01} + y_{01} & ax_{02} + y_{02} & ax_{03} + y_{03} \\ ax_{10} + y_{10} & ax_{11} + y_{11} & ax_{12} + y_{12} & ax_{13} + y_{13} \\ ax_{20} + y_{20} & ax_{21} + y_{21} & ax_{22} + y_{22} & ax_{23} + y_{23} \\ ax_{30} + y_{30} & ax_{31} + y_{31} & ax_{32} + y_{32} & ax_{33} + y_{33} \end{pmatrix}$$

  - Since the output matrix is still *y*, the results need to be written back to the same memory, necessitating the use of a dual-port memory block.

9.4 Design Example: FP AXPY Calculation

- 9.4.1 Design Structure of FP AXPY Calculation
  - Datapath:
    - ``mem0" and ``mem1'':
      - 128 bits data bus
      - The columns of each matrix are stored in memory row-wise
    - Parallel 32-bit FP Multipliers following with Adders
  - Timing controller
    - Reading four rows from memory;
    - Performing four rounds of 128-bit FP multiplications and additions;
    - Writing the outcomes back into memory

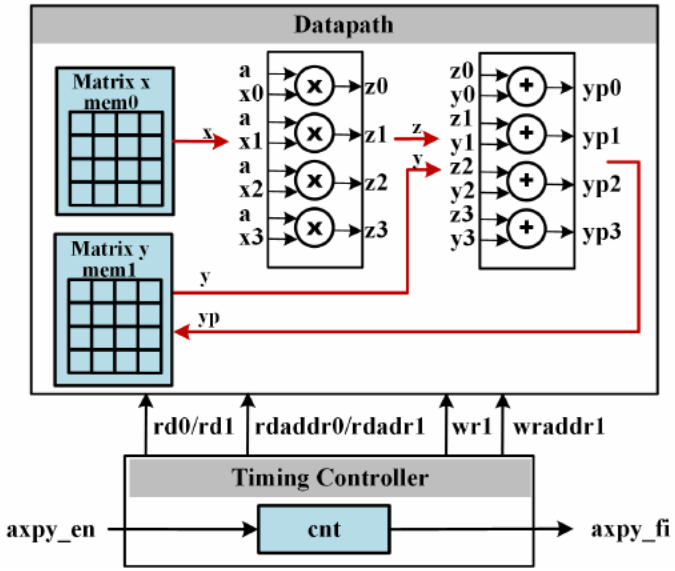


FIGURE 9.12
Design Structure of FP AXPY

9.4 Design Example: FP AXPY Calculation

- 9.4.2 Timing Diagram of FP AXPY Calculation
  - Mem0&1 Read:
    - Four rows are read from ``mem0" and ``mem1" over four clock cycles.
  - AXPY & Mem1 Write
    - In cycles 4-7, The summations are written in ``mem2".
  - FSM Controller
    - In the last cycle, sets the ``axpi_fi" indicator to signal the completion of the axpy.

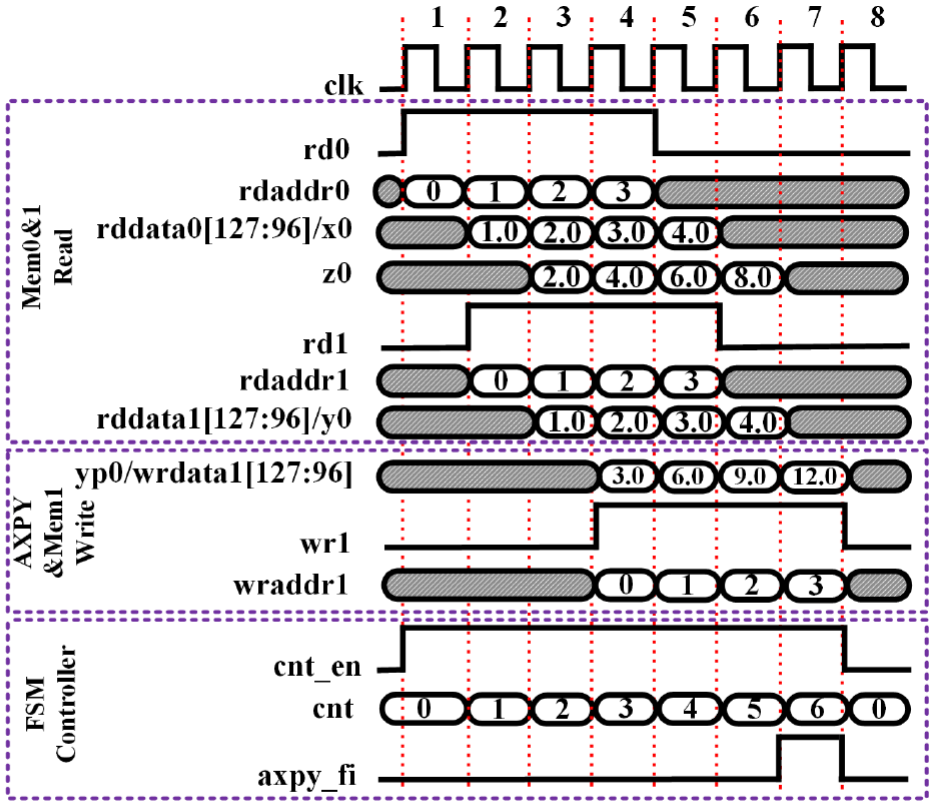
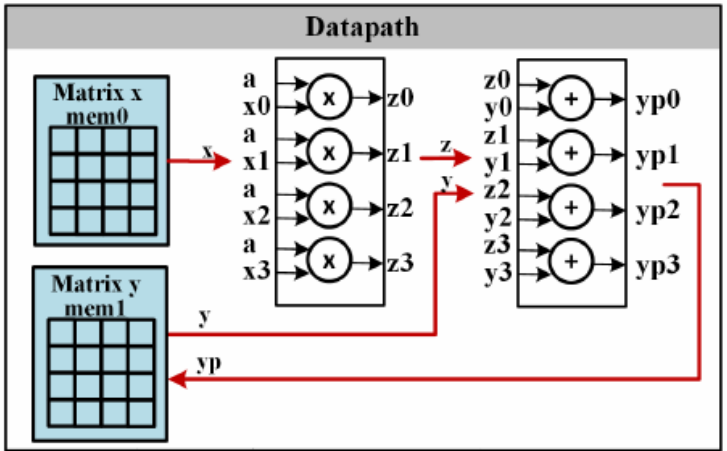


FIGURE 9.13
Timing Diagram of FP AXPY

9.4 Design Example: FP AXPY Calculation

- 9.4.3 Verilog Code for FP AXPY Calculation
  - In line 8, the FP constant ``a'' is assigned as 32'h3f800000 (FP 1.0).
  - Lines 17-40 instantiate four parallel 32-bit FP multipliers

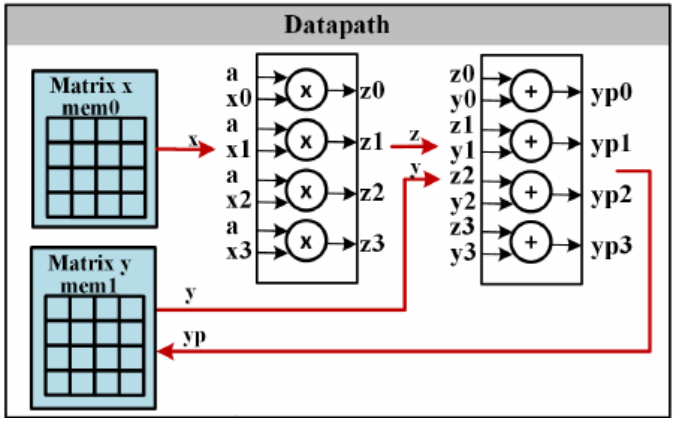
```
1  module axpy (input          clk      ,
2               input          rst      ,
3               input          axpy_en  ,
4               output         axpy_fi  );
5  //**********************************
6  //****     Datapath Design *********
7  //**********************************
8  wire [31:0] a=32'h40000000; //FP 2.0
9  wire [31:0] x0, x1, x2, x3;
10 wire [31:0] y0, y1, y2, y3;
11 wire [31:0] z0, z1, z2, z3;
12 wire [31:0] yp0, yp1, yp2, yp3;
13
14 wire         wr1, rd0, rd1;
15 reg  [1:0]   wraddr1, rdaddr0, rdaddr1;
```




```
17 // FP multipliers instantiation
18 FP_multiplier u0_fp_mul(.clock    (clk  ),
19                         .reset    (rst  ),
20                         .io_in_a  (a    ),
21                         .io_in_b  (x0   ),
22                         .io_out_s(z0   ));
23
24 FP_multiplier u1_fp_mul(.clock    (clk  ),
25                         .reset    (rst  ),
26                         .io_in_a  (a    ),
27                         .io_in_b  (x1   ),
28                         .io_out_s(z1   ));
29
30 FP_multiplier u2_fp_mul(.clock    (clk  ),
31                         .reset    (rst  ),
32                         .io_in_a  (a    ),
33                         .io_in_b  (x2   ),
34                         .io_out_s(z2   ));
35
36 FP_multiplier u3_fp_mul(.clock    (clk  ),
37                         .reset    (rst  ),
38                         .io_in_a  (a    ),
39                         .io_in_b  (x3   ),
40                         .io_out_s(z3   ));
```

9.4 Design Example: FP AXPY Calculation

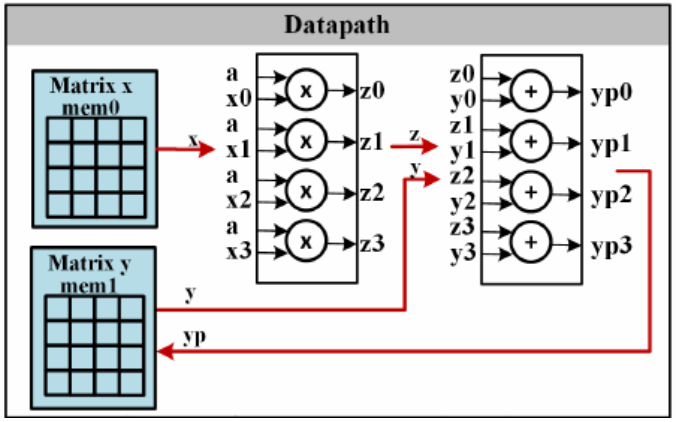
- 9.4.3 Verilog Code for FP AXPY Calculation
  - Lines 42-65 instantiate four parallel 32-bit FP adders.
    - The outputs ``z0-z3'' of the FP multipliers are connected to the inputs of the four 32-bit FP adders, while the other inputs ``y0-y3'' of the adders are read from ``mem1''.
    - The outputs of the adders are sent out on the buses ``yp0-yp3''.




```
42  // FP adders instantiation
43  FP_adder u0_fp_add(.clock   (clk  ),
44                     .reset   (rst  ),
45                     .io_in_a (z0   ),
46                     .io_in_b (y0   ),
47                     .io_out_s(yp0  ));
48
49  FP_adder u1_fp_add(.clock   (clk  ),
50                     .reset   (rst  ),
51                     .io_in_a (z1   ),
52                     .io_in_b (y1   ),
53                     .io_out_s(yp1  ));
54
55  FP_adder u2_fp_add(.clock   (clk  ),
56                     .reset   (rst  ),
57                     .io_in_a (z2   ),
58                     .io_in_b (y2   ),
59                     .io_out_s(yp2  ));
60
61  FP_adder u3_fp_add(.clock   (clk  ),
62                     .reset   (rst  ),
63                     .io_in_a (z3   ),
64                     .io_in_b (y3   ),
65                     .io_out_s(yp3  ));
```

9.4 Design Example: FP AXPY Calculation

- 9.4.3 Verilog Code for FP AXPY Calculation
  - Lines 67-86 show the instantiation of two dual-port memories, ``u0_mem'' for matrix *x* and ``u1_mem'' for matrix *y*.
  - The memory's IO port ``a'' is for write and port ``b'' is for read operations.
    - ``u0_mem'' is only used for read operations, so the ``a'' port is disabled by assigning the enable signal ``ena'' as binary 1'b0.

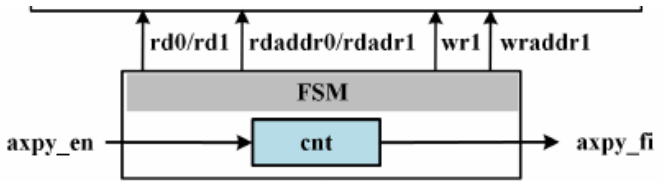



```
67  // memory instantiation
68  simple_dual u0_mem(.clka (clk           ) ,
69                     .ena  (1'b0          ) ,
70                     .wea  (`WEA4         ) ,
71                     .addra(2'h0          ) ,
72                     .dia  (128'h0        ) ,
73                     .clkb (clk           ) ,
74                     .enb  (rd0           ) ,
75                     .addrb(rdaddr0       ) ,
76                     .dob  ({x0,x1,x2,x3}));
77
78  simple_dual u1_mem(.clka (clk               ) ,
79                     .ena  (wr1               ) ,
80                     .wea  (`WEA4             ) ,
81                     .addra(wraddr1           ) ,
82                     .dia  ({yp0,yp1,yp2,yp3}) ,
83                     .clkb (clk               ) ,
84                     .enb  (rd1               ) ,
85                     .addrb(rdaddr1           ) ,
86                     .dob  ({y0,y1,y2,y3}     ));
```

9.4 Design Example: FP AXPY Calculation

- 9.4.3 Verilog Code for FP AXPY Calculation
  - Lines 88-127 describe the timing controller with a simple counter design.
    - Line 92: counter enable:
      - The counter enable signal ``cnt_en'' is activated as soon as the circuit enable ``axpy_en'' is turned on and
      - if it's non-zero using a Reduction OR operation (denoted as |cnt).
    - Line 93: nx_cnt design
      - Uses a Reduction AND operation (represented as &cnt[2:1]) to determine whether the two most significant bits are both binary ones.

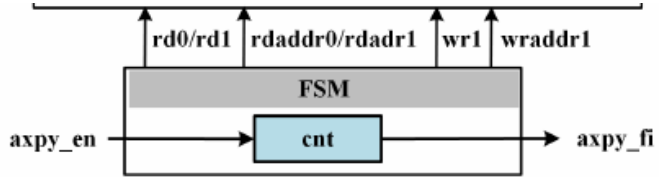
```
88  //*************************************
89  //****     FSM Controller Design *********
90  //*************************************
91  reg [2:0] cnt;
92  wire cnt_en = axpy_en | |cnt ;
93  wire [2:0] nxt_cnt = &cnt[2:1] ? 3'd0 :
94                                   cnt_en ? (cnt+3'd1) : cnt;
95  always @(posedge clk) begin
96    if(rst) begin
97      cnt<=3'd0     ;
98    end else begin
99      cnt<=nxt_cnt ;
100   end
101 end
```



(a) Design Structure of axpy

9.4 Design Example: FP AXPY Calculation

- 9.4.3 Verilog Code for FP AXPY Calculation
  - in line 127, the ``axpy_fi'' indicator is asserted when the write address reaches its maximum value of decimal 2'b11,
    - can be determined with a Reduction AND expression as &wraddr1.



(a) Design Structure of axpy

```
103  assign rd0 = axpy_en ;
104  assign rd1 = cnt>=3'd1 & cnt<=3'd4;
105  wire [1:0] nxt_rdaddr0 = rd0 ? rdaddr0+2'h1 : 2'h0;
106  wire [1:0] nxt_rdaddr1 = rd1 ? rdaddr1+2'h1 : 2'h0;
107  always @(posedge clk) begin
108    if(rst) begin
109      rdaddr0<=2'h0    ;
110      rdaddr1<=2'h0    ;
111    end else begin
112      rdaddr0<=nxt_rdaddr0 ;
113      rdaddr1<=nxt_rdaddr1 ;
114    end
115  end
116
117  assign wr1 = cnt>=3'd3 & cnt<=3'd6;
118  wire [1:0] nxt_wraddr1 = wr1 ? wraddr1+2'h1 : 2'h0;
119  always @(posedge clk) begin
120    if(rst) begin
121      wraddr1<=2'h0        ;
122    end else begin
123      wraddr1<=nxt_wraddr1 ;
124    end
125  end
126
127  assign axpy_fi = &wraddr1;
128  endmodule
```

Outline

- Design Example: Basic Design on FP DDOT
  - Mathematical *ddot* calculation performs a vector multiplication-addition of $z = x^T * y$ where *x*, *y* are FP vectors and *z* is the FP ddot summation.

$$z = \begin{pmatrix} x0 & x1 & x2 & x3 & x4 & x5 & x6 & x7 \end{pmatrix} * \begin{pmatrix} y0 \\ y1 \\ y2 \\ y3 \\ y4 \\ y5 \\ y6 \\ y7 \end{pmatrix} \tag{9.3}$$

$$= x0 \cdot y0 + x1 \cdot y1 + x2 \cdot y2 + x3 \cdot y3 + x4 \cdot y4 + x5 \cdot y5 + x6 \cdot y6 + x7 \cdot y7$$

  - An example implementation would use eight parallel FP multipliers for the eight products and seven cascading FP adders to sum them up as the final result z.

- 9.5.1 Design Structure of Basic Design on FP DDOT
  - Datapath:
    - Eight parallel FP multipliers in the first stage
    - In the subsequent stages, four FP adders in the second stage, two FP adders in the third stage, and finally one FP adder in the last stage are employed to sum up the eight products to generate the final result ``z'' in four clock cycles.
  - Timing controller
    - Control and monitor the datapath operations

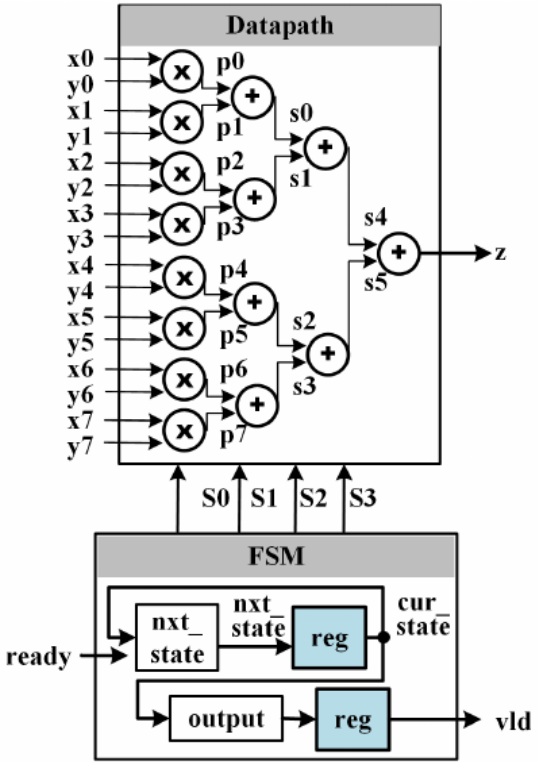


FIGURE 9.14
Design Structure of Basic Design on FP DDOT

- 9.5.1 Design Structure of Basic Design on FP DDOT
  - Latency:
    - $1 + \log_2 N$, where N represents the streaming width of the design engine
    - FSM: ``S0" for the first stage of parallel multiplications and ``S1-S3" for the three cascading stages of additions.
  - ``ready-vld" handshake
    - The input ``ready" indicates that the data on the ``x0-x7" and ``y0-y7" buses are ready to enter the ddot datapath,
    - The output ``vld" indicates that the data on the ``z" output is valid.

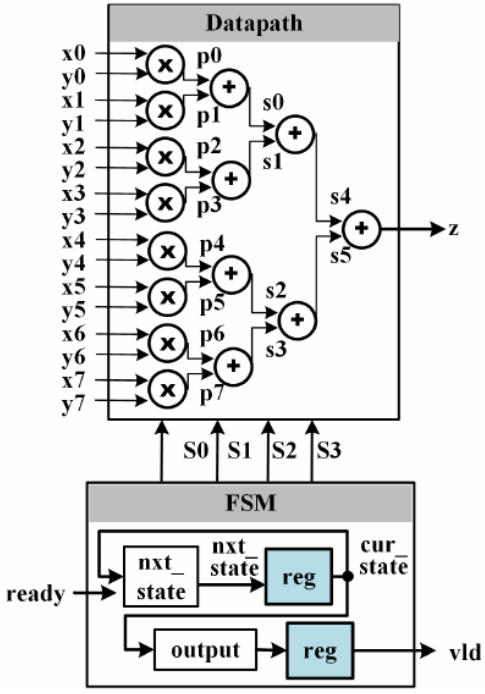


FIGURE 9.14
Design Structure of Basic Design on FP DDOT

- 9.5.2 Timing Diagram of Basic Design on FP DDOT Calculation
  - Multiplications
    - 1st cycle, the state is ``S0'' which activates the eight parallel multipliers.
  - Additions
    - 2nd cycle, the four FP adders are activated, producing the summations ``s0-s3'' in the following cycle.
    - The summations ``s4-s5'' are produced in the third cycle, followed by the final result ``z=FP 16.0'' in the fifth cycle, when the ``vld'' output is asserted.

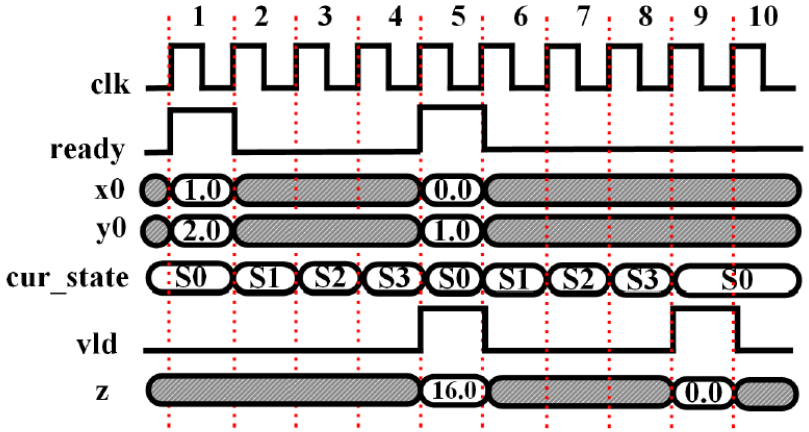
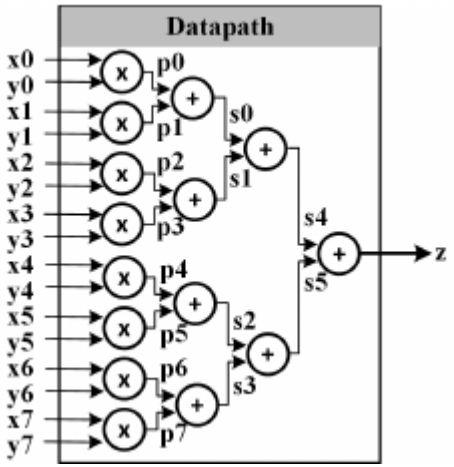


FIGURE 9.15
Timing Diagram of Basic Design on FP DDOT

- 9.5.3 Verilog Code for Basic Design on FP DDOT
  - The design instantiates eight FP multipliers in parallel in lines 14-61, with the products of each FP multiplier being connected to ``p0-p7".

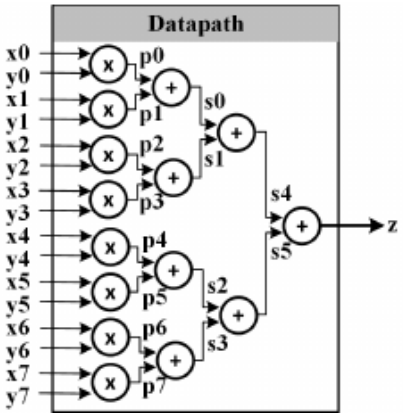
```
1  module basic_ddot (input          clk   ,
2                     input          rst   ,
3                     input          ready ,
4                     input  [31:0]  x0,x1,x2,x3,x4,x5,x6,x7,
5                     input  [31:0]  y0,y1,y2,y3,y4,y5,y6,y7,
6                     output reg     vld   ,
7                     output [31:0]  z     );
8  //*********************************
9  //****     Datapath Design *********
10 //*********************************
11 wire [31:0] p0, p1, p2, p3, p4, p5, p6, p7;
12 wire [31:0] s0, s1, s2, s3, s4, s5;
```




```
14 // The first stage of multipliers
15 FP_multiplier u0_fp_mul(.clock    (clk  ),
16                         .reset    (rst  ),
17                         .io_in_a  (x0   ),
18                         .io_in_b  (y0   ),
19                         .io_out_s (p0   ));
20
21 FP_multiplier u1_fp_mul(.clock    (clk  ),
22                         .reset    (rst  ),
23                         .io_in_a  (x1   ),
24                         .io_in_b  (y1   ),
25                         .io_out_s (p1   ));
26
27 FP_multiplier u2_fp_mul(.clock    (clk  ),
28                         .reset    (rst  ),
29                         .io_in_a  (x2   ),
30                         .io_in_b  (y2   ),
31                         .io_out_s (p2   ));
32
33 FP_multiplier u3_fp_mul(.clock    (clk  ),
34                         .reset    (rst  ),
35                         .io_in_a  (x3   ),
36                         .io_in_b  (y3   ),
37                         .io_out_s (p3   ));
```

9.5 Design Example: Basic Design on FP DDOT

```
39 FP_multiplier u4_fp_mul(.clock    (clk  ),
40                         .reset    (rst  ),
41                         .io_in_a  (x4   ),
42                         .io_in_b  (y4   ),
43                         .io_out_s(p4   ));
44
45 FP_multiplier u5_fp_mul(.clock    (clk  ),
46                         .reset    (rst  ),
47                         .io_in_a  (x5   ),
48                         .io_in_b  (y5   ),
49                         .io_out_s(p5   ));
50
51 FP_multiplier u6_fp_mul(.clock    (clk  ),
52                         .reset    (rst  ),
53                         .io_in_a  (x6   ),
54                         .io_in_b  (y6   ),
55                         .io_out_s(p6   ));
56
57 FP_multiplier u7_fp_mul(.clock    (clk  ),
58                         .reset    (rst  ),
59                         .io_in_a  (x7   ),
60                         .io_in_b  (y7   ),
61                         .io_out_s(p7   ));
```



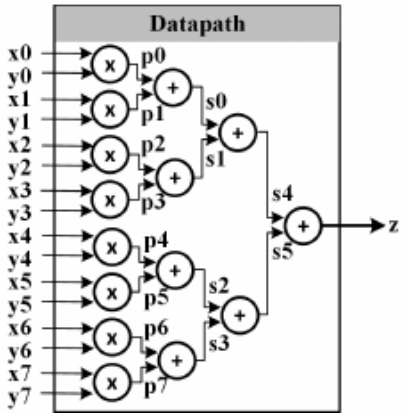
- 9.5.3 Verilog Code for Basic Design on FP DDOT
  - These products are then fed into the second-stage FP adders in lines 63-86, with the summation outputs being connected to 32-bit buses ``s0-s3".

```
63 // The second stage of adders
64 FP_adder u0_fp_add(.clock    (clk  ),
65                     .reset    (rst  ),
66                     .io_in_a  (p0   ),
67                     .io_in_b  (p1   ),
68                     .io_out_s(s0   ));
69
70 FP_adder u1_fp_add(.clock    (clk  ),
71                     .reset    (rst  ),
72                     .io_in_a  (p2   ),
73                     .io_in_b  (p3   ),
74                     .io_out_s(s1   ));
75
76 FP_adder u2_fp_add(.clock    (clk  ),
77                     .reset    (rst  ),
78                     .io_in_a  (p4   ),
79                     .io_in_b  (p5   ),
80                     .io_out_s(s2   ));
81
82 FP_adder u3_fp_add(.clock    (clk  ),
83                     .reset    (rst  ),
84                     .io_in_a  (p6   ),
85                     .io_in_b  (p7   ),
86                     .io_out_s(s3   ));
```

9.5 Design Example: Basic Design on FP DDOT

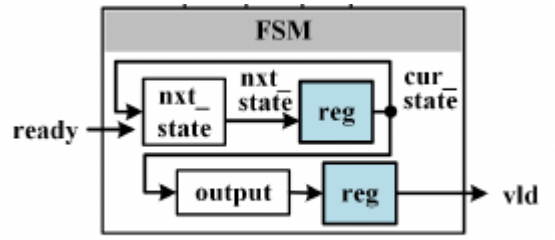
```
88  // The third stage of adders
89  FP_adder u4_fp_add(.clock    (clk  ),
90                     .reset    (rst  ),
91                     .io_in_a  (s0   ),
92                     .io_in_b  (s1   ),
93                     .io_out_s(s4   ));
94
95  FP_adder u5_fp_add(.clock    (clk  ),
96                     .reset    (rst  ),
97                     .io_in_a  (s2   ),
98                     .io_in_b  (s3   ),
99                     .io_out_s(s5   ));
100
101 // The fourth stage of adders
102 FP_adder u6_fp_add(.clock    (clk  ),
103                    .reset    (rst  ),
104                    .io_in_a  (s4   ),
105                    .io_in_b  (s5   ),
106                    .io_out_s(z    ));
```

- 9.5.3 Verilog Code for Basic Design on FP DDOT
  - The third-stage FP adders are instantiated in lines 88-99, taking ``s0-s3'' as inputs and providing 32-bit outputs connected to buses ``s4'' and ``s5''.
  - The final stage is a single FP adder instantiated in lines 101-106, which sums up ``s4'' and ``s5'' to produce the final output ``z''.



- 9.5.3 Verilog Code for Basic Design on FP DDOT
  - The valid output ``z'' is pushed out after the final state ``S3''. To assert the indicator output ``vld'', one more clock cycle is added.
  - In lines 136-143, a register is created to describe the ``vld'' output.

```
136  wire nxt_vld = cur_state==S3;
137  always @(posedge clk) begin
138    if(rst) begin
139      vld <= 1'b0    ;
140    end else begin
141      vld <= nxt_vld;
142    end
143  end
144  endmodule
```



(a) Basic Design Structure of Ddot

```
108  //****************************************
109  //****    FSM Controller Design *********
110  //****************************************
111  parameter S0 = 2'h0 ;
112  parameter S1 = 2'h1 ;
113  parameter S2 = 2'h2 ;
114  parameter S3 = 2'h3 ;
115
116  reg [1:0] nxt_state;
117  reg [1:0] cur_state;
118  always @(cur_state,ready)begin
119    case(cur_state)
120      S0: if(ready) nxt_state = S1;
121      S1:           nxt_state = S2;
122      S2:           nxt_state = S3;
123      S3:           nxt_state = S0;
124      default:      nxt_state = S0;
125    endcase
126  end
127
128  always @(posedge clk) begin
129    if(rst) begin
130      cur_state <= S0       ;
131    end else begin
132      cur_state <= nxt_state;
133    end
134  end
```