Lecture 8 FSM-Datapath Design and Bus Communication

Outline

8.1 FSM-Datapath and Bus Communication

8.2 Bus Communication Mechanisms

8.3 Design Example: I2C Write

8.4 Design Example: MSBUS Communication

Outline

8.1 FSM-Datapath and Bus Communication

8.2 Bus Communication Mechanisms

8.3 Design Example: I2C Write

8.4 Design Example: MSBUS Communication

8.1.1 FSM-Datapath (FSMD) Construction

- Finite State Machine (FSM) and Datapath
  - Timing controller: it typically employs finite state machines (FSMs) or counters to control the timing and behavior of the circuit over different states or clock cycles.
  - Datapath: comprises the functional components responsible for executing the actual computations and data processing tasks
    - For example, the arithmetic unites such as adders, multipliers, audio/image processing modules, bus interfaces for moving data in/out memories, etc. They can be controlled by timing controllers.

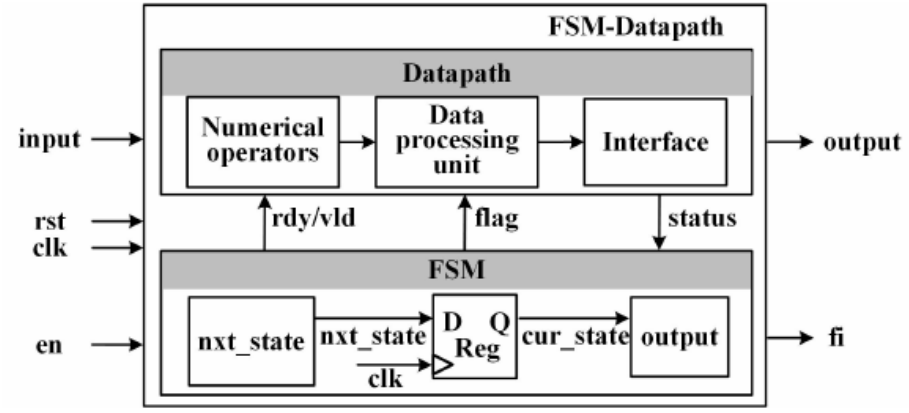


FIGURE 8.1
FSMD Design Structure

8.1.1 FSM-Datapath (FSMD) Construction

- FSM and Datapath
  - Control signals and status signals
    - Facilitats communication between the controller and the datapath.
    - Control signals: initiating or activating specific components within the datapath.
    - Status signals: providing feedback from the datapath to the controller.
  - Ready-valid mechanism (referred to as ``rdy-vld'')
  - Enable-finish handshaking (referred to as ``en-fi'') mechanism.

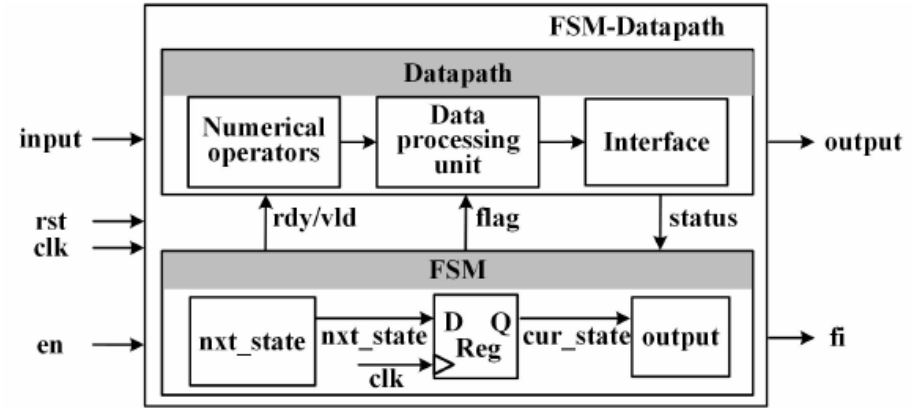


FIGURE 8.1
FSMD Design Structure

8.1.2 Bus Communications

- Terminology:
  - ``Design modules'': smaller-scale elements, such as the state controller and the datapath.
  - ``Hardware devices'': components operating at the system level.
    - Master or slave devices engaged in communication through industry-standard bus protocols such as I2C and UART.
- A. Bus Protocols for Design Modules
  - In RTL design, bus protocols play a vital role in enabling control and data communication among design modules.
    - For example, ready-valid mechanism, enable-finish handshaking, and request-grant arbitration.
- B. Bus Protocols for Hardware Devices
    - Bus communications for hardware devices are typically defined in accordance with industry standards, which may be established by organizations such as IEEE, as well as by self-design teams.
    - I2C, MSBUS

Outline

8.1 FSM-Datapath and Bus Communication

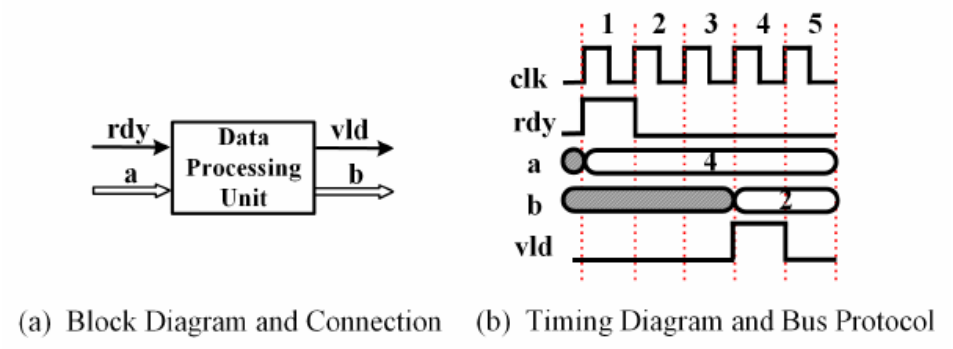
8.2 Bus Communication Mechanisms

8.3 Design Example: I2C Write

8.4 Design Example: MSBUS Communication

8.2.1 Ready-Valid Protocol

- Ready-Valid Protocol
  - A. Bus Protocol
    - Ready: The data on the input bus is ready-to-use;
    - Valid: Validate data output in specific clock cycles.
  - B. Design Examples
    - A crucial aspect of RTL design
      - Data processing and movement are dependent on the availability of data within specific time slots.
    - Typical design example: AMBA AXI Channels



(a) Block Diagram and Connection (b) Timing Diagram and Bus Protocol

FIGURE 8.2
Ready-Valid Protocol

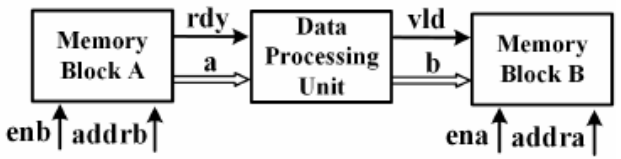
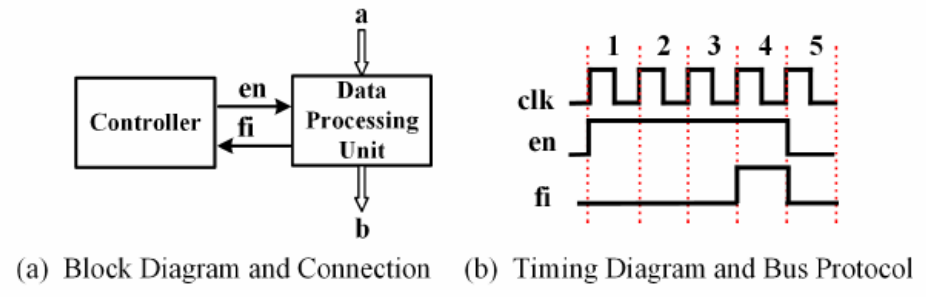


FIGURE 8.3
An Example of Data Movement

8.2.2 Enable-Finish Handshaking

- Enable-Finish Handshaking
  - A. Bus Protocol
    - Enable: initiate a data processing operation
    - Finish: indicate the completion of the process.
  - B. Design Examples
    - A fundamental idea of hardware design:
      - Streamline the coordination, control, and status monitoring among interconnected design modules.
    - Typical design example: Multi-core scheduler



(a) Block Diagram and Connection (b) Timing Diagram and Bus Protocol

FIGURE 8.4
Enable-Finish Handshaking

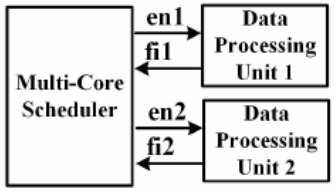
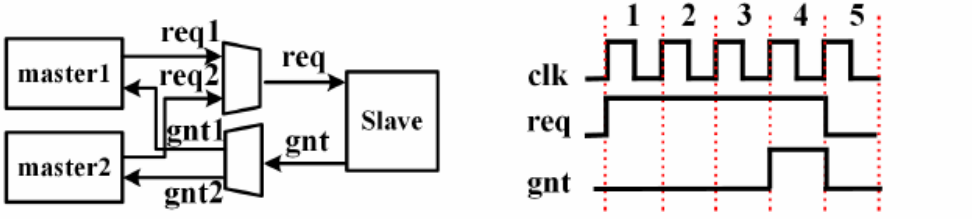


FIGURE 8.5
An Example of Multi-Core Scheduling

8.2.3 Request-Grant Arbitration

- Request-Grant Arbitration
  - A. Bus Protocol
    - Request: request bus access
    - Grant: grant the specific request
    - Used for bus access control in multi-master systems.
  - B. Design Examples
    - Ensures fair and efficient bus access in multi-master systems.
    - Prevent conflicts and maintains a well-coordinated operation of the design system.
    - Typical design example : DMA (Direct Memory Access) arbitration



(a) Block Diagram and Topology (b) Timing Diagram and Bus Protocol

FIGURE 8.6
Request-Grant Arbitration

Outline

8.1 FSM-Datapath and Bus Communication

8.2 Bus Communication Mechanisms

8.3 Design Example: I2C Write

8.4 Design Example: MSBUS Communication

8.3 Design Example: I2C Write

- Serial buses
  - Serial buses have become ubiquitous in modern applications, ranging from general-purpose micro-controllers and processors to various components like EEPROMs and Flash Controllers.
  - There are several serial bus families available, including I2C, SPI, SDIO, GPIO, UART, and many more.
  - From an IC design perspective, all of these serial buses can be implemented using FSMD designs.
- Design Example: I2C Write
  - The I2C bus, invented by Philips Semiconductor in 1982, has become a widely adopted de facto serial bus standard for connecting low-speed peripheral ICs to processors.
  - Its popularity stems from its hardware efficiency and circuit simplicity.

8.3.1 I2C Bus Protocol

- I2C Bus Protocol
  - Only two single-bit lines are required: a serial data line (SDA) and a serial clock line (SCL).
  - There is no strict baud rate requirement as the I2C master provides the bus clock.
  - Serial, 8-bit oriented, bidirectional data transfers can be performed at different speeds including standard-mode (up to 100 kbit/s), fast-mode (up to 400 kbit/s), fast-mode plus (up to 1 Mbit/s), and high-speed mode (up to 3.4 Mbit/s).
  - Simple master-slave relationships exist among all I2C devices. Each device connected to the bus can be addressed using a unique device address.

8.3.1.1 START and STOP Conditions

- START and STOP Conditions
  - All transactions commence with a START (S) signal and conclude with a STOP (P) signal. The generation of START and STOP conditions is exclusively the responsibility of the master controller.
  - START: a transition from HIGH to LOW on the SDA line while SCL is HIGH. Once the START condition is established, the bus is considered busy.
  - STOP: a transition from LOW to HIGH on the SDA line while SCL is HIGH. After the STOP condition, the bus is considered free again after a certain period of time.
  - Repeated START: Similar to a regular START condition, but it occurs before a STOP condition, even when the bus is not idle.

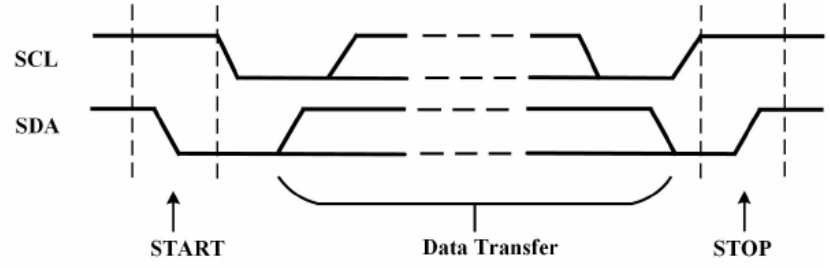


FIGURE 8.7
START and STOP Conditions

3.3.1.2 Data Validity and Byte Format

- Data Validity and Byte Format
  - Each clock pulse of the SCL bus corresponds to the transfer of one data bit on the SDA line.
  - A byte consists of eight bits on the SDA line and can represent various information such as a device address, register address, or data to be written into or read from a slave device.
  - The data is transmitted most significant bit (MSB) first and least significant bit (LSB) last. In the example provided, the data byte being transferred is binary 8'b10101010 or hexadecimal 8'haa..
  - During the high phase of the clock period, the data on the SDA line must remain stable, as any changes in the data line when the SCL bus is high will be interpreted as control commands such as START or STOP.

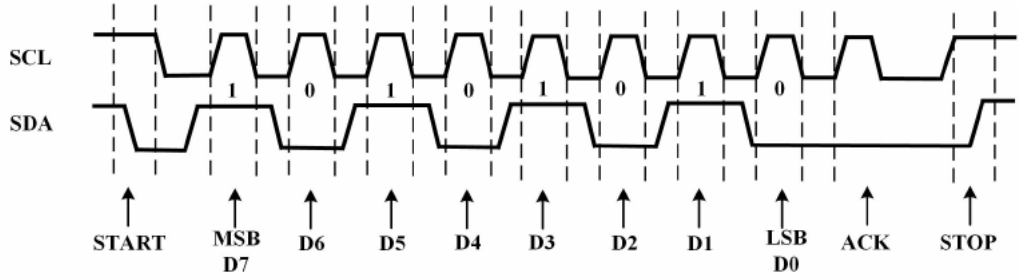


FIGURE 8.8
Single-Byte Data Transfer with ACK

8.3.1.3 Acknowledge (ACK) and Not Acknowledge (NACK)

- Acknowledge (ACK)
  - After each data byte is transferred, the slave must send an ACK (Acknowledge) bit to the master. This ACK bit serves as confirmation to the master that the data byte was successfully received, and it signals that another data byte can be sent.
    - Master: To respond with an ACK bit, the master must release the SDA line, allowing the slave to control the line.
    - Slave: Pulls down the SDA line during the low phase of the ACK/NACK-related clock period (after the last cycle of the data bit 0), ensuring that the SDA line remains stably low during the high phase of the ACK/NACK-related clock period.

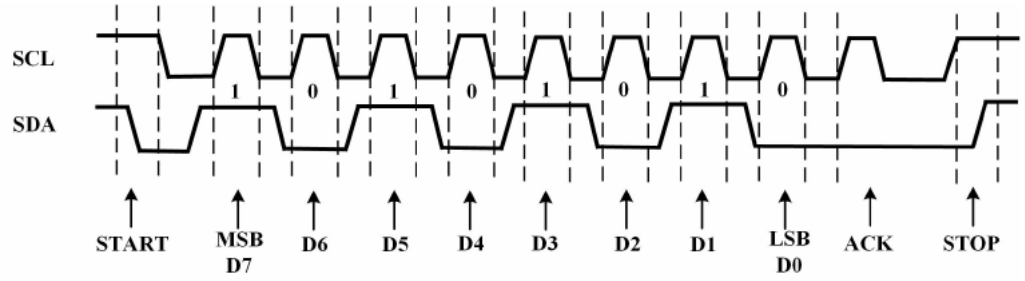


FIGURE 8.8
Single-Byte Data Transfer with ACK

8.3.1.3 Acknowledge (ACK) and Not Acknowledge (NACK)

- Not Acknowledge (NACK)
  - If the SDA line remains high during the ACK/NACK-related clock period, this is interpreted as a NACK (Not Acknowledge).
    - The slave device is not ready to communicate with the master and is unable to receive or transmit data.
    - The slave device receives data or commands that it does not understand during the transfer.
    - The slave device reaches a point where it cannot receive any more data bytes.
    - The master device indicates that it has finished reading data and communicates this to the slave through a NACK signal.

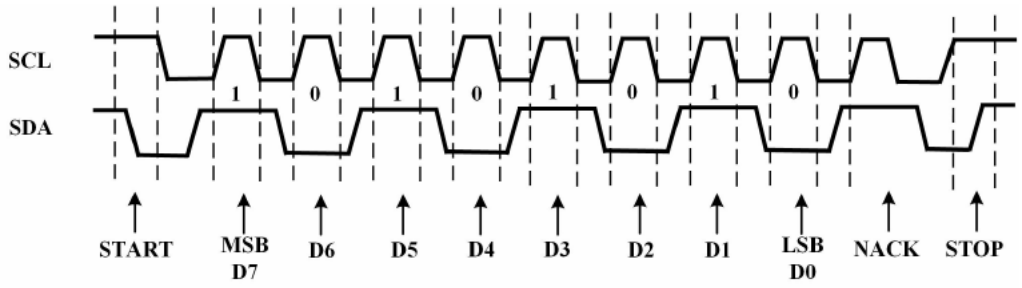


FIGURE 8.9
Single-Byte Data Transfer with NACK

8.3.1.4 Write Operations on I2C Bus

- Write Operations on I2C Bus
  - The master initiates communication by generating a START condition.
  - The master transmits a 7-bit device address (A6-A0) with the final bit (R/W bit) set to zero, indicating a write operation. Upon receiving the device address, the slave confirms the transmission by sending an ACK bit.
  - The master sends the 8-bit register address (B7-B0) of the specific register it intends to write to. The slave acknowledges the register address by transmitting an ACK bit, signifying its readiness to receive the data.
  - The master transmits the actual data (D7-D0) to be written into the register. The slave acknowledges the data transmission.
  - Upon transmitting all the necessary data, the master concludes the operation by issuing a STOP condition on the bus.

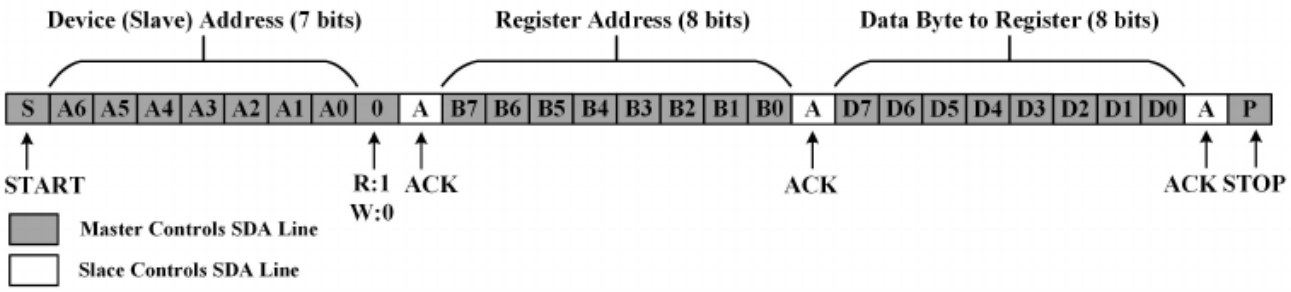


FIGURE 8.10
I2C Write to a Device's Register

8.3.1.5 Read Operations on I2C Bus

- Read Operations on I2C Bus
  - The master initiates communication by generating a START condition.
  - It is important to note that the master should initiate the transmission by sending the device address with the R/W bit set to zero, indicating a write operation firstly. It also includes the register address it wishes to read from. The slave acknowledges both the device address and the register address.
  - The master sends a repeated START condition on the bus, followed by the device address with the R/W bit set to one, indicating a read operation. Here, it starts the I2C read operation. The slave acknowledges the read request.

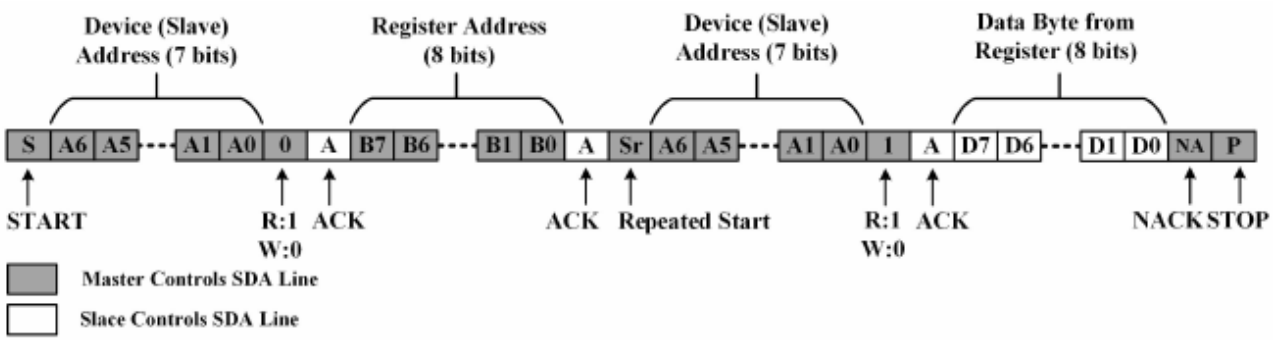


FIGURE 8.11
I2C Read from Device's Register

8.3.1.5 Read Operations on I2C Bus

- Read Operations on I2C Bus
  - The master releases the SDA line, allowing the slave to transmit data. The master continues to provide clock pulses to synchronize the transmission. The slave, acting as the slave-transmitter, sends data on the SDA line during each SCL pulse. After receiving each byte of data, the master sends an ACK signal to the slave, indicating that it is ready for more data.
  - Once the master has received the desired number of bytes, it sends a NACK signal to the slave, signaling the end of the communication and instructing the slave to release the bus.
  - The master concludes the transaction with a STOP condition.

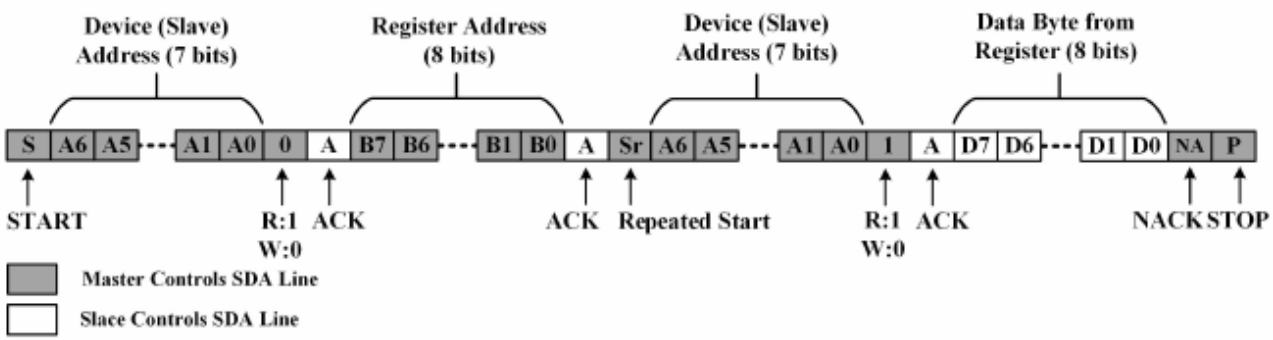


FIGURE 8.11
I2C Read from Device's Register

8.3.2 An FSMD Design Example: I2C Write Operations

- A. Design Requirements for An I2C-Enabled Device
  - The SCL frequency should not exceed 400 KHz as a fast-mode I2C-enabled device.
  - The setup time (rising edge of SCL to falling edge of SDA) and hold time (falling edge of SDA to falling edge of SCL) of the START condition are specified as a minimum of 5 us.
  - The setup time and hold time of the STOP condition are specified as a minimum of 5 us.

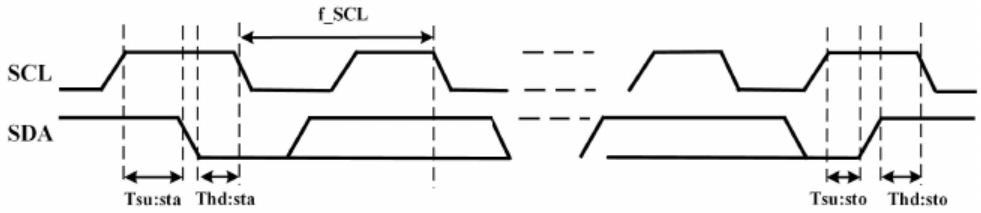


FIGURE 8.12
I2C Write and Read Timing Constraints

TABLE 8.1
Timing Requirements from the I2C Device

Symbo	Parameter	Min	Max	Unit
f_SCL	Clock Frequency		400	KHz
Tsu:sta	START Condition Setup Time	5		us
Thd:sta	START Condition Hold Time	5		us
Tsu:sto	STOP Condition Setup Time	5		us
Thd:sto	Hold Condition Hold Time	5		us

8.3.2 An FSMD Design Example: I2C Write Operations

- B. Design Specifications
  - Firstly, the design module should adhere to the I2C protocol, to initiate the I2C START and STOP conditions and transmit the required data including the device address, register address, and register data.
    - As a results, Figure 8.13 depicts an FSM controller to manage the state transitions including seven states: the initial state (INI), I2C START state (START), device address sending (ID), register address sending (ADDR), data sending (DATA), and the final STOP state (STOP).

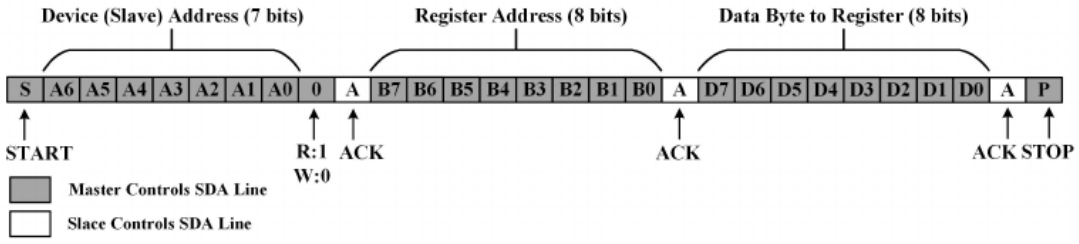


FIGURE 8.10
I2C Write to a Device's Register

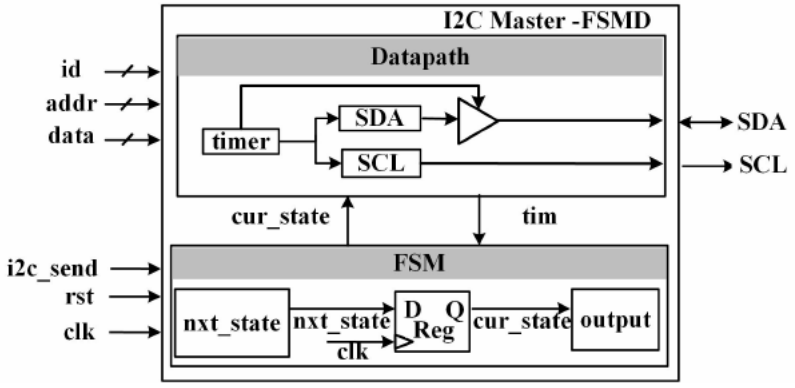


FIGURE 8.13
I2C Master Design with FSMD

8.3.2 An FSMD Design Example: I2C Write Operations

- B. Design Specifications
  - Secondly, it is crucial to adhere to the timing requirements specified in Table 8.1, to ensure proper communication with the I2C device.
    - The serial clock (SCL) must not exceed the specified limit of 400 KHz.
      - Assuming that a 50 MHz clock is available, each clock period is 20 ns.
      - 64x20 ns = 1.28 us, which produces an SCL period of 1.28x2 ns = 2.56 us, or 1/2.56 ≈ 400 KHz.
    - Setup time and hold time requirements for the START and STOP conditions
      - Each set of four timer units takes approximately 5 microseconds (4x1.28 ≈5 us)
      - A timer is essential for counting half of SCL cycles, which equates to 64x50-MHz cycles, or 1.28 us.

TABLE 8.1
Timing Requirements from the I2C Device

Symbo	Parameter	Min	Max	Unit
f_SCL	Clock Frequency		400	KHz
Tsu:sta	START Condition Setup Time	5		us
Thd:sta	START Condition Hold Time	5		us
Tsu:sto	STOP Condition Setup Time	5		us
Thd:sto	Hold Condition Hold Time	5		us

8.3.3 Datapath Design

- Datapath Design
  - A. START and STOP Stages
  - B. ID, ADDR, and DATA Stages
    - Assuming that the I2C device features a 7-bit unique device ID, represented as 7'b0010001, the device address can be expressed as 8'b00100010 by appending the last digit zero, which represents write operations.
    - After the transmission of the 8-bit device address, register address, and register data, a high-impedance (high-Z) status is employed to release the bus from the I2C master's control. Subsequently, the slave drives the bus with either an ACK or NACK status, indicating the slave's readiness to receive the byte or not.

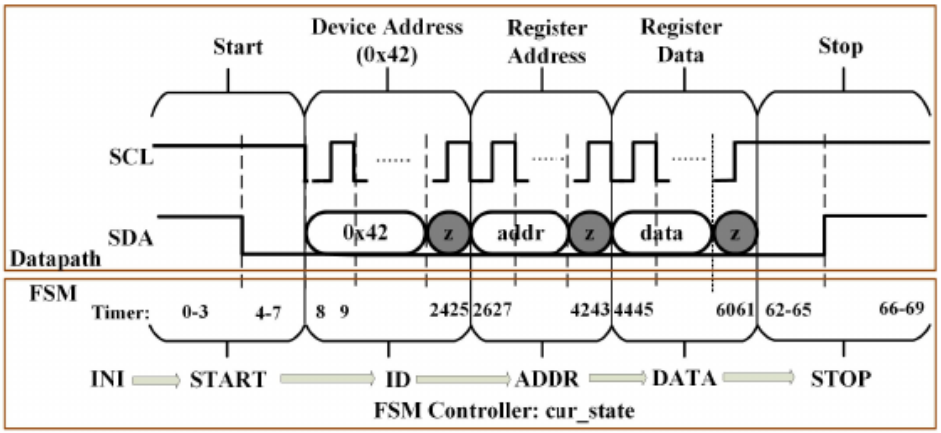


FIGURE 8.14
I2C Write Timing Diagram with FSMD

- FSM Design
  - State Graph
    - The input signals are as follows: the timer signal, denoted as ``tim'', the counter signal, denoted as ``cnt'', and the control signals ``rst'' (reset) and ``i2c_en'' (I2C enable), which govern the FSM's operation.

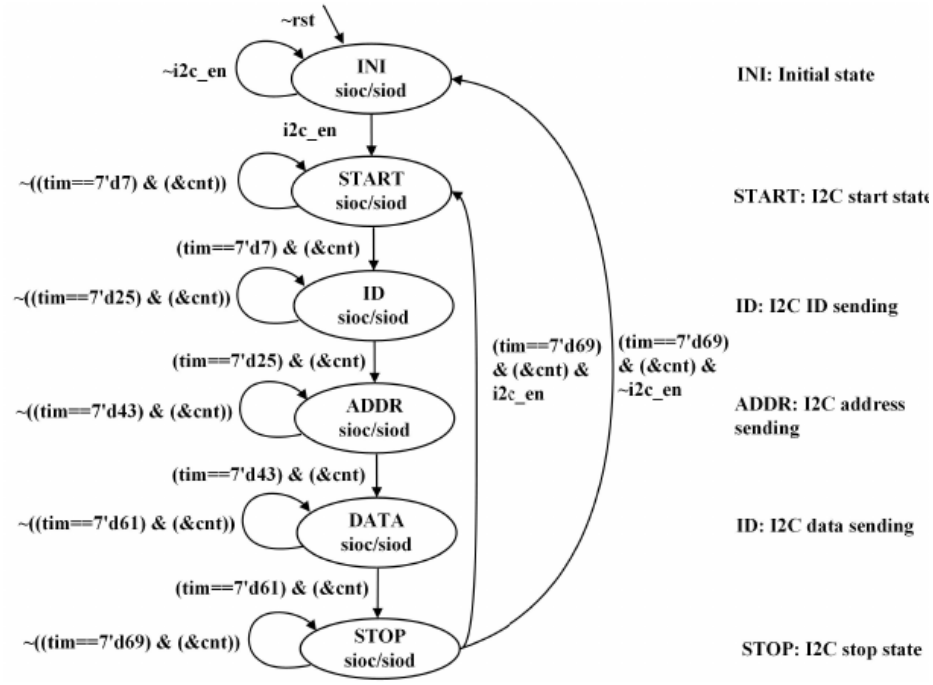


FIGURE 8.15
I2C Write State Graph

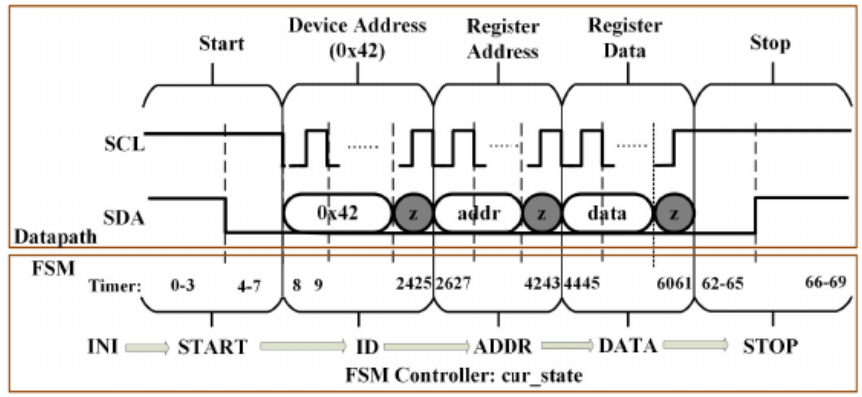


FIGURE 8.14
I2C Write Timing Diagram with FSMD

8.2.5 I2C Master Design with Verilog HDL

- I2C Master Design with Verilog HDL
  - Counter and Timer

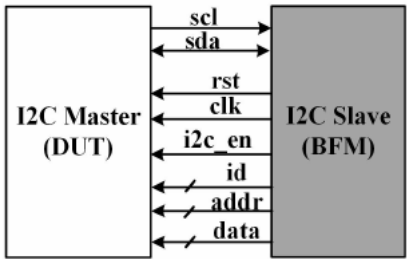


FIGURE 8.16
I2C Write Design Simulation

```
1  module i2c_master  (input        clk      ,
2                      input        rst      ,
3                      input        i2c_en   ,
4                      input [7:0]  id       ,
5                      input [7:0]  addr     ,
6                      input [7:0]  data     ,
7                      inout        sda      ,
8                      output reg   scl      );
9  reg [5:0] cnt       ;
10 reg [6:0] tim       ;
11 reg [2:0] nxt_state ;
12 reg [2:0] cur_state ;
13
14 //-------------------------------------------------//
15 // cnt: 0~64, timer unit - 64x50MHz clock is 1.28 us //
16 // tim: timing control for scl and sda              //
17 //-------------------------------------------------//
18 wire [5:0]  nxt_cnt = i2c_en ? cnt+6'd1 : cnt;
19 always @(posedge clk, negedge rst) begin
20   if(~rst) begin
21     cnt <= 6'd0   ;
22   end else begin
23     cnt <= nxt_cnt;
24   end
25 end
26
27 wire [6:0] nxt_tim = tim==7'd69 & &cnt ? 7'd0 :
28                                  i2c_en & &cnt ? tim+7'd1 : tim;
29 always @(posedge clk, negedge rst) begin
30   if(~rst) begin
31     tim <= 7'd0;
32   end else begin
33     tim <= nxt_tim;
34   end
35 end
```

8.2.5 I2C Master Design with Verilog HDL

- I2C Master Design with Verilog HDL
  - FSM

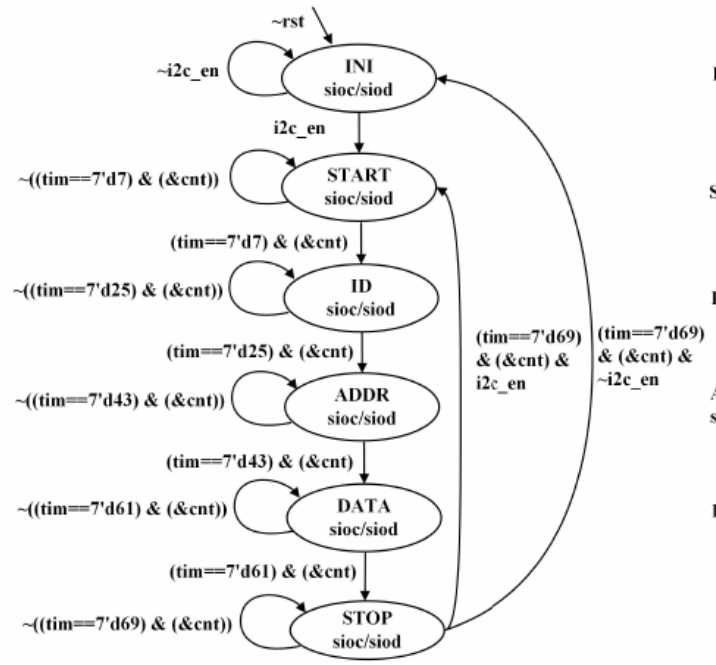


FIGURE 8.15
I2C Write State Graph

```
37 //-------------------------------------------------//
38 //-----------   finite state machine      -----------//
39 //-------------------------------------------------//
40 parameter INI  =3'h0;
41 parameter START=3'h1;
42 parameter ID   =3'h2;
43 parameter ADDR =3'h3;
44 parameter DATA =3'h4;
45 parameter STOP =3'h5;
46
47 always @(posedge clk, negedge rst) begin
48   if(~rst) begin
49     cur_state <= INI      ;
50   end else begin
51     cur_state <= nxt_state;
52   end
53 end
54
55 always @(rst, cur_state, i2c_en, cnt, tim) begin
56   if (~rst) begin
57     nxt_state <= INI;
58   end else begin
59     case (cur_state)
60       INI  : if(i2c_en)              nxt_state <= START;
61       START: if(tim==7'd7  & (&cnt)) nxt_state <= ID   ;
62       ID   : if(tim==7'd25 & (&cnt)) nxt_state <= ADDR ;
63       ADDR : if(tim==7'd43 & (&cnt)) nxt_state <= DATA ;
64       DATA : if(tim==7'd61 & (&cnt)) nxt_state <= STOP ;
65       STOP : if(tim==7'd69 & (&cnt)) begin
66                if(i2c_en) begin
67                  nxt_state <= START ;
68                end else begin
69                  nxt_state <= INI   ;
70                end
71              end
72     endcase
73   end
74 end
```

8.2.5 I2C Master Design with Verilog HDL

- I2C Master Design with Verilog HDL
  - Datapath: SCL and SDA

```
93  //-----------------------------------------------------//
94  //------------------    sda    ----------------------//
95  //-----------------------------------------------------//
96  reg sda_reg;
97  assign sda=(tim>=7'd24 & tim<=7'd25) |
98             (tim>=7'd42 & tim<=7'd43) |
99             (tim>=7'd60 & tim<=7'd61) ? 1'bz : sda_reg;
100
101 always @(rst, cur_state, tim) begin
102   if (~rst) begin
103     sda_reg <= 1'b1;
104   end else begin
105     case (cur_state)
106       INI   : sda_reg <= 1'b1;
107       START: sda_reg <= (tim>=7'd0 & tim<=7'd3);
108       ID:    case (tim)
109                 8, 9 : sda_reg<=id[7];
110                10,11 : sda_reg<=id[6];
111                12,13 : sda_reg<=id[5];
112                14,15 : sda_reg<=id[4];
113                16,17 : sda_reg<=id[3];
114                18,19 : sda_reg<=id[2];
115                20,21 : sda_reg<=id[1];
116                22,23 : sda_reg<=id[0];
117              endcase
118       ADDR: case (tim)
119                26,27 : sda_reg<=addr[7];
120                28,29 : sda_reg<=addr[6];
121                30,31 : sda_reg<=addr[5];
122                32,33 : sda_reg<=addr[4];
123                34,35 : sda_reg<=addr[3];
124                36,37 : sda_reg<=addr[2];
125                38,39 : sda_reg<=addr[1];
126                40,41 : sda_reg<=addr[0];
127              endcase
128       DATA: case (tim)
129                44,45 : sda_reg<=data[7];
130                46,47 : sda_reg<=data[6];
131                48,49 : sda_reg<=data[5];
132                50,51 : sda_reg<=data[4];
133                52,53 : sda_reg<=data[3];
134                54,55 : sda_reg<=data[2];
135                56,57 : sda_reg<=data[1];
136                58,59 : sda_reg<=data[0];
137              endcase
138       STOP:    sda_reg <= (tim>=7'd66 & tim<=7'd69);
139           default: sda_reg <= 1'b1;
140     endcase
141   end
142 end
143 endmodule
```

```
76  //-----------------------------------------------------//
77  //------------------    scl    ----------------------//
78  //-----------------------------------------------------//
79  always @(rst, cur_state, tim) begin
80    if (~rst) begin
81      scl <= 1'b1;
82    end else begin
83      case (cur_state)
84        INI             : scl<=1'b1      ;
85        START           : scl<=1'b1      ;
86        ID, ADDR, DATA: scl<=tim[0]    ;
87        STOP            : scl<=1'b1      ;
88        default         : scl<=1'b1      ;
89      endcase
90    end
91  end
```

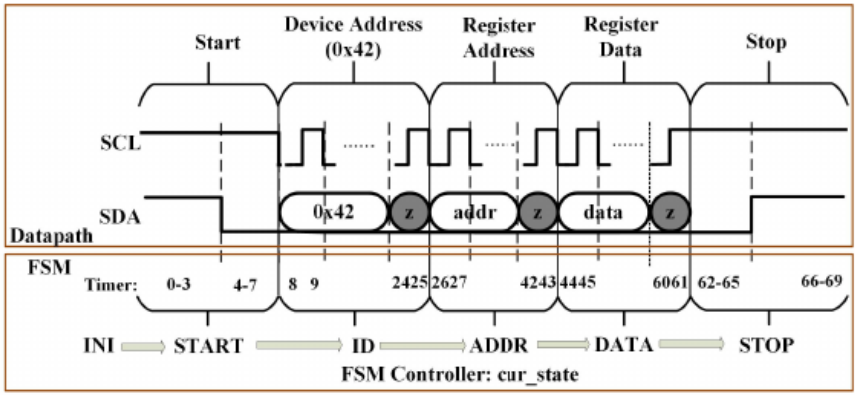


FIGURE 8.14
I2C Write Timing Diagram with FSMD

Outline

8.4.1 MSBUS Protocol

- MSBUS Protocol
  - MSBUS Architecture
    - The single-master and multi-slave bus is named MSBUS, which primarily enables the CPU master to exert control over these slave devices.
      - Single master: a simple bus architecture where the CPU functions as the master of the SoC Bus.
      - Multiple salves: the other devices connected to the bus.

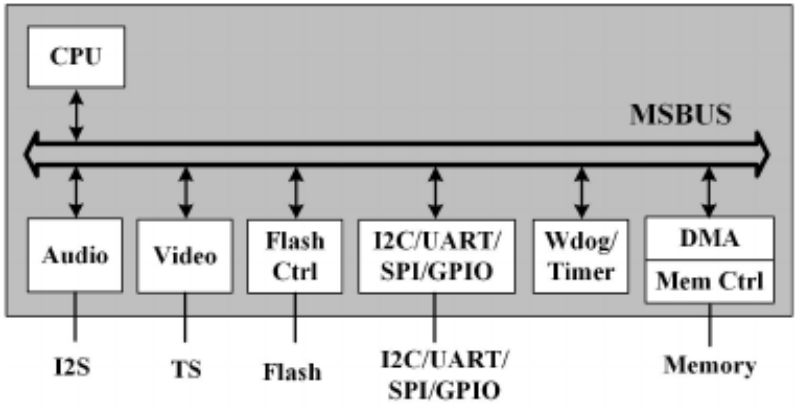


FIGURE 8.17
MSBUS SoC Architecture

8.4.1 Data Transaction Protocol

- MSBUS Protocol
  - Data Transaction Protocol
    - A half-duplex bus: it doesn't allow simultaneous execution of write and read operations.
    - SINGLE transfer: serves as a control bus for configuring functional registers using the SINGLE transfer mode.
      - This mode comprises two stages for each data transaction: the command stage and the data stage, each requiring at least one cycle.
    - Shared bus: the bus ``m_addr_wdata'' serves as a shared bus, accommodating the write address and write data.

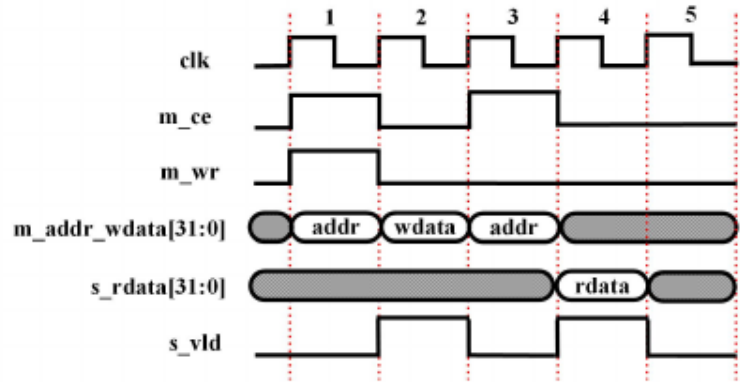
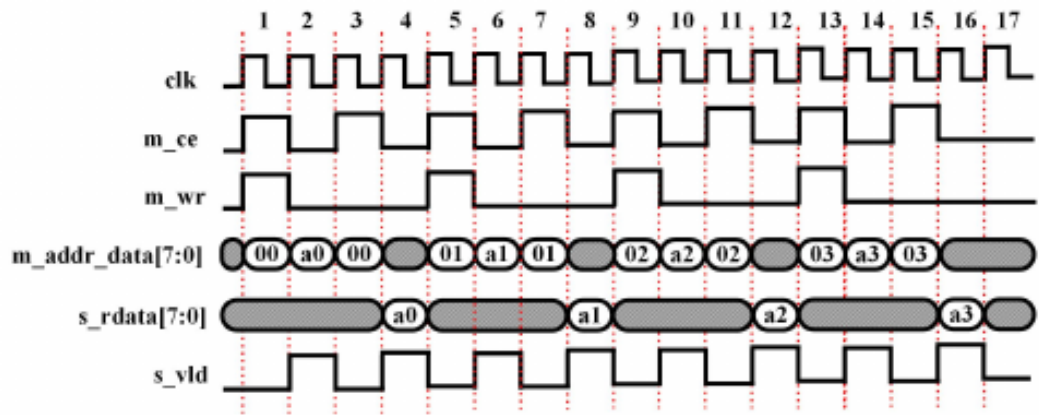


FIGURE 8.18
MSBUS Timing Diagram

8.4.1.3 Functional Register Access with MSBUS

- Functional Register Access with MSBUS
  - A. Byte-Size MSBUS
    - Addresses: 0x00, 0x01, 0x02, 0x03
    - Corresponding data: 0xa0, 0xa1, 0xa2, 0xa3



(a) MSBUS Data Transfer

Bus width: byte

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
.....															
f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	fa	fb	fc	fd	fe	ff

(b) MSBUS Register Access

FIGURE 8.19
Register File Access with Byte-Sized MSBUS

8.4.1.3 Functional Register Access with MSBUS

- Functional Register Access with MSBUS
  - B. Half-Word-Size MSBUS
    - Bus Alignment: In the case of a half-word-size MSBUS, the addresses transmitted on the bus should align with half-word boundaries. This means that the least-significant bit of the address will be disregarded or set to binary zero during MSBUS transactions.
      - An unusual command scenario in which the CPU sends a memory address of 0x0001. At the hardware level, the bus master will discard the least-significant bit of the bus address and set it to binary zero because the bus size is half-word. Consequently, the address transmitted on the bus will be adjusted to 16'h0.

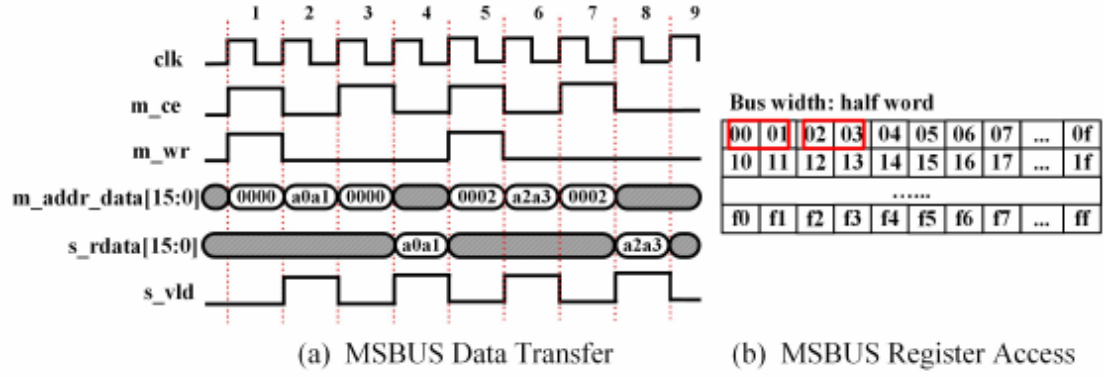


FIGURE 8.20
Register File Access with Half-Word-Sized MSBUS

8.4.2 An FSMD Design Example: MSBUS Master and Slave

- Functional Register Access with MSBUS
  - B. Word-Size MSBUS
    - Bus Alignment: In the case of a word-size MSBUS, the addresses transmitted on the bus should align with word boundaries. This means that the least-significant two bits of the memory address will be disregarded and set to zeros during MSBUS transactions.
      - An abnormal command: where the CPU sends a memory address of 0x00000003. At the hardware level, the bus master will disregard the least-significant two bits of the bus address and set them to binary 2'b00. As a result, the address transmitted on the bus will be corrected to 32'h0.

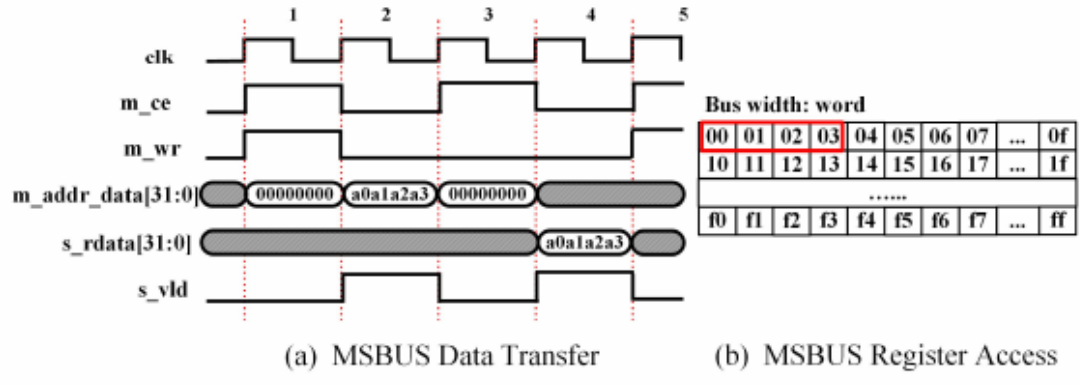


FIGURE 8.21
Register File Access with Word-Sized MSBUS

8.4.2 An FSMD Design Example: MSBUS Master and Slave

- An FSMD Design Example: MSBUS Master and Slave
  - A. Design Structure
    - All essential commands and data are housed in a command file within the MSBUS master. When register access is enabled, the master transmits commands and write data to the slave.
      - The ``cmd_cnt'' value spans from decimal zero to seven, incrementing by one between neighboring registers. The corresponding bus addresses range from hexadecimal 32'h00 to 32'h1c, incrementing by 32'h4 due to the word-sized nature of the MSBUS.
    - After each register write, a read operation is executed to confirm the successful write operation.

```
1  `define CMD_1C  32'h1c
2  `define CMD_18  32'h18
3  `define CMD_14  32'h14
4  `define CMD_10  32'h10
5  `define CMD_0C  32'hc
6  `define CMD_08  32'h8
7  `define CMD_04  32'h4
8  `define CMD_00  32'h0
9
10 `define WD_1C  32'hf7
11 `define WD_18  32'hf6
12 `define WD_14  32'hf5
13 `define WD_10  32'hf4
14 `define WD_0C  32'hf3
15 `define WD_08  32'hf2
16 `define WD_04  32'hf1
17 `define WD_00  32'hf0
```

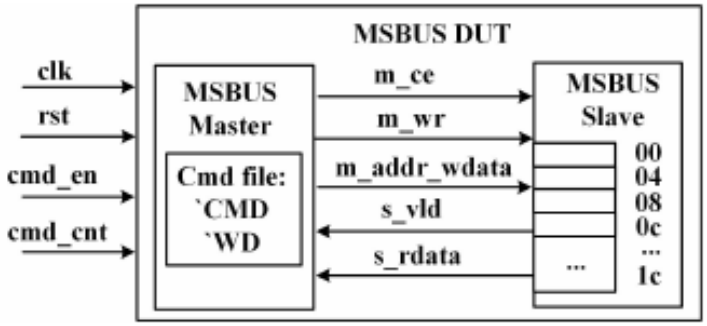


FIGURE 8.22
MSBUS Master-Slave Design Structure

8.4.2 An FSMD Design Example: MSBUS Master and Slave

- An FSMD Design Example: MSBUS Master and Slave
  - B. Design Block Diagram
    - The master FSM is responsible for generating control signals (``mwc_f'', ``mrc_f'', and``mwd_f'').
    - The datapath acknowledges the master FSM by issuing a valid signal (``s_vld'').
    - The slave FSM generates control signals to signify the write data stage (``swd_f'') and the read data stage (``srd_f'').
    - The datapath, in response, provides the master enable signal (``m_ce'') to the slave FSM, enabling it to carry out the required operations.

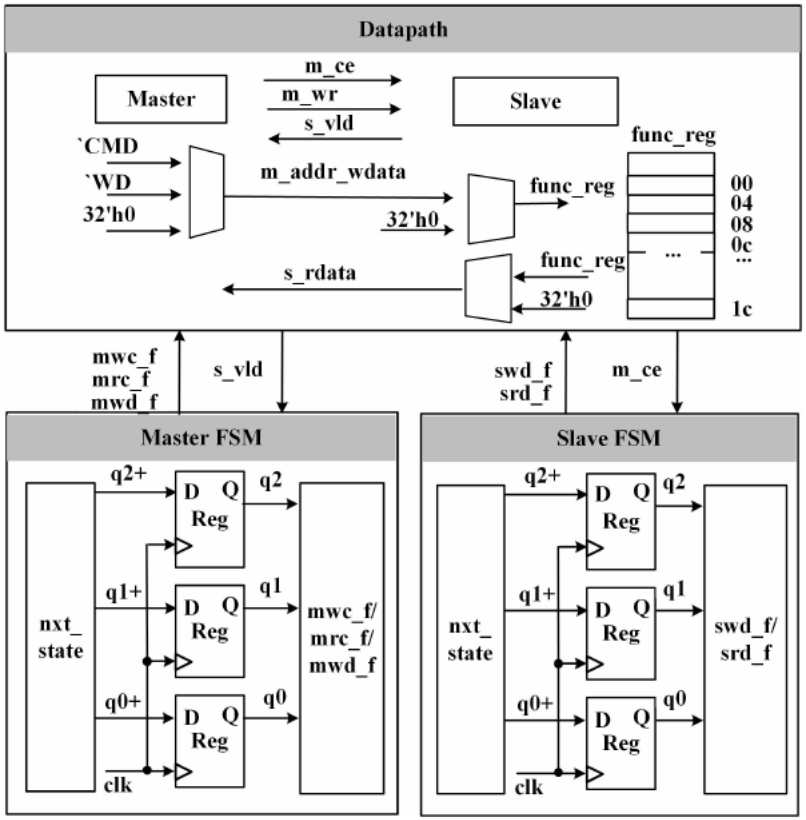


FIGURE 8.23
MSBUS Master-Slave FSMD Structure

8.4.3 MSBUS Master Design with Verilog HDL

- MSBUS Master Design with Verilog HDL
  - A. FSM Design of MSBUS Master
  - B. MSBUS Master Design with Verilog HDL

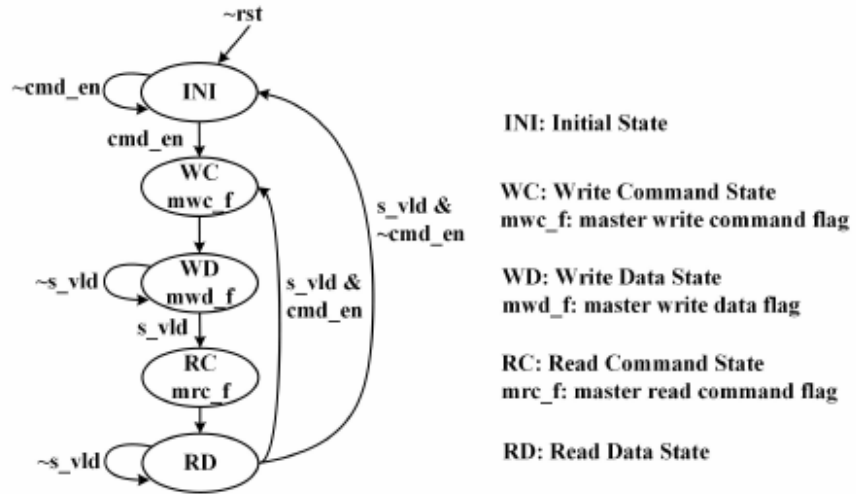


FIGURE 8.24
MSBUS Master State Graph

```
`define CMD_1C  32'h1c
`define CMD_18  32'h18
`define CMD_14  32'h14
`define CMD_10  32'h10
`define CMD_0C  32'hc
`define CMD_08  32'h8
`define CMD_04  32'h4
`define CMD_00  32'h0

`define WD_1C   32'hf7
`define WD_18   32'hf6
`define WD_14   32'hf5
`define WD_10   32'hf4
`define WD_0C   32'hf3
`define WD_08   32'hf2
`define WD_04   32'hf1
`define WD_00   32'hf0
```

```
`include "command.sv"
module msbus_master (input                clk          ,
                     input                rst          ,
                     input                cmd_en       ,
                     input        [2:0]   cmd_cnt      ,
                     output               m_ce         ,
                     output               m_wr         ,
                     output reg   [31:0]  m_addr_wdata ,
                     input        [31:0]  s_rdata      ,
                     input                s_vld        );
parameter INI = 3'h0  ;
parameter WC  = 3'h1  ;
parameter WD  = 3'h2  ;
parameter RC  = 3'h3  ;
parameter RD  = 3'h4  ;

//-------------------------------------
//------- current state--------------
//-------------------------------------
reg [2:0] cur_state, nxt_state;
always @(posedge clk, negedge rst) begin
  if(~rst) begin
    cur_state <= INI      ;
  end else begin
    cur_state <= nxt_state;
  end
end
```

8.4.3 MSBUS Master Design with Verilog HDL

- MSBUS Master Design with Verilog HDL
  - A. FSM Design of MSBUS Master
  - B. MSBUS Master Design with Verilog HDL

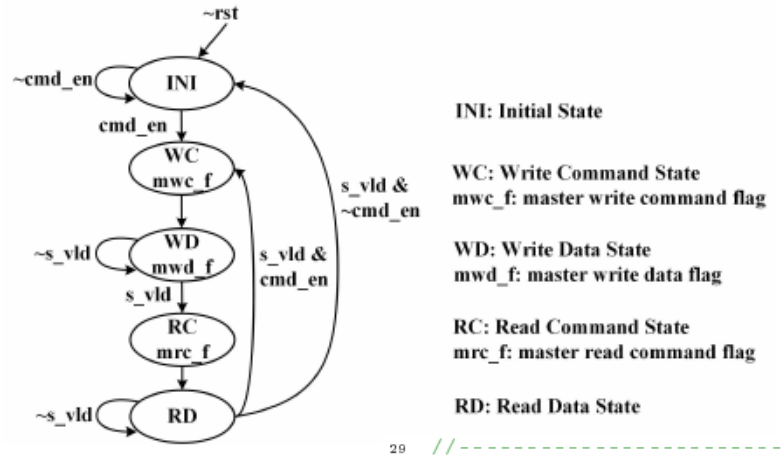


FIGURE 8.24
MSBUS Master State Graph

```
29 //----------------------------------------
30 //------- next state-----------------
31 //----------------------------------------
32 always @(rst, cur_state, cmd_en, s_vld) begin
33   if(~rst) begin
34     nxt_state <= INI;
35   end else begin
36     case (cur_state)
37       INI: if(cmd_en) nxt_state <= WC;
38       WC:             nxt_state <= WD;
39       WD:  if (s_vld) nxt_state <= RC;
40       RC:             nxt_state <= RD;
41       RD:  if (s_vld) begin
42              if (cmd_en) nxt_state <= WC;
43              else        nxt_state <= INI;
44            end
45       default:        nxt_state <= INI;
46     endcase
47   end
48 end
```

```
50 //----------------------------------------
51 //------- output logic------------
52 //----------------------------------------
53 wire mwc_f = cur_state==WC;
54 wire mrc_f = cur_state==RC;
55 wire mwd_f = cur_state==WD;
```

```
57 //----------------------------------------
58 //------- master data path--------------
59 //----------------------------------------
60 always @(rst, mwc_f, mrc_f, cmd_cnt, mwd_f) begin
61   if(~rst) begin
62     m_addr_wdata = 32'h0;
63   end if (mwc_f | mrc_f)  begin
64     case (cmd_cnt)
65       3'd7 : m_addr_wdata = `CMD_1C;
66       3'd6 : m_addr_wdata = `CMD_18;
67       3'd5 : m_addr_wdata = `CMD_14;
68       3'd4 : m_addr_wdata = `CMD_10;
69       3'd3 : m_addr_wdata = `CMD_0C;
70       3'd2 : m_addr_wdata = `CMD_08;
71       3'd1 : m_addr_wdata = `CMD_04;
72       3'd0 : m_addr_wdata = `CMD_00;
73       default: m_addr_wdata = 32'h0;
74     endcase
75   end else if (mwd_f) begin
76     case (cmd_cnt)
77       3'd7 : m_addr_wdata = `WD_1C;
78       3'd6 : m_addr_wdata = `WD_18;
79       3'd5 : m_addr_wdata = `WD_14;
80       3'd4 : m_addr_wdata = `WD_10;
81       3'd3 : m_addr_wdata = `WD_0C;
82       3'd2 : m_addr_wdata = `WD_08;
83       3'd1 : m_addr_wdata = `WD_04;
84       3'd0 : m_addr_wdata = `WD_00;
85       default: m_addr_wdata = 32'h0;
86     endcase
87   end
88 end
89
90 assign m_ce = mwc_f | mrc_f ;
91 assign m_wr = mwc_f | ~mrc_f;
92 endmodule
```

8.4.3 MSBUS Slave Design with Verilog HDL

- MSBUS Slave Design with Verilog HDL
  - A. FSM Design of MSBUS Slave
  - B. MSBUS Slave Design with Verilog HDL
    - Big-Endian: the most significant byte of a multi-byte data word is stored at the lowest memory address or transmitted first over the bus. The subsequent bytes are then stored or transmitted in decreasing order of significance.

      For example, in a 32-bit word (32'haabbccdd) with address 32'h0, the most significant byte (8'haa) should be stored at address 32'h0, followed by byte 2 (8'hbb) at address 32'h4, byte 1 (8'hcc) at address 32'h8, and the least significant byte (8'hdd) at address 32'hc.
    - Little-endian: the least significant byte of a multi-byte data word is stored at the lowest memory address or transmitted first over the bus. The subsequent bytes are then stored or transmitted in increasing order of significance.

      For example, in a 32-bit word (32'haabbccdd) with address 32'h0, the least significant byte (8'hdd) would be stored at address 32'h0, followed by byte 1 (8'hcc) at address 32'h4, byte 2 (8'hbb) at address 32'h8, and the most significant byte (8'haa) at address 32'hc.

8.4.3 MSBUS Slave Design with Verilog HDL

- MSBUS Slave Design with Verilog HDL
  - A. FSM Design of MSBUS Slave
  - B. MSBUS Slave Design with Verilog HDL

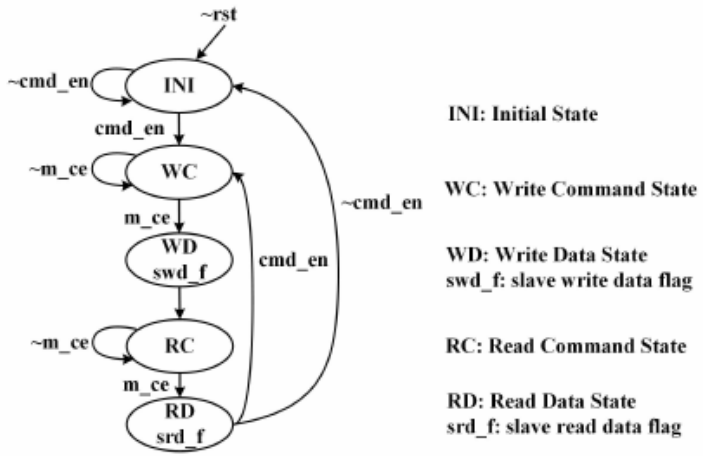


FIGURE 8.25
MSBUS Slave State Graph

```
`define CMD_1C  32'h1c
`define CMD_18  32'h18
`define CMD_14  32'h14
`define CMD_10  32'h10
`define CMD_0C  32'hc
`define CMD_08  32'h8
`define CMD_04  32'h4
`define CMD_00  32'h0

`define WD_1C  32'hf7
`define WD_18  32'hf6
`define WD_14  32'hf5
`define WD_10  32'hf4
`define WD_0C  32'hf3
`define WD_08  32'hf2
`define WD_04  32'hf1
`define WD_00  32'hf0
```

```
`include "command.sv"
module msbus_slave (input           clk           ,
                    input           rst           ,
                    input           cmd_en        ,
                    input  [7:0]    cmd_cnt       ,
                    input           m_ce          ,
                    input           m_wr          ,
                    input  [31:0]   m_addr_wdata  ,
                    output [31:0]   s_rdata       ,
                    output          s_vld         );
reg [7:0] func_reg[0:31] ;

parameter INI = 3'h0;
parameter WC  = 3'h1;
parameter WD  = 3'h2;
parameter RC  = 3'h3;
parameter RD  = 3'h4;

//-------------------------------------
//------- current state--------------
//-------------------------------------
reg [2:0] cur_state, nxt_state;
always @(posedge clk, negedge rst) begin
  if(~rst) begin
    cur_state <= 0;
  end else begin
    cur_state <= nxt_state;
  end
end

//-------------------------------------
//------- next state-----------------
//-------------------------------------
always @(*) begin
  if(~rst) begin
    nxt_state <= INI;
  end else begin
    case (cur_state)
      INI: if (cmd_en) begin
             nxt_state <= WC;
           end else begin
             nxt_state <= INI;
           end
      WC:  if(m_ce)    nxt_state <= WD ;
      WD:               nxt_state <= RC ;
      RC:  if(m_ce)    nxt_state <= RD ;
      RD:  if(cmd_en) nxt_state <= WC ;
           else        nxt_state <= INI;
    endcase
  end
end
```

```
53  //---------------------------------------
54  //------- output logic---------------
55  //---------------------------------------
56  wire swd_f = cur_state==WD;
57  wire srd_f = cur_state==RD;
58
59  //---------------------------------------
60  //------- slave data path- --------------
61  //---------------------------------------
62  integer i;
63  always @(rst, swd_f, m_addr_wdata) begin
64    if(~rst) begin
65      for (i=0; i<32; i=i+1) begin
66        func_reg[i] <= 8'h0;
67      end
68    end else if (swd_f) begin
69  `ifdef BIG_ENDIAN
70      func_reg[4*cmd_cnt+3]   <= m_addr_wdata[7 : 0];
71      func_reg[4*cmd_cnt+2]   <= m_addr_wdata[15: 8];
72      func_reg[4*cmd_cnt+1]   <= m_addr_wdata[23:16];
73      func_reg[4*cmd_cnt  ]   <= m_addr_wdata[31:24];
74  `elsif LITTLE_ENDIAN
75      func_reg[4*cmd_cnt+3]   <= m_addr_wdata[31:24];
76      func_reg[4*cmd_cnt+2]   <= m_addr_wdata[23:16];
77      func_reg[4*cmd_cnt+1]   <= m_addr_wdata[15: 8];
78      func_reg[4*cmd_cnt  ]   <= m_addr_wdata[7 : 0];
79  `endif
80    end
81  end
82
83  `ifdef BIG_ENDIAN
84  assign s_rdata = srd_f ? {func_reg[4*cmd_cnt]   ,
85                            func_reg[4*cmd_cnt+1],
86                            func_reg[4*cmd_cnt+2],
87                            func_reg[4*cmd_cnt+3]} : 32'h0 ;
88  `elsif LITTLE_ENDIAN
89  assign s_rdata = srd_f ? {func_reg[4*cmd_cnt+3],
90                            func_reg[4*cmd_cnt+2],
91                            func_reg[4*cmd_cnt+1],
92                            func_reg[4*cmd_cnt  ]} : 32'h0 ;
93  `endif
94
95  assign s_vld   = swd_f | srd_f ;
96  endmodule
```

8.4.3 MSBUS Master Design with Verilog HDL

- MSBUS Master Design with Verilog HDL
  - A. FSM Design of MSBUS Master
  - B. MSBUS Master Design with Verilog HDL

8.4.5 MSBUS Master-Slave Circuit Analysis

- MSBUS Master-Slave Circuit Analysis

cur_state	nxt_state		mwc_f	mrc_f	mwd_f
	~s_vld	s_vld			
INI	WC	WC	0	0	0
WC	WD	WD	1	0	0
WD	WD	RC	0	0	1
RC	RD	RD	0	1	0
RD	RD	WC	0	0	0

(a) State Table of MSBUS Master

cur_state {q2,q1,q0}	nxt_state {q2+,q1+,q0+}		mwc_f	mrc_f	mwd_f
	~s_vld	s_vld			
000	001	001	0	0	0
001	010	010	1	0	0
010	010	011	0	0	1
011	100	100	0	1	0
100	100	001	0	0	0

INI=3'b000, WC=3'b001, WD=3'b010,
RC=3'b011, RD=3'b100

(b) Transition Table of MSBUS Master

FIGURE 8.26
MSBUS Master State and Transition Table

cur_state	nxt_state		swd_f	srd_f
	~m_ce	m_ce		
INI	WC	WC	0	0
WC	WC	WD	0	0
WD	RC	RC	1	0
RC	RC	RD	0	0
RD	WC	WC	0	1

(a) State Table of MSBUS Slave

cur_state {q2,q1,q0}	nxt_state {q2+,q1+,q0+}		swd_f	srd_f
	~m_ce	m_ce		
000	001	001	0	0
001	001	010	0	0
010	011	011	1	0
011	011	100	0	0
100	001	001	0	1

(b) Transition Table of MSBUS Slave

FIGURE 8.27
MSBUS Slave State and Transition Table

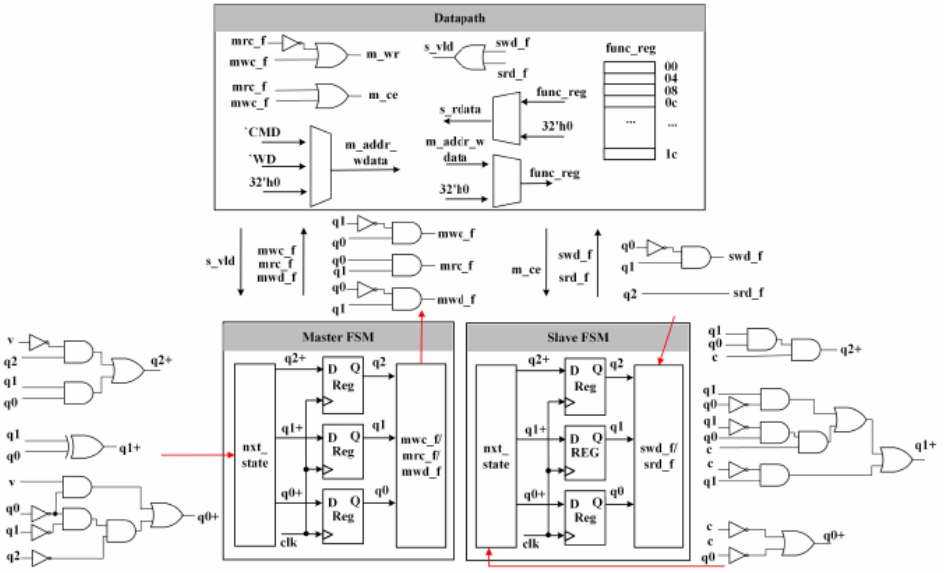


FIGURE 8.28
MSBUS Master-Slave Circuit