delay of "clk_a" before utilizing the signal to de-assert the original request signal "req_a".

Exercises

Please refer to Table 11.2 and Table 11.3 for the timing parameters to be used in the following exercises. It is assumed that the clock skew is always 1.0 ns.

Problem 11.1. Referring to Figure 11.21, please answer the following questions:
1) Is the path "b-e-f-g-i" a true path or a false path? If it is a false path, what is the longest true path?
2) What is the shortest true path?
3) Assuming that the entire combinational circuit is between two register layers: all the signals "a", "b", "c", and "d" are outputs from the first register layer, and the signal "i" is the input to the second register layer. What is the MOF for the entire sequential circuit?
4) Does a hold time violation occur in the circuit?

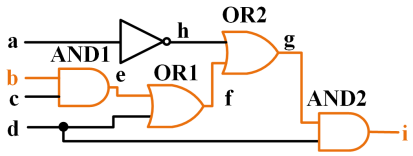

FIGURE 11.21
Design Circuit #1 for True/False Path

Problem 11.2. Referring to Figure 11.22, please answer the following questions:
1) Determine whether the longest path "c-d-e-f-j" is a true path or a false path. If it is a false path, please identify the longest true path.
2) Identify the shortest true path in the circuit.
3) Assuming that the entire combinational circuit is between two register layers: all the signals "a", "b", and "c" are outputs from the first register layer, and the signal "j" is the input to the second register layer. What is the MOF for the entire sequential circuit?
4) Is there a hold time violation?

Problem 11.3. 1) Draw the circuit described with the Verilog code below.
2) Assuming a clock period of 12 ns, identify whether there is a potential

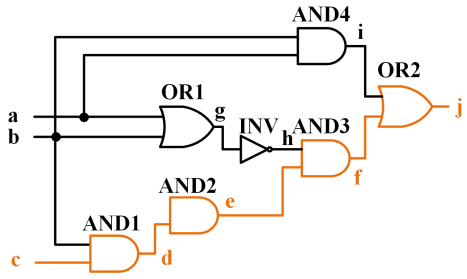

FIGURE 11.22
Design Circuit #2 for True/False Path

setup violation if the logic is synthesized exactly as described. Provide a brief explanation for your answer.
3) Identify whether there is a potential hold violation. Provide a brief explanation for your answer.

```
module (input a, b, c, d, e, f, output reg j);
reg g, h, i;

always @(posedge clk, negedge rst)
if (~rst) begin
    g<=0; h<=0; i<=0; j<=0;
end else begin
    g <= a & b                  ;
    h <= c | d                  ;
    i <= e ^ f                  ;
    j <= i ? ~|{g, h} : &{g, h};
end
endmodule
```

PBL 31: MOF Assessment of FP Numerical Hardware

In Chapter 2, we introduce an analytical approach to approximately estimate the maximum operational frequency (MOF) of register-transfer designs using the following equation:

$$MOF = \frac{1}{T_R - T_{WNS}}. \tag{11.23}$$

In this equation, T_R represents the reference clock period, and T_{WNS} represents the Worst Negative Slack, indicating the available time in each clock cycle.

1) Let's assume that the design specification requires a reference clock frequency of 100 MHz, which corresponds to a minimum clock period of $\frac{1}{100\ \text{MHz}} = 10$ ns. Therefore, we can set T_R to 10 ns. In this scenario, determine the T_{WNS} for the circuit designed in PBL 18 in Chapter 8, which utilizes the single-clock-cycle FP operators. You can achieve this by configuring the constraint file and executing the Vivado synthesis and implementation tools. For detailed configuration information, please refer to Section 2.6.2 in Chapter 2.

2) Follow the same procedure for PBL 19 in Chapter 8, which utilizes the pipeline-designed FP operators.

3) Calculate the MOF for PBL 18 and PBL 19, and provide an explanation of the MOF differences.

4) Consider the performance disparities (latency and bandwidth) among PLB 19, PLB 20, and PLB 21.

PBL 32: Request-Grant Handshaking and CDC

1) Design the handshaking circuit by integrating the master in clock domain A and the slave in clock domain B, following the block diagram depicted in Figure 11.20. The design's IO specifications are outlined in Tables 11.4 and 11.5 below. The "req_a" output from the master serves as the bus request input for the slave, while the "gnt_b" output from the slave facilitates the bus grant to the master.

2) Create a testbench to simulate the circuit and confirm its functionality by generating a pulse signal named "en_a". Monitor the timing behavior of all signals, with special attention to the interaction between the handshaking signals "req_a" and "gnt_b", which constitute the handshaking process.

PBL 33: Synchronous FIFO

1) Design a synchronous FIFO, encompassing the IOs detailed in Table 11.6. Notably, the FIFO incorporates an asynchronous reset signal denoted as "rst", which initializes the FIFO's memory and internal pointers upon activation.

Operating in sync with the clock input labeled "clk", the FIFO facilitates data writing and reading via the "write_en" and "read_en" signals, respectively. Data is written into the FIFO using the "data_in" input, while data retrieval is achieved through the "data_out" output. The FIFO design further encompasses a "full" indicator to signify when the FIFO reaches capacity and can no longer accommodate additional data. Similarly, an "empty" indicator denotes that the FIFO is devoid of data.

TABLE 11.6
Synchronous FIFO IOs Description

Name	Direction	Bit Width	Description
clk	Input	1	Clock, rising edge, 50 MHz
rst	Input	1	Asynchronous reset, 0 valid
write_en	Input	1	FIFO write enable
read_en	Input	1	FIFO read enable
data_in	Input	parameterized	The data bus for FIFO writing
data_out	Output	parameterized	The data bus for FIFO reading
full	Output	1	FIFO full indicator
empty	Output	1	FIFO empty indicator

The following are the design specifications and guiding principles:

- The FIFO design involves a circular buffer structure, realized by utilizing an array of registers, as depicted in line 13 within the provided Verilog code. It allows for a user-configurable depth and width, which can be established using *parameters*, as evident in line 9-10.
- The FIFO module should integrate two internal pointers: a write pointer (referred to as "write_ptr" in line 14) and a read pointer (identified as "read_ptr" in line 15). These pointers will serve to indicate the subsequent locations for data writing and reading, respectively. Upon executing a write operation, the "write_ptr" should increment by one. Similarly, during a read operation, the "read_ptr" should also increment by one. The bit width of these pointers should correspond to the FIFO depth, as indicated in line 11. For instance, in the provided Verilog code example, where the FIFO depth is parameterized as 64, a pointer width of 6 bits is needed.
- Additionally, the FIFO module should include a counter (labeled as "cnt" in line 16), which will keep track of the total number of data items stored within the buffer. Upon each write operation, the counter should increment by one. Conversely, during a read operation, the counter should decrement by one. The state of the counter will dictate the status of the FIFO. When the counter reaches the value of $DEPTH$ the FIFO should be deemed full. Conversely, if the counter is at zero, the FIFO should be considered empty.
- A crucial aspect of the FIFO design is to offer indicators for its fullness or

emptiness. In cases where the counter equals $DEPTH$, the FIFO must not accept further data even if the "write_en" signal is active. Likewise, when the counter is zero, the FIFO should not permit data extraction, even if the "read_en" signal is asserted.

```
module sync_fifo (input                clk      ,
                  input                rst      ,
                  input                write_en ,
                  input                read_en  ,
                  input  [WIDTH-1:0]   data_in  ,
                  output [WIDTH-1:0]   data_out ,
                  output               full     ,
                  output               empty    );
parameter DEPTH     = 64;  // Depth of the FIFO
parameter WIDTH     = 8 ;  // Width of the FIFO
parameter PTR_WIDTH = 6;   // DEPTH = 1<<PTR_WIDTH;

reg [WIDTH-1:0]       mem [0:DEPTH-1];
reg [PTR_WIDTH-1:0]   write_ptr      ;
reg [PTR_WIDTH-1:0]   read_ptr       ;
reg [PTR_WIDTH-1:0]   cnt            ;

endmodule
```

2) Create a testbench to simulate the FIFO design, assessing its functionalities encompassing FIFO write and read operations, along with scenarios where the FIFO is either full or empty.

PBL 34: High-Speed Design on FP Matrix-Matrix Adder

1) Redesign the FP Matrix-Matrix Adder in Section 9.3 in Chapter 9, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 9.11.
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 35: High-Speed Design on FP AXPY Calculation

1) Redesign the FP AXPY Calculation circuit in Section 9.4 in Chapter 9, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 9.13.
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 36: High-Speed Design on FP DDOT

1) Redesign the FP DDOT circuit in Section 9.5 in Chapter 9, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 9.15.
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 37: High-Speed Design on FP Matrix-Matrix Multiplier

1) Redesign the Multiplication-Addition Circuit (MAC) in PBL 25, by leveraging the pipeline-designed FP operators provided in this book.
2) Redesign the FP Matrix-Matrix Multiplier circuit in PBL 26, incorporating the high-speed MAC circuit as the design IP.
3) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input in PBL 26.
4) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 38: High-Speed Streaming Design on DDOT

1) Redesign the Streaming Design on DDOT circuit in Section 10.2 in Chapter 10, by leveraging the pipeline-designed FP operators provided in this book.

2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 10.3.
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 39: High-Speed Iterative Design with Streaming Width Four

1) Redesign the Iterative Design with Streaming Width Four in Section 10.3 in Chapter 10, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 10.8.
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 40: High-Speed Streaming Design on DFT2

1) Redesign the Streaming Design on DFT2 in PBL 29, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input shown in Figure 10.16(b).
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.

PBL 41: High-Speed Streaming Design on DFT4

1) Redesign the Streaming Design on DFT4 in PBL 30, by leveraging the pipeline-designed FP operators provided in this book.
2) Create a testbench to facilitate the simulation of the design and validate its functionality using the same FP data input as shown in Figure 10.18(b).
3) Comments the difference between using single-clock-cycle IPs and pipeline-designed IPs.