EE6205: Hardware Description Language (C-18)

End-Semester Examination - July 2025

Model Answers

Question 1

a) Truth table and circuit diagram

The given RTL model is:

```
module unknown1 (input wire a, input wire b, output wire c);
assign c = (a & b) | (~a & ~b);
endmodule

```

This expression corresponds to the logical XNOR operation ($c = \overline{a \oplus b}$).

Truth Table:

a	b	c
0	0	1
0	1	0
1	0	0
1	1	1

Circuit Diagram:

b) Modified module code

i) Behavioral modeling

```
module unknown1_bhv (
  input wire a,
  input wire b,
  output reg c  );

always @(a or b) begin
if (a == b)  c = 1'b1;
else         c = 1'b0;
end

endmodule

```

Listing 1: Behavioral model for unknown1

ii) Gate-level modeling

```
module unknown1_gate (
input wire a,
input wire b,
output wire c
);
wire not_a, not_b, term1, term2;

// Gate instantiations
not inv_a (not_a, a);
not inv_b (not_b, b);
and and_1 (term1, not_a, not_b);
and and_2 (term2, a, b);
or  or_1  (c, term1, term2);

endmodule

```

Listing 2: Gate-level model for unknown1

c) Testbench analysis

i) Console display

```
a=0, b=0, c=1
a=0, b=1, c=0
a=1, b=1, c=1
a=1, b=0, c=0
a=0, b=0, c=1
```

ii) Waveform output

Question 2

a) ‘initial‘ vs ‘always‘ and ‘wire‘ vs ‘reg‘

i) ‘initial‘ block vs ‘always‘ block

‘initial‘ block executes exactly once at the beginning of a simulation (at time 0). Typically used for initialization, stimulus generation in testbenches, and defining single-shot behaviors. Multiple ‘initial‘ blocks in a module execute concurrently.

‘always‘ block executes throughout the simulation, whenever triggered by events in the sensitivity list. For example, ‘always @(posedge clk)‘ executes on every positive clock edge, while ‘always @(*)‘ executes whenever any of the variables read inside the block change. ‘always‘ blocks are used to model both computational and sequential logic that is always active.

```
module example;
reg clk, reset;

initial begin     // initial block for stimulus (runs once)
clk = 0;
reset = 1;
#10 reset = 0;
end

// always block for clock generation (runs forever)
always #5 clk = ~clk;

endmodule

```

Listing 3: Example of initial and always blocks

ii) ‘wire‘ vs ‘reg‘

A **‘wire‘** is a net data type that represents a physical wire in a circuit. A ‘wire‘ cannot store a value on its own; it must be continuously driven by something, like the output of a gate or an ‘assign‘ statement. It is used to model combinational logic connections.

A **‘reg‘**, is a variable data type that can store a value between assignments. A ‘reg‘ is required for any variable that is a target of an assignment within a procedural block (i.e., ‘initial‘ or ‘always‘ blocks). It can be used to model both combinational and sequential logic.

```
module example2(input a, b, clk, output y_comb, output reg y_seq);

wire w1; // wire for combinational connection

// 'wire' is driven by a continuous assignment
assign w1 = a & b;
assign y_comb = ~w1;

// 'reg' is used for the output of a procedural block
always @(posedge clk) begin
y_seq <= a | b; // y_seq must be a reg
end

endmodule

```

Listing 4: Example of wire and reg

b) Synthesis issues and resolution

The given code snippet is:

```
always @ (posedge clk or in1 or in2) begin
if (in1) q <= 1'b1;
else if (in2) q <= 1'b0;
end

```

This code has two main synthesis issues:

1. **Inferred Latch:** The 'if-else if' structure does not specify what happens to 'q' when neither 'in1' nor 'in2' is true or both 'in1' and 'in2' are true. In a combinational 'always' block, this would imply that 'q' should hold its previous value, leading to the synthesis of an unintended latch. In this mixed-signal block, the behavior is ambiguous but problematic.

2. **Mixed-signal sensitivity list:** The list 'posedge clk or in1 or in2' mixes a clock edge trigger with level-sensitive signals. Synthesis tools generally do not support this for creating standard synchronous logic. It is interpreted as a flip-flop that is sensitive to the clock edge AND asynchronously sensitive to changes in 'in1' and 'in2', which is not standard hardware.

Resolved Code (assuming synchronous logic with active-high enables): The most likely intention is a synchronous flip-flop where 'in1' and 'in2' act as prioritized enables.

```
// Assuming a D-flipflop with prioritized synchronous load signals
always @ (posedge clk) begin
if (in1) begin
q <= 1'b1;
end else if (in2) begin
q <= 1'b0;
end
// Note: To avoid an inferred latch, you might want an else case
// else q <= q; // Explicitly hold value (optional for FFs)
end

```

Listing 5: Corrected synchronous logic

c) 8-bit Full Adder

i) fulladder8b() by modifying fulladder1b()

```
module fulladder_8b (
input  wire [7:0] a,
input  wire [7:0] b,
input  wire       cin, // input carry bit
output wire [7:0] s,   // results
output wire       cout // resulting carry bit
);

// Verilog handles vector arithmetic directly
assign {cout, s} = a + b + cin;

endmodule

```

Listing 6: 8-bit full adder using vector arithmetic

ii) fulladder8bm() by instantiating fulladder1b()

```
module fulladder_8b_m (
input  wire [7:0] a,
input  wire [7:0] b,
input  wire       cin,
output wire [7:0] s,
output wire       cout
);

// Internal wire to chain the carry bits
wire [7:0] carry_chain;

// Instantiate 8 1-bit full adders
fulladder_1b fa0 (.a(a[0]), .b(b[0]), .cin(cin),         .s(s[0]), .cout(carry_chain[0]));
fulladder_1b fa1 (.a(a[1]), .b(b[1]), .cin(carry_chain[0]), .s(s[1]), .cout(carry_chain[1]));
fulladder_1b fa2 (.a(a[2]), .b(b[2]), .cin(carry_chain[1]), .s(s[2]), .cout(carry_chain[2]));
fulladder_1b fa3 (.a(a[3]), .b(b[3]), .cin(carry_chain[2]), .s(s[3]), .cout(carry_chain[3]));
fulladder_1b fa4 (.a(a[4]), .b(b[4]), .cin(carry_chain[3]), .s(s[4]), .cout(carry_chain[4]));
fulladder_1b fa5 (.a(a[5]), .b(b[5]), .cin(carry_chain[4]), .s(s[5]), .cout(carry_chain[5]));
fulladder_1b fa6 (.a(a[6]), .b(b[6]), .cin(carry_chain[5]), .s(s[6]), .cout(carry_chain[6]));
fulladder_1b fa7 (.a(a[7]), .b(b[7]), .cin(carry_chain[6]), .s(s[7]), .cout(cout)); // Final cout

endmodule

// The 1-bit full adder module is assumed to be available
module fulladder_1b (
input wire a, b, cin,
output wire s, cout
);
assign {cout, s} = a + b + cin;
endmodule

```

Listing 7: 8-bit full adder using 1-bit instances

Question 3

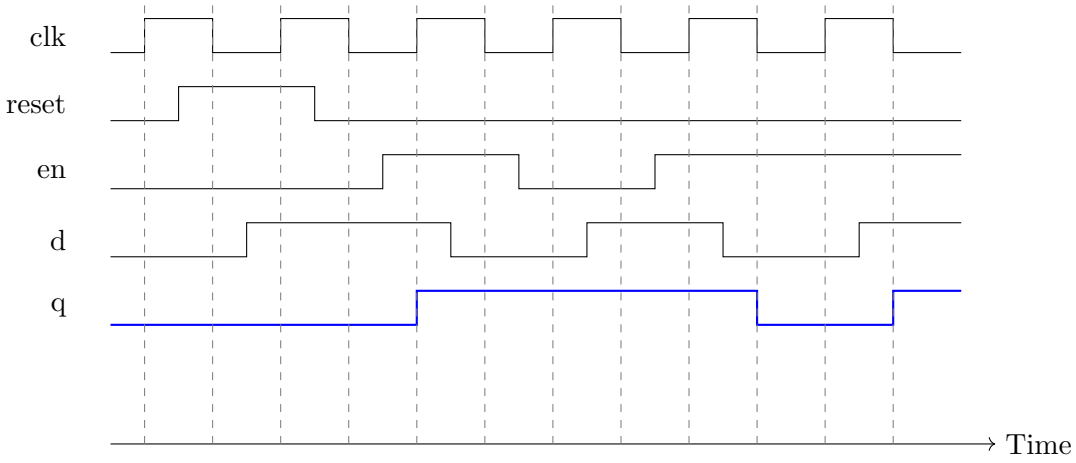
a) D-Flipflop analysis

i) D-Flipflop function and timing diagrams

The module implements a positive-edge triggered D-Flipflop with a synchronous, active-high reset (‘reset‘) and a synchronous, active-high enable (‘en‘). The reset has higher priority than the enable.

- On the positive edge of ‘clk‘, if ‘reset‘ is high, the output ‘q‘ is synchronously reset to 0.
- If ‘reset‘ is low, and ‘en‘ is high, ‘q‘ takes the value of the input ‘d‘.
- If both ‘reset‘ and ‘en‘ are low, ‘q‘ holds its previous value.

Timing Diagram:



ii) Testbench tbdff()

```
1   `timescale 1ns/10ps
2
3   module tb_dff();
4   // Inputs to DUT
5   reg d, clk, en, reset;
6   // Output from DUT
7   wire q;
8
9   // Clock period definition
10  parameter CLK_PERIOD = 10;
11
12  // Instantiate the Device Under Test (DUT)
13  dflipflop uut (
14  .d(d), .clk(clk), .en(en), .reset(reset), .q(q)
15  );
16
17  // Clock generation block
18  initial clk = 1'b0;
19  always #(CLK_PERIOD/2) clk = ~clk;
20
21  // Stimulus generation block
22  initial begin
23  // 1. Initialize and test reset
24  $display("T=%0t: Initializing signals", $time);
25  reset = 1'b1; en = 1'b0; d = 1'b0;
```

```
    #20; // Wait for a couple of clock cycles

    // 2. De-assert reset, test enable=0 (hold state)
    $display("T=%0t: De-assert reset, en=0", $time);
    reset = 1'b0;
    d = 1'b1;
    #20;

    // 3. Test enable=1 (data propagation)
    $display("T=%0t: Assert enable, en=1", $time);
    en = 1'b1;
    #10;
    d = 1'b0;
    #10;
    d = 1'b1;
    #10;

    // 4. Test reset priority over enable
    $display("T=%0t: Assert reset again", $time);
    reset = 1'b1;
    #20;

    $display("T=%0t: Test finished", $time);
    $finish;
    end

    // Monitor changes
    initial begin
    $monitor("T=%0t | reset=%b en=%b d=%b | q=%b", $time, reset, en, d, q);
    end

    endmodule

```

Listing 8: Testbench for the D-Flipflop

iii) Asynchronous reset modification

To make the reset asynchronous, it must be added to the sensitivity list as an edge-triggered event (e.g., 'posedge reset'). The logic inside the 'always' block remains the same.

```
    always @ (posedge clk or posedge reset) begin
    if (reset)
    q <= 1'b0; // This now happens immediately on posedge reset
    else if (en)
    q <= d;
    end

```

Listing 9: D-Flipflop with asynchronous reset

b) LED blinker circuit (1 Hz)

To blink an LED every second from a 50 MHz clock, we need to count 50,000,000 clock cycles.

```
    module led_blinker (
    input wire clk_50mhz,
    input wire reset,
    output reg led_out
    );
    // Counter to divide the clock
    // Needs to count up to 49,999,999 (50 million cycles)
    // 2^26 is > 50,000,000, so we need a 26-bit counter.
```

```
    reg [25:0] counter;

    // The value to count up to
    parameter MAX_COUNT = 50_000_000 - 1;

    always @(posedge clk_50mhz or posedge reset) begin
    if (reset) begin
    counter <= 0;
    led_out <= 0;
    end else begin
    if (counter == MAX_COUNT) begin
    counter <= 0;
    led_out <= ~led_out; // Toggle LED every second
    end else begin
    counter <= counter + 1;
    end
    end
    end
    endmodule

```

Listing 10: 1 Hz LED blinker from 50 MHz clock

c) ‘bus()‘ module for Figure Q3(c)

The circuit diagram shows the following connections:

- P[5] = A[0]
- P[4] = A[1] & B[3]
- P[3] = A[3]
- P[2] = A[2] & B[2]
- P[1] = B[1]
- P[0] = B[0]

```
    module bus (
    input  wire [3:0] A,
    input  wire [3:0] B,
    output wire [5:0] P
    );
    // Using continuous assignments based on the diagram
    assign P[0] = B[0];
    assign P[1] = B[1];
    assign P[2] = A[2] & B[2];
    assign P[3] = A[3];
    assign P[4] = A[1] & B[3];
    assign P[5] = A[0];

    endmodule

```

Listing 11: Module for the bus breakout circuit

Question 4

a) Timing delays and logic values

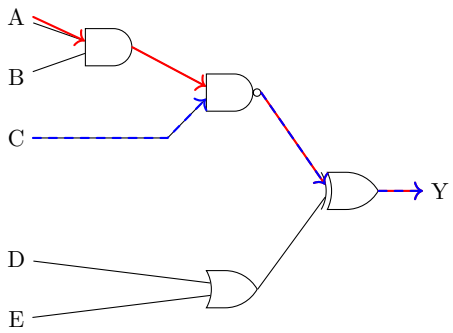
i) Propagation (t_{pd}) and contamination (t_{cd}) delay

- **Contamination Delay (t_{cd}):** The minimum time from when an input changes until the output *begins* to change. Before t_{cd}, the output is guaranteed to hold its previous value.
- **Propagation Delay (t_{pd}):** The maximum time from when an input changes until the output is guaranteed to have settled to its new, stable value.

ii) Circuit delay calculation

Note: The question states $t_{pd} = 60ps$ and $t_{cd} = 100ps$. This is physically impossible as t_{pd} must be greater than or equal to t_{cd}. We will assume a typo and use the more conventional values of $t_{pd} = 100ps$ and $t_{cd} = 60ps$.

Circuit with paths highlighted:



- **Critical Path (Longest Delay Path):** The path from inputs A or B through gates G1, G2, and G4 to the output Y. This path traverses 3 gates.
  - Total Propagation Delay (t_{pd}) $= 3 \times t_{pd,gate} = 3 \times 100ps =$ **300ps**.
- **Shortest Path (Shortest Delay Path):** The paths from C (through G2, G4) or D/E (through G3, G4). These paths traverse 2 gates.
  - Total Contamination Delay (t_{cd}) $= 2 \times t_{cd,gate} = 2 \times 60ps =$ **120ps**.

iii) 'X' and 'Z' values

- **X - The Illegal Value:** In simulation, 'X' represents an unknown or uninitialized logic value. At the circuit level, it represents a condition of contention, where two or more drivers are attempting to drive a single wire to opposite logic levels (e.g., one driver

outputs '1' and another outputs '0' onto the same wire). This can cause a short circuit and an indeterminate voltage level.

- **Z - The Floating Value:** Represents a high-impedance state. At the circuit level, this means a wire is not being actively driven to either '1' or '0'. It is effectively disconnected. This state is essential for implementing shared buses, where multiple devices can be connected to the same wire, but only one is allowed to drive the bus at any given time (using tri-state buffers).

b) FPGA architecture

i) Configurable Logic Block (CLB)

A Configurable Logic Block (CLB) is the fundamental logic resource in an FPGA. A typical CLB consists of a Look-Up Table (LUT), a D-Flipflop (DFF), and multiplexers (MUX) for routing.

Block Diagram:

Functionality:

- **Look-Up Table (LUT):** A small SRAM that can be programmed to implement any Boolean function of its inputs (typically 4 to 6 inputs). It provides the CLB's combinational logic capability.
- **D-Flipflop (DFF):** A storage element that registers the LUT's output on a clock edge. This allows the CLB to implement sequential logic.
- **Multiplexer (MUX):** Selects the output of the CLB. It can route either the direct combinational output from the LUT or the registered sequential output from the D-Flipflop.

ii) Other essential FPGA building blocks

1. **Programmable Interconnect:** A network of routing channels and programmable switches that connect the CLBs, I/O blocks, and other elements together.
2. **I/O Blocks (IOBs):** Located at the periphery of the FPGA chip, these blocks connect the internal logic to the external pins of the device and can be configured for various voltage standards.
3. **Block RAM (BRAM):** Dedicated blocks of on-chip static memory, providing large and fast storage resources.
4. **DSP Slices:** Specialized blocks containing hardware multipliers and accumulators, designed to efficiently implement digital signal processing algorithms.

Question 5

a) State transition diagram for sequence ‘1101‘ (Mealy, Overlapping)

The FSM detects the sequence ‘1101‘. As a Mealy machine, the output depends on the current state and the current input. Overlap is allowed (e.g., in ‘1101101‘, two sequences are detected).

States:

- **S0:** Idle/Reset state. No part of the sequence has been detected.
- **S1:** The last input was ‘1‘.
- **S2:** The last two inputs were ‘11‘.
- **S3:** The last three inputs were ‘110‘.

b) Verilog code for the FSM

```
module sequence_detector (
input  wire clk,
input  wire reset, // Active-high asynchronous reset
input  wire in,
output reg  out
);

// Define states using parameters
parameter S0 = 2'b00; // Idle state
parameter S1 = 2'b01; // Seen '1'
parameter S2 = 2'b10; // Seen '11'
parameter S3 = 2'b11; // Seen '110'

// State registers
reg [1:0] current_state, next_state;

// Sequential logic for state transitions
always @(posedge clk or posedge reset) begin
if (reset) begin
current_state <= S0;
end else begin
current_state <= next_state;
end
end

// Combinational logic for next state determination
always @(current_state or in) begin
case (current_state)
S0: next_state = in ? S1 : S0;
S1: next_state = in ? S2 : S0;
S2: next_state = in ? S2 : S3;
S3: next_state = in ? S1 : S0;
default: next_state = S0;
endcase
end

// Combinational logic for Mealy output
always @(current_state or in) begin
if ((current_state == S3) && (in == 1'b1)) begin
out = 1'b1;
end else begin
out = 1'b0;
end
end
// A more concise way for the output:
// assign out = (current_state == S3) && in;

endmodule

```

Listing 12: Verilog code for 1101 sequence detector