Problem 1: Verilog in All Flavors

In this question, Kirby has a cake vending machine and wants to eat its preferred flavor. It only knows the index of the cake it prefers, key, and needs your help to write the rest of the logic of the vending machine! The module interface declaration for the vending machine is given below:

```verilog
module cake_select(
    input [1:0] key,
    output [7:0] cake
);
// assume these values are properly initialized
reg[7:0] cake_0, cake_1, cake_2, cake_3;
```

(i) **Case Verilog.** You are tasked to write the module body in a `case` statement.

```verilog
reg [7:0] cake_selected;
always @(___.a___) begin
  case (___.b___)
    2'b00: ___(c)___ = cake_0;
    2'b01: ___(c)___ = cake_1;
    2'b10: ___(c)___ = cake_2;
    2'b11: ___(c)___ = cake_3;
    ___(d)___: ___(c)___ = cake_0;
    ___(e)___
  endcase
  assign ___(f)___
end
```

(a) *
(b) key
(c) cake_selected
(d) default
(e) endcase
(f) cake = cake_selected
(g) end
(ii) **Ternary Verilog.** You are tasked to write the module body using a single line of ternary operation.

```verilog
assign cake = ___(a)___
```

(a) `key == 2'b00 ? cake_0 : key == 2'b01 ? cake_1 : key == 2'b10 ? cake_2 : key == 2'b11 ? cake_3 : cake_0`

(iii) **Structural Verilog.** Meta Knight has one magical 4-in-1 Mux luckily. Its module interface definition is given below:

```verilog
module Mux_four(
    input [1:0] sel,
    input [7:0] data_in_0,
    input [7:0] data_in_1,
    input [7:0] data_in_2,
    input [7:0] data_in_3,
    output [7:0] data_out,
);
```

Please write the module body using this `Mux_four` module provided.

```verilog
___(a)___
```

```verilog
Mux_four cake_mux(
    .sel(key),
    .data_in_0(cake_0),
    .data_in_1(cake_1),
    .data_in_2(cake_2),
    .data_in_3(cake_3),
    .data_out(cake)
);
```

(iv) **More Structural Verilog.** Sadly, Meta Knight is out of town, and we have to use King Dee Dee Dee’s 2-in-1 Mux instead. He has plenty of them! Its module interface definition is given below:

```verilog
module Mux_two(
    input sel,
    input [7:0] data_in_0,
    input [7:0] data_in_1,
    output [7:0] data_out,
);
```

Please write the module body using this `Mux_two` module provided.

```verilog
wire[7:0] tmp_0, tmp1;
___(a)___
___(b)___
___(c)___
Mux_two cake_mux_a(
  .sel(key[0]),
  .data_in_0(cake_0),
  .data_in_1(cake_1),
  .data_out(tmp_0)
);

Mux_two cake_mux_b(
  .sel(key[0]),
  .data_in_0(cake_2),
  .data_in_1(cake_3),
  .data_out(tmp_1)
);

Mux_two cake_mux_c(
  .sel(key[1]),
  .data_in_0(tmp_0),
  .data_in_1(tmp_1),
  .data_out(cake)
);
Problem 2: Edge Detection

In this question, you will explore the design of an edge detector. It has three inputs. \texttt{clk}, the clock signal that orchestrates all the synchronous operations of this module. \texttt{rst}, a reset signal that resets the edge detector to a known state. This reset is a positive, synchronous reset. \texttt{test}, a slow one-bit signal that is guaranteed to change at most once per cycle and is never high-z or x after reset is deasserted. It has two outputs, \texttt{rise} and \texttt{fall}, which should assert true for one cycle only if a rising/falling edge is present in the past cycle.

(i) Do I work? Your friend Kirby has written the following Verilog module. Please carefully review the code below. Does it work as the spec above specified? If so, please briefly explain how this module works. If it doesn’t, please propose a fix.

```verilog
module edge_detect(
    input clk,
    input rst,
    input test,

    output reg rise,
    output reg fall
);

reg test_prev;
wire rise_next, fall_next;

assign rise_next = test_prev & test;
assign fall_next = ~test_prev & ~test;

always @(posedge clk) begin
    if (rst) begin
        test_prev <= 1'b0;
        rise <= 1'b0;
        fall <= 1'b0;
    end
    else begin
        test_prev <= test;
        rise <= rise_next;
        fall <= fall_next;
    end
end

endmodule

NO, It doesn’t work. Fix:

```verilog
assign rise_next = ~test_prev & test;
assign fall_next = test_prev & ~test;
```
(ii) Testbenches. Regardless of your answer above, your other friends Waddle-Dees are not convinced. "You're not a testbench after all!", they said. To prove his point, they write a testbench for `edge_detect`. Please carefully read their code below and answer the following questions.

```verilog
`timescale 1ns/1ns

module edge_detect_tb;

    // Define parameters
    parameter TIME_UNIT = 1;
    parameter CLK_PERIOD = 10 * TIME_UNIT;
    parameter DELAY = 1 * TIME_UNIT;
    parameter REST = CLK_PERIOD - DELAY;

    // Define the signals for dut
    reg clk, rst, test;
    wire rise, fall;

    // Define the signals for tb
    reg expected_rise, expected_fall, expected_rise_next;
    reg check_rise, check_fall;
    reg toggle, toggle_next;

    // Instantiate the design under test module
    edge_detect dut (    .clk(clk),
        .rst(rst),
        .test(test),
        .rise(rise),
        .fall(fall)
    );

    initial begin
        // Open the VCD waveform file
        $dumpfile("waveform.vcd");
        // Dump variables to the waveform file
        $dumpvars(0, edge_detect_tb);
        // Signal initialization
        rst = 1'b0;
        test = 1'b0;
        toggle_next = 1'b0;
        // Apply reset and wait for a cycle
        rst = 1'b1;
        #CLK_PERIOD;
        rst = 1'b0;
    end
```

Please carefully read their code below and answer the following questions.
// Apply random stimulus
repeat (100) begin
    #(urandom_range(1, 3) * CLK_PERIOD + DELAY); // Wait for a random time
    test = ~test;
    toggle_next = 1'b1;
    #REST;
    toggle_next = 1'b0;
end
$display("Simulation completed!");
$finish;
end

// Toggle the clock signal
initial clk = 1'b0;
always #(CLK_PERIOD / 2) clk = ~clk;

always @(posedge clk) begin
    if (rst) begin
        toggle <= 1'b0;
        expected_rise <= 1'b1;
    end
    else begin
        toggle <= toggle_next;
        expected_rise <= expected_rise_next;
    end
end

always @(*) begin
    expected_rise_next = toggle ? ~expected_rise : expected_rise;
    expected_fall = ~expected_rise;
    check_rise = expected_rise & toggle;
    check_fall = expected_fall & toggle;
    if (check_rise & ~rise)
        $display("Error: Rising edge not detected at time %t", $time);
    if (check_fall & ~fall)
        $display("Error: Falling edge not detected at time %t", $time);
end
endmodule

(a) Does this testbench fully cover the spec? If so, please briefly explain how it works. If not, please propose a fix.

NO, It doesn’t cover everything. It also needs to check fall and rise is not triggered on other occasions, either on the opposing edge or on a non-changing step. Otherwise, an always-high rise and fall could also pass this testbench.

(b) Imagine Waddle-Doo has a magic synthesis tool that synthesizes all synthesizable constructs as-is, and synthesizes all non-synthesizable constructs as an integrated black-box. (i.e. assume non-synthesizable constructs do not block the synthesis of other cells.) For
each of the following logic, answer what will they be synthesized into: \{wire, register, non-synthesizable blackbox, other\}.

\texttt{clk, expected\_rise, expected\_fall, toggle\_next, toggle, test, rise, fall, check\_rise, check\_fall}

\texttt{clk: non-synthesizable; expected\_rise: register; expected\_fall: wire; toggle\_next: non-synthesizable; toggle: register; test: non-synthesizable; rise: wire; fall: wire; check\_rise: wire; check\_fall: wire.}
Problem 3: Shake My Hand

Handshakes are important in the world of digital design, and one of the good ways of implementing the interface-level handshake is a ready-valid (aka decoupled) interface. Please read this supplemental material to understand the ready-valid interface.

(i) **Ready-Valid Interface Boolean Potpourri.** Based on the reading, please identify whether the following statements about the ready-valid interface is true or false.

(a) The data transfer happens at the same cycle where ready and valid are both asserted high.

(b) It is good practice to internalize the ready and valid driving logics inside the module body.

(c) The ready and valid signals are sampled at the rising edge of the clock only, and any metastability during the clock period is acceptable.

(d) All module interfaces should be ready-valid interface for cleanliness of the design.

(e) You’re designing an IP block module that will be deployed across the system and interact with countless, unknown modules. It is acceptable to derive the valid signal using the ready signal from the same interface.

(a) false

(b) true

(c) true

(d) false

(e) false

(ii) **Depth 1 FIFO.** Now, let’s design a very simple module using the ready-valid interface we learned above. We want to implement a depth 1 first-in-first-out (FIFO) buffer. Since it holds a maximum of one data, ordering needs not to be considered. We can use a register as its internal buffer storage.

```vhdl
module fifo(
    input rst,
    input clk,

    input enq_valid,
    input [31:0] enq_bits,
    output enq_ready,

    output deq_valid,
    output [31:0] deq_bits,
    input deq_ready
    );

    reg [31:0] buffer;
```
reg full;

wire enq_fire, deq_fire;
assign enq_fire = ___(a)___;
assign deq_fire = deq_valid & ___(b)___;

assign enq_ready = ___(c)___;
assign deq_valid = ___(d)___;
assign deq_bits = buffer;

always @(posedge clk) begin
  if (rst) begin
    buffer <= 32'b0;
    full <= 1'b0;
  end
  else begin
    if (enq_fire) begin
      buffer <= ___(e)___;
      full <= ___(f)___;
    end
    if (deq_fire) begin
      full <= ___(g)___;
    end
  end
endmodule

(a) enq_valid & enq_ready
(b) deq_ready
(c) ~full
(d) full
(e) enq_bits
(f) 1'b1
(g) 1'b0

(h) What is the latency from transmitting a data packet from the enq interface to the deq interface in terms of cycle(s), assuming deq_ready and enq_valid are constantly high?
2 cycles, as it goes through

(iii) Optional. This question is optional and will not count towards your grade. While the above fifo is great, Waddle Dee wonders if we could do even better than that. Please design and implement a digital circuit module that uses the same input/output declaration as above but transmits the data packet faster from the enq interface to the deq interface when deq_ready and enq_valid are constantly high. Please also design and implement a testbench to verify its functionality and performance.

A few tips:
• You can use EDA Playground to test your design. The left panel is your testbench and the right panel is your design. You can set Synopsys VCS as simulator, and check Open EPWave after run for waveform debugging.

• Your testbench should cover these cases: normal go-through, where both deq_ready and enq_valid are constantly high; normal blocking, where deq_ready is low: after one enqueue, enq_ready should be deasserted; an off-by one go through, where the buffer is warmed up with one data packet, and then both deq_ready and enq_valid are constantly high. Notice that this is still a FIFO, so the ordering of enqueue should be preserved on the dequeue interface.

• Drawing block designs of your circuit might help you envision how you should implement your design.

• You can look up skid buffers if you’re interested in learning more.
Problem 4: Chiplet Economics

A chiplet is a tiny integrated circuit (IC) that contains a well-defined subset of functionality. It is designed to be combined with other chiplets on an interposer in a single package, to form a full system-on-chip. Among the many considerations, one key driving advantage for chiplet technology is die yield in an era of increasing chip area. In the question, you will explore how chiplets can save yield and cost.

Your digital system has a total area $A_T$. You would like to evaluate whether or not splitting your system across multiple chiplets can reduce manufacturing costs. Assume that:

- Total cost is proportional to the silicon area you purchase.
- Each chiplet must have a PHY to talk to other chiplets. This interface incurs a constant area cost $A_P$.
- The yield of a chip/chiplet with area $A$, where $A$ is measured in $mm^2$, is $Y_A$.
- You can split your area $A_T$ uniformly across any number of chiplets, and that the PHY area is the only source of overhead.

If $A = 10mm^2$, $A_P = 0.5mm^2$, and $Y = 0.9$, find the number of chiplets that minimizes cost. Then compare the cost of your chiplet system versus the cost of a monolithic chip.

Solution:

If you split into $N$ chiplets, each chiplet has area $A_C = A_T/N + A_P$. You need $N$ chiplets to assemble the system, but due to yield, you need to purchase more chiplets so that you will have $N$ chiplets after you account for loss due to yield. To end up with $N$ chiplets, you must purchase $N_C = \frac{N}{Y_A} + A_P$ chiplets. The total area purchased is then

$$ Cost(N) = A_C N_C = \frac{(A_T/N + A_P)N}{Y_A A_T/N + A_P}. $$

Substituting the numbers provided and trying a few values of $N$, we see that the cost is minimized when $N = 5$, giving $Cost(N) = 16.2669$.

Now we compare chiplets to a monolithic implementation. The total cost for a single-chip implementation is

$$ Cost = \frac{A_T}{Y_A A_T} = 28.6797, $$

which is more than the chiplet solution. Thus, splitting into 5 chiplets is optimal for cost.