Australian National University | ANU College of ENGINEERING, COMPUTING AND CYBERNETICS

ENGN 4213/6213 Digital Systems and Microprocessors

Semester 1, 2023

Tutorial 2 – Worked Solutions

Question 1:

a. Explain and compare **Structural** vs **Behavioral** (also sometimes called *procedural*) design procedure in Verilog with the help of Verilog `assign` statement and `always` block.

Structural Design

- Involves declaration of component (e.g. gates) and describing their interconnections (wires). It is just considered as a verbal wiring diagram
- Used mainly for combinational logic design
- Example:

```
assign a = b & c | d;
```

A bad example:

```
assign a = ~a;
//You need to avoid feedback when using assign!
```

Behavioral Design

- Abstract models of the function and sequence
- Describes what a module does through procedural code, and they are sequential (code order matters)
- Example:

```
always @ (condition)
      begin
            //desired behavior
      end
```

b. Use `assign` statement and `always` block to design a 4 to 16 decoder. Explain the advantage and disadvantage of the two designs.

If you use `assign`, you will need to describe the logic gates required to implement the decoder's truth table. This involves writing a Boolean equation for each of the 16 bit of the output, which will be very time consuming.

However, if you use `always` block, the design is much more straight forward:

```
module decoder_always(
input wire [3:0] a,
output reg [15:0] y);

    always @(a) begin
        case (a)
        // a 4:16 decoder
```

```
            4'b0000: y = 16'b0000000000000001;
            4'b0001: y = 16'b0000000000000010;
            4'b0010: y = 16'b0000000000000100;
            4'b0011: y = 16'b0000000000001000;
            4'b0100: y = 16'b0000000000010000;
            4'b0101: y = 16'b0000000000100000;
            4'b0110: y = 16'b0000000001000000;
            4'b0111: y = 16'b0000000010000000;
            4'b1000: y = 16'b0000000100000000;
            4'b1001: y = 16'b0000001000000000;
            4'b1010: y = 16'b0000010000000000;
            4'b1011: y = 16'b0000100000000000;
            4'b1100: y = 16'b0001000000000000;
            4'b1101: y = 16'b0010000000000000;
            4'b1110: y = 16'b0100000000000000;
            4'b1111: y = 16'b1000000000000000;
            default: y = 16'b0000000000000000;
        endcase
    end
endmodule
```

Question 2:

a. Describe the functionality of a blocking and non-blocking assignment in a Verilog `always` block.

Blocking assignment ("="): The blocking assignment operator is an equal sign ("="). A blocking assignment gets its name because a blocking assignment must evaluate the RHS arguments and complete the assignment without interruption from any other Verilog statement. The assignment is said to "block" other assignments until the current assignment has been completed. The one exception is a blocking assignment with timing delays on the RHS of the blocking operator, which is considered to be a poor coding style.

Execution of blocking assignments can be viewed as a one-step process:

1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

Non – Blocking assignment ("<="): The non-blocking assignment operator is the same as the less-than-or-equal-to operator ("<="). A non-blocking assignment gets its name because the assignment evaluates the RHS expression of a nonblocking statement at the beginning of a time step and schedules the LHS update to take place at the end of the time step. Between evaluation of the RHS expression and update of the LHS expression, other Verilog statements can be evaluated and updated, and the RHS expression of other Verilog non-blocking assignments can also be evaluated and LHS updates scheduled.

The non-blocking assignment does not block other Verilog statements from being evaluated. Execution of non-blocking assignments can be viewed as a two-step process:

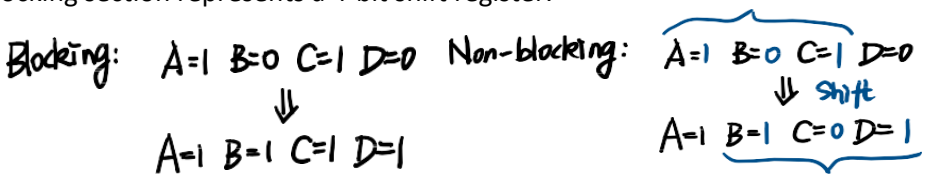
1. Evaluate the RHS of non-blocking statements at the beginning of the timestep.
2. Update the LHS of non-blocking statements at the end of the timestep.

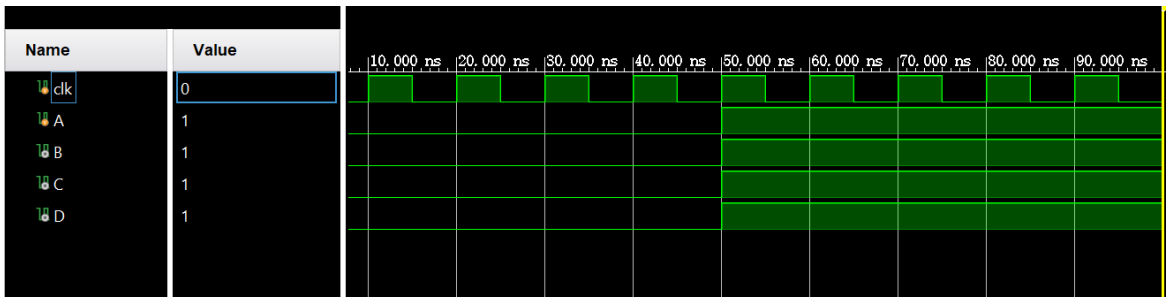
b. Explain the functionality of the following two sets of Verilog codes for a module with input signal `wire A` and output signal `reg D`:

Blocking	Non-blocking

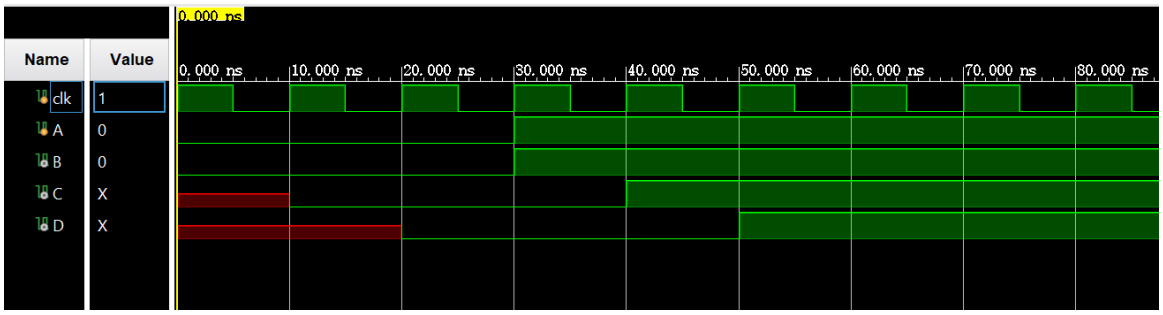
`always @(posedge clk)`
`// block triggered on the rising edge of clock`
 	
`begin`
`B = A;`
`C = B;`
`D = C;`
`end`
	`always @(posedge clk)`
	`// block triggered on the rising edge of clock`
	
	`begin`
	`B <= A;`
	`C <= B;`
	`D <= C;`
	`end`

Blocking section only set D = A. B and C are identical copies of D and, therefore, are meaningless signals in the context of the circuit.
The non-blocking section represents a 4-bit shift register.





Vivado Simulation output for blocking assignment



Vivado Simulation output for non-blocking assignment

(Extra task: If you wrote the module with the given non-blocking code, the C and D output has value 'X' (unknown value) for the first 10 or 20 ns shown above as red lines. Why do you think this

happened? If you are unsure, feel free to ask the tutor)

Question 3:

Use Verilog to design the following hardware:

a. 4 to 1 multiplexer

```
module mux (
input wire [3:0] a,
input wire [1:0] sel,
output reg y);
    always @(*) begin
        case (sel)
            2'b00: y = a[0];
            2'b01: y = a[1];
            2'b10: y = a[2];
            2'b11: y = a[3];
         endcase
        end
endmodule
```

b. 4-bit SIPO register

```
module SIPO (
input wire a,
input wire clk,
output reg [3:0] y);
    always @(posedge clk) begin
        y[0] <= a;
        y[1] <= y[0];
        y[2] <= y[1];
        y[3] <= y[2];
    end
endmodule
```

c. A parallel bank of four D flip-flops with an additional input to disable the output.

```
module ff (
input wire [3:0] a,
input wire clk,
input wire en,
output wire [3:0] out);

reg [3:0] y;
    always @(posedge clk) begin
        y[0] <= a[0];
        y[1] <= a[1];
        y[2] <= a[2];
        y[3] <= a[3];
    end
assign out = y & en;
```

```
endmodule
```

Alternative:

```
module ff (
input wire [3:0] a,
input wire clk,
input wire en,
output reg [3:0] y);
    always @(posedge clk) begin
        if (en == 1) begin
            y[0] <= a[0];
            y[1] <= a[1];
            y[2] <= a[2];
            y[3] <= a[3];
        end
        else
            y <= 4'b0000;
    end
endmodule
```

d. A clock-divider implementing $1/(2^4)$ division.

```
module cd (
input wire clk_in,
output reg clk_out);

reg [3:0] counter = 4'b0000;

    always @(posedge clk_in) begin
        counter <= counter + 4'b0001;
        if (counter == 4'b1111)
            clk_out = ~clk_out;
    end
endmodule
```