

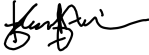
Faculty: FACULTY OF ELECTRICAL ENGINEERING	
Subject : ELECTRICAL ENGINEERING LABORATORY	Review : Release Date : 5 March 2015 Last Amendment : 27 March 2022 Procedure Number :
Subject Code : SKEL 3742	



**FAKULTI KEJURUTERAAN ELEKTRIK
UNIVERSITI TEKNOLOGI MALAYSIA
KAMPUS SKUDAI
JOHOR**

**SKEL 3742
VLSI SYSTEM DESIGN LAB**

Lab Project 2: Traffic Light System

Prepared by : 1. Izam Bin Kamisian 2. Ismahani Ismail Signature & Stamp :  Izam Bin Kamisian PAM VLSI System Design Lab Date: 27 March 2022	Approved by: Name: Prof. Ir. Dr. Rubita binti Sudirman  Signature & Stamp : PROF. IR. DR. RUBITA SUDIRMAN <i>Director (Electronics & Computer Engineering)</i> School of Electrical Engineering Faculty of Engineering Universiti Teknologi Malaysia 81310 UTM Johor Bahru Johor, MALAYSIA Date: 27 March 2022
---	--

Project Introduction

A traffic light at a junction between the main road and a small road is controlled by two motion sensors (S_1 and S_2) as shown in Figure 1 below. The traffic light system has specifications as below: -

- 1) If both sensors are activated at the same time, the sensor with higher priority will take the first turn to green.
- 2) The period for green light for the main road (t_{G-main}) is longer than the small road ($t_{G-small}$).
- 3) For all junctions, the red light must turn on for a certain period (t_R) before the green light for a particular junction can be turned on.
- 4) If the main road in green light and the sensors S_1 and S_2 detected no vehicle at the small road, the light at the main road will remain green.
- 5) The light for main road will turn from green to yellow when the period for green light is over.
- 6) The light will turn from yellow to red when the period for yellow light is over.
- 7) The light at the small road will turn from red to green if: -
 - a. The sensor detected there are vehicle at the small road **AND**
 - b. The period for all lights in red is over.
- 8) The light at the small road will turn from green to yellow if: -
 - a. The sensor detected no vehicle even though the period for green for the small road is still available **OR**
 - b. The period for the green light for the small road is over.
- 9) The light at any junction will turn yellow for a certain period (t_Y) before turning red.
- 10) The period for yellow light is longer than that for red.

The traffic light system is to be designed and implemented but several things need to be determined such as the complete traffic sequence, the timing requirement ($t_{G-main} > t_{G-small} > t_Y > t_R$) and the sensors' priority.

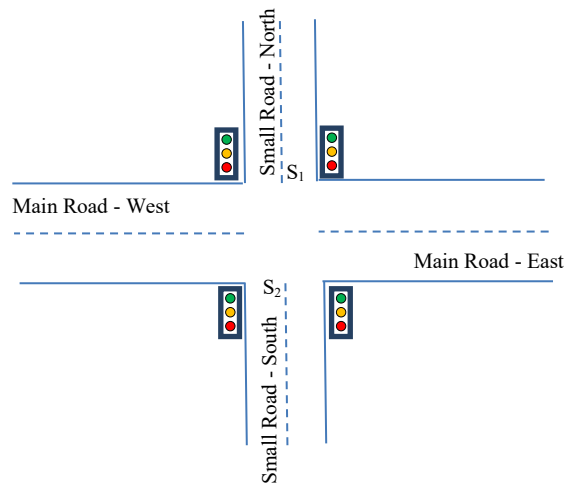


Figure 1: Traffic Light System

The objectives of this project are:

1. To design the traffic light system using a combinational and sequential logic design based on project specifications.
2. To draw the combinational and sequential logic circuit of the traffic light system using schematic design entry/hardware design language (HDL) coding and verify its function using Altera Quartus II Tools software.
3. To prototype and demonstrate the design of the system using Altera FPGA DE2 board.

Project Task

Design the traffic light system using Quartus II schematic entry/HDL coding and FPGA development board. A report with the well-presented design steps, simulation results, and FPGA prototyping methodology is expected to be produced at the end of the project after the third session of the lab.

Some design guidelines:

1. Design a circuit for the traffic light using multiplexers, counters, adders and similar logic blocks.
2. For your design, use appropriate gates, components and mega functions from the Quartus II system library.
3. Use counter to do count-down or count-up for the traffic light change.
4. Use registers or flip-flops for storage.
5. The motion sensor inputs will eventually be connected to the toggle switches on the DE2 board.
6. The countdown for the traffic light will eventually be displayed on the 7-segment displays on the DE2 board.

Identify the input and output of your system, encoding method, circuit topology and suitable algorithm to be used. Design the system using combinational and sequential logic and verify the design using Quartus II CAD Tool. Final task is to prototype and demonstrate your system using FPGA DE2 board. Refer to Quartus II and FPGA tutorial to help you to implement this project.

DPU + CU Design Example

1.1 Procedure

Figure 1 shows the general procedure for designing a complex digital system using the DPU (Datapath Unit) and CU (Control Unit) design approach.

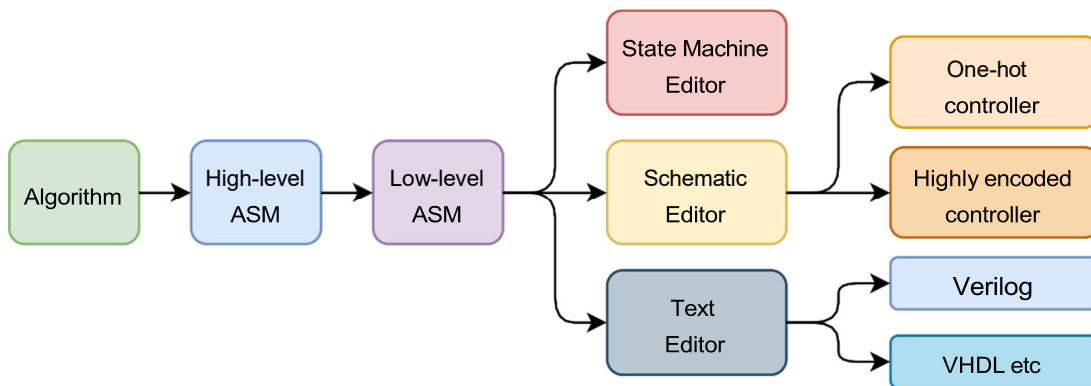


Figure 1: Procedure for complex digital system (DPU + CU design approach)

- **Algorithm**

The problem is first described at a high level, typically as a C code fragment or pseudo-code

- **High-level ASM (RTL-Asmchart)**

The algorithm is then converted into an ASM (Algorithmic State Machine) chart that closely follows the original algorithm using Register Transfer Level (RTL) notation. All variable names used in the original algorithm are preserved.

- **Low-level ASM (Asmchart)**

The RTL statements are then replaced with control and status signals as defined in the datapath unit. This forms the control ASM.

1.2 Naïve multiplier design

The following application notes may be useful references (<https://raden.fke.utm.my/app-notes>)

- AN07 Zebra Crossing Simulator on CPLD
- AN10 Serial Adder with Custom Modules
- AN11 Serial Adder with Library of Parameterized Modules

1.2.2 Problem Formulation

Every datapath unit implements a specific algorithm. To verify correctness, we first describe the algorithm in C (or equivalent) and simulate it at a high level.

Each CU + DU combination corresponds to a potentially different algorithm implementation. Even if the top-level function is the same, the internal circuit structure may vary significantly.

Multiplication in digital systems can be implemented in many ways. The binary multiplier is the most common solution because it is hardware-efficient and fast.

The naïve multiplier, however, implements multiplication by repeated addition of the multiplicand. Although this method is inefficient, it is very useful for illustrating digital system design using simple components.

For example, multiplying 2 by 5 means performing the addition five times:

$$0 + 2 + 2 + 2 + 2 + 2 = 10$$

The algorithm for computing $P \leftarrow M \times N$ is as follows:

```
int naivemultiplier(int M, int N) {  
    int P;  
    P = 0;  
    while( N != 0) { P  
        = P + M;  
        N = N - 1;  
    }  
    return P;  
}
```

Based on the algorithm, the RTL-ASMchart for the naïve multiplier is given in Figure 2.

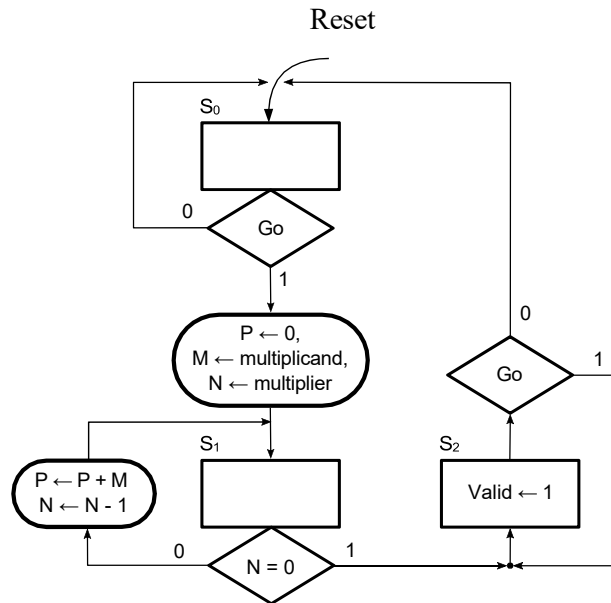


Figure 2: RTL-ASMchart for naïve multiplier.

1.2.3 Identify and Construct the DPU FBD

From the RTL-ASM chart in Figure 2, the components required for a 4-bit $\times$ 4-bit naïve multiplier datapath are listed in Table 1. Figure 3 shows the functional block diagram (FBD), consisting of the DPU and CU blocks

Table 1: Components of the naïve multiplier datapath

Component	Notes
Adder	8 bits adder.
M register	4-bit register with enable. This register must have 4-bit input, 4-bit output and Load control input.
N register	4-bit down-counter with load capability. It must have a 4-bit data input, a Load control input, a Count Down Enable input, and a Zero output. The Zero output is high when the counter reaches 0000. The internal value is only important for debugging.
P register	8-bit register with enable. This register must have 8-bit input, 8-bit output and Load control input.

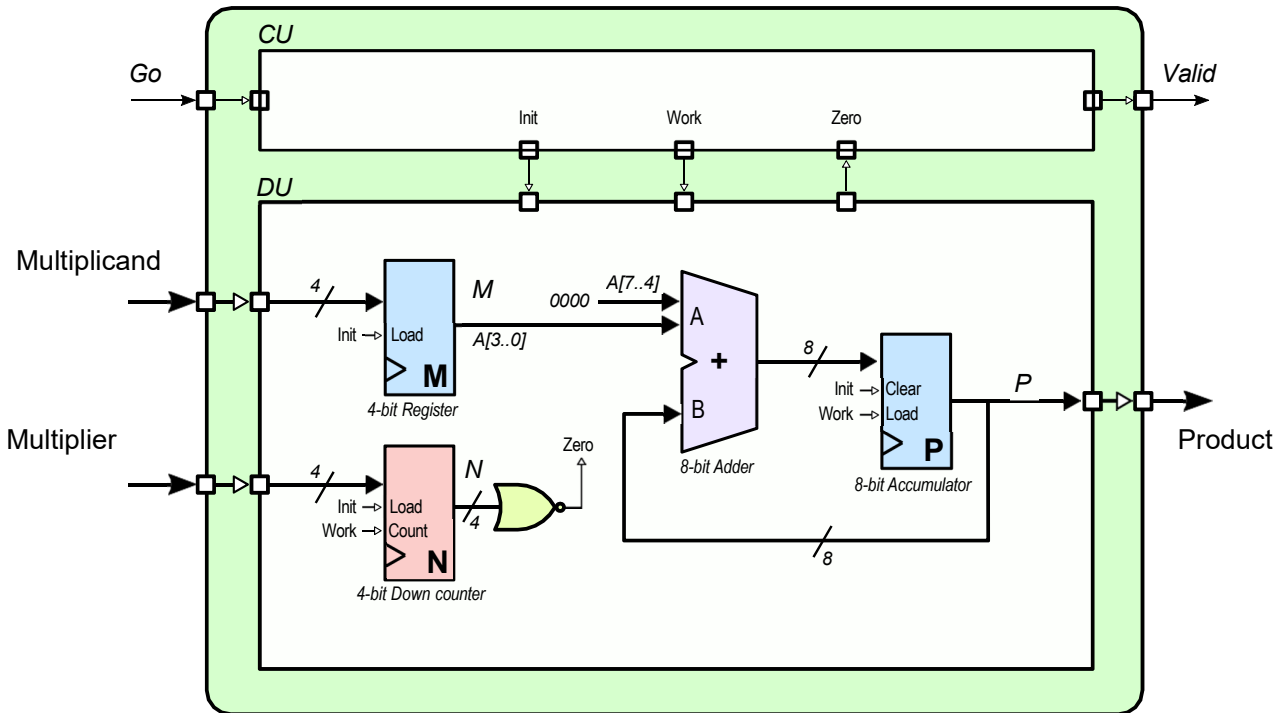


Figure 3: Naïve multiplier DPU + CU.

Table 2: Clock-by-clock countdown of Multiplier datapath.

Data			Signals		
Multiplicand (M)	Multiplier (N)	Product (P)	Init	Work	Zero
?	?	?	1	0	0
5	3	0	0	1	0
5	2	5	0	1	0
5	1	10	0	1	0
5	0	15	0	0	1

For each block in the datapath, you must write your own Verilog modules to implement the required functionality.

1.2.4 Verilog implementation for DPU

1. Design the Datapath

Implement the datapath according to Figure 3. An example Verilog implementation is given in Listing 1.

2. Test the Datapath

Verify the datapath using appropriate test data, as shown in Listing 2. Before running simulation, determine the expected outputs. The ability to choose meaningful input vectors and to verify circuit behavior is an essential skill for digital designers.

3. Create a Symbol

After verification, save the datapath as a symbol file for use in higher-level designs.

Listing 1: Design.v

```

module du( input Clock, Reset, input [3:0] Multiplicand, Multiplier, input Init, Work,
output reg [7:0] Product, output Zero);
reg [3:0] M, N;
reg [7:0] P;

always @ (posedge Clock, posedge Reset) //register M
if (Reset) M<=0;
else if (Init) M <= Multiplicand;
else M <= M;

always @ (posedge Clock, posedge Reset) //downcounter N
if (Reset) N<=0;
else if (Init) N <= Multiplier;
else if (Work) N <= N - 1;
else N <= N;

always @ (posedge Clock, posedge Reset) //register P
if (Reset) P<=0;
else if (Init) P <= 0;
else if (Work) P <= P + {4'b0000,M};
else P <= P;

```

```
assign Zero = ~|N; // (N == 0);
assign Product = P;

endmodule
```

Listing 2 : testbench.v

```
`timescale 10ns/1ns
module testbenchDU();

    reg Clock, Reset;
    reg [3:0] Multiplicand, Multiplier;
    reg Init, Work;
    wire [7:0] Product;
    wire Zero;

    //Instantiate the design-under-test (dut) into testbench
    du dut( .Clock(Clock), .Reset(Reset), .Multiplicand(Multiplicand), .Multiplier(Multiplier),
    .Init(Init), .Work(Work), .Product(Product), .Zero(Zero));

    //report the circuit response (display response)
    initial begin
        $monitor($time, "Clock=%b, Reset=%b, Multiplicand=%b, Multiplier=%b, Init=%b, Work=%b,
        Product=%b, Zero=%b,", Clock, Reset, Multiplicand, Multiplier, Init, Work, Product, Zero);
    end

    initial begin
        Clock=1'b0; forever #10 Clock=~Clock;
    end

    initial begin
        #0 Reset = 1;
        #5 Reset = 0;
    end

    //provide input test vectored
    initial begin
        $dumpvars(1, testbenchDU); //remove this line code if using Quartus
        #0 Multiplier = 3; Multiplicand = 5; Init =0; Work=0;
        #6 Init=1;
        #7 Init =0; Work=1;
        #100 $finish;
    end

endmodule
```

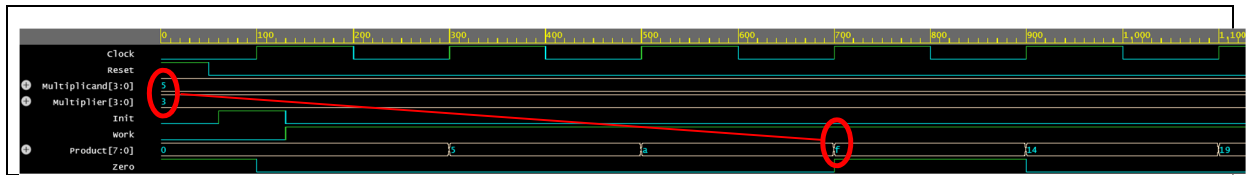


Figure 4: Simulation of naïve multiplier datapath. The *Work* signal enables the counter to count down and the accumulator to load a new value. When the controller is added later, *Work* can be stopped automatically when *Zero* = 1.

1.2.5 Identify and Construct the CU FBD

Starting from the RTL-ASM chart in Figure 2, derive the ASM chart for the controller as shown in Figure 5.

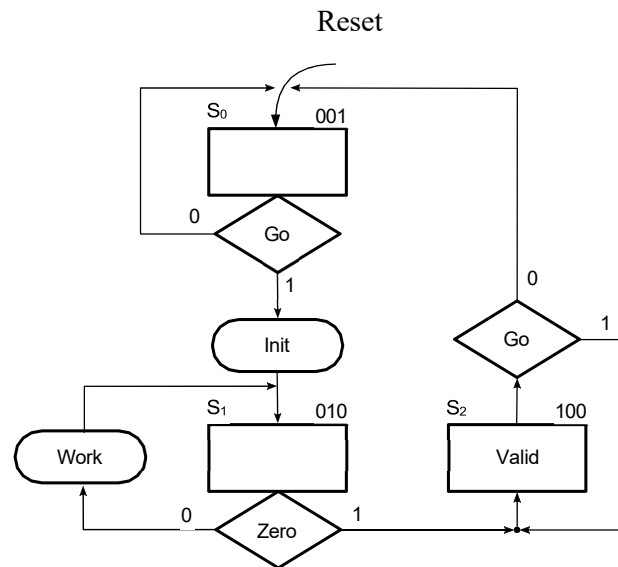


Figure 5: ASMchart for naïve multiplier.

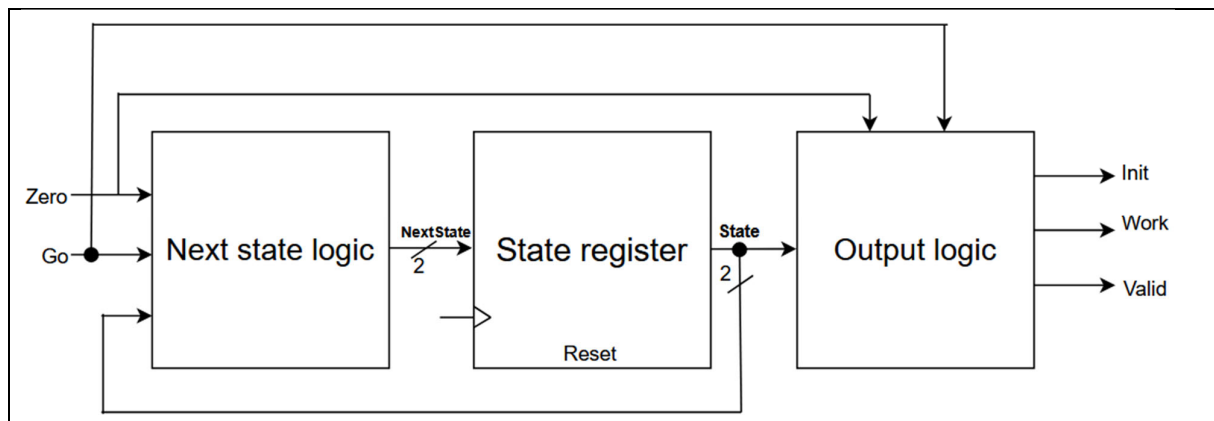


Figure 6: Naïve multiplier CU FBD.

1.2.6 Verilog implementation for CU

From the ASM chart in Figure 5, you can derive the RTL–Control Signal (RTL-CS) table shown in Table 3. This table lists the control signals to be asserted in each state. Using this table, you can write the CU in Verilog using the standard three-segment style: state register, next-state logic, and output logic.

The CU design is verified using the testbench in Listing 4.

Table 3 : RTL-CS Table

RTL code	Control signals activated		
	Init	Work	Valid
S0 : (Go)/ $P \leftarrow 0$, $M \leftarrow \text{Multiplicand}$, $M \leftarrow \text{Multiplier}$; (Go)/ goto S1 :goto S0;	1	0	0
S1 : (Zero)/ $P \leftarrow P + M$, $N \leftarrow N-1$; (Zero)/ goto S2 :goto S1;	0	1	0
S2 : Valid =1;	0	0	1

Listing 3: Design.v

```

module cu(
  input Clock, Reset, Go, Zero,
  output reg Init, Work, Valid, output reg [1:0] State
);
  //reg [1:0] State, NextState;
  reg [1:0] NextState;
  parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;

  always @ (negedge Clock, posedge Reset) // memory part
    if (Reset) State <= S0;
    else State <= NextState;

  always @ (*) // next state logic
    case (State)
      S0: if (!Go) NextState = S0; else NextState = S1;
      S1: if (!Zero) NextState = S1; else NextState = S2;
      S2: if (!Go) NextState = S0; else NextState = S2;
    endcase

  always @ (*) // output logic
    begin
      Valid=0; Work =0; Init = 0;
      case (State)
        S0: if (Go) Init = 1;
        S1: if (!Zero) Work = 1;
        S2: Valid = 1;
      endcase
    end

endmodule

```

Listing 4: Testbench.v

```

`timescale 10ns/1ns
module testbenchCu();
    reg Clock, Reset, Go, Zero;
    wire Init, Work, Valid;
    wire [1:0] State;

    //Instantiate the design-under-test (dut) into testbench
    cu dut( .Clock(Clock), .Reset(Reset), .Go(Go), .Zero(Zero), .Init(Init), .Work(Work),
    .Valid(Valid), .State(State));

    //report the circuit response (display response)
    initial begin
        $monitor($time, "Clock=%b, Reset=%b, Go=%b, Zero=%b, Init=%b, Work=%b,
        Valid=%b, State=%b", Clock, Reset, Go, Zero, Init, Work, Valid, State);
    end
    initial begin
        Clock=1'b1; forever #10 Clock=~Clock;
    end
    //provide input test vectored
    initial begin
        $dumpvars(1, testbenchCu);
        #0 Go=0; Zero= 0;
        #6 Go=1;
        #30 Zero=1;
        #100 $finish;
    end

    initial
    begin
        #0 Reset = 1; Go = 0;
        #5 Reset = 0;
        #1 Go = 1;
        #50 Go = 0;
    end
endmodule

```

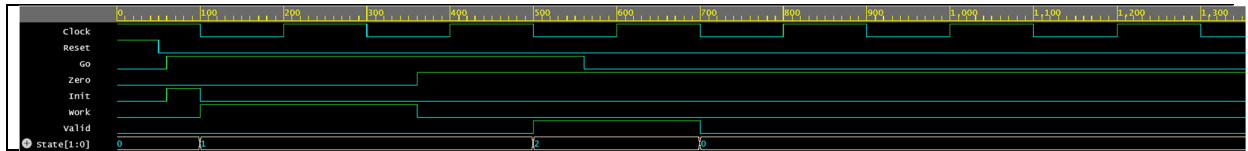


Figure 7: Simulation of naïve multiplier control unit

1.2.7 System Integration

Finally, integrate the CU and DPU to complete the naïve multiplier system.

The top-level functional block diagram of the system is the same as in Figure 3, with the CU and DU connected appropriately. The combined Verilog design is given in Listing 5.

Test the system by applying several input combinations, for example:

- 1×1
- 2×5
- 5×3

Use the testbench in Listing 6. In the testbench:

Assign the values to be multiplied to the Multiplicand and Multiplier signals.

Assert Go = 1 for a suitable duration to load the operands into the datapath and start the multiplication.

Listing 5: Design.v

```

module naivemultiplier(
  input Clock, Reset, Go,
  input [3:0] Multiplicand, Multiplier,
  output Valid,
  output [7:0] Product);

  wire init, work, zero;

  cu u1(.Clock(Clock), .Reset(Reset), .Go(Go), .Zero(zero),
    .Init(init), .Work(work), .Valid(Valid) );
  du u2(.Clock(Clock), .Init(init), .Work(work),
    .Multiplicand(Multiplicand), .Multiplier(Multiplier),
    .Product(Product), .Zero(zero));
endmodule

```

Listing 6: Testbench.v

```

`timescale 10ns/1ns
module testbench();
  reg Clock, Reset, Go;
  reg [3:0] Multiplicand, Multiplier;
  wire Valid;
  wire [7:0] Product;

  //Instantiate the design-under-test (dut) into testbench
  naivemultiplier dut( .Clock(Clock), .Reset(Reset), .Go(Go), .Multiplicand(Multiplicand),
    .Multiplier(Multiplier), .Valid(Valid), .Product(Product));

  //report the circuit response (display response)
  initial begin
    $monitor($time, "Clock=%b, Reset=%b, Go=%b, Multiplicand=%b, Multiplier=%b,
    Valid=%b, Product=%b", Clock, Reset, Go, Multiplicand, Multiplier, Valid, Product);
  end

  initial begin
    Clock=1'b0; forever #10 Clock=~Clock;
  end

  //provide input test vectored
  initial begin
    $dumpvars(1, testbench);
    Multiplier = 3;
    Multiplicand = 5;
    #100 $finish;
  end

  initial
  begin
    #Reset = 1; Go = 0;
    #5 Reset = 0;
    #1 Go = 1;
    #50 Go = 0;
  end
endmodule

```

Timing diagram

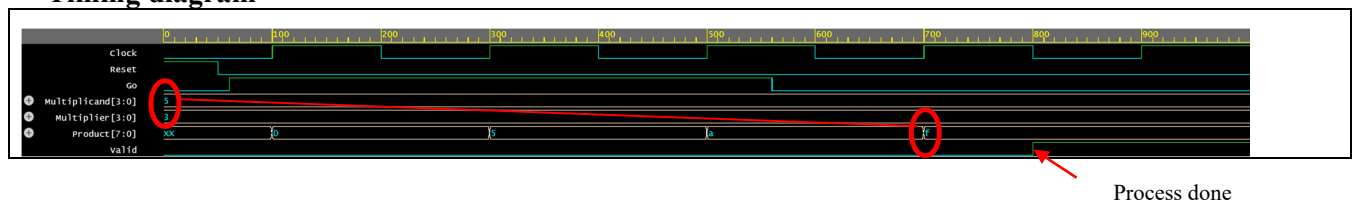


Figure 9: Simulation of 5×3 at the module level. Annotations helps the reader look at the important parts of the waveform.

Full source code of naïve multiplier

<https://edaplayground.com/x/vJxs>