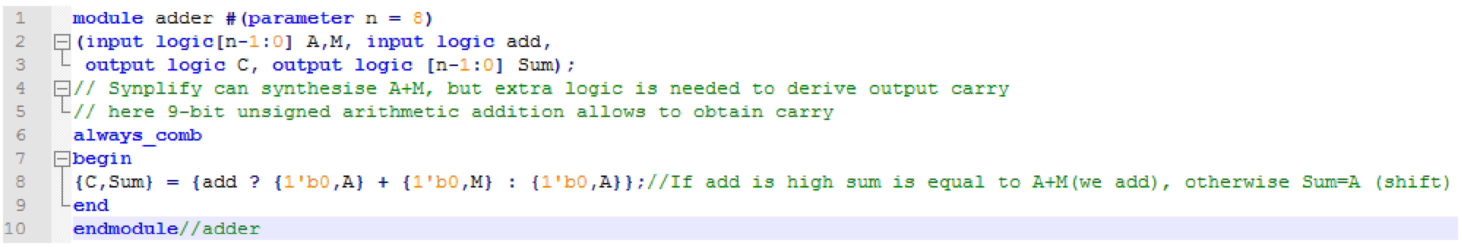
RTLnBitMultiplier Documentation

Samyuel Danyo
28.11.2016

1. Adder design, simulation and synthesis

I parameterised the module, and added “add” as an input. Based on its value the module assigns “Sum” to be either the sum of the accumulator and the multiplicand and “C”- the carry (adding and shifting) or “Sum” passes the value of the accumulator back to it and “C” =0(just shifting).

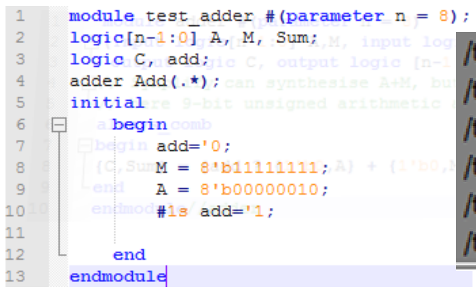


```
module adder #(parameter n = 8)
(input logic[n-1:0] A,M, input logic add,
 output logic C, output logic [n-1:0] Sum);
// Synplify can synthesise A+M, but extra logic is needed to derive output carry
// here 9-bit unsigned arithmetic addition allows to obtain carry
always_comb
begin
{C,Sum} = {add ? {1'b0,A} + {1'b0,M} : {1'b0,A}};//If add is high sum is equal to A+M(we add), otherwise Sum=A (shift)
end
endmodule//adder
```

Figure 1: SV code for “adder” module.

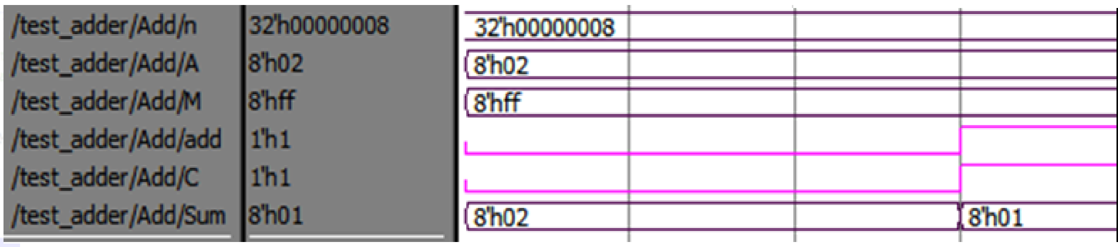
1.1 Adder simulation

The Modelsim simulation (Figure 2) shows “A” being loaded into “Sum” and “C” set to zero when “add” =0. After setting “add” =1 the sum of “A” and “M” is loaded into “Sum” while “C” saves the carry.



```
module test_adder #(parameter n = 8);
logic[n-1:0] A, M, Sum;
logic C, add;
adder Add(.*);
initial
   begin
      add='0;
      M = 8'b11111111;
      A = 8'b00000010;
     #1s add='1;

   end
endmodule
```



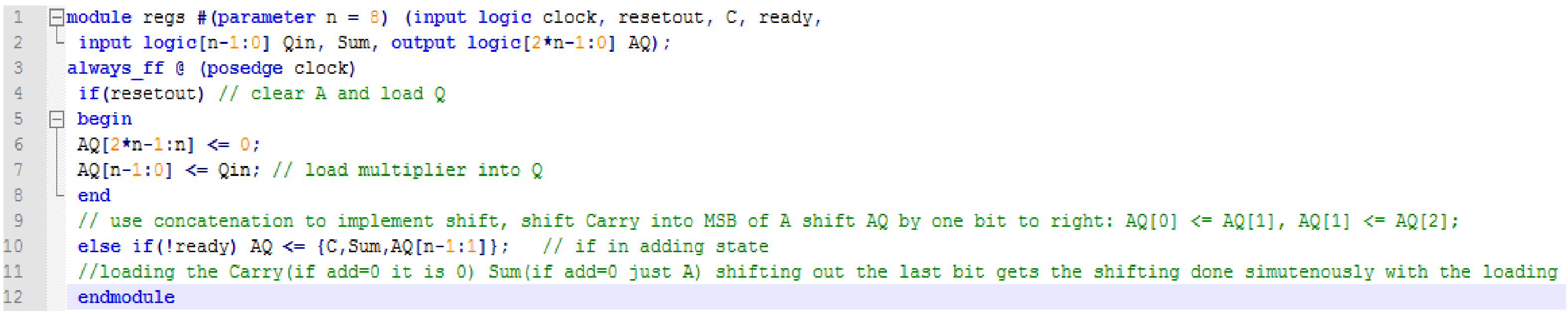
```
/test_adder/Add/n     32'h00000008   32'h00000008
/test_adder/Add/A     8'h02          8'h02
/test_adder/Add/M     8'hff          8'hff
/test_adder/Add/add   1'h1
/test_adder/Add/C     1'h1
/test_adder/Add/Sum   8'h01          8'h02                 8'h01
```

Figure 2: Modelsim simulation of “adder” module.

Figure 3: Testbench SV code for “adder” module

2. Register design, simulation and synthesis

I parameterised the module, and added “ready” as an input, while removing “shift”. When ready is



```
module regs #(parameter n = 8) (input logic clock, resetout, C, ready,
 input logic[n-1:0] Qin, Sum, output logic[2*n-1:0] AQ);
always_ff @ (posedge clock)
 if(resetout) // clear A and load Q
 begin
 AQ[2*n-1:n] <= 0;
 AQ[n-1:0] <= Qin; // load multiplier into Q
 end
 // use concatenation to implement shift, shift Carry into MSB of A shift AQ by one bit to right: AQ[0] <= AQ[1], AQ[1] <= AQ[2];
 else if(!ready) AQ <= {C,Sum,AQ[n-1:1]};   // if in adding state
 //loading the Carry(if add=0 it is 0) Sum(if add=0 just A) shifting out the last bit gets the shifting done simutenously with the loading
 endmodule
```

low the system is in calculation state(“adding”), the register loads “C”, “Sum” and the last (n-1) bits of “AQ”. This behaviour in symbiosis with the “adder” combines the add and shift states into one.

Figure 4: SV code for “regs” module

2.1 Simulation of registers

The Modelsim simulation (Figure 5) shows the registers being reset on the first rising edge with “resetout” =1. After that “AQ” doesn’t change value while “ready” is high even when “resetout” goes low (stopped state). On the first rising edge with “ready” and “stopped” both low the module loads “C” and “Sum”, shifting one bit out**.**

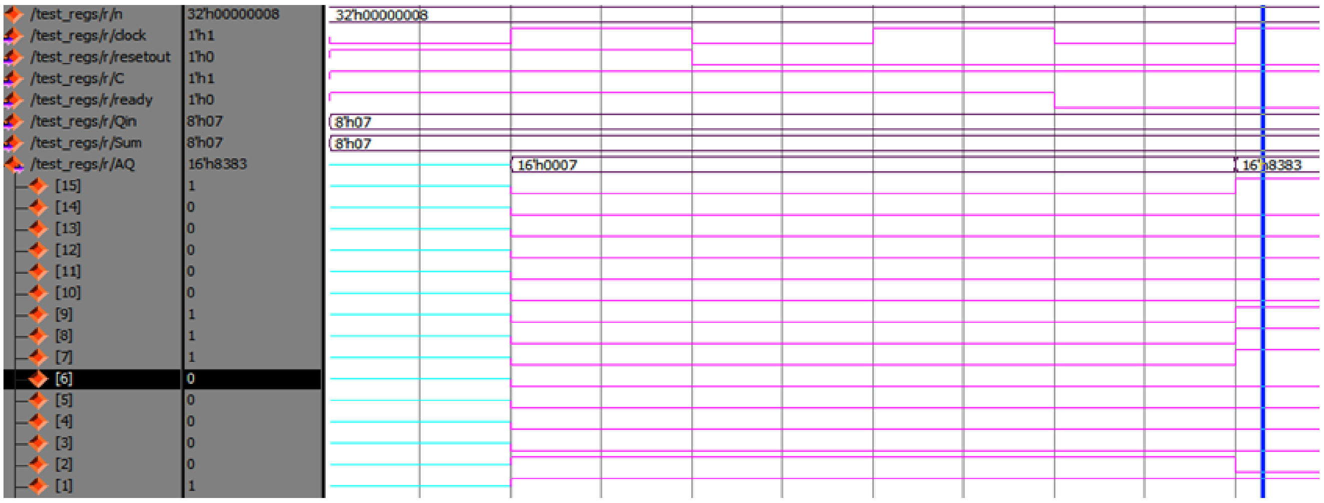
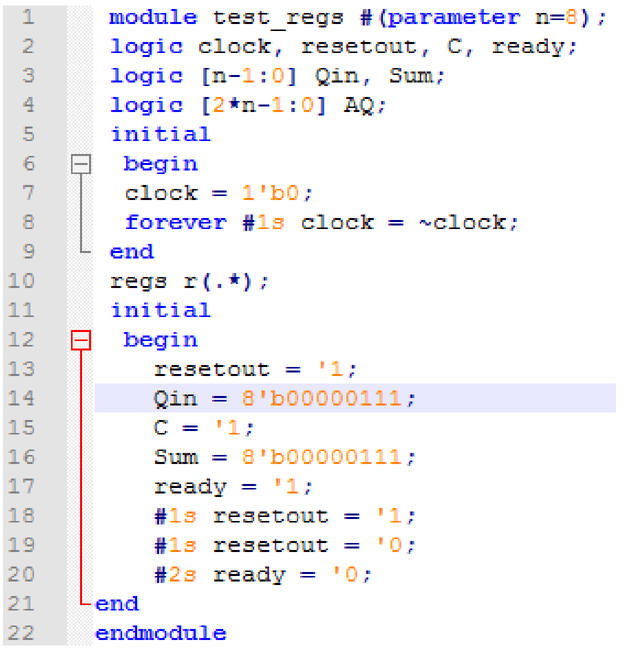


Figure 5: Modelsim simulation of “regs” module.



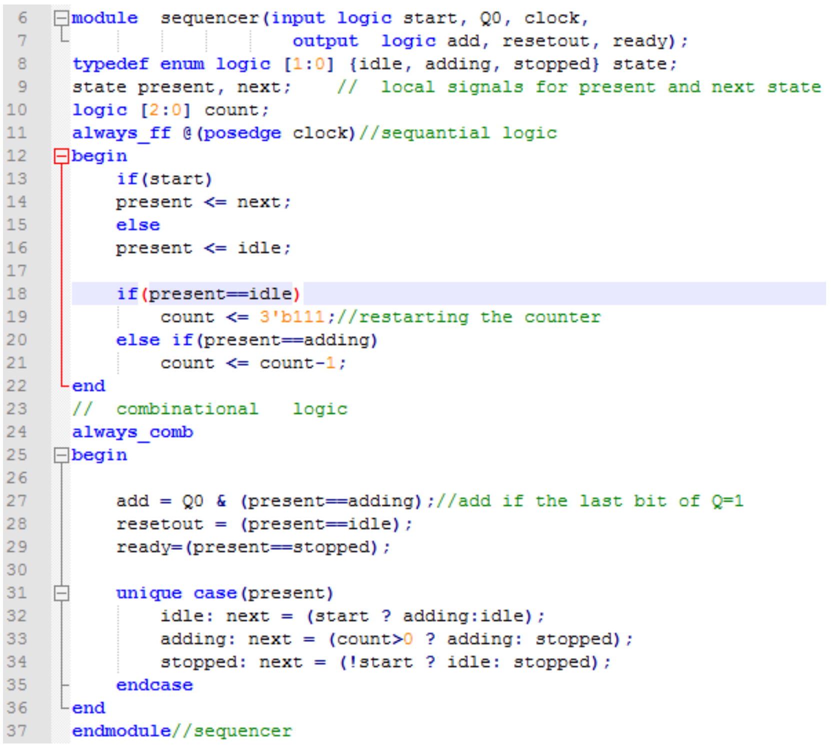
```
module test_regs #(parameter n=8);
logic clock, resetout, C, ready;
logic [n-1:0] Qin, Sum;
logic [2*n-1:0] AQ;
initial
 begin
 clock = 1'b0;
 forever #1s clock = ~clock;
end
regs r(.*);
initial
 begin
   resetout = '1;
   Qin = 8'b00000111;
   C = '1;
   Sum = 8'b00000111;
   ready = '1;
   #1s resetout = '1;
   #1s resetout = '0;
   #2s ready = '0;
end
endmodule
```

Figure 6: Testbench SV code for “regs” module.

3. Sequencer design, simulation and synthesis

From the ASM chart (Figure 8) can be seen the states flow of the sequencer. The code (Figure 7) can be interpreted: the machine has three states: idle, adding (more appropriate name is calculating) and stopped. In the sequential block the progress from the present state to the next is dependent on the start input, if it is high the system changes states if it is low the system stays in idle. The second part of the sequential block is for the cycle counter. If the system is in idle it is reset to (3’b111) for eight bits multiplication, if the system is in calculation state the counter decreases.
In the combinational block first I take care of the outputs. “add” goes high if the system is in calculating state and Q0 is one. The reset – “resetout” goes high if the system is in “idle” state and “ready” goes high if the system is in stopped state.

Last, following the ASM flow I have the next state logic. If the system is in “idle” and start goes high the next state is “adding”, otherwise it stays in “idle”. Once in “adding” it loops in while the counter is more than zero. If the counter is zero, the machine gets in “stopped” state. It stays there until start goes low and it resets.



```
module  sequencer(input logic start, Q0, clock,
                  output  logic add, resetout, ready);
typedef enum logic [1:0] {idle, adding, stopped} state;
state present, next;    //  local signals for present and next state
logic [2:0] count;
always_ff @(posedge clock)//sequantial logic
begin
    if(start)
    present <= next;
    else
    present <= idle;

    if(present==idle)
        count <= 3'b111;//restarting the counter
    else if(present==adding)
        count <= count-1;
end
//  combinational   logic
always_comb
begin

    add = Q0 & (present==adding);//add if the last bit of Q=1
    resetout = (present==idle);
    ready=(present==stopped);

    unique case(present)
        idle: next = (start ? adding:idle);
        adding: next = (count>0 ? adding: stopped);
        stopped: next = (!start ? idle: stopped);
    endcase
end
endmodule//sequencer
```

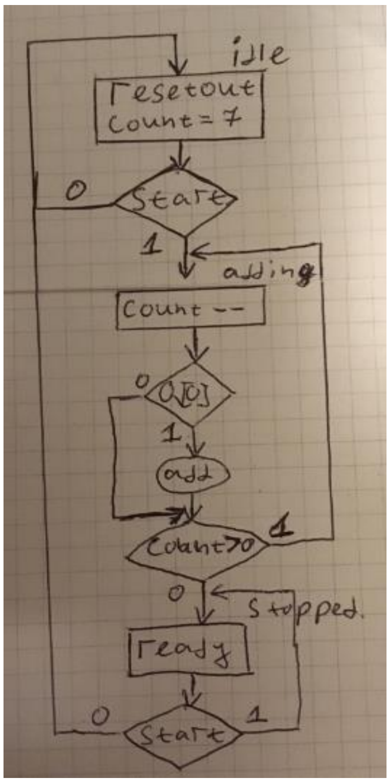


Figure 7: SV code for "sequencer" module. **Figure 8: Sequencer ASM chart**

In the Modelsim simulation I inverted "start" for a better understanding. (Figure 9) shows the system going into "idle" on the first rising edge with "start" being high. The "next" state is "idle" and goes to "adding" only after "start" goes low. It can be seen "resetout" being high in the "idle" state and "add" going high while the machine is in "adding" state and "Q0" is one. The counter is decreasing by one on each rising edge of the clock and when it is equal to zero, the next state becomes "stopped", closing eight cycles of calculating. "Ready" goes high and the system stays in "stopped" while the start is low, once it goes high the system is reset. The system stays two clock cycles in idle, because "start" is high on the rising edge, once it gets low the machine starts calculating again. It can be seen that the counter becomes seven in stopped, because it is decreased after the last cycle, meaning that the initialisation in idle might be unnecessary.

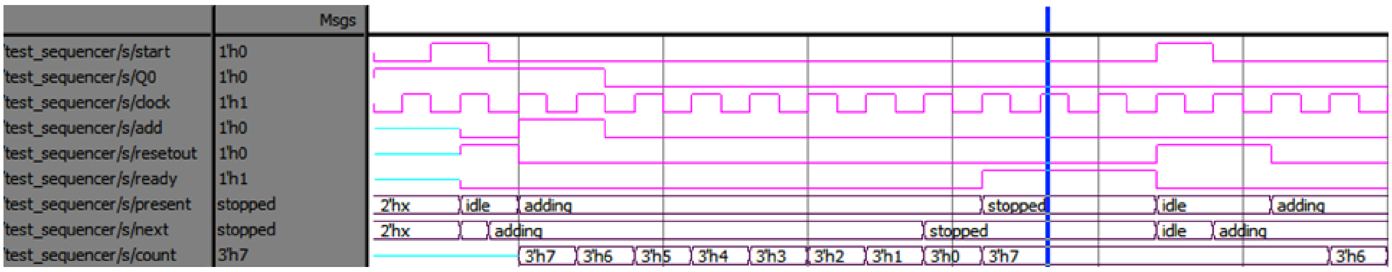


Figure 9: Modelsim simulation of "sequencer" module.

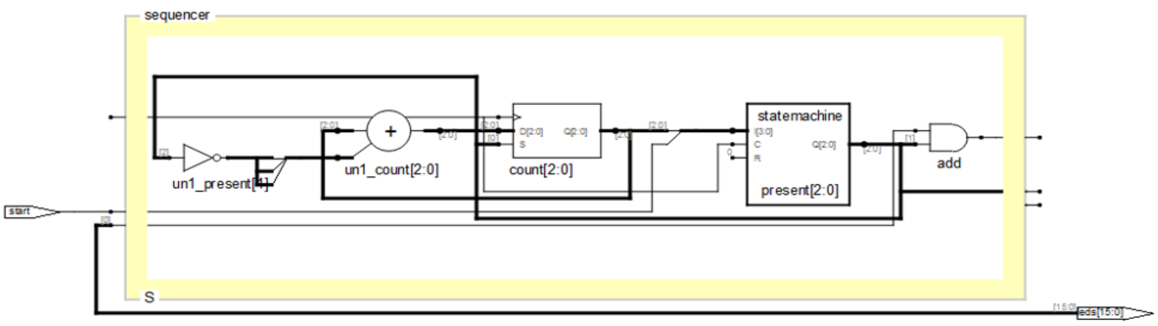
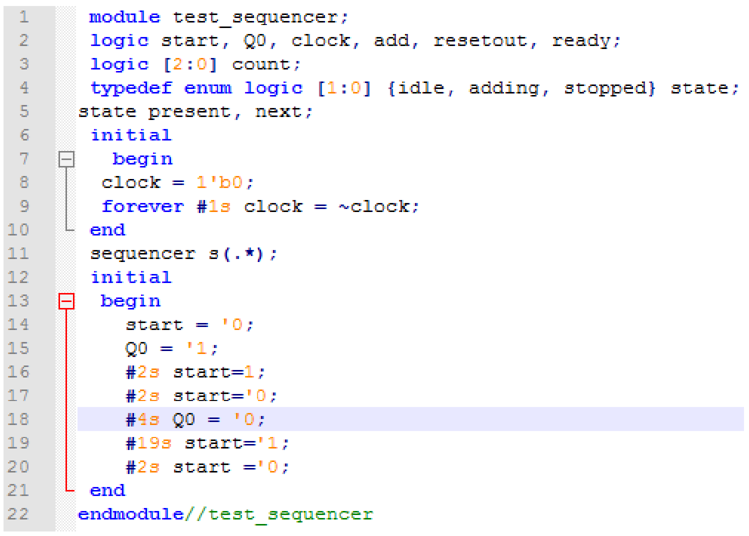


Figure 10: RTL synthesis diagram of "sequencer" module.

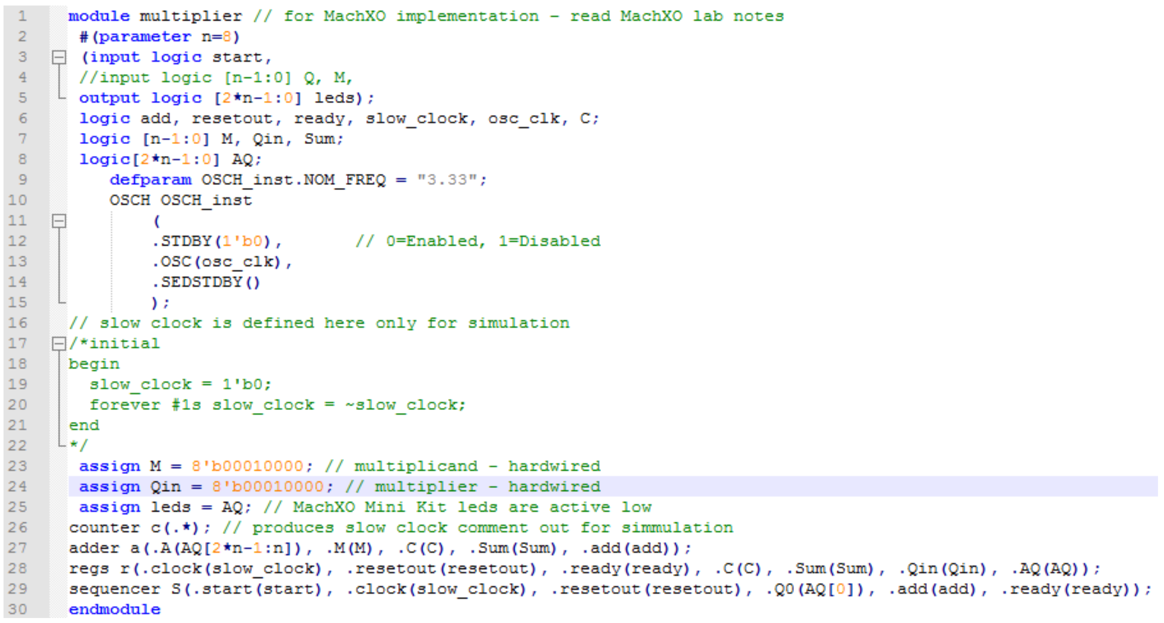


```
module test_sequencer;
 logic start, Q0, clock, add, resetout, ready;
 logic [2:0] count;
 typedef enum logic [1:0] {idle, adding, stopped} state;
state present, next;
 initial
   begin
  clock = 1'b0;
  forever #1s clock = ~clock;
 end
 sequencer s(.*);
 initial
  begin
    start = '0;
    Q0 = '1;
    #2s start=1;
    #2s start='0;
    #4s Q0 = '0;
    #19s start='1;
    #2s start ='0;
 end
endmodule//test_sequencer
```

Figure 11: Testbench SV code “sequencer” module

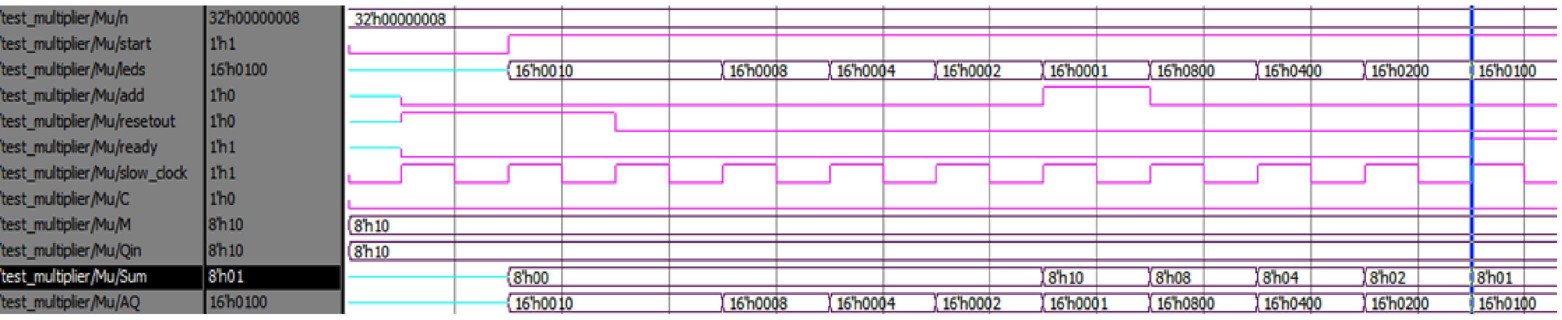
4. Multiplier design, simulation and synthesis

I parameterised the module. M and Qin are hardwired - constant values. My first idea was to make them inputs to the module, I would have connected those to a switch board.
The Modelsim simulation (Figure 13) shows “resetout” going high on the first rising edge with “start” being one. “Qin” is loaded into “AQ” and the leds take its value. AQ is shifted 4 times (Qin=8’b00010000) and when the one comes first add goes high, we get the sum of A and M, which is then shift-loaded into AQ. After that we have three more shifts for the last three zeroes of Qin and ready goes high with the right result.



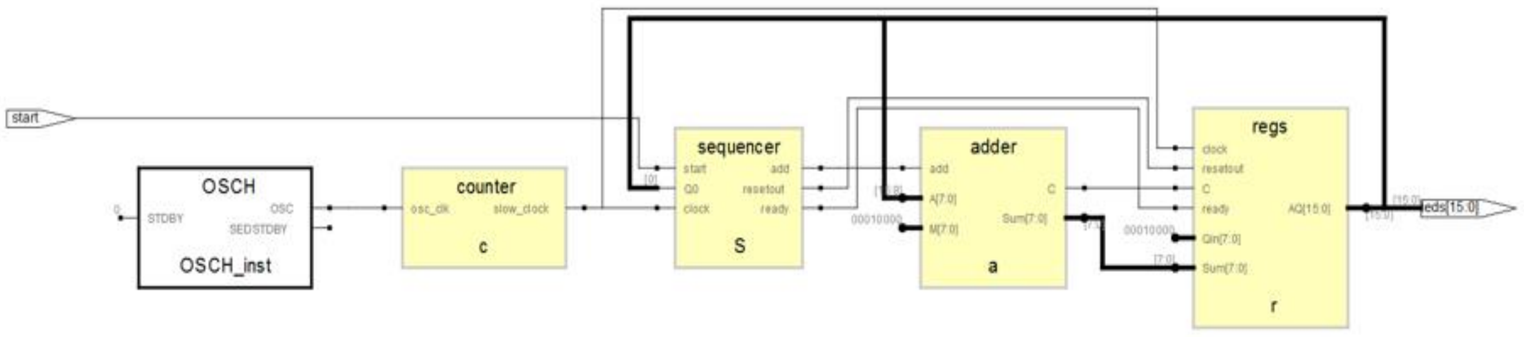
```
module multiplier // for MachXO implementation - read MachXO lab notes
 #(parameter n=8)
 (input logic start,
 //input logic [n-1:0] Q, M,
 output logic [2*n-1:0] leds);
 logic add, resetout, ready, slow_clock, osc_clk, C;
 logic [n-1:0] M, Qin, Sum;
 logic[2*n-1:0] AQ;
    defparam OSCH_inst.NOM_FREQ = "3.33";
    OSCH OSCH_inst
        (
        .STDBY(1'b0),       // 0=Enabled, 1=Disabled
        .OSC(osc_clk),
        .SEDSTDBY()
        );
// slow clock is defined here only for simulation
/*initial
begin
  slow_clock = 1'b0;
  forever #1s slow_clock = ~slow_clock;
end
*/
 assign M = 8'b00010000; // multiplicand - hardwired
 assign Qin = 8'b00010000; // multiplier - hardwired
 assign leds = AQ; // MachXO Mini Kit leds are active low
counter c(.*); // produces slow clock comment out for simmulation
adder a(.A(AQ[2*n-1:n]), .M(M), .C(C), .Sum(Sum), .add(add));
regs r(.clock(slow_clock), .resetout(resetout), .ready(ready), .C(C), .Sum(Sum), .Qin(Qin), .AQ(AQ));
sequencer S(.start(start), .clock(slow_clock), .resetout(resetout), .Q0(AQ[0]), .add(add), .ready(ready));
endmodule
```

12: SV code for encapsulating “multiplier” module.



/test_multiplier/Mu/n	32'h00000008
/test_multiplier/Mu/start	1'h1
/test_multiplier/Mu/leds	16'h0100
/test_multiplier/Mu/add	1'h0
/test_multiplier/Mu/resetout	1'h0
/test_multiplier/Mu/ready	1'h1
/test_multiplier/Mu/slow_clock	1'h1
/test_multiplier/Mu/C	1'h0
/test_multiplier/Mu/M	8'h10
/test_multiplier/Mu/Qin	8'h10
/test_multiplier/Mu/Sum	8'h01
/test_multiplier/Mu/AQ	16'h0100

Figure 13: Modelsim simulation of “multiplier” module.



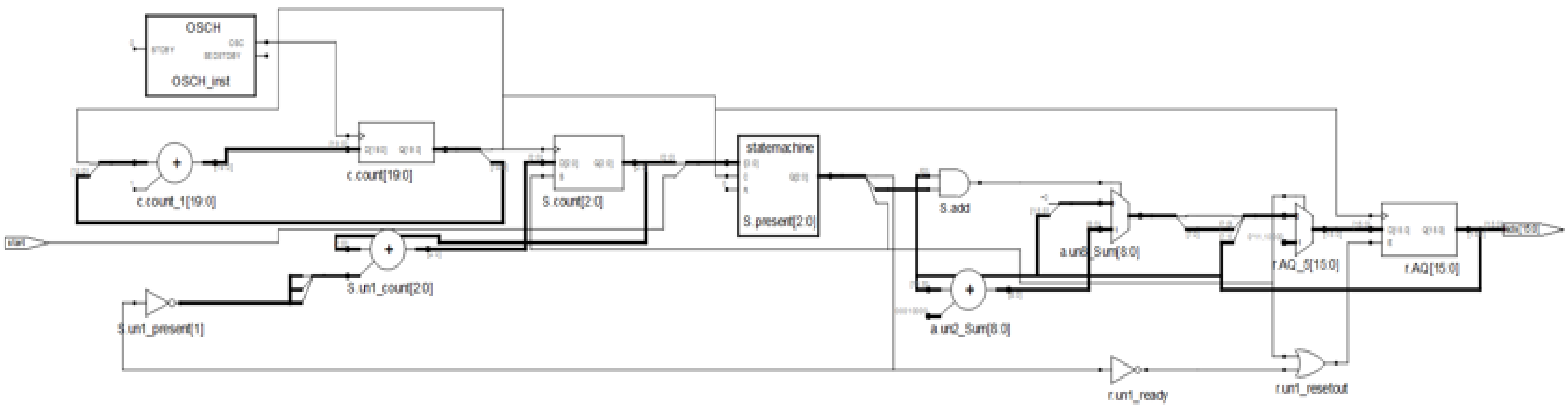
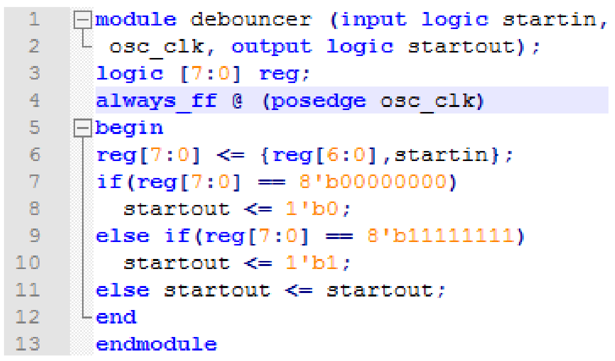


Figure 14: RTL synthesis diagram of encapsulating "multiplier" module.

5. Extensions

Extensions to the original task (4-bit multiplier) are shown above. Shift and add states are combined in one (one clock cycle) and the multiplier is working with 8 bits. The adder and registers are parameterised meaning that the multiplier can easily be extended to more bits with adjusting the counter in the sequencer. The code in (Figure 15) can be used to debounce the start button and so the internal clock to be used, something that is not included in the diagram above.



```
module debouncer (input logic startin,
 osc_clk, output logic startout);
logic [7:0] reg;
always_ff @ (posedge osc_clk)
begin
reg[7:0] <= {reg[6:0],startin};
if(reg[7:0] == 8'b00000000)
  startout <= 1'b0;
else if(reg[7:0] == 8'b11111111)
  startout <= 1'b1;
else startout <= startout;
end
endmodule
```

Figure 15: SV code for "debouncer"