ELEC2032: Electronic Design
D1 – SystemVerilog Design of a Sequential Multiplier

Ashley Robinson
ajr2g10@ecs.soton.ac.uk
Tutor: Dr Nick R Harris

Abstract: A detailed account of design and implementation of a sequential 4-bit multiplier on a MachXO Mini Development board. Achieved by taking different modular designs (written in SystemVerilog) and combining them with an encapsulating module. Then processing the designs (using *Synplify Pro* and *ispLEVER*) to a JEDEC file which can be downloaded to a CPLD. I was able to implement a working 4-bit multiplier on the MachXO and also an 8-bit multiplier by using parameters to vary bit lengths.

1. Introduction

Figure *1.1* is a simple view of the system required to implement a sequential multiplier. The task is broken into three modules which plug together to execute the sequential multiplier algorithm.

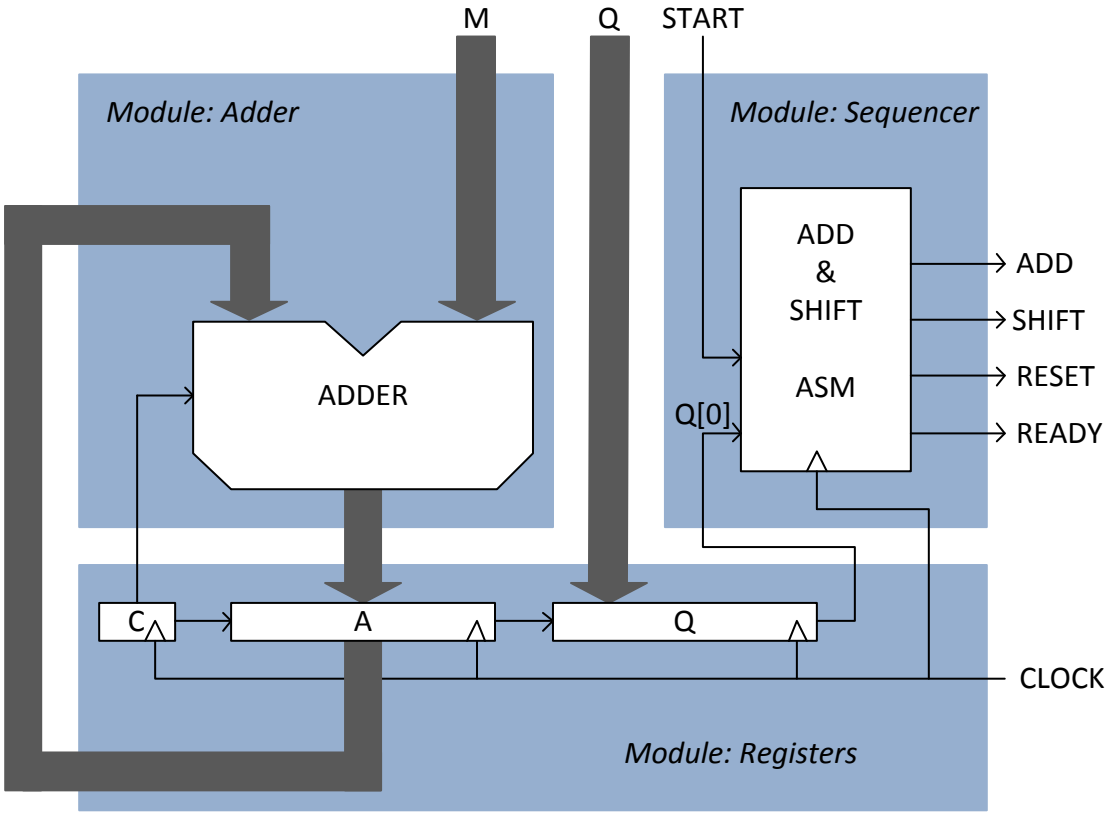

Figure 1.1: A sequential multiplier (Adaption of figure 3 from [1])

The algorithm used for a sequential multiplier is shown in figure *1.2*. This will be implemented in the sequencer design where it can control other modules.

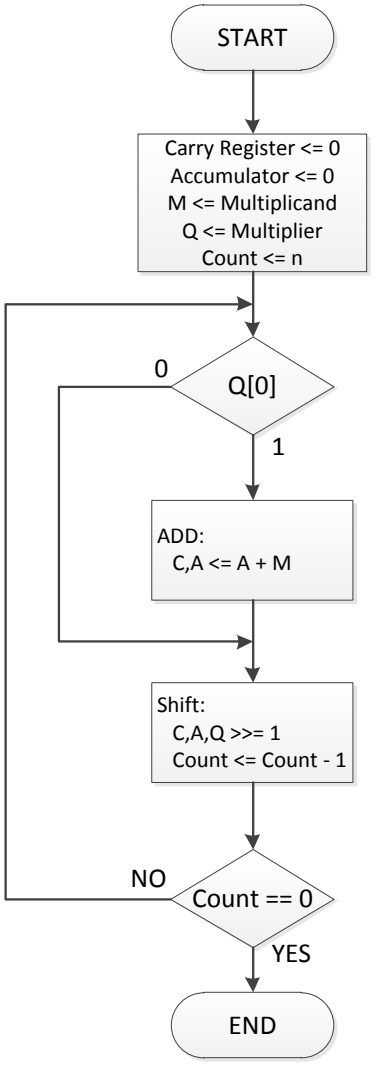

Figure 1.2: The “add and shift” algorithm for a sequential multiplier (adaption of figure 1 from [1])

2. Design and Simulation

2.1 Adder

The system requires the multiplicand (M) and the contents of register A to be summed several times in order to complete the process. SystemVerilog can produce an adder by simply using an addition sign but for a carry bit to be derived combinational logic has to be added.

```
 1   module Adder
 2   #(parameter n = 4)
 3   (
 4       input logic[n - 1:0] A,M,
 5       output logic C,
 6       output logic [n - 1:0] Sum
 7   );
 8
 9   always_comb
10     begin
11       Sum = M + A;
12       //If the output is lower than either
13       //input carry overflow has occured
14       if (Sum < A || Sum < M)
15         C = 1'b1;
16       else
17         C = 1'b0;
18     end
19  endmodule
```

Figure 2.1.1: SV code for a 4-bit adder with a carry bit

Figure *2.1.1* show how the carry bit is derived by comparing the output of the adder to both the inputs of the adder. The simulation in figure *2.1.2* tests both extreme cases of 4 bit addition (0 + 0 and 15 + 15) along with other cases which may or may not require a carry bit.

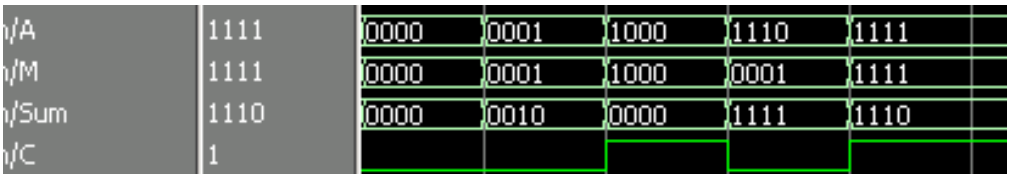

Figure 2.1.2: Simulation of the code from figure *2.1.1* in ModelSim

2.2 Registers

The registers are all flip-flops timed by the clock and 3 different signals cause their contents to change…

- **nreset** – When this signal is low the contents of Creg is cleared and the bits in AQ representing A are cleared (MSB to half the bit length). The bits at input Q are also latched to the other half of AQ. This signal takes priority and is assessed first in the "nested if statements" shown in figure *2.2.1*.
- **add** – This signal copies the output from the adder to the registers Creg and the A part of AQ.
- **shift** – Shift everything towards Q[0] by one bit.

```
1   module Regs
2   #(parameter n = 4)
3   (
4     input logic nreset, add, shift, C,clock,
5     input logic[n-1:0] Q, Sum,
6     output logic[(2*n)-1:0] AQ
7   );
8
9   logic Creg; // MSB carry bit storage
10
11  always_ff @ (posedge clock)
12    if(~nreset) // clear C,A and load Q
13      begin
14        Creg <= 0;
15        AQ[(2*n)-1:n] <= 0;
16        AQ[n-1:0] <= Q; // load multiplier into Q
17      end
18    else
19      begin
20        if (add) // store Sum in C,A
21          begin
22            Creg <= C;
23            AQ[(2*n)-1:n] <= Sum;
24          end
25        else
26          begin
27          if (shift) // shift A,Q
28            begin
29              // use concatenation to implement shift: Carry into MSB of A
30              // and AQ by one bit to right: AQ[0] <= AQ[1], AQ[1] <= AQ[2]; ?
31              // note also that Creg must be cleared
32              {Creg,AQ} <= {1'b0,Creg,AQ[(2*n)-1:1]};
33            end
34          end
35      end
36  endmodule
```

Figure 2.2.1: SV code for the systems registers

The simulation of the registers is split into two parts (I – figure *2.2.2*, II – figure *2.2.3*). Starting from part I it can be seen the state of AQ is unknown until the first clock pulse, at this point everything is reset because nreset is low, this is true until 3rd rising clock edge. Shift is high and add is low so the contents of AQ begin to shift right every clock cycle. Shift is then drawn low and add is pushed high so the contents from the adder (in this case I have replaced the adder with signals driven by the test bench) are placed into the correct position in AQ.

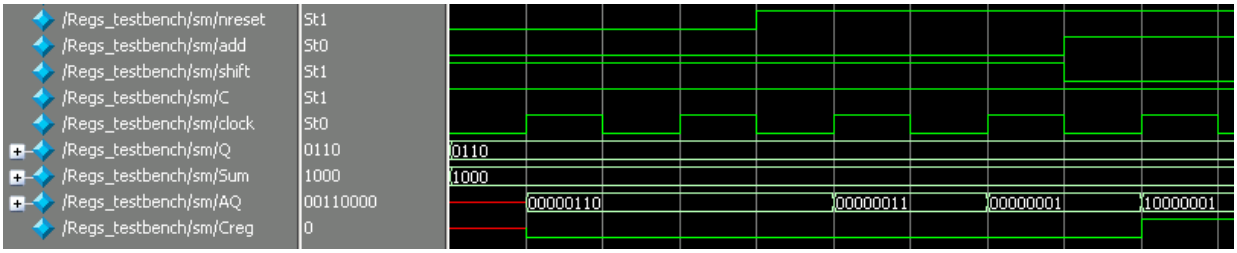

Figure 2.2.2: Simulation of the code from figure *2.2.1* in ModelSim (Part I)

In part II add remains high so the contents AQ do not change because I have not changed the values produced by the adder. The value of add and shift once again switch but this time when everything shifts the first bit introduced is 1 because the carry bit produced by the adder has been placed into Creg and shifted right as required.

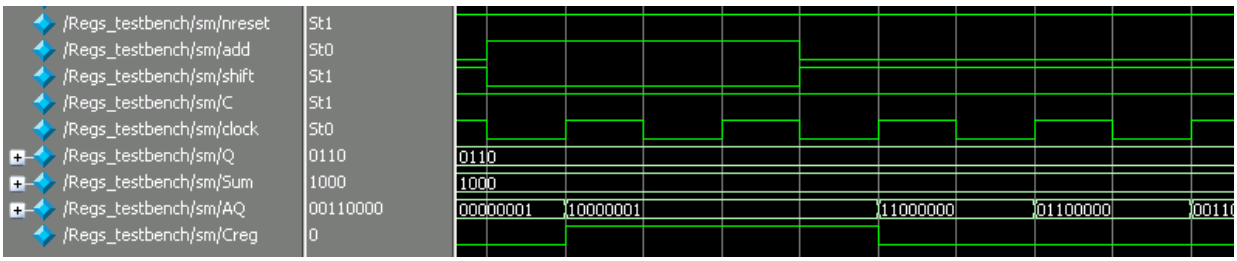

Figure 2.2.3: Simulation of the code from figure *2.2.1* in ModelSim (Part II)

2.3 Sequencer

This module is essentially the "add and shift" algorithm (figure *1.2*) that produces the product of the multiplier and multiplicand at the registers (figure *1.2*).
It has two inputs…

- **START** - When this signal is high the sequence takes the two inputs present at the time and begins the algorithm
- **Q[0]** - The algorithm needs LSB of **Q** to determine when to use the adder module

…and four outputs…

- **ADD** - When this output is high the contents of the **C** and **A** registers are replaced with the output of the adder
- **SHIFT** - Whenever this output is high the contents of the registers are shifted so **A[n]** will be replaced with **C** and the contents of **Q[0]** is lost
- **RESET** - This output is used to reset other modules
- **READY** - When the algorithm finishes this output is sent high

The method of attack I used for this design was to use two banks of flip-flops; the first controlling the state of the ASM and the second used as the counter required to count down from **n**. The header of the code for the sequencer is in figure *2.3.1*; it can be seen how the sates are defined as parameters for simplicity, internal logic for the states are defined and the two banks of flip-flops are defined.

```
1   module Sequencer
2   # (parameter n = 4)
3   (
4     input logic start,Q0,clock,
5     output logic add,shift,nreset,ready
6   );
7
8   parameter //All used states in the cycle (Gray code)
9     idle = 2'b00,
10    adding = 2'b01,
11    shifting = 2'b11,
12    stopped = 2'b10;
13
14  logic [1:0]//Internal logic holding the next state
15    current,
16    next;
17
18  logic [n - 1:0]//now and next states for the counter
19    count_now,
20    count_next;
21
22  //Flip flops that cycle through the states in the sequencer
23  always_ff@(posedge clock)
24      current <= next;
25
26  //Flip flops that hold the state of count
27  always_ff@(posedge clock)
28    count_now <= count_next;
29
```

Figure 2.3.1: SV code header for the sequencer

The progress of the states is determined by the next state logic (figure *2.3.2*). I have used a case statement dependent on the state of current; this can be one of four different states (depending on what stage of the algorithm the system is in). All logic is mentioned in every case statement for completeness and a default case is added so in case current slips into an unknown state the system can correct itself.

```
40  //Combinational logic that determines next state
41  always_comb
42    case(current)
43      idle:
44        begin
45          count_next = n;
46          add = 0;
47          shift = 0;
48          nreset = 0;
49          ready = 0;//intialise values
50          if(start)
51            next = adding;
52          else
53            next = idle;
54        end
55      adding:
56        begin
57          nreset = 1;
58          shift = 0;
59          ready = 0;
60          count_next = count_now - 1;
61          next = shifting;
62          if(Q0)
63            add = 1;
64          else
65            add = 0;
66        end
67      shifting:
68        begin
69          add = 0;
70          shift = 1;
71          nreset = 1;
72          ready = 0;
73          count_next = count_now;
74          if(count_now == 0)
75            next = stopped;
76          else
77            next = adding;
78        end
79      stopped:
80        begin
81          ready = 1;
82          shift = 0;
83          nreset = 1;
84          add = 0;
85          count_next = count_now;
86          if(start)
87            next = idle;
88          else
89            next = stopped;
90        end
91      default://shouldn't be required but added for completeness
```

Figure 2.3.2: SV next state code for the sequencer

The simulation of the sequencer in figure *2.3.3* starts with an unknown state for **current** which causes the case statement to jump to the **default** block. The default block then redirects to **idle** block which sets the value of **count_next** to that the bit length (**n**). **Start** is high over the 2nd clock pulse so the algorithm is set into motion. The state of **current** shuffles between states whilst shifting but because **Q0** is low so is **add**. Eventually **Q0** from the test bench is high so the sequencer alternates between sending **add** and **shift** high. The counter reaches zero and the algorithm is finshed so **ready** sent high.

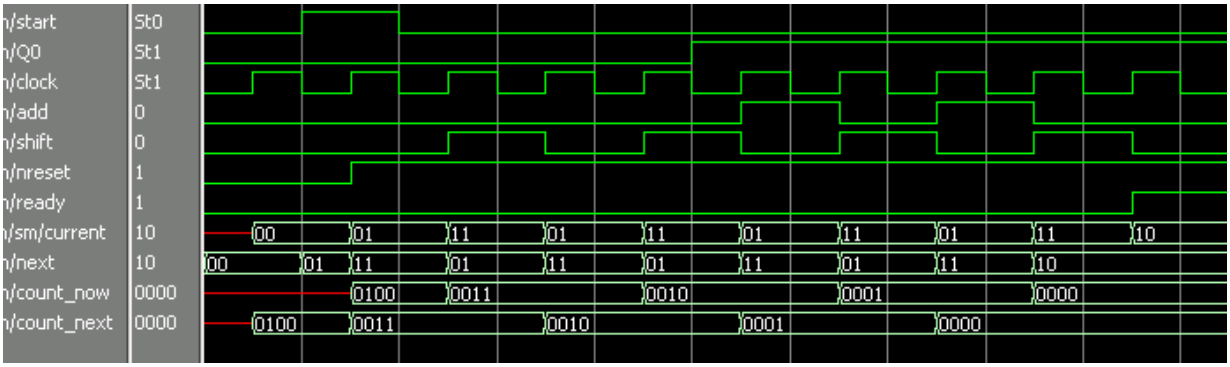

Figure 2.3.3: Simulation of the sequencer in ModelSim

2.4 Encapsulation

Figure *2.4.1* is the encapsulating module and creates instances of all three modules described in section 2 along with logic common to all modules. **M** and **Q** are assigned in the code for simplicity.

```
1   module multiplier
2   #(parameter n=4)
3   (
4     input logic start,clock,
5   );
6
7   logic add, shift, nreset, ready, C;
8   logic [n-1:0] Sum;
9   logic [2*n-1:0] AQ;
10  logic [n-1:0] M, Q;
11
12
13  assign M = 10;
14  assign Q = 10;
15
16    // module instances, note: cannot always use implicit mapping syntax here
17    Adder #(.n(n)) a(.A(AQ[(2*n)-1:n]),
18                     .M(M),
19                     .C(C),
20                     .Sum(Sum)); // must use named mapping here
21    Regs #(.n(n)) r(.clock(clock),
22                    .nreset(nreset),
23                    .add(add),
24                    .shift(shift),
25                    .C(C), .Q(Q),
26                    .Sum(Sum),
27                    .AQ(AQ));
28    Sequencer #(.n(n)) s(.clock(clock),
29                         .start(start),
30                         .Q0(AQ[0]),
31                         .add(add),
32                         .shift(shift),
33                         .nreset(nreset),
34                         .ready(ready));
35  endmodule
```

Figure 2.4.1: SV next state code for the sequencer

Figures *2.4.2* (part I) and *2.4.3* (part II) are the simulation results for squaring the number 10 using the multiplier.

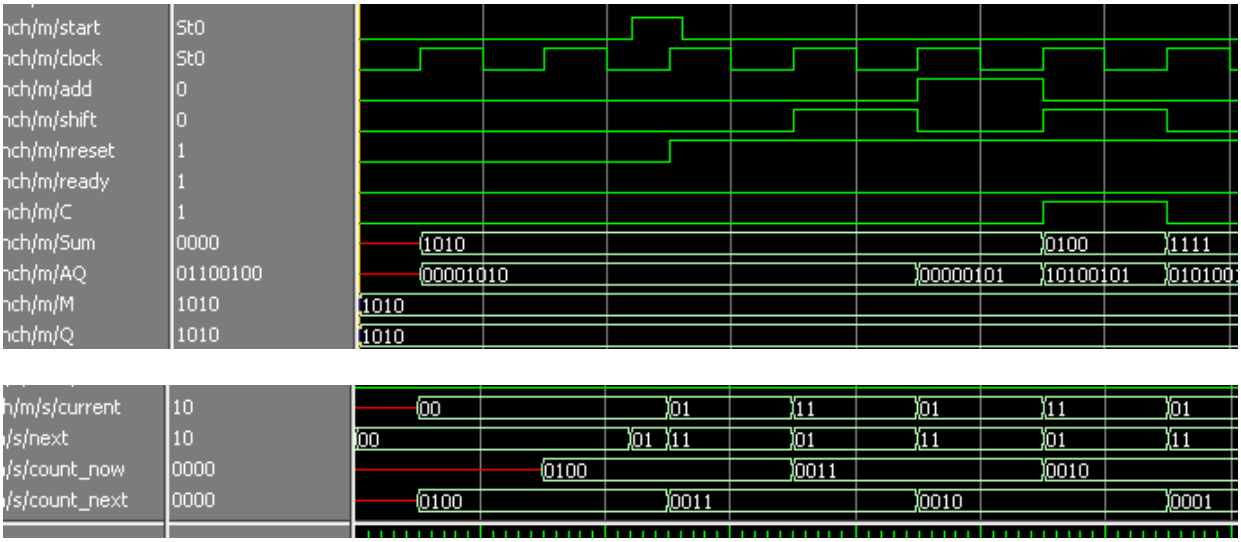

Figure 2.4.2: Simulation of the code from figure *2.4.1* in ModelSim (Part I)

Part I begin with **start** low which means **nreset** is low; everything is therefore reset and the states of **M** and **Q** are placed into **Sum** and **AQ** respectively. This is the case until start is high over the 3rd clock pulse which pushes **nreset** high and beginning the multiplying process. The bits in **AQ** are first shifted then the contents of Sum are copied into **AQ** which is shifted again; introducing a 1 at the MSB because **C** is high. All the time **current** is shuffling between the correct states and **count_now** is being decremented every time a shift occurs.

In part II the system continues to add and shift as expected but eventually the **count_now** reaches zero and at this point **ready** is sent high because the correct value of **M*Q** is held in **AQ**. (10_{10}*10_{10} = 100_{10}) = (1010_2 * 1010_2 = 1100100_2)

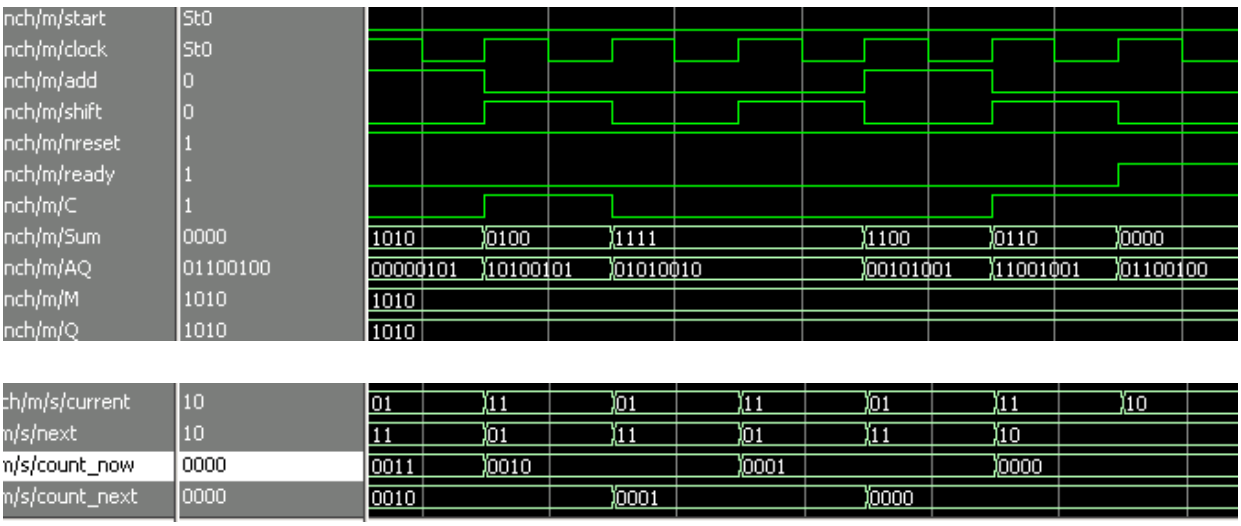

Figure 2.4.3: Simulation of the code from figure *2.4.1* in ModelSim (Part II)

3. Preparation for implementation

3.1 Adding a clock and slowing it down [2]

The MachXO development board comes with an internal clock that can be used to drive sequential logic. Instantiating the clock will produce a signal of around 25MHz. However there are two problems this clock presents…

1. Any testing on intermediate signals outputted would be too quick to record
2. Any switches used to input data will bounce for a few milliseconds after switching so the state may be indeterminate at a rising clock edge

The clock can be slowed by creating a large counter and considering just the MSB.

3.2 Active low

The board also uses active low input/outputs whereas I have considered active high inputs/outputs. The fix is simply to use inverters where required.

4. Synthesis and Testing

All synthesised circuits implemented on the MachXO during the testing process (checked and noted on the design completion form) are held in *8. Appendix.*

4.1 Combination logic (the adder module)

Combinational logic can simply be placed straight on the CPLD but using inverters (as discussed in *3.2 Active low*) to display the output on the boards LEDs.

4.2 Sequential logic (the other modules)

Creating an instance of the clock and the counter allowed me produce a timing signal for each module. The input/output bits were also flipped to become active low.

5. Expanding to an 8-bit multiplier

Using parameters throughout the system design made it simple for me to upgrade to 8-bits however proving functionality on the MachXO required a small adaption to the encapsulation module. See *8.5 The 8-bit Multiplier* for synthesis diagram.

The MachXO development board has just 8 output LEDs but when multiplying two 8-bit numbers the result could be as large as 16 bits. The solution was to multiplex the LEDs and use one switch to select between the first half of the output and the second half of the output (figure *5.1*).

```
22
23    always_comb
24    begin
25      if(selection)
26        leds = ~AQ[7:0];
27      else
28        leds = ~AQ[15:8];
29    end
30
```

Figure 5.1: Adaption to encapsulation module for 8-bit multiplier

6. Combining ADD and SHIFT in a single state

I was unable to successfully implement this on the MachXO in the laboratory but I did manage to write successfully simulating code to combine adding and shifting in one state. I first changed the sequencer to consider **count_next** rather than the**count_now**. Then using **Q0** I created an "if statement" that sends either **AS** (a combination of add and shift) or **SHIFT** high.

```
47      process:
48        begin
49          nreset = 1;
50          ready = 0;
51          count_next = count_now - 1;
52          if(count_next == 0)//Consider the next count for next state
53            next = stopped;
54          else
55            next = process;
56         if(Q0)//Q0 determines wether to add and shift
57            begin
58              AS = 1;//Ass and Shift
59              shift = 0;
60            end
61          else
62            begin
63              AS = 0;
64              shift = 1;
65            end
66        end
```

Figure 6.1: The sequencer now considers shift and AS in just one state

The register also needed changing so outputs from the adder is shifted straight away rather than placed into registers and then shifted with the next clock cycle (figure *6.2*).

```
10  always_ff @ (posedge clock)
11    if(~nreset) // clear C,A and load Q
12      begin
13        AQ[(2*n)-1:n] <= 0;
14        AQ[n-1:0] <= Q; // load multiplier into Q
15      end
16    else
17      begin
18        if (AS) // store C,Sum and AQ all in AQ
19          begin
20            AQ <= {C,Sum,AQ[n-1:1]};
21          end
22        else
23          begin
24          if (shift) // shift A,Q
25            begin
26              // use concatenation to implement shift: Carry into MSB of A
27              // and AQ by one bit to right: AQ[0] <= AQ[1], AQ[1] <= AQ[2]; ?
28              // note also that Creg must be cleared
29              AQ <= {1'b0,AQ[(2*n)-1:1]};
30            end
31          end
32      end
33  endmodule
```

Figure 6.2: Changing the registers to add and shift with one assignment

Figure *6.3* shows the same two binary numbers being multiplied as I simulated in section *2.4* but this time adding and shifting have been combined into one state and therefore the same operation has been completed in 4 less clock pulses.

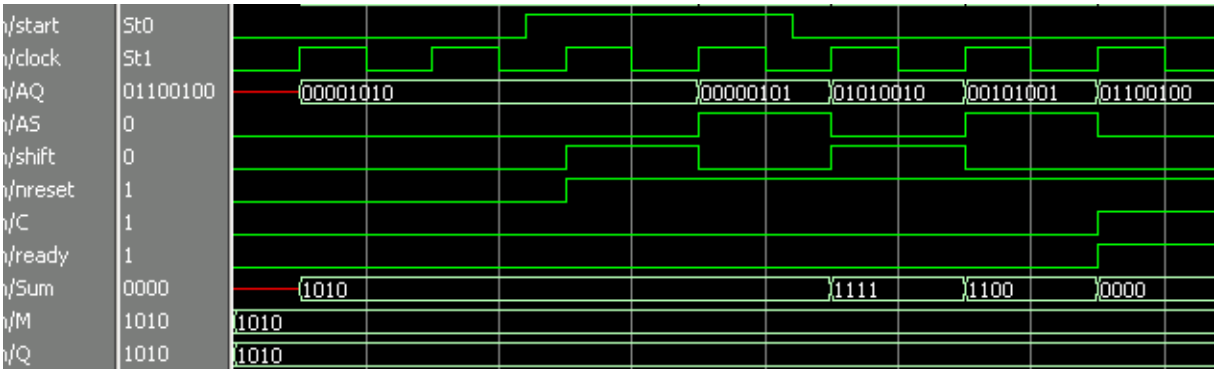

Figure 6.3: Simulation of a multiplier with ADD and SHIFT combined in one state

7. Conclusion

Everything I tested in laboratory functioned as expected with only a few minor errors during synthesis. When synthesising the sequencer Synplify gave a warning that it “did not infer combinational logic” but after assigning to all states in every case block and adding a default block this problem was fixed. Active low logic on the MaxhXO board was also a slight problem for my modules that didn’t adhere to this but once again easily fixed just by adapting the code to invert signals.

8. Appendix

All synthesised circuits in this section were successfully implemented on the MachXO.

8.1 Adder

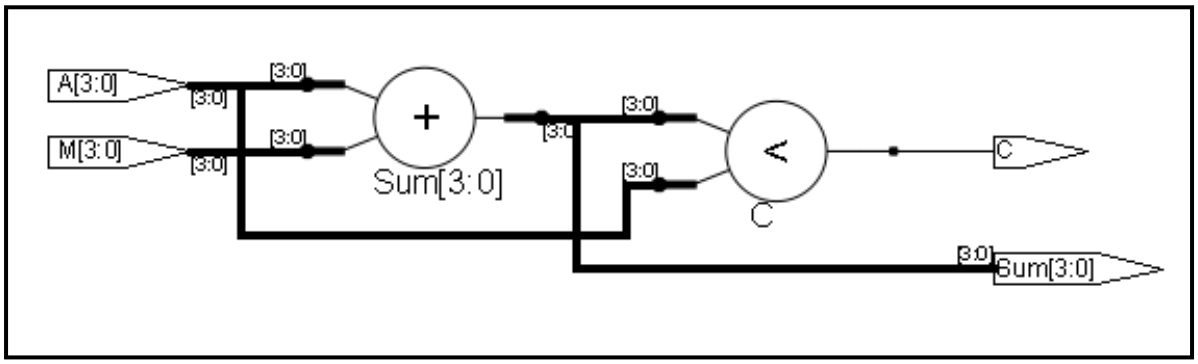

Figure 6.1: Synthesised circuit for the Adder

8.2 Registers

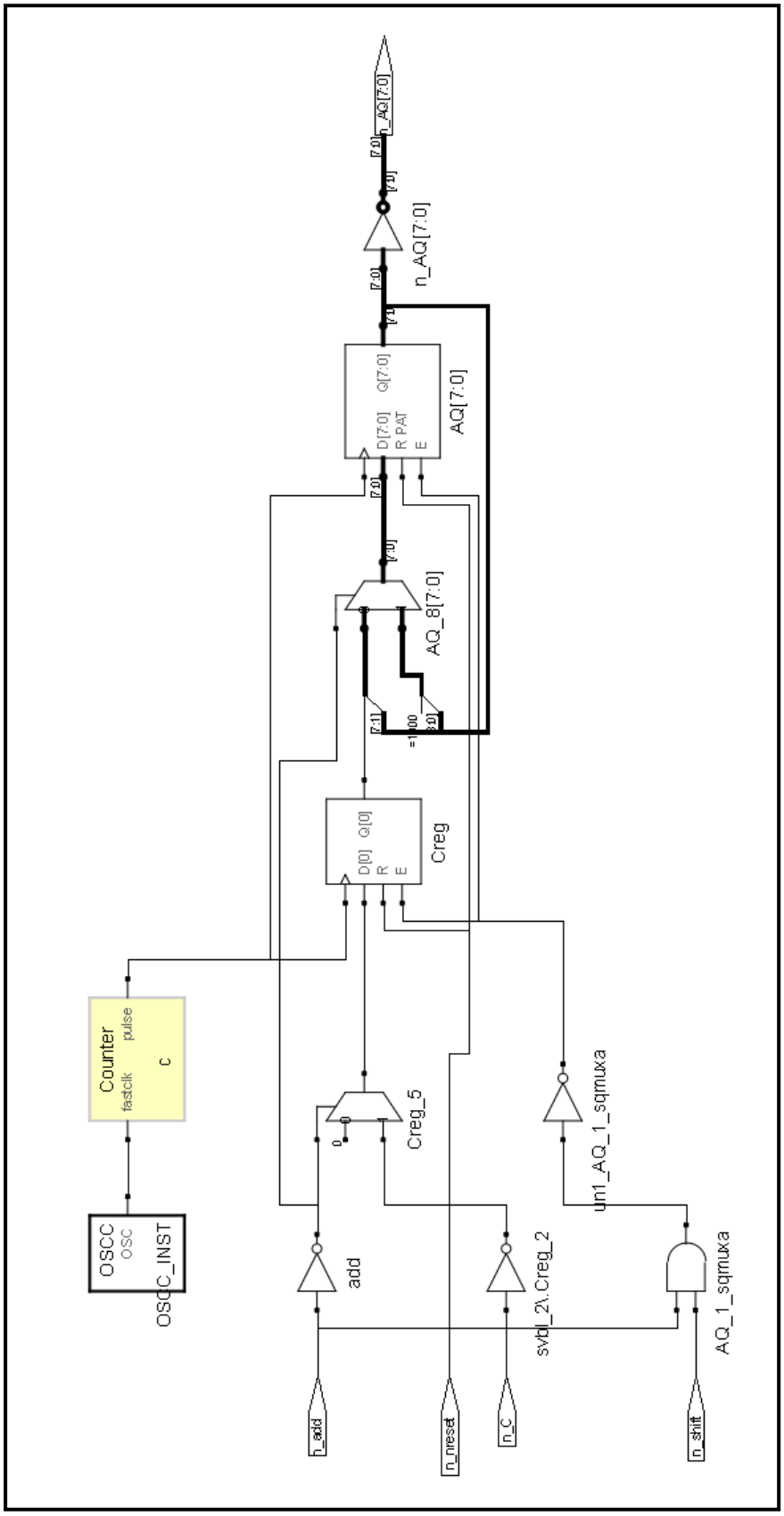

Figure 6.2: Synthesised circuit for the Registers

8.3 Sequencer

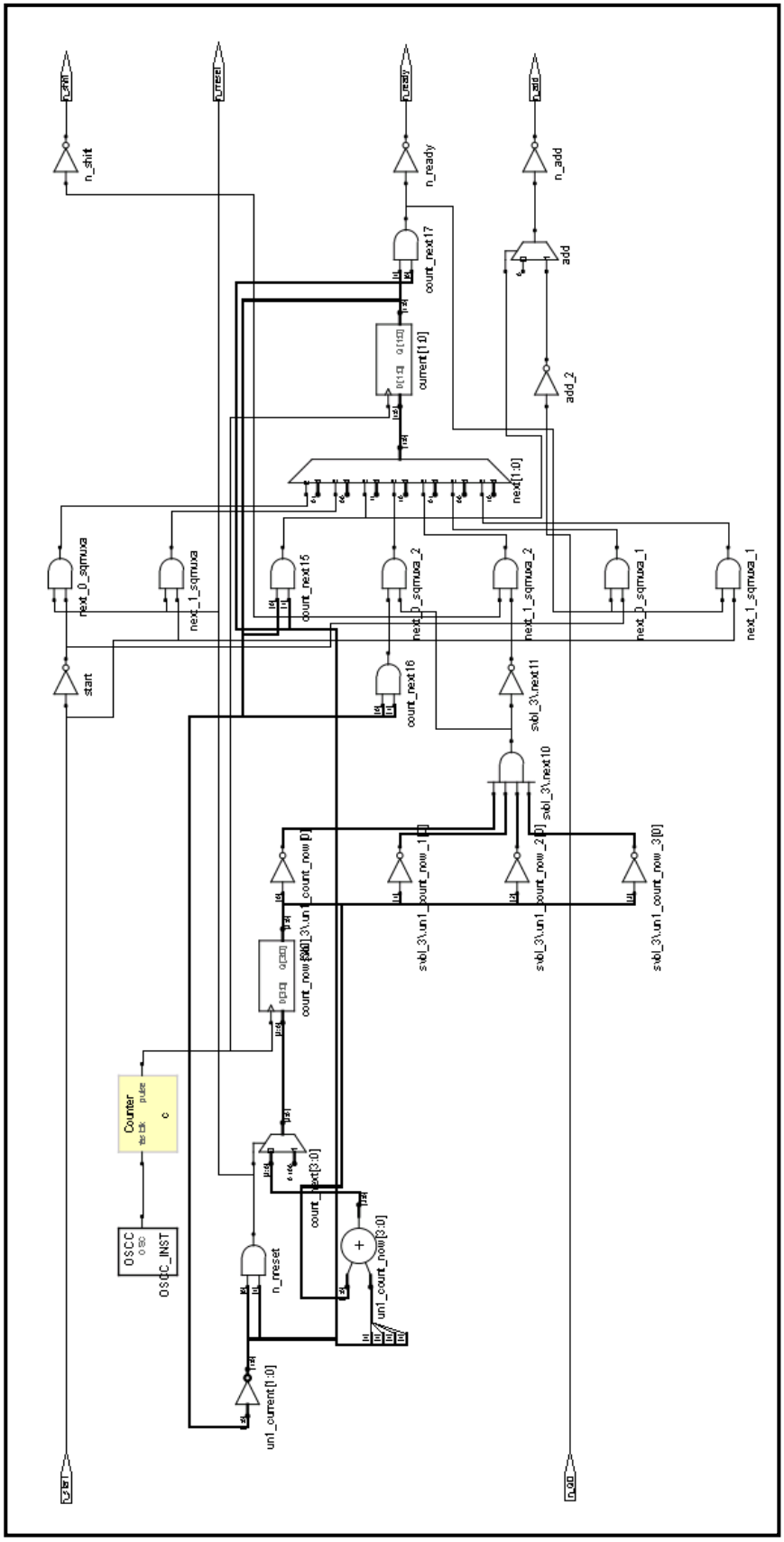

Figure 6.3: Synthesised circuit for the Sequencer

8.4 The 4-bit Multiplier

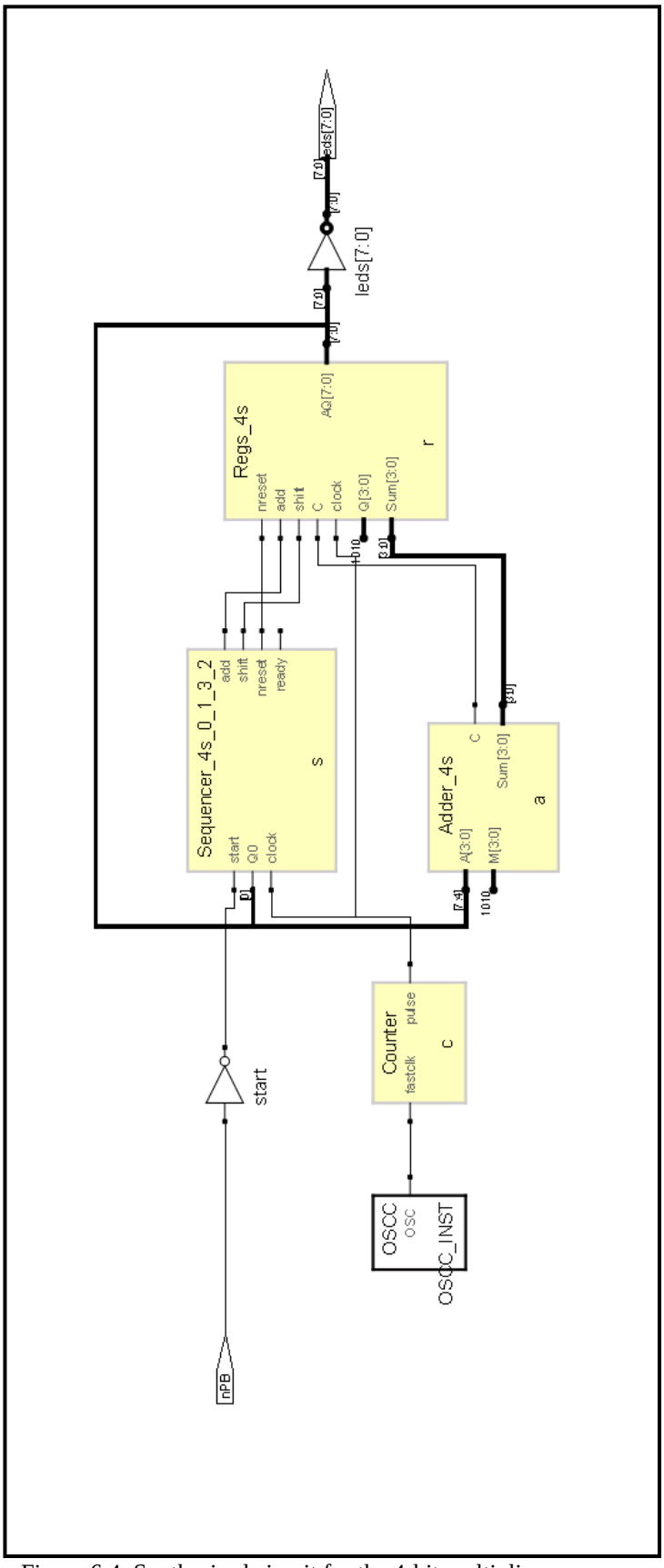

Figure 6.4: Synthesised circuit for the 4-bit multiplier

8.5 The 8-bit Multiplier

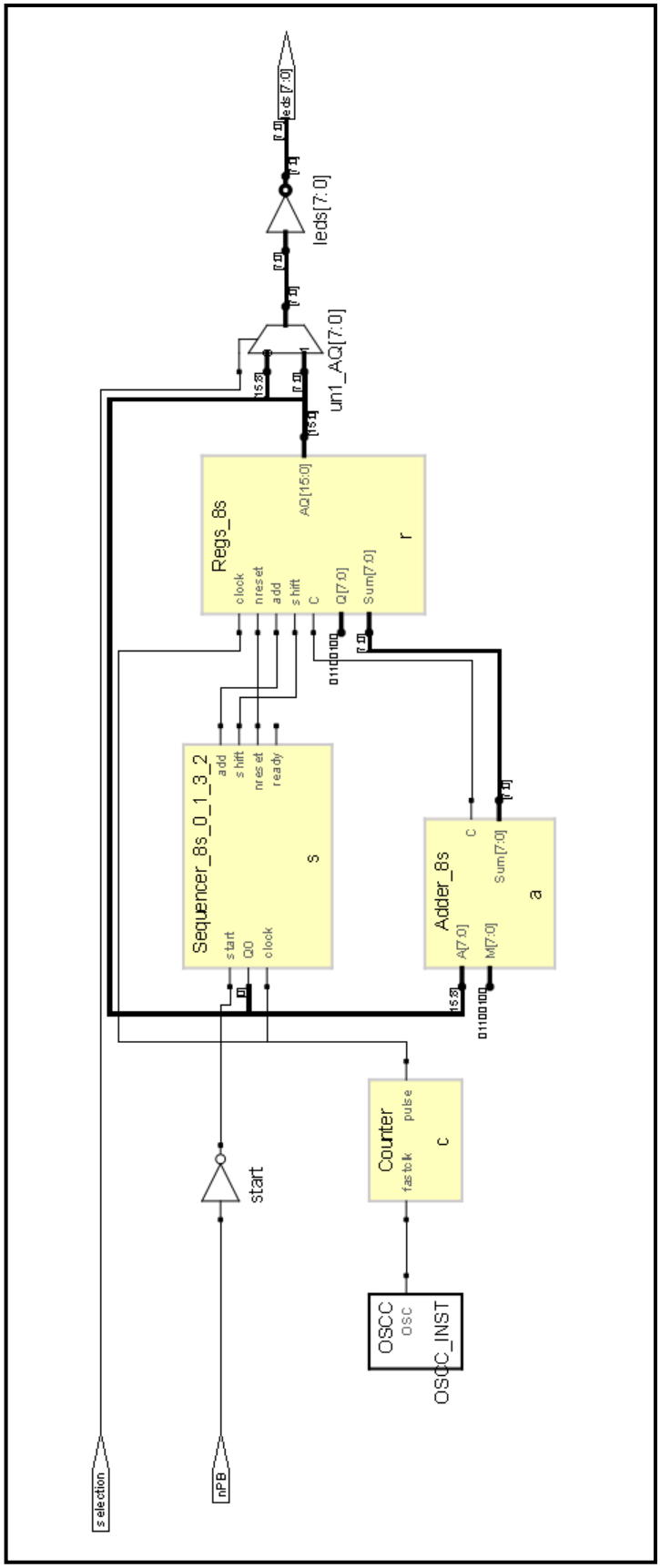

Figure 6.5: Synthesised circuit for the 8-bit multiplier

9. References

[1] Lab brief, *D1 SystemVerilog Design of a Sequential Multiplier*, Dr Tom J Kazmierski
Online: https://secure.ecs.soton.ac.uk/notes/elec2014/D1_2011/D1Instructions2011.pdf

[2] *MachXO Mini Development kit – synthesis and programming walk-through*,
Dr Tom J Kazmierski
Online: https://secure.ecs.soton.ac.uk/notes/elec2014/D1_2011/MachXO_CPLD.pdf