

ELEC2202 2013/14

D1 Design Completion Form

Name: Yubo Zhi

Username: yz39g13

Sequencer simulation shows correct functionality	B.H
Whole system simulation shows correct functionality	B.H
Sequencer for 4-bit multiplication implemented on MachXO2, goes through sequence of states, stops and displays READY	TCG
4-bit adder implemented and shows correct functionality	TCG
C,A,Q registers for 4-bit operation implemented and show correct functionality	TCG
Whole 4-bit system implemented and shows correct functionality	TCG
ADD and SHIFT combined in a single state	TCG
Extension to 8 bits implemented and shows correct functionality	TCG

Supervisor's signature 

Date 21/11/14

Formal report to be submitted through C-Bass/ECS front desk, by 16:00, one calendar week after the scheduled laboratory session.

Include a copy of this completion form with your formal report.

ELEC2221 D1 – Design and test of a sequential multiplier

Yubo Zhi
yz39g13
BEng Electronic Engineering
Professor Alun S Vaughan
Group 29
November 21st 2014

ABSTRACT: Design an n-bit unsigned multiplier, where $n \geq 4$, by using SHIFT-ADD sequential algorithm with SystemVerilog, then implement on MachXO2 Pico Kit. Majority of coding and simulations are done before the lab, then finish the encapsulation modules and implement each part of the design on the CPLD during the lab. Extensions achieved are combined SHIFT and ADD states for faster operation, bi-directional shared data port for multiplier input and output, push button signal debouncing and combinational multiplier.

1. Adder design, simulation and synthesis

The provided adder code [1] works well for any bit, by specify the parameter n, so it is not changed.

1.1 Adder simulation

Adder testbench:

```
module adder_test;
parameter n = 4;

logic [n - 1:0] A, M, Sum;
logic C;
adder #(n(n)) a0 (.*);

initial
begin
    A = 'b0;
    M = 'b0;
    do
    begin
        #5ns A++;
        #5ns M = A;
    end
    while (A != 'b0);
end

endmodule
```

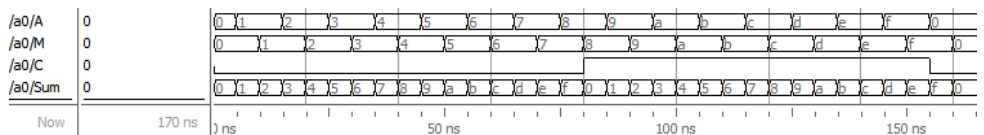


Figure 1 Modelsim 4-bit adder simulation, shows all possible outputs

2. Register design, simulation and synthesis

The code for register is modified. The signal add is changed to add_shift, which modified to do both add and shift operations in 1 clock cycle, thus Creg register becomes unnecessary, so removed.

Register modified code:

```
module register #(parameter n = 4)
(input logic clock, reset, add_shift, shift, C,
input logic[n - 1:0] Qin, Sum, output logic[n * 2 - 1:0] AQ);

always_ff @ (posedge clock)
begin
    if (reset) // clear C,A and load Q
    begin
        AQ[n * 2 - 1:n] <= 0;
        AQ[n - 1:0] <= Qin; // load multiplier into Q
    end
end
```

```

end
else if (add_shift)          // add, then shift
    AQ <= {C,Sum,AQ[n - 1:1]};
else if (shift)             // shift AQ
    AQ <= {1'b0,AQ[n * 2 - 1:1]};
endmodule

```

2.1 Simulation of registers

Register testbench:

```

module register_test;
parameter n = 4;

logic clock, reset, add_shift, shift, C;
logic[n - 1:0] Qin, Sum;
logic[n * 2 - 1:0] AQ;
register #(n(n)) r0 (.*);

// Clock
initial
begin
    clock = 1'b0;
    forever #5ns clock = ~clock;
end

// Test sequence
initial
begin
    reset = 'b0;
    add_shift = 'b0;
    shift = 'b0;
    C = 'b1;
    Qin = 'hE5;
    Sum = 'h47;
    #10ns reset = 'b1;
    #10ns reset = 'b0;
    #10ns shift = 'b1;
    #10ns shift = 'b0;
    #10ns add_shift = 'b1;
    #10ns add_shift = 'b0;
    #10ns reset = 'b1;
    #10ns reset = 'b0;
end
endmodule

```

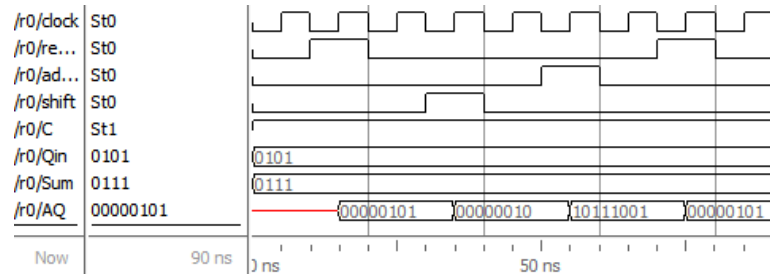


Figure 2 Modelsim 4-bit register simulation.

Qin loaded at 15ns by reset signal, AQ shifted to right 1 bit at 25ns by shift signal, C and Sum loaded to highest 4 bits of AQ then shifted to right 1 bit at 55ns by add_shift signal.

3. Sequencer design, simulation and synthesis

Instead of separate add and shift states, these two steps are combined into one shifting state, and add signal is changed to add_shift, implemented in register. Therefore in shifting state, the sequencer check if Q0 is 1 than assert add_shift signal, otherwise assert shift signal.

Sequencer code:

```

module sequencer #(parameter n = 4)
(input logic start, clock, Q0,
output logic add_shift, shift, ready, reset);

enum logic [1:0] {idle, shifting, stopped} present, next;
logic [$clog2(n) - 1:0] count, next_count;

always_ff @(posedge clock)
begin
    present <= next;
    count <= next_count;
end

always_comb
begin
    add_shift = 1'b0;
    shift = 1'b0;
    ready = 1'b0;
    reset = 1'b0;
end

```

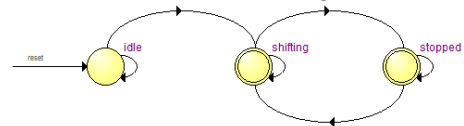


Figure 3 Sequencer state machine diagram, produced from Quartus

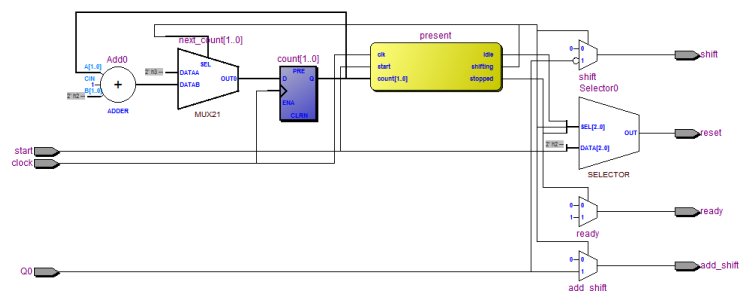


Figure 4 Sequencer RTL diagram, produced from Quartus

```

next = present;
next_count = count;
case (present)
idle:      // State after reset
begin
    reset = 1'b1;
    next_count = n - 1;
    if (start)
        next = shifting;
end
shifting:  // Shifting, n cycle
begin
    next_count = count - 1;
    if (Q0)
        add_shift = 1'b1;
    else
        shift = 1'b1;
    if (count == 0)
        next = stopped;
end
stopped:   // Finished
begin
    ready = 1'b1;
    next_count = n - 1;
    if (start)
    begin
        reset = 1'b1;
        next = shifting;
    end
end
default:
    next = idle;
endcase
end
endmodule

```

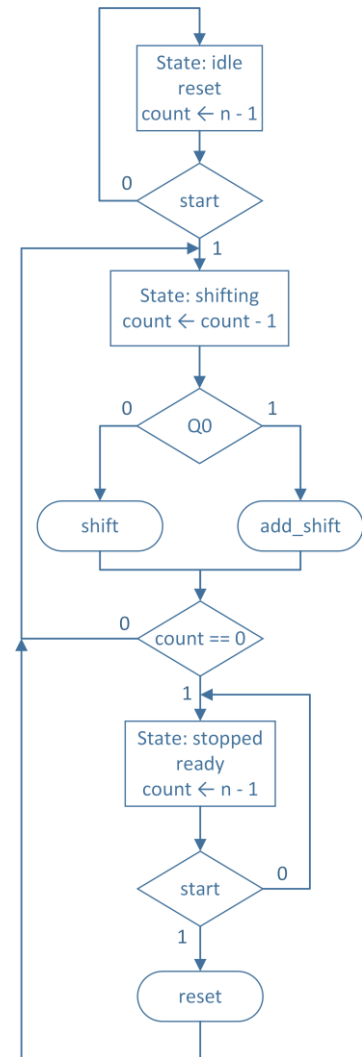


Figure 5 Sequencer ASM chart

3.1 Simulation of sequencer

Sequencer testbench:

```

module sequencer_test;

parameter n = 4;

logic start, clock, Q0;
logic add_shift, shift, ready, reset;

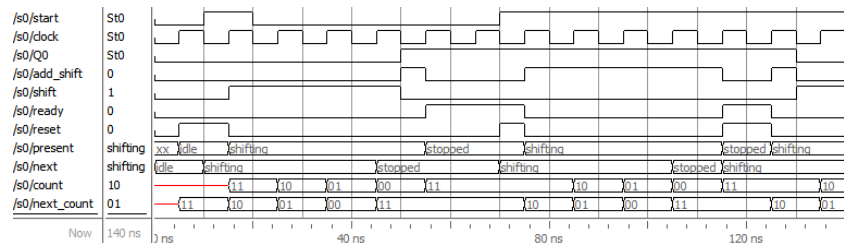
sequencer #(n(n)) s0 (.*) ;

// Clock
initial
begin
    clock = 1'b0;
    forever #5ns clock = ~clock;
end

// Test sequence
initial
begin
    start = 'b0;
    Q0 = 'b0;
    #10ns start = 'b1;
    #10ns start = 'b0;
    #10ns Q0 = 'b0;
    #20ns Q0 = 'b1;
    #20ns start = 'b1;
    #40ns Q0 = 'b1;
    #20ns Q0 = 'b0;
    #10ns start = 'b0;
end

endmodule

```



4. Multiplier design, simulation and synthesis

For be able to input and output n-bit data, I designed a bi-directional data port, with 2-bit port function selection, and two n-bit register for store M and Q_{in}. Therefore, multiplicand M will be loaded from data port at function '00', multiplier Q_{in} will be loaded from data port at function '01', lower n-bit from result AQ will output at '10', higher n-bit from result AQ will output at '11'. For avoid bus contention, an output enable signal OE is also added.

There is a debounce module for debounce start signal, if it is signalled by a push button.

Some compiler directives added to select from combinational multiplier or sequencer multiplier.

Multiplier code:

```
module multiplier #(parameter n = 4, freq = 3330000)
    (input logic startPB, input logic [1:0] funcn,
     input logic oe, output logic ready, inout [n - 1:0] data);

    /// Internal Oscillator 3.33MHz
    logic clock;
    defparam OSCH_inst.NOM_FREQ = "3.33";
    OSCH OSCH_inst (
        .STDBY(1'b0),           // 0=Enabled, 1=Disabled also Disabled with Bandgap=OFF
        .OSC(osc_clk),
        .SEDSTDBY()             // this signal is not required if not using SED
    );
    //counter #(n(24)) c(.);    // produces slow clock
    assign clock = osc_clk;

    /// Debounce
    logic start;
    debounce #(n(freq / 1000)) d0(.clk(osc_clk), .in(~startPB), .out(start));

    /// Blocks
    logic C, reset, shift, add_shift;
    logic [n - 1:0] Sum, M, Qin;
    logic [n * 2 - 1:0] AQ;

    //`define combinational
    `ifndef combinational
        adder #(n(n)) A(.A(AQ[n * 2 - 1:n]), .*);
        register #(n(n)) R(.);
        sequencer #(n(n)) S(.Q0(AQ[0]), .*);
    `else
        combmultiplier #(n(n)) c0 (.A(M), .B(Qin), .Q(AQ));
        assign ready = 'b1;
    `endif

    /// Port
    assign data = oe ? (funcn == 2'b10 ? AQ[n - 1:0] : (funcn == 2'b11 ? AQ[n * 2 - 1:n] :
    'bz)) : 'bz;

    always_ff @(posedge osc_clk)
        case (funcn)
            2'b00:
                M <= data;
            2'b01:
                Qin <= data;
            endcase
endmodule
```

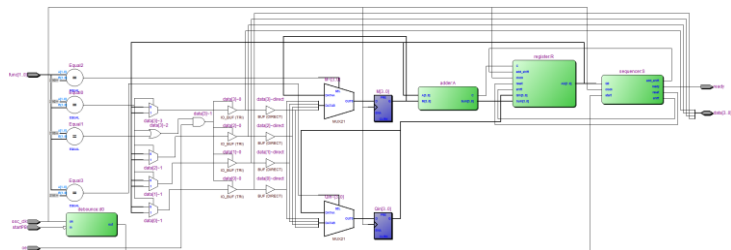


Figure 7 4-bit multiplier RTL diagram, produced from Quartus

4.1 Simulation of multiplier

Sequencer testbench:

```
module test;
    parameter n = 4;

    logic start;
    logic [1:0] funcn;
    logic oe, ready;
    wire [n - 1:0] data;
    multiplier #(n(n)) m(.);
```

```

logic op;
logic [n - 1:0] dataop;
assign data = op ? dataop : 'bz;

```

```

initial
begin
start = 1'b0;
oe = 1'b0;
func = 'b00;
op = 1'b1;
dataop = 'd123;
#10ns func = 'b01;
dataop = 'd234;
#10ns func = 'b11;
op = 1'b0;
start = 1'b1;
#10ns start = 1'b0;
#30ns func = 'b10;
#10ns oe = 1'b1;
#10ns func = 'b11;
#10ns oe = 1'b0;

```

```

#10ns start = 1'b0;
oe = 1'b0;
func = 'b00;
op = 1'b1;
dataop = 'h55;
#10ns func = 'b01;
dataop = 'hAA;
#10ns func = 'b11;
op = 1'b0;
start = 1'b1;
#30ns start = 1'b0;
#10ns func = 'b10;
#10ns oe = 1'b1;
#10ns func = 'b11;
#10ns oe = 1'b0;
end

```

```
endmodule
```

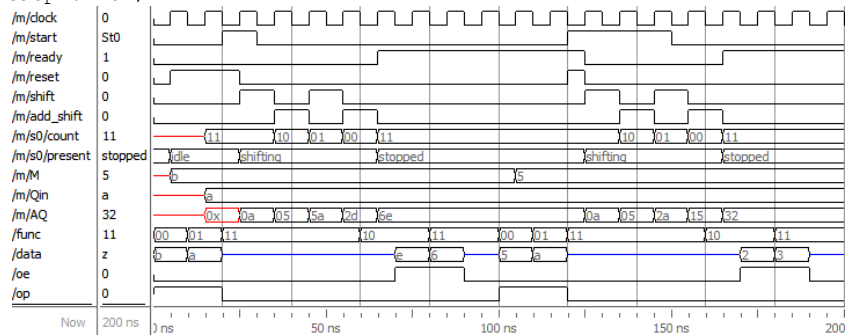


Figure 8 4-bit multiplier Modelsim simulation, bi-directional data input and output.

op is a signal in testbench to control data output from dataop setting signals to data bus, for input M and Qin. First, M loaded into the multiplier through the data bus, by selecting func at function 00. After 1 clock cycle, Qin loaded into the multiplier by using function 01. Change func to function 11, without assert oe signal to prevent M and Qin change during computation. Then assert the start signal, let the multiplier start computation. 8 clock cycles later, ready signal been asserted, means computation finished. Then, by selecting function 10 and assert the oe signal, the lower 8-bit from the AQ result register will appear on the data bus. Selecting function 11 and assert the oe signal will let the higher 8-bit from the AQ result register appear on the data bus. There is another computation cycle in the simulation.

5. Extensions

5.1 Combine add and shift state

Add and shift state combination already done by modifying the register code and combine the two states into a single shifting state in the state machine.

5.2 8-bit multiplier

Building an 8-bit multiplier is easy, as the parameter n to set the bit length is used in every module, so change the parameter n in the encapsulation multiplier module to 8 is the only thing needed to build an 8-bit multiplier.

e.g. the first line of multiplier module definition:

```
module multiplier #(parameter n = 8, freq = 3330000)
```

For simulation, change the paramter line (line 2) in testbench:

```
parameter n = 8;
```

Change following lines in testbench for 4 more computation clock cycles:

Line 27:

```
#70ns func = 'b10;
```

Line 43:

```
#50ns func = 'b10;
```

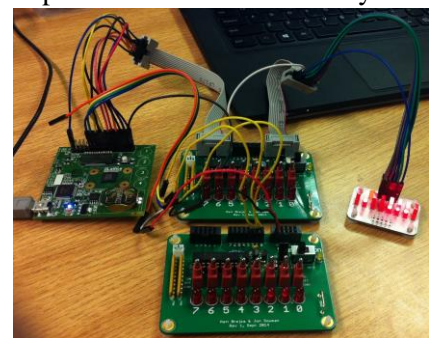


Figure 9 8-bit multiplier implemented on MachXO2 CPLD. Data bus shared for data inputs and outputs. Top switch board with high-z mode for data input, bottom switch board for control signals, LED board for data output.

```
logic a[n][n];
logic c[n][n];
logic s[n][n];

// Generate full adder matrix
genvar x, y;
generate
```

```

        for (y = 0; y < n; y++)
            for (x = 0; x < n; x++)
                fulladder a0 (.A(a[y][x]), .B(A[x] & B[y]), .Cin(x == 0 ? '0 :
c[y][x - 1]), .S(s[y][x]), .Cout(c[y][x]));
    endgenerate

// Connections
generate
    for (x = 0; x < n; x++)
        assign a[0][x] = 'b0;
    for (y = 1; y < n; y++)
        assign a[y][n - 1] = c[y - 1][n - 1];
    for (y = 1; y < n; y++)
        for (x = 0; x < n - 1; x++)
            assign a[y][x] = s[y - 1][x + 1];
    assign Q[n * 2 - 1] = c[n - 1][n - 1];
    for (y = 0; y < n; y++)
        assign Q[y] = s[y][0];
    for (x = 0; x < n - 1; x++)
        assign Q[n + x] = s[n - 1][x + 1];
endgenerate

endmodule

```

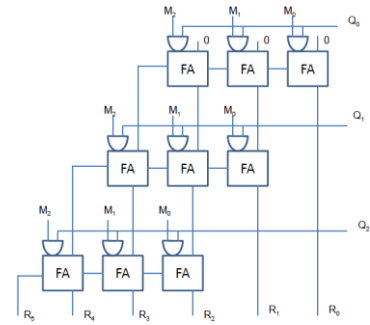


Figure 11 Graphical representation of unsigned combinational multiplier.

The full adder module used in the combinational multiplier:

```

module fulladder (input logic A, B, Cin, output logic S, Cout);

    assign {Cout,S} = A + B + Cin;

endmodule

```

Relative code in multiplier encapsulation module:

```

`define combinational
`ifndef combinational
    adder #(.n(n)) A(.A(AQ[n * 2 - 1:n]), .*);
    register #(.n(n)) R(.);
    sequencer #(.n(n)) S(.Q0(AQ[0]), .*);
`else
    combmultiplier #(.n(n)) c0 (.A(M), .B(Qin), .Q(AQ));
    assign ready = 'b1;
`endif

```

Modelsim testbench:

```

module comb_test;
    parameter n = 8;

    logic start, oe, ready;
    logic [1:0] func;
    wire [n - 1:0] data;
    multiplier #(.n(n)) m(.);

    logic op;
    logic [n - 1:0] dataop;
    assign data = op ? dataop : 'bz;

    initial
    begin
        start = 1'b1;
        oe = 1'b0;
        func = 'b00;
        op = 1'b1;
        dataop = 'd123;
        #10ns func = 'b01;
        dataop = 'd234;
        #10ns func = 'b10;
        op = 1'b0;
        #10ns oe = 1'b1;
        #10ns func = 'b11;
        #10ns oe = 1'b0;

        #10ns func = 'b00;
        op = 1'b1;
        dataop = 'h55;
        #10ns func = 'b01;

```

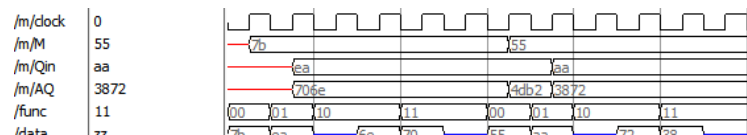


Figure 12 8-bit combinational multiplier Modelsim simulation, bi-directional data input and output.

clock is used by input data storage register M and Qin. start and ready signals are unnecessary for combinational multiplier, and they were set to always 1, so omitted from simulation diagram.

Operations for data input and output are the same as described previously in section 4.1, Figure 8, the difference is combinational multiplier doesn't need to wait for ready signal before getting the results.


```

        dataop = 'hAA;
        #10ns func = 'b10;
        op = 1'b0;
        #10ns oe = 1'b1;
        #10ns func = 'b11;
        #10ns oe = 1'b0;
    end

endmodule

```

6. Conclusion

In this project, a sequential unsigned n-bit multiplier was implemented on MachXO2 Pico Dev Kit. The multiplier uses SHIFT-ADD algorithm, with the SHIFT and ADD states combined for faster operation. A debouncing module is developed for debounce the on-board push button used for signalling the start signal. For be able to input to and output from the multiplier within limited IO ports, a bi-directional data port has also been developed. Furthermore, an n-bit combinational multiplier is also developed, which is very fast compare to sequential multiplier, but will require a large amount of logic for larger n.

By finishing this project, I've learnt using SystemVerilog to design finite state machine, digital hardware design, generate block [2] for generate hardware by for loops, Synthesis and program a MachXO2 device using Lattice Diamond.

To extend it further, a signed multiplier may be a reasonable choice. There are some other combinational multiplier design available as well [3].

7. References

- [1] Tom J Kazmierski (17, Oct. 13). *Suggested SystemVerilog source files* [Online]. Available: <https://secure.ecs.soton.ac.uk/notes/elec2221/tjk2014/D1>
- [2] Jeff Johnson (Jul 18, 2011). *Code templates: Generate for loop* [Online]. Available: <http://www.fpgadeveloper.com/2011/07/code-templates-generate-for-loop.html>
- [3] *Multipliers & Pipelining* [Online]. Available: <http://web.mit.edu/6.111/www/f2011/handouts/L09.pdf>