ECE 440 - Project #10

Collin Heist

14th April 2020

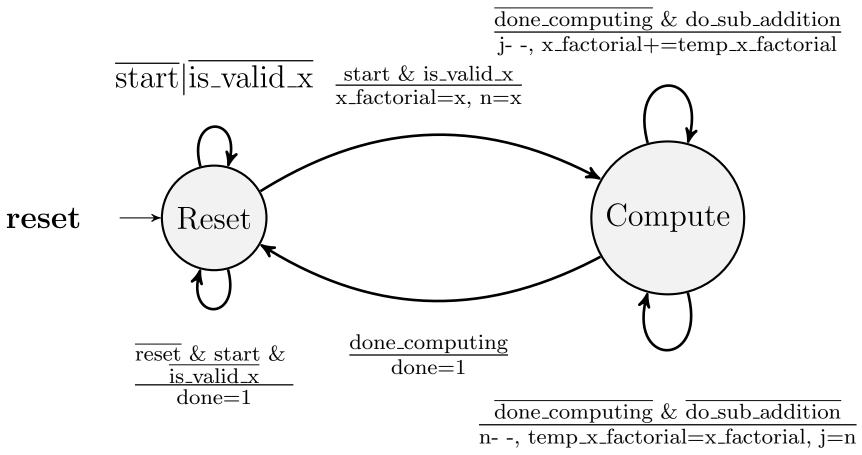
Contents

Figures

Listings

1 Design

I based my factorial (without addition) module on the algorithm we designed on the exam / was shown in class. I started with a three-state FSM, having a reset, compute, and done state. However, I found that by using a Mealy FSM and only asserting the **done** signal as a single-clock pulse (not staying high until a new **start** or **reset**) I was able to remove the done state altogether. The FSM for this module is shown below, in **Figure 1.1**.



$\overline{\text{done\_computing}}$ & do_sub_addition
j- -, x_factorial+=temp_x_factorial
$\overline{\text{start}}|\overline{\text{is\_valid\_x}}$
start & is_valid_x
x_factorial=x, n=x
reset
Reset
Compute
$\overline{\text{reset}}$ & start & is_valid_x
done=1
done_computing
done=1
$\overline{\text{done\_computing}}$ & $\overline{\text{do\_sub\_addition}}$
n- -, temp_x_factorial=x_factorial, j=n

Figure 1.1: Factorial Finite State Machine

Fairly straightforward (as is the algorithm its based on) the FSM only leaves the **compute** state when the module is done computing the result, at which point done is asserted. The two loops on the **compute** state are the two looping operations of the algorithm – either the inner or outer ones. The control signals for this FSM are **done_computing**, **do_sub_addition**, and **is_valid_x**.

These are fairly self-explanatory, but **is_valid_x** just prevents the module from attempting to compute a number with no definable factorial (through this algorithm, i.e. $x \leq 0$), or wasting clock cycles on $x = 1$. If a **start** is initiated but an invalid x is entered, then the **x_factorial** result is automatically assigned to 1 - saving any *computation.* The **do_sub_addition** control signal is used to control the 2nd of the *nested* loops in the computation state. For all values of $j < 1$, no more iterations of the loop are required. In a similar vein, **done_computing** is detected when the number of remaining loops (n) is equal to 1 – meaning all operations are done and **done** can be asserted.

2 Tribulations

With the removal of the SDK portion of this project, I did not struggle with anything in the remaining project. When the SDK part *was* a requirement, I struggled quite a bit with making the clock and reset signals external as a part of the bitstream generation, but otherwise had no hiccups.

3 Simulations

I tested various values of x while designing the module, but for simulations I created a testbench (shown in **Listing 4.2**) to just test $x = 4, 9,$ and 12. The results are shown below:

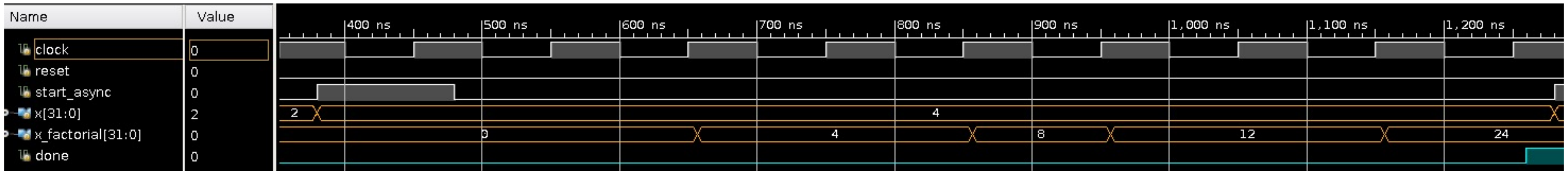


Figure 3.1: Full waveform for the post-synthesis timing simulation, showing $4! = 24$.

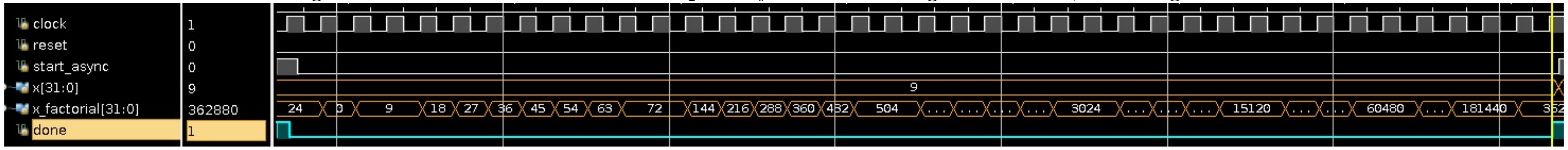


Figure 3.2: Full waveform for the post-synthesis timing simulation, showing $9! = 362,880$.

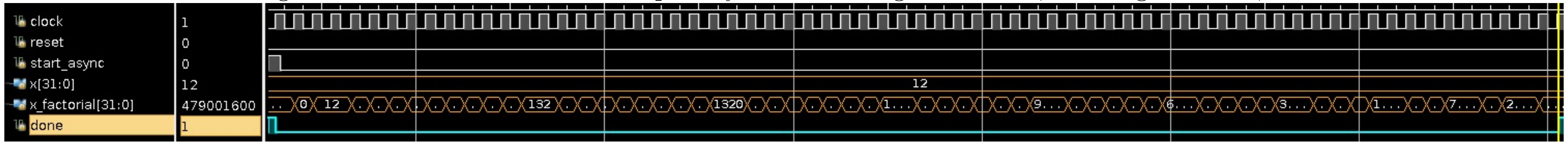


Figure 3.3: Full waveform for the post-synthesis timing simulation, showing $12! = 479,001,600$.

Although the larger factorials (specifically $x = 12$) take fairly long to compute, the end result of **x_factorial** is correct in all the tested cases (as can be seen on the left bar of the final two images).

4 Source Code

Listing 4.1: Factorial Module

```
`timescale 1ns / 1ps

module factorial(clock, reset, start_async, x, x_factorial,
    done);
    input logic clock, reset, start_async;
    input logic [31:0] x;
    output logic [31:0] x_factorial;
    output logic done;

// Pulse-Syncronize the start signal
logic start;
logic [2:0] start_sync_FFs;
always_ff @(posedge clock) begin : start_sync
    if (reset)
        start_sync_FFs <= 0;
    else
        start_sync_FFs <= {start_sync_FFs[1:0], start_async
            };
end : start_sync
assign start = start_sync_FFs[2] ^ start_sync_FFs[1];

// Internal signals
logic done_computing, is_valid_x, do_sub_addition;
logic [31:0] x_in, temp_x_factorial, n, j;
assign is_valid_x = (x > 1);
assign do_sub_addition = (j > 1);
assign done_computing = (n == 1);

// FSM Implementation
typedef enum logic {reset_state, compute} statetype;
statetype state;

always_ff @(posedge clock) begin : fsm_advancement
    if (reset) begin
        state <= reset_state;
        x_factorial <= 0;
        temp_x_factorial <= 0;
        n <= 0;
        j <= 0;
        end
    else begin
        x_in <= start ? x : x_in;
        case (state)
```

```
            reset_state: begin
                state <= (start & is_valid_x) ? compute : reset_state;
                x_factorial <= start ? (is_valid_x ? x : 1) : 0;
                n <= (start & is_valid_x) ? x : 0;
                end
            compute: begin
                state <= done_computing ? reset_state : compute;
                if (~done_computing & do_sub_addition) begin // Do inner loop
                    j <= j - 1;
                    x_factorial <= x_factorial + temp_x_factorial;
                    end
                else if (~done_computing) begin // Do outer loop
                    n <= n - 1;
                    temp_x_factorial <= x_factorial;
                    j <= n - 1;
                    end
                end
        endcase
        end
end : fsm_advancement

// FSM Output
assign done = (state == compute & done_computing & ~reset) | (state == reset_state & start & ~reset & ~is_valid_x);

endmodule : factorial
```

Listing 4.2: Factorial Testbench

```
`timescale 1ns / 1ps

module factorial_testbench();

// Global Parameters
parameter CLOCK_PERIOD = 100;
parameter HOLD_TIME = CLOCK_PERIOD * 0.3;
parameter MAX_SIMULATION_TIME = 200 * CLOCK_PERIOD;

// Internal logic signals
logic clock, reset, start_async, done;
logic [31:0] x, x_factorial;
```

```

// Instantiate the DUT
factorial dut(.*);

// Generate the clock and max-simulation timeout
initial #(MAX_SIMULATION_TIME) $finish;
initial begin clock <= 0; forever #(CLOCK_PERIOD / 2) clock
    = ~clock; end

initial begin : simulation
    reset = 1; start_async = 0; x = 2; #CLOCK_PERIOD;
    @(posedge clock); #HOLD_TIME;

    reset = 0; repeat(2) #CLOCK_PERIOD;

    start_async = 1; x = 4; #CLOCK_PERIOD; start_async = 0;
    //repeat(3) #CLOCK_PERIOD;
    while (~done) #CLOCK_PERIOD;
    $display(start_async, x, x_factorial);

    start_async = 1; x = 9; #CLOCK_PERIOD; start_async = 0;
    //repeat(3) #CLOCK_PERIOD;
    while (~done) #CLOCK_PERIOD;
    $display(start_async, x, x_factorial);

    start_async = 1; x = 12; #CLOCK_PERIOD; start_async =
        0;
    //repeat(3) #CLOCK_PERIOD;
    while (~done) #CLOCK_PERIOD;
    $display(start_async, x, x_factorial);

    repeat(3) #CLOCK_PERIOD; $finish;
end : simulation

endmodule
```