INSTITUTO SUPERIOR TÉCNICO

Design Test and Reliability of Electronic Systems

Project 1 - Digital Controller

Group 2

n° 77973 - Pedro Pina

n° 84779 - Afonso Muralha

Professor Fernando Gonçalves

April 10, 2020

Contents

I. Introduction and Objectives

The goal of this project is to develop a controller that fulfills certain specifications.

The controller must be described in synthesizable Verilog and it must synthesize without any errors or warnings.

To validate the quality of the testbench, the code coverage must be evaluated using the Cadence simulator (Xcelium).

II. Specifications

The controller must synthesize without any errors or warnings.

As for the testbench, the goal should be a code coverage of 100% for the Block and FSM categories.

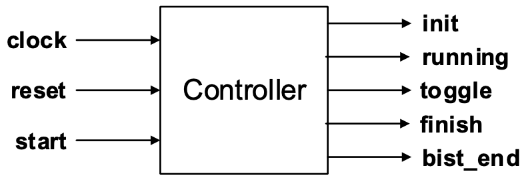


Figure 1: Controller interface

Figure 1 illustrates the controller interface, with the input and output signals. Every signal has certain sensitivity to the edges and logic levels which must be preserved when building the state machine.

Controller inputs

clock: The controller must be sensitive to the rising edges of the clock.

start: After a 0→1 transition in this signal, a new sequence must be initiated (see Figure 2). Its value is captured on the rising edge of the clock. While the full sequence is not completed (bist_end signal set to HIGH), further transitions in the start signal must be ignored.

reset: The reset must be synchronous and active at the logic level ‘1’. This signal is used to restart the state machine and the counters, preparing the controller to start a new sequence. When the reset signal goes LOW and the start signal is HIGH, a new sequence must not be restarted. The controller must wait for a 0→1 transition in the start signal to start a new sequence.

Controller outputs

init: This signal is a pulse with a duration of one clock cycle, indicating the start of a new sequence.

running: This signal must be HIGH for N clock cycles. If the reset signal goes HIGH, then this signal must go LOW.

toggle: This signal must toggle, while the running signal is HIGH. *finish*: This signal is a pulse with a duration of one clock cycle, indicating the end of a sequence.

bist_end: This signal must go HIGH when the sequence is completed. The bist_end signal must go LOW when the reset signal is activated or the start signal has a 0→1 transition.

III. Digital Controller Design

Finite State Machine

The designed finite state machine features 5 states in order to cop with the specifications mentioned above.

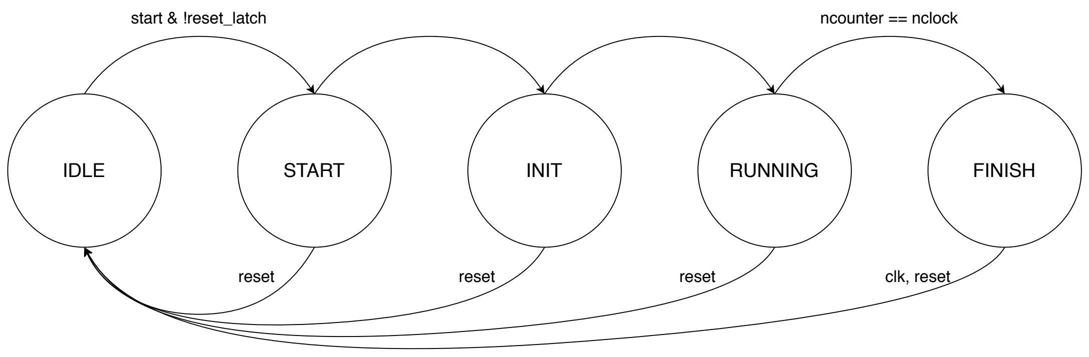


Figure 2: Finite state machine diagram.

Table 1: Output conditions. More detailed information on state description.

Outputs	**Condition**
init	state == INIT
running	(state == RUNNING) & ncounter <nclock + 1
toggle	(state == RUNNING) & toggle_r
finish	state == FINISH
bist_end	(complete) & !(reset | start);

IDLE state:

This state is activated while waiting for a new sequence to be started. When the start conditions are met, the controller changes into the START state.

These start conditions are defined by the following logic expression:

```
start & (state == IDLE) & !reset_latch
```

The main drive for the state change is the start pin becoming enabled but in order to meet requirements, a simple reset latch was implemented, so that if the start pin is already enabled on the falling edge of the reset signal, the controller waits for a full transition of the start signal before changing state. This latch was implemented as follows:

```
always @ (posedge start) begin
       if(start & !(reset))
              reset_latch <= 0;
       else
              reset_latch <= 1;
end
```

Code excerpt 1: Reset latch implementation.

START state:

On the start stage, the sequence is starting. The controller changes to the INIT state on the following positive clock transition.

INIT state:

Whilst on the INIT state, the controller enables the init output and is set to change into RUNNING state on the next positive clock transition.

RUNNING state:

On the RUNNING state, the controller generates the toggle and running outputs for the required number of clocks impulses. The number of clock impulses is set by a parameter and, in order to have a suitable sized register to hold the counter variable, the following primitive was used:

```
parameter NCLOCK = 650; //number of clocks for group 2
reg [$clog2(NCLOCK):0] ncounter; //uses the log2 primitive to calculate a suitable
    size for the register, this is done during compilation.
```

Code excerpt 2: Creation of the register for the NCLOCK parameter.

In this state, the running output is enabled, due to the condition:

```
    assign running = (state == RUNNING) & (ncounter < nclock+1);
```

Code excerpt 3: running output condition.

The toggle output is driven by the state and an auxiliary register that carries the output of a toggle process. This process is also responsible for counting the number of elapsed clocks and iterating them.

```
    assign toggle = (state == RUNNING) & toggle_r;
```

Code excerpt 4: toggle output condition.

```
always @ (posedge clock) begin
       if(reset | (state == FINISH)) begin
              toggle_r <= 0;
              ncounter <= 0;
       end
       else if(state == RUNNING) begin
              if(ncounter < NCLOCK) begin
                     toggle_r = !toggle_r;
              end
              else begin
                     toggle_r <= 0;
              end
              ncounter <= ncounter + 1;
       end
end
```

Code excerpt 5: Toggle generator and counter process.

When the iterating register reaches the determined parameter, the state changes to FINISH.

FINISH state:

Whilst on the FINISH state, the controller enables the finish output and is set to change into IDLE state on the next positive clock transition.

The bist_end signal is triggered by the following condition:

```
assign bist_end = (complete) & !(reset | start);
```

Code excerpt 6: bist_end register drive.

The complete register is used on order to assure a simultaneous transition of the bist_end signal.

```
always @ (negedge finish, posedge start, posedge reset) begin
       if(reset | start)
              complete = 0;
       else
              complete = 1;

end
```

Code excerpt 7: Complete register drive.

III.I Testbench and simulation

In order to test the controller, a test bench was created. This test bench instances the controller modules and features multiple test scenarios that can be set by commenting the non-relevant tests:

```
//===========SET TEST==========
`define normal
`define consequent_test
`define mid_start
`define start_reset
`define mid_reset
```

Code excerpt 8: Testbench test set.

For the following tests, an NCLOCK of 10 was used for easier visualization.

Normal test

Runs one sequence.

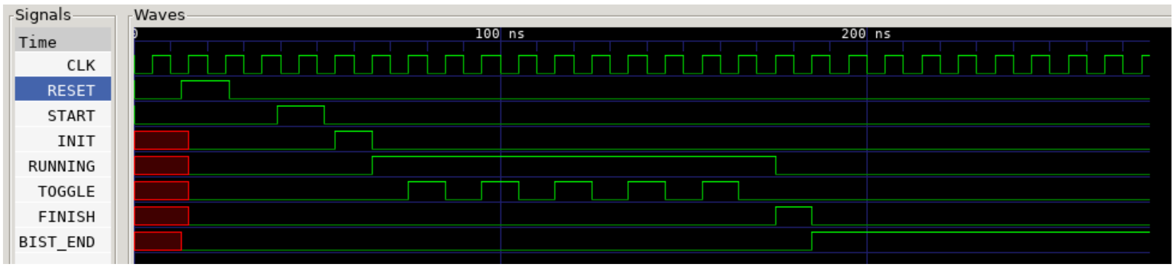


Figure 3: Normal test waveforms.

Consequent test

Runs two sequences in a row:

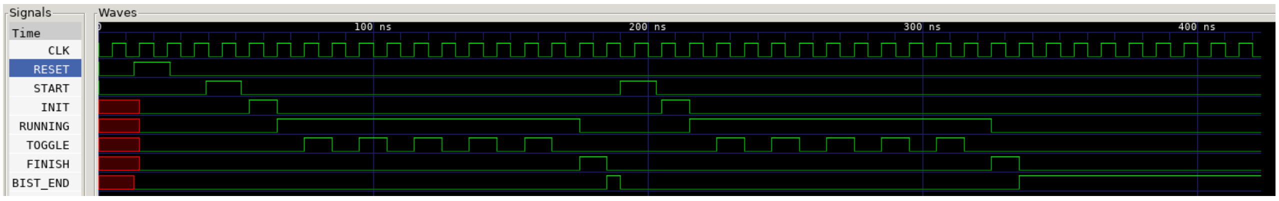


Figure 4: Consequent test waveforms.

Mid-sequence start test

Runs one sequence and toggles the start during the running sequence. The mid-sequence start should not affect the sequence.

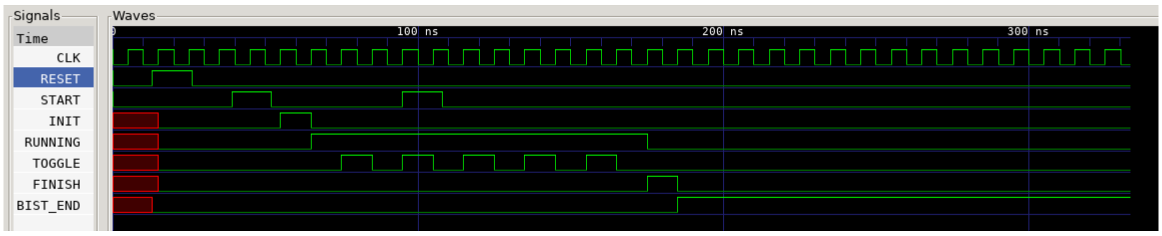


Figure 5: Mid-sequence start test waveforms.

Mid-sequence reset test

Runs one sequence and toggles the reset during the running sequence and starts a new one after that. The mid-sequence reset should abort the sequence and the following sequence should complete without any issues.

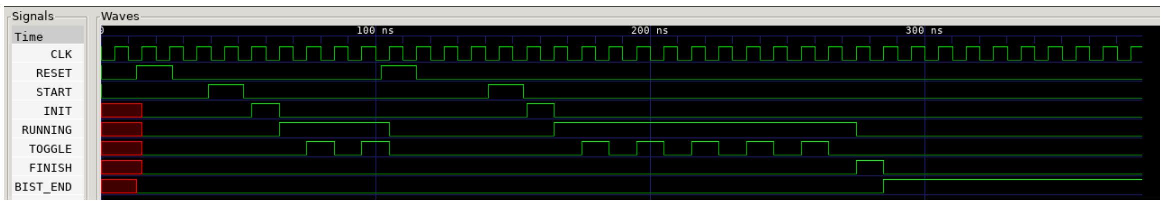


Figure 6: Mid-sequence reset test waveforms.

Start/reset latch test

Runs one sequence and after its completion, enables the start and reset signals and disables the reset signal. In this case the sequence should not start. After that a toggle of the start signal should start a new sequence and it should complete without issues.

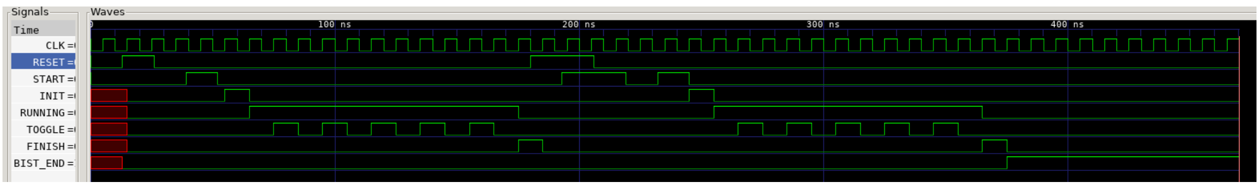


Figure 7: Start/reset latch test waveforms.

In order to validate the running and toggle signals, two counter variables were created on the test bench that are used for measuring then length of the running signal and the number of pulses of the toggle signal.

```
integer pulsecount = 0;
integer runningcount = 0;

always @ (posedge TOGGLE) begin
   pulsecount = pulsecount +1;
end

always @ (posedge CLK) begin
   if(RUNNING)
      runningcount = runningcount + 1;
end
```

Code excerpt 9: Counter variables.

These variables are printed and reseted after a sequence:

```
$display("Number of pulses: %d pulses.",pulsecount);
$display("Time of running: %d clock pulses.",runningcount-1); //-1 because the first
     rising edge of the clock doesent count
pulsecount = 0;
runningcount = 0;
```

Code excerpt 10: Conter variable prints.

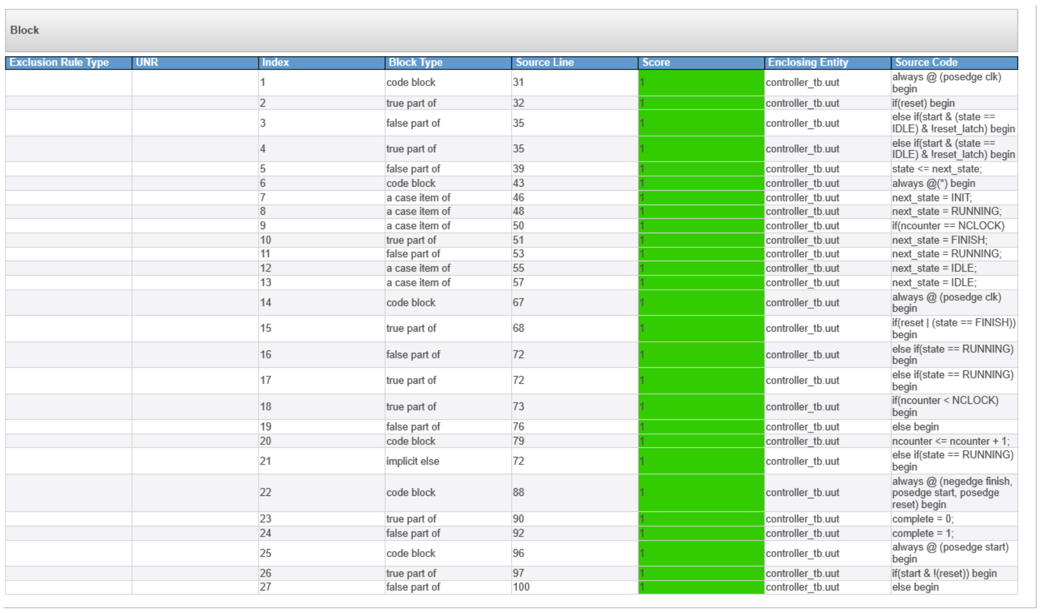
This was tested for multiple NCLOCK values to validate if the method can be trusted for measuring the number of pulses and the length of the running signal. Then, simulating one normal sequence with the NCLOCK set to the group's number (650) we can validate the result with the output of the program:

```
Number of pulses:       325 pulses.
Time of running:        650 clock pulses.
```

Code excerpt 11: Toggle and running validation.

Code Coverage and synthesis validation

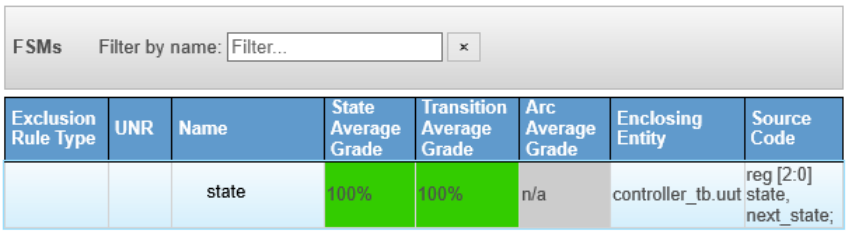
In order to validate the quality of the produced code, the controller module was synthesized on the Vivado IDE and the report can be found alongside the submitted files.
A code coverage and FSM reports were also generated using cadence tools for the controller using NCLOCK = 650.



Block

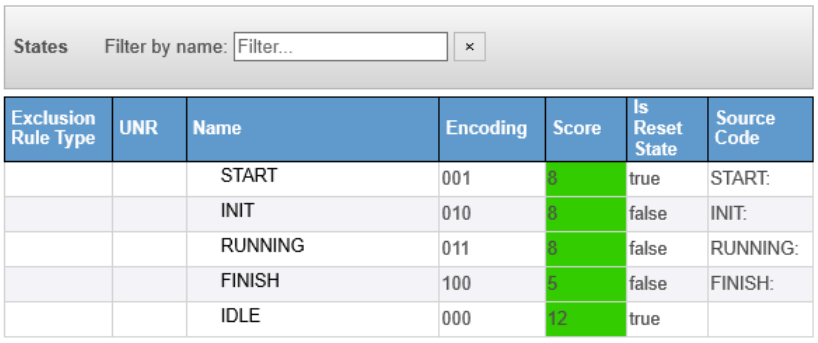
Exclusion Rule Type	UNR	Index	Block Type	Source Line	Score	Enclosing Entity	Source Code
		1	code block	31	1	controller_tb.uut	always @ (posedge clk) begin
		2	true part of	32	1	controller_tb.uut	if(reset) begin
		3	false part of	35	1	controller_tb.uut	else if(start & (state == IDLE) & !reset_latch) begin
		4	true part of	35	1	controller_tb.uut	else if(start & (state == IDLE) & !reset_latch) begin
		5	false part of	39	1	controller_tb.uut	state <= next_state;
		6	code block	43	1	controller_tb.uut	always @(*) begin
		7	a case item of	46	1	controller_tb.uut	next_state = INIT;
		8	a case item of	48	1	controller_tb.uut	next_state = RUNNING;
		9	a case item of	50	1	controller_tb.uut	if(ncounter == NCLOCK)
		10	true part of	51	1	controller_tb.uut	next_state = FINISH;
		11	false part of	53	1	controller_tb.uut	next_state = RUNNING;
		12	a case item of	55	1	controller_tb.uut	next_state = IDLE;
		13	a case item of	57	1	controller_tb.uut	next_state = IDLE;
		14	code block	67	1	controller_tb.uut	always @ (posedge clk) begin
		15	true part of	68	1	controller_tb.uut	if(reset | (state == FINISH)) begin
		16	false part of	72	1	controller_tb.uut	else if(state == RUNNING) begin
		17	true part of	72	1	controller_tb.uut	else if(state == RUNNING) begin
		18	true part of	73	1	controller_tb.uut	if(ncounter < NCLOCK) begin
		19	false part of	76	1	controller_tb.uut	else begin
		20	code block	79	1	controller_tb.uut	ncounter <= ncounter + 1;
		21	implicit else	72	1	controller_tb.uut	else if(state == RUNNING) begin
		22	code block	88	1	controller_tb.uut	always @ (negedge finish, posedge start, posedge reset) begin
		23	true part of	90	1	controller_tb.uut	complete = 0;
		24	false part of	92	1	controller_tb.uut	complete = 1;
		25	code block	96	1	controller_tb.uut	always @ (posedge start) begin
		26	true part of	97	1	controller_tb.uut	if(start & !(reset)) begin
		27	false part of	100	1	controller_tb.uut	else begin

Figure 8: Code coverage report.



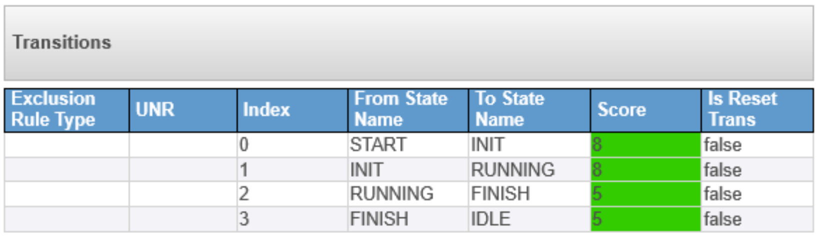
Exclusion Rule Type	UNR	Name	State Average Grade	Transition Average Grade	Arc Average Grade	Enclosing Entity	Source Code
		state	100%	100%	n/a	controller_tb.uut	reg [2:0] state, next_state;

Figure 9: FSM report 1.


States Filter by name: Filter... ×

Exclusion Rule Type	UNR	Name	Encoding	Score	Is Reset State	Source Code
		START	001	8	true	START:
		INIT	010	8	false	INIT:
		RUNNING	011	8	false	RUNNING:
		FINISH	100	5	false	FINISH:
		IDLE	000	12	true	

Figure 10: FSM report 2.


Transitions

Exclusion Rule Type	UNR	Index	From State Name	To State Name	Score	Is Reset Trans
		0	START	INIT	8	false
		1	INIT	RUNNING	8	false
		2	RUNNING	FINISH	5	false
		3	FINISH	IDLE	5	false

Figure 11: FSM report 3.

IV. Conclusion

This project objective was to learn how to program and develop a controller with certain specifications. This meant that, for it to work properly, the ones developing it, needed to have a clear understanding of every input and ouput signals functionality and behaviour - how does a signal influences other signals and how does it develop throw a specific time period.

Therefore, having tools such as the Gtkwave and the Cadence simulator (Xcelium) are important to evaluate the code, which errors and warnings it may have, during the developing stage. In addition to that, having a graphic representation of what has been done can guarantee that the final product is working as intended, with every specification running correctly.

References

[1] Verilog Tutorial (Course slides)
https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382240212/My%20Verilog%20Tutorial.pdf