THE UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

ELEC 402 Assignment 1: SystemVerilog FSM Project

Shidi Xi (90506643)

October 2, 2022

Design

In this project, a finite-state machine (FSM) has been designed and simulated. The design concept and the real-world functionality of the FSM is based upon a Python script that the author has developed during his Co-op at a video surveillance company. Hence, a quick introduction of the script will be given for better understanding of the design of the FSM.

The company's cameras have embedded firmware on them, and for any scenes or videos that a camera is capturing or streaming, the camera's firmware will analyze the amount of motion and details contained in the scene and output two numbers, namely, the motion score and the detail score. In one of the tests that are performed on cameras, videos which deliver specified motion scores and detail scores need to be made by the test engineer. E.g., when this video is played in front of a camera, the camera must give a motion score of say, 5, and a detail score of say, 60. The Python script can automatically generate videos meeting the motion and detail requirements provided by the engineer. It consists of two modules (classes), shown by Figure 1, one is a class containing individual functions of making a video using provided detail and motion parameters. It is analogous to a datapath in digital systems such as the central processing unit. The other class is, not surprisingly, a controller. It calls the function in the datapath, providing detail and motion parameters. The datapath then makes the video, gets the detail and motion scores from the camera's firmware, and feeds the scores back to the controller, which then processes the scores, and generates new parameters. New parameters will produce a video that is closer to the target. The process repeats until the controller receives detail and motion scores that are equal to the targets.

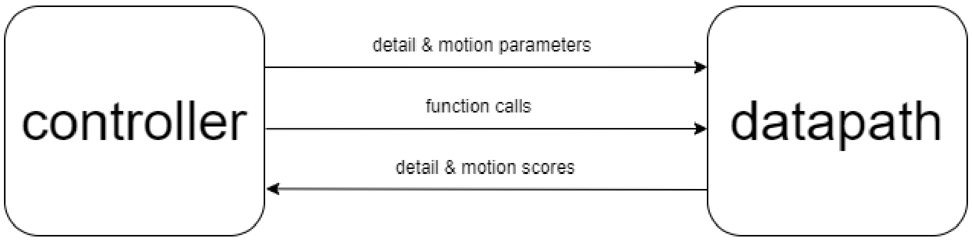


Figure 1: The block diagram showing the two modules (classes) in the Python script.

The FSM designed in this project is in fact the SystemVerilog implementation of the controller class in the Python script. Figure 2 below shows all the inputs and outputs of the FSM module. One can see it has a clock (`clk`) and a reset (`rst`) signal on the input, just like the majority of digital finite-state machines. The 3-bit `aim` signal set the detail target for the FSM, which will be explained later. The 8-bit `motion_score` and `detail_score` signals are from the datapath, if it was to be designed using SystemVerilog. On the output side, there are five 1-bit enable signals. If there is a datapath, these are the enable signals for the functional units inside the datapath. With the analogy to Python, they behave like function calls. For example, the `make_motion` function will make a video containing a moving object and the `check_motion` function will get the motion scores from the camera's firmware. Finally there are the 8-bit `motion_prmt` and `detail_prmt` which are the parameters the controller provides to the datapath to make videos. For instance, larger `motion_prmt` will make the moving object bigger and larger `detail_prmt` will make the video contains more details.

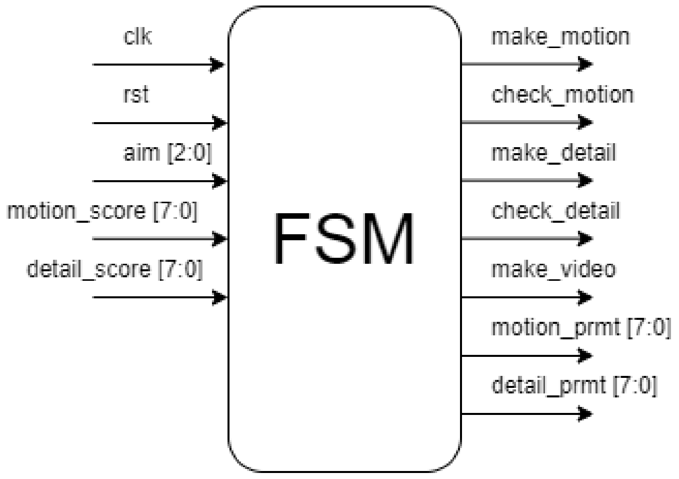


Figure 2: The block diagram of the FSM designed in this project. All I/O ports are labelled.

Figure 3 below shows the state transition diagram of the FSM. There are 14 states in total. Starting from the `reset` state, we initialize `motion_prmt`, `detail_prmt`, `motion_target`, `detail_target` to their intial value, and set all the enable signals to zero. Next we go to `make_motion` if `rst` is not asserted. This state asserts the corresponding enable signal at the output. The following `check_motion` state then de-asserts the previous enable signal and asserts its own correspondent. The `wait_motion_score` state is used to allow one extract cycle for `motion_score` to arrive. Then, the score is compared to `motion_target` . If the score is lower than the target, we go to the `increase_motion` state to raise `motion_prmt`. And, vice versa if the score is higher than the target. After having the new `motion_prmt`, we go back to the state `make_motion`. We stay in this loop until the correct `motion_prmt` has been acquired such that `motion_score` equals to `motion_target`. As the next step we move to compute the correct `detail_prmt`. At the state `decode_detail`, the FSM check its `aim` input. If `aim` is $3'b001$, `detail_target` is set to $'d30$. If `aim` is $3'b010$, `detail_target` is set to $'d45$, and if it is $3'b100$, `detail_target` is $'d60$. Essentially, `aim` is a one-hot select signal, selecting different detail levels one would like to have in the generated video. After having set `detail_target`, the FSM goes to the `make_detail` state, where it outputs the appropriate `motion_prmt` acquired before, together with the initial `detail_prmt`.

One can then see from Figure 3 that the following check-wait-compare-remake loop that the FSM will be staying in, until it has acquired the correct `detail_prmt` such that `detail_score` equals `detail_target` defined at state `decode_detail`. Having both the correct `motion_prmt` and the correct `detail_prmt`, we proceed to `make_video` to generate a video which fulfils `motion_target` and `detail_target`. Then the whole process ends with the `done` state, from where we can go back to the `rst` state to start again if `rst` is asserted.

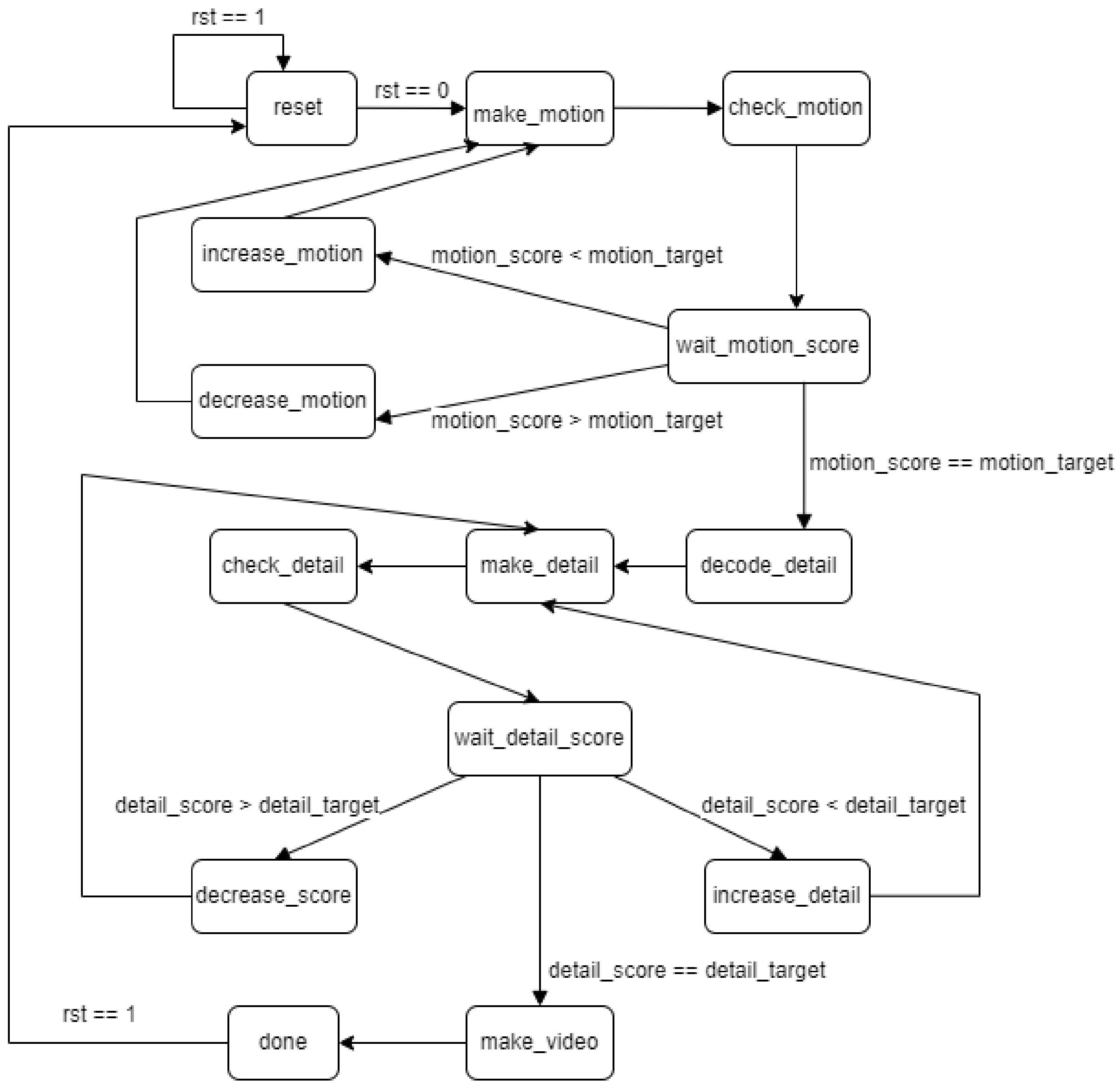


Figure 3: The state transition diagram of the FSM.

Simulation

Figure 4 shows the connection between the testbench module and the device under test (DUT, which is the FSM) it instantiates. Basically the testbench mimics the behavior of the datapath that the FSM is designed to control. It sends `motion_score` and `detail_score` feedback to the FSM, which compares these scores with the targets and makes new `motion_prmt` and `detail_prmt`. Figure 5 shows the simulation waveform of the FSM in ModelSim.

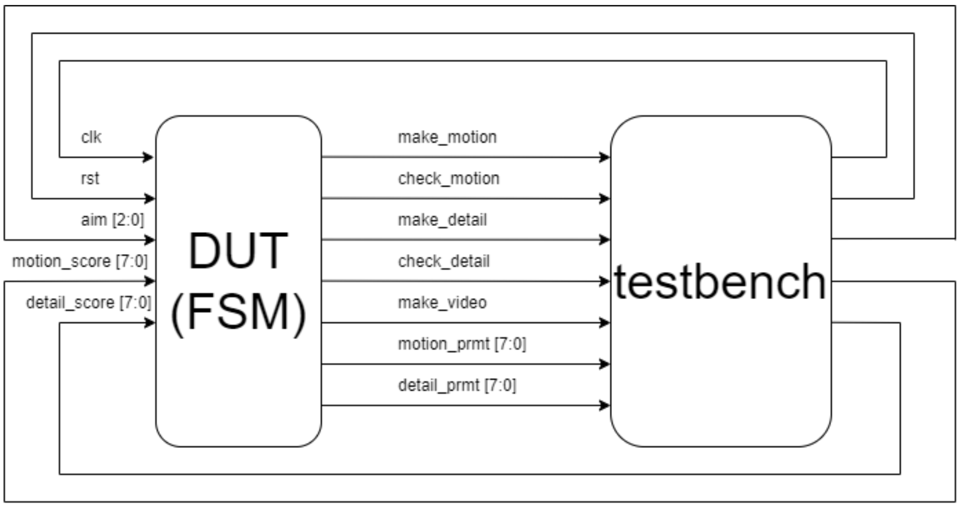
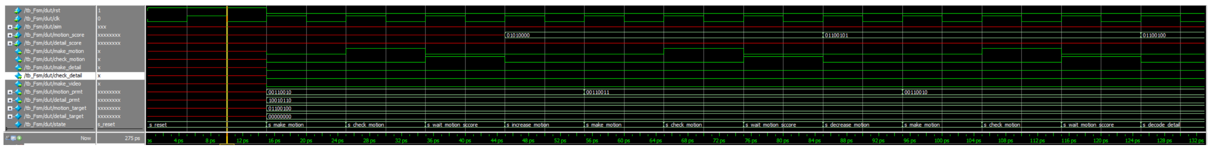
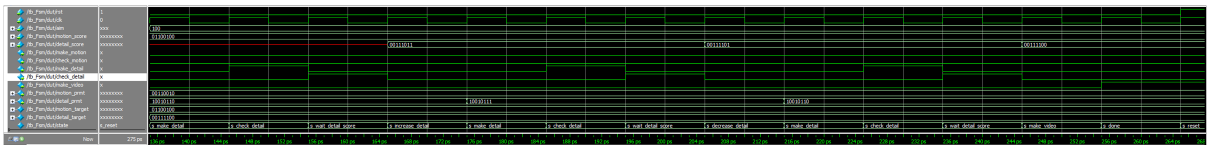


Figure 4: A block diagram showing how the testbench is connected to the DUT (FSM).



(a)



(b)

Figure 5: ModelSim simulation waveform of the FSM.

Appendix A: SystemVerilog code for the FSM design module

```systemverilog
/*
Name: Shidi Xi
Student ID: 90506648
*/

module Fsm (rst, clk, aim, motion_score, detail_score, make_motion, check_motion, make_detail, check_detail, make_video, motion_prmt, detail_prmt);
    input logic rst, clk;
    input logic [2:0] aim;
    input logic [7:0] motion_score, detail_score;
    output logic make_motion, check_motion, make_detail, check_detail, make_video;
    output logic [7:0] motion_prmt, detail_prmt;

    logic [7:0] motion_target, detail_target;

    typedef enum  {s_reset, s_make_motion, s_check_motion, s_wait_motion_sccore, s_increase_motion, s_decrease_motion, s_decode_detail,
    s_make_detail, s_check_detail, s_wait_detail_score, s_increase_detail, s_decrease_detail, s_make_video, s_done} statetype;

    statetype state;



    always_ff @  (posedge clk) begin

        if (rst) begin
            state <= s_reset;
        end

        else begin
            case (state)
                s_reset: begin
                    make_motion <= 0;
                    check_motion <= 0;
                    make_detail <= 0;
                    check_detail <= 0;
                    make_video <= 0;
                    motion_target <= 'd100;
                    motion_prmt <= 'd50;
                    detail_target <= 'd0;
                    detail_prmt <= 'd150;
                    if (~rst) begin
                        state <= s_make_motion;
                    end
                end

                s_make_motion: begin
                    make_motion <= 1;
                    state <= s_check_motion;
                end

                s_check_motion: begin
                    make_motion <= 0;
                    check_motion <= 1;
                    state <= s_wait_motion_sccore;
                end

                s_wait_motion_sccore: begin
                    check_motion <= 0;
                    if (motion_score < motion_target) begin
                        state <= s_increase_motion;
                    end
                    else if (motion_score > motion_target) begin
                        state <= s_decrease_motion;
                    end

                    else if (motion_score == motion_target) begin
                        state <= s_decode_detail;
                    end
                end

                s_increase_motion: begin
                    motion_prmt <= motion_prmt + 1'b1;
                    state <= s_make_motion;
                end

                s_decrease_motion: begin
                    motion_prmt <= motion_prmt - 1'b1;
                    state <= s_make_motion;
                end

                s_decode_detail: begin
                    case (aim)
                        3'b001: detail_target <= 'd30;
                        3'b010: detail_target <= 'd45;
                        3'b100: detail_target <= 'd60;
                    endcase
                    state <= s_make_detail;
                end

                s_make_detail: begin
                    make_detail <= 1;
                    state <= s_check_detail;
                end

                s_check_detail: begin
                    make_detail <= 0;
                    check_detail <= 1;
                    state <= s_wait_detail_score;
                end

                s_wait_detail_score: begin
                    check_detail <= 0;
                    if (detail_score < detail_target) begin
                        state <= s_increase_detail;
                    end
                    else if (detail_score > detail_target) begin
                        state <= s_decrease_detail;
                    end

                    else if (detail_score == detail_target) begin
                        state <= s_make_video;
                    end
                end

                s_increase_detail: begin
                    detail_prmt <= detail_prmt + 1'b1;
                    state <= s_make_detail;
                end

                s_decrease_detail: begin
                    detail_prmt <= detail_prmt - 1'b1;
                    state <= s_make_detail;
                end

                s_make_video: begin
                    make_video <= 1;
                    state <= s_done;
                end

                s_done: begin
                    make_video <= 0;
                    if (rst) begin
                        state <= s_reset;
                    end
                end
            endcase
        end
    end

endmodule
```

Appendix B: SystemVerilog code for the FSM testbench

```
/*
Name: Shidi Xi
Student ID: 90506648
*/

module tb_Fsm ();
    logic rst, clk;
    logic [2:0] aim;
    logic [7:0] motion_score, detail_score;
    logic make_motion, check_motion, make_detail, check_detail,
   make_video;
    logic [7:0] motion_prmt, detail_prmt;

    typedef enum {s_reset, s_make_motion, s_check_motion,
   s_wait_motion_sccore, s_increase_motion, s_decrease_motion,
   s_decode_detail,
    s_make_detail, s_check_detail, s_wait_detail_score,
   s_increase_detail, s_decrease_detail, s_make_video, s_done}
   statetype;

    Fsm dut (rst, clk, aim, motion_score, detail_score, make_motion,
   check_motion, make_detail, check_detail, make_video, motion_prmt,
   detail_prmt); // Instantiate DUT.

    initial begin // Genreate the clock signal.
        clk = 0; #5;
        forever begin
            clk = 1; #5;
            clk = 0; #5;
        end
    end

    initial begin
        rst = 1; #15;
        assert (dut.state === s_reset) else $error ("reset failed."); // When rst is asserted, we should go to the reset state.
        assert (make_motion === 1’b0 && check_motion === 1’b0 && make_detail === 1’b0 && check_detail === 1’b0 && make_video === 1’b0 ) else $error ("Enable signals reset failed."); // Check if all enable signals are 0.
        assert (dut.motion_target === ’d100 && motion_prmt === ’d50 && dut.detail_target === ’d0 && detail_prmt === ’d150) else $error ("Parameters and targets initialization failed."); // Check if the targets and parameters are initialized properly.
        rst = 0; #10; // De-assert the reset signal.
        assert (dut.state === s_make_motion && make_motion === 1’b1) else $error ("make_motion failed."); // As the clock evolves we should go to subsequent states according to the state transition diagram.
        #10;
        assert (dut.state === s_check_motion && make_motion === 1’b0 && check_motion === 1’b1) else $error ("check_motion failed.");
        #10;
        assert (dut.state === s_wait_motion_sccore && check_motion === 1’b0) else $error ("wait_motion_score failed.");
        motion_score = ’d80; #10; // Provide motion_score to check whether the FSM does the comparision with the motion_target correctly or not.
        assert (dut.state === s_increase_motion && motion_prmt === ’d51 ) else $error ("increase_motion failed.");
        #30;
        motion_score = ’d101; #10;
        assert (dut.state === s_decrease_motion && motion_prmt === ’d50 ) else $error ("decrease_motion failed.");
        #30;
        motion_score = ’d100; #10;
        assert (dut.state === s_decode_detail && motion_prmt === ’d50) else $error ("decode_detail failed.");
        aim = 3’b100; #10; // Provide aim to check if the FSM assign the correct detail_target.
        assert (dut.state === s_make_detail && dut.detail_target === ’d60 && make_detail === 1’b1) else $error ("make_detail_high failed.");
        #10;
        assert (dut.state === s_check_detail && make_detail === 1’b0 && check_detail === 1’b1) else $error ("check_detail failed.");
        #10;
        assert (dut.state === s_wait_detail_score && check_detail === 1’b0) else $error ("wait_detail_score failed.");
        detail_score = ’d59; #10; // Provide detail_score to check whether the FSM does the comparision with the detail_target
```

```
correctly or not.
        assert (dut.state === s_increase_detail && detail_prmt === 'd151) else $error ("increase_detail failed.");
        #30;
        detail_score = 'd61; #10;
        assert (dut.state === s_decrease_detail && detail_prmt === 'd150) else $error ("decrease_detail failed.");
        #30;
        detail_score = 'd60; #10;
        assert (dut.state === s_make_video && make_video === 1'b1) else $error ("make_video failed.");
        #10;
        assert (dut.state === s_done && make_video === 1'b0) else $error ("done failed.");
        rst = 1; #10; // Check the reset of the FSM when it is done.
        assert (dut.state === s_reset) else $error ("reset failed.");
        $stop;
    end

endmodule
```