Circuit Reliability and Testing

Boundary Scan Testing - JTAG

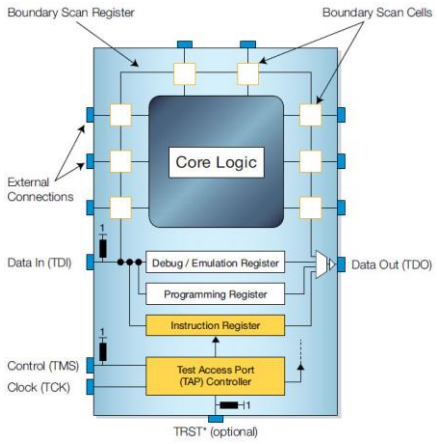
Name: Patsaoglou Panteleimon

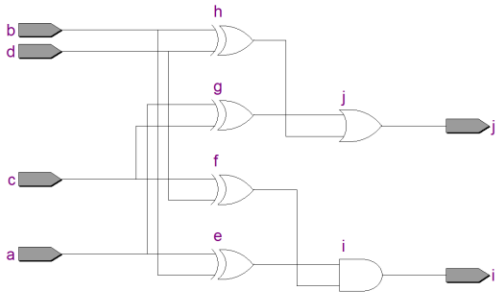
AM: 5102

- ### Assignment

In this assignment, a boundary scan testing methodology is implemented using the JTAG IEEE 1149.1 standard. JTAG provides a serial interface that enables intrusive and non-intrusive access to internal logic through a standardized protocol for probing and testing. It offers a systematic approach to testing not only individual integrated circuits, but also multiple blocks, within a SoC or on a PCB, connected in a chain configuration, eliminating the need for individual block testing.

The basic blocks such as the Tap Controller, Bypass/Instruction register and the Boundary Scan Cells needed to be implemented and their functionality validated through testbenches. The blocks were used to create the 2-bit Instruction register supporting the basic JTAG instructions (BYPASS, SAMPLE/PRELOAD, EXTEST). Together with the Tap Controller and Decoder Logic, they were integrated into the basic Circuit Under Test from the previous assignments by constructing the Boundary Scan Chain using the BSC cells.





- ### JTAG IEEE 1149.1 Test Flow

The JTAG IEEE 1149.1 test flow begins with the tap controller, which manages the state transitions based on the TMS and TCK signaling. Since the amount of testing pins needs to be minimized, the Tap Controller uses an FSM for the state transition logic. The state can be set by inputting the appropriate sequence of the TMS signal pin. Depending on the state being related to IR or DR, the internal mux will link the TDI and TDO pins to the instruction or data logic and prepare the internal blocks.

More precisely, IR states will link the TDI and TDO pins to the instruction logic so that JTAG instructions can be shifted into the instruction register to configure the testing method. Meanwhile, DR states link the pins to test-related blocks—such as scan chains, BIST blocks, etc. — for Datapath testing. This enables test vectors

to be shifted in (intrusive testing) and the responses from the circuits under test to be shifted out, allowing both intrusive and non-intrusive testing.

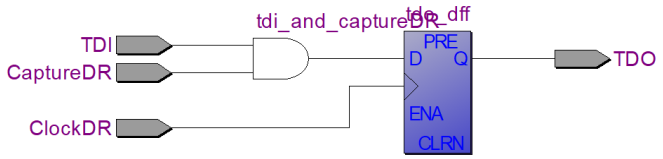
- **Bypass Register Cell Implementation** (3.1)

The Bypass Register is built in Verilog following the typical BR block Diagram.

- **Bypass Register Cell Verilog** (3.1)

```
module BR(ClockDR,TDI,CaptureDR,TDO);
// Defining necessary ports
input ClockDR,TDI,CaptureDR;
output TDO;
// Defining wiring for internal connections
wire tdi_and_captureDR;
// Defining internal BR reg
reg tdo_dff;
// assign AND gate
assign tdi_and_captureDR = TDI&CaptureDR;
// assign TDO
assign TDO = tdo_dff;
// always block for TDO DFF
 always @ (posedge ClockDR)
  begin
    tdo_dff<=tdi_and_captureDR;
end
```

- **Bypass Register Cell Synthesized** (3.1)



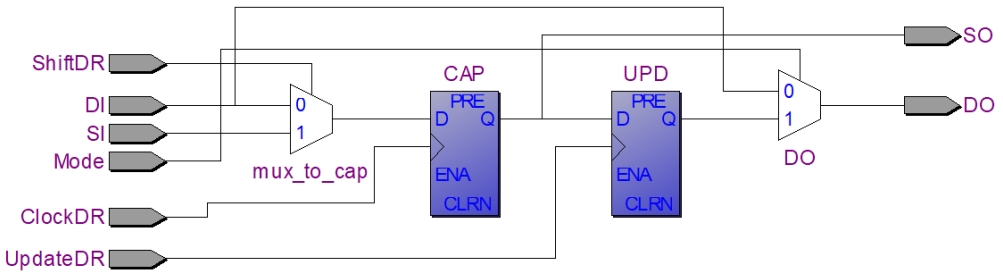
- **Boundary Scan Cell Implementation** (3.1)

The Boundary scan cell is built in Verilog following the typical BSC block Diagram.

- **Bypass Register Cell Verilog** (3.1)

```
module BSC(DI,SI,ShiftDR,ClockDR,UpdateDR,Mode,SO,DO);
// Defining necessary ports
input DI,SI,ShiftDR,ClockDR,UpdateDR,Mode;
output SO,DO;
// Defining internal BSC regs
reg CAP,UPD;
// Defining wiring for internal connections
wire mux_to_cap,cap_to_upd,upd_to_mux;
// assign mux in
assign mux_to_cap=(ShiftDR)?SI:DI;
// assign cap output
assign cap_to_upd=CAP;
// assign mux out
assign DO=(Mode)?upd_to_mux:DI;
// assign upd to mux
assign upd_to_mux=UPD;
// assign Shift out
assign SO=cap_to_upd;
// always block for CAP DFF
 always @ (posedge ClockDR)
 begin
  CAP<=mux_to_cap;
 end
// always block for UPD DFF
 always @ (posedge UpdateDR)
 begin
  UPD<=cap_to_upd;
 end
endmodule
```

- **Bypass Register Cell Synthesized** (3.1)



- **Bypass Register Cell Testbench Verilog** (3.1)

```
module BSC_tb();
// reg declarations to drive BSC inputs
reg DI,SI,ShiftDR,ClockDR,UpdateDR,Mode;
reg [32*8:0] testing_state;
// wires to get SO and DO of BSC output
wire SO,DO;
// wiring up BSC
BSC BSC(
        .DI(DI),
        .SI(SI),
        .ShiftDR(ShiftDR),
        .ClockDR(ClockDR),
        .UpdateDR(UpdateDR),
        .Mode(Mode),
        .SO(SO),
        .DO(DO)
);
// simulating clock
initial begin
        ClockDR = 0;
        forever begin
                #5 ClockDR =! ClockDR;
        end
end
// block to generate random SI and DI bit values
initial begin
        DI=0;
        SI=0;
        forever begin
                DI=$random;
                SI=$random;
                #10;
        end
end
initial begin
        // setting some initial values-Normal Mode(d)
        Mode=0;
        UpdateDR=0;
        // save data to CAP from DI for 2 cycles(a)
        testing_state="DI to CAP";
        ShiftDR=0;
        #20;
        // save data to CAP from SI for 2 cycles(b)
        testing_state="SI to CAP";
        ShiftDR=1;
        #20;
        // transfer data from CAP to UPD-perform update(c)
        testing_state="CAP to UPD";
        UpdateDR=1;
        #2;
        UpdateDR=0;
        #8;
        // push data from UPD to internal logic-Test mode(e)
        testing_state="UPD to DO";
        Mode=1;
        #10;
        //switch back to normal mode-from DI to DO and in internal logic
        testing_state="DI to DO";
        Mode=0;
        #10;
end
endmodule
```

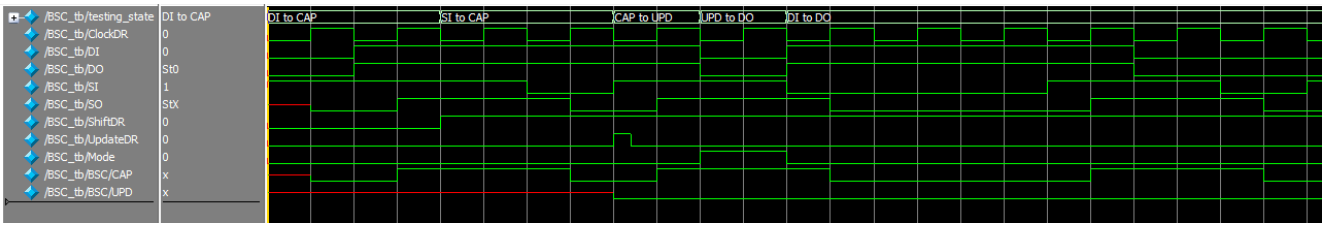
- **Bypass Register Cell Testbench Description** (3.1)

The testbench for the Boundary Scan Cell module is simulated by driving all the necessary signal pins and observing the response on the ModelSim waveform. The clock signal is simulated by toggling the value of the register with a period of 10 time units, while DI and SI inputs are driven with random values to simulate dynamic data inputs.

The testbench tests various functionality cases like storing data for the DI input to the capture FF, then toggles the ShiftDR input so the SI value is stored in the cap FF. After that the update FF functionality is evaluated, by setting the UpdateDR pin to 1 and observing the value of the cap FF getting stored in the UPD FF. Next, the Mode is toggled to 1 so the internal mux links DO with the UPD FF and lastly the mode is set back to 0 and the DI input is fed directly at DO simulating the non-invasive mode of the JTAG chain registers.

Each test case is labeled with the testing_state variable for test state tracking and debugging purposes as seen in the waveform below.

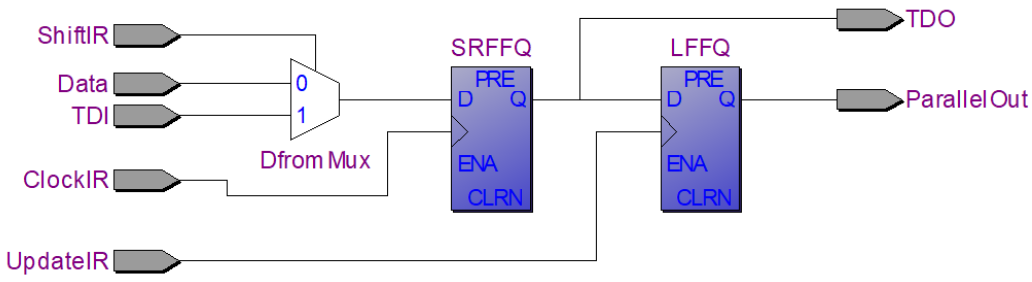
- **Bypass Register Cell Waveform** (3.1)



- **Instruction Register Cell Implementation** (3.1)

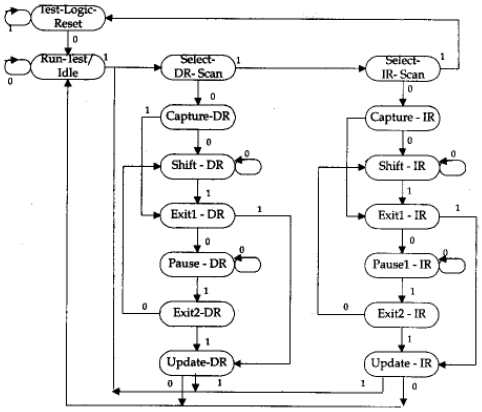
The IR cell module is already implemented and supplied in the assignment material.

- **Instruction Register Synthesized** (3.1)



- **Tap Controller FSM Implementation** (3.2)

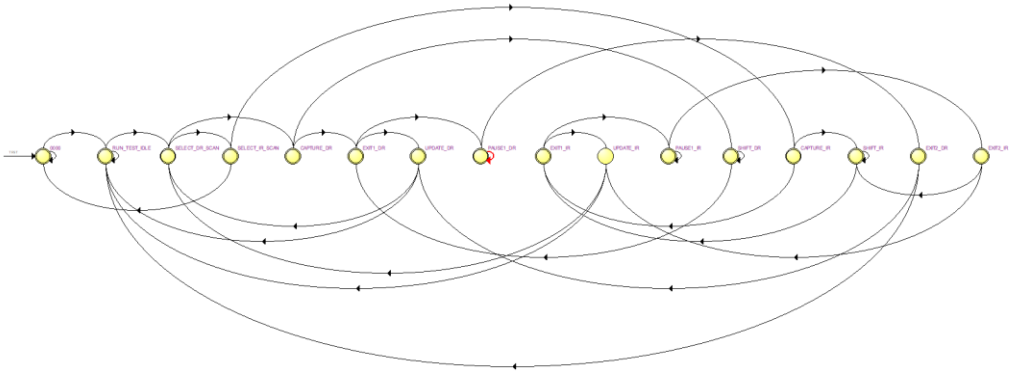
The tap controller FSM is implemented in Verilog following the standard JTAG FSM seen in the diagram below:



As seen from the diagram above, the TMS input value on the clock edge defines the state transitions. There are 2 JTAG idle states and from the RUN-Test/Idle state and TMS=1 the FSM switches to test mode. The are 2 well defined transition columns, one that defines state transitions related to the instruction register and one that defines transitions for the datapath-related registers.

All the transitions depending on the TMS are summarized in the table below, generated by Quartus using the TAP Controller code:

	Source State	Destination State	Condition
1	0000	RUN_TEST_IDLE	(!TMS)
2	0000	0000	(TMS)
3	CAPTURE_DR	SHIFT_DR	(!TMS)
4	CAPTURE_DR	EXIT1_DR	(TMS)
5	CAPTURE_IR	EXIT1_IR	(TMS)
6	CAPTURE_IR	SHIFT_IR	(!TMS)
7	EXIT1_DR	PAUSE1_DR	(!TMS)
8	EXIT1_DR	UPDATE_DR	(TMS)
9	EXIT1_IR	UPDATE_IR	(TMS)
10	EXIT1_IR	PAUSE1_IR	(!TMS)
11	EXIT2_DR	UPDATE_DR	(TMS)
12	EXIT2_DR	RUN_TEST_IDLE	(!TMS)
13	EXIT2_IR	UPDATE_IR	(TMS)
14	EXIT2_IR	SHIFT_IR	(!TMS)
15	PAUSE1_DR	PAUSE1_DR	(!TMS)
16	PAUSE1_DR	EXIT2_DR	(TMS)
17	PAUSE1_IR	PAUSE1_IR	(!TMS)
18	PAUSE1_IR	EXIT2_IR	(TMS)
19	RUN_TEST_IDLE	SELECT_DR_SCAN	(TMS)
20	RUN_TEST_IDLE	RUN_TEST_IDLE	(!TMS)
21	SELECT_DR_SCAN	SELECT_IR_SCAN	(TMS)
22	SELECT_DR_SCAN	CAPTURE_DR	(!TMS)
23	SELECT_IR_SCAN	CAPTURE_IR	(!TMS)
24	SELECT_IR_SCAN	0000	(TMS)
25	SHIFT_DR	SHIFT_DR	(!TMS)
26	SHIFT_DR	EXIT1_DR	(TMS)
27	SHIFT_IR	EXIT1_IR	(TMS)
28	SHIFT_IR	SHIFT_IR	(!TMS)
29	UPDATE_DR	SELECT_DR_SCAN	(TMS)
30	UPDATE_DR	RUN_TEST_IDLE	(!TMS)
31	UPDATE_IR	SELECT_DR_SCAN	(TMS)
32	UPDATE_IR	RUN_TEST_IDLE	(!TMS)



- **Tap Controller FSM Verilog** (3.2)

```
module TAP_CONTROLLER_FSM(TCK,TMS,TRST,STATE);
// Defining necessary ports
input TCK,TMS,TRST;
output[3:0] STATE;
// Defining FSM memory
reg[3:0] tap_state;
// Assigning STATE output to FSM Memory
assign STATE=tap_state;
// Defining known FSM stages
localparam TEST_LOGIC_RESET                =4'd0;
localparam RUN_TEST_IDLE                   =4'd1;
localparam SELECT_DR_SCAN                  =4'd2;
localparam CAPTURE_DR                      =4'd3;
localparam SHIFT_DR                        =4'd4;
localparam EXIT1_DR                        =4'd5;
localparam PAUSE1_DR                       =4'd6;
localparam EXIT2_DR                        =4'd7;
localparam UPDATE_DR                       =4'd8;
localparam SELECT_IR_SCAN                  =4'd9;
localparam CAPTURE_IR                      =4'd10;
localparam SHIFT_IR                        =4'd11;
localparam EXIT1_IR                        =4'd12;
localparam PAUSE1_IR                       =4'd13;
localparam EXIT2_IR                        =4'd14;
localparam UPDATE_IR                       =4'd15;
// always block for TCK and TRST
always @ (posedge TCK or posedge TRST)
 begin
 // if asyn TRST rst tap_state to 0
 if (TRST)
   tap_state<=0;
 else begin
 // switch case for FSM implementation (taken from TAP_CONTROLLER
diagram)
 case(tap_state)
                    TEST_LOGIC_RESET:
            tap_state<=(TMS)?TEST_LOGIC_RESET:RUN_TEST_IDLE;
                    RUN_TEST_IDLE:
            tap_state<=(TMS)?SELECT_DR_SCAN:RUN_TEST_IDLE;
                    SELECT_DR_SCAN:
            tap_state<=(TMS)?SELECT_IR_SCAN:CAPTURE_DR;
                    CAPTURE_DR:
            tap_state<=(TMS)?EXIT1_DR:SHIFT_DR;
                    SHIFT_DR:
            tap_state<=(TMS)?EXIT1_DR:SHIFT_DR;
                    EXIT1_DR:
            tap_state<=(TMS)?UPDATE_DR:PAUSE1_DR;
                    PAUSE1_DR:
            tap_state<=(TMS)?EXIT2_DR:PAUSE1_DR;
                    EXIT2_DR:
            tap_state<=(TMS)?UPDATE_DR:RUN_TEST_IDLE;
                    UPDATE_DR:
            tap_state<=(TMS)?SELECT_DR_SCAN:RUN_TEST_IDLE;
                    SELECT_IR_SCAN:
            tap_state<=(TMS)?TEST_LOGIC_RESET:CAPTURE_IR;
                    CAPTURE_IR:
            tap_state<=(TMS)?EXIT1_IR:SHIFT_IR;
                    SHIFT_IR:
            tap_state<=(TMS)?EXIT1_IR:SHIFT_IR;
                    EXIT1_IR:
            tap_state<=(TMS)?UPDATE_IR:PAUSE1_IR;
                    PAUSE1_IR:
            tap_state<=(TMS)?EXIT2_IR:PAUSE1_IR;
                    EXIT2_IR:
            tap_state<=(TMS)?UPDATE_IR:SHIFT_IR;
                    UPDATE_IR:
            tap_state<=(TMS)?SELECT_DR_SCAN:RUN_TEST_IDLE;
                    default:
            tap_state<=TEST_LOGIC_RESET;
```

- **Tap Controller FSM Testbench Verilog** (3.2)

```
module TAP_CONTROLLER_tb();
// Defining regs to drive TAP inputs
reg TCK,TMS,TRST;
// Defining wire to get state from module
wire[3:0] fsm_state;
// reg to save string state
reg[31*8:0] state_str;
// Wire up TAP_CONTROLLER Module
TAP_CONTROLLER_FSM TAP_CONTROLLER_FSM(
		.TMS(TMS),
		.TCK(TCK),
		.TRST(TRST),
		.STATE(fsm_state)
);
// Simulate clock
initial begin
		TCK = 0;
		forever begin
			#5 TCK =! TCK;
		end
end
initial begin
		//		Hold TRST for 2 cycles
		TRST=1;
		TMS=0;
		#10
		// simulate a FSM path:
		// TEST_LOGIC_RESET
		// RUN_TEST_IDLE
		// SELECT_DR_SCAN
		// CAPTURE_DR
		// EXIT1_DR
		// UPDATE_DR
		// SELECT_DR_SCAN
		// SELECT_IR_SCAN
		// CAPTURE_IR
		// SHIFT_IR
		// EXIT1_IR
		// PAUSE1_IR
		// EXIT2_IR
		// UPDATE_IR
		// RUN_TEST_IDLE
		// TEST_LOGIC_RESET
		#10;
		TRST=0;
		TMS=0; //RUN_TEST_IDLE
		#10;
		TMS=1; //SELECT_DR_SCAN
		// ... TMS Sequence for state path above
		TMS=0; //RUN_TEST_IDLE
		#10;
		#10;
		TRST=1;
end
// Always block to toggle state_str on each cycle
// for debugging purposes
 always @ (posedge TCK)
 begin
case (fsm_state)
    4'd0:  state_str = "TEST_LOGIC_RESET";
   // ... String states for debugging purposes
    4'd15: state_str = "UPDATE_IR";
    default: state_str = "UNKNOWN";
  endcase
 end
endmodule
```

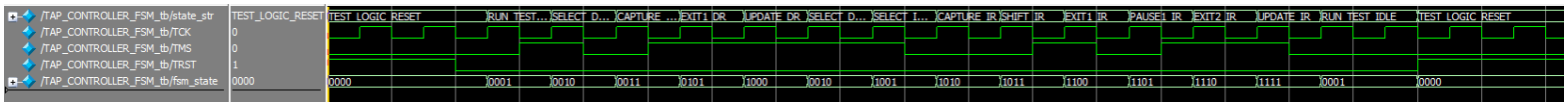
- **Tap Controller FSM Testbench Description** (3.2)

The testbench for the FSM Tap Controller verifies the correct sequencing of the JTAG states. A TCK is driven by toggling the value of the reg with period of 10 time units while for the first 2 clock cycles the TRST is set to 1 to initialize the FSM state to Test Logic Reset.

Once the reset is done, a TMS sequence is generated using the FSM diagram and it is fed in the Tap controller on each clock cycle. The next state of the FSM should strictly follow the transitions of the JTAG standard FSM.

As seen in the waveform below, the correct sequence of states is got for the given TMS transitions (TMS transition values and states are referred to comments at the Verilog code).

- **Tap Controller FSM Testbench Waveform** (3.2)



- **JTAG Blocks – Integration with the CUT** (4)

Once all the necessary blocks are built, it is necessary to be integrated into the CUT and additional logic needs to be added so all the blocks work in order.

One of the first modifications made, is that TRST pins are added to combinational logic like the BR, BSC and IR cells so the blocks are initialized at a known state.

Also, a 2-bit IR needs to be built from the IR cell, so it covers the basic 3 instructions of the JTAG controller, that of BYPASS, EXTEST and SAMPLE/PRELOAD.

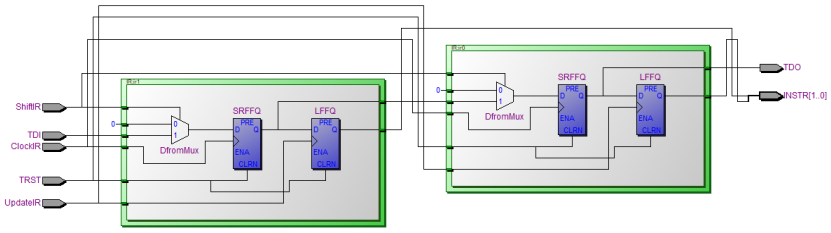
CUT also needs modification. BSC blocks need to be added to the inputs and the outputs of the CUT and to be connected in a chain configuration, so test vectors can be shifted in the chain through the TDI input on DR related states (SAMPLE/PRELOAD, EXTEST) and the response of the cut to be shifted out, through the TDO output.

Lastly, additional logic needs to be added to the tap controller, that generates control signals based on the state of the FSM so CLK/ENABLE/CAPTURE/UPDATE signals and CLKDR/IR are fed correctly in the necessary Blocks.

- **2-bit Instruction Register Verilog**

```
module IR_2b(TRST,TDI,ShiftIR,ClockIR,UpdateIR,TDO,INSTR);
        // declaration of necessary ports for 2bit IR
        input TRST,TDI,ShiftIR,ClockIR,UpdateIR;
        output TDO;
        output[1:0] INSTR;
        // wire to connect ir0 with ir1
        wire tdo1_tdi0;
        wire Data;
        // data BUS is not needed
        assign Data=0;
        // wiring up ir modules
        IR ir1(TRST,Data, TDI, ShiftIR, ClockIR, UpdateIR, INSTR[1], tdo1_tdi0);
        IR ir0(TRST,Data, tdo1_tdi0, ShiftIR, ClockIR, UpdateIR, INSTR[0], TDO);
endmodule
```

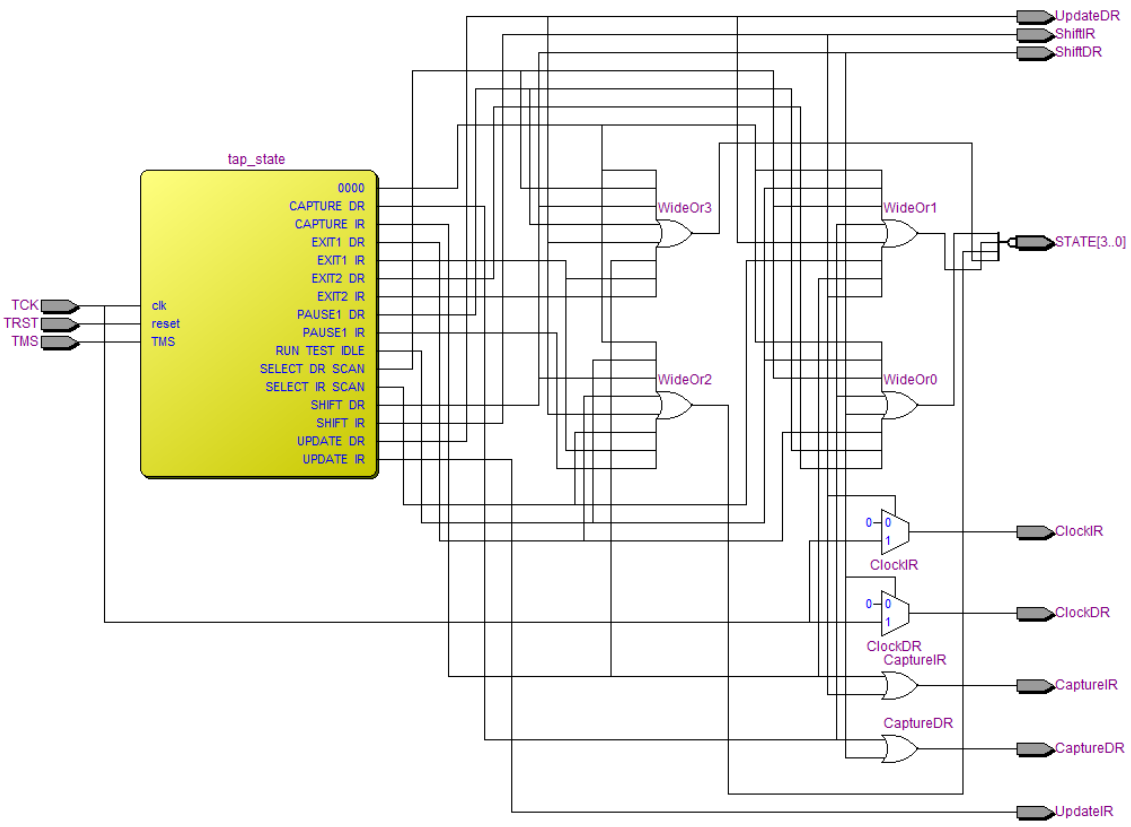
- **2-bit Instruction Register Synthesized**



- **FSM Controller – Addition of Control signals**

```
// addition of control signal ports
output ShiftIR,CaptureIR,UpdateIR,ClockIR;
output ShiftDR,CaptureDR,UpdateDR,ClockDR;
...
// Assigning STATE output to FSM Memory
assign STATE=tap_state;
// Assigning FSM control signals (delay added for simulation timing synchr)
assign #1 ShiftIR=(tap_state==SHIFT_IR);
assign #1 CaptureIR=(tap_state==CAPTURE_IR || tap_state==SHIFT_IR);
assign #1 UpdateIR=(tap_state==UPDATE_IR);
assign #1 ShiftDR=(tap_state==SHIFT_DR);
assign #1 CaptureDR=(tap_state==CAPTURE_DR ||
tap_state==SHIFT_DR)?1'b1:1'b0;
assign #1 UpdateDR=(tap_state==UPDATE_DR);
// enable clock for BSC and IR cell shifts
assign ClockDR=(tap_state==SHIFT_DR)?TCK:1'b0;
assign ClockIR=(tap_state==SHIFT_IR)?TCK:1'b0;
```

- **FSM Controller with control signals - Synthesized**



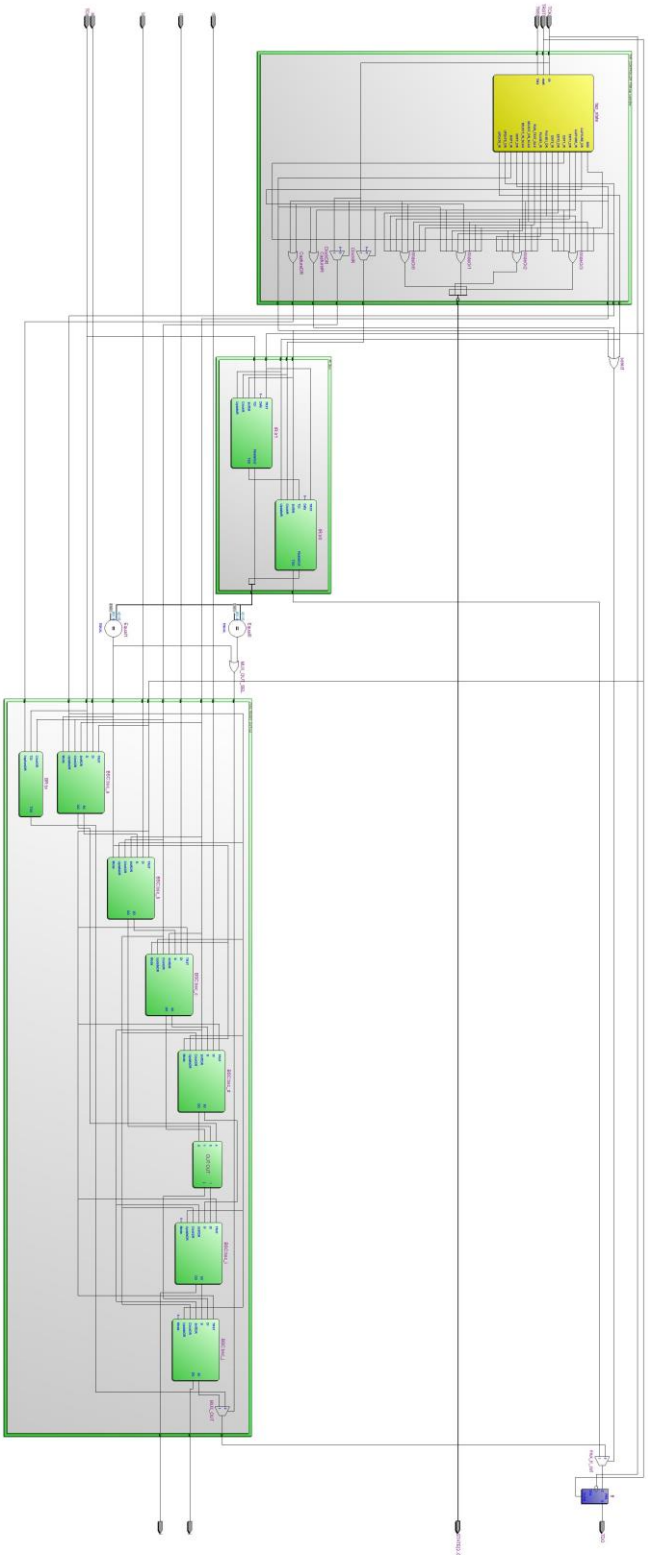
- **Cut With BSC chain Verilog – JTAG_READY_CUT**

```
module
JTAG_READY_CUT(TRST,TDI,ShiftDR,ClockDR,UpdateDR,CaptureDR,Mode,MUX_OUT_SEL,a,b,c,d,i,j,
MUX_OUT);
        // direct interface IO for the CUT
        input a,b,c,d;
        output i,j;
        // JTAG related IO
        input TRST,TDI,ShiftDR,ClockDR,UpdateDR,CaptureDR,Mode,MUX_OUT_SEL;
        output MUX_OUT;
        // wire for JTAG related blocks
        wire br_tdo,SO;
        // wires to wire up BSC to BSC
        wire bsca_bscb,bscb_bscc,bscc_bscd,bscd_bsci,bsci_bscj;
        // wires to wire up BSC to cut
        wire bsc_cut_a,bsc_cut_b,bsc_cut_c,bsc_cut_d;
        // wires to wire up cut to BSC
        wire cut_bsc_i,cut_bsc_j;
        // BSC for CUT inputs
        BSC bsc_a(TRST,a,TDI,ShiftDR,ClockDR,UpdateDR,Mode,bsca_bscb,bsc_cut_a);
        BSC bsc_b(TRST,b,bsca_bscb,ShiftDR,ClockDR,UpdateDR,Mode,bscb_bscc,bsc_cut_b);
        BSC bsc_c(TRST,c,bscb_bscc,ShiftDR,ClockDR,UpdateDR,Mode,bscc_bscd,bsc_cut_c);
        BSC bsc_d(TRST,d,bscc_bscd,ShiftDR,ClockDR,UpdateDR,Mode,bscd_bsci,bsc_cut_d);
        // BSC for CUT outputs - mode set to 0 just to get cut output response
        BSC bsc_i(TRST,cut_bsc_i,bscd_bsci,ShiftDR,ClockDR,UpdateDR,1'b0,bsci_bscj,i);
        BSC bsc_j(TRST,cut_bsc_j,bsci_bscj,ShiftDR,ClockDR,UpdateDR,1'b0,SO,j);
        // BR wire up
        BR br(ClockDR,TDI,CaptureDR,br_tdo);
        // CUT wire up
        CUT CUT(bsc_cut_a,bsc_cut_b,bsc_cut_c,bsc_cut_d,cut_bsc_i,cut_bsc_j);
        // assigning mux output 1: bsc_j_SO 0:BYPASS
        assign MUX_OUT=(MUX_OUT_SEL)?SO:br_tdo;
endmodule
```

- **JTAG Ready IC – Verilog**

```
module JTAG_READY_IC(TDI,TCK,TMS,TRST,a,b,c,d,i,j,TDO,STATE);
        // ports for the JTAG ready ic
        input TDI,TCK,TMS,TRST,a,b,c,d;
        output i,j,TDO;
        output[3:0] STATE;
        // wires to wire up tap controller signals with JTAG_READY_CUT
        wire ShiftIR,CaptureIR,UpdateIR;
        wire ShiftDR,CaptureDR,UpdateDR;
        wire ClockIR,ClockDR;
        // wire bus to wire instruction port of the TAP_CONTROLLER_FSM
        wire[1:0] Instruction;
        // instruction decode logic based on instruction bus
        // to set mode and MUX_OUT_SEL
        localparam [1:0]
        BYPASS  = 2'b00,
        SAMPLE  = 2'b01,
        EXTEST  = 2'b10;
        wire MUX_OUT_SEL;
        wire Mode;
        wire instr_bypass  = (Instruction == BYPASS);
        wire instr_sample  = (Instruction == SAMPLE);
        wire instr_exttest = (Instruction == EXTEST);
        assign MUX_OUT_SEL = instr_sample | instr_exttest;
        assign Mode       = instr_exttest;
        // output mux to select between cut output or IR output
        wire ir_tdo;
        wire cut_mux_out;
        wire select;
        assign select=(ShiftIR==1 || CaptureIR==1 || UpdateIR==1);
        wire mux_ir_cut;
        assign mux_ir_cut=(select)?ir_tdo:cut_mux_out;
        // negative edge output FF declaration and wiring
        reg ff;
        assign TDO = ff;
        always @(negedge TCK or posedge TRST) begin
                if(TRST)
                        ff<=0;
                else begin
                        ff<=mux_ir_cut;
                end
        end
        // wiring up the tap controller IO
        TAP_CONTROLLER_FSM tap_controller(.TCK(TCK),.TMS(TMS),.TRST(TRST),.STATE(STATE),.ShiftIR(ShiftIR),
        .CaptureIR(CaptureIR),.UpdateIR(UpdateIR),.ShiftDR(ShiftDR), .CaptureDR(CaptureDR),.UpdateDR(UpdateDR),
        .ClockDR(ClockDR), .ClockIR(ClockIR));
        // wiring up the cut
        JTAG_READY_CUT cut(.TRST(TRST),.TDI(TDI),.ShiftDR(ShiftDR),.ClockDR(ClockDR),.UpdateDR(UpdateDR),
        .CaptureDR(CaptureDR),.Mode(Mode),.MUX_OUT_SEL(MUX_OUT_SEL),.a(a),.b(b),.c(c),.d(d),.i(i),.j(j),
        .MUX_OUT(cut_mux_out));
        // LSB-first
        // wiring up 2 bit instruction register
        IR_2b ir(.TRST(TRST), .TDI(TDI),.ShiftIR(ShiftIR),.ClockIR(ClockIR),.UpdateIR(UpdateIR),.TDO(ir_tdo),.INSTR(Instruction));
endmodule
```

- **JTAG Ready IC – Synthesized**



- **JTAG Ready IC Bypass Testbench Description**

The JTAG_READY_IC_tb verifies the JTAG interface when the instruction register is configured in **bypass mode**. For sizing purposes, the code is not included in this section, but it can be found in the code material.

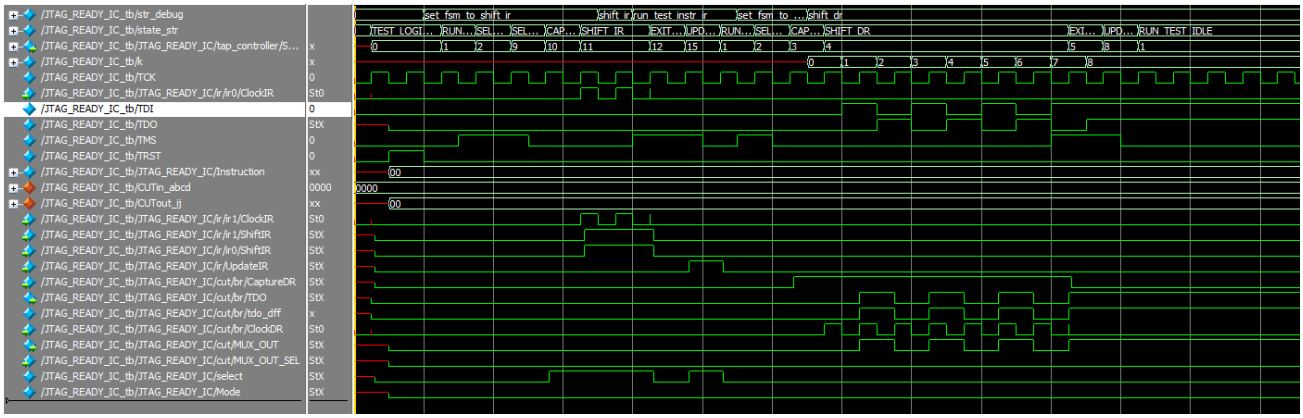
For code reusability ease, the most basic operations like set_fsm_to_shift_ir, set_fsm_to_shift_dr, shift_ir, shift_dr are defined as tasks so they can be used in other test cases, since on the respective task there is going to be the same TMS sequence.

The testbench begins by driving TRST to 1 so the internal logic initializes in a known state and the FSM initializes in TEST_LOGIC_RESET. In the next clock cycles, the proper TMS sequence is input so the tap controller enters in the SHIFT_IR state and the Instruction code can be shifted into the 2b_IR. In the next cycles the ClockIR is enabled so the BYPASS(0x00) instruction is shifted to the instruction register from TDI and once the shifting phase completes, the tap controller enters the TEST_IDLE state.

In the next clock cycles, TMS sequence is entered so the Tap controller enters the SHIFT_DR state on the BYPASS instruction. The vector 0b10101010 is shifted through TDI and ClockDR is enabled so CAP FFs get clock signaling. From the waveform below, it is observed that on the negative clock edge, we get the vector through the BYPASS register on TDO.

Once the shifting completes, the Tap controller, through the proper TMS sequence, enters back to the RUN_TEST_IDLE state.

- **JTAG Ready IC Bypass Testbench Waveform**



- **JTAG Ready IC SAMPLE/PRELOAD-EXTEST Testbench Description**

This testbench follows the same code logic as the previous one for the BYPASS test, but this time it is testing the SAMPLE non-invasive instruction to probe to the IO of the CUT. Also, while the response of the probing is shifted out, through TDI an external test vector is shifted into the chain, so external test can be performed. To initialize the test vector, update must be performed so the values of the CAP FFs are saved in the UPD FFs and the instruction must also be set to EXTEST (0b10). This through the BSCs output mux, will switch the output of the BSC cells to the UPD FFs, so the cut gets the input vector from the shifted external vector and not the standard CUT IO. It is clear that this is an invasive test configuration since the response of the CUT does not come from its IO logic.

On the waveform below, we can observe that TRST is initially set to initialize the combinational logic and the tap controller. Through the correct TMS sequence, enters to the SHIFT_IR state so the SAMPLE instruction can be shifted in through TDI. One the instruction is ready, the tap controller enters in the CAPTURE_DR state and we see the sampled CUT IO response saved in the CAP FFs of the BSC scan chain cells.

In the next clock cycles, state is kept to SHIFT_DR so the capture is shifted out through TDI. While the response is shifted out, the next test vector is fed into TDI and saved into the CAP FFs, so external testing can be performed in the next cycles.

Once the shift from the sample response is complete, a new instruction (EXTEST) must be fed into the instruction register so logic and modes switch to the external test configuration. The typical shift instruction procedures are followed once again, and the tap controller with the correct TMS sequence, enters the CAPTURE/SHIFT_DR state, this time with the instruction being EXTEST. In the next 2 cycles we get the expected CUT response based on the test vector we shifted in.

- **JTAG Ready IC SAMPLE/PRELOAD-EXTEST Testbench Waveform**