Testing and Verification of Verilog-Based Digital Circuit Design

CS 6962 - Software Verification

Anh Luong & Andrzej Forys

1 Abstraction

The difficulty of finding correctness of digital circuit design is dependant on the complexity. With the conventional workflow, circuits are prone to have problems. These problems could be costly in time and material cost to rework. Throughout this paper, we will discuss a newly proposed workflow on how to ensure correctness of digital circuits using new Model Checker and Automatic Test Patterns Generation (ATPG) tools. The paper also briefly discusses related topics to software verification and the current state of the art. This paper will also cover the current hardware design flow along with the new tools we researched. We created a finite state machine, CR-16 controller module, to demonstrate the approach of this flow. In addition, various Scan Flip-Flops were designed to test hardware ATPG. Results will be shown for reference.

2 Introduction

Digital integrated circuit (IC) designs are required to be accurate due to the cost of production. Although testing and verification of IC design can be done in parallel, it requires time and resources which most designing groups do not have. Complications come with intricate designs.

IC design starts by using the basic models of transistors. Schematics for boolean gates are later created, and unit tests are applied to ensure correct functionality. Depending on complexity, it may not be feasible to test the circuit all the way. The layouts for these gates are created using abstracted physical material models. Another tool is used to match the physical characteristics of the layout to the schematic functionality. Analog characteristics are checked to ensure the proper electrical response from the circuits including timing, power consumption, and size. Once each gate‘s correctness is ensured, these layouts and schematics combine into one library file for later use.

There are several ways to create IC designs with modern tools. The option covered in this paper is Synthesis, an automated system. Because of timing constraints, complex designs are usually created autonomously due to the various parameters software can test for and optimize. To begin the automated process; behavioral Verilog is used with the library file to create structural Verilog. This piece of Verilog is a gate level description of the circuit with a wiring netlist. The place and route tool, Encounter, is used to generate the final layout and schematic. Once completed, a digital and analog test runs again to ensure correctness of the circuit. From there the design is generally sent for fabrication.

Designers usually create Unit Tests for each component of the Verilog description. It is quite time consuming to set up and run the test for all possible inputs and outputs. Circuits are only simulated with known input values and results. Verilog CounterExample Abstraction Refinement (VCEGAR) is the new tool that we suggest for this part of the process. Symbolic Model Checker (SMV) enables the circuit to be tested with all possible inputs.

Although testing is done at each level of the IC design, other possible problems with fabrication can cause the IC to operate incorrectly. Post-silicon chip testing is crucial to ensure the expected operational functionality. During this process the tester only has access to the available external pins. There is no way to monitor the inside of the IC. More on this topic will be discussed in a later section. For this, ATPG tools should be used. Particularly, Synopsys suite contains the DFT Compiler and TetraMax to assist the designer in this task.

3 OVL (Open Verification Library)

3.1 Verilog

The two main Hardware Description Languages are Verilog and VHDL. Verilog is the IEEE standard that we will be using throughout this project. It is not only used for designs but also verification of digital circuits, analog circuits, and mixed signal circuits. Even though Verilog is a good unit testing language, creating testbenches is still time consuming.

3.2 Open Verification Library

Acellera created a standardized verification library called Open Verification Library. It supports programming in Verilog, System Verilog, VHDL, PSL, and SystemC to verify behaviors during simulation, emulation and formal verification. This helps engineers in the design, integration, and verification of digital circuits [1].

3.3 Property Specification Language

The same company, Acellera, created Property Specification Language (PSL) [2]. It gave another method to solve this software verification problem. PSL allows the creation of a Boolean representation of the unit under test, called property. Unlike other assertion based checkers that require the property within the same file, PSL allows the property to be in a secondary file. Assertions, created from the property, are instruction sets for the verification tool to verify the program characteristics. PSL can be used dynamically with a simulator or statically with a model checker during formal analysis.

3.4 System Verilog Assertion

Another standard IEEE adopted as an extension to Verilog is System Verilog Assertion (SVA) [3, 4]. It is a combination of Hardware Description Language and Hardware Verification Language. SVA is based on OpenVera which was developed by Synopsys as an open-source hardware verification language. It is a language used for creating testbenches.

4 VCEGAR

Verilog programs are verified using assertions with VCEGAR. The tool takes a Verilog description and a single property that describes the entire circuit. The property can either be validated, or a valid counterexample can be created to show how the property is violated. VCEGAR uses word level predicate abstraction and refinement for checking properties in register-transfer level (RTL) Verilog programs.

4.1 Symbolic Model Checker

VCEGAR does all of the above with the help of the Symbolic Model Checker (SMV). There are several available options for these checkers, such as CMU SMV, Cadence SMV, and NuSMV. Originally, Carnegie Mellon University created SMV during their model checking research [5]. Their goal was to create a method to formally verify finite-state concurrent systems. VCEGAR is built on top of SMV. It simplifies testing by eliminating the need for creating input vectors or a test bench. It simulates patterns for all possible inputs. It differs from formal verification because of its generalization of model checking. Model checking creates an abstract system level model and uses original specifications from the design process to verify the model.

4.2 Cadence SMV

Another extension of CMU SMV is Cadence SMV [6]. It has more verbose output than the standard SMV which allows the designer to easily find violated sections. The modeling language can be synthesizable Verilog. RTL designs are also verifiable. Cadence SMV is an industrial verification tool. It allows large designs by using techniques for compositional verification. Several different specifications are compatible with this tool including Computational Tree Logic (CTL), Linear Temporal Logic (LTL), finite automata, embedded assertions, and refinement specifications.

4.3 NuSMV

Another reimplementation of SMV is NuSMV [7]. It is one of the first model checkers based on Binary Decision Diagrams (BDDs). It uses an open-source architecture for model checking. It is more scalable to industrial design than the original CMU SMV. It can be found in many cores of custom verification tools including VCEGAR. It has become the main testbed for formal verification techniques. The latest version, NuSMV uses CU Decision Diagram (CUDD) library for its BDD-based model checker. Another addition includes a RBD-based bounded model checker which is connected to the MiniSAT Solver and ZChaff SAT Solver for its SAT-based Model Checker.

4.4 CEGAR

VCEGAR uses a model-checking algorithm, called CEGAR. This algorithm is a guided refinement of abstraction that either validates or solidifies a counterexample. CEGAR creates an initial abstraction model that is then passed to a model checker. The model checker returns either whether the model is correct or a counterexample is needed. A counterexample analysis is then triggered to create an example for the violation. Whether a counterexample could or could not be generated, an example depends on the abstraction model. An abstraction refinement step triggers if an example could not be found. After refinement, the new model passes back to a model checker to continue the process. For any given abstraction model, the process will repeat until the model proved or a counterexample found.

5 Hardware Design Process

This section will talk about the basic process of hardware design using Cadence and Synopsys CAD tools. This process is taught during the Digital VLSI classes at the University of Utah. The material will help the reader understand the tools and procedures used in the process of implementing test features in a digital design. The specifics to Scan Flip-Flops (FFs) are detailed in the Scan FF section. The FFs follow the design process up until place and route.

5.1 Cell Descriptions

The first step in the design process is to create a library of all the basic logic functions that make up a complete digital circuit. These include power sources that pull either high or low, boolean gates such as: NAND, XOR, and NOR, and higher level logic functions such as FFs. Each component will have its own ‘cell’ that contains ‘views’ which can correspond to a schematic, Verilog description, layout, physical properties, and more [8].

Schematic views use transistor models and wire interconnects to represent basic logic gates. Schematic symbols are used to represent the contents of an entire schematic, including input and output (I/O) logic. These can be used to build more complex structures while abstracting the transistor level beneath. The Scan FF schematics are available in Appendix A.

To ensure the IC design implementation is free of errors, unit testing is done along the way. Each schematic is tested using a Verilog test bench which can be written to produce console outputs for pre- and post-conditions along with a digital waveform showing how logic behaves for each I/O line. As schematics are completed, each has a Verilog description associated with it called the behavioral view. This can be independently tested from the schematic and compared to ensure a correct representation in the schematic and Verilog behavior of the circuit. A waveform using additional transistor delay information, rise and fall times, in the behavioral file is available in Appendix B.

5.2 Layout

The next step in the design process is to create a layout view which is the blueprint of the physical fabrication of the cell. The various materials used in fabrication are abstracted from the user as specific colors and visual patterns. Sizing and space constraints are generally based on the fabrication process involved. All cell specifications used in this project are based on the MOSIS 0.6um process [8].

An important feature of the layout tool is Design Rules Checking (DRC). This verifies that all placements are within tolerances given the specific rule set for fabrication. Designers become in the habit of frequently using this feature,

because errors can quickly compound until a layout has to be nearly completely redone to clear the issues. Once a layout is completed and DRC shows no errors, the designer uses another tool to extract the physical property information from the layout. This will be used in the Layout Versus Schematic Checking (LVS) tool to compare the behavior of the schematic with the behavior of the layout. This will ensure the library is matching cell layouts with their corresponding Verilog descriptions correctly. When this step completes, the tools can generate an analog extracted view which will be used during the main library file generation. A Scan FF layout is available in Appendix C [8].

Although the Synthesis tools are able to create Verilog functions with a limited number of gates, creating your own schematics and layouts helps optimize the functions for speed and space. By being creative with transistor placements and taking advantage of physical layers, the space between transistors can be minimized and metal layers that would have otherwise routed signals can be removed.

5.3 Design Compiler

A library containing properties of each cell is needed for the Synthesis step in the design process. This is done using the Encounter Library Characterization (ELC) process. Several steps are required using automated scripts and manual file modifications in order to format the library file for correct operation. The designer is able to view the parameters of each cell such as I/O dependencies, electrical timing delays, and physical properties throughout the process [8].

The Synopsys Design Compiler (DC) is the tool that creates a netlist of all the cells and interconnects representing a Verilog IC description. The netlist (structural Verilog) will be used for automatic placing and routing in the next design step. The DC tool uses many simulations to optimize constraints for physical area (number of cells), signal propagation, and timing delays. This tool will also characterize the maximum clock speed that will allow for proper data capture in registers. There are many other options available controlled by setup commands that can be found in the reference manual.

5.4 Place and Route

The next step in the design process is to use the SOC Encounter tool to place and route all the cells and signals, corresponding to the generated netlist, in a large layout [8]. There are many stages in this tool which set up placement for power planes, cell grids, and clock and signal routing. Once this is completed, a final routing is done connecting all I/O pins to pads that are available to access from the physical world. Throughout the process the designer is able to make his or her own modifications to the layouts, including fixes for DRC errors that can be introduced by the tools. A fully routed processor using our controller with regular FFs is available in Appendix D.

5.5 Fabrication and Testing

The final step in the design process is to send all the files required for fabrication to a manufacturer. Once the IC is fabricated it comes back for physical testing. In the case of the Digital IC Testing class, the Tektronix LV500 is used [9]. The tester applies input signals to the device under test (DUT) and records the output patterns. Various clock speeds and voltages can be tested. When testing a processor, the entire instruction set must be verified. This usually starts by testing power/reset, then individual instructions, and finally complete programs.

Although using this methodology to test ICs will show the designer limits in functionality, it is difficult to pinpoint exact failure locations within the chip. To do this, specific test patterns need to be applied where errors at specific output locations will tell the designer where failures occurred. This is important in determining errors due to faulty logic or problems with fabrication, especially when a new manufacturing process is being used.

6 Fault Testing

Testing the integrity of interconnections between gates and transistor level failures becomes a completely new problem in and of itself. If a design is structurally correct but broken wires or stuck transistors persist, how can a designer pinpoint the problem? The idea is to run test vectors through the IC that will show different outputs between the fault-free

design (or simulation) and a faulty version. These vectors, also called patterns, need a high coverage percentage in order to have a high confidence rating in the design [10].

The main algorithms used to create test vectors are D-algorithm, FAN, and PODEM. Although different in run time and solving methodology, they all come down to exploiting the Boolean properties of gates [11].

Example: given a 2-input AND gate. If the first input is 0, the output is 0 no matter what the second input is. This is called a controlling value. On the other hand, if the first input is 1, the output will reflect the value of the second input. This is a non-controlling value.

Another example: given a 2-input OR gate. The controlling value here is a 1, because the output will be 1 no matter what the second input is. If 0 is on the first input, the output will reflect the second input, therefore 0 if the non-controlling value for the OR gate.

Based on the above properties, the algorithms can test each gate and propagate faulty conditions to the output. If a single input to a gate is broken, first the alternate input is given a non-controlling value, then the faulty input is toggled and allowed to propagate to the output. At this point the software can check whether this was the expected value or not. Given a circuit with unknown faults, each input pattern needs to propagate values across specific paths within the circuit.

7 Design For Test Compiler & TetraMax

Synopsys has two main tools used for this type of testing: the Design For Test (DFT) compiler and TetraMax. We researched how to use these tools, how much access script commands give us, and later show our results. These tools can test faults in both combinational and sequential logic [12]. In the case of sequential logic that uses FFs, a special Scan FF needs to be developed. This gives the tools or users direct control over the values inside registers. It allows for propagating values into and out of the data path [13].

7.1 DFT

The DFT compiler uses additional features on top of the standard DC. It modifies the structural Verilog produced through DC by replacing standard FFs with Scan FFs. The tool is smart enough to update all the netlists associated with the new design. It runs its own DRC tests and is able to fix several clock and reset errors by itself. Synopsys also has a GUI based tool called Design Vision which lets the designer view a full schematic of his or her structural Verilog before and after Scan FF replacement [12].

7.2 TetraMax

Another Synopsys tool, called TetraMax ATPG, is used to generate the test patterns. This tool crawls through the netlist by building reduced ordered binary decision diagrams (ROBDDs) corresponding to Synopsys‘ proprietary algorithm [13]. Their algorithm is most likely optimized to take advantage of the best features in current fault detection algorithms. This tool will determine all testable faults along with those that are unreachable or untestable due to issues like redundancy. It will generate a testbench compatible file corresponding to all the generated patterns.

8 Scan Flip-Flop Design

The difference between ordinary FFs and Scan FFs is that the latter has more than one data input pin which is activated during test modes. There are two types of Scan FFs we considered: ones that share a clock with the rest of the IC while utilizing a ‘scan enable to activate the ‘scan input and the kinds that use a separate ‘scan clock’ to switch between data sources [12]. We decided to test the type using the scan enable signal.

The simplest way of designing Scan FFs like this is to add a multiplexer (MUX) on the data input line. The scan enable signal is the select bit on the MUX and controls whether the FF receives its input from the data path or scan

input. We designed 3 slightly different Scan FFs for this project. The first used a FF, which contained an active low reset signal, and a MUX from our own cell library. The second used cells from the UofUDigital_v1_2 library accessible to VLSI students[8]. The third was modeled based on a conservative design found in [10].

9 Future Work

Currently, we have only been successful using combinational ATPG in the stand alone TetraMax tool. The Scan FF designs are causing a problem when being inserted into a chain. Future work would be in getting the sequential ATPG to work so full test vectors can be generated for the controller module. We would then apply this whole process on our entire design. Once completed, fabrication and actual testing of scan enabled circuitry is designed to validate the VCEGAR and TetraMax fault detection.

10 Results

10.1 VCEGAR

We set up the conditionals in VCEGAR based on how our controller model would function in the real world. We assume the machine either boot ups for the first time or is reset to its initial stage. At this time the processor will begin the sequence of reading the first program instruction. The precondition set up is as follows:

```
initial
        begin
                state <= FETCH1;
        end
```

Below is a snippet of the postcondition. The property will check these against the abstraction model. Appendix F shows our original controller.v file and Appendix B shows the complete precondition and postcondition that is inserted into the file for VCEGAR.

```
always begin
                assert prop: 0 < state && state < 15
                                && 0 < nextstate
                                && nextstate < 15
                                ...
```

We were able to verify that our controller met our specifications. The abstraction model was built with 390570 BDD nodes. It exhausted all possible inputs which we could not have been able to do otherwise with a conventional testbench. Appendix G shows the VCEGAR output during abstract model generation. Appendix H shows the BDD generation results.

10.2 DFT & TetraMax

Now that we have formally verified our controller using software, our next change in the standard design flow of IC design is to generate a testable circuit with automatically generated test patterns for physical verification.

The following results compare between the regular synthesized controller using DC and the Scan FF inserted version using DFT. These results will only show our best case Scan FF which turned out the be the one built from the UofUDigital_v1_2 library. Appendix I shows our script for running the DFT compiler.

From	Size (μm square)	Number of Nets (wires with same label)
DC:	384	90
DFT:	417	103

The size increase between these two circuits is only 8%. It's within most sizing constraints to be placed, routed, and fabricated using the same internal chip area as the original would have.

After running the ATPG tool, we found a high coverage percentage of faults. This means that if a fault occurs in the fabricated model, there's a large chance the test patterns can pinpoint the exact area of failure. Below we outline the types of faults detected. Appendix J shows the script for TetraMax.

Fault Type	# of faults
Detected	568
Undetectable	1
ATPG untestable	5
Not detected	0
total faults	574
test coverage	99.13%

Along with the fault coverage information, we have the input test vectors that were generated for the combinational version of our design. Below is a sample of our input patterns for combinational fault coverages.

```
pattern = 0; // 200
#0 PI = 23'b11110100110000110001001;
#0;
XPCT = 8'b10110111;
MASK = 8'b11111111;
#0 ->measurePO_default_WFT;
#100; // 300

pattern = 1; // 300
#0 PI = 23'b01011100011111101010110;
#0;
XPCT = 8'b11011010;
MASK = 8'b11111111;
#0 ->measurePO_default_WFT;
#100; // 400
```

Based on the results from our new design flow, we are confident most logic and fabrication errors could be found and corrected before the main chip was fabricated. Although there is a small number of undetected faults, the errors that cannot be traced would most likely point to those locations. A designer could then use Design Vision to see where those occur in the schematic and make modifications as needed.

11 Contributions

Anh:

- Verilog Tool Research
- VCEGAR (setup, propertys creation, run)
- Experiment with different SMV
- Create script for DFT Compiler and TetraMax
- Test different compiling directives ...

Andrzej:

- Verilog Tool Research
- Scan flip flop designs (schematics, behavioral, layouts, and testing)
- Create script for DFT Compiler and TetraMax
- Analyze the results
- Experiment with different scan design ...

References

[1] A. S. Initiative, "Open verification library (ovl) technical subcommittee."

[2] Doulos, "The designer's guide to psl."

[3] Doulos, "The guide to systemverilog."

[4] D. K. Tala, "Systemverilog assertions."

[5] N. Sinha, "Model checking @cmu model checking @cmu model checking @cmu the smv system."

[6] McMillan, "The cadence smv model checker."

[7] NuSMV, "Nusmv: a new symbolic model checker."

[8] *Digital VLSI Chip Design with Cadence and Synopsys CAD Tools.* Pearson Education Inc., 2010.

[9] E. Brunvand, "Vlsi testing."

[10] V. S. Ashok Kumar Suhag, "Delay testable enhanced scan flip-flop: Dft for high fault coverage," *2011 International Symposium on Electronic System Design*, 2011.

[11] e. a. Shi, Junhao, "Passat: Efficient sat-based test pattern generation for industrial circuits.," *VLSI, 2005. Proceedings. IEEE Computer Society Annual Symposium on. IEEE, 2005.*, 2005.

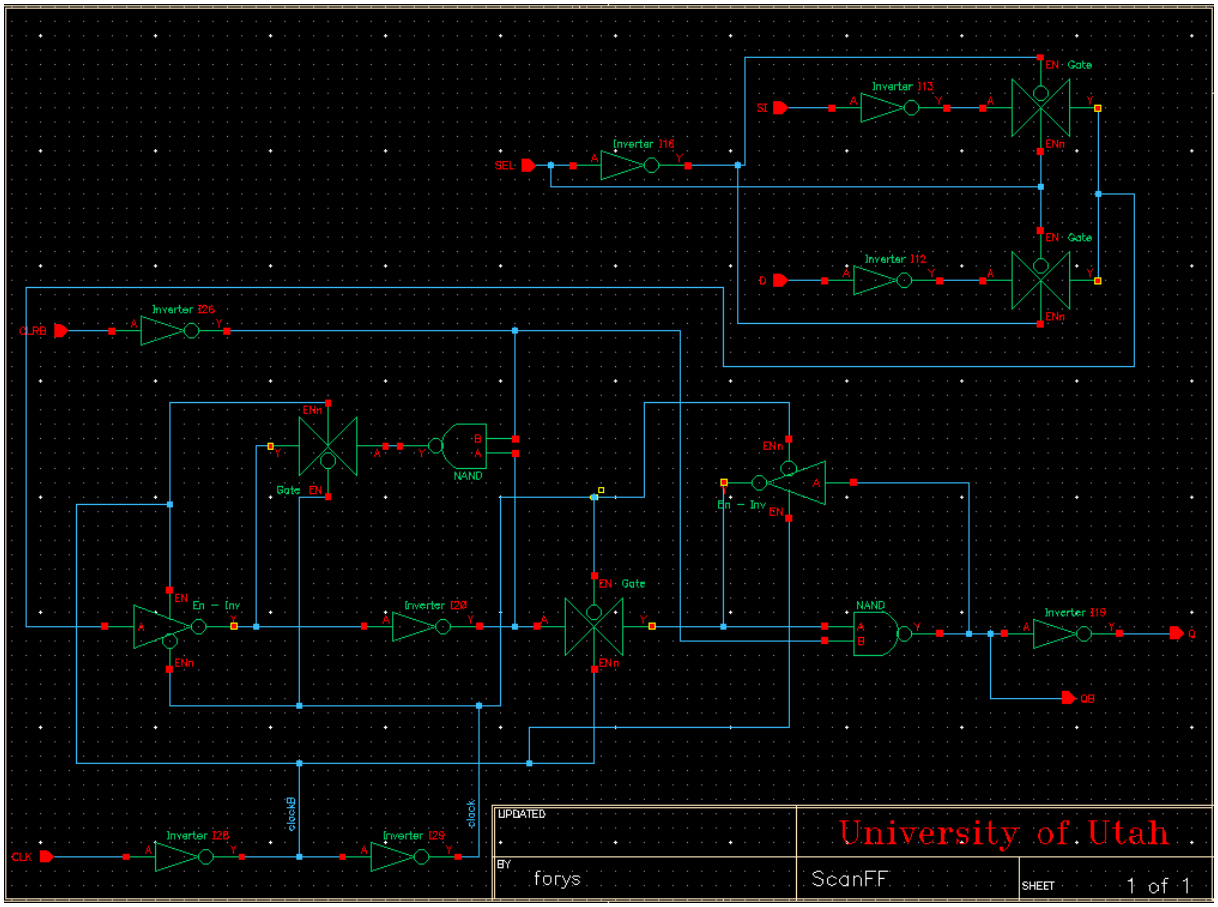
[12] *Synopsys DFT Compiler User Guide*.

[13] *Synopsys TetraMax User Guide*.

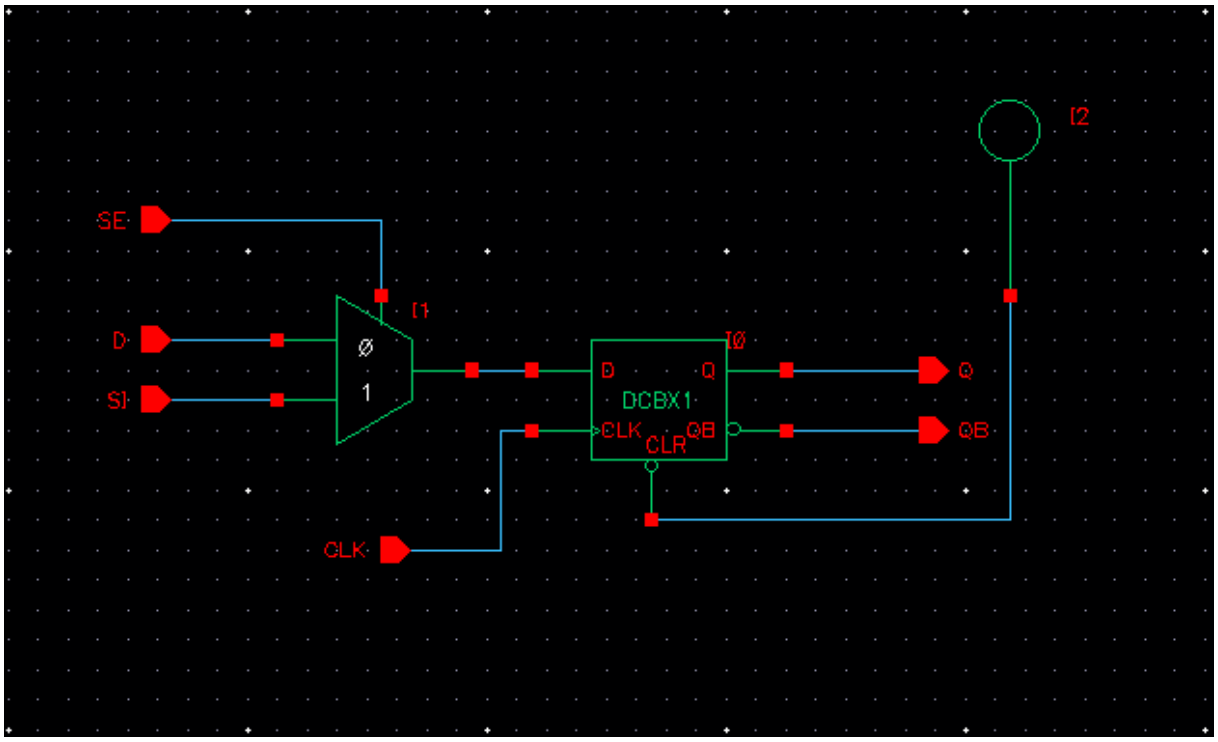
[14] D. V. . S. . P. . J. P. Perinbam, "Reduction of testing power with pulsed scan flip-flop for scan based testing," *Proceedings of 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies (ICSCCN 2011)*, 2011.

A Scan FF Schematics

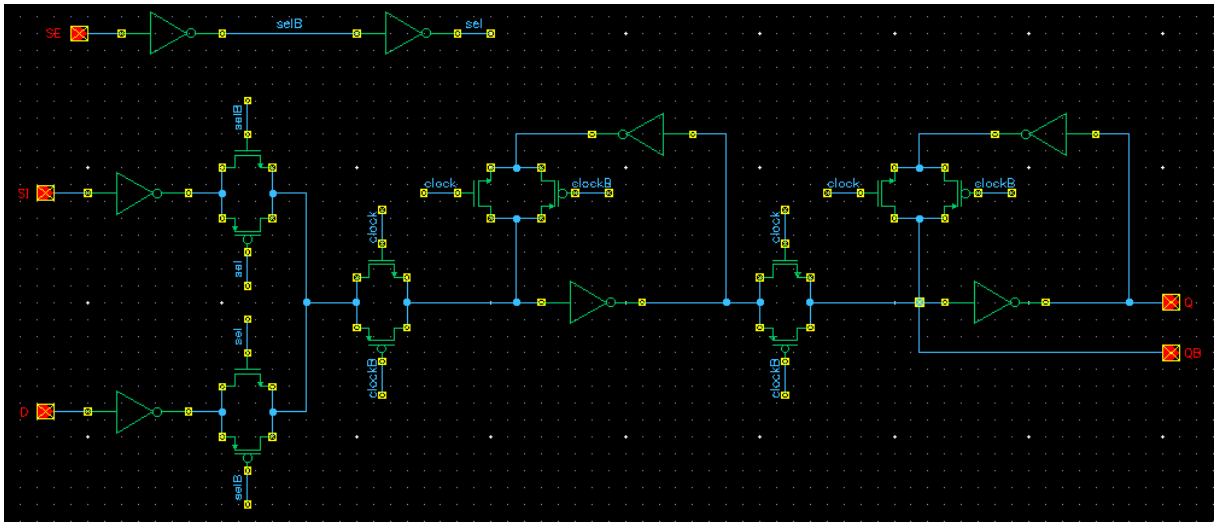
A.1 Our Library



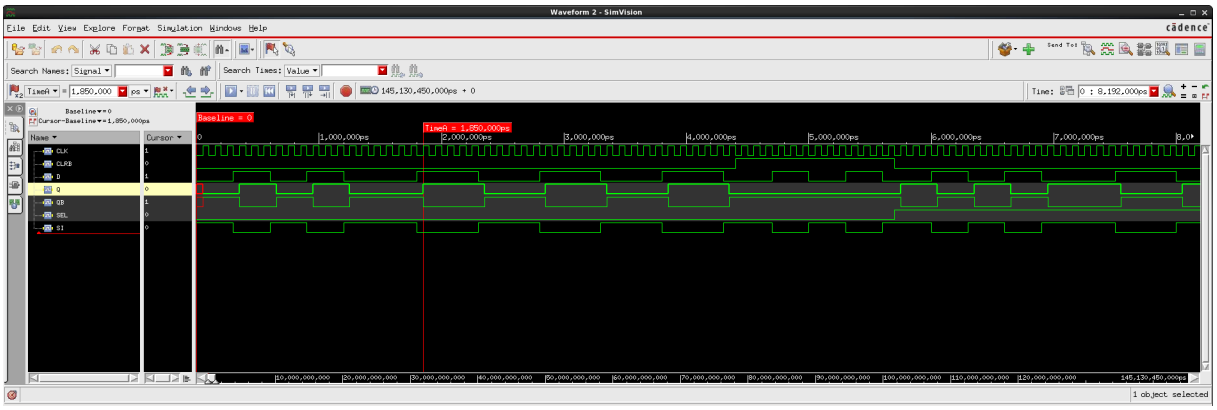
A.2 UofUDigital_v1_2 Library



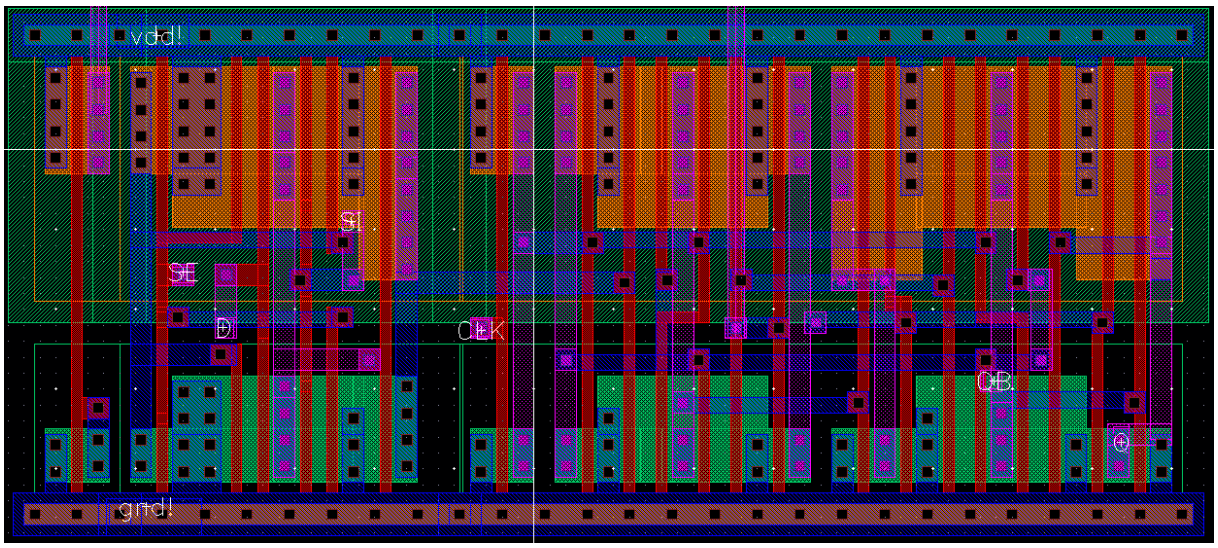
A.3 Based on [14]



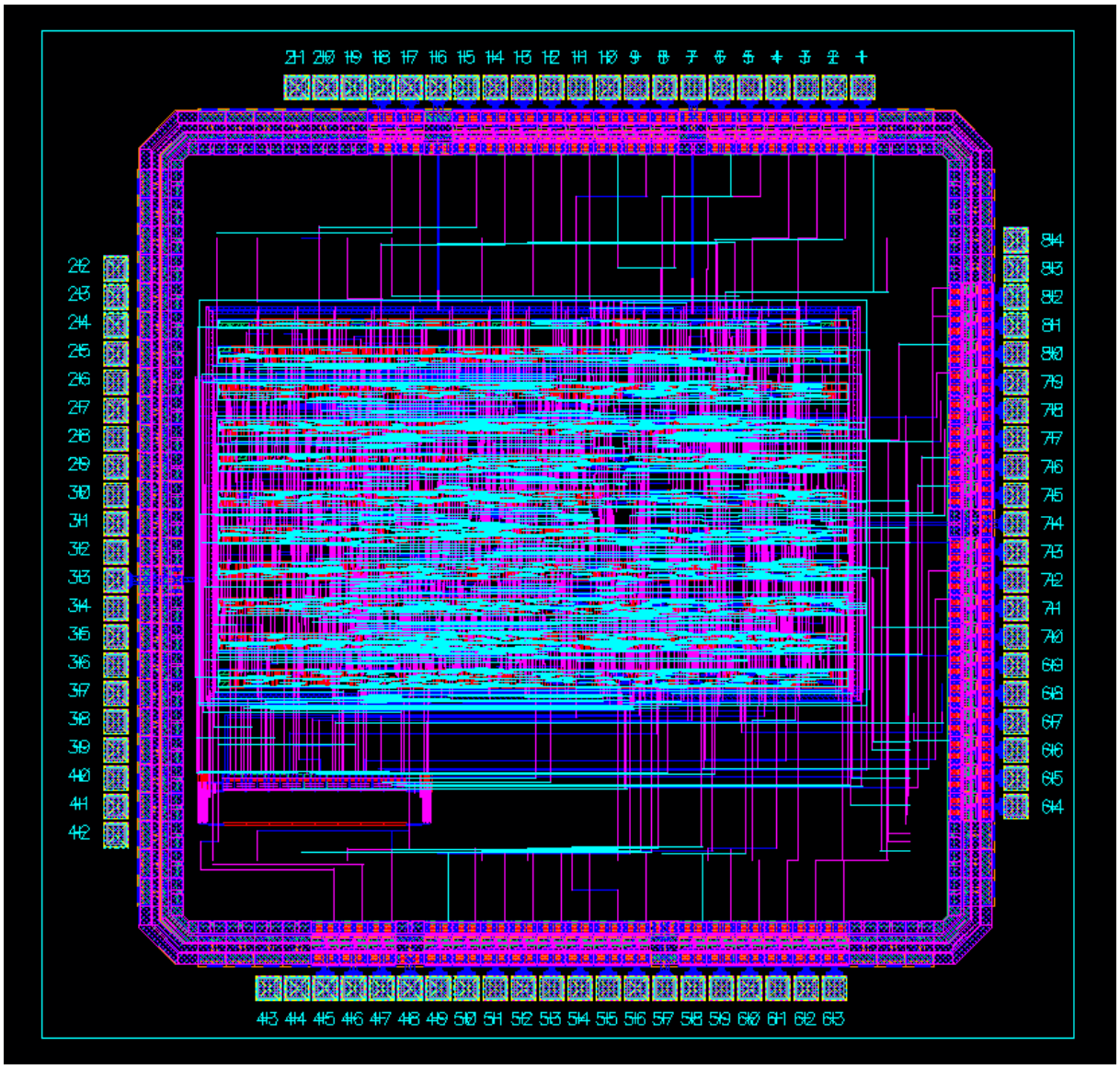
B TestBench Waveform



C Main Scan FF Layout



D Example of Fully Routed Processor (using our controller, no Scan FFs)



E controller.v

```
module main(input clk, reset,
                input       [5:0] op//,
                //input             zero,
                //output reg        memread, memwrite, alusrca, memtoreg, iord,
                //output            pcen,
                //output reg        regwrite, regdst,
                //output reg [1:0]  pcsource, alusrcb, aluop,
                );//output reg [3:0] irwrite);

  parameter    FETCH1  =  4'b0001;
  parameter    FETCH2  =  4'b0010;
  parameter    FETCH3  =  4'b0011;
```

```
parameter   FETCH4  =  4'b0100;
parameter   DECODE  =  4'b0101;
parameter   MEMADR  =  4'b0110;
parameter   LBRD    =  4'b0111;
parameter   LBWR    =  4'b1000;
parameter   SBWR    =  4'b1001;
parameter   RTYPEEX =  4'b1010;
parameter   RTYPEWR =  4'b1011;
parameter   BEQEX   =  4'b1100;
parameter   JEX     =  4'b1101;
parameter   ADDIWR  =  4'b1110; // added for ADDI

parameter   LB      =  6'b100000;
parameter   SB      =  6'b101000;
parameter   RTYPE   =  6'b0;
parameter   BEQ     =  6'b000100;
parameter   J       =  6'b000010;
parameter   ADDI    =  6'b001000; /// added for ADDI

reg [3:0] state, nextstate;
//reg       pcwrite, pcwritecond;

// state register
always @(posedge clk)
   if(reset) state <= FETCH1;
   else state <= nextstate;

// next state logic
always @(*)
   begin
      case(state)
         FETCH1:  nextstate <= FETCH2;
         FETCH2:  nextstate <= FETCH3;
         FETCH3:  nextstate <= FETCH4;
         FETCH4:  nextstate <= DECODE;
         DECODE:  case(op)
                     LB:      nextstate <= MEMADR;
                     SB:      nextstate <= MEMADR;
                     ADDI:    nextstate <= MEMADR;  // added for ADDI
                     RTYPE:   nextstate <= RTYPEEX;
                     BEQ:     nextstate <= BEQEX;
                     J:       nextstate <= JEX;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         MEMADR:  case(op)
                     LB:      nextstate <= LBRD;
                     SB:      nextstate <= SBWR;
                  ADDI:    nextstate <= ADDIWR;  // added for ADDDI
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         LBRD:    nextstate <= LBWR;
         LBWR:    nextstate <= FETCH1;
         SBWR:    nextstate <= FETCH1;
         RTYPEEX: nextstate <= RTYPEWR;
```

```
            RTYPEWR: nextstate <= FETCH1;
            BEQEX:   nextstate <= FETCH1;
            JEX:     nextstate <= FETCH1;
          ADDIWR:  nextstate <= FETCH1; // added for ADDI
            default: nextstate <= FETCH1; // should never happen
          endcase
        end

    always @(*)
        begin
            // set all outputs to zero, then conditionally assert just the appropriate ones
            irwrite <= 4'b0000;
            pcwrite <= 0; pcwritecond <= 0;
            regwrite <= 0; regdst <= 0;
            memread <= 0; memwrite <= 0;
            alusrca <= 0; alusrcb <= 2'b00; aluop <= 2'b00;
            pcsource <= 2'b00;
            iord <= 0; memtoreg <= 0;
            case(state)
                FETCH1:
                    begin
                        memread <= 1;
                        irwrite <= 4'b0001; // change to reflect new memory
                        alusrcb <= 2'b01;   // get the IR bits in the right spots
                        pcwrite <= 1;       // FETCH 2,3,4 also changed...
                    end
                FETCH2:
                    begin
                        memread <= 1;
                        irwrite <= 4'b0010;
                        alusrcb <= 2'b01;
                        pcwrite <= 1;
                    end
                FETCH3:
                    begin
                        memread <= 1;
                        irwrite <= 4'b0100;
                        alusrcb <= 2'b01;
                        pcwrite <= 1;
                    end
                FETCH4:
                    begin
                        memread <= 1;
                        irwrite <= 4'b1000;
                        alusrcb <= 2'b01;
                        pcwrite <= 1;
                    end
                DECODE: alusrcb <= 2'b11;
                MEMADR:
                    begin
                        alusrca <= 1;
                        alusrcb <= 2'b10;
                    end
                LBRD:
```

```
                begin
                    memread <= 1;
                    iord    <= 1;
                end
            LBWR:
                begin
                    regwrite <= 1;
                    memtoreg <= 1;
                end
            SBWR:
                begin
                    memwrite <= 1;
                    iord     <= 1;
                end
            RTYPEEX:
                begin
                    alusrca <= 1;
                    aluop   <= 2'b10;
                end
            RTYPEWR:
                begin
                    regdst   <= 1;
                    regwrite <= 1;
                end
            BEQEX:
                begin
                    alusrca     <= 1;
                    aluop       <= 2'b01;
                    pcwritecond <= 1;
                    pcsource    <= 2'b01;
                end
            JEX:
                begin
                    pcwrite  <= 1;
                    pcsource <= 2'b10;
                end
          ADDIWR:  // new state for addi writeback
          begin
            regwrite <= 1;
                end
        endcase
    end
  assign pcen = pcwrite | (pcwritecond & zero); // program counter enable

endmodule
```

F VCEGAR Pre and Post Conditions

```
 // insert after parameters (but before case statement) for vcegar
  initial
    begin
       state <= FETCH1;
    end
```

```
  // insert after state machine (case statement) for vcegar
  always begin
     assert prop: 0 < state && state < 15 && 0 < nextstate && nextstate < 15
       && ((state == FETCH1  && nextstate == FETCH2)
         || (state == FETCH2  && nextstate == FETCH3)
         || (state == FETCH3  && nextstate == FETCH4)
         || (state == FETCH4  && nextstate == DECODE)
         || (state == DECODE  && ((nextstate == MEMADR  && (op == LB || op == SB || op == AI
                        || (nextstate == RTYPEEX && op == RTYPE)
                        || (nextstate == BEQEX   && op == BEQ)
                        || (nextstate == JEX     && op == J)
                        || (nextstate == FETCH1)))
         || (state == MEMADR  && ((nextstate == LBRD    && op == LB)
                        || (nextstate == SBWR    && op == SB)
                        || (nextstate == ADDIWR  && op == ADDI)
                        || (nextstate == FETCH1)))
         || (state == LBRD    && nextstate == LBWR)
         || (state == LBWR    && nextstate == FETCH1)
         || (state == SBWR    && nextstate == FETCH1)
         || (state == RTYPEEX && nextstate == RTYPEWR)
         || (state == RTYPEWR && nextstate == FETCH1)
         || (state == BEQEX   && nextstate == FETCH1)
         || (state == JEX     && nextstate == FETCH1)
         || (state == ADDIWR  && nextstate == FETCH1))
       && (memread == 0  || (memread == 1  && (state == FETCH1 || state == FETCH2
|| state == FETCH3 || state == FETCH4 || state == LBRD)))
       && (memwrite == 0 || (memwrite == 1 && (state == SBWR)))
       && (alusrca == 0  || (alusrca == 1  && (state == MEMADR || state == RTYPEEX || state
       && (memtoreg == 0 || (memtoreg == 1 && (state == LBWR)))
       && (iord == 0     || (iord == 1     && (state == LBRD   || state == SBWR)))
       && (pcen == 0     || (pcen == 1     && (state == JEX     || state == FETCH1
                                || state == FETCH2
                                || state == FETCH3
                                || state == FETCH4
                                || (state == BEQEX && zero == 1))))
       && (regwrite == 0 || (regwrite == 1 && (state == LBWR   || state == RTYPEWR || state
       && (regdst == 0   || (regdst == 1   && (state == RTYPEWR)))
       && (pcsource < 3  && (pcsource == 0 || (pcsource == 1   && state == BEQEX)
|| (pcsource == 2  && state == JEX)))
       && (alusrcb == 0  || (alusrcb == 1  && (state == FETCH1 || state == FETCH2
|| state == FETCH3 || state == FETCH4))
                || (alusrcb == 2  && (state == MEMADR))
                || (alusrcb == 3  && (state == DECODE)))
       && (aluop < 3     && (aluop == 0    || (aluop == 1      && state == BEQEX)
|| (aluop == 2     && state == RTYPEEX)))
       && (irwrite == 0  || (irwrite == 1  && state == FETCH1)
                || (irwrite == 2  && state == FETCH2)
                || (irwrite == 4  && state == FETCH3)
                || (irwrite == 8  && state == FETCH4));
  end

endmodule
```

G VCEGAR Abstracts Model ([...] = shortened repetitive output for reader convenience)

```
-- automatically generated by VCEGAR

MODULE main

-- variables of the abstract program

VAR b0: boolean;  -- main.aluop < 3
VAR b1: boolean;  -- main.nextstate < 15
VAR b2: boolean;  -- main.pcsource < 3
[...]
VAR b78: boolean; -- main.regdst
VAR b79: boolean; -- main.regwrite
VAR b80: boolean; -- main.zero
-- Hack for Cadence SMV
TRANS (b0 | !b0)
TRANS (b1 | !b1)
TRANS (b2 | !b2)
[...]
TRANS (b78 | !b78)
TRANS (b79 | !b79)
TRANS (b80 | !b80)

-- Initial states cluster no. 0

INIT (b0 & b2 & b3 & b24 & !b25 & !b26 & !b27 & !b28 & !b29 & !b30 & !b31 & !b32 & !b33 &
[...]

-- the transition constraints for cluster no. 1

TRANS (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !next(b9) &
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !ne
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & nex
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & !next(b7) & next(b8) & !nex
              | (next(b1) & !next(b4) & !next(b5) & !next(b6) & next(b7) & !next(b8) & !nex
              | (next(b1) & !next(b4) & !next(b5) & next(b6) & !next(b7) & !next(b8) & !nex
              | (next(b1) & !next(b4) & next(b5) & !next(b6) & !next(b7) & !next(b8) & !nex
              | (next(b1) & next(b4) & !next(b5) & !next(b6) & !next(b7) & !next(b8) & !nex
[...]
TRANS !( b4 & next(b30) )
TRANS !( b4 & next(b31) )
TRANS !( b4 & next(b32) )
[...]
TRANS !( b16 & next(b35) )
TRANS !( b16 & next(b37) )
```

```
TRANS !( b17 & next(b30) )
-- addition of weakest pre-condition constraints

-- the specification

SPEC AG (b71 & b3 & b70 & b1 & (b24 & b5 | b25 & b6 | b26 & b7 | b27 & b8 | b28 & (b9 & (b2
```

H VCEGAR Results

```
Copyright 1996 Cadence Berkeley Labs. Cadence Design Systems.

Garbage collecting nodes (nodes=7837,assocs=83)...done (nodes=7832,assocs=83)

{}
==
user time ..........................................................0 s
system time ........................................................0 s

Variable order
==============
b0
b2
b3
[...]
b77
b69
b80
[...]
[BDD node GC: nodes before = 10051, nodes after: 1230]
[BDD node GC: nodes before = 10028, nodes after: 1493]
.....................................................................487
transition relation ...............................................1409
[BDD node GC: nodes before = 5602, nodes after: 1575]
total bdd nodes = 1575, total used = 1490, wasted = 85 (e.g. 9c1e3e4)
total part nodes = 265, total used = 265, wasted = 0
user time ..................................................0.0333333 s
system time ........................................................0 s

Reachable states
================
[BDD node GC: nodes before = 10403, nodes after: 5792]
[BDD node GC: nodes before = 35111, nodes after: 15107]
Memory limit reached. Garbage collecting.
[BDD node GC: nodes before = 40950, nodes after: 22160]
Memory limit reached. Garbage collecting.
[BDD node GC: nodes before = 53235, nodes after: 33469]
Memory limit reached. Garbage collecting.
increasing apply cache size to 126727
[BDD node GC: nodes before = 131040, nodes after: 102264]
Memory limit reached. Garbage collecting.
increasing apply cache size to 262063
[BDD node GC: nodes before = 163800, nodes after: 132330]
Memory limit reached. Garbage collecting.
[BDD node GC: nodes before = 163800, nodes after: 152843]
```

```
[optimized conjunctive relation: comps = 1, size = 1]
Optimized transition relation size: 1
$$$_mc_0 ..... 181951
Memory limit reached. Garbage collecting.
[BDD node GC: nodes before = 282555, nodes after: 183763]
[optimized conjunctive relation: comps = 1, size = 1]
Optimized transition relation size: 1
iteration 0 ..... 88
[optimized conjunctive relation: comps = 1, size = 947]
Optimized transition relation size: 947
reached states BDD size ..... 144
iteration 1 ..... 144
reached states BDD size ..... 198
iteration 2 ..... 90
reached states BDD size ..... 252
iteration 3 ..... 90
reached states BDD size ..... 369
iteration 4 ..... 131
reached states BDD size ..... 629
iteration 5 ..... 327
reached states BDD size ..... 843
iteration 6 ..... 262
user time ..... 0.566667 s
system time ..... 0 s
Model checking time: 0.566667
user time ..... 0.566667 s
system time ..... 0 s

Model checking results
======================
(AG $$$_mc_0 ) ..... true

See file "vcegar_tmp_abstract.warn" for warnings.
user time ..... 0.566667 s
system time ..... 0 s

Resources used
==============
user time ..... 0.566667 s
system time ..... 0 s
BDD nodes allocated ..... 390570
data segment size ..... 0
```

I Script for DFT

```
# * Modified by Anh Luong & Andrzej Forys
# * FALL 2012
# *
# * Author: Erik Brunvand, University of Utah
# *
# * General synthesis script for Synopsys. There should be
# * some general switches and parameters set in .synopsys_dc.setup
# * but other design-specific things are set here.
# * You should look carefully at everything above the
```

```
# * "below here shouldn't need to be changed" line.
# *
# * Note that lists that continue across a line need a backslash
# * to continue to the next line (if you have a bunch of
# * different verilog files, one per line, for example, or a bunch
# * of target library files). Make SURE there isn't a space after
# * the \ because that can cause Synopsys to complain...
# *
# * Once you've modified things to your project, invoke with:
# *
# * syn-dc -f syn-script
# *
# *

#  This script assumes that the following variables are defined
#  in the .synopsys_dc.setup file. You should make sure that
#  your .synopsys_dc.setup file is configured for your
#  cell library! If you want to override or
#  add to those search paths, you can do that here...
#
#  SynopsysInstall = path to synopsys installation directory
#  synthetic_library = designware files
#  symbol_library = logic symbols for making schematics
#

# search path should include directories with memory .db files
# as well as the standard cells
# Your library path may be empty if your library will be in
# your synthesis directory because "." is already on the path.
#set search_path [list . \
#[format "%s%s"  $SynopsysInstall /libraries/syn] \
#[format "%s%s"  $SynopsysInstall /dw/sim_ver] \
#/uusoc/facility/cad_common/local/Cadence/lib/OA/UofU_Digital_v1_2]

# target library list should include all target .db files
set target_library [list UofU_Digital_v1_2.db]

# synthetic_library is set in .synopsys_dc.setup to be
# the dw_foundation library.
set link_library [concat  [concat  "*" $target_library] $synthetic_library]

####################################
# Print to screen options   #
####################################
set verbose 1     ;# 1 Write reports to screen, 0 do not write reports to screen
set verbose_dft 1 ;# 1 Write reports to screen, 0 do not write reports to screen

####################################
# Synthesis   #
####################################

# below are parameters that you will want to set for each design

# list of all HDL files in the design
```

```
set myFiles [list controller.v]
set fileFormat verilog     ;# verilog or VHDL
set basename controller    ;# Top-level module name
set myClk clk              ;# The name of your clock
set virtual 0              ;# 1 if virtual clock, 0 if real clock

# compiler switches...
#set optimizeArea 0       ;# 1 for area, 0 for speed
set useUltra 1           ;# 1 for compile_ultra, 0 for compile
                         # mapEffort, useUngroup are for
                         # non-ultra compile...
set mapEffort1 medium    ;# First pass - low, medium, or high
set mapEffort2 medium    ;# second pass - low, medium, or high
set useUngroup 1         ;# 0 if no flatten, 1 if flatten

# Timing and loading information
set myPeriod_ns 25        ;# desired clock period (sets speed goal)
set myClkLatency_ns 0.3 ; # clock network latency
set myInDelay_ns 0.25    ;# delay from clock to inputs valid
set myOutDelay_ns 0.25   ;# delay from clock to output valid
set myInputBuf INVX4     ;# name of cell driving the inputs
set myLoadLibrary UofU_Digital_v1_2 ;# name of library the cell comes from
set myLoadPin Y          ;# name of pin that outputs drive


# Control the writing of result files
set runname struct       ;# Name appended to output files

# the following control which output files you want. They
# should be set to 1 if you want the file, 0 if not
set write_v 1            ;# compiled structural Verilog file
set write_ddc 0          ;# compiled file in ddc format (XG-mode)
set write_sdf 1          ;# sdf file for back-annotated timing sim
set write_sdc 1          ;# sdc constraint file for place and route
set write_rep 0          ;# report file from compilation
set write_pow 0          ;# report file for power estimate

####################################
# DFT Switches   #
####################################
set dft_runname scan    ; # name appended to output files
set scan_library [list foo.db] ; # Library with scan chain cells
#set scancell SCANFF ; # Name of ScanFF Cell

# Setup timing variables for dft_drc command
set test_default_delay 0 ; # define time when values are applied to input ports
set test_default_bidir_delay 0 ; # Defines  the  default switching time of bidirectional p
set test_default_strobe 40 ; # default strobe time in a test cycle for output ports and bi
set test_default_period 100 ; # Defines the default length of a test vector cycle

# Setup scan chain for insert_dft
#set test_default_scan_style multiplexed_flip_flop; # Defines  the  default scan style for

#*****************************************************
```

```
#*    below here shouldn't need to be changed...          *
#*********************************************************

###################################
# remove any other designs from design compiler's memory
###################################
remove_design -all
# IMPORTING DESIGN

# analyze and elaborate the files
analyze -format $fileFormat -lib WORK $myFiles
elaborate $basename -lib WORK -update
current_design $basename

# The link command makes sure that all the required design
# parts are linked together.
# The uniquify command makes unique copies of replicated
# modules.
link
uniquify

# SETUP CONSTRAINTS

# now you can create clocks for the design
# and set other constraints
if {  $virtual == 0 } {
  create_clock -period $myPeriod_ns $myClk
} else {
  create_clock -period $myPeriod_ns -name $myClk
}
#*** add this shit
set_clock_latency $myClkLatency_ns $myClk

# Set the driving cell for all inputs except the clock
# The clock has infinte drive by default. This is usually
# what you want for synthesis because you will use other
# tools (like SOC Encounter) to build the clock tree
# (or define it by hand).
if {  $virtual == 0 } {
    set_driving_cell  -library $myLoadLibrary -lib_cell $myInputBuf [all_inputs] \
} else {
  set_driving_cell  -library $myLoadLibrary -lib_cell $myInputBuf  \
       [remove_from_collection [all_inputs] $myClk]
}

# set the input and output delay relative to myClk
if {  $virtual == 0 } {
    set_input_delay $myInDelay_ns -clock $myClk [all_inputs] \
} else {
    set_input_delay $myInDelay_ns -clock $myClk \
    [remove_from_collection [all_inputs] $myClk]
}
set_output_delay $myOutDelay_ns -clock $myClk [all_outputs]
```

```
# set the load of the circuit outputs in terms of the load
# of the next cell that they will drive, also try to fix
# hold time issues
set_load [load_of [format "%s%s%s%s%s" $myLoadLibrary "/" $myInputBuf "/" $myLoadPin]] [al
set_fix_hold $myClk

# FINISH SETUP CONSTRAINTS

# This command will fix the problem of having
# assign statements left in your structural file.
# But, it will insert pairs of inverters for feedthroughs!
set_fix_multiple_port_nets -all -buffer_constants

# COMPILING DESIGN

# now compile the design with given mapping effort
# and do a second compile with incremental mapping
# or use the compile_ultra meta-command
if {  $useUltra == 1 } {
  compile_ultra
} else {
  if {  $useUngroup == 1 } {
    compile -ungoup_all -map_effort $mapEffort1
    compile -incremental_mapping -map_effort $mapEffort2
 } else {
    compile -map_effort $mapEffort1
    compile -incremental_mapping -map_effort $mapEffort2
 }
}


#
check_design
# VIOLATIONS
report_constraint -all_violators

#********************************************************
#* now write out the results                            *
#********************************************************

set filebase [format "%s%s" [format "%s%s" $basename "_"] $runname]

# structural (synthesized) file as verilog
if {  $write_v == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".v"]
    redirect change_names \
   { change_names -rules verilog -hierarchy -verbose }
    write -format verilog -hierarchy -output $filename
}

# write out the sdf file for back-annotated verilog sim
# This file can be large!
if {  $write_sdf == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".sdf"]
```

```
    write_sdf -version 1.0 $filename
}

# this is the timing constraints file generated from the
# conditions above - used in the place and route program
if {  $write_sdc == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".sdc"]
    write_sdc $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later in XG mode (ddc format)
if {  $write_ddc == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".ddc"]
    write -format ddc -hierarchy -o $filename
}

# report on the results from synthesis
# note that > makes a new file and >> appends to a file
if {  $write_rep == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".rep"]
    redirect $filename { report_timing }
    redirect -append $filename { report_area }
}

# report the power estimate from synthesis.
if {  $write_pow == 1 } {
    set filename [format "%s%s%s" ./src/ $filebase ".pow"]
    redirect $filename { report_power }
}

# print report to the screen
if { $verbose == 1 } {
    report_design
    report_hierarchy
    report_timing -path full -delay max -nworst 3 -significant_digits 2 -sort_by group
    report_timing -path full -delay min -nworst 3 -significant_digits 2 -sort_by group
    report_area
    report_cell
    report_net
    report_port -v
    report_power -analysis_effort low
}

# Design reports
set filename [format "%s%s%s" ./reports/ $filebase ".design"]
redirect $filename { report_design }

# Hierarchy reports
set filename [format "%s%s%s" ./reports/ $filebase ".design"]
redirect -append $filename { report_hierarchy }

# Timing reports
set filename [format "%s%s%s" ./reports/ $filebase ".timing"]
```

```
redirect $filename { report_timing -path full -delay max -nworst 5 -significant_digits 2 -

set filename [format "%s%s%s" ./reports/ $filebase ".timing"]
redirect -append $filename { report_timing -path full -delay min -nworst 5 -significant_di

# Report_cell
set filename [format "%s%s%s" ./reports/ $filebase ".area"]
redirect $filename { report_area }

# Report_area
set filename [format "%s%s%s" ./reports/ $filebase ".area"]
redirect -append $filename { report_cell }

# Report port
set filename [format "%s%s%s" ./reports/ $filebase ".ports"]
redirect $filename { report_port -v}

# Report net
set filename [format "%s%s%s" ./reports/ $filebase ".net"]
redirect $filename { report_net }

# Report power
set filename [format "%s%s%s" ./reports/ $filebase ".pow"]
redirect $filename { report_power -analysis_effort low }

########################################
#### Insert Test Structures ###
########################################
# Update filebase
set filebase [format "%s%s" [format "%s%s" $basename "_"] $dft_runname]

# Update target library
set target_library [list $target_library $scan_library]

# Set Scan Chain Type
set_scan_configuration -style multiplexed_flip_flop

# AutoFix for Reset and Clock
set_dft_configuration -fix_reset enable -fix_clock enable

# Set Test Protocol
set test_default_period 100
set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port clk
set_dft_signal -view existing_dft -type Reset -active_state 1 -port reset
set_dft_signal -view existing_dft -type Constant -active_state 1 -port test_mode
create_test_protocol

# DFT Check
dft_drc -verbose

# Add delay in generated clocks
create_clock clk -period 1000
set_input_delay 250 si -clock clk
set_input_delay 150 se -clock clk
```

```
# Partial Scan
#set_scan_element false {state_reg_3_}
#set_scan_element false {state_reg_2_}
#set_scan_element false {state_reg_1_}
#set_scan_element false {state_reg_0_}

# Test-Ready Synthesis
compile -scan

# Read Design & Test Protocol
# Write out the test protocol and scan-ready design
#write_test_protocol -output [format "%s%s%s" ./reports/ $filebase ".spf"]
#write -format ddc -hierarchy -output [format "%s%s%s" ./reports/ $filebase ".ddc"]

# Read design and test protocol
#read_file -format ddc [format "%s%s%s" ./reports/ $filebase ".ddc"]
#current_design [format "%s%s%s" ./reports/ $filebase ".ddc"]:$basename
#link
#read_test_protocol [format "%s%s%s" ./reports/ $filebase ".spf"]

# Specify Scan Chain
set_scan_configuration -chain_count 1
set_scan_configuration -clock_mixing no_mix
set_dft_signal -view spec -type ScanDataIn -port si
set_dft_signal -view spec -type ScanDataOut -port so
set_dft_signal -view spec -type ScanEnable -port se -active_state 1
set_scan_path chain1 -scan_data_in si -scan_data_out so

# Memory Wrapper
#set_test_point_element -type observe [get_object_name [get_pins RAM_64B/D*]] -clock_signa
#set_test_point_element -type observe [get_object_name [get_pins RAM_64B/A*]] -clock_signa
#set_test_point_element -type control_01 [get_object_name [get_pins RAM_64B/Q*]] -clock_sig
#report_test_point_element

# Scan Preview
preview_dft -show all
preview_dft -test_points all

# Scan Chain Synthesis
insert_dft

# Scan Chain Identification
set_scan_state scan_existing

# DRC & Coverage
dft_drc -coverage_estimate

# Report Scan Information
report_scan_path -view existing_dft -chain all
report_scan_path -view existing_dft -cell all

# Prepare TetraMax script
change_names -hierarchy -rule verilog
```

```
write -format verilog -hierarchy -out [format "%s%s%s" ./src/ $filebase ".v"]
write -format ddc -hierarchy -output [format "%s%s%s" ./reports/ $filebase ".ddc"]
write_scan_def -output [format "%s%s%s" ./reports/ $filebase ".def"]
set test_stil_netlist_format verilog
write_test_protocol -output [format "%s%s%s" ./reports/ $filebase ".spf"]

####################################
# Quit dc
####################################
quit
```

J Script for TetraMax

```
#########################################################################
####
#### Modified by Anh Luong & Andrzej Forys
#### University of Utah
#### Fall 2012
####
#### TetraMax Script for ECE 128
#### Performs ATPG Pattern Generation for Synopsys Generic files
#### author: tjf
#### update: wgibb, spring 2010

#### note: this script will only run in TMAX TCL mode
#### start tmax like this:   tmax -tcl
#### source tmax_atpg.tcl
#########################################################################

#############################################################
#### local variables, designer must change these values ####
#############################################################
set top_module controller
set synthesized_files [list ./src/${top_module}_scan.v]
set cell_lib ./UofU_Digital_v1_2.v
set scan_lib ./foo.v
set stil_file [list ./reports/${top_module}_scan.spf]

# Continue execution when command returns an error
#set_command noabort

build -force

################################################
#### read in standard cells and user's design ###
################################################
# remove any other designs from design compiler's memory
read_netlist -delete
# read in standard cell library
read_netlist $cell_lib -library
# read in scan cell library
read_netlist $scan_lib -library
# read in user's synthesized verilog code
read_netlist $synthesized_files
```

```
################################################
#### BUILD and DRC test model
################################################
report_modules -all
run_build_model $top_module
# ignoring warnings like N20 or B10
# Set STIL file from DFT Compiler
set_drc $stil_file
# run check to see if synthesized code violates any testing rules
run_drc

################################################
#### Generate ATPG (patterns)- full sequential
################################################
# capture all faults, 9 capture cycles
#set_atpg -capture_cycles 9 -full_seq_atpg
#remove_faults -all
report_summaries faults patterns
add_faults -all
# run atpg in full sequential mode for better fault coverage
report_summaries faults patterns
run_atpg full_sequential_only
# write out patterns (overwrite old files)
report_summaries faults patterns
write_patterns ./src/${top_module}_tb_patterns.v -replace -internal -format verilog_single

################################################
#### Output reports
################################################
report_patterns -all >> ./reports/${top_module}.tmax.patterns
report_violations -all >> ./reports/${top_module}.tmax.violations
report_faults -summary -collapsed >> ./reports/${top_module}.tmax.coverage

################################################
#### Analyze Faults
################################################
# up to user to run these commands, they can inspect the faults and various reasons for the
#analyze_faults -class an
#analyze_faults -class an -verbose -max 3
#analyze_faults in_a_reg_reg/p_dregscan0/q -stuck 1

# Exit the program
quit
```