The Report Committee for Sharukh Shahajahan Shaikh
Certifies that this is the approved version of the following Report:

Implementation of Verification Methodologies

APPROVED BY
SUPERVISING COMMITTEE:

Jacob A. Abraham, Supervisor

Nur A. Touba

Implementation of Verification Methodologies

by

Sharukh Shahajahan Shaikh

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

The University of Texas at Austin

May 2018

Dedication

Dedicated to Family.

Acknowledgements

I would like to thank Prof. Jacob for providing me direction and liberty to work in domain of my interest. Also, for providing with the opportunity to be his Teaching Assistant for the courses VLSI 1 and Verification of Digital Systems, which helped me bolster my understanding of Digital Design. I would also like to thank the students of Verification course for providing me feedback on the developed lab exercises.

I would take this opportunity to specially thank Prof. Nur Touba for taking out his valuable time to be the reader for my report.

Abstract

Implementation of Verification Methodologies

Sharukh Shahajahan Shaikh, MSE

The University of Texas at Austin, 2018

Supervisor: Jacob A. Abraham

The increasing complexity of design elevates the importance of verification. This report explores different verification methodologies. The second chapter emphasizes the importance of testability and establishes the synthesis and DFT insertion flow using an SoC with ARM-Amber core as an example. Also, formal equivalence check is performed between the golden model, that is, RTL against its netlist. The third chapter delineates the design and formal verification of an Arbiter with APB slave configuration port. The design is extensively verified by writing SystemVerilog properties and we learn that the verification is only as good as the properties. Fourth chapter further explores formal verification with a different approach. The implemented x86 execution unit is formally verified by developing the its reference model and writing simple equality assertion checks. This approach exploits both, completeness of formal as well as includes the UVM reference model which reduces the long list of properties required for formal. The last chapter provides an approach to identify the critical registers in design. The critical flops in the design as a subset of all the registers which may have the most effect on the control flow of a module. This finds application in selecting the relevant auto-generated properties.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

A consequence of increasing size of Integrated Circuits (ICs) is the explosive growth in the complexity of verification, which has been the main bottleneck in the IC design cycle. More than 70% of development time is now required for verification [1], and this portion is still growing. Therefore, effective verification methodologies and techniques are essential.

The report explores different verification methodologies. The next chapter emphasizes the importance of testability and establishes the synthesis and DFT insertion flow. The techniques to fix DFT violations are discussed and formal equivalence check is performed between the RTL and the synthesized netlist. The design example used is an SoC with the ARM-Amber core [9], which is synthesized with DFT insertion using Cadence RTL Compiler and equivalence checking is performed using Cadence Conformal LEC.

The third chapter discusses the architecture of a developed DUT, an Arbiter with the APB slave configuration port and its formal verification by writing SystemVerilog assertions, assumptions, and cover properties. The design is extensively verified, and we learn that the verification is only as exhaustive as the properties specified. The formal verification has been performed with Cadence Jaspergold and the properties are proven to an infinite bound.

Following this, the fourth chapter builds on the exploration of formal verification with an alternate approach. The detailed microarchitecture of a performance-optimized implementation of an x86 execution unit is discussed and the design is formally verified by developing a reference model and writing simple equality assertion checks. The reference model, which is similar to the scoreboard in UVM, reduces the long list of

properties required by conventional formal verification. We are not restricted by the constraint that verification is as exhaustive as the set of properties we write in formal verification. Also, we need not worry about the coverage like in UVM, where it is not possible to cover all the test patterns via random test pattern generation. Such verification techniques can be exploited for DUTs such as execution units, decoders, floating point processors, DSP engines etc., for which it is simple to develop a reference model and the design is mostly combinational. Also, there is an additional advantage if the reference model and the DUT are designed by different designers or they are designed using different implementation styles, such as, Structural DUT vs Behavioral scoreboard or behavioral DUT vs RTL generated by High Level Synthesis (HLS).

The last chapter provides an approach to identify the critical registers in a design which have maximum impact on its control flow. The criticality value 'C' computed based on the several features is associated with each register in the design. The higher the value of 'C', the more crucial is the register. To find the criticality of the flops, feature extraction is first performed on the extracted timing paths of several designs and then a simple Artificial Neural Network (ANN) is employed to train and test on the data set [25]. Identifying critical registers finds application in fault tolerance.

Chapter 2: DFT insertion and Logic equivalence checking

INTRODUCTION

The testability of the design is of major concern to industry, and the DFT techniques provide support to test the fabricated chip comprehensively for quality and coverage [2]. The RTL of the digital block is generally described in the Hardware Description Language (HDL) such as Verilog. Once the design is verified with various verification methodologies, it is synthesized to netlist by synthesis tools which transform it for optimizing logic, area, power or adding DFT structures. Further, the physical design tool may significantly modify the netlist and through every step the behavior of the original RTL must be intact, and this is ensured by logic equivalence checking (LEC). In reality, it is a common practice to make manual edits to the netlist for minor changes known as Engineering Change Order [26], abbreviated as ECO. Therefore, it is a crucial verification step to perform logical equivalence checking of the netlist termed as a revised model with the original RTL of the design termed as the golden reference model.

In this section, an example SoC, the ARM-Amber core is synthesized using Cadence RTL Compiler tool with DFT insertion and LEC is performed with Cadence Conformal LEC tool between the synthesized netlist and the RTL.

SYNTHESIS WITH DFT INSERTION

Design setup

The process of scan insertion replaces the normal flops with special scan flops which allow us to observe and control the state of the design through the dedicated test

ports. With the scan structure support, the Automatic Test Pattern Generator (ATPG) tool such as Cadence Encounter test can generate compact tests for better fault coverage during scan simulation tests.

1. For synthesis setup in Cadence RTL compiler (RC) [3], we need to specify the target library paths and read the libraries to be used in synthesis.

   set_attribute lib_search_path Tcl_list

   set_attribute library library_list

2. After this, the HDL files of the SoC with ARM-Amber core are read, and the design is elaborated in RC. During this step, an RTL sanity check is performed for its feasibility to be synthesized.

   set_attribute hdl_search_path Tcl_list

   read_hdl hdl_files ...

   elaborate ...

3. After elaboration, the timing constraints need to be read; these provide the input/output delays, clock period, false paths, functional modes etc. in Synopsys Design Constraint (SDC) format.

DFT Rule Checker setup

After specifying the design constraints, we need to setup the tool to run the DFT rule checker [4].

1. Select the scan style as shown below. The muxed scan style is the most popular option and it requires a scan enable pin as the mux select and the scan-in and scan-out ports as shown in the Figure 1.

   set_attribute dft_scan_style muxed_scan

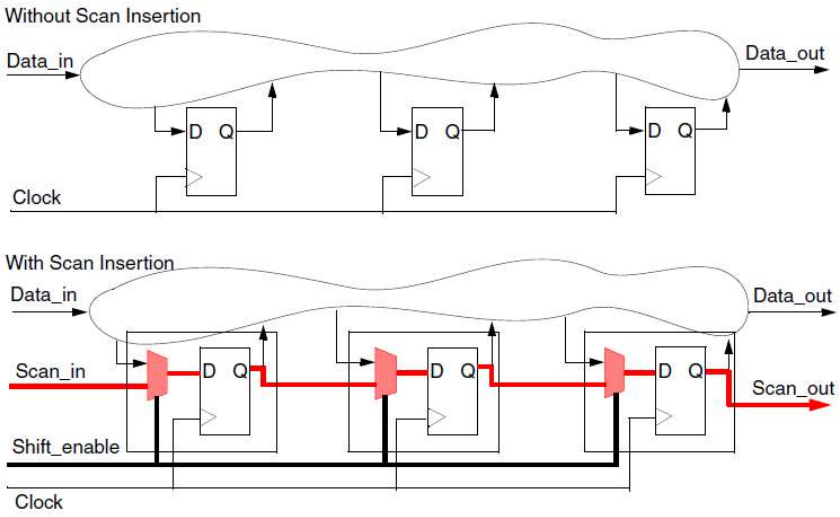

Figure 1: Muxed scan style [4]

2. To control the testable logic in the design, the test-mode signals are constrained during scan as below.

define_dft test_mode [-name test_signal] -active {low | high}

For instance, to make the muxed clock signal controllable, the input 'TM' is defined as an active high test-mode control signal, so that primary input clock 'clk' is propagated to the flops instead of divided clock as shown in the Figure 2.

define_dft test_mode -active high TM

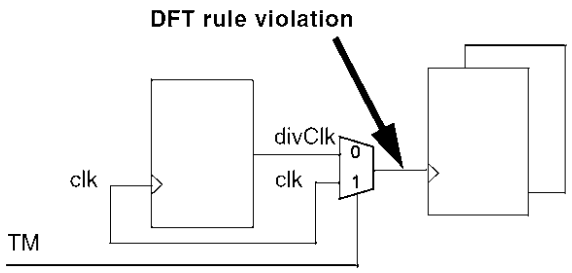

Figure 2: Controllable muxed clock

3. In the case of clock gating, during scan the 'testmode' signal needs to be active high, so that, 'gclk' is controllable. This ensures 'clk2' is passed to flops 'r2' and 'r3' as shown in the Figure 3.

define dft test mode -active high testmode

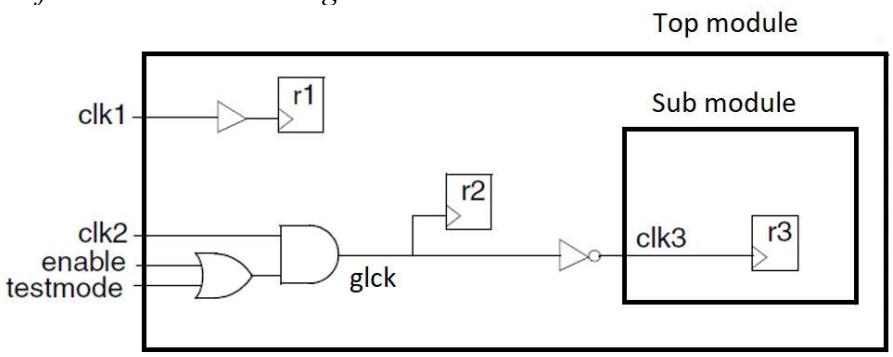

Figure 3: Controllable gated clock

4. All the test clocks in design need to be specified along with their period and phase relation.

define_dft test_clock -period <>...

5. The RTL elaboration will transform RTL to generic logic which may have "don't care" logic which will hinder the DFT rule checker when analyzing the DFT

structure, hence, logic optimization is performed with the following synthesis options.

synthesize -to_generic

synthesize -to_mapped

Run DFT Rule Checker

The check verifies the feasibility of scan insertion by identifying uncontrollable clocks and resets. The flops that fail the check will not be part of scan chain and their scannable status can be reported. Also, at this stage, DFT rule violations can be reported.

check_dft_rules

report dft_registers

report dft_violations

Fix DFT Violations

We need to fix the reported DFT clock rule violations and asynchronous set and reset violations in the design. For instance, the reset/set pins of the flops A to D in the Figure 4 are not controllable as the reset is internally generated [4]. These violations can be fixed as shown below by the insertion of the three separate test points as shown in the Figure 5. The signal TM is scan mode signal which is set to the value '1' during scan testing.

define_dft test_mode -name TM1 -active high TM

fix_dft_violations -async_set -async_reset -test_control TM1

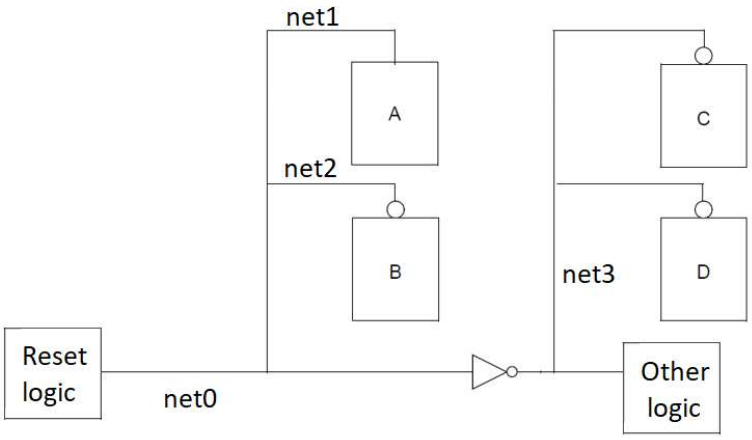

Figure 4: Circuit with asynchronous reset violations

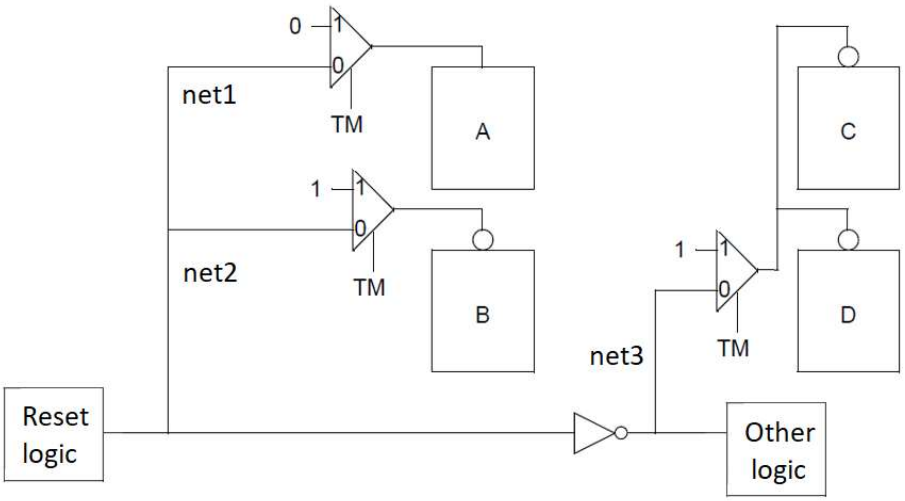

Figure 5: Circuit with fixed asynchronous reset violations

In the design, for optimizing the dynamic power, clock gating is auto-inserted by the synthesis tool, and uncontrollable clock gates may lead to the DFT rule violations. The following commands fixes these violations by inserting the appropriate test points as shown in Figures 6 and 7. The signal TM is scan mode signal which is set to the value '1' during scan testing.

define_dft test_mode -name tm -active high TM

fix_dft_violations -clock -test_control tm -test_clock_pin clk2

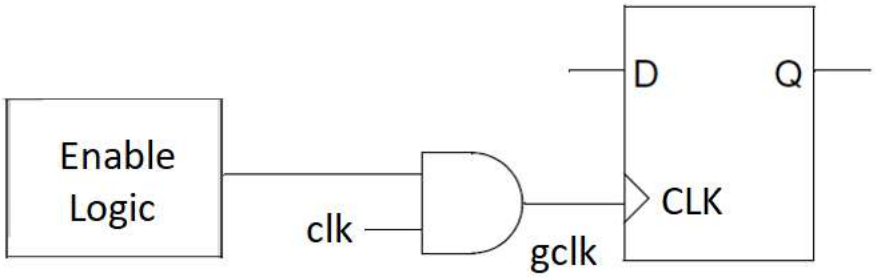

Figure 6: Circuit with clock gate violation

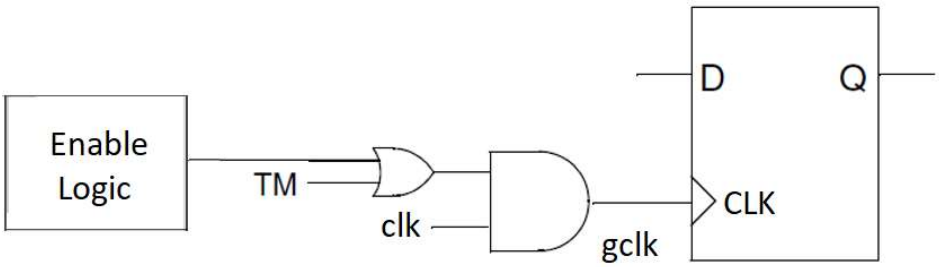

Figure 7: Circuit with fixed clock gate violation

DFT Configuration and scan structure setup

The details of the scan structure, such as, the number of scan chains, associated test ports, nature of chains and their scan chain lengths are specified [4].

1. To set the minimum number of scan chains:

   set_attribute dft_min_number_of_scan_chains integer top_design

2. To set the maximum number of scan chains:

   set_attribute dft_max_length_of_scan_chains integer top_design

3. If the posedge and negedge flops can be mixed in the same scan chains:

set_attribute dft_mix_clock_edges_in_scan_chains {true | false} top_design

4. On defining the scan chains, the DFT engine will connect the scan flops to the specific test in and out ports. If the number of scan chains defined is lower than the global minimum number of chains, then the ports are auto-cretaed to accommodate the other chains.

   define_dft scan_chain [-name name]-sdi s_in][-sdo s_out][-create_ports]

5. The DFT setup and the scan chains can be reported as below:

   report dft_chains

   report dft_setup

6. Once the scan configuration is complete, the scan chains can be connected, and the final netlist can be dumped with write hdl command.

   connect_scan_chains [-auto_create_chains]

LOGIC EQUIVALENCE CHECKING

The netlist of the DFT inserted SoC with amber core serves as the revised design and the RTL as the golden model. Conformal has two modes of operation SETUP and LEC, meant for environment setup and running equivalence check respectively [5]. The flow is elaborate below.

Setup phase

The libraries used for the synthesis of the design, the golden design (RTL) and the revised design (Netlist) are read.

SETUP> read library -both -liberty $lib_files

SETUP> read design -verilog ... -golden

SETUP> read design -verilog ... -revised

After the libraries and the designs are read, we need to write the design constraints to exclude macro blocks such as RAM, analog modules etc, and constrain input ports or internal nets [6].

1. Black boxing a macro is common, and though the internal logic is not analyzed the connections to the black box are still verified. This step is crucial because input pins of the black box are compare points, but the outputs of the black box are fan-ins to the next logic cones.
   - Command to black box before reading design:
     SETUP> ADD NOTRANSLATE MODULE
   - Command to black box unresolved modules with no RTL or library definition:
     SETUP> SET UNDEFINED CELL -black_box
   - Command to black box after reading the design, useful in hierarchical comparison:
     SETUP> ADD BLACK BOX
2. Pin Constraints: The test signals added during DFT insertion need to be constrained to make the RTL and netlist equivalent, since the RTL will not have scan logic or synthesis inserted clock gates as shown in Figure 8.
   SETUP> add pin constraints 0 SCAN_EN -revised

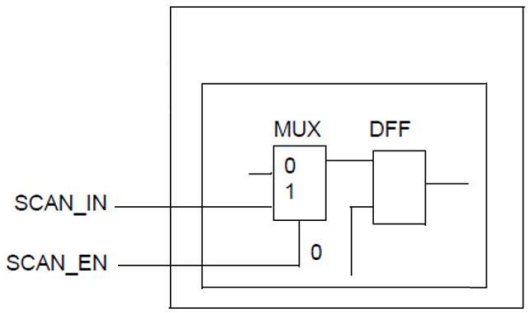

Figure 8: Pin constraints

The synthesis tool performs logic optimization by constant propagation. The command *SET FLATTEN MODEL* specifies conditions for circuit flattening.

1. For instance, to convert a flop or latch (D-pin is set to 0/1) to 0/1 use

   SETUP> set flatten model -seq_constant

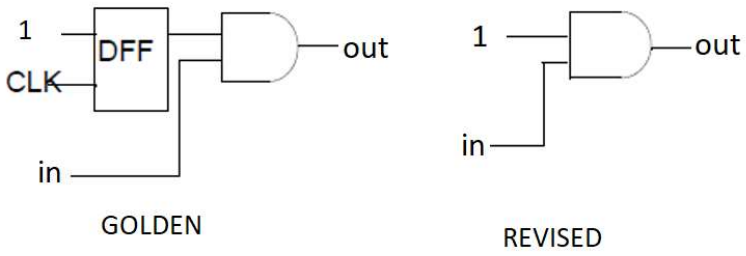

Figure 9: Flattening model for constant propagation

2. Also, the uninitialized flops such as those which feed themselves are assumed ‘x’; we need to set the flatting model to take care of such cases.

   SETUP> set flatten model -seq_constant_x_to 0

3. The clock gating introduced by the synthesis tool for dynamic power optimization may cause problems while performing equivalence check. This can be resolved by setting appropriate analysis options which remodel the revised model clock gates to golden model flop data pin muxing logic, as shown in the Figure 10.

   SETUP> set flatten model -gated clock

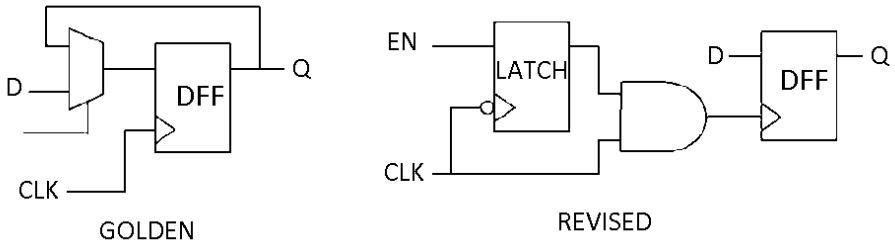

Figure 10: Flattening model for clock gate analysis

LEC phase

After the SETUP phase is complete, LEC Mode is entered with the command *'SET SYSTEM MODE LEC'*, upon which the key points are auto-mapped with name-first default mapping method in which signals with same names are mapped in the golden and revised designs. Conformal can compare all mapped points [5, 6]. The comparison indicates if any point is non-equivalent. Compared points are primary outputs, flip-flops, latches and black boxes. The compare points can also be manually mapped and then compared to obtain the results.

LEC > add mapped points

LEC > add compare points -all

LEC > compare

RESULTS

The DFT insertion is performed during the synthesis of the SoC with amber core in RC [9]. The Synopsys Design Constraints (SDC) [7, 8] and the Tcl synthesis script are given in the Appendix A and B.

The design has four mixed clock edge, muxed-style scan chains of length 685 with four test input and output ports, *'test_si[3:0]'* and *'test_so[3:0]'* respectively. The test-mode pin *'scan_mode'* and scan clock gate enable pin *'scan_cg_en'* help fix the DFT rule violations. The scan chains have one scan clock *'scan_clk'* and a scan shift enable *'scan_enable'* to support struck at fault testing.

The logic equivalence check is performed in Cadence Conformal LEC with the script in the Appendix C. The golden and revised models are proven to be equivalent by choosing appropriate black boxing, pin constraints and analysis options. Figure 11 shows the results of LEC.

```
=============================================================================
Compared points      PO     DFF    BBOX      Total
-----------------------------------------------------------------------------
Equivalent           59     5759   17        5835
=============================================================================
Compare results of instance/output/pin equivalences and/or sequential merge
=============================================================================
Compared points      DFF       Total
-----------------------------------------------------------------------------
Equivalent           75        75
=============================================================================
Num of compare points = 5835
Num of diff points    = 0
Num of abort points   = 0
Num of unknown points = 0
```

Figure 11: Formal equivalence check result

Chapter 3: Formal Verification of configurable arbiter with APB slave

INTRODUCTION

Formal verification is the collection of techniques which use static analysis based on mathematical transformations to check the correctness of hardware, as opposed to dynamic verification such as simulation. It mathematically proves the correctness of a design with respect to a mathematical formal specification.

Many problems can be attacked using decision methods with limited human intervention, such as Boolean equivalence checking, temporal logic model checking and Symbolic trajectory evaluation [10]. This probably accounts for the relative success of formal verification in hardware.

Formal verification is potentially very fast because it does not have to evaluate every possible state to demonstrate that a given piece of logic meets a set of properties under all conditions. In this section, the RTL for the configurable arbiter with an APB slave is developed, and is verified formally by writing assumptions, assertions and cover properties for a design under test. The formal verification tool to be used is Cadence Jaspergold.

DUT DESIGN DESCRIPTION

The Design Under Test (DUT) is a 4-way arbiter with an APB slave interface. As depicted in the architecture section, the module arbiter_top instantiates apb_slave and arbiter. The APB slave interface provides registers for debug and configuration of the arbiter which implements different arbitration schemes.

APB slave interface

The APB is part of the AMBA 3 protocol family [11]. It provides a low-cost interface that is optimized for minimal power consumption and reduced interface complexity. All signal transitions are only related to the rising edge of the clock to enable the integration of APB peripherals easily into any design flow.

Write transfer

Figure 12 shows a basic write transfer with no wait states [11]. The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock. The first clock cycle of the transfer is called the Setup phase. After the following clock edge, the enable signal is asserted, PENABLE, and this indicates that the Access phase is taking place. The address, data and control signals all remain valid throughout the Access phase. The transfer completes at the end of this cycle. The enable signal, PENABLE, is deasserted at the end of the transfer. The select signal, PSEL, also goes LOW.

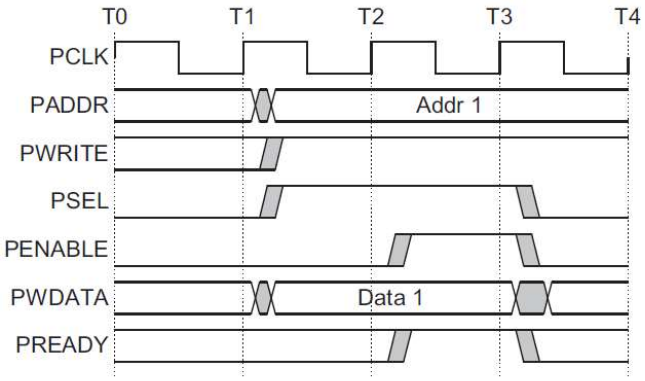

Figure 12: APB write transfer [11]

Read transfer

Figure 13 shows a read transfer. The timing of the address, write, select, and enable signals are as described in Write transfers [11]. The slave must provide the data before the end of the read transfer.

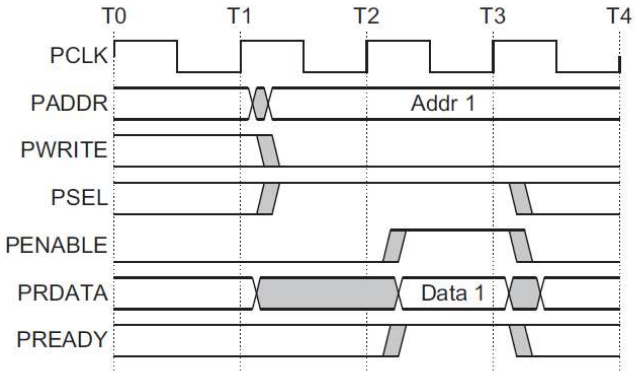

Figure 13: APB read transfer [11]

APB register description

The APB registers are presented as the output ports 'APB_BYPASS', 'APB_REQ[3:0]' and 'APB_ARB_TYPE[2:0]' of the top module 'arbiter_top'. Hence, the register write value (PWDATA) must be reflected at the corresponding output port after the write operation is complete. Also, after the read operation is complete, any register read value (PRDATA) must be same as the value on the corresponding port.

APB bypass register

Register Address: 8’h10

7	6	5	4	3	2	1	0
Reserved							APB_BYPASS

Bit	Field	Description
7:1	Reserved	-
0	APB_BYPASS	Selects which request is fed to the arbiter. When 0: REQ When 1: APB_REQ Reset value: 1’b0 Legal values: 1’b0 or 1’b1

Table 1: APB bypass register details

APB request register

Register Address: 8’h14

7	6	5	4	3	2	1	0
Reserved				APB_REQ			

Bit	Field	Description
7:4	Reserved	-
3:0	APB_REQ	When APB_BYPASS =1, APB_REQ is chosen as the request input to the arbiter. Reset value: 4’b0000 Legal values: Range 4’b0000 to 4’b1111

Table 2: APB request register details

APB arbitration type register

Register Address: 8’h1C

7	6	5	4	3	2	1	0
Reserved					APB_ARB_TYPE		

Bit	Field	Description
7:3	Reserved	-
2:0	APB_ARB_TYPE	Selects the type of arbitration scheme. 3’b000: Priority ‘P0’: req[0] > req[1] > req[2] > req[3] 3’b001: Priority ‘P1’: req[1] > req[0] > req[2] > req[3] 3’b010: Priority ‘P2’: req[2] > req[0] > req[1] > req[3] 3’b011: Priority ‘P3’: req[3] > req[0] > req[1] > req[2] 3’b100: Priority ‘Prr’: Round robin arbitration scheme 3’b101: Priority ‘Prand’: Random arbitration scheme 3’b110 and 3’b111: Invalid Reset value: 3’b100 Legal values: Range 3’b000 to 3’b111

Table 3: APB arbitration type register details

Arbiter

The arbiter module receives the requests and issues the grants in the next clock cycle. It has six arbitration schemes which can be configured by APB_ARB_TYPE as shown the section APB arbitration type register. Following is the detailed description.

1. Four fixed priority arbitration schemes

   P0: req[0] > req[1] > req[2] > req[3]

   P1: req[1] > req[0] > req[2] > req[3]

   P2: req[2] > req[0] > req[1] > req[3]

   P3: req[3] > req[0] > req[1] > req[2]

2. Round robin arbitration scheme: Prr

   The scheduling is round robin, where the grants are given in a round robin manner (0 – 1 – 2 – 3 – 0 ...) when there is a contention. The order of grants will always follow a round robin cycle and can skip ports in round robin order (only if they are not requesting) to grant a port which is requesting. Consider a case where in cycle i, Port 1 was granted. Then, in cycle i+1, Port 2 will be granted if Port 2 requests (independent of any other port requesting). However, if Port 2 is not requesting, then the arbiter will look at Port 3 (and grant it if it requests) and so on continue to Port 0 and then Port 1.

3. Random arbitration scheme: Prand

   The grant is issued on a random basis. This is achieved by prioritizing the requests depending on the current value of the state of the PN sequence generator. The PN sequence {S1,S2,S3} is shown in the Table 4.

Clock	S1	S2	S3
0	1	0	0
1	1	1	0
2	1	1	1
3	0	1	1
4	1	0	1
5	0	1	0
6	0	0	1
7	1	0	0
---	---	---	---

Table 4: Pseudorandom sequence

DUT PORT DESCRIPTION

Signal	Direction	Width	Description
PCLK	IN	1	Clock. The rising edge of PCLK times all transfers on the APB.
PRESETn	IN	1	Reset. The APB asynchronous reset signal is active LOW.
PADDR	IN	8	Address. This is the APB address bus.
PSEL	IN	1	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a PSEL signal for each slave.
PENABLE	IN	1	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	IN	1	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	IN	8	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is HIGH.
PREADY	OUT	1	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	OUT	8	Read Data. The selected slave drives this bus during read cycles when PWRITE is LOW.

Table 5 continued.			
APB_BYPASS	OUT	1	APB register output. Selects which request is fed to the arbiter. When 0: REQ When 1: APB_REQ
APB_REQ	OUT	4	APB register output. When APB_BYPASS =1, APB_REQ is chosen as the request input to the arbiter.
APB_ARB_TYPE	OUT	3	APB register output. Selects the type of arbitration scheme. 3'b000: P0: req[0] > req[1] > req[2] > req[3] 3'b001: P1: req[1] > req[0] > req[2] > req[3] 3'b010: P2: req[2] > req[0] > req[1] > req[3] 3'b011: P3: req[3] > req[0] > req[1] > req[2] 3'b100: Prr: Round robin arbitration scheme 3'b101: Prand: Random arbitration scheme 3'b110 and 3'b111: Invalid
REQ	IN	4	Request port. When APB_BYPASS =0, REQ is chosen as the request input to the arbiter.
GNT	OUT	4	Grant port.

Table 5: Arbiter DUT port description

DUT ARCHITECTURE

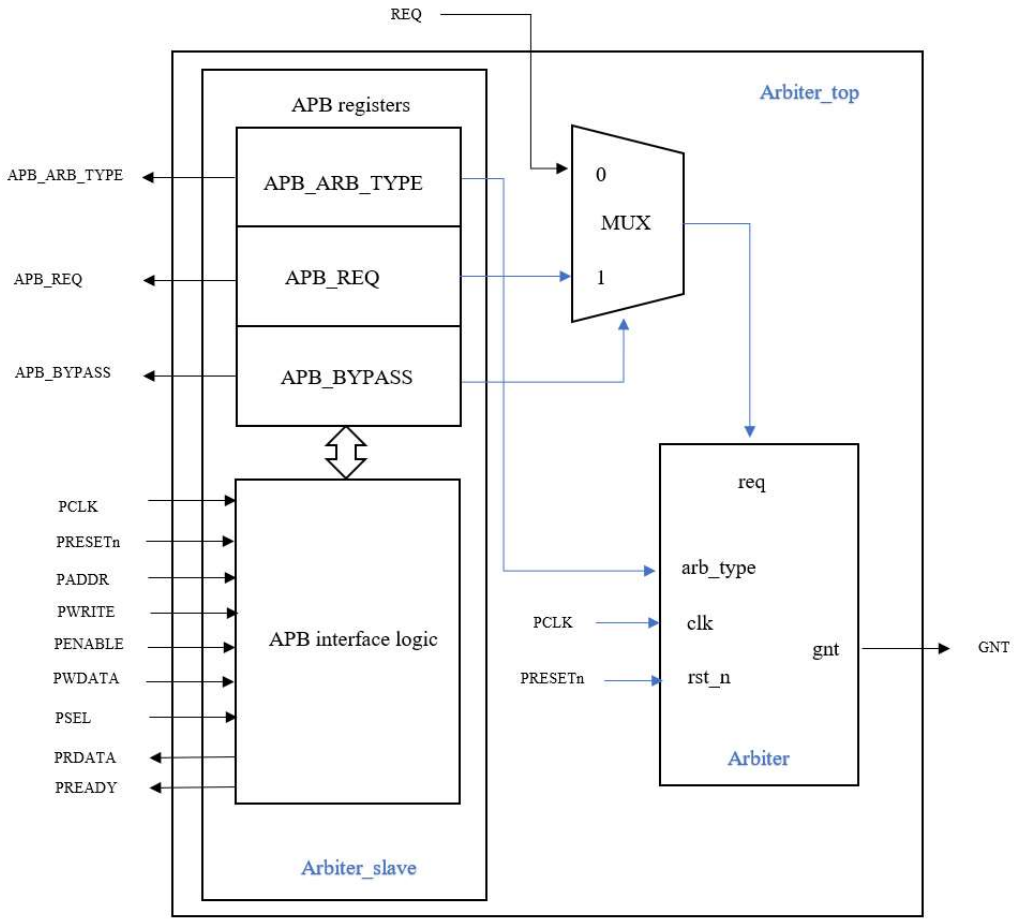

Figure 14: Arbiter DUT microarchitecture

PROPERTIES

The DUT is verified with set of assertions, assumptions and coverage properties [12, 13, 14]. Assertions will be written as properties to the check for correctness of the system behavior. An "Assume" statement specifies a property as an assumption for the verification environment. To ensure functional coverage, certain cover properties must be written for the design.

APB interface properties

Assumptions

1. APB read/write are single transfers and are padded with IDLE phase, that is, once initiated they are always completed with the following sequence only: IDLE (PSEL =0 & PENABLE =0) => PHASE1 (PSEL=1 & PENABLE =0) => PHASE2 (PSEL=1 & PENABLE =1) => IDLE (PSEL =0 & PENABLE =0).
2. PADDR, PWDATA and PWRITE are stable and defined during the transfers.
3. PADDR must take only the legal address values given in the APB register description.
4. For a given PADDR, PWDATA can only take legal write values as given in the APB register description. For instance, if PADDR = 8'h10, PWDATA can only be 8'h00 or 8'h01.

Assertions

1. Check if APB write operation is correct for all registers.
2. Check if APB read operation is correct for all registers.
3. Check if reset values of the registers are correct.

Coverage

1. APB read operation happens at least once.
2. APB write operation happens at least once.

Arbiter properties

Assumptions

Input requests on any port should be held high until they are granted. The arbiter does not keep a history of requests. For correct operation, a port should make a request and then keep it high till it has been granted.

Assertions

1. All grants are mutually exclusive, and a grant is not issued unless the request is asserted. These are safety properties to ensure that no two grants are given in the same cycle.
2. Check for priority order for scheme P0, P1, P2 and P3.
3. For priority schemes Prr and Prand, check for liveness properties to ensure that no port is starved for a grant. That is, for these arbitration schemes, every request should be granted within 5 and 8 clock cycles respectively.

Coverage

1. Each request to go high at least once.
2. All schemes are covered.

RESULTS

The RTL for Arbiter with APB slave is developed in Verilog HDL. The DUT is formally verified by writing SystemVerilog properties from the previous section. The design properties listed in Appendix D. For the specified properties, the results of the verification prove the correctness of the design as shown in the Table 6 and Figure 15. The

formal verification has been performed with Cadence Jaspergold [15] with the help of its visualization tools [16].

Property name	Result	Bound
arbiter_top.u_arbiter.u_arb_props.a_gnt_onehot	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_0_gnt0	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_0_gnt1	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_0_gnt2	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_0_gnt3	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_1_gnt0	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_1_gnt1	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_1_gnt2	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_1_gnt3	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_2_gnt0	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_2_gnt1	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_2_gnt2	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_2_gnt3	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_3_gnt0	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_3_gnt1	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_3_gnt2	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.assert_pri_3_gnt3	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[0].assert_gnt_within5_req_rr	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[0].assert_gnt_within5_req_rand	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[0].assert_no_req_no_gnt	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[0].cover_req	covered	1

Table 6 continued.		
arbiter_top.u_arbiter.u_arb_props.gen[0].cover_gnt	covered	2
arbiter_top.u_arbiter.u_arb_props.gen[1].assert_gnt_within5_req_rr	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[1].assert_gnt_within5_req_rand	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[1].assert_no_req_no_gnt	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[1].cover_req	covered	1
arbiter_top.u_arbiter.u_arb_props.gen[1].cover_gnt	covered	2
arbiter_top.u_arbiter.u_arb_props.gen[2].assert_gnt_within5_req_rr	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[2].assert_gnt_within5_req_rand	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[2].assert_no_req_no_gnt	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[2].cover_req	covered	1
arbiter_top.u_arbiter.u_arb_props.gen[2].cover_gnt	covered	2
arbiter_top.u_arbiter.u_arb_props.gen[3].assert_gnt_within5_req_rr	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[3].assert_gnt_within5_req_rand	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[3].assert_no_req_no_gnt	proven	Infinite
arbiter_top.u_arbiter.u_arb_props.gen[3].cover_req	covered	1
arbiter_top.u_arbiter.u_arb_props.gen[3].cover_gnt	covered	2
arbiter_top.u_apb_props.assert_APB_BYPASS_REG_VALID	proven	Infinite
arbiter_top.u_apb_props.assert_APB_REQ_REG_VALID	proven	Infinite
arbiter_top.u_apb_props.assert_APB_BYPASS_WR_chk	proven	Infinite
arbiter_top.u_apb_props.assert_APB_REQ_WR_chk	proven	Infinite
arbiter_top.u_apb_props.assert_APB_ARB_TYPE_WR_chk	proven	Infinite
arbiter_top.u_apb_props.assert_APB_BYPASS_RD_chk	proven	Infinite
arbiter_top.u_apb_props.assert_APB_REQ_RD_chk	proven	Infinite

Table 6 continued.		
arbiter_top.u_apb_props.assert_APB_ARB_TYPE_RD_chk	proven	Infinite
arbiter_top.u_apb_props.assert_BYPASS_reset_chk	proven	Infinite
arbiter_top.u_apb_props.assert_REQ_reset_chk	proven	Infinite
arbiter_top.u_apb_props.assert_ARB_TYPE_reset_chk	proven	Infinite
arbiter_top.u_apb_props.cover_APB_BYPASS	covered	3
arbiter_top.u_apb_props.cover_APB_WRITE	covered	2
arbiter_top.u_apb_props.cover_APB_READ	covered	2

Table 6: Formal Verification result for Arbiter with APB salve

```
================================================================
SUMMARY
================================================================

    Properties Considered : 102
        assertions      : 41
         - proven       : 41 (100%)
         - marked_proven: 0 (0%)
         - cex          : 0 (0%)
         - ar_cex       : 0 (0%)
         - undetermined : 0 (0%)
         - unprocessed  : 0 (0%)
         - error        : 0 (0%)
        covers          : 61
         - unreachable  : 0 (0%)
         - covered      : 61 (100%)
         - ar_covered   : 0 (0%)
         - undetermined : 0 (0%)
         - unprocessed  : 0 (0%)
         - error        : 0 (0%)
```

Figure 15: Formal verification result for Arbiter with APB slave

Chapter 4: Design and Verification of x86 Execution unit

INTRODUCTION

This chapter describes the design and verification of an implementation of an x86 execution stage. The last chapter discussed the formal verification of the arbiter with APB slave by writing SystemVerilog properties as per the design specification document. Hence, the verification is as exhaustive as the properties. In this chapter, we will formally verify the DUT by writing a reference behavioral model of the design similar to the scoreboard used in UVM and writing simple assertions to check for the equivalence between the model and the DUT outputs.

Contrary to Universal Verification methodology or UVM, formal verification does not require a verification environment [17]. To perform this verification task, UVM environment will need a sequencer, driver, monitor and scoreboard. Also, robust verification will need exhaustive coverage properties. In formal verification we can completely verify the DUT by simple assertions comparing its outputs with the reference model.

Such verification technique can be exploited for the DUTs such as execution units, decoders, floating point processors, DSP engines etc. for which it is simple to develop a scoreboard. For such DUTs, it is cumbersome to write a long list of properties for each operation mode required by conventional formal verification. Also, in our approach, we have additional advantage if the reference model and the DUT are designed by different designers or they are designed using different implementation styles (Structural DUT vs Behavioral scoreboard like in our case).

PROCESSOR OVERVIEW

The Intel Arhitecture, IA-32, is a CISC architecture [18]. Any task running on IA-32 has 32-bit address space. The following resources make up the basic execution environment for an IA-32 processor [20].

Stack

The stack is located in the memory to support procedure or subroutine calls and the passing of parameters between them. Also, stack management resources are included in the execution environment.

Basic program execution registers

The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment [19]. As shown in Figure 16, these registers can be grouped as follows.

General-purpose registers

The eight 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

1. EAX—Accumulator for operands and results data.
2. EBX—Pointer to data in the DS segment.
3. ECX—Counter for string and loop operations.
4. EDX—I/O pointer.
5. ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
6. EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.

7. ESP—Stack pointer (in the SS segment).

8. EBP—Pointer to data on the stack (in the SS segment).

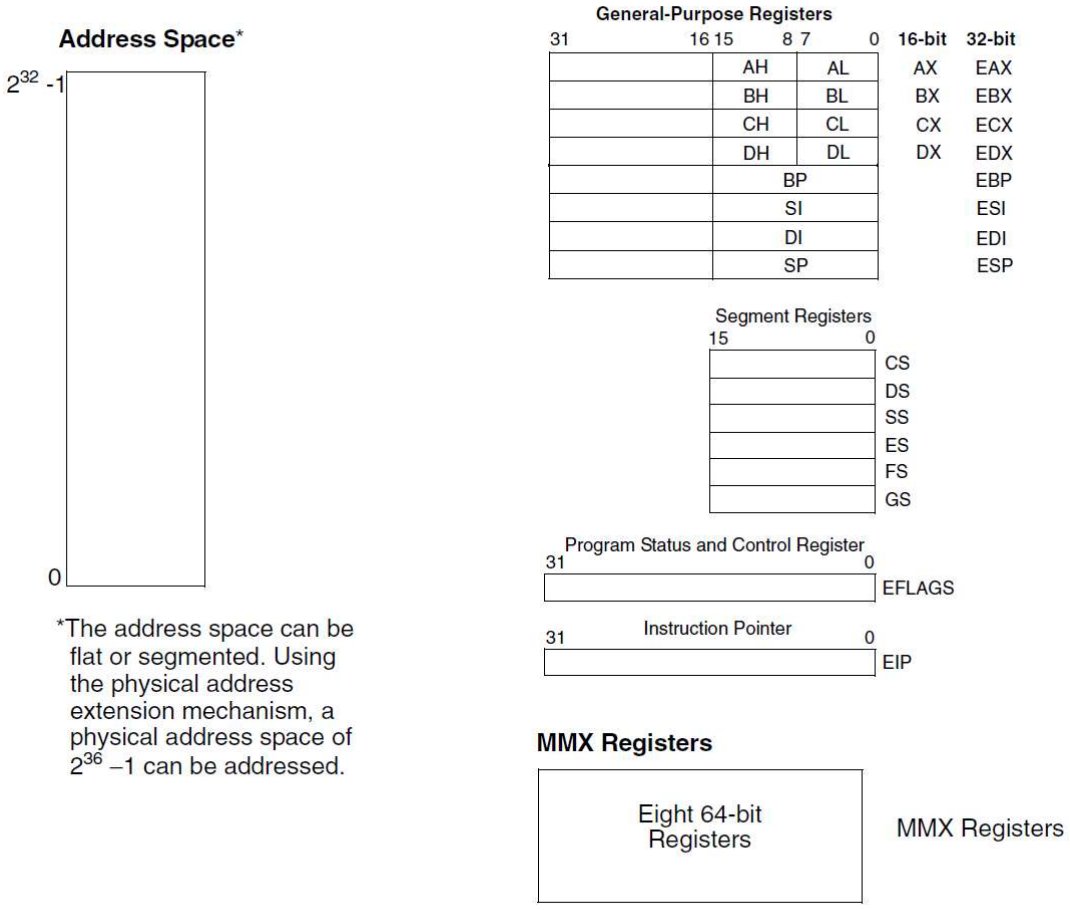

Figure 16: IA-32 programming model [18]

As shown in figure above, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. Each of the lower two bytes of the

EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

Segment registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

EFLAGS (program status and control) register

The EFLAGS register report on the status of the program being executed. The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, INC, DEC, CMP etc. instructions. The functions of the status flags are as follows [18].

1. CF (bit 0) Carry flag: Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
2. PF (bit 2) Parity flag: Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
3. AF (bit 4) Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary coded decimal (BCD) arithmetic.
4. ZF (bit 6) Zero flag: Set if the result is zero; cleared otherwise.

5. SF (bit 7) Sign flag: Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
6. OF (bit 11) Overflow flag: Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.
7. DF (bit 10) Direction flag: Controls the string instructions. Setting the DF flag causes the string instructions to auto-decrement. Clearing the DF flag causes the string instructions to auto-increment. The STD and CLD instructions set and clear the DF flag, respectively.

EIP (instruction pointer) register

The EIP register contains a 32-bit pointer to the next instruction to be executed. The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed.

MMX registers

The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers.

EXECUTION STAGE ARCHITECTURE

The x86 execution unit supports several operations listed in the Table 7. It is implemented in structural Verilog using accurate timing models provided by Cascade Design Automation Corporation. The execution stage functionality is split among three ALUs as shown in the Figures 17, 18 and 19. The first is dedicated for Single Instruction

Single Data (SISD) instructions such as ADD, AND, OR etc. The second performs supporting operations such as pointer increment or decrement for PUSH, POP etc. The last ALU is dedicated for Single Instruction Multiple Data (SIMD) instructions such as PADDD, PSHUFW etc. Structural Verilog is chosen as implementation style to have great degree of control over the critical path design. For instance, in the second ALU, the operation ESP_INC_IMM is in last stage of muxing as it adds 3 operands ESP, imm8 and data size while the operation ESP_INC or ESP_DEC occur in the early stage of muxing as they just add two operands ESP and data size.

The design of the ALUs is performance driven. The 2-operand adders used are conditional sum adders and the 3-operand adders used are Wallace tree adders as these are the fastest adders [21].

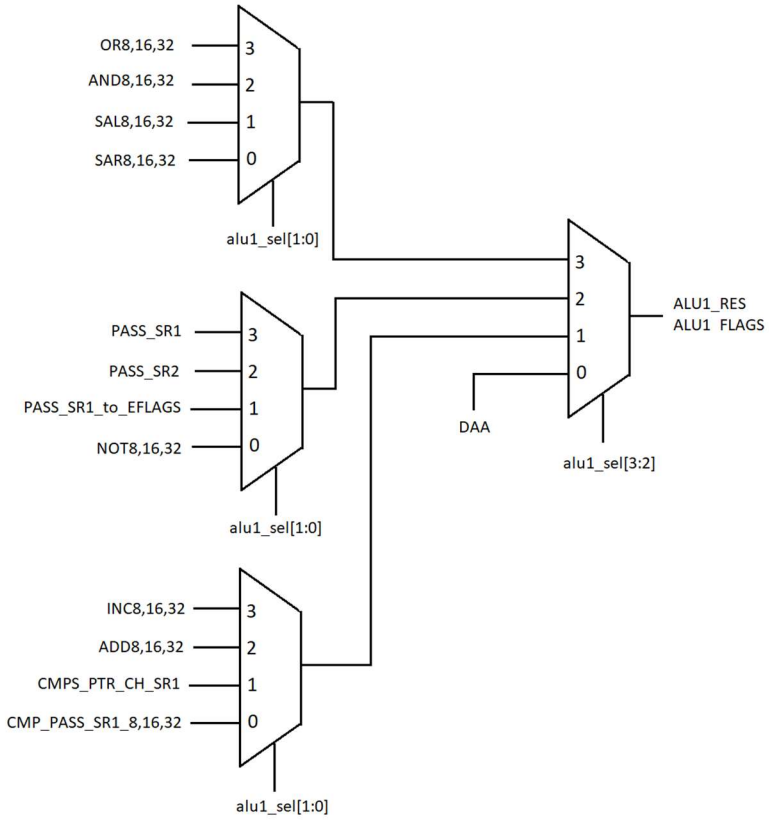

Figure 17: x86 Execute stage ALU 1 architecture

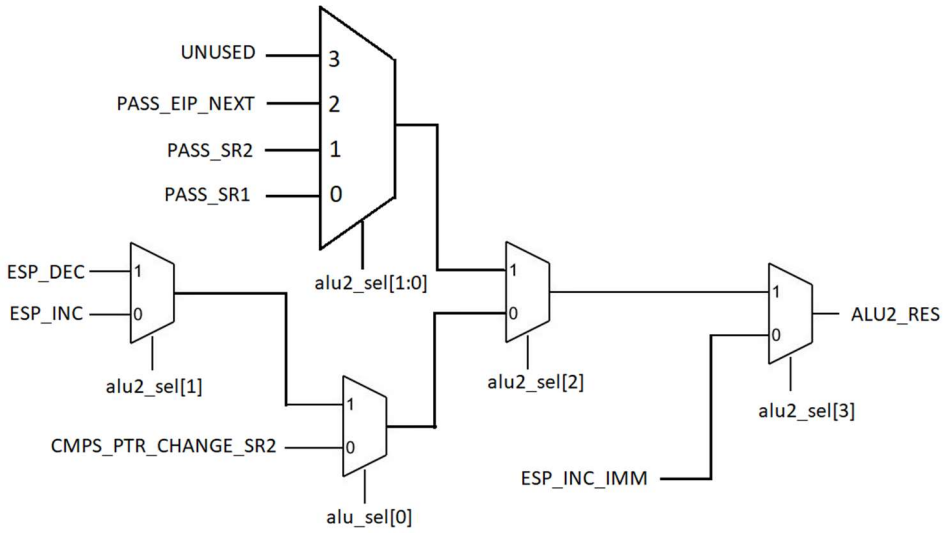

Figure 18: x86 Execute stage ALU 2 architecture

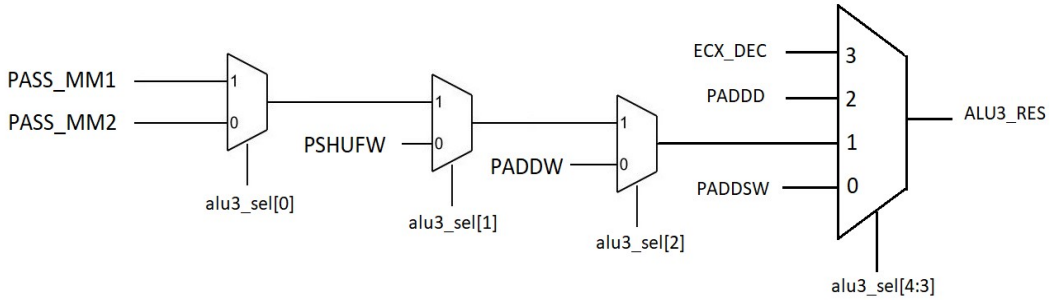

Figure 19: x86 Execute stage ALU 3 architecture

ALU opcode	Supported Instruction	Operation
ALU1		
OR8 OR16 OR32	OR AL,imm8 OR AX,imm16 OR EAX,imm32 OR r/m16,r16 OR r/m32,r32 OR r/m8,r8 OR r16,r/m16 OR r32,r/m32 OR r8,r/m8	RES ← SR1 OR SR2; FLAGS: The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.
AND8 AND16 AND32	AND AL,imm8 AND AX,imm16 AND EAX,imm32 AND r/m16,r16 AND r/m32,r32 AND r/m8,r8 AND r16,r/m16 AND r32,r/m32 AND r8,r/m8	RES ← SR1 AND SR2; FLAGS: The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

SAL8
SAL16
SAL32
	SAL r/m16,1
	SAL r/m32,1
	SAL r/m16,CL
	SAL r/m32,CL
	SAL r/m16,imm8
	SAL r/m32,imm8
	SAL r/m8,1
	SAL r/m8,CL
	SAL r/m8,imm8
		DEST ← SR1;
		COUNT ← SR2;
		tempCOUNT ← (COUNT AND 1FH);
		WHILE (tempCOUNT ≠ 0)
		DO
		CF ← MSB(DEST);
		DEST ← DEST ∗ 2;
		tempCOUNT ← tempCOUNT – 1;
		OD;
		
		FLAGS: (* Determine overflow for the various instructions *)
		IF (COUNT and 1FH) = 1
		OF ← MSB(DEST) XOR CF;
		ELSE IF (COUNT AND 1FH) = 0
		All flags remain unchanged;
		ELSE (* COUNT neither 1 or 0 *)
		OF ← undefined;
		FI;
		RES ← DEST;

Table 7 continued.		

SAR8
SAR16
SAR32
	SAR r/m16,1
	SAR r/m32,1
	SAR r/m16,CL
	SAR r/m32,CL
	SAR r/m16,imm8
	SAR r/m32,imm8
	SAR r/m8,1
	SAR r/m8,CL
	SAR r/m8,imm8
		DEST ← SR1;
		COUNT ← SR2;
		tempCOUNT ← (COUNT AND 1FH);
		WHILE (tempCOUNT ≠ 0)
		DO
		CF ← LSB(DEST);
		DEST ← DEST / 2 (*Signed divide, rounding toward negative infinity*);
		tempCOUNT ← tempCOUNT – 1;
		OD;
		FLAGS: (* Determine overflow for the various instructions *)
		IF (COUNT and 1FH) = 1
		THEN
		OF ← 0;
		FI;
		ELSE IF (COUNT AND 1FH) = 0
		THEN
		All flags remain unchanged;
		ELSE (* COUNT neither 1 or 0 *)
		OF ← undefined;
		FI;
		RES ← DEST;

PASS_SR1	MOV, POP, PUSH, XCHG, CMOVC, CALL, JMP	RES ← SR1
PASS_SR2	MOV, POP, PUSH, XCHG, CMOVC, CALL, JMP	RES ← SR2
PASS_SR1_TO_EFLAGS	IRET	FLAGS = SR1
NOT8 NOT16 NOT32	NOT r/m16 NOT r/m32 NOT r/m8	RES ← NOT SR1;
INC8 INC16 INC32	INC r/m16 INC r/m32 INC r/m8 INC r16 INC r32	RES ← SR1 + 1;

Table 7 continued.		

ADD8
ADD16
ADD32
	ADD AL,imm8
	ADD AX,imm16
	ADD EAX,imm32
	ADD r/m16,imm16
	ADD r/m32,imm32
	ADD r/m16,imm8
	ADD r/m32,imm8
	ADD r/m16,r16
	ADD r/m32,r32
	ADD r/m8,imm8
	ADD r/m8,r8
	ADD r16,r/m16
	ADD r32,r/m32
	ADD r8,r/m8
		Operation
		RES ← SR1 + SR2;
		
		FLAGS: The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

CMPS_PTR_CHANGE_SR1	CMPS m16,m16
	CMPS m32,m32
	CMPS m8,m8
		SR1 ← (E)SI
		IF (byte comparison)
		THEN IF DF = 0
		THEN
		(E)SI ← (E)SI + 1;
		ELSE
		(E)SI ← (E)SI – 1;
		FI;
		ELSE IF (word comparison)
		THEN IF DF = 0
		(E)SI ← (E)SI + 2;
		ELSE
		(E)SI ← (E)SI – 2;
		FI;
		ELSE (* doubleword comparison*)
		THEN IF DF = 0
		(E)SI ← (E)SI + 4;
		ELSE
		(E)SI ← (E)SI – 4;
		FI;
		FI;
		RES ← (E)SI

CMP_PASS_SR1_8 CMP_PASS_SR1_16 CMP_PASS_SR1_32	CMPXCHG r/m16,r16 CMPXCHG r/m32,r32 CMPXCHG r/m8,r8	Compare (EAX,AX or AL) with SR1 (r/m32, r/m16 or r/m8). Set FLAGS based on this compare. Pass SR1 to RES.

Table 7 continued.		

DAA	DAA	old_AL ← AL;
		old_CF ← CF;
		CF ← 0;
		IF (((AL AND 0FH) > 9) OR AF = 1)
		THEN
		AL ← AL + 6;
		CF ← old_CF OR (Carry from AL ← AL + 6);
		AF ← 1;
		ELSE
		AF ← 0;
		FI;
		IF ((old_AL > 99H) OR (old_CF = 1))
		THEN
		AL ← AL + 60H;
		CF ← 1;
		ELSE
		CF ← 0;
		FI;
		RES ←AL;

ALU2		
PASS_SR1	XCHG	RES ←SR1;
PASS_SR2	CMPXCHG	RES ←SR2;

CMPS_PTR_CHANGE_SR2		SR2 ← (E)DI
		IF (byte comparison)
		THEN IF DF = 0 THEN
		(E)DI ← (E)DI + 1;
		ELSE (E)DI ← (E)DI – 1; FI;
		ELSE IF (word comparison)
		THEN IF DF = 0
		(E)DI ← (E)DI + 2;
		ELSE
		(E)DI ← (E)DI – 2;
		FI;
		ELSE (* doubleword comparison*)
		THEN IF DF = 0
		(E)DI ← (E)DI + 4;
		ELSE
		(E)DI ← (E)DI – 4;
		FI;
		FI;
		RES ← (E)DI

Table 7 continued.		
ESP_DEC	CALL, PUSH	IF StackAddrSize = 32 THEN IF OperandSize = 32 THEN ESP ← ESP − 4; ELSE (* OperandSize = 16*) ESP ← ESP − 2; FI; ELSE (* StackAddrSize = 16*) IF OperandSize = 16 THEN SP ← SP − 2; ELSE (* OperandSize = 32*) SP ← SP − 4; FI; FI; RES ← (E)SP
ESP_INC	POP, RET	IF StackAddrSize = 32 THEN IF OperandSize = 32 THEN ESP ← ESP + 4; ELSE (* OperandSize = 16*) ESP ← ESP + 2; FI; ELSE (* StackAddrSize = 16*) IF OperandSize = 16 THEN SP ← SP + 2; ELSE (* OperandSize = 32 *) SP ← SP + 4; FI; FI; RES ← (E)SP
ESP_INC_IMM	RET imm16	THEN IF StackAddressSize=32 THEN ESP ← ESP + SR2; ELSE (* StackAddressSize=16 *) SP ← SP + SR2; FI; RES ← (E)SP

Table 7 continued.		
ALU3		
PADDD	PADDD mm, mm/m64	RES[31..0] ← mm1[31..0] + mm2[31..0]; RES[63..32] ← mm1[63..32] + mm2[63..32];
PADDSW	PADDSW mm, mm/m64	RES[15..0] ← SaturateToSignedWord(mm1[15..0] + mm2[15..0]); * repeat operation for 2nd and 7th words *; RES[63..48] ← SaturateToSignedWord(mm1[63..48] + mm2[63..48]);
PADDW	PADDW mm, mm/m64	RES[15..0] ← mm1[15..0] + mm2[15..0]; * repeat add operation for 2nd and 3th word *; RES[63..48] ← mm1[63..48] + mm2[63..48];
PSHUFW	PSHUFHW xmm1, xmm2/m128, imm8	Shuffle the words in mm2/m64 based on the encoding in imm8 and store the result in mm1. SRC ← mm1; ORDER ← SR2; RES[15-0] ← (SRC >> (ORDER[1-0] ∗ 16))[15-0] RES[31-16] ← (SRC >> (ORDER[3-2] ∗ 16))[15-0] RES[47-32] ← (SRC >> (ORDER[5-4] ∗ 16))[15-0] RES[63-48] ← (SRC >> (ORDER[7-6] ∗ 16))[15-0]
PASS_MM1	CALL, RET	RES ← mm1;
PASS_MM2	MOVQ	RES ← mm2;

ECX_DEC	CMPS m16,m16
	CMPS m32,m32
	CMPS m8,m8
		IF AddressSize = 16
		THEN
		use CX for CountReg;
		ELSE (* AddressSize = 32 *)
		use ECX for CountReg;
		FI;
		CountReg ← CountReg – 1;
		RES ← CountReg;

Table 7: x86 Execution stage functionality [19]

RESULTS

This verification methodology exploits the strengths of both UVM and formal verification. We are not restricted by the constraint that verification is as good as the set of properties we write in formal. Also, we need not worry about the coverage like in UVM, where it is not possible to cover all the test patterns via random test pattern generation. The reference model is developed for the x86 execution stage and equality check assertions are written to check the DUT outputs against that of reference model. The DUT is completely verified and the results are tabulated in Table 8. The formal verification has been performed with Cadence Jaspergold [15] with the help of its visualization tools [16].

Property name	Result	Bound
alu1_top.u_alu1_props.assert_alu_res1_chk	proven	Infinite
alu1_top.u_alu1_props.assert_alu1_flags_chk	proven	Infinite
alu1_top.u_alu1_props.assert_cmps_flags_chk	proven	Infinite
alu1_top.u_alu1_props.assert_df_val_ex_chk	proven	Infinite
alu1_top.u_alu1_props.assert_ld_flag_cf_chk	proven	Infinite
alu1_top.u_alu1_props.assert_ld_flag_pf_chk	proven	Infinite
alu1_top.u_alu1_props.assert_ld_flag_af_chk	proven	Infinite
alu1_top.u_alu1_props.assert_ld_flag_zf_chk	proven	Infinite
alu1_top.u_alu1_props.assert_ld_flag_sf_chk	proven	Infinite
alu2_top.u_alu2_props.assert_alu_res2_chk	proven	Infinite
alu3_top.u_alu3_props.assert_alu_res3_chk	proven	Infinite

Table 8: x86 execution unit formal verification result

The major bugs found are listed below. The design bugs are in corner cases which are hard to debug with simulation-based verification methods like UVM when the functionality is as diverse as the processor execution unit.

1. In opcode SAL8,16 & 32, the OF flags was assigned LSB of SR1 when the shift amount SR2 was zero or one. The bug is fixed to keep OF flag unchanged when SR2 is zero.
2. In DAA operation, the value of AF must go to zero after the end of the operation, but its value was being retained when AL[3:0] < 9.
3. In the case of stack instructions PUSH or POP, the ESP_DEC operation added 16'FFFF to ESP register instead of 32'FFFF_FFFF, which produced an incorrect value in the upper 16 bits of the ALU result.
4. AF must be the carry of the first nibble in add operation and borrow of the first nibble in case of a compare operation. The latter was setting AF flag incorrectly as a carry instead of a borrow.

Chapter 5: Identification of critical registers

INTRODUCTION

This chapter is in support of the thesis "Automatic generation of coverage directives targeting signal relationships by statically analyzing RTL" [22], which focuses on writing SystemVerilog cover properties by analyzing RTL written in Verilog HDL. The coverage problem has been an issue in simulation-based verification. The coverage properties are required to track the progress and justify completeness design verification. The approach [22] discusses statically analyzing the RTL and automatically generating coverage properties which target the ambiguity in signal relationships derived from the RTL, avoiding state-explosion and focus on the control flow of the design. These SystemVerilog properties can be integrated with any simulator to provide coverage goals. However, it can be argued that all possible properties may turn out to be a huge number, therefore this chapter discusses on how to identify the critical flops in the design as a subset of all the registers which may have the most effect on the control flow of a module.

In the design, certain flops store the history of the module, that is, their value depends directly on their previous value where as for other flops, their value is freshly computed each clock cycle. We can refer the latter as data registers and the former as state registers. It is possible that data registers can depend on history of the module, but the relation indirect and is through the state registers. The criticality 'C' will be the value associated with each register in the design based on several criteria discussed further. The value 'C' of the state registers outweighs that of data registers, that is, state registers will be associated with greater 'C' value.

Also, identifying critical design flops has applications in fault tolerance. Alpha particle-induced soft errors, or simply soft errors, refer to transient errors in device caused

by alpha particles emitted by traces of radioactive elements such as thorium and uranium present in the packaging materials of the device [23]. These alpha particles manage to penetrate the die and generate a high density of holes and electrons in its substrate, which creates an imbalance in the device's electrical potential distribution that causes stored data to be corrupted in flops, memory devices etc. We can make the top X% of the critical flops fault tolerant with the fault tolerant flip-flop design [24].

METHODOLOGY

To find the criticality of the flops, several designs are considered. First feature extraction is performed and then a simple Artificial Neural Network (ANN) is employed to trained and test on the data set.

Feature extraction

Feature extraction is the process of selecting a subset of relevant features for use in model construction [27]. It aids the mission to create an accurate predictive model choosing features that will provide as good or better accuracy whilst requiring less data. Cadence RTL compiler (RC) is used to extract the timing paths in the design. The tool can perform generic synthesis and provide the input to register, register to register and register to output timing paths. The script that extracts the timing paths is given in Appendix E.

The timing paths are then processed to obtain 6 features listed below for each register ‘r’ in the design.

1. Register length Len(r)
2. Number of inputs ports directly effecting the register #In(r)
3. Number of output ports directly effected by the register #Out(r)
4. Number of other registers effecting the register ‘r’ #Rin(r)

5. Number of other registers effected by register ‘r’ Rout(r)

6. If self-loop exists Loop(r).

The results of the feature extraction for MMU is shown in the Table 9 and the state registers in the design are labelled.

Register ‘r’	Len(r)	#In(r)	#Out(r)	#Rin(r)	#Rout(r)	Loop(r)	Label
cache_line_rd_buff	256	2	2	1	0	0	0
u_count_reg	3	2	1	1	3	1	1
u_dc_evict_addr_reg	32	3	0	2	1	1	0
u_dc_evict_data_buff	128	3	1	2	0	1	0
u_dc_evict_flag_reg	1	2	0	2	2	1	1
u_dc_evict_gated_reg	1	2	0	1	3	0	0
u_io_m_data_i_reg	32	2	1	1	0	1	0
u_mmu_fsm/curr_st_reg	3	7	8	2	7	1	1
u_temp_addr_reg	32	7	1	4	0	1	0

Table 9: Feature extraction of x86 MMU module

ANN training and testing

An Artificial Neural Network is a network of simple elements called neurons, which receive input, change their internal state (activation) according to that input, and produce output depending on the input and activation [28]. The network forms by connecting the output of certain neurons to the input of other neurons forming a weighted directed graph. The weights as well as the functions that compute the activation can be

modified by a process called learning which is governed by a learning rule which is the back propagation algorithm [25].

The ANN has 6 input neurons corresponding to 6 features extracted, 25 hidden neurons for learning the features and 1 output neuron for evaluating criticality ‘C’ as shown in the Figure 20. The ANN is trained with design set 1 through 6 and tested with designs set 7 through 11 from the Table 10.

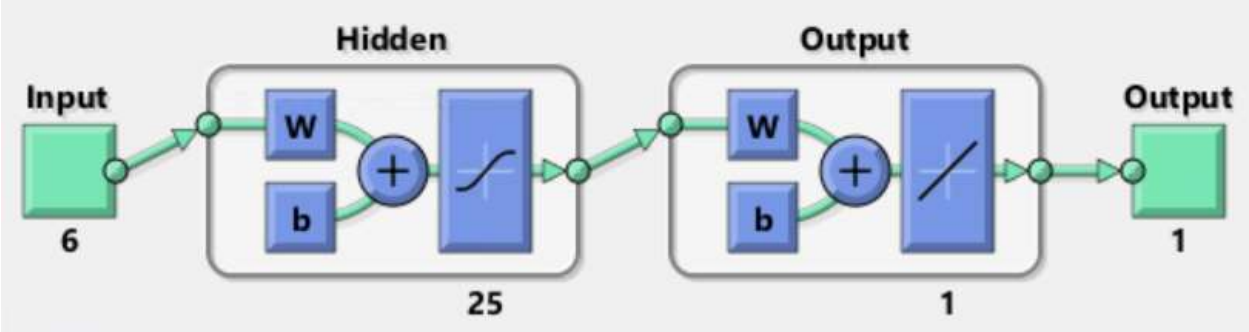

Figure 20: ANN structure (Input-Hidden-Output layers)

Sl.no.	Design	Flops
1	Amber core write back stage	44
2	Synchronous serial protocol	134
3	Sequence detectors	9
4	APB to wishbone bridge	84
5	I2C protocol	159
6	JTAG TAP interface	70
7	Arbiter with APB slave config port	32
8	X86 Memory Management Unit(MMU)	488
9	X86 Direct Memory Access unit(DMA)	213
10	Interrupt and exception handler	85
11	X86 SoC Wishbone arbiter	3

Table 10: ANN training and testing design data set

RESULTS

The detection of critical registers in a design finds application in automatic coverage generation and fault tolerance. Tables 11 and 12 show the results of testing on MMU and DMA modules, respectively. It can be observed that current state register of MMU and DMA FSMs have highest criticality ‘C’ of 1.65 and 1.27 respectively. The short

registers that self-loop and that effect higher number of outputs (#Out) and feed to higher number of flops (#Rout) have greater ‘C’.

The accuracy of training and testing is approximately 99%. The rare incorrect classification is due the short data registers that self-loop, but the ‘C’ value of such registers is lower than the state registers. However, by varying the classification threshold we can eliminate such cases.

Register ‘r’	Len	#In	#Out	#Rin	#Rout	Loop	C
u_mmu_fsm/curr_st_reg	3	7	8	2	7	1	1.6487
u_dc_evict_flag_reg	1	2	0	2	2	1	1.028921
u_count_reg	3	2	1	1	3	1	0.965714
u_dc_evict_data_buff	128	3	1	2	0	1	0.715656
cache_line_rd_buff	256	2	2	1	0	0	0.346443
u_dc_evict_addr_reg	32	3	0	2	1	1	0.079015
u_io_m_data_i_reg	32	2	1	1	0	1	0.066598
u_dc_evict_gated_reg	1	2	0	1	3	0	0.011166
u_temp_addr_reg	32	7	1	4	0	1	-0.13233

Table 11: Criticality of MMU registers

Register ‘r’	Len	#In	#Out	#Rin	#Rout	Loop	C
u_dma_master_if/ u_dma_master_controller/ u_dma_master_controller_fsm/ curr_st_reg	3	4	4	4	3	1	1.267571
u_dma_master_if/ end_offset_reg	2	1	1	2	0	1	0.912753
u_dma_master_if/ transfer_size	2	1	1	2	0	1	0.912753
u_dma_master_if/ start_offset_reg	2	1	1	3	0	1	0.884843
u_dma_slave_if/ init_transfer_reg	32	5	1	0	7	1	0.16873
u_dma_master_if/ u_dma_master_controller/ m_addr	32	2	1	3	0	1	0.049332
u_dma_slave_if/ mem_addr_reg	32	5	1	0	2	1	0.027637
u_dma_master_if/ init_trans_reg	1	1	0	1	6	0	0.022498
u_dma_slave_if/disk_addr_reg	32	5	1	0	0	1	-0.02342
u_dma_slave_if/ transfer_size_reg	32	5	1	0	0	1	-0.02342

Table 12 contined.							
u_dma_master_if/ u_dma_master_controller/ init_trans_reg	1	1	1	2	3	0	-0.05561
u_dma_master_if/ u_dma_master_controller/ d_addr	10	2	1	2	0	1	-0.09613
u_dma_master_if/ u_dma_master_controller/ num_transfers	16	2	0	3	1	1	-0.19043
u_dma_master_if/ num_transfers_reg	16	1	1	2	2	1	-0.38327

Table 12: Criticality of DMA registers

Chapter 6: Conclusions

This report explores different verification methodologies. In the second chapter, synthesis and DFT insertion was performed on an SoC with the ARM-Amber core using Cadence RC and the DFT violations were fixed. To ensure that the original functionality is intact, formal equivalence check was performed between the RTL and the netlist. For future work, the ATPG tool can be used to perform scan simulations using the generated test patterns.

In the third chapter, the RTL for an Arbiter with an APB slave configuration port was developed in Verilog HDL and the design was extensively verified by writing SystemVerilog properties using Cadence Jaspergold. We observed that the verification is only as good as the properties specified. The fourth chapter discussed formal verification with a different approach. The HDL for a performance optimized x86 execution unit was developed and formally verified by developing its reference model in behavioral Verilog and writing simple equality assertion checks. This approach reduced the long list of properties required for the conventional formal methods discussed in the third chapter.

The last chapter provided an approach to identify the critical registers in a design. Feature extraction was performed, and a simple ANN was used for computing the criticality ‘C’. The accuracy of training and testing is approximately 99%. The rare incorrect classification is due to the short data registers that self-loop, but the ‘C’ value of such registers is lower than that of the state registers. This finds application in selecting the relevant auto-generated properties and in fault tolerance.

Appendices

Appendix A: Synopsys design constraints for SoC with amber core

```
#  clock creation
create_clock -period 80 -waveform "0 40" [get_ports brd_clk_p] -name BRD_CLK_P
create_clock -period 400 -waveform "0 200" [get_ports mtx_clk_pad_i] -name mtx_clk
create_clock -period 400 -waveform "0 200" [get_ports mrx_clk_pad_i] -name mrx_clk

# generated clocks
create_generated_clock -name sys_clk -source [get_ports brd_clk_p] -divide_by 1
[get_pins u_clocks_resets/sys_clk_buff/Y]
create_generated_clock -name sys_clk_slow -source [get_ports brd_clk_p] -divide_by 4
[get_pins u_clocks_resets/sys_clk_slow_buff/Y]

# clock uncertainty
set_clock_uncertainty -setup 0.03 [get_clocks sys_clk]
set_clock_uncertainty -hold 0.03 [get_clocks sys_clk]
set_clock_uncertainty -setup 0.1 [get_clocks mtx_clk]
set_clock_uncertainty -hold 0.1 [get_clocks mtx_clk]
set_clock_uncertainty -setup 0.1 [get_clocks mrx_clk]
set_clock_uncertainty -hold 0.1 [get_clocks mrx_clk]

# input & output delay
set in_ports [remove_from_collection [all_inputs] [get_ports *_clk_*]]

set_input_delay -max 0.1 -clock [get_clocks sys_clk] $in_ports
set_input_delay -min 0.1 -clock [get_clocks sys_clk] $in_ports
set_input_delay -max 0.1 -clock [get_clocks mtx_clk] $in_ports
set_input_delay -min 0.1 -clock [get_clocks mtx_clk] $in_ports
set_input_delay -max 0.1 -clock [get_clocks mrx_clk] $in_ports
set_input_delay -min 0.1 -clock [get_clocks mrx_clk] $in_ports

set_output_delay -max 0.1 -clock sys_clk [all_outputs]
set_output_delay -max 0.1 -clock mtx_clk [all_outputs]
set_output_delay -max 0.1 -clock mrx_clk [all_outputs]

# flase path
set_false_path -from [get_ports brd_rst]

# set input transition
set_input_transition 0.02 [all_inputs]
```

```
# output load
set_load 1.5 [all_outputs]

# case analysis statements
set_case_analysis 0 scan_mode
set_case_analysis 0 scan_cg_en
set_case_analysis 0 scan_en
```

Appendix B: Script for synthesis with DFT insertion

```
set current_design system
set WDIR .
set TOP ${WDIR}/..
source  ${TOP}/common/common.tcl

#-------------------------------------------------------------------------------------------------------
# LIBRARY SETUP
#-------------------------------------------------------------------------------------------------------

set corner ss0p95vn40c
set lib_dir $DESIGN_REF_LIB_PATH
set std_library [list saed32hvt_${corner}.lib saed32rvt_${corner}.lib
saed32lvt_${corner}.lib saed32sram_${corner}.lib]

set lib_path [list $lib_dir/stdcell_hvt/db_nldm \
            $lib_dir/stdcell_rvt/db_nldm \
            $lib_dir/stdcell_lvt/db_nldm \
            $lib_dir/sram/db_nldm]

set_attribute lib_search_path $lib_path
set_attribute library $std_library
#-------------------------------------------------------------------------------------------------------
# RTL SETUP
#-------------------------------------------------------------------------------------------------------

set verilog_path    "../../verilog"
set rtl_search_path [list ${verilog_path} \
                ${verilog_path}/system \
                ${verilog_path}/tb \
                ${verilog_path}/lib \
                ${verilog_path}/ethmac \
                ${verilog_path}/amber25]

set_attribute hdl_search_path $rtl_search_path
source ${TOP}/common/rtl.list
set myFiles $RTL_LIST
read_hdl ${myFiles}
elaborate ${current_design}
read_sdc ${TOP}/common/constraints.sdc
```

```
check_design -unresolved
report timing -lint

set_attribute dft_scan_style muxed_scan
define_dft test_mode -active high scan_mode
define_dft test_clock scan_clk

define_dft shift_enable -active high scan_en
define_dft test_mode -active high scan_cg_en
set_attribute lp_insert_clock_gating true /

report dft_setup
check_dft_rules

# Synthesize the design to the generic library
synthesize -to_generic

# Synthesize the design to the target library
synthesize -to_mapped

report dft_setup
check_dft_rules

fix_dft_violations -test_control scan_cg_en -clock

set_attr dft_min_number_of_scan_chains 4 /designs/${current_design}
set_attr dft_mix_clock_edges_in_scan_chains true /designs/${current_design}

define_dft scan_chain -name chain1 -create_ports -sdi test_si[0] -sdo test_so[0]
define_dft scan_chain -name chain2 -create_ports -sdi test_si[1] -sdo test_so[1]
define_dft scan_chain -name chain3 -create_ports -sdi test_si[2] -sdo test_so[2]
define_dft scan_chain -name chain4 -create_ports -sdi test_si[3] -sdo test_so[3]

connect_scan_chains -auto_create_chains -preview
connect_scan_chains -auto_create_chains

write_hdl -m > ${current_design}_cg_scan_netlist.v
write_scandef > ${current_design}_cg_scandef.v
report dft_setup > ${current_design}_dft_setup
```

Appendix C: Script for Logic equivalence checking

```
set_log_file lec_cg_scan.log -replace
set_undefined_cell black_box -noascend -both

set WDIR [pwd]
set TOP ${WDIR}/..

read_library -Both -Replace  -sensitive  -Statetable  -Liberty   \
/usr/local/packages/synopsys_2015/SAED32_EDK/lib/stdcell_hvt/db_nldm/saed32hvt_ss
0p95vn40c.lib \
/usr/local/packages/synopsys_2015/SAED32_EDK/lib/stdcell_rvt/db_nldm/saed32rvt_ss
0p95vn40c.lib \
/usr/local/packages/synopsys_2015/SAED32_EDK/lib/stdcell_lvt/db_nldm/saed32lvt_ss
0p95vn40c.lib

set verilog_path    "../../verilog"
set rtl_search_path [list ${verilog_path} \
                ${verilog_path}/system \
                ${verilog_path}/tb \
                ${verilog_path}/lib \
                ${verilog_path}/ethmac \
                ${verilog_path}/amber25]

add_search_path $rtl_search_path -design -golden

source ${TOP}/common/rtl.list
set my_verilog_files $RTL_LIST

read_design $my_verilog_files -Verilog -Golden   -sensitive   -root system -
continuousassignment Bidirectional   -nokeep_unreach   -nosupply

read_design ./system_cg_scan_netlist.v -Verilog -Revised   -sensitive   -root system -
continuousassignment Bidirectional   -nokeep_unreach   -nosupply

vpxmode
set flatten model -seq_constant -seq_constant_x_to 0
set flatten model -gated_clock
tclmode

add_pin_constraint 0 scan_cg_en -Revised
add_pin_constraint 0 scan_en -Revised
add_pin_constraint 0 scan_mode -both
```

```
set_analyze_option -auto

set_system_mode lec

add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram1
u_boot_mem_wrapper_u_boot_mem/myram1 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram2
u_boot_mem_wrapper_u_boot_mem/myram2 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram3
u_boot_mem_wrapper_u_boot_mem/myram3 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram4
u_boot_mem_wrapper_u_boot_mem/myram4 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram5
u_boot_mem_wrapper_u_boot_mem/myram5 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram6
u_boot_mem_wrapper_u_boot_mem/myram6 -noinvert
add_mapped_points u_boot_mem_wrapper/u_boot_mem/myram7
u_boot_mem_wrapper_u_boot_mem/myram7 -noinvert

add_compared_points -all
compare

puts "No of compare points = [get_compare_points -count]"
puts "No of diff points    = [get_compare_points -NONequivalent -count]"
puts "No of abort points   = [get_compare_points -abort -count]"
puts "No of unknown points = [get_compare_points -unknown -count]"
```

Appendix D: Formal verification properties of Arbiter with APB slave

```
//Formal Property Verification
//
//Modules - apb_props, arb_props and Wrapper
//SystemVerilog Properties for the module - arbiter_top

module apb_props(
// APB interface
input       PCLK,
input       PRESETn,
input       PWRITE,
input       PSEL,
input       PENABLE,
input  [7:0] PADDR,
input  [7:0] PWDATA,

input  [7:0] PRDATA,
input       PREADY,
// APB registers
input       APB_BYPASS,
input  [3:0] APB_REQ,
input  [2:0] APB_ARB_TYPE,
// Arbiter ports
input  [3:0] REQ,
input  [3:0] GNT
);

//Write your properties here - assertions, cover properties and assume properties

// Reset values of the registers
localparam RST_VAL_BYPASS_REG   = 8'h00;
localparam RST_VAL_REQ_REG      = 8'h00;
localparam RST_VAL_ARB_TYPE_REG = 8'h04;

// Address values of the registers
localparam ADDR_BYPASS_REG   = 8'h10;
localparam ADDR_REQ_REG      = 8'h14;
localparam ADDR_ARB_TYPE_REG = 8'h1C;

// APB assumptions
sequence APB_IDLE;
   !PSEL;
```

```
endsequence

// psel without PENABLE (first clock of cycle)
sequence APB_PHASE_1;
   PSEL && !PENABLE;
endsequence

// psel with PENABLE (second clock of cycle)
sequence APB_PHASE_2;
   PSEL && PENABLE;
endsequence

// A complete bus cycle
sequence APB_CYCLE;
   APB_IDLE ##1 APB_PHASE_1 ##1 APB_PHASE_2 ##1 APB_IDLE;
endsequence

property APB_CYCLES_ARE_COMPLETE;
   // Once a cycle has started, it must complete
   @(posedge PCLK) (APB_IDLE |-> APB_CYCLE);
endproperty

property APB_WRITE_AND_ADDR_STABLE;
   // PWRITE and PADDR must be stable throughout the cycle
   @(posedge PCLK) (PSEL |-> $stable({PWRITE, PADDR}));
endproperty

property APB_WRITE_AND_ADDR_VALID;
   // PWRITE and PADDR must be valid throughout the cycle (no X, Z)
   @(posedge PCLK) (PSEL |-> ((^{PWRITE, PADDR}) !== 1'bx));
endproperty

property APB_WRITE_DATA_STABLE;
   // PWDATA must be stable throughout a write cycle
   @(posedge PCLK) ((PSEL && PWRITE) |-> $stable(PWDATA));
endproperty

property APB_NO_PENABLE_OUTSIDE_CYCLE2;
   // If we see PENABLE, it must be in the second clock of a cycle,
   // and it must then go away
   @(posedge PCLK) (PENABLE |-> $stable(PSEL) ##1 (!PENABLE));
endproperty
```

```
property APB_PADDR_RESTRICTED;
   // PADDR can be ADDR_BYPASS_REG = 8'h10 or ADDR_REQ_REG = 8'h14 or
ADDR_ARB_TYPE_REG = 8'h1C;
   @(posedge PCLK) PSEL |-> (PADDR == 8'h10) || (PADDR == 8'h14) || (PADDR ==
8'h1C);
endproperty

property APB_PWDATA_BYPASS_RESTRICTED;
   // PWDATA can be 8'h00 or 8'h01 when PADDR = ADDR_BYPASS_REG
   @(posedge PCLK) PSEL && PWRITE && (PADDR == 8'h10) |-> (PWDATA ==
8'h00) || (PWDATA == 8'h01);
endproperty

property APB_PWDATA_REQ_RESTRICTED;
   // PWDATA must be < 8'd16 when PADDR = ADDR_REQ_REG
   @(posedge PCLK) PSEL && PWRITE && (PADDR == 8'h14) |-> (PWDATA <=
8'd15) && (PWDATA >= 8'd0);
endproperty

property APB_PWDATA_ARB_TYPE_RESTRICTED;
   // PWDATA must be < 8'd6 when PADDR = ADDR_ARB_TYPE_REG
   @(posedge PCLK) PSEL && PWRITE && (PADDR == 8'h1C) |-> (PWDATA <=
8'd05) && (PWDATA >= 8'd0);
endproperty

assume_APB_WRITE_DATA_STABLE : assume
property(APB_WRITE_DATA_STABLE);
assume_APB_WRITE_AND_ADDR_VALID : assume
property(APB_WRITE_AND_ADDR_VALID);
assume_APB_WRITE_AND_ADDR_STABLE : assume
property(APB_WRITE_AND_ADDR_STABLE);
assume_APB_CYCLES_ARE_COMPLETE : assume
property(APB_CYCLES_ARE_COMPLETE);
assume_APB_NO_PENABLE_OUTSIDE_CYCLE2 : assume
property(APB_NO_PENABLE_OUTSIDE_CYCLE2);
assume_APB_PADDR_RESTRICTED : assume
property(APB_PADDR_RESTRICTED);
assume_APB_PWDATA_BYPASS_RESTRICTED : assume
property(APB_PWDATA_BYPASS_RESTRICTED);
assume_APB_PWDATA_REQ_RESTRICTED : assume
property(APB_PWDATA_REQ_RESTRICTED);
assume_APB_PWDATA_ARB_TYPE_RESTRICTED : assume
property(APB_PWDATA_ARB_TYPE_RESTRICTED);
```

```
// APB register write validity check

property APB_BYPASS_REG_VALID;
   // ARB_BYPASS register output can be 0 or 1 at any time
   @(posedge PCLK) (APB_BYPASS == 8'd0) || (APB_BYPASS == 8'd1);
endproperty

property APB_REQ_REG_VALID;
   // APB_REQ register output must be < 8'd16 at any time
   @(posedge PCLK) (APB_REQ <= 8'd15) && (APB_REQ >= 8'd0);
endproperty

property APB_ARB_TYPE_REG_VALID;
   // ARB_TYPE register output must be < 8'd6 at any time
   @(posedge PCLK) (APB_ARB_TYPE <= 8'd05) && (APB_ARB_TYPE >= 8'd0);
endproperty

assert_APB_BYPASS_REG_VALID : assert property(APB_BYPASS_REG_VALID);
assert_APB_REQ_REG_VALID : assert property(APB_REQ_REG_VALID);
assert_APB_ARB_TYPE_REG_VALID : assert
property(APB_ARB_TYPE_REG_VALID);

// APB write check

reg [7:0] pwdata_del;
always @(posedge PCLK) begin
  pwdata_del    <= PWDATA;
end

assert_APB_BYPASS_WR_chk : assert property(@(posedge PCLK) (PREADY &&
PWRITE && (PADDR == 8'h10)) |=> (APB_BYPASS == pwdata_del[0]));
assert_APB_REQ_WR_chk : assert property(@(posedge PCLK) (PREADY &&
PWRITE && (PADDR == 8'h14)) |=> (APB_REQ == pwdata_del[3:0]));
assert_APB_ARB_TYPE_WR_chk : assert property(@(posedge PCLK) (PREADY &&
PWRITE && (PADDR == 8'h1C)) |=> (APB_ARB_TYPE == pwdata_del[2:0]));

// APB read check

assert_APB_BYPASS_RD_chk : assert property(@(posedge PCLK) (PREADY &&
~PWRITE && (PADDR == 8'h10)) |-> ($past(APB_BYPASS) == PRDATA));
assert_APB_REQ_RD_chk : assert property(@(posedge PCLK) (PREADY &&
~PWRITE && (PADDR == 8'h14)) |-> ($past(APB_REQ) == PRDATA));
```

```
assert_APB_ARB_TYPE_RD_chk : assert property(@(posedge PCLK) (PREADY &&
~PWRITE && (PADDR == 8'h1C)) |-> ($past(APB_ARB_TYPE) == PRDATA));

// APB registers reset check
assert_BYPASS_reset_chk: assert property (@(posedge PCLK) $rose(PRESETn) |->
(APB_BYPASS == 8'd0) );
assert_REQ_reset_chk: assert property (@(posedge PCLK) $rose(PRESETn) |->
(APB_REQ == 8'd0));
assert_ARB_TYPE_reset_chk: assert property (@(posedge PCLK) $rose(PRESETn) |->
(APB_ARB_TYPE == 8'd4));

// APB registers cover properties
cover_APB_BYPASS: cover property(@(posedge PCLK) $rose(APB_BYPASS));
cover_APB_WRITE: cover property(@(posedge PCLK) $rose(PREADY & PWRITE));
cover_APB_READ: cover property(@(posedge PCLK) $rose(PREADY & ~PWRITE));

endmodule

module arb_props (
  clk,
  rst_n,
  req,
  arb_type,
  gnt
  );

input       clk;
input       rst_n;
input  [3:0] req;
input  [2:0] arb_type;

input  [3:0] gnt;

// Arbiter properties

a_gnt_onehot : assert property (@(posedge clk) disable iff (~rst_n) $onehot0(gnt));

// Priority scheme when APB_ARB_TYPE = 3'b000
assert_pri_0_gnt0 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[0] &&
$past(arb_type == 3'd0)) |-> $past(req[0]));
assert_pri_0_gnt1 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[1] &&
$past(arb_type == 3'd0)) |-> $past(req[1] & ~req[0]));
```

```
assert_pri_0_gnt2 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[2] &&
$past(arb_type == 3'd0)) |-> $past(req[2] & ~req[1] & ~req[0]));
assert_pri_0_gnt3 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[3] &&
$past(arb_type == 3'd0)) |-> $past(req[3] & ~req[2] & ~req[1] & ~req[0]));

// Priority scheme when APB_ARB_TYPE = 3'b001
assert_pri_1_gnt0 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[0] &&
$past(arb_type == 3'd1)) |-> $past(req[0] & ~req[1]));
assert_pri_1_gnt1 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[1] &&
$past(arb_type == 3'd1)) |-> $past(req[1]));
assert_pri_1_gnt2 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[2] &&
$past(arb_type == 3'd1)) |-> $past(req[2] & ~req[1] & ~req[0]));
assert_pri_1_gnt3 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[3] &&
$past(arb_type == 3'd1)) |-> $past(req[3] & ~req[2] & ~req[1] & ~req[0]));

// Priority scheme when APB_ARB_TYPE = 3'b010
assert_pri_2_gnt0 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[0] &&
$past(arb_type == 3'd2)) |-> $past(req[0] & ~req[2]));
assert_pri_2_gnt1 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[1] &&
$past(arb_type == 3'd2)) |-> $past(req[1] & ~req[2] & ~req[0]));
assert_pri_2_gnt2 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[2] &&
$past(arb_type == 3'd2)) |-> $past(req[2]));
assert_pri_2_gnt3 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[3] &&
$past(arb_type == 3'd2)) |-> $past(req[3] & ~req[2] & ~req[1] & ~req[0]));

// Priority scheme when APB_ARB_TYPE = 3'b010
assert_pri_3_gnt0 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[0] &&
$past(arb_type == 3'd3)) |-> $past(req[0] & ~req[3]));
assert_pri_3_gnt1 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[1] &&
$past(arb_type == 3'd3)) |-> $past(req[1] & ~req[3] & ~req[0]));
assert_pri_3_gnt2 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[2] &&
$past(arb_type == 3'd3)) |-> $past(req[2] & ~req[3] & ~req[0] & ~req[1]));
assert_pri_3_gnt3 : assert property (@(posedge clk) disable iff (~rst_n) (gnt[3] &&
$past(arb_type == 3'd3)) |-> $past(req[3]));

  generate for (genvar i=0; i<=3; i++)
    begin: gen
         assert_gnt_within5_req_rr : assert property( @(posedge clk) disable iff(arb_type
!= 3'b100) $rose(req[i]) |-> ##[1:4] $rose(gnt[i]));
        assert_gnt_within5_req_rand : assert property( @(posedge clk) disable iff(arb_type
!= 3'b101) $rose(req[i]) |-> ##[1:7] $rose(gnt[i]));

     property p_req_until_gnt;
```

```
      @(posedge clk) req[i] |-> req[i][*1:$] ##0 gnt[i];
    endproperty : p_req_until_gnt
    assume_req_until_gnt: assume property (p_req_until_gnt);

    property p_no_req_no_gnt;
      @(posedge clk) $past(req[i]==1'b0) |-> (gnt[i]==1'b0);
    endproperty : p_no_req_no_gnt
    assert_no_req_no_gnt: assert property (p_no_req_no_gnt);

    cover_req: cover property(@(posedge clk) disable iff (~rst_n) $rose(req[i]));
    cover_gnt: cover property(@(posedge clk) disable iff (~rst_n) $rose(gnt[i]));
   end
  endgenerate
endmodule

module Wrapper;
//Binding the properties module with the arbiter module to instantiate the properties
bind arbiter_top apb_props u_apb_props (
  .PCLK(PCLK),
  .PRESETn(PRESETn),
  .PADDR(PADDR),
  .PWRITE(PWRITE),
  .PSEL(PSEL),
  .PENABLE(PENABLE),
  .PWDATA(PWDATA),
  .PRDATA(PRDATA),
  .PREADY(PREADY),
  .APB_BYPASS(APB_BYPASS),
  .APB_REQ(APB_REQ),
  .APB_ARB_TYPE(APB_ARB_TYPE),
  .REQ(REQ),
  .GNT(GNT)
);
bind arbiter arb_props u_arb_props (
  .clk(clk),
  .rst_n(rst_n),
  .req(req),
  .arb_type(arb_type),
  .gnt(gnt)
  );
endmodule
```

Appendix E: RTL Compiler Tcl script to extract timing paths

```
set_attribute hdl_search_path {./} #Set RTL path
set_attribute lib_search_path {./} #Set library path
set_attribute library [list *.lib] #List libraries
set current_design design_name
set myFiles [list *.v]
read_hdl ${myFiles} #Read RTL files
elaborate ${current_design} #Elaborate design
read_sdc ./constraints.sdc #Read timing constraints
check_design -unresolved #Check lint
report timing -lint
synthesize -to_mapped #Synthesize design

report timing -from [all_inputs] -to [all_registers] -worst N > in_to_reg.rpt
report timing -from [all_registers] -to [all_outputs] -worst N > reg_to_out.rpt
report timing -from [all_registers] -to [all_registers] -worst N > reg_to_reg.rpt

puts [all_outputs]
puts [all_registers]
puts [all_inputs]
```

Bibliography

[1] A. C. Cheng, C. C. Yen and J. Y. Jou, "A formal method to improve SystemVerilog functional coverage," *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Huntington Beach, CA, 2012, pp. 56-63.

[2] Luciano Lavagno, Igor L. Markov, Grant E. Martin and Louis K. Scheffer, "*Electronic Design Automation for Integrated Circuits Handbook*", Second Edition.

[3] Cadence Design Systems, Inc., "*Quick Reference for Encounter RTL Compiler"*, Product Version 14.2, October 2014.

[4] Cadence Design Systems, Inc., "*Design For Test in Encounter RTL Compiler"*, Product Version 14.2, June 2016.

[5] Cadence Design Systems, Inc., "*Encounter Conformal Equivalence, Checking Command Reference"*, Conformal L, Conformal XL, and Conformal GXL, Product Version 15.1, April 2015.

[6] Cadence Design Systems, Inc., "*Encounter Conformal Equivalence Checking User Guide*", Conformal L, Conformal XL, and Conformal GXL, Product Version 15.1, April 2015.

[7] Altera Corporation, "*SDC and TimeQuest API Reference Manual*", Quartus II, MNL-SDCTMQ-5.0, 2009.

[8] http://www.vlsi-expert.com/2011/02/synopsys-design-constraints-sdc-basics.html

[9] https://opencores.org/

[10] Y. W. Hsieh and S. P. Levitan, "Model abstraction for formal verification", *Proceedings Design, Automation and Test in Europe*, Paris, 1998, pp. 140-147.

[11] ARM Limited, AMBA™ 3 APB Protocol, v1.0, 2004.

[12] https://verificationacademy.com/

[13] https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/

[14] http://www.asic-world.com/

[15] Cadence Design Systems, Inc., "*JasperGold Apps Command Reference Manual*", Product Version 2015, 09 September 2015.

[16] Cadence Design Systems, Inc., "*JasperGold Visualize GUI Features*", September 2015.

[17] E. Segev, S. Goldshlager, H. Miller, O. Shua, O. Sher and S. Greenberg, "Evaluating and comparing simulation verification vs. formal verification approach on block level design," *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.*, 2004, pp. 515-518.

[18] Intel Corporation, "*IA-32 Intel Architecture Software Developer's Manual on Basic Architecture*", Volume 1, 2003.

[19] Intel Corporation, "*IA-32 Intel Architecture Software Developer's Manual on Instruction Set Reference*", Volume 2, 2003.

[20] Intel Corporation, "*IA-32 Intel Architecture Software Developer's Manual on System Programming Guide*", Volume 3, 2003.

[21] T. K. Callaway and E. E. Swartzlander, "Estimating the power consumption of CMOS adders," *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, Windsor, Ont., 1993, pp. 210-216.

[22] Kshitiz Gupta, "*Automatic generation of coverage directives targeting signal relationships by statically analyzing RTL*", M.S. Thesis, The University of Texas at Austin, TX, 2017.

[23] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2013, pp. 1-10.

[24] Yang Lin and M. Zwolinski, "SETTOFF: A fault tolerant flip-flop for building Cost-efficient Reliable Systems," *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*, Sitges, 2012, pp. 7-12.

[25] Y. Li, Y. Fu, H. Li and S. W. Zhang, "The Improved Training Algorithm of Back Propagation Neural Network with Self-Adaptive Learning Rate," *2009 International Conference on Computational Intelligence and Natural Computing*, Wuhan, 2009, pp. 73-76.

[26] Nikhil Sharma, Gagan Hasteer and Venkat Krishnaswamy, "*Sequential equivalence checking for RTL models*", Article, EE Times, June 2016. Internet: https://www.eetimes.com/document.asp?doc_id=1271433.

[27] https://en.wikipedia.org/wiki/Feature_extraction

[28] https://en.wikipedia.org/wiki/Artificial_neural_network