ECE 510   
FINAL PROJECT REPORT  
VERIFICATION OF PDP8 BASED DUT

Rohit Kulkarni Anuja Vaidya

Contents

[Introduction: 2](#_Toc421800408)

[Design components 3](#_Toc421800409)

[IFD Unit 3](#_Toc421800410)

[Description: 3](#_Toc421800411)

[Execution Unit 4](#_Toc421800412)

[Description: 4](#_Toc421800413)

[Memory 5](#_Toc421800414)

[Description: 5](#_Toc421800415)

[Clock Gen 5](#_Toc421800416)

[Description: 5](#_Toc421800417)

[Unit Level Testbenches 6](#_Toc421800418)

[Unit Level Testbench for IFD unit 6](#_Toc421800419)

[Stimulus: 7](#_Toc421800420)

[Checks to perform 8](#_Toc421800421)

[Coverage 9](#_Toc421800422)

[Unit Level Testbench for EXEC unit 10](#_Toc421800423)

[Stimulus 11](#_Toc421800424)

[Checks to perform 12](#_Toc421800425)

[Coverage: 15](#_Toc421800426)

[Chip level testbench 16](#_Toc421800427)

[Stimulus: 17](#_Toc421800428)

[Checks to perform: 17](#_Toc421800429)

[Coverage: 18](#_Toc421800430)

[Bug Report 20](#_Toc421800431)

[Individual Contribution 22](#_Toc421800432)

[Retrospect 22](#_Toc421800433)

[Figure 1: FSM decode unit 3](#_Toc421799614)

[Figure 2:Fsm: Execution unit 4](#_Toc421799615)

[Figure 3: Unit Level test Bench: IFD unit 6](#_Toc421799616)

[Figure 4: UNIT level test bench: Execution unit 10](#_Toc421799617)

[Figure 5: FULL chip test bench 16](#_Toc421799618)

# Introduction:

Four principal modules in design:

* Clkgen\_driver
* Instr\_decode
* Instr\_exec
* Memory\_pdp

Goal: Build a verification environment around the design

Verification Strategy:

* Unit Level Testbenches for
  + IFD Unit
  + EXEC Unit
* Full chip validation for entire design

# Design components

## IFD Unit

### Description:



Figure : FSM decode unit

* Fetches instruction
* Decodes fetched instruction
* Supports decoding of
  + 6 memory reference instructions
    - Opcode 0 to 5
  + 22 microinstructions
    - Opcode 7 : Group 1 and Group 2

## Execution Unit

### Description:

* Executes instruction decoded by IFD unit.
* Accesses memory if instruction among opcodes 0 to 5.
* Reads operand from memory
* Writes result into memory
* Stalls IFD unit until current instruction completes execution.

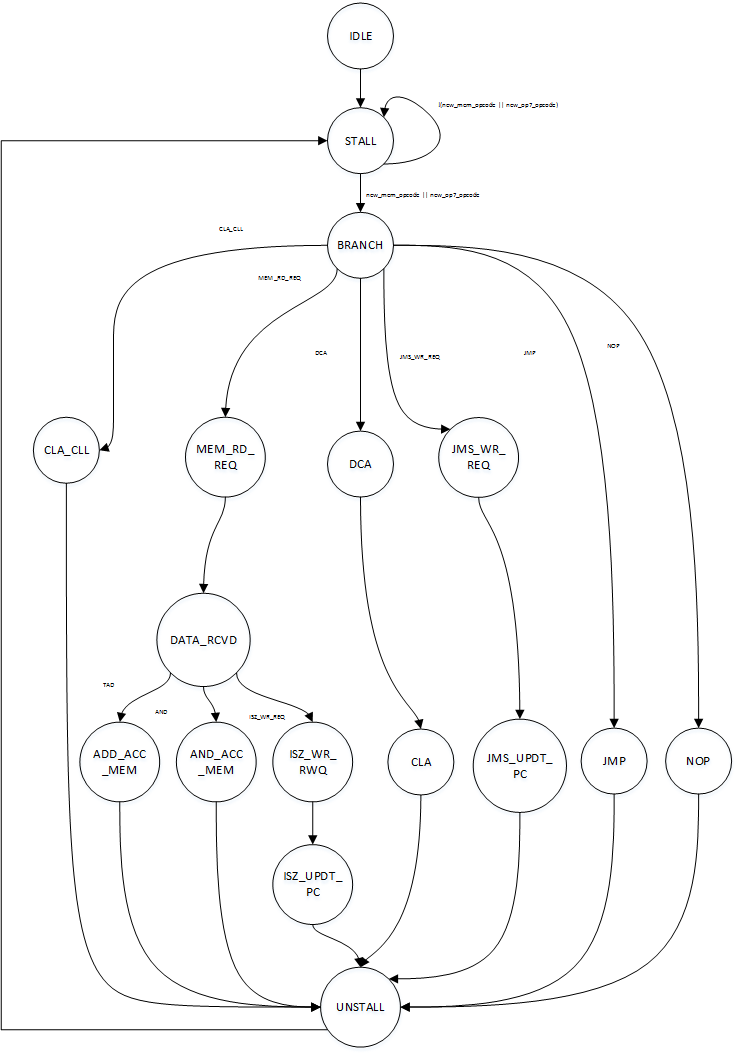


Figure :Fsm: Execution unit

## Memory

### Description:

* The memory unit (4K words of 12bit each) is pre-loaded with data derived from a compiled PDP8 assembly language test. IFD unit fetches instruction from memory. EXEC unit fetches operand from memory and writes back the result.



## Clock Gen

### Description:

* This module provides clock and reset to all the modules in this design.



# Unit Level Testbenches

## Unit Level Testbench for IFD unit



Figure : Unit Level test Bench: IFD unit

### Stimulus:

* **Stimulus:**
  + IFU\_RD\_DATA from MEMORY\_BFM
  + STALL from EXEC\_BFM
  + PC\_VALUE from EXEC\_BFM
  + RESET\_N from CLK\_GEN
* **Type of stimulus:**
* **Purely random stimulus**
* Stall:

Driven by: EXEC\_BFM

Signal to stall instruction decoder.

Stall is randomly generated to be asserted or deasserted.

* PC value:

Driven by: EXEC\_BFM

Current value of Program Counter.

PC value is randomly generated 12 bit number

* IFU\_RD\_DATA:

Driven by: MEMORY\_BFM

Data (Instruction) read by the IFU unit from the memory.

IFU\_RD\_DATA value is randomly generated 12 bit number by memory\_bfm on a ifu\_rd\_req.

**Note:**  Some deterministic stimulus is added in the stimulus generator to obtain hard to achieve cases which could not be generated randomly even after several million simulation cycles in order to gain maximum coverage.

### Checks to perform

* **Type of checking:**

On-the-fly : Primarily, after decoding every opcode.

* **Checks:**
  + Check if opcode sent to EXEC is correctly decoded
  + Check if address sent to EXEC is correctly decoded
  + When STALL is asserted, IFD should not fetch the next instruction and no new requests sent to EXEC unit
  + If reset\_n asserted at any point, then all outputs are cleared within 1 clock cycle
  + On asserting reset, next instruction should be fetched from o200
  + If current state = STALL and PC = 0200, then next state = DONE
  + No instructions are fetched after going into DONE state
  + Check timing requirements such that the FSM stays in the particular state only for 1 cycle.
  + Check whether illegal opcodes are decoded as NOPs
  + If the STALL = 0 and base address is not reached then the FSM should jump into send\_req state or if STALL = 1 and base address has been reached then the FSM should jump to DONE state.
  + On reset internal registers should be reset and the FSM should be in the IDLE state.

### Coverage

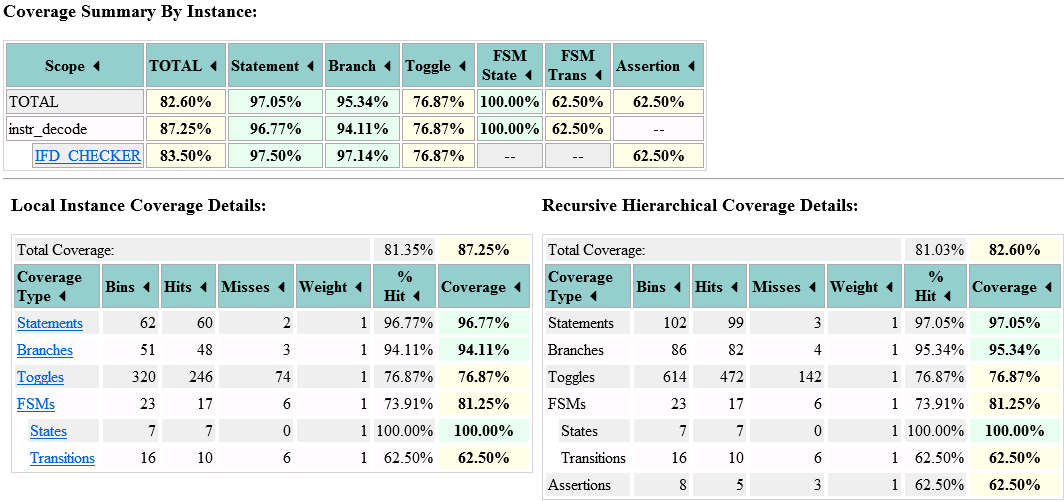
We ran code coverage mechanism on the IFU unit in this unit level testbench.

Following report summarizes the coverage:

Notes:

1. Toggle coverage is slightly low, but the principal reason being base\_address is constant equal to 12'o200 and hence no toggles will be seen on it.
2. If we exclude this, we get 92% toggle coverage.
3. Coverage is calculated by using the code coverage utility tool supported by Questa Sim.
4. A README document will summarize the commands required for running a simulation with coverage in case one has to reproduce results.
5. As promised, we have 100% FSM state coverage.

**IFU UNIT LEVEL TESTBENCH COVERAGE**



## Unit Level Testbench for EXEC unit



Figure : UNIT level test bench: Execution unit

### Stimulus

* **Stimulus:**
* PDP\_MEM\_OPCODE from INSTR\_DECODE\_BFM
* PDP\_OP7\_OPCODE from INSTR\_DECODE\_BFM
* EXEC\_RD\_DATA from MEMORY\_BFM
* RESET\_N from CLK\_GEN
* **Type of stimulus:**
* **Purely random stimulus**
* PDP\_MEM\_OPCODE:

Driven by: INSTR\_DECODE\_BFM

Random pdp\_mem\_opcode is driven by the INSTR\_DECODE\_BFM.

A random number is generated for pdp\_mem\_opcode which is 18 bit packed struct.

* PDP\_OP7\_OPCODE:

Driven by: INSTR\_DECODE\_BFM

Random pdp\_op7\_opcode is driven by the INSTR\_DECODE\_BFM.

A random number is generated for pdp\_op7\_opcode which is a 22 bit packed struct.

* EXEC\_RD\_DATA:

Driven by: MEMORY\_BFM

Data (operand) read by the EXEC unit from the memory.

EXEC\_RD\_DATA value is randomly generated 12 bit number by memory\_bfm on a exec\_rd\_req.

**Note:** Some deterministic stimulus is added in the stimulus generator to obtain hard to achieve cases which could not be generated randomly even after several million simulation cycles in order to gain maximum coverage.

### Checks to perform

* **Type of checking:**

On-the-fly : Primarily, after every instruction retire

* **Method of checking:**
* Computes the golden result to be checked after every instruction retire.
* By retire, we mean the point when instruction has written its result and a new instruction is fetched.
* We use the State [UNSTALL] to detect instruction retire.
* Following golden results are generated to be compared with:
  + 1: **Golden accumulator** contents
  + 2: **Golden link** bit contents
  + 3: **Golden PC** contents
  + 4: **Golden result** written to memory
* Check if Accumulator contents are correct:

For all instructions that require writing the result into Accumulator

1. AND : **C(AC)** <- C(AC) AND C(EAddr)
2. TAD : **C(AC)** <- C(AC) + C(EAddr)
3. DCA : **C(AC)** <- 0
4. CLA\_CLL: **C(AC)** <- 0
5. Other instructions should leave **C(AC)** as they are

* Check if link bit contents are correct:

1. TAD : If carry out then complement **Link**
2. CLA\_CLL: **Link** <- 0
3. No other instruction should modify **link** bit

* Check if PC is computed correctly

1. ISZ : **C(PC)** <- C(PC) + 2
2. JMS : **C(PC)** <- EAddr + 1
3. JMP : **C(PC)** <- EAddr
4. All other instructions: **C(PC)** <- C(PC) + 1

* Check if result is computed correctly and written to memory

1. ISZ : **C(EAddr)** <- C(EAddr) + 1
2. DCA : **C(EAddr)** <- C(AC)
3. JMS : **C(EAddr)** <- C(PC)
4. No other instruction should modify **memory**

**Checks to be performed for each state transition:**

|  |  |  |  |
| --- | --- | --- | --- |
| **State** | **Instruction response check** | **State transition** | **Number of cycles** |
| CLA\_CLL | Clears Acc and link register | Unstall | 1 |
| Mem\_RD\_REQ | Gets the address from the instruction and read request is sent to that ADDRESS | Data\_rcvd | 1 |
| Data\_rcvd | Data received from the memory and latched. Present value of ACC and linker are stored in a temp reg | Jumps to different states based on the instructions/opcodes | 1 |
| ADD\_Acc\_mem | Data from the memory location from the instruction is added with the accumulator. Carry is obtained in the link which is complemented. TAD is done here | Unstall | 1 |
| AND\_Acc\_mem | Data from the memory is ANDed with the data in the accumulator, AND operation is done here. | Unstall | 1 |
| ISZ\_Wr\_Req | Data from memory is incremented. Write add is calculated here | ISZ\_UPDT\_PC | 1 |
| ISZ\_UPDT\_PC | PC is incremented here conditionally ,ISZ is done here | Unstall | 1 |
| DCA | Write request is asserted here and it gets the address location from the opcode. Write data is the data in the Acc | CLA | 1 |
| CLA | ACC is cleared here | Unstall | 1 |
| JMS\_WR\_REQ | Write request is asserted here and it gets the address location from the opcode. Write data is the PC value | JMS\_UPDT\_PC | 1 |
| JMS\_UPDT\_PC | PC is updated with the value from the PC and JMS is done here | Unstall | 1 |
| JMP | PC is updated with the value from the PC and JMP is done here | Unstall | 1 |
| NOP |  | Unstall | 1 |
| Unstall | Stall is cleared  Write request is cleared and PC is incremented internally | Stall | 1 |

### Coverage:

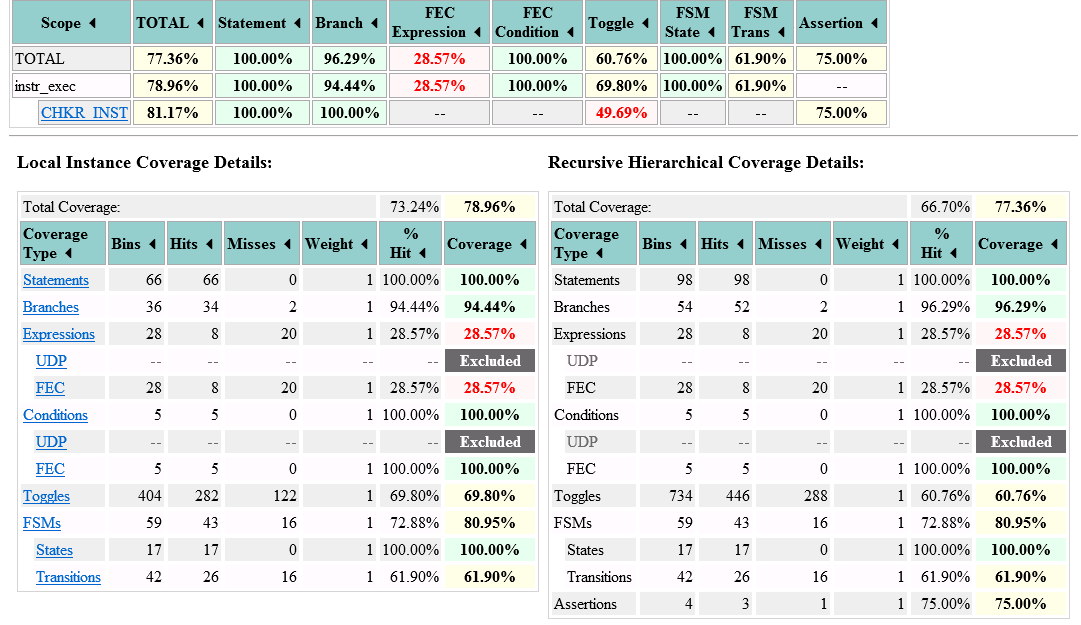
We ran code coverage mechanism on the EXEC unit in this unit level testbench.

Following report summarizes the coverage:

Notes:

1. Coverage is calculated by using the code coverage utility tool supported by Questa Sim.
2. A README document will summarize the commands required for running a simulation with coverage in case one has to reproduce results.
3. As promised, we have 100% FSM state coverage.

**EXEC UNIT LEVEL TESTBENCH COVERAGE**



# Chip level testbench



Figure : FULL chip test bench

### Stimulus:

* **Stimulus:**
* Stimulus: Assembly code loaded in memory
* Stimulus should cover all possible opcodes
* We cannot have illegal opcodes assembled and hence we dropped this possibility
* Stimulus covered corner cases like:
  + TAD : Producing carry
  + TAD : Not Producing carry
  + ISZ : Skipping on zero
  + ISZ : Not Skipping on zero
  + AND : Toggling all 12 bits
* **Type of stimulus:**
* Deterministic:

Assembly code loaded in memory

* Random:

PERL script to generate random test.mem files

A seed is provided to generate reproducible random memory files to be loaded in the PDP memory.

### Checks to perform:

* Both the unit level checkers are reused :

1. IFD checker
2. EXEC checker

* All checks performed by the unit level checkers are performed at the full chip.

### Coverage:

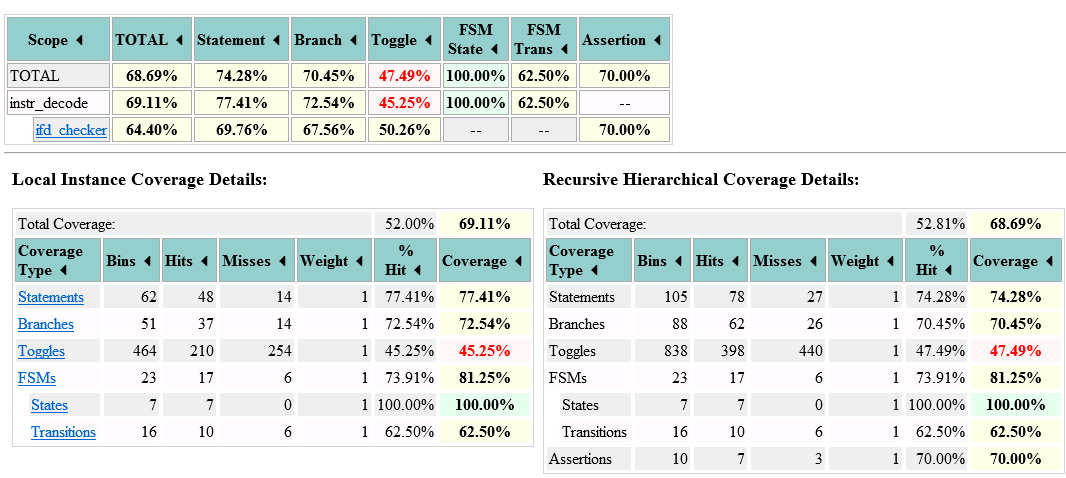
We ran code coverage mechanism on the IFU unit in this full chip level testbench.

Following report summarizes the coverage:

Notes:

1. Coverage is calculated by using the code coverage utility tool supported by Questa Sim.
2. A README document will summarize the commands required for running a simulation with coverage in case one has to reproduce results.
3. Some of the ASM files could not be assembled and resulted in getting lower coverage at the full chip level testbench.
4. It is equally difficult to generate stimulus at full chip level which can give us better or comparable coverage to unit level testbench.

**IFU FULL CHIP TESTBENCH COVERAGE**



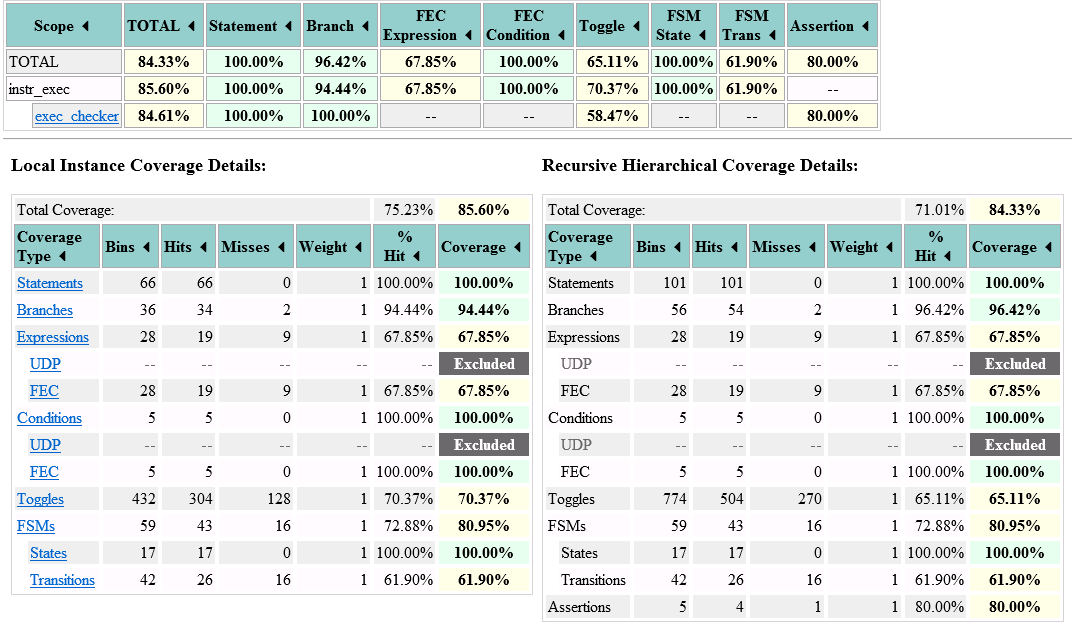
We ran code coverage mechanism on the EXEC unit in this full chip level testbench.

Following report summarizes the coverage:

Notes:

1. Coverage is calculated by using the code coverage utility tool supported by Questa Sim.
2. A README document will summarize the commands required for running a simulation with coverage in case one has to reproduce results.
3. Some of the ASM files could not be assembled and resulted in getting lower coverage at the full chip level testbench.
4. It is equally difficult to generate stimulus at full chip level which can give us better or comparable coverage to unit level testbench.

**EXEC FULL CHIP TESTBENCH COVERAGE**



# Bug Report

With the robust verification environment we built around the DUT, we were in a good state to catch any bugs that crept in the design.

We also did a code review to spot any obvious bugs lurking in the design.

1. **Bug in computing Link bit**

**Type of bug: Implementation bug by designer**

**Description:**

According to the specification, for a TAD operation,

C(AC) <- C(AC) + C(EAddr) and if carry out then complement Link.

Our checker found out that link bit does not complement when the carry is 1.

Instead, the link bit is a function of the previous carry and not the current carry obtained by the instruction retired.

1. **Initial state of Accumulator and Link bit on reset are not defined**

**Type of bug: Ambiguity in architectural specification**

**Description:**

The specification does not define the initial state of accumulator and link bit.

On starting the simulation, both (Acc and Link) are initialized to 'X.

We feel this is an ambiguous specification and should be resolved by talking to the architects.

**Remedy:**

Start all tests by clearing accumulator and link using the CLA\_CLL instruction to get Accumulator and Link bit to a predictable state.

1. **Inherent priority in decoding opcodes not mentioned in the specification**

**Type of bug: Ambiguity in architectural specification**

**Description:**

We see that there is an inherent priority in decoding opcodes. The case statement in IFU is where it is implemented. This is not mentioned anywhere in the design and should be reviewed with the architects.

1. **Ambiguity in specification for ISZ**

**Type of bug: Ambiguity in architectural specification**

**Description:**

Unsure whether to take the old data or the new data for checking.

1. **INSTR\_DEC does not check for stall on coming out of reset**

**Type of bug: Ambiguity in architectural specification / implementation**

**Description:**

Should there be support for a case in which instruction fetch and decode comes out of reset, but exec unit asserts STALL preventing IFU to not fetch the first instruction? Need to clarify with architects

1. **I and M bits in the instruction don't have any significance:**

**Type of bug: Incorrect implementation**

**Description:**

I and M bits in the instruction are just a part of the address instead of their special significance. The address is 9 bits [3: 11] but the address lines are 12 bits and the upper 3 bits are always 0.

An instruction having the format

 TAD I Z <LABEL>

would not execute as expected since we do not support decoding of I and M bits in the instruction.

1. **Code review :**

**Type of bug: Incorrect use of constructs**

**Description:**

In the FSM implementation of IFD and EXEC, we found out that non-blocking assignments are used inside always\_comb procedural blocks.

Since code inside always\_comb is meant to be purely combinational, it is highly recommended to use blocking assignments to describe combinational behaviour. Had this guideline been followed, the link bit bug (bug no.1) would not have crept in the design.

# Individual Contribution

|  |  |
| --- | --- |
| **Anuja** | 1: Owning Full Chip testbench  2: Understanding and writing assembly test case stimulus for full chip environment  3: Monitoring coverage for full chip environment  4: Adding some unit level checks  5: Documenting results |
| **Rohit** | 1: Owning unit level testbenches  2: Developing reusable unit level checkers and stimuli  3: Script to generate random memory files to be used as random stimulus for full chip testbench  4: Monitoring unit level coverage  5: Documenting results and report |

# Retrospect

* We learnt a lot while working on this project.
* We feel we did our best, but we would have liked to do better given more time.

Specifically:

* Add a golden memory model to the verification environment
* Do end-of-test memory check with the golden model
* Develop more robust checks
* Get to 100% coverage on all 3 testbenches.
* Functional coverage for some corner cases