ECE 510 FUNDAMENTAL PRE-SILICON FUNCTIONAL VALIDATION

SPRING 2016

PROJECT REPORT

FOR

VERIFICATION OF HARDWARE IMPLEMENTAION OF PDP8 INSTRUCTION SET ARCHITECTURE (ISA) LEVEL SIMULATOR

BY

BHARATH REDDY GODI,

SURENDRA MADDULA.

Contents

1. Micro-architectural description of the design

1.1 Introduction:

The PDP-8 (Programmed Data Processor) was introduced in 1965 by Digital Equipment Corporation (DEC). The original PDP-8 was built using transistor technology making it a second generation computer but a third generation model using MSI technology, the PDP8/I, was introduced in 1968.The PDP-8 was considered a *mini-computer* because of its small size and low cost. Memory consisted of 4K words, each word being 12 bits. The small 12 bit word size permitted only eight op-codes although two of the eight codes made efficient use of extended op-code fields to raise the actual number of operations to over 50. The cost was $18K which in the 1960's was positively cheap for a computer!

1.2 Micro-architecture of pdp8

The PDP8 Hardware Simulator consists of three main units:

- Instruction Fetch and Decode Unit (IFD)
- Execution Unit (EXE)
- Memory Unit (MEM)

The memory unit has 4K words, each 12bit long. PDP8 assembly language test loads the MEM unit with 12 bit data. The IFD fetches the instruction to be decoded, decodes it and sends the decoded instruction to the EXE unit which performs the required operation. The next instruction to be fetched by the IFD is determined based on the PC value. The entire process is repeated until the program counter value reaches the initial value where it started.

1.3 PDP8-Instructions

The PDP-8 is a "load and store" computer. Each instruction on the PDP-8 is contained in one 12 bit word. These are classified into three types:

- Memory Reference Instructions
- Input Output Transfer Instructions
- Micro Instructions

1.4 Memory Reference Instructions:

Mnemonic Symbol	Octal Code	Operation
AND	0	Logical AND
TAD	1	Two’s Complement Add
ISZ	2	Increment and Skip if Zero
DCA	3	Deposit and Clear Accumulator
JMS	4	Jump to Subroutine
JMP	5	Jump

1.5 Micro Instructions (opcode-7) Group-1:

Mnemonic Symbol	Octal code	Operation
NOP	7000	No Operation
CLA	7200	Clear Accumulator
CLL	7100	Clear Link
CMA	7040	Complement Accumulator
CML	7020	Complement Link
IAC	7001	Increment Accumulator
RAR	7010	Rotate Accumulator and Link Right
RTR	7012	Rotate Accumulator and Link Right twice
RAL	7004	Rotate Accumulator and Link Left
RTL	7006	Rotate Accumulator and link left Twice

1.6 Micro Instructions (opcode-7) Group-2:

Mnemonic Symbol	Octal code	Operation
SMA	7500	Skip on Minus Accumulator
SZA	7440	Skip on Zero Accumulator
SNL	7420	Skip on Non Zero Link
SPA	7510	Skip on Positive Accumulator
SNA	7450	Skip on Non Zero Accumulator
SZL	7430	Skip on Zero Link
SKP	7410	Skip Always
CLA	7600	Clear Accumulator
OSR	7404	Or Switch Register with Accumulator
HLT	7402	Halt

1.7 Addressing Modes:

In PDP-8, each memory reference instruction assumes a particular 5-bit page number. Thus the 7-bit offset address in an offset into the assumed page. A memory reference instruction references a value stored in memory; the address of the value referenced is called the 'effective address'.

Zero Page Addressing	If the memory page bit (bit 4) of a memory reference instruction is cleared to 0. This means that the contents of the 7 bit offset field is extended on the left by 5 zeros to yield a 12-bit effective address on page 0. The assumed page is the
Current Page Addressing	If the memory page bit (bit 4) is set to 1. The 12 bit effective address for a word which is on the same memory page as the instruction. The assumed page is the current page the instruction is on.
Indirect Addressing	It is indicated by setting the Indirect bit (bit 3) to 1. In this case the 12 bit value obtained by zero page or current page addressing is not the effective address, it is the address of the effective address. Thus the PDP-8 fetches the 12 bit address from the memory. This allows an instruction to get around the limitations imposed by zero or current page addressing so that any address in the memory can be referenced
Auto Indexing	The eight memory locations $(0010)_8$ to $(0017)_8$ on page zero are special auto index registers, whenever these locations are accessed its contents is first incremented and effective address is fetched from this newly incremented address.

2. Understanding of the full chip Design:

The Full chip system consists of a Memory Unit, Instruction fetch and decode unit, and an Instruction execution unit. The memory consists of 4096 words. Where word length of each word is 12 bit each. The memory unit reads the data from a compiled pdp8 assembly language test.

The chip will be in reset initially. It comes out of reset when reset_n signal is de-asserted. Once after coming out of reset, the instruction fetch and decode unit fetches the first instruction from a base address i.e. Octal 200.

The instruction fetch and decode unit sends the fetched instruction to instruction execution unit where it does the specified operations based on instruction and reads from memory or writes from memory. After finishing with the current instruction, IFD fetches the next instruction from the memory unit and gives it to EXE unit. This process continues until all the instructions in the memory are completed. The last instruction address is same the base address just to indicate that it is the last it is the instruction.

3. Supported Instructions:

AND: **Logical AND.** The content of memory location specified by 'xxx' is combined with the content of the AC using a bitwise logical AND operation. The result is left in the AC.

TAD: **Two's Compliment Add.** The content of memory location specified by 'xxx' is added to the contents of the AC using two's compliment addition. The result is left in the AC, if the result overflows, the LINK bit is complimented.

ISZ: **Increment and Skip if Zero.** The content of memory location specified by 'xxx' is incremented by one and restored to memory. If the result of the increment was zero, the program counter (PC) is incremented one extra time to cause the execution of the instruction following the ISZ to be skipped.

JMS: **Jump to Subroutine.** The address of the instruction following this one is stored into the memory location specified by 'xxx' and the PC is set to the location following 'xxx'.

JMP: **Jump.** The PC is set to the location specified by 'xxx', causing the machine to execute its next instruction from there.

CLA_CLL: The **accumulator** and **link** will be cleared.

DCA: Deposit and Clear Accumulator. The content of the accumulator (AC) is stored into the memory location specified by 'xxx' and the AC is cleared to zero.

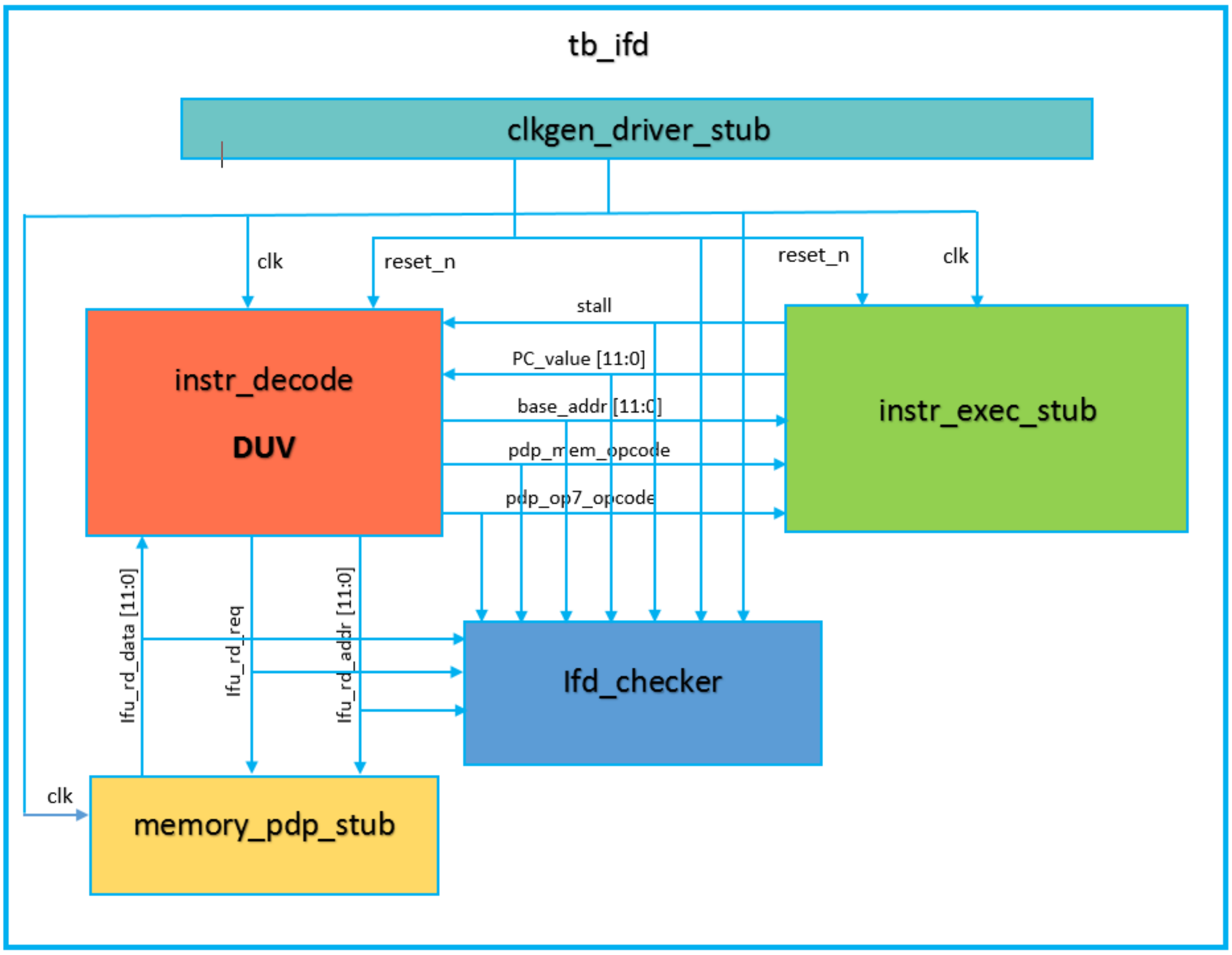
NOP: No Operation

4. Unit Level Testing

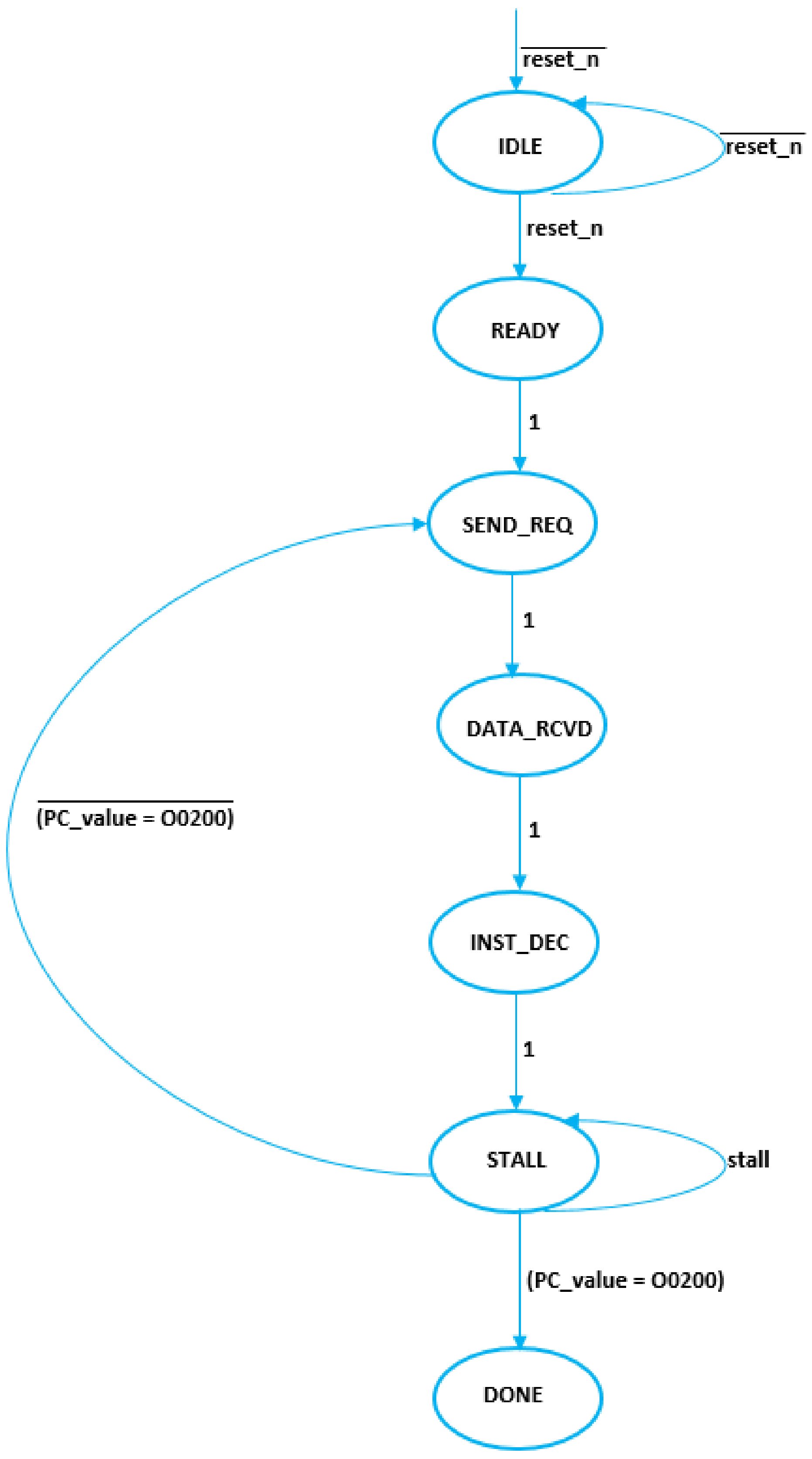
4.1 Instruction Fetch and Decode Unit

When the reset_n is asserted (=1) the start address (o200) is pushed into the base address. Based on this, the decode unit sends the read request and the address of the PC to fetch the instruction from the memory. The memory unit now maps the address received to the address present in its memory and sends the stored value (data) to the decode unit. Now the decoding unit divides the instructions into two branches based on the [11:9] bits of the data obtained from the memory. If the bits [11:9] are <6 the instructions are treated as memory reference instructions or if the bits are equal to 7 they are treated as Micro Instructions: opcode 7. After diving, the decoding unit assigns signals to every instruction so that the execution unit can differentiate and continue with the appropriate operations. Also the decoder unit sends different signals to each of the type. It also sends the Base address to the execution unit so that it can update the PC value and send it back as PC values calculation differs for each instruction.

4.1.1 Unit Level Block Diagram of Instruction Fetch & Decode unit



4.1.2 Finite State Machine for Instruction Decoder Unit


$\overline{\text{reset\_n}}$
IDLE
$\overline{\text{reset\_n}}$
reset_n
READY
1
SEND_REQ
1
DATA_RCVD
$\overline{\text{(PC\_value = 00200)}}$
1
INST_DEC
1
STALL
stall
(PC_value = 00200)
DONE

4.1.3 Verification Strategy of Instruction Fetch & Decode Unit

Verification environment consists of a stimulus generators, instr_exec_stub, memory_pdp_stub, clkgen_driver_stub and ifd_checker.

a. Stimulus Generator:

The test stimulus comprises of both deterministic and random stimuli. Initially we verified with deterministic tests and then moved to the rigours random tests and also created corner cases that we have planned.

b. memory_pdp_stub stimuli:

memory_pdp_stub need to mimic memory module of PDP8. This will be a responder stimulus component, which will be responding to the signal 'ifu_rd_req' signal and will generate an 'Ifu_rd_data' which will be an instruction that need to go under decode stage. Below are some of the test stimulus which serves over 10000 'ifu_rd_req' which receives from instr_decode unit.

S.No	Type of test stimulus	Test stimulus
1.	Deterministic	These are stimulus generated to cover 'Micro Instructions' like NOP, HLT, CLL, CLA1, CLA_CLL, CLA2, IAC, RAL, RTL, RAR, RTR, CML, CMA, CIA, OSR, SKP, SNL, SZL, SZA, SNA, SMA, SPA.
2.	Random	These are the stimulus generated to cover different possible instructions and for regression testing. We will cover the memory reference instructions in this type of stimulus.

c. Instr_exec_stub stimuli:

Instr_exec_stub need to mimic instruction execution unit of PDP8. This will work as a responder stimulus component, this unit will assert a 'stall' whenever a valid instructions is issued and should give a 'PC_value' at the end of the execution. This unit helps in generating two corner cases.

S.No	Type of test stimulus	Test stimulus
1.	Corner Case sequence 1	PC_value should be 12'o0200 for the second time after reset, which makes IFD unit stop fetching data from memory.
2.	Corner Case sequence 2	Create a scenario, where execute unit have a bug which doesn't respond to a reset. So that when reset is issued only IFD unit will respond.

d. Clkgen_driver_stub stimuli:

Clkgen_driver_stub is an initiator stimulus component, which will makes all other units to work according to a clock and reset issued by it. This component will generate a 'reset' once after certain period of time.

e. Checker Component:

This component comprises of both normal and assertion based checking. Below are some of the black box checking rules that were framed to test the IFD unit according to its specifications.

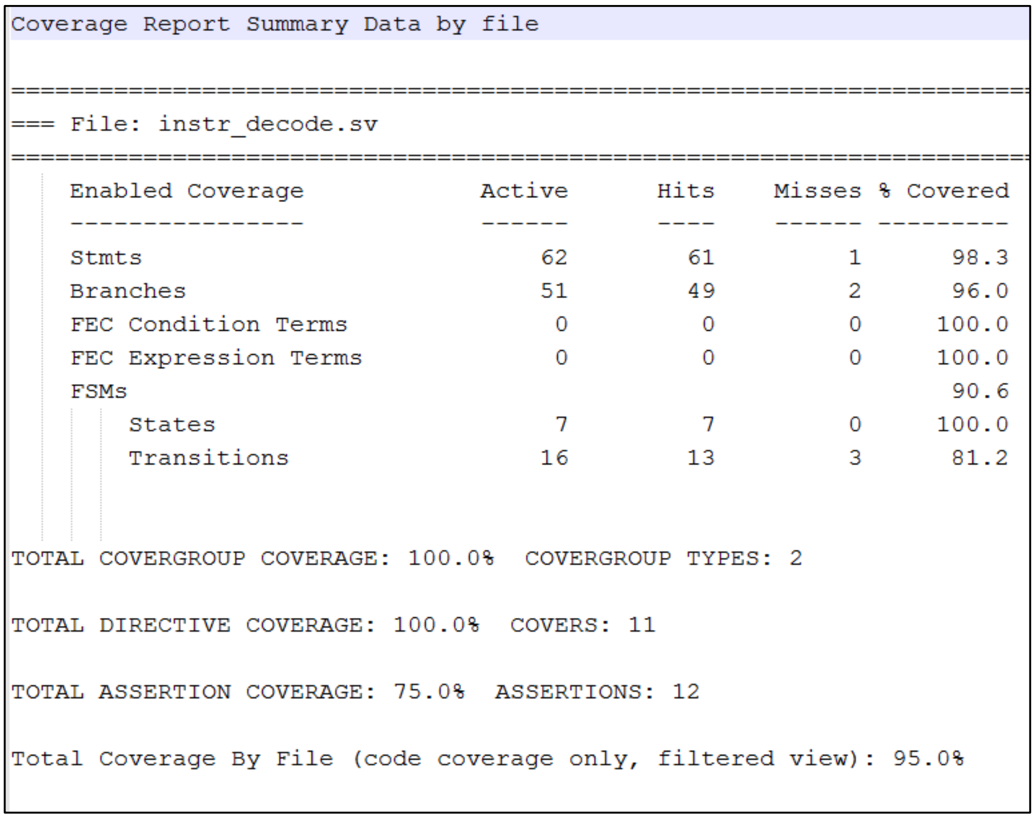
S.No	Checker Rules
1.	After reset_n is asserted low to high base_address should set to a start address (octal 200).
2.	After reset_n is asserted low to high, ifu_rd_req == 0 && ifu_rd_addr == 0 && pdp_mem_opcode === '{0,0,0,0,0,0,9'bz} && pdp_op7_opcode === '{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}.
3.	Ifu_rd_req should never be asserted high when stall is high (**corner case**).
4.	When stall is high, values on outputs ifu_rd_addr, pdp_mem_opcode and pdp_op7_opcode shouldn't change.
5.	When ifu_rd_req is asserted, then PC_value and ifu_rd_addr should be same.
6.	After reset is asserted low to high, the ifu_rd_addr should set to base_addr after one clock cycle.
7.	After ifu_rd_req is asserted low to high , the decoded values should be present on either pdp_mem_opcode or pdp_op7_opcode in the second clock cycle
8.	When an unsupported instruction is issued, pdp_mem_opcode should be '{0,0,0,0,0,0,9'bz} and pdp_op7_opcode should be '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}.
9.	When PC_value and ifu_rd_addr matches with the base address for the first time, **pdp_mem_opcode** = '{0,0,0,0,0,0,9'bz} and **pdp_op7_opcode** = '{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}.
10.	After program starts, when the PC_value is octal 200 for the second time i.e. after the last instruction is executed, the **ifu_rd_req** should not be asserted high.
11.	Int_rd_req should be high only for one clock cycle.
12.	pdp_mem_opcode and pdp_op7_opcode shouldn't set more than one instruction at a time

f. Coverage Rules:

We have framed some rules for functional coverage in order to make sure we have created required stimulus to test specifications.

S.No	Coverage Rules
1.	Check whether all valid instructions (both memory reference instructions and micro-instructions) have been issued or not.
2.	Check whether we have generated invalid instructions.
3.	Corner Case: Check whether we have generated the corner case sequence 1.
4.	Corner Case: Check whether we have generated the corner case sequence 2.

We have achieved 100% of functional coverage and nearly 100% of code coverage.



```
Coverage Report Summary Data by file

=======================================================================
=== File: instr_decode.sv
=======================================================================
    Enabled Coverage              Active      Hits    Misses % Covered
    ----------------              ------      ----    ------ ---------
    Stmts                             62        61         1      98.3
    Branches                          51        49         2      96.0
    FEC Condition Terms                0         0         0     100.0
    FEC Expression Terms               0         0         0     100.0
    FSMs                                                          90.6
        States                         7         7         0     100.0
        Transitions                   16        13         3      81.2


TOTAL COVERGROUP COVERAGE: 100.0%  COVERGROUP TYPES: 2

TOTAL DIRECTIVE COVERAGE: 100.0%  COVERS: 11

TOTAL ASSERTION COVERAGE: 75.0%  ASSERTIONS: 12

Total Coverage By File (code coverage only, filtered view): 95.0%
```

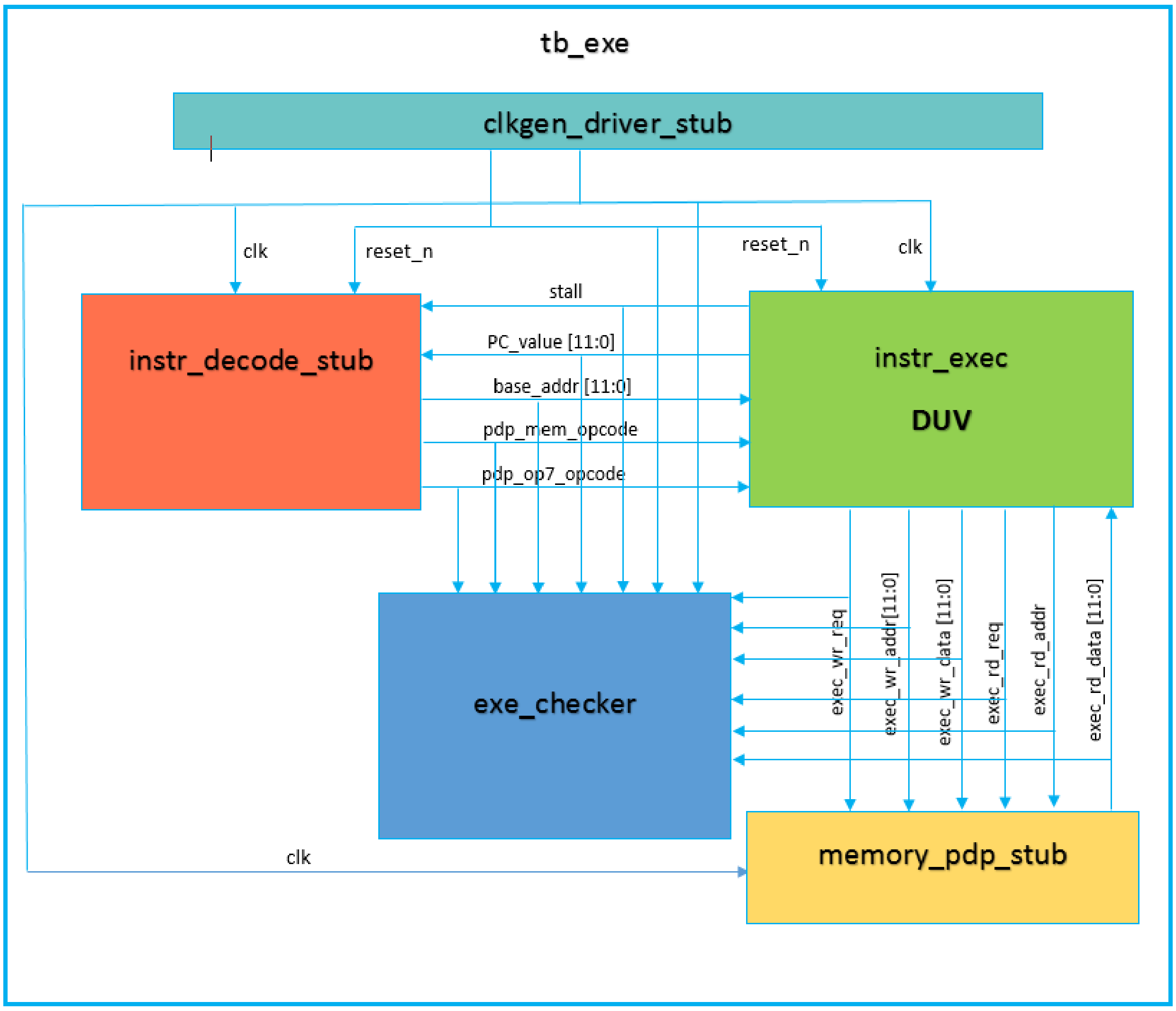
g. Bugs:

1. We have found one bug in IFD unit. It failed at corner case sequence 2. Which is the reason for failure of rules 3, 4, and 5.

4.2 Execution Unit

Only few of the instructions are supported by the execution unit and the rest are treated as no operation. After receiving the signals for the instructions the execution unit identifies the operands and performs the operations. After getting the signal it then sends a stall signal to the decoding unit indicating it to wait until it executes the entire instruction and send an updated value of PC. Based on the instruction if the execution unit had to write data to the memory it first sends the write request, the address in which the data should be written and what data is to be written. For the read operation in a similar way it first sends the request to read and the address to be read and then the memory unit sends the corresponding data. After executing the instruction the executing unit goes to a stall state after updating the PC and sending it back to the decoding unit. As soon as the decoder gets the PC value the entire process starts again.

4.2.1 Unit Level Block Diagram of Execution unit



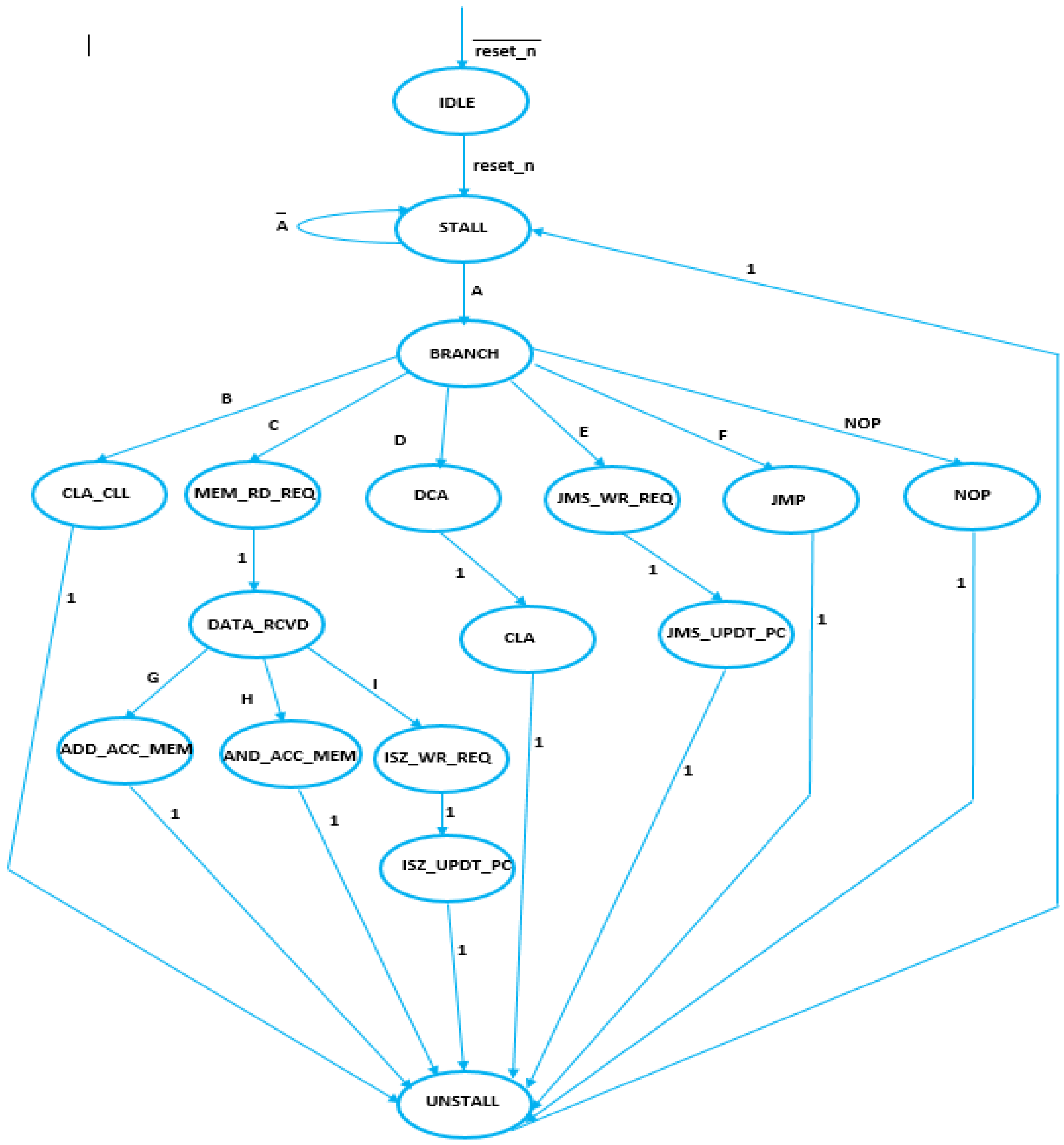
4.2.2 Finite State Machine for Instruction Decoder Unit

A= new_mem_opcode || new_op7_opcode, B= pdp_op7_opcode.CLA_CLL,

C= pdp_mem_opcode.TAD || pdp_mem_opcode.AND || pdp_mem_opcode.ISZ,

D= pdp_mem_opcode.DCA, E= pdp_mem_opcode.JMS, F= pdp_mem_opcode.JMP,

G= pdp_mem_opcode.TAD, H= pdp_mem_opcode.AND, I= pdp_mem_opcode.ISZ



4.2.3 Verification Strategy of Execution Unit

Verification environment consists of a stimulus generators, instr_decode_stub, memory_pdp_stub, clkgen_driver_stub and exe_checker.

a. Stimulus Generator:

The test stimulus comprises of both deterministic and random stimuli. Initially we verified with deterministic tests and then moved to the rigours random tests and also created some corner cases that we have planned.

b. memory_pdp_stub stimuli:

memory_pdp_stub need to mimic memory module of PDP8. This will be a responder stimulus component, which will be responding to the signal 'exec_wr_req' and 'exec_rd_req' signals and will generate an 'exec_rd_data' for an 'exec_rd_addr' or 'exec_wr_addr' that an EXE unit require to compute. Below are some of the test stimulus which serves over 10000 'ifu_rd_data' requested by execute unit.

S.No	Type of test stimulus	Test stimulus
1.	Deterministic	These are the stimulus to generate corner case scenarios like pushing values to the boundaries and overflow conditions. 1. When an 'exec_rd_addr' == 12'o1010, it generates read data stimulus 'exec_rd_data' == 12'o0000. 2. When an 'exec_rd_addr' == 12'o1717, it generates read data stimulus 'exec_rd_data' == 12'o7777.
2.	Random	It generates random data signal for all 'exec_rd_addr' signal except 'exec_rd_addr' == 12'o1717 or 12'o1010.

c. instr_decode_stub stimuli:

instr_decode_stub need to mimic instruction decode unit of PDP8. This will work as an initiator stimulus component, this unit will set base address to 12'o200 when it comes out of to reset state. This unit generates two main inputs to EXE unit, they are 'pdp_mem_opcode_s' and 'pdp_op7_opcode_s' format signals called pdp_mem_opcode and pdp_op7_opcode. Their instruction format is as given below example:

AND instruction with memory location 12'o0763 is represented as {1'h1, 1'h0, 1'h0, 1'h0, 1'h0, 1'h0, 12'h763}

So just before going into the type of stimulus let's see the format of instruction they represent pdp_mem_opcode_s = { AND, TAD, ISZ, DCA, JMS, JMP, mem_inst_addr } &

pdp_op7_opcode_s = {NOP, IAC, RAL, RTL, RAR, RTR, CML, CMA, CIA, CLL, CLA1, CLA_CLL, HLT, OSR, SKP,SNL, SZL, SZA, SNA, SMA, SPA, CLA2}

S.No	Type of test stimulus	Test stimulus
1.	Random	It first generates randomly instructions among memory reference or micro instructions Memory reference instructions will generate memory instruction address randomly (constrained random) within 12’o7777.
2.	Corner case scenario 1	Once after execute unit asserts stall, try changing the instruction type or memory address value. This corner case is checked for memory reference instructions like AND, TAD and ISZ
3.	Corner case scenario 2	Push the values to their boundaries to meet the overflow conditions. Simulate values like 12’o7777 and 12’o0000 during instructions like TAD, ISZ, DCA, AND instructions

d. clkgen_driver_stub stimuli:

clkgen_driver_stub is an initiator stimulus component, which will makes all other units to work according to a clock and reset issued by it. This component will generate a ‘reset’ once after certain period of time.

e. Checker Component:

This component comprises of both normal and assertion based checking. Below are some of the black box checking rules that were framed to test the IFD unit according to its specifications.

S.No	Checker rules	Rule No.
1.	Exec_rd_req and exec_wr_req shouldn’t be high at same time.	Rule 1
2.	If exec_rd_req or exec_wr_req is high, then stall should be high.	Rule 2
3.	When stall fell, PC value should be updated in the same cycle except during instruction ISZ	Rule 3
4.	During instructions AND, TAD, and ISZ, int_exec_rd_addr should be equal to pdp_mem_opcode.mem_inst_addr.	Rule4A
5.	During DCA, JMS & ISZ instructions, int_exec_wr_addr should be equal to pdp_mem_opcode.mem_inst_addr.	Rule 4B

6.	At the end of ISZ instruction execution, PC_value should be incremented by 2, when the content of memory location is zero after incremented	Rule 10
7.	JMS instruction should expect exec_wr_req asserted before its execution ends	Rule 20A
8.	DCA instruction should expect exec_wr_req asserted before its execution ends	Rule 20B
9.	ISZ instruction should expect exec_wr_req asserted before its execution ends	Rule 20C
10.	AND instruction should expect exec_rd_req asserted before its execution ends	Rule 20D
11.	ISZ instruction should expect exec_rd_req asserted before its execution ends	Rule 20E
12.	TAD instruction should expect exec_rd_req asserted before its execution ends	Rule 20F
13.	exec_wr_data should be PC+1(following instruction) value when exec_wr_req is high during execution of instruction JMS	Rule 16
14.	When reset_n is low, PC value should be base address.	Rule 18

In order to check functionality of some instructions we need to check the internal variable and bits like accumulator and link, which need to be checked by white box method of instruction. To avoid touching the DUT code, we used the method of creating an interface with internal variables of a DUT as its ports through the method of binding. Below are some of the rules made in order to check the specifications of the respective instructions.

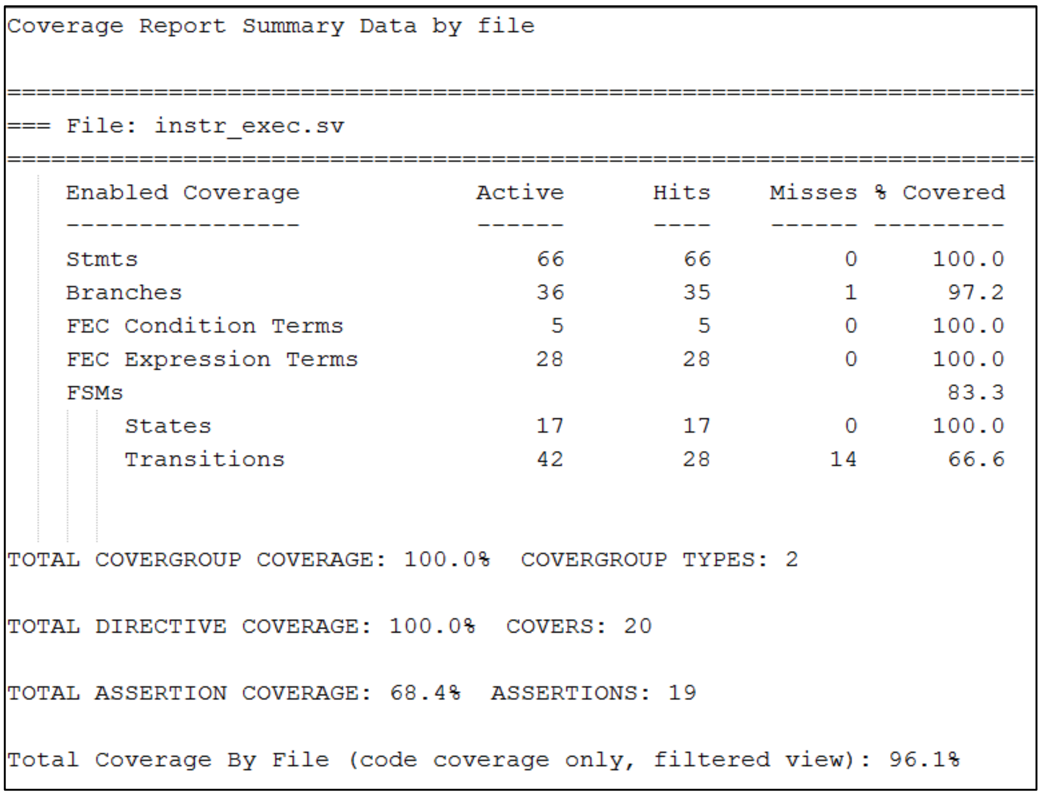
S.No	**Checker rules**	**Rule No.**
1.	Checking an accumulator content after stall is set high .i.e. During AND operation check: AC <- AC & int_exec_rd_data, when stall is high.	Rule 7
2.	During TAD operation check: AC <- AC + int_exec_rd_data, when stall is high & Link <- ~Link, when stall fells	Rule 8
3.	During CLA_CLL operation: Accumulator and link, should be cleared by the end of the instruction.	Rule 11
4.	During DCA operation check whether exec_wr_data is equal to accumulator value, when exec_wr_req is high	Rule 12
5.	During DCA operation check whether Accumulator will be cleared by the end of instruction.(i.e When stall fell)	Rule 13

f. Coverage Rules:

We have framed some rules for functional coverage in order to make sure we have created required stimulus to test specifications.

S.No	Coverage Rules
1.	Check whether all possible instructions in 'pdp_mem_opcode' & 'pdp_op7_opcode' have been issued
2.	Corner Case: Check whether a stimulus generated which causes an overflow after addition
3.	Corner Case: Check whether **exec_rd_data** are 12'o0000 and 12'o7777 during instruction ISZ
4.	Corner Case: Check whether a stimulus sequence generated, where **pdp_mem_opcode. mem_inst_addr** and **pdp_mem_opcode** changes before execution finishes.

We have achieved 100% of functional coverage and nearly 100% of code coverage.



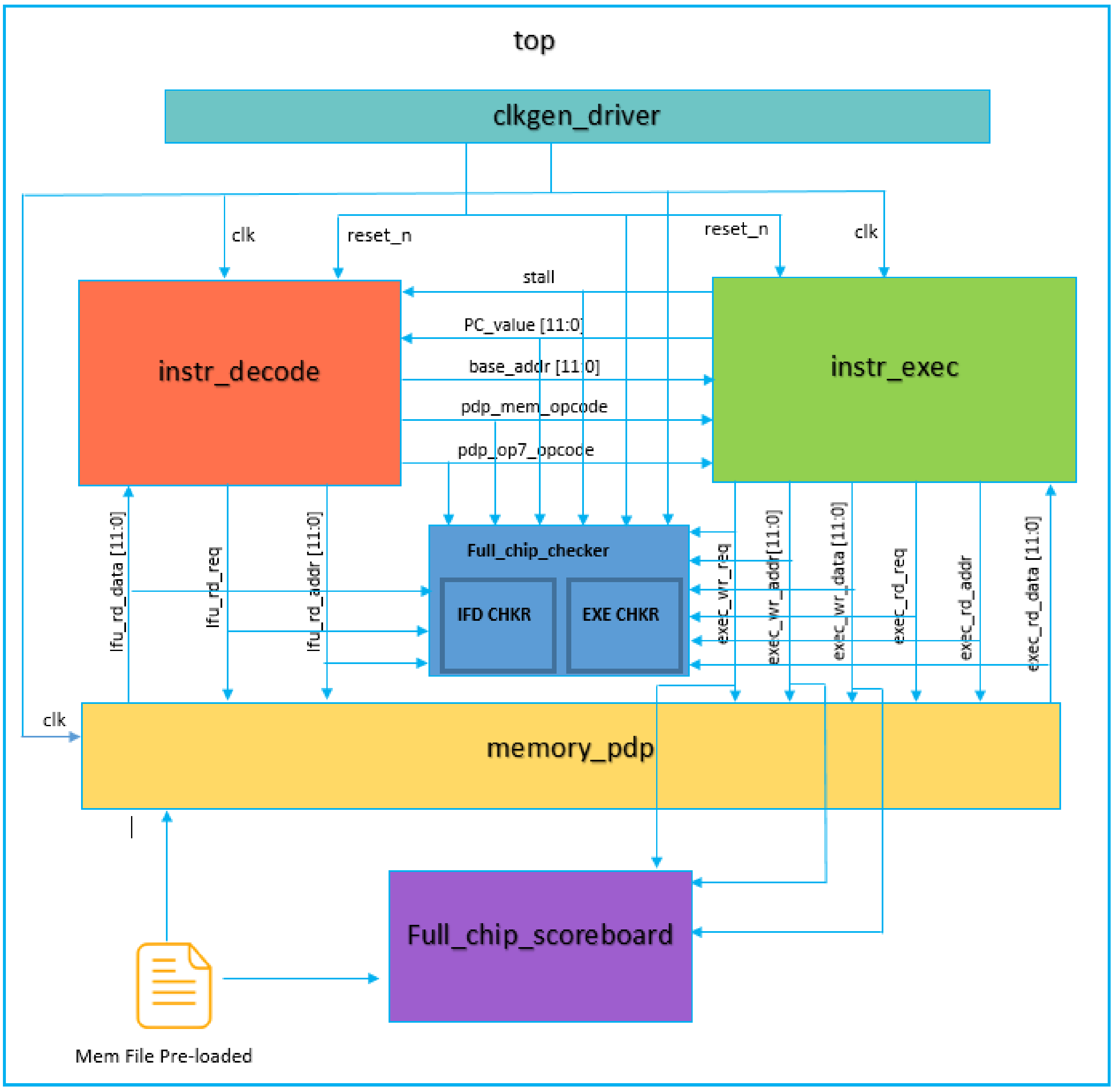
```
Coverage Report Summary Data by file

=====================================================================
=== File: instr_exec.sv
=====================================================================
    Enabled Coverage            Active      Hits    Misses  % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           66        66         0     100.0
    Branches                        36        35         1      97.2
    FEC Condition Terms              5         5         0     100.0
    FEC Expression Terms            28        28         0     100.0
    FSMs                                                        83.3
        States                      17        17         0     100.0
        Transitions                 42        28        14      66.6


TOTAL COVERGROUP COVERAGE: 100.0%  COVERGROUP TYPES: 2

TOTAL DIRECTIVE COVERAGE: 100.0%  COVERS: 20

TOTAL ASSERTION COVERAGE: 68.4%  ASSERTIONS: 19

Total Coverage By File (code coverage only, filtered view): 96.1%
```

g. Bugs:

1. We have found 5 bugs in execute unit related to checker rules 4A, 7, 12, 8 & 10. They are explained clearly in the separate documentation.

5. Full Chip Level Testing

5.1 Chip Level Block Diagram



5.2 Verification Strategy of Full Chip

Verification environment consists of a full_chip_checker, and full_chip_scoreboard for the verification of the pdp8 full chip module.

a. Assembly Stimulus:

Assembly code in memory file are pre-loaded into the ‘memory_pdp’, which issues the instructions to the decode stage of PDP-8. We follow the formal verification procedure.

We have written 5 different possible scenarios covering all the supported and valid functions and have been listed down them in a file.(\Full_Chip\docs\Chip level assembly tests.pdf)

b. Full_chip_checker:

Full_chip_checker is a combination of two checker files, they are ifd_checker and exe_checker re-used, which are verification components that are from the lower hierarchical level (unit level checkers)

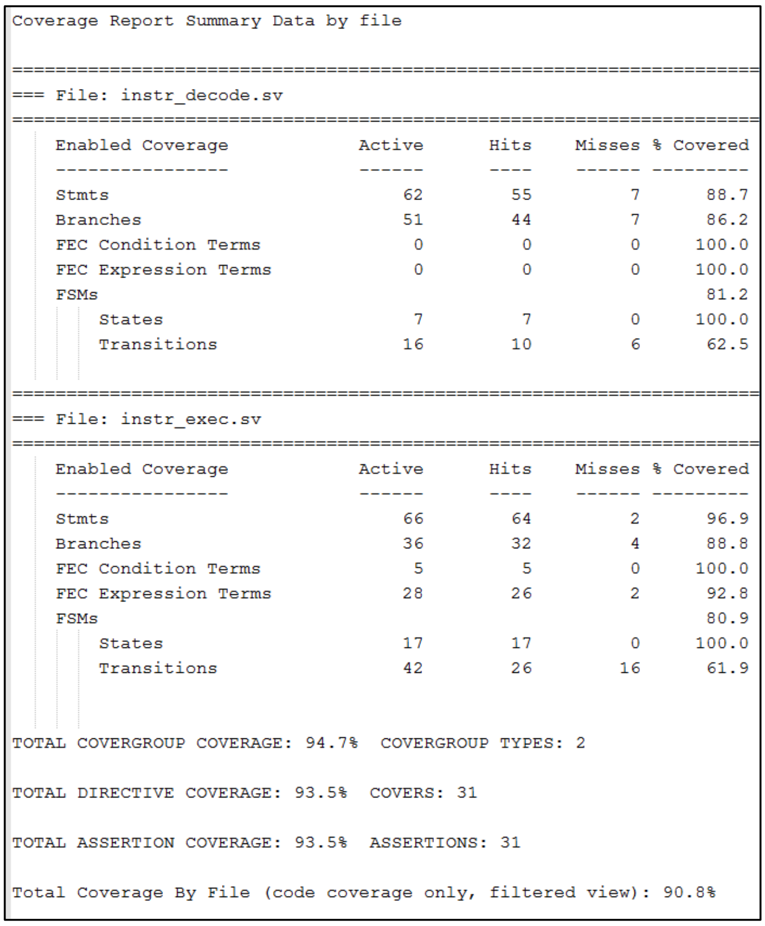
c. Full_chip_scoreboard:

This scoreboard is used to perform the end of test memory check. It will be preloaded by the memory file and it executes instructions by reference model.

d. Coverage Rules:

1. We need to check some of the instruction sequences to be executed like shown below.

S.No	Coverage Rules
1.	Check whether all possible instructions of ‘pdp_mem_opcode’ & ‘pdp_op7_opcode’ have been issued.
2.	Check for the instruction sequence CLA_CLL --> TAD --> TAD --> DCA --> HLT --> JMP.
3.	Check for the instruction sequence TAD--> AND-->ISZ.
4.	Check for the instruction sequence JMP--> JMS.


```
Coverage Report Summary Data by file

=====================================================================
=== File: instr_decode.sv
=====================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           62        55         7      88.7
    Branches                        51        44         7      86.2
    FEC Condition Terms              0         0         0     100.0
    FEC Expression Terms             0         0         0     100.0
    FSMs                                                        81.2
        States                       7         7         0     100.0
        Transitions                 16        10         6      62.5

=====================================================================
=== File: instr_exec.sv
=====================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           66        64         2      96.9
    Branches                        36        32         4      88.8
    FEC Condition Terms              5         5         0     100.0
    FEC Expression Terms            28        26         2      92.8
    FSMs                                                        80.9
        States                      17        17         0     100.0
        Transitions                 42        26        16      61.9


TOTAL COVERGROUP COVERAGE: 94.7%  COVERGROUP TYPES: 2

TOTAL DIRECTIVE COVERAGE: 93.5%  COVERS: 31

TOTAL ASSERTION COVERAGE: 93.5%  ASSERTIONS: 31

Total Coverage By File (code coverage only, filtered view): 90.8%
```

e. Bugs:

1. We have found one bug in full chip related to Rule 5 of EXE checker. They are explained clearly in the separate documentation.

6. Project management aspect:

6.1 Required tools

- Questa sim simulator.
- Version: Questa Sim -64 10.4c
- Verification Language - System Verilog

6.2 Risks and dependencies

Risks: There are no potential risks observed. Since all the requirements are cleared and development code was freeze.

Dependencies: Schedule for the validation activity has been prepared. Resource and tool availability has no problem with executing the plan.

6.4 Resources requirements

Team members:

- Bharath Reddy Godi: Unit level verification of Execution unit.
- Surendra Maddula: Unit level verification of Decode.

Checker rules for IFD and EXE unit, Chip level verification, coverage and report written by both of us.

6.5 Retrospect

- **What went well?**
  Decode unit and Full chip went well. Execution unit was a bit challenging. But at the end, we were able to do it.
- **What could have been better?**
  If the specification was bit detailed it would have been better.
- **What are the things you would improve if you had the opportunity to do it again?**
  If we had more time we could have implemented Constrained Randomization techniques using Class and program blocks.

6.6 Schedule details

S.NO	DATE	TASK
1	05/08/2016	Understanding of Project.
2	05/11/2016	More Clarity about the project. Proposal preparations.
3	05/18/2016	Proposal demo.
4	05/23/2016	Planned completion of IFD unit.
5	05/26/2016	Planned completion of EXE unit.
6	05/29/2016	Planned completion of memory unit.
7	06/05/2016	Full chip verification
8	06/07/2016	Documentation
9	06/08/2016	Final Demo.

7. References:

1. http://www4.wittenberg.edu/academics/mathcomp/shelburne/comp255/notes/PDP8Overview.htm
2. http://homepage.cs.uiowa.edu/~jones/pdp8/man/micro.html#group1
3. https://en.wikipedia.org/wiki/PDP-8
4. **Lecture slides**