International Institute of Information Technology
Bangalore

Functional Verification of SoCs

VL 701

Course Project - CDV of CV32E40P Processor

Submitted by: *Arhant Arora* (IMT2020503)

Asmita Zjigyasu (IMT2020507)

Saket Gurjar (IMT2020520)

Yash Dharmesh Mogal (IMT2020537)

Contents

1 The CV32E40P Processor

The CV32E40P processor core, developed by the OpenHW Group, showcases notable advancements in the RISC-V architecture. Extending the Instruction Set Architecture (ISA) of CV32E40P, the design incorporates a range of supplementary instructions such as hardware loops, post-increment load and store instructions, and additional ALU instructions, which extends the standard RISC-V ISA.

It includes the following instructions:

- Standard RISC-V ISA (I)
- Standard Extension for Compressed Instructions (C)
- Standard Extension for Integer Multiplication and Division (M)
- Performance Counters
- Control and Status Register Instructions (CSR)
- Instruction-Fetch Fence

The following instruction sets can be enabled by setting some parameters in the core of the processor:

- Single-Precision Floating-Point (F) - enabled when FPU = 1
- CORE-V ISA Extensions - enabled when PULP_XPULP = 1
- PULP Cluster Extension - enabled when PULP_CLUSTER = 1
- PULP Share Integer (X) Registers with Floating Point (F) Register Extension - enabled when PULP_ZFINX = 1

The main module of the processor is *cv32e40p_core.sv* which builds connections between other modules.

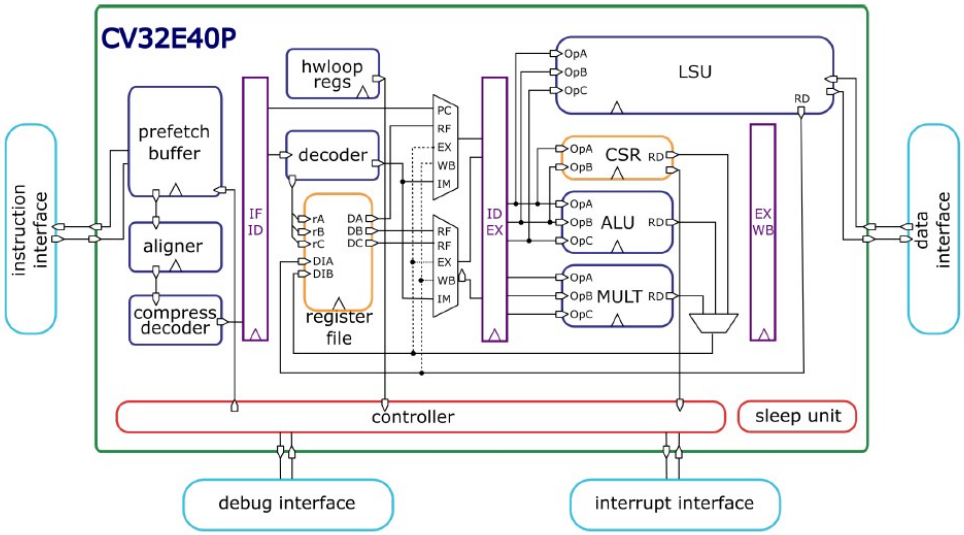

Figure 1: Block Diagram of CV32E40P RISC-V Core

1.1 Overview of the Stages in the Processor

1. **Instruction Fetch stage (IF) -** The pre-fetcher retrieves instructions from the instruction memory, while the aligner ensures that misaligned instructions are properly aligned to either the word boundary (32 bits) or the half-word boundary (16 bits). In the case of compressed instructions, the compress_decoder decompresses them. Furthermore, the aligner and pre-fetch buffer manage the setting of the program counter (PC) to the appropriate jump_address or branch_address whenever a jump or branch signal is encountered.

2. **Instruction Decode Stage (ID) -** In this stage, the instruction is decoded, and its specific instruction set and functionality are determined based on the Opcode. The necessary data is fetched from the register file using the appropriate source register addresses, while the destination register address is forwarded to the ID/EX pipeline for use in the write-back (WB) stage. Additionally, the immediate value is sign-extended to ensure proper handling. Moreover, the controller is responsible for generating signals to manage stalls and facilitate data forwarding in the event of data hazards.

3. **Execute Stage (EX) -** In this stage, arithmetic operations are executed on the data received from the previous stage (ID stage) via the ID/EX pipeline, guided by the ALU operation code provided by the ID stage. The operations encompass various types, including integer arithmetic, multiplication, division, and floating-point operations. Branches (with their condition met) are taken from the EX stage. Multi-cycle instructions will stall this stage until they are complete. The address generation part of the load-store-unit (LSU) is contained in EX as well.

4. **Writeback stage (WB) -** In this stage, the write-back process is performed, involving the storage of data from the ALU into either the register file or the memory, depending on the instruction Opcode. The data generated by the ALU is appropriately handled and stored based on the specific instruction requirements. The writeback to the data memory and the register file is done through the LSU module in this stage.

1.2 Overview of the Pipelines in the Processor

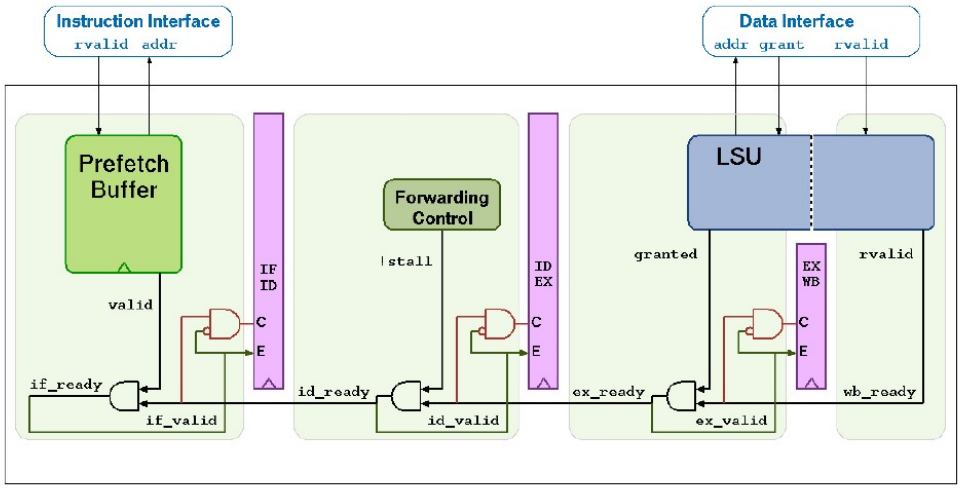

Figure 2: CV32E40P Pipeline

1. **IF/ID Pipeline -** It contains the data to be sent to the ID stage by the IF stage, like, fetched instruction, if the instruction is valid, etc.
2. **ID/EX Pipeline -** It contains the data to be sent to the EX stage, like the source register data, ALU Opcode, destination register address, etc.; or to be forwarded to the next instruction's ID stage like the source registers' data (in case of data hazards), etc.
3. **EX/WB Pipeline -** It contains the data to be sent to the WB (LSU) stage by the EX stage, like, calculated data, calculated address, etc.

2 The Layered Testbench Approach for CDV

A layered testbench has been developed for the CDV approach. Randomized inputs were used instead of directed test vectors as the latter does not guarantee that all test-cases have been covered.

The following classes were used for constructing the layered test-bench:

- Transaction
- Generator
- Driver
- Monitor
- Scoreboard
- Environment
- Test (Program Block)
- Design Under Test (Verilog Module)
- Interface (Interface Data-type)
- Testbench-Top (Verilog Module)

1. **Transaction :** Consists of all the instructions and input-output variables that are needed by the Design Under Test (D.U.T.). These parameters are initialized as random variables of fixed length. Coverage is also measured here.
2. **Generator** : Generates random stimulus based on transaction class and given constraints. The data is sent to the driver via a mailbox
3. **Driver :** Receives the data from the Generator via Mailbox and converts it to the form of inputs to the DUT and passes it on to the interface
4. **Monitor :** Records all inputs-outputs passing into and coming out of the interface. Sends this data to the scoreboard.
5. **Scoreboard :** This implementation includes the reference model inside the scoreboard. It receives the data from the monitor via a mailbox and evaluates it according to the reference model.
6. **Environment :** All the above classes are instantiated here.

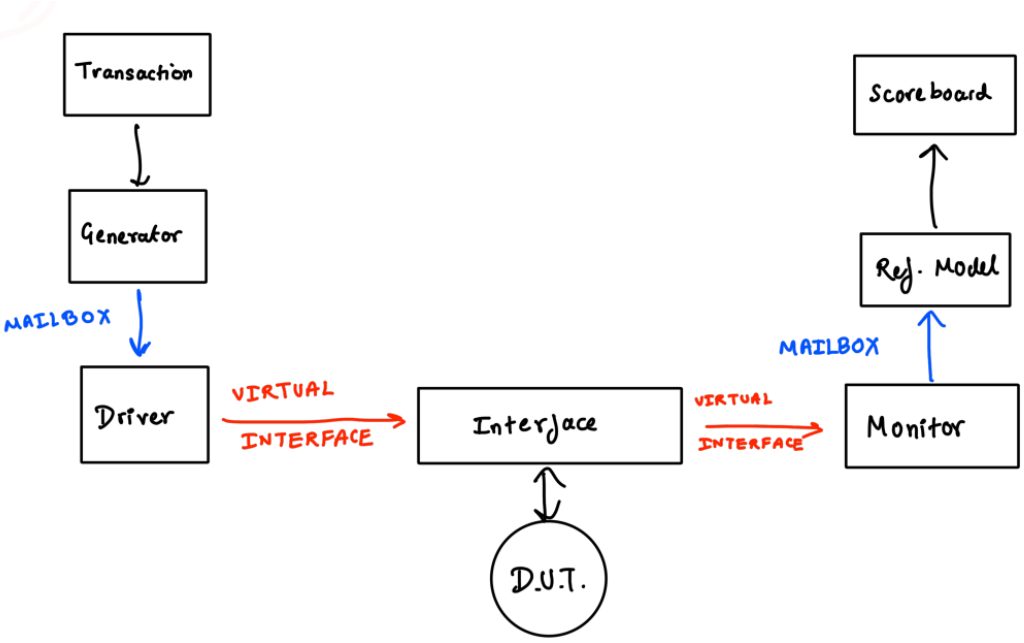

Figure 3: Layered Test-bench

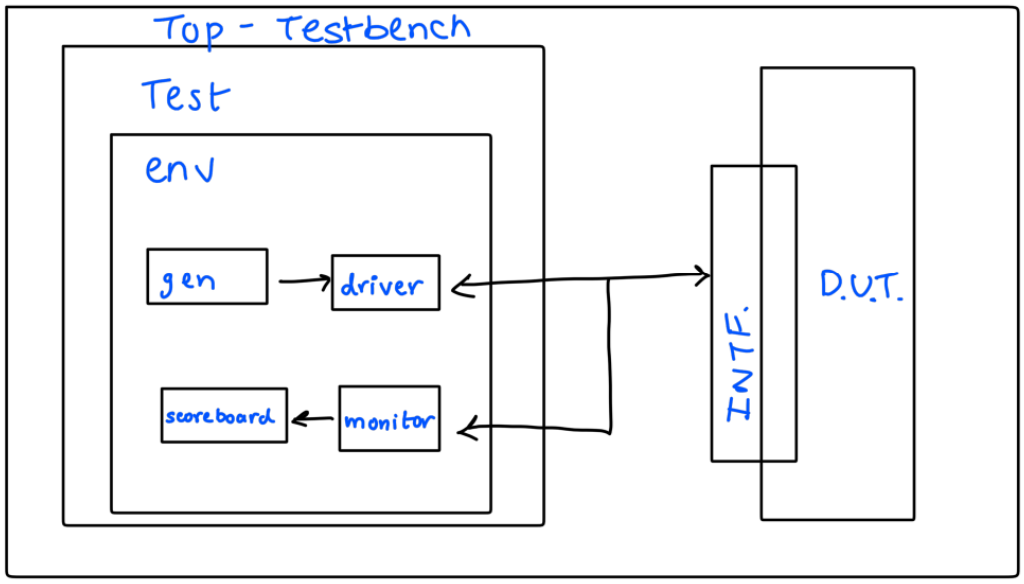

Figure 4: Layered Test-bench

3 Parts of the CV32E40P Processor

3.1 Instruction Fetch Stage

Instruction Fetch is the first stage of our processor. It fetches the instruction from the memory (one per cycle) based on the program counter (PC) value and passes it on to the decoder stage. The main input signals it takes are the instr_rdata_i that has the 32-bit instruction, the instr_rvalid_i signal which indicates the validity of data in the rdata signal and the instr_gnt_i to indicate the request has been accepted. The

main output signals are the instr_addr_o and the instr_req_o.

The following submodules are utilized in this stage:

- **Aligner**: The IF is supposed to handle word-aligned fetches only. In case the instructions are misaligned, it is the aligner that is called upon to deal with this by performing two separate word-aligned fetches. Aligning instructions simplifies the decoding and execution stages of the pipeline.
- **Compressed Decoder**: Compressed instructions are those which are encoded in 16 bits instead of 32 bits. Compressed instructions allow more instructions to fit in the same amount of memory that provides higher performance. The compressed_decoder module converts these compressed instructions into full-sized ones before sending to the further stages.
- **Prefetch Buffer**: The prefetch buffer performs word-aligned 32-bit prefetches from the instruction memory and stores the fetched words in a FIFO with a number of entries depending on the DEPTH parameter. It is used to reduce the memory latency in fetching instructions as it operates ahead of the processor's execution pipeline. It then performs buffering, that is storing the instructions in a buffer depending on its capacity.
  The prefetch buffer also makes use of 3 more submodules:
  - **Prefetch Controller**: This is what controls the buffer. It deals with the validity of transaction requests and response, the readiness of fetch, and how the buffer will connect to the FIFO module.
  - **FIFO**: This deals with storing data. Commands like push, pop, and other buffer-related queries are dealt with by this.
  - **OBI Interface**: This is the on-chip bus interface. It provides an interface for blocks to communicate information and data. Each block can be defined independently and then connected to the OBI. This also allows modularity and scalability.

3.2 Instruction Decode Stage

This stage performs instruction decoding by analyzing instructions received from the instruction fetch (IF) stage. It identifies the instruction set, functionality, and the corresponding ALU operation. Additionally, it decodes source and destination register addresses, accessing the Register file to retrieve data from the source registers for processing by the ALU. The destination register address is forwarded to the ID/EX pipeline for the write-back (WB) stage. If an immediate value is present, it is also decoded and sign-extended. The controller block generates multiple control signals for use in subsequent and preceding stages.

There are 5 submodules being called in the ID stage:

1. **Register File -** Flipflop Register file is being used here as the Latch-based Register file is used for ASIC (Synthesis) implementations. The source register addresses are input into the register file:
   - *Read Port A* - Source 1 address is sent to Register File and the data is fetched and given to the ID stage
   - *Read Port B* - Source 2 address is sent to Register File and the data is fetched and given to the ID stage
   - *Read Port C* - Source 3 address is sent to Register File and the data is fetched and given to the ID stage

- *Write Port A* - Data sent from the WB stage is written into the register file at the address sent from the WB stage (for that instruction, WB address for the register file is forwarded from the ID stage to the register file through the next stages and the pipeline registers)
- *Write Port B* - Data forwarded from the WB stage of the instruction which was the 2nd instruction previous to the current instruction is sent to the register file in this stage. This forwarding is done due to the following RAW hazard: Load instr [dest reg = a] → Some instruction → Add [source reg = a] instruction.

2. **Decoder -** This module decodes the type of instruction (I, M, F, Complex, etc.), the functionality of the instruction (Load, Store, Add, SLL, Mult, etc.), the ALU operation that will be done on the data that will be sent to the ALU (ALU Opcode) and decodes which register file port to access for the source registers according to the instruction breakdown. It also sends out some CSR signals, Hardware loop instruction Signals, jump and branch signals, etc.

3. **Controller -** This module in this stage sends out some CSR signals, Forwarding signals, Debug signals, sleep signals, etc. It takes in certain stall and forwarding signals according to the decoded instructions, interrupt signals, LSU usage signals.

4. **Interrupt Controller -** This module works on the interrupt signals and sends these signals to the decoder to check whether or not to interrupt the pipeline.

5. **Hardware Loop Registers -** To increase the efficiency of small loops, CV32E40P supports hardware loops (HWLoop) optionally. The hardware loop instruction start, end addresses, and the count of the number of loops are sent from the ID stage to the hwloop_regs module for computation.

   - Hardware loops make executing a piece of code multiple times possible, without the overhead of branches or updating a counter.
   - Hardware loops involve zero stall cycles for jumping to the first instruction of a loop.
   - A hardware loop is defined by its start address (pointing to the first instruction in the loop), its end address (pointing to the instruction that will be executed last in the loop), and a counter that is decremented every time the loop body is executed.
   - CV32E40P contains two hardware loop register sets to support nested hardware loops, each of them can store these three values in separate flip-flops. Loop number 0 has a higher priority than loop number 1 in a nested loop configuration, meaning that loop 0 represents the inner loop.

3.3 Execute Stage

3.4 Load-Store Unit

The Load Store Unit (LSU) in the CV32E40P processor is one of the crucial modules which is responsible for managing the memory accesses and data transfers between the processor core and the memory system. Load and store operations are supported on words (32-bit), half words (16 bit), and bytes (8-bit). It interacts with ID stage to obtain decoded instruction that gives information regarding LOAD/STORE operation, from which address data needs to be read and load/store to which address in the memory/reg, and the type of data provided in the source address. For example:

- **Register/Mem to Mem/Register Load:**
  - **cv.lb rD, rs2(rs1)** → $rD = Sext(Mem8(rs1 + rs2))$. *load byte (8bits)*

- **cv.lbu rD, rs2(rs1)** $\rightarrow rD = Zext(Mem8(rs1 + rs2))$. *load byte unsigned (8bits)*
- **cv.lh rD, rs2(rs1)** $\rightarrow rD = Sext(Mem16(rs1 + rs2))$. *load half-word (16bits)*
- **cv.lhu rD, rs2(rs1)** $\rightarrow rD = Zext(Mem16(rs1 + rs2))$. *load half-word unsigned (16bits)*
- **cv.lw rD, rs2(rs1)** $\rightarrow rD = rD = Mem32(rs1 + rs2))$. *load word (32bits)*

- **Register/Mem to Mem/Register Store:**
  - **cv.sb rs2, rs3(rs1)** $\rightarrow Mem8(rs1 + rs3) = rs2$. *load byte (8bits)*
  - **cv.sh rs2, rs3(rs1)** $\rightarrow Mem16(rs1 + rs3) = rs2$. *load half-word (16bits)*
  - **cv.sw rs2, rs3(rs1)** $\rightarrow Mem32(rs1 + rs3) = rs2$. *load word (32bits)*

Here, rD is the destination Reg, rs2 is the offset value of rs2 in load operation(register adjacent to source register) and in store operation rs3 is offset, and rs1 is the source register. So in a 32 bit instruction, one can find opcode in 06 : 00, rD in 11 : 07, func3 (operation to be performed) in 14 : 12, rs1 in 19 : 15, rs2 in 24 : 20, and func7/offset in 31 : 25. For load operations, rd is the destination register, rs1 is the base register and rs2 is the offset. Whereas, for store operations, rd is offset, rs1 is base and rs2 is the source where store operation needs to be performed. Moreover, here, LSU module has to check for misaligned data for different data-types and send the corresponding signal of ack/nack, and perform/check sign extensions for unsigned and signed half-word and byte data-type.

Once, decoded instruction from the ID stage has been fetched, it then generates an address based on the instruction received to perform the necessary read operation from memory so that it can pass fetched data onto EX stage to perform various operations like ADD, SUB, MULT, DIV, XOR, SHIFT, etc, on the data based on the opcode obtained from the decoded instruction. Later, it receives the executed output on the data based on the opcode specified operation, and calculates the address for performing the store operation if recieved such a signal from ID stage and controller via the OBI interface. All-in-all, LSU module handles signals and control information regarding load/store operations on memory, which is provided by the ID stage and then waits for the controller to show green signal to perform necessary read/write operations on the memory address.

4 Verification Results

The following modules have been tried for verification:

IF Stage Module - The IF Stage module is an outer module that contains many submodules. As per the plan, the first idea was to verify the coverage for the main inputs in the specification and how the outputs in the specification are effected by this:

- Inputs: instr_gnt_i, instr_rvalid_i and instr_rdata_i[31:0]
- Outputs: instr_req_o and instr_addr_o[31:0]

The coverages for these can definitely be improved. At the submodule level, IF stage contains aligner and compressed_decoder.

- Compressed Decoder: This has only 1 input, namely [31:0] instr_i. We trace back how we receive the 3 outputs using our given input and write coverages accordingly. 100% coverage has been performed for the same and the three outputs have been checked for.

- Aligner: This module has 7 Coverpoints for the 7 inputs and 4 corresponding outputs. The methodology followed here was to check how each of the outputs are produced and what inputs are used to produce them. Correspondingly, a scoreboard was created.

ID stage module - The ID stage module has been checked for the following instructions for the decoder, controller and Register File signals-

1. RISCV ISA ALU Instructions (Add, Sub, SLTS, SLTU, XOR, OR, And, SLL, SRL, SRA)
2. RISCV ISA Memory Access Instructions (SB, SH, SW, LB, LBU, LH, LHU, LW, LUI)
3. AUIPC
4. OPIMM
5. RISCV M Instructions (MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU)
6. PULP Instructions (cv.lb, cv.lbu, cv.lh, cv.lhu, cv.lw, cv.beqimm, cv.bneimm)

Register File (Flipflop) module - This module has been checked in the following manner - Data A is written at address B in the register file, then the data is read from address B and verified if the data is the same as A. This verifies both the Read as well as the Write Operations in the Register File.
We are also checking that if the data was meant to be written at address B, it should not have been written at any of the adjacent registers (B-2, B-1, B+1 and B+2 addresses) by performing the read operation on them.

Hardware Loop Register module - This module has been checked in the following manner - The functionality of the *cv32e40p_hwloop_regs.sv* module has been replicated in the Scoreboard and the Output Start and End address registers are being checked if they have the correct data at the correct location id which has been given as the input.

ALU Module - This is the block that implements Execute Stage's core functionality. Since it is entirely combinatorial in nature, its verification has been done in the following manner :

- A bottom-to-top approach was followed in verifying the ALU module. The module consisted of 2 sub-modules: *cv32e40p_ff_one* *cv32e40p_popcnt*, and several instructions directly implemented in the main ALU module *cv32e40p_alu*
- The two sub-modules are individually verified for correctness
- The main ALU is checked for against the following operations :
    1. ALU_ABS
    2. ALU_SLETS
    3. ALU_SLETU
    4. ALU_MIN
    5. ALU_MINU
    6. ALU_MAX
    7. ALU_MAXU
    8. ALU_XOR

9. ALU_OR
10. ALU_AND

LSU Module

Verification of LSU module was done in such a manner that functionality of Address generation, misaligned data in the instruction memory, identification of data type, sign extension for identified data-types, and verification of The OBI protocol that is used by the LSU to communicate with a memory along with verification LSU interface signals, has been reproduced in the scoreboard of the layered CDV test-bench. So the following have been covered in the verification of LSU module:

- Verification of all the LSU interface signals mentioned in the document as well as the module interconnections with ports of other stages' modules(i.e., OBI interface, $ID - stage$ module and controller module) has been performed in the CDV of LSU module.
- Verification of Basic Memory Transaction, Back-to-back Memory Transactions,and Multiple Outstanding Memory Transactions ($2ACC.totheprocessorspecification$) has been performed in accordance with the OBI protocol that is used by the LSU to communicate with a memory.
- Verification of Misaligned memory access cases which should be identified and handled by the LSU module has been covered in the CDV of the LSU module.
- Verification of valid address generation for read and write (load/store operation) purposes has also been verified.
- Verification of the signals containing Post-Incrementing Load and Store Instructions has been performed in CDV of LSU module.

Coverage Reports

Coverage Report Screenshots for some modules:

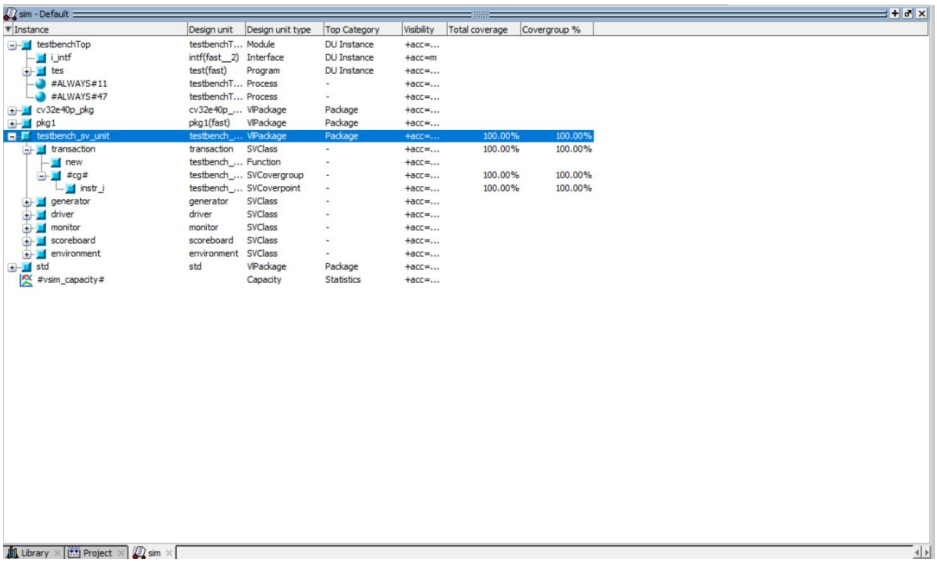

Figure 5: Coverage Report for the Verification of the Compressed Decoder module

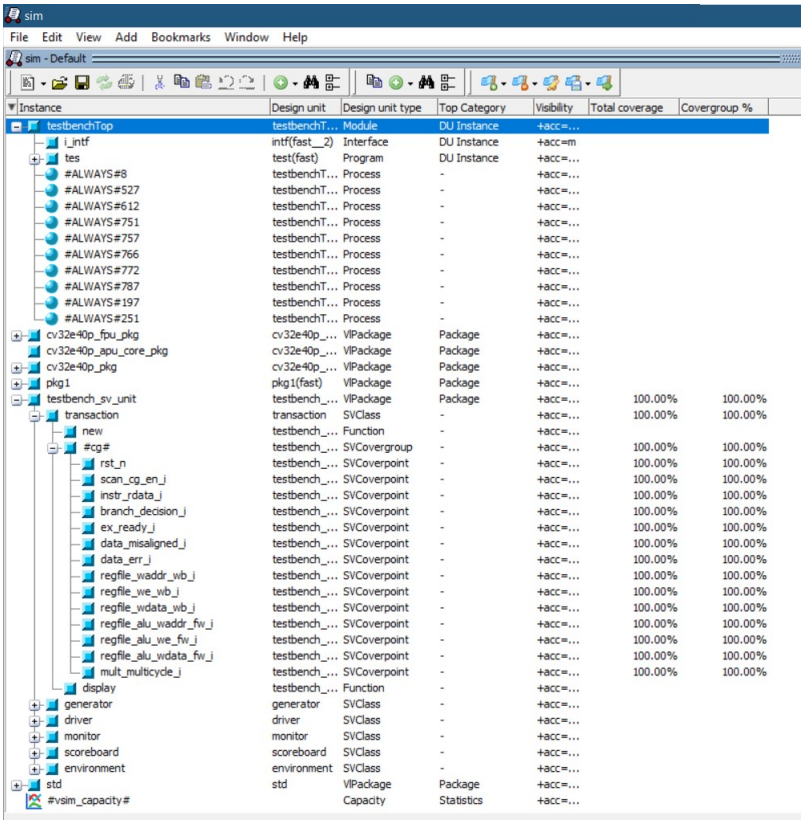

Figure 6: Coverage Report for the Verification of the ID Stage module

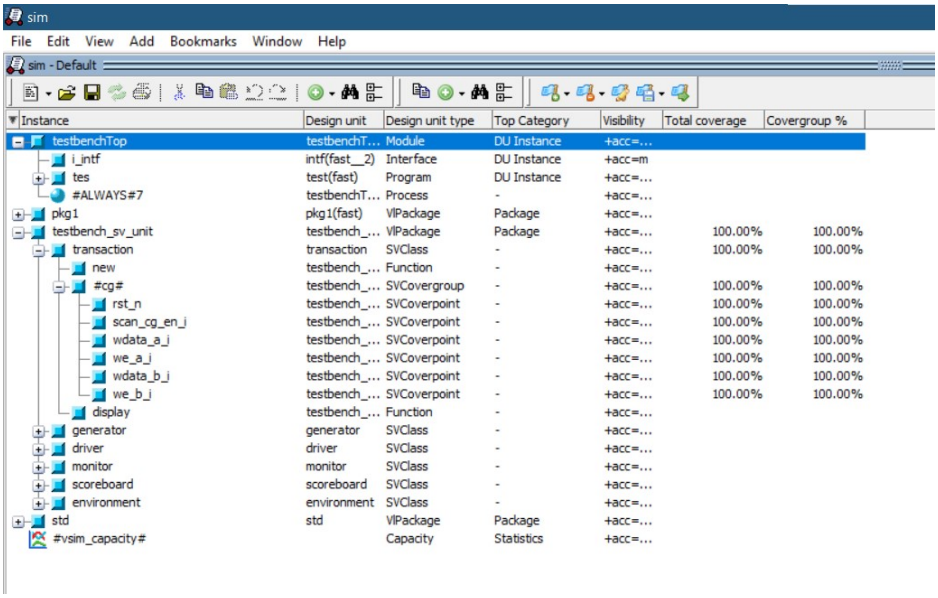

Figure 7: Coverage Report for the Verification of the Register file Flipflop module

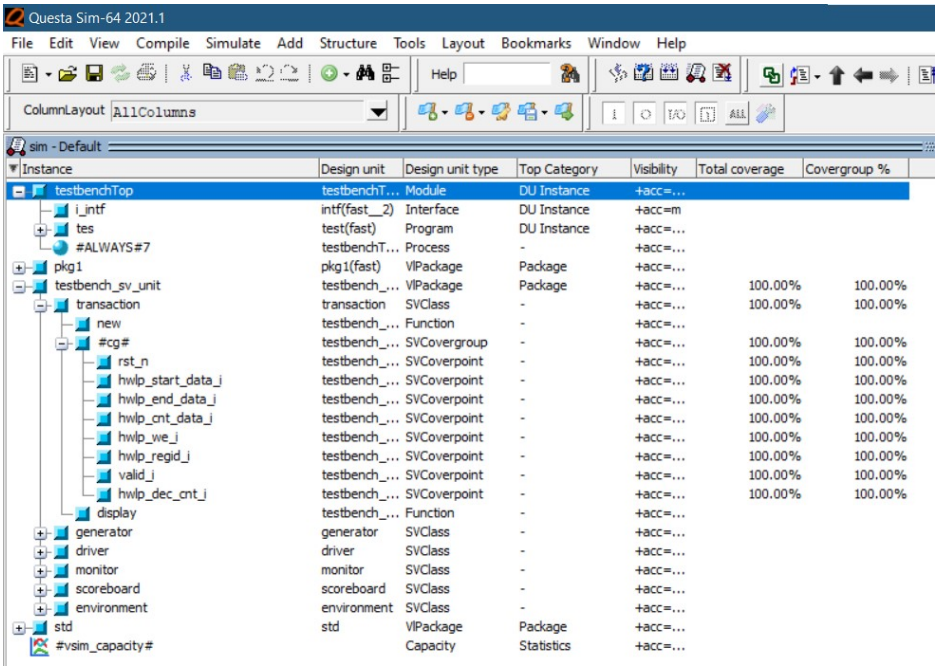

Figure 8: Coverage Report for the Verification of the Hardware Loop Register module

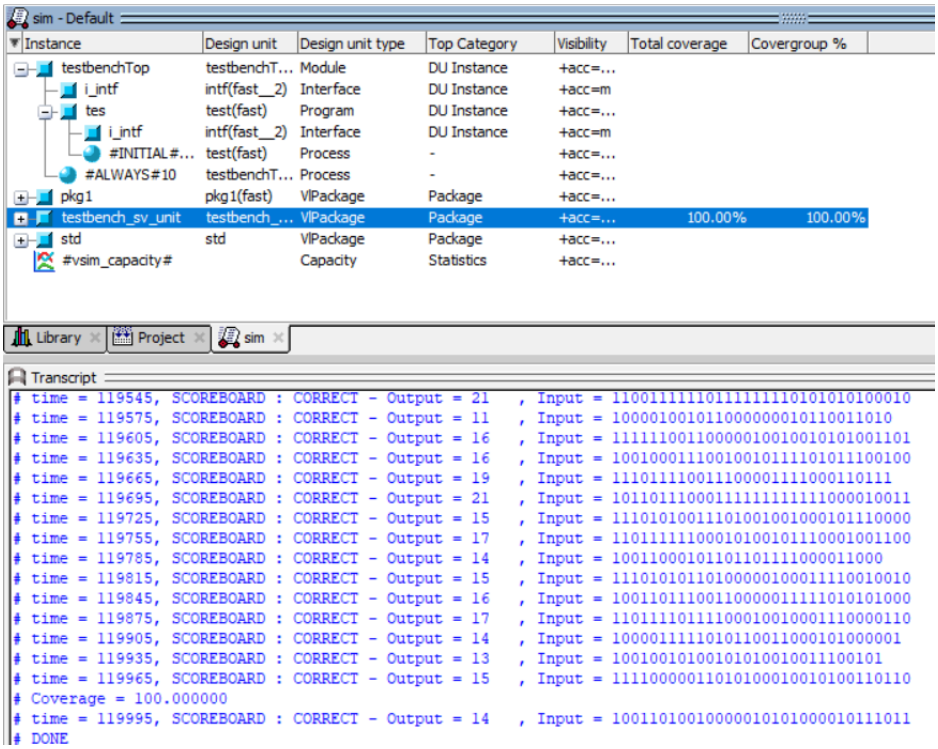

Figure 9: Coverage Report for the Verification of the popcnt module

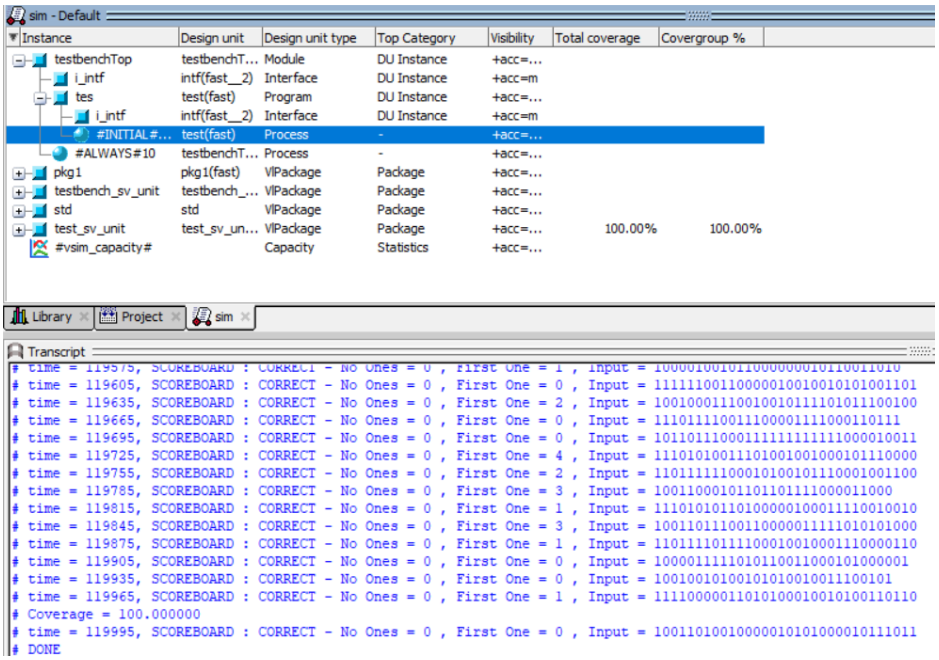

Figure 10: Coverage Report for the Verification of the ff_one module

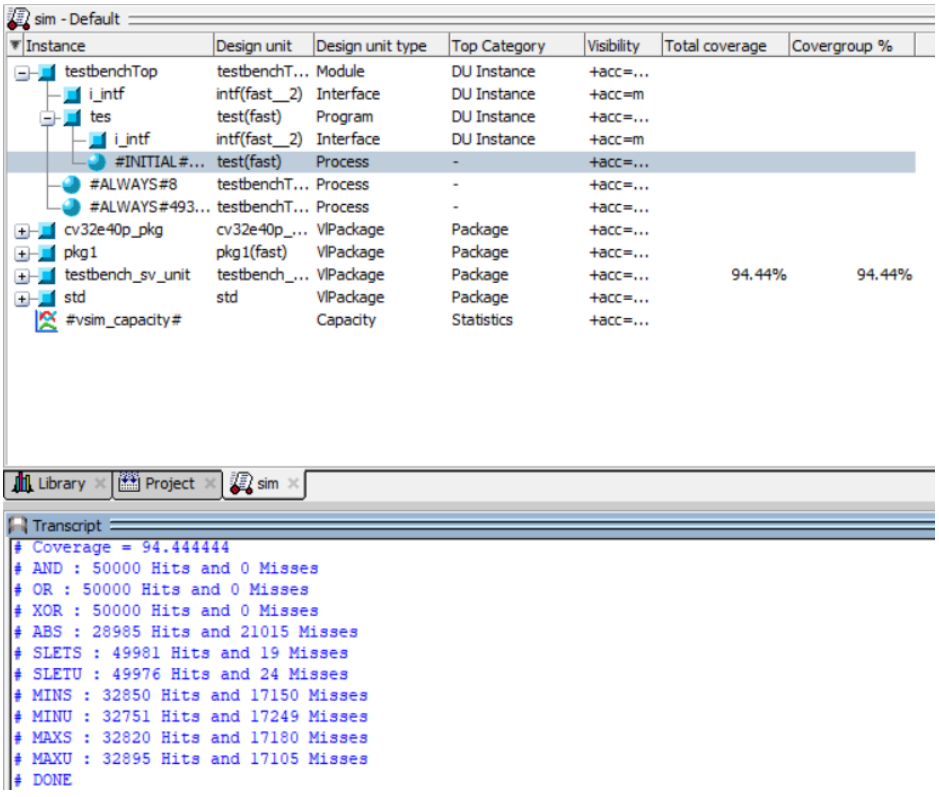

Figure 11: Coverage Report for the Verification of the alu module

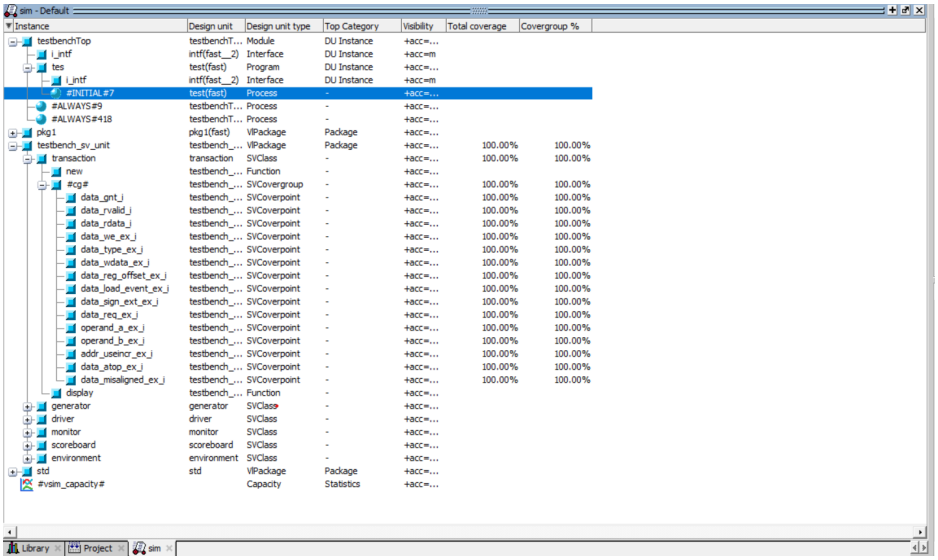

Figure 12: Coverage Report for the Verification of the Load/Store Unit (LSU) module

5 Github link for the CDV codes

The CDV codes for the modules verified have been added to this Github Repository.

6 Challenges Faced

1. Understanding the huge code base was very difficult due to the large number of interconnected modules and lack of documentation explaining the working of the modules.
2. Backtracing the signals from the modules to other modules was very tedious and time-consuming due to the huge number of signals and functionalities, but this was required in order to understand the working of the modules so that their functionalities could be checked through their respective Scoreboards.

7 Work Distribution

- **Arhant** - Worked on CDV of IF stage and its sub modules like prefetch_buffer, compressed_decoder, aligner.
- **Asmita** - Worked on CDV of ID stage module (includes the Decoder and the Controller), Register File module and the Hardware Loop Registers module.
- **Saket** - Worked on the structure of layered testbench and the CDV of ALU
- **Yash** - Worked on CDV of Load/Store Unit (LSU) module and its interacting modules of ID/EX stages and WB stage of the core.

8 Future Works

- Verify other instruction extensions (F and other PULP instructions) and the CSR modules.
- Verify the top module (core) that connects all the stages to ensure the correct working of the entire processor.
- Verify the processor to ensure that it works for instruction blocks having data hazards.
- Increase coverage of modules with lesser coverage numbers.

9 Conclusion

In conclusion, the complexity of the CV32E40P RISCV Processor, with its extensive codebase, numerous modules, and intricate inter-module connections, poses significant challenges in comprehending its operation. While verification efforts have been made for certain modules, understanding and verifying several other modules proved to be difficult. This highlights the need for continued exploration and analysis to enhance our understanding of the processor's functionality and to further advance the verification process.

10 References

1. CV32E40P Github Repository
2. CV32E40P Documentation
3. Layered Testbench Reading Material
4. Layered Testbench YT Resource