Implementation and Functional Verification of RISC-V

By

Abdelrahman Mohamed Adel

Dina Saad Mohamed

Mahmoud Abd El Mawgoed Ibrahim

Mohamed Alaa Sharshar

Zyad Ahmed Aboelkasem

A Thesis Submitted to the
Faculty of Engineering at Cairo University

In Partial Fulfillment of the

Requirements for the Degree of

BACHELOR OF SCIENCE

In

Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

July 2021

ACKNOWLEDGEMENTS

We would like to thank our supervisors: Dr. Hassan Mostafa, for his great support, supervision and encouragement. Moreover, we are very grateful to Dr. Eman El-Mandouh, Eng. Hala Ibrahim for their valuable technical support, patient guidance, enthusiastic encouragement and useful critiques throughout the project's phases. Special thanks to our Fund resources: ONE Lab. (Cairo University) and Mentor Graphics (Mentor Siemens) Egypt.

Abstract

In the world of technology, we’re living in, and the huge increase in the number of IoT (Internet of things) devices leading to a tremendous amount of data being sent mostly using wireless low power technologies such as ZigBee and Bluetooth low energy (BLE), this wireless data is prone to eavesdropping and being hacked and, thus ensuring the security of the data being sent wirelessly in IoT applications is of the utmost importance. Moreover, designs for security chips or ICs need to be verified before fabrication or implementation on FPGAs. Verification has always been an ever-increasing challenge. The gap between what a verification plan can offer nowadays, and the current technology requirements is constantly widened. Verification can be static or dynamic, this work demonstrates the power of static verification using Questa tools as well as the power of dynamic verification using UVM (Universal Verification Methodology). The Universal Verification Methodology has come in action as a literal savior to the whole verification community, by offering a merge between System Verilog and System C into one environment that is completely standardized, constrained, and reusable, allowing a powerful verification methodology to a wide range of design sizes and types. The main contribution that this work introduces for IoT is designing a system on chip based on a processing core designed using RISC-V ISA, the SoC encrypts and decrypts data for IoT devices, and the main contribution for verification is developing an efficient UVM environment for our RISC-V ISA based core and covering all of its corner cases, thus making the design ready for implementation.

.

Table of Contents

List of figures

Chapter 1: Introduction

1.1 Problem Definition

The number of IoT devices is increasing significantly nowadays (almost 22 billion in 2018), and they are used in a lot of fields. Most of the data is being sent wirelessly, making it vulnerable to eavesdropping and hacking, some of these data are really sensitive and confidential (military data), so these data need to be encrypted and checked to provide integrity and security. In this project, we propose a solution which is installing Hardware security modules on the IOT end nodes, where a dedicated System on Chip – with a processor based on RISC-V ISA - has a crypto accelerator that implements a light cryptography algorithm (ACORN algorithm) with a reasonable security level, thus saving power and achieving security simultaneously. Where the end node sends the data to the chip via UART, then after being Encrypted/Decrypted it's sent back to the end node where it's then sent to the server via the gateway as shown in figure 1. SoCs in general are required to having multiple clock domains, so as not to limit the frequency of the whole SoC to the slowest module, in our design, we chose to have three clock/Reset domains. This choice is a pro and a con at the same time, where a lot of issues and challenges will arise due to CDC and RDC which will be explained later, on the other hand we will have a much faster system not limited to a single frequency.

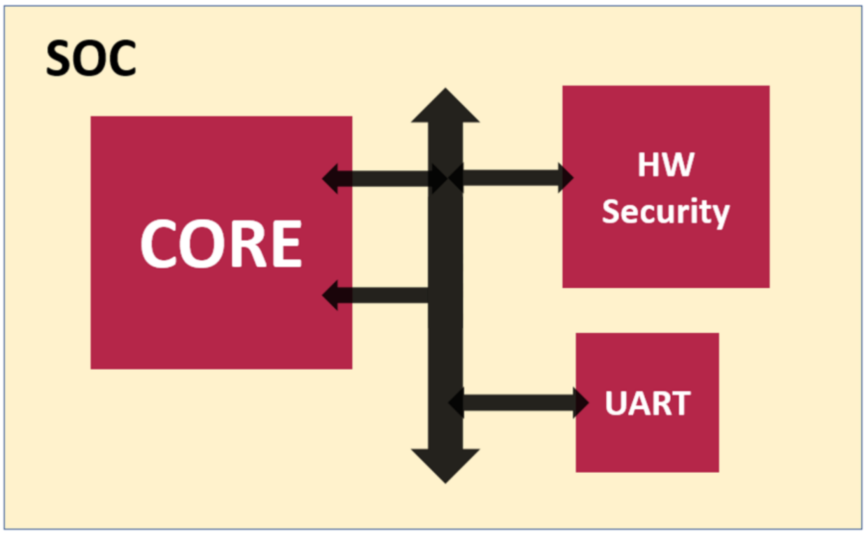


Figure 1: SoC conceptual design

When comparing Hardware security methods like the method we are using to software security, hardware security is much better. Attacks in hardware security needs physical access, so it will be difficult to gain access to the hardware since the system will trigger an alarm or delete the critical data if it detects a threat. It uses dedicated encryption hardware, which makes it faster to execute as it lifts the burden off the system hardware itself and makes it safer as it becomes difficult to crack. On the other hand, software security use shared hardware as it runs on general-purpose processors making it slower to execute. Hardware based security is supported by a dedicated software, so it doesn't depend on the operating system unlike software-based security that depends on the operating system.

Moreover, software security is vulnerable to software bugs or code logical errors which can be exploited by hackers. Encryption is used to secure confidential information by carrying out effective algorithmic schemes based on complex cryptographic mathematics. The objective of conducting data encryption is developing a cryptographic system that achieves confidentiality, authentication, and data integrity.

The other problem targeted in our project developing an efficient universal verification methodology (UVM) environment for our RISC-V based microprocessor. Digital designers spend almost 50% of their time verifying their designs, so not having a UVM environment for efficiently testing the core and covering all of its corner cases increases the time to market and the design may have errors that would lead to a huge loss after the ASIC tape out.

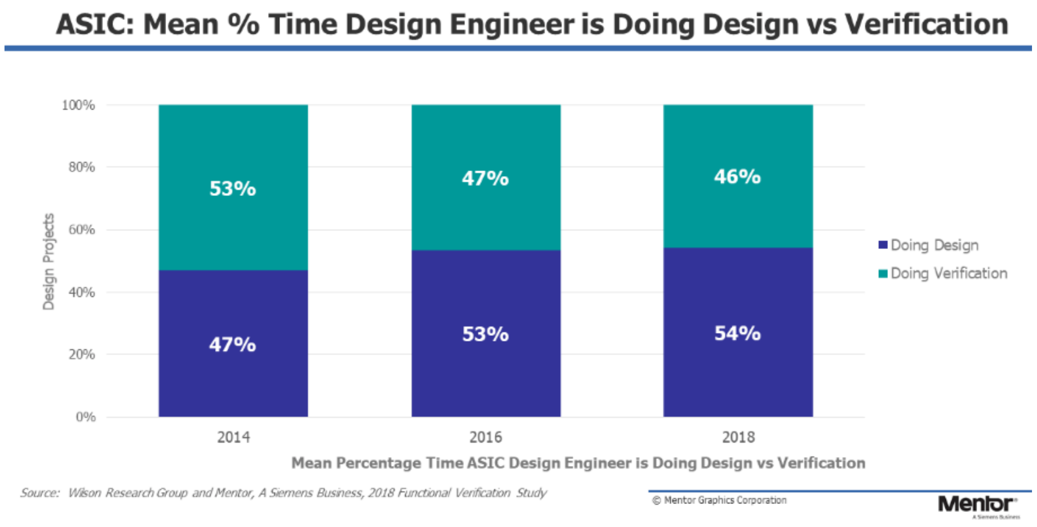


Figure 2: Design Vs Verification time

1.2 What is RISC-V?

RISC-V (pronounced “risk-five") is an open-source free ISA (**I**nstruction **S**et **A**rchitecture) based on RISC principles enabling a new era of processor innovation through open standard collaboration. Born in academia and research, RISC-V ISA delivers a new level of free, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation. An ISA is a group of codes for a computer architecture in machine language which describes the way in which the software talks to the underlying processor. The RISC-V ISA is designed for a wide range of uses. It aims to be a standard ISA for all computing devices, from simple IOT device to complex supercomputers.

1.3 History

It's all started at University of California Berkley in 2010 by a group of academics as an alternative for ARM and x86 architecture. They made it free and open source and there are many variants of RISC-V, for educational uses and for advanced computing. This is the fifth generation; the first two generations influenced the SPARC processor. RISC-V foundation was launched in late 2015 to control the RISC-V activities and the ISA specification itself was published in 2011.

1.4 Why RISC-V?

- **Open source** and globally supported.
- **Simple**: RISC-V is far smaller than other commercial ISAs.
- **Stable**: Base and first standard extensions are already frozen. There is no need to worry about major updates.
- **Extensibility**: Specific functions can be added on the base ISA based on extensions.
- **Cost:** Due to extensibility and simplicity the die size shrinks which leads to lower cost.
- **Modular:** RISC-V has a small standard base ISA (RV32I/E) with multiple standard extensions (depending on the needs of the application). This

modularity enables very small and low energy implementations of RISC-V, which can be critical for embedded applications.

- The conventional approach is incremental ISAs where new processors must implement not only new ISA extensions but also all extensions of the past. The purpose is to maintain backwards binary-compatibility so that binary versions of decades-old programs can still run correctly on the latest processor. As an analogy, suppose a restaurant serves only a fixed-price meal, which starts out as a small dinner of just a hamburger and a milkshake. Over time, it adds fries, and then an ice
  cream sundae, followed by salad, pie, wine, vegetarian pasta, steak, beer, ad infinitum until it becomes a gigantic banquet. It may make little sense in total, but diners can find whatever they've ever eaten in a past meal at that restaurant. The bad news is that diners must pay the rising cost of the expanding banquet for each dinner. RISC-V offers a menu instead of a buffet; the chef need cook only what the customer wants, and the customer pay only for what they order.

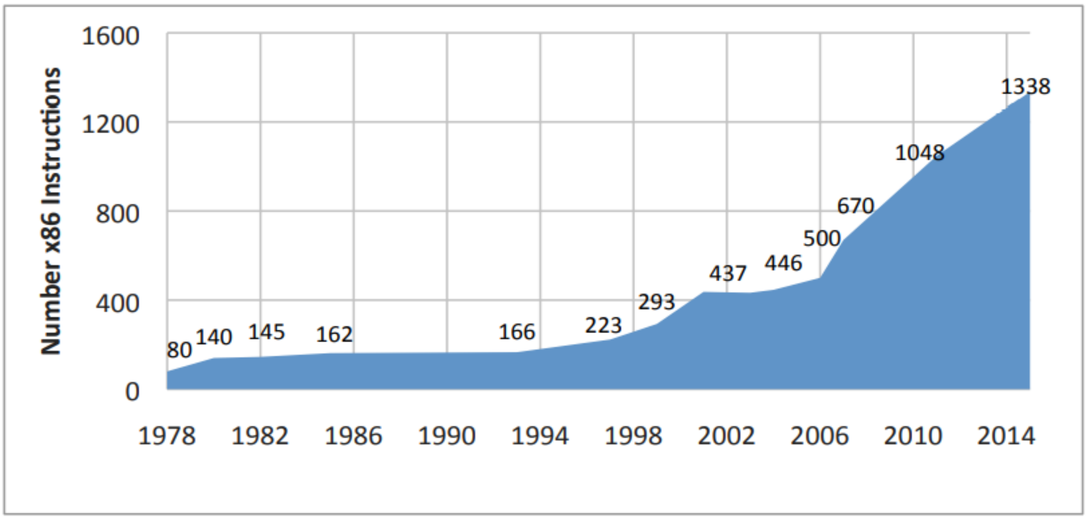


Figure 3: Incremental ISAs of x86 processor

1.5 RISC-V ISA Overview

The base integer ISAs are very similar to the RISC processors except with no branch delay slots and with support for optional variable length instruction encodings.

- **Base ISAs:**
    1. RV32I
    2. RV64I
    3. RV32E subset variant of the RV32I base (support small microcontrollers).
    4. RV128I (future work).

Each base ISA can be extended with one or more optional instruction set extensions to support more general software development. The base integer ISA (“I” extension) contains integer computations, integer loads, integer stores and control flow signals.

- **Standard extensions:**
    1. “M”: Integer multiply and divide extension.
    2. “A”: Atomic memory operations (AMO) (read, modify, and write memory).
    3. “F”: Single-precision floating-point extension.
    4. “D”: Double-precision floating-point extension.
    5. “C”: Compressed instruction extension provides narrower 16-bit forms of common instructions.
    6. “G”: (“IMAFD”) General-purpose ISA which contains all the above extension.

Comparing the RISC-V base ISA with other ISAs, First, there are only six formats in the base ISA and all instructions are 32 bits long, which simplifies instruction decoding. ARM-32 and particularly x86-32 have numerous formats, which make decoding expensive in low-end implementations and a performance challenge for medium and high-end processor designs. Second, RISC-V instructions offer three register operands, rather than having one field shared for source and destination, as with x86-32. When an operation naturally has three distinct operands, but the ISA provides only a two-operand instruction, the compiler or assembly language programmer must use an extra move instruction to preserve the destination operand. Third, in RISC-V the specifiers of the registers to be read and written are always in the same location in all instructions, which means the register accesses can begin before decoding the instruction. Many other ISAs reuse a field as a source in some instructions and as a destination in others (e.g., ARM-32 and MIPS-32), which forces addition of extra hardware to be placed in a potentially time-critical path to select the proper field. Fourth, the immediate fields in these formats are always sign extended, and the sign bit is always in the most significant bit of the instruction. This decision means sign extension of the immediate, which may also be on a critical timing path, can proceed before decoding the instruction.

1.6 Core vs SoC

A core (CPU) is simply a computation engine. It fetches data from memory, and then performs some kind of arithmetic (add, multiply) or logical (and, or, not) operation on that data. The more expensive/complex the CPU, the more data it can process, the faster your chip. A CPU (core) itself is not a standalone system. There must be many other components integrated with the core to form a good and complex application. If we took a personal computer as an example, there should be memory to hold the data, an audio chip to decode and amplify your music, a graphics processor to draw pictures on your monitor, and hundreds of smaller components (beside your core) that all have a very important task. A SoC, or system-on-a-chip to give its full name, integrates almost all these components into a single silicon chip. Along with a CPU (core), a SoC usually contains a GPU (a graphics processor), memory, USB controller, power management circuits, and wireless radios (WiFi, 3G, 4G LTE, and so on). Whereas a CPU cannot function without dozens of other chips, it’s not possible to build complete computers with just a single core. A SoC has some advantages like having much shorter wiring, high level integration, much smaller size, less power and much cheaper.

1.7 Different platforms, SoCs and cores based on RISC-V ISA

- **Platforms**
  - **SiFive**
    - E-Cores
    - S-Cores
    - U-cores
  - **PULP**
    - PULPino – SOC
    - PULPissimo -SOC
      - Riscy - Cores
      - Ariane - Cores
      - Ibex - Cores

1.7.1 SiFive platform

SiFive's founders are the same UC Berkeley professor and PhDs who invented and have been developing the RISC-V Instruction Set Architecture (ISA) since 2010. In 2016, SiFive released the freedom everywhere for about 310 SoCs and the HiFive development board, making SiFive the first company to produce a chip that implements the RISC-V ISA. SiFive founders took the designing stage of your core to another level where they made this stage completely customizable according to the customer's choice. On their site, you can choose the extensions you need, number of pipeline stages, cache size, power modes and the number of cores you need.

As we mentioned before SiFive cores are:

1- **E-cores**: 32-bit embedded cores used for MCUs, AI, IoT and edge computing.

2- **S-cores**: 64-bit embedded cores used for storage, AR/VR and machine learning.

3- **U-cores**: 64-bit application processors used for Linux, datacenter, and network baseband.

1.7.2 PULP platform

It stands for **P**arallel **U**ltra **L**ow **P**ower targeted for high energy efficiency. They have implemented a set of efficient RISC-V based cores such as:

1- 32-bit 4 stage core **CV32E40P** (formerly **RI5CY**)
2- 64-bit 6 stage core **CVA6** (formerly **Ariane**)
3- 32-bit 2 stage core **Ibex** (formerly **Zero-Riscy**)

They also have implemented some SoCs such as:

1- Single core microcontrollers (**PULPissimo**,**PULPino**)
2- Multi-core IoT processors (**OpenPULP**)
3- Multi-cluster heterogeneous accelerators (**Hero**)

- **PULPino**

A minimal single-core RISC-V SoC, the first open-source release that has attracted a lot of attention (the small pulp)

- **PULPissimo**

It's an advanced version of PULPino SoC. It contains a uDMA that can copy data directly between peripherals and memory, as well as optional accelerators that we call Hardware Processing Engines (HWPEs).
Both PULPino and PULPissimo SOCs target high energy efficiency and ultra-low power and used in IoT applications.

- **RI5CY**

A Single-issue core with 4 pipeline stages and it has an IPC close to 1.

It supports:

  - Base integer instruction set (RV32I)
  - Compressed instructions (RV32C)
  - Multiplication instruction set extension (RV32M)
  - It can be configured to have single-precision floating-point instruction set extension (RV32F)

- **Ibex**

A single-issue core that has been designed to target ultra-low-power and ultra-low-area constraints with 2 pipeline stages.

It supports:

  - Base integer instruction set (RV32I)
  - Compressed instructions (RV32C).
  - It can be configured to support the multiplication instruction set extension (RV32M) and the reduced number of registers extension (RV32E).

Chapter 2: Designing the Unpipelined 32 IM RISC-V core

2.1 RV32I (RISC-V Base Integer ISA)

Below listed are the six base instruction formats:

- **R-type** for register-register operations
- **I-type** for short immediate and loads
- **S-type** for stores
- **B-type** for conditional branches
- **U-type** for long immediate
- **J-type** for unconditional jumps

Figure 4 lists the opcodes of the RV32I instructions.

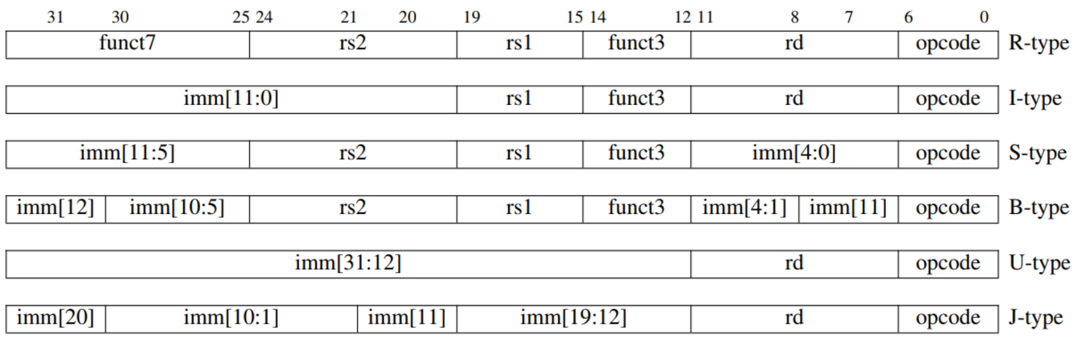
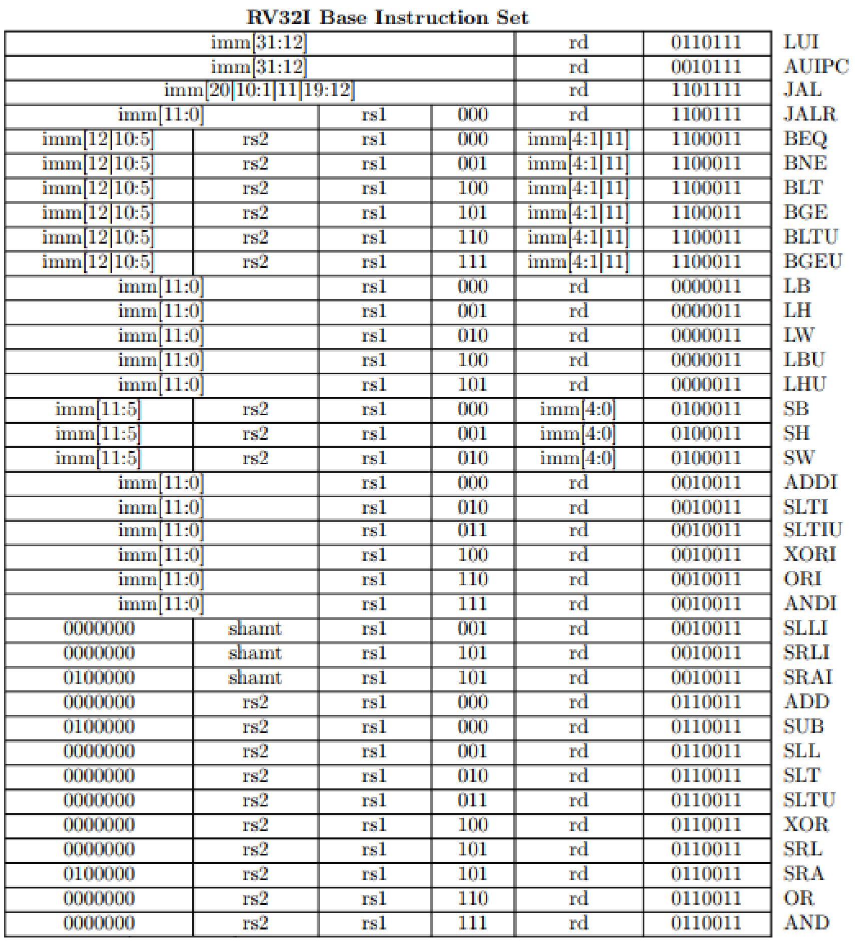

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 4: RV32I opcodes


RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 5: RV 32I instructions

In RISC-V the specifiers of the registers to be read and written are always in the same location in all instructions, which means the register accesses can begin before decoding the instruction. Many other ISAs reuse a field as a source in some instructions and as a destination in others which forces addition of extra hardware. The sign bit is always in the most significant bit of the instruction. This decision means sign extension of the immediate, which may also be on a critical timing path, can proceed before decoding the instruction.

2.1.1 The register file

The register file is required for the processor to load its operands from and to store the result in it with a much faster access than the data memory, being the result of an arithmetic operation or a memory load operation or any operation that needs to store an output. The register file consists of 32 registers (locations) thus much faster than the RAM, each of a width of 32-bits, register of location zero is hardwired to zero to speed up operations that need a zero operand, dedicating a register to zero is a surprisingly large factor in simplifying RISC-V ISA.

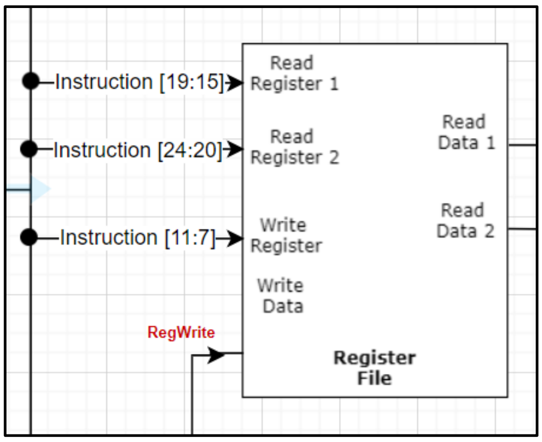
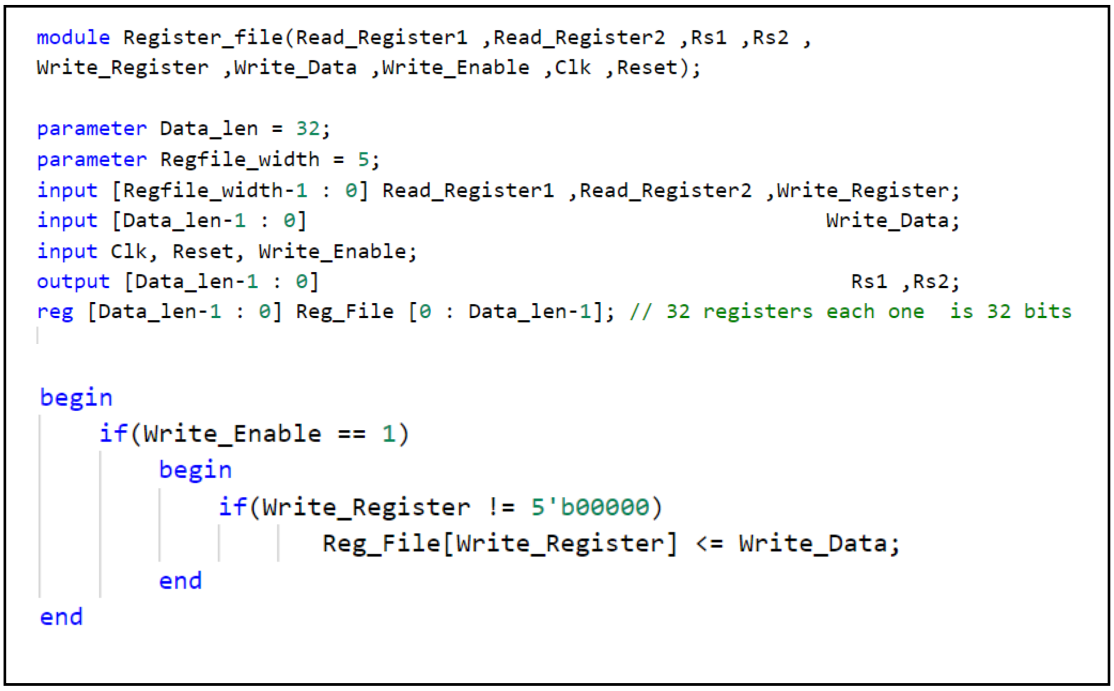


Figure 6: Register file



```
module Register_file(Read_Register1 ,Read_Register2 ,Rs1 ,Rs2 ,
Write_Register ,Write_Data ,Write_Enable ,Clk ,Reset);

parameter Data_len = 32;
parameter Regfile_width = 5;
input [Regfile_width-1 : 0] Read_Register1 ,Read_Register2 ,Write_Register;
input [Data_len-1 : 0]                                        Write_Data;
input Clk, Reset, Write_Enable;
output [Data_len-1 : 0]                                         Rs1 ,Rs2;
reg [Data_len-1 : 0] Reg_File [0 : Data_len-1]; // 32 registers each one  is 32 bits


begin
    if(Write_Enable == 1)
        begin
            if(Write_Register != 5'b00000)
                Reg_File[Write_Register] <= Write_Data;
        end
end
```

2.1.2 Instruction memory

The instruction memory is the memory holding the instructions to be executed by the processor. The instruction memory has a single read port, from which we read the instruction. The instruction memory is of size 256 KB thus require 18-bit address lines. The address of the next instruction to be executed is stored in the program counter register. Each instruction is a fixed 32 bits in length and must be aligned on a four-byte boundary in memory, so the program counter is incremented by 4 each time we load a new instruction.

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on, for simplicity we will reset the program counter to 0.

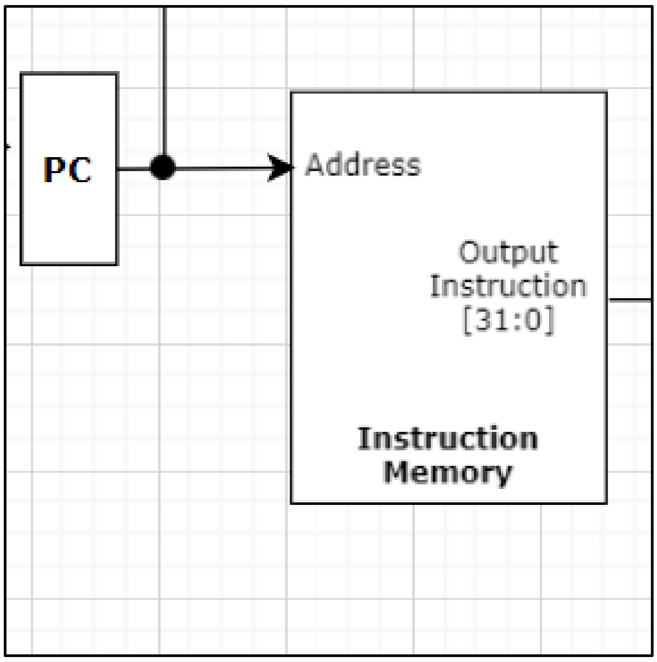


Figure 7: Instruction memory

2.1.3 Data memory

The presence of data memory in addition to the instruction memory is due to following Harvard architecture and eliminating any structural hazards that may arise as will be explained later in the section 3.

The data memory (RAM) is used to store the temporary data used by the processor during its operation, it's a random-access memory with a size of 256 KB, and thus needs 18 address lines ($2^{18} = 256\ K$). The two main instructions for the memory are the load and store instructions with all their derivatives. To perform the store instructions, a MemWrite signal is outputted from the control unit to the data memory, the MemWrite signal determines whether we are storing a byte, half word, or a full word. In order to perform load instructions, the memory would always load a full word and a memory circuit was designed to choose whether we are loading a byte, half word or a full word as shown, where Id_ext and WHB (Word, Halfword, Byte) are signals from the control unit.

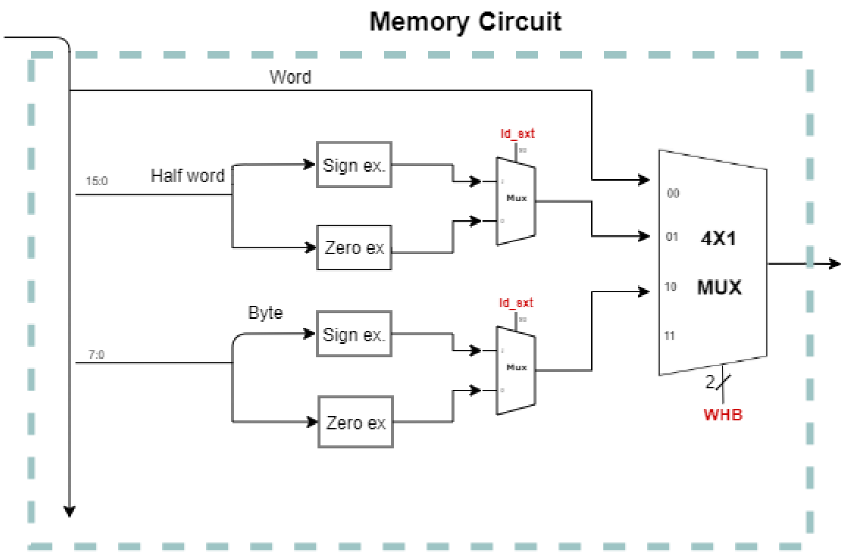


Figure 8: Memory circuit

To use memories in our design, BRAM IPs were instantiated, to prevent the synthesis tool from using up all the registers present in the FPGA's slices. The BRAM however needs 2 clock cycles for correct operation.

So, to interface with our memories, a wrapper was instantiated on top of the instruction and data memory. As 256 KB is very large memory to be synthesized so we must use BRAM inside targeted FPGA (virtex-7 vc707), and VIVADO IP to generate block memory. VIVADO IP memory have different access technique than our memory as VIVADO IP cannot be accessed to get word from 2 different addresses, so we must reconfigure it.

One solution of that issue is to use dual port memory for reading from memory and always assign the first address to get the first word and second address to first address + 1 to get the following word and use wrapper to concatenate the required word from the two accessed words depend on the instruction used (word accessing / half word accessing / byte accessing). Note that this issue will not appear in case we are accessing byte.

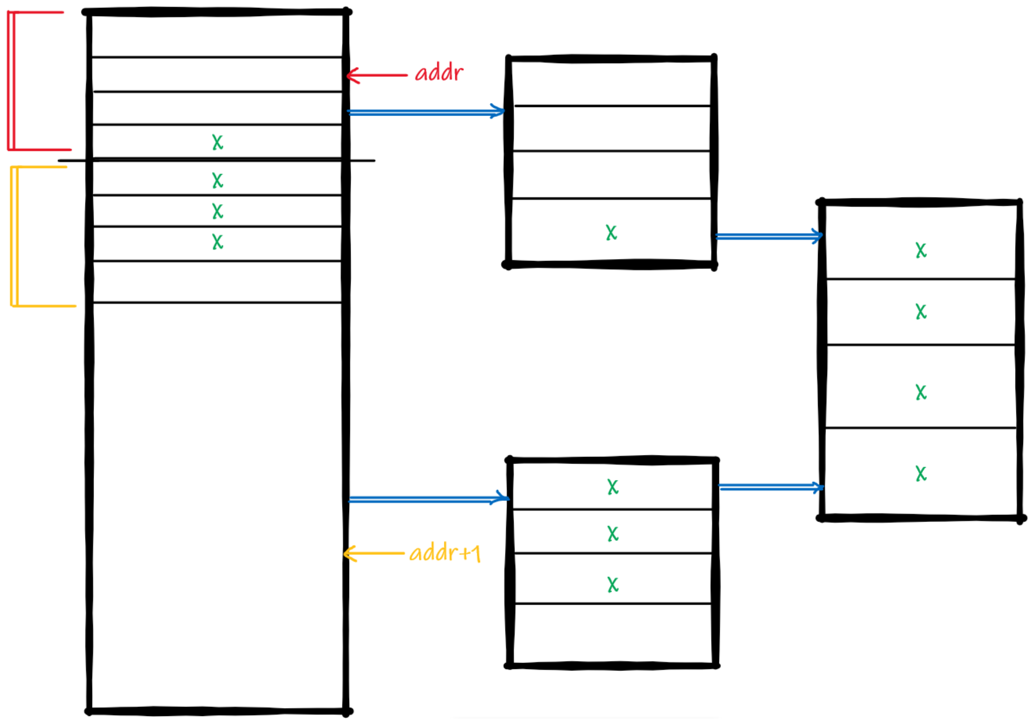


Figure 9: Memory wrapper

2.1.4 ALU

The ALU is basically the muscles of the processor, where any Arithmetic or logic operation of an instruction gets executed in the ALU. An ALUOP signal is outputted from the control unit and given to an ALU controller along with bits 30 and 14 to 12 in the instruction (called ALU_control_in), the ALU controller then outputs an ALU_select signal that determines which operation the ALU will execute on its operands as shown in figure 10.

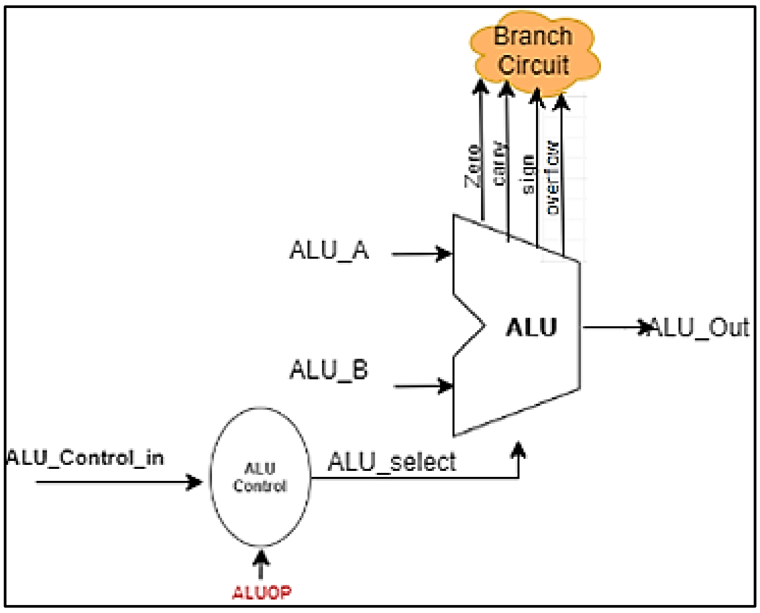


Figure 10: ALU

```
begin
    case(ALU_select)
        3'b000: out_reg        = ALU_A+ALU_B;                      //ADD
        3'b001: out_reg        = ALU_A-ALU_B;                      //SUB
        3'b010: out_reg  = {1'b0, ALU_A << ALU_B[4:0]} ;           //Sll
        3'b011: out_reg  = {1'b0 , ALU_A ^ ALU_B };                //XOR
        3'b100: out_reg  = {1'b0 , ALU_A >> ALU_B[4:0] };          //Srl
        3'b101: out_reg  = {1'b0 , $signed(ALU_A) >>> ALU_B[4:0]} ; //Sra
        3'b110: out_reg  = {1'b0 , ALU_A | ALU_B };                //OR
        3'b111: out_reg  = {1'b0 , ALU_A & ALU_B };                //AND
    endcase
end
```

The ALU outputs the result along with 4 more flags, 3 of which are really important in the branch instructions as will be illustrated, the 4 flags are the carry, zero, overflow and sign flag as shown.

```
assign out      = out_reg[data_len-1:0] ;
assign zero     = (out_reg[data_len-1:0] == 32'b00000000000000000000000000000000) ? 1'b1 : 1'b0;
assign carry    = out_reg[data_len];
assign sign     = out[data_len-1];
assign overflow = carry ^ ALU_A[data_len-1] ^ ALU_B[data_len-1] ^ out[data_len-1];
```

2.1.5 Branch circuit

The Branch circuit serves the B-type instructions that executes the conditional branches. The conditional branches test a given condition and decided upon it whether to branch to a given location in the instruction memory (by loading the program counter) or not. The branch circuit was designed as shown in figure 11.

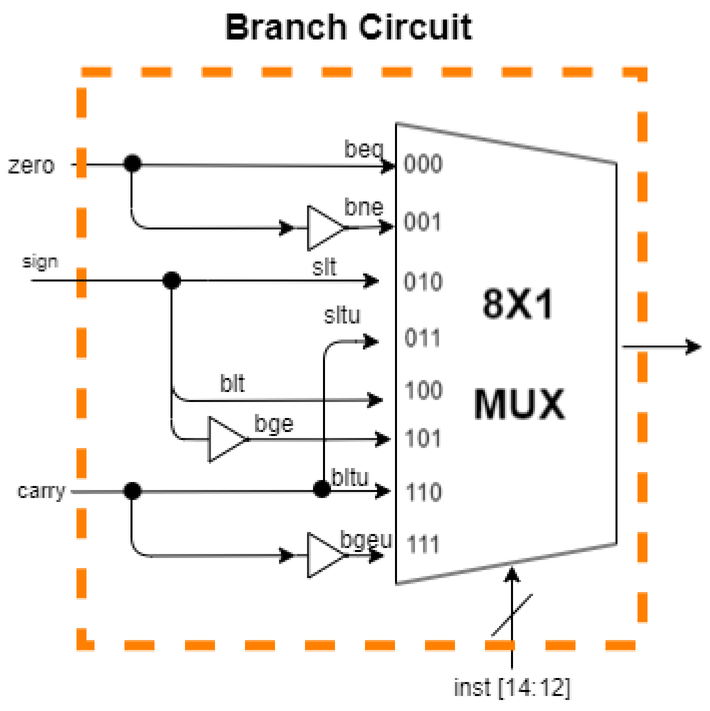


Figure 11: Branch circuit

2.1.6 Immediate generator

For instruction that have an immediate field, some immediate values have to be modified in certain ways for proper execution by the processor. The block that performs these modifications is the immediate generator. The generator takes the instruction as an input and outputs the proper immediate for each type based on the opcode as shown below in figure 12.

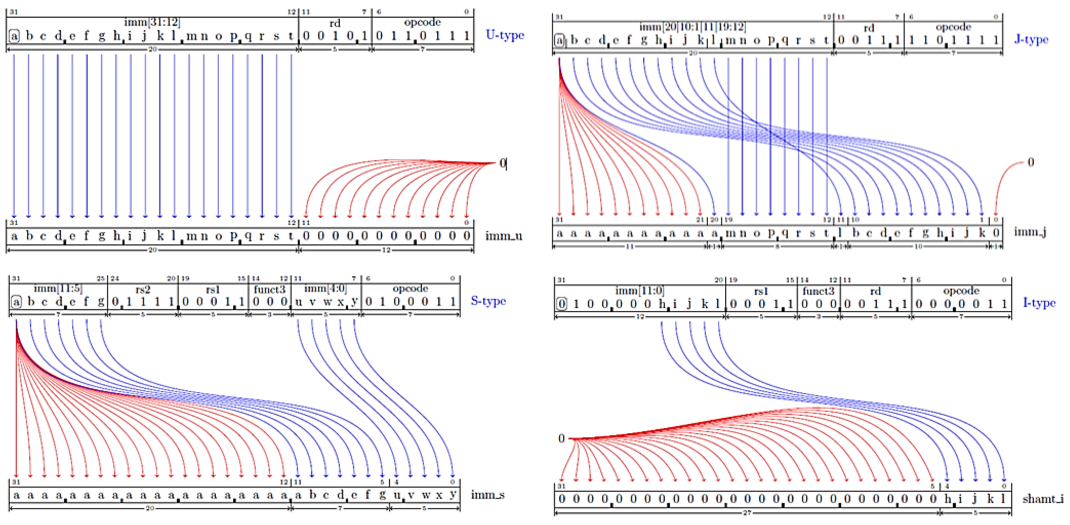
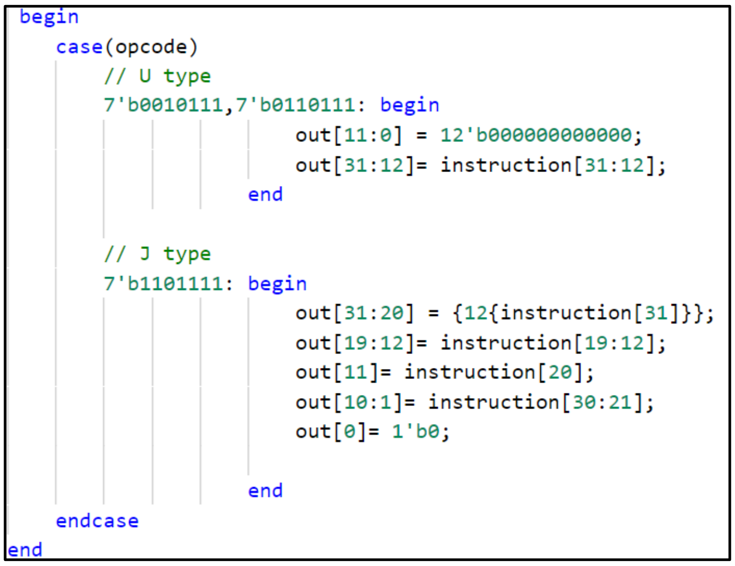


Figure 12: Immediate generator



```
begin
   case(opcode)
      // U type
      7'b0010111,7'b0110111: begin
            out[11:0] = 12'b000000000000;
            out[31:12]= instruction[31:12];
         end

      // J type
      7'b1101111: begin
            out[31:20] = {12{instruction[31]}};
            out[19:12]= instruction[19:12];
            out[11]= instruction[20];
            out[10:1]= instruction[30:21];
            out[0]= 1'b0;

         end
   endcase
end
```

2.1.7 The control unit

The control unit is one of the most important blocks in the processor because it basically acts as the mind of the processor that tells the data path what to do in each instruction, so it has to be designed with care for every single instruction.

After the full processor was drawn with all the required mux signals and all the control signals, the control unit was then designed to determine which signals are going to be high, low or don't care during each input instruction, where the instruction's opcode (bits 6 to 0) and func3 (bits 14 to 12) are mux-ed inside the control unit and the control signals are the outputted accordingly.

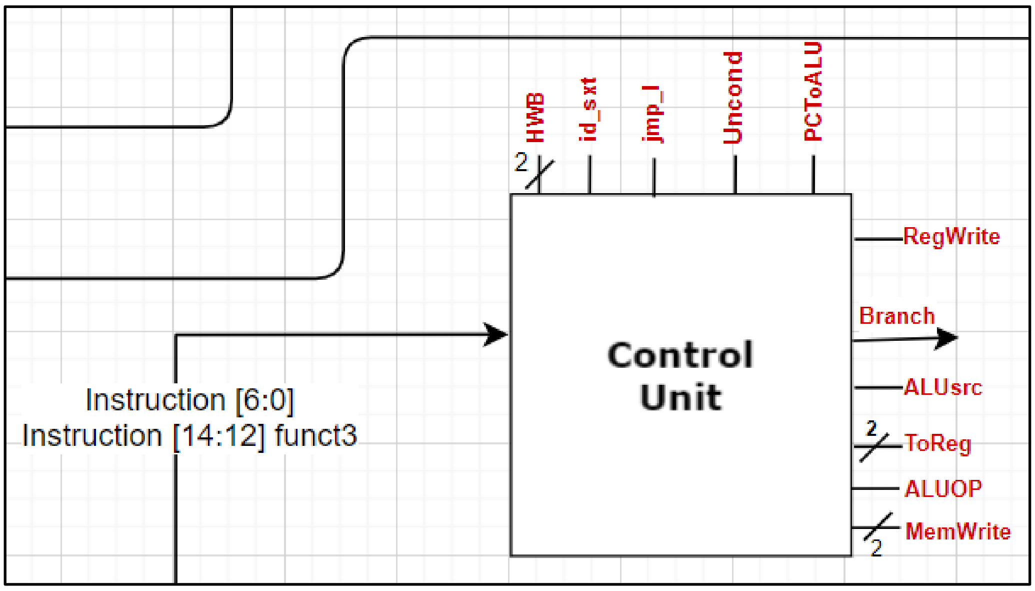
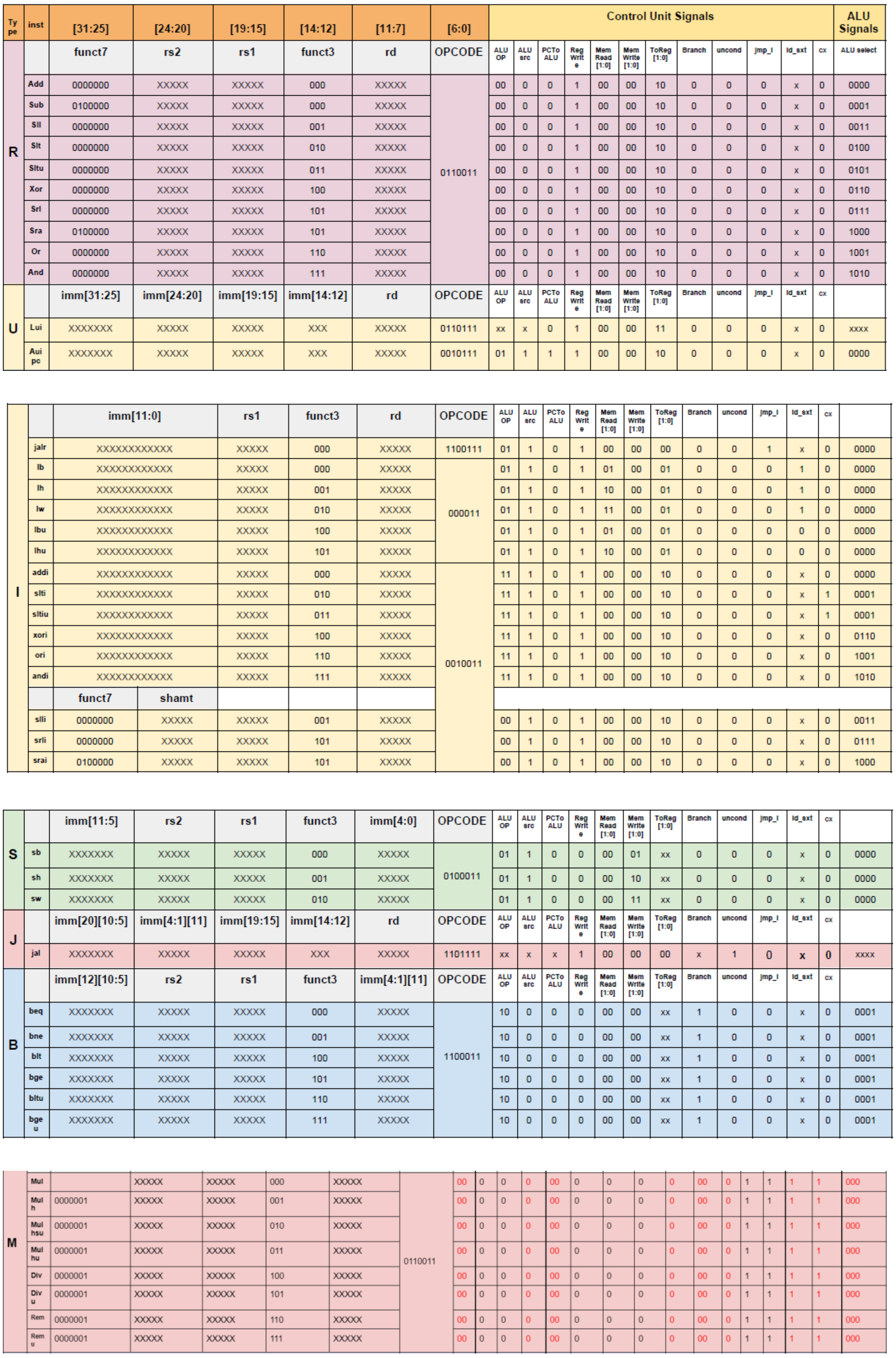


Figure 13: Control unit

```
module Control_unit(instruction,ALUOP,ALUsrc,PCToALU,RegWrite,MemWrite,
ToReg,Branch,Uncond,Jmp_i,Id_sxt,Cx,WHB,MULDIV,Rs1f,Rs2f);

input [31:0] instruction;
output reg [1:0] ALUOP,MemWrite,ToReg,WHB;
output reg ALUsrc,PCToALU,RegWrite,Branch,Uncond,Jmp_i,Id_sxt,Cx,MULDIV,Rs1f,Rs2f;

endmodule
```

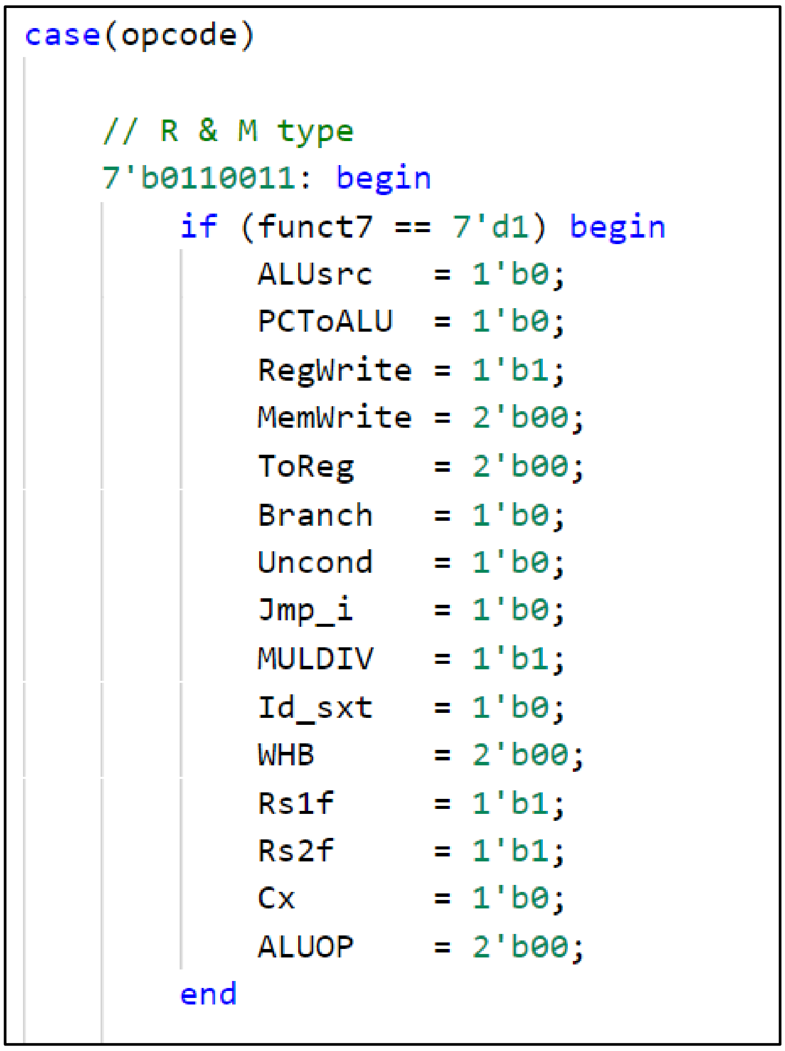
The following are the control signals for the instructions in the "I" and "M" extensions in the control unit design, then writing the RTL code for it.



Type	inst	[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Control Unit Signals												ALU Signals
R		funct7	rs2	rs1	funct3	rd	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	ALU select
	Add	0000000	XXXXX	XXXXX	000	XXXXX	0110011	00	0	0	1	00	00	10	0	0	0	x	0	0000
	Sub	0100000	XXXXX	XXXXX	000	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0001
	Sll	0000000	XXXXX	XXXXX	001	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0011
	Slt	0000000	XXXXX	XXXXX	010	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0100
	Sltu	0000000	XXXXX	XXXXX	011	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0101
	Xor	0000000	XXXXX	XXXXX	100	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0110
	Srl	0000000	XXXXX	XXXXX	101	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	0111
	Sra	0100000	XXXXX	XXXXX	101	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	1000
	Or	0000000	XXXXX	XXXXX	110	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	1001
	And	0000000	XXXXX	XXXXX	111	XXXXX		00	0	0	1	00	00	10	0	0	0	x	0	1010
U		imm[31:25]	imm[24:20]	imm[19:15]	imm[14:12]	rd	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	
	Lui	XXXXXXX	XXXXX	XXXXX	XXX	XXXXX	0110111	xx	x	0	1	00	00	11	0	0	0	x	0	xxxx
	Auipc	XXXXXXX	XXXXX	XXXXX	XXX	XXXXX	0010111	01	1	1	1	00	00	10	0	0	0	x	0	0000
I		imm[11:0]		rs1	funct3	rd	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	
	jalr	XXXXXXXXXXXX		XXXXX	000	XXXXX	1100111	01	1	0	1	00	00	00	0	0	1	x	0	0000
	lb	XXXXXXXXXXXX		XXXXX	000	XXXXX	000011	01	1	0	1	01	00	01	0	0	0	1	0	0000
	lh	XXXXXXXXXXXX		XXXXX	001	XXXXX		01	1	0	1	10	00	01	0	0	0	1	0	0000
	lw	XXXXXXXXXXXX		XXXXX	010	XXXXX		01	1	0	1	11	00	01	0	0	0	1	0	0000
	lbu	XXXXXXXXXXXX		XXXXX	100	XXXXX		01	1	0	1	01	00	01	0	0	0	0	0	0000
	lhu	XXXXXXXXXXXX		XXXXX	101	XXXXX		01	1	0	1	10	00	01	0	0	0	0	0	0000
	addi	XXXXXXXXXXXX		XXXXX	000	XXXXX	0010011	11	1	0	1	00	00	10	0	0	0	x	0	0000
	slti	XXXXXXXXXXXX		XXXXX	010	XXXXX		11	1	0	1	00	00	10	0	0	0	x	1	0001
	sltiu	XXXXXXXXXXXX		XXXXX	011	XXXXX		11	1	0	1	00	00	10	0	0	0	x	1	0001
	xori	XXXXXXXXXXXX		XXXXX	100	XXXXX		11	1	0	1	00	00	10	0	0	0	x	0	0110
	ori	XXXXXXXXXXXX		XXXXX	110	XXXXX		11	1	0	1	00	00	10	0	0	0	x	0	1001
	andi	XXXXXXXXXXXX		XXXXX	111	XXXXX		11	1	0	1	00	00	10	0	0	0	x	0	1010
		funct7	shamt																	
	slli	0000000	XXXXX	XXXXX	001	XXXXX		00	1	0	1	00	00	10	0	0	0	x	0	0011
	srli	0000000	XXXXX	XXXXX	101	XXXXX		00	1	0	1	00	00	10	0	0	0	x	0	0111
	srai	0100000	XXXXX	XXXXX	101	XXXXX		00	1	0	1	00	00	10	0	0	0	x	0	1000
S		imm[11:5]	rs2	rs1	funct3	imm[4:0]	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	
	sb	XXXXXXX	XXXXX	XXXXX	000	XXXXX	0100011	01	1	0	0	00	01	xx	0	0	0	x	0	0000
	sh	XXXXXXX	XXXXX	XXXXX	001	XXXXX		01	1	0	0	00	10	xx	0	0	0	x	0	0000
	sw	XXXXXXX	XXXXX	XXXXX	010	XXXXX		01	1	0	0	00	11	xx	0	0	0	x	0	0000
J		imm[20][10:5]	imm[4:1][11]	imm[19:15]	imm[14:12]	rd	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	
	jal	XXXXXXX	XXXXX	XXXXX	XXX	XXXXX	1101111	xx	x	x	1	00	00	00	x	1	0	x	0	xxxx
B		imm[12][10:5]	rs2	rs1	funct3	imm[4:1][11]	OPCODE	ALU OP	ALU src	PCTo ALU	Reg Write	Mem Read [1:0]	Mem Write [1:0]	ToReg [1:0]	Branch	uncond	jmp_I	ld_sxt	cx	
	beq	XXXXXXX	XXXXX	XXXXX	000	XXXXX	1100011	10	0	0	0	00	00	xx	1	0	0	x	0	0001
	bne	XXXXXXX	XXXXX	XXXXX	001	XXXXX		10	0	0	0	00	00	xx	1	0	0	x	0	0001
	blt	XXXXXXX	XXXXX	XXXXX	100	XXXXX		10	0	0	0	00	00	xx	1	0	0	x	0	0001
	bge	XXXXXXX	XXXXX	XXXXX	101	XXXXX		10	0	0	0	00	00	xx	1	0	0	x	0	0001
	bltu	XXXXXXX	XXXXX	XXXXX	110	XXXXX		10	0	0	0	00	00	xx	1	0	0	x	0	0001
	bgeu	XXXXXXX	XXXXX	XXXXX	111	XXXXX		10	0	0	0	00	00	xx	1	0	0	x	0	0001

M	Mul		XXXXX	XXXXX	000	XXXXX	0110011	00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Mulh	0000001	XXXXX	XXXXX	001	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Mulhsu	0000001	XXXXX	XXXXX	010	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Mulhu	0000001	XXXXX	XXXXX	011	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Div	0000001	XXXXX	XXXXX	100	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Divu	0000001	XXXXX	XXXXX	101	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Rem	0000001	XXXXX	XXXXX	110	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000
	Remu	0000001	XXXXX	XXXXX	111	XXXXX		00	0	0	0	00	0	0	0	0	00	0	1	1	1	1	000

Figure 14: Control unit design for the Core



```
case(opcode)

    // R & M type
    7'b0110011: begin
        if (funct7 == 7'd1) begin
            ALUsrc    = 1'b0;
            PCToALU   = 1'b0;
            RegWrite  = 1'b1;
            MemWrite  = 2'b00;
            ToReg     = 2'b00;
            Branch    = 1'b0;
            Uncond    = 1'b0;
            Jmp_i     = 1'b0;
            MULDIV    = 1'b1;
            Id_sxt    = 1'b0;
            WHB       = 2'b00;
            Rs1f      = 1'b1;
            Rs2f      = 1'b1;
            Cx        = 1'b0;
            ALUOP     = 2'b00;
        end
```

2.2 RV32M (M-extension)

The second extension implemented in the RISC-V core is the M extension (Standard Extension for Integer Multiplication and Division). The M extension consists of 8 instructions.

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

Figure 15: M-extension instructions

The first implementation to the M-extension block was two combinational blocks for Multiplication and division, where the multiplication block consisted of a series of full adders as shown in figure 16.

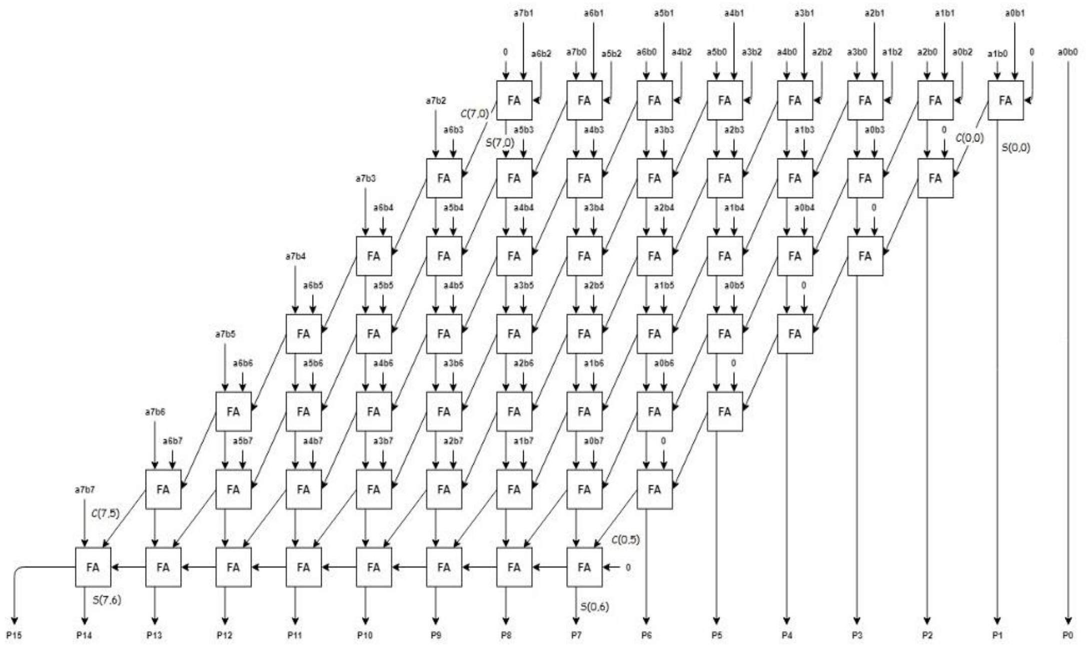


Figure 16: Combinational Multiplier

But after implementing this combinational implementation, the frequency was limited to 2.4 Mhz as the critical path was this combinational multiplier.

In order to solve this problem, a pipelined implementation was used for both the division and multiplication blocks, where the division used the restoring algorithm, and the multiplication block used the booth algorithm for multiplication.

2.2.1 Division using Restoring Algorithm

The Restoring division algorithm depends on the shift and add method. The used hardware is much simpler than the combinational divider, thus uses less area and power. The hardware implemented is a 32-bit ALU, a 64-bit shift register, a simple control test that decides on addition or subtraction in the ALU and decides on shifting left in the shift register or not and decides whether we will write the result from the ALU or not.

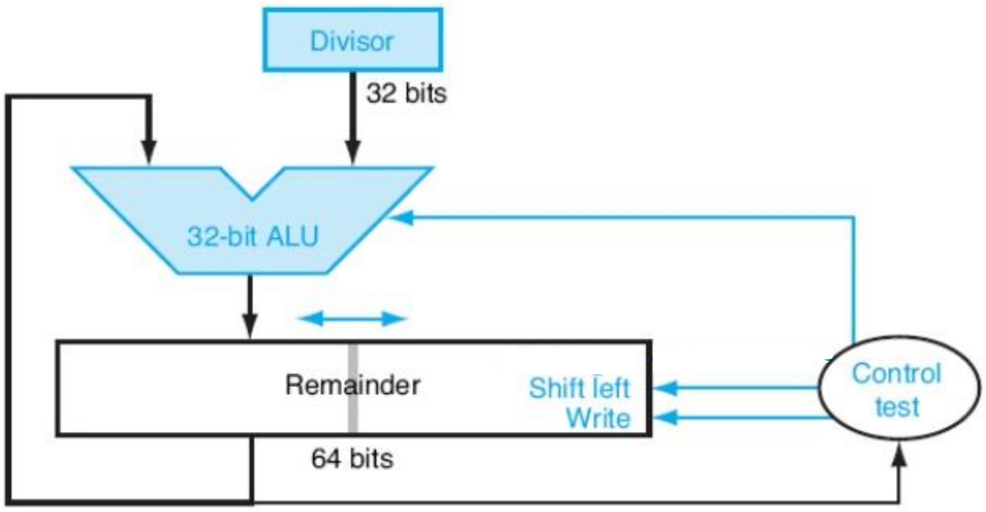


Figure 17: Restoring division algorithm implementation

The Algorithm follows the following flowchart

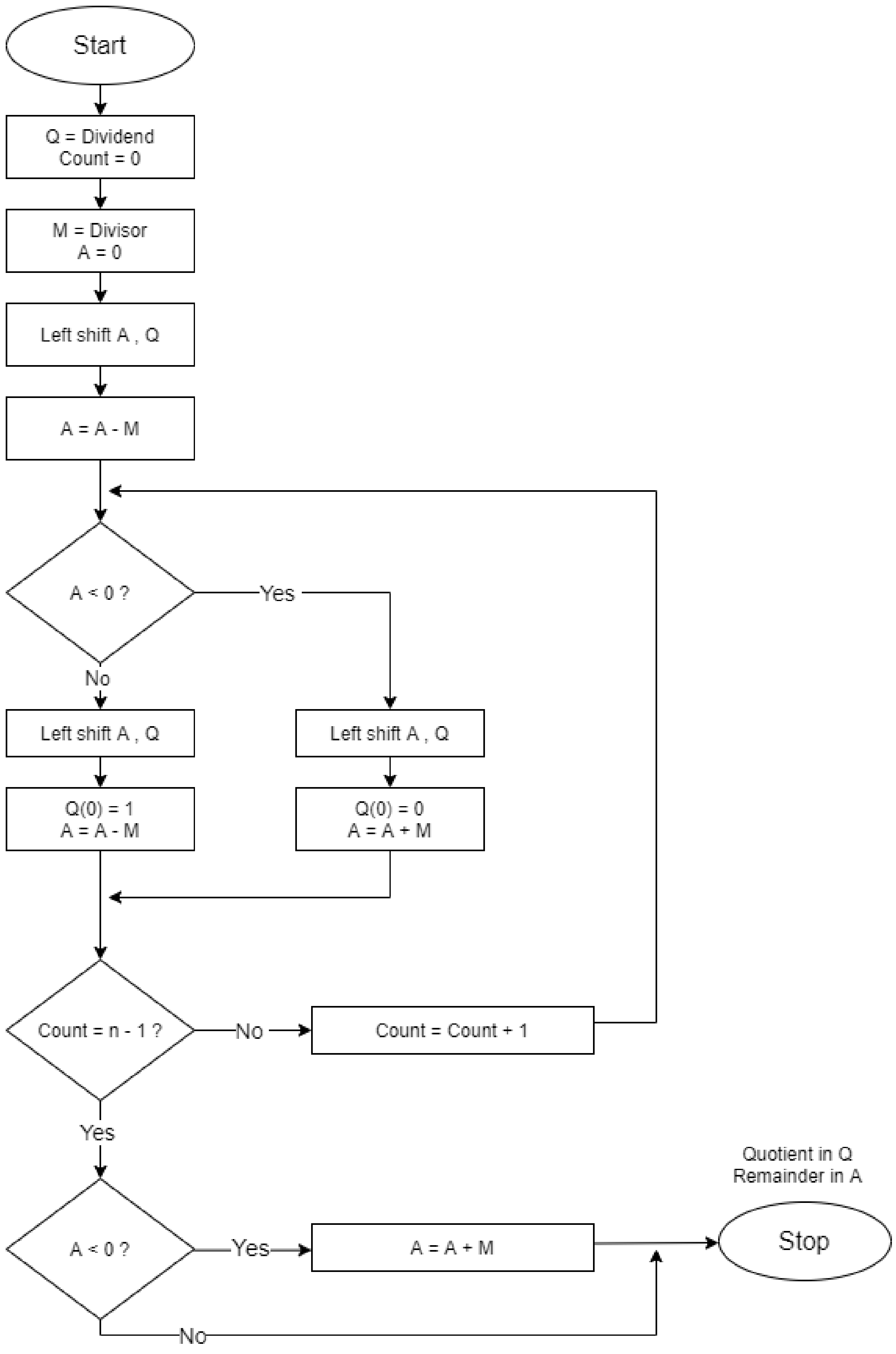


Figure 18: Restoring division flowchart

When we reach the stop state, the quotient will be in lower 32-bits and the remainder in the upper 32-bits of the shift register.

After designing the Divider block using restoring division algorithm, we concluded that it only works with unsigned numbers, therefore a wrapper was instantiated on top of the block in order to perform our signed and unsigned instructions as shown in figure 19.

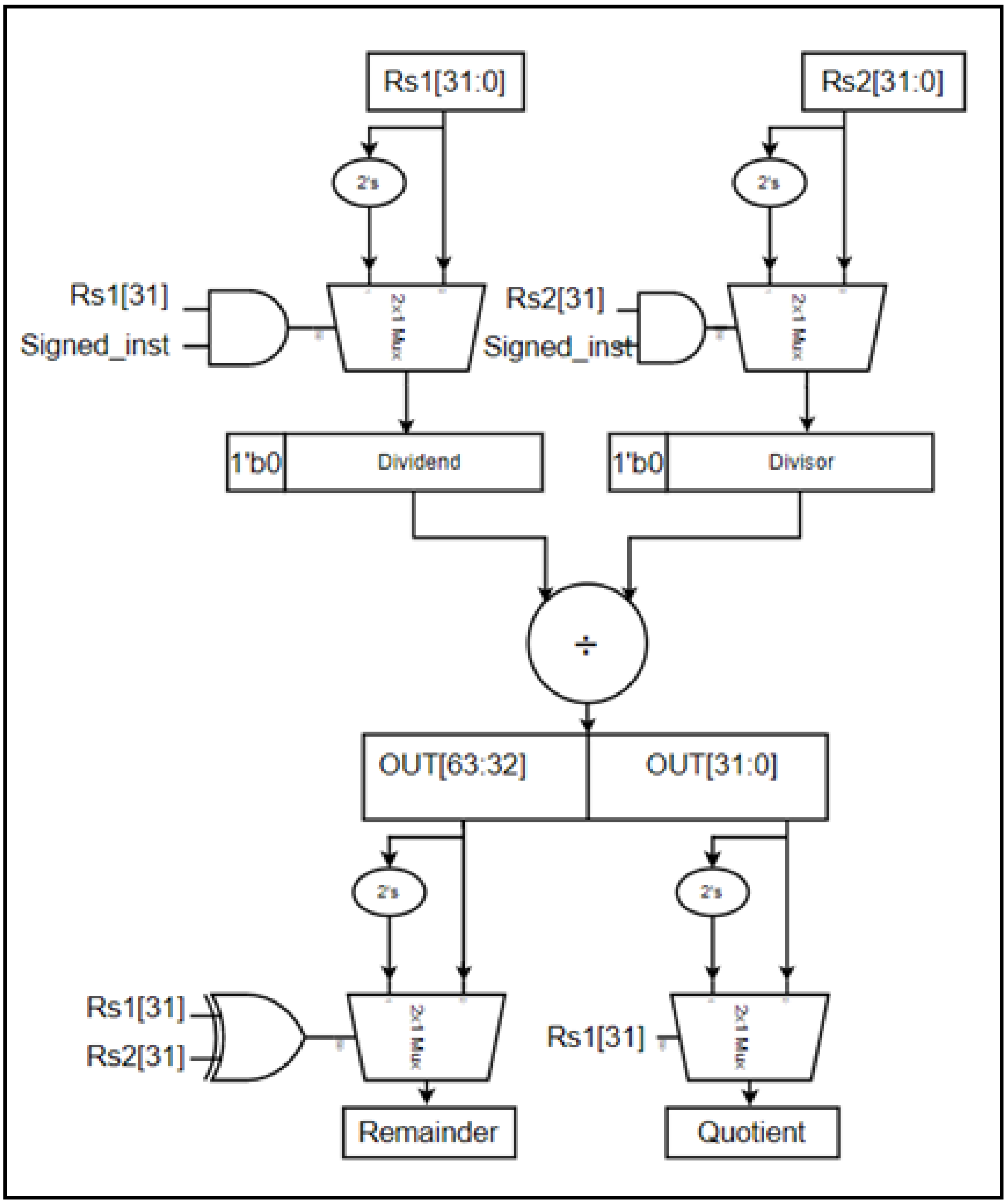


Figure 19: Division Wrapper

Dealing with just unsigned numbers, the algorithm should see the positive version of the number thus, taking the 2’s complement of the inputs if the number is negative and the instruction is signed (“Div” or “Rem”). After that, to ensure that there is a zero in the most significant bit, the Dividend and the divisor gets an appended zero in the most significant bit. After the division algorithm is done, we want to restore the signs of the quotient and the remainder. Then the Quotient would take the sign of the dividend XOR the sign of the divisor. The temporary remainder, however would take the sign of the dividend, so if the dividend was negative, 2’s complement would be performed on the remainder, and if positive the remainder is as it is, then a MUX chooses between the temporary remainder and the original dividend, given that the instruction is “Remu” and the dividend is smaller than the divisor, the remainder is the original dividend Rs1, as shown in the wrapper.

2.2.2 Multiplication using booth Algorithm

Booth algorithm for multiplication is very similar to the restoring division algorithm, it also depends on the shift and add mechanism and consists of the same hardware as the division algorithm which is a 32-bit ALU, a shift register and a small control test to determine the ALU and shift register operation as shown in figure 20.

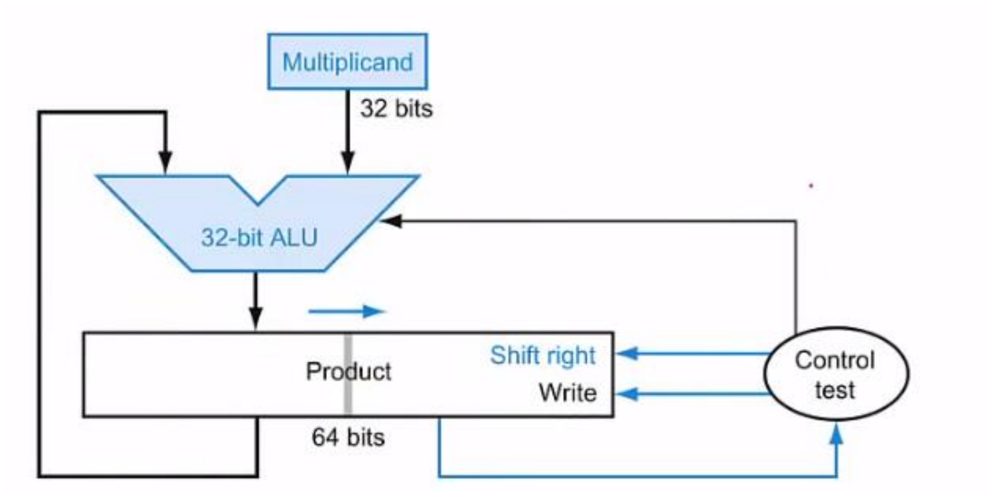


Figure 20: Booth algorithm implementation

The algorithm follows the following flowchart:

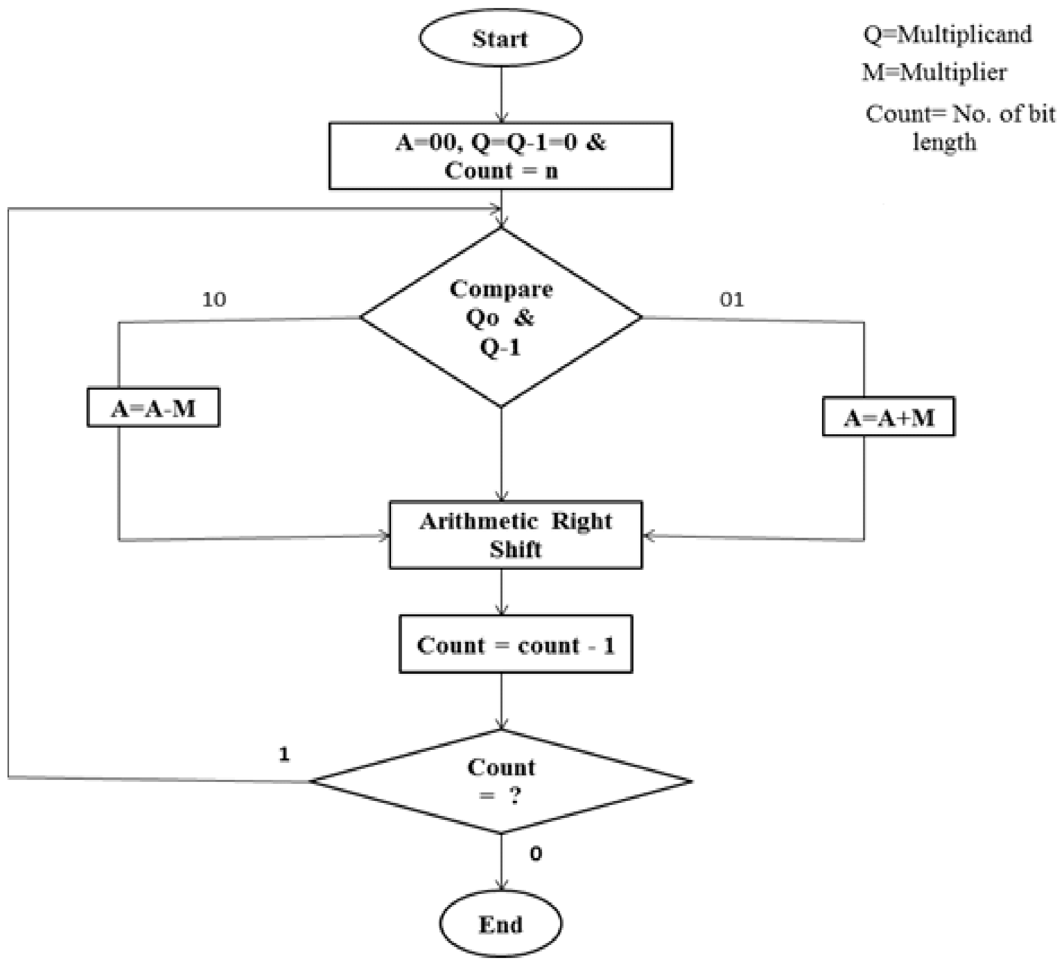


Figure 21: Booth algorithm flowchart

After designing the multiplier block using booth's algorithm, we concluded that it only works with signed numbers, therefore a wrapper was instantiated on top of the block in order to perform our signed and unsigned instructions as shown in figure 22.

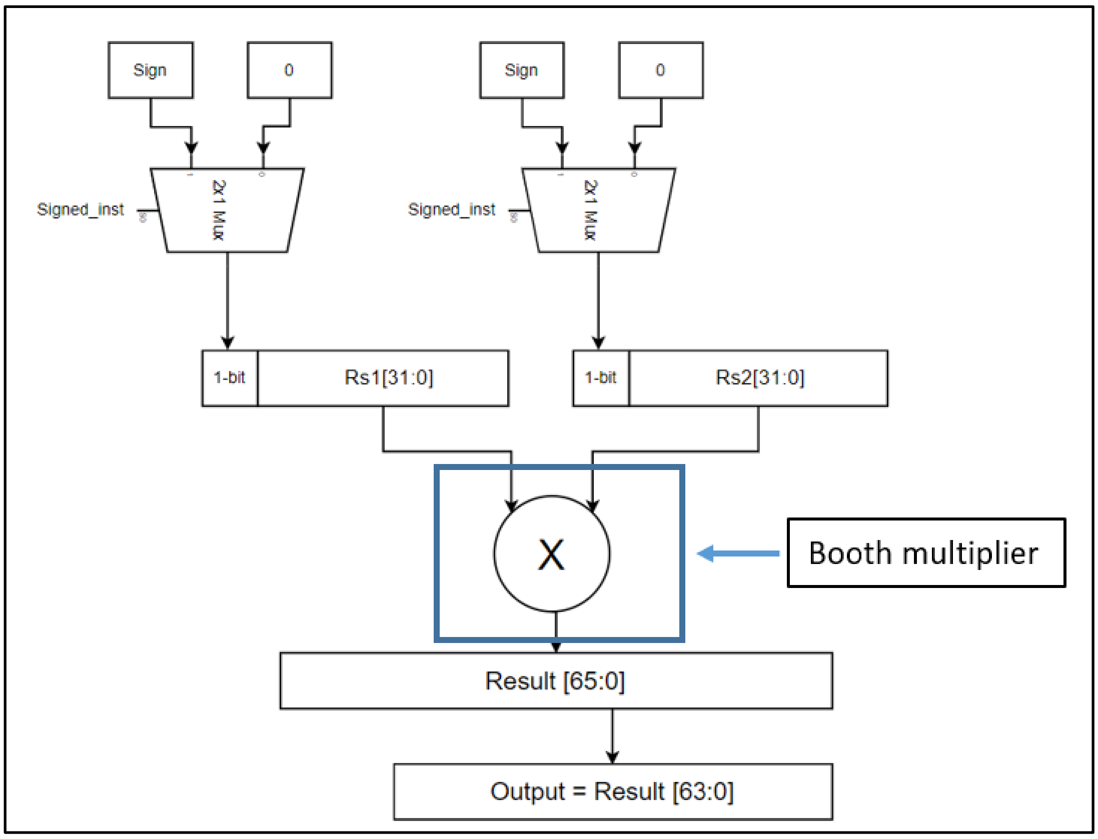


Figure 22: Multiplication Wrapper

Booth algorithm only deals with signed numbers, so to ensure that the most significant bit will contain the sign, the multiplier and multiplicand will get a bit appended in the most significant location which is zero in case the instruction is unsigned, or 1 in case the instruction is signed.

After converting the combinational MUL/DIV block to a pipelined block using the previous two algorithms, the frequency increased to 16 MHz and the Multiplier/Divider were no longer the critical paths.

Chapter 3: Designing the Pipelined 32 IM RISC-V core

Now for speeding up our processor, pipelining is a very important step. Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. It basically breaks up our single instruction path into a number of stages allowing simultaneous execution of more than one instruction takes place in a pipelined processor. The following is a 5-stages pipeline, we chose our processor to be five stages as well.

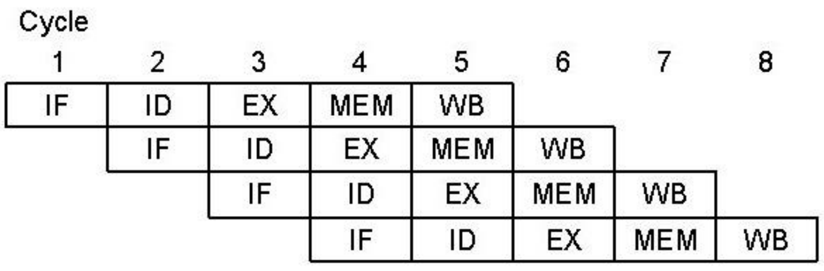


Figure 23: 5-stages pipeline processor

3.1 Hazards

1- Structural hazards: occurs due to resource contention, where in case of having one memory for both instructions and data (von-Neumann architecture), an instruction would want to write in the memory and another would want to read from the memory at the same time, this was avoided using separate instruction and data memories.

2- Data hazards: Data hazards occur due to data dependency, where a value that is still being calculated in a stage is required in another stage at the same time. Ignoring potential data hazards can result in race conditions. There are 3 types of data hazards:

- Read After Write (RAW) - Write After Read (WAR) - Write After Write (WAW)

3- Control hazards: Control hazard occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. The term branch hazard also refers to a control hazard.

3.2 Hazard detection Unit

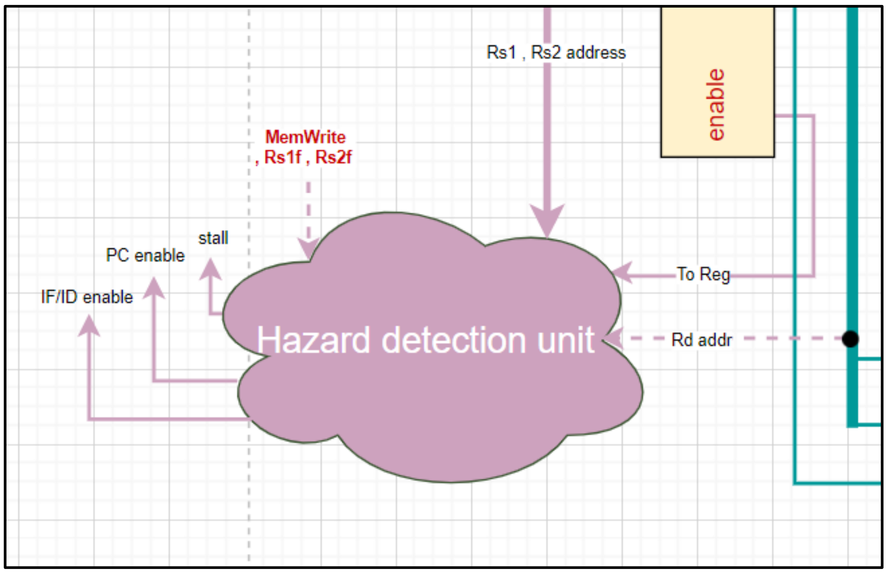
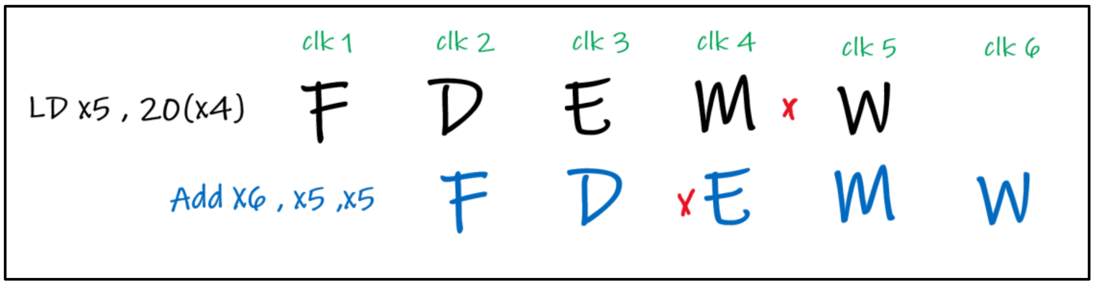


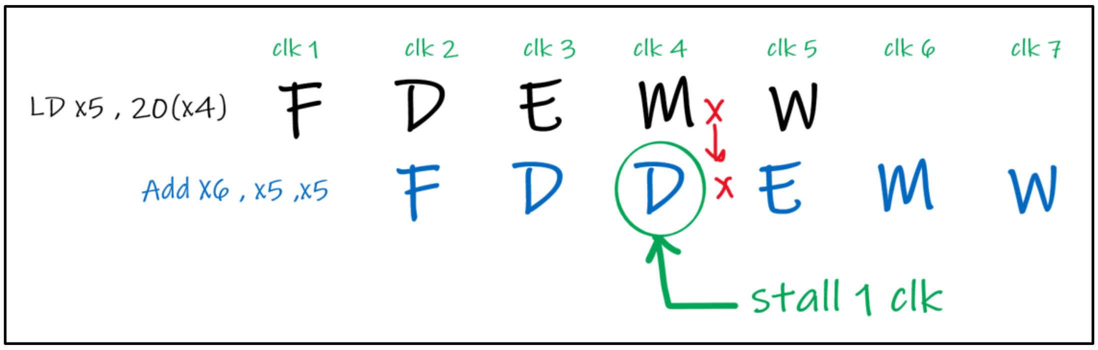
Figure 24: Hazard detection unit

Hazard detection unit is a unit in the decode stage responsible for generating a stall signal to stop the first two stages for one clock cycle, by disabling the PC register enable and IF/ID register enable. But why do we need to generate a stall signal? Data hazards can have 2 different shapes. The first one is data dependency between stages, where the values needed are generated but is not stored yet (no need to generate stalls in this case, problem can be solved with forwarding unit). Second one is data dependency between stages, but the values needed are not generated yet (we need one more clock to generate value to forward it in the next clock cycle). We need to generate stalls for any data dependency between load instructions coming before any R-Type instruction. The following section explains the reason for that.



	clk 1	clk 2	clk 3	clk 4	clk 5	clk 6
LD x5 , 20(x4)	F	D	E	M x	W	
Add X6 , x5 ,x5		F	D	x E	M	W

In this case we there is a load instruction that will update the register x5 with value stored in memory location = 20 + val(R4). For this timing diagram we will notice that the add instruction needs the updated value of x5 at clk4 but the memory will load its value by the end of clk4 so it will be ready at clk5, so we can't forward the value (value isn't generated yet to be forwarded).

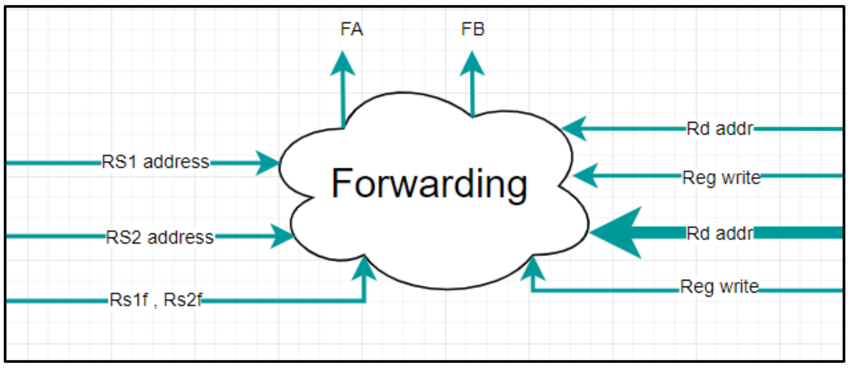


So, solution for that case is to generate stall signal that disable enables of both PC and IF/ID register the instruction in the fetch and decode stage will hold for one more clock. Now the add instruction needs the value of R5 at clk5 and the data loaded from memory will be ready so we can forward it.

3.3 Forwarding

Forwarding unit is a unit that generate signals to control MUXs' selection lines to select data output from other stages instead of regular data bus, there are two forwarding units:

- **Forward unit at stage 3:**
  Used to forward data needed for stage 3 only. We need to check if there are data dependency between stage3 and (stage 4/stage 5). If Rs1 and/or Rs2 addresses are equal to Rd address of stage 4 and/or stage 5, this means that the instruction in stage 3 needs a value from stage 4 or stage 5. The forwarding unit will generate FA (forward operand A) signal and FB (forward operand B) that control the input operands of the ALU to select a proper signal forwarded from the next stages.



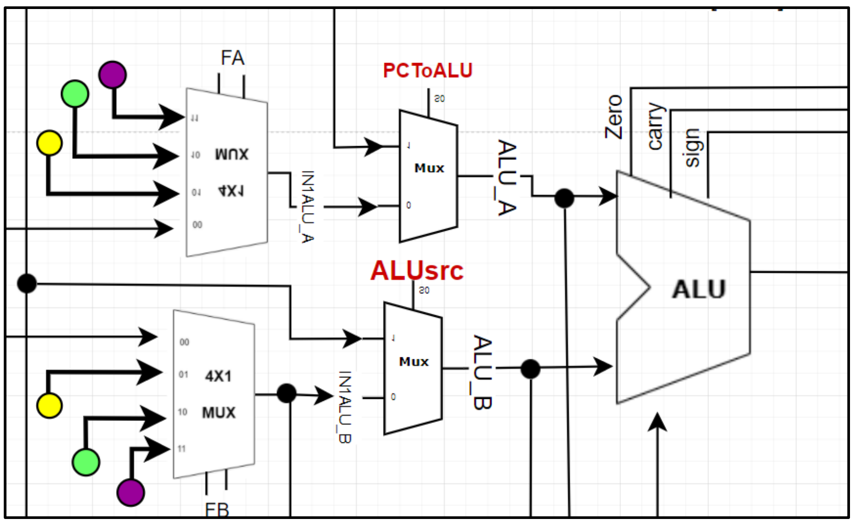
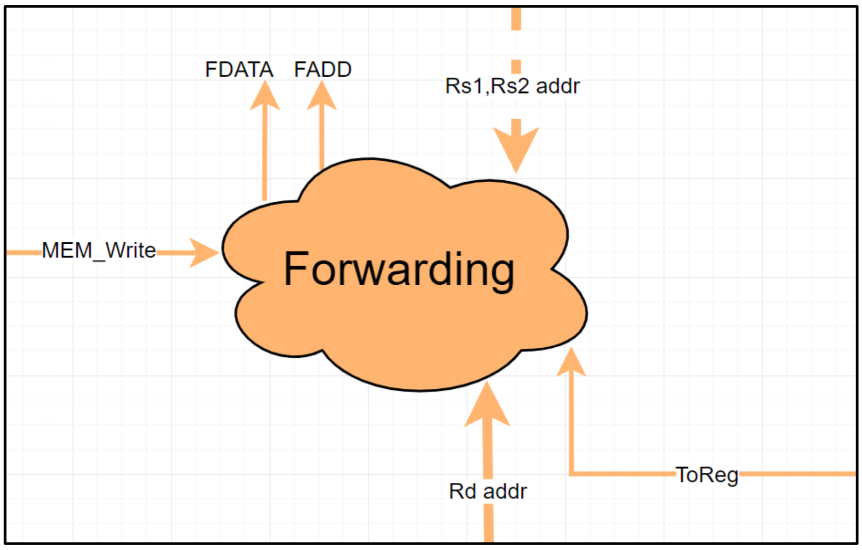


Figure 25: Stage 3 forwarding unit

- **Forward unit at stage 4:**
  Used to forward data needed for stage 4 only that access the memory. We check if there are data dependency between stage 4 and stage 5. If Rs1 and/or Rs2 addresses of stage 4 are equal to Rd address of stage 5, this means that the instruction in stage 4 needs a value from stage 5. The forwarding unit will generate FADD signal and FDATA signal that control the input operands of the memory to select a proper signal forwarded from stage 5.



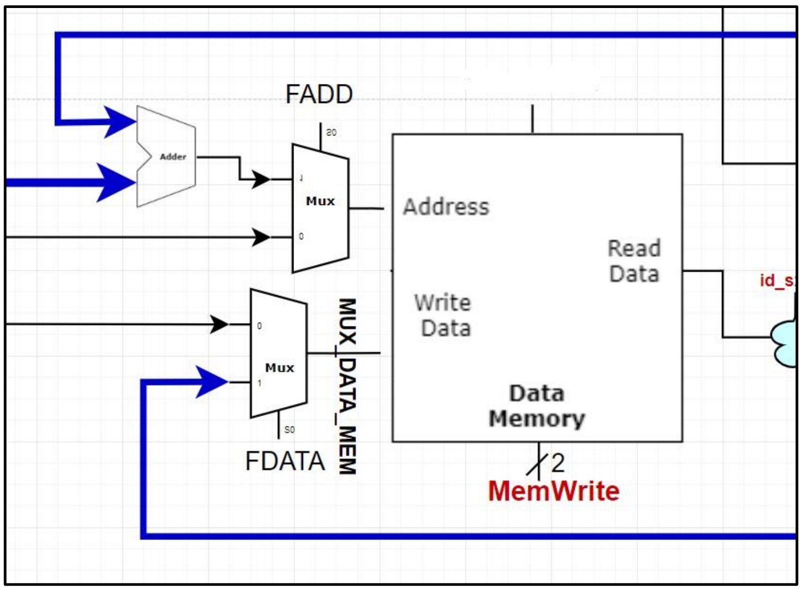


Figure 26: Stage 4 forwarding unit

3.4 Control hazards unit

So far, we have limited our concern to hazards involving arithmetic operations and data transfers. However, there are also pipeline hazards involving conditional branches. Figure 27 shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch or not doesn’t occur until the MEM pipeline stage (4th stage). This delay in determining the proper instruction to fetch is called a control hazard or branch hazard, in contrast to the data hazards we have just examined, control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards.

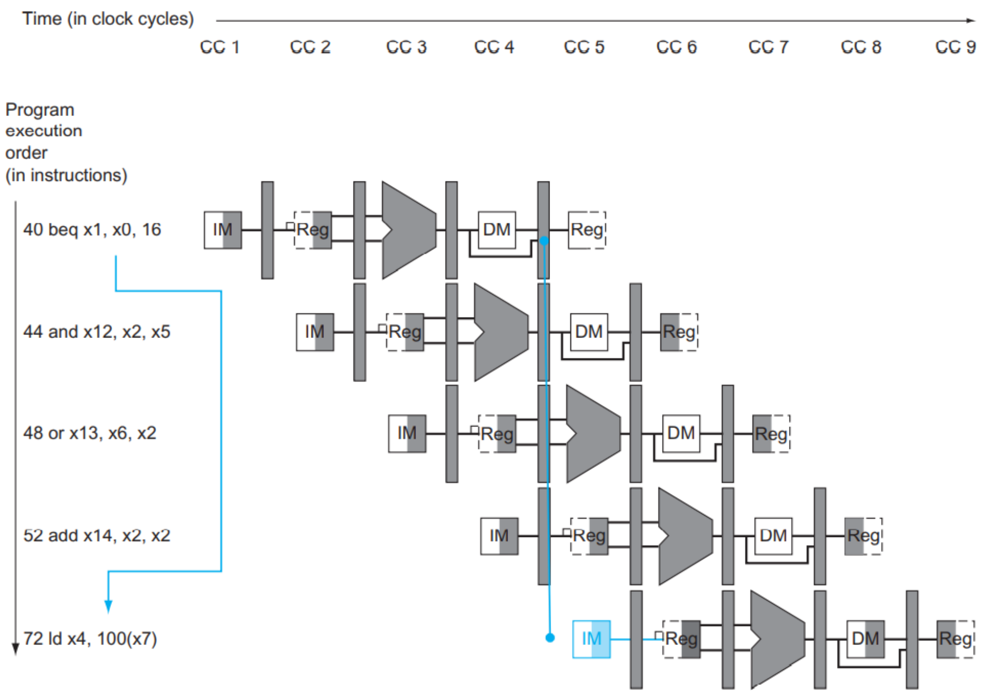
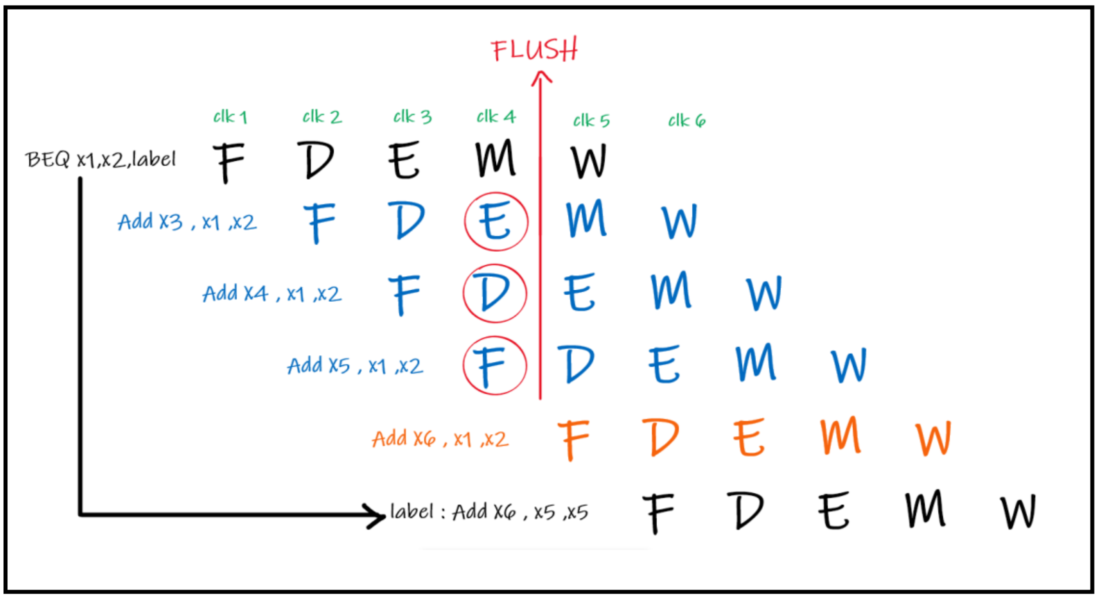


Figure 27: Control hazard in pipeline

Hence, we use simpler schemes. There are multiple schemes for resolving control hazards, the scheme we chose to follow is that we will always assume that the Branch is not taken, and if it's taken, we will flush the three instructions that have entered the pipeline in the first three stages after branch to ensure the right instructions sequence. Assume we have the following instructions:


FLUSH
clk 1
clk 2
clk 3
clk 4
clk 5
clk 6
BEQ x1,x2,label F D E M W
Add X3 , x1 ,x2 F D E M W
Add X4 , x1 ,x2 F D E M W
Add X5 , x1 ,x2 F D E M W
Add X6 , x1 ,x2 F D E M W
label : Add X6 , x5 ,x5 F D E M W

In the previous figure, we have a sequence of instructions starting with instruction (BEQ x1, x2, label), which means that if x1 is equal to x2, the program counter should jump to the instruction that is at the label. Assuming x1 and x2 are equal, so branch is taken in that case and the instructions coming after the BEQ shouldn't be executed and flushed, but unfortunately, branch is evaluated at the fourth stage so 3 instructions would have been fetched and entered the pipeline in the first 3 stages, these instructions must be FLUSHED from the processor (have no effect) as the branch is taken. A flush signal is added in the pipeline control which have one basic operation which is to remove the instructions at the Registers (IF/ID, ID/EX, and Ex/MEM), thus eliminating any control hazards that might occur.

Chapter 4: SoC peripherals and Integration

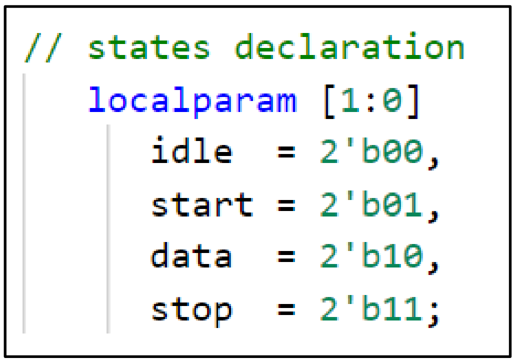
4.1 UART module

The UART (Universal Asynchronous Receiver and transmitter) module is used to send and receive the data required to be encrypted and decrypted by the IoT server or the IoT device. The UART module consists of 3 main blocks: UART transmitter, UART receiver and a baud generator for sampling. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and then reassembles the data. The serial line is 1 when it is idle.

The most commonly used sampling rate is 16 times the baud rate, which means that each serial bit is sampled 16 times. The oversampling scheme basically performs the function of a clock signal. Instead of using the rising edge to indicate when the input signal is valid, it utilizes sampling ticks to estimate the middle point of each bit.

4.1.1 UART Transmitter

The UART transmitter was written as a finite state machine with 4 states as shown.



```
// states declaration
   localparam [1:0]
      idle  = 2'b00,
      start = 2'b01,
      data  = 2'b10,
      stop  = 2'b11;
```

Idle: There is no data to be sent and the UART tx is high.

Start: Tx changed from high to low indicating the start bit

Data: the actual data transmission which is shifting a shift register loaded with the data byte required to be sent

Stop: the data byte was sent and the tx returns to active high indicating idle state.

4.1.2 UART Receiver

The UART receiver has the same states and operates very similarly to the transmitter except for the shift register's input bit, where it receives the MSb as the shift in bit and shifts right, where the transmitter sends the LSb and keep shifting right with zero as the shift in bit.

4.1.3 Baud generator

The Baud generator is responsible for the oversampling of bits, where it generates a sampling signal with frequency equals to 16 times the UART's baud rate, to avoid creating a new clock domain and violating the synchronous design principle, the sampling signal should function as enable ticks rather than the clock signal to the UART receiver and UART transmitter as shown in figure 28.

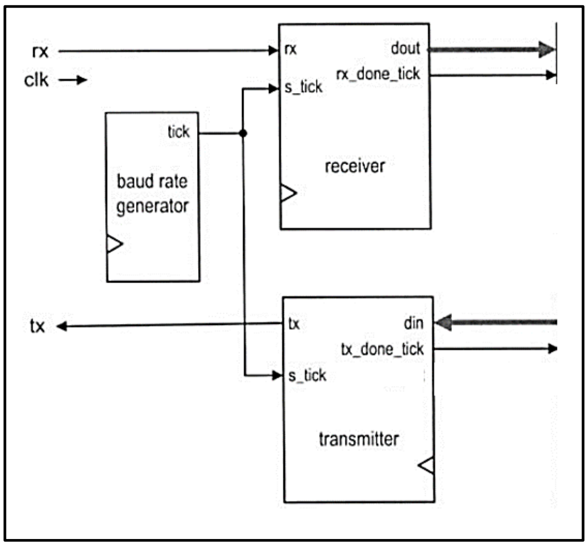


Figure 28: UART block diagram

4.2 Acorn security module

In order to Encrypt/Decrypt our data, we need an Encryption algorithm and the hardware to implement that algorithm. The CEASAR competition which is short for: Competition for Authenticated Encryption: Security, Applicability and Robustness, is a competition for authenticated encryption ciphers which targets 3 main categories, the first category is ciphers for lightweight applications (resource limited environments), the second category is High performance applications, for applications that require high throughput, and the last category is defense in depth, which is for applications which require a very high level of security.

Authenticated encryption ciphers take a message (M), an associated data (AD), a public message number (Npub), and an optional secret message number (Nsec) as an input and generate resulting cipher text (C), Tag (Tag) and optional encrypted (Nsec). Integrity of data and authenticity of sender are ensured by a keyed-hash computation which occurs on all blocks of (Npub), (AD) and (M). The result of these computations is forwarded to the recipient as a Tag. In authenticated decryption, the recipient receives original (AD) and (Npub), along with (C) and (Tag), and uses Key to decrypt (C) to (M).

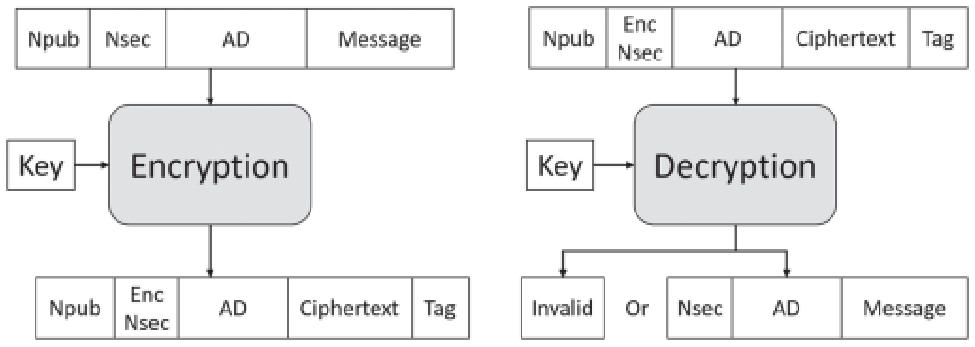


Figure 29: Authenticated Encryption ciphers

The authenticated decryption recreates a Tag (Tag’) and releases the cipher text if and only if Tag = Tag’, the module outputs a word “E0” indicating success, then authentication and integrity of the transaction are assured, otherwise the decrypted cipher text is not released and the module outputs a word “F0” indicating failure, thus achieve Authentication. Our project falls under the first category which is the lightweight applications as our project targets IoT security. Therefore, the algorithm we chose to implement on our SoC is the ACORN algorithm, which is the winner of the lightweight category as it achieves best balance between Area, security level and low power.

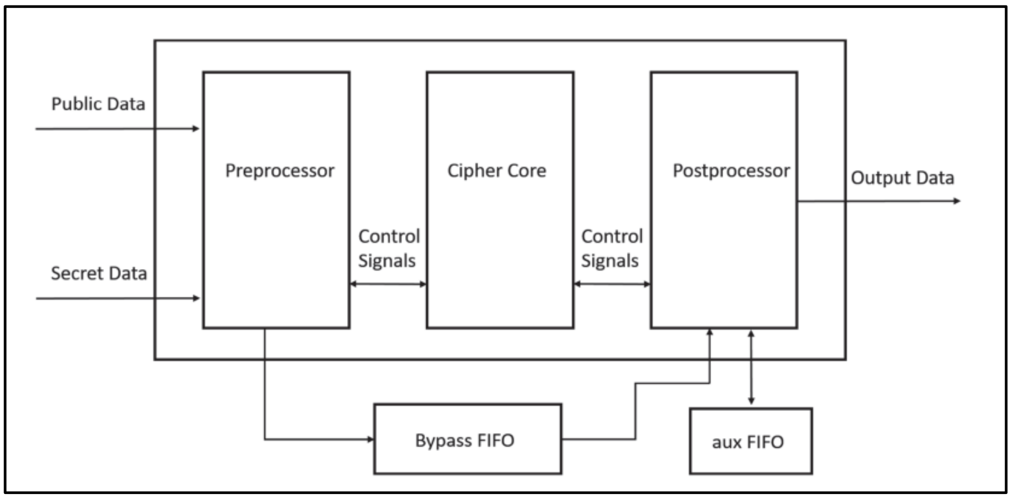


Figure 30: Authenticated ciphers API

In order to be able to easily test the algorithms, the Hardware Application Programming Interface (API) for authenticated ciphers has been developed to meet all the requirements of all algorithms that have been submitted to the CAESAR competition. The top level of the API is the Authenticated Encryption with Associated Data (AEAD) core. The architecture of the AEAD core consists of three main blocks: pre-processor, cipher core, and post-processor. The main difference between the different algorithms is in the cipher core implementation, as it contains the hardware blocks that perform either encryption or decryption and authentication algorithm steps.

Pre-processor, it is the first block of the AEAD core, which receives public and secret data and start processing them.

Postprocessor, the postprocessor carries out the following functions: clearing any portion of the output block that does not belong to the message, parallel-in serial-out conversion of the output blocks and generates the status block.

Cipher Core, the cipher core is divided into two blocks: the core data path and the core controller. The core data path contains the hardware, which is responsible for encryption or decryption and processing the associative data to perform tag generation. The core controller is a state machine that controls the data path.

Bypass First-In-First-Out (FIFO), which bypasses the tags, header, associated data and any data blocks that are used in the authentication process and will not be encrypted.

Auxiliary First-In-First-Out (FIFO), it is the memory used by the post-processor to temporarily store the decrypted message until the result of authentication is ready.

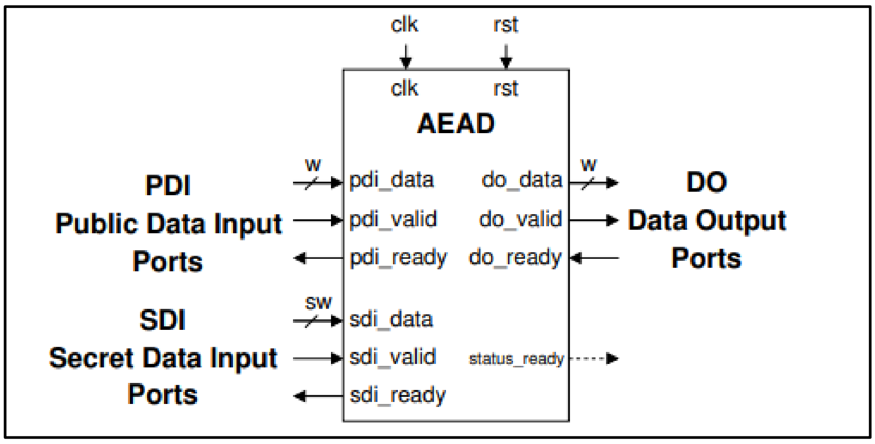


Figure 31: AEAD top module

4.3 Integration of SoC

4.3.1 Reset Domain Crossing

The term "reset domain crossing" (RDC) refers to a design method in which the source and destination parts (flops, latches, and clock gates) operate on different independent resets.

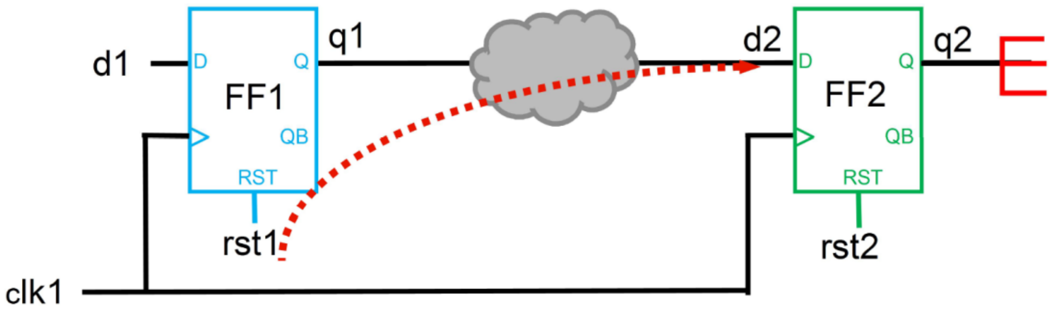


Figure 32: Reset domain crossing

Metastability occurs when an asynchronous reset from one reset domain causes a transition too close to the clock edge of a flip-flop in another reset domain or without a reset, causing a non-deterministic flip-flop value that propagates throughout the design resulting in functional failures, in the next figure, the rst2 signal changes very clock to the 6th clock edge, causing data on q2 to be metastable.

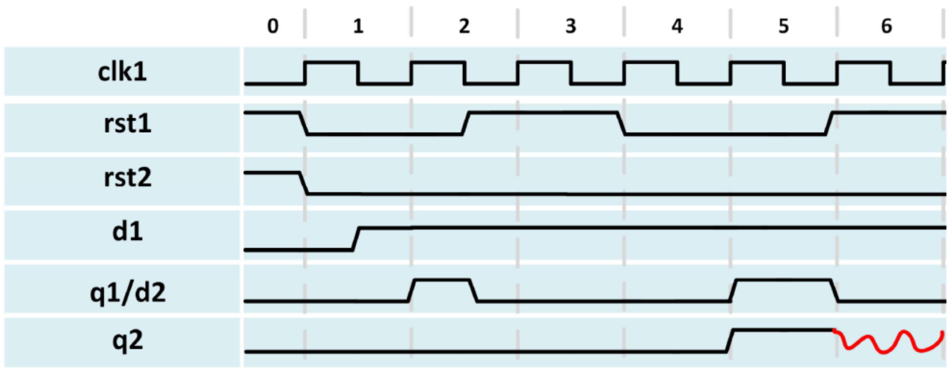


Figure 33: RDC metastability

The most common solutions for RDC issues include performing clock gating on the second domain in case we are resetting the first domain or using an isolation gate to isolate data changing on the flop output during resetting, another solution is adding another flip flop to the receiving domain to act as a synchronizer as shown in figure 32.

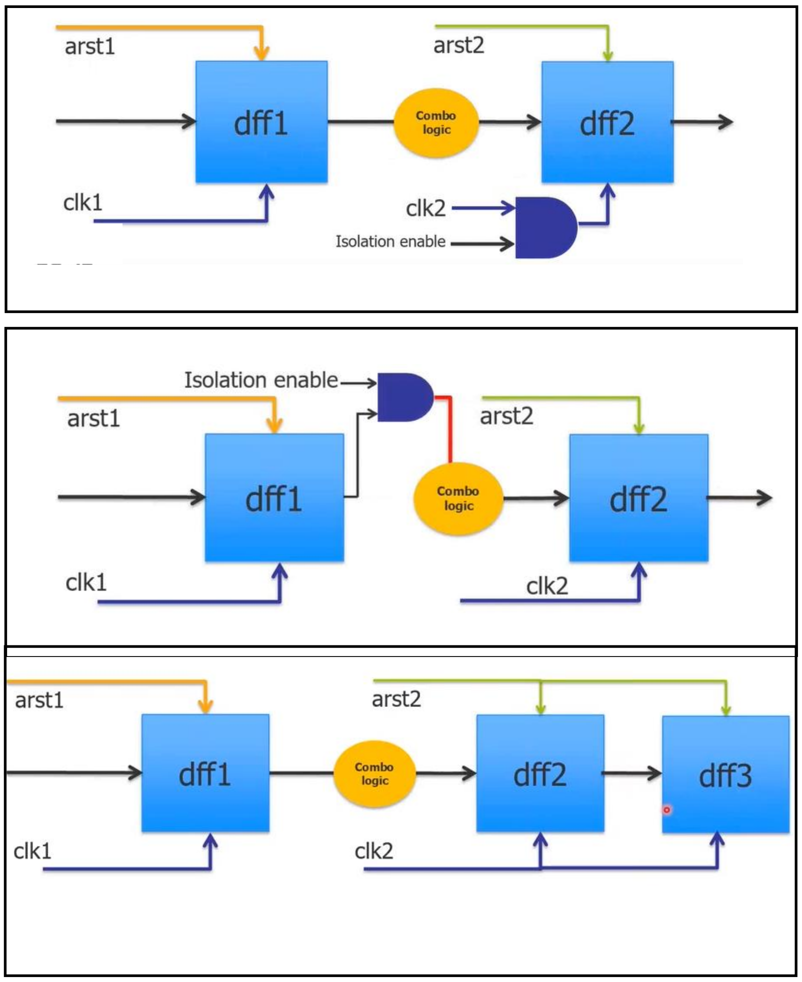


Figure 34: RDC common solutions

Having used asynchronous fifo to eliminate metastability due to CDC as will be explained later, this problem has also been avoided, as when exchanging the pointers between the reading and writing domains, the pointers are gray encoded and a synchronizer is added, thus eliminating the metastability that may occur.

Using asynchronous reset in our SoC however, will cause a similar problem to that in the RDC even in the same reset domain which is the reset de-assertion.

The problem here is that the reset signal can be de-asserted very close to a clock edge thus violating the recovery time (analogous to Tsu) of the reset, where data can change on the flop input just before the clock edge leading to a metastability due to setup violation, and violating the removal time (analogous to Thold) of the reset where if the reset is de-asserted after the clock edge but very close to it, the clock edge may still be in effect so it can cause metastability due to a hold violation.

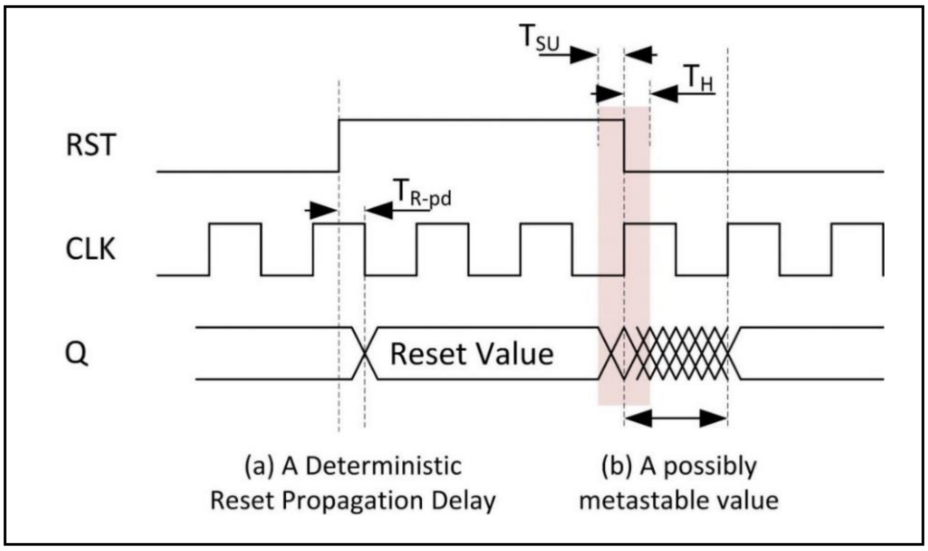

T_{SU}
T_H
RST
$T_{R\text{-}pd}$
CLK
Q
Reset Value
(a) A Deterministic Reset Propagation Delay
(b) A possibly metastable value

Figure 35: Reset de-assertion metastability

The solution we used is a 2D-flop synchronizer for the asynchronous reset signal, so that the reset assertion is done normally, but the reset de-assertion is synchronized to the clock edge and won’t cause the flip-flops to go into a metastable state.

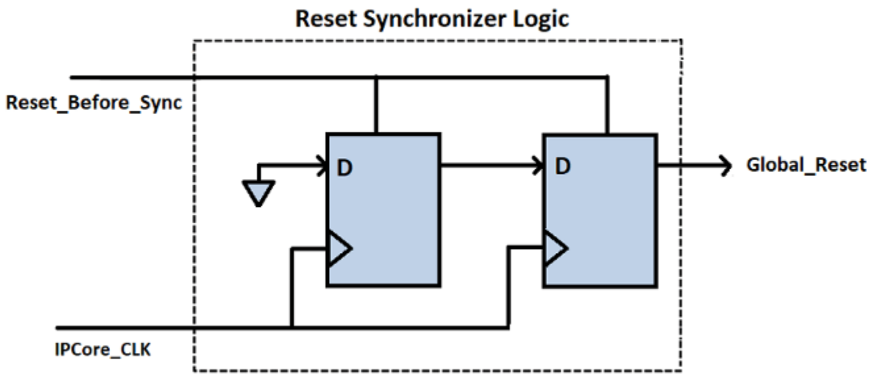


Figure 36: Asynchronous Reset synchronizer

4.3.2 Clock Domain Crossing

Having 3 clock domains in our SoC design, it was required to think of how we are going to synchronize between these 3 domains. But why do we require synchronization?

In multiple clock domains, if we are transferring data from one domain to another domain, if the 2 clocks are not an integer multiple of each other, data from domain one will inevitably arrive at the setup or hold time of the second domain leading to a metastability as shown in the following in figure 35.

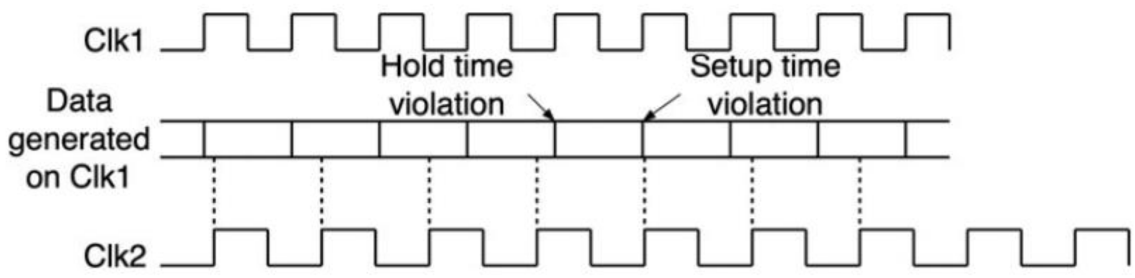


Figure 37: Clock domain crossing metastability

A setup violation means that the data didn't have enough time to be latched correctly in the master latch of the flop before the clock edge came, where a hold violation means that the data changed right after the clock edge, so it would change the data in the flop during the 1-1 overlap of the clock.

When a setup or hold time violation occur, the data enters what is called a metastable state, where we don't know what the value that this data will stabilize into, hence this is a failure. Even if the data stabilized into the correct value, it would need time for this which will be added to the path delay of the next pipeline path, hence it may cause setup violations and still this is a failure. Another problem that may occur due to clock domain crossing is data loss, where if the transmitting domain is faster than the receiving domain even if both are synchronized, some data may not be matched by the second domain and leads to data loss. So now that we concluded that we must synchronize between our clock domains, there are multiple ways to do that.

4.3.4 2-flop synchronizer

The simplest approach used for synchronization is the 2-flop synchronizer as shown below. In some cases, in higher speed designs, a three-flop synchronizer may be used.

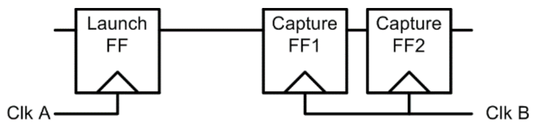


Figure 38: Two flop synchronizer

Simple as it is, it's only useful in synchronizing only a single bit, and fails to synchronize a multiple bit bus between two clock domains, this is due to the re-convergence issue. The re-convergence issue is due to the one cycle uncertainty present in the 2-flop synchronizer.

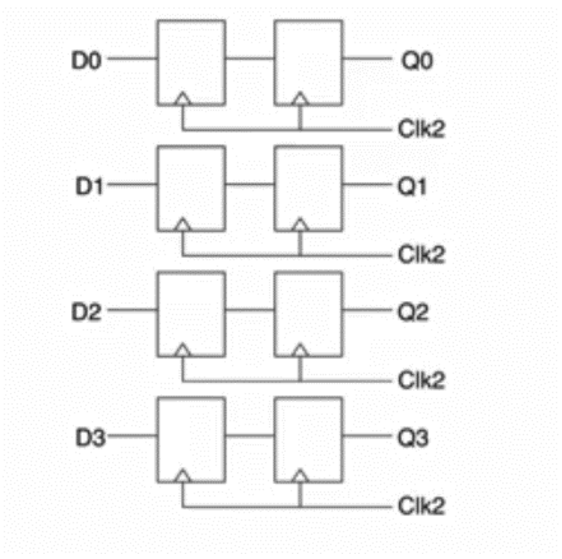


Figure 39: 2-flop synchronizer re-convergence issue

In figure 37, a data bus (D3 to D0) from domain one is being synchronized to domain two with Clk2. The problem here is that we can't determine if the correct data arrived at (Q3 to Q0) after one clock cycle, or a metastability occurred and the correct data was latched correctly after the second clock cycle.

4.3.5 Handshake synchronizer

Another approach was to be used for synchronization which is the handshake synchronizer, where the transmitting domain outputs a request signal and the required data on the data bus, the receiving domain outputs an acknowledgment signal, as shown in figure 38.

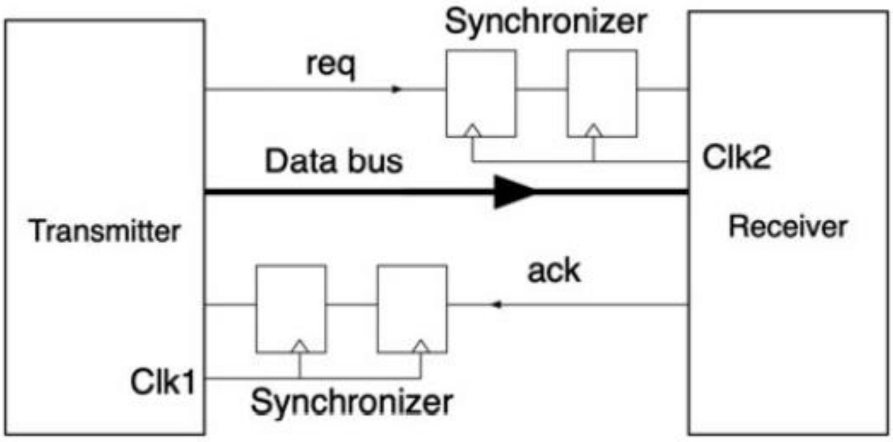


Figure 40: Handshake synchronizer

The req and ack signals are one bit signals, so no re-convergence issue here, the problem here is the large overhead required to send one word of data. The correct transmission of one word of data takes about 9 clock cycles, so this solution would be efficient only for small data rates. The most suitable approach to our design was the Asynchronous fifo. Where we could send words of data between the two clock domains with a much smaller overhead cycles and there would be no data loss and no metastability.

4.3.6 Asynchronous fifo

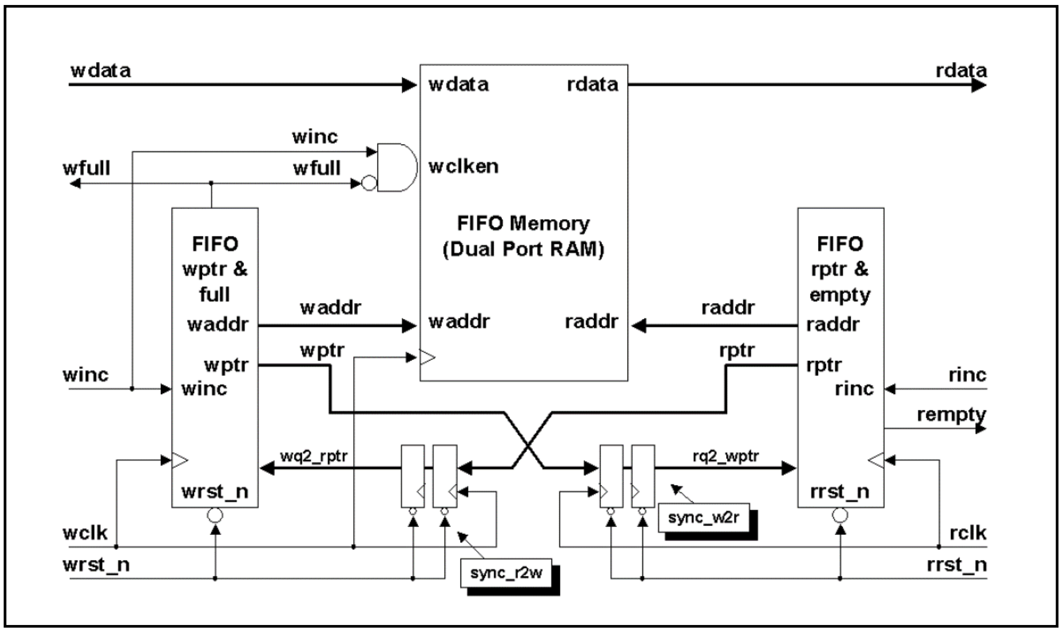
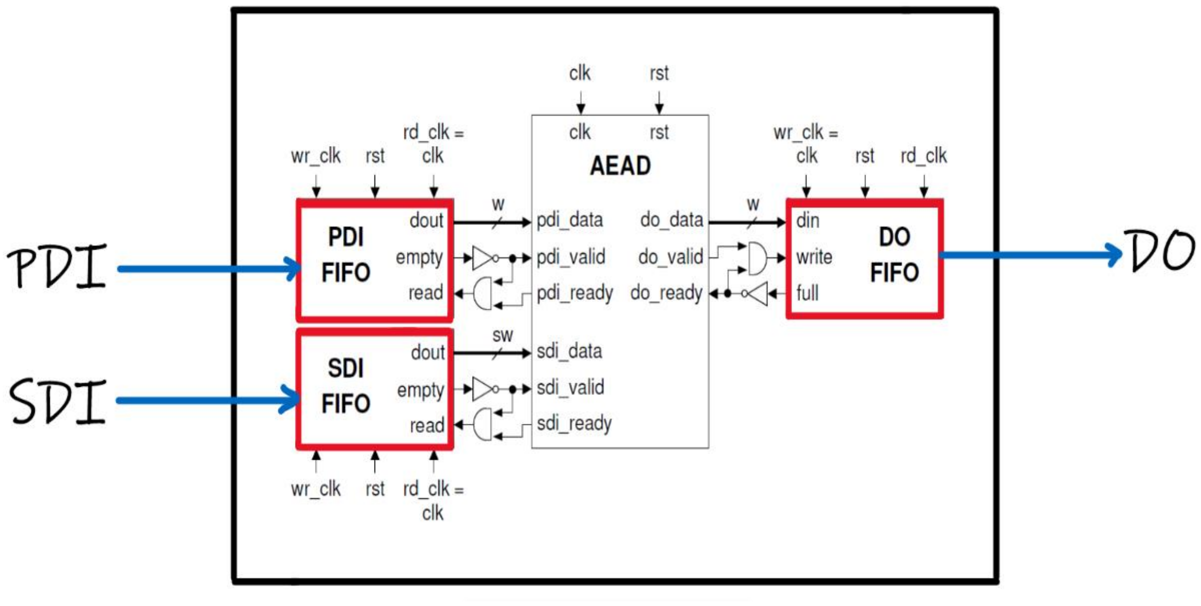


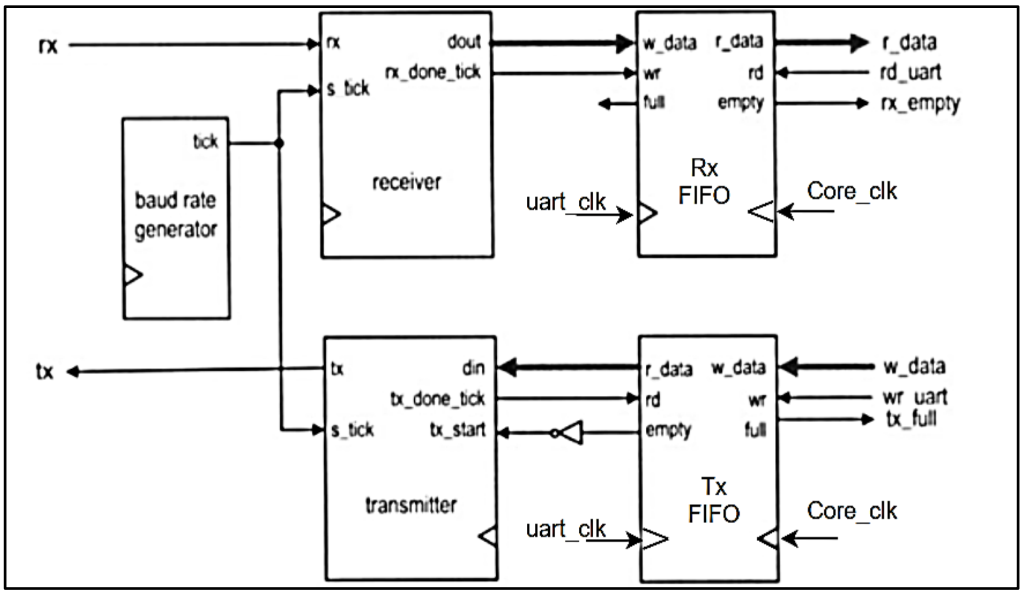
Figure 41: Asynchronous fifo

The Asynchronous fifo basically is a block of memory, with 2 different clocks, no address bus, instead we have two pointers, a read pointer having the address of the next location to be read, and a write pointer having the address of the next location to be written into. We also have 2 important flags, the full flag and the empty flag which show the status of our fifo.

The full flag is synchronized to the writing domain and is calculated by comparing the write pointer in the writing domain with the read pointer from the reading domain, and the empty flag is synchronized to the reading domain and is calculated by comparing the reading pointer in the read domain to the write pointer from the write domain, so to pass these pointers between the 2 domains, a synchronizer is required.

Two-flop synchronizers are used here to synchronize the read pointer to the writing domain and synchronize the read pointer to the writing domain. But we just mentioned that two-flop synchronizers fail to synchronize a multi-bit bus between 2 domains and these pointers are multi-bit. The answer to this is that these pointers should be gray encoded, so only one-bit changes from a count to the next count, so at the other domain we either see the correct change in the synchronized pointer, or we see no change at all. So now that we found our synchronizer, asynchronous fifos were instantiated between the core and the UART and between the core and the security module. After adding the Asynchronous fifos to each of the Security module and the UART module, the connection was as shown.





So now that we have added our synchronizers, it was time we integrated these blocks together on the SoC so that the core can interface with the UART and the security module.

4.3.6 Integrating Core with Peripherals

After finishing the RISC-V IM core's design, the UART peripheral and the hardware security crypto accelerator, it's time to integrate them together on the SoC. For the core to interface with the UART and the security module, the first solution was a shared bus (e.g., AXI) to connect them together with the core being the master and the slaves being the memory, the UART and the security module.

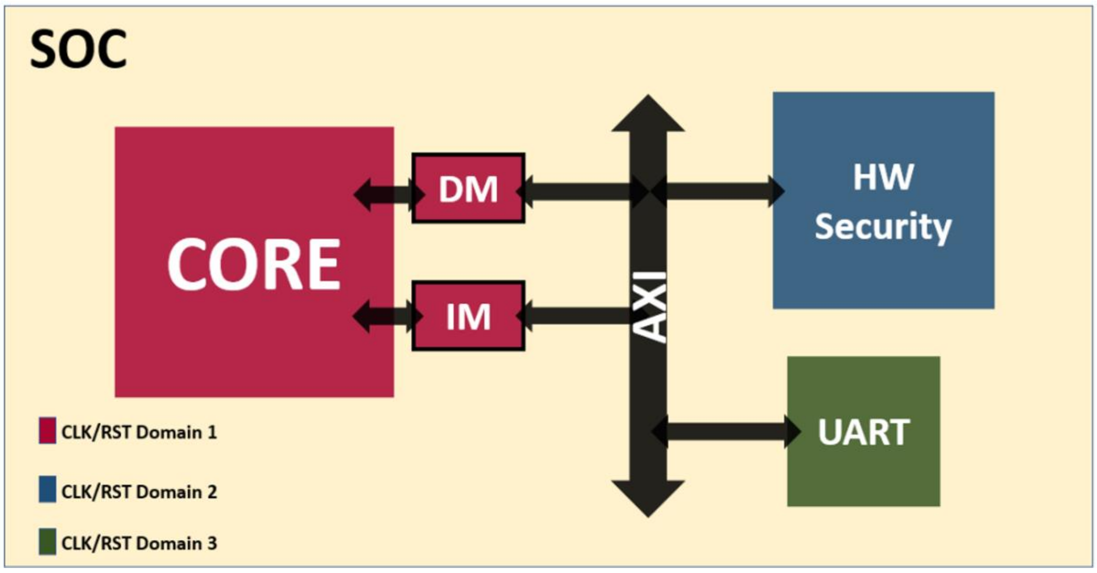
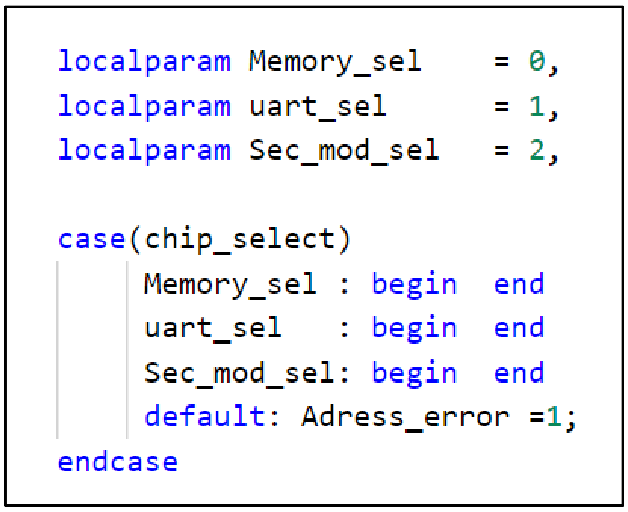


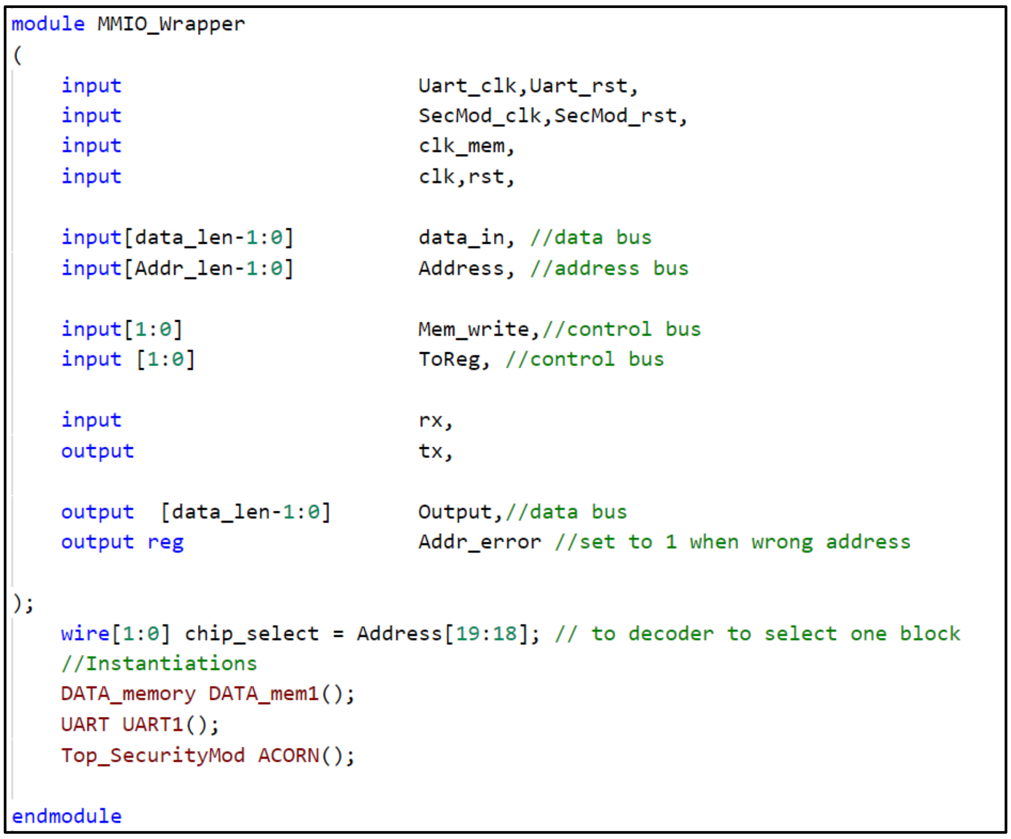
Figure 42: SoC with AXI bus and clock domains

But due to the complexity of this solution, a simpler approach was used, which is the MMIO (Memory Mapped Input Output). The idea of the MMIO is to interface with peripherals on the SoC simply by using Load/store instructions. As mentioned before, we only used 18 bits as address bus for the memory of size 256 KB, 2 more bits were added to the address in the most significant part. These 2 bits were to be given to an address decoder, where the address decoder would output an enable signal which is called a chip select to select either the memory, the UART or the security module to interface with, if a wrong address is entered an address error signal is set to 1.



```
localparam Memory_sel    = 0,
localparam uart_sel      = 1,
localparam Sec_mod_sel   = 2,

case(chip_select)
        Memory_sel : begin  end
        uart_sel   : begin  end
        Sec_mod_sel: begin  end
        default: Adress_error =1;
endcase
```

An MMIO wrapper was designed to instantiate the 3 blocks that the core would interface with as shown:



```
module MMIO_Wrapper
(
    input                           Uart_clk,Uart_rst,
    input                           SecMod_clk,SecMod_rst,
    input                           clk_mem,
    input                           clk,rst,

    input[data_len-1:0]             data_in, //data bus
    input[Addr_len-1:0]             Address, //address bus

    input[1:0]                      Mem_write,//control bus
    input [1:0]                     ToReg, //control bus

    input                           rx,
    output                          tx,

    output  [data_len-1:0]          Output,//data bus
    output reg                      Addr_error //set to 1 when wrong address

);
    wire[1:0] chip_select = Address[19:18]; // to decoder to select one block
    //Instantiations
    DATA_memory DATA_mem1();
    UART UART1();
    Top_SecurityMod ACORN();

endmodule
```

Then a data bus, an address bus and a control bus were connected to the three blocks, where the data bus would carry the required input/output data, the address bus would select a register to write to or read from and the control bus would select either reading or writing and whether we are reading/writing a byte, a half word or a complete word, as shown in figure 41.

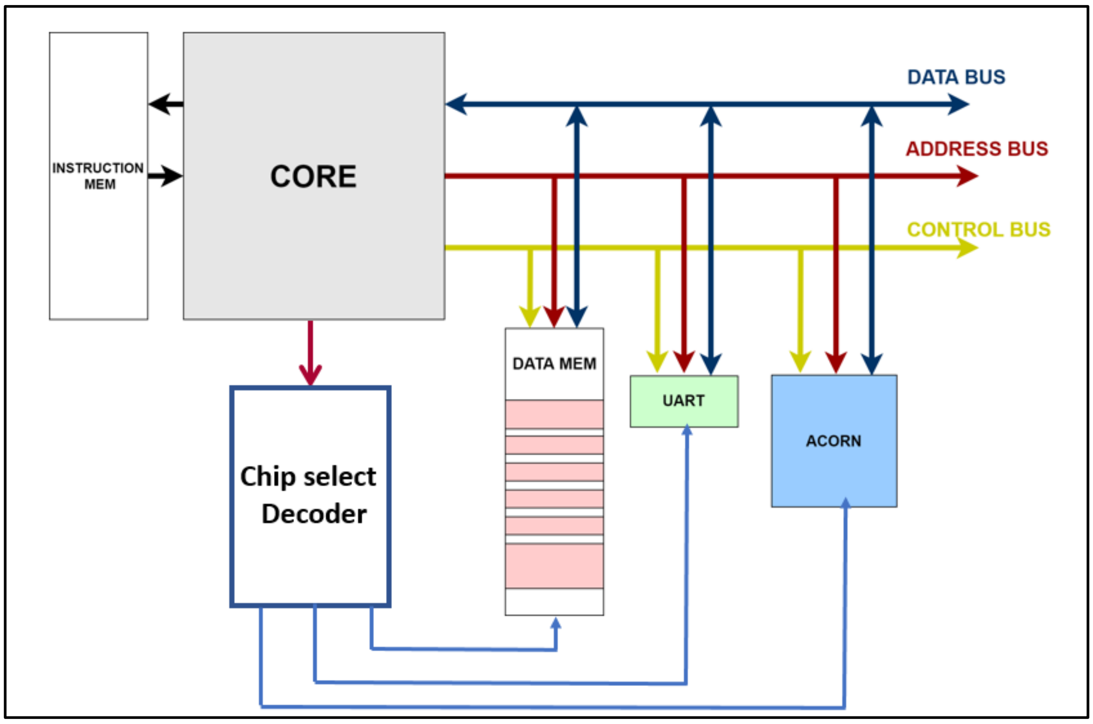


Figure 43: SoC integration using MMIO

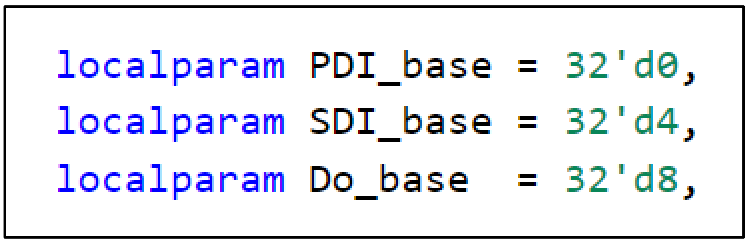
The UART had 3 locations for 3 registers, a data in register, a data out registers and a control and status register (CSR) for controlling the UART peripheral, each register is one byte. The CSR register is coded to be as follows:

	R	R	W	W
4'b0000	Rx_Empty	Tx_full	Rd_UART	Wr_UART

Figure 44: UART CSR

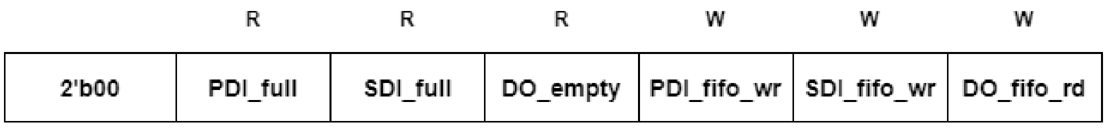
Two Read-only bits were to determine the status of the FIFOs connected to the UART transmitter and receiver, and two Write bits would carry the read and write signals from and to the FIFOs and the core, where the Core is in the writing domain relative to the Tx fifo and in the Reading domain relative to the Rx fifo. The remaining four bits were hardcoded to zero as we will not need them.

The AEAD block as discussed before has 3 ports, the PDI, the SDI and the DO, each is 32-bit long so each had to take 4 locations from our address, starting from a base address, and we would increment to select our desired register in case of byte selectable instructions (e.g. store byte in PDI 2nd byte location).



```
localparam PDI_base = 32'd0,
localparam SDI_base = 32'd4,
localparam Do_base  = 32'd8,
```

A control and status register (CSR) which is one byte. The CSR of the security module was coded as follows:



	R	R	R	W	W	W
2'b00	PDI_full	SDI_full	DO_empty	PDI_fifo_wr	SDI_fifo_wr	DO_fifo_rd

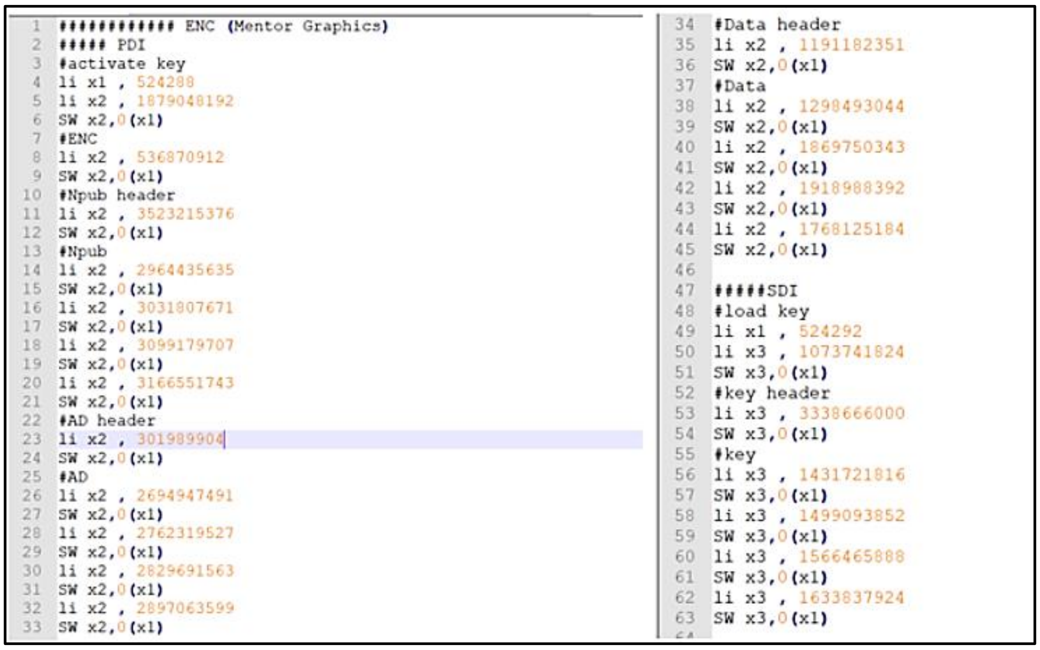
Figure 45: Security module CSR

The Security module has three writable bits which are used to write in the PDI or SDI fifos and read from the DO fifo, and 3 read-only bits to determine the status of each fifo relative to the core, which is in the reading domain in case of the DO fifom and in the writing domain in case of the SDI fifo and PDI fifo. The remaining 2 bits were hard coded to zeros.

Chapter 5: SoC testing and Static verification

5.1 SoC testing

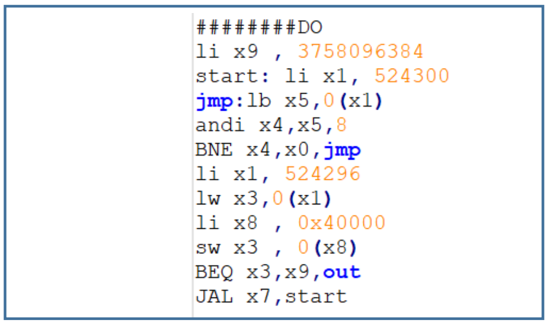
A simple testbench was written using Verilog to verify using waveforms the operation of the SoC after integration. The following Assembly code was written and converted to hex, then loaded into the core's instruction memory.



```
############# ENC (Mentor Graphics)
##### PDI
#activate key
li x1 , 524288
li x2 , 1879048192
SW x2,0(x1)
#ENC
li x2 , 536870912
SW x2,0(x1)
#Npub header
li x2 , 3523215376
SW x2,0(x1)
#Npub
li x2 , 2964435635
SW x2,0(x1)
li x2 , 3031807671
SW x2,0(x1)
li x2 , 3099179707
SW x2,0(x1)
li x2 , 3166551743
SW x2,0(x1)
#AD header
li x2 , 301989904
SW x2,0(x1)
#AD
li x2 , 2694947491
SW x2,0(x1)
li x2 , 2762319527
SW x2,0(x1)
li x2 , 2829691563
SW x2,0(x1)
li x2 , 2897063599
SW x2,0(x1)
#Data header
li x2 , 1191182351
SW x2,0(x1)
#Data
li x2 , 1298493044
SW x2,0(x1)
li x2 , 1869750343
SW x2,0(x1)
li x2 , 1918988392
SW x2,0(x1)
li x2 , 1768125184
SW x2,0(x1)

#####SDI
#load key
li x1 , 524292
li x3 , 1073741824
SW x3,0(x1)
#key header
li x3 , 3338666000
SW x3,0(x1)
#key
li x3 , 1431721816
SW x3,0(x1)
li x3 , 1499093852
SW x3,0(x1)
li x3 , 1566465888
SW x3,0(x1)
li x3 , 1633837924
SW x3,0(x1)
```

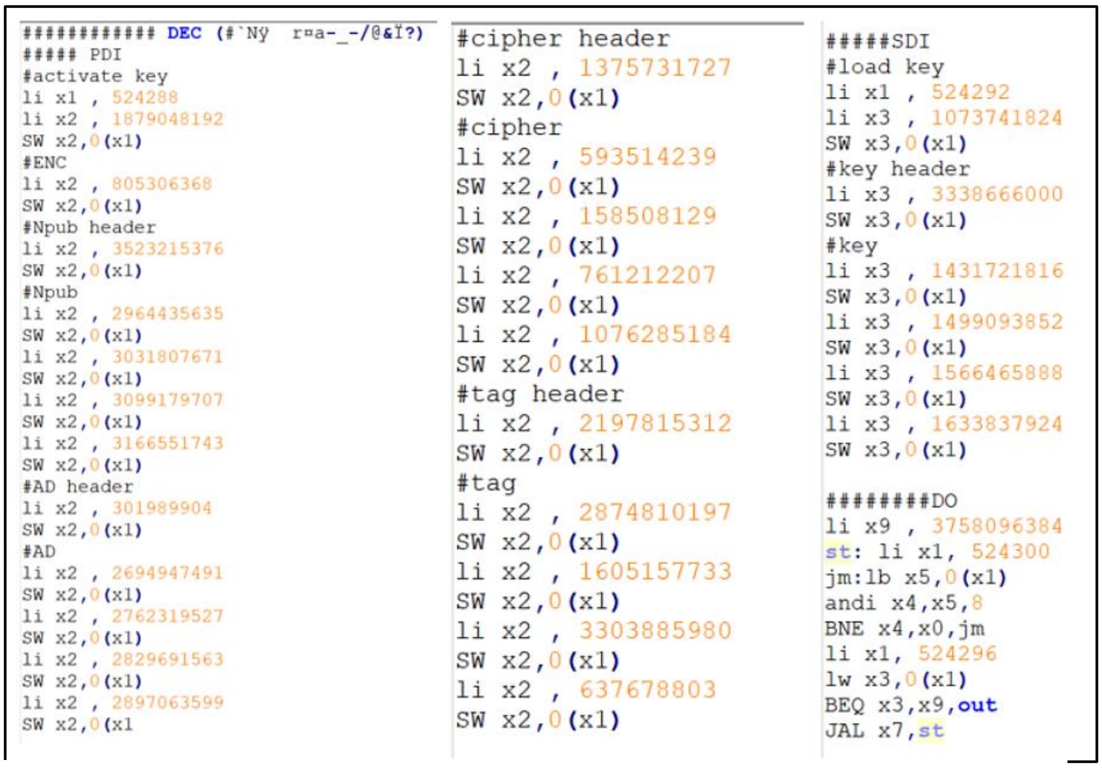
Figure 46: Encryption Assembly code

The AEAD API manual shows how the headers for each of the PDI, and SDI should be entered and the order of entry depending on the size of the words used. The purpose of the simulation was to encrypt the string "Mentor Graphics" and then decrypt it. When converting the string "Mentor Graphics" to hex to enter it as the cipher text to the encryption module, it was "**4d656e746f72204772617068696373**", After encrypting it and the cipher outputted "E0" indicating successful encryption, the resulting string was "**#`Nÿ   r¤a-_-/@&Ï?**", which has absolutely no meaning, and if a hacker tries to decrypt it he won't be able to without the Key. After that we sent this encrypted data over the UART using the following assembly code.



```
########DO
li x9 , 3758096384
start: li x1, 524300
jmp:lb x5,0(x1)
andi x4,x5,8
BNE x4,x0,jmp
li x1, 524296
lw x3,0(x1)
li x8 , 0x40000
sw x3 , 0(x8)
BEQ x3,x9,out
JAL x7,start
```

When entering the encrypted string as the cipher text and testing the decryption, the following assembly code was written:



```
############## DEC (#`Nÿ  r¤a-_-/@&Ï?)
##### PDI
#activate key
li x1 , 524288
li x2 , 1879048192
SW x2,0(x1)
#ENC
li x2 , 805306368
SW x2,0(x1)
#Npub header
li x2 , 3523215376
SW x2,0(x1)
#Npub
li x2 , 2964435635
SW x2,0(x1)
li x2 , 3031807671
SW x2,0(x1)
li x2 , 3099179707
SW x2,0(x1)
li x2 , 3166551743
SW x2,0(x1)
#AD header
li x2 , 301989904
SW x2,0(x1)
#AD
li x2 , 2694947491
SW x2,0(x1)
li x2 , 2762319527
SW x2,0(x1)
li x2 , 2829691563
SW x2,0(x1)
li x2 , 2897063599
SW x2,0(x1
```

```
#cipher header
li x2 , 1375731727
SW x2,0(x1)
#cipher
li x2 , 593514239
SW x2,0(x1)
li x2 , 158508129
SW x2,0(x1)
li x2 , 761212207
SW x2,0(x1)
li x2 , 1076285184
SW x2,0(x1)
#tag header
li x2 , 2197815312
SW x2,0(x1)
#tag
li x2 , 2874810197
SW x2,0(x1)
li x2 , 1605157733
SW x2,0(x1)
li x2 , 3303885980
SW x2,0(x1)
li x2 , 637678803
SW x2,0(x1)
```

```
#####SDI
#load key
li x1 , 524292
li x3 , 1073741824
SW x3,0(x1)
#key header
li x3 , 3338666000
SW x3,0(x1)
#key
li x3 , 1431721816
SW x3,0(x1)
li x3 , 1499093852
SW x3,0(x1)
li x3 , 1566465888
SW x3,0(x1)
li x3 , 1633837924
SW x3,0(x1)

########DO
li x9 , 3758096384
st: li x1, 524300
jm:lb x5,0(x1)
andi x4,x5,8
BNE x4,x0,jm
li x1, 524296
lw x3,0(x1)
BEQ x3,x9,out
JAL x7,st
```

Figure 47: Decryption Assembly code

The cipher outputted "E0" indicating successful decryption and when converting the resulting hex to string, the decoded string was "Mentor graphics". As we can see, both the encryption and decryption were done using the same key, this is because the Authenticated encryption ciphers are Symmetric key ciphers, or sometimes called shared key ciphers, which means that both encryption and decryption are done using the same key, so the key itself is at risk of being stolen. Another type of encryption uses asymmetric key or public key encryption, where the encryption and decryption are performed using different keys, ensuring more security, but using this technique is slower than the symmetric encryption. However, a hybrid of symmetric and asymmetric key ciphers are being developed where only the key is encrypted using asymmetric encryption and the data itself is encrypted using symmetric encryption.

5.2 Static Vs Dynamic Functional Verification

Throughout our design, static verification checks were being performed. Static verification is extremely useful to discover errors that may result in undefined or wrong behavior that might occur in circuit simulation, or any bad RTL coding that might result in a synthesis error in the early stages of the design thus saving time. The main difference between static verification tools (like LINT tool, CDC tool and formal verification tools) and dynamic verification tools is that static verification doesn't need actual data propagation or actual stimulus with a reference model to discover errors in the outputs, instead, static analysis uses lookup tables, structural analysis, state traversal and mathematical models to predict the output and the potential problems without the need of actual data.

5.2.1 Questa LINT

LINTING is one of the basic static checks done for modern designs, it points out errors as in width issues, synthesis issues, RTL issues, Clock issues (sequential block with multi clocks), reset issues (signal used as both synchronous and asynchronous resets) and Naming convention checks. After running Questa LINT on our design, the results were organized into a table with the ones with the highest priority (errors) in the top, followed by the warnings. Some of the warnings were waived.

5.2.3 Questa CDC

Having multiple clock domains in our design, Traditional methods like simulation and static timing analysis alone are not sufficient to verify that the data is transferred consistently and reliably across clock domains. Hence, new verification methodologies are required. Questa CDC was used to detect the metastability in the design and whether the added synchronizers solved them or not. Questa CDC identifies errors using structural analysis to recognize clock-domains, synchronizers, and low power structures via the Unified Power Format (UPF). It generates assertions for protocol verification along with metastability models for reconvergence verification as shown in the figure below.

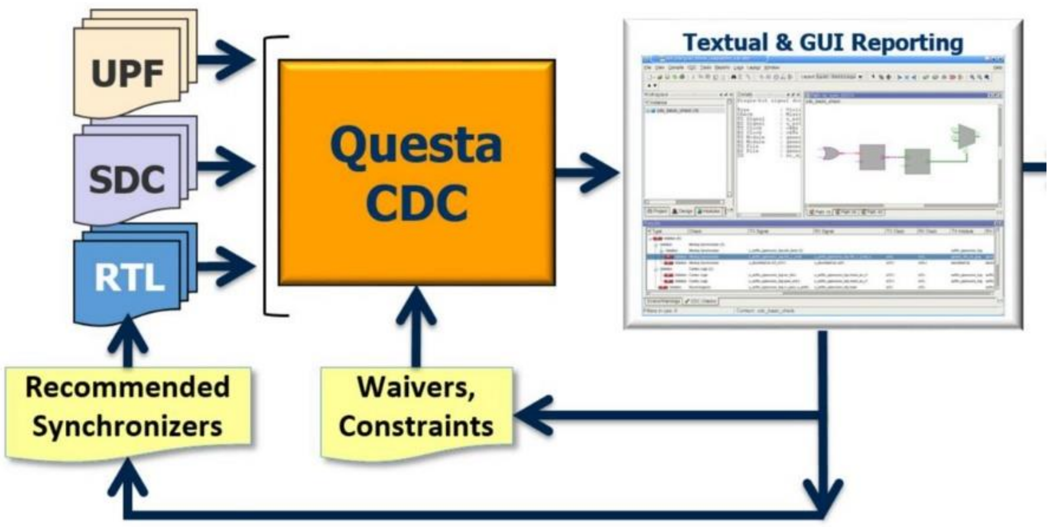
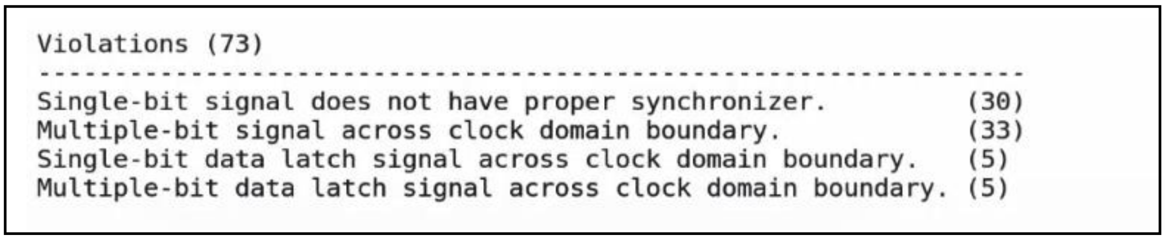


Figure 48: Questa CDC

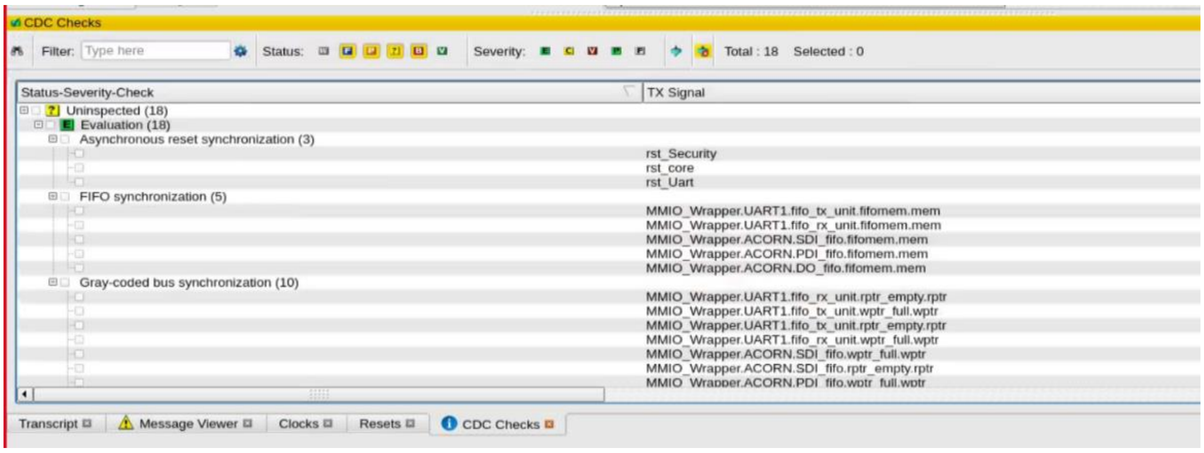
After running Questa CDC, the output reports showed the number of clock groups found, the number of CDC violations and their types (single bit or multi bit), design information and port domain information. When examining the violations, the tool could suggest synchronizers to solve each type of violation.



```
Violations (73)
-----------------------------------------------------------------
Single-bit signal does not have proper synchronizer.        (30)
Multiple-bit signal across clock domain boundary.           (33)
Single-bit data latch signal across clock domain boundary.  (5)
Multiple-bit data latch signal across clock domain boundary. (5)
```

Figure 49: Questa CDC violations

After adding the synchronizers and rerunning Questa CDC, all the violations due to CDC and RDC were solved and the SoC was ready for the dynamic verification.



```
CDC Checks
Filter: Type here   Status:   Severity:   Total : 18   Selected : 0

Status-Severity-Check                          TX Signal
Uninspected (18)
  Evaluation (18)
    Asynchronous reset synchronization (3)
                                               rst_Security
                                               rst_core
                                               rst_Uart
    FIFO synchronization (5)
                                               MMIO_Wrapper.UART1.fifo_tx_unit.fifomem.mem
                                               MMIO_Wrapper.UART1.fifo_rx_unit.fifomem.mem
                                               MMIO_Wrapper.ACORN.SDI_fifo.fifomem.mem
                                               MMIO_Wrapper.ACORN.PDI_fifo.fifomem.mem
                                               MMIO_Wrapper.ACORN.DO_fifo.fifomem.mem
    Gray-coded bus synchronization (10)
                                               MMIO_Wrapper.UART1.fifo_rx_unit.rptr_empty.rptr
                                               MMIO_Wrapper.UART1.fifo_tx_unit.wptr_full.wptr
                                               MMIO_Wrapper.UART1.fifo_tx_unit.rptr_empty.rptr
                                               MMIO_Wrapper.UART1.fifo_rx_unit.wptr_full.wptr
                                               MMIO_Wrapper.ACORN.SDI_fifo.wptr_full.wptr
                                               MMIO_Wrapper.ACORN.SDI_fifo.rptr_empty.rptr
                                               MMIO_Wrapper.ACORN.PDI_fifo.wptr_full.wptr

Transcript | Message Viewer | Clocks | Resets | CDC Checks
```

Figure 50: Questa CDC results

Chapter 6: Universal Verification Methodology (UVM)

6.1 History of UVM

Design Verification is simply the process of checking that a given design parameter correctly implements the target, specifications, and the required functionality. Traditionally, 70% of a chip development cycle is dedicated to design verification. Based on that, the verification to design team ratio ranges from 2:1 to 3:1; that major overhead in the whole process led to a new trend in the verification arena: striving towards standardized, and reusable test benches.

That is when newer methodologies have been introduced like the Open Verification Methodology (OVM) and the Verification Methodology Manual (VMM), both have been fine for a while, however the industry needs a non-proprietary approach; thus, finally in 2011, major technology giants joined together through Accellera and created the Universal Verification Methodology (UVM). UVM has opened new horizons in the verification world, and its highly configurable features have made the generic proposal really possible; as the old saying goes: "You can never go wrong with an object-oriented-based line of code". Normally a System on Chip (SOC) can be verified effectively using a simulation testbench that provides data to the SoC inputs and checks resulting data at the SoC outputs. The problem is that running all the possible testing scenarios is computationally impossible. On the other hand, a modern testbench-based verification environment automatically generates randomized stimulus for the SoC inputs under control of user-specified constraints and checks the results of each test automatically. UVM is the best verification approach that has been created to develop 2 constrained-random testbenches in a uniform fashion and to permit limited reuse of testbench components. The most common term in verification is known as functionality testing or functional verification which is the process of demonstrating functional correctness of a processor design with respect to its specifications, this process is preceded by creating a verification plan that defines: which properties and functionalities need to be verified, different methods and approaches that will be used in processor testing, the expected behavior of an appropriate design, defining functional coverage models and functional specifications of the verification, and finally the testing strategy; such major decisions must be taken in the verification planning phase . Due to complex aspects of an IC design, these processes tend to be very challenging; during the past decade alone, average time spent by verification engineers to verily the complete functionality of their designs wasted more than 60 percent of the total design time. Even developers of smaller chips and FPGAs designs are facing difficulties with former verification approaches. The wanted goal of verification is becoming more difficult to achieve using conventional verification techniques, and hence solving this issue requires a detailed review of common testing methodology. Directed testing was more convenient for testing single functionalities, however, it is hard to hit more complex scenarios using only directed testing. On the other hand, constrained-random verification (CRV) can be very efficient in tackling processor verification challenges, such as: complex instruction sets, multiple pipeline-stages, in6 order or out of order execution strategies, instruction parallelism, and multi precision operations. The most important module of a CRV environment is the test-case generator, which plays a very significant role in most of the recent approaches towards developing automated processor verification environments. A test-case generator generates a large set of valid test cases in a pseudo-random way, controlled and guided by constrained randomness. The development of such test generators has started to get the attention of functional verification engineers, and researchers since the early 2000s. However, the development of these generators has been categorized as a software problem due to poor and weak features of Hardware Description Languages (HDLs) available back then in terms of verification and software, like Verilog and VHDL. However, recent efforts have been spent towards the utilization of SystemVerilog features as a Hardware Verification Language (HVL) to improve stimulus generation quality; and then UVM gradually dominated the verification world, as it covers these needs. UVM is a powerful verification methodology that was designed to be able to verify a variety of different design sizes and design types that could be in Verilog, SystemVerilog, VHDL, and SystemC code. It is an open source SystemVerilog library allowing creation of flexible, reusable verification components and assembling powerful test environments utilizing constrained random stimulus generation and functional coverage models. It is based upon the three C's of random verification:

Checkers: If the stimulus is automated, in addition you must write self-checking test benches in System-Verilog.

Coverage: The question "Are we done yet?" have to be answered, as in addition is known as "Functional Coverage", it is about recording the progress during a verification run and identifying how thoroughly the design have been exercised.

Constraints: What if holes have been covered, or if the design have not been exercised thoroughly enough. That is where constraints come into play, the constraints have to be increased on those random vectors in order to increase test coverage. A UVM test bench is composed of verification components that are encapsulated, reusable, ready-to-use, and configurable elements, checking an interface protocol, a design sub-module, or a full system. The architecture of each component is logical. It includes a complete set of elements enabling the stimulation, check and collection of coverage information related to the specific protocol or design. This test bench instantiates the Design under Test module and the UVM Test class, then configures all connections between them. Module-based components are instantiated under the UVM test bench as well. The UVM Test is dynamically instantiated at run-time, allowing the UVM test bench to be compiled once and run with many different tests.

6.2 Introduction to UVM

The Universal Verification Methodology (UVM) is a set of guidelines for creating and reusing verification environments. It's a set of class libraries developed using System Verilog's syntax and semantics, and it's now an IEEE standard. UVM's major goal is to help creating modular, reusable, and scalable testbench frameworks that may be used to validate numerous designs usually named DUTs (Design Under Test).

The UVM class library includes generic utilities such as configuration databases, TLM, and component hierarchy. It adds a layer of abstraction by assigning roles to each component. A driver class object, for example, will be responsible for driving signals to the design under test (DUT), whereas a monitor will merely monitor the DUT signals but not drive them. We will talk about UVM main components and objects in details in section 10.9.

6.3 UVM classes

The basic building blocks of any verification environment are components (drivers, sequencers, monitors … etc.) and objects (sequences and transactions). Transactions (A class that contain actual data) are used to communicate between objects and components in the environment. Most of the classes in UVM are derived (by inheritance) from a set of core classes and adding onto it more functions than the base functions implemented in these base classes to form your own verification environment.

6.4 UVM factory

UVM uses what is called UVM factory to build its components and objects. Each component or object created gets registered in the factory using UVM factory macros (`uvm_component_utils and `uvm_object_utils), then the object or component is created using the type_id::Create() function, this is called deferred construction as the actual instantiation of the object or component class is done later when calling the new() function of the class. The main purpose of the UVM factory is making the Verification environment as flexible as possible, where if we want to modify the environment with a new version of any class, we don't have to modify classes or instances from the old version, we can simply override the old instances with our new class of the new version we want. Overriding In factory is done using methods such as

set_inst_override_by_type()	set_inst_override_by_name()
set_type_override_by_type()	set_type_override_by_name()

6.5 UVM phases

UVM operation is divided into phases to provide a synchronization mechanism in the lifetime of the simulation. They start from "run_test" and are divided into:

1. Build time phases
2. Run time phases
3. Clean-Up phases

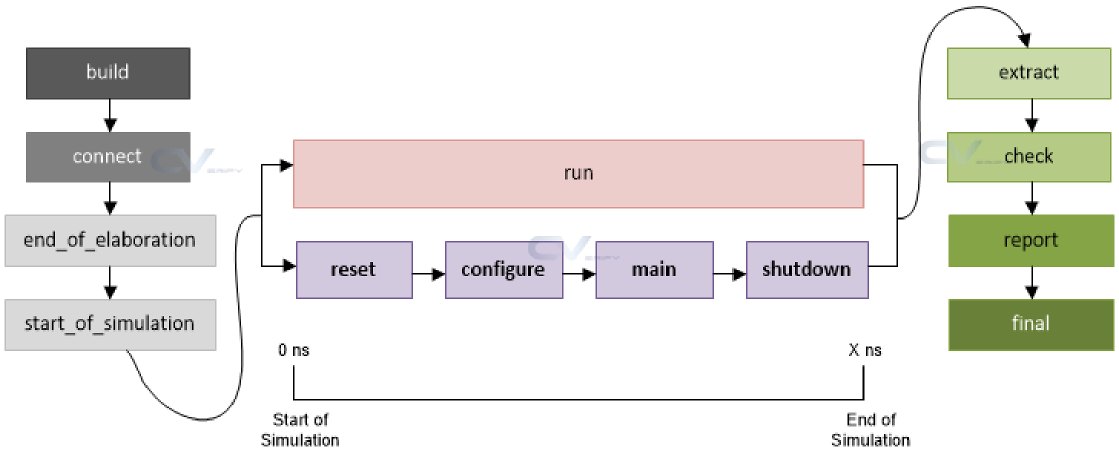


Figure 51: UVM phases

Below is the function of some of the important phases:

A- Build time phases

1- Build phase: A function so it doesn't consume time, used to build testbench components and create their instances, executed top down, which means from the top of the class hierarchy towards the bottom.

2- Connect phase: A function so it doesn't consume time, used to connect between different testbench components via TLM ports, executes bottom up.

B- Run time phase: Actual simulation that consumes time therefore a task, happens in this UVM phase and runs parallel to other UVM run-time phases.

C- Clean-Up phases: Used to extract outputs from scoreboards and perform checks on this and display the results and reports and perform final checks before ending the simulation.

6.6 UVM Environment

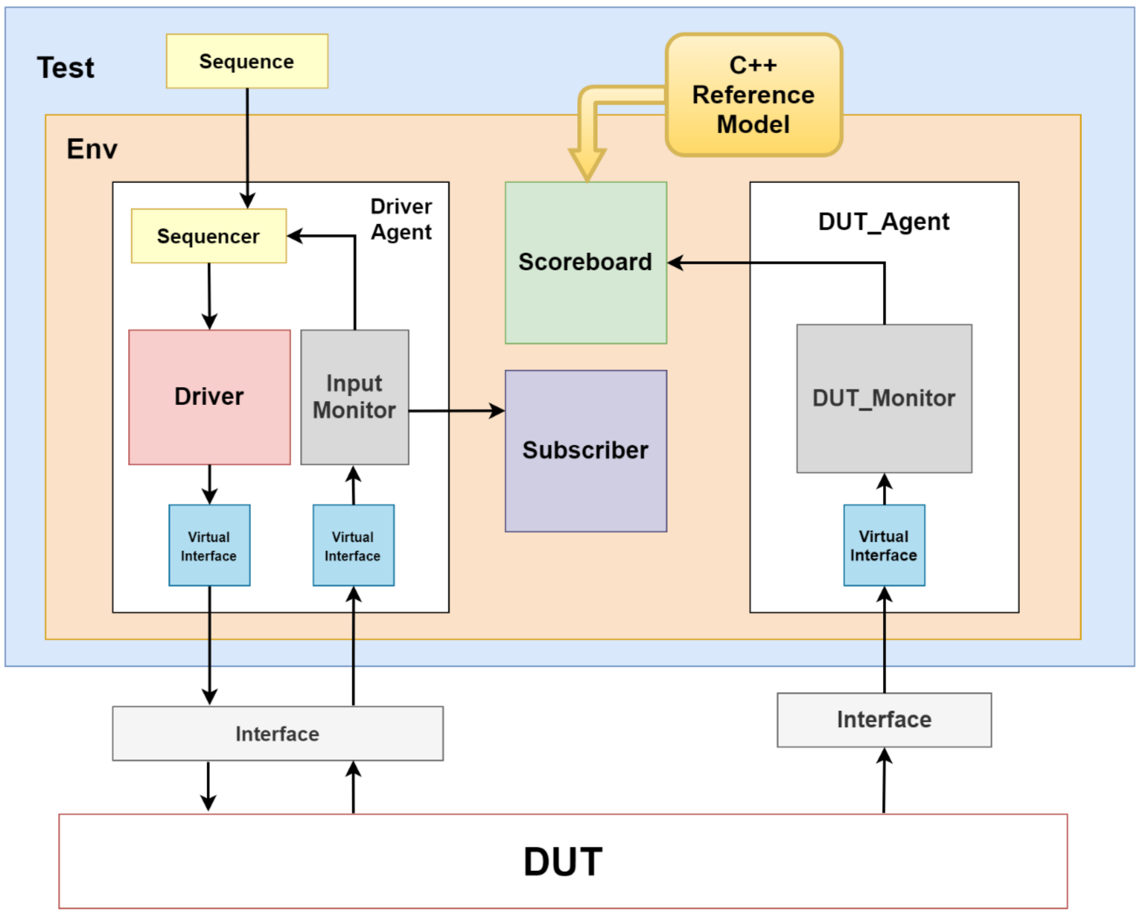


Figure 52: UVM environment

6.7 DUT Wrapper

The DUT in the UVM environment shown in figure 52 is actually a wrapper encapsulating the device under verification (which is the RISC-V core implemented in section 2) and the RAM (Data memory) shown in figure 53. Even if the RAM is not to be verified but it is an essential component to make the processor function correctly as the core contains load and store instructions, so the memory is required to be inserted alongside the core. Inside the wrapper signals to be read and written from and to memory are connected to the core. The wrapper allows having a clear interface in which only the signals to be driven by UVM framework are available.

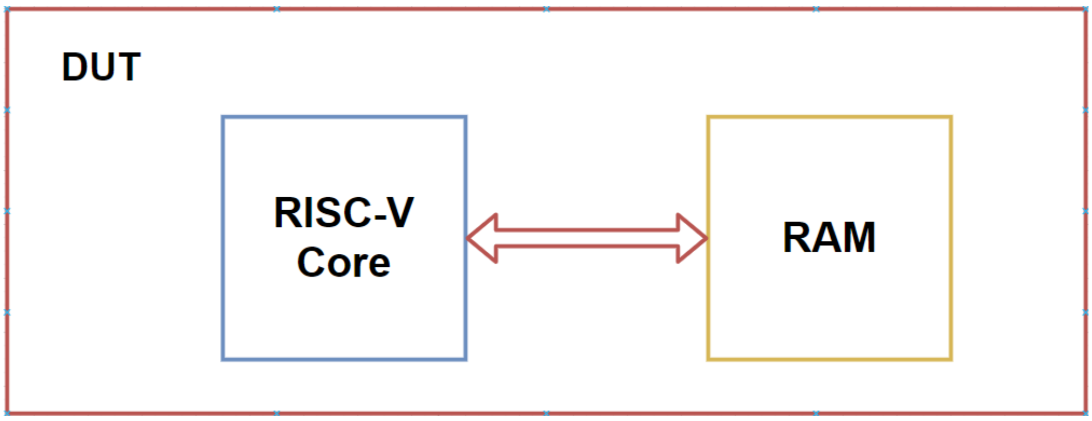


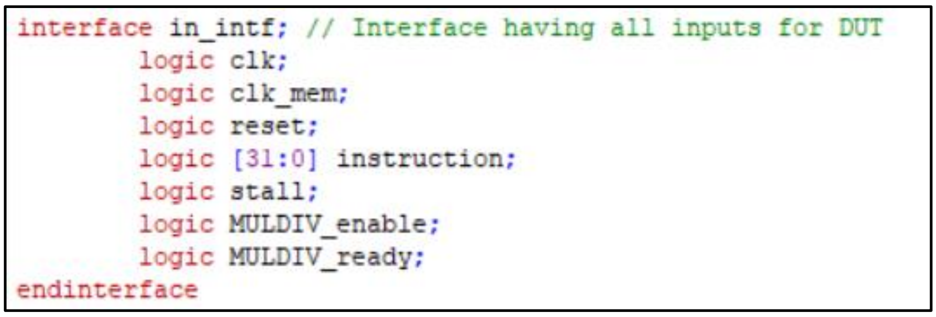
Figure 53: UVM DUT Wrapper

6.8 UVM interfaces

As the DUT is considered static in the UVM, the communication between testbench and the DUT cannot be directly done as the users do in classic testbenches. In UVM, Virtual interface feature is used, it represents a collection of signals used to drive and monitor the DUT from the testbench. Using this approach, the UVM can access the DUT signals through the virtual interface and vice versa. Each UVM component that needs to monitor or drive interface signals must create a virtual interface instance of the interface and obtain the interface's reference from the interface database of UVM configurations. There are two types of interfaces:

- **Input interface**

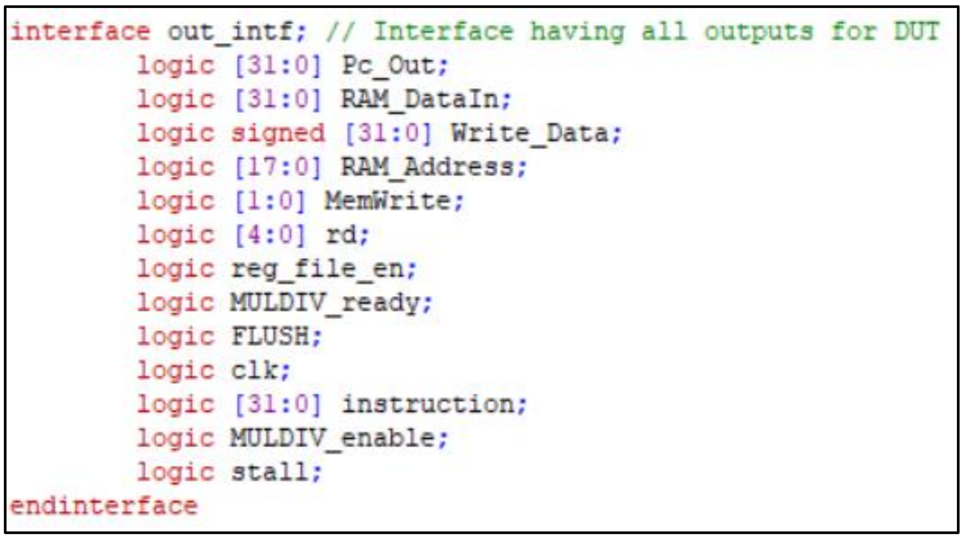
  The input interface, described in “in_intf” (connected with Driver Agent components) and contains a set of signals to be driven and another set of signals to be monitored by input monitor. The following code snippet is the signals used in the input interface.



```
interface in_intf; // Interface having all inputs for DUT
        logic clk;
        logic clk_mem;
        logic reset;
        logic [31:0] instruction;
        logic stall;
        logic MULDIV_enable;
        logic MULDIV_ready;
endinterface
```

- **Output interface**

  The output interface “out_intf” (connected with DUT_Agent components) is used to collect internal signals of the DUT. It represents the collection of the actual results provided by the DUT after the operations is completed. Here signals are going only from the DUT to the UVM. The following code snippet is the signals used in the output interface.

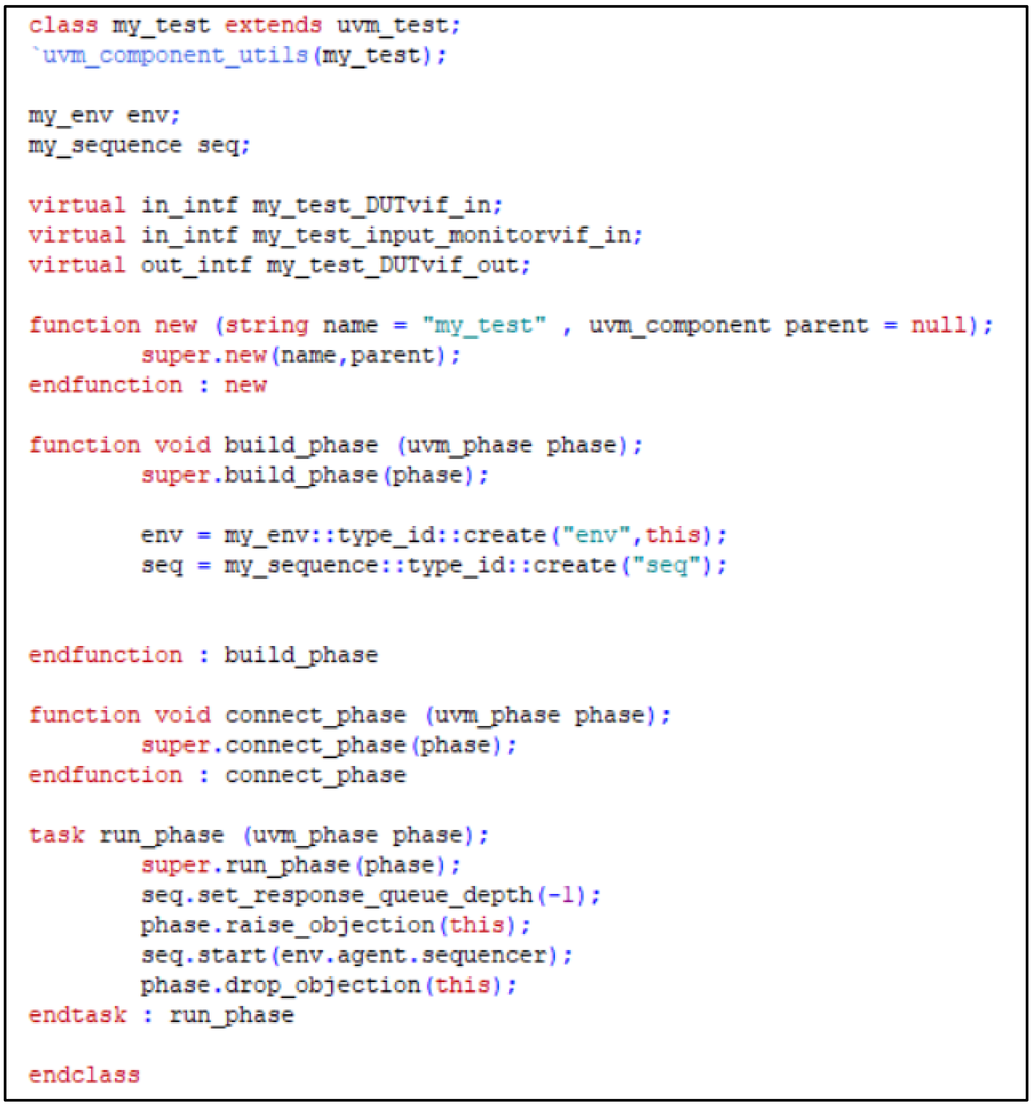
```
interface out_intf; // Interface having all outputs for DUT
        logic [31:0] Pc_Out;
        logic [31:0] RAM_DataIn;
        logic signed [31:0] Write_Data;
        logic [17:0] RAM_Address;
        logic [1:0] MemWrite;
        logic [4:0] rd;
        logic reg_file_en;
        logic MULDIV_ready;
        logic FLUSH;
        logic clk;
        logic [31:0] instruction;
        logic MULDIV_enable;
        logic stall;
endinterface
```

  All these signals are sampled by the output monitor and are sent to the scoreboard to check the results with the C++ model. Write_Data, rd and reg_file_en are used to check the correctness of the R-Type, I-Type and U-Type instructions as they only write their results as data in the Register File. RAM_DataIn, RAM_Address and MemWrite are used to check the correctness of the S-Type instructions as they only write their results into the Data Memory. PC_Out and FLUSH are used to check the correctness of J-Type and B-Type instructions. Stall, MULDIV_ready and MULDIV_enable are used to prevent the output monitor from sampling the output of the DUT when the core is stalled or in multiplication or division process.

6.9 UVM components

6.9.1 Test

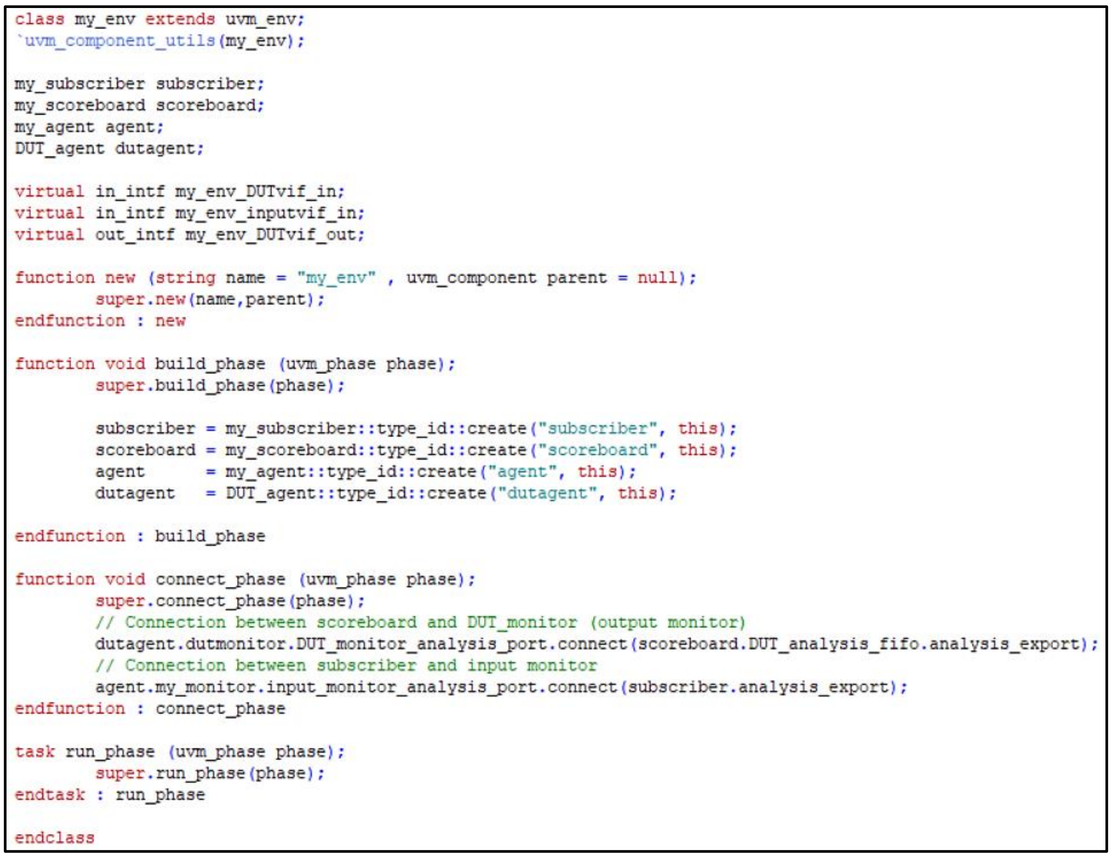
The highest level UVM component in the environment is the UVM test. It is responsible for creating the top-level environment. It provides stimulus to the DUT by invoking UVM sequences through the environment to the DUT. The following is a code snippet from the test class.



```
class my_test extends uvm_test;
`uvm_component_utils(my_test);

my_env env;
my_sequence seq;

virtual in_intf my_test_DUTvif_in;
virtual in_intf my_test_input_monitorvif_in;
virtual out_intf my_test_DUTvif_out;

function new (string name = "my_test" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);

        env = my_env::type_id::create("env",this);
        seq = my_sequence::type_id::create("seq");


endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
endfunction : connect_phase

task run_phase (uvm_phase phase);
        super.run_phase(phase);
        seq.set_response_queue_depth(-1);
        phase.raise_objection(this);
        seq.start(env.agent.sequencer);
        phase.drop_objection(this);
endtask : run_phase

endclass
```

6.9.2 Environment

A UVM environment contains multiple, reusable verification components and defines their default configuration as required by the application. For example, a UVM environment may have multiple agents for different interfaces, a common scoreboard, and a functional coverage collector. The following is a code snippet from the environment class.



```
class my_env extends uvm_env;
`uvm_component_utils(my_env);

my_subscriber subscriber;
my_scoreboard scoreboard;
my_agent agent;
DUT_agent dutagent;

virtual in_intf my_env_DUTvif_in;
virtual in_intf my_env_inputvif_in;
virtual out_intf my_env_DUTvif_out;

function new (string name = "my_env" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);

        subscriber = my_subscriber::type_id::create("subscriber", this);
        scoreboard = my_scoreboard::type_id::create("scoreboard", this);
        agent      = my_agent::type_id::create("agent", this);
        dutagent   = DUT_agent::type_id::create("dutagent", this);

endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
        // Connection between scoreboard and DUT_monitor (output monitor)
        dutagent.dutmonitor.DUT_monitor_analysis_port.connect(scoreboard.DUT_analysis_fifo.analysis_export);
        // Connection between subscriber and input monitor
        agent.my_monitor.input_monitor_analysis_port.connect(subscriber.analysis_export);
endfunction : connect_phase

task run_phase (uvm_phase phase);
        super.run_phase(phase);
endtask : run_phase

endclass
```

6.9.3 Agent

An agent encapsulates the sequencer, driver and monitor into a single entity by instantiating and linking the components together via TLM interfaces. There are different kinds of agents:

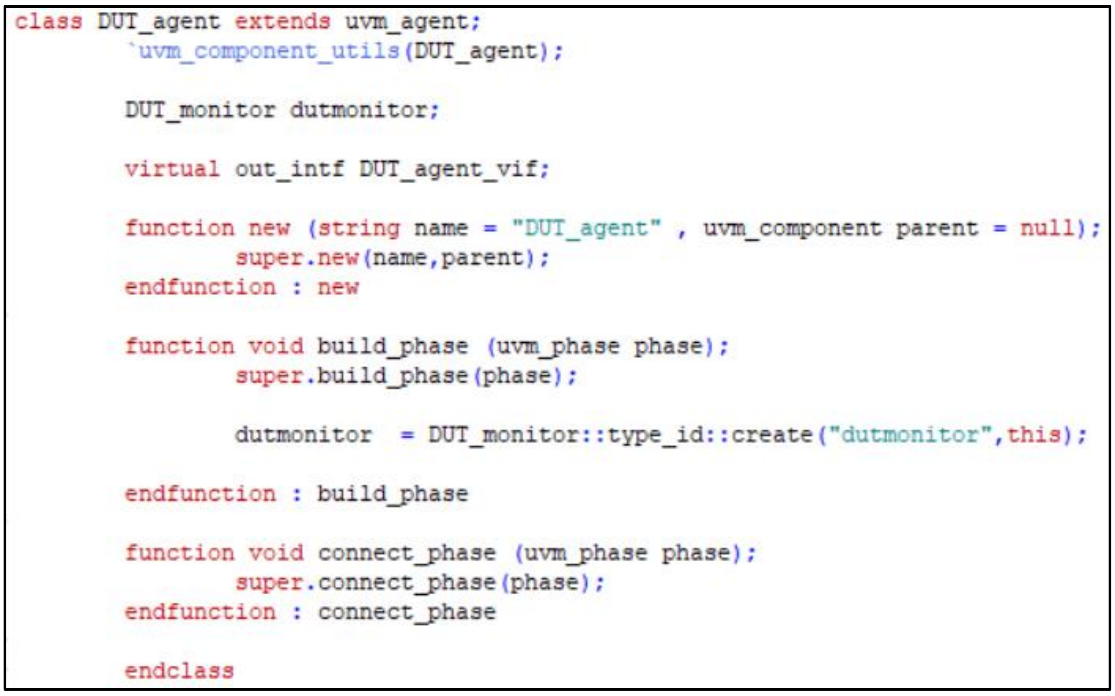
- **Active agent:**
  It contains the driver, sequencer and the monitor. It enables the driver to drive the data to the DUT

- **Passive agent (DUT_Agent):**
  It contains only the monitor which sense data from the DUT, no data to be driven to the DUT.

- **Reactive agent (Driver Agent):**

  They are same as active agents, but the stimulus driven to the DUT by the driver is controlled based on past stimulus or current response from the DUT

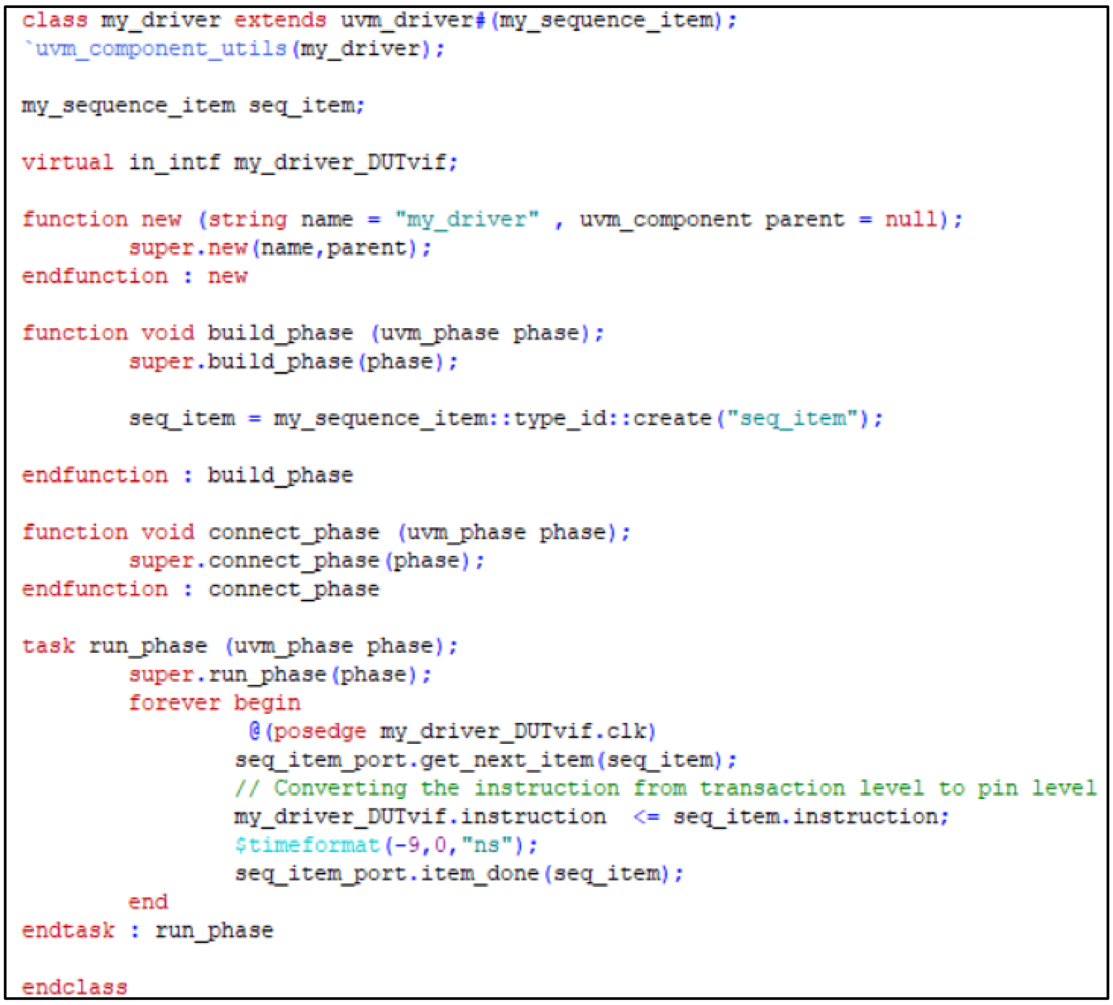
The following is a code snippet from the DUT_Agent class.



```
class DUT_agent extends uvm_agent;
        `uvm_component_utils(DUT_agent);

        DUT_monitor dutmonitor;

        virtual out_intf DUT_agent_vif;

        function new (string name = "DUT_agent" , uvm_component parent = null);
                super.new(name,parent);
        endfunction : new

        function void build_phase (uvm_phase phase);
                super.build_phase(phase);

                dutmonitor  = DUT_monitor::type_id::create("dutmonitor",this);

        endfunction : build_phase

        function void connect_phase (uvm_phase phase);
                super.connect_phase(phase);
        endfunction : connect_phase

        endclass
```

6.9.4 Driver

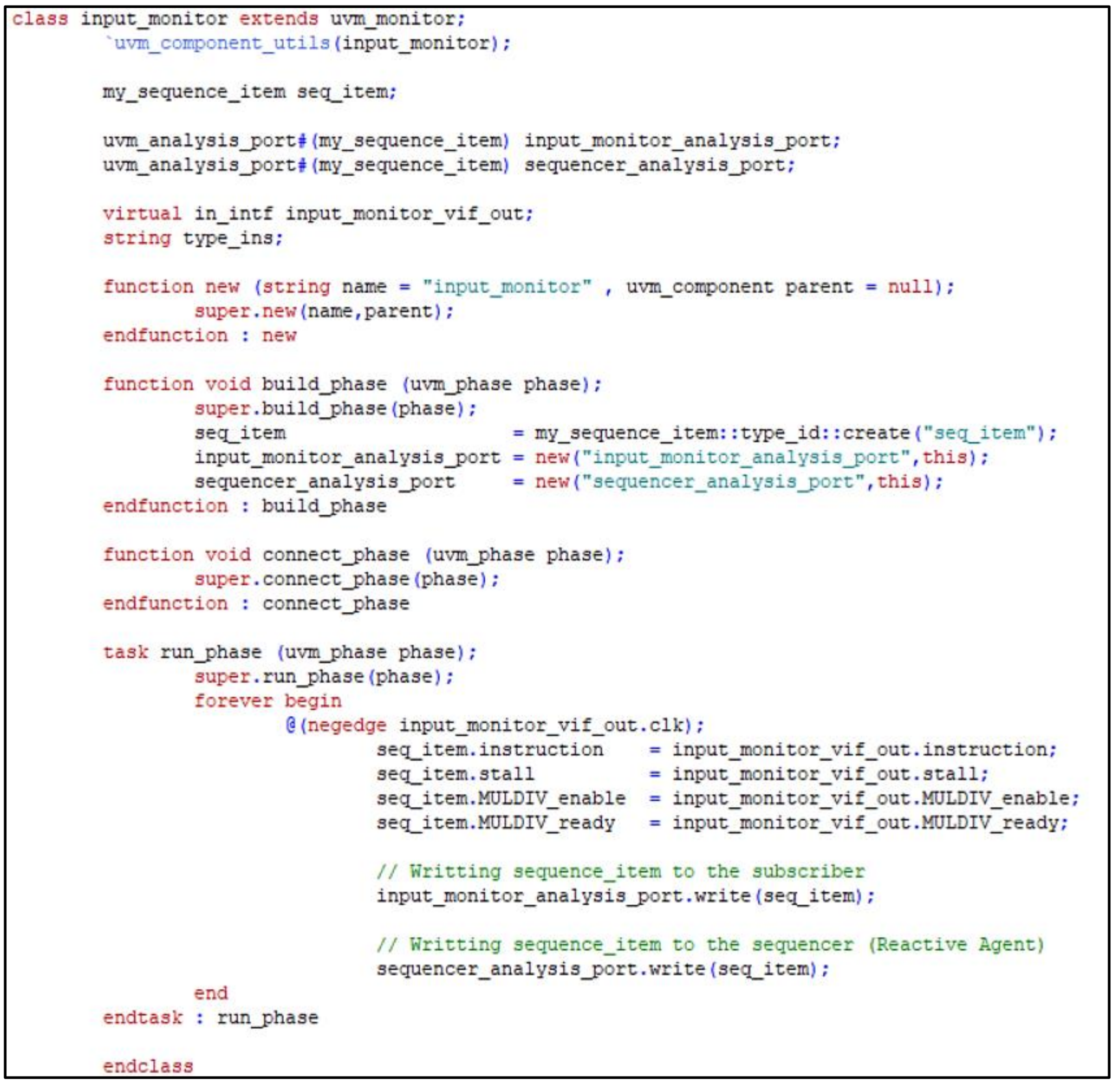
The driver is responsible for receiving sequence items (transactions) from the sequencer and driving it to the DUT. Thus transaction-level stimulus is converted to pin-level stimulus. To accept transactions from the sequencer, it is connected to it through a TLM connection. The following is a code snippet from the Driver class.



```
class my_driver extends uvm_driver#(my_sequence_item);
`uvm_component_utils(my_driver);

my_sequence_item seq_item;

virtual in_intf my_driver_DUTvif;

function new (string name = "my_driver" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);

        seq_item = my_sequence_item::type_id::create("seq_item");

endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
endfunction : connect_phase

task run_phase (uvm_phase phase);
        super.run_phase(phase);
        forever begin
                 @(posedge my_driver_DUTvif.clk)
                seq_item_port.get_next_item(seq_item);
                // Converting the instruction from transaction level to pin level
                my_driver_DUTvif.instruction  <= seq_item.instruction;
                $timeformat(-9,0,"ns");
                seq_item_port.item_done(seq_item);
        end
endtask : run_phase

endclass
```

6.9.5 Monitor

It captures signals from the DUT and converts them from pin level to transaction level, which may subsequently be transmitted to other components in the environment. Monitors could be input monitors which has a crucial importance as the scoreboard need to know if the transactions sent to the DUT has been received correctly. Monitors could also be output monitors which is used to collect signals which represent the actual results of the instructions sent by the driver. The following is a code snippet from input monitor class.

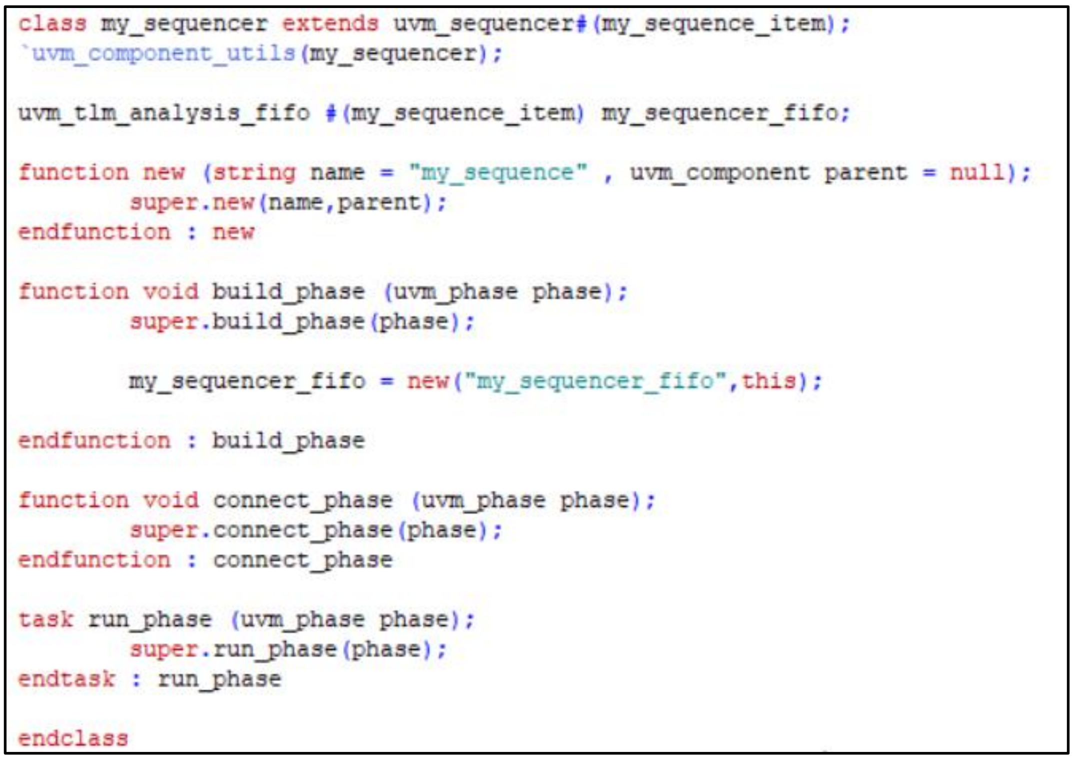


```
class input_monitor extends uvm_monitor;
        `uvm_component_utils(input_monitor);

        my_sequence_item seq_item;

        uvm_analysis_port#(my_sequence_item) input_monitor_analysis_port;
        uvm_analysis_port#(my_sequence_item) sequencer_analysis_port;

        virtual in_intf input_monitor_vif_out;
        string type_ins;

        function new (string name = "input_monitor" , uvm_component parent = null);
                super.new(name,parent);
        endfunction : new

        function void build_phase (uvm_phase phase);
                super.build_phase(phase);
                seq_item                    = my_sequence_item::type_id::create("seq_item");
                input_monitor_analysis_port = new("input_monitor_analysis_port",this);
                sequencer_analysis_port     = new("sequencer_analysis_port",this);
        endfunction : build_phase

        function void connect_phase (uvm_phase phase);
                super.connect_phase(phase);
        endfunction : connect_phase

        task run_phase (uvm_phase phase);
                super.run_phase(phase);
                forever begin
                        @(negedge input_monitor_vif_out.clk);
                                seq_item.instruction    = input_monitor_vif_out.instruction;
                                seq_item.stall          = input_monitor_vif_out.stall;
                                seq_item.MULDIV_enable  = input_monitor_vif_out.MULDIV_enable;
                                seq_item.MULDIV_ready   = input_monitor_vif_out.MULDIV_ready;

                                // Writting sequence_item to the subscriber
                                input_monitor_analysis_port.write(seq_item);

                                // Writting sequence_item to the sequencer (Reactive Agent)
                                sequencer_analysis_port.write(seq_item);
                end
        endtask : run_phase

        endclass
```

.

6.9.6 Sequencer

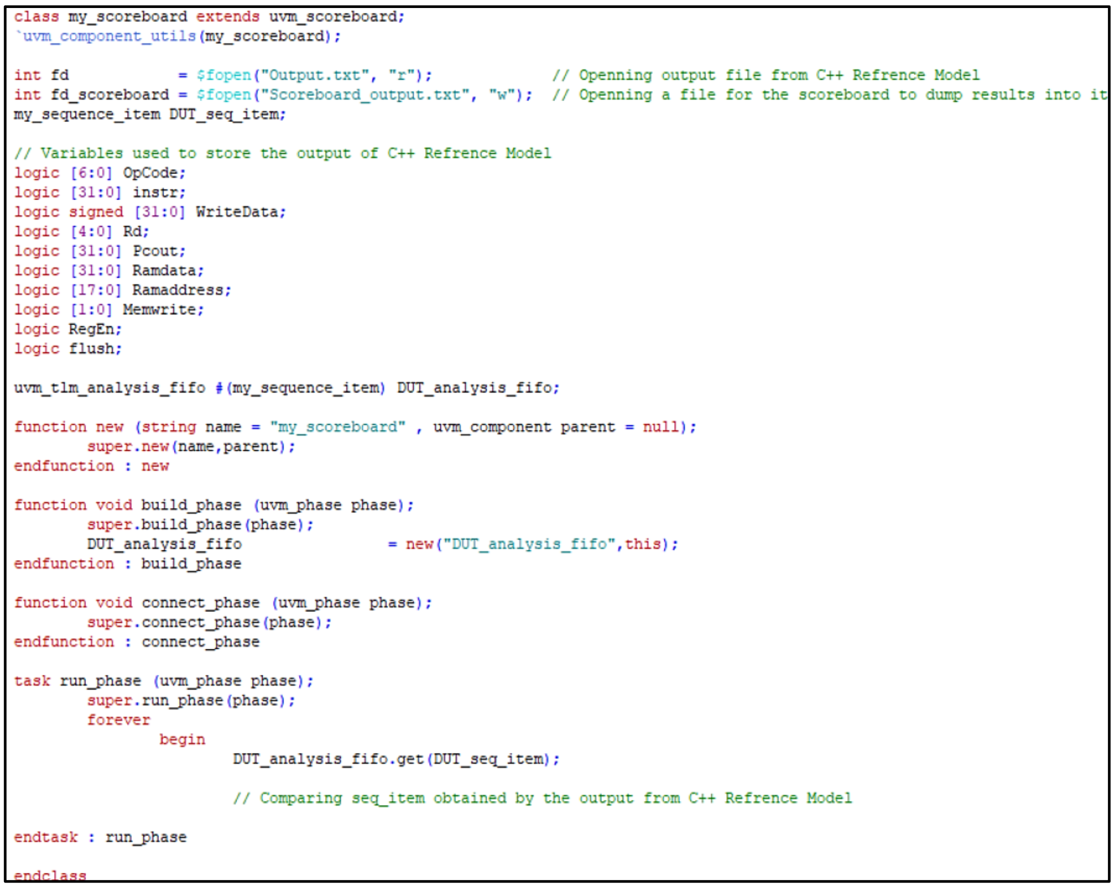
A sequencer controls the flow of transactions generated by one or multiple UVM sequences and send it to the driver to forward it to the DUT. It is recommended to extend uvm_sequencer base class as it contains all the functionality required to allow the sequence to communicate with a driver. The following is a code snippet from sequencer class.



```
class my_sequencer extends uvm_sequencer#(my_sequence_item);
`uvm_component_utils(my_sequencer);

uvm_tlm_analysis_fifo #(my_sequence_item) my_sequencer_fifo;

function new (string name = "my_sequence" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);

        my_sequencer_fifo = new("my_sequencer_fifo",this);

endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
endfunction : connect_phase

task run_phase (uvm_phase phase);
        super.run_phase(phase);
endtask : run_phase

endclass
```

6.9.7 Scoreboard

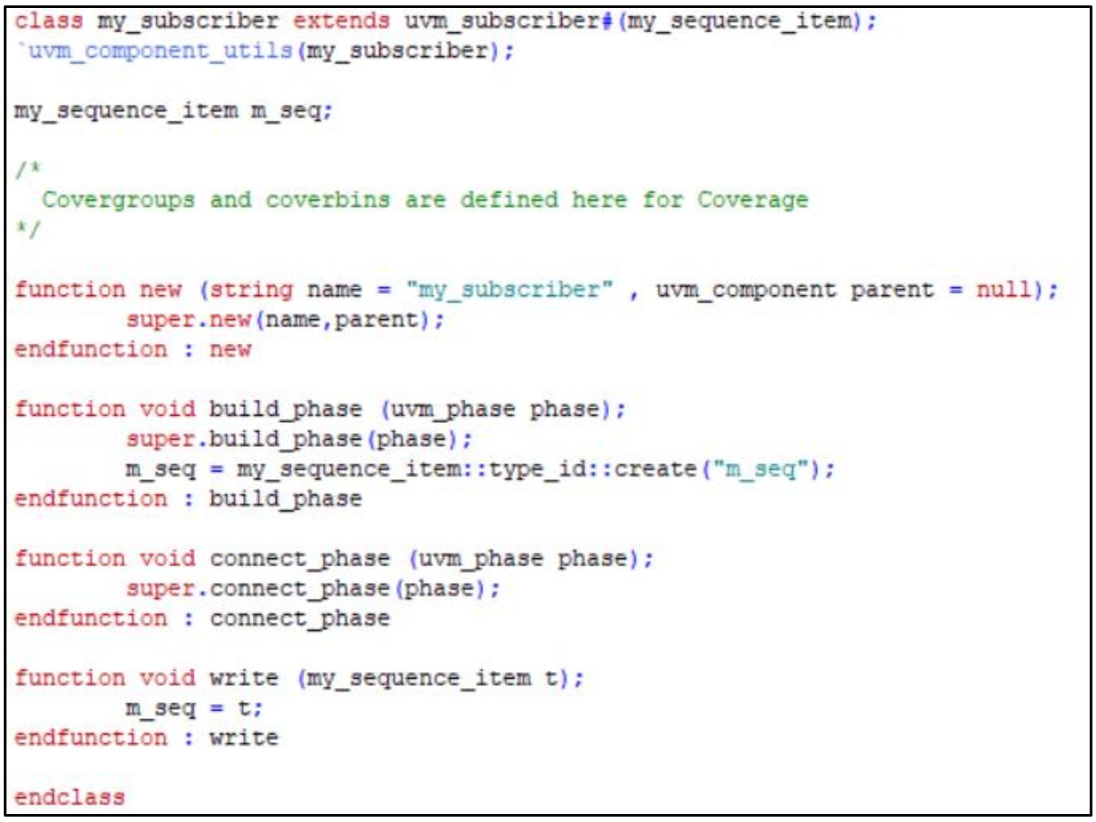
The Scoreboard is the most complex component in the UVM framework. It is inherited from the uvm_scoreboard base class. It has the important role of verifying that everything has worked by comparing the output signals from the DUT and the output from the C++ reference model. The following is a code snippet from scoreboard class.



```
class my_scoreboard extends uvm_scoreboard;
`uvm_component_utils(my_scoreboard);

int fd            = $fopen("Output.txt", "r");             // Openning output file from C++ Refrence Model
int fd_scoreboard = $fopen("Scoreboard_output.txt", "w");  // Openning a file for the scoreboard to dump results into it
my_sequence_item DUT_seq_item;

// Variables used to store the output of C++ Refrence Model
logic [6:0] OpCode;
logic [31:0] instr;
logic signed [31:0] WriteData;
logic [4:0] Rd;
logic [31:0] Pcout;
logic [31:0] Ramdata;
logic [17:0] Ramaddress;
logic [1:0] Memwrite;
logic RegEn;
logic flush;

uvm_tlm_analysis_fifo #(my_sequence_item) DUT_analysis_fifo;

function new (string name = "my_scoreboard" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        DUT_analysis_fifo               = new("DUT_analysis_fifo",this);
endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
endfunction : connect_phase

task run_phase (uvm_phase phase);
        super.run_phase(phase);
        forever
                begin
                        DUT_analysis_fifo.get(DUT_seq_item);

                        // Comparing seq_item obtained by the output from C++ Refrence Model

endtask : run_phase

endclass
```

6.9.8 Subscriber

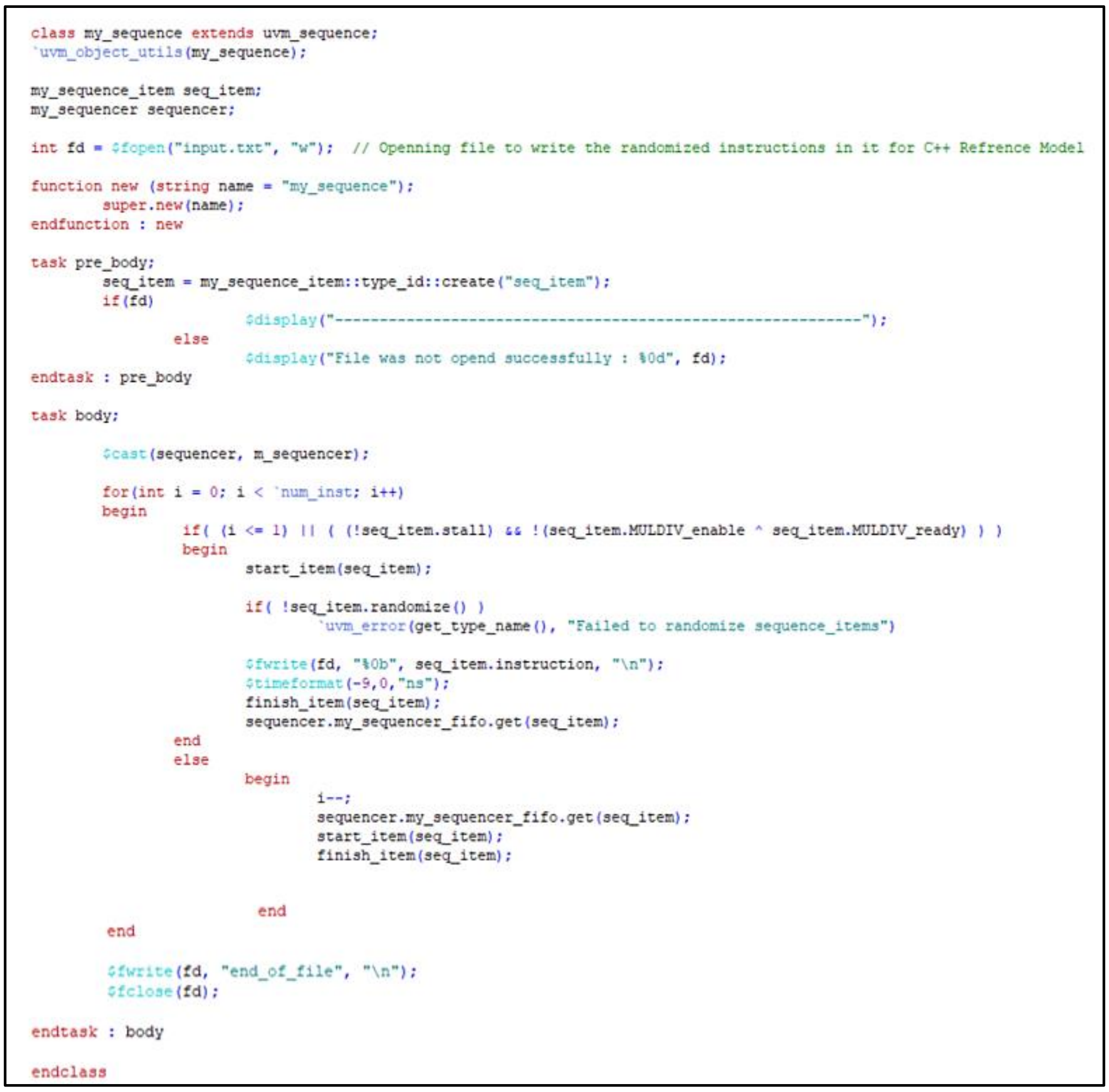
This class implements an analysis export that receives transactions from another analysis export. This component is "subscribed" to any transactions emitted by the linked analysis port when this connection is made. This class is particularly useful when designing a coverage collector. The following is a code snippet from subscriber class.


```
class my_subscriber extends uvm_subscriber#(my_sequence_item);
`uvm_component_utils(my_subscriber);

my_sequence_item m_seq;

/*
  Covergroups and coverbins are defined here for Coverage
*/

function new (string name = "my_subscriber" , uvm_component parent = null);
        super.new(name,parent);
endfunction : new

function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        m_seq = my_sequence_item::type_id::create("m_seq");
endfunction : build_phase

function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
endfunction : connect_phase

function void write (my_sequence_item t);
        m_seq = t;
endfunction : write

endclass
```

6.10 UVM objects

6.10.1 Sequence

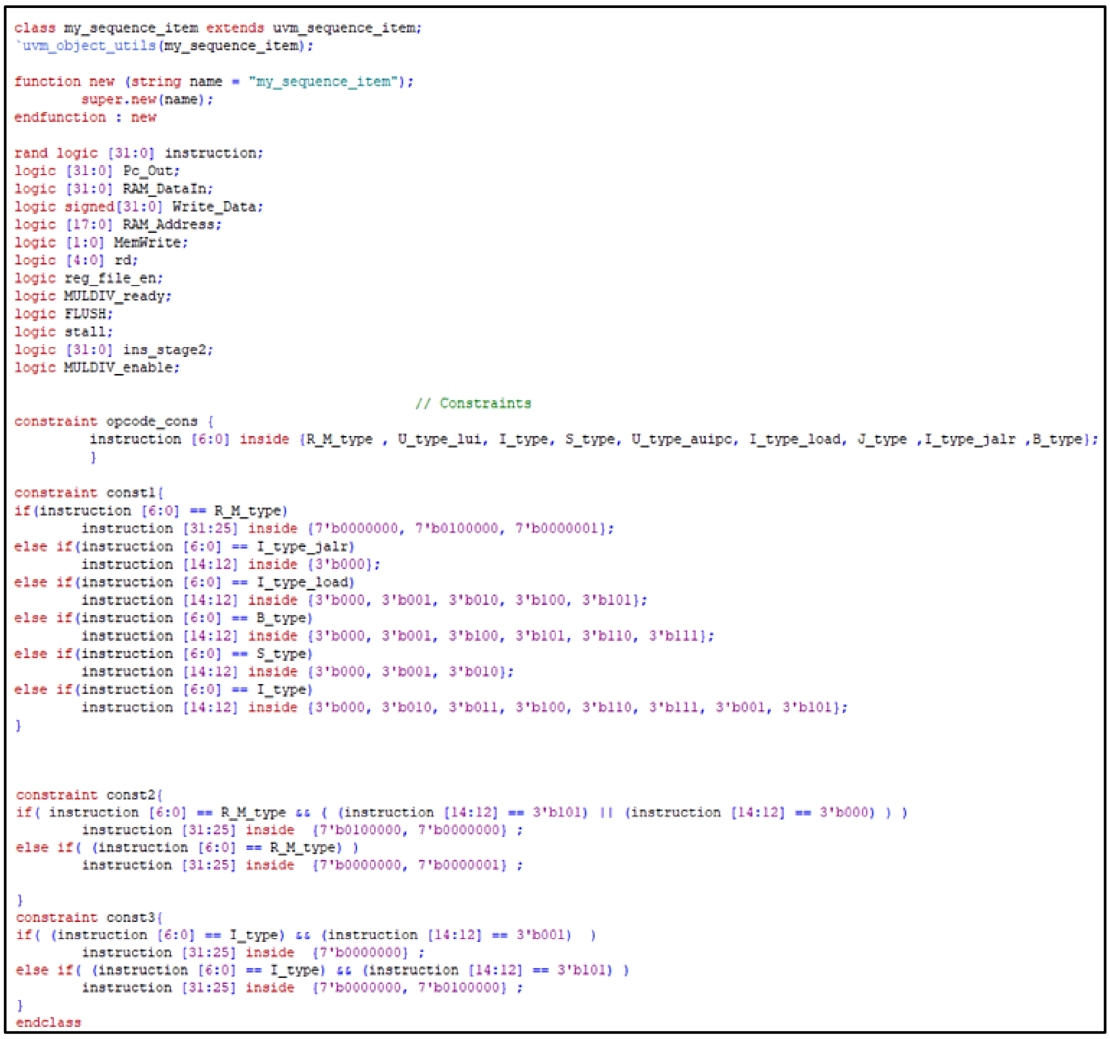
A UVM sequence is a set of System-Verilog code that is executed to make "things happen". In most cases, a sequence generates a transaction, randomizes it, and sends it to a sequencer, which then delivers it to a driver. The created transaction will typically produce some activity on the interface pins in the driver which is connected to the DUT. The following is a code snippet from sequence class.



```
class my_sequence extends uvm_sequence;
`uvm_object_utils(my_sequence);

my_sequence_item seq_item;
my_sequencer sequencer;

int fd = $fopen("input.txt", "w");  // Openning file to write the randomized instructions in it for C++ Refrence Model

function new (string name = "my_sequence");
        super.new(name);
endfunction : new

task pre_body;
        seq_item = my_sequence_item::type_id::create("seq_item");
        if(fd)
                        $display("------------------------------------------------------------");
                else
                        $display("File was not opend successfully : %0d", fd);
endtask : pre_body

task body;

        $cast(sequencer, m_sequencer);

        for(int i = 0; i < `num_inst; i++)
        begin
                if( (i <= 1) || ( (!seq_item.stall) && !(seq_item.MULDIV_enable ^ seq_item.MULDIV_ready) ) )
                begin
                        start_item(seq_item);

                        if( !seq_item.randomize() )
                                `uvm_error(get_type_name(), "Failed to randomize sequence_items")

                        $fwrite(fd, "%0b", seq_item.instruction, "\n");
                        $timeformat(-9,0,"ns");
                        finish_item(seq_item);
                        sequencer.my_sequencer_fifo.get(seq_item);
                end
                else
                        begin
                                i--;
                                sequencer.my_sequencer_fifo.get(seq_item);
                                start_item(seq_item);
                                finish_item(seq_item);


                        end
        end

        $fwrite(fd, "end_of_file", "\n");
        $fclose(fd);

endtask : body

endclass
```

6.10.2 Sequence item

The UVM sequence item base class in the UVM System Verilog library is used to define data items. The transactions to be gathered or driven in the DUT are known as data items. There are two different sequence items, the first one is related to the input transactions which is made up of the data fields that are needed to create the stimulus and are specified as rand in that instance, with constraint ranges defined. The second one is related to the output transactions and contains signals that are used to check the results of the DUT. The following is a code snippet from sequence item class.



```
class my_sequence_item extends uvm_sequence_item;
`uvm_object_utils(my_sequence_item);

function new (string name = "my_sequence_item");
        super.new(name);
endfunction : new

rand logic [31:0] instruction;
logic [31:0] Pc_Out;
logic [31:0] RAM_DataIn;
logic signed[31:0] Write_Data;
logic [17:0] RAM_Address;
logic [1:0] MemWrite;
logic [4:0] rd;
logic reg_file_en;
logic MULDIV_ready;
logic FLUSH;
logic stall;
logic [31:0] ins_stage2;
logic MULDIV_enable;

                                                // Constraints
constraint opcode_cons {
         instruction [6:0] inside {R_M_type , U_type_lui, I_type, S_type, U_type_auipc, I_type_load, J_type ,I_type_jalr ,B_type};
         }

constraint const1{
if(instruction [6:0] == R_M_type)
        instruction [31:25] inside {7'b0000000, 7'b0100000, 7'b0000001};
else if(instruction [6:0] == I_type_jalr)
        instruction [14:12] inside {3'b000};
else if(instruction [6:0] == I_type_load)
        instruction [14:12] inside {3'b000, 3'b001, 3'b010, 3'b100, 3'b101};
else if(instruction [6:0] == B_type)
        instruction [14:12] inside {3'b000, 3'b001, 3'b100, 3'b101, 3'b110, 3'b111};
else if(instruction [6:0] == S_type)
        instruction [14:12] inside {3'b000, 3'b001, 3'b010};
else if(instruction [6:0] == I_type)
        instruction [14:12] inside {3'b000, 3'b010, 3'b011, 3'b100, 3'b110, 3'b111, 3'b001, 3'b101};
}



constraint const2{
if( instruction [6:0] == R_M_type && ( (instruction [14:12] == 3'b101) || (instruction [14:12] == 3'b000) ) )
        instruction [31:25] inside  {7'b0100000, 7'b0000000} ;
else if( (instruction [6:0] == R_M_type) )
        instruction [31:25] inside  {7'b0000000, 7'b0000001} ;

}
constraint const3{
if( (instruction [6:0] == I_type) && (instruction [14:12] == 3'b001)  )
        instruction [31:25] inside  {7'b0000000} ;
else if( (instruction [6:0] == I_type) && (instruction [14:12] == 3'b101) )
        instruction [31:25] inside  {7'b0000000, 7'b0100000} ;
}
endclass
```

6.11 Reactive Agent

As explained in the section 3 of the pipelined core, a stall could be introduced due to dependencies between instructions. This stall should prevent the core from fetching a new instruction and thus we need the randomization of the sequence items in the sequence to be stalled as well to make sure that no new instructions are driven to the DUT by the driver during the stall period. A similar case to the stall is when a multiplication or division instructions enter the core where the randomization process should also be stopped as the whole core is frozen for multiple clock cycles as explained in section 2.2 of the M-extension (for the rest of this section we will take the stall signal as an example for simplicity).

As the randomization process of the sequence items derived to the DUT is controlled by a signal monitored from the DUT itself, a Reactive Agent should be used in this case instead of an Active Agent.

The reactive agent-based verification approach can be used to verify a design that works as shown in figure. Device-1 and Device-2 are communicating with each other, where Device-2 is generating a request (which is the stall signal in our case) whereas Device-1 is responding to the request (By sending a new random sequence item no stalls occurred or by freezing the randomization process if stalls occurred). Device-2 is the Design under Test (DUT) and Device-1 is replaced with the Reactive Agent.

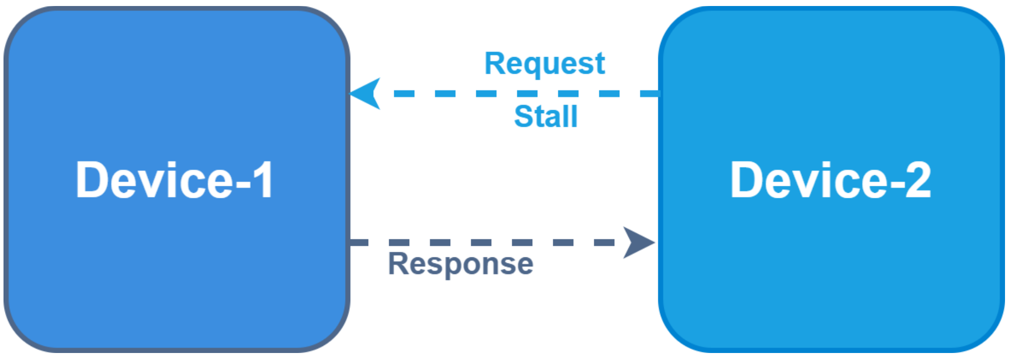


Figure 54: Reactive Agent operation

There are several approaches for implementing a Reactive agent. In our UVM environment, we have used the Monitor-sequencer approach. The monitor serves as the sampler in this method. The monitor delivers the request to the sequencer after sampling it. UVM offers a number of ways for the monitor to transmit the request to the sequencer. However, using the analysis port connection between the monitor and the sequencer is preferred. The user could obtain access to sequencer properties inside the sequence using the m_sequencer handle, which implies the user can monitor sampled requests inside the sequence using this handle. After sampling the stall from the DUT, the sequence could stop the randomization process according to the value of the sampled stall signal.

6.12 Core bugs caught by UVM

Due to large number of randomizations of the instructions derived to the DUT, we caught some extreme cases that might occur that may make the core to fail. The problems caught are:

1- First problem: Branch / Jump instructions with MULDIV instructions

If the MULDIV instruction is at the execute stage and the branch at memory stage, there are two events that will happen, MULDIV unit wants to disable all the five pipeline registers as MULDIV takes 34 clock cycles and the second event is FLUSH if branch is taken. We noticed that the scoreboard would skip an iteration due to MULDIV enable = 1, however the priority of the FLUSH is greater than the MULDIV enable. The three following instructions after the branch must be FLUSHED, so MULDIV enable mustn't be enabled, so scoreboard mustn't skip any iteration. So, we modified the DUT so as when a FLUSH exists, the MULDIV enable will be zero so, scoreboard won't skip anything, and the registers will not be disabled.

2- Second problem: SRA and SRL instructions wasn't implemented efficiently (when comparing with a RISC-V simulator)

If we have SRA x2, x1, x3 and the value inside x1 = 2 and the value inside x3 = 32, so the output of this instruction that will be written back in x2 is 2, why? As we are shifting by the value of the least five bits in x3 (the randomization helped us to catch this problem)

3- Third problem: BLT and SLT instructions

Suppose we have this instruction BLT x1, x2, pcrel_13 and the value inside x1 = 1895109858 and the value inside x2 = -2142972622, branch less than instruction will subtract the value of x1 from x2 so we will get 11110000101100000011111110110000. We will notice that the MSB = 1, which means that the number is negative, however when doing this calculation, we will see that the answer will be positive as we are subtracting positive number from negative number, so the branch will be taken, however it mustn't be taken here, so to solve this problem we added an overflow flag and XORed it with the sign flag to decide if we will branch or no.

4- Fourth problem: Branch / Jump instructions with stall

Suppose we have the following instructions:

beq x0, x0, L1

lw x2, 0(x0)

addi x3, x0, 12

addi x4, x0, 13

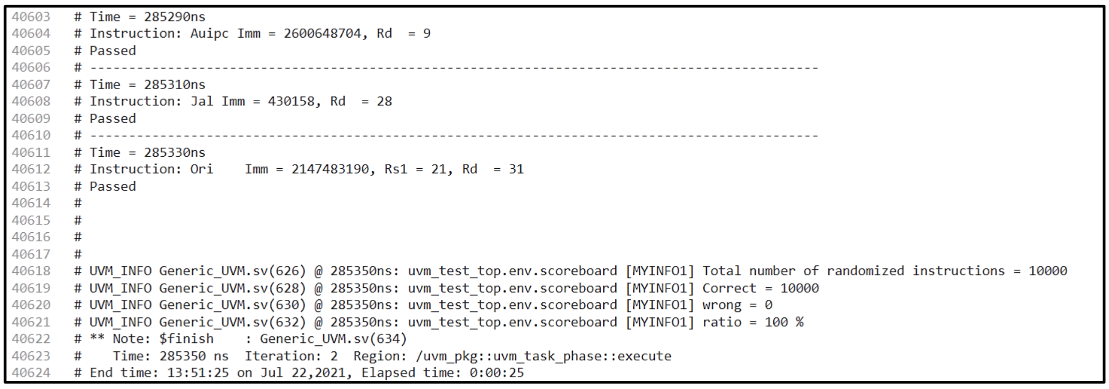
addi x5, x0, 15

L1: addi x6, x0, 16

When branch is at memory stage, load at execute stage and add at decode stage, there will be two events will happen, stall signal will be equal to “1” and FLUSH signal will be “1” if the branch is taken. As stall = 1, the sequence will be blocked and won't send a transaction in the next cycle due to the reactive agent. Stall signal disable the first two registers (PC and IF/ID), however the priority of the FLUSH is greater than the stall, so the first two registers won't be disabled and in the next clock cycle we will see that branch will be at write back stage, load will be at memory stage, (addi x3, x0, 12) will be at execute, (addi x4, x0, 13) will be at decode and also (addi x4, x0, 13) will be at fetch as the sequence is blocked at this cycle as stall signal was equal 1, so (addi x5, x0, 15) will be blocked and won't be sent to the fetch stage, so (addi x4, x0, 13) is now exists twice. Due to FLUSH = 1, we must see three zeros after this signal, but what happened here is that we saw zero (load), then gap (addi x3, x0, 12), then zero (addi x4, x0, 13), then (addi x4, x0, 13), but how we saw (addi x4, x0, 13) and this instruction is one of the three instructions after the branch? We noticed this due to the blocked sequence. We solved this problem by modifying hazard detection unit by making stall = 0 when there is a FLUSH.

6.13 UVM Results

The output of the UVM environment in the verification phase was very acceptable. After modifying the core according to the bugs in section 10.12, we ran 10,000 randomized instructions with no errors at all. As shown in figure 55, a snippet from the report shows these results obtained when comparing the output of the C++ reference model and the output of the DUT.



```
# Time = 285290ns
# Instruction: Auipc Imm = 2600648704, Rd  = 9
# Passed
# ----------------------------------------------------------------------------------------------
# Time = 285310ns
# Instruction: Jal Imm = 430158, Rd  = 28
# Passed
# ----------------------------------------------------------------------------------------------
# Time = 285330ns
# Instruction: Ori    Imm = 2147483190, Rs1 = 21, Rd  = 31
# Passed
#
#
#
#
# UVM_INFO Generic_UVM.sv(626) @ 285350ns: uvm_test_top.env.scoreboard [MYINFO1] Total number of randomized instructions = 10000
# UVM_INFO Generic_UVM.sv(628) @ 285350ns: uvm_test_top.env.scoreboard [MYINFO1] Correct = 10000
# UVM_INFO Generic_UVM.sv(630) @ 285350ns: uvm_test_top.env.scoreboard [MYINFO1] wrong = 0
# UVM_INFO Generic_UVM.sv(632) @ 285350ns: uvm_test_top.env.scoreboard [MYINFO1] ratio = 100 %
# ** Note: $finish    : Generic_UVM.sv(634)
#    Time: 285350 ns  Iteration: 2  Region: /uvm_pkg::uvm_task_phase::execute
# End time: 13:51:25 on Jul 22,2021, Elapsed time: 0:00:25
```

Figure 55: UVM results

Conclusion and Future work

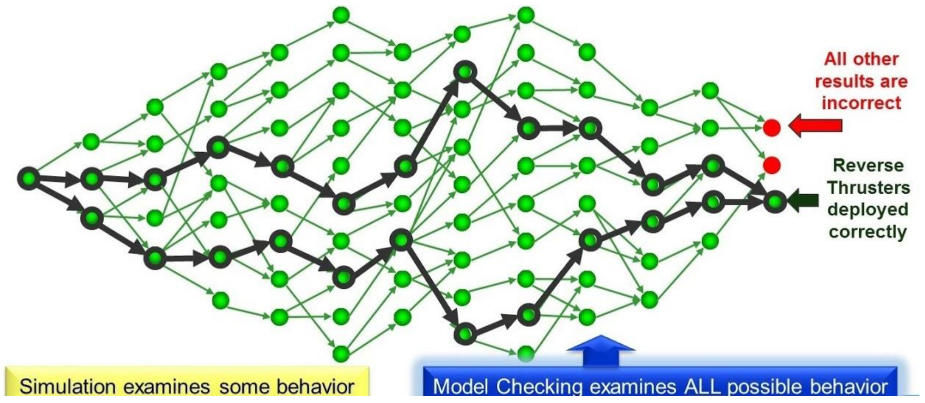
• Conclusion

IoT security is a very important field that we should be aware of, hardware security has proven to be more robust than software security, being faster and immune to software bugs and having a dedicated hardware to execute, thus provide more security to the IoT data. The huge increase in the complexity of SoCs nowadays has led EDA companies to develop powerful tools to deal with the issues that may arise while designing and discover these issues in the early phases to save time and money. In this work, a SoC was designed for IoT hardware security applications, and a generic UVM environment was designed to test our RISC-V based Core and will be able to test other RISC-V cores as well.

Future work

1-Formal verification

Formal verification tools are a collection of technologies that employ static analysis to ensure the correctness of hardware or software behavior in terms of a formal specification or a written property. Formal verification uses state traversal to find the output without actual input stimulus as shown.


All other results are incorrect
Reverse Thrusters deployed correctly
Simulation examines some behavior
Model Checking examines ALL possible behavior

Formal verification has 2 types:

1- **Equivalency checking**: This technology uses mathematical modeling techniques to prove that two representations of design exhibit the same behavior.

2- **Assertion-based Verification**: Assertions are written in languages as System Verilog or PSL to ensure the required properties are satisfied.

2-UVM Coverage

The Coverage collector or the subscriber is responsible for performing the coverage either on our corner cases or our random stimulus. Code and functionality coverage can be furthermore considered to make sure that no corner cases or verification holes are left unchecked. Questa Auto check will also be run to increase the coverage.

3-SoC UVM environment

The UVM environment designed was for testing the RISC-V Core, the UVM environment can be furthermore improved to be able to test the whole SoC.

References

[1] The Morgan Kaufmann Series in Computer Architecture and Design RISC-V Edition.

[2] Doulos.com. (2019). UVM Verification Primer.

[3] FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version 1st Edition.

[4] CAESAR hardware Application Programming Interface (API) for authenticated ciphers. Available at https://eprint.iacr.org/2016/626.pdf

[5] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213

[6] The RISC-V Reader: An Open Architecture Atlas First Edition, 1.0.0

[7] System Verilog Reference Guide. Available at: http://svref.renerta.com/.

[8] MASTERING REACTIVE SLAVES IN UVM. available at: https://www.verilab.com/files/mastering_reactive_slaves.pdf

[9] Verificationacademy.com. (2019). UVM Factory. [online] Available at: https://verificationacademy.com/verification methodologyreference/uvm/docs_1.1a/html/files/base/uvm_factory-svh.html.

[10] Handbook of Digital CMOS Technology, Circuits, and Systems by Dr. Karim abbas

[11] Introduction to UVM at John Aynsley YouTube channel: https://youtu.be/imH4CFmVGWE?list=PLLn6Zp_o9-jSI_HXqN9bkvE70YY4dDfub.

[12] https://competitions.cr.yp.to/caesar.html

[13] https://www.eetimes.com/fifo-handshake-synchronizers-a-challenge-for-cdc-analysis/

[14] https://www.eetimes.com/understanding-clock-domain-crossing-issues/

[15] https://www.embedded.com/asynchronous-reset-synchronization-and-distribution-challenges-and-solutions/

[16] https://vlsiuniverse.blogspot.com/2017/04/recovery-and-removal-checks.html

[17] Low Area and Low Power Implementation for Competition for Authenticated Encryption, Security, Applicability, and Robustness Authenticated Ciphers.

[18] Using UVM Virtual Sequencers & Virtual Sequences CummingsDVCon2016_Vsequencers.pdf

[19] https://www.chipverify.com/uvm/

[20] https://verificationguide.com/uvm

[21] SYSTEMVERILOG FOR VERIFICATION A Guide to Learning the Testbench Language Features