Technische Universität Dresden

Faculty of Electrical and Computer Engineering
Chair of Highly-Parallel VLSI Systems and Neuro-microelectronics

VLSI PROCESSOR DESIGN

WS2020 Project: Designing a RISC-V Architecture based Processor for General Purpose Computing
(svn: lpd19 /group04)

Submitted by: Dilip Kumar Erappa(4825384)(erdi19)
Pruthvi Raj Venkatesha(4820130)(vepr19)

Supervisors: Dr. -Ing. Stefan Scholze and Dr. -Ing. Andreas Dixius
Chair Holder: Prof. Dr.-Ing. habil. Christian Georg Mayr
Date of submission: March 15, 2021

Declaration of Authorship

We, Dilip Kumar Erappa(Matriculation Nr: 4825384) and Pruthvi Raj Venkatesha(Matriculation Nr: 4820130), hereby certify that, this Project on the topic

WS2020 Project: Designing a RISC-V Architecture based Processor for General Purpose Computing (svn: lpd19/group04)

has been composed independently and is based on our own work at the Faculty of Electrical and Computer Engineering, unless stated otherwise. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Dresden, March 15, 2021 Dilip Kumar Erappa and Pruthvi Raj Venkatesha

Acknowledgement

We wish to express our deep gratitude to Prof. Dr.-Ing. habil. Christian Mayr for allowing us to do the VLSI Processor Design Course work at Chair of Highly-Parallel VLSI Systems and Neuro- Microelectronics. We are indebted to Dipl.-Ing. Stefan Scholze and Dipl.-Ing. Andreas Dixius for guiding and supporting us throughout our project. Without their encouragement and guidance this Project would not have been successful.

March 15, 2021
Dresden Dilip Kumar Erappa and Pruthvi Raj Venkatesha

Contents

Chapter 1: Introduction 7
1.1. Computer architectures 7
1.1.1. Harvard architecture 7
1.1.2. Von-Neumann architecture 7
1.2 Instruction Set Architectures 8
1.3 RISC-V[4] 8
1.3.1 RISC-V instruction format[4] 9
1.4 Task Description 10
Chapter 2: PROCESSOR DESIGN APPROACHES 11
2.1 SINGLE CYCLE PROCESSOR ARCHITECTURE[5] 11
2.2 MULTI CYCLE PROCESSOR ARCHITECTURE[5] 11
2.3 PIPELINED PROCESSOR ARCHITECTURE[5] 11
2.4 SELECTION OF MICROARCHITECTURE 11
2.5 BASIC UNITS FOR A PROCCESSOR DESIGN[9] 12
2.6 DESIGN OF A SINGLE CYCLE PROCESSOR 13
2.7 BUILDING A DATAPATH USING INSTRUCTION FORMATS 14
2.8 COMPLICATIONS: 18
2.9 SINGLE CYCLE PROCESSOR 20
2.10 TOP MODULE 21
Chapter 3: Simulation and Verification 22
3.1 Synthesizability Check 22
3.2 Simulation 23
3.3 Design Goal and Work split 24
Chapter 4: Synthesis 25
4.1 POST SYNTHESIS ANALYSIS 26
4.2 PHYSICAL DESIGN OF THE CHIP[9] 28
Chapter 5: PIPELINING 31
5.1 PIPELINED PROCESSOR FOR HIGH THROUGHPUT AND SPEED 32
5.2 SOLVING DATA HAZARDS AND CONTROL HAZARDS[5] 35
5.3 Branch Prediction 36
5.4 HAZARD DETECTION UNIT 38
5.5 Pipelined Processor 40
Chapter 6: SIMULATION AND VERIFICATION RESULTS 42
6.1 Single Cycle Simulation Results: 42
6.2 Pipelined Processor Simulation Results 47
6.3 Verification using C code 51
6.3.1 Pipelined Processor Results 51
6.3.2 Single Cycle Processor Results 52
Chapter 7 SYNTHESIS AND PNR REPORTS 54

Chapter 8 REFERENCES 60

LIST OF TABLES

Table 1 RV32I Instructions 13
Table 2 R-TYPE Instruction details 13
Table 3 Modified Booth's Algorithm 26
Table 4 Single Cycle Processor Details 27
Table 5 User defined inputs for PNR Tool 29
Table 6 Pipelined Processor Summary 41

LIST OF FIGURES

Figure 1 Harvard Architecture 7
Figure 2 Von-Neuman Architecture 7
Figure 3 RISC-V Instruction format[4] 9
Figure 4 Dobby SOC Architecture 10
Figure 5 Interactions in a Processor 12
Figure 6 Basic blocks of a Processor 12
Figure 7 R-type implementation 14
Figure 8 I-type implementation 15
Figure 9 J-type implementation 15
Figure 10 B-type implementation 16
Figure 11 S-type implementation 16
Figure 12 LUI and AUAIPC implementation 17
Figure 13 CSR implementation 17
Figure 14 Branch Unit implementation 18
Figure 15 Interrupt control 18
Figure 16 Hold register Inclusion 19
Figure 17 State machine for Memory initialisation 19
Figure 18 Datapath of a Single cycle processor 20
Figure 19 Dobby Top view 21
Figure 20 External Fetch State Machine 21
Figure 21 Warnings 22
Figure 22 Testbench Top module overview 23
Figure 23 Code snippet 23
Figure 24 Coverage report 24
Figure 25 Constraint file snippet 25
Figure 26 Booth's Algorithm 26
Figure 27 Frequency vs Area Graph with Division operation 27
Figure 28 Place and Route Flow 29
Figure 29 Single Cycle Processor 30
Figure 30 Pipelined Architecture 31
Figure 31 Instruction Flow in Time domain 31
Figure 32 First Stage Resources 32
Figure 33 Second Stage Resources 33
Figure 34 Third Stage Resources 34
Figure 35 Fourth Stage Resources 34
Figure 36 Branch Predictor logic 37
Figure 37 Pipelined Processor 40
Figure 38 PRAM Initialization 42
Figure 39 PRAM Initialisation completion 42
Figure 40 Single cycle Arithmetic Instruction 43
Figure 41 Single Cycle logical Instruction 43
Figure 42 Load from External Memory 43
Figure 43 Load from Internal memory 44

Figure 44 Division Operation 44
Figure 45 Divide by zero case 44
Figure 46 Jump Instruction 45
Figure 47 Store - external memory 45
Figure 48 Store - internal memory 45
Figure 49 Wait for Interrupt 46
Figure 50 Interrupt handler 46
Figure 51 Random bus ready 46
Figure 52 Initialization of Internal PRAM 47
Figure 53 Initialization stop during external Interrupt 47
Figure 54 Single cycle instruction example 47
Figure 55 Load from External 48
Figure 56 Load from Internal PRAM 48
Figure 57 Bitwise Instruction 48
Figure 58 Multiplication operation 49
Figure 59 Division Operation 49
Figure 60 Internal PRAM store 49
Figure 61 Branch Instructions 50
Figure 62 External SRAM store 50
Figure 63 Interrupt Handling 50
Figure 64 Division by zero 51
Figure 65 Prime or Not 51
Figure 66 Factorial 51
Figure 67 Sort and Search 52
Figure 68 Prime number algorithm 52
Figure 69 Factorial algorithm 53
Figure 70 Sort and Search algorithm 53
Figure 71 REG -> Out path 54
Figure 72 In ->REG path 54
Figure 73 REG -> REG path 54
Figure 74 Area report 55
Figure 75 Power report 55
Figure 76 Final Post route Summary 56
Figure 77 REG->REG path 56
Figure 78 In->REG path 57
Figure 79 REG->Out path 57
Figure 80 Power Report 57
Figure 81 Area Report 58
Figure 82 Final PNR Report 58
Figure 83 Pipelined Processor 58

Chapter 1: Introduction

This report describes the project on implementation of Design of 32-bit RISC-V (Simple RISC architecture) processor. Before explaining the fundamentals of RISC-V, this section focuses on basics of CPU.

The CPU also referred as a Central Processing Unit is the hardware design inside computer system which performs some operations based on the instructions given by a computer program[1]. The CPU is the heart of the microprocessor. The computation operations as well as generating control signals to control the rest of the machine are generated in this unit. The CPU is built of different components like Register File, Fetch Unit, ALU, Control unit, Program counter(PC), Instruction Memory, Data memory etc.

1.1. Computer architectures

There are mainly two different types of popular Computer architectures, Von Neumann and Harvard architecture.

1.1.1. Harvard architecture

The Harvard architecture stores machine instructions and data in separate memory units using different buses. Computers designed with the Harvard architecture can run a program and access data independently, and therefore simultaneously[2]. Figure 1 shows simple representation of Harvard Architecture.

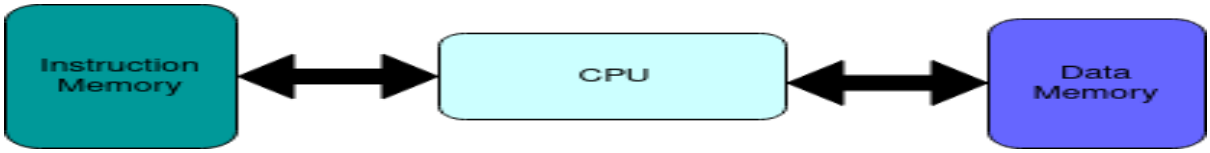

Figure 1 Harvard Architecture

1.1.2. Von-Neumann architecture

This architecture was proposed by John Von-Neumann. Here the same memory and bus are used to store both Data and Instructions. CPU is unable to access program memory and data memory simultaneously[2]. Figure 2 shows simple representation of Von-Neumann Architecture.

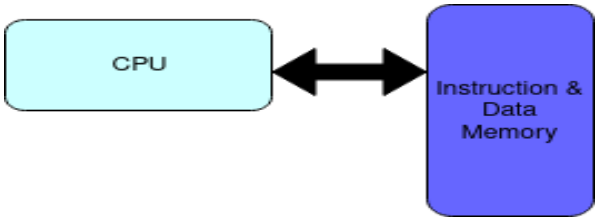

Figure 2 Von-Neuman Architecture

1.2 Instruction Set Architectures

An instruction set, or Instruction Set Architecture (ISA), is the part of the processor architecture related to programming. All processors are supported by instruction set or instructions (Assembly instructions) which are dependent on organization of different components in CPU.

Depending upon the way of supporting different instructions, the ISA is mainly of two types – CISC and RISC.

Complex Instruction Set Computer(CISC), as name suggests it has a large amount of different multi-clock complex instructions[3]. This type of processor emphasis on hardware more than software and CISC processor uses transistors for storing complex instructions. CISC processors are relatively slow in comparison to Reduced Instruction Set Computer(RISC) but at the same time it uses a smaller number of instructions.

Against CISC architecture, RISC architecture-based processors are faster. It emphasizes on software more than hardware[3]. RISC uses simpler and faster instruction that is typically of size one so theoretically it uses fewer transistors which make RISC processor easier and less expensive to design. All operations performed on data apply to data in registers and it changes the entire register so basically all the operations are done on registers. The only operation that affect memory are load and store instructions that move data to and from memory.

1.3 RISC-V[4]

RISC-V pronounced as “RISC-five”, is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.

Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use. Several companies are offering or have announced RISC-V hardware, open-source operating systems are available, and the instruction set is supported in several popular software toolchains.
The ISA somehow decides the application. It is well known that the world’s most famous ISA’s are X86 and ARM. Their application fields are totally different. Most of laptops, desktops, and servers are based on X86 or AMD64 ISA. These IPs belong to Intel and AMD. Most of mobile phones, tablets are based on ARM ISA, the IPs are divided into A series, R series, and M series. While RISC-V is very suitable for use in some specific application fields such as storage, edge computing, and AI applications. The different applications field makes it possible for RISC-V to compete with ARM and X86. Comparing to ARM and X86, RISC-V has below advantages[4]:
1. RISC-V is open-source, hence free of cost – no need to pay for the IP.
2. RISC-V is far smaller than other commercial ISAs.
3. RISC-V has a small standard base ISA, with multiple standard extensions.
4. Base and first standard extensions are already frozen. There is no need to worry about major updates.
5. Specific functions can be added based on extensions. Many more extensions are under development.
As mentioned above, the RISC-V instruction set has modular characteristics. The instruction set is organized in a modular manner. Each module is represented by an alphabet. The

instruction set includes the standard part and the extension part. The standard part(which is the base ISA) must be implemented.
For example, if we want to implement a 32-bit architecture RISC-V processor, the RV32I instruction set must be implemented on the hardware (machine mode must also be implemented in privileged mode). we can also enhance the processor's functionality by adding extensions to ISA. There are already many standard extensions[4], such as the approved MAFDGQLCBJTPVN.
So, we have four base integer ISAs – RV32I, RV32E, RV64I, RV128I. All of them have <50 hardware instructions needed for base[4]. As mentioned we have M : Integer multiply/divide, A: Atomic memory operations, F: single precision floating point, D:Double precision floating point, G=IMAFD: General purpose ISA, Q: Quad-precision floating point, L: Decimal floating point, C: compressed instruction set, B: Bit manipulation, J: extension for dynamically translated languages, T: extension for Transactional memory, P: Packed SIMD instruction, V: Vector operations and N: User level interrupts as Standard extensions to the Base ISA. In our project, we have implemented RV32IM along with required machine mode instructions in privileged mode.

1.3.1 RISC-V instruction format[4]

In RISC-V, there are six different types of instructions: R-type. I-type, J-type, S-type, B-type, and U-type. As shown in Figure 3, each instruction type ends with 7-bit opcode. In addition to this opcode region, R-type instructions has three different register field *rs1*, *rs2* and *rd* corresponding to two source registers and a destination register. Each immediate subfield is labelled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

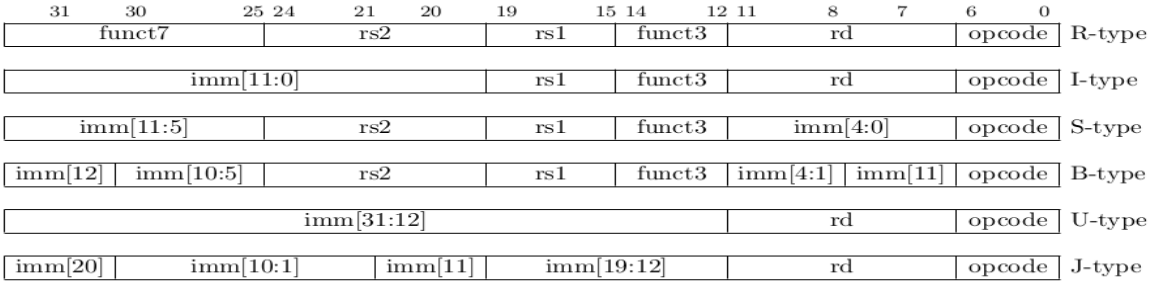

Figure 3 RISC-V Instruction format[4]

As shown in Figure 3, I-type instructions takes two arguments, *rs* and *rd* and 12-bit immediate value, this immediate value is not stored in memory, but it is a part of the instruction. The benefit of such immediate is that we do not need to work with the memory so accessing constant (immediate) is much faster. RISC-V provides two types of controls transfer instructions for unconditional and conditional branches (J-type and B-type respectively). The unconditional jumps JAL use J-type format. All conditional branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 and is added to the current pc to give the target address.
Load and store instructions exchange data between CPU registers and memory. Loads are encoded in I-type format and Store instruction is encoded in S type.
U-type instruction is just like I-type, but it has only destination register encoded in it and the upper immediate value is used for calculations.

1.4 Task Description

The Dobby microcontroller is a RISC-V ISA based processor integrated into a system-on-chip (SoC) and aimed at general processing tasks.
The following features were implemented and verified:

- RISC-V instruction set architecture RV32IM [Base instruction format(I) and Extension M which includes multiplication and division].
- 16 kB built-in program memory (PRAM).
- 2 general purpose interrupts, high-speed bus interface for peripheral device and/or external memory attachment.
- PRAM initialization after reset.
- A memory-controller giving the processor core and init-controller access to on-chip memory and the external bus interface based on the memory layout.

The figure 4 shows Top view of Dobby SOC architecture:

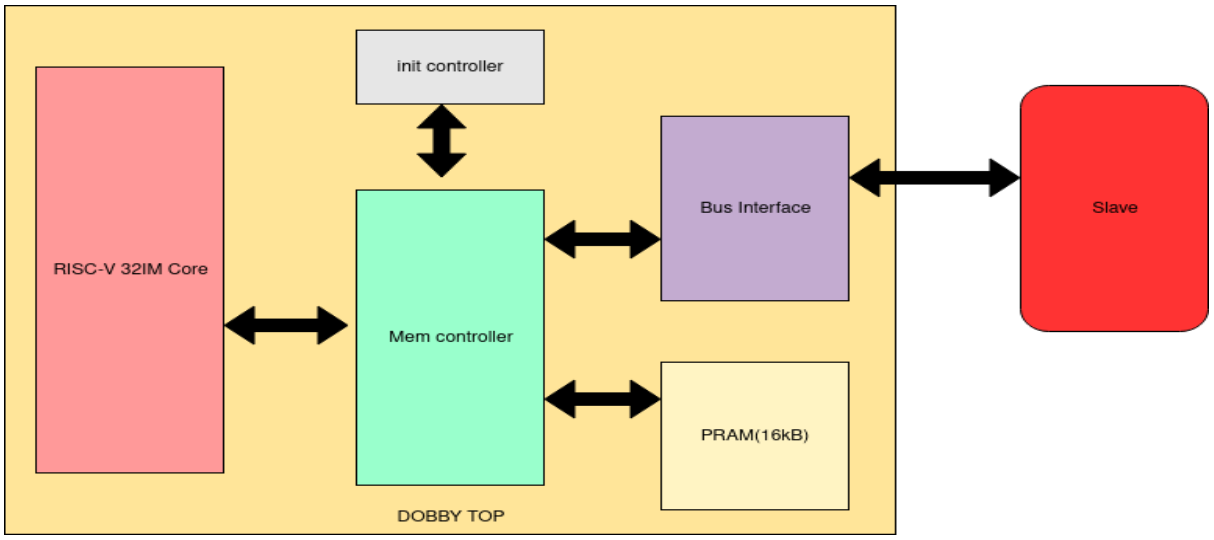

Figure 4 Dobby SOC Architecture

The chip has clock(I_CLK), reset(I_A_RESET_L), bus interfaces(O_BUS_EN, O_BUS_WE, O_BUS_SIZE, O_BUS_ADDR, I_BUS_RDY, B_BUS_DATA) , interrupt(I_INTR_H) and interrupt acknowledgement(O_INTR_ACK) as external pins.

Chapter 2: PROCESSOR DESIGN APPROACHES

In general, a processor can be designed in three ways based on the Microarchitecture used. All these differ in their state element connections and the amount of non-architectural states used in the processor. They are classified as:

1. Single cycle Processor Architecture
2. Multi cycle Processor Architecture
3. Pipelined Processor Architecture

2.1 SINGLE CYCLE PROCESSOR ARCHITECTURE[5]

In this processor, all instructions are executed in single cycle with the help of a simple control unit controlling the entire Datapath. The maximum frequency is limited by the slowest instruction which is executed. For memory or external operations where we need to wait for more cycles, the core is halted until the execution of the instruction completes and then resumes. Simplest design but helps in building or extending it to faster and high throughput processors.

2.2 MULTI CYCLE PROCESSOR ARCHITECTURE[5]

In this processor, all instructions are executed in a sequence of shorter cycles. In this way the simple instructions take few cycles while the complex instructions take more cycles. Hardware cost is very less in this architecture because of the reuse of hardware in every cycle. Example: Adder is reused for different purposes in every cycle . It also needs non architectural registers to store intermediate values. It executes one instruction at a time but completes in a series of cycles. This is one way to increase the frequency of the processor by a large amount and can be made with harder area constraints since resource reuse.

2.3 PIPELINED PROCESSOR ARCHITECTURE[5]

In this processor, a single cycle architecture is used and then the Datapath is split with help of the pipelining registers in between. Every split is a new stage now and there are processors with many stages. Best architecture is possible if the path is split evenly and the maximum frequency is limited by the worst-case delay of any stage. At any time, Multiple instructions are being executed and it also needs logic to take care of the dependencies between the execution of instructions and non-architectural registers split the Datapath. This architecture is very good because of the high throughput it provides.

2.4 SELECTION OF MICROARCHITECTURE

Every processor can be optimised for either Speed, Power or Area. All these factors are interdependent, an optimisation of one would degrade the other two. The selection of optimisation depends on the customer requirements. For our Dobby (RISC-V based Architecture) processor, we chose Speed optimisation and started with building a faster single cycle processor. Our aim was to get as much speed as possible from the single cycle. After building this, to get further speed up and better throughput, we implemented a four stage Pipelined processor which is also the state of art architecture used in today's processors.

Finite state machine-based design was explained in the tutorials of the course and we wanted to try a different approach, so we didn't use the state-based approach.

2.5 BASIC UNITS FOR A PROCCESSOR DESIGN[9]

Before starting the processor design, let us go through the basic units which will be used in any architecture. Every microarchitecture as shown below is divided into two interactive parts one is the Datapath unit which consist of all registers, ALU, MUX and many other blocks. The control path uses the instructions from the data path unit and tells the data path what action to take. Example: MUX selects signals, memory write signals, register enable etc are all controlled here.

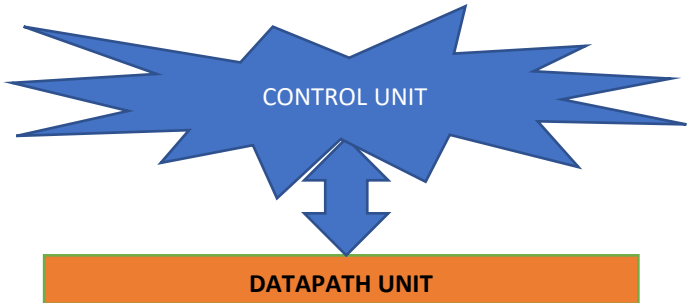

Figure 5 Interactions in a Processor

Any processor contains four basic state elements which are Program counter(PC), Instruction memory(IM), Register file(RF) and Data memory(DM). For Harvard based architectures, IM and DM are different whereas for Von-Neumann based architectures both are same.

Figure 6 shows the basic components of any processor[5]. The program counter is used to store the current pc address which is used for instruction memory access and many other operations. The signal pc next is calculated based on the instruction execution. Instruction memory holds all the instructions which are required to be executed on the processor. Register file is group of temporary storage registers which are very important for any data operations carried out in the processor. The size depends on the specifications and instruction format which decides the maximum number of registers addressable. A two-port combinational read and a single port sequential write is used generally. A1, A2 are read addresses and A3 is the write address. WE is the write enable signal with WD as write data. The Data memory is used to store all the data operation results which will be used by other components.

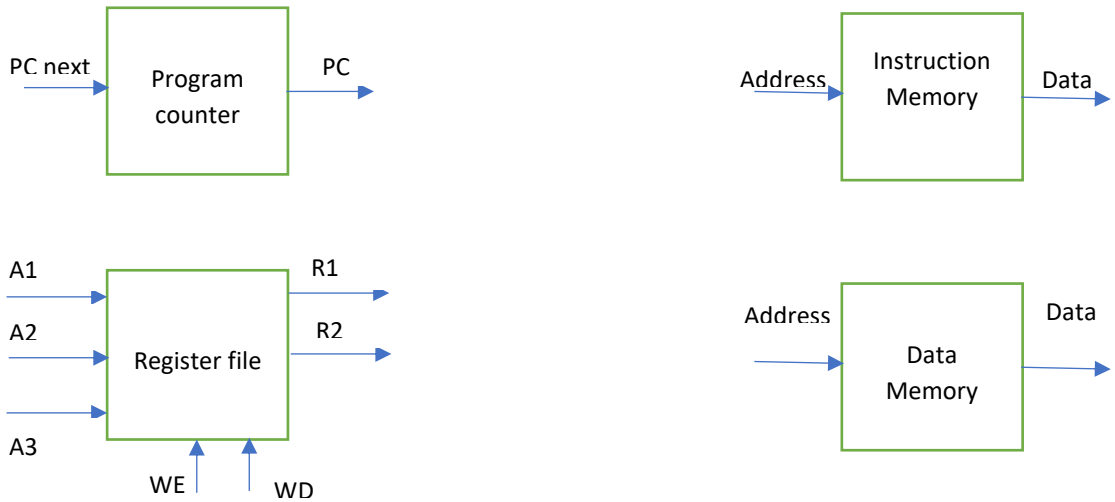

Figure 6 Basic blocks of a Processor

2.6 DESIGN OF A SINGLE CYCLE PROCESSOR

The task is to develop a core with RV32IM instruction encodings along with the privileged mode support. Let us start with RV32I instructions. The register file used here contains 32 registers of 32 bits each with r0 register always zero.
RV32I Instructions: This is the base integer ISA and supports 47 instructions which uses all the RISC Instruction formats(R,I,S,U,B,J).

Instructions Group	Instructions
Computational Instructions	ADD,SLT,SLTU,AND,OR,XOR,SLL,SRL,SUB,SRA, ADDI,SLTI,SLTIU,ANDI,ORI,XORI,SLLI,SRI,SRAI,LUI,AUIPC
Control transfer Instructions	JAL, JALR, BEQ/BNE,BLT[U],BGT[U],BGE[U],BLE[U],
Load store Instructions	LW,LH,LB,SW,SH,SB
Control status registers Instructions	CSRRW[I], CSRRS[I],CSRRC[I],
Environment Instructions	ECALL,EBREAK
Memory Operation Instruction	FENCE,FENCEI

Table 1 RV32I Instructions

All arithmetic and logical instructions can be implemented by ALU block. The processor uses a separate Instruction decoder and a control unit which decides what control signals should be enabled for a particular Instructions.

DECODING THE INSTRUCTION:
Generally, there are RISUBJ instruction formats, and each have their own opcode. Let us see an example.
REGISTER TYPE(R-TYPE)

Func7	RS2	RS1	Func3	RD	Opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Opcode for R type: 0110011
List of operations:

Operation	Func3	Func7
ADD	000	0000000
SUB	000	0100000
SLL	001	0000000
SLT	010	0000000
SLTU	011	0000000
XOR	100	0000000
SRL	101	0000000
SRA	101	0100000
OR	110	0000000
AND	111	0000000

Table 2 R-TYPE Instruction details

This table will be used to write control unit in HDL language for easy determination of control signals. Similar way for other formats, the control unit blueprint is obtained.

2.7 BUILDING A DATAPATH USING INSTRUCTION FORMATS

In this step, one instruction from each format is taken and the data path is built accordingly. In this way, the entire processor is built in a systematic way via addition of extensions in each step.

R-TYPE Implementation:

Consider an example, ***add rd,rs1,rs2*** instruction. This instruction needs to be fetched from the memory and given to decoder to get opcode and addresses of all registers. The control unit decides the ALU control signals, mux signals and the register enable signals.

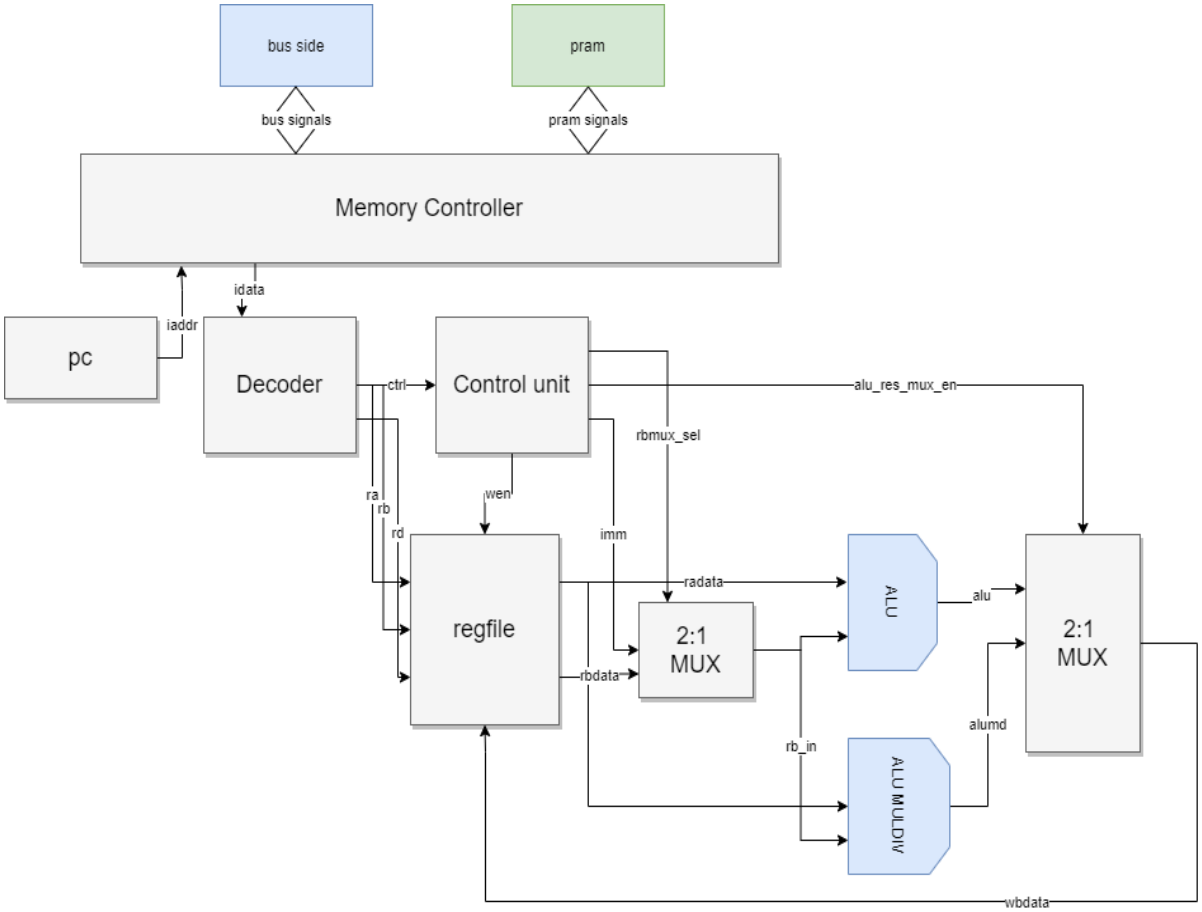

Figure 7 R-type implementation

Figure 7 shows the R-type implementation, the multiplication and division operations are implemented in a separate unit. PC value will be PC plus 4 and control unit provides the signals for register file and all the Muxes. The memory controller controls the access to PRAM and to the external IO devices via bus interface. The ALU output will be selected from the last mux and the data will be written into the register file with respect to the Clock edge.

I-TYPE Implementation:

Consider an example, ***addi rd,rs1,0x20*** instruction. The decoder gets the 32-bit instruction data and using that it extracts all the register addresses, immediate values and sends it to the control unit. This control unit select the immediate based on the instruction type. For the I type instruction, the data for the *rb* input of ALU should be the immediate data.

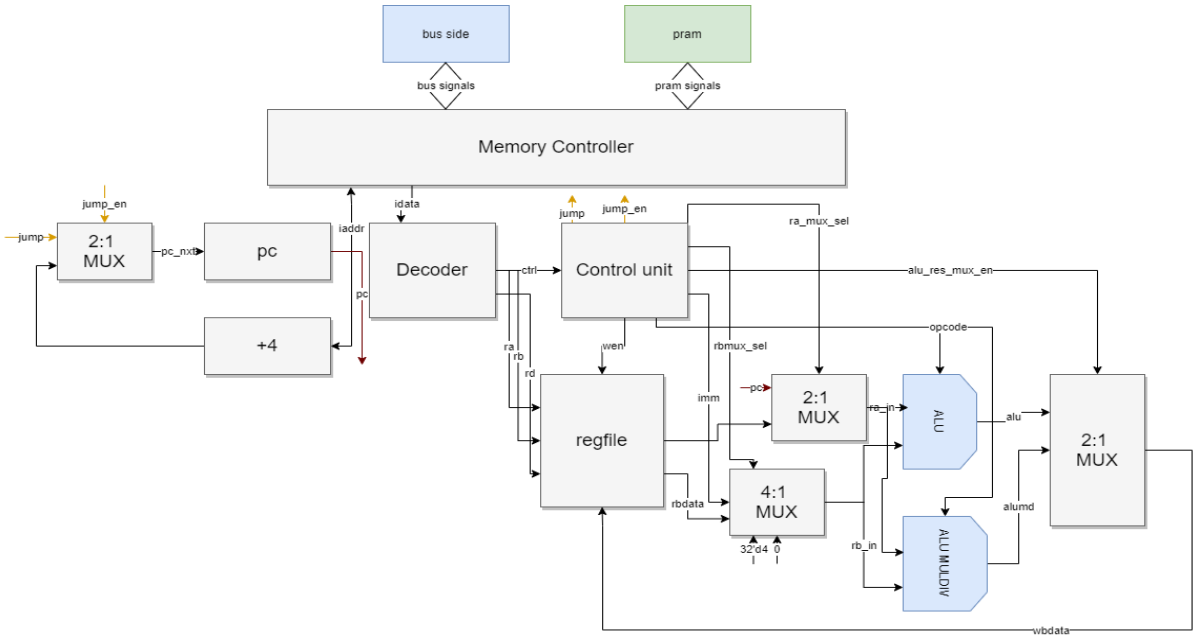

Figure 8 I-type implementation

The JALR is an I type instruction. PC value is used as an input to the new MUX for *ra_data*. The PC value along with the addition of 4 is stored in the register file and the *pc_next* will be updated with the addition of *ra_data* and immediate value which is given to the input of the PC via the mux by controlling the jump signal. Adder inside the control unit is used for the jump calculation.

J-TYPE Implementation:

Consider an example, ***jal rd,offset*** instruction. In this type PC value will be added with the immediate data and will be used for the jump. Immediate value is selected from the 4:1 MUX.

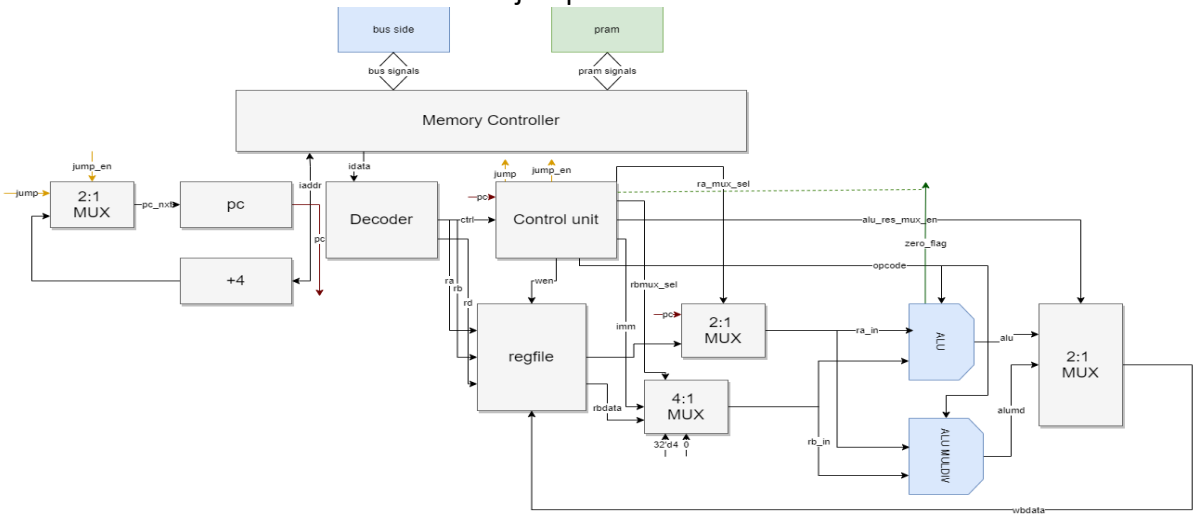

Figure 9 J-type implementation

B-TYPE Implementation:

Consider an example, ***bne x5, x0, label*** instruction. the register x5 should be compared to zero and should jump to the specified label if it's not equal to zero. If the branch is satisfied, then the immediate value will be added to the PC and the *pc_next* value will be updated. For the branch decision, comparators are used inside the ALU and the result will be on the zero-flag output which is given to the control unit.

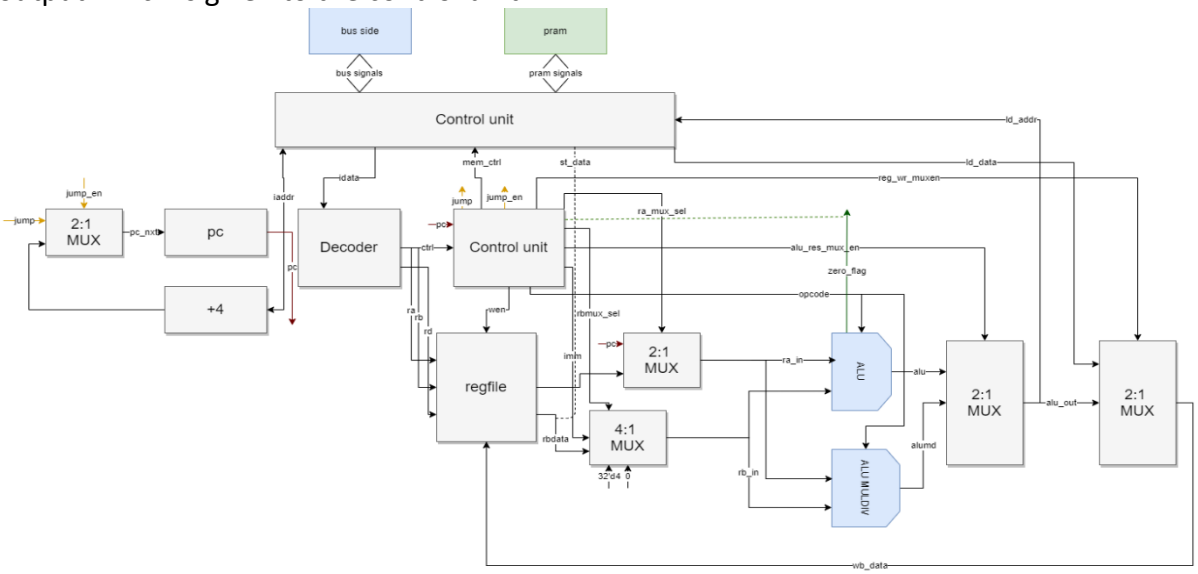

Figure 10 B-type implementation

Load Instruction is also implemented in the above data path. For the load instruction, the output of the ALU will be the address for the load which will be given to the memory control unit. This unit decides whether to access the external storage or internal PRAM for the load. Once the data is available, a new mux at the end of the data path is added to select the load data which will be stored in register file.

S-TYPE Implementation:

Consider an example, ***sw x2, (x1)0*** instruction, the instruction takes the value present in the register file one as the base address and adds zero to get the final destination to the Data Memory and stores the register file 2 value into the memory.

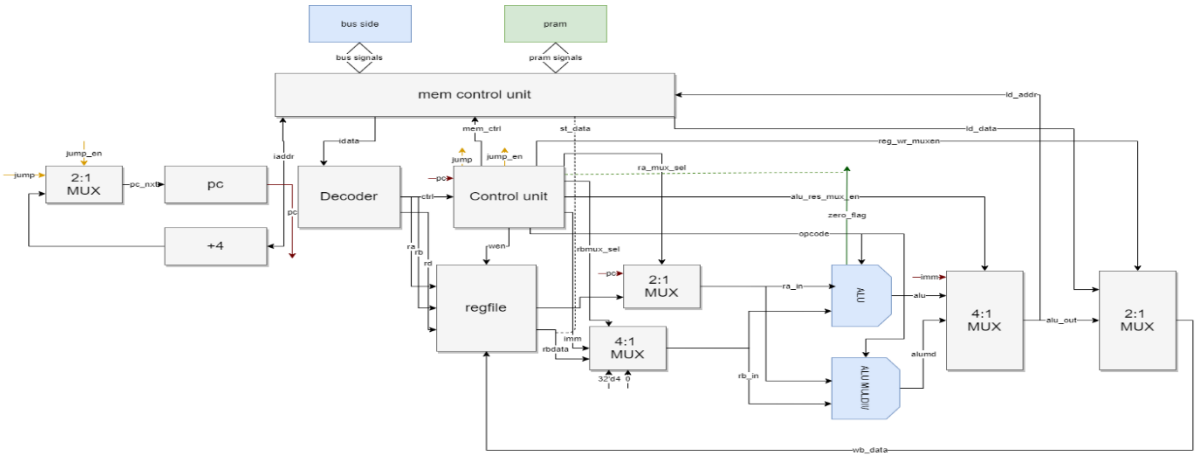

Figure 11 S-type implementation

LUI-TYPE Implementation:
Consider an example, ***lui rd,imm*** instruction, the register file is directly stored with immediate value. So, the ALU output mux gets the immediate data directly.

AUIPC-TYPE Implementation:
Consider an example, ***auipc rd,imm*** instruction, the register file is stored with PC value added with immediate value.

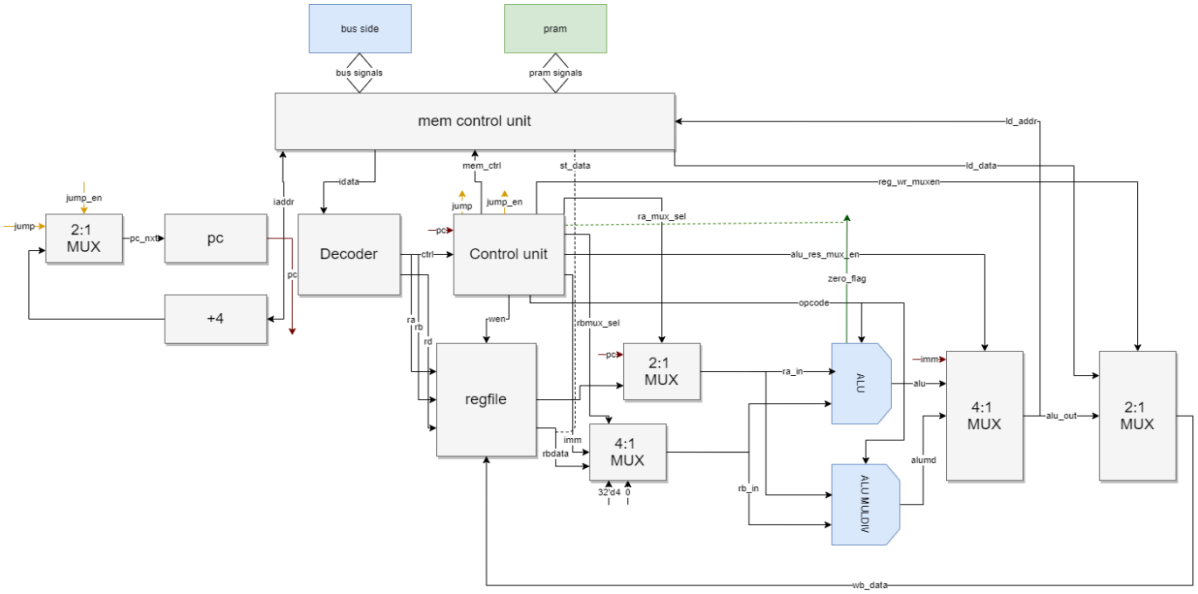

Figure 12 LUI and AUAIPC implementation

CSR Implementation:
All CSR instructions are implemented with additional registers called control status registers. The ISA implemented here supports only machine mode and it contains 5 important CSR's(mepc, mcause, misa, mtvec, mstatus).

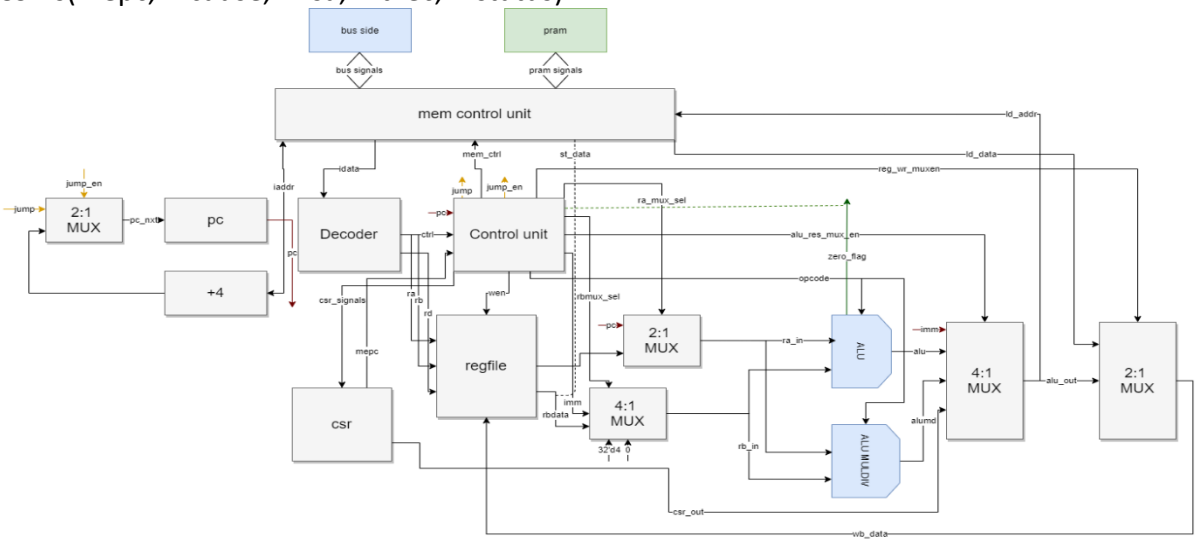

Figure 13 CSR implementation

All PC values used here were one cycle delayed values because of the synchronous memory used since it needs one cycle extra for the instruction data to return from the memory controller.

2.8 COMPLICATIONS:

In the above design, there were few complications and redundant registers which were used for the PC and control unit had extra adders to calculate the jump value which resulted in more dispersed branch decisions. Also, for load and store operations where we need to wait for the data availability, acknowledgment signals were used. However, since we didn't use any states, we needed a work around to hold the instruction during the wait for the data.
The modifications which were done for the design are:
Inclusion of Branch Unit: Because of the synchronous memory, if the PC value is given directly to the memory controller, then the PC value must be latched so that the instruction obtained from the memory and PC value are synchronised. Instead, a branch unit was designed, and the next PC value was given to the memory controller and to the fetcher unit so when the PC was updated the instruction was available from the memory. In this way, all branch decisions were integrated in one block including the jump and the pc+4 computation.

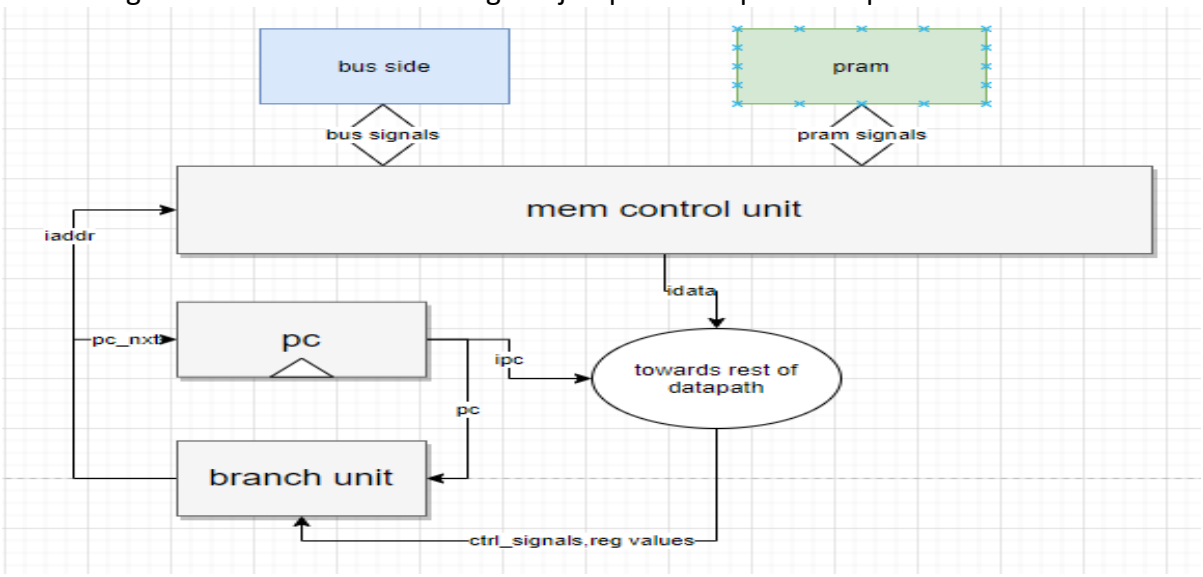

Figure 14 Branch Unit implementation

Inclusion of Interrupt control: The core supports 2 non maskable interrupts. With the help of *mstatus* control register, Machine Interrupt Enable(*MIE*) bit was used for the control of interrupt. The incoming interrupt signals from the external environment was handled as soon as the completion of the current instruction and if the core was handling the previous interrupt then any upcoming interrupt will be on hold which is reflected on the interrupt enable bit. An acknowledgement signal will be sent as soon as the core starts handling the interrupt.

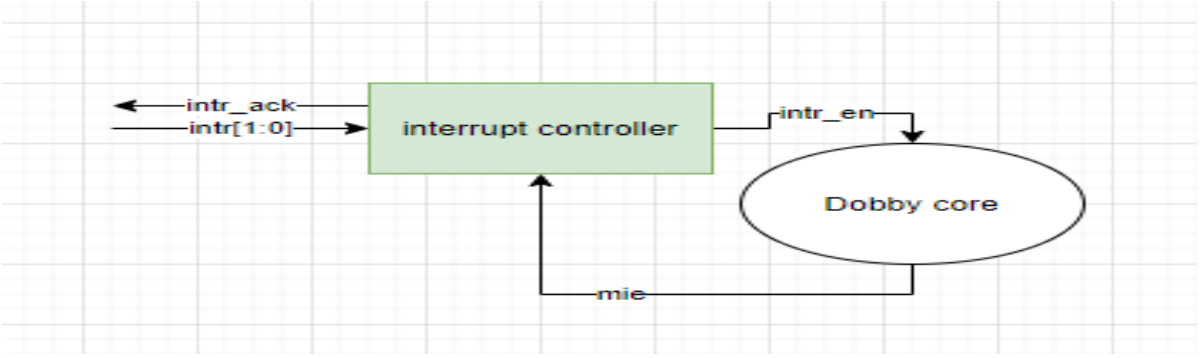

Figure 15 Interrupt control

Instruction Hold register: During the Load and store instructions, all the control signals required for load and store operations must be the same until the data is available or an acknowledgement about the store. So, the decoder should always decode the same instruction until the load/store completes before it can fetch the next one. Any illegal instructions during the wait will result in wrong control signals and a bad execution. To hold an instruction, a 32-bit register was used which was written during the load and store Instruction and read until the acknowledgement of the instruction completion was obtained. A simple 2:1 MUX was used with controls decided based on load/store halt signals.

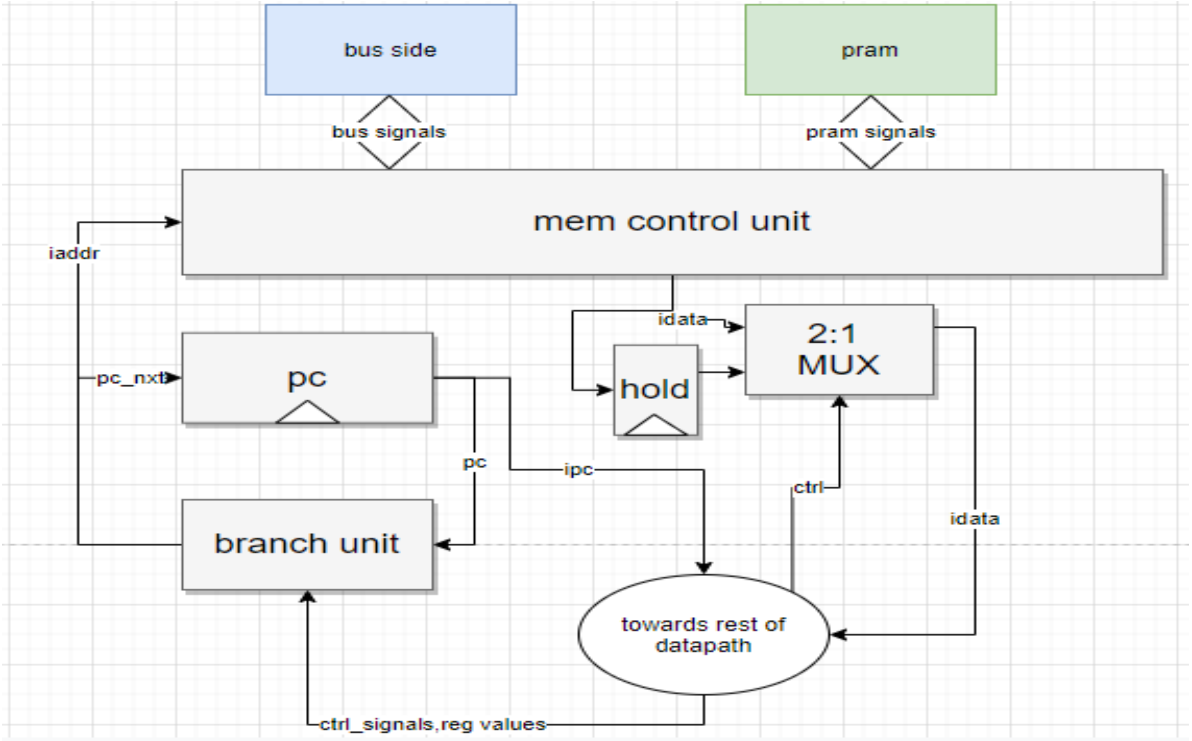

Figure 16 Hold register Inclusion

Init Controller: According to the specification, the processor needs to load the internal PRAM via the external bus from the external memory as soon as there is a hard reset. When reset is triggered, the internal PRAM(0000-03FFF) will be filled with the instructions via the custom BUS implemented. A state machine was designed for this. It had two states: IDLE and COUNTER. By default, the state machine will be in *idle* state and when the reset is enabled it jumps to the *counter* state and then generates the memory address for the copy of instructions into the internal PRAM. Once it reaches the end of the PRAM or an interrupt was triggered then it switches back to the *idle* state again waiting for the reset signal. A *load_from_external* signal was used in the counter state and the core starts its execution as soon as the load is complete.

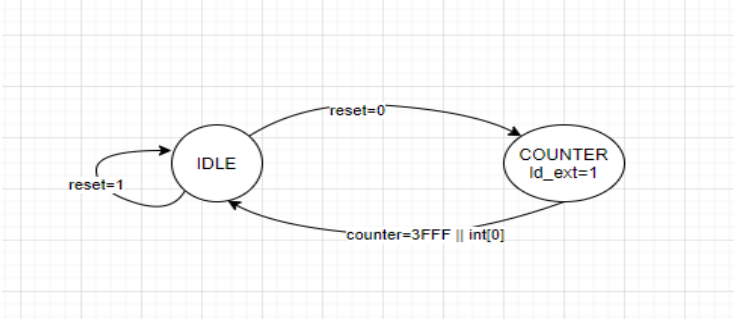

Figure 17 State machine for Memory initialisation

2.9 SINGLE CYCLE PROCESSOR

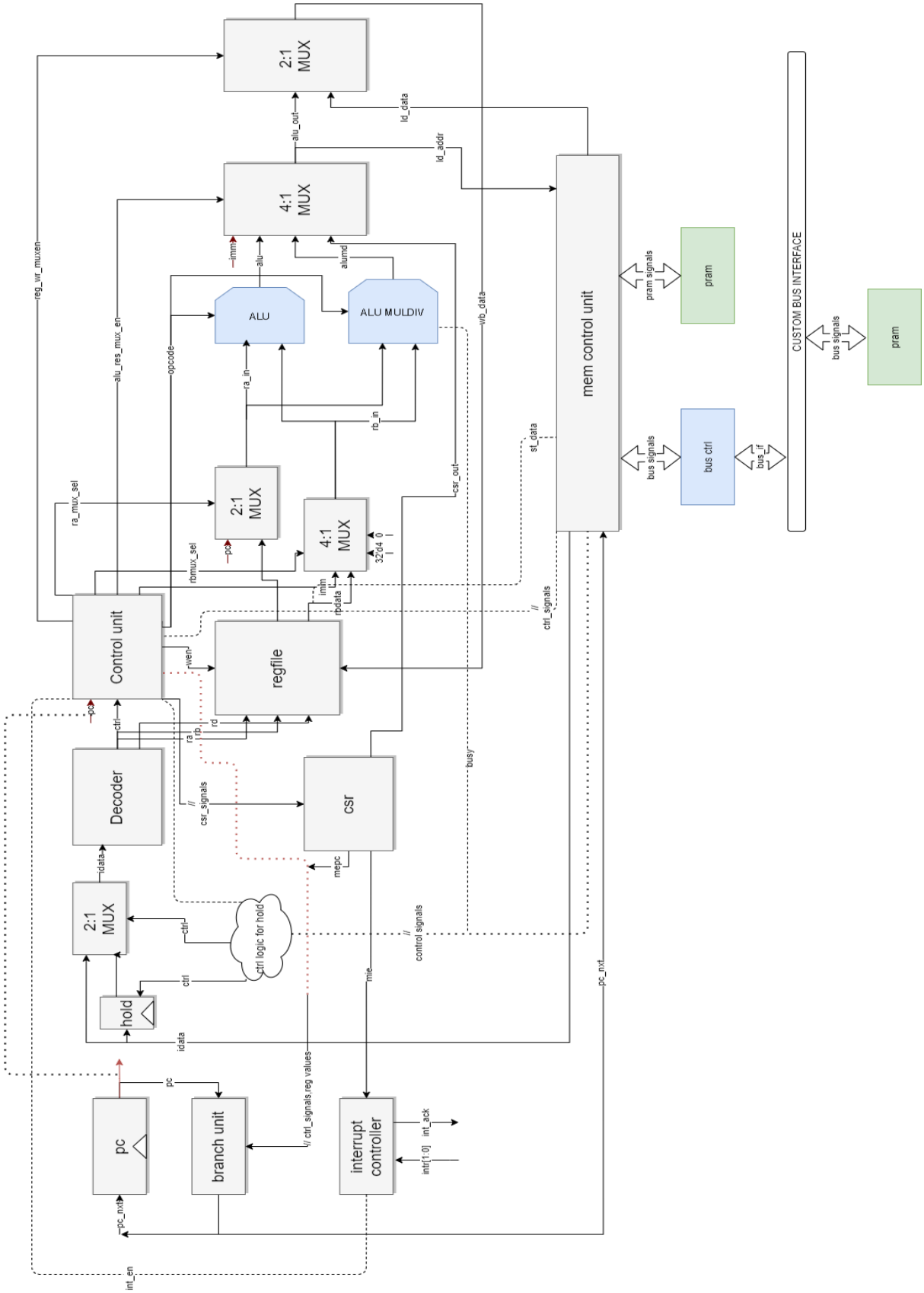

Figure 18 Datapath of a Single cycle processor

2.10 TOP MODULE

Dobby Top consists of pads module and the core instantiated. All the signals handshaking with the dobby core and the external modules happens via the pads. These are the Analog modules which are provided from the library. For the HDL checks, these pads are not included as it generates errors.

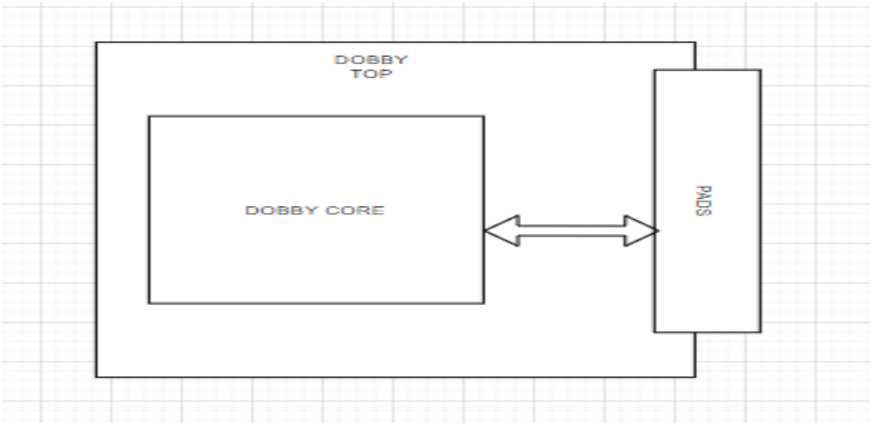

Figure 19 Dobby Top view

External Memory Fetch:

Since SRAM was used as internal program memory, instructions were fetched faster without much latency but if instructions are fetched from external memory then it depends upon slave memory latency and there will be wait cycles until a valid instruction is available. External fetch cannot be pipelined because when there is a jump, a wrong instruction would be fetched instead of target address instruction. Hence, to avoid this complication a state machine was implemented for external fetch. Until the fetch address is less than 3FFF, state will be in Internal fetch state which uses internal memory as program memory works normally as explained in previous section. Once fetch address is greater than 3FFF, *f_bus_en* will be asserted and the control will be shifted to external fetch. *Bus_Fetch_Latch* state issues fetch request to external memory, once valid data available on bus, a *bus_fetch_ack* (separate for bus side fetch) will be issued and this fetched instruction will be executed in the *Bus_Fetch_Decode* state and next pc address for will be decided in this state. After executing the instruction, if still the Fetch address is >3FFF again it will be sent to Bus_Fetch_latch state and the cycle repeats. Figure 20 shows the state diagram of implemented state machine.

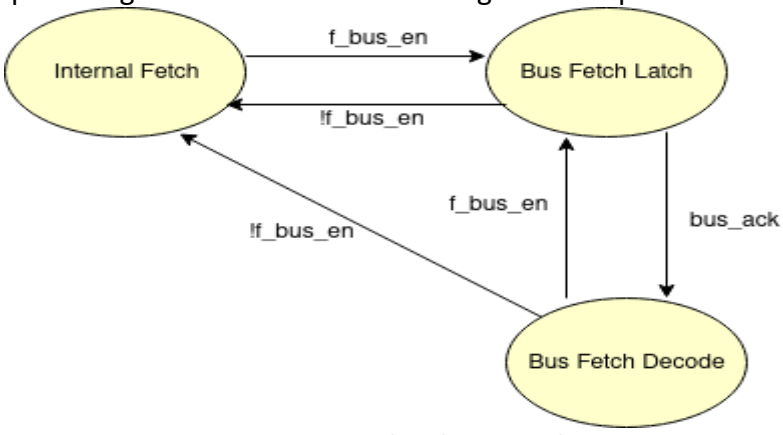

Figure 20 External Fetch State Machine

Chapter 3: Simulation and Verification

The Single cycle architecture for Dobby SOC module has been verified by creating test bench in Verilog. For verifying, the DUT is instantiated at first, then inputs like bus ready, interrupts were randomized and sent to the DUT. The design was verified by writing all instructions in Assembly code and few algorithms using C code. Please use the top modules ***dobby_top*** and ***dobby_pipefinal_top*** for verification of single Cycle and Pipelined processor respectively.

3.1 Synthesizability Check

Before going to verification, the code written must be checked whether it is synthesizable or not. There should not be any HDL errors and warnings which affect the functionality of design. Figure 21 shows that there were no errors for the proposed design and all the critical warnings that were supposed to be mandatorily handled were addressed. Few warnings which are not critical was not handled as it won't affect the functionality.

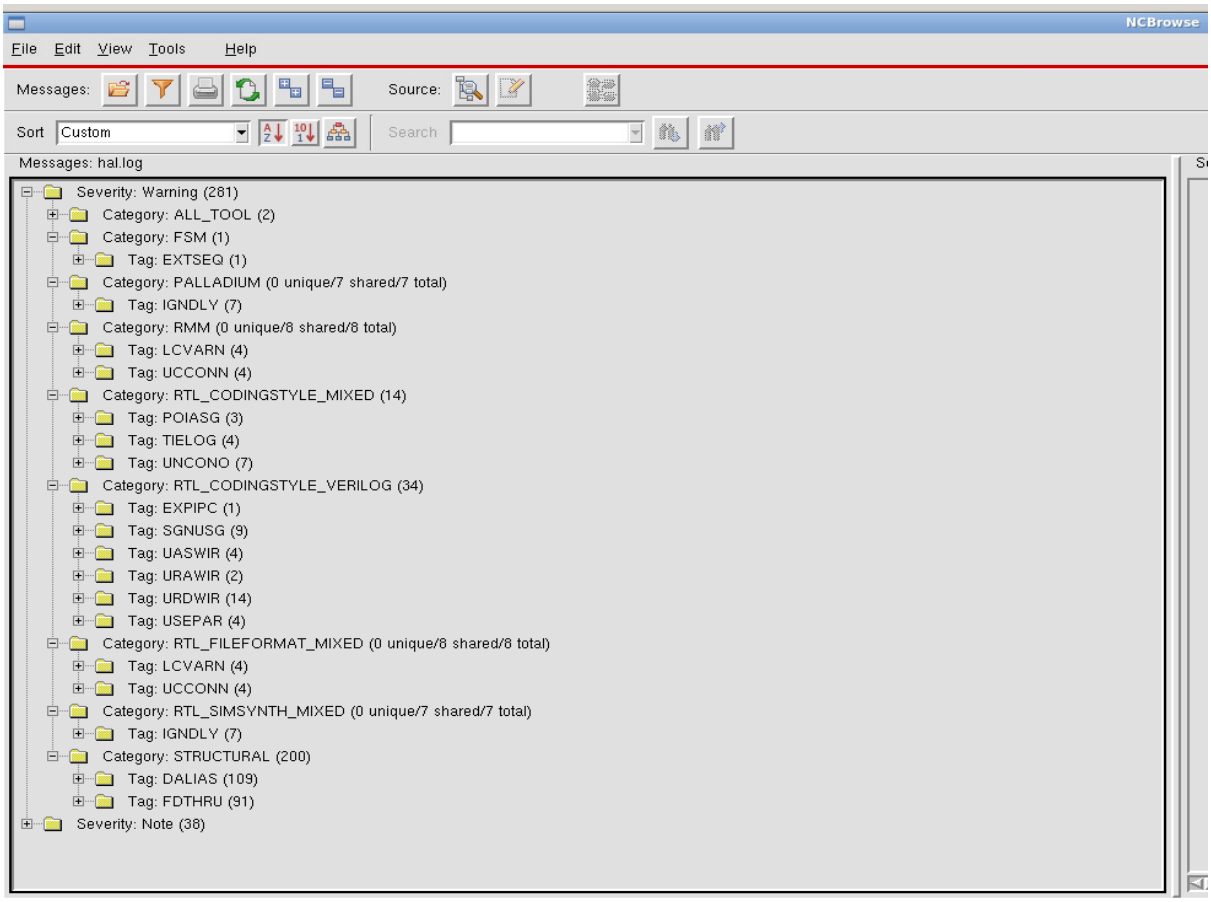

Figure 21 Warnings

The warnings like DALIAS, FDTHRU can be ignored. And warnings like UNASWIR, UNRDWIR were intended when we have additional bits for calculation, and they are not assigned in code to any other variable. These warnings won't affect anything. Also, some warnings like LCVARN - using capital letters for instance declaration but this warning was due to SRAM macro which

couldn't be changed. Similarly, all the warnings which couldn't be resolved through code changes were ignored as these warnings don't affect the functionality.

3.2 Simulation

To make actual use of the Dobby RV32IM processor, a complete software ecosystem was provided. This ecosystem consisted of linker scripts, boot loader and makefiles. The software ecosystem is based on the GCC compiler for RISC-V.

Using this software ecosystem, assembly code was compiled which covered all instructions of the core. A *core.mem* file was generated from the assembly code using make export command. The generated *core.mem* file was not in the format required to load into the slave memory. To handle this, a python script(*memscript.py*) was written to convert that text arrangement as required for loading into slave memory. This script generates *instructions.mem* which will be used to load into the external SRAM.

The core and external memory (sram) were initiated in test bench as shown in figure 22 by giving clock, reset and other required inputs. All the instructions were executed, and it was verified that the correct results were obtained for each of the instruction.

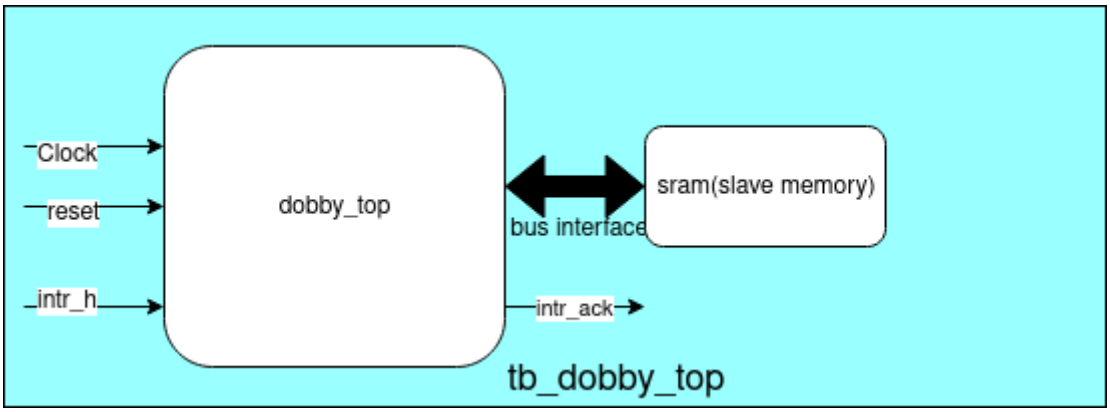

Figure 22 Testbench Top module overview

Figure 23 shows a small snippet of the assembly code which consisted of all instructions (Refer svn repository for entire Assembly code). And the test cases were written in such a way that all the corner cases were covered.

```
#addi to all registers to initialize the values
_START: addi x0,x0,5 #corner check shouldn't write
        #all registers add 1
        addi x1,x0,1
        addi x2,x0,1
        addi x3,x0,1
        addi x4,x0,1
        addi x5,x0,1
        addi x6,x0,1
        addi x7,x0,1
        addi x8,x0,1
        addi x9,x0,1
```

Figure 23 Code snippet

After simulation of the design, coverage analysis was done using *icncsim* coverage command. We were able to achieve functional coverage of 97% as shown below. The missing coverage of 3% were because of implicit defaults in case statements. Thus, all possible instructions were included through which we were able to achieve greater coverage.

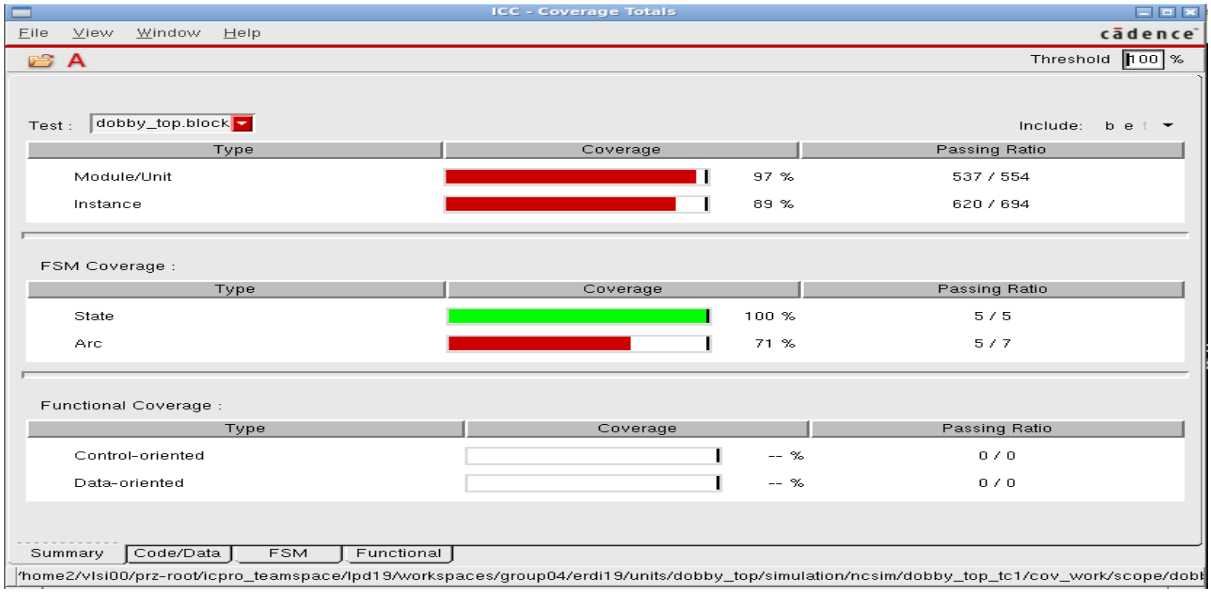

Figure 24 Coverage report

Then we also simulated few algorithms implemented using C code. The following algorithms were used for verification of the core

- Factorial of a number
- Check whether the given number is prime number or not
- Bubble sort algorithm
- Binary search

All the results obtained were verified and can be found in the Results section later. All results use a return address location(by default it was 0X8000).

3.3 Design Goal and Work split

We choose the design goal as **Speed** with keeping checks on area and power. The entire work of the processor was split between us.

- Modules such as branch unit, ALU, Fetch state Machine, Control Status Register(CSR) unit and Fetcher unit were implemented by Dilip.
- Modules such as Control Unit, Initialisation state machine, Register file, Hold Instruction and Interrupt control unit were implemented by Pruthvi.
- Rest of the modules were implemented together.

Chapter 4: Synthesis

After verifying results, the code was synthesized using Synopsis design Compiler tool. Synthesis was setup by mapping fsa0a_c_sc (core logic library), fsa0a_c_io (io pad library), sram_lib (SRAM macro library), setting fsa0a_c_sc as the target library and also defining the operating conditions for best, worst and typical conditions. Then the constraints were defined in the constraints.tcl file as shown in figure 25. This constraint file included the clock period, clock latency, input, and output ports delay. Execution of synthesis was done using ***icdc compile*** command.

```
################################################################################
# definition of clocks
################################################################################
create_clock -period 20 -waveform { 0 10 } [get_ports {I_CLK}] -name CLK50
set_clock_transition 0.3 [get_clocks {CLK50}]
set_clock_uncertainty 0.2 [get_clocks {CLK50}]
set_clock_latency 1.5 [get_clocks {CLK50}]

################################################################################
# Set I/O delays
################################################################################
set_output_delay -clock [get_clocks {CLK50}] 10 [all_outputs]
set_input_delay -clock [get_clocks {CLK50}] 10 [remove_from_collection [all_inputs] [get_ports "I_CLK"]]

################################################################################
# Set input, inout and output port conditions
################################################################################
# Additional load values for all inputs

set_load 1.000  [all_inputs]

# Capacitance for all output or inout ports.
set_load 1.000  [all_outputs]

#
# Design attributes
#
set_dont_touch pads_i/*
```

Figure 25 Constraint file snippet

After completion of Synthesis it would generate timing, area, and power reports. The timing report would include worst delay path (critical path) of 3 groups namely reg2reg, in2reg and reg2out. Here, the Slack must be positive or else it would be considered as timing violation. The power and area report estimates the total power consumption and estimated area before routing of the design respectively.

4.1 POST SYNTHESIS ANALYSIS

After the initial synthesis of the processor, the initial frequency achieved was around 16.6 MHz and the area required was very high. The reason for this was the implementation of multiplication and division operations on the core. Initially we didn't use any algorithms for these and let the tool perform the optimization for the required MUL/DIV hardware. Since it tries to complete both operations with only one cycle each, this was the reason for our low frequency operation. As MUL/DIV are special operations, they require different hardware compared to normal operations. For this, we implemented Booth's algorithm for these operations.

BOOTH's ALGORITHM: An algorithm which multiplies two numbers via additions and subtractions only along with shift operations. The algorithm is as follows[6]:

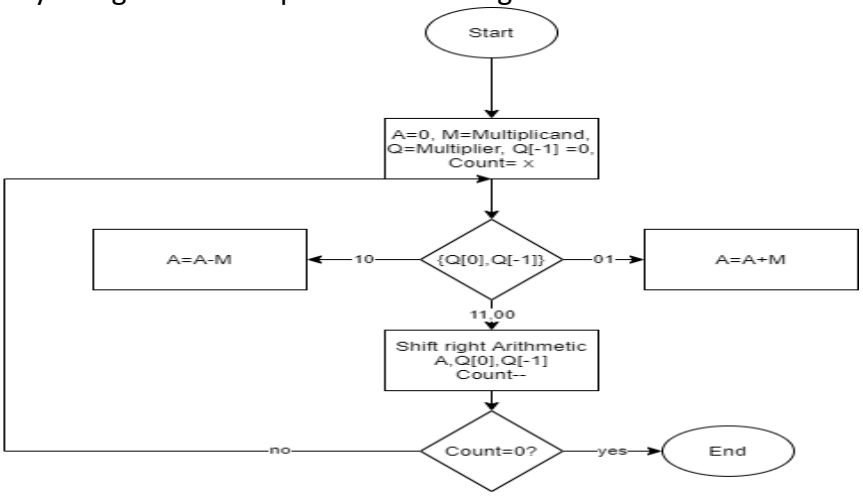

Figure 26 Booth's Algorithm

The algorithm starts with accumulator initialization to zero. Multiplier and multiplicand are copied to Q and M respectively. Count will be the number of bits. Q[-1] represents bit to the right of LSB and initialize to zero. For every iteration of the count, the concatenated value of Q[0] and Q[-1] are checked with all 4 combinations and the corresponding operations are carried out which is shown in Figure 26. Once the loop is completed, accumulator holds the multiplication result. One drawback with this algorithm is, it requires N number of cycles which is equal to the number of bits of operands. To overcome this, a Radix-4 modified Booth's algorithm was used[7]. Also, partial products were computed in parallel. To support all RISC-V multiplication operations, 32 bits operand were extended to 34 bits with upper bits as sign bits. The modified algorithm uses 3 bits for the decision of which operation to do. The table is as shown below:

LSBs (3 bits considered)	Partial product Operation
000	Nothing
001	+M
010	+M
011	+ 2*M
100	- 2*M
101	-M
110	-M
111	Nothing

Table 3 Modified Booth's Algorithm

All the partial products are added at the end by shifting according to their positions. Using this we were able to achieve the multiplication operation in single cycle. Area will be a bit high since we are computing the partial products in parallel, but this gives a very high speed up.

BOOTH's RESTORATION ALGORITHM: This algorithm was used for the division operation[8]. It is like multiplication, but we can't do parallel computations as the above algorithm. One bit is taken at a time, so we need 32 cycles for 32 bits. Unsigned division was implemented and for signed operations pre and post modifications were used. To reduce the cycles, we did an analysis to choose the number of cycles for division. a graph of frequency vs Area was plotted for different division cycles which is as shown below:

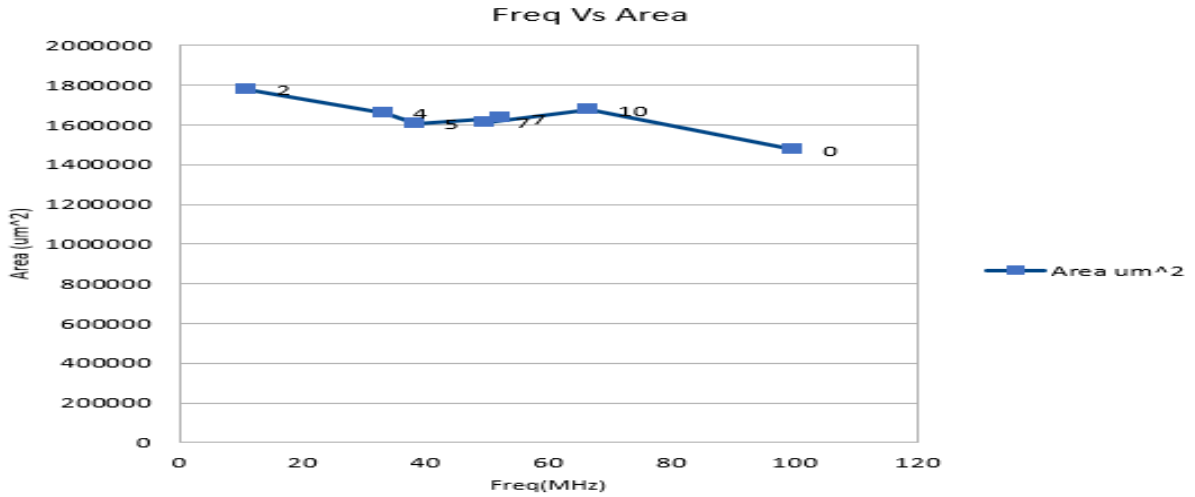

Figure 27 Frequency vs Area Graph with Division operation

Based on results from the figure 27, 7 cycles were chosen for the division operations with a maximum possible frequency of 52.6 MHz . If the core's application has less division operations, then it can be set to 10 cycles per division and frequency can be increased up to 66.1 MHz . The final frequency was selected to be at 50MHz with 7 cycles for division.
Also, from the power reports generated by the tool, we saw that a memory was consuming more power due to all the four SRAM blocks active. This was changed by controlling the chip enable based on the address.

Parameters of Single cycle processor	Values
Frequency achieved	50MHz
Power after synthesis	3.6282 mW
Area after synthesis	1.6133 mm^2
Load-Store (towards external)	3 cycle Latency(with slave ready=1)
Load-Store (towards internal PRAM)	2 cycle latency
Division operation	7 cycle latency
Bus side fetch	2 cycle latency(with slave ready=1)

Table 4 Single Cycle Processor Details

4.2 PHYSICAL DESIGN OF THE CHIP[9]

After the completion of synthesis, the generated netlist will be used for further back end process. The physical design starts with the floor planning of the chip and ends with the routing of the nets followed by all verification steps before it is sent to the foundry.

Floor Planning: In this step, the overall area of the chip which will be used in market is decided. Various blocks and their positions are decided in this step. Due to stringent time to market, many of the blocks are reused from other projects. For example: A memory block, clock gating cell etc will be designed before and just included in the project where it is needed. These are called macros or IP blocks from various vendors. In Floor planning, these blocks are preplaced at required positions. A chip has two area defined which are Core Area and the Die Area. Die Area is the overall chip area and core is where the logic cells are placed. The area between Core and the Die are reserved for pin placements of the chip. This area is reserved, and logical blockage is placed(IO fill) just to make sure the tools further won't place any logic in that area.

Power Planning: This is also a step during the floorplan which creates the power distribution network(PDN) to power each components of the design. In this, a power ring (also called core ring) is created which are connected to VDD and VSS pads. Power straps are also created so that the logical cells draw the required supply in the vicinity which reduces the RL effect of the wires. Best power planning results in less heat dissipation, less IR drop and avoids Electron Migration(EM) problems.

Placement: After the PDN is created, all the standard cells are placed on the available rows in the floorplan area. Placement is carried out in two steps: Global and Detailed.
In Global placement(Fast placement), the tool's speed of placing the cells are high and just tries to find optimal positions for the cells. This is soft placement as the blocks are not fixed to the positions. There can be slight overlap or misalignment when placing between rows. Then a detailed placement takes over the global placement and fixes all the problems related to placement adhering to the rules of the global placement.

Clock Tree synthesis(CTS): Clock is the most important signal in any chip as it defines the working of all registers in the design. The main aim of any design is to have clock latency zero so that all the registers in the design receives the clock at the same time without any variations. If any timing mismatch, then registers might latch the data at different times resulting in wrong functionality. So, there is requirement to develop a tree which routes the clock to all registers such that the skew is almost zero. In this step, clock buffers are introduced into the clock path to ensure signal integrity and better driving capability. H Tree, X trees etc are some of the algorithms used.

Routing: In this step, all the blocks which are placed are now connected by physical wires. The timing analysis which was done till now are with ideal routing(zero delays with all connections). Now we must do the timing analysis once to confirm that real wire delays will not cause any timing violations. When performing routing all the Design Rule checks(DRC) are enabled which ensures proper routing so that it can be fabricated without any issues. Other checks: Signal Integrity , Cross talk , Detailed power analysis, Diode insertions to remove Antenna effect of metals and many more are some of the few extreme checks which are used now a days.

All the above steps mentioned are automated in the *icpro* design flow. Many *tcl* scripts are used to interact with the tools and to perform any modifications if required. The steps used are:

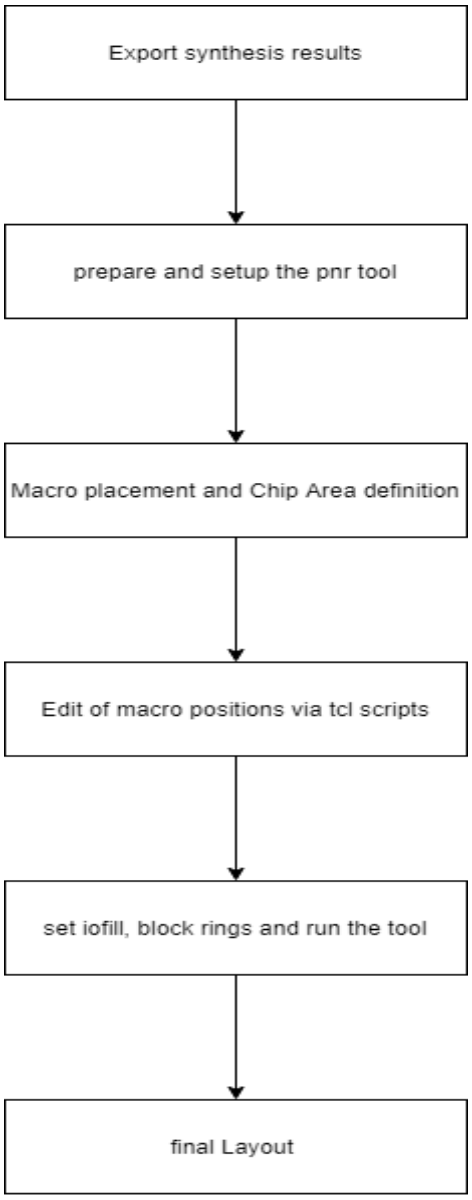

Figure 28 Place and Route Flow

Parameters Used	Values
Floor Area	width=1600u , height=1710u
Core to IO spacing	All 4 sides = 15u
Power wire	width= 5u , spacing=0.44u
IO fill	Enabled
IO placement	Optimised by Tool
Macros (SRAM Block 0,1,2,3)	X=265u constant ,Y0=265u, Y1=565,Y2=865, Y3=1165
Routing Halo	0.01u

Table 5 User defined inputs for PNR Tool

Core density: Any design after the PNR flow will Give details about the core density which represents the actual amount of area used for the entire design. Typically, it should be between 70 to 80% for proper fabrication. If the density is less than 70%, problems related to on chip variation and irregularities on the surface of the Silicon may result. For the RISCV Dobby chip designed we had a density of **76.35%**.

Along with the final layout of the chip, it also generates timing reports at each step (Pre route, CTS, Routing). This consists of Worst Negative Slack(WNS), Total Negative Slack(TNS) And all timing violating paths. The slacks must be positive and there should not be any violations.

For the verification of the design, the results from the pnr tool were exported and once again the simulation was run using the final layout to check the results. All the assembly level instructions of the RISC instruction set and different algorithms like bubble sort, factorial, element search etc were used for the verification.
The simulation results for the assembly instructions and the waveforms of verification of different algorithms are added at the end of the report. The final chip of the RISCV Dobby Single Cycle processor is as shown in Figure 29:

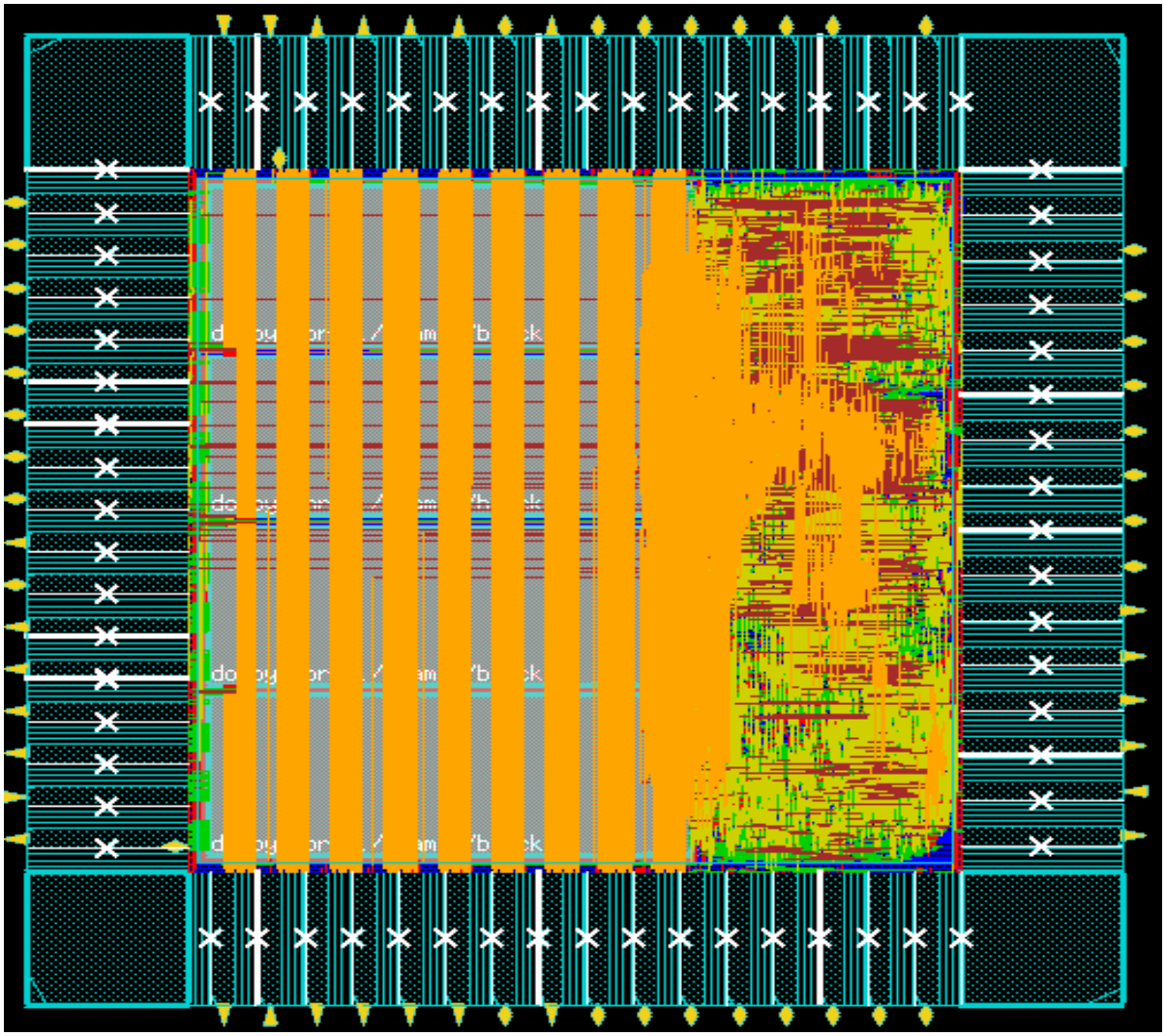

Figure 29 Single Cycle Processor

Chapter 5: PIPELINING

Pipelining refers to a set of registers that is inserted between the blocks to divide them into different stages. Pipelining is a powerful way to improve the throughput of a digital system[5]. Pipelining breaks the critical path into smaller critical paths hence reducing the clock period which in turn increases frequency of operation. All the pipeline registers are clocked synchronously. The input of the pipeline register in the current stage is the output to the next stage. This helps in preventing loss of information of an instruction as it go from one stage to next.

In case of processors the different stages are Instruction Fetch, Instruction Decode & Operand Fetch, Execute, Memory access and Write Back[5]. Generally the pipelining of processors is done with 5 stages but there is no strict rule, it can also be done with 2 stages to any number of stages but one should take care of data and control hazards(discussed later) caused by each stage. Figure 30 represents the 5 staged pipelined architecture of processors

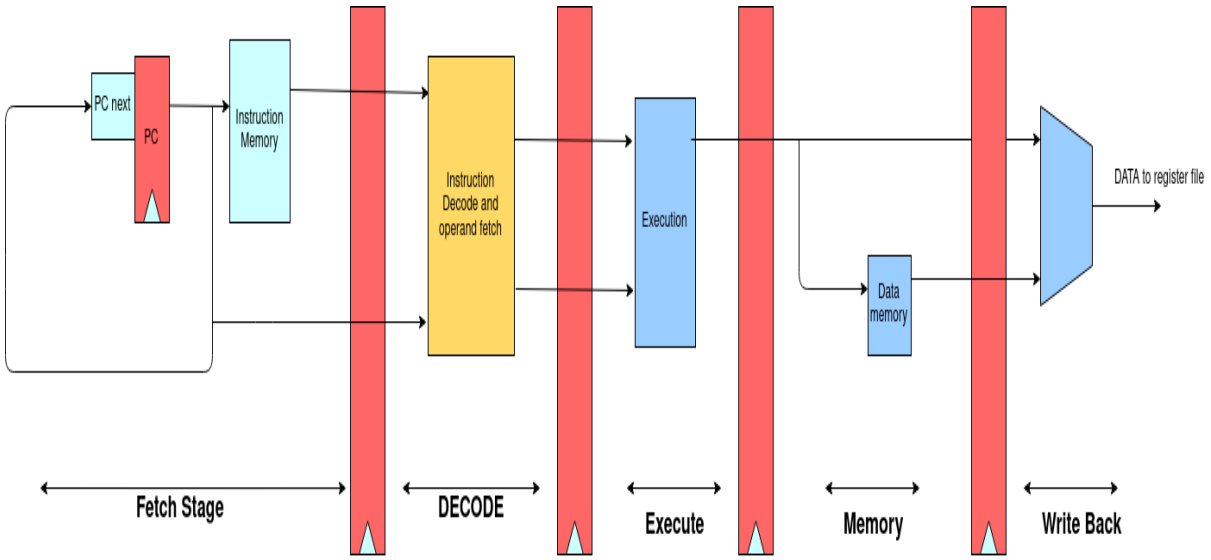

Figure 30 Pipelined Architecture

Pipelining a processor enables the processor to begin executing next instruction before the current one is complete. This enhances the speed/performance of the processor as each stages of the processor can operate different parts of different instructions independently at a single clock cycle[5]. The sample timing diagram of an ideal pipelined processor is as shown in figure 31.

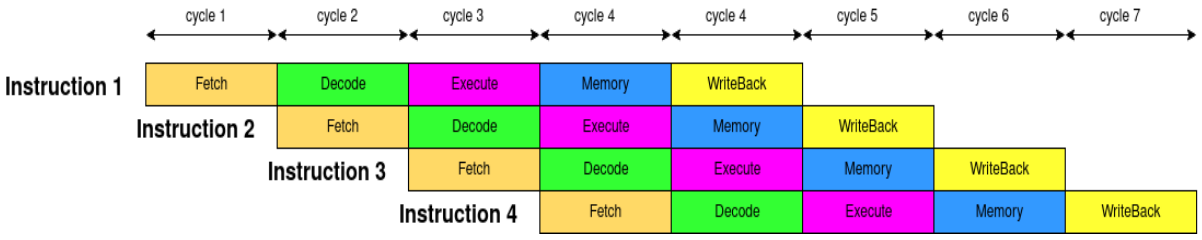

Figure 31 Instruction Flow in Time domain

In our task, a 4-stage pipelined processor was implemented and verified which will be explained in the next section.

5.1 PIPELINED PROCESSOR FOR HIGH THROUGHPUT AND SPEED

The single cycle processor which was implemented had a maximum frequency of 50 MHz . To get the highest speed and better throughput, a 4-stage pipeline design was implemented. The pipeline design had Instruction Fetch(IF), Instruction Decode(ID), Instruction Execute(IE), Memory Writeback(MWB) as four stages.

Instruction Fetch Stage: This is the first stage in the pipeline processor and consists of PC, MUX, PRAM blocks. The reason for inclusion of the PRAM in the first block was to get the instruction in one cycle. Because of the synchronous property of the memory, an inverted main Clock was used which gets active in the negative edge of the main clock. This also helps in sending PC address to the memory instead of pc_next as in single cycle processor. The memory requires around 3ns to produce the output which is within the half cycle of the clock.

By default, pc address will be incremented by four. During branch or jump, multiplexer control signals are changed to get the branch address. All pipeline registers have clear and hold options which will be used during hazards. A Fetch stage can also be divided into prefetch and fetch by using the memory in a separate stage and use main Clock. Again, this would increase the number of stages to 5 but with less speed up ratio. So, only 4 stages were used.

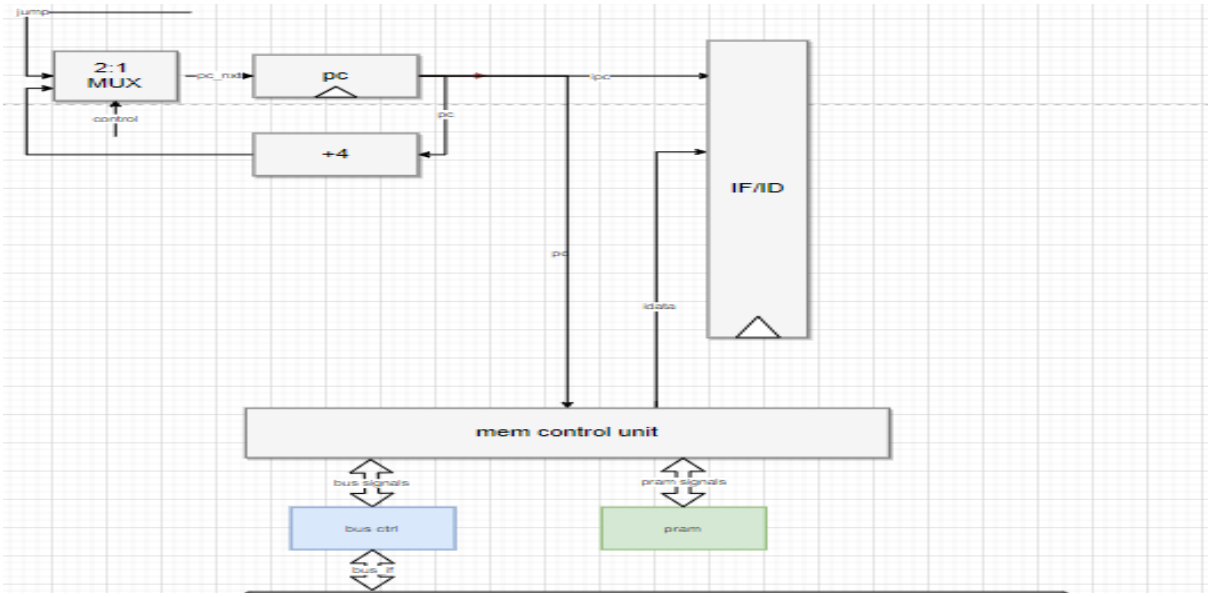

Figure 32 First Stage Resources

Instruction Decode Stage: This is the second stage in the pipeline and consists of Decode, Control, Branch unit, CSR, Register file and Interrupt control. The incoming instruction will be fed to the decoder which provides register file addresses and the opcode required for the control unit to generate control signals. All the CSR related operations are done in the second stage. For the exchange of CSR register values with the register file, CSR write happens in second stage whereas register file write happens in 5th stage and that is the only difference. PC address is given to the branch unit, which decides the next PC value based on instruction. If there is a jump or branch, PC value will be updated from the branch unit output via MUX Control. When the branch decision depends on the future stage results, then the pipeline is

halted or results forwarded(based on which stage), and the fetched instruction will be flushed if the branch/jump happens.
Interrupt control unit is also here which will be handled after the current instruction in the decode stage proceeds into the next stage. During interrupt handling, fetching is halted. Once the control returns from the interrupt handler the pipeline is resumed. The two multiplexers in the figure 33 are the forwarding multiplexers which will be used when destination address of the register file in the further stages matches the operand's read addresses.

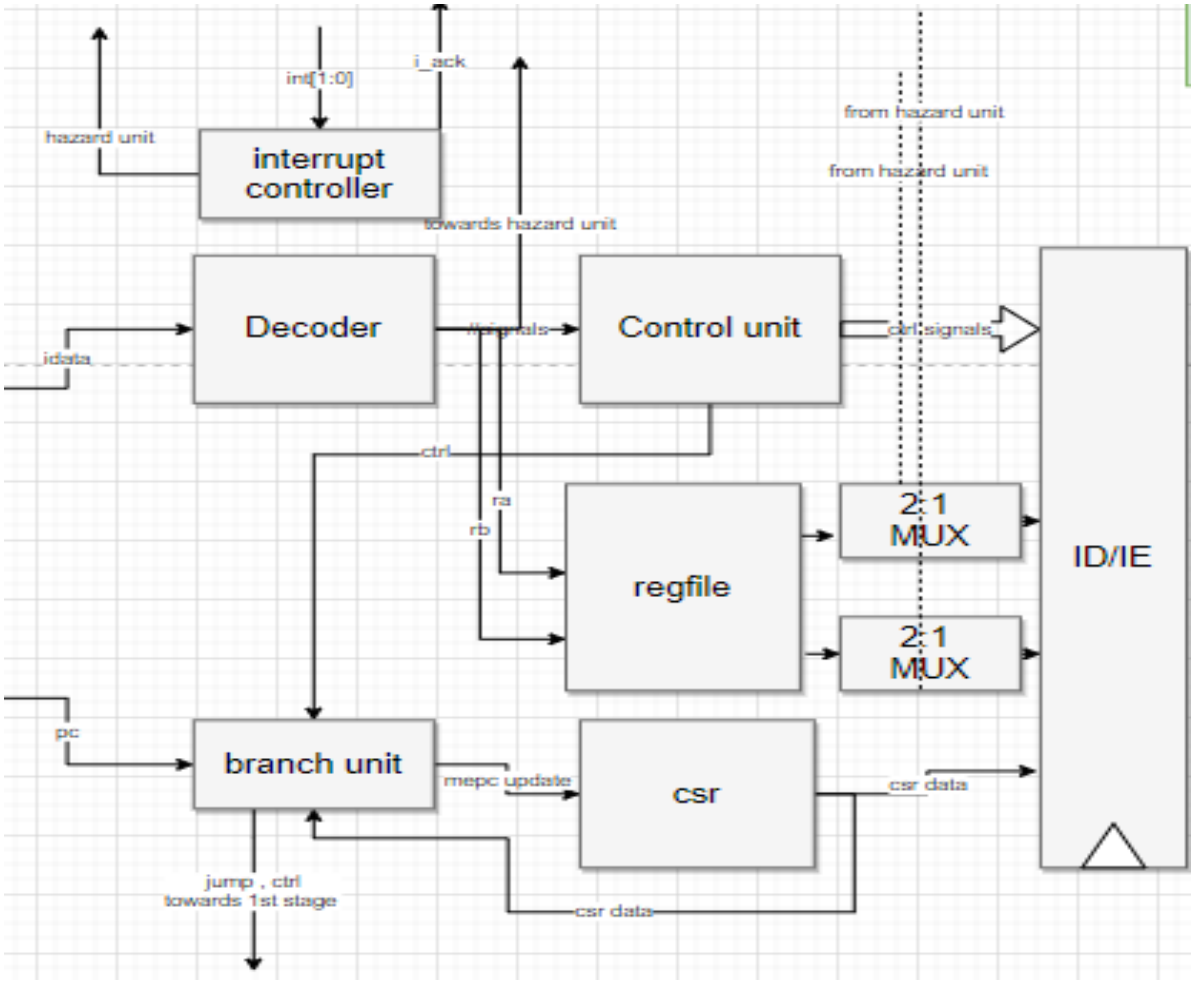

Figure 33 Second Stage Resources

Instruction Execute Stage: This is the third stage in the pipeline and consists of ALU, ALU_MULDIV and several Multiplexers for selection of inputs, data forwarding and output selection. Since the processor was optimised for high frequency, it was not possible for the multiplication operation to execute in single cycle as the combinational path delay was too high to fit in the 3rd stage. Hence, 2 cycles were used for multiplication operation and 7 cycles values for division operation. The pipeline will be halted during multiplication and division operations that is for 2 and 7 cycles. This wait would have been possible to eliminate using multiport register file which again results in area trade off. Hence it was not implemented.

Control signals for the multiplexers comes from the second stage to select appropriate data for the execution units. Destination address of the register file along with the register write

signal will be sent to the hazard unit for its check with the register file read addresses of the second stage. If it matches, then the execution stage result will be forwarded to the second stage. Forwarding multiplexers are also used in the execution stage to get the data from the 4th stage. All these forwarding multiplexer control signals comes from the Hazard unit.

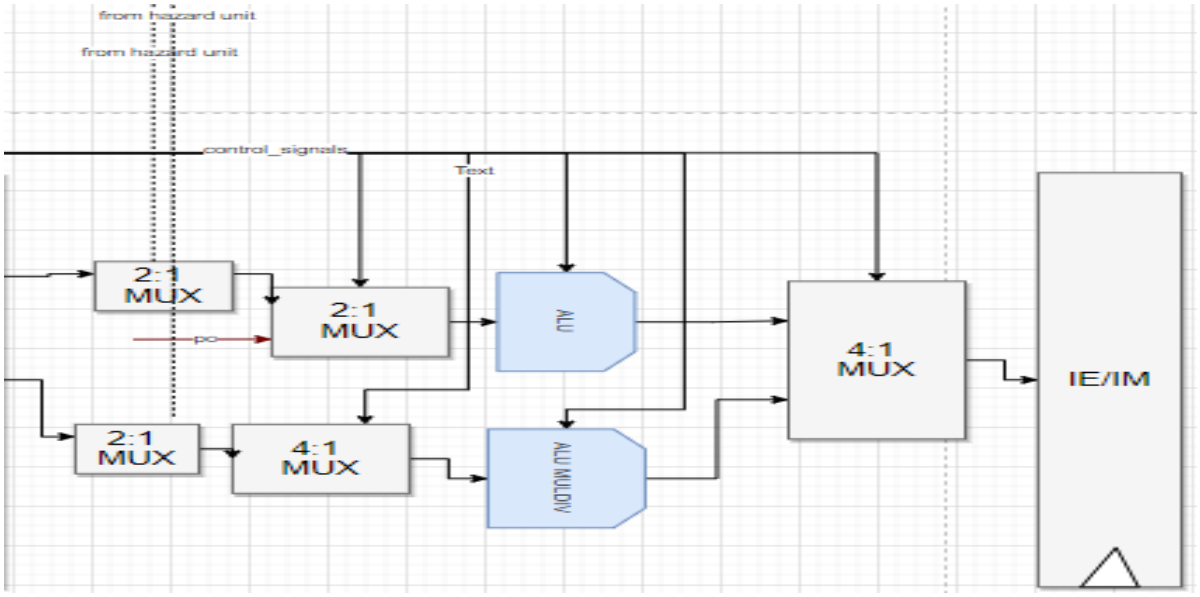

Figure 34 Third Stage Resources

Memory Writeback stage: This is the fourth stage in the pipeline and consists of Memory access and Multiplexers for write to the register file. Separate memory stage was not used since the memory is now operating at the negative edge and would complete the execution within the upcoming cycle of the register file write. For bus side load and store operations, the pipeline execution would stall until the acknowledgement signal comes from the bus interface. The control signal for the result selection from the multiplexer comes from the control unit which is pipelined through the stages. The register file contents would be updated/written with the correct result in the upcoming positive edge. Thus, the instruction would complete its execution in 4 cycles which results in 4 stage operation.

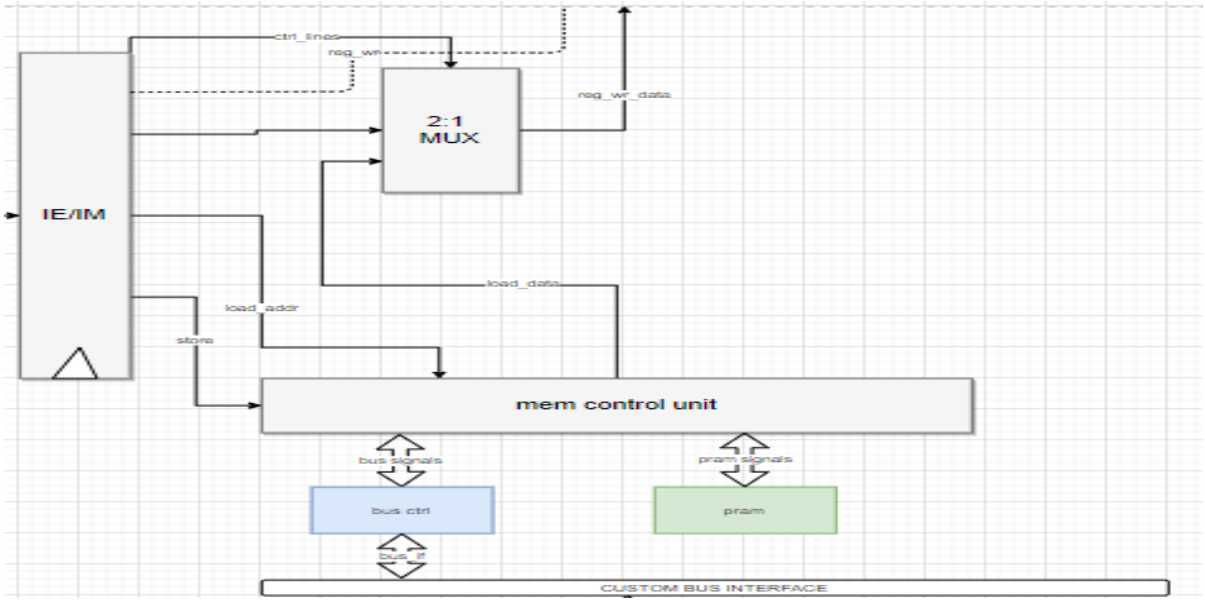

Figure 35 Fourth Stage Resources

5.2 SOLVING DATA HAZARDS AND CONTROL HAZARDS[5]

Pipelining can improve the performance of the processor by increasing the throughput, the average instructions completed per clock cycle. However, it also introduces hazards to the processor. The hazards can be categorized into two different types: Data hazards and Control hazards.

Data Hazards:

A data hazard occurs when an instruction tries to read/write a register that has not yet been written/read by a previous instruction. It can be either Read after write(RAW), Write after read(WAR), Write after Write(WAW) hazards.

The hazards that occur when one instruction writes to a register and subsequent instructions reads the same register is called a read after write (RAW) hazard. Consider two subsequent instructions as

```
instruction 1: add r1,r2,r3
instruction 2: and r5,r1,r2
```

Assuming a 5-stage pipeline the result of instruction 1 will be written to register(r1) in fifth cycle(writeback stage). But the instruction 2 will read wrong value in cycle 3(decode and operand fetch stage). This can be handled either by stalling or forwarding.

The other two Hazards WAR and WAW occur in case of out of order processors which looks ahead across many instructions to issue or begin executing independent instructions as rapidly as possible by handling all dependencies. Consider the below program

```
instruction 1:  lw r1,40(r0)
instruction 2:  add r2,r1,r3
instruction 3:  sub r1, r4,r5
instruction 4:  and r9,r10,r11
instruction 5:  or  r12,r13,r14
```

Assuming a 5-stage pipeline and 2 cycle latency for load instruction(instruction 1) since instruction 2 depends on r1 it will be issued after two cycles hence instruction 3,4,5 execute independently but again instruction 3 and instruction 2 have some dependencies instruction 3 can update r1 only after instruction 2 reads the value so this illustrates the WAR hazard.

A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it.

```
instruction 1:  add r1, r0, r2
instruction 2:  sub r1, r0, r2
```

Both the instructions are writing to r1. The final value in r1 should come from instruction 2 according to the order of the program. If an out-of-order processor attempted to execute instruction 2 first, the WAW hazard would occur.

Control Hazards:
Control hazard, which is also known as branching hazard often take place when there is a branch. The problem arises when the branch is taken, the program flow will be incorrect due to the instructions that should not be executed had been fetched into the fetch stage of the pipelined processor before the branch condition is evaluated at execution or decode stage

Control hazard can be caused by the following events:

- Conditional branch
- Unconditional branch
- Interrupts or Exception

There are several ways can be used to solve this problem, such as stall the next instruction until the branch is completed, flush all the inappropriate stages in the pipeline or through hardware implementation. Also, by using branch predictors.

5.3 Branch Prediction

Ideally pipelined processor should have a CPI of 1 but since branch decision takes place in decode stage or later stages extra instructions will already be fetched so few cycles will be wasted in flushing those instructions. This misprediction penalty is fine if there is only one branch instruction and only one flush but if we have loop with more iterations, misprediction penalty also becomes larger which directly affects the throughput of the processor. To address this problem, branch predictors are used in pipelined processors.

There are mainly two kinds of Branch prediction: Static and Dynamic branch prediction. Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing[10]. Instead, it predicts the outcome of a branch only based on the branch instruction. Dynamic branch prediction uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. Forward branches are difficult to predict without knowing more about the specific program. Therefore, most processors use dynamic branch predictors. Dynamic branch predictors are again split as one-bit and two-bit dynamic branch predictors. Two bits dynamic is implemented using 4 states- strongly taken, weakly taken, weakly not taken, and strongly not taken[5] . A misprediction in 2-bit branch predictor happens only for the last branch of a loop. But it requires large amount of hardware for creating lookup table(Branch target buffer) hence we chose to implement one-bit dynamic branch predictor.

A one-bit dynamic branch predictor stores the branch target address first time when the branch is taken and predicts that it will do the same thing the next time. So, when the loop is repeating, if branch instruction is fetched out then by default the next fetch address will be branch target address stored during the first branch true condition. This holds good until the branch condition is true but for last loop execution until we have a misprediction. Thus, misprediction in one-bit dynamic branch predictor occurs on the first and last branches of a loop. This is implemented with 2 states taken and not taken. Figure 36 shows the implementation of Branch predictor block in fetch stage in our project.

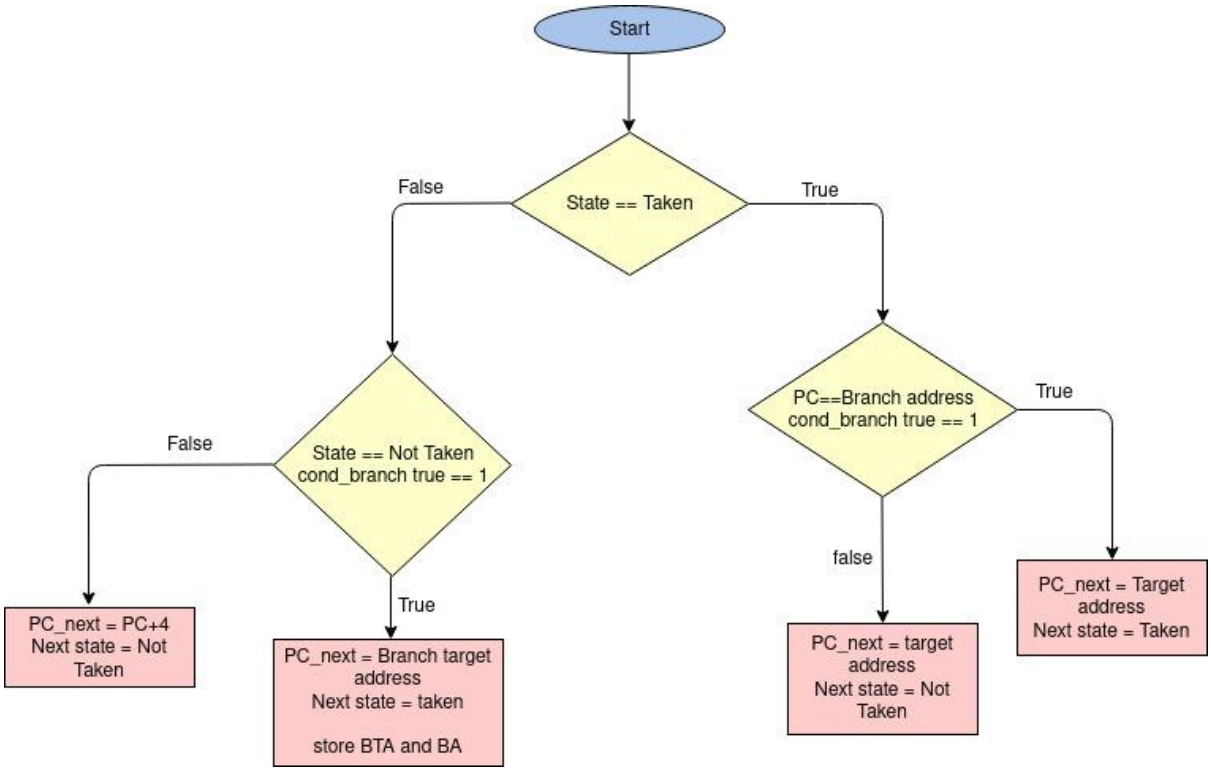

Figure 36 Branch Predictor logic

5.4 HAZARD DETECTION UNIT

To handle the data source and control hazards, a separate Hazard detection unit was designed. This unit keeps the track of the destination address of the register file and compares it with the read register address to enable the data forwarding whenever it is necessary. It is also used to halt the processor when certain operations need to be completed. It uses the signals from different stages and operates completely combinational to control the pipeline registers and forwarding multiplexers.

Forwarding logic: The register data along with their addresses will be forwarded to the execution unit. These addresses will then be forwarded to the hazard unit which compares it with the destination address in the memory writeback stage. If the destination address matches with any of the operand addresses and the register write enable is high, then a forwarding control signal for the multiplexer will be enabled. Similarly, it is also checked for the decode stage with the memory writeback stage. If the addresses match, Then the forwarding signal for the multiplexers in the decode stage will be enabled. The following block shows high level code description of the forwarding logic.

```
if((i_ra_addr_e == i_dest_addr_wb) && i_rf_wen_wb)
        forward_a =1;

if((i_rb_addr_e == i_dest_addr_wb) && i_rf_wen_wb)
        forward_b =1;

if((i_ra_addr_d == i_dest_addr_wb) && i_rf_wen_wb)
        forward_a =1;

if((i_rb_addr_d == i_dest_addr_wb) && i_rf_wen_wb)
        forward_b =1;
```

Halt Logic: The pipelined processor designed has halt conditions for 3 cases which are due to load and store operations from the bus side , waiting for the interrupt instruction and due to multiplication and division operations.

During the external load and store operations, the processor needs 3 cycles for the completion of the instruction. During these cycles, the processor activates the stall signals for the pipelined registers. Once the acknowledgement signal from the bus interface is available, halt signals will be released, and the processor continues the operation.

```
if(stop_fetch){

        hold_f =1;
        hold_d =1;
        hold_e =1;
        hold_wb =1;
}
```

During the wait for interrupt instruction, once the decode is completed the processor waits for the interrupts from the upcoming cycle. To ensure correct operation, once the wait for interrupt instruction completed, the pipeline register following the decode stage is cleared and all the previous stages are kept in hold. After the interrupt is handled, the processor will resume its operation.

```
if(halt_pc_wfi) {

        hold_f =1;
        hold_d =1;
        hold_e =0;
        hold_wb =0;
        clear_e =1;
}
```

During the multiplication and division operations, as the signals enter the execution stage, the multiplication-division unit latches the signals and enables the busy signal. This busy signal is used in the hazard unit which halts the fetch, decode registers, and clears the writeback stage. This ensures that correct output will be written into the register file after the operation is complete.

```
if(i_mul_div_busy) {

        hold_f =1;
        hold_d =1;
        hold_e =1;
        hold_wb =0;
        clear_wb =1;
}
```

Branch prediction halt logic: During the branch instructions, correct operands are needed for the comparison in the branch unit. If the operand addresses match with the destination address in the execution stage, then there is a need for halt as we don't have the correct operands yet. If the branch prediction is true and there is no dependency, then we clear the decode stage pipeline register so that the correct instruction can be fetched in the next cycle.

```
if (branch_instruction) {
    if((i_ra_addr_d == i_dest_addr_e || i_rb_addr_d==i_dest_addr_e) && i_rf_wen_e)
     {
            hold_f =1;
            hold_d =1;
            hold_e =0;
            hold_wb =0;
            clear_e =1;
    }
    else
            clear_d=1;
```

5.5 Pipelined Processor

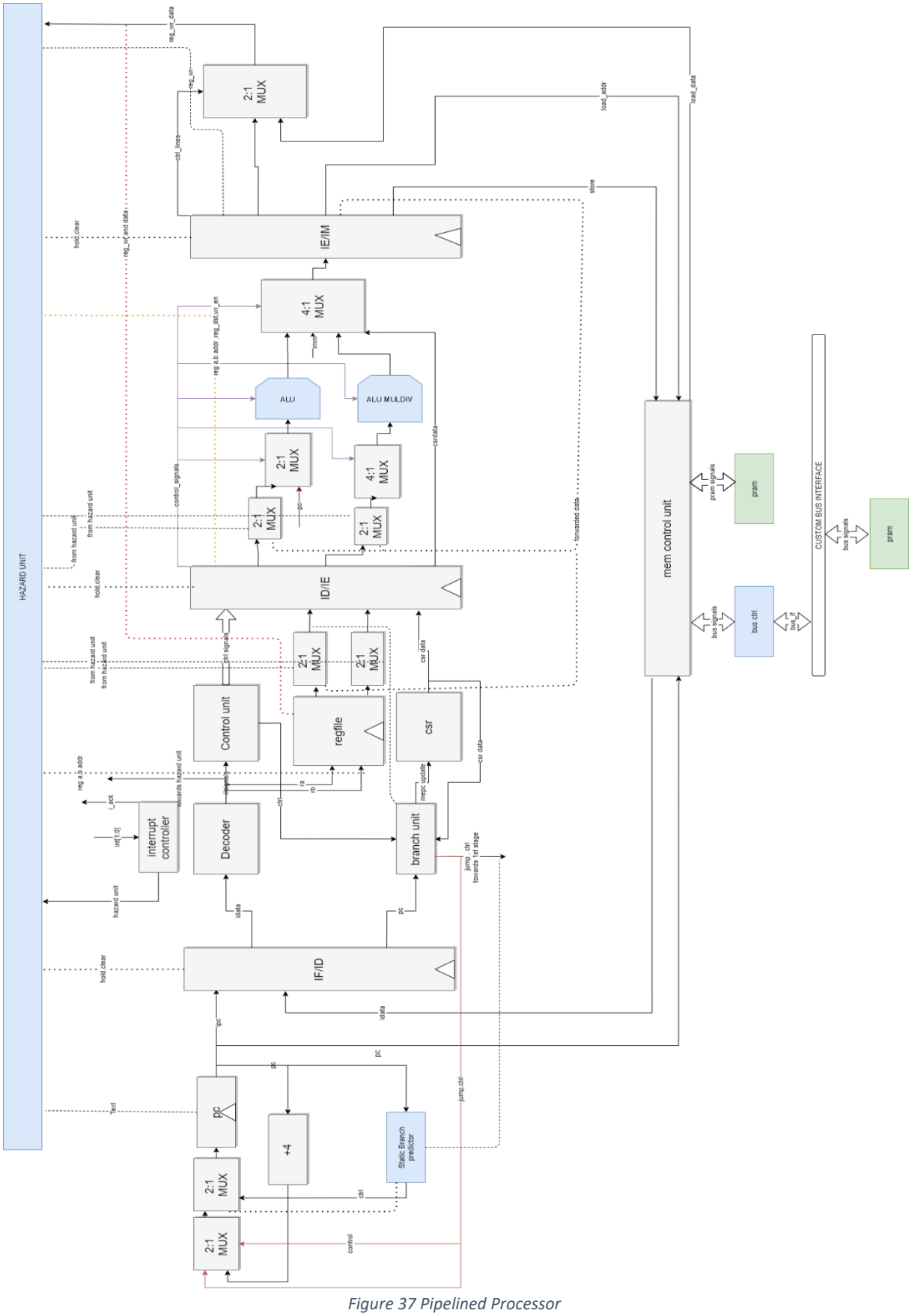

Figure 37 Pipelined Processor

Pipelined Processor Summary:

Parameters of Single cycle processor	Values
Frequency achieved	116.27MHz
Power after synthesis	10.4664 mW
Area after synthesis	1.656 mm^2
Load-Store (towards external)	3 cycle Latency(with slave ready=1)
Load-Store (towards internal PRAM)	2 cycle latency
Division operation	10 cycle latency
Multiplication operation	2 cycle latency

Table 6 Pipelined Processor Summary

Chapter 6: SIMULATION AND VERIFICATION RESULTS

A complete list of all instructions[11] used were verified on the design and different algorithms in C language were also used. The following snippets covers the simulation results on waveform window along with explanations. Showing all the instruction results is not possible due to High code density and we have covered all the important results with respect to the task description. Complete results can be checked in the project folder (***svn: group04)***.These steps were carried out both for single cycle and pipeline processor.

6.1 Single Cycle Simulation Results:

PRAM Initialization

On reset, before starting execution, the internal PRAM is initialized via external bus interface(Program instructions stored in a memory) . Figure 38 shows that the external memory data is accessed continuously and data from external memory is copied into internal program memory.

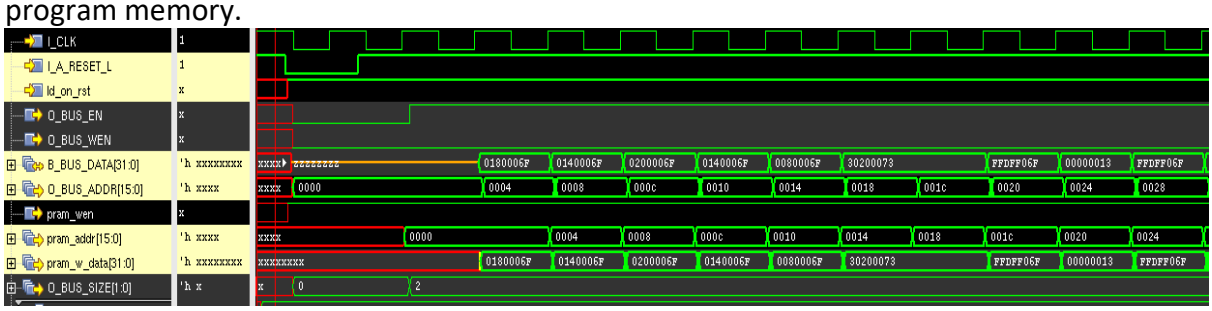

Figure 38 PRAM Initialization

The Initialization should be stopped whenever Interrupt(Interrupt zero) is asserted, or PRAM address is equal to 3FFF. Figure 39 shows when *pram_addr* == 3FFF the core starts executing the instructions from next cycle.

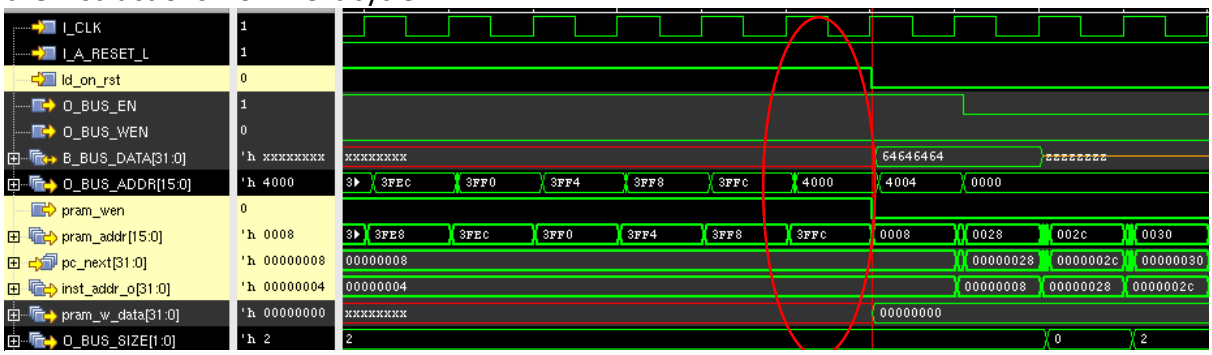

Figure 39 PRAM Initialisation completion

Single cycle Instruction

The single cycle instruction will be decoded and executed in the same cycle and results are written to register file in the next cycle. Hence for execution of next instruction the data is already available. The highlighted part in figure 40 is performing ***addi r1,r0,0*** where the result is updated to r1

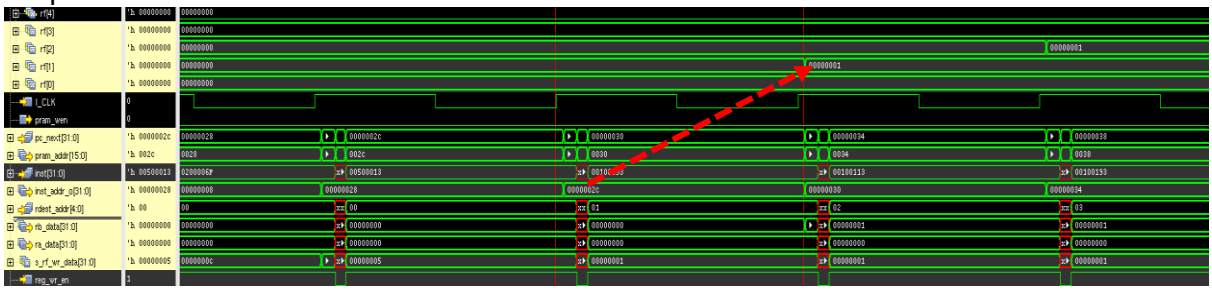

Figure 40 Single cycle Arithmetic Instruction

The highlighted part of figure 41 performs **or r1,r5,r3** where the result is updated to r1.

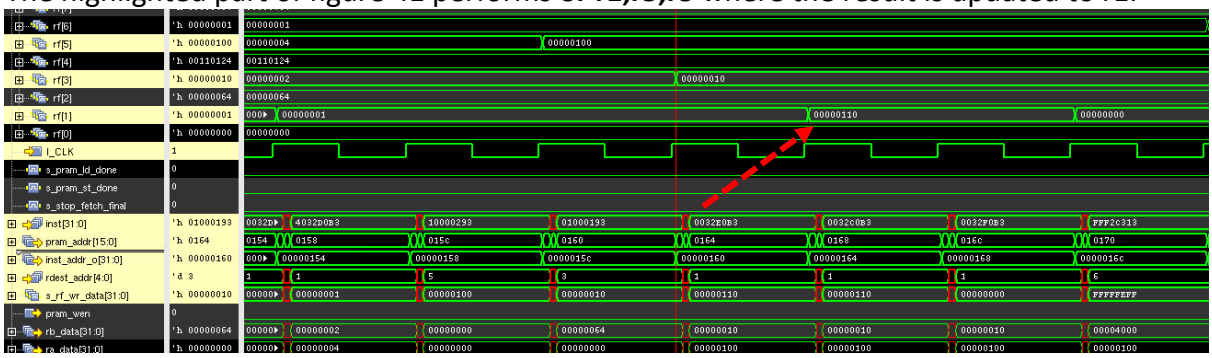

Figure 41 Single Cycle logical Instruction

Load operation:

The load instruction usually takes min 3 cycles and 2 cycles for external and internal memory access load respectively. When a load instruction is encountered, the fetch is stopped by asserting *stop_fetch_final*(through *ls_stop_fetch*) and is released when *bus_ack* is asserted in case of external memory access via bus interface or *pram_ld_done* in case of internal memory access. The execution continues normally after *stop_fetch_final* is released. Figure 42 shows the scenario of external load.

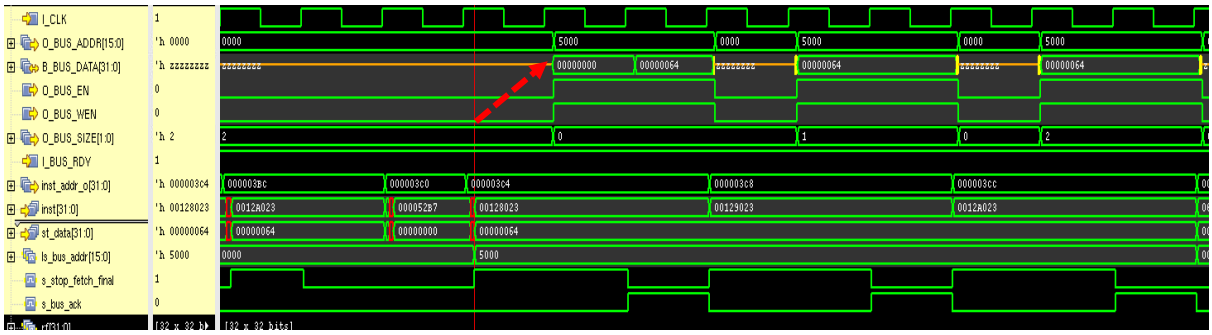

Figure 42 Load from External Memory

Figure 43 shows the scenario of load from Internal memory. In both the images the instruction data and instruction address maintain same value until the load operation is completed.

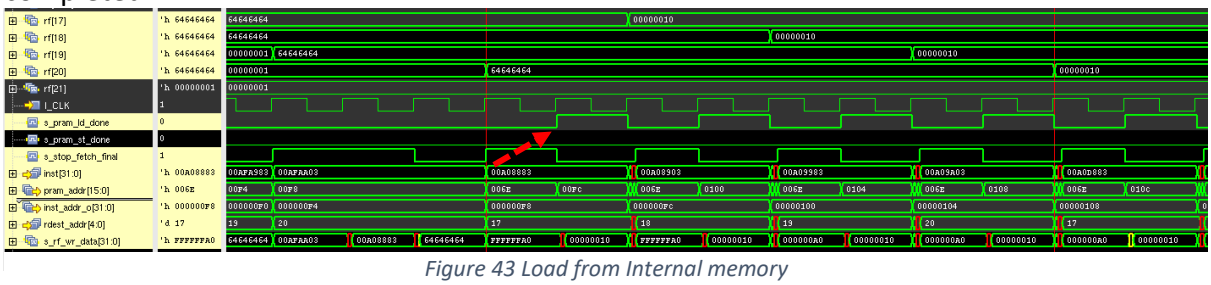

Figure 43 Load from Internal memory

Division operation:

As discussed in design section previously, to increase the speed the division was made a 7-cycle operation(in Single cycle architecture).The fetch is stopped by asserting the signal *stop_fetch_final*(through *s_mul_div_busy*). It is released when *s_mul_div_busy* is de-asserted. The execution continues normally after *stop_fetch_final* is released. From the figure 44, the instruction data and instruction address maintain same value until the division operation is completed. The operation is pipelined, and result is updated to register file only when *stop_fetch_final* is de- asserted.

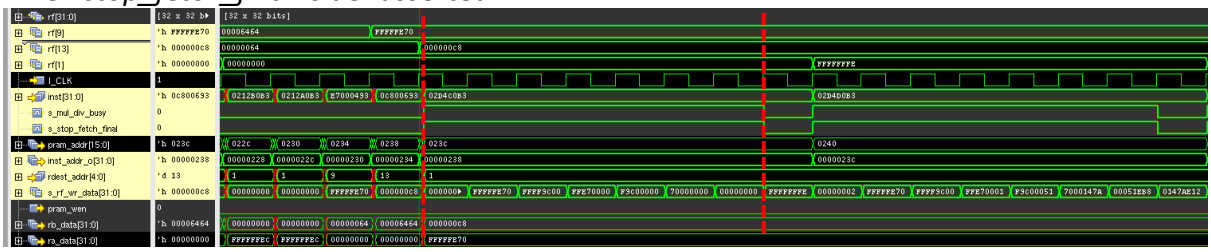

Figure 44 Division Operation

Figure 45 shows the divide by zero case where the division operation was made single cycle. As it's not a valid division the quotient will be all ones as per the task.

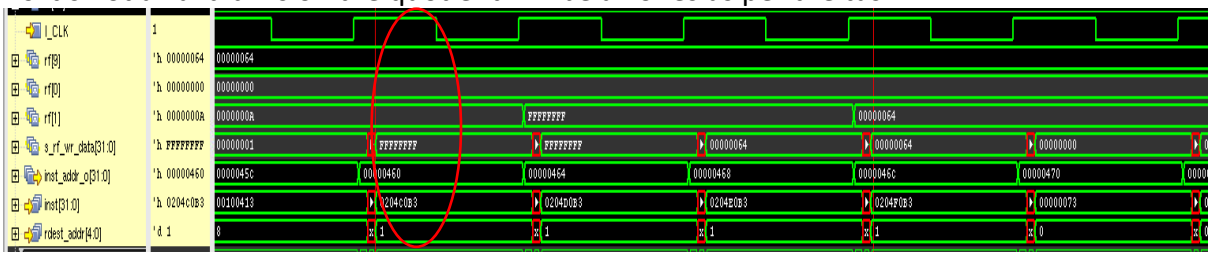

Figure 45 Divide by zero case

Jump or Branch Instruction:

The jump or branch instruction is executed as normal single cycle instruction. For conditional branches(if condition is true) and unconditional branches, the pc next will be updated with branch target address. Figure 46 shows an execution of loop:

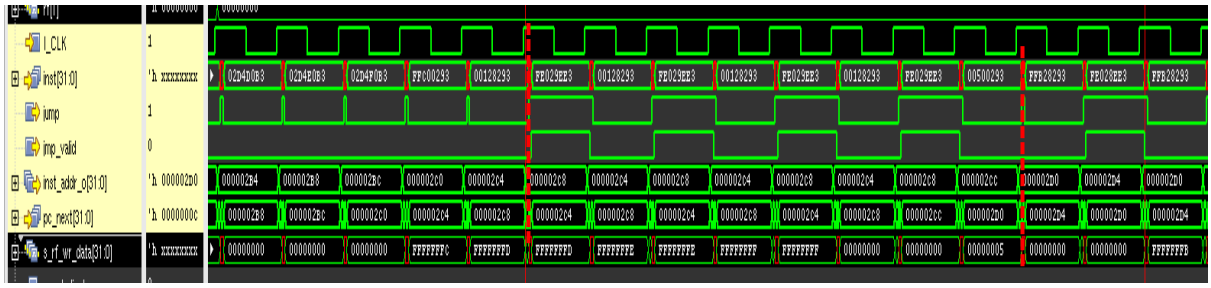

Figure 46 Jump Instruction

Store operation:

The Store instruction usually takes min 3 cycles and 2 cycles for external and internal memory access store respectively. When a store instruction is encountered, the fetch is stopped by asserting *stop_fetch_final*(through *ls_stop_fetch*) and it is released when *bus_ack* is asserted in case of external memory access via bus interface or *pram_st_done* in case of internal memory access. The execution continues normally after *stop_fetch_final* is released.

Figure 47 shows the scenario of external memory Store.

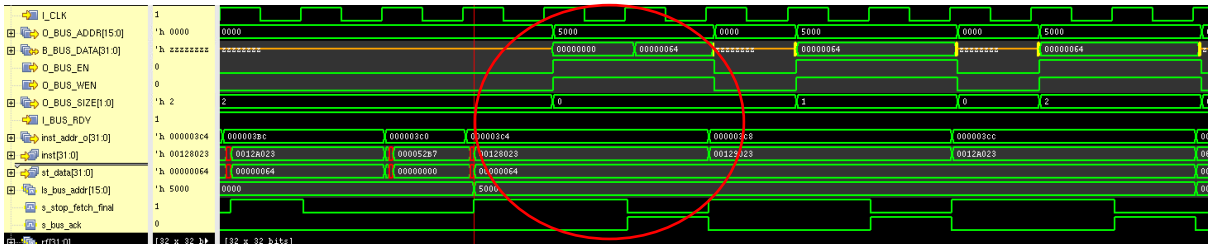

Figure 47 Store - external memory

Figure 48 shows the scenario of store to Internal memory

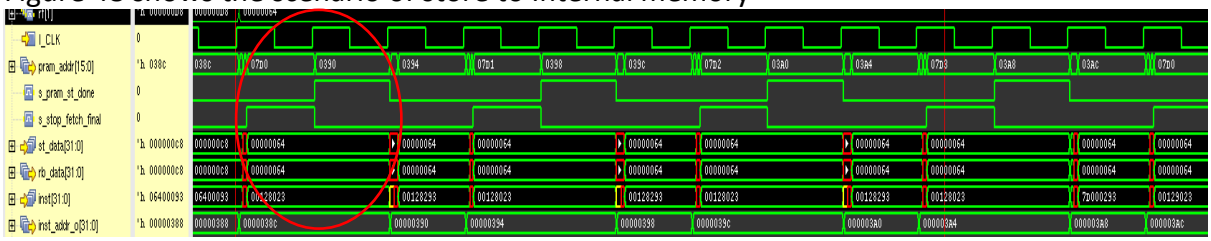

Figure 48 Store - internal memory

In both the above images, the instruction data and instruction address maintain same value until the Store operation is completed.

Wait For Interrupt:

The Wait for Interrupt instruction stalls the core until an interrupt needs to be serviced and once interrupt execution is done, the core continues normal execution. It can be clearly seen from Figure 49 that none of the signals change until an interrupt is available.

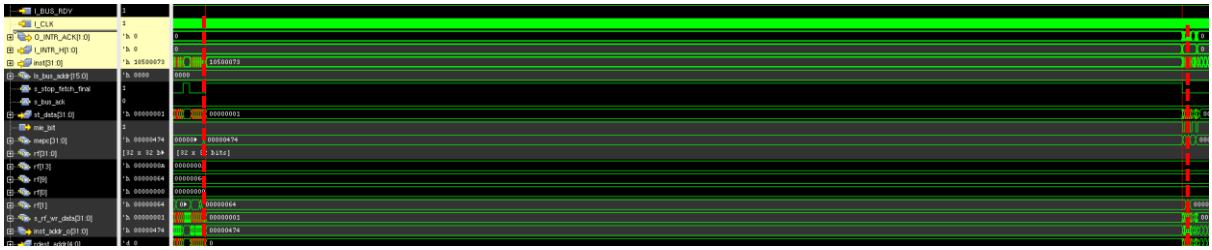

Figure 49 Wait for Interrupt

Interrupts:

The interrupt request is handled after the execution of current instruction once the interrupt handler address is loaded corresponding Interrupt acknowledgement is issued. When the interrupt is accepted, the MIE bit is de-asserted, and the next instruction address is stored into *mepc* register and will be used after returning from the interrupt handler. Figure 50 shows the interrupt handling scenario

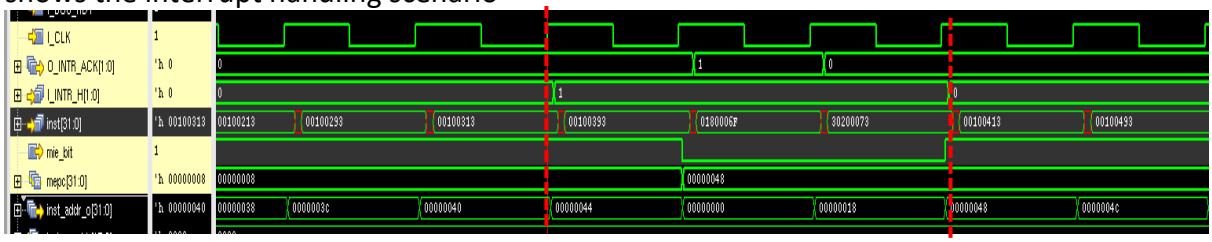

Figure 50 Interrupt handler

Bus Interface with random I_BUS_RDY :

The design was verified using random I_BUS_RDY signal. In the figure below it can be seen that the bus interface data changes only when bus ready is asserted and holds previous values during wait cycles.

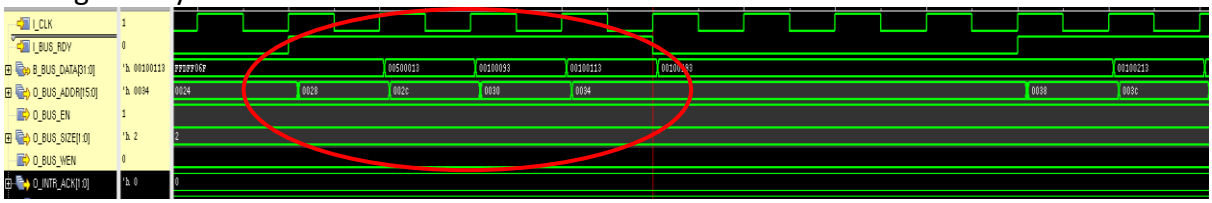

Figure 51 Random bus ready

6.2 Pipelined Processor Simulation Results

Initialisation of Internal memory : First step is the initialization of the internal memory from the bus. When the reset is enabled, memory initialization begins with the state machine entering *counter* state. Since the slave memory is always ready, bus fetch can be pipelined and, in every cycle, we get new instructions which is dumped into the internal PRAM. The address to the PRAM is one cycle delayed with respect to the bus address as the data is available on bus one cycle later as shown in figure 52.

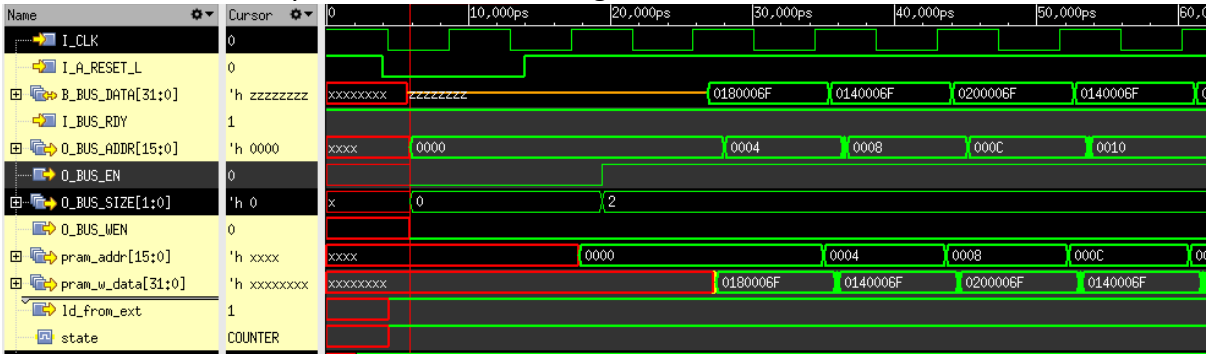

Figure 52 Initialization of Internal PRAM

Initialisation halt with Interrupt : Initialisation of the internal PRAM can also be stopped via the external interrupt. Figure 53 when the interrupt arrives, *ld_frm_ext* and *pram_wen* signal goes low indicating the write to the PRAM disabled and the pram address(*pram_addr*) begins from 0008 which is start of the code.

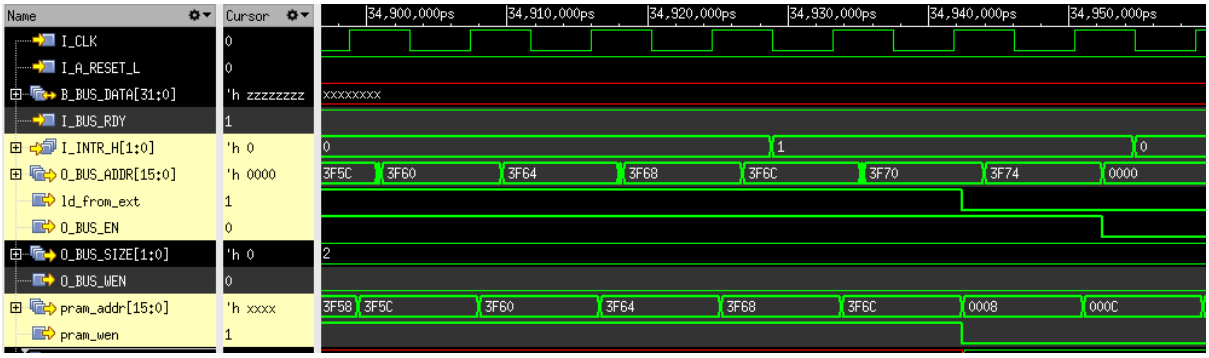

Figure 53 Initialization stop during external Interrupt

Single cycle Instruction :Figure 54 shows one of the examples of add instruction, the code used ***addi x1,x0,1*** adds 1 to the register file x1 as shown below. The red lines mark the 4-stage operation. The instruction enters the decode stage at the first redline and completes the register writeback at the second red line.

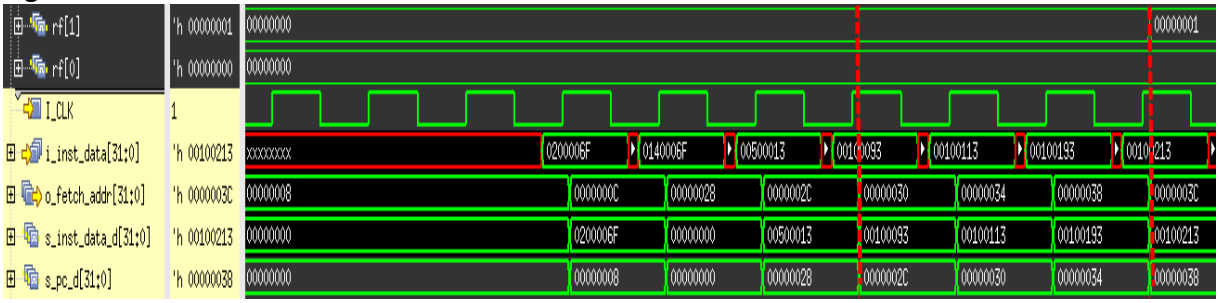

Figure 54 Single cycle instruction example

Load from External : Doing external load from the bus, a stop fetch(*stop_fetch*) signal is enabled as instruction enters the 4th stage. Halt signal will be sent to all the pipeline registers for the duration of 2 cycles as per the bus load latency. Once the data arrives from the bus, it is written into the register file. In the below figure, ***lb x1, 10(x31)*** was the instruction used. The data *0x64646464* arrives on the bus and only the lower byte is written as per the instruction into *rf[1]*. Red lines marked the beginning and ending of the above instruction.

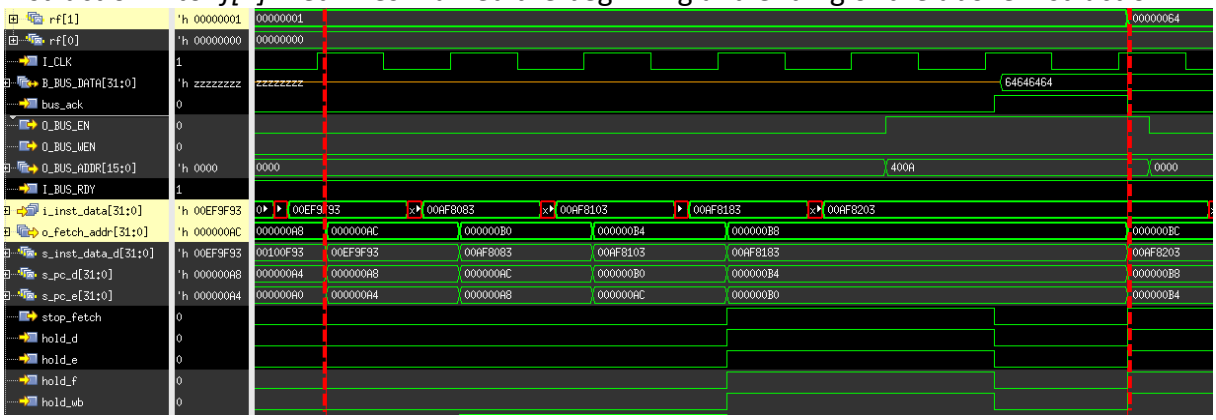

Figure 55 Load from External

Load from Internal PRAM: In this design, Internal PRAM unused locations can also be used for load and store operations. When the internal memory is used as load, it cannot be used for fetch so there is one cycle latency and we halt the pipeline registers for one cycle. All the hold signals in figure 56 Indicates the load from PRAM.

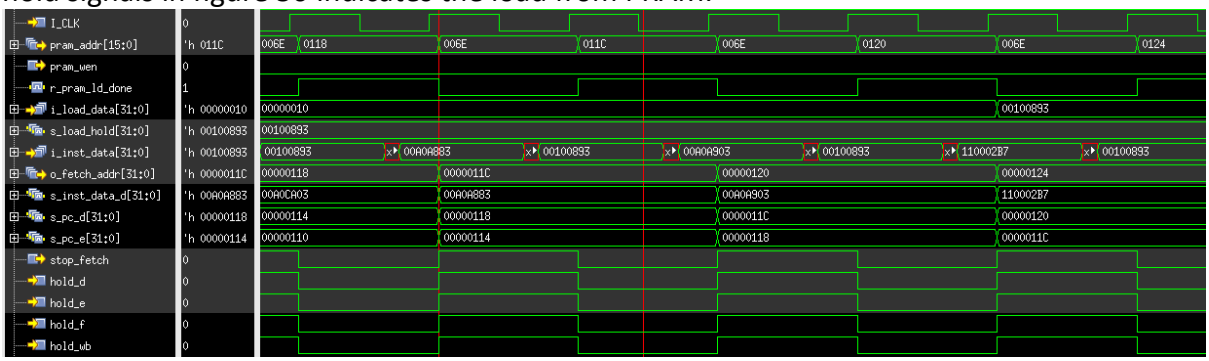

Figure 56 Load from Internal PRAM

Bitwise Instruction: Figure 57 shows one of the logical instructions executed. The instruction ***or x1, x5, x3*** performs logical or operation between the contents of register file rf[5] and rf[3] and stores it in rf[1]. At the first redline, the instruction enters decode stage and completes before the next redline. A value of 110 is written to the rf[1] which is rf[5] | rf[3].

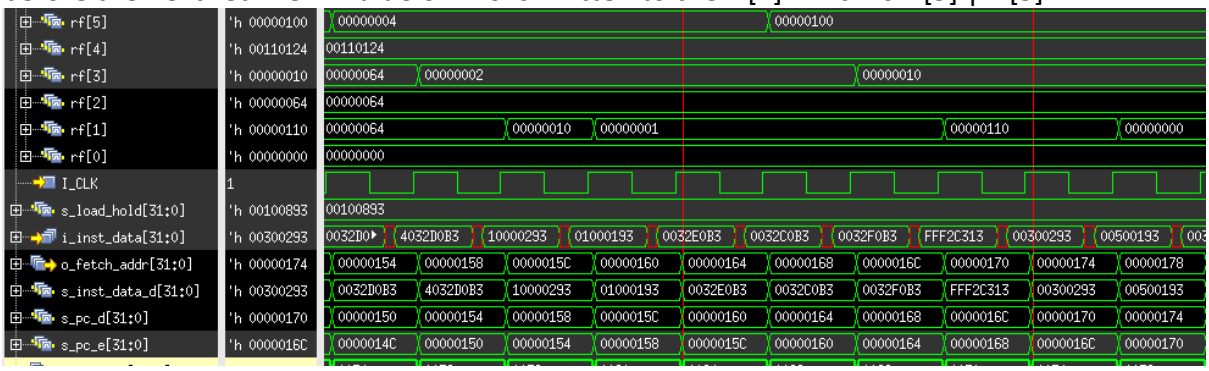

Figure 57 Bitwise Instruction

Multiplication and Division Instructions: The pipeline processor implemented takes 2 cycles(1 cycle for latching operands and then executing) for multiplication and 10 cycles for division(1 cycle for latching operands and then 9 cycles for execution). The latency was due to the complex circuitry of multiplication and division. Doing it in a single cycle will increase the combinational path between the registers and this results in lower frequency of the processor. Since these instructions are not more often compared to the other arithmetic and logical instructions so the number of cycles for its computation was increased. Since it was an in-order processor, halt signal was used for the pipeline registers as soon as the multiplication or division instructions enter the execution stage. Figure 58 shows a one cycle halt signal for the multiplication operation and Figure 59 shows a 10-cycle latency for Division operation.

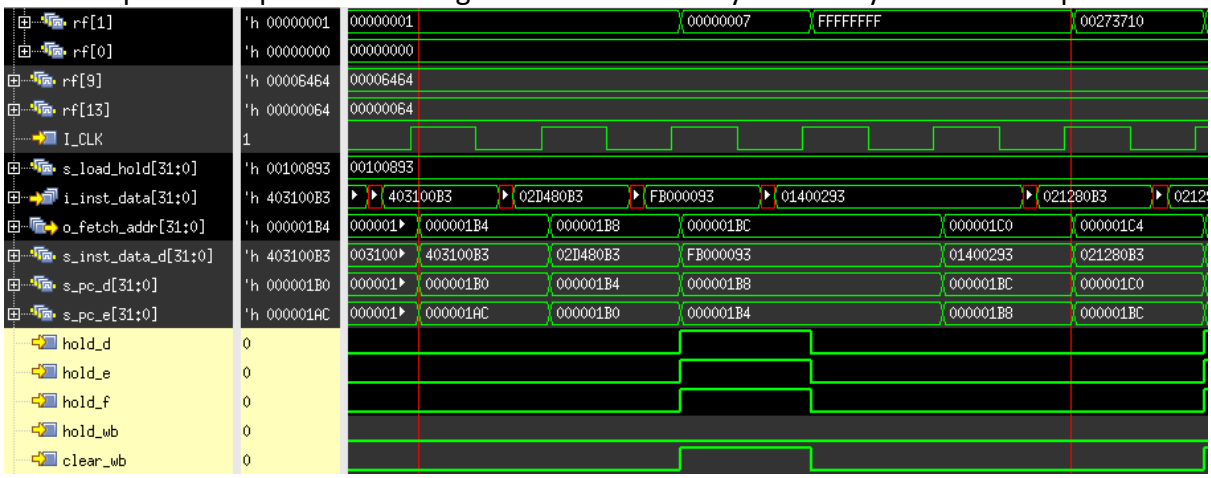

Figure 58 Multiplication operation

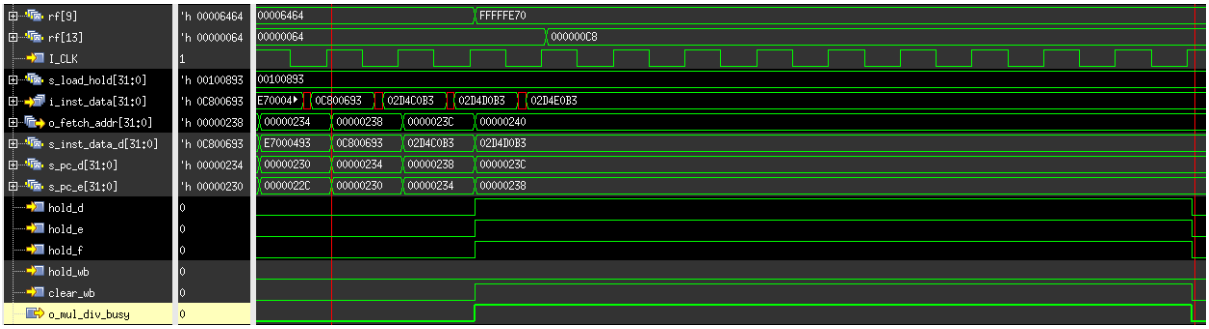

Figure 59 Division Operation

Internal PRAM Store: The internal PRAM can also be used for store and as one cycle latency since it cannot be used for fetch during store. When the store instruction is in the last stage, all the pipeline registers are halted. PRAM write enable(*pram_wen*) and data (*pram_w_data*) will be sent for the write into the PRAM.

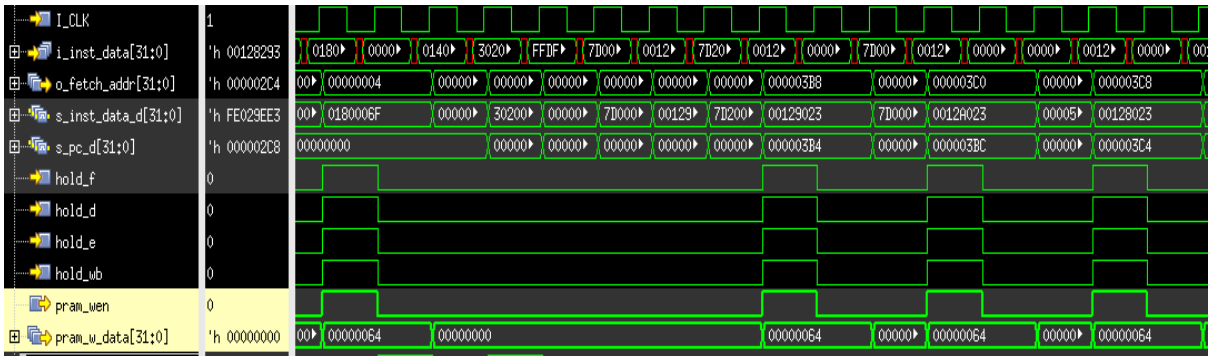

Figure 60 Internal PRAM store

Branch Instructions: During the branch instruction, the branch unit in the decode stage decides the branch. If the branch is enabled, the processor flushes the next instruction which was fetched(as shown by the *s_clear_d_en* signal). Branch uses one-bit branch predictor which helps to avoid flush every time the branch instruction is encountered. The below halt signals are due to the dependency of the registers in the decode stage on the execution stage results. The instruction executed for the figure 61 is:

0x2C0→li x5, -4
0x2c4→ loop_bne: addi x5, x5, 1
0x2c8→bne x5, x0, loop_bne

Loop run 3 times and jumps to next instruction on 4th iteration.

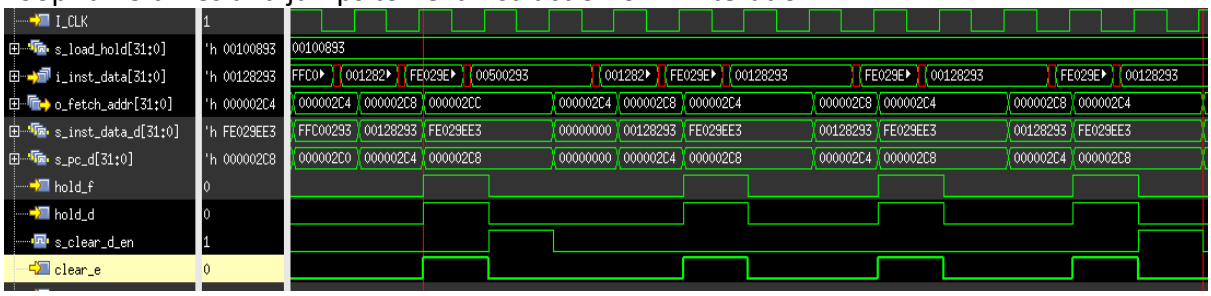

Figure 61 Branch Instructions

External SRAM Store: When an external store instruction enters the 4th stage, all the pipeline registers will be halted for 2 cycles. The first red line indicates start of the store instruction (pc value = 0x3C4) and once it reaches the 4th stage all the halt signals are enabled, and address(0x5000) is written on the external bus. On getting the bus ready signal, the data to be stored in the external SRAM is written on the data bus(00000064 here).

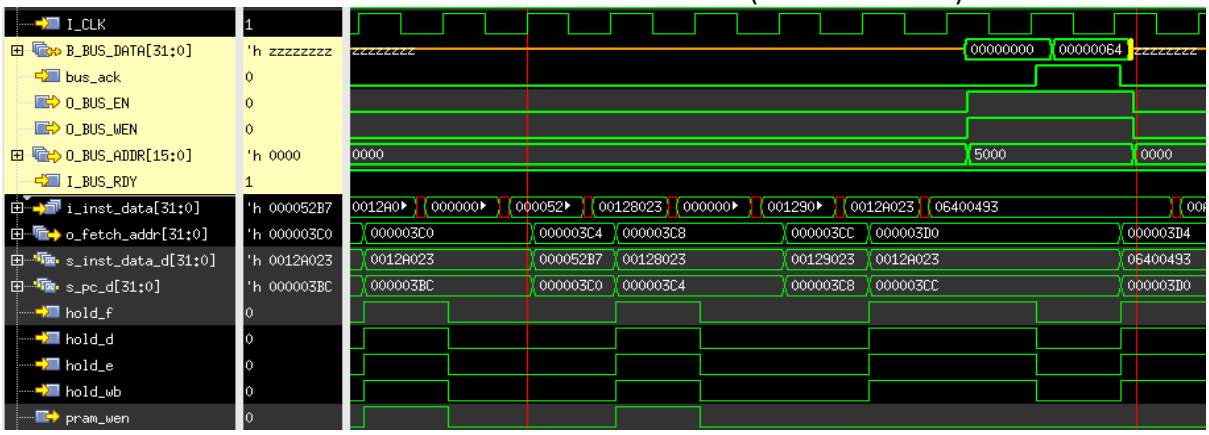

Figure 62 External SRAM store

Interrupt Handling: The interrupt control is in the second stage of the pipeline. Upon receiving an interrupt and accepting, a *clear_d* signal is enabled to remove the wrong instruction fetched and the control jumps to the interrupt handler.

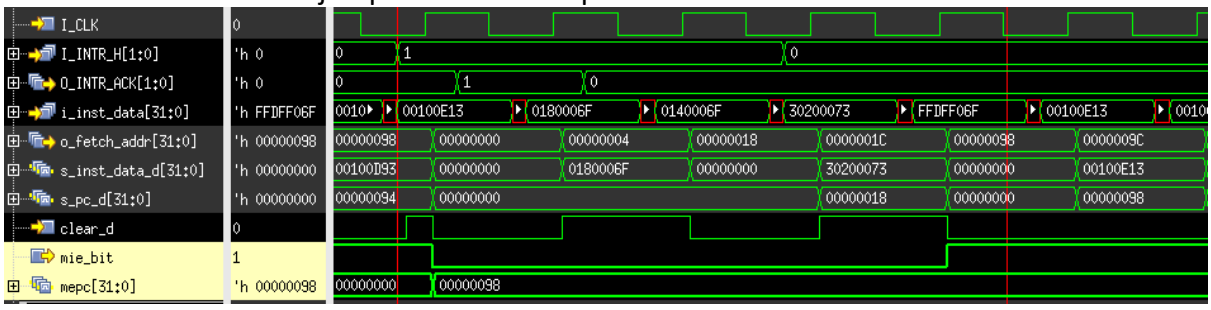

Figure 63 Interrupt Handling

Similarly, *mepc* register will be written with the return address(0x98) and will be used to update the PC after return from the interrupt handler. MIE bit is also made zero during interrupt handling to ensure that no new interrupts are accepted.

Division by zero: Upon division by zero exception, as per the task and RISC ISA, the quotient should be all ones and the instructions shouldn't take more than one cycle. this condition will be checked in the division hardware and will be completed single cycle as shown in the figure 64. The first redline indicates the division by zero operation in the execution stage and after 2 cycles the register file rf[1] is updated to all ones.

Figure 64 Division by zero

6.3 Verification using C code

For the verification of the design, different C codes were used along with Compiler environment which converts the C codes to assembly level and then further to machine language code. Custom scripts were written for this and one can simply execute the ***run.tcl*** script in the testcase directory which generates the required memory file(instructions.mem).

6.3.1 Pipelined Processor Results

Check whether prime or not: A C code was used to check whether the number is prime or not. If the number was prime, then zero will be written at the 0x8000(i.e 8192 below) memory location. The number 29 was used in the code which is prime and thus zero was written. It uses external SRAM positions for data operations and relevant signals are shown. For the code, please check the svn repository (**svn:group04**) and can be verified in the testcase folder.

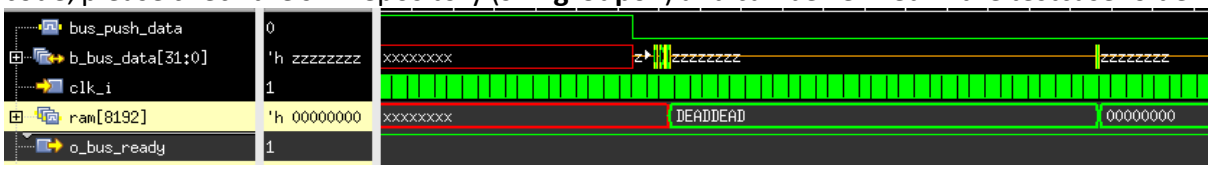

Figure 65 Prime or Not

Similarly, the factorial of number was checked. The number used was 10 and the result is 3628800 which is 0x375F00 in hexadecimal.

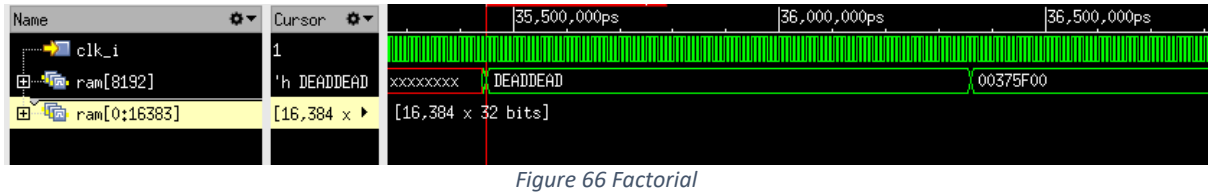

Figure 66 Factorial

Sort and Binary search: Both bubble sort and binary search algorithms were merged and used as a single C code. The code first sorts the elements in the ascending order and then uses the Binary search to search for a particular number after sorting. The location 0x8000(i.e. 8192 below) is written with position of the search number in the array.

Code flow:

Array used →int arr[] = { 2, 3, 4, 10, 40,1,5,9,-1131,-31241,-4141,14115,-1511,14151,-255,5536,53 } ;
Step→Perform Bubble sort
Search for element →x = -255;
Step → Binary Search
Result → 4th position

Figure 67 Sort and Search

6.3.2 Single Cycle Processor Results

Same algorithms as above were used for the verification.

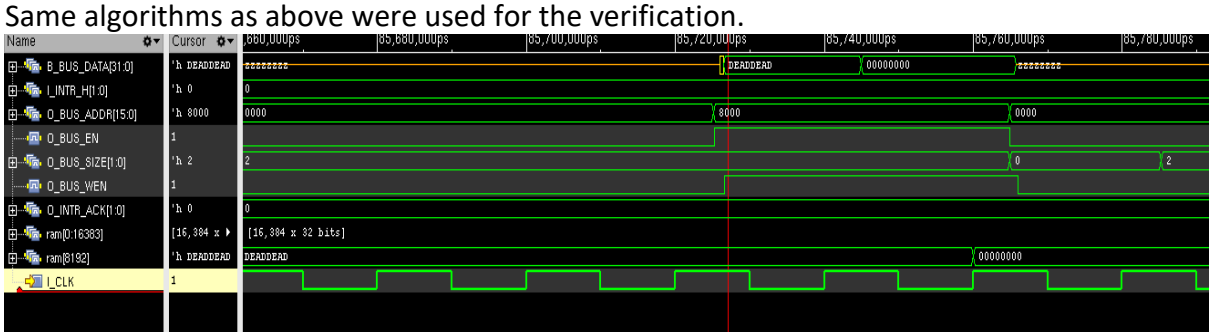

Figure 68 Prime number algorithm

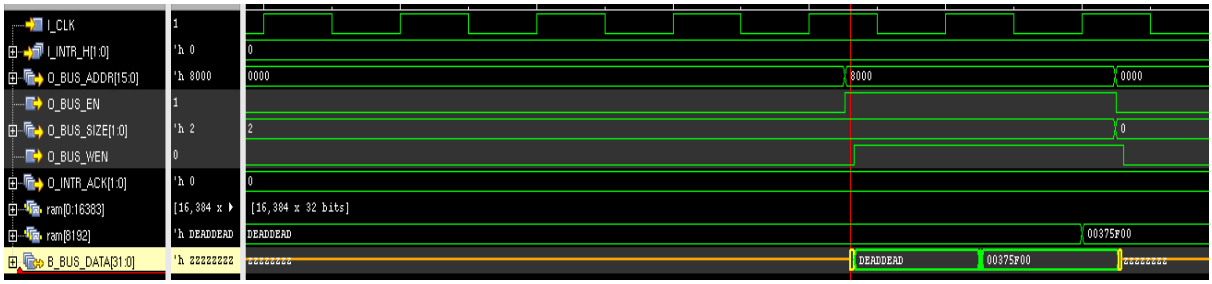

Figure 69 Factorial algorithm

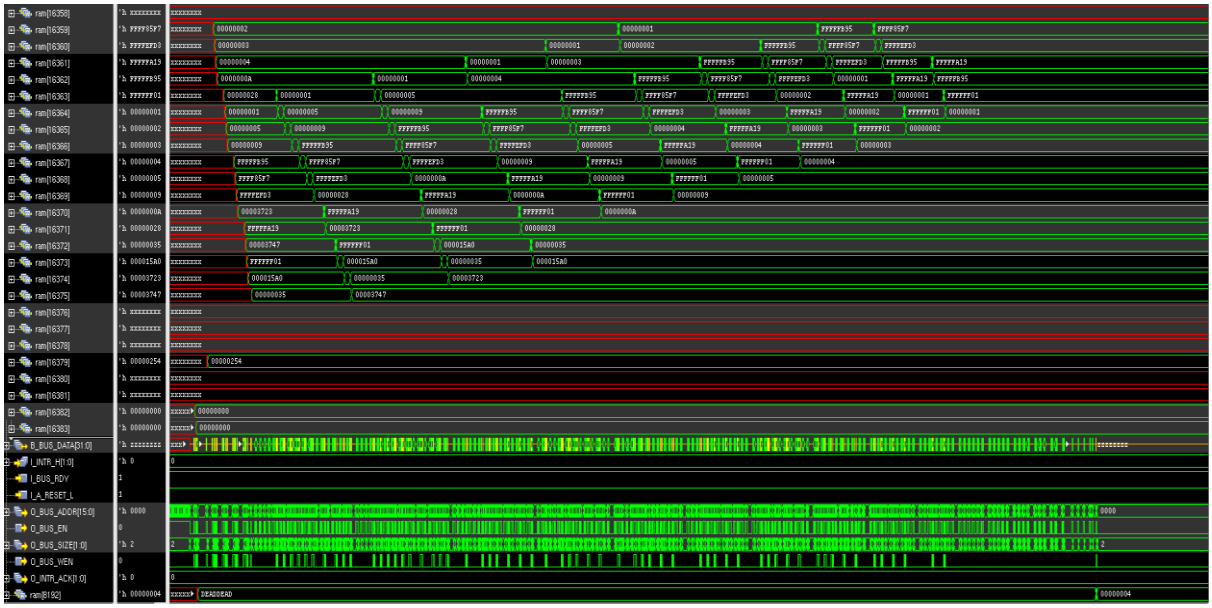

Figure 70 Sort and Search algorithm

Simulation and verification were done at all the stages that is after simulation, synthesis and placement and route to ensure that our design functionality is always satisfied. The above reports are picked from different stages of the design which ensures that the design works in all stages.

Chapter 7 SYNTHESIS AND PNR REPORTS

Single Cycle Processor Synthesis Reports: The synthesis step generates timing, Area, and power reports separately and are as shown in following figures.
The timing reports generates the combination path delay along 3 groups that is in->reg, reg-> 2 out ,reg->reg. Timing errors are represented by slack and any negative values of the slack will results in violations.

```
Startpoint: dobby_core_i/bus_if_i/r_wen_reg
            (rising edge-triggered flip-flop clocked by CLK50)
Endpoint: O_BUS_WEN (output port clocked by CLK50)
Path Group: REGOUT
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
bus_ctrl_DATA_WIDTH32_ADDR_WIDTH16
                   enG5K                 fsa0a_c_generic_core_ss1p62v125c
dobby_top          enG200K               fsa0a_c_generic_core_ss1p62v125c
pads               enG100K               fsa0a_c_generic_core_ss1p62v125c

Point                                                   Incr       Path
-----------------------------------------------------------------------
clock CLK50 (rise edge)                                 0.00       0.00
clock network delay (ideal)                             1.50       1.50
dobby_core_i/bus_if_i/r_wen_reg/CK (QDFFRBS)            0.00       1.50 r
dobby_core_i/bus_if_i/r_wen_reg/Q (QDFFRBS)             0.48       1.98 r
dobby_core_i/bus_if_i/U3/O (INV1S)                      0.44       2.42 f
dobby_core_i/bus_if_i/o_bus_we_BAR (bus_ctrl_DATA_WIDTH32_ADDR_WIDTH16)
                                                        0.00       2.42 f
dobby_core_i/o_bus_wen_BAR (dobby_core_4e4f4e45_4e4f4e45_4e4f4e45_4e4f4e45)
                                                        0.00       2.42 f
pads_i/bus_wen_BAR (pads)                               0.00       2.42 f
pads_i/U3/O (INV4)                                      1.41       3.83 r
pads_i/bus_wen_pad_i/O (YA2GSC)                         2.50       6.33 r
pads_i/O_BUS_WEN (pads)                                 0.00       6.33 r
O_BUS_WEN (out)                                         0.00       6.33 r
data arrival time                                                  6.33

clock CLK50 (rise edge)                                20.00      20.00
clock network delay (ideal)                             1.50      21.50
clock uncertainty                                      -0.20      21.30
output external delay                                 -10.00      11.30
data required time                                                11.30
-----------------------------------------------------------------------
data required time                                                11.30
data arrival time                                                 -6.33
-----------------------------------------------------------------------
slack (MET)                                                        4.97
```

Figure 71 REG -> Out path

```
                                                                  0.00      12.62 r
dobby_core_i/mem_controller_i/fetch_mem_ctrller_i/f_pram_addr[13] (fetch_mem_ctrller)
                                                                  0.00      15.99 f
dobby_core_i/mem_controller_i/U26/O (AO222T)                      0.63      16.62 f
dobby_core_i/mem_controller_i/pram_addr[13] (mem_controller)
                                                                  0.00      16.62 f
dobby_core_i/pram_i/mem_addr[13] (pram_4e4f4e45_4e4f4e45_4e4f4e45_4e4f4e45)
                                                                  0.00      16.62 f
dobby_core_i/pram_i/U61/O (INV2)                                  0.17      16.80 r
dobby_core_i/pram_i/U81/O (ND2P)                                  0.27      17.07 f
dobby_core_i/pram_i/block1/CSB (SY180_1024X8X4CM4)                0.00      17.07 f
data arrival time                                                           17.07

clock CLK50 (rise edge)                                          20.00      20.00
clock network delay (ideal)                                       1.50      21.50
clock uncertainty                                                -0.20      21.30
dobby_core_i/pram_i/block1/CK (SY180_1024X8X4CM4)                 0.00      21.30 r
library setup time                                               -0.54      20.76
data required time                                                          20.76
-----------------------------------------------------------------------------------
data required time                                                          20.76
data arrival time                                                          -17.07
-----------------------------------------------------------------------------------
slack (MET)                                                                 3.69
```

Figure 72 In ->REG path

```
                                                          0.20      19.67 r
dobby_core_i/alu_muldiv_i/new_mul_i/U55/O (XNR2S)         0.28      19.94 f
dobby_core_i/alu_muldiv_i/new_mul_i/prod[58] (new_mul_N34)
                                                          0.00      19.94 f
dobby_core_i/alu_muldiv_i/U339/O (ND2)                    0.11      20.06 r
dobby_core_i/alu_muldiv_i/U45/O (ND3S)                    0.20      20.26 f
dobby_core_i/alu_muldiv_i/alu_res_muldiv[26] (booth_md)
                                                          0.00      20.26 f
dobby_core_i/alu_res_mux/in_two[26] (mux4_1_N32_0)        0.00      20.26 f
dobby_core_i/alu_res_mux/U56/O (AO2222)                   0.43      20.69 f
dobby_core_i/alu_res_mux/outp[26] (mux4_1_N32_0)          0.00      20.69 f
dobby_core_i/rf_wr_mux/in_one[26] (mux2_1_N32_0)          0.00      20.69 f
dobby_core_i/rf_wr_mux/U770B (MXL2HS)                     0.18      20.87 r
dobby_core_i/rf_wr_mux/U17/O (INV2)                       0.11      20.97 f
dobby_core_i/rf_wr_mux/outp[26] (mux2_1_N32_0)            0.00      20.97 f
dobby_core_i/regfile_i/wdata[26] (regfile)                0.00      20.97 f
dobby_core_i/regfile_i/U819/O (BUF12CK)                   0.23      21.21 f
dobby_core_i/regfile_i/rf_reg_1__26_/D (QDFFRBS)          0.00      21.21 f
data arrival time                                                   21.21

clock CLK50 (rise edge)                                  20.00      20.00
clock network delay (ideal)                               1.50      21.50
clock uncertainty                                        -0.20      21.30
dobby_core_i/regfile_i/rf_reg_1__26_/CK (QDFFRBS)         0.00      21.30 r
library setup time                                       -0.09      21.21
data required time                                                  21.21
-----------------------------------------------------------------------
data required time                                                  21.21
data arrival time                                                  -21.21
-----------------------------------------------------------------------
slack (MET)                                                         0.00
```

Figure 73 REG -> REG path

Only the snippets of the timing report are shown here due to more elaborated timing paths. For detailed report, please refer the project in svn.

Area Report: The synthesis step also generates area report which shows separate areas used for combinational and sequential elements. Macro area shown in figure 74 are related to the PRAM blocks used. Total area of the cell is around ***1.61 mm^2*** .

```
Library(s) Used:

    fsa0a_c_generic_core_ss1p62v125c (File: /home2/vlsi00/prz-root/icpro_teamspace/lpd19/workspa
    fsa0a_c_t33_generic_io_ss1p62v125c (File: /home2/vlsi00/prz-root/icpro_teamspace/lpd19/works
    SY180_1024X8X4CM4_WC (File: /home2/vlsi00/prz-root/icpro_teamspace/lpd19/workspaces/group04/

Number of ports:                         63
Number of nets:                         154
Number of cells:                          2
Number of combinational cells:            0
Number of sequential cells:               0
Number of macros/black boxes:             0
Number of buf/inv:                        0
Number of references:                     2

Combinational area:             275510.492596
Buf/Inv area:                    29957.457436
Noncombinational area:           72814.091091
Macro/Black Box area:          1265041.762695
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:               1613366.346383
Total area:                 undefined
```

Figure 74 Area report

Power Report: The synthesis step generates more detailed report on the power. The main contribution to the total power are internal and switching power which represents the operating power and the power consumed due to the switching of the circuits respectively. Leakage power represents the power consumed by the circuit at subthreshold currents. The report also splits the power consumption among different components of the design. The total power used by the single cycle processor is 3.6282 mW

```
Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   2.8567 mW   (83%)
  Net Switching Power  = 570.5322 uW   (17%)
                         ---------
Total Dynamic Power    =   3.4273 mW  (100%)

Cell Leakage Power     = 214.1682 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.3870           0.1401        5.9631e+06            0.5330  (  14.69%)
memory             2.0751        1.5161e-02       1.7306e+08            2.2634  (  62.38%)
black_box          0.0000           0.0000           0.0000            0.0000  (   0.00%)
clock_network  9.2577e-02        5.3790e-02       2.0170e+05            0.1466  (   4.04%)
register           0.1764        5.4681e-03       6.4825e+06            0.1884  (   5.19%)
sequential         0.0000           0.0000           0.0000            0.0000  (   0.00%)
combinational      0.1256           0.3428       2.8460e+07            0.4969  (  13.69%)
--------------------------------------------------------------------------------------------------
Total              2.8567 mW        0.5573 mW    2.1417e+08 pW         3.6282 mW
```

Figure 75 Power report

After the synthesis, placement and routing steps are carried out to generate the physical layout of the chip. The timing reports are also generated here to ensure that after the physical routing of the wires there are no violations related to timing. It also gives information about the core density which represents the total area of the chip with the design. The core utilisation was around ***76.353%*** as shown in the figure 76.

```
#############################################################
#  Generated by:      Cadence Encounter 14.28-s033_1
#  OS:                Linux x86_64(Host ID eeesl1)
#  Generated on:      Sun Mar 14 20:12:12 2021
#  Design:            dobby_top
#  Command:           optDesign -postRoute -drv -outDir reports
#############################################################

------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.645  |  0.645  |  0.858  |  3.965  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|  2935   |  1489   |   40    |  2731   |
+--------------------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 76.353%
Total number of glitch violations: 0
------------------------------------------------------------
```

Figure 76 Final Post route Summary

Pipelined Processor Synthesis Reports: The above reports are generated for Pipelined processor too and are shown in the following figures.

```
                                                          0.14       9.29 r
  dobby_pipelined_top_i/dobby_ppl_core_i/alu_muldiv_i/new_div_i/U1568/O (BUF4)
                                                          0.23       9.52 r
  dobby_pipelined_top_i/dobby_ppl_core_i/alu_muldiv_i/new_div_i/U502/O (ND2S)
                                                          0.09       9.61 f
  dobby_pipelined_top_i/dobby_ppl_core_i/alu_muldiv_i/new_div_i/U1748/O (OAI12S)
                                                          0.14       9.75 r
  dobby_pipelined_top_i/dobby_ppl_core_i/alu_muldiv_i/new_div_i/remainder_reg_24_
                                                          0.00       9.75 r
  data arrival time                                                  9.75

  clock CLK50 (rise edge)                                 8.60       8.60
  clock network delay (ideal)                             1.50      10.10
  clock uncertainty                                      -0.20       9.90
  dobby_pipelined_top_i/dobby_ppl_core_i/alu_muldiv_i/new_div_i/remainder_reg_24_
                                                          0.00       9.90 r
  library setup time                                     -0.15       9.75
  data required time                                                 9.75
  ---------------------------------------------------------------------------
  data required time                                                 9.75
  data arrival time                                                 -9.75
  ---------------------------------------------------------------------------
  slack (MET)                                                        0.00
```

Figure 77 REG->REG path

```
                                                          0.08       9.34 f
  dobby_pipelined_top_i/dobby_ppl_core_i/pc_nxt_mux/U87/O (OAI12S)
                                                          0.15       9.49 r
  dobby_pipelined_top_i/dobby_ppl_core_i/pc_nxt_mux/outp[15] (mux2_1_N32_9)
                                                          0.00       9.49 r
  dobby_pipelined_top_i/dobby_ppl_core_i/branch_tar_mux/in_one[15] (mux2_1_N32_8)
                                                          0.00       9.49 r
  dobby_pipelined_top_i/dobby_ppl_core_i/branch_tar_mux/U7/O (MUX2)
                                                          0.27       9.76 r
  dobby_pipelined_top_i/dobby_ppl_core_i/branch_tar_mux/outp[15] (mux2_1_N32_8)
                                                          0.00       9.76 r
  dobby_pipelined_top_i/dobby_ppl_core_i/fetcher_i/pc_next[15] (fetcher)
                                                          0.00       9.76 r
  dobby_pipelined_top_i/dobby_ppl_core_i/fetcher_i/inst_addr_reg_15_/D (QDFFRBS)
                                                          0.00       9.76 r
  data arrival time                                                  9.76

  clock CLK50 (rise edge)                                 8.60       8.60
  clock network delay (ideal)                             1.50      10.10
  clock uncertainty                                      -0.20       9.90
  dobby_pipelined_top_i/dobby_ppl_core_i/fetcher_i/inst_addr_reg_15_/CK (QDFFRBS)
                                                          0.00       9.90 r
  library setup time                                     -0.14       9.76
  data required time                                                 9.76
  ---------------------------------------------------------------------------
  data required time                                                 9.76
  data arrival time                                                 -9.76
  ---------------------------------------------------------------------------
  slack (MET)                                                        0.00
```

Figure 78 In->REG path

```
  Startpoint: dobby_pipelined_top_i/bus_if_i/r_wen_reg
              (rising edge-triggered flip-flop clocked by CLK50)
  Endpoint: O_BUS_WEN (output port clocked by CLK50)
  Path Group: REGOUT
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  bus_ctrl_DATA_WIDTH32_ADDR_WIDTH16
                     enG5K                 fsa0a_c_generic_core_ss1p62v125c
  dobby_pipefinal_top
                     enG200K               fsa0a_c_generic_core_ss1p62v125c
  pads               enG100K               fsa0a_c_generic_core_ss1p62v125c

  Point                                                              Incr       Path
  ---------------------------------------------------------------------------------
  clock CLK50 (rise edge)                                            0.00       0.00
  clock network delay (ideal)                                        1.50       1.50
  dobby_pipelined_top_i/bus_if_i/r_wen_reg/CK (QDFFRBS)
                                                                     0.00       1.50 r
  dobby_pipelined_top_i/bus_if_i/r_wen_reg/Q (QDFFRBS)
                                                                     0.42       1.92 r
  dobby_pipelined_top_i/bus_if_i/U4/O (BUF1CK)                       0.93       2.85 r
  dobby_pipelined_top_i/bus_if_i/o_bus_we (bus_ctrl_DATA_WIDTH32_ADDR_WIDTH16)
                                                                     0.00       2.85 r
  dobby_pipelined_top_i/bus_wen (dobby_pipelined_top_4e4f4e45_4e4f4e45_4e4f4e45_4e
                                                                     0.00       2.85 r
  pads_i/bus_wen (pads)                                              0.00       2.85 r
  pads_i/U3/O (BUF3)                                                 1.42       4.27 r
  pads_i/bus_wen_pad_i/O (YA2GSC)                                    2.50       6.77 r
  pads_i/O_BUS_WEN (pads)                                            0.00       6.77 r
  O_BUS_WEN (out)                                                    0.00       6.77 r
  data arrival time                                                             6.77

  clock CLK50 (rise edge)                                            8.60       8.60
  clock network delay (ideal)                                        1.50      10.10
  clock uncertainty                                                 -0.20       9.90
  output external delay                                             -1.00       8.90
  data required time                                                            8.90
  ---------------------------------------------------------------------------------
  data required time                                                            8.90
  data arrival time                                                            -6.77
  ---------------------------------------------------------------------------------
  slack (MET)                                                                   2.13
```

Figure 79 REG->Out path

```
Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   8.3242 mW   (81%)
  Net Switching Power  =   1.9532 mW   (19%)
                         ---------
Total Dynamic Power    =  10.2774 mW  (100%)

Cell Leakage Power     = 219.8798 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.8024           0.2868        5.9629e+06            1.0952  (  10.46%)
memory             4.8259        3.5057e-02        1.7306e+08            5.0340  (  48.10%)
black_box          0.0000           0.0000            0.0000            0.0000  (   0.00%)
clock_network      0.4406           0.7641        2.8600e+05            1.2050  (  11.51%)
register           1.9455        7.8915e-02        9.5288e+06            2.0339  (  19.43%)
sequential         0.0000           0.0000            0.0000            0.0000  (   0.00%)
combinational      0.3097           0.7575        3.1041e+07            1.0983  (  10.49%)
--------------------------------------------------------------------------------------------------
Total              8.3242 mW        1.9224 mW     2.1988e+08 pW        10.4664 mW
```

Figure 80 Power Report

```
Design : dobby_pipefinal_top
Version: J-2014.09-SP5-3
Date   : Sun Mar 14 21:20:25 2021
****************************************

Library(s) Used:

     fsa0a_c_generic_core_ss1p62v125c (File: /home2/vlsi00/prz-root/icpro_teamsp
     SY180_1024X8X4CM4_WC (File: /home2/vlsi00/prz-root/icpro_teamspace/lpd19/wo
     fsa0a_c_t33_generic_io_ss1p62v125c (File: /home2/vlsi00/prz-root/icpro_tean

Number of ports:                         63
Number of nets:                         154
Number of cells:                          2
Number of combinational cells:            0
Number of sequential cells:               0
Number of macros/black boxes:             0
Number of buf/inv:                        0
Number of references:                     2

Combinational area:             294918.625951
Buf/Inv area:                    33932.202913
Noncombinational area:           96337.585369
Macro/Black Box area:          1265041.762695
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:               1656297.974016
Total area:                 undefined

Information: This design contains black box (unknown) components. (RPT-8)
1
```

Figure 81 Area Report

```
#  Design:              dobby_pipefinal_top
#  Command:             optDesign -postRoute -drv -outDir reports
#####################################################################

------------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.071  |  0.071  |  0.897  |  0.495  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|  3450   |  1981   |   56    |  3252   |
+--------------------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +---------------+---------------+------------------+
|                | Nr nets(terms)| Worst Vio     | Nr nets(terms)   |
+----------------+---------------+---------------+------------------+
|   max_cap      |     0 (0)     |    0.000      |     0 (0)        |
|   max_tran     |     0 (0)     |    0.000      |     0 (0)        |
|   max_fanout   |     0 (0)     |       0       |     0 (0)        |
|   max_length   |     0 (0)     |       0       |     0 (0)        |
+----------------+---------------+---------------+------------------+

Density: 76.698%
Total number of glitch violations: 0
------------------------------------------------------------------
```

Figure 82 Final PNR Report

Pipelined Processor

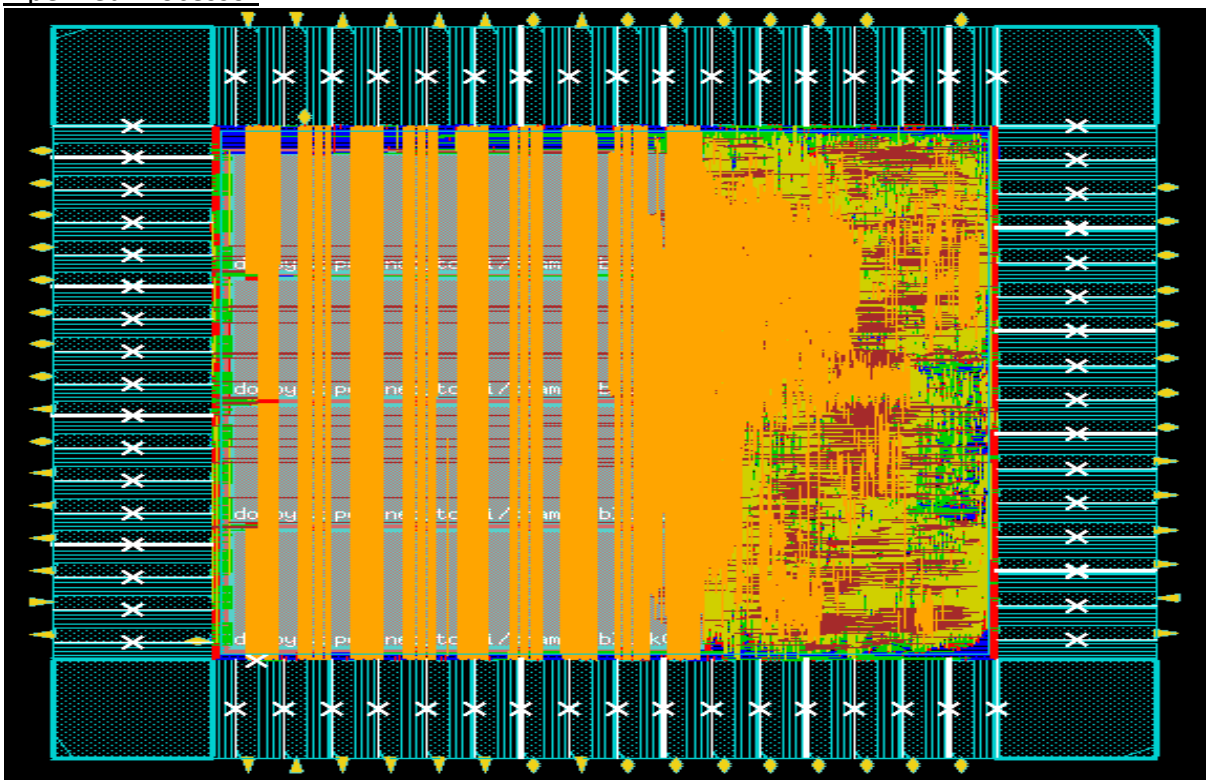

Figure 83 Pipelined Processor

CONCLUSION

Thus, the RISC-V based processor was designed using 2 different architectures. Single cycle processor achieved speed was around 50 MHz and pipeline processor runs at a frequency of 116 MHz . Halt cycles in single processor was mainly due to the load and store operations and the division operation. Halt cycles in pipeline processor was mainly due to the multiplication, dependencies, load and store and the division operations. These architectures were implemented keeping the design goal as high speed along with high throughput.

FUTURE SCOPE

To achieve higher frequencies with no importance of high throughput , state machine-based architectures can be used. The pipeline architecture implemented was based on in order processor which executes the instructions in order. If the compiler had architecture information, then it's possible to create an out of order processor and completion of instructions out of order. Halt signals used for many operations in pipeline-based design can be eliminated if we have multi-port write for the register file. More sophisticated branch predictors can also be implemented with an additional requirements of branch target buffers which holds the target instructions. Pipeline stages can also be increased to improve the speed at an additional cost of complex hazard detection unit.

Chapter 8 REFERENCES

[1] The 8051 Microcontroller and Embedded Systems Using Assembly and C, Second Edition, Muhammad Ali Mazidi, Janice Gillispie Mazidi, Rolin D. McKinlay.

[2] Computer Architecture: A Quantitative Approach, Fifth Edition, John L. Hennessy and David A. Patterson.

[3] An Overview of RISC vs CISC, Alan D. George, Department of Electrical Engineering FAMUFSU College of Engineering Tallahassee, FL 32304.

[4] The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2, University of California, Berkeley

[5] Digital Design and Computer Architecture, second Edition, David Money Harris and Sarah L. Harris

[6] https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

[7] RADIX-4 MODIFIED BOOTH'S MULTIPLIER USING VERILOG RTL, Aamir Bin Hamid, Nadeem Tariq Beigh, Ritu Singh, Sharda University, Department of Electronics & Communication , Greater Noida, UP, India

[8] https://www.geeksforgeeks.org/implementation-of-restoring-division-algorithm-for-unsigned-integer/

[9] VLSI Processor Design Lectures, Christian Georg Mayr , Chair of HPSN, TU Dresden

[10] https://en.wikipedia.org/wiki/Branch_predictor

[11] RISC-V ASSEMBLY LANGUAGE, Programmer Manual Part I, developed by: SHAKTI Development Team, IIT Madras.

[12] The RISC-V Instruction Set Manual Volume II: Privileged Architecture, University of California, Berkeley