ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMENTATION OF A 32-BIT RISC-V CORE

SENIOR DESIGN PROJECT

Yavuz Selim TOZLU
Yasin YILMAZ

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Berna Örs Yalçın
Uygundur 16.06.2021

JUNE 2021

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMENTATION OF A 32-BIT RISC-V CORE

SENIOR DESIGN PROJECT

Yavuz Selim TOZLU
040160232

Yasin YILMAZ
040170001

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JUNE 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

32-BİTLİK BİR RISC-V İŞLEMCİSİ TASARIMI VE GERÇEKLENMESİ

LİSANS BİTİRME TASARIM PROJESİ

Yavuz Selim TOZLU
040160232

Yasin YILMAZ
040170001

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

Haziran, 2021

We are submitting the Senior Design Project Report entitled as “DESIGN AND IMPLEMENTATION OF A 32-BIT RISC-V CORE”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity

Yavuz Selim TOZLU ....
040160232

Yasin Yılmaz ....
040170001

FOREWORD

We would like to express our gratitude to our advisor Prof. Dr. Sıddıka Berna Örs Yalçın, who dedicated her time for us and provided all the help she could. We also acknowledge the support our families gave us; without which we would not succeed.

We are also thankful to the members of the VLSI Lab at our faculty, who were there to provide and aid us with the CAD tools that we used in this project.

June 2021

Yavuz Selim TOZLU
Yasin YILMAZ

TABLE OF CONTENTS

Page

FOREWORD .... **iv**
TABLE OF CONTENTS .... **vi**
ABBREVIATIONS .... **viii**
LIST OF TABLES .... **ix**
LIST OF FIGURES .... **x**
SUMMARY .... **xi**
ÖZET .... **xii**
1. INTRODUCTION **14**
1.1 General Information and Concepts 15
1.1.1 Binary numbers and two’s complement format 15
1.1.2 A processor 16
1.1.3 Instruction set architecture 16
1.1.4 RISC-V instruction set architecture 17
2. DESIGN AND IMPLEMENTATION OF THE INTEGER PIPELINE **20**
2.1 Microarchitecture of the core 20
2.1.1 Designing the datapath for integer instructions 21
2.1.2 Pipelining the datapath 23
2.2 Implementation of the design 27
2.2.1 Verilog description of the design 27
2.2.2 Synthesis of the RTL description 28
3. DESIGN AND IMPLEMENTATION OF THE PRIVILEGED ARCHITECTURE **29**
3.1 About the Privileged Architecture 29
3.2 Design of the Privileged Architecture 30
3.2.1 Requirements 30
3.2.2 Machine-Level Registers 30
3.2.3 Interrupts and Exceptions 31
3.2.4 CSR Instructions 34
3.3 Implementation of the design 37
4. DESIGN AND IMPLEMENTATION OF THE “M” STANDARD EXTENSION **37**
4.1 Multiplier 37
4.1.1 Multiplier algorithm 38
4.1.2 Design of the multiplier 38
4.2 Divider 40
4.2.1 Divider algorithm 40
4.2.2 Design of the divider 41
4.3 The MULDIV Circuit 44
4.3.1 Signed divider circuit 44
4.3.2 Design of the MULDIV circuit 44
4.3.3 Testing of the MULDIV module 51
4.3.4 Integration to the core 51

5. TESTING THE DESIGN 54
5.1 Setting up the environment for testing 54
5.1.1 Installing Ubuntu Linux operating system 55
5.1.2 Installing the toolchain 55
5.1.3 Installing Verilator 56
5.1.4 Installing GTKWave 56
5.2 Compiling a RISC-V program 57
5.3 Simulating a RISC-V program using the core 60
5.3.1 Developing an SoC for simulation 60
5.3.2 Simulating with Cadence Xcelium 60
5.3.3 Simulating with Verilator 61
6. BENCHMARKS AND POST-SYNTHESIS ANALYSIS 62
6.1 Benchmarks 62
6.2 The critical path 64
6.3 Area 66
7. REALISTIC CONSTRAINTS AND CONCLUSIONS 66
7.1 Practical Application of this Project 66
7.2 Realistic Constraints 66
7.2.1 Social, environmental and economic impact 66
7.2.2 Cost analysis 67
7.2.3 Standards 67
7.2.4 Health and safety concerns 67
7.3 Future Work and Recommendations 67
REFERENCES 68
APPENDICES 69
APPENDIX A 70
APPENDIX B 70
CURRICULUM VITAE 71

ABBREVIATIONS

ISA : Instruction Set Architecture
PC : Program Counter
ALU : Arithmetic Logic Unit
ADC : Analog-Digital Converter
CPU : Central Processing Unit
RISC : Reduced Instruction Set Computer
HDL : Hardware Description Language
CAD : Computer Aided Design
RTL : Register-Transfer Level
ASIC : Application-Specific Integrated Circuit
SDF : Standard Delay Format
CSR : Control and Status Register
ASM : Algorithmic State Machine
FSM : Finite-State Machine
SoC : System-on-Chip
MSB : Most-significant bit
LSB : Least-significant bit

LIST OF TABLES

Page

Table 6.1 : Benchmark results 64
Table 6.2 : Area consumption 66

LIST OF FIGURES

Page

Figure 2.1 : List of RV32I instructions...... 20
Figure 2.2 : Datapath for the ADD instruction 21
Figure 2.3 : Datapath for the Load Word instruction 22
Figure 2.4 : Complete datapath for RV32I instructions...... 22
Figure 2.5 : Assembly code that causes a pipeline hazard...... 24
Figure 2.6 : Assembly code that causes a pipeline hazard...... 25
Figure 2.7 : Complete pipeline diagram for RV32I instructions 26
Figure 2.8 : Module hierarchy 28
Figure 3.1 : ASM chart of the finite-state machine...... 33
Figure 3.2 : Encoding of the CSR instructions 34
Figure 3.3 : Updated pipeline diagram with the privileged architecture...... 36
Figure 4.1 : Instruction opcodes for the "M" standard extension 37
Figure 4.2 : Circuit diagram of the multiplier...... 39
Figure 4.3 : Division algorithm 41
Figure 4.4 : Circuit diagram of the divider...... 42
Figure 4.5 : State diagram of the divider 43
Figure 4.6 : Circuit diagram of the MULDIV 45
Figure 4.7 : Bit arrangement of the "AB_status"...... 46
Figure 4.8 : State diagram of the MULDIV...... 48
Figure 4.9 : Fast result conditions and output arrangements 50
Figure 4.10 : Generation of the "muldiv_stall" signal 52
Figure 4.11 : Updated pipeline diagram with MULDIV block 53
Figure 5.1 : C code of the ROM generator program 59
Figure 5.2 : C++ wrapper file for bubble sort 61
Figure 6.1 : 64-bit integer multiplication code 62
Figure 6.2 : 64-bit integer division code 63
Figure 6.3 : 64-bit floating-point multiplication code 63
Figure 6.4 : 64-bit floating-point division code...... 63
Figure 6.5 : Critical path without multiply/divide unit 65
Figure 6.6 : Critical path after adding multiply/divide unit 65
Figure 6.7 : Critical path after changing division to 16 cycles 65
Figure A.1 : Synthesis script for Genus...... 70

DESIGN AND IMPLEMENTATION OF A 32-BIT RISC-V CORE

SUMMARY

The concepts of "Open-source software" and "Open-source hardware" are thriving in the modern society. Hundreds of companies and groups are working hard to provide the humanity with free and open-source software and hardware designs. "Open-source" in this context refers to the fact that the design files are freely available to the public. A significant part of this effort is directed towards the provision of open-source microprocessor designs. In light of this, researchers at University of California, Berkeley developed a license-free Instruction Set Architecture called "RISC-V", which essentially defines the vocabulary of the hardware/software interface. This ISA is warmly welcomed by researchers and companies; thus, many different RISC-V processors, be it open-source or proprietary, emerged and many have seen tape-outs.

Some crucial aspect of an open-source hardware are its extendibility, flexibility, and comprehensibility. Most designs are often extendible and well thought-out, but they are rarely comprehensible, which negatively impacts their extendibility. The common problem is that they either lack documentation, or have hastily written ones. This causes the users to spend so much time decoding and understanding the design, such is the case with most of the open-source RISC-V cores. In this project, we wanted to tackle this problem. We developed a 32-bit RISC-V core that is not only open-source, but also well documented. We disclosed the design diagrams and decisions that we made during the design process, so that the users can get started quickly, and experiment with the core as they wish.

We first started off with learning about computer architecture from textbooks. We learned how we could build a processor from scratch. Then, using that knowledge, we started designing the core. We spent weeks drawing design diagrams and iteratively fixing bugs. Once we were confident in our design, we started implementing it. The implementation was done in ASIC domain. We used industry-standard simulation and synthesis tools. We wrote the RTL description of the design, simulated and synthesized it. Last but not least, we wrote several test programs in C programming language to verify functionality of the processor. We successfully ran these C programs in a simulation environment, and verified the functionality of our design.

In the end, we managed to design and develop a synthesizable 32-bit RISC-V processor that is capable of executing integer instructions, multiply/divide instructions, and handling interrupts and exceptions. We also documented every step as much as we could. Of course, we have published our design on Github, where we also present example system-on-chip designs, useful peripherals, testing infrastructure, documentation, and some useful software libraries.

We named this RISC-V core "Hornet".

32-BİTLİK RISC-V İŞLEMCİSİ TASARIMI VE GERÇEKLENMESİ

ÖZET

"Açık-kaynak yazılım" ve "Açık-kaynak donanım" konseptleri günümüzde oldukça yaygınlaşmış durumda. Yüzlerce şirket ve araştırma grubu, insanlığa açık-kaynak yazılım ve donanım tasarımları kazandırmak için çalışıyor. Bu bağlamda, bir tasarımın "Açık-kaynak" olması, tasarım dosyalarının koşulsuz şartsız topluma açık olması anlamına geliyor. Bu çalışmaların ciddi bir kısmı açık-kaynak işlemci tasarlamaya yönelik. Bu doğrultuda California, Berkeley Üniversitesi'ndeki araştırmacılar, adı "RISC-V" olan, lisans koşulu olmayan, donanım ile yazılım arasındaki arayüzü sağlayan bir Komut Kümesi Mimarisi geliştirdiler. Bu Komut Kümesi Mimarisi araştırmacılar ve şirketler tarafından olumlu karşılandı, böylece birçok çeşit RISC-V işlemcisi, hem açık-kaynak hem de tescilli olmak üzere, ortaya çıktı, ve bunların birçoğu üretime gitti.

Açık-kaynak donanımın kritik özelliklerinden bazıları geliştirilebilir olması, esnek olması, ve anlaşılır olmasıdır. Çoğu tasarımlar genellikle geliştirilebilir ve iyi düşünülmüş oluyorlar, ancak nadiren anlaşılır oluyorlar, ki bu da geliştirilebilirliklerini olumsuz etkiliyor. Yaygın görülen problemlerden birisi, tasarımların ya dökümanlarının eksik olması, ya da üstünkörü yazılmış olmasıdır. Bu, kullanıcıların tasarımı anlamak için çok fazla zaman harcamalarına sebep oluyor. Açık-kaynak RISC-V işlemcilerinin birçoğunda durumun böyle olduğu görülebilir. Bu projede, bahsi geçen problemi çözmeyi hedefledik. 32-bitlik bir RISC-V çekirdeği geliştirdik, ki bu çekirdek sadece açık-kaynak değil, aynı zamanda kapsamlı bir şekilde dökümanlanmış oldu. Tasarım sürecinde çizdiğimiz diagramları ve aldığımız kararları da paylaşıma sunduk. Böylece, kullanıcılar çekirdek ile çalışmaya kolayca başlayabilirler, ve çekirdeği istedikleri gibi kurcalayabilirler.

Çekirdeği tasarlama işine koyulmadan önce, bilgisayar mimarisi alanında yazılan kitaplardan bu işi nasıl yapacağımızı öğrendik. Ders kitaplarından faydanalarak, baştan aşağıya bir işlemcinin nasıl tasarlandığını öğrendik. Öğrendiğimiz bu bilgiler ile çekirdeği tasarlamaya başladık. Haftalarca diagramlar çizdik; iteratif bir biçimde hataları düzeltip, tekrar çizim yaptık. Tasarımımızın hazır olduğunu düşündüğümüzde gerçeklemeye başladık. Gerçekleme işlemini ASIC dömeninde yaptık. Endüstride yaygın olarak kullanılan benzetim ve sentez araçlarını kullandık. Tasarımın RTL kodunu yazdık, simülasyonlarını ve sentez işlemini yaptık. Son olarak da, çekirdeğin fonksiyonelliğini test etmek maksadıyla, C programları yazdık. Bu programları simülasyon ortamında başarılı bir şekilde çalıştırıp, çekirdeğin fonksiyonel olduğunu onayladık.

Sonuç olarak; 32-bitlik, sentezlenebilir bir RISC-V çekirdeğini tasarlayıp gerçekledik. Bu çekirdek; integer ve çarpma/bölme komutlarını çalıştırma kabiliyetine, ayrıca kesmeleri de kontrol edebilme kabiliyetine sahip. Süreç boyunca her adımı raporlamaya çalıştık. Tasarımımızı, kırmık üstü sistem örneklerini, çevre birimlerini,

test alt yapısını, dokümanları, ve yazılım kütüphanelerini de Github platformunda paylaşıma sunduk.

Bu RISC-V çekirdeğinin adını “Hornet” (Eşek Arısı) koyduk.

1. INTRODUCTION

Computers came into play in calculations that humans can't easily overcome, because of several possible reasons like excessive number of operands or extreme number of steps to reach the goal.

Innovation of electronic computing systems started in 40s and it has been a fast pace process both on academic and industrial sides.

The brain of established electronic computer systems of today is the "processor", that speaks with the other components, like memories or a keyboard, and works on the information that it has acquired from them. There are several proposed design styles and circuits for processors that differs in how the data flows or where the data and instructions are stored. Designing new architectures and developing and enhancing the existing ones is still an exciting topic, and a huge academic and industrial area.

The primary objective of this project is to produce a synthesizable 32-bit Reduced Instruction Set Computer-V(RISC-V) core in Verilog HDL. We aim to implement at least the "I" integer base instruction set, the "M" multiply/divide standard extension, and the Machine Level Instruction Set Architecture(ISA). We will verify the functionality of the core by running various C programs on it.

One of the shortcomings of the state-of-the-art RISC-V cores is that it is difficult to understand the low-level design, because it is not documented. We will make sure our design is simple, and easy to understand by thoroughly documenting it.

The project consists of three main stages: Developing the "I" base integer set, the "M" multiply/divide extension and the Machine Level ISA.

The first stage is to design and implement the "I" base instruction set defined in the Unprivileged ISA [1]. This task involves the design of the 5-stage pipeline that can execute the integer instructions listed in the RISC-V ISA.

The second stage is to add the "M" standard extension, which includes the multiply/divide instructions. We will look up the multiplication/division algorithms available in the literature, and pick the ones that suit our needs.

The third stage is to implement the Machine Level instruction set defined in the Privileged ISA [2]. Machine Level instruction set defines the instructions and registers necessary to control and monitor the processor's status.

1.1 General Information and Concepts

We have aimed to design and implement a processor by using the open source Reduced Instruction Set Computer – V (RISC-V) as our Instruction Set Architecture (ISA). We have used the basic information that we had, and also acquired the needed information that we hadn't had beforehand.

1.1.1 Binary numbers and two's complement format

The signals that electronic devices like computers most easily understand are on and off, and the corresponding symbols for the characters are "0" and "1", respectively. This representation is called the "binary representation", and the number system formed with this representation is the "binary numbers". Least Significant Bit (LSB), which is the rightmost bit, is the value 2^0, and the Most Significant Bit (MSB), which is the leftmost bit, is the value 2^{N-1}, where N is the number of bits forming the number. For example, a 4-bit binary number "1010" represents the number 9 in the decimal number system, and it is calculated as follows,

$$0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 9$$

There are a number of ways to represent negative numbers in binary form. The one that is most used in electronic circuits and most suitable one for the binary arithmetic is the "two's complement" representation. With this representation, negative numbers are constructed as taking the complement of a positive binary number, which means inverting each bit of the number, then adding "1" to the emerged number. N+1 bits are needed for representing numbers in the range of -2^N and (2^N-1). For example, to represent -9, "01010" is inverted to "10101" then 1 is added to the inverted number to obtain "10110".

1.1.2 A processor

Today's most popular processors use architectures derived from the design style called Von Neumann defined by John von Neumann and his friends from the ENIAC project, which is the first general-purpose electronic computer [Computer organization and design RISC-V edition]. This model proposes an architecture that in its basic form includes a processing unit with an Arithmetic and Logic Unit (ALU) and Registers, a Control Unit and a Program Counter, and a mutual Memory for the Instructions and Data. Each component has its own role and works together to carry through the processing.

The ALU is responsible for executing the arithmetic operations like addition, and logic operations like bitwise AND. Registers are memory components that are close to the ALU and hold the data that the ALU works on, that is either moved to the registers from the Memory or placed there by the Control unit. The Control Unit controls the rest of the processor with control signals generated specifically for each Instruction. The Memory holds the Data and Instructions in addresses in binary form, and the rest of the circuit can reach its content with the addresses.

1.1.3 Instruction set architecture

The processor obeys the orders given by us, the humans. These orders are given in a form called “instructions”, that are in binary representations; therefore, the hardware of the processor can recognize those instructions and then can follow them. For example, in an instruction set architecture, the instruction represented as “111000111” could employ the processor to perform addition on specified values.

The designs for computer architectures differ in which instructions they can execute with their hardware and how they do it. The design choices around these points make up an architecture. Different architectures may execute the same instruction differently, or they may have instructions specific for that architecture. A design that specifies a set of instructions and the properties of those instructions, that a circuit must be able to carry out is called an Instruction Set Architecture (ISA).

1.1.4 RISC-V instruction set architecture

The Reduced Instruction Set Computer - V (RISC-V) is an ISA, that is a side product of a 5-year project carried out in University of California, Berkeley by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman. The RISC-V ISA specifies the minimum set instructions that a possible implementation should be able to fulfill, and additional subsets of instructions and higher-level specifications for more complex designs. The RISC-V ISA does not define how a design must be implemented or which subsets it must contain, but rather specifies a balanced and well-designed set of instructions that a designer can make decisions on what to use and how to implement.

As stated in the RISC-V organization webpage, “The worldwide interest in RISC-V is not because it is a great new chip technology, the interest is because it is a common free and open standard to which software can be ported, and which allows anyone to freely develop their own hardware to run the software.” [4]. These properties of the RISC-V ISA make it ideal for our desired use.

1.1.4.1 Integer instruction set

The integer instruction set consists of the most basic instructions; such as addition, subtraction, branch, jump, and memory operations. Integer instruction sets are available in 32, 64 and 128-bit options. They are also the base intruction sets, which means that every RISC-V core must subsume at least one of the integer instruction sets.

1.1.4.2 “M” standard extension

The “M” Standard Extension adds multiplication and division features to the repertoire of a RISC-V core. The subset includes the instructions “MUL”, “MULH”, “MULHU”, and “MULHSU” for the multiplication; “DIV”, “DIVU”, “REM”, and “REMU” for division.

The specifications for these instructions are given in the RISC-V Instruction Set Manual [1] and are briefly given below.

With 32-bit operands, “MUL” instruction performs multiplication on two source registers and stores the lower 32 bits of the 64-bit result value to the destination register. “MULH” instruction stores the higher 32 bits of the result after the same

operation. For “MUL” and “MULH” instructions, the operands are treated as signed numbers. “MULHU” instruction stores the higher 32 bits of the result, with both operands treated as unsigned numbers. “MULHSU” again stores the higher 32 bits of the result, with the first operand treated as signed while the other as unsigned.

With 32-bit operands, “DIV” and “DIVU” instructions perform multiplication on two source registers and store the 32-bit quotient value to the destination register with the operands treated as signed or unsigned, respectively. “REM” and “REMU” instructions perform multiplication on two source registers and store the 32-bit remainder value to the destination register when the operands are treated as signed or unsigned, respectively.

Arithmetic circuits in computer architecture are realizations of arithmetic algorithms. These algorithms are selected or constructed with the capabilities of the current hardware technology in mind, they should be suitable to be constructed. For the part we are interested in, which is binary multiplication and division, there are several proposed algorithms and architectures[7][8].

Multiplication

For multiplication, the main steps are generation of partial products and addition of them[7]. Optimizing the generation of partial products is a way to speed up the multiplication, and a popular multiplication method for this purpose is Booth's Algorithm, and its modified variations[7]. For implementing large multipliers that work with big operands, an efficient method is to partition the operands into smaller chunks, then use smaller bit multipliers to obtain partial products. The next part is to align these products properly and add them to acquire the final result. There are several ways for optimizing this addition using Carry-Save Adders(CSA), like Wallace Tree[7].

Division

Division is more complex than multiplication, and they often take the most time to be executed in computer architectures[8]. The types of dividers are sequential dividers, array dividers and dividers implemented with multipliers[8]. The most well-known sequential divider algorithm is SRT Algorithm[7]. To speed up the sequential dividers, High-Radix Divider algorithms are constructed. Array Divider architectures are suitable for pipelining applications, so their critical paths can be adjusted for required speeds. But their design is more complex and they are larger, when implemented.

1.1.4.3 Privileged architecture

The privileged architecture of RISC-V describes privileged instructions and mechanisms to utilize operating systems and external devices. It defines several privileged instructions and registers that allow for interrupts, exceptions, hardware identification, configuration and more. Moreover, in order to satisfy RISC-V compliances, the privileged architecture must be implemented to some degree.

RISC-V defines four privilege modes: User, Supervisor, Hypervisor, and Machine modes. Briefly, the concept of privilege levels is required to ensure stability and safety of the processor. Each privilege mode has a certain degree of access to the hardware, with the User mode being the least privileged and the Machine mode being the most privileged. In other words, only the Machine mode software has complete access to the underlying hardware. Therefore, in a RISC-V compliant processor, at least the Machine privilege mode must be implemented.

Each privilege level has its own set of privileged instructions and registers. These registers are called “CSR Registers”. They are used to configure and monitor the processor’s state. Their functions range from storing the program counter value of the interrupted instruction to enabling and disabling interrupts. In order to read and write to these registers, RISC-V defines 6 CSR instructions.

2. DESIGN AND IMPLEMENTATION OF THE INTEGER PIPELINE

Our first goal in this project was to design the hardware that can execute the integer instructions in the RV32I Base Integer Instruction Set. The instructions that are included in this set are shown in Figure 2.1,

Category	Name	Fmt	RV32I Base	
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link	J&L	UJ	JAL	rd,imm
	Jump & Link Register	UJ	JALR	rd,rs1,imm

Figure 2.1 : List of RV32I instructions

2.1 Microarchitecture of the core

In computer architecture domain, the term "microarchitecture" refers to the underlying implementation of the processor. For example, the number of pipeline stages, the task of each pipeline stage and the inner working mechanisms of each stage are microarchitectural decisions.

2.1.1 Designing the datapath for integer instructions

Initially, we designed a datapath for each instruction in the RV32I set. A datapath in this context refers to the physical flow of execution of an instruction. To design a datapath, we determined what functional units each instruction required, and connected them appropriately to realize the datapath. For example, the "ADD" instruction is supposed to read two registers, add them together, and finally write the result to a register. This means, there must be a register file that houses the registers, and an ALU to perform the binary addition operation. Another example can be the "Load Word" instruction, which is supposed to read a register, add a 32-bit immediate value to that register to form the address of the data, read the data from the memory, and finally write that data to a register. This indicates that there must be a data memory and an immediate decoder in the datapath, in addition to a register file and an ALU. Figures 2.2 and 2.3 show the diagram for these datapaths,

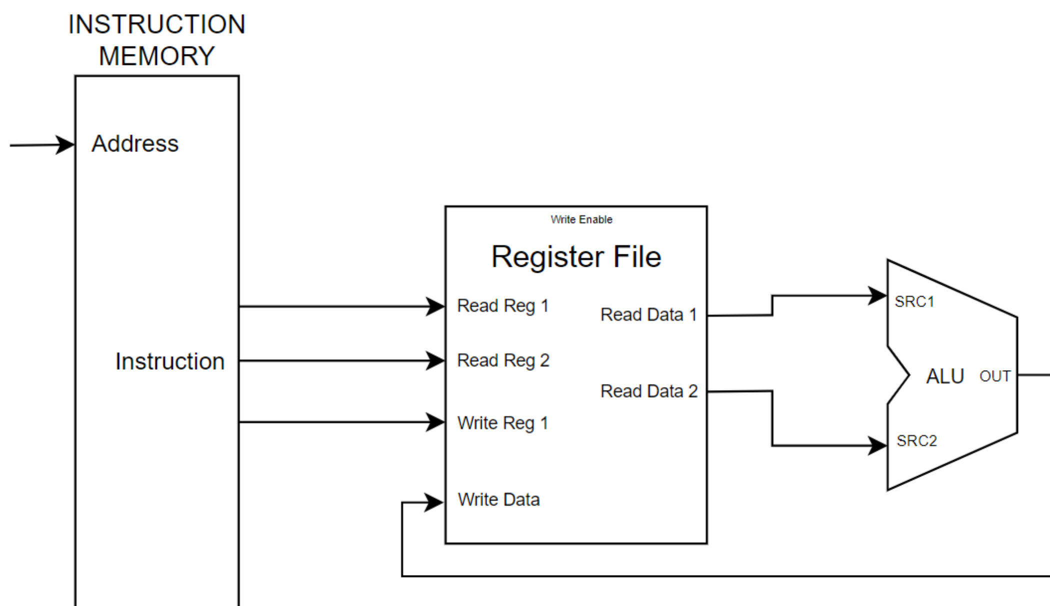


Figure 2.2 : Datapath for the ADD instruction

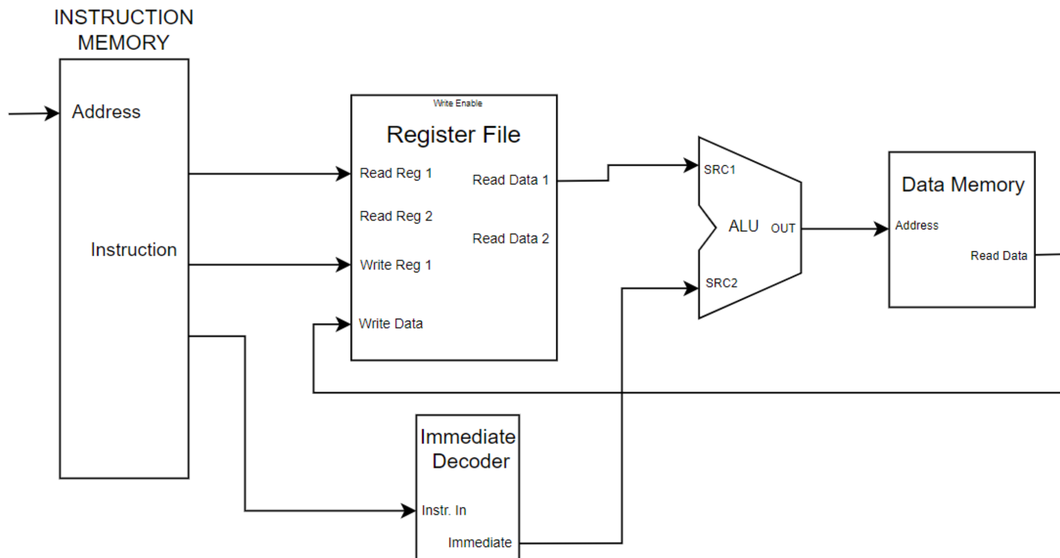


Figure 2.3 : Datapath for the Load Word instruction

Once we determined the datapaths for all of the instructions, we combined them all together to form the complete datapath. We also added muxes where necessary. Figure 2.4 shows the complete datapath for the RV32I instruction set. Note that the control signals for muxes, which are generated by the Control Unit, are omitted in the diagram to avoid cluttering.

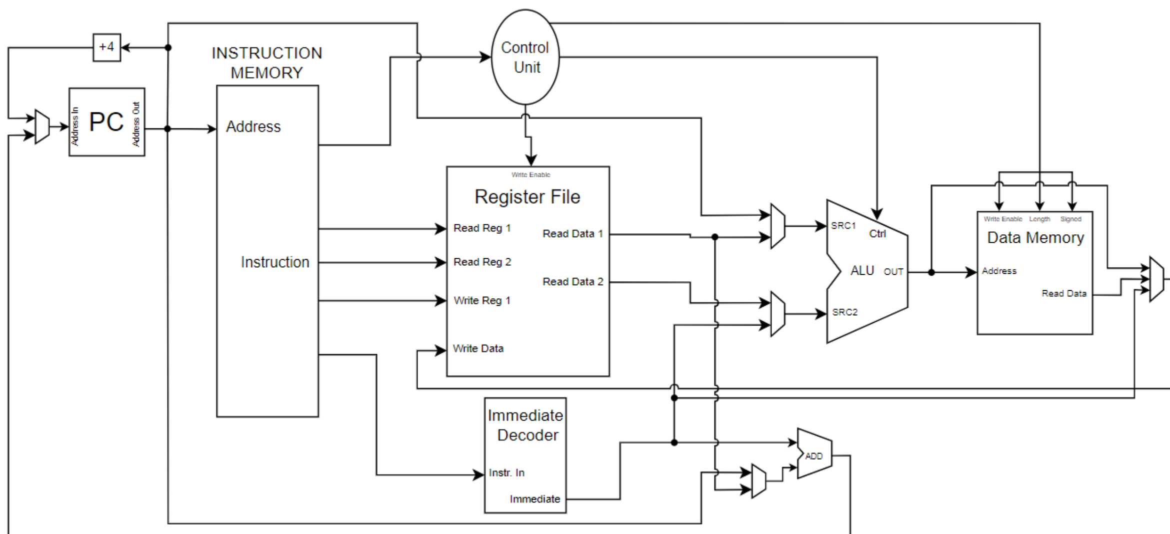


Figure 2.4 : Complete datapath for RV32I instructions

The Instruction Memory is where the instructions are stored. At each clock cycle, a new instruction is read from there and executed in the datapath. The PC is a register which holds the address of the next instruction that will be fetched from the memory.

PC is incremented by 4 at each cycle. The mux at the input of PC determines where the next instruction address comes from. When a branch or a jump occurs, address of the next instruction is determined by the 32-bit adder. Register File houses all 32 of the 32-bit registers defined in the RISC-V standard. It has two ports to read, and one port to write. Control Unit generates the control signals for Register File, muxes, ALU and Data Memory. Immediate Decoder generates 32-bit immediate values from instructions, as defined in the RISC-V standard. ALU executes all the arithmetic operations that the instructions require, which involve addition, subtraction, shifts, logic operations and comparisons. Data Memory stores the program data. The processor can store or load 8, 16 and 32 bits of data to and from the Data Memory.

2.1.2 Pipelining the datapath

2.1.2.1 What is pipelining?

Pipelining is a well-known technique in hardware design. From ADCs to CPUs, many different structures utilize pipelining to increase performance.

The problem with a traditional datapath is that the signal propagation delay from the beginning to the end can be quite long. This long delay is a natural by-product of electronic circuits. In a typical pipelining scheme, the idea is to split the datapath into "pipeline stages", and insert "pipeline registers" between the stages. This way, with every clock cycle, the signals are only required to propagate from one stage to the next. Therefore, the average propagation delay in the datapath is reduced, and the frequency of operation can be increased.

2.1.2.2 Design of the pipeline

In our design, we decided to use a 5-stage pipelined microarchitecture. This is a typical pipelining scheme for simple RISC processors. The pipeline stages are as follows, in order,

- 1. Instruction Fetch (IF)
- 2. Instruction Decode (ID)
- 3. Execute (EX)
- 4. Memory (MEM)

- 5. Writeback (WB)

In the Instruction Fetch stage, the next instruction is fetched from memory.

In the Instruction Decode stage, the instruction is decoded in the Control Unit, and control signals are generated. In addition to that, Immediate Decoder block generates the 32-bit immediate value, and the registers are read from the Register File.

In the Execute stage, the ALU performs the arithmetic operations for the instruction. Branch target address is also calculated in this stage.

In the Memory stage, data is either stored to or loaded from the Data Memory.

In the Writeback stage, the data is written to the Register File.

Between the stages, there are pipeline registers that convey the necessary signals from the previous stage to the next stage. For example, between IF and ID stages, there is a pipeline register that stores the instruction and the associated PC value.

2.1.2.3 Pipeline hazards

In pipelined processors, there are multiple instructions being processed at a given time. The problem is that an instruction at an earlier stage might depend on an instruction that is not yet completed. For example, consider the code shown in Figure 2.5,

```
lw   x1,x4,20
lw   x2,x4,16
lw   x3,x4,16
addi x1,x1,5
slli x1,x1,4
```

Figure 2.5 : Assembly code that causes a pipeline hazard

The code in Figure 2.5 loads three values from the memory, then performs an addition and a logical shift operation on `x1` register. The issue with this code is that the logical shift instruction `slli` depends on the addition instruction `addi`. In the pipeline, the addition instruction will write its result to the Register File in the Writeback stage, but the shift instruction requires the data before that. This issue is called a pipeline hazard. Fortunately, the solution is quite simple. We exploit the fact that the result of the addition instruction is calculated in the EX stage. Then, the shift instruction need not wait for the addition to write the result to the Register File. Instead, when the shift

instruction reaches EX stage, the addition instruction will have reached the MEM stage, we simply forward this result from the MEM stage to the EX stage, before it is written to the Register File. However, not all pipeline hazards can be resolved by forwarding data. As an example, consider the code shown in Figure 2.6,

```
lw   x1,x4,20
addi x1,x1,5
slli x1,x1,4
lw   x2,x4,16
lw   x3,x4,16
```

Figure 2.6 : Assembly code that causes a pipeline hazard

This time, the addition instruction depends on the memory load instruction. The load instruction will have the data ready at the end of MEM stage, but the addition instruction requires the data before that. This hazard cannot be resolved by forwarding alone. The addition instruction must stall and wait for the load instruction to read the data from memory. A stall is when the pipeline basically stops execution. A NOP instruction, which does not change anything visible in the processor, is inserted into the pipeline to realize the stall. After stalling for one cycle in the ID stage, addition instruction can advance to the EX stage, and load instruction can forward the data from the WB stage.

In summary, there are two countermeasures for pipeline hazards: forwarding and stalling. Forwarding can resolve majority of the data dependencies, but a stall is necessary if an instruction depends on a load instruction.

In our core, we designed two modules to handle pipeline hazards: Forwarding Unit and Hazard Detection Unit. The Forwarding Unit detects data dependencies and forwards data when necessary. The Hazard Detection Unit detects data dependencies that require pipeline stalls, and stalls the pipeline.

Figure 2.7 shows the complete diagram of the pipeline that is capable of executing RV32I instructions,

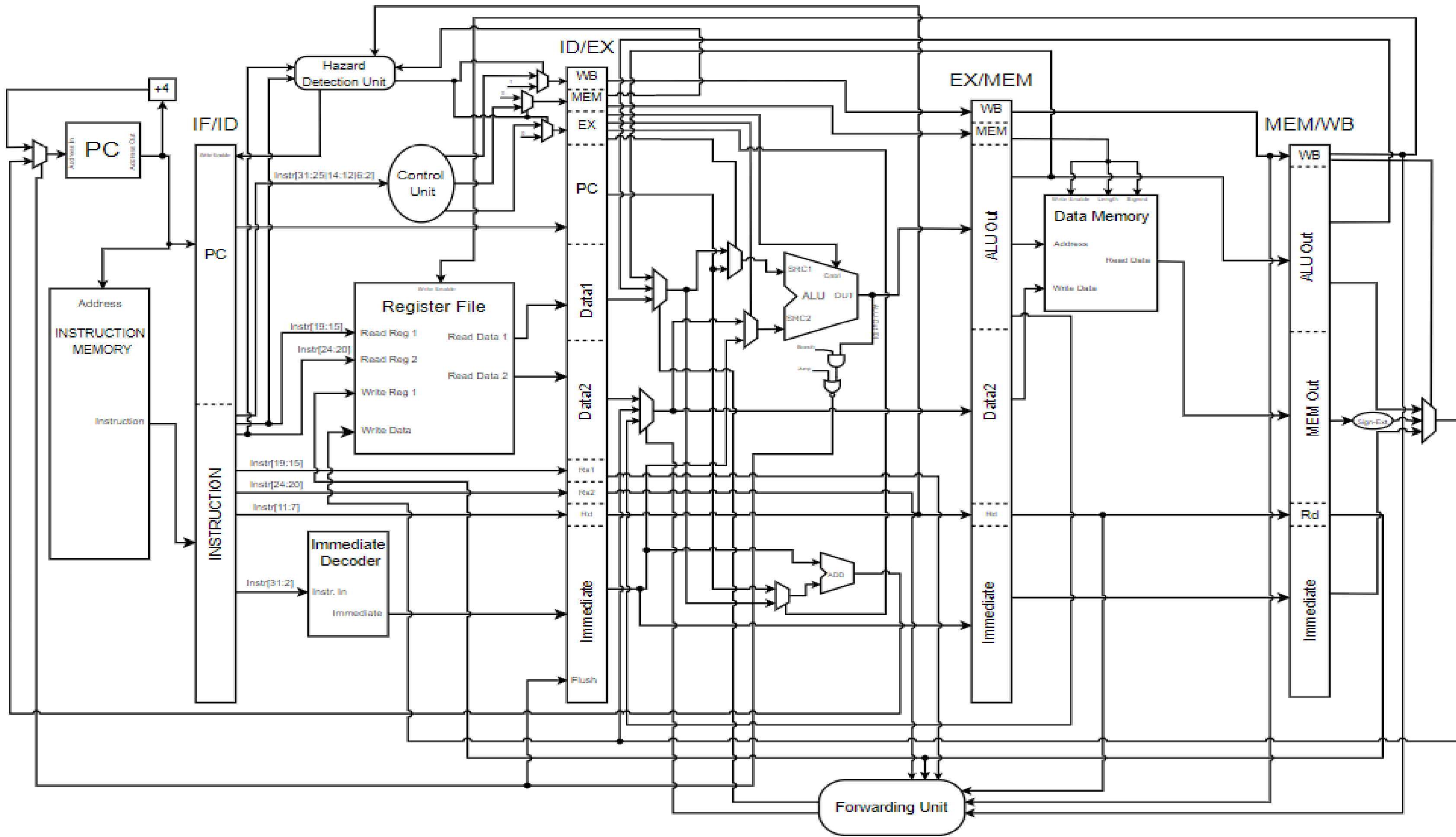


Figure 2.7 : Complete pipeline diagram for RV32I instructions

2.2 Implementation of the design

2.2.1 Verilog description of the design

Implementing a hardware design consists of several iterative steps. In digital design, the first step in implementation is describing the design in a Hardware Description Language. HDLs are very commonly used to describe digital hardware. In a way, HDLs define a set of rules to describe hardware in text format. This allows a formal description of any hardware, which in turn allows different groups of people to efficiently describe and understand hardware. Moreover, modern CAD tools can simulate HDLs in a cycle-accurate manner. This greatly simplifies the design of complex hardware, such as processors. Most importantly, such CAD tools can also synthesize real hardware from the HDL code of the design. Without such conveniences, it would be incredibly difficult to implement any meaningful hardware.

In practice, the HDL description of a design is also called the "RTL" description.

In our project, we used Verilog[5] HDL to describe our design. We wrote the following Verilog modules,

- ALU – Executes arithmetic operations
- Control Unit – Generates the control signals
- Hazard Detection Unit – Detects pipeline hazards and stalls the pipeline
- Forwarding Unit – Detects pipeline hazards and forwards data
- Immediate Decoder – Generates 32-bit immediate values
- Load-Store Unit – Generates Data Memory interface signals
- Core – Top module where all the submodules are instantiated and the pipeline is described

Figure 2.8 shows the module hierarchy in a visual way,

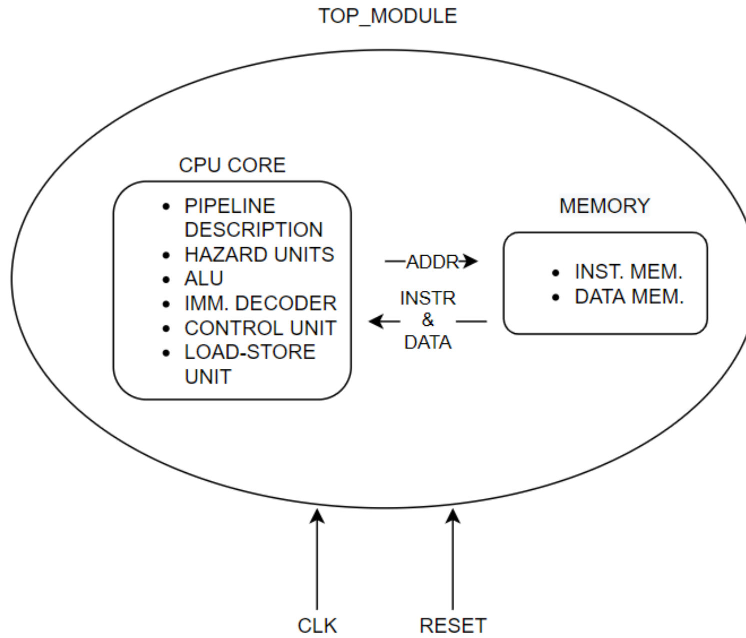


Figure 2.8 : Module hierarchy

It should be noted that the memory is not a part of the core, hence the separation in the diagram.

2.2.2 Synthesis of the RTL description

Once we prepared and tested our RTL descriptions, we proceeded to synthesize the design. Synthesis is the process of generating a netlist that consists of real primitive cells. A "cell" in this context is simply a logic circuit block, such as an And-Or-Invert, a Full-Adder, a NAND gate, a Flip-Flop etc.

There are several synthesis programs out there, some of them are free to use, others are proprietary. We used Genus synthesis tool from Cadence, which is provided by the VLSI Lab at our faculty. Genus expects several inputs from the user, where the essential ones are listed below,

- RTL files of the design – The Verilog source files, in our case
- An ASIC cell library – We used TSMC's 90nm general purpose cell library
- Timing constraints: Clock frequency, input and output delays.
- Load at the output ports
- Driving cells of the inputs

Genus simply generates a netlist of cells that is functionally equivalent to the RTL description of the design, while also meeting the specified constraints, if possible. More specifically, it generates a Verilog file that describes the netlist, an SDF[6] file that contains the cell delays, and report files that contain detailed information about the timing, gate count and area consumption of the design. We then use the Verilog and the SDF file to run post-synthesis timing simulations.

In practice, a synthesis script is written to automate this process. The script contains all the constraints and commands necessary to complete the synthesis process. The script we used in this project is shown in Appendix A.

3. DESIGN AND IMPLEMENTATION OF THE PRIVILEGED ARCHITECTURE

3.1 About the Privileged Architecture

In a typical computer, there are several levels of privilege in the hardware level for security purposes. Traditionally, each level is associated with a mode of operation, and at any given time, the software is running in one of the available modes. The important part is that each mode has a certain level of access to the underlying hardware; hence the name "privilege". For example, the mode with the highest privilege level has access to all the registers and memory regions, whereas the mode with the lowest privilege level might have access to only a limited portion of them. As the privilege levels are enforced in the hardware, the software –under normal circumstances– has no way to bypass this security scheme.

This privileged architecture is necessary to realize a stable and secure execution environment. For example, in a computer with an operating system installed, there are often multiple programs running simultaneously. The only way to ensure that these programs do not interfere with each other, willingly or not, is to utilize a privilege scheme, as described above. This way, the user programs would run in a lower privilege mode, whereas the operating system would run in a higher privilege mode.

Thus, the programs would not be able to interfere with each other, because they would simply lack the privilege to do so.

In the case of RISC-V, the privileged architecture is described in the Volume II of the Instruction Set Manual[2]. There are 4 modes of operation: Machine, Hypervisor, Supervisor and User, in decreasing privilege level. Each mode has its dedicated set of registers called Control and Status Registers and instructions. These registers and instructions allow the core to handle interrupts and exceptions. The spec only mandates the provision of Machine mode; the rest are optional. The more modes are provided, the better the security of the processor.

In our project, we only implemented the Machine mode, which involves several new registers and instructions.

3.2 Design of the Privileged Architecture

3.2.1 Requirements

Design of the Machine-Level architecture involves the following new registers and instructions,

- Machine-Level Control and Status Registers
- MRET instruction
- "Zicsr" standard extension, which includes 6 new instructions

The Machine-Level CSRs provide essential information and functionalities to the core; such as the instructions supported by the core, length of the instructions in bits, interrupt enable and disable, interrupt handler address and more. The instructions included in the "Zicsr" extension are called CSR instructions. They are used to read and write to the CSRs. The MRET instruction is used to return from an interrupt or an exception handler.

3.2.2 Machine-Level Registers

We designed a new module called "CSR Unit", which implements the registers and a finite-state machine that handles interrupts and exceptions. The registers are written on the falling edge of the clock and on the WB stage of the instruction. The unused parts of the registers are hardwired to 0. Following is the list of implemented registers,

- `mstatus` – Machine Status: Contains the global interrupt enable/disable bit.
- `mie` – Machine Interrupt Enable: Contains the Machine-Level interrupt enable/disable bits
- `mip` – Machine Interrupt Pending: Contains the Machine-Level interrupt pending bits. A bit is set if its associated interrupt is pending.
- `mcause` – Machine Cause: Holds the cause of the interrupt or exception.
- `mtvec` – Machine Trap Vector: Holds the base address of the interrupt/exception handler.
- `mepc` – Machine Exception PC: Holds the address of the interrupted instruction
- `mscratch` – Machine Scratch: Dedicated register for Machine-Level code.

3.2.3 Interrupts and Exceptions

In addition to the Machine-Level registers, RISC-V standard defines the following interrupt sources,

- Software interrupts – Set and cleared by a write to a memory-mapped register
- Timer Interrupts – Triggers when the timer register exceeds the timer compare register
- External Interrupts – Set and cleared by an external interrupt controller
- Fast Interrupts, 16 of them – Platform specific interrupts

Moreover, the finite-state machine also handles several sources of exceptions, as listed below,

- Instruction Access Fault – Generated when an error occurs during an instruction access.
- Instruction Address Misaligned – Generated when the address of an instruction is not 4-byte aligned.
- Illegal Instruction – Generated when an illegal instruction is encountered.

- ECALL – Environment Call exception. Generated by an ECALL instruction, which is used to generate system calls.
- EBREAK – Environment Break exception. Generated by an EBREAK instruction, which is used to return control to debugging environment.

Each interrupt/exception source sets the `mcause` register to a unique value. Exceptions are always enabled, by definition. An interrupt is taken if the interrupts are globally enabled, and the associated bits are set in the `mie` and the `mip` registers. Then, depending on the interrupt handling mode, the hardware calculates the handler address, as elaborated below,

- Vectored mode: Handler address = `mtvec` + 4 × `mcause`
- Direct mode: Handler address = `mtvec`

Finally, the PC is set to the address of the handler.

The finite-state machine realizes the task of handling interrupts and exceptions. When an interrupt or an exception occurs, it performs the following actions,

- Flush the pipeline
- Disable interrupts globally
- Save the PC of the interrupted instruction to the `mepc` register
- Set the value of `mcause` register depending on the cause of the interrupt/exception
- Set PC to the address of the handler

When an MRET instruction is encountered, this process is reversed. The global interrupts are enabled; PC is set to the value in the `mepc` register.

Figure 3.1 shows the algorithmic state machine chart of this finite-state machine,

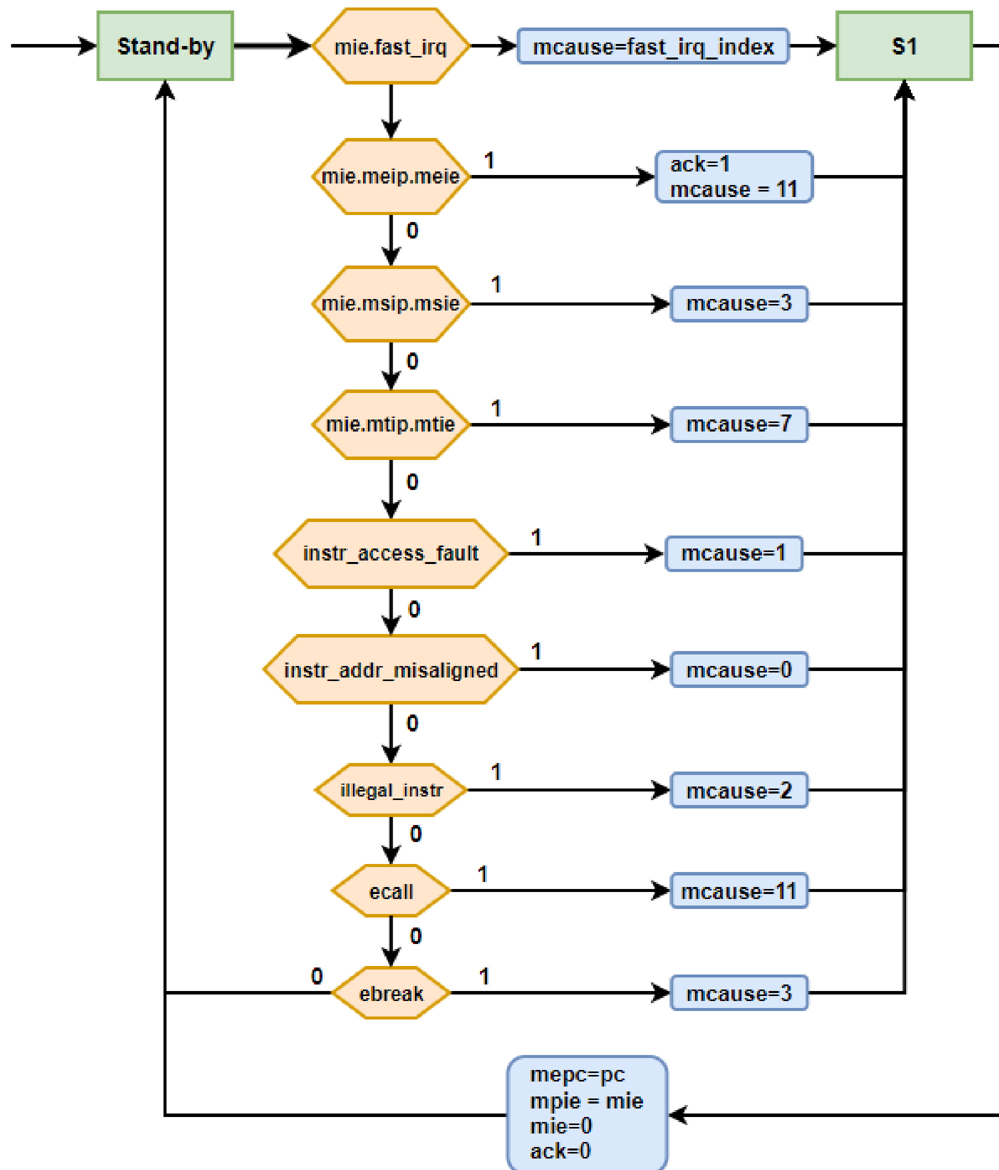


Figure 3.1 : ASM chart of the finite-state machine

The FSM is initially at the Stand-by state. On every rising edge of the clock, if there is a pending interrupt or an exception, FSM will switch to the S1 state, while setting the value of the `mcause` register. On the next rising edge, it will switch back to the Stand-by state, while disabling global interrupts and saving the PC to the `mepc` register.

The CSR Unit module also has a pipeline flushing mechanism that controls the flush signals. When an interrupt or an exception occurs, this mechanism flushes the oldest valid instruction that is not yet retired, and the instructions before that. The PC of the latest flushed instruction is saved. What complicates matters is that not all instructions

might be valid in the pipeline at a given time. For example, there might be a NOP instruction inserted by a pipeline stall. The flushing mechanism can and should detect this dummy instruction, and avoid saving its PC. Another problem is that a store instruction retires at the end of MEM stage, whereas almost all the other instructions retire at the end of WB stage. This means, the flushing mechanism should also avoid flushing a store instruction that is already retired.

3.2.4 CSR Instructions

The Zicsr standard extension defines 6 new instructions. Figure 3.2 shows the encoding of these instructions,

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 3.2 : Encoding of the CSR instructions

All CSR instructions read a CSR into an integer register, and modify the CSR. There are dozens of CSRs defined in the RISC-V standard. Therefore, an addressing scheme is utilized to distinguish the CSRs. Each CSR has a 12-bit address, which is encoded in the instruction as the “csr” field. The “rs1” field encodes either the integer register or the immediate that acts as the bit mask. The “rd” field encodes the integer register where the CSR will be read into.

Following list briefly explains the functionalities of the instructions,

- CSRRW – CSR Read/Write: Swaps the values of a CSR with an integer register
- CSRRS – CSR Read/Set: Reads the value of the CSR into an integer register, and sets the bits in the CSR. The integer register acts a mask.
- CSRRC – CSR Read/Clear: Reads the value of the CSR into an integer register, and clears the bits in the CSR. The integer register acts a mask.

- CSRRWI, CSRRSI, CSRRCI – Similar to the instructions above, except that an immediate is used as a bit mask, instead of an integer register.

We had to do several changes to the pipeline design to implement these instructions. We added several pipeline registers to hold the address of the CSR, modified the ALU to perform masking operations, added new muxes and added forwarding logic for CSRs as well. We also extended the Control Unit to accommodate for the new instructions.

We also significantly improved the overall design of the pipeline. Figure 3.3 shows the updated pipeline diagram.

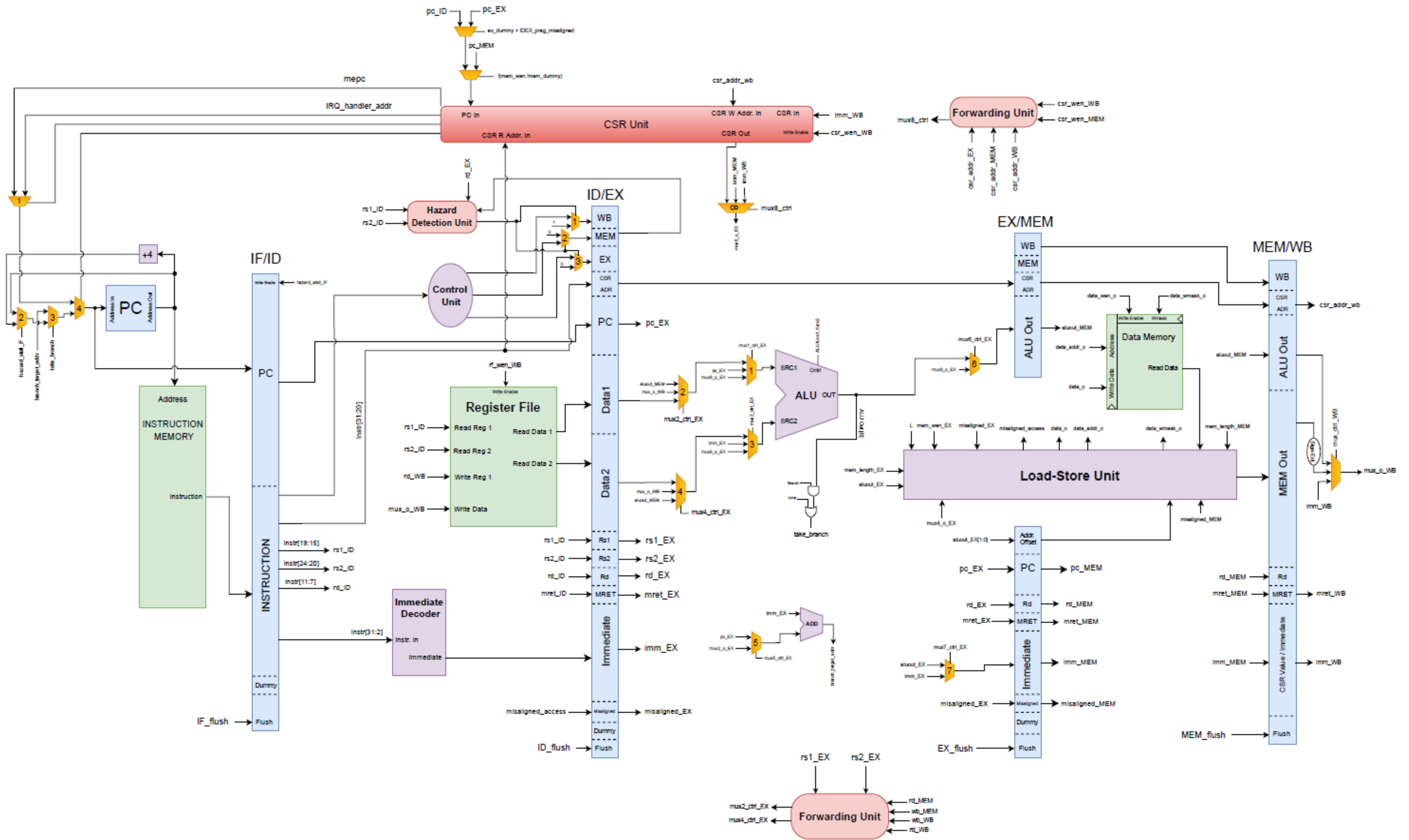


Figure 3.3 : Updated pipeline diagram with the privileged architecture

3.3 Implementation of the design

We wrote the new module, CSR Unit, in Verilog HDL. We also made the necessary changes to the RTL code of the existing modules. We then synthesized the updated design in Genus with the same synthesis script. In other words, implementation of the privileged architecture is identical to the unprivileged architecture.

4. DESIGN AND IMPLEMENTATION OF THE “M” STANDARD EXTENSION

As it was described in the introduction under the “M” extension title, RISC-V ISA introduces a total of eight multiplication and division instructions, four for each. The opcodes for the instructions are given in Figure 4.1.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 4.1 : Instruction opcodes for the “M” standard extension

After reviewing “M” Standard Extension requirements, and design techniques with examples, the effort for designing a suitable multiplier and divider circuit was started.

The symbol “A” is used for the Multiplicand 1 of the multiplication instructions and Dividend of the division instructions; and the symbol “B” is used for the Multiplicand 2 of the multiplication instructions and the Divisor of the division instructions, and they will be frequently used in this chapter.

4.1 Multiplier

After reviewing the literature and circuit examples for desired operation of multiplying 32-bit operands, we decided a design where the operands are first partitioned then

multiplied in an order, and after that they are added again in an order to achieve the final result.

4.1.1 Multiplier algorithm

The partitions are constructed according to the following equations for the multiplication algorithm.

$$A=(A_H 2^{16}+A_L) \quad B=(B_H 2^{16}+B_L) \tag{3.1}$$

With equation (3.1) used in A × B operation:

$$A\times B=(A_H 2^{16}+A_L)\times(B_H 2^{16}+B_L)=A_H B_H 2^{32}+(A_H B_L+A_L B_H)2^{16}+A_L B_L \tag{3.2}$$

With these steps, four partitions $A_H B_H$, $A_H B_L$, $A_L B_H$, and $A_L B_L$ are constructed. Each partition can also be partitioned. For example, for $A_H B_H$:

$$A_H=(A_{HH} 2^8+A_{HL}) \qquad B_H=(B_{HH} 2^8+B_{HL}) \tag{3.3}$$

$$A_H\times B_H=(A_{HH}2^8+A_{HL})\times(B_{HH}2^8+B_{HL})=A_{HH}B_{HH}2^{16}+(A_{HH}B_{HL}+A_{HL}B_{HH})2^8+A_{HL}B_{HL} \tag{3.4}$$

With the use of equations (3.1), (3.2), (3.3) and (3.4), operands are partitioned to A_{HH}, A_{HL}, A_{LH}, and A_{LL} for A, and B_{HH}, B_{HL}, B_{LH}, and B_{LL} for B. These partitions are 8-bit, and are multiplied in an order for generating the second level partitions $A_H B_H$, $A_H B_L$, $A_L B_H$, and $A_L B_L$. These second level partitions are then added in an order to generate the final result. This algorithm will be realised with a design consisting of 8-bit multipliers and adders to construct the 32-bit multiplier circuit.

4.1.2 Design of the multiplier

There are instructions for both signed and unsigned multiplication, so the circuit must be able to carry out both unsigned and signed multiplication. For that purpose, an additional bit is prepended to the 8-bit chunks, making them 9-bit chunks. For there are 9-bit portions, some of the multipliers are 9-bit multipliers. These multiplier modules use the "*" operator of Verilog. Figure 4.2 shows the diagram of the first version of the design.

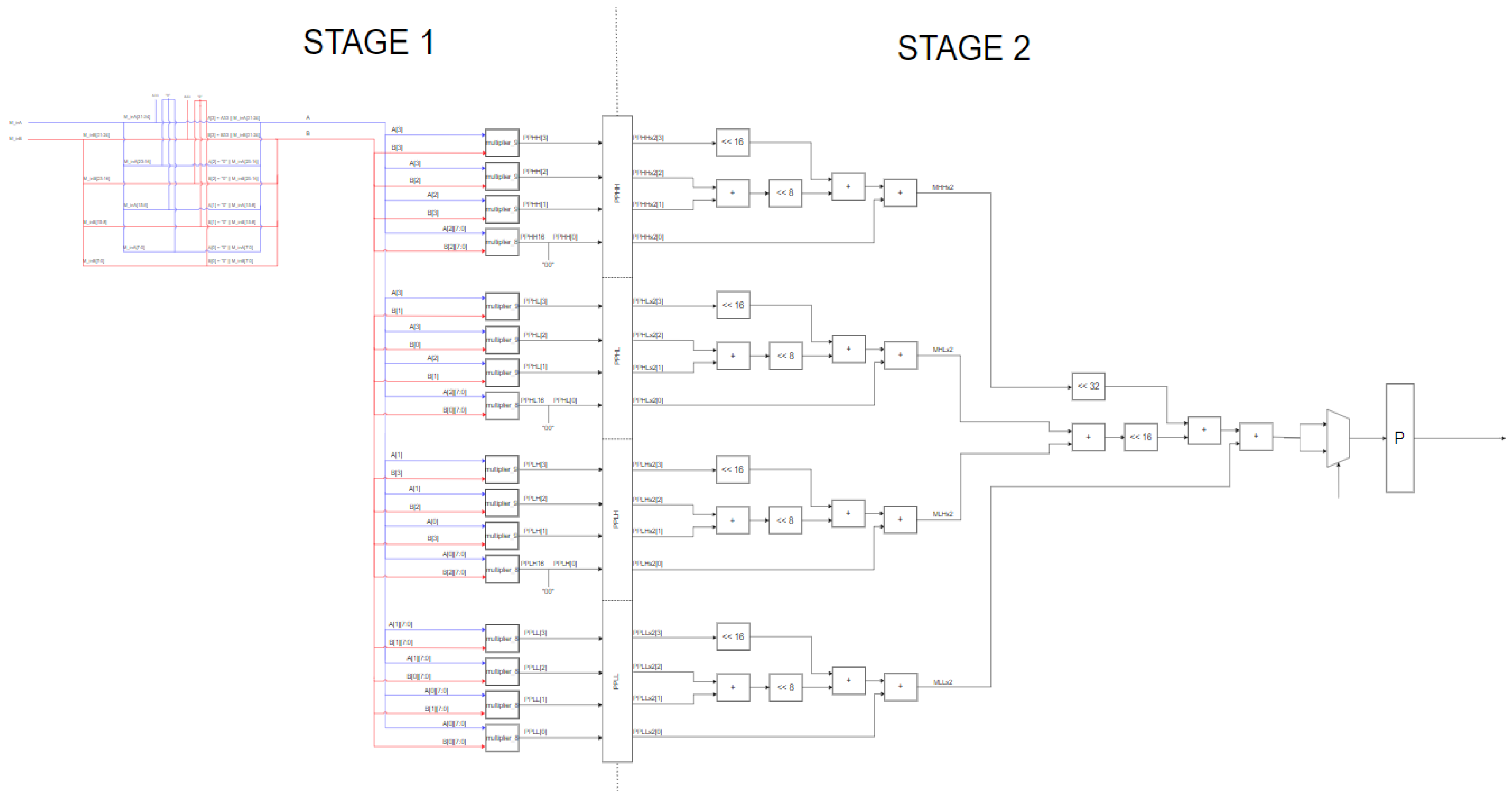


Figure 4.2 : Circuit diagram of the multiplier

4.2 Divider

For desired operation of integer division, there are a number of suitable design styles. A simple sequential divider that takes N clock cycles to calculate the result could be used with little design complexity, where N is the bit length of the operands. Or, a more complex combinational array divider could be utilized, which would have the result ready in a given number of clock cycles. The former wields a small area, but its speed, which is how few number of clock cycles it requires to calculate the result, is low. The latter however is very fast, but it is a relatively large circuit which requires much more area on the implementation. After considering the mentioned trade-offs, we decided to design a circuit that is not a fully combinational array or a simple sequential divider, but a combination of those.

4.2.1 Divider algorithm

Arithmetic circuits realize algorithms that are constructed for a desired operation. We decided to use a combination of sequential and combinational circuits, so we needed an algorithm that would suit this style of circuitry.

A strong candidate for a combination of sequential and combinational style of circuitry would be pipelining of combinational parts. For a 32-bit division operation, at most 32 subtraction operations are needed. These subtractions can be executed in groups combinationally, then these groups can be executed sequentially to achieve the final result. The number of cascaded subtractions in a group will determine the critical path, and the number of groups will determine the number of clock cycles required for finalization of the operation.

The subtraction number in a group was initially chosen as 4, which means there would be 4 32-bit cascaded subtraction operations. This means, division operations would complete in 8 cycles. However, we quickly realized that this scheme had a very long critical path delay. Therefore, we changed the design to 16-cycle division.

With that decision made, the algorithm has been constructed. The algorithm includes two for blocks. In the inner for block, 2 bits of the quotient is produced and a temporary remainder is generated to use in the next turn of the same for block. The outer for block ensures for the inner block to repeat 16 times. The algorithm written in the style of a programming language is given in the Figure 4.3.

```
A[31:0] : Dividend
B[31:0] : Divisor
Q[31:0] : Quotient
R[31:0] : Remainder
R_temp[32:0] : Temporary Remainder
Q_temp[3:0] : Temporary Quotinent Stack


R = 0
R_temp = 0

for(i, 15 downto 0){

        for(j, 1 downto 0){

                R = {R[31:1], A[2 * i + j]}
                R_temp = R - B

                Q_temp[j] = R_temp[32] // Carry Out

                if(Q_temp[j])
                        R = R_temp[31:0]
                else
                        R = R
        }

        Q[2 * i + 3 : 2 * i] = Q_temp
}
```

Figure 4.3 : Division algorithm

4.2.2 Design of the divider

The circuit that realizes the algorithm needs a block that will carry out the operations in the inner for block. And then, that block either can be replicated 16 times and aligned in a serial fashion, or a single block can be used 16 times with control components helping with the adjusting of inputs for the block. The first option would make the circuit capable of performing up to 16 division operations back to back, because a block can take the next division operations information as input after it outputs the previous division operation's information to the next block. With the second option, the circuit would be able to execute the next division operation in line after it finalizes the current one, but in return it would be approximately 16 times smaller in space.

The latter option was chosen, because we didn't have a specific division-heavy application that we are planning on using the core with. Also because of division operations being not that frequent in most of the applications, the second choice was more logical in our case.

With the algorithm and design style decided, a diagram is constructed for realizing the algorithm. Figure 4.4 illustrates the design of the circuit.

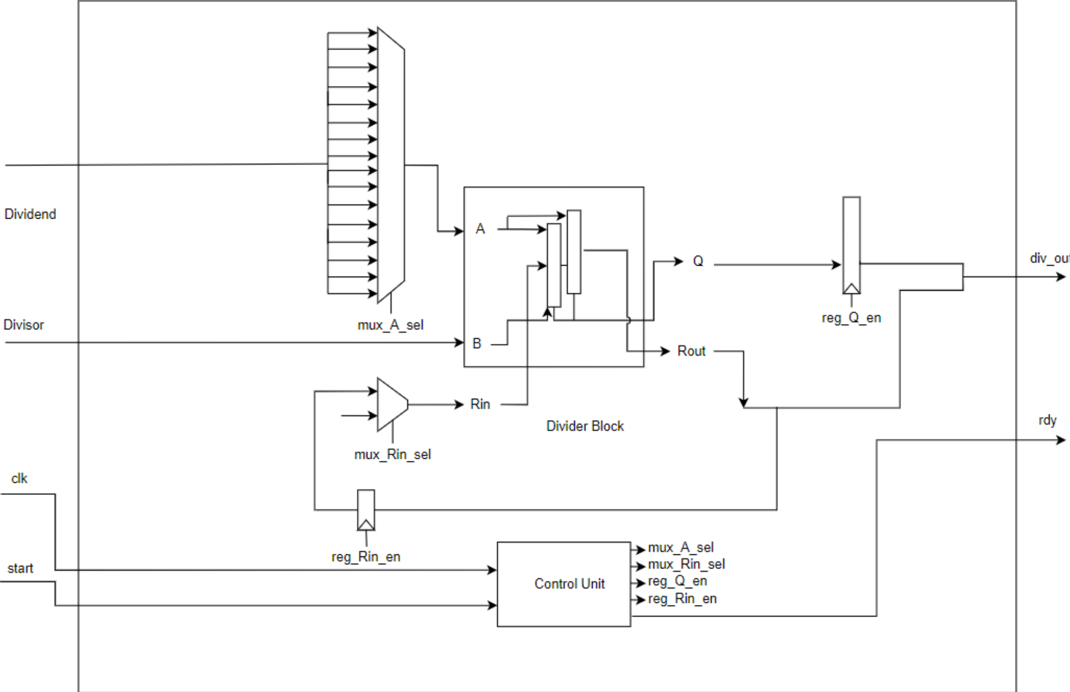


Figure 4.4 : Circuit diagram of the divider

The circuit includes a Divider Block, a Control Unit and registers and multiplexers for controlling the signals proper to the algorithm.

As decided, the Divider Block will be used over and over again in 16 rounds with different inputs, and each round 2 bits of the quotient will be generated and saved. And at the 16th round, the final remainder is ready.

4.2.2.1 Divider block

The divider block needs two arrays that will execute a subtraction on R and B, the divisor. Each array will calculate 1 bit of the quotient and a 32-bit R. The block then will output 2 bits of the quotient and a 32-bit remainder that is the R from the final array. The bits of the quotient need to be saved, and the remainder will be used again by the block at the next round. At the first round, the remainder input needs to be 0.

4.2.2.2 Control unit

The Control Unit is responsible for making sure that the rest of the circuit is operating correctly. It controls which signals will be inputted to the Divider Block, and where

the outputs from the Divider Block will go. The Control Unit is an Algorithmic State Machine, and it controls the circuit by determining the values of the control signals.

The State Diagram of the circuit is given in Figure 4.5.

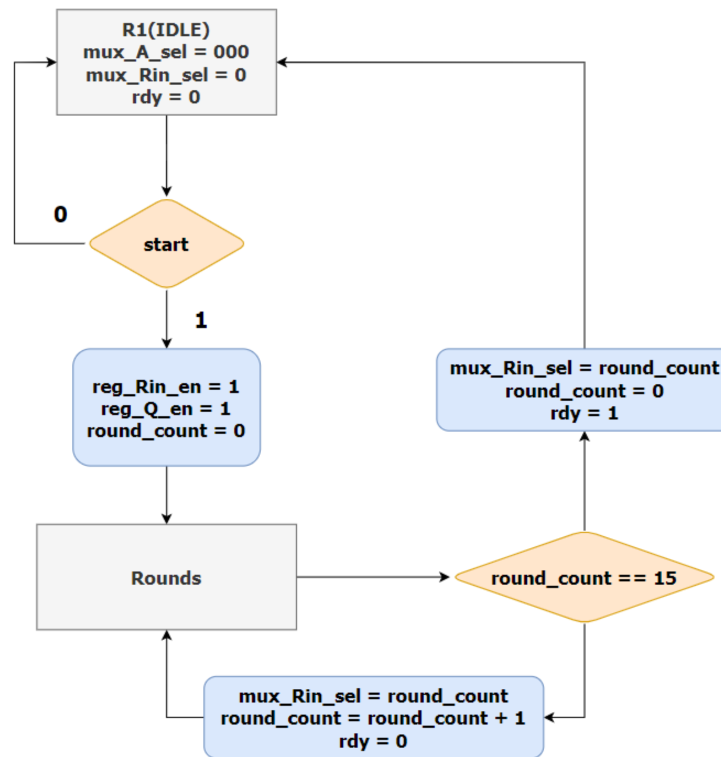


Figure 4.5 : State diagram of the divider

The circuit waits for the operation in the state IDLE. When the start signal is asserted, the division operation begins and the circuit goes through 16 rounds, during where it calculates required results and transfers them to the next round. In the final round, which is the 16th round, it asserts a ready signal and the final result for quotient and remainder is ready.

4.2.2.3 Testing of the initial design

Verilog code for the required modules was written, and then control components, multiplexers and registers, were combined on a top module to construct the final circuit.

The circuit was tested with a simple testbench, and emergent errors were corrected. The first circuit design for the divider was thus finalized.

4.3 The MULDIV Circuit

The initial design of the multiplier circuit is capable of handling signed operands, because of its relatively complex operand alignment and signed 9-bit multipliers. But the first design of the divider circuit is not able to work with signed inputs. To finalize the multiplier and divider unit, a signed divider circuit needs to be constructed. For fulfilling this task,

4.3.1 Signed divider circuit

For constructing a signed integer divider, a strong option from existing solutions is to implement input and output adjuster circuit that converts signed operands to unsigned inputs for the divider circuit, and convert unsigned result to a signed number if necessary, respectively. These adjuster circuits are relatively simple, compared to a possible solution that would change the design and algorithm, and can also be configured to handle the input and output of the multiplier circuit, making it possible to simplify the design of the multiplier circuit to an unsigned multiplier. For these reasons, we decided to use this approach for the signed divider and also configure it to handle the inputs for the multiplier. This decision allowed us to design a combined multiplier/divider circuit that will be the “M” Extension to the core.

4.3.2 Design of the MULDIV circuit

The input and output circuits were used as components together with a multiplier and divider circuit on a newly designed top module to achieve a signed multiplier/divider circuit. We called this circuit the “MULDIV”. The multiplier used in this design was the renewed version of the initial signed multiplier design, which uses the same algorithm without the addition of a sign bit.

The diagram of the design for the MULDIV is given in Figure 4.6.

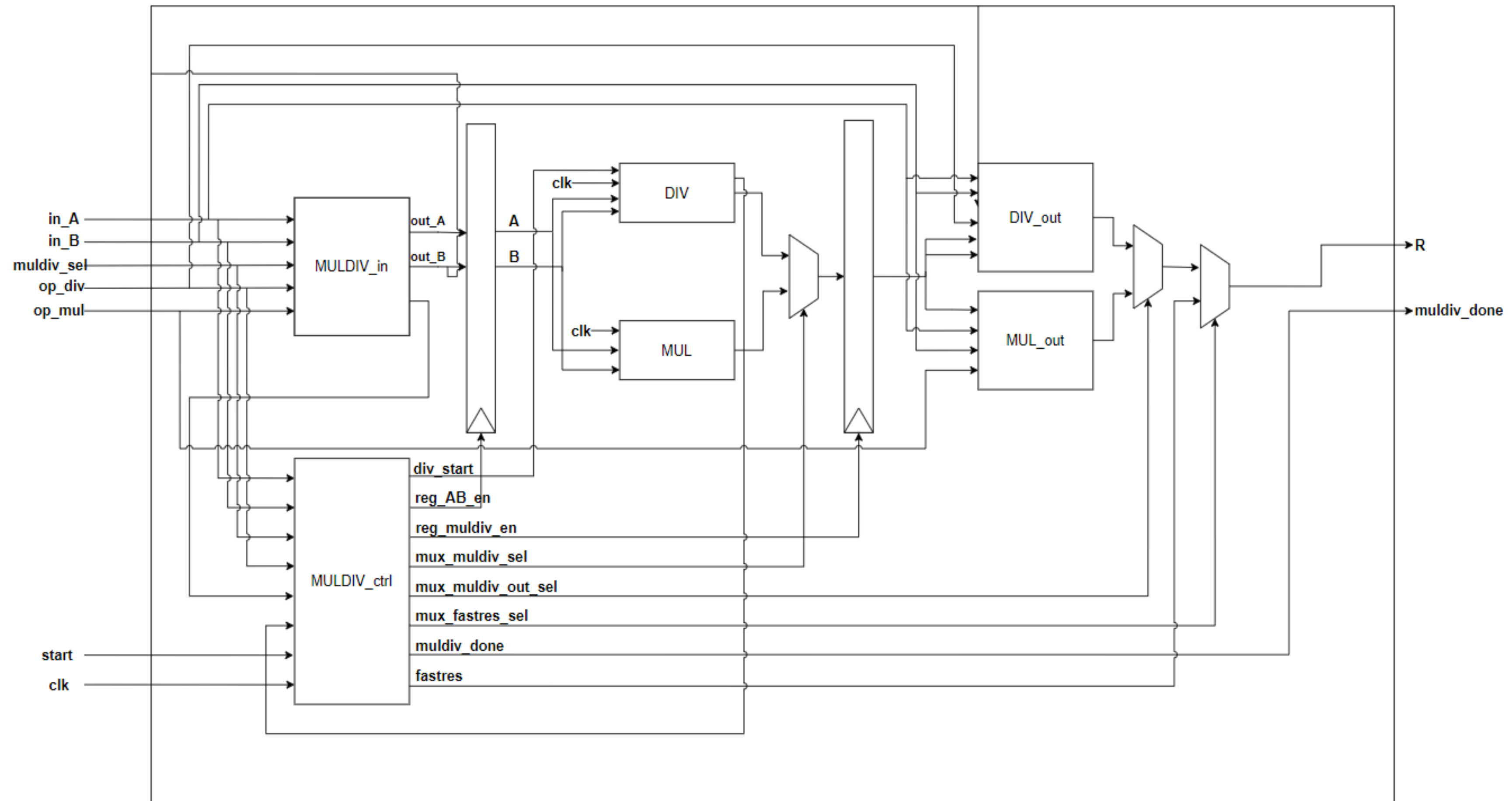


Figure 4.6 : Circuit diagram of the MULDIV

As it can be seen from the figure, the input adjuster is the MULDIV_in block and there are output adjusters DIV_out and MUL_out for the divider and multiplier, respectively. Also there are multiplexers and registers, that are controlled by the Control Unit. The Control Unit MULDIV_ctrl again is an ASM, and it realizes the designed state diagram for the multiplication and division operations.

4.3.2.1 Input and output adjuster circuits

Input adjuster circuit MULDIV_in checks if the current instruction is a signed one, namely MUL, MULH, MULHSU (only for the first operand, in_A), DIV or REM. After the operands are checked whether they are negative or positive, the MSB of the operands reveal their sign; so if it's logic "1", the number is negative and vice versa. Input adjuster circuits then need to convert the negative operands to positive numbers that will be used in the multiplier or the divider. It is done by taking the "two's complement" of determined operands. After converting required operands, the new operands are outputted to the rest of the circuit. The MULDIV_in circuit also checks if either of the operands are "0", "1" or "-1" and expresses the result in the form of a 6-bit output signal "AB_status". Figure 4.7 shows the bit arrangement of the signal "AB_status".

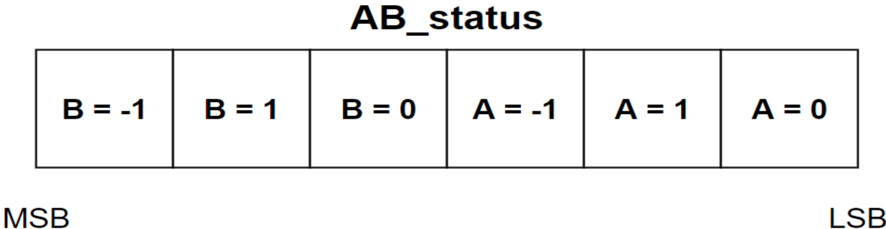


Figure 4.7 : Bit arrangement of the "AB_status"

The bits in the "AB_status" arranged to become 1 if the condition that they represent is true. This signal will be used in the control unit of the circuit, MULDIV_ctrl, for enabling a faster outputting of the result if one of the operands is "-1", "1" or "0". An output generated in one cycle is possible if at least one of the operands are "-1", "1" or "0", because these values are neutral or absorbing elements of multiplication and division. More detailed information will be given there.

Output adjuster circuit, DIV_out and MUL_out, changes the result to negative if necessary. This is done by checking the operation and the sign of the operands, similar to the input adjuster circuit.

4.3.2.2 Control unit

The Control Unit of the MULDIV, MULDIV_ctrl is responsible for the correct operation of the rest of the circuit. The State Diagram that the MULDIV_ctrl realizes is given in Figure 4.8. The states and transitions are designed so that the input adjuster circuits work on the inputs first then transfer them to the divider and multiplier circuits, and after the result is calculated, the output adjuster circuits work on the result and the correct the output from DIV_out or MUL_out is selected as the final result with a multiplexer.

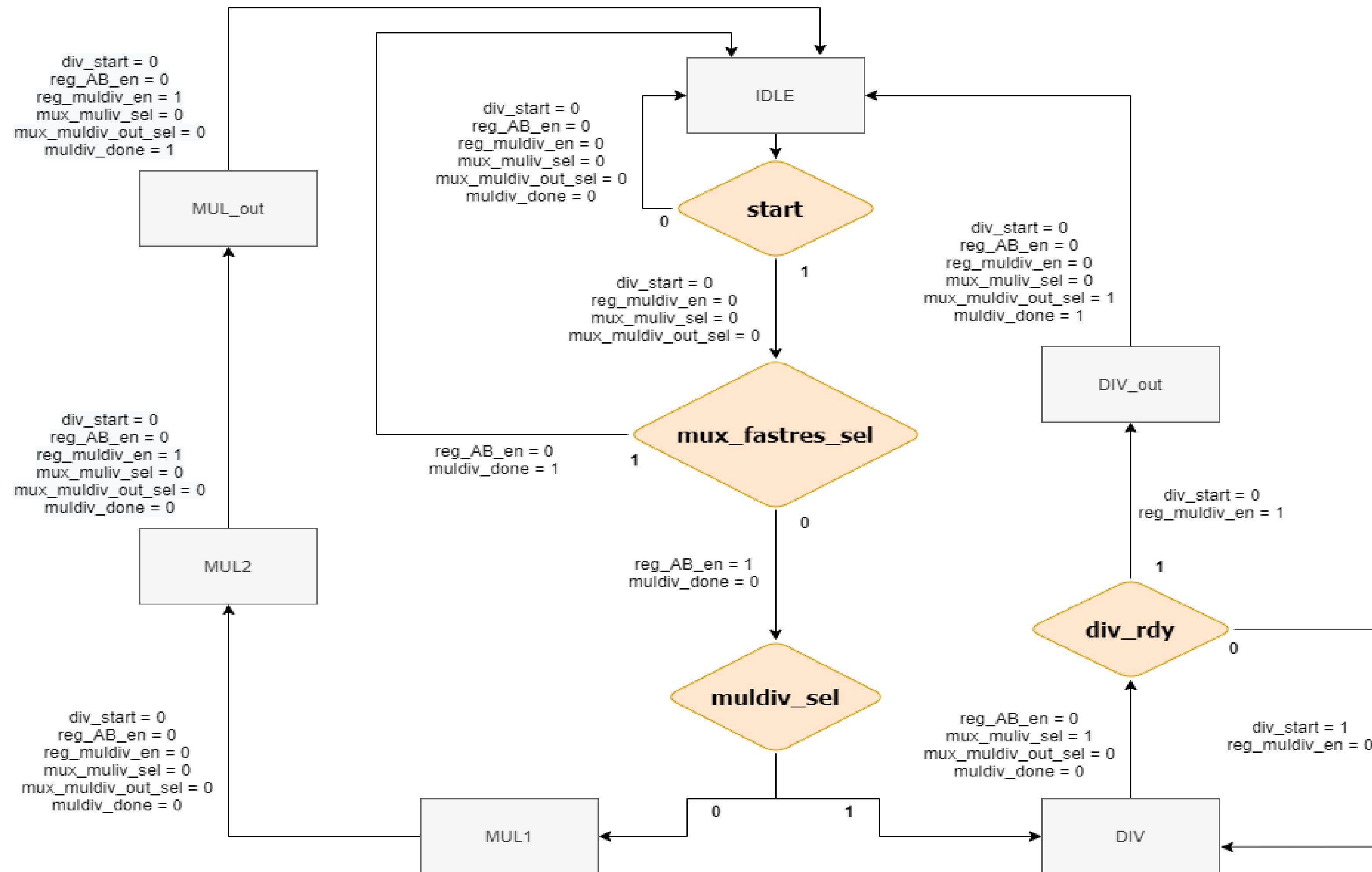


Figure 4.8 : State diagram of the MULDIV

At the state “IDLE”, the signal “mux_fastres_sel” is checked. This signal and another signal “fastres” is generated in the “MULDIV_ctrl” after examining the “AB_status” signal that is produced in the “MULDIV_in”. “fastres” is a 32-bit signal and it is determined together with the “mux_fastres_sel” by the fast result conditions, given as a schematic in the Figure 4.9. For example, if “AB_status” is “000100”, meaning the operand A is “-1”, a fast result is possible because the result will be “-B” if the instruction is “MUL”, “11111111” if the instruction is “MULH”, “MULHSU” “MULHU”, “REM” or “REMU”, and “0” if the instruction is “DIV” or “DIVU”. “mux_fastres_sel” is then used in the MULDIV for selecting the “fastres” as the output.

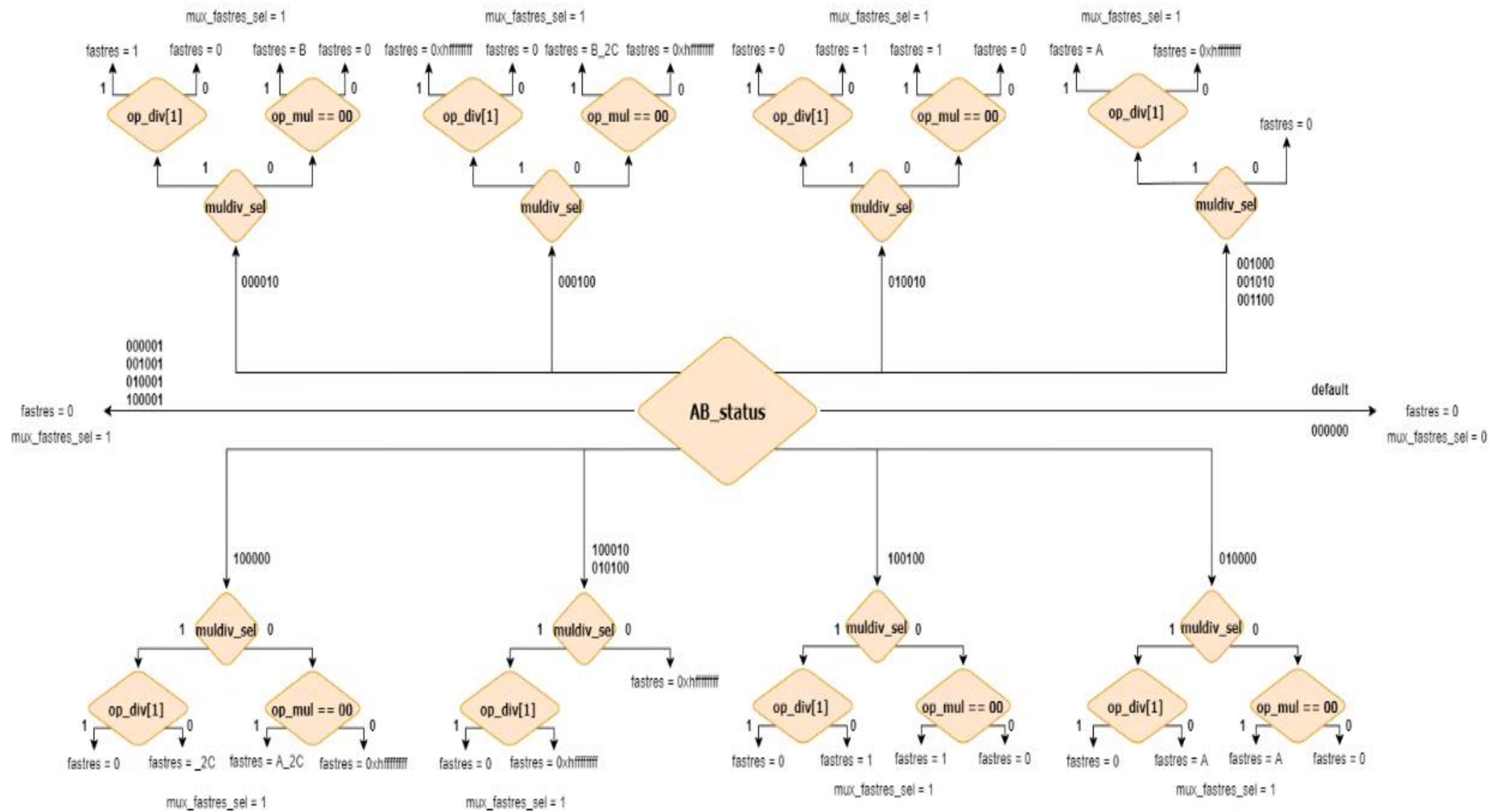


Figure 4.9 : Fast result conditions and output arrangements

4.3.3 Testing of the MULDIV module

After MULDIV was constructed, its functionality firstly needed to be tested. The testing of a module with that size is a challenge, and it is an entire topic on its own. We reviewed our options, from the ones that are more accurate and acceptable but complex and time consuming, to the ones that are very primitive but relatively simple.

We first decided to use the programming language Python, and a library constructed for the testing of HDL designs called "cocotb"[9]. With the help of this library, it was possible to create a testing environment written in Python, which is a very simple language, and using a basic simulation program like Icarus Verilog. We started to create the environment, but tackled upon problems with the simulation program Icarus Verilog that we weren't able to resolve. So, also considering the time we need to spend on it, we chose not to use it.

After the failure with the decent choice of using an established testing environment, we decided to use the Verilog language to write a testbench and test the circuit several times with the random inputs for A and B. This option was still acceptable, because a random number generator was available for Verilog, and our circuit was simple enough in terms of input and outputs. We wrote a testbench that gives random inputs between 0 and $2^{32} - 1$, and checks the output for each instruction, namely "MUL", "MULH", "MULHU", "MULHSU", "DIV", "DIVU", "REM", and "REMU", and it broadcasts a message if the testbench found an error. We can adjust how many test cases there will be for each instruction. With the help of the fast computers at the VLSI Lab in our faculty, we tested each instruction with 200,000 random inputs. The testbench doesn't examine the corner cases, which are the cases where one or both of the operands are the smallest or the biggest number possible, specifically "0" or "2^{32} - 1". So, we tested the circuit for these corner cases separately in another testbench, with smaller test cases.

After completing the test with zero errors, we were confident enough of the circuit's functionality to move on.

4.3.4 Integration to the core

After the circuit for the whole "M" extension was finalized, it was now the time for integrating the circuit to the rest of the core. Challenges for the integration process was

modifying the control unit of the core for the control signals of the MULDIV, proper handling of forwarding and hazard detection for the "M" extension instructions, and required stalling. These challenges except stalling are trivial, because the MULDIV circuit is similar to the ALU, so arranging the control signals other than the ones specific for MULDIV, and organizing the forwarding and hazard detection is the same for the instructions that use the ALU. And for the same reason, the data inputs of MULDIV are the same with the ALU, and data output from MULDIV combines with the one from the ALU with a multiplexer and connects to the pipeline.

The MULDIV circuit prepares the result in 4 clock cycles for the multiplication and 16 for the division, and it can output the result in 1 clock cycle if a fast result condition is met, as it was designed and shown in the state diagram in Figure 4.9. So, if the result is not prepared in a clock cycle, the rest of the pipeline needs to be stalled. This stalling condition is determined with the signal "muldiv_stall", that is generated as shown in Figure 4.10.

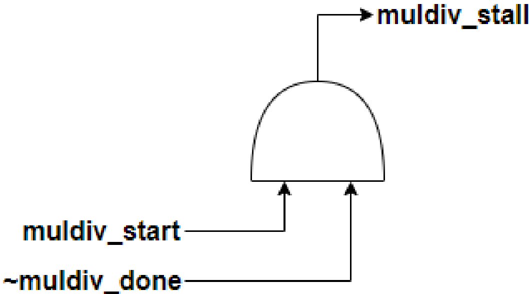


Figure 4.10 : Generation of the "muldiv_stall" signal

The "muldiv_stall" signal becomes logic "1" if an "M" extension instruction was in progress in the execution stage, meaning the "muldiv_start" is logic "1", and the signal that indicates that the calculation in the "MULDIV" is finished, that is "muldiv_done" is logic "0". This signal is then used in the top module of the core to stall the Instruction Fetch and Decode stages of the pipeline.

With these modifications done, the "M" Extension is now added to the core, and it is ready for the testing and the benchmarking. Figure 4.11 shows the updated diagram with the addition of the MULDIV block.

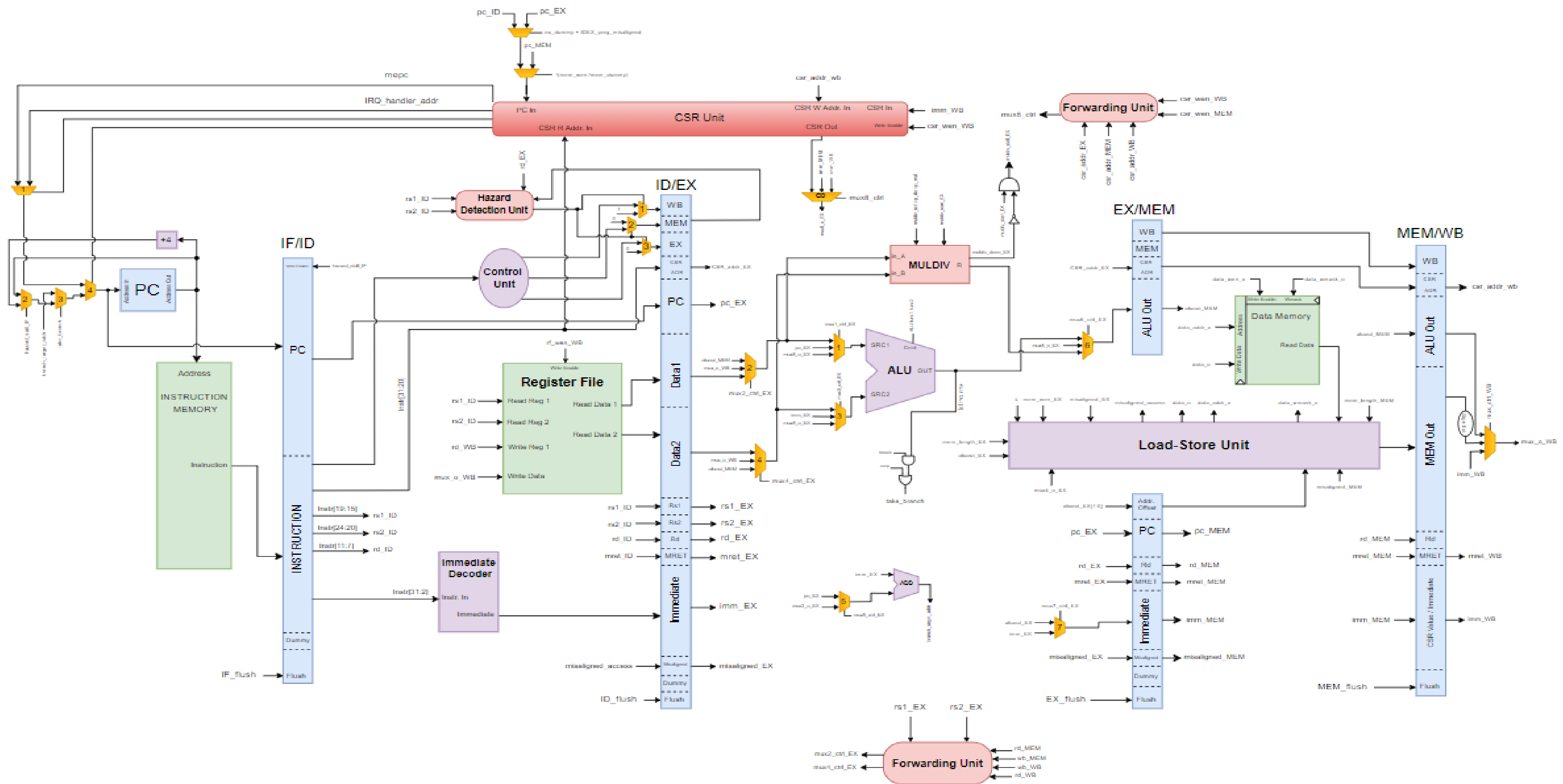


Figure 4.11 : Updated pipeline diagram with MULDIV block

5. TESTING THE DESIGN

Verifying the functionality of a microprocessor is no trivial task. In the case of RISC-V, a formal verification procedure is defined[10]. This procedure involves the implementation of what is called a “formal interface”. Through this interface, a number of relevant signals are monitored while the core executes instructions. The signals are then checked if they are correct or not.

Even though this formal verification process guarantees the functionality of the design, it is tedious and time consuming. Therefore, we took a different path for the verification task. We wrote several programs in C language, and ran them on the core. The test programs are quite comprehensive, but it is not guaranteed that every single instruction is tested, nor that all existing bugs are exposed. However, it is worth noting that we detected and fixed dozens of bugs thanks to this testing scheme. We also check the functionality of interrupts through some of these programs.

The testing procedure consists of the following steps,

1. Development of test programs in C
2. Compilation of the test programs using the RISC-V GNU Toolchain[11]
3. Execution of the program in a simulator environment
4. Analysis of the signals and results

Bugs are detected in the last step, and the design is fixed until no bugs are left. The details of the testing procedure are described in the following sections.

5.1 Setting up the environment for testing

The test programs can be written in any text editor. However, the compilation process requires the RISC-V GNU Toolchain, which is only available on a Linux environment.

5.1.1 Installing Ubuntu Linux operating system

A very popular Linux distribution is Ubuntu[12]. There are several ways to install Ubuntu on a machine. Typically, it is setup as a virtual machine on top of a Windows operating system, which is what we did in our case. Obviously, it can also be installed as the main operating system, or a dual boot configuration can be utilized.

To setup a Ubuntu virtual machine, Oracle’s VirtualBox software can be used[13]. Once the setup is complete, and the virtual machine is ready, it is time to setup the environment.

5.1.2 Installing the toolchain

In Linux operating systems, most software is installed and used through a command line. Such a command line can be access through the Terminal application that the operating system provides. The user can issue the commands through this interface.

In order to install the RISC-V GNU Toolchain on the system, first, the repository must be cloned via the following command,

- `git clone https://github.com/riscv/riscv-gnu-toolchain`

The toolchain requires a bunch of prerequisite software, which can be installed with the following command,

- `sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev`

Once the prerequisites are installed, current directory should be changed using the following command,

- `cd riscv-gnu-toolchain`

Now, the setup must be configured. The following command should do that,

- `./configure –-prefix=/opt/riscv --with-multilib-generator="rv32i-ilp32--;rv32im-ilp32--"`

The first part of the command that starts with “--prefix" sets the installation directory. the second part that starts with “--with" configures the installer so that both RV32I and RV32IM libraries are installed. This is convenient because it allows the user to force the compiler to use only RV32I instructions or RV32IM instructions. If we did not install both of them, the compiler would always use RV32IM (or RV32I) instructions, even if we compiled for RV32I (or RV32IM).

Finally, the toolchain can be installed by executing the following command,

- `sudo make`

The installation may take roughly an hour, beware. Once the installation is complete the installation directory must be added to the `PATH`. In order to do that, we simply need to modify a text file. To open that text file, following command should be executed,

- `sudo nano /etc/environment`

This will open the text file on the Terminal. Now, simply add the following text to the end of the file,

- `:/opt/riscv/bin`

Now press CTRL+O to write the changes to file and hit Enter. Press CTRL+X to exit. You should be able to invoke the RISC-V compiler now.

5.1.3 Installing Verilator

Verilator[14] is a free and open-source software that essentially simulates Verilog designs. It generates a C++ class from the Verilog module, which in turn can be instantiated in a C++ program. As Verilator can be invoked from the command line, the simulation process can be scripted. In our project, we used this software for automated testing. However, this software is not strictly needed, as other simulators can also be used.

Executing the following command should install Verilator on your system,

- `sudo apt-get install verilator`

You should now be able to invoke Verilator from your command line.

5.1.4 Installing GTKWave

GTKWave[15] is another free and open-source software that can be used to view simulation waveforms. It simply visualizes the waveforms generated by a simulator. GTKWave can be used in conjunction with Verilator. Verilator can be invoked with the necessary options to generate a waveform dump file. GTKWave can generate waveform views from that file. Again, this software is not strictly necessary, but it can be convenient for debugging. Following command should install the software,

- `sudo apt-get install gtkwave`

You should now be able to invoke GTKWave.

5.2 Compiling a RISC-V program

We now have everything ready to compile a RISC-V program. Compilation is the process of generating a binary file from a C program. With Verilator, the binary file alone is enough for simulation. However, with real Verilog simulators, a memory file must be generated from the binary file. This memory file can be loaded into the memory of the processor in the testbench. In any case, the first step is to generate the binary file.

Let us work on the bubble sort program provided in the Github repository of this project. First, clone the repository with the following command,

- `git clone https://github.com/yavuz650/RISC-V.git`

Change directory to the bubble_sort folder,

- `cd RISC-V/test/bubble_sort`

As the name implies, bubble sort is a simple program that sorts an array using the bubble sort algorithm. Of course, the source code is available in the `bubble_sort.c` file.

Now, we need to compile this program into an Executable and Linkable File format, i.e. a .elf file. Following command should do this task,

- `riscv64-unknown-elf-gcc bubble_sort.c ../crt0.s –march=rv32i –mabi=ilp32 –T ../linksc.ld –nostartfiles –ffunction-sections –fdata-sections –Wl,--gc sections –o bubble_sort.elf`

This command will generate the `bubble_sort.elf` file. Let us breakdown this verbose command,

- `riscv64-unknown-elf-gcc:` This invokes the GNU C compiler.
- `bubble_sort.c:` This is the source file
- `../crt0.s:` This is an assembly code that performs the initialization routine. It simply initialies the stack pointer and jumps to the main function.
- `-march=rv32i:` Specifies the target ISA
- `-mabi=ilp32:` Specifies the Application Binary Interface

- `-T ../linksc.ld:` Specifies the linker script. A linker script defines the memory regions of the processor. In this case, it has two regions: ROM and RAM. ROM is where the instructions and the read-only data is stored. RAM is where the run-time variables are stored. The linker uses this script to place the instructions at appropriate memory regions.
- `–nostartfiles –ffunction-sections –fdata-sections –Wl,--gc sections:` These options are used to strip the elf file from unused and redundant code.
- `-o bubble_sort.elf:` Specifies the name of the output file.

Now, we need to generate the binary file from this elf file. Following command should do that,

- `riscv64-unknown-elf-objcopy –O binary –j .init –j .text –j .rodata bubble_sort.elf bubble_sort.bin`

This command copies the .init, .text and .rodata sections from the elf file and generates the binary file.

The program is now compiled successfully. One last step before simulating is to generate the .data file which will be loaded into the processor's memory. To do that, we have written a simple C program which is also available in the repository. This program accepts a binary file as an input, and generates a .data file that contains the instruction opcodes, and the read-only data. This .data file can be loaded into the memory of the processor using the `$readmem` command in Verilog.

Figure 5.1 shows the C code of this program,

```c
int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("Error! Expected a .bin file only.\n");
        return -1;
    }
    char cmd[512];
    FILE *infile, *outfile;
    int len;
    uint32_t rom[4096];

    infile = fopen(argv[1], "rb");
    fseek(infile, 0, SEEK_END);
    len = ftell(infile);
    fseek(infile, 0, SEEK_SET);

    if (fread(rom, 4, len/4, infile) != len/4)
    {
        printf("Assembled file read error!\n");
        fclose(infile);
        return -1;
    }

    char file_name[64];
    sprintf(file_name,"%s",argv[1]);
    sprintf(file_name+strlen(argv[1])-4,".data");

    outfile = fopen(file_name,"wb");
    for (int i = 0; i < len/4; i+=1)
    {
        sprintf(cmd,"%08X",rom[i]);
        fwrite(cmd,sizeof(char),strlen(cmd),outfile);
        sprintf(cmd,"\n");
        fwrite(cmd,sizeof(char),1,outfile);
    }
    return 0;
}
```

Figure 5.1 : C code of the ROM generator program

Following sequence of commands can be used to compile this program,

- `cd ..`

  `g++ rom_generator.c -o rom_generator`

  `cd bubble_sort`

Now use the following command to generate the .data file,

- `../rom_generator bubble_sort.bin`

All the files are now prepared for simulation.

In practice, these steps are executed by a makefile. A makefile is basically a script that is used to build programs. Such makefiles are provided in each test folder. Then, instead of executing all the commands above, makefile can be run to do all of them at once. Simply running the command `make` should be enough to invoke the makefile. An example makefile is shown in Appendix B.

5.3 Simulating a RISC-V program using the core

5.3.1 Developing an SoC for simulation

Before we can simulate the core, we need to attach the necessary peripherals to make it work. For simulation purposes, a dual-port memory is enough. We simply combine the core with the memory on a top module. This top module becomes a system-on-chip that can simulate RISC-V binaries. In the repository, an SoC named "`barebones`" is provided for this purpose.

5.3.2 Simulating with Cadence Xcelium

Xcelium is a commercial RTL compiler and simulator developed by Cadence Design Systems. As such, this tool is not freely available to the public. In the scope of this project, we had access to this software through the VLSI Lab at our faculty.

To simulate a program with Xcelium, we must first compile the Verilog source files. Executing the following command while in the barebones directory should compile the necessary source files,

- ```
  xmvlog ../../core/core.v ../../core/ALU.v ../../core/control_unit.v
  ../../core/forwarding_unit.v ../../core/hazard_detection_unit.v

  ../../core/imm_decoder.v ../../core/load_store_unit.v
  ../../core/csr_unit.v \

  ../../peripherals/memory_2rw.v ../../peripherals/mtime_registers.v \

  ../../core/muldiv/divider_32.v ../../core/muldiv/multiplier_32.v
  ../../core/muldiv/MULDIV_ctrl.v ../../core/muldiv/MULDIV_in.v \

  ../../core/muldiv/MUL_DIV_out.v ../../core/muldiv/MULDIV_top.v
  ../../peripherals/debug_interface.v \

  barebones_top.v barebones_top_tb.v
  ```

Then, following command should elaborate the testbench,

- `xmelab –access rwc barebones_top_tb`

Now, the simulation can be launched with the following command,

- `xmsim –gui barebones_top_tb`
  ```

5.3.3 Simulating with Verilator

Simulating with Verilator is quite different than simulating with a traditional RTL simulator. For Verilator, instead of a Verilog testbench, a C++ wrapper file is required. This wrapper file acts like the testbench. In it, an object of the generated C++ class is instantiated. Through the interface of that class, the signals are driven and read. Figure 5.2 shows the wrapper code for bubble sort,

```
Vbarebones_top *barebones_top;
vluint64_t main_time = 0;

int main(int argc, char** argv)
{
    std::ifstream bin_file("../bubble_sort.bin",std::ifstream::binary);
    Verilated::commandArgs(argc, argv);
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    barebones_top = new Vbarebones_top;

    bin_file.seekg(0,bin_file.end);
    int len = bin_file.tellg();
    bin_file.seekg(0,bin_file.beg);
    bin_file.read(reinterpret_cast<char*>(barebones_top->barebones_top->memory->mem),len);

    barebones_top->trace(tfp, 99);
    tfp->open("simx.vcd");
    barebones_top->reset_i = 0;
    while (!Verilated::gotFinish())
    {
        if (main_time > 10) {
            barebones_top->reset_i = 1;
        }
        if ((main_time % 10) == 1) {
            barebones_top->clk_i = 1;
        }
        if ((main_time % 10) == 6) {
            barebones_top->clk_i = 0;
        }
        barebones_top->eval();
        tfp->dump(main_time);
        main_time++;
        if(main_time > 50000)
        {
            std::cout << "Failure - Time out...\n";
            break;
        }
    }

    barebones_top->final();
    tfp->close();
    delete barebones_top;
}
```

Figure 5.2 : C++ wrapper file for bubble sort

This wrapper file instantiates an object of `VBarebones_top` class, which is generated by Verilator based on the Verilog design files. In the while loop, the signals are read and driven, much like a Verilog testbench. Once the wrapper file is ready, Verilator can be invoked to generate the necessary files. While in the bubble_sort directory,

following command can be executed to invoke a script that compiles and runs the Verilog designs using Verilator,

- `./bubble_sort_script`

Once the simulation completes, waveforms can be viewed using GTKWave. To do that, first change the directory using the following command,

- `cd obj_dir`

Then, launch GTKWave,

- `gtkwave simx.vcd`

You should now be able to view waveforms.

6. BENCHMARKS AND POST-SYNTHESIS ANALYSIS

6.1 Benchmarks

In addition to tests, we also developed simple benchmarks to quantify the advantage of having the "M" standard extension for multiply/divide operations. We developed 4 benchmarks; 64-bit integer multiplication and division, 64-bit floating-point multiplication and division. As RV32IM does not have native instructions for 64-bit division and floating point operations, they are emulated on software.

Figures 6.1 to 6.4 show the benchmark programs,

```
int main()
{

    volatile int32_t a = 616959;
    volatile int32_t b = 902274;
    volatile int64_t c; //556666064766
    c = (int64_t)a*b;

    return 0;
}
```

Figure 6.1 : 64-bit integer multiplication code

```
int main()
{

    volatile int64_t a = 556666064766;
    volatile int32_t b = 616959;
    volatile int32_t c; //902274
    c = a/b;

    return 0;
}
```

Figure 6.2 : 64-bit integer division code

```
int main()
{

    volatile double a = 0.06237;
    volatile double b = 0.57238;
    volatile double c; //0.0356993406
    c = a*b;

    return 0;
}
```

Figure 6.3 : 64-bit floating-point multiplication code

```
int main()
{

    volatile double a = 0.0356993406;
    volatile double b = 0.06237;
    volatile double c; // 0.57238
    c = a/b;

    return 0;
}
```

Figure 6.4 : 64-bit floating-point division code

All benchmark programs start with a stack initialization routine, then proceed to the initialization of variables. These initialization routines are common to all programs, therefore the effects on results will be the same. Then, the arithmetic operation is performed. A benchmark ends when the result is stored to the memory.

Each program is compiled with the highest optimization level, `-O3`, for both RV32I and RV32IM instruction sets. The programs are simulated on Cadence Xcelium Simulator, and the number of clock cycles are determined through the waveform window. Table 6.1 summarizes the results,

Table 6.1 : Benchmark results

Benchmark \ Instruction Set	RV32I	RV32IM
64-bit Integer Mul.	**311**	**28**
64-bit Integer Div.	**972**	**183**
64-bit Floating-point Mul.	**2241**	**291**
64-bit Floating-point Div.	**3489**	**448**

6.2 The critical path

In a clocked digital circuit, the critical path refers to the path which starts and ends with a register, and has the longest propagation delay. This path is called the critical path because it determines the maximum frequency of operation of the circuit.

In a typical design flow, the critical path is determined after synthesis. In this project, Genus synthesis tool is used to generate a timing report which includes all the details of the critical path.

We examined the critical path in two configurations: with and without the multiply/divide unit.

Without the multiply/divide unit, the critical path turned out to be between a CSR and a pipeline register. More specifically, the startpoint is the `mstatus` CSR, and the endpoint is the pipeline register between ID and EX stages. The main reason why this path is critical is because the `mstatus` register is written on the falling edge of the clock, as is the case with all the other registers. Moreover, the important detail about this register is that it contains the global interrupt enable bit, which is used in the calculation of the pipeline flush signals. The calculation of the pipeline flush signals involves several cascaded logic operations, which result in a long propagation delay. Combining this with the fact that the path has half a clock cycle to complete, it is natural that this path turned out to be the critical path.

The reason why we designed the registers so that they are written on the falling edge is because it is another form of forwarding data. If the registers were written on the rising edge, then there would have to be additional pipeline stalls on occasions. In

retrospect, this was a poor design choice by us. Because, such stalls would be extremely rare, which means such a tight critical path is not worth it.

After the incorporation of the multiply/divide unit, the critical path turned out to be in the division path. This was expected because the division algorithm initially involved 4 32-bit cascaded subtractions. At that time, a division operation took 8 cycles. After that, we changed the algorithm so that it completed in 16 cycles, and involved 2 32-bit cascaded subtractions. After this change, the critical path changed back to what it was before. Figures 6.5 to 6.7 show the timing reports,

```
Path 1: MET (922 ps) Setup Check with Pin core0_IDEX_preg_data2_reg[28]/CP->D
          Group: clkin1
     Startpoint: (R) core0_CSR_UNIT_mstatus_reg[3]/CP
          Clock: (F) clkin1
       Endpoint: (R) core0_IDEX_preg_data2_reg[28]/D
          Clock: (R) clkin1
```

Figure 6.5 : Critical path without multiply/divide unit

```
Path 1: MET (131 ps) Setup Check with Pin core0_MULDIV_DIV_reg_R_reg[30]/CP->D
          Group: clkin1
     Startpoint: (R) core0_MULDIV_DIV_div_control_current_state_reg[2]/CP
          Clock: (R) clkin1
       Endpoint: (R) core0_MULDIV_DIV_reg_R_reg[30]/D
          Clock: (R) clkin1
```

Figure 6.6 : Critical path after adding multiply/divide unit

```
Path 1: MET (1717 ps) Setup Check with Pin core0_IDEX_preg_data2_reg[11]/CP->D
          Group: clkin1
     Startpoint: (R) core0_CSR_UNIT_mstatus_reg[3]/CP
          Clock: (F) clkin1
       Endpoint: (R) core0_IDEX_preg_data2_reg[11]/D
          Clock: (R) clkin1
```

Figure 6.7 : Critical path after changing division to 16 cycles

6.3 Area

We also compared the change in the consumed area. Table 6.2 summarizes the results,

Table 6.2 : Area consumption

	RV32I	RV32IM
Area (um^2)	106768	175450
Gate Count	7969	13203

As expected, addition of the multiply/divide unit incurs significant area consumption. It is worth noting that we used TSMC's 90nm general purpose ASIC cell library for synthesis.

7. REALISTIC CONSTRAINTS AND CONCLUSIONS

Open-source hardware platforms are growing bigger every day. In this project, we contributed to this growth by developing a well-documented, open-source RISC-V core. The design is free to use and extend. It can be used for research purposes, educational purposes, and even for personal projects.

7.1 Practical Application of this Project

In the simplest case, the design can be used for further research. Moreover, it can be extended and prepared for a chip tape-out. This could be an important contribution to the processor design initiative that is present in the country.

7.2 Realistic Constraints

7.2.1 Social, environmental and economic impact

RISC-V is a license-free ISA. This means that companies or groups do not have to pay for a licensing fee to produce and/or sell RISC-V processors.

7.2.2 Cost analysis

The CAD tools used in this project are not free. Simulation and synthesis tools are quite costly. Fortunately, VLSI Lab in our faculty provided us with the necessary software.

7.2.3 Standards

The only standard to follow in this project is the RISC-V ISA manual.

7.2.4 Health and safety concerns

This project does not involve health and safety concerns.

7.3 Future Work and Recommendations

There is an incredible amount of room for future work. To begin with, the remaining standard extensions, such as the floating-point instruction set, can be implemented. Supervisor mode and virtual memory can be implemented to allow for operating systems. Instruction and Data caches can be implemented. An interface can be provided for communication with peripherals. Last but not least, chip tape-outs can also be done.

REFERENCES

[1] “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213”, Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.

[2] “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified”, Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, June 2019.

[3] Patterson D. and Hennessy John L., *Computer Organization and Design RISC-V Edition,* 2nd ed. Morgan Kauffman, 2020.

[4] “RISC-V Organization”, https://riscv.org

[5] "IEEE Standard for Verilog Hardware Description Language," in *IEEE Std 1364-2005* (Revision of IEEE Std 1364-2001), vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[6] "IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process," *in IEEE Std 1497-2001*, vol., no., pp.1-80, 14 Dec. 2001, doi: 10.1109/IEEESTD.2001.93359.

[7] B. Parhami, *Computer Arithmetic and Hardware Designs*, Oxford University Press, 2000.

[8] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed, A. K. Peters/CRC Press, 2002.

[9] Url-1 <*https://github.com/cocotb/cocotb*>*,* access date 15.06.2021

[10] Url-2 <*https://github.com/SymbioticEDA/riscv-formal*>*,* access date 15.06.2021

[11] Url-3 <*https://github.com/riscv/riscv-gnu-toolchain*>*,* access date 15.06.2021

[12] Url-4 <*https://ubuntu.com/*>*,* access date 15.06.2021

[13] Url-5 <*https://www.virtualbox.org/*>*,* access date 15.06.2021

[14] Url-6 <*https://www.veripool.org/verilator/*>*,* access date 15.06.2021

[15] Url-7 <*http://gtkwave.sourceforge.net/*>*,* access date 15.06.2021

APPENDICES

APPENDIX A: Genus synthesis script
APPENDIX B: Example makefile

APPENDIX A

```
set_db library {/vlsi/kits/tsmc/lib/90gp/TSMCHOME/digital/Front_End/timing_power_noise/NLDM/tcbn90gbwp14t_220a/tcbn90gbwp14ttc.lib}

set_db information_level 11
set_db delete_unloaded_insts false
set_db delete_unloaded_seqs false

# Reading Verilog Codes and Elaborating
read_hdl -v2001 {barebones_top.v ../../core/core.v ../../core/ALU.v ../../core/control_unit.v ../../core/forwarding_unit.v ../../core/hazard_detection_unit.v \
../../core/imm_decoder.v ../../core/load_store_unit.v ../../core/csr_unit.v ../../peripherals/mtime_registers.v}
elaborate barebones_top

# Defining Time Constraints - ns
create_clock -period 10 -name clkin1 -domain domain_1 clk_i
set_clock_latency -max 1 clkin1
# CLOCK UNCERTAINTY (JITTER)
set_clock_uncertainty -setup 1 clkin1
set_clock_uncertainty -hold 1 clkin1
# DELAY FROM INPUT PIN TO CLOCK
set_input_delay -clock clkin1 -clock_rise 1 [all_inputs]
set_output_delay -clock clkin1 -clock_rise 1 [all_outputs]

set_driving_cell reset_i meip_i -cell BUFFPD12BWP14T
set_load 1 irq_ack_o

# Synthesizing
#set_db operating_conditions BCCOM
set_db syn_generic_effort high
set_db syn_map_effort high
set_db syn_opt_effort high
syn_generic barebones_top
syn_map barebones_top
syn_opt barebones_top


# Writing Report Files
report timing > report_time.txt
report gates > report_gates.txt
report area > report_area.txt

# Writing Design Files
write_hdl barebones_top -language v2001 > syn_barebones_top.v
write_sdf -edges check_edge -design barebones_top > barebones_top.sdf
```

Figure A.1 : Synthesis script for Genus

APPENDIX B

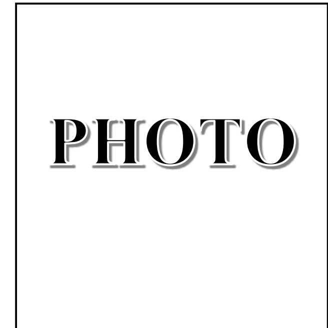
```
CC32=riscv32-unknown-elf
CC64=riscv64-unknown-elf
CCFLAGS=-march=rv32i -mabi=ilp32 -O3 -T ../linksc.ld -nostartfiles -ffunction-sections -fdata-sections -Wl,--gc-sections -o bubble_sort.elf

build:
	$(CC32)-gcc bubble_sort.c ../crt0.s $(CCFLAGS)
	$(CC32)-objcopy -O binary -j .init -j .text -j .rodata bubble_sort.elf bubble_sort.bin
	../rom_generator bubble_sort.bin
	cp bubble_sort.data ../memory_contents

multilib:
	$(CC64)-gcc bubble_sort.c ../crt0.s $(CCFLAGS)
	$(CC64)-objcopy -O binary -j .init -j .text -j .rodata bubble_sort.elf bubble_sort.bin
	../rom_generator bubble_sort.bin
	cp bubble_sort.data ../memory_contents
```

Figure B.1 : Example makefile script

CURRICULUM VITAE



Name Surname :

Place and Date of Birth :

E-Mail :

The details fort the CV of the project group member. Repeat this page for all project group members.