**RISC–V BASE ISA PROCESSOR**

Undergraduate graduation project report submitted in partial fulfillment of the requirements for the

Degree of Bachelor of Science of Engineering

in

The Department of Electronic & Telecommunication Engineering

University of Moratuwa.

Supervisor: Group Members:

Dr. Ajith Pasquel H.A.R.T Wijesekara 120715P

D.M.Y.N Seneviratne 120602R

H.M.N Perera 120465X

M.M.G.U.M Thilakasiri 120660P

April 2017

Approval of the Department of Electronic & Telecommunication Engineering

……………………………………………….

Head, Department of Electronic & Telecommunication Engineering

This is to certify that I/we have read this project and that in my/our opinion it is fully adequate, in scope and quality, as an Undergraduate Graduation Project.

Supervisor: Dr. Ajith Pasquel

Signature: …………………………………

Date: ……………………………………….

# Declaration

This declaration is made on April 1, 2017.

**Declaration by Project Group**

We declare that the dissertation entitled *RISC-V Base ISA Processor* and the work presented in it are our own. We confirm that:

* this work was done wholly or mainly in candidature for a B.Sc. Engineering degree at this university,
* where any part of this dissertation has previously been submitted for a degree or any other qualification at this university or any other institute, has been clearly stated,
* where we have consulted the published work of others, is always clearly attributed,
* where we have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely our own work,
* we have acknowledged all main sources of help

……………………….. ………………………………

Date H.A.R.T Wijesekara (120715P)

………………………………

D.M.Y.N Seneviratne (120602R)

………………………………

H.M.N Perera (120465X)

………………………………

M.M.G.U.M Thilakasiri (120660P)

**Declaration by Supervisor**

I have supervised and accepted this dissertation for the submission of the degree.

……………………………………….. ………………………………

Dr. Ajith Pasquel Date

# Abstract

**RISC-V BASE ISA PROCESSOR**

Group Members: H.A.R.T Wijesekara, D.M.Y.N Seneviratne, H.M.N Perera, M.M.G.U.M Thilakasiri

Supervisor: Dr. Ajith Pasquel

Keywords: RISC-V, processor, cache architecture

Advent of Internet of Things, Systems on Chips and embedded/mobile computing have made processing elements ubiquitous in current high tech products, which in turn has increased demand for a wide range of processors with different power-area-performance balance. Though there are several successful processors in today’s market, patents, closed sources and very high licensing fees have made these processors inaccessible to academia and small companies, stifling research and stagnating the growth in the field.

As a possible solution, UC Berkeley has proposed an open source ISA called RISC-V with the hope of rejuvenating research and development in processor architecture field and ultimately producing a family of efficient, free, open-source processors optimized for different applications. With the ISA itself avoiding many of the pitfalls observed in current commercial architectures, a well-designed implementation of RISC-V has the potential of attaining superior performance over other processors currently in the market. Increasing interest about RISC-V architecture in commercial and academic circles shows that such a RISC-V implementation would have significant success in the market.

Hence this project developed a performance-optimized implementation for the RISC-V ISA targeted towards FPGA and ASIC platforms. Due to the limited development time, only the minimal basic ISA of RISC-V was implemented, onto which further extensions could be carried out later. The processor was designed with an in-order, single-core, prefetch-enabled 10-stage pipeline, supported by a performance optimized, set-associative write-back cache architecture. Peripherals and main memory was connected to the processor through the AXI interface. The hardware architecture was developed, RTL coded and simulated using XSim before being implemented and emulated on a Xilinx FPGA. The final functionality was verified by running several RISC-V Base ISA assembly programs obtained using the toolchain provided by UC Berkeley. Preliminary resource utilization, frequency and performance parameters were compared with some of existing RISC-V implementations.

By analyzing the comparison results, it was seen that the designed RISC-V core had significantly lower resource utilization and higher operating frequency than some of the similar existing designs. As such it could be concluded that, with further improvements and extensions, the designed IP core could become a viable contender for the future processor market.

To our Dearest Parents,

Teachers

&

Friends

# Acknowledgments

We have taken much efforts in completing this final year project. However, it would not have been possible without the kind support and help of many individuals. Standing at the final milestone of an entire year of hard work, we would like to express our profound gratitude to all those who gave their best efforts to make this project a success.

We would like to thank Prof. Rohan Munasinghe, Head of the Department of Electronic and Telecommunication Engineering for giving us his fullest support. Especially we would recall with gratitude Dr. Ajith Pasqual, our project supervisor for his guidance and constant supervision as well as for encouraging us to successfully complete our project. A special thank you goes to Dr. Anjula C. De Silva, our project coordinator for his kind support throughout the project. We would like to acknowledge with much appreciation all the lecturers in the Department of Electronic and Telecommunication Engineering for their guidance and inputs in completing the project.

We are highly indebted to ParaQum Technologies for their guidance and support given to complete the project. Our thanks and appreciation also go to the non-academic staff who gave us their fullest support in the laboratories.

Finally we would like to thank all, whose names have not been mentioned above for their kind co-operation and encouragement which helped us in completion of the project successfully.

Table of Contents

Page

**1. Introduction** 1

1.1 Overview …………………………………………………………………………...1

1.2 Primary Objectives of the Project 3

1.3 Scope of the Project 4

1.4 Literature Review 5

**2. Hardware design and implementation** 7

2.1 System Overview 7

2.2 Main Pipeline 9

2.3 Cache Architecture 18

2.4 Memory Interfaces and Peripherals 30

**3. Results and Discussion** 33

3.1 Resource Utilization 34

3.2 Power Consumption and Frequency 35

3.3 Testing and Verification 36

3.4 Future Work and Improvements 37

**4.** **Conclusion** 40

**5.** **References** 42

**Appendix** 44

# List of Figures

2.1. [System Overview](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419296) 8

2.1. Main Pipeline 10

2.3. [Instruction Decode Stage](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419296) 12

2.4. [Immediate instruction types](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419297) 13

2.5. [Immediate types](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419298) 13

2.6. [Execution stage](#page23) 14

2.7. [Write back stage](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419300) 16

2.8. [Data feedback](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419301) 17

2.9. [Memory interface](file:///E:/Nimashani/Final_Year_Project/Documentation/Final%20Report/Final%20report%20version%202.docx%23_Toc471419301) 31

# List of Tables

Resource Utilization 35

Power Consumption 35

# Acronyms and Abbreviations

FPGA - Field Programmable Gate Array

ISA - Instruction Set Architecture

AXI - Advanced eXtensible Interface

SoC - System on Chip

HDL - Hardware Description Language

ID - Instruction Decode

IF - Instruction Fetch

EXE - Execution

DM - Data Memory

WB - Write Back

DMA - Direct Memory Access

# Chapter 1

INTRODUCTION

### 1.1 Overview

All along the past few decades electronics and computer technologies has been revolutionizing how people work, interact and live, and will continue to do so well into the future. One of the main driving forces of this revolution was the availability of faster, more efficient computers at cheaper and cheaper prices each passing year. At the heart of any computer system is the micro-processor, its electronic brain.

With emerging technologies like Internet of Things, Systems on Chips, Embedded / Mobile Computing, the traditional role of computers is changing from a stand-alone desktop device to a network of small computers embedded within every electronic device, managing sensors, processing data and controlling actuators. Therefore, the future requires a much wider range of processors, specializing towards different applications, and working with different power, chip area, computational performance budgets.

To cope with the difficult future requirements in the field, we need a vibrant, widespread research and development community both in the industry and in academia. Essential for this is the free dissemination of knowledge and techniques within the community. Past experience in the computer software field has shown the effectiveness of open source platforms (for example Linux, C) in stimulating rapid technological development. Sadly, the current processor industry lacks just that.

* Currently, a major share of PC and server processor industry is on the hands of a few giant co-operations such as Intel and AMD. Their most successful processor ISAs and micro-architectures are patent protected, closed source or trademarked, making them unavailable to outside parties.
* Embedded processor industry is similarly dominated by ARM Hld. Companies involved with embedded products often buy soft-core processors from ARM and integrate them with their products. However, these processors are very expensive due to their high licensing fees and is practically inaccessible to individual researchers, freelance coders, most of the academia and small/medium scale industries.
* Microprocessors such as MicroBlaze developed by Xilinx and Nios II developed by Altera for their FPGA families are only available for running on their own hardware platforms. This limits the portability and range of applications for these processors.

All these issues, combined, limits the design and implementation of processors to a few large companies, while completely excluding individual researchers, universities and small start-ups. Currently this has reduced the pace of processor development, with the major companies marketing the same processor architectures again and again with minor improvements. This has stifled the development of other fields too, like embedded systems and IoT, since good processors are not available at an affordable cost to small companies.

As a possible solution, UC Berkeley has introduced RISC-V, a completely free and open source ISA standard, which has been growing in popularity among the community in recent years. It incorporates many RISC style design strategies to provide a platform independent, clutter-free, simple, and efficient framework for building practically implementable processors. Arguably, the from-scratch design of RISC-V allows it to avoid many of the pitfalls and baggage sequentially developed ISAs (like x86 or AMD64) suffers from. Therefore, a well-designed RISC-V processor has the potential of competing with even the products of industry giants like Intel, AMD and ARM.

Though the ISA itself is more or less finalized, the actual hardware implementations of it is still under development. With its growing popularity, a working RISC-V processor has good prospects for success both academically and commercially. Hence the *RISC-V* *Base ISA Processor* project was launched for the purpose of developing such a hardwareimplementation. The objectives and scope of the project is as follows.

### 1.2 Primary Objectives of the Project

The objectives of the project were to,

#### Design a high performance processor using RISC V ISA

After RISC-V’s introduction many academics have used it to develop processors for different applications. Some currently available work is included in the literature review section. This project was another such implementation, concentrating on giving the maximum possible performance in FPGA and ASIC environments. The goals of achieving lower power and area were important but was secondary to the performance goal.

#### Create a portable, extensible and hardware independent soft processor core

The design was coded in Verilog HDL and was organized in such a way to allow later extensions and upgrades (floating point, multiply divide) and also to allow easy instantiation within other HDL based designs. Furthermore, apart from implementing the processor on FPGA, the design is easily extensible to a fabrication on silicon using a ASIC design flow.

#### Achieve equivalent or superior performance to similar processors such as Xilinx MicroBlaze and Altera Nios II

The ultimate objective of the project is to produce a processor design which is capable of competing with currently available soft-core processors like ARM, Xilinx, Nios etc.

### 1.3 Scope of the Project

While pursuing the above goals, we have limited ourselves to the following scope

#### RISC-V Base ISA with only integer operations

The project concentrated only on implementing the integer pipeline of the processor. This was mainly due to the limitation of time. However, the design ensures that future extensions to floating point units will be possible.

#### Multiplication and division extensions

The base ISA includes only addition and subtraction as mathematical operations. The integer multiply and divide instructions are planned to be added at the next stage of the project.

#### Design for optimal performance, and for portability

The main objective of the design was performance, with costs like area and power considered secondary to that goal. In terms of portability, the design will be pluggable to other HDL systems and SoC type applications through standard AXI interfaces.

#### Implement in FPGA, but extendable to ASIC implementation

The generated RTL was coded to be as hardware independent as possible, though for demonstration purposes, synthesis constraints were targeted towards Xilinx FPGAs. But Xilinx or other third party IP Cores was not used in building the processor to allow a portable and licence-issue free design. Therefore, extension to ASIC or other platforms can be carried out with relatively small modifications.

#### Compare with existing designs and implementations

The processor is compared in terms of power, area and performance with the SHAKTHI RISC-V processor, built by IIT Mumbai.

### 1.4 Literature Review

The RISC-V architecture specification was introduced by the Computer Science division, EECS, University of California, Berkeley, with the goal of providing an open source framework for computer architecture research and education. The ISA was designed to be simple, realistic and micro-architecture independent, thus providing adaptability to multiple implementation domains such as embedded/IOT, desktop, mobile and server computing [1]. The architecture is presented in a modular manner, with a simple Base ISA complemented by a series of extensions (such as Single Precision Floating Point, Double Precision Floating Point, Integer Multiply-Divide etc.) [2]. The specifications for the base

ISA to be used in this implementation is included at [2] and the draft for the ongoing supervisor level specification is detailed in [3].

A number of text books can be referenced to gain a basic understanding of the principles of computer micro architectural design needed in this project. The books [4] and [5] provide basic beginner level understanding of the concepts needed. The book [6] is a more advanced text providing more insight into the quantitative aspects of architectural design decisions. It is the main reference textbook for the project and provides extensive examples on real world architecture from Intel, AMD, MIPS and other processors and discusses the good and bad design decisions taken on their design. Additionally, the online Computer Architecture lecture series by Dr. Onur Mutlu of Carnegie Mellon University (available at https://users.ece.cmu.edu/ omutlu/) provides useful insights to processor design.

Currently several other RISC-V processors are being developed by other research teams from around the world. The ISA developers (at UC Berkeley) have released open source design files for a number of different RISC-V implementations with different pipeline stages, controllers and interlocks [7]. But many of these designs are targeted towards ISA simulation and education purposes and not completely optimized for implementation. UC Berkeley has also released a number of practical implementations by using agile design approaches described in [8], some of which include [9], [10], [11]. Out of these, the Rocket Chip is the leading RISC-V implementation and is an available open source at [12]. Pulpino, a RISC-V implementation developed by ETH, Zurich, focuses mainly on parallel computing and low power applications, with source code available at [13]. The SHAKTI processor initiative is another research project funded by the Indian government to develop its own line of RISC-V processors [14].

In addition to researching other RISC-V implementations, it is also important to study other processor architectures. For most popular architecture, the ISA is available but the micro-architectural details are usually undisclosed or is proprietary. Intel x86 architecture [15] is the most popular processor family in the desktop market. The architecture itself is a Complex Instruction Set Complex Instruction Set Computing (CISC) type, but the processors translate each instruction into several low complexity instructions at runtime and executes them in a RISC type core [6]. AMD also borrows extensions of this ISA for most of their processors.

For embedded and mobile applications the processor of choice is ARM. The ARM family of processors uses a RISC architecture [16], which is arguably simpler, power saving and area efficient than x86 computers. The processors are marketed under 3 brands, Cortex-M, Cortex-R and Cortex-A which are optimized for power efficiency, high performance and balanced power/performance respectively [17]. For the FPGA market, Xilinx and Altera provides two soft processors, MicroBlaze [18] and Nios-II [19] respectively. Since in the project, the implementation is being simulated in a FPGA, these two processors hold the best choice for performance comparison and benchmarking.

In this chapter, we have discussed the background over which the project was initiated, objectives of the project, scope and other similar work done so far. In next chapter we will be discussing the hardware design and implementation of our system. It explains the architecture we have followed in our design in detail.

# Chapter 2

HARDWARE DESIGN AND IMPLEMENTATION

This chapter extensively describes the main hardware systems of the design. The full system comprises of four major components,

* pipeline,
* instruction cache,
* data cache,
* interconnection interface

A detailed explanation of the design and functionality of each of these sub modules are included in this chapter.

## 2.1 System Overview

A detailed diagram of how the implemented processor should be integrated to a system is shown in Fig 2.1. The control unit, main execution pipeline and the two L1 caches are implemented within the IP core that we have developed. All these systems have been carefully designed using Verilog, with the aim of achieving maximum performance as well as maximum operating frequency. As such a deeply pipelined design was chosen for both the caches and for the execution/control units. To further increase the performance of the core, several other improvements have also been made, such as the inclusion of stream buffers and victim caches to the L1 caches, implementation of a form of prefetching to further reduce cache misses, inclusion of a forwarding and interlock mechanism to effectively deal with concurrent co-dependent instructions etc.

Though not strictly necessary, a L2 cache would be very important for achieving maximum performance by assisting the L1 cache system in quickly providing necessary instructions and data for the processor. In area constrained designs or low performance designs, the L2 cache can be skipped and the processor could be directly connected to the AXI interconnect. As such L1 caches have been designed with an AXI interface. Main memory and all the other peripherals can be connected to the processor through the AXI interconnect. Optionally a DMA controller should also be connected to the AXI interconnect for applications with heavy memory operations.

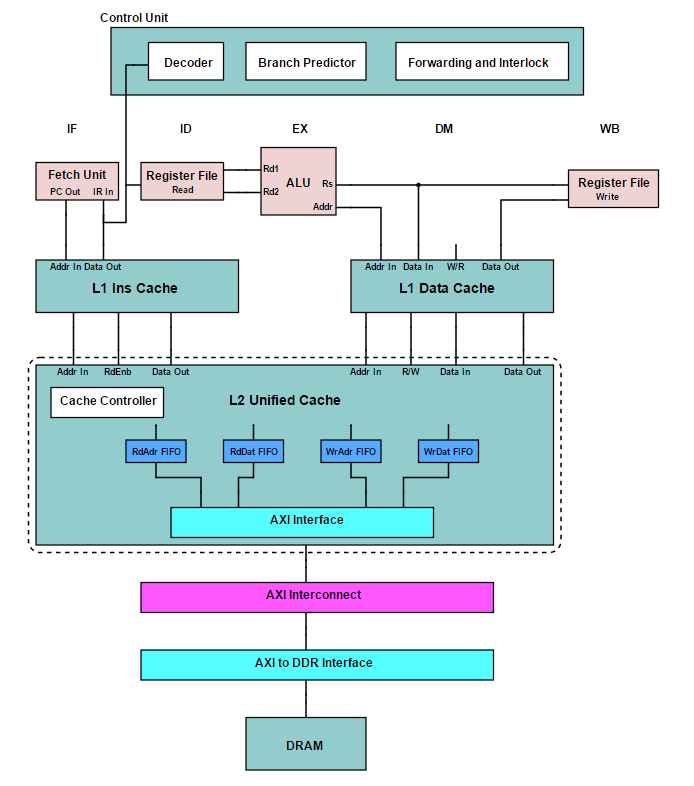


Fig.2.1. System Overview

## 2.2 Main Pipeline

The main pipeline or the execution pipeline is basically the datapath of the processor. It is responsible for the fetching, decoding and execution of the instructions in the processor. In the designed RISC-V architecture, the pipeline is logically divided into 5 main sections,

* Instruction Fetch (IF)
* Instruction Decode (ID)
* Execution (EX)
* Data Memory operations (DM)
* Write Back (WB)

To achieve our high frequency goal, we have had to subdivide some of these stages into further pipelines. The ID stage is basically divided into 4 such stages PC, IF1, IF2 and IF3. Similarly DM stage also had to be subdivided as DM1, DM2 and DM3. The main structure of the pipeline is shown in Fig 2.2. Each of the pipeline stages are described in more detail in the following sections.

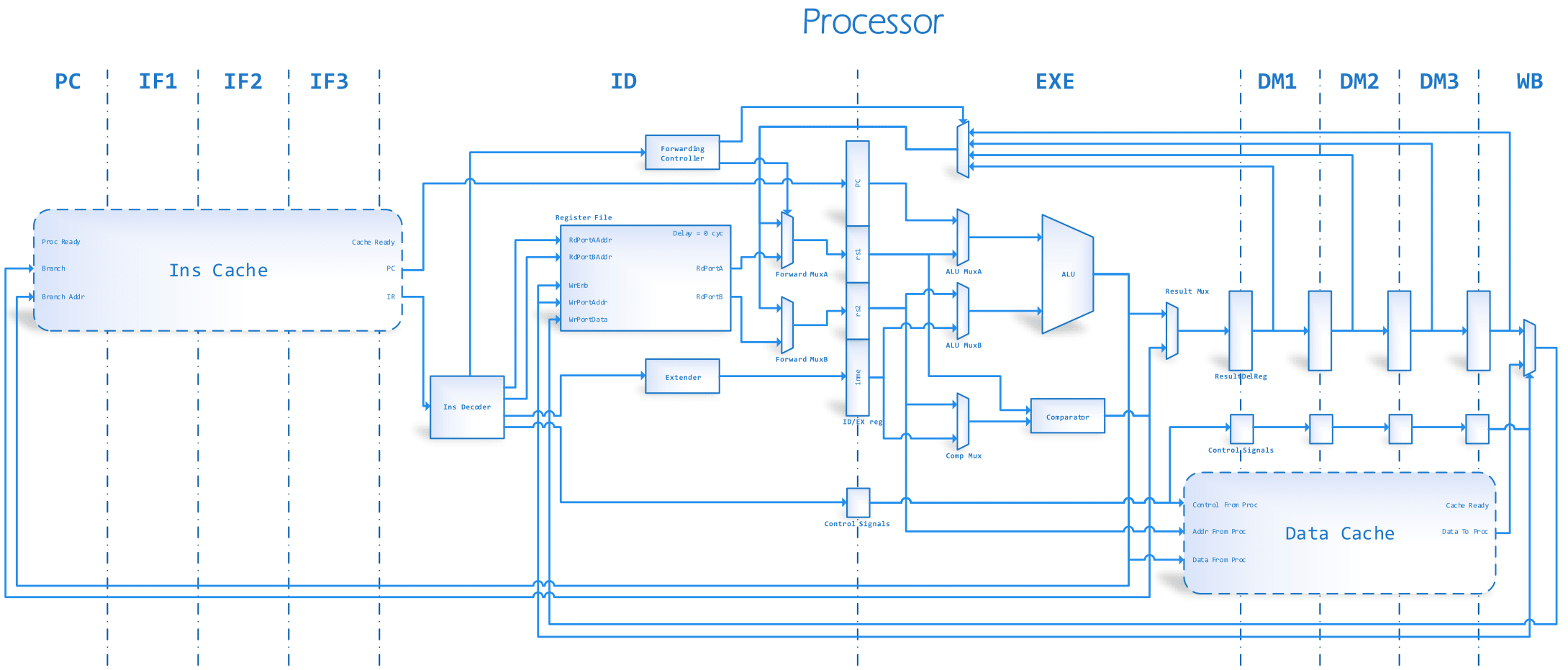


Fig.2.2. Main Pipeline

#### 2.2.1 Instruction Fetch Stage

Instruction fetch stage is primarily integrated into the instruction cache. Every instruction will take three clock cycles to be available at the IF/ID buffer (pipeline register between IF and ID stages) if there is a cache hit. The number of cycles can increase if there is a cache miss, in which case a special CACHE\_READY signal is pulled down to indicate the miss. The control unit will do a special subroutine to ensure that the pipeline will not go to a problematic condition during this period.

The Instruction fetch stage is fully pipelined and in each stage the following functionalities are carried out

* PC stage – Correct PC is generated, considering branch or miss conditions
* IF1 stage – PC is sent to instruction cache, tag search & line search starts
* IF2 stage – Tag search completes and tags are compared
* IF3 stage – Line search completes and data recovered if cache hits

At the end of the IF3 pipeline stage the requested instruction according to the program counter register is available for the decoding stage. The physical structure of the Instruction cache is further elaborated on subsequent sections.

#### 2.2.2 Instruction Decode Stage

Instruction decode stage is the stage that decodes the fetched instruction from the IF operation. The instruction is identified using the OpCode by the decode stage controller and the relevant control instructions are given to the IMM EXTENDER and the Register Array. At the end of the stage, the ID/EX buffer (pipeline register between ID and EX) is updated with the required values.

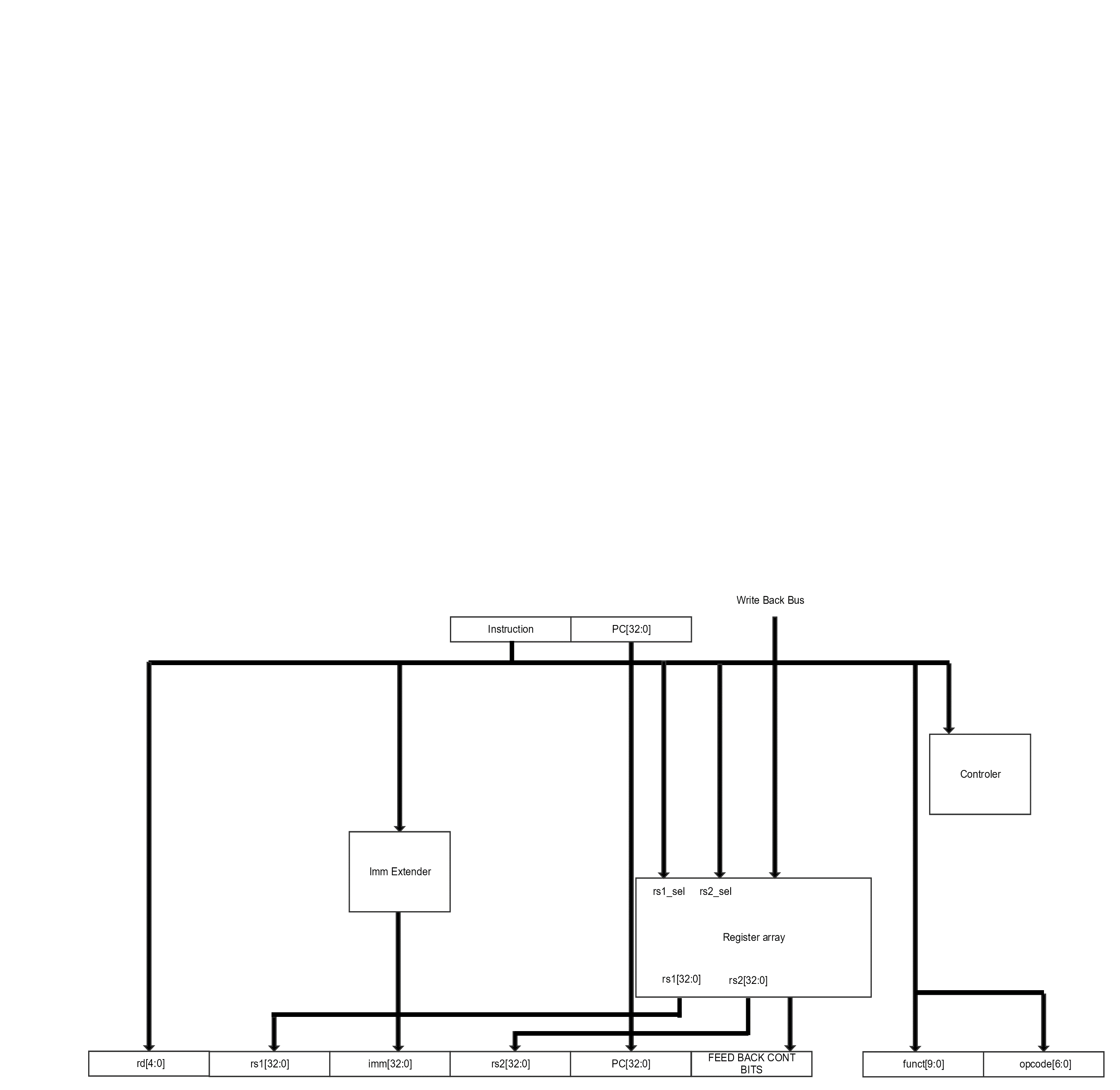


Fig.2.3. Instruction Decode Stage

The main components of the decode stage are IMM EXTENDER, Register Array and Decode stage controller.

* IMM EXTENDER

RISC-V base ISA has six types of immediate instructions with five immediate types defined in the specifications. Therefore, depending on the immediate instruction type, the conversion of the immediate also changes. IMM EXTENDER is the unit which is responsible of the correct conversion of the immediate for the execution stage.

* Register Array

Register Array contains all 32 integer registers defined by RISC-V base ISA. It has separate ports to read and write and in case of a register-register instruction it can have two parallel read accesses (read and output two register values concurrently). Only a single write port is included, since the processor has only a single issue pipeline. Furthermore being a register memory, it has a combinational read operation and a single cycle write operation. Feedback is used to prevent register value duplication and increase the performance of the pipeline. More details on this are included in a following section.

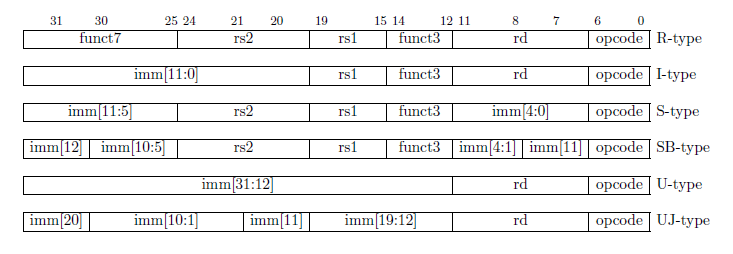


Fig.2.4. Immediate instruction types

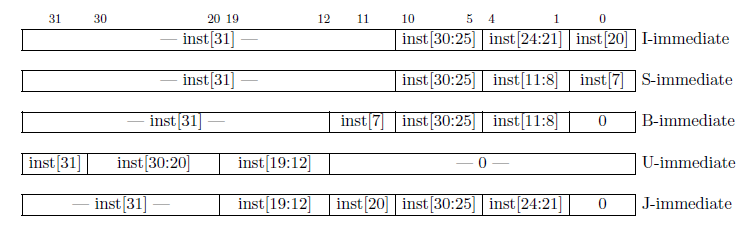


Fig.2.5. Immediate types

#### 2.2.3 Execution Stage

This is the stage in which the processor carries out the required arithmetic and logical operations according to the instruction. The Execution stage consists of data path multiplexers (M1, M2, M3), ALU, COMPARATOR and Execution stage controller. The data at the ID/EX buffer are guided through the required data path and at the end of the stage, processed data is stored in the EX/MEM buffer. The main reason for separating arithmetic and logical operations is because in some instances for example in case of a branch the address calculation and the logical operation can be done in parallel improving the performance of the processor while also being less complex when implementing.

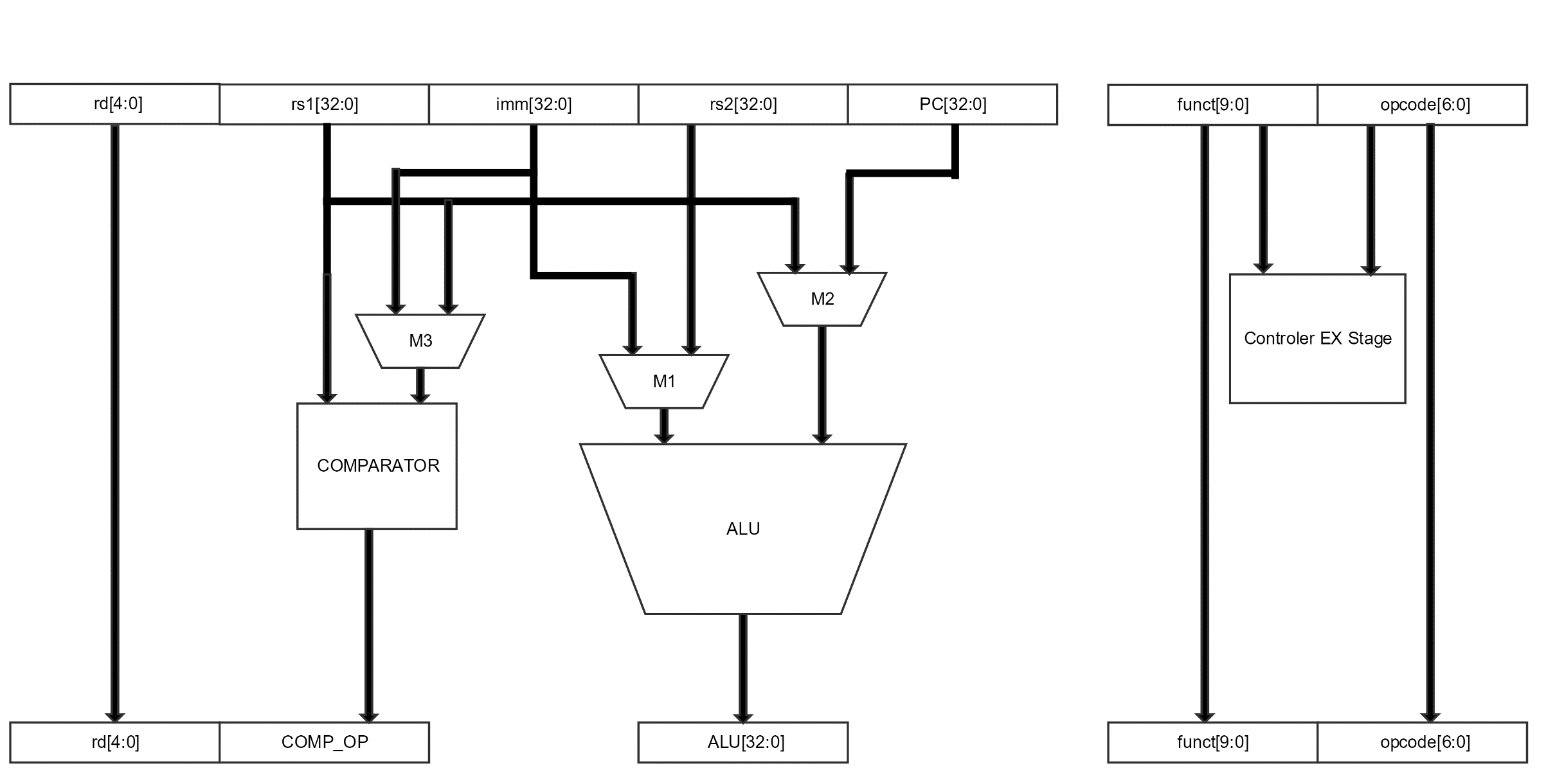


Fig.2.6. Execution stage

* ALU (Arithmetic Logic Unit)

The ALU is responsible for carrying out all the arithmetic operations defined by the instructions. It is capable of executing 8 arithmetic operations namely; Addition, Subtraction, AND, OR, XOR, Shift Left Logical, Shift Right Logical, Shift Right Arithmetic.

* COMPARATOR

The Comparator is responsible of executing all the logical operations if required by an instruction. It is capable of executing four types of logical operations namely: Equal, not Equal, Less than and Greater than or equal. In the case of Greater than and Greater than or equal operations the compiler switches the operands and the available logical operations Less than and Greater than or equal are used.

* Execution stage controller

The Execution stage controller controls the data path, basically the data in ID/EX buffer is properly selected by controlling the path of the multiplexers M1, M2 and M3. Furthermore, it selects the COMPARATOR operation and ALU operation. All this is done by considering the OpCode and the functional bits of the instruction.

#### 2.2.4 Data Memory Stage

Memory stage contained within the data cache, every data read or write can take minimum three clock cycles to be available at the MEM/WB buffer (Memory / Write back buffer) if there is a cache hit. The number of cycles can increase if there is a cache miss, in which case a pipeline stall subroutine is executed by the main controller, similar to what is done in case of an instruction fetch stall. The Memory stage is fully pipelined and in each stage the following functionalities are carried out,

* DM1 stage – Address is sent to data cache, tag search & line search starts
* DM2 stage – Tag search completes and tags are compared
* DM3 stage – Line search completes and data recovered if cache hits

Data cache is an improved version of the instruction cache with the inclusion of data write capability.

At the end of the DM3 pipeline stage the requested data according to the given address is available for the write back stage. The physical structure of the data cache is explained in detail in a following section.

#### 2.2.5 Write Back Stage

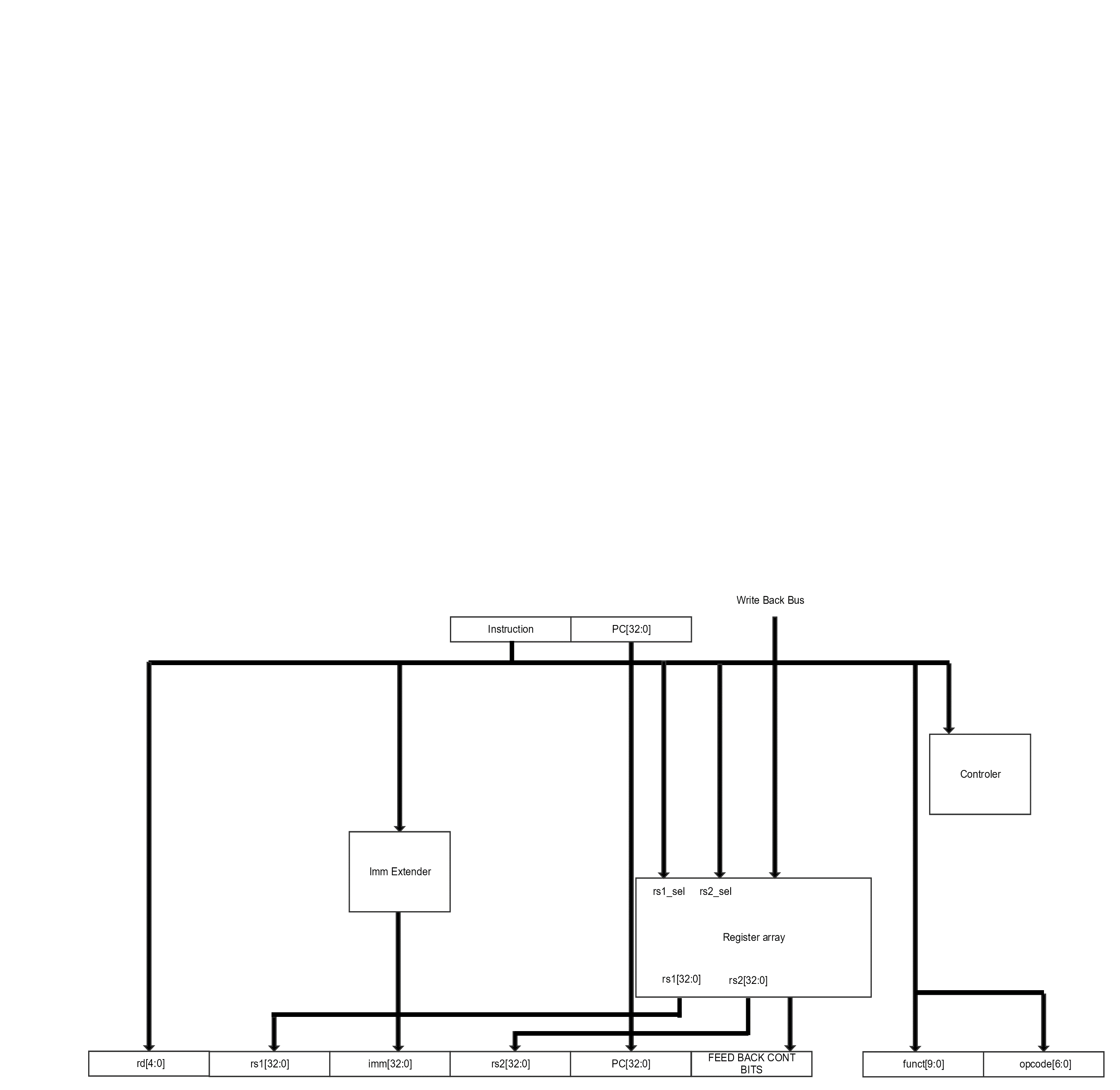


Fig.2.7. Write back stage

The write back stage is the final stage of the pipeline, meaning that an instruction reaches its end of life after this stage. The function of the write back stage is to write the result of the instruction back into the register file. The stage is only utilized by a arithmetic/logic instruction or a data memory instruction.

The main components of the write back stage are the write back bus and the Register Array. The write back bus contains three sub busses which are,

* The data bus
* Data valid control signal
* Data destination register

These contain the control information and data to make sure that the correct register is updated with the correct value.

#### 2.2.6 Data Feedback

One of the problems faced in such a deep pipeline is that when a register is in the pipeline to be updated (in EXE/DM1, DM1/DM2, DM2/DM3, DM3/WB buffers), if another instruction needs the value of that register, the newer instruction will use the old outdated register value for its calculation. To avoid this, the simplest (and the least performing) method would be to stall the pipeline until the source register is properly updated with the new value. An alternative method, called feedback is introduced in this particular design to gain increased performance.

Detection of this issue is done in the instruction decode stage by the Register array. Additional single bit register array is used to store if the indexed register is committed in the pipeline. A shift register array consisting of five 5bit registers that are used to keep track of the stage the register is in. The control signals are generated and passed to the Execution stage. These control signals select which data path should be set to the ALU or Comparator.

Fig.2.8. Data feedback

#### 2.2.7 Alternative Approaches Considered

When designing the pipeline one of the strategies considered was using out of order execution in place of in order execution. The advantage of using out of order execution was that we can exploit the data-level parallelism of the instructions and achieve better performance compared to an in order processor. However, the major issue faced was that designing such a processor was very complicated and considering the time available the final decision was made to make the processor in order.

Different pipeline depths were also considered when designing the processor. First we considered a five stage pipeline. The issue with the five stage pipeline was that the required speeds were not achievable with just five stages, therefore the pipeline had to be extended for ten stages as in the current design. The main timing problems occurred in the two caches, which had to be sub pipelined again into 3 stages. After the added pipelining the critical path of the design went over to the execution stage. A complex strategy of load balancing was implemented to pre-compute some of the intermediate signals needed in the EXE stage at the ID stage, further increasing the clock frequency. Even with all these strategies, the critical path is still inside the EXE stage. The only possible improvement at this stage is to subdivide the whole EXE stage again, as EXE1 and EXE2.

Data forwarding can either be merged with the pipeline in the instruction decode stage or in the execution stage. When a decode feedback strategy a stall of one clock cycle was required to synchronize the pipeline, in a case where two neighboring instructions were co-dependent. On the other hand, the execution feedback does not have this issue. But its disadvantage is that it adds more and more combinational logic for the already complex execution stage, resulting in a lower max frequency. By comparing the two strategies it was decided to take the decode feedback strategy, to achieve a higher clock frequency.

## 2.3 Cache Architecture

Cache memory is a small, fast memory which stores frequently used memory locations with the objective of reducing the latency for data/instruction access. In our design, two separate caches are implemented for data (data cache) and instructions (instruction cache). Each cache it allows parameterization of data cache and instruction cache sizes which is a great advantage for users, since they can select optimal sizes depending on current requirements. However, the maximum size of cache will not exceed 64KB due to practicality (in terms of available area) reasons.

Cache architecture is arranged in a hierarchical manner with 2 levels, consisting of separate L1 caches and a unified L2 cache. L1 caches are optimized for fast operation with minimum latency (while keeping the operating frequency high), while L2 cache is concerned more with reducing misses (by having more data stored inside). Both L1 caches are parameterizable in its associativity, cache size, block size, and downstream bus widths. In the data cache, a write-back policy is implemented for superior performance. Miss penalties are further reduced by the inclusion of stream buffers and victim caches.

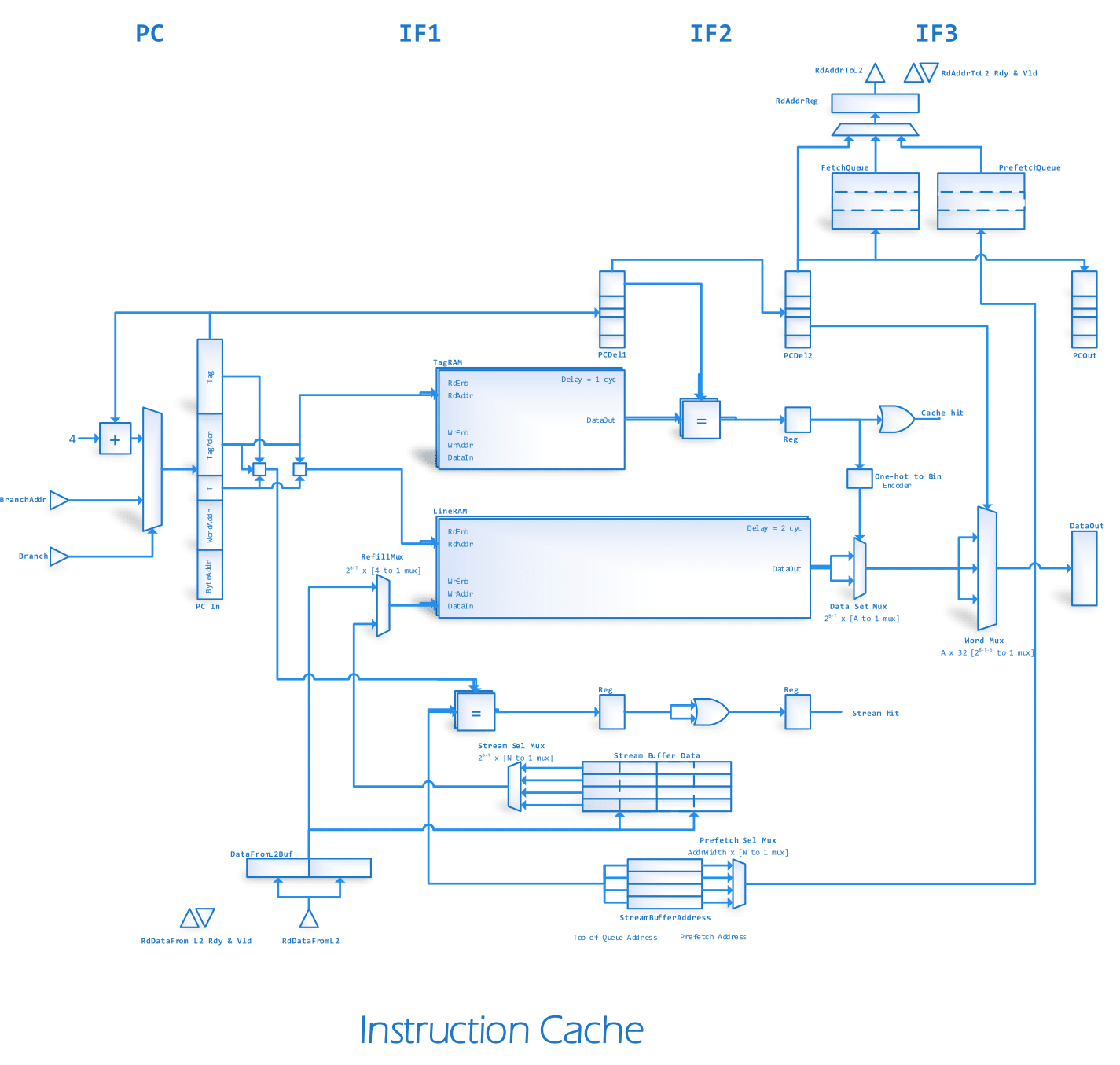
#### 2.3.1 Instruction Cache

Instruction Cache is the memory which stores frequently used instructions and thus requires only the reading capability. Therefore, the design is arguably simpler than the data cache. All the data and address buses coming in from the processor side are 32bit wide. When an instruction is requested by the processor and is available in the instruction cache, the cache hit is notified and the requested instruction is forwarded after 3 clock cycles. The cache is fully pipelined. (Meaning, at the start of the program there will be a delay of 3 clock cycles for first result to appear at output, after which the new data will be available in each cycle.) If it is a cache miss, the requested instruction will have to be taken to the instruction cache from the L2 cache which might take several clock cycles. During this time the processor will have to be notified to not request further instructions.

Incidentally, since the processor and the L1 cache is deeply intertwined, the PC register and its management is also handed over to the L1 instruction cache controller. The main reason for this is that, in case of a cache miss, the state of the processor must be taken to the condition immediately before the miss occurs (which means that the processor must not ‘feel’ the miss occurring from its perspective). To accomplish this the cache has to artificially refill the PC with its own values.

Main inputs/outputs to the system from other sub systems are,

* Between processor and L1 cache,
  + Processor ready (input) – Indicates whether processor is ready to accept any data from cache side or not.
  + Cache ready (output) – Indicates whether cache is ready to accept any requests from processor side or not. If data cache, instruction cache and processor are ready, processor will shift forward its pipeline and execute an instruction.

**

* + Cache hit/miss (input) – When a request is received by the instruction cache, it will search in the tag memory and the line memory. If particular instruction is available in the cache, the cache hit will be notified as an output. Otherwise cache miss will be notified.
  + Branch enable (input) – If the instruction takes the branch, use the address given by the processor. Else simply increment PC by 4
  + Branch address (input) – Corresponding branch address
  + Instruction to processor (output) – Instruction for the given PC
* Between L1 cache and L2 cache
  + Address to L2 cache (output) – Address for requesting from L2
  + Data from L2 cache (input) – Brings requested data from L2 cache
  + Request Source (input/output) – Source which request data from L2 (whether stream buffers or L1)
  + All buses between L2 and L1 cache are managed through a standard Ready/Valid interface

#### Datapath

Instruction cache consists of a tag memory and a line memory which are variable in size as the cache design is parameterized. Only 1 clock cycle is needed for a tag memory to do a tag search while a line memory takes 2 clock cycles for a search. Tag memory, being considerably smaller than the line memory, is assumed to be accessible with a single clock cycle. The cache itself is fully pipelined with four stages, PC stage and 3 instruction fetch stages. The instruction fetch stages are;

* IF1 – tag and line search stage,
* IF2 – tag process stage (while line search is still running) and
* IF3 – multiplex stage

respectively. This 4 stage pipeline was designed with the intention of achieving the required high speed of operation.

In the first stage (PC) the program counter is calculated. The PC value can come either from the processor (as a branch address), from a special adder (sequential increment of PC) or from the cache itself (when dealing with cache miss situations). At the end of PC stage the correct PC source (as defined by the main pipeline controller) is registered to be sent to IF1.

In the next stage, accessing the tag memory and the line memory takes place. For high frequency operation, each of these memories are given one or two clock cycles to complete its access. Furthermore necessary control information needed by IF2 and IF3 are generated and buffered in pipeline registers.

IF2 stage starts with the tag memories providing the tags and valid bits. Afterwards the tag of the request is compared with each of the (valid) tags coming from different sets. The comparison answer for each set is stored to be used in IF3. During this stage, the line memory is still being searched.

IF3 stage starts as soon as the line memory access completes. The line memory provides the whole cache line, but we only require a 32bit section out of it. So a multiplexing stage is required to reduce the long cache line into a single 32bit value. At the IF3 stage, the tag processing is already complete, so Cache Miss or Cache Hit status can be easily calculated. In case of a cache hit, the IF3/ID registers are filled with the fetched instruction.

If the cache does not hit, a request is sent to the L2 cache through the Fetch Queue. The response of the L2 request (which comes after several clock cycles) is received by the RdDataFromL2 buffer, after which it can be written back to the L1 cache. Due to the bus width constraints, it may not be possible to bring a whole cache line back in a single cycle, in which case the buffer acts as an assembly point to construct the full line part by part. Furthermore since the cache supports critical word first and early restart, the response from L2 is expected in a critical word first manner. The critical word is quickly written to the cache (as soon as it arrives) so that the fetching can restart with minimum delay.

When running a sequential stream of instructions, there will always be first-time misses in the cache (called Compulsory misses in literature). A way of reducing these types of misses is to prefetch some of the data from the L2 cache before they are needed. This functionality is achieved by the use of stream buffers. A stream buffer looks for patterns in the instruction requests, locks on to them, predicts possible future requests and prefetches them. These prefetch requests has much lower priority than the L1 miss requests (in essence, prefetch requests are served by the bus only if there are no other requests to serve). Parallel to the L1 cache, the stream buffers are also searched. If a stream buffers hits when the L1 cache does not, the resulting miss can be handled much quickly than a complete miss to L2 (4 clock cycle vs more than 8 cycles). Incidentally, stream buffer allocation, deallocation, updating, invalidation and other control operations are a fairly complex task which should be handled by a separate controller.

#### Control Modules

The instruction cache is a combination of many controlling units. Combination of these controllers help to run instructions requested by the processor properly. Controllers in the design are,

* Pipeline Controller

The pipeline controller is the main controller of the unit. It acts as an interface between the L1 cache and the processor. It controls the pipeline and communicates with the processor to maintain proper functionality. The processor is acknowledged by the controller whether the cache is ready to communicate or not. When a cache miss occurs, the controller will freeze the pipeline until it is resolved.

* Refill Controller

Refill controller is responsible for filling the cache appropriately. It will get data from the stream buffers or the L2 cache. First the stream buffers are checked for data, if data is not there the controller will go to L2. Typically, 4 effective clock cycles will be taken for a refill from stream buffers while 8 clock cycles at minimum will be taken by a refill from L2 cache.

* Stream Buffer Controller

Stream buffer controller is another important unit which controls all the functionalities associated with stream buffers. It monitors streaming behavior of the processor, allocates and deallocates stream buffers accordingly and manages different requests made by various units to stream buffers. The controller also checks stream buffers to identify whether they are full or empty and indicates whether there is a hit or miss after a search in the stream buffer.

* Data from L2 Controller

Data from L2 controller manages the data that is sent to L1 cache and stream buffers from L2 cache. Here the controller supplies the requested data to the correct destination.

* Address to L2 Controller

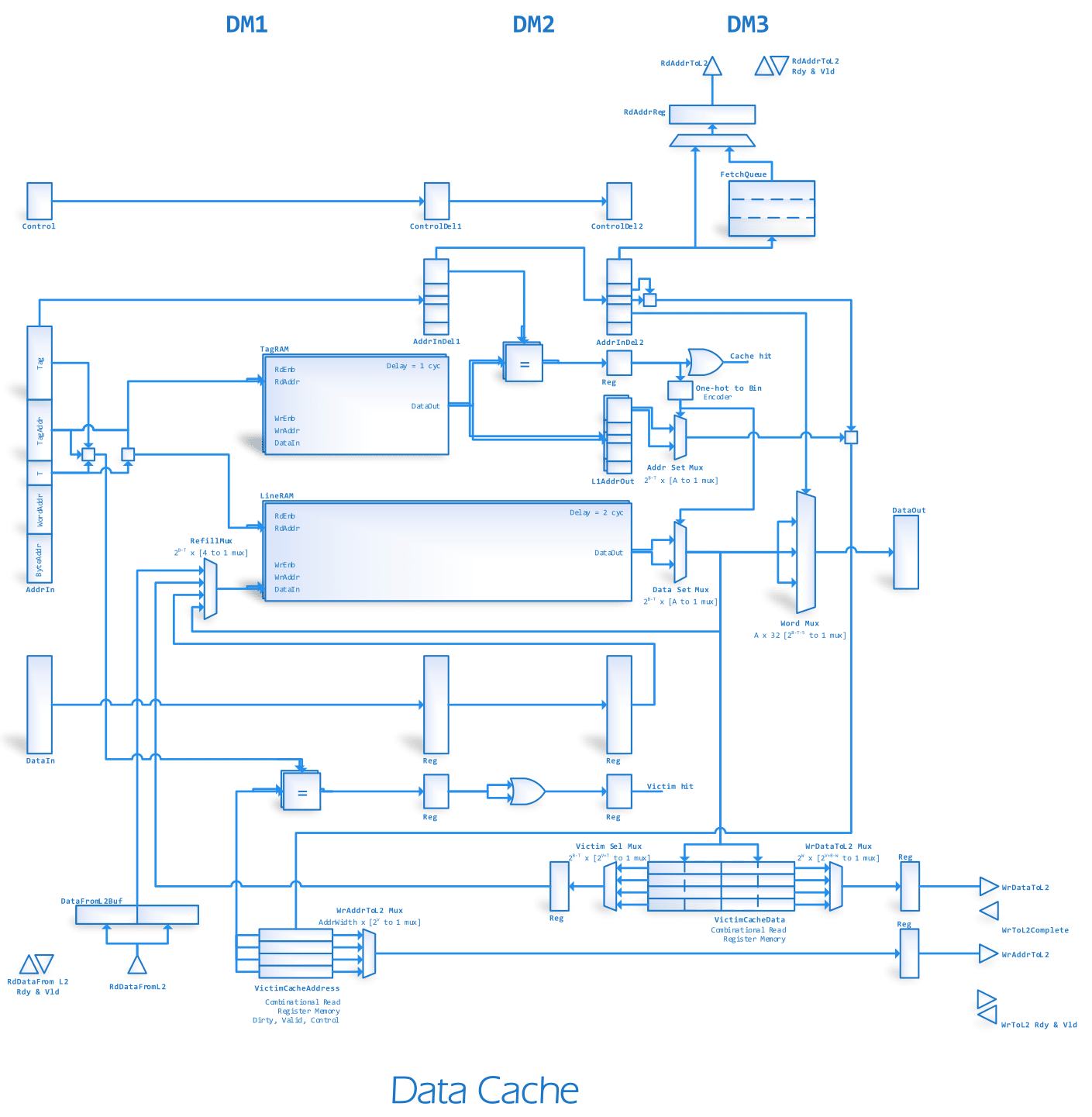
Address to the L2 controller gets the required data from the L2 cache. The data will be requested by either stream buffers or L1 cache. Priority is set for L1 cache. Therefore, if both stream buffers and L1 cache make requests at the same time, the L1 cache request will be considered prior to the stream buffers.

#### 2.3.2 Data Cache

Data Cache is the memory which stores frequently used data. This cache incorporates both read and write capabilities. Therefore, the design is somewhat more complex than the instruction cache. In design point of view, data cache and instruction cache are similar except in some sections. For instance, data cache does not have stream buffers like in instruction cache. This is because data memory requests do not come as streams (like instruction requests do) and thus stream buffers are less effective. Furthermore, data cache has a specific instruction which can forcefully remove a cache line from the cache. This is known as cache flushing, and is a requirement for managing the cache coherency. To improve the performance and also for dealing with some of the coherency issues, a victim cache is implemented for the data cache.

Main inputs/outputs to the system from other sub systems are,

* Between processor and L1 cache,
  + Processor ready signal (input) - Processor ready signal indicates whether the processor is ready to accept any data from cache side or not
  + Cache ready signal (output) - Indicates whether the cache is ready to accept any requests from the processor side or not. If the data cache, instruction cache and the processor are ready, the processor will shift forward its pipeline.



* + Cache hit/miss (output) - When a request is received by cache, it will search in tag memory and line memory. If a particular data is available in the cache, cache hit will be notified as an output. Otherwise the cache miss will be notified.
  + Address from processor (input) - Target address for reads and writes
  + Control from processor (input) – Whether the request is a read, write, idle or a flush
  + Data from processor (input) – For cache writes
  + Data to processor (output) – For cache reads
* Between L1 cache and L2 cache
  + Read address to L2 cache (output) – Address for requesting from L2
  + Data from L2 cache (input) - Brings requested data from L2 cache
  + Write address to L2 cache (output) – Address for writing data to L2
  + Data to L2 cache (output) - Sends requested data to L2 cache
  + Control to L2 (input/output) – Instructions to L2 cache (like flush requests) and feedback on status of issued writes (acknowledges)
  + All buses between L2 and L1 cache are managed through a Ready/Valid interface

#### Data Path

Tag memory and line memory works as the same as in the instruction memory. Though the pipeline has only 3 data memory stages, it sometimes effects the execution stage also (when a L1 cache miss occurs). This arrangement is similar to what happens in the PC stage for the instruction cache, but here EXE stage is affected instead.

The data memory stages are;

* DM1 – tag and line searching stage,
* DM2 – tag processing stage (while line fetching is still running) and
* DM3 – multiplexing (if it is a cache read)/writing (if it is a cache write)

respectively. Data cache architecture is designed as a 3 stage pipeline with the intention of achieving the required high speed of operation. Since the arrangement is very similar to that of the instruction cache, only the differences will be highlighted here.

The main difference from the instruction cache is that this cache is capable of writing data also. Therefore the write port of the memories must be shared between the L2 refill paths, L1 write paths and also victim cache refill paths. The 4 to 1 multiplexer at the L1 write port is used towards this end.

For this particular cache we have decided to go with a victim cache instead of a stream buffer. The victim cache also acts as a write buffer for write requests towards the L2 cache (this allows the data cache to continue operating, without waiting for a write to L2 to complete). The victim cache basically stores all the requests that are evicted from the L1 cache. Being a small, fully associative cache, it is capable of reducing the number of misses by a considerable amount. Just like the stream buffers, the victim cache search runs in parallel with the L1 search, therefore the delay associated with a L1 miss and Victim hit is only 4 clock cycles.

To enable the write capability an additional write port to L2 is also required. A small controller is set up to properly manage the writes and corresponding acknowledgements.

#### Control Modules

Data cache is a combination of many controlling units. Combination of these controllers helps to give the data requested by the processor properly.

Controllers in the design are,

* Victim Cache Controller

This controls the victim cache. Victim cache is a very small memory about 4 cache lines located between L1 and L2 caches. It keeps most recently evicted L1 cache lines. Therefore, if past data is needed again by the L1 cache, these will be supplied immediately by the victim cache.

* Write Data and Write Address to L2 Controller

In case the evicted cache lines are dirty, the relevant L2 cache lines have to be updated immediately by the victim cache. Write Data and Write Address to L2 Controller send the data and the relevant addresses to write for the data cache. Here the controller should supply both data and address to the correct destination. Therefore, this has to manage two buses.

* Address to L2 Controller

Address to the L2 controller gets the required data from L2 cache. Here no stream buffers are included in the implementation as in the data cache. Therefore, the design is much simpler than in data cache.

* Pipeline Controller
* Refill Controller
* Data from L2 Controller

Pipeline Controller, Refill Controller, Data from L2 Controller and Read Address to L2 the controller very similar to that of instruction cache with only slight variations. As the data cache has both reading and writing capability, controllers are more complex than the instruction cache.

#### 2.3.3 Alternative Approaches

The currently available design was implemented after considering many alternative approaches. These approaches were not implemented due to various disadvantages when compared with selected architectural styles.

*Number of Cache Levels*

Data cache is organized in a hierarchical manner, including two levels of cache. The considered alternatives were to implement as a single level cache or go for higher levels more than two. Single level cache will not provide the desired performance of the system. To achieve optimized performance, we need to go to high levels in the cache design. Higher cache levels like 3 or more will increase the complexity of the design making it very difficult to manage.

*Write Back vs Write Through*

Cache write policies considered, were write back and write through. Write back was selected over write through due to various factors. When writing any data to a write through cache, it will update both the cache and the underlying permanent storage before indicating a successful completion. This always keeps the main memory up-to-date with fresh data. However, this practice results in higher latency since the completion time depends on the lowest (and the slowest) memory in the hierarchy.

In write back policy, I/O write is directly sent to the cache and it will not be sent to the next levels of storage (such as L2 and main memory) immediately. This is significantly faster but it creates a mismatch between data available in cache and main memory (referred to as cache incoherency). This incoherency is solved when such a written (dirtied) cache line is evicted in favor of a new line, when it will be taken from L1 and written back to L2 and below. Write back is faster and more bandwidth efficient compared to write through as it only needs time to write into L1 cache. As our main objective is to gain optimization in performance, we chose write back over write through.

*Direct mapping, fully associative and n-way set associative*

Direct mapping, fully associative and n-way set associative were the cache mapping techniques that were available. 2-way set associative mapping technique was selected after considering many pros and cons.

Direct mapped cache is the one way set associative mapping technique which is the simplest form of cache. However, performance is low compared to other practices, because one location in the cache is shared by several addresses in main memory.

Fully associative cache lines are not dedicated to a specific set of memory addresses. They can hold any address that needs to be cached eliminating low hit ratio seen in direct mapping. Therefore, this gives the highest cache hit ratio. However, this might result in huge latency during searching the cache. When each time a request comes, it must search the entire cache as there are no dedicated lines for addresses. Advanced algorithms must be included to decide when a new entry need to be added. Therefore, fully associativity adds complexity, latency and more cost to the design.

It is a better choice to go for a compromise design between direct mapped and fully associative cache. Requirement can be fulfilled by selecting n-way set associativity. Here one memory address can be mapped to n places in the cache. Therefore, when searching for a particular data, we only need to search in corresponding n places. Since the optimal choice of n depends on the application, we have opted to fully parameterize the set associativity so as to provide additional flexibility to the application engineer.

*L1 and L2 Communication through a custom P2P bus*

Communication between L1 and L2 caches are done through a custom P2P bus. This could be implemented a standard bus type such as AXI. Then we have to include all the relevant logic related to the standard design which might result in additional hardware and sometimes degrade the performance. It will also be more complex as the bus needs to handle various situations unique to the design. Therefore, the best solution is to go for a custom bus which is in our case a simple peer to peer bus. This handles various conditions occur during operation. For example, it handles instances when instruction cache and data cache gives a miss at the same time. This custom bus is optimized for our design in order to pass the required timing.

But to be compatible with third-party IPs, and for situations when the L2 have to be removed, we must have some sort of an AXI interface. To deal with this a small converter from our custom interface to an AXI MM interface was created.

## 2.4 Memory Interfaces and Peripherals

AXI or Advanced eXtensible Interface is a part of ARM AMBA, a set of micro controller buses introduced a few decades ago. There are 3 types of AXI interfaces know as full AXI, AXI Lite and AXI Stream. Standard full AXI is implemented as the interface between cache and the main memory in our design. This is the same interface which connects external peripherals to the system. Full AXI has traditional memory mapped data/ address interface and suitable for high performance architectures. It has the burst transaction capability which is quite useful for our cache to memory communications. Master-slave type communication can be seen in AXI protocols. In full AXI, up to 16 slaves can be connected, read and write transactions can be performed in parallel.

#### 2.4.1 Data Path and Controls

Two separate similar memory interfaces are used to connect data cache and instruction cache to the memory. This is achieved via developing an AXI interface. Implemented AXI interface is connected to 4 FIFOs namely Read Address FIFO, Write Address FIFO, Read Data FIFO and Write Data FIFO. These FIFOs store data and addresses that are sent from cache to memory and external peripherals and wise versa. All the address bus lines are 32 bit wide. But data lines from cache memory interface are 128 bit wide. AXI interface is connected to the main memory and peripherals through AXI interconnect. BRAM is used as memory in the system and it is connected via AXI BRAM controller to the Interconnect. As reading and writing through the interface can be done simultaneously, required timing can be passed easily. As full AXI supports burst transactions up to 256 data transfers (AXI4), it is easy to get cache lines from memory to cache. All types of conflicts that occur during data transactions are taken care by the AXI interface.



Fig.2.9. Memory interface

#### 2.4.2 Alternative Approaches

*Full AXI, AXI Lite and AXI Stream*

Interface in our design is built using standard full AXI bus. The alternatives considered were implementing a custom bus, AXI Lite and AXI Stream interfaces. By going through each alternative approach and their pros and cons, full AXI was the best fit for our design. AXI Lite is the simplest version of AXI interface. It has a simple architecture and more suitable for low-throughput memory-mapped communication. It is a lightweight version of AXI and does not support burst transactions. Address must be sent to access any particular data as it is a single transaction memory-mapped communicator. Therefore, this architecture is not suitable for our design. Even though implementation is simple, higher operational speed cannot be achieved by AXI Lite.

AXI Stream is usually used for high performance designs. However, in our case, this interface is not suitable to be used with the RAM as it does not give much control. Not only to communicate with RAM, our system must be able to deal with other external peripherals too in future. AXI Stream is not the ideal option to choose in that case.

*AXI vs Custom system buses*

We could have implemented a custom system bus which is designed according to our requirements. However, then we have to design it to meet every problem that might occur when transferring data in between a cache, the main memory and the external peripherals. More time is needed for this task and also it might need several optimizations to achieve high speed operation. Furthermore, most of the existing IP cores have AXI in their designs. Hence it is easy to connect our system with other external units.

A detailed explanation about main pipeline, execution stage, memory stage, write back stage, data feedback and alternative approaches we used were presented in this chapter. In the next chapter we will be discussing the overview of results we obtained from our implementation.

# Chapter 3

RESULTS AND DISCUSSION

This chapter discusses the results of our final design related to the main pipeline, cache architecture, memory interfaces and peripherals. Current resource utilization, timing and design methodology also have been discussed briefly in this section.

We have designed, RTL coded, synthesized and implemented our processor design using Xilinx Vivado software. Afterwards we have implemented a test setup which uses a Zynq FPGA to provide the necessary environment the processor needs to operate. The ARM core located in the Zynq chip was used to feed data and to take data back from the RISC-V processor. The whole setup was physically implemented in a Zedboard evaluation platform and several assembly programs were run on it to verify the proper functionality of the design.

Afterwards we have unofficially published the results we have gotten on the RISC-V mailing lists (which is a forum in which many RISC-V researchers communicate with each other). The comments have been positive towards the architecture we have presented, and some improvements has also been suggested which we could implement in the future. Furthermore several researchers have sent their own RISC-V implementations to compare and contrast against our own design. It was seen that compared to some of the implementations (like SHAKTHI, from IIT Mumbai) our implementation has drastically reduced area while also managing a fairly high clock frequency.

### 3.1 Resource Utilization (in Zedboard)

After implementation in Zedboard we used the Xilinx Vivado software to get the resource utilization. This utilization was compared with one of the existing processor designs, namely SHAKTHI processor developed by IIT Mumbai. The SHAKTHI processors comes with a much wider variety of options, with several classes implemented concentrating on superior performance, superior power usage or area usage etc. By analysing each of these processor classes we identified the closest processor design to ours, which was SHAKTHI C class, which had in-order execution strategy, similar cache architectures and similar pipelining. We implemented the SHAKTHI processor also on the same Zedboard hardware and compared results (see Table 1).

From the resource utilization it could be concluded that our processor has much lesser resource requirement than the SHAKTHI processor. But these results cannot be directly compared because the Indian processor has implemented several extensions to the base RISC-V ISA. However we feel that even with those extensions (multiplication and division), our processor core will not exceed the 10000 LUT mark.

Another point where our design has shown superior area efficiency is in the BRAM count. We have managed to implement the 64KB instruction and 64KB data cache version with only utilizing 32 BRAM blocks. Since each BRAM is capable of storing 4.5KB of memory we can easily see that the total utilization of 4.5KBx32= 144KB is fairly close to the 128MB theoretical minimum (the additional overhead we have incurred is mainly for tag, valid and dirty memories). However the 44.5 BRAM utilization of the SHAKTHI core is much higher than the minimum required.

With this evidence we could claim that our processors has utilized the resources in a much more efficient manner than the SHAKTHI designers. This could be attributed to the fact that they have been using a High Level Synthesis method compared to our Verilog toolflow. It is generally agreed in the industry that hardware architectures created through HLS toolflows does not utilize resources as well as manually hand-crafted architectures which use Verilog. Though they have been able to build more complex algorithms in a smaller time period than us, it can be seen that if the target is to build a real-world optimized processor, our slow Verilog based toolflow will probably give a more compact, more efficient and faster core.

Table 1- Resource Utilization and Comparison with SHAKTHI processor

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Resource | Our design @8KB | | Cache @64KB | | Shakthi C class | |
|  |  | % |  | % |  | % |
| Slice LUTs | 5966 | 11.21 | 6681 | 12.56 | 16246 | 30.54 |
| Slice Registers | 3948 | 3.71 | 3947 | 3.71 | 5874 | 5.52 |
| F7 Muxes | 107 | 0.4 | 123 | 0.46 | 1605 | 6.03 |
| F8 Muxes | 32 | 0.24 | 32 | 0.24 | 605 | 4.55 |
| BRAM (36Kb) | 18 | 12.86 | 32 | 22.86 | 44.5 | 31.79 |
| DSPs | 0 | 0 | 0 | 0 | 7 | 3.18 |

### 3.2 Power Consumption and Frequency (in Zedboard)

The power utilization reports obtained for the two processors are also indicative of our superior efficiency (see Table 2). Furthermore after implementation we found that the operating frequency of the Shakthi C class core is at only 40MHz for a Zynq Zedboard and 48MHz for an Artix-7 FPGA. In comparison our core achieves a much higher clock frequency of 125MHz for Zynq Zedboard and 140MHz for an Artix-7 FPGA. Similar reasons as above can be provided for the difference between the two designs.

Table 2 - Power Consumption

|  |  |  |
| --- | --- | --- |
| Power consumption | Our design 64KB | Shakthi C class |
| Clocks | 0.029W | 0.032W |
| Signals | 0.037W | 0.053W |
| Logic | 0.028W | 0.059W |
| BRAM | 0.033W | 0.021W |
| DSP |  | 0.007W |
| IO | 0.019W | 0.013W |

### 3.3 Testing and Verification

For a processor design as complex as ours, it is extremely important that we verify the functionality of each and every unit. The verification process for our processors was done in a number of stages.

* As it was being designed each sub-unit was tested for functionality using behavioural simulations
* After integration into larger modules, a more comprehensive testing procedure was carried out. The functionality of the two caches and the functionality of the pipeline without caches were carried out in this fashion.
* After full integration into a standalone IP core another round of comprehensive behavioural simulation was carried out.
* After implementation another round of testing was carried out, running simulations in-hardware using the Zynq Zedboard.

These numerous tests provided us with validation that our design is running at the accuracy we expected. Finally several programs were given to the processor to verify functionality. The outputs from the RISC-V processors was compared with the outputs from a computer which ran the same program. Each and every program produced exactly the same result in both processors, verifying that our design is indeed correct.

Following are the programs we tested. C codes of the below mentioned programs are attached in the Appendix,

1. Add – Addition of two numbers

2. Add function – Add as a function call

3. Array – Instantiate an array of 10 numbers

4. Convolution – Do simple convolution for a sample of 9 by a kernel of size 3

5. Convolution func – Convolution with function call, to test function calls and returns

6. Count – Count 10 numbers

7. Histogram – Generate a 5 bin histogram from a100 number dataset

8. Large convolution – Convolve an array of 100 with a kernel of 1x5

9. Matrix mult – Multiply 1x3 matrix with 3x2 matrix

10. Large matrix multiplication – Multiply 2 matrices of 10 x 10 in size

11. Sort – Sort array of 10 numbers

12. Multiplication – Multiplication of two numbers

**3.4 Future Work and Improvements**

RISC V based ISA processor that we have built for our final year project achieves many of the targets we have planned for at the onset, but nevertheless it needs to be developed further including new features to have a hope of competing with the popularly used processors in the industry. Though we have taken the initial few steps, there is still a long way to go. We hope to achieve following features in our future versions in order to go towards this target.

* *Include multiplication and division operations*

Currently developed version does not contain multiplication and division operations. But the processor is capable of performing multiplication by bit shifting. A hardware approach to multiplication will give a significant performance advantage over a software approach that is currently used. When developing the system, we have developed it with the flexibility to extend the design to multiplication and division operation. We have included logic which will make it easy to add multiplication and division with use of less resources. Basically since a multiplication instruction does not use the data cache, the three pipe stages DM1, DM2, DM3 can be directly utilized for a multicycle, pipelined multiplier/divider. Only small modifcations in the forwarding control unit needs to be done to integrate the new modules. Therefore, resource utilization will not increase drastically due to modification.

* *Achieve expected speed of 250 MHz*

Current system runs at an operstion speed of 120MHz. But our intention is to achieve 250MHz. Most of the sections in our design works at this speed. But the critical path lies within the instruction execution stage (EX stage). This can be overcome by pipelining the EX stage into two substages. The control units will have to be changed slightly to accomadate this modification, therefore the resource utilization might increase slightly, but we believe that a very high frequency increase would be achievable by this step.

* *Design better branch predictors*

The cost of designing a very deeply pipeline as in our processor is that many cycles will be wasted on running useless instructions in the case of a branch misprediction. The currently used strategy of predicting all branches as not-taken is not enough for a very high performance execution. Therefore the design and integration of a more complex branch predictor which can achieve a very low misprediction rate is a crucial future milestone that should be targeted.

* *Provide virtual memory capabilities*

In the current iteration our cache architecture uses a physically mapped, physically tagged memory design. But for the processor to run a OS like Linux, it is required to have a virtual memory and paging capability. Therefore we have to do improvements to the cache architecture to support virtual memory by including a Translate Look Aside buffer and other necessary infrastructure.

* *Extend to floating point operations*

System is capable of performing integer operations at the moment. At the next version we hope to update the system to floating point operations too.

* *Implement CSR instructions*

Currently the processor is built based on RISC V base ISA. There are 48 instructions and we have included 42 instructions to our design. The left 6 instructions are related to CSR instructions which are capable of handling virtual memory. Virtual memory is the memory that appears to exist as main storage although most of it is supported by data held in secondary storage, transfer between the two being made automatically as required.

* + **CSRRW:** Instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd.
  + **CSRRS:** Instruction reads the value of the CSR, zero extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that species bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable.
  + **CSRRC:** Instruction reads the value of the CSR, zero extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. That is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable.
  + **CSRRWI, CSRRSI, and CSRRCI:** The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit immediate (imm[4:0]) which is encoded in the rs1.

For the realization of CSR instruction a Control State Register must be defined in hardware the assigning of values will be automatically developed by the compiler. And a separate region of memory is assigned for CSR storage in case of a state change the relevant CSR will be loaded via the CSR instructions.

Example implementation of CSR instructions can also be found in SHAKTHI processors for reference.

* *Implement in ASIC platform*

Development of the processor is done in RTL level. This is the front end design of the processor. After building the processor completely, it will be built as an ASIC (Application Specific Integrated Circuit) development. This can be said as the final stage of developing the processor. Floor planning, power management, area management are the core sections which we need to pay attention. Finally, the design is fabricated on wafer and packaged as an IC, a hard core to the market.

# Chapter 4

CONCLUSION

The RISC-V Base ISA processor project was started with the aim of building an open source processor microarchitecture implementing the RISC-V 32I instruction set. At the onset of the project, our goal was a working, performance optimized processor which could operate at high frequency while also providing a significant instruction throughput. Implementing the architecture in a platform independent manner, avoiding the use of third-party IPs and thereby avoiding any licence issues, and building an extensible architecture into which further improvements like multiplication, division and floating point modules can be easily integrated were some of the secondary goals outlined at the beginning. After implementation in a FPGA evaluation platform, it was also expected to compare the performance, utilization and clock frequency of our design against similar already available microarchitectures.

After almost a year of hard work on the project we could be satisfied that most of the objectives given at the onset has been achieved. A 10-stage deeply pipelined processor was designed, implemented, tested and verified in the Zynq Zedboard Evaluation Platform, achieving a 120MHz operation frequency, while also having design paths still open which could potentially increase it to over 200MHz. In comparison with other processor cores like Shakthi it had achieved higher operating frequency as well as lower resource utilization by a significant amount. By analysing the architecture and the test results obtained by running several RISC-V assembly programs, it could be claimed that the high throughput goal was also mostly achieved, in that the processor retires an instruction each cycle as long as branch misprediction and cache miss does not occur. With the integration of a better and more complex branch predictor, even higher throughput is easily foreseeable. Additionally, the cache architecture, which had many critical problems in its design time, had been refined into a reliably functioning, high performance memory architecture, with advanced capabilities such as prefetching, stream buffers, victim caches, critical word first and early restart operation etc.

Even though we have come a long way with the progress we have shown upto now, there is still a long way more to go before the RISC-V processor could be a worthwhile competitor in the processor industry. Many improvements and extensions to the current architecture has been proposed, both by ourselves and by other RISC-V researchers who had given input on our architecture, some of which include,

* Implementing an advanced branch predictor
* Implementing virtual memory support for the two caches
* Inclusion of CSR instructions
* Integrating modules for multiplication and division extensions

Some of these changes had been already anticipated in the initial design, in which case the extensions can be fairly easily integrated to the existing design. Others might require quite a bit of clever engineering to properly integrate, but major revisions to the architecture will most likely not be necessary.

The Verilog approach we had been taking in building the current architecture has been the main reason for the utilization and frequency advantages we had been having. So following the same method of building the units carefully from ground up is recommended. In conclusion, it can be said that with enough effort in the subsequent years it will be entirely within capability to produce a high performance, efficient processor IP Core which could compete with other RISC-V, x86 and other ISA processors in the market.

# REFERENCES

[1] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic, “The RISC-V instruction set,” in 2013 IEEE Hot Chips 25 Symposium (HCS), Aug 2013, pp. 1–1.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, “The RISCV instruction set manual, volume 1: User-level ISA, version 2.1,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[3] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, “The RISC-V instruction set manual volume 2: Privileged architecture version 1.9,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129, Jul 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

[4] A. S. Tanenbaum and J. R. Goodman, Structured Computer Organization, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[5] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design), 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A quantitative Approach, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[7] C. Celio. ”Educational microarchitectures for RISC-V ISA”. University of California, Berkeley. [Online]. Available: https://github.com/ucb-bar/riscv-sodor

[8] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P. F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic, “An agile approach to building RISC-V microprocessors,” IEEE Micro, vol. 36, no. 2, pp. 8–20, Mar 2016.

[9] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovi, and K. Asanovi, “A 45nm 1.3ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators,” in European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th, Sept 2014, pp. 199–202.

[10] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtic, B. Keller, S. Bailey, M. Blagojevic, P. F. Chiu, H. P. Le, P. H. Chen, N. Sutardja, R. Avizienis, A. Waterman, B. Richards, P. Flatresse, E. Alon, K. Asanovi?, and B. Nikoli?, “A RISC-V vector processor with tightly-integrated switched-capacitor DC-DC converters in 28nm FDSOI,” in 2015 Symposium on VLSI Circuits (VLSI Circuits), June 2015, pp. C316–C317.

[11] V. Patil, A. Raveendran, P. M. Sobha, A. D. Selvakumar, and D. Vivian, “Out of order floating point coprocessor for RISC V ISA,” in VLSI Design and Test (VDAT), 2015 19th International Symposium on, June 2015, pp. 1–7.

[12] ”Rocket chip generator”. University of California, Berkeley. [Online]. Available: https://github.com/ucb-bar/rocket-chip

[13] ”PULPino”. ETH, Zurich. [Online]. Available: https://github.com/ucb-bar/rocket-chip

[14] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, “SHAKTI processors: An open-source hardware initiative,” in 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), Jan 2016, pp. 7–8.

[15] Intel, Intel 64 and IA-32 Architectures Software Developer’s Manual. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[16] ARM-Holdings, ARM Architecture Reference Manual. [Online]. Available: https: //www.scss.tcd.ie/∼waldroj/3d1/arm arm.pdf

[17] Cortex-M7 Technical Reference Manual. [Online]. Available: <http://infocenter>. arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B cortex m7 trm.pdf

[18] Xilinx, MicroBlaze Processor Reference Guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw manuals/mb ref guide.pdf

[19] Altera, Nios II Classic Processor Reference Guide. [Online]. Available: https://www.altera.com/en US/pdfs/literature/hb/nios2/n2cpu nii5v1.pdf

Appendix

Verification Programs used to test the processor

**1. Add**

#include <stdio.h>

int main(){

int a = 1;

int b = 2;

\*(int\*)0x11FF8 = a+b;

while(1){}

return 0;

}

**2. Add function**

#include <stdio.h>

int add(int a, int b);

int main(){

int a = 1;

int b = 2;

//\*(int\*)0x11FF8 = a+b;

//int c = add(a,b);

\*(int\*)0x11FF8 = add(a,b);

//printf("%d \n", c);

while(1){}

return 0;

}

int add(int a, int b){

return a+b;

}

**3. Array**

#include <stdio.h>

int main () {

int n[ 10 ];

int I;

for ( i = 0; i < 10; i++ ) {

n[ i ] = i;

\*(int\*)0x11FF8 = n[ i ];

}

while(1){}

return 0;

}

**4. Convolution**

#include <stdio.h>

#include <stdlib.h>

int main(){

int i,j;

int k;

int sampleCount = 9;

int kernelCount = 3;

int x[9] = {1,2,3,4,5,6,7,8,9};

int y[9] ;

int h[3] = {1,2,1};

for ( i = 0; i < sampleCount; i++ ){

y[i] = 0; // set to zero before sum

for ( j = 0; j < kernelCount; j++ ){

if((i-j)<0){continue;}

y[i] += x[i - j] \* h[j]; // convolve: multiply and accumulate

}

}

for(k=0;k<sampleCount;k++){

\*(int\*)0x11FF8 = y[k];

}

while(1){}

return 0;

}

**6. Count**

#include <stdio.h>

int main(){

int i=0;

int count=0;

for(i=0;i<10;i++){

count=i;

\*(int\*)0x11FF8 = count;

}

while(1){}

return 0;

}

**7. Histogram**

#include <stdio.h>

int main(){

int temp;

int address = 0x10370;

int i;

int sum=0;

for(i=0;i<100;i++){

temp = \*(int\*)address;

sum+=temp;

address+=0x04;

}

\*(int\*)0x11FF8 = sum;

int min=20000;

address = 0x10370;

for(i=0;i<100;i++){

temp = \*(int\*)address;

if(temp < min){

min = temp;

}

address+=0x04;

}

\*(int\*)0x11FF8 = min;

int max=0;

address = 0x10370;

for(i=0;i<100;i++){

temp = \*(int\*)address;

if(temp > max){

max = temp;

}

address+=0x04;

}

\*(int\*)0x11FF8 = max;

address = 0x10370;

int his0 = 0, his1 = 0, his2 = 0, his3 = 0, his4 = 0;

for(i=0;i<100;i++){

temp = \*(int\*)address;

if(0<=temp && temp<2000){

his0++;

}

if(2000<=temp && temp<4000){

his1++;

}

if(4000<=temp && temp<6000){

his2++;

}

if(6000<=temp && temp<8000){

his3++;

}

if(8000<=temp && temp<10000){

his4++;

}

address+=0x04;

}

\*(int\*)0x11FF8 = his0;

\*(int\*)0x11FF8 = his1;

\*(int\*)0x11FF8 = his2;

\*(int\*)0x11FF8 = his3;

\*(int\*)0x11FF8 = his4;

while(1){}

return 0;

}

**8. Large convolution**

#include <stdio.h>

#include <stdlib.h>

int main(){

int i,j;

int sampleCount = 100;

int kernelCount = 5;

int address = 0x10370;

int tem=0;

int tempAddress = 0;

int y=0 ;

int h[5] = {1,2,3,2,1};

for ( i = 0; i < sampleCount-4; i++ ){

y = 0;

tempAddress = address+ (4\*i);

for ( j = 0; j < kernelCount; j++ ){

tem = \*(int\*)(tempAddress + (4\*j));

y += tem \* h[j]; // convolve: multiply and accumulate

}

\*(int\*)0x11FF8 = y;

while(1){}

return 0;

}

**9. Matrix multiplication**

#include<stdio.h>

int main() {

int i, j, k;

int sum = 0;

int a[1][3]={{1,4,6}};

int b[3][2]={{2,3},{5,8},{7,9}};

//Multiplication Logic

for (i = 0; i < 1; i++) {

for (j = 0; j < 2; j++) {

sum = 0;

for (k = 0; k < 3; k++) {

sum = sum + a[i][k] \* b[k][j];

}

\*(int\*)0x11FF8 = sum;

}

}

while(1){}

return (0);

}

**10. Large matrix multiplication**

#include<stdio.h>

int main(){

int i,j,k;

int addressA = 0x10370;

int addressB = 0x10370 + 400;

int valA,valB;

int temp;

for(i=0;i<10;i++){

for(j=0; j<10;j++){

temp = 0;

for(k=0;k<10;k++){

valA = \*(int\*)(addressA + (40\*i) + (4\*k));

valB = \*(int\*)(addressB + (40\*k) + (4\*j));

temp += valA\*valB;

}

\*(int\*)0x11FF8 = temp;

}

}

while(1){};

return 0;

}

**11. Sort**

#include <stdio.h>

#include <stdbool.h>

int main() {

int list[10] = {1,8,4,6,0,3,5,2,7,9};

int temp;

int i,j,k;

bool swapped = false;

for(i = 0; i < 10-1; i++) {

swapped = false;

for(j = 0; j < 10-1-i; j++) {

if(list[j] > list[j+1]) {

temp = list[j];

list[j] = list[j+1];

list[j+1] = temp;

swapped = true;

}

}

if(!swapped) {

break;

}

}

for(k = 0; k < 10-1; k++){

\*(int\*)0x11FF8 =list[k];

}

while(1){}

return 0;

}

**12. Multiplication**

#include <stdio.h>

int main(){

int a = 11;

int b = 14;

\*(int\*)0x11FF8 = a\*b;

while(1){}

return 0;

}