

Configurable Neuromorphic Processor Architecture for Spiking Neural Networks

ME 420 Mechanical Engineering Individual Research Project

Semester 8 : November 2023

By

R.M.T.N.K Rathnayaka (E/17/286)

Department of mechanical Engineering

Faculty of Engineering

University of Peradeniya

Sri Lanka

**Configurable neuromorphic processor architecture for spiking neural networks**

by

R.M.T.N.K Rathnayaka

Supervised by

Dr. J.C Rajaguru

Dr Isuru Nawinne

**Declaration**

I declare that this report does not incorporate, without acknowledgement, any material previously submitted for any other Degree or Diploma to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due references are made. It has not been accepted for any other course and is not being concurrently submitted to any other person.

Signature of candidate Date: 09/11/2023

R.M.T.N.K Rathnayaka

Name of candidate

....................................................

Countersigned by:

Dr J.C Rajaguru

Name of Supervisor Date: 09/11/2023

..................................................

Signature of supervisor

Dr Isuru Nawinne

Name of Supervisor

..................................................

Signature of supervisor

**Summary**

Neuromorphic processors are based on an architecture that is inspired by the biological brain that uses neurons and synapses as the basic building blocks. Researchers have made various discoveries with the hope of exploiting the huge parallelism and the energy efficiency of spiking neural dynamics of biological brains. This is a review that tries to state the background and the prior implementations of the neuromorphic architecture that could be developed on top of highly parallelized network on chip architectures with von Neumann processing cores.

**Acknowledgement**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my/our own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments

R.M.T.N.K Rathnayaka

Department of mechanical engineering

University of Peradeniya.

09/11/2023

**Nomenclature**

|  |  |
| --- | --- |
| SNN | Spiking Neural Networks |
| NOC | Network On Chip |
| FPGA | Field Programmable Gate Array |
| CNN | Convolutional Neural Networks |
| RNN | Recurrent Neural Networks |
| ANN | Global Positioning System |
| CAE | Computer Aided Engineering |

Content

**Declaration**

**Summary**

**Acknowledgement**

**Nomenclature**

[1 Introduction](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024326) 1

[1.1Background](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024327) 3

[1.2Motivation](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024328)

[1.3Summary of outcomes](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024328)

[2 State of Art](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024329)

[2.1RISC V Pipilined CPU](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024330)

[2.2Neuron moddels](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024331)

[2.3Neromorphic architectures](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024332)

3 Methodology

4.1Examples of reference listings

4.2Parenthetical **citations**

[4 RISC V Processor](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024335)

[3.1Pipelineed datapath](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024336)

[3.2RV32IM](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024337)

[3.2RV32IM F](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024337)

[3.2Interrupts](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024337)

[5 Neuron Bank](file:///C:\\Users\\twintech\\Downloads\\Report-format-template.doc" \l "_Toc306024338)

[4.1Neuron model](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024339)

[4.2Neuron Banks](file:///C:\Users\twintech\Downloads\Report-format-template.doc#_Toc306024340)

6 NoC implemetation

4.1NoC Architectures

4.2Router

4.2Network interface

7 SNN Software implemetation

4.1SNN application

4.2Test nuron application software

8 Testing

4.1Test with RV32IMF

4.2Test with neurmorphic hardware

9 References

4.1Reference listings

4.2Parenthetical **citations**

# Introduction

## Background of the project

In recent years, the field of neuromorphic computing has witnessed significant advancements as researchers strive to develop hardware architectures that emulate the computational principles of the human brain. Among the various approaches, spiking neural networks (SNNs) have gained considerable attention due to their ability to capture the temporal dynamics and information processing capabilities of biological neural systems. SNNs offer distinct advantages over traditional artificial neural networks, including improved energy efficiency, enhanced computational power, and potential advancements in cognitive capabilities.

Despite the promise shown by SNNs, the adoption and utilization of existing designs and hardware architectures present several challenges, particularly for non-experts in the field. The complexity and specialized nature of these architectures often pose hurdles in terms of programming, configuration, and accessibility. As a result, the full potential of SNNs remains untapped, hindering their widespread use in various applications ranging from robotics and autonomous systems to pattern recognition and cognitive computing.

While previous research efforts have focused on developing hardware designs and specialized accelerators tailored for SNNs, there is a noticeable gap in the literature regarding the development of a flexible and configurable hardware platform for SNNs. Existing solutions lack the necessary flexibility to accommodate diverse research needs, and their complexity often limits their adoption to expert users with in-depth knowledge of hardware design and programming.

Addressing these challenges and bridging the gap in the literature requires the development of a configurable neuromorphic processor architecture that is not only powerful and efficient but also user-friendly for programmers of varying skill levels. Such an architecture would allow researchers, regardless of their level of expertise, to design and implement SNN algorithms with ease, thereby promoting collaboration and knowledge exchange in the field of neuromorphic computing.

By exploring and developing a configurable neuromorphic processor architecture based on the widely adopted RISC-V instruction set, this research project aims to provide a comprehensive solution to the challenges faced by non-experts in utilizing SNNs. The proposed architecture intends to offer a flexible and accessible platform that enables researchers to experiment with different configurations of neuromorphic architectures, facilitating innovation and advancements in the field of cognitive computing.

Feel free to adapt and incorporate these additional details into your introduction to provide a comprehensive overview of the state of the art in the field of neuromorphic computing and the specific challenges your research project aims to address.

## 1.2 Motivation

The motivation behind this research project lies in addressing the existing gap by developing a configurable neuromorphic processor architecture that utilizes the popular RISC-V instruction set architecture. By doing so, we aim to create a hardware platform that is not only powerful and efficient but also accessible to programmers with varying levels of expertise. This platform will enable non-experts to effectively design and implement SNN algorithms, expanding the user base and fostering collaboration among researchers in the field.

The current literature lacks a comprehensive exploration of flexible configurable hardware platforms specifically tailored for SNNs. While there are existing hardware designs and specialized accelerators, a need remains for a versatile and user-friendly platform that allows researchers and developers to explore and experiment with different configurations of neuromorphic architectures. This research project aims to bridge this gap and provide a flexible configurable hardware platform for SNNs, offering a broader range of users the opportunity to engage in neuromorphic computing research.

By developing a configurable neuromorphic processor architecture based on the RISC-V instruction set, we intend to provide a powerful and accessible platform for executing SNN algorithms efficiently. This research project seeks to contribute to the field of neuromorphic computing by enabling researchers, regardless of their level of expertise, to design and implement SNNs with ease, opening new possibilities for innovative applications and advancements in cognitive computing.

## Methodology

Proposed methodology is as following steps

1. Literature Survey

* Conduct a comprehensive search on spiking neural networks and existing neuromorphic architectures, identifying available architectures and pinpointing gaps in the current literature

1. Implementing a RV32IMF pipelined CPU

* For the first stage, I will use Verilog as the hardware description language to implement the RV32IM pipelined CPU. This will serve as a foundation for the design and will allow us to build upon existing RISC-v architecture.

1. Add custom Hardware For Neuromorphic computation
2. Implementing NOC (Network on Chip) (with Neuron Bank )

* For the second stage, I will use FPGA to implement the Neuron bank and NOC for SNNs and integrate it into the processor architecture. This will enable us to develop configurable neuromorphic processor architecture for spiking neural networks.

1. Verification the functionally of the hardware with FPGA
2. Evaluate the power consumption and speed of the configurable neuromorphic processor architecture and compare it with existing solutions in the literature

.

## Summary of Outcomes

Table 1.1 Summary of Literature Survey Outcomes

|  |  |  |  |
| --- | --- | --- | --- |
| Title of the Paper | Author | Outcome | Conclusions |
| RISC-V Based Network on Chip Architecture for Spiking Neuron Processing | Heshan Dissanayake Buddhi Perera Dinindu Thilakarathne Department of Computer Engineering University of Peradeniya | A Review on Neuromorphic Architecture  Implementation. RISCV NOC implementation for SNN. have tested the Izhikevich model inside one core. | Verify the high performance and energy efficiency of RISCV NOC implementation |
| A Survey of Neuromorphic Computing and Neural Networks in Hardware | C. D. Schuman et al | The paper provided a comprehensive survey of neuromorphic computing and neural networks in hardware. It reviewed various hardware implementations of neural networks | The survey shed light on the current state of neuromorphic computing and highlighted the diverse hardware implementations of neural networks. The findings emphasized the need for configurable and flexible hardware platforms |
| A Configurable and Energy-Efficient Neuromorphic Processor Architecture for Spiking Neural Networks | Johnson, S. et al. | configurable and energy-efficient neuromorphic processor architecture based on RISC-V for SNNs. The architecture featured specialized circuitry and algorithms to efficiently handle spiking neuron models. | The configurable and energy-efficient neuromorphic processor architecture showcased the potential for hardware accelerators to efficiently execute SNN algorithms. |

# Spiking Neural Networks and Neuromorphic Architecture Implementations

### **2.1. Spiking Neural Networks**

An Artificial Neural Network (ANN) is a computing system inspired by the biological neural networks that exist within the brain in order to solve complex tasks. An ANN is essentially a collection of artificial neurons connected together to form a network using links that imitate synapses in the brain A Spiking Neural Network (SNN) is a relatively new variety of ANNs that more closely resembles actual biological neural networks by incorporating the concept of time, with neurons transmitting information during neuron spike events rather than at each propagation cycle as is typically the case with ANNs. Due to this inherent event-driven nature, SNNs of- fer higher energy efficiency and a greater degree of parallelism in computations. However, it is the same event-driven nature that hinders the realisation of SNNs using general-purpose CPU and GPU hardware, and specialised hardware with native capabilities for simulating neuron spike events as well as hardware support for large-scale parallelism is required.

This is the primary motivation for the development of neuromorphic architectures. In the exploration of SNNs, various neuron models have been developed, ranging from biologically plausible models such as the Hodgkin-Huxley model to simplistic mod- els such as the integrate-and-fire model. Biophysically accurate models are prohibitively expensive in terms of the computational power required and it is difficult to simulate more than a handful of such neurons using currently available hardware. In contrast, simple models such as the integrate- and-fire model are computationally efficient but they are too simple and not realistic enough to imitate the rich spiking and bursting dynamics of natural neurons. Biologically- inspired models such as the Izhikevich model and the Fitzhugh-Nagumo model offer a compromise between the two extremes wherein the model is complex enough to sufficiently emulate the behaviour of natural neurons while also being computationally effective.

### **2.2. Neuromorphic Architecture Implementations**

Given that the development of neuromorphic architectures is an active area of research, there have been several proposed implementations with the intention of simulating and accelerating SNNs.The most prominent architecture is a massive multi-core processor network purpose-built for accelerating SNN workloads named SpiNNaker developed by the Advanced Processor Technologies Research Group (APT) at the Department of Computer Science, University of Manchester. The SpiNNaker system utilises a network of 57600 nodes with each node consisting of 18 ARM968 cores and it is being used as the neuromorphic computing platform for the Human Brain Project. It is implemented with a globally asynchronous locally synchronous (GALS) routing mechanism to allow for energy-efficient and scalable inter-node communications due to the sheer number of nodes. While SpiNNaker boasts impressively high performance and parallelism, it is an extremely expensive, large-scale project requiring 100 kW of power from a 240 V supply and an air-conditioned environment. DYNAP (Dynamic Neuromorphic Asynchronous Processor), by Moradi, S. et al., is yet another neuromorphic proces- sor architecture that boasts high scalability with heterogeneous memory structures for minimising memory utilisation. This design focuses primarily on the interconnect between the processor nodes and proposes a hierarchical routing mechanism for making the event-based communication much more efficient. DYNAPs have been demonstrated to achieve high scalability with minimal memory requirements in simulating a three-layer convolutional neural network. Further, the OpenSpike project by Modaresi, F. et al. and research by Zhang, J. et al. demonstrate the utilisation of Application-Specific Integrated Circuits (ASICs) in developing neuromorphic architectures. Both implementations adopt the leaky integrate-and-fire neuron model at the hardware level along with a network implementation to provide the interconnect between the hardware neurons through event-driven updates. Such ASIC implementations of artificial neurons offer excellent performance but lack the flexibility in programming and configuration offered by higher-level architectures.

Another approach to neuromorphic architectures is demon- strated by Urgese, G. et al. in the ODIN coprocessor for accelerating SNN workloads in Internet-of-Things (IoT) applications. The designed architecture consists of a reconfigurable neuromorphic coprocessor which interfaces with a RISC-V based System on Chip (SoC) via the Serial Peripheral Interface (SPI), allowing the RISC-V core to configure and offload SNN tasks to the ODIN coprocessor. ODIN supports the leaky integrate-and-fire model and a custom Izhikevich inspired neuron model.

The POETS (Partial Ordered Event Triggered Systems) architecture by Shahsavari, M. et al. [15] is another ongoing research project which focuses on building a customizable hardware platform for event-driven parallel programming. The POETS system is a large parallel processing cluster where the SNN is configured using a graph schema and it is capable of simulating both the leaky integrate-and-fire neuron model as well as the Izhikevich model. Part of the POETS project is the development of a hyperthreaded RISC-V core named Tinsel , by Naylor, M. et al., specifically designed to work in tandem with the POETS system to reduce latency. Although numerous such neuromorphic architectures exist, there is a lack of designs that exploit the programming flexibility and platform maturity of existing architectures. Granted that highly-specialised hardware as presented in existing designs deliver very high performance, the authors of this paper believe that a neuromorphic architecture based on the RISC- V architecture, which offers greater flexibility, will yield a valuable compromise.

# RISCV Core (RV32IMF)

## 4.1 Introduction

In the course of this project, we have successfully conceived and implemented a sophisticated five-stage pipeline CPU, meticulously aligned with the RISC-V architecture, specifically adhering to the RV32IMF specification. This architectural framework operates on a robust 32-bit data path, showcasing a comprehensive suite of capabilities that encompasses support for Integer operations, multiplication operations, and intricate floating-point operations.

What sets our CPU design apart is the thoughtful integration of additional features, namely an Interrupt controller and a Random number generation unit, enhancing its versatility and applicability in various computing scenarios. The implementation process was carried out using Verilog, a hardware description language, and rigorous testing procedures were conducted on both Altera DE2 and DE5 boards, ensuring the robustness and reliability of the CPU across different platforms.

To validate the functionality and performance of our CPU, extensive testing scenarios were meticulously executed. This included evaluating its response to Integer operations, assessing the efficiency of multiplication operations, and scrutinizing the precision and accuracy of floating-point operations. The incorporation of an Interrupt controller adds a layer of sophistication, providing the CPU with the capability to seamlessly handle external interruptions and enhance overall system responsiveness.

## 4.2 RISCV ISA

### 4.2.1 Introduction

The ever-increasing demand for efficient and customizable processors has driven the development of new open-source instruction set architectures (ISAs). Among these, the RISC-V (RISC Five) architecture has gained significant attention and adoption due to its simplicity, modularity, and extensibility. In this report, we present the design and implementation of a RV32IMF RISC-V processor using Verilog, which adheres to the RISC-V base integer instruction set (RV32I) and includes support for the multiplication and division extension (RV32M) and for floating points extension (RV32IMF)The RISC-V architecture is an open-source ISA that originated from the University of California, Berkeley, and has since gained widespread acceptance across academia and industry. Its simplicity and modular design make it an ideal choice for educational projects, allowing students and researchers to explore the intricacies of processor design and understand the fundamental principles behind modern computing.

### 4.2.2 Instructions

There are 43 RV32I instructions , 9 instruction in RV32M extension and 23 instruction for RV32F standard extensions . since the CPU supports for total 75+ instructions. They s are organized into six formats: R-type, I-type, S-type, B-type, U-type, and J-type. The R-type format is used for instructions that operate on two registers and produce a result in a third register. The I-type format is used for instructions that operate on a register and an immediate value and produce a result in a register. The S-type format is used for instructions that store a value from a register into memory. The B-type format is used for branch instructions. The U-type format is used for instructions that load an immediate value into a register. The J-type format is used for jump instructions .

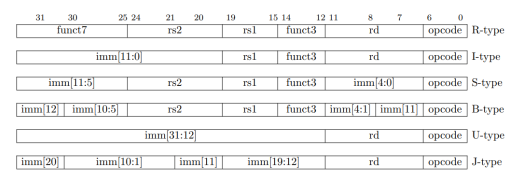
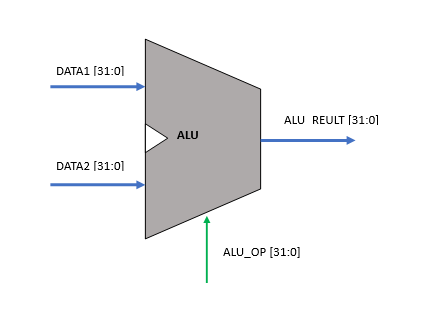


Figure 02:RV32IMF ISA Instruction Encoding format

## 4.3 Modules

### 4.3.1 ALU (Arithmetic and Logic Unit)

The Arithmetic Logic Unit (ALU) stands as a fundamental component within the heart of a computer's central processing unit (CPU). Its primary role is to perform arithmetic and logical operations on binary data, thereby enabling the CPU to execute a broad spectrum of computational tasks. An ALU takes input operands, usually sourced from registers, and carries out operations like addition, subtraction, multiplication, division, logical AND, OR, XOR, and bitwise shifts. The ALU's significance lies in its ability to execute these operations swiftly and efficiently, forming the computational backbone of the processor.



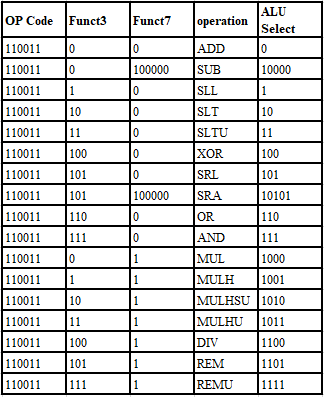
The versatility of the ALU makes it a critical element in the execution of instructions, contributing to the overall performance and functionality of a computer system. Its presence and efficiency are key factors in determining the processing power and capabilities of a CPU, influencing the speed and responsiveness of the entire computing device.

This ALU supports 18 integer operations. For floating points operation there are separate ALU called FPU(Floating point unit ) We decided to take the shifting unit inside the ALU and also we have moved the branching unit outside the ALU. According to our design the CPU gets two 32 bit operands as inputs to the ALU. Also, it takes a 5 bit select signal to the ALU to select the operation. And its output is a single 32 bit result. Following tables describes the operation name and there functionalities.

TABLE 1: INTEGER OPERATIONS

|  |  |
| --- | --- |
| **ALU operation** | **Description** |
| ADD | Addition |
| SUB | Subtraction |
| SLL | Shift Left Logical |
| SLT | Set Larger Than |
| SLTU | Set Larger Than Unsigned |
| XOR | XOR Operation |
| SRL | Shift Right Logical |
| SRA | Shift Right Arithmetic |
| OR | OR Operation |
| AND | AND Operation |
| MUL | Multiplication |
| MULH | Return upper 32 bits of result of the Multiplication (signed x signed) |
| MULHSU | Return upper 32 bits of result of the Multiplication (signed x unsigned) |
| MULHU | Return upper 32 bits of result of the Multiplication (unsigned x unsigned) |
| DIV | Division |
| REM | Signed remainder of integer division |
| REMU | Unsigned remainder of integer division |
| FWD | Additional Instruction built to support |
| other instructions (Not included in |
| RV32IM) |

TABLE 1: INTEGER OPERATIONS



### **4.3.2 FPU Floating point unit**

There are separate unit for the do the floating points operations there are 23 floating point operation

**Table of floating points**

### 4.3.3 Register file

The Register File is a crucial hardware unit in the RV32IM Datapath that serves as a fast and accessible storage location for operands and intermediate results during instruction execution. It consists of a collection of registers, each capable of storing a fixed number of bits of data.

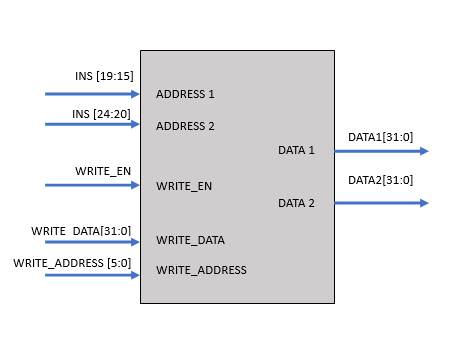
The Register File facilitates efficient and quick access to operands required by instructions during the execution stage. Instructions fetch operands from the Register File, perform computations using these operands, and store the results back into registers.

Figure 06:RV32IM register File

* Register file contains **32** registers and the size of a register is **32 bits.** Register **x0** is set to zero by making all the bits in register **x0** to 0. Registers **x1** to **x31** can be used by the instructions.

In RISC-V ISA **NOP** instruction is encoded as an ADDI instruction. In this case the register x0 containing the value zero will be used

### **4.3.4 Control unit**

The control unit is a crucial component of any processor as it manages the flow of data and instructions between the different functional units of the processor. In the RV32IM processor, the control unit is responsible for generating control signals that control the operation of the different functional units, including the instruction fetch unit, register file, ALU, and memory unit.

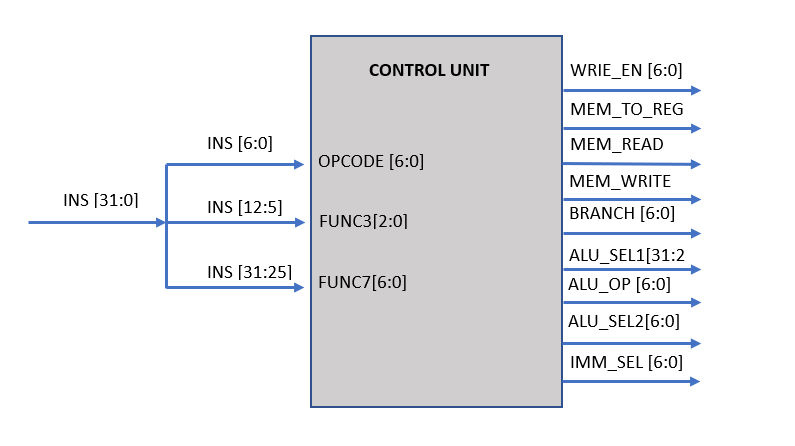


Figure 04:RV32IM Control Unit

Inputs‌ ‌to‌ ‌the‌ ‌control‌ ‌unit‌ ‌are,

* OPCODE [6:0]
* FUNCT3[2:0]
* FUCNT7[6:0]

Outputs‌ ‌generated‌ ‌from‌ ‌the‌ ‌control‌‌ unit‌ ‌are,‌

* ‌IMM\_SEL[2:0]‌ ‌
* ALU\_SEL1‌
* ‌ ALU\_SEL‌ ‌2
* ALU\_OP[4:0]‌
* BRANCH [2:0]‌ ‌
* MEM\_WRITE‌
* ‌MEM\_READ‌ ‌
* WB\_SEL[1:0]‌ ‌
* WRITE\_EN

**IMM\_SEL [2:0]‌ ‌**

This control signal is for the immediate value generation unit. In RISC-V ISA, according to the ordering of the immediate value bits there are 7 variants.

* U - Type
* J - Type
* S - Type
* B - Type
* I - Type signed
* I - Type containing shift amount
* I - Type unsigned

The immediate value generation unit will generate these 7 types of immediate values and the IMM\_SEL control signal will select the relevant immediate value.

**ALU\_SEL‌ 1**

The input operand 1 of the ALU unit is of 2 values.

* PC value - For AUIPC, JAL, B - Type instructions
* DATA1 (value from the register file) - For all the other remaining instructions

This control signal will select between these two values. This is a 1 - bit control signal.

**ALU\_SEL‌ 2**

The input operand 2 of the ALU unit is of 2 values.

* DATA2 (value from the register file) - For R - Type instructions
* Immediate value - For all the other remaining instructions

This control signal will select between these two values. This is a 1 - bit control signal

**ALU\_OP [4:0]‌**

This signal will select the relevant ALU operation out of the 18 ALU operations. This is a 5 - bit control signal and the used for selecting following ALU operations.

* AND
* MUL
* MULH
* MULHU
* MULHSU
* DIV
* DIVU
* REM
* REMU
* ADD
* SUB
* SLL
* SLT
* SLTU
* XOR
* SRL
* SRA
* OR

**BRANCH [2:0]‌ ‌**

This control signal will select the type of branching to be considered by the Branching and Jump detection unit. Instructions in RV32IM can be categorized into 8 categories depending on their branching.

* BEQ - For BEQ instruction
* BNE - For BNE instruction
* J - For J - Type instruction
* BLT - For BLT instruction
* BGE - For BGE instruction
* BLTU - For BLTU instruction
* BGEU - For BGEU instruction
* NO - For all other remaining instructions

BRANCH control signal is a 3 - bit control signal

**MEM\_WRITE‌**

In ‌RISC-V‌ ‌ISA,‌ ‌there‌ ‌are‌ **‌3‌ ‌types‌** ‌of‌ ‌store‌ ‌instructions‌ ‌depending‌ ‌on‌‌ the‌ ‌number‌ ‌of‌ ‌bits‌ stored.‌ ‌This‌ ‌control‌ ‌signal‌ ‌is‌ ‌sent‌ ‌to‌ ‌the‌ ‌data‌ ‌cache‌ ‌memory‌ ‌and‌ ‌the‌ ‌data‌‌ cache‌ ‌memory‌ ‌will‌ store‌ ‌according‌ ‌to‌ ‌the‌ ‌MEM\_WRITE‌ ‌signal.‌

‌Types‌ ‌of‌ ‌store‌ ‌instructions‌‌ are,

* SB‌ ‌-‌ ‌Store‌ ‌byte‌ ‌(least‌ ‌significant‌ ‌8‌ ‌bits‌ ‌from‌ ‌input‌ ‌data)‌ ‌
* SH‌ ‌-‌ ‌Store‌ ‌halfword‌ ‌(least‌ ‌significant‌ ‌16‌ ‌bits‌ ‌from‌ ‌input‌ ‌data)‌ ‌
* SW‌ ‌-‌ ‌Store‌ ‌word‌ ‌(32‌ ‌bits‌ ‌from‌ ‌input‌ ‌data)‌ ‌

**MEM\_READ‌ ‌**

In‌ ‌RISC-V‌ ‌ISA,‌ ‌there‌ ‌are‌ **‌5‌ ‌types‌** ‌of‌ ‌load‌ ‌instructions‌ ‌depending‌ ‌on‌‌ the‌ ‌number‌ ‌of‌ ‌bits‌ ‌loaded/ ‌This‌ ‌control‌ ‌signal‌ ‌is‌ ‌sent‌ ‌to‌ ‌the‌ ‌data‌ ‌cache‌ ‌memory‌ ‌and‌ ‌the‌ ‌data‌‌ cache‌ ‌memory‌ ‌will‌ ‌load ‌according‌ ‌to‌ ‌the‌ ‌MEM\_READ‌ ‌signal.‌

‌Types‌ ‌of‌ ‌load ‌instructions‌‌ are,

* LB‌ ‌-‌ ‌Load‌ ‌byte‌ ‌(8‌ ‌bits‌ ‌from‌ ‌given‌ ‌address)‌ ‌
* LH‌ ‌-‌ ‌Load‌ ‌halfword‌ ‌(16‌ ‌bits‌ ‌from‌ ‌given‌ ‌address)‌ ‌
* LW‌ ‌-‌ ‌Load‌ ‌word‌ ‌(32‌ ‌bits‌ ‌from‌ ‌given‌ ‌address)‌ ‌
* LBU‌ ‌-‌ ‌Load‌ ‌byte‌ ‌unsigned(8‌ ‌bits‌ ‌from‌ ‌given‌ ‌address)‌ ‌
* LHU‌ ‌-‌ ‌Load‌ ‌halfword‌ ‌unsigned(16‌ ‌bits‌ ‌from‌ ‌given‌ ‌address)‌ ‌

**WB\_SEL [1:0]‌ ‌**

There are 4 sources for the write back value to be written to the register file.

* ALU result
* Data from the data memory
* Immediate value - For LUI instruction
* PC + 4 value - For J - Type instruction

This control signal will select between these 4 sources. The WB\_SEL signal is a 2 - bit control signal

**WRITE\_EN**

This control signal will enable writing to the register file. When WRITE\_EN is set, the write back value is written to the register file and when WRITE\_EN is cleared, the write back value is not written to the register file.

### **4.3.5 Branch unit**

The Branch Detection Unit is a hardware unit in the RV32IM DataPath that identifies branch instructions and determines whether a branch should be taken or not. It plays a crucial role in controlling the flow of instructions and enabling conditional branching within the processor. The Branch Detection Unit continuously monitors the instruction stream for branch instructions, such as conditional branches and jumps. It examines register values and other relevant fields to detect the presence of branch instructions.

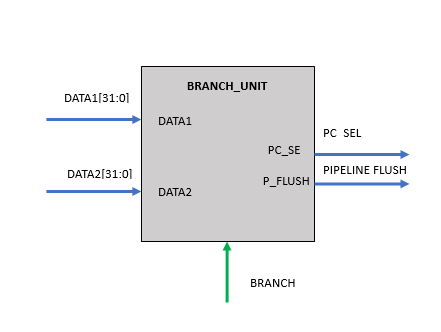
When a branch instruction is detected, the Branch Detection Unit evaluates the condition specified in the instruction. This condition is typically based on a comparison between two values, such as a comparison between registers or a register and an immediate value. The result of the comparison determines whether the branch should be taken or not

Figure 07:RV32IM Branch Unit

This unit is for detecting whether the branch or the jump has to be taken or not. Inputs to this unit are,

* DATA1[31:0]
* DATA2[31:0]
* BRANCH [2:0]

Control signal Output of this unit is,

* PC\_SEL
* P\_FLUSH

This unit will contain a comparator implemented using behavioral modeling and a combinational logic circuit to generate the PC\_SEL and P\_FLUSH control signal.

### **4.3.6 Immediate generation unit**

The Immediate Generation Unit is a hardware unit in the RV32IM DataPath responsible for generating immediate values used in instructions. Immediate values, also known as immediate operands, are constants embedded within instructions to provide additional data or parameters for operations.

The Immediate Generation Unit ensures the efficient and accurate generation of immediate values required by various instructions in the RV32IM architecture. These immediate values can represent signed or unsigned integers, offsets, shift amounts, or other constants needed for arithmetic, logical, or control flow operations

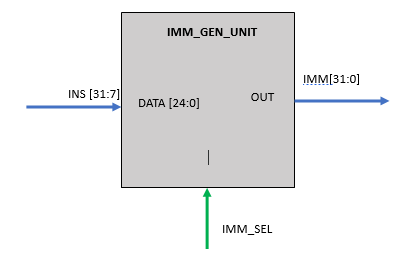


Figure 08:RV32IM Immediate Generation Unit

Instructions ‌‌with‌‌ immediate‌‌ values‌‌ can‌‌ be‌‌ categorized‌‌ into‌‌ 4‌‌ groups‌‌ and‌‌ I-Type‌‌ instructions‌‌ can‌‌ be‌‌ further‌ ‌categorized‌ ‌into‌ ‌3‌ ‌sub‌ ‌groups.‌

* U‌ ‌type‌ ‌Immediate‌ ‌
* J‌ ‌type‌ ‌Immediate‌‌ ‌
* S‌ ‌type‌ ‌Immediate‌ ‌
* B‌ ‌type‌ ‌Immediate‌ ‌
* I‌ ‌type‌ ‌Immediate‌ ‌with‌ ‌signed‌ ‌Extension‌ ‌(Extend‌ ‌with‌ ‌sign‌ ‌bit)‌ ‌
* I‌ ‌type‌ ‌immediate‌ ‌containing‌ ‌shift‌ ‌amount‌ ‌
* I‌ ‌type‌ ‌immediate‌ ‌with‌ ‌unsigned‌ ‌Extension‌ ‌(Extend‌ ‌with‌ ‌Zero)

Inputs‌ ‌to‌ ‌the‌ ‌immediate‌ ‌value‌ ‌generation‌ ‌unit‌ ‌are,‌ ‌

* INS [31:7]‌ ‌
* IMM\_SEL‌ ‌control‌ ‌signal [2:0]‌

Output‌ ‌of‌ ‌the‌ ‌immediate‌ ‌value‌ ‌generation‌ ‌unit‌ ‌is,‌ ‌

* IMM [31:0]

### **4.3.8 Instruction cache memory – INS\_CACHE**

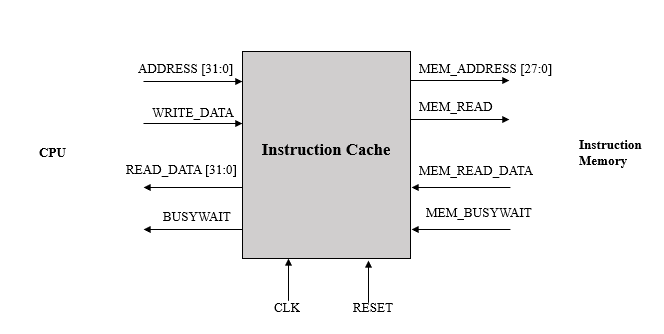


Figure 09:RV32IM Instruction cache

The Instruction Cache is a hardware component in the RV32IM processor that stores a subset of instructions fetched from memory. It provides a fast and efficient mechanism for accessing frequently used instructions, reducing the overall instruction fetch latency and improving the performance of the processor.

The Instruction Cache operates on the principle of locality, exploiting the tendency of programs to exhibit spatial and temporal locality. Spatial locality refers to the likelihood of accessing nearby instructions, while temporal locality suggests that recently accessed instructions are likely to be accessed again in the near future.

When an instruction fetch is required, the Instruction Cache is the first place the processor looks for the requested instruction. If the instruction is present in the cache (a cache hit), it is quickly accessed and provided to the instruction decoder. This avoids the latency associated with fetching the instruction from the main memory.

In case of a cache miss, where the requested instruction is not found in the cache, the processor has to fetch the instruction from the main memory and store it in the cache for future use. This retrieval from main memory takes longer compared to cache hits, introducing additional latency.

The size of the Instruction Cache, measured in terms of the number of instructions it can hold, and its organization, such as direct-mapped, set-associative, or fully associative, affect cache performance and hit/miss rates. Larger caches can store more instructions, reducing the chances of cache misses. Additionally, cache replacement policies, such as least recently used (LRU), determine which instructions are evicted from the cache when it reaches its capacity.

### **4.3.9 Data\_cache\_memory – data\_cache**

Figure 10: RV32IM Instruction cache

The Data Cache is a hardware component in the RV32IM processor that stores frequently accessed data from memory. It provides a fast and efficient mechanism for accessing data operands, reducing memory access latency and improving overall processor performance.

The Data Cache operates on the principle of locality, similar to the Instruction Cache, by exploiting spatial and temporal locality. Spatial locality refers to accessing nearby data elements, while temporal locality suggests that recently accessed data is likely to be accessed again in the near future.

When a data operand is required by an instruction, the Data Cache is the first place the processor looks for the data. If the data is present in the cache (a cache hit), it is quickly accessed and provided to the instruction that needs it. This avoids the latency associated with accessing the data from the main memory.

In case of a cache miss, where the requested data is not found in the cache, the processor has to fetch the data from the main memory and store it in the cache for future use. This retrieval from main memory takes longer compared to cache hits, introducing additional latency

### **4.3.10 Data memory – data\_mem**

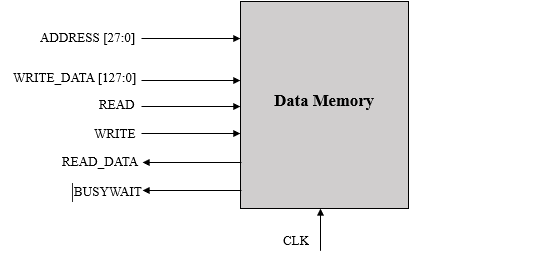


Figure 11:RV32IM Data memory

The Data Memory Module is a hardware component in the RV32IM processor that stores and manages data values accessed during program execution. It serves as an interface between the processor and the main memory, enabling the reading and writing of data.

The Data Memory Module consists of a memory array that stores data values. It provides storage for variables, arrays, and other data structures used by the program. The size of the data memory is determined by the processor's architecture and can vary depending on the specific implementation.

During program execution, the Data Memory Module handles load and store instructions. When a load instruction is encountered, it retrieves the requested data from the main memory and provides it to the processor for further processing. Similarly, when a store instruction is encountered, the Data Memory Module writes the provided data value into the appropriate memory location.

### **4.3.11 Instruction memory - ins\_mem**

The instruction memory module is shown in figure and the memory allocation is shown below.

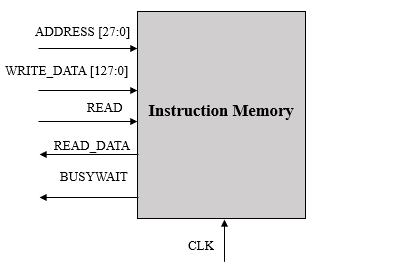


Figure 12:RV32IM Instruction memory

The Instruction Memory Module is a hardware component in the RV32IM processor that stores the instructions of a program. It provides the processor with a sequence of instructions to execute, enabling program control flow and instruction fetching.

The Instruction Memory Module consists of a memory array that stores the machine instructions. These instructions are typically fetched from non-volatile memory (e.g., flash memory) or loaded into memory from secondary storage devices. The size of the instruction memory is determined by the processor's architecture and can vary based on the specific implementation.

During program execution, the Instruction Memory Module fetches instructions based on the program counter (PC) value. The PC holds the address of the next instruction to be fetched. The Instruction Memory Module retrieves the instruction at the specified address and provides it to the instruction decoder for further processing.

### **4.3.12 Forwading unit 1**

Data hazards mean the data dependencies between instructions. We have implemented two forwarding units inside stage three and stage 4. There are several methods to handle data hazards. The easy and basic method is to insert bubbles into the pipeline if there are data dependencies. This is an easy approach, but this method decreases the efficiency of the pipeline. The second m

ethod is to use forwarding methods. That means taking the ALU results from stages 4 and 5 as operand 1 and operand 2 in the execution stage. The following diagram shows the results that should be forwarded in order to handle the dependency data hazards.

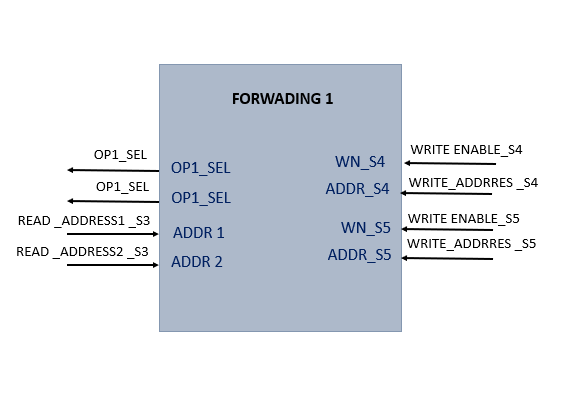
We have implemented a forwarding unit inside stage three for resolved R type data dependencies

Figure 13: Forwarding unit 1

Inputs for forwarding unit1 are

* READ \_ADDRESS1 \_S3
* READ \_ADDRESS2 \_S3
* WRITE ENABLE\_S4
* WRITE\_ADDRRES \_S4
* WRITE ENABLE\_S5
* WRITE\_ADDRRES \_S5

Outputs of forwarding unita1 are

* OP1\_SEL
* OP2\_SEL

This forwarding unit keeps the eye on operand 1 and operand 2 and checks whether there are new values for those lines in Stage 4 or Stage 5. If that is the case the forwarding unit sends the signal to the Hazard Handling Muxes in operand 1 and operand 2 data paths.

Verify whether there are register write operations in the instructions at stages 4 (WN\_S4) and 5 (EN\_S5), and confirm that the register read addresses in stage 3 (ADDR1, ADDR2) are equal to the write addresses in stages 4 (ADDR\_S4) or 5 (ADDR\_S5). If these conditions are met, select the forwarded outputs in the Hazard Handling Muxes.

### **4.3.13 Forwading unit 2**

We have implemented a forwarding unit inside stage 4 for resolve the load store hazards. In this forwarding unit check weather, a there is a load store hazards and if three is a hazard forward data from WB stage to the data memory for store.

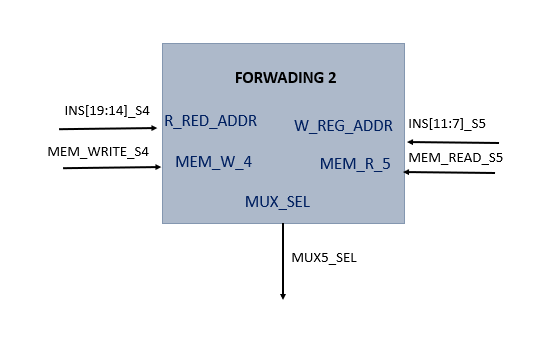


Figure 14: Forwarding unit 2

Inputs for forwarding unit1 are

* R\_REG\_ADDR
* W\_REG\_ADDR
* MEM\_W\_4
* MEM\_R\_5

Outputs of forwarding unita1 are

* MUX\_SEL

If there is a memory write operation in instruction at stage 4 and memory read operation in instruction at stage 5 then check register addresses in stage 5 if they are matched data is forwarded form stage 5 to stage 4. Write data is selected by mux 5.

**4.3.14 Hazaeds detection unit**

We implemented a Hazard detection unit in ID stage to detect Load and use data hazards and add bubble (**NOP)** to the pipeline.

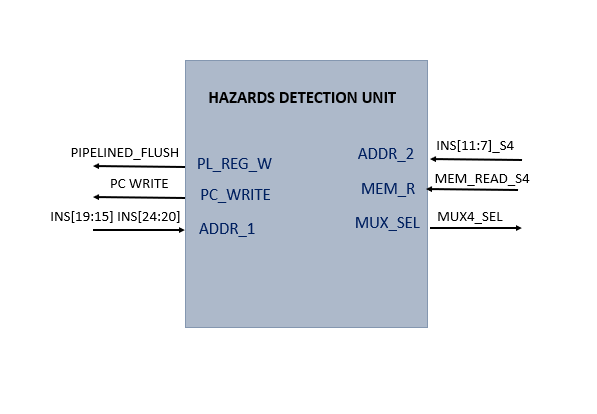


Figure 15:Hazards detection unit

Inputs for forwarding unit1 are

* ADDR1
* ADDR2
* MEM\_R

Outputs of forwarding unita1 are

* PL\_REG\_FLUSH
* PC\_WRITE
* MUX\_SEL

The hazard detection unit is situated at the ID stage. It examines whether a load operation exists in the instruction at the EX stage, and verifies if the write address of the instruction at the EX stage matches the source register in the instruction at the ID stage. If these conditions are met, a load-use hazard is detected. In such cases, the program counter (PC) is decremented by 4, the pipelined registers are flushed by adding a bubble (NOP) to the pipeline. The forwarding unit 1 controls the forwarding process.

### **4.3.15 PC Unit**

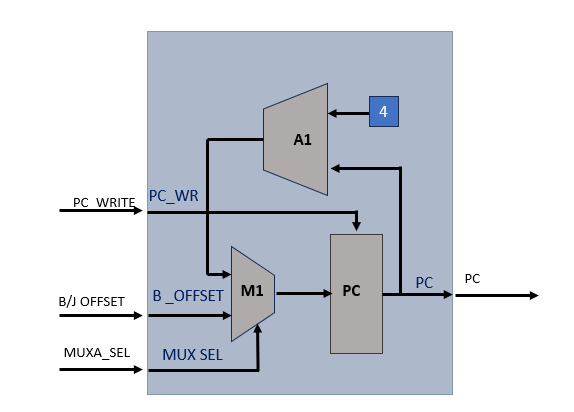


Figure 16:RV32IM PC register

Inputs for PC unit are

* PC\_WRITE
* B\_OFFSET
* MUX\_SEL

Outputs of PC unit are

* PC

Program counter register stores the address of the instruction. The value stored in this register is used when fetching the instruction from the instruction memory. Writing to the PC register is synchronous to the positive clock edge and the reading from the PC register is asynchronous. When the reset signal is set, the value in the PC register will be set to -4 and the program will restart from the next clock cycle.

When there is a branch or jump PC value update with branch jump offset instead of incrementing 4 this is selected by the mux M1.

### **4.3.16 Pipeline registers**

Pipelined registers are inserted between pipeline stages to hold the intermediate results of each instruction. These registers act as temporary storage for data and control signals as they flow through the pipeline. They enable the smooth transfer of information between pipeline stages and ensure that the correct data is available when needed.

‌this‌ ‌CPU‌ ‌design‌ ‌there‌ ‌are‌ ‌5‌ ‌stages,‌ ‌

* Instruction‌ ‌Fetch‌ ‌stage‌ ‌
* Instruction‌ ‌Decode‌ ‌stage‌ ‌
* Execution‌ ‌stage‌ ‌
* Memory‌ ‌Access‌ ‌stage‌ ‌
* Writeback‌ ‌stage‌

In‌ ‌between‌ ‌each‌ ‌stage‌ ‌there‌ ‌is‌ ‌a‌ ‌pipeline‌ ‌register.‌‌ ‌

* IF/ID‌ ‌pipeline‌ ‌register‌ ‌
* ID/EX‌ ‌pipeline‌ ‌register‌ ‌
* EX/MEM‌ ‌pipeline‌ ‌register‌ ‌
* MEM/WB‌ ‌pipeline‌ ‌register‌

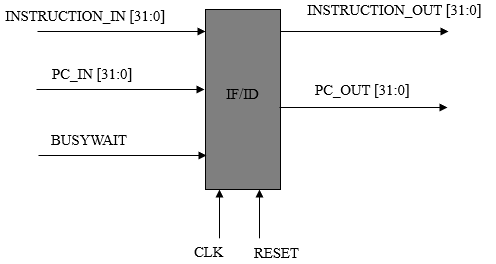


Figure 17:RV32IM IF/ID Pipeline register

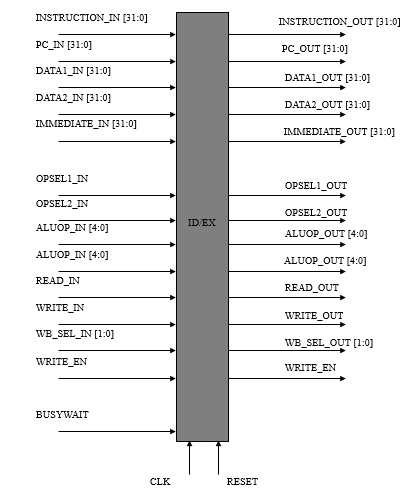


Figure 18:RV32IM ID/EX Pipeline register

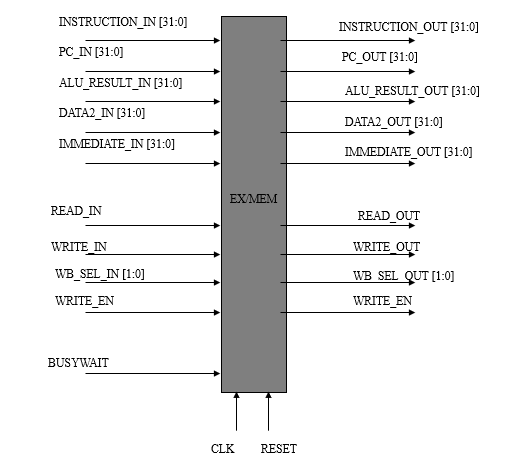


Figure 19:RV32IM EX/MEM Pipeline register

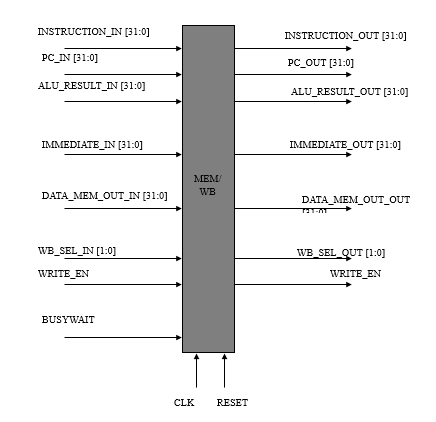


Figure 20:RV32IM EX/MEM Pipeline register

Data‌ ‌and‌ ‌control‌ ‌signals‌ ‌are‌ ‌written‌ ‌to‌ ‌the‌ ‌pipeline‌ ‌registers‌ ‌at‌ ‌the‌ ‌positive‌ ‌edge‌ ‌of‌ ‌the‌ ‌clock‌ ‌cycle‌‌ and‌ ‌when‌ ‌the‌ ‌reset‌ ‌signal‌ ‌is‌ ‌set,‌ ‌the‌ ‌pipeline‌ ‌registers‌ ‌will‌ ‌get‌ ‌reset.

### **4.3.17 Pipeined flushing unit**

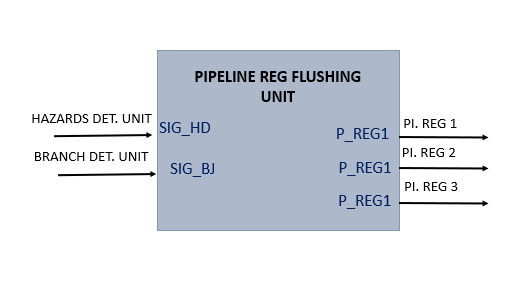
When there is a stall hazards, we need to flush the pipelined registers when there is a branch or jump that branch or jump is taken there should be flushed the pipelined registers as well as there is a Load and uses data hazards flushing pipelined registers is necessary.

Figure 21:RV32IM Pipeline Flushing unit

Inputs for Pipeline flushing unit are

* SIG\_HD
* SIG\_BJ

Outputs of Pipeline flushing unit are

* P\_REG1
* P\_REG2
* P\_REG3

### **4.3.18 Adders**

There are two separate adders in our data path

1. A1-PC ADDER – Incrementing PC value by 4 for next instruction in PC unit
2. A2- PC ADDER2- Incrementing PC value by 4 storing PC value in register file

### **4.3.19 MULTIPLEXURES**

Three 8 multiplexers are used in the design. Depending on the select signal, the multiplexer will output the corresponding value.

1. M1- Selection between PC+4 and branch/jump target
2. M2- Selection between Control signals and reset value 0
3. M3- Selection between DATA1 and forwarding from the stage 4 and 5 for stage for operand 1
4. M4- Selection between DATA2 and forwarding from the stage 4 and 5 for stage for operand 1
5. M5 - Selection between M3\_OUT and PC value for operand 1
6. M6 - Selection between M4\_OUT and Immediate value for operand 2
7. M7 - Selection for data memory write data
8. M8- Selection for write back data

# NoC (Network on chip) implementation

# 5.SNN Software implementation

**6.Testing**

**References**

**Appendix**