UNIVERSITY OF SOUTHAMPTON

Faculty of Physical Sciences and Engineering Electronics and Computer Science

Coarse-Grained Multithreaded RISC-V Processor

By

Mohammad Zeyad Abu Alhalawe

05/09/2019

Supervisor: Mr. Iain McNally

2nd examiner: Prof. CH Kees de Groot

A dissertation submitted in partial fulfilment of the degree of

MSc. Microelectronics Systems Design

Abstract

The increase in performance of the processor is one of the main tasks in modern architectures, multithreaded architecture is one of the concepts introduced in order to maximize the performance of a single-core processor. In this project, the design of coarse-grained multithreaded architecture took place along with cache and main memory design. The system was designed in order to evaluate the performance gain of the processor. According to the results obtained, it is proved that two threaded processor had at least 3.9% improvement over the single-threaded processor, where four threaded processor had at least 8% improvement over the single threaded processor.

Acknowledgement

I would like to thank my parents for their support during my study, I would also like to sincerely thank Mr. McNally for his help and patience during the project time.

Table of Contents

List of Figures

Chapter 1

Introduction

Processor performance is the major concern for the designers. Many approaches were taken to maximize the performance of a single and multi-core system. This project focuses on applying multithreaded architecture and pipelined processors techniques on a single-core processor along with cache memory design. The system was developed using System Verilog hardware description language and the final design was implemented on Altera DE2-115 development board where all functionalities were verified using the board's output. Figure 1.1 shows the 5-stage pipelined processor along with the memory hierarchy designed during the project

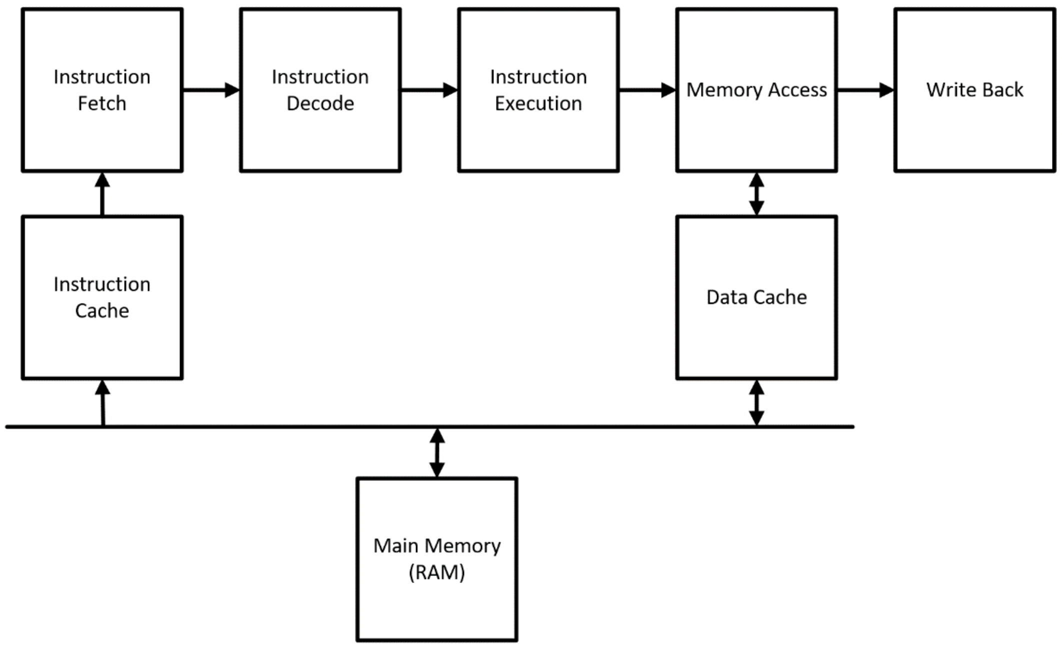


Figure 1.1: 5-stage pipeline design along with cache memories and main memory

The designed processor executes RISC-V instructions. RISC-V is an open-source instruction set which is new and not many projects were implemented using it. The processor executes instructions from RV32I and RV32M RISC-V extensions along with one CSR instruction.

The project aims to hide long memory latencies caused by accessing the main memory. This latency was hidden by exploiting multithreaded architecture which switches between different programs that need to be executed during the main memory access time.

Finally, the final design includes a 5-stage pipelined processor with a coarse-grained multithreaded architecture. Memory hierarchy in this project consists of level 1data and instruction cache memories along with main memory where all data and instructions are stored.

Chapter 2

Research Background

2.1 RISC-V Processor

RISC-V Processor is a modern architecture; it was released in 2014 by the University of California where it was originally designed. The processor is based on SPARC and MIPS processors architecture by combining these two relatively old architectures and producing a modern one. RISC-V foundation offers an open-source instruction set architecture (ISA) that allows designers to use their instruction set for any design and implementation [1]. It offers a wide range of instructions that varies from 32, 64 and 128 bits based on the designer's choice.

RISC-V instruction set is divided into two main instructions types, privileged and non-privileged instructions, privilege refers to the allowance of an instruction to access control and status registers (CSRs). Having privileged instructions allowed the processor to have different modes running in the processor. The highest privilege goes to machine mode, which is the processor where it can read and write to any CSRs, the second privileged mode is the supervisor mode which can access some of the CSRs, and the lowest privilege goes to the user level program which is allowed to access very few numbers of the CSRs [2].

For non-privileged instructions, there are different extensions for instructions based on their functionality. The extensions are I (base integer instructions), M (integer multiplication and division instructions), A (atomic instructions), F, D and Q (different precisions floating-point extensions). Finally, the instructions are divided into different types based on the fields of the instruction, Figure 2.1.1 shows the types along with the fields of each type.

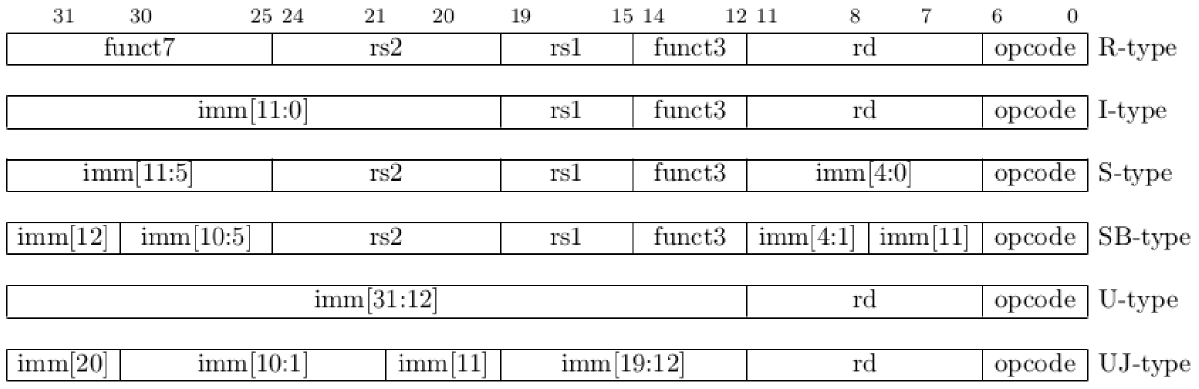

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | SB-type
imm[31:12] | rd | opcode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

Figure 2.1.1: Types of instructions along with the fields of each type (source: The RISC-V Instruction Set Manual [1]).

2.2 Processor Pipeline

Processor pipeline is a technique used to speed up the processor throughput. Pipelining the processor means dividing the processor into stages and storing the output of each stage in a register before proceeding to the following stage, which will allow multiple instructions being executed at the same time in different stages. By this division, the frequency will depend on the stage that takes the most time, wherein a non-pipelined processor the frequency depends on the total time consumed in each processor unit with respect to the slowest instruction.

2.2.1 Classic 5 Stages Pipeline

The design of the 5 stages pipeline depends on dividing the pipeline into the following stages:

1. Instruction Fetch (IF): the instruction memory is accessed during this stage to read the address pointed by the program counter (PC) and load the instruction [3].

2. Instruction Decode (ID): the instruction is decoded, and the register file is accessed to read the values needed to execute the instruction [3].

3. Execute (EX): in the execute stage the ALU executes the instruction with the specified values [3].

4. Memory (MEM): in this stage, the data memory is accessed by load and store instruction [3].

5. Writeback (WB): this stage is the last stage of the pipeline; the data is written back to the register file [3].

Figure 2.2.1 illustrates the 5-stage pipeline and the concurrent execution of instructions among different stages.

Instructions	Cycle1	Cycle2	Cycle3	Cycle4	Cycle 5	Cycle6	Cycle7	Cycle8
addi x6,x7,5	IF	ID	EX	Mem	WB			
or x5,x2,x1		IF	ID	EX	Mem	WB		
Add x4,x3,x2			IF	ID	EX	Mem	WB	
mul x8,x15,x9				IF	ID	EX	Mem	WB

Figure 2.2.1: illustration of a 5-stage pipeline execution

Figure 2.2.2 shows a 5-stage pipelined RISC-V processor architecture as proposed by D. Patterson and J. Hennessey

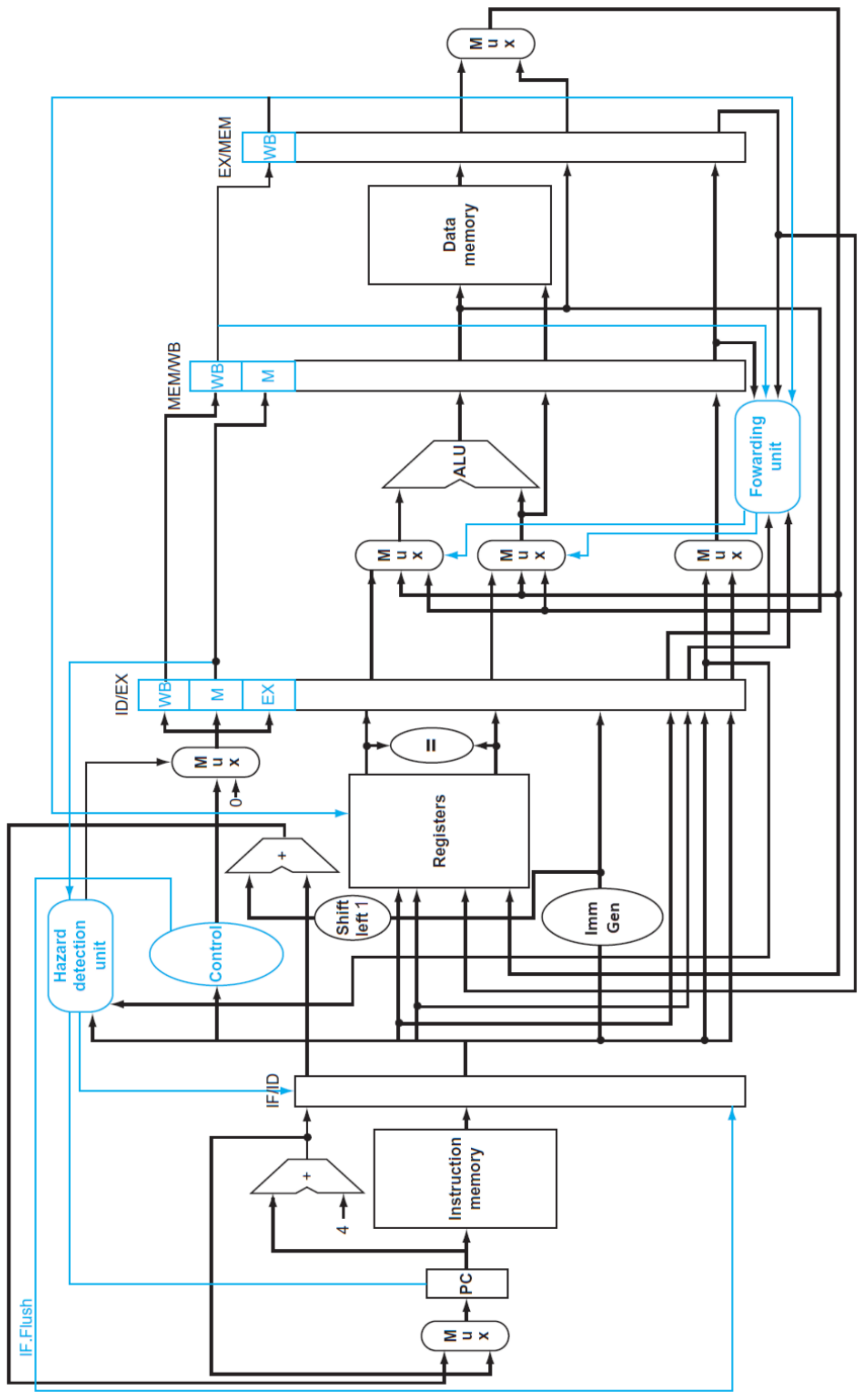


Figure 2.2.2: 5-stage RISC-V pipelined processor design (source: Hennessy, J. L., & Patterson, D. A. [3]).

2.2.2 Hazards

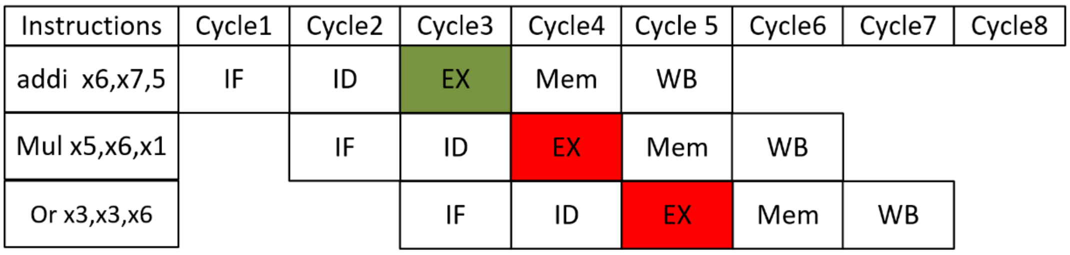
With the pipelined processor, there will be cases where the instruction cannot be executed because the hardware is unable to do so, these cases are called hazards. In order to avoid the wrong execution of an instruction, these hazards must be resolved in the design. The three types of hazards are structural, data and control hazards.

2.2.2.1 Structural Hazards:

structural hazards occur when two or more instructions are trying to use the same resource at the same time and this resource cannot execute in an overlapping style which will result in failure in the processor. Modern architectures avoid such types of hazards by considering these cases during the design

2.2.2.2 Data Hazards:

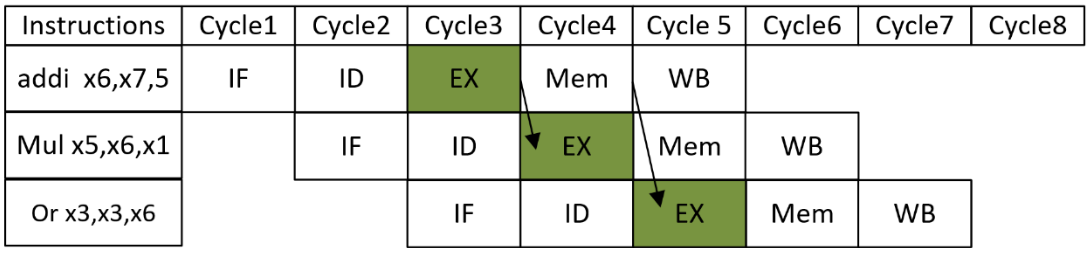
Data hazards occur when an instruction execution depends on a result of an instruction currently being executed in the pipeline and has not written to the register file yet because it has not reached WB stage. This will cause the new instruction to read an outdated value from the register file in ID stage. As a result of that, the instruction will be wrongly executed. This type of data hazards is called read after write (RAW) Figure 2.2.3 illustrates how RAW occur, where the green slot means the new value of x6 and the red slots mean the execution is done with an outdated value of x6.



Instructions	Cycle1	Cycle2	Cycle3	Cycle4	Cycle 5	Cycle6	Cycle7	Cycle8
addi x6,x7,5	IF	ID	EX	Mem	WB			
Mul x5,x6,x1		IF	ID	EX	Mem	WB		
Or x3,x3,x6			IF	ID	EX	Mem	WB	

Figure 2.2.3: RAW hazards in a pipelined processor.

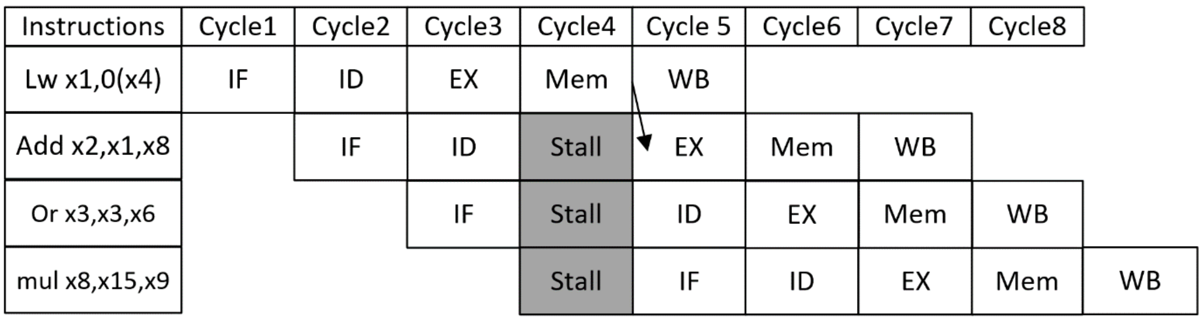
In order to overcome RAW hazards, forwarding should be presented by passing the value of x6 for execution when needed from Mem stage and WB stage, Figure 2.2.4 shows how the forwarding is done, the green slots in the figure show that the instruction has been executed correctly.



Instructions	Cycle1	Cycle2	Cycle3	Cycle4	Cycle 5	Cycle6	Cycle7	Cycle8
addi x6,x7,5	IF	ID	EX	Mem	WB			
Mul x5,x6,x1		IF	ID	EX	Mem	WB		
Or x3,x3,x6			IF	ID	EX	Mem	WB	

Figure 2.2.4: forwarding from memory and writeback stages to avoid RAW

Another type of data hazard would be load-use hazards. This hazard is caused when a load instruction takes place to load a value from the data memory, and the next instruction is using the loaded value. Load-use hazard can be solved by stalling the pipeline for one cycle in order to forward the loaded value from WB stage. The stall signal keeps the instruction in the same stage for the next cycle by not allowing the pipeline registers to be updated. The stall would delay the processor for one cycle, this will lead to a delay in the execution of the instruction for one cycle allowing the load instruction to obtain the data from the memory. Figure 2.2.5 shows how load-use hazards are solved in the pipeline by showing the stall cycle and the forwarded data.

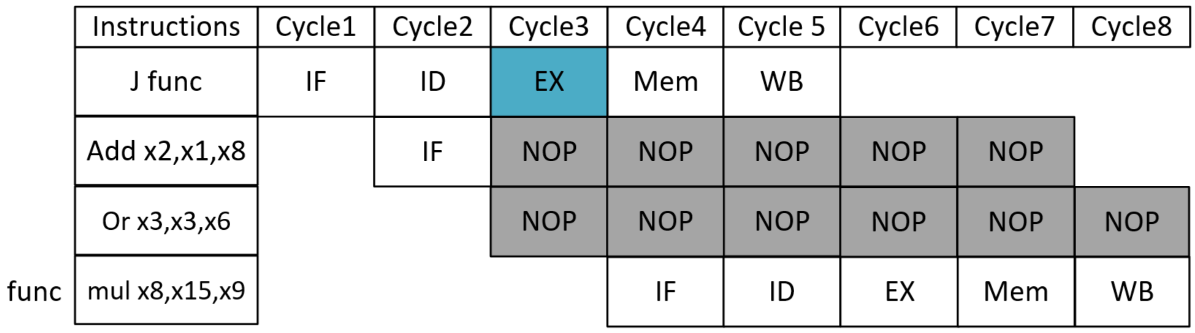


Instructions	Cycle1	Cycle2	Cycle3	Cycle4	Cycle 5	Cycle6	Cycle7	Cycle8	
Lw x1,0(x4)	IF	ID	EX	Mem	WB				
Add x2,x1,x8		IF	ID	Stall	EX	Mem	WB		
Or x3,x3,x6			IF	Stall	ID	EX	Mem	WB	
mul x8,x15,x9				Stall	IF	ID	EX	Mem	WB

Figure 2.2.5: load-use hazards and its solution

2.2.2.3 Control Hazards

The final type of hazards is the control hazards; control hazards arise from having a control transfer instruction. Control transfer instructions are jump and branch-related instructions, where these instructions change the program counter which will cause the next instruction fetched in the pipeline to be flushed because it is not supposed to be executed. The flush is done by inserting a NOP instruction which means the processor is not executing any instruction from the program. Figure 2.2.6 shows a control transfer instruction along with the flush and NOP instruction being inserted in the pipeline, the blue slot indicates the stage where the jump instruction was executed.



	Instructions	Cycle1	Cycle2	Cycle3	Cycle4	Cycle 5	Cycle6	Cycle7	Cycle8
	J func	IF	ID	EX	Mem	WB			
	Add x2,x1,x8		IF	NOP	NOP	NOP	NOP	NOP	
	Or x3,x3,x6			NOP	NOP	NOP	NOP	NOP	NOP
func	mul x8,x15,x9				IF	ID	EX	Mem	WB

Figure 2.2.6: Control Hazards with NOP instruction

2.3 Memory Hierarchy

The memory in processors can be divided into different kinds, the main trades between these kinds of memory are the capacity and memory access time. Both fast memories and high capacity memories are demanded for high performance. Therefore, a combination of both trades is designed which results in a memory hierarchy in all modern systems. The two main types of memory designed for the processor are static random-access memory (SRAM) which is a fast memory with a high cost and dynamic random-access memory (DRAM) which is slow memory with low cost [4].

In terms of processor, the processor reads data for execution from the register files in RISC architecture, and since the register file size in RISC architecture is usually 32 registers for 32-bit instructions. Bigger memory should exist in the system in order to store a large amount of data, this memory is usually the DRAM. SRAM memories were designed to overcome the slow access time issue, by designing a small memory inside the processor to store a small amount of data that exist in the RAM. Figure 2.3.1 illustrates the access time and the capacity of each memory type in the hierarchy, where the top is the smallest with the least access time and the bottom is the slowest with most access time [4].

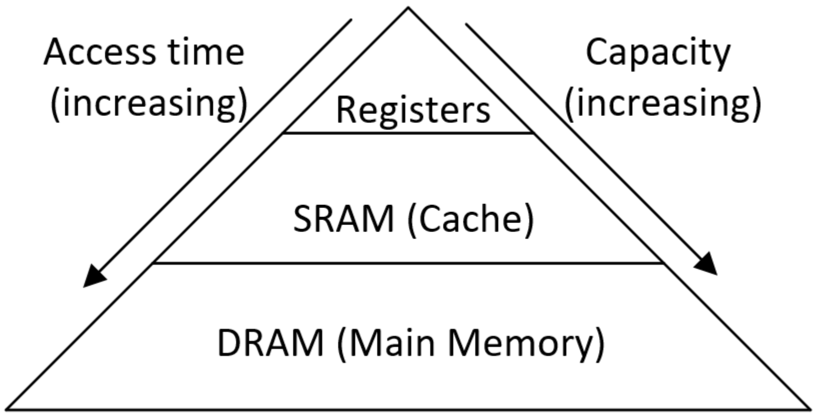


Figure 2.3.1: Memory hierarchy access time and capacity

2.3.1 Cache Memory Design

Since the cache memory is much smaller than the main memory, different approaches to mapping the cache with the main memory exist in order to maximize the possibility of finding the designated data in the cache (hit rate).

There are two types of locality that exploit the cache and the mapping is based on them which are:

- Temporal Locality: in this type of locality the assumption is if the data is accessed once it is likely to be accessed again [5].
- Spatial Locality: in this type of locality the assumption is if the data is accessed it is likely that the addresses around that address will be accessed as well [5].

The mapping techniques of the cache are the following:

1. Direct mapping technique: in this technique, several RAM locations are mapped into one cache location based on the address of the data in the RAM. Since more than one data can be mapped into the same location, the chance of the data being replaced by another one is high which cause to a high cache miss rate, but the hit time of this technique is low. Direct mapping exploits temporal locality and does not exploit spatial locality [5].

2. Associative Cache: in this technique of mapping, an entire block of the RAM memory can be placed in any location inside the cache, when the cache is full, the block that was not used recently is discarded and replaced with another one, the access of such kind of mapping is high compared to direct and set-associative mapping, but the miss rate is significantly reduced. Since an entire block from the RAM is placed in the cache, both spatial and temporal locality are exploited with this technique [5].

3. N-way set-associative mapping: this technique is similar to the associative mapping, but in order to reduce the access, the block of the RAM is not entirely placed in the cache, where N represents the number of data placed from one block inside the cache. Like associative mapping, temporal and spatial locality are exploited. Noting that since the block of the RAM is not entirely placed in the cache, the spatial locality is less effective in this technique [5].

2.4 Multithreading Techniques

Multithreaded architecture is a memory tolerance technique; with this type of architectures, more than one program is running inside the processor. The aim of the architecture is to hide the waste latencies inside the processor. The waste is the number of cycles that the processor cannot perform any operation due to hazards, cache misses and memory access time.

Waste latencies can be classified into two categories. The categories are horizontal and vertical waste. Horizontal waste occurs when the pipeline stages cannot execute any instruction for a single execution time as shown in Figure 2.4.1. while vertical waste means that the pipeline is unable to execute any instruction for multiple execution time as shown in Figure 2.4.1.

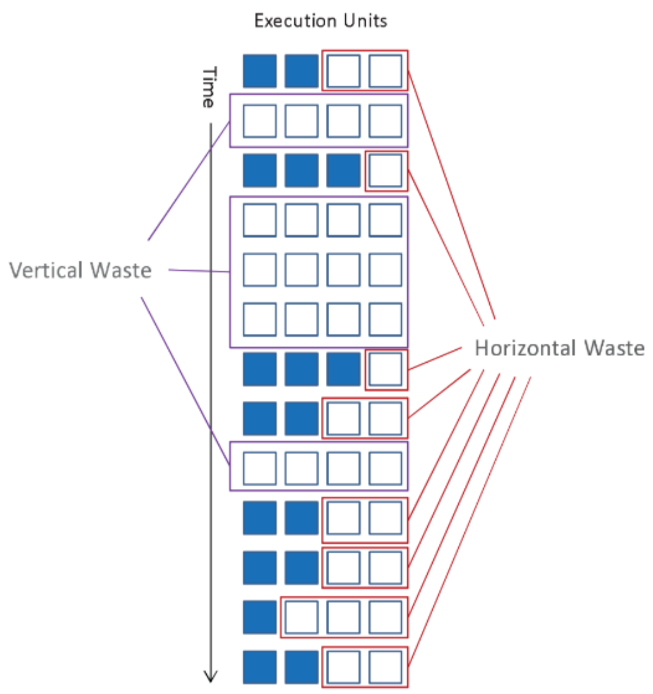


Figure 2.4.1: difference between horizontal and vertical waste (source:] Nemirovsky, M., & Tullsen, D. M. [6]).

Multithreading technique exploits the wastes mentioned before to execute another program and make use of the processor resources as much as possible by exploiting instruction-level parallelism (ILP) and thread-level parallelism (TLP). There are three different multithreading techniques that can be summarized as the following:

2.4.1 Fine-Grained Multithreading (FGMT)

FGMT is used to hide short waste latencies inside the processor, the waste is hidden by switching to a different thread after a static number of clock cycles which is usually one. With this kind of switch, the delays caused by hazards and load transfer instructions are hidden because each pipeline stage is executing a different program. This type of multithreading is useful the most if the number of the threads running in the processor is higher than two; because the maximum number of delays caused by one thread is two as explained in section 2.2.2.3.

The advantages of adding such technique are the cost of implementation and its ability to hide horizontal wastes explained earlier. The disadvantages of this technique are that it cannot hide long waste latencies caused by cache misses and main memory access which lead to a vertical waste, Figure 2.4.2 illustrates how FGMT works, the figure shows a static switch at each cycle.

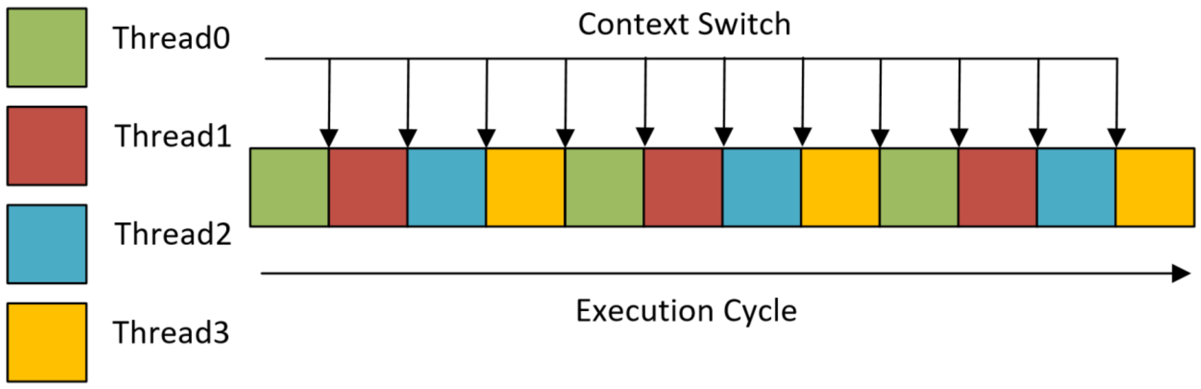


Figure 2.4.2: FGMT with static switch

2.4.2 Coarse-Grained Multithreading (CGMT)

CGMT main aim is to hide long waste latencies in the processor. These long delays can be caused by instruction or data cache miss and page fault. CGMT context is dynamically switched when one of the caches misses. Since accessing the main memory takes a long time, this time can be exploited by executing another thread. CGMT cannot hide short waste latencies, but different techniques can be applied to minimize the loss.

The threads in CGMT can be classified in three modes during the execution, the modes are running, waiting and ready. When a thread switch occurs, the processor will select a ready thread for execution. The ideal number of threads for CGMT is discussed in detail in Chapter 4.

The advantages of CGMT are the low cost of implementation and its ability to hide vertical wastes caused by accessing the main memory. The disadvantages are the loss caused by the context switch and being unable to hide horizontal wastes. Figure 2.4.3 shows how the CGMT switch occurs and how the switch is dynamic.

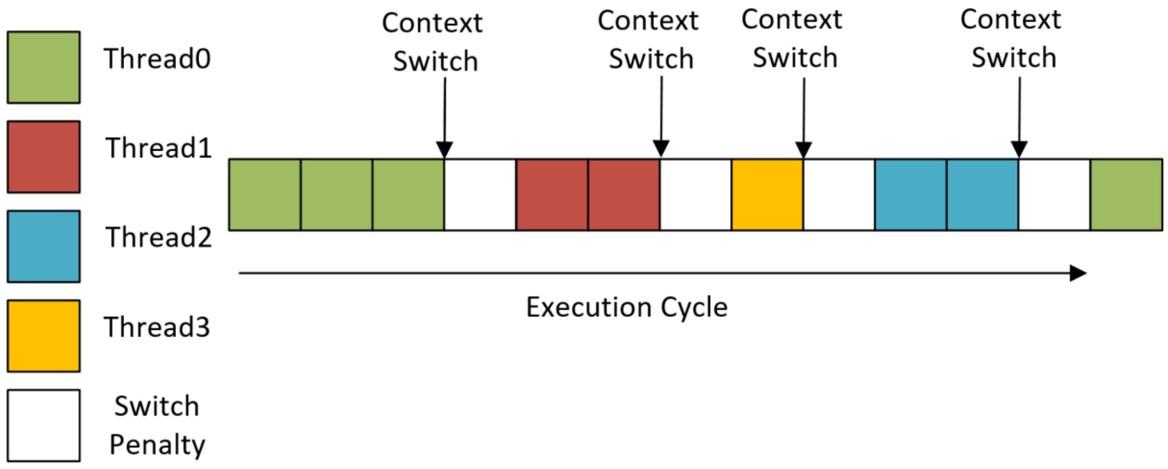


Figure 2.4.3: CGMT with dynamic switch

2.4.3 Simultaneous Multithreading (SMT)

SMT main aim is to hide both vertical and horizontal wastes. this type of multithreading can only be applied on a superscalar processor. Superscalar processor has N-way fetching units, where N is the number of threads running inside the same core. With this feature, multiple instructions can be fetched at the same time from different threads, each one of them executed concurrently with its own processing units since superscalar processor duplicates the processing units.

The advantages of SMT are the ability to hide both vertical and horizontal wastes, where the disadvantages are the high cost of implementation because it duplicates the processing units as mentioned earlier. Figure 2.4.4 shows a 4-way superscalar processor along with the concurrent execution of threads.

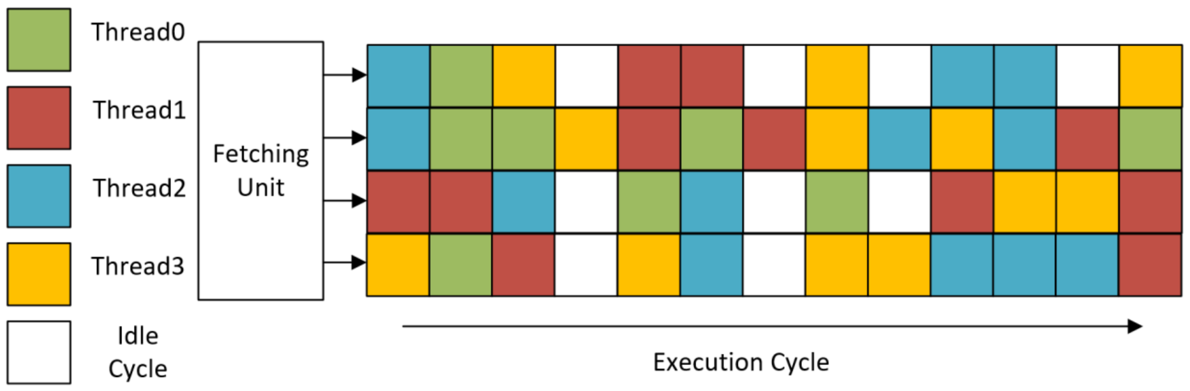


Figure 2.4.4: SMT multithreading with 4-way superscalar processor.

Chapter 3

Design of Coarse-Grained Multithreading Processor

Overview

The aim of the project was to design a 5-stage pipelined multithreaded RISC-V Processor which runs RISC-V instruction set. The design is based on the classical 5 stage pipelined processor with some modifications were made in order to meet the design requirements. These changes were made in order to get a maximum of one cycle being wasted at each switch made between the threads. Memory hierarchy presented in this design consists of level 1 instruction and data cache memory and main memory where all the data and instructions are located. This design with this level of hierarchy allowed the design to have long memory latency which is intended to be hidden by the processor as a result of adding coarse-grained multithreading technique.

The final processor supports 32-bit instructions, a variety of instructions from the RV32I and RV32M instructions were implemented, the instructions that were not implemented during the design are FENCE, ECALL, EBREAK and bit manipulation instructions. Additionally, privileged instruction had to be added to the processor, which is CSRRS instruction. This instruction has a higher privilege than the other instructions because it can read control and status registers while the other instructions could only read registers within the general-purpose registers file. The intended register to be read by the program was mhartID control register which holds the hardware thread ID of each thread.

Finally, as mentioned above the processor modification were successfully implemented and the coarse-grained multithreaded architecture was achieved, Figure 3.1 below illustrates the design of the 5-stage multithreaded pipelined processor.

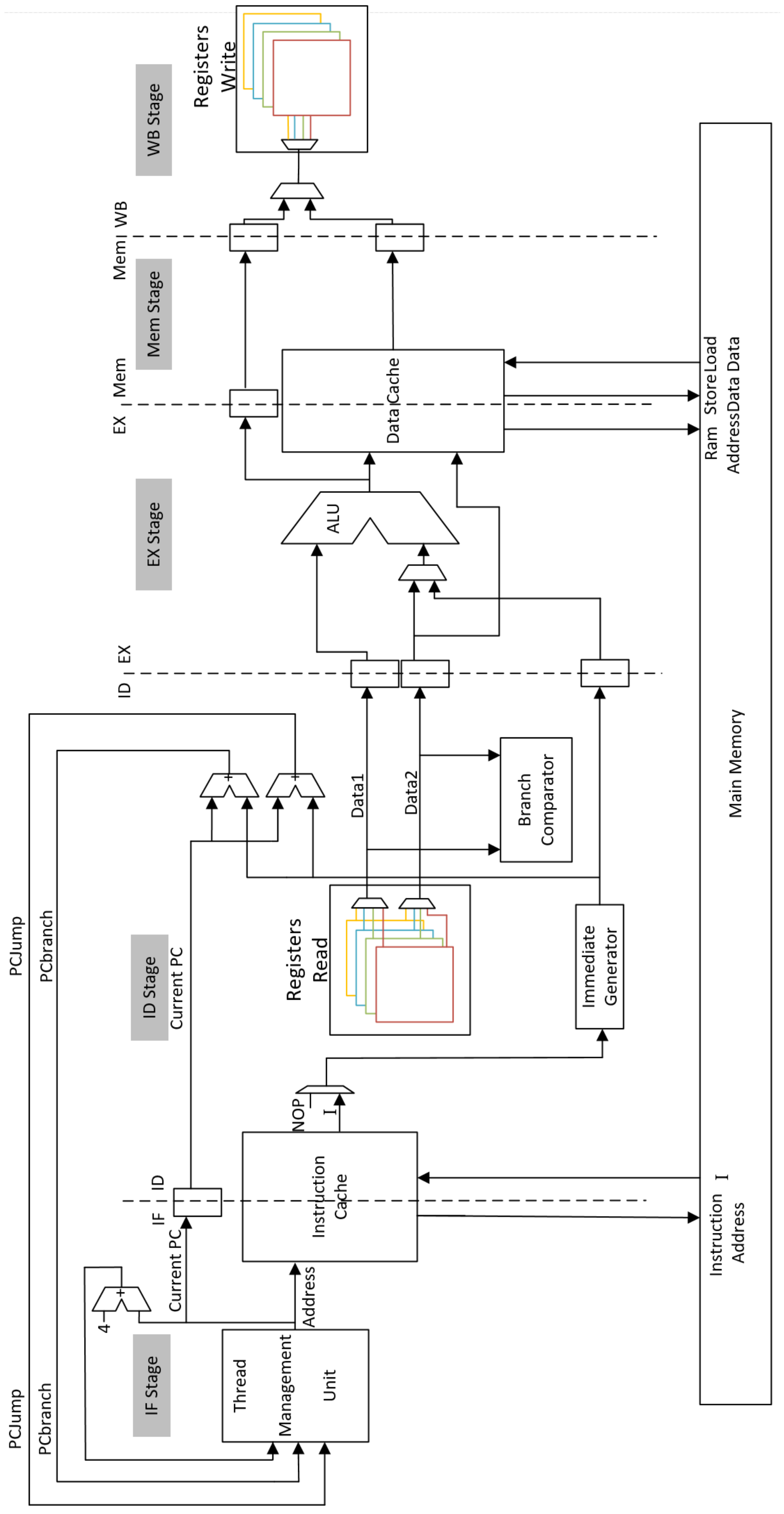


Figure 3.1: the design of the 5-stage multithreaded pipelined processor.

3.1 Instruction Fetch (IF)

The instruction fetch is the first stage of the pipelined processor. The instruction cache is accessed during this stage to look for the instruction. The address of the instruction is provided by the thread management unit. If the intended instruction exists within the instruction cache, the instruction will proceed to the next stage, if there is a cache miss, the instruction cache will inform the thread management unit that the instruction does not exist, and the thread cannot proceed to the next stage. Such a situation will cause a switch to a different thread and a NOP instruction inserted in the pipeline to avoid wrong execution of the threads. The instruction cache address is set during the IF stage where the instruction will be available in the decode stage. Figure 3.1.1 illustrates the IF stage along with the output of the instruction cache in the decode stage.

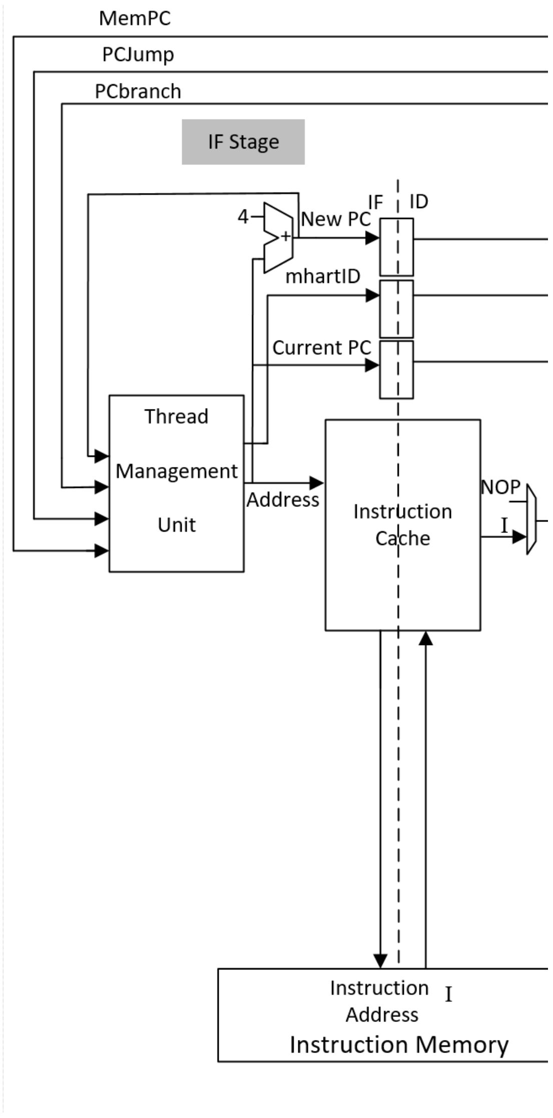


Figure 3.1.1: IF stage with the associated units' connections

3.2 Instruction Decode (ID)

The decode stage is the second stage of the pipeline. The instruction is decoded to its relevant control signals and the general-purpose registers are accessed based on the thread id related to this instruction. The immediate is generated during this stage for instructions with immediate values. For faster execution and for decreasing the loss caused by jump and branch, the jump and branch-related instructions are executed within this stage. Since the branch instructions in the RISC-V processor are executed by comparing the contents of registers, forwarding the ALU results from EX stage and Mem stage is necessary for correct execution, noting that no forwarding is required from writeback stage because the general-purpose register file is designed to assign the writeback data to the register being accessed by the instruction in case the thread ID and the address match the one being decoded. Figure 3.2.1 below shows the path of the decode stage along with the inputs that have been forwarded from different stages.

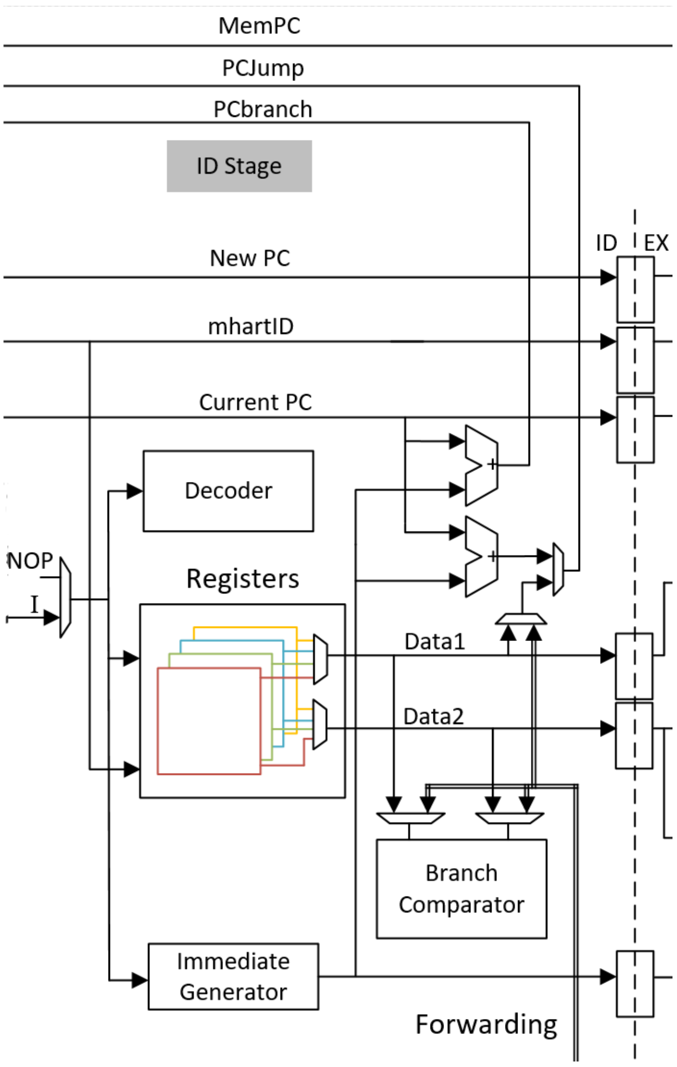


Figure 3.2.1: decode stage with the units related to it

3.3 Execute Stages (EX)

In the execute stage, the ALU performs the specified operation for the instruction, the operations supported by the ALU is determined by the RISC-V instructions where all functionalities for all instructions were added, forwarding is necessary for the ALU to calculate the correct value for the instruction, therefore forwarding the previous ALU outputs from memory stage and writeback stage is required.

3.4 Memory Stage (Mem)

The memory stage is where the data cache is accessed. The address of the cache and the data to be stored are set during EX stage. The ALU result in Mem stage is forwarded back to the EX and ID stage if needed. When the cache memory is accessed, if there is a load instruction and a cache hit, the instruction will proceed to the writeback stage, if a cache miss occurs the address of the instruction in the memory stage is forwarded back to thread management unit. Store operation is described in detail in section 3.5.3.

3.5 Writeback Stage (WB)

WB stage is the stage where the registers file is written back to by the instruction. The ALU result in the writeback stage is forwarded back to EX stage if needed.

Figure 3.2 shows the final design of the processor. Noting that for simplicity of the figure memory controllers are not included. Forwarding between pipeline stages is shown in the figure as two closely parallelized lines connected to one multiplexer input to simplify the figure. Forwarding to EX stage is from memory and writeback stages, where forwarding to branch comparator unit in ID stage is from EX and Mem stages

Figure 3.2 also shows register files of four threads because it is the maximum amount of threads designed for this project as will be discussed later in chapter 4.

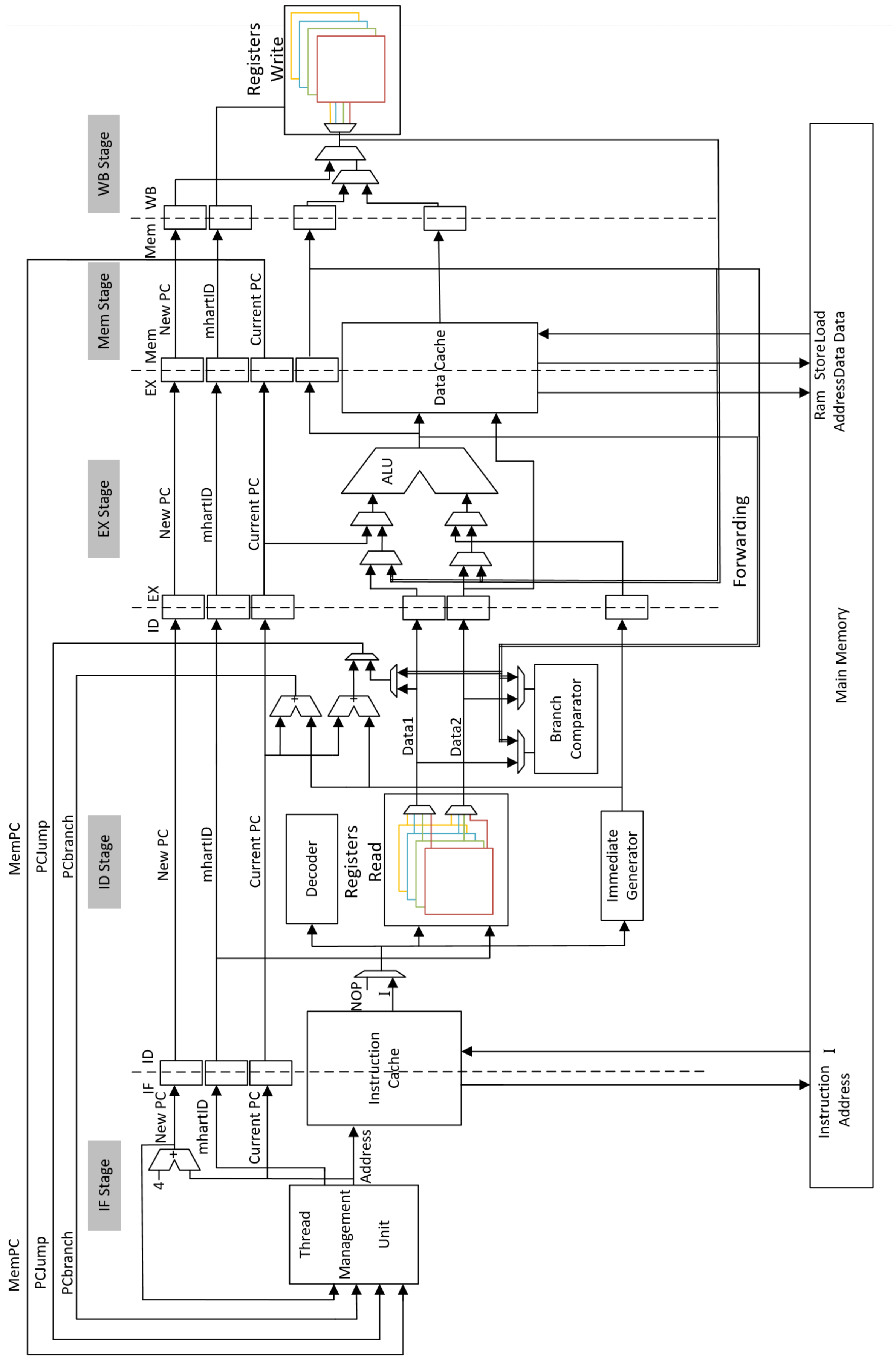


Figure 3.2: The final design of the coarse-grained multithreaded processor.

3.4 Memory Hierarchy Design

3.4.1 Cache Memory Design

The cache memory designed in the project supports direct mapping. The access time of the cache is one cycle as the cache is synchronous to the system clock, after the address is set, the cache memory will divide the address into cache index and address tag parts, the width of tag and index mainly depends on the cache size as a result of using direct mapping, valid bit is the bit to tell the cache control whether this data is valid to be read or not. The size of the cache for both data and instruction are 512 bytes (128 words) which means that the cache index part of the address is 7 bits and tag address is 23 bits, which leaves 2 bits for the byte offset. The dataflow of the cache and the address of data and instruction caches are illustrated in Figure 3.4.1

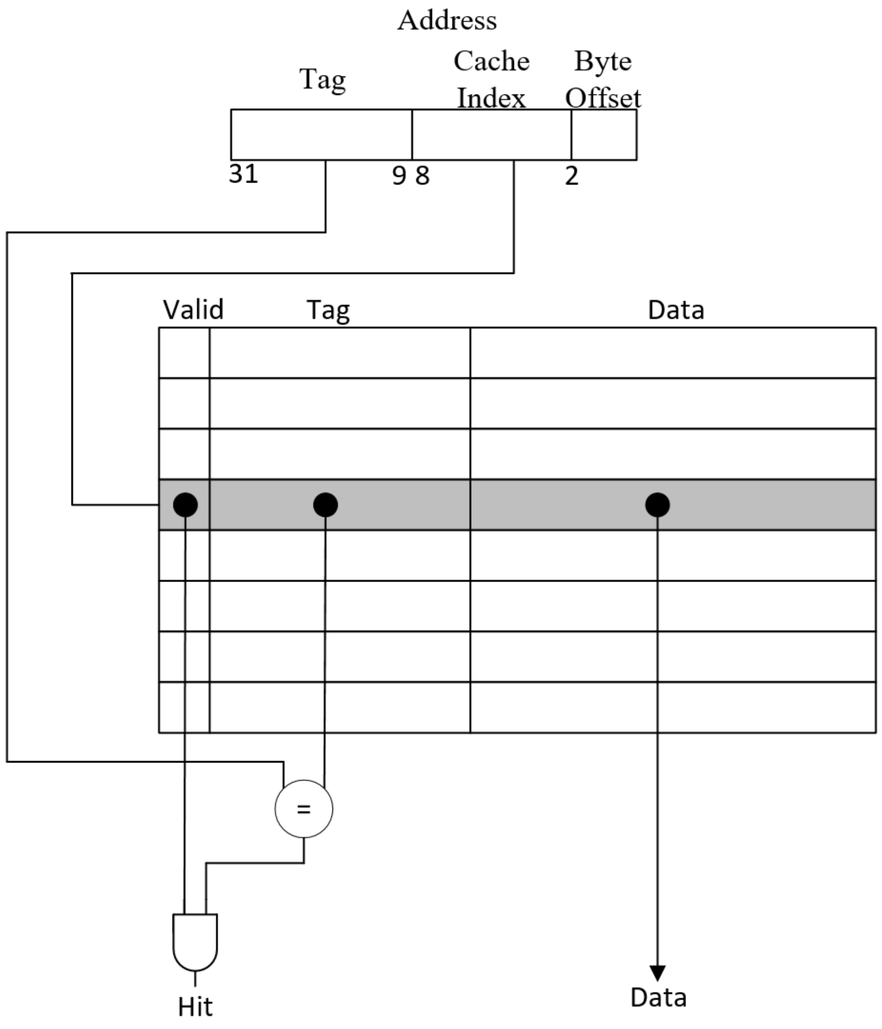


Figure 3.4.1: Cache Design with direct mapping

3.4.2 Instruction Cache Controller

Instruction cache holds some of the instructions that the processor is going to execute. The instructions mainly exist in the RAM, therefore when the processor first starts, the instruction cache is empty and needs to be filled with instructions first before the execution begins. After the status of the address is read, match signal and the data to the referred address are sent to the cache control, if there is a cache hit, the instruction will proceed to the next pipeline stage. If a cache miss occurs, a NOP instruction will be inserted. Noting that the hit and miss signals are forwarded to the thread management unit to restore the address of the instruction that the cache missed and switch to a different thread.

As the processor is multithreaded, more than one instruction cache miss can occur while the instruction of the previous miss has not been retrieved from the main memory. Therefore, a queue-like system was designed in the cache control to hold the address of the next RAM request. Noting that the design does not allow more than one instruction miss per thread, because the thread management unit will not allow any thread to proceed unless the instruction has been successfully retrieved from the memory, for this reason, the queue-like size is the number of threads running in the processor as one pending request for each thread. In order to allow fairness between threads request, the controller will always deal with the requests based on increment of the previous thread, for example, if the last miss occurred in thread 2, when retrieving the instruction is done, the cache will check if there are any pending requests for thread 3,4 and 1 respectively with reference to the last missed thread. If there are no pending requests and the current address has a miss, the instruction of the intended address will be retrieved from the RAM. The out of order approach with this queue can be convenient in a multithreaded processor because any instruction retrieved back from the memory means the processor will keep running and the order of the requests does not really matter. The Cache Controller behavioral module can be found in appendix C.

3.4.3 Data Cache Controller

The data cache controller can be divided into two main parts, load and store, control differ between these two operations as explained in the sections below.

3.4.3.1 Load operation

Load instruction reads the cache status for the address provided to it like the instruction cache. The status of the cache is sent to the controller to inform it if there is a cache miss or hit. If a hit occurs, the value will be written back to the register file, if cache miss occurs, the cache miss flag will be high, informing the thread management unit to store the address of the load instruction to go back to it when retrieving the data is done. Design of load queue-like exists in the controller, similar approach discussed earlier in the instruction cache controller with pending requests is done in the data cache controller, noting that the design does not allow any thread to have more than one data cache miss at a time, therefore the size of the queue-like is the number of threads running in the processor.

3.4.3.2 Store operation

Store instruction stores a value in the cache and the RAM. There is no need to read the cache status when writing to the cache, therefore, the controller writes to valid, tag and data cache memories at memory stage as the address and the write data are set during execute stage. The controller was designed to make the write to cache independent from write to memory since storing in the RAM takes at least six cycles to be done. There could be other store requests during that time, therefore, if a store in the cache occurs and the memory is being written to, the address and the write value will be temporarily stored in a queue-like system.

The technique used to write the data back to the RAM is write-through; with this technique a store occurs in the RAM immediately after the data is written to the cache. The queue-like size mentioned earlier is two times the number of threads. In this queue, two pending store requests for each thread can be stored. The queue is aligned to store for a thread in position threadID and threadID + number of threads; which means that the same thread is always pointed to the same locations. There are cases where a cache miss happens and the data has not been yet written to the RAM, so the controller will check if the address of the miss is pending to be written in the RAM. If so, the data will be forwarded from the queue and will be written back to the register file. Since each thread has fixed locations, it is easy to track the queue and get the data from the queue if needed. The design allows more than one store operation from the same thread to happen. In case the queue is full for a thread and a third request occurs, store hazard flag will be high informing the thread management unit that the queue cannot hold this store request. The address of the store instruction is sent back to the thread management unit and all instructions for the same thread in the pipeline will be flushed. The queue-like system designed in the project is considered to be good in terms of cost and effective in terms of forwarding a value from the queue without the need to check the entire queue for a certain address and then look for the data in the same queue location the address was found.

3.5 Thread Management Unit

Thread management unit is the unit responsible for maintaining the program counter, keeping thread status and switch between threads where necessary. Maintaining the correct program counter for each thread is a challenging task, there could be different cases where the address must be stored in the program counter, therefore a priority system was designed to maintain the right address figure 3.6.1 shows illustration of the priority system.

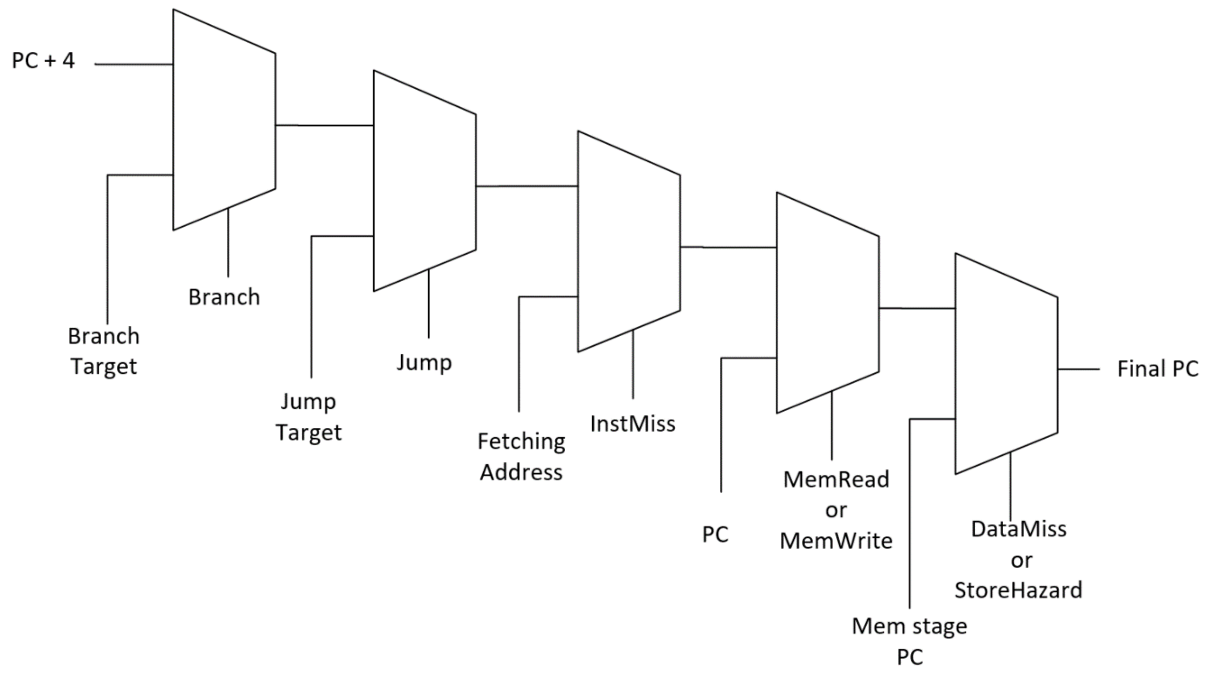


Figure 3.6.1: priority system designed in the thread management unit.

The previous figure shows the system for one thread. It is divided into separate cases, incrementing the program counter should take place every time the thread runs. The branch and jump target address must be calculated in case of branch or jump, where these can be considered as the static conditions for changing the program counter. The dynamic change comes with events happening in the processor. The first case is the instruction cache miss; the program counter should be restored until the instruction is back from the memory. The second case is when there are a load or store instruction, because the switch between threads is dynamically triggered every time one of these two instructions is executed, therefore, the address of the instruction currently being fetched in the instruction cache should be restored in the program counter by keeping the same address. The highest priority of maintaining the correct address is when a data cache miss or store hazard explained earlier happens, the address of the PC in the memory stage should be restored and saved until the thread runs again. With these five multiplexers each connected to the input of the next one, the program counter will be stored with the proper address. In conclusion, the priority is based on how deep the instruction is in the pipeline, the address of the oldest instruction should be restored.

The switch between threads can be classified as static and dynamic switch. The Dynamic switch occurs when one of the cache memories miss. It is dynamic since it is not predictable and can occur at any time during the execution. Static switch between threads occurs when an instruction may cause any of the caches to miss. For example, with load operation, if a switch occurs and the load miss in the cache, there is no need to flush the pipeline since the pipeline at that time belongs to another thread, store follows the same logic but with store hazard instead of cache miss. Another static switch occurs when a control transfer instruction takes place in the pipeline. We can predict that there might be an instruction cache miss because the address has changed. Since the static switch does not have a penalty because all the previous situation can be detected at ID stage, the address of the next instruction can be changed immediately before setting the address of the next instruction, therefore, with the static switch the performance of the processor is enhanced.

Since the program counter of each thread is separated from one another, enable signals for each program counter should exist under the conditions presented earlier, Figure 3.6.2 shows the RTL code of the enable logic, noting that conditions here may overlap, but the priority system presented earlier resolves this overlapping.

```
always_comb

begin

      EnablePCThread = 4'b0000; // default value

      if (InstMiss || InstHit || jump_ID || branch_ID) EnablePCThread
      [mhartID_ID] = 1; // Enable PC of thread in ID stage

      if (SwitchThread && !DontFetch && PendingSameAddress != 4'b1111 )
      EnablePCThread [nextthread] = 1; // Enable PC for next thread in the
      pipeline

      if (InstReady && (RetrievingDoneFor == nextthread))
      EnablePCThread[nextthread] = 1; // Enable PC for next thread in the
      pipeline

      if (CacheMiss) EnablePCThread [mhartID_Mem] = 1;// Enable PC of
      thread in Mem stage

      if (StoreHazard && StoreHazardVector[mhartID_Mem] == 1)
      EnablePCThread [mhartID_Mem] = 1; ;// Enable PC of thread in Mem
      stage

end
```

Figure 3.6.2: Enable signals for the program counters

The previous figure shows that the enable for different threads is dynamic based on the cases presented earlier, in case of instruction miss, hit, jump and branch, the thread in the decode stage should change the program counter. The second case is when a switch between threads occur, the next thread program counter must be incremented, therefore, the program counter should be stored. In case there is no thread ready for execution, DontFetch signal will be high to indicate so. PendingSameAddress signal is useful when the threads are running in a shared program area and the address of all threads are the same. The difference between DontFetch and PendingSameAddress is that DontFetch Will trigger the SwitchThread function in the next cycle, while PendingSameAddress will keep the same thread at the next cycle since all of them will be ready when the instruction is back from the memory.

InstReady signal tells the thread management unit that the instruction has been successfully retrieved from the memory. nextthread Signal represents the thread that will take place next in the pipeline. CacheMiss refers to data cache miss and StoreHazard was explained earlier in this chapter. Since two store instructions can take place in the pipeline, the oldest store should be restored, therefore before enabling the program counter for that thread the code checks whether there is a previously announced store hazard.

The next thing that the thread management unit is responsible for is choosing the next thread to be executed in case of switch request, Figure 3.6.3 shows the RTL code for choosing the next thread logic.

```
always_comb
begin
      nextthread = mhartID;// default value
      DontFetch = 0;// default value
      if (SwitchThread)
      begin
      if (DataVector [mhartID + 1'b1] == 1
      && InstVector [mhartID + 1'b1] == 1
      && PendingSameAddress [mhartID + 1'b1] == 0
      && StoreHazardVector [mhartID + 1'b1] == 1
      && !(StoreHazard && mhartID_Mem == mhartID + 1'b1))
      nextthread = mhartID + 1'b1;
end
```

Figure 3.6.3: Choosing the next thread logic

The previous code shows how the next thread is selected; the status of each thread must be read before choosing it to be the next thread in the pipeline. The conditions above check if there are no data or instruction currently being retrieved from the RAM for this thread and the thread address is not the same as an address that has an instruction miss and that there is no store hazard for the next thread. The previous operations are done with respect to the last thread that was running.

Finally, the thread management unit is responsible for maintaining the status of each thread, Figure 3.6.4 shows the RTL code for updating the status of each thread.

```
always_ff @ (posedge clk,negedge nReset)
begin
    if (!nReset)
    begin
       DataVector <= 4'b1111;
       InstVector <= 4'b1111;
       StoreHazardVector <= 4'b1111;
    end
    else
    begin
       if (InstMiss && !IgnoreMiss) InstVector[mhartID_ID] <= 0;
       if (DoneRetrieving) InstVector [RetrievingDoneFor] <= 1;

       if (StoreHazard) StoreHazardVector [mhartID_Mem] <= 0;
       if (DoneWritingData) StoreHazardVector [DoneWritingFor] <= 1;

       if (MemRead) DataVector [mhartID] <= 0;
       if (CacheHit)DataVector[mhartID_Mem] <= 1;
       if (CacheMiss) DataVector[mhartID_Mem] <= 0;
       if (DoneReadingData) DataVector [DoneForTID] <= 1;
    end
```

Figure 3.6.4: the RTL code for updating the status of each thread.

The previous code shows the vectors related to the thread status, if the value of the vector at position thread ID is one, it means that the thread with respect to the vector is ready for execution, if not, it will not be allowed to go through the pipeline as presented earlier in the next thread logic, noting that in case of memory read, the thread cannot be executed again unless a cache hit or done reading data flag is high to indicate that the thread is ready to be executed, because if the thread is executed again the pipeline must be flushed because the execution of the instruction might be wrong. IgnoreMiss signal tells the thread management unit that this miss should not be considered, this case happens when one or more threads are waiting for the same address.

Chapter 4

Testing and Results

Since day one in the project, verification of the system was required to test all functionalities in the design and ensure that the execution of the program is done correctly. Verifying a processor is not an easy task to achieve, especially if it is done using waveforms, therefore, a testbench was developed for the system in order to keep track of instructions inside the processor and use the testbench to stop the simulation in case of an error in the design.

4.1 Testbench

The testbench of the system was written in SystemVerilog hardware description language. ModelSim simulations tool was used to simulate the system, the testbench of the design gives the following statistics about the design:

1. Total cycles the simulation ran
2. Number of instructions miss times
3. Number of data miss times
4. Number of store hazards
5. Number of times the context switch occurred
6. Number of cycles the processor was idle and not executing instructions
7. Instructions per clock

Verifying memory system and verifying that the proper value is loaded from the memory was a challenging task. In order to do so, an identical memory of the main memory was created in the testbench, whenever a value is stored in the memory it would be stored in the memory inside the testbench as well, verification comes when the values are read from the cache memory. Since the address in the cache should fetch the same data inside the main memory. A comparison between the loaded data from the cache and the memory inside the testbench would take place. The simulation was stopped anytime the values were not the same. This method helped to debug the controllers of the cache and to verify corner cases of store and load operations.

4.2 RISC-V GCC

The tests of the design were generated by a pre-built RISC-V compiler provided by SiFive and it is available on their website. The compiler is compatible with windows, in order to use it properly with the commands of the compilation, GIT bash software was downloaded and used. GIT bash is a Linux-based commands that run on windows. In order to generate the proper files of the C code, the following commands was used:

1. `riscv64-unknown-elf-gcc -nostartfiles -march=rv32im -mabi=ilp32 main.c -o main.elf -mpreferred-stack-boundary=3 -T linker.ld`

   This command is responsible for the specification of the instruction set that were used. `-nostartfiles` option is to ignore the initialization sequence of the processor since it is not required in the design, `-march` and `-mabi` are used to specify that the processor is 32 bit and supports I and M extensions, `-o` is to specify the output file, `-mpreferred-stack-boundary` changes the order of the stack. By default, the compiler assumes that the data in the memory is 128 bits since it is the largest number of bits supported by RISC-V, with the previous command the compiler data width is modified to match 32 bits. Finally, `-T` is used for the linker script that was created in order to specify the location of the data and the instructions.

2. `riscv64-unknown-elf-objdump -d main.elf > inst.txt`

   This command is responsible for writing the assembly of the RISC-V in inst.txt file, `-d` option is to disassemble from the output of the compiler to a readable format.

3. `riscv64-unknown-elf-objcopy -O verilog main.elf inst.vh`

   The final command is used in order to convert the compiled files into Verilog hexadecimal memory files.

4.3 Programs

The test of a multithreaded system can vary with the structure of the code, the test of the system was built with the most exhaustive instructions to run on CGMT. The programs contain a large amount of load, store and control transfer instructions, which means that the results of the following program indicate how efficient is the processor in terms of a continuous misses in both caches even with a higher number of threads which allowed to expose the bottleneck of the system in terms of efficiency.

All the following functions are recursive to test the system with a lot of memory access requests. For recursive functions a large amount of stack memory usage with continuous access of different memory locations allowed the data cache to miss a large number of times, noting that since all the stack pointers start with index 0 in the cache, the data were continuously overwritten even when store requests are one cycle apart.

The C code of the test can be found in appendix C. The programs that were included in the tests are the following:

1. Multiplication of N by 3, the multiplication is done in a recursive addition way, where the stack stores all numbers from N to 0 as a counter of how many times the addition should take place.

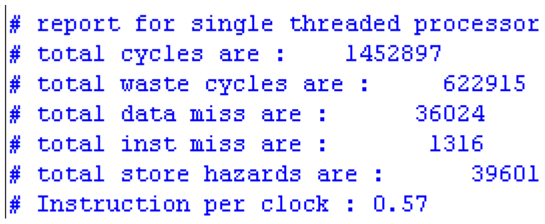
2. Multiplication of N by 4, this function is similar to the previous one.

3. Fibonacci series, this function finds position N in the Fibonacci sequence, where N is the parameter passed to the function. This function continuously accesses the memory since the recursive calculations are based on the addition of previous outputs in the stack.

4. Recursive addition of numbers from 0 to N, where N is the parameter passed to the function.

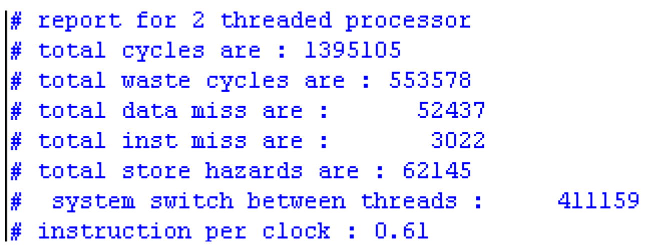
4.4 Results

Figures 4.3.1 to 4.3.3 show the generated report by the testbench of single, two and four threaded processors respectively, noting that the simulation would stop after the same set of inputs for all previous four functions are completed.



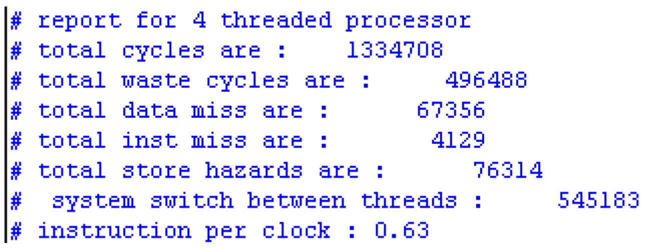
```
# report for single threaded processor
# total cycles are :    1452897
# total waste cycles are :     622915
# total data miss are :      36024
# total inst miss are :       1316
# total store hazards are :      39601
# Instruction per clock : 0.57
```

Figure 4.3.1: Report generated for single threaded processor.



```
# report for 2 threaded processor
# total cycles are : 1395105
# total waste cycles are : 553578
# total data miss are :      52437
# total inst miss are :       3022
# total store hazards are : 62145
#  system switch between threads :     411159
# instruction per clock : 0.61
```

Figure 4.3.2: Report generated for two threaded processor.



```
# report for 4 threaded processor
# total cycles are :    1334708
# total waste cycles are :     496488
# total data miss are :      67356
# total inst miss are :       4129
# total store hazards are :      76314
#  system switch between threads :     545183
# instruction per clock : 0.63
```

Figure 4.3.3: Report generated for four threaded processor.

In order to evaluate the improvement in performance between the previous test, the following formula must be used:

$$system\ improvement = \frac{\#of\ cycles\ in\ n\ threads - \#of\ cycles\ in\ m\ threads}{\#of\ cycles\ in\ n\ threads}$$

The previous test shows an overall improvement by 3.98% between single and two threads, with improvement on IPC. Four threaded processor improvement over the single-threaded is 8.13%, where the improvement between two and four threaded is 4.32%.

One important thing to note that a multithreaded processor does not improve the performance of a single thread. It improves the performance of the overall processor efficiency.

The previous test is the one that has been used in the demonstration. Noting that it was pointed out that the number of data cache miss in two threads was higher than the four threads in the demonstration. The reason behind that was the compiler. As mentioned before in section 4.2, there is a command to modify the stack frame as by default it is 128 bits. The command was accidentally not used which resulted in lines being empty in the cache and only half of the cache memory was used. The test was modified after and the number was logical since two threads should not miss the cache more than four threads.

In conclusion, CGMT architecture increased the performance of a single core as expected. The improvement was proved by testing the system. The tests that were created was supposed to test a heavy program to run on this architecture and compare the results. The obtained results can be considered as a minimal gain since the programs included a massive amount of memory access. On the other hand, as mentioned in the previous reports. Store hazards had a huge impact on the design. Which can conclude that write-through technique is not recommended in case a higher performance and better results are intended, the huge number of store hazards happened when on board FPGA memory was used. In this memory it takes at least 6 cycles to complete the store operation and 7 cycles for load operation. Since the main memory is single port memory. The number of delays caused by the store affected the overall performance. Increasing the capacity of the queue would result in a processor with high cost which would not be suitable for embedded processor.

An important thing to add as well is the most achievable frequency of each processor was not calculated due to lack of time at the end of the project.

Chapter 5

Conclusion

The aim of the project was to design a multithreaded pipelined processor. The design was achieved with a 5-stage pipelined RISC-V processor which has coarse-grained multithreaded architecture. The analysis took place to analyze the results and what could have been done differently to enhance the performance. The processor runs a wide variety of RISC-V instructions. The variety of instruction allowed tests that have been used to run a complicated C program and run a shared program between threads along with private code for each thread. The C program was compiled using RISC-V GCC toolchain and was linked to the processor.

The tests that took place in the processor proved that the processor behaves correctly and executes instructions as intended. The performance gain was calculated and showed an improvement of 3.98% between single and two threaded processors, the improvement between single and four threaded processors was 8.13%. The previous percentages prove that multithreaded architecture increase the performance of a single-core design.

Chapter 6

Further Work

The design can be implemented in different ways in order to test further aspects of CGMT architecture. The cache memory mapping technique can be changed in order to see how the hit and miss ratio change and how can that affect the processor performance.

Additionally, deeper memory hierarchy can be designed rather than level 1 cache memory along with main memory, with deeper hierarchy the miss penalty can be reduced as the access of level 2 and level 3 cache is less than the time needed to access the main memory.

The design does not include a memory that acts as a hard disk as well. Hard disk usually has a bigger size than the RAM, which means that a page fault can occur. Page faults are a result of data not existing in the RAM and it must be retrieved from the hard disk. Accessing the hard disk usually takes the longest time and can cause the system to have more delays.

Finally, bit manipulation instruction like store-byte and store half-word were not implemented during the project. Adding these instructions can result in further testing and more programs running on the processor.

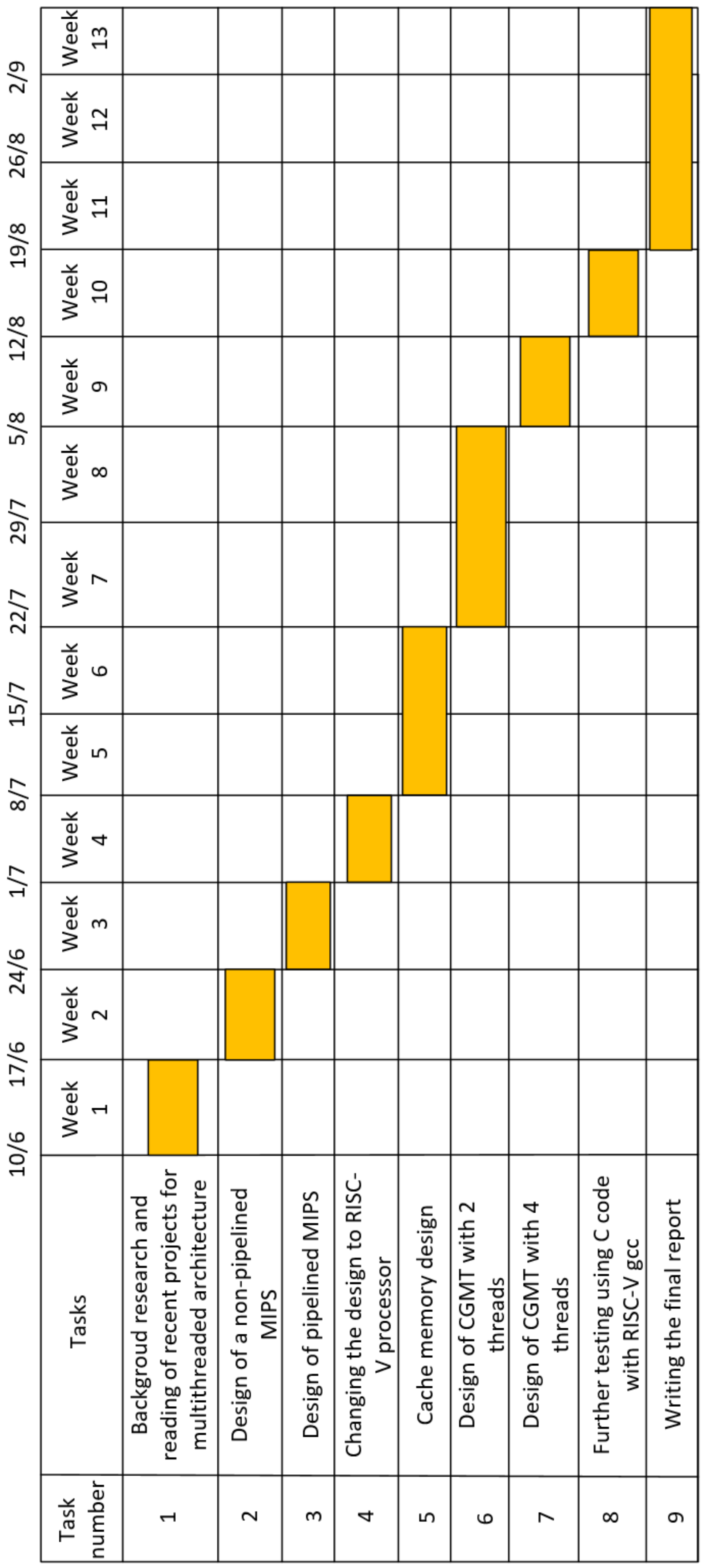
Chapter7

Project Management

At the beginning of the project. A non-pipelined MIPS processor was designed. The pipelined architecture was then successfully added to the processor. After some discussions with my supervisor. We found out that a RISC-V processor would be more convenient for this project, therefore, the design was changed from MIPS to RISC-V. After designing the processor, cache memory design took place and many papers and books were read in order to achieve a functional and realistic memory design. The multithreaded architecture was added then and many tests were conducted in order to verify that the system works properly. Tests were made in order to analyze what is the best number of threads to run on a coarse-grained multithreaded architecture to increase the performance. In order to do that, two and four threaded processors had to be designed.

At the beginning of the project, RIPES assembler was used in order to write the tests for the processor. Afterwards, it was much more efficient to run the codes using RISC-V GCC, all programs were written in C programming language. Working with the compiler took place in the last two weeks of the project time, therefore, not all the chapters of the report were reviewed by the supervisor because there was not enough time for me to test the system and write the results in the report by the review time.

In conclusion, the project was completed on time and the results of the testing were analyzed in order to conclude the project and write future work part for students taking over my project in the future. Figure 7.1 shows the Gantt chart of the actual time spent on each part of the project.



		10/6	17/6	24/6	1/7	8/7	15/7	22/7	29/7	5/8	12/8	19/8	26/8	2/9
Task number	Tasks	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13
1	Backgroud research and reading of recent projects for multithreaded architecture	🟨												
2	Design of a non-pipelined MIPS		🟨											
3	Design of pipelined MIPS			🟨										
4	Changing the design to RISC-V processor				🟨									
5	Cache memory design					🟨	🟨							
6	Design of CGMT with 2 threads							🟨	🟨					
7	Design of CGMT with 4 threads									🟨				
8	Further testing using C code with RISC-V gcc										🟨			
9	Writing the final report											🟨	🟨	🟨

Figure 6.1: Gantt chart with the timeline of the project

References

[1] A. W., & K. A. (Eds.). (2017, May 7). The RISC-V Instruction Set Manual Volume I: User-Level ISA. Retrieved from https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[2] A. W., & K. A. (Eds.). (2017, May 7). The RISC-V Instruction Set Manual Volume II: Privileged Architecture. Retrieved from https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

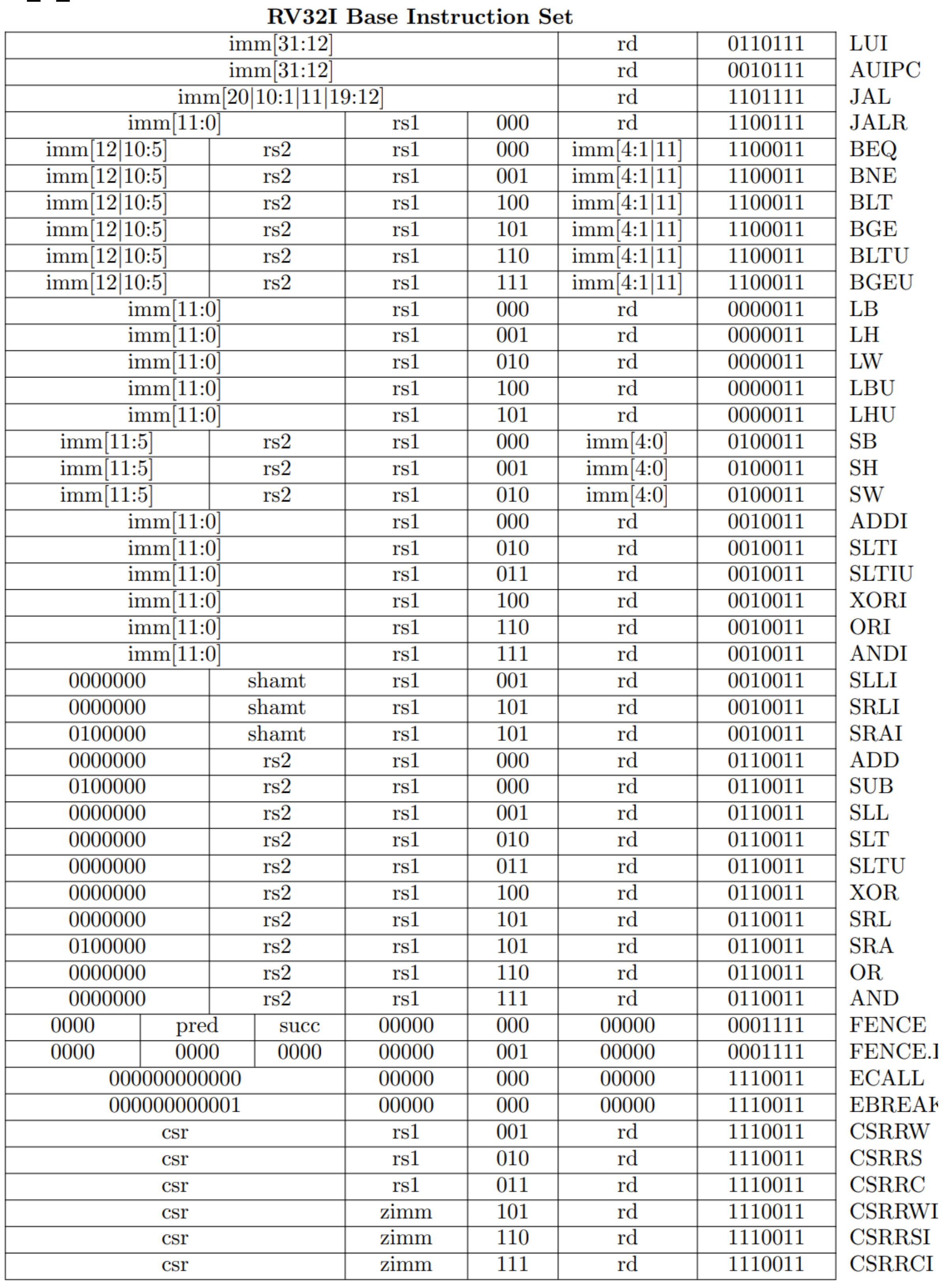
[1] Hennessy, J. L., & Patterson, D. A. (2018). *Computer organization and design risc-v edition*. Boston, MA: Elsevier.

[4] Hennessy, J. L., & Patterson, D. A. (2019). Memory Hierarchy Design in *Computer architecture: a quantitative approach* (pp. 78-84). Cambridge, MA: Elsevier.

[5] Hennessy, J. L. & Patterson, D. A., (2019). *Computer organization and design: the hardware/software interface*. Brantford, Ontario: W. Ross MacDonald School Resource Services Library.

[6] Nemirovsky, M., & Tullsen, D. M. (2013). Multithreading Architecture. *Synthesis Lectures on Computer Architecture*, *8*(1), 1–109. doi: 10.2200/s00458ed1v01y201212cac021

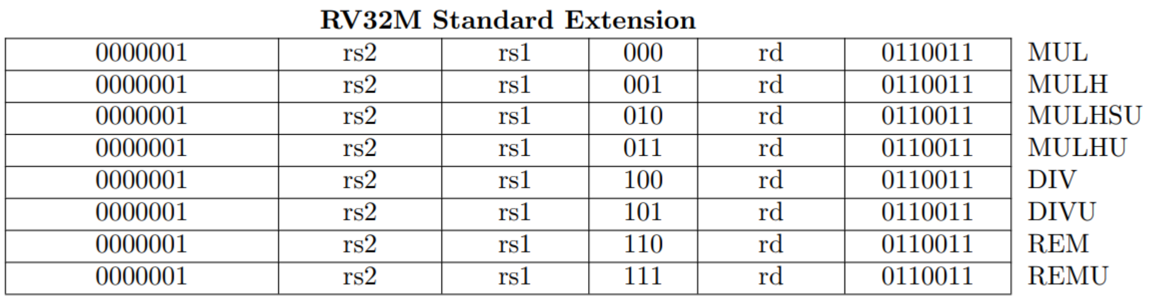
Appendix A: RV32I RISC-V Instructions



RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Appendix B:RV32M RISC-V Instructions



RV32M Standard Extension

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

Appendix C: RTL code of the system

1 Top level module (Pipelined_RISC)

```
//
// File Name: pipelined_RISC.sv
// Function: top level module of the processor
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module pipelined_RISC(
// Sdram
input logic clk,nReset,
output logic LED0,LED1,LED2,LED3,
output logic [31:0] T0out,T1out,T2out,T3out
);

 logic [31:0] I_EX,currentPC_ID,currentPC_EX,currentPC_Mem;
 logic [31:0] Rdata1_EX,Rdata2_EX,imm_EX;
 logic
RegWrite_EX,ALUSrc_EX,MemWrite_EX,MemRead_EX,MemtoReg_EX,/*branch_ID,jump_I
D,*/store_address_EX,aui_EX;
 logic [4:0] ALUOpin_EX;
 logic [31:0] Wdata_Mem,Rdata2_Mem;
 logic RegWrite_Mem,MemWrite_Mem,MemRead_Mem,MemtoReg_Mem;
 logic [31:0] Ram_out_WB,Wdata_WB;
 logic RegWrite_WB,branch_ID,jump_ID,MemtoReg_WB;
 logic [31:0]
I_ID,newpc_ID,newpc_EX,RBranch_EX,absolute_jump_ID,out,MemAddress;
```

```
 logic [4:0]  Waddr_Mem,Waddr_WB;

 logic InstMiss,CacheMiss,SwitchAction,flush,branch,CacheHit,MemRead;

 logic [1:0] ForwardA,ForwardB,ForwardRS1,ForwardRS2;

 logic [1:0]
mhartID,mhartID_ID,mhartID_EX,mhartID_Mem,mhartID_WB,DoneForTID,RetrievingD
oneFor,FetchingmhartID,nextthread,DoneWritingFor,ByteEnable;

 logic [31:0] RamReadAddress,RamWriteAddress,InstAddress;

 logic [31:0] RamWriteData,IntmhartID,RamByteAddress;

 logic [19:0] addr;

 wire [15:0] data;

 logic WE,CS,OE,LBS,HBS;

 logic [7:0] InstOut;

 logic [11:0] PendingStoreRequest;

 logic [15:0] RamData,RamByteData;

 logic DoneReading,DoneWriting;

 logic RamWrite,RamRead,InstRead,StoreHazard;

 //logic [7:0] RamByteData;

 logic [31:0]
InstByteAddress,newPC,ThreadAddress,FetchingAddress,DataCumulativeRegister,
InstCumulativeRegister;

 logic
DoneWritingData,DoneReadingData,DoneReadingInst,MemWrite,InstHit,DoneRetrie
ving,ReadEnable;

 logic Ignore,ReadCacheStatus,RamReadEnable,RamWriteEnable;

 logic InstSource,IgnoreMiss,flushEX,flushID;

 // external memory ports

 logic [7:0] address;

 logic [31:0] datafromexternal;

 logic ReadAccess,ReadFromExternal;
 thread_management TM1 (
       .clk,.nReset,.flushEX,.flushID,

.branch_ID,.jump_ID,.MemRead,.mhartID_Mem,.mhartID,.FetchingAddress,.DoneWr
itingData,

.RBranch_EX,.absolute_jump_ID,.currentPC_Mem,.currentPC_ID,.Ignore,.InstRea
d,.InstAddress,

.InstMiss,.InstHit,.CacheMiss,.MemWrite,.flush,/*.FetchingmhartID,*/.ReadCa
cheStatus,.InstReady(DoneReadingInst),

.RetrievingDoneFor,.DoneRetrieving,.DoneReadingData,.DoneForTID,.nextthread
,.IgnoreMiss,.mhartID_EX,.DoneWritingFor,

.newPC,.ThreadAddress,.InstSource,.IntmhartID,.StoreHazard,.CacheHit,.Switc
hAction,.mhartID_ID,.ReadFromExternal
 );


 externalmemory  EM1 (
      .clk,.ReadAccess,.ReadFromExternal,.address,.datafromexternal
 );



  ROMcontroller ROMC1(
 .InstAddress,
 .InstOut,
 .clk,.nReset,.InstRead,
 .InstByteAddress,.InstCumulativeRegister,
 .DoneReadingInst
);

RamController RC1(
 .RamReadAddress,.RamWriteAddress,
 .RamWriteData,
 .RamData,
 .RamWrite,.RamRead,.clk,.nReset,.DoneReading,.DoneWriting,
 .ByteEnable,
 .RamByteData,
 .RamByteAddress,.DataCumulativeRegister,
 .DoneWritingData,.DoneReadingData,
 .RamReadEnable,.RamWriteEnable
);

 SRAMController SRC1 (
// SRAM INTERFACS
.addr,
.data,
.WE,.CS,.OE,.LBS,.HBS,.DoneReading,.DoneWriting,
// PROCSSSOR Inputs and Outputs
.RamReadEnable,.RamWriteEnable,.clk,.nReset,
.ByteEnable,
.RamByteAddress,
.RamByteData,
.RamData
);

asynchronous_ram AR1
```

```
(
.CS,.WE,.OE,.LBS,.HBS,
.addr,
.data
);


   InstROM  IR1(
 .InstByteAddress,
 .InstRead,.clk,
 .InstOut
);

 IF IF1(
      .clk,.nReset,.flush,
      .I_ID,.newpc_ID,.currentPC_ID,.nextthread,
      .newPC,.FetchingAddress,.branch_ID,.jump_ID,
      .ThreadAddress,.Ignore,.FetchingmhartID,
      .InstReady(DoneReadingInst),
      .InstOut(InstCumulativeRegister),
      .InstAddress,.IgnoreMiss,
      .InstHit,.InstMiss,.mhartID,.mhartID_ID,
      .InstRead,.ReadCacheStatus,
      .RetrievingDoneFor,.DoneRetrieving,.InstSource
      );

 ID ID1(
      .I_ID,.newpc_ID,.Wdata_Mem,.Wdata_WB,.currentPC_ID,.out,
        .clk,.RegWrite_WB,.nReset,.mhartID_ID,.mhartID_WB,.flush(flushID),

        .ForwardRS1,.ForwardRS2,.Waddr_WB,.Rdata1_EX,.Rdata2_EX,.currentPC_EX
,

        .RegWrite_EX,.ALUSrc_EX,.MemWrite_EX,.MemRead_EX,.MemtoReg_EX,.branch
_ID,.jump_ID,

        .store_address_EX,.branch,.aui_EX,.mhartID_EX,.MemRead,.MemWrite,.Int
mhartID,

        .ALUOpin_EX,.I_EX,.newpc_EX,.imm_EX,.RBranch_EX,.absolute_jump_ID

        );


 EX EX1(

.Rdata1_EX,.Rdata2_EX,.RegWrite_EX,.ALUSrc_EX,.flush(flushEX),.MemWrite_EX,
.MemRead_EX,.MemtoReg_EX,.ReadAccess,.address,.LED0,.LED1,
.LED2,.LED3,.store_address_EX,.clk,.ALUOpin_EX,.I_EX,.Waddr_Mem,.ReadEnable
,.Wdata_Mem,.Rdata2_Mem,.RegWrite_Mem,.MemWrite_Mem,.MemRead_Mem,.MemtoReg_
Mem,.nReset,.Wdata_WB,.ForwardA,.ForwardB,.newpc_EX,.imm_EX,.currentPC_Mem,
.currentPC_EX,.aui_EX,.out,.mhartID_EX,.mhartID_Mem,.MemAddress,.T0out,.T1o
ut,.T2out,.T3out);


 Mem Mem1(

.Wdata_Mem,.Rdata2_Mem,.MemAddress,.Waddr_Mem,.RegWrite_Mem,
.MemWrite_Mem,.MemRead_Mem,.MemtoReg_Mem,.clk,.nReset,
.mhartID_Mem,.DoneWritingFor,.DoneReadingData,.DoneWritingData,.ReadEnable,
.Wdata_WB,.Waddr_WB,.mhartID_EX,.ReadFromExternal,.datafromexternal,.RegWri
te_WB,.CacheHit,.StoreHazard,.CacheMiss,.mhartID_WB,.RamData(DataCumulative
Register),.RamReadAddress,.RamWriteAddress,.RamWriteData,.RamRead,.RamWrite
,.DoneForTID);


 forwarding FW1(

        .RegWrite_WB,.RegWrite_EX,.RegisterRS1_ID(I_ID[19:15]),

        .RegisterRS1_EX(I_EX[19:15]),.RegisterRS2_ID(I_ID[24:20]),

        .RegisterRS2_EX(I_EX[24:20]),.Waddr_WB,.Waddr_Mem,.ForwardA,

       .ForwardB,.ForwardRS1,.ForwardRS2,.RegWrite_Mem,.Waddr_EX(I_EX[11:7]),
	.mhartID_Mem,.mhartID_WB,

	.mhartID_ID,.mhartID_EX

	);

Endmodule
```

2. Thread Management Unit

```
//

// File Name: thread_management.sv

// Function: management of threads PC

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module thread_management(

input logic clk,nReset,

input logic branch_ID,jump_ID,MemRead,ReadFromExternal,

input logic [1:0]
mhartID_Mem,mhartID_ID,mhartID_EX,DoneWritingFor,/*mhartID_ID,*/

input logic [31:0]
RBranch_EX,absolute_jump_ID,currentPC_Mem,currentPC_ID,FetchingAddress,Inst
Address,

input logic InstMiss,InstHit,CacheMiss,MemWrite,InstRead,CacheHit,

input logic
DoneRetrieving,DoneReadingData,IgnoreMiss,InstReady,StoreHazard,DoneWriting
Data,

input logic [1:0] RetrievingDoneFor,DoneForTID,

output logic
flush,Ignore,InstSource,ReadCacheStatus,SwitchAction,flushID,flushEX,

output logic [1:0] mhartID,nextthread,

output logic [31:0] ThreadAddress,newPC,IntmhartID

);

     logic [31:0]
T0Mux1,T0Mux2,T0Mux3,T1Mux1,T1Mux2,T1Mux3,T2Mux1,T2Mux2,T2Mux3,T3Mux1,T3Mux
2,T3Mux3;

      logic [3:0]
DataVector,InstVector,PendingSameAddress,StoreHazardVector;

      logic [31:0]
PCBRANCHT0,PCBRANCHT1,finalpcT0,finalpcT1,newPCT0,newPCT1;

      logic [31:0]
PCBRANCHT2,PCBRANCHT3,finalpcT2,finalpcT3,newPCT2,newPCT3,T0,T1,T2,T3;

      logic [3:0] EnablePCThread;

      logic [31:0] JustFinishedForAddress;

      logic MemAccessRegister;

      assign IntmhartID = {30'b0,mhartID};

      assign ReadSuccessful = (CacheHit || ReadFromExternal);
      //logic nextthread;

      logic
initialisation,DontFetch,DontFetchRegister;//,NoAvailableInstruction;

      assign flushEX = ((StoreHazard || CacheMiss) && mhartID_EX ==
mhartID_Mem) ? 1'b1:1'b0;

      assign flushID = ((StoreHazard || CacheMiss) && mhartID_ID ==
mhartID_Mem) ? 1'b1:1'b0;


      pc pc1 (.PCin(T0Mux3),.PCout(T0),.clk,.nReset,

      .EnablePC(EnablePCThread[0]));


      pc pc2 (.PCin(T1Mux3),.PCout(T1),.clk,.nReset,

      .EnablePC(EnablePCThread[1]));


      pc pc3 (.PCin(T2Mux3),.PCout(T2),.clk,.nReset,

      .EnablePC(EnablePCThread[2]));


      pc pc4 (.PCin(T3Mux3),.PCout(T3),.clk,.nReset,

      .EnablePC(EnablePCThread[3]));


      // PC and store Address logic

        assign SwitchThread = (SwitchAction || (DontFetchRegister &&
!MemWrite));

        assign SwitchAction = (branch_ID || jump_ID || MemRead || InstMiss
|| MemAccessRegister );

        assign InstSource = 1'b1;

        assign Ignore = (!initialisation) ? 1'b1 : 1'b0;

        assign flush = (InstMiss || MemWrite) ? 1'b1 : 1'b0;

        always_comb

        begin

        PendingSameAddress = 4'b0000;

        if ((InstMiss && FetchingAddress == T0 && JustFinishedForAddress !=
T0) || (InstRead && InstAddress == T0)) PendingSameAddress [0] = 1;

        if ((InstMiss && FetchingAddress == T1 && JustFinishedForAddress !=
T1) || (InstRead && InstAddress == T1)) PendingSameAddress [1] = 1;

        if ((InstMiss && FetchingAddress == T2 && JustFinishedForAddress !=
T2) || (InstRead && InstAddress == T2)) PendingSameAddress [2] = 1;
```

```
        if ((InstMiss && FetchingAddress == T3 && JustFinishedForAddress !=
T3) || (InstRead && InstAddress == T3)) PendingSameAddress [3] = 1;


        newPCT0 = T0 + 3'b100;

       newPCT1 = T1 + 3'b100;

        newPCT2 = T2 + 3'b100;

        newPCT3 = T3 + 3'b100;

       ///newPC = (nextthread) ? newPCT1 : newPCT0;

         case (nextthread)

         0: newPC = newPCT0;

         1: newPC = newPCT1;

         2: newPC = newPCT2;

         3: newPC = newPCT3;

         endcase

         // next PC logic

       PCBRANCHT0 = (branch_ID && mhartID == 0) ? RBranch_EX : newPCT0;

        finalpcT0 = (jump_ID && mhartID == 0) ? absolute_jump_ID :
PCBRANCHT0;

        PCBRANCHT1 = (branch_ID && mhartID == 1) ? RBranch_EX : newPCT1;

        finalpcT1 = (jump_ID && mhartID == 1) ? absolute_jump_ID :
PCBRANCHT1;

        PCBRANCHT2 = (branch_ID && mhartID == 2) ? RBranch_EX : newPCT2;

        finalpcT2 = (jump_ID && mhartID == 2) ? absolute_jump_ID :
PCBRANCHT2;

        PCBRANCHT3 = (branch_ID && mhartID == 3) ? RBranch_EX : newPCT3;

        finalpcT3 = (jump_ID && mhartID == 3) ? absolute_jump_ID :
PCBRANCHT3;

        // ThreadAddress for cache access

        case (nextthread)

         0: ThreadAddress = ((branch_ID || jump_ID) && mhartID_ID == 0 ) ?
finalpcT0 :  T0;

         1: ThreadAddress = ((branch_ID || jump_ID) && mhartID_ID == 1 ) ?
finalpcT1 :  T1;

         2: ThreadAddress = ((branch_ID || jump_ID) && mhartID_ID == 2 ) ?
finalpcT2 :  T2;

         3: ThreadAddress = ((branch_ID || jump_ID) && mhartID_ID == 3 ) ?
finalpcT3 :  T3;

         endcase
        // final value stored in the program counter is based on
priorities, priorities are from low to high from Mux1 to Mux4, where
priority is based on the olddest address in a thread

        // Thread one store address cases

        newPCT0 = T0 + 3'b100;

        PCBRANCHT0 = (branch_ID && mhartID == 0) ? RBranch_EX : newPCT0;

        finalpcT0 = (jump_ID && mhartID == 0) ? absolute_jump_ID :
PCBRANCHT0;

        T0Mux1 = (InstMiss && mhartID_ID == 0 ) ? FetchingAddress :
finalpcT0;

        T0Mux2 = ((MemRead  || MemWrite)&& mhartID == 0 )  ? T0 : T0Mux1;

        T0Mux3 = ((CacheMiss || StoreHazard) && mhartID_Mem == 0) ?
currentPC_Mem : T0Mux2;

        //Thread two store address cases

        T1Mux1 = (InstMiss && mhartID_ID == 1 ) ? FetchingAddress :
finalpcT1;

        T1Mux2 = ((MemRead  || MemWrite)&& mhartID == 1 )  ? T1 : T1Mux1;

        T1Mux3 = ((CacheMiss || StoreHazard) && mhartID_Mem == 1) ?
currentPC_Mem : T1Mux2;

        // Thread3 store address cases

        T2Mux1 = (InstMiss && mhartID_ID == 2 ) ? FetchingAddress :
finalpcT2;

        T2Mux2 = ((MemRead || MemWrite) && mhartID == 2 )  ? T2 : T2Mux1;

        T2Mux3 = ((CacheMiss || StoreHazard) && mhartID_Mem == 2) ?
currentPC_Mem : T2Mux2;

        // Thread 4 store address cases

        T3Mux1 = (InstMiss && mhartID_ID == 3 ) ? FetchingAddress :
finalpcT3;

        T3Mux2 = ((MemRead  || MemWrite)&& mhartID == 3 )  ? T3 : T3Mux1;

        T3Mux3 = ((CacheMiss || StoreHazard) && mhartID_Mem == 3) ?
currentPC_Mem : T3Mux2;


        end

        // initialisation of the processor

        always_ff @(posedge clk, negedge nReset)

        if (!nReset)

        begin

        initialisation <= 0;

        ReadCacheStatus <= 0;
```

```
         JustFinishedForAddress <= 1;
         DontFetchRegister <= 0;
         MemAccessRegister <= 0;
         end
         else
        begin
        initialisation <= 1;
        ReadCacheStatus <= (DontFetch || PendingSameAddress == 4'hF ||
MemWrite) ? 1'b0 : 1'b1;
        JustFinishedForAddress <= (InstReady) ? InstAddress : 1;
        DontFetchRegister <= (DontFetch) ? 1'b1 : 1'b0;
        MemAccessRegister <= (MemWrite) ? 1'b1 : 1'b0;
        end
        //Invalid cases for Instructions and Data
        always_ff @(posedge clk,negedge nReset)
        begin
        if (!nReset)
        begin
        DataVector <= 4'b1111;
        InstVector <= 4'b1111;
        StoreHazardVector <= 4'b1111;
        end
        else
        begin
        if (InstMiss && !IgnoreMiss) InstVector[mhartID_ID] <= 0;
        if (DoneRetrieving) InstVector [RetrievingDoneFor] <= 1;
        if (StoreHazard) StoreHazardVector [mhartID_Mem] <= 0;
        if (DoneWritingData) StoreHazardVector [DoneWritingFor] <= 1;
        if (MemRead) DataVector [mhartID] <= 0;
        if (ReadSuccessful)DataVector[mhartID_Mem]<= 1;
        if (CacheMiss) DataVector[mhartID_Mem] <= 0;
        if (DoneReadingData) DataVector [DoneForTID] <= 1;


        end
        end
```

```
      // this combinational block decides what the next thread is, out of
order thread selection is enabled to avoid any thread from starting while
it cannot be executed

      always_comb

      begin

      nextthread = mhartID;

      DontFetch = 0;

      if (SwitchThread)

      begin

      if (DataVector [mhartID + 1'b1] == 1 && InstVector [mhartID + 1'b1]
== 1 && PendingSameAddress[mhartID + 1'b1] == 0 && StoreHazardVector
[mhartID + 1'b1] == 1 && !(StoreHazard && mhartID_Mem == mhartID + 1'b1))

      nextthread = mhartID + 1'b1;

      else if (DataVector [mhartID + 2'b10] == 1 && InstVector [mhartID +
2'b10] == 1 && PendingSameAddress[mhartID + 2'b10] == 0 &&
StoreHazardVector [mhartID + 2'b10] == 1 && !(StoreHazard && mhartID_Mem ==
mhartID + 2'b10))

      nextthread = mhartID + 2'b10;

      else if (DataVector [mhartID + 2'b11] == 1 && InstVector [mhartID +
2'b11] == 1 && PendingSameAddress[mhartID + 2'b11] == 0 &&
StoreHazardVector [mhartID + 2'b11] == 1 && !(StoreHazard && mhartID_Mem ==
mhartID + 2'b11))

      nextthread = mhartID + 2'b11;

      else if (PendingSameAddress != 4'b1111)  DontFetch = 1;

      end

      end




      always_ff @(posedge clk, negedge nReset)

      begin

      if (!nReset)

      mhartID <= 0;

      else

      mhartID <= nextthread;

      end

      // For Current Thread Enable

      // conditions here may overlap, but the priority system presented in
the multiplixers before resolves the overlapping
```

```
      always_comb

      begin

      EnablePCThread = 4'b0000;

      if (InstMiss || InstHit || jump_ID || branch_ID) EnablePCThread
[mhartID_ID] = 1;

      if (SwitchThread && !DontFetch && PendingSameAddress != 4'hF )
EnablePCThread [nextthread] = 1;

      if (InstReady && (RetrievingDoneFor == nextthread ) ) EnablePCThread
[nextthread] = 1;

      if (InstReady && (RetrievingDoneFor == mhartID)) EnablePCThread
[mhartID] = 1;

      if (CacheMiss ) EnablePCThread [mhartID_Mem] = 1;

      if (StoreHazard && StoreHazardVector[mhartID_Mem] == 1)
EnablePCThread [mhartID_Mem] = 1;

      end

endmodule
```

3 Program Counter (PC)

```
//
// File Name: pc.sv
// Function: Program Counter for the processor
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module pc #(parameter n = 32) // up to 64 instructions
(input logic clk, nReset,EnablePC,
 input logic [n-1: 0] PCin,
 output logic [n-1 : 0]PCout
);

always_ff @ ( posedge clk , negedge nReset) // async reset
  if (!nReset) // sync reset
     PCout <= 0;
  else
  if (EnablePC)
  PCout <= PCin;
endmodule // module pc
```

2 ROM controller

```
//
// File Name: ROMcontroller.sv
// Function: instruction rom controller
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module ROMcontroller (
 input logic [31:0] InstAddress,
 input logic [7:0] InstOut,
 input logic clk,nReset,InstRead,
 output logic [31:0] InstByteAddress,InstCumulativeRegister,
 output logic DoneReadingInst
);
logic [1:0] ByteEnableInstRead,InstByteEnableRegister;
logic ReadingInst,InstReadFlag;
// Byte Enable incremental
always_ff @(posedge clk, negedge nReset)
if (!nReset)
      begin
      ByteEnableInstRead <= 0;
      InstByteEnableRegister <= 0;
      end
else
      begin
            // reading inst Byte Enable
            if (DoneReadingInst)
            ByteEnableInstRead <= 0;
            else if (InstRead)
            begin
            InstReadFlag <= 1;
            ByteEnableInstRead <= ByteEnableInstRead + 1'b1;
            end
            else
            begin
            InstReadFlag <= 0;
            ByteEnableInstRead <= 0;
            end
            if (ReadingInst)
            InstByteEnableRegister <= ByteEnableInstRead;
      end
// Read Inst logic
always_comb
begin
ReadingInst = 0;
InstByteAddress = 0;
      if (InstRead)
      begin
            case (InstByteEnableRegister)
            3:  begin
                  InstByteAddress = InstAddress + ByteEnableInstRead;
                  ReadingInst = 1;
                  end
            2:  begin
                  InstByteAddress = InstAddress + ByteEnableInstRead;
                  ReadingInst = 1;
                  end
            1:  begin
                  InstByteAddress = InstAddress + ByteEnableInstRead;
                  ReadingInst = 1;
                  end
            0:  begin
                  InstByteAddress = InstAddress + ByteEnableInstRead;
                  ReadingInst = 1;
                  end
            endcase
      end
end
```

```
// InstCumulativeRegister
always_ff @(posedge clk, negedge nReset)
if (!nReset)
begin
InstCumulativeRegister <= 0;
DoneReadingInst <= 0;
end
else
begin
      if (InstReadFlag)
      begin
            case (InstByteEnableRegister)
            0: InstCumulativeRegister[7:0] <= InstOut;
            1: InstCumulativeRegister[15:8] <= InstOut;
            2: InstCumulativeRegister[23:16]<= InstOut;
            3: InstCumulativeRegister[31:24]<= InstOut;
            endcase
      end
      if (InstByteEnableRegister == 3) DoneReadingInst <= 1;
      else DoneReadingInst <= 0;
end

endmodule
```

```
//
// File Name: RamController.sv
// Function: main memory controller
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module RamController(
 input logic [31:0] RamReadAddress,RamWriteAddress,
 input logic [31:0] RamWriteData,
 input logic [15:0] RamData,
 input logic RamWrite,RamRead,clk,nReset,DoneReading,DoneWriting,
 output logic [1:0] ByteEnable,
 output logic [15:0] RamByteData,
 output logic [31:0] RamByteAddress,DataCumulativeRegister,
 output logic DoneWritingData,DoneReadingData,
 output logic RamReadEnable,RamWriteEnable
);

logic ReadingData,WritingData,writingHBS,writingLBS,readingHBS,readingLBS;
// Byte Enable incremental
always_ff @(posedge clk, negedge nReset)
if (!nReset)
      begin
      ByteEnable <= 2'b00;
      DoneWritingData <= 0;
      ReadingData <= 0;
      end
else
      begin
      if (RamRead && !WritingData)
      begin
      if (!readingHBS && !readingLBS) ByteEnable <= 2'b01;
      if (DoneReading && readingLBS) ByteEnable <= 2'b10;
      if (DoneReading && readingHBS) ByteEnable <= 2'b00;
      end
      if (RamWrite && !ReadingData)
      begin
      if (!writingHBS && !writingLBS) ByteEnable <= 2'b01;
      if (DoneWriting && writingLBS) ByteEnable <= 2'b10;
      if (DoneWriting && writingHBS)begin ByteEnable <= 2'b00;
DoneWritingData <= 1; end
      else DoneWritingData <= 0;
      end
      if (RamRead && !WritingData) ReadingData <= 1;
      else ReadingData <= 0;
      if (DoneWriting && writingHBS) DoneWritingData <= 1;
      else DoneWritingData <= 0;
      end

// Data Write & Read
always_comb
begin
RamByteData = 0;
RamByteAddress = 0;
WritingData = 0;
RamReadEnable = 0;
RamWriteEnable = 0;
writingLBS = 0;
writingHBS = 0;
readingHBS = 0;
readingLBS = 0;
if (RamWrite && !ReadingData)
      begin
            RamByteAddress = RamWriteAddress ;
               WritingData = 1;
               RamWriteEnable = 1;
            case(ByteEnable)
            2: begin
               RamByteData = RamWriteData[31:16];
               writingHBS = 1;
               end
            1: begin
               RamByteData = RamWriteData [15:0];
               writingLBS = 1;
               end
      default: begin
               RamByteAddress = 0;
               RamByteData = 0;
               WritingData = 0;
               RamWriteEnable = 0;
               end
         endcase
      end
     if (RamRead && !WritingData)
      begin
               RamByteAddress = RamReadAddress;
               RamReadEnable = 1;
            case(ByteEnable)
            2: begin
               readingHBS = 1;
               end
            1: begin
               readingLBS = 1;
               end
      default:begin
               RamByteAddress = 0;
               RamReadEnable = 0;
               end
               endcase
      end
end
```

```
// Data Read Cumulative Register

always_ff @(posedge clk, negedge nReset)

if (!nReset)

begin

DataCumulativeRegister <= 0;

DoneReadingData <= 0;

end

else

begin

if (DoneReading && readingLBS) DataCumulativeRegister[15:0] <= RamData;

else if (DoneReading && readingHBS) begin DataCumulativeRegister[31:16] <=
RamData; DoneReadingData <= 1; end

else DoneReadingData <= 0;

end


endmodule
```

5 SRAM Controller

```
//
// File Name: SRAMController.sv
// Function: Interface with fpga sram
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module SRAMController (
// SRAM INTERFACS
output logic [19:0] addr,
inout  wire [15:0] data,
output logic WE,CS,OE,LBS,HBS,DoneReading,DoneWriting,
// PROCSSSOR Inputs and Outputs
input logic RamReadEnable,RamWriteEnable,clk,nReset,
input logic [1:0] ByteEnable,
input logic [31:0] RamByteAddress,
input logic [15:0] RamByteData,
output logic [15:0] RamData
);
logic hold,setup;
always_ff @(posedge clk, negedge nReset)
if (!nReset)
begin
CS <= 1;
WE <= 1;
OE <= 1;
LBS <= 1;
HBS <= 1;
DoneReading <= 0;
DoneWriting <= 0;
setup <= 0;
hold <= 0;
end
else
begin
```

```
addr <= RamByteAddress;
RamData <= data;
if ((RamReadEnable || RamWriteEnable) && !(setup || hold)) setup <= 1;
// read logic
if (setup && RamReadEnable)
begin
OE <= 0;
CS <= 0;
hold <= 1;
setup <= 0;
LBS <= 1'b0;
HBS <= 1'b0;
end
if (hold && RamReadEnable)
begin
OE <= 1;
CS <= 1;
LBS <= 1'b1;
HBS <= 1'b1;
hold <= 0;
DoneReading <= 1;
end
else DoneReading <= 0;
// write logic
if (setup && RamWriteEnable)
begin
CS <= 0;
WE <= 0;
hold <= 1;
setup <= 0;
LBS <= 1'b0;
HBS <= 1'b0;
end
if (hold && RamWriteEnable)
```

```
begin
CS <= 1;
WE <= 1;
hold <= 0;
DoneWriting <= 1;
LBS <= 1'b1;
HBS <= 1'b1;
end
else DoneWriting <= 0;
end
assign data = (RamWriteEnable) ? RamByteData : 16'hzzzz;
endmodule
```

6 Sram Memory

```
//

// File Name: asynchronous_ram.sv

// Function: module acts like sram

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module asynchronous_ram #(parameter n = 16,datasize = 262144)

(input logic CS,WE,OE,LBS,HBS,

input logic [19:0] addr,

inout [15:0] data

);

logic [n-1:0] RAM [0:datasize-1];

logic [15:0] dataout;

always_latch

begin

if (!CS && !WE && !LBS) RAM[addr] <= data;

else if (!CS && !WE && !HBS) RAM[addr + 1'b1] <= data;

else if (!OE && !CS && !LBS) dataout <= RAM[addr];

else if (!OE && !CS && !HBS) dataout <= RAM[addr + 1'b1];

end

assign data = (!OE) ? dataout : 16'hzzzz;

endmodule
```

7 Instruction ROM

```
//

// File Name: InstROM.sv

// Function: ROM of instructions

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module InstROM(

 input logic [31:0] InstByteAddress,

 input logic InstRead,clk,

 output logic [7:0] InstOut


);


logic [7:0] InstRom [ 0:1023];

initial

$readmemh("D:/Desktop/fourthreadscompiler/simulation
files/inst.sv",InstRom);

always_ff @(posedge clk)

if (InstRead) InstOut <= InstRom [InstByteAddress];




endmodule
```

8 Instruction Fetch

```
//

// File Name: IF.sv

// Function: instruction fetch module

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module IF(

input logic clk,nReset,flush,Ignore,

input logic [1:0] nextthread,mhartID,

output logic [31:0] I_ID,newpc_ID,currentPC_ID,

input logic InstSource,branch_ID,jump_ID,

//Ram

input logic [31:0]newPC,

input logic [31:0] ThreadAddress,

input logic InstReady,ReadCacheStatus,

input logic [31:0] InstOut,

output logic [31:0] InstAddress,FetchingAddress,

output logic InstRead,DoneRetrieving,

output logic InstMiss,InstHit,IgnoreMiss,

output logic [1:0] RetrievingDoneFor,FetchingmhartID,mhartID_ID

);

logic [31:0] I,Instruction,newPCtemp;

logic initialising;

InstMem IM1 (

.nReset,.clk,.InstReady,.Enable(InstSource),.ReadCacheStatus,.branch_ID,

.Address(ThreadAddress),.InstfromRam(InstOut),.FetchingAddress,.jump_ID,.fl
ush,

.InstMiss,.InstHit,.InstRead,.initialising,.Ignore,.FetchingmhartID,.Ignore
Miss,

.I,.InstAddress,.mhartID_ID(nextthread),.RetrievingDoneFor,.DoneRetrieving

);

assign I_ID = (InstMiss) ? 0 : I;

assign mhartID_ID = FetchingmhartID;

always_ff @(posedge clk, negedge nReset)
```

```
begin
   if (!nReset)
   begin
   newpc_ID <= 0;
   newPCtemp <= 0;
   currentPC_ID <= 0;
   initialising <= 0;
   end
      else
      begin
     newpc_ID <=  newPC;
     currentPC_ID <= ThreadAddress;
      initialising <= 1;
  end
end
endmodule
```

9 Instruction Memory

```
//
// File Name: InstMem.sv
// Function: instruction cache top level
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module InstMem(
input logic nReset,clk,InstReady,Ignore,
input logic Enable,initialising,ReadCacheStatus,branch_ID,jump_ID,flush,
input logic [1:0] mhartID_ID,
input logic [31:0] Address,InstfromRam,
output logic InstMiss,InstHit,InstRead,DoneRetrieving,IgnoreMiss,
output logic [31:0] I,InstAddress,
output logic [31:0] FetchingAddress,
output logic [1:0]  FetchingmhartID,RetrievingDoneFor
);
logic WriteInst,WriteTag,WriteValid,Match,Valid;
logic [6:0] CacheIndexRead,CacheIndexWrite,TagCompare;
logic [31:0] InstData;
logic [31:0] Inst;
logic [6:0] TagAddress,WriteAddressTag;


 always_ff @(posedge clk)
 begin
 TagAddress <= Address [15:9];
 end
 always_comb
 CacheIndexRead = Address [8:2];

 always_ff @(posedge clk, negedge nReset)
 begin
 if (!nReset)
 begin
 FetchingAddress <= 0;
 FetchingmhartID <= 0;
 end
 else
 begin
 FetchingAddress <= Address;
 FetchingmhartID <= mhartID_ID;
 end
 end

assign Match = ((TagCompare == TagAddress)&& Valid);

InstCache IC1 (
.clk,.WriteInst,.Enable,.nReset,
.CacheIndexRead,.CacheIndexWrite,
.InstData,
.Inst
);
InstValid IV1
(
.clk,.WriteValid,.nReset,.Enable,//.nReset,
.CacheIndexWrite,.CacheIndexRead,
.Valid
);
InstTag IT1
(
.WriteTag,.clk,.Enable,.nReset,
.WriteAddressTag,
.CacheIndexWrite,.CacheIndexRead,
.TagCompare
);
InstCacheControl ICC1 (
.InstReady,
.clk,

.Match,

.Ignore,

.nReset,

.ReadCacheStatus,

.branch_ID,

.jump_ID,

.Inst,

.InstfromRam,

.Address(FetchingAddress),

.I,

.InstMiss,

.InstHit,

.WriteInst,

.WriteTag,

.WriteValid,

.InstRead,//Ram

.WriteAddressTag,

.CacheIndexWrite,

.InstAddress,//Ram

.InstData,

.flush,

.mhartID_ID(FetchingmhartID),

.RetrievingDoneFor,

.DoneRetrieving,.initialising,

.IgnoreMiss

);


Endmodule
```

10 Instruction Cache

```
//

// File Name: InstCache.sv

// Function: holds instruction data

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module InstCache #(parameter SIZE = 128)

(

input logic  clk,WriteInst,Enable,nReset,

input logic  [6:0] CacheIndexRead,CacheIndexWrite,

input logic  [31:0] InstData,

output logic [31:0] Inst

);


logic [31:0] InstMem [0: SIZE - 1];

always_ff @(posedge clk)

begin

if(WriteInst) InstMem[CacheIndexWrite] <= InstData;


if (Enable) Inst <= InstMem[CacheIndexRead];

end

endmodule
```

11 Instruction Valid

```
//
// File Name: InstValid.sv
// Function: instruction valid memory
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module InstValid #(parameter SIZE = 128)
(
input logic clk,WriteValid,nReset,Enable,
input logic [6:0] CacheIndexWrite,CacheIndexRead,
output logic Valid
);

logic  Validmemory [0: SIZE - 1];

always_ff @(posedge clk, negedge nReset)
begin
if (!nReset)
begin
for (int i = 0; i <SIZE; i++)
Validmemory[i] <= 0;
end
else
begin
if (WriteValid)
Validmemory [CacheIndexWrite] <= 1'b1;
//if (Enable)
Valid <= Validmemory[CacheIndexRead];
//end
end
end
endmodule
```

12 Instruction Tag

```
//
// File Name: InstTag.sv
// Function: instruction tag memory
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module InstTag #(parameter SIZE = 128)
(
input logic WriteTag,clk,Enable,nReset,
input logic [6:0] WriteAddressTag,
input logic [6:0] CacheIndexWrite,CacheIndexRead,
output logic[6:0] TagCompare
);

logic [6:0] TagInst [0:SIZE - 1];
always_ff @(posedge clk)
begin
if (WriteTag)
TagInst[CacheIndexWrite] <= WriteAddressTag;

if (Enable) TagCompare <= TagInst[CacheIndexRead];
end
endmodule
```

13 Instruction cache control

```
//

// File Name: InstCacheControl.sv

// Function: instruction cache control

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module InstCacheControl(

input logic InstReady,

input logic clk,

input logic Match,

input logic Ignore,

input logic nReset,

input logic flush,

input logic ReadCacheStatus,

input logic branch_ID,

input logic jump_ID,

input logic [1:0] mhartID_ID,

input logic initialising,

input logic [31:0] Inst,

input logic [31:0] InstfromRam,

input logic [31:0] Address,

output logic [31:0] I,

output logic IgnoreMiss,

output logic InstMiss,

output logic InstHit,

output logic WriteInst,

output logic WriteTag,

output logic WriteValid,

output logic DoneRetrieving,

output logic InstRead,//Ram

output logic [6:0] WriteAddressTag,

output logic [6:0] CacheIndexWrite,

output logic [31:0] InstAddress,//Ram
output logic [31:0] InstData,//Ram
output logic [1:0] RetrievingDoneFor
);
logic [1:0] RetrievingFor;
logic [3:0] PendingRequests;
logic [31:0] PendingRequestsAddress [3:0];
logic [31:0] LastRetrievedAddress;
logic [1:0] LastPendingThread;
logic RetrieveRequest;
logic JustFinishRetrieving;
logic nReady;
// for
// Checking for miss or hit
always_comb
begin
InstHit = 0;
InstMiss = 0;
RetrieveRequest = 0;
I = 0;
IgnoreMiss = 0;
            if (InstReady && Address == InstAddress) I = InstfromRam;
            //if (flush) I = 0;
           if (ReadCacheStatus)
             begin
            if (Match)
                  begin
                  InstHit = 1;
                  I = Inst;
                  end
            else if (!Ignore /*&& !jump_ID && !branch_ID*/)
                  begin
                  if ((JustFinishRetrieving && Address ==
LastRetrievedAddress) || (InstReady && Address == InstAddress) ) I =
InstfromRam;
                  else
                  begin
                  InstMiss = 1;
                  if (Address == InstAddress && InstRead) IgnoreMiss = 1;
                  else
                  if (!nReady)  RetrieveRequest = 1;
                  end
                  end
            end
end
// Retrieve Request from Ram
always_ff @(posedge clk, negedge nReset)
if (!nReset)
      begin
      InstAddress <= 1;
      InstRead <= 0;
      nReady <= 0;
      PendingRequests <= 4'b0000;
      JustFinishRetrieving <= 0;
      LastRetrievedAddress <= 1;
      end
else
      begin
      if (InstRead == 0 && PendingRequests != 0)
      begin
                        if (PendingRequests [LastPendingThread + 1'b1] ==
1)
                        begin
                        InstRead <= 1;
                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 1'b1];
                        nReady <= 1;
                        RetrievingFor <= LastPendingThread + 1'b1;
                        end
                        else if (PendingRequests [LastPendingThread +
2'b10] == 1)
                        begin
                        InstRead <= 1;
                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 2'b10];
                        nReady <= 1;
                        RetrievingFor <= LastPendingThread + 2'b10;
                        end
                        else if (PendingRequests [LastPendingThread +
2'b11] == 1)
                        begin
                        InstRead <= 1;
                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 2'b11];
                        nReady <= 1;
                        RetrievingFor <= LastPendingThread + 2'b11;
                        end
      end
      if (InstReady)
      begin
      PendingRequests [RetrievingFor] <= 0;
      JustFinishRetrieving <= 1;
      LastRetrievedAddress <= InstAddress;
      end
      else
      begin
      JustFinishRetrieving <= 0;
      LastRetrievedAddress <= 1;
      end
      if (InstMiss && nReady)
            begin
           if (Address != InstAddress)
             begin
            PendingRequests [mhartID_ID] <= 1;
            PendingRequestsAddress [mhartID_ID] <= Address;
            end
            end
            if (RetrieveRequest)
            begin

            InstAddress <= Address;

            InstRead <= 1;

            nReady <= 1;

            RetrievingFor <= mhartID_ID;

            end

            else if ((InstReady )&& (PendingRequests[LastPendingThread +
1'b1] != 0 || PendingRequests[LastPendingThread + 2'b10] != 0 ||
PendingRequests[LastPendingThread + 2'b11] != 0 ))

                        begin

                        if (PendingRequests [LastPendingThread + 1'b1] ==
1)

                        begin

                        InstRead <= 1;

                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 1'b1];

                        nReady <= 1;

                        RetrievingFor <= LastPendingThread + 1'b1;

                        end

                        else if (PendingRequests [LastPendingThread +
2'b10] == 1)

                        begin

                        InstRead <= 1;

                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 2'b10];

                        nReady <= 1;

                        RetrievingFor <= LastPendingThread + 2'b10;

                        end

                        else if (PendingRequests [LastPendingThread +
2'b11] == 1)

                        begin

                        InstRead <= 1;

                        InstAddress <= PendingRequestsAddress
[LastPendingThread + 2'b11];

                        nReady <= 1;

                        RetrievingFor <= LastPendingThread + 2'b11;

                        end

                  end
```

```
      else if (InstReady && RetrieveRequest)
            begin
            InstAddress <= Address;
            InstRead <= 1;
            nReady <= 1;
            RetrievingFor <= mhartID_ID;
            end
      else if (InstReady)
      begin
      nReady <= 0;
      InstRead <= 0;
      end
      end
// Writing Back to Cache
always_comb
begin
WriteTag = 0;
WriteInst = 0;
WriteValid = 0;
InstData = 0;
WriteAddressTag = 0;
CacheIndexWrite = 0;
DoneRetrieving = 0;
RetrievingDoneFor = 0;
LastPendingThread = RetrievingFor;
if (InstReady)
      begin
      WriteTag = 1;
      WriteInst = 1;
      WriteValid = 1;
      DoneRetrieving = 1;
      RetrievingDoneFor = RetrievingFor;
      InstData = InstfromRam;
      WriteAddressTag = InstAddress[15:9];
```

```
		CacheIndexWrite = InstAddress[8:2];
		end
	end
	endmodule
```

14 Instruction Decode

```
//
// File Name: ID.sv
// Function: instruction decoder
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module ID(
	input logic [31:0] I_ID,newpc_ID,Wdata_Mem,Wdata_WB,currentPC_ID,out,IntmhartID,
	input logic clk,RegWrite_WB,nReset,flush,
	input logic [1:0] ForwardRS1,ForwardRS2,mhartID_ID,mhartID_WB,
	input logic [4:0] Waddr_WB,
	output logic [31:0] Rdata1_EX,Rdata2_EX,currentPC_EX,
	output logic RegWrite_EX,ALUSrc_EX,MemWrite_EX,MemRead_EX,MemtoReg_EX,branch_ID,jump_ID,
	store_address_EX/*,flush*/,branch,aui_EX,MemRead,MemWrite,
	output logic [1:0] mhartID_EX,
	output logic [4:0] ALUOpin_EX,
	output logic [31:0] I_EX,
	output logic [31:0]newpc_EX,imm_EX,RBranch_EX,absolute_jump_ID
);
logic [31:0] Rdata1,Rdata2,imm,ForwardS1Mux,ForwardS2Mux;
logic [4:0] ALUOpin,addr1;
logic [2:0] immSel;
logic RegWrite,ALUSrc,MemtoReg,RegDst,store_address,jump_register,rd0,aui;


	regs gpr1(.clk,.RegWrite(RegWrite_WB),.Wdata(Wdata_WB),.Raddr1(addr1),.Raddr2(I_ID[24:20]),
	.Waddr(Waddr_WB),.Rdata1,.Rdata2,.mhartID_ID,.mhartID_WB);


	Control_pipeline control1(.opcode(I_ID[6:0]),.ALUOpin(ALUOpin),.RegWrite(RegWrite),.ALUSrc(ALUSrc),

.MemWrite(MemWrite),.MemRead(MemRead),.MemtoReg(MemtoReg),.jump(jump_ID),
.store_address(store_address),.immSel,.func3(I_ID[14:12]),.func7(I_ID[31:25
]),.jump_register,.branch,.rd0,.aui);


      branch br1
(.rs1(ForwardS1Mux),.rs2(ForwardS2Mux),.opcode(I_ID[6:0]),.func3(I_ID[14:12
]),.branch(branch_ID));


      immediate_generator Imm1 (.immSel,.I(I_ID[31:7]),.imm,.IntmhartID);

      assign addr1 = (rd0) ? 5'b0:I_ID[19:15];

      //assign flush = (jump_ID || branch_ID || MemRead || MemWrite) ? 1'b1
: 1'b0;

     assign RBranch_EX = currentPC_ID + imm ;

      assign absolute_jump_ID = (jump_register) ? Rdata1 : imm +
currentPC_ID;// for jalr, usually whats before is lui or auipc, which loads
upper 20 bits

      // of target address in register and yu add lower 12, it is used
becasuse you cannot store 32 bits of address in a signle operation


      always_comb

      begin

      ForwardS1Mux = 0;

      ForwardS2Mux = 0;

      case (ForwardRS1)

      0: ForwardS1Mux = Rdata1;

      2: ForwardS1Mux = Wdata_Mem;

      3: ForwardS1Mux = out;

      default: ForwardS1Mux = Rdata1;

      endcase

      case (ForwardRS2)

      0: ForwardS2Mux = Rdata2;

      2: ForwardS2Mux = Wdata_Mem;

      3: ForwardS2Mux = out;

      default: ForwardS2Mux = Rdata2;

      endcase

      end


      always_ff @(posedge clk, negedge nReset)
```

```
if (!nReset)
begin
   Rdata1_EX <= 0;
   Rdata2_EX <= 0;
   RegWrite_EX <= 0;
   ALUSrc_EX <= 0;
   MemWrite_EX <= 0;
   MemRead_EX <= 0;
   MemtoReg_EX <= 0;
   store_address_EX <= 0;
   I_EX <= 0;
ALUOpin_EX <= 0;
   newpc_EX <= 0;
   imm_EX <= 0;
   currentPC_EX <= 0;
   aui_EX <= 0;
   mhartID_EX <= 0;
end
 else if (flush)
begin
   Rdata1_EX <= 0;
   Rdata2_EX <= 0;
   RegWrite_EX <= 0;
   ALUSrc_EX <= 0;
   MemWrite_EX <= 0;
   MemRead_EX <= 0;
   MemtoReg_EX <= 0;
   store_address_EX <= 0;
   I_EX <= 0;
ALUOpin_EX <= 0;
   newpc_EX <= 0;
   imm_EX <= 0;
   aui_EX <= 0;
end
```

```
else
begin
  currentPC_EX <= currentPC_ID;
   Rdata1_EX <= Rdata1;
   Rdata2_EX <= Rdata2;
   RegWrite_EX <= RegWrite;
   ALUSrc_EX <= ALUSrc;
   MemWrite_EX <= MemWrite;
   MemRead_EX <= MemRead;
   MemtoReg_EX <= MemtoReg;
   store_address_EX <= store_address;
   I_EX <= I_ID [31:0];
   ALUOpin_EX <= ALUOpin;
   newpc_EX <=  newpc_ID;
   imm_EX <= imm;
   aui_EX <= aui;
   mhartID_EX <= mhartID_ID;
end
endmodule
```

15 Register file

```
//
// File Name: regs.sv
// Function: register file
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module regs #(parameter n = 32) // n - data bus width
(input logic clk, RegWrite,
input logic [1:0] mhartID_WB,
 input logic [1:0] mhartID_ID, // clk and write control
 input logic [n-1:0] Wdata,
 input logic [4:0] Raddr1, Raddr2,Waddr,
 output logic [n-1:0] Rdata1, Rdata2);

      // Declare 32 n-bit registers
      logic [n-1:0] gpr1 [n-1:0];
      logic [n-1:0] gpr2 [n-1:0];
      logic [n-1:0] gpr3 [n-1:0];
      logic [n-1:0] gpr4 [n-1:0];



      // write process, dest reg is waddr
      always_ff @ (posedge clk)
      begin
            if (RegWrite)
            begin
            case (mhartID_WB)
          0: gpr1[Waddr] <= Wdata;
          1: gpr2[Waddr] <= Wdata;
            2: gpr3[Waddr] <= Wdata;
            3: gpr4[Waddr] <= Wdata;
            endcase
            end
```

```
        end


      always_comb

      begin

         if (Raddr1==5'd0) Rdata1 =  {n{1'b0}};

         else if (Raddr1 == Waddr && mhartID_ID == mhartID_WB && RegWrite)
Rdata1 = Wdata;

        else

        begin

         case(mhartID_ID)

         0: Rdata1 = gpr1[Raddr1];

         1: Rdata1 = gpr2[Raddr1];

         2: Rdata1 = gpr3[Raddr1];

         3: Rdata1 = gpr4[Raddr1];

         endcase

         end

        if (Raddr2==5'd0) Rdata2 =  {n{1'b0}};

        else if (Raddr2 == Waddr && mhartID_ID == mhartID_WB && RegWrite)
Rdata2 = Wdata;

        else

        begin

        case (mhartID_ID)

         0: Rdata2 = gpr1[Raddr2];

         1: Rdata2 = gpr2[Raddr2];

         2: Rdata2 = gpr3[Raddr2];

         3: Rdata2 = gpr4[Raddr2];

        endcase

        end

      end

endmodule // module regs
```

16 instruction decoder

```
//
// File Name: Control_pipeline.sv
// Function: control unit
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
`include "opcodes.sv"
`include "alucodes.sv"
module Control_pipeline(
input logic [6:0] opcode,
input logic [2:0] func3,
input logic [6:0] func7,
output logic [4:0] ALUOpin,
output logic [2:0] immSel,
output logic RegWrite,
output logic ALUSrc,
output logic MemWrite,
output logic jump,
output logic MemRead,
output logic MemtoReg,
output logic store_address,
output logic jump_register,branch,rd0,aui
);
 always_comb
 begin

 ALUSrc = 1;
 RegWrite = 0;
 MemWrite = 0;
 MemRead = 0;
 MemtoReg = 0;
 store_address = 0;
```

```
jump = 0;
ALUOpin = `alu_add;
immSel = 0;
jump_register = 0;
branch = 0;
rd0 = 0;
aui = 0;
 case  (opcode)
  `CSRRS :begin
             immSel = 3'b101;
            RegWrite = 1;
            end
  `LUI : begin
        RegWrite = 1;
              ALUOpin = `alu_add;
              immSel = 0; // case imm {{13{I[31]}}, I[30:12]} case 1
                rd0 = 1;
              end

 `AUIPC: begin
              ALUOpin = `alu_add;
                 immSel = 0; // case imm {{13{I[31]}}, I[30:12]} case 1
                 RegWrite = 1;
                 aui = 1;
                 end

 `JAL  : begin
        jump = 1;
                 store_address = 1;
                 RegWrite = 1;
                 immSel = 3'b001; // case imm
{{12{I[31]}},I[19:12],I[20],I[30:21],1'b0} case 2
                 end

 `JALR : begin
```

```
         jump = 1;
                  store_address = 1;
                  RegWrite = 1;
                  immSel = 3'b010;// case  imm {20{I[11]},I[11:0]} case 3
                  jump_register = 1;
                  end

    `SW : begin
                  MemWrite = 1;
                  ALUOpin = `alu_add;
                  immSel = 3'b100;// case  imm {{21{I[31]}},
I[30:25],I[11:7]} case 5
                  end

  `Btype : begin
                  immSel = 3'b011;// case  imm
{20{I[12]},I[30:25],I[11:7],1'b0} case 4
                  branch = 1;
                  end
  `LW    : begin
         ALUOpin = `alu_add;
                  RegWrite = 1;
                  MemRead = 1;
                  MemtoReg = 1;
                  immSel = 3'b010;// case  imm {20{I[11]},I[11:0]}
                  end

  `Rtype : begin
                  ALUSrc = 0;
               RegWrite = 1;
                 if (func7[0])
                   begin
                   case (func3)
                   0: ALUOpin = `alu_mul;
                   1: ALUOpin = `alu_mulh;
```

```
                                 2: ALUOpin = `alu_mulhu;
                                 4: ALUOpin = `alu_div;
                                 5: ALUOpin = `alu_divu;
                                 6: ALUOpin = `alu_rem;
                                 7: ALUOpin = `alu_remu;
                                 default: ;
                                 endcase
                                 end
                                 else
                                 begin
                                case (func3)
                                0 : begin
                                    if (func7 == 0) ALUOpin = `alu_add;
                                    else ALUOpin = `alu_sub;
                                       end
                                1 : ALUOpin = `alu_sll;
                                2 : ALUOpin = `alu_slt;
                                3 : ALUOpin = `alu_sltu;
                                4 : ALUOpin = `alu_xor;
                                5 : begin
                                    if (func7 == 0) ALUOpin = `alu_srl;
                                    else ALUOpin = `alu_sra;
                                       end
                                6 : ALUOpin = `alu_or;
                                7 : ALUOpin = `alu_and;
                                endcase // func 3
                                end
                                end
`Itype : begin
                                RegWrite = 1;
                                immSel = 3'b010;// case  imm {20{I[11]},I[11:0]}
                                case (func3)
                                0: ALUOpin = `alu_add;
                                1: ALUOpin = `alu_slli;
```

```
                    2 : ALUOpin = `alu_slt;
                    3 : ALUOpin = `alu_sltu;
                    4 : ALUOpin = `alu_xor;
                    5:  ALUOpin = `alu_srai;
                    6 : ALUOpin = `alu_or;
                    7 : ALUOpin = `alu_and;
                    default : ;
                    endcase
                    end
default : ;
endcase
end//always_comb
 endmodule
```

17 Branch comparator

```
//

// File Name: branch.sv

// Function: branch comparator

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

`include "opcodes.sv"

module branch(

input logic [6:0] opcode,

input logic [2:0] func3,

input logic signed [31:0] rs1,rs2,

output logic branch

);

always_comb

begin

branch = 0;

if (opcode == `Btype)

begin

case (func3)

0 : if (rs1 == rs2)   branch = 1;

1 : if (rs1 != rs2)   branch = 1;

4 : if (rs1 < rs2)    branch = 1;

5 : if ( rs1 >= rs2)  branch = 1;

default : ;

endcase

end // if

end // always_comb

endmodule
```

18 Immediate Generator

```
//
// File Name: immediate_generator.sv
// Function: immediate generator
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module immediate_generator(
input logic [2:0] immSel,
input logic [31:0] IntmhartID,
input logic [24:0] I,
output logic [31:0] imm
);
always_comb
begin
imm = 0;
case (immSel)
0 : imm = {I[24:5],{12{1'b0}}};
1 : imm = {{12{I[24]}},I[12:5],I[13],I[23:14],1'b0};
2 : imm = {{20{I[24]}},I[24:13]};
3 : imm = {{20{I[24]}},I[0],I[23:18],I[4:1],1'b0};//branch
4 : imm = {{21{I[24]}}, I[23:18],I[4:0]};
5 : imm = IntmhartID;
default :;
endcase
end
endmodule
```

19 Execute stage

```
//

// File Name: EX.sv

// Function: execute stage

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module EX(

input logic [31:0] Rdata1_EX,Rdata2_EX,Wdata_WB,imm_EX,currentPC_EX,

input logic
RegWrite_EX,ALUSrc_EX,MemWrite_EX,MemRead_EX,MemtoReg_EX,store_address_EX,c
lk,nReset,aui_EX,flush,

input logic [1:0] mhartID_EX,

input logic [4:0] ALUOpin_EX,

input logic [31:0] newpc_EX,

input logic [31:0] I_EX,

output logic [31:0]
Wdata_Mem,Rdata2_Mem,out,MemAddress,currentPC_Mem,T0out,T1out,T2out,T3out,T
0in,T1in,T2in,T3in,

output logic [4:0] Waddr_Mem,

output logic RegWrite_Mem,MemWrite_Mem,MemRead_Mem,MemtoReg_Mem,ReadEnable,

output logic [1:0] mhartID_Mem,

input logic [1:0] ForwardA,ForwardB,

// external data memory

output logic ReadAccess,LED0,LED1,LED2,LED3,

output logic [7:0] address

);

  logic [31:0]
ALUBMUX,ALUAMUX,storeregister;//,store_register,absolute_jump_Mem1;

  logic [4:0] DstMux,ALUOp;

  logic [31:0] ForwardAMux,ForwardBMux;


  alu
alu1(.a(ALUAMUX),.b(ALUBMUX),.out,.ALUOp(ALUOpin_EX),.clk,.sa(I_EX[24:20]))
;


  assign ALUAMUX    = (aui_EX)       ? currentPC_EX : ForwardAMux;
```

```
 assign ALUBMUX    = (ALUSrc_EX)   ?  imm_EX: ForwardBMux;


 assign memorymux = out[20];
 assign storeregister = (store_address_EX) ? newpc_EX : out;
 assign ReadEnable = (MemRead_EX & !memorymux ) ? 1'b1 : 1'b0;
 assign MemAddress = storeregister;
 assign MemWrite_Mem = MemWrite_EX;
 assign ReadAccess = (MemRead_EX & memorymux) ? 1'b1 : 1'b0;
 assign address = {storeregister[9:2]};
 always_ff @(posedge clk, negedge nReset)
 if (!nReset)
 begin
 RegWrite_Mem <= 0;
 MemtoReg_Mem <= 0;
 Waddr_Mem <= 0;
 Wdata_Mem <= 0;
 MemRead_Mem <= 0;
/* LED0 <= 0;
 LED1 <= 0;
 LED2 <= 0;
 LED3 <= 0;
 T0out <= 0;
 T1out <= 0;
 T2out <= 0;
 T3out <= 0;
 T0in <= 0;
 T1in <= 0;
 T2in <= 0;
 T3in <= 0;*/
 mhartID_Mem <= 0;
 currentPC_Mem <= 0;
 end
 else if (flush)
 begin
```

```
  RegWrite_Mem <= 0;
  MemtoReg_Mem <= 0;
  Waddr_Mem <= 0;
  Wdata_Mem <= 0;
  MemRead_Mem <= 0;
 // MemWrite_Mem = 0;
  //Rdata2_Mem <= 0;
  end
  else
  begin
  RegWrite_Mem <= RegWrite_EX;
  MemtoReg_Mem <= MemtoReg_EX;
  Waddr_Mem <= I_EX[11:7];
  Wdata_Mem <= storeregister;
  mhartID_Mem <= mhartID_EX;
  currentPC_Mem <= currentPC_EX;
  MemRead_Mem <= MemRead_EX;
  if (MemWrite_EX && out == 32'h00) begin T0out <= Rdata2_Mem; LED0 <= 1;
end else LED0 <= 0;
  if (MemWrite_EX && out == 32'h02) T0in <= Rdata2_Mem;
  if (MemWrite_EX && out == 32'h04) begin T1out <= Rdata2_Mem; LED1 <= 1;
end else LED1 <= 0;
  if (MemWrite_EX && out == 32'h06) T1in <= Rdata2_Mem;
  if (MemWrite_EX && out == 32'h08) begin T2out <= Rdata2_Mem; LED2 <= 1;
end else LED2 <= 0;
  if (MemWrite_EX && out == 32'h0A) T2in <= Rdata2_Mem;
  if (MemWrite_EX && out == 32'h0C) begin T3out <= Rdata2_Mem; LED3 <= 1;
end else LED3 <= 0;
  if (MemWrite_EX && out == 32'h0E) T3in <= Rdata2_Mem;
  end
  // forwarding mux
 always_comb
 begin
 case (ForwardA)
 0: ForwardAMux = Rdata1_EX;
 1: ForwardAMux =  Wdata_WB;
 2: ForwardAMux = Wdata_Mem;
 default: ForwardAMux = Rdata1_EX;
 endcase
 case (ForwardB)
 0: ForwardBMux = Rdata2_EX;
 1: ForwardBMux =  Wdata_WB;
 2: ForwardBMux = Wdata_Mem;
 default: ForwardBMux = Rdata2_EX;
 endcase
 case (ForwardB)
 0: Rdata2_Mem = Rdata2_EX;
 1: Rdata2_Mem =  Wdata_WB;
 2: Rdata2_Mem = Wdata_Mem;
 default: Rdata2_Mem = Rdata2_EX;
 endcase
 end
// branch execution
Endmodule
```

20 ALU

```
//

// File Name: alu.sv

// Function: ALU

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

`include "alucodes.sv"

module alu #(parameter n = 32)

(input logic [n-1:0] a,b,

input logic clk,

output logic [n-1:0] out,

input logic [4:0] ALUOp,

input logic [4:0] sa

);

logic [n-1:0]
signed_add,signed_sub,unsigned_sub,unsigned_quotient,unsigned_remainder,sig
ned_quotient,signed_remainder;

logic [63:0] signed_mul, unsigned_mul;


signed_add add1(.a(a),.b(b),.signed_add(signed_add));



signed_sub sub1(.a(a),.b(b),.signed_sub(signed_sub));


unsigned_sub sub2(.a(a),.b(b),.unsigned_sub(unsigned_sub));


signed_mult mul1(.a(a),.b(b),.signed_mul(signed_mul));


unsigned_mult mul2(.a(a),.b(b),.unsigned_mul(unsigned_mul));


signed_divider
div1(.a(a),.b(b),.signed_quotient(signed_quotient),.signed_remainder(signed
_remainder));
```

```
unsigned_divider
div2(.a(a),.b(b),.unsigned_quotient(unsigned_quotient),.unsigned_remainder(
unsigned_remainder));


always_comb

begin

out = 0;

case (ALUOp)

`alu_add: out = signed_add;

`alu_sub: out = signed_sub;

`alu_and: out = (a && b);

`alu_or:  out = (a || b);

`alu_xor: out = (a ^^ b);

`alu_nor: out = ~(a || b);

`alu_sll: out = a << b;

`alu_srl: out = a >> b;

`alu_sra: out = a >>> b;

`alu_slli: out = a << sa;

`alu_srli: out = a >> sa;

`alu_srai: out = a >>> sa;

`alu_slt:  out = (signed_sub[31] == 1) ? 1: 0;

`alu_sltu: out = (unsigned_sub[31] == 1) ? 1: 0;

`alu_div:  out = signed_quotient;

`alu_divu: out = unsigned_quotient;

`alu_mul:  out = signed_mul[31:0];

`alu_mulh: out = signed_mul[63:32];

`alu_mulhu: out = unsigned_mul[63:32];

`alu_rem : out = signed_remainder;

`alu_remu: out = unsigned_remainder;

default :  out = 0;

endcase// case ALUOp

end// always_comb

endmodule
```

21 Memory Stage

```
//

// File Name: Mem.sv

// Function: memory stage

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module Mem(

input logic [31:0] Wdata_Mem,Rdata2_Mem,MemAddress,datafromexternal,

input logic [4:0] Waddr_Mem,

input logic
RegWrite_Mem,MemWrite_Mem,MemRead_Mem,MemtoReg_Mem,clk,nReset,DoneReadingDa
ta,DoneWritingData,ReadEnable,ReadFromExternal,

input logic [1:0] mhartID_Mem,mhartID_EX,

output logic [31:0] Wdata_WB,

output logic [4:0] Waddr_WB,

output logic RegWrite_WB,CacheHit,CacheMiss,StoreHazard,

input logic [31:0] RamData,

output logic [31:0] RamReadAddress,RamWriteAddress,

output logic [31:0] RamWriteData,

output logic RamRead,

output logic RamWrite,

output logic [1:0] mhartID_WB,DoneForTID,DoneWritingFor

);

logic [31:0] ReadData,WriteBackData;


CacheMemory CM1 (

.clk,.nReset,.load(MemRead_Mem),.store(MemWrite_Mem),

.WritetoData(Rdata2_Mem),.Address_Mem(MemAddress),

.CacheHit,.CacheMiss,.ReadEnable,

.WriteBackData,.DoneWritingFor,

.RamData,.mhartID_EX,.StoreHazard,

.RamReadAddress,.RamWriteData,.RamWriteAddress,

.RamRead,.RamWrite,

.DoneReadingData,.DoneWritingData,.mhartID_Mem,.DoneForTID
);

assign ReadData = (ReadFromExternal) ? datafromexternal : WriteBackData;
always_ff @(posedge clk, negedge nReset)
if (!nReset)
begin
RegWrite_WB <= 0;
Wdata_WB <= 0;
Waddr_WB <= 0;
mhartID_WB <= 0;
end
else if (CacheMiss)
begin
RegWrite_WB <= 0;
Wdata_WB <= 0;
Waddr_WB <= 0;
mhartID_WB <= mhartID_Mem;
end
else
begin
RegWrite_WB <= RegWrite_Mem;
Wdata_WB <= (MemtoReg_Mem) ? ReadData: Wdata_Mem;
Waddr_WB <= Waddr_Mem;
mhartID_WB <= mhartID_Mem;
end
endmodule
```

22 Cache memory

```
//

// File Name: CacheMemory.sv

// Function: data cache top level

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module CacheMemory (

 input logic
clk,nReset,load,store,DoneReadingData,DoneWritingData,ReadEnable,

 input logic [1:0] mhartID_Mem,mhartID_EX,

 input logic [31:0] WritetoData,Address_Mem,

 output logic CacheHit,CacheMiss,StoreHazard,

 output logic [31:0] WriteBackData,

 //Ram

 input logic [31:0] RamData,

 output logic [31:0] RamReadAddress,RamWriteData,RamWriteAddress,

 output logic RamRead,RamWrite,

 output logic [1:0] DoneForTID,DoneWritingFor

);

 // Data Memory

 logic WriteCache,Writethread;

 logic [6:0] CacheIndexWrite,CacheIndexRead;

 logic [31:0] WriteDataCache;

 logic [31:0] CacheData;

 // TagMemory

 logic WriteTag;

 logic [11:0] TagAddress,WriteAddressTag;

 logic [11:0] TagCompare;

 // ValidRam

 logic WriteValid;

 logic Valid;

 //Cache Control

 logic Match;

 always_ff @(posedge clk)
```

```
TagAddress <= Address_Mem [20:9];



assign Match = ((TagCompare == TagAddress )&& Valid );
assign CacheIndexRead = Address_Mem [8:2];

DataMemory   DM1 (.clk,.WriteCache,.CacheIndexWrite,.ReadEnable,
.WriteDataCache,.CacheData,.CacheIndexRead,.nReset);

TagRAM       TR1 (.clk,.CacheIndexWrite,.WriteTag,.ReadEnable,
.TagAddress,.WriteAddressTag,.TagCompare,.CacheIndexRead,.nReset);

ValidRam     VR1 (.clk,.CacheIndexWrite,.WriteValid,.ReadEnable,
.Valid,.nReset,.CacheIndexRead);

CacheControl CC1 (.Match,.clk,.nReset,.mhartID_Mem,
.CacheData,.WritetoData,.Address_Mem,.ReadEnable,.mhartID_EX,
.load,.store,.DoneWritingData,.DoneReadingData,.RamData,
.WriteBackData,.WriteDataCache,.RamReadAddress,.RamWriteAddress,
.RamWriteData,.WriteAddressTag,.CacheIndexWrite,.StoreHazard,
.WriteValid,.WriteTag,.WriteCache,.RamRead,.RamWrite,.DoneWritingFor,
.CacheHit,.CacheMiss,.DoneForTID);
Endmodule
```

23 Data memory

```
//
// File Name: Datamemory.sv
// Function: data memory
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module DataMemory #(parameter SIZE = 128)
(
input logic clk,WriteCache,ReadEnable,nReset,
input logic [6:0] CacheIndexWrite,CacheIndexRead,
input logic [31:0] WriteDataCache,
output logic [31:0] CacheData
);
logic [31:0] CacheMem [0: SIZE - 1];
always_ff @(posedge clk)
begin
if (WriteCache)
CacheMem [CacheIndexWrite] <= WriteDataCache;

if (ReadEnable)
CacheData <= CacheMem [CacheIndexRead];
end
endmodule
```

24 Tag Memory

```
//

// File Name: TagRAM.sv

// Function: data memory tag

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module TagRAM #(parameter SIZE = 128)

(

input logic clk,WriteTag,ReadEnable,nReset,

input logic [11:0] TagAddress,WriteAddressTag,

input logic [6:0] CacheIndexWrite,CacheIndexRead,

output logic [11:0] TagCompare

);

logic [11:0] TagData [0:SIZE - 1];

always_ff @(posedge clk)

begin

if (WriteTag)

TagData[CacheIndexWrite] <= WriteAddressTag;


if (ReadEnable)

TagCompare <= TagData[CacheIndexRead];

end

endmodule
```

25 Valid Memory

```
//
// File Name: ValidRam.sv
// Function: Data cache valid
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
module ValidRam #(parameter SIZE = 128)
(
input logic clk,WriteValid,nReset,ReadEnable,
input logic [6:0] CacheIndexWrite,CacheIndexRead,
output logic Valid
);
logic  ValidData [0: SIZE - 1];
always_ff @(posedge clk, negedge nReset)
begin
if (!nReset)
begin
for (int i = 0; i < SIZE ; i++)
ValidData[i]<=0;
end
else
begin
if (WriteValid)
ValidData [CacheIndexWrite] <= 1'b1;
if (ReadEnable)
Valid <= ValidData[CacheIndexRead];
end
end
endmodule
```

26 Data Cache Control

```
//

// File Name: CacheControl.sv

// Function: Data Cache Control

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

module CacheControl(

input logic Match,clk,nReset,ReadEnable,

input logic [1:0] mhartID_Mem,mhartID_EX,

input logic [31:0] CacheData,WritetoData,Address_Mem,

input logic load,store,DoneWritingData,DoneReadingData,

input logic [31:0] RamData,

output logic [31:0]
WriteBackData,WriteDataCache,RamReadAddress,RamWriteAddress,RamWriteData,

output logic [11:0] WriteAddressTag,

output logic [6:0] CacheIndexWrite,

output logic WriteValid,WriteTag,WriteCache,RamRead,RamWrite,StoreHazard,

output logic CacheHit,CacheMiss,

output logic [1:0] DoneForTID,DoneWritingFor

);

logic [7:0] PendingStoreRequest;

logic [1:0] WritingForTID,ReadingForTID;

logic [2:0] LastStoringTID;

logic [31:0] WriteAddress,ReadAddress,WritetoDataRegister;

logic [31:0] CacheWriteAddressStore,CacheWriteDataStore;

logic [31:0] PendingStoreAddress [7:0];

logic [31:0] PendingLoadAddress [3:0];

logic [31:0] PendingStoreData [7:0];

logic [3:0]PendingReadRequest;

logic CacheHazardFlag,Reading,Writing,StoreRequest;

// load logic

enum logic {Idle,Update} state,nextstate;


 always_ff @(posedge clk, negedge nReset)
 if (!nReset)
 begin
 ReadAddress <= 0;
 PendingReadRequest <= 4'b0000;
 ReadingForTID <= 0;
 Reading <= 0;
 RamRead <= 0;
 end
 else
 begin
 // for reading cache, must happen every time load takes place
 if (ReadEnable)
 ReadAddress <= Address_Mem;
 // logic for Miss Queue
 if (CacheMiss && (Reading || (!Reading && PendingReadRequest != 0 )) &&
ReadingForTID != mhartID_Mem)
       begin
       PendingReadRequest [mhartID_Mem] <= 1;
       PendingLoadAddress [mhartID_Mem] <= ReadAddress;
       end
 if ((DoneReadingData || !Reading) && PendingReadRequest != 0)
       begin
       if (PendingReadRequest [ReadingForTID + 1'b1] == 1)
             begin
             PendingReadRequest [ReadingForTID + 1'b1] <= 0;
             RamRead <= 1;
             RamReadAddress <= PendingLoadAddress [ReadingForTID + 1'b1];
             ReadingForTID <= ReadingForTID + 1'b1;
             Reading <= 1;
             end
       else if (PendingReadRequest [ReadingForTID + 2'b10] == 1)
             begin
             PendingReadRequest [ReadingForTID + 2'b10] <= 0;
             RamRead <= 1;
             RamReadAddress <= PendingLoadAddress [ReadingForTID + 2'b10];
```

```
              ReadingForTID <= ReadingForTID + 2'b10;
              Reading <= 1;
              end
        else if (PendingReadRequest [ReadingForTID + 2'b11] == 1)
              begin
              PendingReadRequest [ReadingForTID + 2'b11] <= 0;
              RamRead <= 1;
              RamReadAddress <= PendingLoadAddress [ReadingForTID + 2'b11];
              ReadingForTID <= ReadingForTID + 2'b11;
              Reading <= 1;
              end
       end
else if (CacheMiss && !Reading)
       begin
       RamRead <= 1;
       RamReadAddress <= ReadAddress;
       ReadingForTID <= mhartID_Mem;
       Reading <= 1;
       end
else if (DoneReadingData)
       begin
       RamRead <= 0;
       RamReadAddress <= 1;
        Reading <= 0;
       end
 end

 //logic forwarded;
// load control
always_comb
begin
CacheHit = 0;
CacheMiss = 0;
WriteBackData = 0;
```

```
//forwarded = 0;

      if (load)

      begin

      if (Match)

      begin

      CacheHit = 1;

      WriteBackData = CacheData;

      end

      else if (ReadAddress == PendingStoreAddress [mhartID_Mem]) begin
WriteBackData = PendingStoreData [mhartID_Mem]; CacheHit = 1; /*forwarded =
1;*/end

      else if (ReadAddress == PendingStoreAddress [mhartID_Mem + 3'b100])
begin  WriteBackData = PendingStoreData [mhartID_Mem +  3'b100]; CacheHit =
1;/*forwarded = 1;*/ end

      else if (ReadAddress == RamWriteAddress) begin  WriteBackData =
RamWriteData; CacheHit = 1; end

      else if (Reading && ReadingForTID == mhartID_Mem) CacheMiss = 0;

      else

      CacheMiss = 1;

      end

      end

always_ff @(posedge clk, negedge nReset)

if (!nReset)state <= Idle;

else state <= nextstate;


// store logic

always_comb

begin

RamWrite = 0;

Writing = 0;

RamWriteAddress = 0;

RamWriteData = 0;

nextstate = state;

case (state)

Idle : if (store || PendingStoreRequest != 0) nextstate = Update;

Update:
```

```
begin
RamWriteAddress = WriteAddress;
RamWriteData = WritetoDataRegister;
Writing = 1;
RamWrite = 1;
if (DoneWritingData)
begin
if (PendingStoreRequest == 0 && !StoreRequest)
nextstate = Idle;
end
end
endcase
end
```

```
// Write Ram Write Queue
always_ff @(posedge clk, negedge nReset)
begin
if (!nReset)
      begin
       PendingStoreRequest <= 0;
       WritetoDataRegister <= 0;
       WriteAddress <= 0;
       LastStoringTID <= 0;
       WritingForTID <= 0;
       StoreHazard <= 0;
      end
else
begin
 if (store && Writing)
 begin
 if (PendingStoreRequest [mhartID_EX] == 1 && PendingStoreRequest
[mhartID_EX + 3'b100] == 1)
 begin
 PendingStoreRequest[mhartID_EX] <= 1;
 StoreHazard <= 1;
 end
 else
  begin
       if (PendingStoreRequest [mhartID_EX] == 0)
       begin
       PendingStoreRequest [mhartID_EX] <= 1;
       PendingStoreAddress [mhartID_EX]<= Address_Mem;
       PendingStoreData [mhartID_EX]<= WritetoData;
       end
       else if (PendingStoreRequest [mhartID_EX + 3'b100 ] == 0)
       begin
       PendingStoreRequest [mhartID_EX + 3'b100 ] <= 1;
       PendingStoreAddress [mhartID_EX + 3'b100 ]<= Address_Mem;
       PendingStoreData [mhartID_EX + 3'b100 ]<= WritetoData;
       end
       StoreHazard <= 0;
       end
 end
 else StoreHazard <= 0;
 if ((DoneWritingData || !Writing) && PendingStoreRequest != 0)
 begin
        if (PendingStoreRequest [LastStoringTID + 1'b1])
              begin
              PendingStoreRequest [LastStoringTID + 1'b1] <= 0;
              PendingStoreAddress [LastStoringTID + 1'b1] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID + 1'b1];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
1'b1];
              LastStoringTID <= LastStoringTID + 1'b1;
              WritingForTID <= LastStoringTID + 1'b1;
              end
        else if (PendingStoreRequest [LastStoringTID + 2'b10])
              begin
              PendingStoreRequest [LastStoringTID + 2'b10] <= 0;
              PendingStoreAddress [LastStoringTID + 2'b10] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID + 2'b10];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
2'b10];
              LastStoringTID <= LastStoringTID + 2'b10;
              WritingForTID <= LastStoringTID + 2'b10;
              end
        else if (PendingStoreRequest [LastStoringTID + 2'b11])
        begin
              PendingStoreRequest [LastStoringTID + 2'b11] <= 0;
              PendingStoreAddress [LastStoringTID + 2'b11] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID + 2'b11];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
2'b11];
              LastStoringTID <= LastStoringTID + 2'b11;
              WritingForTID <= LastStoringTID + 2'b11;
              end// new queue
        else if (PendingStoreRequest [LastStoringTID + 3'b100])
        begin
              PendingStoreRequest [LastStoringTID + 3'b100] <= 0;
              PendingStoreAddress [LastStoringTID + 3'b100] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID +
3'b100];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
3'b100];
              LastStoringTID <= LastStoringTID + 3'b100;
              WritingForTID <= LastStoringTID + 3'b100;
              end
        else if (PendingStoreRequest [LastStoringTID + 3'b101])
        begin
              PendingStoreRequest [LastStoringTID + 3'b101] <= 0;
              PendingStoreAddress [LastStoringTID + 3'b101] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID +
3'b101];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
3'b101];
              LastStoringTID <= LastStoringTID + 3'b101;
              WritingForTID <= LastStoringTID + 3'b101;
              end
        else if (PendingStoreRequest [LastStoringTID + 3'b110])
        begin
              PendingStoreRequest [LastStoringTID + 3'b110] <= 0;
              PendingStoreAddress [LastStoringTID + 3'b110] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID +
3'b110];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
3'b110];
              LastStoringTID <= LastStoringTID + 3'b110;
              WritingForTID <= LastStoringTID + 3'b110;
              end
        else if (PendingStoreRequest [LastStoringTID + 3'b111])
        begin
              PendingStoreRequest [LastStoringTID + 3'b111] <= 0;
              PendingStoreAddress [LastStoringTID + 3'b111] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID +
3'b111];
              WritetoDataRegister <= PendingStoreData [LastStoringTID +
3'b111];
              LastStoringTID <= LastStoringTID + 3'b111;
              WritingForTID <= LastStoringTID + 3'b111;
              end
        else if (PendingStoreRequest [LastStoringTID])
              begin
              PendingStoreRequest [LastStoringTID] <= 0;
              PendingStoreAddress[LastStoringTID] <= 0;
              WriteAddress <= PendingStoreAddress [LastStoringTID ];
              WritetoDataRegister <= PendingStoreData [LastStoringTID];
              LastStoringTID <= LastStoringTID;
              WritingForTID <= LastStoringTID;
              end
 end
 else if (store && !Writing)
       begin
       WriteAddress <= Address_Mem;
       WritetoDataRegister <= WritetoData;
       WritingForTID <= mhartID_EX;
       LastStoringTID <= mhartID_EX;
       end
       if (store && !Writing )
       StoreRequest <= 1;
       else
       StoreRequest <= 0;
end
end

// Write to cache Queue
always_ff @(posedge clk, negedge nReset)
if (!nReset)
      begin
      CacheWriteAddressStore <= 0;
      CacheWriteDataStore <= 0;
      CacheHazardFlag <= 0;
      end
else
begin
if (DoneReadingData && store)
      begin
      CacheHazardFlag <= 1;
      CacheWriteAddressStore <= Address_Mem;
      CacheWriteDataStore <= WritetoData;
      end
else CacheHazardFlag <= 0;
end
```

```
// write back cache logic
always_comb
begin
WriteTag = 0;
WriteValid = 0;
WriteCache = 0;
WriteAddressTag = 0;
WriteDataCache = 0;
CacheIndexWrite = 0;
DoneForTID = 0;
DoneWritingFor = 0;
if (DoneReadingData)
begin
      WriteTag = 1;
      WriteValid = 1;
      WriteCache = 1;
      DoneForTID = ReadingForTID;
      WriteAddressTag = RamReadAddress [20:9];
      WriteDataCache = RamData;
      CacheIndexWrite = RamReadAddress [8:2];
end
if (store && !DoneReadingData)
begin
      WriteTag = 1;
      WriteValid = 1;
      WriteCache = 1;
      WriteAddressTag = Address_Mem [20:9];
      WriteDataCache = WritetoData;
      CacheIndexWrite = Address_Mem [8:2];
end
if (CacheHazardFlag)
begin
```

```
      WriteTag = 1;
      WriteValid = 1;
      WriteCache = 1;
      WriteAddressTag = CacheWriteAddressStore [20:9];
      WriteDataCache = CacheWriteDataStore;
      CacheIndexWrite = CacheWriteAddressStore [8:2];
end
if (DoneWritingData) DoneWritingFor = WritingForTID;
end
endmodule
```

27 System Testbench

```
//

// File Name: RISC_TEST.sv

// Function: Data Cache Control

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

`include "opcodes.sv"

module RISC_test #( parameter n = 32); // data bus width

 logic clk;

logic nReset;

logic [31:0] test,alu_out,a,b,SwitchCount,tempaddress;

logic
finisht1,finisht2,finisht3,finisht4,t1sw,t2sw,t3sw,t4sw,t1lw,t2lw,t3lw,t4lw
,t1miss,t2miss,t3miss,t4miss,errorT1,errorT2,errorT3,errorT4;

logic flush,branch,jump,threadIDramflag;


enum logic [5:0] {addi ,addiu ,andi,ori,xori,lhi,llo,slti,

sltiu,beq ,blt,bgte,bne,Rtype,

lui,AUIPC,lw ,sb ,sh,sw,j ,jal,

add,addu,and_ins,div,divu,mul, mulh,mulhu,rem,remu,nor_ins,or_ins,Itype,

sll,slli,sllv,sra,srav,srl,srlv,sub,subu,xor_ins,slt,sltu,jalr,jr,mfhi,mflo
,mthi,mtlo,NOP,bubble,error,Enquiry,CSRRS}
fetching,decoding,executing,memory,writeback/*,temp1/*,temp2,Enq*/;

real  T1CPI,T2CPI,T3CPI,T4CPI,OverAllCPI;

real
T1ClockCounter,T2ClockCounter,T3ClockCounter,T4ClockCounter,ProcessorClockC
ounter,T1InstCounter,T2InstCounter,T3InstCounter,T4InstCounter,ProcessorIns
tCounter;

logic [3:0] DataVector,InstVector;

logic [31:0]
fetchingAddress,memoryaddress,Swcount,Lwcount,storeddata,loaddata,MissCount
,branchCount,jumpCount,uncompletedSW,uncompletedLW/*,NOPCount*/,InstMissCou
nt,HazardsCount;

logic
DoneT1,DoneT2,DoneT3,DoneT4,EXT1,EXT2,EXT31,EXT32,EXT4,InstMiss,LED0,LED1,L
ED2,LED3,CacheMiss,SwitchAction;

logic [31:0] T0out,T1out,T2out,T3out;
logic [31:0] t0,t1,t2,t3;

assign test = RISC1.Wdata_WB;

always@(*)

begin

DataVector = RISC1.TM1.DataVector;

InstVector = RISC1.TM1.InstVector;

fetchingAddress = RISC1.IF1.FetchingAddress;

end

end

always_comb

begin

if (!nReset)

begin

MissCount = 0;

branchCount = 0;

jumpCount = 0;

//NOPCount = 0;

InstMissCount = 0;

HazardsCount = 0;

end

else

begin

if (RISC1.TM1.CacheMiss == 1) MissCount = MissCount + 1'b1;

if (RISC1.TM1.jump_ID == 1) jumpCount = jumpCount + 1'b1;

if (RISC1.TM1.branch_ID == 1) branchCount = branchCount + 1'b1;

//if (RISC1.TM1.flush) NOPCount = NOPCount + 1'b1;

if (RISC1.TM1.StoreHazard) HazardsCount = HazardsCount + 1'b1;

if (RISC1.TM1.InstMiss) InstMissCount = InstMissCount + 1'b1;

end

end
```

```
always_comb
begin
t1sw = 0;
t2sw = 0;
t3sw = 0;
t4sw = 0;

 if (memory == sw && RISC1.Mem1.mhartID_Mem == 0) t1sw = 1;
 if (memory == sw && RISC1.Mem1.mhartID_Mem == 1) t2sw = 1;
 if (memory == sw && RISC1.Mem1.mhartID_Mem == 2) t3sw = 1;
 if (memory == sw && RISC1.Mem1.mhartID_Mem == 3) t4sw = 1;
 if (!nReset) Swcount = 0;
 else if (memory == sw) Swcount = Swcount + 1;
end
always_comb
begin
t1lw = 0;
t2lw = 0;
t3lw = 0;
t4lw = 0;

 if (memory == lw && RISC1.Mem1.mhartID_Mem == 0) t1lw = 1;
 if (memory == lw && RISC1.Mem1.mhartID_Mem == 1) t2lw = 1;
 if (memory == lw && RISC1.Mem1.mhartID_Mem == 2) t3lw = 1;
 if (memory == lw && RISC1.Mem1.mhartID_Mem == 3) t4lw = 1;
 if (!nReset) Lwcount = 0;
 else if (memory == lw) Lwcount = Lwcount + 1;
end




always_comb
```

```
begin
t1miss = 0;
t2miss = 0;
t3miss = 0;
t4miss = 0;
if (RISC1.TM1.CacheMiss == 1 && RISC1.TM1.mhartID_Mem == 0) t1miss = 1;
if (RISC1.TM1.CacheMiss == 1 && RISC1.TM1.mhartID_Mem == 1) t2miss = 1;
if (RISC1.TM1.CacheMiss == 1 && RISC1.TM1.mhartID_Mem == 2) t3miss = 1;
if (RISC1.TM1.CacheMiss == 1 && RISC1.TM1.mhartID_Mem == 3) t4miss = 1;
end


always_comb
begin
memoryaddress = 0;
storeddata = 0;
if ((executing == sw || executing == lw ))
memoryaddress = RISC1.EX1.MemAddress;
if (executing == sw )
storeddata = RISC1.EX1.Rdata2_Mem;
if (memory == lw )
loaddata = RISC1.Mem1.ReadData;
else loaddata = 0;
end




always_ff @(posedge clk,negedge nReset)
begin
if (!nReset)
begin
```

```
ProcessorClockCounter <= 0;

T1ClockCounter <= 1;

T2ClockCounter <= 1;

T3ClockCounter <= 1;

T4ClockCounter <= 1;

end

else

begin

ProcessorClockCounter <= ProcessorClockCounter + 1'b1;

if (RISC1.TM1.mhartID == 0 && RISC1.TM1.initialisation != 0 && fetching !=
NOP) T1ClockCounter = T1ClockCounter + 1'b1;

if (RISC1.TM1.mhartID == 1 && RISC1.TM1.initialisation != 0  && fetching !=
NOP) T2ClockCounter = T2ClockCounter + 1'b1;

if (RISC1.TM1.mhartID == 2 && RISC1.TM1.initialisation != 0  && fetching !=
NOP) T3ClockCounter = T3ClockCounter + 1'b1;

if (RISC1.TM1.mhartID == 3 && RISC1.TM1.initialisation != 0  && fetching !=
NOP) T4ClockCounter = T4ClockCounter + 1'b1;

end

end

always_ff @(posedge clk,negedge nReset)

begin

if (!nReset)

begin

T1InstCounter <= 1;

T2InstCounter <= 1;

T3InstCounter <= 1;

T4InstCounter <= 1;

ProcessorInstCounter <= 1;

end

else

begin

if ((RISC1.TM1.mhartID == 0) && fetching != NOP)

begin

case (RISC1.IF1.IM1.I[6:0])

`JAL  :T1InstCounter <= T1InstCounter + 2;

`JALR :T1InstCounter <= T1InstCounter + 2;
```

```
`Btype :T1InstCounter <= T1InstCounter + 2;
`LW : T1InstCounter <= T1InstCounter + 5;
`SW : T1InstCounter <= T1InstCounter + 4;
`Itype : T1InstCounter <= T1InstCounter + 3;
`Rtype : T1InstCounter <= T1InstCounter + 3;
`CSRRS : T1InstCounter <= T1InstCounter + 3;
endcase
end
if ((RISC1.TM1.mhartID == 1) && fetching != NOP)
begin
case (RISC1.IF1.IM1.I[6:0])
`JAL  :T2InstCounter <= T2InstCounter + 2;
`JALR :T2InstCounter <= T2InstCounter + 2;
`Btype :T2InstCounter <= T2InstCounter + 2;
`LW : T2InstCounter <= T2InstCounter + 5;
`SW : T2InstCounter <= T2InstCounter + 4;
`Itype : T2InstCounter <= T2InstCounter + 3;
`Rtype : T2InstCounter <= T2InstCounter + 3;
`CSRRS : T2InstCounter <= T2InstCounter + 3;
endcase
end
if ((RISC1.TM1.mhartID == 2) && fetching != NOP)
begin
case (RISC1.IF1.IM1.I[6:0])
`JAL  :T3InstCounter <= T3InstCounter + 2;
`JALR :T3InstCounter <= T3InstCounter + 2;
`Btype :T3InstCounter <= T3InstCounter + 2;
`LW : T3InstCounter <= T3InstCounter + 5;
`SW : T3InstCounter <= T3InstCounter + 4;
`Itype : T3InstCounter <= T3InstCounter + 3;
`Rtype : T3InstCounter <= T3InstCounter + 3;
`CSRRS : T3InstCounter <= T3InstCounter + 3;
endcase
end
```

```
if ((RISC1.TM1.mhartID == 3) && fetching != NOP)
begin
case (RISC1.IF1.IM1.I[6:0])
`JAL  :T4InstCounter <= T4InstCounter + 2;
`JALR :T4InstCounter <= T4InstCounter + 2;
`Btype :T4InstCounter <= T4InstCounter + 2;
`LW : T4InstCounter <= T4InstCounter + 5;
`SW : T4InstCounter <= T4InstCounter + 4;
`Itype : T4InstCounter <= T4InstCounter + 3;
`Rtype : T4InstCounter <= T4InstCounter + 3;
`CSRRS : T4InstCounter <= T4InstCounter + 3;
endcase
end
case (RISC1.IF1.IM1.I[6:0])
`JAL  :ProcessorInstCounter <= ProcessorInstCounter + 2;
`JALR :ProcessorInstCounter <= ProcessorInstCounter + 2;
`Btype :ProcessorInstCounter <= ProcessorInstCounter + 2;
`LW : ProcessorInstCounter <= ProcessorInstCounter + 5;
`SW : ProcessorInstCounter <= ProcessorInstCounter + 4;
`Itype : ProcessorInstCounter <= ProcessorInstCounter + 3;
`Rtype : ProcessorInstCounter <= ProcessorInstCounter + 3;
`CSRRS : ProcessorInstCounter <= ProcessorInstCounter + 3;
endcase
end
end
always @(*)
begin
T1CPI = T1InstCounter / T1ClockCounter ;
T2CPI = T2InstCounter / T2ClockCounter;
T3CPI = T3InstCounter / T3ClockCounter ;
T4CPI = T4InstCounter / T4ClockCounter ;
OverAllCPI = ProcessorInstCounter/ ProcessorClockCounter;
end
```

```
enum logic [1:0]  {Thread0,Thread1,Thread2,Thread3} ThreadID;
 pipelined_RISC  RISC1 (.*);
//assign stall = RISC1.HD1.stall;
assign flush = RISC1.TM1.flush;
assign jump = RISC1.TM1.jump_ID;
assign branch = RISC1.TM1.branch_ID;
assign alu_out = RISC1.EX1.out;
assign a = RISC1.EX1.alu1.a;
assign b = RISC1.EX1.alu1.b;
always_comb
begin
 case (RISC1.TM1.mhartID)
 0: ThreadID = Thread0;
 1: ThreadID = Thread1;
 2: ThreadID = Thread2;
 3: ThreadID = Thread3;
 endcase
 end
initial
begin
  clk =  0;
    forever #5ns clk = ~clk;
end



always@(*)
assert (RISC1.TM1.T1 != 1) else $stop;
initial
begin
nReset = 0;
#1ns
nReset=1;
#1ns
```

```
nReset = 1;
 $monitor ($realtime, " Read Data = %h" ,RISC1.IF1.IM1.I);
end
always @(*)
begin
if (RISC1.IF1.flush || InstMiss) fetching = NOP;
else
begin
case (RISC1.IF1.IM1.I[6:0])
`LW : fetching = lw;
`CSRRS: fetching = CSRRS;
`SW : fetching = sw;
`AUIPC : fetching = AUIPC;
`LUI : fetching = lui;
`JAL : fetching = jal;
`JALR : fetching = jalr;
`Btype : case (RISC1.IF1.IM1.I[14:12])
0: fetching = beq;
1: fetching = bne;
4: fetching = blt;
5: fetching = bgte;
default: fetching = error;
endcase// func3
`Rtype: begin
if (RISC1.IF1.IM1.I[25])
begin
case (RISC1.IF1.IM1.I[14:12])
                    0: fetching = mul;
                    1: fetching = mulh;
                    2: fetching = mulhu;
                    4: fetching = div;
                    5: fetching = divu;
                    6: fetching = rem;
                    7: fetching = remu;
                    endcase
end
else
begin
case (RISC1.IF1.IM1.I[14:12])
0: begin
   if (RISC1.IF1.IM1.I[31:25] == 0) fetching = add;
   else fetching = sub;
   end
1: fetching = sll;
2: fetching = slt;
3: fetching = sltu;
4: fetching = xor_ins;
5: begin
   if (RISC1.IF1.IM1.I[31:25] == 0) fetching = srl;
   else fetching = sra;
   end
6: fetching = or_ins;
7: fetching = and_ins;
endcase//func3
end
end
`Itype: case (RISC1.IF1.IM1.I[14:12])
0: fetching = addi;
1: fetching = slli;
2: fetching = slti;
3: fetching = sltiu;
4: fetching = xori;
6: fetching = ori;
7: fetching = andi;
default : fetching = error;
endcase // func3
endcase
end
```

```
end
always @(*)
begin
if (RISC1.ID1.I_ID == 0 ) decoding = NOP;
else
begin
case (RISC1.ID1.I_ID[6:0])
`LW : decoding = lw;
`CSRRS: decoding = CSRRS;
`SW : decoding = sw;
`AUIPC : decoding = AUIPC;
`LUI : decoding = lui;
`JAL : decoding = jal;
`JALR : decoding = jalr;
`Btype : case (RISC1.ID1.I_ID[14:12])
0: decoding = beq;
1: decoding = bne;
4: decoding = blt;
5: decoding = bgte;
default: decoding = error;
endcase// func3
`Rtype: begin
if (RISC1.ID1.I_ID[25])
begin
case (RISC1.ID1.I_ID[14:12])
                    0: decoding = mul;
                    1: decoding = mulh;
                    2: decoding = mulhu;
                    4: decoding = div;
                    5: decoding = divu;
                    6: decoding = rem;
                    7: decoding = remu;
                    endcase
end
```

```
else
begin
case (RISC1.ID1.I_ID[14:12])
0: begin
   if (RISC1.ID1.I_ID[31:25] == 0) decoding = add;
   else decoding = sub;
   end
1: decoding = sll;
2: decoding = slt;
3: decoding = sltu;
4: decoding = xor_ins;
5: begin
   if (RISC1.ID1.I_ID[31:25] == 0) decoding = srl;
   else decoding = sra;
   end
6: decoding = or_ins;
7: decoding = and_ins;
endcase//func3
end
end
`Itype: case (RISC1.ID1.I_ID[14:12])
0: decoding = addi;
1: decoding = slli;
2: decoding = slti;
3: decoding = sltiu;
4: decoding = xori;
6: decoding = ori;
7: decoding = andi;
default : decoding = error;
endcase // func3
endcase
end
end
always @(*)
```

```
begin
if (RISC1.EX1.I_EX == 0) executing = NOP;
case (RISC1.EX1.I_EX[6:0])
`LW : executing = lw;
`CSRRS: executing = CSRRS;
`SW : executing = sw;
`AUIPC : executing = AUIPC;
`LUI : executing = lui;
`JAL : executing = jal;
`JALR : executing = jalr;
`Btype : case (RISC1.EX1.I_EX[14:12])
0: executing = beq;
1: executing = bne;
4: executing = blt;
5: executing = bgte;
default: executing = error;
endcase// func3
`Rtype: begin
if (RISC1.EX1.I_EX [25])
begin
case (RISC1.EX1.I_EX [14:12])
                    0: executing = mul;
                    1: executing = mulh;
                    2: executing = mulhu;
                    4: executing = div;
                    5: executing = divu;
                    6: executing = rem;
                    7: executing = remu;
                    endcase
end
else
begin
case (RISC1.EX1.I_EX [14:12])
0: begin
```

```
   if (RISC1.EX1.I_EX [31:25] == 0) executing = add;
   else executing = sub;
   end
1: executing = sll;
2: executing = slt;
3: executing = sltu;
4: executing = xor_ins;
5: begin
   if (RISC1.EX1.I_EX [31:25] == 0) executing = srl;
   else executing = sra;
   end
6: executing = or_ins;
7: executing = and_ins;
endcase//func3
end
end
`Itype: case (RISC1.EX1.I_EX[14:12])
0: executing = addi;
1: executing = slli;
2: executing = slti;
3: executing = sltiu;
4: executing = xori;
6: executing = ori;
7: executing = andi;
default : executing = error;
endcase // func3
endcase
end
logic [31:0] temp2address;
always_ff @(posedge clk, negedge nReset)
if (!nReset) tempaddress <= 0;
else
begin
if (fetching == addi) tempaddress <= fetchingAddress;
```

```
temp2address <= tempaddress;

end


logic [15:0] copy [0:262143];

logic [31:0] stored;

always_ff @(posedge clk)

begin

if (RISC1.Mem1.MemWrite_Mem)

begin

copy [RISC1.Mem1.MemAddress] <= RISC1.Mem1.Rdata2_Mem [15:0];

copy [RISC1.Mem1.MemAddress + 1'b1] <= RISC1.Mem1.Rdata2_Mem [31:16];

end

end

always_comb

begin

stored = 0;

if (RISC1.Mem1.MemWrite_Mem) stored = RISC1.Mem1.Rdata2_Mem;


end

logic [31:0] tempdata;

logic dagdog;

always_ff @(posedge clk)

begin

if (RISC1.EX1.ReadEnable) tempdata <= {copy[memoryaddress + 1'b1],
copy[memoryaddress]};

if ((RISC1.TM1.CacheMiss || RISC1.TM1.StoreHazard) &&
(RISC1.TM1.mhartID_Mem == RISC1.TM1.mhartID) && RISC1.IF1.I_ID != 0) dagdog
<= 1;

else dagdog <= 0;

end

initial

begin

forever #10ns

if (RISC1.Mem1.CacheHit == 1 && (RISC1.Mem1.ReadData != tempdata) &&
(RISC1.Mem1.ReadFromExternal == 0) )

begin
```

```
$error ("Fetched Wrong Data");
$stop;
end

end
logic debug;
always_comb
begin
debug = 0;
if (fetchingAddress == 32'h0b4) debug = 1;
end




logic [31:0] t0in,t1in,t2in,t3in;
always @(posedge clk, negedge nReset)
begin
if (!nReset)
begin
t0 <= 0;t1<= 0;t2 <= 0 ;t3 <= 0;
t0in <= 0;t1in <= 0;t2in <= 0 ;t3in <= 0;
end
else
begin
t0 <= {RISC1.AR1.RAM[1],RISC1.AR1.RAM[0]};
t0in <= {RISC1.AR1.RAM[3],RISC1.AR1.RAM[2]};
t1 <= {RISC1.AR1.RAM[5],RISC1.AR1.RAM[4]};
t1in <= {RISC1.AR1.RAM[7],RISC1.AR1.RAM[6]};
t2 <= {RISC1.AR1.RAM[9],RISC1.AR1.RAM[8]};
t2in <= {RISC1.AR1.RAM[11],RISC1.AR1.RAM[10]};
t3 <= {RISC1.AR1.RAM[13],RISC1.AR1.RAM[12]};
t3in <= {RISC1.AR1.RAM[15],RISC1.AR1.RAM[14]};
```

```
end
end
always_ff @(posedge clk,negedge nReset)
begin
if (!nReset)
begin
finisht1 <=0;
finisht2 <=0;
finisht3 <=0;
finisht4 <=0;
SwitchCount <= 0;
end
else
begin
if (t0in == 1000) finisht1 <= 1;
if (t1in == 1000) finisht2 <= 1;
if (t2in == 1000) finisht3 <= 1;
if (t3in == 1000) finisht4 <= 1;
if (RISC1.TM1.SwitchAction) SwitchCount <= SwitchCount + 1'b1;
end
end
logic [31:0] waste;
logic [31:0] DataMisstimes;
logic [31:0] InstMisstimes,Storehazardtimes;
real efficency;
always_ff @(posedge clk, negedge nReset)
if (!nReset)
begin
waste <= 0;
DataMisstimes <= 0;
InstMisstimes <= 0;
Storehazardtimes <= 0;
end
else
```

```
begin

waste <= (executing == NOP) ? waste + 1'b1 : waste;

DataMisstimes <= (RISC1.TM1.CacheMiss) ? DataMisstimes + 1'b1 :
DataMisstimes;

InstMisstimes <= (RISC1.TM1.InstMiss) ? InstMisstimes + 1'b1 :
InstMisstimes;

Storehazardtimes <= (RISC1.TM1.StoreHazard) ? Storehazardtimes + 1'b1 :
Storehazardtimes;

end

always_comb

begin

if (finisht1 && finisht2 && finisht3 && finisht4)begin

 $display("report for 4 threaded processor");

 $display("total cycles are : %d",ProcessorClockCounter);

 $display("total waste cycles are : %d",waste);

 $display("total data miss are : %d",DataMisstimes);

 $display("total inst miss are : %d",InstMisstimes);

 $display("total store hazards are : %d",Storehazardtimes);

 $display(" system switch between threads : %d", SwitchCount);

 $display("instruction per clock : %0.2f ", ((ProcessorClockCounter -
waste)/ProcessorClockCounter));

 $stop;

 end

end

always_ff @(posedge clk) memory <= executing;

always_ff @(posedge clk) writeback <= memory;

endmodule
```

28 opcode file

```
//

// File Name: opcode.sv

// Function: instruction opcode

// Author: Mohammad Abu Alhalawe

// Last rev.: 05/09/19

//

`define LUI    7'b0110111

`define AUIPC  7'b0010111

`define JAL     7'b1101111

`define JALR    7'b1100111

`define Btype   7'b1100011

`define LW       7'b0000011

`define SW       7'b0100011

`define Itype    7'b0010011

`define Rtype    7'b0110011

`define Enquiry  7'b1111111

`define CSRRS    7'b1110011
```

29 ALU codes

```
//
// File Name: alucodes.sv
// Function: ALU codes
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
`define     alu_add   5'h00
`define     alu_addu  5'h01
`define     alu_sltu  5'h02
`define     alu_subu  5'h03
`define     alu_slli  5'h04
`define     alu_srli  5'h05
`define     alu_srai  5'h06
`define     alu_sub   5'h07
`define     alu_and   5'h08
`define     alu_or    5'h09
`define     alu_xor   5'h0A
`define     alu_sll   5'h0B
`define     alu_srl   5'h0C
`define     alu_sra   5'h0D
`define     alu_nor   5'h0E
`define  alu_slt  5'h0F
`define alu_mul   5'h10
`define alu_div   5'h11
`define alu_mulh  5'h12
`define alu_divu  5'h13
`define alu_rem   5'h14
`define alu_remu  5'h15
`define alu_mulhu 5'h16
```

30 Test C file code

```
//
// File Name: main.c
// Function: test for 4 threads
// Author: Mohammad Abu Alhalawe
// Last rev.: 05/09/19
//
 asm("csrrs tp,mhartid,0");//0
 asm("li t0,0");//4
 asm("li t1,1");//8
 asm("li t2,2");//c
 asm("li t3,3");//10
 asm("bne tp,t0,0x2c  ");//14
 asm("li sp, 0x00002000");//18
 asm("li s0, 0x00002000");//1c
 asm("li ra, 0x00000000");//20
 asm("j main ");//24
 asm("bne tp,t1,0x44  ");//28
 asm("li sp, 0x00004000");//2c
 asm("li s0, 0x00004000");//30
 asm("li ra, 0x00000000");//34
 asm("j main ");//38
 asm("bne tp,t2,0x5c  ");//3c
 asm("li sp, 0x00006000");//40
 asm("li s0, 0x00006000");//44
 asm("li ra, 0x00000000");//48
 asm("j main ");//4c
 asm("li sp, 0x00008000");//50
 asm("li s0, 0x00008000");//54
 asm("li ra, 0x00000000");//58
 int thread0(int n);
 int thread1(int n);
 int thread2(int n);
 int thread3(int n);
int main(){

        int threadID,t0out=0,t1out=0,t2out=0,t3out=0;
        int i = 0,j = 0,k = 0,w = 0;
        asm("csrrs %[result],mhartid,0" : [result] "=r" (threadID) :);



                if (threadID == 0)
                {
                        while (1){
                        t0out = thread0(i);
                        *(volatile int*)0x00000000 = t0out;
                        *(volatile int*)0x00000002 = i;
                        i++;
                        }

                }

                if (threadID == 1)
                {
                        while (1){
                        t1out = thread1(j);
                        *(volatile int*)0x00000004 = t1out;
                        *(volatile int*)0x00000006 = j;
                        j++;
                        }
                }

                if (threadID == 2)
                {
                        while (1){
                        t2out = thread2(k);
                        *(volatile int*)0x00000008 = t2out;
```

```
                    *(volatile int*)0x0000000A = k;
                    k++;
                    }


              }
              if (threadID == 3)
              {
                    while (1){
                    t3out = thread3(w);
                    *(volatile int*)0x0000000C = t3out;
                    *(volatile int*)0x0000000E = w;
                    w++;
                          }
              }
        }
 int thread0(int n){
if (n == 0)
 return 0;
  else
    return(3 + thread0(n-1));
 }
 int thread1(int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return (thread1(n-1) + thread1(n-2));
}
 int thread2(int n){
 if (n == 0)
 return 0;
  else
    return(4 + thread2(n-1));
 }
```

```
int thread3(int n)
{
  if (n == 0)
    return 0;
  else
    return(n + thread3(n-1));
}
```