

Design and Implementation of a 4 way Out-Of-Order Processor based on ARMv7 ISA

A project Report

submitted by

Bhavana V

Sandeep G S P

in partial fulfillment of the requirements

for the award of the degree of

Master of Technology

Under the guidance of

Dr. Madhu Mutyam



Department of Electrical Engineering
Indian Institute of Technology Madras

May 2015

Thesis Certificate

This is to certify that the thesis titled **Design and Implementation of a 4 way Out-Of-Order Processor based on ARMv7 ISA**, submitted by **Bhavana V** and **Sandeep G S P**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Madhu Mutyam

Project Guide

Associate Professor

Dept. of Computer Science and Engineering

IIT-Madras, 600 036

Place : Chennai

Date :

Acknowledgments

We wish to express our gratitude to everyone who contributed in making this work a reality. We would like to thank Indian Institute of Technology Madras for giving an opportunity to pursue post graduation. We are grateful to electrical engineering department for providing the facilities for the same.

This work would not have been possible without the support and guidance of our guide Dr. Madhu Mutyam, Associate Professor, Dept of Computer Science and Engineering. We would like to express our deepest gratitude to him for the guidance.

We extend our sincere gratitude to Dr. Nitin Chandrachoodan, Associate Professor, Dept of Electrical Engineering for providing good lab facilities, access to Xilinx license and FPGA boards needed for the project.

We also express our sincere thanks to Dr.V.Kamakoti, Professor, Dept of Computer Science and Engineering, Dr.C. Chandra Sekhar, Professor, Dept of Computer Science and Engineering and all our Embedded systems labmates for their profound support in our project journey.

Finally, we are deeply grateful to our parents for their love and affection over years and thanks to all our well-wishers and friends for their cheerful dispositions, which are vital for sustaining the effort required for completing this project work.

Abstract

Microprocessors have evolved greatly over the past few decades from single cycle state machines, to pipelined architectures, to wide issue superscalar processors to out of order execution implementations. This project implements one such 4 way single threaded out-of-order processor based on ARM v7 Instruction Set Architecture (ISA) in Verilog HDL. The motivation behind the project is the implementation of single cycle MIPS processor design in VLSI Design Lab (EE5703). This microprocessor was designed to exploit instruction level parallelism as much as possible while still maintaining reasonable amount of logic per clock cycle. Ultimately the design implemented is capable of fetching, decoding, renaming, issuing, executing, and retiring up to four instructions per clock cycle with speculative execution, with appropriate optimization performed in each stage in order to improve the efficiency. Upon successful implementation, the design is simulated using a series of test benches and synthesized for Virtex 5 FPGA.

Contents

1	Introduction	1
1.1	Classification of Microarchitectures:	1
1.2	Importance of Instruction Set Architecture:	2
1.3	ARMv7 Instruction Set	2
1.4	ARM processor variants	3
2	Processor Micro-architecture	5
3	Instruction Fetch Stage	7
3.1	Program Counter	8
3.2	Instruction Memory design	8
3.2.1	Instruction Fetch Alignment	8
4	Instruction Decode stage	9
4.1	Instruction Decode Stage-1	9
4.1.1	Branch Prediction Unit	10
4.2	Instruction Decode Stage-2	14
5	Dispatch Stage	16
5.1	Register file read ports bottleneck	16
5.2	Dispatch Stage - 1	18

5.2.1	ROB Allocation	18
5.2.2	Intra-dependency	19
5.2.3	Unique Reads	21
5.2.4	Branch Buffer	22
5.3	Dispatch Stage - 2	24
5.3.1	The $O(1)$ complexity problem	24
5.3.2	Register file design	25
5.3.3	Register Renaming	25
5.3.4	Register file data bus	27
5.3.5	Reservation slot Allocation	27
5.3.6	Dependency matrix	27
6	Reservation Station	28
6.1	Issue logic	30
7	Execution Stage	31
7.1	Integer Unit	32
7.2	Shifted Integer Unit	33
7.3	MAC Unit	35
7.4	Load and Store Unit	36
7.4.1	Data Memory design	36
7.4.2	Store Buffer	37
7.5	Branch Unit	38
7.6	Common Data Bus (CDB)	39
7.6.1	CDB Arbiter	40
7.7	Program Status Register (PSR)	40
8	The commit stage	41

8.1	Re-Order Buffer	42
8.2	Write-back	43
8.3	Exception handling	43
9	Centralized Control Unit	44
10	Synthesis results	46
11	Conclusion and Future Work	52
	Bibliography	54
A	Assembler	57
B	Matlab based RS and ROB simulator	58
C	Instruction Set Architecture	62
D	Processor Variants	67
D.1	ARM processor Variants	67
D.1.1	Single Cycle Design	67
D.1.2	Cannonical 5 Stage pipeline design	68
D.2	MIPS processor Implementations	69
D.2.1	Single and Multi-cycle design	71
D.2.2	Canonical 5 stage pipeline design	71
D.2.3	Two-Way In-order Super-scalar processor Design	72

List of Figures

2.1	High level data-path of processor's (a) front end and (b) back end	5
3.1	Instruction fetch block diagram	7
4.1	Block diagram of stage 1 of instruction decode	10
4.2	Performance of branch instruction routing module under different scenarios (a) External stall before branch routing (b) Flushing of unconditional branches in decode stage and (c) External stall in between multicycle stall: the branch routing begins all over again	11
4.3	State diagram of (a) two bit predictor and (b) modified two bit counter	12
4.4	Hybrid branch predictor architecture	12
4.5	Misprediction per 1000 branch instructions of branch predictors across 40 benchmarks from championship branch prediction framework	13
4.6	Block diagram of stage 2 of instruction decode	15
5.1	Block diagram of (a) stage 1 of instruction dispatch and (2) stage 2 of instruction dispatch	17
5.2	Simulation of ROB allocation for the above cases	18

LIST OF FIGURES

5.3	Comparison algorithm among the instructions in a cluster	19
5.4	(a) Internal block diagram of intra-dependency checker (IDC), (b) Internal block diagram of a comparator used in IDC and (c) A three level architecture to perform intra-dependency removal among the instruction cluster.	20
5.5	Example illustrating the functionality of unique reads module	21
5.6	Overview of interface of the branch buffer with other stages in pipeline	22
5.7	Internal architecture of the branch buffer describing the different parts of the branch buffer and the limited access rights to different stages	23
5.8	Register file divided in to regions	24
5.9	Register file read ports allocation	25
5.10	Register renaming mechanism through re-order buffer .	26
6.1	Simulations of a simple out-of-order issue logic implemented in this design	30
7.1	Block diagram of execution unit containing all the functional units and reservation stations	31
7.2	Block diagram of integer unit and program status register organization	32
7.3	(a) Block diagram of the 2 staged shifted integer unit and (b) architecture of stage 2 of shifted integer unit .	33
7.4	(a) High level block diagram of the 3 stage MAC unit, (b) partial products generation and level 1 CSA in stage 1	34

LIST OF FIGURES

7.5	(a) block diagram of stage 2 of MAC indicating all levels of CSA and (b) Load and store stage and its interconnection with the store buffer	36
7.6	Example illustrating misalignment in data memory . .	37
7.7	Load forwarding cases (a) without exception and (b) with exception	38
8.1	Re-order buffer high level interface block diagram . . .	41
8.2	ROB port requirements	42
B.1	The matlab simulator	59
B.2	Simulation Results for the configuration 32-8-2-2-2-4 (a) Reservation Station slots availability and overall reservation station stall and (b) re-order buffer stall during the course of simulation	60

List of Tables

4.1	Truth table for updating the selector table	13
5.1	Offset generation for ROB allocation for some conditions of valid instructions in a cluster	18
6.1	Number of entries in each reservation station.	28
7.1	Number of carry save adders in each of the CSA level .	35
7.2	Conditional codes of ARMv7 for branch instructions . .	39
9.1	Priorities of stall signals in centralized control unit. Sig- nals generated to different stages are indicated as stall/flush	45
10.1	Synthesis results of each stage	51
B.1	The delay ranges for dependency and execution unit for different types of instructions	58
B.2	The reservation station and ROB stall percentages for different combinations of sizes. The configuration is to be read as ROB Size - IU RS size - SIU RS size - MAC RS size - LS RS size - B RS size	59
D.1	Mini MIPS ISA	70

Abbreviations

ISA	I nstruction S et A rchitecture
ILP	I nstruction L evel P arallelism
CPI	C locks P er I nstruction
IPC	I nstructions P er C ycle
IF	I nstruction F etch
PC	P rogram C ounter
ID	I nstruction D ecode
BPU	B ranch P rediction U nit
BTB	B ranch T arget B uffer
BTA	B ranch T arget A ddress
FTA	F ollow T hrough A ddress
DI	D ispatch I nstruction
IDC	I ntradependency C hecker
RF	R egister F ile
RS	R eservation S tation
ALU	A rithmetic and L ogical U nit
IU	I nteger U nit
SIU	S hifted I nteger U nit
MAC	M ultiply and A Ccumulate
CSA	C arry S ave A dder
LSU	L oad and S tore U nit
BU	B ranch U nit
RAW	R ead A fter W rite
WAW	W rite A fter W rite
CDB	C ommon D ata B us
ROB	R e- O rders B uffer
NOP	N o O peration

Chapter 1

Introduction

The processor microarchitecture has undergone a continuous evolution. This evolution is fuelled by 2 types of factors namely technology scaling and workload evolution. According to technology scaling, every generation provides transistors that are smaller, faster and less energy consuming. On the other hand, each generation resulted in an increasing number of architectural features in the processor to better exploit the characteristics of user applications [23].

1.1 Classification of Microarchitectures:

Initially microprocessors used pipelining concept to overlap the execution of instructions and improve the performance. This potential overlap among instructions is called instruction-level parallelism (ILP). Later a wide range of techniques have been emerged for extending the basic pipelining concepts by increasing the amount of parallelism that is exploited among instructions.

There are two largely separable approaches to exploit ILP: an approach that relies on hardware to exploit the parallelism dynamically, and an approach that relies on software technology to find parallelism, statically at compile time [25]. Processors using the dynamic, hardware-based approach dominate in the market; whereas those using the static approach, have more limited uses in scientific or application-specific environments.

Over the past decade, superscalar microprocessors have become a source of tremendous computing power. To satisfy the ever-growing need for higher levels of comput-

1.2 Importance of Instruction Set Architecture:

ing power, computer architects need to investigate techniques that continue improving the performance of superscalar microprocessors, while considering both changing technology and applications. The maximum number of instructions processed in parallel, also known as the width of the microarchitecture, is typically four for the fastest microprocessors available today.

There is an enormous need for investigating superscalar microarchitectures that judiciously use hardware complexity for exploiting significant levels of instruction-level parallelism, while permitting a fast clock. We call such microarchitectures complexity-effective superscalar microarchitectures [3]. Specifically, the purpose of choosing out-of-order processor architecture compared to in-order type is to increase the amount of ILP by providing more freedom to hardware for choosing which instructions need to process in each cycle at the cost of complexity in hardware. This is referred to as dynamic instruction scheduling

1.2 Importance of Instruction Set Architecture:

The question of ISA design and specifically RISC vs. CISC is an important concern, when chip area and processor design complexity are the primary constraints. In the past decade, the ARM ISA (RISC) has dominated mobile and low power embedded computing domains and the x86 ISA (CISC) has dominated desktops and servers. Rather than being exclusively desktops and servers, today's computing landscape is significantly shaped by smartphones and tablets. While area and chip design complexity were previously the primary constraints, energy and power constraints predominantly dominate at present [1].

1.3 ARMv7 Instruction Set

The ARM architecture has evolved significantly since its introduction. There are eight major versions with the latest one ARMv8 targeted at high performance processors for current generation smartphones [2]. The architectural simplicity of ARM processors leads to very small implementations, which means devices consume lower power.

The ARM architecture is a Reduced Instruction Set Computing (RISC) architecture, as it incorporates these architectural features:

- A large uniform register file
- A load / store architecture, where data-processing operations only operate on register contents, not directly on memory contents
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.

In addition, the ARM architecture also provides:

- Instructions that combine a shift with an arithmetic or logical operation
- Auto increment and auto decrement addressing modes to optimize program loops.
- Load and Store Multiple instructions to maximize data throughput.
- Conditional execution of many instructions to maximize execution throughput.

These enhancements to a basic RISC architecture leads to designs, that achieve a good balance of high performance, small program size, low power consumption, and small silicon area. All these factors and architectural support available lead us to use ARMv7 ISA.

1.4 ARM processor variants

ARM released many processor architectures based on ARMv7 ISA. Some of the application oriented architectures that impacted the smartphone and tablets market are [27]:

- **Cortex A5:** This processor is the smallest, lowest cost and lowest power ARMv7 application processor designed for smart devices like wearables, feature phones and low cost smart phones.
- **Cortex A7:** This processor powers sub-\$100 entry-level smartphones, as well as a number of high-end wearable devices. The processor led the multicore revolution for entry-level and mid-range mobile smartphones. The Cortex-A7 processor is architecturally aligned with the high-performance Cortex-A17 and Cortex-A15 processors.

- **Cortex A8:** This was the first ARMv7 based single core processor introduced in the market for smartphones and printers. It is a 2-way in-order machine.
- **Cortex A9:** This processor design delivered exceptional capabilities while using considerably low power than high-performance computer platforms. It is one of the first processor that implements the entire ARMv7 architecture. It is a dual issue superscalar and out-of-order processor with a dynamic pipeline length from 8 - 11 stages.

Cortex A15 and A17 are improved versions of cortex A7. The current flagship processors A53,A57 and A72 are all based on ARMv8 which was designed for performance.

Chapter 2

Processor Micro-architecture

The project is aimed at the design of a 4 way super-scalar out-of-order processor based on ARMv7 instruction set architecture inspired from several standard designs discussed in [3, 4, 5, 6]. As a 4 way processor, the design is capable of processing 4 instructions in each of its pipeline stage. Fig. 2.1 shows the high level data-path

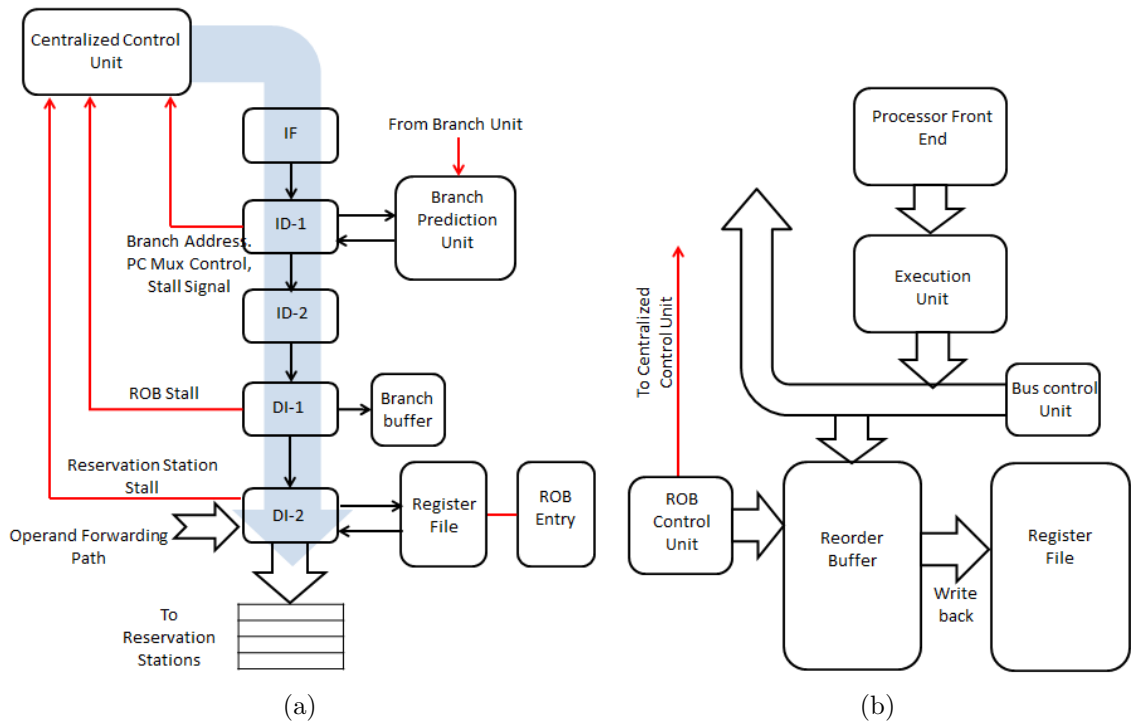


Figure 2.1: High level data-path of processor's (a) front end and (b) back end

for the CPU's front and back end. This processor follows a I2OI based design where fetch, decode and write back stages are done in-order, where as execution happens out of order.

Front end of the processor is responsible for fetching instructions, decoding the raw instruction words. They are then renamed so that back-end can handle dependencies. Speculative control flow is done in the front end to provide a constant stream of instructions. Upon resolving a speculated flow direction, the re-order buffer (ROB) can direct the front end to fetch in the correct direction in case of an exception. This will ensure precise exception. The front end of the processor will be stalled if the back end runs out of space to buffer instructions.

The back end of the CPU is responsible for taking the renamed instructions and issuing them to be executed as their operands become available. The only dependency that remains is the RAW dependency which is ultimately the limiting factor in how fast the processor can execute the program. The out-of-order execution unit can execute the other independent instructions while other dependent instructions wait for source operands in the reservation station. The sequential ordering of the program is maintained through the re-order buffer which commits instructions in order as they complete execution. The major components of the back end are the reservation stations, load-store queues, the reorder buffers. There are 6 functional units (2 Integer units, 1 shifted-integer unit, 1 multiply and accumulate unit, 1 load & store unit and 1 branch unit) in the execution stage. As the results are computed by the execution unit, they are forwarded to the reservation stations.

Each of the pipeline stage functionality is discussed in detail in its respective chapter.

Chapter 3

Instruction Fetch Stage

The instruction fetch unit is responsible for feeding the processor with instructions to execute and hence it is the first stage of the pipeline. The fetch stage mainly comprises of instruction memory, logic needed to compute fetch address or the instruction memory address i.e. Program counter and logic for identifying alignment issues. Since, the design is a 4 way, out-of order processor, the fetch unit has to supply 4 instructions every cycle to the pipeline.

Conventional high performance processor design incorporates a branch prediction unit as a part of the fetch stage so as to aid in computation of next fetch address when a branch is encountered, but with the increase in the fetch width, the number of ports needed to process all the instructions in parallel for the branch prediction unit

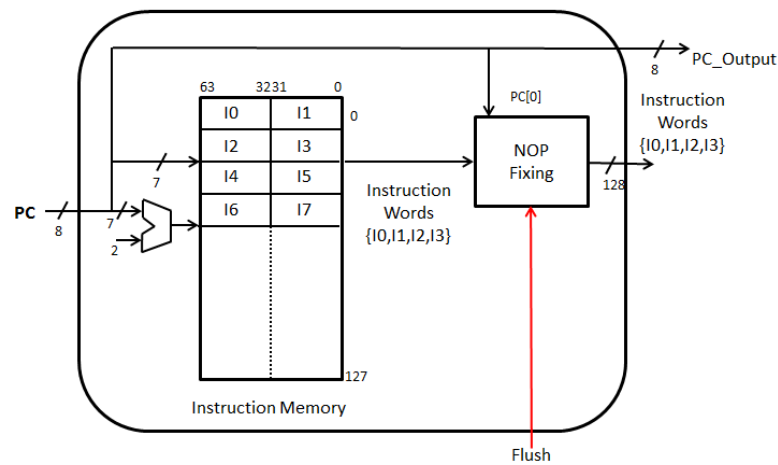


Figure 3.1: Instruction fetch block diagram

also increases. Owing to the increase in hardware complexity, the branch prediction unit has been moved to decode stage. This incurs a compulsory 1 cycle loss whenever a conditional branch is predicted taken.

3.1 Program Counter

The program counter (PC) is a register that acts as a pointer to the instruction memory, in other words it is the fetch address. In the current design the PC is an 8 bit register. The program counter is updated every clock cycle by the centralized control unit. In case of a stall, the program counter is not updated.

3.2 Instruction Memory design

The instruction memory holds instructions of the program. In this design the memory is implemented using block memory IP core from xilinx. The memory is a dual ported block ROM of depth 128 locations each of 64 bits wide (1KB). The memory has two ports to read 4 instructions of a cluster. The two read ports are driven by PC and $(PC + 2)$. Instruction fetch alignment needs to be taken care in this type of design.

3.2.1 Instruction Fetch Alignment

Apart from the advantage of reducing the number of read ports by going for a wider instruction memory, we have memory alignment issues that will reduce the performance of the processor, for example in fig. 3.1 if the control based instruction points the next fetch address to I5 instruction, then due to limitation of wide instruction memory even I4 instruction is also fetched. This is flushed later by alignment monitor in the instruction fetch stage, leading to a degradation of performance.

Chapter 4

Instruction Decode stage

The instruction decode stage understands the schematics of an instruction and defines how it should be executed in the processor. This stage identifies the type of instruction and the resources needed to execute the instruction. The input to this stage is usually a stream of raw instruction bits from the instruction memory.

This decode functionality is implemented in two stages. The first stage of the decode unit is responsible for resolving the branch instructions by predicting the outcome of conditional branches and then second stage of the decode unit contains four control units that can decode the instructions in parallel.

4.1 Instruction Decode Stage-1

The first stage of instruction decode is carefully modeled to resolve all branch instructions among the 4 instruction cluster (Fig.4.1). Conventional scalar pipeline processor designs resolve branches in the fetch stage itself so that the new fetch address is available in the same cycle. The latest processor design that exploits ILP by issuing more than 1 instruction every cycle cannot afford to resolve for branch instruction during fetch stage, as we would be needing a heavily ported branch prediction unit to accommodate all instructions.

This lead to, moving the branch resolving unit to decode stage, providing flexibility to optimize the hardware, one such optimization is to have a single ported branch resolution unit, as the percentage of branch instructions in program varies between 10% to 20%. This means that we might have a maximum of 1 or 2 branches

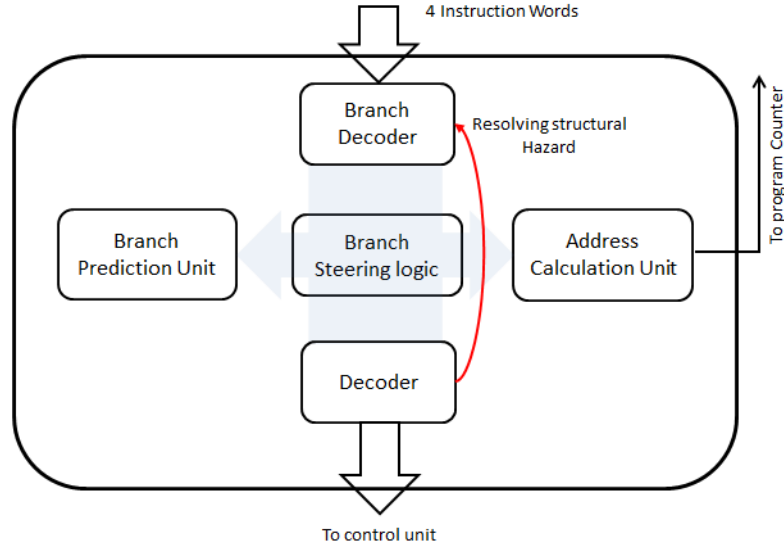


Figure 4.1: Block diagram of stage 1 of instruction decode

per instruction cluster. An instruction router is designed to route the leading branch instruction, if present, in a cluster to the branch predictor and address calculation unit. A structural hazard occurs based on the outcome of first branch (i.e., if not taken) and there are more branches in the cluster. Figs. (4.2a, 4.2b, 4.2c) shows the simulations of certain cases of structural hazard during branch routing.

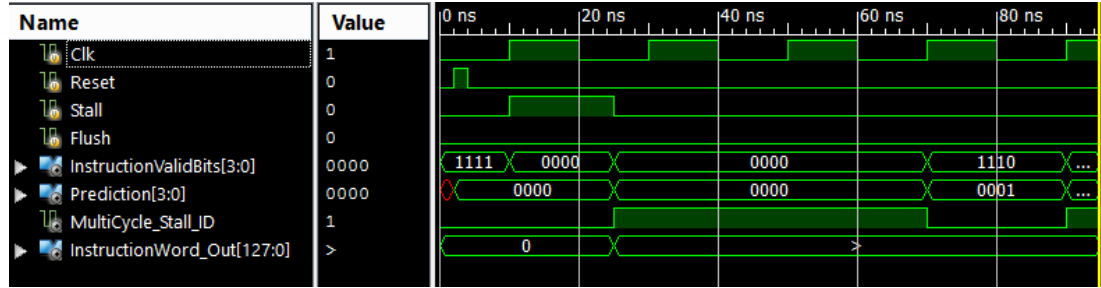
The routed branch instruction is also sent to address calculation unit to determine the next fetch address. This address is sent to the program counter based on the prediction of the conditional branch. The next section deals with the hybrid branch predictor.

4.1.1 Branch Prediction Unit

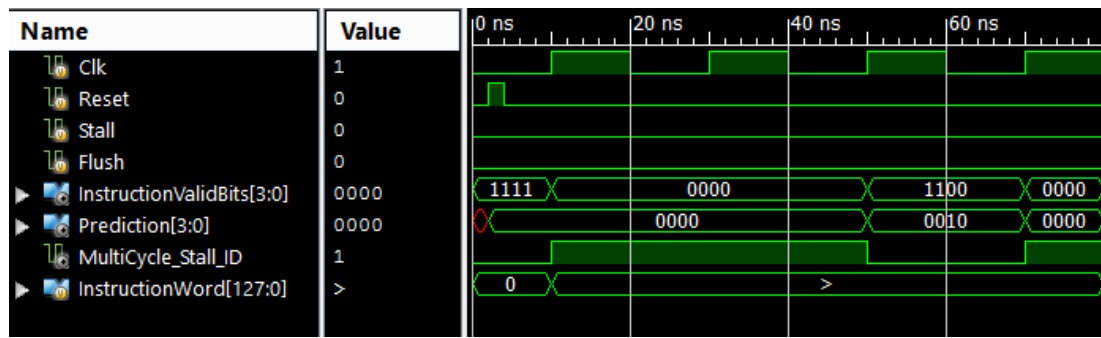
High performance processors have capacity to issue more than 1 instruction every cycle. In order to achieve this level of instruction parallelism, we will have to compute the next fetch address every cycle. However, fetch address from a conditional branch cannot be determined until we execute the branch instruction which leads to degradation of performance. Branch instruction becomes an intrinsic part in determining the processor performance.

The branch performance was improved by predicting the outcome of the branch at an early stage and later validating the prediction. There are two approaches in

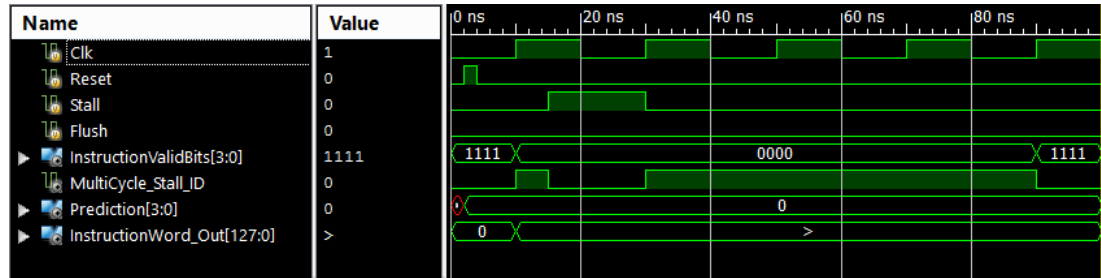
4.1 Instruction Decode Stage-1



(a)



(b)



(c)

Figure 4.2: Performance of branch instruction routing module under different scenarios (a) External stall before branch routing (b) Flushing of unconditional branches in decode stage and (c) External stall in between multicycle stall: the branch routing begins all over again

achieving better branch performance: First approach is to design a better branch predictor with lower misprediction rate and second is to decrease the branch misprediction penalty. Some well known branch predictors [8] are bimodal (two bit predictor), global history based predictor, local history based predictor, correlating branch predictor, gshare predictor, bi-mod predictor, g-skewed predictor, two level

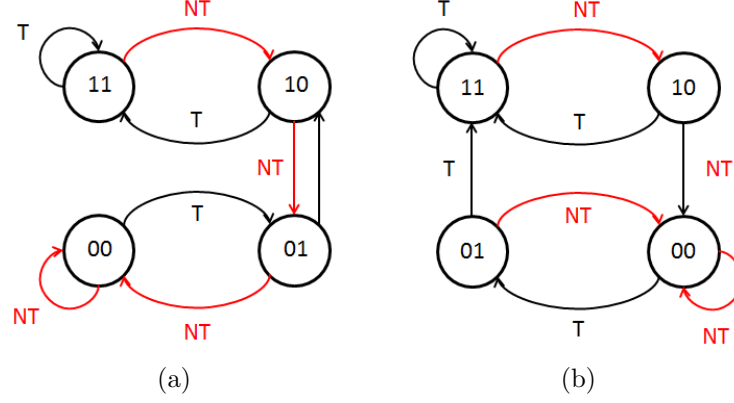


Figure 4.3: State diagram of (a) two bit predictor and (b) modified two bit counter

adaptive branch predictor [6], tournament branch predictor [4] etc.

Farlang [7] introduced the concept of combining the branch predictors to achieve better prediction accuracy. He investigated that the standard branch predictor has distinct advantages on certain scenarios. Global branch predictors perform well when the direction taken by sequentially executed branches are highly correlated and local

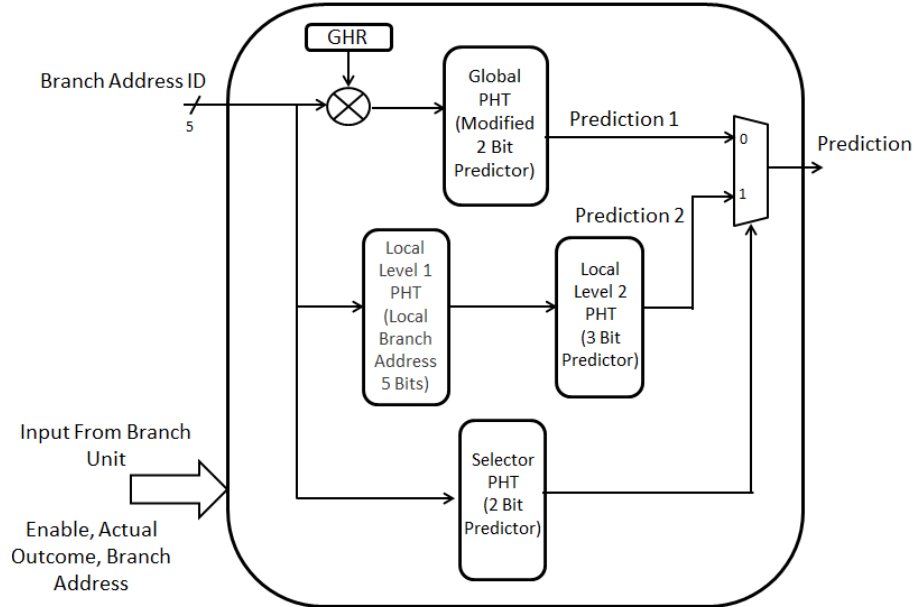


Figure 4.4: Hybrid branch predictor architecture

Predictor 1	Predictor 2	Outcome
Incorrect	Incorrect	No Change
correct	Incorrect	Decrement
Incorrect	correct	Increment
correct	correct	No Change

Table 4.1: Truth table for updating the selector table

branch predictor works well when the branches have repetitive patterns. The current design incorporates a branch predictor which is a hybrid of gshare and local history based branch prediction. Fig. 4.4 shows the structure of the predictor. A selector or choice predictor is used to dynamically select between the two predictions. The selector predictor is an array of saturating 2 bit counters. Table 4.1 explains the updation scheme of the selector. Gshare predictor reduces the warm up phase needed for the branch predictor and local history based predictor takes time in training to the type of patterns.

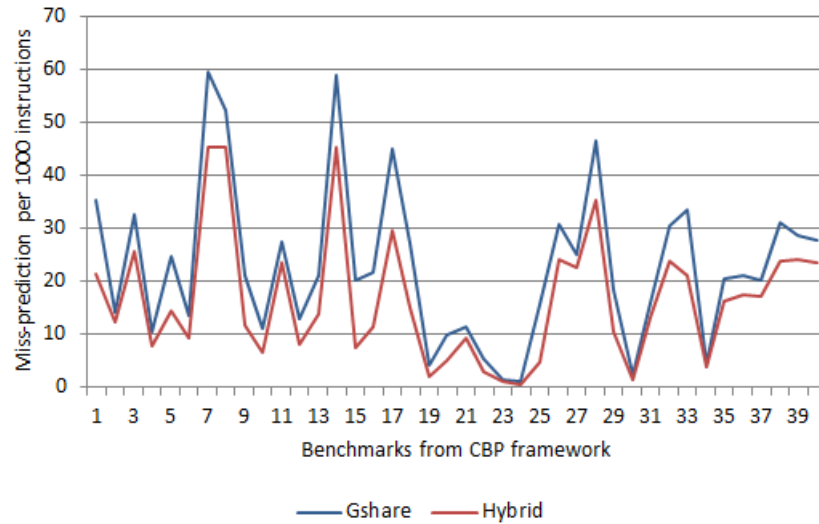


Figure 4.5: Missprediction per 1000 branch instructions of branch predictors across 40 benchmarks from championship branch prediction framework

The gshare predictor has 32 entries, each containing a saturating 2 bit counter (Fig. 4.3a). The table is indexed by branch address hashed with the global branch history register. The local history based predictor on the other hand is a two level prediction scheme that stores the history of each branch separately and then the local branch history is used to index another table of three bit saturating counters (Fig. 4.3b). Both the local predictor tables have 32 entries. The overall hardware budget of this branch predictor is 48 bytes.

The predictor is updated whenever the respective branch result is validated in the branch unit of execution stage. This updation might not be done in order but waiting for the branch instruction to update till the commit stage will mean that several branches will use the old versions of history [10].

The performance of the predictor was evaluated on the championship branch predictor framework [26] across 40 benchmarks. The benchmark contains all sorts of branches - conditional, unconditional, call and returns. Fig. 4.5 illustrates the performance difference between the gshare and hybrid predictor of similar sizes. The hybrid predictor has an average misprediction rate of 16.46 mispredictions per 1000 branch instructions.

4.2 Instruction Decode Stage-2

The next stage of the pipeline deals with generating control signals and functional unit allocated to all the valid instructions that enter this stage. As all the control based instructions are resolved, we can perform decoding in parallel. The input to this stage is also a stream of raw instruction word bits but each instruction has an additional bit to indicate if it is valid or not. All NOP instructions are made invalid in this stage so that they are not further processed. Fig. 4.6 gives a high level block diagram of the instruction decode. This stage is capable of decoding 4 instructions in parallel. Hence, it does not pose any structural hazard. The decoding unit consists of a 3 level look-up table for generating the control signals. The decoding has become simple because of the RISC based ISA of ARM. The efficiency of control logic is determined by the number of bits that is needed to differentiate instructions, for example if we have 100 instructions in the ISA, then we must use a maximum of only 7 bits. Higher efficiency can be obtained if we get to the root of how the instruction set architecture is built. The information extracted from decoding the instruction

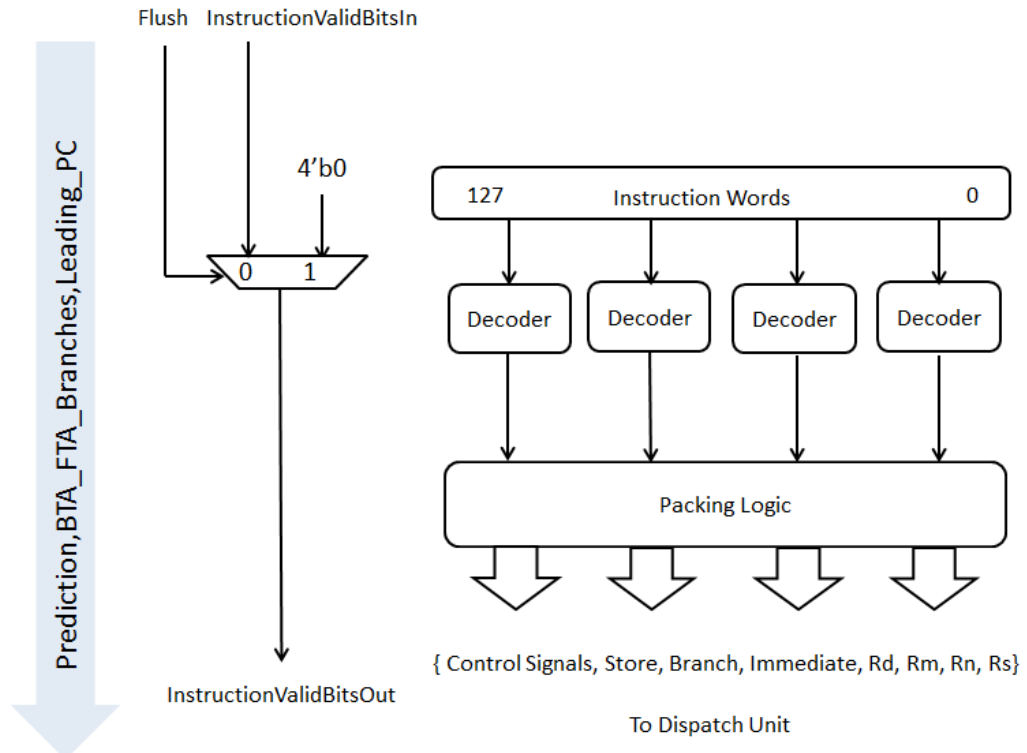


Figure 4.6: Block diagram of stage 2 of instruction decode

includes the type of source registers, type of instruction and control signals that are needed.

The decoder apart from generating control signals also decides which functional unit or reservation station the instruction must be routed. This is important because the out of order nature is introduced at the execution stage. The instruction can take one of the many paths available to writeback based on the type of instruction. This step is also known as allocation of reservation station.

The number of control signals needed for each instruction varies, as they can take different paths before writeback. The architecture supports for maximum number of control signals until reservation station, where they split in to different functional units.

Chapter 5

Dispatch Stage

The next stage in the pipeline is the dispatch stage. The main functionality of this stage is to allocate resources to instructions, reading operands from the register file, removing false dependencies through register renaming and dispatching instructions to execution unit. The allocation normally includes reserving some of the resources that instructions will use in future, like entries in reservation station and re-order buffer. If the resources are not available, then the instructions are stalled in that stage. A standard ROB based register renaming is done, to get rid of the name dependencies dynamically.

5.1 Register file read ports bottleneck

The bottle neck is with the number of read ports for a register file. [11, 12] have discussed the complexities involved in register file designs as the fetch width increases. Several techniques such as use of delayed write-back queues [13], use of bypass hint [14], register file caching and register file banking for write-back [14, 15]. The current design has two additional pre-processing techniques that play an important role in reducing the load on register file read ports.

The 4 way design of a processor can fetch 4 instructions per cycle. ARMv7 instruction set architecture supports some instructions that require 3 operands per cycle. This means that, per instruction cluster, a maximum of 12 operand reads need to be performed per cycle. Some processor designs implement as many read ports as needed, assuming the worst case scenario where the issue width is fully unitized

5.1 Register file read ports bottleneck

and all instructions read operands from the register file [4, 5]. The area, power and access latency of the register file increases with the number of read ports [29]. This can be possible if we have 12 read ports, but designing 12 ports will incur a lot of hardware requirement as we will also need to have 12 read ports for ROB for renaming. Reducing the number of read ports would incur structural hazard. A conventional register file supports up to 4 or 5 read ports.

In this unconventional design, we have a register file of 4 read ports, this means that a maximum of 3 cycles are needed. It has been shown in the literature [29] that most of the sources are read from the operand forwarding path rather than the register file, exploiting this we propose two pre-processing techniques: intra-dependency removal and unique reads that aid in reducing the load on the register file each cycle. These techniques are dealt in detail in respective sections.

The dispatch functionality is divided across two stages, the block diagram of both the stages are shown in fig. 5.1.

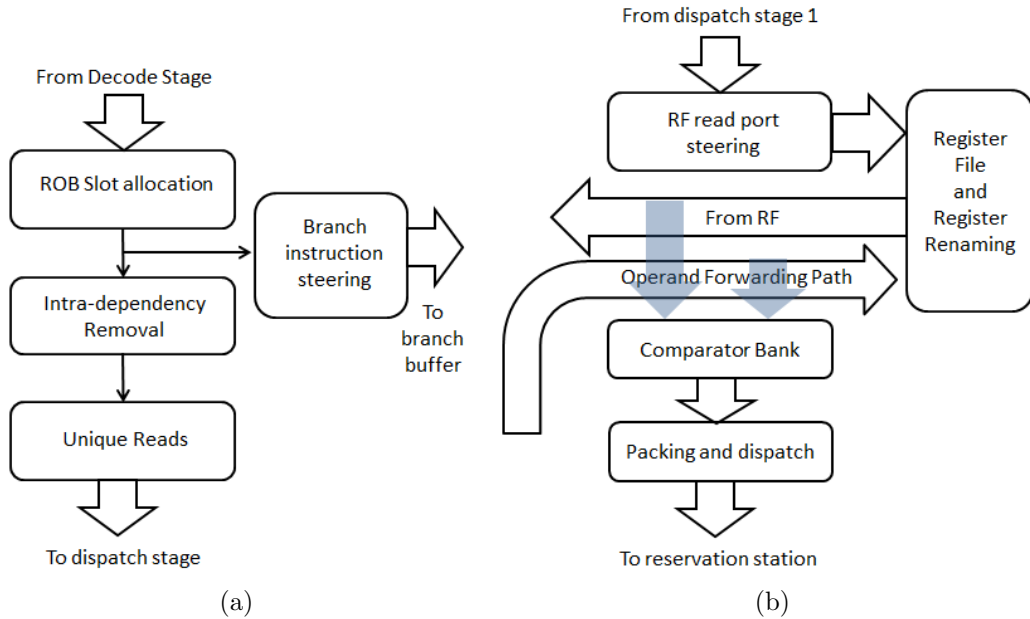


Figure 5.1: Block diagram of (a) stage 1 of instruction dispatch and (2) stage 2 of instruction dispatch

5.2 Dispatch Stage - 1

Stage 1 of the instruction dispatch stage is responsible for ROB entry allocation and pre-processing for register file reading. This stage also does background work of writing the branch target address (BTA), follow through address (FTA) and prediction in to the branch buffer which is addressed by ROB number of the instruction.

5.2.1 ROB Allocation

Once the instructions enter this stage, it is likely sure that they are going to be executed unless there is an exception. The ROB allocation module reserves ROB entries in sequential order so that during writeback, it can be done in program order ensuring precise exception. The allocation needs to be done only for all valid instructions, it is in this stage that all NOP's and invalid instructions due to misalignment are flushed.

The available ROB slots are assigned to the instructions by generating offset to be added to head pointer of ROB. These offsets are generated from a look-up table. Table 5.1 lists down certain cases of offset generation in the allocation truth table.

Instruction Valid Bits	Offset I0	Offset I1	Offset I2	Offset I3	Offset of Updated ROB
0101	00	00	00	01	010
1011	00	00	01	10	011
0111	00	00	01	10	011
1111	00	01	10	11	100

Table 5.1: Offset generation for ROB allocation for some conditions of valid instructions in a cluster

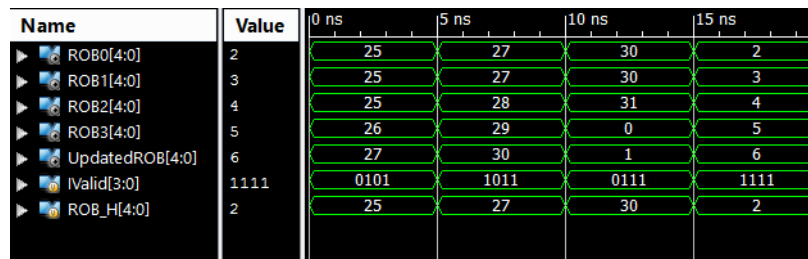


Figure 5.2: Simulation of ROB allocation for the above cases

The current design does not allocate slots to instruction cluster, if the ROB is full. The design does not have provision for partial allocation. In case of an external stall, care is taken not to allocate slots more than once. Fig. 5.2 shows the xilinx simulation for the above cases. The circular buffer feature of the ROB is also taken care here.

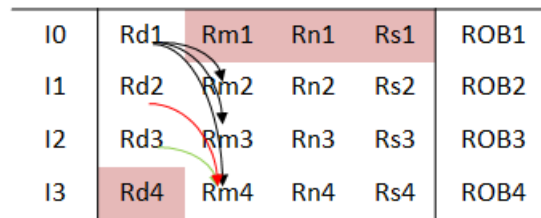
Once ROB entry allocation is done, conventional processors proceed for operand fetch with register renaming, but in this design we perform two preprocessing steps to reduce the load on the register file. They are Intra-dependency removal and unique reads computation which are explained in the following sections.

5.2.2 Intra-dependency

Maximum level of instruction level parallelism is not being achieved because of data and name dependencies between instructions. In an out of order processor where instructions can be executed in any order, these dependencies will limit the throughput. These dependencies were resolved by dynamically renaming the registers.

The current design has register renaming based on ROB to remove the name dependencies. Every operand needs to access the register file to check if the valid data is present or not. If data is not present, the respective ROB number is returned. We exploit this concept to reduce the structural hazard.

Fig. 5.3a lists the 4 instructions in a cluster along with the assigned ROB number as well as the source and destination operands. The idea is to remove the data dependencies inside this cluster itself. As the ROB number of the instructions are present, this can be accomplished by comparing the destination operand of I_n instruction with source operand of the $I_{(n+1)}$ instruction. If there is a match, then the



I0	Rd1	Rm1	Rn1	Rs1	ROB1
I1	Rd2	Rm2	Rn2	Rs2	ROB2
I2	Rd3	Rm3	Rn3	Rs3	ROB3
I3	Rd4	Rm4	Rn4	Rs4	ROB4

(a)

Figure 5.3: Comparison algorithm among the instructions in a cluster

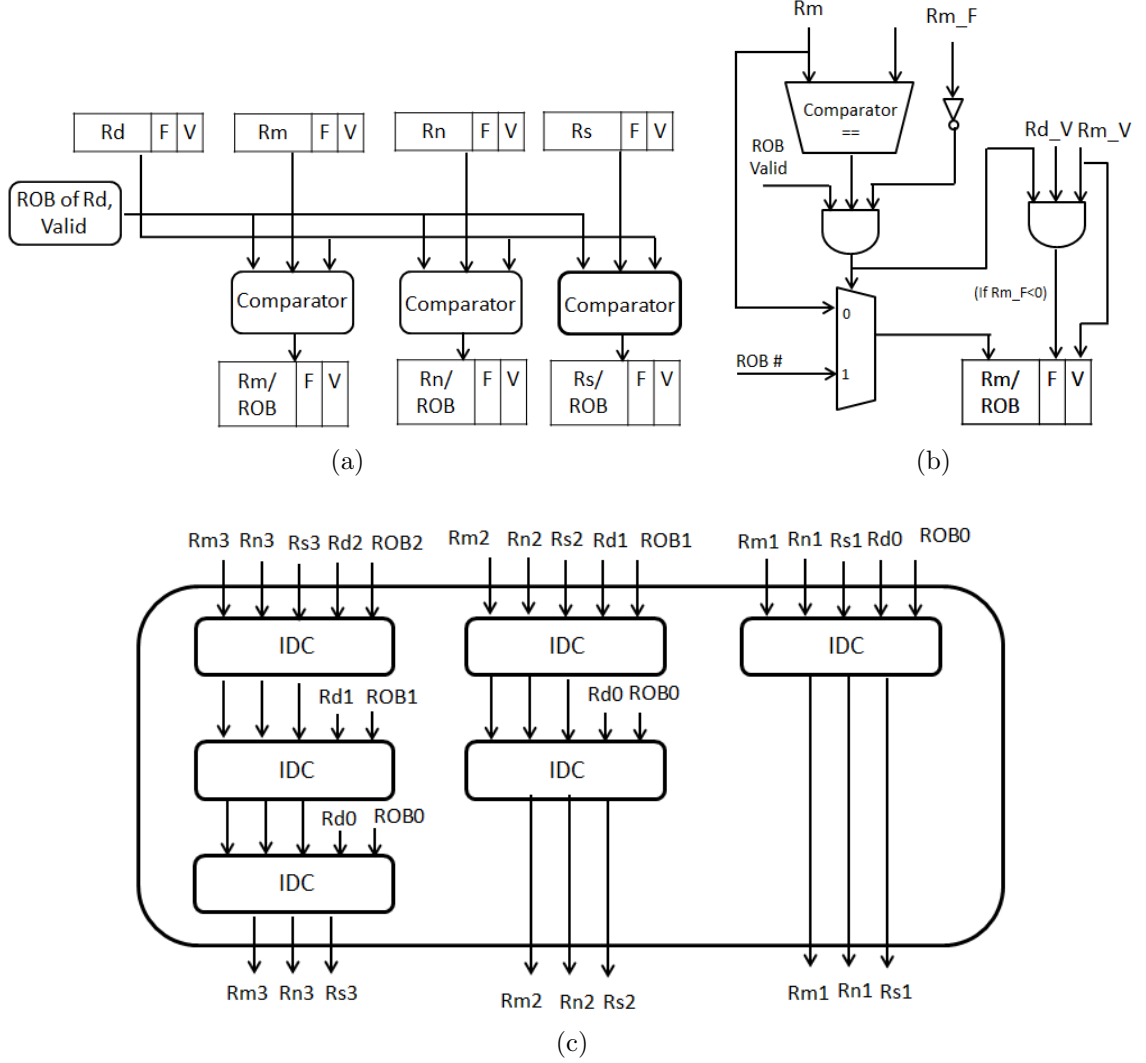


Figure 5.4: (a) Internal block diagram of intra-dependency checker (IDC), (b) Internal block diagram of a comparator used in IDC and (c) A three level architecture to perform intra-dependency removal among the instruction cluster.

operand is marked fetched.

The comparison needs to be done in-program order and hence requires 3 stages of comparison as shown in Fig. 5.4c. The number of stages depends on the number of instructions present in every cluster or the instruction fetch width. The internals of intra dependency checker (IDC) and modified comparator design are shown in Fig. 5.4a and Fig. 5.4b respectively. The comparator design needs to incorporate

parameters like I_n instruction must be valid and the destination operand must be valid as well as validity of $I_{(n+1)}$ instruction and its source operands.

Intuitively this concept would work because ARM being a RISC ISA, almost all instructions depend on register file values. So, while compiling the high level code, the compiler uses registers to load data from memory or store results and immediately uses them for further operations because we cannot have the register file storing the data idle for long time, since the number of available register file slots are limited.

5.2.3 Unique Reads

The intra-dependency removal will get rid of all data dependencies present in the cluster. Another optimization technique incorporated in the design is to perform only unique reads, i.e., when there is a request of performing reads of the same register, then read it only once. The output of this module will give a 15 bit value containing at max 12 bits set in the register. The bits set indicate the requirement of those register values from the register file, after both intra-dependency removal and unique reads. This 15 bit register is sent to stage 2 of dispatch for operand read.

The module takes in to account if the instruction is valid and the respective source operand is valid and not fetched. Fig. 5.5 illustrates an example of unique read module functionality. In the above example, there are 7 operands that need to be read after intra-dependency, if unique reads functionality was not implemented,

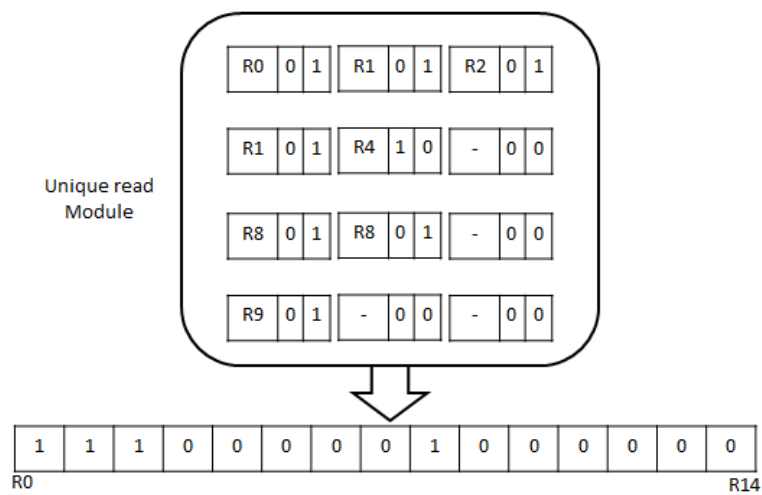


Figure 5.5: Example illustrating the functionality of unique reads module

it would need 2 cycles to read with 4 read ports. The unique reads technique would accomplish it in 1 cycle.

5.2.4 Branch Buffer

Once ROB entries are allocated to valid instructions, they are subjected to the pre-processing techniques described above. In parallel to the pre-processing techniques, the valid branch instructions are routed to a branch buffer, which is addressed by ROB number of the branch instruction to store its program counter, branch target address, follow through address and prediction, which are calculated in stage 1 of instruction decode.

The branch buffer designed is a two way associative 8 entry buffer. This buffer is written by stage 1 of dispatch and read by branch unit and ROB. A high level interface diagram is shown in Fig.5.6. When a write request is received from dispatch stage, a slot is allotted based on the availability in a particular way. First priority is given to way 0. Similarly, the slot is cleared after write-back of that particular branch instruction. The entire branch buffer busy bits are cleared when an exception occurs.

This design ensures that branch buffer contains entries of only inflight - backend branch instructions, which will reduce the requirement of larger buffers to store data, as they are addressed by the ROB numbers of the branch instructions which are

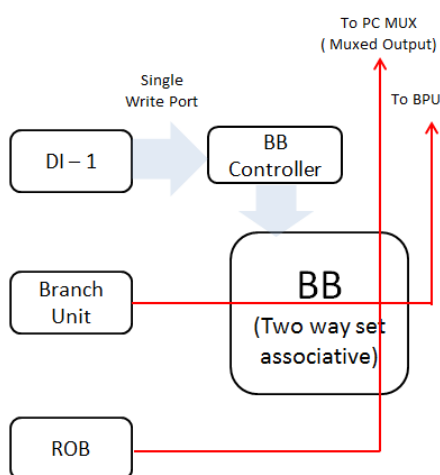


Figure 5.6: Overview of interface of the branch buffer with other stages in pipeline

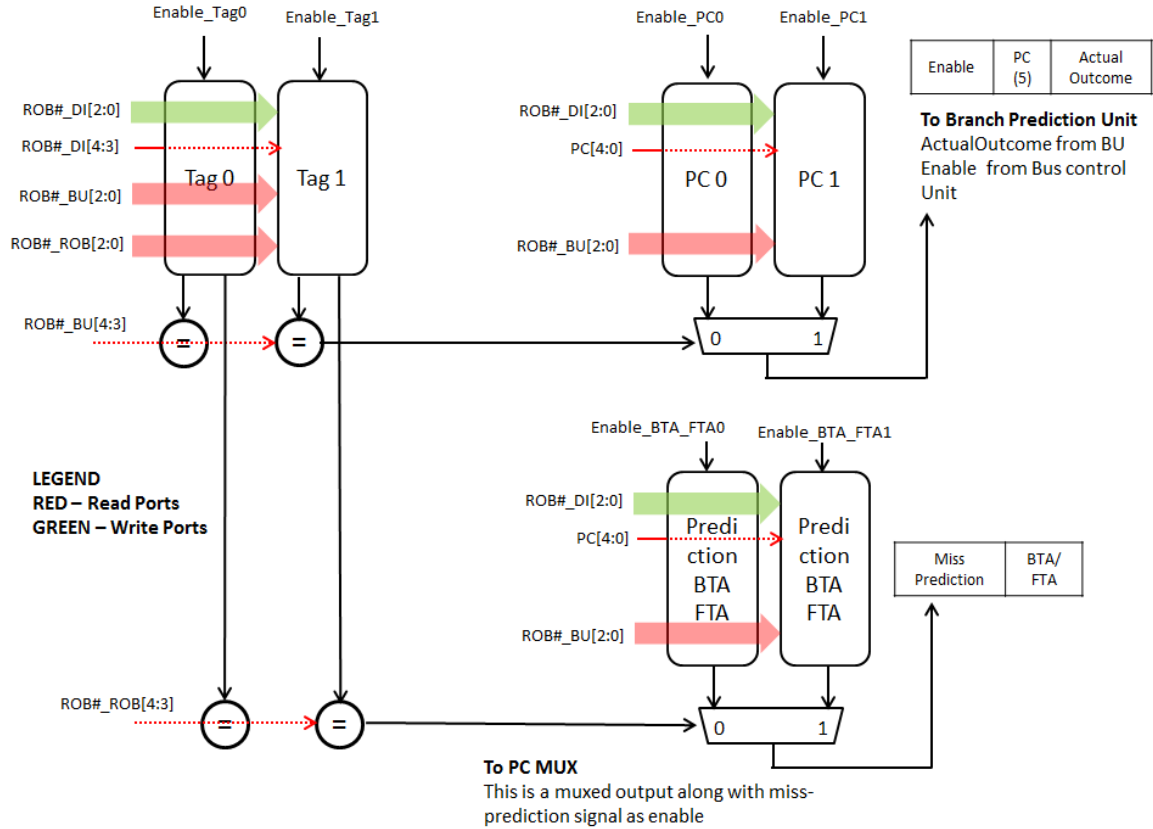


Figure 5.7: Internal architecture of the branch buffer describing the different parts of the branch buffer and the limited access rights to different stages

limited by the size of the ROB. The size of the branch buffer largely depends on the number of stages post dispatch and also how long it takes to clear dependencies of instructions.

The branch buffer is a single ported design, that leads to certain structural hazards if there are multiple valid branch instructions in the cluster. A stall is generated in stage 1 of dispatch, this case is similar to instruction decode stage 1 where the branch predictor has only one port. Another possible stall would be the lack of availability of slots in the branch buffer for writing, the instruction needs to be stalled because overwriting is not allowed in the buffer.

5.3 Dispatch Stage - 2

The main functions in stage 2 of dispatch are operand read and dispatch of instructions to their respective reservation stations.

5.3.1 The $O(1)$ complexity problem

Conventional processor designs directly route the register read requests to the read ports. In the current design, that would lead to a structural hazard for three cycles. The pre-processing techniques in stage 1 of instruction dispatch generate a 15 bit register value where bits are set to indicate the need for reading the respective registers.

A 100% implementation would be to read only those registers which are needed, but that poses a $O(1)$ complexity problem to read 4 unique registers. The hardware complexity would be to generate a 10^{15} size LUT or to have a cascade of 4 pairs of 15 bit priority encoder and decoder pairs. This would lose the purpose of optimization performed in Dispatch stage 1.

A simplistic approach to solve this problem is that to fetch the register file in regions as shown in Fig. 5.8. Stage 1 of the dispatch will provide the source operand of the leading non-branch instruction. This instruction's source operands by default will not be fetched, as there is no information about earlier clusters. The region in which this source operand is present is read. During the process of reading, the register is updated and then checked if there are any more reads to be made. If there are some set bits, then the region which is to be next fetched is computed. This is repeated until all the required registers are read.

This technique requires way less hardware, but the only negative part is that complete efficiency cannot be extracted using this. A worst case of 4 structural stalls might occur using this technique.

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14
----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----

Figure 5.8: Register file divided in to regions

5.3.2 Register file design

The register file has 15 general purpose registers and the 16th register is the program counter. The RF follows the conventional design and has 4 read ports (Fig. 5.9 shows the assignment of read ports) and 4 write ports. The register file has valid bits along with a mapping table which addresses to ROB as a part of the renaming mechanism.

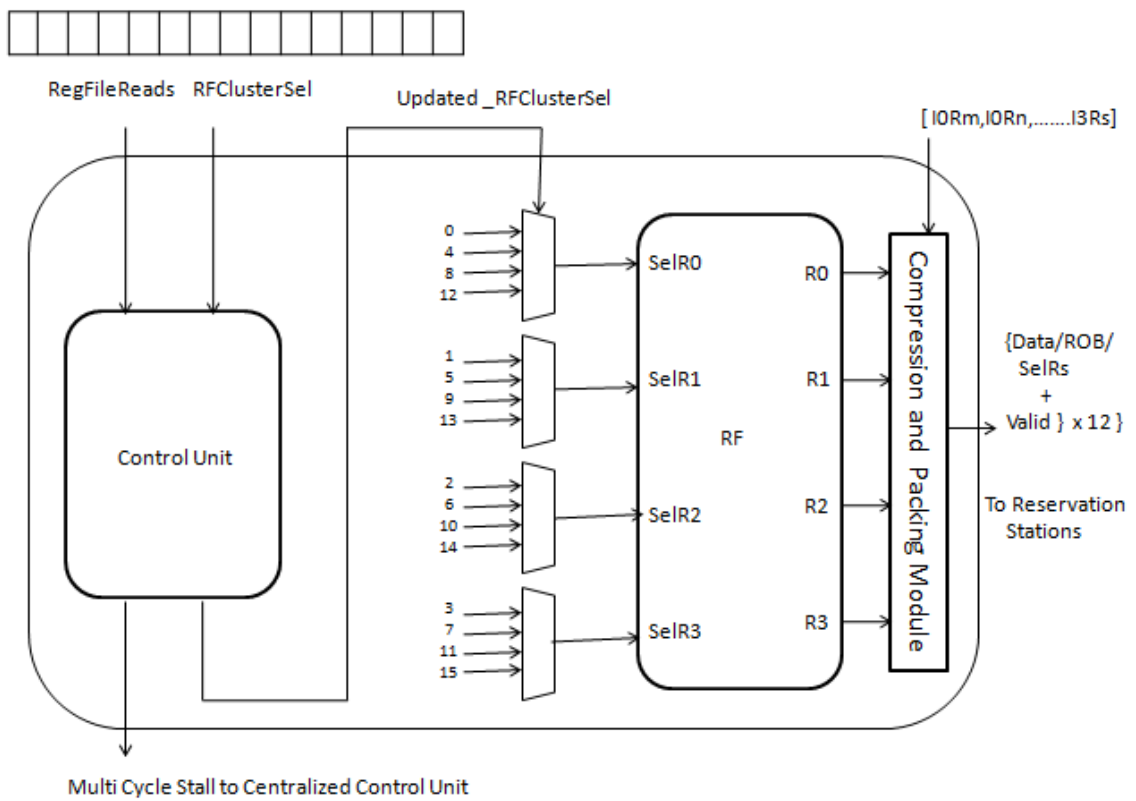


Figure 5.9: Register file read ports allocation

5.3.3 Register Renaming

In an out of order processor the instructions of a program execute in an order which is different from the program order that is generated by the compiler. The instructions are re-ordered to extract higher instruction level parallelism, but it is constrained by the dependencies among instructions. A dependency between instructions will

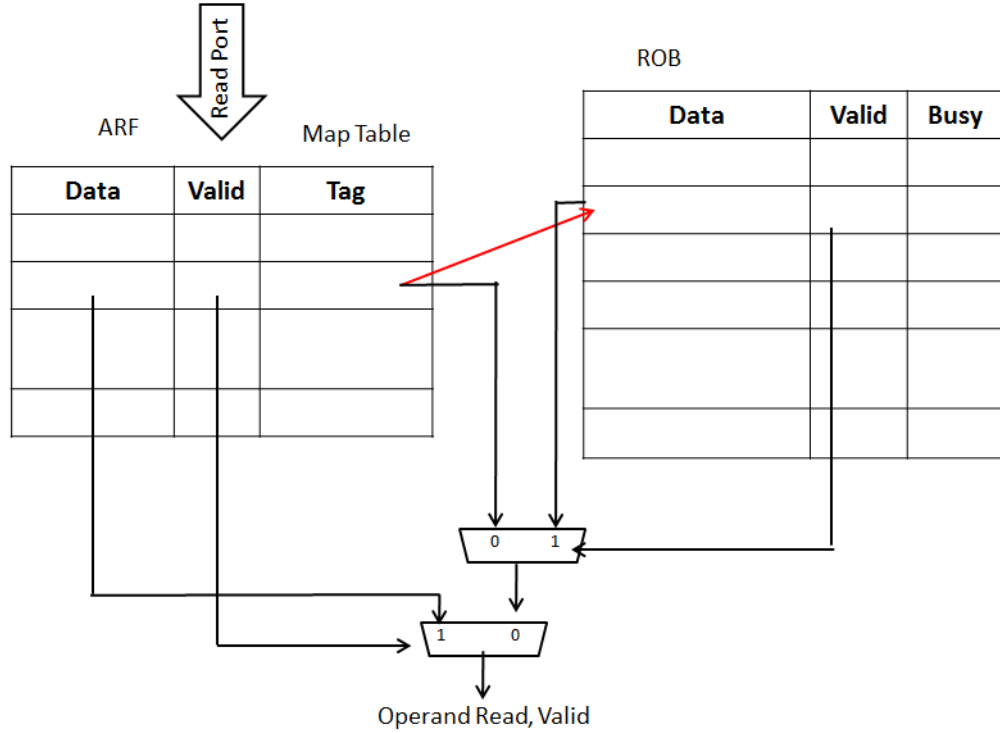


Figure 5.10: Register renaming mechanism through re-order buffer

govern the order of instructions. Dependencies can be of two types: Data and name dependencies, the former occurs when an instruction produces a data element that is consumed by another instruction. This will require the producer instruction to execute before consumer instruction.

Name dependencies are due to the shortage of registers in the register file. This leads to reuse of the same register which causes name dependencies. These are dynamically resolved by register renaming technique. There are different techniques proposed; renaming through ROB, renaming through physical register file and renaming through merged register file [24].

The current design implements an ROB based renaming technique. The ROB ensures in-order write-back and also serves as a renaming mechanism. Fig. 5.10 illustrates the block diagram of the renaming mechanism. The register file holds an additional tag called the map table that indicates for each architectural register whether its latest definition is in the ROB or the architectural register file. The ROB stores the results of all non-committed instructions, whereas architectural register file

stores the latest committed value for each architectural register. In order to facilitate the access to operands in the ROB, the map table also contains an additional field that indicates the location in the ROB where the operand is currently present.

When the instruction commits the data is written from ROB to register file. The map table is updated when ever an instruction changes the value of a register and also it is updated during write-back.

5.3.4 Register file data bus

This processor design has a register file data bus, and instructions in dispatch read data from both the bus and operand forwarding path together. This is the additional piece of hardware needed when compared to the conventional register file reading mechanisms. This additional comparators bank is needed because of the pre-processing techniques proposed that will lead to only unique reads and the register file read will not have any relation to the instruction. This drawback also puts a limitation on the number of read ports the design can support.

5.3.5 Reservation slot Allocation

This is the last step of allocation where the instructions are allocated an entry in their respective reservation station. A stall is generated if the reservation stations are full. This allocation is done based on the tag the instructions carry which indicates to which functional unit the instruction must be routed. This tag is generated in decode stage.

5.3.6 Dependency matrix

A dependency matrix is generated similar to dependency graphs for compilers. This matrix has as many number of entries as the re-order buffer each 1 bit wide. These bits are set if that result of the instruction, in that ROB entry is needed by a later instruction. This matrix is used to decide which results to send in the operand forwarding path when there is a bus contention, since we will have to first forward the results that are most awaited in the reservation station. Another option would be to send the values of operands whose instructions are close to the instructions waiting for write-back.

Chapter 6

Reservation Station

The data-dependencies between instructions will lead to stalls in dispatch stage, if there are no buffers where the instructions can wait. Reservation stations act as virtual execution units, where the dispatch stage is given the freedom to dispatch the instructions, even if the sources operands are not ready. The reservation stations can be shared or private, based on the design requirement. The reservation station is better utilized in-case of shared when compared to private as it depends on the type of instructions in the program that determines its usage. The instructions wait in the reservation station until they get the source operands from the common data bus. The issuing logic in the reservation station will introduce out-of-order nature in to the processor ([16, 17, 18]). The backend of the processor begins from the

Reservation Station	Sizes/Slots
Shared Integer Unit	8
Shifter Unit	2
MAC Unit	2
Load/Store Unit	4
Branch Unit	2

Table 6.1: Number of entries in each reservation station.

reservation station.

The current design has separate reservation stations for each of its functional units. There are two integer functional units that share the same reservation station, the rest of the functional units have private reservation stations (Table 6.1). The number of read ports for a reservation station depends on the number of functional units connected to it and number of write-ports depend on the probability of that type of instruction occurring in an instruction cluster. The number of entries in each of the reservation station plays a crucial role in the overall performance of the processor. The number of reservation station slots directly impacts:

- The number of comparators needed to snoop and read data from the common data bus increases with increase in number of reservation station entries. The number of comparators used in this design are:
 1. Shared integer unit reservation station : $(3 \times 8 \times 4 = 96)$, where 3 indicates the number of dependent variables, 8 indicates the number of slots and 4 indicates the number of operand forwarding buses.
 2. Shifted integer unit reservation station : $(4 \times 2 \times 4 = 32)$
 3. MAC unit reservation station : $(3 \times 2 \times 4 = 24)$
 4. Load-store unit reservation station : $(2 \times 4 \times 4 = 32)$
 5. Branch unit reservation station : $(1 \times 2 \times 4 = 8)$

A total of 192 5-bit comparators are needed for the reservation stations to snoop data from the common data bus.

- The size of re-order buffer is also directly related to number of reservation station entries because, re-order buffer must have the capacity to hold all the in-flight instructions.
- With the increasing size of the reservation station, the complexity of issue logic also increases.

The parameters that impact the sizes of the reservation station are probabilities of different instructions in a program. More number of slots are provided for instructions that occur most frequently because there must not be any stalls in the dispatch stage.

Upon theoretical calculations, the number of slots for each reservation station is listed in table 6.1.

The dispatch stage will stall, if any of the reservation station is full and unable to issue a new entry for an instruction or if there is a requirement more than the number of write ports of that respective reservation station. The number of entries of ROB is considered to be 32, based on the analysis of number of in-flight instructions. There can be 4 instructions in the decode stage, 18 instructions in reservation stations, 9 instructions in all stages of the functional units. This totals upto 31 in-flight instructions, hence the ROB size is fixed to be 32.

A matlab based back-end simulator was designed to determine the sizes of the reservation stations and ROB (appendix B). The simulation results also show promising results for this configuration.

6.1 Issue logic

Issue logic introduces out-of-order feature in the design, allowing an independent later instruction to execute before the dependent instruction. The issue logic decides which instruction to be issued based on which instruction is ready. This is a fairly simple implementation and fig. 6.1 shows the simulation of issue logic for an 8 entry shared integer unit reservation station. There are many proposals that improve the issue logic, like considering the age of the instruction using a counter or checking the which ready instruction is close to the ROB_tail pointer and issue them first. These improvements would need a lot of hardware support but will provide better results.

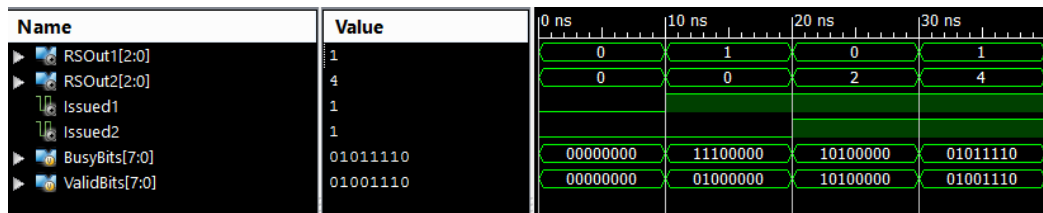


Figure 6.1: Simulations of a simple out-of-order issue logic implemented in this design

Chapter 7

Execution Stage

Execution Stage plays a crucial role in computing the results of a given program. There are several types of operations, an execution stage can perform in a given pro-

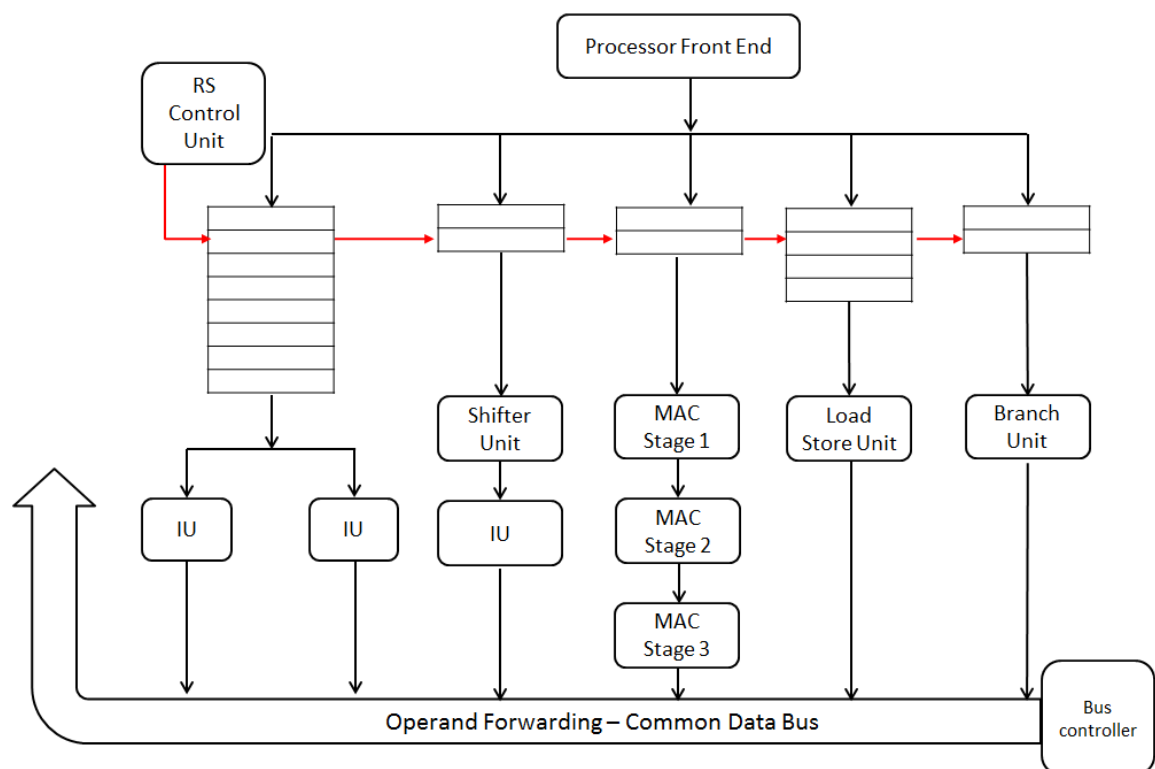


Figure 7.1: Block diagram of execution unit containing all the functional units and reservation stations

with the appropriate control signals needed for the required operation.

The operations in this unit include Negation (NOT), Logical (AND, EOR, OR) and Arithmetic (ADD, SUB) which are shown in fig 7.2. Additional features to perform reverse subtract (RSUB), addition and subtraction with carry (ADC, SBC - These instructions need the value of the latest carry flag) and Saturation addition and subtraction (QADD, QSUB) are also provided. Depending on the control signal, one among the above specified operations is performed and the result along with flags are generated. This stage does not support any shift based instructions. This unit completes its execution in 1 clock cycle, upon synthesis, results show a combinational delay of 7.755 ns.

7.2 Shifted Integer Unit

In ARM V7 ISA, there are certain instructions, where one of the 2 operands needs to be shifted by an amount, before performing the required operation. This unit

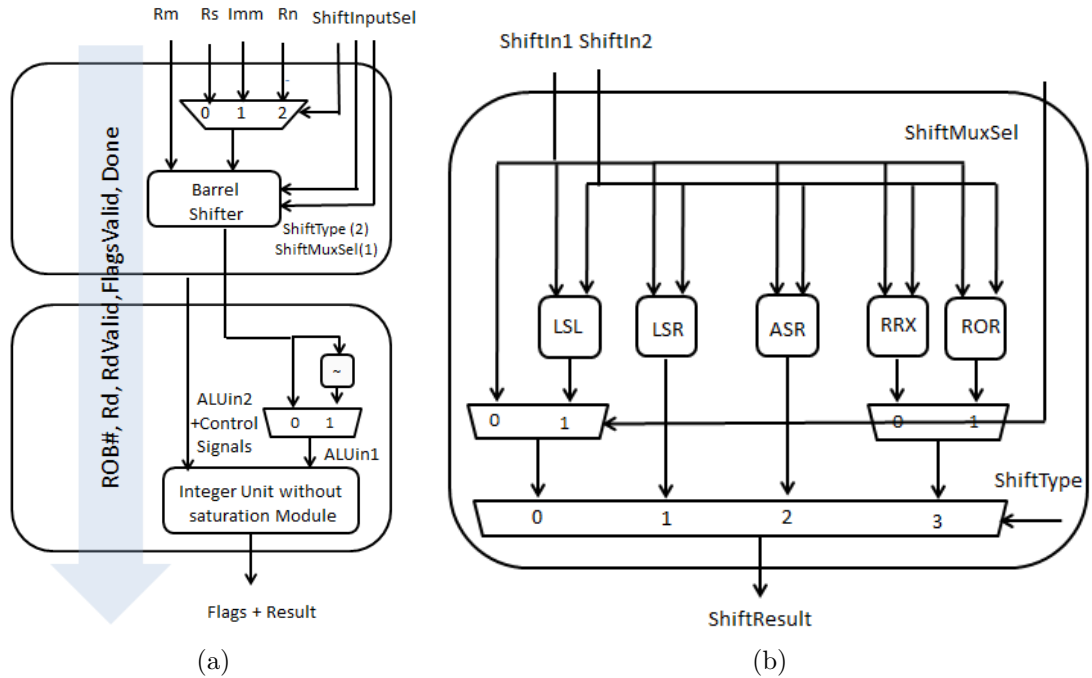


Figure 7.3: (a) Block diagram of the 2 staged shifted integer unit and (b) architecture of stage 2 of shifted integer unit

is similar to integer unit, but has an additional shifter unit before the integer unit. As a result, only 1 operand along with the shift value (either stored in a register or an immediate value based on ShiftInputSel) is sent to ShiftedIntegerUnitStage1. Depending on the control signals such as ShiftMuxSel and ShiftType, one among the shift operations such as logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR), rotate right with extend (RRX) and rotate right (ROR) is performed as in fig 7.3b.

Figure 7.3a shows that, in the second stage of shifted integer unit, the shifted result and the third operand of the instruction are subjected to an arithmetic operation based on the control signal. This stage architecture is replicated as in integer unit and generates the required result depending on the control signals, along with the flags. The only modification to this unit is that, there is no saturation unit support. The combinational path delays of 2 stages are 10.642 ns and 9.335 ns.

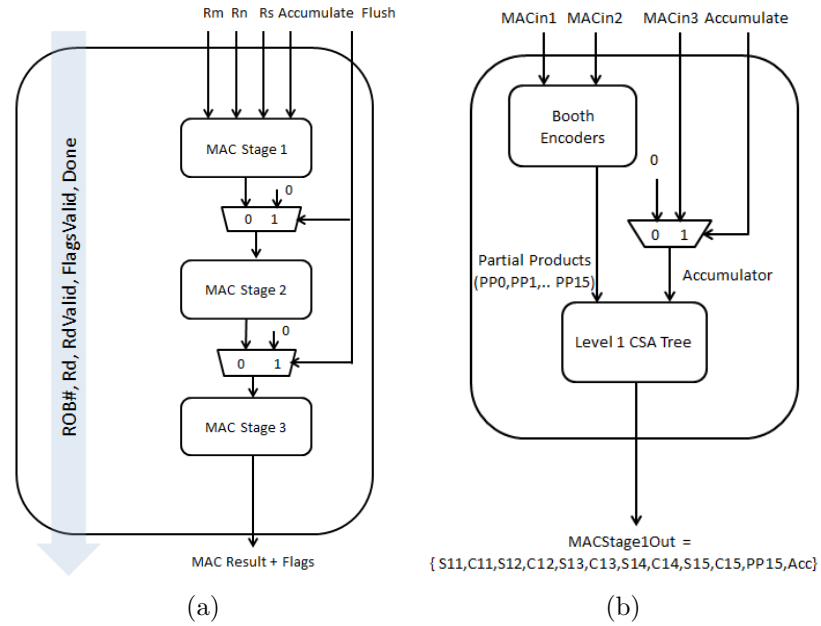


Figure 7.4: (a) High level block diagram of the 3 stage MAC unit, (b) partial products generation and level 1 CSA in stage 1

7.3 MAC Unit

Integer multiplication is the most complicated function in the current design. This instruction is not supported by the ALU, because of its high complexity and area cost. Also the probability of integer instructions is very high. So having a multiplication unit in IU will degrade the performance of the processor. This led to a separate MAC unit in the processor.

MAC unit is used by two instructions MUL and MLA which use 2 and 3 source operands respectively. The complexity of the operation led to division of MAC Unit into 3 stages that can perform either multiplication or multiply and accumulate operation depending on the control signal (Fig. 7.4a).

A three bit booth recoding technique is used to generate 16 partial products (PP0, PP1 ... PP15) as shown in fig. 7.4b [19, 28]. The architecture follows wallace tree approach to reduce time complexity. These partial products along with the accumulator are added using stages of carry save adders (CSA) as shown in Fig. 7.5a. This design is also called as tree based multiplier and the CSA are the 3-2 reducers. Table 7.1 lists down the number of carry save adders used in each level of the tree multiplier.

The stages are carefully spaced across the 3 stages to ensure cycle time. The final two operands are added in carry look ahead fashion. Upon synthesis, each of the MAC stage takes a combinational path delay of 7.266 ns, 7.020 ns and 7.104 ns respectively.

CSA Level	No. of summands	No. of groups	No. of remaining summands
1	17	5	2
2	12	4	0
3	8	2	2
4	6	2	0
5	4	1	1
6	3	1	0

Table 7.1: Number of carry save adders in each of the CSA level

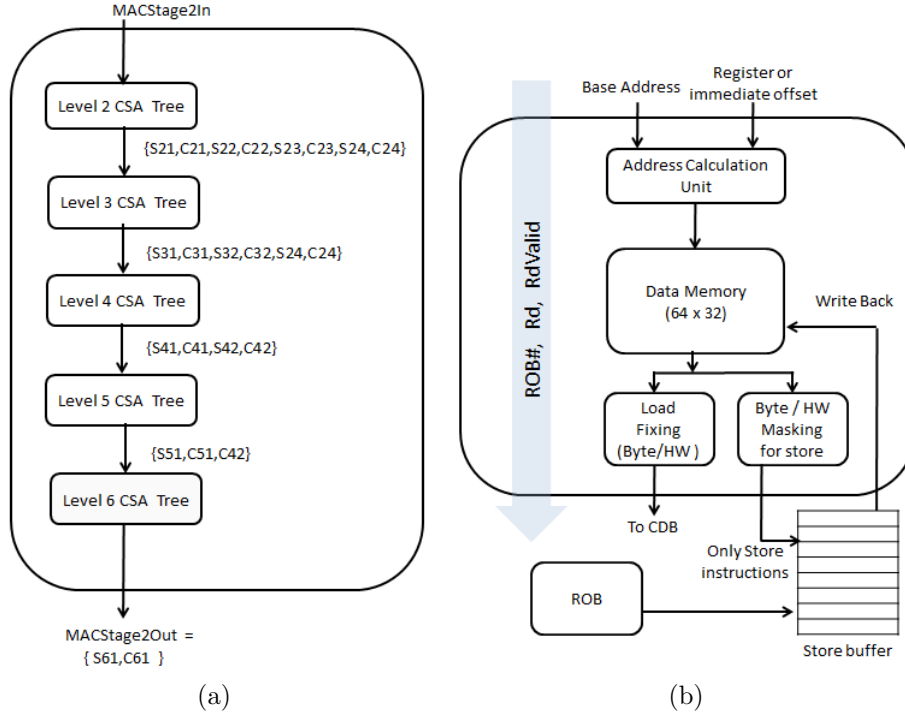


Figure 7.5: (a) block diagram of stage 2 of MAC indicating all levels of CSA and (b) Load and store stage and its interconnection with the store buffer

7.4 Load and Store Unit

This unit performs all data memory related operations (load and store). The ISA supports variants of load store instructions such as load word, load half word, load byte with either zero or sign extension, store word, store half word and store byte. The design of the load and store unit is shown in fig. 7.5b. This design does not employ a caching system rather has a direct memory.

7.4.1 Data Memory design

As the ISA supports different types of load and store instructions of different widths, memory design becomes a crucial part. Distributed RAM IP from xilinx is used to synthesize data memory. There are two design aspects:

- **A byte addressed RAM:** This memory would require 4 read ports or a structural hazard of 4 cycles when reading a word. A structural hazard would

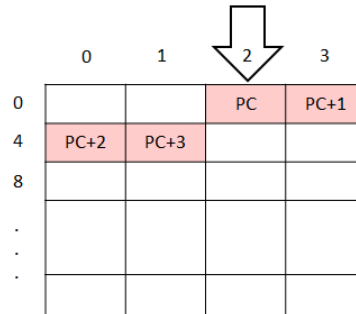


Figure 7.6: Example illustrating misalignment in data memory

need lot of hardware to store bytes of data and also maintain the order.

- **A word addressed RAM:** This memory would require a single read port and single write port. If a byte or half word needs to be read, then the entire word is read. A control logic will select the correct data that needs to be sent. The current architecture implements a word addressed RAM. There are certain limitations to this design which have been overlooked.

Alignment Issues

Alignment is a huge limitation in this design, for simplicity it has been assumed that the data request that is received from the processor is always aligned. In case of a misalignment during read request, the design must support reading the memory twice. This same problem holds during store operation. Fig.7.6 illustrates an example of misalignment while reading a word from a non aligned location.

7.4.2 Store Buffer

To ensure inorder write back of instructions, separate store buffer is maintained to store data, address along with write enable control (WE) before writing into data memory. This is similar to ROB entry for other instructions, a separate buffer is maintained because having so much space for all instructions in ROB will incur a lot of area requirement. The store buffer is a direct ROB addressed cache structure. Data is written from the store buffer to data memory when the respective instruction comes for write-back.

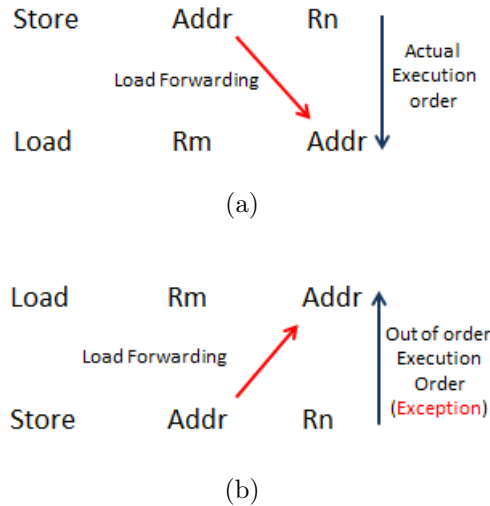


Figure 7.7: Load forwarding cases (a) without exception and (b) with exception

Load instructions forward data from the store buffer if there is an address match. This is also known as load forwarding. This might also lead to exception (Fig. 7.7a and 7.7b) during the write-back of load. Fig. 7.5b explains the integration of the store buffer with the load-store unit.

7.5 Branch Unit

Branch unit is responsible to execute conditional branch instructions, as unconditional branch instructions are flushed in decode stage itself. Conditional branch instructions are validated based on the current status of the program status register and the condition on the branch instruction. Table 7.2 lists down all the conditional branches available in ARMv7 ISA.

The only dependency on the inputs of branch instruction are the flags. Once the actual outcome of the branch is computed, it needs to update the branch prediction unit. This is done by accessing the branch buffer to get PC and it is routed to the branch prediction unit. The branch unit does not write to the operand forwarding path rather it directly updates the misprediction bit in the ROB.

Condition	Mnemonic + Meaning	Conditional Flags
0000	EQ - Equal	$Z = 1$
0001	NE - Not Equal	$Z = 0$
0010	CS - Carry Select	$C = 1$
0011	CC - Carry Clear	$C = 0$
0100	MI - Minus and Negative	$N = 1$
0101	PL - Plus or positive or zero	$N = 0$
0110	VS - Overflow	$V = 1$
0111	VC - No Overflow	$V = 0$
1000	HI - Unsigned Higher	$C = 1 \ \& \ Z = 0$
1001	LS - Unsigned Lower or Equal	$C = 0 \text{ --- } Z = 1$
1010	GE - Signed greater or equal	$N = V$
1011	LT - Signed less than	$N \neq V$
1100	GT - Signed greater than	$Z = 0 \ \& \ N = V$
1101	LE - Signed less than or equal	$Z = 1 \text{ --- } N \neq V$
1110	Unconditional	No Condition

Table 7.2: Conditional codes of ARMv7 for branch instructions

7.6 Common Data Bus (CDB)

In a deeply pipelined machine, to improve the performance, dependent instructions are given provision to execute speculatively by reading the source operands from the non-committed instructions through operand forwarding paths or common data bus. The results are forwarded to reservation stations where the dependent instructions wait for source operands.

The functional units write to the operand forwarding path synchronously as they write in to Re-order buffer. The design incorporates a maximum of 4 writes to re-order buffer and operand forwarding path from the functional units. The number of forwarding paths directly affect the number of comparators needed at the reservation stations affecting the cycle time.

7.6.1 CDB Arbiter

A common data bus arbiter decides which among the 6 functional units should write to the CDB. The CDB arbiter assigns slots to functional units based on the done signals generated from functional units and the dependency matrix. For example if 5 functional units complete their operation and only 4 of its outputs are needed by the instructions waiting in the reservation station, then priority is given to those 4 units. CDB arbiter also differentiates between load and store instructions; as store instructions write data only to the store buffer and updates the done bit in the ROB.

7.7 Program Status Register (PSR)

The program status register holds the copies of the arithmetic logic unit status flags. This will determine whether conditional instructions are to be executed or not. The ISA specifies it to be a 32 bit register with only 9 bits used as flags, the remaining bits are reserved for future purposes. This architecture implements only 5 bits among these:

- Negative Flag (N) : The MSB bit of the result is set as negative flag, if the result is regarded as a signed integer.
- Zero Flag (Z) : This bit is set if the result is 0.
- Carry Flag (C) : Set to 1 if an instruction generates a carry.
- Overflow Flag (V) : Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.
- Saturation Flag (Q) : Set to 1 to indicate overflow or saturation occurred in case of saturation based instructions,

Chapter 8

The commit stage

The current day high performance processor designs execute instructions out-of order to extract maximum instruction level parallelism. If the instructions directly write to the architectural register file after execution then maintaining the validity of data becomes extremely complex. The processors incorporate an additional stage called commit where the instructions that are executed out-of order are ensured to write-back in-order. A re-order buffer is implemented to ensure in-order write-back, this also ensures precise exception.

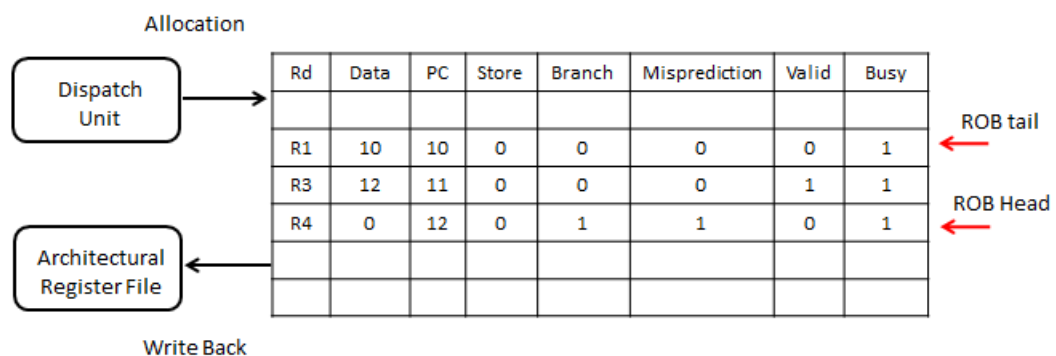


Figure 8.1: Re-order buffer high level interface block diagram

8.1 Re-Order Buffer

A re-order buffer is a circular buffer that contains all the in-flight instructions. This has multiple functionality in the design: used as free register file for register renaming and also ensures sequential write-back. The buffer has two pointers; the head pointer that points to the next free slot in the ROB and the tail pointer points to the leading instruction that is waiting for write-back. The number of slots allocated in each cycle depends on the allocation bandwidth of the dispatch stage and the number of slots freed every cycle depends on the number of instructions that are ready for write-back and the write-back bandwidth.

The size and the design of the re-order buffer plays a crucial role in the performance of the processor. [20, 21] have proposed low complexity and low power designs as ROB is a crucial part of any out-of-order processor that has an in-order commit. The ROB must be able to hold all the in-flight instructions. In the current design, the ROB size of 32 is chosen as the design can have 4 instructions in the dispatch stage, 18 instructions waiting in the reservation stations and 9 instructions in the execution unit. This means a maximum of 31 in-flight instructions excluding the ones waiting in the ROB for writeback, hence a ROB size of 32 is a decent option. A matlab based back simulator was designed to understand the impact of varying reservation station and ROB sizes. Appendix B deals with the details of the simulator and parameters

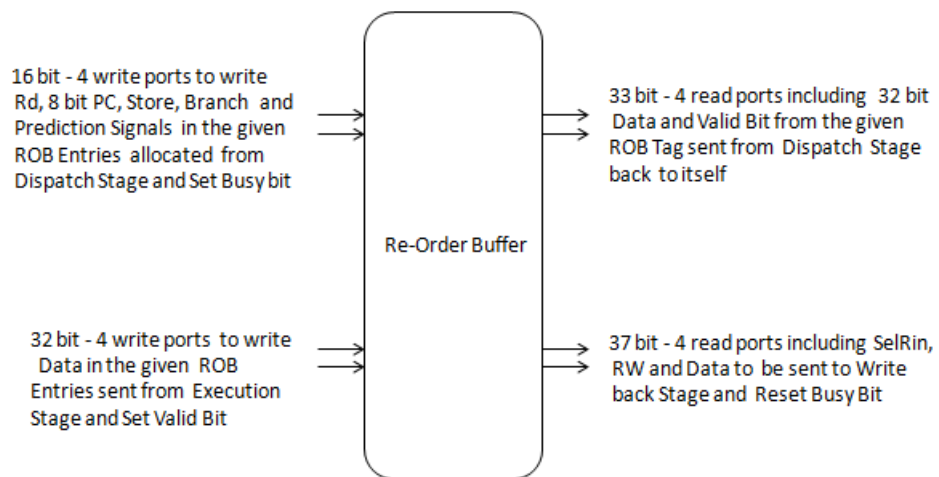


Figure 8.2: ROB port requirements

considered for performance evaluation.

Fig.8.1 briefs the interface between ROB and different stages in the pipeline and the various entries in the buffer. Separate store and branch buffers are used as re-order buffers for store and branch instructions. ROB will reserve a slot for branch and store instructions but the data will be stored in their respective buffers. Fig. 8.2 lists down the different read ports and their purposes in the design.

8.2 Write-back

Write back or commit [22] happens when the instruction pointed by ROB tail pointer is ready for write-back in to architectural register file i.e., all the instructions prior to this instruction are written back and the result for this instruction is ready. The write-back bandwidth is governed by the number of write ports in a register file.

The write-back system should ensure that there is no WAW dependency and the order of instructions write-back is maintained. The system must be capable of checking the maximum write-backs that are possible within the writeback bandwidth.

8.3 Exception handling

Exceptions are usually handled at commit time. This is because, the commit stage ensures in-order write-back, which means that all instructions before the instruction that triggered the exception are written back. There are two possible exceptions, one due to branch misprediction and the other due to load forwarding error.

When an exception is encountered, the entire pipeline is flushed and the front end of the processor is redirected to start fetching instructions from the exception handler. All the buffers are cleared including reservation stations and re-order buffer.

Chapter 9

Centralized Control Unit

The centralized control unit is the heart of the processor. It is responsible for the flow of data through the entire pipeline. This unit monitors signals from all the stages and assigns priority to situations based on the signals that it receives, for example if there is stall in the dispatch stage due to non-availability of slots in the reservation station and also if an exception arises from the write back stage due to a branch misprediction, then the control unit gives priority to the exception. The control unit generates the stall and flush signals based on the situation. Different type of stalls:

- Stage 1 of instruction decode : Multicycle stall due to routing of branch instruction
- Stage 1 of instruction dispatch : ROB allocation stall and multicycle stall due to routing of branch instruction
- Stage 2 of instruction dispatch: Multicycle stall due to register file structural hazard and reservation station stall
- Execution unit : Common data bus allocation stall
- Exception : Load forwarding and branch misprediction

The implementation aspect of the control unit is similar to priority encoder. Table 9.1 lists the priorities among different stalls. Another important role played by the control unit is to select the next fetch address among the several options available. The next fetch address is computed speculatively, one address comes from the decode

Stall Priority	IF	ID1	ID2	DI1	DI2	Execution
5	0/1	0/1	0/1	0/1	0/1	0/1
4	0/0	0/0	0/0	0/0	0/0	Respective 1/0
3	1/0	1/0	1/0	1/0	1/0	0/0
2	1/0	1/0	1/0	1/0	0/0	0/0
1	1/0	1/0	0/0	0/0	0/0	0/0

Table 9.1: Priorities of stall signals in centralized control unit. Signals generated to different stages are indicated as stall/flush

stage in case of a taken branch instruction and another address comes from the branch buffer in case of a branch misprediction.

Chapter 10

Synthesis results

The design has been synthesized using Xilinx ISE for Virtex 5 xc5vlx110t-1-ff1136. All default settings were used. The design strategy was set to optimize the timing performance with a high effort. Important points in synthesis results of each stage are described below

Instruction Fetch Stage

Timing Summary:

Minimum period: No path found
Minimum input arrival time before clock: 2.498ns
Maximum output required time after clock: 5.007ns
Maximum combinational path delay: 3.918ns

Instruction Decode stage 1 : Branch Prediction

Global predictor has 64 bit register (32×2), Global history register is a 5 bit register, Local level 1 history table has 160 flip flops (32×5) and level 2 local history table has 96 registers (32×3). Selector table has 64 bit register (32×2). Therefore overall flip-flops needed are $64 + 5 + 160 + 96 + 64 = 389$.

HDL Synthesis report:

# Adders/Subtractors	: 2
5-bit adder	: 2

```

# Registers : 389
  Flip-Flops : 389
# Multiplexers : 16
  1-bit 32-to-1 multiplexer : 10
  2-bit 32-to-1 multiplexer : 4
  3-bit 32-to-1 multiplexer : 2
# Xors : 4
  1-bit xor2 : 2
  5-bit xor2 : 2

```

Control Signal

Delay: 5.296ns (Levels of Logic = 5)

```

# BELS : 17
# GND : 1
# LUT2 : 3
# LUT3 : 2
# LUT4 : 1
# LUT5 : 5
# LUT6 : 5

```

Timing Summary:

Minimum period: 4.786ns (Maximum Frequency: 208.958MHz)

Minimum input arrival time before clock: 5.353ns

Maximum output required time after clock: 7.595ns

Maximum combinational path delay: 7.975ns

Instruction Decode stage 2

Stage 2 of an instruction decode is a look-up-table based implementation.

HDL Synthesis report:

```

# ROMs : 4
  4x1-bit ROM : 4
# Multiplexers : 20

```

18-bit 16-to-1 multiplexer	: 12
5-bit 16-to-1 multiplexer	: 8
Maximum combinational path delay	: 6.316ns

Stage 1 of Instruction Dispatch : ROB Allocation

A look-up table to hold the offsets for ROB number allocation based on the instruction valid bits. Adders are to compute the exact address using the head pointer of ROB and the offset.

HDL Synthesis report:

16x9-bit ROM	: 1
5-bit adder	: 5
Delay:	5.563ns (Levels of Logic = 5)

Stage 1 of Instruction Dispatch : Intradependency check between registers, Intradependency between flags and unique reads

HDL Synthesis report:

Maximum combinational path delay:	9.861ns
4-bit comparator equal	: 18

Intra-dependency flags module

Maximum combinational path delay: 4.828ns

Unique Reads

16x15-bit ROM	: 12
Maximum combinational path delay:	5.735ns

Stage 2 of instruction dispatch

The registers indicate the register file, map table and the valid bits field. Multiplexers are for the read ports, comparators are the most important aspect in this design as we need 112 comparators to read data from both register file bus and the operand forwarding bus. Priority encoders are used to give priority from which bus to read.

HDL Synthesis report:

Macro Statistics

# ROMs	: 2
4x15-bit ROM	: 1
4x16-bit ROM	: 1
# Adders/Subtractors	: 12
1-bit adder	: 12
# Registers	: 51
1-bit register	: 17
15-bit register	: 1
2-bit register	: 1
32-bit register	: 16
5-bit register	: 16
# Comparators	: 112
4-bit comparator equal	: 48
5-bit comparator equal	: 64
# Multiplexers	: 12
1-bit 16-to-1 multiplexer	: 4
32-bit 16-to-1 multiplexer	: 4
5-bit 16-to-1 multiplexer	: 4
# Priority Encoders	: 32
5-bit 1-of-9 priority encoder	: 32

Timing Summary:

Minimum period: 2.956ns (Maximum Frequency: 338.278MHz)
Minimum input arrival time before clock: 4.009ns
Maximum output required time after clock: 5.971ns
Maximum combinational path delay: 6.551ns

Branch Buffer

72 registers needed for each slot in a way, requiring $(72 \times 8 \times 2)$ 1152 registers. The multiplexers indicate the different read ports for each way in the buffer.

HDL Synthesis report:

Macro Statistics

# Registers	: 1152
Flip-Flops	: 1152
# Comparators	: 2
2-bit comparator equal	: 2
# Multiplexers	: 10
1-bit 8-to-1 multiplexer	: 6
5-bit 8-to-1 multiplexer	: 2
65-bit 8-to-1 multiplexer	: 2
# Decoders	: 2
1-of-8 decoder	: 2

Timing Summary:

Minimum period: 2.584ns (Maximum Frequency: 387.028MHz)
Minimum input arrival time before clock: 3.222ns
Maximum output required time after clock: 6.180ns
Maximum combinational path delay: 6.859ns

Re-order Buffer There are 1401 registers of ROB which are divided among the different type of registers. Multiplexers indicate the 8 read ports of the ROB.

HDL Synthesis report:

Macro Statistics

# Registers	: 141
1-bit register	: 68
32-bit register	: 36
4-bit register	: 4
5-bit register	: 33
# Multiplexers	: 21
1-bit 32-to-1 multiplexer	: 9
32-bit 32-to-1 multiplexer	: 8

5-bit 32-to-1 multiplexer

: 4

Timing Summary:

Minimum period: 3.034ns (Maximum Frequency: 329.635MHz)

Minimum input arrival time before clock: 4.479ns

Maximum output required time after clock: 4.872ns

Maximum combinational path delay: 5.374ns

Table 10.1 lists down the synthesis results of each stage separately.

Pipeline Stage	Stage Name	Minimum Period (ns)	Maximum operating frequency (MHz)	Maximum Combinational path delay(ns)
IF	Instruction Fetch	-	-	3.92
ID 1	Instruction Decode 1	5.16	193.56	10.70
ID 2	Instruction Decode 2	-	-	6.32
DI 1	Dispatch Instruction 1	1.77	563.88	11.09
DI 2	Dispatch Instruction 2	2.96	338.28	6.55
RS	Reservation Station			8.33
EX 1	Integer Unit	-	-	7.75
EX 2	Shifted Integer Unit 1	-	-	10.64
EX 2	Shifted Integer Unit 2	-	-	9.33
EX 3	MAC Unit 1	-	-	7.27
EX 3	MAC Unit 2	-	-	7.02
EX 3	MAC Unit 3	-	-	7.10
EX 4	Load Store Unit			8.23
EX 5	Branch Unit	-	-	4.99
ROB	Re-order Buffer	3.03	329.63	8.37

Overall minimum period (ns): 12.28 ns

Maximum operating frequency (MHz): 81.43

Table 10.1: Synthesis results of each stage

Chapter 11

Conclusion and Future Work

Conclusion

Processor micro-architecture evolution has been constantly aiming at improving the instruction level parallelism (ILP). Super-scalar architecture provides high performance by using hardware techniques to execute multiple instructions every cycle. This project commenced with the selection of ARM V7 ISA by considering 90 instructions, after analysing all the prominent ISAs.

Initial implementation aimed at single cycle architecture followed by conventional 5 stage pipelined architecture including operand forwarding and branch prediction techniques to avoid all the data and control hazards.

Then we started realizing single threaded superscalar out-of-order design, including simulation of Branch prediction scheme, Reservation station (RS) slots, Re-order buffer (ROB) slots. Implementation of different pipeline stages including fetch, decode, dispatch, issue, execute and write back was done. The maximum operating frequencies of each of the stages are noted and appropriate optimization were done for efficiency improvement, including pre-processing logic in dispatch stage, pipelining the execution unit etc.

Finally, series of test benches were generated using a self developed assembler and these architectures that were implemented, were run through them and performance comparison was done. All these architectures were synthesized on Virtex 5 FPGA to understand the hardware requirements and clock speeds.

Scope for future work

The complexities involved in processor design would need a lot of effort and time in bringing out a robust design. Some of the features that we feel can be added to the processor as a part of future work are:

1. Improving the performance of reading from a register file after pre-processing steps. This will reduce the structural hazards at the read ports and will ensure efficient use of the read ports. This would need to design an efficient solution to solve the $O(1)$ problem.
2. Upgrading the design to provide support for conditional instructions (non-branch). This would need better renaming techniques and pre-processing techniques.
3. Designing a software model for the current architecture, so that the design can be evaluated against different benchmarks and provision for parameterization can be made available.
4. A cache hierarchy structure instead of the direct block memory used for instruction and data memory. That would require design of caches and a cache controller.
5. Improving the exception handling in the processor, like selective replay where not all instructions are flushed.
6. Provide bus support for the processor, so that peripherals can be interfaced.

Bibliography

- [1] Emily Blem, Jaikrishnan Menon and Karthekeyan Sankaralingam, A detailed analysis of contemporary ARM and x86 architectures, 19th IEEE international symposium on high performance computer architecture, 2013
- [2] ARMv7 architecture reference manual, 2014 re-print.
- [3] S. Palacharla, Complexity-Effective superscalar processors, PhD thesis, University of Wisconsin-Madison, 1998
- [4] E. J. Mclellan and D. A. Webb, The alpha 21264 microprocessor architecture, Proceedings of the international conference on computer design, 1998
- [5] Scott Thomas Bingham, A MIPS R10000 like out-of-order microprocessor implementation in verilog HDL, Thesis submitted to Cornell university, 2007
- [6] V. Zyuban, Inherently lower power High performance superscalar architectures, PhD thesis, university of Notre dame, Jan 2000
- [7] S. McFarlaing, Combining branch predictors, Technical Note (TN-36), DEC western research laboratory, 1993
- [8] J. E. Smith, A study of branch prediction strategies, In proceedings of the international symposium of computer architecture, pp. 135-148, 1981
- [9] T. Y. Yeh and Y. N. Patt, Alternative implementations of two level adaptive branch prediction, In proceedings of the international symposium of computer architecture, pp. 124-134, 1992
- [10] E. Hao, P. Chang and Y. Patt, The effect of speculatively updating branch history on branch prediction accuracy, revisited, In proceedings of the 27th annual international symposium on microarchitecture, pp. 228- 219, 1994

- [11] Scott E. Breach, T. N. Vijaykumar and Gurindar S.Sohi, The anatomy of the register file in a multiscalar processor, Proceedings of the 27th annual international symposium on microarchitecture, 1994
- [12] K. Farkas, N. Jouppi and P. Chow, Register file design considerations in dynamically scheduled processors, Technical report 95/10, Digital equipment corporation western research lab, November 1995
- [13] Nam Sung Kim and Trevor Mudge, Reducing Register ports using delayed write-back queues and operand pre-fetch, Proceedings of the 17th annual international conference on supercomputing, 2003
- [14] Rajeev Balasubramonian, Sandhya Dwarkadas and David H. Albonesi, Reducing the complexity of the register file in dynamic superscalar processors, 34th international symposium on microarchitecture 2001.
- [15] I. L . Park, Michael D. Powell and T. N. Vijaykumar, Reducing register ports for higher speed and lower energy, Proceedings of the 35th international symposium on microarchitecture, 2002
- [16] Dezso Sima, Superscalar instruction issue, IEEE Micro, 1997
- [17] R. M. Tomasula, An efficient algorithm for exploiting multiple arithmetic units, IBM journal of research and development, pp. 25-33, Jan 1967
- [18] A. Moshovos, S. Breach, T. Vijaykumar and G. Sohi, Dynamic speculation and synchronization of data dependencies, In proceedings of the 24th annual international symposium on computer architecture, pp. 181-193, 1997
- [19] Oscar Gustafsson, Andrew G. Dempster and Lars Wanhammer, Multiplier blocks using carry save adders
- [20] Gurhan Kucuk, Diitry Ponomarev and Kanad Ghose, Low Complexity reorder buffer architecture, International conference on supercomputing, June 2002
- [21] J. D. Fisher, C. Romo, E. John and W. Lin, Design and low power implementation of a reorder buffer, Thesis submitted to University of texas at San Antonio

- [22] Cristal, Ortega, Llosa and Valero, Out-of-order commit processors, IEE Proceedings in software, 2004.
- [23] Antonio Gonzalez, Fernando Latorre and Grigorios Magklis, Processor Microarchitecture-An implementation perspective, Synthesis lectures on computer architecture, Morgan and Claypool 2011.
- [24] John Paul Shen and Mikko H. Lipasti, Modern processor design- Fundamentals of superscalar processors.
- [25] John L. Hennessy and David A Patterson, Computer Architecture- A quantitative approach
- [26] Championship branch prediction, <http://www.jilp.org/cbp2014/>
- [27] ARM information center - <http://infocenter.arm.com/help/index.jsp>
- [28] P. S. Tulasiram, D. Vaithiyathan and R. Seshasayanan, Implementation of modified booth recoded wallace tree multiplier for fast arithmetic circuit, International journal of advanced research in computer science and software engineering, Vol. 4, Issue 10, 2014
- [29] R.Canal, J. M. Parcerisa, and A. Gonzalez, Dynamic cluster assignment mechanisms, In proceedings of the 6th international symposium on high performance computer architecture, France, 2000

Appendix A

Assembler

The increasing complexity in design demanded a lot of test benches to validate design. Initially manual assertion based testing was done which was time consuming. As the processor design does not support the entire ARMv7 ISA, the use of standard compilers and assemblers was ruled out.

As a supplement to the processor, an assembler was designed for the supported instructions. The assembler was designed in python. A simple look-up table based architecture was followed, each line of the assembly code is read. Then each instruction is broken down into segments using comma and space as delimiters. Then these segments are further processed to generate the required instruction word. This architecture is also known as single pass algorithm and does not have provisions to resolve assembler directives and labels. Hence the assembler cannot use labels. This design does not have any memory requirements. A limitation of this assembler is that errors cannot be detected and it simply quits in case of errors. The assembler takes in an assembly file as input and generates a binary file. The entire assembly file is scanned until end of file is reached. This assembler is compatible to all the processor designs we designed based on ARMv7 ISA. The out of order processor demands an additional requirement of generating the binary code that is compatible with the wide instruction memory implemented in this design. An additional python script is provided to do the same, it takes two adjacent instructions (each 32 bits) and combines to form one 64 bit entry in the instruction memory.

Appendix B

Matlab based RS and ROB simulator

A back-end architectural simulator based on matlab was designed to simulate the optimum sizes required for re-order buffer and different reservation stations. These sizes have immense effect on the throughput of the processor. They are not fixed quantities as they vary based on the architecture. The simulator only incorporated the design parameters from ROB allocation to write-back phase. The idea behind this design is to model the source dependencies and execution unit delays as random cycle delays rather than actually implementing the internals of the processor. This reduces the complexity in the simulator design. Out of order nature of the processor is introduced by randomness in picking the delays for instruction.

Execution Unit Type	Execution unit delay range	Dependency delay range
Integer unit	(1,2)	(1,6)
Shifted integer unit	(2,3)	(1,6)
MAC unit	(3,4)	(1,6)
Load and store unit	(1,2)	(1,6)
Branch unit	(1,2)	(1,6)

Table B.1: The delay ranges for dependency and execution unit for different types of instructions

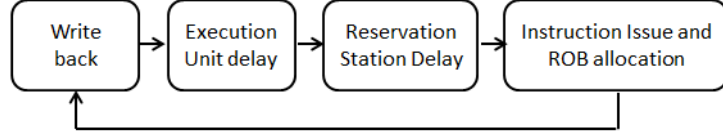


Figure B.1: The matlab simulator

The simulator randomly picks 4 instructions per cluster each cycle from an instruction set. The instruction set is designed to incorporate the probability of occurrences of different types of instructions in a code, for example integer based instructions have an occurrence probability of 40%.

Once the instructions are fetched, if there is an ROB slot and an available slot in the respective reservation station, then the simulator picks a random number from the respective ranges of both execution unit delay and dependency delay. Table B.1 lists down the delays for different type of instructions, which is largely dependent on the architecture of the processor. These delays are decremented every iteration and if the instruction is ready for writeback, then its ROB slot is freed. The iterations

Configuration	32-8-2-2-2-8	32-4-2-2-2-4	32-8-2-2-2-2-4	16-8-2-2-2-4	64-8-2-2-2-4
RS-IU Stall	9.05	54	6.62	0.6	7.1
RS-SIU Stall	34.09	21	29.12	15.04	29.4
RS-MAC Stall	33.96	21	28.54	15.02	29.14
RS-B Stall	33.99	22	29.05	15.07	29.2
RS-LS Stall	2.52	28	41.447	14.6	41.8
Overall RS Stall	38.24	57	48.126	24.12	48.64
ROB Stall	6.45	0.02	1.13	51.8	0

Table B.2: The reservation station and ROB stall percentages for different combinations of sizes. The configuration is to be read as ROB Size - IU RS size - SIU RS size - MAC RS size - LS RS size - B RS size

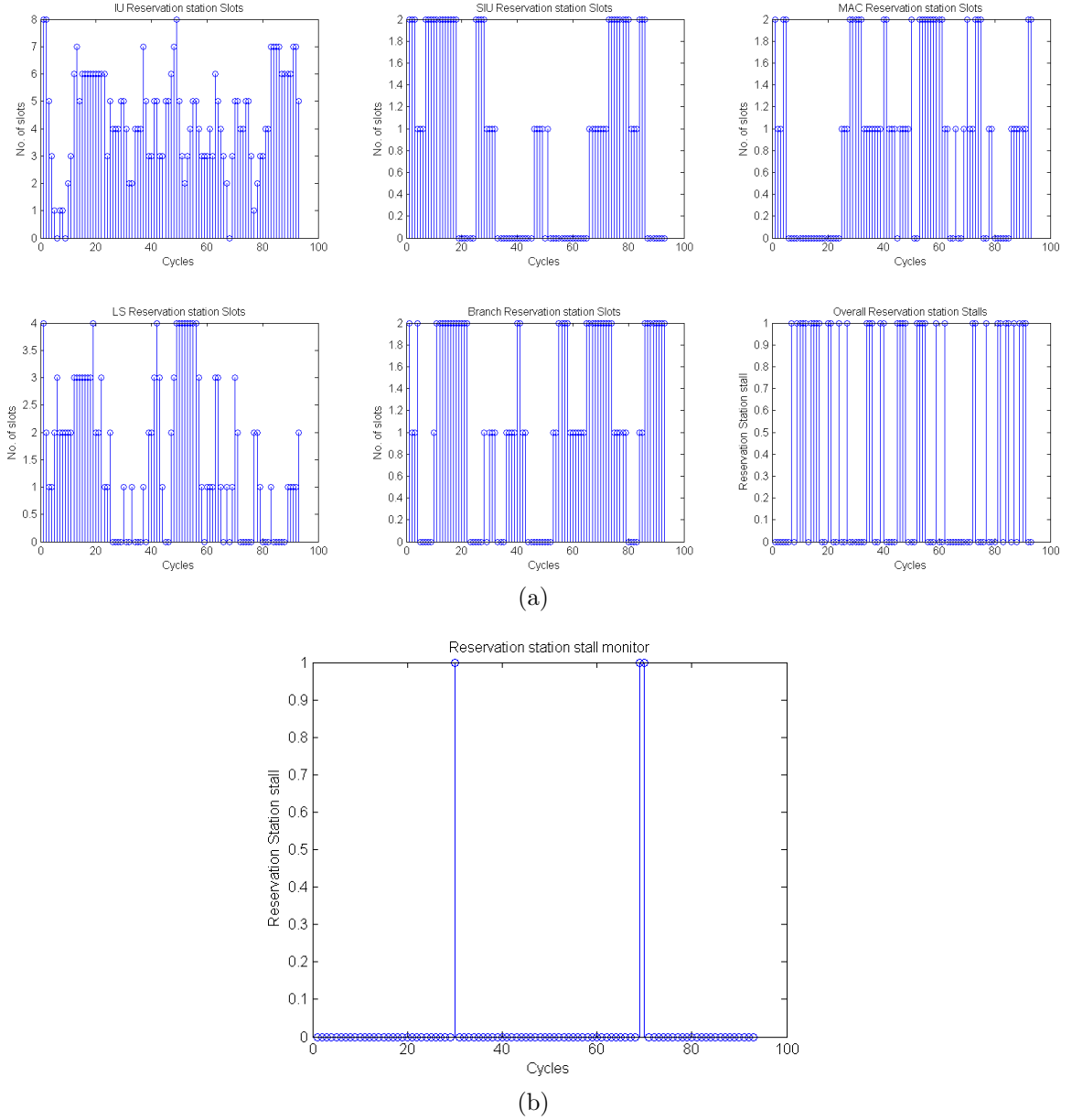


Figure B.2: Simulation Results for the configuration 32-8-2-2-2-4 (a) Reservation Station slots availability and overall reservation station stall and (b) re-order buffer stall during the course of simulation

order of the simulator is discussed in Fig. B.1.

The sizes of ROB and Reservation stations are optimized by running a monte carlo across different combinations of ROB and Reservation station sizes for 10,00,000

instructions each. The optimization was done based on the percentage of ROB and Reservation station stall for the entire simulation.

Table B.2 lists down the percentages of ROB and Reservation station stalls across different configurations. The stall percentages are computed based on the number of cycles that particular stall persisted with respect to the total number of cycles taken to execute the simulation. Fig.B.2 lists down the variation in reservation station stalls across the simulation. The simulation results closely match the intuitive sizes discussed in sections 6 and 8.1.

Appendix C

Instruction Set Architecture

As discussed in section 1.3, the processor was designed based on the ARMv7 instruction set architecture. This processor is compatible with only 90 instructions of the ISA, not considering the complex instructions that gives 64 bit results and conditional arithmetic instructions. These instructions were not considered as this will need a lot of hardware support. This chapter deals with all the instructions which are supported by the processor. The instructions are divided in to data processing, multiply and multiply accumulate, saturation, load and store and branch instructions. These instructions optionally update flags based on the flags valid bit in the instruction word

Data Processing Instructions

Instruction	Operation	Notes
AND	$Rd = Rn \& Rm$, $Rd = Rn \& \text{ShiftedRm}$, $Rd = Rn \& \text{Const}$	Logical AND operation
EOR	$Rd = Rn \wedge Rm$, $Rd = Rn \wedge \text{ShiftedRm}$, $Rd = Rn \wedge \text{Const}$	Logical Exclusive OR operation
SUB	$Rd = Rn - Rm$, $Rd = Rn - \text{ShiftedRm}$, $Rd = Rn - \text{Const}$	Subtraction operation
RSB	$Rd = Rm - Rn$, $Rd = \text{ShiftedRm} - Rn$, $Rd = \text{Const} - Rn$	Reverse subtraction

Instruction	Operation	Notes
ADD	$Rd = Rn + Rm$, $Rd = Rn + \text{ShiftedRm}$, $Rd = Rn + \text{Const}$	Simple Addition
ADC	$Rd = Rn + Rm + C$, $Rd = Rn + \text{ShiftedRm} + C$, $Rd = Rn + \text{Const} + C$	Addition along with carry flag
SBC	$Rd = Rn - Rm - C$, $Rd = Rn - \text{ShiftedRm} - C$, $Rd = Rn - \text{Const} - C$	Subtraction along with carry flag
RSC	$Rd = Rm - Rn - C$, $Rd = \text{ShiftedRm} - Rn - C$, $Rd = \text{Const} - Rn - C$	Reverse subtraction along with carry flag
TST	$Rd = Rn \& Rm$, $Rd = Rn \& \text{ShiftedRm}$, $Rd = Rn \& \text{Const}$	Logical AND updates only flags
TEQ	$Rd = Rn \wedge Rm$, $Rd = Rn \wedge \text{ShiftedRm}$, $Rd = Rn \wedge \text{Const}$	Logical EOR updates only flags
CMP	$Rd = Rn - Rm$, $Rd = Rn - \text{ShiftedRm}$, $Rd = Rn - \text{Const}$	Compare, only flags are updated
CMN	$Rd = Rn + Rm$, $Rd = Rn + \text{ShiftedRm}$, $Rd = Rn + \text{Const}$	Compare Negative, only flags are updated
ORR	$Rd = Rn \parallel Rm$, $Rd = Rn \parallel \text{ShiftedRm}$, $Rd = Rn \parallel \text{Const}$	Logical OR operation
MOV	$Rd = Rm$	Transfer of data, does not update flags
LSL	$Rd = Rm \ll \text{Imm}(5)$, $Rd = Rm \ll Rn$	Left shift and conditionally affects flags
LSR	$Rd = Rm \gg \text{Imm}(5)$, $Rd = Rm \gg Rn$	Logical shift right
ASR	$Rd = Rm \gg \text{Imm}(5)$, $Rd = Rm \gg Rn$	Arithmetic shift right
RRX	$Rd = Rm \ll 1$	Rotate right through carry flag
ROR	$Rd = Rm \ll Rn$, $Rd = Rm \ll \text{Const}$	Rotate right by Rn or immediate

Instruction	Operation	Notes
BIC	$Rd = Rn \& \sim Rm$, $Rd = Rn \& \sim \text{ShiftedRm}$, $Rd = Rn \& \sim \text{Const}$	Logical AND of Rn with compliment of Rm
MVN	$Rd = \sim Rm$, $Rd = \sim \text{ShiftedRm}$, $Rd = \sim \text{Const}$	Movies complement of a register in to destination

Multiply and MAC instructions

Instruction	Operation	Notes
MUL	$Rd = Rm * Rn$	Multiplication operation and only lower 32 bits are stored; overflow and carry flags are not updated
MLA	$Rd = Rm * Rn + Rs$	Multiply and accumulate operation; Lower 32 bits are only stored; carry and overflow flags are not updated

Saturation based instructions

Instruction	Operation	Notes
QADD	$Rd = Rm + Rn$	Saturation based addition will saturate to most positive or negative number; Will update saturation flag
QSUB	$Rd = Rm - Rn$	Saturation based subtraction will saturate to most positive or negative number; Will update saturation flag

Load and store based instructions

Instruction	Operation	Notes
STR	$\text{DMEM}[\text{Rn} + \text{Const}] = \text{Rt}$, $\text{DMEM}[\text{Rn} + \text{Rm}] = \text{Rt}$	Stores a word in to memory; offset for address calculation can be register or immediate
LDR	$\text{Rt} = \text{DMEM}[\text{Rn} + \text{Const}]$, $\text{Rt} = \text{DMEM}[\text{Rn} + \text{Rm}]$	Loads a word from memory to a register; offset for address calculation can be register or immediate
LDRB	$\text{Rt} = \text{DMEM}[\text{Rn} + \text{Const}]$, $\text{Rt} = \text{DMEM}[\text{Rn} + \text{Rm}]$	Loads a byte from memory to a register; The remaining data is zero extended
LDRSB	$\text{Rt} = \text{DMEM}[\text{Rn} + \text{Const}]$, $\text{Rt} = \text{DMEM}[\text{Rn} + \text{Rm}]$	Loads a byte from memory to a register; The remaining data is sign extended
LDRH	$\text{Rt} = \text{DMEM}[\text{Rn} + \text{Const}]$, $\text{Rt} = \text{DMEM}[\text{Rn} + \text{Rm}]$	Loads a half word from memory to a register; The remaining data is zero extended
LDRSH	$\text{Rt} = \text{DMEM}[\text{Rn} + \text{Const}]$, $\text{Rt} = \text{DMEM}[\text{Rn} + \text{Rm}]$	Loads a half word from memory to a register; The remaining data is sign extended
STRB	$\text{DMEM}[\text{Rn} + \text{Const}] = \text{Rt}$, $\text{DMEM}[\text{Rn} + \text{Rm}] = \text{Rt}$	Stores a byte in to memory; offset for address calculation can be register or immediate
STRH	$\text{DMEM}[\text{Rn} + \text{Const}] = \text{Rt}$, $\text{DMEM}[\text{Rn} + \text{Rm}] = \text{Rt}$	Stores a half word in to memory; offset for address calculation can be register or immediate

Branch instructions

Instruction	Operation	Notes
BEQ	If(ZF==1); PC = PC + Imm	Branch if zero flag is set
BNE	If(ZF==0); PC = PC + Imm	Branch if zero flag is clear
BCS	If(CF==1); PC = PC + Imm	Branch if carry flag is set
BCC	If(CF==0); PC = PC + Imm	Branch if carry flag is clear
BMI	If(NF==1); PC = PC + Imm	Branch if negative flag is set
BPL	If(NF==0); PC = PC + Imm	Branch if negative flag is clear
BVS	If(OF==1); PC = PC + Imm	Branch if overflow flag is set
BVC	If(OF==0); PC = PC + Imm	Branch if overflow flag is clear
BHI	If(CF==1) and (ZF==0); PC = PC + Imm	Branch if carry is set and zero flag is clear
BLS	If(CF==0) or (ZF == 1); PC = PC + Imm	Branch if CF is clear or zero flag is set
BGE	If(NF==OF); PC = PC + Imm	Branch if negative flag equal to overflow flag
BLT	If(NF!=OF); PC = PC + Imm	Branch if negative flag is not equal to overflow flag
BGT	If(NF==OF) and (ZF == 0) ; PC = PC + Imm	Branch if negative flag is equal to overflow flag and zero flag is clear
BLE	If(NF!=OF) or (ZF == 1) ; PC = PC + Imm	Branch if negative flag is not equal to overflow flag or zero flag is set
B	PC = PC + Imm	Unconditional Branch

Appendix D

Processor Variants

As a part of learning the architecture at hardware level, we implemented several processor designs. listed below are highlights of these processor designs:

D.1 ARM processor Variants

D.1.1 Single Cycle Design

A subset of 140 instructions were selected for implementation, which includes all arithmetic and logical instructions, MAC instructions, saturation instructions, conditional and unconditional branch instructions. Among these instructions there are certain instructions that need multiple cycles for execution (Multiply or MAC instructions that writes 64 bit result). The Architectural description of the processor is as listed:

- Instruction memory is Byte addressed and Little endian based memory organized. It is 256x8 bits in size. In each instruction fetch, 4 bytes are fetched.
- The register file has 16 32 bit registers. r14 and r15 are special purpose registers - Link register and program counter respectively. The register file has 3 read ports and 1 write port. An additional read and write port is provided for the program counter. An additional barrel shifter is provided at the output of register file for shifted register based instructions.

- The implementation has a separate ALU and MAC unit. The output of saturation unit depends on the outputs of ALU, Saturation unit is placed in the datapath as the timing budget is very less compared to the area improvement.
- Data memory is also byte addressed and little endian based memory organized. It is 256x8bits in size. It has asynchronous read and synchronous write. Data memory supports byte, Half word and Word based store and load instructions.

The design was tested using self written codes to test corner cases (Fibonacci series, code to test MAC instructions). The design was successfully synthesized on Xilinx Virtex 5 FPGA and has a maximum operating frequency of 53.207 MHz.

D.1.2 Canonical 5 Stage pipeline design

In order to improve the efficient utilization of resources in the processor, we went to a classic 5 stage pipeline design from the single cycle design, that consists of fetch, decode, execute, memory access and write back stages (IF-ID-EX-MEM-WB).

Some of the modifications done in this design from single cycle implementation are listed below,

- Instruction memory is a distributed ROM whose size is 256 x 32 bits, that limits the PC size to only 7 bits.
- The MAC instructions that generates 64 bit results and take 2 cycles for completion have been removed from the ISA.
- Also, R15 in the register file is no longer PC and link register which was previously R14 is removed and hence all the instructions related to it were removed from the ISA.
- Data dependencies are taken care by data forwarding paths. Hazard detection circuit was designed to identify such dependencies. The design had paths from memory stage and Write back stage to Execution stage.
- Load after store hazard was dealt with an internal forwarding path in memory stage which stores the loaded value for one cycle.
- Jump instruction incurs a compulsory penalty of 1 clock cycle.

Dynamic branch prediction

- To further reduce the 1 cycle penalty for branch instructions, we implemented a dynamic 2 bit branch predictor in IF stage.
- This unit has a branch target buffer(BTB) which is a cache that stores the branch target address and prediction bits, that further helps in deciding whether branch should be taken or not. The BTB has 32 entries and each entry is 14 bits wide, that store the LSB 3 bits of the program counter as Tag, 8 bits of branch target address, 2 prediction Bits and 1 valid bit.
- Irrespective of the branch instruction, it checks for the entry in the BTB and makes an entry in the next clock cycle , if it is a branch instruction and miss occurs.
- When a branch is encountered first time we will have a compulsory miss. A mechanism must be provided to recover from this stage and make a fresh entry in to the branch target buffer. First time the prediction entry in BTB is such that, branch is always strongly taken.
- The second case, When a branch is detected in ID stage and if there is a misprediction, The instruction fetched must be flushed and Program counter must be redirected using recovery Program Counter. The recover program counter is stored in a buffer for 1 cycle as we can detect the status of branch in ID stage itself. Thus an early decode for branch was provided to reduce 1 cycle branch penalty, if misprediction occurs.
- In case of wrong prediction or correct prediction, the prediction bits in the BTB for the respective branch needs to be updated.

With the inclusion of pipeline buffers, operand forwarding and dynamic branch prediction, the operating frequency after synthesis was improved to 91.477 MHz.

D.2 MIPS processor Implementations

Due to sheer complexity of implementing the design using ARM V7 ISA, we have considered a smaller ISA (Mini MIPS) and implemented single cycle, multi cycle and

D.2 MIPS processor Implementations

pipelined architectures in the initial phases of the project.

This ISA comprises of the following instructions.

Type	Instruction	Operation
Register based Arithmetic	Add Rd,Ra,Rb	$Rd = Ra + Rb$
Register based Arithmetic	Sub Rd,Ra,Rb	$Rd = Ra - Rb$
Register based Logical	And Rd,Ra,Rb	$Rd = Ra \text{ AND } Rb$
Register based Logical	Or Rd,Ra,Rb	$Rd = Ra \text{ OR } Rb$
Immediate based Arithmetic	Addi Rd,Ra,imm	$Rd = Ra + \text{imm}$
Immediate based Memory	Lw Rd,imm(Ra)	$Rd = \text{DMEM}[Ra + \text{imm}]$
Immediate based Memory	Sw Rd,imm(Ra)	$\text{DMEM}[Ra + \text{imm}] = Rd$
Conditional Jump	Beq Rd,Ra,imm	If($Ra == Rd$) $PC = PC + \text{imm} * 2$
Unconditional Jump	J addr	$PC = \text{addr} * 2$

Table D.1: Mini MIPS ISA

some of the features assumed in this architecture implementation are

- Instruction Memory is 256x16 bits width and it is addressed by an 8 bit program counter.
- The register file has 8 8 bit registers, where R0 is always tied to zero. The register file has 2 read ports and 1 write port.
- Arithmetic and logic unit performs 4 functions. It takes 2 inputs and gives 1 output along with the zero flag used for a branch instruction.
- Data memory is byte addressed of size 256x8 bits. This is accessed by load and store instructions. Address calculation is done in the ALU.

These specifications are maintained same across all the implementations using MIPS ISA.

D.2.1 Single and Multi-cycle design

In this architecture, only one instruction flows through the data path at any clock cycle, i.e no of cycles taken for each instruction (CPI) is 1. This results in a lot of hardware remaining idle and very long data path, affecting the operating frequency. The implementation was optimized to achieve a maximum operating frequency of 138.398 MHz.

Here, the single-cycle processor data path is divided into multiple stages. The architecture is implemented in such a way that all the arithmetic and logical instructions take 4 clock cycles, store instruction takes 4 cycles, load instruction takes 5 cycles and both conditional and unconditional branches take 3 cycles and 2 cycles respectively. This architecture does not issue an instruction every clock cycle. The critical data path is significantly shrunk leading to an operating frequency of 212.096 MHz. The disadvantage of this design is that CPI is greater than 1

D.2.2 Canonical 5 stage pipeline design

In order to get the benefits of $CPI = 1$ in single cycle processor and a higher operating frequency as in multi cycle implementation, we went into a classic 5 stage pipeline (IF-ID-EX-MEM-WB) design where we pipeline the control logic also allowing it to fetch an instruction every cycle. Some of the highlights of this design are listed below,

- Next instruction speculation was used. This means a static branch prediction that always the branch is not taken.
- An early decode for branch was provided to reduce the branch penalty. As only one type of branch instruction is present, we provided a small logic in ID stage to determine if branch was taken or not. Branch and Jump instructions incur a compulsory 1 cycle penalty.
- There are no structural dependencies in this design as both instruction and data memory take only 1 cycle for read or write.
- Data dependencies are taken care by data forwarding paths. Hazard detection circuit was designed to identify such dependencies. The design had paths from memory stage and Write back stage to Execution stage.

- Load after store hazard was dealt with an internal forwarding path in memory stage which stores the loaded value for one cycle.
- As expected the operating frequency was close to multicycle but due to data forwarding paths and additional logic to ensure the functionality, the frequency dropped to 194.554 MHz.

Dynamic Branch Prediction To further reduce the 1 cycle penalty for branch instructions, as we encounter a branch every 8 instructions at least, we implemented a dynamic 2 bit branch predictor in IF stage. A similar Branch target buffer was used as in the pipelined ARM design. With the inclusion of Dynamic branch prediction the reduction in operating frequency was not that significant as we fetch from the branch target buffer in parallel with the instruction memory. The operating frequency after synthesis was around 182.052 MHz.

D.2.3 Two-Way In-order Superscalar processor Design

To further improve the CPI of the pipelined processor, the design was upgraded to a two way in-order superscalar processor design. The design has two pipeline paths A and B to support execution of two instructions simultaneously. Listed below are modifications done to 5 stage pipelined processor to make it a two way superscalar processor:

- The instruction fetch stage fetches two instructions every cycle from the instruction memory. The branch target buffer is also provided with two read ports to enable dynamic branch prediction, since both instructions can be branch instructions. The BTB also can update two locations in the buffer every cycle. The conditions for determining the next program counter drastically increases.
- The decode stage will take care of WAW, RAW hazards. Few of the hazards that are considered are listed below, Data dependencies between instruction in the same stage are handled by letting the instruction in Pipe A to flow and stalling Pipe B for 1 cycle. Branches are resolved in the similar way.
- Execution stage / Memory stage / Write back stage These stages are similar to the 5 stage pipeline but they are modified to handle 2 instruction in every stage.

D.2 MIPS processor Implementations

- The Execution unit is modified to handle 4 data forwarding paths. Two paths from both memory units and two from write-back stages.
- Similar changes are performed to register file. The register file has 4 read ports and 2 write ports. Only instruction in Pipe B is written when both instructions write to the same register.

Ideally, a two way superscalar processor must give IPC of 2, but due to hazards we may be able to execute 2 instructions every cycle. There is significant improvement in performance compared to a scalar pipelined processor. Post synthesis maximum operating frequency is about 150MHz.