Implementation of Dual Issue and Out Of Order execution for RISC processor

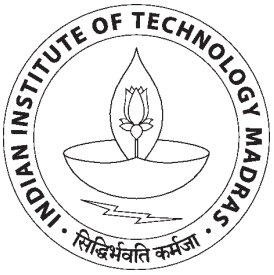
A Project Report

submitted by

SURESH KUMAR MEESALA

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

May 2013

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementation of Dual Issue and Out Of Order execution for RISC processor**, submitted by **Suresh Kumar Meesala**, 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 carried out by him under my 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. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I Would like to express my sincere gratitude to my guide, ***Dr. V. Kamakoti*** for his valuable suggestions, encouragement and support. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to ***Mr. G.S. Madhusudan*** for his encouragement and motivation through out the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work.

I would like to thank my faculty advisor ***Dr. Nagendra Krishnapura*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to project team members Neel, Naveen, Vikas, Abhishek, Avinash, Bhasker, Debi, Rahamthulla, Chidhambaranathan, Lokesh and Dinesh for their help and support.

Finally, I would like to thank all my friends for making my stay at IIT Madras a memorable one.

ABSTRACT

KEYWORDS: Common Data Bus, Data forwarding, Dual Issue

project work involves “Implementation of Dual Issue and Out Of Order Execution” for the in-house RISC processor. This work involves improvising the Out Of Order execution of the processor by implementing a Common Data Bus for data Forwarding. Using a Common Data Bus for data forwarding leads to improved clock frequency and simpler structure, Compared to the existing architecture. Some phases of the processor were redesigned to support data forwarding through common data bus.

To attain a higher degree of functional unit utilization, dual instruction issue has been implemented. Necessary modifications were employed to convert the single issue machine into a dual issue one. Significant changes have been incorporated in the issue phase, Write back phase and Commit Phases in order to allow for dual issue.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
RISC	Reduced Instruction Set Computer
BSV	Bluespec System Verilog
RTL	Register Transfer Level
CDB	Common Data Bus
ROB	Reorder Buffer
OOO	Out Of Order
RAW	Read After Write
WAW	Write After Write
WAR	Write After Read
OF	Operand Fetch
IPF	Issue Packet formation
DIP	Dispatch Of Issue Packets
PSW	Processor Status Word
PC	Program Counter
ISR	Interrupt Service Routine

CHAPTER 1

INTRODUCTION

1.1 Overall Architecture

The overall architecture of the proposed in-house processor is as shown in Figure: 1.1.

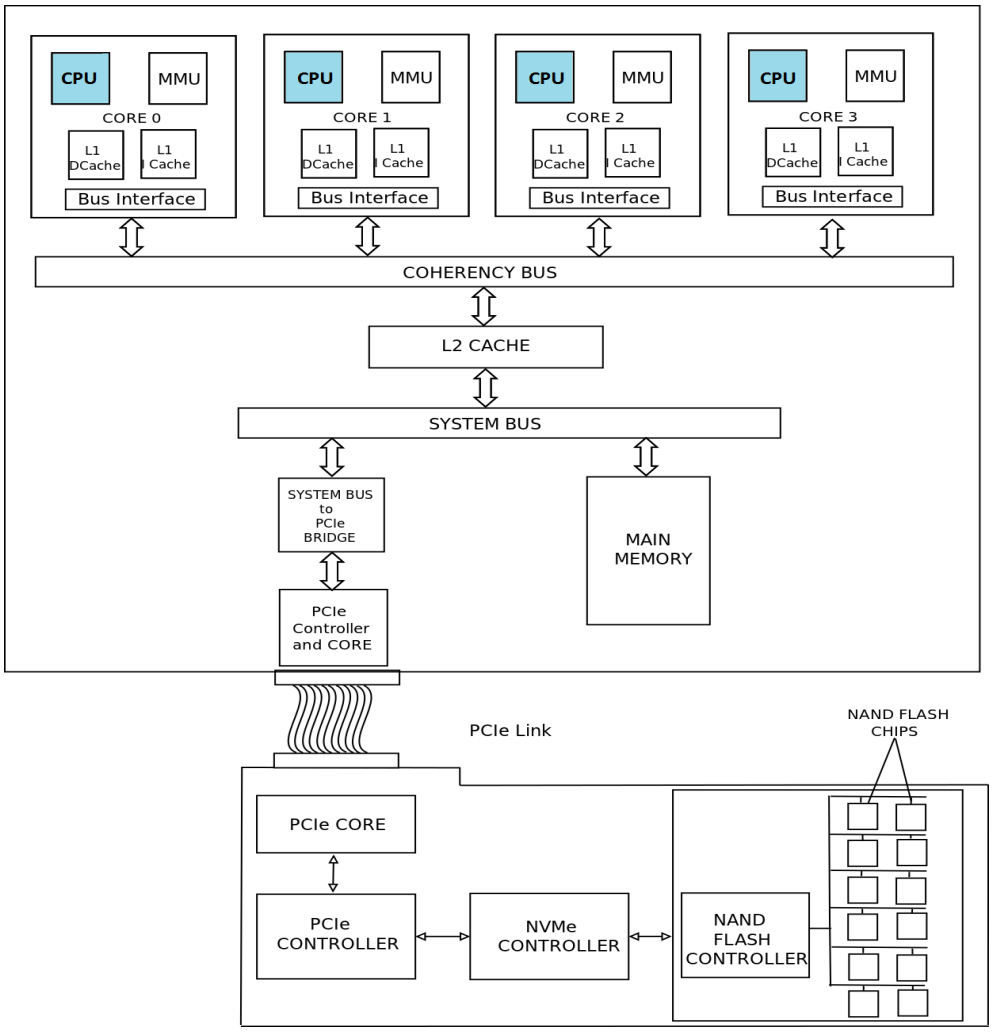


Figure 1.1: Overall architecture

The proposed processor has quad-core architecture, with support for two levels of cache hierarchy and a single on-chip DRAM. It implements cache coherency at L1 cache level, with coherency bus. It has NVM Express based I/O Subsystem with a PCI Express interface. The CPU core is designed based on Tomasulo algorithm. It supports dual instruction issue and out of order execution. project work involves design related to CPU as highlighted in Figure: 1.1.

1.2 Contribution

Project work involves the following two tasks.

- Improvisation of Out Of Order Execution in the processor, by using a Common Data Bus for data forwarding.
- Implementation of Dual Instruction Issue.

1.3 Need

Out Of Order Execution : In the existing architecture, there is no data forwarding among the reservation stations. So, there is a need to redesign the Out of order execution mechanism in the processor to facilitate data forwarding among the reservation stations. This can be achieved by using a Common Data Bus in Write back phase in place of existing huge multiplexing logic. This also results in a simple and efficient design compared to the existing design. Further, this also leads to reduced latency in data forwarding.

Dual Issue : By implementing dual instruction issue, higher degree of functional unit utilization can be attained, which leads to improved performance of the processor.

1.4 Outline

Chapter 2 presents background information regarding Super-scalar processor design and Bluespec System Verilog. It describes the design methodology and implementation details of the existing design of the processor. It also describes the limitations of the existing design and scope for improvisation.

Chapter 3 describes the implementation of Out Of Order execution using Common Data Bus for data forwarding and design modifications in various stages. It describes the setup for verification and verification strategy. It also presents the design challenges and synthesis results of modified design of single issue processor with Out Of Order execution (with CDB).

Chapter 4 describes the implementation of dual instruction issue. It deals with verification strategy and simulation results. It also presents the design challenges and Synthesis results of dual issue processor.

Chapter 5 gives conclusion and possible directions for future work.

CHAPTER 2

BACKGROUND

This chapter describes briefly about basic concepts of Superscalar processor design[1]. It also describes the basis concepts involved in building a design in Bluespec System Verilog (BSV)[2]. It then presents the details of the earlier design of the processor[3]. It also presents the limitations of the existing design and scope for its improvisation.

2.1 SUPERSCALAR ARCHITECTURE

A superscalar CPU architecture exploits a form of parallelism called instruction level parallelism within a single processor[4]. Superscalar describes a microprocessor design that makes it possible for more than one instruction at a time to be executed during a single clock cycle.

In a superscalar design, the processor or the instruction compiler is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it has a dependency on another instruction and must be executed in sequence with it. The processor then uses multiple execution units to simultaneously carry out two or more independent instructions at a time.

2.1.1 Dynamic Scheduling

Dynamic scheduling is a method in which the hardware determines which instructions to execute, as opposed to a statically scheduled machine, in which the compiler determines the order of execution. Dynamically scheduled machines can take advantage of parallelism which would not be visible at compile time.

In essence, the processor executes instructions out of order. In Dynamic scheduling, instructions don't execute based on the order in which they appear, but rather on the availability of the source operands and required functional unit.

2.1.2 Tomasulo algorithm

Tomasulo algorithm [1] is a method of implementing dynamic scheduling. Tomasulo algorithm uses register renaming to eliminate output and anti-dependencies, i.e. WAW and WAR hazards.

Tomasulo algorithm implements register renaming through the use of what are called reservation stations. Reservation stations are buffers which fetch and store instruction operands as soon as they are available. Source operands point to either the register file or to other reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station. The reservation station then sends the result to the register file and any other reservation station which is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other

results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction). In Tomasulo algorithm, the control logic is distributed among the reservation stations.

2.1.3 Speculative Execution

While out-of-order execution can lead to performance improvements, it can also lead to some problems. When implementing branch prediction, a wrong prediction would result in a flushed pipeline. However, when dealing with out-of-order execution, the problem arises where an instruction which should not have been executed has already written its results to the register file. One solution, of course, is to not allow instructions past a branch to execute until the branch has been resolved. But this mitigates the effects of branch prediction.

A better solution, and the one used in modern processors, is to require that instructions be completed in order, in other words, instructions are issued in-order, executed out-of-order, then committed to the register file in-order. Thus, to anything outside of it, the microprocessor looks like it is executing instructions in-order; the first instruction that comes in is the first that is completed.

In-order completion is accomplished by placing a reorder buffer right before writing to the registers. The reorder buffer is an ordered buffer which holds the results of instructions waiting to be committed. The buffer is ordered in program order, thus the top of the buffer

is the first instruction that should be committed. If an instruction completes, but earlier instructions are still executing, the result is stored in the reorder buffer until all earlier instructions have been committed.

When used in conjunction with dynamic branch prediction, the reorder buffer can be used to perform speculative execution.In case of misprediction, the reorder buffer can be flushed starting at the branch since no instructions past the branch have permanently changed the state of the machine. The reorder buffer also allows for precise exception handling since instructions are completed in order. Thus, if an exception occurs, a flag is set,and when the instruction goes to commit, the flag is detected and the exception is handled.

2.1.4 Multiple Instruction Issue

In multiple issue processor, the processor tries to issue more than one instruction per cycle so as to keep all of the functional units busy. Instruction issue logic is the primary challenge with multiple issue processor. Using dynamic scheduling, the CPU can issue multiple instructions with dependencies and serialize them later using hazard detection logic.

Tomasulo algorithm can be extended to support multiple instruction issue. The instructions are not issued to the reservation stations out of order, since this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling we may allow the pipeline to issue any combination of two instructions in a clock.

Extra logic is necessary to handle multiple instructions at once, including any possible dependencies between the instructions.At the back end of the pipeline, the processor must also be able to complete and commit multiple instructions per clock.

2.2 Bluespec System Verilog

2.2.1 Building a design in BSV

Figure: 2.1 illustrates the various steps involved in building a design in Bluespec System Verilog[2].

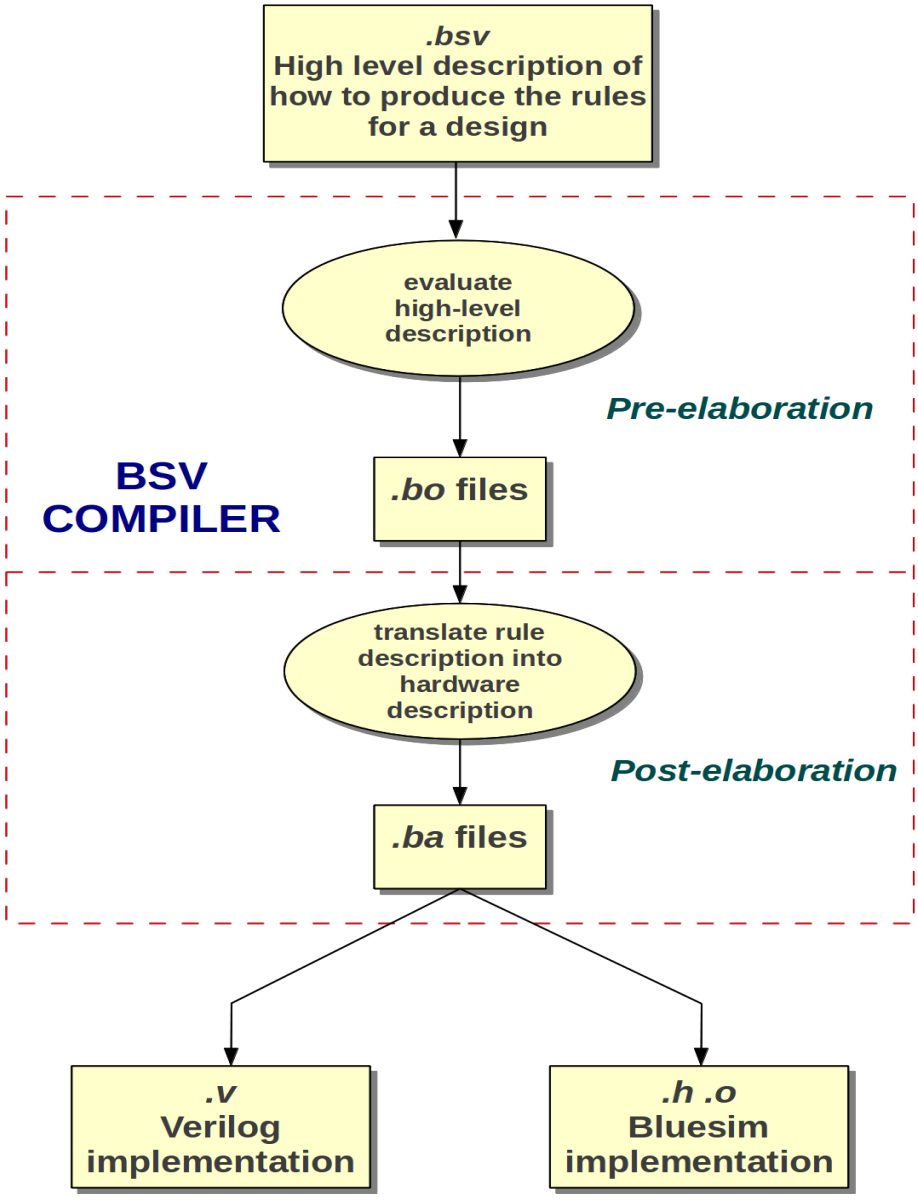


Figure 2.1: Building a design in BSV

- Designer writes a BSV program. It may optionally include Verilog, SystemVerilog, VHDL,and C components.
- The BSV program is compiled into a Verilog or Bluesim specification. This step has two distinct stages:
  1. pre-elaboration - parsing and type checking
  2. post-elaboration - code generation
- The compilation output is either linked into a simulation environment or processed by a synthesis tool.

2.2.2 Rules

BSV does not have always blocks like Verilog. Instead, rules are used to describe all behavior (how state evolves over time). Rules are made up of two components:

Rule Condition: a Boolean expression which determines if the rule body is allowed to execute(*"fire"*).

Rule Body: a set of actions which describe the state updates that occur when the rule fires.

We can logically think of a rule's execution as ***instantaneous, complete and ordered*** w.r.t execution of all other rules.

Instantaneous:

- Conceptually, all the actions in the rule body occur at a single,common instant - there is no sequencing of actions within a rule.

Complete:

- When fired, the entire rule body executes. There is no concept of "partial" execution of a rule body.

Ordered:

- Each rule execution conceptually occurs either before or after every other rule execution, but never simultaneously.

Some constraints are imposed on rules. These constraints are:

- Each rule fires at most once within a clock.
- Certain pairs of rules, which we will call conflicting, cannot both fire in the same clock.

2.2.3 Module hierarchy and Interfaces

In BSV, a module's interface is an abstraction of its verilog port list. In BSV, the interface declaration and module declaration are separate. So, a common interface can be used by several modules, with out having to repeat the declaration in each of its implementation modules.

An interface declaration specifies the methods provided by every module that provides the interface,but does not specify the methods implementation. The implementation of the interface methods can be different in each module that provides that interface. The definition of the interface and its methods is contained in the providing module.
BSV classifies interface methods into three types:

- ***Value Methods***: These are methods which return a value to the caller, and have no"*actions*"(i.e., when these methods are called, there is no change of state, no side-effect).
- ***Action Methods***: These are methods which cause actions (state changes) to occur. One may consider these as input methods, since they typically take data into the module.
- ***Action Value Methods***: These methods couple Action and Value methods, causing an action (state change) to occur and they return a value to the caller.

Every module uses the interface(s) just below it in the hierarchy. Every module provides an interface to the module above it in the hierarchy.

2.3 EARLIER DESIGN OF PROCESSOR

The design of the existing processor [3] is based on the Tomasulo algorithm described in section-2.1.2. It supports Out Of Order execution and precise exceptions. However, it has limitations in the implementation methodologies for some phases of the design, leading to the under-performance of the processor.

The instructions after being fetched are decoded and sent to the issue stage. The fetch and decode phases are not modified in the new design. The following sections describes the details of the earlier design starting from the Issue stage.

2.3.1 Issue Stage

The instruction after being decoded enters the issue phase. The block diagram of issue stage showing its interface is as shown in figure -2.2.

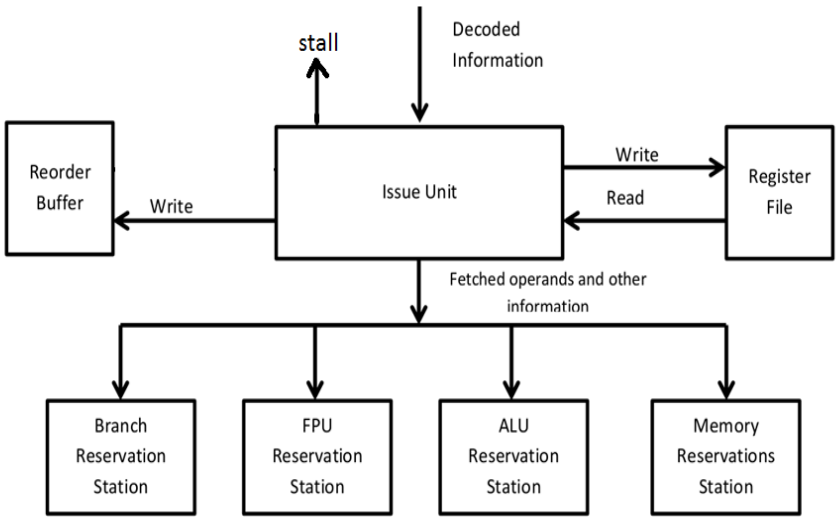


Figure 2.2: Earlier design : Issue stage

In the issue phase the instruction is allotted the reservation station corresponding to the functional unit which executes the instruction. The issue unit dispatches the instruction if there is an empty slot in both the reservation station and the ROB.

Before dispatching the instruction, the issue stage fetches the operands. The operands will first be checked for in the register file using their source addresses of the operands. However, if the register file does not have the updated value of the operands, it contains the ROB entry number (rob_number) of instruction which updates the value later. The issue stage sends this ROB number, if updated operands are not yet available.

Figure - 2.3 shows the operand fetching technique in the issue phase.

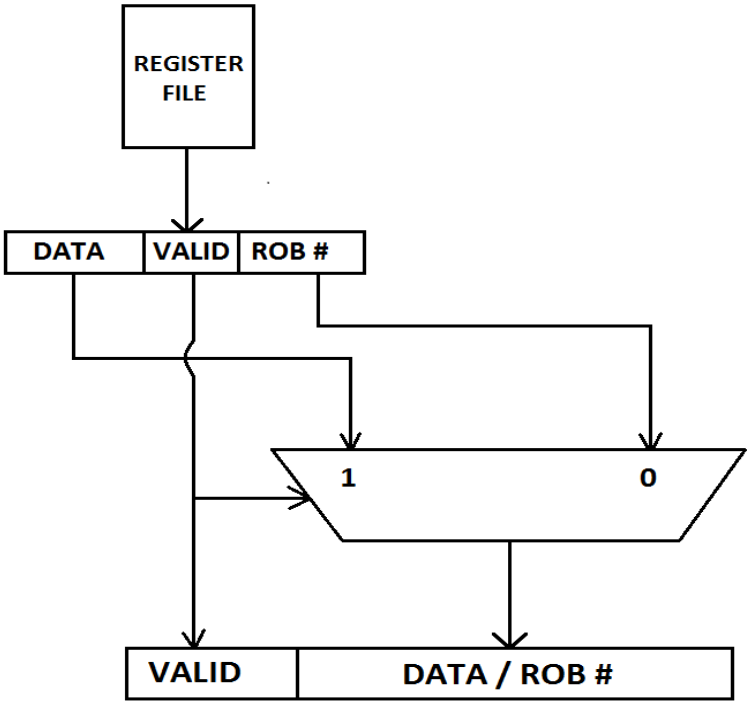


Figure 2.3: Earlier design : Operand Fetching

2.3.2 Execute and Write back stage

The instruction after being issued resides in the respective reservation station. In the reservation station, if one or more operand values are not yet available, the reservation station continuously scans the corresponding ROB entry by the rob_number of that operand.In bluespec implementation, this is accomplished through the method ***need_forwarding***.

As soon as the particular ROB entry receives a valid result, it will be forwarded to the reservation station to update the operands through the method ***forward_operand***. Only when all operands are available, the reservation station sends them to the functional unit for execution. The results from the functional units are sent to the corresponding ROB entry by its reservation station.Each reservation station updates the ROB individually through its own methods. Figure - 2.4 illustrates the operation of Execute and Write back stage.

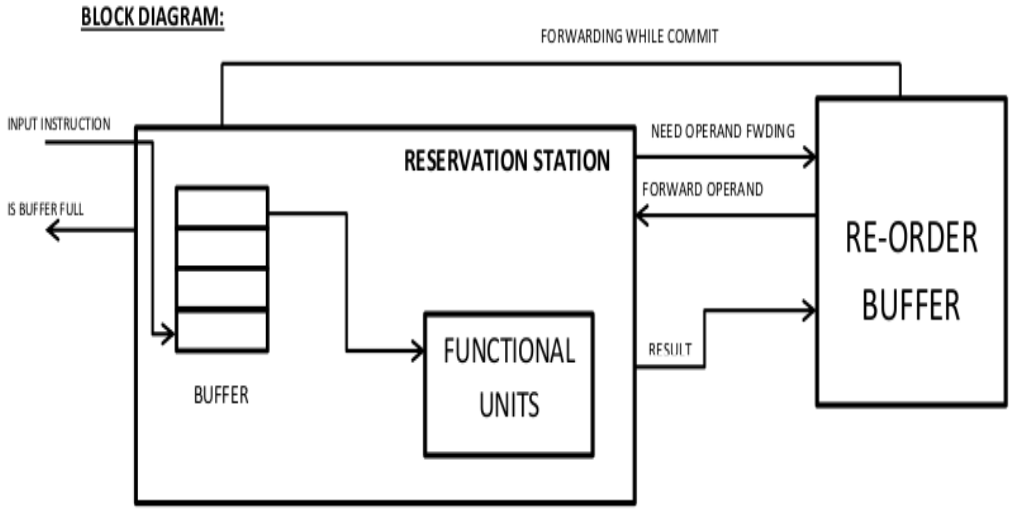


Figure 2.4: Earlier design : Execute and Write back stage

2.3.3 Commit Stage

In Commit stage, the results of the instruction execution are allowed to modify the state of the processor. On completion of instruction execution, the results are stored in Reorder buffer in the issue order i.e., in program order. The reorder buffer was implemented as a circular queue. The results are written into the register file strictly in program order, preserving the precise architectural state of processor in case of exceptions.

Another important task of the commit stage is to forward the results back to the reservation stations while committing an instruction. Each time an instruction is committed, the data that is to be written into the register file is also forwarded to all the reservation stations along with the rob_number of the ROB entry which is being committed. The reservation stations update operands of the instructions waiting for the committed data. This operand forwarding mechanism during instruction commit is needed to ensure that all instructions dependent on the committed instructions gets their operands. In bluespec implementation, this is accomplished by the method ***forward_while_commit***.

Figure -2.5 illustrates the operations in commit stage.

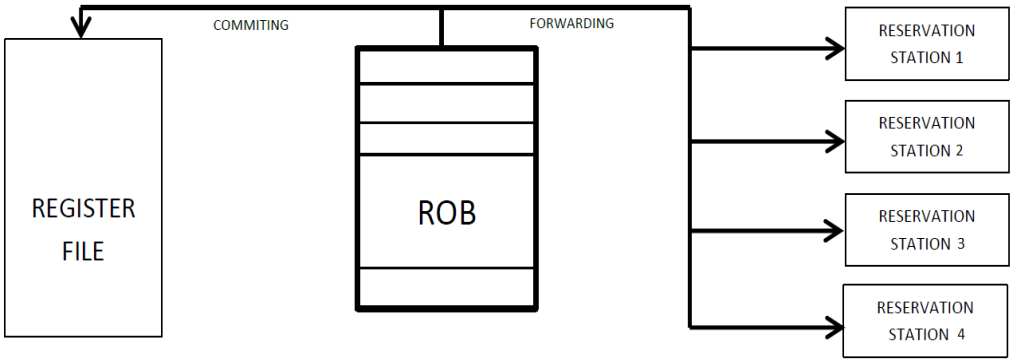


Figure 2.5: Earlier design : Commit stage

2.3.4 Limitation of the design

As mentioned before, the existing processor has limitations in implementation methodologies for some phases of the design. They are listed below :

ISSUE PHASE:

- The Status of Reorder buffer is not checked for Operands in the issue stage during Operand fetch.
- Only one instruction can be issued in a clock cycle (No support for multiple issue).

WRITE BACK PHASE:

- No data forwarding among the reservation stations.
- Extra latency of one cycle in data forwarding.
- Huge multiplexing logic for write Phase (CDB not present).

COMMIT PHASE:

- Need data forwarding from ROB while Committing instructions.
- Huge logic for data forwarding from ROB to individual Reservation stations.
- Doesn't Support Multiple instruction Commit.

2.3.5 Scope for Improvisation

To overcome most of the limitations of the existing design of the processor mentioned earlier, a Common Data Bus need to be implemented, in place of huge Multiplexing logic, for data forwarding among the reservation stations and to write the result in ROB. By using a common Data bus the latency in data forwarding can be decreased and also the need for forwarding the results while committing can be eliminated, as the results are forwarded directly from the corresponding reservations station through the common data bus.

The existing design can be further modified to issue two instructions in a clock cycle, so that the execution units are kept busy most of the time and are not under utilized. Also, Some functional units can be duplicated to reduce stalls due to structural hazards.

CHAPTER 3

OUT OF ORDER EXECUTION

This chapter describes the Implementation of Out Of Order execution with data forwarding through Common data bus. It starts with Microarchitectural description followed by implementation of Common Data Bus and modifications in various phases of the design. The Verification strategy, Design challenges and Synthesis results are dealt in the later sections.

3.1 MICROARCHITECTURAL DESCRIPTION

The design is based off of Tomasulo algorithm[1]. Figure:3.1 shows the main components and paths between the modules. *The highlighted portions represents the functional blocks of the processor, which I have modified for the new design.*. The Issue unit takes decoded instructions, reads the operands out of the register file and ROB, and issues the instructions to an appropriate reservation station. This unit also does a write to the register file that marks the destination register as a result that is currently being computed.

Each issued instruction is placed in a reservation station waiting for a specific type of execution unit. These stations hold the operands (or tags corresponding to pending operands). These operands listen to the common data bus (CDB) and update themselves when the results have been computed. Each cycle, each functional unit can take an instruction that has all of its operands ready and begin executing it. Once the result of the instruction is ready,

it is stored in a result register for that functional unit.Each cycle, all of the result registers from the functional units are inspected and, if available, a result is chosen to be broadcast on the CDB. This broadcast updates instructions that depend on that result that are waiting in various reservation stations. It also updates the corresponding ROB entry.

During commit, ROB updates the register file with the result, if the tag in the register file matches the tag of the result.

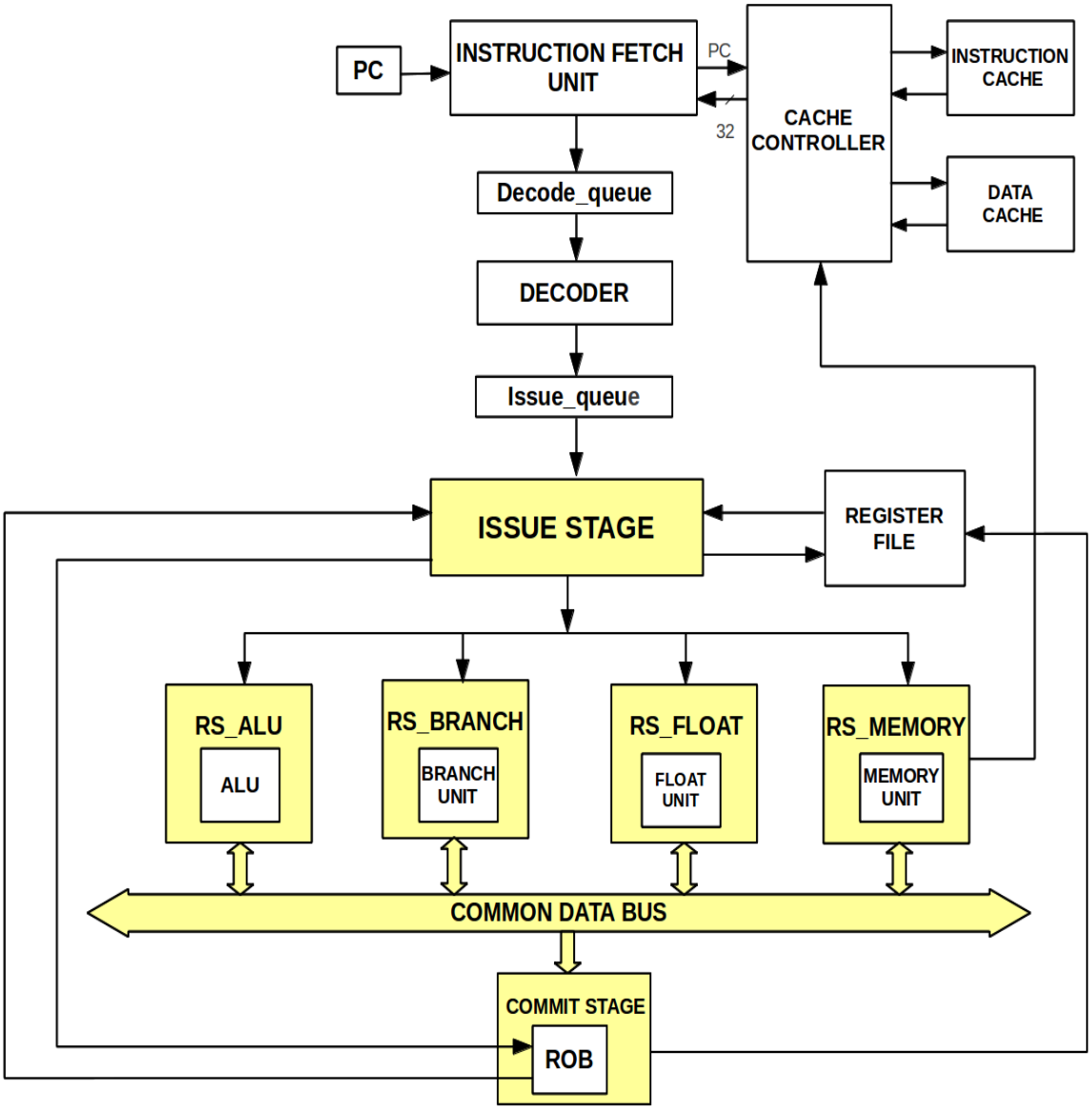


Figure 3.1: Microarchitectural organization of single issue processor with CDB

3.2 IMPLEMENTATION

3.2.1 Common Data Bus

To enable data forwarding among reservation stations a Common Data Bus need to be implemented.

3.2.1.1 Design Decision: Choosing the Bus Structure

To implement the Common Data Bus, Various bus structures to choose from are as follows.

1. AND-OR based Implementation.
2. Multiplexer based Implementation.
3. Tri-state based Implementation.

The bus structures based on Multiplexers and tri-states has been implemented and the implementation with tri-states was found to be better in terms of performance.

***Advantages of Tri-state bus*:**

- takes up fewer wires and smaller area foot print compared to the Multiplexer based design.
- Delay and Power consumption are not high.

3.2.1.2 Tri-state Bus Physical Structure

Figure 3.2 shows the operation of a tri-state buffer with a high true enable.

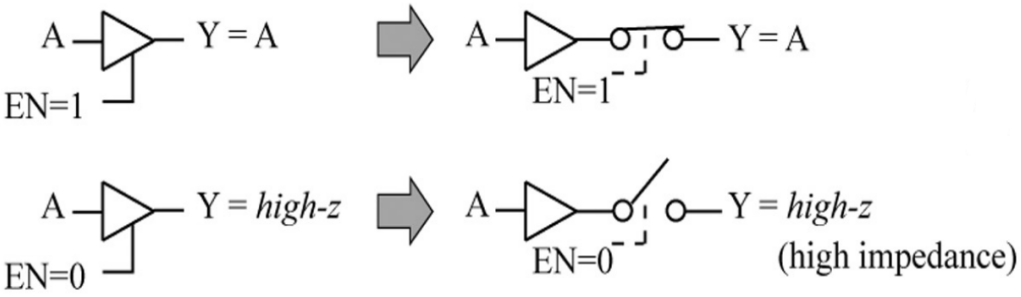


Figure 3.2: Operation of tristate buffer

A tristate buffer can be thought of as a non-inverting buffer with a switch on its output. When the buffer is enabled, the output is driven by the non-inverting buffer. When the buffer is disabled, the output is disconnected from the non-inverting buffer, and the output is left floating or in the high-impedance logic state.

The bus implementation based on tri-state buffers is illustrated in figure 3.3.

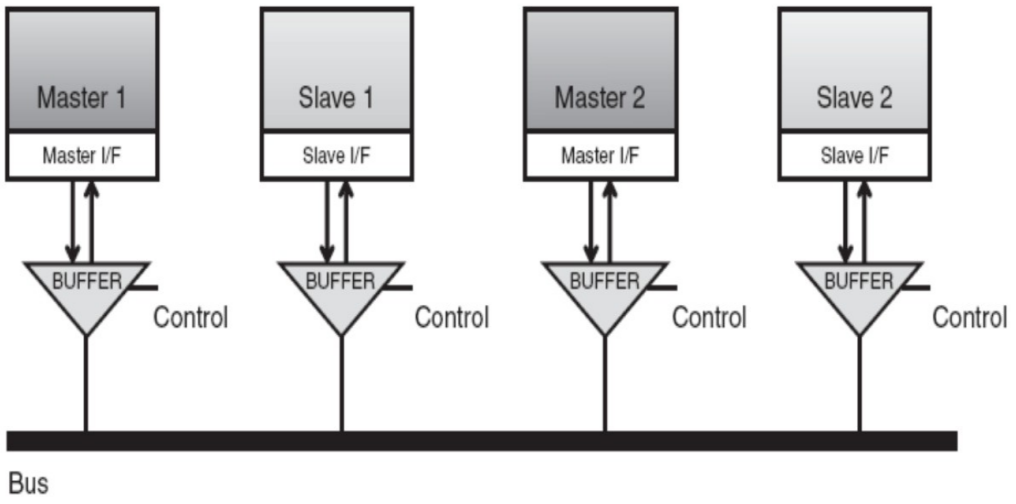


Figure 3.3: Tristate bus Physical structure

3.2.1.3 Implementation using Z BUS Library in Bluespec

BSV provides ***ZBus*** library which allows to implement and use tri-state buses [5]. Since BSV does not support high impedance or undefined values internally, the library encapsulates the tri-state bus implementation in a module that can only be accessed through predefined interfaces which do not allow direct access to internal signals .

ZBus consists of a series of clients hanging off a bus. The combination of the client and the bus is provided by the *ZBusDualIFC* interface which consists of 2 sub-interfaces, the client and the bus. The client sub-interface is provided by the *ZBusClientIFC* interface. The bus sub-interface is provided by the *ZBusBusIFC* interface.

No manipulations are needed to be done on the bus side, this is all done internally. The bus has to be built out of *ZBusDualIFC* and then drive values onto the bus and reads values from the bus using the *ZbusClientIFC*. The *mkZBus* module constructor function takes a list of *ZBusBusIFC* interfaces as arguments and creates a module which ties them all together in a bus.

The *ZBusClientIFC* allows a BSV module to connect to the tri-state bus. It has the following methods.

- ***drive*** method is used to drive a value onto the bus.
- ***get*** method allows each bus client to access the current value on the bus. If the bus is in an invalid state (i.e., has a high-impedance value or an undefined value because it is being driven by more than one client simultaneously), then the get method returns 0. In all other cases it returns the current value of the bus.
- ***fromBusValid*** method returns False if the bus is in invalid state, otherwise it returns True.

By using the BSV library, Zbus interface has been implemented/provided in all reservation stations and reorder buffer. The interfaces to all these modules are tied together to facilitate data forwarding. Necessary modifications are made to the entire design to ensure correct functionality and improve performance. The working of various units and the modifications made are described in detail in later sections.

3.2.1.4 Design Decision:Choosing the prioritization scheme

To avoid contention for driving the Z bus, prioritization of reservation stations has to be implemented.Two types of prioritization schemes have been implemented.

- Fixed prioritization scheme.
- Round-robin arbitration Scheme.

Fixed prioritization scheme has simpler structure, but is leading to starvation of lower priority reservation stations in some cases.

Round-robin scheme is a fair arbitration scheme with changing priorities. It has relatively complex structure, but gives fair chance to all reservation stations. Hence round robin arbitration has been chosen for the final implementation.

When multiple reservation stations are ready to drive Z bus in a given cycle, then the reservation station with highest priority among them is allowed to drive the Z bus. The other reservation stations wait for their chance to drive the bus in following cycles. Fixed priorities are assigned to the reservation stations.

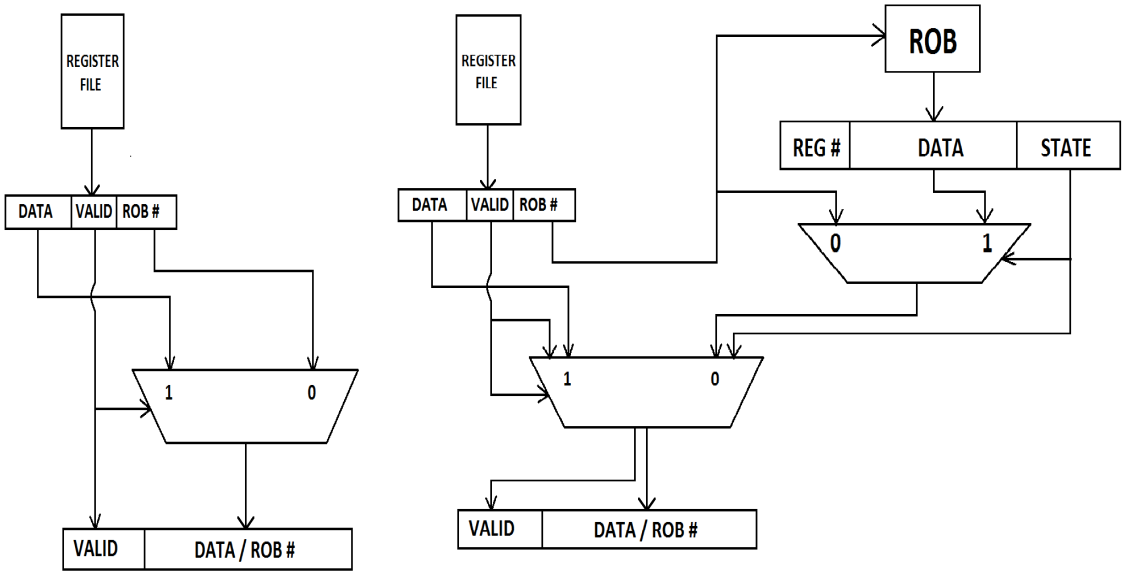
3.2.2 Issue Phase

The Issue phase of the design is almost similar to the earlier design described in section-2.3.1, except that *the Operand fetch mechanism is modified in the new design.*

3.2.2.1 Modified Operand fetch mechanism

In the earlier design, the Issue phase does not look in ROB for the updated operands if they are not available in register. It simply sends the corresponding ROB entry number in place of actual operands.

The operand fetch mechanism in the modified design is illustrated in figure: 3.4



(a) ***Earlier design*** (b) ***Modified design***

Figure 3.4: Issue Stage- Modified Operand Fetch Mechanism

In the modified design, the Issue unit first checks for the operands in the register file. If the updated values are not available, the register file has the rob_number of the instruction which updates its value. Using this index, Issue unit checks the corresponding reorder buffer entry. If the updated value is not available even in ROB, then it dispatches the instruction with that rob_number (instead of actual operand) to the reservation station. The reservation station uses this to obtain the value when the results are forwarded on the common data bus after execution.

3.2.2.2 Bluespec Implementation details

Figure :3.5 illustrates the bluespec implementation details of the modified issue phase.

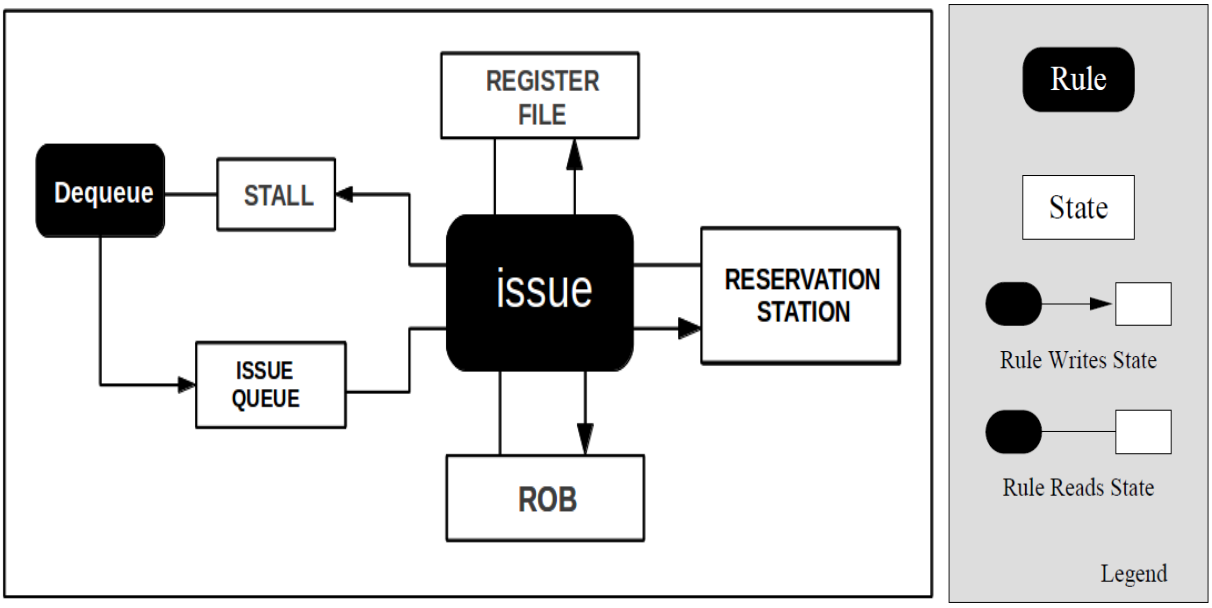


Figure 3.5: Issue stage: Bluespec Implementation details

The issue rule(***rl_issue***) is responsible for getting an instruction from the decode Unit and deciding which functional unit to issue the instruction to. This rule determines the type of the instruction and reads the value of the operands from the Register File. It also does a write to the main register file that updates the destination register to be a tag for the result of this instruction.

In addition, the issue rule checks the result that is currently waiting in ROB for being committed, if the updated value is not yet available in the register file. Each instruction is also issued a tag which is an entry in ROB. This tag is given for renaming purposes. In the end, the issue packet is queued into the appropriate Functional Units reservation station.

*The issue rule(***rl_issue***) fires only when the ROB has at least one free entry in ROB and the Issue queue has decoded instruction*. Issue rule reads from register file and ROB for operands. It also writes into register file and ROB. The issue rule reads the state of reservation station for free entries and writes to it. It also updates the ***stall*** signal which represents the issue status of the instruction, which is read by the dequeue rule (***rl_dequeue***) to dequeue the issue queue.

3.2.3 Execute and Write back Phase

The execution Core consists of the functional units and their associated reservation stations. Each functional unit has dedicated reservation station (distributed reservation station scheme). The instructions waiting in the reservation station for source operands are updated with the value when they are available on the common data bus.

In the earlier design, the values are updated only after they had been written in the ROB. But in the modified design they are forwarded directly through the common data bus, hence reducing the latency in data forwarding.

When all the source operands necessary for the operation are ready, it proceeds from the reservation stations to the functional unit(in the order in which it was pushed into the reservation station). The reservation station thus acts as a buffer between instruction schedule and instruction issue, where in the Issue unit can schedule subsequent instructions, while the current instruction waits in the reservation station for its operands.

The results from the functional units are driven on the common data bus along with the corresponding rob_number. The reorder buffer reads it from bus and updates the corresponding ROB entry (WRITE phase). Also the instructions waiting for the operands in reservation station update their values by continuously sensing the common data bus (Zbus). The implementation of the execute and write phase is as illustrated in Figure: 3.6.

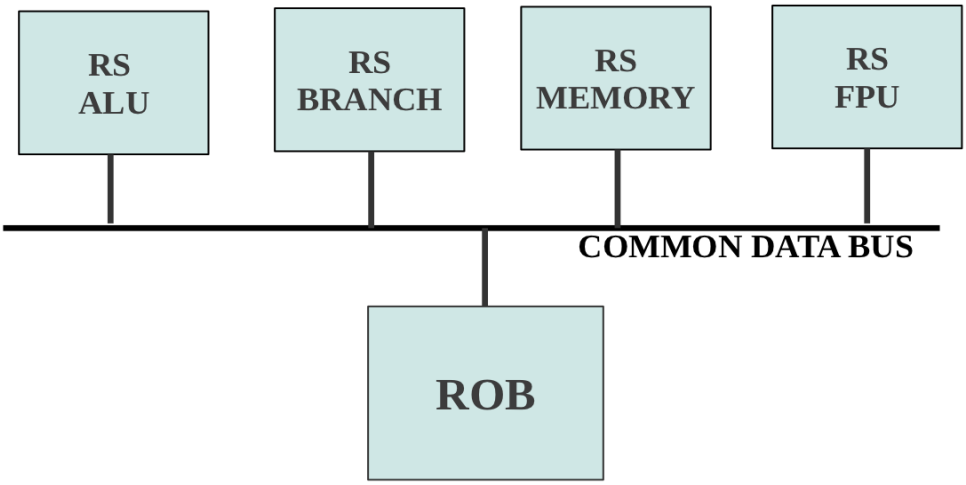


Figure 3.6: Execute and Write back Phase

All the functional units were modified so that the results are available at the output until they are driven onto the common data bus. The design also has result registers. The presence of result register divides the execution delay and the write-back delay into two clock cycles, thereby eliminating a potential critical path. The results are held in the result register until common data bus is available for broadcast.

3.2.3.1 Reservation Station

Distributed reservation station scheme is adopted for the design i.e., each execution unit has its own reservation station. *ZBus interface is provided for all reservation stations to facilitate data forwarding through Z Bus.* The reservation station holds the instructions to be dispatched to the execution unit. All the reservation stations have similar organization except for minor changes depending on the execution unit. The corresponding execution unit is also instantiated as a sub module in reservation station.

The detailed diagram of the reservation station with its interfaces is as shown in figure3.7

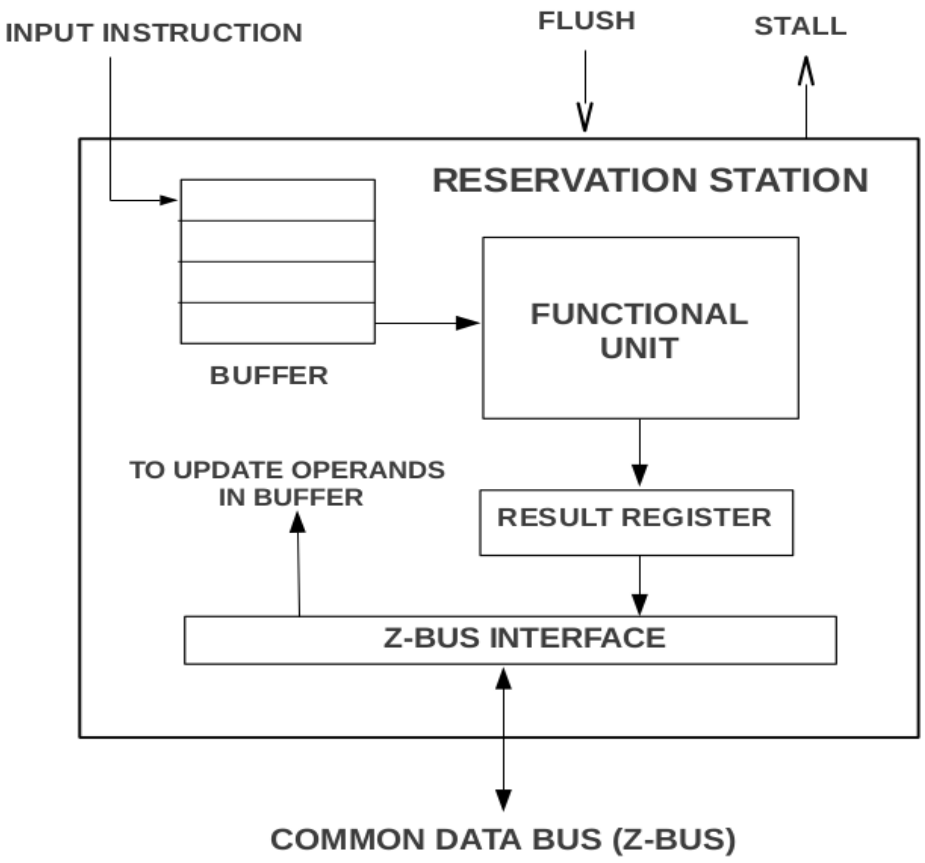


Figure 3.7: Reservation Station

The instructions issued are stored in FIFO manner in Reservation station buffer. Entries in the buffer for an instruction may have the actual operands or the reorder buffer entry number (rob_number) of the instruction which updates the operand value. The reservation station dispatches the instruction to the execution unit only when actual operands are available. So, the reservation station continuously senses the common data bus (ZBus) and checks the rob_number of the instructions which are being written on the bus and updates the operands for the instruction in its buffer, if a match occurs.

As mentioned, the presence of result registers (implemented as FIFOs in Execution units) divides the execution delay and the write back delay into two clock cycles, thereby eliminating a potential critical path.

The earlier design had two methods *OperandForwarding* and *forwardOperand* to facilitate operand forwarding from reorder buffer. In the modified design no forwarding of operands is needed from reorder buffer as they are forwarded through Common Data Bus in write phase (rule rl_forwarding_while_write).

The reservation stations were modified so that the execution units are released only after the result has been driven on CDB by the reservation station. Minor changes were made also to the execution units.

The ZBus package was found to be not supporting tagged unions. Hence the entire design was modified accordingly to avoid use of tagged unions wherever needed. The file ***anupama_types.bsv*** was also modified accordingly.

3.2.3.2 Data Forwarding

Figure: 3.8 illustrates the data forwarding mechanism among the reservation stations through Common Data Bus (CDB).

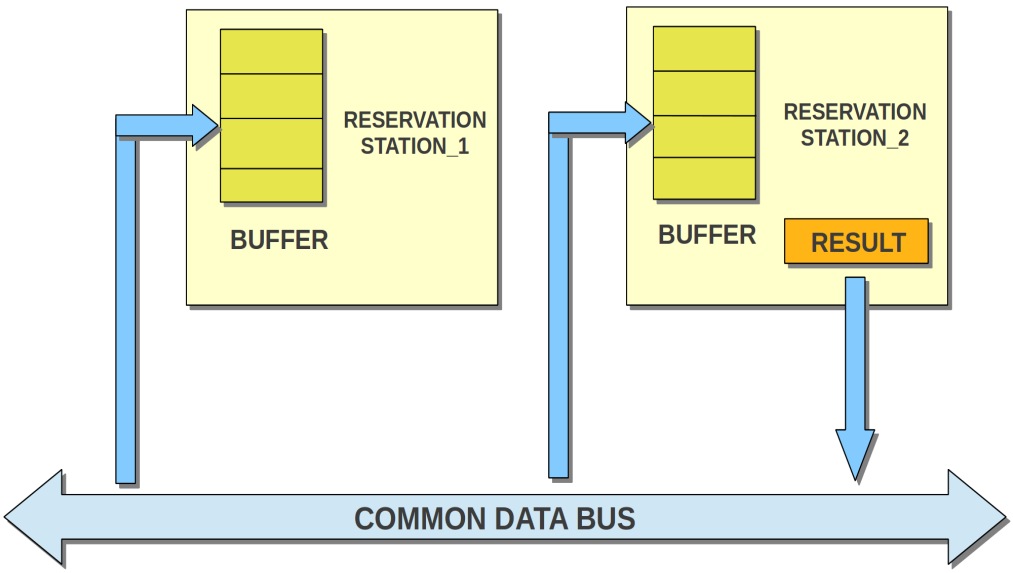


Figure 3.8: Data Forwarding

Whenever a reservation station gets a chance to drive result of instruction execution on CDB, it puts the result on the bus along with the ROB number of the corresponding instruction. All the reservation stations read the data on CDB for updating operands of instructions waiting in their buffers. If a match occurs, the reservation station updates the value of the operand with the data read from Common Data Bus (CDB).

3.2.3.3 Bluespec Implementation details

The bluespec implementation details of Execute and Write back phase is as shown in fig3.9.

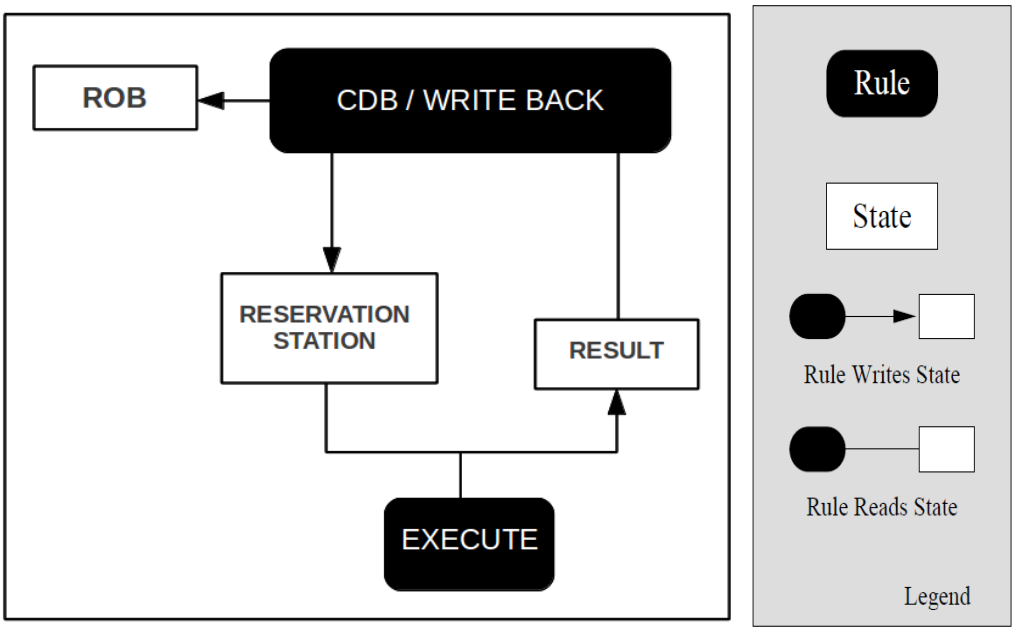


Figure 3.9: Write back: bluespec implementation

Rules corresponding to execution unit writes into its result register. The rules corresponding to Write back (***rl_drive_zbus, rl_forward_while_write***) fires only when result register of at least one execution unit has data to be sent on the bus. The rule drives data from only one execution unit based on the priorities. So, the rule reads from the result register of a reservation station and writes back / forwards to all the reservation stations. The rule also writes into the corresponding entry in ROB.

3.2.4 Commit Phase

The commit phase is similar to the earlier design, except that, *there is no need to forward results to reservation stations while committing an instruction*. The results are forwarded directly from the corresponding reservation through the Common Data Bus. So, there is no need to forward the results while committing an instructions. The commit stage checks for exceptions and updates the architectural state of the processor.

Figure 3.10 illustrates the operations in Commit stage.

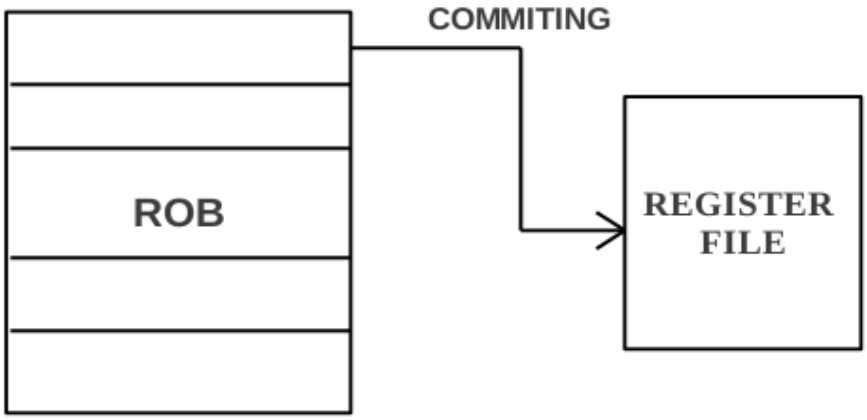


Figure 3.10: Commit Phase

To maintain the in-order commit flow, only those instructions reaching the head of the Reorder buffer are committed. Before committing the instruction at the head, first it checks if any exceptions are raised. If any exceptions have been raised, the program counter is reset to the new address pointing to the ISR,the PC and PSW are pushed on to the stack and all the units including the remaining filled entries of the ROB are flushed.

3.2.4.1 Bluespec Implementation details

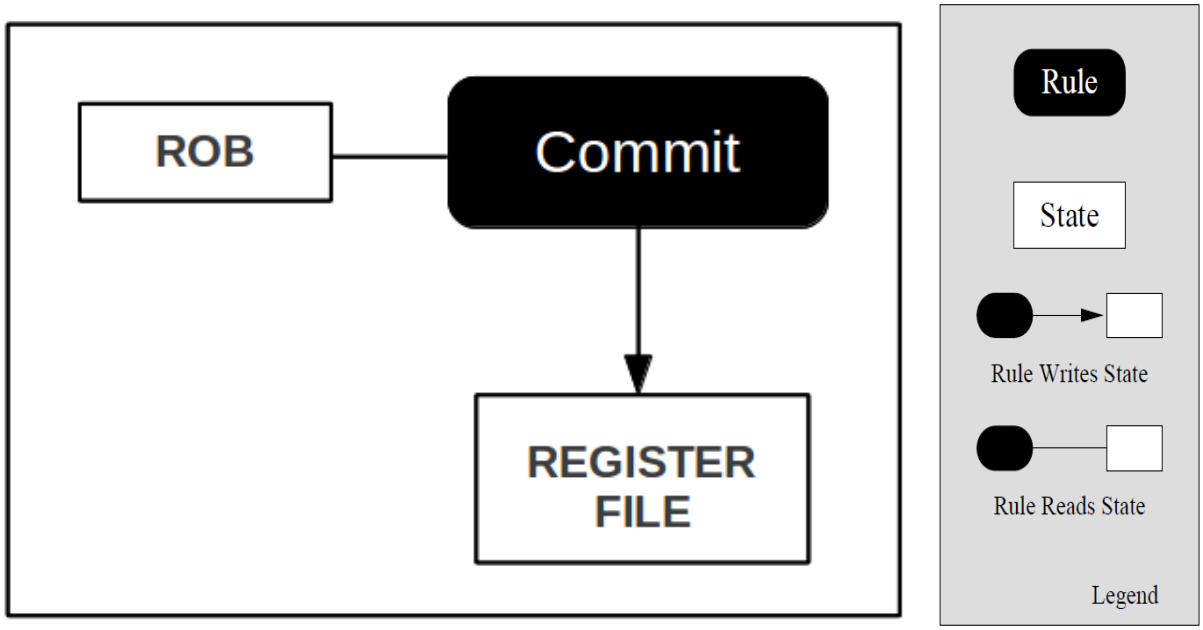


Figure 3.11: Commit Phase: bluespec implementation

The bluespec implementation details of Commit phase is as shown in fig3.11. The commit rule (***rl_commit***) fires only when the head of the ROB has an instruction ready to be committed. The rule reads from the head of the ROB and writes into the register file. *The Commit rule(***rl_commit***) fires only when the processor is not handling an exception.* Each cycle, the Commit rule drains an entry from the ROB and checks the error code. If the error code signals an exception, the commit rule will flag that the processor is now in exception handling mode.

3.3 VERIFICATION

3.3.1 Verification Setup

The set up for verification process is shown in fig: 3.12.

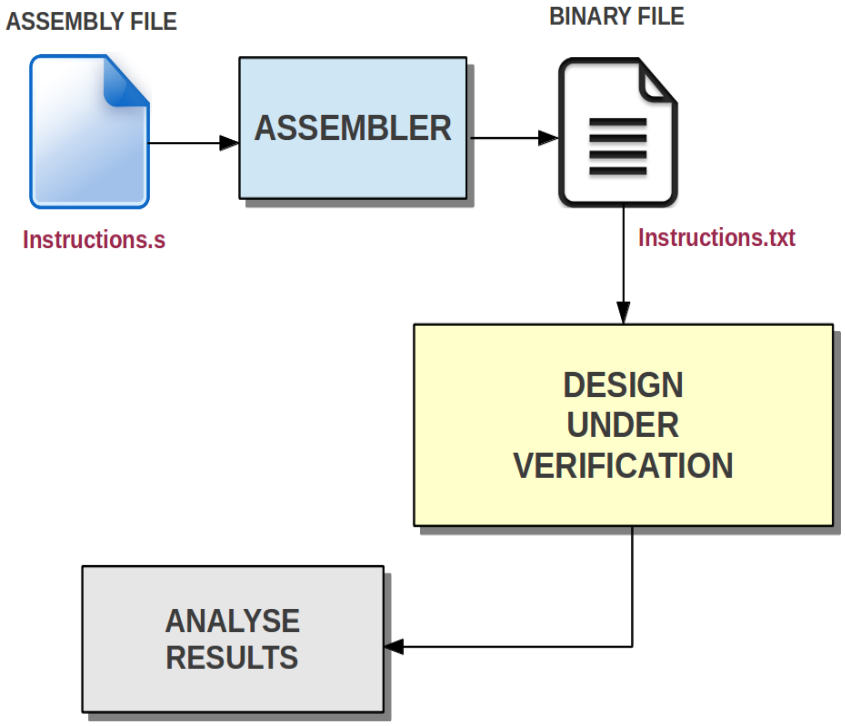


Figure 3.12: Verification Setup

Bluespec Compiler converts the design in BSV to synthesizable Verilog code. Questa simulator was used to verify the various functionalities of the design.

The functionality of the design was verified using a test bench which provides a set of instructions composed of various test scenarios to the main module. ***display*** statements

were embedded/included in the code to monitor the functionality of the modules in various clock cycles. The design was simulated using Questa simulator (Modelsim).

The test instructions in mnemonics form (assembly code) [6] are given to the ABACUS assembler. The assembler gives the instructions in binary form. These are stored in a data file and are used in the test bench for verifying the design.

3.3.2 Verification Strategy

The verification process mainly focused on verifying those functionalities of the processor, which were modified in the new design. They were considered as critical aspects for the verification process.

So, the test scenarios were generated to verify the following aspects of the design.

- Fetching of operands from the Reorder Buffer in Issue stage.
- Data forwarding on Common Data Bus.
- Behavior in case of contention for Common data Bus.
- Bus arbitration.
- Release of Execution units.

3.4 IMLPEMENTATION EVALUATION

3.4.1 Design Challenges

The data forwarding mechanism which was implemented as explained in section - 3.2.3.2 worked for most combinations of the instructions. But while verifying the data forwarding operation, it was found to be not functioning properly. Some instructions were waiting in their reservation stations forever as required operands were not forwarded to them through Common Data Bus. The reason for this is illustrated in figure :3.13

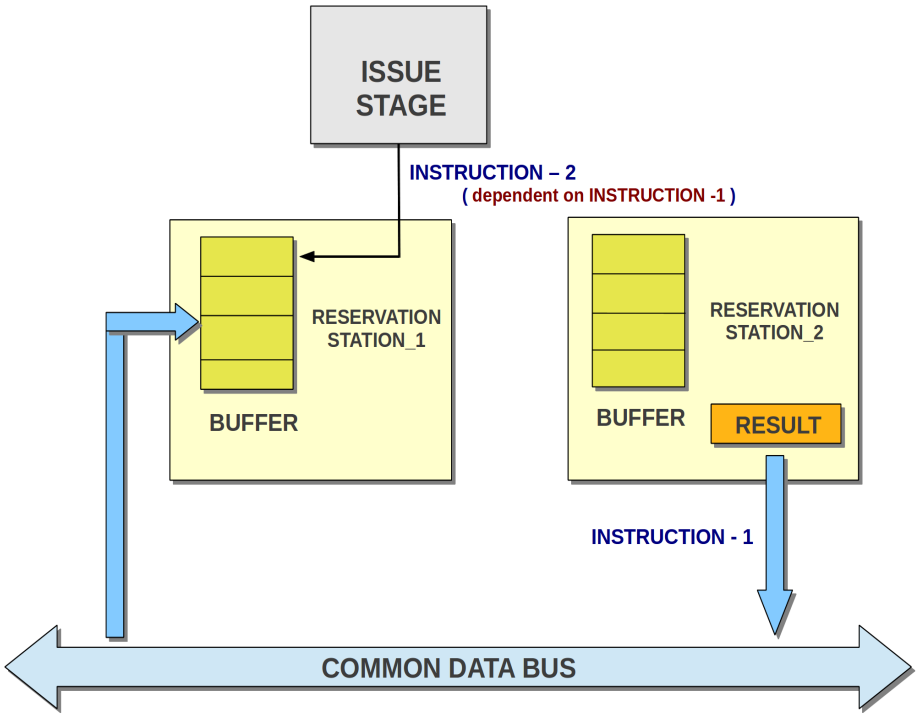


Figure 3.13: Design Challenge : Data Forwarding

As shown in the figure above, if a reservation station writes the result of an instruction execution (*instruction-1*) on CDB and if the Issue stage dispatches another instruction (*instruction-2*) which is dependent on *instruction-1* in same cock cycle, then *instruction-2*

will not get the forwarded data from *instruction-1*. This is because, loading *instruction-2* in reservation station buffer and the data forwarding from *instruction-1* are done in the same cycle.

In order to overcome this limitation, for the reservation station buffer entry, which contains the last updated instruction, the data is forwarded both from the data available on the CDB in the current clock cycle and also from the data forwarded on CDB in previous clock cycle. For this purpose, *a register is used, which holds the data forwarded on the bus in last clock cycle. The data is forwarded to the tail of the buffer from both the Common Data Bus and this register.*

3.4.2 Bluespec Coding Details

Bluespec implementation of entire processor with modified mechanism for out of order execution and data forwarding is as shown in fig 3.14

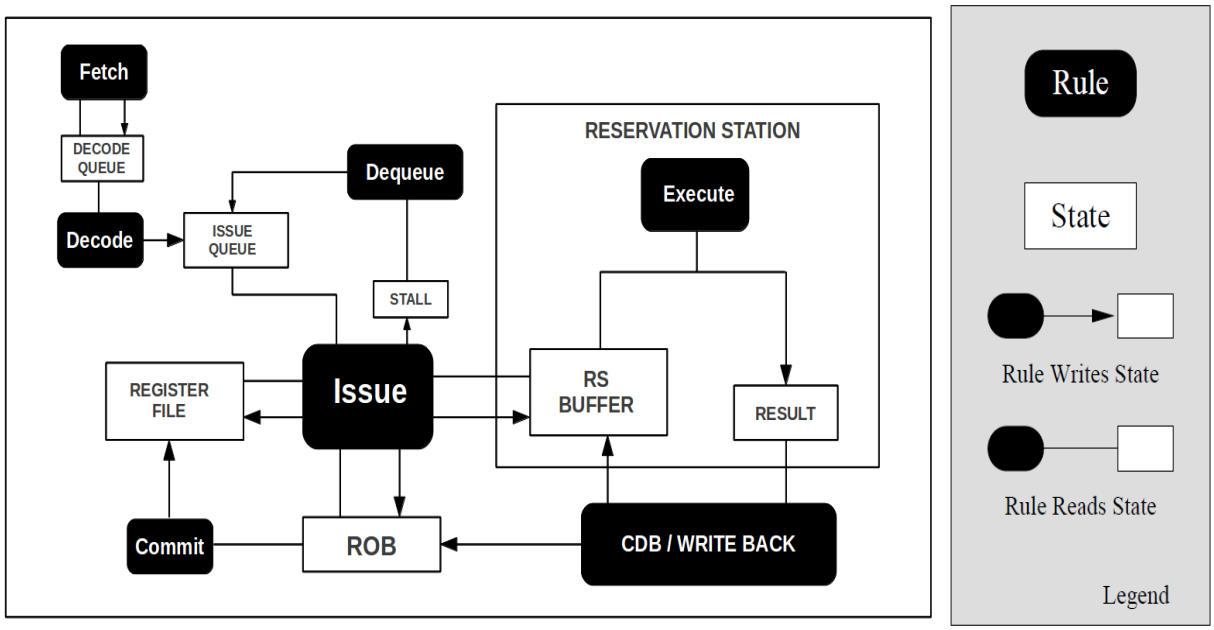


Figure 3.14: Single issue processor with CDB: bluespec implementation

The bluespec implementation details of the earlier design of the processor (with out CDB) are illustrated below in fig: 3.15.

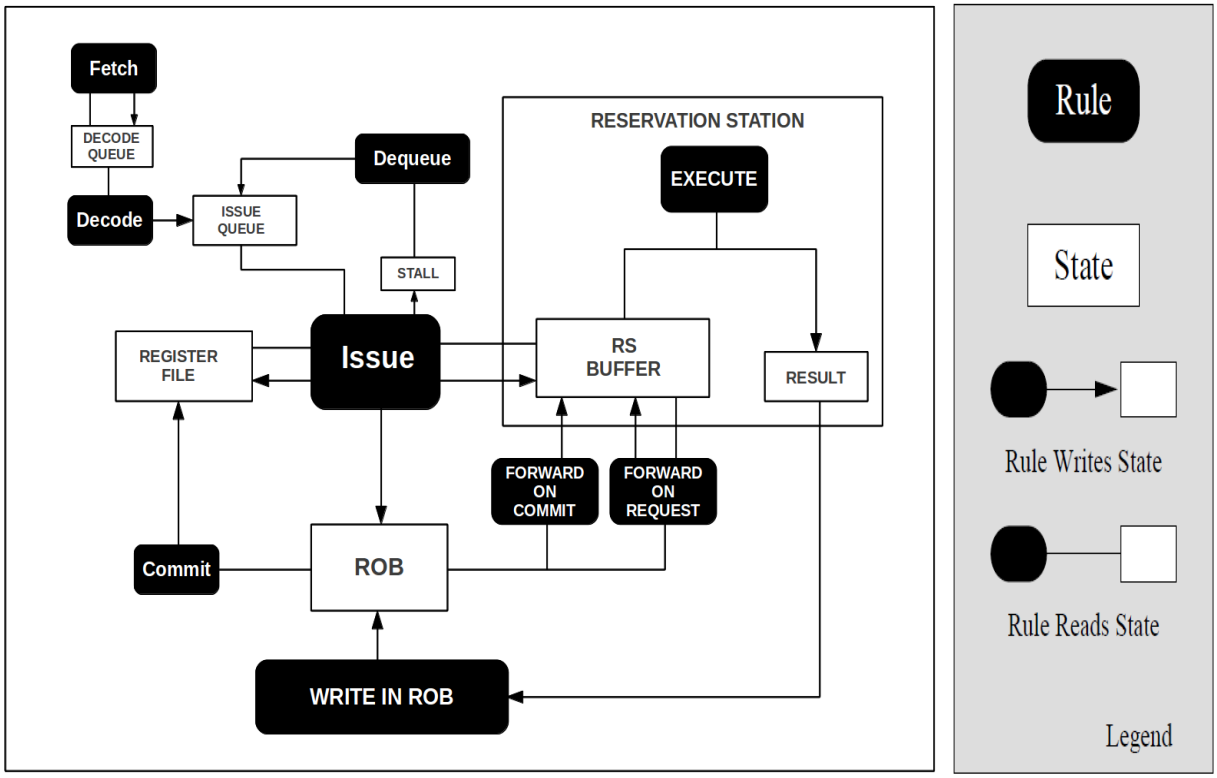


Figure 3.15: Earlier design of the processor : bluespec implementation

From the bluespec implementation of the two processors represented in Figure: 3.14 and Figure: 3.15, the difference in the data forwarding mechanism can be observed.

In the earlier design the data is forwarded to reservation stations from ROB as indicated in figure:3.15. But in the new design (Figure: 3.14) the data forwarding is done through CDB and there is no data forwarding from ROB. Minor differences can also be observed in the Issue rule.

Files associated with the design:

Top Module:

Tomasulo.bsv - (1500 lines)

Reservation Stations:

rs_arith.bsv - (800 lines)

rs_memory.bsv - (700 lines)

rs_branch.bsv - (500 lines)

rs_floating.bsv - (650 lines)

3.4.3 Synthesis Report

The design has been synthesized using Xilinx ISE for ***Virtex 6 XC6VLX240T-FF1156***. All default settings were used. The design strategy was set to *"optimization for speed"*. The slice utilization and timing summary are provided below.

Device Utilization Summary :

Selected Device: 6vlx240tff1156-1

Slice Logic Utilization:

Number of Slice Registers : 21049 out of 301440

Number of Slice LUTs : 33029 out of 150720

Number used as Logic : 32555 out of 150720

Number used as Memory : 474 out of 58400

Number used as RAM : 474

Timing Summary :

Minimum period: 7.095 ns

Maximum Frequency: 140.944 MHz

Minimum input arrival time before clock: 5.706ns

Maximum output required time after clock: 4.471ns

Maximum combinational path delay: 3.198ns

CHAPTER 4

DUAL ISSUE

This chapter describes the Implementation of Dual instruction issue in the single issue processor with Out Of Order Execution (with CDB). It starts with Microarchitectural description followed by implementation details of various phases of the design. The verification strategy, Design challenges and Synthesis results are presented in the later sections.

4.1 MICROARCHTECTURAL DESCRIPTION

The performance of the processor can be improved further if it can issue multiple instructions in a clock cycle. Modifications are needed in all stages of the pipeline to convert the existing single issue machine into a dual issue one. Major changes have to be incorporated in the Issue stage. The back end of the pipeline also need to be modified to support forwarding results of two instructions on CDB simultaneously in same clock cycle and also to commit two instructions in single clock cycle. *The ALU is duplicated to reduce structural hazards, which helps in issuing pair of arithmetic instructions in the same clock cycle with out stalls*.

Two instructions have to be fetched and decoded in single cycle. So,the decoder is duplicated to decode two instructions simultaneously. The fetch unit fetches two instructions every clock cycle and feeds the decoders through buffers. The decoded instructions goes to the issue stage, where required operands are fetched and all the dependencies between two

instructions are handled before dispatching the issue packets to the corresponding reservation stations. The Architecture of the dual issue processor is shown in the figure -4.1.

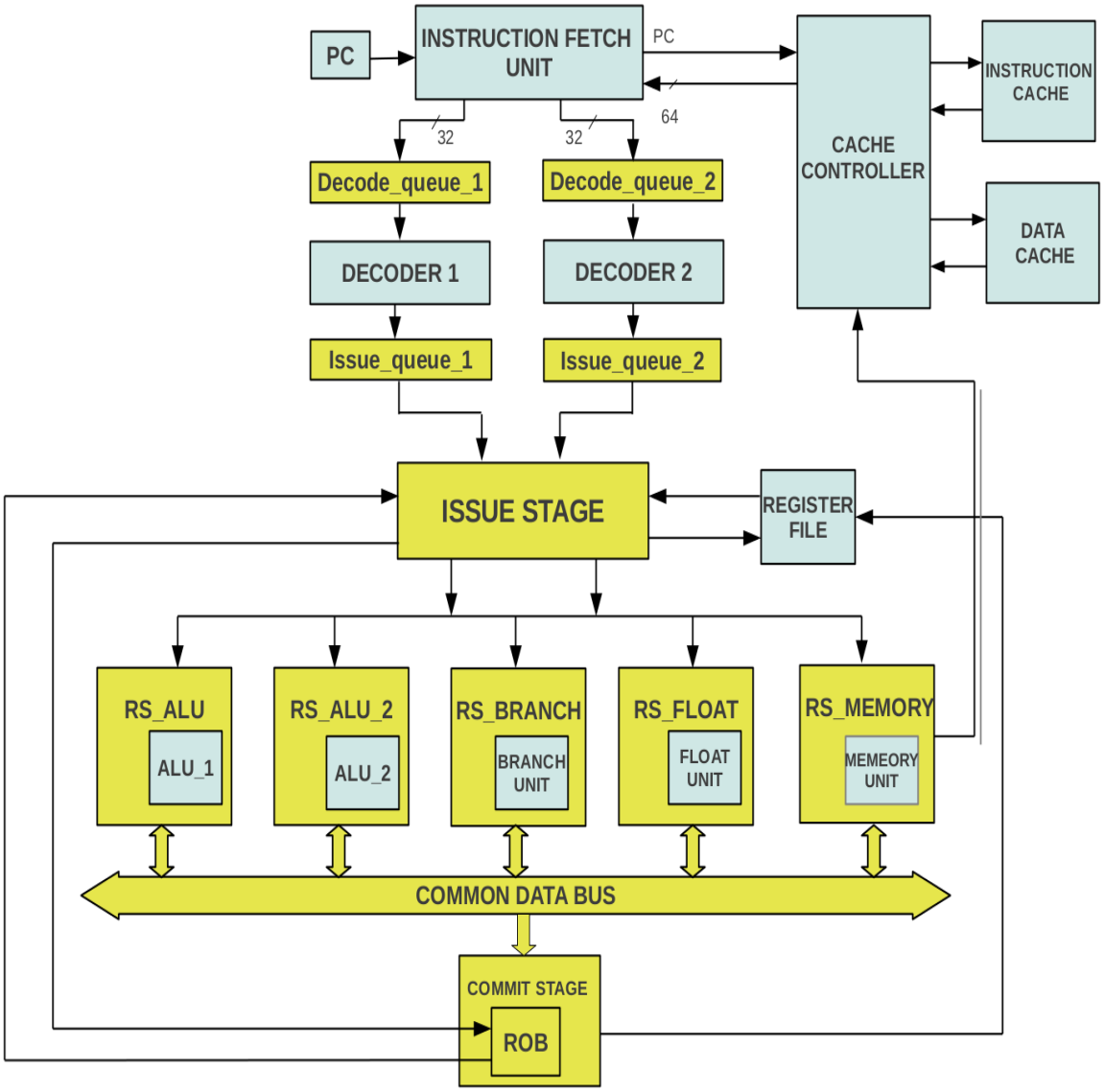


Figure 4.1: Microarchitectural organization of Dual issue processor

As mentioned before, the design need to be modified in almost all stages of an instruction execution cycle. The design methodology and implementation details of various phases of instruction execution for the dual issue processor are explained in the subsequent sections.

4.2 IMPLEMENTATION

4.2.1 Issue Stage

The functionality of the issue stage can be divided into three tasks. They are :

1. Operand Fetch (OF).
2. Issue Packet Formation (IPF).
3. Dispatch of Issue Packets (DIP).

For dual issue, the above tasks cannot be treated independently for the two instructions as the instructions can be dependent on each other. So in the issue stage, the two instructions have to be treated together, exploring possible parallelism in the tasks for reducing performance degradation due to longer data path due to complex logic.

Figure 4.2 illustrates the implementation of issue stage.

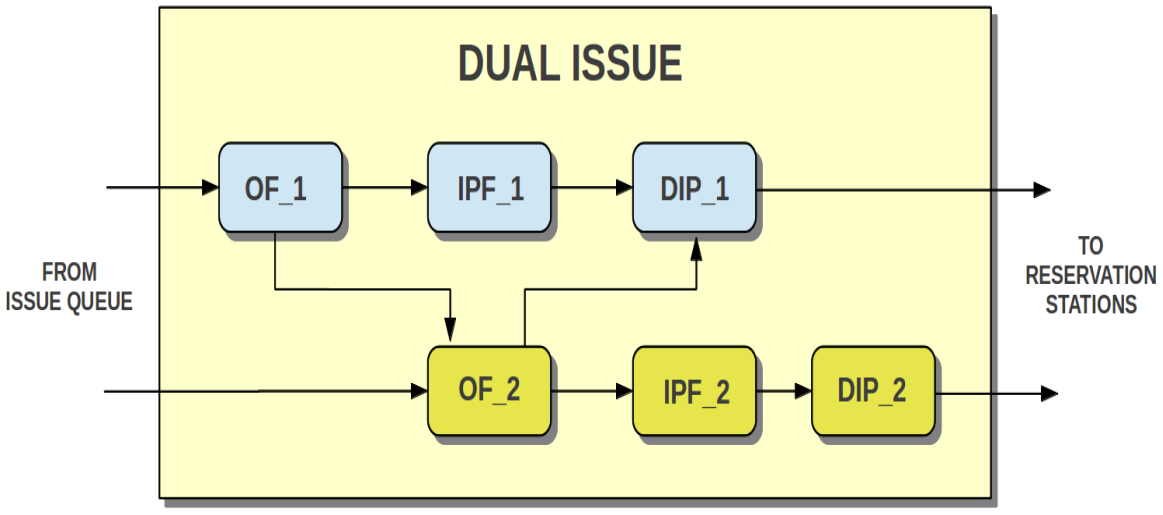


Figure 4.2: Issue stage of Dual Issue Processor

The issue logic checks for availability of at least two empty slots in ROB to be alloted for the two instructions being issued. If ROB has required slots, the issue stage proceeds further. Otherwise, the issue operation is stalled until slots become available. The implementation details of the issue stage are explained in the following sections.

4.2.1.1 Operand Fetch - First Instruction

The required operands for the first instruction are fetched from register file or ROB. The Operand fetch for the First instruction is done in the same way as a single issue machine. Additionally, the destination registers of the first instruction are identified in the Operand Fetch stage itself, as this information is needed for handling dependency, While fetching operands for second instruction.

4.2.1.2 Operand Fetch - Second Instruction

While fetching operands for the second instruction, its dependence on first instruction is handled before looking for operands in the register file and ROB. *If the source operand of the second instruction is same as one of the destination registers of the first instruction, then the issue unit sends the ROB number alloted for the first instruction in place of the actual operand for the second instruction.* Hence, it reads its actual operand later from Common Data Bus or Reorder Buffer, after completion of execution of first instruction. Hence, the possible RAW hazard as a result of issuing two instructions simultaneously is handled.

4.2.1.3 Issue Packet Formation - First Instruction

The information / data to be sent to the corresponding reservation is formed as packets in the issue stage. The issue packets contain information required for instruction execution like the actual operands (or corresponding ROB number), PSW, ROB slot alloted (tail of ROB) etc,. The information from the Operand fetch stage is used to form the issue packets.

4.2.1.4 Issue Packet Formation - Second Instruction

The Issue packet formation for the second instruction is done in the same way as done for the first instruction. Additionally, the information regarding ROB number alloted is not the same in all cases. If the first instruction is a NOP, then no ROB slot is alloted for the first instruction and hence the ROB number assigned for the second instruction will be different (*(tail) of ROB instead of (tail+1)*). Some exceptional conditions like this need to taken care while forming Issue packets for the second instruction.

4.2.1.5 Dispatch Of Issue Packets - First Instruction

Once the issue packets are formed, the packets are dispatched to the corresponding reservation station, if it has free entries. If the reservation station has no free entries, then the issue stage is stalled and waits until free entries are available. The OF and IPF for the instruction are done in every cycle (to get latest updated values for operands) until the Packets are dispatched to the corresponding reservation station. ROB number is alloted to the instruction and the corresponding entry in ROB is tagged as alloted.

The destination registers of the instruction are invalidated and tagged with the ROB number of the instruction. *However, if any destination register of the first instruction is same as one of the destination registers of the second instruction, then the tagging (with ROB number of first instruction) is not done to avoid multiple writes to the corresponding register*. Information from the Operand fetch phase of the second instruction is used for handling this case.

4.2.1.6 Dispatch Of Issue Packets - Second Instruction

Dispatching the issue packets of the second instruction is done in the same cycle as the first instruction, only if both instructions are of different kind. The exception for this is the case when both instructions are Arithmetic. The Arithmetic unit and the corresponding reservation station are duplicated. Hence, two arithmetic instructions can be issued in the same cycle.

Also, the instruction issue has to be done in-Order, to maintain Program Order. Hence, the issue packets of the second instruction are dispatched only if the first instruction is also ready for dispatch or was dispatched in earlier clock cycle. ROB entry is alloted and the destination registers are tagged with the ROB number of the instruction.

4.2.1.7 Bluespec Implementation Details

The bluespec implementation details of the issue stage in dual issue processor are illustrated in Figure :4.3

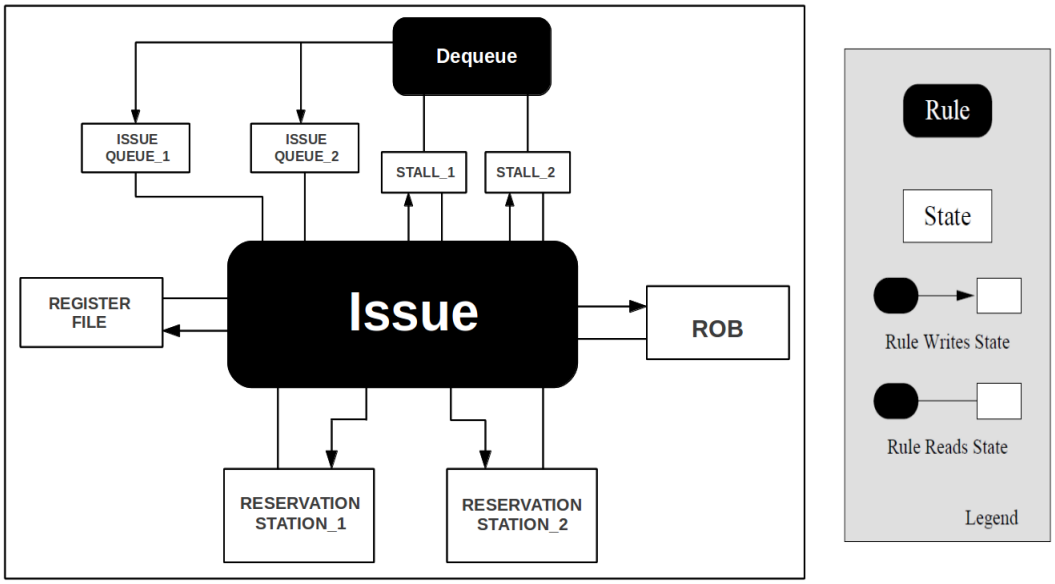


Figure 4.3: Bluespec Implementation of Issue stage in Dual Issue Processor

The issue rule (***rl_issue***) in dual issue processor is responsible for getting decoded instructions from the two Decoder units and deciding which functional units to issue the instructions to. This rule determines the type of the instruction and reads the value of the operands from the Register File after checking for dependency between two instructions. It also does a write to the main register file that updates the destination register to be a tag for the result of this instruction.

In addition, the issue rule checks the result that is currently waiting in ROB for being committed, if the updated value is not yet available in the register file. Each instruction is also issued a tag which is an entry in ROB. This tag is given for renaming purposes. In the end, the issue packets are queued into the appropriate Functional Unit's reservation station.

The issue rule (***rl_issue***) fires only when the ROB has at least two free entry in ROB and both the issue queues have decoded instructions. Issue rule reads from register file and ROB for operands. It also writes into register file and ROB. The issue rule reads the state of reservation station for free entries and writes to it. It checks the status of signals ***stall_1***and ***stall_2*** to know the issue status of previous cycle. It also updates the signals *stall_1* and *stall_2*, at the end of the cycle, which represents the issue status of the two instructions being issued. These signals are read by the dequeue rule (***rl_dequeue***) to dequeue the issue queues.

4.2.2 Execute and Write back

The Execute phase of the design doesn't need significant modifications for supporting dual issue. As mentioned earlier, the Arithmetic execution unit and corresponding reservation station is duplicated to support issuing two Arithmetic instructions in the same cycle.

The write-back phase of the design is modified to support writing results of two instructions in the same clock cycle. *Hence the size of the common data bus is increased by duplicating the existing common data bus.* The data forwarding logic in reservation stations also need to be modified to update the reservation station buffers with results from the two CDBs.

To avoid contention for the Common data Buses, the reservation stations are assigned fixed priorities for each common data bus. The priorities assigned for two CDBs are in the opposite order.

4.2.2.1 Bluespec Implementation Details

The bluespec implementation details of Execute an Write-back phase is as shown in figure- 4.4.

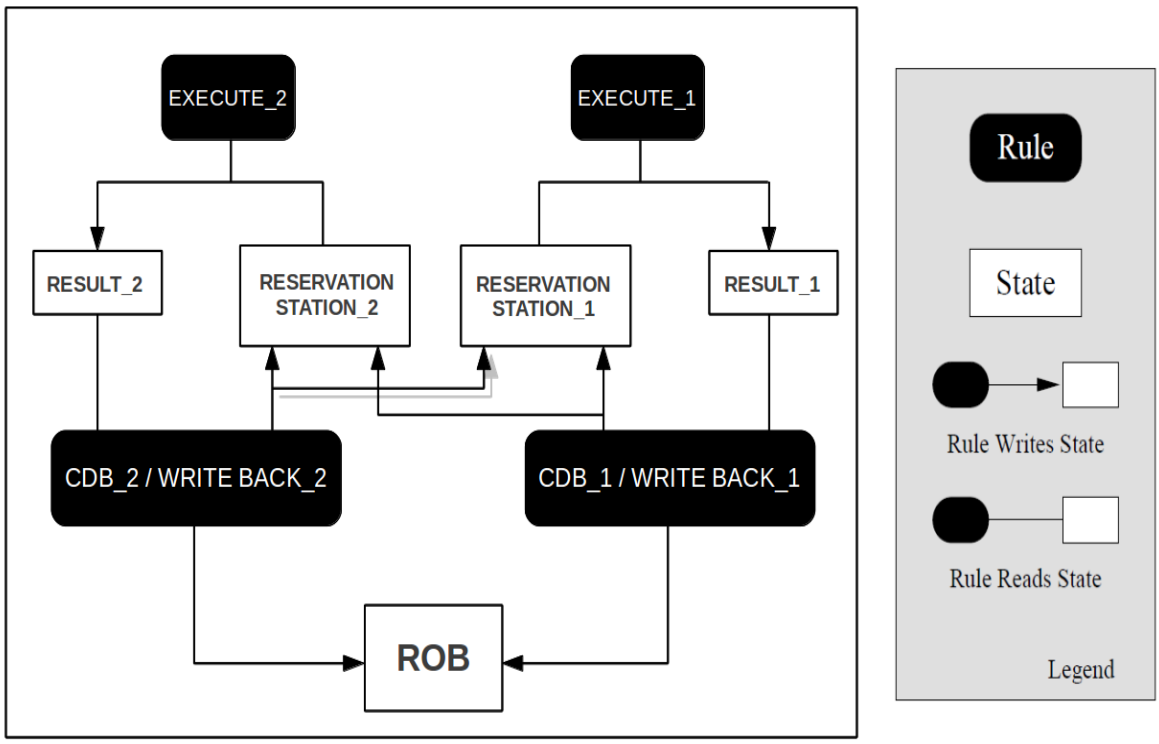


Figure 4.4: Bluespec Implementation of Write back stage in Dual Issue Processor

Each write back rule broadcasts one result out of the available ones in the various reservation stations. The result from reservation station with highest priority is driven on CDB 1 by first write back rule and the next priority instruction on CDB 2 by the other write back rule. The write back rules also update all the reservation stations and the corresponding ROB entries.

4.2.3 Commit Stage (Dual Commit)

The commit stage has to commit two instructions in single clock cycle. It checks if If the oldest two instructions in the ROB are ready to be committed. Commit stage writes results into the corresponding destination registers of the two instructions. The commit logic is replicated for committing two instructions simultaneously in the same clock cycle. Not many modifications are needed in the commit stage as all the dependencies are handled in the issue itself. However, exceptions and branch mis-predictions have to be handled properly.

If the first instruction has a branch mis-prediction or an exception, then the second instruction is not committed. Also, if the first instruction is memory instruction (store operation), then the second instruction is committed only after it finishes committing the first instruction, because some memory instructions (store operation) take multiple cycles to complete [7]. Hence, the commit stage has to ensure that exceptions like *" bus error"*] are not present before committing the second instruction. Also, the existing memory execution unit supports committing only single instruction at a time. Hence, dual commit is not possible if the first instruction is a memory instruction.

4.3 VERIFICATION

4.3.1 Verification Strategy

The strategy for verification is to generate the test cases which verifies the specific features of dual issue processor. The test cases were generated to test the following critical aspects of the design.

Issue Stage :

- Second instruction dependent on First instruction.
- Two instructions having same destination.
- Instructions for same reservation station.
- Handling of stalls in issue stage.
- Combination of different types of instructions.

Write back Stage :

- Contention for Common Data Bus (CDB).
- Bus arbitration.
- Data forwarding from two buses.
- Combination of different types of instructions.

Commit Stage :

- Combination of different types of instructions.
- Handling exceptions.
- Handling branch mis predictions.

4.3.2 Sample Test Cases and Results

One of the Sample test case for verifying the issue stage of dual issue processor and the corresponding output of the Questa simulator are shown below.

Sample instructions :

- addu %r3, % r1, % r2
- or %r2, %r4, % r2

Output Of Questa Simulator :

- Fetching Operands for first Instruction : 00820c10
- OF-1st instruction : Fetched data from reg file: 1 for op2
- IPF-1st instruction: ARITHMETIC INSTRUCTION
- Fetching Operands for second Instruction : 0a050840
- *OF-2nd instruction :op1 dependent on first instruction*
- OF-2nd instruction : Fetched data from reg file: 4 for op2
- IPF-2nd instruction: ARITH INSTRUCTION
- DISPATCH-1: ARITHMETIC INSTRUCTION
- *DISPATCH-1: same destination*
- DISPATCH-2: ARITHMETIC INSTRUCTION
- *DISPATCH-2: register: 2 tagged with ROB: 3*
- Dequeuing instructions from buffers

As the second instruction is dependent on the first instruction, the corresponding operand is assigned the ROB number of the first instruction. Also,as both the instructions have same destination register, register invalidation (tagging with its ROB number) is not done for the first instruction.

4.4 IMPLEMENTATION EVALUATION

4.4.1 Design Challenges

Issue Stage:

1. For the case when first instruction has been issued and the second one is stalled, the issue stage should be designed such that, the first instruction is not been issued again while trying to issue the second instruction in later cycle. *Also, dependency on first instruction need not be taken into account while Fetching operands for the second instruction in the later cycles as the corresponding registers and ROB were already tagged.* To handle this, two signals ***stall_1*** and ***stall_2*** are used, which hold the issue status of the two instructions inn previous clock cycle.

- Stall_1 = True =>First instruction was not issued.
- Stall_1 = False =>First instruction was issued.
- Stall_2 = True =>Second instruction was not issued.
- Stall_2 = False =>Second instruction was issued.

The state machine describing the operation is as shown in figure 4.5

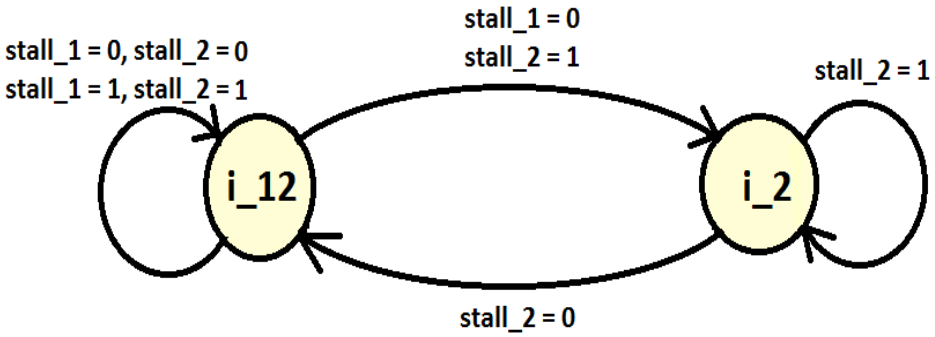


Figure 4.5: Handling Stalls (Issue stage) in Dual Issue Processor

- **i_12** : Both instructions can be issued.
- **i_2** : Only Second instruction has to be issued.

2. For the case in which the first instruction is being issued and second instruction is stalled, we should not check if both instructions have the same destination register fields, while tagging the destination registers for first instruction in dispatch phase.

3. The decode queues which feed the issue stage should be dequeued simultaneously only after completing the issue of both the instructions. *If the two decode queues are dequeued independently, the later instructions may get issued before issuing the instructions preceding them.*

The state machine representing this operation is show in figure 4.6.

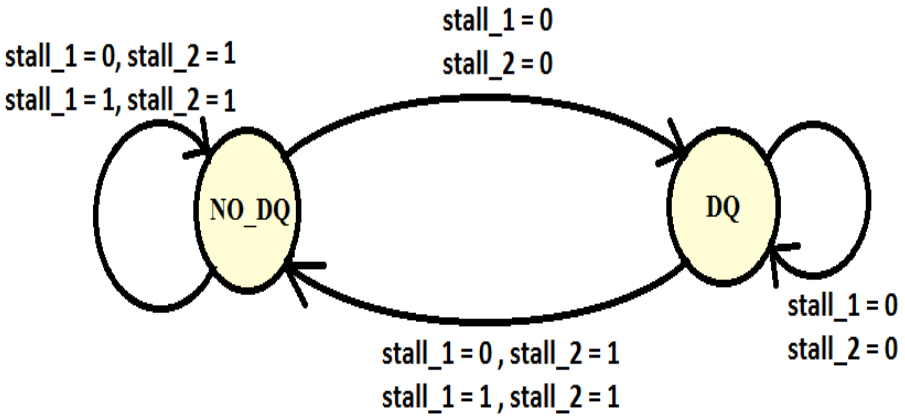
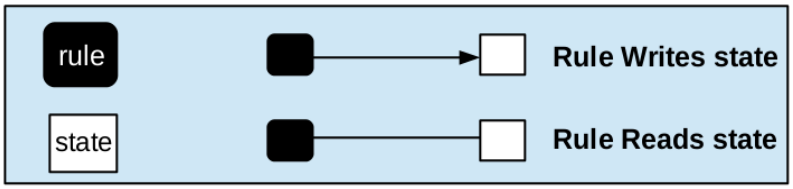


Figure 4.6: Dequeuing Issue queues in Dual Issue Processor

- **NO_DQ** : Issue Queue cannot be dequeued.
- **DQ** : Issue Queue can be dequeued.

4.4.2 Bluespec coding Details

Bluespec implementation of entire dual issue processor with modified mechanism for out of order execution and data forwarding is as shown in fig 4.7.



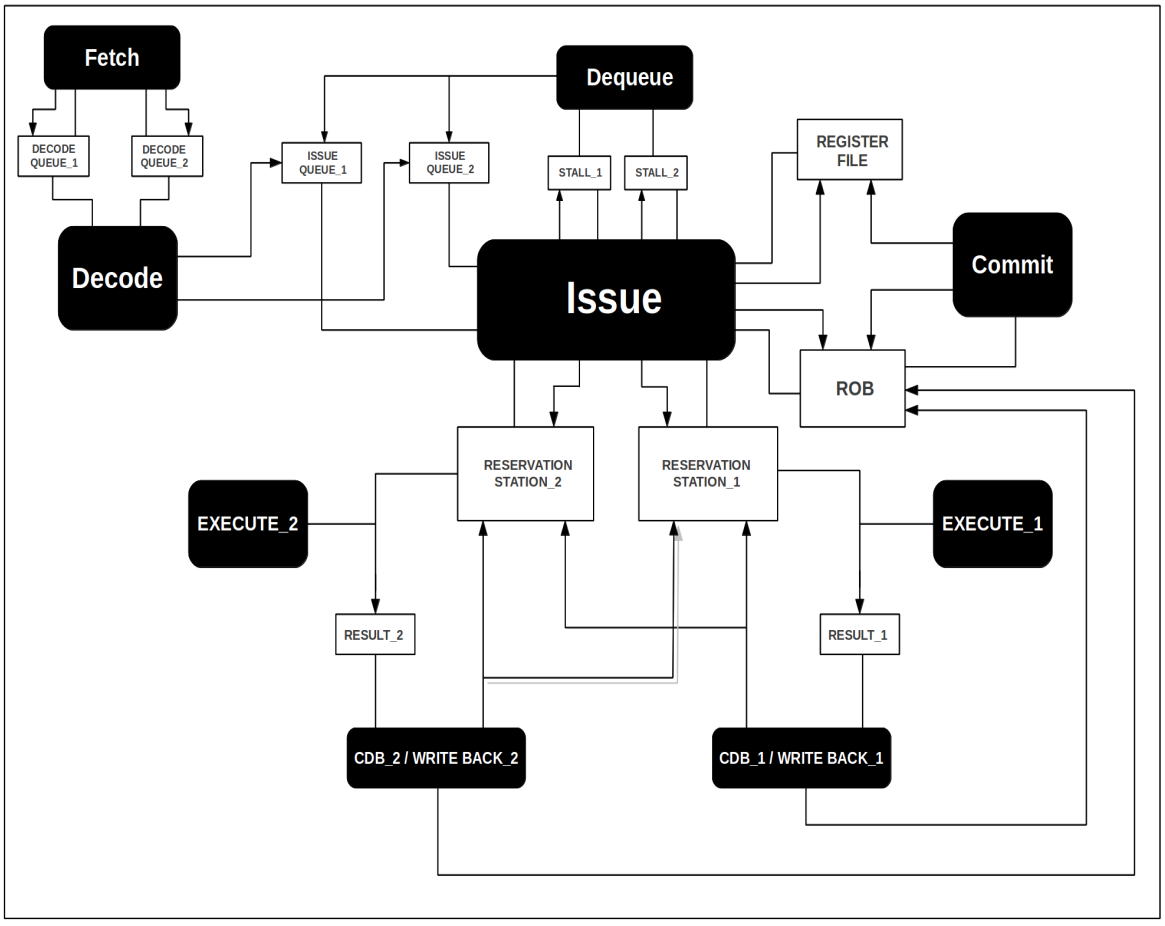


Figure 4.7: Bluespec Implementation of Dual Issue Processor

The decode rule (***rl_decode***) reads two instructions from the *decode_queues* and decodes the instructions. It puts the decoded instructions in *issue_queues*. The state of *issue_queues* is read by issue rule (***rl_issue***). The operation of ***rl_issue*** was explained in earlier section.

Dual Write back is achieved through two different sets of rules for driving the two CDBs. The Write back rules (***rl_forward_while_write***) forwards the results from its CDB to all the reservation stations.

Commit rule (***rl_commit***) does the dual commit operation. If two instructions at the head of the Re-order buffer have valid results, then ***rl_commit*** updates the register file with the results of both the instructions simultaneously.

Files associated with the design:

Top Module:

Tomasulo.bsv - (2600 lines)

Reservation Stations:

rs_arith.bsv - (1200 lines)

rs_memory.bsv - (1000 lines)

rs_branch.bsv - (750 lines)

rs_floating.bsv - (1050 lines)

4.4.3 Synthesis Report

The design has been synthesized using Xilinx ISE for ***Virtex 6 XC6VLX240T-FF1156***. All default settings were used. The design strategy was set to *"optimization for speed"*. The slice utilization and timing summary are provided below.

Device Utilization Summary :

Selected Device: 6vlx240tff1156-1

Slice Logic Utilization:

Number of Slice Registers : 26823 out of 301440

Number of Slice LUTs : 44411 out of 150720

Number used as Logic : 43937 out of 150720

Number used as Memory : 474 out of 58400

Number used as RAM : 474

Timing Summary :

Minimum period: 8.224 ns

Maximum Frequency: 121.594 MHz

Minimum input arrival time before clock: 6.217 ns

Maximum output required time after clock: 3.823 ns

Maximum combinational path delay: 3.481 ns

CHAPTER 5

Conclusion and Future Work

The existing Out Of Order execution mechanism in the processor has been modified by implementing the data forwarding among reservation stations through Common Data Bus. The necessary modifications were made in some stages of the pipeline to support the data forwarding through Common Data Bus. On synthesizing the Modified design on FPGA, it gave improved clock frequency and less resource utilization compared to the earlier design.

In order to utilize the functional units effectively, Dual instruction issue has been implemented. Modifications were made to the design to support dual instruction issue. All the necessary modifications were made starting from decode stage to the commit stage of the processor. The Arithmetic Execution unit was replicated to improve the performance by reducing stalls due to structural hazards in the dual issue processor.

Possible Directions for Future Work: The existing design assumes that the fetch unit fetches two instructions in a clock cycle. This work can be carried out further to implement fetching of two instructions from the memory in single clock cycle. The fetch stage, cache controller and some other modules of the processor have to be redesigned to support fetching multiple instructions in a clock cycle. The Memory execution unit has to be modified to support committing two memory instructions in same clock cycle. The Design can be further modified to support even larger issue widths. However, the effect on clock frequency and area utilization has to be taken into account while choosing the optimal issue width.

Apart from the Arithmetic execution unit, some other functional units can be replicated to improve the performance further. However, this results in increased area utilization and power Consumption by the processor. Rigorous verification of the entire design has to be carried out before implementing it on ASIC.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture (5th Edition)*. Morgan Kaufmann, 2012.

[2] R. S. Nikhil and K. Czeck, *BSV by Example*. Bluespec, Inc, 2010.

[3] *Report on Bluespec ANUPAMA*.

[4] J. P. Shen and M. H. Lipasti, *Modern Processor Design - Fundamentals of Superscalar processors*. TATA McGraw-Hill Publishing Company Private Limited, 2005.

[5] Bluespec, Inc, *Bluespec System Verilog Reference Guide*, revision: 17 ed., 2012.

[6] *ABACUS Processor based SOC - User Guide*.

[7] *ANUPAMA reference Manual*.