Design and Implementation of Quad Issue and Out of Order Execution for Multithreaded Superscalar RISC Processor

A Project Report

submitted by

SENTHIL KUMAR R

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 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **Design and Implementation of Quad Issue and Out of Order Execution for Multithreaded Superscalar RISC Processor**, submitted by **SENTHIL KUMAR R**, 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 deepest gratitude to my guide, ***Dr. V. Kamakoti*** for his valuable guidance, encouragement and advice. 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. Deleep R.Nair*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to project team members Neel, Rishi Naidu, Arjun ,Naveen, Chidambaranathan, Sachin, Sarath, Chaitanya, Keerthi for their help and support.

ABSTRACT

KEYWORDS: Quad Issue, Centralized Reservation Station, Common Data Bus, Quad Commit

In Modern Superscalar Processors, Instruction issue step has become a primary challenge affecting the performance of the Processor if the instruction issue width is less. Also, in the conventional distributed reservation station, there is no efficient utilization of resources. The Common Data Bus(CDB) can forward the result of at most one execution unit at a time.

My Project work involves implementing a quad issue to increase the Instructions Per Cycle(IPC), implementing a centralized reservation station which can dispatch the instructions from any slot to the execution units, implementing a Common Data Bus which can forward the results of at most four execution units and also implementing a quad commit stage which can commit up to four instructions at a time.

The entire project work is implemented in Bluespec System Verilog and synthesized in Xilinx ISE.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
RISC	Reduced Instruction Set Computer
BSV	Bluespec System Verilog
CDB	Common Data Bus
ROB	Reorder Buffer
RAW	Read After Write
WAW	Write After Write
WAR	Write After Read
PC	Program Counter
HDL	Hardware Description Language
ISA	Instruction set Architecture
ILP	Instruction level Parallelism
ALU	Arithmetic Logic Unit
FPU	Floating Point Unit
BSW	Bluespec Development Workstation

CHAPTER 1

INTRODUCTION

1.1 Overall Micro-Architecture

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Dept. of IIT Madras has been actively involved in building a Superscalar processor for Server applications. The proposed processor is a 64 bit, single core, quad threaded superscalar processor. The processor strictly follows RISC-V Instruction set Architecture (ISA). The entire design of the processor is done using a Hardware Description Language (HDL) called Bluespec System Verilog (BSV).

The CPU core is based on the Tomasulo Algorithm. The Microarchitecture of the CPU core is an Out of order microprocessor that is capable of fetching, decoding and issuing 4 instructions every clock cycle. A Centralized Reservation station is implemented to utilize the reservation station entries in a more efficient manner. The Execution units are duplicated to support for Quad issue. The Common Data Bus (CDB) used for Operand forwarding can forward the results of at most 4 execution units. The Reorder buffers are designed to commit maximum of four instructions in every clock cycle. My project work involves the design of issue, dispatch, CDB and commit stage related to the CPU core.

1.2 Objective

- To design Quad Issue which can issue maximum of 4 instructions of the same thread.
- To design Centralized Reservation Station which can be filled and dispatched from any slot.
- To design a Common Data Bus which can forward the results of at most 4 execution Units.
- To design 4 circular Reorder Buffers (one for each thread) which can commit at most 4 instructions in the same clock cycle.

1.3 Organization of the thesis

Chapter 2 describes about the basics of superscalar processor in which scheduling techniques, out of order execution, Tomasulo algorithm are discussed. It also describes about the HDL called Bluespec System Verilog, its key features, BSV constructs. It finally describes about the proposed microarchitecture of the superscalar processor based on the Tomasulo algorithm.

Chapter 3 starts with design of the instruction queue buffer. It compares the centralized with distributed reservation station. It then talks about the design of the centralized reservation station. It also describes about the Operand fetch mechanism for both single issue and quad issue. Then, it briefly describes about the design of Quad Issue for the proposed microarchitecture of the superscalar processor.

Chapter 4 describes about dispatch mechanism that is implemented in the processor. Then, it describes about design of CDB to forward the results of execution units. It describes the committing mechanism to commit multiple instructions at the clock cycle. Finally, it describes about how the processor design that is build is verified in Bluespec

System Verilog and also describes about the test cases generated to verify the design.

Chapter 5 concludes with a short description about the future work.

CHAPTER 2

BACKGROUND

This chapter deals about the basics of the Superscalar Processor Architecture. It also deals about the key features of Bluespec System Verilog (BSV), why Bluespec is used, how to build a design in the BSV. It describes about the proposed microarchitecture of the superscalar processor based on the Tomasulo algorithm.

2.1 Basics of Superscalar Processor Architecture

In a classic Scalar Pipeline processor, an instruction has to be executed in order. Due to data dependencies and stalls, we cant achieve Instructions per cycle (IPC) equal to 1.

In a Superscalar Processor, multiple instructions are dispatched and issued in every clock cycle. They follow Instruction level Parallelism (ILP) which is overlap among the instructions. They can process multiple instructions in the same pipeline stage and also in different pipeline stages. Here, we can achieve IPC greater than 1.

The instruction sequence for both Scalar and Superscalar Processor is shown in the Figure: 2.1.

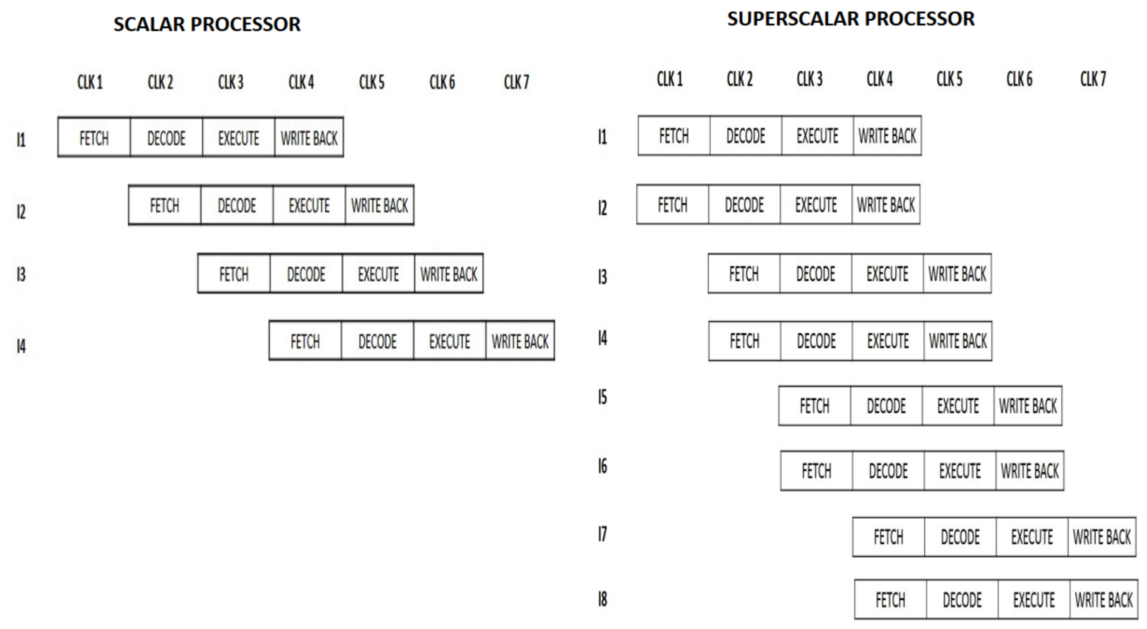


Figure 2.1: Instruction sequence for Scalar and Superscalar Processor

The above sequence of instructions can be executed in that manner if there is no dependencies among the instructions. If there are some dependencies among the instructions, there will be stalls in the pipeline and the instruction need to wait for the previous instruction to complete.

2.1.1 Scheduling techniques

To achieve more instruction level parallelism, two types of scheduling mechanisms are followed. They are

- Static Scheduling
- Dynamic Scheduling

In Static Scheduling, scheduling is done during compile time by the compiler. The compiler decides the order in which the instruction has to execute.

Example: Loop Unrolling

In Dynamic Scheduling, scheduling is done by the hardware. The hardware rearranges the instruction execution to reduce stalls while maintaining the data flow. The instructions are issued in order, execution is done out of order and written back into register in order. Out of order execution is the most commonly followed technique to implement dynamic scheduling. There is a lot of hardware complexity involved in dynamic scheduling.

Example: Hardware designed using Tomasulo Algorithm

2.1.2 Out of Order vs In-Order Execution

The order in which the program can be executed in the execution units can be briefly classified as follows

- Out of Order Execution
- In-Order Execution

In-Order Execution of Instructions: In In-order execution of instructions, the processing of instructions is normally done by following the steps given below.

1. The Instructions are fetched from the instruction cache.
2. If source operands are available in the register, then the instruction is sent to corresponding execution units. If it is not available, the processor is stalled until they are available.
3. After execution of instructions, the results are written in the destination register.

If there is a stall in any of the stage, all the instructions in the pipeline before that stage gets stalled and will continue to execute only after all the stalls are cleared. The instructions in in-order execution are statically scheduled.

Out of Order Execution of instructions: In out of order execution of instructions, the processing of instructions is normally done by following the steps given below.

1. The Instructions are fetched from the instruction cache.
2. The instructions are sent to reservation station which is a buffer that stores the operands.
3. When all the operands are available for a particular instruction, the instruction is sent to the execution unit even though the earlier instructions are still there in the reservation station waiting for the operands.
4. After execution of instruction, the results are stored in a buffer called reorder buffer which stores the results in program order.
5. Only after all the older instruction write their results into the register file, this instruction writes its results into the register file.

The instruction are dynamically scheduled in out of order execution of instructions. The main advantage is that whenever the operands are available the instructions are sent to the execution units and it does not wait for the older instructions to get operands. The problems in out of order execution is it introduces WAW and WAR hazards.

2.1.3 Out of Order Execution and Hazards

Out of order execution is the most common paradigm used in high performance Superscalar Processors. In Out of Order execution of instructions, the instructions are sent to execution units whenever all the operands are available.

The main problems in out of order execution are the hazards. There are three types of hazards namely

- Structural Hazard

  When two or more instructions conflicting for the limited resource, structural hazard occurs. Structural hazards can be avoided by stalling the processor pipeline.

- Data Hazard

  When an instruction depends on the result of the previous instruction, data hazard occurs. There are three types of data hazard namely

  - Read after Write (RAW) hazard (Data dependencies)
  - Write After Write (WAW) hazard (Output dependencies)
  - Write After Read (WAR) hazard (Anti dependencies)

  RAW hazards can be avoided by sending the instruction to the execution unit only when all the operands are available. WAW and WAR hazards can be avoided by implementing register renaming techniques using Reorder Buffers (ROB).

- Control Hazard

  Control hazard occurs due to the presence of branch instructions when the result of the branch instruction is not known. Control hazards can be avoided by using Branch prediction techniques.

2.1.4 Tomasulo algorithm and the principle of Register renaming

Tomasulo Algorithm is a method that uses the concept of register renaming to reduce WAW and WAR hazard. It uses the principle of dynamic scheduling for scheduling instructions. Almost all the modern processors use the derivative of the technique used in Tomasulo algorithm. The basic hardware implementation of Tomasulo algorithm is given below in Figure: 2.2.

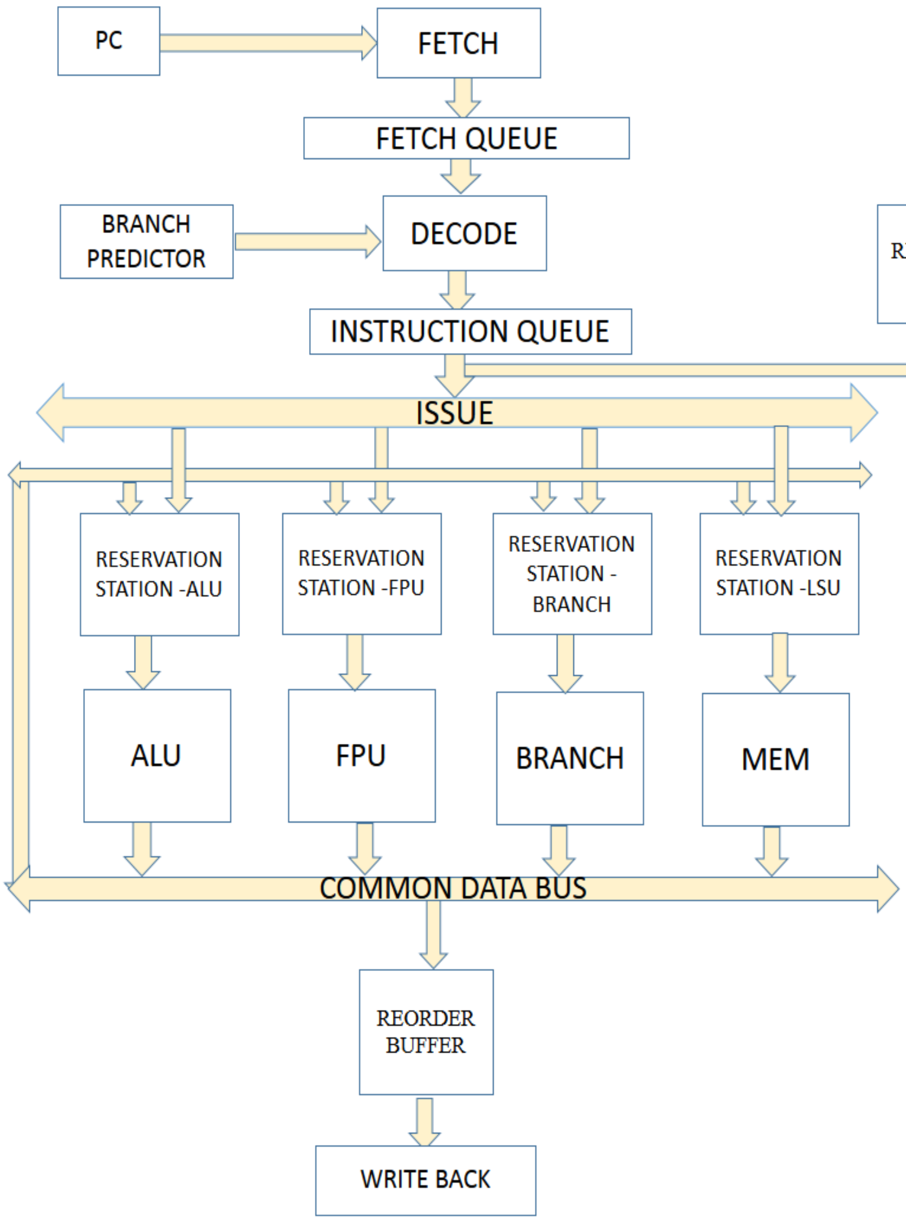


Figure 2.2: Hardware implementation of Tomasulo Algorithm

For Register renaming, a special type of buffer called Reorder buffer is used. Reorder buffer is a circular queue with both head and tail pointers. After fetch and decode, the entire Tomasulo algorithm can be divided into four stages. They are as follows

- Issue
- Dispatch
- Write Result
- Commit

2.1.4.1 Issue

The instructions after fetch and decode will be stored in the instruction queue. First, for an instruction to get issued there must be empty slot in both Reorder buffer and also in the Reservation station. If there is an empty slot in both ROB and Reservation station, an instruction is ready to be issued and if there is no empty slot either in ROB or reservation station, issue of the instructions is stalled.

The instruction is assigned an entry (entry pointed by the tail) in the reorder buffer and the same ROB number is updated in the destination register of the instruction and is tagged INVALID. The tail of the ROB is incremented by 1.

For Operand fetching, first the corresponding source registers are looked in to and if it matches tagged VALID, the operands are fetched from it and is sent to the reservation station. If it is tagged INVALID, then the value in the ROB number mentioned in the destination register is checked. If the value in the ROB number is also tagged VALID, then the result in ROB is sent to the reservation station. If the value in the ROB number is also tagged INVALID (i.e the results are not yet updated), then the ROB number in the source register is sent to the reservation station. The ROB number of the destination register assigned for the instruction is also sent to the reservation station which will be useful during result writing in the ROB. Since the instructions are issued in in-order, all the instructions in the reorder buffer will be in in-order starting from the head.

For Example, let us take the sequence of instructions below and the ROB tail currently as ROB8.

ADD R0, R1, R2

SUB R4, R0, R2

After issue, the instructions in the reservation station will be stored as

ADD ROB8, R1, R2

SUB ROB9, ROB8, R2

This process of renaming the destination register with the ROB number is called Register Renaming.

2.1.4.2 Dispatch

Dispatch is the second step in Tomasulo algorithm where the instructions are sent to the corresponding execution units when they are ready. As we see in the Figure: 2.2, each functional unit has a set of buffers associated with it called Reservation station. The main purpose of Reservation station is to store the operands or the reorder buffer number (assigned during register renaming). Only when all the operands are available, the instructions will be sent to the corresponding execution unit. The instructions in the reservation station will keep on monitoring the CDB for the results.

2.1.4.3 Write Result

Once the execution unit finishes and produces its results, the result along with its reorder buffer number are broadcasted in the Common Data Bus (CDB). In the reservation station, the reorder buffer number is matched with the operands and whichever operands ROB number matches, the operands are updated with the data in the Common data bus. By forwarding of operands, RAW hazard is avoided. The results are also written into the corresponding entry in the Reorder buffer which is assigned during issue.

2.1.4.4 Commit

Commit stage is the final stage of the Tomasulo algorithm. An instruction in the ROB is allowed to commit only when that instruction reaches the head of the ROB and its results are available to be written in to destination register. The valid bit in the destination register is updated as VALID only if the ROB number stored in the destination matches the head of the ROB. Otherwise, only the result is written into the destination register and tag field is updated as INVALID.

If the instruction is a branch instruction, the instruction is committed only if there is no misprediction. If there is misprediction, the entire pipeline is flushed and the program counter of the instruction successor to the branch is loaded. It the instruction is a STORE instruction, the result is written into the memory unit. If an instruction commits, the instruction is removed from the ROB and head pointer is incremented by 1.

2.1.5 Extensions in Tomasulo Algorithm

2.1.5.1 Branch Prediction techniques

Branch instructions change the flow of program control. Usually, Branch follows through two paths namely branch taken or branch not taken. Branch instructions causes control hazards and reduces the processor performance. Predicting the outcome of the branch instructions is important to increase the processor performance. There are two types of branch prediction. They are

- Static Branch Prediction
- Dynamic Branch Prediction

Static Branch Prediction

This is the simplest branch prediction technique. In static branch prediction, the outcome of the branch is predicted during the compile time and the prediction solely depends upon the branch instruction itself. The branch will be predicted as always taken or always not taken.

Dynamic Branch Prediction

Dynamic branch prediction uses a branch history table to store the history of the branch instructions. The outcome of the branch is predicted by recording the information about the past branch history in branch history table during a programs execution and is therefore done at run-time.

2.1.5.2 Multiple Instruction Issue

In order to improve the performance of the processor, we need to increase the IPC. IPC cannot be increase if we issue only one instruction at a time. So, we need to increase the issue width of the processor. In multiple issue, more than one instruction is issued every clock cycle to keep the functional units busy. There are two approaches in implementing multiple issue. One is issuing 1st instruction in the first half cycle and the 2nd instruction in the second half cycle. The second approach is to issue both the instructions simultaneously taking care of the dependencies between them. Even in multiple issue, we need to issue the instructions only in in-order. Another important thing in multiple issue is it needs to support any combination of instructions.

Multiple instruction issue will result in greater hardware complexity and longer wire length. The main problem in multiple issue is as the width of the issue is increased, dependency check between the instructions increases exponentially. If a multiple issue is implemented, the reservation station should be ready to dispatch multiple instructions at the same time. The common data bus must be modified such that it is wide enough to forward multiple results to the reservation station at the same clock cycle. The Reorder buffer must commit multiple instructions at the same clock cycle.

2.2 Bluespec System Verilog

BSV is a language used in design of electronic systems (ASICs, FPGAs and systems). It is a very high level hardware description language and the code written in Bluespec is completely synthesizable to hardware. Because of its high level and completely synthesizable features, it has made many activities that are done in software simulation move to FPGA based simulation.

2.2.1 Key Features of BSV

- High level atomic rules in place of Verilog method of always block.
- High level Interfaces instead of Verilog method of port list.
- Nested, parameterizable interfaces, allowing easy construction of complex interfaces.
- Automatic synthesis of the control logic to manage complex concurrency which is the most error prone part of RTL design.
- High level constructs for types, with very flexible type parameterization and strong static type-checking.

2.2.2 Bluespec System Verilog Constructs

2.2.2.1 Rules

BSV expresses synthesizable behavior with rules instead of synchronous always blocks. Rules are powerful concepts for achieving correct concurrency and eliminating race conditions. A primary feature of rules in BSV is that they are atomic; each enabled rule can be considered individually to understand how it maintains or transforms state. Atomicity allows the functional correctness of a design to be determined by looking at each of the rules in isolation, without considering the actions of other rules. This one-rule-at-a-time semantics greatly simplifies the process of determining the functional correctness of a design.

In the hardware implementation compiled by the BSV compiler, multiple rules will execute concurrently. The compiler ensures the actual behavior is consistent with the logical behavior, thus preserving functional correctness while achieving performance goals.

Components of Rules

Rules are made up of two components:

Rule Condition: A boolean expression which determines if the rule body is allowed to execute or not.

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

Properties of Rules:

Execution of a rule w.r.t to all other rules is instantaneous, complete and ordered.

Instantaneous: All the actions in the rule body occur at a single, common instant and there is no sequencing of actions within a rule.

Complete: When the rule fires, the entire body of the rule 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.

2.2.2.2 Modules

A module consists of three things: state, rules that operate on that state, and an interface to the outside world (surrounding hierarchy). A module definition specifies a scheme that can be instantiated multiple times.

2.2.2.3 Methods

Signals and buses are driven in and out of modules using methods. These methods are grouped together into interfaces. There are three kinds of methods:

Value Methods: Take 0 or more arguments and return a value.

Action Methods: Take 0 or more arguments and perform an action (side-effect) inside the module.

ActionValue Methods: Take 0 or more arguments, perform an action, and return a result.

2.2.2.4 Interfaces

Interfaces provide a means to group wires into bundles with specified uses, described by methods. An interface is a reminiscent of a struct, where each member is a method. Interfaces can also contain other interfaces.

2.2.3 Building a design in Bluespec System Verilog

The various steps involved in building a design in BSV is shown in Figure: 2.3.

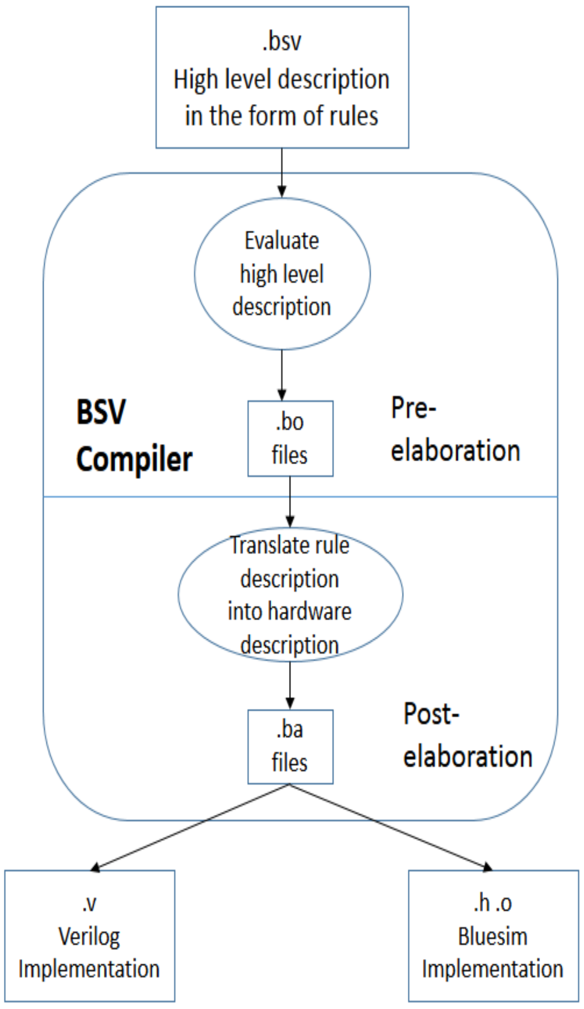


Figure 2.3: Building a design in BSV

1. The designer writes the BSV code and it may contain Verilog, VHDL and C components.
2. The BSV code is compiled into a Verilog or a Bluesim specification. This step has 2 stages:
   - Pre-elaboration parsing and type checking.
   - Post-elaboration code generation.
3. The compiled output is either linked to a simulation environment or processed by synthesis tool.

2.3 Microarchitectural description of the Superscalar Processor

The Microarchitectural description of the proposed Superscalar Processor based on the Tomasulo Algorithm is as shown in Figure: 2.4.

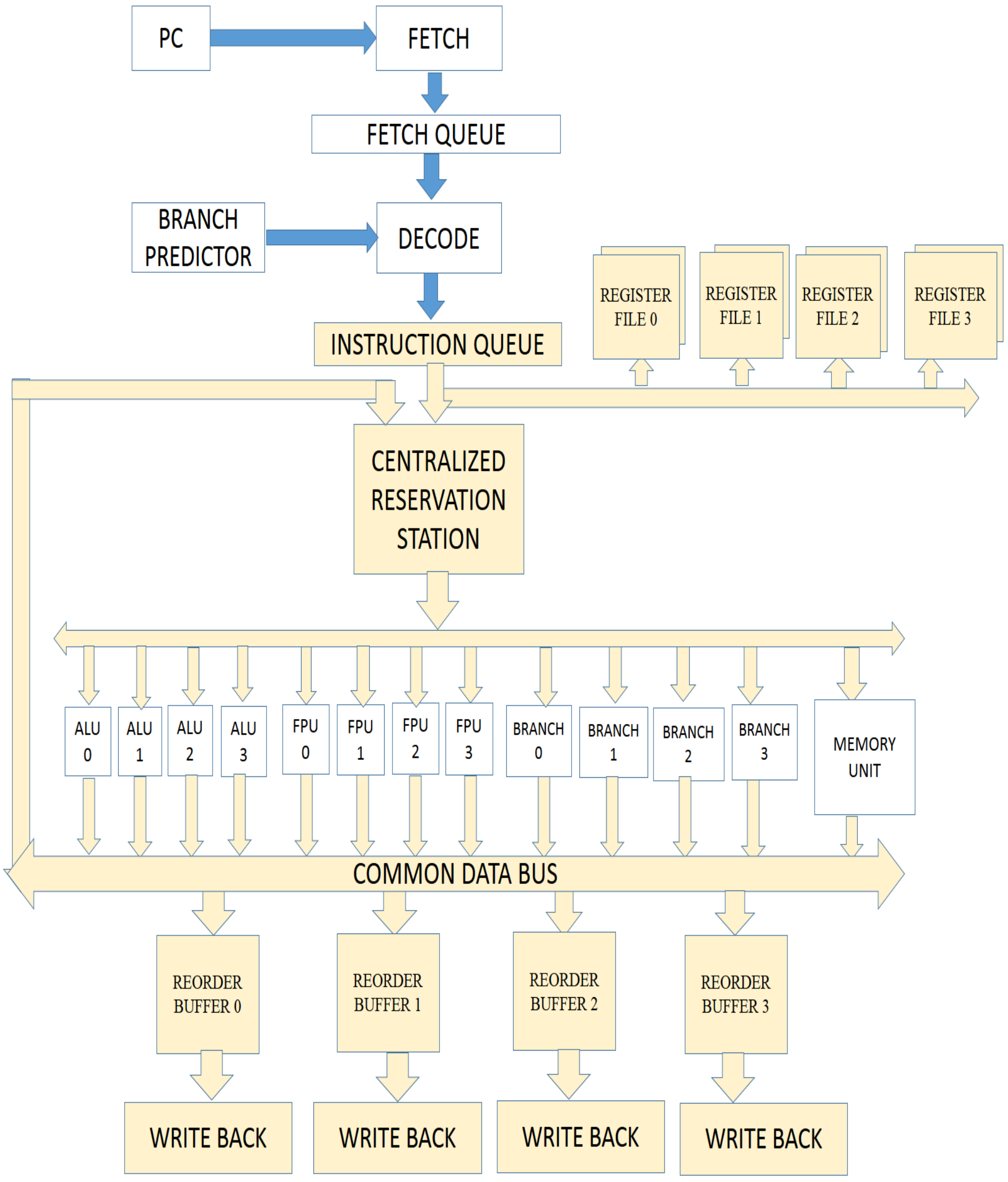


Figure 2.4: Microarchitectural description of the Superscalar Processor

The Microarchitecture is a 64bit, Quad threaded, Out of order microprocessor that is capable of fetching, decoding and issuing 4 instructions each clock cycle. The operands are fetched during issue of the instructions. The instructions after getting issued are stored in the centralized reservation station. The Reservation station can dispatch a maximum of 12 instructions: 4 of ALU, 4 of FPU, 4 of BRANCH in the same clock cycle.

There are 8 Register files: 2 Register files Integer register file, Floating register file for each thread. There are 4 Reorder Buffers (ROB): One Reorder Buffer for each thread. The Execution units are of ALU, FPU, BRANCH, MEMORY type. The Execution units are duplicated 4 times to avoid waiting of the ready instructions in Reservation station. The results from the execution units are forwarded to the reservation station to update the operands with the data. The Common Data Bus (CDB) used for Operand forwarding can forward the results of at most 4 execution units. Selecting the execution units which need to drive the bus is done by prioritization schemes. The Reorder Buffer can commit 4 instructions in a clock cycle. Committing the instruction can be to either register file or memory unit.

My project work involves the implementation of stages highlighted in the Figure: 2.4.

CHAPTER 3

QUAD ISSUE

This chapter starts with how the instruction queue is designed to support quad issue. It also compares the centralized with distributed reservation station and justifies why centralized reservation station is used in the designed superscalar processor. It then talks about the design of the centralized reservation station. Later, it describes about the Operand fetch mechanism during issue of an instruction for both single issue and quad issue of instruction. Then, it briefly describes about the design of Quad Issue for the proposed microarchitecture of the superscalar processor.

3.1 Design of Instruction Queue Buffer

Instruction queue is a buffer which holds the instructions after decoding of the instructions. The instruction queue has to be modified to support instructions of multiple threads. Figure: 3.1 shows the design of the instruction queue.

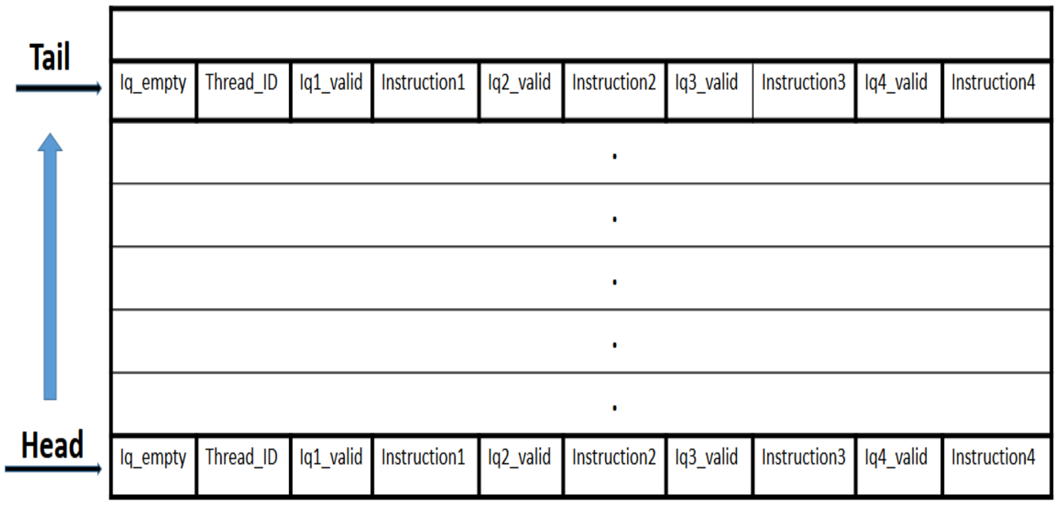


Figure 3.1: Design of Instruction Queue buffer

Instruction queue buffer is a circular buffer with both head and tail pointers. The instructions after decoding will be always added to the slot pointed by the tail. All the instructions starting from Instruction1 to Instruction4 will always be of the same thread for a particular slot. The Head pointer indicates the slot from which the instructions has to be issued to the reservation station. The instructions will be issued only in order starting from the head.

If an instruction gets issued, then the corresponding validity field is made INVALID. For example: If Instruction1 is issued, then Iq1 valid is made INVALID. When all the four instructions of a slot gets issued and all the four validity field has become INVALID, then the corresponding slot is made empty. Iq empty field in the corresponding slot is made TRUE. Then, the head pointer is incremented by one.

3.2 Comparison between Centralized and Distributed Reservation Station

Reservation Station is a buffer that holds operands for those instructions that are issued. It updates the operands with results from the CDB for those instructions that are waiting for the data every clock cycle. Whenever all the operands are available, then the instruction will be sent to execution unit based on the availability of the execution units and the corresponding slot in the reservation station will be made empty. There are two ways of implementing reservation station. They are

- Distributed Reservation station
- Centralized Reservation station

3.2.1 Distributed Reservation station

In Distributed Reservation station, there will be separate buffers for buffering operands for each execution unit like RS for ALU, RS for FPU, RS for BRANCH. During issue, the instructions in the instructions queue buffer will be issued to corresponding reservation stations based on the type of the execution unit. The instructions from the reservation station will be sent to corresponding execution unit based on the availability of the execution unit. The advantages and disadvantages of using distributed reservation station is described below. Figure: 3.2 shows the distributed reservation station connected with instruction queue buffer and with execution units.

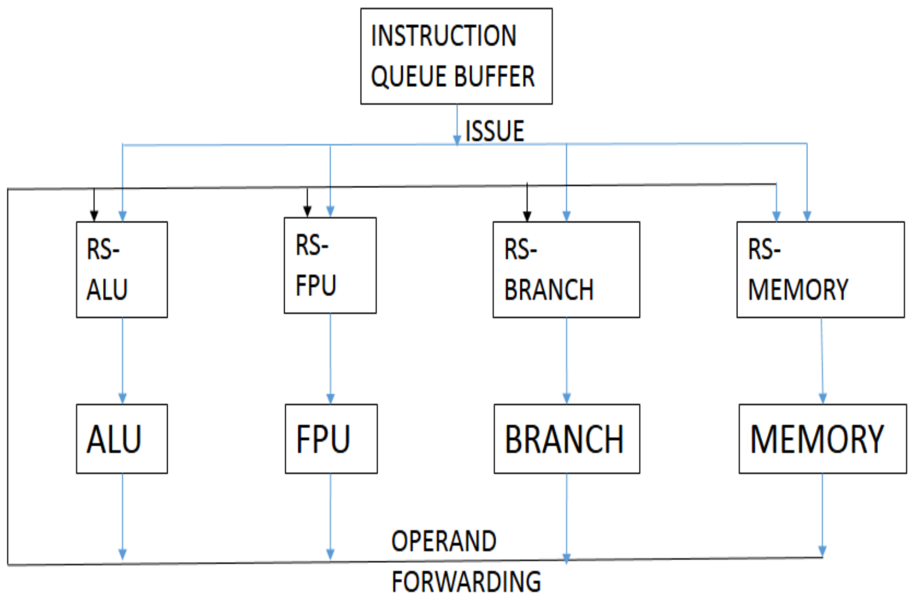


Figure 3.2: Distributed Reservation station

Advantages of Distributed Reservation Station

- Complexity of the issue logic is less as compared to centralized reservation station. This is because in the distributed reservation station, control is distributed among the reservation station and each RS only needs to examine a subset of all instructions in the corresponding buffer.
- Hardware for interconnects between RS and functional units are lower than centralized reservation station. This is because only the RS slots associated with the particular functional unit need to be connected as they are the only slots that can issue any instruction to that functional unit.
- Due to the reduction in complexity, clock cycle time is usually shorter during issue in Distributed reservation station and thus higher clock speed is possible.

Disadvantages of Distributed Reservation Station

- Inefficient use of available slots in the reservation station. For example, if a program contains purely integer instructions and no floating point instructions and if all the available integer reservation station are used up, the processor will have to stall as it cannot use the floating point reservation station for integer instructions.
- Due to inefficient use of the reservation station slots, a larger number of reservation station slots are required to match Centralized RS design's efficiency and thus more hardware may be required.

3.2.2 Centralized Reservation station

In Centralized Reservation station, the instructions from instruction queue buffer will be issued to a buffer which is common for all the execution units. The instructions in the reservation station will be sent to corresponding execution units depending upon the instruction type and the availability of the execution units when all the operands are available. The advantages and disadvantages of using Centralized reservation station is described below. Figure: 3.3 shows the distributed reservation station connected with instruction queue buffer and with execution units.

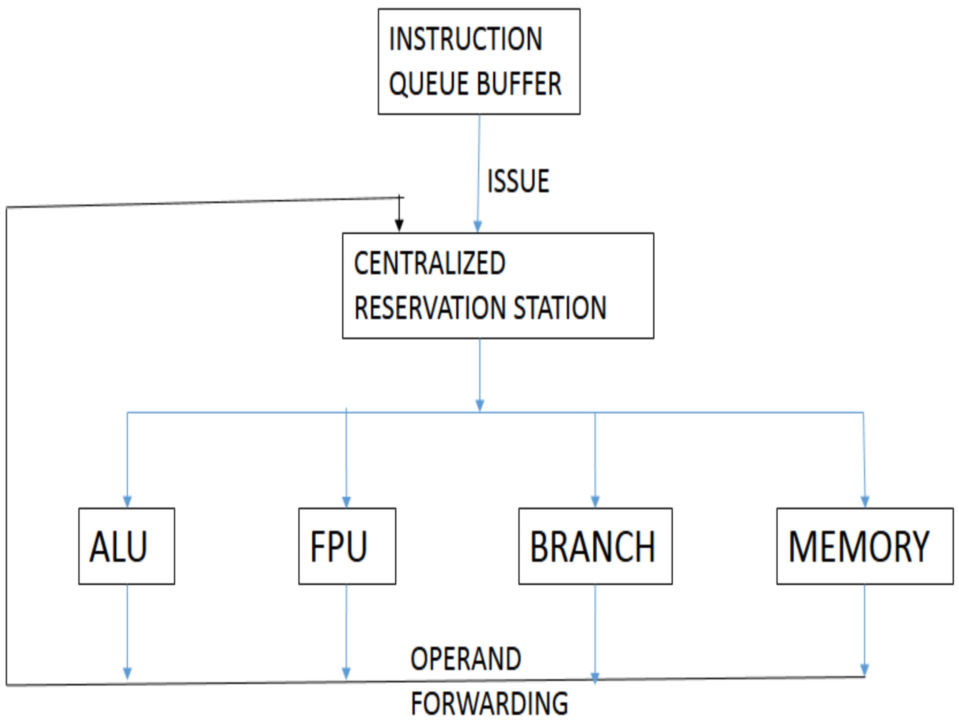


Figure 3.3: Centralized Reservation station

Advantages of Centralized Reservation Station

- There is no need to implement a separate scheduling logic at each reservation station which saves a lot of area and hence hardware.
- Efficient utilization of entries of reservation station can be made compared to distributed reservation station. For example, if there are 4 ALU instruction to issue to

reservation station of type ALU and the reservation station has only 2 slots, we can issue only two instructions even though reservation station of other types are free. But if we use centralized reservation station, we will be able to issue all the four instructions.

- Number of slots in the reservation station can be made less compared to the distributed reservation station since all the slots are used efficiently.

Disadvantages of Centralized Reservation Station

- Scheduling logic becomes complex. We need to search all entries in the reservation station to issue instructions to the execution units. The complexity increases more as the issue width is increased.
- Due to the increased complexity, clock cycle time during issue is usually more in centralized reservation station and hence clock speed is less.

3.2.3 Design Decision: Choosing the type of Reservation station

One of the important factors to consider during the design of the processor is choosing among the two reservation station. As we discussed, both centralized as well as distributed reservation station has their own merits and demerits. When looking at the type of reservation station used in the CPU core of latest processors, some companies prefer one type and some prefer another type. Choosing a particular reservation station depends on which factor we needs to give more importance. As the processor design team wants to give more preference to the efficient use of the reservation slots and were ready to compromise on hardware complexity, centralized reservation station was chosen and was successfully implemented.

3.3 Methods in filling the slots of Reservation Station during Issue

There are many ways in which the reservation station can be filled by instruction queue buffer and dispatched to execution units. Two ways had been suggested to the processor design team in which the instructions from the instruction queue buffer can be filled into reservation station. Finally, one method was chosen based on the hardware complexity and implementation issues. The two ways are

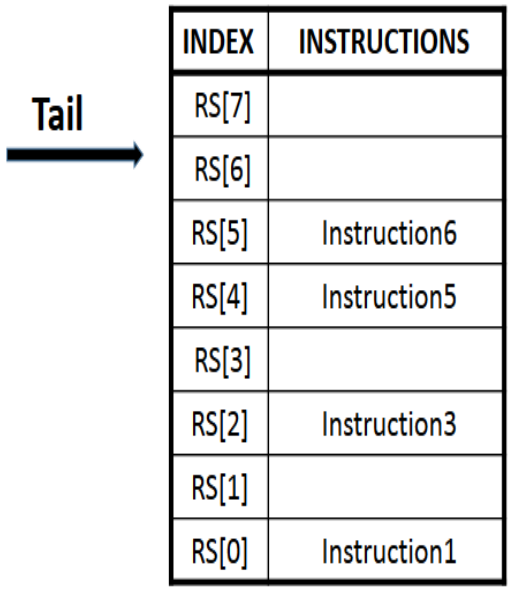
- Filling at the tail pointer of the reservation station.
- Filling anywhere in an empty entry in the reservation station.

3.3.1 Method1: Filling at the tail pointer of the reservation station

In this method, the instructions that are issued are always filled at the tail pointer of the reservation station. After an instructions gets filled in the slot, the tail pointer of the reservation station will be increased by one. If an instruction is sent to the execution unit from any of the slots in the reservation station, the corresponding entry is made empty.

The main advantage of using this method is that the instructions in the reservation station are in order. So, it will be helpful during the dispatch of the instructions to the execution units.

The main problem in this method is that the tail pointer will quickly reach the last entry of the reservation station while filling the reservation station since we are always filling the reservation station at the tail and removing the instruction at any slot. We need to have a reservation station with a large number of slots. Empty reservation station slots below the tail pointer will not be useful to fill the issued instructions unless the tail pointer is decreased by dispatching instructions pointed by the tail pointer to the execution units. There will be stalls in the reservation station if the tail pointer reaches the last entry. The methods to solve this problem will be discussed briefly in Chapter : Out of Order Execution in dispatch section. Figure: 3.4 shows Method1 implementation of reservation station.



INDEX	INSTRUCTIONS
RS[7]	
RS[6]	
RS[5]	Instruction6
RS[4]	Instruction5
RS[3]	
RS[2]	Instruction3
RS[1]	
RS[0]	Instruction1

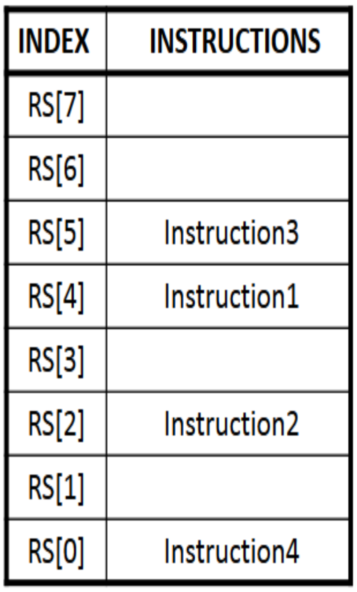
Figure 3.4: Method1 implementation of Reservation station

3.3.2 Method2: Filling anywhere in an empty entry in the reservation station

In this method, the instructions that are issued can be filled in any empty slots of the reservation station. There is no head and tail pointers in this method as we are filling anywhere in the reservation station. If an instruction is sent to the execution unit from any of the slots in the reservation station, the corresponding entry is made empty.

The main advantage of using this method is we can use the reservation station entries in a more efficient manner compared to the first method. Every slot in the reservation station can be used efficiently since we can fill anywhere in reservation station and the reservation station is not filled quickly by the issue unit. Stalls are less compared to the first method.

The main problem for this method comes during dispatch of instructions into the execution units when the instructions are ready in the reservation station. Since all the instructions in the reservation station are not in order they are issued, we need to find an efficient way of choosing the instructions among the ready instructions. We need to choose the oldest instructions among the ready instructions and send it to execution units based on the availability of the functional units. The methods to solve this problem will be discussed briefly in Chapter : Out of Order Execution in dispatch section. Figure: 3.5 shows the Method2 implementation of reservation station.



INDEX	INSTRUCTIONS
RS[7]	
RS[6]	
RS[5]	Instruction3
RS[4]	Instruction1
RS[3]	
RS[2]	Instruction2
RS[1]	
RS[0]	Instruction4

Figure 3.5: Method2 implementation of Reservation Station

3.3.3 Design Decision: Choosing the method for filling the reservation station

While choosing between the methods, one needs to consider the area occupied, efficiency in using a method. The two methods discussed has its own advantages and disadvantages. If we look into method1, the reservation station slots are not used efficiently and there seems to be a lot of empty slots that remain idle below the tail pointer of the reservation station. Also, the area occupied will be more as more slots are needed for method1. In method2, the reservation slots are used very efficiently and there are no empty slots that remain idle.

Since method2 is more efficient and occupies less area compared to method1, method2 was chosen and was implemented successfully in Bluespec.

3.4 Design of Centralized Reservation Station

The Figure: 3.6 shows the Centralized Reservation station that is implemented in the superscalar processor.

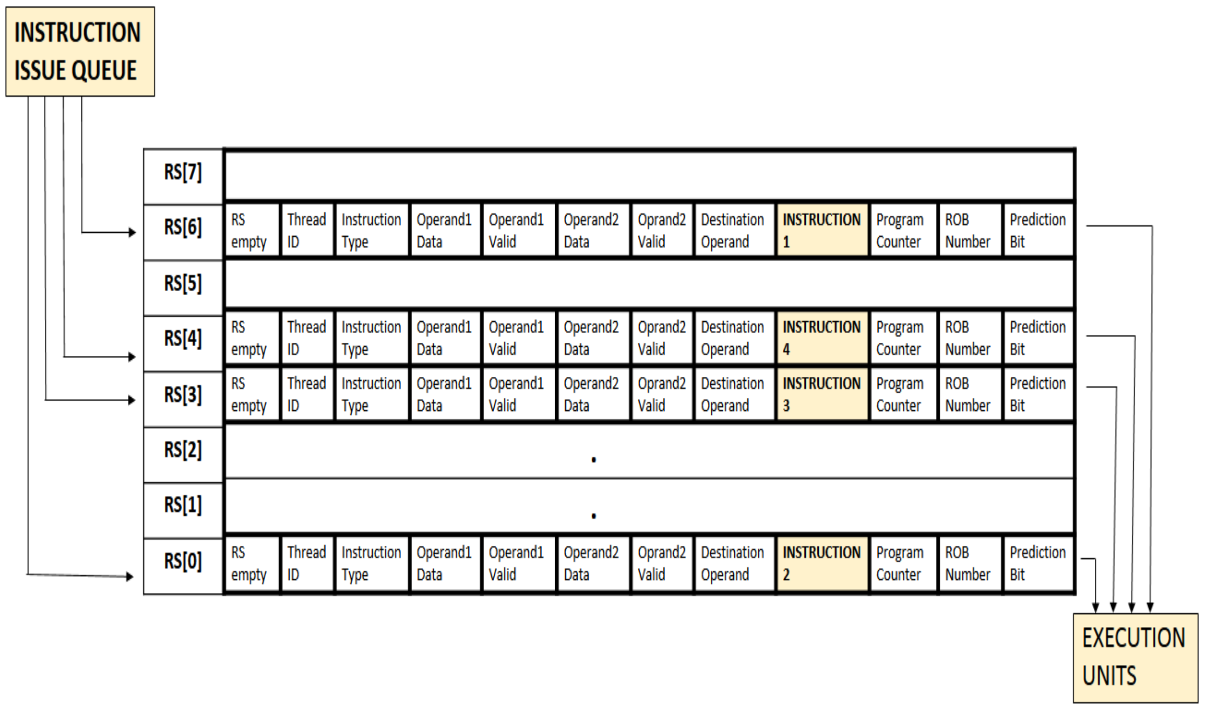


Figure 3.6: Design of Centralized Reservation Station

The centralized reservation station has been implemented based on method2. As we see in the Figure: 3.6, the instructions from the instruction queue buffer can be issued to any slot in the reservation station. Also, the instructions in the reservation station which are ready to execute can be sent to execution unit from any slot. The fields in the reservation station are described below.

Rs empty To check whether the reservation station is empty or not.

Thread ID Stores the thread ID for each instruction which is needed during writing the result into the corresponding reorder buffer in commit stage.

Instruction type Stores the execution type namely ALU, FPU, BRANCH, MEMORY which is needed during dispatch of instruction into the corresponding execution unit.

Operand1 data, Operand2 data Stores the data or ROB number of the corresponding thread which is to be sent to execution unit.

Operand1 valid, Operand2 valid Set to 1 if data is present in Operand data, set to 0 if ROB number is present in Operand data.

Destination operand Stores the destination address of the register where the result has to be written during commit stage.

Instruction Stores the instruction bits which is needed during execution of some of the instructions.

Program counter Stores the program counter number of the instruction which is needed when there is misprediction of BRANCH instructions in commit stage.

ROB number Stores the allocated ROB number of the instruction which is needed when writing the result from the execution unit to allocated slot in the reorder buffer.

Prediction Bits Stores the predicted bit in case of BRANCH instruction and the bits will indicate whether the BRANCH instruction should TAKE THE BRANCH or NOT.

3.5 Operand Fetch Mechanism

The issue stage in the superscalar processor mainly involves the operand fetch for the source operands. The operands can be fetched either from register file or Reorder buffer. For floating point instructions, the operands are fetched from floating point register file. A simple operand fetch mechanism is described below.

3.5.1 Basic Operand Fetch Mechanism for a single issue processor

The operands are fetched either from Register file or ROB of the corresponding thread. First, the corresponding source registers are looked in to and if it matches tagged VALID, the data is taken as operands and is sent to the reservation station. If it is tagged INVALID, then the value in the ROB number mentioned in the destination register is checked. If the value in the ROB number is tagged VALID, then the data in ROB is sent to the reservation station. If the value in the ROB number is also tagged INVALID, then the results are not yet updated and the ROB number in the source register is sent to the reservation station.

The instruction is assigned an entry (entry pointed by the tail) in the reorder buffer and the same ROB number is updated in the destination register of the instruction and is tagged INVALID.

Figure: 3.7 shows the simple operand fetch mechanism for a single issue processor.

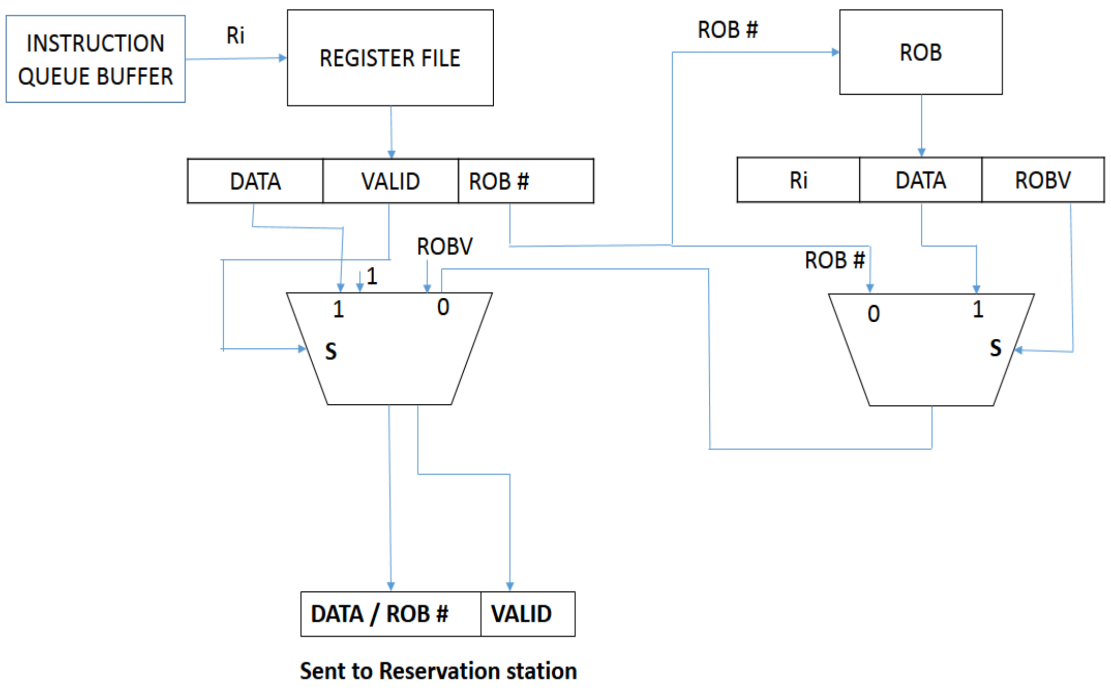


Figure 3.7: Simple operand fetch mechanism for a single issue processor

3.5.2 Operand Fetch Mechanism for a Quad issue processor

In a quad issue processor, maximum of four instructions are issued every clock cycle to the reservation station. During issue of four instructions, one needs to take care about the dependencies between the instructions. In a modern superscalar processor, every possible combination of dependent instructions that is allowed to issue in the same clock cycle must be considered. Since the number of possibilities climbs as the square of the number of instructions that can be issued in a clock cycle, the issue step is a likely bottleneck for attempts to go beyond four instructions per clock cycle. Also, the four instructions can be of any type and processor should be ready to issue four instructions of different type. Let us now look into how the operand fetch for a quad issue is implemented.

3.5.2.1 Operand fetch for 1st instruction

The operand fetch for the 1st instruction is same as that in single issue because there is no need to check the dependencies among the instructions since this is the first instruction to be issued.

But, for updating the ROB field in the destination register with the ROB slot number allotted to the instruction, the destination register is compared with the destination register of the next three instructions that are yet to be issued and if any one of the destination register matches, then the ROB field in destination register is not updated with the allocated ROB slot number. If it does not match with any of the instructions, then the ROB filed in destination register is updated with the allocated ROB slot number. Figure: 3.8 show how the destination operand is updated for the 1st instruction.

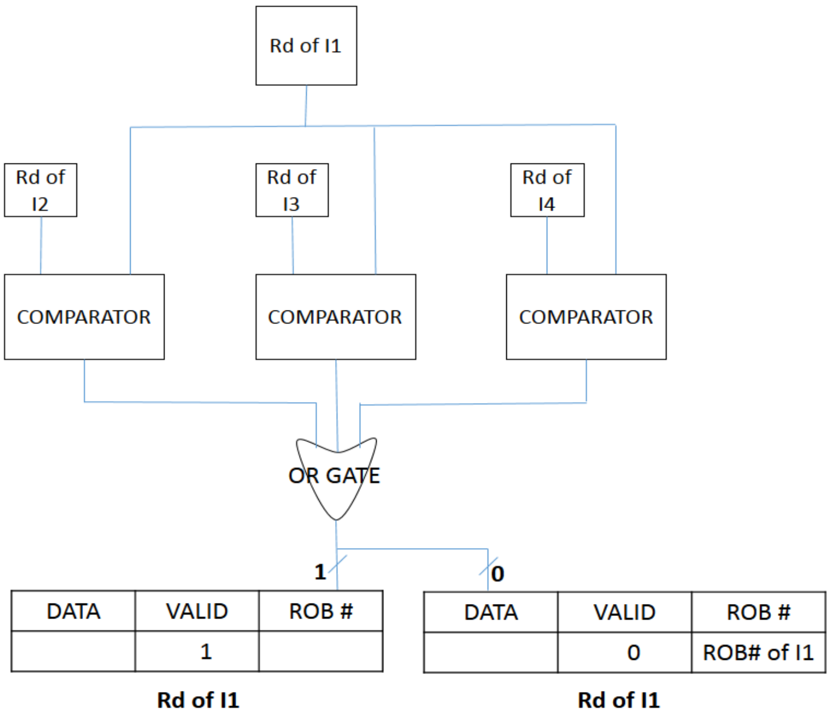


Figure 3.8: Updating destination operand for Instruction1

3.5.2.2 Operand fetch for 2nd instruction

During the operand fetch of the 2nd instruction, dependency is checked with the instruction 1. The source operands of 2nd instruction are compared with the destination operand of the 1st instruction. For those source operands that matches, the allocated ROB slot of the 1st instruction is used as operand and for those operands that does not match, the operands are fetched as in single issue. All those dependent source operands will update the operands with data during forwarding of data in Common data bus. The 2nd instruction is allocated a slot in ROB which is tail + 1.

While updating the ROB field in the destination register with the ROB slot number allotted to the 2nd instruction, the destination register is compared with the destination register of the next two instructions that are yet to be issued and if any one of the destination register matches, then the ROB field in destination register is not updated with the allocated ROB slot number. If it does not match with any of the instructions, then the ROB filed in destination register is updated with the allocated ROB slot number of the 2nd instruction. Figure: 3.9 show the operand fetch mechanism for Instruction 2 and how the destination operand is updated for the 2nd instruction.

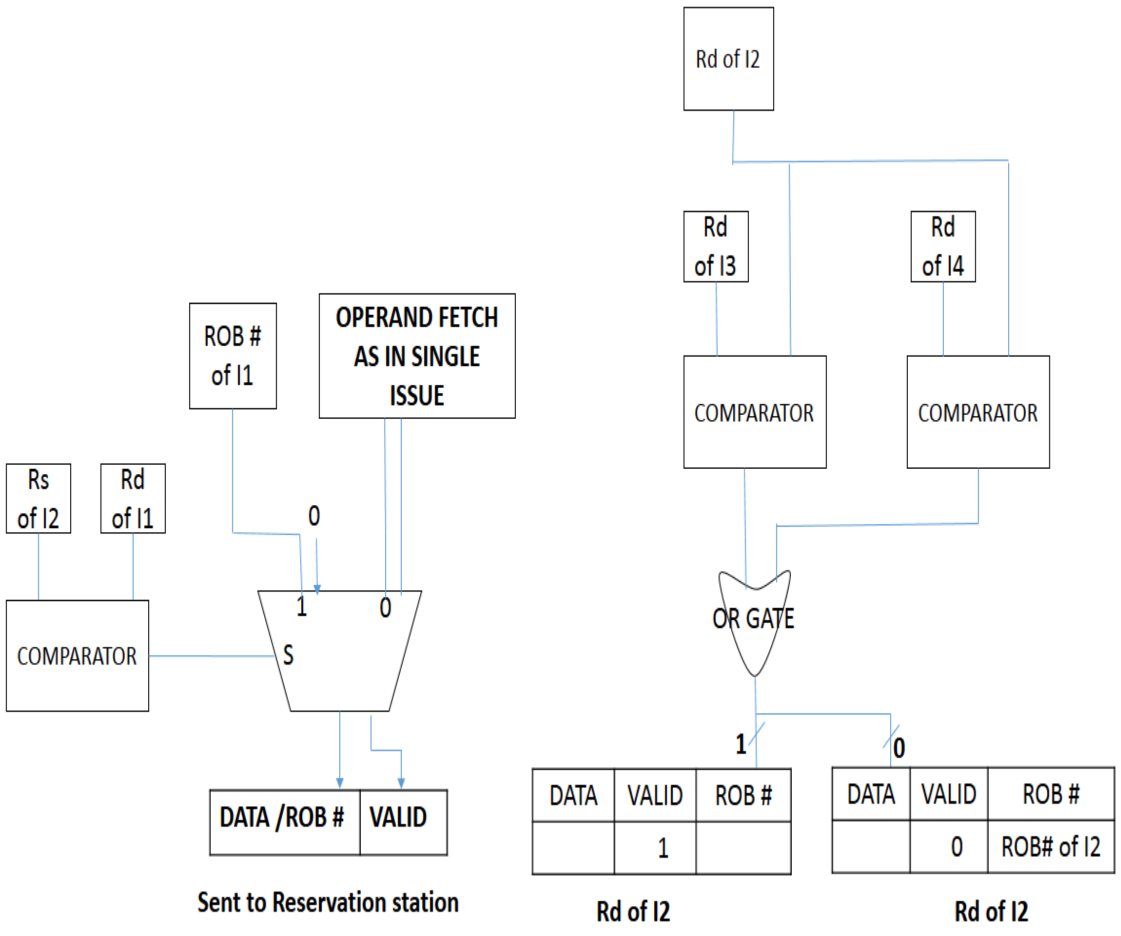


Figure 3.9: Operand fetch mechanism for Instruction2

3.5.2.3 Operand fetch for 3rd instruction

During the operand fetch of the 3rd instruction, dependency is first checked with the instruction 2. The source operands of 3rd instruction is compared with the destination operand of the 2nd instruction. For the source operands that matches, the allocated ROB slot of the 2nd instruction is used as operand and for those operands that does not match, the source operands are again compared with the destination operand of the 1st instruction. If it matches the allocated ROB slot of the 1st instruction is used as operand and for those operands that does not match, the operands are fetched as in single issue. All those dependent source operands will update the operands with data during forwarding of data in Common data bus. The 3rd instruction is allocated a slot in ROB which is tail + 2.

While updating the ROB field in the destination register with the ROB slot number allotted to the 3rd instruction, the destination register is compared with the destination register of the fourth instructions that is yet to be issued and if the destination operand matches, then the ROB field in destination register is not updated with the allocated ROB slot number. If it does not match, then the ROB filed in destination register is updated with the allocated ROB slot number of the 3rd instruction. Figure: 3.10 show the operand fetch mechanism for Instruction 3 and how the destination operand is updated for the 3rd instruction.

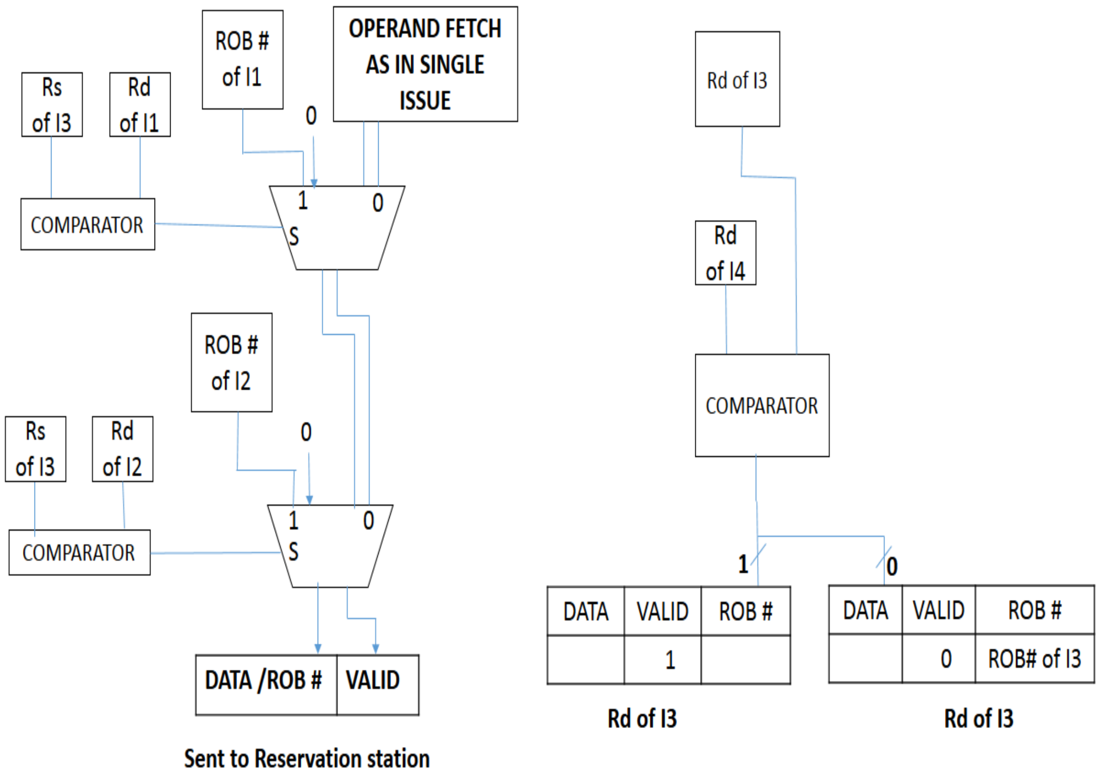


Figure 3.10: Operand fetch mechanism for Instruction3

3.5.2.4 Operand fetch for 4th instruction

During the operand fetch of the 4th instruction, dependency is first checked with the instruction 3. The source operands of 4th instruction is compared with the destination operand of the 3rd instruction. For the source operands that matches, the allocated ROB slot of the 3rd instruction is used as operand and for those operands that does not match, the source operands are compared with the destination operand of the 2nd instruction. If it matches the allocated ROB slot of the 2nd instruction is used as operand and for those operands that does not match, the source operands are again compared with the destination operand of the 1st instruction. If it matches the allocated ROB slot of the 1st instruction is used as operand and for those operands that does not match, the operands are fetched as in single issue. All those dependent source operands will update the operands with data during forwarding of data in Common data bus. The 4th instruction is allocated a slot in ROB which is tail + 3.

The ROB field in the destination register is updated with the ROB slot number allotted to the 4th instruction and there is no comparison needed as this is the last instruction going to be issued. Figure: 3.11 show the operand fetch mechanism for Instruction 4.

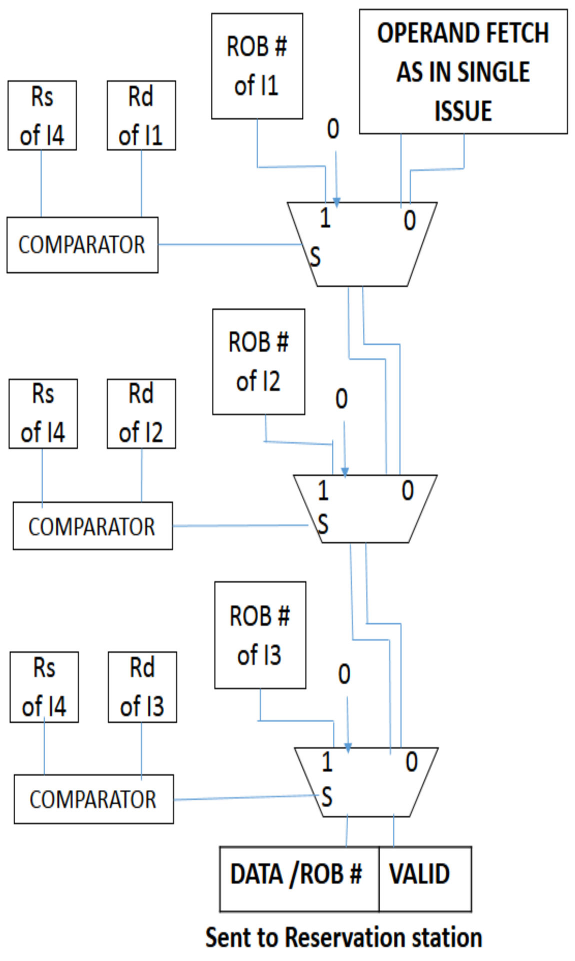


Figure 3.11: Operand fetch mechanism for Instruction4

3.5.2.5 Important points to consider

- For floating point instructions like single precision and double precision floating point instructions, we need to fetch operands from Floating point register file and for integer type instructions like ALU, BRANCH, MEMORY type instructions, we need to fetch operands form Integer register file.
- While comparing the destination operand for updating the ROB field among the instructions, we need to compare only if the type of instructions: Integer and floating point type instructions matches.
- Depending upon the thread ID, the operand fetch has to be made from the corresponding ROB since each thread has a separate ROB.
- During allocating the slot for the instructions, once we reach the last entry (ROB size - 1) we need to allot the slots from the slot 0 of the ROB since ROB is a circular buffer.

- While comparing the source operands for dependency among the instructions, we need to compare only if the type of instructions namely Integer and floating point type instructions matches.

3.6 Steps involved in implementing a Quad issue for a Multithreaded Processor

Till now we have discussed about the design of instruction queue buffer, design of centralized reservation station, the operand fetch mechanism for a quad issue processor. This section deals about how all those that were discussed are put together and how the quad issue is implemented by using those. The Figure: 3.12 shows the steps involved in implementing a quad issue in a superscalar processor.

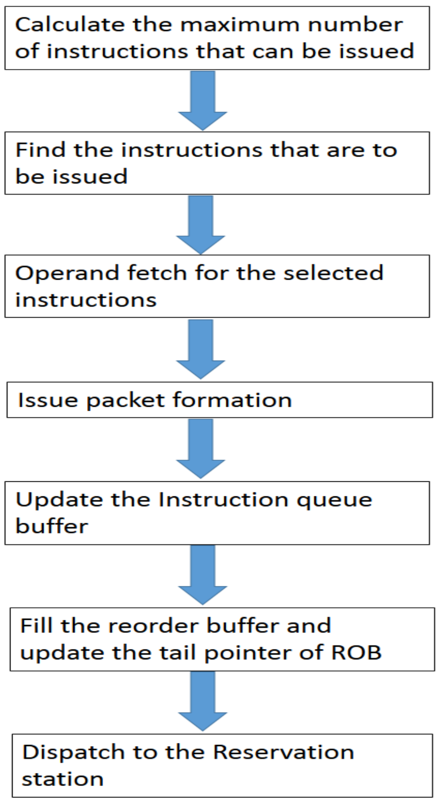

Calculate the maximum number of instructions that can be issued
Find the instructions that are to be issued
Operand fetch for the selected instructions
Issue packet formation
Update the Instruction queue buffer
Fill the reorder buffer and update the tail pointer of ROB
Dispatch to the Reservation station

Figure 3.12: The steps involved in implementing a quad issue for multithreaded processor

The following are the steps involved in implementing quad issue in a superscalar processor.

1. As the first step, the maximum number of instructions that can be issued is calculated based on the available number of empty slots in reservation station and reorder buffer. The number of empty slots in reservation station is calculated. Based on the thread ID, the number of empty slots in corresponding reorder buffer is calculated. The minimum of these two numbers is taken as the maximum number of instructions that can be issued.
2. The above step gives only the maximum number of instructions that can be issued. But actually the number of instructions to issue can be either equal to or lesser than the calculated number. There are 4 instructions of same thread for a particular slot in instruction queue buffer and only when all the instructions of one particular slot are issued, then only the head pointer will be moved to next slot. At a particular clock cycle, not all 4 instructions will be issued.

   For example, lets say the 1st instruction is issued in the previous clock cycle. Now during calculation of maximum number of instructions to issue, if the number is

   0: no instruction is issued.

   1: only 2nd instruction is issued.

   2: 2nd and 3rd instructions are issued.

   3: 4 and above: 2nd, 3rd, 4th instructions are issued.

   Only after the 4th instruction is issued, Iq empty field in the instruction queue buffer of that particular slot is made TRUE and the head pointer is incremented by one.
3. After selecting the instructions that are to be issued from the previous step, the operands are fetched by following the same procedure as discussed before in operand fetch for a quad issue processor. The reorder buffer should be accessed based on the thread ID. The important thing to consider is that the instructions can be of any combination.

   For example, the combination can be 1st and 2nd instructions or all the 4 instructions or only 4th instruction or 3rd and 4th instruction and many more. So, dependency checking and operand fetching should be made based on that.

4. After the operands are fetched, issue packets are formed that contain all the information that needs to be sent to the reservation station. The issue packets contain information about the operands fetched, ROB number allocated, type of instructions, thread ID, destination operand, program counter, prediction bits, etc.

5. After formation of issue packets, the valid bit for the issued instruction should be updated in the instruction queue buffer as INVALID. If the 4th instructions valid bit is made INVALID, then the head pointer will be incremented by one making the slot as empty.

6. The reorder buffer entries for which the instructions are allotted are made occupied and the tail pointer of the reorder buffer corresponding to the thread ID is updated accordingly based on the number of instructions issued.

7. The issue packets that were formed are sent to the reservation station entries that are empty and corresponding slot in the reservation station are made occupied.

CHAPTER 4

OUT OF ORDER EXECUTION

This chapter begins with how the instructions are dispatched to the execution units. It explains the methods to select the oldest set of ready instructions during dispatch. Then it describes about how the CDB is designed to forward the results of 4 execution units. Finally, it talks about committing multiple instructions at the same time.

4.1 Dispatch of instructions to Execution units

In out of order execution of instructions, the instructions in reservation station are sent to the execution unit whenever all the operands are available as we discussed in Chapter: 2. The main problem comes when choosing a set of instructions among a number of ready instructions. We cannot randomly send instructions that are ready to the execution units. It will affect the performance of the processor if we send the ready instructions to the execution units in a random manner. We need to choose the oldest instructions among the ready instructions. Oldest instruction refers to those instructions which entered the reservation station early compared to other instructions. There are many ways of selecting the oldest instructions from a set of ready instructions and some of the methods are discussed below.

4.1.1 Methods in selecting the oldest set of ready instructions during dispatch

There are many ways in which the oldest set of instructions can be selected from a set of ready instructions. Three ways had been suggested to the processor team in which the instructions from the instruction queue buffer can be filled into reservation station and dispatched to the execution units. Finally, one method was chosen based on the hardware complexity and implementation issues. The three ways are

- Issuing at the tail pointer of the reservation station and dispatching from the head pointer.
- Issuing anywhere in an empty reservation station slot and using extra buffer for dispatch.
- Issuing anywhere in an empty reservation station slot and using ROB number for dispatch.

4.1.1.1 Method 1: Issuing at the tail pointer of the reservation station and dispatching from the head pointer

In this method, the instructions that are issued are always filled at the tail pointer of the reservation station and are dispatched to the execution units starting from the head pointer as we discussed in Chapter : quad issue.

The main advantage of using this method is during the dispatch of the instructions to the execution units. During dispatch of the instructions, choosing the oldest instructions among the ready instructions is very easy in this method as the instructions starting from the head pointer of the reservation station will always be the oldest. The instructions in the reservation station are checked for ready condition starting from the head pointer and going toward tail pointer and they are sent to the execution units.

The main problem in this method is that the tail pointer will quickly reach the last entry of the reservation station while filling the reservation station since we are always filling the reservation station at the tail. This problem can be solved by shifting the instructions in the reservation station. Shifting of the instructions is done by calculating the total number of empty slots below a filled instruction and shifting by the calculated number of times. Figure: 4.1 shows how the instructions in the reservation station are shifted.

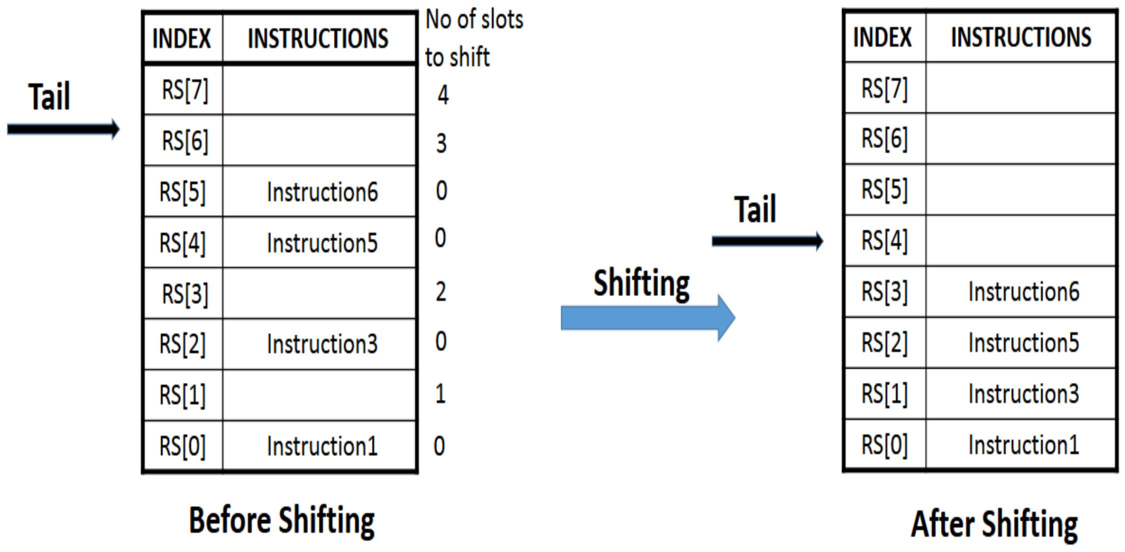


Figure 4.1: Shifting the instructions of Reservation station in Method1

As we see in the Figure: 4.1, the empty slots are numbered stating from 1 and each empty slot is assigned a number indicating the total number of empty slots below it (including the same empty slot also). The filled reservation station slot entries are always assigned a number equal to 0. Based on the calculated numbers, the filled instructions are shifted by a number that is allotted to last empty slot below it. For example, in the Figure: 4.1, Instruction 1 shifted by 0 slots

Instruction 3 shifted by 1 slot

Instruction 5 shifted by 2 slots

Instruction 6 shifted by 2 slots

So, by this manner the instructions are shifted in every clock cycle to maintain the reservation slots efficiently.

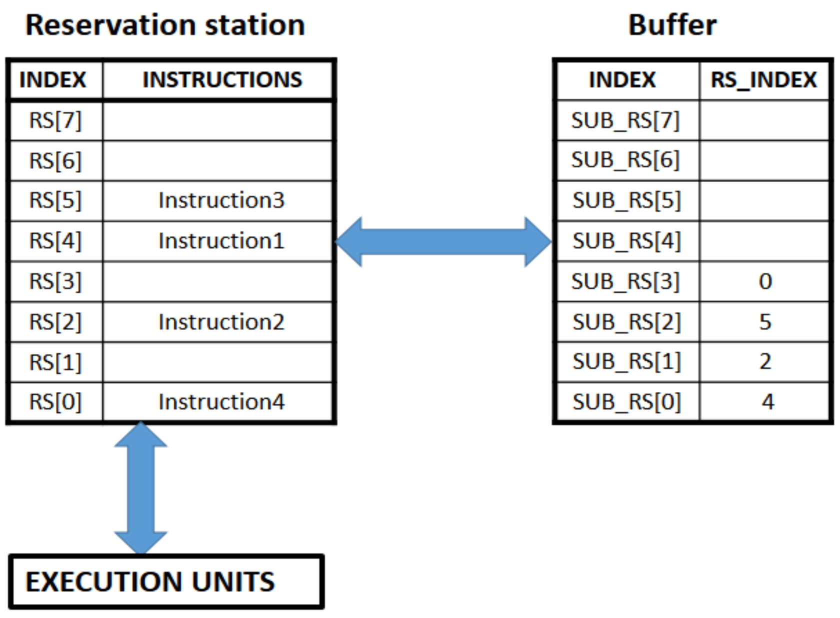
4.1.1.2 Method 2: Issuing anywhere in an empty reservation station slot and using extra buffer for dispatch

In this method, the instructions that are issued can be filled in any empty slots of the reservation station. There is no head and tail pointers in this method as we are filling anywhere in the reservation station.

The main advantage of using this method is there is no need of shifting of the instructions when the instruction gets dispatched to the execution units. Shifting of instructions takes a lot of hardware and usually shifting operation is avoided.

The problem for this method comes during dispatch of ready instructions into the execution units when they are ready in the reservation station. Since all the instructions in the reservation station are not in order they are issued and they are filled in random empty slots, we need to find an efficient way of choosing a set of instructions among the ready instructions. We need to choose the oldest instructions among the ready instructions and send it to execution units based on the availability of the functional units.

The above problem is actually sorted out by having a separate buffer that has the reservation station numbers of the instructions that were issued in order. Choosing the oldest instructions is accomplished by using the reservation number stored in the separate buffer as index for the reservation station starting from the head slot of the buffer and the corresponding instruction in the reservation station is checked whether all the operands are available. If they are available, the instruction is sent to the execution unit else the next entry in the buffer is checked. Here also, shifting of slots is necessary but only the reservation station numbers are shifted instead of the shifting the entire slot as in method1. Figure: 4.2 shows the method2 implementation of reservation station.



INDEX	INSTRUCTIONS
RS[7]	
RS[6]	
RS[5]	Instruction3
RS[4]	Instruction1
RS[3]	
RS[2]	Instruction2
RS[1]	
RS[0]	Instruction4

INDEX	RS_INDEX
SUB_RS[7]	
SUB_RS[6]	
SUB_RS[5]	
SUB_RS[4]	
SUB_RS[3]	0
SUB_RS[2]	5
SUB_RS[1]	2
SUB_RS[0]	4

Figure 4.2: Method2 implementation of Reservation Station and Buffer for Dispatch

4.1.1.3 Method 3: Issuing anywhere in an empty reservation station slot and using ROB number for dispatch

The method for issue is same as Method2 but the procedure to dispatch the instructions to the execution units is different. Reorder buffer number is used to dispatch the instructions to the execution units. Since the instructions are issued to the reservation station in order and also the instructions are assigned entries in reorder buffer in order, reorder buffer number can be used to determine the oldest instructions among the available ready instructions. Figure: 4.3 shows the use of reorder buffer number to dispatch the instructions to the execution units.

INDEX	THREAD_ID	ROB_NUMBER	INSTRUCTIONS
RS[7]			
RS[6]	0	2	Instruction5
RS[5]	2	3	Instruction3
RS[4]	1	1	Instruction1
RS[3]			
RS[2]	1	0	Instruction2
RS[1]			
RS[0]	3	2	Instruction4

RESERVATION STATION

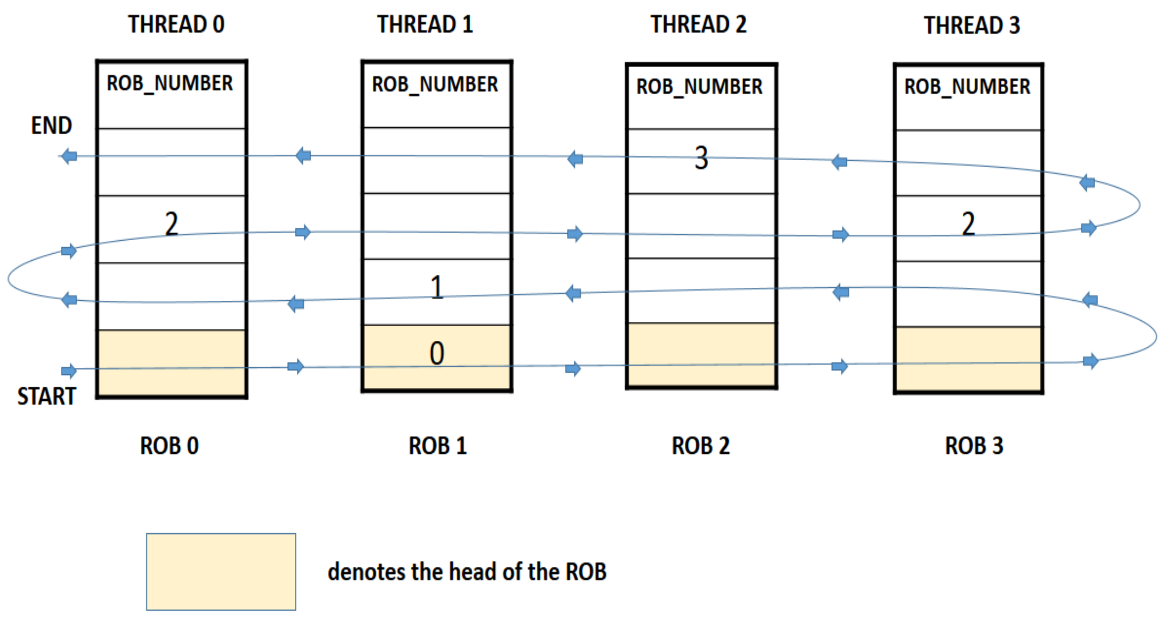


Figure 4.3: Method3 implementation of Reservation station and dispatch procedure

The reorder buffer numbers starting from the head of ROB 0 is compared with the ROB number of the ready instructions. If it matches and the instruction is ready, then that instruction is sent to the corresponding execution unit and the corresponding slot in the reservation station is made empty. Likewise, the ROB number is traversed as in the Figure: 4.3 and the instructions are sent to the execution units. The ROB numbers are traversed till it reaches the tail of the ROB 0.

The advantage of using this method is that there is no need of shifting the instructions as in Method 1 and Method 2. But, this method needs a lot comparators to compare the ROB numbers in ROB and in the Reservation station. Also, as the size of the ROB increases, the number of comparators required will also increase tremendously.

4.1.1.4 Design Decision: Choosing the method for selecting the oldest set of ready instructions during dispatch

While choosing between the methods, one needs to consider the hardware complexity, speed and the area occupied. All the three methods has its own advantages and disadvantages. If we look into Method 1, shifting the entire register of size nearly equal to 400 bits will consume a lot of hardware and also shifting is dynamic process. In Method 2, shifting is done only to a register of nearly 4 to 5 bits in size depending on the size of the reservation station. Using Method 3 requires a lot of comparators and the number of comparators increases by large amount if the ROB size is increased.

All the three methods were implemented separately in Bluespec and synthesis results showed that the Method 2 occupies less area and had less hardware complexity compared to other methods. So, Method 2 was chosen and was implemented successfully in Bluespec.

4.1.2 Dispatching the selected instructions to the Execution units

Till now, we have discussed about how the oldest set of instructions are selected in the reservation station. After selecting an instruction, the instruction is sent to the corresponding execution unit based on the availability of the execution unit. If an instruction is sent to a particular execution unit, the execution unit is made busy so that the same execution unit is not selected for the next ready instruction. The execution units are duplicated so that more number of ready instructions can be sent from the reservation station whenever they are ready. So, considering this the execution units are duplicated four times. The ready instructions are sent to the execution units by checking first ALU0, then ALU1, then ALU2 and finally ALU3. The same is for FPU and BRANCH execution units. If all the execution units of one particular type are busy, then no instructions are sent to the execution units. Duplication of execution units mainly reduce structural hazards and more number of instructions can be issued in the same clock cycle. Once the execution units finishes executing, the results are written in to the result register. The result register holds the data unless the execution unit is released.

4.2 Common Data Bus

Tomasulo algorithm uses Common data bus (CDB) which is used to broadcast the results of the execution units to the reservation station to update the operands which are waiting for the results of the dependent instructions. The CDB keeps on monitoring the result register of the execution units every cycle. If the result in the result register is available, then it is written into the CDB and the corresponding execution unit is freed. During writing the result to the CDB, it also writes the result into corresponding Reorder buffer entry.

4.2.1 Design of Common Data Bus

CDB is implemented by using the multiplexer based Bus structure. It contains number of multiplexers that are connected between execution unit and reservation station to match the reorder buffer number in the reservation station and forward the results. It also forwards the thread ID along with the ROB number because it should only match the ROB number of the corresponding thread.

The important thing to consider in CDB is not all the results can be forwarded from the execution units to the reservation station at all time. At a time there may be a situation where more than one result is ready to be driven into the CDB. If we have a bus width of only one packet, only one result can be forwarded to the reservation station at a time and this decreases the performance of the processor. So, we need to increase the bus width so that many results can be forwarded at a time. Also, we cannot have a bus width equal to number of execution units to forward all the results in the worst case. We need to have a sufficient amount of bus width so that not much of hardware is used. Considering all these, the bus structure has been increased to four packets so that results of four execution units can be forwarded at a time. The Figure: 4.4 shows the bus structure of the common data bus.

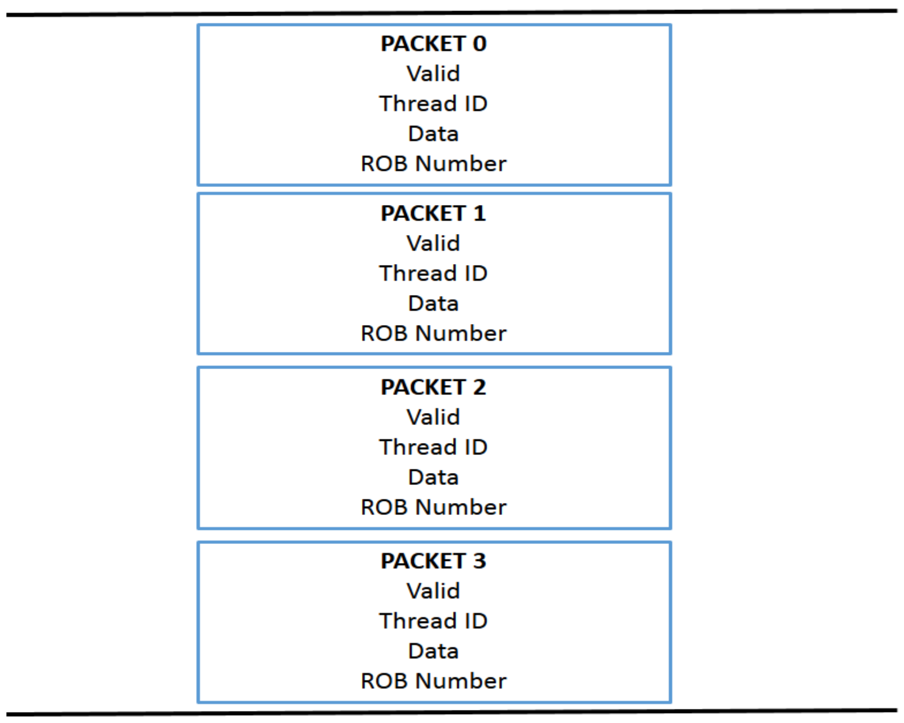


Figure 4.4: Bus structure of Common Data Bus

4.2.2 Updating the Operands in the reservation station

Let us see now how the operands are matched from the data forwarded by the CDB. Before the ROB number is matched in the reservation station, it first checks whether the operand valid is 1 or 0. If the bit is 1, then the source operand already has the data and the data is retained. If the bit is 0, then it starts matching with all packets.

At first, it checks whether the ROB number and thread ID of Packet 0 are equal to ROB number and thread ID in the reservation station. It also checks whether the packet is valid or not. If all the above three parameters are equal and valid, then the operand is updated with data and the operand valid is made as 1. If any of the three does not match, then the same steps are repeated for Packet 1, and then to Packet 2, and finally to Packet 3. If none of the packets matches, then the operand is retained with the ROB number already stored. When all the operand valid bits of all the operands are 1 for an instruction, then the instruction is ready to be sent to the execution unit.

The Figure: 4.5 shows how the operands are updated in the reservation station from the CDB.

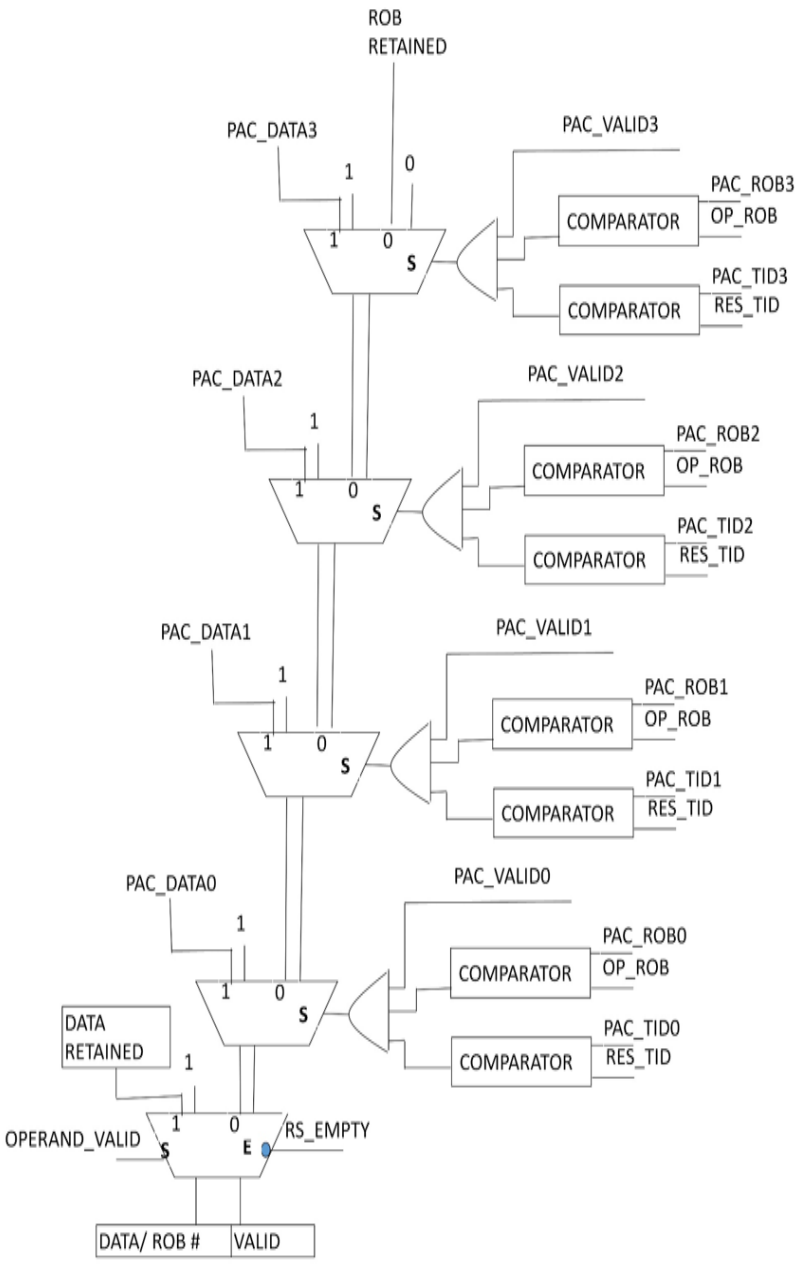


Figure 4.5: Updating operands in reservation station

4.2.3 Prioritizing the Execution units to drive the CDB

In the superscalar processor designed, there are 12 execution units: 4 of ALU, 4 of FPU, 4 of BRANCH and 1 of MEMORY type. We can drive the results of maximum of 4 execution units in the CDB since we have CDB of 4 packets. The problem comes when results of more than 4 execution units are ready to drive the CDB. We cant randomly the results of the execution units in the CDB since it will affect the performance of the processor. There should be some arbitration scheme to choose which of the execution units will drive the CDB. There are many scheduling algorithms which can be implemented to share resources like

- Fixed priority scheduling scheme
- Round robin scheduling scheme

In fixed priority scheduling scheme, a fixed priority is assigned to every execution units. The scheduler arranges the execution units which are ready based on the priority. Requests form the lower priority units will be interpreted by the higher priority execution units.

In round robin scheduling scheme, there is no fixed priority for the execution units to drive the CDB. The priorities keep on changing for every clock cycle. One execution unit may get priority in one clock cycle and the other may get in next clock cycle.

Of these two scheduling schemes, round robin scheduling scheme is chosen since it gives priority to all execution units. After choosing the scheduling scheme, we need to decide the execution units needed to be mapped to a particular packet in CDB. After a lot of analysis, the following order has been chosen and implemented.

Packet 0 - ALU0, FPU1, BRANCH2, MEMORY

Packet 1 - FPU0, BRANCH1, MEMORY, ALU3

Packet 2 - BRANCH0, MEMORY, ALU2, FPU3

Packet 3 - MEMORY, ALU1, FPU2, BRANCH3

The Figure: 4.6 shows about how the execution units are prioritized using arbiter.

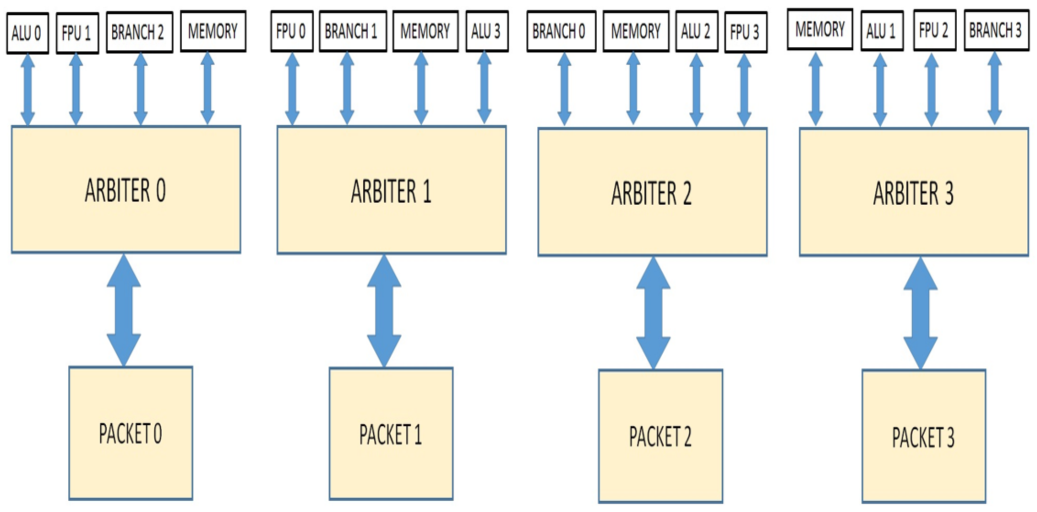


Figure 4.6: Prioritization of Execution units by using arbiter

Since Execution units 0 are used first, they are given more priority compared to others. When only execution units 0, 1, 2 are not available, then only we go and check for availability of execution unit 3. So, execution unit 3 is given least priority. The priority keeps on changing within the arbiter from one execution unit to another. Once an execution unit is selected by the arbiter, then the execution unit is released and its results are broadcasted to the bus. When none of the execution units of a particular arbiter are ready, then the valid bit of the corresponding packet is made 0.

4.3 Commit Stage

The instructions after driving the data in the CDB will also write the results in the corresponding ROB slot which is allocated during issue stage. In Commit stage, the instructions in the reorder buffer will write the result back into the register file. Write back into the destination register will always happen in order. An instruction in the ROB is allowed to commit only when that instruction reaches the head of the ROB and its results are available to be written in to destination register. When an instruction commits, the corresponding slot in the reorder buffer is freed and can be used during issue by other instructions.

4.3.1 Quad Commit

Since quad issue is implemented, it is necessary that the commit stage also write backs multiple instructions at the same time. We need to increase the maximum number of instructions that can be committed at a time. So, a quad commit is implemented that can commit maximum of 4 instructions at a time.

During commit, the valid bit in the destination register is updated as VALID only if the ROB number stored in the destination matches the head of the ROB. Otherwise, only the result is written into the destination register and tag field is updated as INVALID. Also, the head pointer of the ROB is incremented according to the number of instruction that are committed.

4.3.2 Important things to consider during quad commit

- When two or more instructions have the same destination register. We cannot write the register two or more times in the same clock cycle. To avoid this problem, the destination register is modified only for the last instruction among the group of instructions that has the same destination register.

- During commit of multiple instructions, an instruction can be committed only if the instruction before that instruction commits.
- In case of branch instructions, the instruction is committed only if there is no misprediction. If there is misprediction, the entire pipeline is flushed and the program counter of the instruction successor to the branch is loaded. If there is misprediction in the first instruction, the second instruction is not committed. This is applicable for third and fourth instructions also.
- The integer type instructions will commit into the integer register file and the floating point type instructions will commit into the floating point register file. While considering the instructions with same destination register, the instructions destination address should be matched only with corresponding type of register file.

4.4 Verification

This section deals about how the processor design that is build is verified in Bluespec System Verilog. It also describes about the test cases generated to verify the design.

4.4.1 Verification Setup

The Bluespec code that is written in .bsv format is given to the Bluespec compiler in the Bluespec Development Workstation (BSW). The Bluespec Compiler compiles and generates the Verilog code in .v format. The Verilog code is simulated in the ICARUS Verilog simulator to check for the functionality of the design. The Verilog code is also synthesized in Xilinx ISE to get the clock frequency and to know about the amount of hardware generated by the design. Figure: 4.7 shows the verification steps in Bluespec System Verilog.

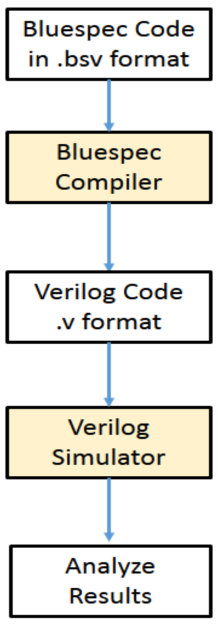


Figure 4.7: Verification steps in BSV

The functionality of the design is verified by using a test bench which provides a set of instructions given to the main module. display statements were written in the code in all the necessary places to monitor the functionality of the modules in various clock cycles.

4.4.2 Verification Strategy

The verification process mainly focused on verifying the functionalities of the processor. The necessary test instructions are directly loaded in to the Instruction queue buffer and the simulation is started. The test cases were generated to test the following critical aspects of the design.

Issue Stage

- Dependency check between instructions.
- Checking the issue of four instructions.
- Instructions with same destination address.
- Fetching of operands.
- Filling the instruction in empty slot of reservation station.

Dispatch stage

- Dispatch to all types of execution units.
- Contention for the execution units.

Common Data Bus

- Operand Forwarding.
- Contention for CDB.
- Bus arbitration.
- Writing into the ROB of corresponding threads.

Commit Stage

- Committing into integer and floating point register file.
- Commit of two or more instructions.
- Instructions with same Destination address.

Files associated with the design:

Top Module:

tomasulo.bsv - (11000 lines)

Other modules:

centralizedrs.bsv - (1300 lines)

iqbuffer.bsv - (200 lines)

reorderbuffer.bsv - (1000 lines)

alu0.bsv, alu1.bsv, alu2.bsv, alu3.bsv, fpu0.bsv, fpu1.bsv, fpu2.bsv, fpu3.bsv, branch0.bsv, branch1.bsv, branch2.bsv, branch3.bsv

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 : 32419 out of 301440

Number of Slice LUTs : 56751 out of 150720

Number used as Logic : 51845 out of 150720

Number of LUT Flip Flop pairs used: 30123

Timing Summary :

Minimum period: 5.907ns

Maximum Frequency: 169.279MHz

Minimum input arrival time before clock: 1.663ns

Maximum output required time after clock: 0.777ns

Maximum combinational path delay: 0.722ns

4.4.4 ASIC Synthesis Report

Technology used: **Synopsys 65 nm**

Library used: **uk65lscllmvbbl 110c0 bc**

Area Utilization Summary :

Number of ports: 602

Number of nets: 582267

Number of cells: 581328

Number of combinational cells: 533102

Number of sequential cells: 48226

Number of buf/inv: 52199

Number of references: 385

Combinational area: 1224156.604563

Buf/Inv area: 72933.122357

Noncombinational area: 338606.279684

Total cell area: 1562762.884247

Timing Summary :

Minimum period: 1.5 ns

Maximum Frequency: 670 MHz

Power Consumption:

Cell Internal Power: 239.1141 mW

Net Switching Power: 9.2372 mW

Total Dynamic Power: 248.3513 mW

Cell Leakage Power: 2.7257 mW

CHAPTER 5

CONCLUSION AND FUTURE WORK

Based on the Tomasulo algorithm, stages such as issue, dispatch, CDB, commit stages of the CPU core are successfully implemented in Bluespec System Verilog. In order to achieve a higher Instructions per Cycle (IPC), quad issue was implemented and all dependencies were taken into consideration. Centralized Reservation station was implemented to achieve high degree of functional unit utilization. Execution units were duplicated so that more ready instructions can be dispatched. Common data Bus which can forward the results of four execution units was implemented to avoid contention for data busses. Since quad issue was implemented, quad commit was also implemented to commit four instructions at a time. The entire design was synthesized in Xilinx and a clock frequency of 170 MHz was achieved.

Future Work:

In the existing design, the following things needed to be modified or added to enhance the performance of the processor:

- The issue unit takes a lot of hardware and it needs to be optimized.
- The memory unit was not incorporated in the design and it needs to be designed and integrated into the CPU core.
- Exception handlers were not implemented and it needs to be designed and added into the CPU core.
- Optimization of dispatch stage of reservation station is necessary as the hardware will tend to increase tremendously with the existing design with the increase in slots in the reservation station.

REFERENCES

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

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

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

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

[5] User Level RISC-V ISA, *University of California, Berkeley*, revision: 2 ed., 2014.