Implementation of Execution Units of a 64 bit RISC Processor

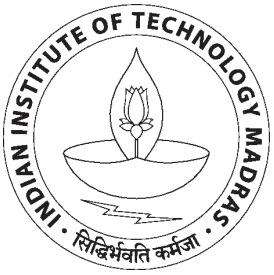
A Project Report

submitted by

VIKAS KUMAR S

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 Execution Units of a 64 bit RISC Processor**, submitted by **Vikas Kumar S**, 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

The successful completion of this project work would not have been possible without the guidance, support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to them.

First and foremost I offer my profound gratitude to my guide, **Dr. V. Kamakoti** for his exemplary guidance, monitoring and constant encouragement throughout my project. I also take this opportunity to express a deep sense of gratitude and thanks to Mr. G.S. Madhusudan whose energy, enthusiasm and constant support inspired me from the beginning till the end of the project. The invaluable inputs and suggestions from them were instrumental in steering the project work in the right direction.

My special thanks and deepest gratitude to Suresh, Neel, Naveen, Abhishek, Avinash, Bhasker, Sahoo, Amogh, Rahamthula and Chidhambaranathan who have been very supportive and have provided friendly atmosphere in lab.

I would like to thank my parents for supporting and encouraging me throughout my life and dedicate this project to them.

ABSTRACT

KEYWORDS: 64-bit processor, execution unit, decoder, ISA

The processor design team in RISE lab has been involved in development of processor for defence and security related applications. As the instruction set architecture plays an important role in implementation of processor, the study on the ANUPAMA, MIPS and Power Instruction Set architectures were carried out and Power ISA was chosen for implementation, as it is one of the most popular ISA for embedded and desktop PC design and also for its exceptional branch handling capability. It also provides multimedia and signal processing support with Vector-Scalar Instruction Set Architecture. The project deals with the architecture and design of the fixed-point scalar execution units and decoder of a 64-bit RISC processor. The design has been modeled in Bluespec System Verilog HDL and functional verification policies adopted for each unit have been described.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
ISA	Instruction Set Architecture
NVMe	Non Volatile Memory Express
PCIe	Peripheral Component Interconnect Express
GPR	General Purpose Register
CR	Condition Register
XER	Exception Register
LR	Link Register
CTR	Count Register
LSB	Least Significant Bit
MSB	Most Significant Bit
FSB	Finished Store Buffer
ROB	Re-order Buffer
EA	Effective Address
CIA	Current Instruction Address
NIA	Next Instruction Address
BSV	Bluespec System Verilog
RTL	Register Transfer Level
MMU	Memory Management Unit

CHAPTER 1

Introduction

1.1 The Overall Architecture

The overall architecture is as shown in Figure 1.1.

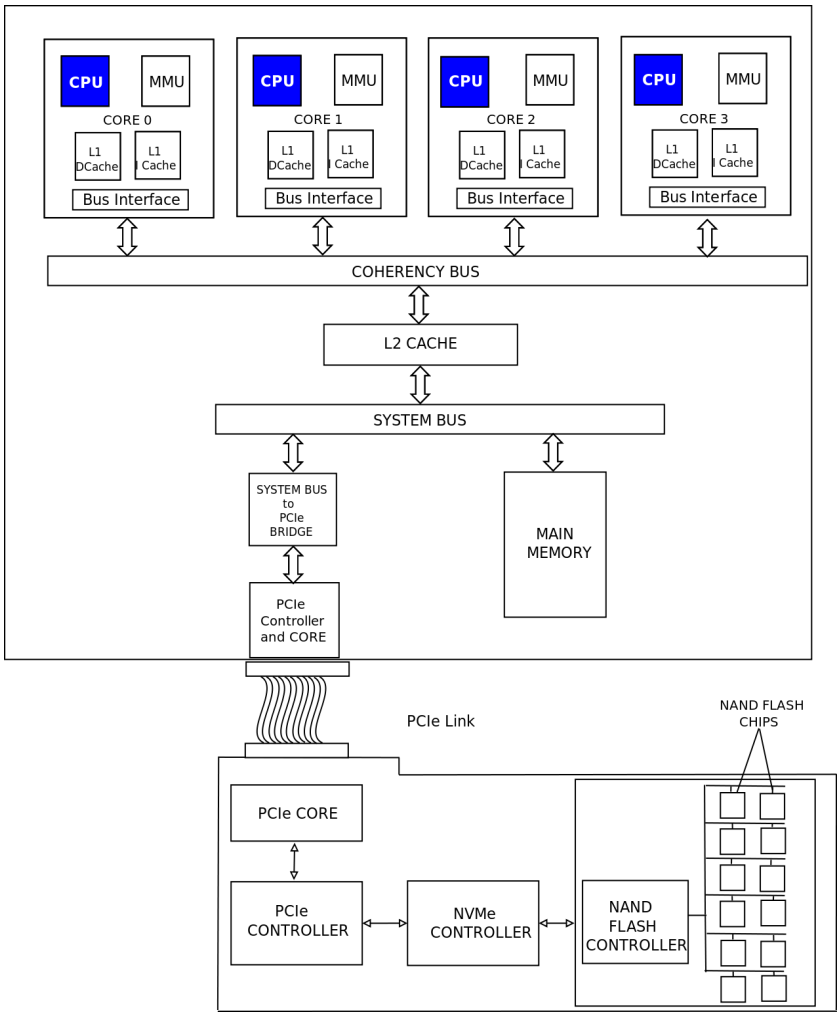


Figure 1.1: Overall Architecture

The processor design team in RISE lab is involved in development of processor for defence applications, the overall view of which is as shown in figure 1.1. It has four cores (quad-core) and each core is composed of a CPU unit, Memory Management Unit, L1 Dcache and L1 Icache. The CPU core is designed to support 64-bit PowerPC Instruction Set Architecture. It supports dual issue and out of order execution. It has two level cache hierarchy and supports cache coherency at L1 cache level, with coherency bus. It also has an on-chip main memory (DRAM). It has a Memory Management Unit for better data transfer between main memory and cores of the processor. Then in order to interact with external device there is a NVM Express based I/O subsystem with PCI Express interface.

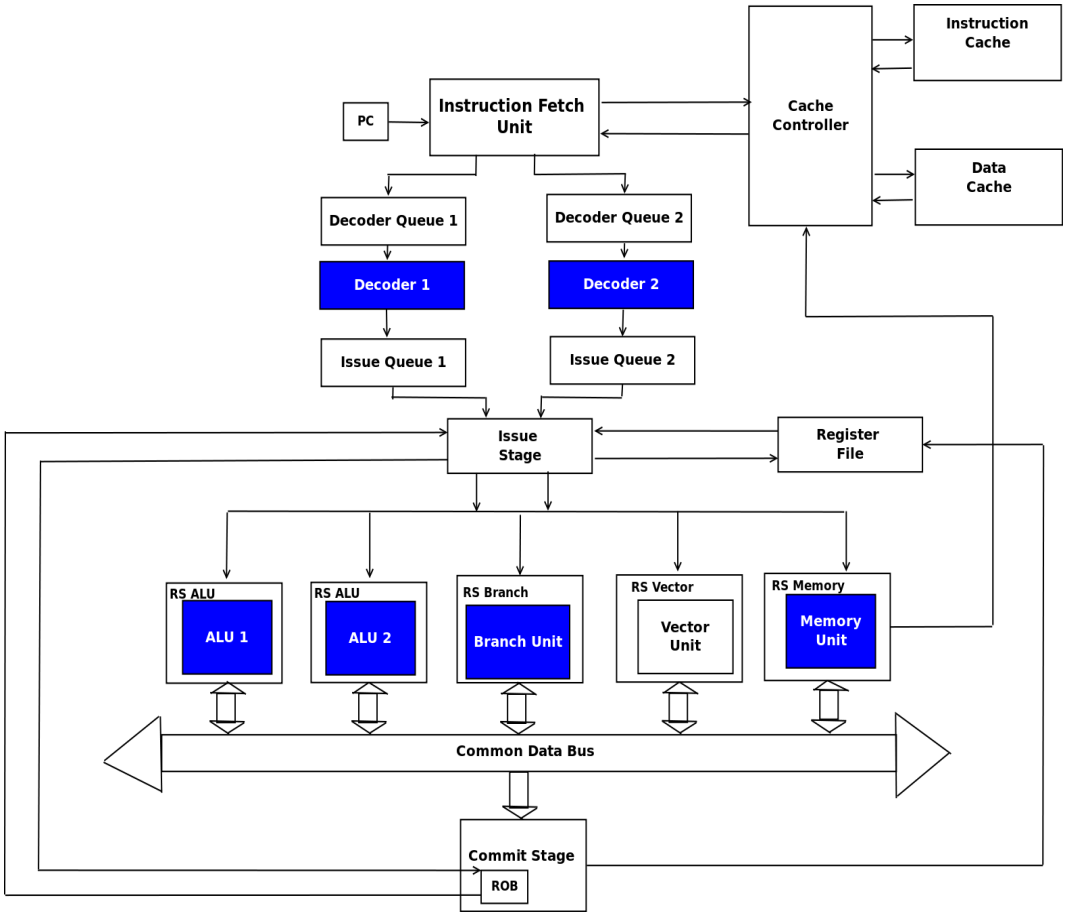


Figure 1.2: Stages of a pipelined processor

My contribution :

Different stages of pipelined processor is as shown in Figure 1.2. This is a pipelined superscalar processor with the instruction fetch, decode, issue, execution and commit stages. My work deals with architecture and design of the fixed-point scalar execution units and decoder of a this 64-bit RISC processor. Highlighted portion in the figure is my contribution to the development of targeted overall processor architecture.

1.2 Overview of content

Chapter-2 gives the background study of the Instruction Set Architecture and the role of execution unit along with the introduction to Bluespec System Verilog.

Chapter-3 is the introduction to the Power instruction set architecture.

Chapter-4 includes the design and implementation of various execution units and decoder.

Chapter-5 concludes with a short description on the future work that can be done.

Appendix A provides the set of all fixed-point scalar instructions that have been implemented along with their opcodes.

CHAPTER 2

Background

2.1 Instruction Set Architecture

Instruction Set Architecture (ISA) is the portion of the processor or computer that is available to the compiler writer or programmer. They play important role in building a processor [1]. There are three areas where these ISA are applied namely:

- Desktop computers: Here the importance is given to performance of programs with integer and floating point data types.
- Servers : These are used primarily for database, file server, and Web applications and some time-sharing applications.
- Embedded applications : Here the importance is given to power and cost hence code size is important and some classes of instructions like floating point becomes optional.

Broadly the ISA is classified into four types :

- Stack Architecture : The operands in this architecture are implicitly on the TOS (top of the stack).
- Accumulator Architecture : In this architecture one of the operand is implicitly accumulator.
- Register-Memory Architecture : The accessing of memory can be part of any instruction in this architecture.
- Load-Store Architecture : The accessing of memory can be done only with load store instructions.

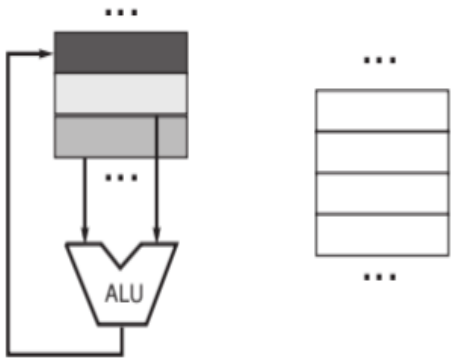


Figure 2.1: Load-Store Architecture

The most popular and the one chosen for implementation is the Load-Store Architecture as shown in Figure 2.1. The major reasons for the success of general purpose register (GPR) computers are that , registers like other forms of storage internal to the processor are faster than memory and registers are more efficient for a compiler to use than other forms of internal storage [1].

- Advantages : Simple, fixed-length instruction encoding, simple code generation model and instructions take similar numbers of clocks to execute.
- Disadvantages : Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density may lead to larger programs.

2.1.1 Memory Addressing

Usually the instruction sets are byte addressed and can provide access to byte, halfword, word and doubleword.

The byte ordering is done in one of the following ways [1]:

- Little Indian : In this type of ordering the byte with address ending with ”000” is put in the LSB of the double word and so on.

- Big Indian : In this type of ordering the byte with address ending with ”000” is put in the MSB of the double word and so on.

Whenever there is a need to access the memory of size larger than byte then it should be usually aligned. An access to an object of size s bytes at byte address A is aligned if A mod s = 0.

Addressing Modes

Addressing modes are the ways by which addresses are specified by the instruction.When a memory location is used, the actual memory address specified by addressing mode is called effective address [1]. Some of the addressing modes widely used are:

- Register
- Immediate
- Displacement
- Register indirect
- Direct
- Indexed
- Memory indirect
- Autoincrement
- Scaled
- Autodecrement

Each of the addressing modes have their own uses like register addressing mode is used when the value is in the register, immediate addressing mode is used for constants, displacement mode for accessing local variables, register indirect for accessing using the pointer. Indexed addressing mode is used in array handling while direct mode is used

for accessing static data. Displacement and immediate addressing modes are mostly used addressing modes. In these cases choosing the displacement field sizes is important because they directly affect the instruction length. Because of popularity, we would expect a architecture to support at least displacement, immediate, and register indirect addressing modes. Also we would expect the size of the address for displacement mode to be at least 12 to 16 bits and we would expect the size of the immediate field to be at least 8 to 16 bits.

2.1.2 Type of operands and operations in instruction set

The most common form of operands are byte (8 bits), half word (16 bits), word (32 bits), double word (64 bits), single and double precision floating point. Integers are represented in two's complement form. Sometimes there is also use of decimal operands. The operators supported by most instruction set architectures can be categorized as [1]:

- Arithmetic and logical
- Data transfer
- Branch Control
- System
- Floating point

Most of the computers will provide support to all of these operations. The Arithmetic and logical operations are like add, and etc. The data transfer operations are the load and store. The control operations are branch conditionally , jump etc. System functions are the system call etc while the floating point operations involve floating point add, multiply etc. For a control flow instruction destination address must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases procedure return being the major exception, since for return the target is not known at compile time. The

most common way to specify the destination is to supply a displacement that is added to the program counter (PC). Control flow instructions of this sort are called PC-relative. It is advantageous because the target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independently of where it is loaded. This property, called position independence, can eliminate some work when the program is linked and is also useful in programs linked dynamically during execution. In case of branch instructions large number of the comparisons are simple tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a compare and branch instruction is being used. The three primary techniques are:

- Condition code : These test the bits which are set by other alu operations.
- Condition register : These test a particular register to compare with the result.
- Compare and branch : Here compare forms a part of branch.

2.1.3 Encoding an Instruction Set

All of the above mentioned choices will effect how the instructions are encoded. This not only affects the length of the program but also the implementation of the processor. The good encoding helps in quickly decoding the instruction and finding its operands and operation. The important decision is how to encode the addressing modes with the operations. This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Number of register and number of addressing modes have significant impact on size of the instruction as the register field and addressing mode field may appear many times in a single instruction.

Points to focus on when encoding the instruction set are the desire to have as many reg-

isters and addressing modes as possible [1], its impact on instruction size (average) and also a suitable length that will be easy to handle in a pipelined implementation. Its better if the instructions are multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size. Three types of encoding are :

- Variable
- Fixed
- Hybrid

Variable : Since the instruction length is variable it can support all kinds of addressing modes and operations but the decoding becomes the problem. E.g.,Intel 80x86, VAX.

Fixed : This combines the operation and the addressing mode into the opcode. It will have only a single size for all instructions and works best when there are few addressing modes and operations. E.g., MIPS, PowerPC, Arm.

Hybrid : The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.E.g., IBM 360/370, MIPS16, Thumb.

2.2 Advantages of Power ISA

There is one register that always has the value 0 when used in address modes [1], the absolute address mode with limited range can be synthesized using zero as the base in displacement addressing. Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of PowerPC architectures. Register + offset and update register is one of

the features that is unique.

PowerPC using PowerISA uses four condition codes: less than, greater than, equal, and summary overflow, but it has eight copies of them. This allows the PowerPC instructions to use different condition codes without conflict, essentially giving PowerPC eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer op is followed by a compare to zero that sets the first condition register. PowerPC also lets the second register be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers , allowing more complex conditions to be tested by a single branch. PowerPC adds 64-bit right shift, load, store, divide, and compare and has a separate mode determining whether instructions are interpreted as 32 or 64 bit operations In the Endian row, Big/Little means there is a bit in the program status register that allows the processor to act either as Big Endian or Little Endian This can be accomplished by simply complementing some of the least-significant bits of the address in data transfer instructions [1].

- Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, PowerPC puts the address into a special register called the link register. Since many procedures will return without calling another procedure, link doesnt always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.
- PowerPC has a count register to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.
- Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting

the target address early in the pipeline PowerPC architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (BCLR, BCCTR).

- Load multiple and store multiple save or restore up to 32 registers in a single instruction.
- Rotate with mask instructions support bit field extraction and insertion. One version rotates the data and then performs logical AND with a mask of ones, thereby extracting a field. The other version rotates the data but only places the bits into the destination register where there is a corresponding 1 bit in the mask, thereby inserting a field.
- Algebraic right shift sets the carry bit (CA) if the operand is negative and any 1 bits are shifted out. Thus a signed divide by any constant power of 2 that rounds toward 0 can be accomplished with a SRAWI followed by ADDZE, which adds CA to the register.
- SUBFIC computes (immediate RA), which can be used to develop a ones or twos complement.
- Logical shifted immediate instructions shift the 16-bit immediate to the left 16 bits before performing AND, OR, or XOR.

2.3 Basic Processor Architecture

The part of the computer that performs bulk of the data processing operations is called the CPU (central processing unit). The CPU is formed of three basic components like register set which is used to store the data which is used during the execution of instructions. The alu unit required to perform the required micro operations for executing the instructions.The control unit will supervise the transfer of information among the registers and instructs the alu as to which operation to perform. Pipelining is a technique of decomposing a sequential process into the sub operations and each of these sub operations are executed in dedicated segment which operates concurrently with all other segments. A collection of processing segments through which the information flow is called pipelining.

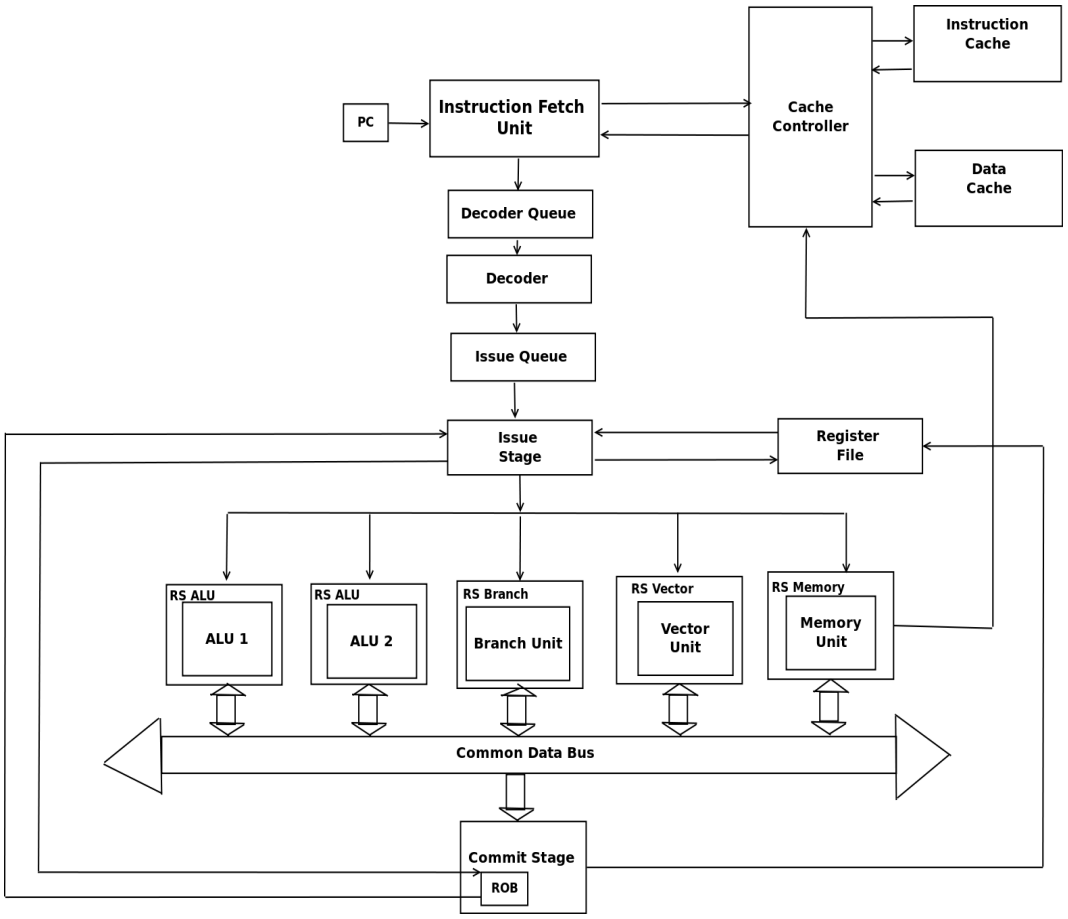


Figure 2.2: Basic Processor Architecture

The result obtained from each segment will be passed onto other segment. simplest way to view a pipeline structure is to imagine each segment consisting of the input register followed by a combinational logic circuit. The register holds the data and the combinational circuit performs the required suboperations in the segment.

The basic pipelined RISC Processor will have the following stages in it.

Instruction Fetch

- This stage supplies the instructions to the rest of the pipeline.
- There can be use of instruction cache containing the recently used instructions which will reduce latency due to memory access.
- Program counter will provide Icache with the instruction address.

- The default instruction fetching method is to increment the program counter by 4 and use the incremented program counter to fetch the next instruction.
- In case of branch instruction which redirect the flow of control, however, the fetch mechanism must be redirected to fetch instructions from the branch target.

Instruction Decode

- During this stage the instruction from the fetch unit which is present in the fetch buffer is decoded based on the instruction set architecture chosen.
- This stage will also classify the instruction into different types.
- Then it forms the bundle of data involving the addresses of operands and the destination address.
- The decode bundle is then forwarded to the issue stage.

Instruction Issue

- The instruction from the decode stage enter issue stage along with additional information from the decode stage.
- This stage will allot the reservation station to each instruction based on its type and the reservation station will execute the instruction.
- The dispatch of the instruction will occur only if both the slot in reservation station and the reorder buffer (ROB) are empty.
- The operands are also fetched in this stage based on the addresses obtained from the previous stage.
- First register file will be checked to obtain the data , but if its not present then it will have the ROB entry which will provide the valid value.
- If neither ROB nor the register file has the valid value then ROB number is passed to the reservation station as operand sources and this method is called register renaming.
- This ROB number, which is passed to the reservation station along with other data, is used to put the result from the reservation station (result coming from the execution unit) into the specific ROB entry.

Instruction Execution and write

- Now the instruction is in the reservation station and if any operand is missing then it continuously monitor the particular ROB number.
- As soon as the required ROB entry receives the valid result then the value is forwarded to the reservation station to update the operands of instruction being executed.
- This step checks for RAW hazards. Only when both or all operands are available the reservation station will send them to the functional units for execution purpose.
- The execution unit performs the required operation based on the instruction.
- The execution unit will compute the arithmetic and logical results for the alu instructions. It will compute the effective address in case of load/store instruction along with the value to be stored. Then it computes the new program counter value in case of control flow instructions.
- The results from the functional units will then be sent to the ROB to the entry specified by the ROB number of the destination address which was assigned during issue phase. This process of writing the result into the ROB is known as write phase.

Instruction Commit

- The final stage of the instruction is the commit stage.
- The main purpose of this stage is to maintain the sequential flow of execution even though the actual execution of instructions may occur non sequentially which is nothing but out of order execution.
- Once the execution of instruction is done the results from various functional units are gathered and is sorted as per the program issue order.
- Then the results are written into the register file in same order.
- Reorder buffer is a circular queue data structure.
- The actions of the commit stage will now depend on the type of instruction stored in the ROB. In case of normal arithmetic instructions only the register file will be updated.
- In case of control flow instructions it may be required to flush the pipe and start fetch from a new program counter. Committing a store operation requires accessing the external memory/device and updating it with the respective data.
- Another feature important to the Commit stage is the operand forwarding mechanism. Each time an instruction is committed, the data that is to be written into the register file is sent to all reservation station along with the ROB number of the ROB entry which is committed.

2.4 Bluespec System Verilog

BSV (Bluespec SystemVerilog) is a language used in the design of electronic systems (ASICs, FPGAs and systems)[2]. BSV is used across the spectrum of applications like processors, memory subsystems, interconnects, DMAs and data movers, multimedia and communication I/O devices, multimedia and communication codecs and processors, signal processing accelerators, high-performance computing accelerators, etc. BSV is also used across marketsfrom low-power, portable consumer items to enterprise-class server-room systems.

Some of the major application areas are :

- Executable specifications
- Virtual platforms
- Architectural modeling
- Design and Implementation
- Verification Environments

Some of the key features of BSV are listed below :

- Bluespec presents the hardware designer an new way to simplify the complexity of constructing control logic while retaining full control over the architecture and performance of the design[3].
- For behavioral description Bluespec System Verilog introduces rules and interface methods which helps in expressing complex control and concurrency logic.
- Parallelization is achieved implicitly in Bluespec.
- Bluespec provides higher level of abstraction than Verilog, VHDL , System Verilog.
- The design is simulated and debugged using Bluesim simulation tool.
- It is best suited for hardware designers designing complex control logic and data path such as processing elements, DMA controller, memory controller etc.

- With Bluespec, the quality of results is comparable to hand-coded RTL.
- It has a powerful static checking, parameterization and static elaboration.
- Familiar module hierarchy from Verilog.
- Atomic Transactions (Rules) instead of verilog always blocks. A rule is an action guarded by boolean condition.
- Atomic Methods instead of Verilog port lists. There are three kinds of methods namely Value , Action and ActionValue method.
- More powerful Types and strong Type-checking.
  - Function arguments are of correct type
  - Modules parameters, interface is of correct type
  - In case of case mismatch, issues an error message
  - No automatic sign or zero extension also no automatic truncation
- Rich parameterized libraries of interconnect IPs.
- Fully synthesizable.
- Types play a important role in Bluespec System Verilog.
- Some of the basic types are Integer,Bool,String,bit and it also allows defining of new types.
- Bluespec also has powerful notation for construction of state machines with automatic generation of necessary state variables and registers.

CHAPTER 3

Power Instruction Set Architecture

The Power Instruction Set Architecture (ISA) 1.10. The original architecture defined in the 1990s by Apple, IBM, and Motorolas Semiconductor Products Sector (SPS) (now Freescale). This mature architecture continues to form the basis for developing PowerPC processors that use Freescales G2, e300, and e600 processor cores. Power Architecture processors are fundamentally a classic load/store RISC architecture. In general, such RISC architectures have fixed-width instruction length, and contain a substantial number of selectable registers to provide inputs and outputs of computations. The instructions that are defined are generally less complex, with the expectation that those instructions can be implemented simply, and that the implementation will more easily be able to execute instructions out of order, resulting in an implementation that is smaller, lower in power usage, and less complex. The architecture supports demand-paged virtual memory as well other memory management schemes that depend on precise control of effective-to-physical address translation and flexible memory protection. The mapping mechanism consists of software-managed unified (translate both instruction and data references) TLBs that support variable-sized pages with per-page properties and permissions.

Implementations of EIS may be either 64-bit implementations or 32-bit implementations [4]. 64-bit implementations provide for 64-bit effective addresses and provide 64-bit registers, and instructions for manipulating 64-bit addresses and 64-bit integer data. 32-bit implementations provide for 32-bit effective addresses and provide 32-bit registers, and

instructions for manipulating 32-bit addresses and 32-bit integer data. There are vector units that support the Altivec instructions.

3.1 Register Model

In order to implement the fixed point scalar instructions the registers that are involved are as described below.

General Purpose Registers : In order to perform the integer operations there are 32 general purpose registers named as GPR0 to GPR31. Since there are only 32 GPRs the 5-bit fields are provided by the instruction format for specifying the the GPRs.

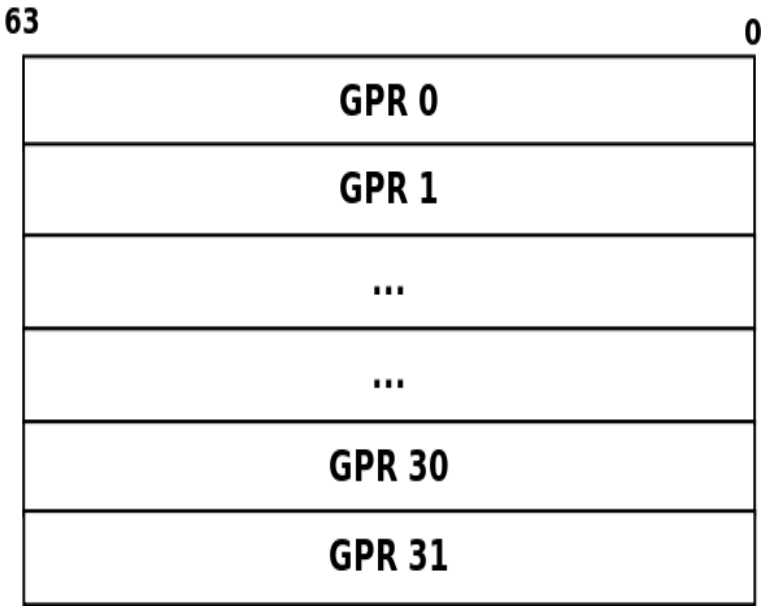


Figure 3.1: General Purpose Registers

Integer Exception Register : The bits of this register is set based on the result of the integer operation. It is a 64 bit register. The important bits that are set are:

- **Bit 31 :** Summary Overflow
- **Bit 30 :** Overflow
- **Bit 29 :** Carry

The Summary overflow bit is set whenever the overflow bit is set by the instruction and it remains set till it is cleared by the move to special register instruction.Executing an mtspr instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.

The overflow bit is set whenever a overflow occurs in a arithmetic instruction. The instruction in order to set the overflow bit must have the OE field to be equal to 1. The OV bit is not altered by compare or other instructions. It is to 1 if the carry out of bit 63 is not equal to the carry out of bit 62, and set it to 0 otherwise.

The Carry bit (CA) is set during execution of certain instructions like Add Carrying, Subtract From Carrying, Add Extended, and Subtract From Extended types of instructions and set it to 1 if there is a carry out of bit 63 , and set it to 0 otherwise. Shift Right Algebraic instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by Compare instructions, or by other instructions.Remaining bits are reserved.

Condition Register : This is one of the important register for the branch operation[5]. It is a 32 bit register and it provides the mechanism for testing and branching. CR bits are grouped into eight 4-bit fields, CR0 to CR7, which are set as follows:

- Specified CR fields can be set by a move to the CR from a GPR (mtcrf, mtocrf).
- A specified CR field can be set by a move to the CR from another CR field (mcrf).
- CR0 can be set as the implicit result of an integer instruction if the RC bit in instruction is set.
- CR1 can be set as the implicit result of a floating-point instruction.

- CR6 can be set as the implicit result of a vector floating-point instruction.
- A specified CR field can be set as the result of either an integer compare instruction.
- Any CR bit can also be set to 1 or 0 based on various condition register logical operations.

The first three bits of CR Field 0 (bits 3:1 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 0 of the Condition Register) is copied from the SO field of the XER. If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

- **Bit 0:** Summary overflow
- **Bit 1:** Zero
- **Bit 2:** Positive
- **Bit 3:** Negative

Here it should be noted that for the bluespec implementation the bit order naming is done as 31 : 0 from left to right when compared to that of the PowerPC specification which is 0 : 31 . This way of naming is followed everywhere and this is the thing to be noted while giving data to the processor from the assembly code. Just reversing of each bit field is to be done , then the instructions work fine here.

Link Register : The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions for which LK=1.

Counter Register : The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the Branch Conditional to Count Register instruction.

3.2 Instruction Format

Instructions are 32 bits in length and contain a primary opcode in the first six bits. Instructions are described using 32-bit numbering. For most primary opcode values, bits 10-1 contain the secondary opcode. Instructions which encode large immediate values generally occupy a single primary opcode, and use bits 15-0 to provide a signed or unsigned immediate field. Most computational instructions are triadic (that is, they contain three register operands). One of the register operands specifies a destination or target (where the output of the computation is placed). The other register operands specify inputs to the computation. All instructions are four bytes long and word-aligned. Thus, whenever instruction addresses are presented to the processor the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Bits [31:26] always will specify the opcode in every instruction. There are quite a bit of instructions with the extended opcode (XO). Remaining bits vary based on the various instruction formats. Some of the instruction formats and field descriptions are described here:

OPCD : Primary opcode field.

LI : Immediate field used to specify a 24-bit signed twos complement integer which is concatenated on the right with 2'b00 and sign-extended to 64 bits.

AA : Absolute Address bit.

LK : LINK bit. If 1 then address of the instruction following the Branch instruction is placed into the Link Register.

BO : Field used to specify options for the Branch Conditional instructions.

BI : Field used to specify a bit in the CR to be tested by a Branch Conditional instruction.

Figure 3.2: I Instruction Format

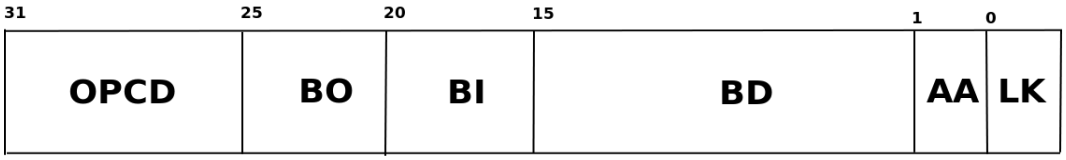


Figure 3.3: B Instruction Format

BD : Immediate field used to specify a 14-bit signed twos complement branch displacement which is concatenated on the right with 2'b00 and sign-extended to 64 bits.

BT : Field used to specify a bit in the CR.

BA : Field used to specify a bit in the CR to be used as a source.

BB : Field used to specify a bit in the CR to be used as a source.

XO : Extended opcode field.

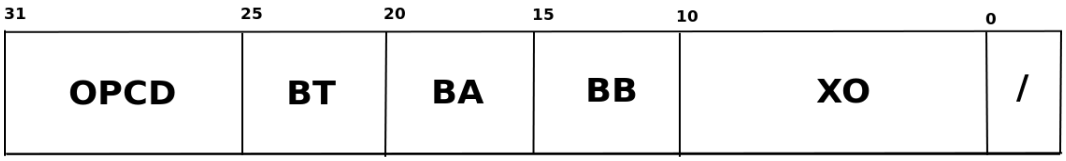


Figure 3.4: XL Instruction Format

RA : Field used to specify a GPR to be used as a source or as a target.

RB : Field used to specify a GPR to be used as a source.

RS : Field used to specify a GPR to be used as a source.

RT : Field used to specify a GPR to be used as a target.

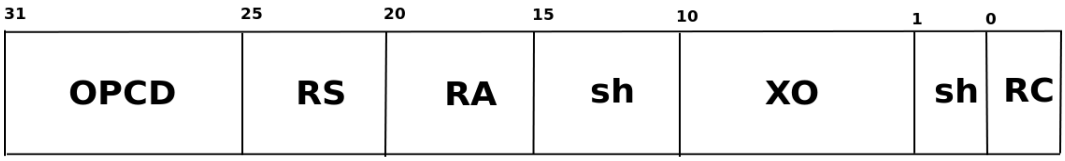


Figure 3.5: XS Instruction Format

sh : Field used to specify a shift amount.

RC : RECORD bit. If 1 then alter the condition register field.

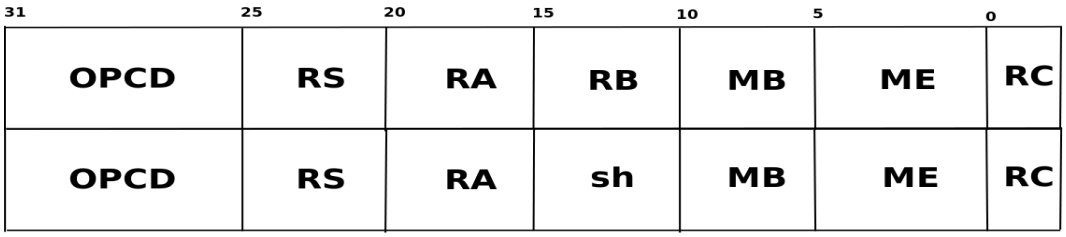


Figure 3.6: M Instruction Format

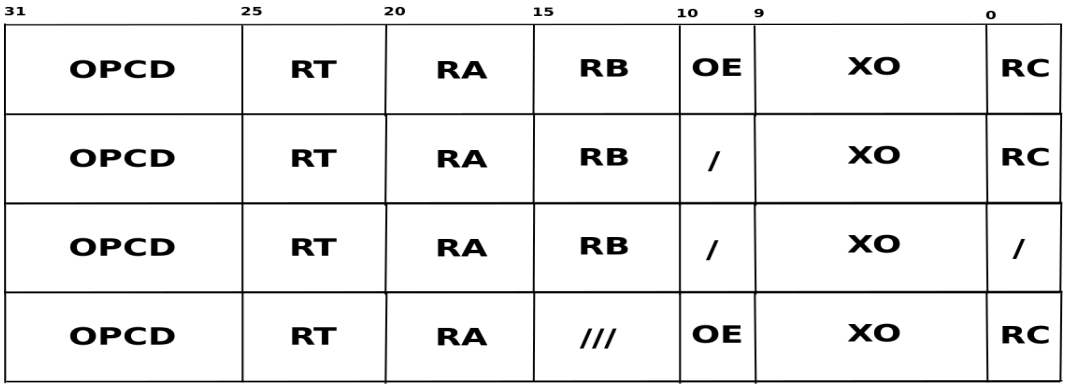


Figure 3.7: XO Instruction Format

3.3 Addressing Modes

The Power Instruction Set Architecture supports various addressing modes. The addressing modes supported for the calculation of effective address are Register Indirect with

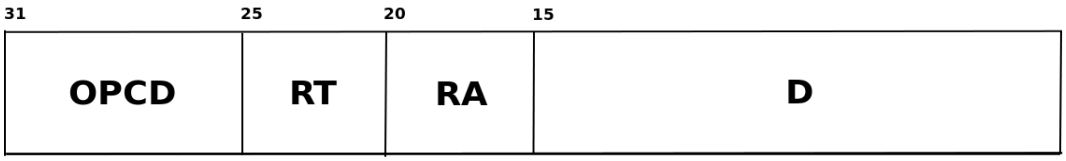


Figure 3.8: D Instruction Format

31	25	20	15	10	0
OPCD	RT	RA	RB	XO	RC
OPCD	RS	RA	RB	XO	RC
OPCD	RT	RA	RB	XO	/
OPCD	RS	RA	RB	XO	/
OPCD	RT	RA	/	XO	/
OPCD	RS	RA	/	XO	/

Figure 3.9: X Instruction Format

Immediate Index Addressing and Register Indirect with Index Addressing. Also there are various data and instruction addressing modes supported by it.

- Base+displacement addressing modeThe 16-bit D field is sign-extended and added to the contents of the GPR designated by rA or to zero if rA = 0. Instructions that use this addressing mode are of the D instruction format.
- Base+index addressing modeThe contents of the GPR designated by rB are added to the contents of the GPR designated by rA or to zero if rA = 0. Instructions that use this addressing mode are of the X instruction format.
- For I instruction format the 24-bit LI field is concatenated on the right with 2'b00, sign-extended, and added either to the address of the branch instruction if AA = 0, or to 0 if AA = 1.
- For B instruction format the 14-bit BD field is concatenated on the right with 2'b00, sign-extended, and added either to the address of the branch instruction if AA = 0, or to 0 if AA = 1.

CHAPTER 4

Implementation

This chapter describes the implementation of various execution units and decoder. We shall start with branch execution unit where we present its design consideration, interface and micro-architectural design decisions. The verification process and synthesis reports are also provided. Similarly this is done for condition register, arithmetic and logical, memory execution units and decoder unit. The different Functional Units make up a distributed Execution Unit, with all of the units sharing a common interface. Each unit contains its own set of Reservation Stations, subunits for performing its respective operation, and a result register. Each unit is responsible for having a set of rules that actually execute instructions. For simple functional units, this is just one rule that finds a ready instruction in the reservation stations, calculates the result, and stores it in the result register. A ready instruction is an instruction where all of the operands have values rather than tags that refer to results of other instructions. Each unit differs in how it implements the execution of an instruction.The decode rule is responsible for getting an instruction from the Fetch Unit and deciding which functional unit to issue the instruction to.

4.1 Branch Execution Unit

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word-aligned; processors ignore the two low-order bits of the generated branch target address.The inputs and outputs are both registered.The model of

program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the sequential execution model.The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching). All the operations take 1 clock cycle.The input PC to this module is obtained from the branch prediction unit which will tag the PC to be either Predicted taken or Predicted not taken, this also takes in the load register and counter register values and outputs them.The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7). For all fixed-point instructions in which RC=1, and for addic., andi., and andis., the first three bits of CR Field 0 (bits 3:1 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 0 of the Condition Register) is copied from the SO field of the XER. If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

- **Bit 0:** Summary overflow
- **Bit 1:** Zero
- **Bit 2:** Positive
- **Bit 3:** Negative

Here it should be noted that for the bluespec implementation the bit order naming is done as 31 : 0 from left to right when compared to that of the powerpc specification which is 0 : 31 . This way of naming is followed everywhere and this is the thing to be noted while giving data to the processor from the assembly code. Just reversing of each bit field is to be done , then the instructions work fine here.

The sequence of instruction execution can be changed by the Branch instructions. Because all instructions are on word boundaries, bits 1 and 0 of the generated branch target

address are ignored by the processor in performing the branch. Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the Branch instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken. The instructions that are implemented are b,ba,bl,bla,bc,bca,bcla,bcl,bclr,bclrl,bcctr,bcctrl.

4.1.1 Branch Module Interface

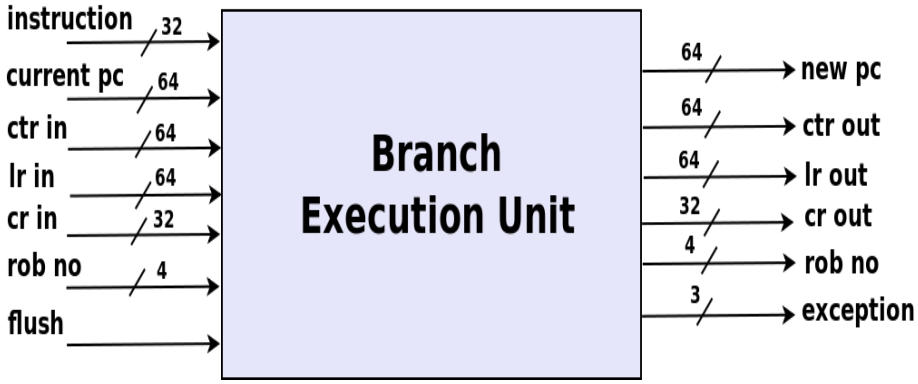


Figure 4.1: Branch Module Interface

The method _inputs will be invoked by the reservation station to supply the required inputs to the module. While there are several output methods for supplying the reservation station with the computed result.

The Branch instructions compute the effective address (EA) of the target in one of the following ways:

- Adding a displacement to the address of the Branch instruction (Branch or Branch Conditional with AA=0).
- Specifying an absolute address (Branch or Branch Conditional with AA=1).

Table 4.1: Description of branch module interface

Method	*Type*	*Description*
_inputs	input	This method is used to supply the module with required data.This involves 32 bit instruction, 64 bit program counter, 64 bit counter register value, 64 bit link register value, 32 bit condition register value and rob number.
_flush	input	Signal used to flush the pipeline. This abandons any current execution and brings all the registers to their default state.
new_pc_	output	This method gives the new pc which is the primary output of this module , contains the final branch address
ctr_out_	output	This method outputs the counter register value
lr_out_	output	This method outputs the link register value
cr_out_	output	This method outputs the condition register value
rob_number_	output	This method outputs the rob number assigned to this instruction
exception_	output	This provides all the exception that were generated during computation

- Using the address contained in the Link Register (Branch Conditional to Link Register).
- Using the address contained in the Count Register (Branch Conditional to Count Register).

4.1.2 Branch to absolute addressing and relative addressing

Instructions that use branch to absolute addressing mode and relative addressing mode to generate the next instruction address by sign extending and appending 2'b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this relative addressing mode have the absolute addressing option disabled (AA field, bit 1, in the instruction encoding = 0) while branches using absolute addressing mode have the absolute addressing option enabled (AA field, bit 1, in the instruction encoding = 1). The LR update option can be enabled (LK field, bit 31, in the

instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

4.1.3 Branch conditional to relative and absolute addressing

If branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending results to the immediate displacement operand (BD) and adding the resultant value to the current instruction address.

Different type of branching is accomplished by checking the value of BO field.

Table 4.2: Description of BO field

BO	*Description*
x0000	Decrement the CTR, then branch if decremented CTR !=0 and CR[BI] =0
x1000	Decrement the CTR, then branch if decremented CTR =0 and CR[BI] =0
xx100	Branch if CR[BI] =1
x0010	Decrement the CTR, then branch if decremented CTR !=0 and CR[BI] =1
x1010	Decrement the CTR, then branch if decremented CTR =0 and CR[BI] =1
xx110	Branch if CR[BI] =1
x00x1	Decrement the CTR, then branch if decremented CTR !=0
x10x1	Decrement the CTR, then branch if decremented CTR =0
xx1x1	Branch Always

Branches using this relative addressing mode have the absolute addressing option disabled (AA field, bit 1, in the instruction encoding = 0) while branches using absolute addressing mode have the absolute addressing option enabled (AA field, bit 1, in the in-

struction encoding = 1). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

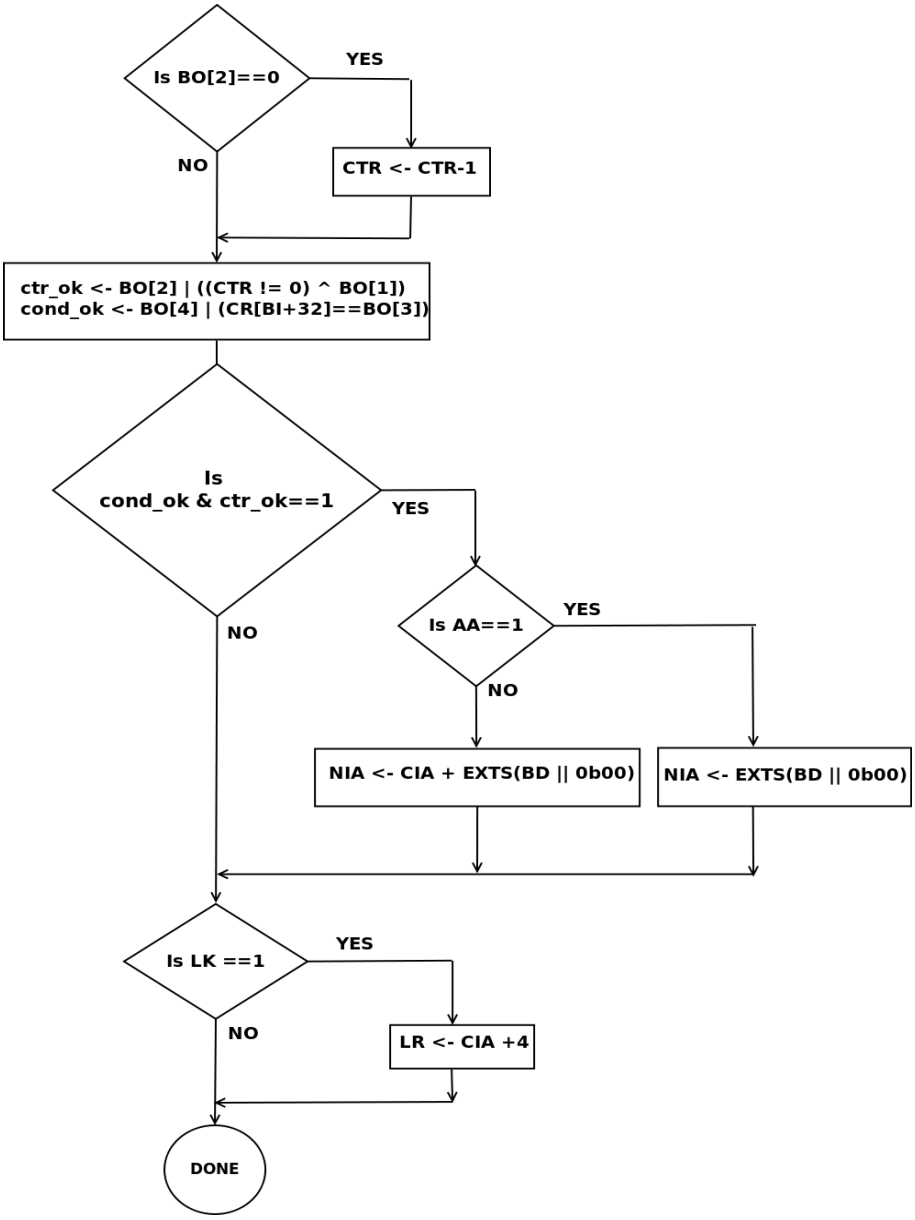


Figure 4.2: Branch Conditional

In order to compute the overall condition the cond_ok and ctr_ok are computed. The NIA is the next instruction address and CIA is the current instruction address. The LR is

the link register which is updated based on LK bit.

4.1.4 Branch Conditional to Link Register

If the branch conditions are met, the branch conditional to LR instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 0, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

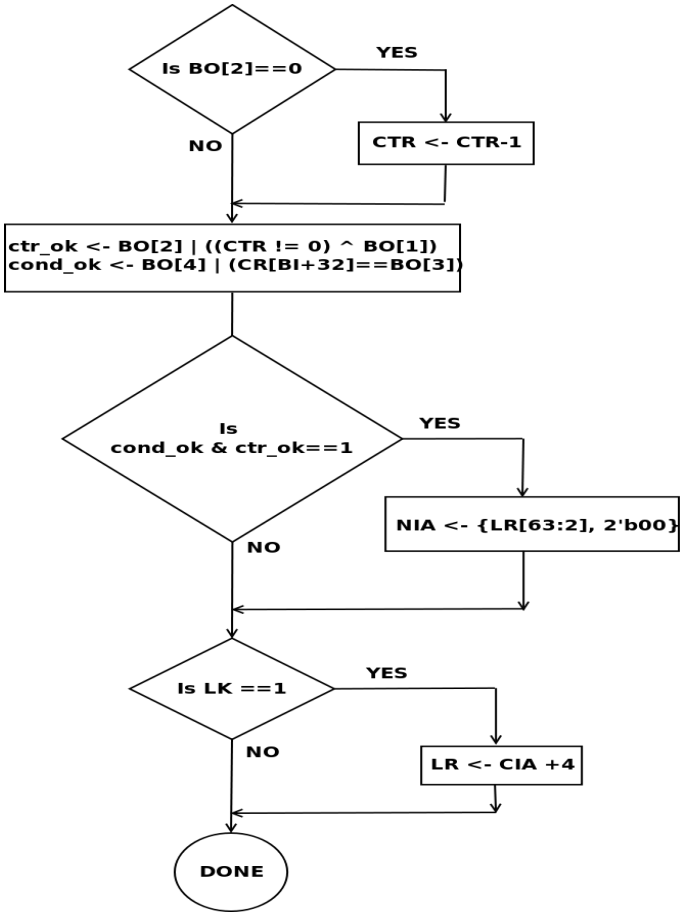


Figure 4.3: Branch Conditional to Link Register

In order to compute the overall condition the cond_ok and ctr_ok are computed. The

NIA is the next instruction address and CIA is the current instruction address. The LR is the link register which is updated based on LK bit.

4.1.5 Branch Conditional to Count Register

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 0, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

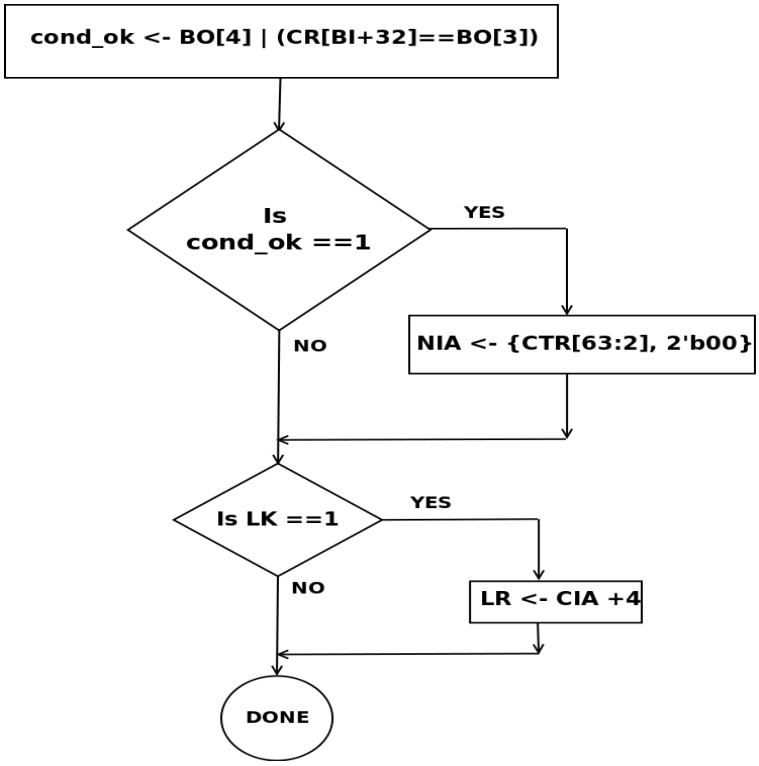


Figure 4.4: Branch Conditional to Link Register

In order to compute the overall condition the cond_ok is computed. The NIA is the next instruction address and CIA is the current instruction address. The LR is the link register

which is updated based on LK bit.

This bluespec module interaction can be described by the figure below.

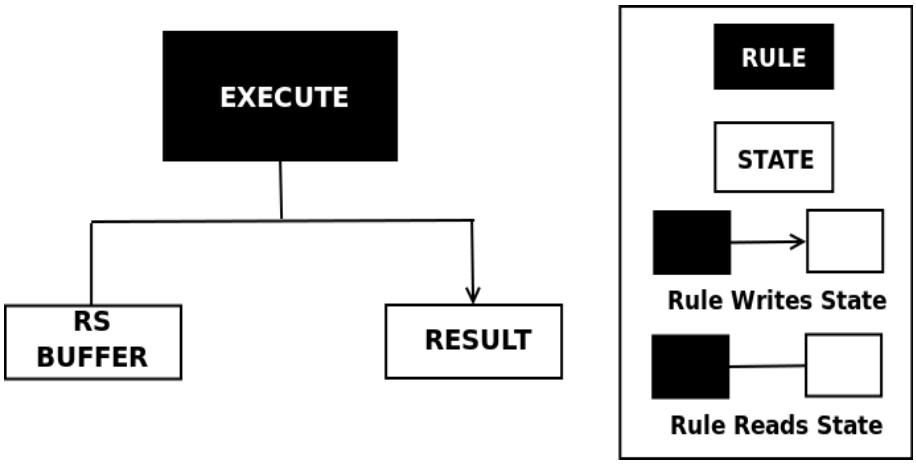


Figure 4.5: Bluespec module interaction

4.1.6 Verification

The verification setup involves development of testbench which will act as the Reservation Station for the execution units by providing the required inputs to the execution unit. The result obtained that is the next instruction address present in the program counter is checked to verify the functioning of the branch execution unit. It is needed to check if all kinds of branch is supported so the plan is to provide the required instruction (opcode) and also the different register values. Then the obtained results after execution is checked for correctness.

- Verified if the correct instruction is been selected for the given opcode. This verifies the correctness of the case structure designed. Then proceed to individual instructions.
- For Branch conditional type instruction seen if desired mode of condition is selected based on the BO field.
- Checked if sign-extend function is working as intended.

- Checked if proper addressing modes selected based on AA bit.
- Checked if Link register gets updated only if LK bit is one.

4.1.7 Synthesis Report

To Synthesis the Bluespec System Verilog code it is compiled using "compiled to verilog" option upon which the corresponding verilog code is generated. This code along with library verilog codes which includes the following files, found in "BlueSpecHome/lib/verilog", was given as input to "Xilinx ISE" on Virtex 6 evaluation board (6vlx240tff1156-1):

Table 4.3: Device utilization summary

Attribute	*Statistics*
No. of Slice Registers	487
No. of Slice LUT's	870
No. of 64 bit adders	4
No. of multiplexers	203

Table 4.4: Timing summary

Attribute	*Statistics*
Minimum period	4.762ns (Maximum Frequency: 209.974MHz)
Minimum input arrival time before clock	3.236ns
Maximum output required time after clock	1.342ns
Maximum combinational path delay	No path found

4.2 Arithmetic and Logical Execution Unit

The arithmetic unit is responsible for the execution of arithmetic instructions like add, subtract,rotate, shift etc. This execution unit is pipelined. The ALU unit is one of the most

important execution unit as most of the programs require the arithmetic and logical operations to be performed. The PowerPc architecture supports various kinds of arithmetic, logical , shift and rotate operations. The input to this module is obtained from the reservation station and the result produced is given back to the reservation station along with the destination address. As this is load/store model the arithmetic operations are performed on the GPR in case of integer arithmetic instructions. The registers that are altered by these instructions are the GPR's, exception register which has the summary overflow, overflow and the carry bits. There are move from and to special register instructions which is used to alter the special registers like XER,LR,CTR and CR. The method _inputs will be invoked by the reservation station to supply the required inputs to the module. While there are several output methods for supplying the reservation station with the computed results.

4.2.1 Arithmetic and Logical Module Interface

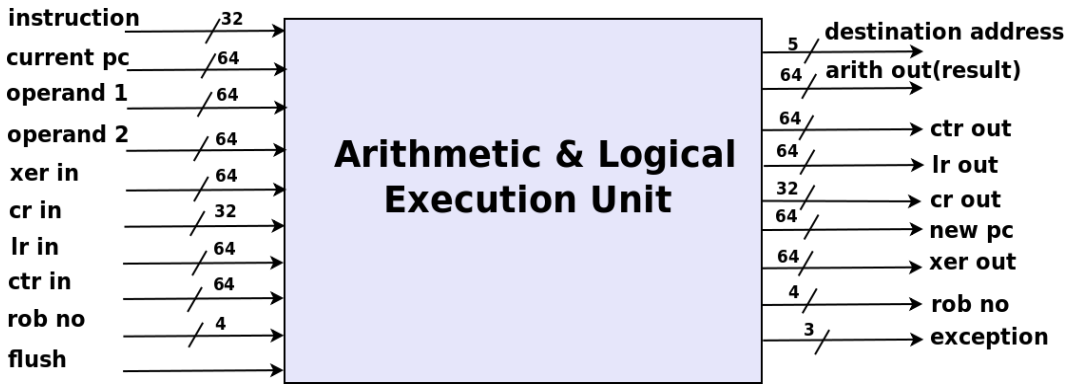


Figure 4.6: Arithmetic and Logical Module Interface

The description of the methods in the interface are as given in the table 4.5.

Table 4.5: Description of Arithmetic and Logical Module Interface

Method	*Type*	*Description*
_inputs	input	This method is used to supply the module with required data.This involves 32 bit instruction, 64 bit program counter, 64 bit counter register value, 64 bit operand1,64 bit operand2, 64 bit counter register value, 64 bit link register value, 32 bit condition register value 64 bit exception register and rob number.
_flush	input	Signal used to flush the pipeline. This abandons any current execution and brings all the registers to their default state.
arith_out_	output	This is the primary output method which provides the computed result. This method is invoked by the reservation station to obtain the result
destination_address_	output	This method gives the address of the GPR to which the result is to be put into.
new_pc_	output	This method gives the new pc which is the primary output of this module , contains the final branch address
ctr_out_	output	This method outputs the counter register value
lr_out_	output	This method outputs the link register value
cr_out_	output	This method outputs the condition register value
xer_out_	output	This method outputs the exception register value
rob_number_	output	This method outputs the rob number assigned to this instruction
exception_	output	This provides all the exception that were generated during computation

4.2.2 Micro-Architecture

The inputs are obtained using the inputs method then the opcode and extended opcode is checked in systematic manner to decide the instruction. This checking happens systematically as the width of the muxes implemented is determined by this. Then the instruction can be one of the following :

- Logical
- Arithmetic
- Rotate / Shift

- others

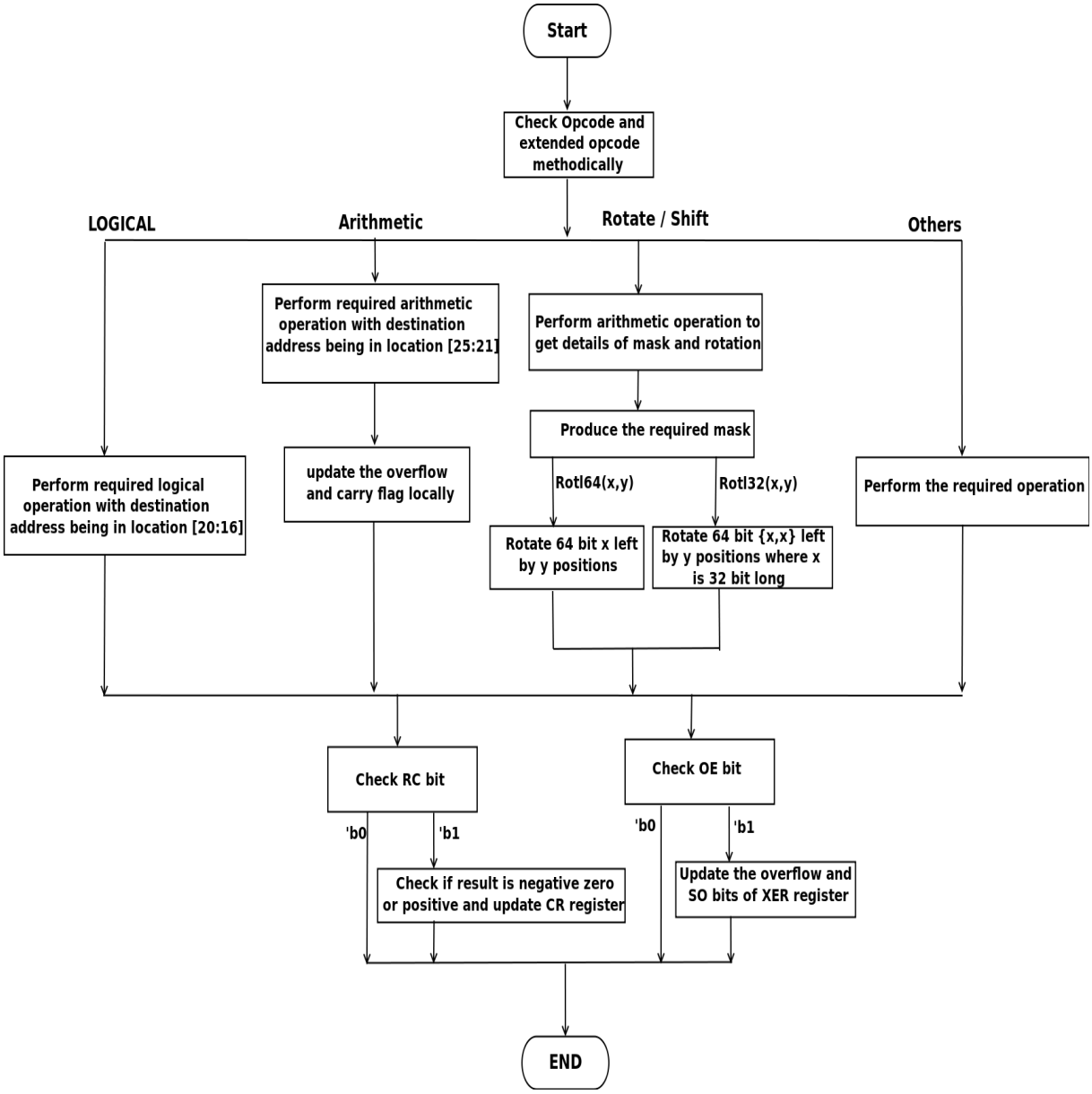


Figure 4.7: Arithmetic and Logical Module Architecture flow

Logical instructions : There are two major forms in which these instructions occur namely X-form with RC bit and with D form.

- The destination address of the result of these instructions are specified by the bits [20:16] of the 32 bit instruction.

- The immediate instructions are all unsigned and hence zero extended. The executed form of the no-op instruction is actually the xor immediate instruction.
- There are logical instructions which perform logical operation on contents from two GPR's.
- There are sign-extension instructions which are used to provide sign-extend byte, halfword or a word to form a 64 bit value.
- There are also parity word and double word instructions which check for odd parity.

Arithmetic instructions : There are two major forms in which these instructions occur namely XO-form with RC bit and with D form.

- The destination address of the result of these instructions are specified by the bits [25:21] of the 32 bit instruction.
- There are instruction which always set the carry flag to reflect the carry out of bit 63.
- These instructions set the SO and the OV bits when the OE bit in the instruction is set to 1. In case of the immediate operations the immediate data is considered as signed value and it is sign-extended to form the 64 bit value.
- The subtraction is done by the 2's complement method.
- There are add and subtract extended instructions which take into consideration the carry bit while performing addition or subtraction.
- There is also a negate instruction.

Rotate / Shift instructions :

- The destination address of the result of these instructions are specified by the bits [20:16] of the 32 bit instruction.
- These instructions performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.
- The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 63 enter at position 0.
- There are two types of rotate instructions available i.e., rotate 64 bits or rotate the 32 bits.

- The Rotate and Shift instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit through and including a stop bit, and 0-bits elsewhere.
- The values of start bit and stop bit range from 0 to 63. If start bit ¡ stop bit, the 1-bits wrap around from position 0 to position 63.
- There is no way to specify an all-zero mask.
- Rotate and Shift instructions do not change the OV and SO bits.
- Rotate and Shift instructions, except algebraic right shifts, do not change the CA bit.
- The Rotate Left instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64-n, where n is the number of bits by which to rotate right.
- Immediate-form logical shift operations are obtained by specifying appropriate masks and shift values for certain Rotate instructions.

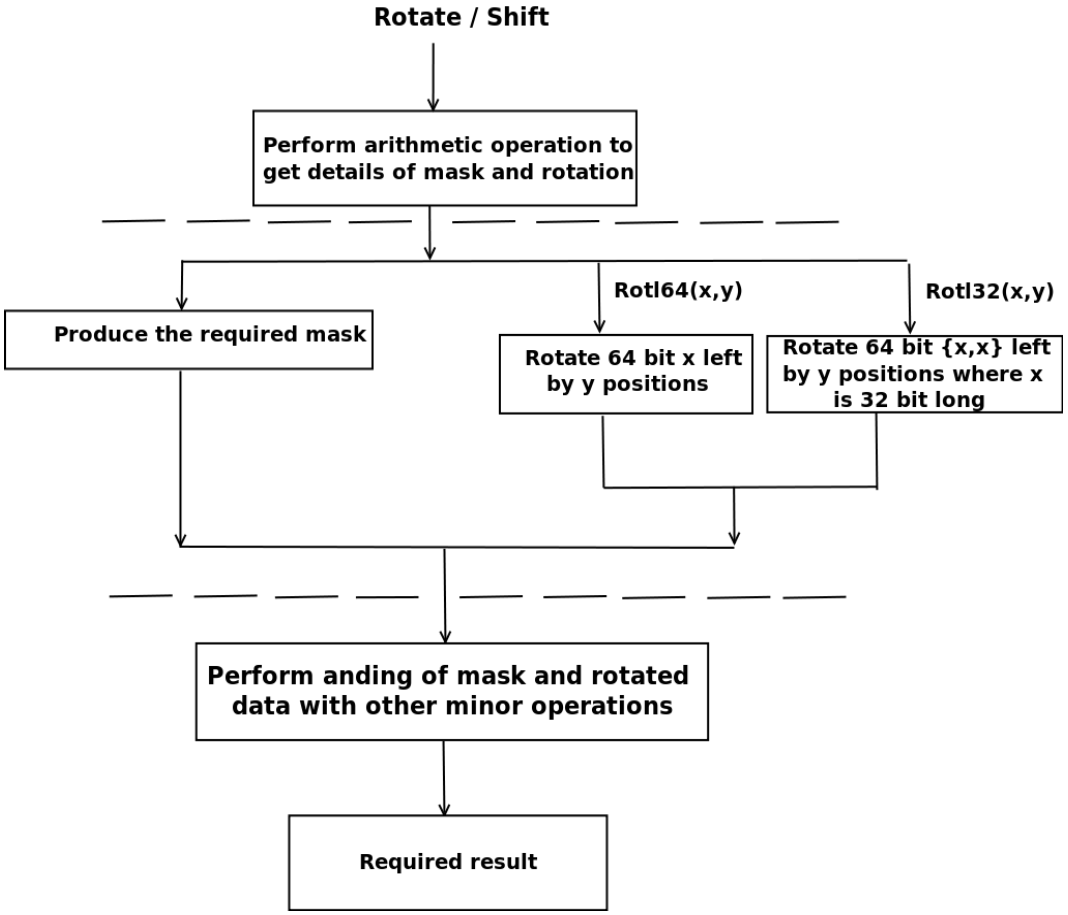


Figure 4.8: Pipelined rotate ans shift

Since a huge logic is required in executing these instructions it can be pipelined as below:

- The calculation of start and stop bits for mask and rotate operation.
- The mask generation along with rotate operation performed.
- Next anding of the mask and the rotated data is performed along with other minor operations based on instructions.

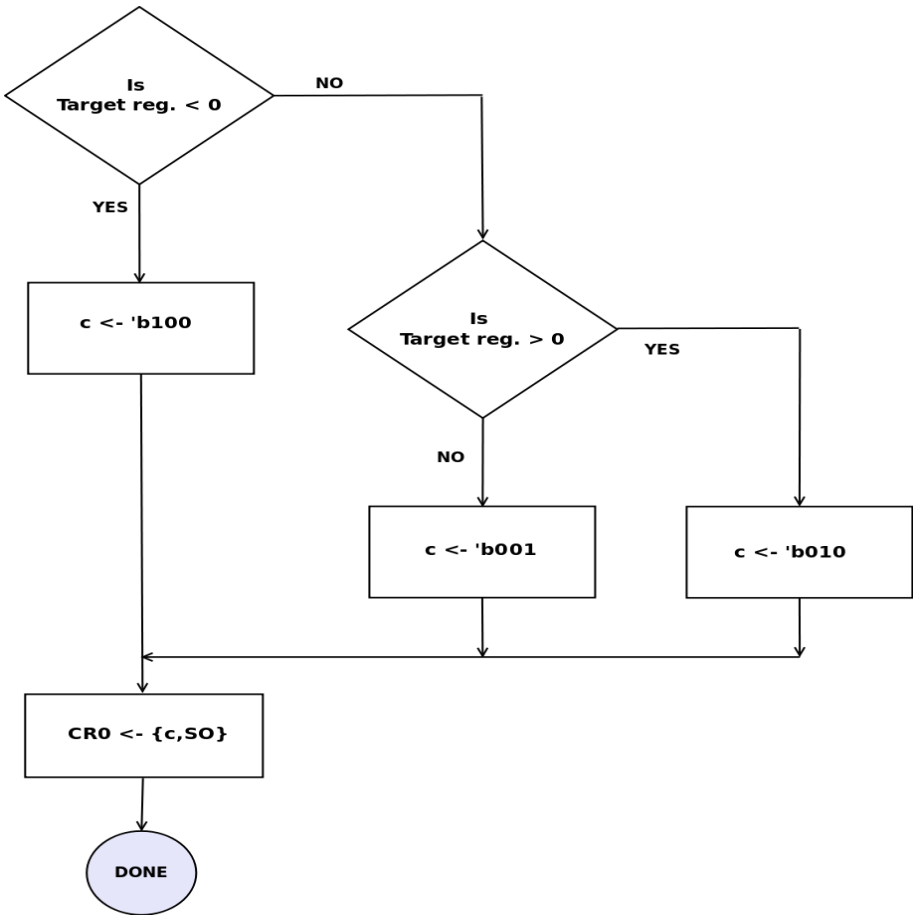


Figure 4.9: Value for CR0 field

Other instructions : These involve the move from and to special registers. The register to be altered is specified by the SPR field as indicated in table 4.6.

For all the above instructions when RC bit of the instruction is 1 then the result is used to modify the CR0 field of the condition register.

Table 4.6: SPR field and register

SPR	$Register$
0000000001	XER
0000001000	LR
0000001001	CTR

- **Bit 0:** Summary overflow
- **Bit 1:** Zero
- **Bit 2:** Positive
- **Bit 3:** Negative

And when the OE bit is 1 then the overflow and the summary overflow bits are set based on the result obtained. The overflow is computed by xoring the carry out of 63 and 62 bits.

This bluespec module interaction can be described by the figure below.

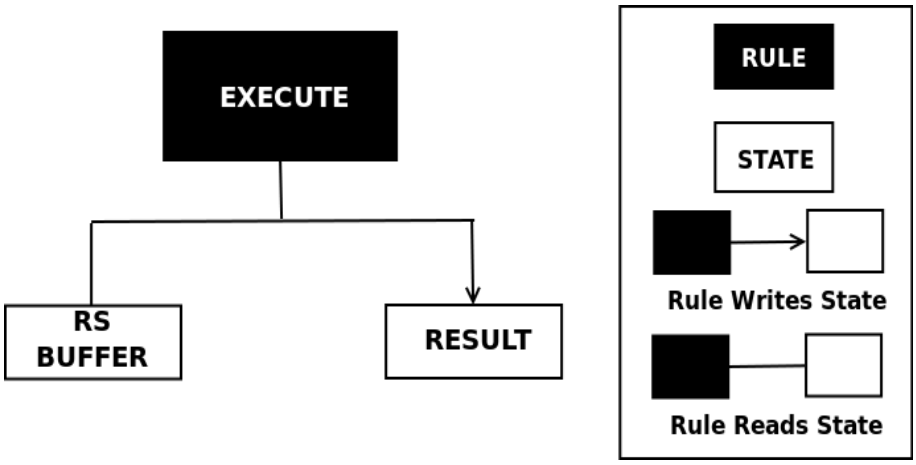


Figure 4.10: Bluespec module interaction

4.2.3 Verification

The verification setup involves development of testbench which will act as the Reservation Station for the execution units by providing the required inputs to the execution unit. Then the obtained results after execution is checked for correctness. Also one more method is followed where the functioning is described in high level language and the results are matched for correctness.

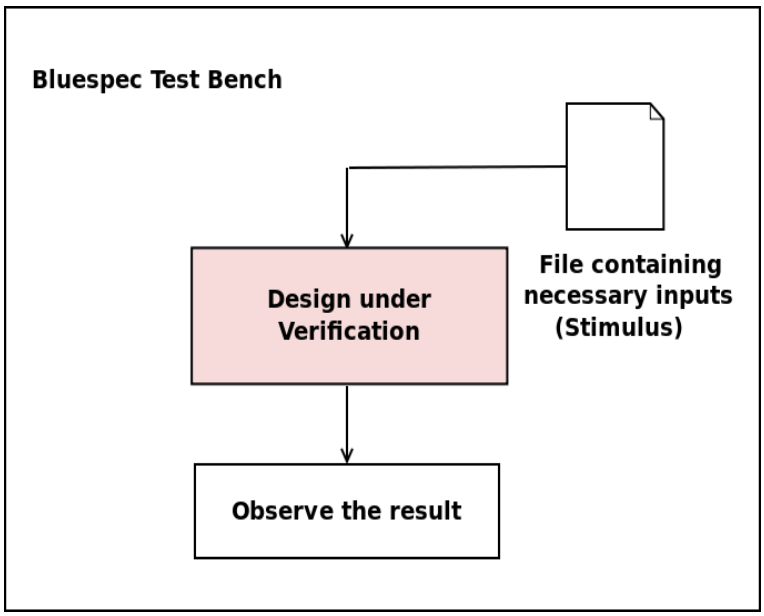


Figure 4.11: Verification setup

- Verified if the correct instruction is been selected for the given opcode. This verifies the correctness of the case structure designed. Then proceed to individual instructions.
- For all ALU instruction the result is passed through the compare with zero stage to update the CR0 field of Condition Register. Hence checking if this comparison produces accurate results.
- Checked if Overflow bit and CR0 field gets updated only if OE bit is 1 and RC bit is 1 respectively.
- For logical instruction checked their logical functioning.
- For arithmetic instructions checked if overflow and summary overflow is detected.

- For rotate and shift instructions checked if proper mask is produced, required shift occurs, by displaying results in intermediate stages. Also checked for extreme ends of shift in this case.
- For move to and from special register type of instructions checked if proper register is selected.

4.2.4 Synthesis Report

To Synthesis the Bluespec System Verilog code it is compiled using "compiled to verilog" option upon which the corresponding verilog code is generated. This code along with library verilog codes which includes the following files, found in "BlueSpecHome/lib/verilog", was given as input to "Xilinx ISE" on Virtex 6 evaluation board (6vlx240tff1156-1):

Table 4.7: Device utilization summary

Attribute	*Statistics*
No. of Slice Registers	161
No. of Slice LUT's	3723
No. of 64 bit , 8 bit ,6 bit adders	11,3,2
No. of 1 bit ,6 bit comparators	11,2

Table 4.8: Timing summary

Attribute	*Statistics*
Minimum period	1.707ns (Maximum Frequency: 585.981MHz)
Minimum input arrival time before clock	7.661ns
Maximum output required time after clock	1.453ns
Maximum combinational path delay	1.023ns

4.3 Condition Register Execution Unit

This execution unit executes the condition register logical instructions along with the compare instructions associated with the condition register. The significant point about this unit is that it will alter only the condition register value. PowerPC uses four condition codes: less than, greater than, equal, and summary overflow, but it has eight copies of them. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. PowerPC provides logical operations among these eight 4-bit condition code registers which allows more complex conditions to be tested by a single branch.

The compare instruction compares the contents of register RA(operand1) with either of the following:

- Sign extended value of the SI field in D form instructions
- Zero extended value of the UI field in D form instructions
- Contents of register RB (operand 2) in X form instructions

The comparisons are of two types:

- Signed Comparison
- Unsigned Comparison

The Compare instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two bits to 0. The summary overflow bit from the exception register is placed in the LSB location of the CR field.

- **Bit 0:** Summary overflow from XER.
- **Bit 1:** This bit is set to 1 if the contents of RA is equal to Sign extended SI, Zero extended UI or contents of RB.

- **Bit 2:** This bit is set to 1 if contents of RA is greater than Sign extended SI or contents of RB in case of signed comparison and contents of RA is greater than Zero extended UI or contents of RB in case of unsigned comparison.
- **Bit 3:** This bit is set to 1 if contents of RA is less than Sign extended SI or contents of RB in case of signed comparison and contents of RA is less than Zero extended UI or contents of RB in case of unsigned comparison.

The instructions implemented in this module are cmpi, cmp, cmpli, cmpl, crand, crnand, cror, crnor, crxor, creqv, crandc, crorc.

4.3.1 CR Module Interface

The method _inputs will be invoked by the reservation station to supply the required inputs to the module. While there are several output methods for supplying the reservation station with the computed result. The description of the methods are as given in the table.

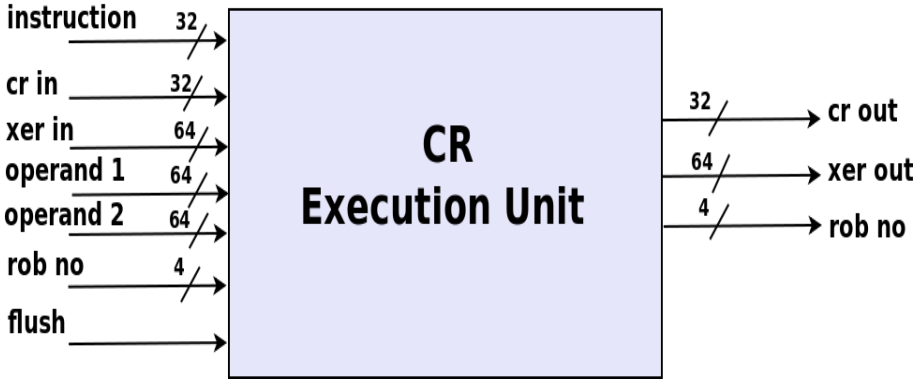


Figure 4.12: CR Module Interface

4.3.2 Micro-Architecture

In order to implement this module the micro-architecture followed is described here. First once the inputs are obtained from the reservation station the opcode and extended opcode

Table 4.9: Description of CR module interface

Method	*Type*	*Description*
_inputs	input	This method is used to supply the module with required data.This involves 32 bit instruction, 64 bit operand 1 (RA), 64 bit operand 2 (RB), 64 bit exception register value, 32 bit condition register value and rob number.
_flush	input	Signal used to flush the pipeline. This abandons any current execution and brings all the registers to their default state.
xer_out_	output	This method outputs the exception register value
cr_out_	output	This method outputs the condition register value
rob_number_	output	This method outputs the rob number assigned to this instruction

is checked to decide the kind of instruction i.e., either compare or the condition logical type.

In case of the condition logical type based on the extended opcode the different kinds of logical operation is performed with the bits from the specific fields (BA and BB fields of the instruction) of the condition register and the computed result is then placed in the destination bit of the condition register (BT field of the instruction).

In case of the compare instruction one of the value for comparison is either the 64 bit contents of RA register or the sign-extended lower word of the RA register. This choice is done based on the L bit(Bit 21) of the instruction, L=0 means choose the sign-extended value else the actual 64 bit value of the RA register. This obtained value is compared with either sign extended value of the SI, zero extended value of the UI or contents of RB register based on the opcode. Once both the values are obtained the comparison is performed and the 4 bit result is obtained based on the previously described method. Now the 4 bit value is put into the specific field of the condition register based on the BF bits of the instruction. This is accomplished by implementing a 1:8 De-Mux with the BF bits as

select lines. The remaining bits of the condition register is left unaltered.

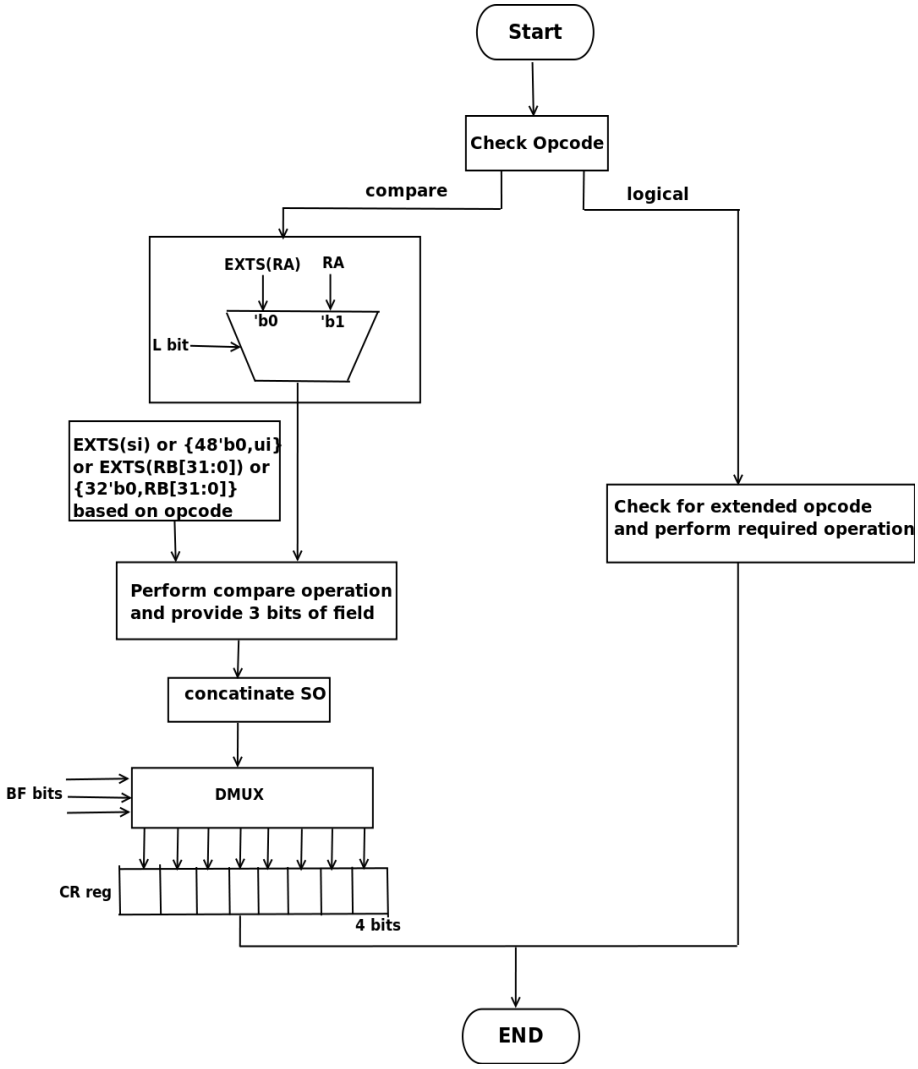


Figure 4.13: CR Module micro-architecture

4.3.3 Verification

The verification setup involves development of testbench which will act as the Reservation Station for the execution units by providing the required inputs to the execution unit. The result obtained that is the updated value of the condition register. Then the obtained results after execution is checked for correctness.

- Verify if the correct instruction is been selected for the given opcode. This verifies the correctness of the case structure designed. Then proceed to individual instructions.
- Check the working of signed and unsigned comparator designed.
- Check if sign-extend function is working as intended.
- Verify working of de-mux stage by giving various BF bit values.

This bluespec module interaction can be described by the figure below.

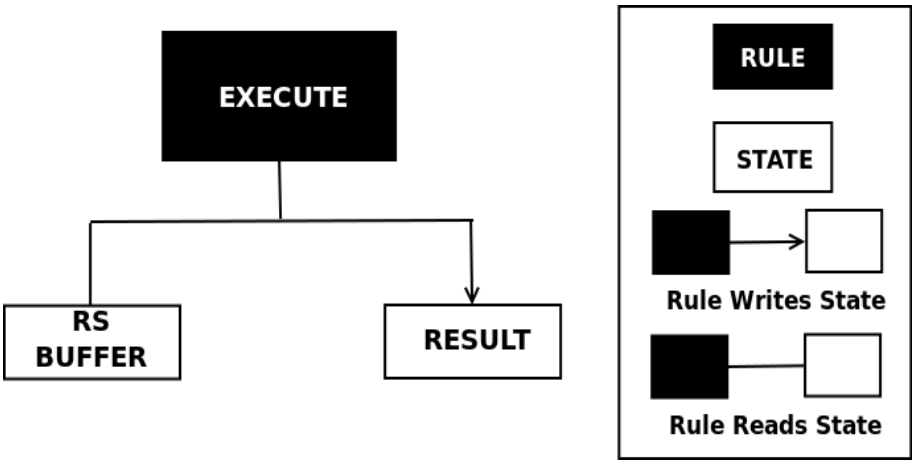


Figure 4.14: Bluespec module interaction

4.3.4 Synthesis Report

To Synthesis the Bluespec System Verilog code it is compiled using ”compiled to verilog” option upon which the corresponding verilog code is generated. This code along with library verilog codes which includes the following files, found in ”BlueSpecHome/lib/verilog”, was given as input to ”Xilinx ISE” on Virtex 6 evaluation board (6vlx240tff1156-1):

Table 4.10: Device utilization summary

Attribute	*Statistics*
No. of Slice Registers	32
No. of Slice LUT's	662
No. of 64 bit comparators	4
No. of multiplexers	20

Table 4.11: Timing summary

Attribute	*Statistics*
Minimum period	0.877ns (Maximum Frequency: 1140.251MHz)
Minimum input arrival time before clock	5.756ns
Maximum output required time after clock	0.783ns
Maximum combinational path delay	0.984ns

4.4 Memory Execution Unit

Memory instructions are responsible for moving the data between the main memory and the register file, and hence are essential for execution of ALU instructions. Register operands needed for ALU instructions must be first loaded from the memory. Not all the operands can be kept in the register file due to limited number of registers. Hence there is the need of the load/store instructions. For the working of this there is a need to calculate the memory address and also need to access the location. The integer load and store instructions can be classified into following types:

- Integer Load instructions.
- Integer Store instructions.
- Integer load and store with byte-reverse instructions.
- Integer load and store multiple instructions.

4.4.1 Memory Module Interface

The method _inputs will be invoked by the reservation station to supply the required inputs to the module. While there are several output methods for supplying the reservation station and memory management unit with the computed result.

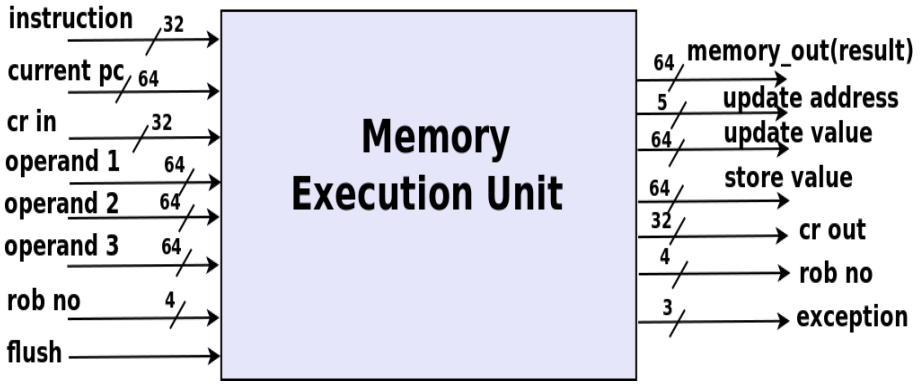


Figure 4.15: Memory Module Interface

The description of the methods are as given in the table 4.12.

4.4.2 Micro-Architecture

The memory unit is responsible for the execution of the memory related operations like Load and Store. This is the only unit which accesses the data cache. Finished store buffer is used in implementing this module. The purpose of the using finished store buffer (FSB) is to store the address and data of the recent store operations [6]. The size of the FSB chosen is 8 and hence it can store data of up to 8 past store operations. When the 9th store operation enters it is overwritten on the first store operation data. Thus each a time a Store operation occurs, it updates the FSB with the address and the data to be stored in the cache. The data cache is only written when the store operation reaches the head of the Re-order buffer and is ready to commit. At this time, the Re-order buffer will send

Table 4.12: Description of Memory module interface

Method	*Type*	*Description*
_inputs	input	This method is used to supply the module with required data.This involves 32 bit instruction, 64 bit operand 1 (RA), 64 bit operand 2 (RB),64 bit operand 3 (RT), 64 bit current program counter , 32 bit condition register value and rob number.
_flush	input	Signal used to flush the pipeline. This abandons any current execution and brings all the registers to their default state.
memory_out_	output	This method outputs the type of instruction either load byte, store byte, load halfword etc.. with the effective address.
update_address_	output	This method gives the register address for update forms of instructions.
update_value_	output	This method gives the update value for update forms of instructions.
store_value_	output	This method provides with the value to be stored in the memory location in case of the store instructions.
cr_out_	output	This method outputs the condition register value
rob_number_	output	This method outputs the rob number assigned to this instruction
exception_	output	This provides all the exception that were generated during computation

the store operation address and data to the memory unit. The memory unit will only now initiate the write operation in the data cache.

When a load instruction enter the memory unit, the FSB is first checked to check is it contains the address from which data is to be loaded. If the FSB, has an entry corresponding to the address then the data is read from the FSB and the data cache is bypassed by tagging the address as the arithmetic operation. If the FSB has no information regarding the load address then the data cache read cycle is initiated. The memory unit sends the address to the cache controller on the data cache port. The memory unit will now wait until the data cache has returned with value. During this time the memory unit is busy and cannot accept any other input or carry out any other operation. Once the data is received

the value is then sent to the Re-order buffer to commit.

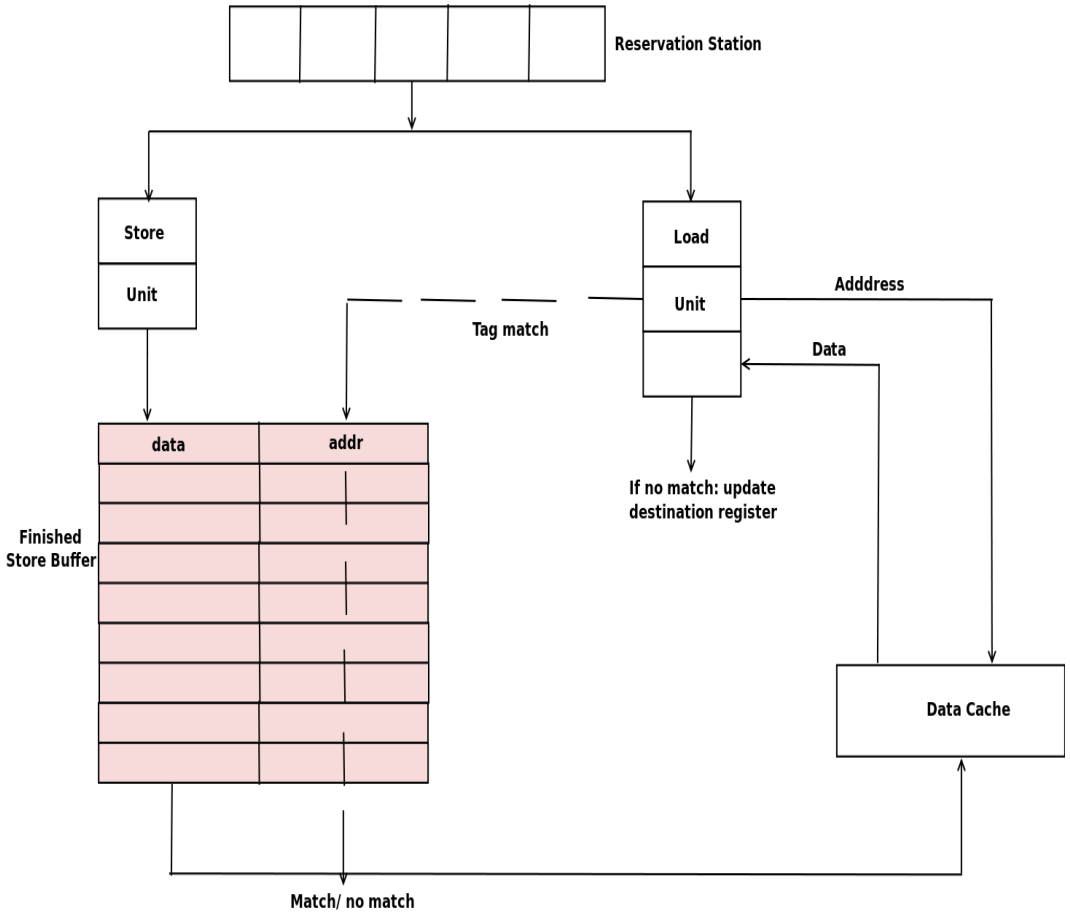


Figure 4.16: Working with use of FSB

The effective address generation is done in two ways namely,

- Register Indirect with Immediate Index Addressing.
- Register Indirect with Index Addressing.

Register Indirect with Immediate Index Addressing : In this mode the 16-bit immediate index is sign extended and this is added to the ra operand which is the contents of the general purpose register (GPR) specified in the instruction. When the specified GPR is R0 then 0 is added to the sign extended value.

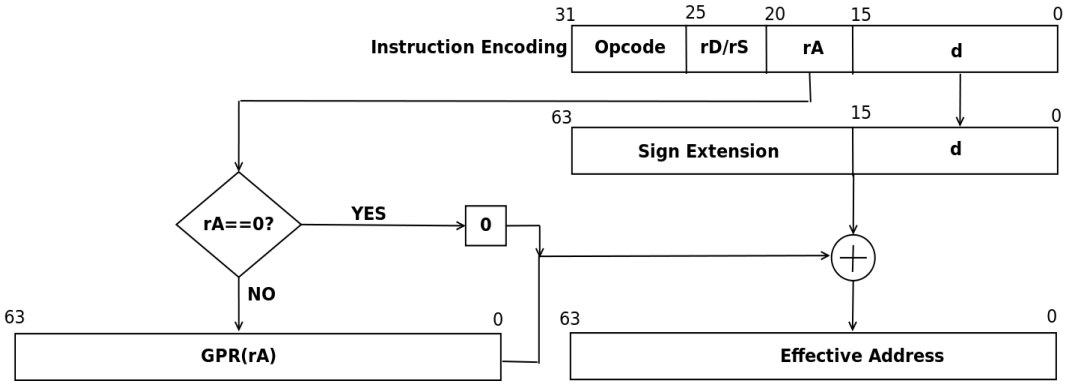


Figure 4.17: Register Indirect with Immediate Index Addressing

Register Indirect with Index Addressing : In this mode the rb operand is added to the ra operand which is the contents of the general purpose register (GPR) specified in the instruction. When the specified GPR is R0 its value is chosen as 0.

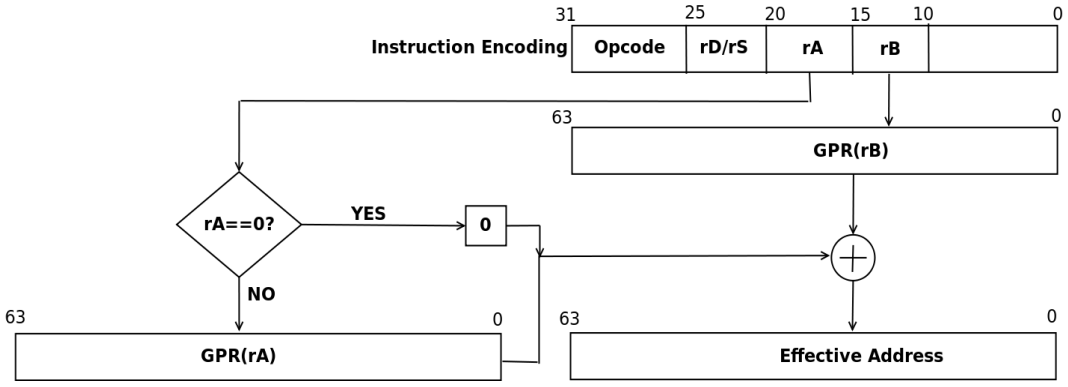


Figure 4.18: Register Indirect with Index Addressing

When the inputs are given to this module first the opcode and extended opcode is checked to determine the instruction. This is achieved with the help of case statements. Next the computation of the effective address is done. The effective address is obtained either by Register Indirect with Index Addressing or by Register Indirect with Immediate Index Addressing . Now for the load operation first the address is checked in the FSB to obtain a match, if matched then it is tagged arithmetic and the GPR will be loaded with the required value. In case of no match then the data is obtained by accessing the data cache.

For the store instruction the store is performed and in case of multiple store then the FSB is cleared.

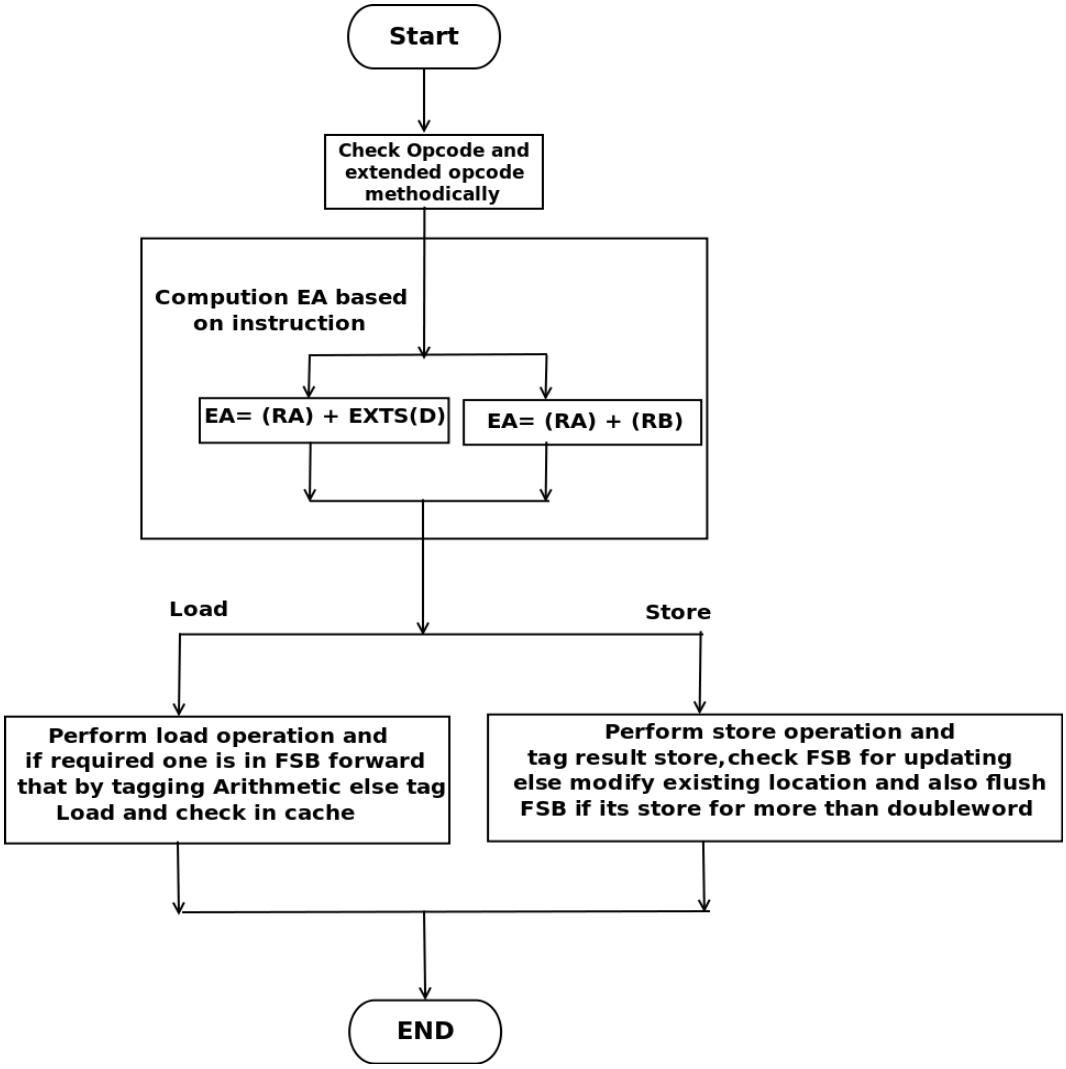


Figure 4.19: Working of memory execution module

Integer Load Instructions : The instructions coming under this category are Load Byte , Load Halfword, Load Word, Load Double word , where the zero extended load is performed in Register Indirect with Immediate Index Addressing mode and the resulting value is stored in specified GPR. Load Halfword Algebraic and Load Word Algebraic , where sign extended load is performed Register Indirect with Immediate Index Addressing

mode and the resulting value is stored in specified GPR. Load Byte Indexed , Load Halfword Indexed, Load Halfword Algebraic Indexed, Load Word Indexed ,Load Word Algebraic Indexed ,Load Doubleword Indexed are the instructions where load is performed in Register Indirect with Index Addressing mode. Load Byte with update , Load Byte Indexed with update, Load Halfword with update, Load Halfword Algebraic with update, Load Halfword Indexed with update, Load Halfword Indexed Algebraic with update, Load Word with update, Load Word Algebraic with update, Load Word Indexed Algebraic with update, Load Doubleword with update, Load Doubleword Algebraic with update, Load Doubleword Indexed with update, Load Doubleword Indexed Algebraic with update are the instructions where one of the specified GPR is updated with the computed effective address.

Integer Store Instructions : The instructions coming under this category are Store Byte , Store Halfword, Store Word, Store Double word , where the zero extended store is performed in Register Indirect with Immediate Index Addressing mode.Store Byte Indexed , Store Halfword Indexed, Store Word Indexed, Store Doubleword Indexed are the instructions where store is performed in Register Indirect with Index Addressing mode. Store Byte with update , Store Byte Indexed with update, Store Halfword with update, Store Halfword Indexed with update, Store Word with update, Store Doubleword with update, Store Doubleword Indexed with update are the instructions where one of the specified GPR is updated with the computed effective address.

Integer load and store with byte-reverse instructions : The instructions coming under this category are Load Halfword Byte-Reverse Indexed, Store Halfword Byte-Reverse Indexed, Load Word Byte-Reverse Indexed, Store Word Byte-Reverse Indexed, Load Doubleword Byte-Reverse Indexed, Store Doubleword Byte-Reverse Indexed. These instruc-

tions have the effect of loading and storing data in the opposite byte ordering from that which would be used by other Load and Store instructions.

Integer load and store multiple instructions : The instructions coming under this category are Load Multiple Word and Store Multiple Word. The combination of the effective address and RT (RS) is such that the low-order byte of GPRs is loaded from or stored into the last byte of an aligned quadword in storage.

This bluespec module interaction can be described by the figure below.

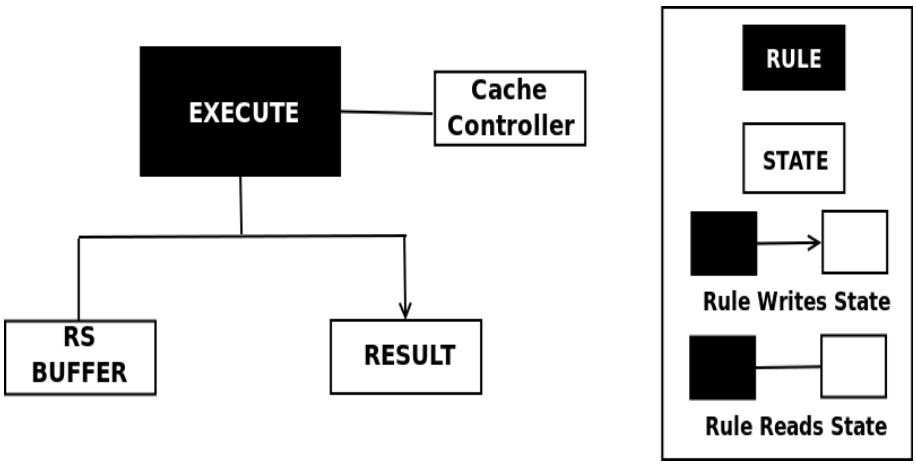


Figure 4.20: Bluespec module interaction

4.4.3 Verification

The verification setup involves development of testbench which will act as the Reservation Station for the execution units by providing the required inputs to the execution unit. Also a set up where the data is provided by the test bench given the address. Then the obtained results after execution is checked for correctness.

- Verified if the correct instruction is been selected for the given opcode. This verifies the correctness of the case structure designed. Then proceed to individual instructions.

- For memory instruction checked if logic for calculation of effective address is accurate.
- Checked if the result is tagged arithmetic if the desired data is present in FSB for load instructions.
- Checked if the cache controller is given the proper signals in case of miss in FSB.
- Checked that FSB is updated for store instruction and also the cache controller is given the signals like store, address and data.

4.4.4 Synthesis Report

To Synthesis the Bluespec System Verilog code it is compiled using ”compiled to verilog” option upon which the corresponding verilog code is generated. This code along with library verilog codes which includes the following files, found in ”BlueSpecHome/lib/verilog”, was given as input to ”Xilinx ISE” on Virtex 6 evaluation board (6vlx240tff1156-1):

Table 4.13: Device utilization summary

Attribute	*Statistics*
No. of Slice Registers	847
No. of Slice LUT's	1286
No. of 64 bit comparators	8
No. of multiplexers	70

Table 4.14: Timing summary

Attribute	*Statistics*
Minimum period	4.373ns (Maximum Frequency: 228.695MHz)
Minimum input arrival time before clock	3.309ns
Maximum output required time after clock	1.432ns
Maximum combinational path delay	1.082ns

4.5 Decoder Unit

Decoder unit is responsible for getting the instruction from the fetch unit and supply the necessary inputs to the issue stage. The decoder gets the instructions from the decode queue and then the decoder puts the data into the issue queue for it to access them. The 32 bit instruction is the input to this unit. Based on the opcode and the extended opcode the type of instruction is determined and the decoder bundle is formed.
The decoder bundle consists of the following information:

- Address of operand 1
- Address of operand 2
- Address of operand 3
- Destination address
- Type of instruction

The operand addresses are the location of the registers in the GPR unit, hence the width of this is 5 bits each. The destination address is also the destination GPR location and hence it is 5 bit. The type of instruction specifies the instruction to be of arithmetic, branch , memory, vector etc.

4.5.1 Micro-Architecture

The case statements are built in such a way that no huge muxes are present thus ensuring the less combinational delay of the module. This division requires finding relation between the different opcodes , extended opcodes and the instructions. Following method is used to built the decoder.

- First the two MSB bits are checked i.e, bits [31:30] of the instruction.

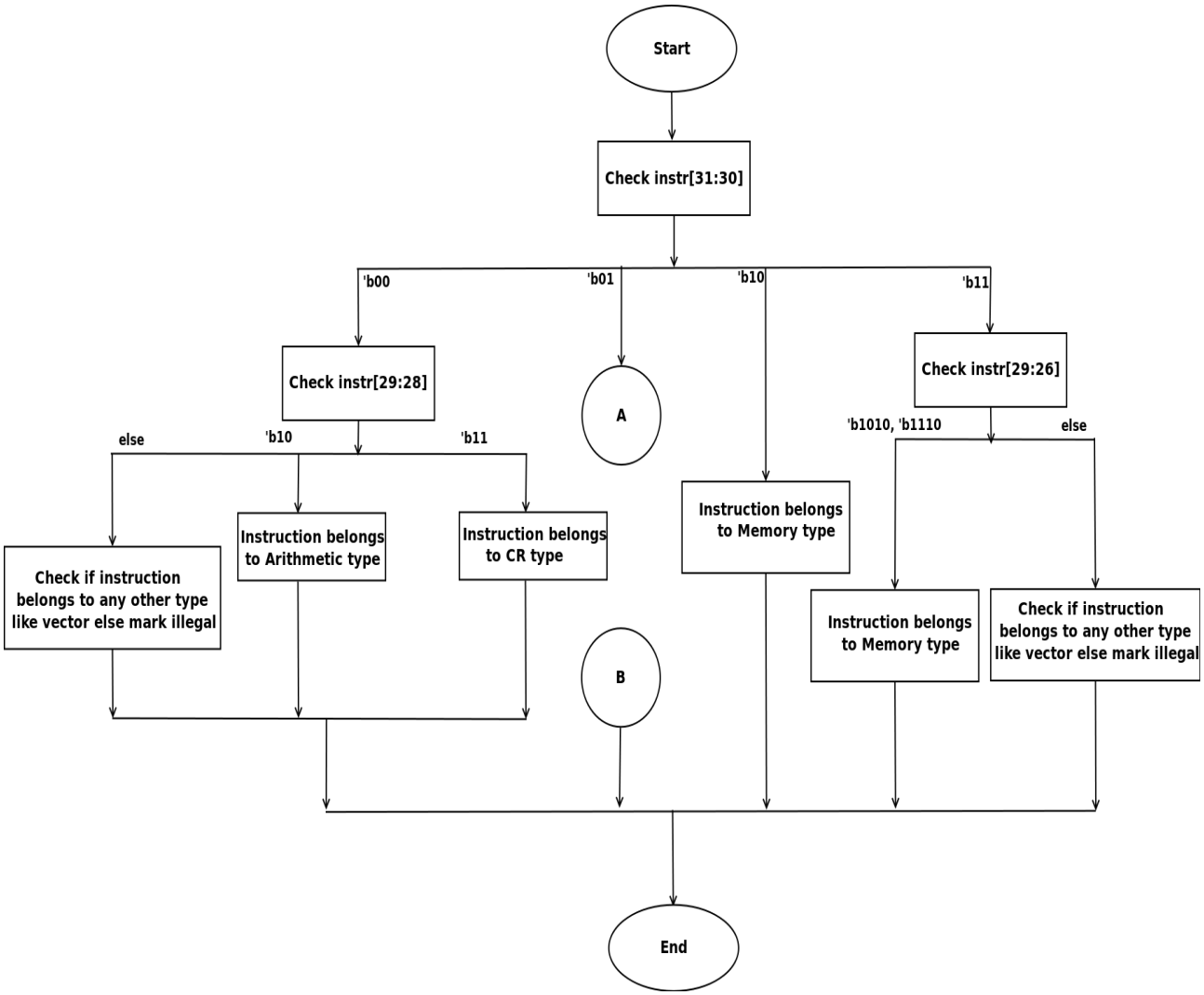


Figure 4.21: Working of decoder

- If these bits are 2'b10 then the instruction is of memory type and the corresponding bundle of data is made.
- If these bits are 2'b00 then check the bits [29:28] . If it turns out to be 2'b10 then its of arithmetic type and else if 2'b11 then its of CR type.
- If these bits are 2'b11 Then check bits [29:26] and if it is 4'b1010 or 4'b1110 then its of memory type.

- If the two MSB bits are 2'b01 then further study of opcode needs to be done.

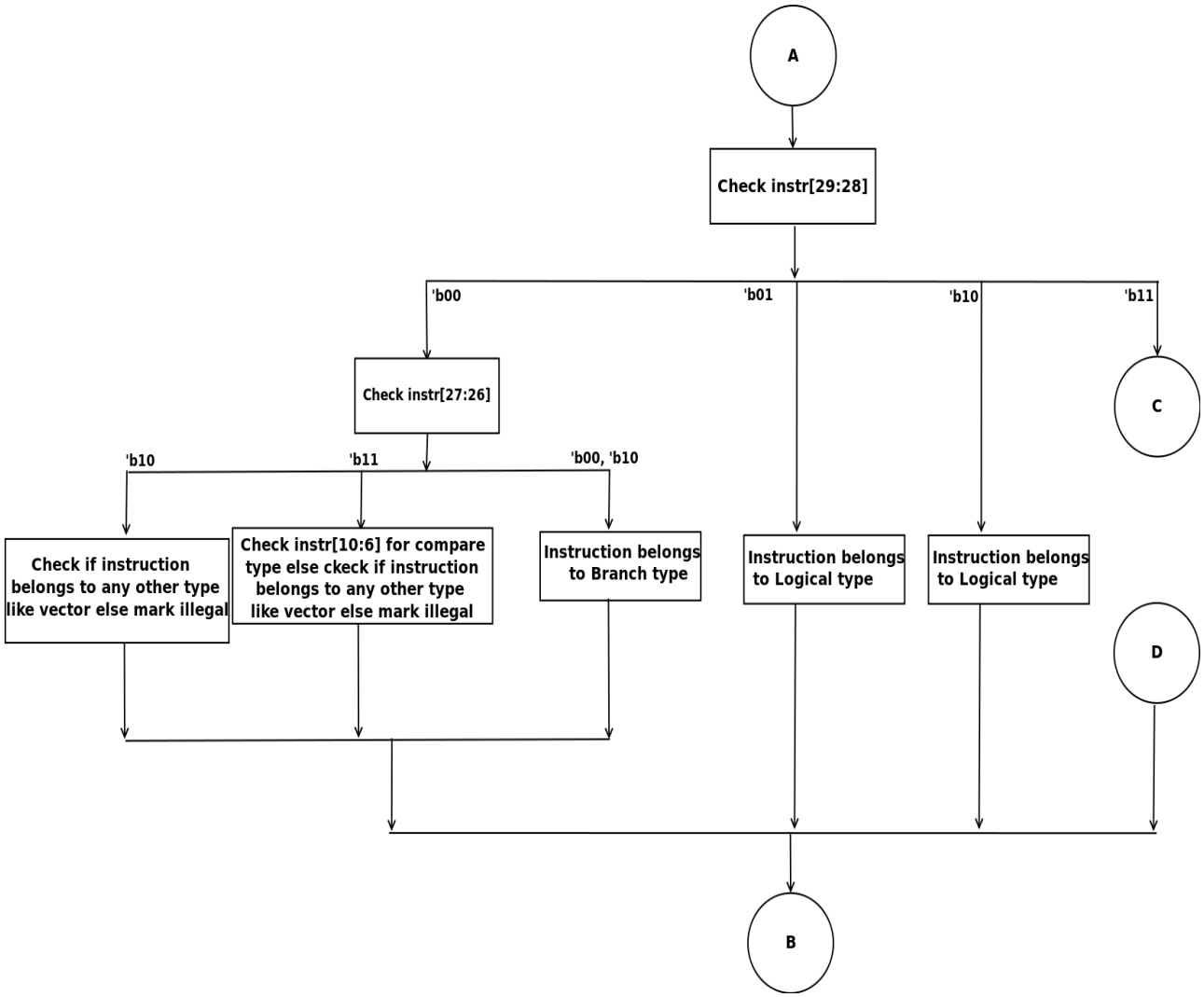


Figure 4.22: Working of decoder contd.

- Now if the bits [29:28] are either 2'b01 or 2'b10 then it is logical type of instruction.
- If bits [29:28] are 2'b00 then in case if bits [27:26] are 2'b00 or 2'b10 then its branch type else check for the extended opcode region bits [10:6] to check if it is arithmetic compare instructions.
- If these bits are 2'b11 then the extended opcode [5:1] bits to further determine the type of instruction.

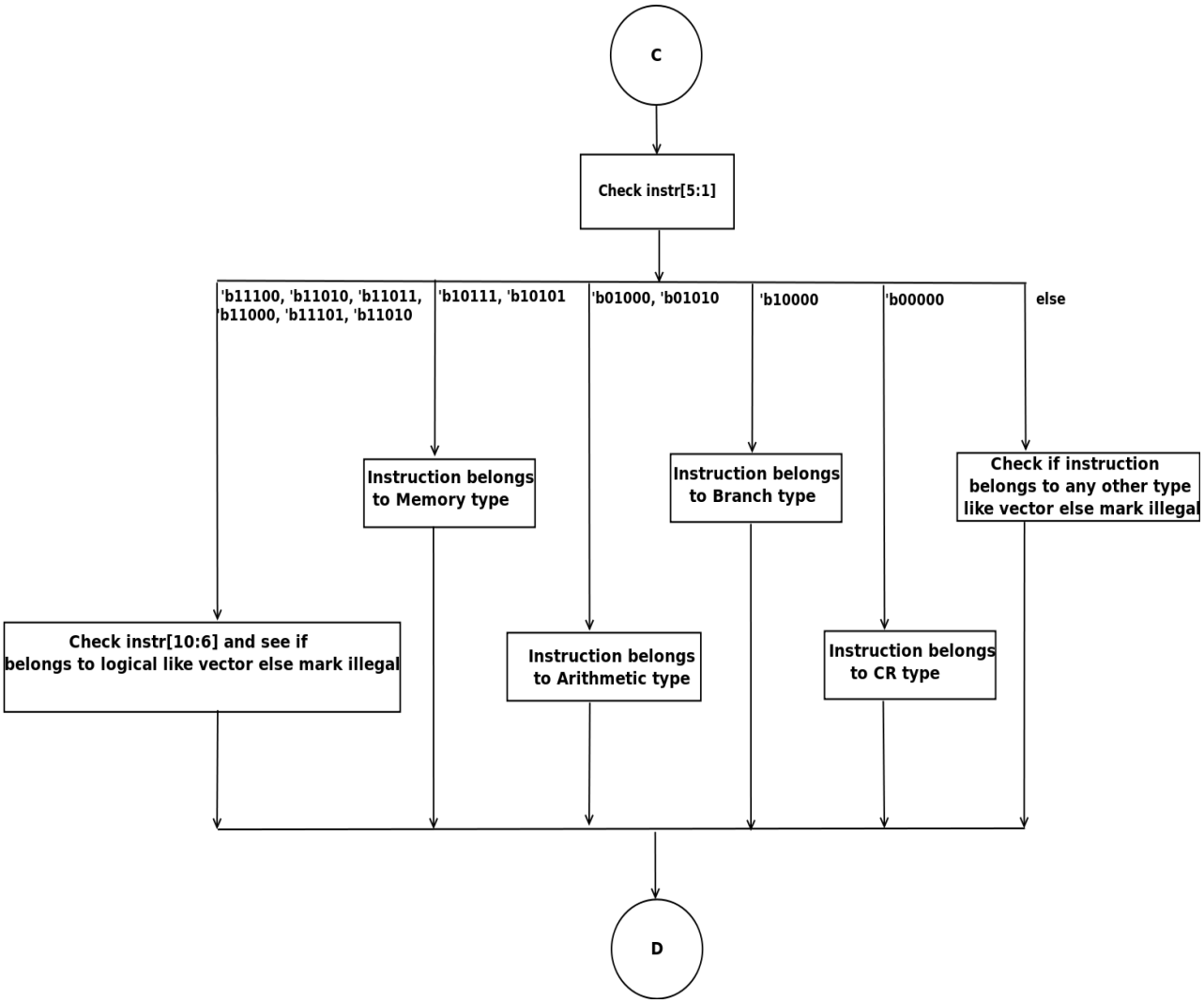


Figure 4.23: Working of decoder contd.

4.5.2 Verification

The verification setup involves development of testbench which will act as fetch unit supplying with instructions and then the result obtained is checked to see the functional correctness.

- Verified if the correct instruction is been selected for the given opcode. This verifies the correctness of the case structure designed. Then proceed to individual instructions.
- Verified each element in the bundle of data produced is correct.

This bluespec module interaction can be described by the figure 4.24.

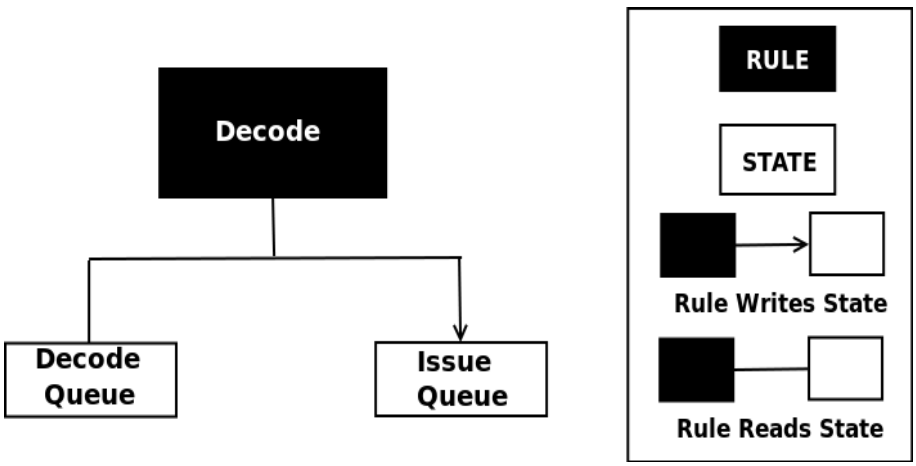


Figure 4.24: Bluespec module interaction

4.5.3 Synthesis Report

To Synthesis the Bluespec System Verilog code it is compiled using ”compiled to verilog” option upon which the corresponding verilog code is generated. This code along with library verilog codes which includes the following files, found in ”BlueSpecHome/lib/verilog”, was given as input to ”Xilinx ISE” on Virtex 6 evaluation board (6vlx240tff1156-1):

Table 4.15: Device utilization and Timing summary

Attribute	*Statistics*
No. of Slice LUT's	201
No. of multiplexers	88
Maximum combinational path delay	4.702ns

4.6 Design Challenges

During the implementation of the various execution units and decoder, there were several challenges to take care of.

- First and foremost the design of decoder and also the selection of various instructions within the execution units, required to understand the encoding of the opcodes and extended opcodes in depth and come up with strategy to split the opcode to check the type of instruction. The strategy followed was to check the 6 bit opcode, 2 bit at a time and checking the extended opcodes by splitting them into group of 5 bits to reduce width of muxes thus increasing the frequency of operation. This building up of mux can be done in other ways to see if there is any further improvement in operating frequency.
- During the implementation of execution unit for branch conditional instruction, the type of conditional branch is decided by the 5 bit BO field. The distinction of particular type was done based on the pattern found in encoding of BO fields. Then the conditional equations were formed. But this can also be done in different way.
- For the implementation of compare instructions (signed as well as unsigned) a compact structure needed for identifying the type of comparison and also then updating the exact CR field, was done using Mux-De mux combination which made the logic compact.
- In order to have a better performance of load instructions finished state buffers were used.
- Further to increase the frequency of operation a 3 stage pipelined implementation of rotate/shift operation is also implemented and this can be used based on necessity.

CHAPTER 5

Conclusion and Future Work

The Power ISA was chosen and various fixed-point scalar execution units were designed as per the PowerISA 2.06 requirement to provide support for these instructions. The decoder stage for the processor was also designed as part of the project. The aim of reduced hardware and good operating frequency was achieved with careful implementation of each modules, by efficient use of functions and using neat hardware techniques wherever necessary. All the execution units and decoder unit were verified individually for their functional correctness.

In future all the execution stages and other stages of the pipelined architecture can be integrated and verified for the overall functioning of the CPU core. Also slight modifications of certain structures described in each modules can be done to further improve its performance.

APPENDIX A

PowerISA Instructions Implemented

The fixed-point scalar instructions that have been implemented are described in this appendix[7]. Here the AA bit and LK bit values are determined by the mnemonic option of [a] and [l] respectively, the RC bit and OE bit values are determined by the mnemonic option of [.] and [o] respectively.

Table A.1: Branch Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
b[l][a]	18	/	Branch
bc[l][a]	16	/	Branch Conditional
bcctr[l]	19	528	Branch Conditional to Count Register
bclr[l]	19	16	Branch Conditional to Link Register

Table A.2: Compare Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
cmpb	31	508	Compare Bytes
cmpi	11	/	Compare Immediate
cmp	31	0	Compare
cmpl	31	32	Compare Logical
cmpli	10	/	Compare Logical Immediate

Table A.3: Conditional Logical Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
crand	19	257	Condition Register AND
crandc	19	129	Condition Register AND with complement
creqv	19	289	Condition Register Equivalent
crnand	19	225	Condition Register NAND
crnor	19	33	Condition Register NOR
cror	19	449	Condition Register OR
crorc	19	417	Condition Register OR with complement
crxor	19	193	Condition Register XOR

Table A.4: Logical Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
andi.	28	/	AND Immediate
ori	24	/	OR Immediate
andis.	29	/	AND Immediate Shifted
oris	25	/	OR Immediate Shifted
xori	26	/	XOR Immediate
xoris	27	/	XOR Immediate Shifted
and[.]	31	28	AND
or[.]	31	444	OR
xor[.]	31	316	XOR
nand[.]	31	476	NAND
nor[.]	31	124	NOR
eqv[.]	31	284	Equivalent
andc[.]	31	60	AND with Complement
orc[.]	31	412	OR with Complement
extsb[.]	31	954	Extend sign byte
extsh[.]	31	922	Extend sign halfword
extsw[.]	31	986	Extend sign word

Table A.5: Rotate and shift, Arithmetic Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
rlwinm[.]	21	/	Rotate Left Word Immediate then AND with Mask
rlwnm[.]	23	/	Rotate Left Word then AND with Mask
rlwimi[.]	20	/	Rotate Left Word Immediate and then Mask Insert
rldicl[.]	30	0	Rotate Left Doubleword Immediate then Clear Left
rldicr[.]	30	1	Rotate Left Doubleword Immediate then Clear Right
rldic[.]	30	2	Rotate Left Doubleword Immediate then Clear
rldcl[.]	30	8	Rotate Left Doubleword then Clear Left
rldcr[.]	30	9	Rotate Left Doubleword then Clear right
rldimi[.]	30	3	Rotate Left Doubleword Immediate Mask Insert
slw[.]	31	24	Shift Left Word
srw[.]	31	536	Shift right Word
srawi[.]	31	824	Shift Right Algebraic Word Immediate
sraw[.]	31	792	Shift Right Algebraic Word
sld[.]	31	27	Shift Left Doubleword
srd[.]	31	539	Shift Right Doubleword
sradi[.]	31	413	Shift Right Algebraic Doubleword Immediate
srad[.]	31	794	Shift Right Algebraic Doubleword
mtspr	31	467	Move To Special Purpose Register
mfspr	31	339	Move From Special Purpose Register
mtocrf	31	144	Move To One Condition Register Field
mfocrf	31	19	Move From One Condition Register Field
mcrxr	31	512	Move To Condition Register from XER
addi	14	/	ADD Immediate
addis	15	/	ADD Immediate Shifted
add[o][.]	31	266	ADD
subf[o][.]	31	40	Subtract From
addic	12	/	ADD Immediate Carrying
addic.	13	/	ADD Immediate Carrying and Record
subfic	8	/	Subtract From Immediate Carrying
addc[o][.]	31	10	ADD Carrying
subfc[o][.]	31	8	Subtract From Carrying
adde[o][.]	31	138	ADD extended
subfe[o][.]	31	136	Subtract From extended
addme[o][.]	31	234	ADD to Minus one extended
subfme[o][.]	31	232	Subtract From Minus one extended
addze[o][.]	31	202	ADD to Zero extended
subfze[o][.]	31	200	Subtract From Zero extended
neg[o][.]	31	104	Negate

Table A.6: Load Store Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
lbz	34	/	Load byte and zero
lbzx	31	87	Load byte and zero Indexed
lbzu	35	/	Load byte and zero with update
lbzux	31	119	Load byte and zero with update indexed
lhz	40	/	Load Halfword and zero
lhzx	31	279	Load Halfword and zero Indexed
lhzu	41	/	Load Halfword and zero with update
lhzux	31	311	Load Halfword and zero with update indexed
lha	40	/	Load Halfword Algebraic
lhax	31	279	Load Halfword Algebraic Indexed
lhau	41	/	Load Halfword Algebraic with update
lhaux	31	311	Load Halfword Algebraic with update indexed
lwz	32	/	Load word and zero
lwzx	31	23	Load word and zero Indexed
lwzu	33	/	Load word and zero with update
lwzux	31	55	Load word and zero with update indexed
lwa	58	2	Load word Algebraic
lwax	31	341	Load word Algebraic Indexed
lwaux	31	373	Load word Algebraic with update indexed
ldz	58	0	Load Doubleword and zero
ldzx	31	21	Load Doubleword and zero Indexed
ldzu	58	1	Load Doubleword and zero with update
ldzux	31	53	Load Doubleword and zero with update indexed
stb	38	/	Store byte
stbx	31	215	Store byte Indexed
stbu	39	/	Store byte with update
stbux	31	247	Store byte with update indexed
sth	44	/	Store Halfword
sthx	31	407	Store Halfword Indexed
sthu	45	/	Store Halfword with update
sthux	31	439	Store Halfword with update indexed
stw	36	/	Store Word
stwx	31	151	Store Word Indexed
stwu	37	/	Store Word with update
stwux	31	183	Store Word with update indexed
std	62	0	Store Doubleword
stdx	31	149	Store Doubleword Indexed
stdu	62	1	Store Doubleword with update
stdux	31	181	Store Doubleword with update indexed

Table A.7: Load Store Instructions

Mnemonics	*Opcode*	*Extendedopcode*	*Instruction*
lhbrx	31	790	Load halfword Byte-Reverse Indexed
sthbrx	31	918	Store halfword Byte-Reverse Indexed
lwbrx	31	534	Load word Byte-Reverse Indexed
stwbrx	31	662	Store word Byte-Reverse Indexed
ldbrx	31	532	Load Doubleword Byte-Reverse Indexed
stdbrx	31	660	Store Doubleword Byte-Reverse Indexed
lmw	46	/	Load Multiple Word
stmw	47	/	Store Multiple Word

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

[4] Freescale Semiconductor, *e5500 Core Reference Manual*, rev. 1 ed., 2012.

[5] Freescale Semiconductor, *EREF 2.0:A Programmers Reference Manual for Freescale Power Architecture Processors*, rev. 0 ed., 2011.

[6] J. P. Shen and M. H.Lipasti, *Modern Processor Design*. Tata McGraw-Hill., 2005.

[7] IBM, *Power ISA*, version 2.06 revision b ed., 2010.