PERFORMANCE EVALUATION OF QUEUE PROCESSORS Vs RISC ARCHITECTURE

A

Thesis

Presented to

African University of Science and Technology

In Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

By

MADUAGWU DOROTHY

Supervisor: Prof. Ben Abderazek Abdallah

December 2010

ABSTRACT

Nowadays, shifts in Hardware and Software technologies have forced designers and users to look at micro-architecture that process instructions stream with high performance and low power consumption.

In Striving for such high performance, the Queue Processor has been designed with architecture which has the following features:

- Low power consumption
- Smaller code size
- Simple Hardware
- High Performance in terms of Speed
- High Instruction level parallelism

This research aims at comparing and evaluating these performance features of the Queue Processor architecture with the traditionally used RISC architecture. Evaluation will be done in terms of Software (code size, execution time) and Hardware (Logical Elements, power and speed). This evaluation is performed using Quartus II IDE by Altera.

The QSoC will be used as case study for the Queue Processor while Aquarius will be used as case study for the RISC processor.

I'm confident that this evaluation research will show a significant improvement in the performance of the Queue Processor over the RISC Architecture.

ACKNOWLEDGEMENT

I am grateful to my God Almighty for His guidance and sustenance during the course of my Masters' program.

I am heartily thankful to my supervisor, Prof. Ben Abdallah Abderazek, whose guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

I also owe my deepest gratitude to my dear husband who inspired, supported and encouraged me all the way.

Lastly, I offer my regards and blessings to all my colleagues who supported me in any respect during the completion of this Thesis.

Dorothy Maduagwu

TABLE OF CONTENTS

TABLE OF FIGURES/TABLES/CHARTS

CHAPTER ONE

INTRODUCTION

1.1 Importance of Performance Evaluation

Performance evaluation is at the foundation of computer architecture research and development. Contemporary microprocessors are so complex that architects cannot design systems based on intuition and simple models only.

Adequate performance evaluation methods are absolutely crucial to steer the research and development process in the right direction. However, rigorous performance evaluation is non-trivial as there are multiple aspects to performance evaluation, such as picking workloads, selecting an appropriate modelling or simulation approach, running the model and interpreting the results using meaningful metrics. Each of these aspects is equally important and a performance evaluation method that lacks rigor in any of these crucial aspects may lead to inaccurate performance data and may drive research and development in a wrong direction [04].

The major aims of Performance Evaluation are to:

- Collect and disseminate information relative to performance aspects, and in particular to a specific topic.
- Promote interdisciplinary flow of technical information among researchers and professionals.
- Serve as a publication medium for various special interest groups in the performance community at large.

1.2 Research Objectives

This research studies extensively, the Queue Processor Architecture in general and evaluates the QSoC (Queue System on Chip) in specific.

This research compares two different processor architectures: Queue Processor (using QSoC from ASL as case study) and RISC Processor (using Aquarius from OpenCores as case study).

Through extensive simulation experiments, the performance of the Queue Processor is evaluated alongside the RISC Architecture.

This evaluation is done in terms of Hardware:

- Logical elements
- power
- speed

And Software:

- Code size
- Execution time

The work consists of three parts: initial analysis, implementation and benchmarking.

During the initial analysis, the processor architectures will be analyzed and compared based on characteristics such as pipeline depth, Instruction Set Architecture, data path and control path.

Each processor is synthesized and implemented on a DE-II FPGA board.

Characteristics such as gate count, maximum clock frequency, and performance is measured. Performance of the implemented processors is measured with a set of standard benchmarks.

This research is aimed at identifying the significant improvement in the performance of the Queue Processor over the RISC Architecture.

1.3 Motivation for Research

Nowadays, shifts in Hardware and Software technologies have forced designers and users to look at micro-architecture that process instructions stream with high performance and low power consumption.

Queue computing and architecture design approaches take into account performance and power consumption considerations early in the design cycle and maintain a power-centric focus across all levels of design abstraction.

This is especially useful since power has become a problem in most countries. The importance of the use of a processor which consumes and dissipates less power cannot be over emphasized.

To address this issue, and especially to increase processing speed, it is believed that the Queue processor provides an interesting alternative to the design of embedded systems.

1.4 Queue Computing

The accelerated demand in increasing performance has resulted in the research into and the development of higher performance and less power consuming architectures which employ queue computing.

Queue Computing is simply processing data using queues. The queue data structure uses the FIFO (First In First Out) scheme whereby data that comes in first is processed first.

Queue computing model refers to the evaluation of expression using FIFO queue, called operand queue instead of registers as intermediate storage of results [02].

This model establishes two rules for the insertion and removal of elements from the operand queue. Operands are inserted, or en-queued, at the rear of the queue. And operands are removed, or de-queued, from the head of the queue. Two references are needed to track the location of the head and the rear of the queue. The Queue Head (QH) points to the head of the queue, and Queue Tail (QT) points to the rear of the queue [02].

Queue processors offer a very attractive alternative for the design of embedded processors given their characteristics of

- Small Instructions
- Simple Hardware

- High Instruction Level Parallelism
- Free from False Dependencies

1.5 Thesis Outline

This thesis work is divided into six chapters. The first chapter gives an introduction to queue computing, explaining the research objectives and approach taken to develop the topic.

The second chapter develops a literature review of processor architecture. It gives detailed analysis of the Produced Order Queue Computing, the Circular Queue Register structure, and the Queue core in terms of architecture, ISA (Instruction Set Architecture), data path and control.

The third chapter discusses the specific features (like level order traversal, operands not explicitly specified, absence of register renaming, etc) which enable it extract high instruction level parallelism (ILP), lower power consumption and smaller code size. The RISC processor is also closely examined and analysed alongside the Queue processor.

Chapter four presents the methodology employed to carry out not just a theoretical analysis but simulations to support the higher performance of the Queue architecture over the RISC architecture.

The fifth chapter focuses on the results of the simulation and experiments carried out.

The final chapter summarizes and draws conclusions based on the work done. It further outlines future research areas with respect to the performance evaluation carried out.

CHAPTER TWO

LITERATURE REVIEW

Since computer was invented in the 40's, computer programmers and users have been requesting faster computers to solve larger and more complex computational tasks. This need for more computing power is going to be endless, because there will always be problems that any computer cannot solve fast enough. The request for more powerful computers has been, however, realized amazingly well. The performance of computers has been doubled in every second year during the last five decades. This has been the result of faster and smaller components, better integration of circuitries and better processor architectures.

2.1 A short history of Processor Architecture

A processor is the brain of the computer. It is the portion of the computer that tells the computer what to do and when to do it. It executes computing tasks according to given instructions.

Physically it consists of numerous interconnected digital gates. These gates form logical entities that preserve data (registers, latches, buffers), carry out the calculation (arithmetic and logical unit, ALU), take care of the communication to and from memory (memory unit, MU), or control the processor (sequencers, SEQ). ALU, MU, and SEQ are called processing elements or functional units (FU), because they process the data provided by the instructions. Some functional units may be assigned to special uses like address units (ADR) and compare units (CMP), which are actually ALUs dedicated to address calculations and comparing operands [05].

Different trends in scheduling the execution of instructions in a processor reflect distinctively the development of processor architectures.

The first processors were designed so that they executed instructions strictly sequentially [01]. These processors are called non-pipelined processors or scalar processors.

In the 50's pipelined execution or pipelining was invented to speed up the execution of instructions. The execution of instructions is divided into several parts called pipeline stages. The stages are connected to the next to form a pipe. Several instructions can be overlapped in a

pipeline by executing different stages of consecutive instructions simultaneously. We call processors that execute instructions in this manner pipelined processors.

In the late 70's processors using a smaller number of simpler machine language instructions were reinvented, because CISC processors were shown inefficient [01]. The optimization of programs was left to the job of a compiler. Processors using a smaller number of simpler instructions are called reduced instruction-set computer (RISC) processors.

Lately, the quest for high performance computing with minimal power has led designers at the micro-architectural level to create the queue processor which is discussed extensively in this chapter.

2.2 Measuring Processor Performance

We measure performance in an attempt to determine fitness for a particular purpose. A processor can be exceptionally fast at performing a certain kind of computation but offer insufficient performance for a different task. For example, the main processor of a desktop computer is not specialized for any particular type of programs and tries to perform all tasks equally well, while excelling at none. A GPU on the other hand, is specialized towards the graphics related operations needed for advanced 3D graphics.

To really know how well a processor performs a certain task, one would ideally have to implement the specific algorithm on that specific processor. Naturally, this is an unfeasible approach for processor evaluation. Simple metrics such as clock speed provide a hint of performance, but is almost useless by itself. Average amount of cycles per instruction reveals a bit more. However, to really get an idea we must put the processor in motion—we must run a program on it. By executing a mix of instructions corresponding to a real program we can get an estimate of the number of average instructions per clock cycle. However, a simple mix of instructions may not accurately model dependencies between instructions, which may or may not cause the processor to stall, leading to an optimistic performance estimate.

Benchmarks are programs designed to measure the performance of an entire computer system or a part thereof. Compared to simple instruction mixes, they better model inter-instruction dependencies and more accurately estimates performance. A synthetic benchmark performs no

real work, but tries to mimic the operations performed by a real program, while an application benchmark performs a real, application specific, task. Naturally, one benchmark does not fit all.

As previously stated, performance is application dependent and benchmarking programs must take this into account. Choosing the right benchmark is an important first step. In this thesis we are looking to measure general purpose speed and power performance.

2.3 Conventional Processors

Conventional processors begin one operation per cycle. To reduce Cycle Per Instruction (CPI) therefore requires starting more than one operation per cycle. This requires the processor to keep many instructions in flight, use dynamic scheduling and register renaming.

2.3.1 Issues with conventional Processors

- Processor gets 'hung up' on instructions requiring more than one clock cycle
- Low performance in terms of speed, time and power

2.3.2 Architectural Techniques

An attempt to achieve scalar and better performance resulted in several architectural techniques as outlined below.

Architectural Techniques

- **Instruction Pipelining** – an implementation technique whereby multiple instructions are overlapped in execution. This accelerates instruction execution.

Example:

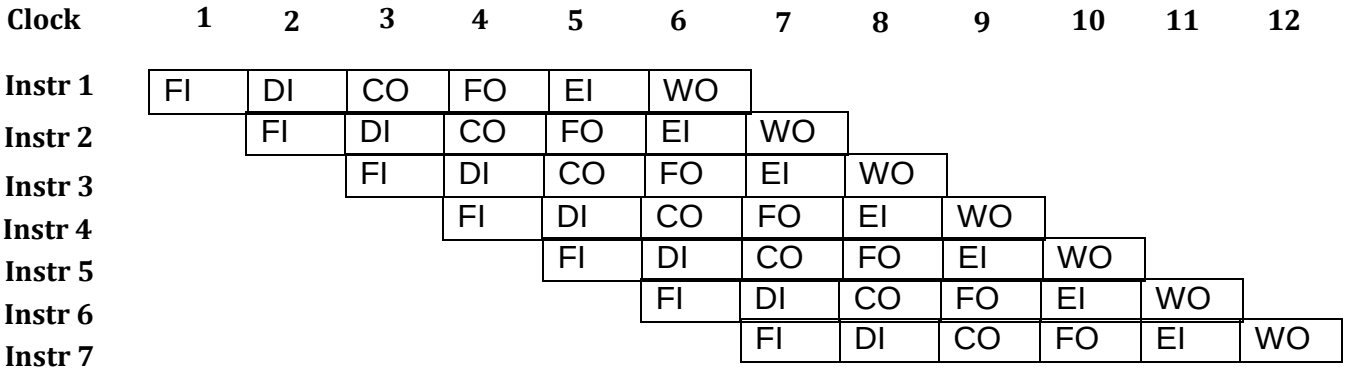


Fig 1 Pipelined Execution

Execution time of instructions without pipelining takes 7*6 = 42 time Units (Assuming equal duration for various stages.

Execution time of instructions using a 6 – stage pipeline will be reduced to 12 time Units as shown above.

FI – Fetch Instruction

DI – Decode Instructions

CO – Calculate Operand

FO – Fetch Operand

EI – Execute Instructions

WO – Write Operand

- **Super Scalar** – super scalar machines use their hardware to schedule parallelism. It employs instruction pipelining technique.

- **Very Long Instruction Word (VLIW)** – this approach executes operations in parallel on a fixed schedule determined when programs are compiled. VLIW CPUs offer significant computational power with less hardware complexity (but greater software complexity).

- **Out of Order (OoO) Execution** – this new paradigm is a technique used in high-performance microprocessors to make use of cycles that would otherwise be wasted by delay.

The key concept is to allow the processor to avoid ‘stalls’ that occur when the data needed to perform an operation are unavailable. OoO processors fill these slots in time with other instructions that are ready, and then re-order the results at the end to make it appear that the instructions were processed as normal.

2.4 Produced order Queue Computing

The produced order queue computing model uses a circular queue-register instead of random access registers to store intermediate results. Data is loaded in the QREG (Queue Register) in produced order scheme and can be reused [07, 08, and 09].

This feature has a profound implication in the areas of parallel execution, program compactness, hardware simplicity and high execution speed. [03, 10]

A special register called queue head, (QH), points to the first data in the QREG. Another pointer, named queue tail pointer (QT), points to the location of the QREG in which the result is stored. A live queue head pointer (LQH) is also used to keep used data that could be re-used from being overwritten. [11]

To demonstrate how this works, given the expression (a * b) + (c / d) = x. The figure below describes how this expression is evaluated using the produced order queue computing model.

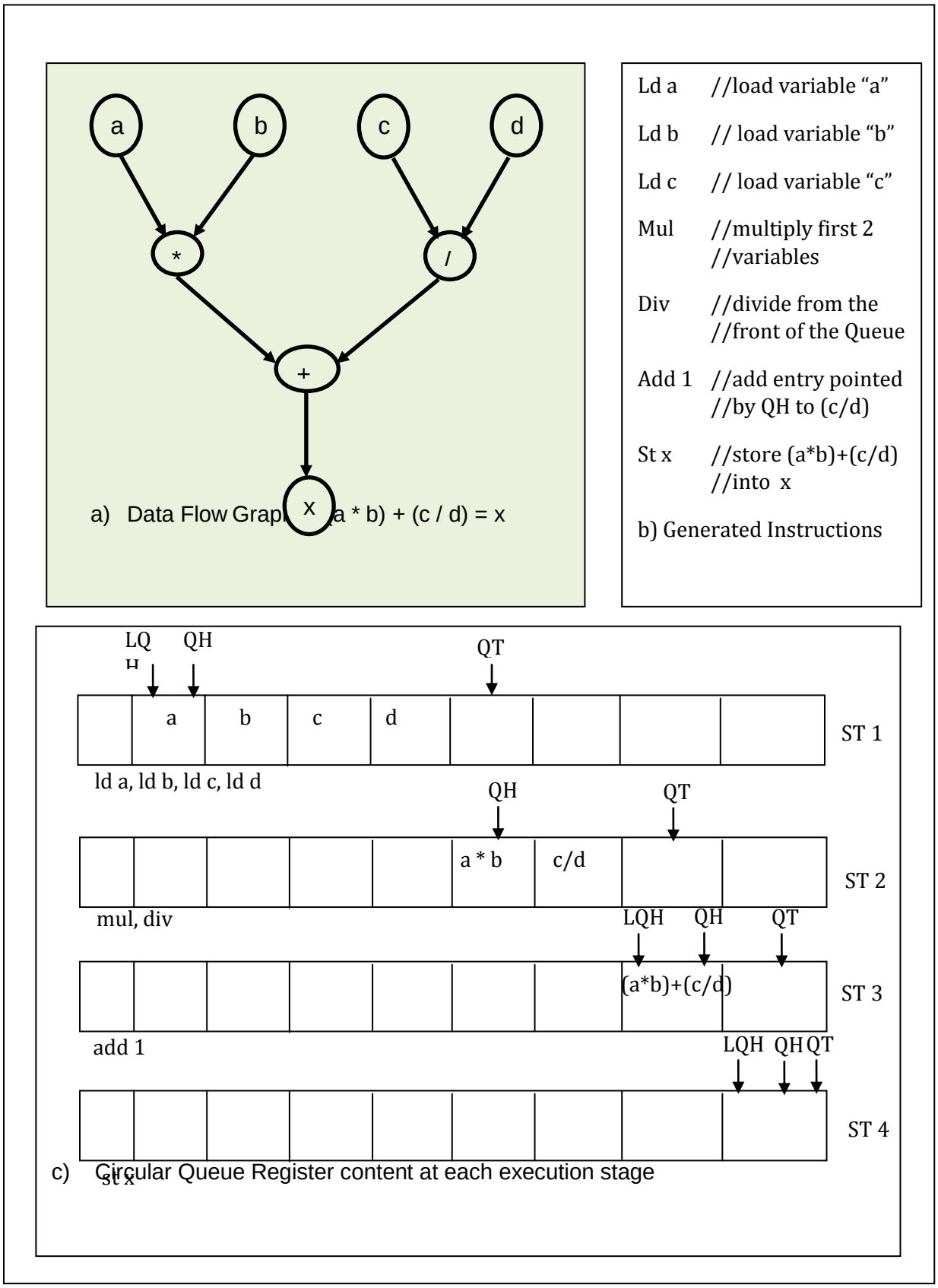


Fig. 2 Demonstration of Produced Order Queue Computing

Datum is loaded with load instruction (ld), computed with multiply (∗), add (+), and divide (/) instructions. The result is stored back in the data memory with store instruction (st).

The instruction sequence for the queue execution model is correctly generated when we traverse the data flow graph (shown in Fig. 1(a)) from left to right and from the highest to the lowest level. The generated instruction sequence from the data flow graph is shown in Fig. 1(b). The content of the QREG at each execution stage is shown in Fig. 1(c).

A special register, called queue head pointer, points to the first data in the QREG. Another pointer, named queue tail pointer, points to the location of the QREG in which the result is stored.

A live queue head pointer (LQH) is also used to keep used data that could be reused and thus should not be overwritten. These data, which are found between QH and LQH pointers, are called live-data. The live-data entries in the QREG are statically controlled. Two special instructions are used to stop or release the LQH pointer. Immediately after using the data, the QH is incremented so that it points to the data for the next instruction. QT is also incremented after the result is stored.

The four load instructions load in parallel a, b, c, and d data and place them into the QREG. At this state, QH points to datum a and the QT points to an empty location as shown in Fig. 1(c) (ST 1).

The fifth and sixth instructions are also executed in parallel. The mul refers a and b then inserts (a ∗ b) into the QREG. The div refers c and d then inserts (c / d) into the QREG. At this state, the QH, and QT are incremented as shown in Fig. 1(c) (ST 2).

The seventh instruction (add 1) adds the data pointed by QH (in this case (a * b)) by the data located at +1, offset, from QH (in this case (c / d)) as shown in Fig. 1(c) (ST 3).

The last instruction stores back the result in the data memory. Since the QREG becomes empty, LQH, QH, and QT point to the same empty location (ST 4).

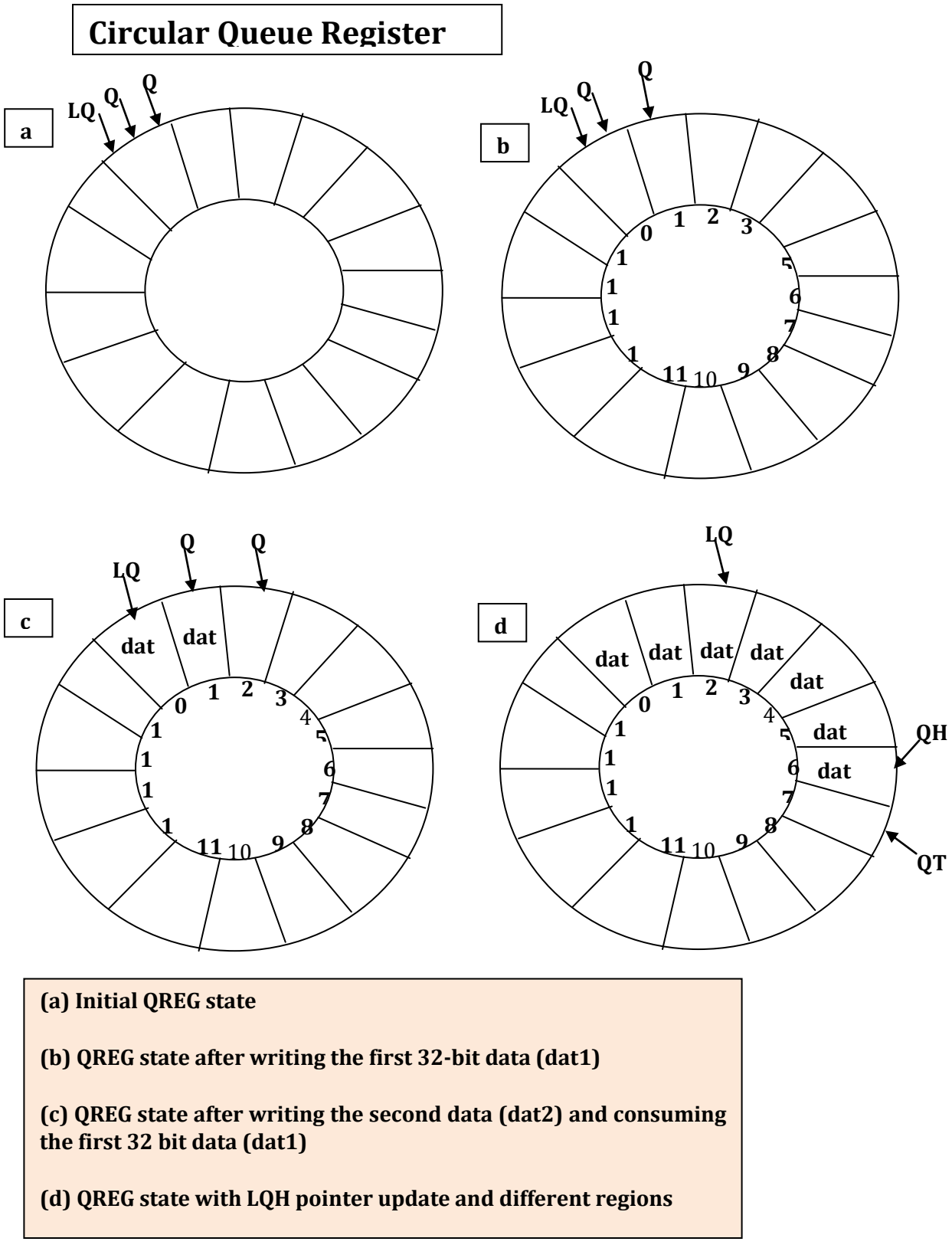


Fig 3. Circular Queue Register Structure

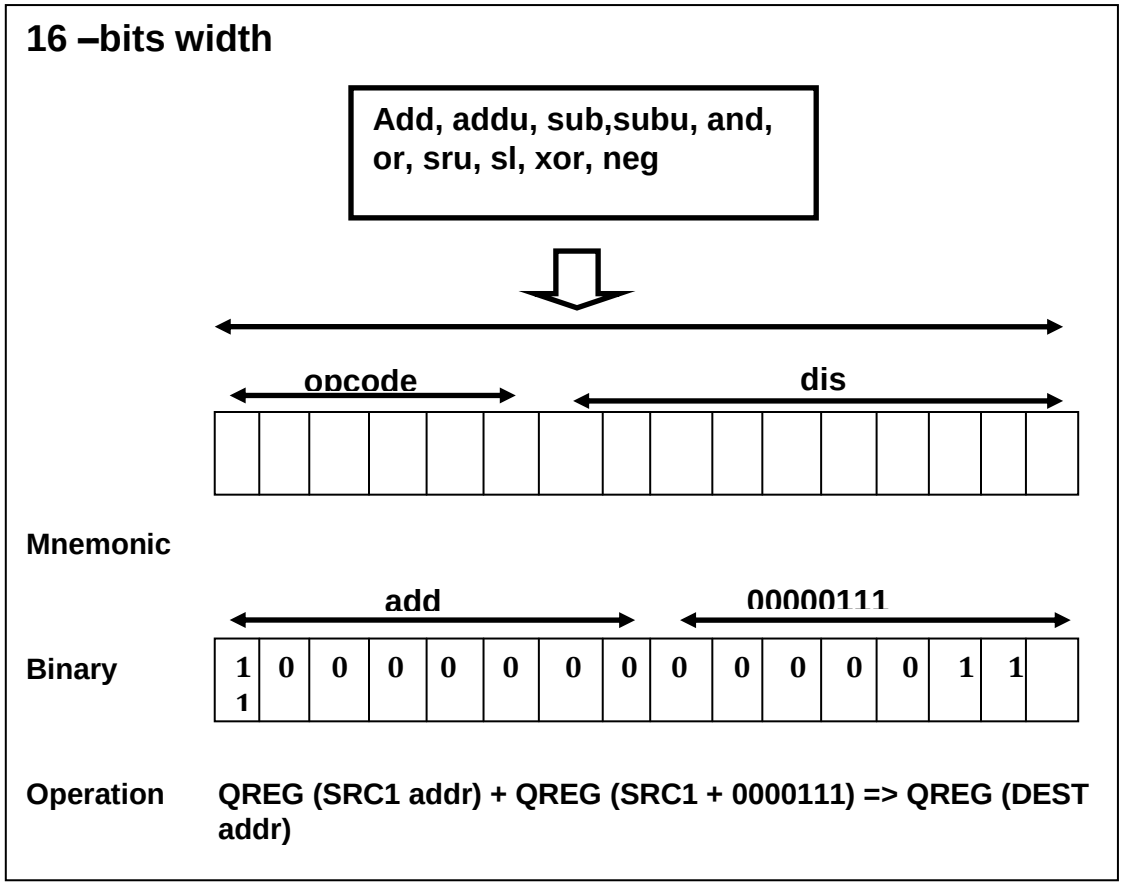
2.5 Queue - Core Architecture

In queue computing, all instructions are 16-bit wide, allowing simple instructions fetch and decode stages and facilitating pipelining of the processor.

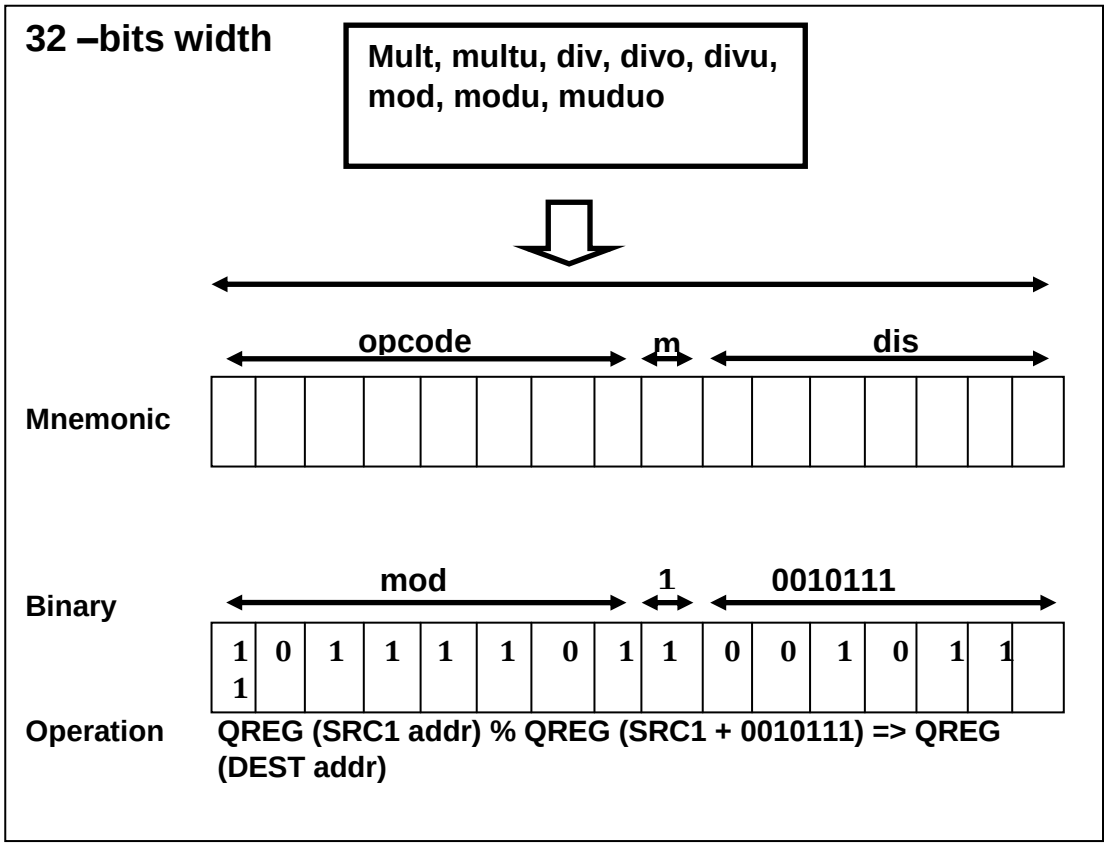
The instruction format reserves 8-bits for the Opcode and 8-bits for the Operand. The operand field is used in binary operations to specify the offset reference value with respect to QH from which the second source operand is dequeued, QH-N.

For cases where 8-bits are not enough to represent an immediate value or an offset for a memory instruction, a special instruction called 'covop' is used to precede the conflicting instruction. This special instruction extends the operand field of the instruction following it.

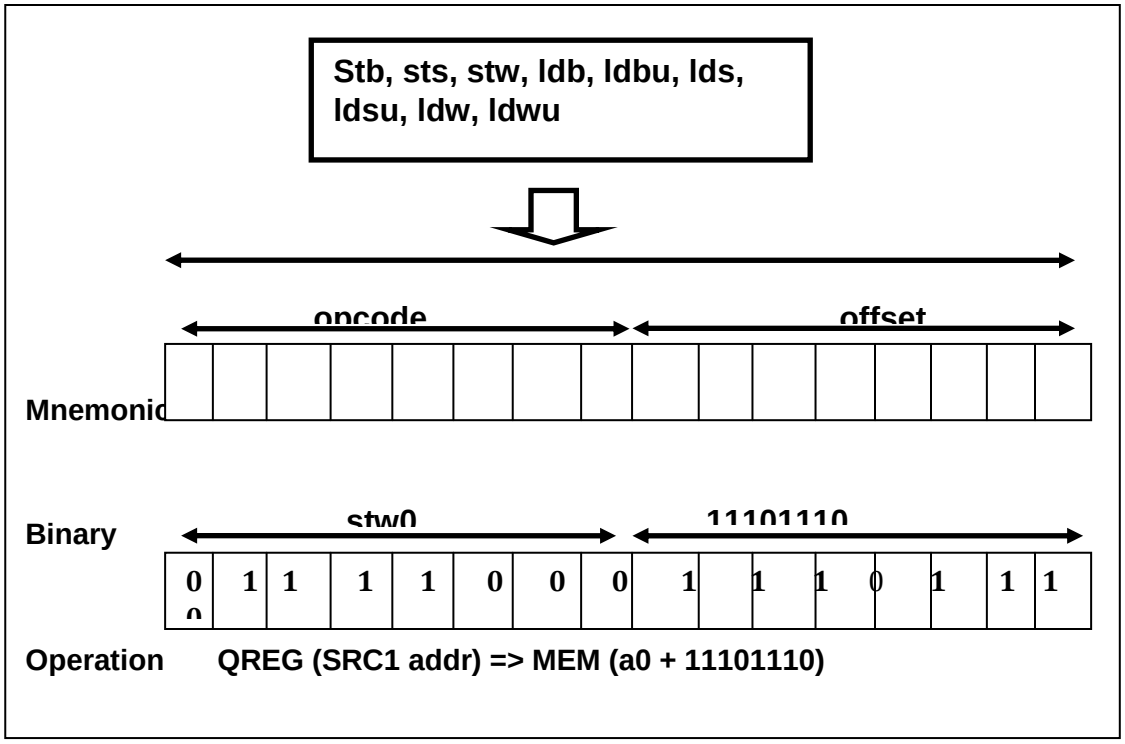
2.5.1 ALU (Arithmetic Logic Unit)



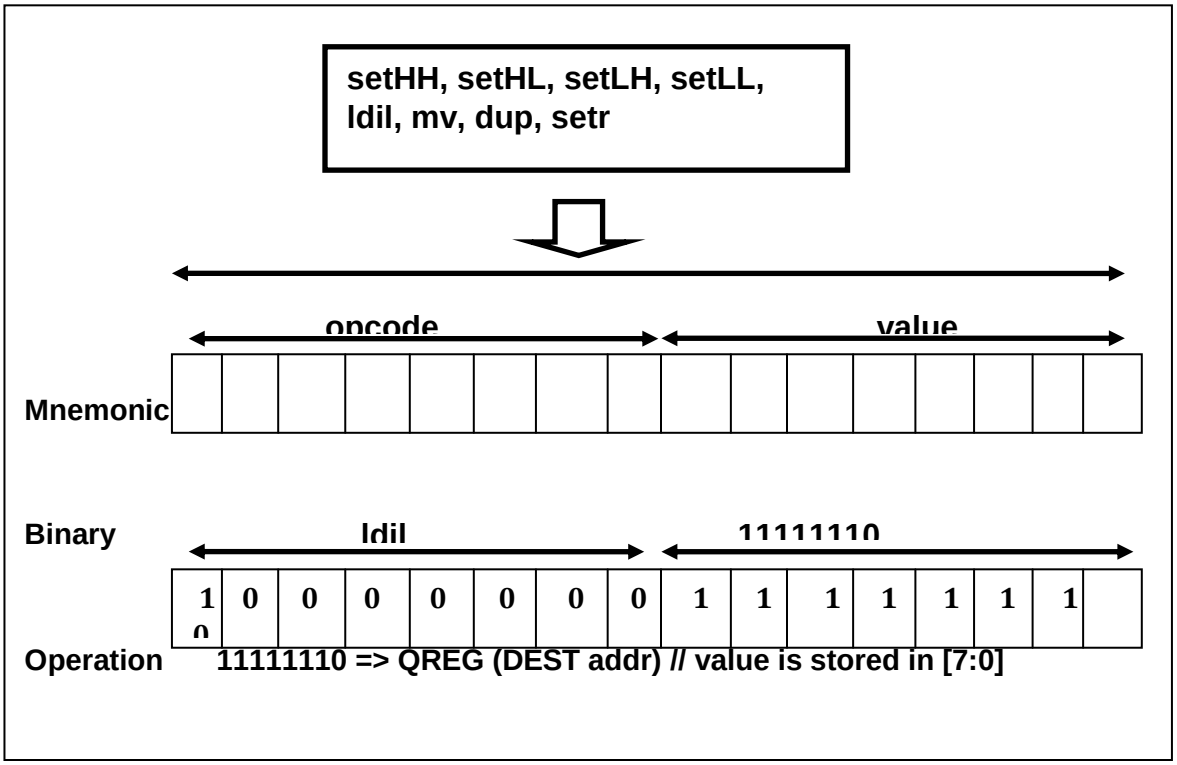
2.5.2 MLT – Signed 32 bit multiplier, divider and mod instructions



2.5.3 LOAD/STORE



2.5.4 SET



2.5.5 Branch

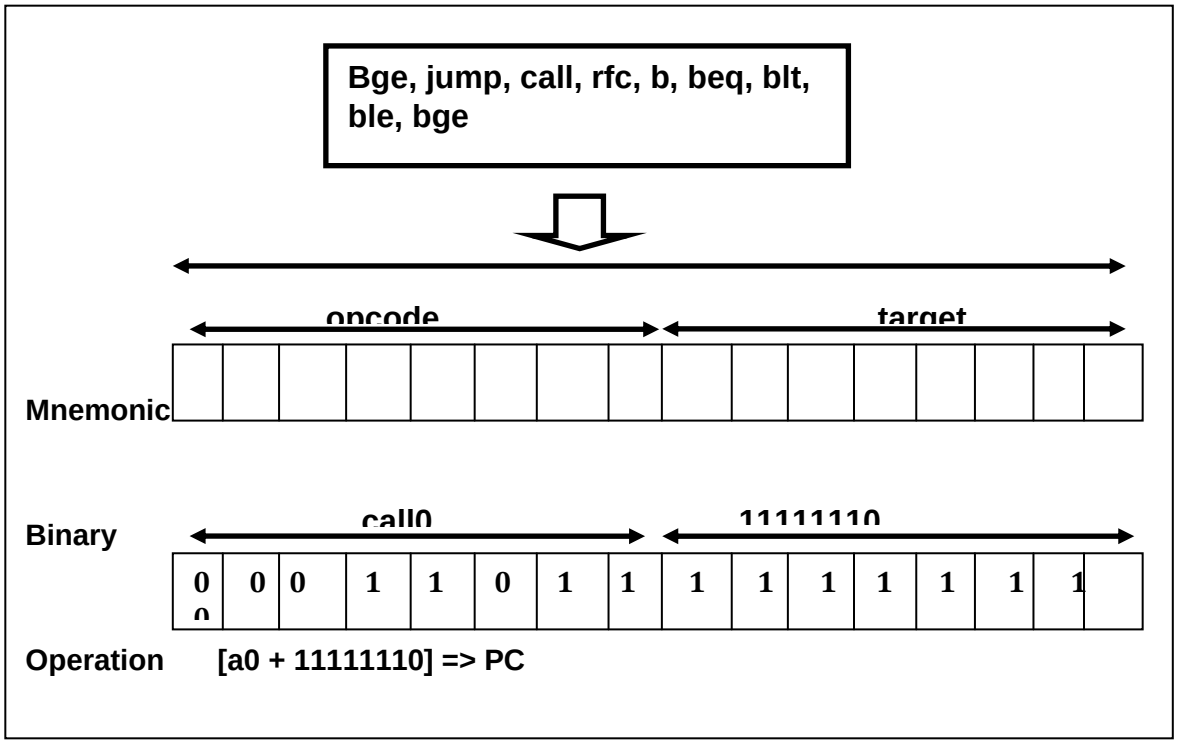


Fig 4. Queue ISA

2.6 Instruction Pipeline Structure

The execution pipeline operates in six stages combined with five pipeline-buffers to smooth-en the flow of instructions through the pipeline. The stages have been described below:

Fetch Unit – the fetch stage delivers four instructions to the decode unit each cycle. The address pointer hardware (APH) of the fetched instructions issues a new address to the memory system. This address is the previous address plus 8 bytes or the target address of the currently executing flow-control instruction.

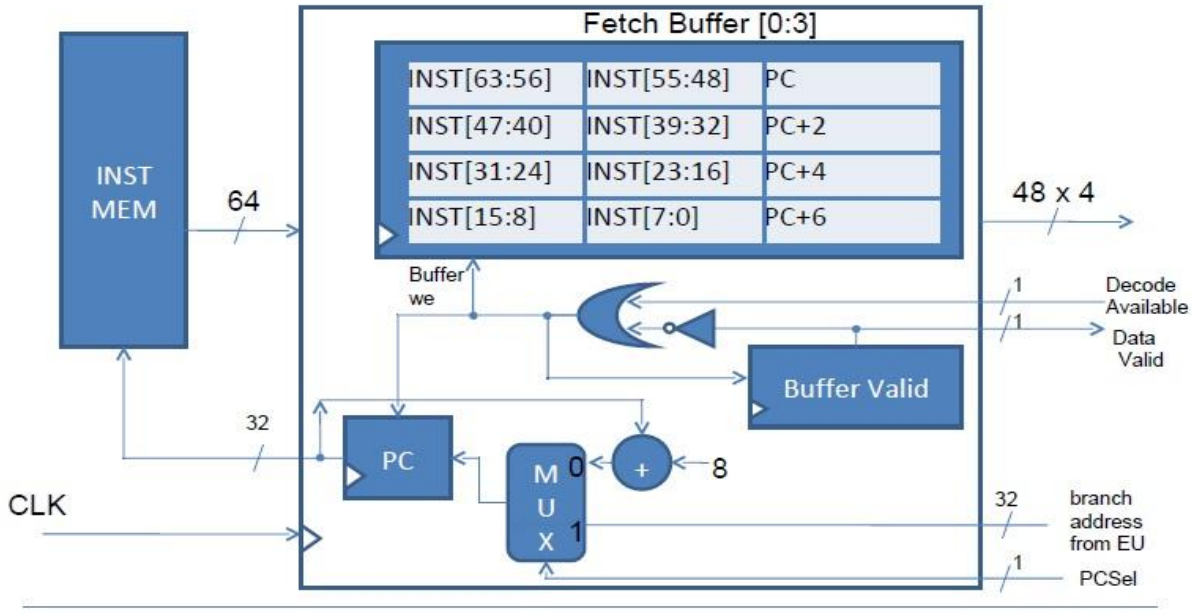


Fig 5. Instruction Fetch Data Path

Decode Unit (DU) – The DU decodes four instructions in parallel during the second phase and writes them into the decode buffer. This stage also calculates the number of consumed and produced data for each instruction which are used by the next pipeline stage to calculate the sources and destination locations for each instruction. Decoding stops if the queue buffer becomes full or if a halt signal is received from one or more stages following the decode stage.

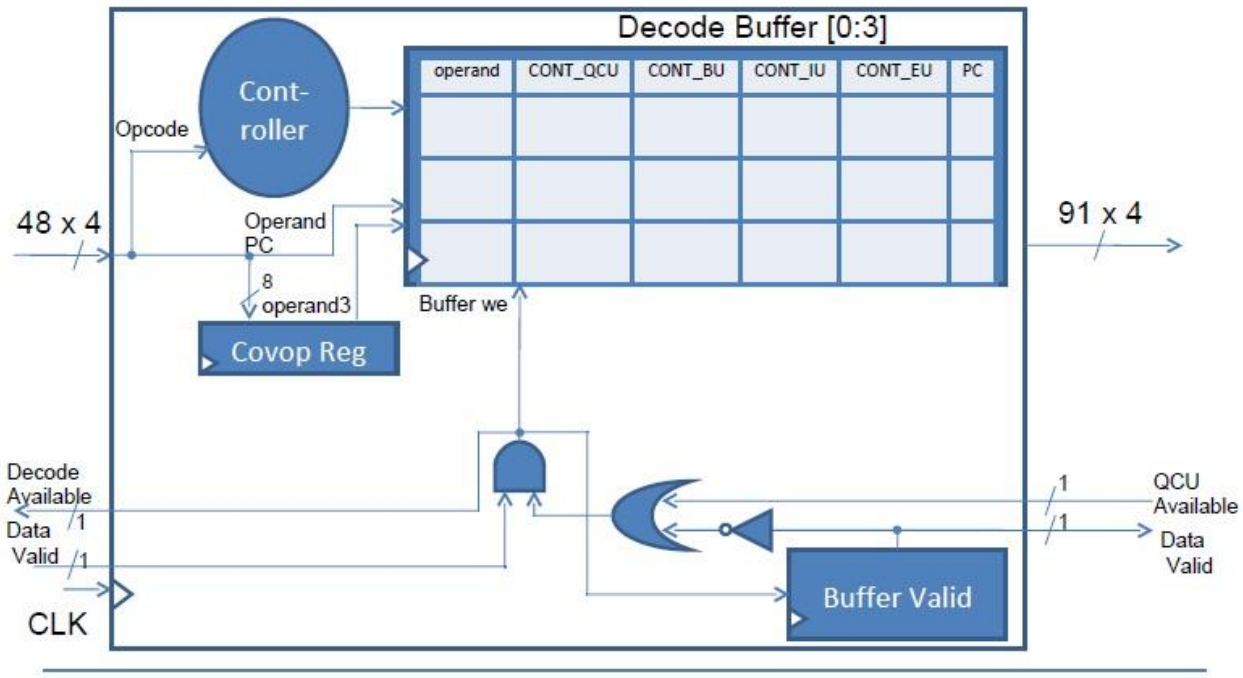


Fig 6. Instruction Decode Data Path

Queue Computation (QCU) - Four instructions arrive at the QCU unit each cycle. The QCU calculates the first operand (source1) and destination addresses for each instruction. The QCU unit keeps track of the current value of the QH and QT pointers.

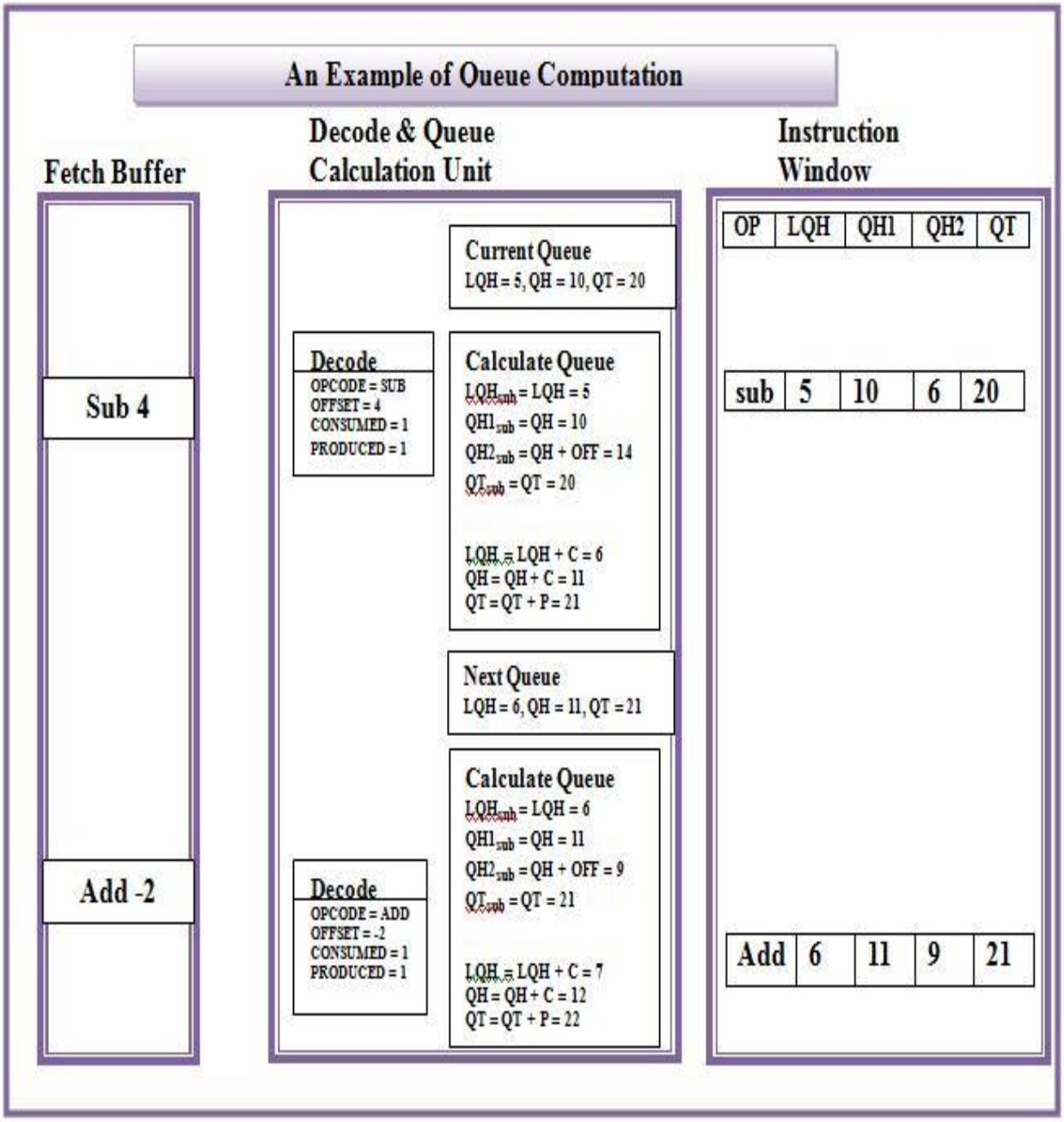


Fig 7 Queue Computation Example

Barrier - Inserts barrier flags for dependency resolutions.

Issue Stage - Four instructions are issued for execution each cycle. In this stage, the second operand (source2) of a given instruction is first calculated by adding the address source1 to the displacement that comes with the instruction. The second operand address calculation is

performed in the QCU stage. However, for a balanced pipeline consideration, the source2 is calculated at the beginning of the IS stage. An instruction is ready to be issued if its data and its corresponding functional unit are available. The processor reads the operands from the QREG in the second half of the IS stage.

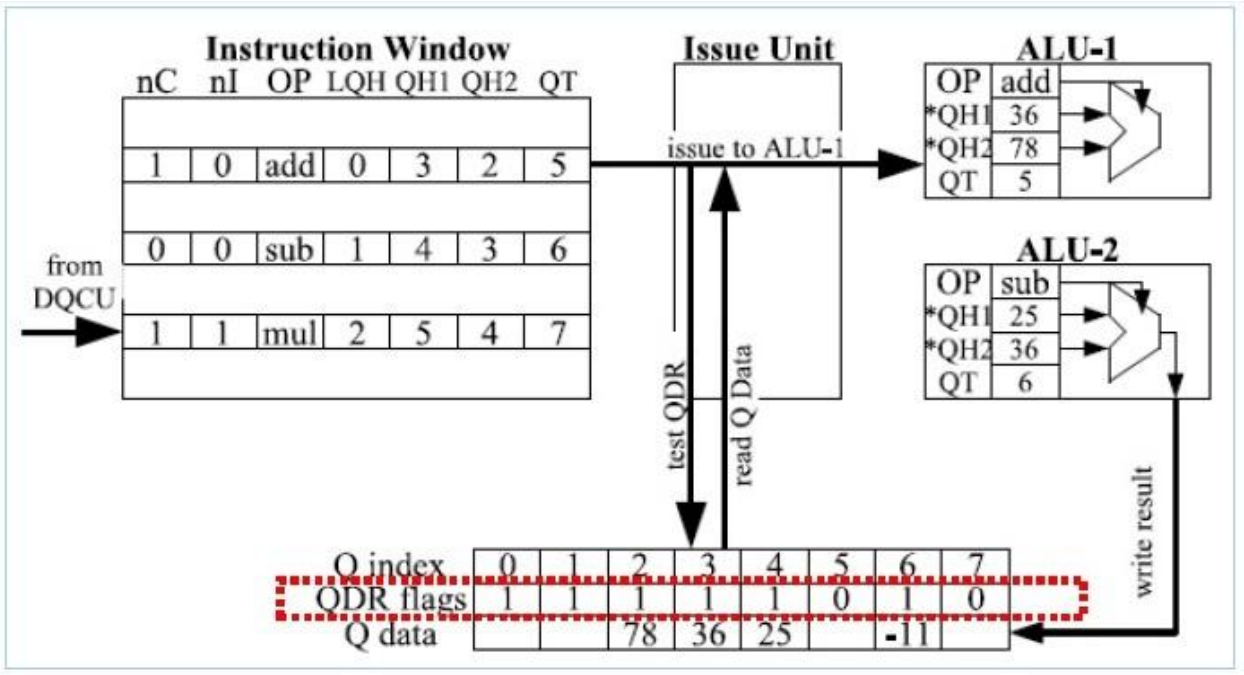


Fig 8 Instruction Issue Unit

Execution (EXE) - The macro data flow execution core consists of four integer ALU units, two floating-point units, one branch unit, one multiply unit, four set units, and two load/store units. The load/store units have their own address generation logic. Stores are executed to memory in-order.

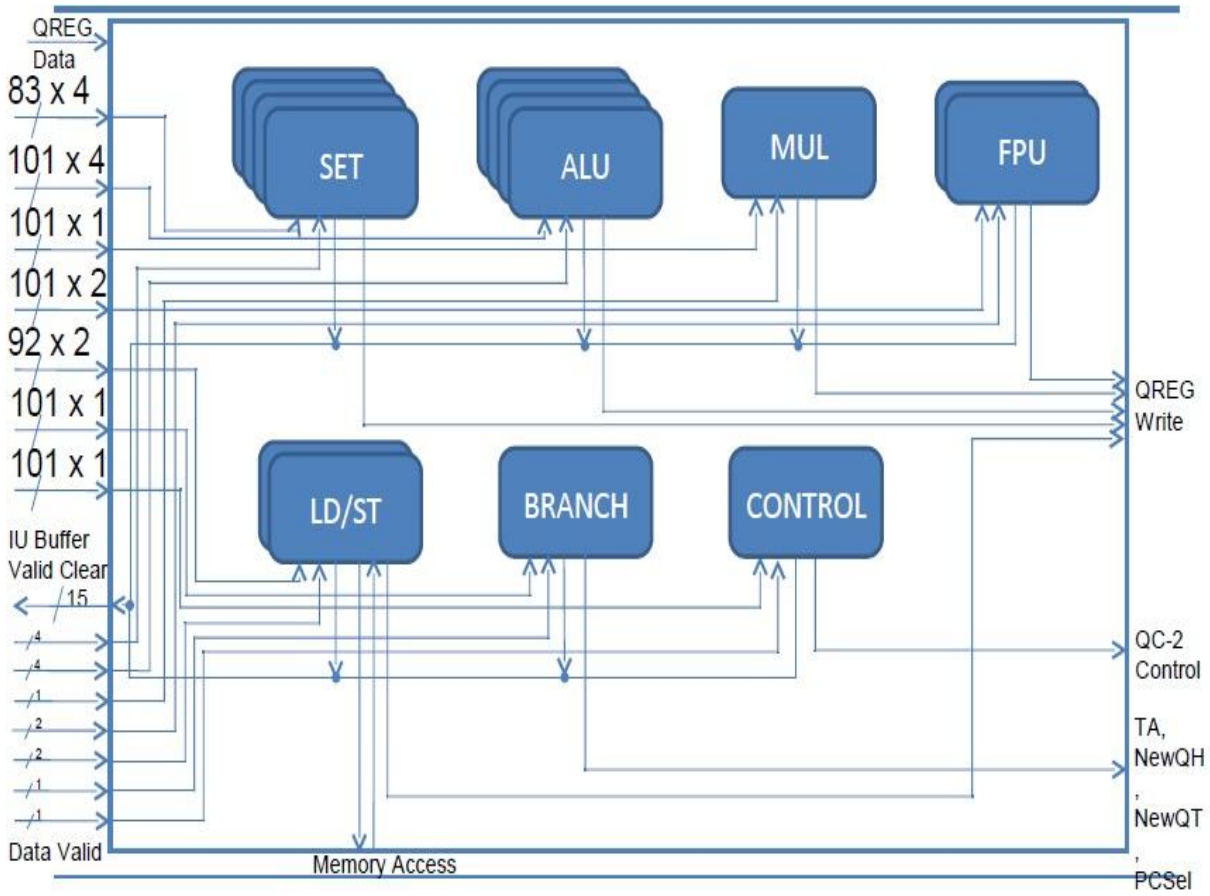


Fig 9. Execution Unit Data Path

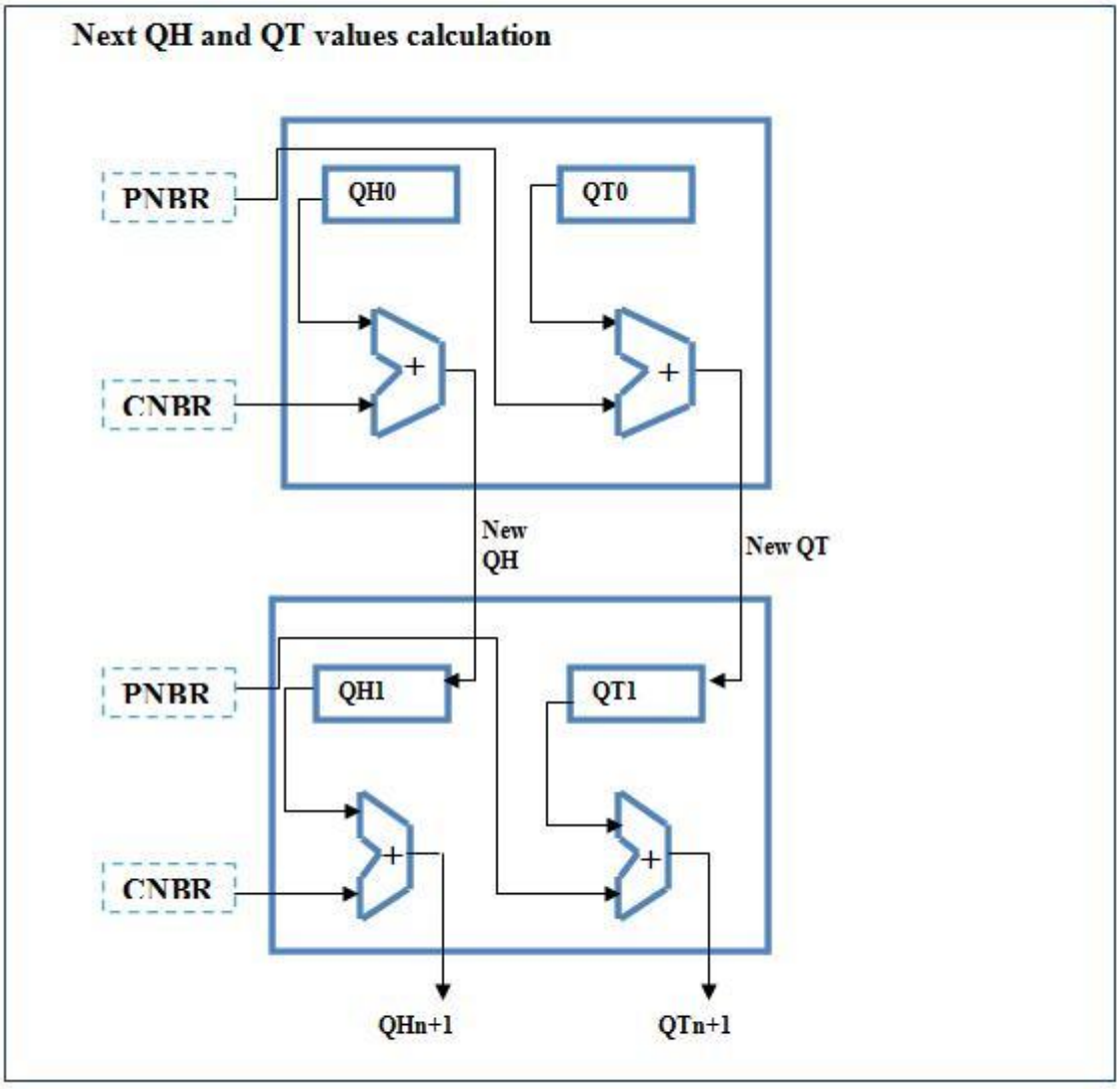


Fig 10. Calculation of next QH and QT values

Dynamic Operands Address Calculation

- To execute instructions in parallel, the QC-2 processor must calculate each instruction's operand(s) and destination address dynamically.
- To calculate the source1 address of a given instruction, the number of consumed data (CNBR) field is added to the current Queue Head value (QHn).

- The destination address on the next instruction (INSTn+1) is calculated by adding the PNBR field (8-bit) to the current queue tail value (QTn).

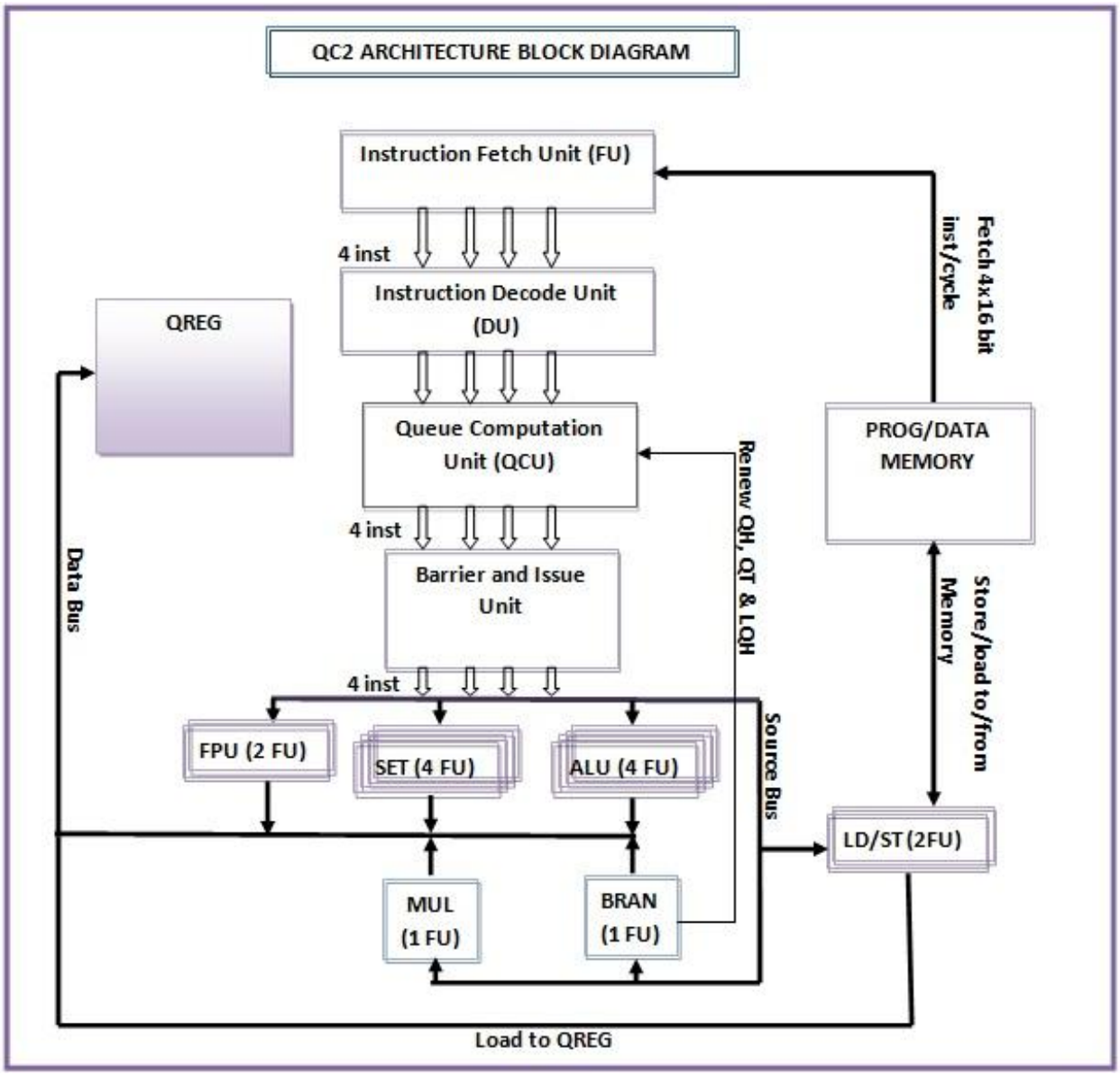


Fig 11. QC2 Architecture Block Diagram

CHAPTER 3

QUEUE Vs RISC MACHINES

The queue processor presents the bulk of work carried out in this thesis. This section presents a thorough analysis of the Queue processor alongside the RISC processor. It also examines the advantages of the Queue processor over the RISC processor. Some of its drawbacks are also outlined. Furthermore, all development tools used for software and hardware development are described. A flow chart showing the methodology is found below.

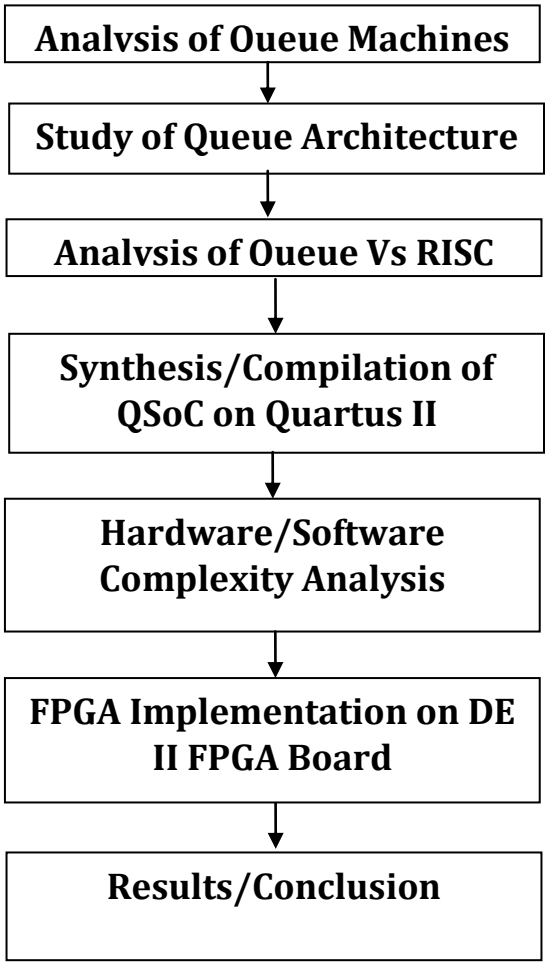


Fig 12 Research methodology adopted

3.1 Queue Machines Analysis

The analysis of the queue machines revealed the following advantages over its RISC counterpart:

3.1.1 Higher Instruction Level Parallelism (ILP)

ILP is the measure of how many operations in a computer program can be executed in parallel. It allows a processor to execute multiple instructions that are data independent in parallel. This is the key to improving performance in modern general purpose architectures.

The queue processor groups instructions that are data independent and executes them in parallel. This is called **Grouped ILP**. ILP allows the independent instructions of a sequential program to be executed in parallel on multiple functional units. Careful scheduling of instructions is crucial to achieving high performance.

Furthermore, such grouped ILP leads to Smaller Instruction Window which in turn requires less complexity. This leads to less power consumption

The queue processor exploits high Instruction Level Parallelism by executing programs using a Level Order Traversal scheme.

Its counterpart RISC processor does not exhibit such Grouped ILP and so has large Instruction Window which leads to more complexity and thus consuming a greater amount of power.

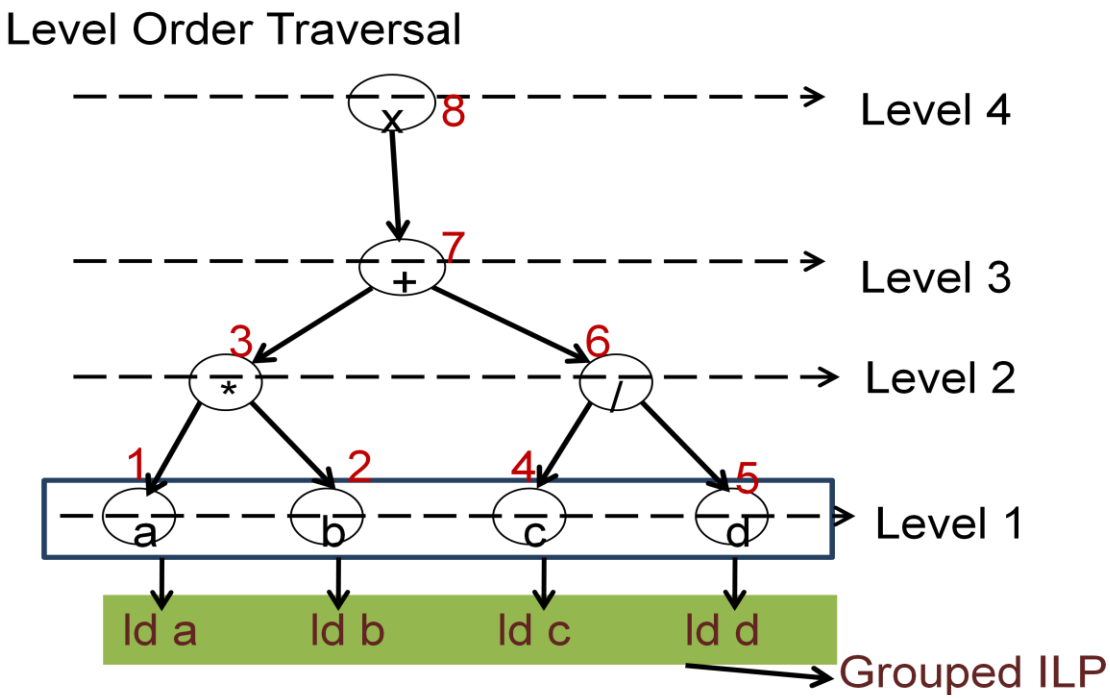


*Fig 13 Evaluation of (a * b) + (c / d) using Level Order Traversal*

In a RISC machine the expression in Fig 12 would be evaluated in eight (8) phases using a Post Order Traversal. The queue machine however, employs the use of Level order Traversal to evaluate the expression in four (4) phases. The load instructions for operands a, b, c and d will be executed simultaneously, that is in parallel since they are of the same level on the parse tree. Similarly, the multiplication and the division instructions will be executed at the same time, after which the addition is carried out on the third level and lastly the result gets stored into x at the fourth level.

It executes data independent instructions on the same level in parallel. This is a core way in which queue machines exploit maximum parallelism and thus higher performance.

3.1.2 Reduced Instruction width

Instructions are shorter since they do not need to be explicitly specified. Data is implicitly taken from queue head and results stored in queue tail of the queue register (QREG). Not having explicit operands in the instructions make instructions short, improving code density.

The advantage of such code density is that smaller memory is required. Also, the smaller the memory usage, the less power that is consumed.

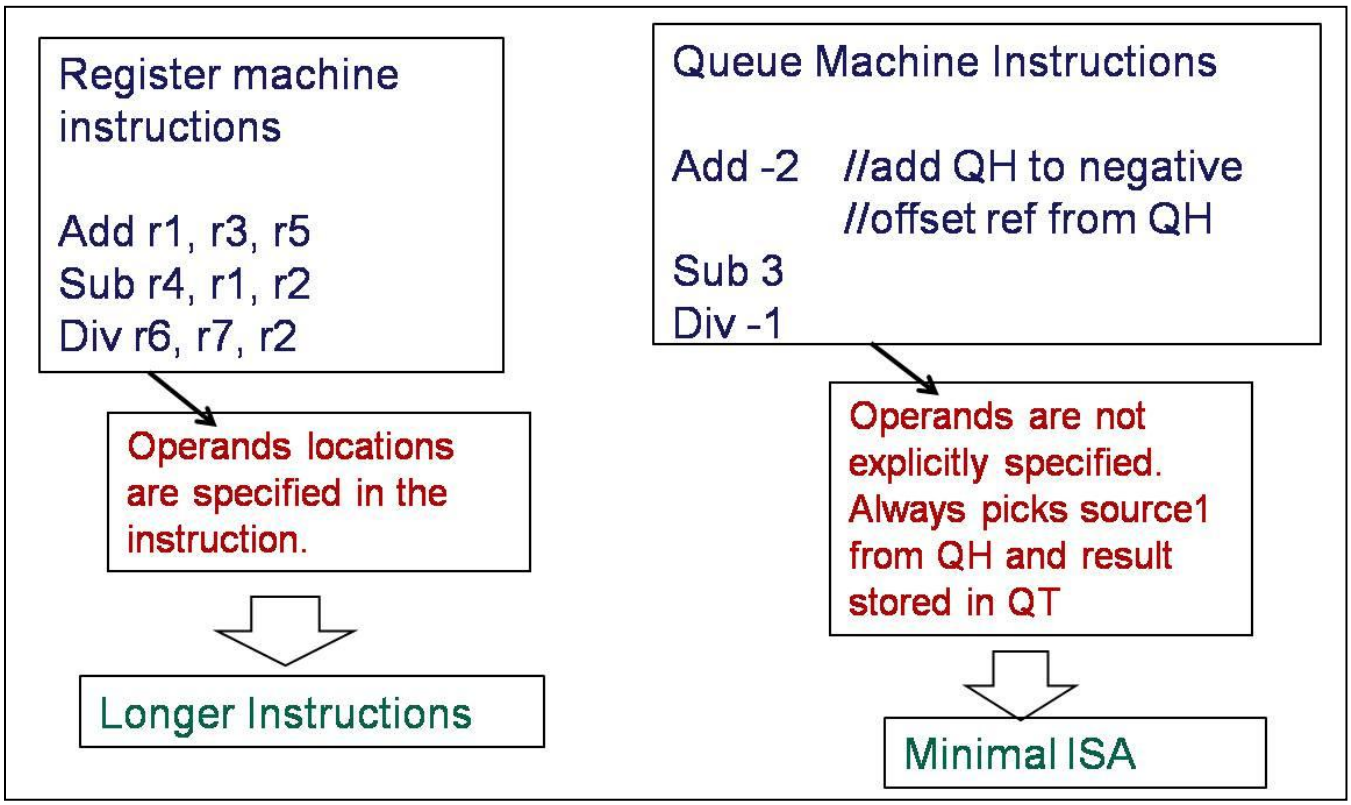


Fig 14 Comparison of program size

The Fig 13 above gives an example of instructions for both the queue machine and the RISC machine. The longer instructions of the RISC machine are as a result of the operands' location being explicitly specified in the instruction. Such long instructions consume more memory and invariably use more power.

Such smaller instruction width as seen in the queue machine instructions, places the queue processor at yet another advantage over its counterpart RISC processor.

3.1.3 Free from False dependencies

False dependencies are also called anti-dependencies. This is a WAR (Write After Read) dependency.

Example:

mul r1, r2, r3

add r2, r4, r5

This can be eliminated through register renaming as shown below:

mul r1, r2, r3

add r6, r4, r5

For Queue computing, because instructions read and write their operands implicitly, the design makes instructions independent of the actual number of physical queue words (QREG). This feature is thus free from false dependency as described above. Therefore register renaming unit is eliminated. This reduces circuitry and improves power consumption.

3.1.3.1 Register Renaming

Register renaming is a technique used to allow multiple execution paths without conflicts between different execution units trying to use the same registers. Instead of just one set of registers being used, multiple sets are put into the processor. This allows different execution units to work simultaneously without unnecessary pipeline stalls.

Register renaming refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations.

Register renaming is used to correct Write-after-read (WAR) data hazard.

- **Write-after-read (WAR)**

A data hazard is caused by data dependencies between instructions. If the first instruction writes to the same register that the second reads from, the write may not complete before the read, resulting in incorrect data being read.

A read from a register or memory location must return the last prior value written to that location, and not one written programmatically after the read. This is the sort of false dependency that can be resolved by renaming. WAR dependencies are also known as anti- dependencies.

Instead of delaying the write until all reads are completed, two copies of the location can be maintained, the old value and the new value. Reads that precede, in program order, the write of the new value can be provided with the old value, even while other reads that follow the write are provided with the new value. The false dependency is broken and additional opportunities for out-of-order execution are created. When all reads needing the old value have been satisfied, it can be discarded. This is the essential concept behind register renaming.

Results have shown that register renaming takes up to 4% of overall on-chip power in RISC processors.

The absence of false dependencies in the Queue processor has also means that Register Renaming technique is not required. This leads to Queue machines saving considerable amount of power.

This is thus another significant improvement of the queue based processor over the RISC based processor.

However, the following drawbacks were noted too:

3.1.4 Drawbacks of Queue Machines

For queue machines to evaluate an instruction, it first builds an expression tree for the instruction, and then schedules nodes in Level – Order. The tree is then traversed by level-order scheme.

If the expression tree is optimized into a directed acyclic graph (DAG) as shown in Fig 14, level – order scheduling no longer holds. Thus, DAGs are not executed correctly.

Furthermore, instructions are always read from the queue head and written to the queue tail. This does not allow for much flexibility.

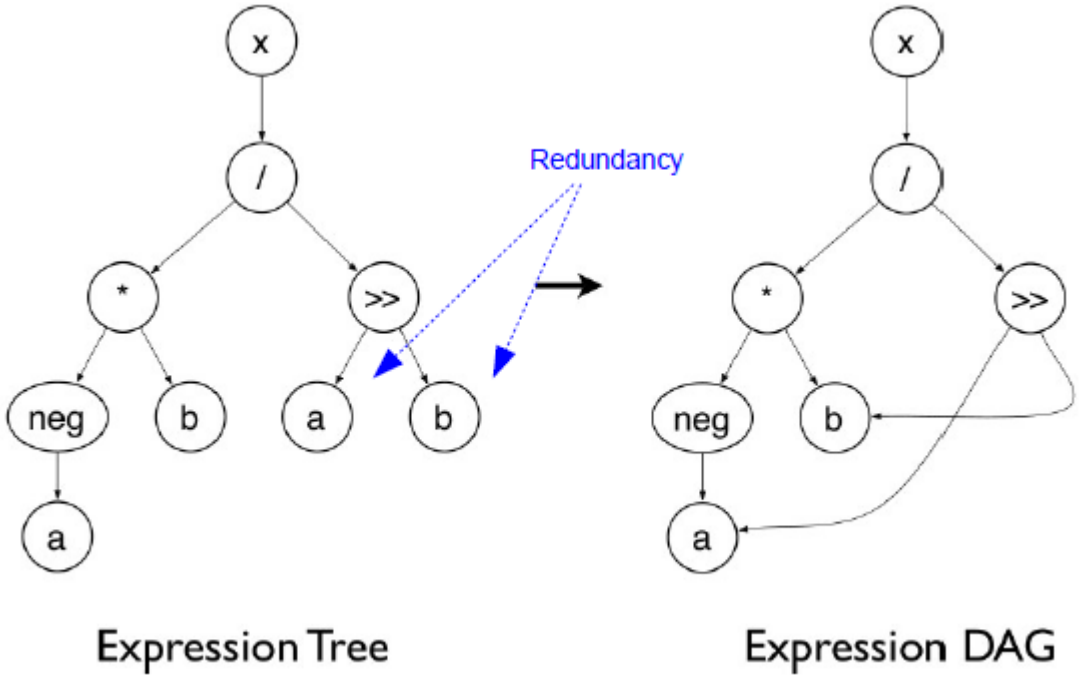


Fig 15 Problem of Queue Machines

3.2 QSoC Simulation and Synthesis

A thorough analysis of the queue processor was carried out. This led to identifying the features in it that enabled the exploitation of high Instruction Level Parallelism, lower speed and lower power consumption. After the theoretical analysis of both processors, simulations were carried out to back this up.

The QSoC was developed using Verilog Hardware Description Language (HDL). The HDL code was synthesized using Altera Quartus II 9.0 sp2 Web Edition.

After synthesizing the HDL code, the designed processor showed characteristics that enable investigation of the actual hardware performance and functional correctness. It also gives the possibility to study the effect of coding style and instruction set architectures over various optimizations.

For the QSoC processor to be useful for these purposes a modular approach was adopted to facilitate easier modifications – only relevant parts requiring modification need to be changed. Also, the processor description is synthesizable to derive actual implementations.

The QSoC was synthesized for a DE II FPGA board.

3.3 Quartus II Overview

Quartus II is a software tool produced by Altera for analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

The Web Edition was used because it is a free version that can be downloaded or delivered by mail for free. This edition also provides compilation and programming for a limited number of Altera devices.

The low-cost Cyclone family of FPGAs is fully supported by this edition.

3.4 FPGA Implementation of QSoC

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL).

In this research, the QSoC was compiled and synthesized using Quartus II and subsequently, the netlist generated was downloaded unto the Cyclone DE II FPGA Board for implementation. The FPGA emulates the design of my hardware.

The testbench used was that of a Clock which behaviour on the FPGA device is decribed below:

- The clock program enables a User set the time as he/she desires.
- The lower 4-bit and higher 4-bit of slide switches (SW0-SW7) indicate the 10's and 1's of the second, minute and hour.
- These slide switches can be set as desired. 32 bits hexadecimal notations were used to specify constants.
- Key 0 sets the clock to the default state of 0 second, 0 minute and 0 hour.
- Keys 3, 2 or 1 can be used to set the clock time. (These keys have the same function)
- After setting, the clock starts running from the time set

The results obtained from this synthesis are described in the following chapter.

3.5 Pictorial Summary of Queue Machines Vs RISC Machines

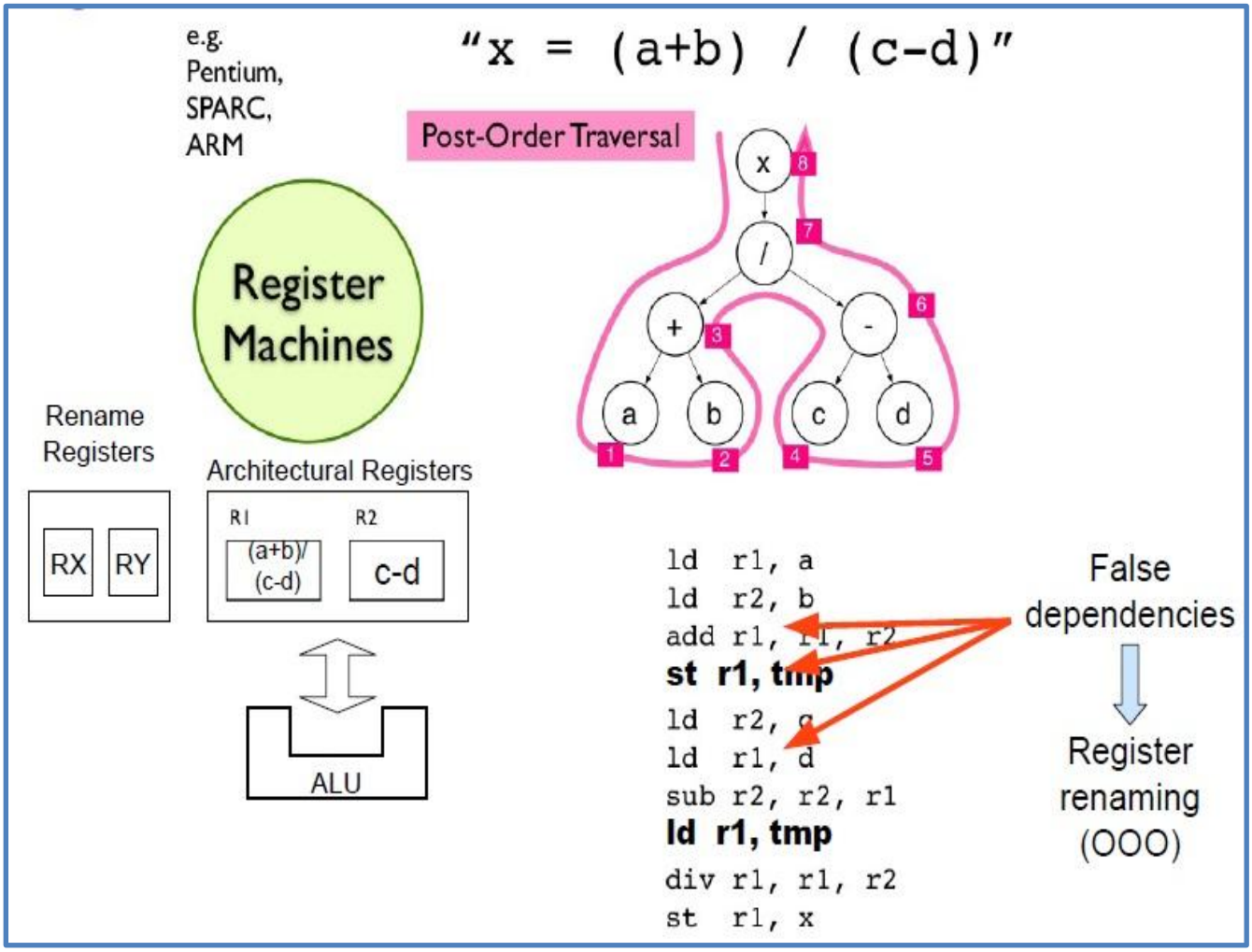


Fig 16 Summary of Register Machines

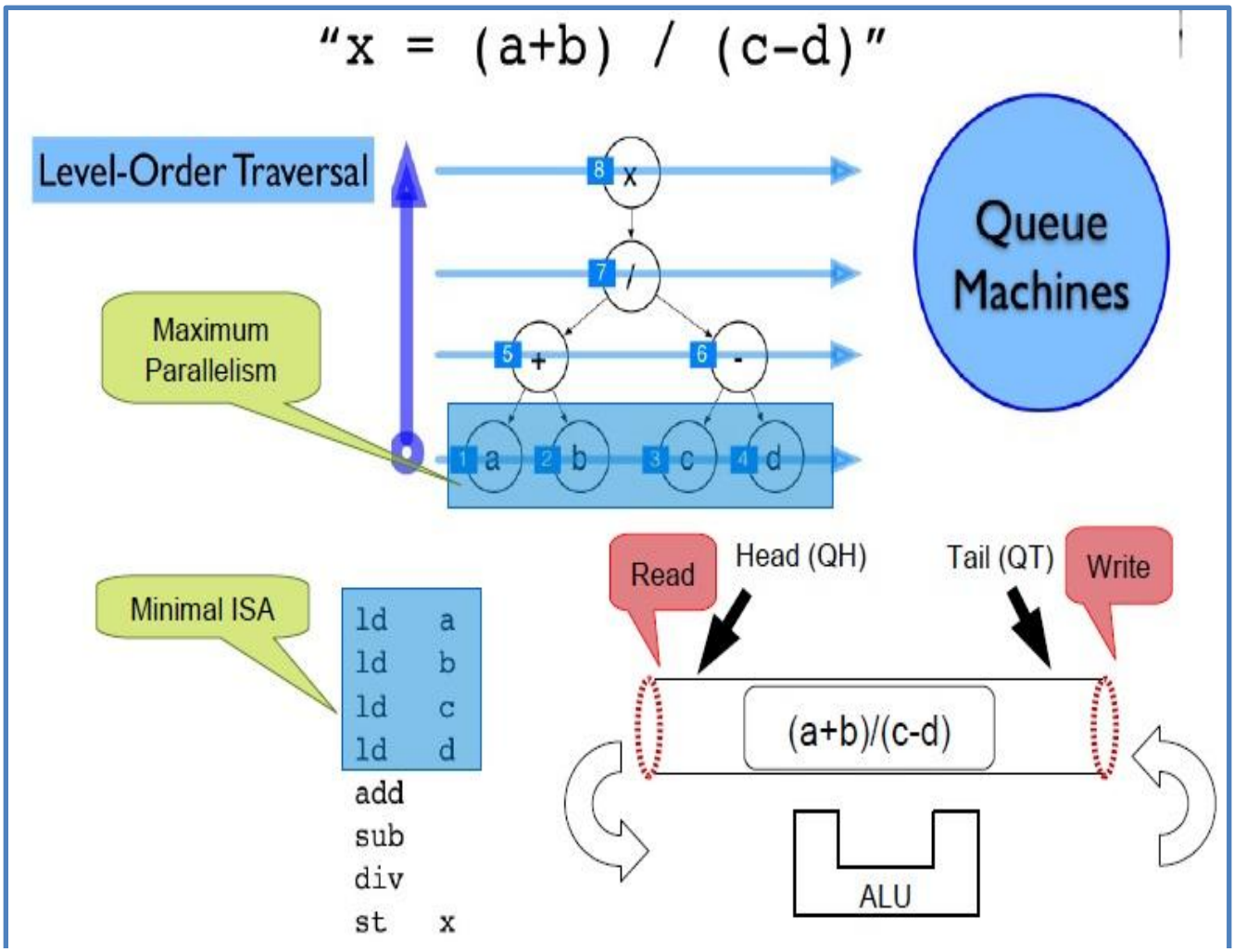


Fig 17 Summary of Queue Machines

The analysis above clearly demonstrates the advantage of the queue processor over its RISC counterpart with features like maximum parallelism, minimal ISA, level order traversal and requiring no register renaming owing to the absence of false dependencies.

To establish that these analysis are not simply a product of mere chance, statistics of the simulation and synthesis results are described in the following chapter.

CHAPTER 4

COMPLEXITY ANALYSIS

4.1 Code size

The table below shows the result of the code size analysis of various RISC cores as compared to the Queue core.

Benchmarks	**MIPS 16**	**ARM**	**X86**	**QC-2**
H.263	58.00	83.66	57.20	41.34
MPEG2	53.09	78.40	53.22	36.75
Susan	47.34	80.48	46.66	35.12
AES	51.27	86.67	44.62	34.93
Blowfish	54.59	86.38	57.45	45.49
FFT	58.09	100.74	46.27	36.77
Average	**53.73**	**86.06**	**50.90**	**38.40**

Table 1 Code Size Comparison

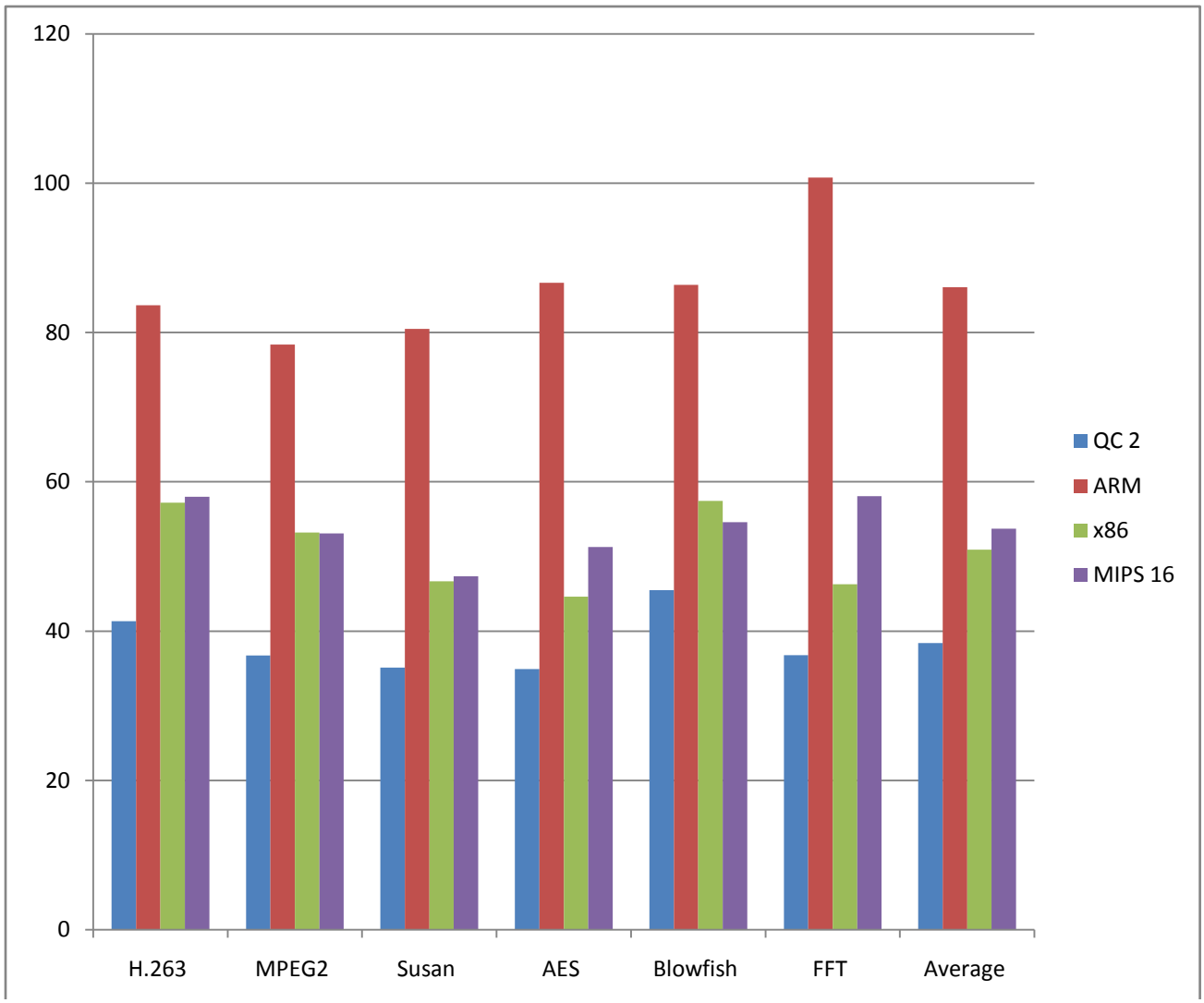


Chart 1 Bar Chart showing code size Comparison Results

4.2 Synthesis Result (Logical Elements)

Description	Modules	Logical Elements (LE)	Total Combinational Functions (TCF)
Instruction Fetch	IF	483	483
Instruction Decode	ID	150	150
Queue Compute Unit	QCU	70	60
Issue Unit	IS	5544	4008
Execution Unit	EXE	882	845
Memory Access	MEM	1241	729
QSoC	QSoC	8370	6275

Table 2 LE and TCF Results

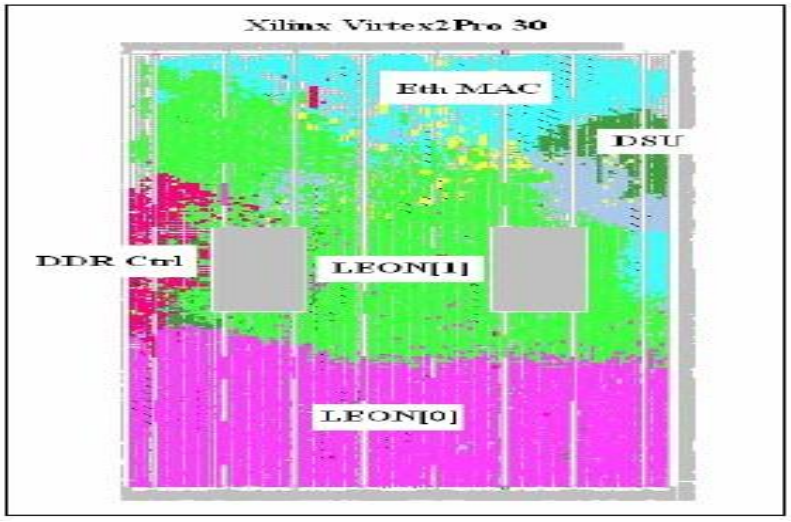


Fig 18 Floorplan of Leon Processor

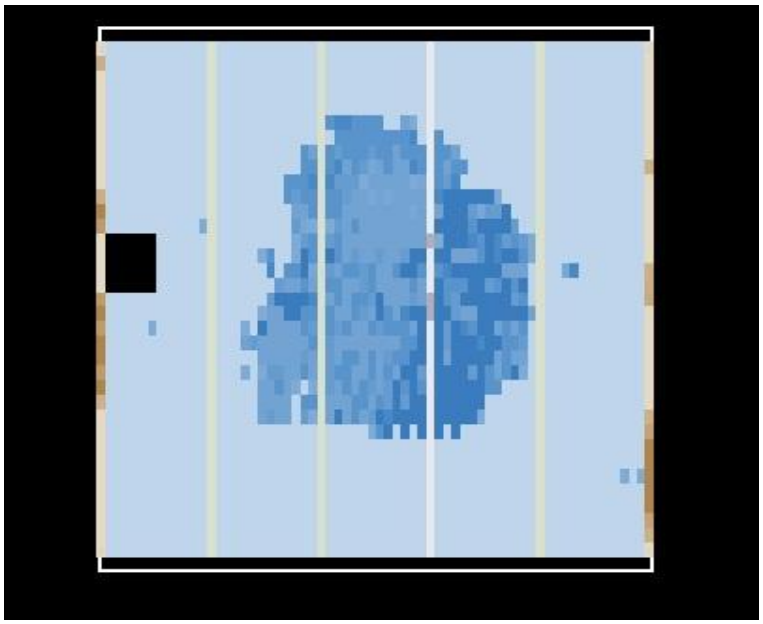


Fig 19 Floorplan of QSoC showing space occupied on the target FPGA device

4.3 Power and Speed Comparison Results

Cores	Speed (Spd)	Area (ARA)	Av. Power (mW)
PQP	22.5	21.5	120
SH-2	15.3	14.1	187.5
Leon2	27.5	26.7	458
Micro Blaze	26.7	26.7	135
QC -2	25.5	24.2	90
QSoC			**61.1**

Table 3 Speed and power consumption comparisons for various Synthesizable CPU cores over speed (SPD) and area (ARA) optimizations

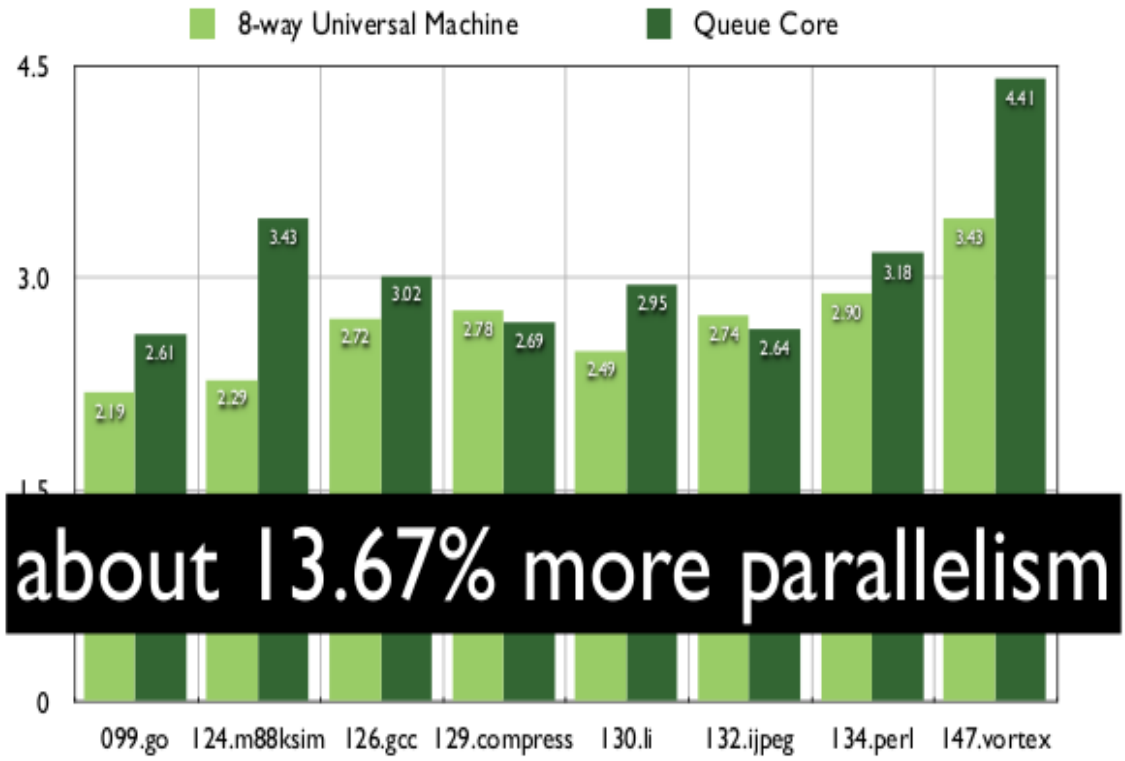


Chart 2 18 Parallelism Result

CHAPTER 5

DISCUSSION OF RESULTS

In QC-2 processor, all instructions designed are fixed format 16-bit words with minimal decoding effort. As a result, the QC-2 architecture has much smaller programs than either RISC or CISC machines. As shown in *Table 1,* programs sizes for the QC-2 architecture are found to be 50–70% smaller than programs for conventional architectures.

Table 2 summarizes the synthesis results of the QC-2 for the FPGA and HardCopy targets. The complexity of each module as well as the whole QC-2 core are given as the number of logic elements (LEs) for the FPGA device and as the total combinational functions (TCF) count for the HardCopy device (Structured ASIC). The design was optimized for balanced optimization guided by a properly implemented constraint table.

Performance of QC-2 in terms of speed and power consumption is compared with various synthesizable CPU cores as illustrated in Table 3.

From the result shown in Table 3, the QC-2 processor core shows better speed performance for both area and speed optimizations when compared with SH-2, PQP, and ARM7 (hard core) processors. The QC-2 has higher speed for both SPD and ARA optimizations when compared with SH-2 processor (about 40% for speed optimization and 41.73% for area optimization). QC-2 core also shows 25% less power consumption when compared with PQP and consumes less power than LEON2 and MicroBlaze processors.

On average the QC-2 has about 40.87% higher speed than SH-2 processor. QC-2 core also shows 25% less power consumption when compared with PQP, and consumes less power than SH-2, LEON2, and MicroBlaze cores.

CHAPTER 6

CONCLUSION

In this research work, the architecture, and features of the queue processor was evaluated alongside the RISC architecture. The theoretical analysis backed up by the synthesis results showed that the queue processor exhibits faster speed than its RISC counterpart.

Comparison with other synthesizable cores showed that on average the QC-2 has about 40.87% higher speed than SH-2 processor.

The lower Logical Elements (LE) accounts for its low memory usage, thus queue cores show less power consumption when compared with SH-2, LEON2, and MicroBlaze cores which use the RISC architecture.

The complete absence of the register renaming unit also accounts for its less power consumption. This unit is almost always present in RISC architectures and account for about 4% of the overall on-chip power consumption in RISC processors.

Since processor performance is measured by how much parallelism it can exploit, the queue processor stands out in this regard. This is as a result of the high parallelism results obtained with the QSoC due to its employing the Level Order Traversal Scheme.

Conclusively, these results show that the queue processor offers an interesting alternative to the design of embedded systems.

6.1 Future work

The result of this performance evaluation revealed a little problem with the queue machine. To get around this, future work which could proffer a solution to this has been outlined below:

- Eliminating the problem of the Queue machine with respect to DAG (Directed Acyclic Graph) expressions.
- Improving the Issue mechanism (like merging the Barrier and Issue stages)

REFERENCES

[01] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

[02] M. Fernandes, J. Llosa, N. Topham, *Using queues for register file organization in VLIW*, Technical Report ECS-CSG-29-97, Department of Computer Science, University of Edinburgh, 1997.

[03] B.A. Abderazek, *Dynamic instructions issue algorithm and a queue execution model toward the design of hybrid processor architecture*, Ph.D. Thesis, Graduate School of Information Systems, the University of Electro-Communications, 2002.

[04] Lieven Eeckhout , *Computer Architecture Performance Evaluation Methods*, Synthesis Lectures on Computer Architecture, June 2010, Vol. 5, No. 1, Pages 1-145

[05] Martti Forsell, *Implementation of Instruction-Level and Thread-Level Parallelism in Computers*, 1997.

[06] B.A. Abderazek, *Queue Machines: an unknown alternative* – PDCAT 2009

[07] B.A. Abderazek, M. Arsenji, S. Shigeta, T. Yoshinaga, M. Sowa, *Queue processor for novel queue computing paradigm based on produced order scheme*, in: HPC2004, International Conference on High Performance Computing, Tokyo, July 2004, pp. 169–177.

[08] B.A. Abderazek, S. Kawata, T. Yoshinaga, M. Sowa, *Modular Design Structure and High-Level Prototyping for Novel Embedded Processor Core*, in: EUC 2005, The 2005 IFIP International Conference on Embedded and Ubiquitous Computing, Nagasaki, Japan, December 6–9, 2005, pp. 340–349.

[09] B.A. Abderazek, T. Yoshinaga, M. Sowa, *High-level modeling and FPGA prototyping of produced order parallel queue processor core*, J. Supercomputing 38 (1) (2006) 3–15.s

[10] M. Sowa, B.A. Abderazek, T. Yoshinaga, *Parallel Queue processor architecture based on produced order computation model*, J. Supercomputing 32 (3) (2005) 217–229.

[11] B.A. Abderazek, T. Yoshinaga, A. Canedo M. Sowa, *The QC-2 parallel Queue processor architecture* J. Parallel Distributed Computing 68 (2008) 235– 245

[12] B. A. Abderazek, S. Shigeta, T. Yoshinaga, M. Sowa, *Reduced Bit-Width Instruction Set Architecture for Q-mode, Execution*, IPSJ Arch. Conf. pp. 19-23, June 2003

APPENDICES

APPENDIX A

VERILOG CODES

FOR TOP LEVEL MODULE (QP_top.v)

```
//-------------------------------------------------------------------
---------
// Title         : Top Module
// Project       : Queue Processor Implementation in CFS 2009
//-------------------------------------------------------------------
---------
// File          : QP_top.v
// Created       : 28.08.2009
// Last modified : 28.08.2009
//-------------------------------------------------------------------
---------

`define SIM

module QP_top(
`ifdef SIM
`else
              CLK,
              RESET
`endif
              );
`ifdef SIM
   reg CLK;
   reg RESET;
   reg IntReq;
   reg [31:0] IntAddress;

`else
```

```
   input CLK;
   input RESET;
`endif

   wire [31:0] Top_Address_wire;
   wire [31:0] Top_DataToCPU_wire, Top_DataFromCPU_wire;
   wire [1:0]  Top_ReadWrite_wire;

`ifdef SIM
   integer     i;
   initial
     begin
        // Instruction Memory, width=16bit, depth=1024
        $readmemh("../QASM/int_matrix_inst.hex",
cpu0.fu0.InstructionMemory);
      //$readmemh("SIM/mat_inst.dat", cpu0.fu0.InstructionMemory);
        // Data Memory, width=32bit, depth=1024
        $readmemh("../QASM/int_matrix_data.hex", mem0.dm0.DataMemory);
      //$readmemh("SIM/mat_data.dat", mem0.dm0.DataMemory);
       end // initial begin

   initial
     begin
        CLK = 1'b1;
        RESET = 1'b1;
      IntReq = 1'b0;
      IntAddress = 32'h00000000;
      #10;
        RESET = 1'b0;
        #10;
        RESET = 1'b1;

      #1000
        IntReq = 1'b1;
      //IntAddress = 32'h00000048;
```

```
      IntAddress = 32'h00000380;
      #1050
        IntReq = 1'b0;
      #100000;

        // QREG Contents
        $display("+------ QREG ------+");
        for(i=0;i<32;i=i+1)
          $display("QREG[%d]: %h", i, cpu0.iu0.QREG[i]);
        $display("+------ SPR ------+");
        for(i=0;i<16;i=i+1)
          $display("SPR[%d]: %h", i, cpu0.iu0.SPR[i]);

        // Display Memory Contents
        $display("+------ Inst Memory ------+");
        for(i=0;i<10;i=i+1)
          $display("IMEM[%d]: %h", i, cpu0.fu0.InstructionMemory[i]);
        $display("+------ Data Memory ------+");
        for(i=0;i<50;i=i+1)
          $display("DMEM[%d]: %h", i, mem0.dm0.DataMemory[i]);

      $display("IMEM[896]: %h", cpu0.fu0.InstructionMemory[896]);


        $finish;
      end // initial begin

    always #50 CLK = ~CLK;

    initial
      begin
       $dumpfile("tb_top.dsn");
       $dumpvars(0, QP_top);
      end
`endif
```

```
    QP_CPU cpu0(
                .CLK(CLK),
                .RESET(RESET),

                .QP_I_PeripheralData(Top_DataToCPU_wire),
                .QP_I_IntReq(IntReq),
                .QP_I_IntAddress(IntAddress),

                .QP_O_PeripheralData(Top_DataFromCPU_wire),
                .QP_O_PeripheralControl(Top_ReadWrite_wire),
                .QP_O_PeripheralAddress(Top_Address_wire)
                );

    PERI_MEM mem0(
                  .CLK(CLK), .RESET(RESET),

                  .MEM_I_Address(Top_Address_wire),
                  .MEM_I_Data(Top_DataFromCPU_wire),
                  .MEM_I_ReadWrite(Top_ReadWrite_wire),

                  .MEM_O_Data(Top_DataToCPU_wire)
                  );
endmodule // QP_top

`include "QP_CPU.v"
`include "PERI_MEM.v"
```

Verilog codes for Memory unit (QP_MU)

```
//------------------------------------------------------------------------
---------
// Title         : Memory Unit
// Project       : Queue Processor Implementation in CFS 2009
//------------------------------------------------------------------------
---------
// Modification history :
// 27.08.2009 : created
//------------------------------------------------------------------------
---------

module QP_MU(
             CLK, RESET,

             MU_I_Result,
             MU_I_WriteDataToMem,
             MU_I_DstAddress,
             MU_I_ControlMem,
             MU_I_ControlWB,
             MU_I_ReadDataFromPeripheral,

             MU_O_Result,
             MU_O_ReadDataFromMem,
             MU_O_DstAddress,
             MU_O_ControlWB,
             MU_O_ControlPeripheral,
             MU_O_PeripheralAddress,
             MU_O_WriteDataToPeripheral
             );
   parameter QPOINTER_WIDTH = 6;

   input CLK, RESET;
```

```
    input [31:0] MU_I_Result;
    input [31:0] MU_I_WriteDataToMem;
    input [QPOINTER_WIDTH-1:0] MU_I_DstAddress;
    input [1:0]                MU_I_ControlMem;
    input [1:0]                MU_I_ControlWB;
    input [31:0]               MU_I_ReadDataFromPeripheral;

    output [31:0]              MU_O_Result;
    output [31:0]              MU_O_ReadDataFromMem;
    output [QPOINTER_WIDTH-1:0] MU_O_DstAddress;
    output [1:0]                MU_O_ControlWB;
    output [1:0]                MU_O_ControlPeripheral;
    output [31:0]               MU_O_PeripheralAddress;
    output [31:0]               MU_O_WriteDataToPeripheral;

    // Output Generation
    assign MU_O_Result = MU_I_Result;
    assign MU_O_ReadDataFromMem = MU_I_ReadDataFromPeripheral;
    assign MU_O_DstAddress = MU_I_DstAddress;
    assign MU_O_ControlWB = MU_I_ControlWB;

    // To Peripheral
    assign MU_O_ControlPeripheral = MU_I_ControlMem;
    assign MU_O_PeripheralAddress = MU_I_Result;
    assign MU_O_WriteDataToPeripheral = MU_I_WriteDataToMem;

endmodule // QP_MU
```

Verilog codes for Queue Computation Unit (QP_QCU)

```
//-------------------------------------------------------------------
---------
// Title         : Queue Computation Unit
// Project       : Queue Processor Implementation in CFS 2009
//-------------------------------------------------------------------
---------
// Modification history :
// 27.08.2009 : created
//-------------------------------------------------------------------
---------

module QP_QCU(
              CLK, RESET,

              QCU_I_PN,
              QCU_I_CN,
              QCU_I_RegSel,
              QCU_I_RegNum,
              QCU_I_Operand,
              QCU_I_ControlExe,
              QCU_I_ExeOp,
              QCU_I_ControlMem,
              QCU_I_ControlWB,
              QCU_I_PC,
              QCU_I_Branch,
              QCU_I_RenewQH,
              QCU_I_RenewQT,

              QCU_O_Src1Address,
              QCU_O_Src2Address,
              QCU_O_DstAddress,
              QCU_O_Operand,
              QCU_O_ControlExe,
```

```
            QCU_O_ExeOp,
            QCU_O_ControlMem,
            QCU_O_ControlWB,
            QCU_O_PC,
            //===== for interrupt =====
            QCU_I_IntReq,
            QCU_I_IntEnable,
            QCU_I_RFI
            );
parameter QPOINTER_WIDTH = 6;

input CLK, RESET;

input QCU_I_PN;
input [1:0] QCU_I_CN;
input [2:0] QCU_I_RegSel;
input [7:0] QCU_I_RegNum;
input [7:0] QCU_I_Operand;
input [4:0] QCU_I_ControlExe;
input [3:0] QCU_I_ExeOp;
input [1:0] QCU_I_ControlMem;
input [1:0] QCU_I_ControlWB;
input [31:0] QCU_I_PC;
input        QCU_I_Branch;
input [QPOINTER_WIDTH-2:0] QCU_I_RenewQH,
                           QCU_I_RenewQT;

//===== for interrupt =====
input                      QCU_I_IntReq,
                           QCU_I_IntEnable,
                           QCU_I_RFI;

output [QPOINTER_WIDTH-1:0] QCU_O_Src1Address,
                            QCU_O_Src2Address,
                            QCU_O_DstAddress;
```

```
    output [7:0]                QCU_O_Operand;
    output [4:0]                QCU_O_ControlExe;
    output [3:0]                QCU_O_ExeOp;
    output [1:0]                QCU_O_ControlMem;
    output [1:0]                QCU_O_ControlWB;
    output [31:0]               QCU_O_PC;

    //===== register =====
    //===== for interrupt =====
    reg [QPOINTER_WIDTH-2:0]    QCU_IQHR_reg,
                                QCU_IQTR_reg;

    wire [QPOINTER_WIDTH-2:0]   QCU_Src1Address_wire,
                                QCU_Src2Address_wire,
                                QCU_DstAddress_wire;
    wire [QPOINTER_WIDTH-6:0]   QCU_ZeroSpace_wire;

    assign QCU_ZeroSpace_wire = 0;

    // QREG Pointer Register
    reg [QPOINTER_WIDTH-2:0]    QCU_QH_reg, QCU_QT_reg;

    // Output Generation
    assign    QCU_O_Src1Address    =    QCU_I_RegSel[2]    ?    {1'b1,
QCU_ZeroSpace_wire, QCU_I_RegNum[3:0]} : {1'b0, QCU_Src1Address_wire};
    assign    QCU_O_Src2Address    =    QCU_I_RegSel[1]    ?    {1'b1,
QCU_ZeroSpace_wire, QCU_I_RegNum[3:0]} : {1'b0, QCU_Src2Address_wire};
    assign    QCU_O_DstAddress    =    QCU_I_RegSel[0]    ?    {1'b1,
QCU_ZeroSpace_wire, QCU_I_RegNum[7:4]} : {1'b0, QCU_DstAddress_wire};
    assign QCU_O_Operand = QCU_I_Operand;
    assign QCU_O_ControlExe = QCU_I_ControlExe;
    assign QCU_O_ExeOp = QCU_I_ExeOp;
    assign QCU_O_ControlMem = QCU_I_ControlMem;
    assign QCU_O_ControlWB = QCU_I_ControlWB;
    assign QCU_O_PC = QCU_I_PC;
```

```
   //==== QREG Pointer Calculation ====//
   // Src1
   assign QCU_Src1Address_wire = QCU_QH_reg;
   // Src2, case 1 is for store instruction write data
   assign  QCU_Src2Address_wire  =  QCU_I_RegSel[2]  ?  QCU_QH_reg  :
((|QCU_I_Operand) ? QCU_QH_reg + QCU_I_Operand : QCU_QH_reg + 1);
   // Dst
   assign QCU_DstAddress_wire = QCU_QT_reg;

   //==== Renew QREG Pointer ====//
   always @(posedge CLK or negedge RESET)
     begin
        if(!RESET)
          begin
             QCU_QH_reg <= 0;
             QCU_QT_reg <= 0;
          end
        else
          begin
             //===== for interrupt =====
             if((QCU_I_IntReq == 1'b1) && (QCU_I_IntEnable == 1'b1))
               begin
                  QCU_IQHR_reg <= QCU_QH_reg;
                  QCU_IQTR_reg <= QCU_QT_reg;
                  QCU_QH_reg <= 0;
                  QCU_QT_reg <= 0;
               end
             else if(QCU_I_RFI == 1'b1)
               begin
                  QCU_QH_reg <= QCU_IQHR_reg;
                  QCU_QT_reg <= QCU_IQTR_reg;
               end
             else
               begin
```

```
                    if(QCU_I_Branch)
                      begin
                         QCU_QH_reg <= QCU_I_RenewQH;
                         QCU_QT_reg <= QCU_I_RenewQT;
                      end
                    else
                      begin
                         QCU_QH_reg <= QCU_QH_reg + QCU_I_CN;
                         QCU_QT_reg <= QCU_QT_reg + QCU_I_PN;
                      end
                end // else: !if(QCU_I_IntReq)
            end // else: !if(!RESET)
        end // always @ (posedge CLK or negedge RESET)

endmodule // QP_QCU
```

Verilog codes for Write Back Unit (QP_QCU)

```
//------------------------------------------------------------------------
---------
// Title         : Write Back Unit
// Project       : Queue Processor Implementation in CFS 2009
//------------------------------------------------------------------------
---------
// Modification history :
// 27.08.2009 : created
//------------------------------------------------------------------------
---------

module QP_WBU(
              CLK, RESET,

              WBU_I_Result,
              WBU_I_ReadDataFromMem,
              WBU_I_DstAddress,
              WBU_I_ControlWB,

              WBU_O_WriteDataToReg,
              WBU_O_DstAddress,
              WBU_O_RegWrite
              );
   parameter QPOINTER_WIDTH = 6;

   input CLK, RESET;

   input [31:0] WBU_I_Result;
   input [31:0] WBU_I_ReadDataFromMem;
   input [QPOINTER_WIDTH-1:0] WBU_I_DstAddress;
   input [1:0]                WBU_I_ControlWB;

   output [31:0]              WBU_O_WriteDataToReg;
```

```
    output [QPOINTER_WIDTH-1:0] WBU_O_DstAddress;
    output                      WBU_O_RegWrite;

    wire                        WBU_DataSel_wire, WBU_RegWrite_wire;

    // Control Signals
    assign {WBU_DataSel_wire, WBU_RegWrite_wire} = WBU_I_ControlWB;

    // Output Generation
    assign      WBU_O_WriteDataToReg      =      WBU_DataSel_wire      ?
WBU_I_ReadDataFromMem : WBU_I_Result;
    assign WBU_O_DstAddress = WBU_I_DstAddress;
    assign WBU_O_RegWrite = WBU_RegWrite_wire;

endmodule // QP_WBU
```

Verilog Codes for Execution Unit

```
//-------------------------------------------------------------------
---------
// Title         : Execution Unit
// Project       : Queue Processor Implementation on CFS 2009
//-------------------------------------------------------------------
----------
// Modification history :
// 25.08.2009 : created
//-------------------------------------------------------------------
---------

module QP_EU(
             CLK, RESET,

             EU_I_Src1Address,
             EU_I_Src1,
             EU_I_Src2,
             EU_I_DstAddress,
             EU_I_Operand,
             EU_I_ControlExe,
             EU_I_ExeOp,
             EU_I_PC,
             EU_I_ControlMem,
             EU_I_ControlWB,

             EU_O_Result,
             EU_O_WriteDataToMem,
             EU_O_DstAddress,
             EU_O_ControlMem,
             EU_O_ControlWB,
             EU_O_BranchAddress,
             EU_O_Branch,
             EU_O_RenewQH,
```

```
          EU_O_RenewQT,
          //===== for interrupt =====
          EU_I_IntReq,
          EU_I_IntEnable,
          EU_I_RFI
          );
parameter QPOINTER_WIDTH = 6;

input CLK, RESET;

input [QPOINTER_WIDTH-1:0] EU_I_Src1Address;
input [31:0]               EU_I_Src1, EU_I_Src2;
input [QPOINTER_WIDTH-1:0] EU_I_DstAddress;
input [7:0]                EU_I_Operand;
input [4:0]                EU_I_ControlExe;
input [3:0]                EU_I_ExeOp;
input [31:0]               EU_I_PC;
input [1:0]                EU_I_ControlMem;
input [1:0]                EU_I_ControlWB;

//===== for interrupt =====
input                      EU_I_IntReq,
                           EU_I_IntEnable,
                           EU_I_RFI;

output [31:0]              EU_O_Result;
output [31:0]              EU_O_WriteDataToMem;
output [QPOINTER_WIDTH-1:0] EU_O_DstAddress;
output [1:0]                EU_O_ControlMem;
output [1:0]                EU_O_ControlWB;
output [31:0]               EU_O_BranchAddress;
output                      EU_O_Branch;
output [QPOINTER_WIDTH-2:0] EU_O_RenewQH, EU_O_RenewQT;

reg [31:0]                  EU_O_Result;
```

```
   reg [31:0]                  EU_SetResult_reg;
   reg [QPOINTER_WIDTH-1:0]    EU_SPRAddress_reg;
   reg [31:0]                  EU_SPRReg_reg;
   reg [1:0]                   EU_CC_reg;

   //===== for interrupt =====
   reg [QPOINTER_WIDTH-1:0]    EU_ISPRAddress_reg;
   reg [31:0]                  EU_ISPRReg_reg;
   reg [1:0]                   EU_ICC_reg;

   wire                                EU_AG_wire,  EU_Branch_wire,
EU_Immediate_wire;
   wire [1:0]                  EU_ResultSel_wire;
   wire [31:0]                 EU_ExtendedOperand_wire;
   wire [31:0]                 EU_SelectedSrc1_wire;
   wire [31:0]                 EU_ALUSrc1_wire, EU_ALUSrc2_wire;
   wire [3:0]                  EU_FuncSel_wire;
   wire                        EU_CCWrite_wire;
   wire [31:0]                 EU_ALUResult_wire;
   wire                        EU_ALUZero_wire;
   wire [31:0]                 EU_MultResult_wire;


   // Output Generation
   assign EU_O_WriteDataToMem = EU_I_Src2;
   assign EU_O_DstAddress = EU_I_DstAddress;
   assign EU_O_RenewQH = EU_I_Src1Address[QPOINTER_WIDTH-2:0];
   assign EU_O_RenewQT = EU_I_DstAddress[QPOINTER_WIDTH-2:0];
   assign EU_O_ControlMem = EU_I_ControlMem;
   assign EU_O_ControlWB = EU_I_ControlWB;
   always @(*)
     begin
        case (EU_ResultSel_wire)
          2'b00: EU_O_Result <= EU_ALUResult_wire;
          2'b01: EU_O_Result <= EU_SetResult_reg;
```

```
          2'b10: EU_O_Result <= EU_MultResult_wire;
          default: EU_O_Result <= EU_ALUResult_wire;
        endcase // case (EU_ResultSel_wire)
     end


   // Execution Unit Control Signals
   assign     {EU_AG_wire,     EU_Branch_wire,     EU_Immediate_wire,
EU_ResultSel_wire} = EU_I_ControlExe;

   // Sign Extention
   assign  EU_ExtendedOperand_wire  =  EU_I_Operand[7]  ?  {24'hffffff,
EU_I_Operand} : {24'b0, EU_I_Operand};


   // ==== Set Type ==== //
   assign     EU_SelectedSrc1_wire     =     (EU_SPRAddress_reg     ==
EU_I_Src1Address) ? EU_SPRReg_reg : EU_I_Src1;
   always @(*)
     begin
        case (EU_I_ExeOp)
          3'h0:        EU_SetResult_reg        <=        {EU_I_Operand,
EU_SelectedSrc1_wire[23:0]};
          3'h1:   EU_SetResult_reg   <=   {EU_SelectedSrc1_wire[31:24],
EU_I_Operand, EU_SelectedSrc1_wire[15:0]};
          3'h2:   EU_SetResult_reg   <=   {EU_SelectedSrc1_wire[31:16],
EU_I_Operand, EU_SelectedSrc1_wire[7:0]};
          3'h3:   EU_SetResult_reg   <=   {EU_SelectedSrc1_wire[31:8],
EU_I_Operand};
          3'h4: EU_SetResult_reg <= EU_SelectedSrc1_wire;
          3'h5: EU_SetResult_reg <= EU_ExtendedOperand_wire;
          default:  EU_SetResult_reg <= 32'b0;
        endcase // case (EU_I_ExeOp)
     end // always @ (*)
   always @(posedge CLK or negedge RESET)
```

```
begin
   if(!RESET)
     begin
        EU_SPRAddress_reg[QPOINTER_WIDTH-1] <= 1'b1;
        EU_SPRAddress_reg[QPOINTER_WIDTH-2:0] <= 0;
        EU_SPRReg_reg <= 32'b0;
     end
   else
     begin
        //===== for interrupt =====
        if((EU_I_IntReq == 1'b1) && (EU_I_IntEnable == 1'b1))
          begin
             EU_ISPRAddress_reg <= EU_SPRAddress_reg;
             EU_ISPRReg_reg <= EU_SPRReg_reg;
             EU_SPRAddress_reg[QPOINTER_WIDTH-1] <= 1'b1;
             EU_SPRAddress_reg[QPOINTER_WIDTH-2:0] <= 0;
             EU_SPRReg_reg <= 32'b0;
          end
        else if(EU_I_RFI == 1'b1)
          begin
             EU_SPRAddress_reg <= EU_ISPRAddress_reg;
             EU_SPRReg_reg <= EU_ISPRReg_reg;
          end
        else
          begin
             if(EU_ResultSel_wire[0] & (EU_I_ExeOp != 4'h5))
               begin
                  EU_SPRAddress_reg <= EU_I_Src1Address;
                  EU_SPRReg_reg <= EU_SetResult_reg;
               end
             else
               begin
                  EU_SPRAddress_reg <= EU_SPRAddress_reg;
                  EU_SPRReg_reg <= EU_SPRReg_reg;
               end
```

```
              end   //   else:   !if((EU_I_Int_Req   ==   1'b1)   &&
(EU_I_Int_Status == 1'b0))
          end
     end // always @ (posedge CLK or negedge RESET)


   // ==== ALU Type ==== //
   assign EU_ALUSrc1_wire = EU_SelectedSrc1_wire;

   assign   EU_ALUSrc2_wire   =   EU_Immediate_wire   ?   (EU_AG_wire   ?
EU_ExtendedOperand_wire : EU_ExtendedOperand_wire) : EU_I_Src2;
   // If 8bit-width-memory is used, the following line is used insted
of above one.
   //  assign  EU_ALUSrc2_wire  =  EU_Immediate_wire  ?  (EU_AG_wire  ?
{EU_ExtendedOperand_wire[29:0],  2'b0}  :  EU_ExtendedOperand_wire)  :
EU_I_Src2;

   ALUControler aluCont0(
                        .ALUControler_I_ExeOp(EU_I_ExeOp),
                        .ALUControler_O_FuncSel(EU_FuncSel_wire),
                        .ALUControler_O_CCWrite(EU_CCWrite_wire)
                        );
   ALU alu0(
            .ALU_I_Src1(EU_ALUSrc1_wire),
            .ALU_I_Src2(EU_ALUSrc2_wire),
            .ALU_I_FuncSel(EU_FuncSel_wire),
            .ALU_O_Result(EU_ALUResult_wire),
            .ALU_O_Zero(EU_ALUZero_wire)
            );
   always @(posedge CLK or negedge RESET)
     begin
        if(!RESET)
          begin
             EU_CC_reg <= 2'b00;
          end
```

```
        else
          begin
             if((EU_I_IntReq == 1'b1) && (EU_I_IntEnable == 1'b1))
               begin
                  EU_ICC_reg <= EU_CC_reg;
                  EU_CC_reg <= 2'b00;
               end
             else if(EU_I_RFI == 1'b1)
               EU_CC_reg <= EU_ICC_reg;
             else
               begin
                  if(EU_CCWrite_wire & (EU_ResultSel_wire == 2'b00))
                    begin
                       EU_CC_reg[1] <= EU_ALUResult_wire[31];
                       EU_CC_reg[0] <= EU_ALUZero_wire;
                    end
                  else
                    begin
                       EU_CC_reg <= EU_CC_reg;
                    end
               end   //   else:   !if((EU_I_Int_Req   ==   1'b1)   &&
(EU_I_Int_Status == 1'b0))
          end // else: !if(!RESET)
     end // always @ (posedge CLK or negedge RESET)


   // ==== Mult Type ==== //
   Multiplyer mult0(
                  .Multiplyer_I_Src1(EU_I_Src1),
                  .Multiplyer_I_Src2(EU_I_Src2),
                  .Multiplyer_O_Result(EU_MultResult_wire)
                  );


   // ==== Branch Type ==== //
```

```
   assign EU_O_BranchAddress = EU_I_PC + EU_ExtendedOperand_wire;
   // If instruction memory is 8-bit-width memory, following line is
used
   //      assign      EU_O_BranchAddress      =      EU_I_PC      +
{EU_ExtendedOperand_wire[30:0], 1'b0};
   BranchConditionCheck bcc(
                            .BranchConditionCheck_I_CC(EU_CC_reg),
                            .BranchConditionCheck_I_ExeOp(EU_I_ExeOp),

.BranchConditionCheck_I_Branch(EU_Branch_wire),
                            .BranchConditionCheck_O_PCSel(EU_O_Branch)
                            );

endmodule // QP_EU

module ALUControler(
                    ALUControler_I_ExeOp,
                    ALUControler_O_FuncSel,
                    ALUControler_O_CCWrite
                    );
   input [3:0] ALUControler_I_ExeOp;
   output [3:0] ALUControler_O_FuncSel;
   output       ALUControler_O_CCWrite;

   reg [3:0]    ALUControler_O_FuncSel;
   reg          ALUControler_O_CCWrite;

   always @(*)
     begin
        case (ALUControler_I_ExeOp)
          4'b0000:
            begin
               ALUControler_O_FuncSel <= 4'b0000;
               ALUControler_O_CCWrite <= 1'b0;
            end
```

```
4'b0001:
  begin
     ALUControler_O_FuncSel <= 4'b0001;
     ALUControler_O_CCWrite <= 1'b0;
  end
4'b0010:
  begin
     ALUControler_O_FuncSel <= 4'b0010;
     ALUControler_O_CCWrite <= 1'b0;
  end
4'b0011:
  begin
     ALUControler_O_FuncSel <= 4'b0011;
     ALUControler_O_CCWrite <= 1'b0;
  end
4'b0100:
  begin
     ALUControler_O_FuncSel <= 4'b0001;
     ALUControler_O_CCWrite <= 1'b1;
  end
4'b0101:
  begin
     ALUControler_O_FuncSel <= 4'b1000;
     ALUControler_O_CCWrite <= 1'b0;
  end
4'b0110:
  begin
     ALUControler_O_FuncSel <= 4'b1010;
     ALUControler_O_CCWrite <= 1'b0;
  end
4'b0111:
  begin
     ALUControler_O_FuncSel <= 4'b1011;
     ALUControler_O_CCWrite <= 1'b0;
  end
```

```
          4'b1000:
            begin
               ALUControler_O_FuncSel <= 4'b0100;
               ALUControler_O_CCWrite <= 1'b0;
            end
          default:
            begin
               ALUControler_O_FuncSel <= 4'b0000;
               ALUControler_O_CCWrite <= 1'b0;
            end
        endcase // case (ALUControler_I_ExeOp)
     end // always @ (*)
endmodule // ALUControler

module ALU (
            ALU_I_Src1,
            ALU_I_Src2,
            ALU_I_FuncSel,
            ALU_O_Result,
            ALU_O_Zero
            );

   input [31:0] ALU_I_Src1, ALU_I_Src2;
   input [3:0]  ALU_I_FuncSel;

   output [31:0] ALU_O_Result;
   output        ALU_O_Zero;

   reg [31:0] ALU_O_Result;

   assign ALU_O_Zero = ~|ALU_O_Result;

   always @(*)
     begin
        case (ALU_I_FuncSel)
```

```
            4'b0000: ALU_O_Result <= ALU_I_Src1 + ALU_I_Src2;
            4'b0001: ALU_O_Result <= ALU_I_Src1 - ALU_I_Src2;
            4'b0010: ALU_O_Result <= ALU_I_Src1 | ALU_I_Src2;
            4'b0011: ALU_O_Result <= ALU_I_Src1 & ALU_I_Src2;
            4'b0100: ALU_O_Result <= ~ALU_I_Src1;
            4'b1000: ALU_O_Result <= ALU_I_Src1 << ALU_I_Src2;
            4'b1010: ALU_O_Result <= ALU_I_Src1 >> ALU_I_Src2;
            4'b1011: ALU_O_Result <= ALU_I_Src1 >> ALU_I_Src2;
            default: ALU_O_Result <= 32'b0;
         endcase // case (ALU_I_FuncSel)
      end // always @ (*)

endmodule // ALU

module Multiplyer(
                  Multiplyer_I_Src1,
                  Multiplyer_I_Src2,
                  Multiplyer_O_Result
                  );
   input [31:0] Multiplyer_I_Src1, Multiplyer_I_Src2;
   output [31:0] Multiplyer_O_Result;

   //====== modification =====
   //wire signed [31:0] Multiplyer_I_Src1, Multiplyer_I_Src2;

   assign Multiplyer_O_Result = Multiplyer_I_Src1 * Multiplyer_I_Src2;

endmodule

module BranchConditionCheck(
                            BranchConditionCheck_I_CC,
                            BranchConditionCheck_I_ExeOp,
                            BranchConditionCheck_I_Branch,
                            BranchConditionCheck_O_PCSel
                            );
```

```
   input [1:0]  BranchConditionCheck_I_CC;
   input [3:0]  BranchConditionCheck_I_ExeOp;
   input        BranchConditionCheck_I_Branch;
   output       BranchConditionCheck_O_PCSel;
   reg          BranchConditionCheck_O_PCSel;

   wire         BranchConditionCheck_Condition1bit_wire;
   wire [1:0]   BranchConditionCheck_Condition2bit_wire;

   assign           BranchConditionCheck_Condition1bit_wire           =
BranchConditionCheck_I_ExeOp[1];
   assign           BranchConditionCheck_Condition2bit_wire           =
BranchConditionCheck_I_ExeOp[0] ? 2'b10 : 2'b00;

   always @(*)
     begin
        if(BranchConditionCheck_I_Branch)
          if(BranchConditionCheck_I_ExeOp[2])
            if(BranchConditionCheck_I_CC                              ==
BranchConditionCheck_Condition2bit_wire)
              BranchConditionCheck_O_PCSel <= 1'b1;
            else
              BranchConditionCheck_O_PCSel <= 1'b0;
          else

if(BranchConditionCheck_I_CC[BranchConditionCheck_I_ExeOp[0]]        ==
BranchConditionCheck_Condition1bit_wire)
              BranchConditionCheck_O_PCSel <= 1'b1;
            else
              BranchConditionCheck_O_PCSel <= 1'b0;
        else
          BranchConditionCheck_O_PCSel <= 1'b0;
     end // always @ (*)

endmodule
```

APPENDIX B

1. Analysis and Synthesis Summary Report

Analysis & Synthesis Status	Successful - Thu Sep 16 11:44:46 2010
Quartus II Version	9.0 Build 235 06/17/2009 SP 2 SJ Web Edition
Revision Name	QP32b_system
Top-level Entity Name	QP32b_system
Family	Cyclone II
Total logic elements	8,456
Total combinational functions	5,811
Dedicated logic registers	4,373
Total registers	4373
Total pins	106
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	6
Total PLLs	0