ANALYSIS OF HARDWARE SORTING UNITS IN PROCESSOR DESIGN

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Carmelo Furlan

June 2019

COMMITTEE MEMBERSHIP

TITLE:	Analysis of Hardware Sorting Units in Processor Design
AUTHOR:	Carmelo Furlan
DATE SUBMITTED:	June 2019
COMMITTEE CHAIR:	Andrew Danowitz, Ph.D. Assistant Professor of Electrical Engineering
COMMITTEE MEMBER:	Bridget Benson, Ph.D. Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Joseph Callenes-Sloan, Ph.D. Assistant Professor of Electrical Engineering

ABSTRACT

Analysis of Hardware Sorting Units in Processor Design

Carmelo Furlan

Sorting is often computationally intensive and can cause the application in which it is used to run slowly. To date, the quickest software sorting implementations for an N element sorting problem runs at $O(nlogn)$. Current techniques, beyond developing better algorithms, used to accelerate sorting include the use of multiple processors or moving the sorting operation to a GPU. The use of multiple processors or a GPU can lead to increased energy consumption and heat produced by the device as compared to a single-core GPU-less implementation. To address these problems, specialized instructions and hardware units can be added to the processors to accelerate the sorting operation directly. This thesis studies and records the performance implications from implementing a sorting accelerator into a modern RISC-V processor pipeline. This thesis also explores the additional energy and area costs of implementing such hardware units in the processor.

ACKNOWLEDGMENTS

Thanks to:

- My advisor, Professor Andrew Danowitz for his guidance, advice, and time throughout this project.
- My other committee members, Professor Bridget Benson and Professor Joseph Callenes-Sloan, for their expertise and time.
- My parents for all their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
INTRODUCTION

Sorting is an important function in a number of applications, including pattern recognition [20], image processing [19], multi-media processing (video, audio, etc.) [14], signal processing [14], and high-energy physics [16]. Sorting is computationally intensive with the fastest software implementations running at $O(nlogn)$. Current solutions to speed up the sorting process include moving the sorting algorithm to multiple processors or onto a GPU [31]. The use of a GPU or multiple processors can cause an increase in the energy consumption and the heat produced by a device. Along with the energy and heat caused by GPUs and multiple processors, users often need to refactor the code for sorting applications in such a way that allows the compiler to efficiently divide sections of the code between the multiple processors or GPU.

A hardware sorting unit has the potential to significantly improve the run time of sorting without the need of additional cores or a GPU. This thesis created a hardware sorting unit which was added into a RISC-V processor pipeline. A specialized vector instruction was added to the instruction set which uses the hardware sorting unit to perform sort operations. This ”SIMD” implementation decision promises lower energy and higher performance for sorting operations, without the need for significant code-refactoring.

This thesis produced a hardware sorting unit that is capable of outperforming the current sorting algorithms on a modern processor. This thesis also discusses how the hardware was implemented into a RISC-V processor core and ISA. Using a commercial Xilinx FPGA-based test platform [3], we document the performance, area, and energy implications of the hardware sorting unit running a sample application. The final

results show that the hardware sort engine is able to obtain a speedup of 17x and an energy reduction of 93% on a 64x64 median filter operation.

Chapter 2

BACKGROUND

2.1 Related Work

The work in this thesis uses several previous published papers as building blocks to develop the final hardware sorting unit and specialized sort instructions. The paper *Modular High-throughput and Low-latency Sorting Units for FPGAs in the Large Hadron Collider* describes the techniques for designing high-throughput, low-latency sorting units for FPGAs [16]. In the paper the researchers discuss the creation of the hardware sorting algorithms. The main design of the paper's sorting units are based on the widely used Batcher's bitonic sorting algorithm described in the paper *Sorting Networks and Their Applications* [8].

A *bitonic* sequence is the juxtaposition of two monotonic sequences in which one is an ascending monotonic sequence and the other is a decreasing monotonic sequence [8]. A monotonic sequence is one in which the numbers in a sequence are constantly increasing ($a_n \leq a_{n+1}$) or decreasing ($a_n \geq a_{n+1}$). A bitonic sequence can be made by arranging any two monotonic sequences in such a way that the combination of the two monotonic sequences is either first increasing then decreasing or first decreasing then increasing. Since any two monotonic sequences may be used to create a bitonic sequence an algorithm can be created that rearranges a bitonic sequence into monotonic order. This algorithm can then be used as a merging network. The iterative nature of bitonic sorters allows for large bitonic sorters to be made of smaller bitonic sorters with the smallest bitonic sorter (2 number sorter) simply being a comparison element. The bitonic sorting network recursively merges the monotonic sequences of length N/2 to form a bitonic sequence of length N.

The *Modular High-throughput and Low-latency Sorting Units for FPGAs in the Large Hadron Collider* paper discusses the creation of modular sorting units that return only the M largest values (in no particular order) in a list of N elements [16]. The paper first describes the process of creating an 8-4 Max Selection Unit in which the four largest elements of a list of 8 are returned. In order to use these 8-4 Max Selection Units as building blocks for larger units their outputs need to be put in either ascending or descending order. The 8-4 Units with ordered outputs can then be used to create even larger Max Selection Units by using a combination of 8-4 Units to obtain the desired number of elements to return.

Max Selection Units of various sizes ranging from 16-to-4 to 256-to-4 were synthesized on a Virtex-5 FPGA device to obtain hardware resource requirements, overall latency, and the maximum frequency of the different units. The paper compared the results of the Max Selection Units to results gathered when using a full sorting network and proceeded to take the number of max values desired from that full network. Although this is not the exact functionality we are trying to obtain in this thesis their concepts can be adapted to fully sort a list. The results show that their approach beat the full network not only in the latency for results, but also in terms of resources required to perform the computations. The work in the paper by produces a top-M sorter but by adding additional hardware the ideas used to create a top-M sorter can be used to create a full sorter , which is what is desired for this thesis.

In the paper *A Comparison of Three Representative Hardware Sorting Units* three different types of sorting units are designed and then tested against each other to find the most efficient hardware sorting unit [24]. The three types of hardware sorting units designed and tested in the paper are an Insertion Sorting Unit, a FIFO-based Merge Sorting Unit, and a Sorting Network Unit.

The Insertion Sorting Unit is composed of an array of comparison/insert cells. These comparison/insert cells are created using comparators, multiplexers, registers to hold data, and control logic. The array contains the same number of comparison/insert cells as the number of elements that need to be sorted. The unit also contains control tags that drive the control logic of the cells to find the location where the new data needs to be placed [26]. When a new element needs to be sorted it is broadcast to all the cells and the comparison takes place to find the correct location in which the new element needs to be placed. This makes the computational complexity of the sorting unit $O(N)$.

The Sorting Network Unit proposed in this paper is based on the traditional approach to sorting networks, but, attempts to reduce the hardware cost that most sorting network implementations have. The sorting network is constructed by iteratively using a single row in the sorting network algorithm to create a one level pipeline where the hardware is reused every clock cycle. This means that all the computing stages of the sorting network use the same hardware, resulting in a need for significantly fewer hardware resources. Using this solution, the system only needs X comparator-swap components and a simple switch network as compared to $Xlog^2(X)$ comparator-swap components required by most hardware sorting networks, where X is equal to the size of the sorting unit [32]. The system works by switching a pair of elements every clock cycle until all the elements are sorted. Although, the sorting network is implemented using an iterative approach the computational complexity is $O(N)$.

The final sorting unit proposed in the paper is a FIFO Based Merge Sorting Unit. This sorting unit consists of two input FIFO queues where the merge is implemented by presenting the data from the two input FIFOs to a comparator-multiplexer block

[27]. The comparator-multiplexer block indicates which element is greater and that element is then written into the output FIFO. The FIFO Based Merge Sorting Unit requires much less hardware resources and has a computational complexity of $O(N)$. The drawback to this system, however is that the data in the two input FIFOs are required to be sorted. This requires either the software to presort the two input lists to the FIFOs or for a dedicated sorting unit be added to perform the sort on the two inputs before passing them to the two FIFOs.

The paper then discusses the results obtained by implementing the three sorting units on a Xilinx Virtex-5 SXT FPGA. The results taken from the paper show that all three units can easily be scaled up or down depending on the need and the hardware resources available. The Sorting Network Unit best performed in systems where several data elements could be loaded and stored simultaneously and in these environments was shown to run at up to 194 MHz. The Insertion Sorting Unit performed best in systems with serial data loading and either serial or parallel data storing and was shown to run up to 265 MHz. The drawback to both of these sorting units is the fact that both require a lot of hardware resources to implement. The FIFO Based Merge Sorting Unit required the least amount of hardware resources but also performed the worst running at a max clock frequency of 156 MHz.

The previous papers focused on the top level design of sorting units and did not look into the details of the building blocks to help improve the latency that these sorting networks have. The takeaways from the first two related papers are ideas and techniques that can be used to build hardware sorting units.

A high-performance comparator design can have a significant impact on the performance of an entire sorting hardware block [11]. In the paper *High Speed Comparator Architectures for Fast Binary Comparison* two different comparison architectures are proposed and compared to the traditional implementation [13]. The baseline design that the two proposed architectures are tested against is a traditional magnitude comparator [22].

The first proposed architecture has two outputs *H*$(A > B)$ and *S*$(A < B)$ and takes a parallel approach to comparing the two numbers A and B. In the first stage of the architecture it identifies and extracts the 1's of A which have a 0 in the corresponding spot in B. This process is done in parallel with one side performing the operation previously described and the other side performing the identical operation but with B in respect to A. After stage 1, two numbers (A' and B') have been formed each containing only the 1's that make that number greater than the other number in their corresponding positions. The second stage then zeros all 1's in both A' and B', except for the 1 in the most significant position. Stage two creates two new numbers A" and B" with zeros except for the 1 in the most significant position of A' and B' (the logic used in this stage is similar to the priority logic of a priority encoder). The final stage takes A" and B" finds which number has the 1 in the most significant position and then creates the two output signals H and S accordingly.

The second proposed architecture uses look-ahead-logic along with a parallel architecture design. This architecture works by using compare *look-ahead-logic* to create compare signals for each bit position. For each two number input pair into the *look-ahead-logic* only one of the compare signals will go high. The compare signal will go high if either the ith bit of A and B are unequal or the most significant bits are equal but the next most significant bit of A and B are unequal. These compare signals are then used to create the two outputs of the compare unit (*H*$(A > B)$ *and S*$(A < B)$).

Verilog and the Xilinx ISE 8.2i platform were used to create a 32-bit implementation of all three of the architectures discussed in the paper. Only the results for performance was published in the paper, the results for energy and area were not listed. The results from the implementation proved that both proposed architectures outperformed the traditional architecture, with the *compare look-ahead-logic* outperforming the first proposed architecture. The first proposed architecture was shown to have a simulation decrease of 23.77% in path delay when compared to the traditional architecture. The *compare look-ahead-logic* was shown to have a decrease of 35.22% over the traditional architecture. The fast compare unit design proposed in this paper will be used in the work in this thesis to be speed up the components of the hardware sorting units.

2.2 RISC-V Instruction Set

For the work in this thesis the RISC-V instruction set architecture was chosen. RISC-V is a relatively new instruction set architecture with some aspects that differ it from many of the other instruction set architectures.

RISC-V is an open source instruction set architecture (ISA) developed at the University of California at Berkeley. RISC-V is based on the reduced instruction set computer (RISC) design. The RISC-V ISA also has significant open source documentation, compiler tool chains, operating system ports, reference software simulators, cycle-accurate FPGA emulators, high-performance softcores, efficient ASIC implementations of various target platform designs, configurable processor generators, architecture test suites, and teaching materials to accompany it [17].

The RISC-V ISA is defined as a base integer ISA with two types of optional extensions (standard and nonstandard) to extend the instruction-set. Standard extensions are useful in most cases and should not conflict with other standard or nonstandard extensions, while nonstandard extensions are usually specialized for specific purposes and may conflict with other extensions. Although these extensions can be added to the instruction set the base instructions can never be redefined. The base instruction set also has two variants, RV32I and RV64I. These variants are characterized by the width of the registers and the size of the user address space, RV32I provides a 32-bit user-level address space while RV64I provides a 64-bit address space.

The RISC-V ISA comes with numerous benefits over other instruction set architectures. One of the main differences between RISC-V and the other ISAs is that RISC-V is a frozen ISA [17, 29]. What this means is that the base instructions for RISC-V are frozen in time and are not going to be changed. These base instructions are also kept to a minimal set of instructions that provide a reasonable target for compilers, linkers, and assemblers. The base instructions can be viewed as the foundation to which extensions can be added to help customize the ISA for each situation [29]. Other instruction set architectures typically treat their ISA as a single entity with needed instructions being added to the instruction set. This often results in those ISAs becoming bloated with unnecessary and outdated instructions that still require support. RISC-V avoids this issue with their base instruction being a fully supported standalone ISA with additional instructions being added to optional extensions.

Chapter 3

DESIGN AND IMPLEMENTATION

To test the performance and costs of implementing a hardware sorting unit along with an added instruction to the RISC-V ISA, a RISC-V processor was set up to be both the baseline for results and the foundation for implementing the hardware sorting units. This chapter documents how the RISC-V baseline processor design. The chapter also documents the construction and implementation of the hardware sorting unit and corresponding instruction into the RISC-V processor.

3.1 Processor Design

The processor for this work is based on the Phelmino processor [5] and was adapted and modified to fit the needs of this project. The processor is a 4-stage in-order RISC-V CPU with support for the RV32I (Base Integer Instruction Set), RV32C (Standard Extension for Compressed Instructions), multiplication support only for RV32M (Standard Extension for Integer Multiplication and Division), RV32V (Standard Extension for Vector Operations), and PULP instruction extensions [5]. The processor's stages are: instruction fetch, instruction decode, execute, and write-back. Each pipeline stage takes two control inputs, enable and clear. The enable activates the pipeline stage and moves to the next instruction while the clear empties the instruction from the stage when the instruction is completed. Figure 1 shows the basic processor overview along with the signals connecting the various stages.

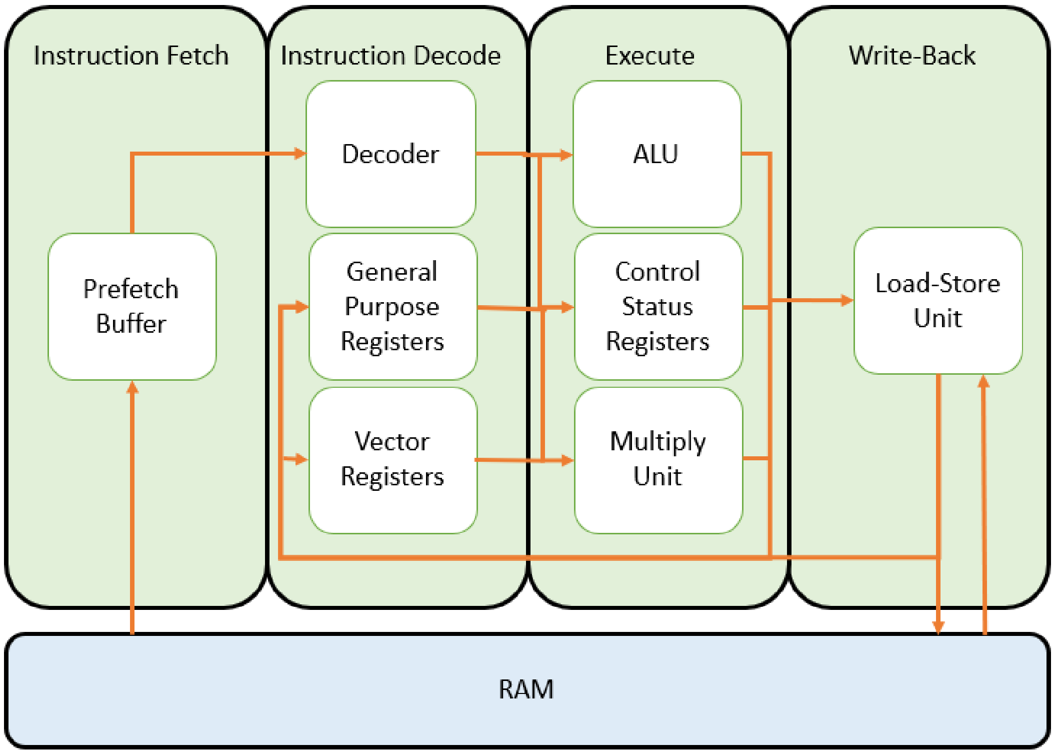


Figure 1: Overview of the Processor

3.1.1 Instruction Fetch

The instruction fetch stage supplies a single instruction per cycle. The instruction fetch stage makes use of a prefetch buffer to provide optimal performance. The prefetch buffer stores the fetched words from memory in a three entry FIFO.

3.1.2 Instruction Decode

The instruction decode stage is where the current instruction is decoded and various signals are set for the next stages.

The instruction decode stage contains a decoder unit and two sets of registers; general purpose registers and vector registers. The decoder unit is responsible for taking in the current instruction and constructing the appropriate signals to execute that instruction.

The general purpose register file contains 32 registers of 32 bits, although register 0 is bound to 0 and can only be read (the processor will throw in a zero whenever it sees the zero register being used). Since the processor has a register that constantly contains a zero the processor really only contains 31 hardware registers. A vector register file was added to the processor design to help support the sorting hardware units. The vector register file contains 32-bit vector registers and also contains 32 registers of size 16 bits, containing the type of each of the vector registers. Each vector register is capable of holding a configurable type of data. This data type does not have to be identical to the other vector registers. The length of the vector registers can be altered by changing the value of the vector length register. The vector length register is a global register for the entire vector register file. Specific vector operations can then be performed on these vector registers.

Once the signals are produced from the decoder unit and the needed values are taken from the register files they are passed to the execute stage.

3.1.3 Execute Stage

The execute stage takes the control signals and values from the instruction decode stage and executes the instruction. The signals and values from the decode stage indicate to the various components in the execute stage what needs to be done. The execute stage contains the Arithmetic Logic Unit (ALU), Control Status Registers (CSR), and Multiply unit. The custom hardware sorting units proposed in this work were also added to the execute stage.

3.1.4 Write-Back Stage

The write-back stage accesses the data memory for the processor. The write-back stage contains the Load-Store-Unit (LSU). The LSU loads and stores words in memory; it is capable of handling words (32 bits), half words (16 bits), and bytes (8 bits). The LSU is capable of performing misaligned data access. However, by performing two separate word-aligned accesses internally, for misaligned load and stores, the write-back stage requires two cycles.

3.2 Sorting Units

Three different sized sorting units are created to perform sort operations on data of various sizes.

3.2.1 32-Bit Sorting Unit

The smallest and most basic sorting unit that can be created is a two-element sorting unit (this will be the building block for all the other sorting units). These basic sorting units take as inputs two N-bit numbers and output the two numbers in an ordered sequence. These two-element sorting units are composed of a comparator and two multiplexers. The output of the comparator is connected to the data selection input of the multiplexers and the two inputs to the sorting unit are fed as inputs to the two multiplexers. The circuit for the two-element sorting unit is shown in Figure 2.

A number of these two-element sorting units can then be used to produce larger hardware sorting units as shown in Figure 3. The four inputs to the four-element

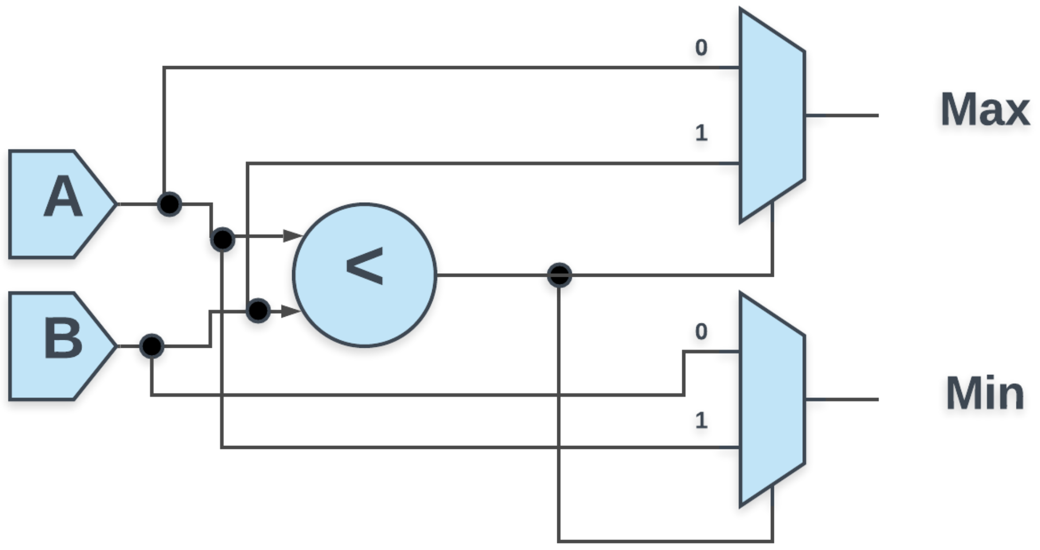


Figure 2: Circuit Diagram for Two Element Hardware Sorting Unit

sorting unit are first split into two different groups of two numbers. We then use two two-element sorting units to put the groups into two ordered sequences. The largest number of each sequence is compared with the largest number of the other sequence; the same is done for the smallest numbers. The smaller number of the larger group is put into the fifth two element sorting unit with the larger number of the smaller group to find the second largest number in the input. Through the above process, we can transform four random inputs into an ordered sequence of outputs. Figure 4 shows the hardware circuit required to construct the four element sorting unit.

Hardware sorting units of any size can be made by using the same type of logic that made it possible to extend the two element sorting unit into a four element sorting unit. For example, a sorting unit of size 2N (2N elements need to be sorted) would require the use of five sorting units of size N. These five sorting units would be arranged in the same way that the 5 two element sorting units are arranged. The first two sorting units serve the purpose of putting the 2N inputs into two ordered sequences of size N. Then one of the next two sorting units then takes the largest half

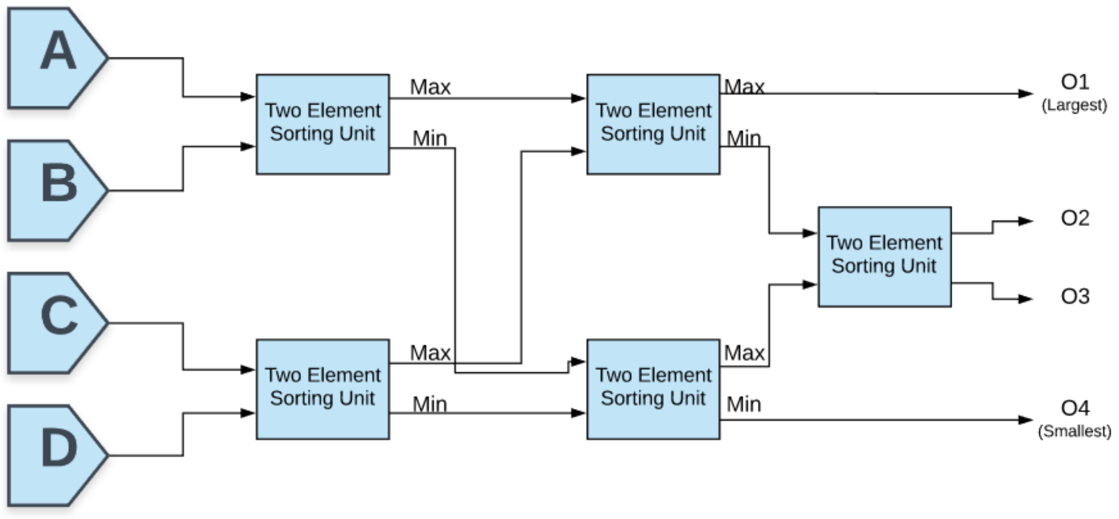


Figure 3: Four Element Sorting Unit Block Diagram

of each ordered sequence to put the largest N/2 inputs in order. The other sorting unit takes the smallest half of each ordered sequence of size N to put the smallest N/2 inputs in order. The last sorting unit is then used to put the remaining N inputs in order. A block diagram showing the configuration for a sorting unit of size 2N is shown in Figure 5.

The largest sorting unit size used in this thesis was a 32 element sorting unit. This choice was made since the RISC-V architecture uses 32 registers, so the use of a 32 element sorting unit allows for all 32 registers to be sorted without having unnecessary hardware. The sorting unit was placed in the execute stage of the processor and receives its inputs from the vector registers. The decision to use the vector registers as inputs to the sorting unit was made based on a few different factors. One of the primary factors was that in a RISC-V processor some of the regular registers are required to contain particular values (e.g. register 0 contains a value of 0 and other registers contain the PC, stack pointer, etc.). This means that not all 32 registers could be used in a sorting unit since this can cause undesired results if those values are used as inputs to the sorting unit. Another advantage to using the vector registers

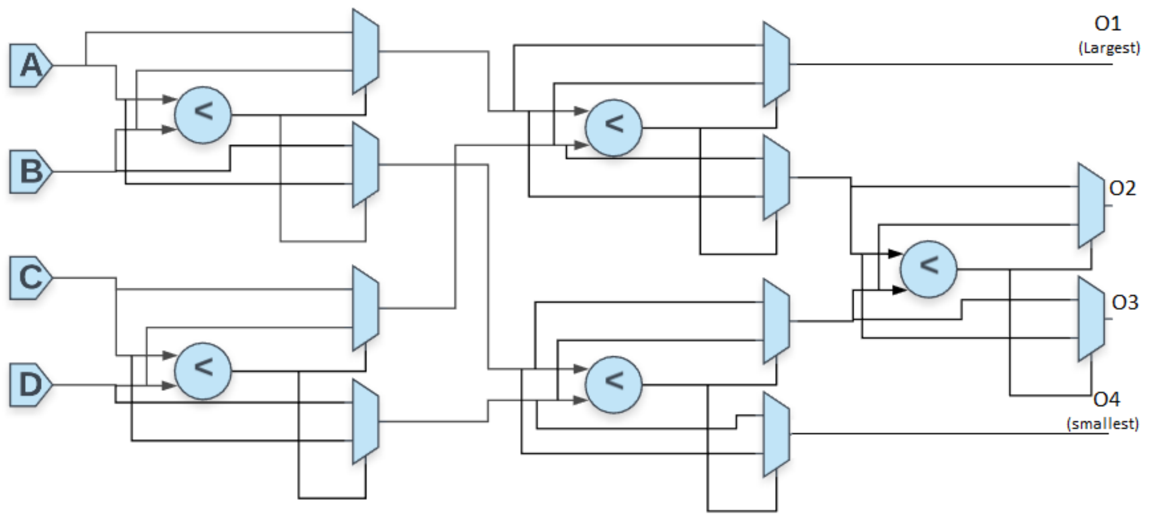


Figure 4: Circuit Diagram for a Four Element Hardware Sorting Unit

as inputs to the sorting unit is that the RISC-V Standard Vector Extension has instructions to load values into registers in particular patterns.

3.2.2 16-Bit and 8-Bit Sorting Units

The 16-bit and 8-bit sorting units both take in 32 elements and use the same format as the 32 bit sorting unit. The only difference between the sorting units is that the 16 bit and 8 bit sorting units take as inputs 32 elements of size 16 and 8 bits respectively instead of 32 elements of size 32 bits. Since the registers of the RISC-V processor are 32 bits wide using a single 16-bit or single 8-bit sorting unit will leave at least half of the bits in the register unused. For the systems proposed in this work, two 16-bit sorting units and four 8-bit sorting units are used in parallel to maximize the throughput of the systems. For example if 16-bit data is required to be sorted, two sets of data can be loaded into the vector registers with the first set using bits 31 down to 16 and the second set using bits 15 down to 0. The output from each sorting unit will be one sorted set of data, this would then be stored in the vector registers

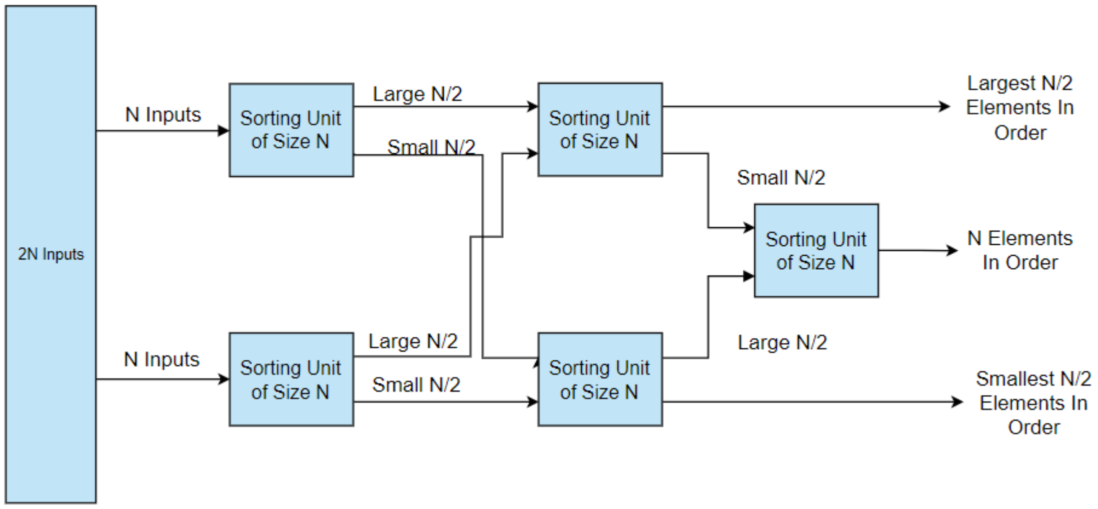


Figure 5: Block Diagram for a Sorting Unit of Size 2N

in the corresponding bit locations in which it was taken from. Figure 6 shows how multiple sorting units are used in parallel to maximize the throughput of the system when dealing with data of size 8 and 16 bits.

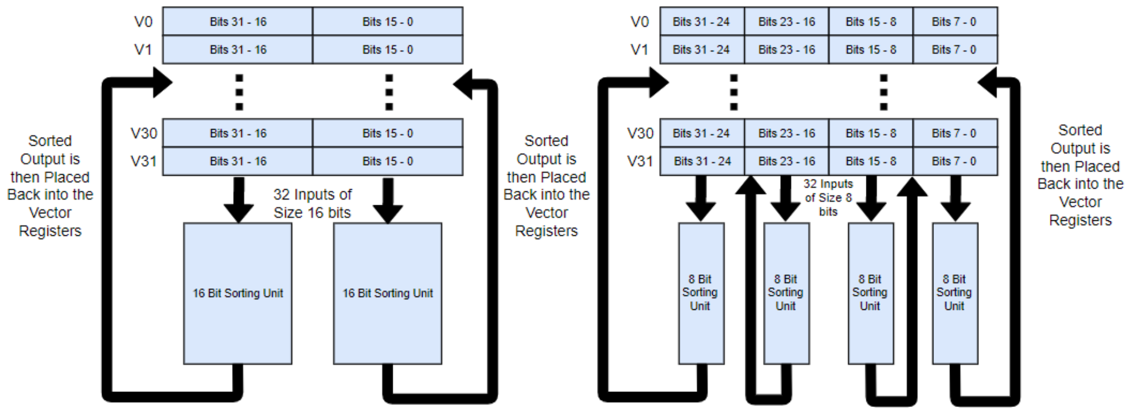


Figure 6: Multiple Sorting Unit Design

3.2.3 Sort Instruction

To use the implemented hardware sorting units, instructions need to be added to the processor. The RISC-V Instruction Set Architecture (ISA) allows for custom extensions to be made to help processors function most efficiently in their various applications. The RISC-V ISA has instruction encoding spaces and optional variable-length instruction encoding that were designed to make it easier to extend the ISA toolchain when building more customized processors [29]. The RISC-V Base Instruction Set has four core instruction formats (R,I,S,U) all of which have a fixed length of 32 bits. All of these instruction formats are shown in Figure 7. R-type instructions use 3 register

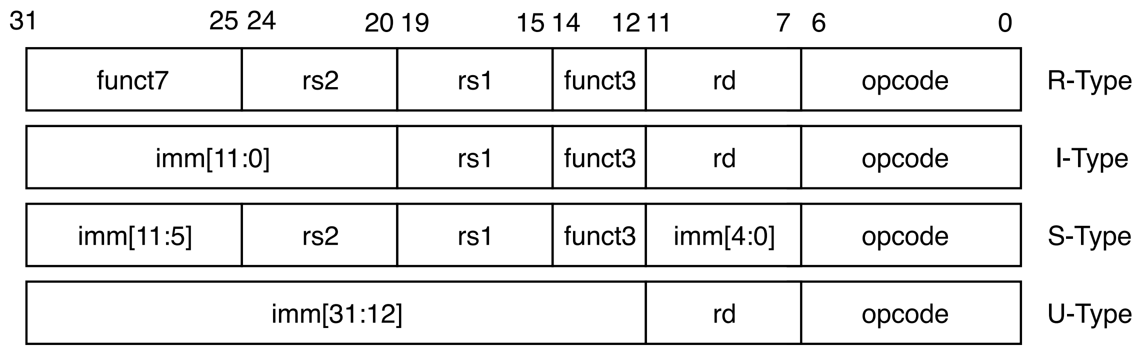


Figure 7: RISC-V Base Instruction Format

inputs. I-type instructions have immediates while U-type instructions use the upper bit values from the immediates. S-type are store instructions. The instructions are composed of five or fewer fields (opcode, rd — destination register, funct3/funct7 — type of operation, rs1/rs2 — source registers 1/2, imm — immediate value).

To extend the RISC-V ISA to have instructions that use our hardware sorting units only one op-code needs to be added to the instruction set. This can be done by using a single opcode to specify the SORT instruction and then by using the funct3 field to specify the type of operation that needs to be applied.

For the implementation of the sort instruction the I-type instruction format was used with the destination register and source register values left as ”don’t cares”. These values are not used because the sort instruction takes as input all the vector registers and stores the results back into all the vector registers. Each of the three sort instruction types (8-bit, 16-bit, and 32-bit) have a unique funct3 value to specify to the processor which operation needs to be performed.

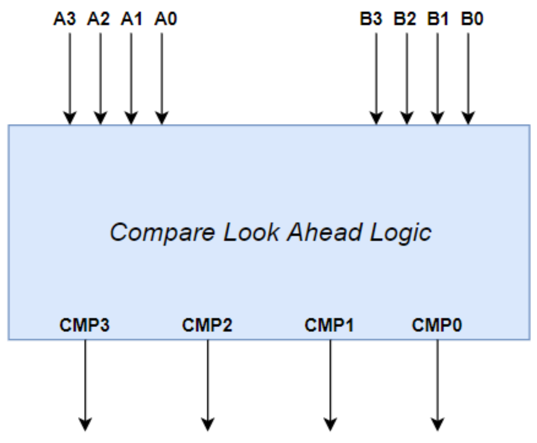
3.3 Fast Compare Unit

The sorting unit improves the performance of the processor on instructions that result in sorting but, due to its complexity, will slow the overall maximum clock frequency the processor can run at. To help improve the logic delay of the sorting unit, various components of the sorting unit were redesigned. This section discusses the creation of a fast compare unit that reduces the delay of the sorting unit.

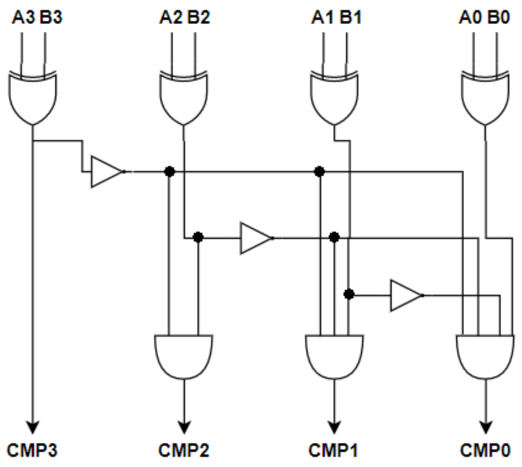
The design of the fast compare unit is based on the the second proposed architecture in the paper *High-speed Comparator Architecture for Fast Binary Comparison* by Deb and Chaudhury [13] as discussed in Chapter 2.1. Figure 8a shows the block level diagram of the compare look ahead stage of this architecture. To generate the signals (called CMPi) for each bit position (i), the logic compares the bits of each input (A and B). The compare signals are set high if:

- the bit of A and B are unequal
- the most significant bits are equal and the more significant position of A and B are unequal.

This means that for any two inputs A and B only one of the compare signals can be high. The compare look ahead logic is comprised of XOR, AND, and Inverters. The circuit diagram for a four bit compare look ahead logic is shown in Figure 8.



(a) Block Diagram



(b) Circuit Diagram

Figure 8: Block Diagram and Circuit Diagram for 4 Bit Compare Look Ahead Logic

The second stage takes the compares signals from the look ahead stage and combines it with logic to produce two output signals ($(A > B)$ *and* $(A < B)$). The second stage logic is comprised of AND gates, inverters, and two OR gates. The first set of AND gates tests to see if the *ith* bit of A and B are unequal. If the two bits are unequal then it produces a high signal, if not, the signal goes low. The second set of AND gates takes the output of the first set of AND gates and the compare signals from the *compare look ahead* stage. The output of the second set of AND gates is then summed in one of the two OR gates to produce the two output signals. Figure 9 shows the complete circuit for a four bit fast compare unit.

My implementation of the 32, 16, and 8 element compare units uses a four element compare unit and a two element compare unit as building blocks. The decision to use four and two element compare units as building blocks instead of expanding the compare logic was made to help reduce the design complexity.

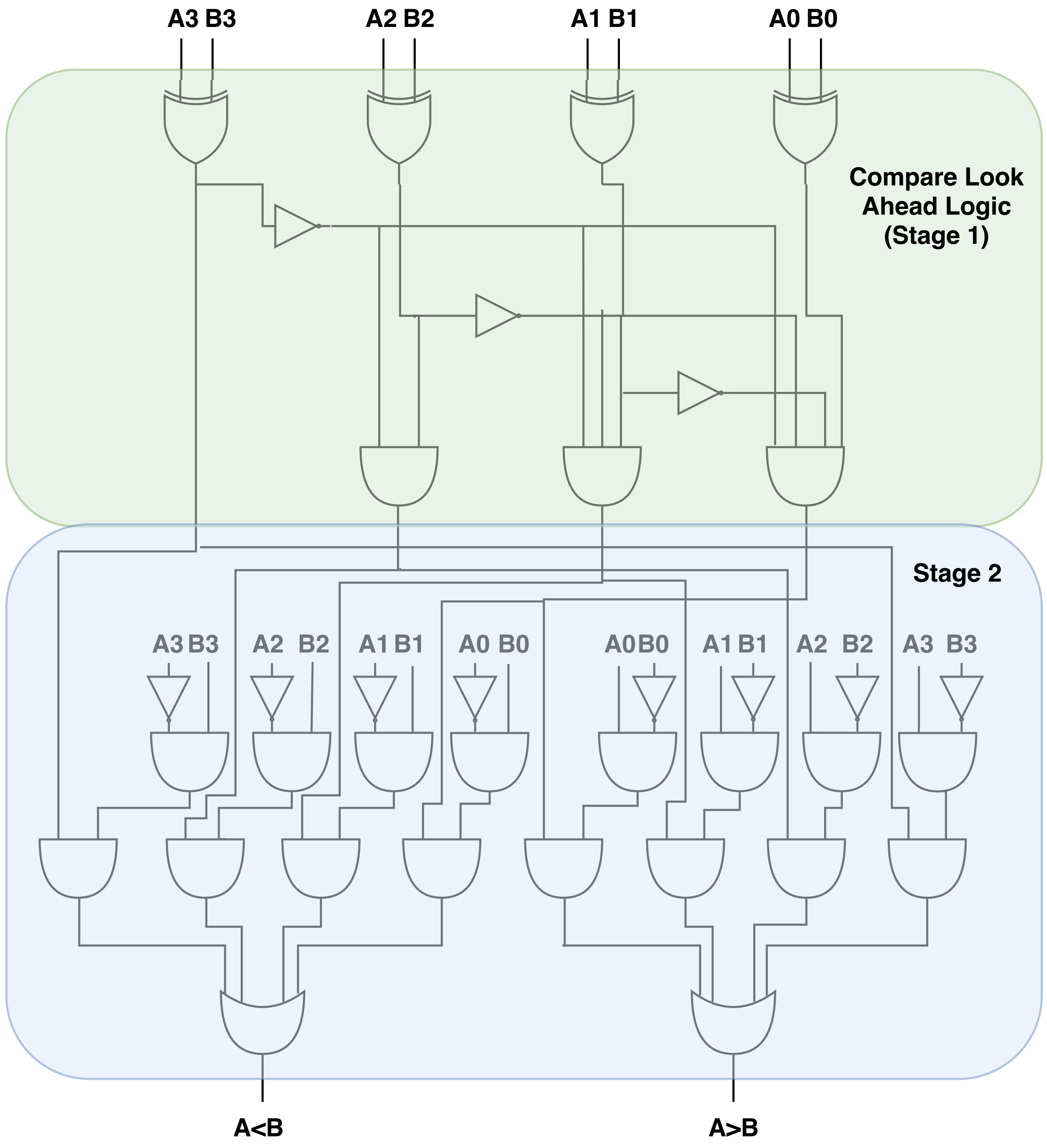


Figure 9: Circuit Design for 4-Bit Fast Compare Unit

In the designs for the 8,16, and 32 bit compare units the two element compare unit will only be receiving its inputs from the output of the four element compare units. The four element output bits ($(A > B)\ and\ (A < B)$) can never be high at the same time. This means that the input to the two element compare unit can never have particular combinations as inputs. The following combinations are a list of the inputs that the two element compare units can never receive: (10,10), (10,11), (11,10), (11,01), (01,11), and (01,01). Due to the restriction of particular input combinations, the logic for the two element compare unit can be reduced. The new Boolean expression for the reduced two element compare unit becomes:

$$(A < B) = A_1 + \overline{\mathrm{A}_1} A_0 \overline{\mathrm{B}_1 \mathrm{B}_0}$$

. The circuit logic for the two element compare unit is shown in Figure 10.

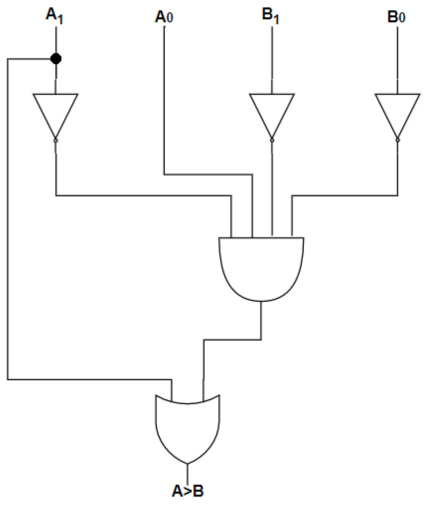


Figure 10: Two Element Compare Unit Logic

3.3.1 8-Bit Fast Compare Unit

The 8-Bit Fast Compare Unit is comprised of two stages. The first stage uses two four element fast compare units to compare every 4 bits of the input numbers. The second stage is comprised of a single two element fast compare unit and takes in the result of the 2 sets of 2-bit numbers and computes the final result for the 8-bit numbers. Figure 11 shows the block diagram for the 8-bit Fast Compare Unit.

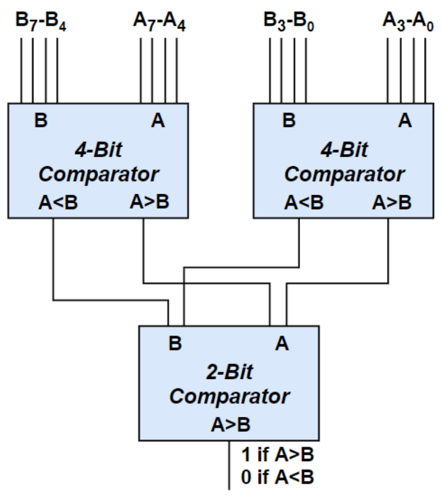


Figure 11: Block Level Diagram For 8-Bit Fast Compare Unit

3.3.2 16-Bit Fast Compare Unit

The 16-Bit Fast Compare Unit is comprised of two stages. The first stage uses four four-element fast compare units to compare every 4-bits of the input numbers. The second stage is comprised of a single four element fast compare unit and takes in the result of the four sets of 2-bit numbers and computes the final result for the 16-bit numbers. Figure 12 shows the block diagram for the 16-bit Fast Compare Unit.

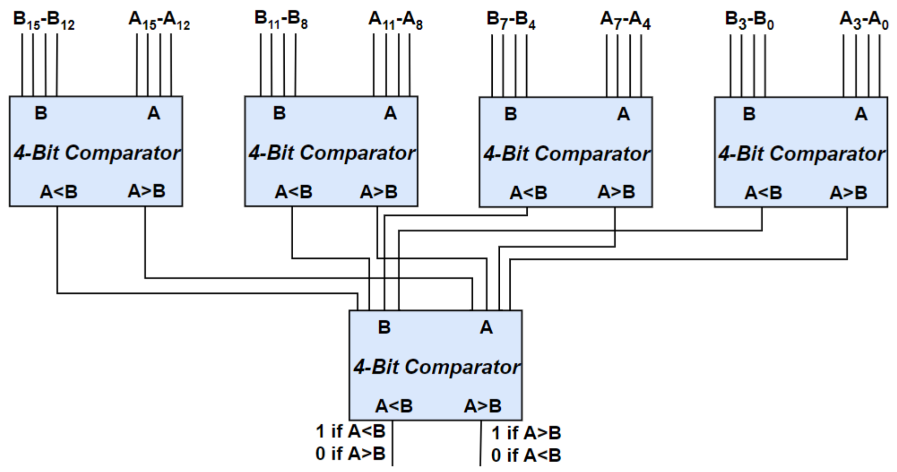


Figure 12: Block Level Diagram For 16-Bit Fast Compare Unit

3.3.3 32-Bit Fast Compare Unit

The 32-Bit Fast Compare Unit is comprised of three stages. The first stage uses eight four-element fast compare units to compare every 4-bits of the input numbers. The second stage is comprised of two four-element fast compare units and takes in the result of the four sets of 2-bit numbers and computes the result for the 16-bit numbers. The third and final stage takes the output of the second stage and puts it through a single two element fast compare unit to compute the final result for the 32-bit numbers. Figure 13 shows the block diagram for the 32-bit Fast Compare Unit.

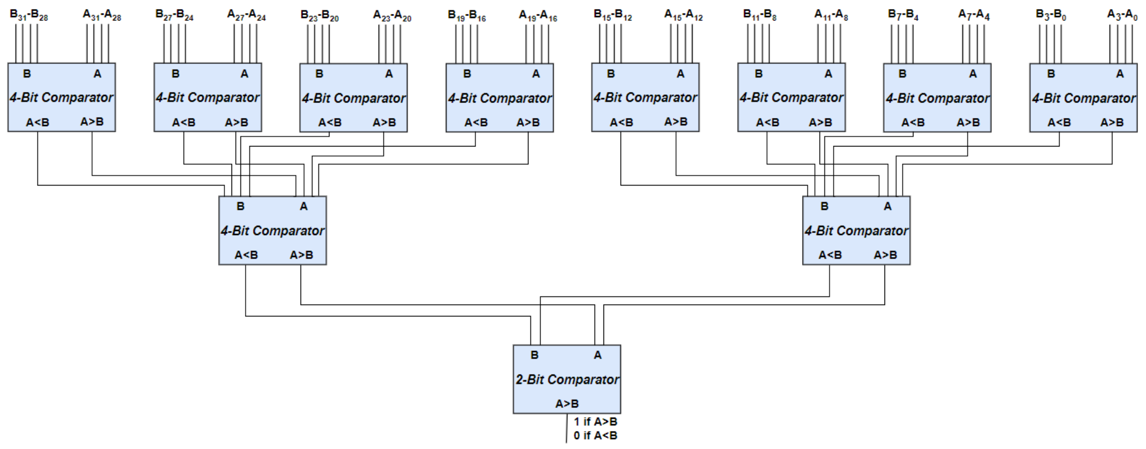


Figure 13: Block Level Diagram For 32-Bit Fast Compare Unit

3.4 Pipelined Sorting Unit

The addition of the hardware sorting units to the execute stage causes the system to run at a slower frequency than the 167 MHz frequency of the unaltered processor. This is caused by the fact that the hardware sorting units adds logic delay to the execute stage. For applications that are heavily dependent on the sorting instruction, the reduction of the maximum frequency is a small price to pay for the overall performance the sorting unit provides. However, in applications that do not use the sorting instruction heavily, the reduction of the maximum frequency can offset the performance increase the sorting unit provides. For example a 3x3 median filter application on a 512x512 image uses about 52,540,000 instructions to perform the median filter on the original RISC-V processor, which runs at a maximum frequency of 166.6 MHz. Of the 52,540,000 instructions 95.8% of the instructions are used to perform the sorting application. This means that when using a hardware sorting unit it would require 2,863,000 instructions to run the same application. For this example, to see an overall speedup in the application run time the slowest max frequency possible would be 8.7 MHz. Not all applications will be as dependent on sorting operations as median filtering is. If an application only has 10% of its instructions as sort instructions the reduction in the clock frequency caused by the hardware sorting units will outweigh the benefits that they provide.

Pipelining can be used to combat the clock frequency problem that is caused by the addition of the hardware sorting units. Pipelining is the process of breaking a large hardware block into multiple stages in order to speed up the maximum clock frequency of the processor. The hardware unit is broken into multiple stages by inserting flip-flop registers in between combinational logic blocks.

Using the pipelining technique on the hardware sorting units proposed in this work, the clock frequency can remain unchanged from the RISC-V reference design. To pipeline the sorting units, flip-flops were added to the sorting units in the 2-element compare blocks. The flip-flops were then placed in their optimal locations using the Vivado Synthesis Retiming Tool. The Retiming Tool is an automated process built in to the Vivado Tool Chain that finds the optimized locations to place the flip-flops in order to maximize the clock frequency of the design and minimize the hardware resources the flip-flops require. By adding the flip-flops and using the Vivado Retiming

Tool the hardware sorting units were broken into 28 stages. By breaking the hardware sorting units into 28 stages the maximum frequency for all the designs was increased from their unpipelined frequencies. The pipelined designs' maximum frequencies were all increased to 167 MHz (the clock frequency of the unaltered processor). The increase of the clock frequency, however, did not come without cost. The addition of the flip-flop registers to the hardware sorting units increased the resources required for the systems, in particular the added flip-flop registers used an additional 1500 flip-flops out of the 19730 flip-flops required to implement the entire unpipelined processor.

Chapter 4

RESULTS AND ANALYSIS

The proposed systems were synthesized using Vivado 2018.1 and placed on a Digilent Nexys 4 DDR Development board with a Xilinx Artix-7 FPGA. To collect the timing measurements for each system, signals were output to pins on the board and the timing measurements were taken using an oscilloscope. The benchmark used to test the performance for all the systems was a 3x3 median filter on various image sizes (with the size of the data being 8 bits) ranging from 16x16 to 64x64. For the unaltered RISC-V processor (no sorting unit/instruction) the sorting algorithm used was an Insertion Sort algorithm. All other processors used the added sort instruction to perform the sort operation required for median filtering. Adequate testing for each design was done to verify that the system was performing as designed. The naming convention used in the results and analysis is as follows: original RISC-V processor with no sorting unit or sorting instruction (unaltered processor), 32-Bit Sorting Unit with basic VHDL comparator (32-Bit VHDL Compare), 32-Bit Sorting Unit with fast compare unit (32-Bit Fast Compare), 16-Bit Sorting Unit with basic VHDL comparator (16-Bit VHDL Compare), 16-Bit Sorting Unit with fast compare unit (16-Bit Fast Compare), 8-Bit Sorting Unit with basic VHDL comparator (8-Bit VHDL Compare), and 8-Bit Sorting Unit with fast compare unit (8-Bit Fast Compare).

4.1 Results

The timing results were measured by outputting a start signal to a pin on the board when the code first begins. Once the code has finished a done signal is then output to a pin. The oscilloscope measures the time difference between when the two signals go high, to determine the time the code runs on each system. Table 1 shows the timing results for all the unpipelined systems. Figure 14 shows the timing results for all the unpipelined systems in a graph format while Figure 15 shows the timing results for the systems normalized with respect to the timing results for the unaltered processor.

Table 1: Timing Results of Unpipelined Systems

Processor Type	Max Freq	16x16	32x32	64x64
Unaltered Processor	167 MHz	742 us	2.51 ms	9.72 ms
32-Bit VHDL Compare	9.1 MHz	119 us	544 us	2.17 ms
32-Bit Fast Compare	9.34 MHz	107.4 us	532.1 us	2.15 ms
16-Bit VHDL Compare	10 MHz	64.5 us	255 us	1.352 ms
16-Bit Fast Compare	10.2 MHz	62.2 us	253.7 us	1.338 ms
8-Bit VHDL Compare	11.76 MHz	52.0 us	147.2 us	576.4 us
8-Bit Fast Compare	11.76 MHz	52.0 us	147.0 us	577.0 us

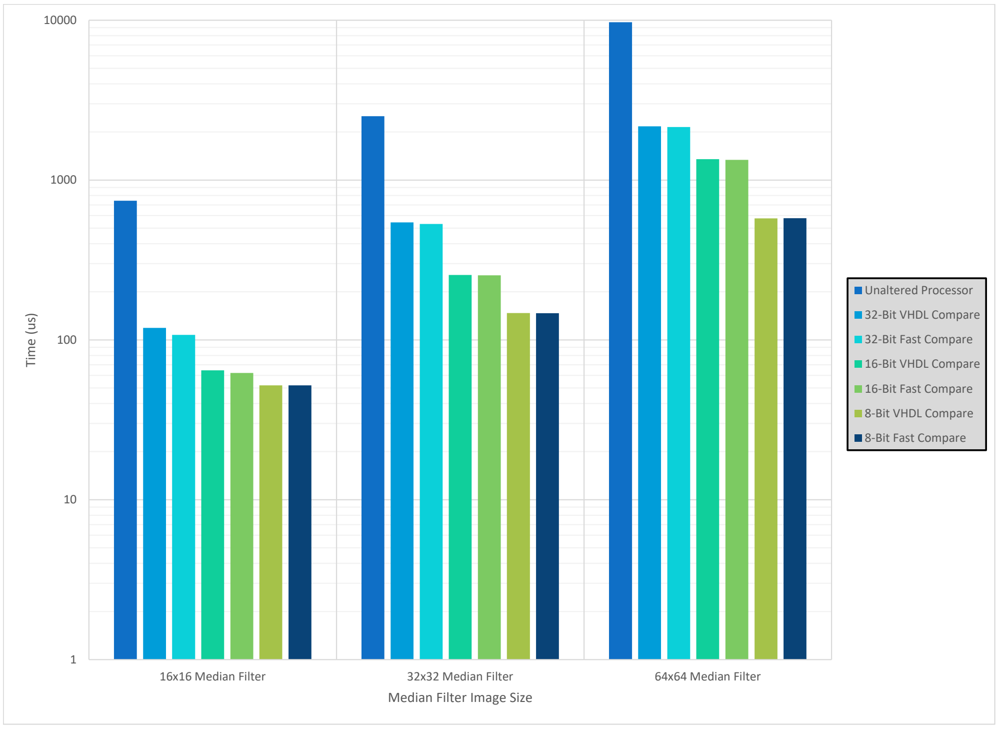


Figure 14: Timing Results of Unpipelined Systems

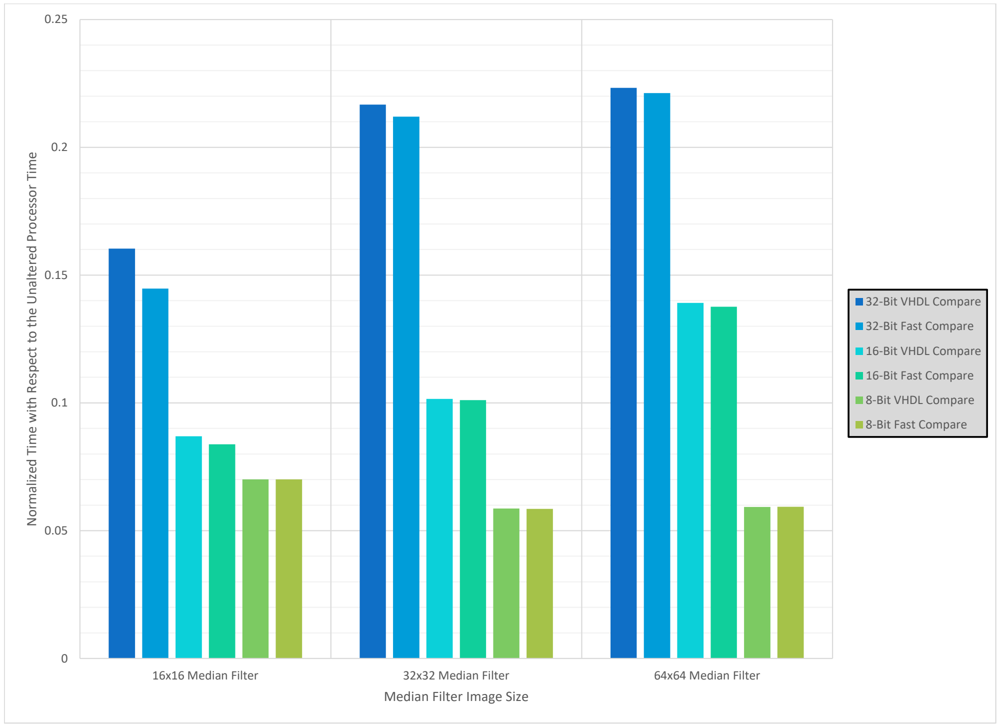


Figure 15: Normalized Timing Results of Unpipelined Systems

The addition of the hardware sorting units have been proven to improve the performance of the system on sorting applications, however, this improvement comes at a cost. The addition of sorting units require the use of additional hardware resources to implement. The fast compare units also improve the performance of the systems but they too come at a hardware cost. These hardware resource costs are measured in look-up table (LUTs) slices used as reported by Vivado after implementation. Table 2 and Figure 16 show the hardware cost of the various sorting unit implementations. For a more detailed break down of the hardware resources required for the various processor components see Appendix A. Figure 17 shows a graph of the normalized LUT usage of the sorting units with the fast compare units. The sorting units with the fast compare units are normalized with respect to the sorting units of the same size with the basic compare.

Table 2: Area Usage of Sorting Units

Sorting Unit Type	LUTs Used
32-Bit VHDL Compare	7028
32-Bit Fast Compare	25546
16-Bit VHDL Compare	3520
16-Bit Fast Compare	9827
8-Bit VHDL Compare	1762
8-Bit Fast Compare	2777

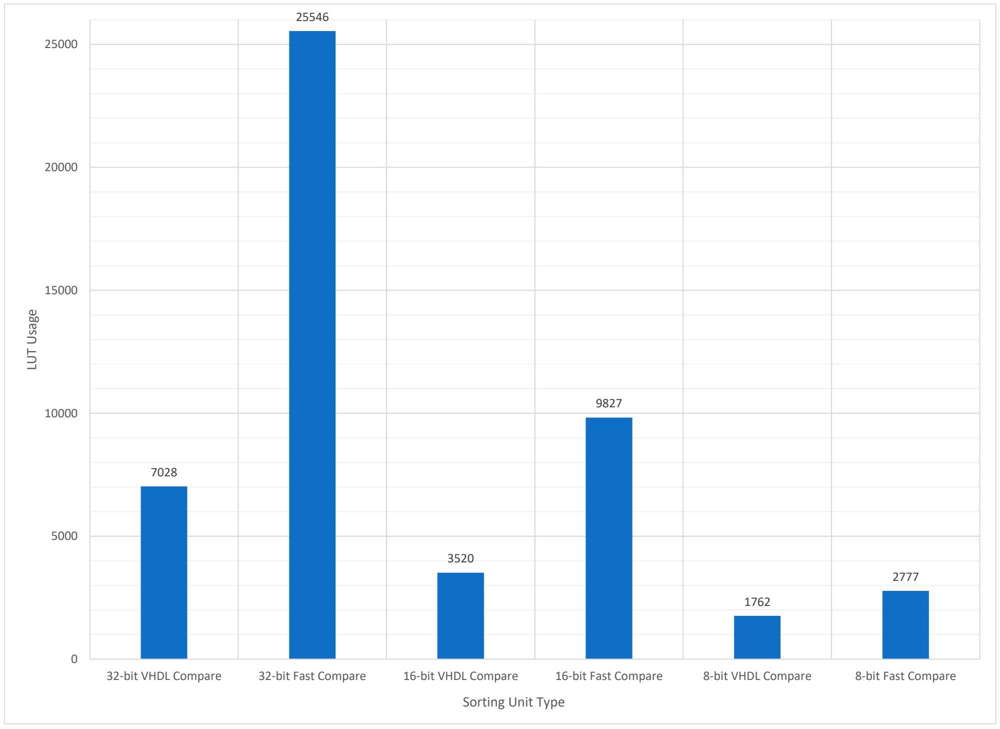


Figure 16: Area Usage of Sorting Units

The proposed systems that use the 16-bit and 8-bit sorting units use multiple (two and four respectively) sorting units each in their designs. Therefore, the systems that utilize these units require additional hardware resources to accommodate the parallel

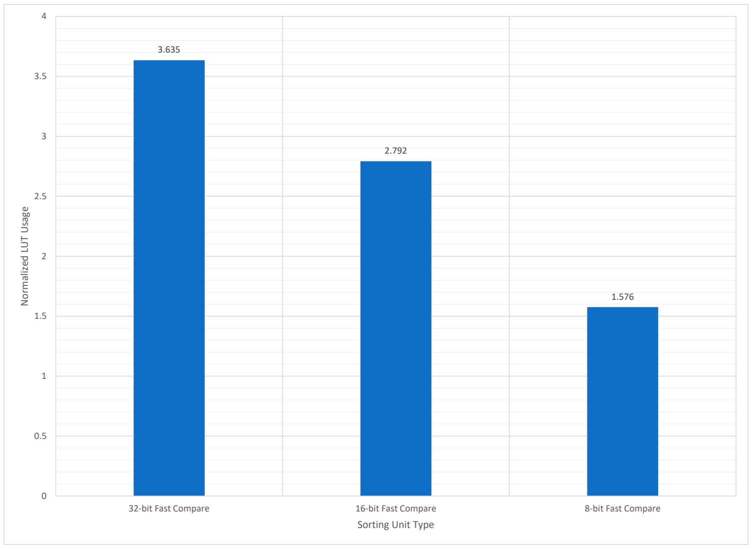


Figure 17: Normalized Area Usage of Sorting Units

sorting units. The total hardware resources required to implement the each system are shown in Table 3 and Figure 18. The normalized area used with respect to the area used by the unaltered processor is shown in Figure 19.

Table 3: Area Usage of Processors

Processor Type	LUTs Used	Normalized LUTs Used
Unaltered Processor	31321	1
32-Bit VHDL Compare	40765	1.301
32-Bit Fast Compare	59754	1.907
16-Bit VHDL Compare	39854	1.272
16-Bit Fast Compare	50753	1.620
8-Bit VHDL Compare	39875	1.273
8-Bit Fast Compare	46761	1.492

4.1.1 Pipelined Sorting Units

The timing results for the systems that use the pipelined sorting units were gathered identically to how the other timing results were captured. The pipelined systems were

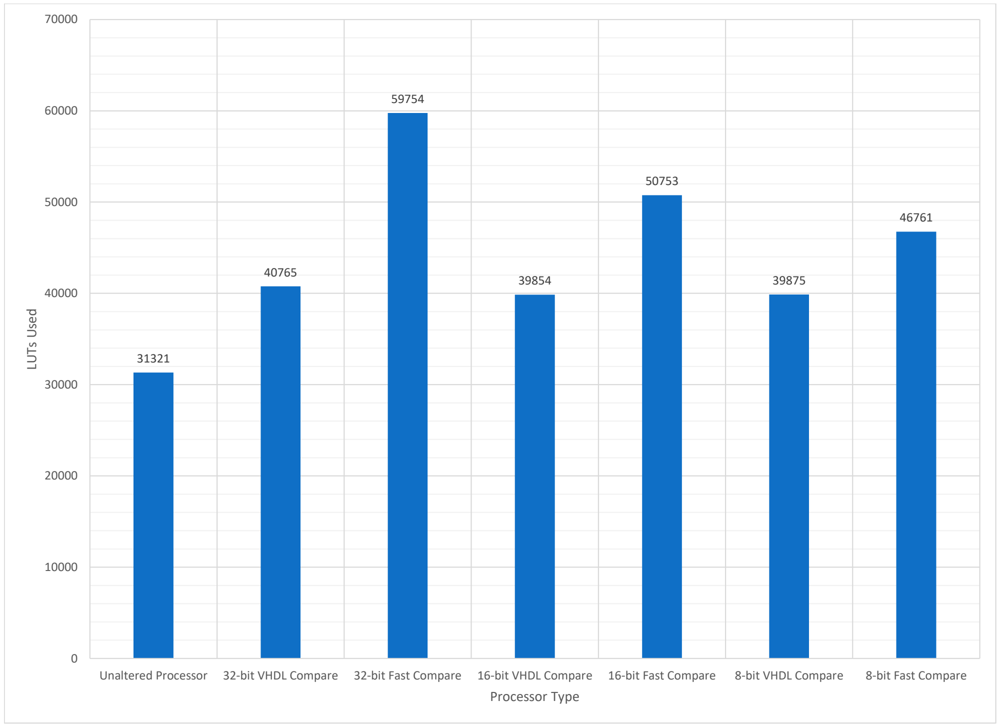


Figure 18: Area Usage of Processors

implemented on the same in-order processor as the other systems with the hazard unit stalling the processor for the sort instructions. Each time a sort instruction was received by the processor the processor would stall all future instructions while it waited for the sort instruction to complete. Table 4 shows the timing results for all the pipelined systems and the unaltered processor. Figure 20 shows the timing results for all the pipelined systems in a graph format while Figure 21 shows the timing results for the pipelined systems normalized with respect to the timing results for the unaltered processor. The systems with the fast compare units were also tested but were shown to run at the same time as the systems that use the VHDL compare. This can be attributed to both systems requiring the sorting units to be broken into 28 stages. This means that the additional timing benefits the fast compare units gained were

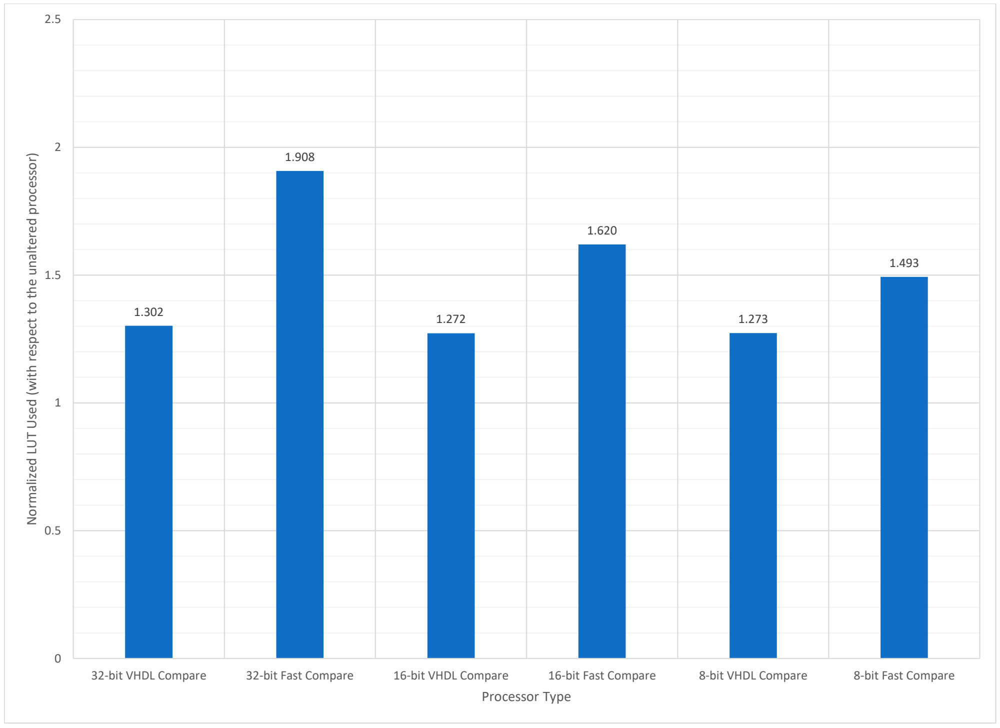


Figure 19: Normalized Area Usage of Processors

lost when the entire sorting units were pipelined. The hardware resources required to implement the pipelined systems were identical to their unpipelined counterparts in the number of LUTs required, however, the number of flip-flops differed. The pipelined systems used an additional 1500 flip-flops versus the unpipelined systems.

Table 4: Timing Results of Pipelined Systems

Processor Type	Max Freq	16x16	32x32	64x64
Unaltered Processor	167 MHz	742 us	2.51 ms	9.72 ms
32-Bit Pipelined	167 MHz	184 us	838 us	3.35 ms
16-Bit Pipelined	167 MHz	147 us	784 us	2.07 ms
8-Bit Pipelined	167 MHz	109.4 us	677 us	1.63 ms

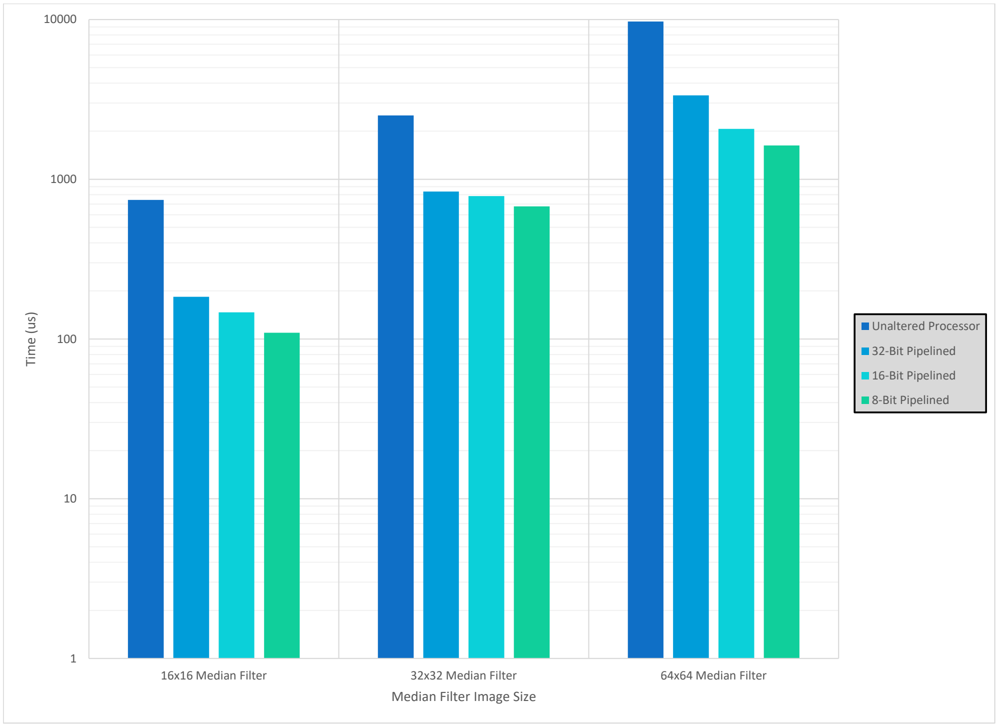


Figure 20: Timing Results of Pipelined Systems

4.1.2 Estimated Power Consumption

Along with the timing results and area usage of the systems the power consumption of each system was also recorded. The power consumption for each system however, is only the estimated power consumption. The estimated power consumption for each of the systems was recorded using the Vivado Power Analysis Tool. Table 5 shows the estimated power consumption of each design along with the normalized power consumption with respect to the unaltered processor. Figure 22 shows a graphical representation of the estimated power of the systems while Figure 23 shows the normalized power consumption of each system with respect to the unaltered processor.

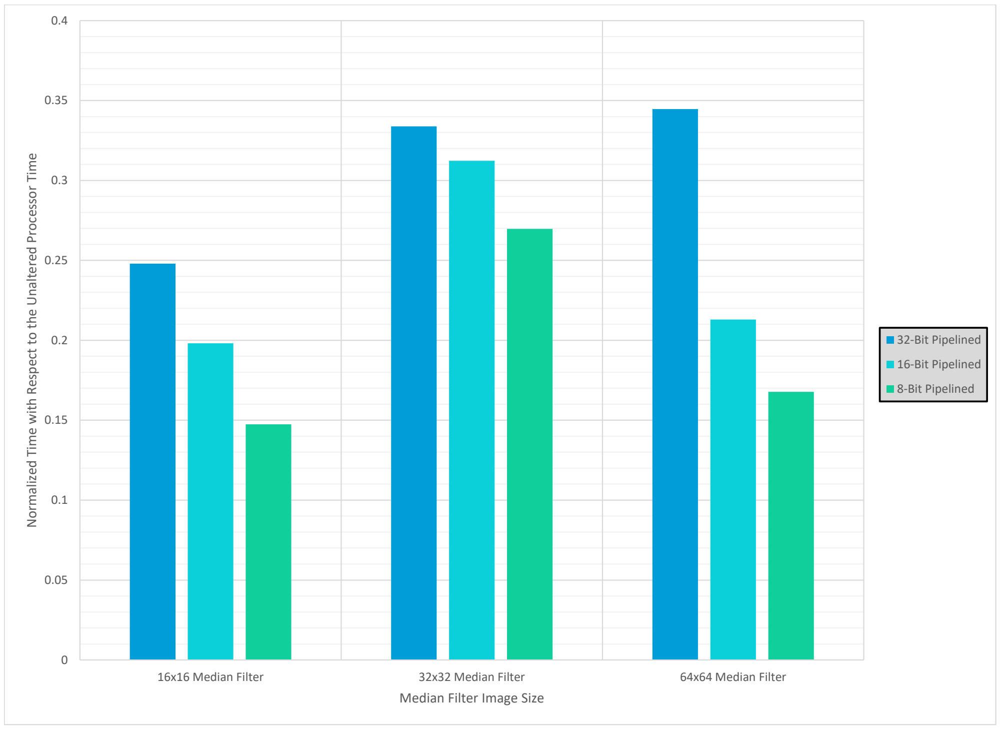


Figure 21: Normalized Timing Results of Pipelined Systems

Table 5: Power Consumption of Systems

Processor Type	Power Consumption (W)	Normalized Power
Unaltered Processor	0.523	1
32-Bit VHDL Compare	0.682	1.30
32-Bit Fast Compare	0.786	1.50
16-Bit VHDL Compare	0.681	1.30
16-Bit Fast Compare	0.760	1.45
8-Bit VHDL Compare	0.679	1.29
8-Bit Fast Compare	0.734	1.40
32-Bit Pipelined	1.507	2.88
16-Bit Pipelined	1.319	2.52
8-Bit Pipelined	1.027	1.96

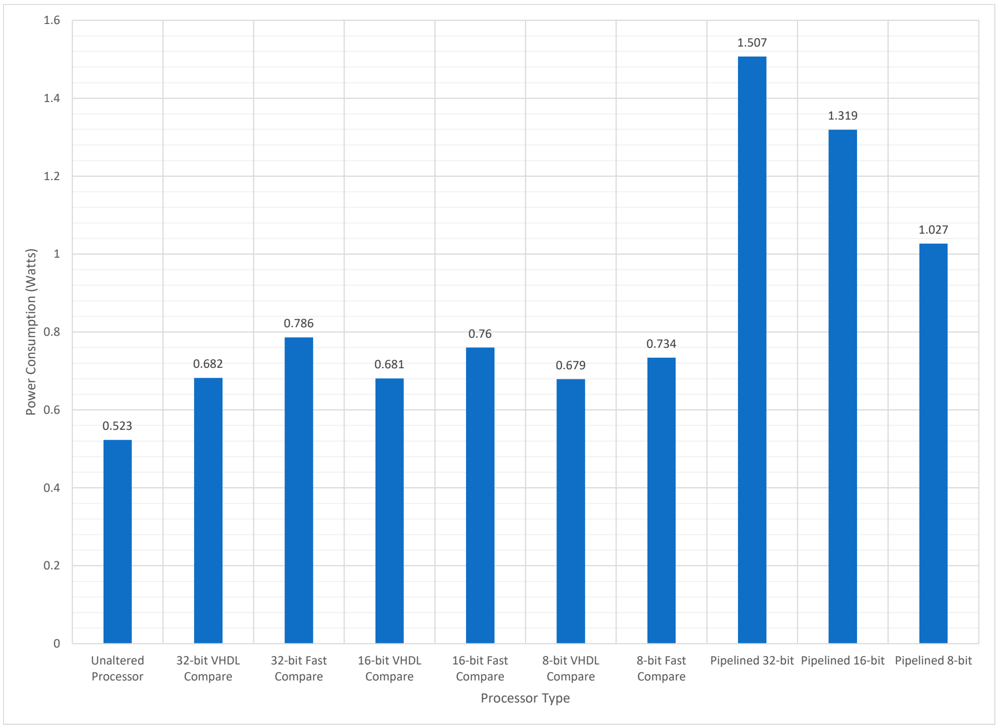


Figure 22: Power Consumption of Systems

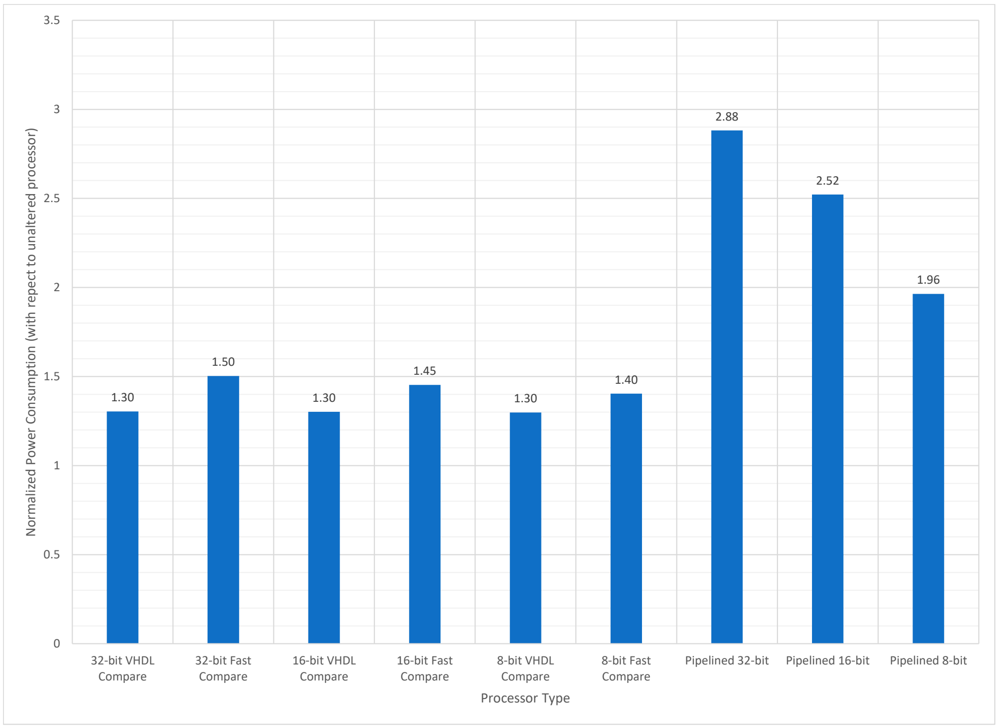


Figure 23: Normalized Power Consumption of Systems

4.2 Analysis

4.2.1 32-Bit VHDL Compare Unpipelined

The 32-bit sorting unit with the basic VHDL compare performed well against the unaltered processor. The implementation of the sorting unit caused a 5x to 7x increase in speed at which the code ran. Although, the median filter code saw a large decrease in the runtime the implementation of the 32-bit sorting unit caused a 18.35x reduction in the maximum frequency of the processor. This means that the sorting unit will reduce the performance on code that does not rely heavily on sorting. The 32-bit sorting unit with the basic VHDL compare also increased the hardware resources required for the processor by 130%.

4.2.2 32-Bit Fast Compare Unpipelined

The 32-bit sorting unit with the fast compare unit performed slightly better than the 32-bit sorting unit with the VHDL compare. The 32-bit fast compare ran at 9.34 MHz, which is .24 MHz faster than the VHDL compare but still 17.88x slower than the unaltered processor. The fast compare unit gave the 32-bit sorting unit a slight advantage over the basic VHDL compare, but, it came at a steep hardware resources cost. The 32-bit fast compare unit used 3.6x the LUTs that the VHDL compare used. This means that when implemented into the processor the 32-bit fast compare sorting unit used nearly 2x the hardware resources that the unaltered processor used. If performance is the only measurement important to a design or the hardware resources are abundant, the 32-bit fast compare unit can be used, otherwise the steep hardware resource cost may make it impractical to implement into a processor design.

4.2.3 16-Bit VHDL Compare Unpipelined

The 16-bit sorting unit with the basic VHDL compare was shown to outperform both the unaltered processor and both the 32-bit sorting units. The implementation of the sorting unit caused a 7x to 11.5x increase in speed at which the code ran. The 16-bit sorting unit not only was able to run at a faster max frequency (10 MHz) than the 32-bit processor but the ability to use two sorting units in parallel improved the performance of the design. The 16-bit sorting unit still has the same problem as the 32-bit sorting units with the max frequency being 16.7x slower than the unaltered processor. Although two 16-bit sorting units were implemented in the design, the hardware resource cost was still less than the 32-bit sorting units. The 16-bit sorting unit saw a 127% increase in hardware resources when compared to the unaltered processor.

4.2.4 16-Bit Fast Compare Unpipelined

The 16-bit sorting unit with the fast compare unit had a slightly smaller performance increase compared to the 16-bit sorting unit with the VHDL compare than its 32-bit counterparts. The decrease in the performance increase between the fast compare unit and VHDL compare in the 16-bit sorting unit is most likely caused by the fact that there were less bits to work with. With the 32-bit units there were 2x the number of bits to optimize when switching from the VHDL compare to the fast compare unit. This is a common occurrence when optimizing hardware, the more hardware to optimize the larger the percent yield of those optimizations. The 16-bit fast compare, however, still saw an increase in the max frequency of .2 MHz when compared to the VHDL compare. Similar to the 32-bit sorting units the 16-bit fast compare came at a steep hardware resource cost. The 16-bit fast compare unit used 2.79x the LUTs that the 16-bit VHDL compare used. When implemented into the processor the 16-bit fast compare sorting unit used 1.6x the hardware resources that the unaltered processor used.

4.2.5 8-Bit VHDL Compare Unpipelined

The 8-bit sorting unit with the basic VHDL compare was shown to outperform all previous processor designs. The implementation of the sorting unit caused a 14.27x to 16.86x increase in speed at which the code ran. The 8-bit sorting unit was able to run at the fastest frequency (11.76 MHz) of any of the processors with an implemented sorting unit. The 8-bit sorting unit was not only able to run at the fastest max frequency but because the benchmark's data size was 8 bits the four sorting units working in parallel also allowed the processor to perform multiple operations during a single clock cycle. This size of the benchmark data and the multiple sorting units allowed the 8 bit sorting unit to have a performance advantage over its 16 and 32 bit counterparts. The 8-bit sorting unit saw a similar LUT usage as the 16-bit sorting unit, with the 8-bit sorting unit using 1.27x the hardware resources as the unaltered processor.

4.2.6 8-Bit Fast Compare Unpipelined

The 8-bit sorting unit with the fast compare unit was shown to perform nearly identically to the 8-bit sorting unit with the VHDL compare. Both designs had a maximum clock frequency of 11.76 MHz and ran the median filter benchmarks in nearly identical times. Similar to how we saw a decrease in the performance increase between the fast compare unit and VHDL compare when going from 32-bit to 16-bit systems a decrease in the performance increase occurred when going from 16-bit to 8-bit systems.

This decrease caused the 8-bit systems to perform nearly identically. This non-change in performance between the two systems is most likely caused by the fact that the optimizations to the hardware were so minuscule that the overall performance did not change. The addition of the fast compare unit did result in an increase in the hardware resources the sorting unit requires. The 8-bit fast compare sorting unit requires 1.57x the LUT resources that the 8-bit sorting unit with the VHDL compare uses. This means that the system with the 8-bit fast compare sorting unit uses 1.49x the LUTs that the unaltered processor uses while the 8-bit sorting unit with the VHDL compare uses only 1.27x the LUTs. Due to the increase in the LUTs required to implement the 8-bit fast compare sorting unit and the identical performance to the 8-bit sorting unit with the VHDL compare there is no reason to use the fast compare unit in the designs with 8 bits.

4.2.7 Pipelined Sorting Units

The pipelined sorting units were shown to outperform the unaltered processor but had a decrease in performance when compared to their unpipelined equivalents. The decrease in performance can be attributed to the implementation of the pipelined sorting units and the type of benchmark being used. The pipelined sorting units were implemented by adding flip-flop registers into the sorting unit to break the sorting unit into 28 different stages. Each of these stages then takes 6ns to run. Since the pipelined sorting units were implemented in an in-order processor, holds were added to the hazard unit to accommodate the extra stages in the execute stage caused by the multi-stage sorting units. These holds were added to the hazard unit to stop the pipeline until the sort instruction has completed in the execute stage. This would cause every sort instruction to take 168ns (28 stages x 6ns per stage = 168ns) to run while all other instructions take 6ns. While this pipelining caused all other instructions to run at a much faster rate when compared to the unpipelined systems it reduced the speed at which sort instructions can be run. The addition of pipelining would then be a great alternative to the unaltered processor or the unpipelined systems when an application uses sort instructions but does not heavily rely on them. However, the benchmark used in testing was heavily reliant on sort instructions which made the pipelined systems perform worse than the unpipelined systems. Although the pipelined systems did not outperform their unpipelined counterparts on this benchmark they still were shown to have a significant performance advantage over the unaltered processor without the significant clock frequency reduction the unpipelined systems had. The clock frequency improvement caused by pipelining the sorting units also did not come at a high hardware resource cost. The flip-flop registers required to meet the unaltered processor clock frequency came at a cost of only 1500 flip-flops. When compared to the nearly 20,000 flip-flops required to implement the entire system, the additional flip-flops is less than 10% additional overhead for timing elements.

Chapter 5

CONCLUSION

This work demonstrates that an addition of a hardware sorting unit to a RISC-V processor has the potential to increase the overall performance of the processor. With the addition of the fast compare units described by Deb and Chaudhury the hardware sorting units can be improved upon to increase the performance of the sorting operations even further. The addition however, of the fast compare units required a large number of hardware resources to implement and provided only a slight performance advantage. This means that the sorting units with fast compares are only practical to use in situations where the speed is of the utmost importance or where the design has extra unused area that can accommodate the fast compare units.

The proposed designs were not only shown to work in simulation but the work was extended to show the results carried over when implemented on the Xilinx Artix-7 FPGA. The sorting units were shown to reduce the runtime of a median filter application by a factor of between 5 and 16 when compared to the original RISC-V processor. The increased speed at which the median filter application was able to run did not come without costs. The maximum clock frequencies of the processors with the implemented hardware sorting units were up to 18x slower than the original processor. The only exception to this was the pipelined processor that was shown to run at the same frequency as the original processor. To see how the various design components proposed in this work compared to the components mentioned in the related work see Appendix B.

5.1 Future Work

A natural next step to the work presented is to keep extending the median filter benchmark to larger image sizes. With image sizes always expanding and current images of size 4096x2160, logging the performance of the proposed designs on larger image sizes would be beneficial. Expanding the benchmarks to larger images requires the use of more hardware resources and longer synthesis and implementation times. The pixel values of the image are stored in a RAM module for the processor to access when needed. So as the image sizes increase, so too does the RAM module which increases the hardware resources required and increases the time Vivado needs to synthesize/implement the design. Currently the benchmarks shown in this work used all the hardware resources the Xilinx Artix-7 FPGA could provide. Given a larger FPGA and a more powerful computer to reduce the compile times, larger images can be benchmarked on the proposed designs.

Another possible addition to the work presented would be to implement the pipelined hardware sorting units on a RISC-V processor that uses out-of-order execution. The processor used in this work was an in-order processor. An in-order processor does not allow for the pipelined hardware sorting unit to be run on the side while other instructions are being run (the optimal way to use the pipelined sorting unit). In order to implement the pipelined sorting unit the processor would need to be converted to an out-of-order processor. This would allow for the pipelined hardware sorting unit to be used to its full potential.

BIBLIOGRAPHY

[1] Artix-7 FPGA Family. `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`.

[2] Cal Poly Github. `http://www.github.com/CalPoly`.

[3] Nexys 4 DDR. `https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start`.

[4] Nexys 4 DDR Schematic. `https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-4-ddr/nexys-4-ddr_sch.pdf`.

[5] Phelmino Mircoprocessor Github. `https://github.com/phelmino/phelmino`.

[6] Vivado Design Suites. `https://www.xilinx.com/products/design-tools/vivado.html`.

[7] A. Abdelgawad and M. Bayoumi. High speed and area-efficient multiply accumulate (mac) unit for digital signal prossing applications. In *2007 IEEE International Symposium on Circuits and Systems*, pages 3199–3202. IEEE, 2007.

[8] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[9] G. Bergland. Fast fourier transform hardware implementations–an overview. *IEEE Transactions on Audio and Electroacoustics*, 17(2):104–108, 1969.

[10] B. Betkaoui, D. B. Thomas, and W. Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101. IEEE, 2010.

[11] S.-W. Cheng. A high-speed magnitude comparator with small transistor count. In *10th IEEE International Conference on Electronics, Circuits and Systems, 2003. ICECS 2003. Proceedings of the 2003*, volume 3, pages 1168–1171. IEEE, 2003.

[12] A. Colavita, A. Cicuttin, F. Fratnik, and G. Capello. Sortchip: A vlsi implementation of a hardware algorithm for continuous data sorting. *IEEE Journal of Solid-state circuits*, 38(6):1076–1079, 2003.

[13] S. Deb and S. Chaudhury. High-speed comparator architectures for fast binary comparison. In *2012 Third International Conference on Emerging Applications of Information Technology*, pages 454–457. IEEE, 2012.

[14] S. Dong, X. Wang, and X. Wang. A novel high-speed parallel scheme for data sorting algorithm based on fpga. In *2009 2nd International Congress on Image and Signal Processing*, pages 1–4. IEEE, 2009.

[15] R. Espasa. Risc V Vector Extension Proposal.

[16] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton. Modular high-throughput and low-latency sorting units for fpgas in the large hadron collider. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 38–45. IEEE, 2011.

[17] R.-V. Foundation. Getting started with risc-v, 2019.

[18] G. K. Gultekin and A. Saranli. An fpga based high performance optical flow hardware design for computer vision applications. *Microprocessors and Microsystems*, 37(3):270–286, 2013.

[19] M. Karaman, L. Onural, and A. Atalar. A general purpose vlsi median filter and its applications for image processing. In *Proceedings. Electrotechnical Conference Integrating Research, Industry and Education in Energy and Communication Engineering',*, pages 366–369. IEEE, 1989.

[20] G. Lin and B. Shi. A expansible current-mode sorting integrated circuit for pattern recognition. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*, volume 5, pages 3123–3127. IEEE, 1999.

[21] Z. Long and Z. Zhang. Fpga-based collaborative hardware sorting unit for embedded data processing system. In *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, pages 260–264. IEEE, 2017.

[22] M. M. Mano. *Digital logic and computer design.* Pearson Education India, 2017.

[23] R. Marcelino, H. Neto, and J. M. Cardoso. Sorting units for fpga-based embedded systems. In *IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 11–22. Springer, 2008.

[24] R. Marcelino, H. C. Neto, and J. M. Cardoso. A comparison of three representative hardware sorting units. In *2009 35th Annual Conference of IEEE Industrial Electronics*, pages 2805–2810. IEEE, 2009.

[25] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of

fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[26] B. Parhami and D.-M. Kwai. Data-driven control scheme for linear arrays: application to a stable insertion sorter. *IEEE Transactions on Parallel and Distributed Systems*, 10(1):23–28, 1999.

[27] R. Perez-Andrade, R. Cumplido, C. Feregrino-Uribe, and F. M. Del Campo. A versatile linear insertion sorter based on an fifo scheme. *Microelectronics Journal*, 40(12):1705–1713, 2009.

[28] J. Suomela. Median filtering is equivalent to sorting. *arXiv preprint arXiv:1406.1717*, 2014.

[29] S. I. Waterman, Asanovi. *The RISC-V Instruction Set Manual*. University of California, Berkeley.

[30] R. Woods, J. McAllister, G. Lightbody, and Y. Yi. *FPGA-based implementation of signal processing systems*. Wiley Online Library, 2017.

[31] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core gpus. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[32] Y. Zhang and S. Zheng. An efficient parallel vlsi sorting architecture. *VLSI Design*, 11(2):137–147, 2000.

APPENDICES

Appendix A

AREA USAGE OF COMPONENTS

Table 6 shows the number of Look-Up-Table used to implement a few of the components in the processor.

Table 6: Area Usage of Processor Components

Processor Component	Look-Up-Table Usage
32-Bit Fast Compare Unit	18514
32-Bit Hardware Sorting Unit	9145
16-Bit Fast Compare Unit	6307
16-Bit Hardware Sorting Unit	4132
Vector Register File	3054
8-Bit Hardware Sorting Unit	2884
General Purpose Register File	1665
8-Bit Fast Compare Unit	1015
Prefetch Buffer	209
Arithmetic Logic Unit (ALU)	19
Decoder Unit	18
32-Bit VHDL Compare	4
16-Bit VHDL Compare	2
8-Bit VHDL Compare	1

Appendix B

RELATED WORK RESULTS COMPARISON

Table 7 shows the various metrics of the hardware sorting unit designs proposed in this paper and the metrics of the similar components from the papers discussed in the related work. Table 8 shows the metrics of the proposed fast compare units compared to the results achieved in the papers discussed in the related work. Not all metrics were listed in the related work papers so only the information posted will be compared.

Table 7: Hardware Sorting Metrics Compared to Results in the Related Works

Sorting Unit	LUT Usage	Total Latency
32-to-4 Sorting Units [16]	2714	40 ns
16-to-4 Sorting Units [16]	1199	25.2 ns
Folded Sorting Units (128 Elements) [24]	14923	Not Provided
Insertion Sorting Units (128 Elements) [24]	16420	Not Provided
Balanced FIFO Sorting Units (1k Elements) [24]	362	Not Provided
32-Bit Sorting Unit (Proposed)	7024	110 ns
32-Bit Fast Compare Sorting Unit (Proposed)	25546	107 ns
16-Bit Sorting Unit (Proposed)	3420	100 ns
16-Bit Fast Compare Sorting Unit (Proposed)	9827	99 ns
8-Bit Sorting Unit (Proposed)	1762	85 ns
8-Bit Fast Compare Sorting Unit (Proposed)	2777	85 ns

Table 8: Fast Compare Unit Metrics Compared to Results in the Related Works

Processor Component	Latency Improvement Over Traditional
32 Bit Fast Compare Unit [13]	7 ns reduction
32 Bit Fast Compare Unit (Proposed)	3 ns reduction
16 Bit Fast Compare Unit (Proposed)	1 ns reduction
8 Bit Fast Compare Unit (Proposed)	0 ns reduction