DESIGN AND IMPLEMENTATION OF AN INSTRUCTION SET ARCHITECTURE AND AN INSTRUCTION EXECUTION UNIT FOR THE REZ9 COPROCESSOR SYSTEM.

By

Daniel Anderson

Bachelor of Science in Computer Engineering

University of Nevada Las Vegas

2011

A thesis submitted in partial fulfillment

of the requirements for the

Master of Science in Engineering - Electrical Engineering

Department of Electrical and Computer Engineering

Howard R. Hughes College of Engineering

The Graduate College

University of Nevada Las Vegas

December 2014

Copyright by Daniel Anderson, 2015

All Rights Reserved

We recommend the thesis prepared under our supervision by

Daniel Spencer Anderson

entitled

Design and Implementation of an Instruction Set Architecture and an Instruction Execution Unit for the REZ9 Coprocessor System

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering
Department of Electrical Engineering

R. Jacob Baker, Ph.D., Committee Chair

Venkatesan Muthukumar, Ph.D., Committee Member

Henry Selvaraj, Ph.D., Committee Member

Evangelo Yfantis, Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

December 2014

ABSTRACT

While the use of RNS has provided groundbreaking theory and progress in this field, the applications still lack viable testing platforms to test and verify the theory. This Thesis outlines the processing of developing an instruction set architecture (ISA) and an instruction execution unit (IEU) to help make the first residue based general processor a viable testing platform to address the mentioned problems.

Consider a 32-bit ripple adder. The delay on this device will be 32N where N is the delay for each adder to complete its operation. The delay of this process is due to the need to propagate each carry signal generated by each adder to the next one. This was solved by the creation of the Carry Look Ahead (CLA), which could drastically reduce the delay by 2/3. However, like the ripple adder, the CLA is still encumbered by propagation delay. A residue processor in the same situation would have a delay of 1N regardless of bit size since carry propagation is no longer a concern.

The Thesis discusses how prior challenges using residue number systems in computers has been overcome by Digital System Research (DSR).

TABLE of CONTENTS

ABSTRACT ii
LIST of TABLES vi
LIST of FIGURES vii
CHAPTER 1: INTRODUCTION AND MOTIVATION 1
1.1 BACKGROUND 1
1.2 MOTIVATION 1
1.3 THESIS ORGANIZATION 2
CHAPTER 2: BACKGROUND ON THE RESIDUE NUMBER SYSTEM 3
2.1 RESIDUE NUMBER REPRESENTATION 3
2.2 RESIDUE NUMBER CONVERSIONS 5
2.2.1 PAIRWISE PRIMES 5
2.2.2 MIXED RADIX CONVERSIONS 6
2.3 FRACTIONAL REPRESENTATIONS 10
CHAPTER 3: REZ9 COPROCESSOR BACKGROUND 12
3.1 DEVELOPMENT ENVIRONMENT 12
3.2 NIOS II PROCESSOR 12
3.3 REZ9 ALU ARCHITECTURE 14
3.4 REGISTER FILES 15
3.4.1 ADDRESSING MODES 15
3.4.2 STATUS REGISTERS 17
CHAPTER 4: INSTRUCTION SET ARCHITECTURE 20
4.1 DESIGN METHODOLOGY 20
4.2 INSTRUCTION TYPES 21
4.2.1 DATA PROCESSING AND ARITHMETIC INSTRUCTIONS 21
4.2.2 LOAD, STORE, AND MOVE INSTRUCTIONS 23
4.2.3 CONVERSION INSTRUCTIONS 24
4.2.4 SIGNED INSTRUCTIONS 25
4.3 INSTRUCTION ENCODING 27
4.5 PARALLELISM 28
CHAPTER 5: INSTRUCTION EXECUTION UNIT 30
5.1 OVERVIEW 30

5.2 SOFTWARE IEU 30
5.3 HARDWARE IEU IMPLEMENTATION 31
5.4 QSYS SYSTEM INTEGRATION TOOL 32
5.5 NIOS CUSTOM INSTRUCTION BACKGROUND 34
CHAPTER 6: SIMULATION & TESTING RESULTS 38
6.1 VERILOG SIMULATION RESULTS 38
6.2 ECLIPSE IDE TEST ROUTINES & RESULTS 41
6.2.1 ARITHMETIC ALU TESTS 42
6.2.2 FRACTIONAL MULTIPLY TESTS 43
6.2.3 COMPARISON TESTS 43
6.3 MANDELBROT TEST ROUTINE AND RESULTS 44
CHAPTER 7: CONCLUSION 46
APPENDIX A: VERILOG CODE 47
APPENDIX B: ALU_INSTRUCT.C 111
APPENDIX C: ALU_INSTRUCT.H 135
APPENDIX D: ARITHMETIC ALU TESTS 137
APPENDIX E: ARITHMETIC ALU TEST CODE 139
APPENDIX F: FRACTIONAL MULTIPLY TEST 142
APPENDIX G: FRACTIONAL MULTIPLY TEST CODE 145
APPENDIX H: LOAD AND STORE TEST CODE 150
APPENDIX I: LOAD AND STORE TEST RESULTS 152
APPENDIX J: COMPARISON TEST 153
APPENDIX K: COMPARISON TEST CODE 160
APPENDIX L: MANDELBROT.C 165
APPENDIX M: MANDELBROT TEST CODE & RESULTS 174
APPENDIX N: COMPARISON OF PAIRWISE VS. NON-PAIRWISE MODULI 178
APPENDIX O: DATA PROCESSING AND ARITHMETIC INSTRUCTIONS 179
APPENDIX P: DATA CONVERSION INSTRUCTIONS 180
APPENDIX Q: REZ9-A COPROCESSOR 181
REFERENCES 182
VITA 183

LIST of TABLES

Table 1: Residue Representation of the Numbers 0 to 29 for Moduli 2, 3, 5 4
Table 2: Modulo-5 Addition, Subtraction, and Multiplication Tables 4
Table 3: Mixed Radix Value Calculations 7
Table 4: Mixed-Radix Number System Example 8
Table 5: Conversion from Residue to Mixed-radix Representation 10
Table 6: ASR Table 18
Table 7: CSR Table 19
Table 8: Parallel Data Processing and Arithmetic Instructions 23
Table 9: Load, Store, and Move Instructions 24
Table 10: Altera Custom Instruction Format 27

LIST of FIGURES

Figure 1: NIOS II Processor 12
Figure 2: Example of a NIOS II Processor System 14
Figure 3: Instruction Execution Unit Model 30
Figure 4: IEU and REZ9 ALU Interface 32
Figure 5: ALU Bus Selector for REZ9 33
Figure 6: ALU Port Bus Connections 34
Figure 7: Hardware Block Diagram of a NIOS II Custom Instruction 36
Figure 8: Multicycle Custom Instruction Timing Diagram 37
Figure 9: Waveform Simulation-Addition 39
Figure 10: Waveform Simulation-Load 39
Figure 11: Waveform Simulation-Integer Multiply 40
Figure 12: Waveform Simulation-Store 40
Figure 13: Waveform Simulation-Subtraction 41

CHAPTER 1: INTRODUCTION AND MOTIVATION

1.1 BACKGROUND

One of the conventions of computer architecture is that computer systems run in binary. The reality is that the binary number system is used in computer architecture design because it has been the most successful number system to date; however, this does not mean that a computer cannot be designed around another numerical system. For example, binary coded decimal BCD, is a numerical system often used in cases where applications only require digits zero to nine, such as calculators. The use of BCD can be costly when performance is advantageous due to the wasted bit representation. [5] However, this is an example where sacrificing the usefulness of a binary number system can be beneficial.

1.2 MOTIVATION

Specifically, one of the main problems facing computer engineers today is the width, or word size, of data. Generally, within a given architecture, the word size of a computer is fixed. A problem then arises when an application begins to deal with data that is wider than a computer's words size. In the prior art larger word sizes are handled by breaking apart data to fit into a computer's set word size. However, this method results in increased execution time that grows exponentially as the word size of the data grows; Eventually it is no longer efficient to use a dedicated computer width for applications that often require extremely word sizes that will exceed the set width. This “slowdown” in processing is often seen when dealing carry propagation within the computer’s arithmetic logic unit (ALU). [3]

To combat these claims research was performed to find a solution to this width size and propagation problem. One solution was to create an architecture that no longer depended on carry and propagation to compute data. The Residue Number System (RNS) provide an ideal environment where carry and propagation are not necessary for computation. This idea led to the research and creation of the first general purpose residue based processor.

1.3 THESIS ORGANIZATION

This Thesis details the design, implementation and testing of an instruction set architecture and instruction execution unit designed for the REZ9: the first residue-based general processor. Chapter 2 focuses on the background theory of the RNS and how its unique characteristics can be beneficial to an arithmetic processor. Chapter 3 outlines the background and current progress on the REZ9 coprocessor system, providing insight on design decisions and overall architecture. Chapter 4 focuses on the creation and development of the instruction set architecture. It provides created instructions along with motivation behind decision, and it provides specific details on how the instructions were designed benefit the REZ9 system. Chapter 5 details the creation and development of the instruction execution unit and details on how the IEU integrates with the REZ9. Chapter 6 provides background behind testing results and simulations along with interpretations and conclusions revealed from the results. Chapter 7 concludes this Thesis with the status of the REZ9 system since the integration of the ISA and IEU. This chapter also details future work within the development of the REZ9 coprocessor.

CHAPTER 2: BACKGROUND ON THE RESIDUE NUMBER SYSTEM

2.1 RESIDUE NUMBER REPRESENTATION

Traditionally, numerical systems are radix based such as decimal (base 10) and binary(base 2). Residue is a non-radix, modulus based, number system. This means it does not positional notation such as the tens place in decimal. This characteristic of residue allows traditional operations such as addition, subtraction, and multiplication to be performed without carry. Applying this to digital circuitry, mathematical operations done in the REZ9 can be performed in a single clock cycle without carry propagation. This has led to designs that can rival other solutions for carry propagation, such as carry-lookahead adders (CLA).

For example, the number 25 is represented as two in the 10s place and five in the 1's place in the decimal number system. In the residue number system, representation of a number is based on the selected moduli. Because the selected moduli are not static like the radix in other number systems, the representation of a residue number can change by changing the selected set of moduli. This allows residue numbers to have multiple representations, a trait not present in radix-based systems.

Consider the numbers 2, 3, and 5, as a set of moduli. Performing the mod function on each number will result in the residue representation of 25. 25 mod 2 results in 1; 25 mod 3 results in 1 and 2 mod 5 results in 0. Therefore, the residue representation of 25 is {1, 1, 0}. This representation is only valid for the selected moduli. A change in any one of the moduli will change the number representation. For example, if we choose to use the modulus 7 instead of 2 the representation would be {4, 1, 0}. Table 1 provides additional examples of representing numbers using moduli of two, three, and five.

INTEGERS	RESIDUE DIGITS MODULI			INTEGERS	RESIDUE DIGITS MODULI		
	2	3	5		2	3	5
0	0	0	0	15	1	0	0
1	1	1	1	16	0	1	1
2	0	2	2	17	1	2	2
3	1	0	3	18	0	0	3
4	0	1	4	19	1	1	4
5	1	2	0	20	0	2	0
6	0	0	1	21	1	0	1
7	1	1	2	22	0	1	2
8	0	2	3	23	1	2	3
9	1	0	4	24	0	0	4
10	0	1	0	25	1	1	0
11	1	2	1	26	0	2	1
12	0	0	2	27	1	0	2
13	1	1	3	28	0	1	3
14	0	2	4	29	1	2	4

Table 1: Residue Representation of the Numbers 0 to 29 for Moduli 2, 3, 5 [1]

The real benefit of residue in computing systems is its use in arithmetic instructions. For example, performing the operation 2+3 will return a value of five. Given a modulus of four, the residue representation is generated by dividing 5 by 4 and taking the remaining value as the representation.

ADDITION					
+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	5	1
3	3	4	0	1	2
4	4	0	1	2	3

SUBTRACTION						
	-	0	1	2	3	4
Minuend	0	0	1	2	3	4
	1	1	2	3	4	0
	2	2	3	4	5	1
	3	3	4	0	1	2
	4	4	0	1	2	3

MULTIPLICATION					
x	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	5	1
3	3	4	0	1	2
4	4	0	1	2	3

Table 2: Modulo-5 Addition, Subtraction, and Multiplication Tables [1]

By applying this unique feature of residue to processor technology, digital arithmetic without carry is possible. In a binary-based processor, most arithmetic operations require some level of carry. This carry causes operations to have to wait for each digit to finish its operation so that it can propagate through to the next digit. In the REZ9 design, waiting for digits to propagate has been eliminated since all arithmetic operations are performed in residue. This allows operations such as addition, subtraction, and multiplication to be performed in a single single-clock cycle regardless of the number of digits in the value.

2.2 RESIDUE NUMBER CONVERSIONS

2.2.1 PAIRWISE PRIMES

While it has been established how to represent numbers into residue, it is critical to convert numbers from residue as well. However, this creates a unique problem where a single residue number can represent multiple values if the same certain moduli are used or if the range set by the moduli is exceed. For example, given the moduli 3, 5, and 7 it is apparent that the only common factor among any two of them is one. To find the unique number of residue representations, one would multiply all of the moduli and divide them by each modulus. Observation of 105, the product of the three moduli, it is apparent that it cannot be evenly divided by any of these moduli more than once. Therefore, from 0 to 104, every residue number will be unique, but at 105, the sequence will start over, making the residue representation of 0 and 105 identical for the given moduli. Table 1 shows a representation of residue number from 0 to 29. If the table continued beyond its

range of 29 and went to 30 the table would reflect that both 0 and 30 have the same representation {0,0,0}.

Now change the set of moduli to 4, 8, and 12. On the surface, the number of representations appears larger with the product of these moduli being 384. However, when the product is divided by four, what is left is a result that be divided by four again. This creates an overlap in representations. To find the actual number of representations, the product (384) must be divided by the product of the greatest common factor of each pair of moduli. Applying it to this example, results in 384/ (4*4) = 24 which is an accurate number of representations. Inspection of the tables in Appendix N supports these claims. Comparing the tables reveals that the number of unique representation with a set of moduli is greater when the set of moduli only share the common factor of 1. A set of moduli that satisfy this condition is considered Pairwise Prime. Pairwise prime numbers are the key component in successful conversions.

2.2.2 MIXED RADIX CONVERSIONS

The Chinese Remainder Theorem (CRT) is a mathematical process used for converting residue numbers. In the prior art, most study in the field of residue numbers use CRT with great success. However, during the development of the REZ9 processor, it was determined that while mathematically CRT was suitable for residue conversion, it presented various problems when attempting to use residue in the realm of computer processing.

To compensate for this, another technique was developed that does not rely on the CRT. This method is called Mixed Radix Conversion (MRC). MRC provides two key features necessary for residue conversion. First, it provides a weighted number system,

which acts as an intermediate step when converting residue numbers. The second key feature is that mixed-radix conversion is relatively fast when used in residue-based computers, an ability that is not present when using the CRT.

CONVERSION OF RNS TO BINARY USING MIXED RADIX CONVERSION

When using place-value notation, numbers are often dependent on a single base to compute it digit value. For example, in the decimal number system, the value of each number is a digit multiplied by a power 10. Given the number 4165, it breaks down to the numerical equivalent of $4(10^3) + 1(10^2) + 6(10^1) + 5(10^0)$. Taken a step further, these powers of 10 break down into sections making the equation become 4(10 x 10 x 10) + 1(10 x 10) + 6(10) + 5(1). The mixed radix number system uses different values in

Radices			Digits	Weighted Equation	Weighted Value
R1	2		a1	1	1
R2	3		a2	1*2	2
R3	5		a3	1*2*3	6

Table 3: Mixed Radix Value Calculations

From Table 3 it becomes clear that given a set of pairwise prime radices, a set of mixed radix digits will be generated from the numbers. For example, given the radices, R1 = 2, R2=3 and R3 = 5, three mixed radix digits will be generated. The first step is assigning weights for each of the digits. Just like other numerical systems, the weight of the first digit (named a1) is always one. The second digit (a2) is the product of the a1 and first radix (R1). The third digit is the product of a2 and R2. Perhaps oddly, the last digit does not reference the last radix R3 by a product. Instead, the fourth digit may assume a

legal value within the range of the last radix, R3. However, the number of accessible digits is equal to the number of residue digits. Therefore if a residue value consists of only 3 residue digits (i.e. moduli) then the mixed-radix value will also only contain 3 digits.

Number	Mixed-Radix Digits		
	a3	a2	a1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	0	2	0
5	0	2	1
6	1	0	0
7	1	0	1
8	1	1	0
9	1	1	1
10	1	2	0
11	1	2	1
12	2	0	0
13	2	0	1

Table 4: Mixed-Radix Number System Example [1]

CONVERSION FROM RESIDUE TO MIXED RADIX REPRESENTATION

The REZ9 ALU was developed to function based on any selected moduli. A main design goal is to find a set of modulus that will define a suitable range for the designated application. At the same time, it is crucial to select a set of moduli that can accomplish other objectives such as high encoding efficiency, high implementation efficiency, and support for power based modulus for advance applications such as division. For example, the set {2,3,5} will generate 30 representations in residue. By simply changing the 2 into a 4 the new set {4,3,5} will generate 60 representations. The

new set still maintains the rule of primes since they only share a common factor of 1. Using a power of a prime modulus allows the REZ9 to support a “power based digit architecture” as defined in [3].

With the concept of range in mind, converting a residue value to a mixed radix value requires that a proper set of moduli have been selected. Take the example of converting the residue value of {3, 4, 2, 1} into a mixed radix number. If given a set of radices (8, 5, 7, 3) for the mixed radix format, it can be determined that the final answer will appear in the following form: a1 (8*7*5) +a2 (8*7) +a3 (8) +a4 = x, where x is the natural number value of the mixed radix number.

For this selecting the moduli of 8,5,7,and 3 provides a set that still maintains the pairwise prime status, but it also has an increased range for residue numbers. The method of converting a residue number to a mixed radix number involves subtracting values to zero out specific moduli (similar to polynomial division.) Using the example modulus set above, this results in a mixed radix representation of {1, 5, 2, 3}. Using the supplied radices it can be determined that the final value is: a1(8*7*5) +a2(8*7) +a3(8) +a4 = 1(8*7*5) +5(8*7) +2(8) +3 = 499

	Moduli:	8	5	7	3
a1 = 3		3	4	2	1
subtract a1 = 3	-	3	3	3	3
		0	1	6	1
multiply 1/8	x		2	1	2
a2 = 2			2	6	2
subtract a2 = 2	-		2	2	2
			0	4	0
multiply 1/5	x			3	2
a3 = 5				5	0
subtract a3 = 5	-			5	2
				0	1
multiply 1/7	x				1
a4 = 1					

Table 5: Conversion from Residue to Mixed-radix Representation

2.3 FRACTIONAL REPRESENTATIONS

All of the discussed information regarding residue is related to integer operations. In the prior art, the majority of residue conversions focused on the use of integer value only. It was deemed by the majority of the RNS community that fractional values in residue do not exist, or at least the difficulty of performing fractional operations in a residue format is too complex or too burdensome. However, research performed at Digital System Research by Olsen [3] has tackled these issues. The research resulted in the creation of fractional operations that were useful, accurate, and extendable. Regarding the REZ9, the residue values are manipulated and processed using mixed radix conversions. The process used in the prior art only applied to basic integers operations. To handle arithmetic of fractional values, new techniques and processes were created and implemented in hardware. .

The process for performing residue computations using fractional values is discussed in detail in [3]. A simple explanation of performing a fractional multiplication

is as follow: Given two fractional residue numbers to compute, the first process is to treat them as integers and simply multiply the two values. This first multiplication will generate what is referred to as the intermediate product. This intermediate product is divided by a quantity known as the fractional range. This range is used as a method of scaling this intermediate product. Once this scaling is complete, the intermediate product can be converted into mix radix format. Once in the mixed radix format, the intermediate product is scaled, or truncated, before it is converted back to residue into its final format. It is crucial to realize that this form of "division" is meant to perform normalization of the intermediate residue result, such that the result exists in the same fractional representation of the operands.

Fractional residue numbers are formed by grouping a residue number into a product of two distinct ranges. One range is the fractional range, which is defined by the range of the associated (and so designated) fractional digits. The other range is the integer range, which is defined by the range of the associated integer digits. By default, fractional residue numbers are represented by having their integer component multiplied by a fractional range (similar to how a fraction is a ratio of two integers in the decimal system.) A similar example is multiplying two decimal numbers. The first step is to find the actual magnitude of the numbers and then find the proper location of the decimal point. The key is to find proper representation and then perform the necessary operations and conversions. Additional details can be found in [3].

CHAPTER 3: REZ9 COPROCESSOR BACKGROUND

3.1 DEVELOPMENT ENVIRONMENT

The development of the REZ9 and its IEU consist of two major portions. The starting portion of the development was done within Altera's Quartus II software. The initial development and design of the REZ9 coprocessor was done using a Field Programmable Gate Array (FPGA). The initial testing and development was executed on a Stratix IV GX FPGA Development Board, (EP4SGX530KH40,). This design was later condensed and moved to a DE2-115 Development and Education Board (EP4CE115F29C7.)

3.2 NIOS II PROCESSOR

The REZ9 is a binary-residue hybrid coprocessor system. The binary component consists of Altera's NIOS processor and memory systems. The NIOS II Processor is an embedded soft processor developed by Altera. The NIOS II contains three configurable 32-bit Harvard architecture cores. [7]

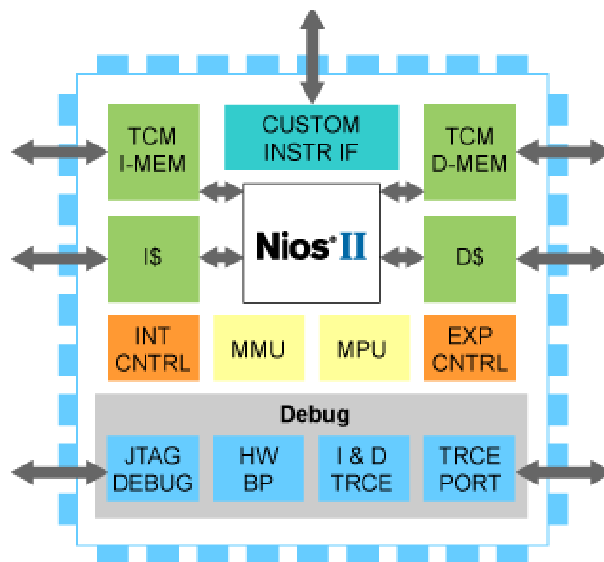


Figure 1: NIOS II Processor [7]

The NIOS II is a configurable soft processor IP core. This differs from a standard off-the-shelf processor since it grants the ability to add and remove features per design specifications. The example in figure 2 shows a possible outline design configuration that taps into the NIOS II. In this particular design, the NIOS II has several peripherals such as timers, LCD drivers, and UART connections. Because the NIOS is a soft processor, it can be compiled to interface with all of said peripherals. One benefit of using a soft processor is that it only uses any many resources as requested. Therefore, by integrating the NIOS II design with the desired peripherals, it can lead to a design specific device that takes up as little or as much resources as desired. In regards to the REZ9 development, the NIOS II is primarily used to control the function of the residue ALU. Another benefit of using a soft processor with the REZ9 ALU is that it allows access to testing various features within the FPGA platform for future expansion and experimentation. For example, to test the versatility of the REZ9, the initial version was created on a Stratix IV GX FPGA (EP4SGX530KH40). Later the original version was ported to a smaller FPGA for size comparison and portability purposes (Cyclone IV FPGA EP4CE115F29C7.) This presented two testing platforms to prove that changing of resources such as dedicated memory and number of logic gates does not adversely affect the overall performance of the processor.

The NIOS II processor handles all binary arithmetic instructions. Operations such as fetching instructions and updating the program counter (PC) are performed by the NIOS II processor. The NIOS II processor is also responsible for high-level conversions, such as floating point and fixed point values that are then transferred to the residue section to be converted and executed by the residue ALU.

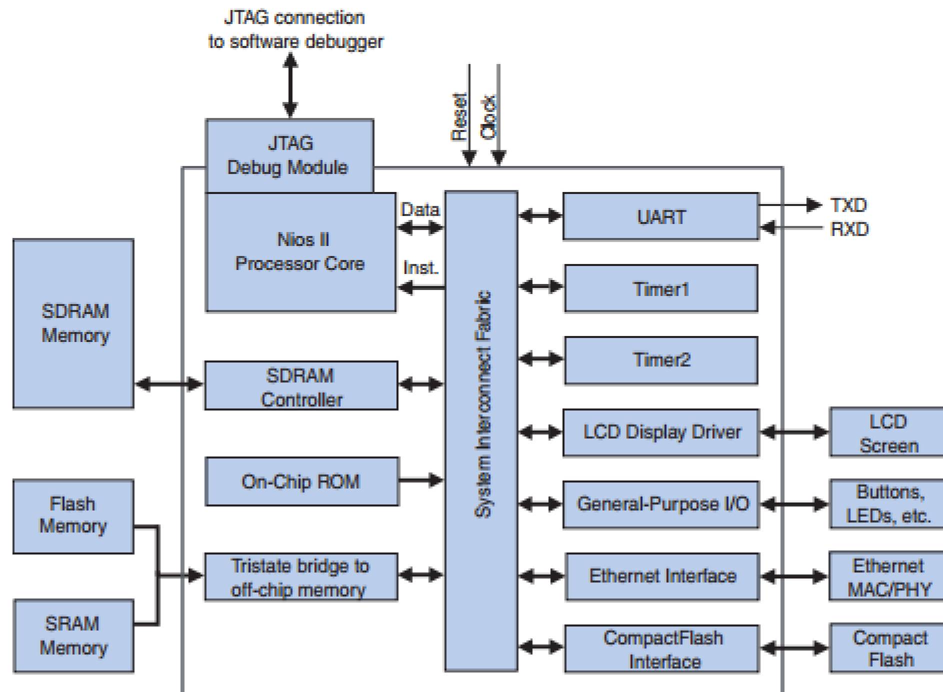


Figure 2: Example of a NIOS II Processor System [7]

3.3 REZ9 ALU ARCHITECTURE

The (RNS) ALU, which was developed by Digital Systems Research (DSR), is used for all of the arithmetic instructions supported by the REZ9-A. The RNS ALU system contains dual ALUs and dual accumulators. These RNS based accumulators are18 digits wide. The system contains an 18 digit, dual ported RNS register file. The dual ported Register file allows both accumulators to access RNS data simultaneously. This design allows for the use of dual instructions to be executed, giving the processor Single Instruction Multiple Data capabilities. Included in the RNS section are three conversion units that are supported by the REZ9. Appendix Q contains the layout of the coprocessor system.

There are two forward conversion units: one integer and one fractional forward conversion unit. Both forward converters are used to convert fixed-point binary values into RNS values. These fixed-point binary values are generated when the NIOS II routines convert floating-point values into fixed-point values. Once they are converted into residue these values can be computed by the residue ALUs. The reverse converter used to convert values back to binary. The converters themselves are connected directly to the NIOS processor so binary values can be immediately sent or received once they are ready. Special conversion assembly language instructions will be introduced later which allow the REZ9 to rapidly perform forward and reverse conversion at assembly language speed.

3.4 REGISTER FILES

The REZ9 coprocessor contains 1024 registers. These registers can be configured to be either 18- or 32-digits wide. However, the mode of access is dependent on the addressing mode that is currently set.

3.4.1 ADDRESSING MODES

The REZ9 supports three addressing modes, 1) register direct mode, 2) register indirect mode, and 3) mixed mode. To activate register direct mode, bit readrx, readry, and writez are set to zero. In this mode, the REZ9 registers accessed are located locally in one of the REZ9 registers. While this mode only offers 32 available registers, register direct mode provides the fastest execution for any REZ9 instructions. With all relevant data present, instructions can be executed immediately instead of loading data from the NIOS first.

Register indirect mode allows the REZ9 instructions to reference registers within the NIOS II processor. By enabling bits, readrx, readry, writez, instructions will refer to rX, rY, and rZ fields to determine the appropriate register address for the called instruction. Register indirect mode has must load all relevant data before instruction execution can begin. However, it has access to all 1024 registers of the REZ9.

The mixed register access mode enables certain REZ9 instructions to access the registers of the NIOS II processor. It also grants access data directly from the instruction based on how the readrx, ready, and writez bits are set. Mixed register access was designed for certain hybrid instructions that will be developed in the future.

Every register also has a section dedicated to sign bits and skipped digit flags. Two sign bits relay two pieces of information. One bit, the sign flag, signifies what the current sign of the stored value is; the second bit, the sign valid bit, reveals whether or not the sign bit is valid. There are some instances when errors may occur that may alter the flag. As a verification protocol the valid sign bit acts as a secondary check for any operations that are sign dependent. If the sign is not valid, but must be known, then the REZ9 must perform a sign extension operation to validate the sign bit, which also sets the sign valid bit to true.

The skipped digit flags are used to check if a digit is defined. For example, during Mixed Radix conversion, once a digit modulus is divided out, it is no longer valid. However, the processor must still properly interpret the remaining RNS value (the valid digits), and therefore, the skip digit flag allows the REZ9 to correctly interpret the state of the remaining “valid” RNS digits. This is an advanced feature, which allows the REZ9 processor the ability to process and store partially extended residue values [2][3].

3.4.2 STATUS REGISTERS

The status registers were developed to provide access to the status of the RZ9 ALU. These registers are considered to be "read only" as a user (programmer) would not be able to modify these registers. While the status registers are a feature of the REZ9, their importance and design are directly influenced the creation of the ISA.

Accumulator Status Register

The accumulator status register (ASR) is used to track the current state of both accumulators within the REZ9 ALU. This shared status register keeps track of vital information regarding the accumulators. For example, the status register can inform when the accumulators are equal to zero or whether the values in accumulators A and B are the same. The status register also monitors smaller bits of information such as if any of the digits in the accumulator are zero.

Comparator Status Register

The comparator status register is used to hold the results of the comparison instruction. Since the instructions are accumulator based, the comparator is always comparing the accumulator with a secondary source. It should be mentioned that the comparison instruction still returns the result of a comparison. It is intended that the result of the comparison operation affect the comparison status register. The purpose of the comparison status register is to maintain this result so it is not lost until the REZ9 queries the result of the comparison in a subsequent instruction execution. A done flag is included for each comparison status of the A and B accumulator. For example, given a routine that wishes to take the larger of two numbers and use if for another calculation,

by checking the comparator done flag, the REZ9 knows to wait until the comparison is finished before attempting to select witch value is required.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
--	Sign B	Sign Valid	--	Any skip	Any Zero	B=1	B=0	A=B	Sign A	Sign Valid	--	Any skip	Any zero	A=1	A=0

Table 6: ASR Table [2]

Accumulator status registers condition codes from DSR manual.

Bits 0-7 are mapped to ALU A.

A =0: Set when all valid digits of the accumulator A are zero.

A =1: Set when all valid digits of the accumulator A are one.

Any Zero: Set when there is at least one valid digit equal to zero in accumulator A.

Any skip: Set when there is at least one digit with a skip digit flag set in accumulator A.

Sign valid: Set when the sign A bit is valid.

Sign A: Set to indicate negative, otherwise, clear to indicate a positive number.

A=B: Set when all double valid digits are equal.

The bits 15-8 are mapped to ALU B.

B =0: Set when all valid digits of the accumulator B are zero.

B=1: Set when all valid digits of the accumulator B are one.

Any Zero: Set when there is at least one valid digit equal to zero in accumulator B.

Any skip: Set when there is at least one digit with a skip digit flag set in accumulator B.

Sign valid: Set when the sign B bit is valid.

Sign B: Set to indicate negative, otherwise, clear to indicate a positive number.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Done B	--	--	--	B = mr	B > mr	B = reg	B > reg	Done A	--	--	--	A = mr	A > mr	A = reg	A > reg

Table 7: CSR Table [2]

The comparator status codes are:

A > reg: The accumulator A is larger than the register value

A = reg: The accumulator A is equal to the register value

A > mr: The accumulator A is greater than the mixed radix constant

A = mr: The accumulator A is equal to the mixed radix constant

Done A: The compare a process is complete, a status codes are valid

B > reg: The accumulator B is larger than the register value

B = reg: The accumulator B is equal to the register value

B > mr: The accumulator B is greater than the mixed radix constant

B = mr: The accumulator B is equal to the mixed radix constant

Done B: The compare B process is complete, B status codes are valid

CHAPTER 4: INSTRUCTION SET ARCHITECTURE

4.1 DESIGN METHODOLOGY

The ISA was developed specifically to support the REZ9-A architecture. The ISA is based on a hybrid ISA, with the majority of the instructions performing simple, single actions on an accumulator, making the overall REZ9-A a reduced instruction set computer (RISC). However, the REZ9 does support some complex instructions as well. The ISA was developed to support a dual accumulator based design, which supports three threads of simultaneous instruction execution. Two of these threads are accumulator based with the third thread supporting register-to-register instructions that mirror basic MIPS instructions.

The choice of using an accumulator based architecture stemmed from the philosophy that short, simple accumulator based instructions are quick to decode, and common in arithmetic calculations, such as product summing. This again ties to the overall motivation of capitalizing on the benefits of the REZ9 architecture instead of simply mirroring what works best in a binary architecture. Much of the growth of the original REZ9 is a reflection of this. In the initial design by DSR, the REZ11 used only a single accumulator with the intent of supporting an absolute baseline architecture. In a second-generation design, the REZ9 architecture was advanced such that the ALU support multiple functional units, such that a higher degree of parallelism can be achieved during processing. For this and other reasons, the REZ9 ISA was designed to support a dual accumulator architecture. The accumulators, named A and B, each have their own set of instructions. When development of the ISA began, the focus was to capitalize on the strengths of the existing accumulator based "data paths" of the REZ9 architecture and

to provide additional parallelism through innovative instruction design. This was only part of the design objectives, as the residue-based operations also impose special requirements, which differ greatly from binary processors.

Since two of the three threads are accumulator based, the majority of the instructions will store their final value in ALU A or ALU B. Complex instructions were designed to take advantage of the dual accumulator design. These instructions behave as multithreaded instructions but a single instruction is executed instead of executing two instructions in parallel. For example, the instruction sqr_AB takes advantage of the dual accumulator setup to square both values in both accumulators simultaneously. Assuming both values are already in the prospective accumulators, the result of both squares can be returned in a single clock cycle without any conflict from each other. Instructions like this allow the programmer to create complex programming sequences by taking advantage of this form of parallelism. One example would be the Mandelbrot calculation with requires the x and y coordinates to be continually squared.

4.2 INSTRUCTION TYPES

4.2.1 DATA PROCESSING AND ARITHMETIC INSTRUCTIONS

These instructions produce the bulk of the instruction set architecture. As previously mentioned, these instructions are all accumulator based, with the accumulator being designated as the destination of the result.

NIOS CONTROLLED INSTRUCTIONS

Several instructions are not supported by the REZ9. Control and branch instructions are controlled by the NIOS instruction set instead of the REZ9. This design choice was elected to allow the focus of the REZ9 to be on arithmetic instructions.

Allowing the NIOS to handle non-arithmetic instructions keeps the focus of the overall design on the residue operations.

ARITHMETIC INSTRUCTIONS

These instructions support both register direct and register indirect modes, however, only register indirect mode is currently implemented. The first operand is always the accumulator with the second operand being the register file. These instructions do not include logical operations such as AND or NOT. Since bitwise instructions apply to binary values, the NIOS processor handles them.

COMPARISON INSTRUCTIONS

Comparison instructions compare the accumulator value with the selected register address. The result of the comparison is reflected by the status register.

MULTIPLY INSTRUCTIONS

Multiplication is performed in its own separate entity, and because of this, the instructions are placed in a separate category. These instructions will take a fractional residue value and multiply it by another fractional residue value. The resulting residue product is normalized, and is stored in the accumulator in the identical fractional format as the operands.

DUAL ACTION ARITHMETIC INSTRUCTIONS

These instructions are essentially the same as the arithmetic instructions. However, these instructions allow both ALU A and B to be accessed and executed simultaneously. Also, since the register file is dual ported, these instructions are not

prohibited from accessing different register addresses (indirectly) for the same instruction.

Mnemonic	Operation	Action
add_AB	A← A + rX, B←B + rY	add to A & B
sub_AB	A← A - rX, B←B - rY	sub from A & B
mult_AB	A← A * rX, B←B * rY	integer multiply of A & B
sqr_AB	A← A * A, B←B * B	integer square of A & B
fsqr_AB	A← A * A, B←B * B	fractional square of A & B
cmpu_AB	ASR← (A > rX), (A > rY)	unsigned compare A & B
cmps_AB	ASR← (A > rX), (A > rY)	signed compare A & B
neg_AB	A ← neg(A), B ← neg(B)	negate A & B
clr_AB	A ← 0, B ← 0	clear A & B

Table 8: Parallel Data Processing and Arithmetic Instructions [2]

4.2.2 LOAD, STORE, AND MOVE INSTRUCTIONS

These instructions allow residue data to be transferred within the RZ9 ALU. Information from the register files can be loaded into the accumulator before arithmetic operation occurs. In addition, data can be stored from the accumulators into specified registers for later use. The move instructions are designed for register-to-register transfer in situations where accumulator access is not necessary for the transfer of data. There exists are set of dual instruction as well which can improve program speed by performing multiple operations simultaneously.

Mnemonic	Operation	Action
Load_A	A ← Rs	Loads the accumulator A from the register source
Store_A	A ← Rd	Stores the accumulator A to the register destination
Load_B	B ← Rs	Loads the accumulator B from the register source
Store_B	B ← Rd	Stores the accumulator B to the register destination
Load_AB	A ← Rs, B ← Rs	Loads the accumulator A & B from the register sources
Store_AB	A ← Rd, B ← Rd	Stores the accumulator A & B to the register sources
Move	Rd ← Rs	Move value at Rs to Rd
Swap_AB	T ← A, A ← B, B ← T	Swap accumulator A & B
Write_S	S ← Rs	Writes the special register S with register value Rs
Read_S	Rn ← S	Reads special register S to a NIOS register Rn

Table 9: Load, Store, and Move Instructions [2]

4.2.3 CONVERSION INSTRUCTIONS

While residue is useful for performing computations quickly, it has one serious drawback. Residue numbers have no real value in the real world without the context of moduli. For example, given two decimal numbers it can instantly be determined which number is greater since it is a weighted system. The same can be said for two binary numbers, regardless if they are converted back to decimal format or not. Given two residue numbers, it cannot be determined purely by inspection, which is larger since their values are not organized by a weighted system of digits.

Therefore, representation of computed arithmetic values in RNS must be converted to binary for use by the host binary processor. Likewise, input data to be processed by the REZ9 ALU must be converted from binary to residue before processing

begins. While software programs written using the REZ9 ALU can perform conversion of residue results to binary, this process is often too slow. Therefore, to remedy this situation, the REZ9 supports full hardware conversion of binary data to RNS and RNS results back to binary. Special hardware conversion instructions are defined by the ISA to invoke various data conversions at the speed of code execution. This solution greatly solves the problem of lengthy conversion times confronted in the prior art. This allows the REZ9 processor to process the same information a standard processor would receive.

Therefore, before any piece of data can be utilized by the REZ9 ALU, it first must be converted into residue format. While most arithmetic operations can be achieved in a single clock cycle, the number of clock cycles to perform a conversion will be approximately equal to the number of digits of the value (i.e. a 10-digit residue number requires 10 clock cycles for conversion.)

4.2.4 SIGNED INSTRUCTIONS

Examination of the Standard Data Processing and Arithmetic Instructions table and the Data Conversion Instructions table located in appendices O and P, it is apparent that while the REZ9 accepts signed and unsigned values, the majority of the instructions are not sign specific. The reason for this is that the unique architecture of the REZ9 allows some instructions to generate the proper sign for the result once computed. The REZ9 determines the sign of a value by checking the magnitudes of the number or by checking whether it has a valid sign bit or not.

The sign and valid sign bits support residue values since they use the "method of complements." When signs are not valid, measurements need to be made to find the true residue value. Valid sign bits signify that one can trust the current sign. While

instructions may generate a sign, some instructions invalidate this sign bit. Other instructions ignore or correct the sign, making the sign bit valid again. The instructions that validate or invalidate a sign are the advanced instructions such as fractional multiply and conversion. Most arithmetic and movement instructions (load and store) do not affect the sign bit during computation

For simple instructions like addition and subtraction, sign flags are compared before computation. If addition is being executed and the signs are valid, and they are the same, then the REZ9 only has to preserve the incoming sign and make sure it matches the output. If signs are different, then REZ9 must modify the negative value to make sure the proper residue value is found. To convert a positive residue number into its negative equivalent the value is subtracted from its modulus, or the value zero (the modulus value is zero!). The process does create more combinational circuitry in the REZ, but it also provides the benefit that saving instructions space. Instead of having instructions for both signed and unsigned values, the REZ just has instructions that operate regardless of sign. The more complex instructions do not require extra circuitry simply because they are designed to compare values during execution.

The best example is the multiplication of fractional values. The architecture of the REZ9 is designed to compute the intermediate value and its complement simultaneously. During the fractional multiply process, both the original intermediate value and its complement is compared against the negative value range. Only one of the intermediate values can be positive, the other negative. Therefore, the fractional multiply algorithm discards the negative intermediate value, since its value cannot be normalized. The positive intermediate value is chosen, and is then normalized; its value represents the

absolute value of the correct answer. If the original intermediate value is negative, then the chosen normalized value is complemented, since the final answer was intended to be negative (based on method of complements), otherwise, the result is positive. The multiply operation also sets the sign valid flag true, and sets the appropriate value of the sign flag of the result.

4.3 INSTRUCTION ENCODING

Instructions for the REZ9 coprocessor are implemented using the Avalon interface connected to the NIOS II CPU. Details on both the NIOS II CPU and Altera's custom instructions are presented in chapter 5.

31 27	26 22	21 17	16	15	14	13 6	5 0
rX	rY	rZ	readrx	readry	writez	REZ9 OPCODE	0x32

Table 10: Altera Custom Instruction Format [2]

The first three fields of the custom instruction format indexes up to three operands. By default, if an instruction requires access to only one register it will always default to the first operand rX. For instructions that access two registers it will use both operands rX and rY. Recall that the REZ9 is an accumulator based ALU so most instructions use only the first operand as a location for the data operand. Instructions that use two simultaneous registers are primarily the dual instructions. For example, the supported instruction LOAD_AB can access two separate registers, so both rX and rY are needed to allow the instruction to operate immediately. Table 9 illustrates the variations of the load instruction.

4.5 PARALLELISM

Implementation of the multithreaded execution required the modification of the base execution scheme in the NIOS processor. Specifics of this process are covered in Chapter 5.The main change is to modify when an instruction tells the processor it is finished. Normally, the next instruction will only start after it receives confirmation that the current instruction has finished executing. This confirmation comes in the form of a "done" signal generated by the REZ9 IEU and transmitted to the NIOS IEU. To provide instruction parallelism, the done signal is modified so it is sent before an instruction has finished execution. By prematurely sending this signal, the NIOS can begin fetching and decoding the next instruction. This act of pre-fetching is designed to reduce the start cycles (fetch and decode) and have the instruction prepared to execute immediately after the current instruction finishes its execution.

With the above pre-fetching mechanism is in place, both instructions will run in parallel. Thanks to the dual accumulator design of REZ9, the processor also supports dual instruction execution, since ALU A and B can operate independently regardless of the threaded execution scheme. The REZ9 dual instructions are single instruct ructions that perform two operations simultaneously. The instructions themselves are designed to trigger the control logic of both ALU A and B at the same time. multithreaded form of parallelism will have individual instructions (LOAD_A, STORE_B) operate in parallel without the creation of a dedicated instruction (LOAD_AB). The primary benefit of this proposed setup is that time costly instructions such as conversion can be executed in the background, while simple instructions are still being performed in the foreground. This avenue also opens up the possibility of modifying the REZ9 ALU to accommodate

additional accumulators so more accumulator based instructions can be executed. This level of parallelism would still be able to tie in with the third thread: register-to-register instructions. Even without adding accumulators, the register-to-register instructions will be able to execute alongside the accumulator instructions creating another thread of parallelism.

One of the required features to aid in optimization of the parallelism is the creation of a resource register. This register will be designed for the sole purpose of keeping track of accumulator operations so no overlap can occur when executing multiple instructions in parallel. For example, if given two instructions such as a fractional multiply and an addition instruction that both pertain to accumulator A, this resource register will make sure that both instructions do not overlap simultaneously such they are both accessing the same resource (i.e. the A accumulator.)

CHAPTER 5: INSTRUCTION EXECUTION UNIT

5.1 OVERVIEW

The purpose of the Instruction Execution Unit (IEU) was to create a hardware implementation of the software coprocessor system. The design methodology behind the IEU was precision and functionality above all other factors. The creation of the IEU focused primarily on creating a control path system that could connect to the already existing datapath of the REZ9 processor.

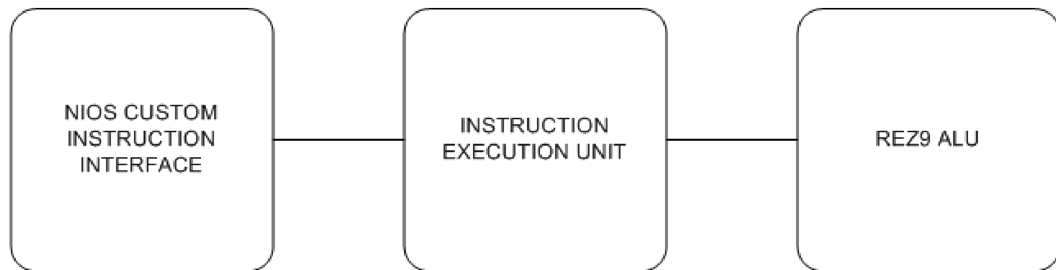


Figure 3: Instruction Execution Unit Model

5.2 SOFTWARE IEU

The original hardware implementation of the REZ9 IEU supports a software based control system. In this implementation, port pin IO drives the many control lines within the REZ9 ALU. Control algorithms drive the port pins, which then drive the REZ9 ALU hardware. These algorithms are implemented in the Eclipse IDE.

This software controller was created using Altera Quartus II software. The existing software controller was developed using an Eclipse IDE. This IDE aptly named NIOS II Software Build Tools for Eclipse, allowed manipulation of the NIOS II

processor on the target FPGA. The creation of this software environment was for verification and debugging of control logic that will be implemented in hardware.

Initially the IEU was created using Altera's Qsys software. This first iteration of the IEU was absent of any hardware control signals. It functioned by being entirely controlled by software stimulus. When this "software controller" was compiled into the FPGA, the NIOS processor behaved on a purely software driven level. Every instruction that existed was implemented using C code in the Eclipse IDE

The code in Appendix B and C contain the instructions implemented in software. The C code explains the behavior of each signal required to execute the instructions in the software controller. When the hardware controller was implemented, its function was derived from these functions.

5.3 HARDWARE IEU IMPLEMENTATION

The creation of the IEU was segmented into two phases: a block diagram that connected to the existing REZ9 processor and Verilog code which articulated the behavior of the control system. The Verilog code located in Appendix A contains instantiations of the current instructions that can be implemented within the REZ9 ALU. Each module details the hardware behavior for each instruction. The difference compared to the software controller is that this code is hardwired into the FPGA during compilation instead of being declared in the Eclipse IDE.

The hardware controller was built into the existing software controller. This design created control system that supported both a software and hardware controller.

5.4 QSYS SYSTEM INTEGRATION TOOL

Qsys is a system integration tool developed and supported by Altera that allows connections between intellectual property components easier. [8] Qsys was used to create the NIOS II processor of the REZ9. Through Qsys, a custom processor component was developed based on the written Verilog code. In the Verilog code, each instruction is instantiated in a module. These modules are all designated by an 8-bit numerical value making room for up to 256 possible instructions. When the Qsys generates the NIOS processor, these values are still used for reference in the Eclipse IDE. In Appendix C, the header file list the hardware instructions generated in the NIOS. The values used to define those instructions are the same values instantiated in the Verilog code. For example, STORE _A instruction is defined with a value of 4 in Verilog. In the header file alu_instruct.h the custom instruction STORE_A is also defined with a value of 4 (ALT_CI_NIOS_ALU_INST1_0(4,n,0)).

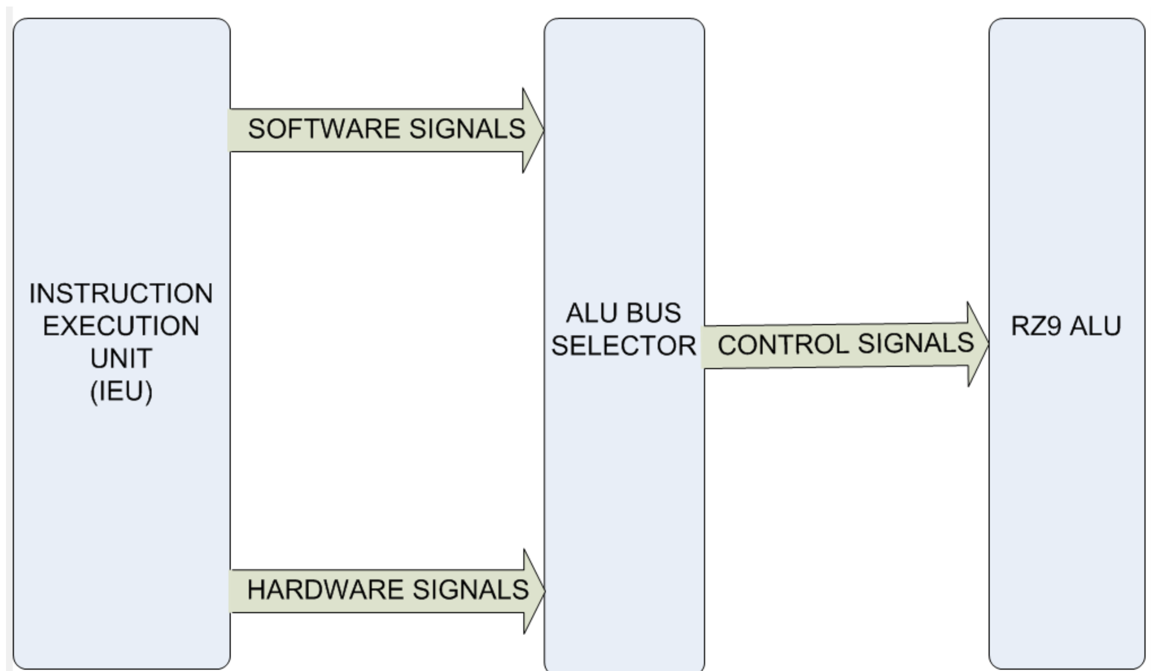


Figure 4: IEU and REZ9 ALU Interface

To facilitate support of both the hardware and the software control methods, an internal bus selector was created. . From Figure 4, the port connections for the bus selector are visible. The connections are labeled as either "port" or as "hard". Based on which value the selector chooses, these controls are connected to the ALU as seen in Figure 4. The "hard" connections refer to the hardware operations from the NIOS. When these "hard" connections are used, the control signals sent to the ALU correspond to hardware actions on the FPGA. When the "port" connections are selected, the ALU still uses the same internal controls; however, during "port" control, certain signals can be tapped and exported during operations allowing for testing and debugging. The cost of using these port signals instead of the "hard" signals is the overall speed of the instructions is greatly reduced. The purpose of this design, allowed for simultaneous debugging and development of new and existing instructions for the REZ9.

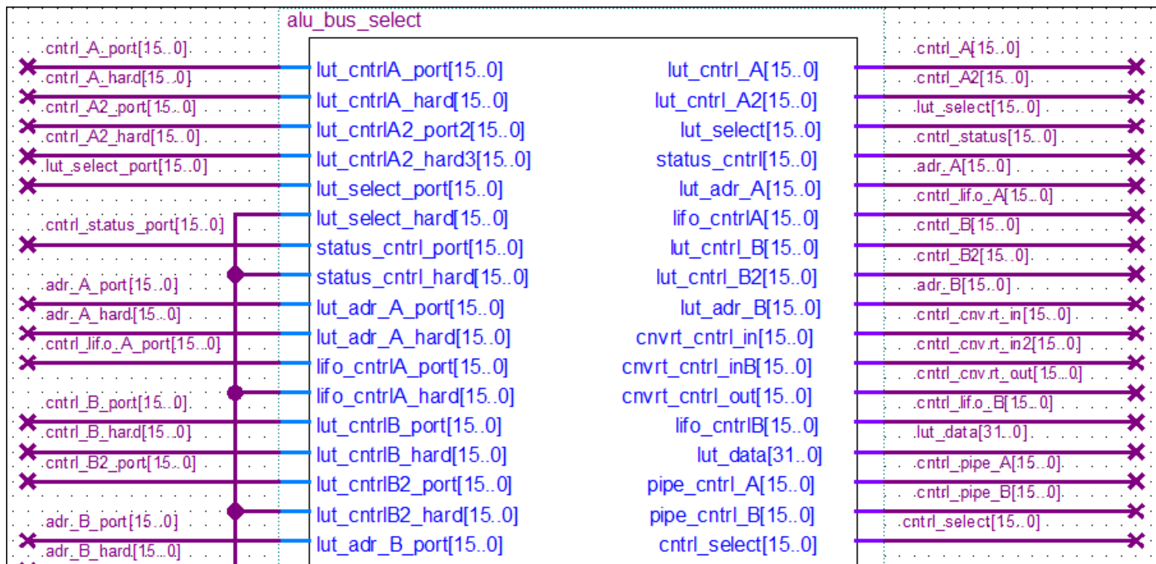


Figure 5: ALU Bus Selector for REZ9

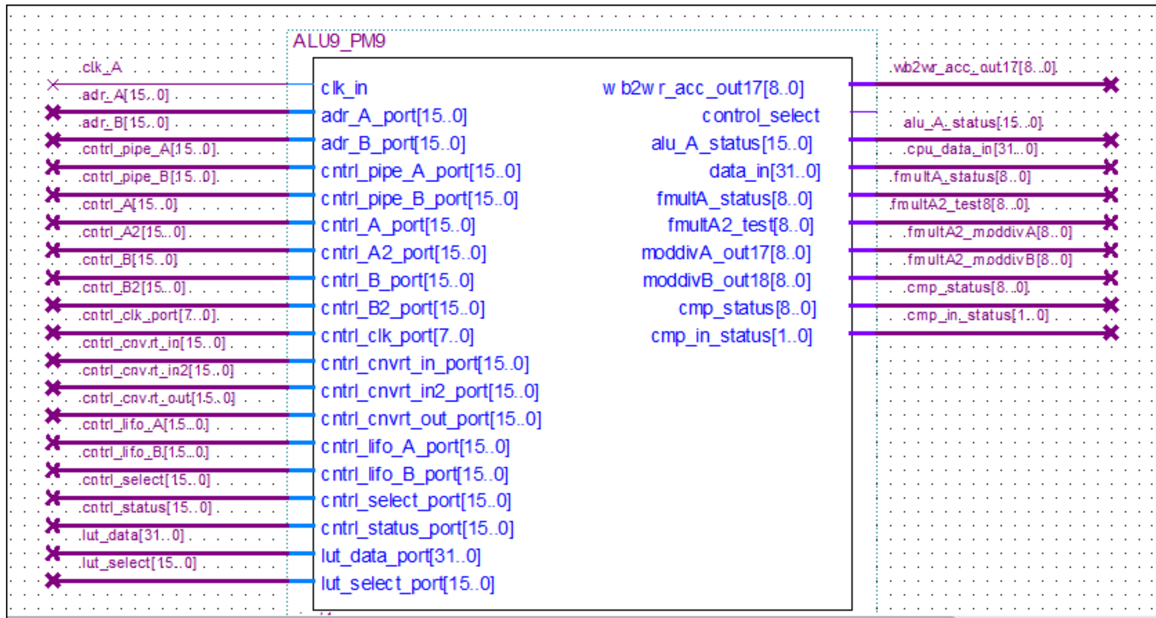


Figure 6: ALU Port Bus Connections

5.5 NIOS CUSTOM INSTRUCTION BACKGROUND

By using the NIOS Custom Instructions (NCI), we bypass most challenges with designing an IEU for the REZ9. The NCI requires an opcode and an address location. After the NCI receives its data from the Eclipse IDE C code commands, the NCI will send the address information and opcode information to the control unit. The IEU will interpret the data and based on the opcode, it will select the proper instruction and send out the necessary control signals to the data section of the processor. When our instruction finishes executing, a "done" signal is sent to the NCI waits for the next instruction.

The behavior of the NCI is dictated by its type. There are five types of custom Instructions each building upon the previous one. At the first level, there are the *Combinational Instructions*. These custom instructions assume all logic being performed can be completed in a single clock cycle. The only required internal port connection is

the result bus that sends data back to the NIOS. The second type is *Multicycle Instructions*. As the name suggest these instructions take into account that whatever logic being performed may take more than a single clock cycle to operate. Because of this feature, the NCI will require a port for a system clock, clock enable, and reset ports to connect to the conduit. The third level contains the extended instructions, which allow multiple instructions to be executed in a single logic block. For example, a simple swap routine requires the use of load and store instructions performed in a specific order. The order of these instructions is index, so a single call can trigger up to 256 different operations (the max index is eight bits wide.) The *Internal Register File Instructions* allow the user access to either the NIOS II processor's register file or the internal register files of the custom instructions. More importantly, it allows instructions to write back results to the internal register files instead or the NIOS II processor's register file. The fifth and final type *is External Interface Custom Instructions*. These instructions encompass all of the features of the previous types with the added feature of connecting to an external interface. The IEU of this thesis was created using the External Interface Custom Instruction so it could attach the existing REZ9-A ALU. This can be seen by reviewing the instruction-encoding format given in chapter 4. If the REZ9 only supported combinational instruction types then there would be no need for the creation of the readrx, readry, and writez bits since these correspond to the ports used the more advanced instructions. [6]

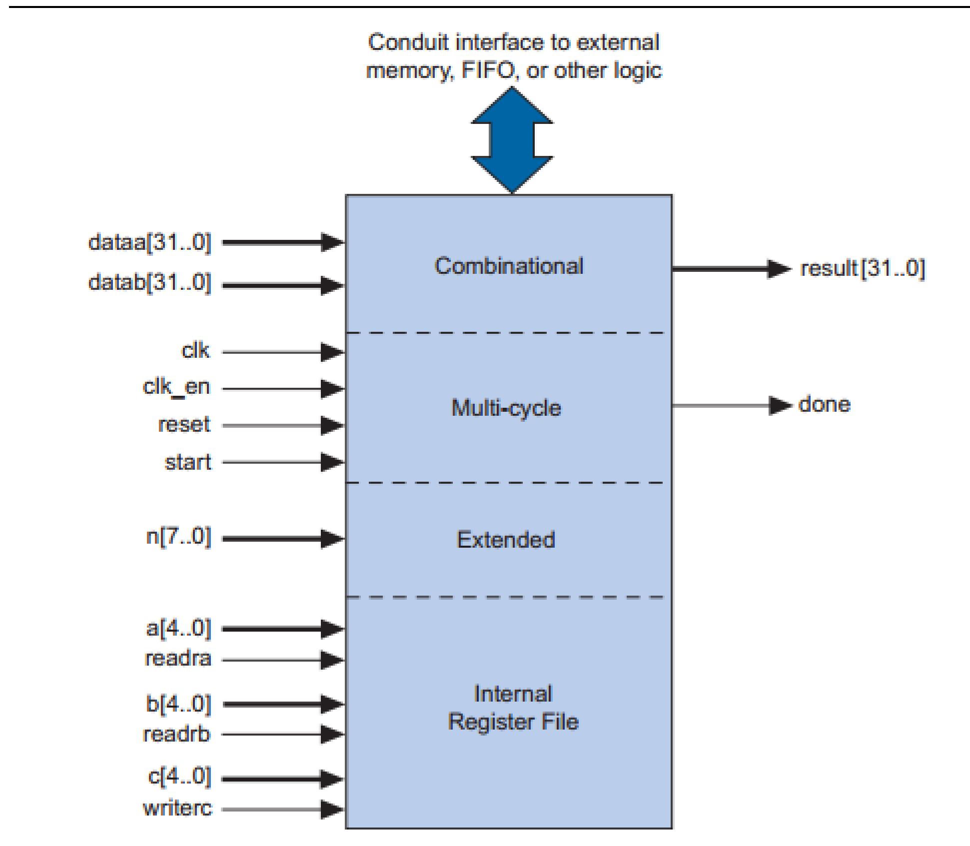


Figure 7: Hardware Block Diagram of a NIOS II Custom Instruction[6]

The NIOS Custom Instruction simplifies the process of creating assembly level code for the instructions. Many of the movement commands such as branch and jump are handled by the NIOS. All of the arithmetic operands such as add, sub, and multiply are implemented instruction execution unit (IEU control unit) implemented in Verilog. This method allows the use of inline assembly so non-arithmetic instructions can be executed by the NIOS processor through the Eclipse IDE.

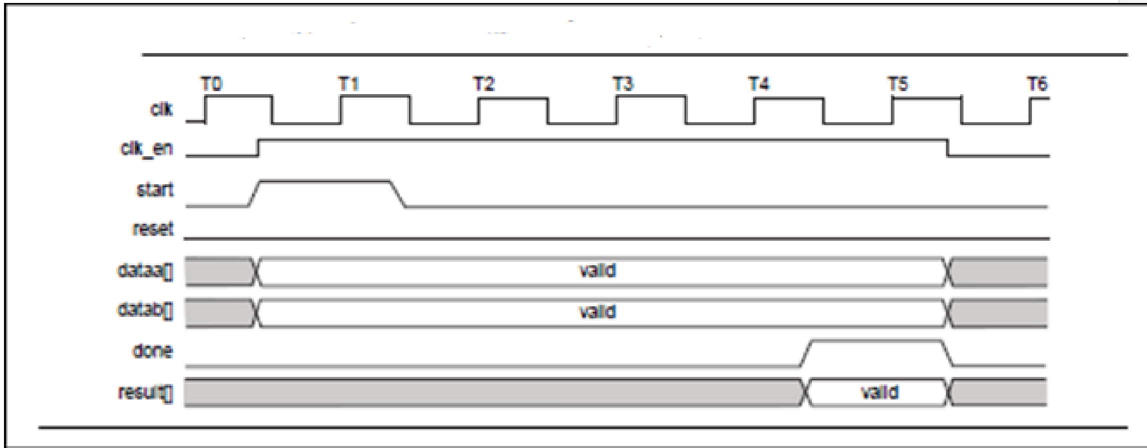


Figure 8: Multicycle Custom Instruction Timing Diagram [6]

When the NCI is initialized it will wait to receive and instruction (i.e. for the C to reach an instruction command.) Once a valid command is received, the NCI will send out a "start" signal to the REZ9 IEU, which informs it that a valid instruction was received. The IEU will take the given instruction code and locate the created instruction module. The Verilog code for the selected module details the sequence of events that need to occur in hardware to perform the necessary instructions. When the module is complete, it returns a "done" signal to the NCI so that it knows it has finished. When this done signal is received the NCI stops the "CPU clock" (clock controlling all REZ9 hardware) and looks for the next instruction to be executed.

CHAPTER 6: SIMULATION & TESTING RESULTS

6.1 VERILOG SIMULATION RESULTS

The testing and verification of the ISA and IEU were performed on two fronts. The Verilog code was tested within the Quartus II waveform simulator. The simulator was used to verify not only if the proper signals were propagating through the ALU, but they were also used estimate and verify timing assumptions. One of the initial claims of this thesis is verification that the REZ9 can support operations that can be performed in single clock-cycles as opposed to their binary equivalent. From the waveform simulations, it is apparent that basic arithmetic operations all performed within a single clock cycle. Longer operations such as the fractional multiply, reveal that clock cycle time is not static; however, based on the simulation results it can be hypothesized that the speed of a fractional multiplication operation will require only as many clock cycles as there are residue digits being used. This suggest that execution time of fractional operations of the REZ9 versus that of a comparable binary ALU will most likely show that for small operations the REZ9 is not the optimal solution. However, dealing with extremely large values with large digit sizes will be in favor of the REZ9 over a comparable binary ALU since the execution time of the binary ALU does not increase linearly, while the REZ9 cycle time does.

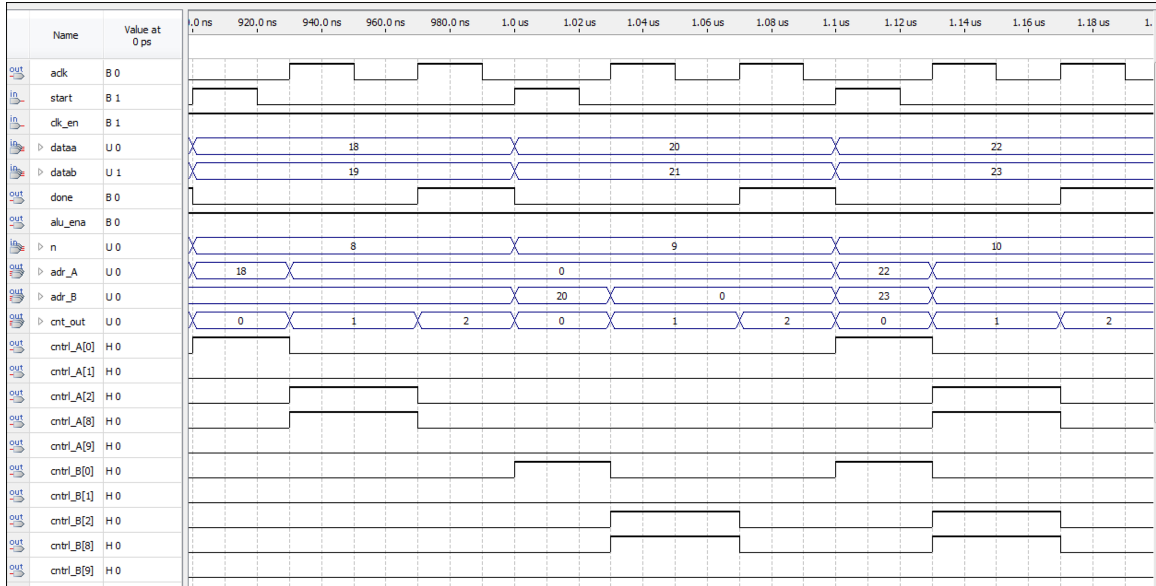


Figure 9: Waveform Simulation-Addition

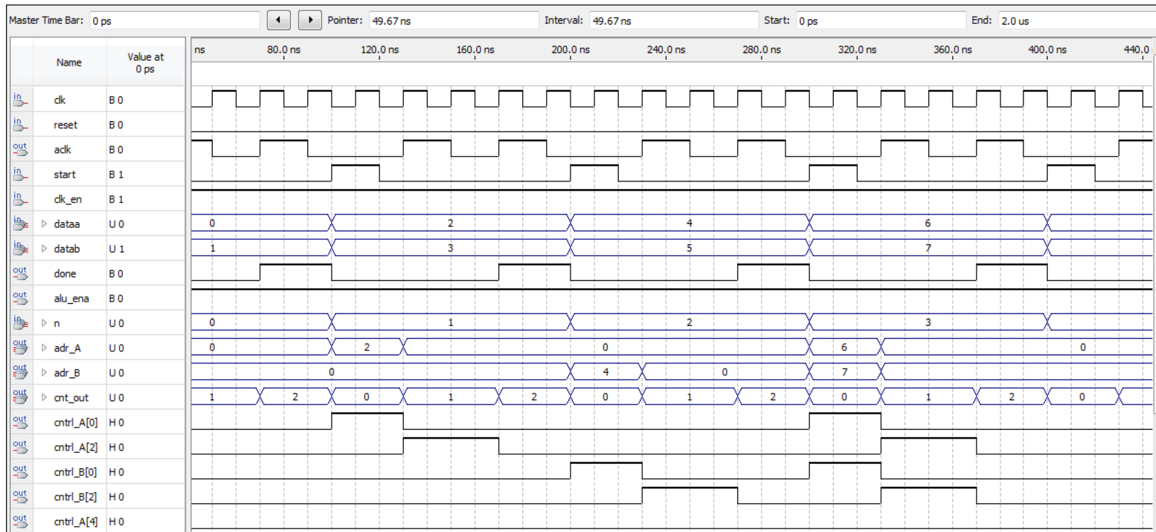


Figure 10: Waveform Simulation-Load

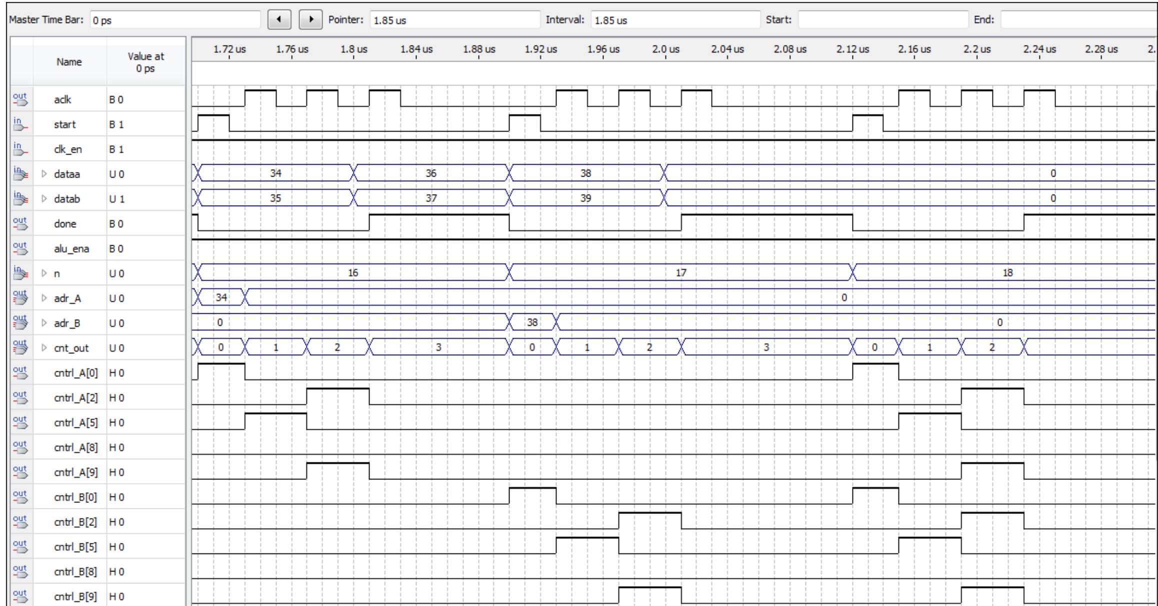


Figure 11: Waveform Simulation-Integer Multiply

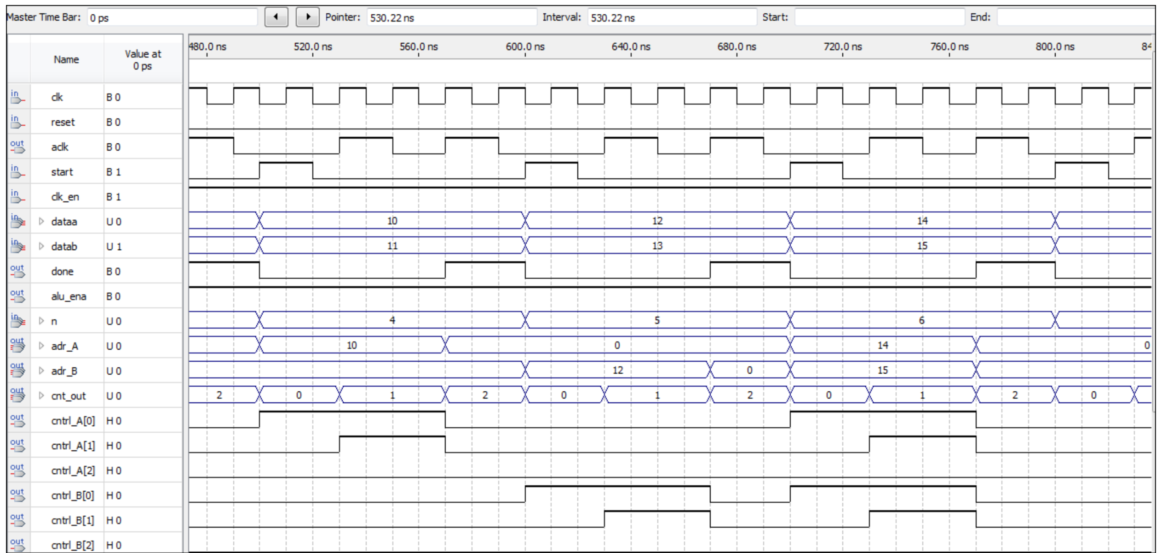


Figure 12: Waveform Simulation-Store

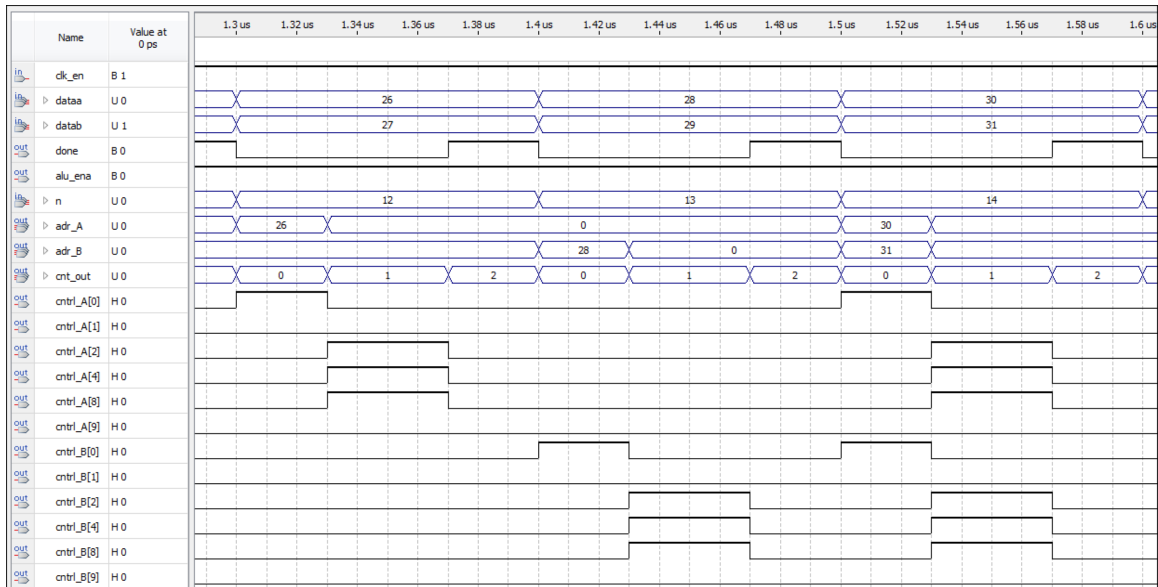


Figure 13: Waveform Simulation-Subtraction

6.2 ECLIPSE IDE TEST ROUTINES & RESULTS

The second testing platform of REZ9 instructions is performed directly within the Eclipse IDE. In this environment, the IEU and ISA were tested for accuracy and functionality within the FPGA. Of primary importance was verification that control signals produce the correct results. The Eclipse testing environment was also instrumental in finding the optimal testing speed to run the NIOS II processor.

Since initial testing was performed on the more powerful EP4SGX530KH40, optimum speed held around 25MHz for the NIOS clock speed. This was done before circuitry optimization and without the removal of extra hardware used to support the software controller. It would be reasonable to assume that the REZ9 will be able to reach higher speeds with proper optimization techniques. The design was eventually moved to a lower end FPGA to verify is the system would remain functional using a less powerful platform. Results indicated that the moving the design to the EP4CE115F29C7 did not adversely affect the performance of the REZ9 ALU.

Appendixes D through M contain the results of the various test routines that were implemented in the Eclipse IDE. These tests were designed to verify the proper functions of the instruction execution unit. For each test, there is software and a hardware routine running. The software routines are used as controls to verify that the hardware is behaving properly. If the results of the hardware match the software results then the routine displays the word PASS.

6.2.1 ARITHMETIC ALU TESTS

The results printed during the test routines typically display the accumulator A or accumulator B of the REZ9. Printed values are shown for each individual accumulator digit, where each digit is displayed in either a decimal or a hexadecimal representation.

For example, the number 7 can fit in everyone one of the moduli being used in the REZ9 ALU, so when the result of an operation (4+3 for example) occurs, the output will be printed as such:

(V)+ 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>

If the value were higher than one of the moduli then the result would have a “roll over effect” as explained in the residue chapter. For example, if the first modulus were six then when the result of the operation produces a seven the output would appear as such:

(V)+ 1 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>

Since 7-6 = 1, the first residue digit rolls back around to one. However, since the other moduli in this example all have values larger than six they do not reset back to 1Smaller values are used for simple testing to prove the functions of the basic arithmetic, load, and store functions. For the testing of the REZ9 processor, the following 18 moduli were used testing. Recall for from chapter 2 these digits are all pairwise prime.

{121, 125, 169, 243, 356, 289, 343, 361, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509};

The "+" symbol signifies that the sign of the current number in the accumulator is positive. Also the numbers on the end represent the number of skipped digits and the number of zeros located in the accumulator. Recall back to chapter 3 that each register has a dedicated area for the sign bits and skip digits. These values are present for testing and verification that the accumulator is processing the correct data. Appendixes D and E contain both the C code and there results of the arithmetic ALU tests.

6.2.2 FRACTIONAL MULTIPLY TESTS

The fractional multiply test results are presented in a different format than the arithmetic ALU tests. The more complex tests such as the fractional multiplication have their answers displayed as a single output instead of listing the value of each residue digit. This provides an easier viewing of complex test. The results displayed during the fractional multiplication displayed in decimal form. This format was selected so that the level of precision obtained by the fractional multiply could be displayed.

6.2.3 COMPARISON TESTS

The comparison tests were generated to as a test bench for higher-level instructions. At its core, the comparison test already exists in the fractional multiply instruction. The main difference and reason for testing a standalone comparison function was to verify the REZ9 could properly return requested data based on an outcome. The arithmetic tests all deposit their answer within an accumulator and the routine read the value place in the accumulator. The reasons that led to the creation of the comparison routine test was to first verify that a comparison instruction would work properly as a standalone instruction instead of being imbedded into another. The second reason was to

verify that data could be actively sent to the NIOS from the REZ9 instead of it being statically read.

The test routines essentially run a software comparison of the two numbers and when finished display if the number is greater than, less than, or equal to the number that is being compared. The hardware runs the same routine however; it sends a value to the NIOS. The NIOS then compares the value from the hardware routine to the computed value of the software routine. If the values match then it is considered a PASS verifying that the hardware is working properly. The printed results on the screen are for the software test; the only indication of the hardware routine working is the PASS or FAIL message. This is by design since the hardware cannot be "paused" as the software controller can. Creating breakpoints and print routines allows the software to print results and wait for a user response before continuing. This is intentional for debugging and testing purposes, and would not be accessible in a final production version. The results and code for the comparison test can be found in Appendixes J and K.

6.3 MANDELBROT TEST ROUTINE AND RESULTS

A simple Mandelbrot test routine was generated to test the majority of the available instruction modules to verify their effectiveness. The test routine was not designed with zoom functionality. Instead, each test allows the user to change the number of iterations desired for the routine. As the number of iterations increase, the displayed image changes as the edges become more detailed since. This test effectively proves that the processor has the capability to perform the routine regardless of iteration size and still be able to compute the value of each location with accuracy. Appendix L

and M contain the code for the Mandelbrot routine along with the results of each test with the specified number of iterations.

CHAPTER 7: CONCLUSION

The project is still in the testing and verification stages. The initial goal behind the development of the REZ9 coprocessor was to perform real comparisons against comparable hardware platforms. While initial testing is promising, there are still goals that need to be accomplished. In its current state, the REZ9 has laid the groundwork for the majority of all instructions but not all are fully implemented. The biggest focus is to fully test the implementation of fractional and integer division instructions. Completion of the integer instructions will complete the primary arithmetic operations allowing the REZ9 to achieve its main goal of being considered a residue-based general processor.

Future work still includes optimizing the overall REZ9 so that comparable tests are accurate. The current state of the REZ9 retains the majority of its software controller environment, which makes it useful for testing and verification. In the final iteration this component will be removed which will drastically reduce hardware size. Implementation on the parallelism will also grow as the current design is successful on the software simulation level but it has not been fully tested and verified within the hardware controller. Register instructions are also still in the testing phase. While they are supported within the ISA, the majority of the testing and operations have been, utilizing the accumulator based design and a future goal is to implement register-to-register operations that can work simultaneously with accumulator-based instructions.

APPENDIX A: VERILOG CODE

```
/*
  THIS FILE CONTAINS COMPLETED WORKING MODULES:  LOAD, STORE,
ADD, SUB, MULT,(all previous include A, B, AB),  FMULT(A only) CNVRTFS,
MAGNITUDE COMPARE,
*/


module Nios_alu_inst1           (clk,
                                reset,
                                 dataa,
                                datab,
                                 n,
                                clk_en,
                                start,
                                a,
                                b,
                                c,
                                readra,
                                readrb,
                                readrc,
                                done,
                                result,
                                aclk,
                                cnt_out,
                                done_out,
                                adr_A,
                                cntrl_A,
                                adr_B,
                                cntrl_B,
                                alu_ena,
                                cntrl_A2,
                                alu_A_status,
                                fmultA_status,
                                cntrl_cnvrt_in,
                                cntrl_cnvrt_out,
                                cntrl_select,
                                cntrl_pipe_A,
                                cmp_out_status);


  input clk;
  input reset;
  input [31:0] dataa;
  input [31:0] datab;
```

```
  input [7:0] n;
  input clk_en;
  input start;
  input [4:0] a;
  input [4:0] b;
  input [4:0] c;
  input readra;
  input readrb;
  input readrc;
  input [15:0] alu_A_status;
  input [8:0] fmultA_status;
  input [8:0] cmp_out_status;

  output done;
  output [31:0] result;
  output [4:0] cnt_out;                 // for analyzer tracking
  output done_out;                      // for scope testing
  output aclk;                  // Clock output to ALU
  output [15:0] adr_A;                  // Address_A Bus output to ALU
  output [15:0] cntrl_A;                // Control_A Bus output to ALU
  output [15:0] adr_B;                  // Address_B Bus output to ALU
  output [15:0] cntrl_B;                // Address_B Bus output to ALU
  output alu_ena;

  output [15:0] cntrl_A2;               //control A2 bus to output to ALU (fmult)
  output [15:0] cntrl_pipe_A;
  output [15:0] cntrl_cnvrt_in;
  output [15:0] cntrl_cnvrt_out;
  output [15:0] cntrl_select;
  //output[31:0] lut_data;

//
***********************************************************************
***************
  reg [4:0] cnt;
  reg aclk_ff;                  // clock and done signal for the ALU
  wire inst1_done;
  wire inst2_done;
  wire inst3_done;
  wire inst4_done;
  wire inst5_done;
  wire inst6_done;
  wire inst7_done;
  wire inst8_done;
  wire inst9_done;
  wire inst10_done;
```

```
  wire inst11_done;
  wire inst12_done;
  wire inst13_done;
  wire inst14_done;
  wire inst15_done;
  wire inst16_done;
  wire inst17_done;
  wire inst18_done;
  wire inst19_done;
  wire inst20_done;
  wire inst21_done;
  wire inst22_done;
  wire inst23_done;

  assign cnt_out[4:0] = cnt; //{1'b0, cnt[4:1]};           // may need to increase counter width
  assign aclk = aclk_ff;
  assign done =  inst1_done | inst2_done | inst3_done | inst4_done | inst5_done |
inst6_done | inst7_done | inst8_done | inst9_done |  inst10_done | inst11_done |
inst12_done | inst13_done | inst14_done | inst15_done | inst16_done | inst17_done |
inst18_done | inst19_done | inst20_done | inst21_done | inst22_done | inst23_done;

// don't forget to insert the instruction done signal here
  assign done_out = done;

  wire [31:0] result1;
  wire [31:0] result2;

  assign result[31:0] = result1 | result2;

 //
*************************************************************************
**************

  wire RegA_clk_ena;                              // Control_A wires
  wire RegA_wr_ena;
  wire accA_clk_ena;
  wire accA_clear;
  wire add_sub_A;
  wire multA_clk_ena;
  wire divA_clk_ena;
  wire clr_skipA_ena;
  wire [1:0] accA_sel;
  wire modcntA_rst;
  wire modcntA_load;
  wire modcntA_clk_ena;
  wire start_lat_A;
```

```
  assign cntrl_A[0] = RegA_clk_ena;            // internal ALU control wire assignments to
the exported cntrl_A bus
  assign cntrl_A[1] = RegA_wr_ena;
  assign cntrl_A[2] = accA_clk_ena;
  assign cntrl_A[3] = accA_clear;
  assign cntrl_A[4] = add_sub_A;
  assign cntrl_A[5] = multA_clk_ena;
  assign cntrl_A[6] = divA_clk_ena;
  assign cntrl_A[7] = clr_skipA_ena;
  assign cntrl_A[9:8] = accA_sel[1:0];
  assign cntrl_A[11:10] = 2'b0;                    // reserved for larger accA select
  assign cntrl_A[12] = modcntA_rst;
  assign cntrl_A[13] = modcntA_load;
  assign cntrl_A[14] = modcntA_clk_ena;
  assign cntrl_A[15] = start_lat_A;

//  assign RegA_clk_ena = 0;                   // tie unused ALU control wire assignments
to 0
//  assign RegA_wr_ena = 0;
//  assign accA_clk_ena = 0;
 // assign accA_clear = 0;
 // assign add_sub_A = 0;
  //assign multA_clk_ena = 0;
 // assign divA_clk_ena = 0;
  //assign clr_skipA_ena = 0;
//  assign accA_sel[1:0] = 0;
 // assign modcntA_rst = 0;
  assign modcntA_load = 0;
//  assign modcntA_clk_ena = 0;
  //assign start_lat_A = 0;

   //
***********************************************************************
**************

  wire RegB_clk_ena;                          // Control_B wires
  wire RegB_wr_ena;
  wire accB_clk_ena;
  wire accB_clear;
  wire add_sub_B;
  wire multB_clk_ena;
  wire divB_clk_ena;
  wire clr_skipB_ena;
  wire [1:0] accB_sel;
  wire modcntB_rst;
```

```
  wire modcntB_load;
  wire modcntB_clk_ena;
  wire start_lat_B;

  assign cntrl_B[0] = RegB_clk_ena;             // internal ALU control wire assignments to
the exported cntrl_B bus
  assign cntrl_B[1] = RegB_wr_ena;
  assign cntrl_B[2] = accB_clk_ena;
  assign cntrl_B[3] = accB_clear;
  assign cntrl_B[4] = add_sub_B;
  assign cntrl_B[5] = multB_clk_ena;
  assign cntrl_B[6] = divB_clk_ena;
  assign cntrl_B[7] = clr_skipB_ena;
  assign cntrl_B[9:8] = accB_sel[1:0];
  assign cntrl_B[11:10] = 2'b0;                          // reserved for larger accA select
  assign cntrl_B[12] = modcntB_rst;
  assign cntrl_B[13] = modcntB_load;
  assign cntrl_B[14] = modcntB_clk_ena;
  assign cntrl_B[15] = start_lat_B;

//  assign RegB_clk_ena = 0;                     // tie unused ALU control wire assignments
to 0
  //assign RegB_wr_ena = 0;
//  assign accB_clk_ena = 0;
  assign accB_clear = 0;
  //assign add_sub_B = 0;
 // assign multB_clk_ena = 0;
  assign divB_clk_ena = 0;
  assign clr_skipB_ena = 0;
 // assign accB_sel[1:0] = 0;
  assign modcntB_rst = 0;
  assign modcntB_load = 0;
  assign modcntB_clk_ena = 0;
  assign start_lat_B = 0;

  //*************************************CONTROL A2
WIRES***************************************************************
*********

  wire cross_shftA_init;
  wire cross_shftA_ena;
  wire decompA_subload;
  wire modmultA_ena;
  wire modmultA_acc_ena;
  wire modmultA_acc_clr;
  wire macBcompA_ena;
```

```
  wire macA_select;

  assign cntrl_A2[0] = cross_shftA_init;                  // internal ALU control wire
assignments to the exported cntrl_A bus
  assign cntrl_A2[1] = cross_shftA_ena;
  assign cntrl_A2[2] = decompA_subload;
  assign cntrl_A2[3] = modmultA_ena;
  assign cntrl_A2[4] = modmultA_acc_ena;
  assign cntrl_A2[5] = modmultA_acc_clr;
  assign cntrl_A2[6] = macBcompA_ena;
  assign cntrl_A2[7] = macA_select;

//  assign cross_shftA_init = 0;                          // internal ALU control wire
assignments to the exported cntrl_A bus
 // assign cross_shftA_ena = 0;
 // assign decompA_subload = 0;
 // assign modmultA_ena = 0;
 // assign modmultA_acc_ena = 0;
 // assign modmultA_acc_clr = 0;
 // assign macBcompA_ena = 0;
 // assign macA_select = 0;


  //**************************************CONTROL PIPE A
WIRES*****************************************************************
*********

  wire cmp_shftA_init;                         // internal ALU control wire assignments to
the exported cntrl_Pipe_AA bus
  wire cmp_shftA_ena;
  wire cmp_decompA_subload;
  wire cmp_modcntA_rst;
  wire cmp_modcntA_load;
  wire cmp_modcntA_clk_ena;
  wire cmp_moddivA_ena;
  wire cmp_clr_skipA_ena;

  assign cntrl_pipe_A[0] = cmp_shftA_init;             // internal ALU control wire
assignments to the exported cntrl_Pipe_AA bus
  assign cntrl_pipe_A[1] = cmp_shftA_ena;
  assign cntrl_pipe_A[2] = cmp_decompA_subload;
  assign cntrl_pipe_A[3] = cmp_modcntA_rst;
  assign cntrl_pipe_A[4] = cmp_modcntA_load;
  assign cntrl_pipe_A[5] = cmp_modcntA_clk_ena;
  assign cntrl_pipe_A[6] = cmp_moddivA_ena;
  assign cntrl_pipe_A[7] = cmp_clr_skipA_ena;
```

```
 // assign cmp_shftA_init = 0;                    // internal ALU control wire assignments to
the exported cntrl_Pipe_AA bus
  //assign cmp_shftA_ena = 0;
 // assign cmp_decompA_subload = 0;
 // assign cmp_modcntA_rst = 0;
  assign cmp_modcntA_load = 0;
 // assign cmp_modcntA_clk_ena = 0;
 // assign cmp_moddivA_ena = 0;
 // assign cmp_clr_skipA_ena = 0;




  //*********************************************CONVERSION
WIRES*****************************************************************
*************

  wire fbin_latch_sel;
  wire fbin_latch_ena;
  wire fbin_latch_reset;
  wire wbin_latch_sel;
  wire wbin_latch_ena;
  wire wbin_latch_reset;
  wire fb2fr_load;
  wire fb2fr_done;
  wire fb2fr_shft_ena;
  wire wb2wr_load;
  wire wb2wr_shft_ena;
  wire wb2wr_reset;
  wire fb2fr_acc_ena;
  wire wb2wr_acc_ena;
  wire fb2fr_rnd_sel;


  assign cntrl_cnvrt_in[0] = fbin_latch_sel;
  assign cntrl_cnvrt_in[1] = fbin_latch_ena;
  assign cntrl_cnvrt_in[2] = fbin_latch_reset;
  assign cntrl_cnvrt_in[3] = wbin_latch_sel;
  assign cntrl_cnvrt_in[4] = wbin_latch_ena;
  assign cntrl_cnvrt_in[5] = wbin_latch_reset;
  assign cntrl_cnvrt_in[6] = fb2fr_load;
  assign cntrl_cnvrt_in[7] = fb2fr_done;
  assign cntrl_cnvrt_in[8] = fb2fr_shft_ena;
  assign cntrl_cnvrt_in[9] = wb2wr_load;
```

```
  assign cntrl_cnvrt_in[10] = wb2wr_shft_ena;
  assign cntrl_cnvrt_in[11] = wb2wr_reset;
  assign cntrl_cnvrt_in[12] = fb2fr_acc_ena;
  assign cntrl_cnvrt_in[13] = wb2wr_acc_ena;
  assign cntrl_cnvrt_in[14] = fb2fr_rnd_sel;
  //assign cntrl_cnvrt_in[15] = 0

  //assign fbin_latch_sel = 0;
  assign fbin_latch_ena = 0;
  assign fbin_latch_reset = 0;
  //assign wbin_latch_sel = 0;
  //assign wbin_latch_ena = 0;
  assign wbin_latch_reset = 0;
 // assign fb2fr_load = 0;
 // assign fb2fr_done = 0;
  //assign fb2fr_shft_ena = 0;
 // assign wb2wr_load = 0;
 // assign wb2wr_shft_ena = 0;
 // assign wb2wr_reset = 0;
 // assign fb2fr_acc_ena = 0;
  //assign wb2wr_acc_ena = 0;
 // assign fb2fr_rnd_sel = 0;

  wire RegF_lut_wr;
  wire RegF_lut_wr_all;
  wire [1:0]regA_sel_in;
  wire [1:0]regB_sel_in;

  assign cntrl_select[0] = RegF_lut_wr;
  assign cntrl_select[1] = RegF_lut_wr_all;
  assign cntrl_select[3:2] = regA_sel_in[1:0];
  assign cntrl_select[6:5] = regB_sel_in[1:0];


  assign RegF_lut_wr = 0;                        //signal is 1 for operations (ld, add, sub,
mult)
  assign RegF_lut_wr_all = 0;
  assign regA_sel_in[1:0] = 2'b0;
  //assign regB_sel_in[1:0] = 2'b0;


 // ************************************COMBINATIONAL
LOGIC****************************************************************
****

  reg [9:0] adrA_reg;   // may not need
```

```
  assign adr_A[15:0] = RegA_clk_ena ? {6'b0, dataa[9:0]}:{16'b0};           // 1024 10bit
general purpose registers available in this form

  reg [9:0] adrB_reg;   // may not need
  assign adr_B[15:0] = ((RegB_clk_ena) & ((n!=3)&(n!=6)&(n!=10)&(n!=14)&(n!=18)))
? {6'b0, dataa[9:0]} : (((RegB_clk_ena) & ((n==3)|(n==6)|(n==10)|(n==14)|(n==18))) ?
{6'b0, datab[9:0]} : {16'b0});
          //---------------------------------------------------------------------------------------------
---
 //controls both A and B ALU
  wire regAB_clk_ena1;
  wire regAB_clk_ena2;
  wire regAB_clk_ena3;
  wire regAB_clk_ena4;
  wire regAB_clk_ena5;

  wire regAB_wr_ena1;

  wire accAB_clk_ena1;
  wire accAB_clk_ena2;
  wire accAB_clk_ena3;
  wire accAB_clk_ena4;
  wire accAB_clk_ena5;


  wire add_sub_AB1;
  wire multAB_clk_ena1;
  //---------------------------------------------------------------------------------------------
  //ALU A controls
  wire regA_clk_ena1;
  wire regA_clk_ena2;
  wire regA_clk_ena3;
  wire regA_clk_ena4;
  wire regA_clk_ena5;
  wire regA_clk_ena6;  //compare
  wire regA_clk_ena7;
  wire regA_clk_ena8;
  wire regA_clk_ena9;

  assign RegA_clk_ena = regA_clk_ena1 | regA_clk_ena2 | regA_clk_ena3 |
regA_clk_ena4 | regA_clk_ena5 | regA_clk_ena6 | regA_clk_ena7| regA_clk_ena8 |
regAB_clk_ena1 | regAB_clk_ena2 | regAB_clk_ena3 | regAB_clk_ena4 |
regAB_clk_ena5;

  wire regA_wr_ena1;
```

```
assign RegA_wr_ena = regA_wr_ena1 | regAB_wr_ena1;


wire accA_clk_ena1;
wire accA_clk_ena2;
wire accA_clk_ena3;
wire accA_clk_ena4;
wire accA_clk_ena5;
// wire accA_clk_ena42;
wire accA_clk_ena6;
wire accA_clk_ena7;

wire fmult_accA_clk_ena1;
wire fmult_accA_clk_ena2;

assign accA_clk_ena = accA_clk_ena1 | accA_clk_ena2 | accA_clk_ena3 |
accA_clk_ena4 | accA_clk_ena5 | accA_clk_ena6 | accAB_clk_ena1 | accAB_clk_ena2 |
accAB_clk_ena3 | accAB_clk_ena4 | fmult_accA_clk_ena1 | fmult_accA_clk_ena2;

wire add_sub_A1;
assign add_sub_A = add_sub_A1 | add_sub_AB1;

wire multA_clk_ena1;
wire multA_clk_ena2; //fmult
assign multA_clk_ena = multA_clk_ena1 | multA_clk_ena2 | multAB_clk_ena1;

wire accA_clear1;
wire accA_clear2;
assign accA_clear = accA_clear1 | accA_clear2;


wire divA_clk_ena1;
wire divA_clk_ena2;
assign divA_clk_ena = divA_clk_ena1 | divA_clk_ena2;

wire clr_skipA_ena1;
wire clr_skipA_ena2;
assign clr_skipA_ena = clr_skipA_ena1 | clr_skipA_ena2;

wire modcntA_rst1;
wire modcntA_rst2;
assign modcntA_rst = modcntA_rst1 | modcntA_rst2;

wire modcntA_clk_ena1;
wire modcntA_clk_ena2;
assign modcntA_clk_ena = modcntA_clk_ena1 | modcntA_clk_ena2;
```

```
wire start_lat_A1;
wire start_lat_A2;
assign start_lat_A = start_lat_A1 | start_lat_A2;

wire cross_shftA_ena1;
wire cross_shftA_ena2;
assign cross_shftA_ena = cross_shftA_ena1 | cross_shftA_ena2;

wire cross_shftA_init1;
wire cross_shftA_init2;
assign cross_shftA_init = cross_shftA_init1 | cross_shftA_init2;

wire decompA_subload1;
wire decompA_subload2;
assign decompA_subload = decompA_subload1 | decompA_subload2;

wire modmultA_ena1;
wire modmultA_ena2;
assign modmultA_ena = modmultA_ena1 | modmultA_ena2;

wire modmultA_acc_ena1;
wire modmultA_acc_ena2;
assign modmultA_acc_ena = modmultA_acc_ena1 | modmultA_acc_ena2;

wire modmultA_acc_clr1;
wire modmultA_acc_clr2;
assign modmultA_acc_clr = modmultA_acc_clr1 | modmultA_acc_clr2;

wire macBcompA_ena1;
wire macBcompA_ena2;
assign macBcompA_ena = macBcompA_ena1 | macBcompA_ena2;

wire macA_select1;
wire macA_select2;
assign macA_select = macA_select1 | macA_select2;




//----------------------------------------------------------------------------------------------------
//ALU B controls
wire regB_clk_ena1;
wire regB_clk_ena2;
wire regB_clk_ena3;
```

```
  wire regB_clk_ena4;
  wire regB_clk_ena5;
  wire regB_clk_ena6;
  wire regB_clk_ena7;

  assign RegB_clk_ena = regB_clk_ena1 | regB_clk_ena2 | regB_clk_ena3 |
regB_clk_ena4 | regB_clk_ena5 |  regB_clk_ena6 | regB_clk_ena7 | regAB_clk_ena1 |
regAB_clk_ena2 | regAB_clk_ena3 | regAB_clk_ena4 | regAB_clk_ena5;



  wire accB_clk_ena1;
  wire accB_clk_ena2;
  wire accB_clk_ena3;
  wire accB_clk_ena4;
  //wire accB_clk_ena5;
  assign accB_clk_ena = accB_clk_ena1 | accB_clk_ena2 | accB_clk_ena3 |
accB_clk_ena4 | accAB_clk_ena1 | accAB_clk_ena2 | accAB_clk_ena3 |
accAB_clk_ena4;// | accAB_clk_ena5;

  wire add_sub_B1;
  assign add_sub_B = add_sub_B1 | add_sub_AB1;

  wire multB_clk_ena1;
  assign multB_clk_ena = multB_clk_ena1 | multAB_clk_ena1;

  //convert control signals
  wire regB_wr_ena1;
  wire regB_wr_ena2;
  wire regB_wr_ena3;

  wire fbin_latch_sel1;
  wire fbin_latch_ena1;
  wire fbin_latch_reset1;
  wire wbin_latch_sel1;
  wire wbin_latch_ena1;
  //wire wbin_latch_reset1;
  wire fb2fr_load1;
  wire fb2fr_done1;
  wire fb2fr_shft_ena1;
  wire wb2wr_load1;
  wire wb2wr_shft_ena1;
  wire wb2wr_reset1;
  wire fb2fr_acc_ena1;
  wire wb2wr_acc_ena1;
  wire fb2fr_rnd_sel1;
```

```
  assign RegB_wr_ena = regB_wr_ena1 | regB_wr_ena2 | regB_wr_ena3 |
regAB_wr_ena1;
  assign fbin_latch_sel = fbin_latch_sel1;
  //assign fbin_latch_ena = fbin_latch_ena1;
  //assign fbin_latch_reset = fbin_latch_reset1;
  assign wbin_latch_sel = wbin_latch_sel1;
  assign wbin_latch_ena = wbin_latch_ena1;
  //assign wbin_latch_reset = wbin_latch_reset1;
  assign fb2fr_load = fb2fr_load1;
  assign fb2fr_done = fb2fr_done1;
  assign fb2fr_shft_ena = fb2fr_shft_ena1;
  assign wb2wr_load = wb2wr_load1;
  assign wb2wr_shft_ena = wb2wr_shft_ena1;
  assign wb2wr_reset = wb2wr_reset1;
  assign fb2fr_acc_ena = fb2fr_acc_ena1;
  assign wb2wr_acc_ena = wb2wr_acc_ena1;
  assign fb2fr_rnd_sel = fb2fr_rnd_sel1;

  wire [1:0] regB_sel_in1;
  wire [1:0] regB_sel_in2;

  assign regB_sel_in[1:0] = regB_sel_in1[1:0] | regB_sel_in2[1:0];


//*******************************************************************A
LU SELECT
CONTROLS************************************************************
******************************
          assign accA_sel[1:0] = (accA_clk_ena1 | accAB_clk_ena1) ? {2'b0} :
((accA_clk_ena2 | accAB_clk_ena2 | accA_clk_ena3 | accAB_clk_ena3) ? {2'b01} :
(accA_clk_ena4 | accAB_clk_ena4 | accA_clk_ena5) ? (2'b10): ((fmult_accA_clk_ena1|
fmult_accA_clk_ena2) ? (2'b11):{2'b0}));
   assign accB_sel[1:0] = (accB_clk_ena1 | accAB_clk_ena1) ? {2'b0} : ((accB_clk_ena2
| accAB_clk_ena2 | accB_clk_ena3 | accAB_clk_ena3) ? {2'b01} : ((accB_clk_ena4 |
accAB_clk_ena4) ? (2'b10):{2'b0}));

 //
**********************************************************************
**********************************************************************
************************************/

  //
**********************************************************************
*********************************/
```

```
 always @ (posedge clk or posedge reset)                        // main module to set-up
ALU clock and control state count
 begin
 if (reset)
          begin
                    aclk_ff <= 0;
          end
 else
   begin
                    if(start)
                    begin
                              aclk_ff <= 0;
                    end
                    else if (clk_en & ~done)
                              begin
                                        aclk_ff <= ~aclk_ff;
                              end
                    else
                              begin
                              aclk_ff <= 0;
                              end
           end

end


//SENSITIVITY LIST FOR COUNT///
 always @ (posedge aclk or posedge start)     //
begin
            if (start)
                              begin
                                        cnt <= 0;
                              end
                    else if(clk_en & ~done)   //count will only update during valid period
                              begin
                                        cnt <= cnt + 1;
                              end
                    else
                              begin
                                        cnt <= 0;
                              end
end


                                                                               // OPCODE defines
```

```
// Generation of the alu_ena signal allows for the ALU to be switched over to the NIOS
instruction control lines
// the only time this is not desired is during a clock pulse instruction for software
controlled operation.
reg alu_ena_ff;                                   // storage for the alu_ena signal generation

assign alu_ena = alu_ena_ff;


`define CLK_PULSE_OP            255

always @ (posedge clk or posedge reset)
begin

	if(reset)
		begin
			alu_ena_ff <= 0;
		end
	else
		begin

			if(clk_en)
				begin
					if(n != `CLK_PULSE_OP)
						alu_ena_ff <= 1;
					else
						alu_ena_ff <= 0;
				end
			else
				alu_ena_ff <= 0;
		end

end


// Need to finalize datapath definition to fulfill instruction definitions
// and to reduce changes to the verilog code
//d.a. may need to move define to top so that they can be used in later codes sections
(refer to combinational logic)

`define DEMO_OP                 0                       // Note that the define needs a
grave mark preceding it
`define LOAD_REG_A              1
`define LOAD_REG_B              2
`define LOAD_AB                 3
```

```
`define STORE_A             4
`define STORE_B             5
`define STORE_AB            6
`define MOVE                7
`define ADDS_A              8                   // signed add
`define ADDS_B              9
`define ADDS_AB             10                  // adds both accumulators
simultaneously
`define ADDS_A_STORE        11                  // adds the accumulator with
register and stores in register
`define SUBS_A              12
`define SUBS_B              13
`define SUBS_AB             14
`define SUBS_A_STORE        15
`define MULTS_A             16
`define MULTS_B             17
`define MULTS_AB            18
`define MULTS_A_STORE       19
`define DIVU_A              20
`define DIVU_B              21
`define DIVS_A              22
`define DIVS_B              23
`define FMULT_A             24
`define FMULT_B             25
`define FMULT_A_STORE       26
`define FMULT_B_STORE       27
`define FDIV_A              28
`define FDIV_B              29
`define COMPS_A             30
`define COMPS_B             31
`define COMPS_AB            32
`define COMPS               33                  // register to register compare
`define COMP_A              34
`define COMP_B              35
`define COMP_AB             36
`define COMP                37                  // register to register compare
`define CNVRTFS             38                  // convert integer forward
signed
`define FCNVRTF             39                  // fractional forward convert
(always signed)
`define CNVRTRS             40                  // convert integer reverse
`define FCNVRTR             41                  // convert fractional reverse
`define RET_TEST            42                  //test for returning data to
nios
`define NORM_A              43
```

```
My_Firsts first_one(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out), .done_flag(inst1_done));
                    // add your instructions here
Clock_Pulse pulse1(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out), .done_flag(inst2_done));
                    // instruction for generating a single ALU clock pulse
Load_accumA load_A(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst3_done), .reg_clk_ena(regA_clk_ena1), .acc_clk_ena(accA_clk_ena1));
Load_accumB load_B(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst4_done), .reg_clk_ena(regB_clk_ena1), .acc_clk_ena(accB_clk_ena1));
Store_accumA store_A(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst18_done), .reg_clk_ena(regA_clk_ena7), .reg_wr_ena(regA_wr_ena1));
Store_accumB store_B(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst21_done), .reg_clk_ena(regB_clk_ena7), .reg_wr_ena(regB_wr_ena3));
Store_accumAB store_AB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst22_done), .reg_clk_ena(regAB_clk_ena5),
.reg_wr_ena(regAB_wr_ena1));

Test_return_data test_return_data(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst20_done), /*.reg_clk_ena(regA_clk_ena42),
.acc_clk_ena(accA_clk_ena42),*/ .result(result1));

Comp_accumA comp_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst19_done), .reg_clk_ena(regA_clk_ena6),
/*.acc_clk_ena(AccA_clk_ena6),*/ .cross_shft_init(cmp_shftA_init),
.cross_shft_ena(cmp_shftA_ena),.decomp_subload(cmp_decompA_subload),
.modcnt_clk_ena(cmp_modcntA_clk_ena), .moddiv_ena(cmp_moddivA_ena),
.clear_skip(cmp_clr_skipA_ena), .modcnt_reset(cmp_modcntA_rst),
.status(cmp_out_status), .result(result2));
Add_accumA add_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst5_done), .reg_clk_ena(regA_clk_ena2), .acc_clk_ena(accA_clk_ena2));
Add_accumB add_accumB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst6_done), .reg_clk_ena(regB_clk_ena2), .acc_clk_ena(accB_clk_ena2));
Load_accumAB load_AB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst7_done), .reg_clk_ena(regAB_clk_ena1),
.acc_clk_ena(accAB_clk_ena1));


Add_accumAB add_accumAB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst8_done), .reg_clk_ena(regAB_clk_ena2),
.acc_clk_ena(accAB_clk_ena2));
Sub_accumA sub_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst9_done), .reg_clk_ena(regA_clk_ena3), .acc_clk_ena(accA_clk_ena3),
.add_sub(add_sub_A1));
Sub_accumB sub_accumB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst10_done), .reg_clk_ena(regB_clk_ena3), .acc_clk_ena(accB_clk_ena3),
.add_sub(add_sub_B1));
```

```
Sub_accumAB sub_accumAB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst11_done), .reg_clk_ena(regAB_clk_ena3),
.acc_clk_ena(accAB_clk_ena3),  .add_sub(add_sub_AB1));
Mult_accumA mult_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst12_done), .reg_clk_ena(regA_clk_ena4), .acc_clk_ena(accA_clk_ena4),
.mult_clk_ena(multA_clk_ena1));
Mult_accumB mult_accumB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst13_done), .reg_clk_ena(regB_clk_ena4), .acc_clk_ena(accB_clk_ena4),
.mult_clk_ena(multB_clk_ena1));
Mult_accumAB mult_accumAB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst14_done), .reg_clk_ena(regAB_clk_ena4),
.acc_clk_ena(accAB_clk_ena4),  .mult_clk_ena(multAB_clk_ena1));
Fmult_accumA fmult_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst15_done), .reg_clk_ena(regA_clk_ena5), .acc_clk_ena(accA_clk_ena5),
.fmult_acc_clk_ena(fmult_accA_clk_ena1), .mult_clk_ena(multA_clk_ena2),

        .cross_shft_init(cross_shftA_init1), .cross_shft_ena(cross_shftA_ena1),
.decomp_subload(decompA_subload1), .modmult_ena(modmultA_ena1),
.modmult_acc_ena(modmultA_acc_ena1),

        .modmult_acc_clr(modmultA_acc_clr1), .modcnt_clk_ena(modcntA_clk_ena1),
.moddiv_ena(divA_clk_ena1), .clear_skip(clr_skipA_ena1),
.modcnt_reset(modcntA_rst1), .start_latch(start_lat_A1),

        .macBcomp_ena(macBcompA_ena1), .mac_select(macA_select1),
.fmult_status(fmultA_status), .acc_clear(accA_clear1));     //*/

Cnvrts_regB cnvrts_regB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst16_done), .reg_clk_ena(regB_clk_ena5), .reg_wr(regB_wr_ena1),
.reg_sel_in(regB_sel_in1), .wb2wr_load(wb2wr_load1),
.wb2wr_acc_ena(wb2wr_acc_ena1),

        .wb2wr_shft_ena(wb2wr_shft_ena1), .wbin_latch_ena(wbin_latch_ena1),
.wbin_latch_sel(wbin_latch_sel1), .wb2wr_reset(wb2wr_reset1));

 Fcnvrt_regB fcnvrt_regB(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst17_done), .reg_clk_ena(regB_clk_ena6), .reg_wr(regB_wr_ena2),
.reg_sel_in(regB_sel_in2), .fb2fr_load(fb2fr_load1), .fb2fr_acc_ena(fb2fr_acc_ena1),
 .fb2fr_shft_ena(fb2fr_shft_ena1), .fbin_latch_ena(fbin_latch_ena1),
.fbin_latch_sel(fbin_latch_sel1), /*.fb2fr_reset(fb2fr_reset1),*/
.fb2fr_rnd_sel(fb2fr_rnd_sel1), .fb2fr_done(fb2fr_done1));


Norm_accumA norm_accumA(.aclk(aclk), .reset(start), .n(n), .cnt(cnt_out),
.done_flag(inst23_done), .reg_clk_ena(regA_clk_ena8), .acc_clk_ena(accA_clk_ena6),
.fmult_acc_clk_ena(fmult_accA_clk_ena2),
```

```
	.cross_shft_init(cross_shftA_init2), .cross_shft_ena(cross_shftA_ena2),
.decomp_subload(decompA_subload2), .modmult_ena(modmultA_ena2),
.modmult_acc_ena(modmultA_acc_ena2),

	.modmult_acc_clr(modmultA_acc_clr2), .modcnt_clk_ena(modcntA_clk_ena2),
.moddiv_ena(divA_clk_ena2), .clear_skip(clr_skipA_ena2),
.modcnt_reset(modcntA_rst2), .start_latch(start_lat_A2),

	.macBcomp_ena(macBcompA_ena2), .mac_select(macA_select2),
.fmult_status(fmultA_status), .acc_clear(accA_clear2));	//*/



endmodule



//
*************************************************************************
********************************
// Instruction template model
// The following instruction is a template for all NIOS extended instructions

module My_Firsts(aclk, reset, n, cnt, done_flag);			// prototype
instruction

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;

reg done;

assign done_flag = done;

always @(posedge aclk or posedge reset)
begin

	if(reset)
		begin
		done <= 0;
		end
	else
		if(n == `DEMO_OP)			// put your instruction opcode here
```

```
			begin
			if(cnt < 1)					// extend your cycle counts here
				done <= 0;
			else
				done <= 1;
			end

end


endmodule

// ---------------------------


// Instruction to generate a single ALU clock pulse.
// This will replace the PIO port method of driving a single clock pulse for our ALU
development environment.

module Clock_Pulse(aclk, reset, n, cnt, done_flag);			// this instruction
simply provides a single clock pulse to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;

reg done;

assign done_flag = done;

always @(posedge aclk or posedge reset)
begin

	if(reset)
			begin
			done <= 0;
			end
	else
			if(n == `CLK_PULSE_OP)					// put your instruction
code here
			begin
//			if(cnt < 1)					// extend your cycle counts here
				done <= 1;
			end
```

```
end

endmodule

// ---------------------------------------------------------------------------------------------------



//simple test of returning data to the nios

module Test_return_data(aclk, reset, n, cnt, done_flag, result);

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output [31:0] result;

reg done;
reg [31:0] result_ena_ff;

assign done_flag = done;
assign result = result_ena_ff;

always @(posedge aclk or posedge reset)
begin

	if(reset)
		done <= 0;
	else
		begin
		result_ena_ff <= 0;
		if(n == `RET_TEST)				// decode the LOAD
instruction
			begin
			if(cnt < 1)							//
assert the acc clk ena here
				begin
				done <= 0;
				result_ena_ff <= 305419896;
				end
			else
				begin
				done <= 1;
```

```
                    end
                end
            end

end


endmodule

//-----------------------------------------------------------------------------------------------------------*/



// LoadA Register instruction  (load accum from register file)
module Load_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
    // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin

    if(reset)
        begin
        done <= 0;

        if(n == `LOAD_REG_A)
            reg_clk_ena_ff <= 1;                    // first cycle, kick off
the reg_clk_ena
        else
            reg_clk_ena_ff <= 0;
```

```
                    acc_clk_ena_ff <= 0;
                    end
          else
                    begin
                    reg_clk_ena_ff <= 0;                              // (place this outside the
conditional for better logic!)

                    if(n == `LOAD_REG_A)                                        // decode the LOAD
instruction
                              begin

                              if(cnt < 1)                                                        //
assert the acc clk ena here
                                        begin
                                        done <= 0;
                                        acc_clk_ena_ff <= 1;
                                        end
                              else
                                        begin
                                        done <= 1;
                                        acc_clk_ena_ff <= 0;
                                        end
                              end
                    end
end

endmodule

/////////////////////////////////////////////////////////////////////////*/

// Load_B Register instruction  (load accum from register file)
module Load_accumB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
          // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
```

```
assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `LOAD_REG_B)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off
the reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;

                acc_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;                    // (place this outside the
conditional for better logic!)

                if(n == `LOAD_REG_B)                    // decode the LOAD
instruction
                        begin

                        if(cnt < 1)                                     //
assert the acc clk ena here
                                begin
                                done <= 0;
                                acc_clk_ena_ff <= 1;
                                end
                        else
                                begin
                                done <= 1;
                                acc_clk_ena_ff <= 0;
                                end
                        end
                end

end

endmodule
```

```
// ----------------------------------------------------------------------------------------------------
---


// Store_A
module Store_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, reg_wr_ena);


input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output reg_wr_ena;


reg done;
reg reg_clk_ena_ff;
reg reg_wr_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign reg_wr_ena = reg_wr_ena_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `STORE_A)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off
the reg_clk_ena to read register (may not be needed for store)
                else
                        reg_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;                    // (place this outside the
conditional for better logic!)

                if(n == `STORE_A)                       // decode the STORE
instruction
                        begin
```

```
                    if(cnt == 0)
                        begin
                        done <= 0;
                        reg_clk_ena_ff <= 1;
                        reg_wr_ena_ff <= 1;
                        end
                    else if(cnt == 1)
                        begin
                        done <= 1;
                        reg_clk_ena_ff <= 0;
                        reg_wr_ena_ff <= 0;
                        end
                    end
            end
end

endmodule


// ----------------------------------------------------------------------------------------------------------
---
// Store_B
module Store_accumB(aclk, reset, n, cnt, done_flag, reg_clk_ena, reg_wr_ena);


input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output reg_wr_ena;


reg done;
reg reg_clk_ena_ff;
reg reg_wr_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign reg_wr_ena = reg_wr_ena_ff;

always @(posedge aclk or posedge reset)
begin
```

```
	if(reset)
		begin
		done <= 0;

		if(n == `STORE_B)
			reg_clk_ena_ff <= 1;				// first cycle, kick off
the reg_clk_ena to read register (may not be needed for store)
		else
			reg_clk_ena_ff <= 0;
		end
	else
		begin
		reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

		if(n == `STORE_B)				// decode the STORE
instruction
			begin

			if(cnt == 0)
				begin
				done <= 0;
				reg_clk_ena_ff <= 1;
				reg_wr_ena_ff <= 1;
				end
			else if(cnt == 1)
				begin
				done <= 1;
				reg_clk_ena_ff <= 0;
				reg_wr_ena_ff <= 0;
				end
			end
		end
end

endmodule
// ---------------------------------------------------------------------------------------------------
---


// ---------------------------------------------------------------------------------------------------
---
// Store_AB
module Store_accumAB(aclk, reset, n, cnt, done_flag, reg_clk_ena, reg_wr_ena);
```

```
input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output reg_wr_ena;


reg done;
reg reg_clk_ena_ff;
reg reg_wr_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign reg_wr_ena = reg_wr_ena_ff;

always @(posedge aclk or posedge reset)
begin

	if(reset)
		begin
		done <= 0;

		if(n == `STORE_AB)
			reg_clk_ena_ff <= 1;				// first cycle, kick off
the reg_clk_ena to read register (may not be needed for store)
		else
			reg_clk_ena_ff <= 0;
		end
	else
		begin
		reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

		if(n == `STORE_AB)				// decode the STORE
instruction
			begin

			if(cnt == 0)
				begin
				done <= 0;
				reg_clk_ena_ff <= 1;
				reg_wr_ena_ff <= 1;
				end
			else if(cnt == 1)
```

```
                                        begin
                                        done <= 1;
                                        reg_clk_ena_ff <= 0;
                                        reg_wr_ena_ff <= 0;
                                        end
                              end
                    end
end

endmodule
// -----------------------------------------------------------------------------------------------------
---


// ADD_A Register instruction  (add accum with register file)
module Add_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
          // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin
          if(reset)
                    begin
                    done <= 0;

                    if(n == `ADDS_A)
                              reg_clk_ena_ff <= 1;                    // first cycle, kick off the
reg_clk_ena
                    else
                              reg_clk_ena_ff <= 0;
```

```
			acc_clk_ena_ff <= 0;
			end
		else
			begin
			reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

			if(n == `ADDS_A)				// decode the LOAD
instruction
				begin

				if(cnt < 1)				// assert the acc clk ena here
					begin
					done <= 0;
					acc_clk_ena_ff <= 1;
					end
				else
					begin
					done <= 1;
					acc_clk_ena_ff <= 0;
					end
				end
			end

end

endmodule
//----------------------------------------------------------------------------------------------------




//----------------------------------------------------------------------------------------------------
// ADD_B Register instruction  (add accum with register file)
module Add_accumB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
	// load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
```

```
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin
        if(reset)
                begin
                done <= 0;

                if(n == `ADDS_B)
                                reg_clk_ena_ff <= 1;            // first cycle, kick off the
reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;

                acc_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;                            // (place this outside the
conditional for better logic!)

                if(n == `ADDS_B)                                // decode the LOAD
instruction
                        begin

                        if(cnt < 1)                             // assert the acc clk ena here
                                begin
                                done <= 0;
                                acc_clk_ena_ff <= 1;
                                end
                        else
                                begin
                                done <= 1;
                                acc_clk_ena_ff <= 0;
                                end
                        end
                end

end

endmodule
```

```
//------------------------------------------------------------------------------

// Load_AB Register instruction  (load both accum from register files)
module Load_accumAB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
          // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin

          if(reset)
                    begin
                    done <= 0;

                    if(n == `LOAD_AB)
                              reg_clk_ena_ff <= 1;                              // first cycle, kick off
the reg_clk_ena
                    else
                              reg_clk_ena_ff <= 0;

                    acc_clk_ena_ff <= 0;
                    end
          else \
                    begin
                    reg_clk_ena_ff <= 0;                              // (place this outside the
conditional for better logic!)
                    if(n == `LOAD_AB)                              // decode the LOAD
instruction
                              begin
```

```
				if(cnt < 1)							//
assert the acc clk ena here
					begin
					done <= 0;
					acc_clk_ena_ff <= 1;
					end
				else
					begin
					done <= 1;
					acc_clk_ena_ff <= 0;
					end
				end
			end

end

endmodule

// -----------------------------------------------------------------------------------------------------
---*/

// ADD_AB Register instruction  (add accum with register file)
module Add_accumAB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena);
	// load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;


always @(posedge aclk or posedge reset)
begin
	if(reset)
		begin
```

```
			done <= 0;

			if(n == `ADDS_AB)
				reg_clk_ena_ff <= 1;			// first cycle, kick off the
reg_clk_ena
			else
				reg_clk_ena_ff <= 0;

			acc_clk_ena_ff <= 0;
			end
		else
			begin
			reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

			if(n == `ADDS_AB)				// decode the LOAD
instruction
				begin

				if(cnt < 1)				// assert the acc clk ena here
					begin
					done <= 0;
					acc_clk_ena_ff <= 1;
					end
				else
					begin
					done <= 1;
					acc_clk_ena_ff <= 0;
					end
				end
			end

end

endmodule
// -----------------------------------------------------------------------------------------------------------
---


// SUB_A Register instruction  (sub accum with register file)
module Sub_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena, add_sub);
			// load to ALU

input aclk;
input reset;
input [7:0] n;
```

```
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output add_sub;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg add_sub_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign add_sub = add_sub_ff;

always @(posedge aclk or posedge reset)
begin
	if(reset)
		begin
		done <= 0;

		if(n == `SUBS_A)
			reg_clk_ena_ff <= 1;			// first cycle, kick off the
reg_clk_ena
		else
			reg_clk_ena_ff <= 0;

		acc_clk_ena_ff <= 0;
		end
	else
		begin
		reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

		if(n == `SUBS_A)				// decode the LOAD
instruction
			begin

			if(cnt < 1)					// assert the
acc clk ena here
				begin
				done <= 0;
				acc_clk_ena_ff <= 1;
				add_sub_ff <= 1;
				end
```

```
                else
                        begin
                        done <= 1;
                        acc_clk_ena_ff <= 0;
                        add_sub_ff <= 0;
                        end
                end
        end

end

endmodule

// -----------------------------------------------------------------------------------------------------
---


// SUB_B Register instruction  (sub accum with register file)
module Sub_accumB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena, add_sub);
        // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output add_sub;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg add_sub_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign add_sub = add_sub_ff;

always @(posedge aclk or posedge reset)
begin
    if(reset)
        begin
        done <= 0;
```

```
			if(n == `SUBS_B)
				reg_clk_ena_ff <= 1;				// first cycle, kick off
the reg_clk_ena
			else
				reg_clk_ena_ff <= 0;

			acc_clk_ena_ff <= 0;
			end
		else
			begin
			reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

			if(n == `SUBS_B)				// decode the LOAD
instruction
				begin

				if(cnt < 1)				// assert the acc clk ena here
					begin
					done <= 0;
					acc_clk_ena_ff <= 1;
					add_sub_ff <= 1;
					end
				else
					begin
					done <= 1;
					acc_clk_ena_ff <= 0;
					add_sub_ff <= 0;
					end
				end
			end

end

endmodule
//----------------------------------------------------------------------------------------------------

// -----------------------------------------------------------------------------------------------------
---


// SUB_AB Register instruction  (sub accum with register file)
module Sub_accumAB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
add_sub);				// load to ALU

input aclk;
```

```
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output add_sub;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg add_sub_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign add_sub = add_sub_ff;

always @(posedge aclk or posedge reset)
begin
        if(reset)
                begin
                done <= 0;

                if(n == `SUBS_AB)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off the
reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;

                acc_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;                            // (place this outside the
conditional for better logic!)

                if(n == `SUBS_AB)                               // decode the LOAD
instruction
                        begin

                        if(cnt < 1)                             // assert the acc clk ena here
                                begin
                                done <= 0;
                                acc_clk_ena_ff <= 1;
                                add_sub_ff <= 1;
```

```
                    end
               else
                    begin
                    done <= 1;
                    acc_clk_ena_ff <= 0;
                    add_sub_ff <= 0;
                    end
               end
          end

end

endmodule
//------------------------------------------------------------------------------------------------------



//------------------------------------------------------------------------------------------------------

// MULTS_A Register instruction  (mult accum with register file)
module Mult_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
mult_clk_ena);                          // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output mult_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg mult_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign mult_clk_ena = mult_clk_ena_ff;

always @(posedge aclk or posedge reset)
```

```
begin

	if(reset)
		begin
		done <= 0;

		if(n == `MULTS_A)
			reg_clk_ena_ff <= 1;			// first cycle, kick off the
reg_clk_ena
		else
			reg_clk_ena_ff <= 0;

		mult_clk_ena_ff <= 0;
		acc_clk_ena_ff <= 0;
		end
	else
		begin
		reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

		if(n == `MULTS_A)				// decode the LOAD
instruction
			begin

			if (cnt == 0)
				begin
				done <= 0;
				mult_clk_ena_ff <= 1;
				end
			else if(cnt == 1)
	// assert the acc clk ena here
				begin
				done <= 0;
				mult_clk_ena_ff <= 0;
				acc_clk_ena_ff <= 1;
				end
			else if (cnt >= 2)
				begin
				done <= 1;
				acc_clk_ena_ff <= 0;
				end
			end
		end

end
```

```
endmodule

//------------------------------------------------------------------------------------*/
//--------------------------------------------------------------------------------------------------------

// MULTS_B Register instruction  (mult accum with register file)
module Mult_accumB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
mult_clk_ena);                              // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output mult_clk_ena;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg mult_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign mult_clk_ena = mult_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `MULTS_B)
                        reg_clk_ena_ff <= 1;                        // first cycle, kick off
the reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;

                mult_clk_ena_ff <= 0;
                acc_clk_ena_ff <= 0;
                end
        else
                begin
```

```
                    reg_clk_ena_ff <= 0;                              // (place this outside the
conditional for better logic!)

                    if(n == `MULTS_B)                                  // decode the LOAD
instruction
                              begin

                              if (cnt == 0)
                                        begin
                                        done <= 0;
                                        mult_clk_ena_ff <= 1;
                                        end
                              else if(cnt == 1)
          // assert the acc clk ena here
                                        begin
                                        done <= 0;
                                        mult_clk_ena_ff <= 0;
                                        acc_clk_ena_ff <= 1;
                                        end
                              else if (cnt >=2)
                                        begin
                                        done <= 1;
                                        acc_clk_ena_ff <= 0;
                                        end
                              end
                    end

end

endmodule

//------------------------------------------------------------------------------*/
//----------------------------------------------------------------------------------------------------

// MULTS_AB Register instruction  (mult accum with register file)
module Mult_accumAB(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
mult_clk_ena);                              // load to ALU

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output mult_clk_ena;
```

```
reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg mult_clk_ena_ff;

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign mult_clk_ena = mult_clk_ena_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `MULTS_AB)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off the
reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;

                mult_clk_ena_ff <= 0;
                acc_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;                            // (place this outside the
conditional for better logic!)

                if(n == `MULTS_AB)                              // decode the LOAD
instruction
                        begin

                        if (cnt == 0)
                                begin
                                done <= 0;
                                mult_clk_ena_ff <= 1;
                                end
                        else if(cnt == 1)
        // assert the acc clk ena here
                                begin
                                done <= 0;
                                mult_clk_ena_ff <= 0;
```

```
                    acc_clk_ena_ff <= 1;
                    end
            else if (cnt >=2)
                    begin
                    done <= 1;
                    acc_clk_ena_ff <= 0;
                    end
            end
    end

end
//
endmodule

//-------------------------------------------------------------------------------*/

// FMULT_A Register instruction  (fractional multiply)
module Fmult_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
mult_clk_ena, fmult_acc_clk_ena, cross_shft_init, cross_shft_ena,
decomp_subload, modmult_ena, modmult_acc_ena, modmult_acc_clr,  modcnt_clk_ena,
moddiv_ena, clear_skip, modcnt_reset,
start_latch, macBcomp_ena, mac_select, fmult_status, acc_clear);



input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
input [8:0] fmult_status; //status and compare bits

output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output mult_clk_ena;  //needed to perform multiply before decomp
output fmult_acc_clk_ena;  //separate acc clk for fmult

output cross_shft_ena;
output cross_shft_init;
output decomp_subload;
output modmult_ena;
output modmult_acc_ena;
output modmult_acc_clr;
output modcnt_clk_ena;
output moddiv_ena;
output clear_skip;
```

```
output modcnt_reset;
output start_latch;
output macBcomp_ena;
output mac_select;
output acc_clear;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg fmult_acc_clk_ena_ff;
reg cross_shft_ena_ff;
reg cross_shft_init_ff;
reg decomp_subload_ff;
reg modmult_ena_ff;
reg modmult_acc_ena_ff;
reg modmult_acc_clr_ff;
reg modcnt_clk_ena_ff;
reg moddiv_ena_ff;
reg clear_skip_ff;
reg modcnt_reset_ff;
reg start_latch_ff;
reg macBcomp_ena_ff;
reg mac_select_ff;
reg mult_clk_ena_ff;
reg [1:0]fmult_status_val; //holds status of fmult at completion time
reg [1:0]comp_state;   //holds comparison value
reg [4:0]tmp_cnt;
reg divclk_ena;            //moddiv and modclk enables
reg acc_clear_ff;

assign acc_clear = acc_clear_ff;
assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign fmult_acc_clk_ena = fmult_acc_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign cross_shft_ena = cross_shft_ena_ff;
assign cross_shft_init = cross_shft_init_ff;
assign decomp_subload = decomp_subload_ff;
assign modmult_ena = modmult_ena_ff;
assign modmult_acc_ena = modmult_acc_ena_ff;
assign modmult_acc_clr = modmult_acc_clr_ff;
assign modcnt_clk_ena = divclk_ena | modcnt_clk_ena_ff;
assign moddiv_ena =  divclk_ena | moddiv_ena_ff;
assign clear_skip = clear_skip_ff;
assign modcnt_reset = modcnt_reset_ff;
assign start_latch = start_latch_ff;
```

```
assign macBcomp_ena = macBcomp_ena_ff;
assign mac_select = mac_select_ff;
assign mult_clk_ena = mult_clk_ena_ff;


always @(posedge aclk or posedge reset)
begin

	if(reset)
		begin
		done <= 0;

		if(n == `FMULT_A)
			begin
			reg_clk_ena_ff <= 1;				// first cycle, kick off
the reg_clk_ena
			acc_clear_ff <= 0;				//make sure accum is
not cleared
			end
		else
			reg_clk_ena_ff <= 0;

		mult_clk_ena_ff <= 0;
		acc_clk_ena_ff <= 0;
		fmult_acc_clk_ena_ff <= 0;
		acc_clear_ff <= 0;				//make sure accum is not
cleared
		end
	else
		begin
		reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

		if(n == `FMULT_A)
			begin //global reset for all cases
				done <= 0;
				mult_clk_ena_ff <= 0;
				acc_clk_ena_ff <= 0;
				cross_shft_ena_ff <= 0;
				cross_shft_init_ff <= 0;
				decomp_subload_ff <= 0;
				modmult_ena_ff <= 0;
				modmult_acc_ena_ff <= 0;
				modmult_acc_clr_ff <= 0;
				clear_skip_ff <= 0;
				modcnt_reset_ff <= 0;
```

```
				start_latch_ff <= 0;
				divclk_ena <= 0;
				modcnt_clk_ena_ff <= 0;
				moddiv_ena_ff <= 0;
			if (cnt == 0)					//perform
multiplication first
				mult_clk_ena_ff <= 1;
			else if(cnt == 1)
				acc_clk_ena_ff <= 1;
			else if(cnt == 2)
	//mult done now do decomp
				cross_shft_init_ff <= 1;
			else if(cnt == 3)
	// assert the acc clk ena here
				begin
				modcnt_clk_ena_ff <= 1;
				modcnt_reset_ff <= 1;
				start_latch_ff <= 1;
				end
			else if(cnt == 4)
	// assert the acc clk ena here
				begin
				divclk_ena <= 1;
				clear_skip_ff <= 1;
				end
			else if(cnt == 5)
	// first iteration of load; from AccumA
				begin
				cross_shft_ena_ff <= 1;
				decomp_subload_ff <= 1;
				divclk_ena <= 1;
				end
			else if((cnt >= 6) & (cnt <= 11))
		//decomp start here
				begin
				cross_shft_ena_ff <= 1;
				divclk_ena <= 1;
				end
			else if(cnt == 12)
	// assert the acc clk ena here
				begin
				cross_shft_ena_ff <= 1;
				modmult_acc_clr_ff <= 1;
				modmult_ena_ff <= 1;
				divclk_ena <= 1;
				end
```

```
                        else if(cnt == 13)
        // assert the acc clk ena here
                                begin
                                cross_shft_ena_ff <= 1;
                                modmult_ena_ff <= 1;
                                divclk_ena <= 1;
                                end
                        else if ((fmult_status & 'h3) == 0)             //continue decomp
while status is not 1,2,3
                                begin
                                cross_shft_ena_ff <= 1;
                                modmult_ena_ff <= 1;
                                modmult_acc_ena_ff <= 1;
                                divclk_ena <= 1;
                                tmp_cnt <= cnt;
                                end
                        else if (~((fmult_status & 'h3) == 0))
        //status must be 1,2,3,
                                begin
                                if (cnt == (tmp_cnt + 1))
                                        begin
                                                fmult_acc_clk_ena_ff <= 0;
                                                modmult_acc_ena_ff <= 1;
                                                macBcomp_ena_ff <= 0;
                                                mac_select_ff <= 0;
                                                fmult_status_val <= fmult_status[1:0];
//saves value of fmult status since it will change during next clk
                                                comp_state <= fmult_status[5:4];
                                        end
                                else if (cnt == (tmp_cnt + 2))   //start checking compare
states of accumA and B
                                        begin
                                                if (fmult_status_val == 1)
                                                        fmult_acc_clk_ena_ff <= 1;
                                                else if (fmult_status_val == 2)
                                                begin
                                                        macBcomp_ena_ff <= 1;
                                                        mac_select_ff <= 1;  //make sure so
hold until accum is latched
                                                end
                                                else if(fmult_status_val == 3)
                                                begin
                                                        if (comp_state == 2) //A>B take
compliment
                                                        begin
                                                                macBcomp_ena_ff <= 1;
```

```
                                        mac_select_ff <= 1;
                                end
                        else if (comp_state == 1) //cmp_cnt < 1
                                        acc_clear_ff <= 1;  //clear the
accum since it is empty
                                else            //A=B  accum values
are both 0 (empty) if not greater than it will be equal or less than; in both cases we latch
the final value of accum
                                        fmult_acc_clk_ena_ff <= 1;
                        end
                end
            else if((cnt >= (tmp_cnt + 3)))
                begin
                        if ((fmult_acc_clk_ena == 1)  || (acc_clear
== 1))
                        begin
                                done <= 1;
                                fmult_acc_clk_ena_ff <= 0;
                                mac_select_ff <= 0; //ensures select
was held long enough
                                acc_clear_ff <= 0;
                                //macBcomp_ena_ff <= 0;
                        end
                else //the only other reason is for the compliment
                        begin
                                fmult_acc_clk_ena_ff <= 1;
                                macBcomp_ena_ff <= 0;
                        end
                end
            end
        end
    end
end
endmodule

//-----------------------------------------------------------------------------------*/

// CNVRTFS_Rb Register instruction  (convert whole number integer to RNS)
module Cnvrts_regB(aclk, reset, n, cnt, done_flag, reg_clk_ena, reg_wr, reg_sel_in,
wb2wr_load, wb2wr_acc_ena, wb2wr_shft_ena, wbin_latch_ena, wbin_latch_sel,
wb2wr_reset);

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
```

```
output done_flag;
output reg_clk_ena;
output reg_wr;
output [1:0] reg_sel_in;          //can be 1,2, or 3 based on conversion
output wb2wr_load;
output wb2wr_acc_ena;
output wb2wr_shft_ena;
output wbin_latch_ena;
output wbin_latch_sel;
output wb2wr_reset;

reg done;
reg reg_clk_ena_ff;
reg reg_wr_ff;
reg [1:0]reg_sel_in_ff;
reg wb2wr_load_ff;
reg wb2wr_acc_ena_ff;
reg wb2wr_shft_ena_ff;
reg wbin_latch_ena_ff;
reg wbin_latch_sel_ff;
reg wb2wr_reset_ff;


assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign reg_wr = reg_wr_ff;
assign reg_sel_in = reg_sel_in_ff;
assign wb2wr_load = wb2wr_load_ff;
assign wb2wr_acc_ena = wb2wr_acc_ena_ff;
assign wb2wr_shft_ena = wb2wr_shft_ena_ff;
assign wbin_latch_ena = wbin_latch_ena_ff;
assign wbin_latch_sel = wbin_latch_sel_ff;
assign wb2wr_reset = wb2wr_reset_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `CNVRTFS)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off
the reg_clk_ena
```

```
			else
				begin
				reg_clk_ena_ff <= 0;
				reg_sel_in_ff <= 2'b0;
				end
			end
		else
			begin
			reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)
			reg_sel_in_ff <= 2'b0;
			if(n == `CNVRTFS)
				begin //global reset for all cases
					wb2wr_shft_ena_ff <= 0;
					wb2wr_load_ff <= 0;
					reg_clk_ena_ff <= 0;
					reg_wr_ff <= 0;
					reg_sel_in_ff <= 0;
					done <= 0;
					wb2wr_shft_ena_ff <= 0;
					wb2wr_acc_ena_ff <= 0;
					wb2wr_acc_ena_ff <= 0;
					wb2wr_reset_ff <= 0;

				case (cnt)
					//0:		wbin_latch_ena_ff <= 1;

				//	1:		begin
					//		wbin_latch_ena_ff <= 1;
					//		wbin_latch_sel_ff <= 1;
					//		end

					0:		begin //resets accum and shift reg
							wb2wr_acc_ena_ff <= 1;
							wb2wr_reset_ff <= 1;
							end

					1:		begin		//loads bits to converter unit
							wb2wr_shft_ena_ff <= 1;
							wb2wr_load_ff <=1;
							end

					10:	begin		//write value to register file
							reg_clk_ena_ff <= 1;
							reg_wr_ff <= 1;
							reg_sel_in_ff <= 2'b01;
```

```
                                        end
                        11:     done <=1;
                        default: begin  //loop counter; should run 8 times
                                        wb2wr_shft_ena_ff <= 1;
                                        wb2wr_acc_ena_ff <= 1;
                                        end
                endcase
                //next portion is to create valid sign bit



                end
        end
end
endmodule//*/

//---------------------------------------------------------------------------------*/

// FCNVRTF Register instruction  (fractional multiply)
module Fcnvrt_regB(aclk, reset, n, cnt, done_flag, reg_clk_ena, reg_wr, reg_sel_in,
fb2fr_load, fb2fr_acc_ena, fb2fr_shft_ena,
                                        fbin_latch_ena, fbin_latch_sel,
/*fb2fr_reset,*/ fb2fr_rnd_sel, fb2fr_done);

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;


output done_flag;
output reg_clk_ena;
output reg_wr;
output [1:0] reg_sel_in;        //can be 1,2, or 3 based on conversion
output fb2fr_load;
output fb2fr_acc_ena;
output fb2fr_shft_ena;
output fbin_latch_ena;
output fbin_latch_sel;
//output fb2fr_reset;
output fb2fr_rnd_sel;
output fb2fr_done;

reg done;
reg reg_clk_ena_ff;
reg reg_wr_ff;
```

```
reg [1:0]reg_sel_in_ff;
reg fb2fr_load_ff;
reg fb2fr_acc_ena_ff;
reg fb2fr_shft_ena_ff;
reg fbin_latch_ena_ff;
reg fbin_latch_sel_ff;
//reg fb2fr_reset_ff;
reg fb2fr_done_ff;
reg fb2fr_rnd_sel_ff;


assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign reg_wr = reg_wr_ff;
assign reg_sel_in = reg_sel_in_ff;
assign fb2fr_load = fb2fr_load_ff;
assign fb2fr_acc_ena = fb2fr_acc_ena_ff;
assign fb2fr_shft_ena = fb2fr_shft_ena_ff;
assign fbin_latch_ena = fbin_latch_ena_ff;
assign fbin_latch_sel = fbin_latch_sel_ff;
//assign fb2fr_reset = fb2fr_reset_ff;
assign fb2fr_done =    fb2fr_done_ff;
assign fb2fr_rnd_sel = fb2fr_rnd_sel_ff;


always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `FCNVRTF)
                        reg_clk_ena_ff <= 1;                    // first cycle, kick off
the reg_clk_ena
                else
                        begin
                        reg_clk_ena_ff <= 0;
                        reg_sel_in_ff <= 2'b0;
                        end
                end
        else
                begin
                reg_clk_ena_ff <= 0;                    // (place this outside the
conditional for better logic!)
                reg_sel_in_ff <= 2'b0;
```

```
if(n == `FCNVRTF)
	begin //global reset for all cases
		fb2fr_shft_ena_ff <= 0;
		fb2fr_load_ff <= 0;
		reg_clk_ena_ff <= 0;
		reg_wr_ff <= 0;
		reg_sel_in_ff <= 0;
		done <= 0;
		fb2fr_shft_ena_ff <= 0;
		fb2fr_acc_ena_ff <= 0;
		fb2fr_acc_ena_ff <= 0;
	//	fb2fr_reset_ff <= 0;
		fb2fr_rnd_sel_ff <= 0;
		fb2fr_done_ff  <= 0 ;

	case (cnt)
		//0:		fbin_latch_ena_ff <= 1;

	//	1:		begin
		//		fbin_latch_ena_ff <= 1;
		//		fbin_latch_sel_ff <= 1;
		//		end

		//currently no reset call for fractional (check software PIO)
		//0:	//	begin //resets accum and shift reg
				//fb2fr_acc_ena_ff <= 1;
				//fb2fr_reset_ff <= 1;
			//	end

		0:		begin		//loads bits to converter unit
				fb2fr_shft_ena_ff <= 1;
				fb2fr_load_ff <=1;
				fb2fr_acc_ena_ff <= 1;
				end

		17:	begin		//write value to register file
				reg_clk_ena_ff <= 1;
				reg_wr_ff <= 1;
				reg_sel_in_ff <= 2'b10;
				end
		18:	done <=1;
		default: begin  //loop counter; should run 8 times
					fb2fr_shft_ena_ff <= 1;
					fb2fr_acc_ena_ff <= 1;
					if (cnt > 8 )
						fb2fr_done_ff  <= 1 ;
```

```
                                        if(cnt == 16)
                                            fb2fr_rnd_sel_ff <= 1;
                                        end

                    endcase
                    //next portion is to create valid sign bit



                    end
                end
end
endmodule

//-------------------------------------------------------------------------------------*/


module Comp_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, cross_shft_init,
cross_shft_ena,decomp_subload, modcnt_clk_ena, moddiv_ena, clear_skip,
modcnt_reset, status, result);

input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
input [8:0] status; //status and compare bits

output done_flag;
output reg_clk_ena;
output cross_shft_ena;
output cross_shft_init;
output decomp_subload;
output modcnt_clk_ena;
output moddiv_ena;
output clear_skip;
output modcnt_reset;
output [31:0] result;

reg done;
reg reg_clk_ena_ff;
reg cross_shft_ena_ff;
reg cross_shft_init_ff;
reg decomp_subload_ff;
reg modcnt_clk_ena_ff;
reg moddiv_ena_ff;
reg clear_skip_ff;
```

```
reg modcnt_reset_ff;
reg [31:0] result_ff;

reg [1:0]status_val; //holds status of fmult at completion time
reg [1:0]comp_state;   //holds comparison value
reg [4:0]tmp_cnt;
reg [1:0]curr_comp; //holds current compare state
reg divclk_ena;            //moddiv and modclk enables

assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign cross_shft_ena = cross_shft_ena_ff;
assign cross_shft_init = cross_shft_init_ff;
assign decomp_subload = decomp_subload_ff;
assign modcnt_clk_ena = divclk_ena | modcnt_clk_ena_ff;
assign moddiv_ena =  divclk_ena | moddiv_ena_ff;
assign clear_skip = clear_skip_ff;
assign modcnt_reset = modcnt_reset_ff;
assign result = result_ff;

always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `COMP_A)
                        reg_clk_ena_ff <= 1;                                   // first cycle, kick off
the reg_clk_ena
                else
                        reg_clk_ena_ff <= 0;
                end
        else
                begin
                reg_clk_ena_ff <= 0;
                if(n == `COMP_A)
                        begin //global reset for all cases
                                done <= 0;
                                cross_shft_ena_ff <= 0;
                                cross_shft_init_ff <= 0;
                                decomp_subload_ff <= 0;
                                clear_skip_ff <= 0;
                                modcnt_reset_ff <= 0;
                                divclk_ena <= 0;
                                modcnt_clk_ena_ff <= 0;
```

```
                                moddiv_ena_ff <= 0;

                                //compare_state must happen 2 clocks later than when
status is update; 1 clk for current compare + 1 clk to update compstate
                                if (cnt >  3)  //compare does not happen until decomp;
ignore early values
                                        begin
                                                if (curr_comp != 1)  //compares state from
last clock
                                                        comp_state <= curr_comp;
                                                end

                        if (cnt == 0)

                                begin
                                comp_state <= 1; //equivalent of variable declaration;
initialize here so it preserves over clk cycles
                                curr_comp <= 1; //initialize it to 1 so it does not create
conflict on first check (when status == 0)
                                modcnt_reset_ff <= 1;
                                modcnt_clk_ena_ff <= 1;
                                cross_shft_init_ff <= 1;
                                result_ff <= 0;
                                end
                        else if(cnt == 1)
                                begin
                                divclk_ena <= 1;
                                clear_skip_ff <= 1;
                                end
                        else if(cnt == 2)
        // comparison value here does not matter
                                begin
                                decomp_subload_ff <= 1;
                                cross_shft_ena_ff <= 1;
                                divclk_ena <= 1;
                                tmp_cnt <= cnt;
                                end
                        else if (((status & 'h3) == 0)   || (cnt == 3))              //stay here if
status not = to 1,2,3 (make sure to always come here when cnt = 3 regardless of status)

                                begin
                                cross_shft_ena_ff <= 1;
                                divclk_ena <= 1;
                                tmp_cnt <= cnt;
                                if( cnt > 3 )
                                        curr_comp <= status[3:2];
```

```
					end
				else if (~((status & 'h3) == 0))  //status must equal 1,2 or 3
					begin

					if (cnt == (tmp_cnt + 1))
						begin
							status_val <= status[1:0];
							curr_comp <= status[3:2]; //maybe
redundant
						end
					else if (cnt == (tmp_cnt + 3))
						begin
							done <= 1;		//send done signal

							if (status_val  == 3)	//TERMINATE
TOGETHER
							begin
								if (comp_state == 1) //EQUAL A=B
status = 01xx
									result_ff <= 1; //EQUAL
DIGIT COMPARE
								else if (comp_state == 2) //
		status = 10xx
									result_ff<= 2;  //GREATER
DIGIT COMPARE
								else //if (comp_state == 0) //
		status = 10xx
									result_ff <= 0; //LESSER
DIGIT COMPARE
							end
							else if (status_val  == 2)//GREATER A>B
EARLY TERMINATE
								result_ff <= 2;
							else if(status_val  == 1) //LESSER EARLY
TERMINATE
								result_ff <= 0;
						end
					end
		end
	end
end

endmodule
//----------------------------------------------------------------------------------------------------
-----------*/
```

```
//-------------------------------------------------------------------------------*/

//Normalize instruction (fmult without the mult)
module Norm_accumA(aclk, reset, n, cnt, done_flag, reg_clk_ena, acc_clk_ena,
mult_clk_ena, fmult_acc_clk_ena, cross_shft_init, cross_shft_ena,
decomp_subload, modmult_ena, modmult_acc_ena, modmult_acc_clr,  modcnt_clk_ena,
moddiv_ena, clear_skip, modcnt_reset,
start_latch, macBcomp_ena, mac_select, fmult_status, acc_clear);



input aclk;
input reset;
input [7:0] n;
input [4:0] cnt;
input [8:0] fmult_status; //status and compare bits

output done_flag;
output reg_clk_ena;
output acc_clk_ena;
output mult_clk_ena;  //needed to perform multiply before decomp
output fmult_acc_clk_ena;  //separate acc clk for fmult

output cross_shft_ena;
output cross_shft_init;
output decomp_subload;
output modmult_ena;
output modmult_acc_ena;
output modmult_acc_clr;
output modcnt_clk_ena;
output moddiv_ena;
output clear_skip;
output modcnt_reset;
output start_latch;
output macBcomp_ena;
output mac_select;
output acc_clear;

reg done;
reg reg_clk_ena_ff;
reg acc_clk_ena_ff;
reg fmult_acc_clk_ena_ff;
reg cross_shft_ena_ff;
reg cross_shft_init_ff;
reg decomp_subload_ff;
```

```
reg modmult_ena_ff;
reg modmult_acc_ena_ff;
reg modmult_acc_clr_ff;
reg modcnt_clk_ena_ff;
reg moddiv_ena_ff;
reg clear_skip_ff;
reg modcnt_reset_ff;
reg start_latch_ff;
reg macBcomp_ena_ff;
reg mac_select_ff;
reg mult_clk_ena_ff;
reg [1:0]fmult_status_val; //holds status of fmult at completion time
reg [1:0]comp_state;   //holds comparison value
reg [4:0]tmp_cnt;
reg divclk_ena;           //moddiv and modclk enables
reg acc_clear_ff;

assign acc_clear = acc_clear_ff;
assign done_flag = done;
assign reg_clk_ena = reg_clk_ena_ff;
assign fmult_acc_clk_ena = fmult_acc_clk_ena_ff;
assign acc_clk_ena = acc_clk_ena_ff;
assign cross_shft_ena = cross_shft_ena_ff;
assign cross_shft_init = cross_shft_init_ff;
assign decomp_subload = decomp_subload_ff;
assign modmult_ena = modmult_ena_ff;
assign modmult_acc_ena = modmult_acc_ena_ff;
assign modmult_acc_clr = modmult_acc_clr_ff;
assign modcnt_clk_ena = divclk_ena | modcnt_clk_ena_ff;
assign moddiv_ena =  divclk_ena | moddiv_ena_ff;
assign clear_skip = clear_skip_ff;
assign modcnt_reset = modcnt_reset_ff;
assign start_latch = start_latch_ff;
assign macBcomp_ena = macBcomp_ena_ff;
assign mac_select = mac_select_ff;
assign mult_clk_ena = mult_clk_ena_ff;


always @(posedge aclk or posedge reset)
begin

        if(reset)
                begin
                done <= 0;

                if(n == `NORM_A)
```

```
				begin
				reg_clk_ena_ff <= 1;				// first cycle, kick off
the reg_clk_ena
				acc_clear_ff <= 0;				//make sure accum is
not cleared
				end
			else
				reg_clk_ena_ff <= 0;

			mult_clk_ena_ff <= 0;
			acc_clk_ena_ff <= 0;
			fmult_acc_clk_ena_ff <= 0;
			acc_clear_ff <= 0;				//make sure accum is not
cleared
			end
		else
			begin
			reg_clk_ena_ff <= 0;				// (place this outside the
conditional for better logic!)

			if(n == `NORM_A)
				begin //global reset for all cases
					done <= 0;
					mult_clk_ena_ff <= 0;
					acc_clk_ena_ff <= 0;
					cross_shft_ena_ff <= 0;
					cross_shft_init_ff <= 0;
					decomp_subload_ff <= 0;
					modmult_ena_ff <= 0;
					modmult_acc_ena_ff <= 0;
					modmult_acc_clr_ff <= 0;
					clear_skip_ff <= 0;
					modcnt_reset_ff <= 0;
					start_latch_ff <= 0;
					divclk_ena <= 0;
					modcnt_clk_ena_ff <= 0;
					moddiv_ena_ff <= 0;

				if (cnt == 0)				//perform
multiplication first
				//	mult_clk_ena_ff <= 1;
			//	else if(cnt == 1)
				//	acc_clk_ena_ff <= 1;
				//else if(cnt == 1)
		//mult done now do decomp
					cross_shft_init_ff <= 1;
```

```
			else if(cnt == 1)
	// assert the acc clk ena here
				begin
				modcnt_clk_ena_ff <= 1;
				modcnt_reset_ff <= 1;
				start_latch_ff <= 1;
				end
			else if(cnt == 2)
	// assert the acc clk ena here
				begin
				divclk_ena <= 1;
				clear_skip_ff <= 1;
				end
			else if(cnt == 3)
	// first iteration of load; from AccumA
				begin
				cross_shft_ena_ff <= 1;
				decomp_subload_ff <= 1;
				divclk_ena <= 1;
				end
			else if((cnt >= 4) & (cnt <= 9))
		//six times
				begin
				cross_shft_ena_ff <= 1;
				divclk_ena <= 1;
				end
			else if(cnt == 10)
	// i = 7
				begin
				cross_shft_ena_ff <= 1;
				modmult_acc_clr_ff <= 1;
				modmult_ena_ff <= 1;
				divclk_ena <= 1;
				end
			else if(cnt == 11)
	// i = 8
				begin
				cross_shft_ena_ff <= 1;
				modmult_ena_ff <= 1;
				divclk_ena <= 1;
				end
			else if ((fmult_status & 'h3) == 0)			//decomp rns
accumulate
				begin
				cross_shft_ena_ff <= 1;
				modmult_ena_ff <= 1;
```

```
                    modmult_acc_ena_ff <= 1;
                    divclk_ena <= 1;
                    tmp_cnt <= cnt;
                    end
                else if (~((fmult_status & 'h3) == 0))
        //status must be 1,2,3,
                    begin
                    if (cnt == (tmp_cnt + 1))
                        begin
                            fmult_acc_clk_ena_ff <= 0;
                            modmult_acc_ena_ff <= 1;
                            macBcomp_ena_ff <= 0;
                            mac_select_ff <= 0;
                            fmult_status_val <= fmult_status[1:0];
//saves value of fmult status since it will change during next clk
                            comp_state <= fmult_status[5:4];
                        end
                    else if (cnt == (tmp_cnt + 2))   //start checking compare
states of accumA and B
                        begin
                            if (fmult_status_val == 1)
                                fmult_acc_clk_ena_ff <= 1;
                            else if (fmult_status_val == 2)
                            begin
                                macBcomp_ena_ff <= 1;
                                mac_select_ff <= 1;  //make sure so
hold until accum is latched
                            end
                            else if(fmult_status_val == 3)
                            begin
                                if (comp_state == 2) //A>B take
compliment
                                begin
                                    macBcomp_ena_ff <= 1;
                                    mac_select_ff <= 1;
                                end
                             else if (comp_state == 1) //cmp_cnt < 1
                                    acc_clear_ff <= 1;  //clear the
accum since it is empty
                                else                //A=B  accum values
are both 0 (empty) if not greater than it will be equal or less than; in both cases we latch
the final value of accum
                                    fmult_acc_clk_ena_ff <= 1;
                            end
                        end
                    else if((cnt >= (tmp_cnt + 3)))
```

```
					begin
						if ((fmult_acc_clk_ena == 1)  || (acc_clear
== 1))
						begin
							done <= 1;
							fmult_acc_clk_ena_ff <= 0;
							mac_select_ff <= 0; //ensures select
was held long enough
							acc_clear_ff <= 0;
							//macBcomp_ena_ff <= 0;
						end
					else //the only other reason is for the compliment
						begin
							fmult_acc_clk_ena_ff <= 1;
							macBcomp_ena_ff <= 0;
						end
					end
				end
			end
		end
end
endmodule

//--------------------------------------------------------------------------------*/
```

APPENDIX B: ALU_INSTRUCT.C

```
/*
 * alu_instruct.c
 *
 *  Created on: Jun 29, 2013
 *      Author: Eric
 */


#include <stdio.h>
#include <system.h>
#include <io.h>
#include <math.h>
#include "alu.h"
#include "utilities.h"
#include "alu_prims.h"
#include "alu_instruct.h"




// this routine reads the regA file at (reg_page + regadr)
// any circuit receiving the regA file may receive the referenced value
// NOTE:  clears all CONTROL_A lines
void read_regA(uint regf_page, uint regadr)
{

   IOWR(ADDRESS_A_BASE, 0, regf_page + regadr);          // set up the register file
address
   IOWR(CONTROL_A_BASE, 0x0, REG_CLK_ENA);
   clock_pulse();                              // pulse the clock
   IOWR(CONTROL_A_BASE, 0x0, 0);             // clear the register clock enable

}


// this routine reads the regB file at (reg_page + regadr)
// any circuit receiving the regA file may receive the referenced value
// NOTE:  clears all CONTROL_B lines
void read_regB(uint regf_page, uint regadr)
{

   IOWR(ADDRESS_B_BASE, 0, regf_page + regadr);          // set up the register file
address
   IOWR(CONTROL_B_BASE, 0x0, REG_CLK_ENA);
```

```
   clock_pulse();                          // pulse the clock
   IOWR(CONTROL_B_BASE, 0x0, 0);           // clear the register clock enable

}


// this routine writes the input to the regA file at (reg_page + regadr)
// source of the write is selected via the input_srce value, see alu.h
// NOTE:  clears all CONTROL_A lines
void write_regA(uint regf_page, uint regadr, uint input_srce)
{

   IOWR(ADDRESS_A_BASE, 0, regf_page + regadr);          // set up the register file
address
   IOWR(CONTROL_SELECT_BASE, 0x0, ((input_srce & REGA_SEL_IN_MASK)
<< REGA_SEL_IN_OFFSET));

   IOWR(CONTROL_A_BASE, 0x0, REG_CLK_ENA | REG_WR);
   clock_pulse();                          // pulse the clock
   IOWR(CONTROL_A_BASE, 0x0, 0);           // clear the register clock enable
   IOWR(CONTROL_SELECT_BASE, 0x0, 0);

}


// this routine writes the input to the regB file at (reg_page + regadr)
// source of the write is selected via the input_srce value, see alu.h
// NOTE:  clears all CONTROL_B lines
void write_regB(uint regf_page, uint regadr, uint input_srce)
{

   IOWR(ADDRESS_B_BASE, 0, regf_page + regadr);          // set up the register file
address
   IOWR(CONTROL_SELECT_BASE, 0x0, ((input_srce & REGB_SEL_IN_MASK)
<< REGB_SEL_IN_OFFSET));

   IOWR(CONTROL_B_BASE, 0x0, REG_CLK_ENA | REG_WR);
   clock_pulse();                          // pulse the clock
   IOWR(CONTROL_B_BASE, 0x0, 0);           // clear the register clock enable
   IOWR(CONTROL_SELECT_BASE, 0x0, 0);

}

void clear_accumA(uint control)
{
```

```
    control |= ACC_CLEAR;

    IOWR(CONTROL_A_BASE, 0x0, control);
    clock_pulse();

    control &= ~ACC_CLEAR;
    IOWR(CONTROL_A_BASE, 0x0, control);

}

void clear_accumB(uint control)
{

    control |= ACC_CLEAR;

    IOWR(CONTROL_B_BASE, 0x0, control);
    clock_pulse();

    control &= ~ACC_CLEAR;
    IOWR(CONTROL_B_BASE, 0x0, control);

}

// load the accumulator A with the regF value located at regadr
void load_accumA(uint regf_page, uint regadr)
{
uint control = 0;          // shadow the control_A register


    read_regA(regf_page, regadr);          // read the regA file at reg_page+regadr

    control = select_accA_input(REGF_TO_ACC, 0);        // set control bits to select the
register file input
    control |= ACC_CLK_ENA;

    IOWR(CONTROL_A_BASE, 0x0, control);

    clock_pulse();
    IOWR(CONTROL_A_BASE, 0x0, 0);

}

// load the accumulator B with the regF value located at regadr
void load_accumB(uint regf_page, uint regadr)
{
uint control = 0;          // shadow the control_A register
```

```
   read_regB(regf_page, regadr);          // read the regA file at reg_page+regadr

   control = select_accB_input(REGF_TO_ACC, 0);       // set control bits to select the
register file input
   control |= ACC_CLK_ENA;

   IOWR(CONTROL_B_BASE, 0x0, control);

   clock_pulse();
   IOWR(CONTROL_B_BASE, 0x0, 0);

}

// store the accumulator A to the register file
void store_accumA(uint regf_page, uint regadr)
{
uint control;

   read_regA(regf_page, regadr);          // read the regA file at reg_page+regadr

   control = REG_CLK_ENA | REG_WR;
   IOWR(CONTROL_A_BASE, 0x0, control);

   clock_pulse();
   IOWR(CONTROL_A_BASE, 0x0, 0);

}

// store the accumulator B to the register file
void store_accumB(uint regf_page, uint regadr)
{
uint control;

   read_regB(regf_page, regadr);          // read the regA file at reg_page+regadr

   control = REG_CLK_ENA | REG_WR;
   IOWR(CONTROL_B_BASE, 0x0, control);

   clock_pulse();
   IOWR(CONTROL_B_BASE, 0x0, 0);

}
```

```
// add the register file value at regadr to the accumulator A
void add_accumA(uint regf_page, uint regadr)
{
uint control = 0;

   read_regA(regf_page, regadr);          // read the register at reg_page+regadr
   control |= ADD;                        // no bits set for add operation
   latch_accumA(ADDSUB_TO_ACC, control);             // latch the ADD_SUB unit

}

// add the register file value at regadr to the accumulator B
void add_accumB(uint regf_page, uint regadr)
{
uint control = 0;

   read_regB(regf_page, regadr);          // read the register at reg_page+regadr
   control |= ADD;                        // no bits set for add operation
   latch_accumB(ADDSUB_TO_ACC, control);             // latch the ADD_SUB unit

}


// subtract the register file value at regadr from the accumulator A
void sub_accumA(uint regf_page, uint regadr)
{
uint control = 0;

   read_regA(regf_page, regadr);              // read the register at reg_page+regadr
   control |= SUBTRACT;                       // control bit set for subtract operation
   latch_accumA(ADDSUB_TO_ACC, control);      // latch the ADD_SUB unit

}

// subtract the register file value at regadr from the accumulator B
void sub_accumB(uint regf_page, uint regadr)
{
uint control = 0;

   read_regB(regf_page, regadr);              // read the register at reg_page+regadr
   control |= SUBTRACT;                       // control bit set for subtract operation
   latch_accumB(ADDSUB_TO_ACC, control);      // latch the ADD_SUB unit

}

// routine to complement the accumulator B
```

```
// destroys control_B
void comp_accumB(uint control)
{

   latch_accumB(COMP_TO_ACC_B, control);

}



void mult_accumA(uint regf_page, uint regadr)
{
uint control = 0;

   read_regA(regf_page, regadr);

   control |= MULT_CLK_ENA;
   IOWR(CONTROL_A_BASE, 0x0, control);
   clock_pulse();

   control = 0;
   latch_accumA(MULT_TO_ACC, control);

}

void mult_accumB(uint regf_page, uint regadr)
{
uint control = 0;

   read_regB(regf_page, regadr);

   control |= MULT_CLK_ENA;
   IOWR(CONTROL_B_BASE, 0x0, control);
   clock_pulse();

   control = 0;
   latch_accumB(MULT_TO_ACC, control);

}



// ******************************  Fractional Multiplier Control
**********************
```

```
// skip digit control
void clear_skips(void)
{
uint control;

   control = CLEAR_SKIPS;
   IOWR(CONTROL_A_BASE, 0x0, control);

   clock_pulse();
   IOWR(CONTROL_A_BASE, 0x0, 0);


}

//  initialize the cross control
void init_cross_shft(ulong control)
{

   control |= CROSS_SHFT_INIT;
   IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_INIT);
   clock_pulse();

//  printf("control_A = %lx\n", read_test(CROSS_DATA, 0));
   control &= ~CROSS_SHFT_INIT;
   IOWR(CONTROL_A2_BASE, 0x0, control);

//  printf("cross shift control: %d, %d\n", read_test(CROSS_SHFT_CNTRL, 0),
read_test(CROSS_DATA, 0));


}

// shift the cross control once
void shift_cross_cntrl(ulong control)
{

   control |= CROSS_SHFT_ENA;
   IOWR(CONTROL_A2_BASE, 0x0, control);
   clock_pulse();

   control &= ~CROSS_SHFT_ENA;
   IOWR(CONTROL_A2_BASE, 0x0, control);

//  printf("cross shift control: %d, %d\n", read_test(CROSS_SHFT_CNTRL, 0),
read_test(CROSS_DATA, 0));
```

```
}

// clears the modcnt A counter, for testing purposes
void clear_modcntA(uint control)
{

   control |= (MODCNT_RESET | MODCNT_CLK_ENA);        // set the control signals
   IOWR(CONTROL_A_BASE, 0x0, control);
   clock_pulse();

   control &= ~(MODCNT_RESET | MODCNT_CLK_ENA);        // clear the control
signals
   IOWR(CONTROL_A_BASE, 0x0, control);

}

// increment the modcntA counter, for testing only
void inc_modcntA(uint control)
{

//  printf("entering modcnt inc\n");
   control |= (MODCNT_CLK_ENA);           // set the control signals
   IOWR(CONTROL_A_BASE, 0x0, control);
   clock_pulse();

   control &= ~(MODCNT_CLK_ENA);       // clear the control signals
   IOWR(CONTROL_A_BASE, 0x0, control);

}

// test utility print program
void print_compare(uint comp)
{

//  printf("compare: %x\n", comp);

   comp &= 0x3;        // mask just two bits for now

   if(comp == GREATER)  {

      printf("A > B\n");
   }
   else if(comp == EQUAL)  {

      printf("A = B\n");
   }
```

```
   else {

      printf("A < B\n");
   }

}


// routine returns the next comparison status
uint compare_digits(uint state, uint cur_comp)
{

   if(cur_comp == EQUAL) {
      return(state);
   }
   else {
      return(cur_comp);
   }

}

// continue  testing of the fractional multiplier with immediate recomposition
// working with both A and B accumulators
void fmult_accumA(into reg_page, into regard)
{
into i;
uint control, control2;
uint comp_state = EQUAL;                // start with digits being equal
uint cur_comp, status;

//  printf("FMULT TEST\n");

//  printf("accumulator A at start:\n");
//  print_test_as_reg(ACC_A_CHANNEL);
//  printf("\n");
//  store_unit(0);              // store the unit value into register 0
//  write_bin_2_regf(1, 509);   // store integer value into register 1
//  write_bin_2_regf(1, 2); // store integer value into register 1
//  write_bin_2_regf(2, 0);     // don't need this anymore, since we have clear accum!

//  load_accumA(0, 0);          // load the uit value into accumulator A
   mult_accumA(reg_page, regard);

//  mult_accumA(0, 0);          // square the accumulator
//  mult_accumA(0, 1);
```

```
    store_accumA(0, 20);            // store the accumulator in register 20

//  clear_accumA(0);
//  sub_accumA(0, 3);           // negate the accumA register

    clear_accumB(0);
    sub_accumB(0, 20);          // make the complement of the A value and store in AccB
//  load_accumB(0, 3);          // for testing

//  load_accumB(0, 3);          // store the non-negated version to accumB
//  comp_accumB(0);             // complement the accumB

    printf("accumulator A:\n");
    print_test_as_reg(ACC_A_CHANNEL);
    printf("\n");

    printf("accumulator B:\n");
    print_test_as_reg(ACC_B_CHANNEL);
    printf("\n");

//  store_accumA(0, 2);         // need this routine!


// now perform the decomposition

     init_cross_shft(0);

//   clear_modcntA(0);
    control = (MODCNT_RESET | MODCNT_CLK_ENA | FMULT_START_LTCH);
// set the control signals
    IOWR(CONTROL_A_BASE, 0x0, control);
    clock_pulse();

     control = (MODDIV_CLK_ENA | CLEAR_SKIPS | MODCNT_CLK_ENA);
     IOWR(CONTROL_A_BASE, 0x0, control);
     clock_pulse();
     IOWR(CONTROL_A_BASE, 0x0, 0);
     control = 0;
     control2 = 0;

     i = 0;
//   while(!(read_test(ALU_A_STATUS, 0) & 0x4)) {
//   while(i < NUM_DIGITS) {

     do {
         if(i == 0) {
```

```
            IOWR(CONTROL_A2_BASE, 0x0, DECOMP_SUB_LOAD |
CROSS_SHFT_ENA);        // first iteration is a load from ACC
        }
        else if(i < 7) {                                    // next iterations is decomp only
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA);
        }
        else if(i == 7) {                // set up decomp plus RNS power accumulate
            IOWR(DATA_BASE, 0x0, 0);      // set up the modmult LUT to access first
power (one)
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA | MODMULT_ENA
| MODMULT_ACC_CLR);
        }
        else if(i == 8) {
            IOWR(DATA_BASE, 0x0, i-7);
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA |
MODMULT_ENA);
        }
        else {                          // continue decomp plus RNS accumulate
            IOWR(DATA_BASE, 0x0, i-7);
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA | MODMULT_ENA
| MODMULT_ACC_ENA);
        }

        printf("lut_data= %x, moddiv_cnt = %x, status = %x\n", read_test(LUT_DATA,
0), read_test(DIV_MODCNT_A, 0), read_test(ALU_A_STATUS, 0));
        printf("crossbar_A = %x\n", read_test(CROSS_DATA_A, 0));
        printf("crossbar_B = %x\n", read_test(CROSS_DATA_B, 0));

        control = MODDIV_CLK_ENA | MODCNT_CLK_ENA;
        IOWR(CONTROL_A_BASE, 0x0, control);
        clock_pulse();

        IOWR(CONTROL_A_BASE, 0x0, 0);
        IOWR(CONTROL_A2_BASE, 0x0, 0);

        printf("decompA: ");
        print_test_as_reg_skip(MODDIV_A_CHANNEL);
        printf("modmultA:");
        print_test_as_reg(MODMULT_A_DATA);
        printf("accA    :");
        print_test_as_reg(MAC_A_DATA);
        printf("\n");

        printf("decompB: ");
        print_test_as_reg_skip(MODDIV_B_CHANNEL);
        printf("modmultB:");
```

```
        print_test_as_reg(MODMULT_B_DATA);
        printf("accB    :");
        print_test_as_reg(MAC_B_DATA);

        cur_comp = ((read_test(ALU_A_STATUS, 0) & 0xc0) >> 6);          // read the
current compare
        print_compare(cur_comp);
        comp_state = compare_digits(comp_state, cur_comp);

        wait_key();

        i++;

        status = read_test(ALU_A_STATUS, 0);

    } while(!(status & 0xc));

   // must clock the accumulator one last time
    IOWR(CONTROL_A2_BASE, 0x0, MODMULT_ACC_ENA);
    clock_pulse();
    IOWR(CONTROL_A2_BASE, 0x0, 0);

//    status = read_test(ALU_A_STATUS, 0);
    printf("i= %d, status = %x\n", i, status);
    printf("the final compare is: ");
    print_compare(comp_state);

    printf("final MacA:");
    print_test_as_reg(MAC_A_DATA);
    printf("final MacB:");
    print_test_as_reg(MAC_B_DATA);

    status &= 0xc;
    if(status == 0xc) {           // both decomposers terminate together, go to comparison
status

        if(comp_state == EQUAL) {     // result must be zero
           clear_accumA(0);
        }
        else if(comp_state == GREATER)  {

           printf("latching MacB via A>B\n");

           IOWR(CONTROL_A2_BASE, 0x0, MACB_COMP_ENA | MAC_SELECT);
           clock_pulse();
           latch_accumA(FMULT_TO_ACC_A, 0);
```

```
            IOWR(CONTROL_A2_BASE, 0x0, 0);

        }
        else {        // comp_state = Lesser
            printf("latching MacA via A<B\n");

            latch_accumA(FMULT_TO_ACC_A, 0);
        }

    }
    else if(status == 0x4) {  // MacA terminates first, answer is positive, gate MacA to
AccA
        printf("latching MacA via MacA=0\n");

        latch_accumA(FMULT_TO_ACC_A, 0);
    }
    else if(status == 0x8)  { // MACB terminates first, answer is negative, gate the
complement of MacB to AccA

        printf("latching MacB via MacB=0\n");

        IOWR(CONTROL_A2_BASE, 0x0, MACB_COMP_ENA | MAC_SELECT);
        clock_pulse();
        latch_accumA(FMULT_TO_ACC_A, 0);

        IOWR(CONTROL_A2_BASE, 0x0, 0);
    }

    printf("final: accumulator A:\n");
    print_test_as_reg(ACC_A_CHANNEL);

}

// continue  testing of the fractional multiplier with immediate recomposition
// working with both A and B accumulators
// working with module
void fmult_accumA2(into reg_page, into regard)
{
into i;
uint control, control2;
uint comp_state = EQUAL;               // start with digits being equal
uint cur_comp, status;
long double dval2;          //d.a. check rns value as it is being calculated

//  printf("FMULT TEST\n");
```

```
   printf("accumulator A at start:\n");
   print_test_as_reg(ACC_A_CHANNEL);
//  printf("\n");
//  store_unit(0);              // store the unit value into register 0
//  write_bin_2_regf(1, 509);   // store integer value into register 1
//  write_bin_2_regf(1, 2); // store integer value into register 1
//  write_bin_2_regf(2, 0);     // don't need this anymore, since we have clear accum!

//  load_accumA(0, 0);          // load the uit value into accumulator A
   mult_accumA(reg_page, regard);

   printf("accumulator A during fmult after into mult:\n");  //d.a test value in accA during
fmult process
   print_test_as_reg10(ACC_A_CHANNEL);
   wait_key();

//  mult_accumA(0, 0);          // square the accumulator
//  mult_accumA(0, 1);

//  store_accumA(0, 20);            // store the accumulator in register 20

//  clear_accumA(0);
//  sub_accumA(0, 3);           // negate the accumA register

//  clear_accumB(0);
//  sub_accumB(0, 20);          // TEST - TAKE OUT make the complement of the A value
and store in AccB

//  load_accumB(0, 3);          // for testing
//  load_accumB(0, 3);          // store the non-negated version to accumB
//  comp_accumB(0);             // complement the accumB

//  printf("accumulator A after into mult:\n");
//  print_test_as_reg(ACC_A_CHANNEL);
//  printf("\n");

//  printf("accumulator B:\n");
//  print_test_as_reg(ACC_B_CHANNEL);
//  printf("\n");

//  store_accumA(0, 2);         // need this routine!

   ////////////////////////////////////////////////////
/*  //normalize test segment
   STORE_A(4);         //d.a. test for normalization
```

```
   printf("VALUE before normalize \n");
   print_test_as_reg10(ACC_A_CHANNEL);  //d.a. test for normalization/
   wait_key();  //d.a. test for normalization

   NORM_A(4);          //d.a. test for normalization
   printf("VALUE AFTER normalize \n");
   print_test_as_reg10(ACC_A_CHANNEL);  //d.a. test for normalization/
   wait_key();  //d.a. test for normalization
//////////////////////////////////////////////////////////*/

// now perform the decomposition

    init_cross_shft(0);

//   clear_modcntA(0);
   control = (MODCNT_RESET | MODCNT_CLK_ENA | FMULT_START_LTCH);
// set the control signals
   IOWR(CONTROL_A_BASE, 0x0, control);
   clock_pulse();

    control = (MODDIV_CLK_ENA | CLEAR_SKIPS | MODCNT_CLK_ENA);
    IOWR(CONTROL_A_BASE, 0x0, control);
    clock_pulse();
    IOWR(CONTROL_A_BASE, 0x0, 0);
    control = 0;
    control2 = 0;

    i = 0;
//   while(!(read_test(ALU_A_STATUS, 0) & 0x4)) {
//   while(i < NUM_DIGITS) {

     printf("modmultA at start:");
     print_test_as_reg(FMULTA2_MODMULTA);

    do {
        if(i == 0) {
           IOWR(CONTROL_A2_BASE, 0x0, DECOMP_SUB_LOAD |
CROSS_SHFT_ENA);        // first iteration is a load from ACC
        }
        else if(i < 7) {                               // next iterations is decomp only
           IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA);
        }
        else if(i == 7) {               // set up decomp plus RNS power accumulate
         //  IOWR(DATA_BASE, 0x0, 0);       // set up the modmult LUT to access first
power (one)
```

```
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA | MODMULT_ENA
| MODMULT_ACC_CLR);
        }
        else if(i == 8) {
          //  IOWR(DATA_BASE, 0x0, i-7);
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA |
MODMULT_ENA);
        }
        else {                          // continue decomp plus RNS accumulate
          //  IOWR(DATA_BASE, 0x0, i-7);
            IOWR(CONTROL_A2_BASE, 0x0, CROSS_SHFT_ENA | MODMULT_ENA
| MODMULT_ACC_ENA);
        }

        printf("status = %x\n", read_test(FMULTA_STATUS, 0));
        printf("crossbar_A = %x\n", read_test(FMULTA2_TEST, 0));
        printf("crossbar_B = %x\n", read_test(FMULTA2_TEST, 1));

        control = MODDIV_CLK_ENA | MODCNT_CLK_ENA;
        IOWR(CONTROL_A_BASE, 0x0, control);
        clock_pulse();

        IOWR(CONTROL_A_BASE, 0x0, 0);
        IOWR(CONTROL_A2_BASE, 0x0, 0);

        printf("decompA: ");
        print_test_as_reg_skip(FMULTA2_MODDIVA);
        printf("modmultA:");
        print_test_as_reg(FMULTA2_MODMULTA);
        printf("accA    :");
        print_test_as_reg(FMULTA2_MODMACA);
        printf("\n");

        printf("decompB: ");
        print_test_as_reg_skip(FMULTA2_MODDIVB);
        printf("modmultB:");
        print_test_as_reg(FMULTA2_MODMULTB);
        printf("accB    :");
        print_test_as_reg(FMULTA2_MODMACB);

        status = read_test(FMULTA_STATUS, 0);        // adjust data like old read status
port
        printf("status after clock = %x\n", status);

//      cur_comp = ((read_test(ALU_A_STATUS, 0) & 0xc0) >> 6);          // read the
current compare
```

```
        cur_comp = ((status & 0x30) >> 4);          // read the current compare
        print_compare(cur_comp);
        comp_state = compare_digits(comp_state, cur_comp);

        printf("current value of i is :%x\n", i);

     //     dval2 = convert_outA_ldouble();
       //  printf("convert out : %3.15Lf\n", dval2);  //print out current converted rns value


        i++;
        wait_key();

//       status = read_test(ALU_A_STATUS, 0);
//       status = read_test(FMULTA_STATUS, 0) << 2;          // adjust data like old read
status port

     } while(!(status & 0x3));

    // must clock the accumulator one last time
     IOWR(CONTROL_A2_BASE, 0x0, MODMULT_ACC_ENA);
     clock_pulse();
     IOWR(CONTROL_A2_BASE, 0x0, 0);

//    status = read_test(ALU_A_STATUS, 0);
//    printf("i= %d, status = %x\n", i, status);
//    printf("the final compare is: ");
//    print_compare(comp_state);

//    printf("final MacA:");
//    print_test_as_reg(FMULTA2_MODMACA);
//    printf("final MacB:");
//    print_test_as_reg(FMULTA2_MODMACB);

     status &= 0x3;
     if(status == 0x3) {           // both decomposers terminate together, go to comparison
status

        if(comp_state == EQUAL) {     // result must be zero
           clear_accumA(0);
        }
        else if(comp_state == GREATER)  {

//           printf("latching MacB via A>B\n");

           IOWR(CONTROL_A2_BASE, 0x0, MACB_COMP_ENA | MAC_SELECT);
```

```
            clock_pulse();
            latch_accumA(FMULT_TO_ACC_A, 0);

            IOWR(CONTROL_A2_BASE, 0x0, 0);

         }
         else {        // comp_state = Lesser
//          printf("latching MacA via A<B\n");

            latch_accumA(FMULT_TO_ACC_A, 0);
         }

      }
      else if(status == 0x1) {  // MacA terminates first, answer is positive, gate MacA to
AccA
//       printf("latching MacA via MacA=0\n");

         latch_accumA(FMULT_TO_ACC_A, 0);
         //dval2 = convert_outA_ldouble();
        // printf("convert out : %3.15Lf\n", dval2);  //print out current converted rns value

      }
      else if(status == 0x2)  { // MACB terminates first, answer is negative, gate the
complement of MacB to AccA

//       printf("latching MacB via MacB=0\n");

         IOWR(CONTROL_A2_BASE, 0x0, MACB_COMP_ENA | MAC_SELECT);
         clock_pulse();
         //IOWR(CONTROL_A2_BASE, 0x0, MAC_SELECT);
         latch_accumA(FMULT_TO_ACC_A, 0);

         IOWR(CONTROL_A2_BASE, 0x0, 0);
      }
     /* printf("accA    :");
      print_test_as_reg(FMULTA2_MODMACA);
      printf("\n");

      printf("accB    :");
      print_test_as_reg(FMULTA2_MODMACB);
      printf("\n");

      printf("final: accumulator A:\n");
      print_test_as_reg(ACC_A_CHANNEL);//*/

}
```

```
// ******************************  Compare Unit Routines
************************************

#define COMPARE_A_BASE  CNTRL_PIPE_A_BASE          // borrow the PIPE
control lines for now

// BASIC MAGNITUDE COMPARISON
// Compares the accumulator A magnitude with the register magnitude
// returns basic comparison status
into comp_accumA(into reg_page, into regard)
{
into i;
uint control;
uint comp_state = EQUAL;              // start with digits being equal
uint cur_comp, status;
long double dval2;         //d.a. check rns value as it is being calculated



//  NEED TO IMPLEMENT STATUS REGISTER RESULTS
//  NEED TO IMPLEMENT sign check COMPARISON
//  NEED TO CHECK FOR EQUALITY
//  and zero/non-zero check comparison
//  and error checking, and secondary comparison
//  AND ACCUMULATOR SIGN EXTENSION


//  printf("accumulator A before comp:\n");  //d.a test value in accA during fmult process
//  print_test_as_reg10(ACC_A_CHANNEL);

   read_regA(reg_page, regard);       // set-up the RNS value to be compared against
//  printf("register value:\n");
//  print_test_as_reg10(REGF_A_CHANNEL);

//  wait_key();


//  The following can be reduced to a single clock with right logic, AND
//  SHOULD BE PROCESSED IN ADVANCE TO REDUCE THE CLOCKS TO
EFFECTIVE ZERO
   //  printf("status = %x\n", read_test(CMP_OUT_STATUS, 0));      // DIRECT
STATUS PORT READ

   control = (CMP_MODCNT_RST | CMP_MODCNT_ENA | CMP_SHFT_INIT);
// clear the modcnt counter
```

```
   IOWR(COMPARE_A_BASE, 0x0, control);
   clock_pulse();
//    printf("status = %x\n", read_test(CMP_OUT_STATUS, 0));        // DIRECT
STATUS PORT READ

   //  cur_comp = ((status & 0x0c) >> 2);             // read the current compare
   //  printf("compare state at 1st clock = %3d\n", cur_comp); //compare state actual value
//d.a.



   control = (CMP_MODDIV_ENA | CMP_CLEAR_SKIP | CMP_MODCNT_ENA);
// first moddiv clock  (seed moddiv?), clear skip FF, load adr=0 to div_base_mlab
   IOWR(COMPARE_A_BASE, 0x0, control);
   clock_pulse();
   IOWR(COMPARE_A_BASE, 0x0, 0);
   control = 0;

   i = 0;

   //  cur_comp = ((status & 0x0c) >> 2);             // read the current compare
   //  printf("compare state at 2nd clock = %3d\n", cur_comp); //compare state actual
value

     do {

        control = 0;
         if(i == 0) {
             control = CMP_DECOMP_LOAD | CMP_SHFT_ENA;
         }
         else {    // next iterations is decomp only
             control = CMP_SHFT_ENA;
         }


         printf("status = %x\n", read_test(CMP_OUT_STATUS, 0));        // DIRECT
STATUS PORT READ
         printf("crossbar_A = %x\n", read_test(CMP_OUT_TEST,
CMP_CROSS_A_TEST));
         printf("crossbar_B = %x\n", read_test(CMP_OUT_TEST,
CMP_CROSS_B_TEST));

         control |= (CMP_MODDIV_ENA | CMP_MODCNT_ENA);         // always doing
these
         IOWR(COMPARE_A_BASE, 0x0, control);
         clock_pulse();
```

```
        IOWR(COMPARE_A_BASE, 0x0, 0);


        printf("decompA: ");
        print_test_as_reg_skip(CMP_MODDIVA);

        printf("decompB: ");
        print_test_as_reg_skip(CMP_MODDIVB);
   //   printf("\n");

        status = read_test(CMP_OUT_STATUS, 0);          // adjust data like old read
status port
   //       printf("status after clock = %x\n", status);

        cur_comp = ((status & 0x0c) >> 2);         // read the current compare
      //  printf("digit compare: ");
      //  print_compare(cur_comp);
       // printf("compare state before current comparison = %3d\n", comp_state);
//compare state actual value
        comp_state = compare_digits(comp_state, cur_comp);
      //  printf("compare state: ");
   //   print_compare(comp_state);

      //  printf("loop cnt i is :%x\n", i);

       // printf("current compare = %3d\n", cur_comp);//check value of current compare
//D.A
      //  printf("compare state after current comparison = %3d\n", comp_state); //compare
state actual value //D.A.
   //   printf("\n"); //compare state actual value //D.A.

        i++;
        wait_key();


    } while(!(status & 0x3));


    status &= 0x3;
    if(status == 0x3) {          // both decomposers terminate together, go to comparison
status

        //printf("comparison values terminate together\n");
        if(comp_state == EQUAL) {     // result must be zero
//          printf("EQUAL: all digits the same\n");
```

```
            return(EQUAL);
         }
         else if(comp_state == GREATER)  {
   //        printf("GREATER: by digit comparison\n");
            return(GREATER);
         }
         else {        // comp_state = Lesser
      //    printf("LESSER: by digit comparison\n");
            return(LESSER);
         }

      }
      else if(status == 0x1) {  // A terminates first, it is smaller in magnitude
   //    printf("LESSER: by early termination of A\n");
         return(LESSER);
      }
      else if(status == 0x2)  { // B terminates first, A is larger
       //  printf("GREATER: by early termination of B\n");
         return(GREATER);
      }

}

// FOR DEVELOPMENT OF MIXED RADIX CONSTANT CONVERSION
INSTRUCTION
// TAKE ACCUMULATOR A VALUE AND CONVERT TO MR, AND STORE IN
REGISTER
void cnvtmr_accumA(into reg_page, into regard)
{
into i;
uint control;
uint comp_state = EQUAL;              // start with digits being equal
uint cur_comp, status;
long double dval2;         //d.a. check rns value as it is being calculated


   printf("accumulator A before comp:\n");  //d.a test value in accA during fmult process
   print_test_as_reg(ACC_A_CHANNEL);

//  The following can be reduced to a single clock with right logic, AND
//  SHOULD BE PROCESSED IN ADVANCE TO REDUCE THE CLOCKS TO
EFFECTIVE ZERO

   control = (CMP_MODCNT_RST | CMP_MODCNT_ENA | CMP_SHFT_INIT);
// clear the modcnt counter
   IOWR(COMPARE_A_BASE, 0x0, control);
```

```
   clock_pulse();

   control = (CMP_MODDIV_ENA | CMP_CLEAR_SKIP | CMP_MODCNT_ENA);
// first moddiv clock  (seed moddiv?), clear skip FF, load adr=0 to div_base_mlab
   IOWR(COMPARE_A_BASE, 0x0, control);
   clock_pulse();
   IOWR(COMPARE_A_BASE, 0x0, 0);
   control = 0;


   i = 0;

    do {

        control = 0;
         if(i == 0) {
            control = CMP_DECOMP_LOAD | CMP_SHFT_ENA;
         }
         else {     // next iterations is decomp only
            control = CMP_SHFT_ENA;
         }

         printf("status = %x\n", read_test(CMP_OUT_STATUS, 0));        // DIRECT
STATUS PORT READ
         printf("crossbar_A = %x\n", read_test(CMP_OUT_TEST,
CMP_CROSS_A_TEST));

         control |= (CMP_MODDIV_ENA | CMP_MODCNT_ENA);         // always doing
these
         IOWR(COMPARE_A_BASE, 0x0, control);
         clock_pulse();

         IOWR(COMPARE_A_BASE, 0x0, 0);

         printf("decompA: ");
         print_test_as_reg_skip(CMP_MODDIVA);

         status = read_test(CMP_OUT_STATUS, 0);            // adjust data like old read
status port
         printf("status after clock = %x\n", status);

         printf("loop cnt i is :%x\n", i);

         i++;
         wait_key();
```

```
      } while(!(status & 0x1));


   // NOW LATCH THE MIXED RADIX VALUE TO REGISTER

    write_regA(reg_page, regard, MR_A_TO_REG);            // write the mixed radix
conversion port (0x1)

}
```

APPENDIX C: ALU_INSTRUCT.H

```
/*
 * alu_instruct.h
 *
 *  Created on: Jun 29, 2013
 *      Author: Eric
 */

#ifndef ALU_INSTRUCT_H_
#define ALU_INSTRUCT_H_

extern void read_regA(uint regf_page, uint regadr);
extern void read_regB(uint regf_page, uint regadr);

extern void write_regA(uint regf_page, uint regadr, uint input_srce);
extern void write_regB(uint regf_page, uint regadr, uint input_srce);

extern void clear_accumA(uint control);
extern void clear_accumB(uint control);

extern void load_accumA(uint regf_page, uint regadr);
extern void load_accumB(uint regf_page, uint regadr);

extern void store_accumA(uint regf_page, uint regadr);
extern void store_accumB(uint regf_page, uint regadr);

extern void add_accumA(uint regf_page, uint regadr);
extern void add_accumB(uint regf_page, uint regadr);

extern void sub_accumA(uint regf_page, uint regadr);
extern void sub_accumB(uint regf_page, uint regadr);

extern void comp_accumB(uint control);

extern void mult_accumA(uint regf_page, uint regadr);
extern void mult_accumB(uint regf_page, uint regadr);

extern void fmult_accumA(into reg_page, into regard);
extern void fmult_accumA2(into reg_page, into regard);

into comp_accumA(into reg_page, into regard);        // MAGNITUDE ONLY
comparison instruction

void print_compare(uint comp);                  // comparison result print routine
void cnvtmr_accumA(into reg_page, into regard);      // NEW ADVANCED
COMPARISON
```

```
/*

        RZ9 Nios Custom Instructions

*/

#define  CLK_PULSE()       ALT_CI_NIOS_ALU_INST1_0(255,0,0)
#define  LOAD_A(n)         ALT_CI_NIOS_ALU_INST1_0(1,n,0)            // defines of
the NIOS custom RZ9 instructions
#define  LOAD_B(n)         ALT_CI_NIOS_ALU_INST1_0(2,n,0)
#define  LOAD_AB(n1,n2)    ALT_CI_NIOS_ALU_INST1_0(3,n1,n2)
#define  STORE_A(n)        ALT_CI_NIOS_ALU_INST1_0(4,n,0)
#define  STORE_B(n)        ALT_CI_NIOS_ALU_INST1_0(5,n,0)
#define  STORE_AB(n1,n2)   ALT_CI_NIOS_ALU_INST1_0(6,n1,n2)
#define  ADD_A(n)          ALT_CI_NIOS_ALU_INST1_0(8,n,0)
#define  ADD_B(n)          ALT_CI_NIOS_ALU_INST1_0(9,n,0)
#define  ADD_AB(n1,n2)     ALT_CI_NIOS_ALU_INST1_0(10,n1,n2)
#define  SUB_A(n)          ALT_CI_NIOS_ALU_INST1_0(12,n,0)
#define  SUB_B(n)          ALT_CI_NIOS_ALU_INST1_0(13,n,0)
#define  SUB_AB(n1,n2)     ALT_CI_NIOS_ALU_INST1_0(14,n1,n2)
#define  MULT_A(n)         ALT_CI_NIOS_ALU_INST1_0(16,n,0)
#define  MULT_B(n)         ALT_CI_NIOS_ALU_INST1_0(17,n,0)
#define  MULT_AB(n1,n2) ALT_CI_NIOS_ALU_INST1_0(18,n1,n2)
#define  FMULT_A(n)        ALT_CI_NIOS_ALU_INST1_0(24,n,0)
#define  FMULT_B(n)        ALT_CI_NIOS_ALU_INST1_0(25,n,0)
#define  CNVRTFS(n)        ALT_CI_NIOS_ALU_INST1_0(38,n,0)
#define  FCNVRTF(n)        ALT_CI_NIOS_ALU_INST1_0(39,n,0)
//#define CNVRTRS(n)       ALT_CI_NIOS_ALU_INST1_0(40,n,0)
#define  TEST_COMP()       ALT_CI_NIOS_ALU_INST1_0(42,0,0) //returns the result
of the comparison
#define  COMP_A(n)         ALT_CI_NIOS_ALU_INST1_0(34,n,0)
#define  NORM_A(n)         ALT_CI_NIOS_ALU_INST1_0(43,n,0)


#endif /* ALU_INSTRUCT_H_ */
```

APPENDIX D: ARITHMETIC ALU TESTS

INDIVIDUAL ACCUMULATOR OPERATIONS.
Load accumulator A with 7 and accumulator B with 14.
This is the load accumulator A test.
(V)+ 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>
PASS
This is the load accumulator B test.
(V)+ 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 <0 0> <0 0>
PASS

Add accumulator A with 3 and accumulator B with 6.
This is the add accumulator B test.
(V)+ 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 <0 0> <0 0>
PASS
This is the add accumulator B test.
(V)+ 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 <0 0> <0 0>
PASS

Subtract accumulator A with 1 and accumulator B with 2.
This is the sub accumulator A test.
(I)+ 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 <0 0> <0 0>
PASS
This is the sub accumulator B test.
(I)+ 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 <0 0> <0 0>
PASS

Multiply accumulator A with 5 and accumulator B with 10.
This is the mult accumulator A test.
(I)+ 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 <0 0> <0 0>
PASS
This is the mult accumulator B test.
(I)+ 59 55 11 180 180 180 180 180 180 180 180 180 180 180 180 180 180 180 <0 0>
<0 0>
PASS

DUAL ACCUMULATOR OPERATIONS.
Load accumulator A with 14 and accumulator B with 7.
accumulator A after dual load:
(V)+ 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 <0 0> <0 0>
accumulator B after dual load:
(V)+ 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>

Add accumulator A with 6 and accumulator B with 3.
accumulator A after dual signed addition:
(V)+ 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 <0 0> <0 0>

accumulator B after dual signed addition:
(V)+ 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 <0 0> <0 0>

Subtract accumulator A with 2 and accumulator B with 1.
accumulator A after dual signed subtraction:
(I)+ 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 <0 0> <0 0>
accumulator B after dual signed subtraction:
(I)+ 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 <0 0> <0 0>

Multiply accumulator A with 10 and accumulator B with 5.
accumulator A after dual integer multiply:
(I)+ 59 55 11 180 180 180 180 180 180 180 180 180 180 180 180 180 180 180 <0 0> <0 0>
accumulator B after dual integer multiply:
(I)+ 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 <0 0> <0 0>

APPENDIX E: ARITHMETIC ALU TEST CODE

```
    //////////////////////////////////TESTBECNH OF
MODULES/////////////////////////////////////////////////////////////
        //write values to registers for up coming instruction test

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////*/
          printf("Beginning of instruction module test.\n");
          wait_key();
          //INDIVIDUAL ACCUMULATOR OPERATIONS
          printf("INDIVIDUAL ACCUMULATOR OPERATIONS.\n");
          wait_key();
          //LOAD A AND B
          printf("Load accumulator A with%3d", test_values[1]);
          printf(" and accumulator B with%3d.\n", test_values[2]);
          wait_key();
          load_accmA_test(1);
          load_accmB_test(2);
          wait_key();


          //ADD A AND B
          printf("Add accumulator A with%3d", test_values[3]);
          printf(" and accumulator B with%3d.\n", test_values[4]);
          wait_key();
          add_accmA_test(3);
          add_accmB_test(4);
          wait_key();

          //SUBTRACT A AND B
          printf("Subtract accumulator A with%3d", test_values[5]);
          printf(" and accumulator B with%3d.\n", test_values[6]);
          wait_key();
          sub_accmA_test(5);
          sub_accmB_test(6);
           wait_key();

           //MULTIPLY A AND B
           printf("Multiply accumulator A with%3d", test_values[7]);
           printf(" and accumulator B with%3d.\n", test_values[8]);
           wait_key();
           mult_accmA_test(7);
           mult_accmB_test(8);
           wait_key();
```

```
//DUAL ACCUMULATOR OPERATIONS//*/
printf("DUAL ACCUMULATOR OPERATIONS.\n");
wait_key();
//LOAD A AND B
printf("Load accumulator A with%3d", test_values[2]);
printf(" and accumulator B with%3d.\n", test_values[1]);
wait_key();
LOAD_AB(2,1);       //load both a and B acc at same time  //d.a.
printf("accumulator A after dual load:\n");
print_test_as_reg10(ACC_A_CHANNEL);
printf("accumulator B after dual load:\n");
print_test_as_reg10(ACC_B_CHANNEL);
wait_key();

//ADD A AND B
printf("Add accumulator A with%3d", test_values[4]);
printf(" and accumulator B with%3d.\n", test_values[3]);
wait_key();

ADD_AB(4,3);
printf("accumulator A after dual signed addition:\n");
print_test_as_reg10(ACC_A_CHANNEL);
printf("accumulator B after dual signed addition:\n");
print_test_as_reg10(ACC_B_CHANNEL);
wait_key();

//SUBTRACT A AND B
 printf("Subtract accumulator A with%3d", test_values[6]);
 printf(" and accumulator B with%3d.\n", test_values[5]);
 wait_key();
 SUB_AB(6,5);
 printf("accumulator A after dual signed subtraction:\n");
 print_test_as_reg10(ACC_A_CHANNEL);
 printf("accumulator B after dual signed subtraction:\n");
 print_test_as_reg10(ACC_B_CHANNEL);
 wait_key();

 //MULTIPLY A AND B
 printf("Multiply accumulator A with%3d", test_values[8]);
 printf(" and accumulator B with%3d.\n", test_values[7]);
 wait_key();
 MULT_AB(8,7);
 printf("accumulator A after dual integer multiply:\n");
 print_test_as_reg10(ACC_A_CHANNEL);
```

```
printf("accumulator B after dual integer multiply:\n");
print_test_as_reg10(ACC_B_CHANNEL);
wait_key();
```

APPENDIX F: FRACTIONAL MULTIPLY TEST

BEGINNING FRACTIONAL MULTIPLY SERIES TESTS.
FMULT TEST 1: 16.135792468000002 * 16.135792468000002
converted decimal value from RNS : 260.363798570365589

FMULT TEST 2: 1613.579246800000192 * 1613.579246800000192
converted decimal value from RNS : 2603637.985703655984253

FMULT TEST 3: -59.963258740999997 * -59.963258740999997
converted decimal value from RNS : 3595.592398840112764

FMULT TEST 4: -5996.325874099999965 * -5996.325874099999965
converted decimal value from RNS : 35955923.988401129841805

FMULT TEST 5: 16.135792468000002 * -59.963258740999997
converted decimal value from RNS : -967.554698749763020

FMULT TEST 6: 1613.579246800000192 * -5996.325874099999965
converted decimal value from RNS : -9675546.987497631460428

FMULT TEST 7: -59.963258740999997 * 0.000000000000000
converted decimal value from RNS : -0.000000000000000

FMULT TEST 8: 0.258741369000000 * 5432.159752999999910
converted decimal value from RNS : 1405.524451117921899

FMULT TEST 9: 0.814785236900000 * 0.512365478900000
converted decimal value from RNS : 0.417467828104918

FMULT decomposing value test
FMULT demo loop: 0.900000000000000 * 1.000000000000000
converted decimal value from RNS : 0.900000000000000

FMULT demo loop: 0.900000000000000 * 0.900000000000000
converted decimal value from RNS : 0.810000000000000

FMULT demo loop: 0.900000000000000 * 0.810000000000000
converted decimal value from RNS : 0.729000000000000

FMULT demo loop: 0.900000000000000 * 0.729000000000000
converted decimal value from RNS : 0.656100000000000

FMULT demo loop: 0.900000000000000 * 0.656100000000000
converted decimal value from RNS : 0.590490000000000

FMULT demo loop: 0.900000000000000 * 0.590490000000000
converted decimal value from RNS : 0.531441000000000

FMULT demo loop: 0.900000000000000 * 0.531441000000000
converted decimal value from RNS : 0.478296900000000

FMULT demo loop: 0.900000000000000 * 0.478296900000000
converted decimal value from RNS : 0.430467210000000

FMULT demo loop: 0.900000000000000 * 0.430467210000000
converted decimal value from RNS : 0.387420489000000

FMULT demo loop: 0.900000000000000 * 0.387420489000000
converted decimal value from RNS : 0.348678440100000

FMULT demo loop: 0.900000000000000 * 0.348678440100000
converted decimal value from RNS : 0.313810596090000

FMULT demo loop: 0.900000000000000 * 0.313810596090000
converted decimal value from RNS : 0.282429536481000

FMULT demo loop: 0.900000000000000 * 0.282429536481000
converted decimal value from RNS : 0.254186582832900

FMULT demo loop: 0.900000000000000 * 0.254186582832900
converted decimal value from RNS : 0.228767924549610

FMULT demo loop: 0.900000000000000 * 0.228767924549610
converted decimal value from RNS : 0.205891132094649

FMULT demo loop: 0.900000000000000 * 0.205891132094649
converted decimal value from RNS : 0.185302018885184

FMULT demo loop: 0.900000000000000 * 0.185302018885184
converted decimal value from RNS : 0.166771816996666

FMULT demo loop: 0.900000000000000 * 0.166771816996666
converted decimal value from RNS : 0.150094635296999

FMULT demo loop: 0.900000000000000 * 0.150094635296999
converted decimal value from RNS : 0.135085171767299

FMULT demo loop: 0.900000000000000 * 0.135085171767299
converted decimal value from RNS : 0.121576654590569

FMULT demo loop: 0.900000000000000 * 0.121576654590569

converted decimal value from RNS : 0.109418989131512

FMULT demo loop: 0.900000000000000 * 0.109418989131512
converted decimal value from RNS : 0.098477090218361

FMULT demo loop: 0.900000000000000 * 0.098477090218361
converted decimal value from RNS : 0.088629381196525
End of FMULT loop test.

APPENDIX G: FRACTIONAL MULTIPLY TEST CODE

```
//////////////////////////////////////////////////////////////////////////////////////////////
unsigned long long val_ll, val_ll2;
long long sval_ll, sval_ll2, vall;
float fval, fval2;
long double dval, dval2, dval3, stop_loop;
into i;
//////////////////////////////////////////////////////////////////////////////////////////////

   //////////////////////////////////FMULT TESTS/////////////////////////////////

          //FRACTIONAL MULTIPLY A AND B
         // reset_accm_reg(10);  //clear and reset
         dval  = 16.135792468L;
         convert_inReg_ldouble(dval, 0, 9);  //convert and load first fraction
         dval2 = -59.963258741L;
         convert_inReg_ldouble(dval2, 0, 10);  //convert and load 2nd fraction

         printf("BEGINNING FRACTIONAL MULTIPLY SERIES TESTS.\n");
         wait_key();
         /////////////////////////FMULT TEST 1/////////////////////////////////
         LOAD_A(9);
         FMULT_A(9);
         printf("FMULT TEST 1: %3.15Lf", dval);
         printf(" * %3.15Lf\n", dval);
        dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
        printf("converted decimal value from RNS : %3.15Lf\n", dval3);
        wait_key();


         /////////////////////////FMULT TEST 2/////////////////////////////////
        //   dval  = 16.135792468L;
        //dval2 = 59.963852741L;
         convert_inReg_ldouble(dval*100, 0, 9);  //convert and load first fraction
         LOAD_A(9);

         FMULT_A(9);
         printf("FMULT TEST 2: %3.15Lf", dval*100);
         printf(" * %3.15Lf\n" , dval*100);

         dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
         printf("converted decimal value from RNS : %3.15Lf\n", dval3);
         wait_key();
```

```
////////////////////////////FMULT TEST 3//////////////////////////////////
        convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
        LOAD_A(10);
        FMULT_A(10);
        printf("FMULT TEST 3: %3.15Lf", dval2);
        printf(" * %3.15Lf\n", dval2);

        dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
        printf("converted decimal value from RNS : %3.15Lf\n", dval3);
        wait_key();


        ////////////////////////////FMULT TEST 4//////////////////////////////////
        convert_inReg_ldouble(dval2*100, 0, 10);  //convert and load first fraction
        LOAD_A(10);
        FMULT_A(10);
        printf("FMULT TEST 4: %3.15Lf", dval2*100);
        printf(" * %3.15Lf\n", dval2*100);
        dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
        printf("converted decimal value from RNS : %3.15Lf\n", dval3);
        wait_key();


        ////////////////////////////FMULT TEST 5//////////////////////////////////
        convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
        convert_inReg_ldouble(dval, 0, 9);  //convert and load first fraction
        LOAD_A(10);
        FMULT_A(9);
        printf("FMULT TEST 5: %3.15Lf", dval);
        printf(" * %3.15Lf\n", dval2);

        dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
        printf("converted decimal value from RNS : %3.15Lf\n", dval3);
        wait_key();


        ////////////////////////////FMULT TEST 6//////////////////////////////////
        convert_inReg_ldouble(dval2*100, 0, 10);  //convert and load first fraction
        convert_inReg_ldouble(dval*100, 0, 9);  //convert and load first fraction
        LOAD_A(10);
        FMULT_A(9);
```

```
	printf("FMULT TEST 6: %3.15Lf", dval*100);
	printf(" * %3.15Lf\n", dval2*100);

	dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
	printf("converted decimal value from RNS : %3.15Lf\n", dval3);
	wait_key();


	//////////////////////////FMULT TEST 7/////////////////////////////////
	convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
	convert_inReg_ldouble(dval*0, 0, 9);  //convert and load first fraction
	LOAD_A(10);
	FMULT_A(9);
	printf("FMULT TEST 7: %3.15Lf", dval2);
	printf(" * %3.15Lf\n", dval*0);

	dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
	printf("converted decimal value from RNS : %3.15Lf\n", dval3);
	wait_key();


	//////////////////////////FMULT TEST 8/////////////////////////////////
	dval = 5432.159753L;
	dval2 = 0.258741369L;
	convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
	convert_inReg_ldouble(dval, 0, 9);  //convert and load first fraction
	LOAD_A(10);
	FMULT_A(9);
	printf("FMULT TEST 8: %3.15Lf", dval2);
	printf(" * %3.15Lf\n", dval);

	dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
	printf("converted decimal value from RNS : %3.15Lf\n", dval3);
	wait_key();

	//////////////////////////FMULT TEST 9/////////////////////////////////
	dval =  0.5123654789L;
	dval2 = 0.8147852369L;

	convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
	convert_inReg_ldouble(dval, 0, 9);  //convert and load first fraction
	LOAD_A(10);
	FMULT_A(9);
```

```
        printf("FMULT TEST 9: %3.15Lf", dval2);
        printf(" * %3.15Lf\n", dval);

        dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
        printf("converted decimal value from RNS : %3.15Lf\n", dval3);
        wait_key();

    ////////////////////////////////////////////FRACTIONAL MULTIPLY
LOOP/////////////////////////////////////

        dval = 1.0L;
        dval2 = 0.9L;
        stop_loop = 0.09L;
        printf("FMULT decomposing value test", dval2);
        while (dval > stop_loop)
        {
           convert_inReg_ldouble(dval2, 0, 10);  //convert and load first fraction
           convert_inReg_ldouble(dval, 0, 9);  //convert and load first fraction
           LOAD_A(10);
           FMULT_A(9);
           printf("FMULT demo loop: %3.15Lf", dval2);
           printf(" * %3.15Lf\n", dval);

           dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
           printf("converted decimal value from RNS : %3.15Lf\n", dval3);
           wait_key();
           dval = dval3;
        }
       printf("End of FMULT loop test.\n");
       wait_key();

  //////////////////////////////////////////////////////////CONVERSION
TEST///////////////////////////////////////////////////////
       //convert_inReg_llint(72, 0, 11);  //convert and load first fraction
       vall = 150;
       printf("Integer conversion Test:\n");
       wait_key();

       printf("Binary value to be converted to RNS is : %lld\n", vall);
       wait_key();
       latch_wbin_in(vall);     //LATCH integer value into reg
       CNVRTFS(11);            //store value into regfile
        //write_regB(0, 9, WB2WR_TO_REGB);
```

```
	LOAD_A(11);		//load converted integer into accumulator
	printf("Converted integer to RNS result:\n");
	print_test_as_reg10(ACC_A_CHANNEL);
	wait_key();


	dval = 0.123456789101112L;
	printf("Fractional conversion Test:\n");
	wait_key();
	printf("fractional value to be converted to RNS is : %3.15Lf\n", dval);
	val_ll = ldouble_to_fix64(dval);

	FCNVRTF(12);
	LOAD_A(12);
	printf("Converted fraction to RNS result:\n");
	print_test_as_reg10(ACC_A_CHANNEL);
	dval3 = convert_outA_ldouble();  //D.A. beware conversion  changes value in
accumA and accumB
	printf("converted decimal value from RNS : %3.15Lf\n", dval3);
	wait_key();

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/

	 printf("End of instruction module test.\n");
	 wait_key();
```

APPENDIX H: LOAD AND STORE TEST CODE

```
////////////////////////////////////////////////////////////////////////////////
//second battery of tests//D.A
//compare, store,
void test_bat2()
{
	// test_values[10] = {0, 7, 14, 3, 6, 1, 2, 5, 10};

  //////////////////////////////////LOAD AND STORE A/////////////////
	//printf("Simple load and store test");
	LOAD_A(1);
	printf("value to be stored in register:\n");
	print_test_as_reg10(ACC_A_CHANNEL);
	wait_key();

	STORE_A(3);
	wait_key();

	LOAD_A(2);
	printf("Loading different value into accumulator A:\n");
	print_test_as_reg10(ACC_A_CHANNEL);
	wait_key();

       LOAD_A(3);
       printf("loading stored value into accumulator A:\n");  //print accm b value //d.a.
	print_test_as_reg10(ACC_A_CHANNEL);
	wait_key();
 ////////////////////////////////////////////////////////////////////////////////

 ////////////////////////////////LOAD AND STORE B///////////////
		printf("Simple load and store test");
		LOAD_B(4);
		printf("value to be stored in register:\n");
		print_test_as_reg10(ACC_B_CHANNEL);
		wait_key();

		STORE_B(6);
		wait_key();

		LOAD_B(5);
		printf("Loading different value into accumulator B:\n");
		print_test_as_reg10(ACC_B_CHANNEL);
		wait_key();
```

```
        LOAD_B(6);
        printf("loading stored value into accumulator B:\n");  //print accm b value //d.a.
        print_test_as_reg10(ACC_B_CHANNEL);
        wait_key();
  //////////////////////////////////////////////////////////////////////////////////

////////////////////////////////LOAD AND STORE AB////////////////*/
            printf("Simple load and store test");
            LOAD_AB(4,1);
            printf("values to be stored in registers:\n");
            print_test_as_reg10(ACC_A_CHANNEL);
            print_test_as_reg10(ACC_B_CHANNEL);

            wait_key();

            STORE_AB(6,3);
            wait_key();

            LOAD_AB(5,2);
            printf("Loading different value into accumulator A and B:\n");
            print_test_as_reg10(ACC_A_CHANNEL);
            print_test_as_reg10(ACC_B_CHANNEL);
            wait_key();

           LOAD_AB(6,3);
           printf("loading stored value into accumulator A and B:\n");  //print accm b value
//d.a.
           print_test_as_reg10(ACC_A_CHANNEL);
           print_test_as_reg10(ACC_B_CHANNEL);
           wait_key();
  //////////////////////////////////////////////////////////////////////////////////
```

APPENDIX I: LOAD AND STORE TEST RESULTS

value to be stored in register:
(V)+ 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>
Loading different value into accumulator A:
(V)+ 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 <0 0> <0 0>
loading stored value into accumulator A:
(V)+ 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 <0 0> <0 0>
Simple load and store testvalue to be stored in register:
(V)+ 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 <0 0> <0 0>
Loading different value into accumulator B:
(V)+ 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <0 0> <0 0>
loading stored value into accumulator B:
(V)+ 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 <0 0> <0 0>
Simple load and store testvalues to be stored in registers:

APPENDIX J: COMPARISON TEST

Comparison test.
TEST 1
status = 3
crossbar_A = 6
crossbar_B = 5
decompA: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
decompB: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
the value on result bus is 2
GREATER: by early termination of B
PASS

TEST 2
status = 0
crossbar_A = 4
crossbar_B = 4e
decompA: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
decompB: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
the value on result bus is 0
LESSER: by early termination of A
PASS

TEST 3
status = 0
crossbar_A = 64
crossbar_B = 64
decompA: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
decompB: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
the value on result bus is 1
EQUAL: all digits the same
PASS

TEST 4
status = 2
crossbar_A = 4
crossbar_B = 0
decompA: (I)+ *** 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <1 0> <1 0>
decompB: (I)+ *** 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <1 0> <1 0>
status = 8
crossbar_A = 1
crossbar_B = 1
decompA: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>

decompB: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>
the value on result bus is 2
GREATER: by early termination of B
PASS

TEST 5
status = 0
crossbar_A = 51
crossbar_B = 16
decompA: (I)+ *** 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 <1 0> <1 0>
decompB: (I)+ *** 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 <1 0> <1 0>

status = 8
crossbar_A = 54
crossbar_B = 78
decompA: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>
decompB: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>
the value on result bus is 0
LESSER: by early termination of A
PASS

TEST 6
status = 0
crossbar_A = 77
crossbar_B = 77
decompA: (I)+ *** 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c <1 0> <1 0>
decompB: (I)+ *** 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c 7c <1 0> <1 0>

status = 4
crossbar_A = 7c
crossbar_B = 7c
decompA: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>
decompB: (I)+ *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <3 0> <1fe 1ff>
the value on result bus is 1
EQUAL: all digits the same
PASS

TEST 7
status = 0
crossbar_A = 4e
crossbar_B = 4
decompA: (I)+ *** 52 52 52 91 ef 77 25 1b1 de b0 c1 e 1dd 34 f4 156 c0 <1 0> <1 0>
decompB: (I)+ *** 0 0 0 3f 9d 25 13c 15f 8c 5e 6f 19b 18b 1cd a2 104 6e <1 0> <f 0>

status = 8
crossbar_A = 52
crossbar_B = 0
decompA: (I)+ *** *** 0 0 6b 1d fa 112 18a 26 143 1b6 160 9f 13a 95 144 15b <3 0> <e 0>
decompB: (I)+ *** *** 0 0 6b 1d fa 112 18a 26 143 1b6 160 9f 13a 95 144 15b <3 0> <f 0>

status = 8
crossbar_A = 0
crossbar_B = 0
decompA: (I)+ *** *** *** 0 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 <7 0> <e 0>
decompB: (I)+ *** *** *** 0 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 f3 <7 0> <f 0>

status = 4
crossbar_A = 0
crossbar_B = 0
decompA: (I)+ *** *** *** *** 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <f 0> <e 0>
decompB: (I)+ *** *** *** *** 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <f 0> <f 0>

status = 4
crossbar_A = 1
crossbar_B = 1
decompA: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0> <1f0 1ff>
decompB: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0> <1f0 1ff>
the value on result bus is 2
GREATER: by early termination of B
PASS

TEST 8
status = 0
crossbar_A = 6d
crossbar_B = 14

decompA: (I)+ *** 79 35 ed 8b ed 146 75 92 95 166 17e 18e 26 13f c4 c 9 <1 0> <1 0>
decompB: (I)+ *** 5 6f 8a ac 45 af 131 98 85 94 34 d7 87 1b6 77 1ca 16f <1 0> <1 0>

status = 8
crossbar_A = 79
crossbar_B = 5
decompA: (I)+ *** *** 3f ac fa f6 b4 101 b7 c0 178 9f 90 1a8 58 12c 139 114 <3 0> <3 0>
decompB: (I)+ *** *** 24 26 f3 38 a6 db 122 11d 35 22 28 28 f7 1c8 192 eb <3 0> <2 0>

status = 8
crossbar_A = 3f
crossbar_B = 24
decompA: (I)+ *** *** *** a6 c3 d0 9f 5d be 117 5f c2 3c 142 1d1 114 1b7 1bc <7 0> <4 0>
decompB: (I)+ *** *** *** c5 b7 30 123 12a 146 16f 9f d2 19b 62 c1 197 17 1d1 <7 0> <4 0>

status = 8
crossbar_A = a6
crossbar_B = c5
decompA: (I)+ *** *** *** *** af af af af af af af af af af af af af af <f 0> <8 0>
decompB: (I)+ *** *** *** *** c6 c6 c6 c6 c6 c6 c6 c6 c6 c6 c6 c6 c6 c6 <f 0> <8 0>

status = 0
crossbar_A = af
crossbar_B = c6
decompA: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0> <1f0 1ff>
decompB: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0> <1f0 1ff>
the value on result bus is 0
LESSER: by early termination of A
PASS

TEST 9
status = 0
crossbar_A = 31
crossbar_B = 31
decompA: (I)+ *** 6a 3d 8a 7 65 68 b0 18d 63 101 12e 2a e6 bf 106 1d3 67 <1 0> <1 0>

decompB: (I)+ *** 6a 3d 8a 7 65 68 b0 18d 63 101 12e 2a e6 bf 106 1d3 67 <1
0> <1 0>

status = 4
crossbar_A = 6a
crossbar_B = 6a
decompA: (I)+ *** *** 4a 94 a1 68 6b f e5 19d b3 1a4 1bc bc 100 7d 165 ab <3
0> <2 0>
decompB: (I)+ *** *** 4a 94 a1 68 6b f e5 19d b3 1a4 1bc bc 100 7d 165 ab <3
0> <2 0>

status = 4
crossbar_A = 4a
crossbar_B = 4a
decompA: (I)+ *** *** *** f2 ff 48 7c 73 37 1b1 a5 60 1a0 166 155 14b 152 6d
<7 0> <4 0>
decompB: (I)+ *** *** *** f2 ff 48 7c 73 37 1b1 a5 60 1a0 166 155 14b 152 6d
<7 0> <4 0>

status = 4
crossbar_A = f2
crossbar_B = f2
decompA: (I)+ *** *** *** *** ff ff ff ff ff ff ff ff ff ff ff ff ff ff <f 0> <8 0>
decompB: (I)+ *** *** *** *** ff ff ff ff ff ff ff ff ff ff ff ff ff ff <f 0> <8 0>

status = 4
crossbar_A = ff
crossbar_B = ff
decompA: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0>
<1f0 1ff>
decompB: (I)+ *** *** *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 <1f 0>
<1f0 1ff>
the value on result bus is 1
EQUAL: all digits the same
PASS

TEST 10
status = 0
crossbar_A = a
crossbar_B = 1b
decompA: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
decompB: (I)+ *** 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a <1 0>
<1 0>
the value on result bus is 0
LESSER: by early termination of A
PASS

TEST 11
status = 0
crossbar_A = 5c
crossbar_B = 77
decompA: (I)+ *** 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 54 <1 0> <1 0>
decompB: (I)+ *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <1 0> <1ff 1ff>
the value on result bus is 2
GREATER: by early termination of B

PASS
TEST 12
status = 0
crossbar_A = 52
crossbar_B = 51
decompA: (I)+ *** 5e 8a c4 66 11e ab 9c 48 15d 103 4f 12 99 1d8 88 1d7 e1 <1 0> <1 0>
decompB: (I)+ *** 3e 8d 5 f9 103 e5 cf a9 113 6f 107 f8 18c 178 44 fb 120 <1 0> <1 0>

status = 8
crossbar_A = 5e
crossbar_B = 3e
decompA: (I)+ *** *** a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 <3 0> <2 0>
decompB: (I)+ *** *** 94 c0 97 40 9b a0 b4 155 c1 16c 1be 16d 55 18 fb 160 <3 0> <2 0>

status = 8
crossbar_A = a8
crossbar_B = 94
decompA: (I)+ *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <7 0> <1fc 1ff>
decompB: (I)+ *** *** *** cb cb cb cb cb cb cb cb cb cb cb cb cb cb cb <7 0> <4 0>
the value on result bus is 0
LESSER: by early termination of A
PASS

TEST 13
status = 0
crossbar_A = 78
crossbar_B = 70
decompA: (I)+ *** 7c a8 f2 3e 9c 24 13b 15e 8b 5d 6e 19a 18a 1cc a1 103 6d <1 0> <1 0>

decompB: (I)+ *** 74 a0 da 7c 13 c1 b2 5e 173 119 65 28 af 3 9e 1ed f7 <1 0> <1 0>

status = 8
crossbar_A = 7c
crossbar_B = 74
decompA: (I)+ *** *** a8 f2 6a 1c f9 111 189 25 142 1b5 15f 9e 139 94 143 15a <3 0> <2 0>
decompB: (I)+ *** *** a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 a8 <3 0> <2 0>

status = 8
crossbar_A = a8
crossbar_B = a8
decompA: (I)+ *** *** *** f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 <7 0> <4 0>
decompB: (I)+ *** *** *** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 <7 0> <1fc 1ff>
the value on result bus is 2
GREATER: by early termination of B
PASS

End of Comparison test.

APPENDIX K: COMPARISON TEST CODE

```
///////////////////////////COMPARISON TEST//////////////////////
    //while (1){


        uint test2 = 0;
        uint test = 0;
        uint comp_status = 0;
        uint x = 0;
        test2 = test2-1;

        printf("Comparison test.\n");

        for(x=0; x<16; x++) {

        //TERMINATE TOGETHER SET (FIRST NUMBER)
        //TEST 1
        //TERMINATES TOGETHER (GREATER)
        if (x == 0){
            printf("TEST 1\n");     //MAX = 120
            write_bin_2_regf(2,6LL);//6LL);
            write_bin_2_regf(1,5LL);
        }
        //TEST 2
        //TERMINATES TOGETHER (LESSER)
        if (x == 1){
            printf("TEST 2\n");  //MAX = 120
            write_bin_2_regf(2,4LL);
            write_bin_2_regf(1,78LL);
        }
        //TEST 3
        //TERMINATES TOGETHER (EQUAL)
        if (x == 2){
            printf("TEST 3\n");  //MAX 120
            write_bin_2_regf(2,100LL);
            write_bin_2_regf(1,100LL);
        }

        //EARLY TERMINATION SET
        //B TERMINATES BEFORE A (GREATER)
        else if (x == 3){
            printf("TEST 4\n");  //MAX = 15124
            write_bin_2_regf(2,125LL);
            write_bin_2_regf(1,121LL);
        }
```

```
//A TERMINATES BEFORE B (LESSER)
else if (x == 4){
    printf("TEST 5\n"); //15124
    write_bin_2_regf(2,10245LL);
    write_bin_2_regf(1,14542LL);
}
//TERMINATE TOGETHER (EQUAL)
else if (x == 5){
    printf("TEST 6\n");
    write_bin_2_regf(2,15123LL);
    write_bin_2_regf(1,15123LL);
}

//TERMINATE TOGETER SET (LAST NUMBER)
//TERMINATE TOGETHER (GREATER)
else if (x == 6){
    printf("TEST 7\n"); //MAX = 1.59011E11   (4 DIGITS)
    write_bin_2_regf(2,621148375LL);
    write_bin_2_regf(1,621138379LL);
}
//TERMINATE TOGETHER (LESSER)
else if (x == 7){
   printf("TEST 8\n");//MAX = 1.59011E11   (4 DIGITS)
   write_bin_2_regf(2,109124500000LL);
   write_bin_2_regf(1,123489500000LL);
}
//TERMINATE TOGETHER (EQUAL)
else if (x == 8){
   printf("TEST 9\n");//MAX = 1.59011E11   (4 DIGITS)
   write_bin_2_regf(2,159010000000LL);
   write_bin_2_regf(1,159010000000LL);  //problem occurs at 20 digits
}


//MIX AND MATCH CASES
//1 and 2 digits
//A TERMINATES BEFORE B (LESSER)
else if (x == 9){
    printf("TEST 10\n");     //MAX = 120
    write_bin_2_regf(2,10LL);  //1 digits
    write_bin_2_regf(1,14789LL); //2 digits
}
//TEST 2
//TERMINATES TOGETHER (LESSER)
else if (x == 10){
    printf("TEST 11\n");  //MAX = 120
```

```
    write_bin_2_regf(2,10256LL); // 2 digits
    write_bin_2_regf(1,119LL); //1 digits
}
//3 and 4 digits

//A TERMINATES BEFORE B (LESSER)
else if (x == 11){
    printf("TEST 12\n"); //15124
    write_bin_2_regf(2,2552456LL); //3 digits
    write_bin_2_regf(1,521139458LL); //4 digits
}
//TERMINATE TOGETHER (GREATER)
else if (x == 12){
    printf("TEST 13\n");
    write_bin_2_regf(2,621138374LL); //4 digits
    write_bin_2_regf(1,2555148LL); //3 digits

}

//TEST 14
//TERMINATES TOGETHER (GREATER)
else if (x == 13){
    printf("TEST 14\n");
    mod_digit(16);
    SUB_A(0);
    STORE_A(2);
    SUB_A(0);
    STORE_A(1);
}

//TERMINATES TOGETHER (EQUAL)
else if (x == 14){    //16 DIGITS
    printf("TEST 15\n");
    mod_digit(17);
    SUB_A(0);
    STORE_A(1);
    STORE_A(2);
}
//TERMINATE TOGETHER (LESSER)
else if (x == 15){  //16 DIGITS
    printf("TEST 16\n");
    mod_digit(18);
    SUB_A(0);
    STORE_A(1);
    SUB_A(0);
    STORE_A(2);
```

```
            }


            //digit 16 and 17 tests
               load_accumA(0,2);  //load with 7
            comp_status = comp_accumA(0,1);  //compare with 14
            wait_key();
            LOAD_A(2);
            test = COMP_A(1);
            wait_key();

            printf("the value on result bus is %3d\n", test);
            if(test == EQUAL)  //1
                printf("EQUAL: all digits the same\n");
            else if(test == GREATER){  //2
              //  printf("GREATER: by early termination of B\n");
                printf("GREATER: A is larger than B");
                test = GREATER;}
            else if (test == LESSER){  // 0
              //  printf("LESSER: by early termination of A\n");
                printf("GREATER: A is less than B.");
                test = LESSER;}

//section used for testing digit comparison
            /*  else if(test == 3){
               printf("GREATER: by digit comparison\n");
               test = GREATER;}
            else if(test == 4){
                printf("LESSER: by digit comparison\n");
                test = LESSER;}
            */
            else
                printf("ERROR!!\n");  //should never happen but just in case value is not 0, 1,
or 2

            if (comp_status == test)  //check if hardware and software produce same results
              printf("PASS\n");
           else
              printf ("FAIL\n");

              wait_key();
            }
            printf("End of Comparison test.\n");
            wait_key();
//}
////////////////////////////////////////////////////////////////////////////////////////
```

```
}
////////////////////////////////////////////////////////////////////////////////////////////////////
```

APPENDIX L: MANDELBROT.C

```
/////////////////////////////////MANDELBROT TEST///////////////////////////////////////////
              printf("Begin Mandelbrot Test.\n");
              wait_key();

             bin_draw_mandelbrot(-1.5, 1.0, 0.05,1000);

              printf("Done with Mandelbrot Test.\n");
              wait_key();
  ///////////////END OF MANDELBROT TEST/////////////////////////////////



/*
 * mandelbrot.c
 *
 *  Created on: Mar 10, 2014
 *      Author: Eric
 */
#include <stdio.h>
#include <system.h>
#include <io.h>
#include <math.h>

#include "alu.h"
#include "utilities.h"
#include "alu_prims.h"
#include "alu_instruct.h"
#include "conversion.h"
#include "cnvrt_out.h"
#include "long_types.h"


#define PIX_MAP_HEIGHT  40
#define PIX_MAP_WIDTH  48

// application variable register addresses
#define RNS_Y       0
#define RNS_X       1
#define RNS_CR      2       // defines for register locations for mandelbrot routine
execution
#define RNS_CI      3
#define RNS_4       4
#define RNS_XSQR    5
#define RNS_YSQR    6
```

```
#define RNS_TEMP1   7
#define RNS_TEMP2   8
#define RNS_2       9
#define RNS_1       10
#define LOOP_1      11      // add a loop counter so we are totally RNS



#define NUMBER_OF_COORDINATES  13
/* data structure to make it easier to get at the data contained in "demo_coordinates.h" */
struct coordinates {
  double x;
  double y;
  double dim_x;
  into max_iterations;
};

const struct coordinates demo_locations2[NUMBER_OF_COORDINATES] = {
      {1.4781768172979355, 0.0, 0.01508731722831726, 50},  // 2
      {0.35805511474609375, 0.6436386108398438, 0.0259552001953125, 50},
      {-0.006125070457284523, -0.8077499668488658, 1.6338581430231508E-4, 500},
// 3
      {-1.760, 0.0, 0.05, 500},  // 1
      {-0.5570354129425543, 0.6352957489439464, 9.274065304888445E-4, 100},  // 4
      {0.42450820043388005, 0.2075353308053001, 3.301533972620563E-4, 100},  // 5
      {-0.7625592537807768, 0.08955441532683481, 4.171464390594348E-4, 200},  //
6
      {-0.10489989636229237, 0.9278526131989105, 2.3876997962775084E-4, 500},  //
8
      {-1.447650909, 0.000000000, 0.003, 500},  // 9
      {-1.457741737, 0.000000000, 0.000366211, 500},  // 10
      {-0.563892365, 0.667407990, 0.001464844, 500},  // 11
      {-0.597400665, 0.663105011, 0.005859375, 500},  // 12
      {-0.488316298, 0.609600782, 0.000122070, 700},  // 13
};

//////////////////////////////////INIT_RNS_MANDELBROT//////////////////////////////

void init_rns_mandelbrot(void)
{

   convert_inReg_ldouble(4.0, REGF_REG_BASE, RNS_4);          // use conversion unit
for initialization for now
   convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_XSQR);
   convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_YSQR);
   convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_X);
```

```
    convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_Y);
    convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_TEMP1);
    convert_inReg_ldouble(0.0, REGF_REG_BASE, RNS_TEMP2);
    convert_inReg_ldouble(2.0, REGF_REG_BASE, RNS_2);
    convert_inReg_ldouble(1.0, REGF_REG_BASE, RNS_1);


}

// same as init_rns_mandelbrot, but also inits the iteration loop cnt
void init_mandel_vars(into max_iter)
{

    init_rns_mandelbrot();

    write_bin_2_regf(LOOP_1, (long long)(max_iter));

}
 ////////////////////////////////////////////////////////////////////////////////////////////////////



 ////////////////////////////////////INT_MANDELBROT2///////////////////////////////
// pass casted double type times 0x10000000 for coordinates
// this routine calculates the mandelbrot algorithm with integer math
into int_mandelbrot2(long long cr, long long ci, into max_iter)
{
  long long xsqr=0, ysqr=0, x=0, y=0;
  into iter=0;

  // go ahead and shift these up to the new decimal offset
  ci = ci<<28;
  cr = cr<<28;

  while( ((xsqr + ysqr) < 0x0400000000000000LL) && (iter < max_iter) )
  {
    xsqr = x * x;
    ysqr = y * y;

    y = ((2 * x * y) + ci) >> 28;
    x = (xsqr - ysqr + cr) >> 28;

    iter++;
  }

  return(iter);
```

```
}


// floating point version of mandelbrot point test
into float_mandelbrot(double cr, double ci, into max_iter)
{
  double xsqr=0, ysqr=0, x=0, y=0;
  into iter=0;


  while( ((xsqr + ysqr) < 4.0) && (iter < max_iter) )
  {
    xsqr = x * x;
    ysqr = y * y;

//    printf("xsqr=%f, ysqr=%f, xsqr+ysqr=%f\n", xsqr, ysqr, xsqr+ysqr);

    y = ((2 * x * y) + ci);
    x = (xsqr - ysqr + cr);

    iter++;
  }

  return(iter);
}
 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



 /////////////////////////////////////RNS_MANDELBROT_ORIG///////////////////////////////
// the original port pin based control RNS mandelbrot prototype test routine!
// cleaned up with latest support for hard instructions
into rns_mandelbrot_orig(double cr, double ci, into max_iter)
{
into i;
into iter = 0;
double xsqr=0, ysqr=0, x=0, y=0;



    convert_inReg_ldouble(cr, REGF_REG_BASE, RNS_CR);       // NEED HARD
INSTRUCTION HERE!
    convert_inReg_ldouble(ci, REGF_REG_BASE, RNS_CI);

    LOAD_A(REGF_CONST_BASE+0);          // load zero into the accumulator
```

```
    //hardware store //D.A
    STORE_A(RNS_X);
    STORE_A(RNS_Y);
    STORE_A(RNS_TEMP1);


    while(iter < max_iter) {

        LOAD_A(RNS_TEMP1);

        //if(comp_accumA(0, RNS_4) == GREATER) {          // SOFTWARE COMPARE

        if(COMP_A(RNS_4) == GREATER) {                  //HARDWARE COMPARE

            break;
        }

        LOAD_A(RNS_X);                  // (this can be replaced by fractional square)
        FMULT_A(RNS_X);
        STORE_A(RNS_XSQR);      // x*x -> xsqr

        LOAD_A(RNS_Y);                  // (how about fractional square AB! - only needs
single multiplier!)
        FMULT_A(RNS_Y);
        STORE_A(RNS_YSQR);      // y*y -> ysqr


        LOAD_A(RNS_XSQR);
        ADD_A(RNS_YSQR);
        STORE_A(RNS_TEMP1);     // xsqr + ysqr -> temp1


        LOAD_A(RNS_X);
        MULT_A(REGF_CONST_BASE + 2);
        FMULT_A(RNS_Y);
        ADD_A(RNS_CI);                  // 2*x*y+ci -> A
        STORE_A(RNS_Y);         // A -> Y


        LOAD_A(RNS_XSQR);
        SUB_A(RNS_YSQR);
        ADD_A(RNS_CR);
        STORE_A(RNS_X);     // x = (xsqr - ysqr + cr);

        iter++;
```

```
    }

    return(iter);
}
 ////////////////////////////////////////////////////////////////////////////////////////////////////////////


/////////////////////////////MODIFIED WITH DUAL INSTRUCTIONS////////////////////////
into rns_mandelbrot_orig_dual(double cr, double ci, into max_iter)
{
into i;
into iter = 0;
double xsqr=0, ysqr=0, x=0, y=0;



    convert_inReg_ldouble(cr, REGF_REG_BASE, RNS_CR);       // NEED HARD
INSTRUCTION HERE!
    convert_inReg_ldouble(ci, REGF_REG_BASE, RNS_CI);

    LOAD_A(REGF_CONST_BASE+0);         // load zero into the accumulator


    STORE_A(RNS_X);
    STORE_A(RNS_Y);
    STORE_A(RNS_TEMP1);


    while(iter < max_iter) {

        LOAD_A(RNS_TEMP1);

//      if(comp_accumA(0, RNS_4) == GREATER) {           //SOFTWARE COMPARE
        if(COMP_A(RNS_4) == GREATER) {                   // HARDWARE COMPARE

            break;
        }

        LOAD_A(RNS_X);                  // (this can be replaced by fractional square)
        FMULT_A(RNS_X);
        STORE_A(RNS_XSQR);

        LOAD_A(RNS_Y);                  // (how about fractional square AB! - only needs
single multiplier!)
        FMULT_A(RNS_Y);
        STORE_A(RNS_YSQR);
```

```
        ADD_A(RNS_XSQR);
        STORE_A(RNS_TEMP1);

        LOAD_AB(RNS_X, RNS_XSQR);       // example use of parallel intructions, subs
for the loads

        MULT_A(REGF_CONST_BASE + 2);
        FMULT_A(RNS_Y);


        LOAD_B(RNS_XSQR);          // block above but using ALU B, only need one
active
        SUB_B(RNS_YSQR);

        ADD_AB(RNS_CI, RNS_CR);        // another parallel instruction, subs for the adds

        STORE_AB(RNS_Y, RNS_X);        // store AB is now working

        iter++;
    }

    return(iter);
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////

 ////////////////////////////////////BIN_DRAW_MANDELBROT////////////////////////////////
// this routine is the loop overhead for the integer mandelbrot algorithm
void bin_draw_mandelbrot(double current_x, double current_y, double step_dim, into
max_iter)
{
  into iters, iters2, iters3, i, j, diff_cnt;
  double leftmost_x;
  long long ix, iy, istep, diff_val;

//  unsigned long *colour_lookup_table = (unsigned long
*)COLOUR_LOOKUP_TABLE_BASE;
//  unsigned long index = 0;

  diff_cnt = 0;
  diff_val = 0;

  leftmost_x = current_x;
//  init_rns_mandelbrot();

//  init_mandel_vars(max_iter);
```

```
  init_rns_mandelbrot();                          // make this faster using moves from
temporary storage
  write_bin_2_regf(LOOP_1, (long long)(max_iter));

  //process the image
  for(i=0; i<PIX_MAP_HEIGHT ; i++)  // i counts the rows in the image
  {
    for(j=0; j<PIX_MAP_WIDTH; j++)  // j counts the columns in the row
    {

//      iters3 = int_mandelbrot2(current_x*0x10000000, current_y*0x10000000,
max_iter);    // evaluate this coordinate

      iters3 = rns_mandelbrot_orig(current_x, current_y, max_iter);
    //  iters3 = rns_mandelbrot_orig_dual(current_x, current_y, max_iter);
//      iters = float_mandelbrot(current_x, current_y, max_iter);
//      iters = int_mandelbrot2(current_x*0x10000000, current_y*0x10000000, max_iter);
// evaluate this coordinate

//      iters3 = iters;         // disable the rns part
    iters = iters3;

      if(iters <= -1) {
         iters2 = float_mandelbrot(current_x, current_y, max_iter);
         iters3 = rns_mandelbrot_orig(current_x, current_y, max_iter);
         printf("press any key to continue\n");
         wait_key();
      }


      if(iters == iters3) {
         if(iters == max_iter) {
            printf("*** ");
         }
         else {
            printf("%3d ", iters);
         }
      }
      else {
         printf("<%d,%d> ", iters, iters3);
         diff_cnt += 1;
         diff_val += abs(iters-iters3);
//       printf("*** ");
//       printf("%3d ", iters);
      }
//      if(iters == max_iter) // it's in the set
```

```
//      {
//        IOWR_32DIRECT(frame_buffer, index, SET_COLOUR);
//      }
//      else  // it's out of the set
//      {
//        IOWR_32DIRECT(frame_buffer, index, colour_lookup_table[iters &
COLOUR_MASK]);
//      }

      current_x += step_dim;    // increment coordinate to next column
    }
    printf("\n");
    current_x = leftmost_x;     // reset coordinate to first column of image
    current_y -= step_dim;      // increment coordinate to next row in image

        //wait_key();

    /* The remainder of this row loop is to make sure we don't have to wait
       for software to fill an entire frame buffer if we switch to hardware
       mode, pause, or change the colour palette.
     */
//    updateScreen(screen);  // updates the state of the touchpanel

  }

  printf("Number of differences: %d, total magnitude of differences: %llu\n", diff_cnt,
diff_val);

}
```

APPENDIX M: MANDELBROT TEST CODE & RESULTS

```
Begin Mandelbrot Test.

Number of differences: 0, total magnitude of differences: 0
Done with Mandelbrot Test.
```

Mandelbrot Test: 10 Iterations

```
Begin Mandelbrot Test.


  3   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8  12  10  12  44   6   5   5   5   5   4   4   4   4   3   3   3   3   3   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   8   9  11  24  11   8   7   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7   9  10  37  25  12   9   7   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7  12  42  16  24 ***  19  17   8   6   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   7   7   8   9  15 *** *** *** ***  19   9   7   6   6   6   6   5   5   4   4   4   4   4   4   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   7   7   8   8  10  23 *** *** *** ***  35   9   8   7   7   6   6   6   5   5   4   4   4   4   4   3   3   3
  4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8   9   9   9   9  10  11  15 *** *** *** ***  14  11  10   9   7   7   7  19   6   5   5   4   4   4   4   4   3   3
  4   4   4   4   4   4   4   4   5   5   5   5   5   6   6   6   7   8  14  50  13  11  75  21 *** ***  63  77  76 ***  24  38  14  42   9   9   9  16   8   6   5   4   4   4   4   4   4   3
  4   4   4   4   4   4   4   5   5   5   5   6   6   6   6   7   7   8  13  43 ***  17  27 *** *** *** *** *** *** *** *** *** ***  21  13  15  16  17  16   6   5   4   4   4   4   4   4   3
  4   4   4   4   4   4   5   5   6   6   6   6   6   6   7   7   7   9  11 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  25  10   6   6   5   4   4   4   4   4   4
  4   4   4   4   5   5   6   6   6   6   6   6   6   7   7   7   9  15  13  63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  21   8   7   6   5   4   4   4   4   4   4
  4   4   5   5   6   7   7   7   6   6   6   7   7   7   8   8  22 ***  26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  35  13   8   7   6   5   5   4   4   4   4   4
  5   5   6   6   8  13   8   8   8   8   8   8   8   8   8   9  12  21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  18  10  20   6   5   5   4   4   4   4   4
  5   6   6   7   8  16  10  10   9  13  11   9   9   9   9  11 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  19   6   5   5   4   4   4   4   4
  6   6   6   7   8  11  17  21  13  14  36  14  12  10  10  12  25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  16   9   7   5   5   4   4   4   4   4
  6   6   6   8   8  10  16 ***  25 *** *** ***  26  14  12  14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  15   9   6   5   5   5   4   4   4   4
  6   6   7   8   9  12  19 *** *** *** *** *** *** ***  16  17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  32   8   6   5   5   5   4   4   4   4
  7   8  18  10  11  20 *** *** *** *** *** *** *** *** ***  23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  29   7   6   5   5   5   4   4   4   4
  8   9  13  14  15  23 *** *** *** *** *** *** *** *** ***  34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***   9   7   6   5   5   5   4   4   4   4
  9  11  15 *** ***  45 *** *** *** *** *** *** *** *** ***  64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  11   8   7   6   5   5   5   4   4   4   4
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  13   9   8   7   6   5   5   5   4   4   4   4
  9  11  15 *** ***  45 *** *** *** *** *** *** *** *** ***  64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  11   8   7   6   5   5   5   4   4   4   4
  8   9  13  14  15  23 *** *** *** *** *** *** *** *** ***  34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***   9   7   6   5   5   5   4   4   4   4
  7   8  18  10  11  20 *** *** *** *** *** *** *** *** ***  23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  29   7   6   5   5   5   4   4   4   4
  6   6   7   8   9  12  19 *** *** *** *** *** *** ***  16  17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  32   8   6   5   5   5   4   4   4   4
  6   6   6   8   8  10  16 ***  25 *** *** ***  26  14  12  14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  15   9   6   5   5   5   4   4   4   4
  6   6   6   7   8  11  17  21  13  14  36  14  12  10  10  12  25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  16   9   7   5   5   4   4   4   4   4
  5   6   6   7   8  16  10  10   9  13  11   9   9   9   9  11 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  19   6   5   5   4   4   4   4   4
  5   5   6   6   8  13   8   8   8   8   8   8   8   8   8   9  12  21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  18  10  20   6   5   5   4   4   4   4   4
  4   4   5   5   6   7   7   7   6   6   6   7   7   7   8   8  22 ***  26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  35  13   8   7   6   5   5   4   4   4   4   4
  4   4   4   4   5   5   6   6   6   6   6   6   6   7   7   7   9  15  13  63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  21   8   7   6   5   4   4   4   4   4   4
  4   4   4   4   4   4   5   5   6   6   6   6   6   6   7   7   7   9  11 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  25  10   6   6   5   4   4   4   4   4   4
  4   4   4   4   4   4   4   5   5   5   5   6   6   6   6   7   7   8  13  43 ***  17  27 *** *** *** *** *** *** *** *** *** ***  21  13  15  16  17  16   6   5   4   4   4   4   4   4   3
  4   4   4   4   4   4   4   4   5   5   5   5   5   6   6   6   7   8  14  50  13  11  75  21 *** ***  63  77  76 ***  24  38  14  42   9   9   9  16   8   6   5   4   4   4   4   4   4   3
  4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8   9   9   9   9  10  11  15 *** *** *** ***  14  11  10   9   7   7   7  19   6   5   5   4   4   4   4   4   3   3
  4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   7   7   8   8  10  23 *** *** *** ***  35   9   8   7   7   6   6   6   5   5   4   4   4   4   4   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   7   7   8   9  15 *** *** *** ***  19   9   7   6   6   6   6   5   5   4   4   4   4   4   4   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7  12  42  16  24 ***  19  17   8   6   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7   9  10  37  25  12   9   7   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3
4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   8   9  11  24  11   8   7   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3   3
Number of differences: 0, total magnitude of differences: 0
Done with Mandelbrot Test
```

Mandelbrot Test: 100 Iterations

```
Begin Mandelbrot Test.

3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 7 8 12 10 12 44 6 5 5 5 5 4 4 4 4 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 8 9 11 24 11 8 7 6 5 5 5 5 4 4 4 4 4 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 7 9 10 37 25 12 9 7 6 6 5 5 5 5 4 4 4 4 4 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 7 12 42 16 24 *** 19 17 8 6 6 6 5 5 5 5 4 4 4 4 4 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 7 7 8 9 15 *** *** *** *** 19 9 7 6 6 6 6 5 5 4 4 4 4 4 4 3 3 3
4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 7 7 7 8 8 10 23 *** *** *** *** 35 9 8 7 7 6 6 6 5 5 4 4 4 4 4 3 3 3
4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 7 8 9 9 9 9 10 11 15 *** *** *** *** 14 11 10 9 7 7 7 19 6 5 5 4 4 4 4 4 3 3
4 4 4 4 4 4 4 4 5 5 5 5 5 6 6 6 7 8 14 50 13 11 75 21 177 200 63 77 76 130 24 38 14 42 9 9 9 16 8 6 5 4 4 4 4 4 4 3
4 4 4 4 4 4 4 5 5 5 5 6 6 6 6 7 7 8 13 43 *** 17 27 *** *** *** *** *** *** *** *** *** *** 21 13 15 16 17 16 6 5 4 4 4 4 4 4 3
4 4 4 4 4 4 5 5 6 6 6 6 6 6 7 7 7 9 11 298 *** *** *** *** *** *** *** *** *** *** *** *** *** *** 653 *** *** 25 10 6 6 5 4 4 4 4 4 4
4 4 4 4 5 5 6 6 6 6 6 6 6 7 7 7 9 15 13 63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 21 8 7 6 5 4 4 4 4 4 4
4 4 5 5 6 7 7 7 6 6 6 7 7 7 8 8 22 137 26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 35 13 8 7 6 5 5 4 4 4 4 4
5 5 6 6 8 13 8 8 8 8 8 8 8 8 8 9 12 21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 18 10 20 6 5 5 4 4 4 4 4
5 6 6 7 8 16 10 10 9 13 11 9 9 9 9 11 195 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 783 19 6 5 5 4 4 4 4 4
6 6 6 7 8 11 17 21 13 14 36 14 12 10 10 12 25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 16 9 7 5 5 4 4 4 4 4
6 6 6 8 8 10 16 *** 25 272 *** 126 26 14 12 14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 15 9 6 5 5 5 4 4 4 4
6 6 7 8 9 12 19 *** *** *** *** *** *** *** 16 17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 32 8 6 5 5 5 4 4 4 4
7 8 18 10 11 20 *** *** *** *** *** *** *** *** *** 23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 29 7 6 5 5 5 4 4 4 4
8 9 13 14 15 23 *** *** *** *** *** *** *** *** *** 34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 9 7 6 5 5 5 4 4 4 4
9 11 15 134 *** 45 *** *** *** *** *** *** *** *** *** 64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 11 8 7 6 5 5 5 4 4 4 4
145 177 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 13 9 8 7 6 5 5 5 4 4 4 4
9 11 15 134 *** 45 *** *** *** *** *** *** *** *** *** 64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 11 8 7 6 5 5 5 4 4 4 4
8 9 13 14 15 23 *** *** *** *** *** *** *** *** *** 34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 9 7 6 5 5 5 4 4 4 4
7 8 18 10 11 20 *** *** *** *** *** *** *** *** *** 23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 29 7 6 5 5 5 4 4 4 4
6 6 7 8 9 12 19 *** *** *** *** *** *** *** 16 17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 32 8 6 5 5 5 4 4 4 4
6 6 6 8 8 10 16 *** 25 272 *** 126 26 14 12 14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 15 9 6 5 5 5 4 4 4 4
6 6 6 7 8 11 17 21 13 14 36 14 12 10 10 12 25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 16 9 7 5 5 4 4 4 4 4
5 6 6 7 8 16 10 10 9 13 11 9 9 9 9 11 195 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 783 19 6 5 5 4 4 4 4 4
5 5 6 6 8 13 8 8 8 8 8 8 8 8 8 9 12 21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 18 10 20 6 5 5 4 4 4 4 4
4 4 5 5 6 7 7 7 6 6 6 7 7 7 8 8 22 137 26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 35 13 8 7 6 5 5 4 4 4 4 4
4 4 4 4 5 5 6 6 6 6 6 6 6 7 7 7 9 15 13 63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 21 8 7 6 5 4 4 4 4 4 4
4 4 4 4 4 4 5 5 6 6 6 6 6 6 7 7 7 9 11 335 *** *** *** *** *** *** *** *** *** *** *** *** *** *** 653 *** *** 25 10 6 6 5 4 4 4 4 4 4
4 4 4 4 4 4 4 5 5 5 5 6 6 6 6 7 7 8 13 43 *** 17 27 *** *** *** *** *** *** *** *** *** *** 21 13 15 16 17 16 6 5 4 4 4 4 4 4 3
4 4 4 4 4 4 4 4 5 5 5 5 5 6 6 6 7 8 14 50 13 11 75 21 177 200 63 77 76 130 24 38 14 42 9 9 9 16 8 6 5 4 4 4 4 4 4 3
4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 7 8 9 9 9 9 10 11 15 *** *** *** *** 14 11 10 9 7 7 7 19 6 5 5 4 4 4 4 4 3 3
4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 7 7 7 8 8 10 23 *** *** *** *** 35 9 8 7 7 6 6 6 5 5 4 4 4 4 4 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 7 7 8 9 15 *** *** *** *** 19 9 7 6 6 6 6 5 5 4 4 4 4 4 4 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 7 12 42 16 24 *** 19 17 8 6 6 6 5 5 5 5 4 4 4 4 4 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 7 9 10 37 25 12 9 7 6 6 5 5 5 5 4 4 4 4 4 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 8 9 11 24 11 8 7 6 5 5 5 5 4 4 4 4 4 3 3 3 3 3 3
Number of differences: 0, total magnitude of differences: 0
Done with Mandelbrot Test.
```

Mandelbrot Test: 1000 Iterations

```
Begin Mandelbrot Test.

   3   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8  12  10  12  44   6   5   5   5   5   4   4   4   4   3   3   3   3   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   8   9  11  24  11   8   7   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7   9  10  37  25  12   9   7   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7  12  42  16  24 ***  19  17   8   6   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   7   7   8   9  15 *** *** *** ***  19   9   7   6   6   6   6   5   5   4   4   4   4   4   4   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   7   7   8   8  10  23 *** *** *** ***  35   9   8   7   7   6   6   6   5   5   4   4   4   4   4   3   3   3
   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8   9   9   9   9  10  11  15 *** *** *** ***  14  11  10   9   7   7   7  19   6   5   5   4   4   4   4   4   3   3
   4   4   4   4   4   4   4   4   5   5   5   5   5   6   6   6   7   8  14  50  13  11  75  21 177 200  63  77  76 130  24  38  14  42   9   9   9  16   8   6   5   4   4   4   4   4   4   3
   4   4   4   4   4   4   4   5   5   5   5   6   6   6   6   7   7   8  13  43 ***  17  27 *** *** *** *** *** *** *** *** *** ***  21  13  15  16  17  16   6   5   4   4   4   4   4   4   3
   4   4   4   4   4   4   5   5   6   6   6   6   6   6   7   7   7   9  11 298 *** *** *** *** *** *** *** *** *** *** *** *** *** *** 653 *** ***  25  10   6   6   5   4   4   4   4   4   4
   4   4   4   4   5   5   6   6   6   6   6   6   6   7   7   7   9  15  13  63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  21   8   7   6   5   4   4   4   4   4   4
   4   4   5   5   6   7   7   7   6   6   6   7   7   7   8   8  22 137  26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  35  13   8   7   6   5   5   4   4   4   4   4
   5   5   6   6   8  13   8   8   8   8   8   8   8   8   8   9  12  21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  18  10  20   6   5   5   4   4   4   4   4
   5   6   6   7   8  16  10  10   9  13  11   9   9   9   9  11 195 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 783  19   6   5   5   4   4   4   4   4
   6   6   6   7   8  11  17  21  13  14  36  14  12  10  10  12  25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  16   9   7   5   5   4   4   4   4   4
   6   6   6   8   8  10  16 ***  25 272 *** 126  26  14  12  14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  15   9   6   5   5   5   4   4   4   4
   6   6   7   8   9  12  19 *** *** *** *** *** *** ***  16  17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  32   8   6   5   5   5   4   4   4   4
   7   8  18  10  11  20 *** *** *** *** *** *** *** *** ***  23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  29   7   6   5   5   5   4   4   4   4
   8   9  13  14  15  23 *** *** *** *** *** *** *** *** ***  34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***   9   7   6   5   5   5   4   4   4   4
   9  11  15 134 ***  45 *** *** *** *** *** *** *** *** ***  64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  11   8   7   6   5   5   5   4   4   4   4
 145 177 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  13   9   8   7   6   5   5   5   4   4   4   4
   9  11  15 134 ***  45 *** *** *** *** *** *** *** *** ***  64 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  11   8   7   6   5   5   5   4   4   4   4
   8   9  13  14  15  23 *** *** *** *** *** *** *** *** ***  34 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***   9   7   6   5   5   5   4   4   4   4
   7   8  18  10  11  20 *** *** *** *** *** *** *** *** ***  23 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  29   7   6   5   5   5   4   4   4   4
   6   6   7   8   9  12  19 *** *** *** *** *** *** ***  16  17 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  32   8   6   5   5   5   4   4   4   4
   6   6   6   8   8  10  16 ***  25 272 *** 126  26  14  12  14 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  15   9   6   5   5   5   4   4   4   4
   6   6   6   7   8  11  17  21  13  14  36  14  12  10  10  12  25 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  16   9   7   5   5   4   4   4   4   4
   5   6   6   7   8  16  10  10   9  13  11   9   9   9   9  11 195 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** 783  19   6   5   5   4   4   4   4   4
   5   5   6   6   8  13   8   8   8   8   8   8   8   8   8   9  12  21 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  18  10  20   6   5   5   4   4   4   4   4
   4   4   5   5   6   7   7   7   6   6   6   7   7   7   8   8  22 137  26 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  35  13   8   7   6   5   5   4   4   4   4   4
   4   4   4   4   5   5   6   6   6   6   6   6   6   7   7   7   9  15  13  63 *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***  21   8   7   6   5   4   4   4   4   4   4
   4   4   4   4   4   4   5   5   6   6   6   6   6   6   7   7   7   9  11 335 *** *** *** *** *** *** *** *** *** *** *** *** *** *** 653 *** ***  25  10   6   6   5   4   4   4   4   4   4
   4   4   4   4   4   4   4   5   5   5   5   6   6   6   6   7   7   8  13  43 ***  17  27 *** *** *** *** *** *** *** *** *** ***  21  13  15  16  17  16   6   5   4   4   4   4   4   4   3
   4   4   4   4   4   4   4   4   5   5   5   5   5   6   6   6   7   8  14  50  13  11  75  21 177 200  63  77  76 130  24  38  14  42   9   9   9  16   8   6   5   4   4   4   4   4   4   3
   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   8   9   9   9   9  10  11  15 *** *** *** ***  14  11  10   9   7   7   7  19   6   5   5   4   4   4   4   4   3   3
   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   6   6   7   7   7   8   8  10  23 *** *** *** ***  35   9   8   7   7   6   6   6   5   5   4   4   4   4   4   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   7   7   8   9  15 *** *** *** ***  19   9   7   6   6   6   6   5   5   4   4   4   4   4   4   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7  12  42  16  24 ***  19  17   8   6   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   7   9  10  37  25  12   9   7   6   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3
   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   6   6   6   8   9  11  24  11   8   7   6   5   5   5   5   4   4   4   4   4   3   3   3   3   3   3
Number of differences: 0, total magnitude of differences: 0
Done with Mandelbrot Test.
```

Mandelbrot Test: 10000 Iterations

APPENDIX N: COMPARISON OF PAIRWISE VS. NON-PAIRWISE MODULI

INTEGERS	RESIDUE DIGITS			INTEGERS	RESIDUE DIGITS		
	MODULI				MODULI		
	2	3	4		2	3	4
0	0	0	0	16	0	1	0
1	1	1	1	17	1	2	1
2	0	2	2	18	0	0	2
3	1	0	3	19	1	1	3
4	0	1	0	20	0	2	0
5	1	2	1	21	1	0	1
6	0	0	2	22	0	1	2
7	1	1	3	23	1	2	3
8	0	2	0	24	0	0	0
9	1	0	1	25	1	1	1
10	0	1	2	26	0	2	2
11	1	2	3	27	1	0	3
12	0	0	0	28	0	1	0
13	1	1	1	29	1	2	1
14	0	2	2	30	0	0	2
15	1	0	3	31	1	1	3

INTEGERS	RESIDUE DIGITS			INTEGERS	RESIDUE DIGITS		
	MODULI				MODULI		
	2	3	5		2	3	5
0	0	0	0	16	0	1	1
1	1	1	1	17	1	2	2
2	0	2	2	18	0	0	3
3	1	0	3	19	1	1	4
4	0	1	4	20	0	2	0
5	1	2	0	21	1	0	1
6	0	0	1	22	0	1	2
7	1	1	2	23	1	2	3
8	0	2	3	24	0	0	4
9	1	0	4	25	1	1	0
10	0	1	0	26	0	2	1
11	1	2	1	27	1	0	2
12	0	0	2	28	0	1	3
13	1	1	3	29	1	2	4
14	0	2	4	30	0	0	0
15	1	0	0	31	1	1	1

MODULI	2,3,4	2,3,5
Product	24	30
Product of GCD Pairs	2	1
Actual # of Residue Representations	12	30

APPENDIX O: DATA PROCESSING AND ARITHMETIC INSTRUCTIONS [2]

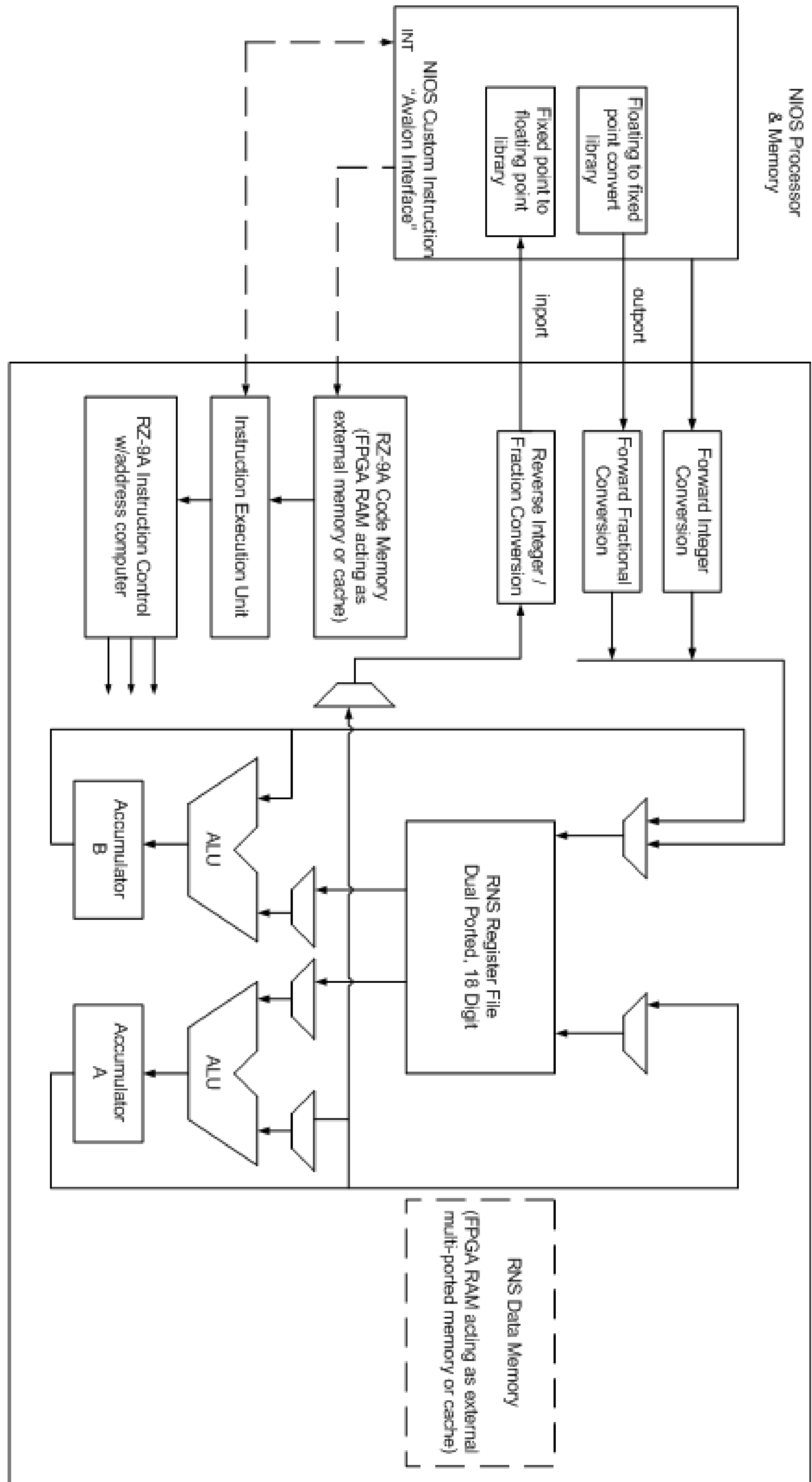
Mnemonic	Operation	Action
add_A	A ←A + rX	add register value to Accumulator A
sub_A	A← A - rX	Subtract register value from Accumulator A
mult_A	A ←A * rX	Integer multiply accumulator A
sqr_A	A ←A * A	integer square accumulator A
fmult_A	A ←A * rX	fractional multiply accumulator A
norm_A	A ← A / frng	divide by fractional range
fsqr_A	A ←A * A	fractional square accumulator A
divu_A	A ← A / rX	unsigned integer divide A
divs_A	A ← A / rX	signed integer divide A
fdiv_A	A ← A / rX	fractional integer divide A
cmpu_A	ASR ← (A >rX)	unsigned compare A
cmps_A	ASR ← (A >rX)	signed compare A
neg_A	A ← negate (A)	negate A
clr_A	A ← 0	clear accumulator A
sign_A	ASR ← sign(A)	sign extended A
add_B	B ←B + rX	add register value to accumulator B
sub_B	B← B - rX	Subtract register value from accumulator B
mult_B	B ←B * rX	Integer multiply accumulator B
sqr_B	B ←B * B	integer square accumulator B
fmult_B	B ←B * rX	fractional multiply accumulator B
norm_B	B ← B / frng	divide by fractional range
fsqr_B	B ← B / rX	Fractional square of B
divu_B	B ← B / rX	unsigned integer divide B
divs_B	B ← B / rX	signed integer divide B
fdiv_B	B ← B / rX	fractional integer divide B
cmpu_B	ASR ← (B >rX)	unsigned compare B
cmps_B	ASR ← (B >rX)	signed compare B
neg_B	B ← negate (B)	negate B
clr_B	B ← 0	clear B
sign_B	ASR ← sign(B)	sign extended B

APPENDIX P: DATA CONVERSION INSTRUCTIONS [2]

Mnemonic	Operation	Action
cvrtmr_A	A → reg	Convert A to mixed radix
fcnvrt_u	[u] → B	Forward convert unsigned integer full
fcnvrt_s	[s] → B	Forward convert signed integer full
fcnvrt_u32	u32 → B	Forward convert unsigned int. 32 bit
fcnvrt_s32	s32 → B	Forward convert signed int. 32 bit
fcnvrt_u64	u64 → B	Forward convert unsigned int. 64 bit
fcnvrt_s64	s64 → B	Forward convert signed int. 64 bit
rcnvrt_u	B → [u]	Reverse convert unsigned integer full
rcnvrt_s	B → [s]	Reverse convert signed integer full
rcnvrt_u32	B → u32	Reverse convert unsigned int. 32 bit
rcnvrt_s32	B → s32	Reverse convert signed int. 32 bit
rcnvrt_u64	B → u64	Reverse convert unsigned int. 64 bit
rcnvrt_s64	B → s64	Reverse convert signed int. 64 bit
fcnvrt_f64	[32.32] → B	Forward convert fractional 32.32 bit
fcnvrt_f128	[64.64] → B	Forward convert fractional 64.64 bit
rcnvrt_f64	B → [32.32]	Reverse convert fractional 32.32 bit
rcnvrt_f128	B → [64.64]	Reverse convert fractional 64.64 bit
fcnvrt_fp	FP → B (B=64.64)	Forward convert double FP to 64.64
rcnvrt_fp	B → FP (B=64.64)	Reverse convert 64.64 to double FP

[x] denotes forward or reverse conversion holding register
u32 denotes unsigned NIOS register argument
s32 denotes signed NIOS register argument
u64 represents unsigned register argument
s64 represents signed register argument
FP represents a double precision floating point argument

APPENDIX Q: REZ9-A COPROCESSOR



REFERENCES

[1] Szabo, N. S., & Tanaka, R. I. (1967*). Residue Arithmetic and its Applications to Computer Technology.* New York: McGraw-Hill Book Company.

[2] Digital Systems Research Inc. (2014). *REZ9-A ALU Architecture.* Las Vegas.

[3] Olsen, E. (2013). *Patent No. US20140129601.* United States of America.

[4] Shen, J. P., & Lipasti, M. H. (2005). *Modern Processor Design: Fundamentals of Superscalar Pcoessors.* Long Grove, IL: Wveland Press Inc.

[5] Harris, D. M., & Harris, S. L. (2013). *Digital Design and Computer Architecture.* Waltham, MA: Elsevier.

[6] Altera, *Nios II Custom Instructions User Guide*, ver. 2.0, 2011

[7] Altera, *Nios II Processor Reference Handbook*, ver. 13.1.0 2014

[8] Altera, *Quartus II Handbook Vol. 1*, ver. 14.0.0 2014

VITA

Graduate College

University of Nevada, Las Vegas

Daniel Anderson

Degree:

Bachelor of Science, Computer Engineering, 2011

University of Nevada, Las Vegas

Employment:

JT3, 2014

Digital System Research, 2013-2014

Thesis Title: Design and Implementation of an Instruction Set Architecture an and Instruction Execution Unit for the REZ9 Coprocessor System

Thesis Examination Committee:

Chairperson, R. Jacob Baker, Ph. D.

Committee Member, Venkatesan Muthukumar, Ph.D.

Committee Member, Henry Selvaraj, Ph. D.

Committee Member, Evangelo Yfantis, Ph. D.