VHDL Pipelined Processor Project

ECE 449 - Computer Architecture

April 6, 2018

Final Report

Pearson Brownlee **(V00802876)**

Kyle Cathers **(V00851761)**

Table of Contents

List of Tables and Figures

FIGURES

TABLES

Executive Summary

Processors have been a key component of technological innovation since the mid- 20th century. This project revolves around designing and implementing a 16-bit pipelined RISC processor in the VHDL hardware programming language. The processor was designed to be implemented onto the Basys 3 FPGA, which contained numerous switches and buttons for control. The processor was required to be able to operate on assembly code to be loaded onto an STM32F4 bootloader. The assembly instructions were of the formats, A, B, and L.

The processor contains 5 stages:

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write-back

The instruction fetch stage was designed to include the program counter, ROM (bootloader), STM32F4, and RAM components. The program counter would send an address to memory which would cause the memory to send an instruction to the decode stage. Depending on the address of the program counter, either the ROM or the RAM would be selected to read from.

The decode stage was designed to decode the incoming instruction into the OPCODE and, depending on the instruction type, the appropriate data and or register indexes. Using the indexes, data would be obtained from the register file and passed along to the execute stage.

The execute stage contains the processor ALU (Arithmetic Logic Unit). The ALU was responsible for executing the various logic and or arithmetic instructions fed from the decode stage. Forwarding was also implemented in this stage in order to avoid data dependency hazards of the pipeline. If the incoming instruction didn't need the ALU, it would be passed directly onto the memory access stage.

The final two stages are the memory access stage and writeback stage. The memory access stage was designed to write and or read from the RAM during load or store instructions. The writeback stage uses the destination register index and writes the result of the instruction calculation back into the register file.

The final design was a success after implementation onto the Basys 3 board. Two programs were ran, which aimed at ensuring the branch, overflow handling, hazard handling, and general functionality was tested. Upon successful execution, it was found that the processor was able to run at a frequency of approximately 50MHz before program mishandling. Additionally, the speed of the multiplier was tested and found to have a time delay of 20.400ns, 3ns over the optimized scenario.

Glossary of Acronyms

ALU	Arithmetic Logic Unit
FPGA	Field Programmable Array
LSB	Least Significant Bits
MSB	Most Significant Bits
PC	Program Counter. Increments addresses in order to control the flow of the processor pipeline.
RAM	Random Access Memory
ROM	Read Only Memory
RAR	Read after Read (not a hazard)
RAW	Read after Write hazard
VHDL	VHSIC Hardware Description Language. This is the hardware programming language used in the project.
WAR	Write after Read hazard
WAW	Write after Write hazard

1 Objective

The goal of this project is to design and implement a 16-bit pipelined 5-Stage Processor in VHDL.

Section 1.1 of the objective will cover the different design specifications while Section 1.2 will cover the scope of this document.

1.1 Design Specifications

For the 5-stage pipeline VHDL processor project, the designed processor was to be able to fetch program instructions from memory, decode, and then execute them. The required instruction formats were:

- A – Format
- B – Format
- L – Format

The processor was to be able to handle each type, as well as deal with any hazards which may occur from executing them in any sequence. A pinout of the processor system (including memory) can be seen below (Figure 1.1).

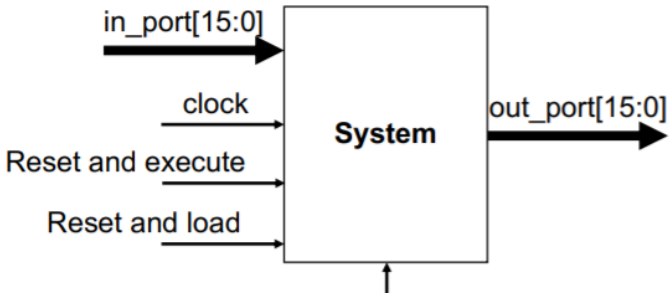

Figure 1.1: System Pinout [1]

It should be noted that two of the input signals are named “Reset & Execute” and “Reset & Load”. These inputs are tied to buttons which are used by the initial bootloader program. On the Reset & Load button press, the program counter must vector to address 0x0002 in order to load instructions through the STM32 (connected by the in_port and out_port PMOD header pins on the Basys 3). When Reset & Load is pressed, the program counter must vector to address 0x0000 in order to execute the instructions. This will be covered further in the design sections.

The processor is required to contain 5 stages: fetch, decode, execute, memory access, and write-back. In order to perform each stage, the processor system is to contain the following key components:

- Program Counter
- Stage Registers (boundary between each pipeline stage)
- Register File
- ALU
- ROM
- RAM
- Bootloader
- Hex Driver

A simple system level block diagram can be seen in Figure 1.2 and can also be found in the introductory slides.

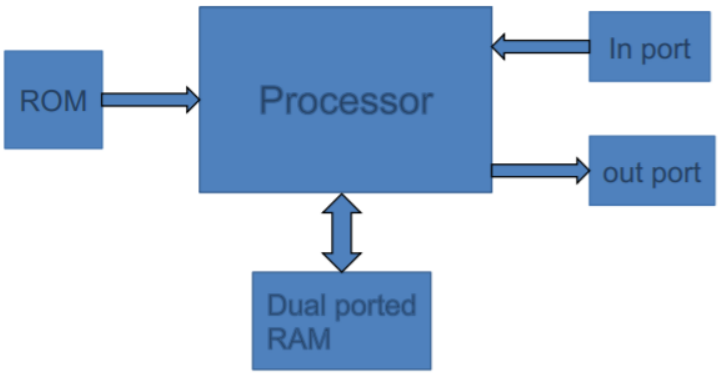

Figure 1.2: System Description Diagram [1]

In order to provide a platform for the processor, the Basys 3 FPGA is to be used (see Figure 1.3). In addition to the Basys 3, an STM32F0 is to be connected in order to provide functionality to the bootloader, as designed by Brent Sirna.

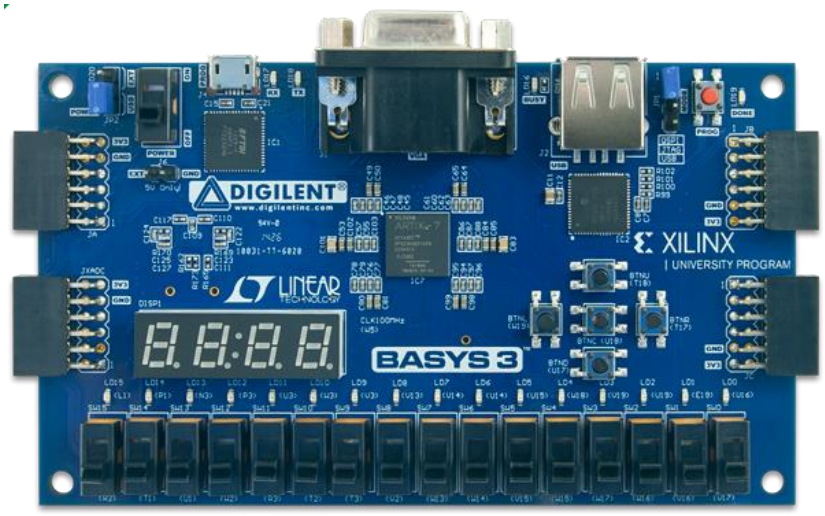

Figure 1.3: Basys 3 FPGA [2]

The goal of the final version is to be able to execute a series of test programs and ensure that each facet of the processor is functional. A copy of the final test codes can be found in Appendix C. Additionally, it is expected that the 16-bit multiplier is able to complete an execution in a time which is within 5ns of the control algorithm execution time.

1.2 Scope

This report will cover the design and implementation of the pipelined VHDL processor. Each component of the processor and its implementation will be covered. In addition, the methods of dealing with hazards will also be covered. The results of the final test code and will be covered along side with conclusions which were obtained from the project.

2 Introduction

2.1 Processor History

A processor or CPU refers to a component in computing which is responsible for containing a computer system's main logic and control units. The first CPUs were said to have had their origin in the form of stored-program computers which were designed and tested in the late 1940s [4]. These computers operated by executing operations from limited "instruction sets". One computer of note is the ENIAC which was primarily designed by J. Presper Eckert and John William Mauchly. This computer is credited as being one of the originators and inspirators for modern CPUs. [4]

2.2 Reduced Instruction Set Computer CPU's

Reduced Instruction Set Computer (RISC) CPUs are ones which allow a lower CPI (cycles per instruction) than complex instruction set computers (CISC), due to their limited and simple instruction sets [5]. RISC CPUs are traditionally comprised of a 5 - stage pipelined processor. The five stages of the processor are as follows:

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Writeback

Another common feature of RISC architecture is their load/store architecture, in which memory accesses are made as per specific instructions rather than as a part of executing instructions. [5]

The processor made for this project is based off the 5-stage RISC design.

2.3 VHDL

VHSIC Hardware Description Language (VHDL) is a hardware description language which is used commonly used for electronic design. The language supports the digital design and simulation of various technologies such as field-programmable gate arrays (FPGAs), integrated circuits (ICs), and processors.

For the purposes of the project, VHDL will be used to create, test, and implement the processor. Vivado, a software developed by Xilinx will act as the IDE for the coding language.

The standard libraries used for the project are:

- IEEE
- IEEE.STD_LOGIC_1164
- IEEE.NUMERIC_STD.ALL

2.4 Instruction Sets

For the project, the processor is to be designed to take in instructions from three key instruction formats: A, B, and L. Every instruction is 2-bytes long (1 word), with the 7 most significant bits belonging to the OPCODE. The opcode allows the IF/ID stage register to determine the type and format of the instruction. Each instruction type has a different route through the data path and structure. All the required instructions can be seen in Table A of Appendix A, along with the type and a brief description.

The following three sections will cover what each format entails and the structure of the types within each format.

2.4.1 A Format

A Format instructions refer to instructions which consist of arithmetic and logic. The structure of all the Format A types can be seen in Figure 2.1. Format A instructions were the first instructions to be implemented into the processor

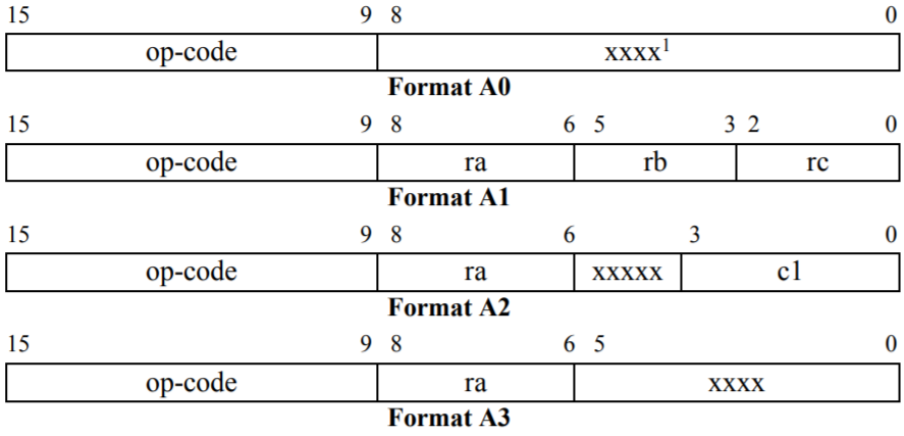

Figure 2.1: A – Format Instructions [1]

2.4.2 B Format

B Format instructions refer to instructions which consist of branching and subroutines. The structure of the Format B instructions is shown below in Figure 2.2. B format instructions were the second instruction type to be implemented in the processor.

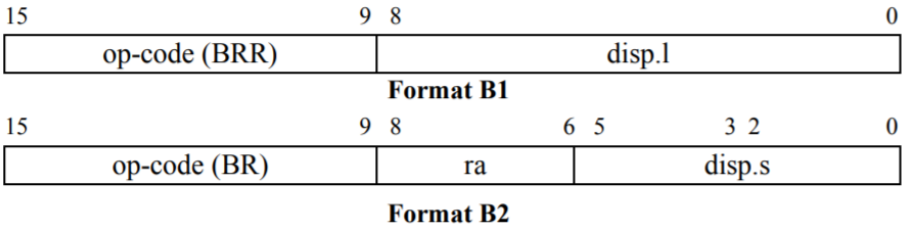

Figure 2.2: B – Format Instructions [1]

2.4.3 L – Format

L Format instructions refer to instructions which deal with loading, storing, and moving data between the register file and memory. The structure of Format L instructions can be seen in Figure 2.3. Format L was the final instruction set to be implemented into the processor.

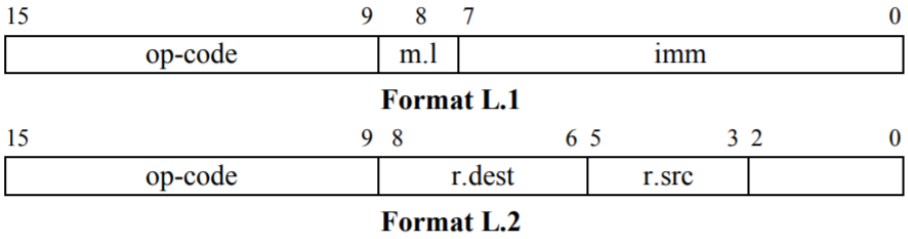

Figure 2.3: L – Format Instructions [1]

3 Design of 5-Stage Data Path

A 5-stage pipelined processor is designed to encompass the 5 stages of the data path:

1. Fetch
2. Decode
3. Execute
4. Memory Access
5. Writeback

Each stage was designed to be separated by synchronous (rising edge sensitive) “Stage Registers”, which can be seen in Figure 3.1 below. The stage registers were responsible for collecting stage signals, performing decoding, and directing the flow of the data path. The following subsections will sequentially cover each stage and describe the operations within each.

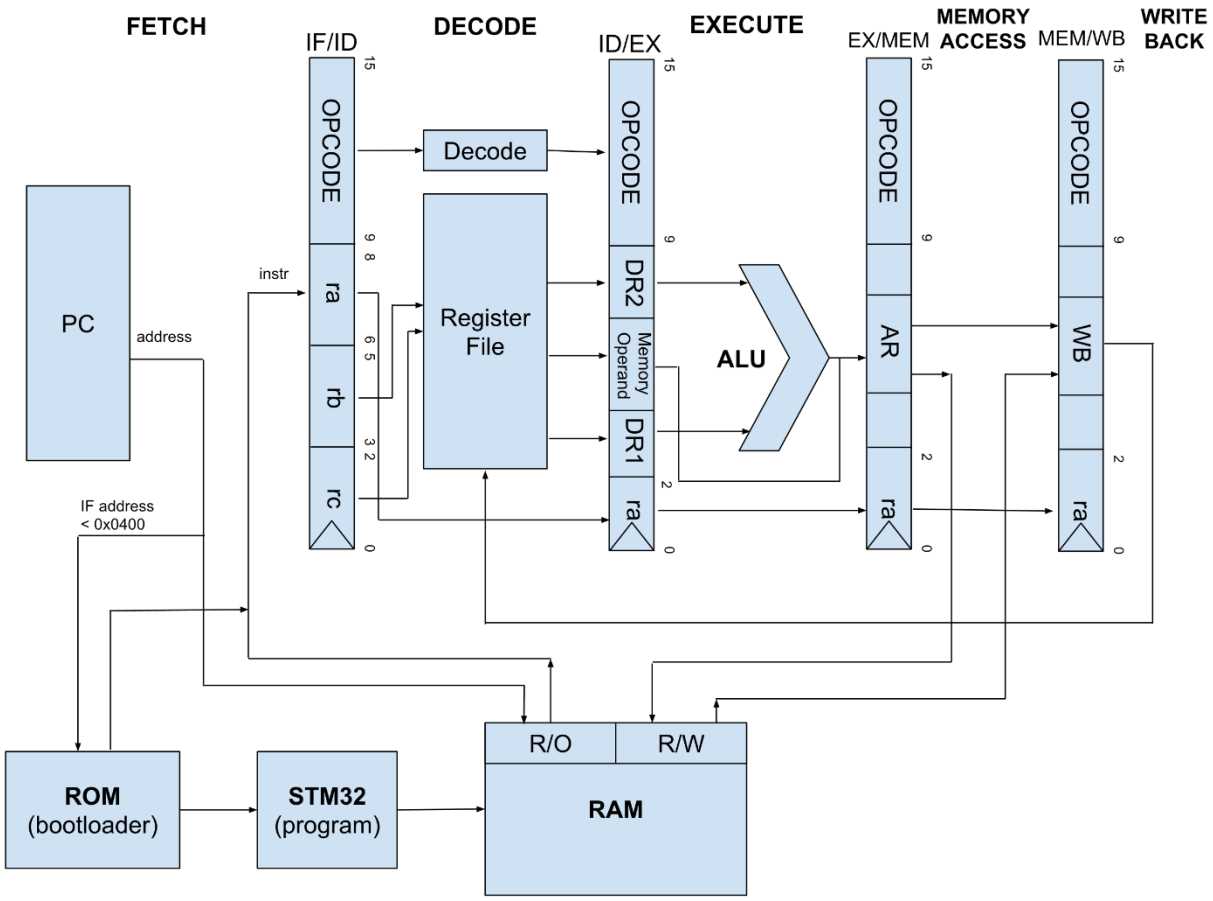

Figure 3.1: Processor Pipeline Diagram

3.1 Fetch

The goal of the Fetch stage is to fetch instructions from the ROM and RAM using the value of the Program Counter. The Program Counter outputs addresses which correspond to instructions which have been loaded into the memory. A diagram highlighting the important blocks and paths of the Fetch stage can be seen below (Figure 3.2).

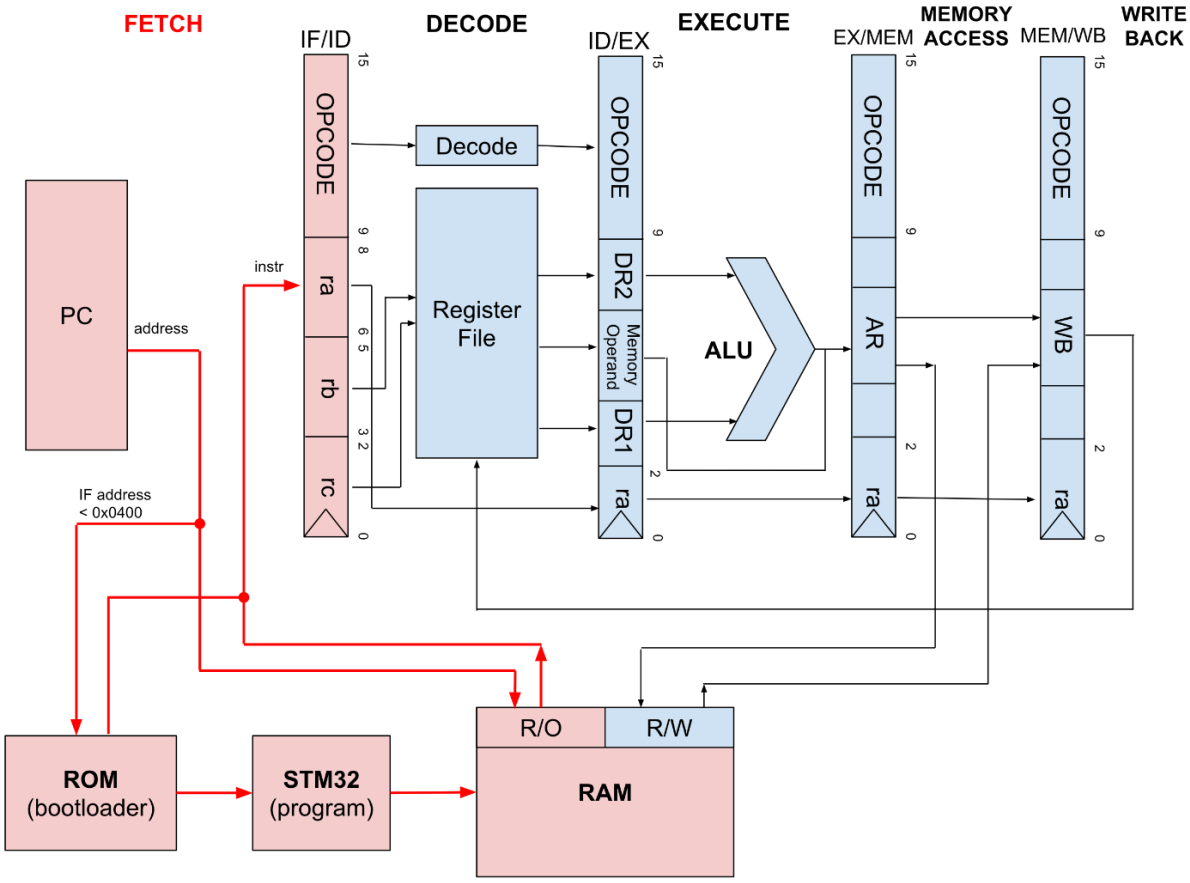

Figure 3.2: Fetch Stage Diagram

When the processor begins the pipeline, the Program Counter begins counting. If there is no signal on RESETLOAD and RESETEXECUTE, then the Fetch stage will take the address value of the Program Counter and pass it to memory (either ROM or RAM).

If the Program Counter address is less than 0x0400 (1024 bytes) then the instruction will be retrieved from ROM, if the address is greater than 0x0400, then the instruction will be retrieved from RAM.

It is extremely important to note that the methods for deciding whether to read from ROM or RAM, is programmed into the processor itself. This will cause issue for compatibility of the CPU with other sizes of RAM and ROM. It is recommended that in the next revision, that this be fixed by moving the address comparator out of the processor in order to allow for flexibility.

The instruction is then passed to the IF/ID stage register for the decode stage.

3.2 Decode

The decode stage is responsible for decoding the incoming instructions, storing the instruction components to the register file, and passing the decoded instructions onto the execute stage.

The IF/ID stage register is responsible for decoding the instruction which is received from memory. The most significant bits (MSB) of the instruction are first read and designated as the OPCODE of the instruction (see Appendix A). Using the OPCODE, the format and format type are determined, and the remainder of the instruction is divided into the appropriate signals (according to the instruction set) as described in Figures 2.1-2.3.

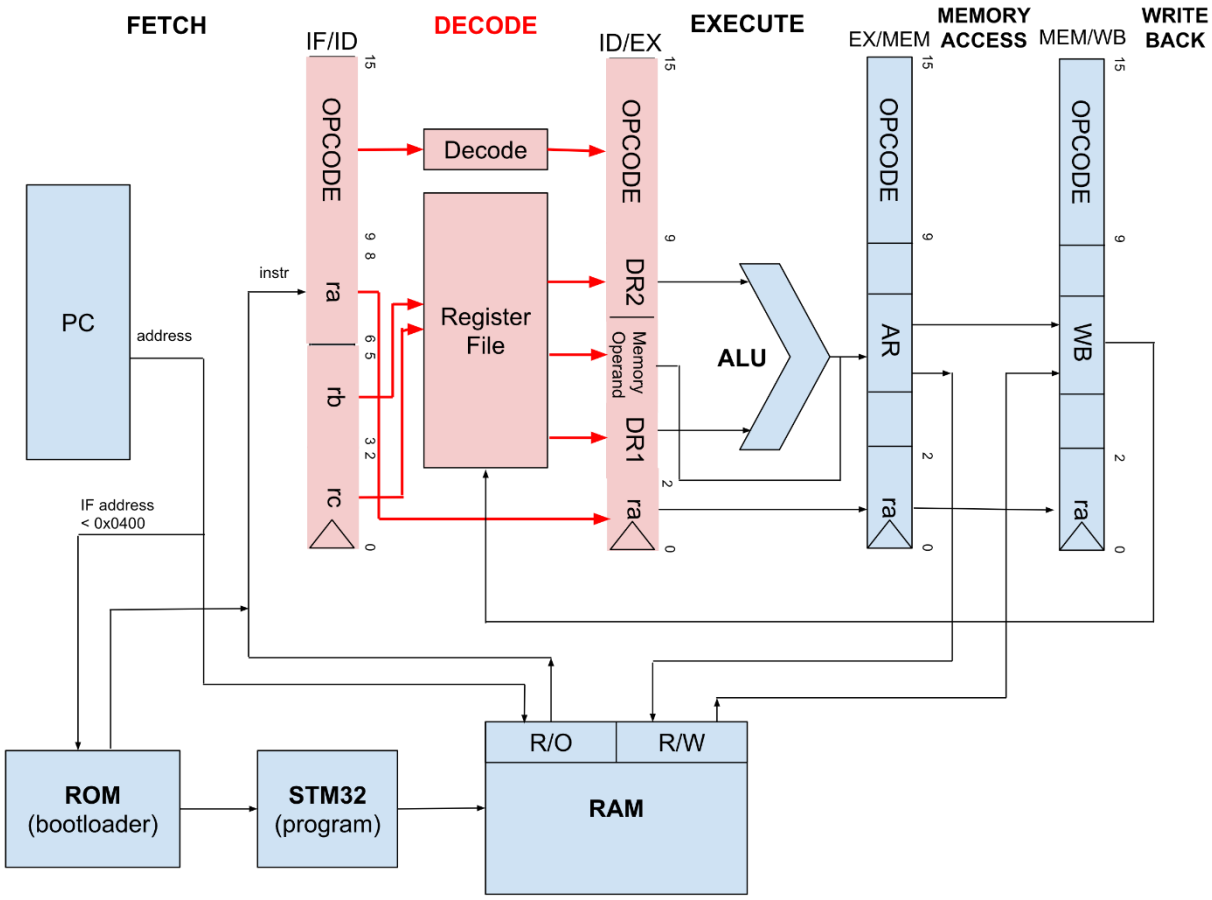

Figure 3.3: Decode Stage Diagram

Depending on the OPCODE of the instruction, certain data is retrieved from the register file based on the indexes of the instruction.

The final step of the decode stage is to send the required data to the ID/EX stage register.

3.3 Execute

The purpose of the execute stage is to perform any arithmetic or logic calculations which may be required by the instruction. The data from the decode stage is first passed to the execute stage through the ID/EX stage register, where the inputs are then passed to the ALU if needed (determined through the OPCODE). A diagram highlighting the important blocks and paths of the Execute stage can be seen in Figure 3.4.

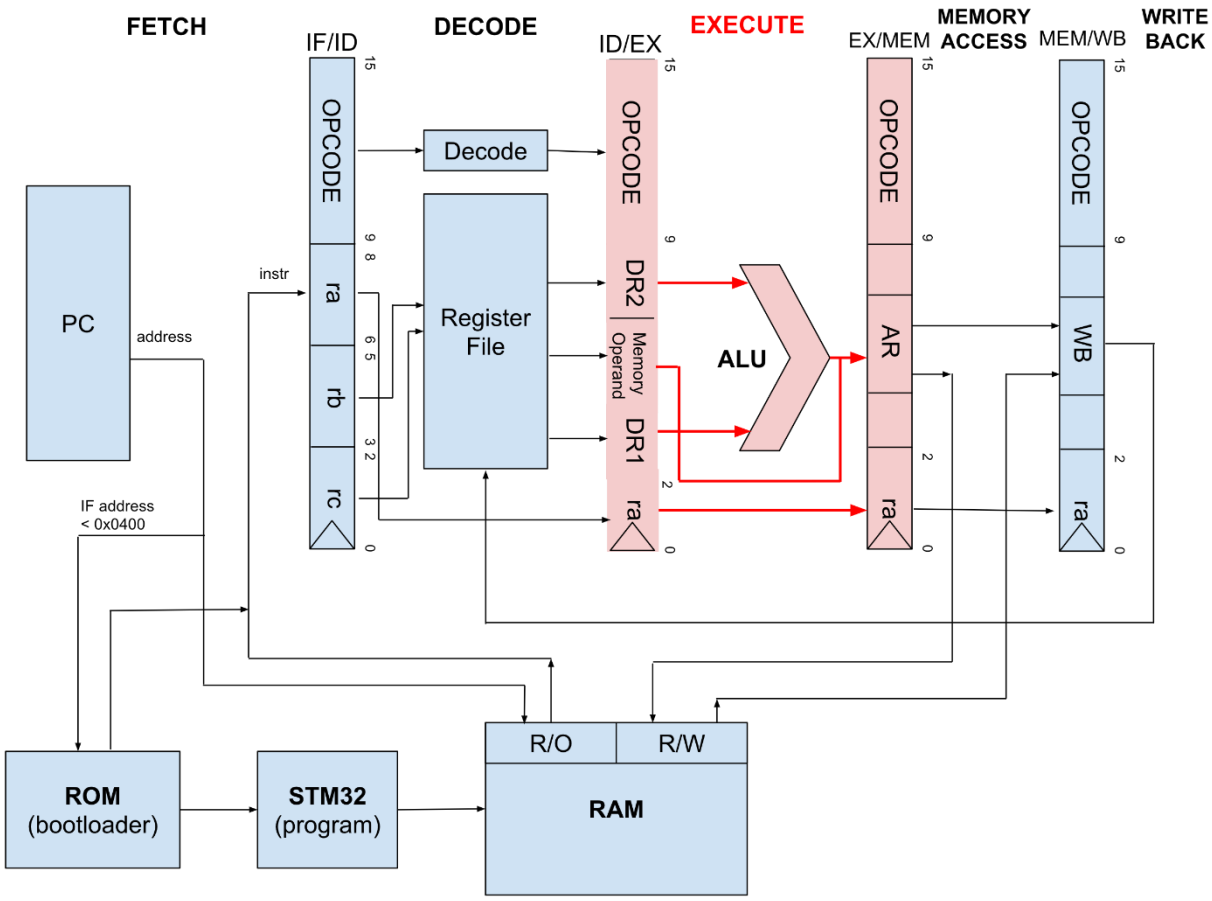

Figure 3.4: Execute Stage Diagram

For most A Format instructions, the data1 and data2 values from ID/EX are passed as aluin1 and aluin2 to the ALU component. The ALU computes the arithmetic using the inputs and outputs the result as result_in_mem (AR in diagram).

For most B Format instructions, the new values of the Program Counter are calculated and passed as pc_target_in to the next stage register EX/MEM.

For most L Format instructions, the destination or source values are determined, and the result is passed to result_in_mem.

The execute stage is also where forwarding is handled to prevent data dependencies. More information on forwarding and hazards is covered in Section 3.6.

3.4 Memory Access

The goal of the memory access stage is to access the RAM component as necessary. Depending on the type of instruction, the RAM may be written to, read from, or the data and results may just be passed to the MEM/WB stage register. Below is a diagram which shows the components used in the memory access stage (Figure 3.5)

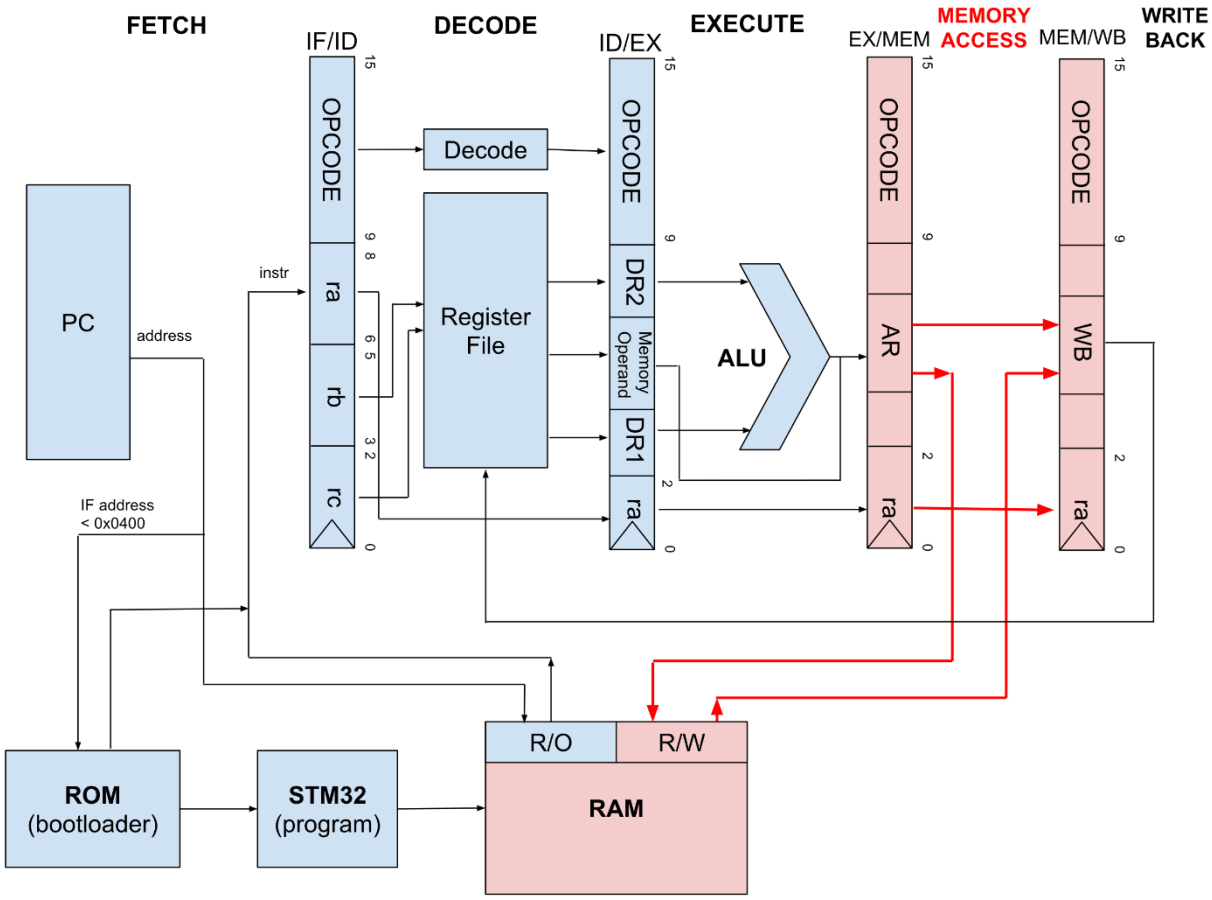

Figure 3.5: Memory Access Stage Diagram

Another goal of the memory access stage is to enable writes to the program counter, and pass the results of the B Format instructions to the addr_assign_s signal to allow for the branches to execute. This signal writes to the PC if the write_en_pc enable is active.

If the instruction is a STORE instruction, the wea_RAM is set equal to '1' which allows for writing to the RAM. If the destination of the STORE instruction is 0xFFF2, then the data will be sent to be displayed on the HEX Display. During a load instruction, only a read from the RAM is executed.

All A format instructions bypass the RAM and the result is sent straight to the MEM/WB stage register.

3.5 Writeback

The purpose of the writeback stage is to write the result of the execute and memory access stages into the register file component. Additionally, it is the point where the BR.SUB subroutine writes the appropriate address from MEM/WB to R7. A diagram highlighting the key components and paths of the writeback stage can be seen below (Figure 3.6).

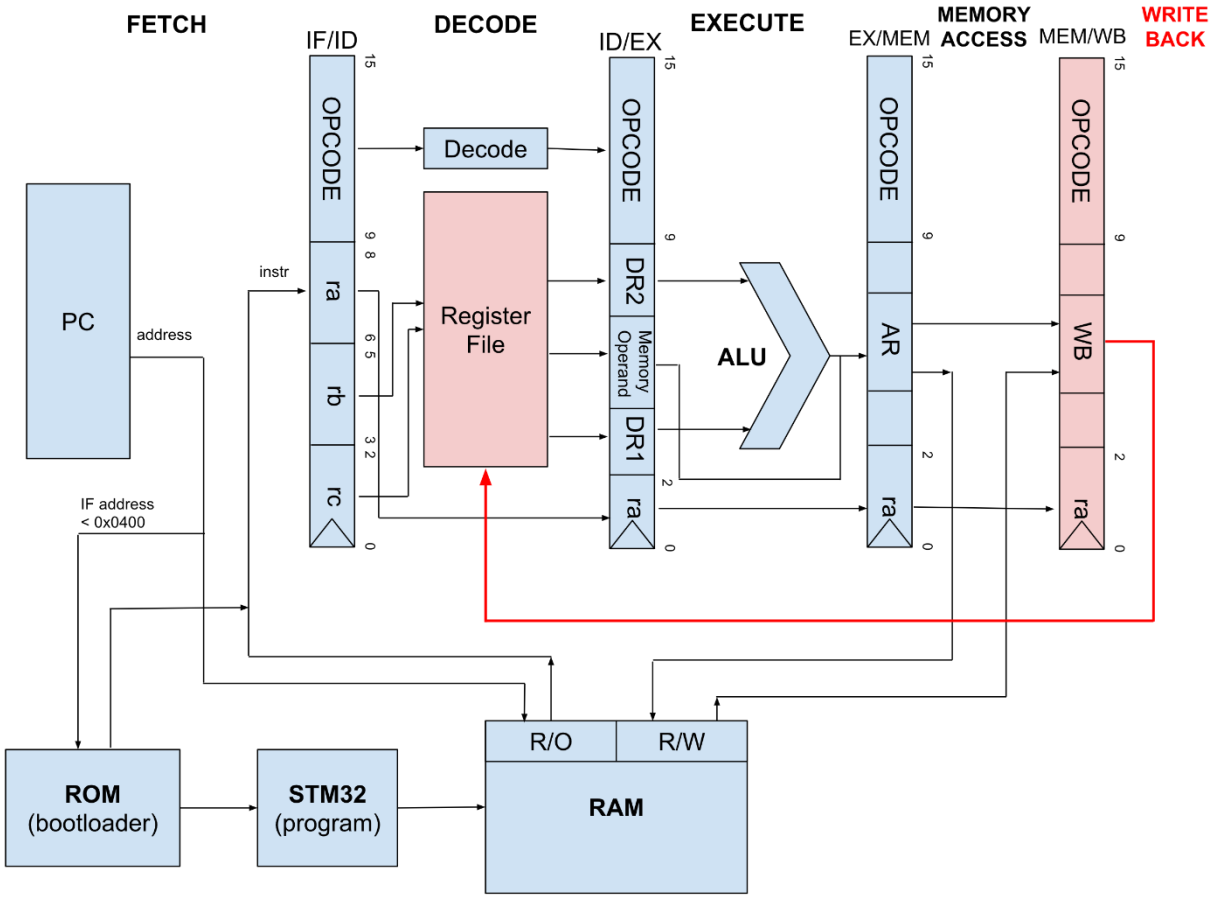

Figure 3.6: Writeback Stage Diagram

If the OPCODE corresponds to any of the following instructions, then the result_out from MEM/WB is written to the register corresponding to ra.

- A1 Format
- A2 Format
- IN
- LOAD
- LOADIMM
- MOV

3.6 Hazard Handling

In order to deal with hazards in the final design. Two main methods were utilized: forwarding, and pipeline stalls (bubbles). Forwarding was implemented strictly in the execute stage of the pipeline (see execute stage of processor.vhd for code), while branch stalling was accomplished by using a processor level branch flag.

Forwarding is a method used to prevent read-after-write (RAW) data hazards that might occur due to one instruction operating on registers that are yet to be updated to their appropriate, new value, by a previous instruction that has not completed.

For example:

ADD R1, R2, R3 (R1 initially = 10, R2 = 3, R3 = 1)
MUL R1, R1, R2

Without data handling, the ADD instruction would not be finished by the time the MUL instruction fetches R1 data, causing it to operate on an R1 value of 10 rather than 4. Forwarding can be used to efficiently (without stalls) handle RAW data hazards. With forwarding, the execute stage will perform tests on the state of the processor (EXMEM and MEMWB stage registers) in order to detect data hazards. For this example, the appropriate data (R1 = 4) will be in the memory access stage during MUL execution, causing an EXMEM data hazard. To handle this, the forwarding functionality will check the target index (R1) in the EXMEM register and compare it to both operands of the execute stage (R1 and R2). Since there is a match (R1 = R1), and the opcode of the ADD instruction in the EXMEM register also writes to the register file (identified through the opcode), the ALU will instead grab the data in the EXMEM stage rather than the IDEX stage. Therefore, data is fed back from an instruction further along in the pipeline rather than waiting for a writeback to the register file.

The branch flag is an ‘OR’ of 3 sub-branch flag signals held within the IFID, IDEX, and EXMEM stage registers. The stage registers set their branch flag to ‘1’ when the opcode input to the component corresponds to a branch instruction, and ‘0’ if it does not. If any of the 3 output signals from the stages are ‘1’, the pipelines branch flag is then set to one (OR). The fetch stage of the pipeline is the only section of the processor that is affected by this flag, if the flag is ‘1’ then the fetch will simply force NOP instructions on to the next stage, rather than reading from memory. This behavior is equivalent to a pipeline stall, allowing instructions that occurred before the branch to complete execution, but halting instructions that occur after the branch until the branch is evaluated. This method of pausing the pipeline is less efficient than a branch prediction method (as the processor is doing ‘nothing’ for 3 cycles each branch) of handling but is also much simpler as it does not require restoring the state of the processor if the branch is incorrectly evaluated.

WAW (write-after-write) and WAR (write-after-read) hazards are not significant issues with the instructions used in this processor design. WAW hazards occur when two instructions have two differing completion times and try to write to the same register, causing writebacks to occur out of order. Since the instruction implementations used in the pipeline processor have similar execution times, this hazard is mainly avoided. Likewise, for WAR hazards, as the instructions take similar times to execute and the same amount of time to write back to the register file, these hazards are avoided.

3.7 Overflow Handling

Overflow is initially set through the arithmetic instructions as they are executed by the ALU. Note that only the ADD, SUB, and MUL instructions are capable of causing data overflow, as the other operations do not affect the length of the result. Once the ALU completes its execution of the instruction, the overflow status is appended onto the most significant bit of the result, then sent forward through the pipeline for writeback. Therefore, each register in the register file has been modified to hold 17-bit data (as opposed to the default 16 bits). Note that this 17^{th} bit is only read by the TEST instruction, with all other operations simply using the lowest 16 bits. The TEST instruction has also been modified to include an overflow flag, based on the 17^{th} bit of register RA, as well as the standard Z and N flags. The test instruction then copies the overflow bit of RA into the overflow flag, which may then be used by an additional branch instruction. The branch instruction (BRR.O) operates the same way as the BRR.Z, and BRR.N instructions. If the overflow flag is set to 1, the program counter will branch to PC + 2*disp.l, while if the overflow flag is set to 0, the program counter will increment normally. In order to ensure that the BRR.O instruction is executed properly; a test instruction of the target register should be inserted before the branch instruction. Also note that the OPCODE selected for this additional instruction is 72.

3.7.1 Branch Overflow Instruction

In order to have the overflow_factorial test program function properly, a method to branch given an overflow needed to be implemented.

Handling for a new BRR.O instruction was programmed into the processor to solve this. The instruction had the OPCODE of 72, and was modelled the same way as BRR.Z and BRR.N. Instead of checking the status of the N_FLAG or Z_FLAG signals, the BRR.O instruction checks for the OVERFLOW_FLAG. The BRR.O instruction then branches if the OVERFLOW_FLAG has been set to 1. More information on the instruction can be found in Appendix A.

4 Design Components

The 5-Stage Processor contains a series of components and sub-components in its construction. The overall design comprises of 5 main components. These components are as follows:

- Processor
- RAM
- ROM
- Bootloader
- HEX Driver

The overall design takes in 6 inputs and has 1 output. The inputs and outputs are detailed in Table 4.1. The lone output signal is the least significant bit of the OUT_PORT, known as the ACK_SIGNAL. This signal is sent to pin M18 which is connected to the STM32F0. If the NEXT button had been pressed on the STM32F0 then upon receiving the ACK_SIGNAL, the clock will stop.

The IN_PORT is responsible for receiving the instructions from the STM32F0. The bootloader dictates which instructions are pulled from the STM32 into the FPGA. The IN_PORT takes in 14 bits at a time. The first 8 bits are half the data of the instruction and the remaining 6 bits are used for the handshaking protocol required for data transfer. Therefore, it takes two transmissions for a complete instruction to be inputted into the FPGA (byte-wise).

The RESETEXECUTE and RESETLOAD signals are triggered by the two pushbuttons on the Basys 3 (see Figure 4.9). Both resets are seen as the same in terms of clearing the processor, however they vector the program counter to different values. When RESETLOAD is pressed, the PC vectors to address: 0x0002. When RESETEXECUTE is pressed, the PC vectors to address 0x0000. These addresses correspond to separate BRR instructions within the bootloader. At the address 0x0002, a BRR instruction branches to a subroutine which loads instructions into memory. At address 0x0000, a BRR instruction branches to a subroutine which begins execution of the instructions currently in memory.

The input SWITCH_DATA_IN, refers to the vector which is created by toggling the 16 switches on the Basys 3 FPGA (see Figure 4.9). When a LOAD instruction of the form “LOAD r.drst 0xFFF0” is executed, the source of the LOAD will be SWITCH_DATA_IN and by extension, the switches from the FPGA.

The final two inputs of the system are CLOCK and DISPLAY_CLK. The CLOCK signal is a global clock which is used by all peripherals with the exception of DISPLAY_CLK. The CLOCK signal is retrieved from the STM32 and is controlled by the STEP, NEXT, and RESUME, buttons. The frequency of the clock is controlled by the python app on the PC. The DISPLAY_CLK input is responsible for providing a clock signal to the HEX display. The clock signal is internally produced by the Basys 3 and is divided down to 200Hz within the HEX Driver architecture. Note that an external clock may also be used instead of the clock of the STM32.

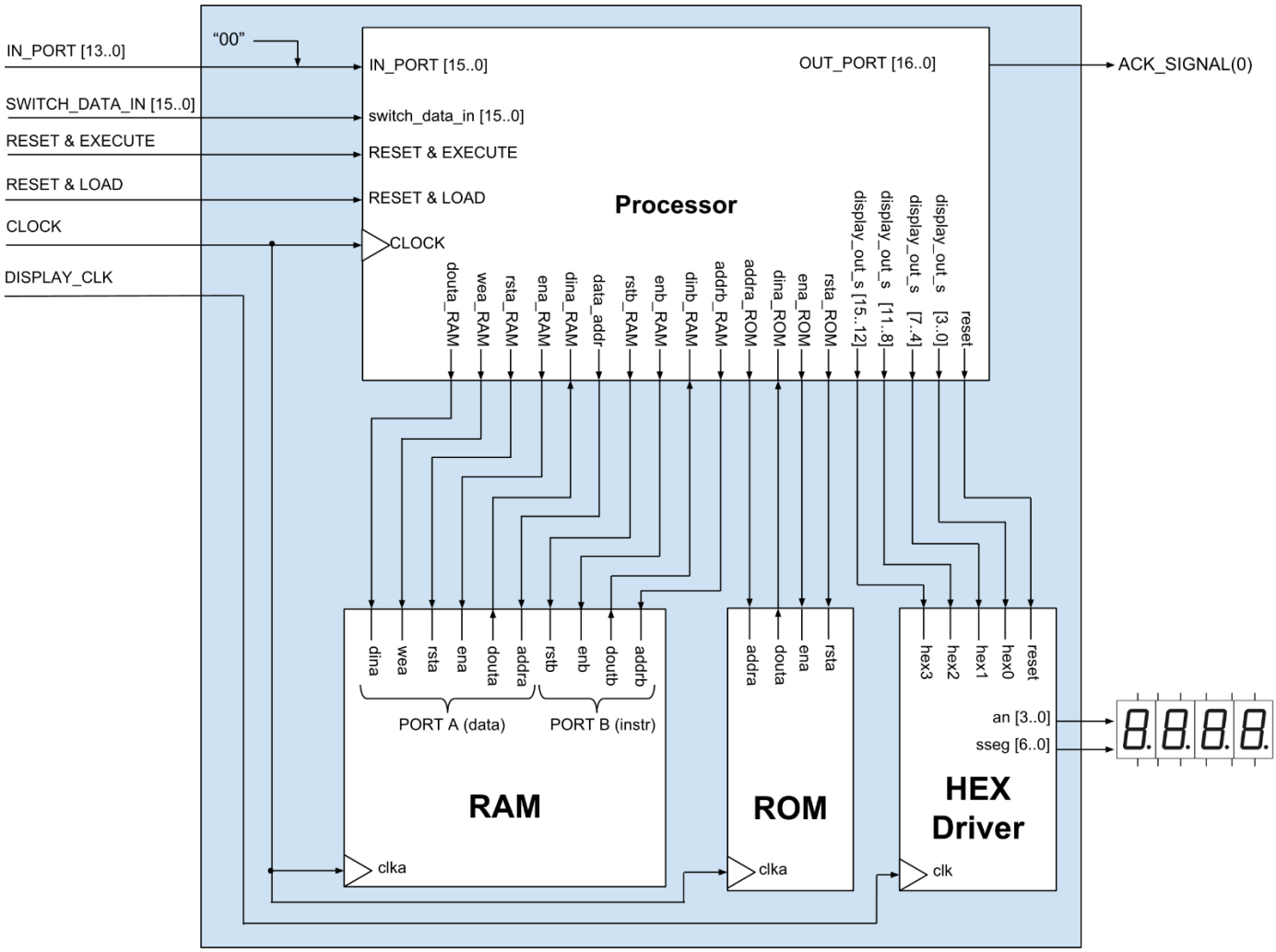

Figure 4.1: System Connection Diagram

Table 4.1: System Pin Description

Name	Mode	Type	Comments
IN_PORT	IN	STD_LOGIC_VECTOR (13 downto 0)	Instruction received from STM32F0, 2 bits are added as it enters the processor block
SWITCH_DATA_IN	IN	STD_LOGIC_VECTOR (15 downto 0)	16 bit string set by 16 switches on Basys 3
RESETEXECUTE	IN	STD_LOGIC	Sets PC to 0x0000 & executes instructions
RESETLOAD	IN	STD_LOGIC	Sets PC to 0x0002 & loads instr. to RAM
CLOCK	IN	STD_LOGIC	External clock from STM32F0. Controls all peripherals except HEX driver
DISPLAY_CLK	IN	STD_LOGIC	Internal Basys 3 clock for the HEX driver
ACK_SIGNAL	OUT	STD_LOGIC_VECTOR (0 downto 0)	LSB of OUT_PORT, signifies completion when sent to STM32F0

The following subsections will cover each of the major components listed above, as well as the subcomponents within the processor. Each component will have a pinout diagram as well as a table listing all the pins and their functions.

4.1 Bootloader

The purpose of the boot loader is to allow the remote loading of applications into the processor memory without needing to rebuild the core. This allows different test programs to be tested without having to recompile the whole project, which is time consuming.

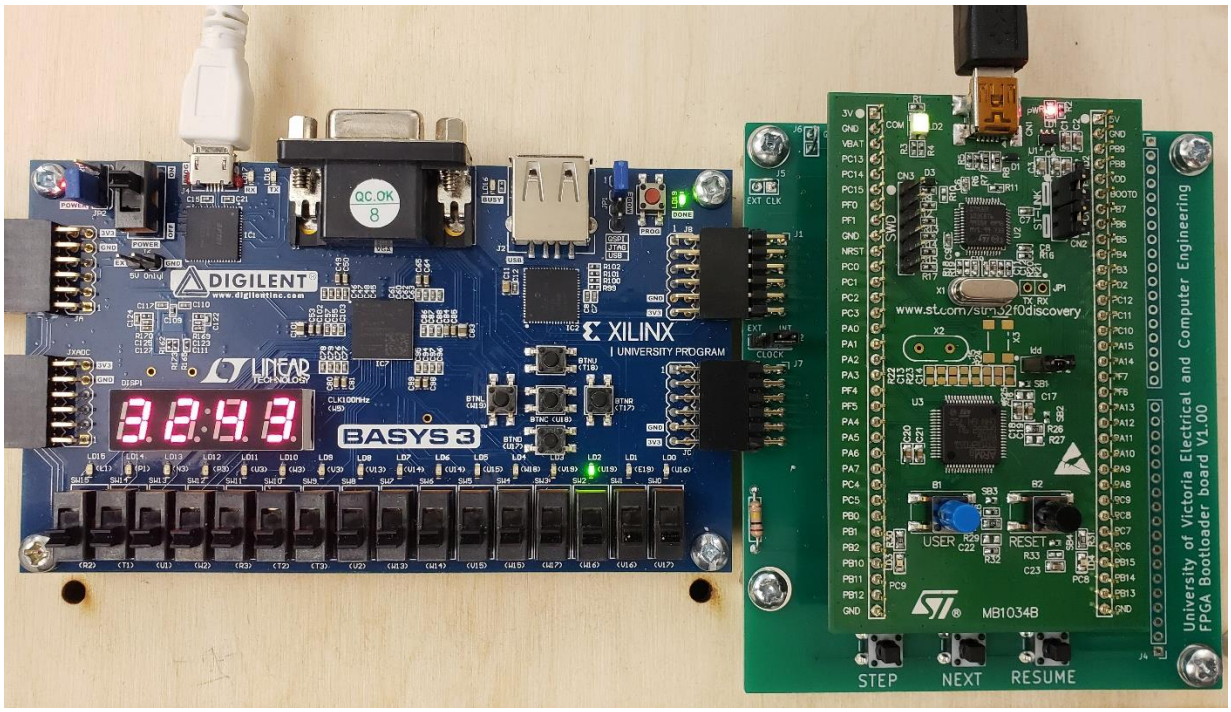

Figure 4.2: Basys 3 FPGA and STM32F0 Bootloader

The Bootloader component consists of a firmware written by Brent Sirna, which is assembled and loaded into the ROM component of the system. Upon loading the component, the ROM acts as the bootloader for the system.

Instructions are loaded onto the STM32F0 using a python applet on a PC. The role of the bootloader is to control the STM32F0 microcontroller to read and send instructions from the STM32 to the RAM component in the processor system.

When the RESETLOAD button is pressed on the Basys 3, the PC is set to address 0x0002. This triggers the loading of the code into RAM. When the RESETEXECUTE button is pressed on the Basys 3 FPGA, the PC is set to address 0x0000.

The RAM component then receives an address through Port A from the Program Counter. The instruction output from the RAM is then sent to the IF/ID stage register.

The bootloader and the STM32 allow for stepping through instructions in order to execute one instruction at a time. This is done by utilizing the STEP, NEXT, and RESUME buttons (see Figure 4.2). The button functions are as follows:

- STEP – Manual clock (push for rising edge, release for falling edge)
- NEXT – Keep clock running until ACK_SIGNAL output is received by STM32F0
- RESUME – Run clock continuously

4.2 Processor

The designed processor contains 5 stages and includes the following components: program counter, register file, ALU, and stage registers. The processor is designed to take in signals from the "IN_PORT", "RESET & EXECUTE" button, "RESET & LOAD" button, and CLOCK signal. From there, the processor is designed to output ACK_SIGNAL to the bootloader which is the least significant bit of the output. This bit acts as a handshaking protocol between the processor and the STM32F0 and bootloader.

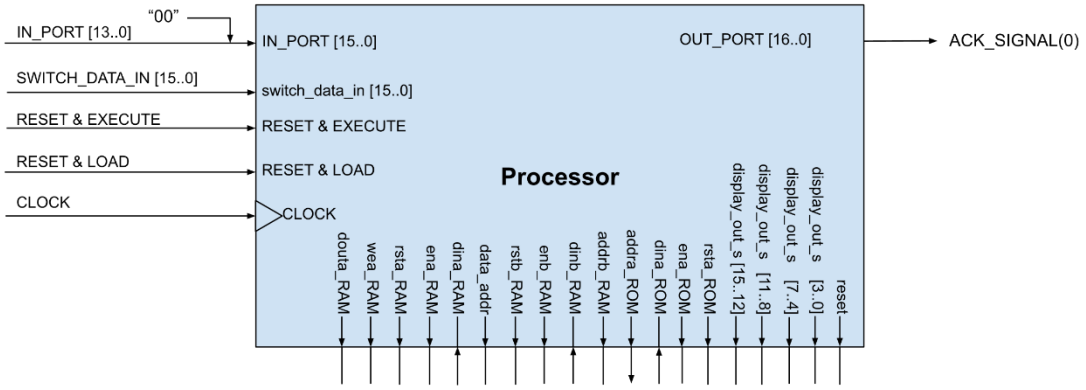

Figure 4.3: Processor Pinout Diagram [1]

4.2.1 Program Counter

The program counter is responsible for incrementing the program address by 2 every rising edge of the clock cycle. This allows the instructions from the memory to be sequentially passed onto the IF/ID stage register.

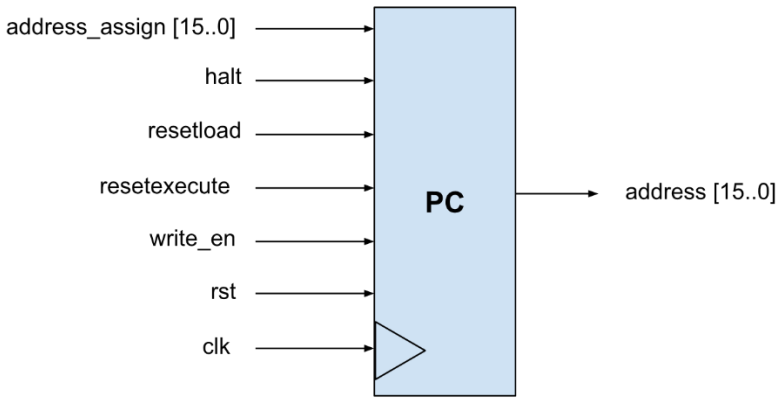

Figure 4.4: Program Counter Pinout Diagram

Table 4.2: Program Counter Pin Description

Name	Mode	Type	Comments
address_assign	IN	STD_LOGIC_VECTOR (15 downto 0)	Address to write to PC (if writing to PC)
write_en	IN	STD_LOGIC	Enable bit to enable writing to PC
halt	IN	STD_LOGIC	Halts counting of PC to avoid hazards
resetexecute	IN	STD_LOGIC	Vectors the PC to 0x0000
resetload	IN	STD_LOGIC	Vectors the PC to 0x0002
rst	IN	STD_LOGIC	Input signal from RESET signals
clk	IN	STD_LOGIC	Input signal from global CLOCK
address	OUT	STD_LOGIC_VECTOR (15 downto 0)	Output address from PC

If the write_en bit is equal to 1, then the program counter will write the value in the address_assign vector, into the address of the counter. This typically occurs when a branch instruction is executed which changes the value of the program counter.

In addition to the address being written to, in the event of a branch instruction, the halt signal will be set to 1. This will stop the counter from counting which helps eliminate some hazards.

Inside the program counter process, if a Reset & Execute or Reset & Load signal is detected, the PC will set it's address to 0x0000 and 0x0002 respectively. This is used for to allow for bootloader functionality.

4.2.2 Register File

The register file is responsible for storing the registers and reading and writing to them. The file contains eight 16-bit registers (r0…r7). The register file is a key component in both the decode, and writeback stages. Below is a pinout diagram and a pin description table.

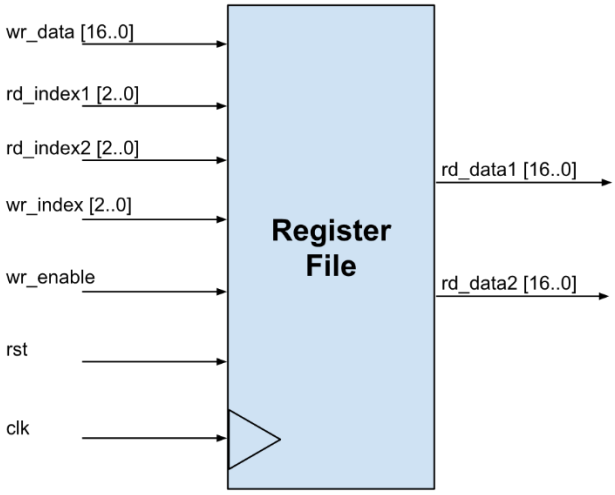

Figure 4.5: Register File Pinout

Table 4.3: Register File Pin Description

Name	Mode	Type	Comments
wr_data	IN	STD_LOGIC_VECTOR (16 downto 0)	Data to write into register corresponding to wr_index. (comes from Writeback stage)
rd_index1	IN	STD_LOGIC_VECTOR (2 downto 0)	Read index for register (Port 1)
rd_index2	IN	STD_LOGIC_VECTOR (2 downto 0)	Read index for register (Port 2)
wr_index	IN	STD_LOGIC_VECTOR (2 downto 0)	Index to write the data from Writeback into
wr_enable	IN	STD_LOGIC	Enables writing to register file
rst	IN	STD_LOGIC	Reset signal
clk	IN	STD_LOGIC	Clock input
rd_data1	OUT	STD_LOGIC_VECTOR (16 downto 0)	Operand 1 data out
rd_data2	OUT	STD_LOGIC_VECTOR (16 downto 0)	Operand 2 data out

In the decode stage, rb and rc from the incoming instruction correspond to rd_index1 and rd_index2 respectively. The indexes are passed into the register file, while the data stored at the indexes are then passed onto rd_data1 and rd_data2.

During the writeback stage the register file will write the data of wr_data to a particular register denoted by wr_index. When a rst signal is detected, the contents of the register file are emptied and set to 0.

4.2.3 ALU

The Arithmetic Logic Unit (ALU) is responsible for performing the arithmetic and logical functions which are required by the incoming instruction. The implemented ALU consists of the following components:

- Adder
- Subtractor
- Multiplier
- Left Barrel Shifter
- Right Barrel Shifter

The pinout diagram can be found below in Figure 4.3 and the pin description table is in Table 4.3.

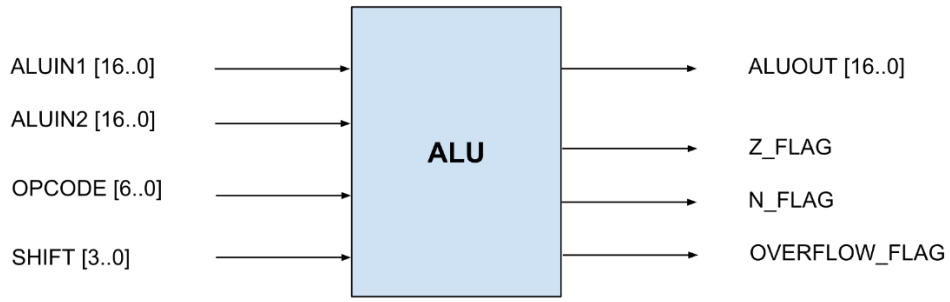

Figure 4.6: ALU Pinout Diagram

Table 4.4: ALU Pin Description

Name	Mode	Type	Comments
ALUIN1	IN	STD_LOGIC_VECTOR (16 downto 0)	Operand 1
ALUIN2	IN	STD_LOGIC_VECTOR (16 downto 0)	Operand 2
OPCODE	IN	STD_LOGIC_VECTOR (6 downto 0)	Dictates the type of instruction and the arithmetic, logic, or action to complete
SHIFT	IN	STD_LOGIC_VECTOR (3 downto 0)	Dictates the amount to shift when instruction is SHR or SHL
ALUOUT	OUT	STD_LOGIC_VECTOR (16 downto 0)	Output of ALU (passed to EX/MEM)
Z_FLAG	OUT	STD_LOGIC	Flag indicates output is zeros
N_FLAG	OUT	STD_LOGIC	Flag indicates output is negative
OVERFLOW_FLAG	OUT	STD_LOGIC	Flag indicates is ALU result has exceeded 16 bits

The ALU operates by taking in the instruction OPCODE from the ID/EX stage register and determining the type of instruction to execute. The values on ALUIN1, ALUIN2, and SHIFT are utilized differently depending on the OPCODE.

It should be noted that the number format for the ALUIN1, ALUIN2, and ALUOUT logic vectors is two's compliment along with an extra MSB which represents the overflow bit. When the value of ALUOUT exceeds the max 16 bit number, the overflow

bit is set to 1 and outputted. If the value of ALUOUT is negative, then the lower 16 bits will be in two's compliment form, while the upper overflow bit is unaffected.

Using the architecture of the ALU and the internal components listed above, the ALU is responsible for generating a result on the ALUOUT output and, in some cases, the Z, N, and OVERFLOW flags. Table 4.4 describes the instruction types that the ALU handles, the corresponding inputs, outputs, and operation.

Table 4.5: ALU Instruction Description

Instr.	Input 1	Input 2	Outputs	Operation
NOP	-	-	NULL	A NOP instruction is designed to do nothing
ADD	ALUIN1	ALUIN2	ALUOUT	Adds two inputs (IN1 + IN2 = OUT)
SUB	ALUIN1	ALUIN2	ALUOUT	Subtracts inputs (IN1 – IN2 = OUT)
MUL	ALUIN1	ALUIN2	ALUOUT	Multiplies inputs (IN1 * IN2 = OUT)
SHL	ALUIN1	SHIFT	ALUOUT	Shifts IN1 left by SHIFT places
SHR	ALUIN1	SHIFT	ALUOUT	Shifts IN2 right by SHIFT places
NAND	ALUIN1	ALUIN2	ALUOUT	The inputs IN1 and IN2 are NAND'ed together
TEST	ALUIN1	-	Z_FLAG N_FLAG OVERFLOW_FLAG	If IN1 is negative, Z_FLAG = 1, else Z_FLAG = 0 If IN1 is zeros , N_FLAG = 1, else N_FLAG = 0 If IN1 overflowed, OVERFLOW_FLAG = 1, else = 0

4.2.3.1 Barrel Shifters

In order to execute the logical shift, two-barrel shifters (left shift and right shift) are implemented. The barrel shifters each input 2 operands, the amount to shift (4 bits), and the operand. The operand then goes through 4 sequential shifts dependent on the value of the shift input. Bit 0 (LSB) of the shift input will shift the operand by 1 bit if the shift bit is 1, or 0 if the shift bit is 0. Bit 1 of the shift input will shift by 2 bits, bit 2 of the shift input will shift by 4 bits, and bit 3 (MSB) of the shift input will shift by 8 bits. This method allows for the operand to be shifted either left or right by anywhere from 0-15 bits. Note the operand is shifted, not rotated, and therefore the lost bits will be replaced by '0'. VHDL implementation of the barrel shifters can be seen in ALU.vhd code section document.

4.2.3.2 Multiplier

To execute the MUL instructions a signed 16-bit multiplier was implemented, along with an overflow functionality. Inputs of the multiplier (operands rb and rc) are assumed to be 2's complement, as well as the output (ra). First, the inputs are converted from 2's complement to a signed-magnitude format, with the MSB representing the sign, and the other 15 bits representing the magnitude (unsigned) of the data. First, a product variable is initialized to 0. The two magnitudes are then multiplied with each other using a shift and sum algorithm. This is an iterative algorithm in which a single bit of rc is multiplied by rb (using an AND operation), resulting in a partial product which is added onto the 'product' variable. Next iteration, the same process is repeated using the next bit of rc, rb is also shifted left

once per iteration in order to assure proper magnitude of the partial products. After completion (15 iterations), the 'product' will then hold the resultant product magnitude (30 bits). Overflow can simply be checked by looking at the 15 MSB of the product variable, if any of the bits are '1' an overflow flag is set and output to the processor by the ALU. The 15 lowest bits of the product variable are then appended with the sign bit (sign ra = sign rb XOR sign rc) and output to the ALU.

The outputs of the ALU are sent directly to the next stage register EX/MEM where they are processed. The flags from the TEST instruction are passed off the main processor. They are read later when a branch instruction like BR.Z or BRR.N requires them.

4.3 ROM (XPM_MEMORY_SPROM)

The ROM was implemented using the structure and code of the single port XPM_MEMORY_SPROM component in the Xilinx Architecture Libraries. The size of the ROM is 1024 byte.

The read-only memory (ROM) component of the system has had two different roles over the course of the project. In early versions of the project, the ROM was responsible for storing the instructions entirely, while in the final design the ROM was strictly used for the bootloader. The ROM would send instructions corresponding to the address received by the program counter (addra), to the IF/ID stage register. The VHDL code for the ROM can be found in Appendix B. A pinout of the ROM component can be seen in Figure 4.4.

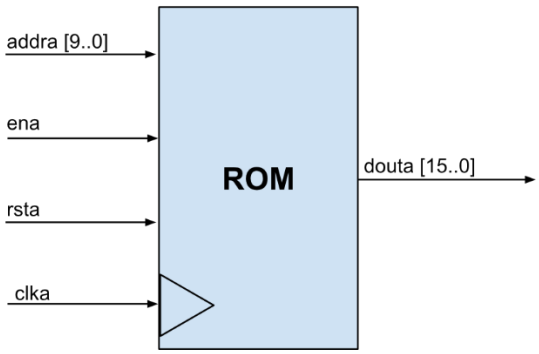

Figure 4.7: ROM Pinout Diagram

Table 4.6: ROM Pin Description

Name	Mode	Type	Comments
addra	IN	STD_LOGIC_VECTOR (8 downto 0)	Input address
ena	IN	STD_LOGIC	Enable bit to enable ROM
rsta	IN	STD_LOGIC	Input signal from RESET signals
clka	IN	STD_LOGIC	Input signal from global CLOCK
douta	OUT	STD_LOGIC_VECTOR (15 downto 0)	Output data from ROM

To load a program into the ROM, the instructions must first be assembled using the assember19.py program provided by Brent Sirna. Upon assembly, the assembled hex values are coded into the memory_init.mem file in Vivado. Any hex instruction inside this file will be loaded into the ROM.

During the Fetch stage of the process, the ROM is read from when the address sent from the program counter are less than 0x0400. From here, the ROM sends the corresponding instruction to the Decode stage of the pipeline. If the address from the program counter is above 0x0400 then the RAM is accessed instead.

In the latest versions of the code, the ROM is responsible for storing the compiled bootloader firmware (see Section 4.1). Source code for the bootloader can be found on the ECE 449 Lab Website [3]

4.4 RAM (XPM_MEMORY_DPDISTRAM)

The main memory of the system is the RAM component. The RAM is based on the XPM_MEMORY_DPDISTRAM component in the Xilinx Architecture Libraries. The RAM is a dual port module (emulating Harvard Architecture) and has a size of 1024-bytes. In the final version of the system, the RAM is responsible for storing the instructions sent by the STM32 (program), sending the instructions to the decode stage, and storing data (during loads and stores).

When the program counter outputs an address which is greater than 0x0400, then the address is sent to the RAM where the corresponding instruction is read and transmitted to the IF/ID stage register. The RAM is also involved in the memory access stage where data is read and written to the RAM (depending on the incoming instruction). More on this can be seen in Section 3.4.

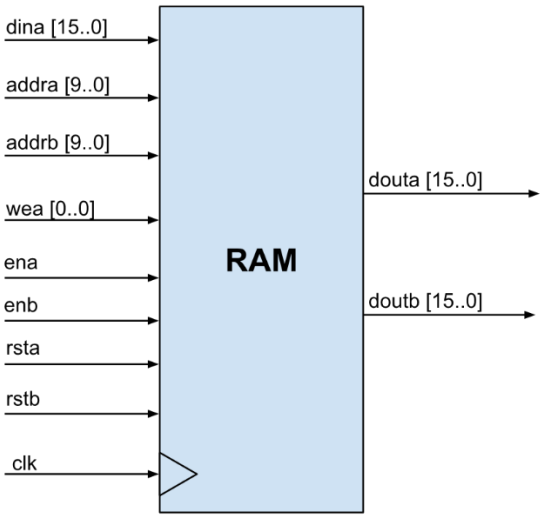

Figure 4.8: RAM Pinout Diagram

Table 4.7: RAM Pin Description

Name	Mode	Type	Comments
dina	IN	STD_LOGIC_VECTOR (15 downto 0)	RAM input
addra	IN	STD_LOGIC_VECTOR (8 downto 0)	Port A address input
addrb	IN	STD_LOGIC_VECTOR (8 downto 0)	Port B address input
wea	IN	STD_LOGIC_VECTOR (0 downto 0)	Write enable input
ena	IN	STD_LOGIC	Enable bit for Port A
enb	IN	STD_LOGIC	Enable bit for Port B
rsta	IN	STD_LOGIC	Input reset signal for Port A
rstb	IN	STD_LOGIC	Input reset signal for Port B
clk	IN	STD_LOGIC	Input signal from global CLOCK
douta	OUT	STD_LOGIC_VECTOR (15 downto 0)	Port A output from RAM
doutb	OUT	STD_LOGIC_VECTOR (15 downto 0)	Port B output from RAM

4.5 HEX Driver

When the instruction "STORE 0xFFF2 r.src" is decoded, the data in the r.src register is to be displayed on the 7-segment display. To accomplish this, a HEX driver component is used (pre-provided). To convert the bits to hex values and select the segments to light on the display. A pinout diagram can be seen in Figure 4.8 and a pin description table can be found in Table 4.8.

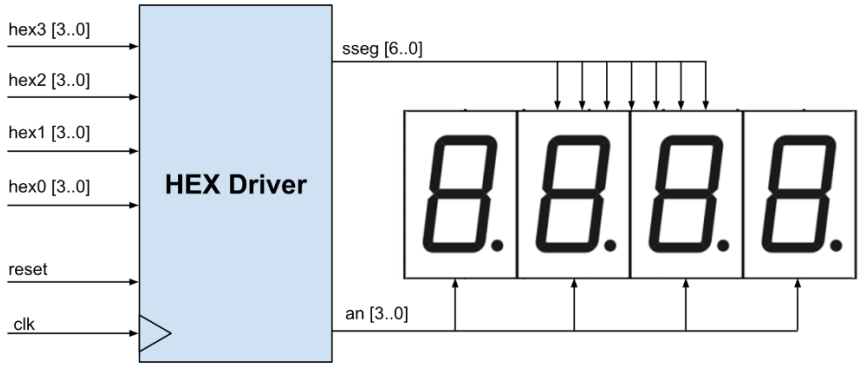

Figure 4.9: HEX Display Pinout Diagram

Table 4.8: HEX Display Pin Description

Name	Mode	Type	Comments
hex3	IN	STD_LOGIC_VECTOR (3 downto 0)	display_out [15..12] from processor
hex2	IN	STD_LOGIC_VECTOR (3 downto 0)	display_out [11..8] from processor
hex1	IN	STD_LOGIC_VECTOR (3 downto 0)	display_out [7..4] from processor
hex0	IN	STD_LOGIC_VECTOR (3 downto 0)	display_out [3..0] from processor
reset	IN	STD_LOGIC	Input signal from RESET signals
clk	IN	STD_LOGIC	DISPLAY_CLK input. Divided down to 200Hz in display_controller entity
an	OUT	STD_LOGIC_VECTOR (3 downto 0)	Selects which digit to light on 7-seg display
sseg	OUT	STD_LOGIC_VECTOR (6 downto 0)	Selects segments to light on 7-seg digit

When the STORE 0xFFF2 instruction is detected in the memory access stage, the source register is sent to the “display_out” signal. The “display_out” signal is divided into four 4-bit vectors, which represent the 4 hex digits to display (as seen in Table 4.8).

The clock input is provided by the DISPLAY_CLK signal, and is divided down in the HEX driver to 200Hz. This is to allow the 7-seg to properly display the digits as each digit is strobed sequentially. The entire string is strobed at a frequency of 200Hz, therefore each digit updates at 50Hz. DISPLAY_CLK is obtained internally on the Basys 3 board, unlike the CLOCK signal which comes from the STM32F0.

The segments are chosen based on case statements inside the component. The case statements are shown below.

```
with hex select
       sseg(6 downto 0) <=
              "1000000" when "0000", -- 0
              "1111001" when "0001", -- 1
              "0100100" when "0010", -- 2
              "0110000" when "0011", -- 3
              "0011001" when "0100", -- 4
              "0010010" when "0101", -- 5
              "0000010" when "0110", -- 6
              "1111000" when "0111", -- 7
              "0000000" when "1000", -- 8
              "0010000" when "1001", -- 9
              "0001000" when "1010", -- A, which signifies "10"
              "0000011" when "1011", -- B, which signifies "11"
              "1000110" when "1100", -- C, which signifies "12"
              "0100001" when "1101", -- D, which signifies "13"
              "0000110" when "1110", -- E, which signifies "14"
              "0001110" when others; -- F, which signifies "15"
```

Once the segments are decided, the variable “an” is sent along with “sseg” as outputs. The signal “an” dictates which digit will be lit, while the signal “sseg” dictates which segments will be lit to in order to display the digit.

4.6 Basys 3 Board Inputs

The Basys 3 FPGA contains 5 buttons and 16 switches as part of it’s input mechanisms. For the processor project, two buttons and 16 switches are used. Below is a diagram of the FPGA which shows the key components of the board (Figure 4.9).

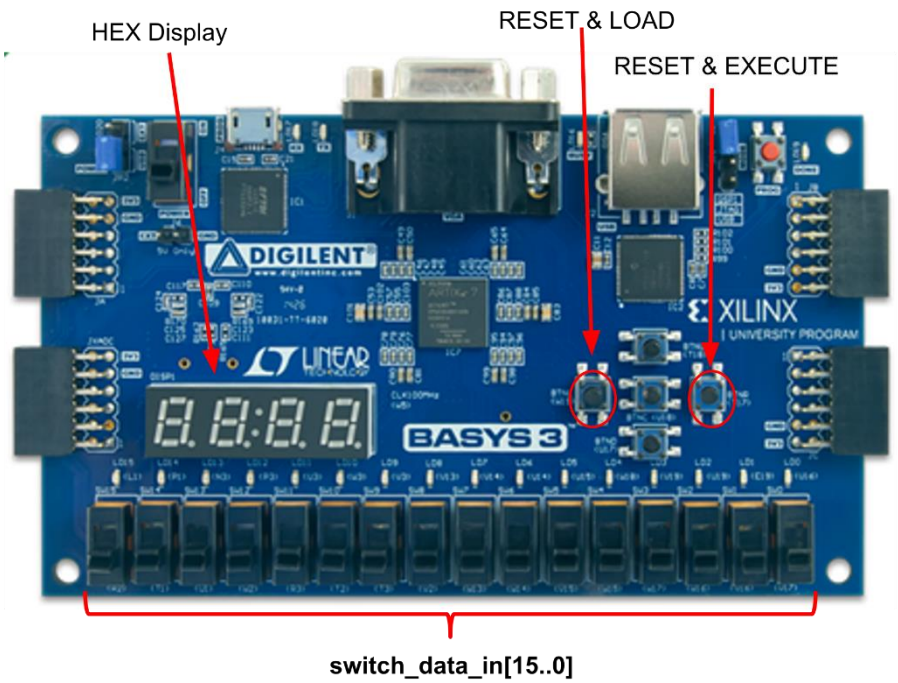

Figure 4.10: Basys 3 Peripheral Diagram

5 Results

5.1 Timing Tests

To verify the timing abilities of the system, tests were performed on the processor to verify the highest clock rate which could be applied. Using the function generator, an external clock signal was connected to the EXT_CLK pin on the STM32F4. The jumper on the bootloader PCB was then moved and connected to the EXT CLK pins.

The factorial.asm code was loaded onto the bootloader for testing. The accuracy of the factorial result, and correct overall operation was observed. After each successful trial, the clock frequency was increased. The processor was able to reach a frequency of 50MHz before it started to miss inputs upon the Reset & Execute button being pressed.

Table 5.1: Clock Frequency Test Result

Max Clock Frequency
50MHz

5.2 Hardware Utilization

Upon implementation generation a hardware utilization map was generated. See Figure 5.1 and Figure 5.2 for the circuit of the ALU and utilization map respectively. The highlighted blocks in 5.2 are utilized blocks for the processor. Only the ALU is included for the circuit as the entire system is too large to be meaningful in image format.

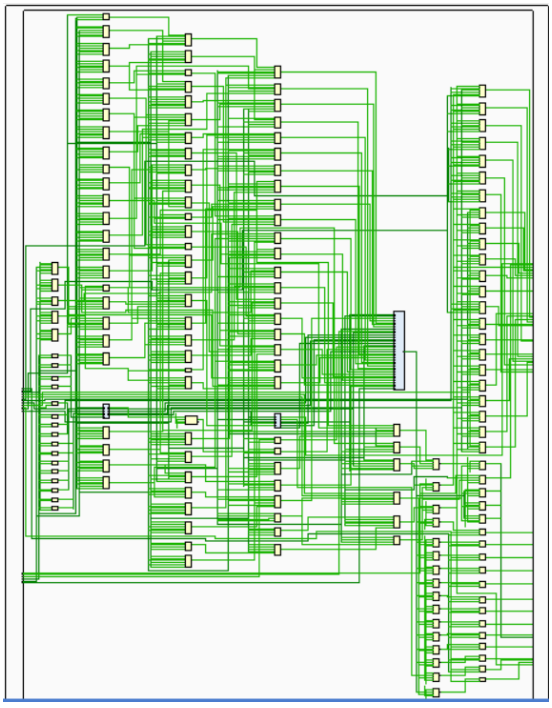

Figure 5.1: ALU Circuit Implementation

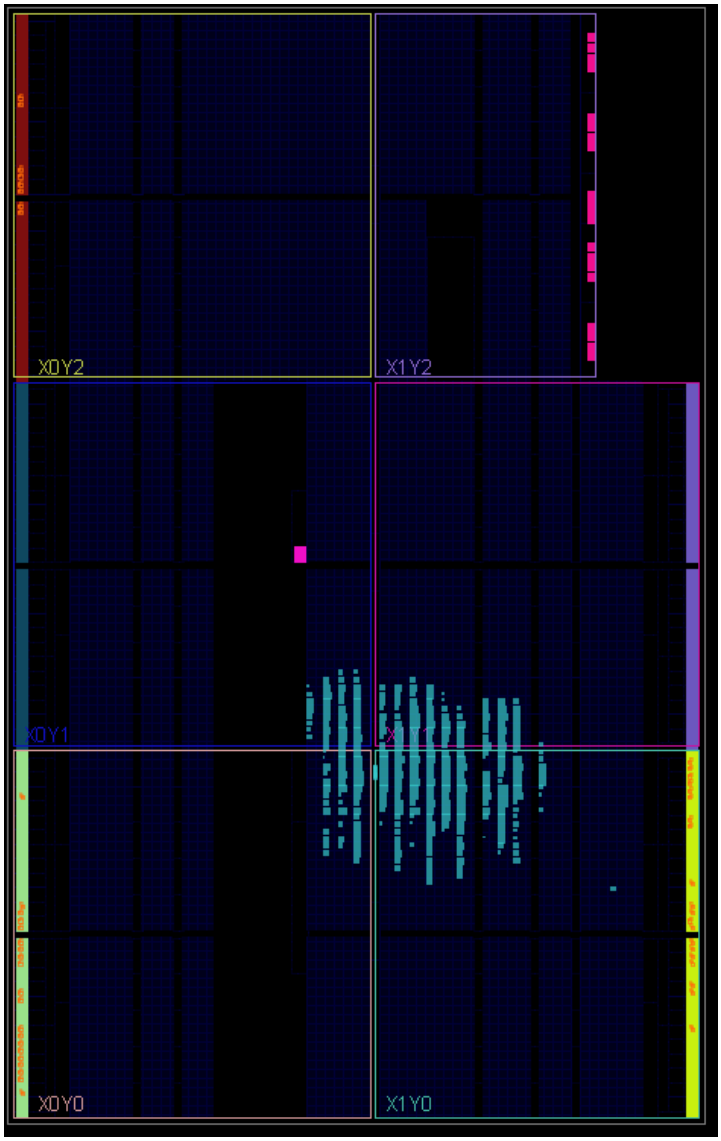

Figure 5.2: Implementation Map of Processor

5.3 Final Test Code Results

The final two test programs consisted of the instructions described in Appendix C. The first test program was designed to test the general functionality of a factorial calculation. The second test was designed to test the system's overflow method using a multiplication algorithm (overflow results in 0 being displayed). Both test codes were assembled and loaded onto the bootloader for execution.

In order to achieve functionality with the factorial overflow test, a new instruction BRR.O needed to be created. After successfully implementing the new instruction, the processor was able to output "0000" onto the HEX display once an overflow was detected.

Both codes were successfully tested and run on the Basys 3 board. The functionality was demonstrated to the instructor and TA.

5.4 Multiplier Speed Test

Using the multiplier speed test program provided by Brent Sirna, the performance of the multiplication algorithm was tested. 32 multiplication instructions were passed through the algorithm and the results were compared to the expected results. If the multiplication result matched the expected result, a pulse would be produced. The time between the clock pulse and the comparison pulse was measured to receive the multiplication time.

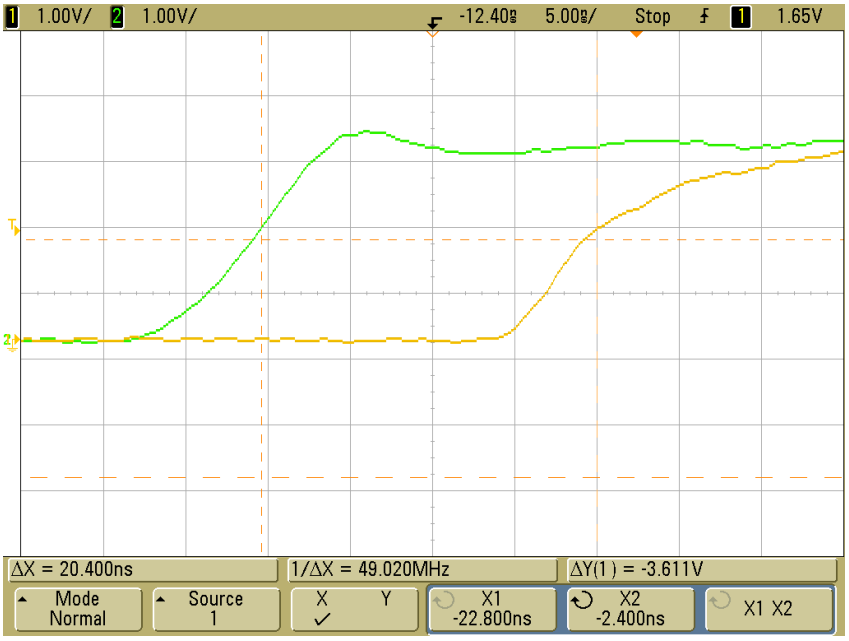

Figure 5.3: Multiplier Test Waveforms

The results were compared against the control algorithm, implemented by simply using the '*' operator to perform multiplication (optimized through the ISE). The results of the test are shown below in Table 5.2.

Table 5.2: Multiplication Test Results

Brent Sirna's Code	Design Team's Code
17.40ns	20.40ns

As can be seen, the two times were relatively close to one another. The time results for the multiplication algorithm are therefore considered a success.

6 Discussion

The processor design covered in this report completed our main objectives of running all the required instructions within the provided instruction set (A, B, and L format). As required, the processor is also capable of inputting data from the switches using the load instruction, and outputting data to the hex display using the store instructions. However, the final processor design also has a few limitations in its operation. The first, and probably most apparent limitation, is the use of branch stalling. Upon receiving any sort of branch (including non-conditional branch instructions such as BR or BRR), the processor will enter a stalled state for 3 clock cycles while the branch is evaluated. During the stalled state, the processor will not be able to handle any further instructions, therefore slowing down program execution time (and efficiency). With a different design, this limitation could've possibly been reduced to a 2 clock cycle stall, improving execution time by 1 cycle per branch. This possibility was not explored very in depth due to time constraints. Another alternative to branch stalling is by use of branch prediction, by assuming that the branch is or is not taken upon decode of the instruction, execution of the program would speed up. However, consideration of branch mispredicts must be handled with this design option, increasing complexity, and reducing efficiency upon misprediction.

Another limitation of the processor is by implementation of the 16-bit multiplier component within the ALU. This multiplier is based off a shift and sum algorithm, where one bit of operand RC is ANDed (multiplied) with each bit of operand RB in parallel. The partial products are then added to themselves after each calculation. RB is then shifted one bit to the left (with the MSB scrapped) each iteration in order to hold the proper magnitude of the product. The final product is then checked for overflow by checking if any of the 15 MSB are 1, if so then the overflow flag is output from the multiplier block to the ALU. This multiplication algorithm is inherently slow as it is completed entirely (except for the bitwise AND stage) sequentially. The multiplication design could be improved to allow the partial products to be calculated entirely in parallel, while the summation can be calculated logarithmically, resulting in faster execution time.

This design is also limited to processing only A, B, and L format instructions. The optional instructions (push, pop, load, rti) were not implemented into this design, but could possibly be added in a later revision with some modification.

7 Conclusions

The objective of the project was to design a 5-stage processor in VHDL which is to be programmed to the Basys 3 FPGA. The processor was required to be able to decode and execute instructions from the A, B, and L formats, as well as input data from 16 switches and output results to the HEX display. Finally, the FPGA was to be able to connect with an STM32F0 loader which loads instructions to the memory of the processor system.

The final implementation of the processor was able to successfully decode each of the individual instruction format tests as well as successfully run the final test code provided by the instructor.

Finally, the multiplication algorithm was successfully tested and measured. The resulting computation time of the multiplier was 20.8ns which was considered close to the control case of 17.4ns.

Once again it is important to note that the logic for comparing the program counter address to decide whether to read from either ROM or RAM, is programmed into the processor itself (fetch stage). This causes problems for compatibility of the CPU with other sizes of RAM and ROM. It is recommended that in the next revision, the logic should be moved to the top-level instead of inside the processor component. This will allow for different sizes of ROM and RAM to be used without compatibility issues.

From this project, the fundamentals and process behind creating VHDL projects were successfully communicated. The participants in the project were able to take away a greater understanding of processors and their internal workings and challenges. Additionally, the process behind testing and implementing VHDL code onto an FPGA was successfully communicated.

Overall, the final version of the project was a success at completing the requirements laid out at the beginning of the project, in addition to providing a wealth of knowledge to the project participants.

References

[1] N. Dimopolous, "ECE 449 Project Introductory Slides", *ECE 449 Project Website*, 2019. [Online]. Available: https://www.ece.uvic.ca/~ece449/lab/index.html. [Accessed: 01- Apr- 2019].

[2] B. Users, "Basys 3 Artix-7 FPGA Trainer Board: Recommended for Introductory Users", *Digilent*, 2019. [Online]. Available: https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/. [Accessed: 01- Apr- 2019].

[3] B. Sirna, "Bootloader Source Code", *ECE 449 Lab Website*, 2019. [Online]. Available: https://www.ece.uvic.ca/~ece449/lab/index.html. [Accessed: 01- Apr- 2019].

[4] "Central processing unit", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Central_processing_unit. [Accessed: 05- Apr- 2019].

[5] "Reduced instruction set computer", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer. [Accessed: 05- Apr- 2019].

Appendix A – Table of Instructions

Instruction	OPCODE	Format	Usage	Comment
NOP	0000000	A0	NOP	Blank instruction (do nothing)
ADD	0000001	A1	ADD ra,rb,rc	Add rb and rc and store result into ra
SUB	0000010	A1	SUB ra,rb,rc	Subtract rc from rb, store result in ra
MUL	0000011	A1	MUL ra,rb,rc	Multiply rb and rc, store result into ra
NAND	0000100	A1	NAND ra,rb,rc	NAND rb and rb, store result into ra
SHL	0000101	A2	SHL ra #n	Logical shift ra left by #n
SHR	0000110	A2	SHR ra #n	Logical shift ra right by #n
TEST	0000111	A3	TEST ra	Test ra if negative, zero, or overflow
OUT	0100000	A3	OUT ra	Send ra to OUT port
IN	0100001	A3	IN ra	Send ra to IN port
BRR	1000000	B1	BRR disp.l	Add 2*disp.l to Program Counter (PC)
BRR.N	1000001	B1	BRR.N disp.l	If N=1 then add 2*disp.l to PC If N=0 then add 2 to PC
BRR.Z	1000010	B1	BRR.Z disp.l	If Z=1 then add 2*disp.l to PC If Z=0 then add 2 to PC
BR	1000011	B2	BR ra disp.s	PC = ra + 2*disp.s (non-relative)
BR.N	1000100	B2	BR.N ra disp.s	If N=1 then PC = ra + 2*disp.s If N=0 then PC = PC + 2
BR.Z	1000101	B2	BR.Z ra disp.s	If Z=1 then PC = ra + 2*disp.s If Z=0 then PC = PC + 2
BR.SUB	1000110	B2	BR.SUB ra disp.s	Store PC+2 into r7; PC = ra + 2*disp.s
RETURN	1000111	A0	RETURN	PC = r7
BRR.O	1001000	B1	BRR.O disp.l	If Overflow = 1 then PC = ra +2*disp.s If Overflow = 0 then PC = PC + 2
LOAD	0010000	L2	LOAD r.drst @r.src	Load memory at the address (r.src) into destination register r.drst
STORE	0010001	L2	STORE @r.dest, r.src	Story contents of register r.src into the Memory at address (r.drst)
LOADIMM	0010010	L1	LOADIMM.upper #n LOADIMM.lower #n	Load #n into bits <15..8> of r7 Load #n into bits <7…0> of r7
MOV	0010011	L2	MOV dest,src	Move contents of r.src into r.dest

Appendix B – Circuit Diagram

The following schematic is the system connection diagram for the entire processor unit. It is crucial to mention that the components responsible for deciding whether to read from ROM or RAM (based on address from PC) are internal to the processor. Should a new RAM or ROM component be implemented, the processor will be unable to be compatible and must be updated. This should be fixed in the next revision.

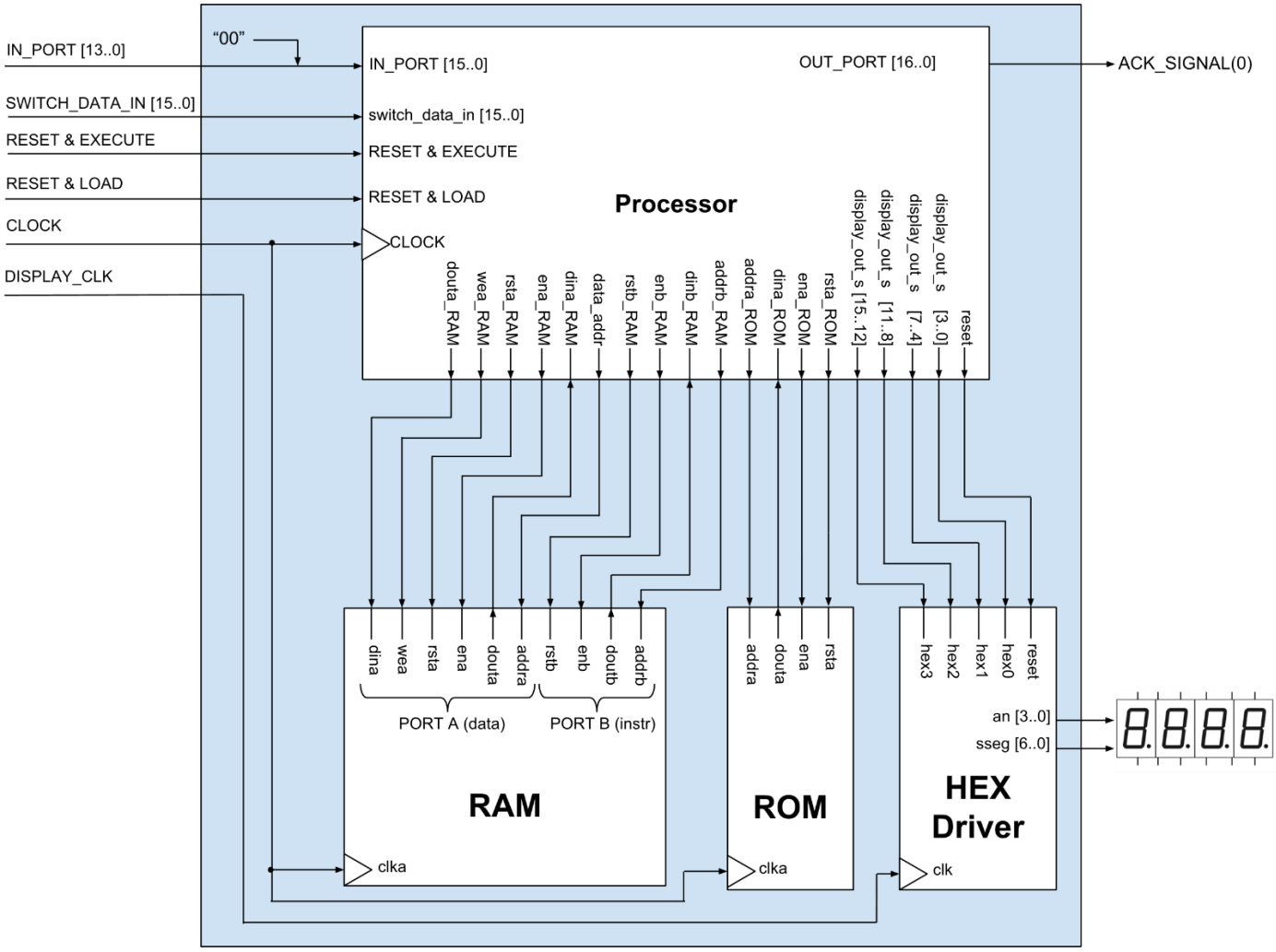

Figure B.1: System Connection Diagram

Appendix C – Final Test Code

This appendix contains the two test codes which were given to test the processor.

TEST 1 – COUNTER

```
LedDisplay:     equ             0xFFF2
DipSwitches:    equ             0xFFF0

;.DATA
;.CODE
                org             0x456
start:          loadimm.upper   LedDisplay.hi
                loadimm.lower   LedDisplay.lo
                mov             r5, r7
                loadimm.upper   0x00
                loadimm.lower   0x00
                mov             r6, r7
loop:           nop
                nop
                nop
                store           r5, r6
                loadimm.upper   0x00
                loadimm.lower   0x01
                add             r6, r6, r7

                loadimm.lower   0x0F
                nand            r4, r6, r7
                nand            r4, r4, r4
                loadimm.lower   0x0a
                sub             r4, r4, r7
                test            r4
                brr.z           ones_zero
                brr             loop

ones_zero:      loadimm.upper   0xff
                loadimm.lower   0xf0
                nand            r6, r6, r7
                nand            r6, r6, r6

                loadimm.upper   0x00
                loadimm.lower   0x10
                add             r6, r6, r7

                loadimm.upper   0x00
                loadimm.lower   0xf0
                nand            r4, r6, r7
                nand            r4, r4, r4

                loadimm.upper   0x00
                loadimm.lower   0xA0
                sub             r4, r4, r7
                test            r4
                brr.z           tens_zero
                brr             loop

tens_zero:      loadimm.upper   0x0F
                loadimm.lower   0x00
                nand            r6, r6, r7
                nand            r6, r6, r6
```

```
                loadimm.upper    0x01
                loadimm.lower    0x00
                add              r6, r6, r7

                loadimm.upper    0x0A
                sub              r4, r6, r7
                test             r4
                brr.z            start
                brr              loop
```

TEST 2 – FACTORIAL OVERFLOW

```
LedDisplay:         equ          0xFFF2
DipSwitches:        equ          0xFFF0
DipSwitchMask:      equ          0xFF                ; Binary multiple as a mask


                    ORG          0x466
main:               LOADIMM.UPPER DipSwitches.hi
                    LOADIMM.LOWER DipSwitches.lo
                    LOAD         r6, r7
                    LOADIMM.UPPER DipSwitchMask.hi
                    LOADIMM.LOWER DipSwitchMask.lo
                    NAND         r6, r6, r7    ; Nand the switch settings
                    NAND         r6, r6, r6    ; one's compliment result to AND it
                    LOADIMM.UPPER 0x00
                    LOADIMM.LOWER 0x01
                    MOV          r4, r7
                    MOV          r3, r7
                    TEST         r6
                    BRR.Z        DONE
                    SUB          r6, r6, r3    ; 0 and 1 factorial should return 1
                    TEST         r6
                    BRR.Z        DONE
                    LOADIMM.UPPER 0x00
                    LOADIMM.LOWER 0x02
                    MOV          r5, r7
LOOP:               MUL          r4, r4, r5
                    TEST         r4
                    BRR.Z        OVERFLOW
                    ADD          r5, r5, r3
                    SUB          r6, r6, r3
                    TEST         r6
                    BRR.Z        DONE
                    BRR          LOOP
OVERFLOW:           LOADIMM.UPPER 0x00
                    LOADIMM.LOWER 0x00
                    MOV          r4, r7
                    BRR          DONE
DONE:               LOADIMM.UPPER LedDisplay.hi
                    LOADIMM.LOWER LedDisplay.lo
                    STORE        r7, r4
                    BRR          DONE
                    END
```

Appendix D – VHDL Code

Please see emailed PDF file for VHDL code.