University of Victoria

Department of Electrical and Computer Engineering

ECE 449 - Computer Systems Architecture

Project Report

April 12, 2021

Calder Staude	V00870522
Isaac Morton	V00878076
Solomon Lindsay	V00873054

Abstract

The ECE 449 processor design project involved creating a 16-bit CPU that features eight sixteen bit general purpose registers and 16 byte addressable memory. The processor has been designed around a provided ISA that features RISC like architecture. Given the parameters the team designed and implemented a 16 bit processor.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviated	**Unabbreviated**
CPU	Central Processing Unit
VHDL	VHSIC Hardware Description Language
FPGA	Field Programmable Gate Array
RISC	Reduced Instruction Set Computer
ISA	Instruction Set Architecture
LED	Light Emitting Diode
ALU	Arithmetic Logic Unit

1.0 Introduction

The project involved is to design a 16-bit CPU in VHDL on an FPGA board. This requires the design and programming of the instruction set architecture including pipelining. The CPU will use a RISC-like architecture with hybrid 1-word aligned instructions in 3 formats: A-format arithmetic instructions, B-format branch instructions, and L format load instructions. The processor features eight sixteen bit general purpose registers and 16 byte addressable memory with a total capacity of 65,536 bytes of memory.

1.1 Objective

The project's objective was to create a 16 bit pipelined processor designed from the provided RISC-like ISA. It has been specified that the processor must feature 8 16 bit registers and utilize dual channel memory. The design is to be implemented on the basys 3 board. The bays 3 board's seven segment LED display is to be configured to display data that is being stored in memory. By viewing the processor's entries into memory, one may see that the processor is correctly executing a provided program.

3.0 Processor Design

3.1 Format A

The A format instructions are 2 bytes long with a 7 bit Op-code in the 7 most significant bits of the 16-bit instruction as shown in Figure 1. All of the processor's instructions are retrieved from the processor's dual-channel memory and are decoded in the Instructions Fetch and Decode latch. The instructions are received and decoded within the latch. The latch then outputs control signals that are sent to the instruction Decode and Execute latch and ALU to execute the instruction.

Format A0 is used for an instruction that does not perform an operation. This instruction is used to stall the processor to handle hazards and is further discussed in section 3.4 of this report.

The A1 format is used for the processor's addition, subtraction, multiplication, and NAND instructions. Following the A1 format instructions being decoded in the Instructions Fetch and Decode latch, the register file is sent to the addresses of the registers that are involved in the operation. The register file then retrieves the data at the provided addresses and passes the data on to the next latch. The operation of the register file is outlined further in section 3.2.1 of the report. In the Instruction Decode and Execute Latch the register's contents are passed to two multiplexers for ALU input A and ALU input B. These multiplexers determine whether to send the contents of the registers or other data used for branching. For format A instructions the multiplexers will always choose to send the data from register files through to the ALU and appear invisible to the processor's signal flow. The ALU is also sent the ALU mode signals which is a signal used by the ALU to determine what operation to perform. The ALU mode signal is provided by the processor's decoder and is passed through the Instruction Decode Execute latch onto the ALU. The details of how the ALU operates is outlined in section 3.1.1 of the report. After

the ALU performs an arithmetic or logical operation, the ALU's result will be passed on and interpreted by the instruction Execute and Memory latch. Once the latch receives the ALU result, it will check the register wite flag called reg_wr_en. If the flag is raised then the latch will pass the signal on to the Memory and Write Back latch along with the register address indicating the location to write to. The Memory and Write Back latch will then receive these signals and write the alu result into the register file storing it for a future use.

Format A2 is processed in the same fashion as format A1. The instructions are decoded and control signals as passed to the Decode and Execute latch and ALU. A case statement has been incorporated into the ALU to shift the bits by C1. While this method requires a large amount of space on the chip, it allows the ALU to perform the operation quickly. The A3 format is used to perform Test, In, and Out instructions as shown in Table 1. The Test instruction is used to test a 16-bit value to determine if it is positive or negative and if it is zero using the ALU's flags. The ALU's condition flags are outlined further in section 3.1.1 of this report. The In and Out are used to put values into and take out of memory. These instructions are implemented by having the instructions decoded in the instruction Fetch and Decode latch. This latch then sends control signals to the register file for In and Out instructions to input or take out data of memory. The Test instruction is performed by sending the ALU the ALU mode indicating it should perform a test on its inputs. The register file is given the address of the instruction to get the ALU Input A value. This input value and the ALU mode are passed through the instruction Decode Fetch latch to the ALU to perform the operation.

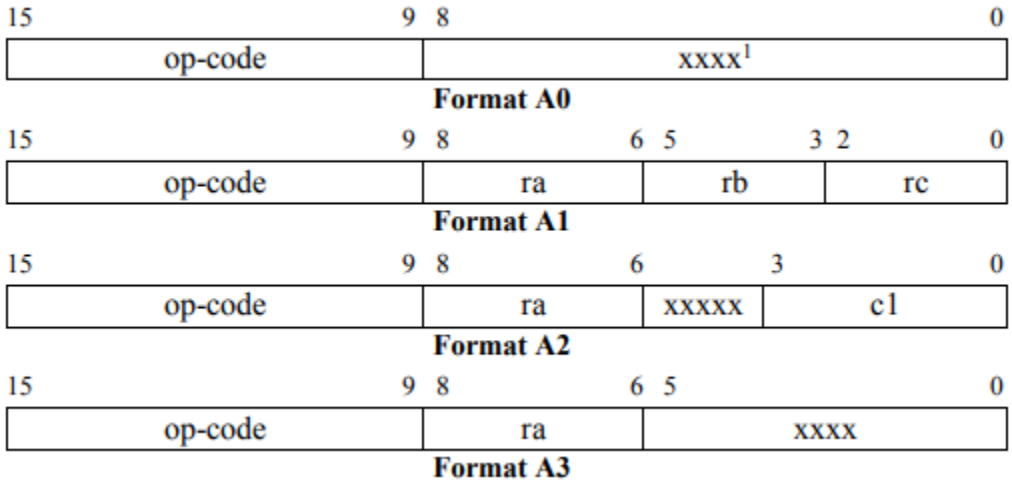

Figure 1: Format A Instruction Format

Table 1: A Format Instructions

Mne-monic	Op-code	Function	Type	Syntax
NOP	0	Nothing	A0	*NOP*
ADD	1	R[ra] ← R[rb] + R[rc];	A1	*ADD ra,rb,rc*
SUB	2	R[ra] ← R[rb] – R[rc];	A1	*SUB ra,rb,rc*
MUL	3	R[ra] ← R[rb] × R[rc];	A1	*MUL ra,rb,rc*
NAND	4	R[ra] ← R[ra] NAND R[rb]:	A1	*NAND ra,rb,rc*
SHL	5	n:=c1<3…0>): (n|0) → (R[ra]<15…0>←R[ra]<15-n…0>#(n@0)): (n=0) →() ;	A2	*SHL ra#n*
SHR	6	n:=c1<3…0>): (n|0) → (R[ra]<15…0>←(n@0)#R[ra]<15…n>): (n=0) →() ;	A2	*SHR ra#n*
TEST	7	(R[ra) = 0) →Z ← 1; else →Z ← 0: (R[ra] < 0) →N ← 1; else →N ← 0;	A3	*TEST ra*
OUT	32	OUT.PORT ← R[ra];	A3	*OUT ra*
IN	33	R[ra] ← IN.PORT;	A3	*IN ra*

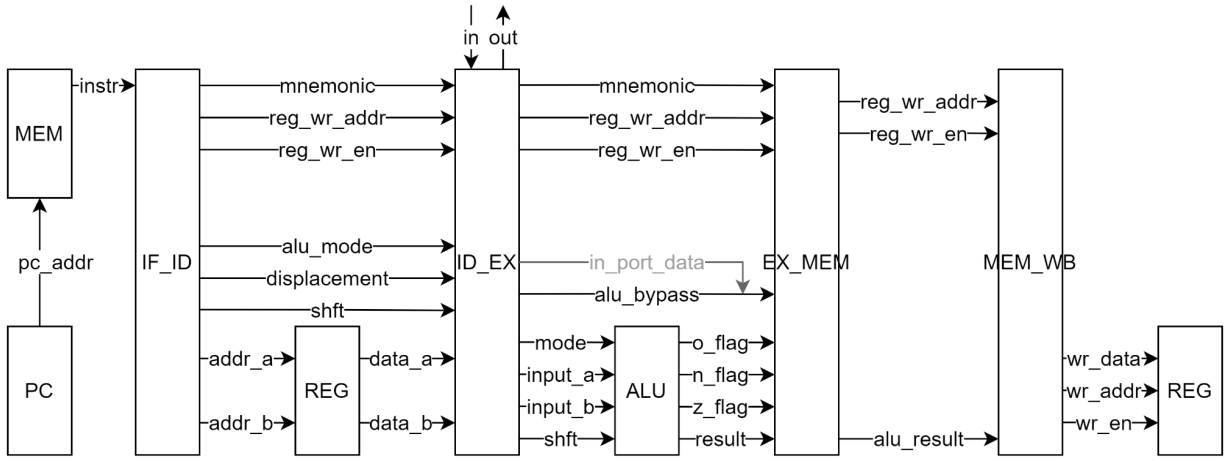

Figure 2: Format A Block Diagram

3.1.1 ALU

The ALU unit is responsible for performing arithmetic and logic operations. These operations include NOP, Add, Subtract, Multiply, NAND, Shift left logical, shift right logical, and Test. The ALU executes these operations by receiving an ALU mode that is a unique three-bit code. A case statement is used to handle this code and operate the ALU. For example, if the ALU mode is 001 this will indicate that the ALU should perform an add operation and the case for add will be executed. Table 3 shows the operations that correspond to each ALU Mode.

NOP instructions are used to stall the pipeline by informing each part of the processor to not operate for this instruction. The ALU handles a NOP by sending input A through to the result and ignoring Input B. The ALU utilizes the IEEE arithmetic library to perform add, subtract, multiply, and NAND operations. The SHL and SHR operations are handled with nested case statements. If the ALU mode receives an ALU mode of 101 or 110 as outlined in Table 1 the ALU will take the input A value and shift it by the value in input B. The shifting operations are executed by utilizing one of sixteen shifters

that shift a fixed amount. The TEST instruction is used to test an ALU input to determine if it is zero or negative or for overflow. The ALU tests for zero by comparing the ALU input A value to zero. If the two are equal the ALU will raise its zero flag that will stay raised until another test instruction is sent to the ALU. The ALU tests for a negative value by comparing the ALU input A to be less than zero. If this is true the ALU will raise its zero flag. These flags are passed to the Instruction Execute and Memory latch. If this latch sees that a test instruction is being executed, it will check the ALU flags received from the ALU operation and determine if it should raise its branch taken flag. Further details on the handling of the ALU flags and branching operations are outlined in section 3.2 of the report.

When the ALU performs ADD, SUB, MUL, and SHL operations, there is a chance that an overflow will occur. The ALU overflow handling works by first performing the ALU operation and storing the result as a 32-bit value. As the ALU inputs are both 16 bits, thus any arithmetic or logical operation will not exceed 32 bits. This 32-bit value is then checked to determine if it is either greater than 32767 or less than -32768 as these are the maximum values of a 16-bit two's complement signed number. If this condition is true the overflow flag is raised and the output of the ALU is set to zero.

Two instructions have been added to the provided ISA that allows a program to branch based on overflow. The details on these instructions are shown in Table 2. The instruction may be used by a programmer to branch on overflow. These additional instructions allow the programmer to have more tools when creating his or her program.

Table 2: Overflow Branch Instructions

Mnemonic	Op-code	Function	Type	Syntax
BRR.O	63	(O=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (O=0) → PC ← PC+2 { 2's complement};	B1	BRR.O +disp.l
BR.O	72	(O=1) → PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement}; (O=0) → PC ← PC+2 { 2's complement};	B1	BR.O ra+disp.s

Table 3 ALU Modes

ALU Mode in binary	Operation
000	NOP
001	ADD
010	SUB
011	MUL
100	NAND
101	SHL
110	SHR
111	TEST

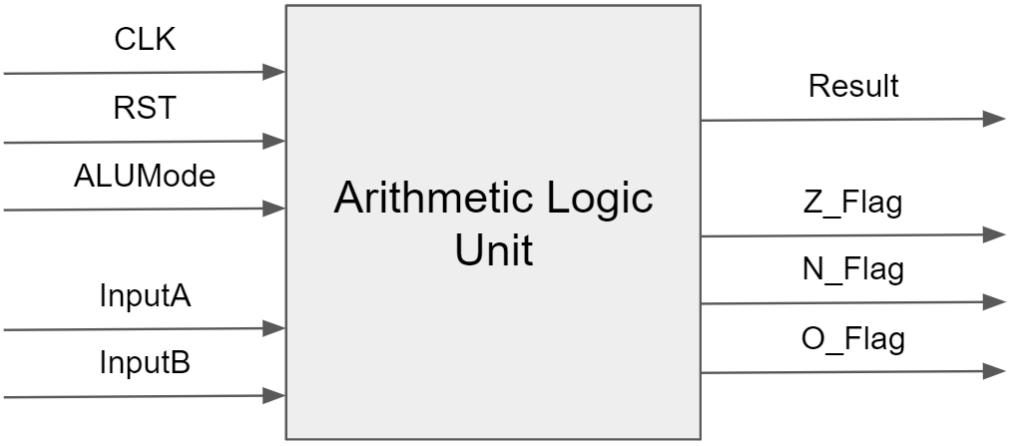

Figure 3: ALU

3.1.2 Register File

The register file reads and writes to the registers 8 16-bit registers. The register file can read 2 registers asynchronously and write to one register on the falling edge of the clock signal to allow reading and a result in the same clock cycle. To perform arithmetic operations, the processor's decoder will decode the intrusion and pass the address to the registers that hold data that is required for the instruction to execute. These signals are sent as input through the rd_index1 and rd_index2 and the data is outputted using the rd_data1 and rd_data2 signals as shown in Figure 4. To write the register file the write enable flag, wr_en, must be set to high. The register file also requires the index of the address to write to and the data to write into it. This is provided by the wr_index and wr_data signals. The register file is positioned between the Instruction Fetch and Decode latch and the Instruction Decode and Execute latch. This placement allows the processor to retrieve data from the register file at the beginning of the pipeline allowing the subsequent stages of the pipeline to act off of the data retrieved from the register file.

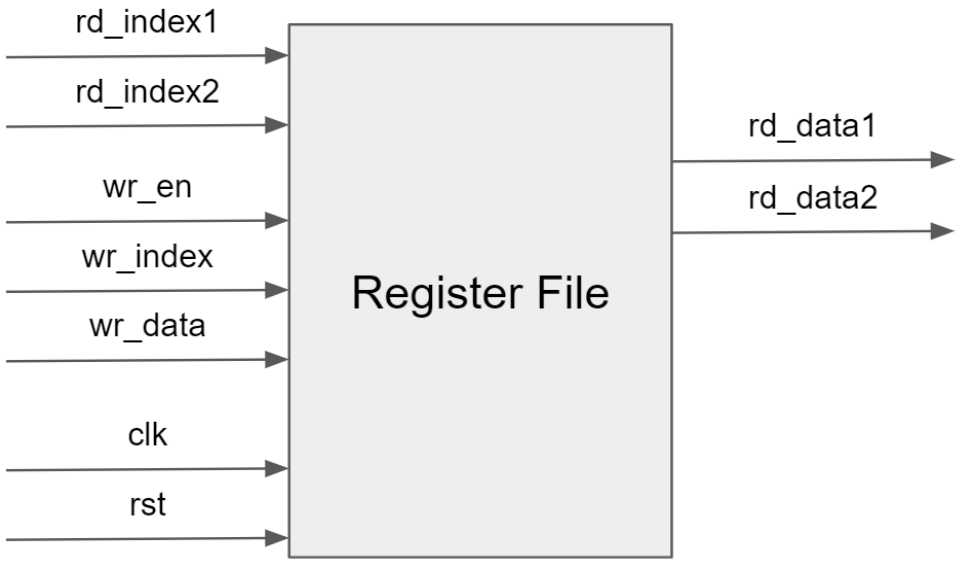

Figure 4: Register File

3.2 Format B

Format B instruction handles branch operations within the processor. There are two sub-formats, B1 and B2. B1 instructions contain a 9bit displacement, meaning it can be used to displace up to 256 instructions forward or backward . Since the displacement size is prioritized within the instruction, no register can be specified; B1 instructions will always branch relative to the program counter, hence they are known as branch relative instructions. B2 format instructions use a smaller displacement field, so they can only branch a maximum of 32 instructions forward or backward, however they also include a target register that will be used in the branching calculation. These are known as branch absolute as the destination is controlled by the contents of a register, not where the PC is at that point in code. The return instruction is a format A0 instruction as it does not contain any data, but it does still perform a branch. For a return, the contents of R7 are used as the destination address of the branch. The return instruction operates in tandem with the subroutine instruction that places the value of PC+2 into R7 before it branches.

Branch instructions are decoded in the IF_ID latch logic where the displacement is calculated and multiplied by two. The multiplication by two is done since each instruction is two bytes wide. The target address is fetched from the register file for branch absolute and return instruction. The ALU is then used to perform the addition required for the destination address calculation. For branch relative instruction the ALU adds the displacement to the program counter, for branch absolute instruction the ALU adds the displacement to the data from the register file. For conditional branch instructions, the appropriate flag is checked inside the EX_MEM latch logic to determine whether the branch should be taken or not. If the branch is taken, the branch taken signal is asserted and the destination address is sent to the program counter module. The program counter module, during regular operation, increments it's output address every clock cycle by two to go to the next instruction. If a branch is occurring, the program counter outputs the destination address and continues normal operation.

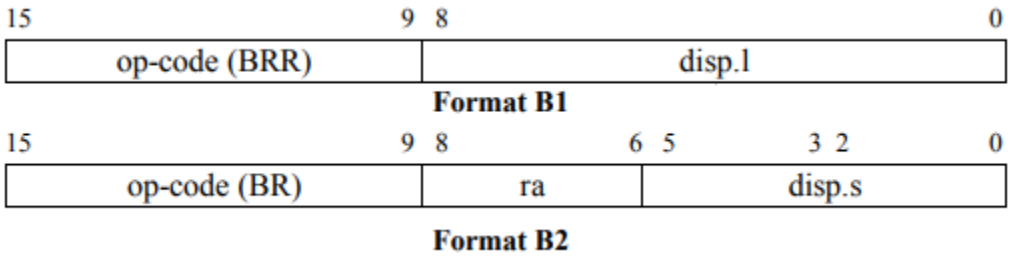

Figure 5: Format B Instruction Format

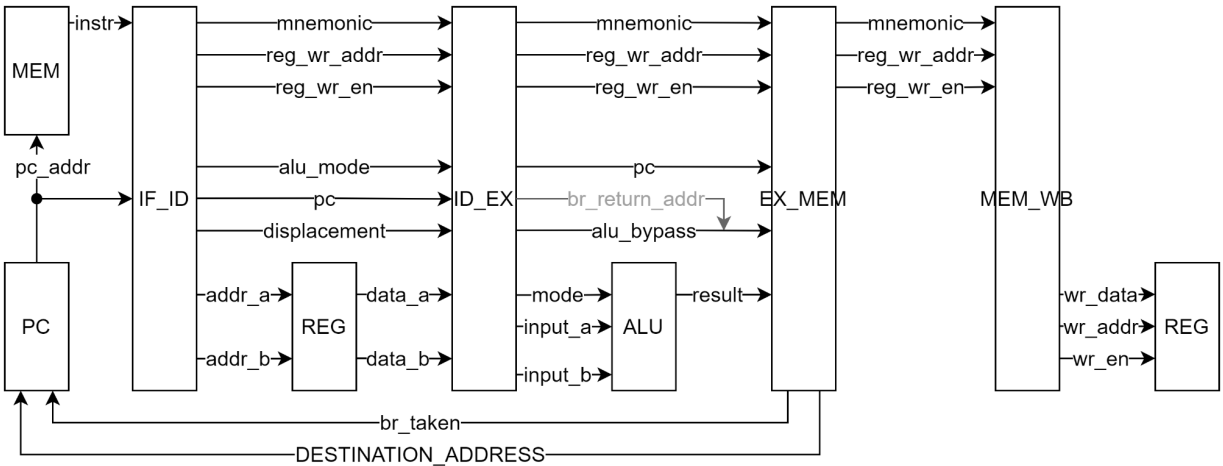

Figure 6: Format B Block Diagram

3.3 Format L

Format L instructions handle the movement of data into, out of, and between memory and the register file. There are four instructions described below in Table 4 following the two formats described in Figure 7.

LOAD and STORE instructions move data from the register file into memory and vice versa. The memory address must be loaded from a register since it is 16 bits long and therefore cannot fit in the instruction itself. This means load and store instructions will always specify the register address where the memory address is stored, and the register address where the memory data will be taken from or stored into. Similarly the Move instruction specifies a source register address and a destination register address, however the MOV instruction simply moves the data from the source register directly into the destination register rather than loading or storing anything in memory.

The LOADIMM instruction is slightly more complicated because it loads an 8 bit immediate value into the top or bottom half of register 7. This allows 8 bits to be defined as the immediate value in the instruction and then the top/bottom bit defines which side of the register the value will be loaded into. In order to prevent register 7 from being overwritten when the second half of an immediate is loaded into the register we require a feedback from register 7 that logic in the mem/wb latch copies and overwrites only the half required by the LOADIMM instruction before writing it back on the clock edge. This is done in the mem/wb latch combinational so the resulting register 7 value is available through the mem forwarding line in case of a later instruction requiring the result before it's been written back to the register file.

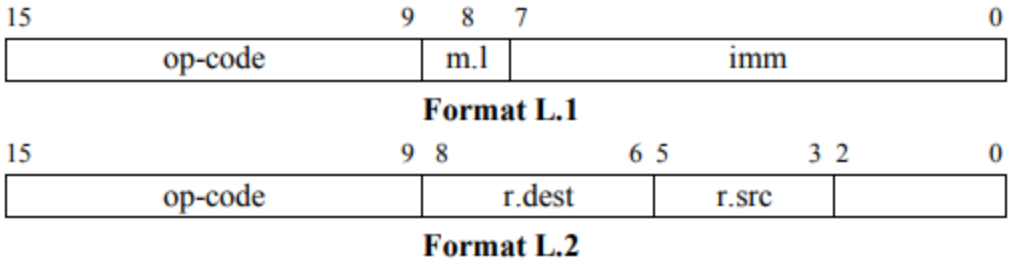

Figure 7: Format L Instruction Format

Table 4: Format L Instructions

Mnemonic	Op-code	Function	Type	Syntax
LOAD	16	R[r.dest] ← M[R[r.src]];	L2	*LOAD r.dest, r.src* *LOAD r.drst, @r.src*
STORE	17	M[R[r.dest]] ← R[r.src];	L2	*STORE r.dest, r.src* *STORE @r.dest, r.src*
LOADIMM	18	(m.l=1) →R7⟨15⋯8⟩← imm: (m.l=0) →R7⟨7⋯0⟩← imm;	L1	*LOADIMM.upper #n* *LOADIMM.lower #n*
MOV	19	R[r.dest] ←R[r.src];	L2	*MOV dest,src*

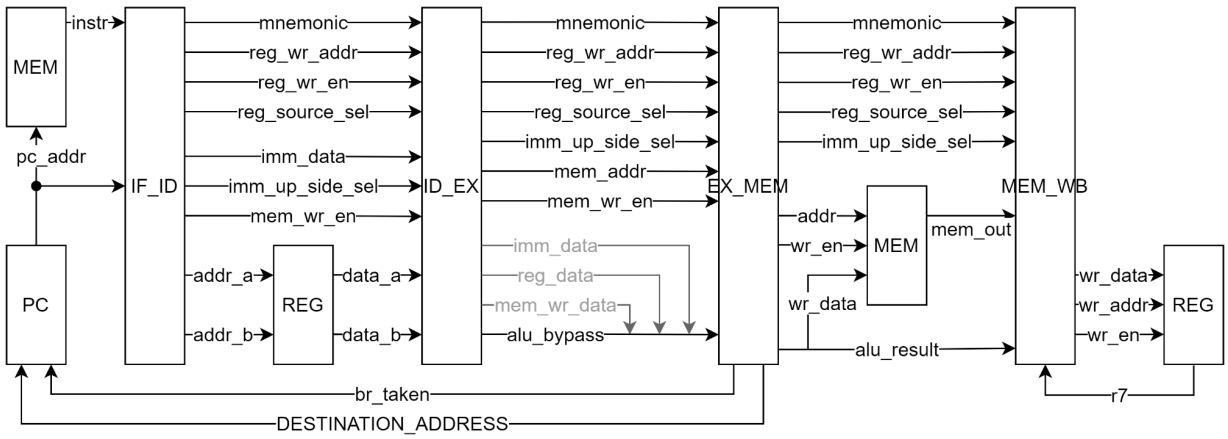

Figure 8: Format L Block Diagram

3.3.1 Memory

The memory interface abstracts the systems memory hardware as a 16-bit addressable space of continuous memory. In reality the memory consists of 1kB of ROM and 1kB of RAM, plus an input and output port. All unimplemented memory is read as 0 and cannot be written to. Only addresses 0x400 through 0x800 and 0xFFF2 can be written to as only the ram and output port are writable. The interface has two ports, one of which is used for instruction fetch/decode and the other is used for Load and Store operations which prevents memory data hazards. As seen in Figure 9, the inputs to the interface are the memory address, write data, and write enable (for load and store instructions) and the program counter for instruction fetch. Since the memory interface uses the Xilinx macros, the latency could be modified.

Using a latency of 1 cycle for both the ROM and the RAM, the memory interface presents it's outputs 1 clock cycle after it receives its inputs. This was the same latency as all the other stages in the pipeline, meaning no stalls were needed for regular load or store operations.

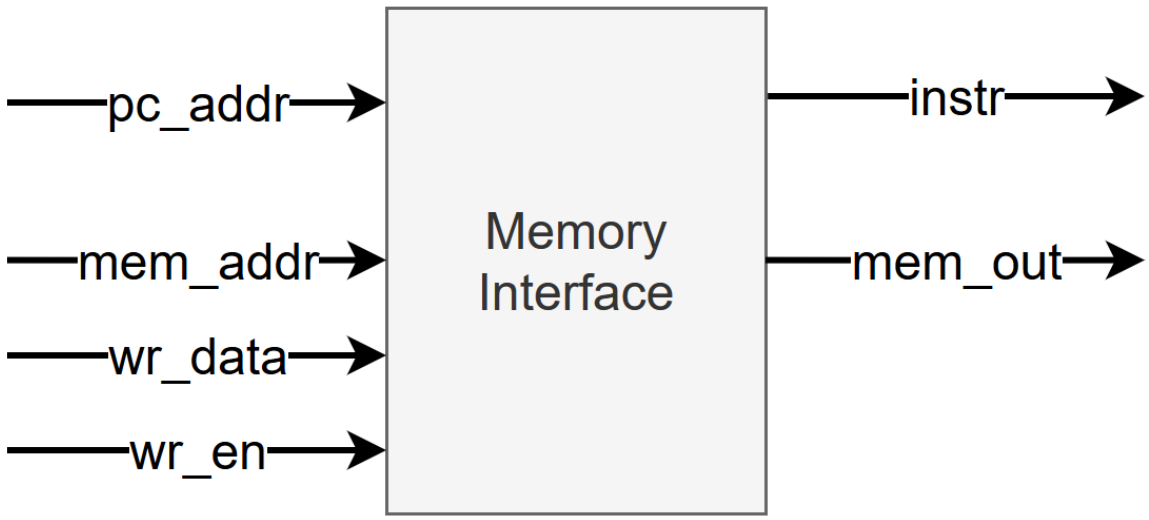

Figure 9: Memory interface block diagram

3.4 Hazards

Hazards occur when an instruction attempts to do something that our architecture can't handle. Due to our pipelined architecture with a limited instruction set there are only two types of hazards we need to handle: Data hazards and Control hazards.

Specifically the data hazards we need to handle occur when an instruction attempts to read data from the register file before it has been written to by a preceding instruction. This results from our pipelined architecture which allows the possibility of instructions being executed while another is attempting to read a result from that instruction. In order to handle these hazards we implemented data forwarding which provides access to results from previous instructions before they are written back to the register file.

To implement the data forwarding we first had to detect when a data hazard was occuring. To do this we created an internal instruction history that recorded what previous two instructions were executed and what register they were writing back to. We then compared this history to the inputs required by the current instruction and when there was a match we knew what input had to be forwarded to and where it had to be forwarded from. For most instructions the data would be forwarded from the alu_result or the output of the ex/mem latch depending on if there was a single instruction buffer between the two sequential instructions or not. If there was more than one instruction buffer between the two instructions then no forwarding would be required since the result would be written back to the register before the next instruction attempted to read it. This meant that no stalls would ever be required. LOAD and LOADIMM however did require up to one stall to handle because the result of the instructions would not be available until the mem/wb latch meaning an instruction immediately following either of these would have gone through the ALU before the result was even available rendering the forwarding useless. As

such when a data hazard was detected where an instruction required a result from the previous LOAD or LOADIMM instruction, a NOP would be injected into the system and the program counter would be stalled once so the instruction is sent back again at which point the hazard could be resolved by a simple data forwarding from the mem/wb latch.

The other type of hazard that had to be handled was control hazards caused by a branch prediction being incorrect. Because we are using a pipelined system where the branch cannot be tested until the ex/mem stage we would have to either stall the pipeline until the decision was made or run instructions with the chance that the prediction was wrong and we would have to flush the system. We decided to go with a predict branch not taken because it meant we did not have to add special logic that calculated the branch location in the if/id stage and instead allowed the system to continue on normally until a branch is decided to be taken at which point the if/id, id/ex, and ex/mem latches would be reset to flush the system and allow it to restart at the new branch location. While this did reduce efficiency somewhat because there was a potential two instruction delay when a branch was taken, it allowed the system to remain simple but still fully functional.

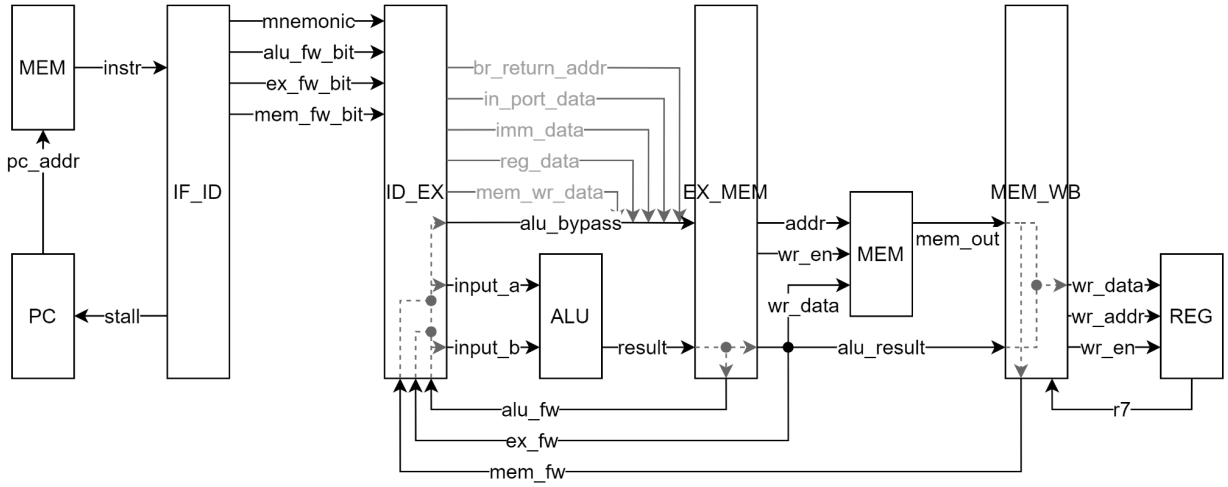

Figure 10: Data Hazard Handling Block Diagram

4.0 Results

4.1 CPI

The cycles per instruction (CPI) are calculated by counting the number of clock cycles and dividing by the number of instructions executed. Using the factorial example code, the CPI was calculated to be 1.25. For the counter example code, the CPI was calculated to be 1.31. The difference between the two values could be attributed to either a difference in the number of stalls due to data hazards, or a difference in the number of branches, which take more than one clock cycle.

4.2 Clock Rate

Following simulation of the processor, it was synthesized and implemented. From Vivado's timing tab shown in Figure 11, one can see the processor has a worst negative stalk of -5.82 ns and a Worst Hold Slack of 0.095 ns. Using these two values the Equation 1, one may determine the processor may achieve a maximum frequency of 63.6 MHz.

$$Max\ Frequency = \frac{1}{Clock\ Period - WNS - WHS} = \frac{1}{10ns - (-5.820) - (0.095)} = 63.6\ MHz$$

Equation 1: Max Frequency

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-5.820 ns	Worst Hold Slack (WHS):	0.095 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	-1590.647 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	1609	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2910	Total Number of Endpoints:	2910	Total Number of Endpoints:	788

Figure 11: Implemented Design Timing

4.3 Power

After implementing the design using Vivado and opening the power tab, one may see that total on chip power is 0.113 W as shown in Figure 12. The allocation of the power is displayed in the on chip power section of the power tab. Improvements to the processor's power requirements may be made by improving the processor ALU as it's methods of performing arithmetic and logic operations have not been greatly optimised.

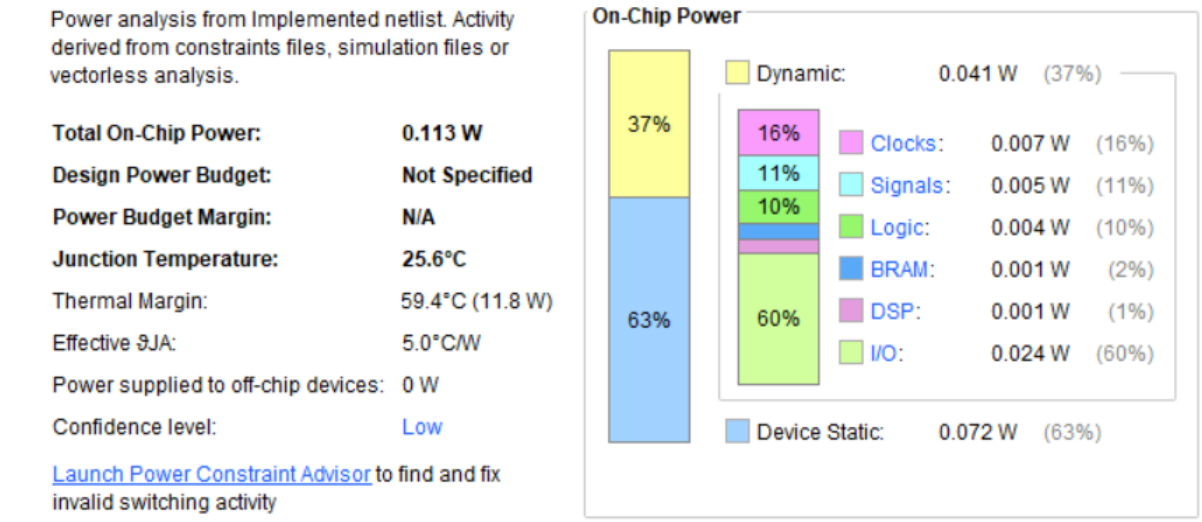

Figure 12: Power Requirements of Implemented Design

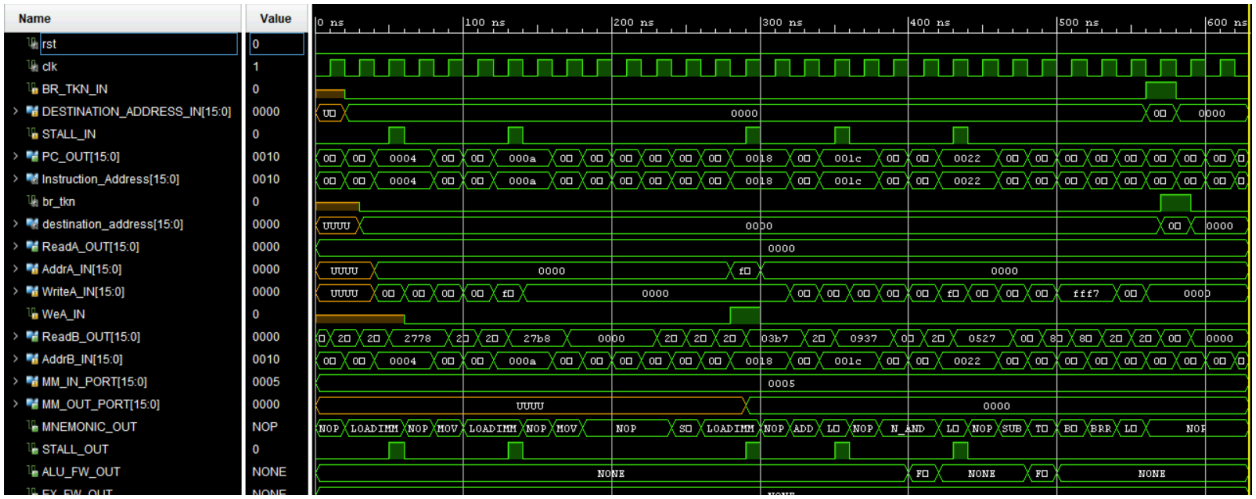

Figure 13: Wave Forms of Processor

One may see the simulated results of the processor in Figure 13. The waveforms display how instructions are pressessed and executed by the processor. As shown in the figure, one may see that the processor utilized the rising edge of the clock for latching data, while the falling edge is used for processor hardware such as the ALU. Looking at the waveforms, one may see instructions being operated on in sequential order as instructions are executed in the 5 stage pipeline.

The output of the counter program may be see in Figures 14 and 15. These figures show how the seven segment LED display was used to display the program's output.

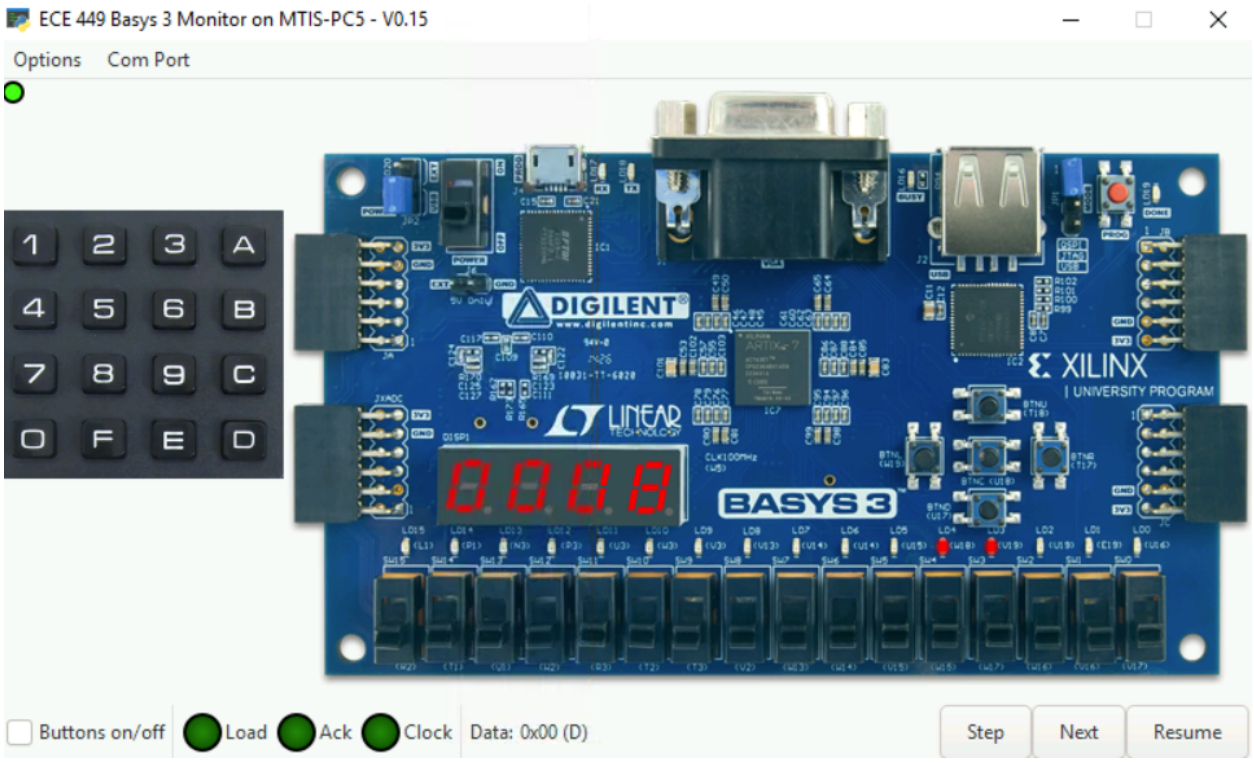

Figure 14: VHDL Counter Program at Counter Value 8 on Basys 3 Monitor

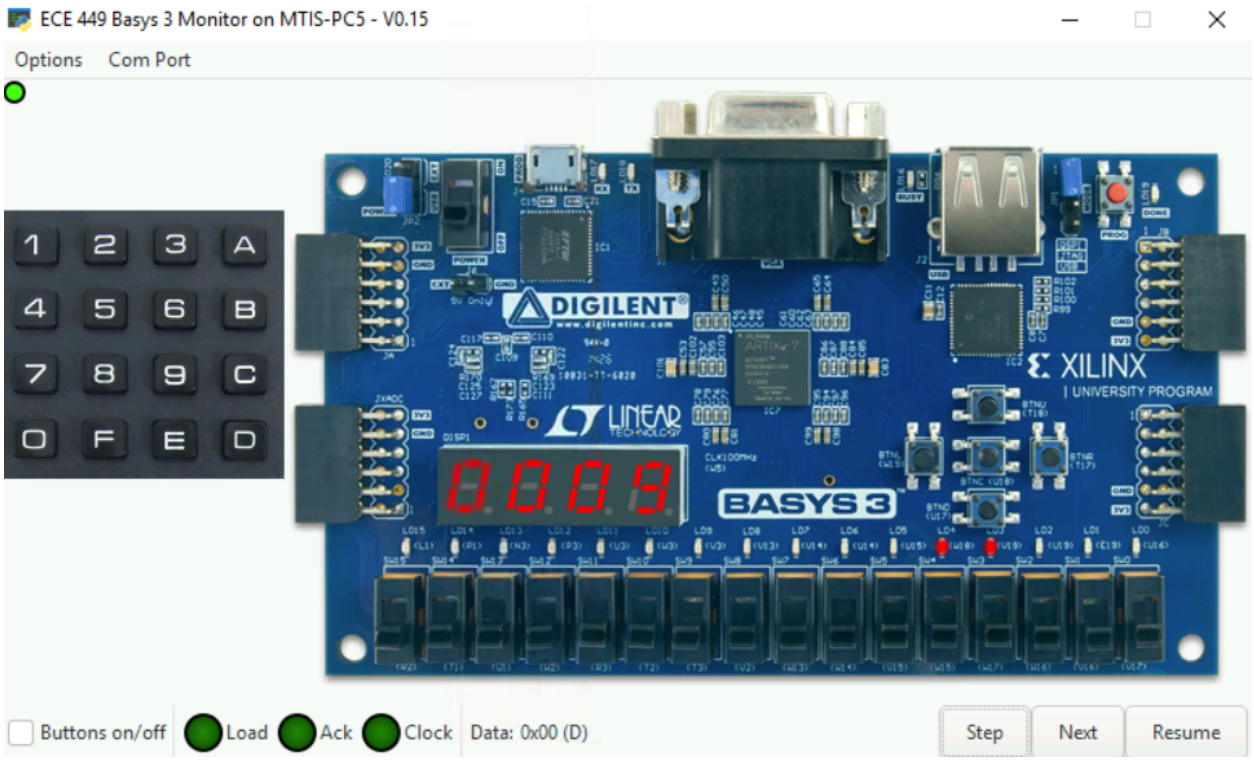

Figure 15: VHDL Counter Program at Counter Value 9 on Basys 3 Monitor

5.0 Discussion

5.1 Overflow

The system's overflow handling could be improved to increase consistency among the branch instructions and flag operation. Currently, the system sets (or clears) the overflow flag after every ALU operation, which is unlike the other flags that only change after a TEST operation. This also means the programmer must place all branch-on-overflow instructions directly after the ALU operation they want to check for, as no overflow information is currently stored. This difference in operation could be confusing to the programmer as they would expect all the flags to operate the same and interact with the test instruction. Ideally some method of storing whether a given value is the results of ALU overflow needs to be saved so that the TEST instruction can check for this. One method that was considered was to extend the register file by a single bit to store this overflow information. This extra bit would have to be ignored by the ALU aside from if a test instruction is being executed. This extra bit would be checked by TEST instruction and the overflow flag would be set accordingly. In this way the overflow flag would operate similarly to the other flags as it relates to the TEST instruction and branch operations.

5.2 System Shortcomings and Limitations

System shortcoming and limitations include the seven segment LED display being reset to display zeros when memory is not being written to. While this feature doesn't limit the team's ability to show that the processor is functioning as outlined, it could be improved to not reset between store instructions. Fixing this bug was pursued, however, after the system performed as expected in simulation and the signals used to reset the display were investigated, the issue was not resolved.

Another system shortcoming includes the system being unable to utilize the bootloader. While this issue was not resolved the team was able to rebuild the processor core after loading programs into the processor's memory to demondate the processor functioning for the in class demonstration. The issue regarding the bootloader was thoroughly investigated by the entire team. After configuring the Basys 3 board's buttons and seven segment display to display the instruction the processor was executing, the program counter, the data in memory and the output of the processor's ALU, the team stepped through the bootloader and saw that while all the instructions appeared to execute as intended the bootloader would get caught in an infinite loop trying to read data provided by the FPGA programming utility. Overall these minor shortcoming did not prevent the team from being able to present the processor functioning as intended. After utilizing workarounds such as rebuilding the processor's core with a program loaded into memory, the team was able to present the processor performing provided test bench code.

The current system's memory interface is byte addressable, however it cannot address odd numbered addresses. This limitation was imposed by the complexity of adapting the underlying modules (ROM and RAM macros) to byte addressable rather than word addressable. In the future, logic could be added to full enable byte addressability for all addresses. This would be done for odd addresses by

addressing the lower word, accessing its upper eight bits and concatenating these with the lower eight bits of the next word. This would introduce extra delays as two words would need to be addressed for odd addressees, however, these could be accounted for by stalling the pipeline.

6.0 Conclusion

In conclusion, the 16 bit CPU utilizes hardware such as a register file to act as memory, an arithmetic logic unit to perform arithmetic operations and control logic to execute instructions. Currently in the processor has been designed to process format A instructions. Instructions are fetched from the reggistor by the control logic then sent to the ALU to be computed. The output of the ALU, ALU_Result, will then be stored in the register file for later use. Future work of the project includes implementing subsequent instructions as outlined in the ISA document.

7.0 References

[1]	A Simple Arithmetic Logic Unit - https://learnabout-electronics.org/Digital/dig58.php
[2]	Digital Logic Textbook - https://www.academia.edu/12138790/Contemporary_Logic_Design_DSD_
[3]	MIPS 16 Bit Processor - https://www.fpga4student.com/2017/09/vhdl-code-for-mips-processor.html
[4]	FPGA 4 Students - Matrix multiplier - https://www.fpga4student.com/2016/11/matrix-multiplier-core-design.html
[5]	VHDL Combinational Logic Multiplier - http://wakerly.org/DDPP/DDPP3_mkt/c05samp3.pdf
[6]	Pipelined 32-bit CPU - https://www.researchgate.net/publication/281101032_VHDL_PROTOTYPING_OF_A_5-STAGES_PIPELINED_RISC_PROCESSOR_FOR_EDUCATIONAL_PURPOSES

Appendix A

See the attached appendix A document with all the relevant code used in implementing the project.

Appendix B

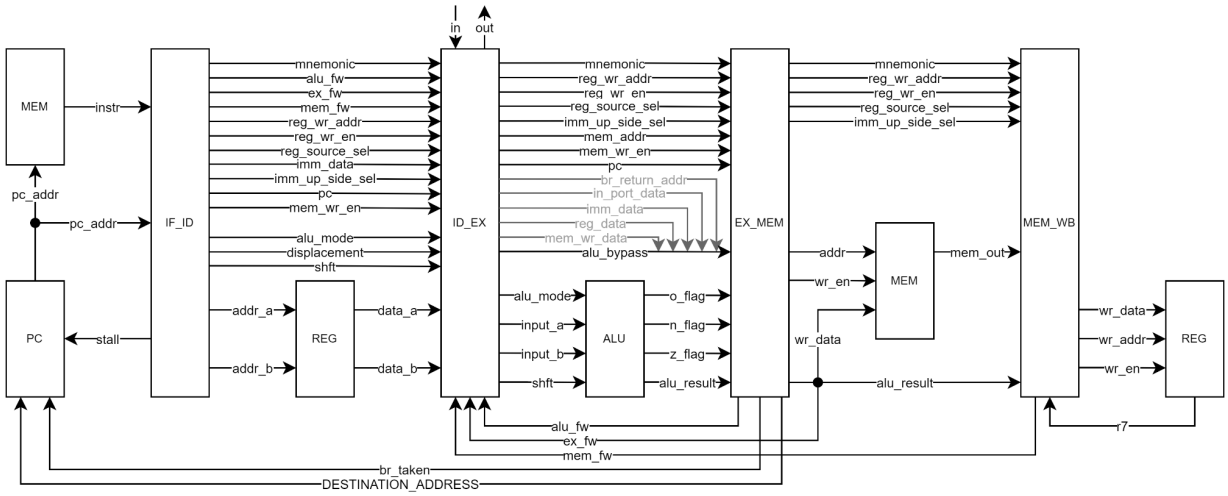

Figure B1: CPU Overview Block Diagram