SCHOOL OF SCIENCE AND ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EGC442 INTRO TO COMPUTER ARCHITECTURE

Final Project Report: ARM LEG v8 Processor

Date: 12/10/2018	Semester: Fall 2018

GROUP MEMBERS	DEPARTMENT	MAJOR CONTRIBUTION
Anthony Allwood	CE	Report and Conception
Andrew Mancinelli	CE	Report and Conception

Class Instructor: *Baback Izadi*

ABSTRACT

This project involves the creation of a detailed design and layout of an ARM LEGv8 processor. Given a set of unique instructions for the processor to perform, the data path of each class of instruction as well as the overall datapath of the processor was constructed. Efficient control logic was developed in order support each given instruction, along with a precise and detailed functional table that makes control implementable in a field-programmable gate array (FPGA). In order to perform these specific instructions for the ARM LEGv8 processor, a Register File and an Arithmetic Logic Unit (ALU) along with a variety of other components must be incorporated within the processor and datapath. A detailed layout of the register file and a detailed design of an ALU were also established. All designs and complex layouts mentioned above were carefully constructed using the Altera Quartus II 15.0 (64-Bit) programmable logic device design software program.

TABLE OF CONTENTS

INTRODUCTION

An ARM processor is considered one member of a family system of Central Processing Units based heavily on reduced instruction set computer (RISC) architecture. ARM processors such the ARM LEGv8 processor are designed in order to perform various types of instructions such as R-type instructions including AND, ORR, ADD, SUB and I-type instructions such as LDUR (load) and STUR (store). ARM processors are utilized in a variety of electronic devices in today's society such as tablets, mobile phones, and multimedia players due to their reduced complexity and ability to consume low amounts of power. This project serves as a way of understanding and analyzing the abilities and functions of an ARM processor by establishing the design layout and datapath of an ARM LEGv8 processor.

PROCEDURE & DESIGN

Register Unit

The ARM LEGv8 processor incorporates a 32 x 64- bit register file for frequently accessed data. The register file is the component that contains all the general purpose registers of the microprocessor.Within the ARM LEGv8 design, registers serve as essential storage locations inside the processor that hold data and addresses. The general purpose registers utilized within the processor are labeled X0 to X30, along with XZR or register 31. The figure below shows the detailed design layout of the register file. A complete register file includes two main sections, the read register and the write register.

The read register section of a register file contains a block holding the data for each general purpose register labeled Register 0 to Register n-1. In the case of an ARM LEGv8 processor, the register labeled Register 0 would represent the procedure argument/result register known as X0. This concept represents each of the general purpose registers going up to Register n-1 which represents register XZR. In order for the processor to access or retrieve any of these registers for a specific instruction, two multiplexers are implemented within the design. These two multiplexers correspond with read register 1 and read register 2. Once the multiplexers select the appropriate registers needed for instruction, they are brought brought out of the register file to be used for the execution stage of the processor. This concept is indicated by the register data/information being lead out the multiplexer and represented as Read Data 1 and Read 2. The second section of the register file refers to the write register section that is utilized to manage the execution results or write data being fed into the register file and the write register itself.

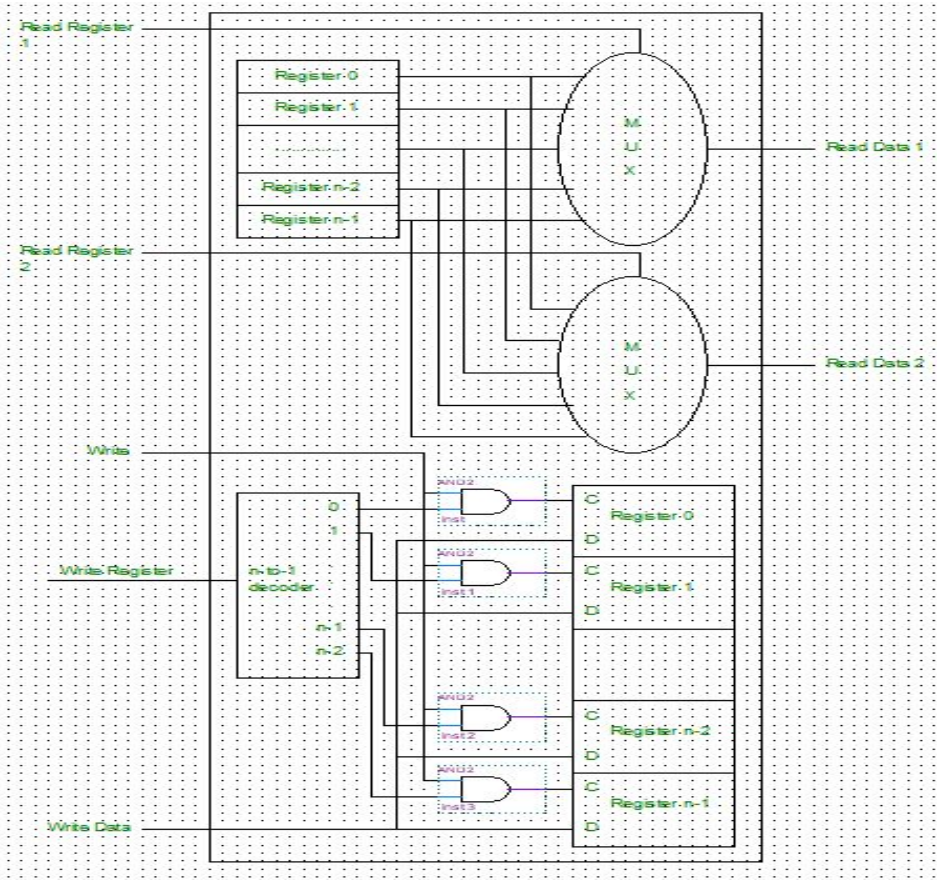


Figure 1: Register File

The write register section of a register file contains a n-to-1 decoder that is used to take in the write register information from the instruction memory and select a register that will be used to store the execution results/write data. The output of the decoder is fed into an “and” gate along with the write bus value of 1 in order to select which register to write data. The register block similar to the one in the read register section is also utilized for register selection. If the “and” gate corresponding which each general purpose register has an output value of 1, that specific register is used for write data. Notice how the write data information is connected to each possible write register. However, only the register that received the value of 1 from the “and” gate became the write register.

ALU

The ARM LEGv8 processor also incorporates an Arithmetic-Logic Unit in order to execute the specific instructions given. Certain instructions require arithmetic operations to be executed efficiently while other instructions require logical operations to executed. An arithmetic logic unit (ALU) is a combinational electronic circuit that performs arithmetic and logical operations in relation to binary numbers. It is the component of the processor that receives register information from the register file or instruction memory in order to execution operations such as AND, OR, ADD, or SUB. The execution results from the ALU can be sent to the data memory or be written back to the register file where it is stored in a write register, as mentioned before. The figures below portrays a detailed design of the complete 32-bit ALU circuit, the simple ALU internal architecture for each given instruction, and the ALU's function table.

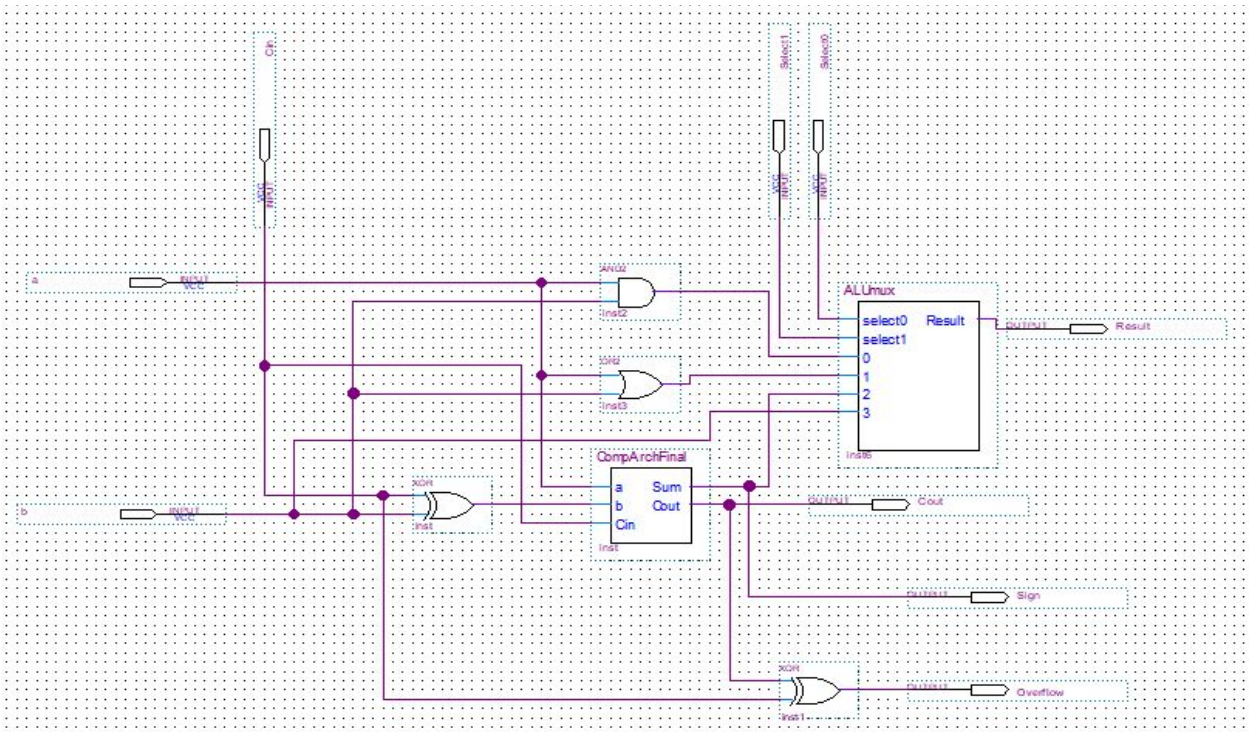


Figure 2. Simple ALU design and internal architecture for instruction set.

ARM LEGv8 Instructions	ALU Function
AND	AND
ORR	OR
ADD	Addition
SUB	Subtraction
LDUR	Addition
STUR	Addition
CBZ	Pass input b
CBNZ	Pass input b
B	No function

Figure 3. ALU function table

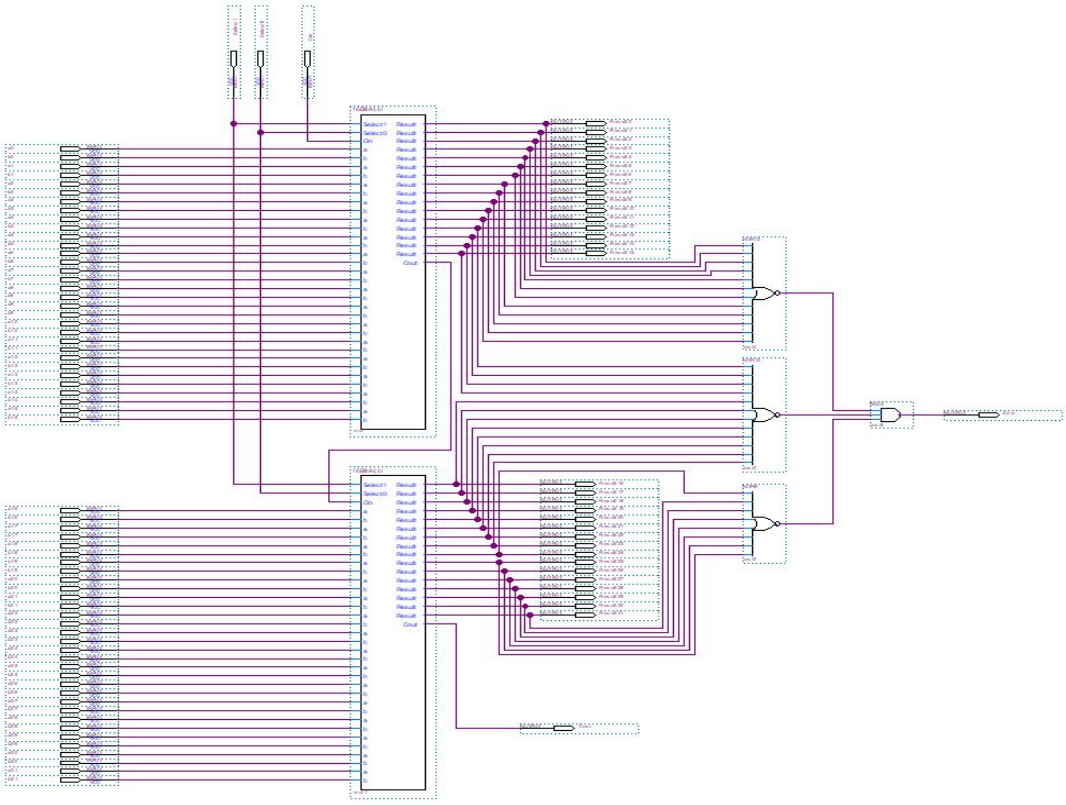


Figure 4: 32-bit ALU circuit

This design of the 32-bit ALU accounts for the four operations necessary for the set of given instructions LDUR, STUR, CBZ, CBNZ, B, and R-types. These four operations are AND, ORR, ADD, and SUB. The design portrayed above was established by creating a 1-bit ALU with the necessary operations, inputs (a) and (b) to represent the register values coming from the register file, an input for carry-in (Cin), a multiplexer with two selection lines coming from the ALU control which selects the operation that is required for a given instruction, an output for carry-out, and an output for the result value. This 1 bit ALU was replicated four times in order to create a 4-bit ALU. The 4 bit-ALU was replicated four times to create a 16-bit ALU, and as seen above, a 16-bit ALU was replicated twice to establish a 32-bit ALU. This process was established in order to support easier and efficient design within the Altera Quartus II 15.0 (64-bit) software program.

The design takes into account the 32-bit value of (a) and the 32-bit value of (b) that enters the ALU from the register file and outputs the 32-bit value of (Result) that comes from operation execution. The operation to be executed is determined by the two select lines (select 1) and (select 0). The design also includes the value of the zero flag that outputs from the ALU. All 32-bits of register value were taken using three NOR gates. The values of the NOR gates were taken in by a three-input OR gate. If the output value of the OR gate is equal to 1, then the zero flag is set to 1. The design of the 32-bit ALU was analyzed, tested, and simulated using the Altera Quartus program, proving that it was efficient and implementable using a FPGA circuit or board.

Control Unit

To create the processor's control unit, eight of the available 11 bits were grabbed from each instruction's opcode and applied to a simple logic circuit in order to distinguish between each individual instruction and control the processor accordingly. These bits are used to throw nine flags, which tell the processor exactly where to forward the information that is being requested. These nine flags are as follows:

- "WRITEMEMORY" - Controls whether or not the information forwarded past the Ex/MEM pipeline is written to memory. This flag will be active if bit 1 reads "1" and bit 9 reads "0", indicating a STUR instruction.

- "READMEMORY" - Controls whether or not information is read from memory for use elsewhere in the processor, primarily by a register. This flag will be active if bit 9 reads "1", indicating an LDUR instruction.

- "MEMTOREG" - Controls whether information in memory is written to a register or not. In this case, we only have one instruction that can both this and "READMEMORY" (LDUR). Therefore, this flag will always read the same as "READMEMORY".

- "BCONDIT" - Indicates if the current instruction, if it is a branch, requires a condition to perform the operation. This flag will be active if bit 0 reads "0", indicating a B instruction.

- “BRANCH” - Indicates whether or not the instruction is a Branch instruction. This flag will be active if bit 4 reads “0” and bit 5 reads “1”, indicating any of the branch instructions.

- “ALUOP[0-2]” - Tells the ALU what operation to perform on R-Type or Branch instructions.
    - “011” - AND
    - “001” - OR
    - “010” - ADD
    - “110” - SUB, this will always be the case if a Branch instruction is currently present.

  ALUOP[0] will be active if either bits 1 and 7 read “1”, or if the BRANCH flag reads “1”. ALUOP[1] will be active if either the product of bit 1 NAND bit 7 results in “0”, or if the BRANCH flag reads “1”. ALUOP[2] will be active if bit 7 reads “0”. ALUOP[2] will also be forced to read “0” if the BRANCH flag reads “1”.

- “ALUSRC” - Tells the ALU whether to read its second operand from ReadData2 (Register File), or directly from the instruction after being sign extended. This flag will be active either when bits 3 through 5 read “110” respectively, indicating a LDR or STUR instruction, or when bits 3 - 5 read “101” respectively, indicating a branch instruction.

The circuit for this control unit is as follows:

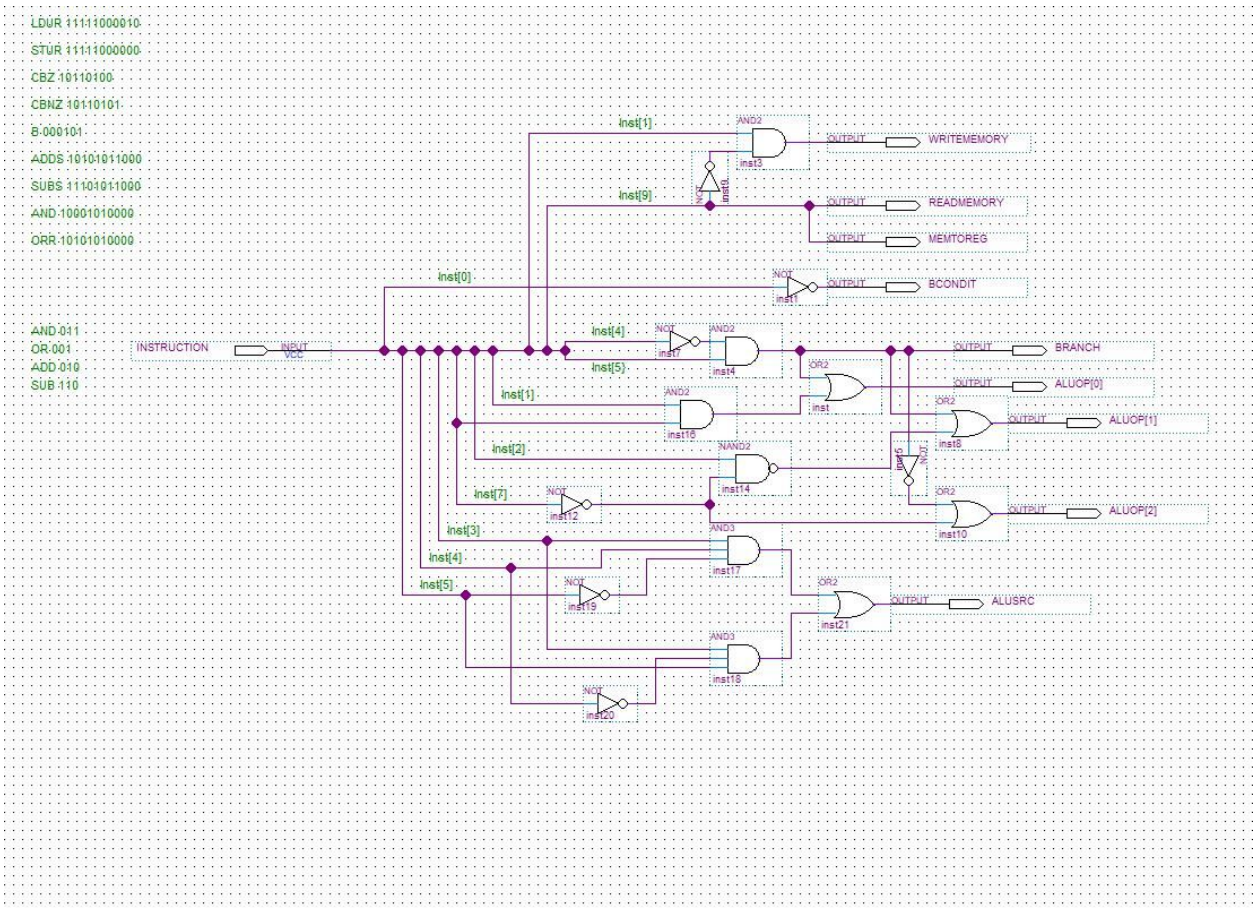


Figure 5: Control Unit

Reference 1 contain a table of every instruction's opcode, what each bit in the opcode can indicate, what flags each bit can affect, and the meaning of the instructions in our processors instruction set.

Datapaths

With a limited set of instructions, the processor had a fairly simple datapath. The top-level view is as follows:

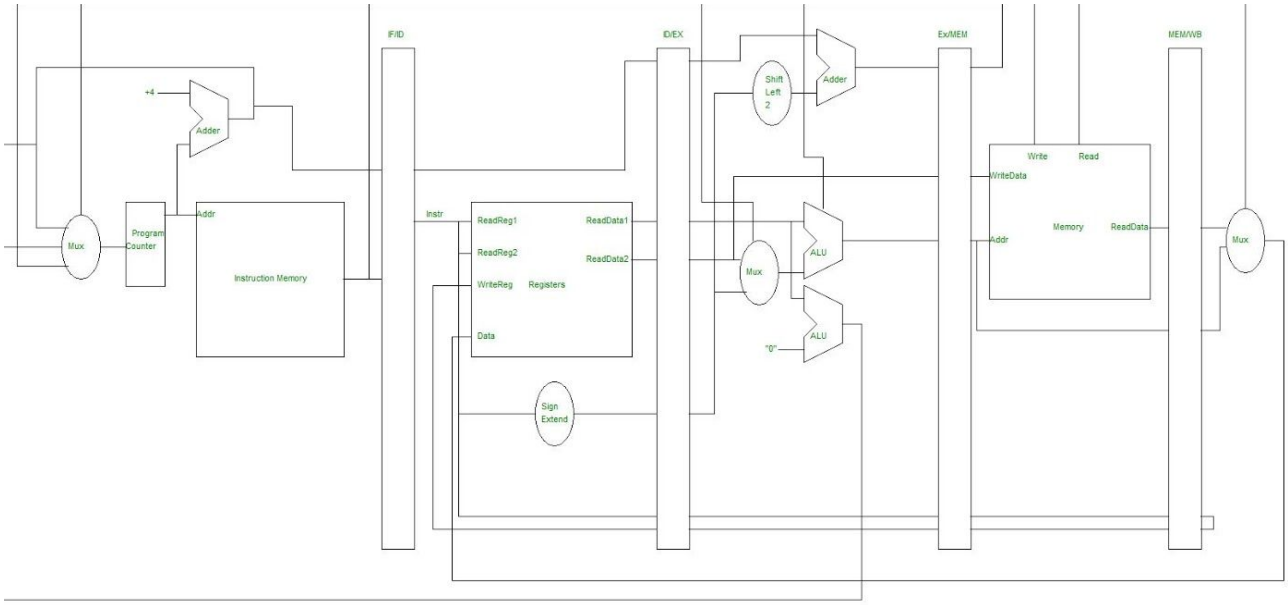


Figure 6: Top-down view of processor architecture.

This set of data paths can be split into two primary paths - one for branch instructions and one for load, store, and R-type instructions.

Branch

For conditional branches, the opcode will provide an address into the register unit, which will then be fed into an ALU, which then checks for the presence of a “0” in the referenced register. For CBZ, the unit will pass that branch address to the program counter, given by the opcode, and passed through a sign extension, and left shift (2 bits) as long as the register contains

a zero. For CBNZ, the same will happen as long as the referenced register does NOT contain a zero. Here is the circuit for conditional branches:

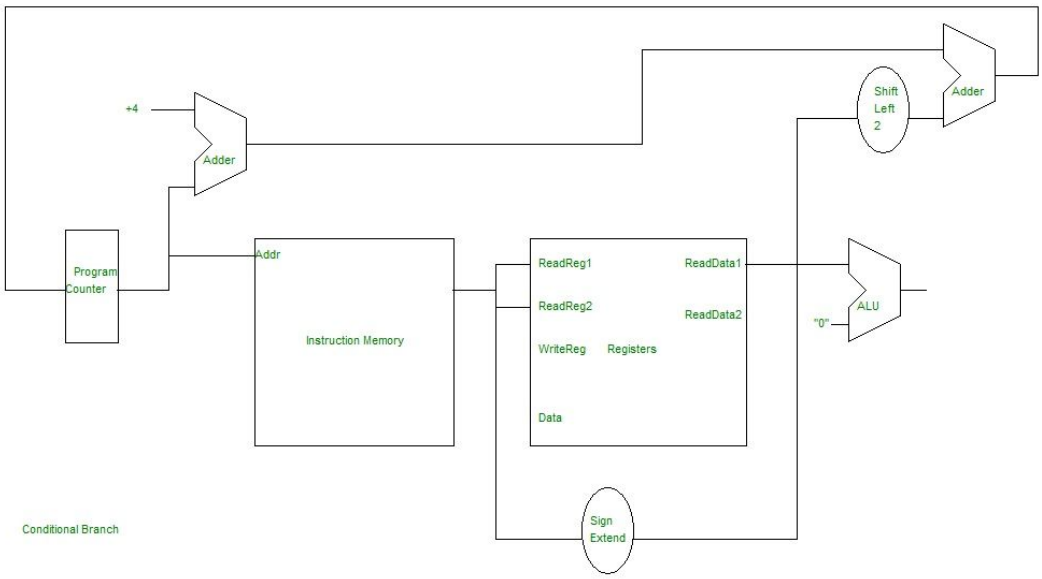


Figure 7: Datapath for conditional branch instructions

Unconditional branches have a very similar circuit, except the ALU is bypassed, which will cause the branch to happen regardless of whether or not the referenced register contains a zero. Here is the circuit for unconditional branches:

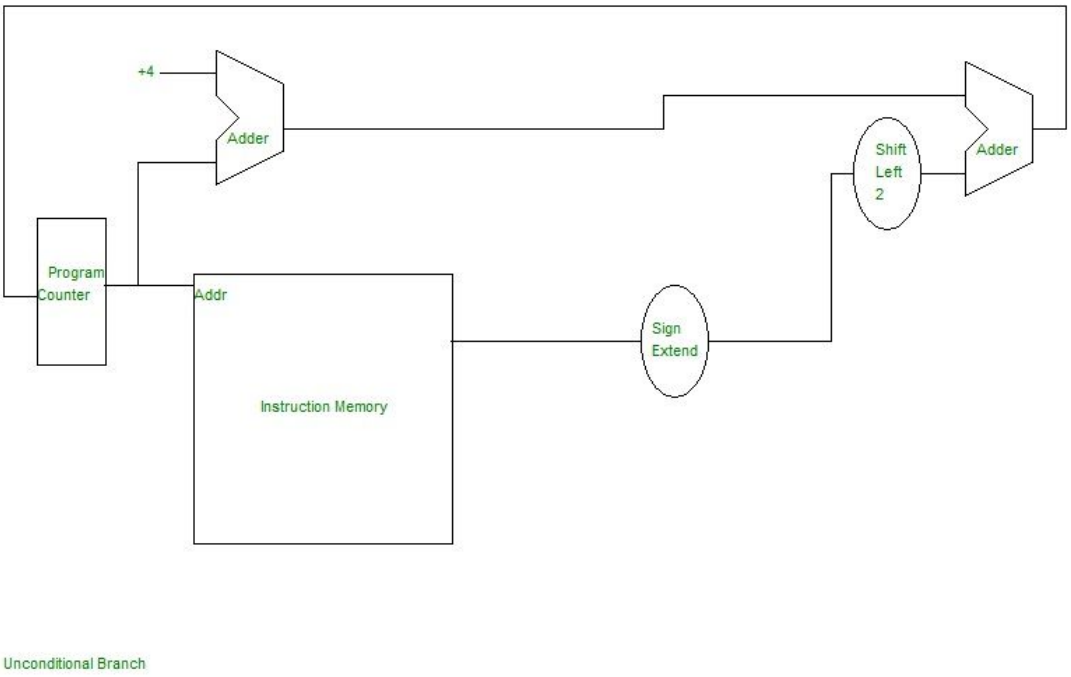


Figure 8: Datapath for unconditional branch instructions

R-type, Load, Store

The circuits for load, store, and R-type are all very similar, and therefore can be easily compared in a single data path diagram. For R-type instructions, the opcode will provide two registers which contain the information to be passes through an operation, be it addition, subtraction, AND, or OR. After the needed data is read from its registers, it passes through the ALU, which is controlled by the ALUOP flags of the control circuit to indicate the proper operation to be performed on the information from the two referenced registers. After the operation is performed, the result is then passed to its proper register's address, which is indicated by the "WriteReg" input, and passed via the "Data" input of the register unit.

For Load instructions, instead of passing any data between registers, an address will be passed to the memory unit, which will pull corresponding information from memory (passed thanks to a multiplexor, controlled by the control unit) and transfer it to a register.

Store instructions go the opposite direction, meaning the referenced address will be given to the register unit. The data at the referenced register will then be stored into memory. For both Load and Store instructions, the ALU will use information coming from the sign-extend unit instead of a second piece of data from a register, which is used solely for R-type instructions. This is what the data path for these instructions looks like:

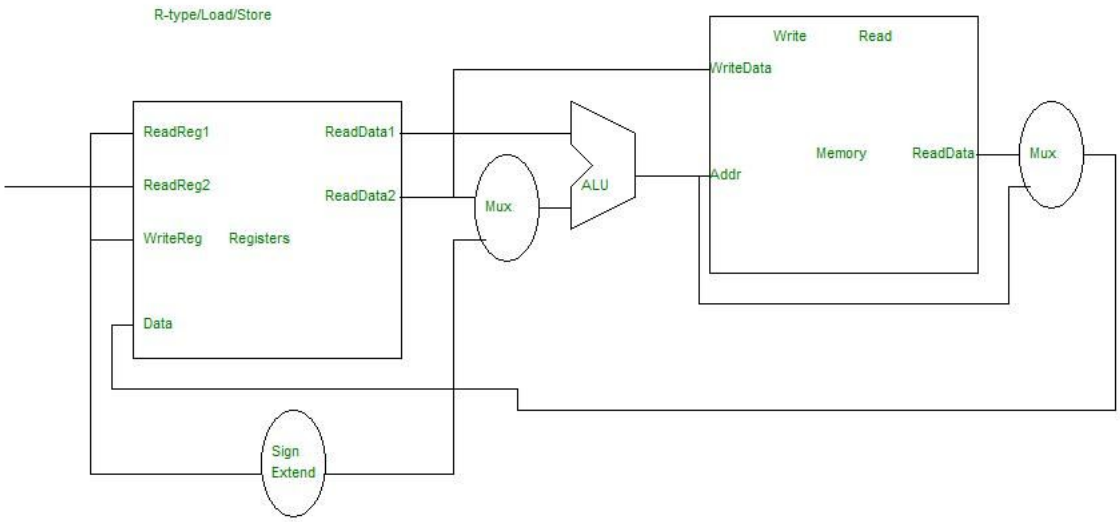


Figure 9: Datapath for R-Type, Load, and Store instructions

Reference 2 shows the completed data path, included pipelining architecture. This completed data path combines these two previously mentioned data paths into one, providing us with our full processor.

CONCLUSION

This project exposed to us the process of creating a 32-bit ARM LEGv8 processor, and with that came an excellent learning experience on how these types of processors work architecturally.

With great learning experiences, come great problems that require troubleshooting, research, and the occasional hypothesis to solve. Although many problems and unknown issues arose, some of the greatest hurdles were during the development of the control circuit as well as the ALU. For example, a large portion of our time worked on the project involved creating a control logic, as seen in Reference 1 as well as the diagram in the "Control Unit" portion of this report. The logic took much brain power, and finding the unique bits per instruction took a very long time. However, this problem was eventually solved, and the control logic seems to work fine for the intents and purposes of our processor.

Another problem we faced was being able to create a "Zero" flag across the numerous outputs of our 32-bit ALU. Eventually, it was discovered that by put a NOR gate on the individual outputs, and then an AND gate connected the outputs of those NOR gates together, the result would be the "Zero" flag we were looking for.

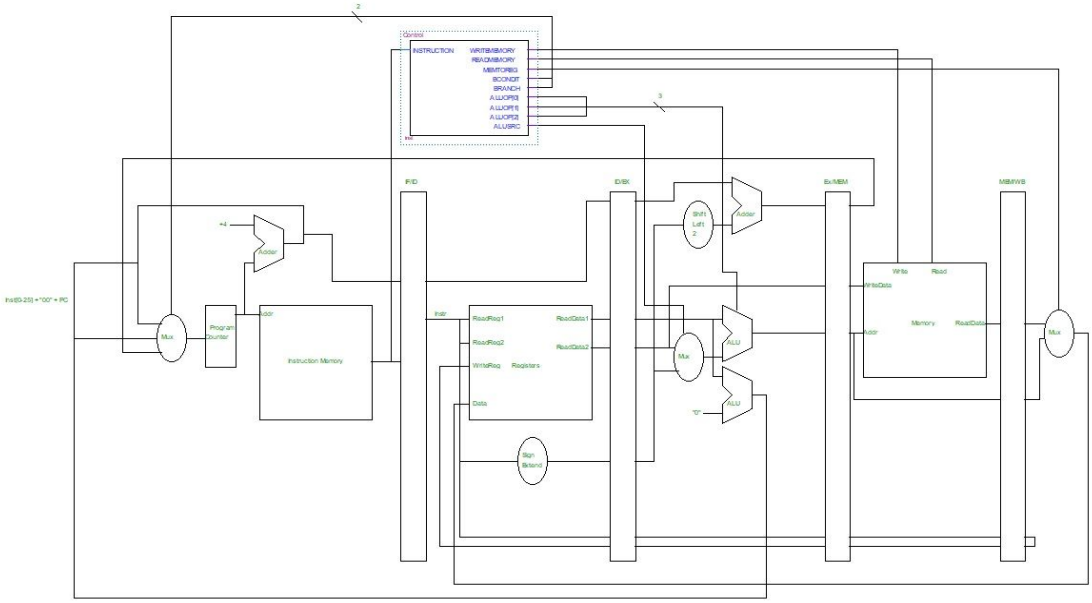
Although there were issues during the design process, we feel that this project has glued together the concepts of the course taken prior to this project, and we definitely feel more comfortable with the theory of computer architecture. In all, being able to do this project was a great experience, and due to this, we have greatly enhanced our knowledge of computer architecture.

- Reference 1, created by Andrew Mancinelli.

Identifying Bits		Affected Flags
Instruction[0]	If 0, Unconditional Branch	BCONDIT
Instruction[1]	If 1, Load, Store, or SUBS	ALUOP[0], WRITEMEMORY, READMEMORY
Instruction[2]	If 0, Unc. Branch, or AND	ALUOP[1]
Instruction[3]	If 0, R-Type instruction	ALUSRC
Instruction[4]	If 0, Branch instruction	BRANCH, ALUSRC, ALUOP[0-2]
Instruction[5]	If 1, Branch Instruction	ALUSRC, BRANCH, ALUOP[0-2]
Instruction[6] is unused.		
Instruction[7]	If 1, CBNZ, ADDS, or SUBS	ALUOP[0-2]
Instruction[8] is unused.		
Instruction[9]	If 1, Load	WRITEMEMORY, READMEMORY, MEMTOREG
Instruction[10] is unused		

Opcode and Instructions			
LDUR	Load	11111000010	I-Type
STUR	Store	11111000000	I-Type
CBZ	Branch if Zero	10110100	I-Type
CBNZ	Branch if Not Zero	10110101	I-Type
B	Branch	000101	I-Type
ADDS	Add	10101011000	R-Type
SUBS	Subtract	11101011000	R-Type
AND	And	10001010000	R-Type
ORR	Or	10101010000	R-Type

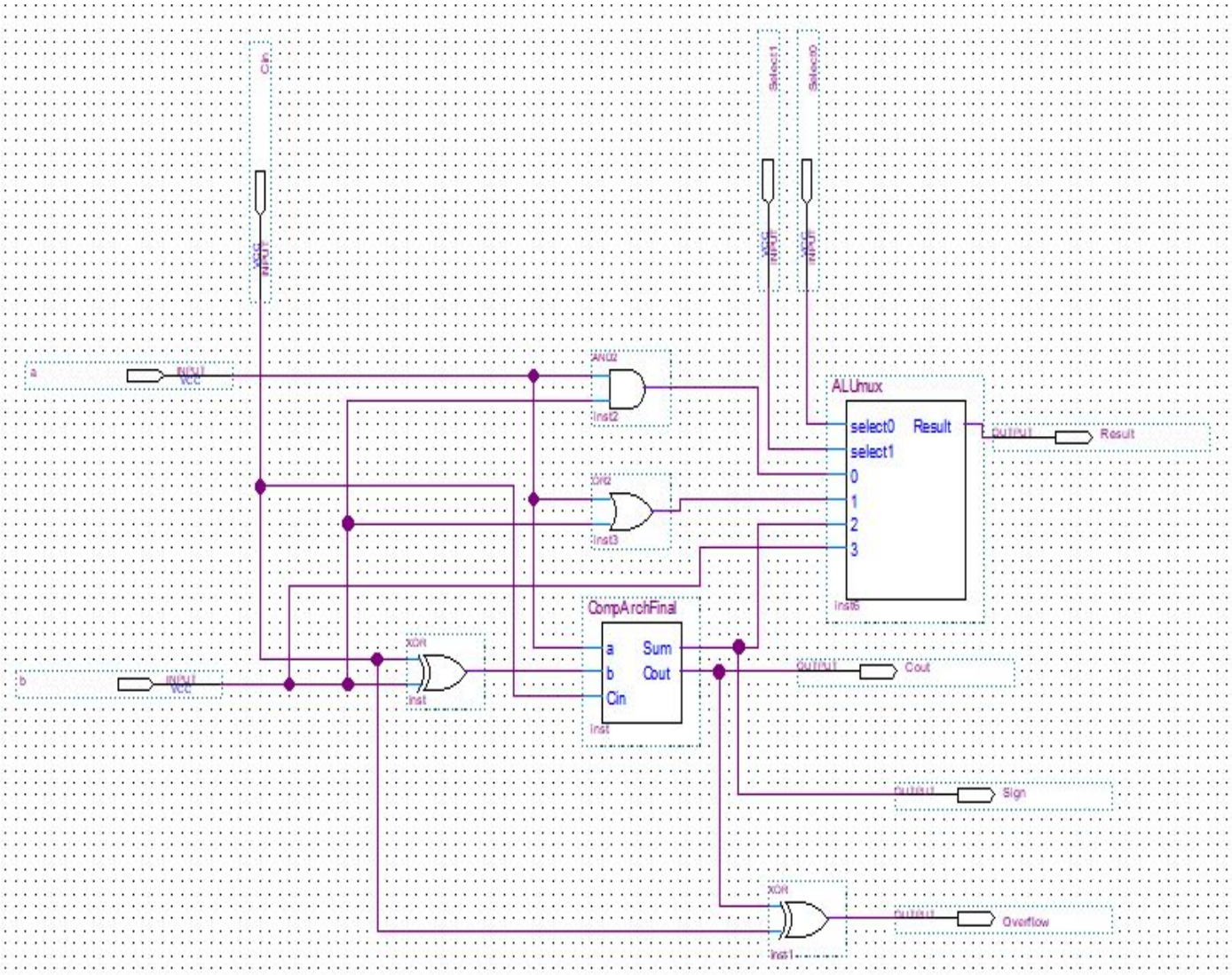
- Reference 2, Created by Andrew Mancinelli



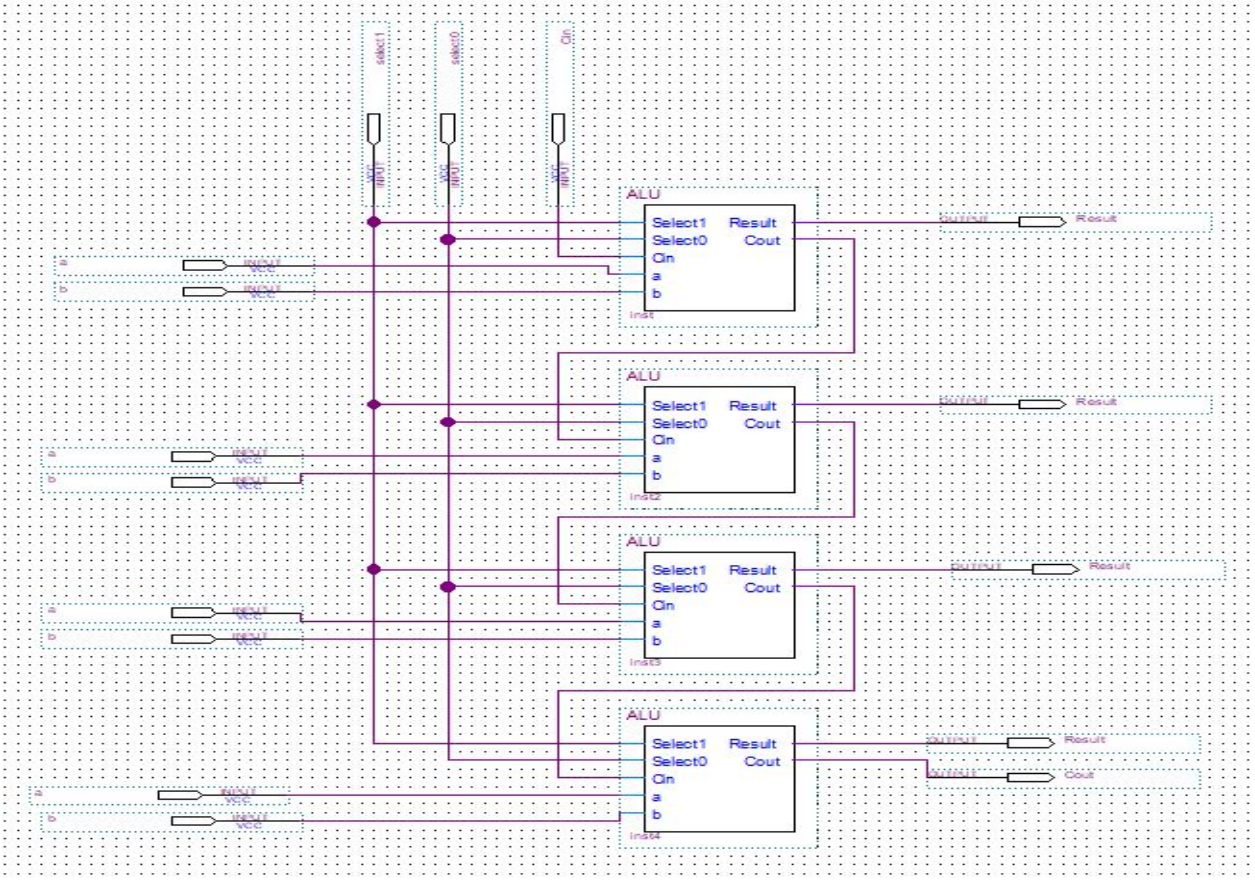
- Reference 3, Credit Dr. Baback Izadi, Ph.D.

Izadi, B. (2018). [online] Engr.newpaltz.edu. Available at: http://engr.newpaltz.edu/~bai/EGC442/Project_fall18.pdf [Accessed 10 Dec. 2018]

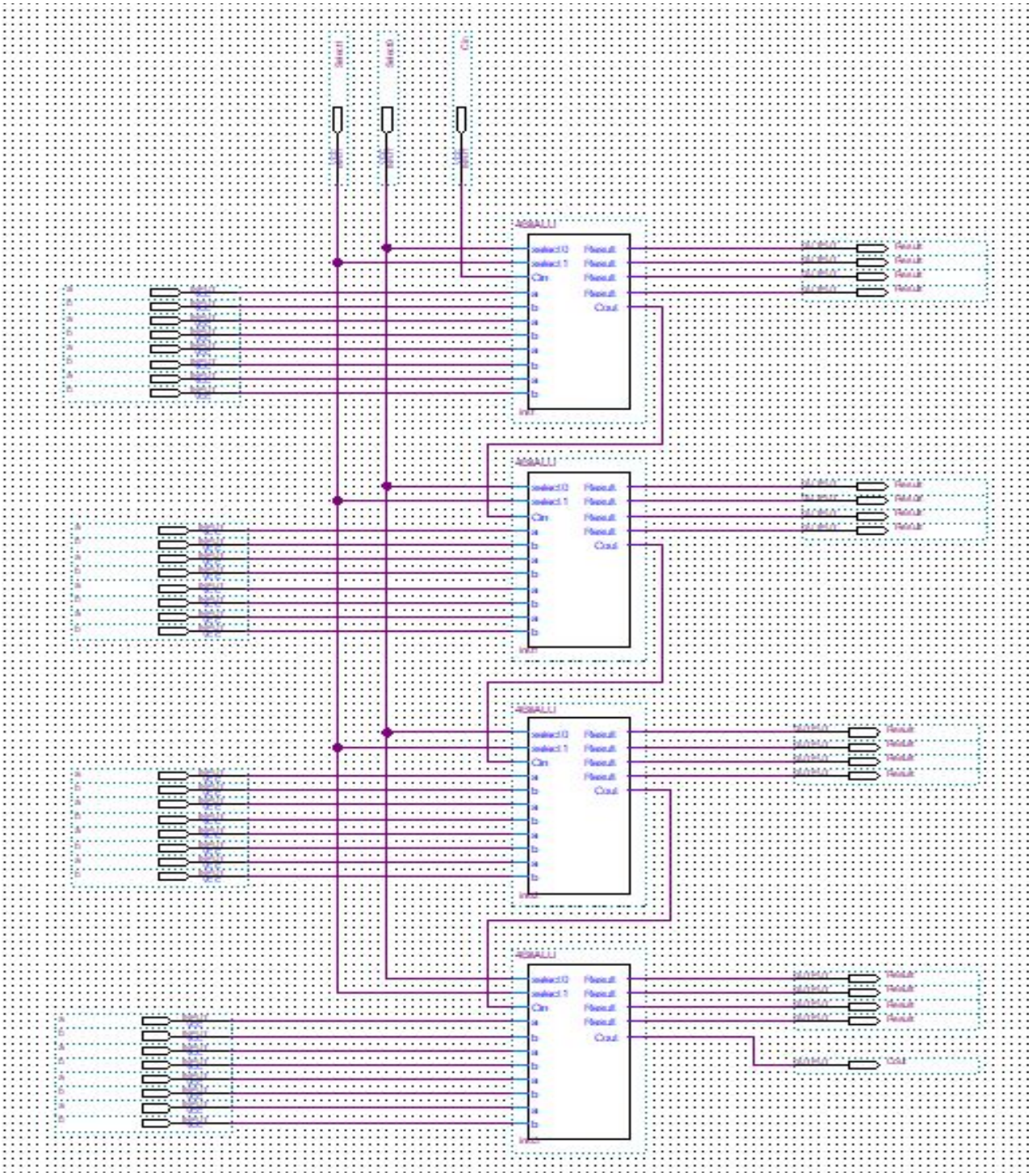
Simple ALU Internal Architecture for Instruction Set



4-Bit ALU



16-Bit ALU



32-Bit ALU