CE-222 COAL

Instructor: Dr. Ghulam Abbas

Project Report

Members:

- Zaid Dandia **(2021719)**
- Tahir Muzaffar **(2021665)**
- Sarim Ahmad **(2021572)**
- Mohammad Omar Khan **(2021305)**

Table of Contents

Abstract

Although the CPU is the core of any computer system, it is made to be easily adapted and modified to suit a user's specific need. There are three features that, we believe, makes our CPU unique and innovative:

- The use of two separate accumulators, both as a source register and a destination register.
- Including both the source registers **(from which we receive data for an operation)** and the destination register **(where we store the results of an operation)** in one singular instruction.
- Direct transfer of data between registers, without using an intermediate register.
- The use of an index register, which allows us to access 12 bit memory addresses, even though the architecture is designed with 8 bits in mind.

When used in combination, these four features provide the following advantages:

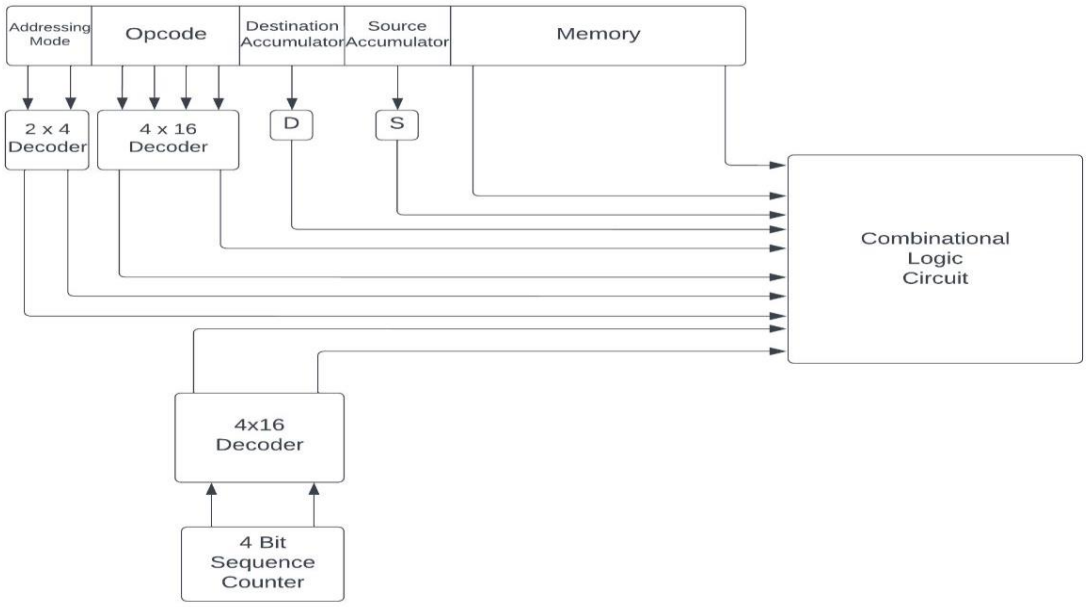
Improved Performance and Efficiency: By implementing these changes, we eliminate the need to use an intermediate register to store data. Due to this, an operation can be completed in fewer clock cycles, which improves overall performance and efficiency.

Reduced Memory Usage: Through the use of these improvements, we can perform a certain operation with fewer instructions. Additionally, the architectures is only designed with 8 bit memory addresses in mind, but can be used to access 12 bit memory addresses. Both of these combine to reduce the overall memory requirements of the CPU.

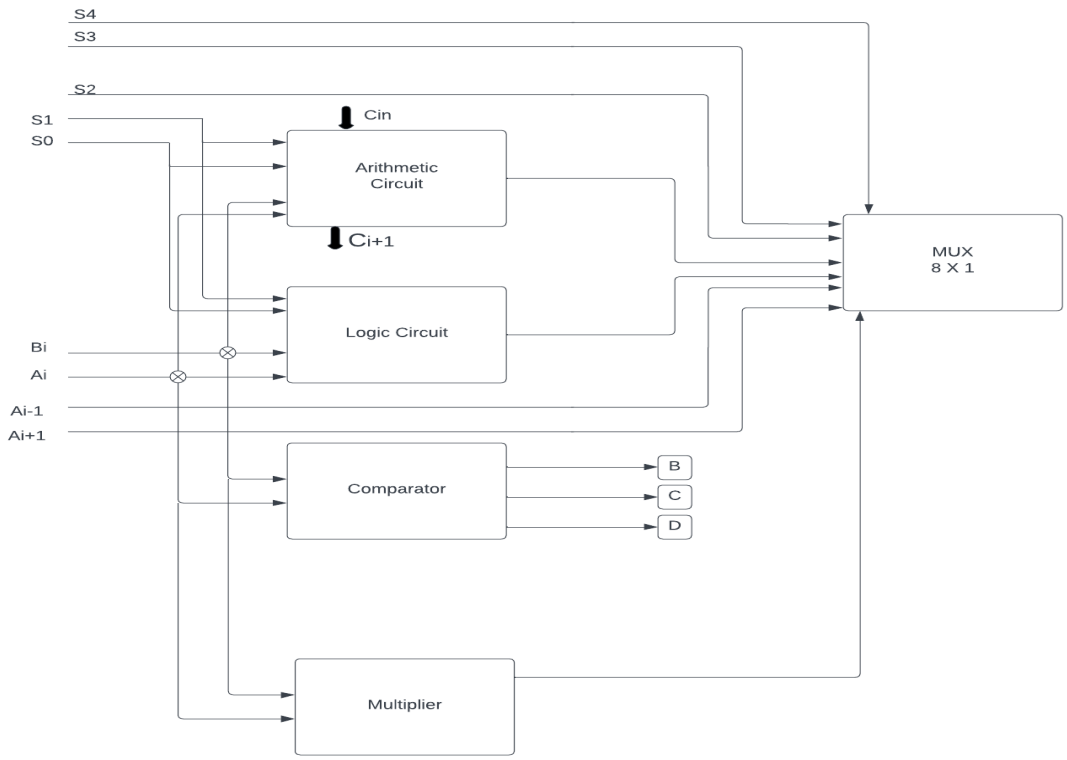
Fewer Instructions Required: If we make use of these innovations, the CPU would not need to perform additional instructions to transfer to and from an intermediate register. Additionally, the instruction would only contain an 8 bit memory address, but would be able to access 12 bit memory addresses. Both of these would combine to not only simplify the instruction set, but also further reduce memory usage.

Theory and Design

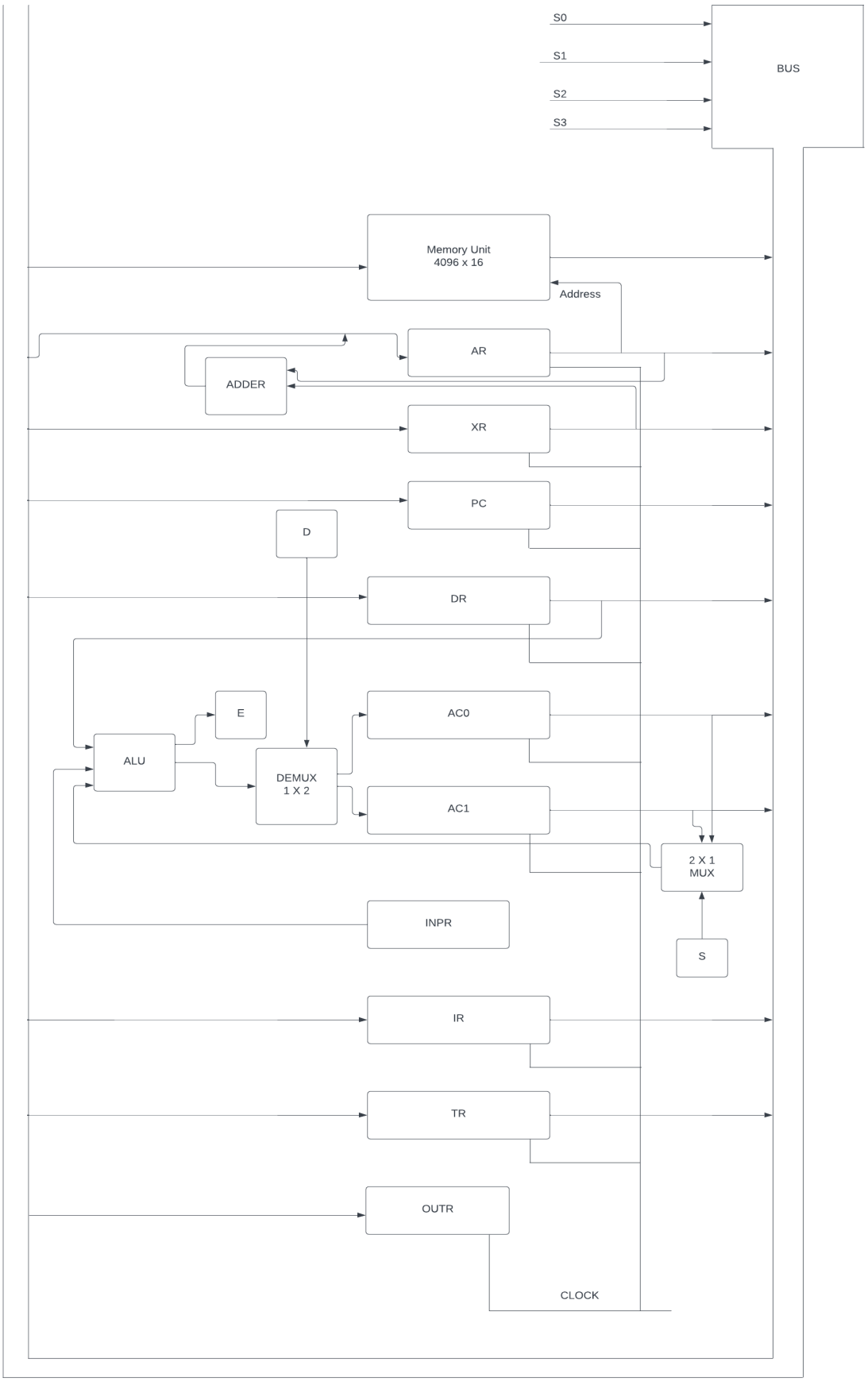
Control Unit Diagram



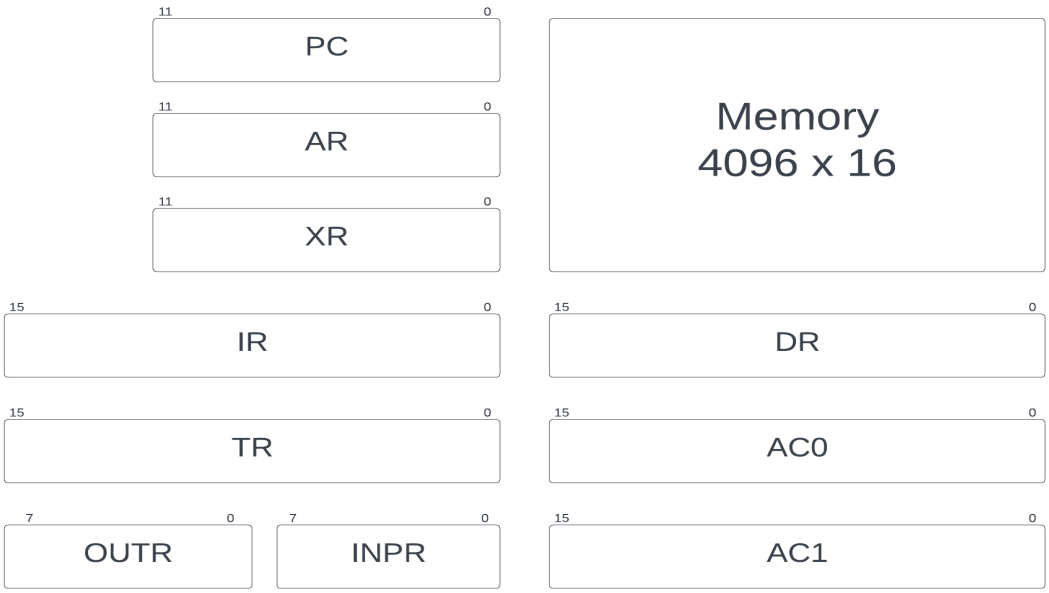
ALU Diagram



Bus System Diagram



Components in Bus System



When a component needs to send data to another component, it must send a signal to the bus, which contains information about the type of operation **(read or write)**, and the address it is accessing. There are 4 select lines **(S3, S2, S1, S0)** connected to the bus, which decide which of the nine components will be able to send a signal to the bus. After the transfer of data is completed, the next component will be able to send a signal to the bus.

RAM (4096 x 16): The RAM has a maximum capacity of 4096 words **(i.e. memory locations)**, each of which can hold 16 bit long instructions. Each memory location can be identified by its 12 bit address. We can think of the RAM as a table that contains 256 rows and 16 columns.

When we try to access a specific memory location in the RAM, the memory address is sent to the RAM, which determines which row corresponds to that memory address, and returns the data stored at that memory location.

Similarly, when trying to write data to a specific memory location in the RAM, both the memory address and the data is sent to the RAM, which determines the appropriate row, and writes the data at that memory location.

Arithmetic Logic Unit (ALU): It is the part of the CPU that performs arithmetic and logical operations on the operands stored in registers and in memory.

The E-bit in an ALU indicates whether an arithmetic operation causes a register to overflow. It the E-bit is 1 then the results must have caused the register to overflow, but if the E-bit is 0 then the results must have fit completely within the bits of the register.

Accumulators (AC0 and AC1): There are two accumulators, AC0 and AC1, of size 16 bits each. The accumulator can either be used as a destination to store the results of the operation or can be used as a source to provide the values for an operation.

We can select either of the two accumulators as a source or a destination through the S-flag and the D-flag respectively. If the S-flag or D-flag is 0, then AC0 is selected as the source and destination respectively. If the S-flag or D-flag is 1, then AC1 is selected as the source and destination respectively.

Program Counter (PC): An 8 bit register in the CPU that stores the memory address of the next instruction to be executed, starting with **00000000**. The 8 bit **Program Counter** can be added to the **Index Register** in order to access a 12 bit memory address.

Index Register (XR): An 8 bit register that stores a value that would be added to the **Program Counter**, if specified. This allows us to access 12 bit memory addresses, despite only having an 8 bit **Program Counter**.

Instruction Register (IR): A 16 bit register that temporarily holds the current instruction that has been fetched from the memory, so that it can be decoded and executed by the CPU.

Data Register (DR): A 16 bit register that temporarily stores the data being processed by the CPU for e.g. the intermediate results of arithmetic and logical operations. The data register acts as a source to provide the values for an operations.

Address Register (AR): An 8 bit register that holds a memory address, either for the instruction being performed, or for the data to be accessed.

Temporary Register (TR): A 16 bit register that temporarily stores the data being processed by the CPU for e.g. the intermediate results of arithmetic and logical operations. The temporary register acts as a source to provide the values for an operations.

Input Register (INPR): An 8 bit register that acts as a data buffer, which temporarily stores the data that is being received from an input device in a serial manner, i.e. one bit or byte at a time.

Output Register (OUTR): An 8 bit register that acts as a data buffer, which temporarily stores the data that is being transmitted to an output device in a serial manner, i.e. one bit or byte at a time.

Flags

Flag	Description
E	Specifies if register has overflowed.
S	Specifies source accumulator.
D	Specifies destination accumulator.
B	Specifies result of **Equal To** operation.
C	Specifies result of **Greater Than** operation.
F	Specifies result of **Less Than** operation.

Comparator

The comparator is a feature built upon the existing basic computer architecture that enhances the programmer's experience by providing comparisons between the two accumulators. XNOR gates help to identify the equality of the two values and a combination of AND gates and OR gates compute the **greater than** and **less than** logics as explained in the diagrams.

The results of the comparator are stored using the B, C and F flags. The B-flag is 1 if the values are equal, but 0 if the values are not equal. The C-flag is 1 if the **greater than** operation returns true, and the F-flag is 1 if the **lesser than** operation returns true.

The gate level logic is defined in **Comparator Diagram A**, and the concatenation of 2 comparators is shown in the **Comparator Diagram B**. One comparator computes the equality of 4 bits, and concatenating another comparator with it provides the equality of an 8 bit binary number.

Multiplier

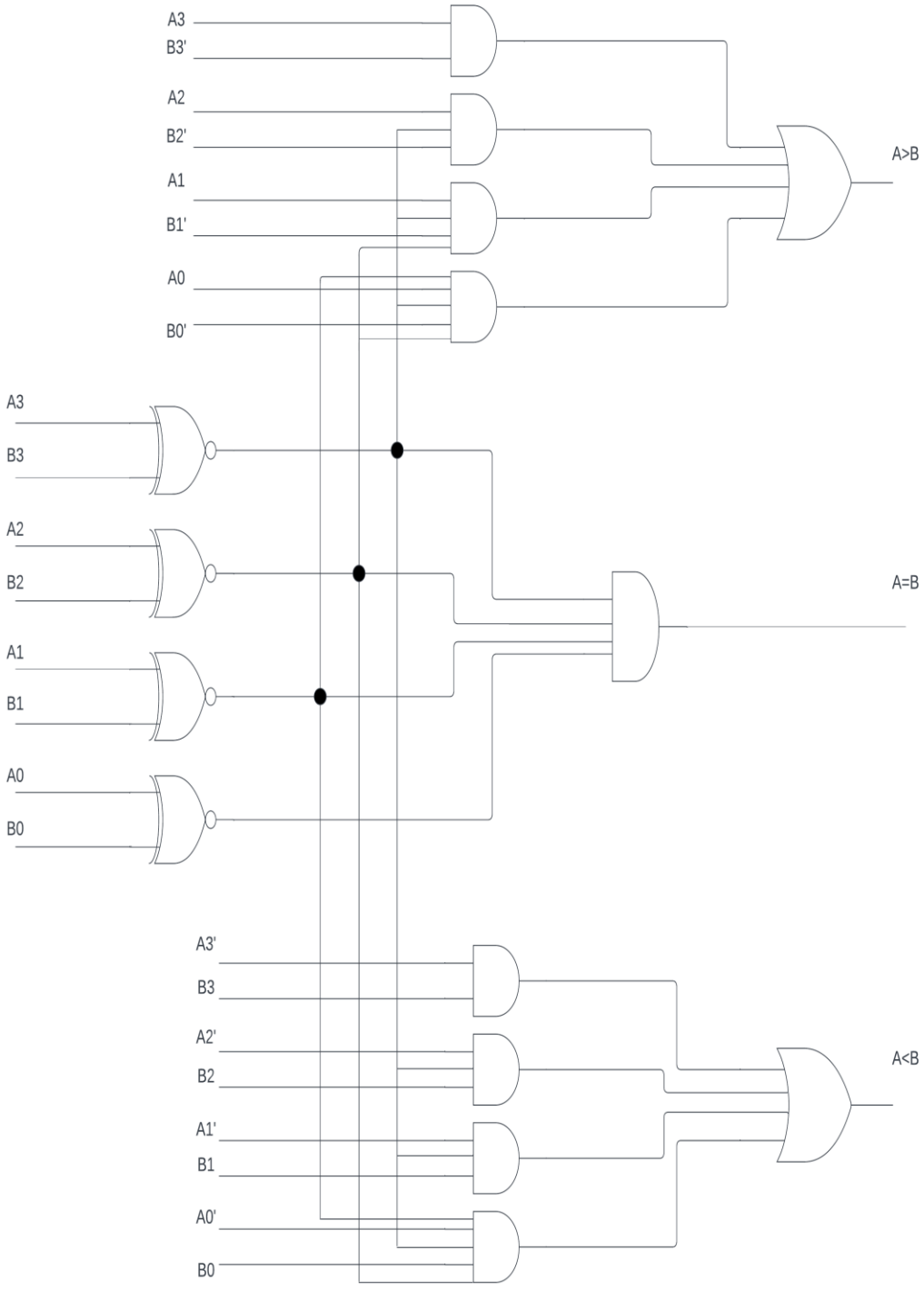
The multiplier is a feature built upon the existing basic computer architecture that enhances the programmer's experience by allowing them to perform a multiplication operation directly, rather than through the use of subprogram.

The inner working of a 4X4 Multiplier is defined in **4X4 Multiplier Diagram**, which is then further used to create an 8X8 Multiplier, the inner working of which is defined in **8X8 Multiplier Diagram**.

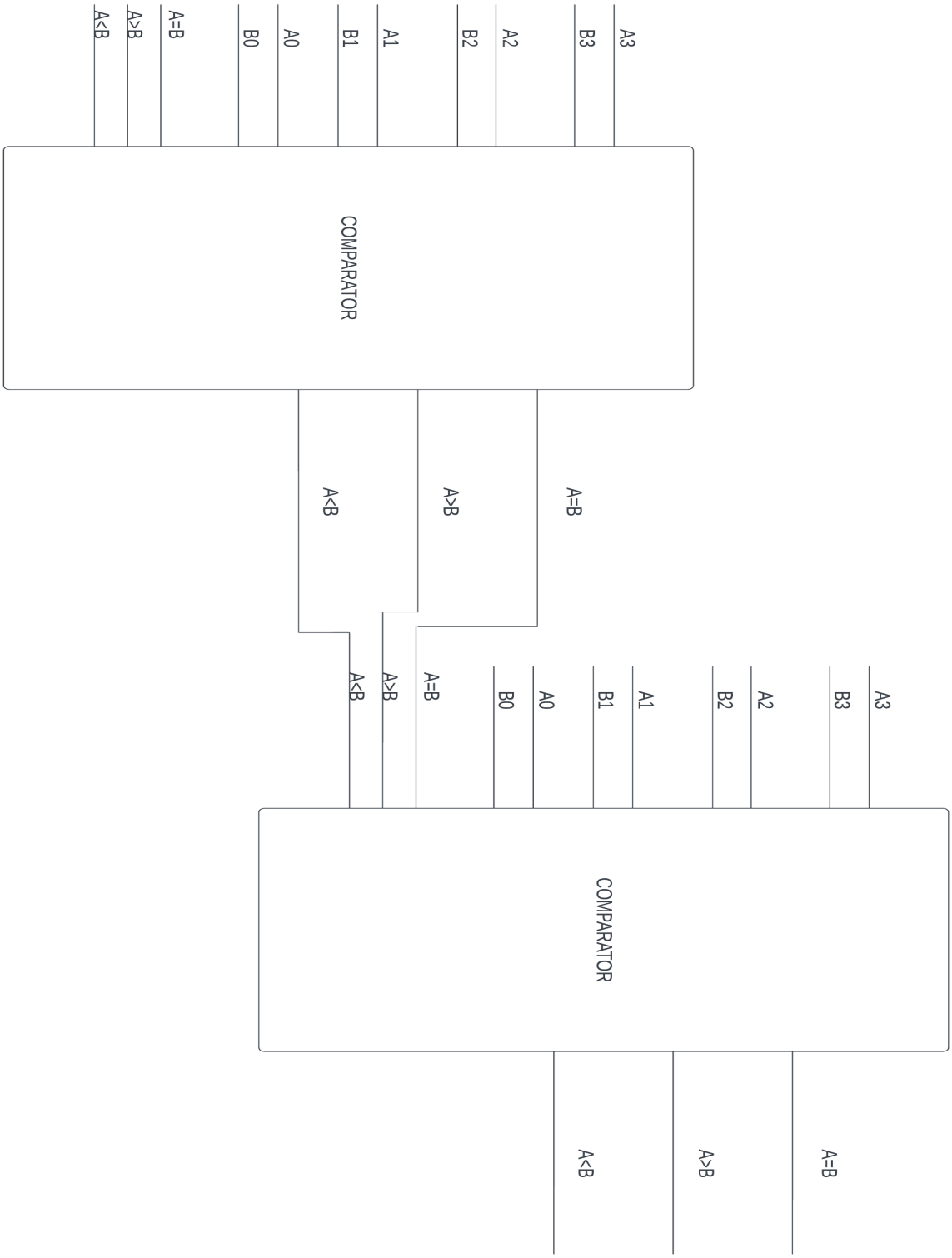
Divider

We will perform the division operation through a subroutine, rather than creating a divider to perform a division operation directly. This was done due its easier implementation.

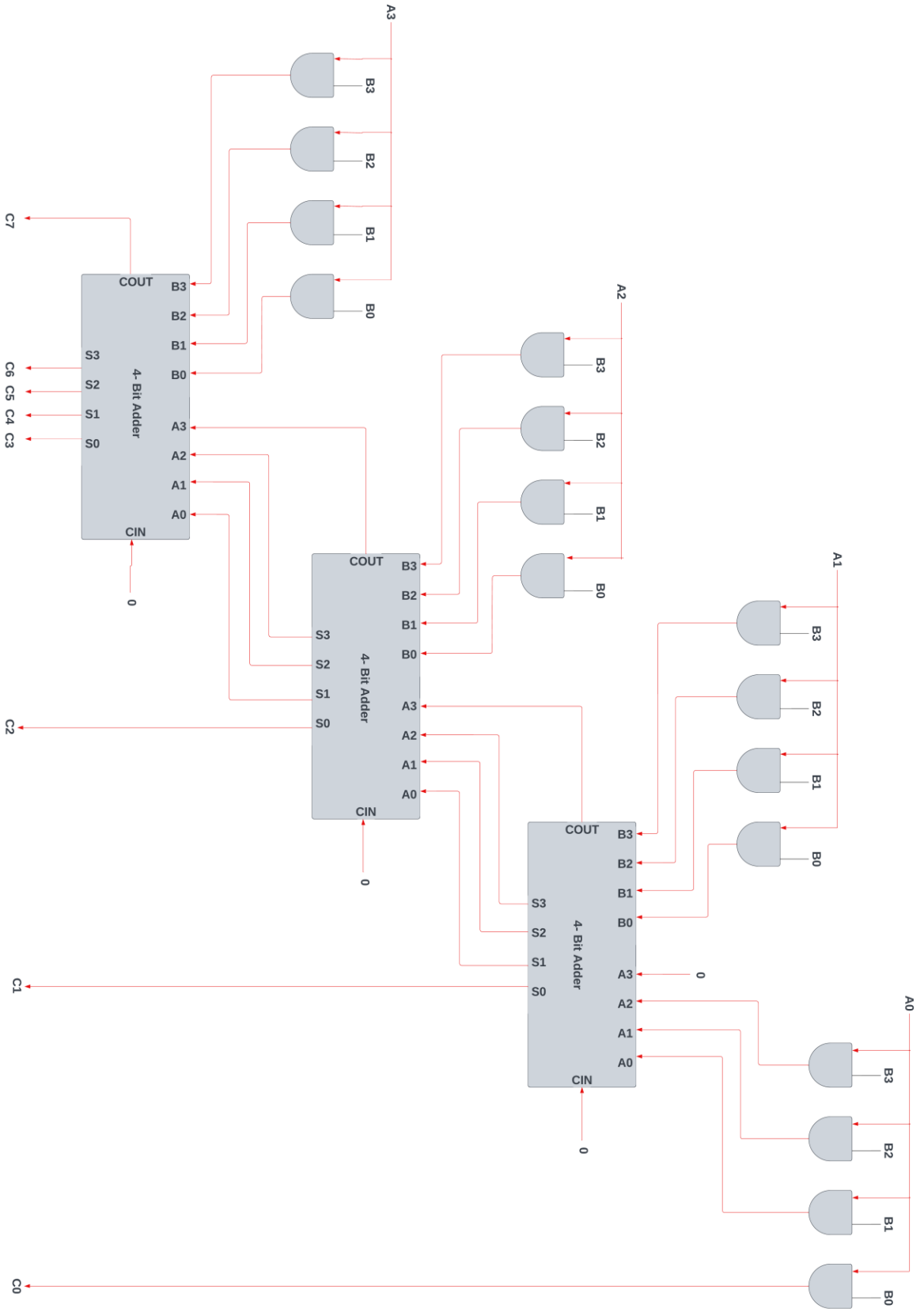
Comparator Diagram A



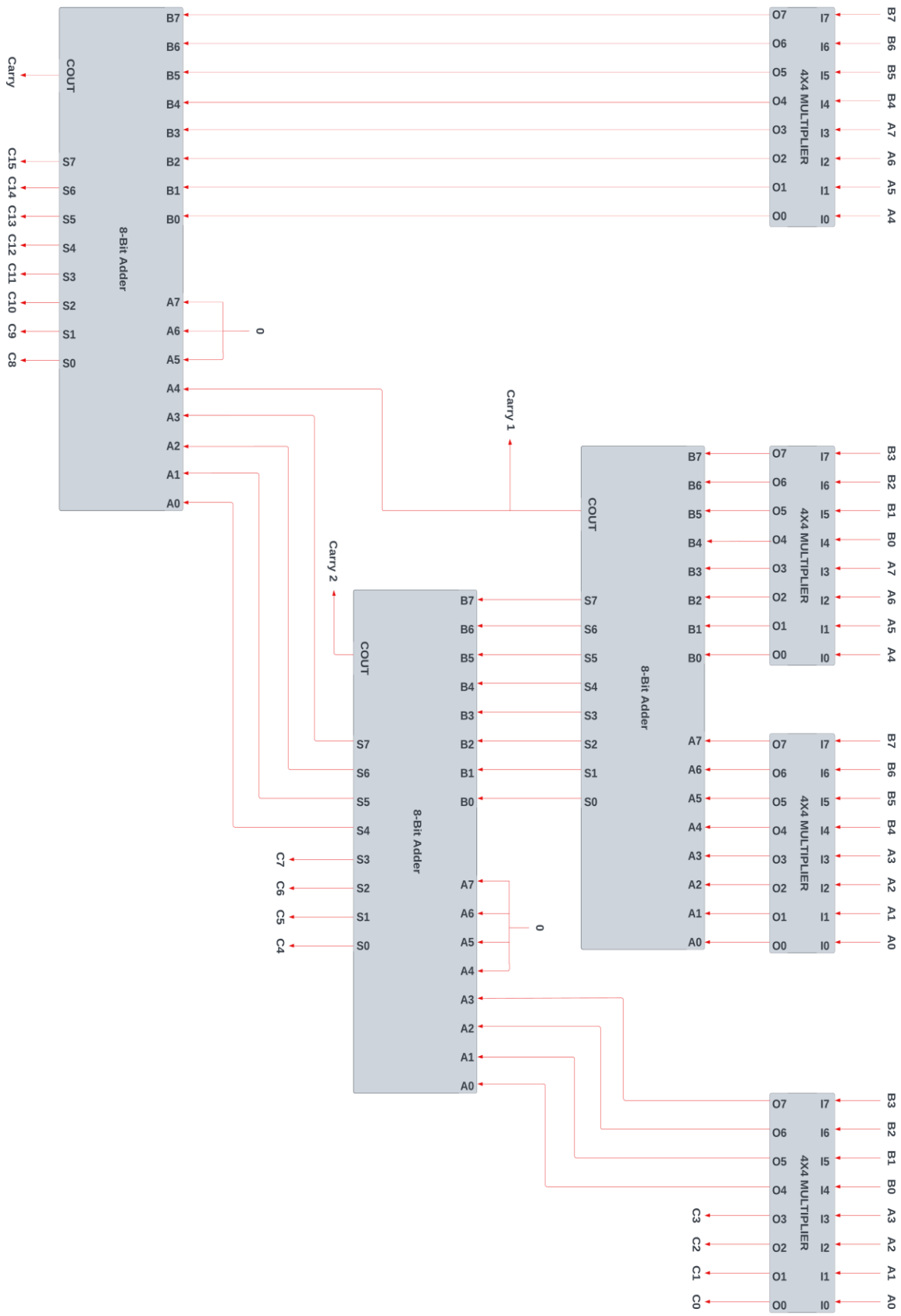
Comparator Diagram B



4X4 Multiplier Diagram



8X8 Multiplier Diagram



Instruction Set

Control Word

Addressing	Op-Code	Destination Accumulator	Source Accumulator	Memory Address
2 bits	4 bits	1 bit	1 bit	8 bits

Addressing: The first two bit of every instruction specifies the type of addressing for that instruction; 00 for direct addressing, and 01 for indirect addressing, 10 for immediate load, and 11 for using index register.

Addressing Mode	Binary Code
Direct	00
Indirect	01
Immediate	10
Indexed	11

In the case of direct addressing, the **Memory Address**, i.e. the last 8 bits, specifies the actual memory address where the data is stored. However, in the case of indirect addressing, the **Memory Address**, i.e. the last 8 bits, points to another memory address, which is the actual address of the data.

In the case of immediate addressing, the value stored in the last 8 bits is not a memory address, but is in fact a value that will be sent to a register as it is.

In the case of indexed addressing the Program Counter will be added to the Index Register, which will give us a value that points to a 12 bit memory address. The purpose for this is that our Program Counter is 8 bits long, meaning it cannot access a 12 bit memory address by itself.

Op-Code: The next 4 bits of every instruction specifies the op-code for that instruction, which in turn specifies the operation that is performed during that instruction. Every **Memory Reference Instruction** has a unique 4-bit op-code, ranging from **0000** to **1110**. However, all **Register Reference Instructions** and **I/O Reference Instructions** have

the same op-code, i.e. **1111**, with the only way to differentiate them being their addressing bits.

Destination Accumulator: The next bit of every instruction specifies which accumulator of two will act as the destination for the results of the operation. If it is 0 then we use AC0, and if it is 1 then we use AC1.

Source Accumulator: The next bit of every instruction specifies which accumulator of two will act as the source for the results of the operation. If it is 0 then we use AC0, and if it is 1 then we use AC1.

Memory Address: The last 8 bits specifies the memory address, which will be then be added to the **Index Register** to obtain the actual 12 bit memory address.

Instructions

Memory Reference Instructions: A type of instruction that accesses and manipulates **(i.e. reads or writes)** data stored in a memory location. The Op-Codes for **Memory Reference Instructions** range from **0000** to **1110**, and can be performed through either direct or indirect addressing.

Register Reference Instructions: A type of instruction that accesses and manipulates **(i.e. performs arithmetic and logical operations)** data stored in a register. The addressing bit for **Register Reference Instructions** is always 00, and the Op-Code is always **1111**, meaning the only way to differentiate between them is through their **Memory Address**, which is unique for each instruction.

Input/Output Reference Instructions: A type of instruction that transfers data between the CPU and external devices **(for both input and output)**, such as keyboards and monitors. The addressing bit for **Register Reference Instructions** is always 11, and the Op-Code is always **1111**, meaning the only way to differentiate between them is through their **Memory Address**, which is unique for each instruction.

Symbol	Op-Code	Description
Memory Reference Instructions		
LDD	0000	**LOAD:** Loads a value from a specified memory location into a specified register.
STR	0001	**STORE:** Stores a value from a specified register into a specified memory location.
STX	0010	**STORE IN XR:** Stores the immediate value, or a value from a specified memory location, in the Index Register.
ADD	0011	**ADD:** Adds the values in two specified registers, and stores the result in a third specified register.
SUB	0100	**SUBTRACT:** Subtracts the values in two specified registers, and stores the result in a third specified register.
MUL	0101	**MULTIPLY:** Multiplies the values in two specified registers, and stores the result in a third specified register.
DIV	0110	**DIVIDE:** Divides the values in two specified registers, and stores the result in a third specified register.
AND	0111	**ADD:** Performs logical **AND** operation on the values in two specified registers, and stores the result in a third specified register.
OR	1000	**OR:** Performs logical **OR** operation on the values in two specified registers, and stores the result in a third specified register.
XOR	1001	**EXCLUSIVE OR:** Performs logical **XOR** operation on the values in two specified registers, and stores the result in a third specified register.
JMP	1010	**JUMP:** Jumps unconditionally to the instruction at the specified memory address.

JEQ	1011	**JUMP IF EQUAL:** Jumps to the instruction at the specified memory address, if the value in two specified registers are equal.
JNE	1100	**JUMP IF NOT EQUAL:** Jumps to the instruction at the specified memory address, if the value in two specified registers are not equal.
JGT	1101	**JUMP IF GREATER THAN:** Jumps to the instruction at the specified memory address, if the value in one specified register is greater than the value in another specified register.
JLT	1110	**JUMP IF LESS THAN:** Jumps to the instruction at the specified memory address, if the value in one specified register is less than the value in another specified register.
Register Reference Instructions		
MOV	1111	**MOVE:** Moves a value from one specified register to another.
INC	1111	**INCREMENT:** Increases the value in a specified register by 1.
DEC	1111	**DECREMENT:** Decreases the value in a specified register by 1.
CMP	1111	**COMPARE:** Compares the values in two specified registers, and sets a flag register to indicate the result.
NOT	1111	**NOT:** Performs logical **NOT** operation the value in a specified register, and stores the result in another specified register.
SHL	1111	**SHIFT LEFT:** Shifts the bits in a specified register left by a specified number of bits.
SHR	1111	**SHIFT RIGHT:** Shifts the bits in a specified register right by a specified number of bits.
CLA	1111	**CLEAR AC:** Sets specified accumulator to zero.
CLE	1111	**CLEAR E:** Sets E of specified accumulator to zero.

CLB	1111	**CLEAR B:** Clears B flag.
CLC	1111	**CLEAR C:** Clears C flag.
CLF	1111	**CLEAR F:** Clears F flag.
SPA	1111	**SKIP IF POSITIVE AC:** Skips the instruction in the next memory location, if the value in accumulator is positive.
SNA	1111	**SKIP IF NEGATIVE AC:** Skips the instruction in the next memory location, if the value in accumulator is negative.
SZA	1111	**SKIP IF ZERO AC:** Skips the instruction in the next memory location, if the accumulator is zero.
HLT	1111	**HALT:** Halts computer.
Input/Output Reference Instructions		
INP	1111	Input character to specified accumulator
OUT	1111	Output character from specified accumulator.
SKI	1111	Skip on input flag.
SKO	1111	Skip on output flag.
ION	1111	Interrupt on.
IOF	1111	Interrupt off.

Micro-Operations

At memory location **00000000**, the Program Counter starts looking for instructions. When an instruction is detected, its address is stored in the Address Register, while instruction itself is stored in the Instruction Register. In the next step, the PC gets incremented by 1 in order to get the address of the next instruction that is to be executed.

The Program Counter is only capable of storing 8 bits, but our RAM can contain 12 bit memory addresses. If we wish to access any memory address beyond 8 bits, we add the Program Counter to the Index Register, which gives us a new value than points to a memory address that is larger than 8 bits.

Through these micro-operations, the CPU fetches the next instruction from the Program Counter, which is then updated to point to the next instruction from the memory.

The CPU then decodes the 16-bit instruction it receives, according to the defined control word. The instruction is split into the Addressing Bit **(2 bits)**, the Op-Code **(4 bits)**, the Destination Accumulator **(1 bit)**, the Source Accumulator **(1 bit)**, and the Memory Address **(8 bits)**.

Depending on the Op-Code in the decoded instruction, the following micro-operations would then be executed:

Symbol	Instruction	Micro-Operations
-	Fetch	**R`TO:** AR <= PC;  **R`T1:** IR <= M[AR], PC <= PC + 1;
-	Decode	**R`T2:** D0-D16 <= Decode IR(10-13), AR <= IR(0-7), I0-I3 <= Decode IR(14-15), D <= IR(9), S <= IR(8);
-	Indirect	**I1T3:** AR <= M[AR];
-	Interrupt	**T0`T1`T2`(IEN)(FGI+FGO):** R <= 1; **RT0:** AR <= 0;, TR <= PC; **RT1:** M[AR] <= TR, PC <= 0; **RT2:** PC <= PC + 1, IEN <= 0, R <= 0, SC <= 0;
Memory Reference Instructions		
LDD **(Direct)**	00 0000 XX XXXXXXXX	**D0I0T4:** DR <= M[AR]; **DOI0D`T5:** AC0 <= DR, SC <= 0; **DOI0DT5:** AC0 <= DR, SC <= 0;
LDD **(Immediate)**	10 0000 XX XXXXXXXX	**D0I2T4:** DR <= M[AR]; **DOI2D`T5:** AC0 <= DR, SC <= 0; **DOI2DT5:** AC0 <= DR, SC <= 0;
LDD **(Indexed)**	11 0000 XX XXXXXXXX	**D0I3D`T4:** AR <= XR + AR; **D0I3D`T5:** DR <= M[AR]; **DOI3D`T6:** AC0 <= DR, SC <= 0; **DOI3DT6:** AC1 <= DR, SC <= 0;

STR	XX 0001 XX XXXXXXXX	**D1D`T4:** M[AR] <= AC0, SC <= 0; **D1DT4:** M[AR] <= AC1, SC <= 0;
STX	XX 0010 XX XXXXXXXX	**D14D`T4:** XR <= AC0, SC <= 0; **D14DT4:** XR <= AC1, SC <= 0;
ADD	XX 0011 XX XXXXXXXX	**D3T4:** DR <= M[AR]; **D3D`S`T5:** AC0 <= AC0 + DR, E <= Cout, SC <= 0; **D3D`ST5:** AC0 <= AC1 + DR, E <= Cout, SC <= 0;  **D3DS`T5:** AC1 <= AC0 + DR, E <= Cout, SC <= 0; **D3DST5:** AC1 <= AC1 + DR, E <= Cout, SC <= 0;
SUB	XX 0100 XX XXXXXXXX	**D4T4:** DR <= M[AR]; **D4D`S`T5:** AC0 <= AC0 – DR, E <= Cout, SC <= 0; **D4D`ST5:** AC0 <= AC1 – DR, E <= Cout, SC <= 0;  **D4DS`T5:** AC1 <= AC0 – DR, E <= Cout, SC <= 0; **D4DST5:** AC1 <= AC1 – DR, E <= Cout, SC <= 0;

MUL	XX 0101 XX XXXXXXXX	**D5T4:** DR <= M[AR]; **D5D`S`T5:** AC0 <= AC0 * DR, SC <= 0; **D5D`ST5:** AC0 <= AC1 * DR, SC <= 0; **D5DS`T5:** AC1 <= AC0 * DR, SC <= 0; **D5DST5:** AC1 <= AC1 * DR, SC <= 0;
DIV	XX 0110 XX XXXXXXXX	**D6T4:** DR <= M[AR]; **D6D`S`T5:** AC0 <= AC0 / DR, SC <= 0; **D6D`ST5:** AC0 <= AC1 / DR, SC <= 0; **D6DS`T5:** AC1 <= AC0 / DR, SC <= 0; **D6DST5:** AC1 <= AC1 / DR, SC <= 0;
AND	XX 0111 XX XXXXXXXX	**D7T4:** DR <= M[AR]; **D7D`S`T5:** AC0 <= AC0 ∧ DR, SC <= 0; **D7D`ST5:** AC0 <= AC1 ∧ DR, SC <= 0; **D7DS`T5:** AC1 <= AC0 ∧ DR, SC <= 0; **D7DST5:** AC1 <= AC1 ∧ DR, SC <= 0;
OR	XX 1000 XX XXXXXXXX	**D8T4:** DR <= M[AR]; **D8D`S`T5:** AC0 <= AC0 ∨ DR, SC <= 0; **D8D`ST5:** AC0 <= AC1 ∨ DR, SC <= 0; **D8DS`T5:** AC1 <= AC0 ∨ DR, SC <= 0; **D8DST5:** AC1 <= AC1 ∨ DR, SC <= 0;

XOR	XX 1001 XX XXXXXXXX	**D9T4:** DR <= M[AR]; **D9D`S`T5:** AC0 <= AC0 ⊕ DR, SC <= 0; **D9D`ST5:** AC0 <= AC1 ⊕ DR, SC <= 0;  **D9DS`T5:** AC1 <= AC0 ⊕ DR, SC <= 0; **D9DST5:** AC1 <= AC1 ⊕ DR, SC <= 0;
JMP	XX 1010 XX XXXXXXXX	**D10T4:** M[AR] <= PC, AR <= AR + 1; **D10T5:** PC <= AR, SC <= 0;
JEQ	XX 1011 XX XXXXXXXX	**D11T4: if (**B = 1**) then** PC <= AR;
JNE	XX 1100 XX XXXXXXXX	**D12T4: if (**B = 0**) then** PC <= AR;
JGT	XX 1101 XX XXXXXXXX	**D13T4: if (**C = 1**) then** PC <= AR;
JLT	XX 1110 XX XXXXXXXX	**D14T4: if (**F = 1**) then** PC <= AR;
Register Reference Instructions		
-	-	**D15I0T3 = r** **IR(i) = Bi** (i = 0, 1, 2,. . ., 7) **r:** SC <= 0;
MOV	00 1111 XX 00000001	**rB7`B0DS`T4:** TR <= AC0; **rB7`B0DS`T5:** AC1 <= TR; **rB7`B0D`ST4:** TR <= AC1; **rB7`B0D`ST5:** AC0 <= TR;
INC	00 1111 **XX** 00000010	**rB7`B1D`:** AC0 <= AC0 + 1; **rB7`B1D:** AC1 <= AC1 + 1;

DEC	00 1111 XX 00000100	**rB7`B2D`:** AC0 <= AC0 – 1; **rB7`B1D:** AC1 <= AC1 + 1;
CMP	00 1111 XX 00001000	**rB7`B3:** B, C, F <= AC0 **CMP** AC1;
NOT	00 1111 XX 00010000	**rB7`B4D`:** AC0 <= ~AC0; **rB7`B4D:** AC1 <= ~AC1;
SHL	00 1111 XX 00100000	**rB7`B4D`:** AC0 <= shl AC0; **rB7`B4D:** AC1 <= shl AC1;
SHR	00 1111 XX 01000000	**rB7`B4D`:** AC0 <= shr AC0; **rB7`B4D:** AC1 <= shr AC1;
CLA	00 1111 XX 10000000	**rB7D`:** AC0 <= 0; **rB7D:** AC1 <= 0;
CLE	00 1111 XX 10000001	**rB7B0:** E <= 0;
CLB	00 1111 XX 10000010	**rB7B1:** B <= 0;
CLC	00 1111 XX 10000100	**rB7B2:** C <= 0;
CLF	00 1111 XX 10001000	**rB7B3:** F <= 0;

SPA	00 1111 XX 10010000	**rB7B4D`: if (**AC0(15) = 0**) then** PC <= PC + 1; **rB7B4D: if (**AC1(15) = 0**) then** PC <= PC + 1;
SNA	00 1111 XX 10100000	**rB7B5D`: if (**AC0(15) = 1**) then** PC <= PC + 1; **rB7B5D: if (**AC1(15) = 1**) then** PC <= PC + 1;
SZA	00 1111 XX 11000000	**rB7B6D`: if (**AC0 = 0**) then** PC <= PC + 1; **rB7B6D: if (**AC1 = 0**) then** PC <= PC + 1;
HLT	00 1111 XX 11000001	**rB7B6B5:** SC <= 0;
Input / Output Reference Instructions		
-	-	**D15I3T3 = p** **IR(i) = Bi** (i = 0, 1, 2,. . ., 5) **p:** SC <= 0;
INP	11 1111 XX 00000001	**pB0D`:** AC0(0-7) <= INPR, FGI <= 0; **pB0D:** AC1(0-7) <= INPR, FGI <= 0;
OUT	11 1111 XX 00000010	**pB1D`:** OUTR <= AC0[0-7], FGO <= 0;  **pB1D`:** OUTR <= AC1[0-7], FGO <= 0;
SKI	11 1111 XX 00000100	**pB2: if (**FGI = 1**) then** PC <= PC + 1;
SKO	11 1111 XX 00001000	**pB3: if (**FGO = 1**) then** PC <= PC + 1;
ION	11 1111 XX 00010000	**pB4:** IEN <= 1;
IOF	11 1111 XX 00100000	**pB5:** IEN <= 0;

Bibliography

- Lecture Slides

- https://www.geeksforgeeks.org/magnitude-comparator-in-digital-logic/

- https://www.geeksforgeeks.org/computer-organization-instruction-formats-zero-one-two-three-address-instruction/

- https://www.geeksforgeeks.org/common-bus-system/