2022

Ghulam Ishaq Khan Institute of Engineering Sciences and Technology

Design and Model of the Basic & Automatic Computing Engine (BACE)

__Submitted To:__

Dr. Ghulam Abbas (Course Instructor)

__Submitted By:__
Muhammad Farae (2020292)
Muhammad Saim Iqbal (2020348)
Muhammad Umer Asif (2020372)
Syed Zaeem Shakir (2020487)
Uzair Rahman (2020509)

__Date of Submission:__ January, 2022

COURSE: COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE
COURSE CODE: CE222
FACULTY OF COMPUTER SCIENCE (FCSE)

Acknowledgements

The authors of this report would like to extend their gratitude and acknowledge the support provided graciously to us by Dr. Ghulam Abbas without whom this report would not have been possible. The report has enabled the authors to not only understand the material imparted in the course fully, but also to critically analyze such material and make any design, implementation and selection changes to the ends of fabricating a more efficient computer organizational model. Overall, the project was a joy to partake and the team has utilized the opportunity fully to learn as much as possible.

Table of Contents

Figure Listings

1. Abstract

In this report, the designing process imparted to us in the course CE222-Computer Organization and Assembly Language is followed insofar as to construct the logical foundations of a structurally sound and functional Central Processing Unit (CPU) followed by the complete instruction set definition. The report follows the chronological steps of: designing the bus structure, ALU, and CU; selecting the necessary and appropriate registers and memory configurations; defining the instruction set and format of the BACE processor from a repertoire of options and customizations. Each section details the purpose and theoretical background behind each component whilst also keeping in view the organizational aspect through extensive use of block diagrams and flowcharts. Moreover, the new additions made in the BACE processor as compared to the original Basic Computer have been detailed and justified for their inclusion and need, and as required, their working has been further elaborated through subsections dedicated to explaining their functionality. The crux of the report is the enhanced ALU and increased number of instructions in the instruction set. Lastly, the representations of the computer microperations in the Complete Computer Description were facilitated through the use of Register Transfer Language (RTL).

2. Theory and Design

2.1 Objectives

To the ends of implementing a logical CPU design with exhaustive operational capabilities, the four main features of any digital computer system as enlisted below and encapsulated in Fig. 1:

- Data Transfer and Communication
- Short-Term and Long-Term Data Storage
- Data Processing
- Data Control Function

For a complete computer description, the design process has incorporated manifold functionality by taking inspiration from the Basic Computer such as a completely redesigned processor with enhanced arithmetic & logic operations, register organization, bus structure and input/output procedures. To facilitate data movement, the logical design of the bus has been elaborated with multiple addressing modes. For the purpose of storage, processor registers and main memory has been defined appropriately. Data processing has been defined through the description of a complete Instruction Set Architecture and achieved through the ALU; moreover, to accommodate and control all such requirements of the BACE processor, an original Control Unit has been implemented with detailed breakdown of each of its components in a hierarchical fashion.

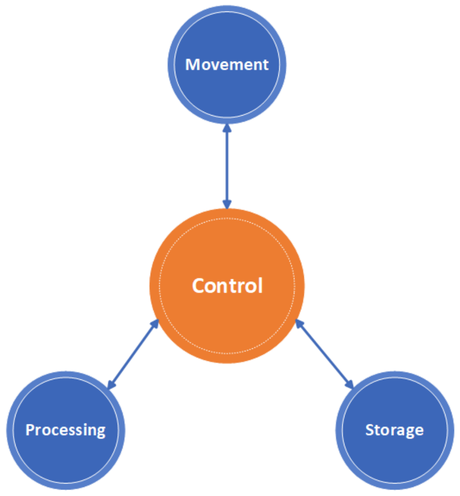
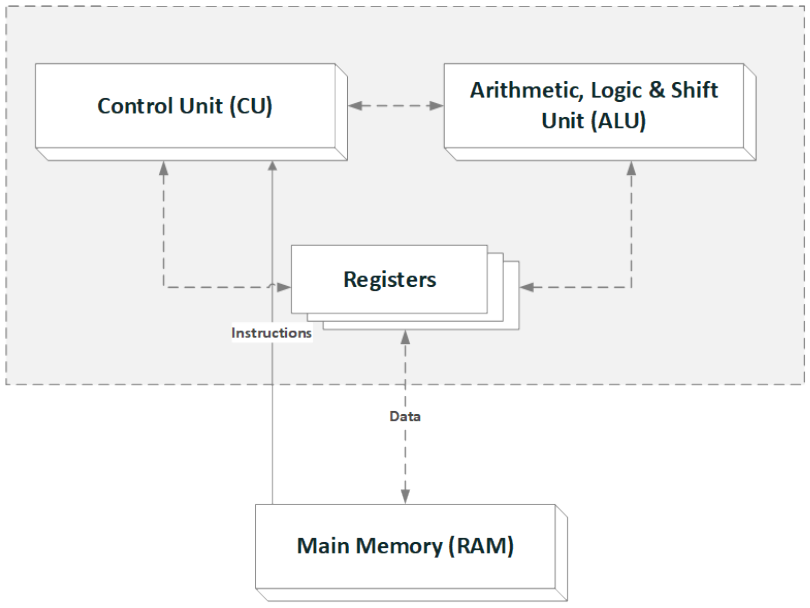


Figure 1-Four Operational Attributes of BACE

2.2 Theoretical Background:

In this section, some prerequisite concepts pertaining to the design of the BACE processor are uncovered. In essence, the BACE processor among numerous other processors realizes the use of the general-purpose Von Neumann Architecture as portrayed in Fig. 2. Although the Harvard Architecture is well-known for its unique features such as a separation in storage and movement between data and instructions unlike the von Neumann architecture which relies on stored program computers (i.e. data and instructions stored in the same memory medium), the latter was preferred owing to its simplicity in designing, modeling and implementation.

Figure 2-General Purpose von Neumann Architecture



The remainder of the report will follow a top-down approach and dissect the internal workings of each of these four components, namely the Arithmetic Logic & Shift Unit (ALU), Control Unit (CU), and the manifold registers to be used by the aforementioned two units for the purpose of data processing and short-term storage. By the same token, the size and overall configuration of the main memory must also be defined for the purpose of long-term data and instructions storage.

2.2.1 Noteworthy Assumptions

Before proceeding, it is imperative to note some of the assumptions that have been made during the fabrication of this report. Firstly, if a transfer of data between two registers is shown by an RTL statement A $\leftarrow$ B where ‘A’ and ‘B’ are registers, it is assumed for all such cases that the necessary intermediate data bus connections have already been established and have not been mentioned explicitly. Moreover, it must also be noted that all such statements succeeding in the report assume parallel load capacity of the registers. As for the clock signal, it is assumed that the microperations are executed under one clock pulse and are positive-edge triggered.

2.3 Definition of the BUS Network

Due to the immense hardware and computational overhead caused by a communication system in which each register connects to every other register, requiring n(n-1) total wire connections internally, it was construed to design the overall structure of a common bus system that would centralize majority of the transfer of data that occurs in the BACE processor from any source to all possible destinations.

The BACE processor makes use of decoders to implement the bus system and combines the use of three-state gates, both of which are briefly elaborated below. Unlike the Basic Computer and its implementation of bus structure, our communication system has managed to reduce the amount of hardware required for implementing the same functionality.

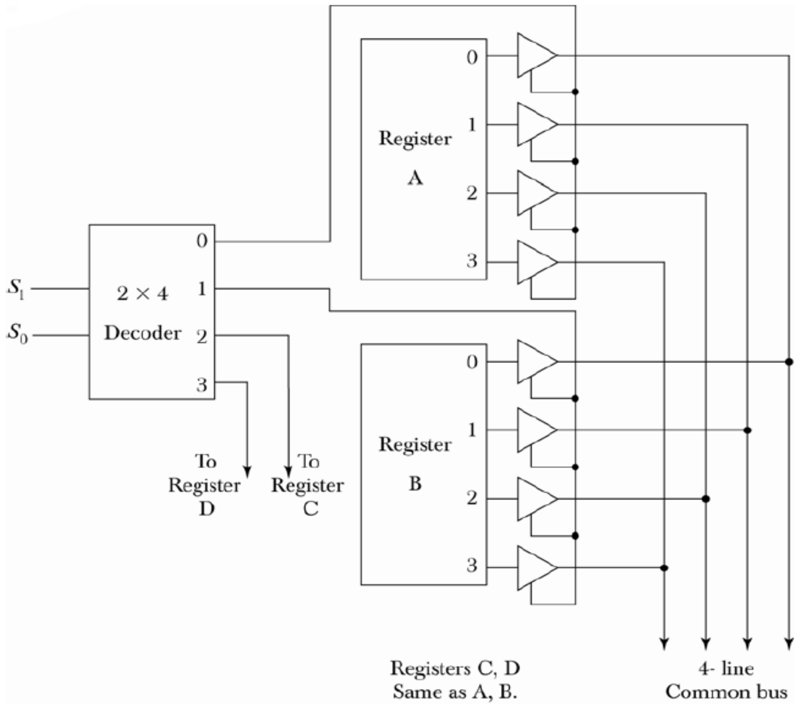


Figure 3-Bus Structure using Decoder

The Basic Computer incorporated the use of 'n' decoders for n-bit registers and $\log_2 m\ by\ m$ was the size of each decoder for 'm' registers. Furthermore, each decoder output was connected to three-state gates in order to ensure only one bit of the corresponding register was selected and passed onto the bus system by the decoder at any given time. On the other hand, the BACE processor decreases the number of decoders required to one, which has the dimensions same $\log_2 m\ by\ m$ but the three-state gates now connect to every bit of the registers directly as shown in Fig. 3.

By using a decoder with same number of output enable pins as number of registers and attaching the three-state gates with the registers instead of the decoder, a slight reorganization eliminated the need of multiple decoders in one bus connection. The internal working of the decoder and three-state gate as well as their truth table are given in Appendices A and B respectively.

An alternative implementation using multiplexers was also considered as shown in Fig. 4. For a sample system of 4 registers with 4-bits each, we considered using a quad multiplexer with $\log_2 8 = 3$ select lines and 4 separate quad-inputs; the benefit of this structure was that it only required one 4by1 quad multiplexer but it was quickly realized implementing a single MUX to multiplex all registers in BACE processor would make the multiplexer circuit comparatively too large. An alternative structure is shown in Fig. 5.

It must be noted that in Fig. 3, Fig. 4 and Fig. 5, the actual sizes of the registers in BACE processor were not used and instead assumed to be 4-bits each for the sake of simplification. In the succeeding sections, such characteristics are defined exhaustively. For the information of the reader, it is mentioned beforehand that address registers typically have smaller sizes than data storing registers, and thus a default binary value of 0 was passed into each input pin of the decoder and multiplexer starting from the most significant bit which were not included in the size of the address registers but were present in the bus.

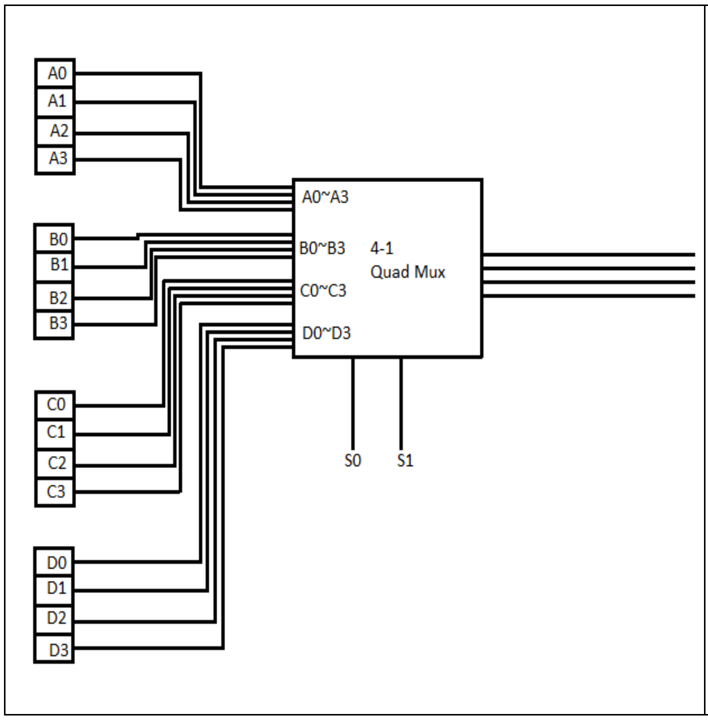


Figure 4-Bus Structure using a single quad multiplexer

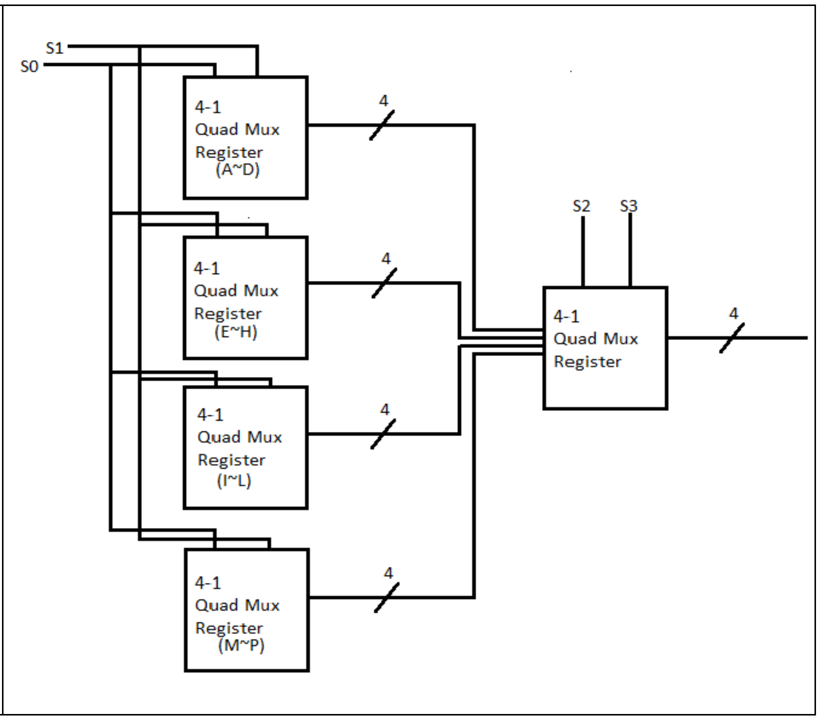

S1
S0
4-1 Quad Mux Register (A~D)
4-1 Quad Mux Register (E~H)
4-1 Quad Mux Register (I~L)
4-1 Quad Mux Register (M~P)
4
4
4
4
S2
S3
4-1 Quad Mux Register
4

Figure 5-Bus Structure using cascaded quad multiplexers

2.4 Arithmetic Logic & Shift Unit (ALU)

This section continues and completes the discussion on all the types of microoperations that can be performed by the hardwired components of the BACE processor, namely the transfer, arithmetic, logical, and shift microoperations. The previous sections covered the transfer microoperations by defining the bus interconnections. The remaining three categories will be implemented entirely by the Arithmetic Logic & Shift Unit (i.e. ALU).

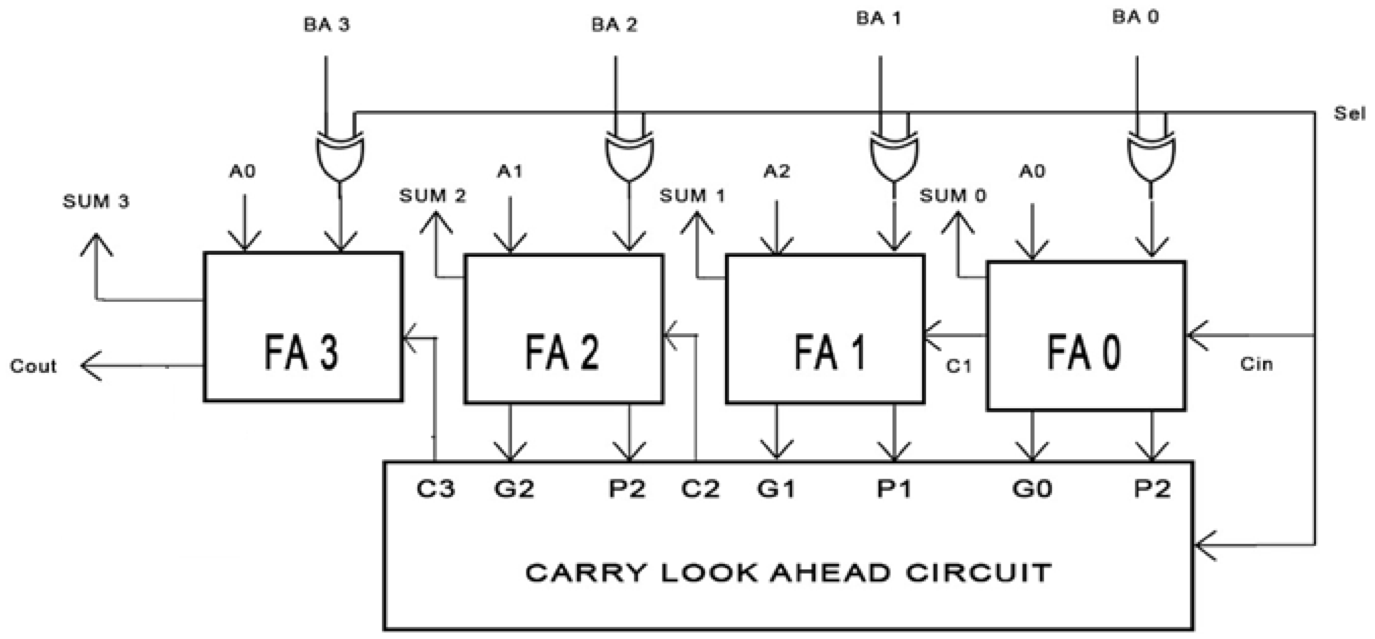
2.4.1 Arithmetic Microoperations

In this section, the most basic arithmetic microoperations performed by the ALU are summarized, followed by a discussion on a single composite arithmetic circuit that hardwires all such implementation into one unit.

The BACE processor implements the typical arithmetic operations such as increment, decrement, addition and subtraction through the use of full-adders. Fig. 6 shows the adder-subtractor circuit alongside its truth table whereas Fig. 7 portrays the look-ahead logic. Full-adder can be derived from the connection of two half-adders. In other words, the increment operation, which would usually be performed by the cascading of as many half-adders as the number of bits in each register, can also be performed by full-adders if appropriate configuration is achieved. Therefore there are no standalone half-adder circuits in the BACE processor.

The full-adder circuit was designed to create the adder-subtractor circuit to decrease the number of components required to create a binary subtractor independently from the binary adder. Depending on value of Cin, addition or subtraction is performed (if cin is 0, normal addition and if it's 1, the two's complement is applied on the addend input for subtraction. Furthermore, as an extension of the adder-subtractor used in the Basic Computer, the BACE processor counterpart also contains a look-ahead circuit in conjunction. This enables the arithmetic calculations to be done in parallel through carry generation and propagation instead of the carry rippling through each full-adder to the next sequentially in a cascaded manner, making the execution time of the BACE processor's adder-subtractor more efficient.

Figure 6-Adder-Subtractor Circuit with Look-Ahead Logic



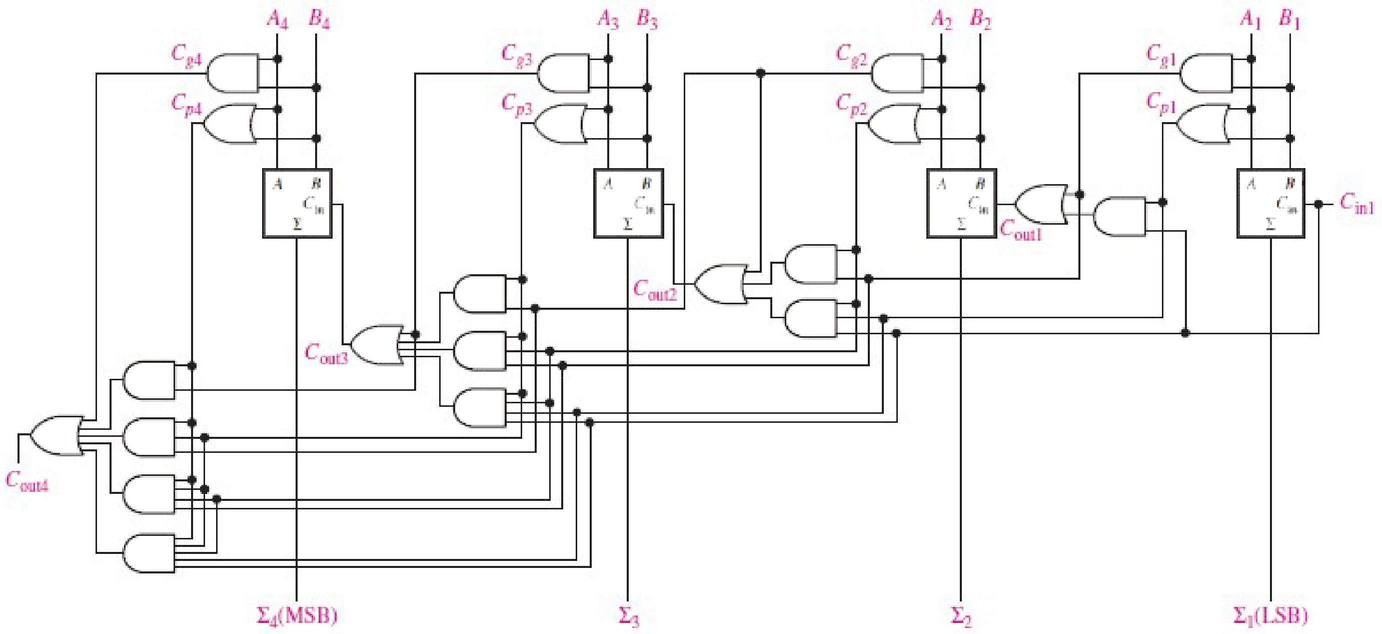


Figure 7- Look-Ahead Logic Circuit Internal Working

Full-adder 1:

$Cout_1 = Cg_1 + Cp_1Cin_1$

Full-adder 2:

$Cin_2 = Cout_1$
$Cout_2 = Cg_2 + Cp_2Cin_2 = Cg_2 + Cp_2Cout_1 = Cg_2 + Cp_2(Cg_1 + Cp_1Cin_1)$
$= Cg_2 + Cp_2Cg_1 + Cp_2Cp_1Cin_1$

Full-adder 3:

$Cin_3 = Cout_2$
$Cout_3 = Cg_3 + Cp_3Cin_3 = Cg_3 + Cp_3Cout_2 = Cg_3 + Cp_3(Cg_2 + Cp_2Cg_1 + Cp_2Cp_1Cin_1)$
$= Cg_3 + Cp_3Cg_2 + Cp_3Cp_2Cg_1 + Cp_3Cp_2Cp_1Cin_1$

Full-adder 4:

$Cin_4 = Cout_3$
$Cout_4 = Cg_4 + Cp_4Cin_4 = Cg_4 + Cp_4Cout_3$
$= Cg_4 + Cp_4(Cg_3 + Cp_3Cg_2 + Cp_3Cp_2Cg_1 + Cp_3Cp_2Cp_1Cin_1)$
$= Cg_4 + Cp_4Cg_3 + Cp_4Cp_3Cg_2 + Cp_4Cp_3Cp_2Cg_1 + Cp_4Cp_3Cp_2Cp_1Cin_1$

Overall, maximum functionality can be achieved from the look-ahead adder-subtractor circuit by using a modified composite arithmetic circuit as shown in Fig. 8.

The table below encapsulates the overall working of the circuit. Although eight total microoperations can be executed, the transfer procedure is repeated twice, and thus there are seven distinct microoperations that can be performed by the arithmetic circuit.

Selection S_1	S_0	Cin	Selected Input	Output = A + Selected Input + Cin	Microoperation
0	0	0	B	D = A + B	Addition
0	0	1	B	D = A + B + 1	Increment and addition
0	1	0	B’	D = A + B’	Subtraction with borrow
0	1	1	B’	D = A + B’ + 1	Subtraction
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A (Both Inc. & Dec.)

All the above basic arithmetic operations can be implemented through a unified composite arithmetic circuit that implements the logic of the adder-subtractor, incrementer, and decremented in one combinational logic unit. A single composite arithmetic circuit unit will perform all such operations on a single bit and n circuits will be connected in series for operations on n-bit register inputs.

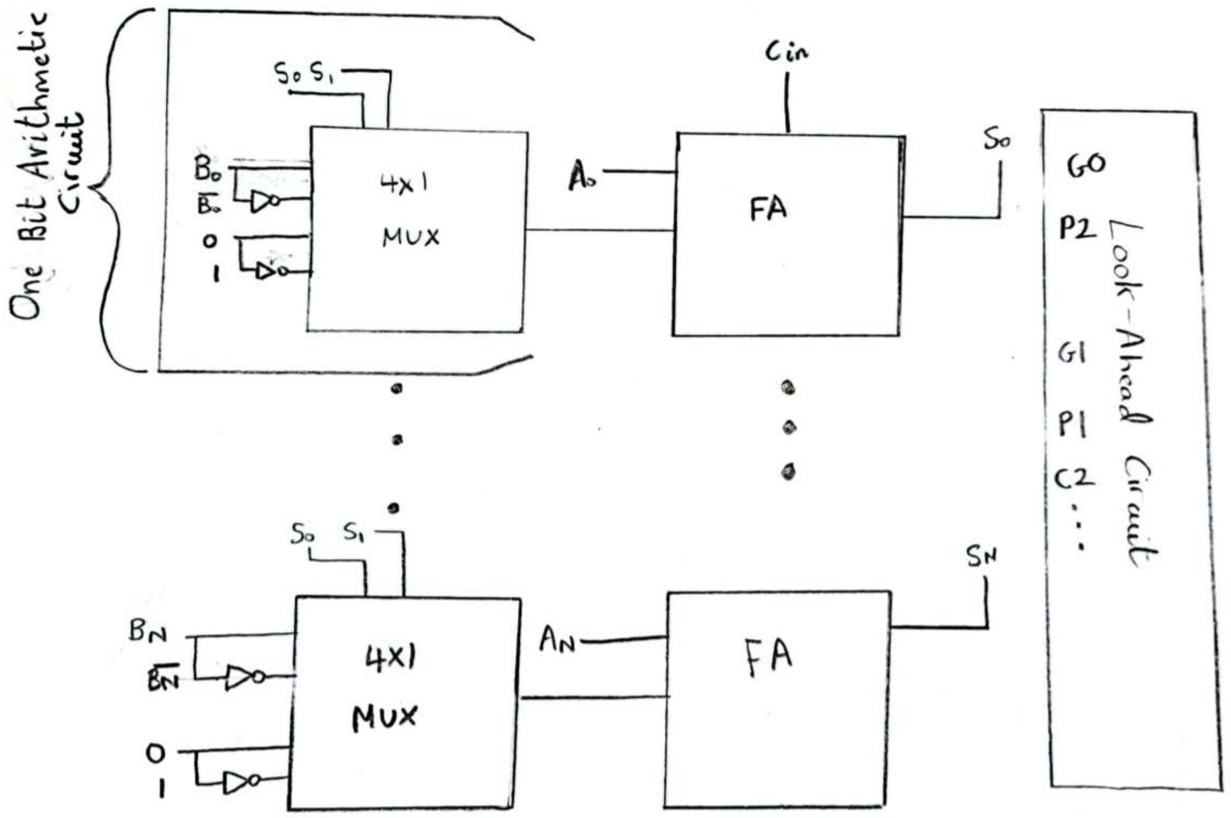


Figure 8-Arithmetic Circuit

2.4.2 Logic Microoperations

The logic unit contains eight gates instead of the previous four to increase the number of logic microoperations that can be performed within one clock cycle. Fig. 9 below showcases the logic unit. As shown, pins A and pin B connect separately to NOT gates. This means if the value at pin B needs to be inverted, it does not have to be transferred through the arithmetic circuit first to pin A that can be complemented directly, which is another improvement as compared to the Basic Computer. The logic unit uses an 8x1 MUX with 3 selection inputs to implement the circuit.

The table below summarizes the logical operations and their Boolean equivalents.

S_2	S_1	S_0	Output	Microoperation
0	**0**	**0**	$Y = A \wedge B$	**AND**
0	**0**	**1**	$Y = A \vee B$	**OR**
0	**1**	**0**	$Y = A \oplus B = (A \wedge B') \vee (A' \wedge B)$	**XOR**
0	**1**	**1**	$Y = (A \wedge B)' = A' \vee B'$	**NAND**
1	**0**	**0**	$Y = (A \vee B)' = A' \wedge B'$	**NOR**
1	**0**	**1**	$Y = (A \oplus B)' = (A \wedge B) \vee (A' \wedge B')$	**XNOR**
1	**1**	**0**	$Y = A'$	**NOT**
1	**1**	**1**	$Y = B'$	**NOT**

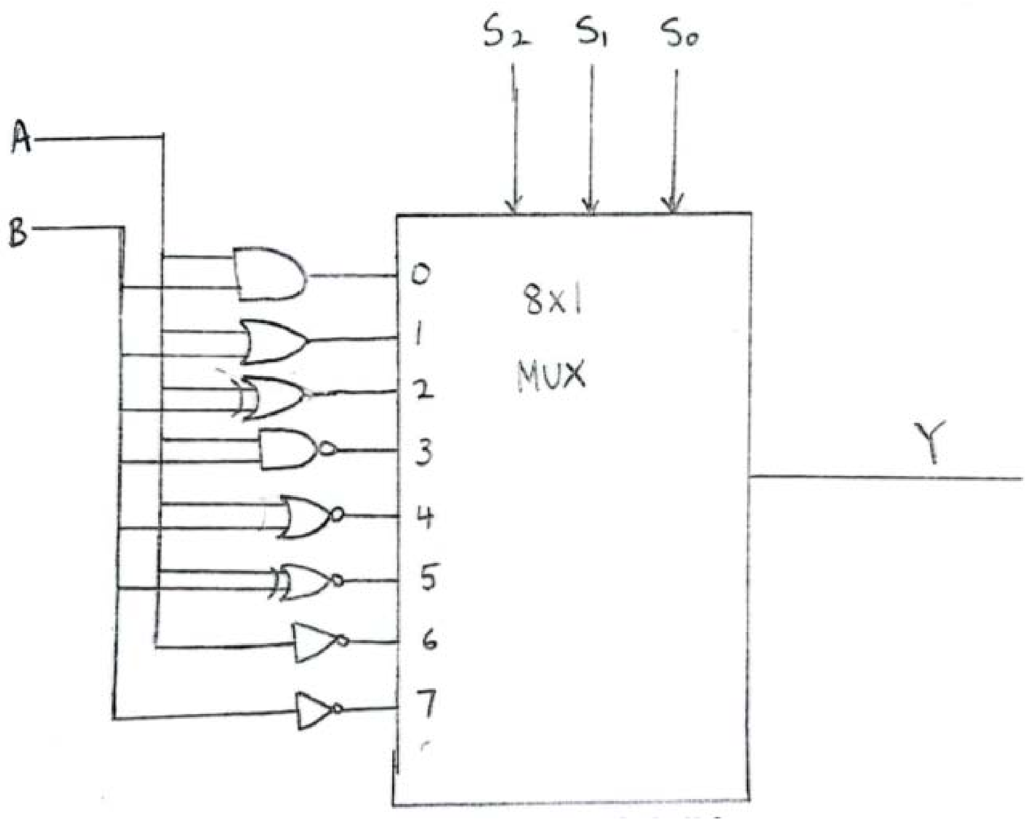


Figure 9-Logic Unit

2.4.3 Shift Microoperations

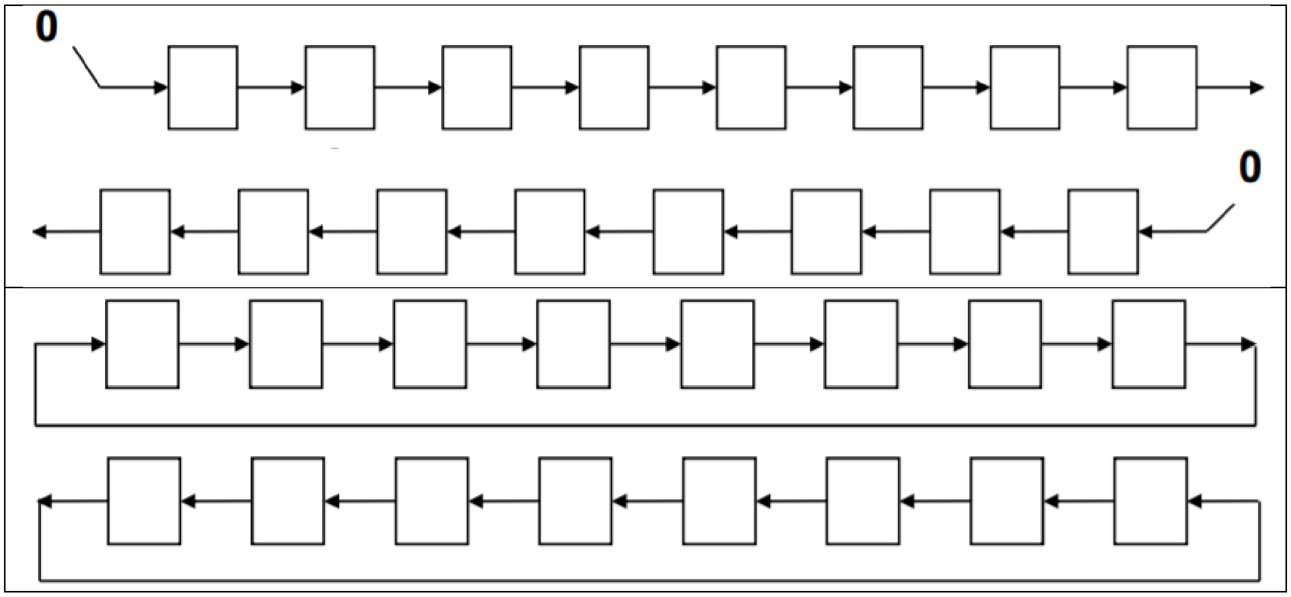
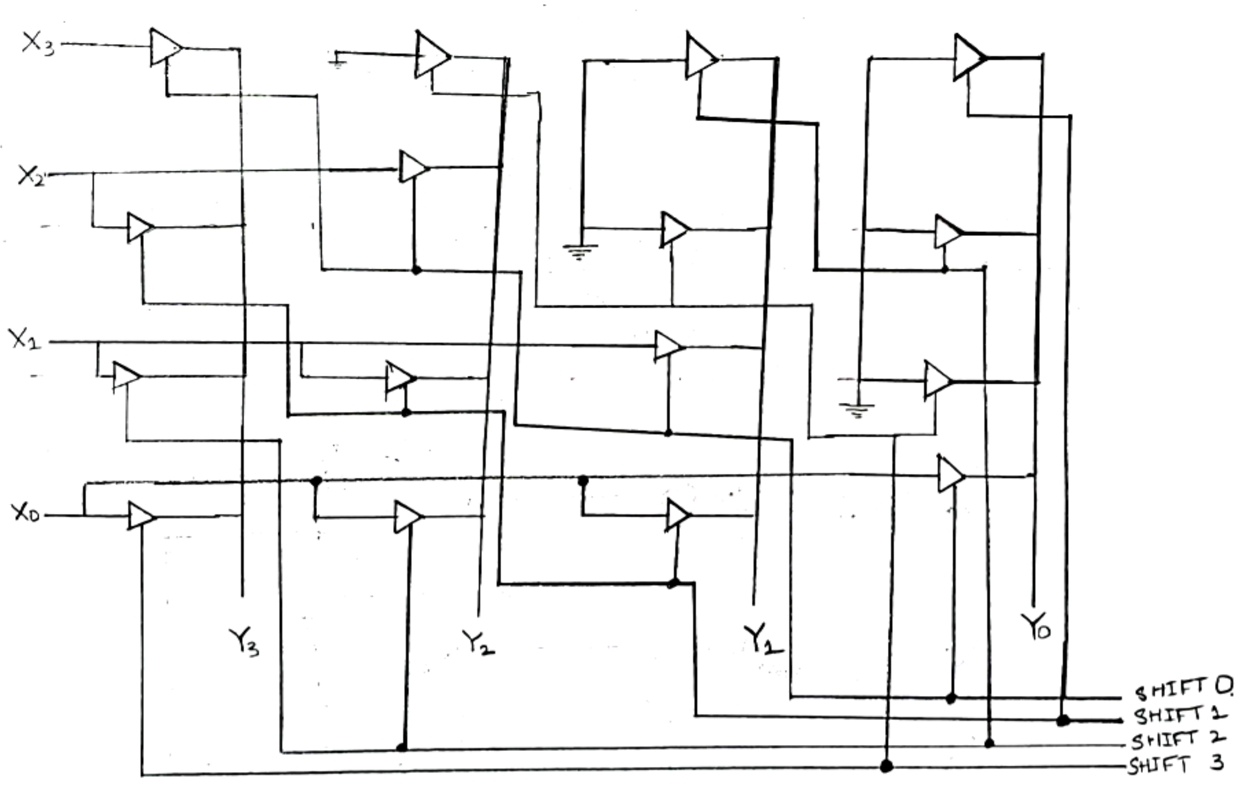
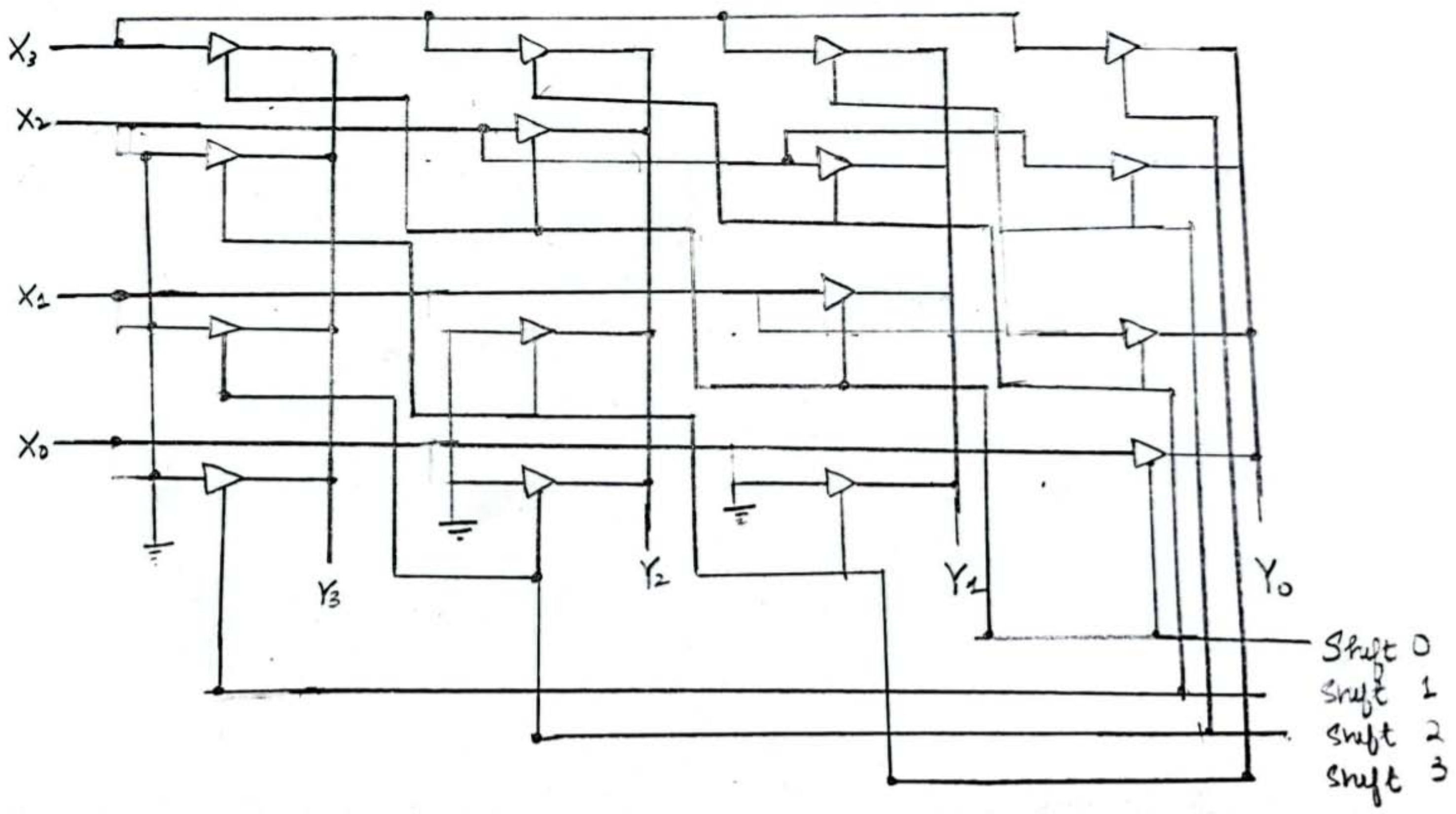
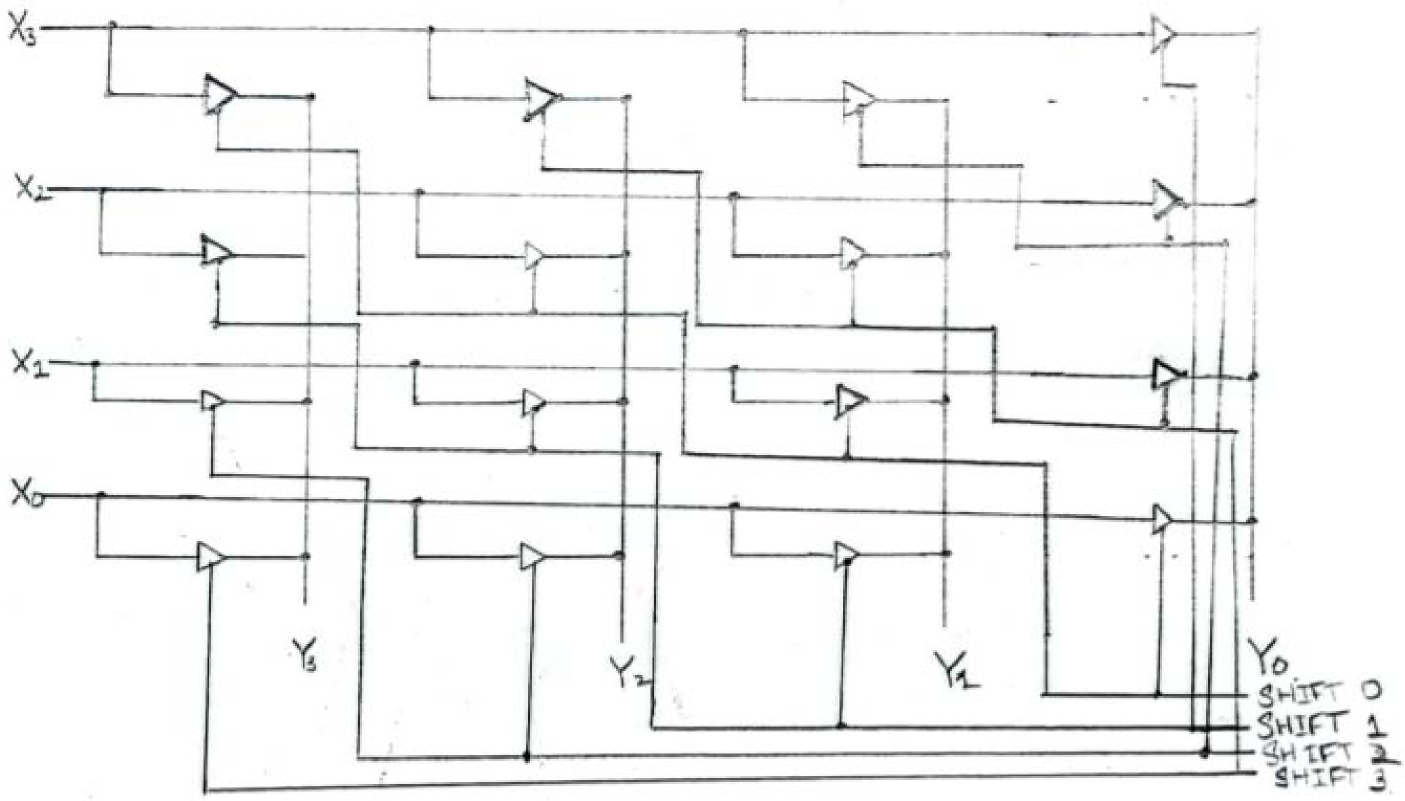


Figure 10-Logical shift right,   Logical shift left, circular shift right, circular shift left (in order from top to bottom)

Fig. 10 summarizes the main shift operations that will be performed by the shift circuits of the ALU. The logical shift operations will append a 0 binary bit at the right-end (left shift). Unlike the Basic Computer, the BACE processor utilizes the concept of barrel-shifter circuits, as shown below. The main advantage of this design selection is that the given circuit can perform up to three shifts in both left and right directions, and the circuit can be easily modified so that both circular and logical shifts can be implemented. The only major disadvantage that must be noted is that the barrel shifter circuit utilizes excessive hardwiring and thus is costly to implement practically. By implementing this logic, multiple shifts can be implemented under just one clock cycle. To select the number of shifts that will be performed on the operand, a 4 by 2 decoder will be connected with two select input lines.




X3
X2
X1
X0
Y3
Y2
Y1
Y0
Shift 0
Shift 1
Shift 2
Shift 3


X3
X2
X1
X0
Y3
Y2
Y1
Y0
SHIFT 0
SHIFT 1
SHIFT 2
SHIFT 3

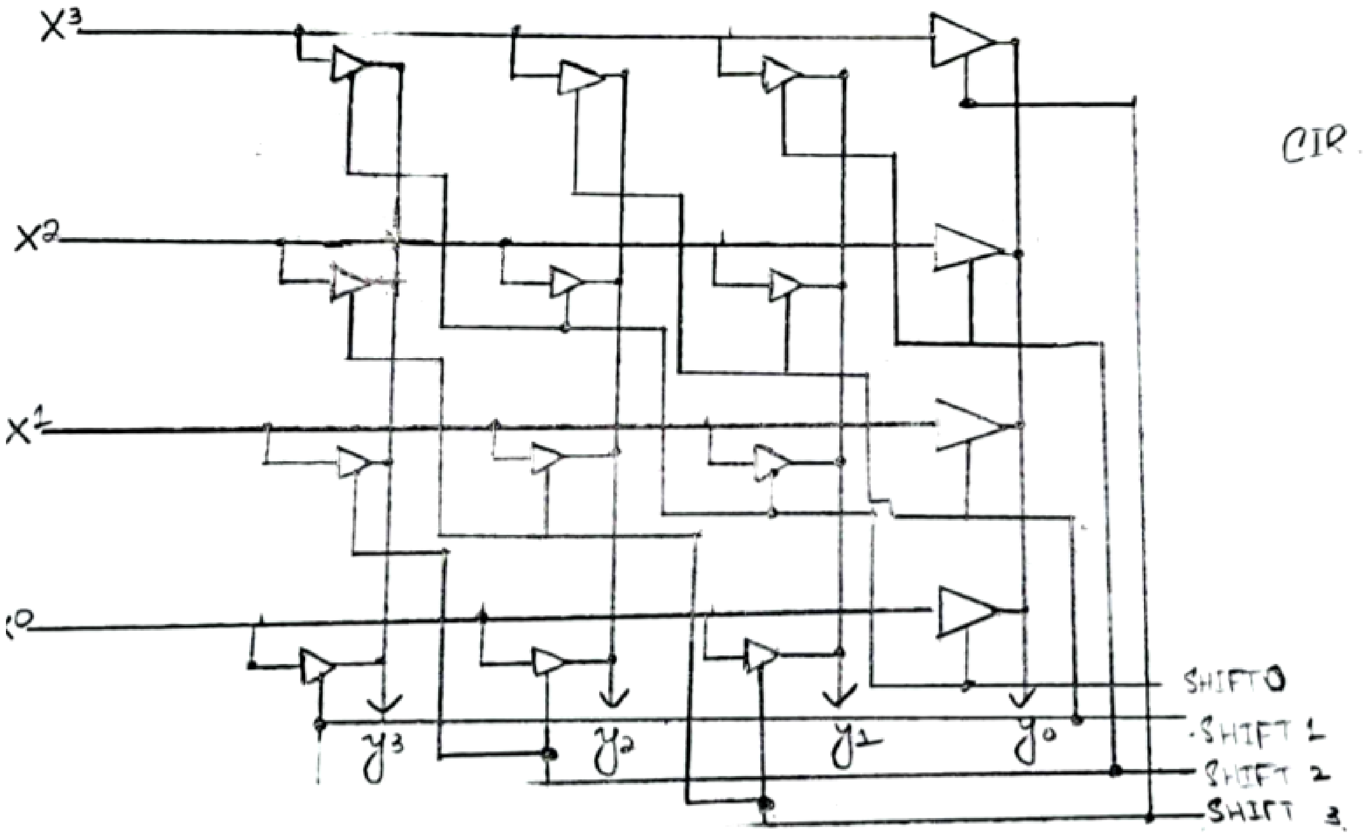


Figure 11-Logical shift left, Logical shift right, circular shift left, circular shift right (in order from top to bottom)

2.4.4 Miscellaneous Circuit Units for New Microoperations

In this section, some combinational circuits have been included in order to increase the functionality of BACE processor, namely the comparator, gray to binary code converter and binary to gray code converter.

The circuit on the left hand side in Fig. 12 presents the gray code to binary converter whereas the one on the right specifies binary to gray code converter. The diagrams display their respective structures on a 4-bit input, but the same generalized pattern will follow from an n-bit register input in the ALU.

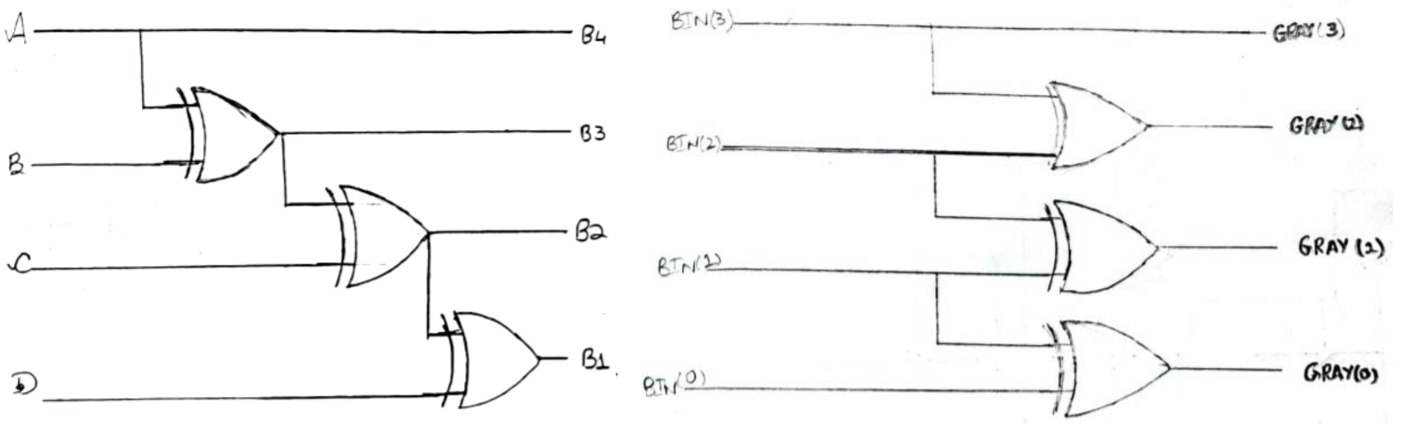


Figure 12-Gray-to-Binary (left) and Binary-to-Gray (right)

Lastly, Fig. 13 showcases the circuit of the comparator circuit. Once again, the input is only for a 4-bit register but the same pattern in logic can be enlarged to fit any n-bit register input application and thus is included as the final piece in the ALU of BACE processor.

Comparator utilizes the use of as many XNOR gates as the number of input bits, which outputs 1 only if the two bits are equal. All the inputs are fed into a single AND gate which outputs 1 if and only if all the corresponding bits in the A and B inputs are the same.

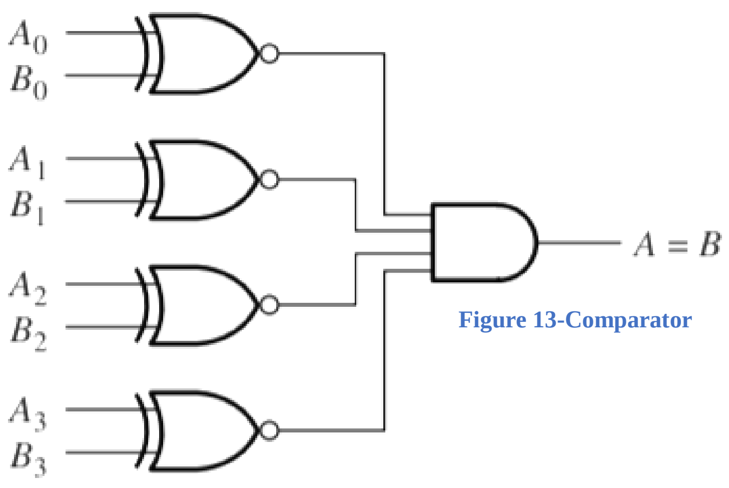


Figure 13-Comparator

2.4.5 Complete ALU Design

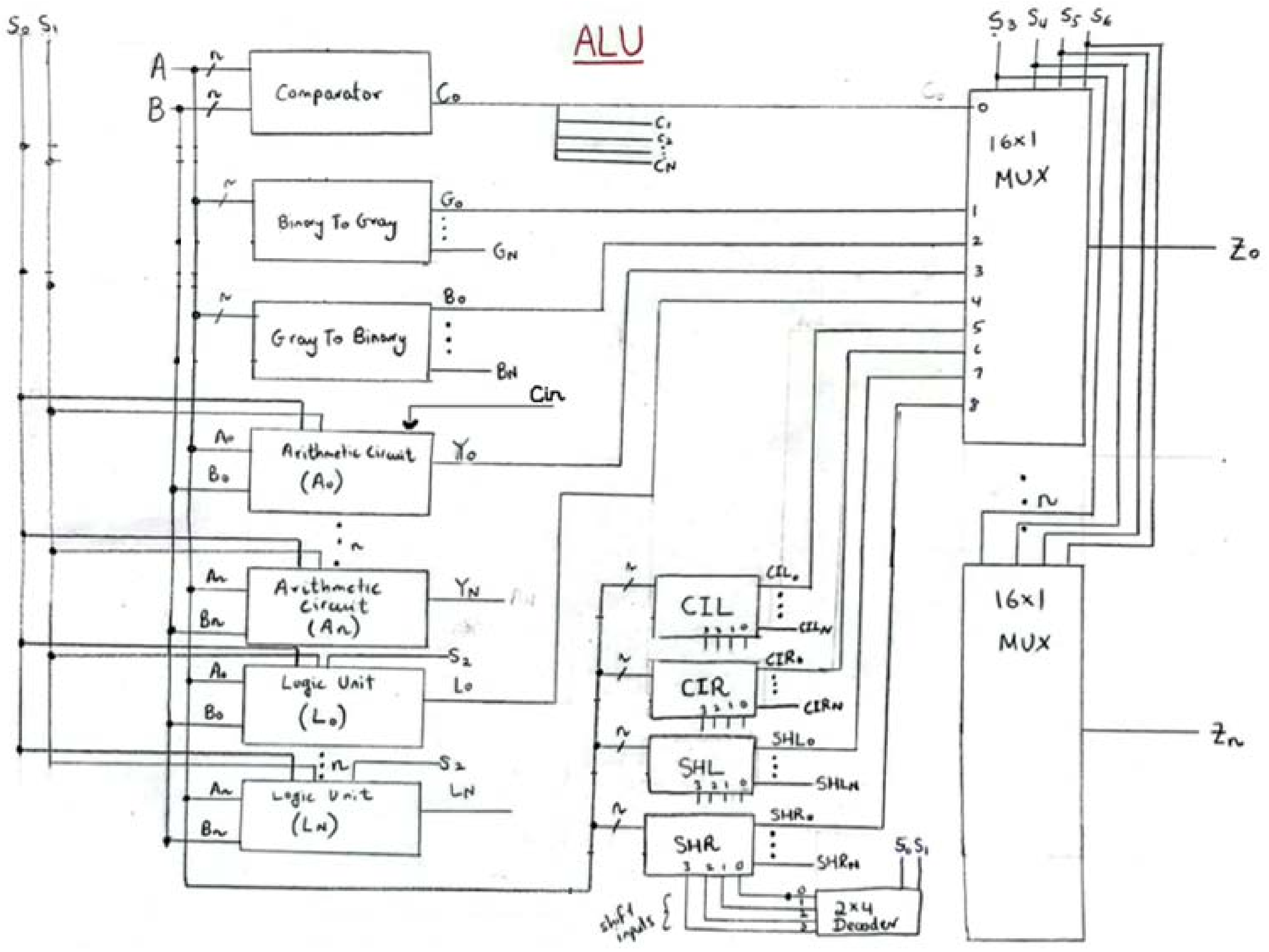


Figure 14-Complete ALU

S_6	S_5	S_4	S_3	S_2	S_1	S_0	CIN	Operation	Function
0	0	0	0	x	x	x	x	A = = B	Equality check using comparator
0	0	0	1	x	x	x	x	BIN → GRAY	Conversion from binary to gray
0	0	1	0	x	x	x	x	GRAY → BIN	Conversion from gray to binary
0	0	1	1	x	0	0	0	A + B	Addition
0	0	1	1	x	0	0	1	A + B + 1	Addition with carry
0	0	1	1	x	0	1	0	A + B’	Subtraction with borrow
0	0	1	1	x	0	1	1	A + B’ + 1	Subtraction
0	0	1	1	x	1	0	0	A	Transfer
0	0	1	1	x	1	0	1	A + 1	Increment
0	0	1	1	x	1	1	0	A – 1	Decrement
0	0	1	1	x	1	1	1	A	Transfer
0	1	0	0	0	0	0	x	$A \wedge B$	AND
0	1	0	0	0	0	1	x	$A \vee B$	OR
0	1	0	0	0	1	0	x	$A \oplus B$	XOR
0	1	0	0	0	1	1	x	$(A \wedge B)'$	NAND
0	1	0	0	1	0	0	x	$(A \vee B)'$	NOR
0	1	0	0	1	0	1	x	$(A \oplus B)'$	XNOR
0	1	0	0	1	1	0	x	A’	COMPLEMENT A
0	1	0	0	1	1	1	x	B’	COMPLEMENT B
0	1	0	1	x	0	0	x	CIL 0	Circular shift left 0 times
0	1	0	1	x	0	1	x	CIL 1	Circular shift left 1 times
0	1	0	1	x	1	0	x	CIL 2	Circular shift left 2 times
0	1	0	1	x	1	1	x	CIL 3	Circular shift left 3 times
0	1	1	0	x	0	0	x	CIR 0	Circular shift right 0 times
0	1	1	0	x	0	1	x	CIR 1	Circular shift right 1 times
0	1	1	0	x	1	0	x	CIR 2	Circular shift right 2 times
0	1	1	0	x	1	1	x	CIR 3	Circular shift right 3 times
0	1	1	1	x	0	0	x	SHL 0	Logical shift left 0 times
0	1	1	1	x	0	1	x	SHL 1	Logical shift left 1 times
0	1	1	1	x	1	0	x	SHL 2	Logical shift left 2 times
0	1	1	1	x	1	1	x	SHL 3	Logical shift left 3 times
1	0	0	0	x	0	0	x	SHR 0	Logical shift right 0 times
1	0	0	0	x	0	1	x	SHR 1	Logical shift right 1 times
1	0	0	0	x	1	0	x	SHR 2	Logical shift right 2 times
1	0	0	0	x	1	1	x	SHR 3	Logical shift right 3 times

The above ALU designed specifically for the BACE processor can perform arithmetic, logic, shift, comparison, and conversion operations on the inputs fed into the circuit. The corresponding truth table of the ALU succinctly captures the overall working of the circuit.

It must be noted that the output of the comparator is an n-bit output of all 1’s or 0’s depending on whether the two n-bit inputs are equal or not respectively. It must be noted that the n^{th} bit output of the individual circuits is inputted into the nth 16 by 1 multiplexer. In addition, the barrel shifter circuits are connected to a 2 by 4 decoder that determines the number of shifts that will be performed on the n-bits of the register A.

Furthermore, the 16 by 1 MUX receives 9 inputs, and therefore the remaining 7 input pins are idle and unused, and thus not included in the truth table either for clarity.

2.5 Register Selection and Main Memory Organization

As already mentioned, the inclusion of a main memory for the storage of data and instructions is a necessity to correctly implement the von Neumann architecture. Main memory forms the backbone of all memory-reference instructions. It contains a unidirectional address bus for accessing specific words as indexed by the control unit whilst also incorporating a bidirectional data bus for reading and writing operations; in addition, 'Read' and 'Write' enable pins connect with the memory to specify the particular operation being performed by the control unit at given time.

The main memory comprises of 24-bit words, where bits 0 to 17 are used to store the operand or address.

Some other noteworthy features of the main memory are listed below:

- Memory contains 2^18 = 262144 unique storage cells of length 24 each.
- The address and data bus are 18- and 24-bits wide respectively.
- The Memory Address Register (i.e. AR) specifies the particular address in memory. A write operation can be represented in RTL as M[AR] $\leftarrow$ B and read operation can be written as B $\leftarrow$ M[AR] where B is any arbitrary register in both cases.

2.5.1 Register Organization

The below table summarizes the processor registers that will be used to delegate certain roles to each register to form a common integrator in the next section which will be run and managed by the control unit to execute the Instruction Set Architecture (ISA).

Name of Register	Size (bits)	Function
Data Register (DR)	24	Holds the operand from memory which can be fed directly into the ALU for arithmetic and logic operations with the accumulator.
(Memory) Address Register (AR)	18	Stores the memory location where some data is to be written or stored usually after performing some memory-reference instruction.
Accumulator (AC)	24	Part of ALU; used to store intermediate arithmetic and logic operation results. The accumulator also connects to an E flip-flop called the Extended AC in case of any overflow conditions or for implementing shift operations in the
Program Counter (PC)	18	Points to the next address in memory storing the next instruction to be executed; increments by 1 sequentially

		at the start of every fetch-decode-execute cycle.
Temporary Register (TC)	24	Stores any data for short-term. Acts as an extra storage unit when needed. Useful for instructions such as exchange (EXC).
Instruction Register (IC)	24	Stores the instruction code of the instruction format.
Input Register (INPR)	16	Holds input character received from the input device; only the 16 least significant bits are connected to the bus; its contents are transferred to accumulator.
Output Register (OUTR)	16	Holds output character received from the accumulator and transfers it to the output device interface, and is not connected to any other register.

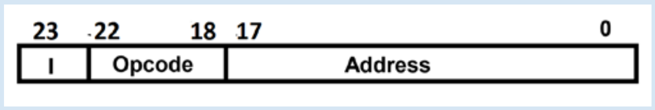
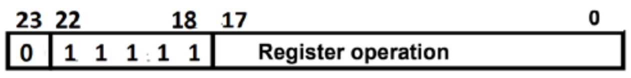
Some key points to note include:

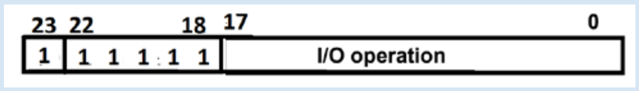
- Each register has load (LD), increment (INC), clear (CLR) and clock (CLK) pins. Clock is fed into the registers from the Sequencer Counter (SC) and the former three pins are managed by the control logic of the control unit.
- The accumulator's ALU takes input from the outputs of DR, INPR, and AC itself.
- The INPR and OUTR both have 16-bits so that the BACE processor is compatible with the Unicode Standard and can store 16-bit defined characters whereas the Basic Computer was only compatible with ASCII Character Set.

2.6 Instruction Format

The format of the memory word can describe three different categories of instructions, namely:

- Memory-Reference Instructions-operate by manipulating the data and instructions in the main memory for various applications.
- Register-Reference Instructions-operate directly on the registers defined within the processor without the need of specifying an operand in the main memory.
- Input-Output Instructions-deal with external input and output operations which may be required during the execution of any program.

Type of Instruction	Description	Format
Memory-Reference Instruction	Uses one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address. *(OP-code = 00000-11110)*	
Register-Reference Instruction	The addressing mode bit I is always 0. *(Op-code = 11111, I = 0)*	 23 22 18 17 0 0 1 1 1 1 1 Register operation

Input/Output- Instruction	The addressing mode bit is always 1. *(Op-code = 11111, I = 1)*	

The BACE processor accomplishes indirect and direct addressing by dedicating the 23rd dedicated to specifying the addressing modes (i.e. indirect and direct). In the case of direct addressing, the operand stored in memory will be accessed and transferred directly for the execution phase; in contrast, indirect addressing mode entails the use of an extra referencing step from memory to retrieve the effective address (i.e. EA) of the operand.

Furthermore, 5 bits are dedicated to specify the operation code (here onward referred to as opcode).

2.7 The Common Integrator

This section briefly details the overall structural organization of the processor registers defined in the previous sections. As shown, the registers connect to a 24-bit common bus and the loading and reading to and from the registers is performed by the three select inputs S_2, S_1, and S_0.

Some notable features of the common integrator include how the AC does not connect directly with the bus for any write operations and instead is fed inputs by the ALU which received inputs from the outputs of AC, DR, and INPR. The OUTR does not connect with the common bus for any reading purposes as contents are shown directly on the external output interface; by the same token, INPR does not connect to the bus for either reading or writing since the input is fed into the register externally and its value is directly passed to the ALU. The implementation of the common bus saves circuitry that would otherwise have been used to connect individual registers in direct connections with each other.

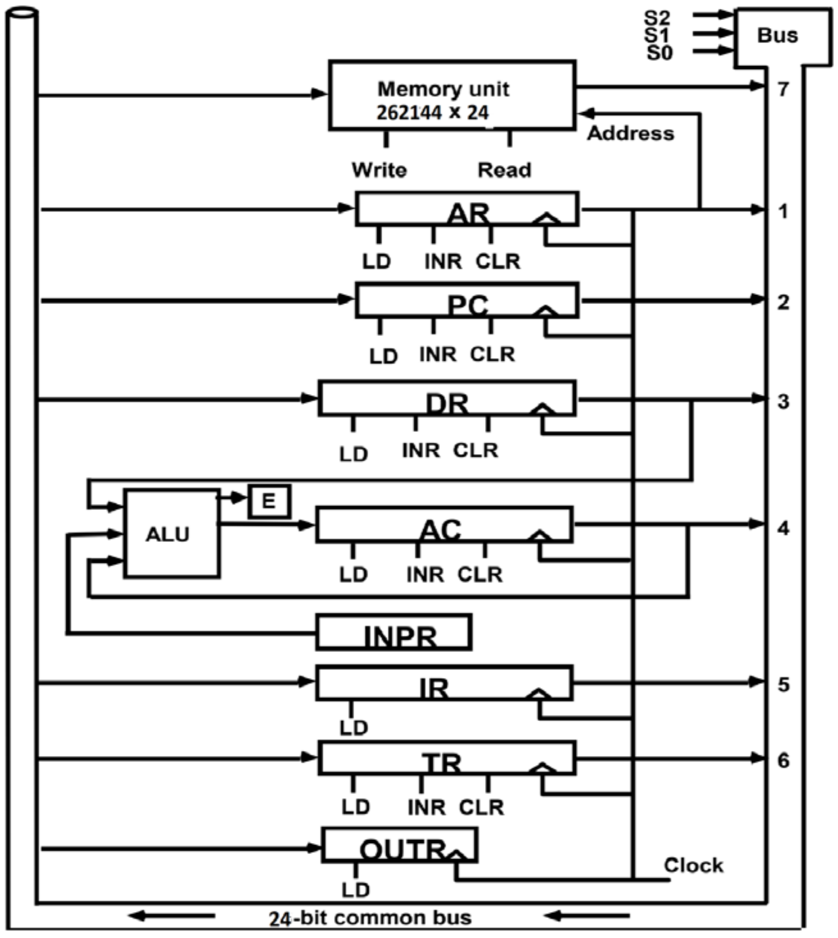


Figure 15-Common Integrator

3. Definition & Implementation of the Instruction Set

The instructions implemented by the BACE processor are outlined in the table below. For convenience, the opcode instructions have been digitally represented using octal number system as the base 8 system is divisible by the size of BACE processor's instruction size of 24 bits.

Symbol	**Octal Code**		**Description**
	I = 0	**I = 1**	
Memory-Reference Instructions			
SUM	00xxxxxx	40xxxxxx	Add content of memory to accumulator.
DIF	01xxxxxx	41xxxxxx	Subtract content of memory from accumulator.
GOTO	02xxxxxx	42xxxxxx	Jumps/branches execution to a specified point.
CALL	03xxxxxx	43xxxxxx	Calling subroutines in the program.
WMA	04xxxxxx	44xxxxxx	Load content into accumulator from memory.
WAM	05xxxxxx	45xxxxxx	Store content into memory from accumulator.
AND	06xxxxxx	46xxxxxx	AND memory with accumulator.
OR	07xxxxxx	47xxxxxx	OR memory with accumulator.
XOR	10xxxxxx	50xxxxxx	XOR memory with accumulator.
NAND	11xxxxxx	51xxxxxx	NAND memory with accumulator.
NOR	12xxxxxx	52xxxxxx	NOR memory with accumulator.
XNOR	13xxxxxx	53xxxxxx	XNOR memory with accumulator.
NOT	14xxxxxx	54xxxxxx	NOT memory with accumulator.
INC	15xxxxxx	55xxxxxx	Increment contents of accumulator.
DEC	16xxxxxx	56xxxxxx	Decrement contents of accumulator.
GTB	17xxxxxx	57xxxxxx	Convert gray to binary code.
BTG	20xxxxxx	60xxxxxx	Convert binary to gray code.
EQL	21xxxxxx	61xxxxxx	Check equality of content between AC and memory word.
EXC	22xxxxxx	62xxxxxx	Exchange the contents of AC and memory.
TCA	23xxxxxx	63xxxxxx	Twos complement of the AC.
SHL	24xxxxxx	64xxxxxx	Logical shift left.
SHR	25xxxxxx	65xxxxxx	Logical shift right.
CIL	26xxxxxx	66xxxxxx	Circular shift left.
CIR	27xxxxxx	67xxxxxx	Circular shift right.
ISZ	30xxxxxx	70xxxxxx	Increment and skip if zero.
Register-Reference Instructions			
CLA	37000001		Clear accumulator.
CLE	37000002		Clear extended accumulator.
CMA	37000003		Complement accumulator.
CME	37000004		Complement extended AC flip-flop.
CIR	37000005		Circular shift right AC and E.
CIL	37000006		Circular shift left AC and E.
INC	37000007		Increment AC.
SPA	37000010		Skip next instruction if AC is positive.

SNA	37000011	Skip next instruction if AC is negative.
SZA	37000012	Skip next instruction if AC is zero.
SZE	37000013	Skip next instruction if E is zero.
END	37000014	Halt computer execution.
Input/Output Reference Instructions		
INP	77000001	Input data to register.
OUT	77000002	Output data to register.
SKI	77000003	Skip on input flag.
SKO	77000004	Skip on output flag.
ION	77000005	Interrupt on.
IOF	77000006	Interrupt off.

The instruction set as outlined above is functionally complete as it includes all the categories of instructions used by assembly language programmers to execute their programs, namely: the arithmetic, shift and logic instructions; data transfer instructions; program sequencing control and status instructions; input and output instructions. Since the usage of subroutines is common across all applications of computer processing, it was imperative to provide this functionality hardwired within the machine instruction set of the BACE processor to facilitate subroutine entry to and fro the main program.

3.1 The Control Unit

Control unit fetches code for instructions from microprograms and directs other units and models by providing control and timing signals. The function of the Control unit is as follows:

- Controls sequential instruction execution
- Interprets instructions
- Regulates and controls processor timing
- Sends and receives control signals from other computer devices
- Handles multiple tasks, such as fetching, decoding, execution handling and storing results.

The control unit comprises of two decoders (5-32) and (4-16), one four-bit_sequence counter and other logical circuits. The 5-32 decoder takes five bits from the instruction register and decodes it to acquire the desired set of microoperations. The five bits are used to produce the 32 instructions where each instruction has its own micro-operation. The control unit also provides the CPU timing signals to sequentially execute the instructions to be performed. The binary value of the 4-bit sequence counter is fed into the decoder, and together the two units function similar to a ring counter, in which a bit value of 1 is cycled through the timing states one at a time.

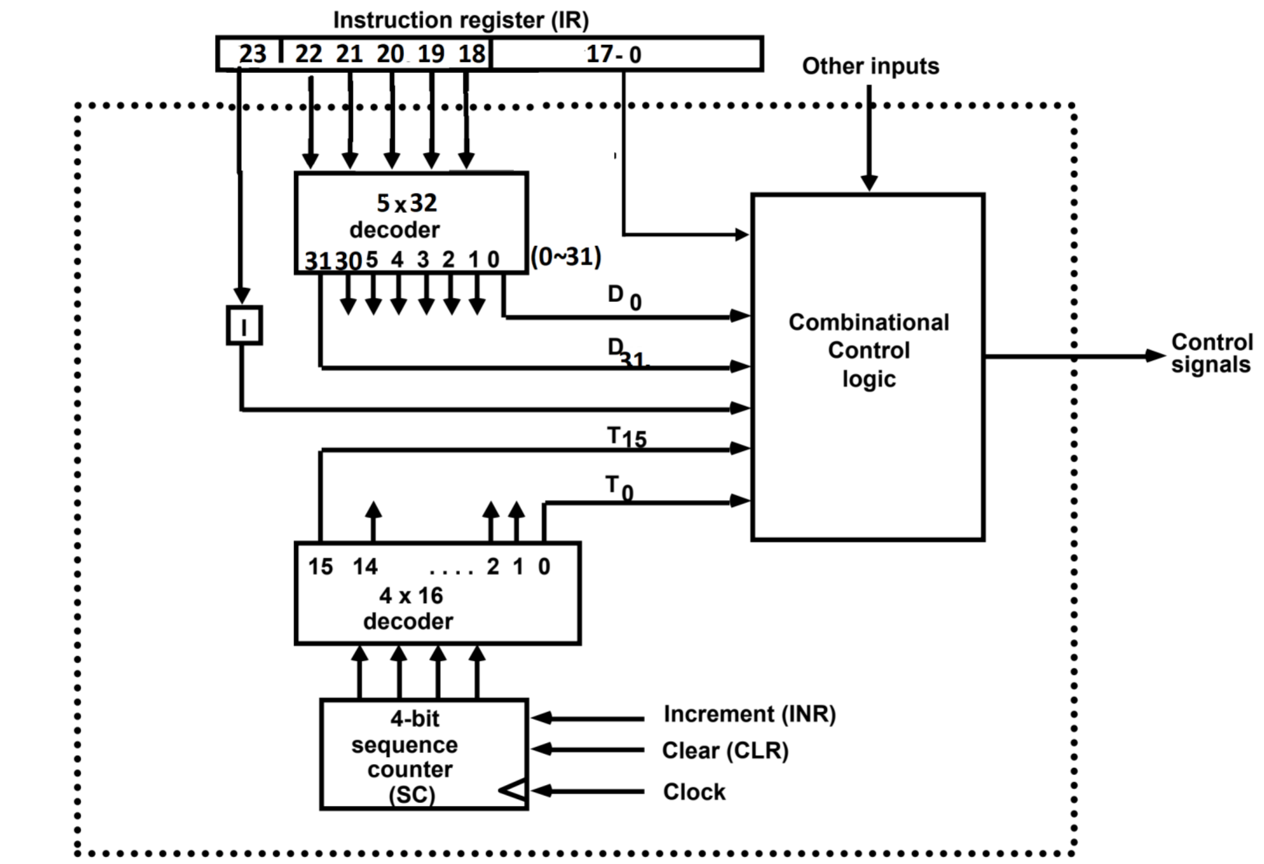
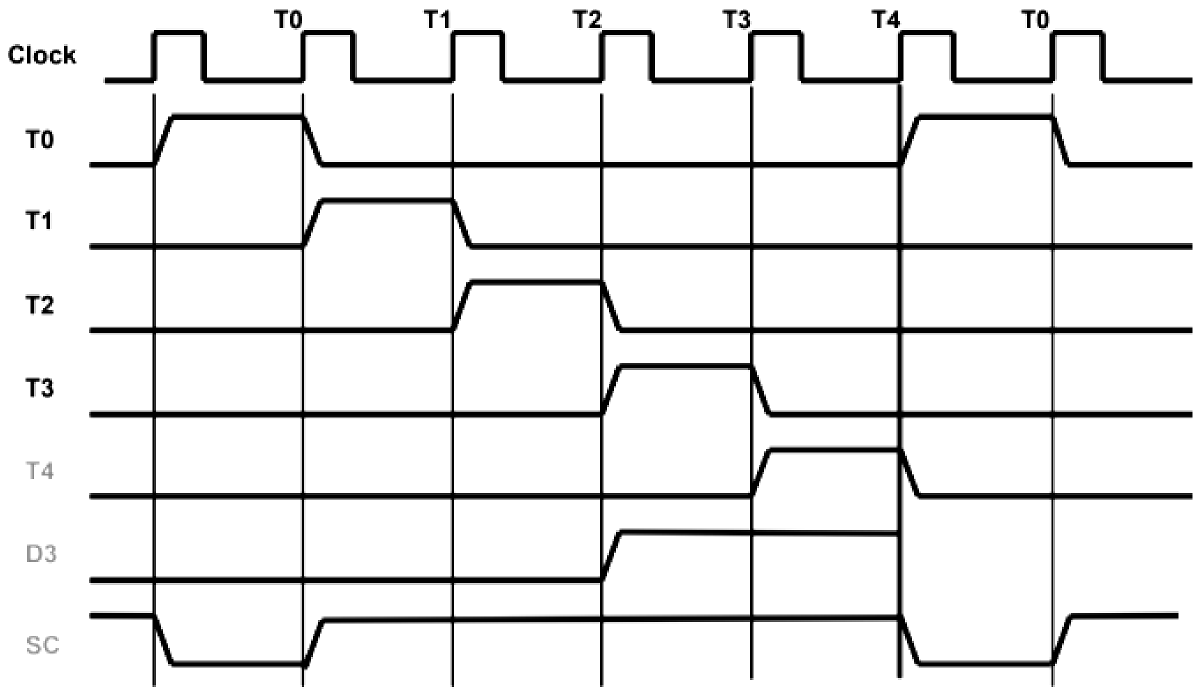
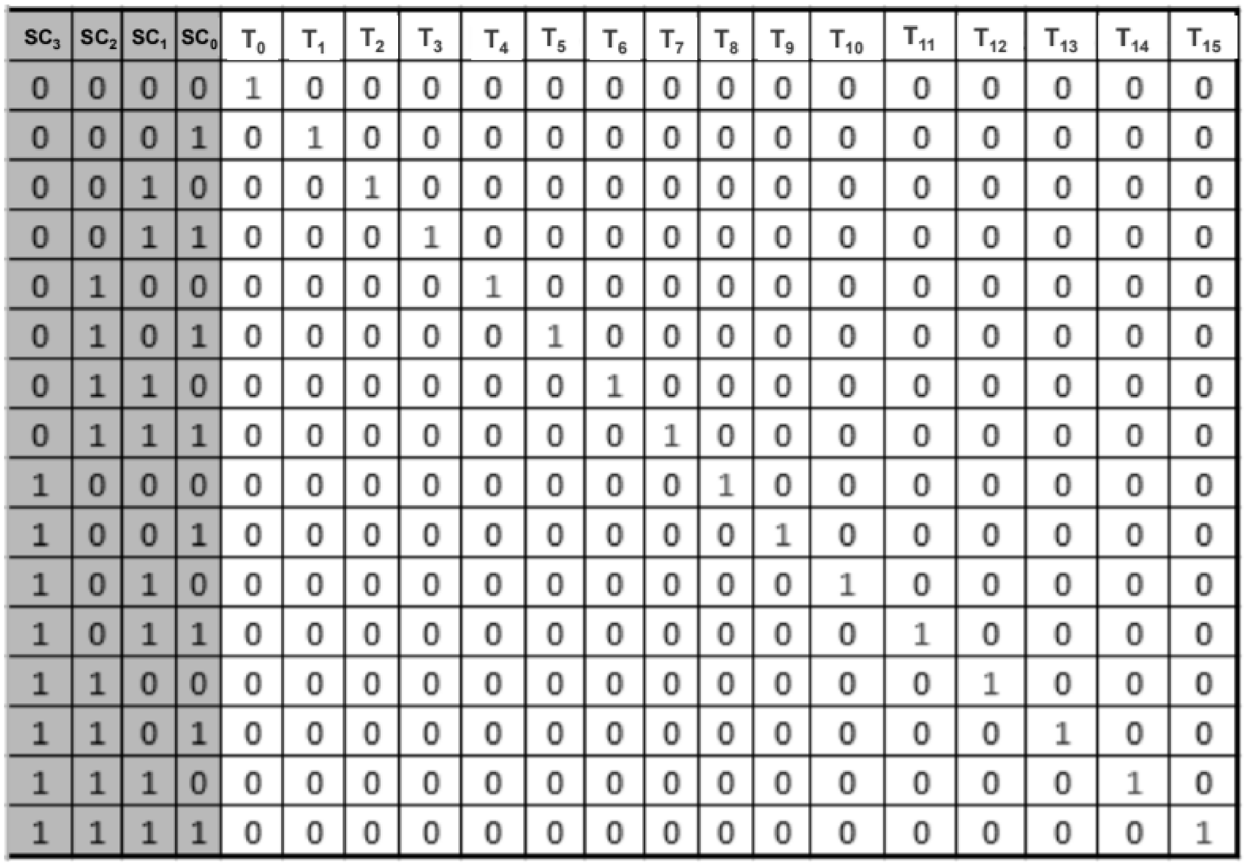


Figure 16-Control Unit



-

Figure 17-Sequencer and Decoder Combination Waveform



SC_3	SC_2	SC_1	SC_0	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 18-Sequencer and Decoder Combination Truth Table

Four-bit sequence counter generates the output which will be decoded in the decoder to generate the control signals from T0 to T15. After every instruction the sequence counter is reset to zero to again perform the fetch and decode cycle. Initially the sequence counter is zero so the decoder will generate the timing signal T0. The timing signal T0 will select the PC from the 24-bit common bus and AR from the logic circuit. This control signal is received by the integrated circuit and transfers the address from PC to AR during the clock transition associated with timing signal T0. At timing signal T1, AR will be selected and the instruction read from memory will be placed in the instruction register.

3.3 Fetch-Decode Cycle

The below flowchart summarizes the fetch and decode cycles that occurs at the start of every timing state. Following this cycle, the specific execute sequences microoperations for each instruction will be executed, given in the complete computer description.

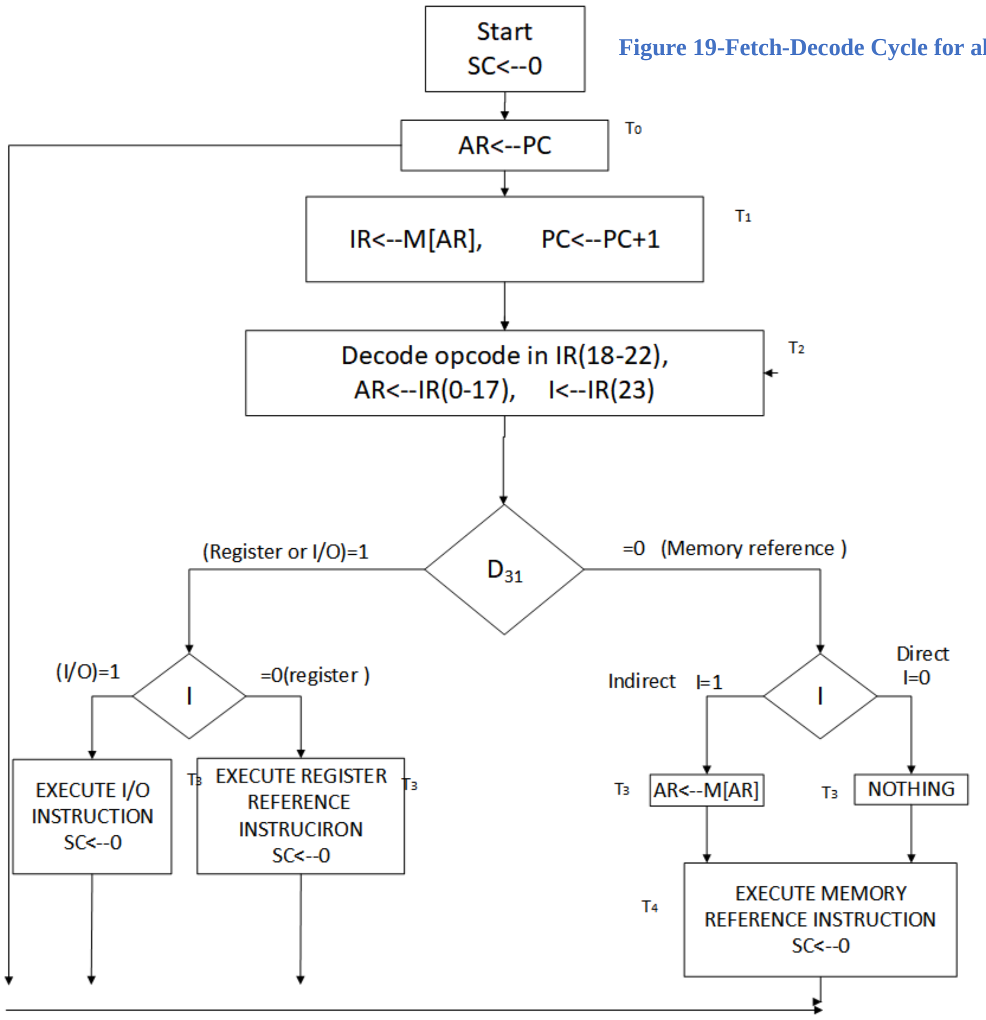

Start
SC<--0
AR<--PC
T0
IR<--M[AR], PC<--PC+1
T1
Decode opcode in IR(18-22),
AR<--IR(0-17), I<--IR(23)
T2
D31
(Register or I/O)=1
=0 (Memory reference)
(I/O)=1
I
=0(register)
Indirect I=1
I
Direct I=0
EXECUTE I/O INSTRUCTION SC<--0
T3
EXECUTE REGISTER REFERENCE INSTRUCIRON SC<--0
T3
T3
AR<--M[AR]
T3
NOTHING
T4
EXECUTE MEMORY REFERENCE INSTRUCTION SC<--0

Figure 19-Fetch-Decode Cycle for all instructions

3.2 Input/Output and Interrupt Handling

Computer contains output flag and input flag to inform the processor what the user wants to input data or output data. The processor continuously checks that any of the flag are active (flag=1) or not (flag=0).it is one of the reasons that the cursor blinks and when user strikes a key, or input data the input flag ((FGI=1) is active and processor store data in input register and then to accumulator. similarly, when the user wants to output data, the output flag (FGO=1) is active and the data from accumulator is copied to output register. In basic computer it has an interrupt enable function (IEN). The interrupt helps in a way that when interrupt is active, it stops all background processing and performs the task. As in displayed in figure below which show how a computer works combining IEN, FGO, FGI. First the processor checks that if the interrupt is active or not. If not, it fetches and decodes instructions. If (IEN=1) it checks that whether the output flag is active or input flag and enables Interrupt r flip-flop. All operations are paused, and the return address is stored. In location 0 for example and the instruction stored in the location ‘1’ is performed first, which may contain the input/output subroutine. At the end of subroutine is the indirect BUN which resumes the main program executions.

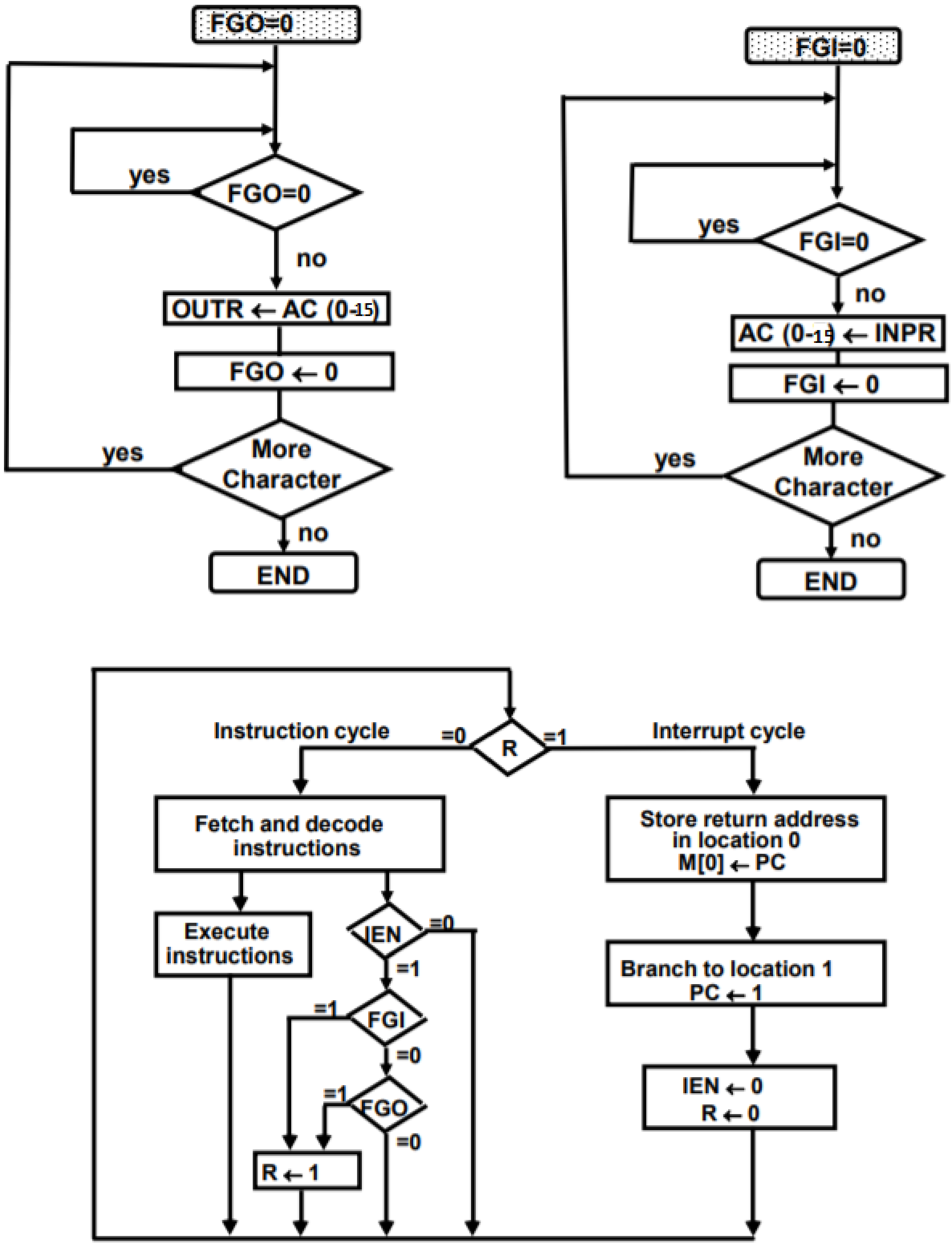


Figure 20-Input/Output Handling Flowcharts

4. Complete Computer Description

Fetch-Decode Cycle	
Fetch	
R’T_0:	AR ← PC
R’T_1:	IR ← M[AR], PC ← PC + 1
Decode	
R’T_2:	D_0, …, D_{31} ← Decode IR(18~22), AR ← IR(0~17), I ← IR(23)
Indirect Addressing	
D_7’IT_3:	AR ← M[AR]
Interrupt	
T_0’T_1’T_2’(IEN)(FGI+FGO)	R ← 1
RT_0	AR ← 0, TR ← PC
RT_1	M[AR] ← TR, PC ← 0
RT_2	PC ← PC + 1, IEN ← 0, SC ← 0

Execution Cycles		
Memory-Reference		
SUM	D_0T_4	DR ← M[AR]
	D_0T_5	AC ← AC + DR, E ← C_{out}, SC ←0
DIF	D_1T_4	AC ← TR, M[AR] ← DR
	D_1T_5	DR ← AC
	D_1T_6	AC ← AC’
	D_1T_7	AC ← AC + 1
	D_1T_8	AC ← AC + TR, E ← C_{out}, SC ← 0
GOTO	D_2T_4	AR ← PC, SC ← 0
CALL	D_3T_4	M[AR] ← PC, AR ← AR +1
	D_3T_5	PC ← AR, SC ← 0
WMA	D_4T_4	M[AR] ← PC, SC ← 0
WAM	D_5T_4	DR ← M[AR]
	D_5T_5	AC ← DR, SC ← 0
AND	D_6T_4	DR ← M[AR]
	D_6T_5	AC ← AC $\wedge$ DR, SC ←0
OR	D_7T_4	DR ← M[AR]
	D_7T_5	AC ← AC $\vee$ DR, SC ←0
XOR	D_8T_4	DR ← M[AR]
	D_8T_5	AC ← AC $\oplus$ DR, SC ←0
NAND	D_9T_4	DR ← M[AR]
	D_9T_5	AC ← AC (nand) DR, SC ←0
NOR	$D_{10}T_4$	DR ← M[AR]
	$D_{10}T_5$	AC ← AC (nor) DR, SC ←0
XNOR	$D_{11}T_4$	DR ← M[AR]

	$D_{11}T_5$	AC 🡐 AC (xnor) DR, SC 🡐0
NOT	$D_{12}T_4$	DR 🡐 M[AR]
	$D_{12}T_5$	AC 🡐 DR
	$D_{12}T_6$	AC 🡐 AC’, SC 🡐 0
INC	$D_{13}T_4$	DR 🡐 M[AR]
	$D_{13}T_5$	AC 🡐 DR
	$D_{13}T_6$	AC 🡐 AC + 1, SC 🡐 0
DEC	$D_{14}T_4$	DR 🡐 M[AR]
	$D_{14}T_5$	AC 🡐 DR
	$D_{14}T_6$	AC 🡐 AC - 1, SC 🡐 0
GTB	$D_{15}T_4$	DR 🡐 M[AR]
	$D_{15}T_5$	AC 🡐 DR
	$D_{15}T_6$	AC 🡐 gtb AC, SC 🡐 0
BTG	$D_{16}T_4$	DR 🡐 M[AR]
	$D_{16}T_5$	AC 🡐 DR
	$D_{16}T_6$	AC 🡐 btg AC, SC 🡐 0
EQL	$D_{17}T_4$	DR 🡐 M[AR]
	$D_{17}T_5$	AC 🡐 AC = = DR, SC 🡐 0
EXC	$D_{18}T_4$	DR 🡐 M[AR]
	$D_{18}T_5$	M[AR] 🡐 AC
	$D_{18}T_6$	AC 🡐 DR, SC 🡐 0
TCA	$D_{19}T_4$	DR 🡐 M[AR]
	$D_{19}T_5$	AC 🡐 DR
	$D_{19}T_6$	AC 🡐 AC’
	$D_{19}T_7$	AC 🡐 AC + 1, SC 🡐 0
SHL	$D_{20}T_4$	DR 🡐 M[AR]
	$D_{20}T_5$	AC 🡐 DR
	$D_{20}T_6$	AC 🡐 shl AC, SC 🡐 0
SHR	$D_{21}T_4$	DR 🡐 M[AR]
	$D_{21}T_5$	AC 🡐 DR
	$D_{21}T_6$	AC 🡐 shr AC, SC 🡐 0
CIL	$D_{22}T_4$	DR 🡐 M[AR]
	$D_{22}T_5$	AC 🡐 DR
	$D_{22}T_6$	AC 🡐 cil AC, SC 🡐 0
CIR	$D_{23}T_4$	DR 🡐 M[AR]
	$D_{23}T_5$	AC 🡐 DR
	$D_{23}T_6$	AC 🡐 cir AC , SC 🡐 0
ISZ	$D_{24}T_4$	DR 🡐 M[AR]
	$D_{24}T_5$	DR 🡐 DR + 1
	$D_{24}T_6$	M[AR] 🡐 DR, if (DR=0) then (PC 🡐 PC + 1), SC 🡐 0

Execution Cycles		
Register-Reference ($r = D_{31}I'T_3$) Bi where i is the decimal equivalent of the octal code		
CLA	rB_1	$AC \leftarrow 0$
CLE	rB_2	$E \leftarrow 0$
CMA	rB_3	$AC \leftarrow AC'$
CME	rB_4	$E \leftarrow E'$
CIR	rB_5	AC←shr AC, AC(23)←E, E←AC(0)
CIL	rB_6	AC←shl AC, AC(0)←E, E←AC(23)
INC	rB_7	$AC \leftarrow AC + 1$
SPA	rB_8	If (AC(23) =0) then (PC ← PC + 1)
SNA	rB_9	If(AC(23) =1) then (PC ← PC + 1)
SZA	rB_{10}	If(AC = 0) then (PC ← PC + 1)
SZE	rB_{11}	If(E=0) then (PC ← PC + 1)
END	rB_{12}	$S \leftarrow 0$

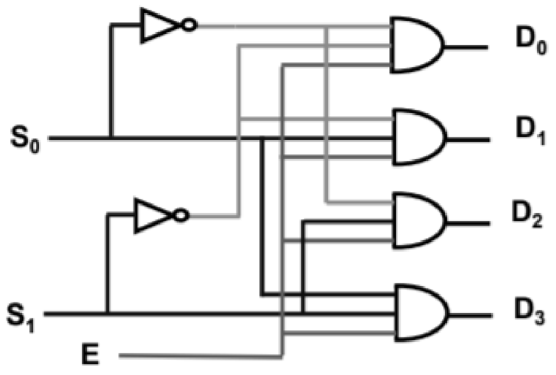
Execution Cycles		
Input/Output-Reference ($p = D_{31}IT_3$) Bi where i is the decimal equivalent of the octal code		
INP	pB_1	AC(0-15) ← INPR, FGI ← 0
OUT	pB_2	OUTR ← AC(0-15), FGO ← 0
SKI	pB_3	If(FGI=1) then (PC ← PC + 1)
SKO	pB_4	If(FGO=1) then (PC ← PC + 1)
ION	pB_5	IEN ← 1
IOF	pB_6	IEN ← 0

5. C++ Simulator

In order to facilitate a visual implementation of the theoretically constructed BACE processor's organization, an original simulator was programmed by members of our team to emulate the basic computer in order to give an idea of how the BACE processor would be implemented. The simulator was programmed in C++ and can be downloaded with this project. A separate handbook detailing the working of the simulator can also be found which contains all the required information to operate the simulator.

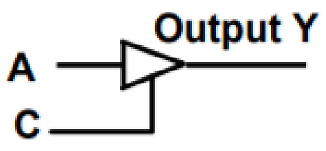
Appendix

A. Decoder



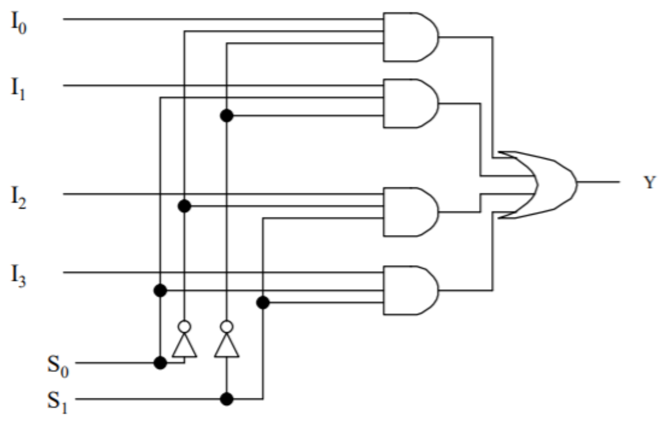
E	S_1	S_0	D_0	D_1	D_2	D_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

B. Three State Gate



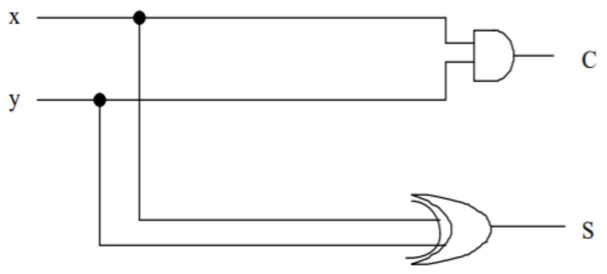
C	A	Y
1	0	0
1	1	1
0	0	Disabled (High Impedance)

C. Multiplexer



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

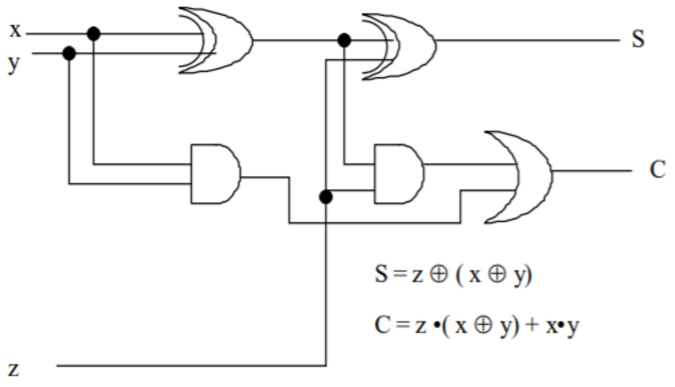
D. Half Adder



S = x ⊕ y
C = xy

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

E. Full Adder



S = z ⊕ (x ⊕ y)

C = z •(x ⊕ y) + x•y

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Bibliography

[1] Thomas L.Floyd, Digital Fundamentals, 11th Edition, Pearson, July 2014,
ISBN: 9780132737968.

[2] Microsoft Visio, 2013.

[3] Paint, 2013.

[4] Morris Mano, Computer System Architecture, 3rd Edition,
ISBN: 9788120308558.

[5] Albert P. Malvino & Jerald A. Brown, Digital Computer Electronics, 3rd Edition, McGraw-Hill, 1993,
ISBN: 9780028005942.

[6] Mohamed Rafiquzzaman, Fundamentals of Digital Logic and Microcomputer Design, 5th Edtition, Wiley-Interscience, 2005, ISBN: 9780471727842.

[7] David Harris & Sarah L. Harris, Digital Design and Computer Architecture, 2021,
ISBN: 9780128200643

[8] William Stallings, Computer Organization and Architecture: Designing for performance, 11th Edition, ISBN: 9780134997193.

[9] Douglas Comer, Essentials of Computer Architecture, 2004,
ISBN: 9780367573959.

[10] John P. Shen & Mikko H. Lipasti, Modern Processor Design, 2003,
ISBN: 9780070570641.