



MTE PROJECT REPORT

COMPUTER ORGANISATION AND ARCHITECTURE

Team Members:-

Ayush Kumar (DTU/2K19/SE/026)

Ayush sharma (DTU/2K19/SE/027)

Submitted to

Dr. Pawan Singh Mehra

TOPIC: 4-BIT ARITHMETIC LOGIC UNIT

CANDIDATE'S DECLARATION

We, Ayush Sharma and Ayush Kumar, Roll Nos. 2K19/SE/027 & 2K19/SE/026 students of B.Tech Software Engineering, hereby declare that the project title “4- Bit Arithmetic Logic Unit” which is submitted by us to the Department of Software Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Bachelor of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi
Date: 17 April 2021

Ayush Sharma and Ayush Kumar

SOFTWARE ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

CERTIFICATE

I hereby certify that the Project Dissertation titled “4-Bit Arithmetic Logic Unit” which is submitted by Ayush Sharma and Ayush Kumar, Roll Nos. 2K19/SE/0271 & 2K19/SE/026, Software Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Bachelor of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi
Date: 17 April 2021

Dr. Pawan Singh Mehra

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher Prof. Dr. Pawan Mehra who gave us the golden opportunity to do this wonderful project on the topic **4-BIT ARITHMETIC LOGIC UNIT** which also helped us in doing a lot of research and we came to know about so many new things in digital electronics. We would also like to thank our university, Delhi Technological University for giving us this opportunity to explore and research in the field of computer science.

Thanking you,

AYUSH KUMAR/ AYUSH SHARMA

ABSTRACT

With the Arithmetic Logic Unit, we can mathematically and logically process two 4-bit numbers using multiplexers.

- The front-side buses (left panel) are the inputs A and B.
 - The back-side buses (right panel) are the data output and the function selection
 - Data passing through the ALU circuit does so on a system of buses. These buses consists of groups of wires (4 parallel bits in simple systems) each carrying a single byte of binary data.
- We'll try to put our best foot forward to make it work on a simulator.

CONTENTS

Candidate's Declaration	2
Certificate	3
Acknowledgement	4
Abstract	5
Contents	6
List of Figures	7
List of Tables	8
List of Symbols	9
CHAPTER 1 INTRODUCTION	
1.1 Arithmetic Circuit	10
1.1.1 Input Carry	
1.1.2 Addition	
1.1.3 Subtraction	
1.1.4 Increment	
1.1.5 Decrement	
1.2 Logic Microoperations	12
1.2.1 Special Symbol	
1.2.2 List of logic microoperation	
1.2.3 Hardware implementation	
1.3 Some Application	15
1.3.1 Selective set	
1.3.2 Selective complement	
1.3.3 Selective Clear	
1.4 Shift Microoperation	17
1.4.1 Circular Shift	
1.4.2 Arithmetic Shift	
1.4.3 Hardware Implementation	
1.5 Shifter	19
1.6 Arithmetic Logic Shift Unit	19
CHAPTER 2 Result And Conclusion	22
References	23

List of Figures

Figure 1.1 – 4-Bit Arithmetic Circuit

Figure 1.2 – One phase of logic circuit

Figure 1.3 – Arithmetic shift right

Figure 1.4 – Shifter Circuit

Figure 1.5 – One stage of arithmetic logic shift unit

Figure 2.1 –Multisim Model

List of Tables

Table 1.1: Arithmetic Microoperation

Table 1.2: Arithmetic Circuit function table

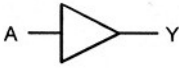
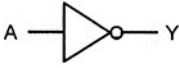

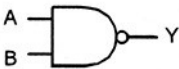
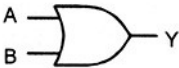
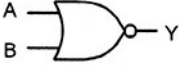


Table 1.3: Truth Table for 16 function of 2 variable

Table 1.4: 16 Logic Microoperation

Table 1.5: Shift Microoperation

Table 1.6: Function table for Arithmetic Logic shift unit

List of symbols

Logic function	Logic symbol	Truth table	Boolean expression															
Buffer		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Y	0	0	1	1	$Y = A$									
A	Y																	
0	0																	
1	1																	
Inverter (NOT gate)		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0	$Y = \bar{A}$									
A	Y																	
0	1																	
1	0																	
2-input AND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \cdot B$
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
2-input NAND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = \overline{A \cdot B}$
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
2-input OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
2-input NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	$Y = \overline{A + B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
2-input EX-OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
2-input EX-NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1	$Y = \overline{A \oplus B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

CHAPTER 1 INTRODUCTION

An arithmetic logic unit (ALU) is combinational rationale circuit which is ordinarily used to execute a CPU's arithmetic and rationale tasks. An ALU will normally take a couple of info operands and yield an outcome alongside a bunch of status bits. A choice info will decide the activity performed.

1.1 Arithmetic Circuit

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Table 1.1

The arithmetic miniature activities recorded in **Table 1.1** can be carried out in one composite arithmetic circuit. The essential segment of a number juggling circuit is the equal viper. By controlling the information contributions to the snake, it is feasible to acquire various kinds of number juggling tasks.

The chart of a 4-cycle arithmetic circuit is appeared in Fig 1.1. It has four full-viper circuits that establish the 4-cycle snake and

four multiplexers for picking various activities. There are two 4-digit inputs A and B and a 4-cycle yield D. The four contributions from A go straightforwardly to the X contributions of the twofold viper. Every one of the four contributions from B are associated with the information contributions of the multiplexers. The multiplexer's information inputs likewise get the supplement of B. The other two information inputs are associated with rationale 0 and rationale 1. Rationale 0 is a fixed voltage esteem (0 volts for TTL coordinated circuits) and the rationale 1 sign can be created through-an inverter whose info is 0. The four multiplexers are constrained by two determination data sources, S1 and S2 thus.

1.1.1 Input Carry

The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next. The output of the binary adder is calculated from the following arithmetic:

$$\text{sum: } D = A + Y + C_{in}$$

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y **inputs of the binary adder**. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_i and S_o **and making C_{in}** equal to 0 or 1, it is possible to generate the eight arithmetic micro operations listed in Table 1.2.

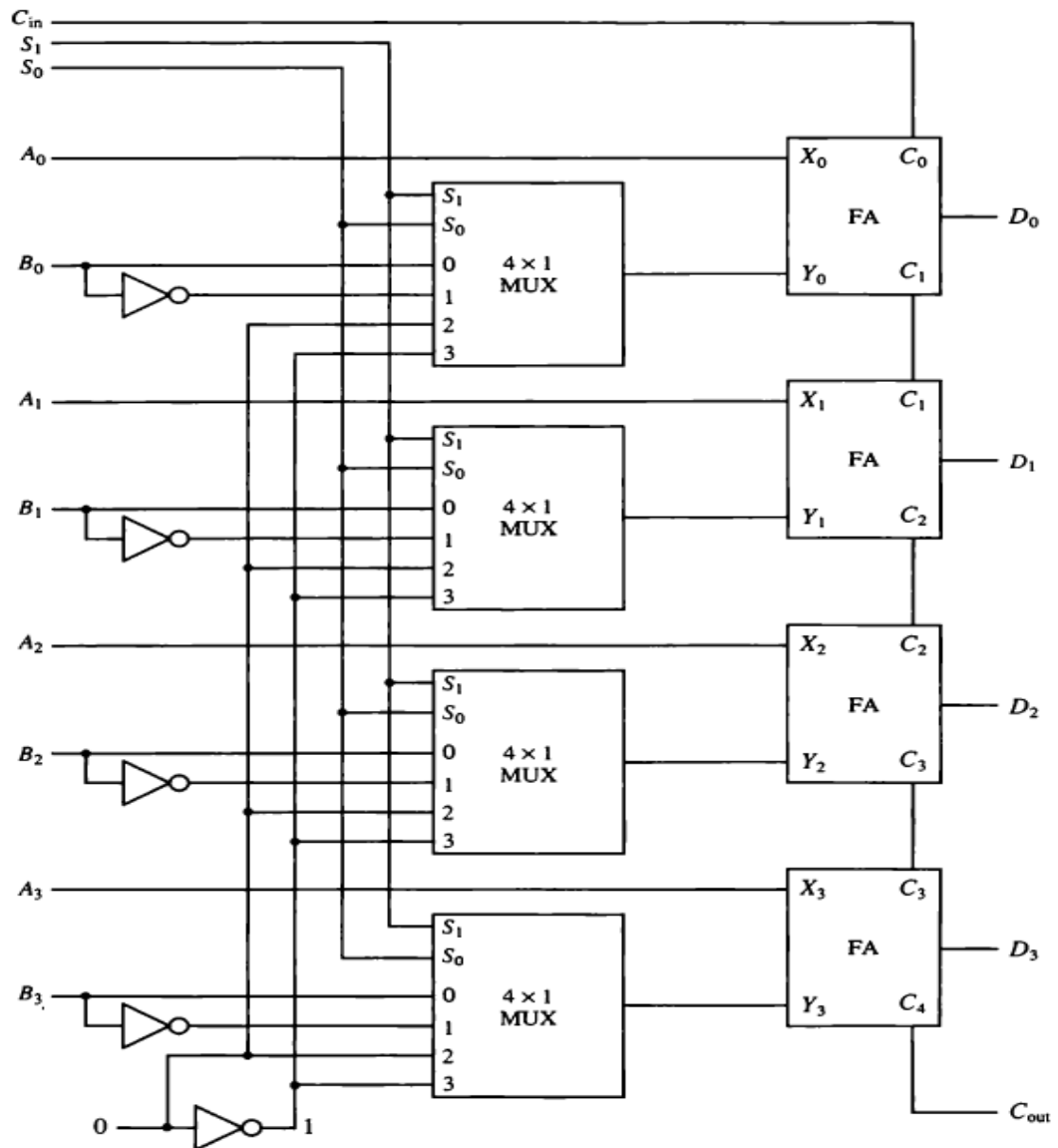


Figure 1.1

S_1	S_0	C_{in}	X
0	0	0	A+B
0	0	1	A+B+1
0	1	0	A+B'
0	1	1	A+B'+1
1	0	0	A
1	0	1	A+1
1	1	0	A-1
1	1	1	A

Table 1.2

1.1.2 Addition

When $S_iS_o = 00$, the value of B is applied to the Y inputs of the adder. If $C_{in} = 0$, the output $D = A + B$. If $C_{in} = 1$, output $D = A + B + 1$. Both cases perform the add microoperation with or without adding the input carry.

1.1.3 Subtraction

When $S_iS_o = 01$, the complement of B is applied to the Y inputs of the adder. If $C_{in} = 1$, then $D = A + B + 1$. This produces A plus the 2's complement of B , which is equivalent to a subtraction of $A - B$. When $C_{in} = 0$, then $D = A + B$. **This is equivalent to** subtract with borrow, that is, $A - B - 1$.

1.1.4 Increment

When $S_iS_o = 10$, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes $D = A + 0 + C_{in}$. This gives $D = A$ when $C_{in} = 0$ and $D = A + 1$ when $C_{in} = 1$. In the first case we have a direct transfer from input A to output D . In the second case, the value of A is incremented by 1.

1.1.5 Decrement

When $S_iS_o = 11$, all 1's are inserted into the Y inputs of the adder to produce the decrement operation $D = A - 1$ when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces $F = A + 2's \text{ complement of } 1 = A - 1$. When $C_{in} = 1$, then $D = A - 1 + 1 = A$, which causes a direct transfer from input A

to output D . Note that the microoperation $D = A$ is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

1.2 Logic Microoperations

Rationale microoperations indicate twofold tasks for series of pieces put away in registers. These activities consider each piece of the register independently and treat them as paired factors. For instance, the selective OR microoperation with the substance of two registers R1 and R2 is represented by the assertion:

$$P: R1 \leftarrow R1 \oplus R2$$

It indicates a rationale microoperation to be executed on the individual pieces of the registers gave that the control variable $P = 1$. As a arithmetic matical model, expect that each register has four pieces. Leave the substance of R1 alone 1010 and the substance of R2 be 1100. The exdusive-OR microoperation expressed above symbolizes the accompanying rationale calculation:

1010	Content of R1
1100	Content of R2
0110	Content of R1 after $P = 1$

The substance of R1, after the execution of the microoperation, is equivalent to the step by step selective OR procedure on sets of pieces in R2 and past estimations of R1. The rationale microoperations are only occasionally utilized in logical calculations, yet they are extremely valuable for digit control of twofold information and for settling on legitimate choices.

1.2.1 Special Symbols

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol will be utilized to signify an OR microoperation and the image A to indicate an AND microoperation. The complement microoperation is equivalent to the I's complement and uses a bar on top of the image that means the register name. By utilizing various images, it will be feasible to separate between a rationale microoperation and a control (or Boolean) work. Another justification receiving two arrangements of images is to have the option to recognize the image + , when used to represent a arithmetic additionally, from a rationale OR activity. Albeit the + image has two implications, it will be feasible to recognize them by taking note of where the image happens. At the point when the image + happens in a microoperation, it will mean a arithmetic in addition to. At the point when it happens in a control (or Boolean) work, it will mean an OR activity. We won't ever utilize it to represent an OR microoperation. For instance, in the articulation

$$P + Q: R1 + - R2 + R3, R4 < - R5 \vee R6$$

the + between P and Q is an OR activity between two parallel factors of a control work. The + somewhere in the range of R2 and R3 determines an add microoperation. The OR microoperation is assigned by the image between registers R5 and R6.

1.2.2 List of Logic Microoperations

There are 16 diverse logic activities that can be performed with two parallel factors. They can be resolved from all conceivable truth tables got with two parallel factors as demonstrated in Table 1.3 . In this table, every one of the 16 segments F0 through F15 addresses a reality table of one potential Boolean capacity for the two factors x and y. Note that the capacities are resolved from the 16 binary combination that can be allocated to F.

x	y	F.	F1	F2	F ₃	F ₄	F ₅	F ₆	F ₇	F ₈	F ₉	F ₁₀	$\frac{F1}{1}$	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 1.3

The 16 Boolean functions of two variables x and y have expressed within the algebraic form within the first column of Table 1.4. The 16 logic micro-operations are derived from these functions by replacing variable x with the binary content of register A and variable y with the binary content of register B. it's important to understand that the Boolean functions listed within the first column of Table 1.4 represent a relation between two binary variables x and y . The logic micro-operations listed within the second column represent a relationship between the binary content of two registers A and B. Each little bit of the register is treated as a binary variable and therefore the microoperation is performed on the string of bits stored within the registers.

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Table 1.4

1.2.3 Hardware Implementation

The hardware implementation of logic micro operations requires that logic gates be inserted for every bit or pair of bits within the registers to perform the specified logic function. Although there are 16 logic microoperations, most computers use only four—AND, OR, XOR (exclusive-OR), and complement—from which all others are often derived.

Figure 1.2 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the specified logic. The outputs of the gates are applied to the info inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i . For a logic circuit with n

bits, the diagram must be repeated n times for $i = 0, 1, 2, \dots, n - 1$. The selection variables are applied to all stages. The function table in Fig. 1.2(b) lists the logic micro operations obtained for each combination of the selected variables.

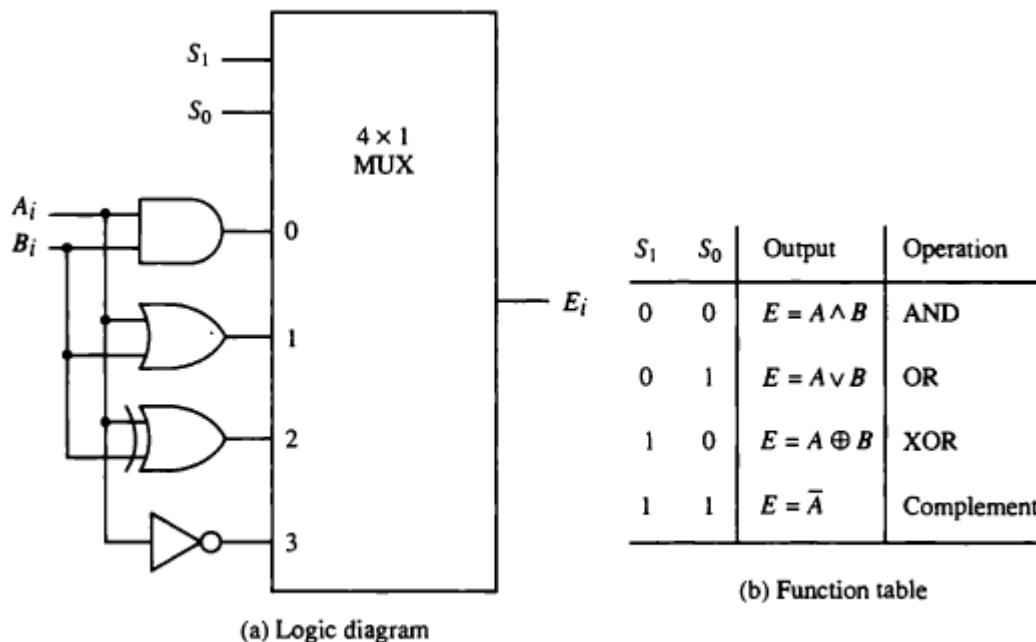


Figure 1.2

1.3 Some Applications

Logic microoperations are very useful for manipulating individual bits or some of a word stored during a register. they will be wont to change bit values, delete a gaggle of bits, or insert new bit values into a register. the subsequent examples show how the bits of 1 register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). during a typical application, register A may be a processor register and therefore the bits of register B

constitute a logic operand extracted from memory and placed in register B.

1.3.1 Selective set

The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. It doesn't affect bit positions that have 0's in B. the subsequent numerical example clarifies this operation:

1010	<i>A</i> before
1100	<i>B</i> (logic operand)
1110	<i>A</i> after

The two leftmost bits of B are 1's, therefore the corresponding bits of A are set to 1. one among these two bits was already set and therefore the other has been changed from 0 to 1. the 2 bits of A with corresponding 0's in B remain unchanged. the instance above is a truth table since it's all four possible combinations of two binary variables. From the reality table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and former values of A. Therefore, the OR microoperation are often wont to selectively set bits of a register.

1010	<i>A</i> before
1100	<i>B</i> (logic operand)
0110	<i>A</i> after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

1.3.3 Selective-clear

The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B . For example:

```

1010 A before
1100 B (logic operand)
0010 A after

```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB' . The corresponding logic microoperation is

$$A \leftarrow A \cdot B'$$

The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B . The mask operation is an AND micro operation as seen from the following numerical example:

```

1010 A before
1100 B (logic operand)
1000 A after masking

```

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. the 2 leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide

an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The insert operation inserts a replacement value into a gaggle of bits. this is often done by first masking the bits then ORing them with the specified value. for instance , suppose that an A register contains eight bits, 0110 1010. to exchange the four leftmost bits by the worth 1001 we first mask the four unwanted bits:

0110 1010

A before

0000 1111

B (mask)

0000 1010

A after masking

and then insert the new value:

0000 1010

A before

1001 0000

B (insert)

1001 1010

A after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words in *A* and *B* and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010	<i>A</i>	
1010		<i>B</i>
0000	<i>A</i> \oplus <i>B</i>	

When *A* and *B* are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

1.4 Shift Microoperations

Shift microoperations are used for serial transfer of knowledge . they're also utilized in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register are often shifted to the left or the proper . At an equivalent time that the bits are shifted, the primary flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a touch into the rightmost position. During a shift-right operation the serial input transfers a touch into the leftmost position. the knowledge transferred through the serial input determines the sort of shift. There are three sorts of shifts: logical, circular, and arithmetic.

A logical shift is one that transfers 0 through the serial input. we'll adopt the symbols shl and shr for logical shift-left and shift-right microoperations. For example:

R1 s—shl R1

R2 F shr R2

are two microoperations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the proper of the content of register R2. The register symbol must be an equivalent on each side of the arrow. The bit transferred to the top position through the serial input is assumed to be 0 during a logical shift.

1.4.1 Circular shift

The circular shift (also referred to as a rotate operation) circulates the bits of the register round the two ends without loss of data . this is often accomplished by connecting the serial output of the register to its serial input. we'll use the symbols cil and cir for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 1.5.

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Table 1.5

1.4.2 Arithmetic shift

An arithmetic shift may be a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the amount by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the amount remains an equivalent when it's multiplied or divided by 2. The leftmost bit during a register holds the sign bit, and therefore the remaining bits hold the amount. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 1.4 shows a typical register of n bits. Bit R_{n-1} within the leftmost position holds the sign bit. R_{n-2} is that the most vital little bit of the amount and R_0 is that the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the amount (including the sign bit) to the proper. Thus R_{n-1} remains an equivalent, R_{n-2} receives the bit from R_{n-1} , then on for the opposite bits within the register. The bit in R_0 is lost.

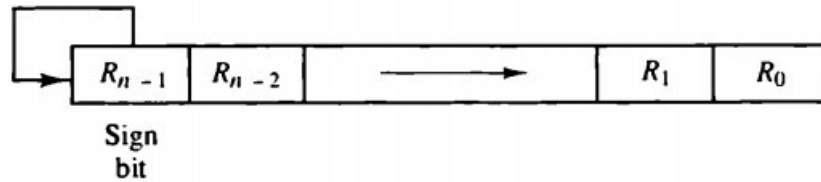


Figure 1.3

The arithmetic shift-left inserts a 0 into R_0 , and shifts all other bits to the left. The initial little bit of R_{n-1} is lost and replaced by the bit from R_{n-2} . a symbol reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} isn't adequate to R_{n-2} . An overflow flip-flop V_5 are often wont to detect an arithmetic shift-left overflow.

$$V_5 = R_{n-1} \oplus R_{n-2}$$

If $V_5 = 0$, there's no overflow, but if $V_5 = 1$, there's an overflow and a symbol reversal after the shift. V_5 must be transferred into the overflow flip-flop with an equivalent clock pulse that shifts the register.

1.4.3 Hardware Implementation

A possible choice for a shift unit would be a bidirectional register with parallel load. Information are often transferred to the register in parallel then shifted to the proper or left. during this sort of configuration, a clock pulse is required for loading the info into the register, and another pulse is required to initiate the shift. during a processor unit with many registers it's more efficient to implement the shift operation with a combinational circuit. during this way the content of a register that has got to be shifted is first placed onto a standard bus whose output is

connected to the combinational shifter, and therefore the shifted number is then loaded back to the register. this needs just one clock pulse for loading the shifted value into the register.

1.5 Shifter

A combinational circuit shifter are often constructed with multiplexers as shown in Fig.1.5. The 4-bit shifter has four data inputs, A_0 through A_3 , and four data outputs, H_0 through H_3 . There are two serial inputs, one for shift left (0 and therefore the other for shift right (1i). When the choice input $S = 0$, the input file are shifted right (down within the diagram). When $S = 1$, the input file are shifted left (up within the diagram). The function table in Fig. 1.5 shows which input goes to every output after the shift. A shifter with n data inputs and outputs requires n multiplexers. the 2 serial inputs are often controlled by another multiplexer to supply the three possible sorts of shifts.

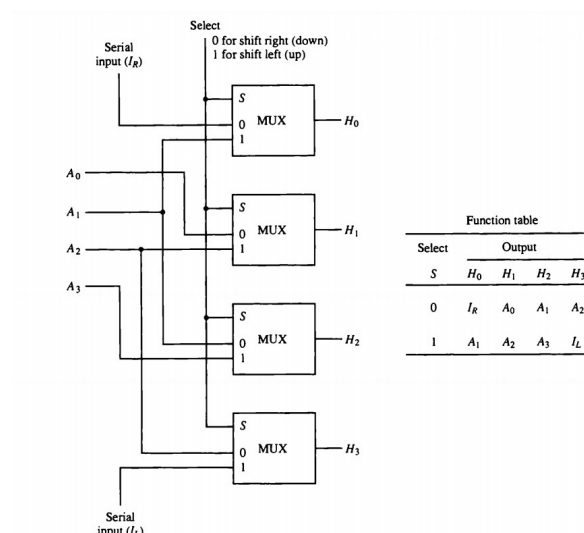


Figure 1.4

1.6 Arithmetic Logic Shift Unit

Instead of having individual registers performing the microoperations directly, computer systems employ sort of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed within the inputs of the common ALU. The ALU performs an operation and thus the results of the operation is then transferred to a destination register. The ALU could also be a combinational circuit so as that the entire register transfer operation from the source registers through the ALU and into the destination register are often performed during one clock pulse period. The shift microoperations are often performed during a separate unit, but sometimes the shift unit is made a neighborhood of the overall ALU,

The arithmetic, logic, and shift circuits introduced in previous sections are often combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 1.6. The subscript i designates a typical stage. inputs are applied to both the arithmetic and logic units. a selected microoperation is chosen with inputs S_3, S_2, S_1, S_0 . A 4×1 multiplexer at the output chooses between an arithmetic output in E , and a logic output in H . the data within the multiplexer are selected with inputs S_3 and S_2 . the other two data inputs to the multiplexer receive inputs A_{i-1} , for the shift-right operation and $A_i + 1$ for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig. 1.6 must be repeated n times for an n -bit ALU. The output carry C_i , of a given arithmetic stage must be connected to the input carry C_{i+1} , of subsequent stage in sequence. The input carry to the first stage is that the input carry C_0 , which provides a spread variable for the arithmetic operations.

The circuit whose one stage is laid call at Fig. 1.6 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is chosen with the five variables S_3 , S_2 , S_1 , S_0 , and C_i . The input carry C_u , is used for selecting an mathematical operation only.

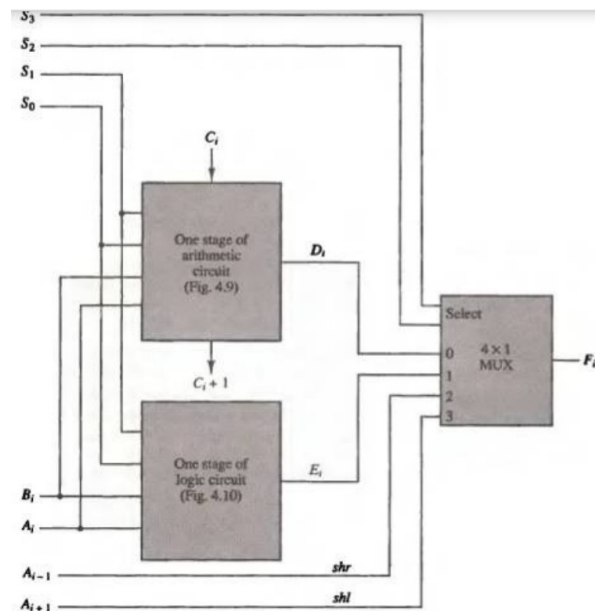


Figure 1.5

Table 1.4 lists the 14 operations of the ALU. the primary eight are arithmetic operations and are selected with $S_3S_2 = 00$. subsequent four are logic operations and are selected with $S_3S_2 = 01$. The input carry has no effect during the logic operations and is marked with don't-care x's. The last two operations are shift operations and are selected with $S_3S_2 = 10$ and 11 . the opposite three selection inputs haven't any effect on the shift.

S3	S2	S1	S0	Cin	X
0	0	0	0	0	A+B
0	0	0	0	1	A+B+1
0	0	0	1	0	A+B'
0	0	0	1	1	A+B'+1
0	0	1	0	0	A
0	0	1	0	1	A+1
0	0	1	1	0	A-1
0	0	1	1	1	A
0	1	0	0	X	$A \wedge B$
0	1	0	1	X	$A \vee B$
0	1	1	0	X	$A \oplus B$
0	1	1	1	X	A'
1	0	X	X	X	<< A
1	1	X	X	X	>> A

Table 1.6

CHAPTER 2 RESULTS & CONCLUSION

Multisim Model

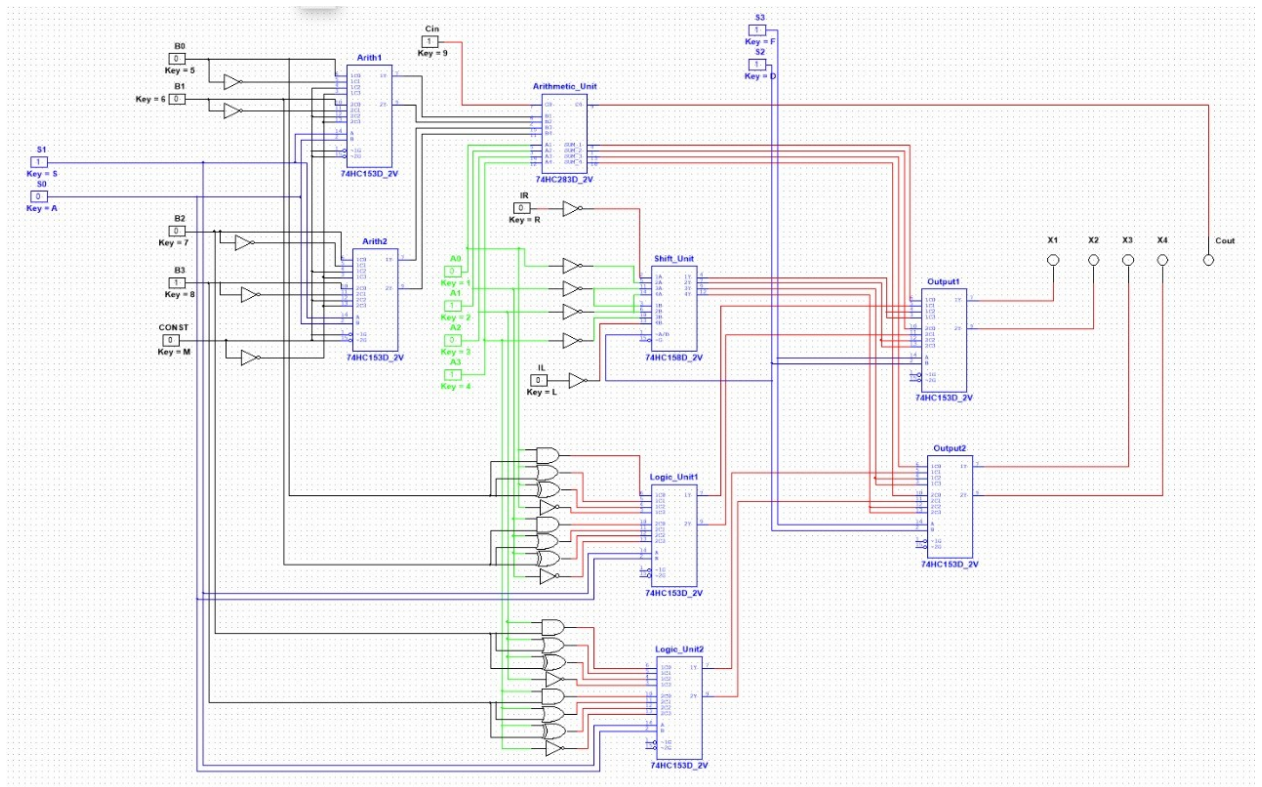


Figure 2.1

In conclusion, our 4-bit ALU design and implementation worked correctly. The methods we used to design each component in the system made for proper execution of the ALU. While at first we had issues switching individual registers performing the microoperations directly, between them. Once we figured this out, the project worked correctly. This project let us use the knowledge we have gained over the past labs.

References

- https://en.wikipedia.org/wiki/Arithmetic_logic_unit
- <https://www.tutorialspoint.com/arithmetic-logic-unit-alu>
- <https://ece.umaine.edu/wp-content/uploads/sites/203/2012/05/ALU.pdf>
- <https://www.geeksforgeeks.org/introduction-of-alu-and-data-path/>
- <https://www.youtube.com/watch?v=h8PAobl4xUk>