NAND to Logic Circuit – Design of ALU and Memory in Simulator using HDL

Project Report

Submitted in partial fulfillment of the requirement
for the award of the degree of
Bachelor of Technology in Electronics Engineering

By

ADEEB ALI
22ELB521

ZAIF
22ELB469

MOHD. ANAS
22ELB524

Supervisor
Dr.TAJINDER SING ARORA

DEPARTMENT OF ELECTRONICS ENGINEERING ZAKIR HUSAIN COLLEGE OF ENGINEERING & TECHNOLOGYALIGARH MUSLIM UNIVERSITY ALIGARH ALIGARH-202002 (INDIA)

STUDENTS’ DECLARATION

I here by certify that the work presented in this project report entitled NAND to Logic Circuit – Design of ALU and Memory in Simulator using HDL, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics Engineering, is an authentic record of my own work carried out during the Third year of B.Tech. under the guidance of Dr.Tajinder Singh Arora, Professor, Department of Electronics Engineering, Zakir Husain College of Engineering & Technology, Aligarh Muslim University, Aligarh.

The project demonstrates the systematic design of digital logic Circuits, starting from fundamental NAND gates to complex components such as the Arithmetic Logic Unit (ALU) and memory modules, implemented and verified using Hardware Description Language (HDL). The work adheres to modular design principles and has been validated through functional simulation.

(Adeeb Ali) **(Zaif)** **(Mohd Anas)**

(Dr. TAJINDER SING ARORA)
Project Guide

Date: 16/04/2025

ABSTRACT

The growing complexity of digital systems necessitates a fundamental understanding of logic design, from basic gates to complex computing components. This project explores the hierarchical design of digital circuits, starting from the NAND gate as a universal building block, and extends to the implementation of an Arithmetic Logic Unit (ALU) and memory modules using Hardware Description Language (HDL). The primary objective is to demonstrate the systematic construction of computational elements, emphasizing modularity, efficiency, and scalability in digital design.

A bottom-up approach is adopted, beginning with the implementation of basic logic gates (AND, OR, NOT, XOR) using NAND gates, followed by the integration of these gates into higher-level components such as multiplexers, adders, and shifters. These components are then combined to design a functional ALU capable of performing arithmetic (addition,subtraction) and logical (AND, OR, NOT, XOR) operations. Additionally, memory units, including registers and RAM, are implemented to support data storage and retrieval, forming a foundational memory hierarchy.

The design is simulated and verified using industry-standard HDL tools, ensuring correctness in functionality and timing behavior. By synthesizing these components in a simulator, this project highlights the transition from theoretical logic design to practical hardware implementation. The results validate the feasibility of constructing complex digital systems from basic gates while adhering to structured design principles.

This work serves as an educational framework for understanding digital circuit design, offering insights into processor architecture and memory systems. Future enhancements may include pipelining, cache integration, and expanded instruction set support, further bridging the gap between foundational logic design and advanced computer architecture.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my project supervisor, Dr.TAJINDER SING ARORA, for his invaluable guidance, constant encouragement, and unwavering support throughout the completion of this project. His deep expertise, insightful suggestions, and patient mentorship were instrumental in shaping this work. I am truly grateful for the time he devoted to reviewing my progress, providing constructive feedback, and helping me overcome challenges at every stage.

I would also like to extend my heartfelt thanks to Dr. Anwar Sadat for his support and technical advice, which greatly enriched my understanding of digital logic design and HDL implementation. His encouragement and motivation kept me focused on achieving the objectives of this project.

I am deeply thankful to the Department of Electronics Engineering, Zakir Husain College of Engineering & Technology, Aligarh Muslim University, for providing the necessary resources and an excellent academic environment that facilitated this work.

Lastly, I would like to acknowledge my family and friends for their constant encouragement, patience, and moral support, which kept me motivated throughout this journey.

Zaif

Md. Anas

Adeeb Ali

NAND to Logic Circuit - Design of ALU and Memory in Simulator using HDL

TABLE OF CONTENTS

Chapter 1: Basic Logic Gates using NAND

1.1 Introduction to NAND Logic
- Explanation of NAND gate universality - Significance of constructing all logic circuits from NAND
1.2 Gate Construction from NAND
- Designing basic gates: AND, OR, NOT, XOR, etc. - Circuit diagrams for each gate - Truth tables for all gates
1.3 HDL Implementation and Testing
- Implementation using Tetris HDL - Code snippets for all gates - Test benches and simulation outputs - Verification with expected results

Chapter 2: Foundation of ALU and Memory Design
2.1 Combinational Building Blocks
- Design and logic of:
- Half Adder
- Full Adder
- Multiplexers (MUX)
- Demultiplexers (DEMUX)
- Add16 (16-bit Adder)

2.2 Truth Tables and Diagrams
- Complete truth tables for each logic block
- Hand-drawn or simulated circuit diagrams
2.3 HDL Implementation
- HDL code using Tetris language for each module
- Test bench logic and verification
- Error handling and debugging process
Chapter 3: ALU and Memory Development
3.1 Arithmetic Logic Unit (ALU)
- Functions supported: Add, AND, NOT, Zero, Negative flags

- ALU control bits and design architecture
- Complete circuit diagram
- Truth table showing outputs for various input conditions

3.2 Memory Design Using D Flip-Flop

- Building RAM8 and RAM64 from basic flip-flop designs
- Address selection mechanism
- Read/Write functionality and data storage logic

3.3 Complete HDL and Simulation

- Code structure for ALU and memory
- Integrated testing of all modules
- Truth table and waveform analysis from HDL simulator

Conclusion and Future Scope

- Summary of the components built using only NAND
- Learning outcomes and conceptual clarity
- Limitations in current design
- Future work: building a basic CPU, adding assembler, control logic, ROM, etc

LIST OF FIGURES

Fig. 1.0 Circuit Diagram and Truth Table of NAND Gate

Fig. 1.1 Circuit Diagram and Truth Table of NOT Gate

Fig. 1.2 Circuit Diagram and Truth Table of AND Gate

Fig. 1.3 Circuit Diagram and Truth Table of OR Gate

Fig. 1.4 Circuit Diagram and Truth Table of XOR Gate

Fig. 2.0 Circuit Diagram and Truth Table of Mux

Fig. 2.1 Circuit Diagram and Truth Table of Demux

Fig. 2.2 Diagram of Or8Way Logic

Fig. 2.3 Circuit Diagram and Truth Table of Full Adder

Fig. 2.4 Block Diagram of 16-bit Adder (Add16)

Fig. 2.5 Block Diagram of Incrementer

Fig. 2.6 Circuit Diagram and Truth Table of D Flip-Flop

Fig. 3.0 Hardware Design and Operation of ALU

Fig. 3.1 RAM8 Circuit Diagram

Fig. 3.b Hardware Definition of Memory (Bit, Register)

List of Abbreviations

HDL - Hardware Description Language

ALU - Arithmetic Logic Unit

RAM - Random Access Memory

XOR - Exclusive OR (Logic Gate)

XNOR - Exclusive NOR (Logic Gate)

AND- Logical Conjunction

OR-Logical Disjunction

DEMUX-Demultiplexer

MUX - Multiplexer

D Flip-Flop - Data or Delay Flip-Flop

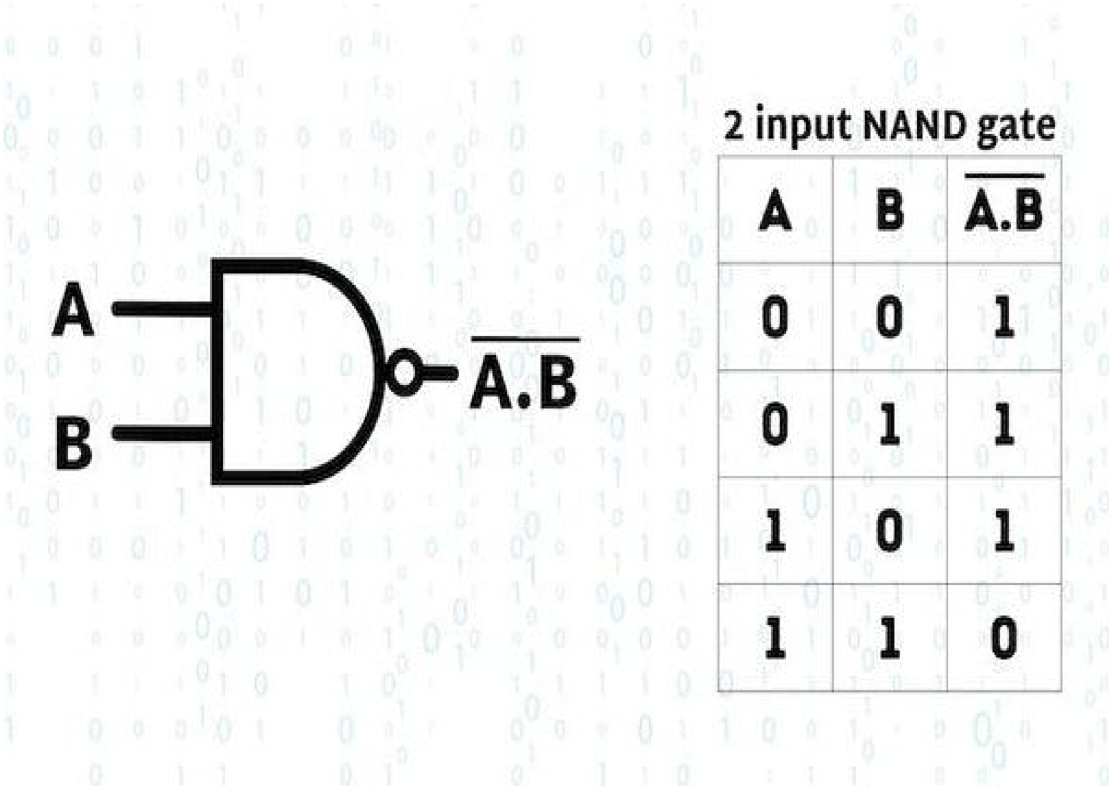
REG - Register

ADDR - Address

CHAPTER 1:Introduction to Logic Design Using NAND Gate

Design and Analysis of Basic Logic Circuits:

1. Nand Gate

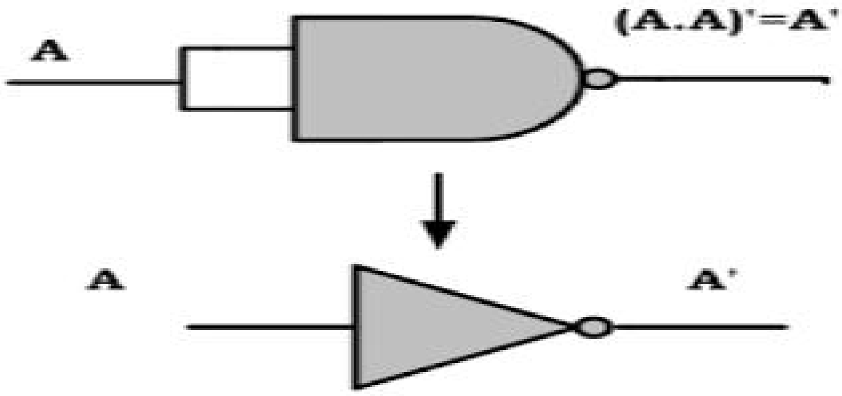


A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

Fig(1.0):Circuit Diagram and Truth Table

Assumption:This assumption is based on the principle of *functional completeness*. The NAND gate is functionally complete, meaning that any other basic logic gate (such as NOT, AND, OR, NOR, XOR, and XNOR) can be constructed using only NAND gates. This makes it possible to design and implement any digital logic circuit using just NAND gates.

1.1 Not Gate



Input	Output
A	Y
0	1
1	0

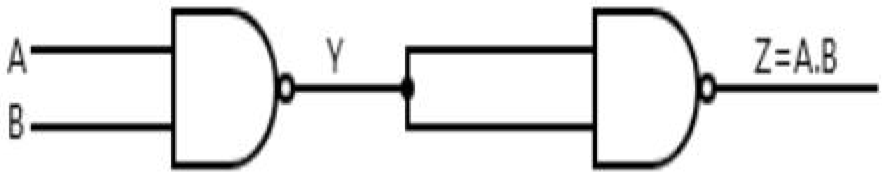
Fig(1.1)-Circuit Diagram and Truth Table

Code
Nand2Tetris HDL (a simplified Hardware Description Language):

```
CHIP Not {
    IN in;
    OUT out;
    PARTS:
 Nand(a=in,b=in,out=out);}
```

1.2

1.2 And Gate



Input		Output
A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

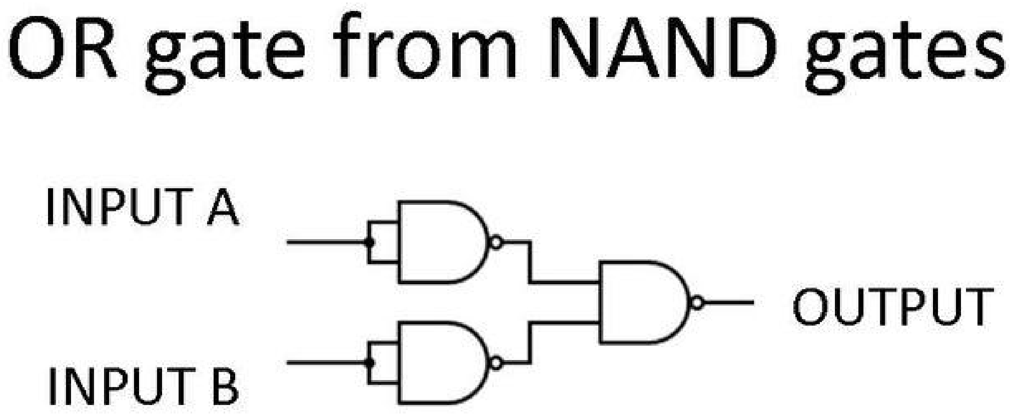
Fig(1.3)-Circuit Diagram and Truth Table

```
Code
Nand2Tetris HDL (a simplified Hardware Description Language):
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
    Nand(a=a ,b=b ,out= NandOut);
    Not(in=NandOut ,out=out );

}
```

Test Bench:

```
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

1.3 Or Gate



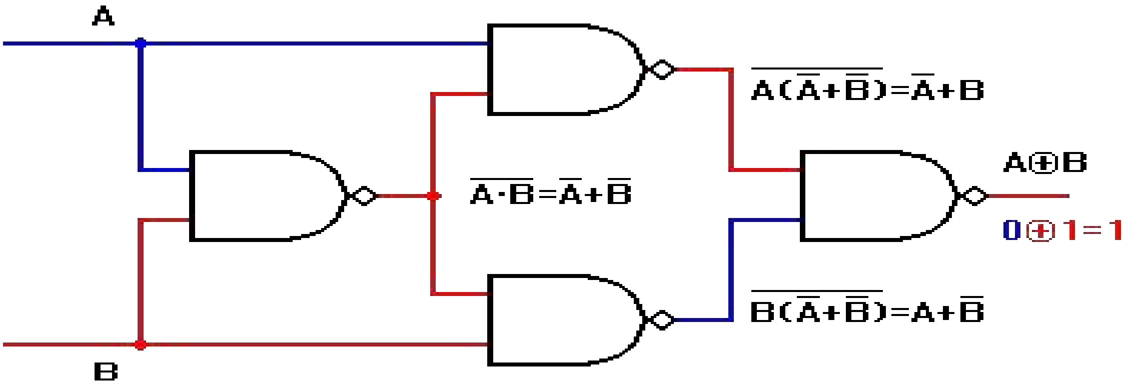
Input		Output
A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

Fig(1.3)-Circuit Diagram and Truth Table

```
Code
Nand2Tetris HDL (a simplified Hardware Description Language):
CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
     Not(in=a ,out=NotA );
    Not(in=b ,out=NotB );
    And(a=NotA ,b=NotB ,out=ABareBothZero);
    Not(in=ABareBothZero ,out=out);

}
```

```
//Test_Bench
load Or.hdl,
output-file Or.out,
compare-to Or.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

1.4 Xor Gate :



Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Fig(1.4)-Circuit Diagram and Truth Table

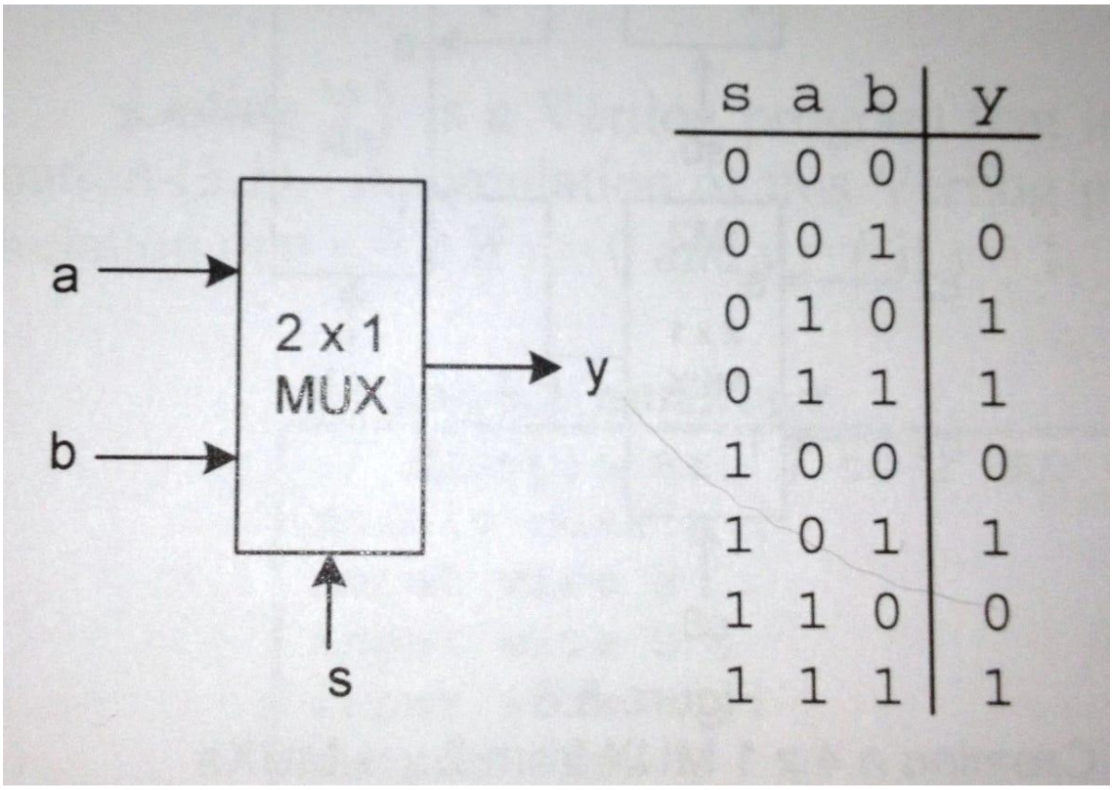
```
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not(in=a, out=notA);
    Not(in=b, out=notB);
    And(a=a, b=notB, out=aAndNotB);
    And(a=notA, b=b, out=notAAndB);
    Or(a=aAndNotB, b=notAAndB, out=out);
}
```

Truth Table:

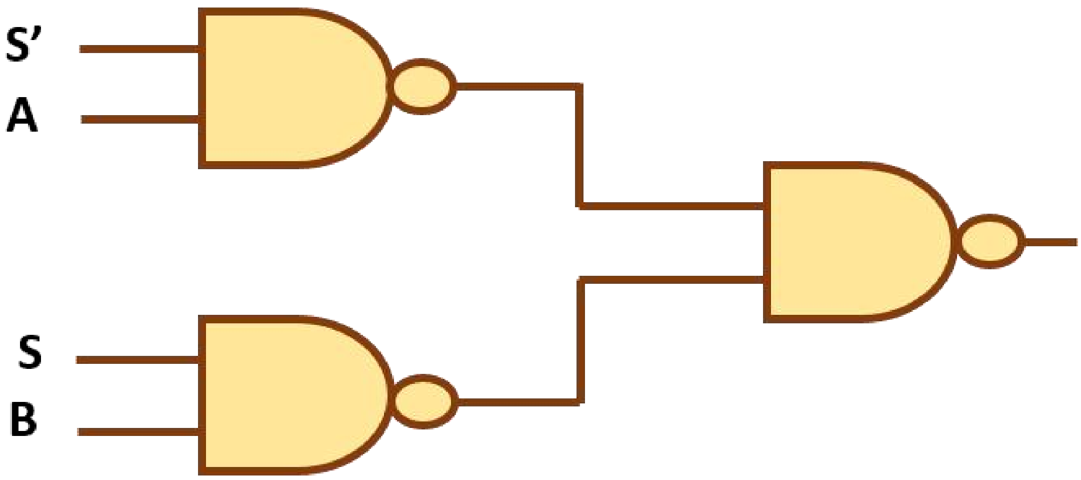
```
load Xor.hdl,
compare-to Xor.cmp,
output-list a b out;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Chapter 2-**foundation of Arithmetic and Logic operations**

2 Mux:



s	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

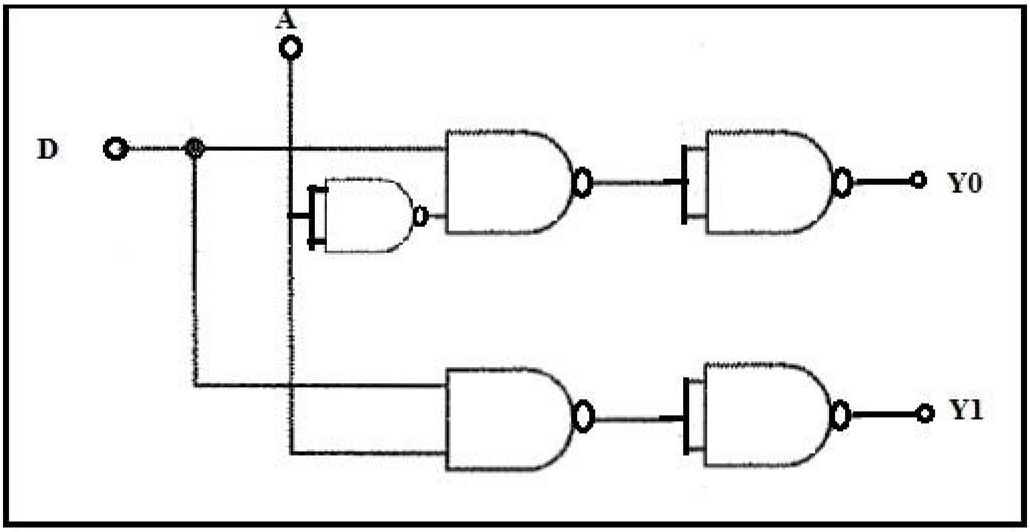


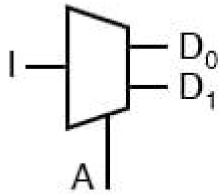
Fig(2.0)-Circuit Diagram and Truth Table

```
//Code
CHIP Mux {
    IN a, b, sel;
    OUT out;
    PARTS:
    Not(in=sel, out=notSel);
    And(a=a, b=notSel, out=aAndNotSel);
    And(a=b, b=sel, out=bAndSel);
    Or(a=aAndNotSel, b=bAndSel, out=out);
}
```

```
set a 0, set b 0, set sel 0;
eval, output;

set sel 1;
eval, output;

set a %B0000000000000000, set b %B0001001000110100, set sel 0;
eval, output;

set sel 1;
eval, output;

set a %B1001100001110110, set b %B0000000000000000, set sel 0;
eval, output;

set sel 1;
eval, output;

set a %B1010101010101010, set b %B0101010101010101, set sel 0;
eval, output;

set sel 1;
eval, output;
```

2. 1Demux





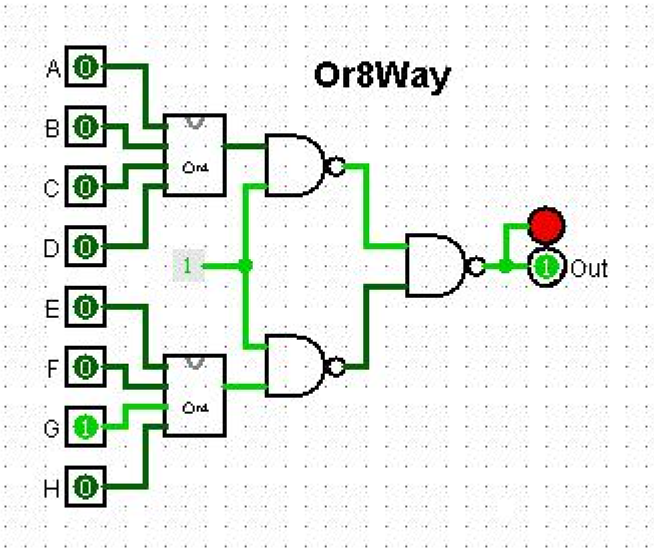
E	A	B	D0	D1	D2	D3
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Fig(2.1)-Circuit Diagram and Truth Table

```
CHIP DMux {
    IN in, sel;
    OUT a, b;
    PARTS:
    Not(in=sel, out=notSel);
    And(a=in, b=notSel, out=a);
    And(a=in, b=sel, out=b);
}
```

```
load DMux.hdl,
compare-to DMux.cmp,
output-list in sel a b;

set in 0,
set sel 0,
eval,
output;

set sel 1,
eval,
output;

set in 1,
set sel 0,
eval,
output;

set sel 1,
eval,
output;
```

2.2 Or8Way:



Or8Way -is an **8-input logic gate** that returns 1 if **at least one of the 8 inputs** is 1, and 0 otherwise

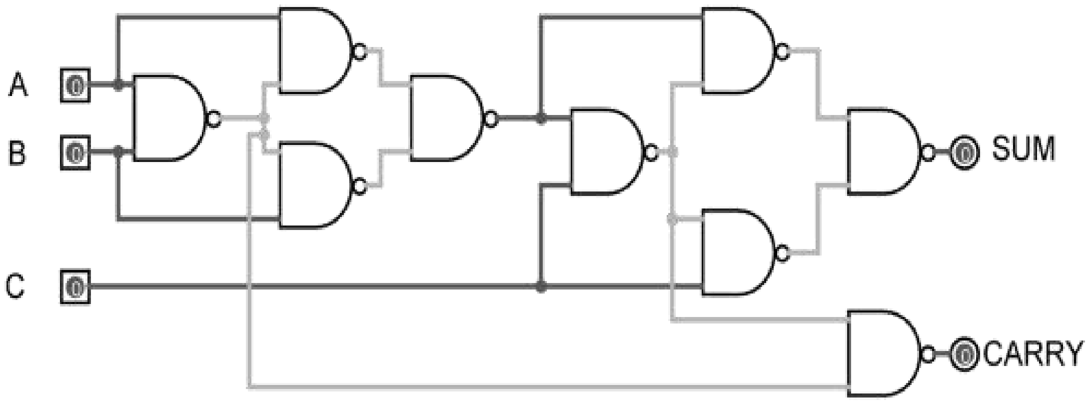
hdl

```hdl
CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
    Or(a=in[0], b=in[1], out=or1);
    Or(a=or1, b=in[2], out=or2);
    Or(a=or2, b=in[3], out=or3);
    Or(a=or3, b=in[4], out=or4);
    Or(a=or4, b=in[5], out=or5);
    Or(a=or5, b=in[6], out=or6);
    Or(a=or6, b=in[7], out=out);
}
```

Test Bench:

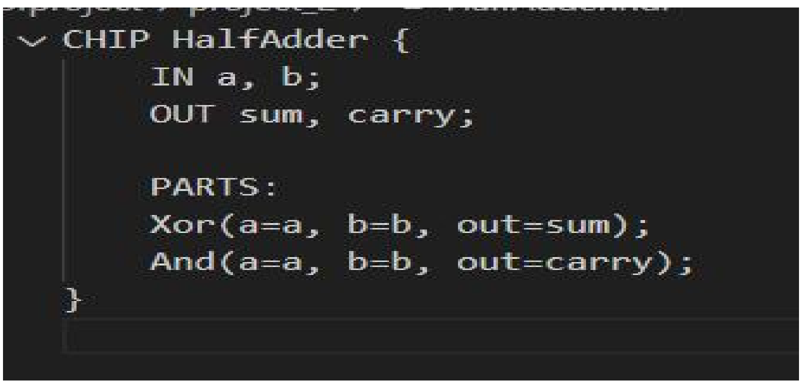
```
load Or8Way.hdl,
compare-to Or8Way.cmp,
output-list in%B2.8.2 out;

set in %B00000000,
eval,
output;

set in %B11111111,
eval,
output;

set in %B00010000,
eval,
output;

set in %B00000001,
eval,
output;

set in %B00100110,
eval,
output;
```

2.3 Full adder:



Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig(2.3)-Circuit Diagram and Truth Table



```
CHIP HalfAdder {
    IN a, b;
    OUT sum, carry;

    PARTS:
    Xor(a=a, b=b, out=sum);
    And(a=a, b=b, out=carry);
}
```

```
CHIP FullAdder {
    IN a, b, c;
    OUT sum, carry;

    PARTS:
    HalfAdder(a=a, b=b, sum=s1, carry=c1);
    HalfAdder(a=s1, b=c, sum=sum, carry=c2);
    Or(a=c1, b=c2, out=carry);
}
```

```
// FullAdder test script (condensed version)

load FullAdder.hdl,
compare-to FullAdder.cmp,
output-list a b c sum carry%B2.1.2;

// Test all 8 input combinations
set a 0, set b 0, set c 0, eval, output;
set a 0, set b 0, set c 1, eval, output;
set a 0, set b 1, set c 0, eval, output;
set a 0, set b 1, set c 1, eval, output;
set a 1, set b 0, set c 0, eval, output;
set a 1, set b 0, set c 1, eval, output;
set a 1, set b 1, set c 0, eval, output;
set a 1, set b 1, set c 1, eval, output;
```

2.4-ADD 16:

Add16 is a **combinational logic circuit** that performs **binary addition** on two 16-bit inputs.

inputs: a[16], b[16] — Two 16-bit binary numbers

Output: out[16] — The 16-bit sum of a + b

Code

```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    FullAdder(a=a[0], b=b[0], c=false, sum=out[0], carry=c1);
    FullAdder(a=a[1], b=b[1], c=c1, sum=out[1], carry=c2);
    FullAdder(a=a[2], b=b[2], c=c2, sum=out[2], carry=c3);
    FullAdder(a=a[3], b=b[3], c=c3, sum=out[3], carry=c4);
    FullAdder(a=a[4], b=b[4], c=c4, sum=out[4], carry=c5);
    FullAdder(a=a[5], b=b[5], c=c5, sum=out[5], carry=c6);
    FullAdder(a=a[6], b=b[6], c=c6, sum=out[6], carry=c7);
    FullAdder(a=a[7], b=b[7], c=c7, sum=out[7], carry=c8);
    FullAdder(a=a[8], b=b[8], c=c8, sum=out[8], carry=c9);
    FullAdder(a=a[9], b=b[9], c=c9, sum=out[9], carry=c10);
    FullAdder(a=a[10], b=b[10], c=c10, sum=out[10], carry=c11);
    FullAdder(a=a[11], b=b[11], c=c11, sum=out[11], carry=c12);
    FullAdder(a=a[12], b=b[12], c=c12, sum=out[12], carry=c13);
    FullAdder(a=a[13], b=b[13], c=c13, sum=out[13], carry=c14);
    FullAdder(a=a[14], b=b[14], c=c14, sum=out[14], carry=c15);
    FullAdder(a=a[15], b=b[15], c=c15, sum=out[15], carry=ignore);
}
```

2.5-Increament :

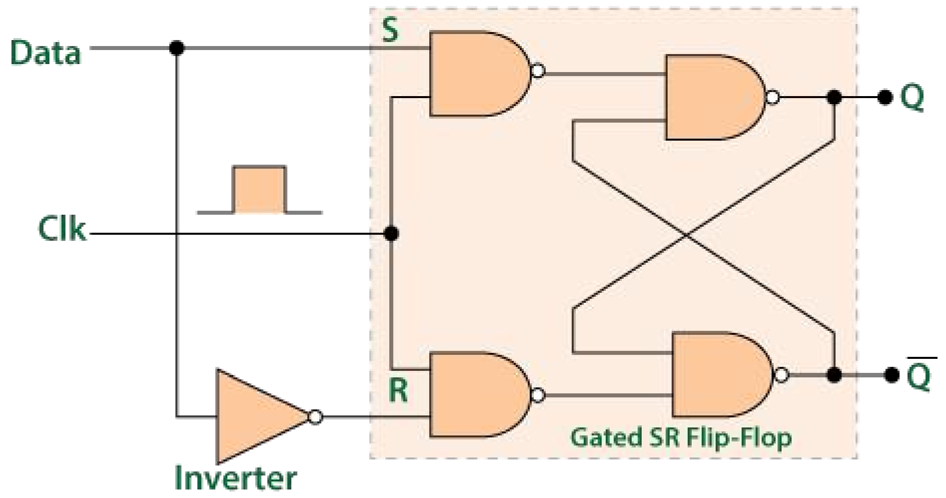
Code:

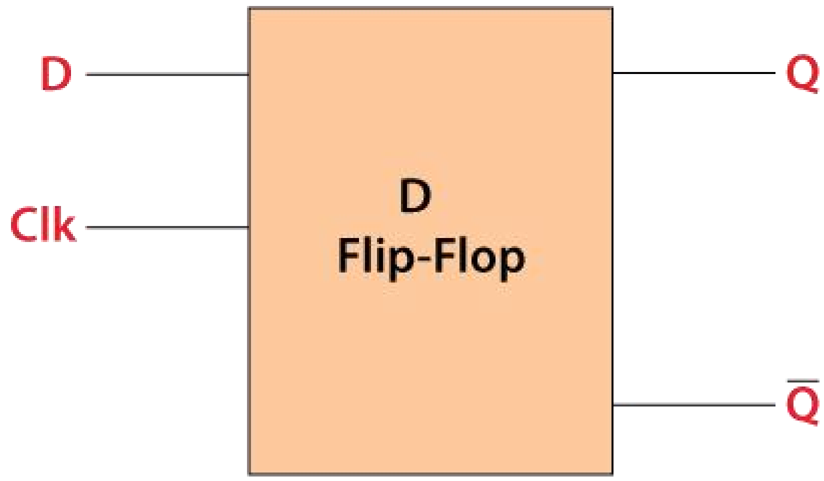
```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Add16(a=in,
          b[0]=true, b[1]=false, b[2]=false, b[3]=false,
          b[4]=false, b[5]=false, b[6]=false, b[7]=false,
          b[8]=false, b[9]=false, b[10]=false, b[11]=false,
          b[12]=false, b[13]=false, b[14]=false, b[15]=false,
          out=out);
}
```

Test bench:

```
load Inc16.hdl,
compare-to Inc16.cmp,
output-list in%B1.16.1 out%B1.16.1;

set in %B0000000000000000,  // in = 0
eval,
output;

set in %B1111111111111111,  // in = -1
eval,
output;

set in %B0000000000000101,  // in = 5
eval,
output;

set in %B1111111111111011,  // in = -5
eval,
output;
```

2.6 D_Flip_Flop :



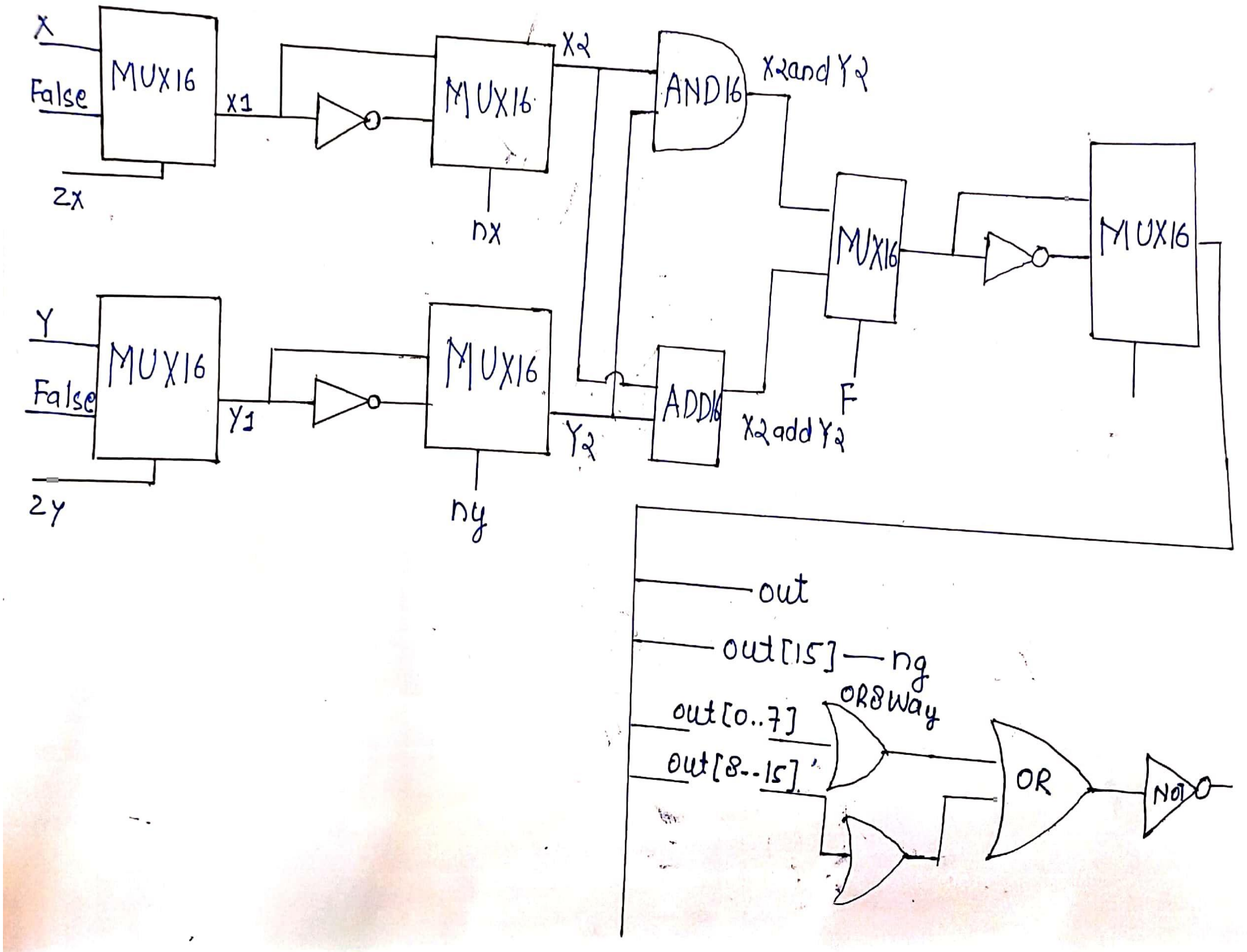


Fig(2.6)-Circuit diagram ,D_FF Symbole and Truth Table

A **D flip-flop** (Data or Delay flip-flop) is a **sequential logic circuit** that stores **1 bit** of data. It's a **clocked memory element**, meaning it updates its output only when the **clock signal** triggers it

Chapter 3 part a-ALU (Arithmetic Logic

Unit):


X
False
MUX16
2X
X1
MUX16
nX
X2
AND16
X2and Y2
Y
False
MUX16
2Y
Y1
MUX16
ny
Y2
ADD16
X2 add Y2
MUX16
F
MUX16
out
out[15] — ng
OR8Way
out[0..7]
out[8..15]
OR
NOT

To cause the ALU to compute a function, set the control bits to the binary combination listed in the table.

0	1	0	1	0	1	x|y
0	0	0	1	1	1	y-x
for all 2 codes x$y						

2x	nx	2y	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Fig(3.0)-Hardware design and Operation &Chip of ALU

Certainly! You're referring to the fact that the **ALU**(Arithmetic Logic Unit) in your code is capable of performing multiple operations, and these operations are **defined and controlled** by the input signals (such as zx, nx, zy, ny, f, and no).

The **operations** performed by the ALU are often defined in a **truth table** that specifies how the inputs control the ALU's behavior. Here's an **elaboration** on that context, where we can break down each of the operations and how they are controlled through the inputs.

Code & Test_Bench

```
//CODE OF ALU
CHIP ALU {
    IN x[16], y[16], zx, nx, zy, ny, f, no;
    OUT out[16], zr, ng;

    PARTS:
    //make negations
    Not16(in=Xa ,out=notX );
    Not16(in=Ya ,out=notY );
    Not16(in=resultA ,out=notResultA);

    //zeros
    Mux16(a=x ,b=false ,sel=zx ,out=Xa );
    Mux16(a=y ,b=false ,sel=zy ,out=Ya );

    //negations
    Mux16(a=Xa ,b=notX ,sel=nx ,out=Xb );
    Mux16(a=Ya ,b=notY ,sel=ny ,out=Yb );

    ///opertions
    And16(a=Xb ,b=Yb ,out=and );
    Add16(a=Xb ,b=Yb  ,out=sum );

    //choose operation
    Mux16(a=and ,b=sum ,sel=f ,out=resultA);

    //choose result or negation
    Mux16(a=resultA ,b=notResultA ,sel=no ,out[0..7]=resultI, out[8..15]=resultII, out=out, out[15]=sigBit);

    // set zr if zero
    Or8Way(in=resultI ,out=orA );
    Or8Way(in=resultII ,out=orB );
    Or(a=orA ,b=orB ,out=notZeroResult );
    Not(in=notZeroResult ,out=zr);

    // set ng if most sig bit is 1
    And(a=true ,b=sigBit ,out=ng );

}
```

```
//ZX
Mux16(a=x,b=false,sel=zx,out=X1);
    Mux16(a=y,b=false,sel=zy,out=Y1);

    //ZY
    Not16(in=X1,out=notX1);
    Not16(in=Y1,out=notY1);
    Mux16(a=X1,b=notX1,sel=nx,out=X2);
    Mux16(a=Y1,b=notY1,sel=ny,out=Y2);

    //Function a^b&&a+b
    And16(a=X2,b=Y2,out=X2andY2);
    Add16(a=X2,b=Y2,out=X2addY2);
    Mux16(a=X2andY2,b=X2addY2,sel=f,out=out1);

    //outnot
        Not16(in=out1, out=notOut1);
    Mux16(a=out1, b=notOut1, sel=no, out=out);

    //check zero
    Or8Way(in=out[0..7],out=or1);
    Or8Way(in=out[0..7], out=or1);
    Or8Way(in=out[8..15],out=or2);
    Or(a=or1,b=or2,out=notzr);
    Not(in =notzr,out=zr);

     And(a=out[15], b=true, out=ng);
```

```
load ALU.hdl,
compare-to ALU.cmp,
output-list x%B1.16.1 y%B1.16.1 zx nx zy ny f no out zr ng;

set x %B0000000000000000, set y %B1111111111111111;

// Compute 0
set zx 1, set nx 0, set zy 1, set ny 0, set f 1, set no 0, eval, output;

// Compute 1
set zx 1, set nx 1, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute -1
set zx 1, set nx 1, set zy 1, set ny 0, set f 1, set no 0, eval, output;

// Compute x
set zx 0, set nx 0, set zy 1, set ny 1, set f 0, set no 0, eval, output;

// Compute y
set zx 1, set nx 1, set zy 0, set ny 0, set f 0, set no 0, eval, output;

// Compute !x
set zx 0, set nx 0, set zy 1, set ny 1, set f 0, set no 1, eval, output;

// Compute !y
set zx 1, set nx 1, set zy 0, set ny 0, set f 0, set no 1, eval, output;

// Compute -x
set zx 0, set nx 0, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute -y
set zx 1, set nx 1, set zy 0, set ny 0, set f 1, set no 1, eval, output;
```

```
// Compute x + 1
set zx 0, set nx 1, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute y + 1
set zx 1, set nx 1, set zy 0, set ny 1, set f 1, set no 1, eval, output;

// Compute x - 1
set zx 0, set nx 0, set zy 1, set ny 1, set f 1, set no 0, eval, output;

// Compute y - 1
set zx 1, set nx 1, set zy 0, set ny 0, set f 1, set no 0, eval, output;

// Compute x + y
set zx 0, set nx 0, set zy 0, set ny 0, set f 1, set no 0, eval, output;

// Compute x - y
set zx 0, set nx 1, set zy 0, set ny 0, set f 1, set no 1, eval, output;

// Compute y - x
set zx 0, set nx 0, set zy 0, set ny 1, set f 1, set no 1, eval, output;

// Compute x & y
set zx 0, set nx 0, set zy 0, set ny 0, set f 0, set no 0, eval, output;

// Compute x | y
set zx 0, set nx 1, set zy 0, set ny 1, set f 0, set no 1, eval, output;

set x %B0000000000010001, set y %B0000000000000011;

// Compute 0
set zx 1, set nx 0, set zy 1, set ny 0, set f 1, set no 0, eval, output;
```

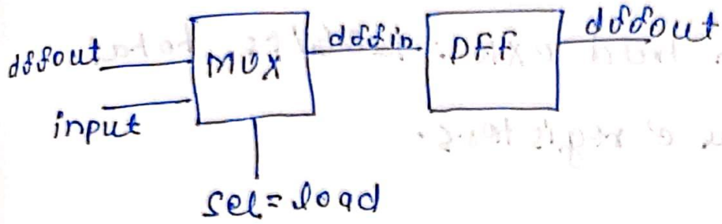
```
// Compute 1
set zx 1, set nx 1, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute -1
set zx 1, set nx 1, set zy 1, set ny 0, set f 1, set no 0, eval, output;

// Compute x
set zx 0, set nx 0, set zy 1, set ny 1, set f 0, set no 0, eval, output;

// Compute y
set zx 1, set nx 1, set zy 0, set ny 0, set f 0, set no 0, eval, output;

// Compute !x
set zx 0, set nx 0, set zy 1, set ny 1, set f 0, set no 1, eval, output;

// Compute !y
set zx 1, set nx 1, set zy 0, set ny 0, set f 0, set no 1, eval, output;

// Compute -x
set zx 0, set nx 0, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute -y
set zx 1, set nx 1, set zy 0, set ny 0, set f 1, set no 1, eval, output;

// Compute x + 1
set zx 0, set nx 1, set zy 1, set ny 1, set f 1, set no 1, eval, output;

// Compute y + 1
set zx 1, set nx 1, set zy 0, set ny 1, set f 1, set no 1, eval, output;

// Compute x - 1
set zx 0, set nx 0, set zy 1, set ny 1, set f 1, set no 0, eval, output;
```

```
// Compute y - 1
set zx 1, set nx 1, set zy 0, set ny 0, set f 1, set no 0, eval, output;

// Compute x + y
set zx 0, set nx 0, set zy 0, set ny 0, set f 1, set no 0, eval, output;

// Compute x - y
set zx 0, set nx 1, set zy 0, set ny 0, set f 1, set no 1, eval, output;

// Compute y - x
set zx 0, set nx 0, set zy 0, set ny 1, set f 1, set no 1, eval, output;

// Compute x & y
set zx 0, set nx 0, set zy 0, set ny 0, set f 0, set no 0, eval, output;

// Compute x | y
set zx 0, set nx 1, set zy 0, set ny 1, set f 0, set no 1, eval, output;
```

3 part_b-Memory

Memory → Bits & Reg.

1. Bits :-



A bit is the samller unit of data in computer
It can be either 0 or 1

2. Register :-
A Register is a small unit of memory made up of multiple bit (like many bits group work together). It store a multi-bit value - usually 8, 16, 32 bit.

Fig(3b)-Define the Hardware of Memory(Bit,Resister)

Code &Test_Bench

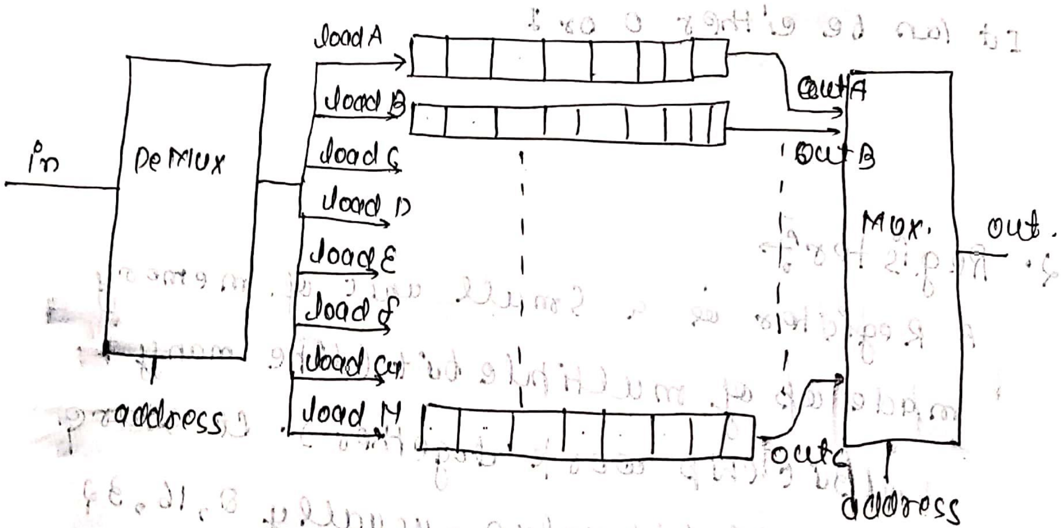
```
CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
    // Selects whether to load new input or keep old value
    Mux(a=outDFF, b=in, sel=load, out=dffInput);

    // Stores the bit using a Data Flip-Flop
    DFF(in=dffInput, out=outDFF);

    // Outputs the stored value
    Or(a=outDFF, b=false, out=out);
}
```

```
CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
    Bit(in=in[0], load=load, out=out[0]);
    Bit(in=in[1], load=load, out=out[1]);
    Bit(in=in[2], load=load, out=out[2]);
    Bit(in=in[3], load=load, out=out[3]);
    Bit(in=in[4], load=load, out=out[4]);
    Bit(in=in[5], load=load, out=out[5]);
    Bit(in=in[6], load=load, out=out[6]);
    Bit(in=in[7], load=load, out=out[7]);
    Bit(in=in[8], load=load, out=out[8]);
    Bit(in=in[9], load=load, out=out[9]);
    Bit(in=in[10], load=load, out=out[10]);
    Bit(in=in[11], load=load, out=out[11]);
    Bit(in=in[12], load=load, out=out[12]);
    Bit(in=in[13], load=load, out=out[13]);
    Bit(in=in[14], load=load, out=out[14]);
    Bit(in=in[15], load=load, out=out[15]);
}
```

*:- RAM→(Random Access Memory):-

Ram 8 is a memory chip that can store 8 words where each word is 16 bits wide

That means it can hold 8X16 = 128 bits total organized as 8 registers.



Fig(3.1)- RAM8

Code of RAM:

```
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address,
            a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);

    Register(in=in, load=load0, out=out0);
    Register(in=in, load=load1, out=out1);
    Register(in=in, load=load2, out=out2);
    Register(in=in, load=load3, out=out3);
    Register(in=in, load=load4, out=out4);
    Register(in=in, load=load5, out=out5);
    Register(in=in, load=load6, out=out6);
    Register(in=in, load=load7, out=out7);

    Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address, out=out);
}
```

```
load RAM8.hdl,
compare-to RAM8.cmp,
output-list time%S1.3.1 in%D1.6.1 load%B2.1.1 address%D3.1.3 out%D1.6.1;

// Test Case 1: Set initial values and check output
set in 0,
set load 0,
set address 0,
tick,
output;
tock,
output;

set load 1,
tick,
output;
tock,
output;

// Test Case 2: Write value 11111 to address 1
set in 11111,
set load 0,
tick,
output;
tock,
output;

set load 1,
set address 1,
tick,
output;
tock,
output;
```

```
// Test Case 3: Test with multiple address writes
set in 3333,
set address 3,
tick,
output;
tock,
output;

set load 1,
tick,
output;
tock,
output;

// Test Case 4: Test read operations
set load 0,
set address 0,
eval,
output;

set address 1,
eval,
output;

set address 3,
eval,
output;

set address 7,
eval,
output;
```

```
// Test Case 4: Test read operations
set load 0,
set address 0,
eval,
output;

set address 1,
eval,
output;

set address 3,
eval,
output;

set address 7,
eval,
output;

// Test Case 5: Set all values and verify them in sequence
set load 1,
set in %B0101010101010101,
set address 0,
tick,
output;
tock,
output;

set address 1,
tick,
output;
tock,
output;
```

```
set address 2,
tick,
output;
tock,
output;

set address 3,
tick,
output;
tock,
output;

set address 4,
tick,
output;
tock,
output;

set address 5,
tick,
output;
tock,
output;

set address 6,
tick,
output;
tock,
output;

set address 7,
tick,
output;
tock,
```

```
// Test Case 6: Test with different values for each address
set load 0,
set address 0,
tick,
output;
tock,
output;

set address 1,
set in %B1010101010101010,
tick,
output;
tock,
output;

set address 2,
set in %B0101010101010101,
tick,
output;
tock,
output;

set address 3,
set in %B1010101010101010,
tick,
output;
tock,
output;

// Test Case 7: Verify values after resetting and loading
set load 1,
set address 0,
set in %B1010101010101010,
```

```
set load 0,
set address 1,
eval,
output;

set address 2,
eval,
output;

set address 3,
eval,
output;
set address 4,
eval,
output;
```

Conclusion :

Memory System Implementation
- Designed and implemented basic memory chips including:
- Bit: The simplest storage unit
- Register: 16-bit storage that maintains its state
- RAM8 : Memory units of increasing capacity
- Gained understanding of sequential logic and clocked components
- Learned how memory hierarchy builds from simple bits to larger addressable memory units

ALU (Arithmetic Logic Unit) Implementation
- Constructed a fully functional 16-bit ALU capable of:
- Arithmetic operations (addition, subtraction)
- Logical operations (AND, OR, NOT)
- Comparison operations
- Developed understanding of how simple NAND gates can be combined to perform complex computations
- Learned about two's complement representation and arithmetic

Through this project, I've experienced firsthand how complex computer systems can be constructed from elementary logic gates. The project demonstrated the power of abstraction in computer architecture, where simple components are combined to create increasingly sophisticated functionality. This foundation prepares me well for the subsequent projects in the Nand to Tetris course where we'll build the complete computer system.

The hands-on implementation of these components has given me deeper insight into the inner workings of computer hardware that I previously took for granted.