Department of Artificial Intelligence

Project Report

ELEMENTS OF COMPUTING-1

Team Members:

Varshitha Thilak Kumar-CB.SC.U4AIE23258

Siri Sanjana S-CB.SC.U4AIE23249

Shreya Arun-CB.SC.U4AIE23253

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY IN

CSE(AI)

Supervised By:

Dr.Vipin Venugopal

Assistant Professor

Department of Artificial Intelligence

Amrita Vishwa Vidyapeetham

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM COIMBATORE - 641 112 (INDIA)

Signature of the Project Supervisor:

TABLE OF CONTENTS:

S.No	CONTENT	PAGE No.
1	ABSTRACT	3
2	LIST OF FIGURES	4
3	TABLE OF ABBREVATION	4
4	1.IMPLEMENT THE HACK COMPUTER IN HDL AND TEST IT	4
5	1.1 INTRODUCTION	4
6	1.2 METHODOLOGY	5
7	1.3 EXPERIMENTS	18
8	1.4 RESULTS	19
9	1.5 CONCLUSION	19
10	2.DESIGN AND IMPLEMENT A SYNCHRONOUS COUNTER THAT COUNT DOWN FROM DECIMAL DIGIT 9 ONWARDS TO 0	20
11	REFERENCE	25

ABSTRACT

PART-A:

This project involves implementing and testing the HACK computer architecture using HDL. The implementation will cover components such as registers, ALU, control units, memory modules, and input/output interfaces. Programs written in HACK assembly language will be used to test the computer's functionality in executing instructions, performing operations, accessing memory, and handling input/output. Through this project, a practical understanding of computer architecture and HDL programming will be gained.

PART-B:

This report outlines the design and implementation of a synchronous down counter, starting from the initial value of 9 and decrementing to 0. The project involves the application of digital circuit design principles and synchronous sequential logic to achieve a reliable and efficient counter circuit. The primary objectives include the utilization of flip-flops and combinational logic to create a synchronous counter that meets the specified requirements. . Results demonstrate a successfully implemented counter, with potential applications discussed for future development in digital circuit design.

List of Figures

Figure 1. 1 HACK COMPUTER
Figure 1. 2 HACK CPU.
Figure 1. 3 Data memory
Figure 1. 4 RAM
Figure 1. 5 Screen
Figure 1. 6 Keyboard
Figure 1. 7 Instruction Memory
Figure 1. 8 Von Neumann Architecture
Figure 1. 9 HACK Computer Implementation
Figure 1. 10 Loaded Computer.hdl in Hardware Simulator
Figure 1. 11 Test file for Computer.hdl

Table of Abbreviations

CPU - Central Processing Unit

RAM - Random Access Memory

ROM - Read Only Memory

ALU - Arithmetic Logic Unit

1. Implement the HACK computer in HDL and test it.

1.1 INTRODUCTION

The Hack computer is a simplified 16-bit computer architecture with a specific set of instructions and memory organization. Starting with basic logic gates, we have implemented the HACK computer in Hardware Description Language (HDL). The HACK computer is comprised of several essential components that work together to enable its functionality.

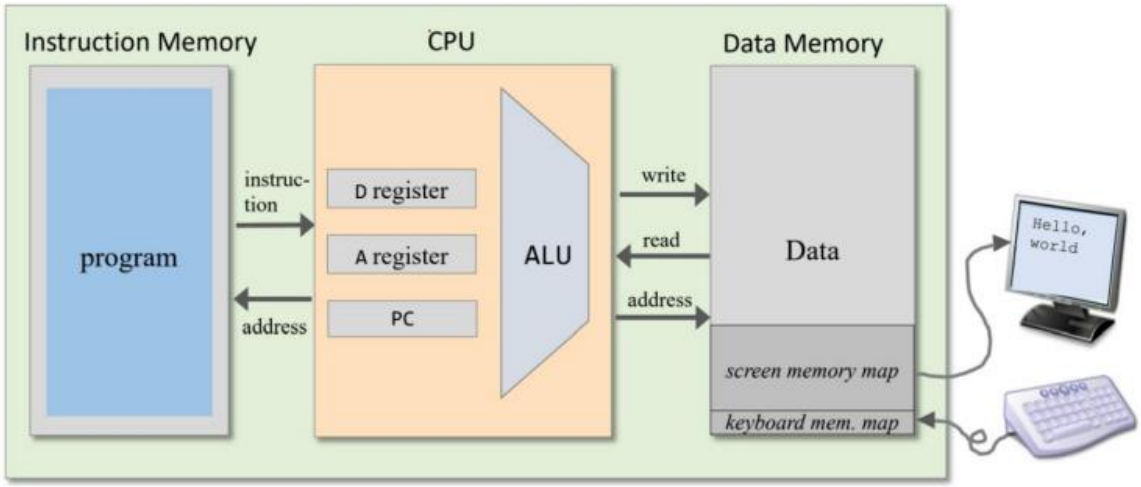


Figure 1. 1 HACK COMPUTER

Figure 1. 1 HACK COMPUTER

The three main parts of HACK Computer are:

Central Processing Unit (CPU)

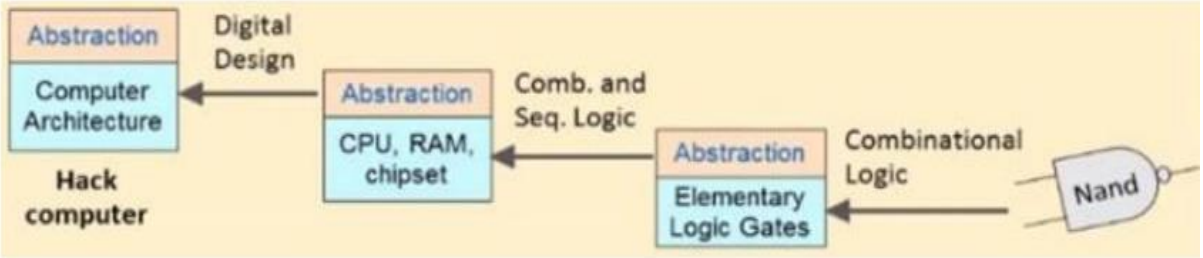
Data Memory

Instruction Memory

By integrating the above components and following the HACK architecture specifications, a complete and functional computer system which is capable of executing programs written in the HACK assembly language can be built.

1.2 METHODOLOGY

1.2.1 FLOW DIAGRAM



We have in-built NAND gate in nand2teris. We are building HACK Computer from this NAND gate. It can be divided into four steps:

1. The elementary logic gates like AND, OR, NOT, XOR, MUX, DEMUX using the NAND gates. This step can be split up into two:

i) 1-bit elementary logic gates are built.

ii)16-bit logic gates can be extended from 1-bit.

2. Then, using the elementary logic gates built in the above step, we are building Add16, Half Adder ,Full Adder, Inc16 and ALU.

3. Till the above step, combinatorial circuits are only built. Now, sequential circuits are built. For this, an inbuilt DFF is provided in nand2tetris. From the above, bit, register, RAM8, RAM64,RAM512, RAM4K, RAM16K PC are built.

4. Finally, the CPU is built followed by Memory and integration of CPU, ROM32K and Memory gives us the HACK Computer.

1.2.2 CENTRAL PROCESSING UNIT

The CPU is the centrepiece of the computer's architecture. CPU is in charge of executing the Instructions of the currently loaded program. These instructions tell the CPU to carry out various calculations, to read and write values from and into the memory, and to conditionally jump to execute other instructions Hack CPU consists of:

- ALU 16 bit
- Mux 16 bit
- Program counter/PC
- Not gate
- And gate
- Or gate
- A-register
- D-register
- Or16 gate

CPU is designed to

1. Execute the current instruction.
2. Managing control flow.
3. Perform Arithmetic and Logical calculations.
4. Input/Output Handling

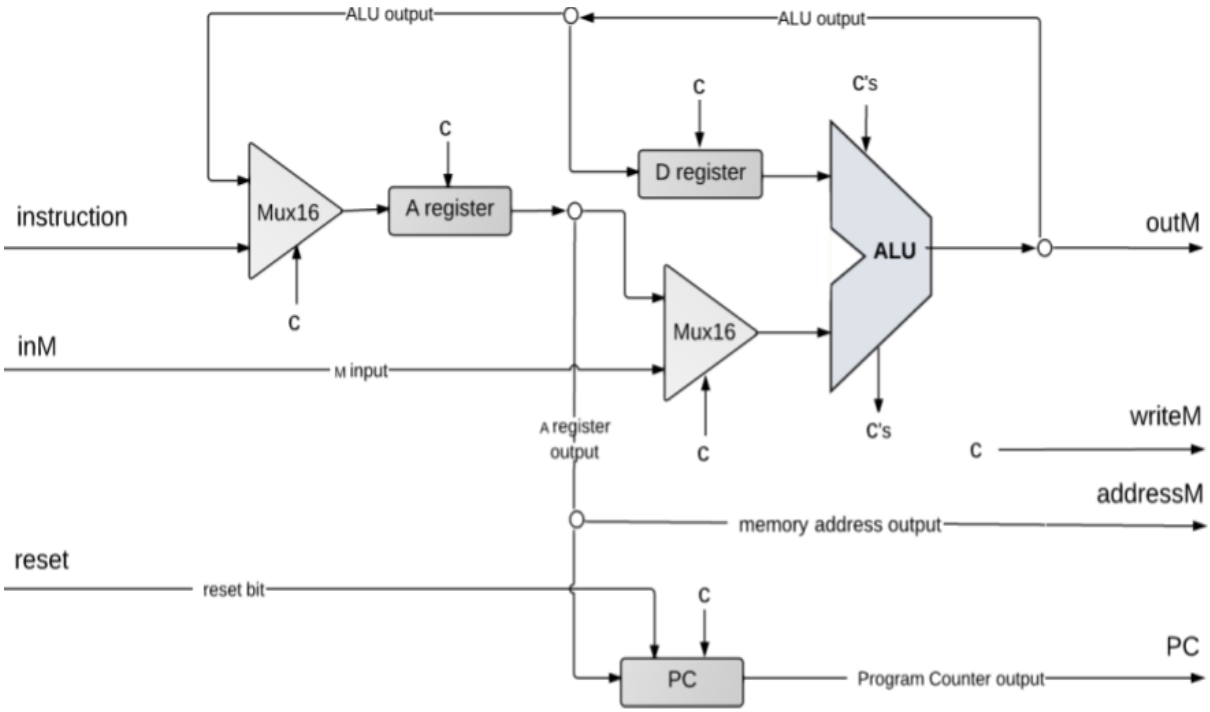


Figure 1.2 HACK CPU

HDL Code:

// This file is part of www.nand2tetris.org

// and the book "The Elements of Computing Systems"

// by Nisan and Schocken, MIT Press.

// File name: projects/05/CPU.hdl

/**

* The Central Processing unit (CPU).

* Consists of an ALU and a set of registers, designed to fetch and

* execute instructions written in the Hack machine language.

* In particular, the ALU executes the inputted instruction according

* to the Hack machine language specification, as follows.

```
* The D and A in the language specification refer to CPU-resident registers,
* while M refers to the memory register addressed by A, i.e. to Memory[A].
* The inM input holds the value of this register. If the current instruction
* needs to write a value to M, the value is placed in outM, the address
* of the target register is placed in the addressM output, and the
* writeM control bit is asserted. (When writeM=0, any value may
* appear in outM). The outM and writeM outputs are combinational:
* they are affected instantaneously by the execution of the current
* instruction. The addressM and pc outputs are clocked: although they
* are affected by the execution of the current instruction, they commit
* to their new values only in the next time unit. If reset == 1, then the
* CPU jumps to address 0 (i.e. sets pc = 0 in next time unit) rather
* than to the address resulting from executing the current instruction.
*/

CHIP CPU {
  IN  inM[16],         // M value input  (M = contents of RAM[A])
      instruction[16], // Instruction for execution
      reset;           // Signals whether to re-start the current program
                       // (reset == 1) or continue executing the current
                       // program (reset == 0).
  OUT outM[16],        // M value output
      writeM,          // Write into M?
      addressM[15],    // RAM address (of M)
      pc[15];          // ROM address (of next instruction)
```

PARTS:

```
  // get type of instruction
  Not(in=instruction[15], out=Ainstruction);
  Not(in=Ainstruction, out=Cinstruction);
  And(a=Cinstruction, b=instruction[5], out=ALUtoA);    // C-inst and dest to A-reg?
  Mux16(a=instruction, b=ALUout, sel=ALUtoA, out=Aregin);
  Or(a=Ainstruction, b=ALUtoA, out=loadA);    // load A if A-inst or C-inst&dest to A-reg
```

```
    ARegister(in=Aregin, load=loadA, out=Aout);
    Mux16(a=Aout, b=inM, sel=instruction[12], out=AMout);   // select A or M based on a-bit
    And(a=Cinstruction, b=instruction[4], out=loadD);
    DRegister(in=ALUout, load=loadD, out=Dout);    // load the D register from ALU
   ALU(x=Dout, y=AMout, zx=instruction[11], nx=instruction[10],
    zy=instruction[9], ny=instruction[8], f=instruction[7],
    no=instruction[6], out=ALUout, zr=ZRout, ng=NGout); // calculate
    // Set outputs for writing memory
    Or16(a=false, b=Aout, out[0..14]=addressM);
    Or16(a=false, b=ALUout, out=outM);
    And(a=Cinstruction, b=instruction[3], out=writeM);
    // calc PCload & PCinc - whether to load PC with A reg
    And(a=ZRout, b=instruction[1], out=jeq);    // is zero and jump if zero
    And(a=NGout, b=instruction[2], out=jlt);    // is neg and jump if neg
    Or(a=ZRout, b=NGout, out=zeroOrNeg);
    Not(in=zeroOrNeg, out=positive);           // is positive (not zero and not neg)
    And(a=positive, b=instruction[0], out=jgt); // is pos and jump if pos
    Or(a=jeq, b=jlt, out=jle);
    Or(a=jle, b=jgt, out=jumpToA);             // load PC if cond met and jump if cond
    And(a=Cinstruction, b=jumpToA, out=PCload); // Only jump if C instruction
    Not(in=PCload, out=PCinc);                 // only inc if not load
    PC(in=Aout, inc=PCinc, load=PCload, reset=reset, out[0..14]=pc);
}
```

1.2.3 DATA MEMORY

In the HACK computer architecture, the data memory is a component of the computer's memory system that is dedicated to storing variables and data used by programs during execution. It serves as a space for reading from and writing to specific memory locations.

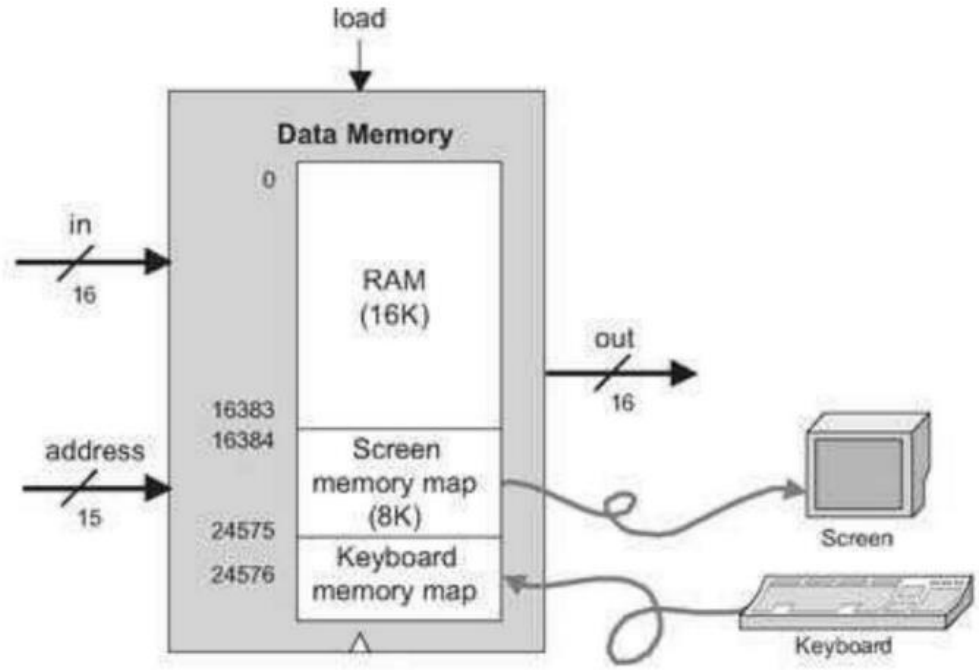


Figure 1.3 DATA MEMORY

1.2.2.1 RAM16K

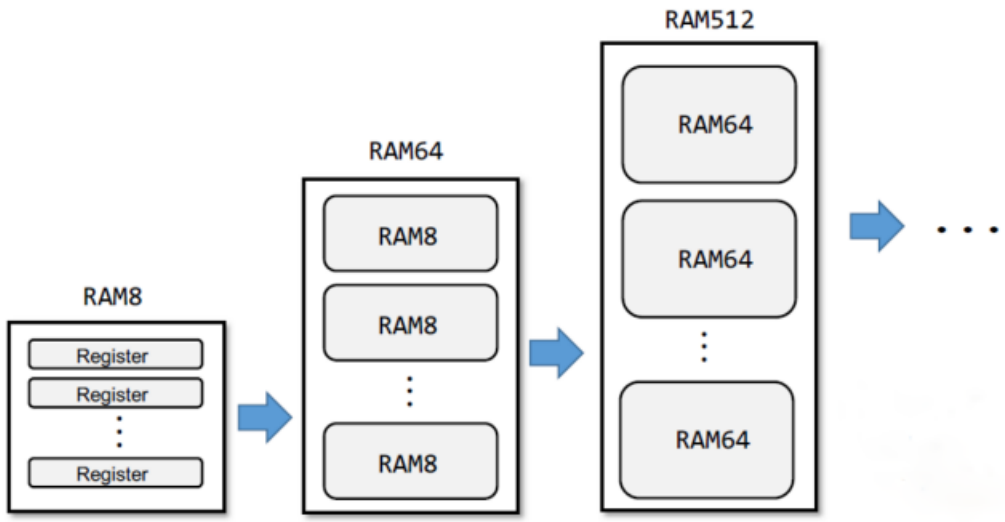


Figure 1. 4 RAM

HACK Computer uses RAM16K. RAM16K can be implemented by using Registers as shown in the above figure. Registers are built using Bit and Bits by the Data Flip Flop(DFF).

HDL Code for RAM64:

```
CHIP RAM64 {
 IN in[16], load, address[6];
 OUT out[16];
```

PARTS:

```
DMux8Way(in=load,sel=address[3..5],a=a,b=b,c=c,d=d,e=e,f=f,g=g,h=h);
RAM8(in=in,load=a,address=address[0..2],out=oa);
RAM8(in=in,load=b,address=address[0..2],out=ob);
RAM8(in=in,load=c,address=address[0..2],out=oc);
RAM8(in=in,load=d,address=address[0..2],out=od);
RAM8(in=in,load=e,address=address[0..2],out=oe);
RAM8(in=in,load=f,address=address[0..2],out=of);
RAM8(in=in,load=g,address=address[0..2],out=og);
RAM8(in=in,load=h,address=address[0..2],out=oh);
Mux8Way16(a=oa,b=ob,c=oc,d=od,e=oe,f=of,g=og,h=oh,sel=address[3..5],out=out);
}
```

Using RAM64 in a similar way as above, we built RAM512 followed by RAM4K and then finally the required RAM16K is implemented.

HDL Code for RAM16K:

```
CHIP RAM16K {
 IN in[16], load, address[14];
 OUT out[16];
```

PARTS:

```
DMux4Way(in=load,sel=address[12..13],a=a,b=b,c=c,d=d);
RAM4K(in=in,load=a,address=address[0..11],out=oa);
RAM4K(in=in,load=b,address=address[0..11],out=ob);
RAM4K(in=in,load=c,address=address[0..11],out=oc);
RAM4K(in=in,load=d,address=address[0..11],out=od);
Mux4Way16(a=oa,b=ob,c=oc,d=od,sel=address[12..13],out=out);
}
```

1.2.2.2 MEMORY MAPS

The Hack platform provides user interaction capabilities through two peripheral devices: a screen and a keyboard. These devices communicate with the computer platform using memory-mapped buffers.To

display images on the screen, users can write words to a specific memory segment known as the screen memory map. Similarly, users can read words from the screen memory map to retrieve information from the displayed image.

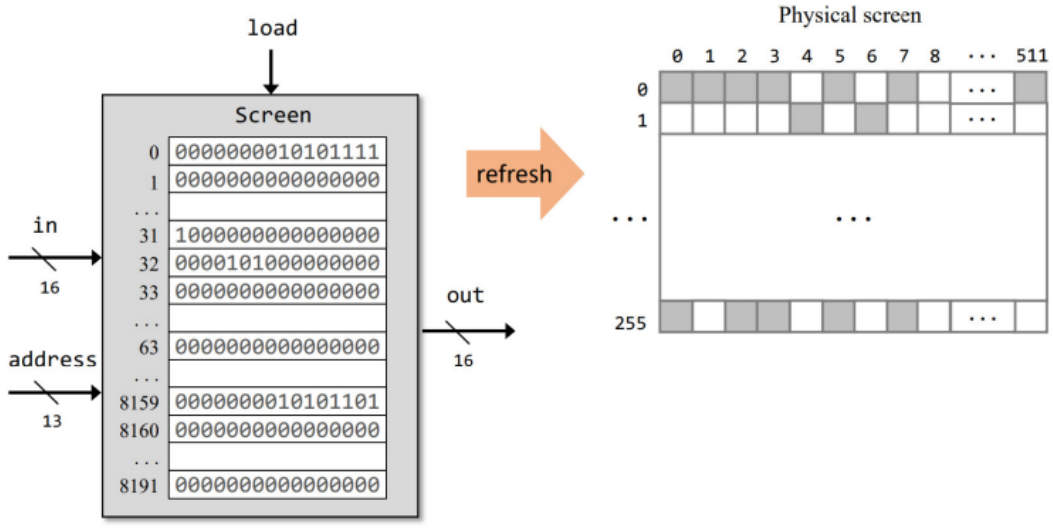


Figure 1. 5 SCREEN

The keyboard's interaction involves checking which key is currently pressed by accessing a designated memory word called the keyboard memory map.

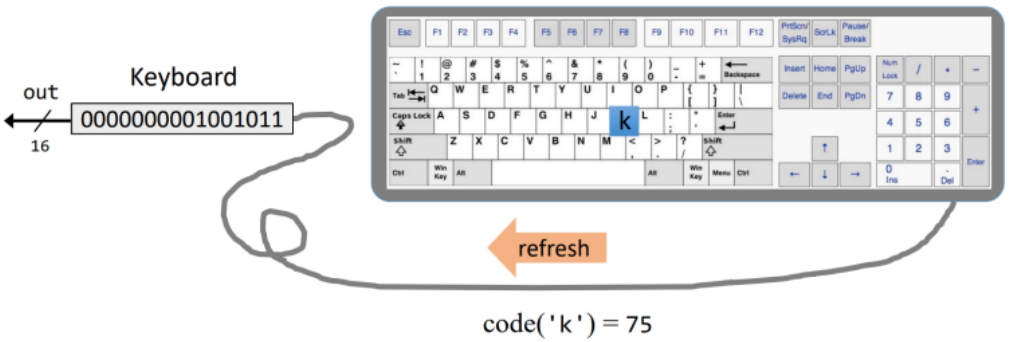


Figure 1.6. KEYBOARD

The peripheral logic, located outside the computer, facilitates the communication between the memory maps and the corresponding I/O devices. Whenever a bit is modified in the screen memory map, the corresponding pixel is drawn on the physical screen. Likewise, when a key is pressed on the physical keyboard, the corresponding key code appears in the keyboard memory map.

1.2.2.3 IMPLEMENTATION

Memory architecture

• An aggregate of three chip-parts: RAM16K, Screen, Keyboard.

• Single address space, 0 to 24576

• Maps the address input onto the corresponding address input of the relevant chip-part.

Screen and Keyboard are in-built chips and RAM16K is implemented in HDL.

HDL Code:

```
CHIP Memory {
 IN in[16], load, address[15];
 OUT out[16];
 PARTS:
DMux4Way(in=load,sel=address[13..14],a=firstRAM,b=secondRAM,c=screenLoad,d=kbdLoad);
Or (a=firstRAM, b=secondRAM, out=ramLoad);
 RAM16K (in=in, load=ramLoad, address=address[0..13], out=ramOut);
 Screen (in=in, load=screenLoad, address=address[0..12], out=screenOut);
 Keyboard (out=kbdOut);
 Mux4Way16 (a=ramOut, b=ramOut, c=screenOut, d=kbdOut, sel=address[13..14],
 out=out);
}
```

Code Explanation:

• DMux4Way chip is used to demultiplex the load input signal based on the two most significant bits (address[13] and address[14]) of the address input. It generates four output signals: a (for the first RAM chip), b (for the second RAM chip), c (for the screen), and d (for the keyboard).

• Or gate to compute the logical OR of the firstRAM and secondRAM signals. The resulting output ramLoad is used as the load signal for the RAM16K chip.

• RAM16K takes the data to be written as input, the ramLoad signal as the write enable signal, and the lower 14 bits of the address input as the memory address. The output of the RAM16K chip is stored in the ramOut variable.

• Then the Screen chip is instantiated, which represents the screen or display module. It takes the data to be written to the screen as the , the screenLoad signal as the write enable signal, and the lower 13 bits of the address input as the screen memory address. The output of the Screen chip is stored in the screenOut variable.

• Then output of the Keyboard chip is stored in the kbdOut variable. It provides keyboard-related functionality such as detecting key presses and generating corresponding key codes.

• Mux4Way16 chip to multiplex the output data based on the two most significant bits of the address input. It selects the output data from either the RAM (ramOut), screen (screenOut), or keyboard (kbdOut) based on the values of address[13] and address[14] and assigns it to the out output bus.

```
CHIP Keyboard {
 OUT out[16]; // The ASCII code of the pressed key,
 // or 0 if no key is currently pressed,
 // or one the special codes listed in Figure 5.5.
BUILTIN Keyboard;
}
CHIP Screen {
 IN in[16], // what to write
 load, // write-enable bit
 address[13]; // where to read/write
 OUT out[16]; // Screen value at the given address
 BUILTIN Screen;
 CLOCKED in, load;
}
```

1.2.3 INSTRUCTION MEMORY

The Instruction Memory holds the machine code instructions that the CPU fetches and executes.

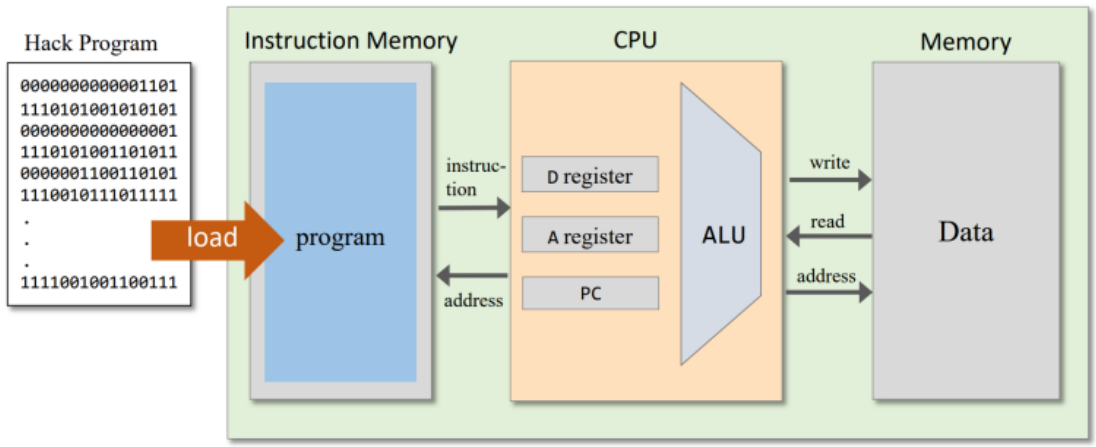


Figure 1.7. INSTRUCTIONS MEMORY

The instruction memory is implemented as a built-in chip named ROM32K.

To load a program to ROM32K, use the load program service and replace the ROM32K chip.

HDL CODE:

```
CHIP ROM32K {
 IN address[15];
 OUT out[16];
 BUILTIN ROM32K;
}
```

1.2.4 HACK COMPUTER

The Hack computer is a 16-bit von Neumann machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory, and two memory-mapped I/O devices: a screen and a keyboard.

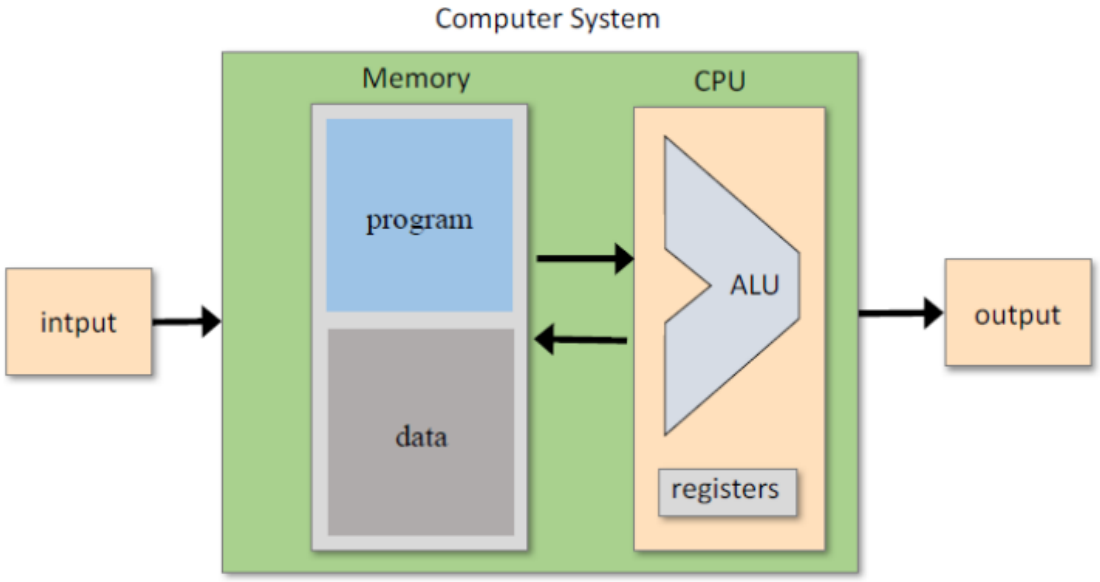


Figure 1.8 VON NUWMANN ARCHITECTURE

The computer is loaded with a program written in the Hack machine language. To execute the stored program:

if (reset == 1), executes the first instruction in the stored program

if (reset == 0), executes the next instruction in the stored program

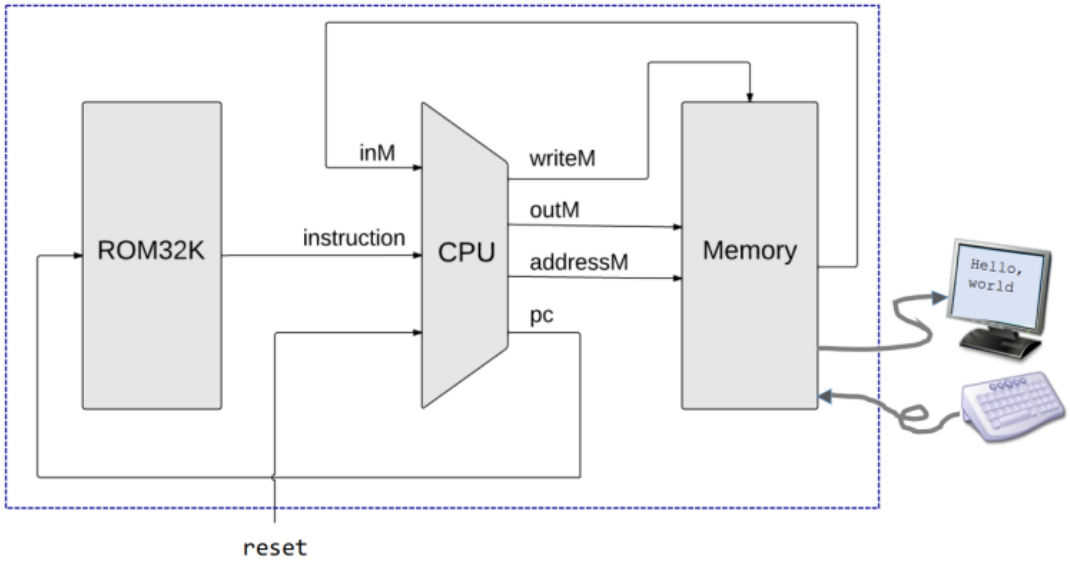


Figure 1.9. HACK COMPUTER IMPLEMENTATION

On integrating the components we implemented above using the above figure, we get a fully functional computer.

HDL CODE:

CHIP Computer {

IN reset;

PARTS:

Memory(in=outMemory ,load=writeM ,address=addressM ,out=inMemory);

ROM32K (address=pc, out=instruction);

CPU (reset=reset, instruction=instruction, inM=inMemory, addressM=addressM,

writeM=writeM, outM=outMemory, pc=pc);

}

1.2.5 TESTING

Now that we have built a computer, we should check whether it works properly. To test our computer, following steps are used:

• First, load Computer.hdl into the hardware simulator.

• Then, a Hack program is loaded into the ROM32K chip-part.

• Run the clock enough cycles to execute the program.

TEST CODE:

load Computer.hdl,

output-file ComputerAvg.out,

compare-to ComputerAvg.cmp,

output-list time%S1.4.1 reset%B2.1.2 ARegister[]%D1.7.1 DRegister[]%D1.7.1 PC[]%D0.4.0 RAM16K[1]%D1.7.1 RAM16K[2]%D1.7.1;

ROM32K load average.hack,

set RAM16K[1] 5,

output;

repeat 150 {

tick, tock, output;

}

set reset 1,

set RAM16K[0] 0,

set RAM16K[2] 0,

tick, tock, output;

After this, test script is loaded in the Hardware Simulator.

1.3 EXPERIMENTS

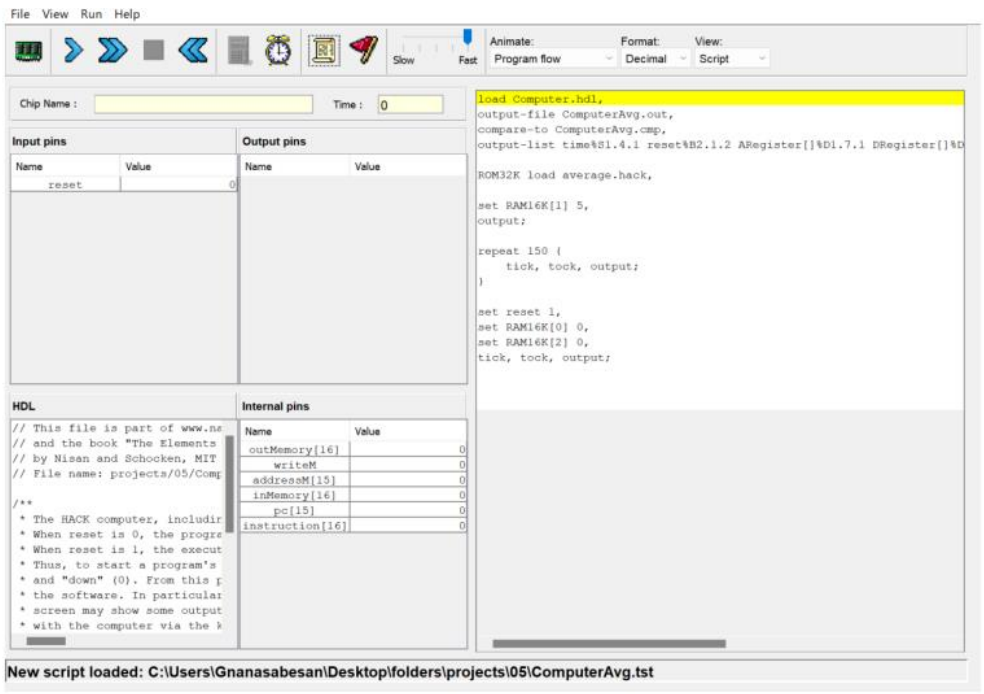
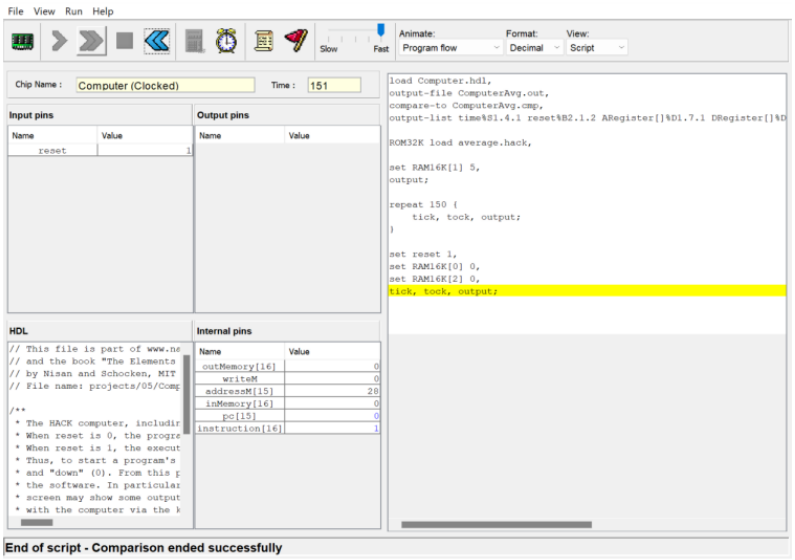


Figure 1.10. LOADED COMPUTER HDL IN HARDWARE SIMULATOR

The Computer.hdl is loaded in the hardware simulator and the ComputerRect.tst test script is loaded

1.4. RESULTS



The comparison ended successfully. Now, we have a fully functional computer which is capable of executing any hack assembly code.

1.5 CONCLUSION

We successfully implemented HACK Computer in nand2tetris. This project serves as a foundation for further exploration and understanding of complex computer systems and their underlying principles. Throughout the project, we have successfully designed and interconnected various components, including logic gates, arithmetic units, memory modules, and control units, all using Hardware Description Language (HDL)

2. SYNCHRONOUS COUNTER THAT COUNT DOWN FROM DECIMAL DIGIT 9 ONWARDS TO 0:

2.1.synchronous counter:

A synchronous counter is a type of digital counter circuit in which all flip-flops or stages change state simultaneously in response to a clock signal. In contrast to asynchronous counters, where the flip-flops change state independently based on their individual clock inputs, synchronous counters use a common clock signal to coordinate the timing of state changes across all stages.

The primary advantage of synchronous counters is their ability to eliminate the issues of propagation delay and race conditions that can occur in asynchronous counters. Propagation delay refers to the time it takes for a signal to propagate through a circuit, and race conditions can occur when different flip-flops within the counter change state at different times, leading to unpredictable behavior. In a synchronous counter, the clock signal ensures that all flip-flops change state together, providing a more predictable and reliable operation. The design of synchronous counters typically involves the use of flip-flops and combinational logic circuits to generate the necessary control signals for the counter. Synchronous counters are widely used in digital systems for applications such as frequency dividers, event counters, and various other tasks where accurate and synchronized counting is required. The most common types of synchronous counters are binary counters, where each flip-flop represents a binary digit in the counting sequence.

2.2.Downcounter:

A down counter is a type of digital counter circuit that counts down from a predetermined value to zero. Counters, in general, are electronic circuits used to count events or occurrences. The direction in which they count—either up or down—depends on their design and application.In a down counter, the count decreases by one for each clock pulse until it reaches zero. When the counter reaches zero, it may stop counting or possibly trigger a specific action, depending on its design and purpose. Down counters are commonly implemented using flip-flops and combinational logic

circuits. The flip-flops store the current count, and the combinational logic is responsible for generating the necessary control signals to decrement the count on each clock pulse. down counters depending on the requirements of the application.

To begin with the code:

We already have D-flipflop(DFF) inbuilt in the nand2teris. Therefore using DFF we are making Toggle flipflop

Q4	Q3	Q2	Q1	Q4*	Q3*	Q2*	Q1*	T4	T3	T2	T1
0	0	0	0	1	0	0	1	1	0	0	1
0	0	0	1	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	1	0	0	1	1
0	0	1	1	0	0	1	0	0	0	0	1
0	1	0	0	0	1	1	1	0	1	1	1
0	1	0	1	0	1	0	0	0	0	0	1
0	1	1	0	0	1	0	1	0	0	1	1
0	1	1	1	0	1	1	0	0	0	0	1
1	0	0	0	1	0	1	1	1	1	1	1
1	0	0	1	0	0	0	0	0	0	0	1

TRUTH TABLE

Karnaugh Map:

For T4:

Q3Q4 / Q1Q2	00	01	11	10
00	**1**			
01				
11	**x**	**x**	**x**	**x**
10	**1**		**x**	**x**

= Q1'Q2'Q3'

For T3:

Q3Q4 / Q1Q2	00	01	11	10
00				
01	1			
11	x	x	x	x
10	1		x	x

=Q3Q2'Q1'+Q4Q2'Q1'

For T2:

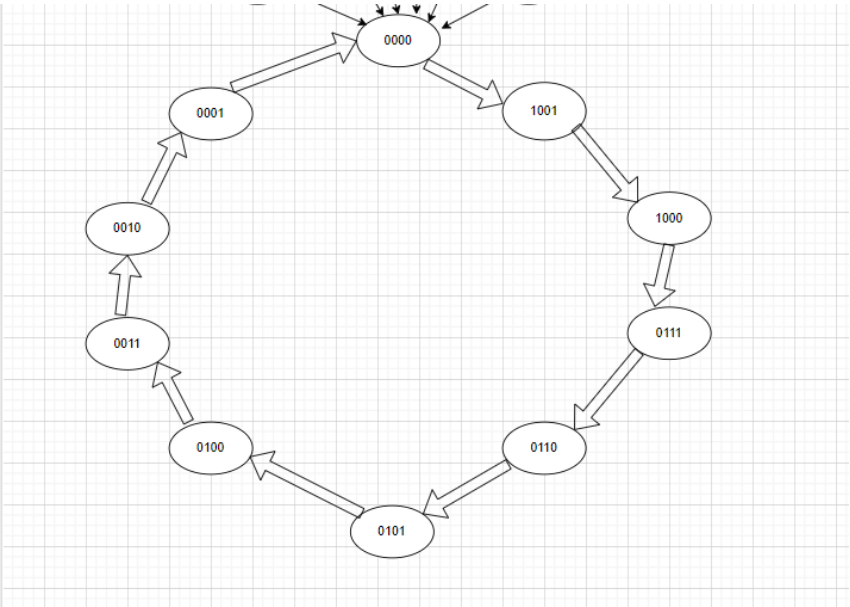
Q3Q4 / Q1Q2	00	01	11	10
00				1
01	1			1
11	x	x	x	x
10	1		x	x

=Q2Q1'+Q3Q2'Q1'+Q4Q2'Q1'

For T1:

Q3Q4 / Q1Q2	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	X	X	X	X
10	1	X	X	X

=1



STATE DIAGRAM

Code of inbuilt DFF:

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: tools/builtIn/DFF.hdl
/**
* Data Flip-flop: out(t) = in(t-1)
* where t is the current time unit, or clock cycle.
*/
CHIP DFF {
    IN  in;
    OUT out;
    BUILTIN DFF;
    CLOCKED in;
}
```

Code for TFF:

```
CHIP TFF {
  IN T;
```

```
 OUT out;
 PARTS:
 Xor(a = T, b = dffout, out = xout);
 DFF(in = xout, out = dffout, out=out);
}
```

Code for and 3 input pin:

```
CHIP And3in{
IN a,b,c;
OUT  out;
PARTS:
And(a=a,b=b,out=o1);
And(a=o1,b=c,out=out);
}
```

Code for or 3 input pin:

```
CHIP Or3in{
IN a,b,c;
OUT out;
PARTS:
//Put your code here:
Or(a=a,b=b,out=o1);
Or(a=o1,b=c,out=out);
}
```

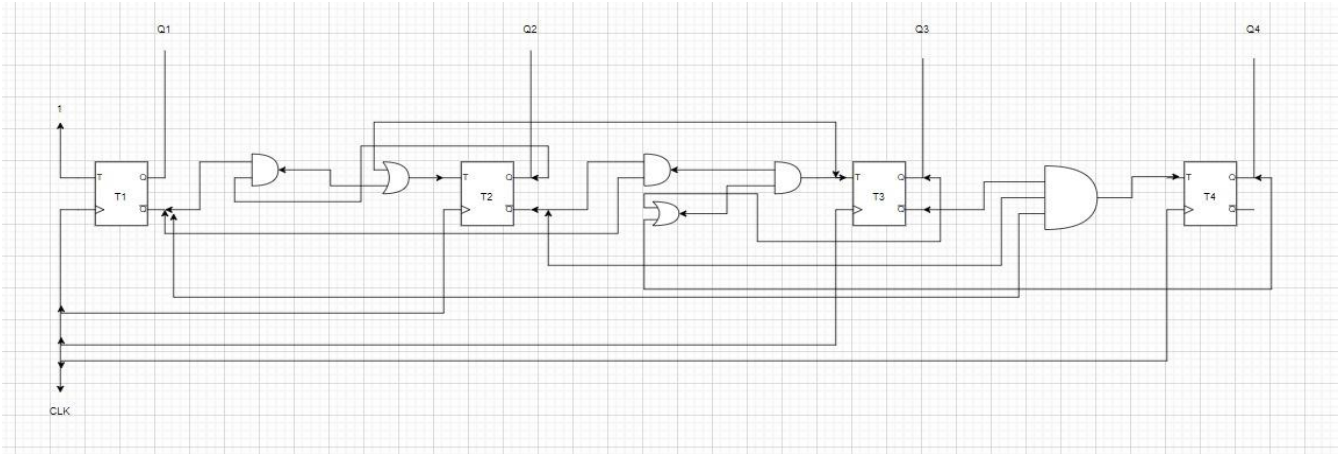
Code for downcounter:

```
CHIP Mod10Counter1
{
   OUT q[4];

   PARTS:
```

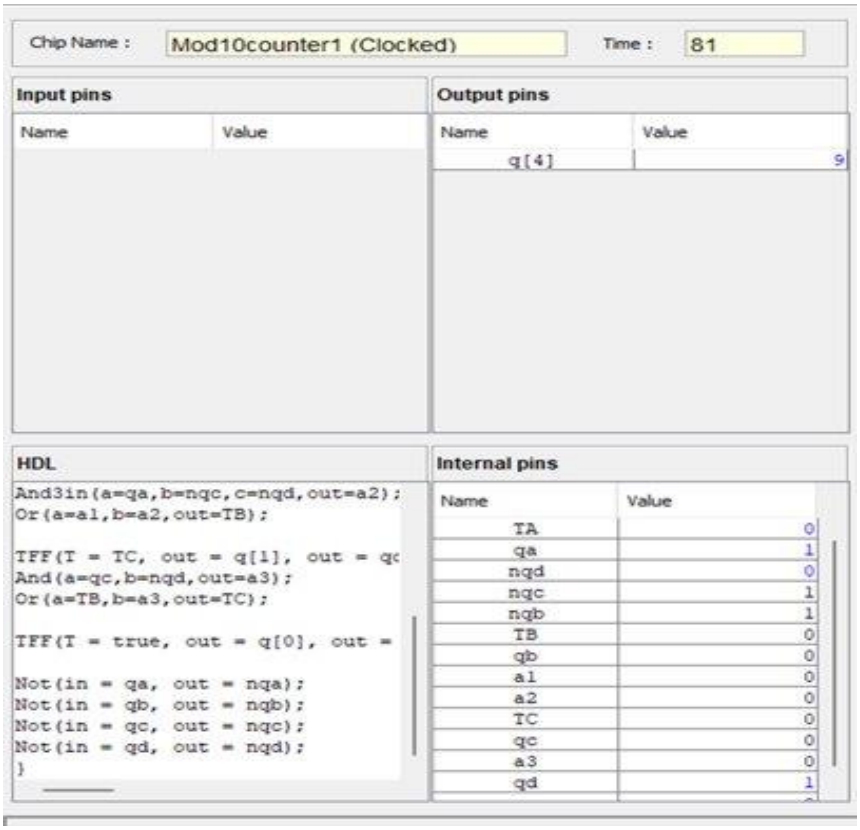
```
        TFF(T = TA, out = q[3], out = qa);
        And3in(a = nqd, b = nqc, c = nqb, out = TA);

        TFF(T = TB, out = q[2], out = qb);
        And3in(a = qb, b = nqc, c = nqd, out = a1);
        And3in(a = qa, b = nqc, c = nqd, out = a2);
        Or(a = a1, b = a2, out = TB);

        TFF(T = TC, out = q[1], out = qc);
        And(a = qc, b = nqd, out = a3);
        Or(a = TB, b = a3);
        TFF(T = true, out = q[0], out = qd);
        Not(in = qa, out = nqa);
        Not(in = qb, out = nqb);
        Not(in = qc, out = nqc);
        Not(in = qd, out = nqd);
}
```

Output:

BLOCK DIAGRAM



Output : NAND2TETRIS



Conclusion:

In summary, our investigation has led to the development and examination of a 4-bit synchronous down counter engineered to count in a decrementing sequence from 9 to 0. Employing four flip-flops to represent individual digits, the synchronous design guarantees simultaneous state changes across all flip-flops with each clock pulse. The counter initiates at 1001(9) and successively transitions through binary states, concluding at 0000(0).The precision and synchronized counting exhibited by the synchronous down counter render it apt for applications demanding accurate countdowns. Rooted in the principles of digital logic design, this counter furnishes a dependable and predictable method for achieving a decrementing sequence, starting from an initial value of 9 and concluding at 0.This counter design holds practical utility across various scenarios, particularly in applications such as timers, where a precise countdown is imperative to trigger specific events or actions in digital systems. The synchronous down counter's adeptness at maintaining synchronization and mitigating race conditions underscores its reliability in real-world implementations. Its application serves as a testament to the principles of digital design and synchronous operation, contributing to the advancement of digital systems.

3.REFERENCES:

[1].https://drive.google.com/file/d/1nEptWuRpFF9zmqlKYq6s1UfDB_dd16vx/viEw

[2]. EoC-1_Lecture_13__Sequential Logic-Part 1.pptx

[3]. The Elements of Computing Systems by Noam Nisan and Shimon Schocken

[4]. EoC-1_lecture_16__Machine Language - Part 1.pptx

[5].https://sean.eulenberg.de/posts/2019-05-14-what-nand-2-tetris-has-taught-me-about-computers and-more-importantly-about-learning/images/03_abstraction_levels_slide.png

[6].https://www.computerscience.gcse.guru/wp-content/uploads/2016/04/Von-NeumannArchitecture-Diagram.jpg

[7]. https://zhangruochi.com/Machine-Language/2019/09/22/15.png

[8]. www.javatpoint.com

[9]. www.geeksforgeeks.org

[10]. https://www.knowledgehut.com/

[11]. https://www.programiz.com/

[12]. https://www.freecodecamp.org