HACK computer

Table of Contents:

1.1INTRODUCTION ---------- 1

1.2 METHODOLOGY ---------- 2

1.3 CPU ---------- 2

1.4 RAM ---------- 2

1.5 ROM ---------- 3

1.5 Memory segments ---------- 4

1.6 CPU ---------- 5

1.7 Registors ---------- 5

1.8 COMPUTER ---------- 6

1.9 Hdl codes ---------- 6

1.10 Outputs ---------- 22

1.11 CONCLUSION ---------- 25

LIST OF FIGURES:

Figure 1.1:Output 22

Figure 1.2: Output with tst file 23

Figure 1.3:Testing 24

Figure 1.4:Testing with tst file 24

1.1 INTRODUCTION:

The Nand2tetris project led to the development of the HACK computer, a design that enables students to comprehend computer science and computer architecture from the ground up. The von Neumann architecture-based HACK computer is a simplified yet complete system created using a Hardware Description Language (HDL).

At its core, the HACK computer is a 16-bit system with a single memory unit that holds both instructions and data. The memory is implemented using a RAM module, providing 32,768 memory locations. The CPU, which consists of an ALU (Arithmetic Logic Unit) and a control unit, fetches and executes instructions stored in memory. The ALU is responsible for performing arithmetic and logical operations on 16-bit data, while the control unit manages the instruction execution process.

Input and output modules are combined in order to communicate with the HACK computer. The screen module allows the computer to display text and graphics while the keyboard module allows user input. These modules improve the computer's capabilities and give users a way to interact with it.

The HACK computer is programmed in assembly language. The assembly language is made up of mnemonic commands that correspond exactly to the binary machine code commands carried out by the computer. These assembly language programmes are converted into binary machine code by the HACK assembler, allowing the computer to run them.

The HACK computer is a thorough educational tool that offers a practical approach to comprehending computer architecture. Learners gain a thorough understanding of how a computer system functions at the hardware level by starting with simple logic gates and moving on to build the ALU, CPU, and memory. They also learn about input and output methods as well as how computers run assembly language programmes.

Overall, the HACK computer provides a chance to investigate key ideas in computer architecture.

1.2 METHODOLOGY:

There are three fundamental components that make up the Hack computer hardware. A central

processing unit (CPU) and two separate 16-bit memory units are present. The Hack computer is categorised as a 16-bit architecture because it moves and processes data in 16-bit words. The assembled binary programme code for execution is stored in the instruction memory, which the computer implements as read-only memory and refers to as ROM. The computer's memory-mapped I/O mechanism uses the random access memory, also known as RAM, as a storage medium for data from programmes that are currently running. The CPU manages programme control and performs data processing.

1.3 CPU:

The Central Processing Unit (CPU), which executes commands from the programme and manages every other component, serves as the brain of a computer. The CPU executes programme instructions, reads information from memory, performs arithmetic operations, and then sends the appropriate signals to the other parts. An instruction register, an arithmetic logic unit, and a memory address register are examples of parts found in a CPU. The instruction that the CPU is currently processing is stored in the Instruction Register. The Arithmetic Logic Unit manipulates data using logic and arithmetic. The address of the data that the CPU needs to access is stored in the memory address register.

1.4 RAM:

An essential part of all computer architectures, including the Hack architecture, is random access memory (RAM). It acts as the computer's primary memory, providing momentary storage for information and commands actively used during computations. RAM is volatile, which means that when the power is turned off, its contents are lost, unlike Read-Only Memory (ROM). The addressable cells in the RAM of the Hack computer enable the processor to read from and write to particular places. Data can be quickly and directly retrieved thanks to its random access nature, regardless of where it is physically located in the memory. RAM enables quick read and write operations, which improves system performance and programme execution. The architecture of the Hack computer has a 16-bit address space that can address up to 65,536 memory locations, each of which can store 16 bits of data. RAM serves as the main workspace, making it easier to run programmes and work with data. The Hack specification calls for a single, fixed-size RAM module, but actual computer systems have different RAM capacities, allowing for more memory

and better performance.

1.5 ROM:

The main memory (RAM) and read-only memory (ROM) of the Hack computer are housed on separate chips. It has a fixed set of pre-programmed instructions that are immutable and cannot be altered while the programme is running. These instructions are in charge of carrying out operations like initialising the computer's hardware, configuring the memory, and loading the first programme into memory.

Typically, combinational logic circuits or specialised memory chips are used to implement the ROM.

Implementation of computer

How does a computer work?

1. The first thing the computer does is fetch the subsequent instruction from memory. The memory address of the instruction that needs to be fetched is stored in the programme counter (PC). The instruction register (IR) is where the instruction is kept.

2. Decoding of Instructions: The control unit decodes an instruction from an IR signal to identify the operation to be carried out and the operands involved.

3. Data Fetch: The instruction or a register is used to determine the memory address of the data if the instruction calls for data to be fetched from memory. The memory address register (MAR) receives the address.

4. Data Transfer: The memory data register (MDR) is loaded with the data that was retrieved from memory. Data is transferred to the appropriate register if the instruction calls for accessing a register or memory location.

5. Execution: Based on the instruction, the arithmetic logic unit (ALU) performs the necessary arithmetic or logical operation on the data in the registers..

6. Memory Access: The ALU result or the contents of a register are stored in the memory location specified by the instruction if the instruction involves writing data back to memory.

7. Control Flow: The control unit updates the programme counter (PC) to point to the following instruction in memory based on the instruction. Additionally, it manages jumps and branches, modifying the PC to change programme execution as necessary.

8. Repeat: The process is repeated, fetching the next instruction from memory, and executing it. This continues until the program completes or a termination condition is met. Control Flow: Based on the instruction, the control unit modifies the programme counter (PC) to point to the next instruction in memory. Furthermore, it controls jumps and branches, modifying the PC to alter programme execution as required.

The construction of an ALU typically involves combining several logic gates and multiplexers to implement the desired functionality. The ALU takes input data from registers, performs the specified operation based on control signals, and produces the output result.

To assemble the ALU, one must carefully design the circuitry to handle various arithmetic and logical operations efficiently and accurately. This involves selecting appropriate logic gates and determining the necessary wiring connections to enable the desired functionality.
The ALU design may vary based on the specific computer architecture and instruction set being implemented. Common operations that an ALU can perform include addition, subtraction, logical AND, logical OR, bitwise shifting, and comparison operations.

Assembling the ALU is a critical step in building a computer system as it provides the computational capability required for executing instructions and performing calculations. It serves as the core component within the CPU (Central Processing Unit) and plays a vital role in the overall functionality and performance of the computer system.

1.5 Memory segments:

Memory modules are essential components that provide the computer with the ability to store and access both instructions and data. They are responsible for holding information temporarily or permanently, depending on the type of memory used.

The construction of memory modules involves selecting and implementing the appropriate memory technology, such as Random Access Memory (RAM) or Read-Only Memory (ROM). RAM is used for temporary data storage, while ROM is used for storing permanent instructions or data that should not be modified.

The design and construction of memory modules require a careful consideration of factors such as capacity, speed, and reliability. The modules are organized into addressable locations or cells, each capable of storing a fixed amount of data. The arrangement and addressing scheme determine how data can be accessed and retrieved.

The construction process may involve integrating memory chips, addressing circuitry, and control logic to enable data storage and retrieval operations. Depending on the computer architecture, memory modules may include different levels of cache memory, main memory, and secondary storage devices like hard drives or solid-state drives.

1.6 CPU:

The Control Unit receives instructions from memory, decodes them to determine the operation to be performed, and generates control signals that coordinate the execution of the instruction. It

manages the sequencing of instructions, determining the order in which they are executed, and controls the flow of data between registers, memory, and the ALU.

The Control Unit typically includes components such as instruction registers, program counters, and control logic. The instruction register holds the current instruction being executed, while the program counter keeps track of the address of the next instruction to be fetched. The control logic generates the necessary control signals based on the decoded instruction, directing the flow of data and operations within the CPU.

By assembling the Control Unit, the computer system gains the ability to fetch, decode, and execute instructions in a controlled and orderly manner. It ensures that each instruction is executed correctly, coordinating the flow of data and controlling the operations performed by the ALU, memory, and other components. The Control Unit plays a crucial role in managing the instruction execution process, facilitating the proper functioning of the computer system.

The control unit manages the execution of instructions by decoding and coordinating the flow of data within the CPU and between other components. It interprets the instructions fetched from memory and generates control signals to regulate the flow of data and operations within the CPU. This involves controlling the ALU, accessing memory, and managing input/output operations.

The ALU, as mentioned earlier, performs arithmetic and logical operations on binary data. It carries out calculations, comparisons, and logical operations based on the instructions received from the control unit. The ALU's circuitry is designed to accommodate various operations, such as addition, subtraction, logical AND, logical OR, and shifting.

By creating the CPU, which consists of the control unit and the ALU, the computer system gains the ability to execute instructions, perform calculations, and control data flow. The CPU's design and construction involve careful consideration of instruction set architecture, microarchitecture, and the coordination between various components to ensure efficient and accurate execution of programs.

1.7 Registors:

Registers play a crucial role in data manipulation, as they provide quick access to operands and intermediate results during arithmetic, logical, and data transfer operations. They are typically part of the CPU and are directly accessible by the control unit and the ALU.

The number and types of registers vary depending on the computer architecture. Common types of registers include the program counter (PC), which holds the address of the next instruction to be executed, and general-purpose registers (GPRs), used for holding data during computations. Special-purpose registers, such as the accumulator, index registers, and stack pointer, serve specific purposes based on the computer's design.

1.8 COMPUTER:

The topmost chip in the Hack hardware hierarchy is a complete computer system designed to execute programs written in the Hack machine language. This abstraction is described in . The Computer chip contains all the hardware devices necessary to operate the computer including a CPU, a data memory, an instruction memory (ROM), a screen, and a keyboard, all implemented as internal parts. In order to execute a program, the program's code must be preloaded into the ROM. Control of the screen and the keyboard is achieved via their memory maps, as described in the Screen and Keyboard chip specifications.

1.9 Hdl codes:

1.Cpu code:

```
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 to M?
       addressM[15],    // Address in data memory (of M)
       pc[15];          // address of next instruction

   PARTS:
   // Put your code here:
        Not(in=instruction[15],out=ni);
         Mux16(a=outtM,b=instruction,sel=ni,out=i);

         Or(a=ni,b=instruction[5],out=intoA);
         ARegister(in=i,load=intoA,out=A,out[0..14]=addressM);

         And(a=instruction[15],b=instruction[12],out=AorM);
         Mux16(a=A,b=inM,sel=AorM,out=AM);

         ALU(x=D,y=AM,zx=instruction[11],nx=instruction[10],zy=instruction[9],ny
=instruction[8],f=instruction[7],no=instruction[6],out=outtM,out=outM,zr=zr,ng=n
g);

         And(a=instruction[15],b=instruction[4],out=intoD);
         DRegister(in=outtM,load=intoD,out=D);

         And(a=instruction[15],b=instruction[3],out=writeM);

         Not(in=ng,out=pos);
         Not(in=zr,out=nzr);
         And(a=instruction[15],b=instruction[0],out=jgt);
```

```
        And(a=pos,b=nzr,out=posnzr);
        And(a=jgt,b=posnzr,out=ld1);

        And(a=instruction[15],b=instruction[1],out=jeq);
        And(a=jeq,b=zr,out=ld2);

        And(a=instruction[15],b=instruction[2],out=jlt);
        And(a=jlt,b=ng,out=ld3);

        Or(a=ld1,b=ld2,out=ldt);
        Or(a=ld3,b=ldt,out=ld);

        PC(in=A,load=ld,inc=true,reset=reset,out[0..14]=pc);
}
```

2.Memory :

```
CHIP Memory {
    IN in[16], load, address[15];
    OUT out[16];

    PARTS:
    // Put your code here:
     DMux4Way(in=load, sel=address[13..14], a=loadram1, b=loadram2, c=loadscreen,
d=loadkbd);
    Or(a=loadram1, b=loadram2, out=loadram);
    RAM16K(in=in, load=loadram, address=address[0..13], out=ramout);
    Screen(in=in, load=loadscreen, address=address[0..12], out=scrout);
    Keyboard(out=kbout);
    Mux4Way16(a=ramout, b=ramout, c=scrout, d=kbout, sel=address[13..14], out=out);
}
```

3.Register

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

    PARTS:
    // Put your code here:
   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]);

}
```

4.Ram512

```
CHIP RAM512 {
   IN in[16], load, address[9];
   OUT out[16];

   PARTS:
   // Put your code here:
DMux8Way(in=load,sel=address[6..8],a=a,b=b,c=c,d=d,e=e,f=f,g=g,h=h);
RAM64(in=in,load=a,address=address[0..5],out=o1);
RAM64(in=in,load=b,address=address[0..5],out=o2);
RAM64(in=in,load=c,address=address[0..5],out=o3);
RAM64(in=in,load=d,address=address[0..5],out=o4);
RAM64(in=in,load=e,address=address[0..5],out=o5);
RAM64(in=in,load=f,address=address[0..5],out=o6);
RAM64(in=in,load=g,address=address[0..5],out=o7);
RAM64(in=in,load=h,address=address[0..5],out=o8);
Mux8Way16(a=o1,b=o2,c=o3,d=o4,e=o5,f=o6,g=o7,h=o8,sel=address[6..8],out=out);
}
```

5.Ram 64

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

   PARTS:
   // Put your code here:
        DMux8Way(in=load, sel=address[3..5], a=loada, b=loadb, c=loadc, d=loadd, e=loade,
f=loadf, g=loadg, h=loadh);
        RAM8(in=in, load=loada, address=address[0..2], out=outa);
        RAM8(in=in, load=loadb, address=address[0..2], out=outb);
        RAM8(in=in, load=loadc, address=address[0..2], out=outc);
        RAM8(in=in, load=loadd, address=address[0..2], out=outd);
        RAM8(in=in, load=loade, address=address[0..2], out=oute);
        RAM8(in=in, load=loadf, address=address[0..2], out=outf);
        RAM8(in=in, load=loadg, address=address[0..2], out=outg);
        RAM8(in=in, load=loadh, address=address[0..2], out=outh);
        Mux8Way16(a=outa, b=outb, c=outc, d=outd, e=oute, f=outf, g=outg, h=outh,
sel=address[3..5], out=out);
}
```

6.Ram16k

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

    PARTS:
    // Put your code here:
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=o1);
RAM4K(in=in,load=b,address=address[0..11],out=o2);
RAM4K(in=in,load=c,address=address[0..11],out=o3);
RAM4K(in=in,load=d,address=address[0..11],out=o4);
Mux4Way16(a=o1,b=o2,c=o3,d=o4,sel=address[12..13],out=out);


}
```

7.Ram8

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

    PARTS:
    // Put your code here:
    DMux8Way(in=load, sel=address, a=a1, b=b1, c=c1, d=d1, e=e1, f=f1, g=g1, h=h1);
    Register(in=in, load=a1, out=o1);
    Register(in=in, load=b1, out=o2);
    Register(in=in, load=c1, out=o3);
    Register(in=in, load=d1, out=o4);
    Register(in=in, load=e1, out=o5);
    Register(in=in, load=f1, out=o6);
    Register(in=in, load=g1, out=o7);
    Register(in=in, load=h1, out=o8);
    Mux8Way16(a=o1, b=o2, c=o3, d=o4, e=o5, f=o6, g=o7, h=o8, sel=address, out=out);
}
```

8.Ram4k

```
CHIP RAM4K {
    IN in[16], load, address[12];
    OUT out[16];

    PARTS:
    // Put your code here:
DMux8Way(in=load,sel=address[9..11],a=a,b=b,c=c,d=d,e=e,f=f,g=g,h=h);
RAM512(in=in,load=a,address=address[0..8],out=o1);
RAM512(in=in,load=b,address=address[0..8],out=o2);
```

```
RAM512(in=in,load=c,address=address[0..8],out=o3);
RAM512(in=in,load=d,address=address[0..8],out=o4);
RAM512(in=in,load=e,address=address[0..8],out=o5);
RAM512(in=in,load=f,address=address[0..8],out=o6);
RAM512(in=in,load=g,address=address[0..8],out=o7);
RAM512(in=in,load=h,address=address[0..8],out=o8);
Mux8Way16(a=o1,b=o2,c=o3,d=o4,e=o5,f=o6,g=o7,h=o8,sel=address[9..11],out=out);

}
```

9.Pc

```
CHIP PC {
   IN in[16],load,inc,reset;
   OUT out[16];

   PARTS:
   // Put your code here:
Nand(a=a1,b=true,out=x1);
Nand(a=a1,b=x1,out=x2);
Nand(a=x1,b=true,out=x3);
Nand(a=x2,b=x3,out=x4);
Nand(a=x4,b=false,out=x5);
Nand(a=x4,b=x5,out=x6);
Nand(a=x5,b=false,out=x7);
Nand(a=x6,b=x7,out=out0);
Nand(a=x5,b=x1,out=y0);

Nand(a=a2,b=false,out=x8);
Nand(a=a2,b=x8,out=x9);
Nand(a=x8,b=false,out=x10);
Nand(a=x9,b=x10,out=x11);
Nand(a=x11,b=y0,out=x12);
Nand(a=x11,b=x12,out=x13);
Nand(a=x12,b=y0,out=x14);
Nand(a=x13,b=x14,out=out1);
Nand(a=x12,b=x8,out=y1);

Nand(a=a3,b=false,out=x15);
Nand(a=a3,b=x15,out=x16);
Nand(a=x15,b=false,out=x17);
Nand(a=x16,b=x17,out=x18);
Nand(a=x18,b=y1,out=x19);
Nand(a=x18,b=x19,out=x20);
Nand(a=x19,b=y1,out=x21);
Nand(a=x20,b=x21,out=out2);
Nand(a=x19,b=x15,out=y2);

Nand(a=a4,b=false,out=x22);
Nand(a=a4,b=x22,out=x23);
```

```
Nand(a=x22,b=false,out=x24);
Nand(a=x23,b=x24,out=x25);
Nand(a=x25,b=y2,out=x26);
Nand(a=x25,b=x26,out=x27);
Nand(a=x26,b=y2,out=x28);
Nand(a=x27,b=x28,out=out3);
Nand(a=x26,b=x22,out=y3);

Nand(a=a5,b=false,out=x29);
Nand(a=a5,b=x29,out=x30);
Nand(a=x29,b=false,out=x31);
Nand(a=x30,b=x31,out=x32);
Nand(a=x32,b=y3,out=x33);
Nand(a=x32,b=x33,out=x34);
Nand(a=x33,b=y3,out=x35);
Nand(a=x34,b=x35,out=out4);
Nand(a=x33,b=x29,out=y4);

Nand(a=a6,b=false,out=x36);
Nand(a=a6,b=x36,out=x37);
Nand(a=x36,b=false,out=x38);
Nand(a=x37,b=x38,out=x39);
Nand(a=x39,b=y4,out=x40);
Nand(a=x39,b=x40,out=x41);
Nand(a=x40,b=y4,out=x42);
Nand(a=x41,b=x42,out=out5);
Nand(a=x40,b=x36,out=y5);

Nand(a=a7,b=false,out=x43);
Nand(a=a7,b=x43,out=x44);
Nand(a=x43,b=false,out=x45);
Nand(a=x44,b=x45,out=x46);
Nand(a=x46,b=y5,out=x47);
Nand(a=x46,b=x47,out=x48);
Nand(a=x47,b=y5,out=x49);
Nand(a=x48,b=x49,out=out6);
Nand(a=x47,b=x43,out=y6);

Nand(a=a8,b=false,out=x50);
Nand(a=a8,b=x50,out=x51);
Nand(a=x50,b=false,out=x52);
Nand(a=x51,b=x52,out=x53);
Nand(a=x53,b=y6,out=x54);
Nand(a=x53,b=x54,out=x55);
Nand(a=x54,b=y6,out=x56);
Nand(a=x55,b=x56,out=out7);
Nand(a=x54,b=x50,out=y7);

Nand(a=a9,b=false,out=x57);
Nand(a=a9,b=x57,out=x58);
```

```
Nand(a=x57,b=false,out=x59);
Nand(a=x58,b=x59,out=x60);
Nand(a=x60,b=y7,out=x61);
Nand(a=x60,b=x61,out=x62);
Nand(a=x61,b=y7,out=x63);
Nand(a=x62,b=x63,out=out8);
Nand(a=x61,b=x57,out=y8);

Nand(a=a10,b=false,out=x64);
Nand(a=a10,b=x64,out=x65);
Nand(a=x64,b=false,out=x66);
Nand(a=x65,b=x66,out=x67);
Nand(a=x67,b=y8,out=x68);
Nand(a=x67,b=x68,out=x69);
Nand(a=x68,b=y8,out=x70);
Nand(a=x69,b=x70,out=out9);
Nand(a=x68,b=x64,out=y9);

Nand(a=a11,b=false,out=x71);
Nand(a=a11,b=x71,out=x72);
Nand(a=x71,b=false,out=x73);
Nand(a=x72,b=x73,out=x74);
Nand(a=x74,b=y9,out=x75);
Nand(a=x74,b=x75,out=x76);
Nand(a=x75,b=y9,out=x77);
Nand(a=x76,b=x77,out=out10);
Nand(a=x75,b=x71,out=y10);

Nand(a=a12,b=false,out=x78);
Nand(a=a12,b=x78,out=x79);
Nand(a=x78,b=false,out=x80);
Nand(a=x79,b=x80,out=x81);
Nand(a=x81,b=y10,out=x82);
Nand(a=x81,b=x82,out=x83);
Nand(a=x82,b=y10,out=x84);
Nand(a=x83,b=x84,out=out11);
Nand(a=x82,b=x78,out=y11);

Nand(a=a13,b=false,out=x85);
Nand(a=a13,b=x85,out=x86);
Nand(a=x85,b=false,out=x87);
Nand(a=x86,b=x87,out=x88);
Nand(a=x88,b=y11,out=x89);
Nand(a=x88,b=x89,out=x90);
Nand(a=x89,b=y11,out=x91);
Nand(a=x90,b=x91,out=out12);
Nand(a=x89,b=x85,out=y12);

Nand(a=a14,b=false,out=x92);
Nand(a=a14,b=x92,out=x93);
```

```
Nand(a=x92,b=false,out=x94);
Nand(a=x93,b=x94,out=x95);
Nand(a=x95,b=y12,out=x96);
Nand(a=x95,b=x96,out=x97);
Nand(a=x96,b=y12,out=x98);
Nand(a=x97,b=x98,out=out13);
Nand(a=x96,b=x92,out=y13);

Nand(a=a15,b=false,out=x99);
Nand(a=a15,b=x99,out=x100);
Nand(a=x99,b=false,out=x101);
Nand(a=x100,b=x101,out=x102);
Nand(a=x102,b=y13,out=x103);
Nand(a=x102,b=x103,out=x104);
Nand(a=x103,b=y13,out=x105);
Nand(a=x104,b=x105,out=out14);
Nand(a=x103,b=x99,out=y14);

Nand(a=a16,b=false,out=x106);
Nand(a=a16,b=x106,out=x107);
Nand(a=x106,b=false,out=x108);
Nand(a=x107,b=x108,out=x109);
Nand(a=x109,b=y14,out=x110);
Nand(a=x109,b=x110,out=x111);
Nand(a=x110,b=y14,out=x112);
Nand(a=x111,b=x112,out=out15);
Nand(a=x110,b=x106,out=drop);

///mux16

Nand(a=inc,b=inc,out=c);
Nand(a=c,b=a1,out=l1);
Nand(a=inc,b=out0,out=l2);
Nand(a=l1,b=l2,out=lOut0);

Nand(a=c,b=a2,out=l3);
Nand(a=inc,b=out1,out=l4);
Nand(a=l3,b=l4,out=lOut1);

Nand(a=c,b=a3,out=l5);
Nand(a=inc,b=out2,out=l6);
Nand(a=l5,b=l6,out=lOut2);

Nand(a=c,b=a4,out=l7);
Nand(a=inc,b=out3,out=l8);
Nand(a=l7,b=l8,out=lOut3);

Nand(a=c,b=a5,out=l9);
Nand(a=inc,b=out4,out=l10);
Nand(a=l9,b=l10,out=lOut4);
```

```
Nand(a=c,b=a6,out=l11);
Nand(a=inc,b=out5,out=l12);
Nand(a=l11,b=l12,out=lOut5);

Nand(a=c,b=a7,out=l13);
Nand(a=inc,b=out6,out=l14);
Nand(a=l13,b=l14,out=lOut6);

Nand(a=c,b=a8,out=l15);
Nand(a=inc,b=out7,out=l16);
Nand(a=l15,b=l16,out=lOut7);

Nand(a=c,b=a9,out=l17);
Nand(a=inc,b=out8,out=l18);
Nand(a=l17,b=l18,out=lOut8);

Nand(a=c,b=a10,out=l19);
Nand(a=inc,b=out9,out=l20);
Nand(a=l19,b=l20,out=lOut9);

Nand(a=c,b=a11,out=l21);
Nand(a=inc,b=out10,out=l22);
Nand(a=l21,b=l22,out=lOut10);

Nand(a=c,b=a12,out=l23);
Nand(a=inc,b=out11,out=l24);
Nand(a=l23,b=l24,out=lOut11);

Nand(a=c,b=a13,out=l25);
Nand(a=inc,b=out12,out=l26);
Nand(a=l25,b=l26,out=lOut12);

Nand(a=c,b=a14,out=l27);
Nand(a=inc,b=out13,out=l28);
Nand(a=l27,b=l28,out=lOut13);

Nand(a=c,b=a15,out=l29);
Nand(a=inc,b=out14,out=l30);
Nand(a=l29,b=l30,out=lOut14);

Nand(a=s,b=a16,out=oo31);
Nand(a=inc,b=out15,out=oo32);
Nand(a=oo31,b=oo32,out=lOut15);
////mux16

Nand(a=load,b=load,out=L);
Nand(a=L,b=lOut0,out=O1);
Nand(a=load,b=in[0],out=O2);
Nand(a=O1,b=O2,out=oo0);
```

```
Nand(a=L,b=lOut1,out=O3);
Nand(a=load,b=in[1],out=O4);
Nand(a=O3,b=O4,out=oo1);

Nand(a=L,b=lOut2,out=O5);
Nand(a=load,b=in[2],out=O6);
Nand(a=O5,b=O6,out=oo2);

Nand(a=L,b=lOut3,out=O7);
Nand(a=load,b=in[3],out=O8);
Nand(a=O7,b=O8,out=oo3);

Nand(a=L,b=lOut4,out=O9);
Nand(a=load,b=in[4],out=O10);
Nand(a=O9,b=O10,out=oo4);

Nand(a=L,b=lOut5,out=O11);
Nand(a=load,b=in[5],out=O12);
Nand(a=O11,b=O12,out=oo5);

Nand(a=L,b=lOut6,out=O13);
Nand(a=load,b=in[6],out=O14);
Nand(a=O13,b=O14,out=oo6);

Nand(a=L,b=lOut7,out=O15);
Nand(a=load,b=in[7],out=O16);
Nand(a=O15,b=O16,out=oo7);

Nand(a=L,b=lOut8,out=O17);
Nand(a=load,b=in[8],out=O18);
Nand(a=O17,b=O18,out=oo8);

Nand(a=L,b=lOut9,out=O19);
Nand(a=load,b=in[9],out=O20);
Nand(a=O19,b=O20,out=oo9);

Nand(a=L,b=lOut10,out=O21);
Nand(a=load,b=in[10],out=O22);
Nand(a=O21,b=O22,out=oo10);

Nand(a=L,b=lOut11,out=O23);
Nand(a=load,b=in[11],out=O24);
Nand(a=O23,b=O24,out=oo11);

Nand(a=L,b=lOut12,out=O25);
Nand(a=load,b=in[12],out=O26);
Nand(a=O25,b=O26,out=oo12);

Nand(a=L,b=lOut13,out=O27);
```

```
Nand(a=load,b=in[13],out=O28);
Nand(a=O27,b=O28,out=oo13);

Nand(a=L,b=lOut14,out=O29);
Nand(a=load,b=in[14],out=O30);
Nand(a=O29,b=O30,out=oo14);

Nand(a=L,b=lOut15,out=O31);
Nand(a=load,b=in[15],out=O32);
Nand(a=O31,b=O32,out=oo15);
///mux16

Nand(a=reset,b=reset,out=d);
Nand(a=d,b=oo0,out=n1);
Nand(a=reset,b=false,out=n2);
Nand(a=n1,b=n2,out=nu0);

Nand(a=d,b=oo1,out=n3);
Nand(a=reset,b=false,out=n4);
Nand(a=n3,b=n4,out=nu1);

Nand(a=d,b=oo2,out=n5);
Nand(a=reset,b=false,out=n6);
Nand(a=n5,b=n6,out=nu2);

Nand(a=d,b=oo3,out=n7);
Nand(a=reset,b=false,out=n8);
Nand(a=n7,b=n8,out=nu3);

Nand(a=d,b=oo4,out=n9);
Nand(a=reset,b=false,out=n10);
Nand(a=n9,b=n10,out=nu4);

Nand(a=d,b=oo5,out=n11);
Nand(a=reset,b=false,out=n12);
Nand(a=n11,b=n12,out=nu5);

Nand(a=d,b=oo6,out=n13);
Nand(a=reset,b=false,out=n14);
Nand(a=n13,b=n14,out=nu6);

Nand(a=d,b=oo7,out=n15);
Nand(a=reset,b=false,out=n16);
Nand(a=n15,b=n16,out=nu7);

Nand(a=d,b=oo8,out=n17);
Nand(a=reset,b=false,out=n18);
Nand(a=n17,b=n18,out=nu8);

Nand(a=d,b=oo9,out=n19);
```

```
Nand(a=reset,b=false,out=n20);
Nand(a=n19,b=n20,out=nu9);

Nand(a=d,b=oo10,out=n21);
Nand(a=reset,b=false,out=n22);
Nand(a=n21,b=n22,out=nu10);

Nand(a=d,b=oo11,out=n23);
Nand(a=reset,b=false,out=n24);
Nand(a=n23,b=n24,out=nu11);

Nand(a=d,b=oo12,out=n25);
Nand(a=reset,b=false,out=n26);
Nand(a=n25,b=n26,out=nu12);

Nand(a=d,b=oo13,out=n27);
Nand(a=reset,b=false,out=n28);
Nand(a=n27,b=n28,out=nu13);

Nand(a=d,b=oo14,out=n29);
Nand(a=reset,b=false,out=n30);
Nand(a=n29,b=n30,out=nu14);

Nand(a=d,b=oo15,out=n31);
Nand(a=reset,b=false,out=n32);
Nand(a=n31,b=n32,out=nu15);

Nand(a=inc,b=inc,out=oRn1);
Nand(a=load,b=load,out=oRn2);
Nand(a=oRn1,b=oRn2,out=oRnut);

Nand(a=oRnut,b=oRnut,out=n1oR);
Nand(a=reset,b=reset,out=n2oR);
Nand(a=n1oR,b=n2oR,out=nr);
/////register

Nand(a=nr,b=nr,out=s);
Nand(a=s,b=a1,out=f11);
Nand(a=nr,b=nu0,out=f21);
Nand(a=f11,b=f21,out=m1);
DFF(in=m1,out=out[0],out=a1);

Nand(a=s,b=a2,out=f12);
Nand(a=nr,b=nu1,out=f22);
Nand(a=f12,b=f22,out=m2);
DFF(in=m2,out=out[1],out=a2);

Nand(a=s,b=a3,out=f13);
Nand(a=nr,b=nu2,out=f23);
Nand(a=f13,b=f23,out=m3);
```

```
DFF(in=m3,out=out[2],out=a3);

Nand(a=s,b=a4,out=f14);
Nand(a=nr,b=nu3,out=f24);
Nand(a=f14,b=f24,out=m4);
DFF(in=m4,out=out[3],out=a4);

Nand(a=s,b=a5,out=f15);
Nand(a=nr,b=nu4,out=f25);
Nand(a=f15,b=f25,out=m5);
DFF(in=m5,out=out[4],out=a5);

Nand(a=s,b=a6,out=f16);
Nand(a=nr,b=nu5,out=f26);
Nand(a=f16,b=f26,out=m6);
DFF(in=m6,out=out[5],out=a6);

Nand(a=s,b=a7,out=f17);
Nand(a=nr,b=nu6,out=f27);
Nand(a=f17,b=f27,out=m7);
DFF(in=m7,out=out[6],out=a7);

Nand(a=s,b=a8,out=f18);
Nand(a=nr,b=nu7,out=f28);
Nand(a=f18,b=f28,out=m8);
DFF(in=m8,out=out[7],out=a8);

Nand(a=s,b=a9,out=f19);
Nand(a=nr,b=nu8,out=f29);
Nand(a=f19,b=f29,out=m9);
DFF(in=m9,out=out[8],out=a9);

Nand(a=s,b=a10,out=f110);
Nand(a=nr,b=nu9,out=f210);
Nand(a=f110,b=f210,out=m10);
DFF(in=m10,out=out[9],out=a10);

Nand(a=s,b=a11,out=f111);
Nand(a=nr,b=nu10,out=f211);
Nand(a=f111,b=f211,out=m11);
DFF(in=m11,out=out[10],out=a11);

Nand(a=s,b=a12,out=f112);
Nand(a=nr,b=nu11,out=f212);
Nand(a=f112,b=f212,out=m12);
DFF(in=m12,out=out[11],out=a12);

Nand(a=s,b=a13,out=f113);
Nand(a=nr,b=nu12,out=f213);
Nand(a=f113,b=f213,out=m13);
```

```
DFF(in=m13,out=out[12],out=a13);

Nand(a=s,b=a14,out=f114);
Nand(a=nr,b=nu13,out=f214);
Nand(a=f114,b=f214,out=m14);
DFF(in=m14,out=out[13],out=a14);

Nand(a=s,b=a15,out=f115);
Nand(a=nr,b=nu14,out=f215);
Nand(a=f115,b=f215,out=m15);
DFF(in=m15,out=out[14],out=a15);

Nand(a=s,b=a16,out=f116);
Nand(a=nr,b=nu15,out=f216);
Nand(a=f116,b=f216,out=m16);
DFF(in=m16,out=out[15],out=a16);
}
```

10.Bit

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

    PARTS:
    // Put your code here:
  Mux(a=a,b=in,sel=load,out=o1);
  DFF(in=o1,out=out,out=a);
}
```

11.Inc16

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

    PARTS:
   // Put you code here:
Nand(a=in[0],b=true,out=oA1);
Nand(a=in[0],b=oA1,out=oA2);
Nand(a=true,b=oA1,out=oA3);
Nand(a=oA1,b=oA1,out=carry1);
Nand(a=oA2,b=oA3,out=out[0]);

Nand(a=in[1],b=carry1,out=oB1);
Nand(a=in[1],b=oB1,out=oB2);
Nand(a=carry1,b=oB1,out=oB3);
Nand(a=oB1,b=oB1,out=carry2);
Nand(a=oB2,b=oB3,out=out[1]);

Nand(a=in[2],b=carry2,out=oC1);
Nand(a=in[2],b=oC1,out=oC2);
Nand(a=carry2,b=oC1,out=oC3);
```

```
Nand(a=oC1,b=oC1,out=carry3);
Nand(a=oC2,b=oC3,out=out[2]);


Nand(a=in[3],b=carry3,out=oD1);
Nand(a=in[3],b=oD1,out=oD2);
Nand(a=carry3,b=oD1,out=oD3);
Nand(a=oD1,b=oD1,out=carry4);
Nand(a=oD2,b=oD3,out=out[3]);


Nand(a=in[4],b=carry4,out=oE1);
Nand(a=in[4],b=oE1,out=oE2);
Nand(a=carry4,b=oE1,out=oE3);
Nand(a=oE1,b=oE1,out=carry5);
Nand(a=oE2,b=oE3,out=out[4]);


Nand(a=in[5],b=carry5,out=oF1);
Nand(a=in[5],b=oF1,out=oF2);
Nand(a=carry5,b=oF1,out=oF3);
Nand(a=oF1,b=oF1,out=carry6);
Nand(a=oF2,b=oF3,out=out[5]);


Nand(a=in[6],b=carry6,out=oG1);
Nand(a=in[6],b=oG1,out=oG2);
Nand(a=carry6,b=oG1,out=oG3);
Nand(a=oG1,b=oG1,out=carry7);
Nand(a=oG2,b=oG3,out=out[6]);


Nand(a=in[7],b=carry7,out=oH1);
Nand(a=in[7],b=oH1,out=oH2);
Nand(a=carry7,b=oH1,out=oH3);
Nand(a=oH1,b=oH1,out=carry8);
Nand(a=oH2,b=oH3,out=out[7]);


Nand(a=in[8],b=carry8,out=oI1);
Nand(a=in[8],b=oI1,out=oI2);
Nand(a=carry8,b=oI1,out=oI3);
Nand(a=oI1,b=oI1,out=carry9);
Nand(a=oI2,b=oI3,out=out[8]);

Nand(a=in[9],b=carry9,out=oJ1);
Nand(a=in[9],b=oJ1,out=oJ2);
Nand(a=carry9,b=oJ1,out=oJ3);
Nand(a=oJ1,b=oJ1,out=carry10);
Nand(a=oJ2,b=oJ3,out=out[9]);


Nand(a=in[10],b=carry10,out=oK1);
Nand(a=in[10],b=oK1,out=oK2);
Nand(a=carry10,b=oK1,out=oK3);
```

```
Nand(a=oK1,b=oK1,out=carry11);
Nand(a=oK2,b=oK3,out=out[10]);


Nand(a=in[11],b=carry11,out=oL1);
Nand(a=in[11],b=oL1,out=oL2);
Nand(a=carry11,b=oL1,out=oL3);
Nand(a=oL1,b=oL1,out=carry12);
Nand(a=oL2,b=oL3,out=out[11]);


Nand(a=in[12],b=carry12,out=oM1);
Nand(a=in[12],b=oM1,out=oM2);
Nand(a=carry12,b=oM1,out=oM3);
Nand(a=oM1,b=oM1,out=carry13);
Nand(a=oM2,b=oM3,out=out[12]);


Nand(a=in[13],b=carry13,out=oN1);
Nand(a=in[13],b=oN1,out=oN2);
Nand(a=carry13,b=oN1,out=oN3);
Nand(a=oN1,b=oN1,out=carry14);
Nand(a=oN2,b=oN3,out=out[13]);

Nand(a=in[14],b=carry14,out=oO1);
Nand(a=in[14],b=oO1,out=oO2);
Nand(a=carry14,b=oO1,out=oO3);
Nand(a=oO1,b=oO1,out=carry15);
Nand(a=oO2,b=oO3,out=out[14]);

Nand(a=in[15],b=carry15,out=oP1);
Nand(a=in[15],b=oP1,out=oP2);
Nand(a=carry15,b=oP1,out=oP3);
Nand(a=oP1,b=oP1,out=carry16);
Nand(a=oP2,b=oP3,out=out[15]);

}
```

12.Dmux8Way

```
CHIP DMux8Way {
   IN in, sel[3];
   OUT a, b, c, d, e, f, g, h;

   PARTS:
   // Put your code here:
DMux(in=in,sel=sel[2],a=a1,b=b1);
DMux(in=a1,sel=sel[1],a=a2,b=b2);
DMux(in=b1,sel=sel[1],a=c1,b=d1);
DMux(in=a2,sel=sel[0],a=a,b=b);
DMux(in=b2,sel=sel[0],a=c,b=d);
DMux(in=c1,sel=sel[0],a=e,b=f);
DMux(in=d1,sel=sel[0],a=g,b=h);
```

}

13.Mux4way16

```
CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16], sel[2];
    OUT out[16];

    PARTS:
    // Put your code here:
Mux16(a=a,b=b,sel=sel[0],out=o1);
Mux16(a=c,b=d,sel=sel[0],out=o2);
Mux16(a=o1,b=o2,sel=sel[1],out=out);
}
```

1.10 Outputs:

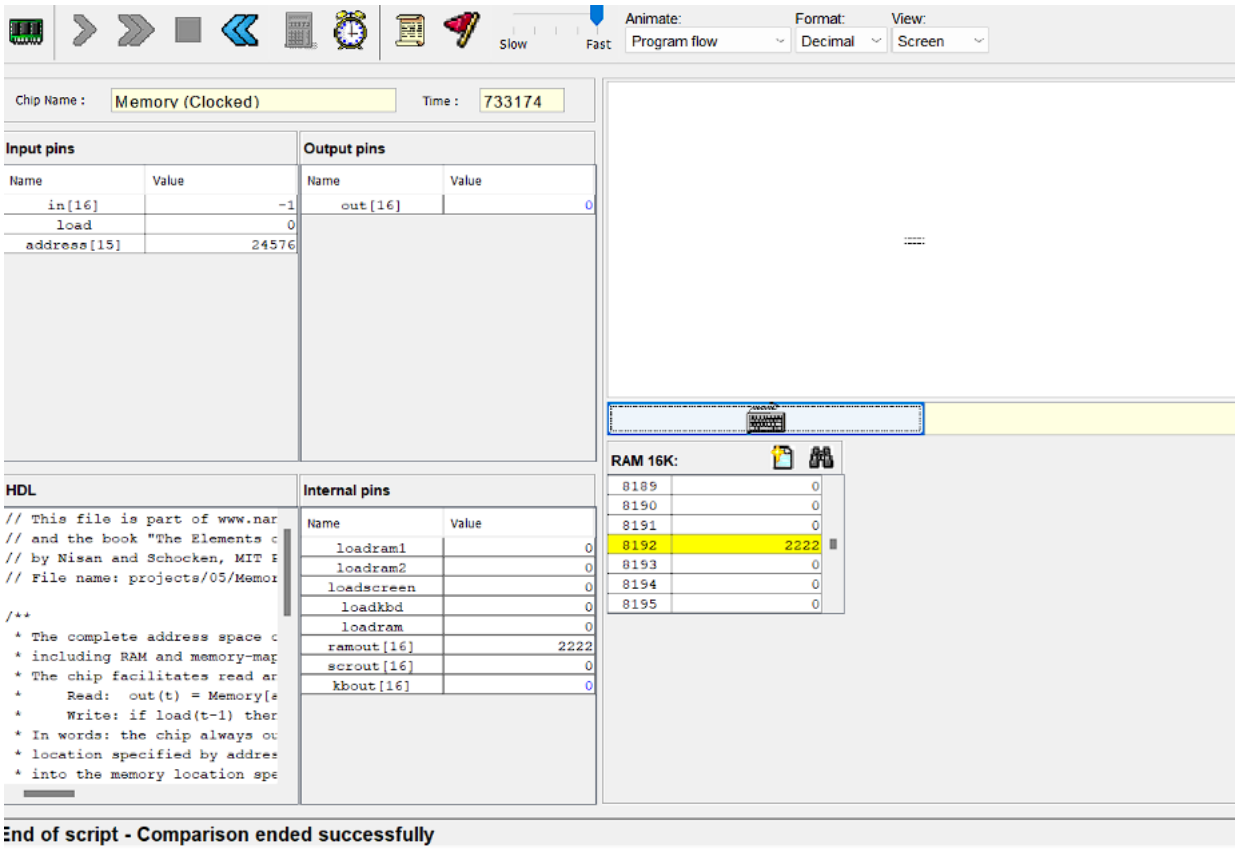

Fig 1.1

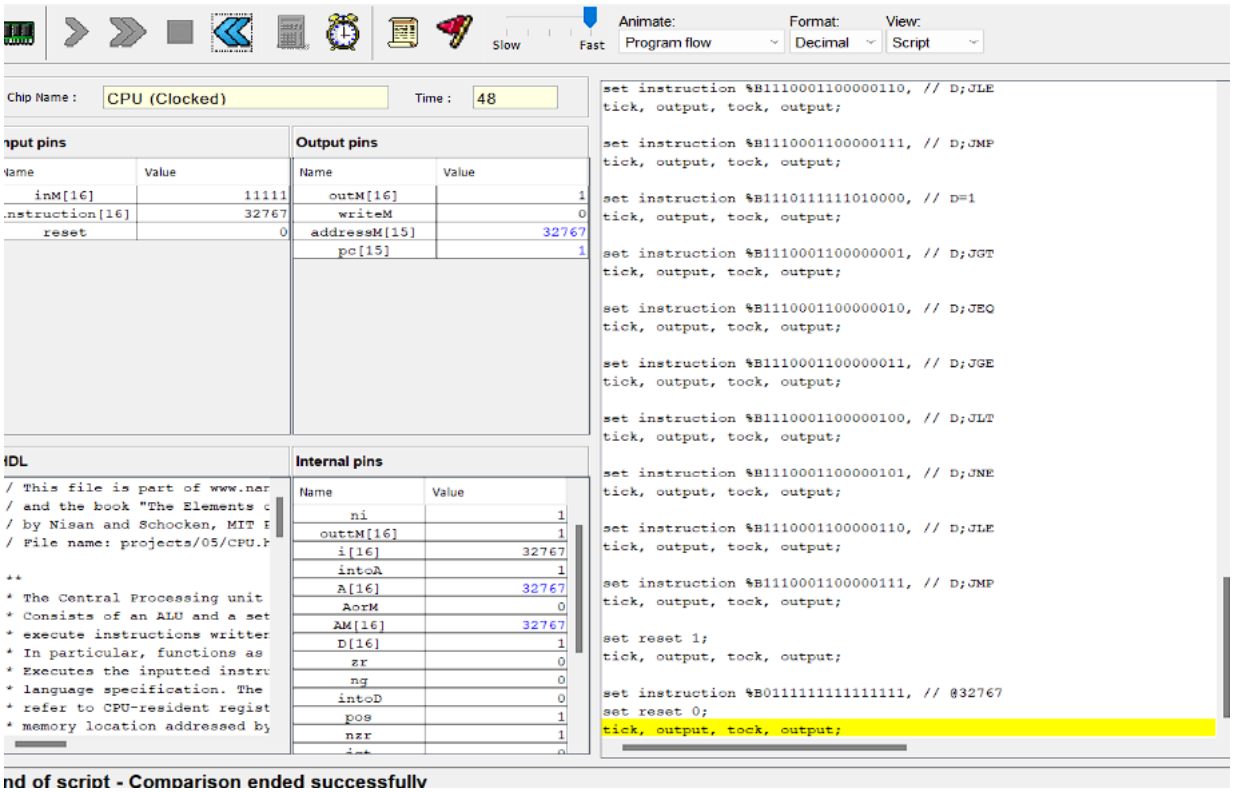

Fig 1.2

load Computer.hdl,

output-file ComputerMain.out,

compare-to ComputerMain.cmp,

output-list time%S1.4.1 reset%B2.1.2 ARegister[0]%D1.7.1 DRegister[0]%D1.7.1 PC[]%D0.4.0 RAM16K[0]%D1.7.1 RAM16K[1]%D1.7.1 RAM16K[2]%D1.7.1;

// Load a program written in the Hack machine language.

// The program finds the factorial of 3 and writes the result in RAM[0].

ROM32K load ComputerMain.hack,

output;

// First run (at the beginning PC=0)

repeat 375 {

    tick, tock, output;

}

Testing the tst file :

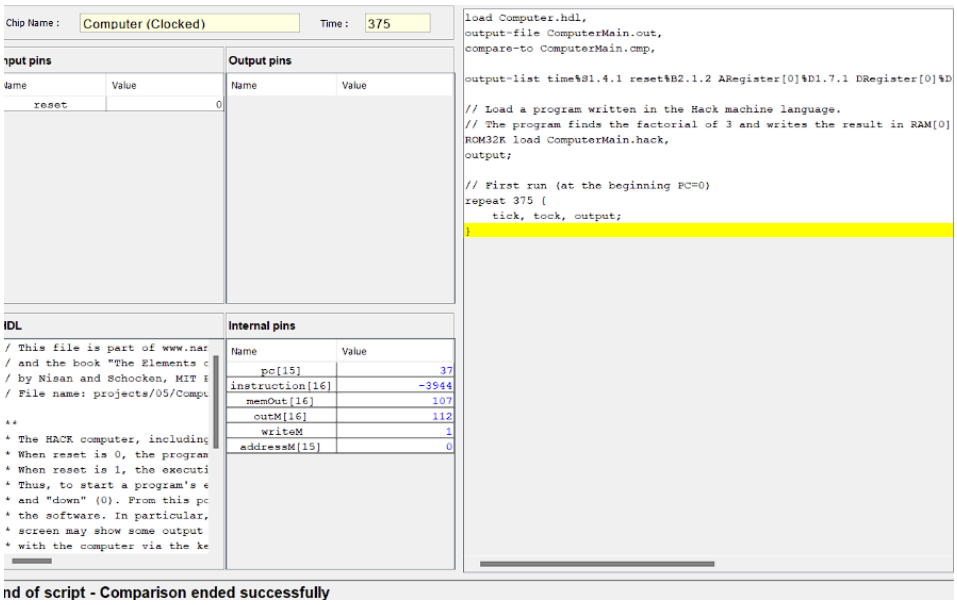

Fig 1.3

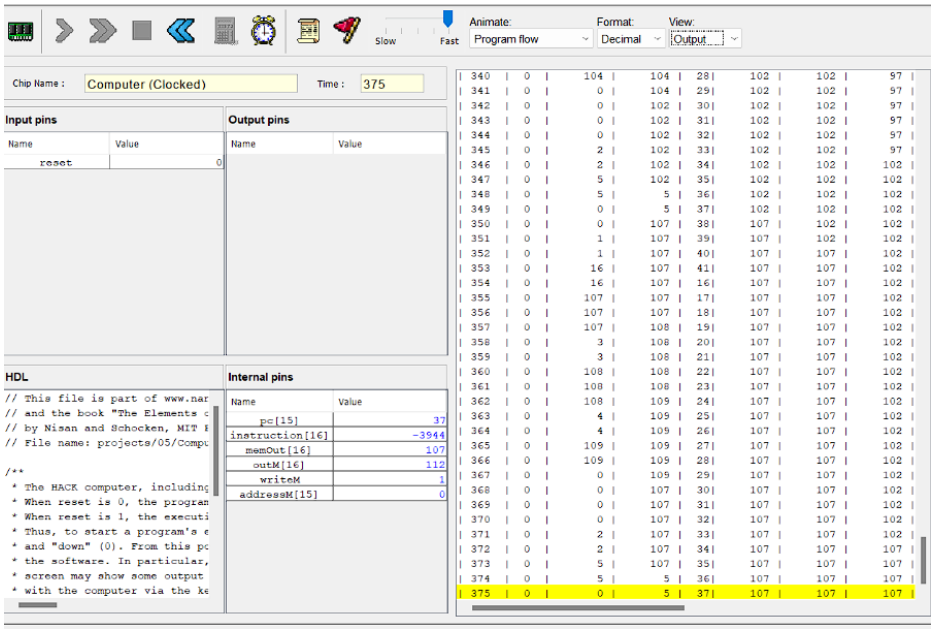

Fig 1.4

1.11 CONCLUSION:

In conclusion, this project aimed to implement the Hack computer architecture using Hardware Description Language (HDL) within the Nand2Tetris framework. By constructing the Hack computer from simple logic gates and developing the digital circuitry for its CPU, RAM, ROM, and other components using HDL, we have gained a deeper understanding of computer architecture, digital logic, and low-level programming.

Throughout the project, various modules such as the arithmetic logic unit (ALU), control unit, memory units, and input/output devices were designed and connected using HDL code. This ensured the natural interaction and seamless functionality of these modules. By doing so, our abilities in HDL programming and circuit design have significantly grown.

The successful implementation of the Hack computer in HDL will result in a fully functional computer system capable of running programs written in the Hack Assembly Language. This achievement highlights the practical application of our knowledge and skills in computer engineering and provides a solid foundation for further exploration in the field.

Overall, this project has been instrumental in enhancing our understanding of computer architecture, strengthening our HDL programming and circuit design abilities, and demonstrating the feasibility of building a functioning computer system from basic logic gates. The skills and knowledge acquired throughout this endeavor will undoubtedly contribute to our future endeavors in the realm of digital design and computer engineering.