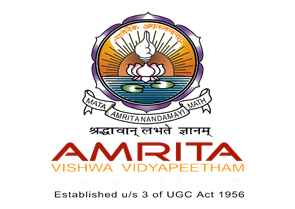
****

**Elements of Computing Systems – 2\_B.Tech.. 2019.R.AIE.1.19AIE112**

**TEAM - 6**

**PROJECT BY:**

1. **ABIJITH PRADEEP CB.EN.U4AIE19002**
2. **ROHITH G CB.EN.U4AIE19026**
3. **KAUSHIK M CB.EN.U4AIE19036**
4. **TARUN SANJEEV S CB.EN.U4AIE19064**
5. **VIJAI SIMMON S CB.EN.U4AIE19068**

**Assembly Language**

In computer programming, assembly language (assembler language) abbreviated as **asm**, is a low-level programming language, in which there is a very strong correspondence between the instructions in the language and the architecture.  Because assembly depends on the machine code instructions, every assembler has its own assembly language which is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code.

Assembly code is converted into executable machine code by a [utility program](https://en.wikipedia.org/wiki/Utility_software) referred to as an [assembler](https://en.wikipedia.org/wiki/Assembly_language#Assembler). The conversion process is referred to as assembly, as in assembling the [source code](https://en.wikipedia.org/wiki/Source_code). Assembly language usually has one statement per machine instruction , but [comments](https://en.wikipedia.org/wiki/Comment_(computer_programming)) and statements that are assembler [directives](https://en.wikipedia.org/wiki/Directive_(programming)), [macros](https://en.wikipedia.org/wiki/Macro_instruction), and symbolic [labels of program](https://en.wikipedia.org/wiki/Label_(programming)) and [memory locations](https://en.wikipedia.org/wiki/Memory_location) are often also supported

Each assembly language is specific to a particular [computer architecture](https://en.wikipedia.org/wiki/Computer_architecture) and sometimes to an [operating system](https://en.wikipedia.org/wiki/Operating_system). However, some assembly languages do not provide specific [syntax](https://en.wikipedia.org/wiki/Syntax_(programming_languages)) for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the [processor](https://en.wikipedia.org/wiki/Computer_processor), upon which all [system call](https://en.wikipedia.org/wiki/System_call) mechanisms ultimately rest. In contrast to assembly languages, most [high-level programming languages](https://en.wikipedia.org/wiki/High-level_programming_language) are generally [portable](https://en.wikipedia.org/wiki/Porting) across multiple architectures but require [interpreting](https://en.wikipedia.org/wiki/Interpreter_(computing)) or [compiling](https://en.wikipedia.org/wiki/Compiler), a much more complicated task than assembling.

Assembly language uses a [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) to represent each low-level [machine instruction](https://en.wikipedia.org/wiki/Machine_code) or [opcode](https://en.wikipedia.org/wiki/Opcode), typically also each [architectural register](https://en.wikipedia.org/wiki/Register_(computing)#ARCHITECTURAL), [flag](https://en.wikipedia.org/wiki/Bit_field), etc. Many operations require one or more [operands](https://en.wikipedia.org/wiki/Operand#Computer_science) in order to form a complete instruction. Most assemblers permit named constants, registers, and [labels](https://en.wikipedia.org/wiki/Label_(computer_science)) for program and memory locations, and can calculate [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science)) for operands. Thus, the programmers are freed from tedious repetitive calculations and assembler programs are much more readable than machine code. Depending on the architecture, these elements may also be combined for specific instructions or [addressing modes](https://en.wikipedia.org/wiki/Addressing_mode) using [offsets](https://en.wikipedia.org/wiki/Offset_(computer_science)) or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid [debugging](https://en.wikipedia.org/wiki/Debugging).

**Assembler**

An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer. An assembler is sometimes referred to as the compiler of assembly language.

An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.

An assembler primarily serves as the bridge between symbolically coded instructions written in assembly language and the computer processor, memory and other computational components. An assembler works by assembling and converting the source code of assembly language into object code or an object file that constitutes a stream of zeros and ones of machine code, which are directly executable by the processor.

**Machine language**

Machine language is the lowest-level [programming language](https://www.webopedia.com/TERM/P/programming_language.html) . Machine languages are the only [languages](https://www.webopedia.com/TERM/L/language.html) understood by [computers](https://www.webopedia.com/TERM/C/computer.html).

machine language is a collection of binary digits or bits that the computer reads and interprets. A computer cannot directly understand the programming languages used to create computer programs, so the program code must be compiled.

**Difference between assembly language and Machine language:**

Machine language and assembly language are both low-level languages, but machine code is below assembly in the hierarchy of computer languages. Assembly language includes human-readable commands, such as mov, add, and sub, while machine language does not contain any words or even letters. Some [developers](https://techterms.com/definition/developer) manually write assembly language to optimize a program, but they do not write machine code. Only developers who write software compilers need to worry about machine language.

**INSTRUCTION TYPES**

There are two types of instructions – **A-instruction** and   
**C-Instruction**.

The A-instruction deals with is used to set the A-register to a 15-bit value.

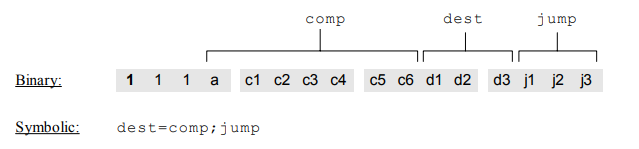


The symbolic representation is:

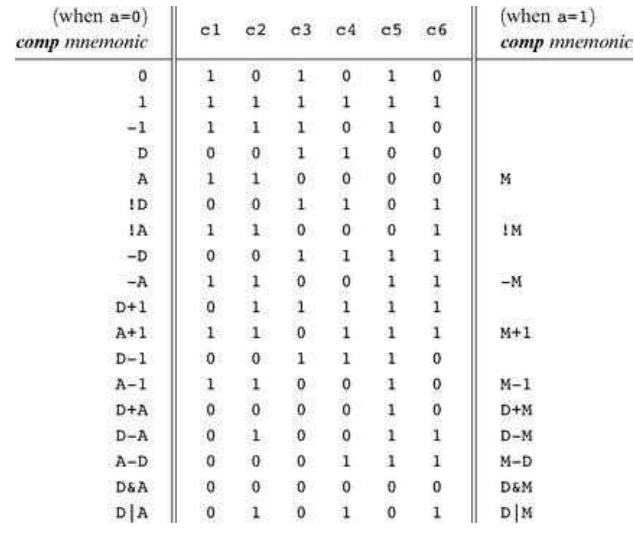
**Symbolic: @value**

The 15-bit constant is the binary representation of the value (in decimal). This instruction enables to store the constant in the A register.

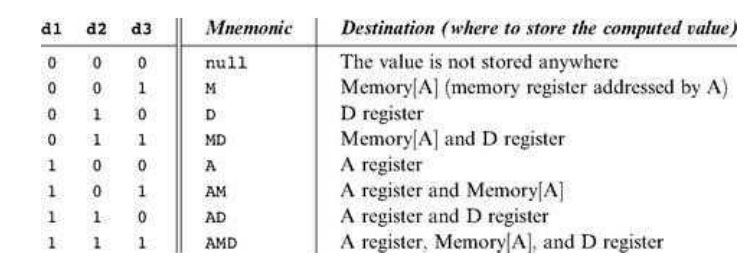
The main uses of A-instruction are to insert constant values into the computer, setting up the stage for C-instruction so that computed value is stored at the Ram Location pointed from A-instruction, and setting up stage for C-instruction for jumping commands as jumping destinations get stored at A register.

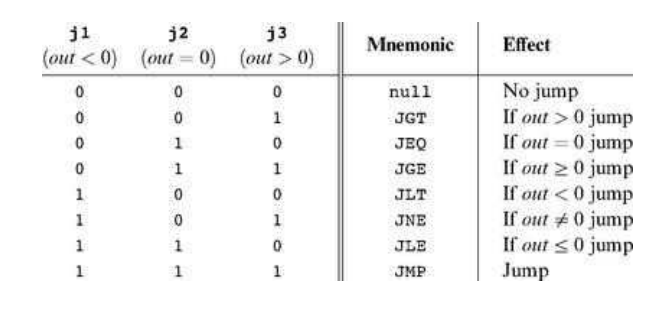
On the other hand, the C-instruction deals almost with everything required for computation. This C-instruction has three major parts-What to compute, Where to store and What to do next. 

The MSB of a C-instruction is **1**. The ***comp*** field lets the CPU know what to compute. ***Dest*** field indicates the RAM location where the computed value has to be stored. ***Jump*** field specifies the jump condition. The computation specification is listed below:



We see that when a = 0, A register is used and when a = 1, M registers are used.

For the destination, specification is: We see that d1 is 1 whenever A register is present in the mnemonic, d2 is 1 whenever D register is mentioned and d3 is 1 whenever M register is used.

The Jump instructions are as follows: 

Based on these specifications, we are able to change the assembly language code to 16-bit machine language code.

**EXPERIMENT ASSIGNED**

**AIM:**

The aim of our project is to show an output 0 when the input given is greater than 50, else the output is set to 1.

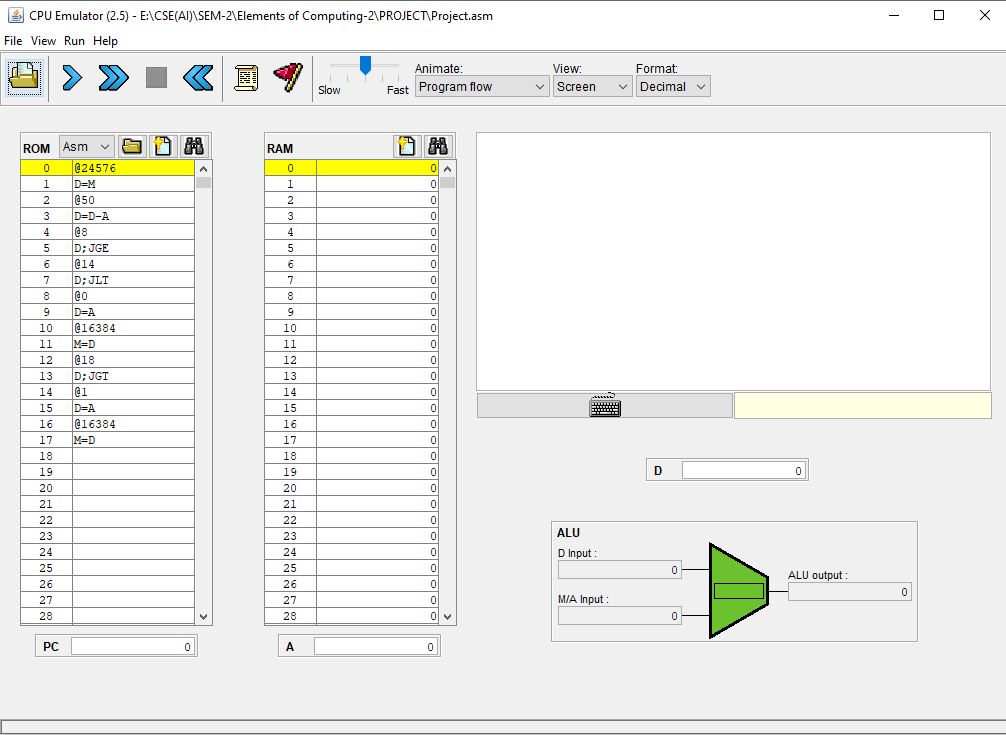
**The project has to be done in 3 parts:**

1. Creation of the asm code and simulate in CPU Emulator and check the output.
2. Pass the asm file to the assembler built and retrieve the machine code. Verify the machine code obtained from the hack assembler and that which was built.
3. Create a hack computer using hdl files and show the corresponding working in hack simulator.

**PART-1**

**THE ASM CODE FOR THE PROJECT IS:**

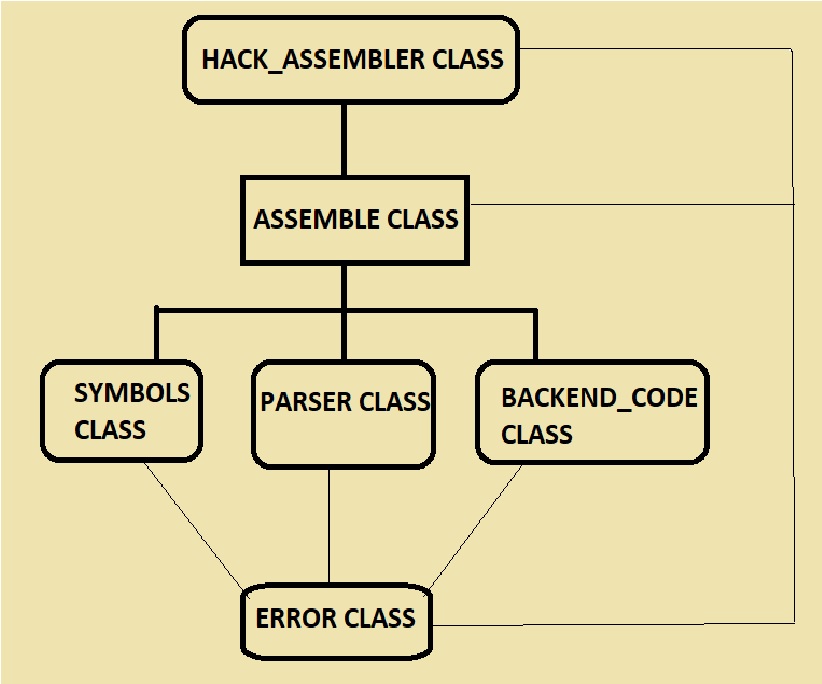
@KBD   
 D=M  
 @50  
 D=D-A  
 @temp1  
 D;JGE  
 @temp2  
 D;JLT  
(temp1)  
 @0  
 D=A  
 @SCREEN  
 M=D  
 @END  
 D;JGT  
(temp2)  
 @1  
 D=A  
 @SCREEN  
 M=D  
(END)



We use the CPU Emulator, load our asm file and click the keyboard symbol and press any key from the keyboard and start the simulation.

If the ASCII value of the input is less than 50, we get a black dot at the screen depicting the 1 pixel of the screen and the RAM location of SCREEN stores the content 1.   
  
If the value is greater than 50, there is no change in screen as the RAM location of SCREEN would have the value of 0.

**PART-2**



The Assembler created consists of the classes mentioned in the figure. The diagram is the skeletal structure of the whole assembler that has been created. The **Hack\_Assembler class** is the front-end class. It calls the **Assemble class** and the assemble class calls functions present in 3 more classes, namely-**Symbols class, Parser class, and Backend\_Code class**. All these classes at some point of time lead to some exceptions or errors. So, we create a **Error class** linked to all the classes and would deal with corresponding error that has been occurred.

Now, we will include the coding of the Assembler:

**Parser Class:** It deals with traversing through the asm file and then analysing the commands, like determining whether it is a A-command, or C-command or the L-Command.

It also deals with splitting of the commands in a manner to convert this assembly language to the machine language based on the specifications mentioned in the previous section.

**CODE FOR PARSER CLASS:**package EOC;  
import java.io.\*;

public class Parser

{

// current command of a line

public String currentCommand = "";

// file being assembled

public String inputFile;

// current line number in the file

public int lineNumber = 0;

// current line in file

public String currentLine;

//Input file reader

private BufferedReader fileReader;

// We open the input file.

public Parser(String file) throws FileNotFoundException

{

inputFile = file;

fileReader = new BufferedReader(new FileReader(file));

lineNumber = 0;

}

//To read through the input file

public boolean advance() throws IOException

{

while (true)

{ //Reading line by line from input .asm file.

currentLine = fileReader.readLine();

lineNumber++;

if (currentLine == null)

return false;

//We remove all the comments that we type in the .asm file and remove the space by trim function

currentCommand = currentLine.replaceAll("//.\*$", "").trim();

if (currentCommand.equals(""))

continue;

return true;

}

}

// The function to return type of command

public int commandType()

{

if (currentCommand.startsWith("@"))

{ //A-Command

return 1;

} else if (currentCommand.startsWith("("))

{ //L-Command

return 2;

} else

{ //C-Command

return 3;

}

}

// For A-Command / L-Command

public String symbol()

{

return currentCommand.substring(1).replace(")", "");

}

// For the C-Command

public String dest()

{

String dest = "";

if (currentCommand.contains("="))

{

String[] array = currentCommand.split("=");

dest = array[0];

}

return dest;

}

// returns the comp field of the current command (28 possibilities)

// only applies to C\_COMMAND

public String comp()

{

String comp;

if (currentCommand.contains("="))

{

String[] array = currentCommand.split("=");

String[] array1 = array[1].split(";");

comp = array1[0];

} else

{

String[] array = currentCommand.split(";");

comp = array[0];

}

return comp;

}

// returns the jump field of the current command (8 possibilities)

// only applies to C\_COMMAND

public String jump()

{

String jump = "";

if (currentCommand.contains(";"))

{

String[] array = currentCommand.split(";");

jump = array[1];

}

return jump;

}

// close input file

public void close() throws IOException

{

fileReader.close();

return;

}

}

**Symbols Class:** It deals with the predefined RAM locations such as the RAM locations of the Keyboard(KBD) and the Screen. The class uses the inbuilt HashMap datatype so as to create a look up table and as we access a label, we get the corresponding RAM address.

**CODE:**package EOC;  
import java.util.\*;

public class Symbols{

private HashMap<String, Integer> symbolTable;

// create an empty symbol table - Hash Map

public Symbols()

{

symbolTable = new HashMap<String, Integer>();

}

// Adding Entry to the symbol HashMap table

public void addEntry(String label, int address)

{

symbolTable.put(label, address);

return;

}

//We check whether the Label exists as a key in the HashMap Table.

public boolean contains(String label)

{

return symbolTable.containsKey(label);

}

/\*The Label is the Key, When we use get(), we get the Value of the Key-Value pair.

So, the address location is returned from the HashMap.

\*/

public int getAddress(String label)

{

return symbolTable.get(label);

}

// We Initialize the HashMap with key-aspects of Registers.

public void initialize()

{

// Virtual Registers

for (int i = 0; i < 16; i++)

this.addEntry("R" + i, i);

// The Input/Output Pointers

this.addEntry("KBD", 24576);

this.addEntry("SCREEN", 16384);

//Other Labels for the arguments-Note that address is considered as assumption.

this.addEntry("SP", 0);

this.addEntry("LCL", 1);

this.addEntry("ARG", 2);

this.addEntry("THIS", 3);

this.addEntry("THAT", 4);

return;

}

}

**Backend\_Code Class:**This class deals with specification for the commands to the corresponding binary commands. The class helps us find the binary values for ***dest*, *comp*,** and ***jump*** commands.

**CODE:**package EOC;  
public class Backend\_Code{

// The binary code for destination is computed - "d1\_d2\_d3"

public static String dest(String mnemonic) throws InvalidDestException

{

String d1 = "0";

String d2 = "0";

String d3 = "0";

//We obtain the Test Cases from the Truth Table.

if (mnemonic.contains("A"))

d1 = "1";

if (mnemonic.contains("D"))

d2 = "1";

if (mnemonic.contains("M"))

d3 = "1";

/\* The following test case is when the "d1\_d2\_d3" is "000" code it throws an exception as it depicts null

destination.

\*If there is no mnemonic, we throw out this exception.

\*/

if ((d1+d2+d3).equals("000") && !mnemonic.equals(""))

{

throw new InvalidDestException();

}

return d1 + d2 + d3;

}

// The Binary code for comp String is computed

public static String comp(String mnemonic) throws InvalidCompException

{

String a = "0";

if (mnemonic.contains("M"))

{ /\* When "M" is present in a command, the value of a = 1 and test cases are changed in such a manner

that instead of "A" in test case, we use "M", Eventhough only few cases are present for "M".

\*/

a = "1";

mnemonic = mnemonic.replace("M", "A");

}

String c = "000000";

//The Switch cases for Mnemonic are obtained from Truth Table

switch (mnemonic)

{

case "0": c = "101010";

break;

case "1": c = "111111";

break;

case "-1": c = "111010";

break;

case "D": c = "001100";

break;

case "A": c = "110000";

break;

case "!D": c = "001101";

break;

case "!A": c = "110001";

break;

case "-D": c = "001111";

break;

case "-A": c = "110011";

break;

case "D+1": c = "011111";

break;

case "A+1": c = "110111";

break;

case "D-1": c = "001110";

break;

case "A-1": c = "110010";

break;

case "D+A": c = "000010";

break;

case "D-A": c = "010011";

break;

case "A-D": c = "000111";

break;

case "D&A": c = "000000";

break;

case "D|A": c = "010101";

break;

default: throw new InvalidCompException();

}

return a + c;

}

// The three bits assigned for Jump mnemonics are listed.

public static String jump(String mnemonic) throws InvalidJumpException

{

switch (mnemonic)

{

case "": return "000";

case "JGT": return "001";

case "JEQ": return "010";

case "JGE": return "011";

case "JLT": return "100";

case "JNE": return "101";

case "JLE": return "110";

case "JMP": return "111";

default: throw new InvalidJumpException();

}

}

// converts a numeric symbol to 15 bit binary

public static String toBinary(String symbol)

{

int value = Integer.*valueOf*(symbol);

String binaryVALUE = Integer.*toBinaryString*(value);

return String.*format*("%1$15s", binaryVALUE).replace(" ", "0");

}

}

**Assemble Class:** This class deals with the concatenation of the binary values returned for the ***dest*,*comp*** and ***jump*** commands. Also, it creates the Hack file enabling us to use this as input for the machine as it finally obtains the machine language and writes it into a file.

**CODE:**package EOC;  
import java.io.\*;  
public class Assemble {

// We Declare the input and output Files.

private String inputFile;

private PrintWriter out;

// Symbol Table for the predefined labels

private Symbols table = new Symbols();

public Assemble(String file) throws IOException

{

// Creating the Output File

inputFile = file;

String outputFile = inputFile.replaceAll("\\..\*", "") + ".hack";

out = new PrintWriter(new FileWriter(outputFile));

/\* Creating a new output file and

Initialize symbol table for checking the labels such as SCREEN and KEYBOARD(KBD)\*/

table.initialize();

}

public void assemble1() throws FileNotFoundException, IOException

{

Parser parser = new Parser(inputFile);

int ramAddress = 0;

String symbol;

while (parser.advance())

{

if (parser.commandType() == 2)

// If the command is a L-instruction

{

symbol = parser.symbol();

if (!table.contains(symbol))

table.addEntry(symbol, ramAddress);

} else

{

ramAddress++;

// print warning when memory is all used

if (ramAddress > 32768)

System.***err***.println("Warning: all ROM is in use");

}

}

parser.close();

return;

}

//We handle the commands and bring about the 16-bit binary codes

public void assemble2() throws FileNotFoundException, IOException

{

Parser parser = new Parser(inputFile);

String dest, comp, jump;

String symbol, value;

// starting address for variables

int ramAddress = 16;

while (parser.advance())

{

try

{

if (parser.commandType() == 3) // If the command is a C-instruction

{

dest = parser.dest();

comp = parser.comp();

jump = parser.jump();

out.println("111" + Backend\_Code.*comp*(comp) + Backend\_Code.*dest*(dest) + Backend\_Code.*jump*(jump));

}

else if (parser.commandType() == 1)

// If the command is a A-instruction

{

symbol = parser.symbol();

if (Character.*isDigit*(symbol.charAt(0)))

{

value = Backend\_Code.*toBinary*(symbol);

}

else if (table.contains(symbol))

{

value = Integer.*toString*(table.getAddress(symbol));

value = Backend\_Code.*toBinary*(value);

}

else

{

// Memory Usage Warnings are printed

if (ramAddress > 16383)

System.***err***.println("Warning: allocating variable in I/O memory map");

if (ramAddress > 24576)

System.***err***.println("Warning: no more RAM left");

table.addEntry(symbol, ramAddress);

value = Backend\_Code.*toBinary*("" + ramAddress);

ramAddress++;

}

out.println("0" + value);

}

}

catch (InvalidDestException ex) {

Error.*error*("Invalid destination", inputFile, parser.lineNumber, parser.currentLine);

}

catch (InvalidCompException ex) {

Error.*error*("Invalid computation", inputFile, parser.lineNumber, parser.currentLine);

}

catch (InvalidJumpException ex) {

Error.*error*("Invalid jump", inputFile, parser.lineNumber, parser.currentLine);

}

}

parser.close();

return;

}

// We close the output .hack file

public void close() throws IOException

{

out.close();

return;

}

}

**Hack\_Assembler Class:** This class deals with the front-end part, that is, it combines all the above classes together and encapsulates into a single unit. The file is accessed by specifying the file name along with the destination and so we would access the asm file and create a hack file and save it in the same folder with the same name as that of the asm file.

**CODE:**package EOC;  
import java.io.\*;  
public class Hack\_Assembler {

public static void main(String[] args)

{

String args1 = "E:\\CSE(AI)\\SEM-2\\Elements of Computing-2\\PROJECT\\CSE332Project\\File\\Project.asm";

// Checking whether the File-location and name is mentioned or not.

if (args1 == null)

Error.*error*("Enter The File Name");

String inputFile = args1;

try {

Assemble assembler = new Assemble(inputFile);

assembler.assemble1();

assembler.assemble2();

assembler.close();

}

catch (FileNotFoundException ex)

{

Error.*error*("File \'" + inputFile + "\' not found!");

}

catch (IOException ex)

{

Error.*error*("An i/o exception has occured");

}

}

}

**Error Class:** We create an error class based on whatever error has been encountered while computing. We extend the Exception class so as to ensure the full running of the code and to avoid unnecessary errors.

**CODE:**package EOC;  
public class Error {

public static void error(String message, String fileName, int lineNum, String line)

{

System.***err***.println(fileName + ":" + lineNum + ": Error: " + message);

System.***err***.println("\t" + line);

System.*exit*(1);

}

public static void error(String message, String fileName, int lineNum)

{

System.***err***.println(fileName + ":" + lineNum + ": Error: " + message);

System.*exit*(1);

}

public static void error(String message)

{

System.***err***.println("Error: " + message);

System.*exit*(1);

}

}

// Exceptions for each case

/\*InvalidDestException is when Destination is null or not given.

\* InvalidCompException is when Computation to binary format gives out error.

\* InvalidJumpException is when Jump command to binary gives error. \*/

class InvalidDestException extends Exception {}

class InvalidCompException extends Exception {}

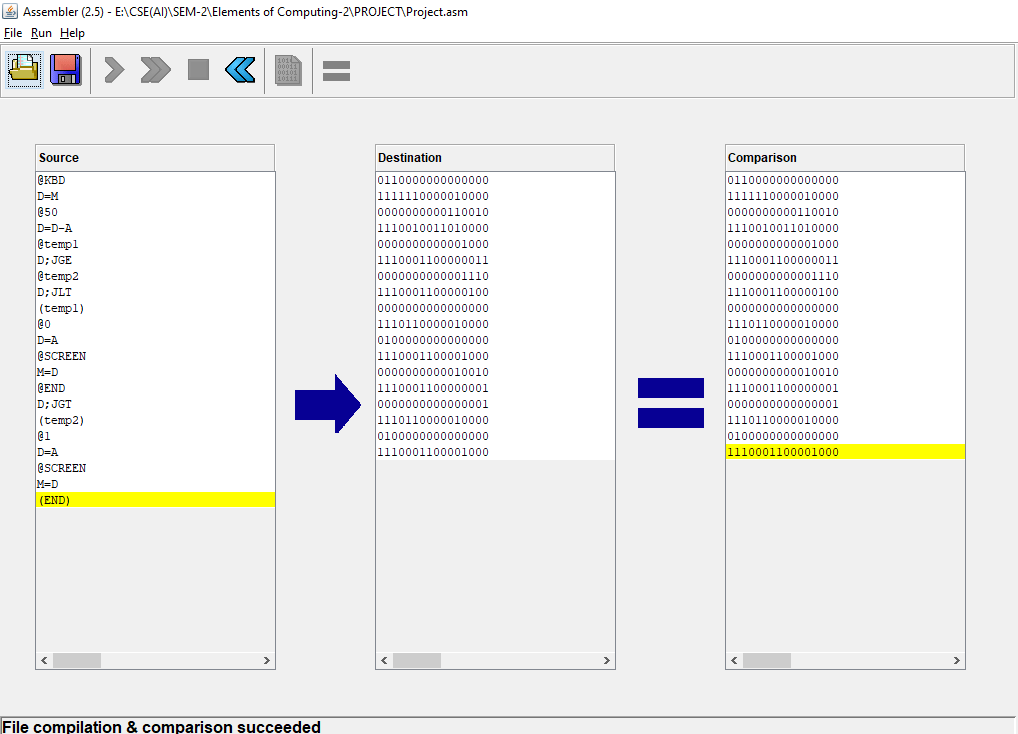
class InvalidJumpException extends Exception {}

**THE OUTPUT FROM THE BUILT ASSEMBLER**

**The machine language output obtained is as follows:**

0110000000000000  
1111110000010000  
0000000000110010  
1110010011010000  
0000000000001000  
1110001100000011  
0000000000001110  
1110001100000100  
0000000000000000  
1110110000010000  
0100000000000000  
1110001100001000  
0000000000010010  
1110001100000001  
0000000000000001  
1110110000010000  
0100000000000000  
1110001100001000

We now compare this machine code with the actual machine code produced by the assembler. This is the snapshot which we got upon comparing both the files:



We can see that the machine codes produced by both the assemblers are same and hence our part-2 has been implemented successfully.

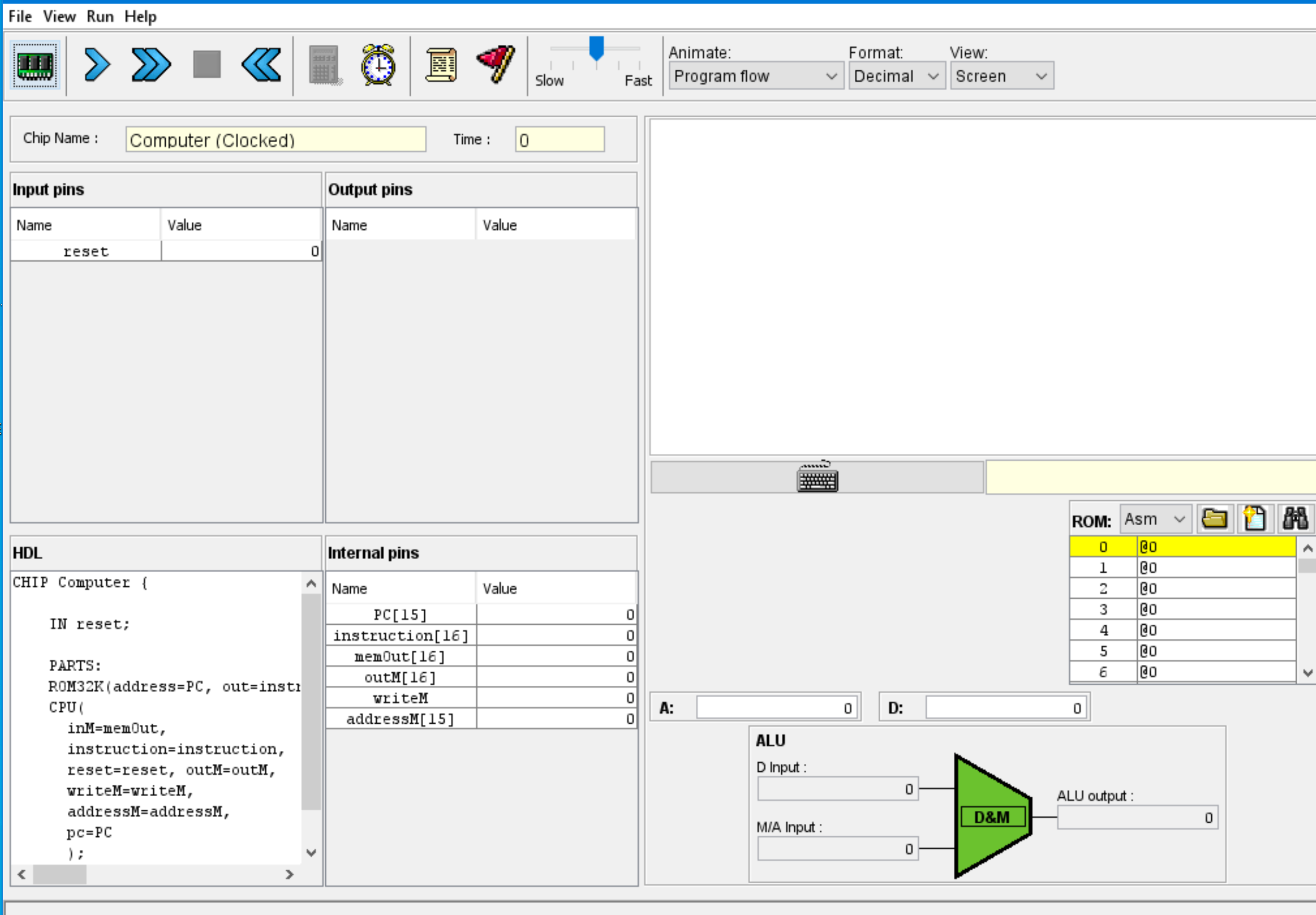
**PART-3**

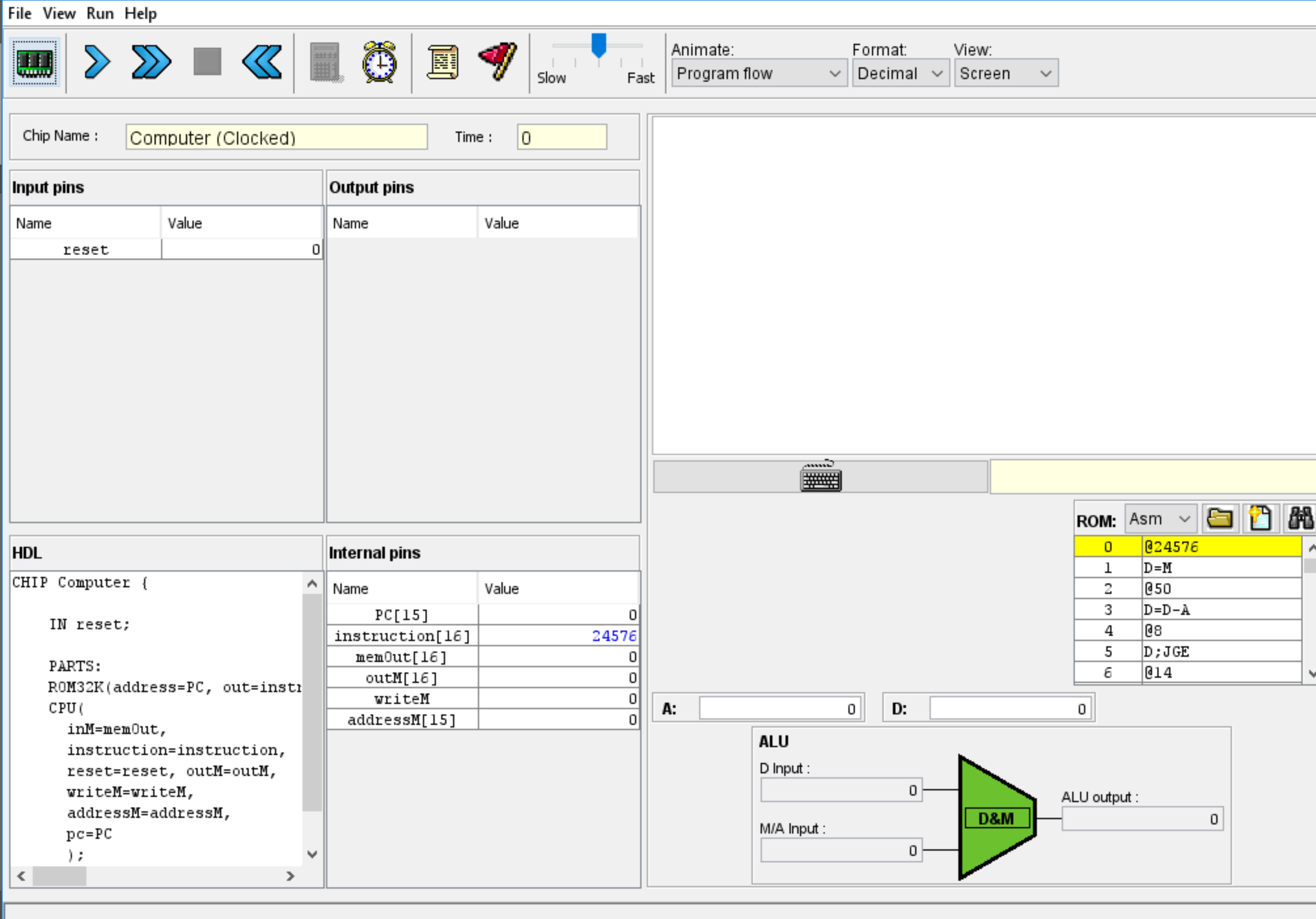
The main HDL files required for this task are:

1. CPU.hdl
2. ROM32K.hdl
3. Memory.hdl

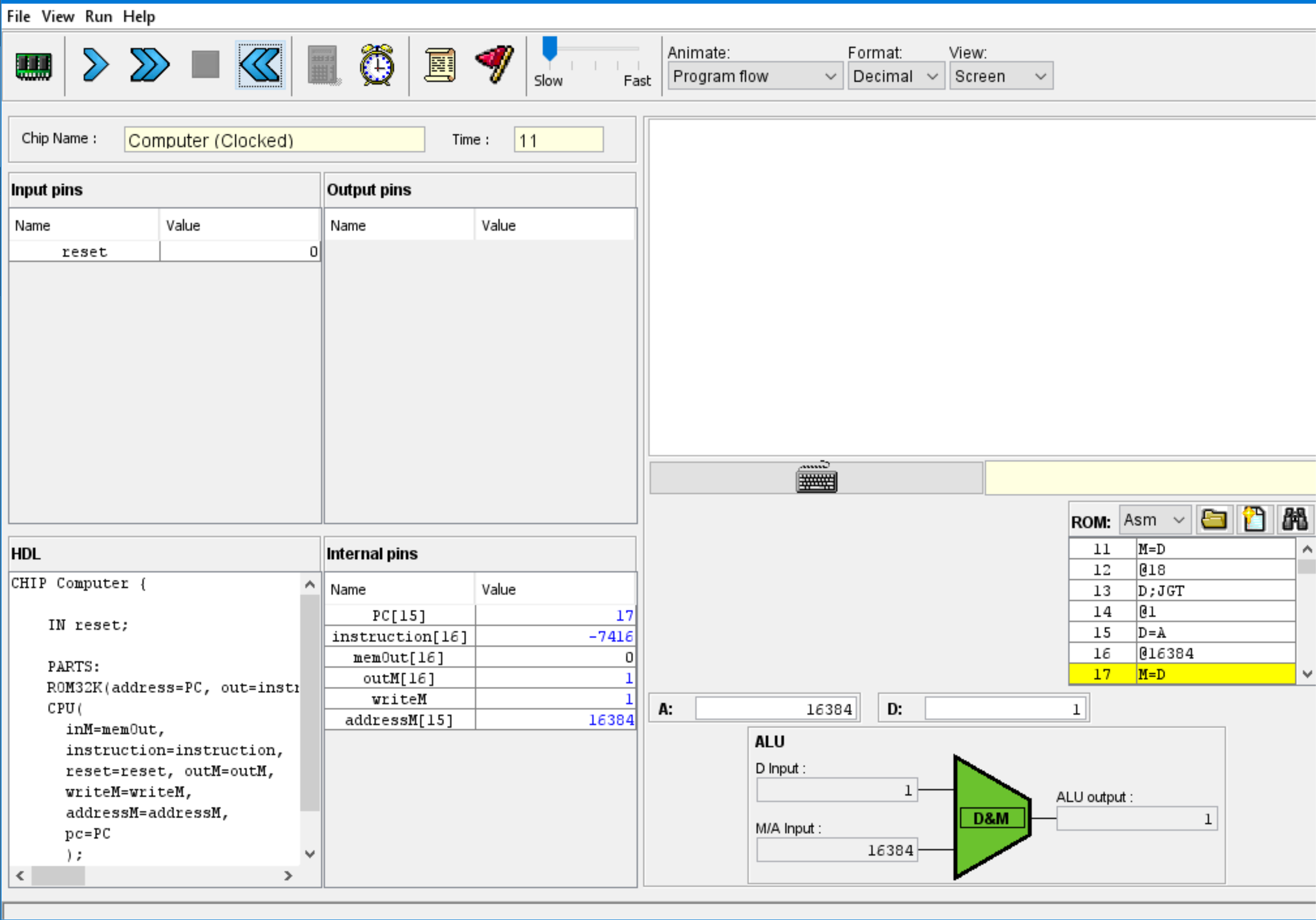
All of these combined makes another .hdl file called the Computer.hdl. This is our virtual computer that we use for the verification of our ASM code.

After loading the Computer.hdl in the Hardware Simulator, there will be ROM generated on the bottom right. Here we can load our project asm code (Project.asm). After this we run the simulator and we can see the progress happening.





Final Output



HDL files:

CPU.hdl

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:

// "decode" which instruction to execute

Or(a=instruction[15], b=false, out=isCInstruction); // if(instruction[15]) isCInstruction=true;

Not(in=instruction[15], out=isAInstruction); // if(!instruction[15]) isAInstruction=true;

And(a=isCInstruction, b=instruction[5], out=isCWriteA); // isCInstruction && dest==A

Or(a=isAInstruction, b=isCWriteA, out=loadA); // writeA if (A-instr || (isCInstruction && dest==A))

Mux16(a=instruction, b=outALU, sel=isCWriteA, out=inAReg);

ARegister(in=inAReg, load=loadA, out=outAreg, out[0..14]=addressM); //outAreg==addressM

And(a=isCInstruction, b=instruction[4], out=loadD); // isCInstruction && dest==D

DRegister(in=outALU, load=loadD, out=outDReg);

Mux16(a=outAreg, b=inM, sel=instruction[12], out=outAorM); // A or M, sel=a (a bit selects A register)

ALU(

x=outDReg,

y=outAorM,

zx=instruction[11], // c1

nx=instruction[10], // c2

zy=instruction[9], // c3

ny=instruction[8], // c4

f =instruction[7], // c5

no=instruction[6], // c6

out=outALU,

out=outM,

zr=isZeroOut,

ng=isNegOut

);

// Set out flags and ProgramCounter

Not(in=isNegOut, out=isNonNeg);

Not(in=isZeroOut, out=isNonZero);

And(a=isNonNeg, b=isNonZero, out=isPositive); // is positive? !zero && !negative

And(a=isCInstruction, b=instruction[3], out=writeM); // write M? isCInstruction && dest==M

And(a=isPositive, b=instruction[0], out=JGT);

And(a=isZeroOut, b=instruction[1], out=JEQ);

And(a=isNegOut, b=instruction[2], out=JLT);

Or(a=JEQ, b=JLT, out=JLE);

// is jump?

Or(a=JLE, b=JGT, out=jumpToA);

And(a=isCInstruction, b=jumpToA, out=loadPC); // loadPC? if(isCInstruction && jump)

Not(in=loadPC, out=PCinc);

PC(in=outAreg, inc=PCinc, load=loadPC, reset=reset, out[0..14]=pc);

}

Memory.hdl

CHIP Memory {

IN in[16], load, address[15];

OUT out[16];

PARTS:

DMux4Way(in=load, sel=address[13..14], a=selRam1, b=selRam2, c=selScreen, d=selKbd);

Or(a=selRam1, b=selRam2, out=selRam);

RAM16K(in=in, load=selRam, address=address[0..13], out=outRam); // RAM

Screen(in=in, load=selScreen, address=address[0..12], out=outScreen); // Screen

Keyboard(out=outKbd); // KBD - we don't ACTUALLY write in RAM..

// select what to output

Mux4Way16(a=outRam, b=outRam, c=outScreen, d=outKbd, sel=address[13..14], out=out);

}

ROM32K.hdl

CHIP ROM32K {

IN address[15];

OUT out[16];

BUILTIN ROM32K;

}

Keyboard.hdlCHIP

Keyboard {

OUT out[16]; // The ASCII code of the pressed key,

// or 0 if no key is currently pressed,

BUILTIN Keyboard;

}

Screen.hdl

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;

}

PC.hdl

CHIP PC {

IN in[16],load,inc,reset;

OUT out[16];

PARTS:

// increment the output of the register

Inc16(in = feedback, out = pc);

// "Sequential chips always consist of a layer of DFFs sandwiched

Mux16(a = feedback, b = pc, sel = inc, out = w0);

Mux16(a = w0, b = in, sel = load, out = w1);

Mux16(a = w1, b = false, sel = reset, out = cout);

// the output from the register also needs to get fed back through

// the combinational logic to get processed for the next clock cycle.

Register(in = cout, load = true, out = out, out = feedback);

}

Computer.hdl

CHIP Computer {

IN reset;

PARTS:

ROM32K(address=PC, out=instruction);

CPU(

inM=memOut,

instruction=instruction,

reset=reset, outM=outM,

writeM=writeM,

addressM=addressM,

pc=PC

);

Memory(in=outM, load=writeM, address=addressM, out=memOut);

}