The Design and Implementation of the Hack Computer

Harith Al-Safi [el20hzaa, 201416467]

ELEC2665 Microprocessors and Programmable Logic

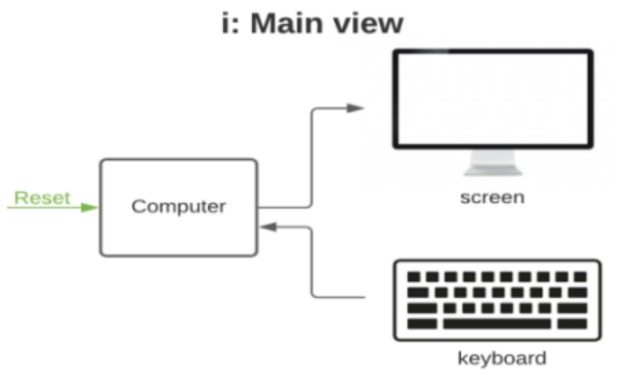
Academic Integrity

Table of Contents

Academic Integrity 1
Chapter 1 - The Hack Computer Architecture 3
Main block diagram........ 3
Detailed diagram of other CPU components (B) 4
Detailed diagram of RAM (C) 4
Chapter 2 – Software Execution 5
Code functionality........ 5
Assembler process 5
Initialization 5
First pass........ 6
Second pas........ 6
Hardware process 7
Architecture overview........ 7
Instruction execution 7
Chapter 3 – Reflection........ 9
Reference 10
Appendix 11
Computer.hdl........11
Memory.hdl 11
CPU.hdl........12
PC.hdl 14
ALU.hdl 15

Chapter 1 - The Hack Computer Architecture

Main block diagram



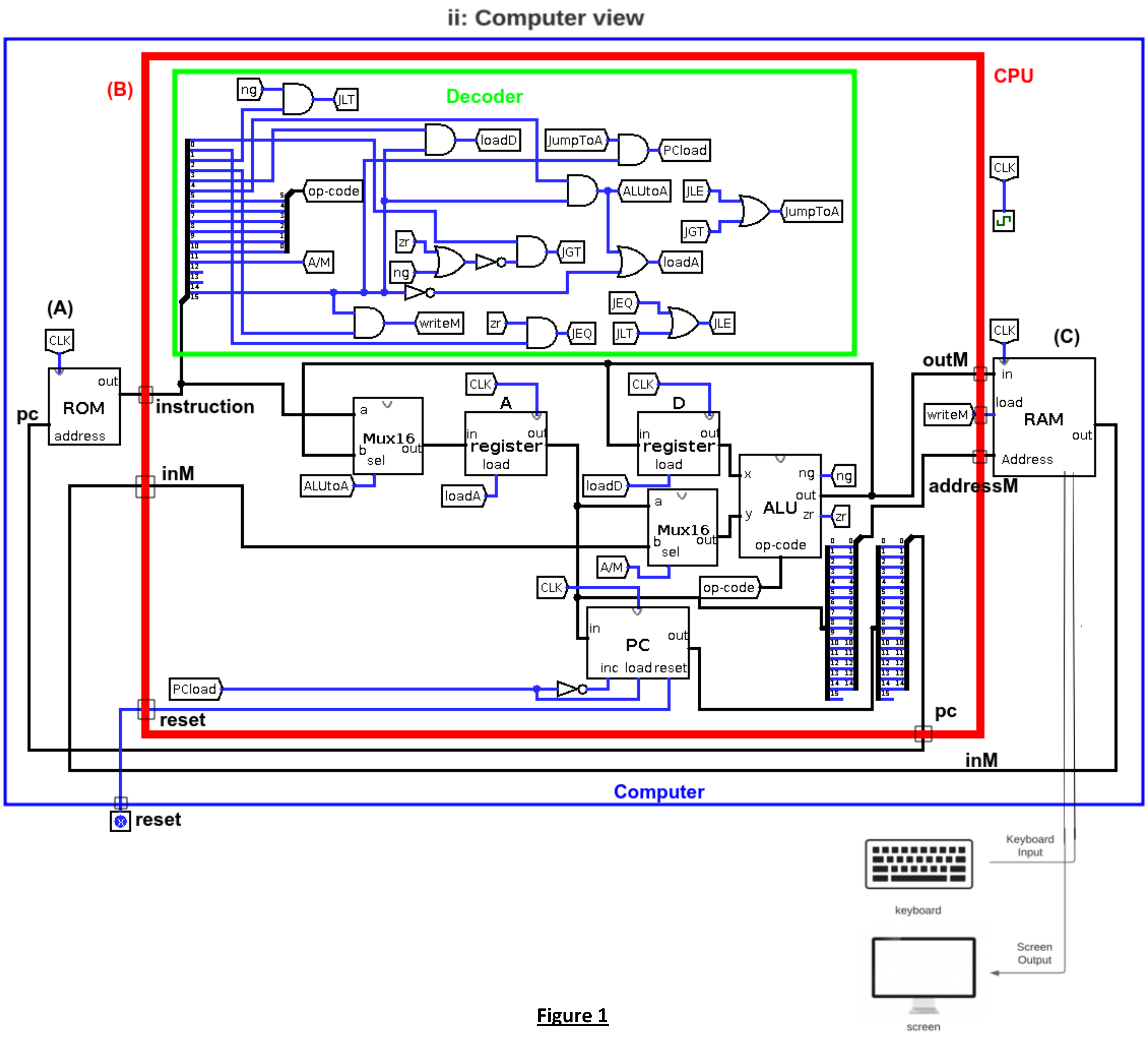


Figure 1

Detailed diagram of other CPU components (B)

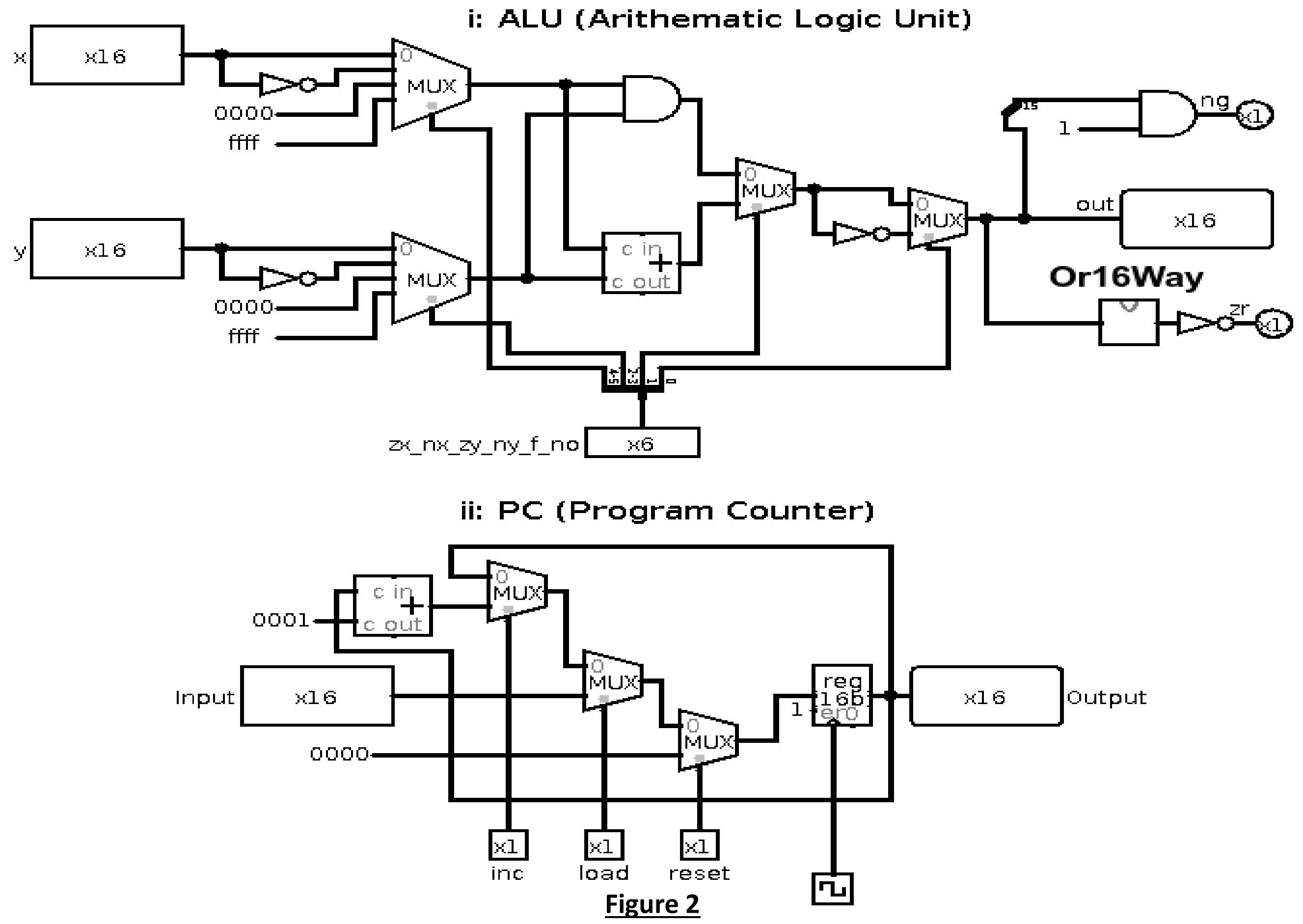


Figure 2

Detailed diagram of RAM (C)

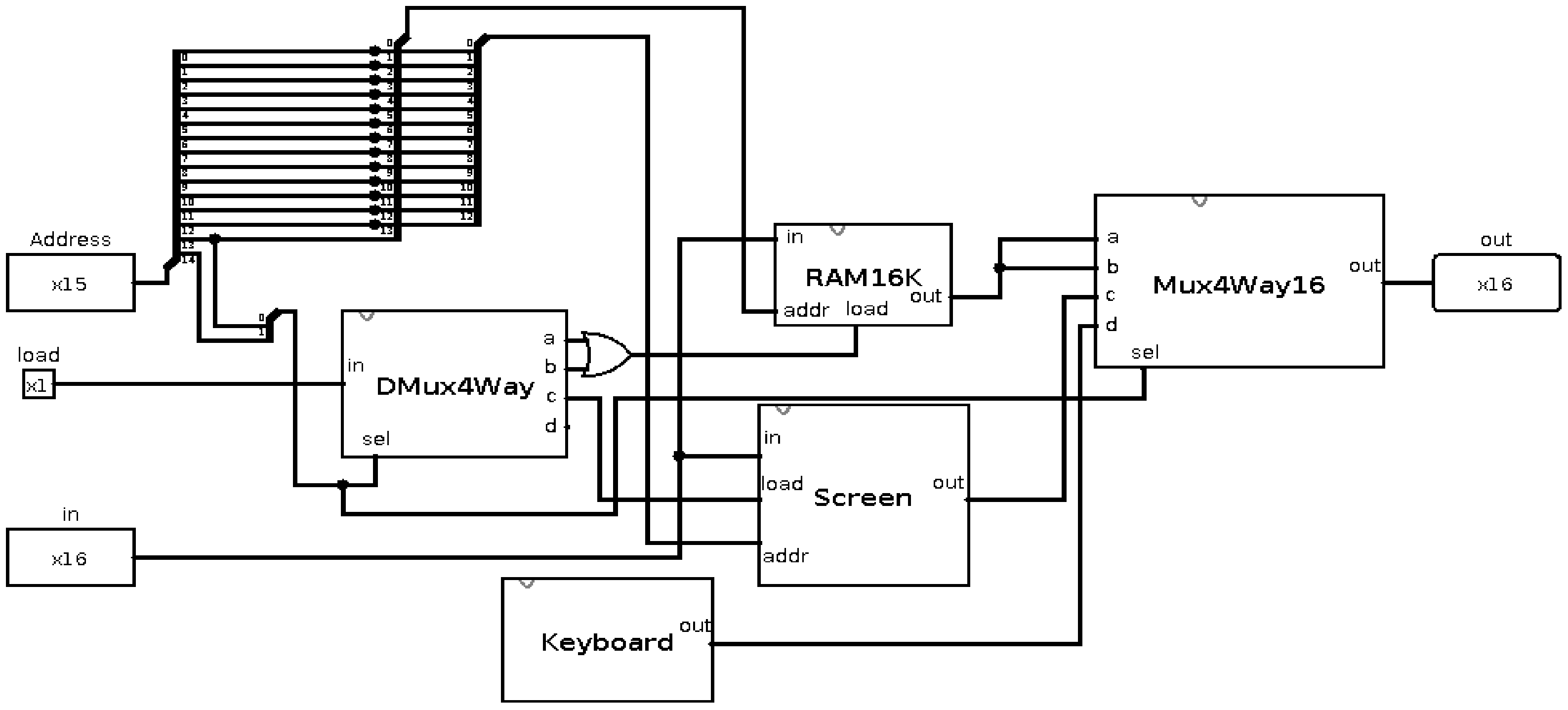
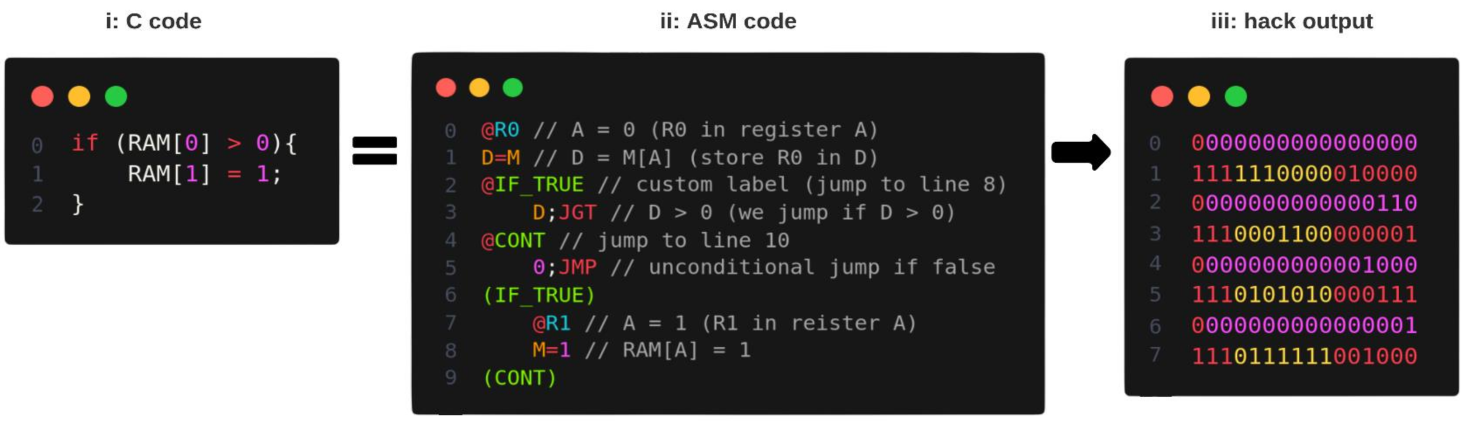


Figure 3

Chapter 2 – Software Execution

Code functionality



i: C code

```
if (RAM[0] > 0){
    RAM[1] = 1;
}
```

ii: ASM code

```
0 @R0 // A = 0 (R0 in register A)
1 D=M // D = M[A] (store R0 in D)
2 @IF_TRUE // custom label (jump to line 8)
3     D;JGT // D > 0 (we jump if D > 0)
4 @CONT // jump to line 10
5     0;JMP // unconditional jump if false
6 (IF_TRUE)
7     @R1 // A = 1 (R1 in reister A)
8     M=1 // RAM[A] = 1
9 (CONT)
```

iii: hack output

```
0 0000000000000000
1 1111110000010000
2 0000000000000110
3 1110001100000001
4 0000000000001000
5 1110101010000111
6 0000000000000001
7 1110111111001000
```

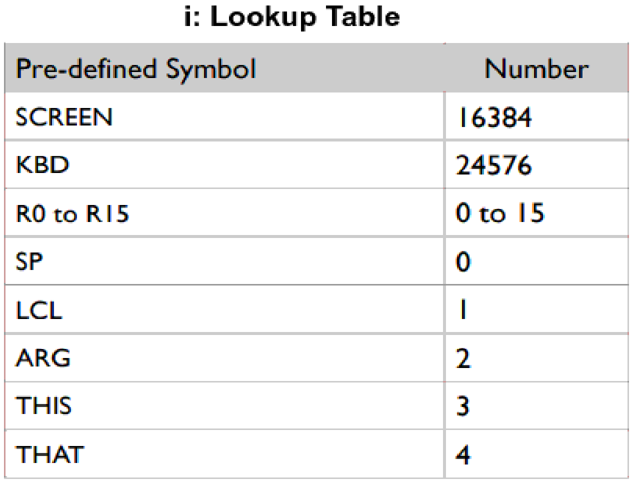
Figure 4

Figure 4.ii, shows the assembly code with its equivalent C code (Figure 4.i) and the assembled binary output (Figure 4.iii). The code is simple as it checks the data in address 0 of the Random Access Memory (RAM), and if its greater than 0 we assign the value of RAM address 1 to 1. The main idea of the code is to show the combination of A-instructions and C-instructions in order to produce a functional logic.

Assembler process

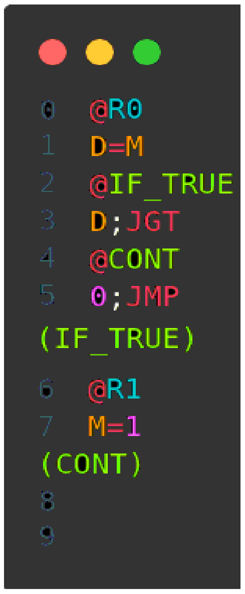
Initialization

i: Lookup Table



Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

ii: new code

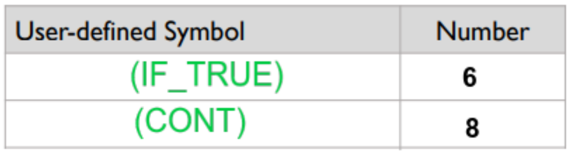


```
0 @R0
1 D=M
2 @IF_TRUE
3 D;JGT
4 @CONT
5 0;JMP
(IF_TRUE)
6 @R1
7 M=1
(CONT)
8
9
```

Figure 5

The main idea of the assembler is to convert our assembly code into machine code such that it will be loaded into our Read Only Memory (ROM) in order to be fetched and executed. Figure 5 shows the initialization process of the assembler. We start by generating a look up table as shown in Figure 5.i, this table will contain all the pre-defined symbols with the corresponding value that they refer to and usually that is related to the address. For example, **R0** indicates a value of 0, which refers to RAM address 0. Figure 5.ii shows the code after eliminating the white space and comments. Any used label throughout the code becomes without a line number since the label will be referring to the line after it.

First pass

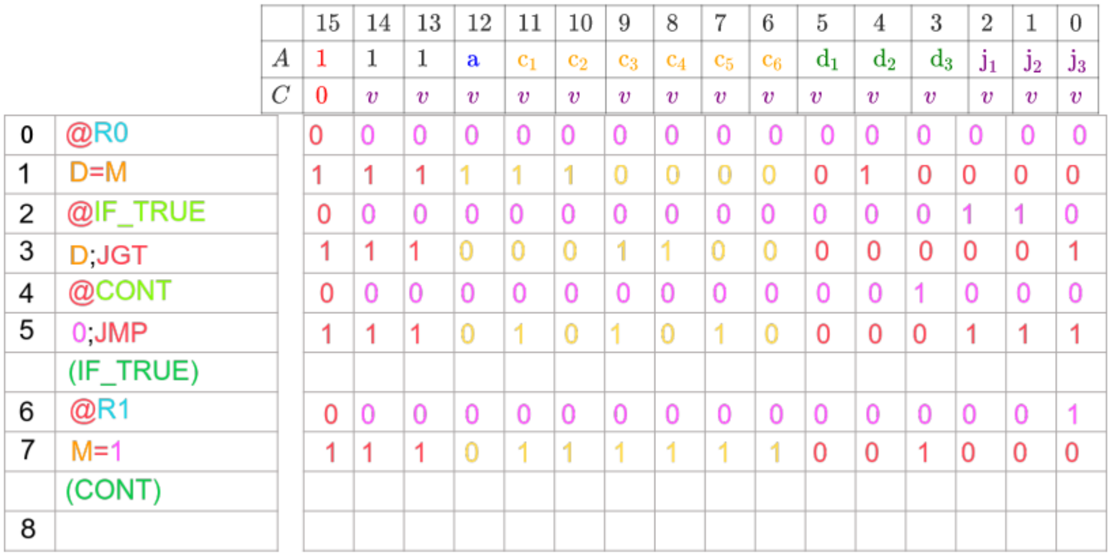


User-defined Symbol	Number
(IF_TRUE)	6
(CONT)	8

Figure 6

Figure 6, shows the first pass process of the assembler such that we go through the source code line by line and for each label declaration we add it to the look up table with the corresponding line number that it refers to. For example **(IF_TRUE)** is declared in line 6 in Figure 4.ii, however after the initialization process the label becomes without a line number as shown in Figure 5.ii and it refers to the line after it which is line 6.

Second pas



		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	A	1	1	1	a	c_1	c_2	c_3	c_4	c_5	c_6	d_1	d_2	d_3	j_1	j_2	j_3
	C	0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
0	@R0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	D=M	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0
2	@IF_TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
3	D;JGT	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	1
4	@CONT	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
5	0;JMP	1	1	1	0	1	0	1	0	1	0	0	0	0	1	1	1
	(IF_TRUE)																
6	@R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
7	M=1	1	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0
	(CONT)																
8																	

Figure 7

Figure 7 shows the second pass of the assembler process, we go through the source code, line by line and we translate the A (15th bit = 0) and C (15th bit =1) instructions to binary code according to their format. If we used **@symbol** we go through the look up table and replace the it with the corresponding value.

Translating A instruction:

Line 6 in Figure 7 shows an A instruction, as **@R1** will correspond to the value of **R1** in binary format being loaded into the A register. Thus, we will have a 0 in the 15th bit followed by the 15-bit binary format of **1** as this is what **R1** refers to in the look up table in Figure 6. Thus, we will get **0000000000000001**.

Translating C instruction:

Line 1 in Figure 7 shows a C instruction as **D=M** corresponds to assigning the value of D-register to the value of the RAM as the RAM address is loaded in the A-register and in our case its address **R0** (address 0). On that note since it's a C-instruction, the 15th bit will be 1, the 14th and 13th bits are also 1. Since we are using the RAM (M) instead of A register in our operation, the 12th bit will be 1. Following this we begin to assess the original statement and find the correct ALU opcode that corresponds with the expression, thus bits 11 to 6 will correspond to the ALU opcode. We only have **=M** as our expression and according to the Arithmetic Logic Unit (ALU) this translates to an opcode of 110000. Bits 5 to 3 are related to the destination bits and in our case the destination is register-D. According to our Central Processing Unit (CPU) this functionality corelates to an instruction of 010. Bits 2 to 0 corelates to the jump bits and since we don't have any jump in our expression, thus this functionality will have an instruction of 000. The final instruction will be **1111110000010000**.

Hardware process

Architecture overview

To begin with the hack computer is used to do simple instruction sets regarding storing, loading and manipulating 16-bit data. The architecture starts with our binary instruction sets being loaded into our 32KiloByte ROM, which is shown in Figure 1.ii.A, each line of our code will be stored in a separate address in the ROM. The program counter (PC) located in the CPU In Figure 1.ii.B is in control of fetching the next instruction or a certain instruction to the CPU according to the current loaded instruction set. The PC uses the ROM instruction address in order to control its functionality. The 24 Kilobyte RAM in Figure 1.ii.C is responsible for data storage and retrieval. Both the RAM and the ROM contain similar architecture as they consist of multiple registers whereas the address system is controlled by a demultiplexer and a multiplexer. However, the RAM is more complex since it contains the memory map for the screen, the keyboard and the memory itself. Figure 3 shows how the RAM is implemented; we can see that the address is divided into sections such that a set of bits is used to control the parts of the RAM. In our case the 13th and 14th bits are used as selection bits for the 4-way demultiplexer in order to choose between RAM1, RAM2, Screen and keyboard. RAM1 and RAM2 are considered the same memory that contains the data however they are just divided in order to be used within the demultiplexer. Thus addresses 0 to 16383 is used for the memory, addresses 16384 to 24575 are used for the screen and address 24576 is used for the keyboard.

Moving on to the CPU, we can see it's the most complex part as it contains the heart of the functionalities. The CPU mainly consists of multiplexers, registers, the PC, the ALU and the decoder. Having the ability to do multiple instructions is implemented in the decoder as we use a set of OR and AND gates within the instruction bits in order to decode them into sending the correct signals to the other CPU components. The ALU does most of the computation regarding arithmetic's and logical expressions. Its implementation is shown in Figure 2.i, we can see that the opcode is divided into multiple selection bit inputs for multiplexers across the circuit. We also use a 16-bit Adder to do the arithmetic computation since we could negative or zero any of the inputs, thus we can do addition, subtraction or only assignment operations. This is also used in logical expressions however an AND gate is used instead. The ALU also contain flags that check if the output is negative or zero such that this will be used in jump statements later. The PC is shown in Figure 2.ii, as its implementation is simple, containing an adder and some multiplexers. The adder will be used with one of the inputs fixed at 1 such that it increments the other input by 1. The load, reset and increment bits are used as selection bits to provide the functionality for the PC. The PC will be used in conjunction with the decoder such that it either loads a new value or increments the previous one as this value will be the instruction address.

Instruction execution

We can start by assessing line 0 in Figure 7 as it provides a simple A-instruction. We will begin with the procedure from the ROM. The CPU starts by fetching the first line of the instruction set and in our case its line 0 which should be loaded in ROM address 0. The instruction set is divided into single bits and passed into the decoder which is shown in Figure 1.ii. As mentioned earlier the 15th bit is used with the decoder to differentiate between A and C instruction. In our case the 15th bit will be 0. From figure 1.ii, *ALUtoA* will be 0 meaning that we won't take input from the ALU as we will take the instruction as the input to register A. *loadA* will be 1 meaning that we load the provided instruction (which translates to a value of 0) into the A-register and then it will move into the address output which is *addressM* and it will be provided to the RAM as an address input. *writeM* will be 0 meaning that we won't write anything to the RAM. *PCload* will also be 0 meaning that we won't load any new instruction into the PC. On the other hand, the increment bit will be 1 meaning that we will increment and move on to the next instruction address which is address 1. The output *pc* will be linked to the ROM address.

As now we are on line 1 in Figure 7 which relates to address 1 in the ROM with a C-instruction. 15th, 14th, and 13th bit are all 1. Moving on to the decoder flags, *ALUtoA* will be 0 providing the same previous functionality. *loadA* will be 1, meaning that we won't load anything into register A. The 12th bit (*A/M* flag) choses between register A or the RAM as our 2nd input to the ALU and since its 1, the previous RAM output will be our input to the ALU which is *inM*. *loadD* uses the 4th bit in order to make sure we load the D register, on that note it will be 1 meaning that we will load the output of the ALU to register D. Moving on the opcode will be 110000 which translates to assigning the value of the y input to the output of the ALU. Thus, the ALU output will be what was stored in the RAM. *PCload* in addition to all jump flags will be 0. We will not load anything to the PC, instead we will increment the current value to 2. *writeM* will be 0 thus we will not write anything to the RAM, as the A register will stay the same.

Now we will be on address 2 in the ROM which relates to line 2 in Figure 7 with an A-instruction. Thus, we will follow the same procedure as we did in line 0. However, what is different in this case is that we declared a label. Since we already had a look up table in addition to our compiled outputs this label refers to line 6 in Figure 7. Thus, instead of loading a 0 into our A-register we will load a value of 6. Afterwards we will increment the PC and move to address 3.

As we are on address 3 for line 3 in Figure 7, we can see that it's a C-instruction, which follows the same procedure as line 1 for the 15th,14th,13th bit. Since we don't have an assignment operator in the assembly code neither we have a destination bit in our instruction, thus *loadD* will be 0. On that note *loadA* will also be 0 since it's a C-instruction. Moving on, *A/M* is 0 thus we will use the A-register as our input to the ALU in addition to the D register as our 2nd input. The A register will be from line 2 as for the D-register it will be from line 1. The opcode 001100 means that the ALU output will stay at x which is the D-register. Thus, the flags that specify the jump will be related to it. Since we didn't have anything stored in our RAM, the jump condition will not be fulfilled so we move onwards to address 4 which is line 4. However, if we had a value in our RAM that gives out both *ng* and *zr* flags as 0, we will have a jump condition. Thus, *JGT* flag will be 1 which makes *JumpToA* flag as 1 so we will end up with *PCload* being 1. So, the PC will have a load bit as 1, increment bit as 0 and the input will be from the A register. Thus, the output pc will load into our ROM as the value in A-register.

Line 4 is an A-instruction exactly similar to line 2 as we also use a label. However, this label has a value of 8, meaning that we will have a value of 8 stored in our A-register. Afterwards we will move to line 5.

Line 5 is a C-instruction with a jump. This means that it's similar to line 3, however we will use a certain jump rather than a conditional one in this case. The opcode is 101010, this translates to having an ALU output of 0 regardless of what the input is. This will make the *JLT* flag 1, which makes *JumpToA* and *PCload* 1. This will load the PC with the previous A-register value which is 8 in our case. Thus, we jump to instruction 8 which is nothing.

Line 6 occurs only if the jump on address 3 was fulfilled. Line 6 is an A-instruction which is quite similar to line 0. Instead of loading 0, we load 1 into the A-register which will be the RAM address for out next instruction

Line 7 also only occurs if the jump on line 3 was fulfilled. Line 7 is a C-instruction similar to line 1. What is different is that we will be writing to the RAM, as *writeM* is 1. The ALU inputs will be the D-register and A-register since *A/M* is 0. As the opcode is 111111, the ALU output will be 1 regardless of what is in both registers. This means that *outM* will be 1, the address will be the one stored in the A-register, and since we have a load bit of 1, we will write to the memory with a value of 1. If we had more lines after line 8, we would continue executing them wether the jump in line 3 was fulfilled or not.

Chapter 3 – Reflection

All in all, this procedure helped me understand the computer architecture in a much more advanced way. Since we coded the components using hardware defined language, this means that we were able to understand every single component of our hack computer. Since, we made it using NAND gates, which we also designed using transistors, this made us responsible for the design of fundamental gates too such as AND and OR gates. Moving on, we started by understanding the assembly language which helped us to see how mnemonics ae used at the lowest level in order to be translated to machine code. On that note we took how certain functionality such as if statements and while loops are implemented in assembly. They need more lines of codes compared to high-level programming and this gave us the ability to manipulate the tiniest detail in our computer such as accessing the registers. Afterwards we started learning how the assembler takes the assembly code and outputs the binary version of it. Using the software provided by nand2tetris, we were able to assess our HDL files in addition to our assembly code in order to make sure its functional and working. From what we learned I was also able to implement the hack computer in Logisim such that it was fully working which made understanding the computer simpler since I was able to test and try different instructions. The whole process made me change my perspective on day to day computers. This made me think on how complex would be the assembly language of advanced computers such as the x86 architecture. Since we also have more components such as network card, GPU and many more. This made me curious on researching more about the x86 architecture and how would the assembly code convert to binary within such complex hardware.

Reference

- N. Nisan and S. Schocken, The Elements of Computing Systems, Cambridge: The MIT Presss, 2021
- N. Nisan and S. Schocken, “From Nand to Tetris”, nand2tetris.org, 2017. [Online]. Available: https://www.nand2tetris.org/ [Accessed: Dec. 26, 2021]

Appendix

Computer.hdl

```
CHIP Computer {

   IN reset;

   PARTS:

   CPU(inM=Memout, instruction=instruction, reset=reset,

   outM=outM, writeM=writeM, addressM=addressM, pc=PCout);

   ROM32K(address=PCout, out=instruction);

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

}
```

Memory.hdl

```
CHIP Memory {

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

   OUT out[16];

   PARTS:

   // DIVIDING THE INPUT TO SEGMENTS

   // 00 - address must be less than 8192 – RAM1

   // 01 - address must be less than 16384 – RAM2

   // 10 – address must be greater or equal to 16384 – Screen

   // 11 – address must be greater or equal to 24576 - Keyboard

           DMux4Way(in=load, sel=address[13..14],

   a=loadram1, b=loadram2, c=loadscreen, d=loadkbd);


   // RAM1 is same as RAM2

   // any of the RAM's selected will get loadram=1

           Or(a=loadram1, b=loadram2, out=loadram);


   // getting ram output

   RAM16K(in=in, load=loadram, address=address[0..13], out=ramout);


   // getting screen output

   Screen(in=in, load=loadscreen, address=address[0..12], out=scrout);
   Keyboard(out=kbout);


   // Chosing the final output

   Mux4Way16(a=ramout, b=ramout, c=scrout, d=kbout,

   sel=address[13..14], out=out);


}
```

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:

   // DECODER

   // aSignal -> 1 -> A-instruction

   // aSignal -> 0 -> C-instruction

   Not(in=instruction[15], out=aSignal);


   // cSignal -> 1 -> C-instruction

   // cSignal -> 0 -> A-instruction

   Not(in=aSignal, out=cSignal);


   // load D
```

```
And(a=cSignal, b=instruction[4], out=loadD);

// C-instruction AND destination to A-register
And(a=cSignal, b=instruction[5], out=ALUtoA);

// load A (a-instruction OR c-instruction + destination A)
Or(a=aSignal, b=ALUtoA, out=loadA);

// input to a
Mux16(a=instruction, b=ALUout, sel=ALUtoA, out=toA);
ARegister(in=toA, load=loadA, out=outA);

// ALU inputs
Mux16(a=outA, b=inM, sel=instruction[12], out=ALUy); // A or M
DRegister(in=ALUout, load=loadD, out=ALUx);

// ALU
ALU(x=ALUx, y=ALUy,
zx=instruction[11],
nx=instruction[10],
zy=instruction[9],
ny=instruction[8],
f=instruction[7],
no=instruction[6], out=ALUout, zr=ALUzrOut, ng=ALUngOut);

// Memory outputs
Or16(a=false, b=ALUout, out=outM);
Or16(a=false, b=outA, out[0..14]=addressM);
And(a=cSignal, b=instruction[3], out=writeM);

// jumps
// ALUout == 0 (JEQ)
```

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


   // ALUout < 0 (JLT)

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


   // <= 0

   Or(a=ALUngOut, b=ALUzrOut, out=ZeroOrNeg);

   // > 0

   Not(in=ZeroOrNeg, out=positive);

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


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

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


   // Only jump if C instruction

   And(a=cSignal, b=jumpToA, out=PCload);

   // only inc if not load

   Not(in=PCload, out=PCinc);

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

}
```

PC.hdl

```
CHIP PC {

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

   OUT out[16];

   PARTS:

   Inc16(in=or, out=ori);

   Mux16(a=or, b=ori, sel=inc, out=m1);

   Mux16(a=m1, b=in, sel=load, out=m2);

   Mux16(a=m2, b=false, sel=reset, out=m3);

   Register(in=m3, load=true, out=out, out=or);

}
```

ALU.hdl

```
CHIP ALU {

  IN

    x[16], y[16],  // 16-bit inputs

    zx, // zero the x input?

    nx, // negate the x input?

    zy, // zero the y input?

    ny, // negate the y input?

    f,  // compute out = x + y (if 1) or x & y (if 0)

    no; // negate the out output?


  OUT

    out[16], // 16-bit output

    zr, // 1 if (out == 0), 0 otherwise

    ng; // 1 if (out < 0),  0 otherwise


  PARTS:

    // zeroing

    Mux16(a=x, b[0..15]=false, sel=zx, out=outx1);

    Mux16(a=y, b[0..15]=false, sel=zy, out=outy1);


    // negating

    Not16(in=outy1, out=noty);

    Not16(in=outx1, out=notx);

    Mux16(a=outx1, b=notx, sel=nx, out=outx2);

    Mux16(a=outy1, b=noty, sel=ny, out=outy2);


    // addition and bit wise and

    And16(a=outx2, b=outy2, out=outand);

    Add16(a=outx2, b=outy2, out=outadd);

    Mux16(a=outand, b=outadd, sel=f, out=out3);
    // negation

    Not16(in=out3, out=notout3);

    Mux16(a=out3, b=notout3, sel=no, out=out4);


    // checking if all zeros

    Or16Way(in=out4, out=out5);

    Not(in=out5, out=zr);


    // checking negative

    And16(a[0..15]=true,b=out4,out[15]=ng,out[0..14]=drop);


    // output

    Or16(a=out4,b[0..15]=false,out=out);


}
```