Madison Area Technical College

School of Science, Technology, Engineering and Mathematics

Department of Electrical Engineering Technology

16-bit Microcomputer in a FPGA

Ryan November

ELECENG 10-662-801

Honors – Electrical Engineering

Prepared for Professor Grant Emmel

August 1, 2024

Abstract: This project uses the course *Nand to Tetris* and its corresponding book, *The Elements of Computing Systems* as a framework to develop an understanding of basic computer architecture principles. *Nand to Tetris* lays out the design for a simple microcomputer called “Hack” over the course of 6 hardware simulation projects. These projects cover the construction of a virtual computer capable of running simple programs. The following report details the completion of these 6 projects in a virtual simulation environment and elaborates on the VHDL design and physical FPGA implementation of a 16-bit microcomputer based on the Hack architecture. The result is a microcomputer that can be connected to peripheral devices such as an external monitor and is capable of executing programs written by the user.

Table of Contents

Figures and Tables

1.0 Introduction

The course *Nand to Tetris* and corresponding book *The Elements of Computing Systems – Building a modern computer from first principles* outline the construction of a basic computer architecture called the "Hack" computer. The course uses layers of abstraction to implement the computer from the ground up.

Logic gates are the lowest level building blocks used in digital systems. However, the construction of a computer from discrete logic gates would be unnecessarily complicated; thousands of gates would be needed to construct a simple computing system. Instead, the *Nand to Tetris* course uses these gates to construct slightly more complex devices at which point the gates can be "abstracted" away. The devices can then be analyzed by their behavior, not what they are made of. In other words, the device becomes a box which takes in some input data and outputs some other data in a predictable manner. The contents of the box is not important, all that matters is how the box manipulates some input data to produce the desired output. These slightly more complicated devices can be connected to create even more complex circuits at which point, they too can be abstracted away. Over the course of several projects this process is repeated, each time resulting in increasingly complex devices until they can be put together to form the computer itself. The most basic components of the computer are created directly with NAND gates and upon completion the platform can run programs such as Tetris, hence the name *Nand to Tetris.*

This project involves two major parts: 1. Completion of projects 1-6 of the *Nand to Tetris* course using HDL (hardware description language), a language created by the authors of the course. These projects are all completed in web-based simulation environment provided as part of the course. 2. Implementation of a 16-bit microcomputer based on the Hack architecture using VHDL (very high-speed integrated circuit hardware description language) and running it in a FPGA (field-programmable gate array).

The goal of this project is to build on the foundational knowledge provided in the Digital Circuit Principles and Digital Circuit Design courses to develop an understanding of how computers are constructed, using the Hack architecture as a guide. This project also seeks to develop a stronger understanding of VHDL, and further explore the capabilities of the FPGA as a prototyping platform.

2.0 Nand to Tetris Project Overview

The *Nand to Tetris* course is divided into two parts; part-one focuses on hardware and part-two on software. Part-one consists of 6 projects that develop the hardware platform of the Hack computer; this project is focused on part-one and the hardware design of the Hack platform.

The 6 projects focus on constructing progressively more complicated devices, until the Hack computer is developed in project 5. Projects 1 and 2 focus on combinational logic devices. Project 3 introduces sequential logic concepts. Project 4 outlines a basic assembly language for

the Hack platform. Project 5 utilizes components created in projects 1, 2 and 3 to create the Hack computer. Project 6 implements a program to translate assembly instructions into binary machine language instructions that can be executed by the Hack computer.

2.1 Project 1 – Boolean Logic

2.1.1 Objective

Using the NAND gate several other logic gates, bitwise versions of these gates and several variations of multiplexers and demultiplexers must be created. These can be constructed using a combination of NAND gates and the other gates created along the way.

2.1.2 Background

Boolean logic and digital circuits utilize two data states to process information: logical high and logical low. These states or "logic levels" are typically represented numerically as 1 and 0, respectively. Logic gates and logical devices perform logical operations on input data and return a predictable output. The behavior of a given device can be determined by examining its truth table. A truth table shows the expected output of a device for all possible input conditions. Figure 1 shows the truth table for a NAND gate with two inputs a, b.

NAND

a	b	output
0	0	1
0	1	1
1	0	1
1	1	0

Figure 1

Logic gates themselves are made up of transistors, connected in specific ways to achieve the desired logical operation. However, for digital design, the gates themselves are typically used as the lowest level of abstraction with which more complicated devices can be built. The NAND gate is considered a fundamental logic gate because all other logic gates can be constructed using only a combination of NAND gates. The gates and devices constructed in projects one and two are all combinational logic devices, meaning they operate without the need for a clock and that a change in their inputs directly relates to a change in output.

2.1.2.1 Gates

The basic gates required to create the hack computer are NAND, NOT, AND, OR, and XOR. These basic gates each have two inputs and a single output and perform their respective logical operation.

2.1.2.2 Bitwise and Multiple-Input Gates

Some of the basic gates also require a bitwise version that performs their logical operation bit-for-bit on a longer binary number. The Hack computer is a 16-bit system, meaning it performs operations using "words" that are 16 bits in length. A 16-bit bitwise gate would have 16-bit inputs and outputs. For example, a 16-bit bitwise NOT gate would take in a 16-bit word and return a 16-bit word with each individual bit inverted.

Alternatively, multiple-input gates perform a logical operation on more than two inputs simultaneously and output the result.

2.1.2.3 Multiplexers and Demultiplexers

Multiplexers are devices that have a minimum of two data inputs, along with a controlling select input and a single output. The value of the select input determines which of the data inputs is sent to the output. Conversely, a demultiplexer is a device that has a single data input, controlling select input and multiple outputs. The value of the select input determines which output the input data is sent to.

For the hack computer, two input/output, multiple input/output and bitwise versions of these devices must be created.

2.1.3 Methodology

First, one NAND gate was used to create a NOT gate by connecting the two inputs together. Then one NOT and one NAND gate were used to create an AND gate by connecting the NOT gate to the output of the NAND. By analyzing the expected behavior and using a combination of NAND gates and previously created gates, all required gates and devices were created. Appendix A contains detailed block diagrams of how each device was created for all *Nand to Tetris* projects. The gates and devices created in project one serve as the building blocks for the more complicated devices constructed in future projects.

2.2 Project 2 – Boolean Arithmetic

2.2.1 Objective

Using the gates and devices created in project one, several devices that perform mathematical operations are constructed in project two. These include a half-adder, full-adder, 16-bit adder, 16-bit incrementor, and the ALU (arithmetic logic unit).

2.2.2 Background

The Hack computer can perform mathematical and logical operations on 16-bit signed integers. These operations include bitwise NOT, bitwise AND, and integer two's complement addition. The Hack CPU processes all data in binary form. Understanding the binary numbering system is fundamental in realizing these operations.

2.2.2.1 Binary Numbers

The binary numbering system uses a base of 2 in contrast with the base of 10 used in decimal numbering. Similar to how a base 10 decimal number is identified by the values of the 1's (10^0), 10's (10^1), 100's (10^2), etc. places, a binary number is identified by the 2^0, 2^1, 2^2, etc. places. Bit significance is an important concept when manipulating binary numbers. The highest power of two in a binary number is its most significant bit (MSB) and the lowest power of two (2^0) is the least significant bit (LSB). Figure 2 shows the comparison between the structure of a decimal number and its corresponding binary representation.

Decimal		
10^2	10^1	10^0
1	3	9
MSB		LSB

Binary							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	1	1
MSB							LSB

Figure 2

In this example, 8 bits are required to represent the decimal number 139 in binary. To find the binary equivalent, the largest power-of-two that is less than the decimal number must be found. In this case that is 2^7 or 128 so the 2^7 place is a 1. Then, it must be determined if the power of two of each less significant bit can be added to this number and still be less than the decimal number. If so, that place is a '1' and if not, it is a '0'. This procedure is repeated for each bit. Once all bits have been calculated, adding up the powers-of-two of each '1' bit will result in the original decimal number. For the example of 139: $2^7+2^3+2^1+2^0 = 128+8+2+1 = 139$.

2.2.2.2 Binary Addition

Binary addition can be carried out with the same algorithm used for decimal addition. Since there are only 1's and 0's in binary, adding a 1 to a 1 will result in a carry to a bit of higher significance. Therefore, adding two *n*-bit numbers will require a *n*+1 bit output to effectively perform addition on any input values of *n*-bits. Figure 3 shows the steps to adding two binary numbers 15 and 7.

1	1 1	1 1 1	1 1 1 1	1 1 1 1
1111	1111	1111	1111	1111
+ 0111	+ 0111	+ 0111	+ 0111	+ 0111
0	10	110	0110	10110

Figure 3

As seen in this example, 5 bits are required to hold the result of adding these two 4-bit binary numbers. Binary addition is implemented in hardware using a device called an adder.

2.2.2.3 Signed Binary Numbers and Two's Complement

Binary numbers must use a '1' or '0' to represent the sign of a number. For a signed binary number, the MSB is used to indicate sign. A MSB of 0 indicates a positive number and a MSB of 1 indicates a negative number. To find a positive binary number's corresponding negative, the two's complement must be found. The two's complement can be found by performing a bitwise inversion of the positive integer and then adding 1. Subtraction in binary can be performed by adding the two's complement of the number to be subtracted.

2.2.3 Methodology

A two-bit half adder was constructed using one XOR and one AND Gate. The logical operations of these gates perform the addition and carry out functions of binary addition, respectively. The defining characteristic of a half adder is the lack of a carry-in bit. Cascading two half adders together and using an OR gate for the carry allows for the creation of a full adder. The full adder takes in two inputs and a carry and returns a sum and carry out. A 16-bit adder was created using 16 full adders by connecting the carry in of the next significant bit to the carry out of the previous. A 16-bit incrementor was implemented using one 16-bit adder. One input of the adder was connected to the input of the incrementor and the other hard wired to be the binary value 1. The output returns the input value + 1, effectively incrementing the input value by 1.

2.2.3.1 Arithmetic Logic Unit

The arithmetic logic unit performs the mathematical and logical operations on the 16-bit words in the hack computer. It consists of two 16-bit inputs, one 16-bit output, 6 control bit inputs and two output flag bits. The values of the control bits determine what operation is performed by the ALU, on the two 16-bit inputs (operands) with the result being sent to the output. The output flags (*zr*, *ng*) show whether the output is or is not zero or negative. These are used later in the construction of the hack CPU. The ALU was implemented using a series of 16-bit multiplexers and several 16-bit logic devices. The control bits determine the output of the multiplexers such that certain combinations of control bit values will result in specific operations being performed by the ALU. A detailed block diagram of the ALU implementation is included in Appendix A.

2.3 Project 3 - Memory

2.3.1 Objective

Using the D-type flip flop as a basic building block, project 3 requires the construction of a single bit register, multi-bit register, several different sizes of RAM modules and a program counter.

2.3.2 Background

Project 3 introduces the concept of sequential, or clocked logic. This is in contrast with the combinational logic used in projects 1 and 2. Whereas combinational logic is somewhat instantaneous- that is, the outputs respond immediately to a change in input, sequential logic requires a clock edge to change the output. A change in input could theoretically happen at any

time, however the output will not be updated until the next clock cycle. Timing diagrams are used to analyze sequential circuits instead of truth tables. Timing diagrams show how signals in the circuit change over time in relation to the system clock.

2.3.2.1 D-Flip Flop

Although it can be constructed out of discrete logic gates, the D flip flop is given as a primitive building block for this project. This device can "remember" data and provides the fundamental building block for computer memory. The D flip flop has one data input, a data output and a clock input. Typically, it would also have set and reset inputs that function outside of the clock edge (asynchronously), but the model for a D flip flop given in this course does not utilize these features. Functionally, when a clock signal is applied to the clock input and a rising edge is detected, the data at input is sent to the output.

2.3.3 Methodology

The single bit register was created using a 2-bit multiplexer and one D flip flop. The register consists of a single-bit input and output, as well as a single load bit. When the load bit is set to 1, the input is sent to output on the next rising edge of the clock. Using 16 single bit registers, the 16-bit registers used in the hack CPU can be constructed. Each single bit register holds one bit of a 16-bit word, with all load pins connected. Several RAM modules of increasing size were needed. In addition to having the load functionality seen in the registers, the RAM modules also have addressing capabilities. The RAM is made of an array of registers and the address input selects which registers contents is seen at the output. To create the first module, RAM8, 8 16-bit registers were used. An 8-way multiplexer was used to select which register is sent to output by connecting the address bits to the multiplexers select port. Similarly, an 8-way demultiplexer was used to enable the loading function on a specific register again by connecting the addressing bits to the select port of the demultiplexer. Each subsequent RAM module uses these same principles to increase the size of the memory array. Instead of using registers to create large RAM modules, an array of the previously constructed RAM module was used. As the RAM8 module was constructed using 8 16-bit registers, a RAM64 module was constructed using 8 RAM8 modules, a RAM512 module was constructed using 8 RAM64 modules and so on.

The program counter is a counting device that keeps track of the current and next instructions to be executed by the hack CPU. It was created using one 16-bit register, several multiplexers and one incrementor. The program counter has a 16-bit input and output, which determine the next and current instruction. Additionally, it has a reset, load, and increment input. These three control bits are connected to the select port of the three multiplexers and determine whether the register is set to zero, set to the value at input or if the register's current value is incremented by one.

2.4 Project 4 – Machine Language

2.4.1 Objective

Using the Hack assembly language detailed in *The Elements of Computing Systems*, two programs must be written and tested. The programs that must be written are Mult.asm and Fill.asm. Mult.asm reads two numerical values stored in two specific memory registers, multiplies them together and stores the result in another specific memory location. Fill.asm monitors the memory location that stores keyboard data and upon detecting a key press, writes the value of each screen pixel to black.

2.4.2 Background

Machine language consists of a series of binary instructions that can be read by a computer's central processing unit (CPU) to execute a task. The Hack machine language uses 16-bit binary instructions. Typically, programs to be run on a computer are not written directly in machine language, as that would be incredibly tedious and prone to error. The lowest level of programming language typically used is assembly language- a textual representation of binary instructions. Assembly is a single layer of abstraction above binary machine language and easier to understand. Assembly programs are translated into binary machine language instructions by the assembler designed in project 6. Instructions written in assembly language provide a direct interface with the CPU to control its behavior.

2.4.2.1 Hack Architecture

Understanding the Hack machine language and the underlying hardware architecture go hand in hand. The following information provides enough background on the Hack architecture to understand the machine language instructions. Details on the Hack architecture will be further expanded upon in section 2.5 Computer Architecture.

Hack memory consists of a RAM (random access memory) module and a ROM (read-only memory) module. RAM, or data memory is where the CPU stores data and can be read from or written to by the CPU. The ROM, or instruction memory is where Hack instructions are stored and can only be read from. To execute a program on the Hack computer, binary instructions must be independently loaded into the ROM module, where they can then be read by the CPU. The address selection for the instruction memory is handled by the program counter.

The Hack CPU contains two internal registers used to execute instructions. The A-register or address register can be used for addressing memory locations or storing data. The D-register or data register is used to store data. A third register "M" is also referenced in Hack assembly. The M-register is not internal to the CPU, but rather refers to the contents of the currently selected memory register.

The A-register output is connected to the input of the program counter, the address input of memory and to a multiplexer which connects to the *y* input of the ALU. When a value is stored in the A-register, several things happen:

1. The current memory address becomes the value in the A-register

2. If program counter “load” is asserted, the value of the A-register is loaded into the program counter (the current instruction is set to the A-register value).
3. If the connected multiplexer is set to pass the A-register input through to output, the value of the A-register is sent to the y input of the ALU

Therefore, the A-register can be used to access a specific memory address, jump to a specific instruction in instruction memory, or send a constant value to the ALU for manipulation.

The D-register is connected to the *x* input of the ALU and is used to store data to be manipulated by the ALU or sent to a location in memory.

2.4.2.2 Hack Machine Language

The hack machine language consists of two main instruction types: A-instructions and C-instructions. A-instructions load a value into the A-register and are followed by a C-instruction, which instructs the CPU to perform an operation. The function realized by a given A-instruction is determined by the following C-instruction. Figure 2.4-1 was adapted from *The Elements of Computing Systems* and details the hack assembly specification.

	A-Instruction	C-Instruction
Symbolic	"*@xxx*"	*dest=comp;jump*
Binary	*0vvvvvvvvvvvvvvv*	*111accccccdddjjj*

Comp		**c**	**c**	**c**	**c**	**c**	**c**
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	*a=1*						

Dest	**d**	**d**	**d**	comp stored in:
null	0	0	0	not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
DM	0	1	1	D register, RAM[A]
A	1	0	0	A register
AM	1	0	1	A register, RAM[A]
AD	1	1	0	A register, D register
ADM	1	1	1	A register, D register, RAM[A]

Jump	**j**	**j**	**j**	effect:
null	0	0	0	no jump
JGT	0	0	1	jump if *comp* > 0
JEQ	0	1	0	jump if *comp* = 0
JGE	0	1	1	jump if *comp* ≥ 0
JLT	1	0	0	jump if *comp* < 0
JNE	1	0	1	jump if *comp* ≠ 0
JLE	1	1	0	jump if *comp* ≤ 0
JMP	1	1	1	unconditional jump

Figure 4 (Nisan & Schocken, 2008, 109)

The format for A-instructions in hack assembly are as follows: *"@xxx"*. Where *xxx* is a non-negative integer. The A-instruction is 15 bits wide, with the 16th (MSB) being assigned as an opcode that determines whether the instruction is an A instruction or C instruction. A-instructions have a MSB of '0' and C-instructions have a MSB of '1'. Integers between 0 and 32767 (2^{15}-1) can be used on the hack platform. Figure 5 demonstrates an A-instruction in hack assembly and its corresponding machine code.

Hack Assembly	Machine Code
@143	0000000010001111

Figure 5

C-instructions contain a 1-bit identifying opcode (MSB), two ignored bits that are conventionally set to 1, a 7-bit computation or *comp* field, a 3-bit destination or *dest* field and a 3-bit *jump* field. The computation field determines the operation performed by the ALU to manipulate data in the registers. The destination field determines where the result of the computation is stored. The jump field determines what instruction is fetched and executed next. In a C-instruction, the computation field is required, the destination and jump fields are optional. The *dest* and *comp* fields are separated by an equal sign ('='), the *comp* and *jump* fields are separated by a semicolon (';'). Figure 6 shows a generic C-instruction as well as examples of possible C-instructions and their corresponding machine codes.

	Hack Assembly	Machine Code	Action
Generic	*dest=comp;jump*	*111accccccdddjjj*	
No jump	D=D+A	1110000010010000	Sets D-register equal to value in D-register added to value in A-register
With jump	D;JGT	1110001100000001	[if preceded by A-instruction] Sets next instruction equal to value in A-register if value of D-register > 0

Figure 6

The hack assembly language also supports the use of variables and labels. Variable declaration uses A-instruction syntax, with *xxx* being a text string instead of an integer value (*@variable)*. Variables have a specific section of RAM allocated to keep track of them and are simply converted into their numerical address by the assembler when translated into binary instructions. The base RAM address for variables is RAM[16]. When variables are declared in a Hack assembly program, they will be assigned to the next available RAM address starting with RAM[16]. Labels are declared using the syntax *(label)* and can then be called upon using the syntax *@label*. The *(label)* declaration is used to differentiate between labels and variables in hack assembly programs. *@label* is used to reference the location *(label)* in instruction memory and therefore can be used to initiate a conditional jump to another part of a program. The use of labels and variables can make code easier to implement and understand.

2.4.3 Methodology

For mult.asm, the product of two numbers is found using a repetitive addition algorithm. This program requires the two values to be multiplied to be stored in RAM[0] and RAM[1]. An index value is then derived from the value in RAM[1] and stored in a variable. The index variable tracks the number of iterations of addition required. RAM[2] is used to store the result of the algorithm and is initialized to zero. Then, a loop is implemented in which the number in RAM[0] is added to the value in RAM[2] if the index is greater than zero, and then the index variable is decremented by one. Once the index variable reaches zero, the jump condition is met, and the program jumps to its end loop. The resulting behavior causes the value in RAM[0] to be added to itself the value of RAM[1] times, effectively multiplying the two numbers.

Fill.asm is implemented using a loop which looks for a value to appear in the KEY register (RAM[24576]), indicating a key press. If a key press is detected, The program loops through each SCREEN register (RAM[16384]-RAM[24575]) and sets the value to -1 (1111111111111111 in binary). Each pixel is represented by a bit in the SCREEN memory block; setting the pixel to 0 makes it white and setting it to 1 makes it black. The loop is implemented using an index value stored in RAM[1] that counts down through all of the screen registers, and upon reaching zero initiates a jump condition to the beginning of the program where it looks for the next key press. The Hack assembly program files Mult.asm and Fill.asm are included in Appendix C.

2.5 Project 5 – Computer Architecture

2.5.1 Objective

Project 5 calls for the implementation of the CPU, Hack memory module, and the connection of these components with the A and D registers, instruction memory, screen and keyboard to complete the Hack computer. The registers, instruction ROM, screen and keyboard are given as pre-built modules, however, the registers and ROM could be easily implemented using the devices created in project 3.

2.5.2 Background

The Hack computer consists only of the CPU, a ROM instruction memory and a RAM module. A reset control restarts the execution of the program stored in instruction memory. The CPU is responsible for fetching, decoding and executing instructions stored in ROM. These instructions may include the need to read or write data to memory. The CPU is connected to the RAM module in a way that allows for these transactions to occur. The hack computer uses *memory mapped I/O* (in/out) to interface with peripheral devices such as a keyboard or monitor. Memory mapped I/O allocates specific segments of memory to store data associated with these devices (Nisan & Schocken, 2008, 129-130).

2.5.2.1 Hack CPU

The central processing unit contains the ALU, program counter, A and D registers, and many other devices constructed in previous projects. It is responsible for obtaining, processing and

executing instructions and performing any necessary computations. The process of obtaining and executing instructions is known as the fetch-execute cycle. The Hack CPU has 2 16-bit inputs: *instruction* and *inM*. The *instruction* input takes in an instruction from instruction memory. The *inM* input takes in data from a selected RAM register. The CPU also has a reset, which resets the program counter to 0, causing the program loaded in ROM to start over at the first instruction. The CPU has 3 outputs that interface with the RAM module: *outM*, *writeM*, and *addressM*. *addressM* determines which register in RAM is accessed. *outM* sends data that has been processed by the CPU to RAM[*addressM*]. The *writeM* output determines whether or not the *outM* value is written to RAM[*addressM*]. An additional *pc* output (from the internal program counter) is connected to the address port of instruction memory to determine the next instruction.

2.5.2.2 Hack Memory

The Hack platform has read only instruction memory made of 2^{15} (32,768) 16-bit registers and a read/write data memory that consists of 3 segments: a segment of 2^{14} (16,384) registers used as general-purpose memory, a segment of 2^{13} (8,192) registers dedicated to the screen data and a single register segment dedicated to the keyboard data.

2.5.3 Methodology

The CPU utilizes the major components created in previous sections including the ALU, registers and program counter. The specific routing of the 16 instruction bits allows the CPU to decode the instructions and perform desired operations. The MSB (instruction[15]) determines if an instruction is processed as an A-instruction or a C-instruction; this is facilitated by a series of multiplexers controlled by the MSB. In the case of an A-instruction, a value is loaded into the A-register. For a C-instruction, a multiplexer controlled by the *a* bit (instruction[12]) determines if *inM* or the contents of the A-register are sent to the *y* input of the ALU. The six *c* bits (instruction[11..6]) are sent directly to the ALU control bits. Three *d* bits (instruction[5..3]) are used with additional logic to control the A and D register *load* functions, as well as the *writeM* output bit which initiates a memory write. Finally, the three *j* bits (instruction[2..0]) are used with a combinational logic circuit to control the program counter's *load* port. Using the *Zr* and *Ng* output flags from the ALU, a series of boolean expressions were derived and simplified using boolean algebra to create the jump logic circuitry.

The Hack memory unit was constructed using a 16K RAM module called Base RAM, an 8K RAM module called Screen RAM, a single keyboard register, one demultiplexer and one multiplexer. A single-bit demultiplexer, whose *select* is controlled by the MSB of the address input, controls the load bits for the base and screen RAM. The keyboard register does not need to be written to by the hack CPU, so it does not have a *load* function. Instead, it is used only to access data provided by the keyboard. The memory output is determined using a 4-way multiplexer controlled by address[14..13]. The hack memory unit uses a 15-bit address; however, the base RAM segment only requires 14 address bits and the screen RAM only requires 13. The two MSB are used to select between the RAM segments, and the less significant bits are used to select a specific address within that segment. If the two most significant address bits, address[14..13] are observed, it is seen that values of "00" or "01" correspond to base RAM

addresses, a value of “10” corresponds to screen RAM addresses, and a value of “11” corresponds to the keyboard register.

The Hack computer is then assembled by combining the CPU, RAM and instruction memory. The instruction memory is connected to the CPU’s instruction input and program counter output. The RAM is connected to the CPU’s *inM* input and *outM*, *writeM* and *addressM* outputs.

2.6 Project 6 - Assembler

2.6.1 Objective

A program must be written to translate Hack assembly instructions into binary machine codes that can be executed by the Hack computer. The program reads in a text file with the *.asm* extension that contains Hack assembly instructions, and outputs a binary file with the *.hack* extension that can be read by the Hack CPU.

2.6.2 Background

The assembler must parse lines of assembly code and translate them into binary instructions. The first computers constructed would have required their assembler to be written in the native machine language. Fortunately today, a higher level language running on an already functional computer can be utilized to write this program.

2.6.3 Methodology

The hack assembler was written using the Python programming language. The assembler takes in a text file containing Hack assembly instructions and outputs a modified text file with the decoded binary instructions. The program uses tables and dictionaries to look up assembly commands, store variables and labels, and store translated lines of code. The input code must be parsed in multiple passes to remove comments and whitespace, identify labels and variables and add them to the symbol table, and update the binary instructions.

First, a dictionary consisting of the pre-defined symbols listed in the hack assembly specification was created. The dictionary contains each symbol and its corresponding numerical value. This “symbol table” is updated with user-defined symbols as the program parses the input file.

The assembler opens the input .asm file and loops through each line, writing it to a list. This results in an instruction list that contains each assembly instruction as a separate string. The instruction list is then looped through line-by-line to remove any whitespace (spaces, line or tab breaks) and comments. Then a pass is made to identify labels by their *(label)* syntax. The labels must be associated with the next line address. As the assembler encounters labels, they are added to the symbol table and a label index is used to associate the text label with the proper numerical instruction memory address.

A second pass checks instructions that start with the ‘@’ symbol to determine if they are A-instructions, references to a label, or variable declarations. The assembler achieves this by first determining whether it is an A instruction. This is done by checking if the text following ‘@’ is a number. If it is, the instruction is an A-instruction and ignored, if not the program moves on to

check for label or variable reference. Next, the program checks for label reference by comparing what follows the '@' to the contents of the symbol table. If a match is found, the label reference is replaced by its numerical representation. If a match is not found, the instruction must be a variable declaration and the symbol is saved in the symbol table. Hack variables are stored starting in memory register 16, so a variable register index is created and incremented each time a variable is found. This index is used to associate a variable symbol with the next available variable memory segment. The variable is replaced with its numerical representation. The assembler then loops through each line and converts A instructions into their binary numerical representation.

Finally, the C instructions are decoded. Three reference lists are created with all possible values of the *dest*, *comp* and *jump* fields of the C instructions and their corresponding numerical codes. The assembler uses regular expression matching to acquire the corresponding numerical code from the reference list. The numerical *dest*, *comp* and *jump* codes are then concatenated and then written over the original assembly instruction. Then the list containing the binary instructions is looped through and written line-for-line to an output text file, which is given the same name as the input file with a *.hack* extension.

3.0 Hardware Implementation

The hardware implementation of the Hack platform was completed using VHDL and a Terasic DE10-Standard FPGA Development Board. VHDL is a hardware description language that can be used to write an abstract description of a physical circuit. The VHDL code is then processed by development software; Intel Quartus Prime Lite Edition was used for this project. The software analyzes the written code and creates lower-level code that can be read by the FPGA.

A FPGA is a device that consists of a two-dimensional array of logic cells and switches. The switches can be programmed to determine connections between logic cells. Each logic cell contains a configurable combinational logic circuit and a D flip flop (Chu, 2008, 11-12). When VHDL code is compiled by the development software, the lower-level code created translates VHDL instructions into instructions that organize these logic cells and switches in a way that realizes the functionality of the VHDL module.

The FPGA provides a useful platform for the prototyping and development of digital circuits and is used to host the implementation of the microcomputer based on the Hack platform. Hack assembly programs can be run by initializing the instruction memory with the program binaries. Appendix B contains simulation diagrams and details pertaining to the execution of assembly programs on the FPGA implementation of the microcomputer.

The Hack computer was implemented using a top-level *hack* module and several main submodules: *hack_cpu*, *hack_mem*, *instruction_ram*, *display_map*, *vga* and *debug*. The *hack_cpu*, *hack_mem* and *instruction_ram* modules make up the Hack computer. The *display_map* module is an intermediary device which interfaces with the screen RAM segment to send display information to a VGA controller. The *vga* module is a VGA controller which

provides the timing and data signals required to interface with a VGA capable monitor. Finally, the *debug* module utilizes the onboard 7-segment displays and several switches/buttons to view useful internal data while operating the Hack computer.

3.1 Approach

In the *Nand to Tetris* course, more complex devices are built by only using previously constructed, simpler devices. Each higher-level device is simply a combination of previously built devices. This method requires a holistic understanding of each device's behavior to form larger devices. Considering this approach was used during the simulation part of this project, a slightly different approach was utilized for the hardware implementation. The hack computer could be implemented in VHDL by re-creating each chip made in the *Nand to Tetris* Projects and connecting them all together in the same way. However, this wouldn't utilize a large benefit of using a hardware description language like VHDL. VHDL allows for the inference of circuitry by a software compiler when using more abstract terminology in coding. Where in *Nand to Tetris* HDL each chip had to be manually created from previously built chips, VHDL allows for the use of abstract statements to imply the logic that must be created. This allows for code that is easier to write and understand. It is important to note that while modules written in VHDL may be referred to as "code", VHDL is very different from a computer programming language. VHDL code is a textual description of a physical circuit, whereas programming languages are instructions meant to be executed sequentially by a computer (like the Hack platform).

There are some instances in the hardware implementation where single modules were written for discrete components like in the *Nand to Tetris* projects. However, most modules use typical VHDL methods to imply the lower-level devices. For example, instead of creating a multiplexer module and instantiating it for use in the Hack CPU, a *case* statement was used.

Each main module also has a corresponding testbench, in which input stimuli are provided. The outputs and internal signals can be observed in simulation to determine proper functionality before use in the FPGA implementation.

3.1.1 CPU

The CPU implementation realizes the base functionality of the Hack CPU and additionally provides several output signals for use with the *debug* module. Three submodules were written and instantiated in the CPU module: *reg_generic, hack_alu* and *program_counter*. The *reg_generic* module provides a single memory register with a configurable bus width (16 bits for the Hack platform). This module is instantiated twice in the CPU for the A and D registers. The ALU and program counter were written as separate modules to allow for efficient testing and debugging. The CPU module makes proper connections between all the internal components and maps the inputs and instruction bits to their proper locations. Multiplexers are realized using *case* statements. Several signals were created to pass specific data out of the CPU; these will later be used by the *debug* module to assist in debugging the computer when programmed onto the FPGA. The *d_reg_ext* and *a_reg_ext* signals send out the output of the D and A registers, the *d_ld_flag* and *a_ld_flag* are connected to the D and A registers *load* port, and the *z_flag* and *n_flag* are connected to the *zr* and *ng* output flags on the ALU. The CPU runs on an 8MHz clock,

which was generated using The Altera PLL (phase-locked loop) IP core. A PLL is a frequency control system that can be used to generate different clock frequencies from a single input clock.

3.1.1.1 ALU

The ALU does utilize several individual modules written for the 16-bit bitwise chips. The VHDL for these chips is repetitive, so they were written in separate modules to make the ALU code more readable. Modules for *not_16, adder_16* and *or_16way* chips were written. The ALU uses these modules combined with multiplexers inferred using *case* statements to realize proper functionality.

3.1.1.2 Program Counter

The program counter uses *if* statements to set the count and output to zero if *reset* is asserted, set the count to the input value if *load* is asserted, and increment count by one if *inc* is asserted. The *if* statements imply several cascading multiplexers and the count signal implies a 16-bit register (made of 16 D flip flops) which stores the count value.

3.1.1.3 Register

The *reg_generic* module is configurable for different sizes. In the hack CPU it is configured to be 16-bits wide. It uses an asynchronous reset and an *if* statement to send input data to output data on a clock edge if *load* is asserted.

3.1.2 Memory

3.1.2.1 Data Memory

The hack memory module utilizes IP Cores provided with the Intel Quartus design software to instantiate embedded SRAM on the FPGA board. Embedded RAM segments must be utilized for memory because there is not enough room to construct the RAM in the FPGA itself. The base RAM segment uses single-port RAM and the screen RAM segment uses a dual-port RAM. The dual-port RAM must be used for the screen segment to allow for the *display_map* module to strobe RAM addresses independently from the CPU. The keyboard register was implemented using a single *reg_generic*. This module also utilizes multiplexers implied by *with select* statements to control the memory output and load functions of each memory segment.

3.1.2.2 Instruction Memory

The instruction memory was also implemented using a single-port RAM instantiation from the Intel IP Cores. Since the instruction memory is read-only, the *write* port is hard coded to ‘0’ so that it can never be written to.

3.1.2.3 Memory Initialization

The instruction memory must be initialized with a *.hack* binary file. Unfortunately, Quartus does not read *.hack* files directly. Instead, a *.mif* (memory initialization file) must be created. This is a basic text file with specific format that can be used to initialize the memory modules. Using a similar framework to the hack assembler, another Python program *bin_to_mif.py* was written that

takes in a *.hack* binary file and outputs a *.mif* file that can be read by the Quartus software and loaded into memory. The Instruction memory can be initialized with a hack program and the data RAM can be initialized to zero values to ensure a uniform starting point to run a program.

3.1.3 VGA Controller

VGA, or Video Graphics Array, is a graphics standard used to interface a computer with a graphical display. The standard consists of a set of resolution-specific parameters that must be implemented for proper function. VGA is made up of horizontal and vertical sync signals (*h_sync* and *v_sync*), data signals for each color channel (red, green and blue), and a VGA clock signal. The VGA controller is responsible for generating the VGA clock signal and all synchronization and data signals necessary for use of an external VGA monitor. This VGA controller implementation uses a resolution of 640x480 @60Hz. This means the generated display is 640 pixels wide by 480 pixels tall and refreshes at a rate of 60 times per second.

VGA sync signals are used to generate synchronizing pulses which keep track of where data is displayed on the screen. The horizontal and vertical sync signals are broken up into several zones or regions. The time or pixel count the signal spends in each zone is dependent on the screen resolution. The signal zones consist of the visible are, front porch, sync pulse and back porch. The visible area is the area displayed on screen. The off-screen area is broken up into the front porch (time before synchronization pulse), the actual sync pulse, and the back porch (time after synchronization pulse). At the end of the back porch zone, the signal is repeated. Figure 7 details the zones of the synchronization signals.

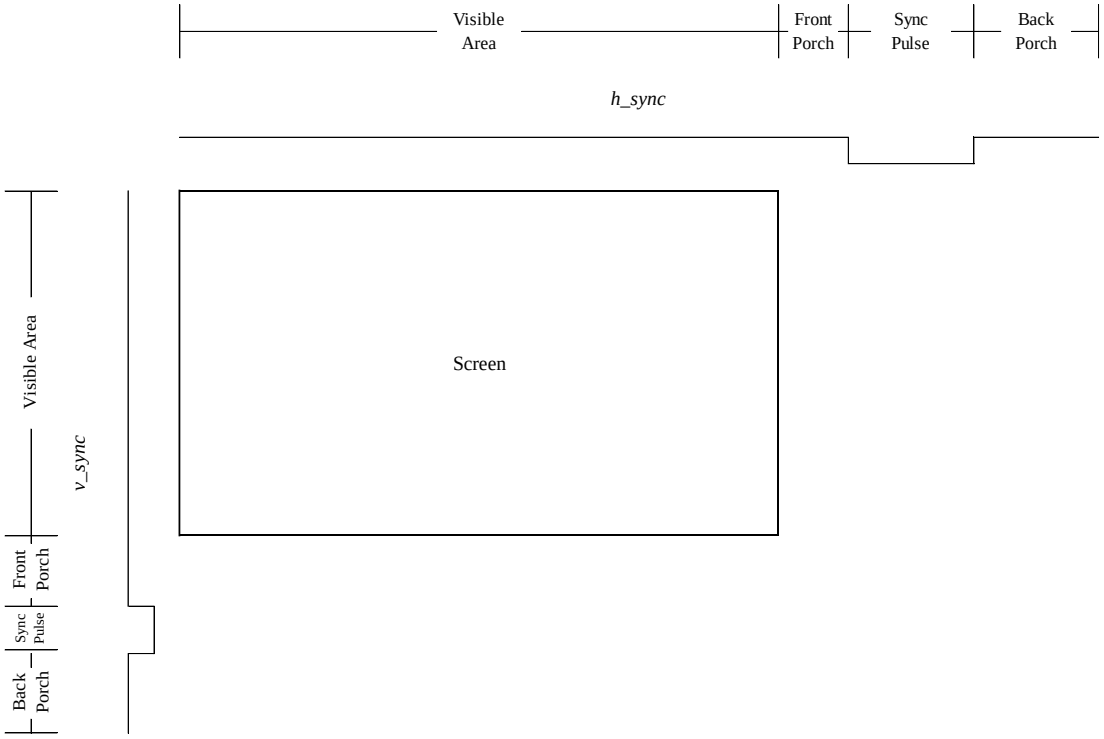

Figure 7

The length of the synchronization signals and the VGA clock speed is determined by the desired screen resolution and refresh rate. For the 640x480 resolution, the Visible Area is 640 pixels wide and 480 pixels tall. When taking the out of screen zones into account, the total horizontal pixel count is 800 and the total vertical pixel count is 525. The timing specifications and pixel count for each zone are shown in Figure 8 (adapted from *TinyVGA.com)*.

General Timing	
Screen Refresh	60Hz
Vertical Refresh	31.46815kHz
Pixel Frequency	25.175MHz

Horizontal Timing		
Zone	*Pixel Count*	*Time in µs*
Visible Area	640	25.422046
Front Porch	16	0.6355511
Sync Pulse	96	3.8133069
Back Porch	48	1.9066534
Whole Line	800	31.777557

Vertical Timing		
Zone	*Pixel Count*	*Time in ms*
Visible Area	480	15.253227
Front Porch	10	0.3177756
Sync Pulse	2	0.0635551
Back Porch	33	1.0486594
Whole Line	525	16.683217

Figure 8 (www.tinyvga.com, n.d.)

The VGA controller uses counters to generate the horizontal and vertical sync signals and is operated via a 25.175MHz VGA clock corresponding with the proper pixel frequency for the 640x480 resolution. When clocked at the proper frequency the counters quite literally count through each pixel on the screen. The horizontal sync pulse is triggered by a counter that counts 0-799, repeatedly. The horizontal sync is an active low signal, so it is initialized to a logic HIGH value. Once the count reaches 655, the signal is pulled low (generating the sync pulse) through count 751, then returns high. The count values that trigger the beginning and end of the sync pulse were determined using the pixel count data in Figure 3.1-2. Every time the horizontal count reaches 799 and subsequently resets to 0, a vertical count is incremented by one until it reaches 524 at which point it resets to 0. The reset of the vertical count marks a “refresh” of the screen; this happens 60 times per second.

The VGA controller does not govern what data is sent to the screen. It takes in external color data and turns it “on” during the visible region of the count cycle. The color data must be pulled to a logic LOW during the off-screen region of the count cycle; this action is also performed by the VGA controller by using *if* statements and the horizontal and vertical counts.

3.1.4 Display Map Interface

The display map serves as an intermediary between screen memory and the VGA controller. Its primary objective is to map the pixel data stored in screen memory to a specific location on a display. This is achieved using several counters that operate in sync with the VGA clock. Since the VGA synchronization signals count through each screen pixel line-by-line, a synchronized count must be generated to count through each screen memory location to route screen data to the proper pixel location on the display. Dual-port RAM is used for the screen memory segment to allow the display map independent access to the screen memory regardless of the behavior of the CPU.

A horizontal and vertical sync count identical to the one found in the VGA controller is maintained in the display map module to keep track of where data will be output on the screen. Similarly to how the VGA controller uses counters to maintain sync signals, the display map uses a horizontal and vertical counter to keep track of screen pixels. These counters count from 0-511 and 0-255 and are triggered by the VGA *h_sync* and *v_sync* counters.

The hack computer supports a display resolution of 512x256, so some scaling was necessary to use 640x480 VGA. The writeable 512x256 area was centered within the actual 640x480 screen with the inactive edges set to black. To center the hack screen's 512 horizontal pixels, the horizontal count is triggered to begin when the *h_sync* count reaches 64 and end when it reaches 576. The 64 pixels left on either side of the 512-pixel screen make up the inactive edge buffer. Similarly, the vertical count is triggered when the *v_sync* signal is between 112 and 368, leaving a 112-pixel inactive buffer on the top and bottom of the "active" screen. It is important to note that this scaling solution allows for the use of the hack memory module as designed by the authors of the *Nand to Tetris* course. The hack memory design only contains enough room in the screen memory to store pixel data for a 512x256 screen. Higher resolutions could be fully supported by expanding the memory module to store more screen pixel data.

The screen memory segment can be thought of as a one-dimensional array; it is linear and consists of 8192 16-bit registers. Each of these registers store data for a single pixel on the screen. The screen itself, however, is a two-dimensional array of pixels consisting of a horizontal and vertical position. The main task of the display map is to synchronize a count through the screen memory addresses with the position counters to match the linear memory addresses with their corresponding two-dimensional screen position. This results in a pixel's data being accessed from memory each time it appears in the positional count. The pixel data stored in RAM can then be sent to the monitor and displayed in the correct location. Figure 9 provides a visual representation of how linear memory addresses are mapped to the display to hold pixel data.

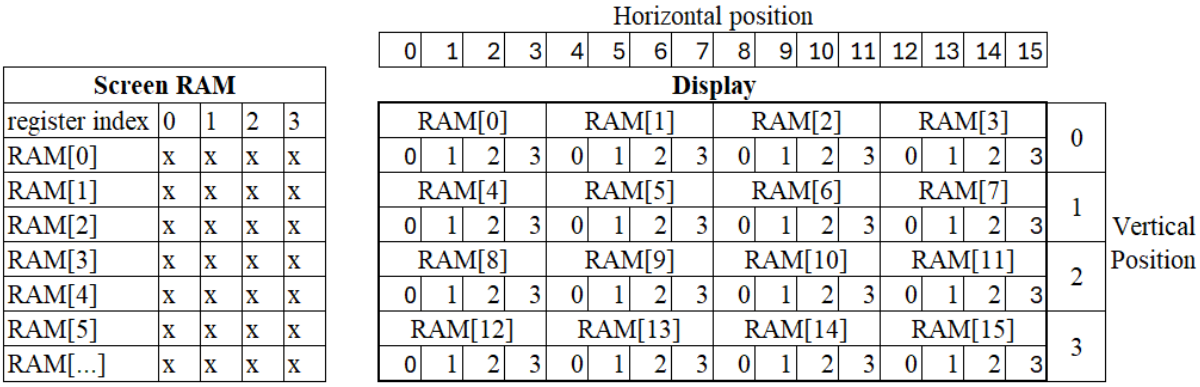

Figure 9

This example is scaled down from the actual 512x256 size of the hack screen, instead using 4-bit registers and a 16x4 screen. However, this illustrates how linear RAM can be used to represent a

physical location on a two-dimensional screen, regardless of size. The horizontal and vertical position counts keep track of the x, y coordinates of the pixels, while the register and index counts keep track of the pixel data stored in memory. If the screen register addresses are counted through and accessed in synchronization with the position counts, the contents of screen RAM will display at a predictable location on the display.

Each memory register in screen RAM holds pixel data for 16 pixels, so the horizontal and vertical position counters must operate in sync with two additional counters: a position index counter *pos_index* that counts 0-15 through each register location and a register counter *reg_count* which counts 0-8191 through each register in screen RAM. The position index counter is triggered off the horizontal position counter: As the horizontal counter counts from 0-511, the position index counts from 0-15 repeatedly equating the horizontal position on screen with its corresponding location within a specific register. The register count is triggered off the position index: each time the position index completes a count and resets (indicating 16 bits, the size of one register), the register count is incremented by 1. The register count resets once each register has been counted through, this occurs at the same time as the vertical position reset (end of screen). The register count is used to access the corresponding register's RAM address. Then the data in that register is sent to the VGA data output using the position index to ensure the proper pixel is being output.

The Hack platform only supports black and white- that is, the data stored for each pixel is 1-bit and it can only either be on or off. Since VGA uses three color channels to display more complex color data, the single-bit pixel data must be used to trigger all three VGA color channels to display either black (R=0, G=0, B=0) or white (R=1, G=1, B=1). Note that the values to display black and white using VGA are opposite of those detailed in the Hack specification.

3.1.5 Debug

The debug module provides a "window" into the internal function of the Hack computer to ensure proper function and aid in debugging potential problems. The DE10-Standard FPGA development board has several sliding switches, push buttons, LEDs and 7-segment displays which are utilized by the debug module to provide certain data to the user.

The sliding switches are used to select what internal data is displayed on the 7-segment displays. Data is shown in hexadecimal notation. The module uses multiplexers to decode the binary data from the hack computer to 7-bit strings of data used to control the display segments. A switch was set up to activate manual clocking of the CPU using a push-button. 8MHz is far too fast to be able to monitor what is happening during the execution of a program, so this feature is incredibly useful for debugging. A program can be clocked through one instruction at a time and the 7-segment displays can be used to view internal data during each instruction. Figure 10 provides specifications on the configuration of the switches and data that can be displayed on the 7-segment displays and the LEDs.

LEDS	
LEDR[0]	RAM load
LEDR[1]	Screen load
LEDR[2]	D-Register load
LEDR[3]	A-register load
LEDR[8]	ALU Ng Flag
LEDR[9]	ALU Zr Flag

Switches	
SW[0]	Master Reset
SW[1]	7-segment Select
SW[2]	7-segment Select
SW[3]	7-segment Select
SW[8]	Clock Select
SW[9]	Clock Reset

Push Buttons	
KEY[0]	Manual Clock

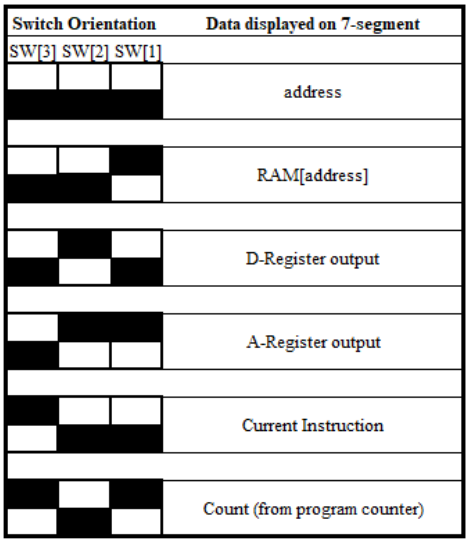

Figure 10

3.2 Challenges

The main challenges encountered in the FPGA hardware implementation of the Hack computer can be broken down into two areas: timing and peripheral connections. Correct timing and handling of clock signals in the Hack system is crucial for proper performance. Data must arrive in different locations within the computer in a predictable manner, so that it can be effectively processed. The Connection of peripheral devices to the Hack computer, including the use of external RAM modules also proved to be challenging.

The general architecture of the Hack platform was detailed extensively in *The Elements of Computing Systems*, the timing specifications less so, and the intermediary logic required to connect to peripheral devices was not covered at all. These areas posed unique challenges that required a great deal of research and testing to overcome. The following subsections detail several specific problems that were encountered and solved in the implementation process.

3.2.1 Timing

The *Nand to Tetris* course provides a web-based simulation environment to test the devices created in each project. For the sequential circuits that are created, all clock signals are handled by the software. In the VHDL hardware implementation of the Hack computer, all clock signals must be configured manually for the device to function. One of the main challenges of implementing the Hack computer in hardware was the correct management of clock signals. This process was done largely through trial-and-error and extensive testing of each individual module. Several Timing issues were identified and corrected during the hardware implementation process.

3.2.1.1 ALU

An issue with the ALU is being included in this section on timing because the nature of the problem involved data that was not being updated in time to be read by the next stage of

processing logic. This is not, however, related to the system clock because the internal ALU circuitry is not directly governed by the clock and consists only of combinational logic.

The original VHDL design for the ALU utilized *case* statements to imply multiplexers for the decoding of the control bits and subsequent execution of the decoded operation. These *case* statements can only occur in something called a *process*. VHDL processes use a sensitivity list that contains signals that can trigger the process to execute. If a signal is included in a process's sensitivity list, that process will execute any time the signal changes. The multiplexers within the ALU are controlled by the 6 control signals (*zx*, *nx*, *zy*, *ny*, *f* and *no*); the original ALU implementation used processes that were sensitive to their control signal to synthesize a multiplexer used to select between corresponding input data. However, this did not function correctly; the control signals only update once per instruction and, in turn, only execute their corresponding processes once and concurrently. This is problematic because the multiplexers in the ALU are hierarchical: the output value of a previous process is needed to determine the output of the following process.

For example, the *zx* control bit is used to determine if the *x* input must be set to zero. Then the output of that operation is sent to the *nx* multiplexer, where the *nx* bit is used to determine whether that input is negated. The output of the operation governed by the *nx* bit requires input that is dependent on the operation governed by the *zx* bit. If the process is only sensitive to the *nx* control bit, it will be executed at the same time as the *zx* process- before the input data needed from that process exists. This results in an unknown/uninitialized output state. To correct this issue, the processes written for multiplexing operations must be sensitive to the outputs from the previous stage that drive the inputs of the current stage. This solution ensures that the processes will be executed after the required input has been updated from the previous multiplexer.

In the process of determining a solution to the *case* statement problem, another version of the ALU was created using *with select* statements instead. These statements imply similar if not identical logic to the *case* statements (when the *case* statements are written properly). Both versions of the ALU function identically, showing how the same result can be achieved using different methods in VHDL.

3.2.1.2 Program Counter

Determining proper timing function for the program counter was one of the more challenging tasks to accomplish. The initial VHDL design for the program counter used an asynchronous reset and load with a synchronous counter. In simulation, this program counter design seemed to function properly, as the test programs appeared to run without issue. However, when downloaded to the FPGA, the program counter behaved unpredictably. The asynchronous nature of the control functions caused instructions to be obtained immediately after the count was updated. This caused an issue where the next instruction was fetched before the previous instruction could be properly executed. This problem necessitated the creation of the *debug* module so the performance of the hack CPU could be monitored while running. It was found that the counter must operate completely synchronously to properly function in the hack platform. The counter was corrected and implemented using synchronous reset and load signals. As a

result, the count leads the instruction by half a clock cycle; the count updates on the falling edge of the clock, and the CPU receives the corresponding instruction at the next rising edge.

3.2.1.3 Memory

Initially the data output of Hack memory was assigned on a clock edge. If a command was issued to access a memory register, it would take two clock cycles for the data in that register to show up at the CPU's *inM* port. On the first rising edge, *addressM* would update and on the following rising edge, the data in that address would be accessible. This was problematic because memory access commands are typically followed by a computational C-instruction where the data accessed is saved in one of the CPU's internal registers or manipulated by the CPU in some way. Each instruction is fed to the CPU in one full clock cycle, so the memory data would not be accessible in time for manipulation by the C-instruction.

Assigning the memory data output within a process sensitive to the clock, essentially created an additional flip flop at the memory output, delaying the accessibility of the accessed data by an additional clock cycle. By moving this output assignment outside of a clock process, this delay was eliminated. Due to the design of the internal CPU registers, their values update on the falling edge of the clock. If a command to access a memory register is given, it's address is loaded into the A-register on the next falling edge, then the data is accessible at the *inM* input of the CPU at the next rising edge. So, by updating the memory design, it takes exactly one clock cycle to access data from memory. Figure 11 shows a timing diagram of memory access in the Hack platform.

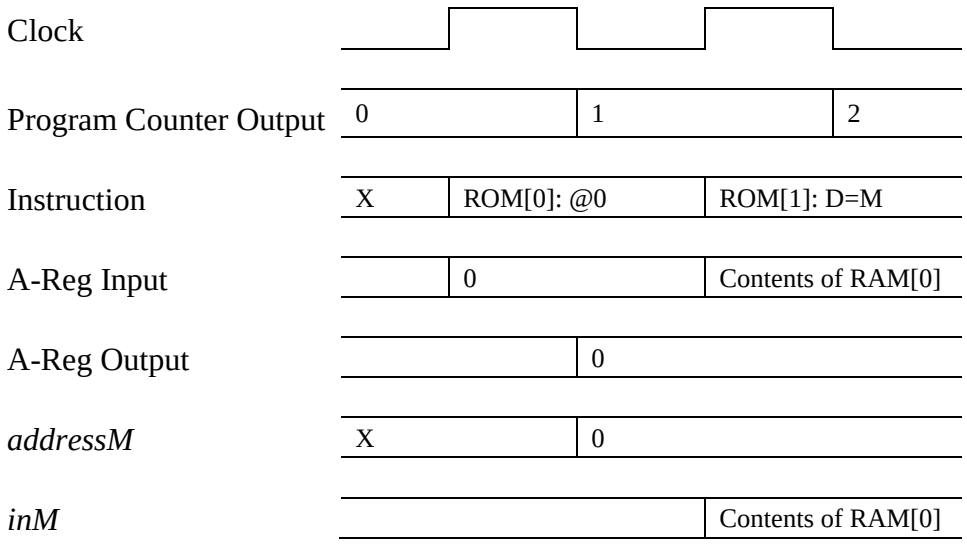

Figure 11

This diagram shows how the instruction @0 accesses the data residing in that location on the rising edge of the next clock cycle. This coincides with the following computation instruction, D=M. This ensures the data is accessible for manipulation by the CPU during the execution of the computation instruction.

3.2.2 Peripherals

Computers often connect to peripheral devices such as a keyboard, mouse and monitor. The CPU itself is not typically designed for use with a specific device. Given that there are countless options available for peripheral devices, the CPU interfaces with them in a generic manner. The devices themselves contain device drivers that ensure proper operation on a variety of host computers. For the hack computer, this intermediary logic was created to interface with a VGA monitor.

Additionally, the FPGA development board used contains embedded SRAM modules that were used for hack memory. There was not enough room on the FPGA itself to instantiate the hack memory module as designed, so the external embedded RAM was utilized.

The following subsections note some issues that were overcome to utilize the embedded RAM and create a VGA interface for use with the Hack computer.

3.2.2.1 Ram Instantiation

Initially, the memory was implemented using an array of registers similar to how the Hack memory chip was constructed in the *Nand to Tetris* course. However, there are a finite number of logic blocks available for use in the FPGA and the implementation of the Hack memory unit within the FPGA itself far exceeded the logic available. Simply, the Hack memory unit would not fit in the FPGA as designed. Instead, the FPGA development board contains embedded SRAM which was instantiated for use as memory for the Hack platform. Arriving at and implementing this solution required some additional research and testing.

3.2.2.2 Display Interface

The initial approach to implementing the memory map for the display used the custom Hack memory chip modified in such a way to bring each register out of the device as an output. Then each register could be counted through in the display map to provide output data to the screen. However, when it was determined that this memory design could not fit in the FPGA, a different solution had to be found. Upon researching how memory maps are implemented in actual computers, dual-port RAM was found to be a viable solution. Dual-port RAM provides two sets of independent data inputs and outputs, each with their own address input. This allows one in/out/address set to be utilized by the CPU for normal operation. The other output and address are used by the display map to strobe through all the screen data registers in sync with the VGA signals. Since the display map only needs to read data in these registers, the input is unused.

3.3 Conclusions

When looking at a complicated system such as a computer, it can be difficult to relate the top-level design to its fundamental logic. This project provides a step-by-step pathway from basic logic gates to a more complex computing system. A system can be more easily and thoroughly understood when broken down into discrete parts. By thinking about a computing system as a

sum of its parts, the connection between basic components and the higher-level system becomes clear.

When designing or implementing a system made up of smaller parts, a detailed and methodical approach is crucial. Through the process of implementing this microcomputer, it became clear that each individual module must be thoroughly designed and tested. Additionally, the larger system being created must always be kept in consideration. Detailed testing of each individual module proved to be valuable in debugging the whole system.

The way a computer processes and manipulates data is characteristic to its design. There is not a singular correct way to design any one component in the computer. All components that are governed by the clock must be carefully designed to function together so that the overarching goal of the computer can be realized: fetch, decode and execute instructions. The components that play a large role in this process cannot only be looked at individually. They are interconnected and their respective designs must take one another into account. If any individual part is changed, others may need to be changed as well. The timing specifications of each of these components were carefully designed and tested to work with one another to ensure proper execution of tasks on the computer as a whole.

4.0 References

Noam Nisan and Shimon Schocken (2008). *The Elements of Computing Systems*. MIT Press.

www.tinyvga.com. (n.d.). *VGA Signal 640 x 480 @ 60 Hz Industry standard timing*. [online] Available at: http://www.tinyvga.com/vga-timing/640x480@60Hz [Accessed 27 Jul. 2024].

Chu, P.P. (2011). *FPGA Prototyping by VHDL Examples : Xilinx Spartan-3 Version*. Somerset: Wiley.

Appendix A – Nand to Tetris Device Block Diagrams

Note: Many of the 16-bit devices are simplified to 4-bit devices to keep the diagrams clear and concise.

Figure A-1: NOT gate created with one NAND gate.

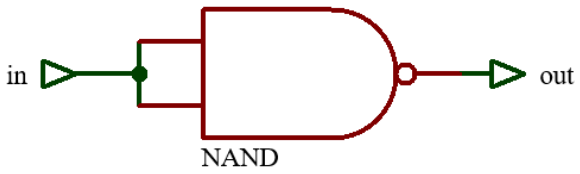

Figure A-1

Figure A-2: AND gate created with one NOT gate and two NAND gates.

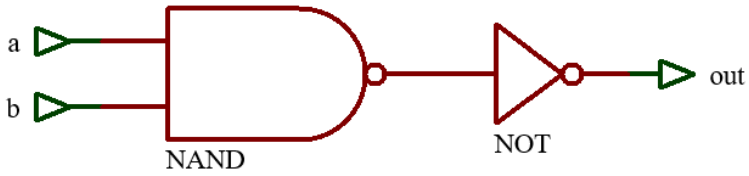

Figure A -2

Figure A-3: OR gate created with two NOT gates and one NAND gate.

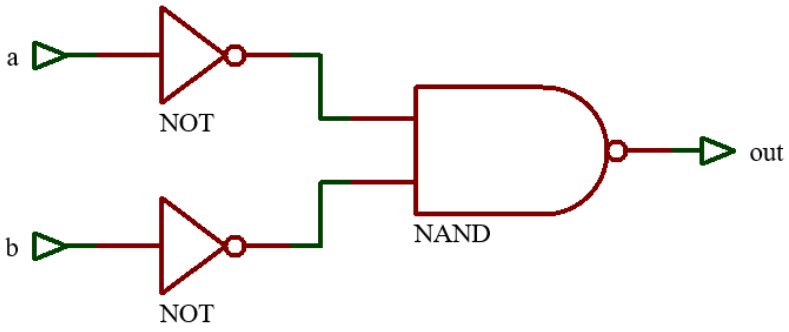

Figure A-3

Figure A-4: XOR gate created with two NOT gates, two AND gates and one OR gate.

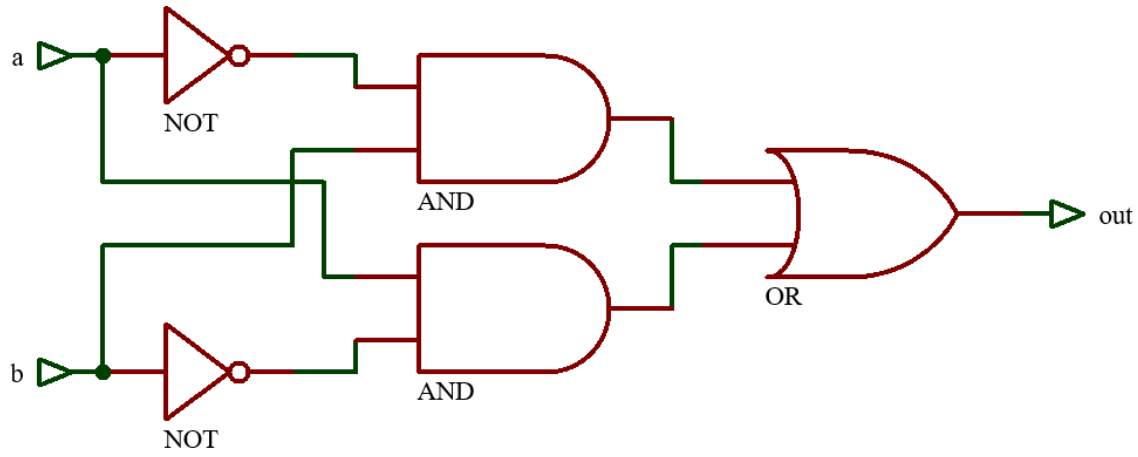

Figure A-4

Figure A-5: A 2-input, single-bit Multiplexer

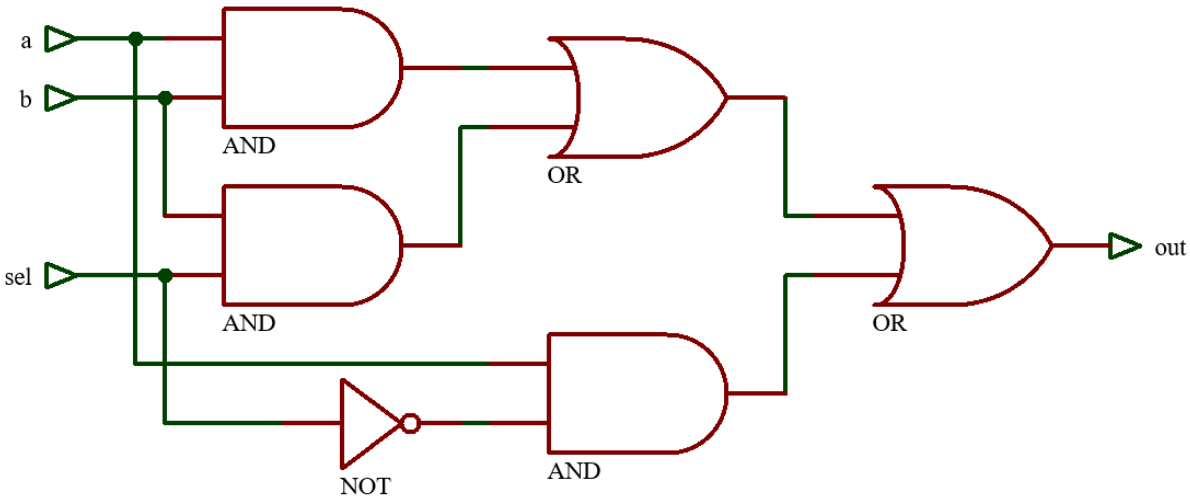

Figure A-5

Figure A-6: A 2-output, single-bit demultiplexer.

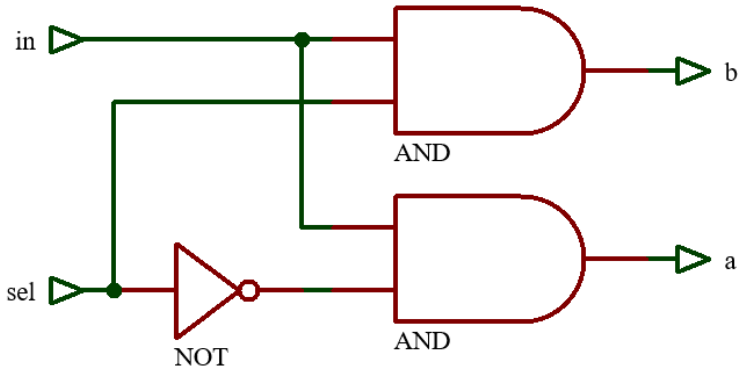

Figure A-6

Figure A-7: A 4-bit bitwise NOT gate.

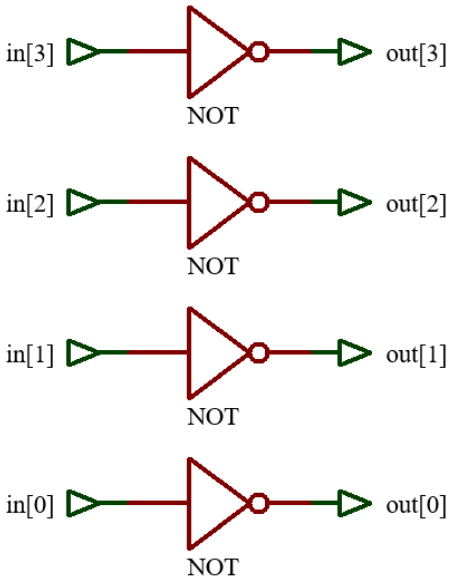

Figure A-7

Figure A-8: A 4-bit bitwise AND gate.

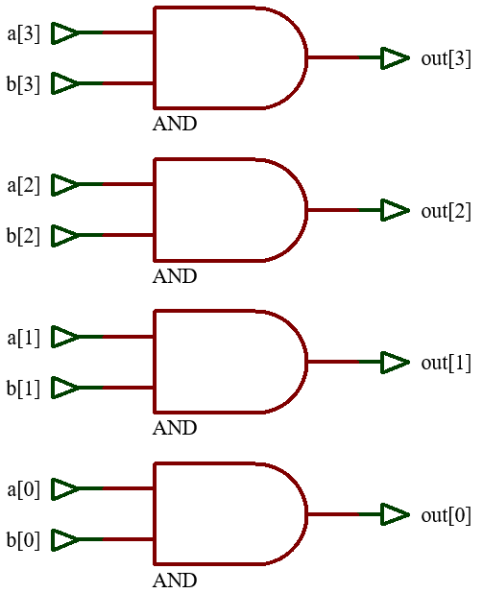

Figure A-8

Figure A-9: A 4-bit bitwise OR gate.

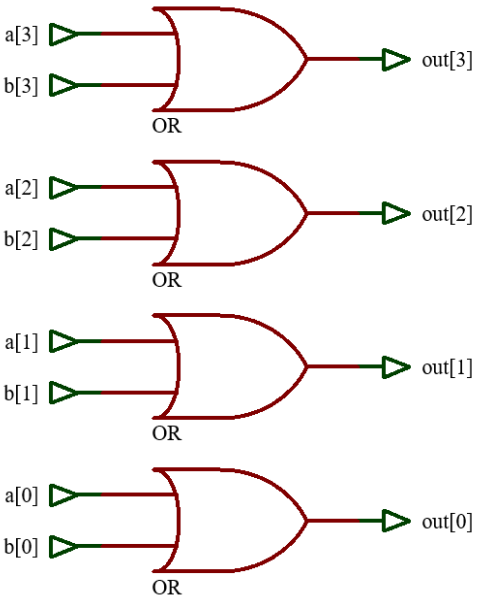

Figure A-9

Figure A-10: A 2-input 4-bit multiplexer.

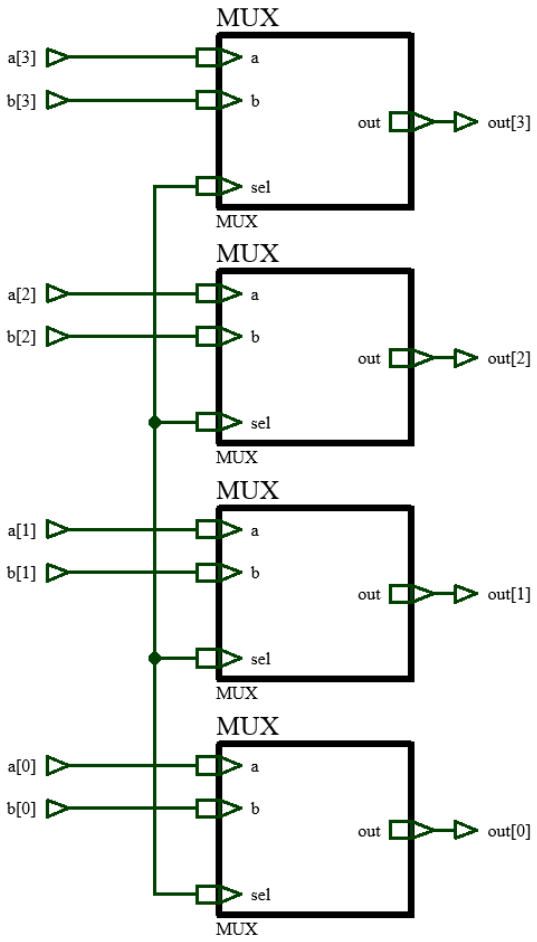

Figure A-10

Figure A-11: A 4-way OR gate

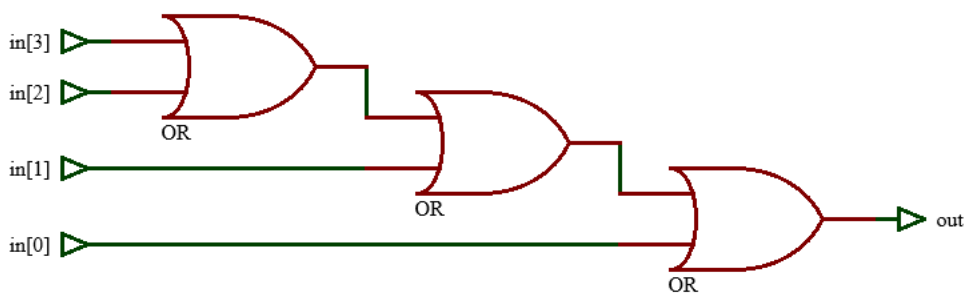

Figure A-11

Figure A-12: A 4-input, 16-bit multiplexer

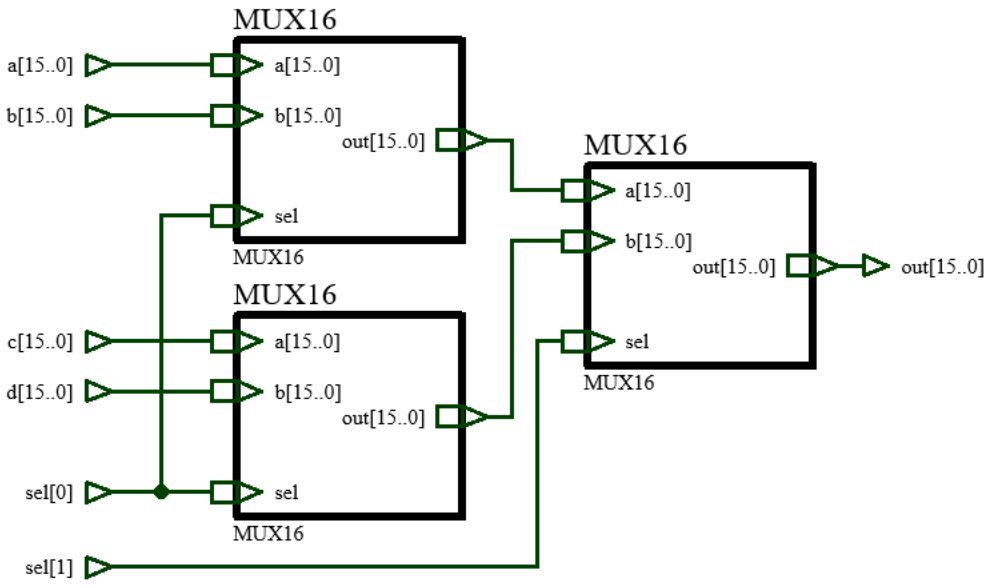

Figure A-12

Figure A-13: An 8-input, 16-bit multiplexer

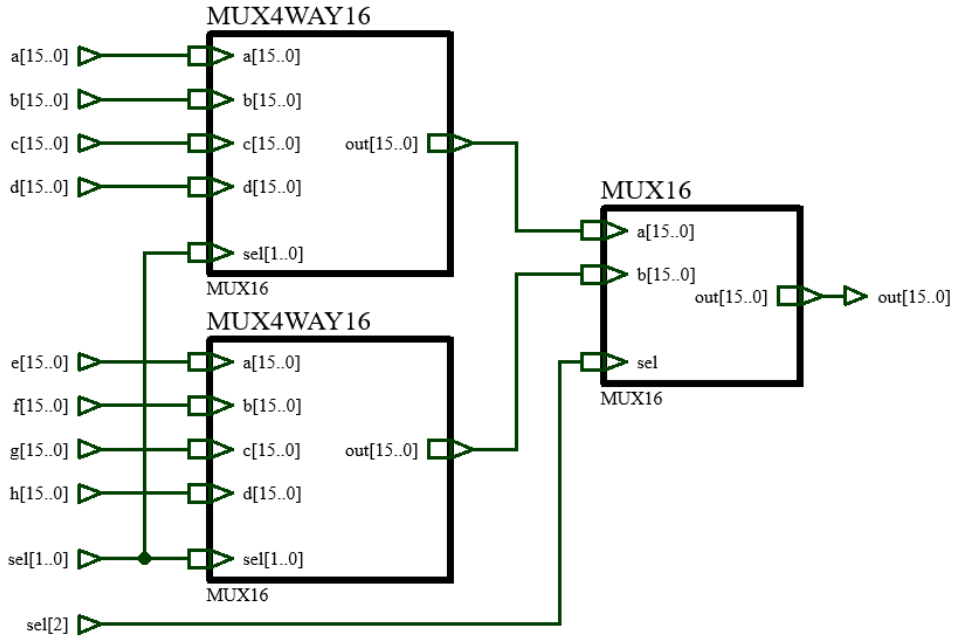

Figure A-13

Figure A-14: A 4-output demultiplexer

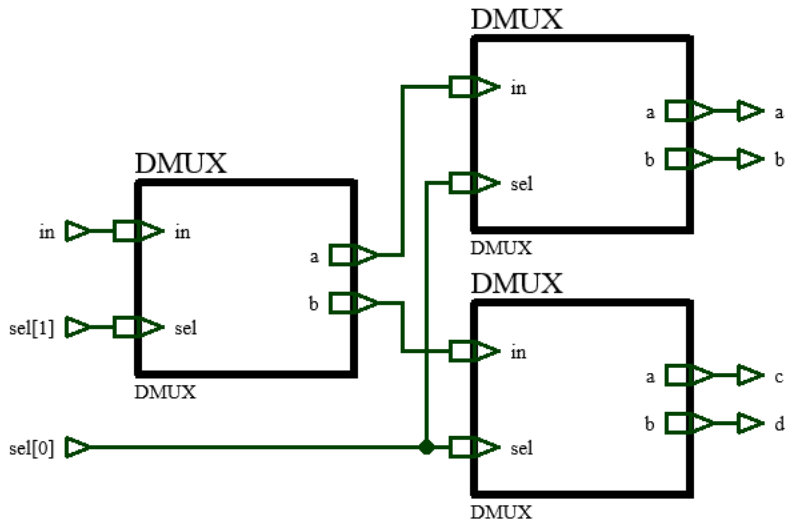

Figure A-14

Figure A-15: An 8-output demultiplexer

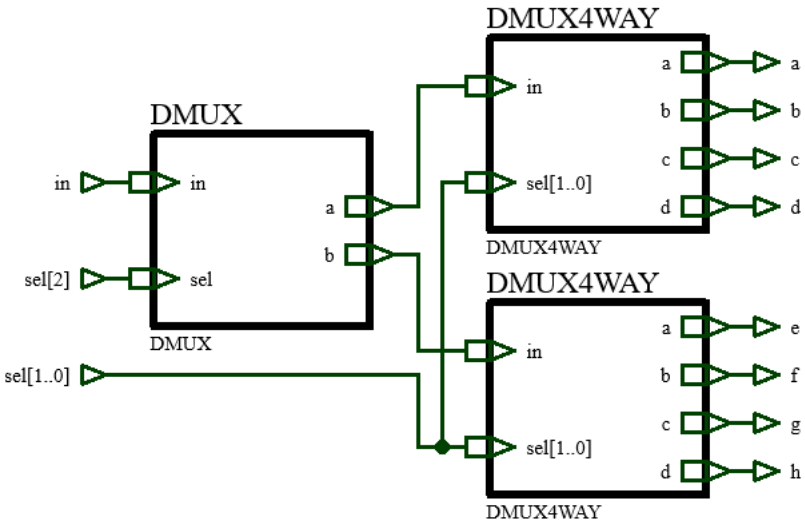

Figure A-15

Figure A-16: Half adder

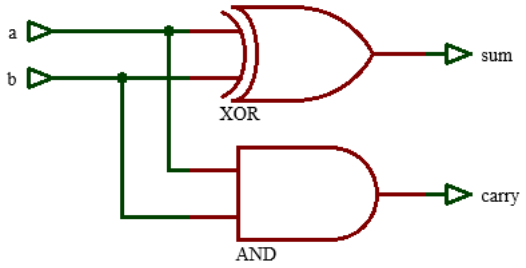

Figure A-16

Figure A-17: Full adder

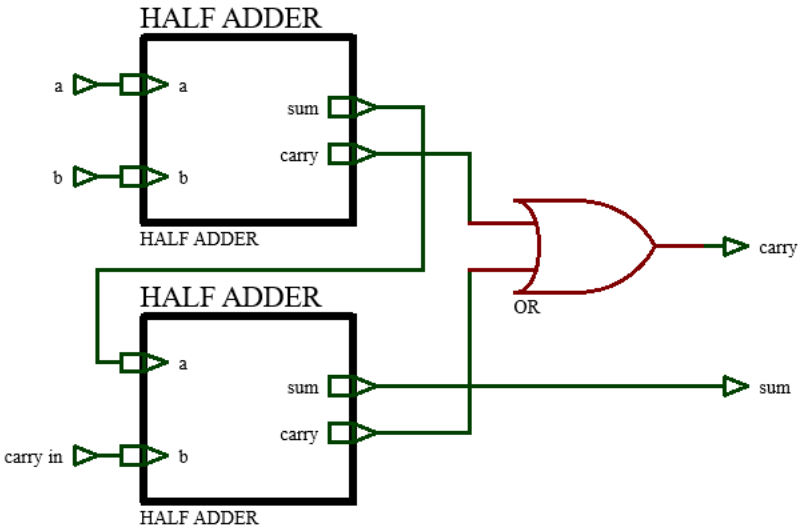

Figure A-17

Figure A-18: A 4-bit adder

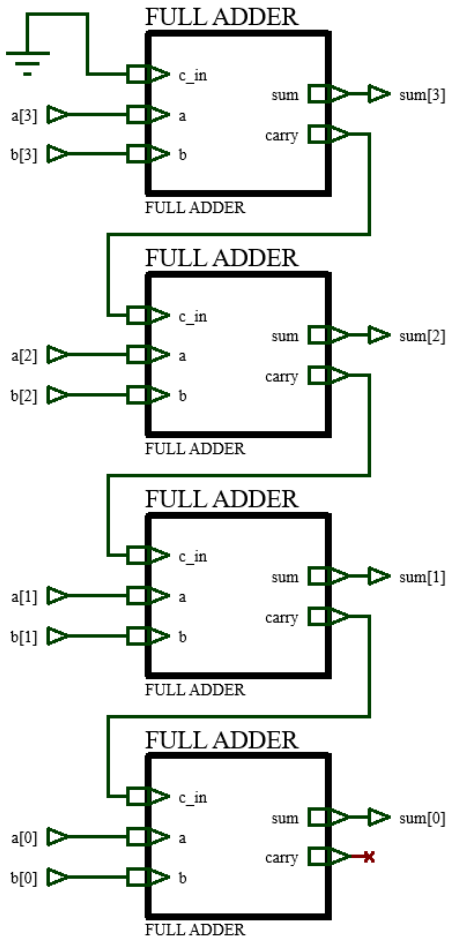

Figure A-18

Figure A-19: A 16-bit incrementor

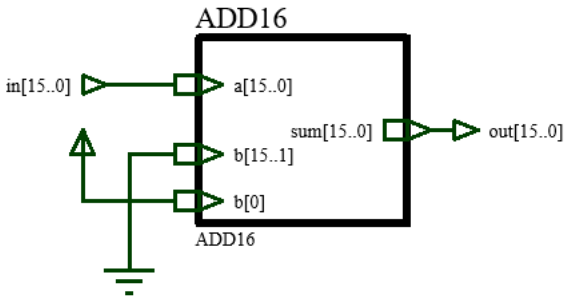

Figure A-19

Figure A-20: The Hack ALU

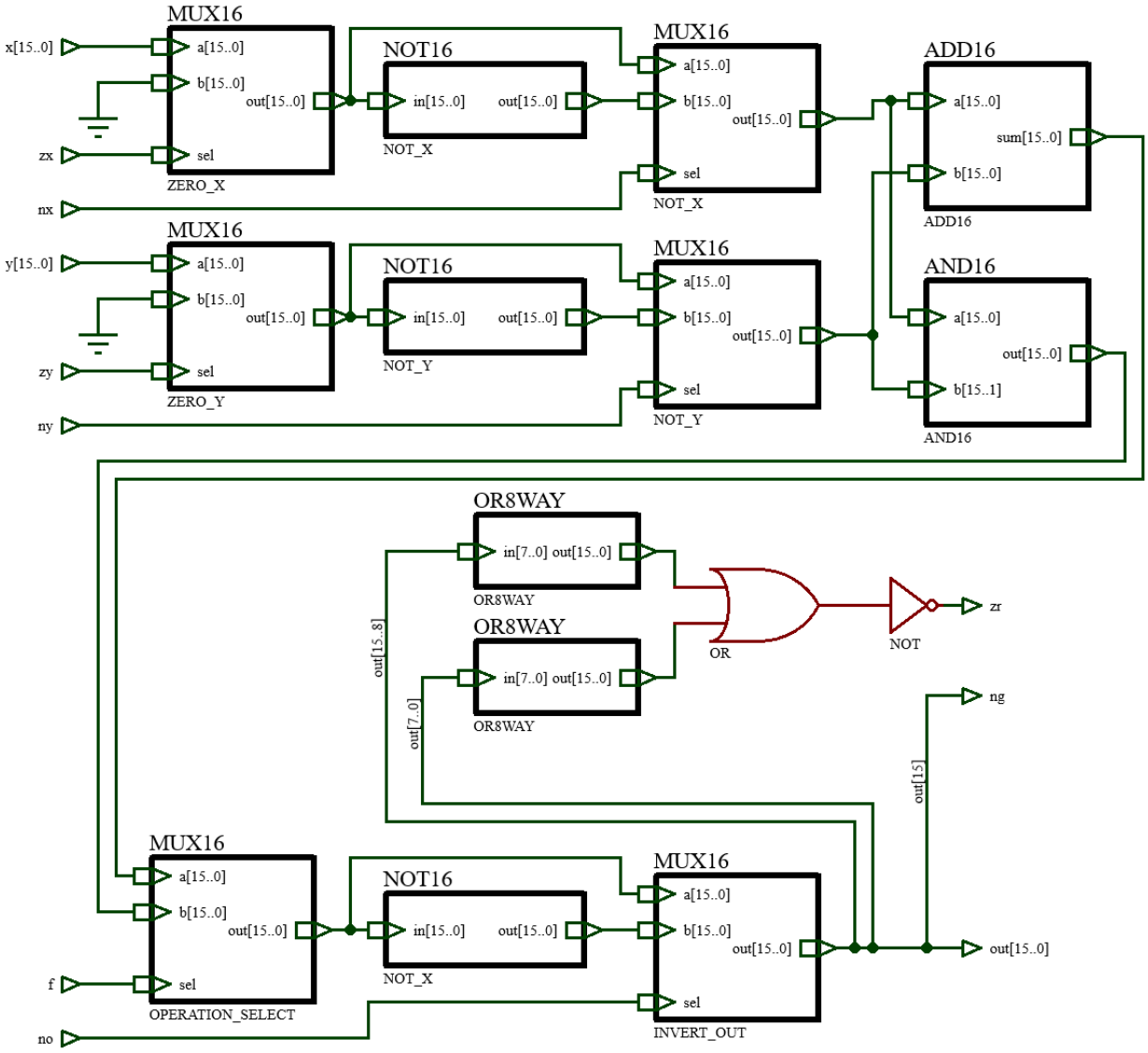

Figure A-20

Note: Due to the repetitive nature of the implementation of the memory modules, only the *Bit*, *Register* and top-level *Memory* modules are shown.

Figure A-21: 1-Bit Register

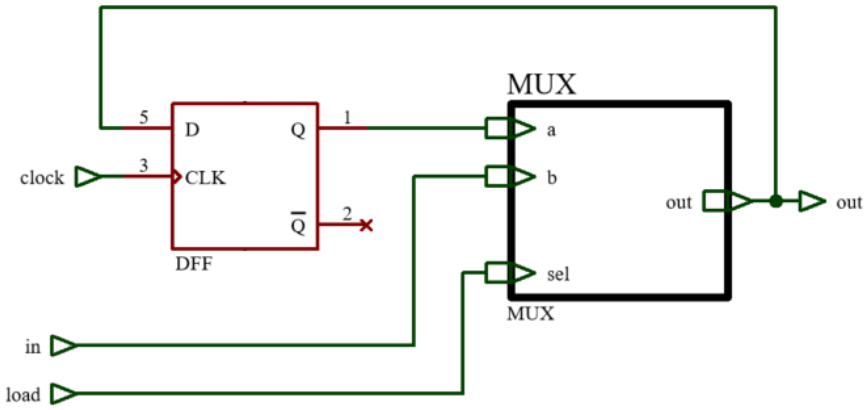

Figure A-21

Figure A-22: Register (4-bit)

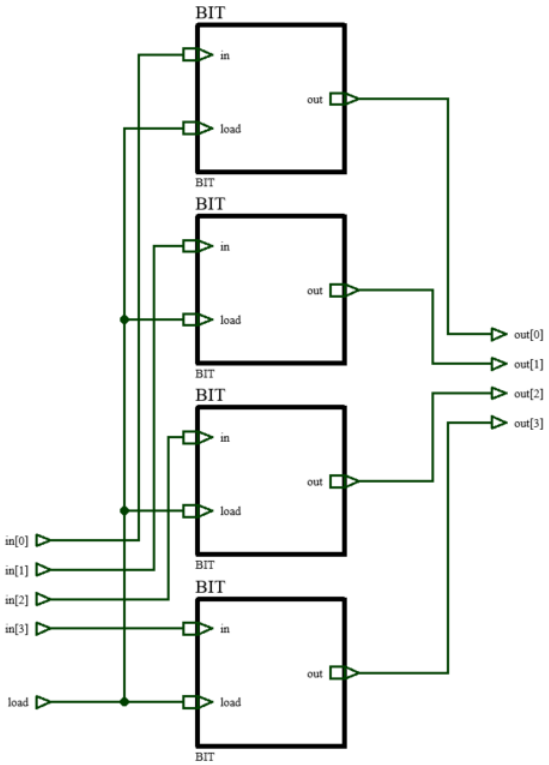

Figure A-22

Figure A-23: Program Counter

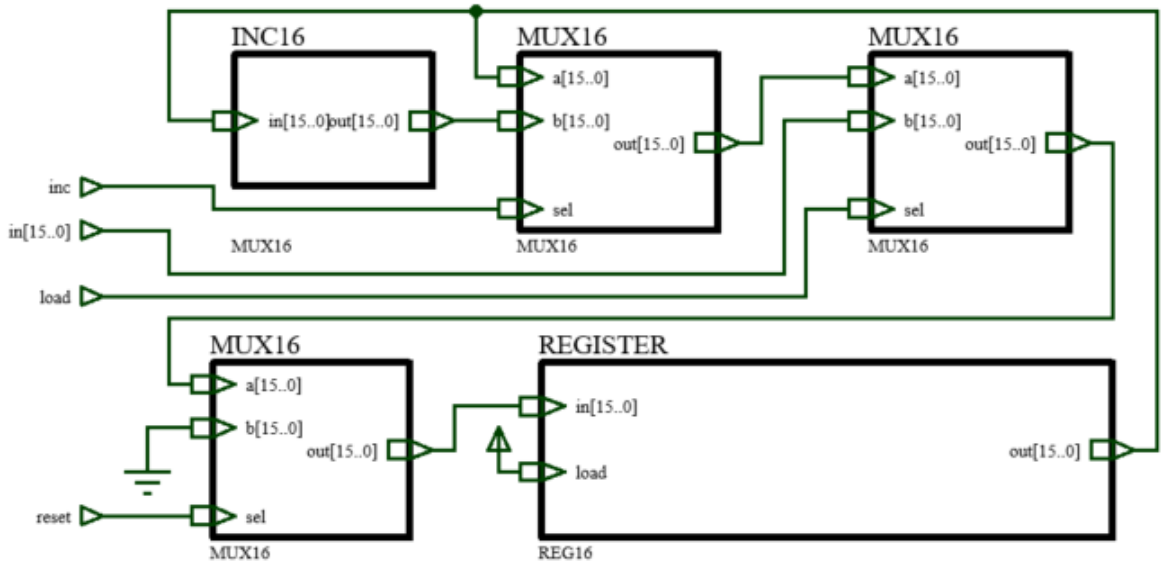

Figure A-23

Figure A-24: Hack Memory

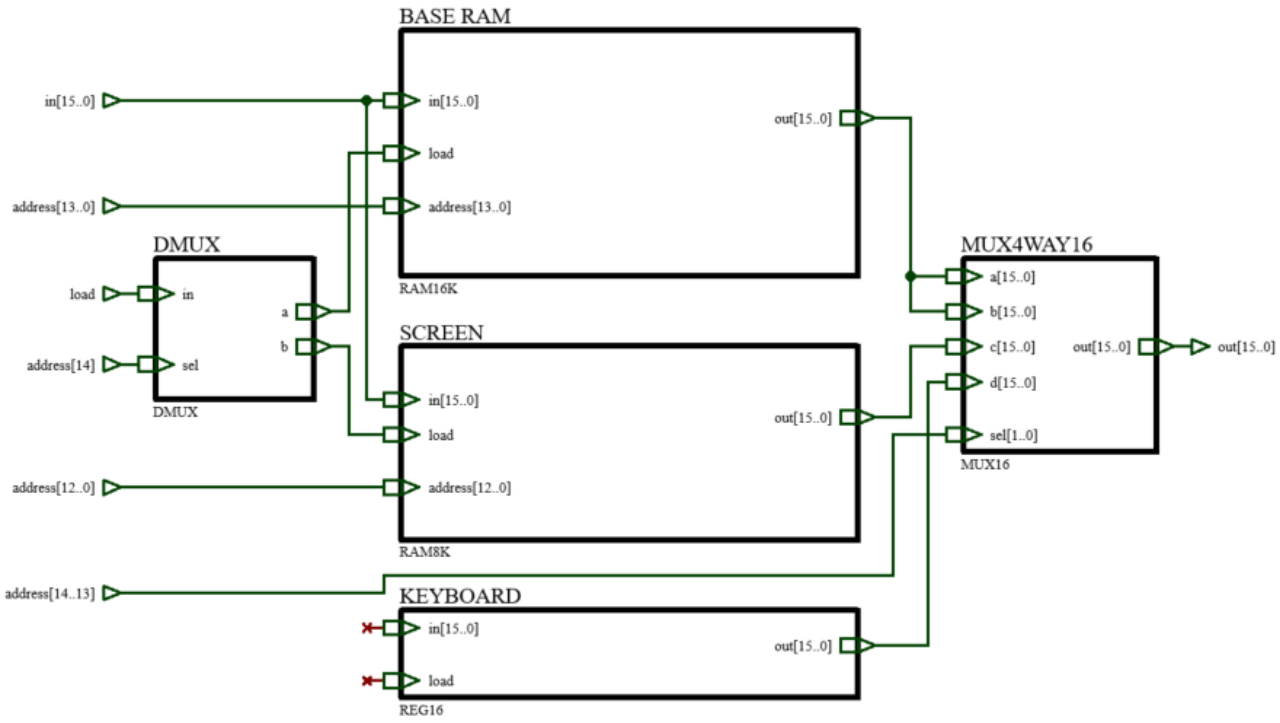

Figure A-24

Figure A-25: Hack CPU

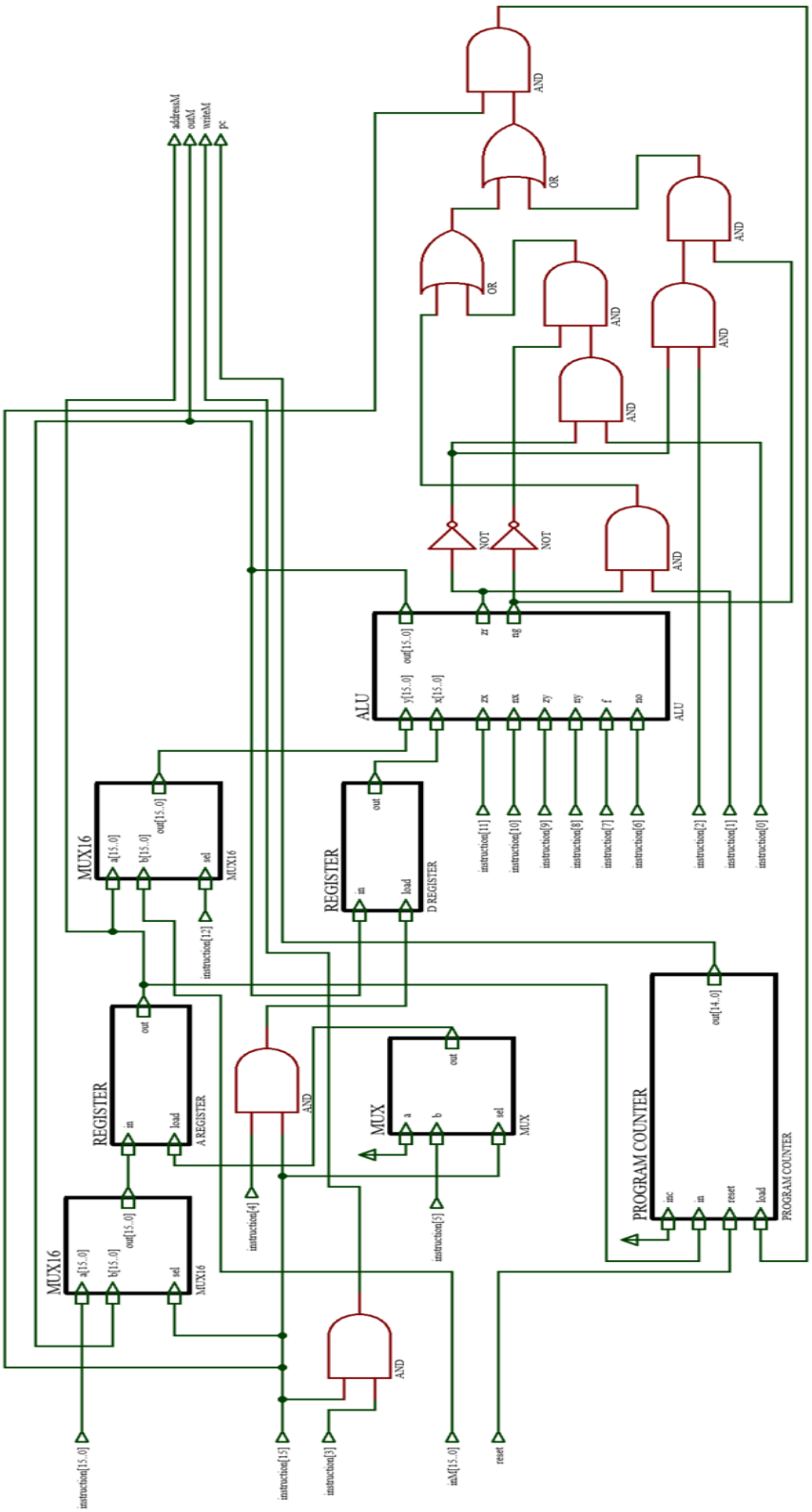

Figure A-25

Figure A-26: Hack Computer

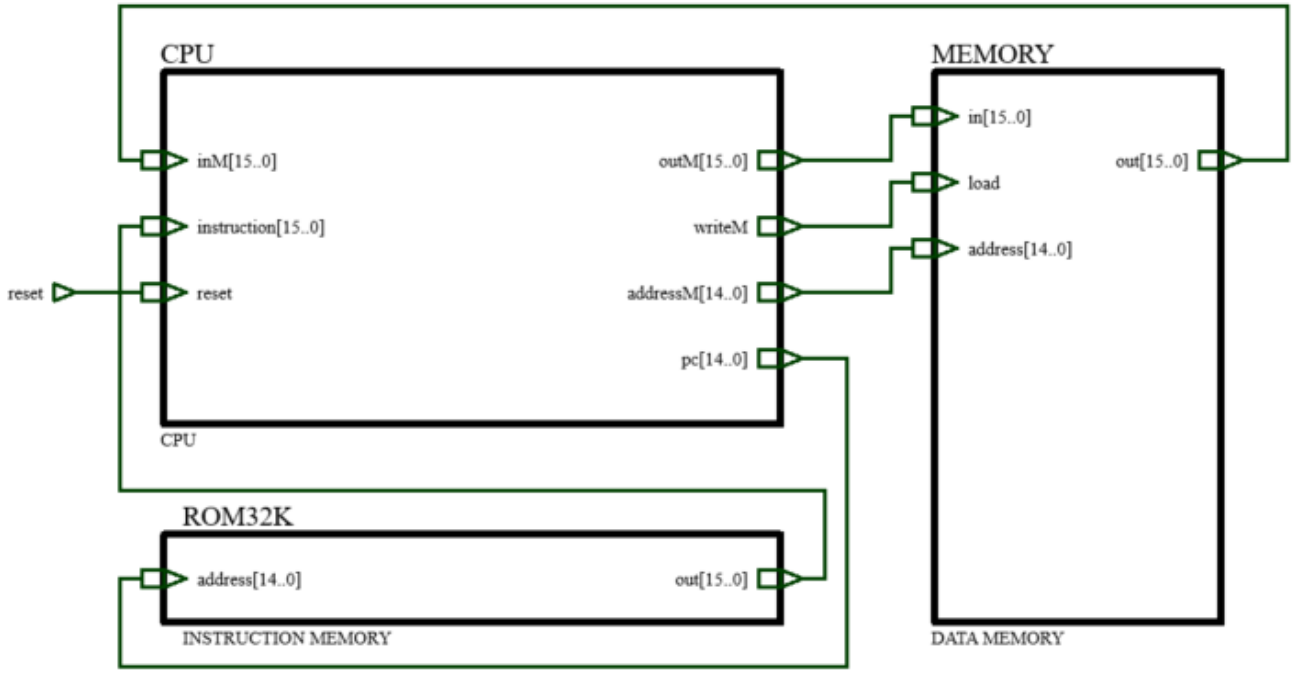

Figure A-26

Appendix B - Simulation Diagrams

Mult.asm

The following diagrams walk through the simulated execution of Mult.asm by the Hack computer. Mult.asm was modified slightly to initialize the two values to be multiplied in RAM[0] and RAM[1] at the start of the program.

Figure B-1 shows the initialization of RAM[0] to the value of 50 and RAM[1] to the value of 5. The instructions are shown in hexadecimal notation and all other values are shown in decimal for readability.

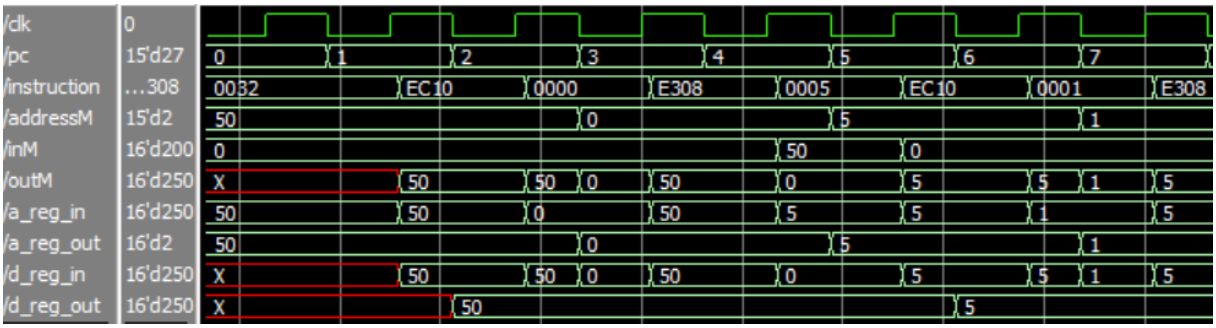

Figure B-1

Table B-1 describes the steps taken by the CPU during each instruction.

Instruction		Action Taken by CPU
Assembly	**Hexadecimal**	
@50	0032	Loads constant 50 into A-reg
D=A	EC10	Loads constant 50 into D-reg
@0	0000	Loads 0 into A-reg, results in access of RAM[0]
M=D	E308	Sets RAM[0] equal to 50 (contents of D-reg)
@5	0005	Loads constant 5 into A-reg
D=A	EC10	Loads constant 5 into D-reg
@1	0001	Loads 1 into A-reg, results in access of RAM[1]
M=D	E308	Sets RAM[1] equal to 5 (contents of D-reg)

Table B-1

The first 8 instructions of mult.asm load the values 50 and 5 into RAM[0] and RAM[1], respectively. These will be the two numbers multiplied together in the program.

Figure B-2 and accompanying Table B-2 show the next stage of the program. This set of instructions begins by declaring an index variable, *i*, and initializing it to zero. Then the contents of RAM[0] and RAM[1] are checked to determine if either is zero. If either of these values are zero, the multiplied number would also be zero and the program would jump to the (zero) label.

Lastly, RAM[2] is initialized to zero. The result of the multiplication calculation will be stored in RAM[2], so it is important to make sure that it starts at zero.

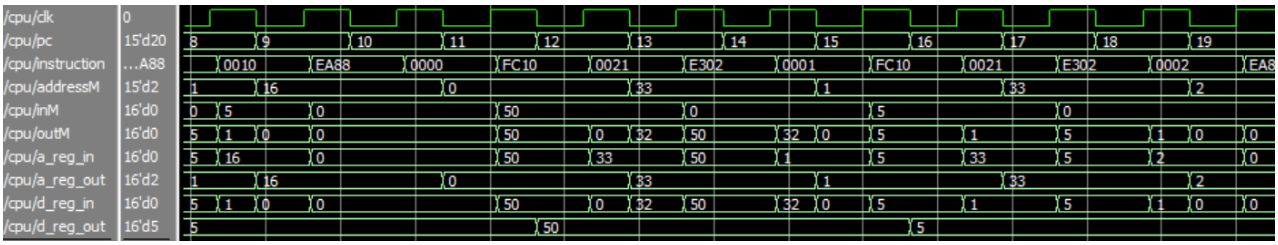

Figure B-2

Instruction		Action Taken by CPU
Assembly	**Hexadecimal**	
@i	0010	RAM[16] is used to stored variable *i*
M=0	EA88	*i* set to 0
@0	0000	Loads 0 into A-reg, results in access of RAM[0]
D=M	FC10	Sets D-reg to contents of RAM[0]
@zero	0021	Reference to a label at ROM[33] for subsequent jump condition
D;JEQ	E302	Jumps to (zero) if D=0
@1	0001	Loads 1 into A-reg, results in access of RAM[1]
D=M	FC10	Sets D-reg to contents of RAM[1]
@zero	0021	Reference to a label for subsequent jump condition
D;JEQ	E302	Jumps to (zero) if D=0
@2	0002	Loads 2 into A-reg, results in access of RAM[2]
M=0	EA88	Sets RAM[2] to zero

Table B-2

In this case, the two numbers being multiplied are 50 and 5, so the zero jump conditions are not met and the program continues to execute.

Figure B-3 and Table B-3 detail the main loop of mult.asm that makes up the repetitive addition algorithm.

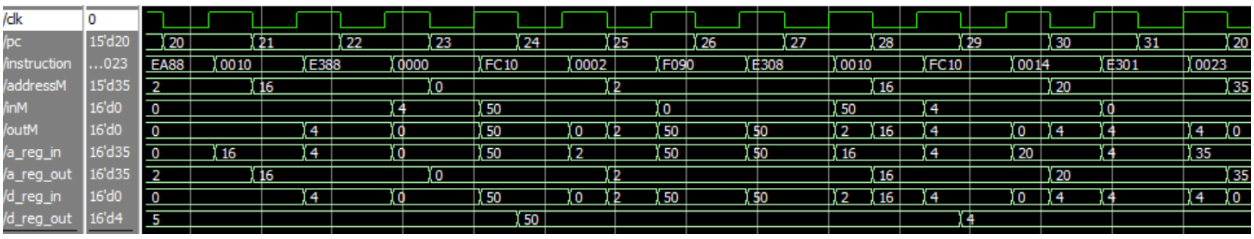

Figure B-3

Instruction		Action Taken by CPU
Assembly	**Hexadecimal**	
@i	0010	Accesses RAM[16]
M=D-1	E388	*i* set to contents of D-reg - 1
@0	0000	Loads 0 into A-reg, results in access of RAM[0]
D=M	FC10	Sets D-reg to contents of RAM[0]
@2	0002	Loads 2 into A-reg, results in access of RAM[2]
D=D+M	F090	Adds contents of RAM[2] with D-reg (RAM[0] and stores in D-reg
M=D	E308	Stores contents of D-reg in RAM[2]
@i	0010	Accesses RAM[16] (*i*)
D=M	FC10	Stores *i* in D-reg
@loop	0014	Label reference to ROM[20]
D;JGT	E301	Jump to ROM[20] if D-reg is greater than zero

Table B-3

The structure of mult.asm takes the value of RAM[0] and adds it to itself *n* times, where *n* is the value in RAM[1]. The value of *n* is used as the index to keep track of how many times RAM[0] must be added to itself. At the beginning of the loop stage, the D-register already contains the value of RAM[1]. One is subtracted from this value to indicate the first loop of addition and then the value is stored in the index variable. Then the contents of RAM[0] must be added to the running total in RAM[2], with the result stored in RAM[2]. After the first loop of addition shown in Figure B-3, it is seen that at one clock cycle following the E308 (M=D) instruction, *inM* is 50. This indicates that RAM[2] was properly set to the value 50. After the addition is completed, the D-register is updated with the index value and used to evaluate a jump condition: If the index value is greater than zero, more addition must be done and the program jumps back to the beginning of the loop (ROM[20]).

To multiply 50 and 5, mult.asm will execute the loop 5 times. Since the D-register is updated with the current index value at the end of the loop, the index can be properly decremented at the beginning of each loop. The end of loop 2 would result in RAM[2] = 100, loop 3 would result in RAM[2] = 150, and so on. Figure B-4 shows the last iteration of the loop.

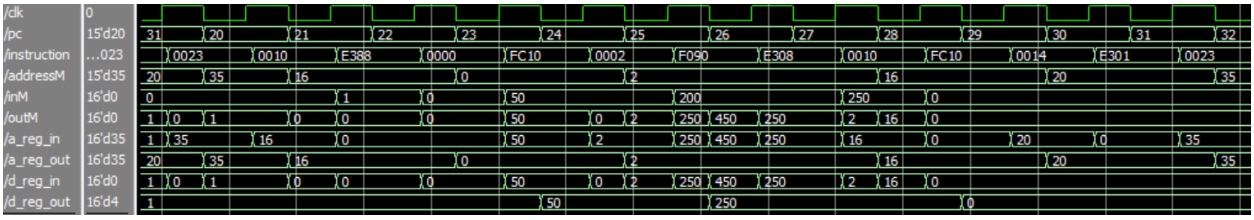

Figure B-4

It is seen that after the final loop of addition, the proper value of 250 is show in RAM[2] indicating the multiplication operation was successfully carried out. Note at the end of the final loop, the index stored in the D-register has reached zero. This no longer meets the jump condition of the loop, so the program continues.

Figure B-5 and Table B-4 show the end of the program.

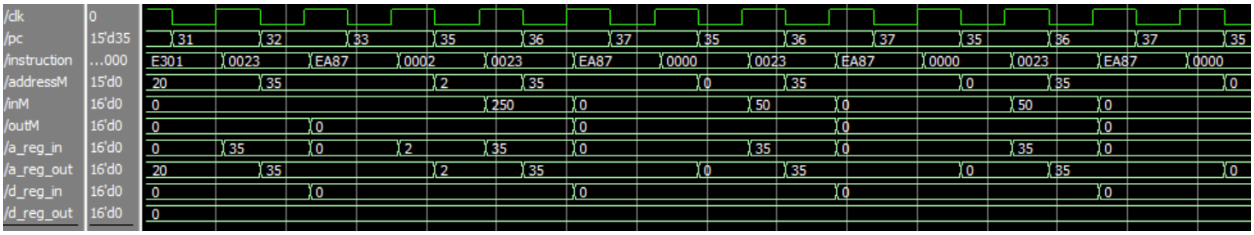

Figure B-5

Instruction		Action Taken by CPU
Assembly	**Hexadecimal**	
@end	0023	Reference to end label, ROM[35]
0;JMP	EA87	Unconditional Jump
@2	0002	
M=0	NA	
@end	0023	Reference to end label, ROM[35]
0;JMP	FC10	Unconditional Jump

Table B-4

The @end and 0;JMP commands directly following the loop are necessary to skip over the two following commands and reach the end loop. The @2 and M=0 commands are referenced by the zero label. These commands are only executed if RAM[0] or RAM[1] is zero and serve to set the result in RAM[2] to zero. The final @end and 0;JMP commands make up the end loop. The program ends by looping through these last two commands until it is reset.

VGA Timing and Hello.asm

The Hello.asm program consists of over 300 lines of code- too long to break down line by line. However, when executed, Hello.asm display “Hello, World” on a monitor when connected to the microcomputer via VGA cable. To display text, the visible screen is partitioned into 8x16 pixel tiles, which each can display a single character. Figure B-6 shows one tile for the letter “H” and the corresponding data values.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	1	1	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure B-6

In the Hack specification, a pixel value of 0 sets it to white and a pixel value of 1 sets it black. This same convention was used to write Hello.asm, however each word was inverted so the text would appear as black on a white background on the physical VGA monitor. Additionally, the data must be read from right to left for each word holding color data. This is necessary because the screen position counters count up and the registers are counted through in a downward fashion.

One horizontal line of two adjacent tiles is written with one 16 bit word. Therefore, two tiles can be written using 16 16-bit words. The screen RAM registers are all initialized to -1, providing a white background. Character tile maps like the one in Figure B-6 were created for each character needed to display “Hello, World”. Hello.asm simply writes each character tile pair line-by-line to the proper location in memory. Once a tile is completed, the next one is written until all the text is displayed. The display map interface is then responsible for reading from these memory locations and displaying the data on screen. The proper execution of this program illustrates the functionality of the VGA controller and display map modules.

Figure B-7 shows one full horizontal count through a single line on the screen.

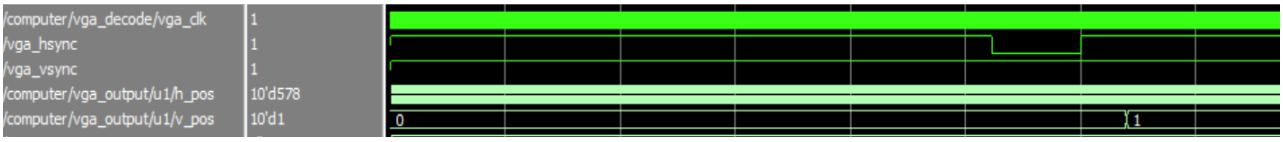

Figure B-7

The sync pulse of the *h_sync* signal can be seen shortly before the horizontal count resets and the vertical count is incremented.

Figure B-8 shows an expanded view of the horizontal count reset.

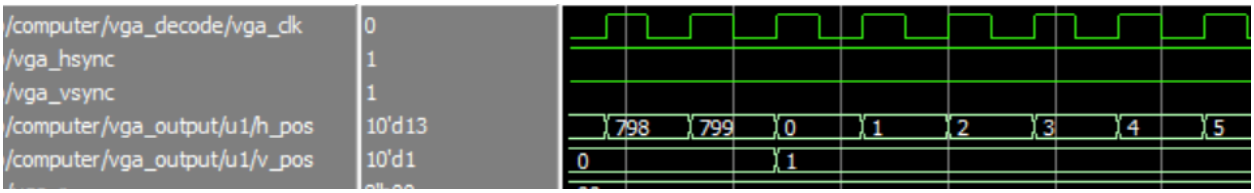

Figure B-8

Figure B-9 shows a zoomed-out view of the sync and data signals.

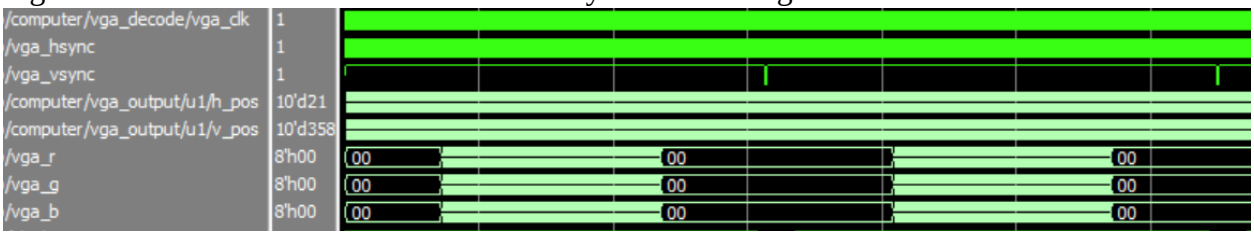

Figure B-9

The *v_sync* pulses indicate a screen refresh. The portion of the waveform where color data is not zero makes up the visible area of the screen. The color data is deactivated during the non-visible portion of the screen.

Figure B-10 shows the color data output for a single horizontal line on the screen.

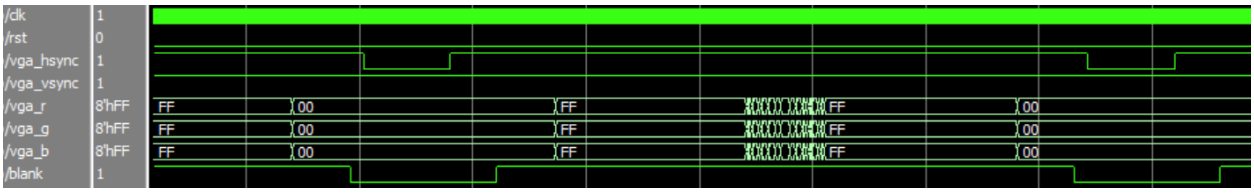

Figure B-10

It is seen that the color data is zeroed before the blanking portion of the screen, this provides the scaled border for the 512x256 screen. The visible screen is set to FF, or -1 which displays a white background screen. The data displayed on the screen sets certain values to zero to display black text on the screen.

Appendix C – Code

Mult.asm

Fill.asm

Hello.asm

Assembler

Binary to .mif