LAB 4: Basic Computer Organization

Course Coordinator: Dr. Voicu Groza

CEG 2136 - Computer Architecture I

Laboratory Section: B01

University of Ottawa

Teaching Assistants: Ayman
Daksh
Surbhi

Group #10

Yuqing Ma 300325103
Mai Anh Hoang 300278143

Experiment Date: November 8, 2023
November 15, 2023
November 22, 2023
November 29, 2023

Submission Date: December 6, 2023

Table of Contents

Index

Lab Theory

I. Introduction

The goal of the lab is to analyze the structure of a basic computer, devise and design its control unit, as well as use opcodes to write simple programs in machine code. The design should work in simulation and on the Altera board.

II. Discussion of the Problem

For part one of the lab, we were required to complete pre-lab questions by analyzing the provided .bdf files to make sense of the hierarchy of the files and to gain a greater understanding of a basic computer. The questions for 5.1 were answered, and the answers are provided in the Appendix section of the lab report.

For part 5.2, we were to derive control signals that would be generated by the computer control unit, which is a component that controls all the activities that occur in the basic computer. Specifically, it can be said that the control unit controls the CPU datapath, bus and memory. Typically, in a computer, these are functional units that work together to execute the desired micro-operations. The CPU datapath consists of these functional units, as it includes registers, the bus and memory units. Also, it is important to note that control signals are basically inputs of the functional units of the datapath, directing these datapath components to successfully execute instructions. Below is a diagram outlining the key components of the basic computer described. Afterwards, we were expected to design a control unit in accordance with the control signal inputs derived.

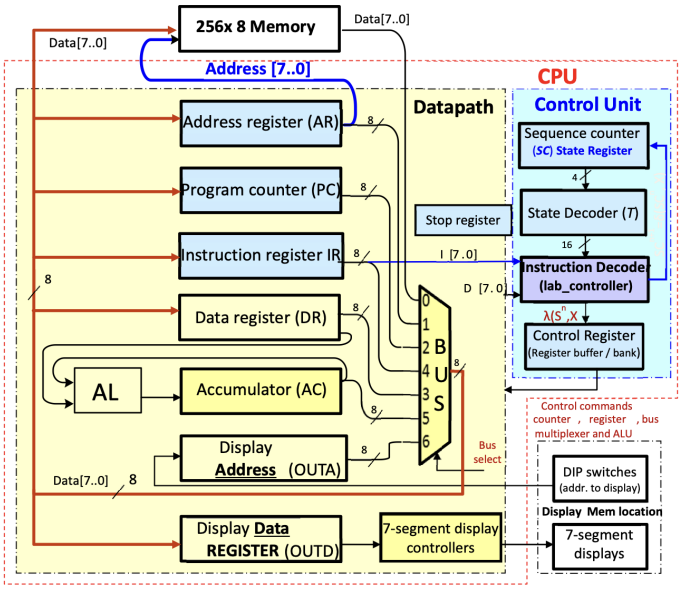


Figure 1: Basic Computer Block Diagram

For part 7.1, we were required to analyze the program outlined below, and gain a working understanding of the values from the AC, also known as the accumulator (which is a processor register in the CPU datapath, primarily used to store operands after executing an operation on them). The instructions reference symbols, such as LDA, SMA etc, which indicate they are RTL instructions, and are Register Transfer Language instructions. RTL describes the transfer of data between registers, essentially directing the functional units of a datapath to perform the desired micro operation.

```
%  PROGRAM IS IN the RANGE Of ADDRESSES 00 TO 7F %
00: 04;   %  LDA (direct)  %
01: a0;   %  from address a0 %
02: 42;   %  CMA  %
03: 08;   %  STA (direct)  %
04: a0;   %  AC to address a0  %
05: 20;   %  ISZ (direct)  %
06: a0;   %  counter stored at a0  %
07: 10;   %  BUN (direct)  %
08: 20;   %  to address 20  %
09: 60;   %  HLT  %
20: 84;   %  LDA (indirect)  %
21: a1;   %  the number pointed to by the memory location a1  %
22: 82;   %  ADD (indirect)  %
23: a2;   %  the number pointed to by the memory location a2  %
24: 88;   %  STA (indirect)  %
25: a3;   %  to the memory location pointed to by a3  %
26: 04;   %  LDA (direct)  %
27: a1;   %  from the address a1  %
28: 50;   %  Inc  %
29: 08;   %  STA (direct)  %
2a: a1;   %  AC to the memory address a1  %
2b: 50;   %  Inc  %
2c: 08;   %  STA (direct)  %
2d: a2;   %  store AC to the memory address a2  %
2e: 50;   %  Inc  %
2f: 08;   %  STA (direct)  %
30: a3;   %  send AC to the memory address a3  %
31: 10;   %  BUN (direct)  %
32: 05;   %  to the memory address 05  %
80: 01;   %  DATA ARE FOUND AT ADDRESSES 80 TO FF  %
81: 01;
a0: 0a;   %  loop counter, which will be done 10 times  %
a1: 80;   %  pointer to the first number to be added  %
a2: 81;   %  pointer to the second number to be added  %
a3: 82;   %  pointer to memory location where result will be stored %
```

Figure 2: A simple program written in machine code that we were required to analyze

After analyzing the program and gaining a functional understanding of its workings, we were required to write pseudo code to describe the algorithm it follows, with necessary substitutions when referencing variables of memory addresses A0, A1, A2, and A3, as indicated in the table below.

Address	Name of variable
A0	Counter
A1	X
A2	Y
A3	Z

Table 1: Variables stored at Memory Addresses

For part 7.2, instead of being required to analyze a program and its functionality, we were required to design a program that consecutively added the following numbers: 21, B5, 37, 08, 5C, 84, A1, 1D, 72, FF, F6, 43, 03, A9, D4, 19, 31, D9, 47, 82, 14, 52, 07, CA, 04. Hexadecimal numbers are numbers in the number system with a base number of 16. The program was also required to detect when the sum would be 00, displaying the number the last number added, and finally stopping.

III. Discussion of the Algorithmic Solution

To successfully analyze, design and implement the control unit components, several crucial steps happened. For part 5.2, deriving the equation of the control signals generated by the control unit for the CPU datapath required analyzing the RTL expressions of the tables shown below. The instructions table was used to determine the order of processes happening within the datapath, as it shows what operations are occurring at a given time T and what components are involved.

State	Description	Notation RTL
T_0	Load the address register AR with the contents of OUTA	T_0: $AR \leftarrow OUTA$
T_1	Read memory location pointed to by AR to the *data output register* OUTD	T_1: $OUTD \leftarrow M[AR]$
T_2	▪ Load AR register with the ADDRESS of the *opcode* of the current instruction (PC) ▪ Increment PC (if the program is running, i.e., if *Stop* FF S = 0) to point to the address of the next byte to be read, which can be: ○ either the next instruction, if the current instruction is a *register-reference instruction* ○ or the 2nd byte of the current instruction, if it is a *memory-reference instruction*.	T_2: $AR \leftarrow PC$ T_2S': $PC \leftarrow PC + 1$
T_3	Read the instruction's first byte (*opcode*) from memory location specified by AR to IR	T_3 : $IR \leftarrow M[AR]$
T_4	This state is a delay that allows the *opcode* to be decoded in the *Control Unit*.	(nothing)
T_5	If X_1, the byte in IR is a *register - reference instruction* and will be executed now	T_5X_1: execute instruction from Table 3 T_5X_1 :$SC \leftarrow 0$
	If X_0 or X_2 (*memory - reference instruction*), ▪ PC is copied to AR, i.e, the address of the instruction's 2nd byte goes from PC to AR => now AR contains the ADDRESS of ○ the *operand address* if *direct addressing*, or ○ the **address** *of the operand address* if *indirect addressing* ▪ Increment PC if the program is running (*Stop* FF S=0). Note that $(X_0 + X_2) = \overline{IR_6}$.	$T_5IR'_6$: $AR \leftarrow PC$ $T_5IR'_6S'$: $PC \leftarrow PC + 1$
T_6	Read from memory location pointed to by AR to AR; the read byte is ▪ the *operand address*, if *direct addressing*, or ▪ the **address** *of the operand address, if indirect addressing*	$T_6IR'_6$: $AR \leftarrow M[AR]$
T_7	If *indirect addressing*, read the *operand address* from memory location pointed to by AR	T_7X_2 : $AR \leftarrow M[AR]$
	If *direct addressing*, don't do anything, as the operand's address is already in AR since T_6	T_7X_0 : (nothing)
T_8 & after	Execute the ***memory - reference instruction*** as described in Table 4.	(see Table 4)

Table 2: Instruction Fetch Cycle

Symbol	RTL Notation
CLA	$T_5X_1IR_0 : AC \leftarrow 0$
CMA	$T_5X_1IR_1 : AC \leftarrow \overline{AC}$
ASL	$T_5X_1IR_2 : AC \leftarrow \text{ashl } AC$
ASR	$T_5X_1IR_3 : AC \leftarrow \text{ashr } AC$
INC	$T_5X_1IR_4 : AC \leftarrow AC + 1$
HLT	$T_5X_1IR_5 : S \leftarrow 1$

Table 3: RRI

Symbol	RTL Notation
AND	$T_8Y_0 : DR \leftarrow M[AR]$ $T_9Y_0 : AC \leftarrow AC \wedge DR, SC \leftarrow 0$
ADD	$T_8Y_1 : DR \leftarrow M[AR]$ $T_9Y_1 : AC \leftarrow AC + DR, SC \leftarrow 0$
SUB	$T_8Y_2 : DR \leftarrow M[AR]$ $T_9Y_2 : AC \leftarrow AC - DR, SC \leftarrow 0$
LDA	$T_8Y_3 : DR \leftarrow M[AR]$ $T_9Y_3 : AC \leftarrow DR, SC \leftarrow 0$
STA	T_8 : (cycle not allocated to allow the address bus to stabilize) $T_9Y_4 : M[AR] \leftarrow AC, SC \leftarrow 0$
BUN	$T_8Y_5 : PC \leftarrow AR, SC \leftarrow 0$
ISZ (assuming that the next instruction is a memory-reference instruction, stored at 2 memory location further down)	$T_8Y_6 : DR \leftarrow M[AR]$ $T_9Y_6 : DR \leftarrow DR + 1$ $T_{10}Y_6 : M[AR] \leftarrow DR$ $T_{11}Y_6 : \text{si } (DR = 0) \text{ alors } (\overline{S} : PC \leftarrow PC + 1)$ $T_{12}Y_6 : \text{si } (DR = 0) \text{ alors } (\overline{S} : PC \leftarrow PC + 1), SC \leftarrow 0$

Table 4: MRI

Tables 3 and 4 are particularly relevant because they showcase crucial instructions performed during the execution cycle, indicating what operations with which functional components on the datapath occur at which state. RRI stands for register reference instruction, and MRI stands for memory reference instruction. These former are instructions mainly pertaining to registers (ie involving shift registers etc), whereas the latter involves instructions mainly pertaining to memory, such as reading/writing things from memory.

Thus, using *Table 2,3,4,* analyzing RTL from the instruction fetch cycle, and instruction execution cycle, we were able to derive control input signals and write them in logic expression form. Essentially, all instances of a certain control signal occurring were identified in the table (i.e. for memwrite, all the events where the RTL notation indicated that something had been written to memory M[AR] <- .. were observed, and the condition of that occurrence was noted as well: in this case that would be T_9Y_4 and $T_{10}Y_6$. These conditions were 'added' together (in a boolean context) to form the logical expression).

After these logic equations were derived, the control unit components were designed in accordance with the input signal logic equations, with appropriate logic gates and inputs, using Quartus software. Afterwards, a block diagram was completed showing the top-level entity of the computer design. Afterwards, the program was run in a simulation to obtain a waveform to verify that it was working properly.

For part 7.1, we interpreted the given program in accordance with our working knowledge of computer architecture concepts. Since the program was written via RTL symbols, we used RTL tables, such as Table 3, and Table 4 to understand what was happening with the program. Therefore, using pseudo code and our understanding of how the program was working at each step of the process, we were able to gain an accurate understanding of what the program was doing and concluded that it was calculating a Fibonacci sequence. To verify the results, a .mif file was created so that the value of the A1 and A2 pointers could be seen, as well as the values of what the other locations were pointing to. To verify its successful functionality, a simulation was run to ensure that the memory cells in the .mif file would have the right Fibonacci number stored in the right cell.

For part 7.2, to develop our own program in accordance with the specifications listed, we wrote a series of program processes, similar to how they were given to us in part 7.1. A list of instructions and their respective addresses was created, using RTL language, to indicate how the datapath would handle the instructions in order to perform the desired operations successfully. Essentially, the program was also written in machine code, and was virtually similar in structure compared to the program in Figure 2*,* except, under the heading "all of the data that would be added" were the numbers that were provided to us in the specifications. Essentially, this program

was designed by repurposing and reworking the machine code that we saw in part 7.1, making the necessary modifications where necessary to suit our purposes.

Lab Design

I. Presentation of design that solves lab problems

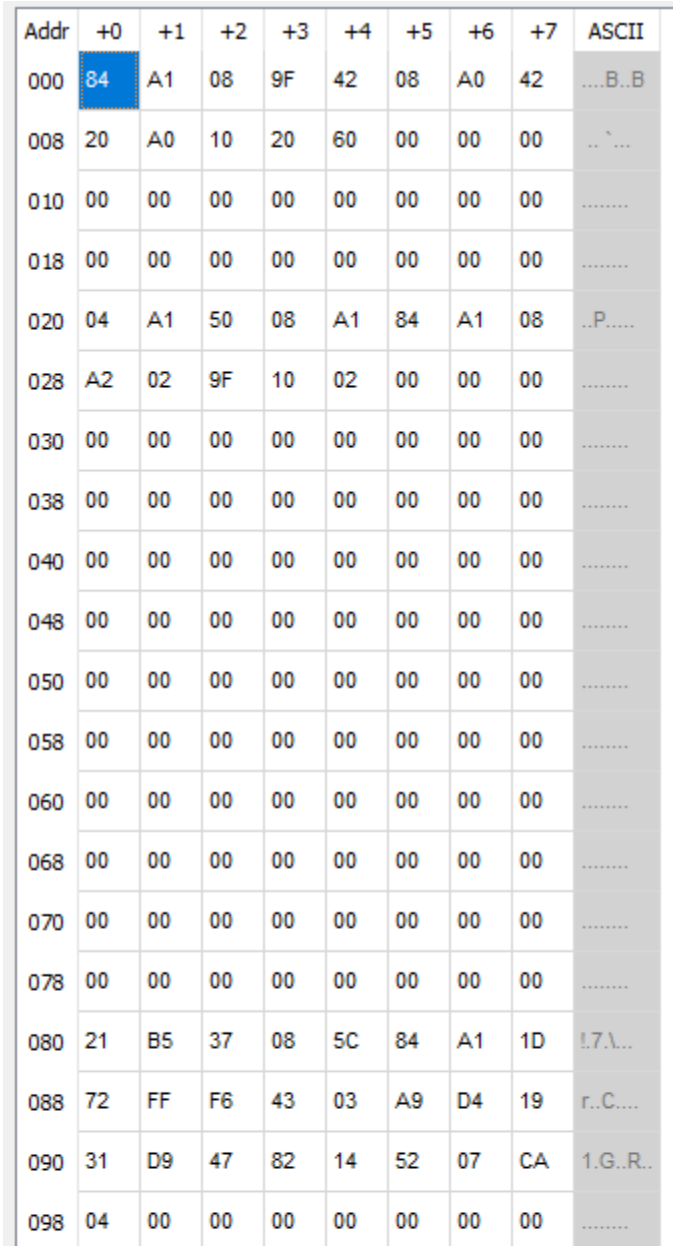
Firstly, we needed to derive the logical expressions of all the control signals by analyzing the RTL expressions in Table 2, Table 3, and Table 4. We then added the logic diagram for these equations into the lab3controller file to complete the circuit diagram.

Figure 3: Logical expressions of the control signals

1. $memWrite = T_9 Y_4 + T_{10} Y_6$
2. $AR\_Load = T_0 + T_2 + IR_6' (T_5 + T_6) + T_7 X_2$
3. $PC\_Load = T_8 Y_5$
4. $PC\_Inc = (T_2 + T_5 IR_6' + Y_6 (T_{11} + T_{12}) DR'_{(7...0)})$ Stop
5. $DR\_Load = T_8 (Y_0 + Y_1 + Y_2 + Y_3 + Y_6)$
6. $DR\_Inc = T_9 Y_6$
7. $IR\_Load = T_3$
8. $AC\_Clear = T_5 X_1 IR_0$
9. $AC\_Load = T_5 X_1 (IR_1 + IR_2 + IR_3) + T_9 (Y_0 + Y_1 + Y_2 + Y_3)$
10. $AC\_Inc = T_5 X_1 IR_4$
11. $OUTP\_Load = T_1$
12. $ALU\_Sel2 = T_9 (Y_0 + Y_3) + T_{10} Y_6 + T_5 X_1 IR_1$
13. $ALU\_Sel1 = T_5 X_1 (IR_1 + IR_2 + IR_3) + T_9 Y_3 + T_{10} Y_6$
14. $ALU\_Sel0 = T_5 X_1 (IR_1 + IR_3) + T_9 Y_2$
15. $BusSel2 = T_0 + T_9 Y_4$
16. $BusSel1 = T_{10} Y_6 + T_0 + T_2 + T_5$
17. $BusSel0 = T_{10} Y_6 + T_9 Y_4 + T_8 Y_5$
18. $SC\_clear = T_5 X_1 + T_{12} Y_6 + T_8 Y_5 + T_9 (Y_0 + Y_1 + Y_2 + Y_3 + Y_4)$
19. $Halt = T_5 X_1 IR_5$

For the second part of the lab, we were required to design a program that consecutively adds each number of the given sequence of hexadecimal numbers. Below is the program that we designed written in .mif file format.

Figure 4: .mif file for the program designed in 7.2



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	84	A1	08	9F	42	08	A0	42	....B..B
008	20	A0	10	20	60	00	00	00	.. `...
010	00	00	00	00	00	00	00	00	....
018	00	00	00	00	00	00	00	00	....
020	04	A1	50	08	A1	84	A1	08	..P.....
028	A2	02	9F	10	02	00	00	00	....
030	00	00	00	00	00	00	00	00	....
038	00	00	00	00	00	00	00	00	....
040	00	00	00	00	00	00	00	00	....
048	00	00	00	00	00	00	00	00	....
050	00	00	00	00	00	00	00	00	....
058	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	....
068	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	....
078	00	00	00	00	00	00	00	00	....
080	21	B5	37	08	5C	84	A1	1D	!7.\...
088	72	FF	F6	43	03	A9	D4	19	r..C....
090	31	D9	47	82	14	52	07	CA	1.G..R..
098	04	00	00	00	00	00	00	00	....

II. Discussion of used components

Regarding gates, AND gates, NOT gates, and OR gates were connected to create the circuit schematics for the control input signal input/output configurations. AND gates have the characteristic quality of only producing an output of one if both inputs are 1. NOT gates produce an output opposite of their respective input (ex, input = 1; output = 0). OR gates produce an output of 1 as long as there is at least one input that is equivalent to 1.

The respective components and gates can be seen below:

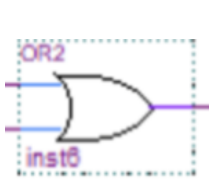


Figure 5: Quartus OR gate

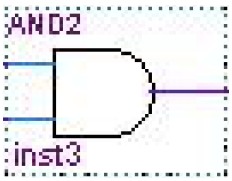


Figure 6: Quartus AND gate

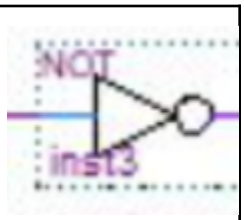


Figure 7: Quartus NOT gate

Other non-gate components were used in the block diagram for the computer. Here, a multitude of components were used. Registers, counters, a RAM component, controllers and multiplexers were used to create the necessary functional components for the CPU datapath, as shown below.

For the registers within the datapath, such as the output data register, data register, address register, output address register, these were represented and implemented using an 8 bit register. A register is used to store n bits of data using n flip flops. It also featured the necessary inputs, such as Clear, Load, Increment, Clk, etc.

Figure 8: 8-bit register

For the program counter and accumulator, an 8-bit counter was used. A counter counts clock pulses that occur, and in this case because an 8-bit counter was implemented, this indicates that it can count from 00000000 (also in decimal form: 0) to 11111111 (also in decimal form: 1). It is often composed of a multitude of flip flops connected to each other to perform its function. A 4-bit counter was also used as a sequence counter for the program overall.

Figure 9: 8-bit counter

For the memory, a RAM component of 256 x 8 was used. This is indicative of the computer's memory, which has a capacity of 256 words of 8 bits, thus the program has a memory space of 2^8 memory locations.

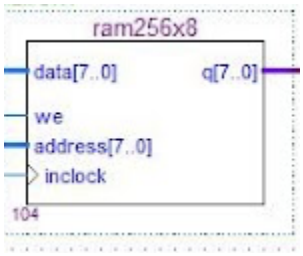


Figure 10: 256 x 8 ram

In addition, an 8 bit ALU was also used, connected to a counter, to create a full accumulator implementation. An ALU typically has the functionality of being able to perform arithmetic and logic operations, which can be chosen by the selection inputs.

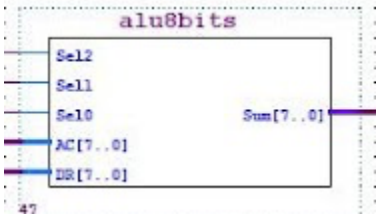


Figure 11: ALU 8 bits

Finally, there were also the 8 to 1 multiplexers, representing the bus. Essentially, the functional units on the datapath load their data into the bus, which are the inputs of the multiplexer. It is used to transfer data between these units, in essence. A multiplexer has multiple input lines, and selects a single input line to output, using selection line values. In addition, controllers were also used to control the inputs of the functional unit components, as can be seen, it has the necessary input values, and based on the combination of inputs, chooses a control signal instruction in the control register, which then sends the signal to the appropriate functional unit, as an input to that specific unit. In essence, a controller manages the flow of input data in a way that directs the components of the CPU datapath to function effectively and successfully.

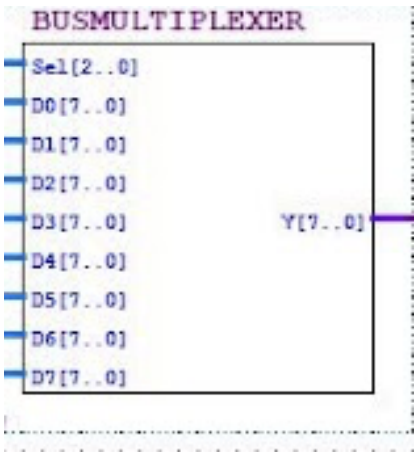


Figure 12: 8-to-1 Multiplexer

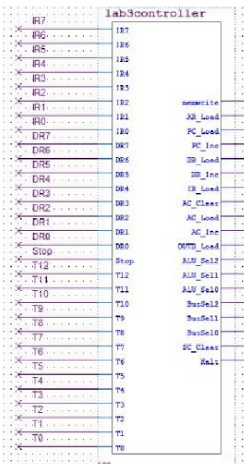


Figure 13: controller

III. Discussion of actual solution

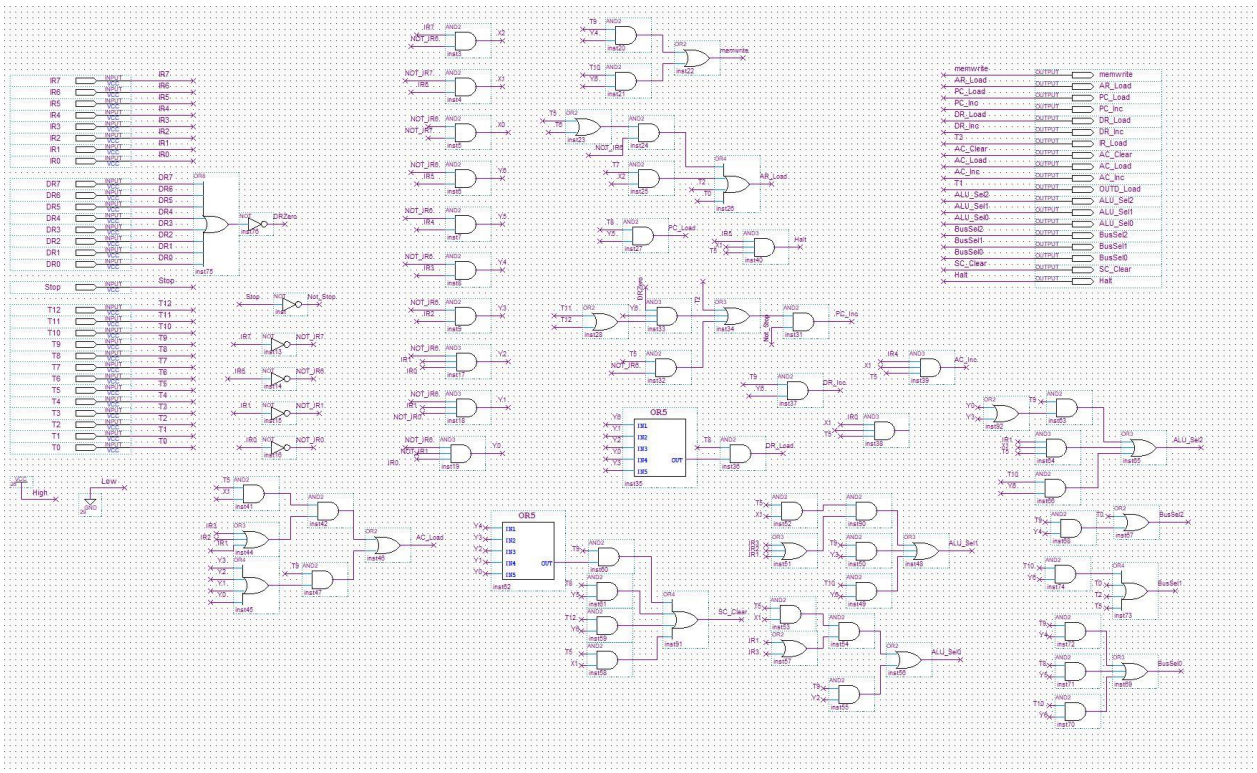


Figure 14: The block digram of the controller

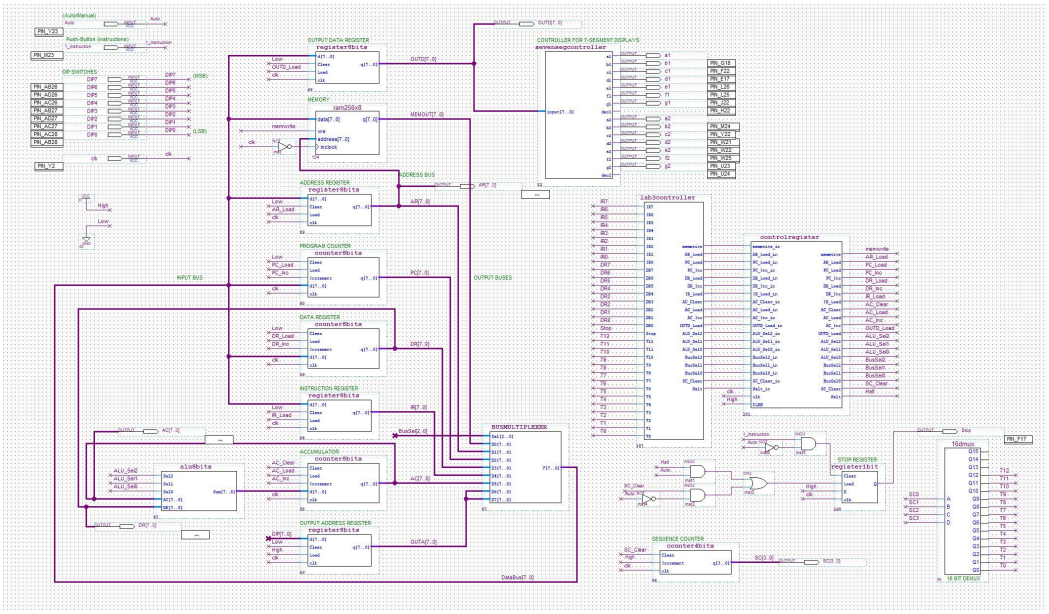


Figure 15: The block diagram of the top-level entity

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	84	A1	08	9F	42	08	A0	42	....B..B
008	20	A0	10	20	60	00	00	00	.. `...
010	00	00	00	00	00	00	00	00	....
018	00	00	00	00	00	00	00	00	....
020	04	A1	50	08	A1	84	A1	08	..P.....
028	A2	02	9F	10	02	00	00	00	....
030	00	00	00	00	00	00	00	00	....
038	00	00	00	00	00	00	00	00	....
040	00	00	00	00	00	00	00	00	....
048	00	00	00	00	00	00	00	00	....
050	00	00	00	00	00	00	00	00	....
058	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	....
068	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	....
078	00	00	00	00	00	00	00	00	....
080	21	B5	37	08	5C	84	A1	1D	!.7.\...
088	72	FF	F6	43	03	A9	D4	19	r..C....
090	31	D9	47	82	14	52	07	CA	1.G..R..
098	04	00	00	00	00	00	00	00	....

Figure 16: The designed program for 7.2 written in .mif file

As can be seen in the figures, the components were integrated into the circuit. The circuit schematics were constructed in accordance with the equations derived. The key components outlined earlier were placed first in the graphic editor and then connected to construct the correct configuration. Finally, the entire circuit was compiled and simulated. The correct pins were assigned to each input/output in accordance with lab instructions, then the circuit was connected to the Altera Board for testing, which allowed us to compare our experimental results to our theoretical results to verify functionality. Then, we created a .mif file and compiled and created a simulation for the second part of the lab.

IV. Discussion of Tools

A Quartus II 13.0 Service-Pack 1, Altera DE2-115 circuit board and computer were used.

V. Discussion of challenging problems

We experienced several problems during this lab, but we were able to solve them. During the demonstration of the circuit, at first, we couldn't get the Altera board to show the results we expected although we had the correct simulation waveform. However, thanks to the help of the TAs, we managed to get the correct result.

Simulation and Verification of Real Implementation

I. Simulation results

Figure 17: Hardware part Waveform Simulation with DIP switches pointing to A0

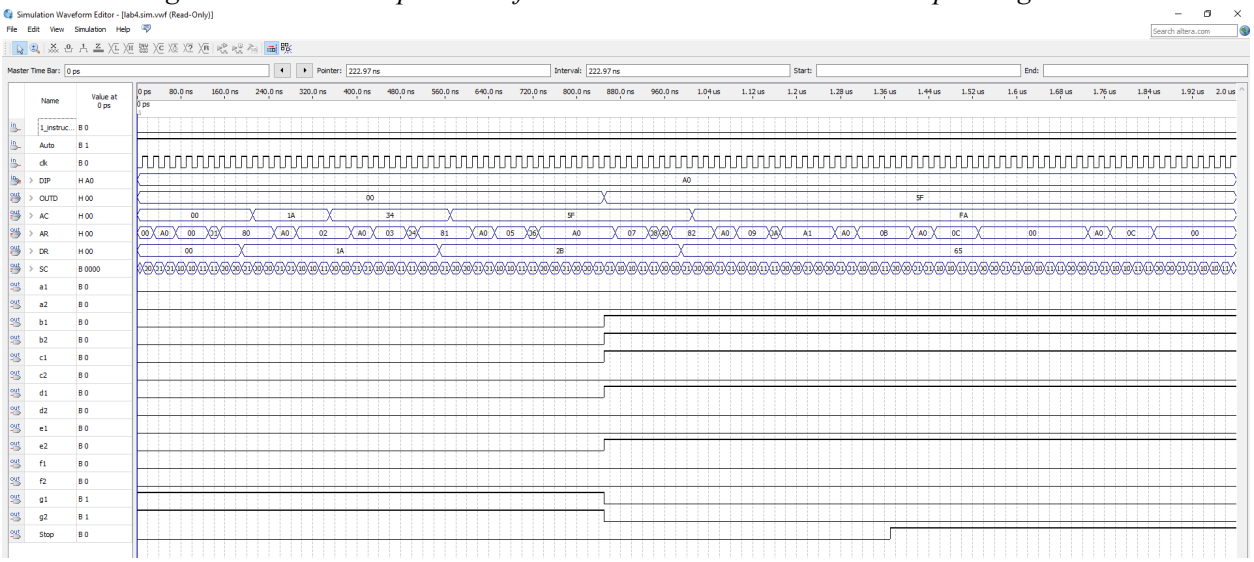


Figure 18: Hardware part Waveform Simulation with DIP switches pointing to A1

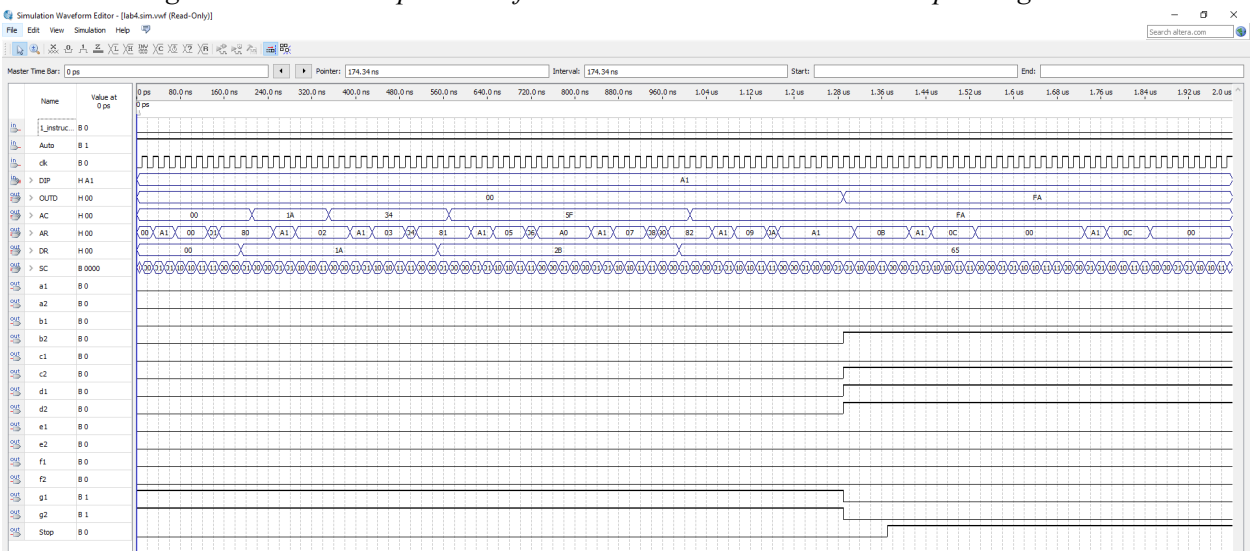
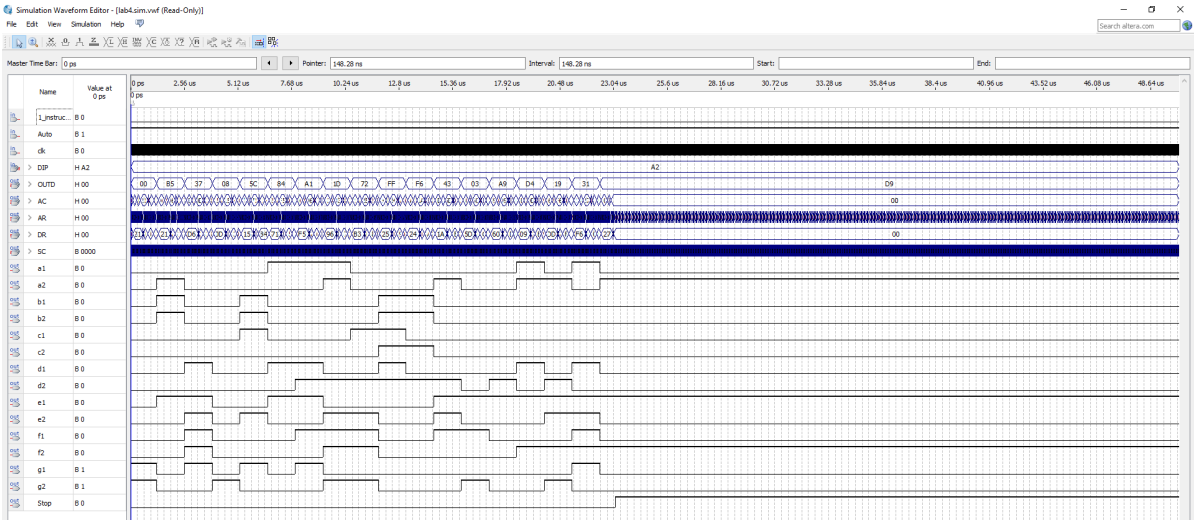


Figure 19: Software part Waveform Simulation



II. Experimental verification of the operation of the circuit

Figure 20: Output for hardware part with DIP switches pointing to A0

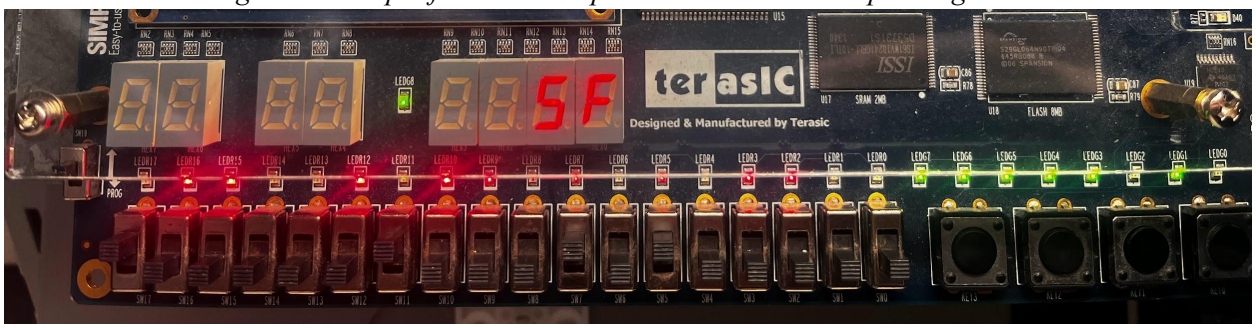


Figure 21: Output for hardware part with DIP switches pointing to A1

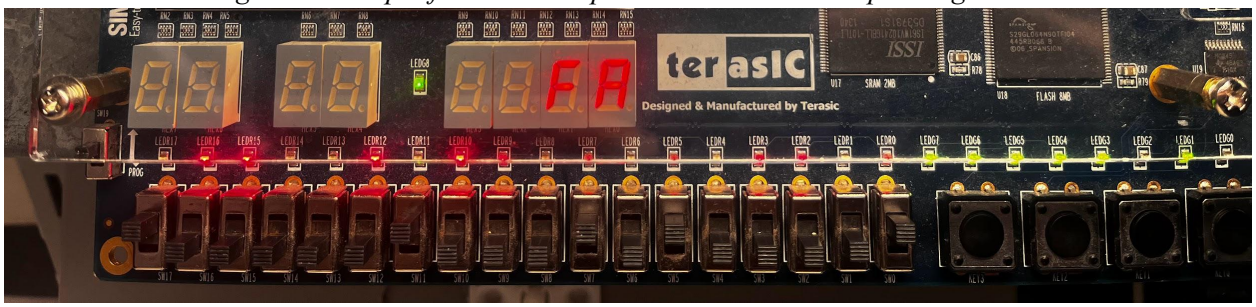
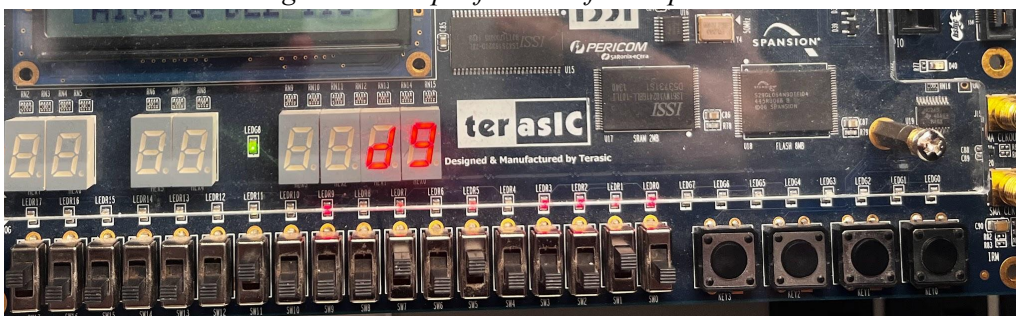


Figure 22: Output for the software part 7.2



Evidently, looking at the waveforms and the pictures of the experimental outputs given, it is evident that the experimental results were identical to the expected results. Therefore, we ended up with a working circuit design and program design that compiled and verified our results successfully.

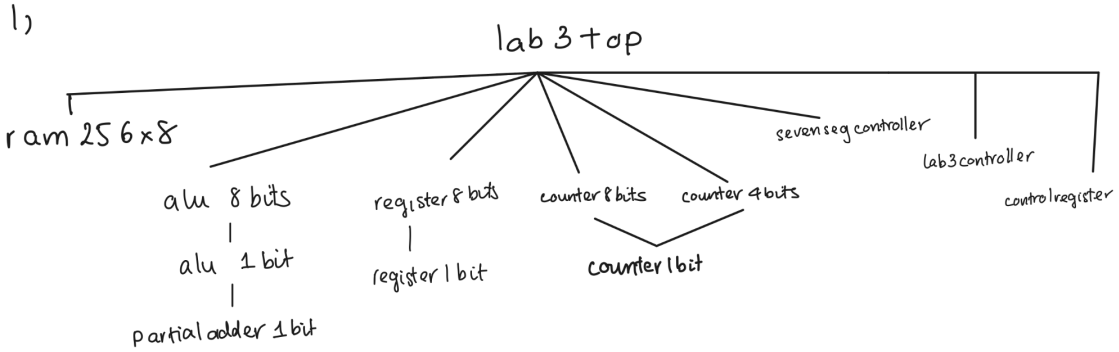
Discussion and Conclusions

Overall, we only faced a minor issue with the demonstration of the hardware part. Ultimately, the simulation results we obtained were correct, and the Altera board demonstrated exactly what we expected. This lab was beneficial in familiarizing us with using op-codes to write simple programs in machine code as well as designing our own machine code to satisfy a specified requirement. We were able to have a chance to apply the concepts that were taught in class, regarding the workings of a computer and its components (control unit) and the process on how to design a working computer program in accordance to specifications, which helped us gain a deeper understanding of those concepts.

Pre-lab/Appendix

Hardware

5.1



2. The bus is an 8-to-1 multiplexer, therefore, only 1 register can be selected at a time.
3. Synchronous since they are connected to a universal clock.
4. The register will reset. The D flip-flops are in a reset state when the signal of the reset is low logic level. The reset is high since load and reset are simultaneously sent to a register. Therefore, we have a low logic level from the AND gate which causes the register to reset.
5. Because the memory needs an address to write the data.
6. Because these registers do the load, increment, and decrement since they store the data. As the counter can do all of these operations, the program counter, data register, and accumulator are implemented as counters.
7. Increment has the highest priority, while reset has the lowest priority. Analyzing the tables in the lab manual, we see that increment is used the most while reset is used the least.
8. It is not possible as they aren't directly connected

9)

S_2	S_1	S_0	X	Y	Operation
0	0	0	ACi	DRi	AC + DR
0	0	1	ACi	DRi	AC + DR' + 1
0	1	0	ACi+1	xx	AC*2 (ashl AC
0	1	1	ACi-1	xx	AC/2 (ashr AC)
1	0	0	ACi	DRi	AC^DR
1	0	1	ACi	DRi	AC v DR
1	1	0	xx	DRi	DR
1	1	1	ACi	xx	AC'

The shift operations are arithmetic

5.2

1. $memWrite = T_9 Y_4 + T_{10} Y_6$
2. $AR\_Load = T_0 + T_2 + IR_6' (T_5 + T_6) + T_7 x_2$
3. $PC\_Load = T_8 Y_5$
4. $PC\_Inc = (T_2 + T_5 IR_6' + Y_6 (T_{11} + T_{12}) DR'_{(7...0)}) Stop$
5. $DR\_Load = T_8 (Y_0 + Y_1 + Y_2 + Y_3 + Y_6)$
6. $DR\_Inc = T_9 Y_6$
7. $IR\_Load = T_3$
8. $AC\_Clear = T_5 x_1 IR_0$
9. $AC\_Load = T_5 x_1 (IR_1 + IR_2 + IR_3) + T_9 (Y_0 + Y_1 + Y_2 + Y_3)$
10. $AC\_Inc = T_5 x_1 IR_4$
11. $OUTP\_Load = T_1$
12. $ALU\_Sel2 = T_9 (Y_0 + Y_3) + T_{10} Y_6 + T_5 x_1 IR_1$
13. $ALU\_Sel1 = T_5 x_1 (IR_1 + IR_2 + IR_3) + T_9 Y_3 + T_{10} T_6$
14. $ALU\_Sel0 = T_5 x_1 (IR_1 + IR_3) + T_9 Y_2$
15. $BusSel2 = T_0 + T_9 Y_4$
16. $BusSel1 = T_{10} Y_6 + T_0 + T_2 + T_5$
17. $BusSel0 = T_{10} Y_6 + T_9 Y_4 + T_8 Y_5$
18. $SC\_clear = T_5 x_1 + T_{12} Y_6 + T_8 Y_5 + T_9 (Y_0 + Y_1 + Y_2 + Y_3 + Y_4)$
19. $Halt = T_5 x_1 IR_5$

Software

7.1

1.

00: Load the content of memory location a0 into the accumulator (AC).
01: Operand for the previous instruction. It specifies that the value to be loaded into the accumulator comes from the memory address stored in a0.
02: Complement the AC.
03: Store the complemented value back into the memory location specified by the next instruction.
04: Load the content of memory location a0 into the AC.
05: Increment the value stored in the memory at the address specified by the next instruction (a0). If the result is zero, skip the next instruction.
06: Operand for the previous instruction. It specifies the memory address where the increment operation is applied (‘a0’)
07: Branch unconditionally to the memory location specified by the next instruction

08: Operand for the previous instruction. It specifies the target address for an unconditional branch (‘20’)
09: Halt the program
20: Load the content of the memory indirectly pointed to by the memory location specified by the next instruction into the accumulator.
21: Operand for the previous instruction. It specifies the memory address containing a pointer to the first number to be added (‘a1’)
22: Add the content of the memory indirectly pointed to by the memory location specified by the next instruction to the accumulator.
23: Operand for the previous instruction. It specifies the memory address containing a pointer to the second number to be added (‘a2’)
24: Store the result of the addition into the memory location indirectly pointed to by the memory location specified by the next instruction
25: Operand for the previous instruction. It specifies the memory address containing a pointer to where the result will be stored (‘a3’)
26: Load the content of the memory at the address specified by the next instruction into the accumulator.
27: Operand for the previous instruction. It specifies the memory address containing the first number to be added (‘a1’)
28: increments the value in the accumulator
29: stores incremented value from the accumulator back into the memory location a1.
2a: loads the content of the memory at address a1 into the accumulator
2b: increments the value in the AC
2c: stores the incremented value from the accumulator back into the memory location a1
2d: loads the content of the memory at address a2 into the AC
2e: increments the value in the AC
2f: stores the incremented value from the accumulator back into the memory location ‘a2’
30: loads the content of the memory at address a3 into the AC
31: unconditionally jumps to the address specified by the next instruction
32: Operand for the previous instruction. It specifies the address (‘05’)

2. Pseudocode

```
AC <- Counter
AC <- AC’
Counter <- AC
Counter <- Counter+1
If Counter = 0, then PC <- PC + 1 and HLT else
        PC <- 20
        AC <- M[X]
        AC <- AC+M[Y]
        M[Z] <- AC
        AC <- X
        AC <- AC+1
        X <- AC
        AC <- AC+
        Y <- AC
        AC <- AC+1
        Z <- AC
        Loop back to line 4
```

3.
The program sums the operands pointed to by X and Y and stores the sum in the address pointed to by Z. Then it increments the pointers X, Y and Z by 1.
It calculates a Fibonacci sequence of size 12, where every number starting from position 3 is a sum of 2 previous numbers

4.
Cause it needs to access and modify data stored at different memory locations based on pointers. Indirect addressing allows it to retrieve values from memory addresses pointed to by other memory locations (X, Y, Z) and perform computation accordingly

7.2

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	84	A1	08	9F	42	08	A0	42	....B..B
008	20	A0	10	20	60	00	00	00	.. `...
010	00	00	00	00	00	00	00	00	....
018	00	00	00	00	00	00	00	00	....
020	04	A1	50	08	A1	84	A1	08	..P.....
028	A2	02	9F	10	02	00	00	00	....
030	00	00	00	00	00	00	00	00	....
038	00	00	00	00	00	00	00	00	....
040	00	00	00	00	00	00	00	00	....
048	00	00	00	00	00	00	00	00	....
050	00	00	00	00	00	00	00	00	....
058	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	....
068	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	....
078	00	00	00	00	00	00	00	00	....
080	21	B5	37	08	5C	84	A1	1D	!.7.\...
088	72	FF	F6	43	03	A9	D4	19	r..C....
090	31	D9	47	82	14	52	07	CA	1.G..R..
098	04	00	00	00	00	00	00	00	....