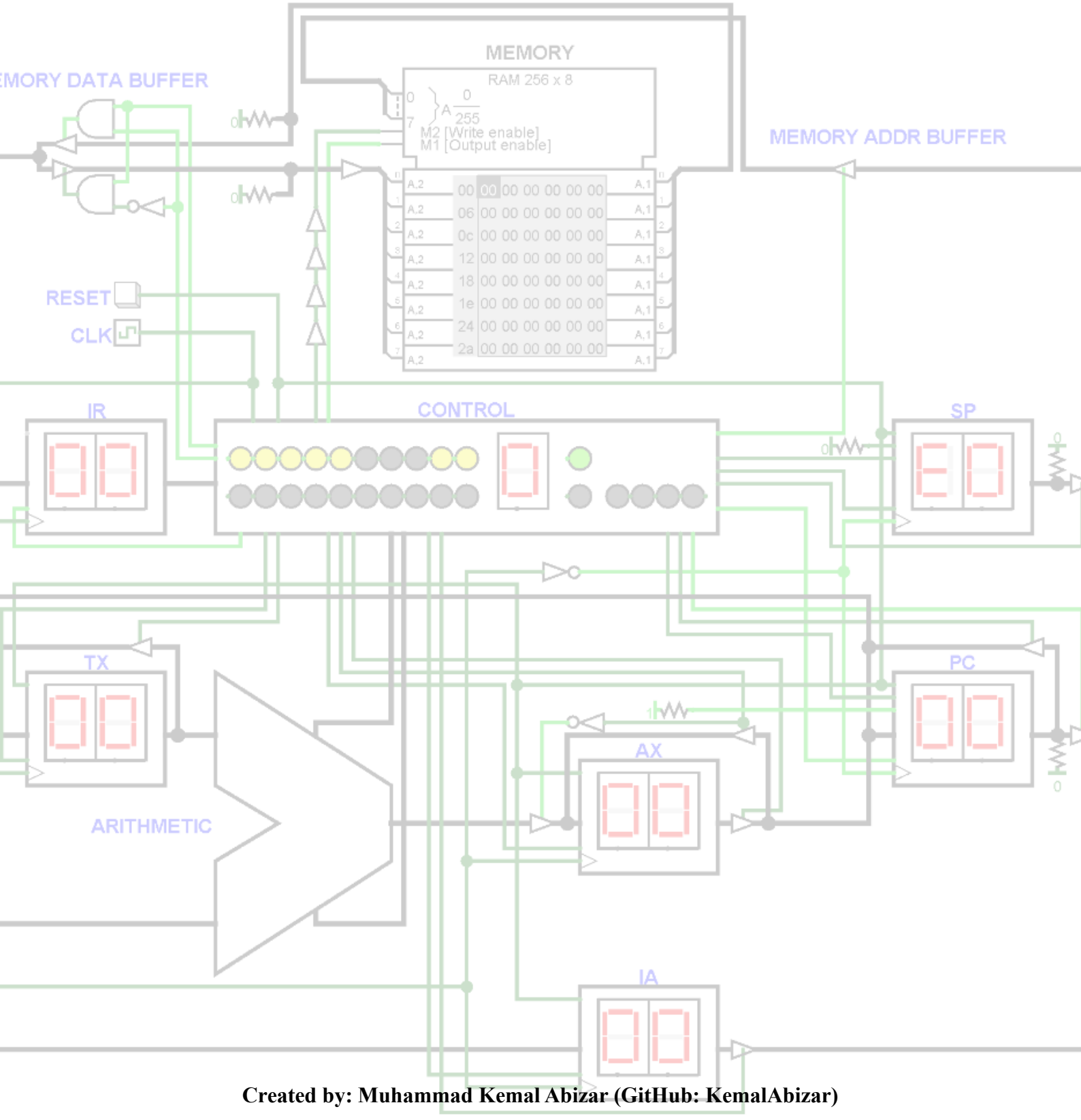
Abacus-8 Project Documentation



Created by: Muhammad Kemal Abizar (GitHub: KemalAbizar)

MIT License

Table of Contents

Instruction Set Architecture

1. µ-Architecture

The Abacus-8's micro-architecture (µ-arch) is designed with simplicity as its main goal, in regards of the datapath and user interfacing. Hence, it presents minimum register count, by only having Accumulator (AX) and Program Counter (PC) registers accessible for the programmer to manipulate; said design decision also meant that instructions would heavily involve reading from, or writing to Memory (RAM) chip. An additional feature that was later added in its design, was stack-addressing capability, enabling stack memory upto 32 locations; on the Arithmetic and Logic Unit (ALU), a multiplication module (default in Logisim-Evo) is added, which enables the Abacus-8 computer to multiply two integers at once.

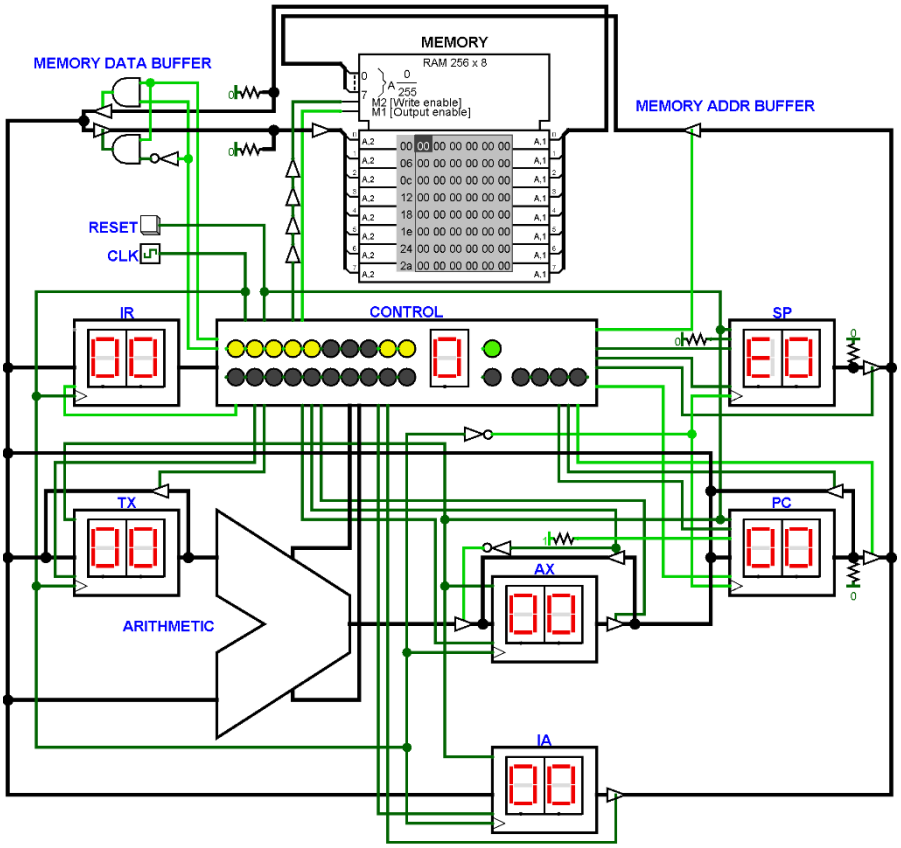


Figure 1.1. Abacus-8 Complete Circuitry.

Aside from minimal register count, the Abacus-8' architecture is designed to resemble that of CPUs commonly found in early ages of personal/home computing devices (e.g. MOS 6502, Intel 8008), along with other inspirations from educational or hobby computer kits (e.g. Ben Eater's 8-bit TTL computer, Gigatron TTL). This drives the decision to make both data and address bus being 8-bit wide. Being a proof-of-concept, Abacus-8's interface method is rather a minimum; there was no input device beyond manually uploading hexfiles into the RAM, and the register's outputs were displayed by 7-segment LEDs. Aside from registers, the Control Unit' (CU) signals, flags, clock cycle, and its run/halt status are displayed by simple LED bulbs and 7-segment LEDs.

2. Arithmetic and Logic Unit (ALU)

Responsible for computing arithmetic, bitwise comparison and logic operations are the Arithmetic and Logic Unit (ALU), having the input of two 8-bit integers A and B, along with the operation code (opcode, labeled Op). All operations are accomplished simultaneously, as shown in the circuit diagram below where both A and B integer inputs are fed into each arithmetic/logic modules; the array of 3-AND decoders are wired to Opcode input, that controls which modules' computation result, or the desired operation, can be outputted from the ALU. For the full adder-subtractor unit, shown here as two topmost modules,

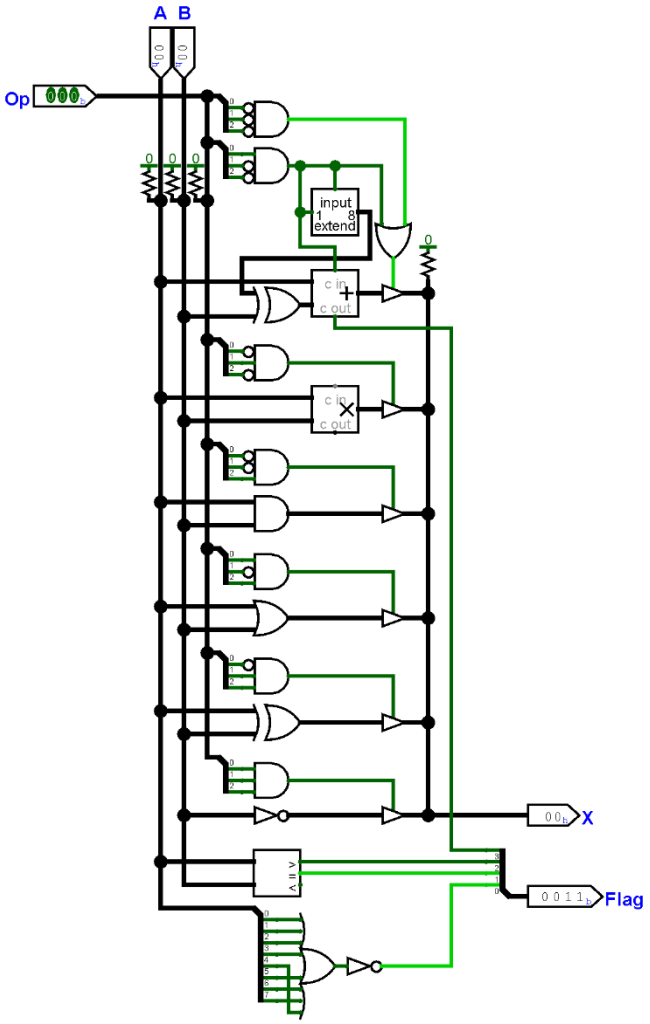


Figure 1.2. Abacus-8 Arithmetic and Logic Unit' Internal Circuitry.

Aside from arithmetic (full adder-subtractor, integer multiplier) and logic modules, the ALU is also equipped with a comparator module (shown at the bottom), operating in unsigned mode by default. This comparator module outputs three flags, or status bits, regarding whether A is larger than B (flag A, pinout ">"), A equal to B (flag E, pinout "="), or that A is zero (flag Z, output from NOR gate). Combined with the carry (flag C) output from full adder-subtractor module, which signifies whether there's an overflow in addition or subtraction, the ALU outputs 4-bit flags.

3. Control Unit (CU)

Datapath control are accomplished by the Control Unit (CU), based on the inputs of instruction opcode (INST), clock signal (CLK), and 4-bit flags from the ALU. Its design are based on that of hardwired systems, using combinational logic gates (NOT-AND-OR) to drive control signals of all registers/memory/counters; such design rule follows that of older CPUs aforementioned in the μ-Architecture section. In future iterations, PLA (programmable logic array) may replace these hardwired combinational logic, to ease in implementing new or improving existing instructions.

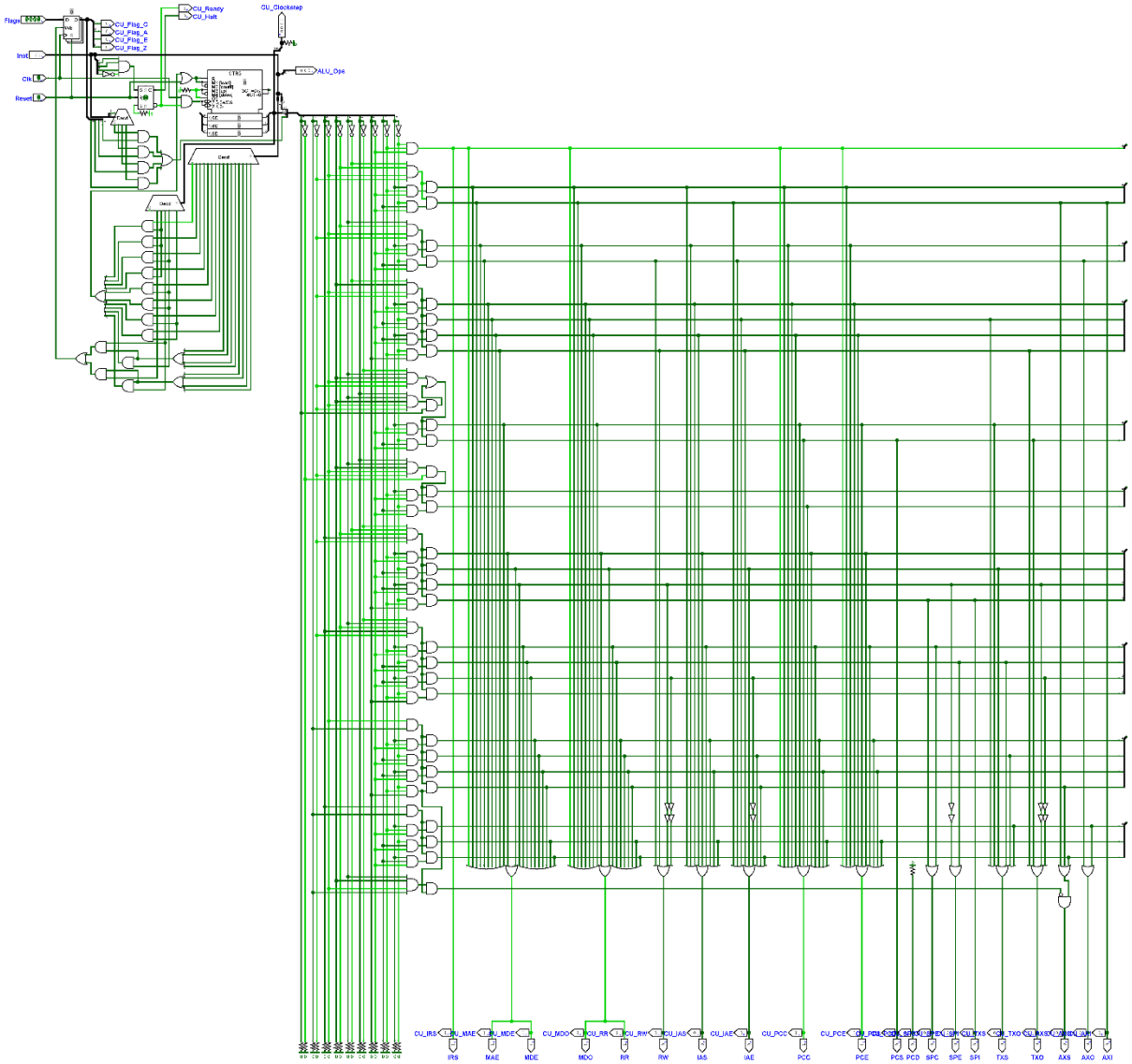


Figure 1.3. Abacus-8 Control Unit Internal Circuitry.

The clock input drives a counter circuit, which divides instruction execution upto 8 clock cycles/sequence. Instructions differ in their time, or clock cycle counts, to be executed; hence a reset circuit is implemented, comprised of AND gates that outputs logic '1' for certain instruction at certain counter cycle, thus resetting the cycle counter after each instruction is completely executed. This reset circuit are indicated by the 10-OR gate, its inputs fed by 2-AND gates that are triggered by instruction type and clock cycle count.

Flag inputs are stored in a small 4-bit register, shown below on top left corner, that is updated every positive-edge clock signal after any ALU operations. Stored flag bits are then compared against flag indications found in the opcode byte (see Instruction Encoding for further details) with the use of matching circuit, similar in mechanism to the counter reset circuit. A 2-bit decoder is connected to instruction bit 2 and bit 1, its outputs fed into four 2-AND gates below. If either of these AND gates are triggered, it signifies that the flags requested by current instruction are valid, or otherwise; this information is useful to determine whether or not to jump into another instructio loop, known as conditional jumps (see Instruction Set section).

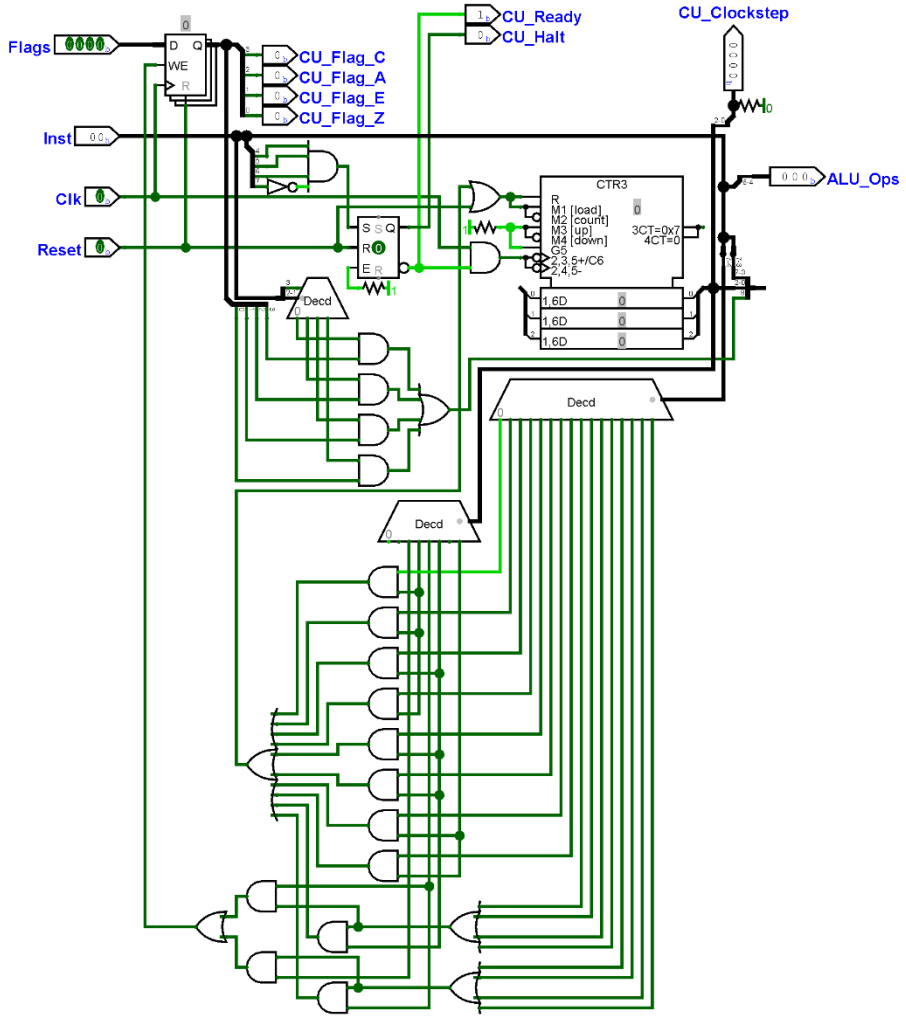


Figure 1.4. Abacus-8 Control Unit's Clock Cycle Counter, Reset Circuitry, Flag Register and Matching Circuit (inset from Fig. 1.3).

4. Memory Mapping

Address Range	Size	Usage
0x00 – 0x7F	128 bytes	Instruction storage
0x80 – 0xDF	96 bytes	Data storage
0xE0 – 0xFF	32 bytes	Stack segment

5. Instruction Encoding

Mnemonic	Opcode Byte	Operand 1	Operand 2	Notes
LDA	0000 ----	AAAA AAAA	-	
STA	0001 ----	AAAA AAAA	-	
MOV	0010 ----	AAAA AAAA	BBBB BBBB	
JMP	0011 0---	QQQQ QQQQ	-	
JCB	0011 100-	QQQQ QQQQ	-	
JAL	0011 101-	QQQQ QQQQ	-	
JEQ	0011 110-	QQQQ QQQQ	-	
JZR	0011 111-	QQQQ QQQQ	-	
PHS	0100 ----	AAAA AAAA	-	
PLS	0101 ----	AAAA AAAA	-	
NOP	0110 ----	-	-	
HLT	0111 ----	-	-	
ADD	1000 ----	AAAA AAAA	BBBB BBBB	
SUB	1001 ----	AAAA AAAA	BBBB BBBB	
MUL	1010 ----	AAAA AAAA	BBBB BBBB	
CMP	1011 ----	AAAA AAAA	BBBB BBBB	
ANA	1100 ----	AAAA AAAA	-	
ORA	1101 ----	AAAA AAAA	-	
XRA	1110 ----	AAAA AAAA	-	
NOT	1111 ----	AAAA AAAA	-	

Assembly Language

1. Instruction Set

Opcode	Operand	Clock Cycles	Description
LDA	$A	2	Loads data from address $A to AX (accumulator).
STA	$A	2	Stores data to address $A from AX.
MOV	$A,$B	4	Moves data at address $A to address $B.
JMP	$Q	2	Jumps to next instruction at address $Q.
JCB	$Q	2	Jumps to $Q if “Carry” flag is high.
JAL	$Q	2	Jumps to $Q if “A Larger” flag is high.
JEQ	$Q	2	Jumps to $Q if “Equal” flag is high.
JZR	$Q	2	Jumps to $Q if “Zero” flag is high.
PHS	$A	4	Push data at address $A to stack.
PLS	$A	4	Pull data to address $A from stack.
NOP		8	No operation (useful for sleep).
HLT		1	Halt until reset.
ADD	$A,$B	4	Add the data contained within addresses $A and $B.
SUB	$A,$B	4	Subtract the data contained within addresses $A and $B.
MUL	$A,$B	4	Multiply the data contained within addresses $A and $B.
CMP	$A,$B	4	Bitwise compare data at addresses $A against $B.
ANA	$A	3	Logical AND data in AX with data at address $A.
ORA	$A	3	Logical OR data in AX with data at address $A.
XRA	$A	3	Logical XOR data in AX with data at address $A.
NOT	$A	3	Logical NOT data at address $A.

2. Sourcefile Structure

```
; ****************** ABACUS-8 ASSEMBLY GUIDE ******************
; Version 0.1 (26/01/2026)
; (c) 2026 KemalAbizar
;
; This is how comments are written. Current version of Abacus-8'
; assembler does not support multi-line comments, such as ones
; in Python utilizing triple quote or double-quote symbols. Hence,
; one must manually write new lines and add semicolon symbol, only
; then can "multi-line" comments be made.
;
; NOTE: be sure to only have maximum of one semicolons per line!


; Program segment. This is where one writes the program he/she
; desires the computer to execute. Any other instruction lines, or
; instruction loops written outside of this scope, will be ignored.
.prog:
count:          ; Loop descriptor / label.
      add x,y
      sta z     ; To call variables, one just needs to write the
      mov y,x   ; variable name / label.
      mov z,y
      cmp x,l
      jal terminate
      jmp count
terminate:      ; Another loop descriptor / label.
      lda mssg
      hlt

; Data segment. All variables necessary are declared here.
; NOTE: variables must always be declared with ':=' symbols.
.data:
x := $00
y := $01
z := $00
l := $5f
mssg := $e2
```

3. How to Write, Assemble and Run the Program

Writing the assembly sourcefile for Abacus-8 won't require anything more than basic text editors (Notepad/++, MS Word etc.) or program editors (MS Visual Studio Code). After one finishes writing the program, it must be saved in the project folder titled "assembly," which also contains a Python program titled "assembler.py," as shown in figures below.

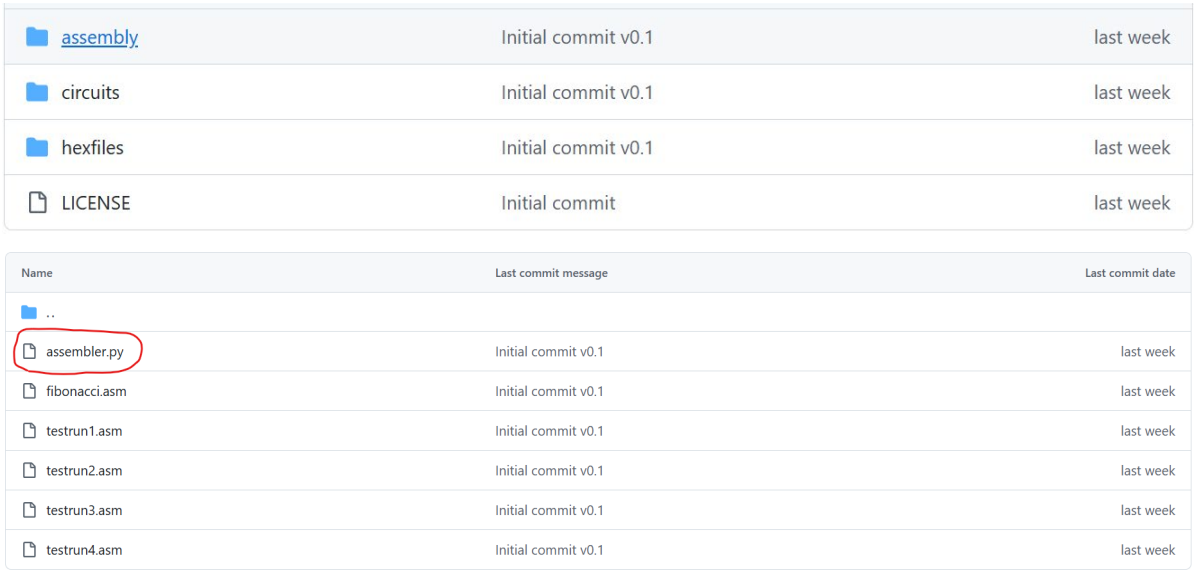


Figure 2.1. Project Files.

Open a Command Prompt (Windows) or Terminal (Linux), then change the directory to access the "assembly" folder. To compile (technically it's called assemble) the sourcefile that one just wrote, type the following command shown in green box, then press Enter. The output file, highlighted by yellow box, contains the generated hexadecimal data and their corresponding memory locations. Open another empty text/program editor window, then copy all generated memory contents into it; save it with identical name as the assembly sourcefile, but this time with .hex extension. Save the .hex file into the "hexfiles" folder.

```
C:\Users\M. Kemal Abizar\Documents\Projek\Computer\assembly> python assembler.py fibonacci.asm
v3.0 hex words addressed
00: 00 80
02: 80 80 81
05: 10 82
07: 20 81 80
0a: 20 82 81
0d: b0 82 83
10: 3a 14
12: 30 00
14: 00 80
16: 70
80: 00
81: 01
82: 00
83: 1f

C:\Users\M. Kemal Abizar\Documents\Projek\Computer\assembly>
```

Figure 2.2. Command Prompt Example.

The final step is to open the circuit file (ones with .circ extension), then right-click the RAM or “Memory” module, which would show a dropdown menu named “Load Image.” Click this, then double-click on your recently compiled hexfile; this way, one have successfully loaded your program into the Abacus-8 computer. All one needs to do, is to enable auto-clock by pressing Ctrl+K, and observe the computer coming to life.

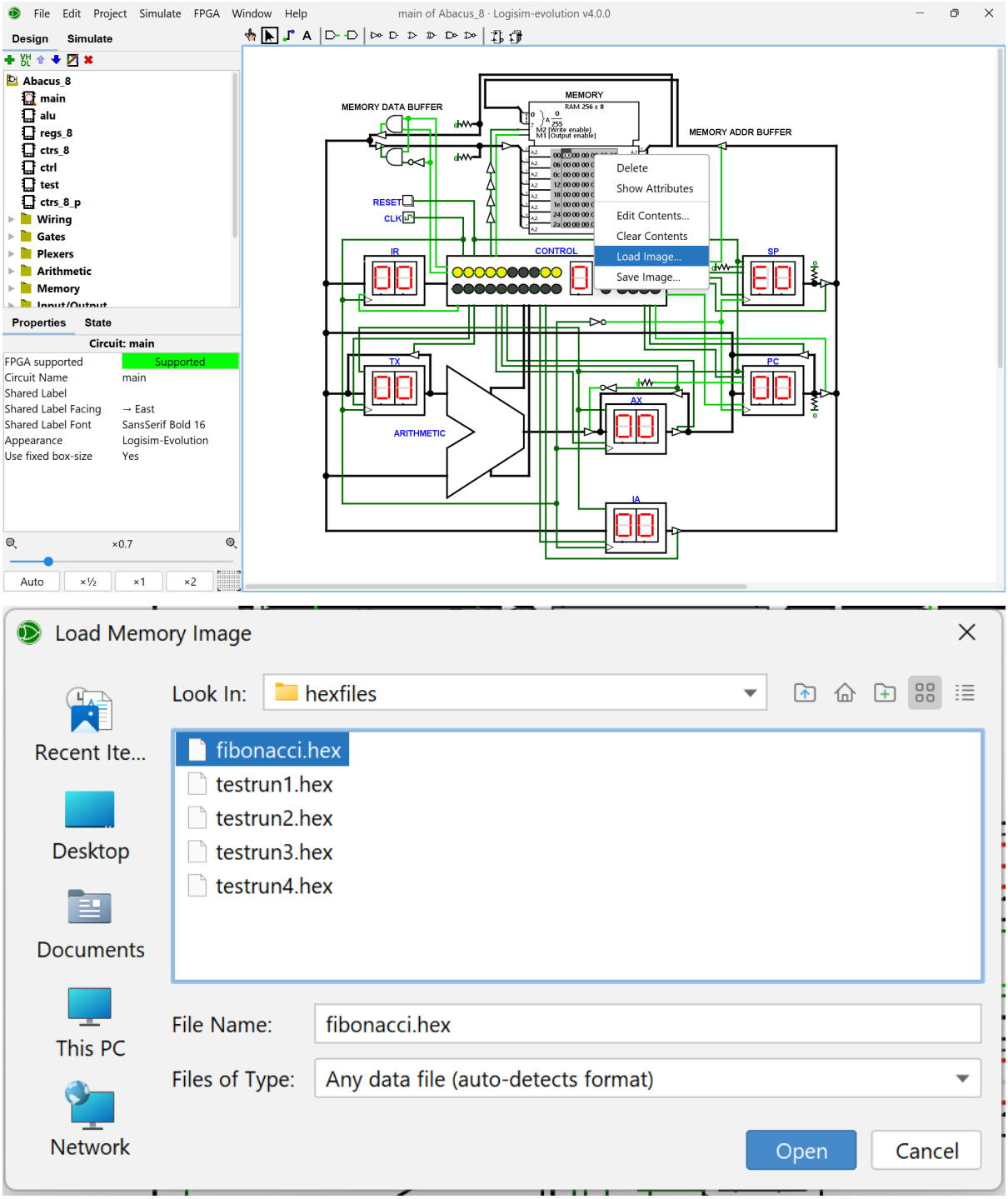


Figure 2.3. Loading the Hexfiles into RAM.