VM252* Logisim computer

Supreme Paudel

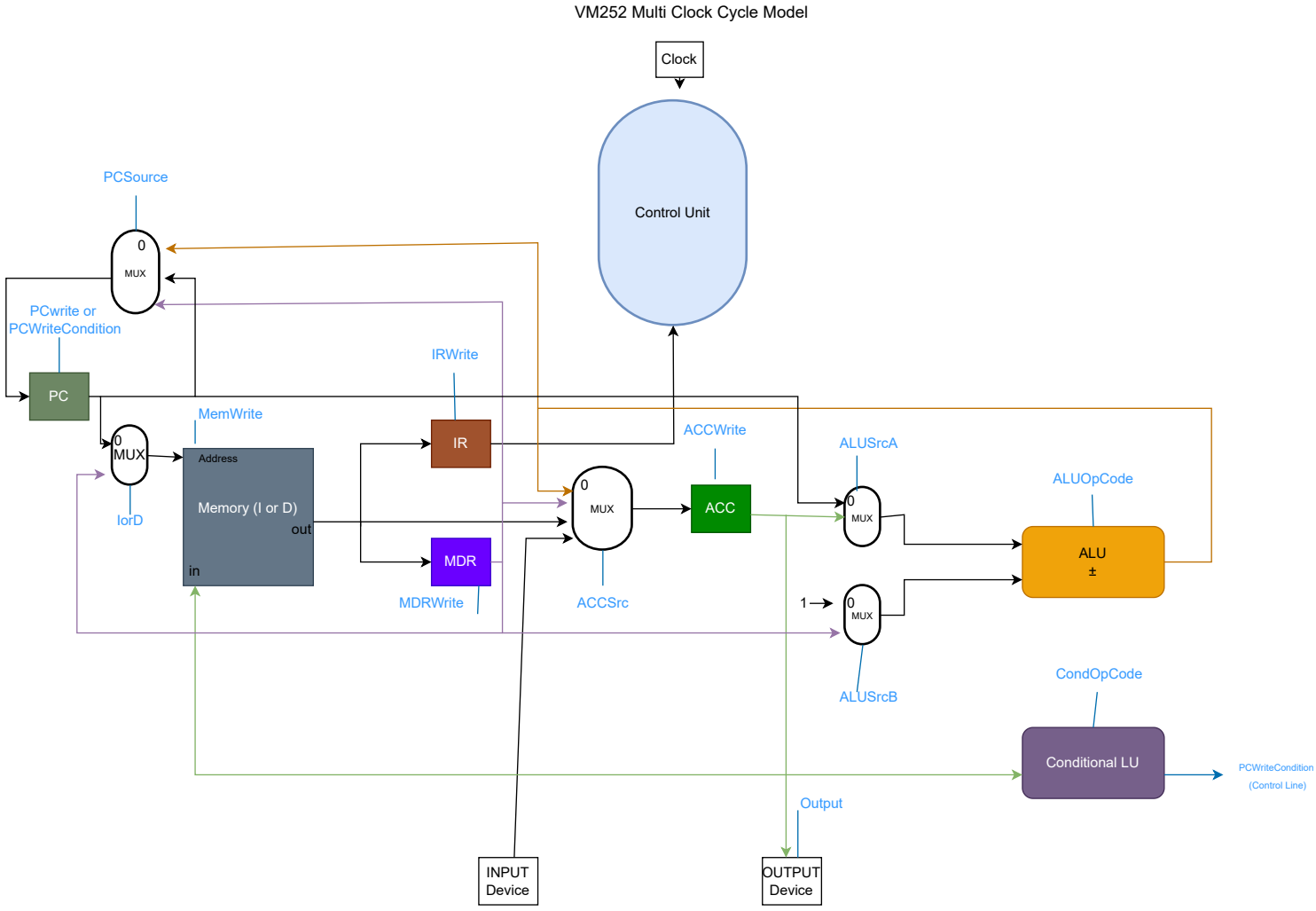
CS375

Contents

1 Introduction

VM252* Logisim circuit is a multi clock cycle CPU design that simulates the VM252 assembly language instructions.
The multi-clock cycle VM252* model can be seen below:

Look at *VM252ArchitectureSoftwareAndProgrammingInformation.pdf* file inside the *resources* directory in the github repository to learn about the VM252 Assembly language in detail.


VM252 Multi Clock Cycle Model
Clock
Control Unit
PCSource
0
MUX
PCwrite or
PCWriteCondition
PC
IRWrite
MemWrite
0
MUX
IorD
Address
Memory (I or D)
out
in
IR
MDR
MDRWrite
ACCWrite
0
MUX
ACC
ACCSrc
ALUSrcA
0
MUX
ALUOpCode
ALU
±
1
0
MUX
ALUSrcB
CondOpCode
Conditional LU
PCWriteCondition
(Control Line)
Output
INPUT
Device
OUTPUT
Device

The blue lines are the control lines coming from the control unit and have been omitted in this diagram. Moreover, some bit extenders and setting certain bits to 0 for memory addresses have not be shown in the diagram either.

2 Why VM252* and not VM252?

1. The VM252 memory is collection of 8192 eight-bit bytes whereas the VM252* memory is 8192 cells of size 16 bits each.
2. All instructions in the VM252* are 16 bits long unlike the VM252 language where certain instructions are only 8 bits long.
3. The Program Counter is always incremented by 1 to execute the next instruction unlike the VM252 because of (2).

3 Instructions encoding

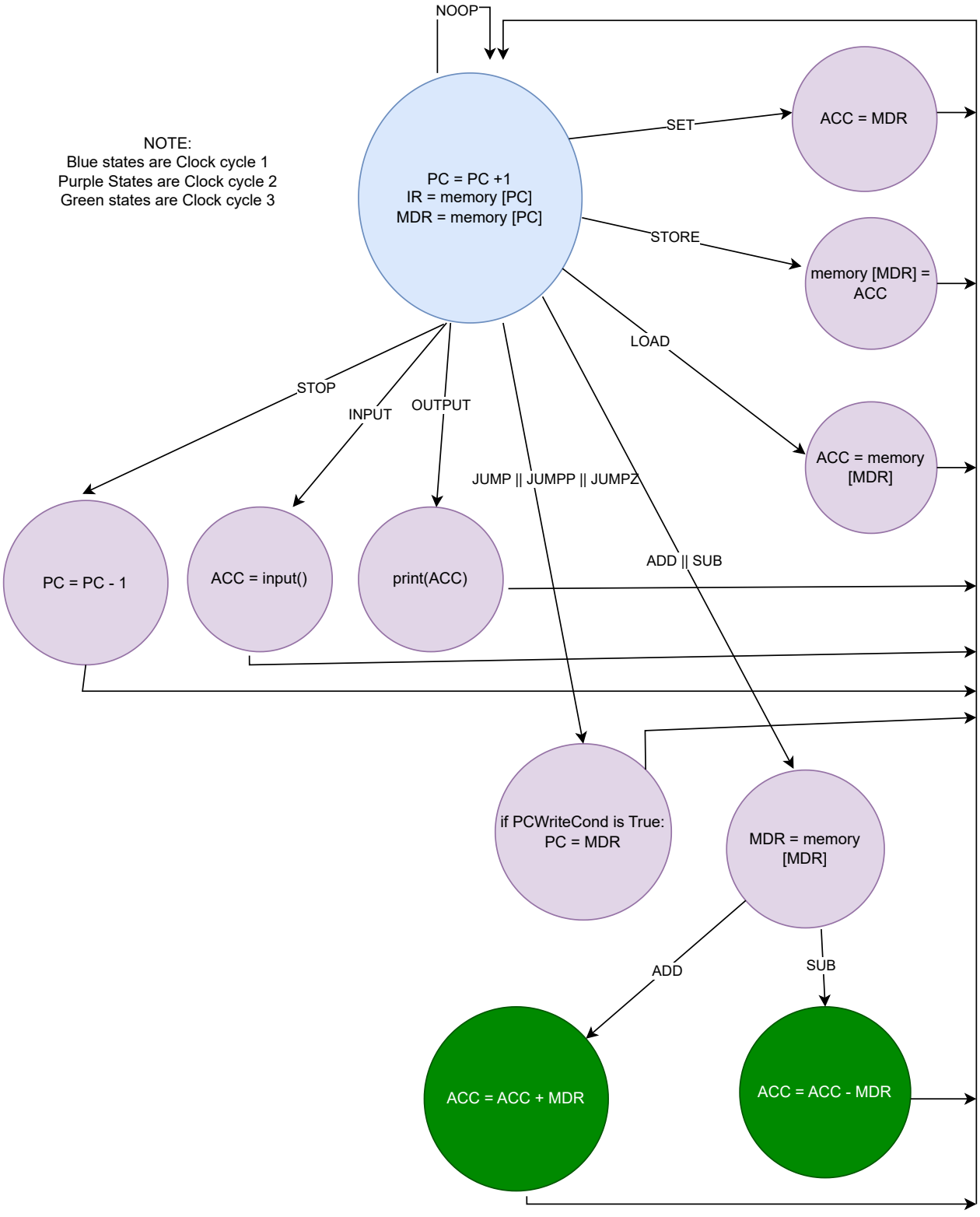
- Instructions are encoded in the same way as it is in the VM252 language except for the one big difference that $INPUT$, $OUTPUT$, $NOOP$ and $STOP$ instructions are now extended to be 16 bits instructions.
- The table below shows how the instructions are encoded and their respective hexadecimal value with the unknown bits set to 0:

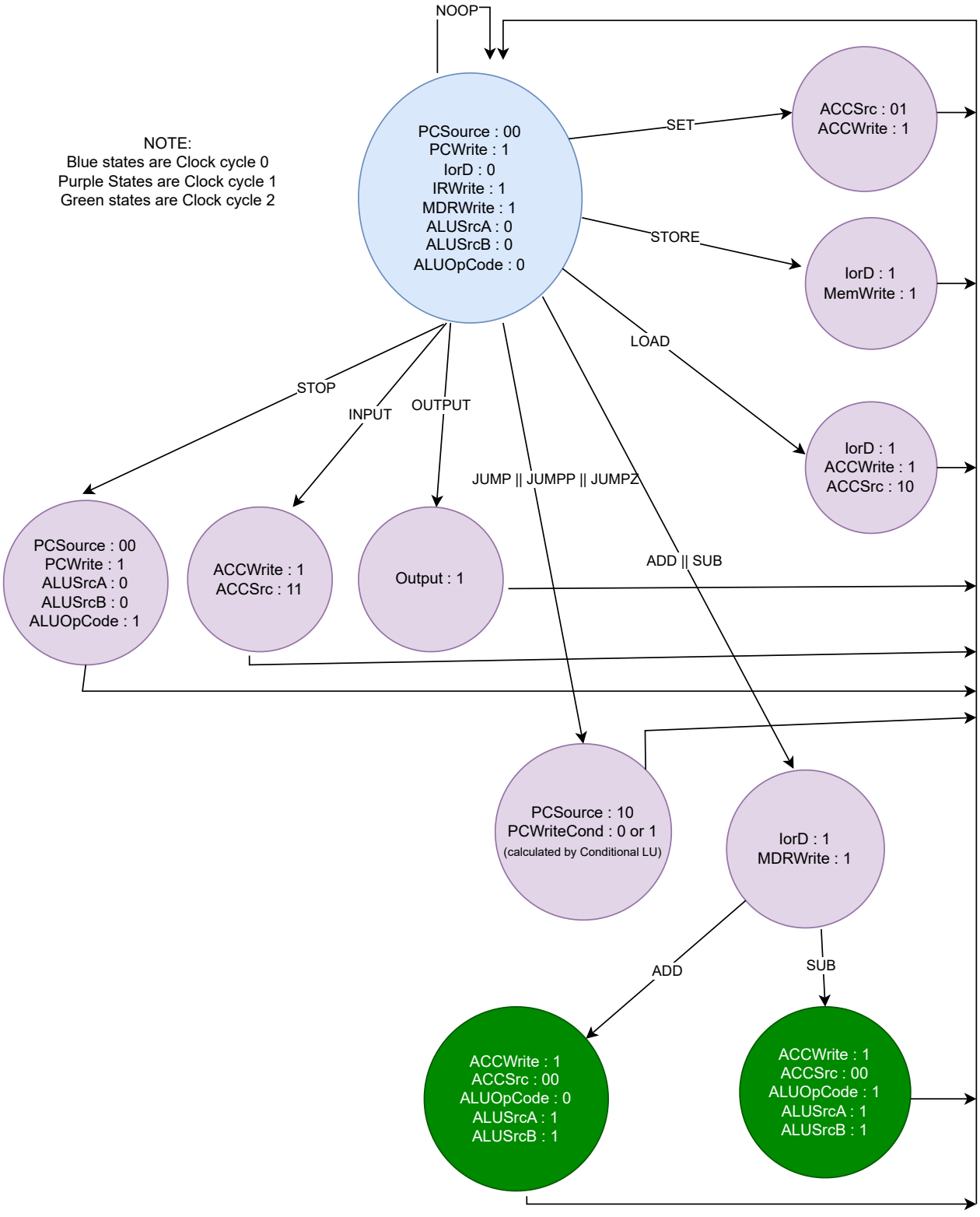
Instruction	Instruction encoded in 16 bits	Hex value with unknown bits set to 0
INPUT	111100xxxxxxxxxx	F000
OUTPUT	111101xxxxxxxxxx	F400
NOOP	111110xxxxxxxxxx	F800
STOP	111111xxxxxxxxxx	FC00
LOAD a	$000a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	0000
STORE a	$001a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	2000
ADD a	$010a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	4000
SUB a	$011a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	6000
JUMP a	$100a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	8000
JUMPZ a	$101a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	A000
JUMPP a	$110a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	C000
SET c	$1110c_{11}c_{10}c_9c_8c_7c_6c_5c_4c_3c_2c_1c_0$	E000

- For example, a STORE instruction in address 9_{hex} would have hex value 2009.

4 Finite State machine for the VM252*

For our multi-clock cycle design, we will need to implement a FSM to specify control. The FSM for our VM252* is shown below :


NOOP
NOTE:
Blue states are Clock cycle 1
Purple States are Clock cycle 2
Green states are Clock cycle 3
PC = PC +1
IR = memory [PC]
MDR = memory [PC]
SET
ACC = MDR
STORE
memory [MDR] = ACC
LOAD
ACC = memory [MDR]
STOP
INPUT
OUTPUT
JUMP || JUMPP || JUMPZ
ADD || SUB
PC = PC - 1
ACC = input()
print(ACC)
if PCWriteCond is True:
PC = MDR
MDR = memory [MDR]
ADD
SUB
ACC = ACC + MDR
ACC = ACC - MDR


NOOP
NOTE:
Blue states are Clock cycle 0
Purple States are Clock cycle 1
Green states are Clock cycle 2
PCSource : 00
PCWrite : 1
IorD : 0
IRWrite : 1
MDRWrite : 1
ALUSrcA : 0
ALUSrcB : 0
ALUOpCode : 0
SET
ACCSrc : 01
ACCWrite : 1
STORE
IorD : 1
MemWrite : 1
LOAD
IorD : 1
ACCWrite : 1
ACCSrc : 10
STOP
INPUT
OUTPUT
JUMP || JUMPP || JUMPZ
PCSource : 00
PCWrite : 1
ALUSrcA : 0
ALUSrcB : 0
ALUOpCode : 1
ACCWrite : 1
ACCSrc : 11
Output : 1
ADD || SUB
PCSource : 10
PCWriteCond : 0 or 1
(calculated by Conditional LU)
IorD : 1
MDRWrite : 1
ADD
SUB
ACCWrite : 1
ACCSrc : 00
ALUOpCode : 0
ALUSrcA : 1
ALUSrcB : 1
ACCWrite : 1
ACCSrc : 00
ALUOpCode : 1
ALUSrcA : 1
ALUSrcB : 1

5 Implementing the FSM

FSM is implemented in the VM252* with the help of a ROM. The thing to realise is that if we know our current state with

- which instruction we are executing
- and the clock cycle we are in

we can then specify control for that state and also know what the next state should be.

1. Our ROM has addresses which are 14-bits long each of which holds value 15-bits long.
2. The address is made up based on the instruction and the current clock cycle we are in.
3. A 14-bits address is of the form $a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ where
   - a_{13} is 1 if OPCODE = INPUT, 0 otherwise
   - a_{12} is 1 if OPCODE = OUTPUT, 0 otherwise
   - a_{11} is 1 if OPCODE = NOOP, 0 otherwise
   - a_{10} is 1 if OPCODE = STOP, 0 otherwise
   - a_9 is 1 if OPCODE = LOAD, 0 otherwise
   - a_8 is 1 if OPCODE = STORE, 0 otherwise
   - a_7 is 1 if OPCODE = ADD, 0 otherwise
   - a_6 is 1 if OPCODE = SUB, 0 otherwise
   - a_5 is 1 if OPCODE = JUMP, 0 otherwise
   - a_4 is 1 if OPCODE = JUMPZ, 0 otherwise
   - a_3 is 1 if OPCODE = JUMPP, 0 otherwise
   - a_2 is 1 if OPCODE = SET, 0 otherwise
   - a_1 and a_0 form the current clock cycle number for the instruction where a_1 is the significant bit.
4. The value of each cell specifies the control lines for the current state and also provides the next clock cycle which is stored in a register so that it can then be used to form the address in the next clock cycle.
5. A 15-bit value is of the form $v_{14}v_{13}v_{12}v_{11}v_{10}v_9v_8v_7v_6v_5v_4v_3v_2v_1v_0$ such that
   - PCSource : $v_{14}v_{13}$
   - PCWrite : v_{12}
   - IorD : v_{11}

- MemWrite : v_{10}
- IRWrite : v_9
- MDRWrite : v_8
- ACCSrc : $v_7 v_6$
- ACCWrite : v_5
- ALUSrcA : v_4
- ALUSrcB : v_3
- Output : v_2
- v_1 and v_0 form the next clock cycle number for the instruction where v_1 is the significant bit.

6. This helps us implement the FSM.

All the used addresses and their values along with their respective instructions are shown below:

Instruction	14 bits Address	Address in hex	15 bits Value	Value in hex
Beg. of Program	00001000000000	0200	001001100000001	1301
LOAD	00001000000001	0201	000100010100010	08A2
LOAD	00001000000010	0202	001001100000001	1301
INPUT	10000000000001	2001	000000011100010	00E2
INPUT	10000000000010	2002	001001100000001	1301
NOOP	00100000000001	0801	001001100000001	1301
OUTPUT	01000000000001	1001	000000000000110	0006
OUTPUT	01000000000010	1002	001001100000001	1301
STOP	00010000000001	0401	001000000000010	1002
STOP	00010000000010	0402	001001100000001	1301
SET	00000000000101	0005	000000001100010	0062
SET	00000000000110	0006	001001100000001	1301
STORE	00000100000001	0101	000110000000010	0C02
STORE	00000100000010	0102	001001100000001	1301
JUMP	00000000100001	0021	100000000000010	4002
JUMP	00000000100010	0022	001001100000001	1301
JUMPZ	00000000010001	0011	100000000000010	4002
JUMPZ	00000000010010	0012	001001100000001	1301
JUMPP	00000000001001	0009	100000000000010	4002
JUMPP	00000000001010	000A	001001100000001	1301
ADD	00000010000001	0081	000100100000010	0902
ADD	00000010000010	0082	000000000111011	003B
ADD	00000010000011	0083	001001100000001	1301
SUB	00000001000001	0041	000100100000010	0902
SUB	00000001000010	0042	000000000111011	003B
SUB	00000001000011	0043	001001100000001	1301

6 How to run programs?

- Download the VM252.circ, VM252ControlUnit if you haven't already.
- Open VM252.circ with logisim and navigate to the *MultiClockCycleVM252* circuit which is the main circuit.
- Initialise the Control Unit ROM if it hasn't been initialized already. To do so, right-click on the ROM → Select "Load image" → Select VM252ControlUnit
- Select a program to run as follows:
  - You can either write your own program by modifying the VM252 memory. Make sure to properly encode instructions.
  - Or, you can load up some programs from the *programs* directory. To do so, right-click on the memory → Select "Load image" → Select a valid program from the *programs* folder.