Table of Contents

Common Bus System

The basic computer has **eight registers**, a memory unit, and a control unit, Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between each register's outputs and the other registers' inputs. A more efficient scheme for transferring information in a system with many registers is to use a common bus.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables $S_2S_1S_0$ The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of **DR** is **3**. The **16-bit** outputs of **DR** are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal **3**. The lines from the common bus are connected to each register's inputs and the memory's data inputs. The register whose **LD (load)** input is enabled receives the data from the bus during the next clock pulse transition. The memory gets the contents of the bus when its write input is activated. The memory places its **16-bit** output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

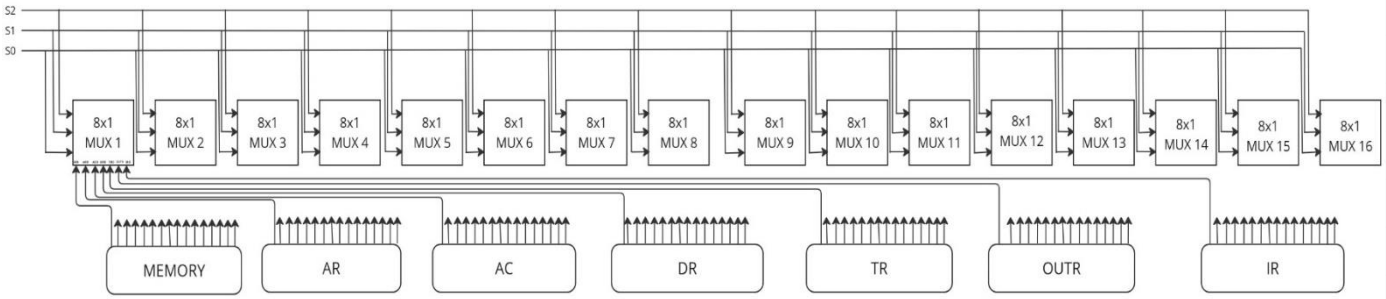
The input data and output data of the memory are connected to the common bus, but the memory address is connected to **AR**. Therefore, **AR** must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except **AC**.

The 16 inputs of **AC** come from an adder and logic circuit. This circuit has three sets of inputs. One set of **16-bit inputs** comes from the outputs of **AC**. They are used to implement register micro-operations such as complement **AC** and shift **AC**. Another set of 16-bit inputs come from the data register **DR.** The inputs from **DR and AC** are used for arithmetic and logic micro-operations, such as **adding DR to AC** or **AND DR to AC**. The result of an addition is transferred to **AC** and

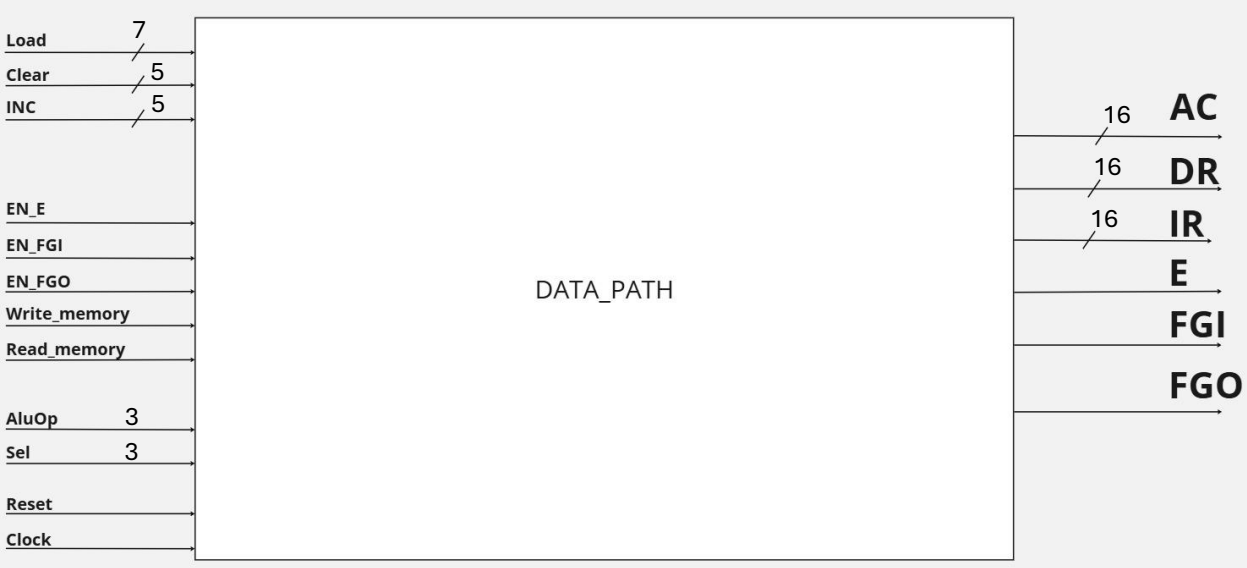
the end carry-out of the addition is transferred to **flip-flop E** (extended AC bit). A third set of **8-bit inputs** comes from the input register **INP**

The **common bus system** can be constructed with **sixteen 8 x 1 multiplexers** in a configuration which shown in Figure below

(Made by Amad Alhaj and Nermeen Nedal)



Datapath



Modules

(Made by Amad Alhaj and Omar khateeb)

1) Modification And Improvement:

In this section we show the modification and improvement to previous design:

1) Memory:
   In the new design, we moved the instruction area starting address from location 2048 to location 0. This adjustment was made to streamline memory implementation. You can refer to the phase one report for better insight into how memory operates in the design.
2) ALU:

   In the new ALU design, we modified its functionality by removing comparison operations (such as SPA, SNA... etc.).

   Instead, ALU now performs additional tasks (CLE and CME) operations. These new operations enable replacing the E flip-flop with a register controlled by an enable signal from the control unit.

   This table include the operation names and AluOp in ALU:

AluOp [3]	AluOp [2]	AluOp[1]	AluOp[0]	operation
0	0	0	0	AND
0	0	0	1	ADD
0	0	1	0	LOAD DR Value to AC
0	0	1	1	Complement AC = (~AC)
0	1	0	0	Circulate ac and e to the right
0	1	0	1	Circulate ac and e to the left
0	1	1	0	Load INPR value to AC
0	1	1	1	complement E
1	0	0	0	Clear E

3) Registers:
   Register don't have big changed, we modified the read and write operations to be active on the negative clock edge.
   This improvement allows the system to achieve reading data immediately after one cycle of a write operation, optimizing the timing for sequential operations. If you want to understand the implementation, you can refer to the phase one report, which provides better insight into how memory operates in the design.

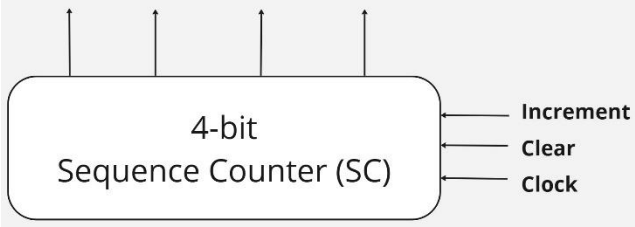
2) New module:

In this section , we show and explain new design in data path:

1) BUS:
   BUS architecture implementation used 16 bits 8×1 mux , have 3 selector lines to select data transfer between register, A bus selector helps manage these connections by choosing the correct source for each bus line based on control signals. a bus is a crucial part of the bus system that ensures the correct routing of data between registers, memory, and other components, providing a controlled mechanism for data transfer within a digital system.
2) E flip flop:
   The E flip-flop in this bus architecture serves as an extension to the accumulator (AC), particularly for handling arithmetic operations involving carry. It works alongside the adder and logic circuitry to store the carry bit resulting from an addition or subtraction operation performed by the ALU. The E flip-flop allows the system to handle larger computations that extend beyond the 16-bit width of the AC register. Its state is critical for determining overflow or underflow conditions during arithmetic operations, ensuring accurate processing of multi-word arithmetic tasks.
3) FGI flip flop:
   the FGI (Final Gate Input) register is involved in handling the input of data from external devices, such as a keyboard in this setup. It works in tandem with the INPR register, which stores the input data.
   The FGI register signals when the data from the external input (keyboard) is ready to be processed by the system. It operates as a control flag to coordinate the timing of the data transfer from the input device into the computer's register, ensuring smooth communication through the serial communication interface.
4) FGO flip flop:
   The FGO (Final Gate Output) register is an essential component in the I/O system that facilitates the communication of data from the Accumulator (AC) to the output terminal, specifically to the printer in this architecture.
   The FGO plays a key role in indicating when data stored in the AC is ready for output through the serial communication interface. It essentially acts as a flag or control signal that helps trigger the transfer of data from the AC to the printer, ensuring the output process is executed correctly and at the right moment.
5) I flip flop:
   We added this module because it is a crucial part of the decode stage is loading the 16th bit of IR into the I flipflop, which helps determining whether the instruction is direct (if the bit is 0) or indirect (if the bit is 1) in case it was a memory-referenced instruction.

6) Sequence Counter:

The Sequence Counter is a crucial component in the control unit of a computer, responsible for tracking and managing the flow of operations within the processor during the execution of an instruction cycle.



the 4-bit Sequence Counter keeps track of the current state in the instruction cycle, with each state corresponding to a specific step of the process. The sequence counter holds a binary value that is incremented (or cleared) at the appropriate times, directing the control unit to execute the next instruction phase. For example, the sequence counter might move from one state to the next to fetch an instruction, decode it, execute it, and finally store the result.

the Sequence Counter ensures that the computer follows the proper sequence of steps, allowing it to execute instructions accurately and efficiently. It serves as the backbone for the timing and synchronization of control signals within the processor, enabling the system to carry out complex tasks in a structured and predictable manner.

SIGNAL

(Made by Amad Alhaj and Nermeen Nedal)

Clock and Reset

- ✓ **Clock** : clock signal used to synchronize operations within the Datapath module.
- ✓ **Reset**: initializes or resets the Datapath's internal states to their default values**.**

Load [6:0]

control signals (**7 bits**) for loading data into specific registers

- ✓ **load [0]:** load accumulator (**AC**)
- ✓ **load [1]:** load output register **(OUTR)**
- ✓ **load [2]:** load temporary register **(TR)**
- ✓ **load [3]:** load instruction registers (**IR**)
- ✓ **load [4]:** load Data register **(DR)**
- ✓ **load [5]:** load program counter register **(PC)**
- ✓ **load [6]:** load address register **(AR)**

Write memory || Read memory

- ✓ **write memory**: triggers writing data from a register to memory**.**
- ✓ **read memory**: triggers fetching data from memory into a register.

Write_E || Enable_I

- ✓ **Write_E:** enables writing to E flip flop when an add operation occurs it gets the carry out value
- ✓ **Enable_I:** activates or enables Indirect addressing flag

---

Enable_FGO || Enable_FGI || IN_FGO || IN_FGI

- ✓ **Enable_FGO:** enables output flag **(FGO)** for external communication.
- ✓ **Enable_FGI:** enables input flag **(FGI)** for external communication**.**
- ✓ **IN_FGI:** input data for the **FGI** signal.
- ✓ **IN_FGO:** input data for the **FGO** signal.

ALU [2:0]

controls the arithmetic logic unit **(ALU)** operation. specific 3-bit combinations determine **ALU** behavior (e.g., addition, subtraction, bitwise operations).

INC [4:0]

control signals to increment specific registers

- ✓ **INC [0]:** increment accumulator **(AC)**
- ✓ **INC [1]:** increment temporary register **(TR)**
- ✓ **INC [2]:** increment data register **(DR)**
- ✓ **INC [3]:** increment program counter **(PC)**
- ✓ **INC [4]:** increment address register. **(AR)**

CLEAR [4:0]

control signals to clear (**reset to 0**) specific registers

- ✓ **CLEAR [0]:** clear accumulator **(AC).**
- ✓ **CLEAR [1]:** clear temporary register **(TR).**
- ✓ **CLEAR [2]:** clear data register **(DR).**
- ✓ **CLEAR [3]:** clear program counter **(PC).**
- ✓ **CLEAR [4]:** clear a memory address register **(AR)**

S selectors

- ✓ **000 (S [2:0] = 0)**: No selection (**bus is idle or tri-stated**).
- ✓ **001 (S [2:0] = 1)**: selects the **AR (address register)**.
- ✓ **010 (S [2:0] = 2)**: selects the **PC (program counter)**.
- ✓ **011 (S [2:0] = 3)**: selects the **DR (data register)**.
- ✓ **100 (S [2:0] = 4)**: selects the **AC (accumulator)**.
- ✓ **101 (S [2:0] = 5)**: selects the **IR (instruction register)**.
- ✓ **110 (S [2:0] = 6)**: selects the **TR (temporary register)**.
- ✓ **111 (S [2:0] = 7)**: selects the **Memory (read only)**.

Sequence Counter

- ✓ **CLOCK** : triggers the counter to evaluate its next state on the **rising edge.**

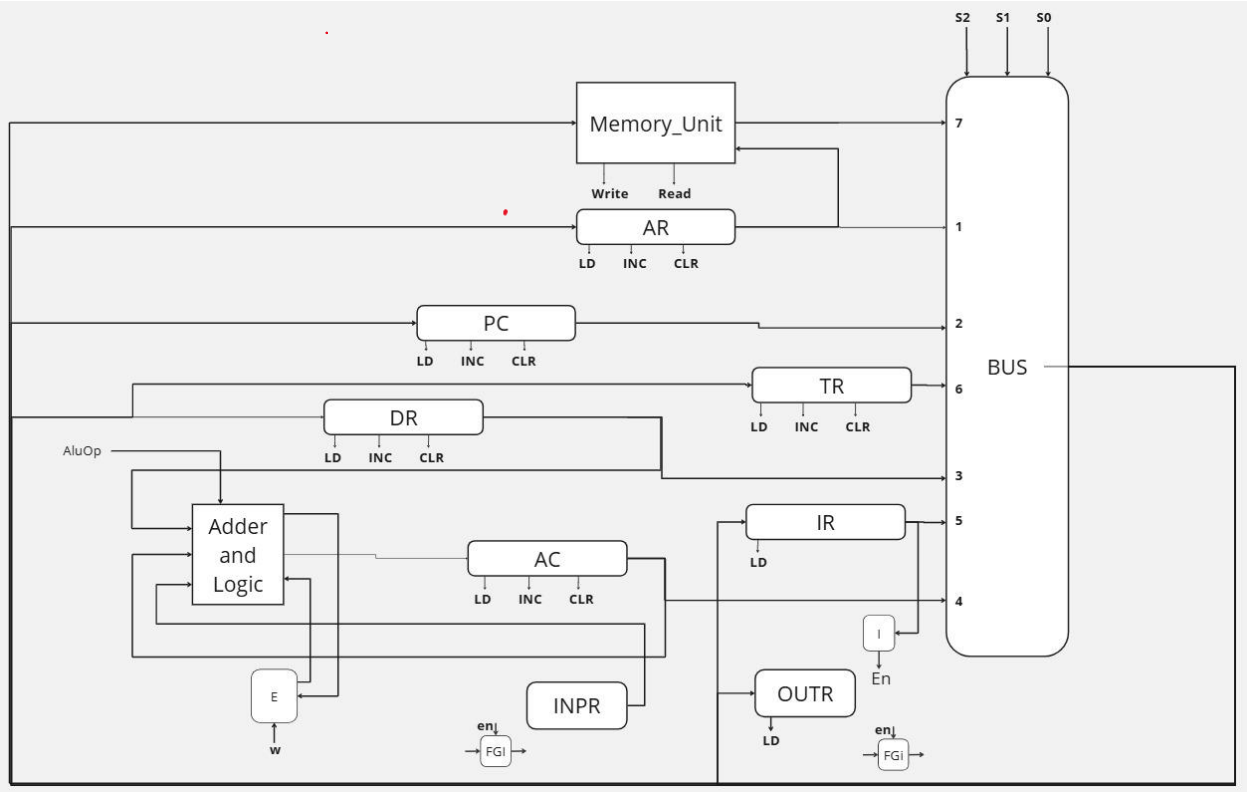
- ✓ **Increment (INC)**: when **high (1),** the counter moves to the next state in the sequence; when **low (0),** the counter holds its current state.

- ✓ **CLEAR (CLR)**: resets the counter to its initial state **(e.g., 000 for a 3-bit counter)** and has higher priority over the increment sign

Animating the Datapath

(Made by Amad Alhaj, Nermeen Nedal, Takreet Omar Alzyadat)

Animation plays a transformative role in the design and analysis of data paths in digital systems. A data path, comprising core components such as the ALU, registers, flip flops, and memory units, is fundamental to the functionality of any processor. Understanding the flow of data and signals across these interconnected units is crucial for ensuring accuracy, efficiency, and optimal performance.



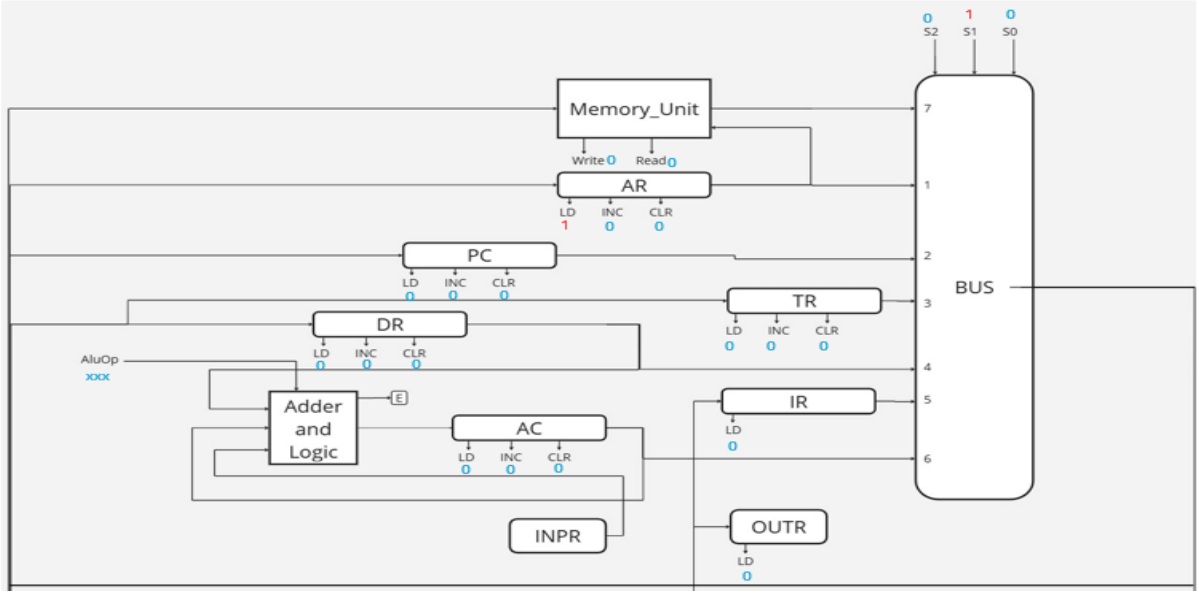
In our work, we leveraged animation to gain a deeper understanding of how the data path operates at every level of execution. By animating the movement of signals and instructions, we were able to track how control signals are generated, how they activate specific components, and how data flows seamlessly between them. Additionally, animation proved invaluable in communicating our design to others, whether for educational purposes or collaborative discussions. It bridged the gap between theoretical ideas and practical implementation, presenting the data path not just as a static design but as a dynamic, functional system that can be easily understood and improved.

1) Fetch Instruction:

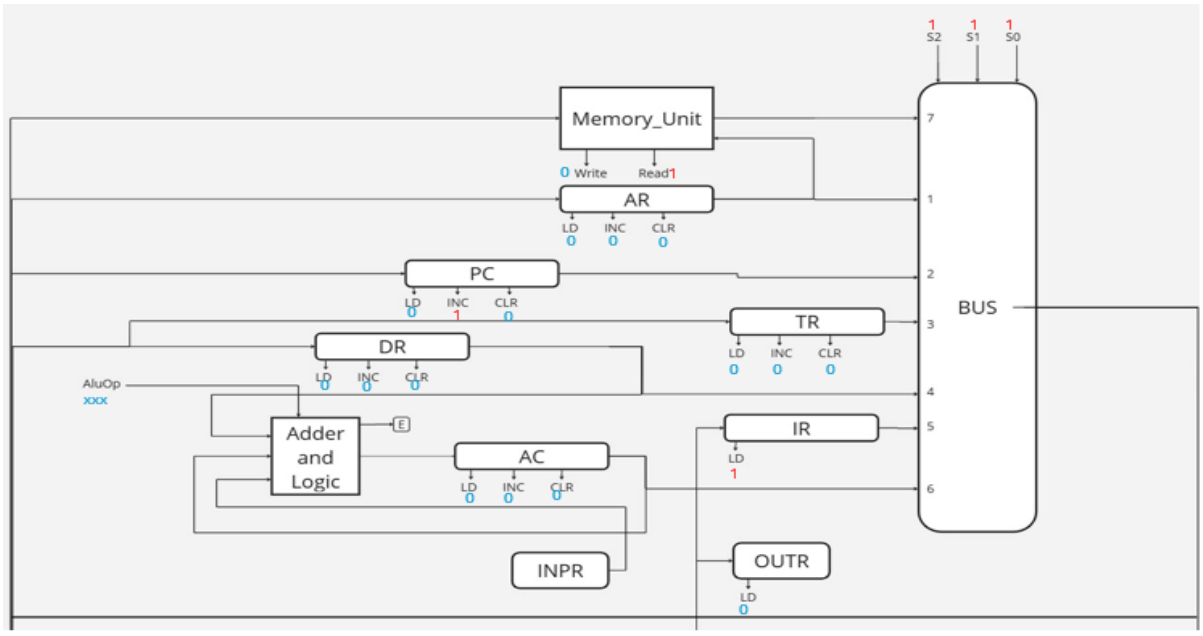
For each instruction, we have two common steps to read the instruction from memory and prepare it inside the register. To deal with this process, we need two cycles to fetch the instruction.

T0	T1
AR←PC	IR←M[AR]
---------	PC ←PC+1

T0: (AR←PC)



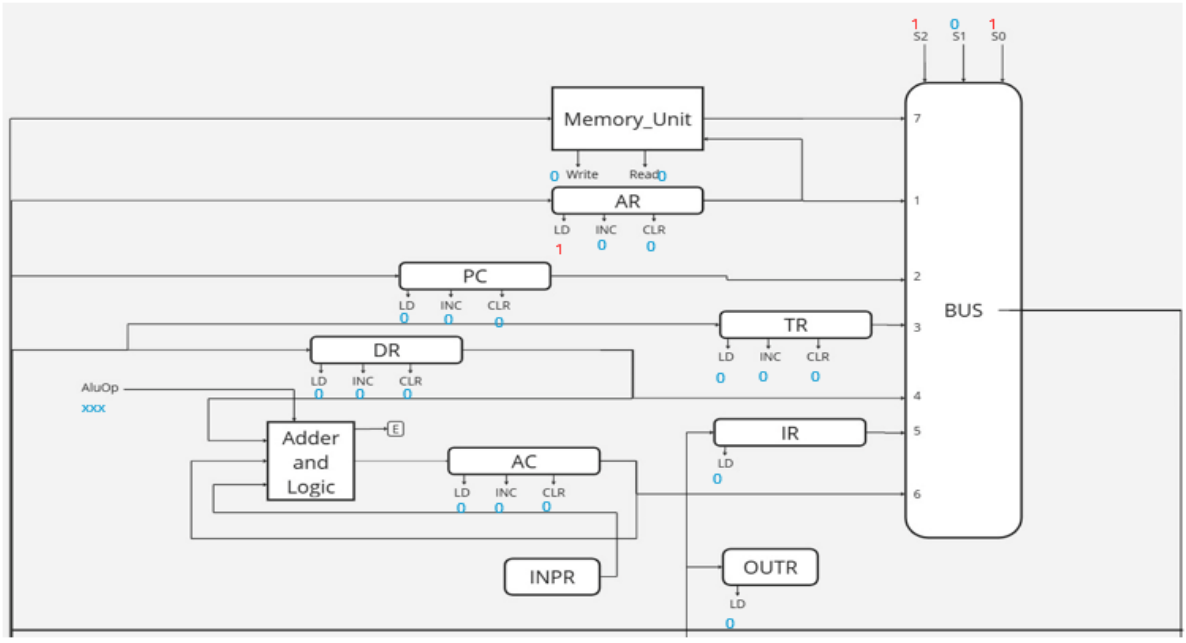
T1: (IR←M[AR], PC ←PC+1)



2) Decode instruction:

T2
D0..D7 ← Decode IR(12-14)
AR ← IR(0-11)
I ← IR(15)

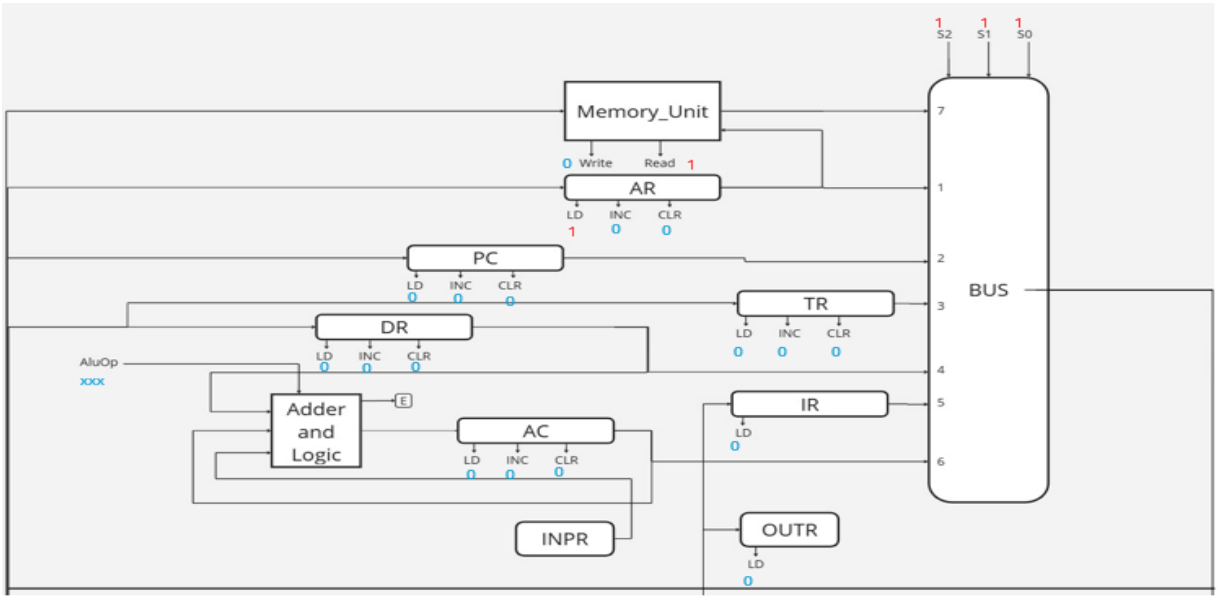
T2: (AR ← IR(0-11))



3) Direct and Indirect:

If (I=0) I` T3	If (I=0) -- IT3
Nop	AR ← M[AR]

IT3: (AR ← M[AR])



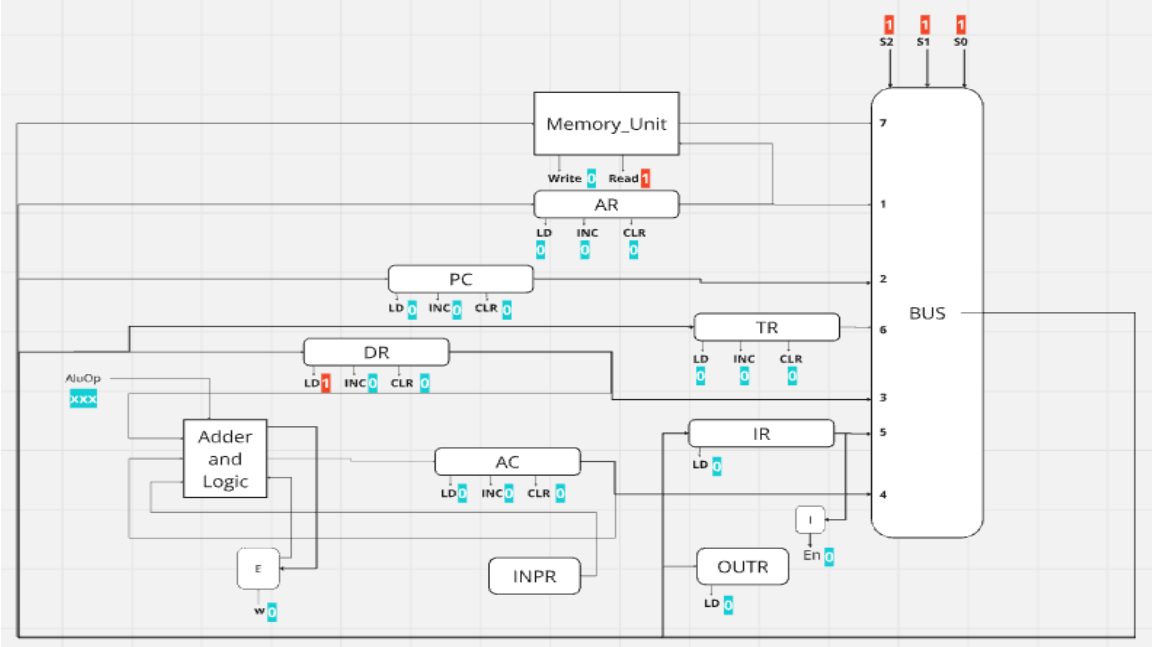
4) Execute:

4.1 Memory reference Instruction:

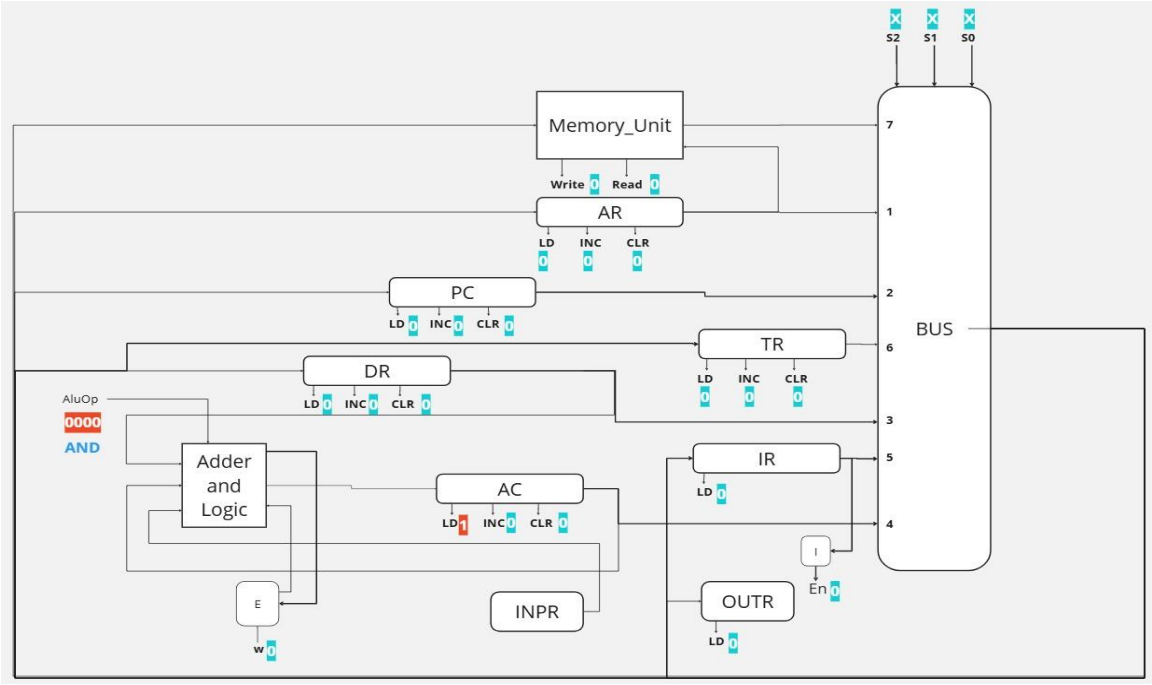
a) AND Instructions:

T4	T5
DR ← M[AR]	AC ← AC ^ DR

T4: DR ← M[AR]



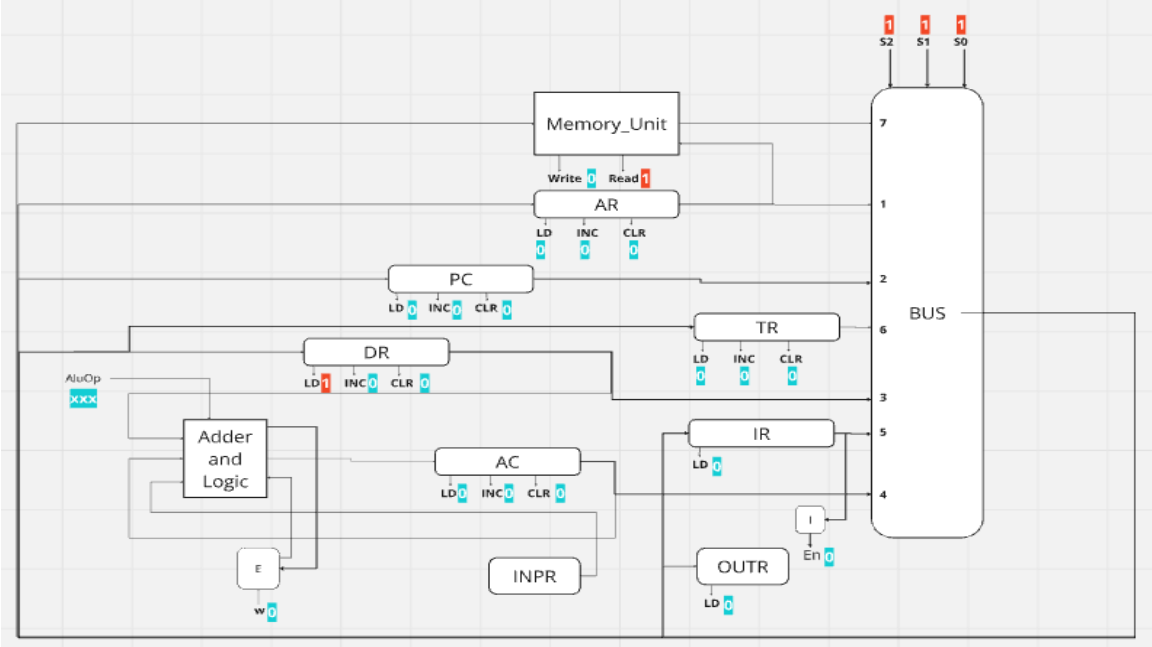
T5: AC ← AC ^ DR



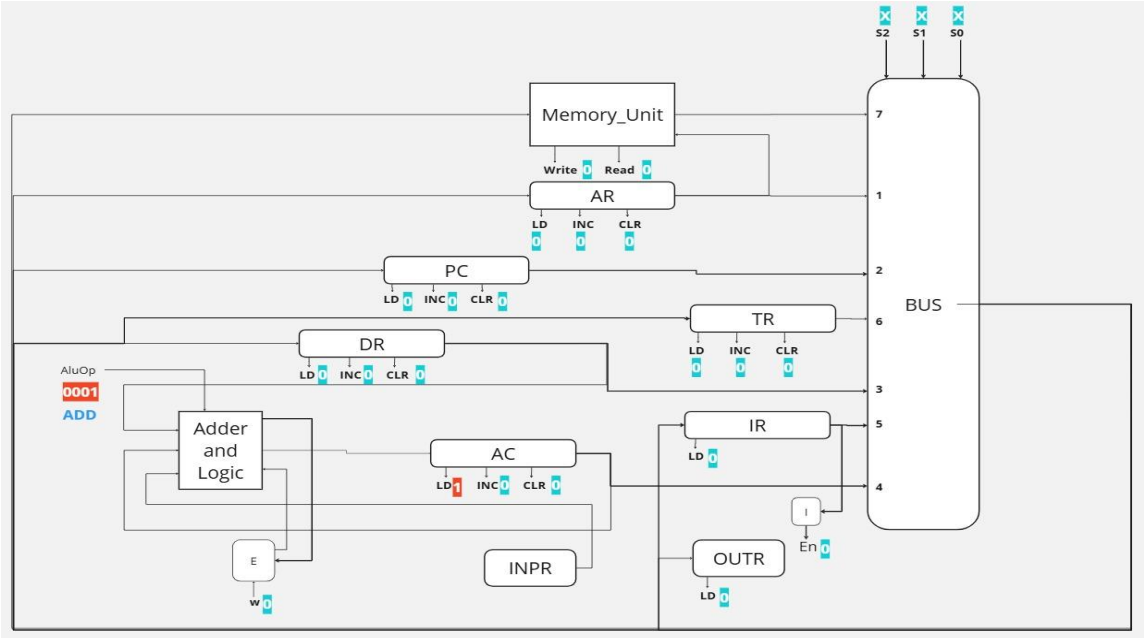
b) ADD Instruction:

T4	T5
DR ← M[AR]	AC ← AC + DR

T4: DR ← M[AR]



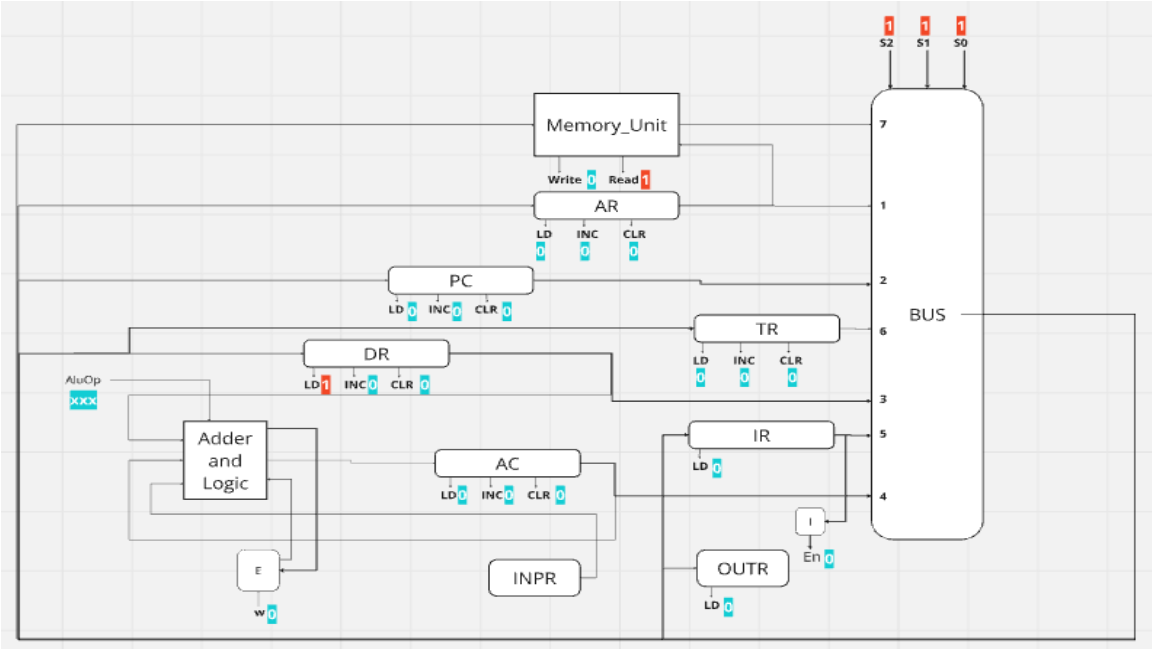
T5: AC ← AC + DR



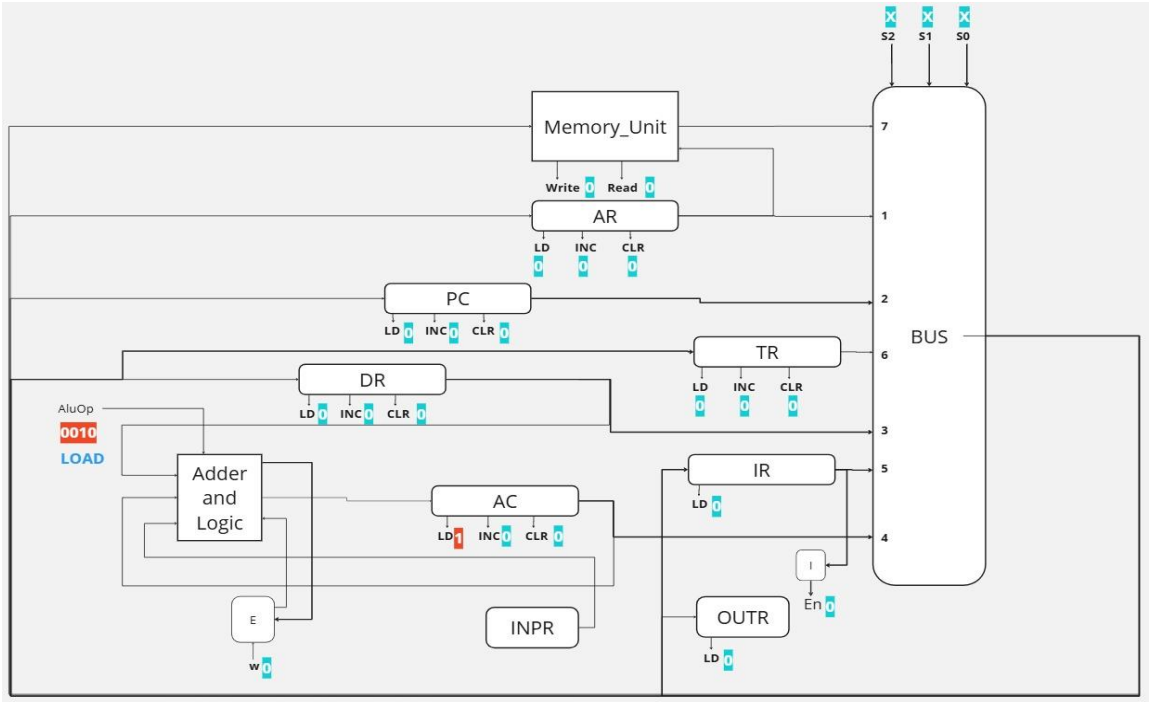
c) LDA Instruction:

T4	T5
DR ← M[AR]	AC ← DR

T4: DR ← M[AR]



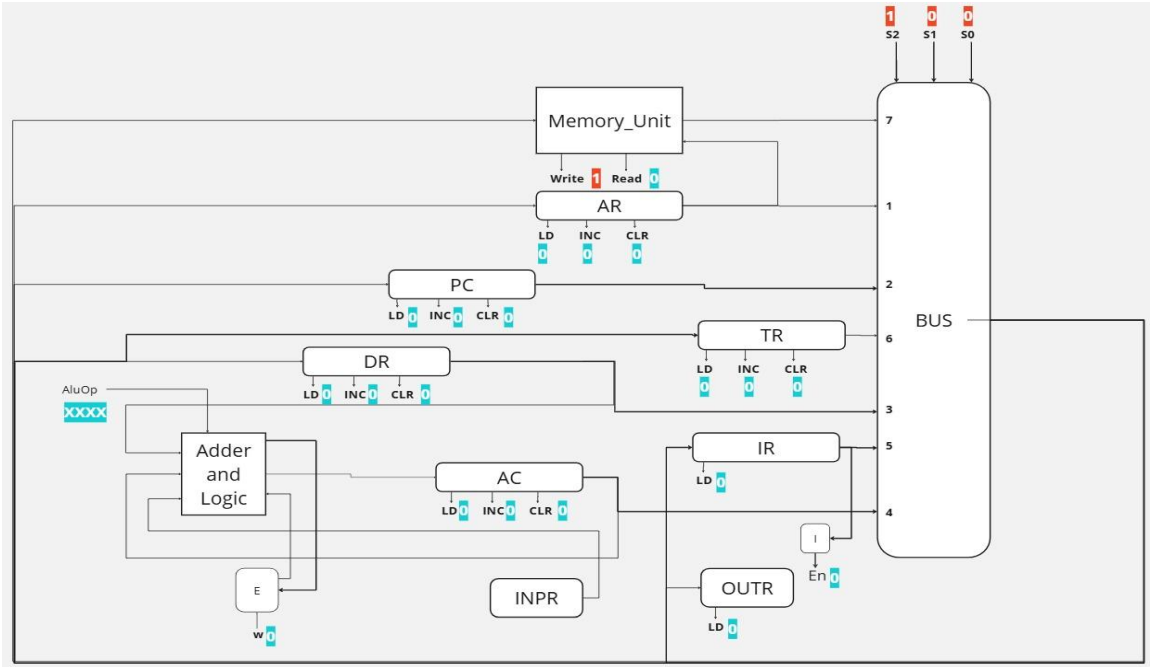
T5: AC ← DR



d) STA Instruction:

T4
M[AR] ← AC

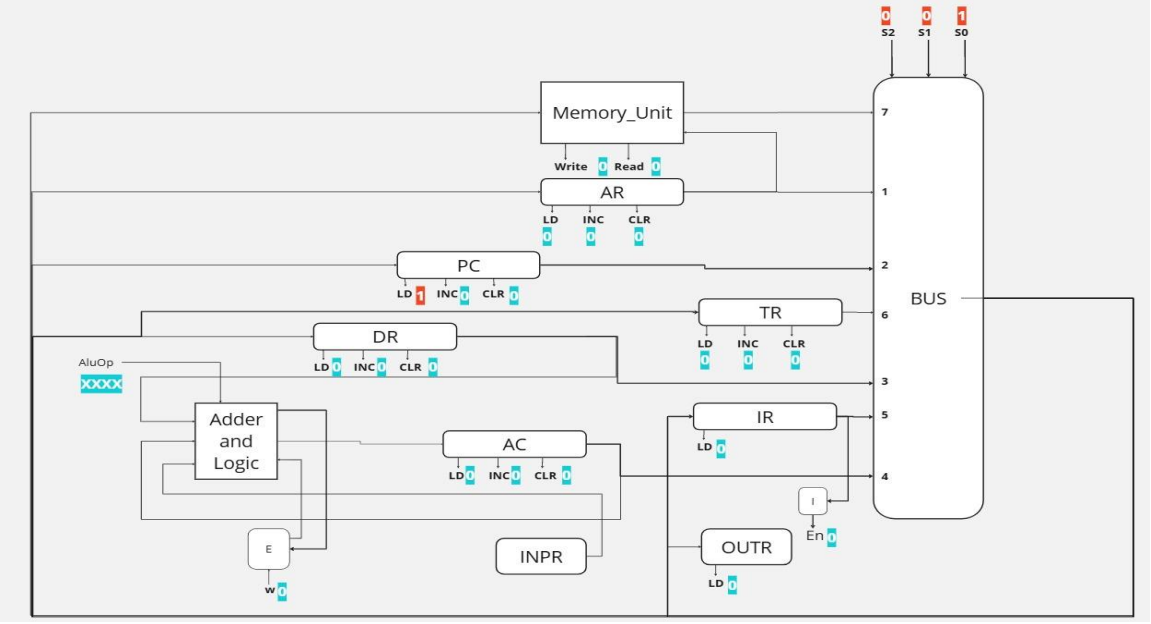
T4: M[AR] ← AC



e) BUN Instruction:

T4
PC ← AR

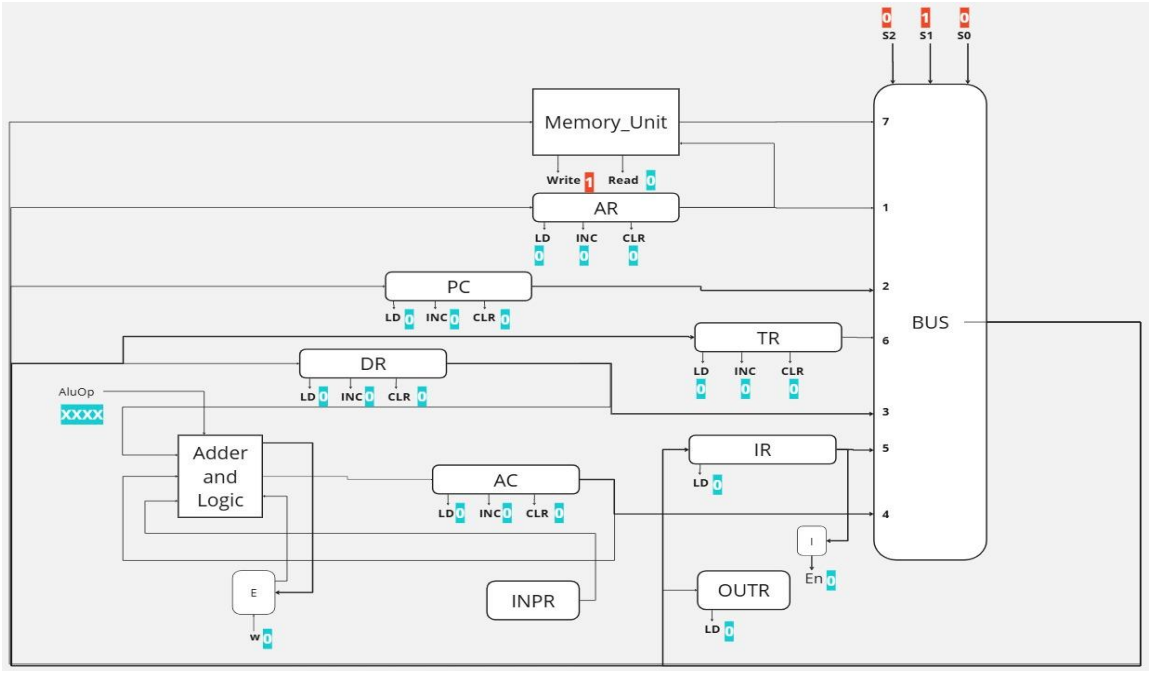
T4: PC ← AR



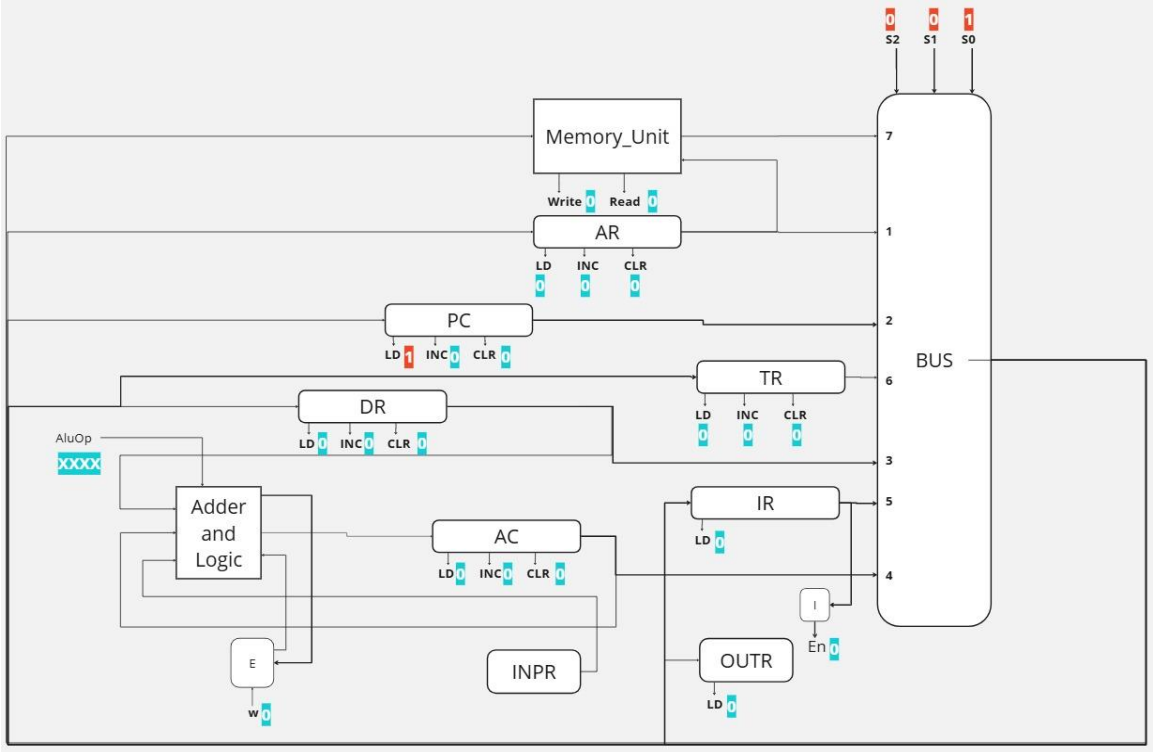
f) BSA Instruction:

T4	T5
M[AR] ← PC	PC ← AR

T4: M[AR] ← PC



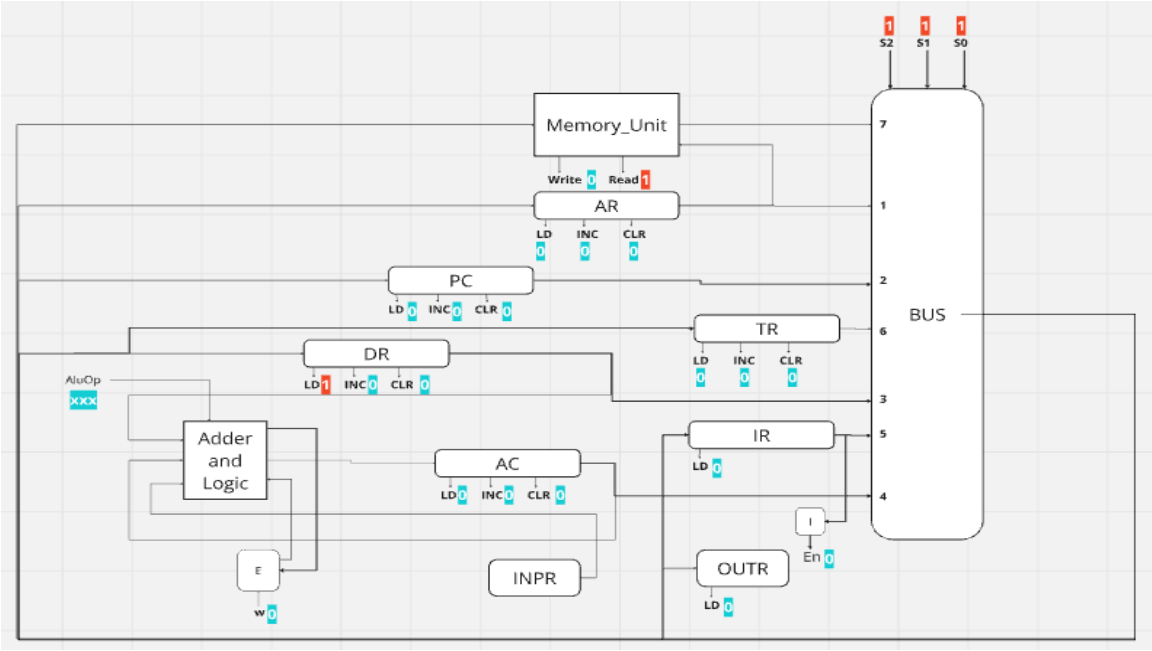
T5: PC ← AR



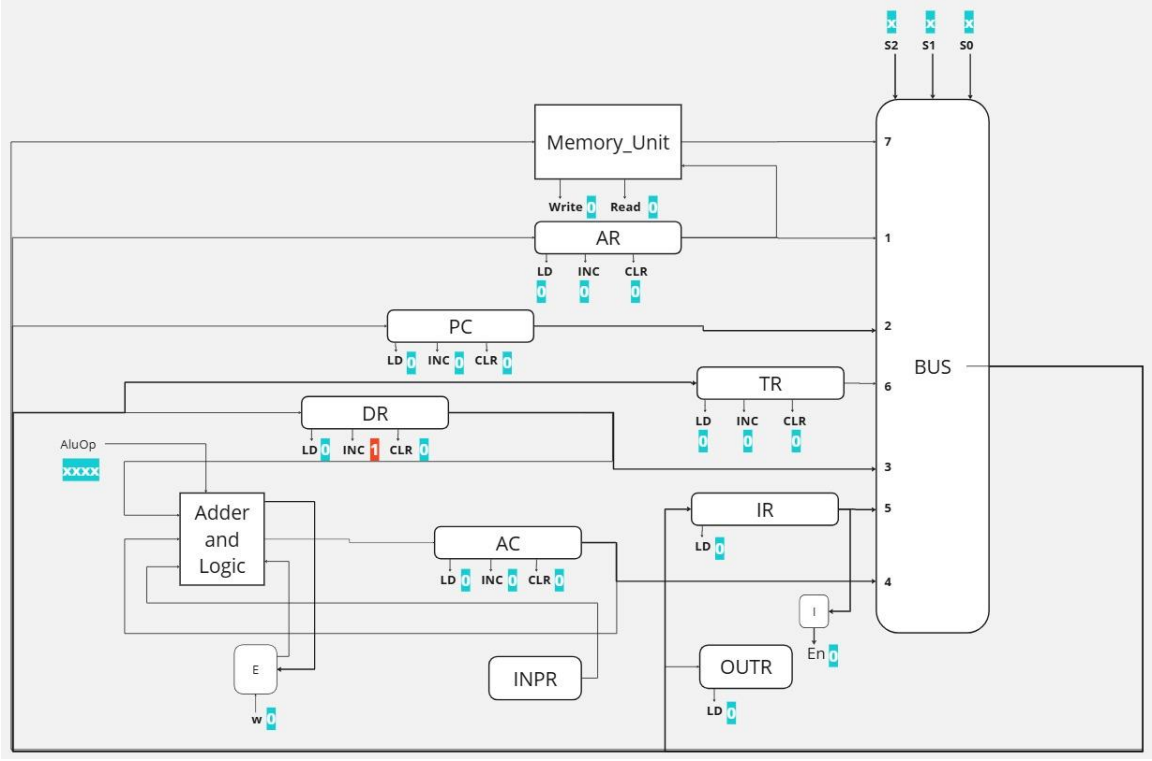
g) ISZ Instruction:

T4	T5	T6
DR ← M[AR]	**DR ← DR +1**	M[AR] ← DR
--------	--------	DR ← DR +1
--------	--------	

T4: DR ← M[AR]

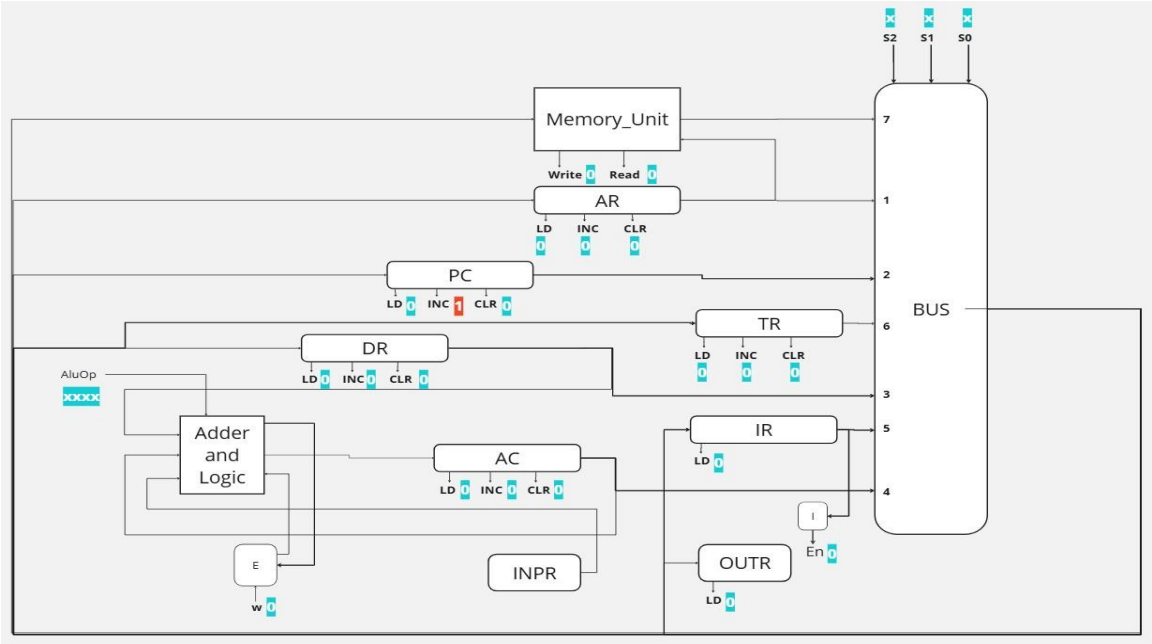


T4: **DR ← DR +1**



T5: M[AR] ← DR , DR ← DR +1

If (DR=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

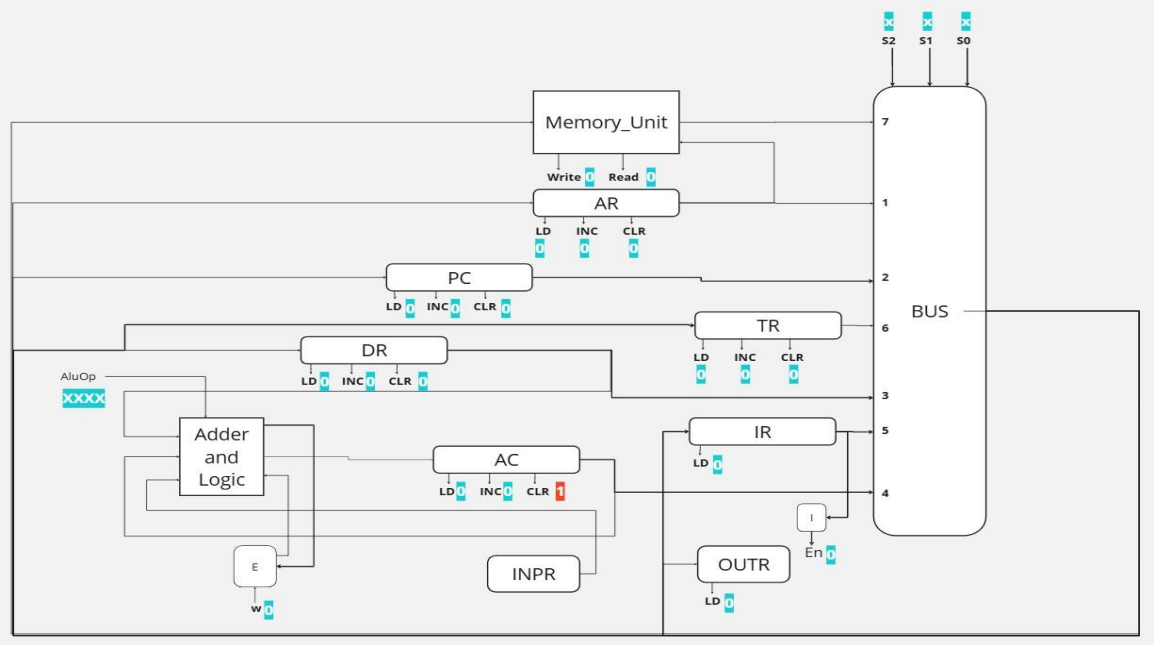


4.2 Register reference Instruction: Note: D_7 I` T_3 = r

a) CLA Instruction:

rB_{11}
AC ← 0

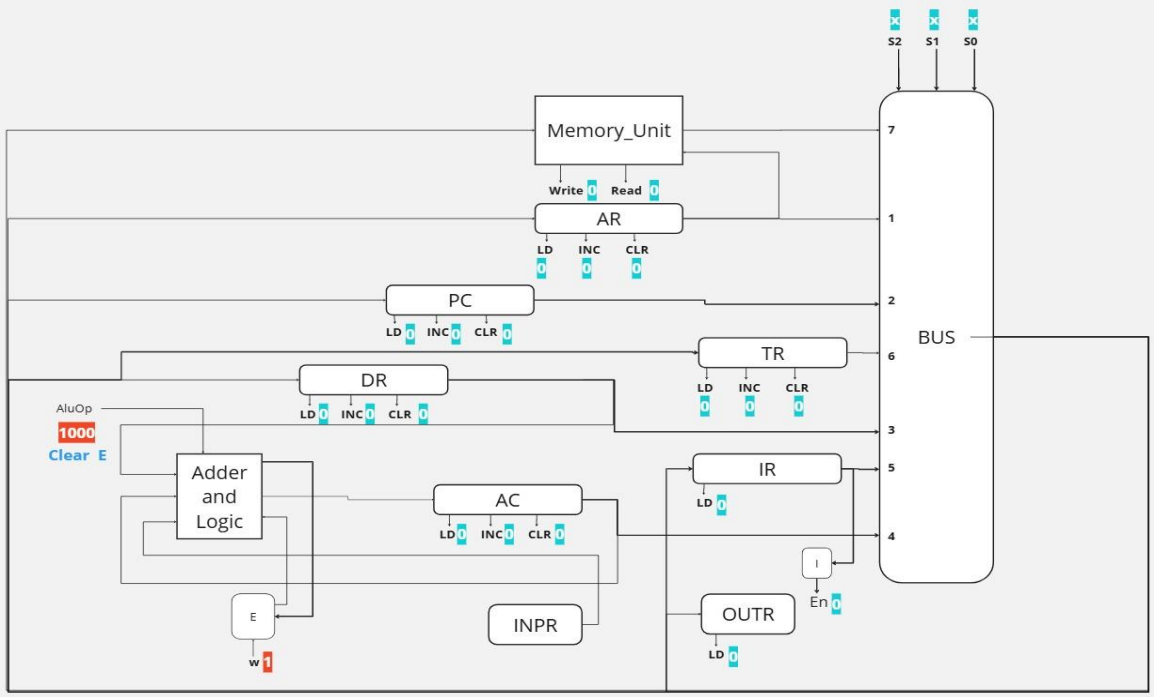
T3: AC ← 0



b) CLE Instruction:

rB_{10}
E ← 0

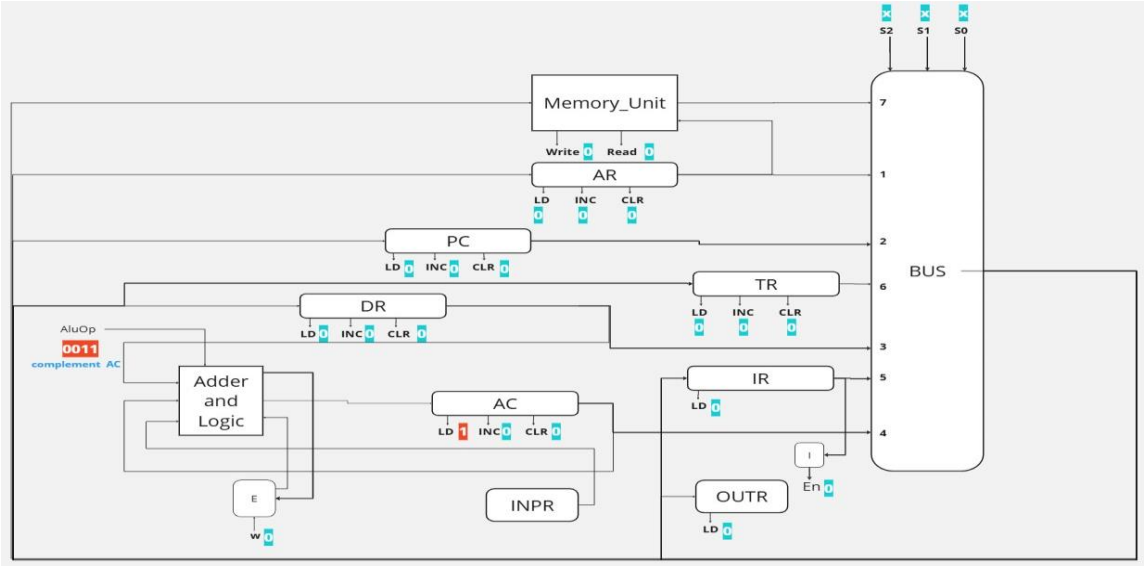
T3: E ← 0



c) CMA Instruction:

rB_9
$AC \leftarrow \overline{AC}$

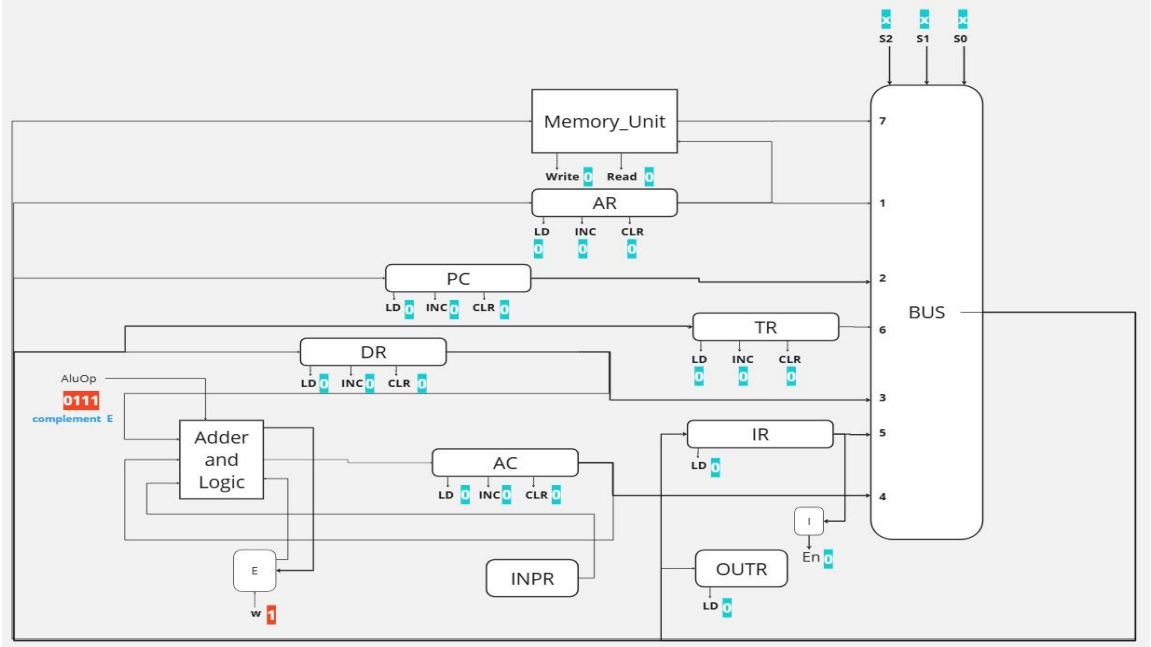
T3: $AC \leftarrow \overline{AC}$



d) CME Instruction:

rB_8
$E \leftarrow \overline{E}$

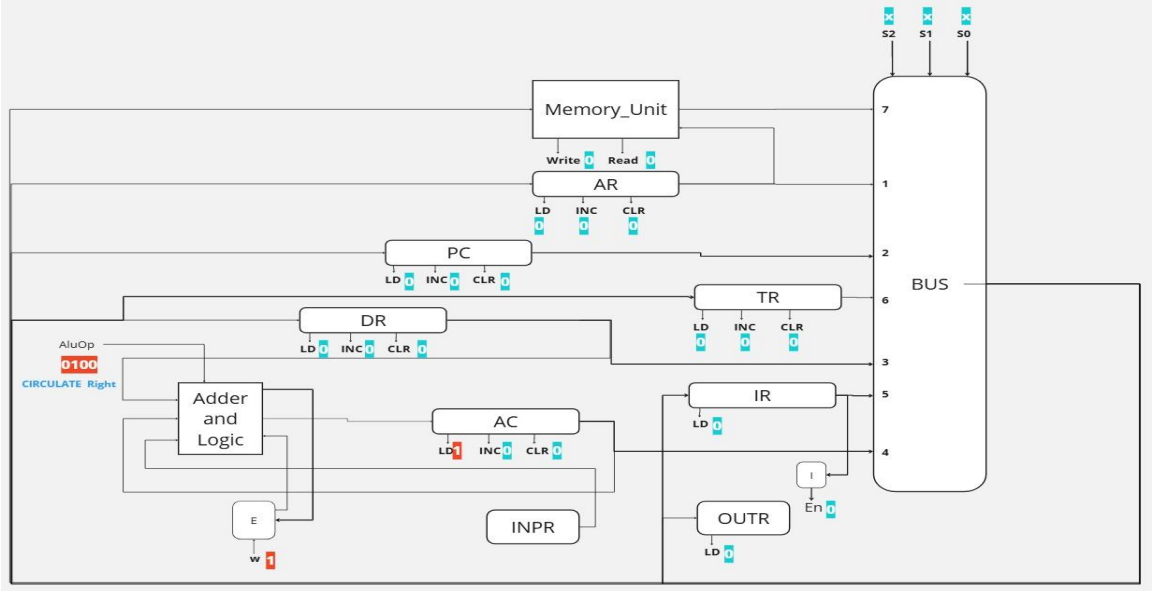
T3: $E \leftarrow \overline{E}$



e) CIR Instructions:

rB_7
AC ← shr AC
AC[15] ← E
E ← AC[0]

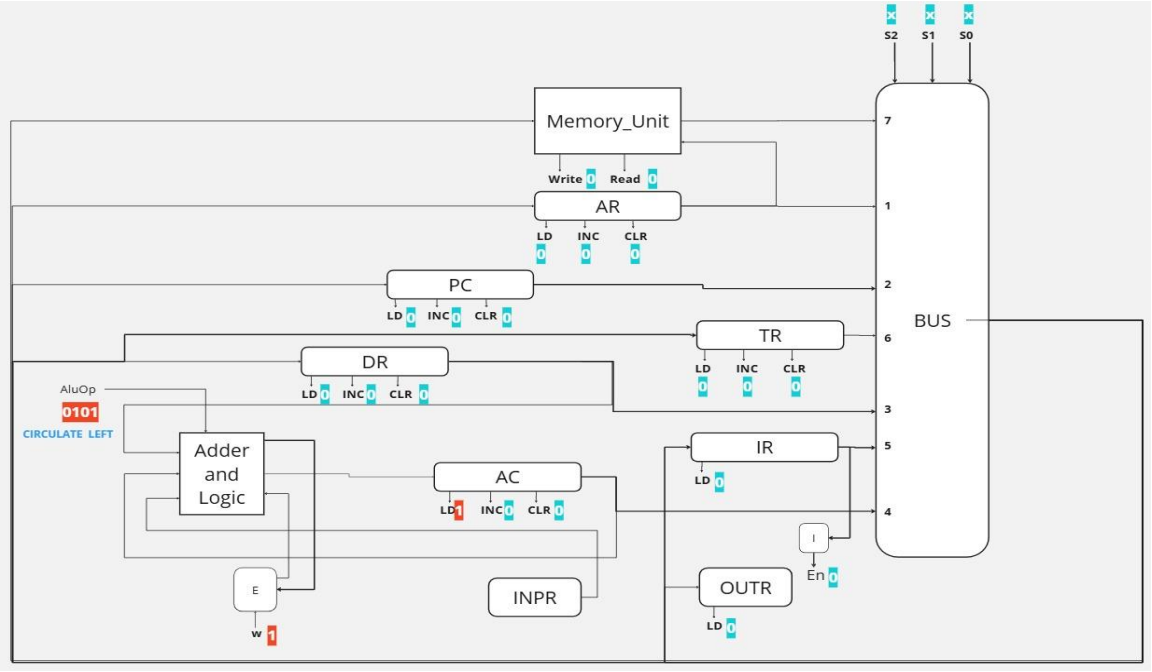
T3: AC ← shr AC, AC[15] ← E, E ← AC[0]



f) CIL Instructions:

rB_6
AC ← shl AC
AC[0] ← E
E ← AC[15]

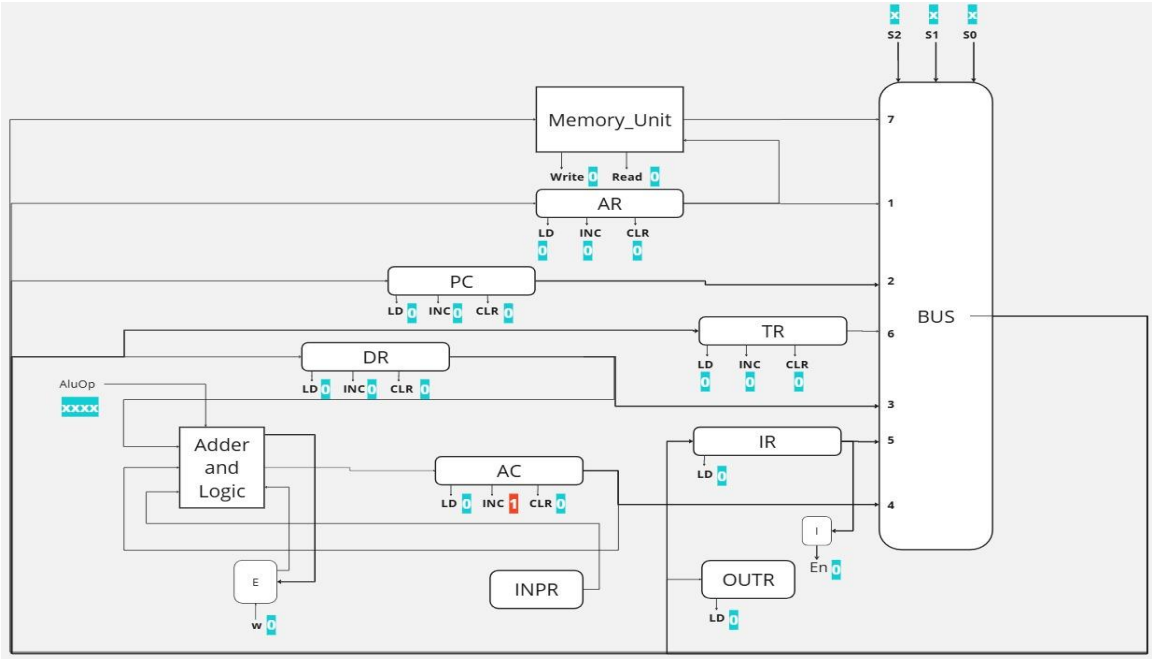
T3: AC ← shl AC, AC[0] ← E , E ← AC[15]



g) INC Instruction:

rB_8
AC ← AC +1

T3: AC ← AC +1

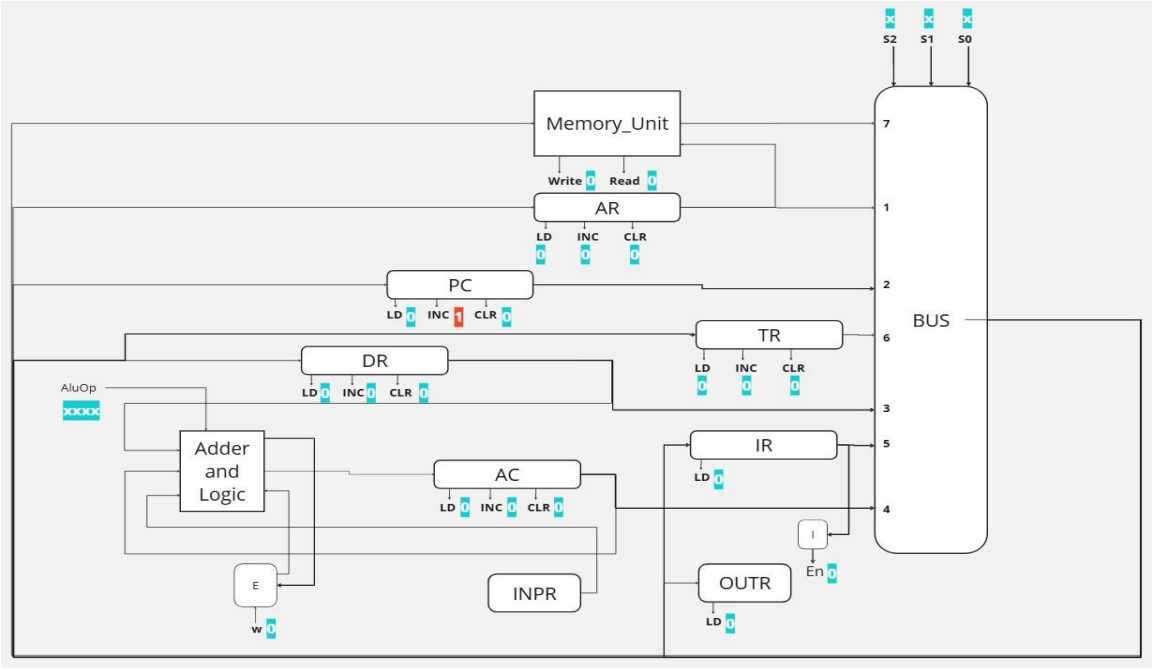


h) SPA Instruction:

If (AC[15]=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

rB_8
If (AC[15] == 0) Then PC ← PC +1

T3: If (AC[15] == 0) Then PC ← PC +1

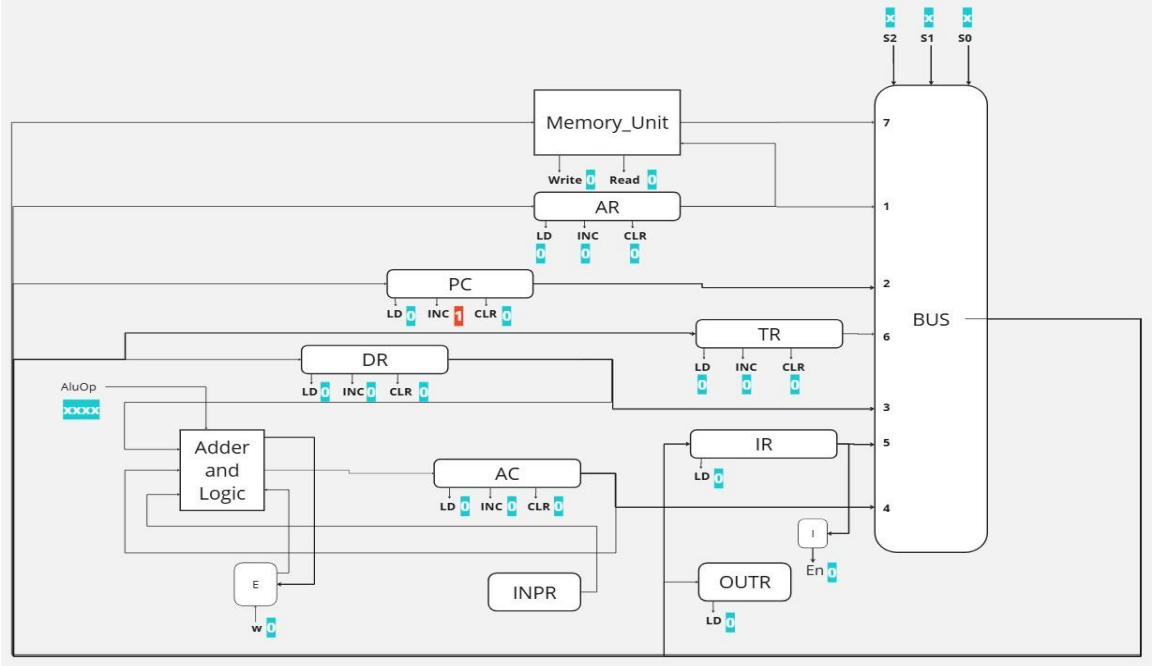


i) SNA Instruction:

If (AC[15]=1), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

rB_8
If (AC[15] == 1) Then PC ← PC +1

T3: If (AC[15] == 1) Then PC ← PC +1

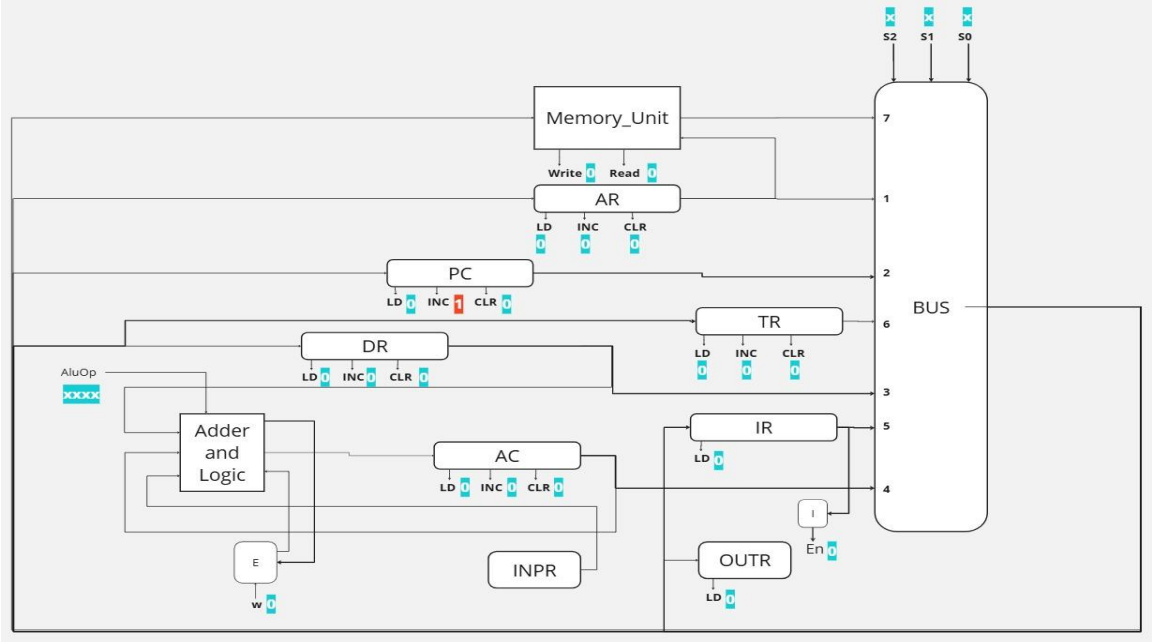


j) SZA Instruction:

If (AC=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

rB_8
If (AC == 0) Then PC ← PC + 1

T3: If (AC == 0) Then PC ← PC + 1

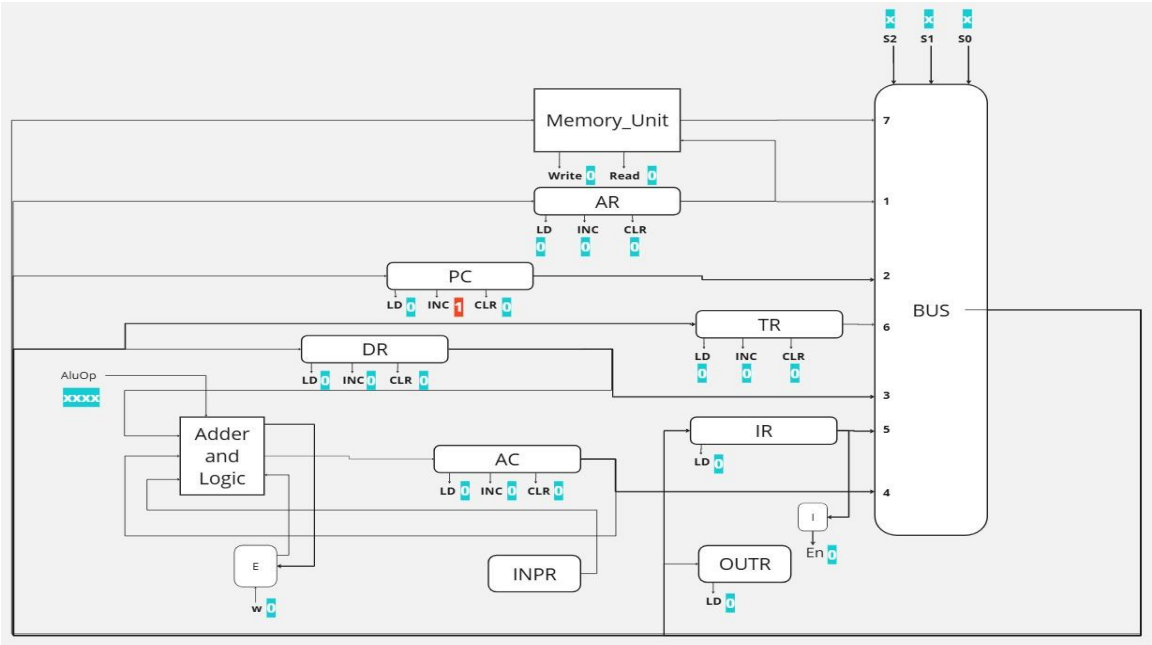


k) SZE Instruction:

If (E=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

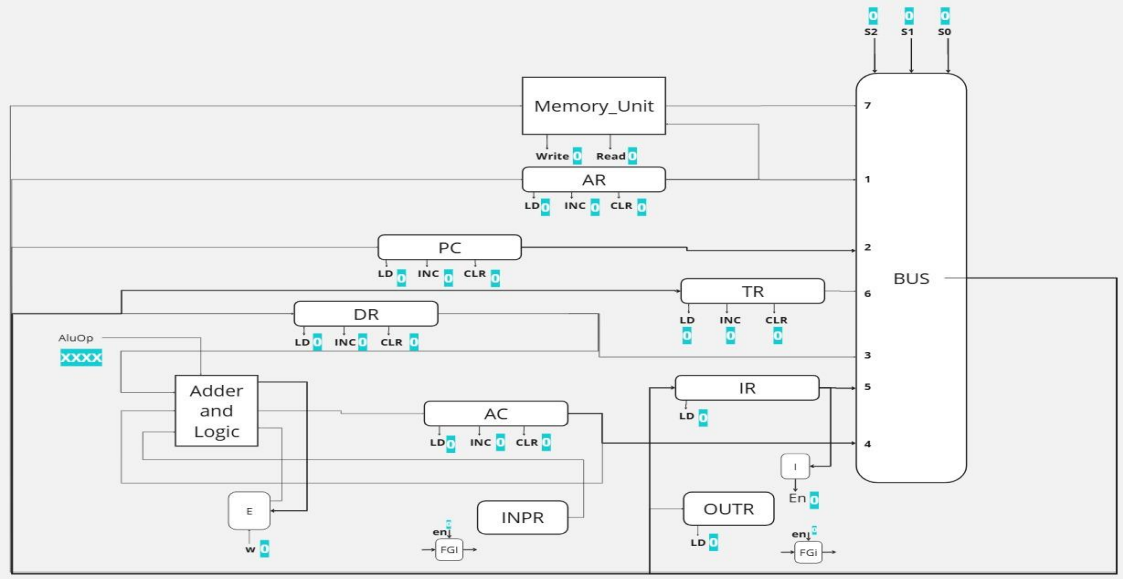
rB_8
If (E == 0) Then PC ← PC + 1

T3: If (E == 0) Then PC ← PC + 1



l) HLT Instructions:

The HLT (Halt) instruction in a bus architecture is a control signal or machine instruction that stops the execution of the processor. It is typically used to indicate the end of a program or to place the processor in a low-power or idle state until further action, such as a reset or interrupt, occurs. Reset all signal and stop sequence counter;

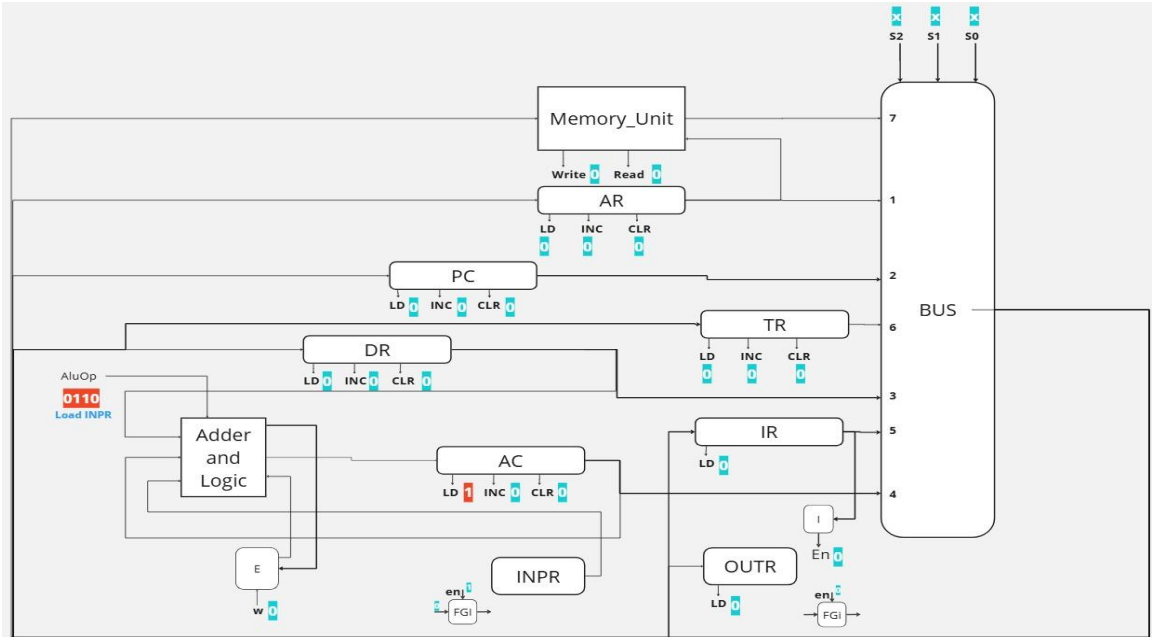


4.3 Input/Output reference Instruction: Note: $D_7 I T_3 = p$

a) INP Instruction

pB_{11}
AC(0-7) ← INPR , FGI ← 0

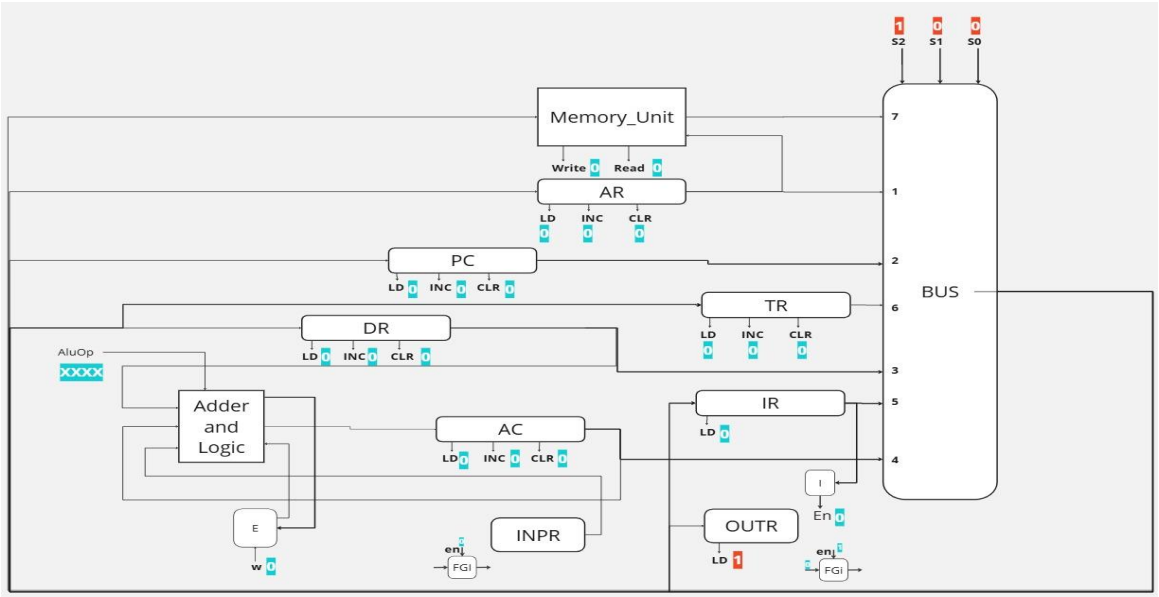
T3: AC(0-7) ← INPR , FGI ← 0



b) OUT Instruction:

pB_{11}
OUT ← AC(7-0) , FG0 ← 0

T4: OUT ← AC(7-0) , FG0 ← 0

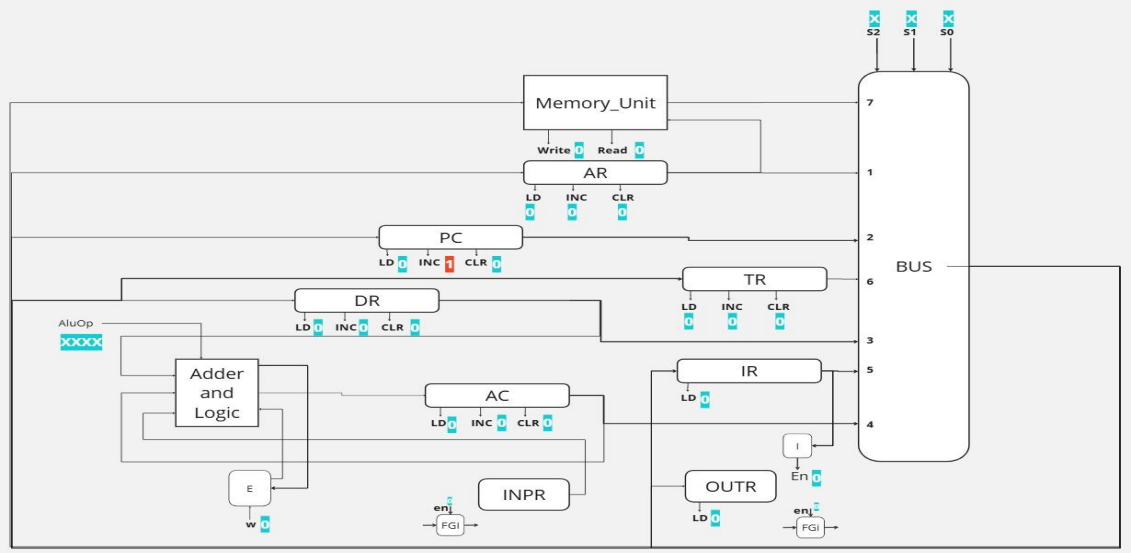


c) SKI Instruction:

If (FGI=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

pB_{11}
If (FGI == 0) Then PC ← PC + 1, FGI ← 0

T3 : If (FGI == 0) Then PC ← PC + 1, FGI ← 0

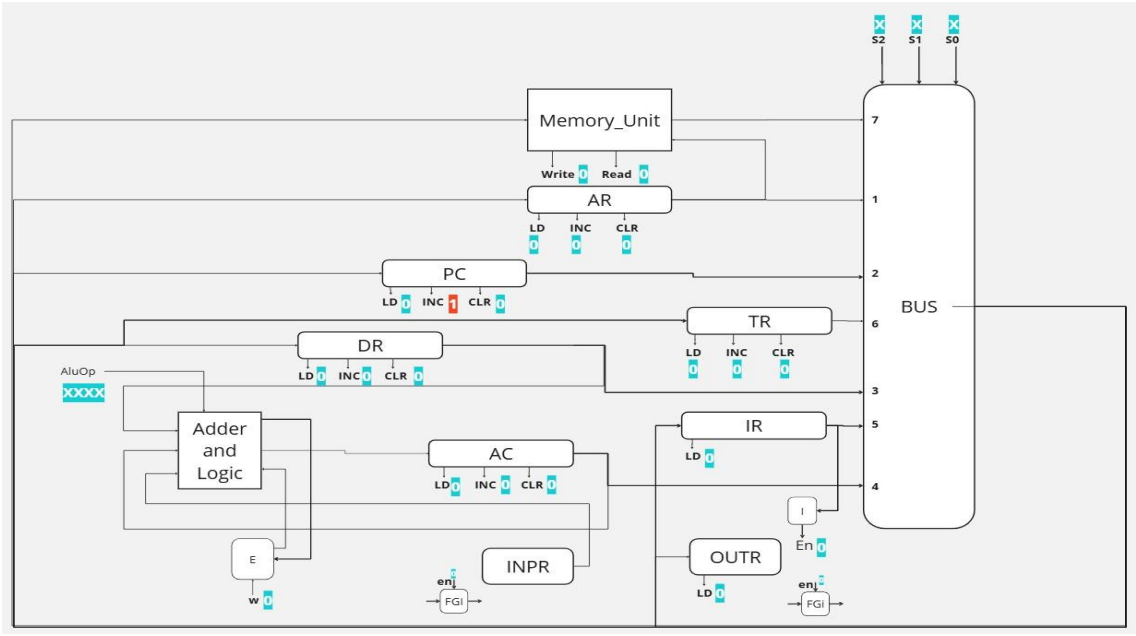


d) SKO Instruction:

If (FGO=0), then (PC=PC+1). If this statement is true, we proceed to execute this statement.

pB_{11}
If (FGO == 0) Then PC ← PC + 1, FGI ← 0

T3: If (FGO == 0) Then PC ← PC + 1, FGI ← 0



TESTING

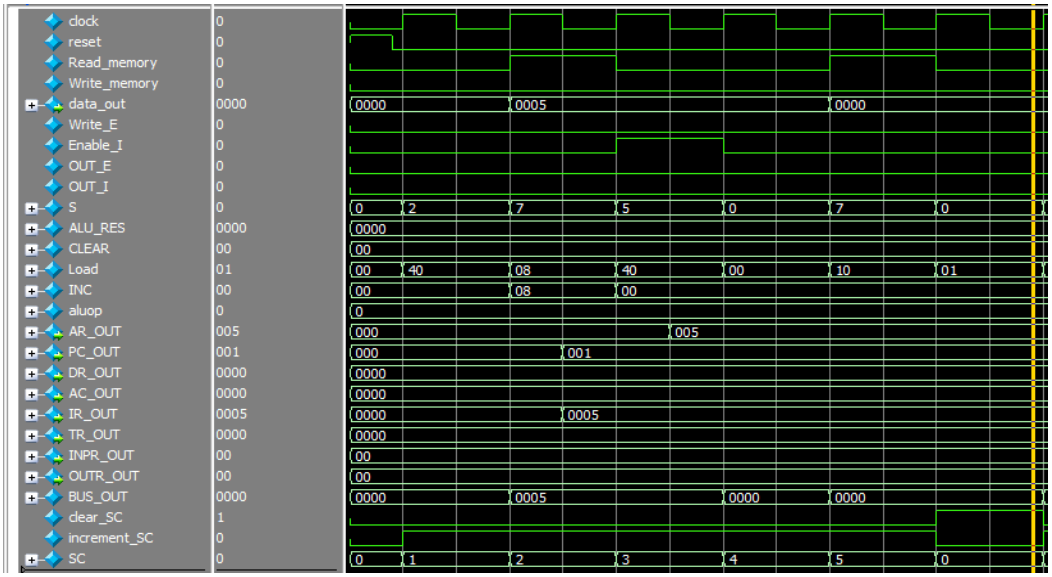
(Made by Amad Alhaj , Nermeen Nedal and Takreet Alzyadat)

1) Memory Reference Instructions:

instruction	Machine code.	T0	T1	T2	T3 IF(I=1)	T4	T5	T6
AND	0XXX OR 8XXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	AR<-MEM[AR]	DR<-M[AR]	AC<-AC^DR	--------
ADD	1XXX OR 9XXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	AR<-MEM[AR]	DR<-M[AR]	AC<-AC+DR	--------
LDA	2XXX OR AXXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AR<-MEM[AR]	DR<-M[AR]	AC<- DR	--------
STA	3XXX OR BXXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AR<-MEM[AR]	AC<-M[AR]	--------	--------
BUN	4XXX OR CXXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AR<-MEM[AR]	PC<- AR	--------	--------
BSA	5XXX OR DXXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AR<-MEM[AR]	M[AR]<-PC AR<-AR+1	PC<- AR	--------
ISZ	6XXX OR EXXX	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AR<-MEM[AR]	DR<-M[AR]	DR <- DR +1	M[AR]<-DR IF(DR=0) ?PC+1 : PC;

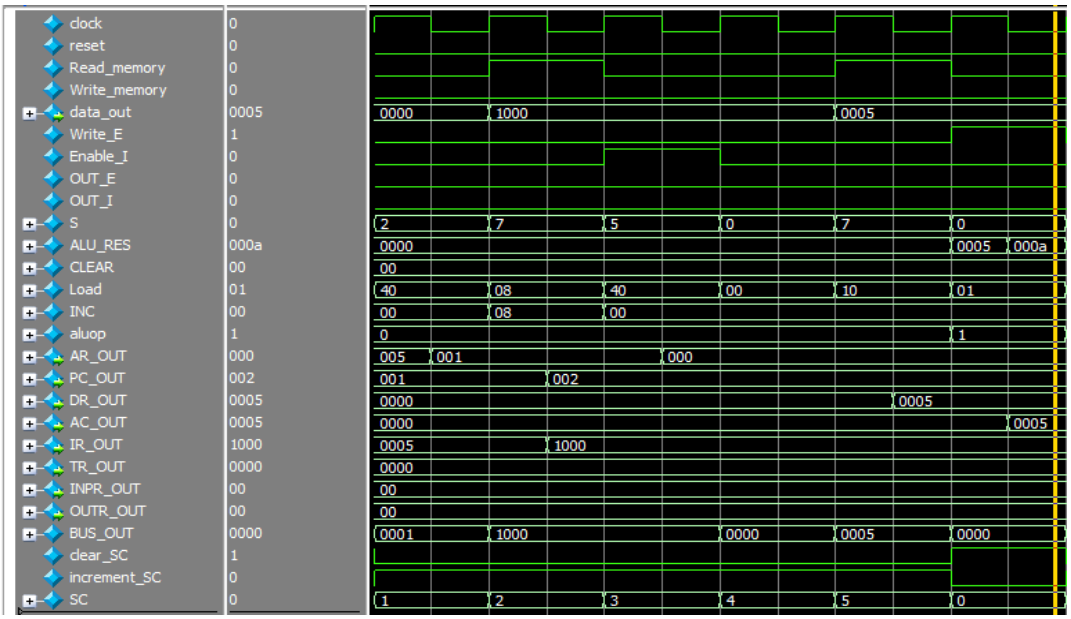
---

1) Instruction 1: AND, machine code (0XXX or 8XXX), Example: AND, machine code (0005)



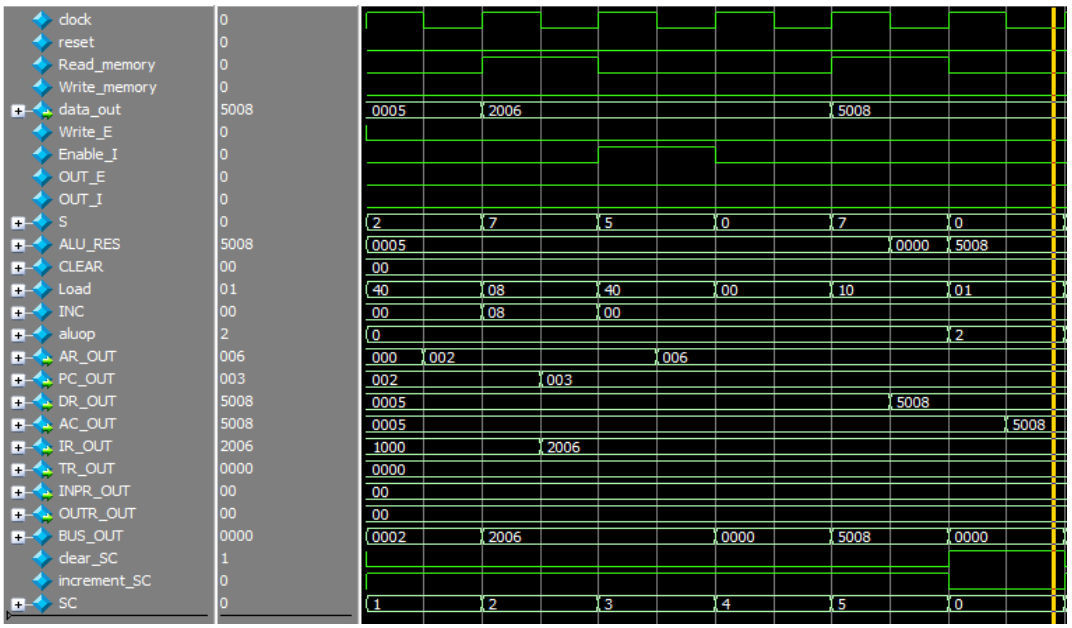

```
# Time    Load     INC     Clear   S       aluop   Read_memory  Write_memory  Write_E  Enable_I
# 0ns     0000000  00000   00000   000     0000    0            0             0        0
# 5ns     1000000  00000   00000   010     0000    0            0             0        0
# 15ns    0001000  01000   00000   111     0000    1            0             0        0
# 25ns    1000000  00000   00000   101     0000    0            0             0        1
# 35ns    0000000  00000   00000   000     0000    1            0             0        0
# 45ns    0010000  00000   00000   111     0000    1            0             0        0
# 55ns    0000001  00000   00000   000     0000    0            0             0        0
```

2) Instruction 1: ADD , machine code (1XXX or 9XXX) , Example: ADD, machine code(1000)



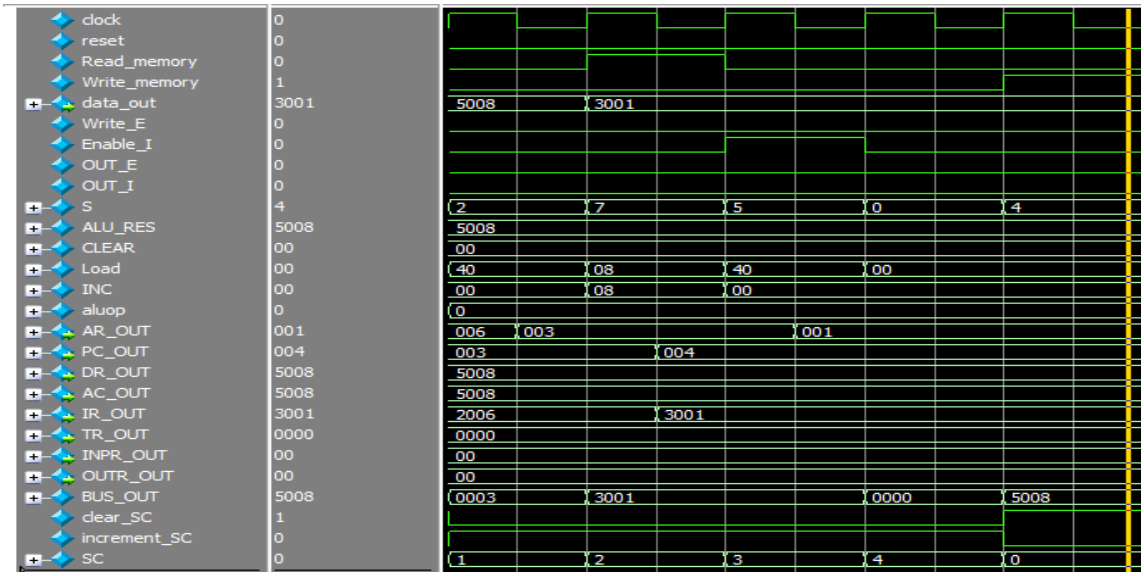
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 65ns	1000000	00000	00000	010	0000	0	0	0	0
# 75ns	0001000	01000	00000	111	0000	1	0	0	0
# 85ns	1000000	00000	00000	101	0000	0	0	0	1
# 95ns	0000000	00000	00000	000	0000	1	0	0	0
# 105ns	0010000	00000	00000	111	0000	1	0	0	0
# 115ns	0000001	00000	00000	000	0001	0	0	1	0

3) instruction 3 : LDA , machine code (2XXX or AXXX) , Example: ADD, machine code(2006)



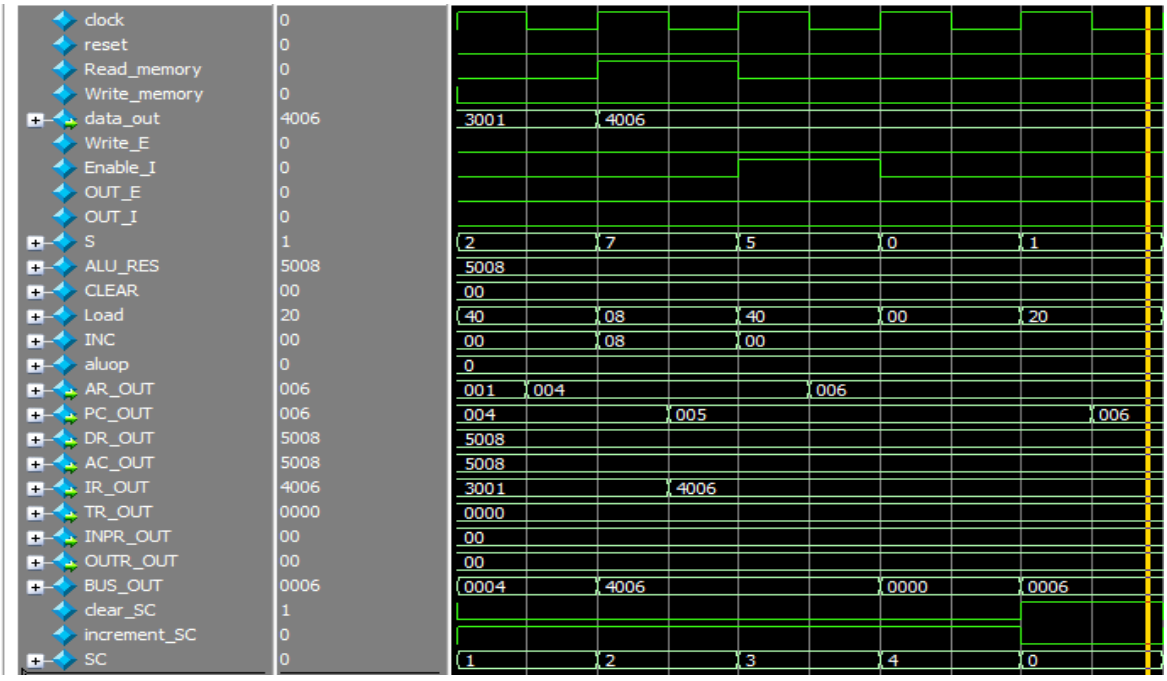
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 125ns	1000000	00000	00000	010	0000	0	0	0	0
# 135ns	0001000	01000	00000	111	0000	1	0	0	0
# 145ns	1000000	00000	00000	101	0000	0	0	0	1
# 155ns	0000000	00000	00000	000	0000	1	0	0	0
# 165ns	0010000	00000	00000	111	0000	1	0	0	0
# 175ns	0000001	00000	00000	000	0010	0	0	0	0

4) instruction 4 : STA machine code (3XXX or BXXX) , Example: ADD, machine code(3001)



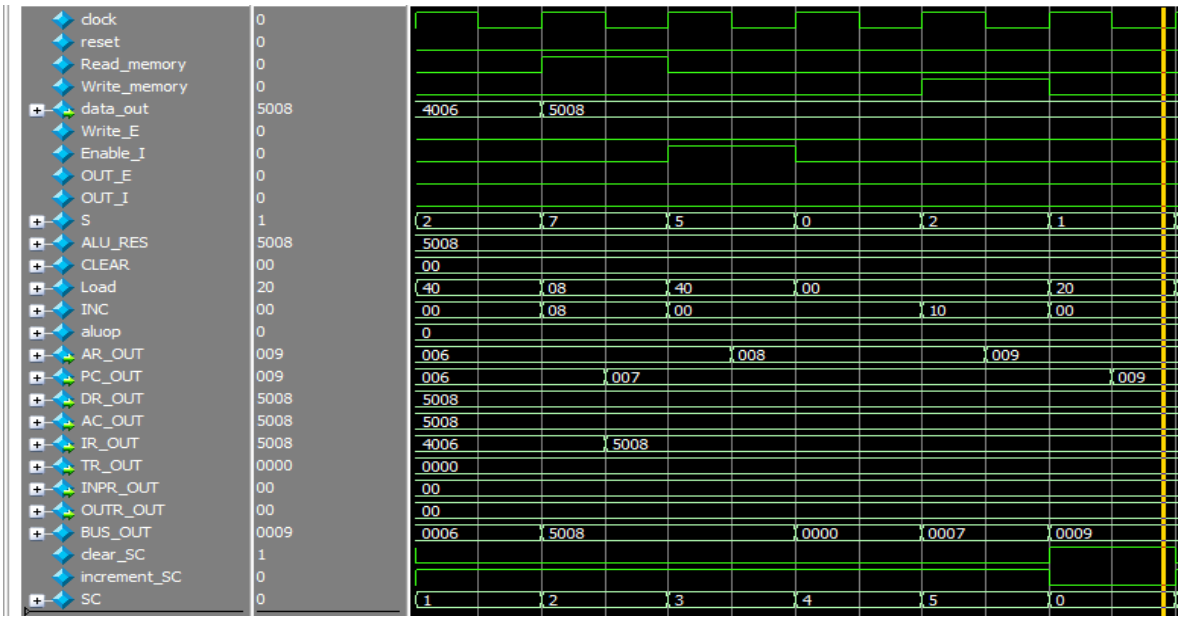
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 185ns	1000000	00000	00000	010	0000	0	0	0	0
# 195ns	0001000	01000	00000	111	0000	1	0	0	0
# 205ns	1000000	00000	00000	101	0000	0	0	0	1
# 215ns	0000000	00000	00000	000	0000	1	0	0	0
# 225ns	0000000	00000	00000	100	0000	0	1	0	0

5) instruction 5: BUN , machine code (4XXX or CXXX) , Example: ADD, machine code(4006)



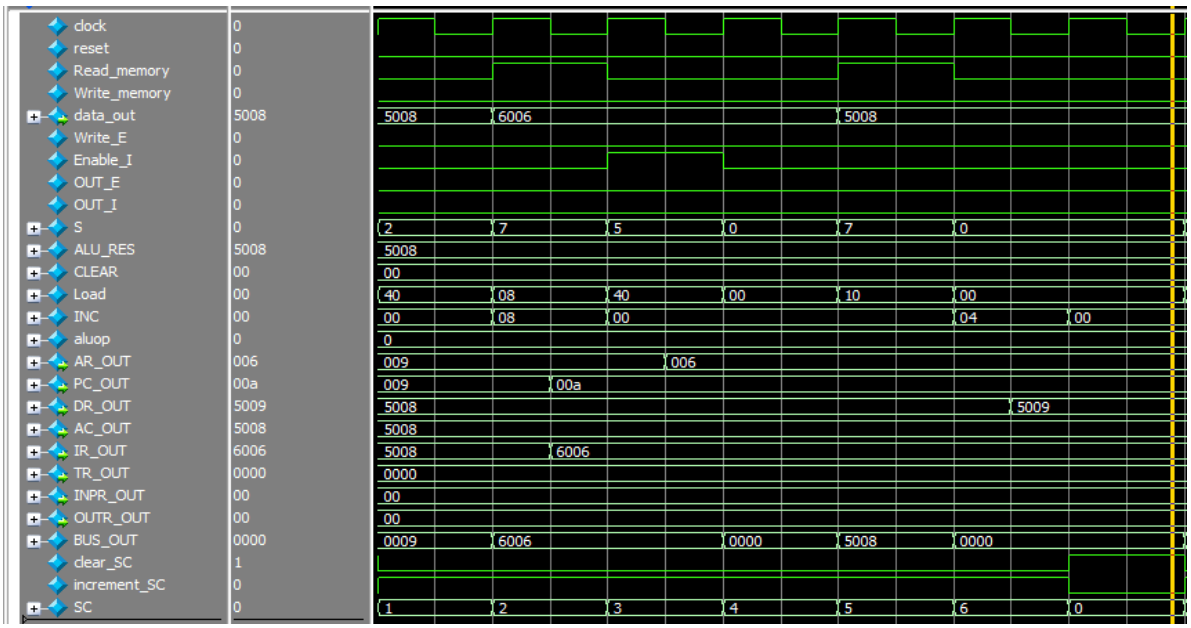
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 235ns	1000000	00000	00000	010	0000	0	0	0	0
# 245ns	0001000	01000	00000	111	0000	1	0	0	0
# 255ns	1000000	00000	00000	101	0000	0	0	0	1
# 265ns	0000000	00000	00000	000	0000	1	0	0	0
# 275ns	0100000	00000	00000	001	0000	0	0	0	0

6) instruction 6: BSA , machine code (5XXX or DXXX) , Example: ADD, machine code(5008)



# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 285ns	1000000	00000	00000	010	0000	0	0	0	0
# 295ns	0001000	01000	00000	111	0000	1	0	0	0
# 305ns	1000000	00000	00000	101	0000	0	0	0	1
# 315ns	0000000	00000	00000	000	0000	1	0	0	0
# 325ns	0000000	10000	00000	010	0000	0	1	0	0
# 335ns	0100000	00000	00000	001	0000	0	0	0	0

7) instruction 7: BSA , machine code (6XXX or EXXX) , Example: ADD, machine code(6006)

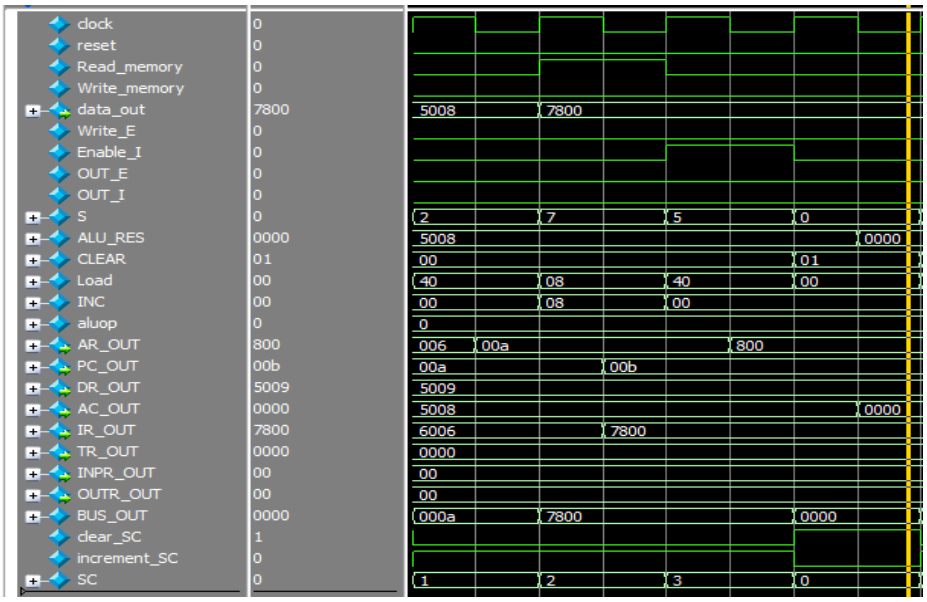


# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 335ns	0100000	00000	00000	001	0000	0	0	0	0
# 345ns	1000000	00000	00000	010	0000	0	0	0	0
# 355ns	0001000	01000	00000	111	0000	1	0	0	0
# 365ns	1000000	00000	00000	101	0000	0	0	0	1
# 375ns	0000000	00000	00000	000	0000	0	0	0	0
# 385ns	0010000	00000	00000	111	0000	1	0	0	0
# 395ns	0000000	00100	00000	000	0000	0	0	0	0
# 405ns	0000000	00000	00000	000	0000	0	0	0	0

2) Register Reference Instruction:

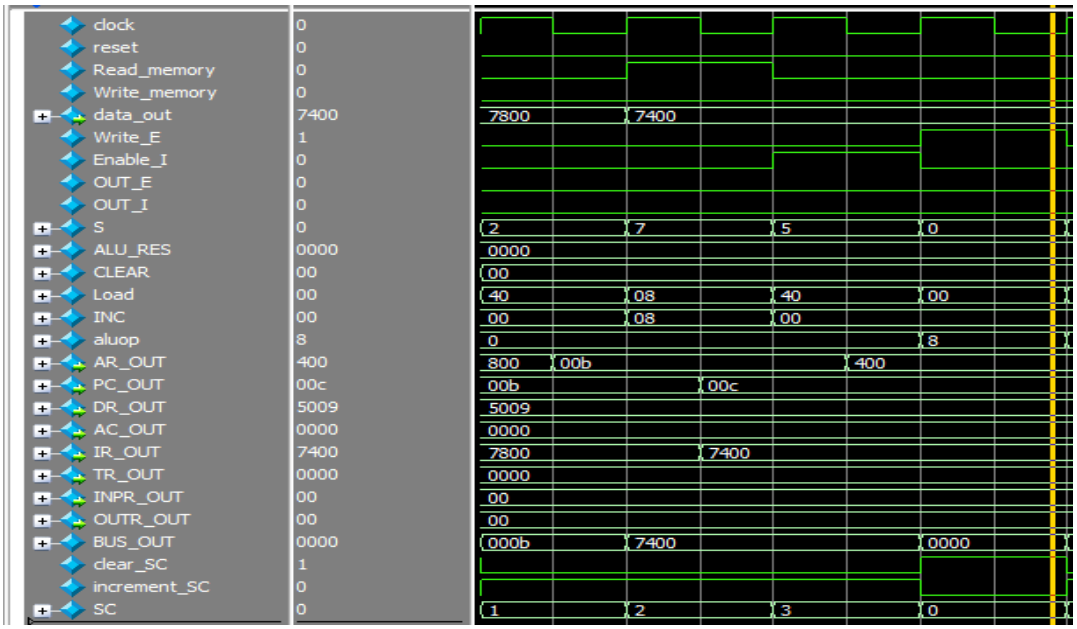
instruction	Machine code.	T0	T1	T2	T3
CLA	7800	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	AC ← 0
CLE	7400	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	E ← 0
CMA	7200	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AC ← AC`
CME	7100	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	E ← E`
CIR	7080	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AC ← Shr AC AC[15]←E, E← AC[0]
CIL	7040	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AC ← Shl AC AC[0]←E, E← AC[15]
INC	7020	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	AC ← AC+1
SPA	7010	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (AC[15] == 0) Then PC ← PC +1
SNA	7008	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (AC[15] == 1) Then PC ← PC +1
SZA	7004	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (AC== 0) Then PC ← PC +1
SZE	7002	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (E == 0) Then PC ← PC +1
HLT	7001	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	HLT

1) instruction: CLA, machine code(7800)



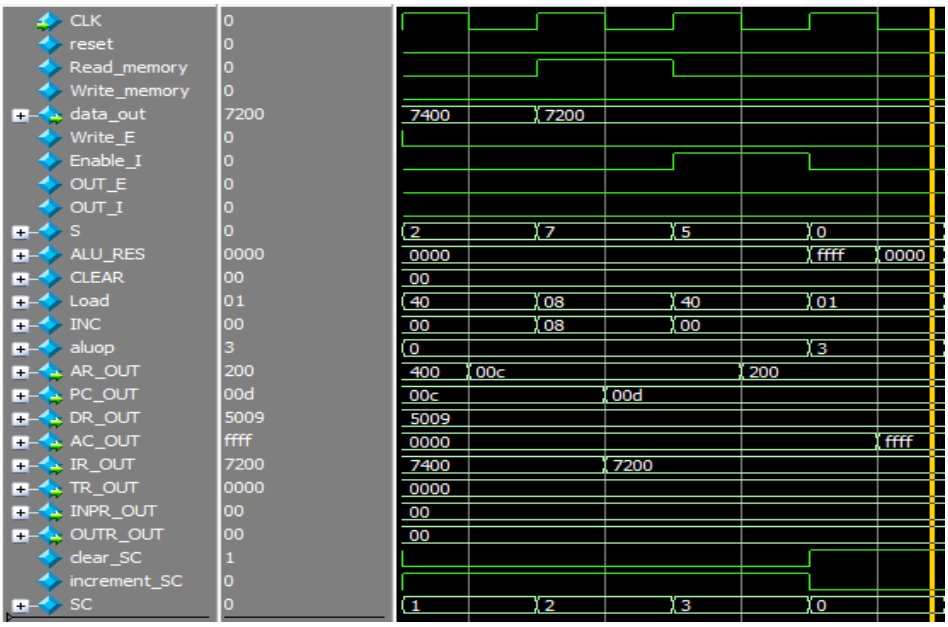
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 415ns	1000000	00000	00000	010	0000	0	0	0	0
# 425ns	0001000	01000	00000	111	0000	1	0	0	0
# 435ns	1000000	00000	00000	101	0000	0	0	0	1
# 445ns	0000000	00000	00001	000	0000	0	0	0	0

2) instruction: CLE, machine code(7400)



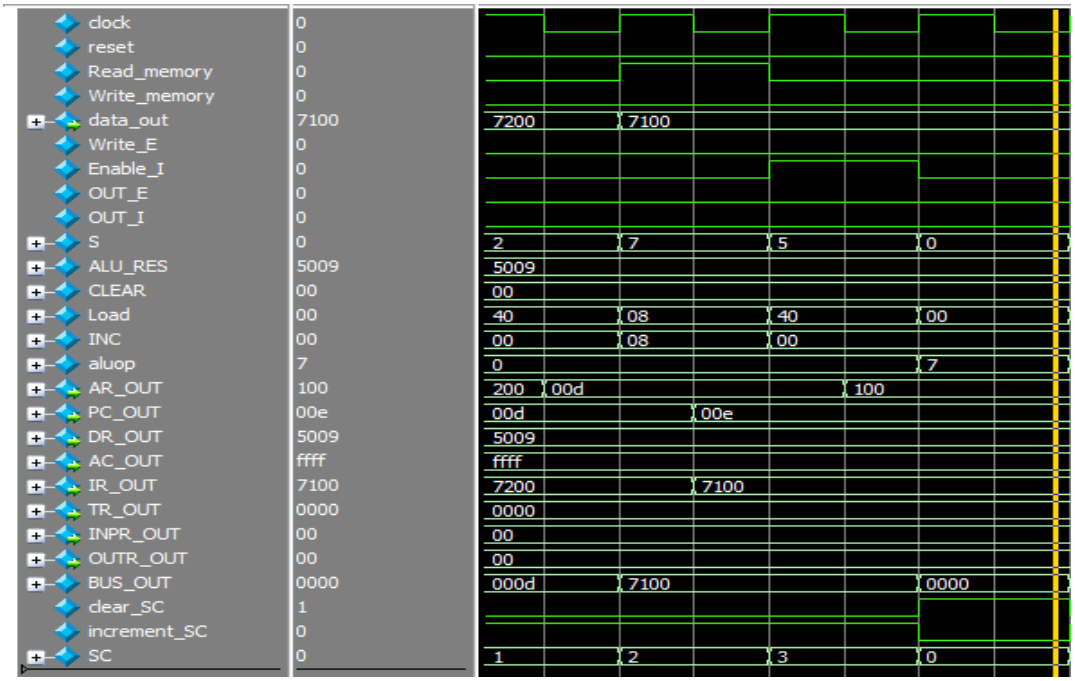
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 455ns	1000000	00000	00000	010	0000	0	0	0	0
# 465ns	0001000	01000	00000	111	0000	1	0	0	0
# 475ns	1000000	00000	00000	101	0000	0	0	0	1
# 485ns	0000000	00000	00000	000	1000	0	0	1	0

3) instruction: CMA, machine code(7200)



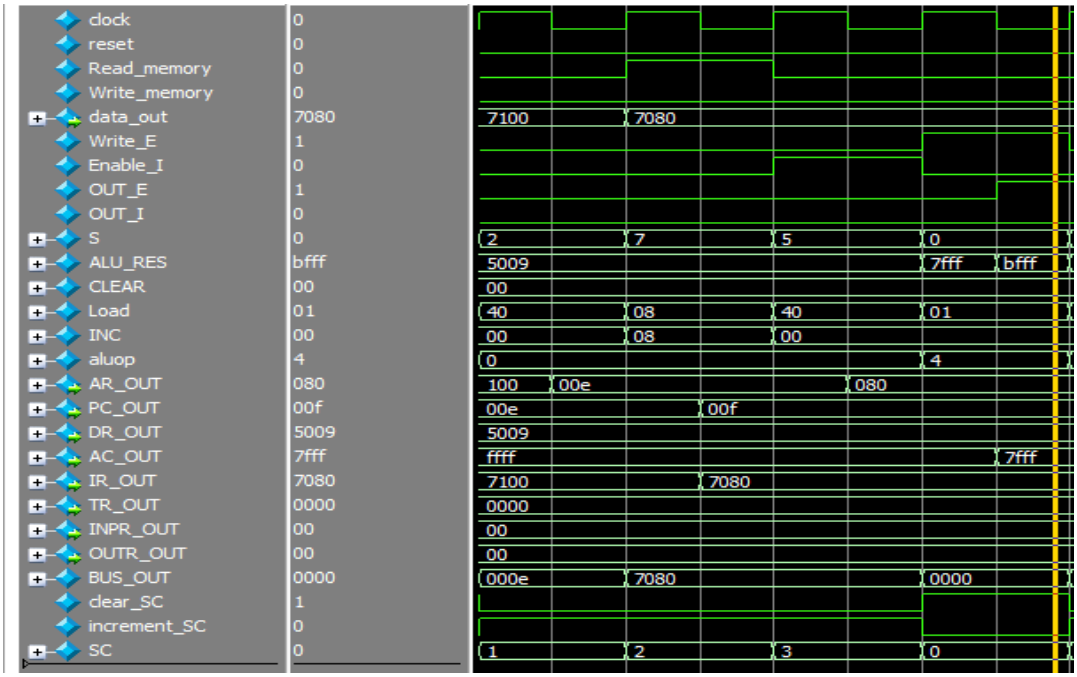
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 495ns	1000000	00000	00000	010	0000	0	0	0	0
# 505ns	0001000	01000	00000	111	0000	1	0	0	0
# 515ns	1000000	00000	00000	101	0000	0	0	0	1
# 525ns	0000001	00000	00000	000	0011	0	0	0	0

4) instruction: CME, machine code(7100)



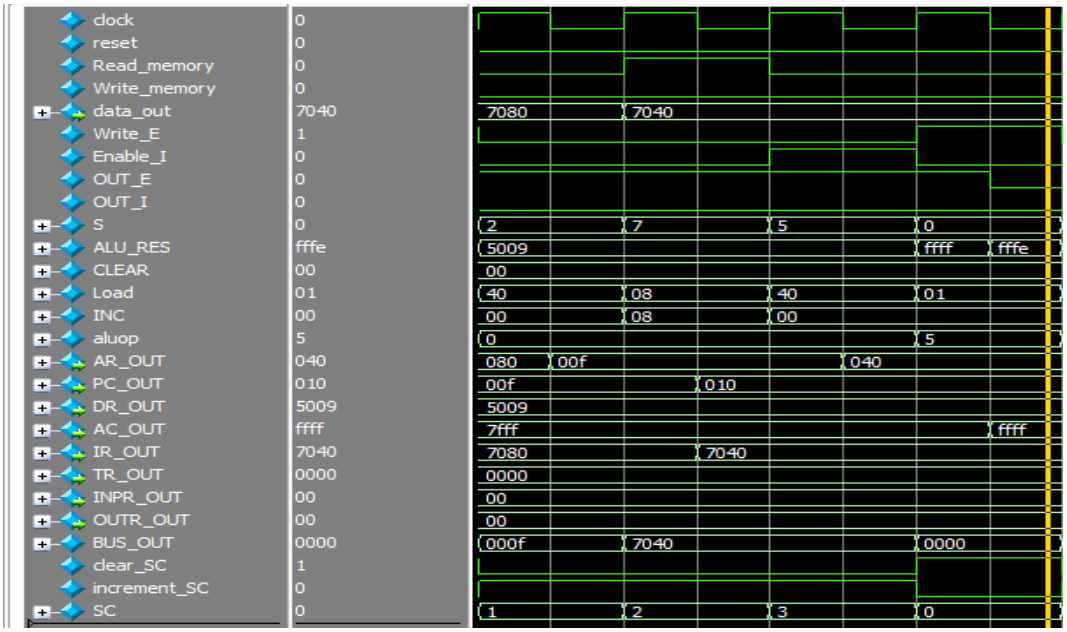
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 535ns	1000000	00000	00000	010	0000	0	0	0	0
# 545ns	0001000	01000	00000	111	0000	1	0	0	0
# 555ns	1000000	00000	00000	101	0000	0	0	0	1
# 565ns	0000000	00000	00000	000	0111	0	0	0	0

5) instruction: CIR, machine code(7080)



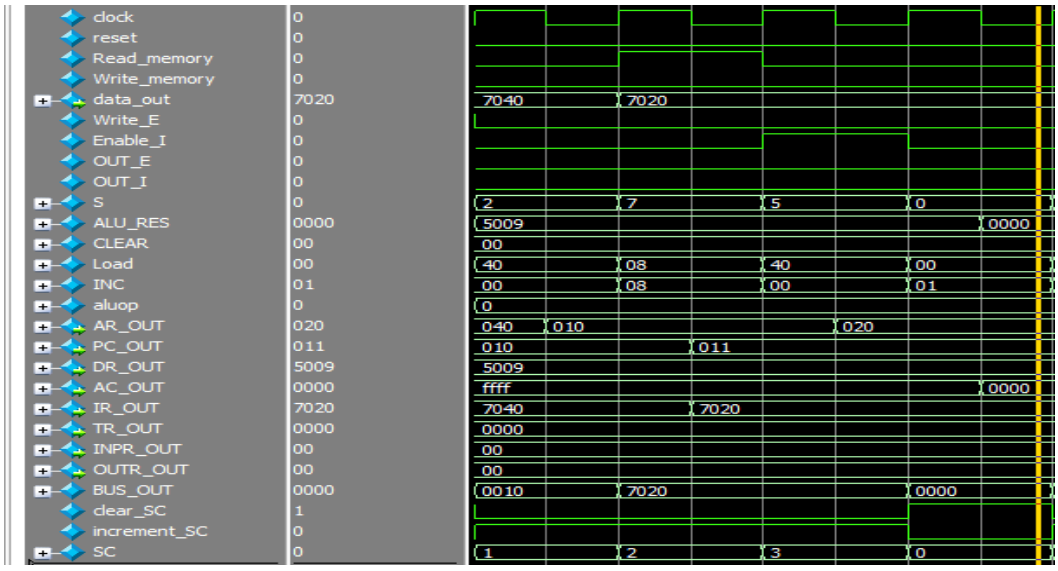
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 575ns	1000000	00000	00000	010	0000	0	0	0	0
# 585ns	0001000	01000	00000	111	0000	1	0	0	0
# 595ns	1000000	00000	00000	101	0000	0	0	0	1
# 605ns	0000001	00000	00000	000	0100	0	0	1	0

6) instruction: CIL, machine code(7040)



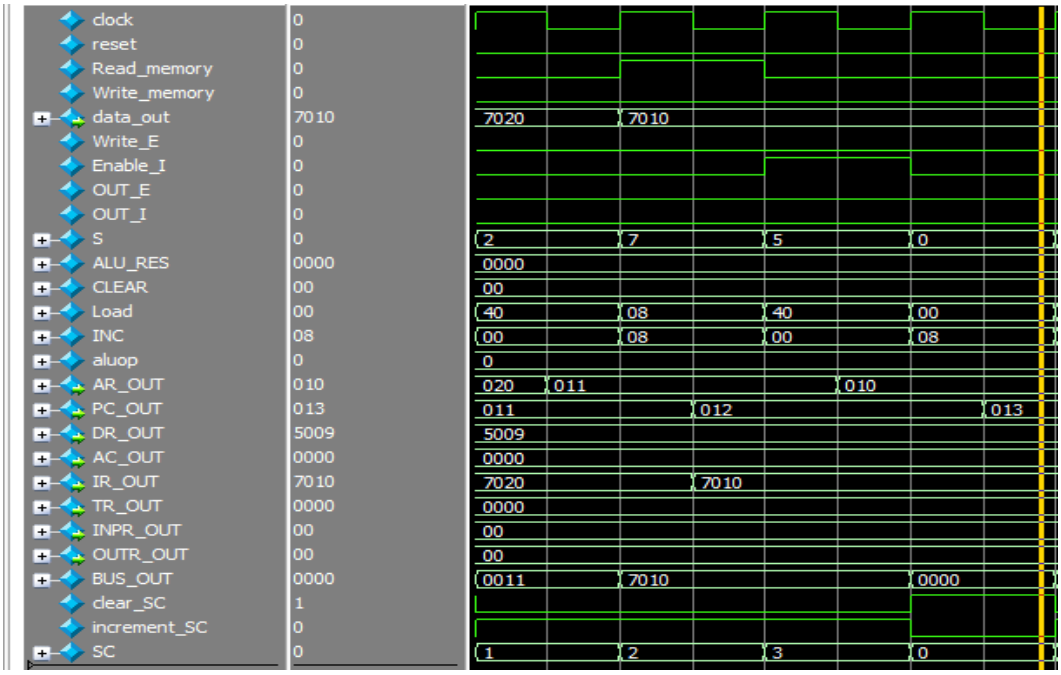
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 615ns	1000000	00000	00000	010	0000	0	0	0	0
# 625ns	0001000	01000	00000	111	0000	1	0	0	0
# 635ns	1000000	00000	00000	101	0000	0	0	0	1
# 645ns	0000001	00000	00000	000	0101	0	0	1	0

7) instruction: INC, machine code(7020)



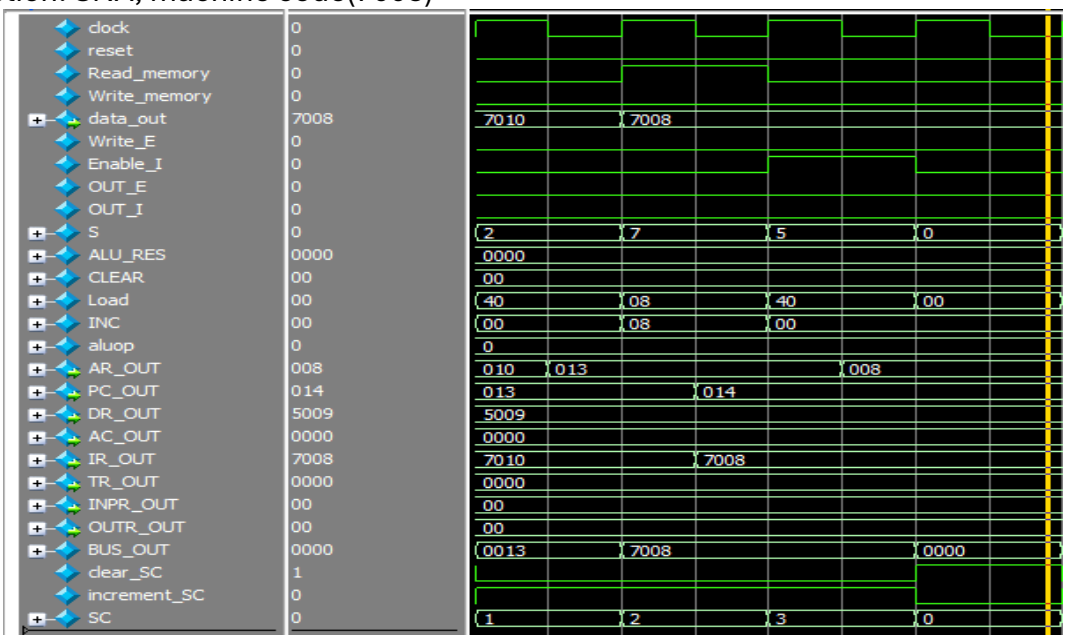
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 655ns	1000000	00000	00000	010	0000	0	0	0	0
# 665ns	0001000	01000	00000	111	0000	1	0	0	0
# 675ns	1000000	00000	00000	101	0000	0	0	0	1
# 685ns	0000000	00001	00000	000	0000	0	0	0	0

8) instruction: SPA, machine code(7010)



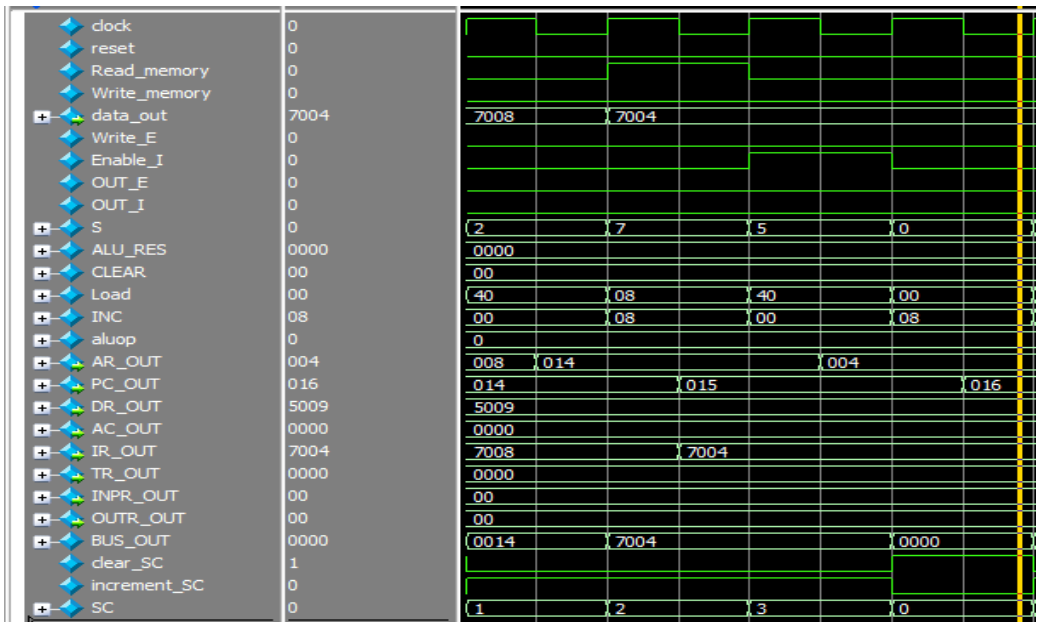
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 695ns	1000000	00000	00000	010	0000	0	0	0	0
# 705ns	0001000	01000	00000	111	0000	1	0	0	0
# 715ns	1000000	00000	00000	101	0000	0	0	0	1
# 725ns	0000000	01000	00000	000	0000	0	0	0	0

9) instruction: SNA, machine code(7008)



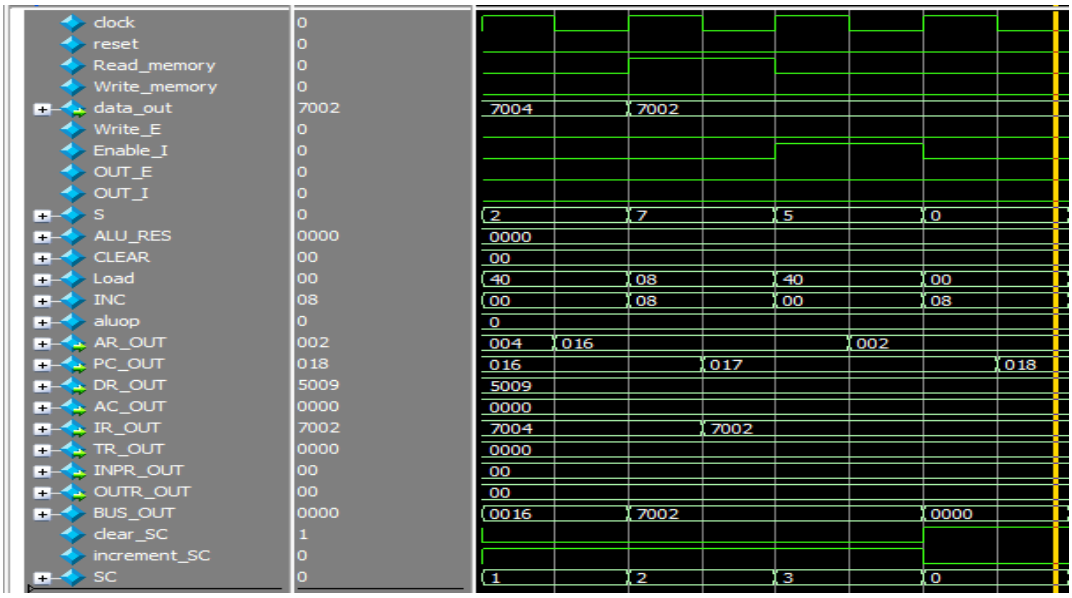
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 735ns	1000000	00000	00000	010	0000	0	0	0	0
# 745ns	0001000	01000	00000	111	0000	1	0	0	0
# 755ns	1000000	00000	00000	101	0000	0	0	0	1
# 765ns	0000000	00000	00000	000	0000	0	0	0	0

10) instruction: SZA, machine code(7004)



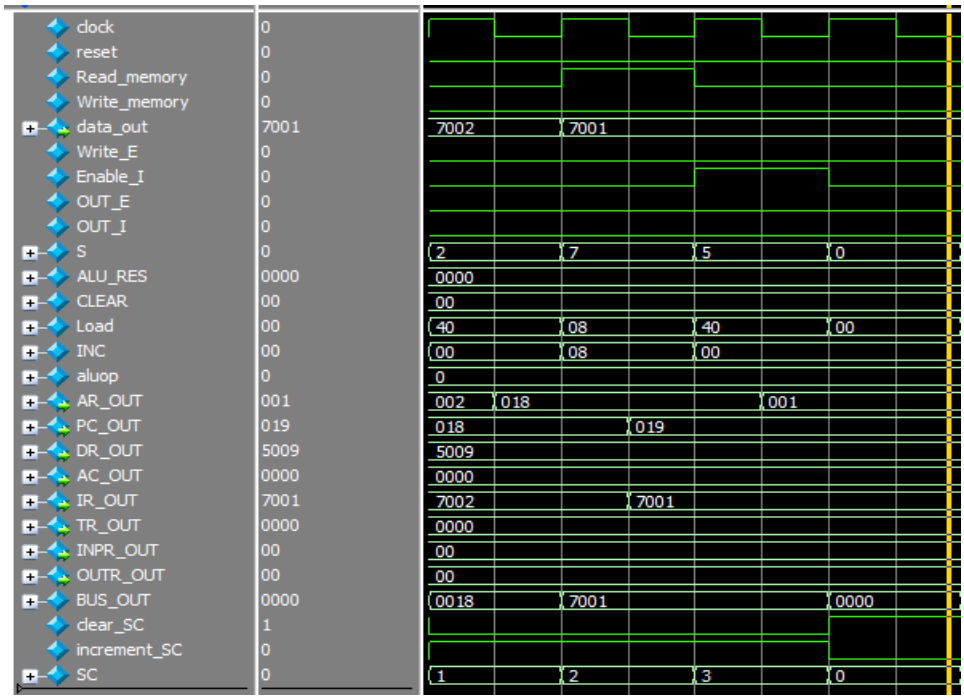
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 775ns	1000000	00000	00000	010	0000	0	0	0	0
# 785ns	0001000	01000	00000	111	0000	1	0	0	0
# 795ns	1000000	00000	00000	101	0000	0	0	0	1
# 805ns	0000000	01000	00000	000	0000	0	0	0	0

11) instruction: SZE, machine code(7002)



# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 815ns	1000000	00000	00000	010	0000	0	0	0	0
# 825ns	0001000	01000	00000	111	0000	1	0	0	0
# 835ns	1000000	00000	00000	101	0000	0	0	0	1
# 845ns	0000000	01000	00000	000	0000	0	0	0	0

12) instruction: HLT, machine code(7001)



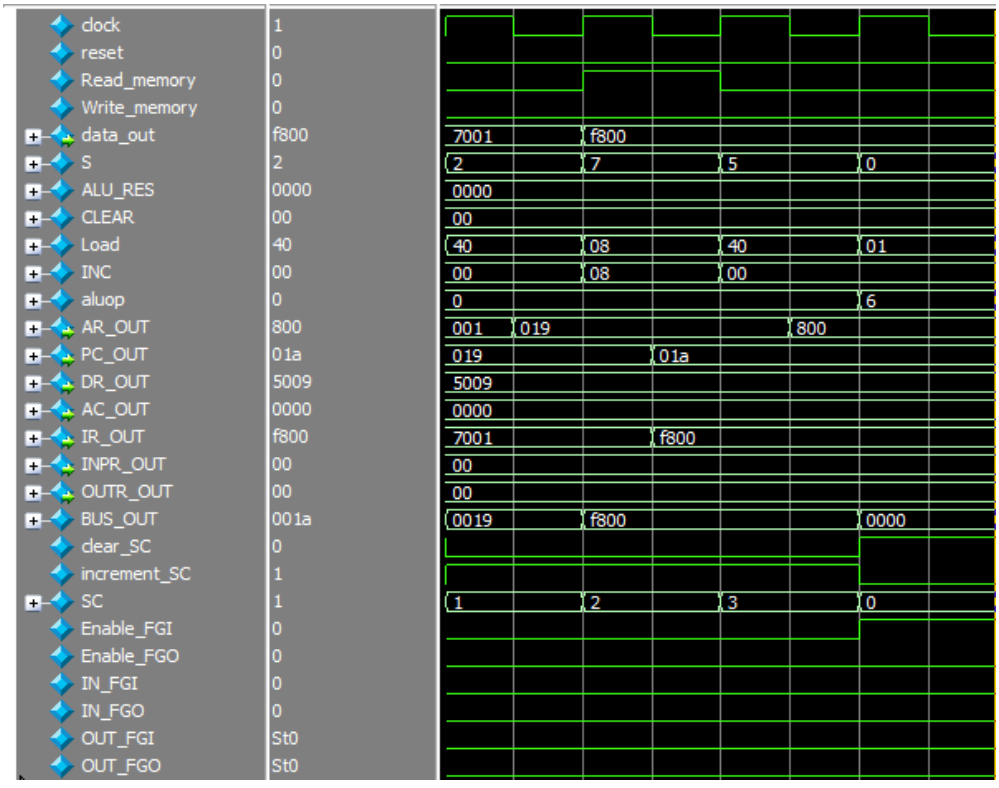
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 855ns	1000000	00000	00000	010	0000	0	0	0	0
# 865ns	0001000	01000	00000	111	0000	1	0	0	0
# 875ns	1000000	00000	00000	101	0000	0	0	0	1
# 885ns	0000000	00000	00000	000	0000	0	0	0	0

3) Input/Output Reference Instructions:

instruction	Machine code.	T0	T1	T2	T3
INP	7800	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	AC(0-7) ← INPR FGI ← 0
OUT	7400	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<-IR(0-11) I<-IR(15)	OUTR ← AC(0-7) FGO ← 0
SKI	7200	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (FGI == 1) Then PC ← PC +1
SKO	7100	AR<-PC	IR<-M[AR] PC<-PC+1	D0-D7<-IR[12-14] AR<- IR(0-11) I<-IR(15)	IF (FGO == 1) Then PC ← PC +1

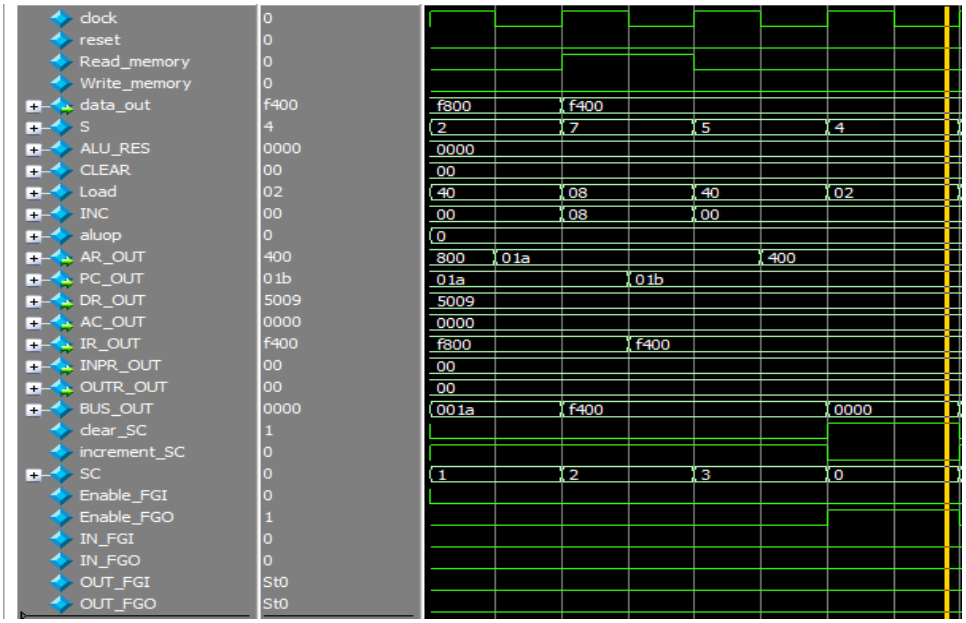
---

1) instruction: INP, machine code(F800)



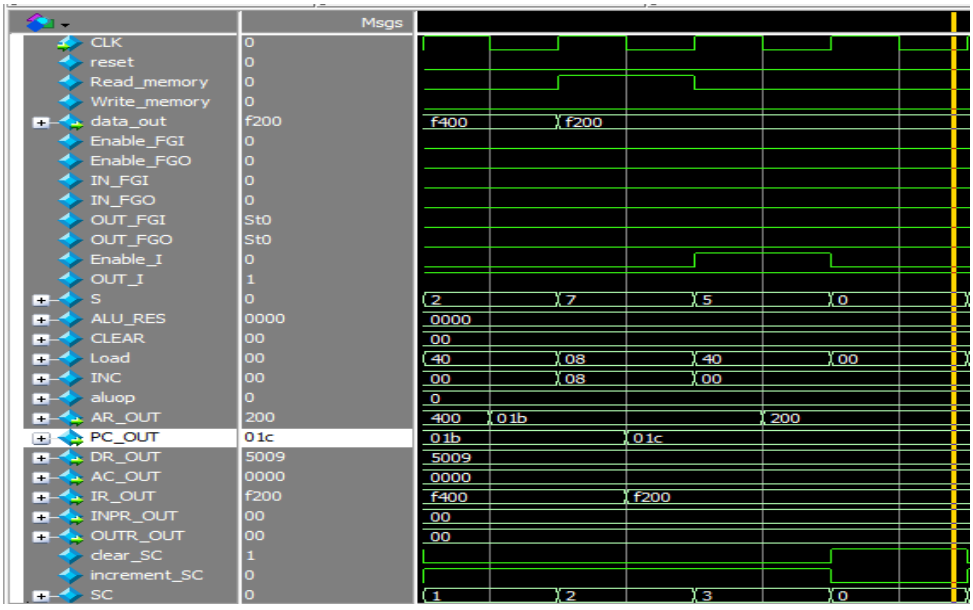
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 895ns	1000000	00000	00000	010	0000	0	0	0	0
# 905ns	0001000	01000	00000	111	0000	1	0	0	0
# 915ns	1000000	00000	00000	101	0000	0	0	0	1
# 925ns	0000001	00000	00000	000	0110	0	0	0	0

2) Instruction: OUT, machine code(F400)



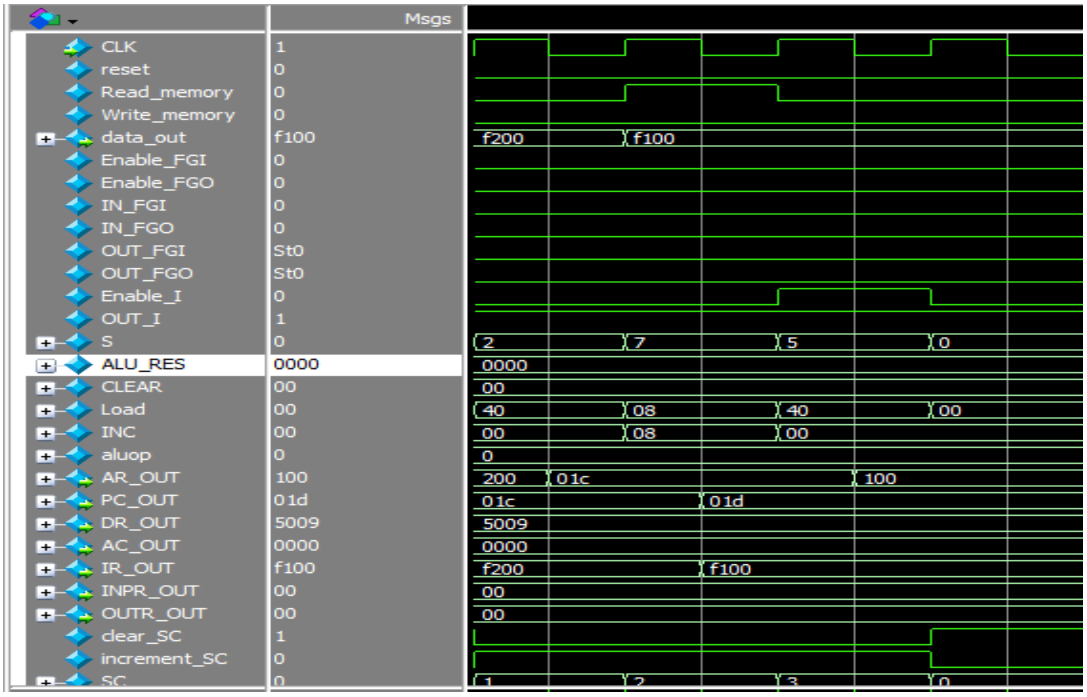
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 935ns	1000000	00000	00000	010	0000	0	0	0	0
# 945ns	0001000	01000	00000	111	0000	1	0	0	0
# 955ns	1000000	00000	00000	101	0000	0	0	0	1
# 965ns	0000010	00000	00000	100	0000	0	0	0	0

3) instruction: SKI, machine code(F200)



# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 975ns	1000000	00000	00000	010	0000	0	0	0	0
# 985ns	0001000	01000	00000	111	0000	1	0	0	0
# 995ns	1000000	00000	00000	101	0000	0	0	0	1
# 1005ns	0000000	00000	00000	000	0000	0	0	0	0

4) instruction: SKO, machine code(F100)



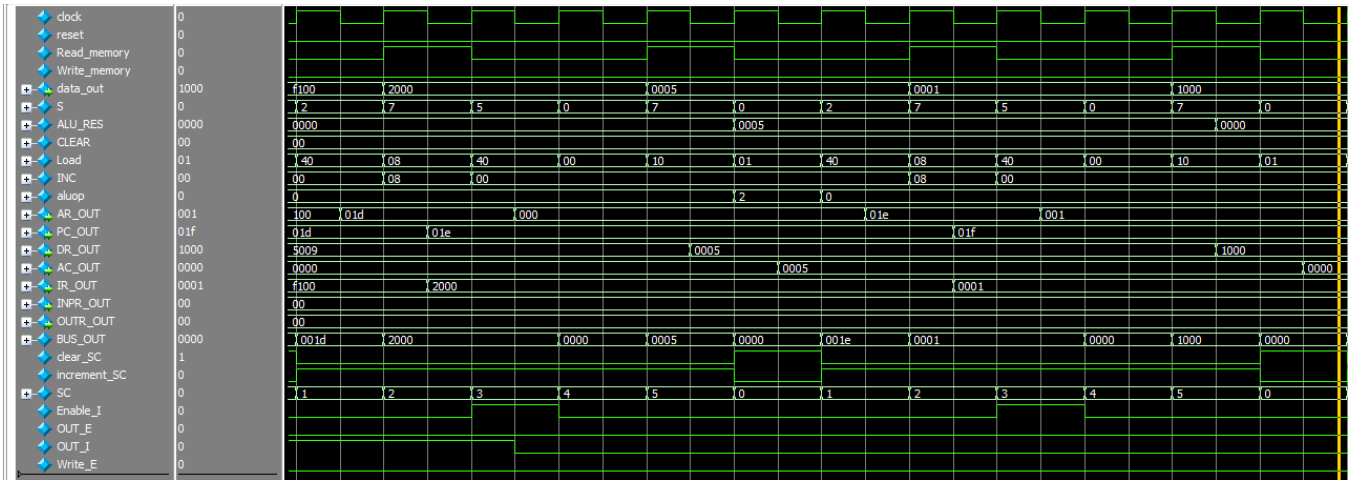
# Time	Load	INC	Clear	S	aluop	Read_memory	Write_memory	Write_E	Enable_I
# 1005ns	0000000	00000	00000	000	0000	0	0	0	0
# 1015ns	1000000	00000	00000	010	0000	0	0	0	0
# 1025ns	0001000	01000	00000	111	0000	1	0	0	0
# 1035ns	1000000	00000	00000	101	0000	0	0	0	1

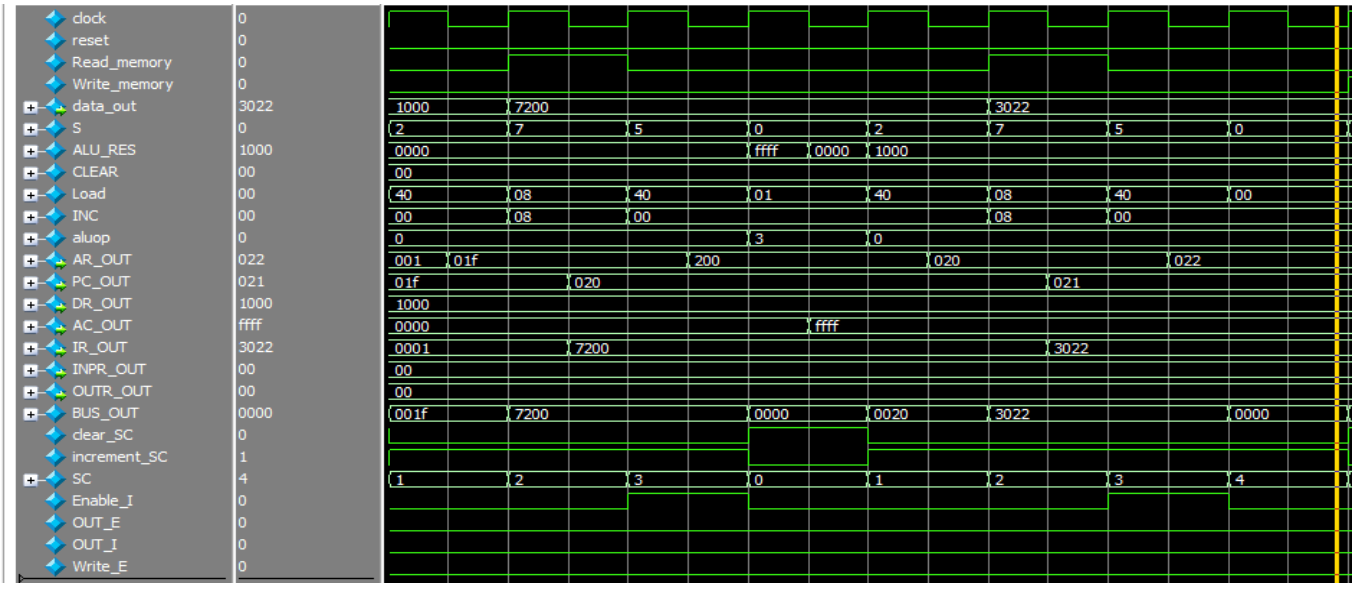
4) Pseudo Instruction:

Pseudo Instructions are simplified assembly language commands that are not directly executed by the hardware but are translated by the assembler into one or more actual machine instructions.

- Example : Y = X NAND Z
  X , Y , Z location in memory

LDA X	**2000**	**AC** ←0005
AND Z	0001	AC ← 5 ^ 1000
CMA	7200	AC ← ~AC AC ← ~(0005)
STA Y	3022	MEM[34] ← FFFF





- Example : Y = X – Z , (X,Y , Z → location in memory)

LDA Z	**2000**	**AC** ←0005
CMA	7200	AC ← ~AC AC ← ~(0005)
ADD X	1001	AC ← ~AC AC ← ~(0005)
STA Y	3101	MEM[34] ← FFFF