Project Report

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Course Number: EEE 3104

Course Title: Microprocessor and Interfacing Laboratory

Prepared For:
Dr. Sazzad Muhammad Samaun Imran
Associate Professor
Department of Electrical and Electronic Engineering
University of Dhaka

Prepared By:
Rifah Ahmed Maisha
Student ID: SN-073-057
Level-Term: 3-II
Date of Submission: 01/04/2022

DESIGN OF A 4-BIT COMPUTER IN VerilogHDL WITH A FIXED INSTRUCTION SET

Introduction

The goal of this project is to design a 4-bit computer with a fixed instruction set in Verilog HDL. The instruction set that I was assigned is as follows:

No. of Instruction	Instruction	Opcodes
1	ADD A,B	0001
2	SUB A,B	0010
3	XCGH B,A	0011
4	RCL B	0100
5	SHR A	0101
6	MOV [ADDRESS],A	0110
7	MOV A, [ADDRESS] (Initially XOR A,[Address] but was changed later)	0111
8	AND A,B	1000
9	OR B, [ADDRESS]	1001
10	OUT A	1010
11	JZ ADDRESS	1011
12	PUSH B	1100
13	POP B	1101
14	CALL ADDRESS	1110
15	RET	1111
16	HLT	0000

The rest of the report is outlined as follows:

- In PART-1, I will explain the given instruction set and what each instruction is supposed to do
- In PART-2, I will discuss about my computer's architecture in great detail and it's unique features
- In PART-3, I will present the simulated waveforms and verify the functionality of my computer
- Finally, I will discuss about my Assembler that I have written for my PC

Points to Note:

- For this 4-bit computer, all ALU operations performed are done using 4-bit registers
- To demonstrate the correct execution of each instruction, a testbench is written for all of them and they are attached in the .zip submission file
- All simulations have been performed using EDA Playground Tool
- The code of the assembler is written at the end and it is also attached with the .zip file
- All the waveforms can also be found inside the 'Results' folder inside the .zip file

PART-1: Explanation of the Instruction Set

The instruction set contains 16 instructions in total. Each of them is explained below with a short description:

- ADD A,B:

Add two 4-bit numbers A and B. Store the result in A. A and B are unsigned integers initially.

If A=4’b0001 and B=4’b0100, then after ‘ADD A, B’, A=4’b0101. If A+B exceeds decimal 15, then final 4 bits are stored into A and the additional carry bit is stored elsewhere.

- SUB A,B:

Subtract a 4-bit number B from A. Store the result in A. A and B are unsigned integers initially. There is no restriction on which one is larger.

If A=4’b0101 and B=4’b0100, then after ‘SUB A, B’, A=4’b0001. If A-B is a negative number, then final 4 bits are stored into A and the additional borrow bit is stored elsewhere.

- XCHG A,B:

Exchange two 4-bit numbers A and B. If, initially, A = 4’b0001 and B=4’b0100, then after ‘XCHG A, B’, A=4’b0100 and B=4’b0001.

- RCL B:

Rotate the bits of B counter clockwise through the carry. The MSB of B replaces the original carry and the LSB of B is replaced by the original carry.

If initially, B = 4’b0111 and carry = 1’b1, then after RCL B, B = 4’b1111 and carry = 1’b0.

- SHR A:

Shift the bits of A logically right by one bit through the carry. The LSB of A replaces the original carry and the MSB of A becomes 0.

If initially, A = 4’b1001 and carry=1’b0, then after ‘SHR A’, A=4’b0100 and carry = 1’b1.

- MOV [Address], A:

Move the current content in A to the location in RAM indexed by [Address]. ‘MOV 9h, A’ refers to storing the content in A to RAM[9], or the 10th index location of RAM.

- MOV A, [Address]:

Move the current content indexed by [Address] in RAM to A. ‘MOV A, 9h’ refers to loading the content in RAM[9], or the 10th index location of RAM to A.

- AND A,B:

Perform bitwise AND operation between two 4-bit numbers A and B. Store the result in A. A and B are unsigned integers initially.

If A=4’b0001 and B=4’b0100, then after ‘AND A, B’, A=4’b0001.

- OR B, [Address]:

Perform bitwise OR operation between B and the current content indexed by [Address] in RAM. ‘OR B, 9h’ refers to performing a bitwise OR between B and RAM[9], or the content in the 10th index location of RAM.

If RAM[9]=4’b0001 and B=4’b0100, then after ‘OR B, 9h’, B=4’b0101.

- OUT A:

Send the content in accumulator, A to the output register to send to the output port. If A = 4’b0101, then after ‘OUT A’, the content in the output register is 4’b0101.

- JZ Address:

Jump to the location denoted by ‘Address’ if the current zero flag in the Flag Register is set.

- PUSH B:

Push the content in B to the location pointed to in the stack by the stack pointer. First, the content in B is sent to the stack and then stack pointer is decremented by 1.

- POP B:

Pop the content from the location pointed to in the stack by the stack pointer into B. First, the stack pointer is incremented by 1 and then the content at the location pointed to by the pointer is sent to B.

- CALL Address:

Call a subroutine that starts at the location denoted by ‘Address’ in RAM. The current value of program counter is sent to stack and then the program counter is set the Call address.

- RET:

Returns the return address previously stored in the stack by ‘CALL Address’. The program counter value is restored to that original value.

Demo Code:

```
Mov a, 9h ----------- at location 0
Call 3h --------at location 1
Shr a ----------at location 2
Xchg a,b -------- at location 3
Mov a,10h --------at location 4
Add a,b --------at location 5
Ret --------at location 6
```

In the program snippet written above, when the ‘Call 3h’ is executed, the address of the next instruction, 2 is stored in the stack as the return address. And, the program counter jumps to 3, to the beginning address of the subroutine. When Ret is executed, the program counter is loaded with 2 again, the location stored in the stack.

- HLT:

Stops the program counter from incrementing. So, the same HLT instruction is fetched forever.

PART-2: ARCHITECTURE

I have tried to closely follow the SAP-1(Simple as Possible) architecture as my design choice with some modifications. In Fig.1, I have drawn the architecture showing the connections between the different modules used in the construction of my 4-bit Computer. For better understanding of the connections, I have deliberately omitted the control signals from the figure.

From the figure, it is seen that there are 11 different modules in the computer. The SAP diagrams that we are familiar with usually have a common ‘Write Bus’ for establishing communications between the different modules. **But instead of using a common bus for communicating between the different modules, I have used separate buses for all of them**. **This is the unique feature of my computer**.

The reason to do so is twofold. For starters, I thought it is simpler to write Verilog code for such an architecture since I can be absolutely certain that on a clock edge, there will be no unwanted overwrites or loss of information in the Bus. Secondly, even though this design is not practical for all purposes, in my opinion, it makes things easier to understand to the naked eye. This also makes data communication faster because the bus doesn’t become a bottleneck at any given time. It also makes my design unique because I have not attempted using simple ‘if-else’ structure based on opcode or the classic SAP structure.

Because of using a separate bus for all communication nodes, the amount of control signals used in this architecture is also very high. For example, in the usual SAP-1 architecture, there is only 1 control signal ‘Ce’ that is used to signal that RAM is being accessed and on next clock edge the addressed data will be put on the ‘Common Write Bus’. But, in my architecture, since I am using separate buses to send data from RAM to IR and RAM to A, I am having to use two separate control signals ‘Cei’ and ‘Cea’, respectively indicating IR and A as the destination of the content at the addressed RAM location.

Fig.1 can be taken as a stepping stone for what is to follow. I will gradually discuss the functionality of each module and the purpose of the control signals sent to each of them.

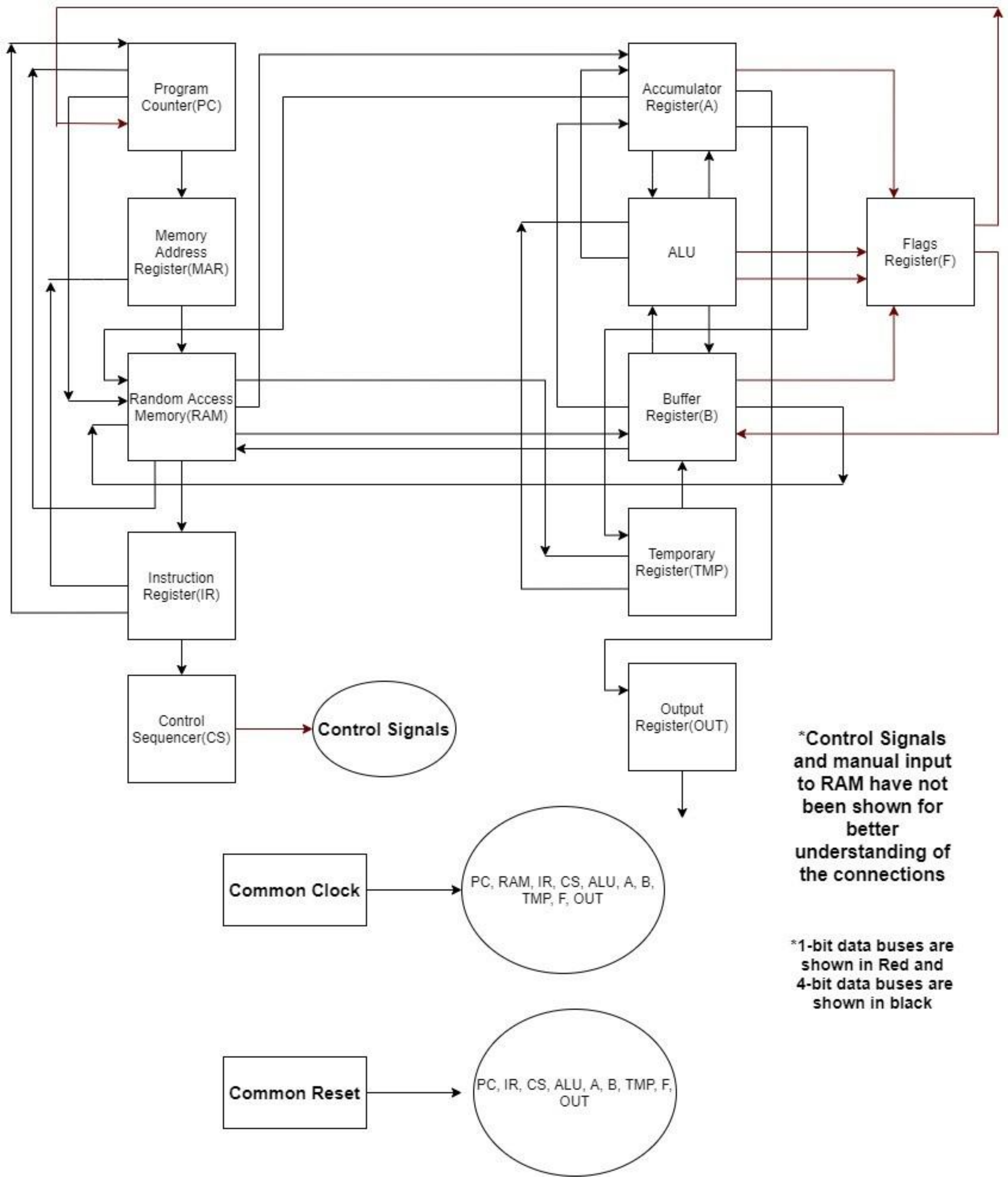

Program Counter(PC)
Memory Address Register(MAR)
Random Access Memory(RAM)
Instruction Register(IR)
Control Sequencer(CS)
Control Signals
Accumulator Register(A)
ALU
Flags Register(F)
Buffer Register(B)
Temporary Register(TMP)
Output Register(OUT)
*Control Signals and manual input to RAM have not been shown for better understanding of the connections
*1-bit data buses are shown in Red and 4-bit data buses are shown in black
Common Clock
PC, RAM, IR, CS, ALU, A, B, TMP, F, OUT
Common Reset
PC, IR, CS, ALU, A, B, TMP, F, OUT

Fig 1. Full Architecture of the 4-bit Computer

Construction of Different Modules and their Functionalities:

1. Program Counter (PC):

Functionality:

The program counter holds the current 4-bit address of the instruction to be executed. During each machine cycle, the program counter sends this address to the MAR and then incremented by 1. If a jump condition is satisfied or a Call instruction is executed, the program counter is set to that address location.

The connections of the Program Counter module is shown in Fig.2.

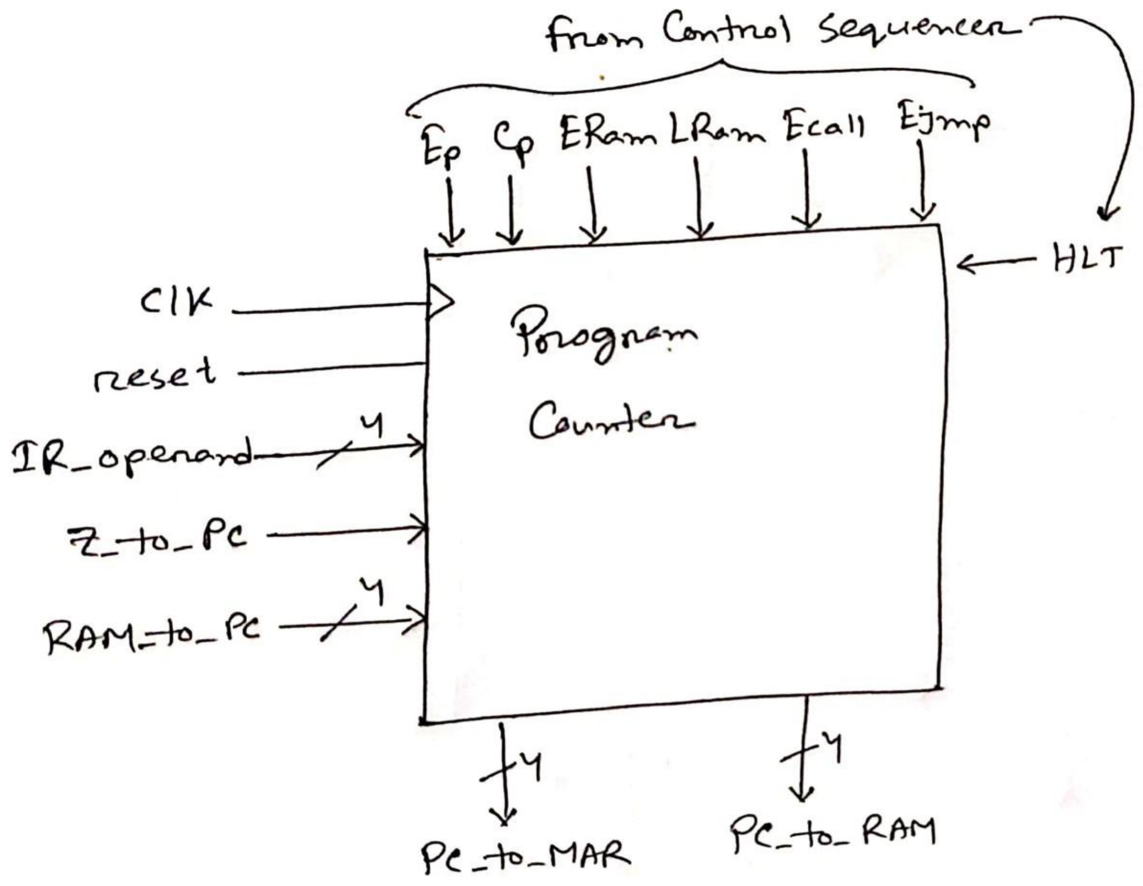


Fig 2. Program Counter

Inputs:

- Clk and Reset: The common clock and common reset. At the negative edge of reset, the Program Counter is reset to zero.
- E_p: When E_p is set, the current value of the program counter is latched onto output PC_to_MAR and sent to MAR on the next positive clock edge.
- C_p: When C_p is set, the current value of the program counter is incremented by 1 on next positive clock edge. If the current value is 4'b1111, then it is set to 4'b0000.
- E_{jmp}: When E_{jmp} is set, it means that current instruction is a 'jump if zero' instruction.
- Z_to_PC: The 'Zero Flag' from the Flag Register(F) is fed to the PC. When this is set, and E_{jmp} is also set(indicating a jump instruction), the current value of the program counter is replaced by the last 4 bits of the Instruction Register (IR_operand) on next positive clock edge.
- IR_operand: The last 4 bits of the Instruction Register is fed to the PC. When there is a jump instruction, it contains the jump address.
- E_{Ram}: For a 'Call Address' instruction, when E_{Ram} is set, the current value of the program counter is sent to the stack on next positive clock edge.
- E_{call}: For a 'Call Address' instruction ,when E_{call} is set, the current value of the program counter is replaced by the last 4 bits of the Instruction Register(IR_operand) which holds the 'Call Address' on next positive clock edge.
- L_{Ram}: For a 'RET' instruction, when L_{Ram} is set, the return address is loaded from the stack to the program counter on next positive clock edge.
- RAM_to_PC: 4-bit data bus that holds the return address that is loaded from the stack.
- HLT: Indicating current instruction is a HLT instruction. When this is set, the program counter first decrements by 1 on next positive clock edge. This happens so because by the time it is realized that the current instruction is a HLT, the value of PC has already incremented by 1. So, the decrementing operation is done to make PC point again to HLT. After that, C_p is permanently set to zero inside Control Sequencer which disables the PC from incrementing and therefore keeps fetching and executing the same HLT instruction forever.

Output:

- PC_to_MAR: 4-bit address that indicates the location of the current instruction in the RAM. Equals to high impedance when E_p is not set.

- PC_to_RAM: 4-bit address that is stored into the stack inside the RAM during a 'Call Address' instruction.

2. Memory Address Register (MAR):

Functionality:

The Memory Address Register is a 4-bit register that stores the address location of from the Program Counter during each machine cycle. This is typical to a MAR we see in SAP-1.

The connections of the Memory Address Register module is shown in Fig. 3.

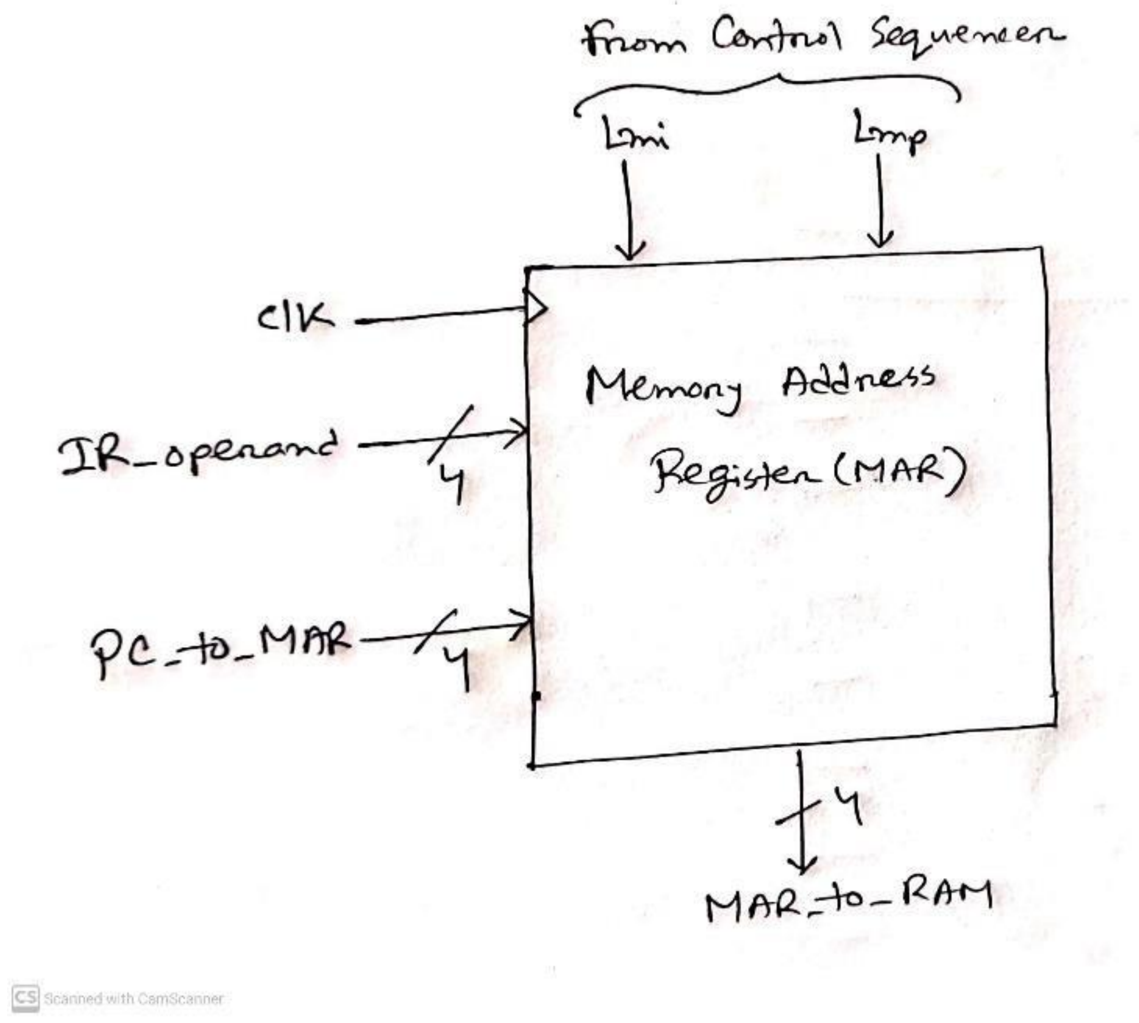


Fig 3. Memory Address Register

<u>Inputs</u>:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the Memory Address Register is reset to zero.
- L_{mp}: When L_{mp} is set, the input address location coming from Program Counter (PC_to_MAR) is latched onto MAR and put on the bus to RAM. This operation is asynchronous, meaning regardless of the clock, when this signal is set, the output MAR_to_RAM is latched to PC_to_MAR.
- PC_to_MAR: Output of Program Counter. Same as before.
- L_{mi}: When L_{mi} is set, the address field/last 4 bits of the Instruction Register (IR_operand) gets loaded into the MAR. An example of this is when there is a Load instruction. So, the last 4 bits of IR indicates where to look for the data in the RAM that is to be loaded into A. This is also asynchronous.
- IR_operand: Last 4-bits of instruction register indicating address field. Same as before.

Output:

- MAR_to_RAM: The 4-bit address location used to address into RAM. This can come from PC or IR depending on the state of the current instruction.

3. Random Access Memory (RAM):

Functionality:

The Random Access Memory (RAM) is used to store both data and instruction in the same 2D structure. I have used a 16*8 RAM, meaning I have in total 16 address locations and each of them is size 1 byte.

Among these 16 locations, the first 9 are used for storing instructions and the next 5 are for storing data. The last 2 locations denote the stack. This can be visualized by Fig.4.

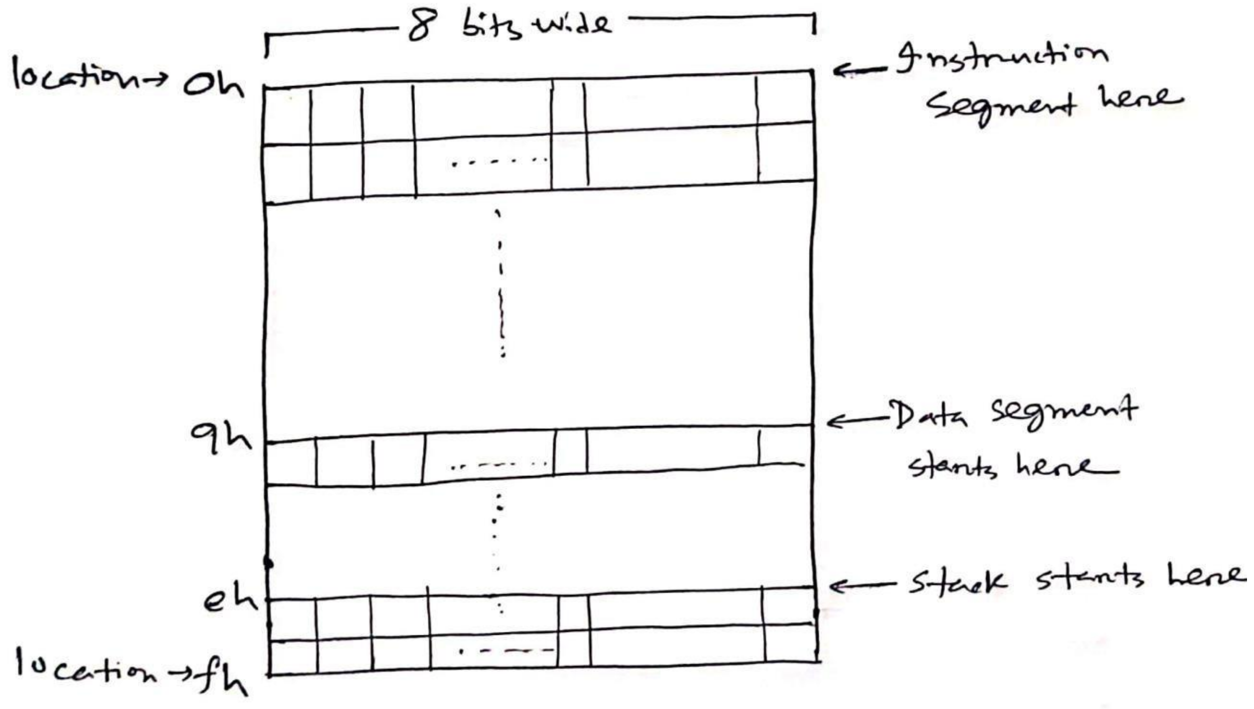


Fig 4. Inside the RAM

The connections of the Random Access Memory module is shown in Fig.5.

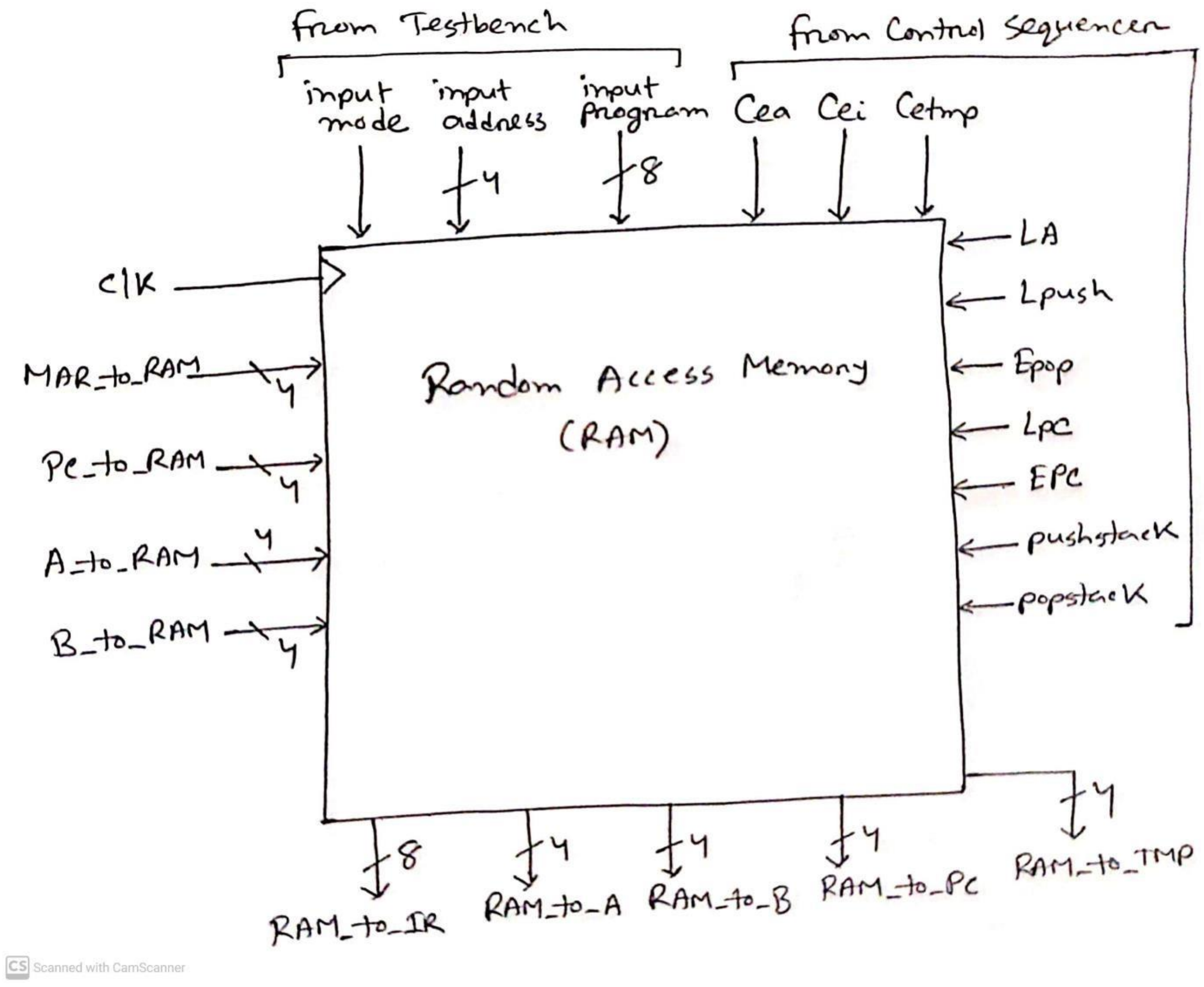


Fig 5. Random Access Memory

Inputs:

- Clk: Common Clock
- C_{ei}: When C_{ei} is set, on the next positive clock edge, the content at the addressed location is put on the bus RAM_to_IR. This is for sending a typical 8-bit instruction to the Instruction Register.

- C_{ea}: When C_{ea} is set, on the next positive clock edge, the content at the addressed location is put on the bus RAM_to_A. This is for ‘Mov A, [Address]’ instruction, where the data at the ‘Address’ is put on the bus connecting RAM to A.
- MAR_to_RAM: Denotes the current location to be addressed. Same as before.
- C_{etmp}: When C_{etmp} is set, on the next positive clock edge, the content at the addressed location is put on the bus RAM_to_TMP to send to the Temporary Register (TMP). This is used for temporary saving data to register for ‘OR B, [Address]’ instruction.
- L_{pc}: When L_{pc} is set, the address coming from PC_to_RAM is stored in the stack. This is an asynchronous operation.
- PC_to_RAM: 4-bit address that is stored into the stack inside the RAM during a ‘Call Address’ instruction. Same as before.
- E_{pc}: When E_{pc} is set, on the next positive clock edge, the return address stored in the stack is put on the bus to PC (RAM_to_PC).
- L_{push}: For ‘Push B’ instruction, when L_{push} is set and this indicates storing the current value of B into the stack. This is an asynchronous operation.
- B_to_RAM: This is the current value of B that is loaded into the stack inside the RAM when there is a ‘PUSH B’ instruction.
- E_{pop}: For ‘Pop B’ instruction, when E_{pop} is set, the content at the address location pointed to by the stack pointer inside the RAM is put on the bus to B, denoted by RAM_to_B.
- pushstack: When pushstack is set, on the next positive clock edge, the stack pointer is decremented by 1.
- popstack: When popstack is set, on the next positive clock edge, the stack pointer is incremented by 1.
- L_A: For ‘Mov [Address], A’ instruction, when L_A is set, the value coming from Accumulator (A_to_RAM) is stored into the addressed location.
- A_to_RAM: 4-bit data coming fed to the RAM from the Accumulator.

Output:

- RAM_to_IR: 8-bit bus that carries the instruction from RAM to IR. When unused, this is set to high impedance.
- RAM_to_A: 4-bit data bus connecting RAM to A used for ‘Mov A, [Address]’ instruction. When unused, this is set to high impedance.
- RAM_to_TMP: 4-bit data bus connecting RAM to TMP used for ‘OR B, [Address]’ instruction. When unused, this is set to high impedance.

- RAM_to_PC: This is the return address that is loaded from the stack. Same as before.
- RAM_to_B: 4-bit data bus connecting RAM to B used for 'POP B' instruction.

4. Instruction Register (IR):

<u>Functionality</u>:

The Instruction Register (IR) stores the current instruction to be executed, splits it into two equal parts and sends the most significant 4-bits, or the opcode, to the control sequencer (CS). The least significant 4 bits, or the operand, are sent to the PC (for 'JZ Address') or MAR (for 'MOV') depending on the type of instruction.

The connection of the Instruction Register module is shown in Fig.6.

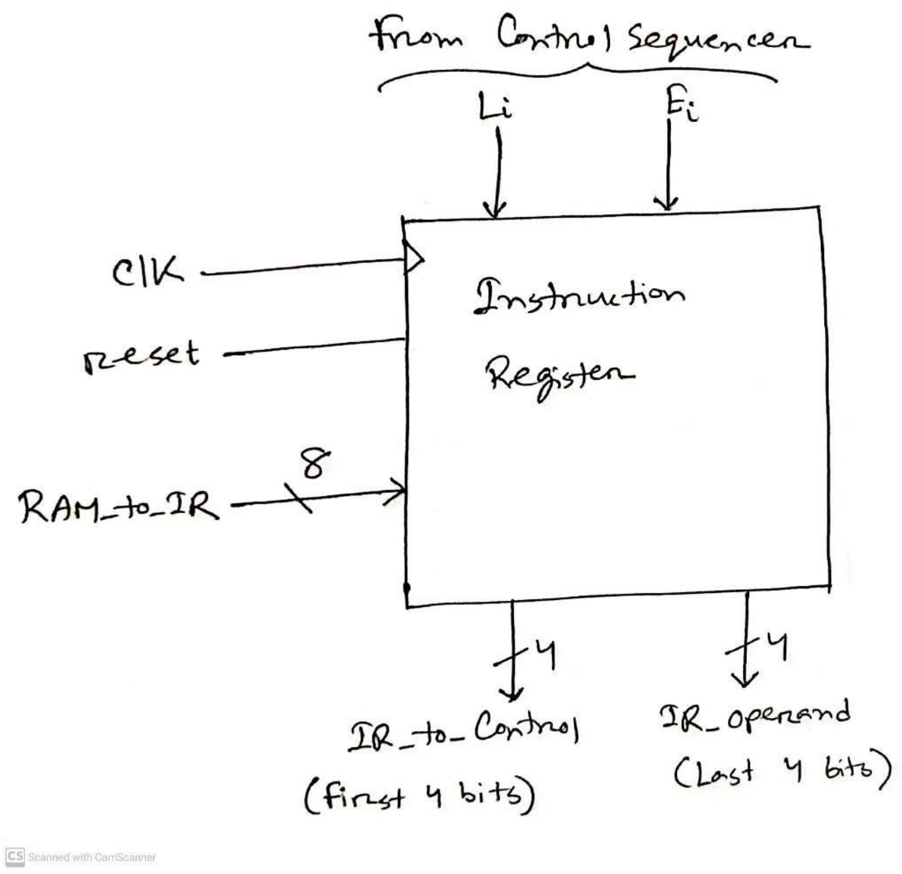


Fig 6. Instruction Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the Instruction Register is reset to high impedance so it does not produce any invalid control signals by passing 4’b0000 to Control Sequencer.
- L_i: For each instruction, when L_i is set, the instruction is loaded into the instruction register. This is an asynchronous operation
- E_i: For any instruction that needs memory referencing, e.g. MOV, JZ Address, Call Address, this signal is set. And, then the last 4-bits of the current instruction in IR is put on the bus to MAR/PC depending on the type of instruction.
- RAM_to_IR: 8-bit instruction fed to IR by RAM

Output:

- IR_to_Control: The first 4-bits or, the opcode of the instruction is sent to the Control Sequencer to generate control signals and these bits are put on the bus IR_to_Control
- IR_operand: The last 4-bits, or the operand of the instruction is sent to MAR/PC and put on the bus IR_operand.

5. Accumulator (A):

<u>Functionality</u>:

The accumulator is a 4-bit register. It is used for the following purposes:

a) To store the result of ADD, SUB and AND instructions
b) To move data from the RAM and to the RAM
c) To move data into B using the 'XCHG A,B'
d) To display data using OUT
e) To rotate the bits logically right with 'SHR A'

We will understand more about its functionalities through the simulation of the instruction set later. The connections of the Accumulator module is shown in Fig.7.

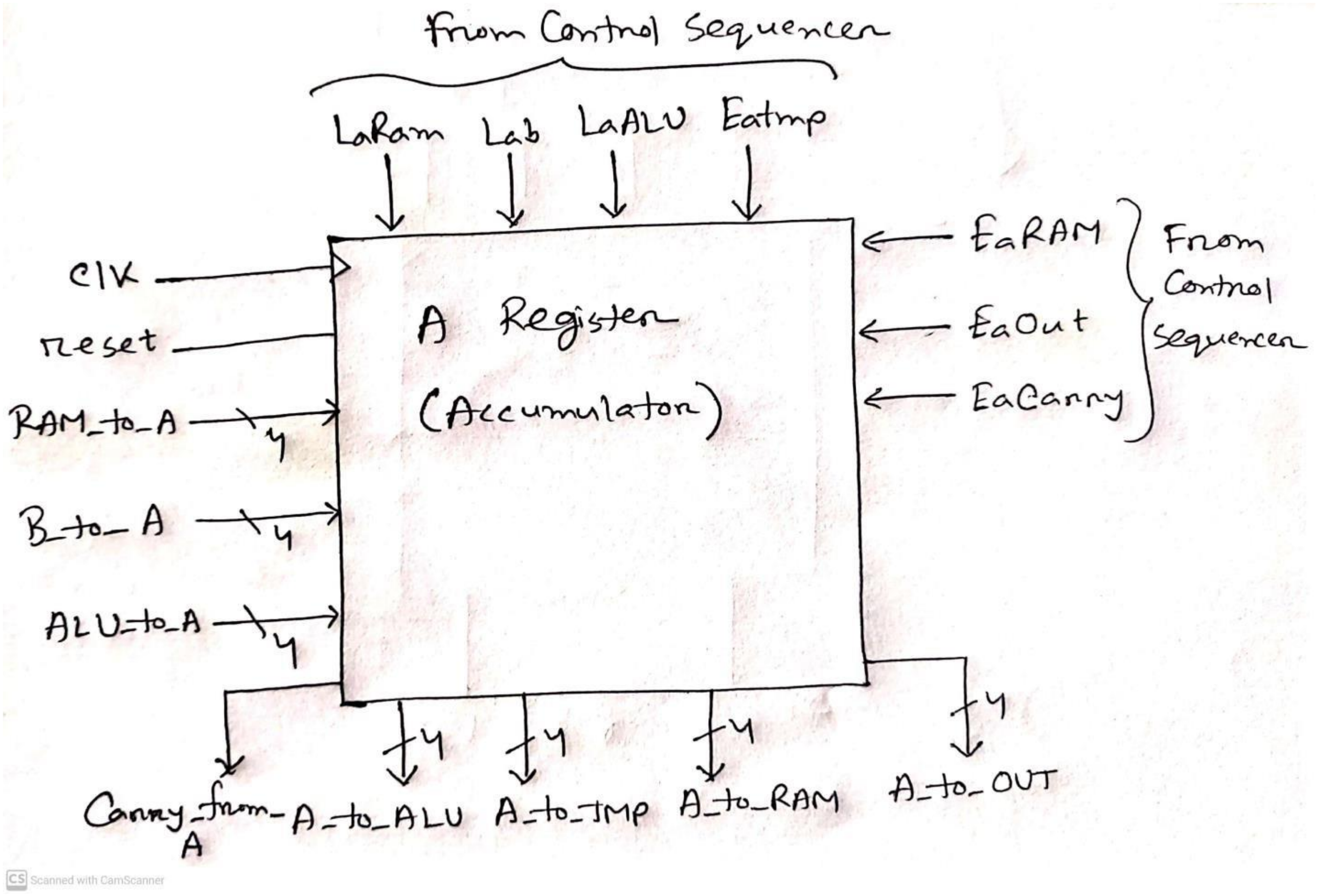


Fig 7. Accumulator Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the Accumulator Register is reset to high impedance.
- L_{aRam}: For 'Mov A, [Address]' instruction, when L_{aRam} is set, the data coming from the RAM is latched onto the Accumulator register.
- RAM_to_A: 4-bit data bus carrying the data from the RAM. Same as before.
- L_{ab}: For 'XCHG A, B' instruction, when L_{ab} is set, the data coming from the B register is latched onto the Accumulator register.
- B_to_A: 4-bit data bus carrying the data from B. Only used for 'XCHG A, B'. Set to high impedance when unused.
- L_{aALU}: For any instruction that performs ALU operation and stores result into A, e.g. ADD, SUB, AND, this signal is set. The data coming from the ALU is latched onto the Accumulator register.
- ALU_to_A: 4-bit data bus carrying the result from the ALU. Always set to the ALU result.
- E_{atmp}: Set for 'XCHG A, B' instruction. The current value stored in A is put on the bus to Temporary Register (A_to_TMP) on the next positive clock edge.
- E_{aRam}: Set for 'Mov [Address], A' instruction. The current value stored in A is put on the bus to RAM (A_to_RAM) on the next positive clock edge.
- E_{aOut}: Set for 'OUT A' instruction. The current value stored in A is put on the bus to Output Register (A_to_OUT) on the next positive clock edge.
- E_{aCarry}: Set for 'SHR A' instruction. The LSB of current value in A is sent to the bus connecting A to the Flag Register to update the carry value

Output:

- A_to_ALU: 4-bit data bus connecting A to ALU. Always holds the current value in A and continuously feeds the ALU.
- A_to_TMP: 4-bit data bus connecting A to Temporary Register (TMP). Carries data from A to TMP. Used for 'XCHG A,B'.
- A_to_RAM: 4-bit data bus connecting A to RAM. Carries data from A to RAM. Used for 'Mov [Address], A'
- A_to_OUT: 4-bit data bus connecting A to Output Register (TMP). Carries data from A to OUT. Used for 'OUT A'
- Carry_from_A: 1-bit wire connecting A to Flag Register (F). Carries the updated carry sent by A after 'SHR A' operation.

6. Buffer Register (B):

Functionality:

The Buffer Register is a 4-bit register. It serves the following purposes:

a) Used as a second operand for ADD, SUB and AND instructions
b) Used for rotating bits through carry ('RCL B')
c) Used for pushing data onto stack and popping data from stack
d) Used to exchange data with B

The connections of the Buffer Register module is shown in Fig.8.

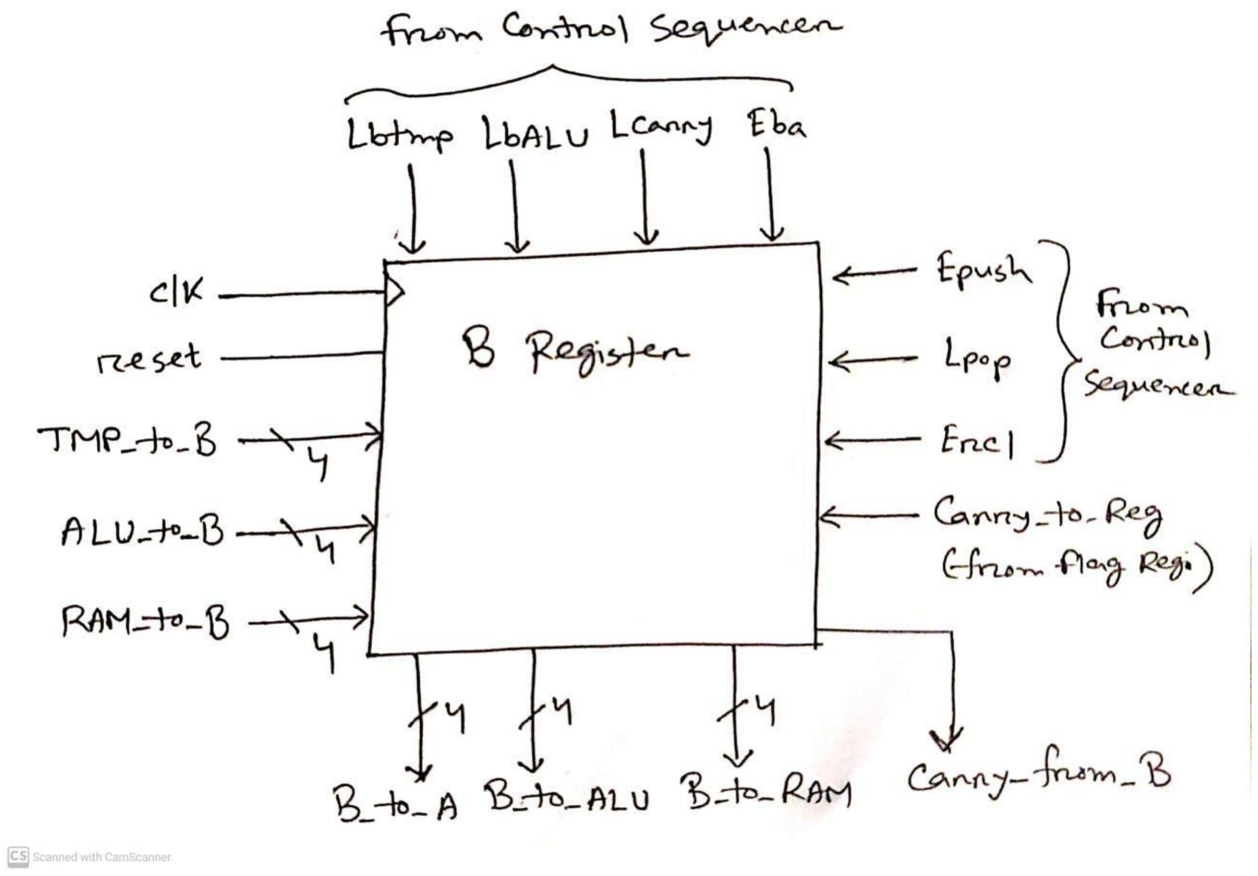


Fig 8. Buffer Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the Buffer Register is reset to zero.
- L_{btmp}: For 'XCHG A, B' instruction, when L_{btmp} is set, the data coming from the Temporary Register (TMP_to_B) is latched onto the Buffer register. This is an asynchronous operation.
- TMP_to_B: 4-bit data bus carrying the data from Temporary Register to B. Used for 'XCHG A,B'.
- L_{bALU}: For 'OR B, [Address]', this signal is set. The data coming from the ALU (ALU_to_B) is latched onto the B register on next positive edge of clock.
- ALU_to_B: 4-bit data bus carrying the result of ALU to B. Used for 'OR B, [Address]'
- E_{push}: For 'Push B operation', this signal is set and on the next positive edge of clock, the current value of B is put on the bus to RAM (B_to_RAM).
- L_{pop}: For 'Pop B operation', this signal is set and on the next positive edge of clock, the data bits coming from RAM (RAM_to_B) is latched onto B register on next positive edge of clock.
- RAM_to_B: 4-bit data bus carrying the previously stored value of B. Used for 'Pop B' operation.
- L_{carry}: For 'RCL B' operation, the carry from the Flag Register (Carry_to_Reg) is latched into a register inside this module. This is an asynchronous operation.
- Carry_to_Reg: 'For RCL B' operation, this bus feeds the current carry bit from the flag register to B.
- E_{rcl}: 'For RCL B' operation, this signal is set and on next positive edge of clock, the bits in B register are rotated. The updated carry is stored and put on the bus connecting B to Flag Register (Carry_from_B).
- E_{ba}: 'For XCHG A,B' operation, this signal is set and on the next positive edge of clock, the current value in B is put on the bus connecting B to A (B_to_A).

Outputs:

- B_to_ALU: 4-bit data bus connecting B to ALU. Always holds the current value in B and continuously feeds the ALU.
- B_to_RAM: 4-bit data bus connecting B to RAM. Carries data from B to RAM. Used for 'PUSH B'
- B_to_A: 4-bit data bus connecting B to A. Carries data from B to A. Used for 'XCHG A,B'
- Carry_from_B: 1-bit data bus connecting B to Flag Register, F. Carries the updated carry after 'RCL B' to F.

7. Temporary Register (TMP):

<u>Functionality</u>:

The Temporary Register is a 4-bit register. It is used for the following purposes:

a) Used as an intermediary storage for ‘XCHG A,B’ operation
b) Used as an intermediary storage for ‘OR B, [Address]’ operation

The connections of the Temporary Register module is shown in Fig.9.

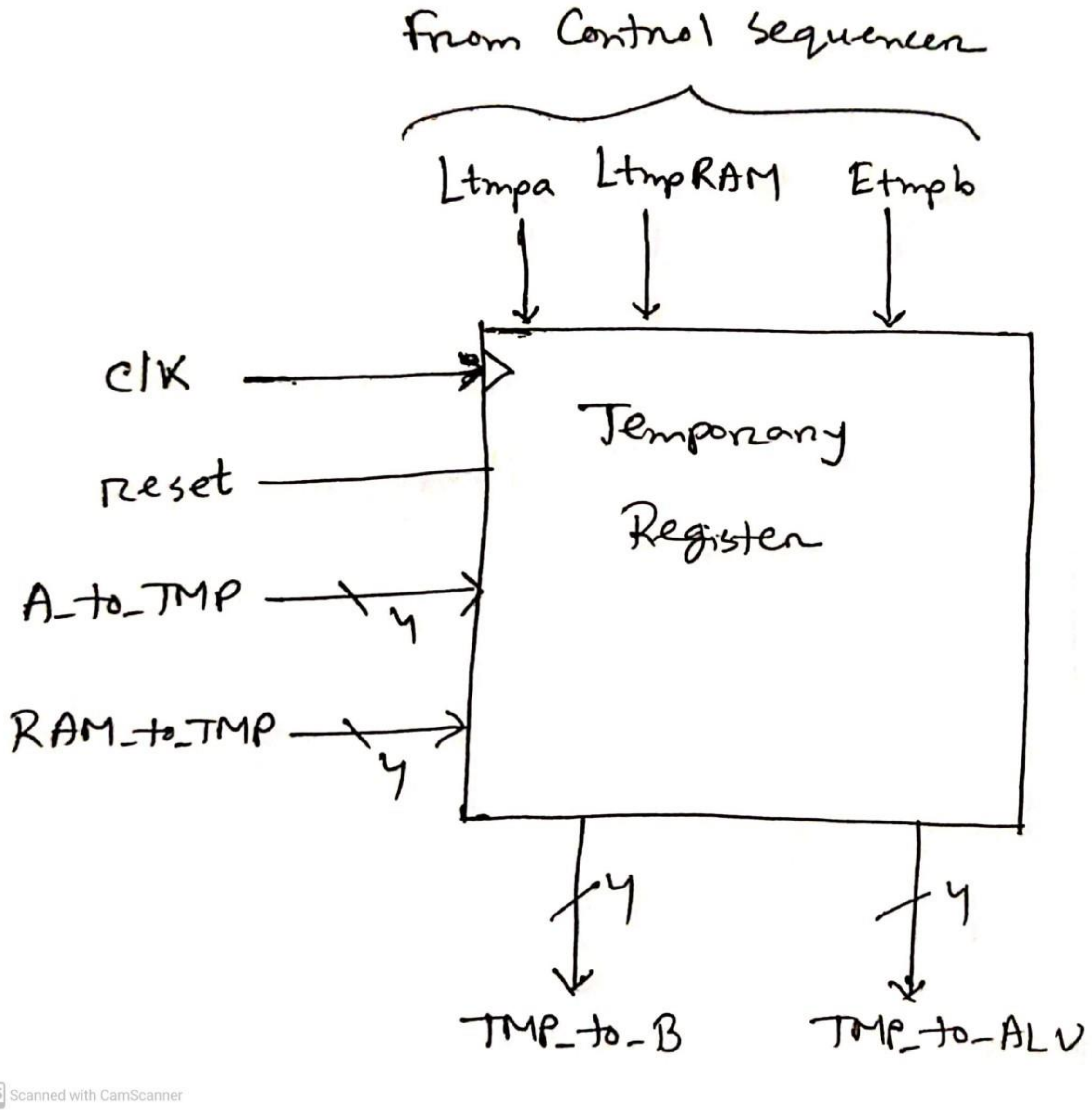


Fig 9. Temporary Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the Temporary Register is reset to high impedance.
- L_{tmpa}: For 'XCHG A, B' operation, this signal is set and data coming from A is latched onto the temporary register. This is an asynchronous operation.
- A_to_TMP: 4-bit data bus connecting A to TMP. Carries data from A to TMP. Used for 'XCHG A, B' operation.
- L_{tmpRam}: For 'OR B, [Address]' operation, this signal is set and data coming from RAM is latched onto the temporary register. This is an asynchronous operation.
- RAM_to_TMP: 4-bit data bus connecting RAM to TMP. Carries data from RAM to TMP. Used for 'OR B, [Address]' operation.
- E_{tmpb}: For 'XCHG A, B' operation, this signal is set and on the next positive clock edge, current data in TMP is put on the bus connecting TMP to B (TMP_to_B).

Output:

- TMP_to_ALU: 4-bit data bus connecting TMP to ALU. Always holds the current value in TMP and continuously feeds the ALU.
- TMP_to_B: 4-bit data bus connecting TMP to B. Carries data from TMP to B. Used for 'XCHG A, B' operation.

8. Arithmetic Logic Unit (ALU):

Functionality:

The ALU performs all arithmetic instructions in the computer. All of these instructions are synchronous, that is, they are performed on the positive edge of the clock. After each instruction, the carry flag and zero flag are updated.

It must be mentioned here that an adder-subtractor module has been written separately to handle the ADD/SUB operations. This module is reported in Fig. 10. It is exactly the same as the adder-subtractor module typically used in a digital electronics circuit. The only difference is at the end the carryout is xor'ed with the initial add/sub control signal. That makes the carry equal to 1 when there is a final borrow. The connections of the Temporary Register module is shown in Fig.11.

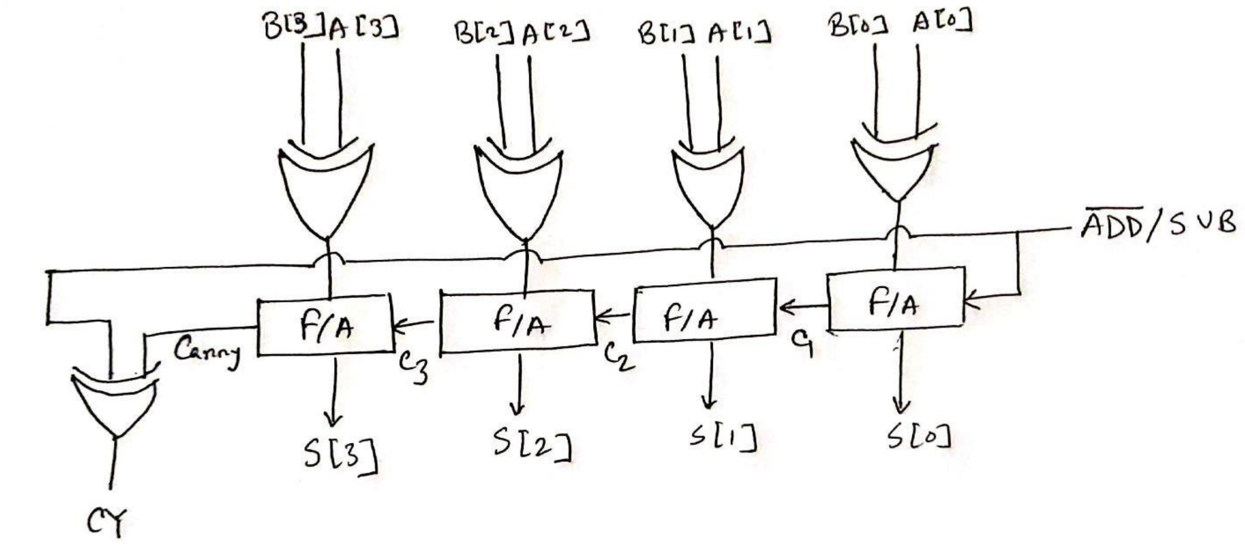


Fig 10. Adder Subtractor Module

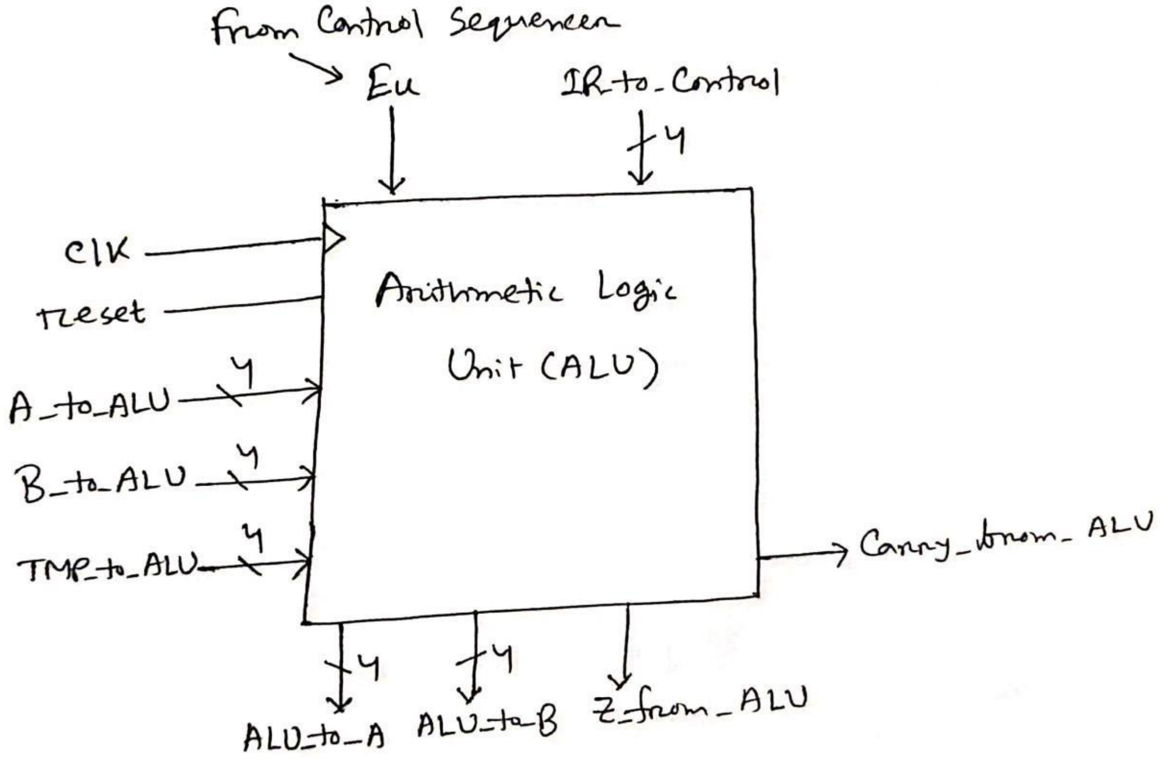


Fig 11. Arithmetic Logic Unit

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the result of the ALU is set to high impedance and the zero flag and carry flag are both set to zero.
- E_u: For any instruction that uses the ALU, this signal is set. On the next positive clock edge, the result of the ALU is put on the bus to A and B. (ALU_to_A, ALU_to_B)
- A_to_ALU: 4-bit data fed to ALU from A. Same as before.
- B_to_ALU: 4-bit data fed to ALU from B. Same as before.
- TMP_to_ALU: 4-bit data fed to ALU from TMP. Same as before.
- IR_to_control: 4-bit opcode indicating the type of ALU instruction e.g ADD/SUB/AND/OR etc.

Outputs:

- ALU_to_A: 4-bit data bus carrying the ALU result to A. Used for all ALU instructions that finally store the result in A
- ALU_to_B: 4-bit data bus carrying the ALU result to B. Used for all ALU instructions that finally store the result in B, e.g OR B,[Address]
- Z_from_ALU: 1-bit wire connecting the ALU to the Flag Register. Used to update the zero flag after every ALU operation
- Carry_from_ALU: 1-bit wire connecting the ALU to the Flag Register. Used to update the carry flag after ADD/SUB operation

9. Output Register (OR):

Functionality:

This register is used for 'OUT A' operation. It takes the data from A and sends it through the output port for display purposes. The connections of the Output Register module is shown in Fig.12.

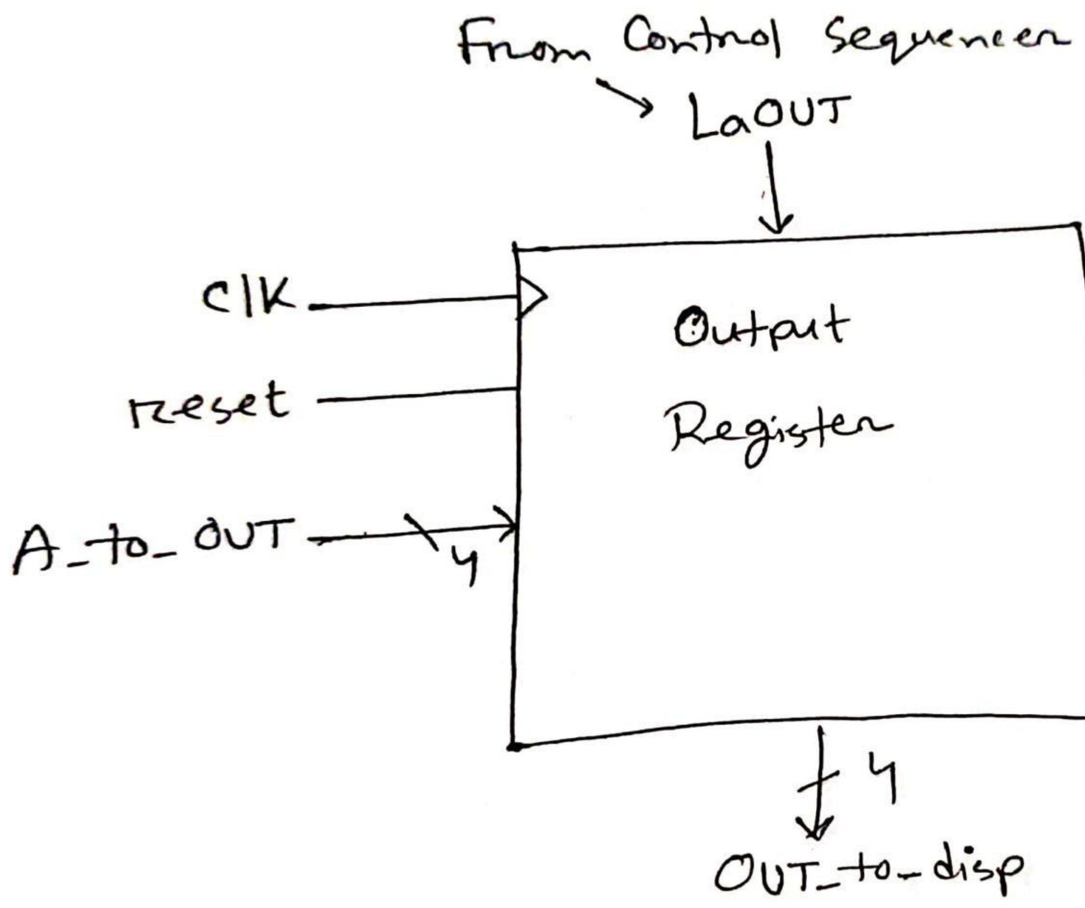


Fig 12. Output Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the output register is set to high impedance.
- L_{aOut}: For 'OUT A' operation, this control signal is set and data coming from A is loaded onto the output register. This is an asynchronous operation.
- A_to_Out: 4-bit data bus carrying the data stored in A to Output Register

Output:

- OUT_to_disp: 4-bit data bus carrying the data stored in Output Register to display for 'OUT A' operation.

10. Flag Register (F):

Functionality:

This register is used to store the zero flag and carry flag from the ALU result. Then, for 'RCL B'/ 'SHR A' operation, it sends the carry to B or A register and for 'JZ Address', it sends the zero flag to the PC. The connections of the Flag Register module is shown in Fig.13.

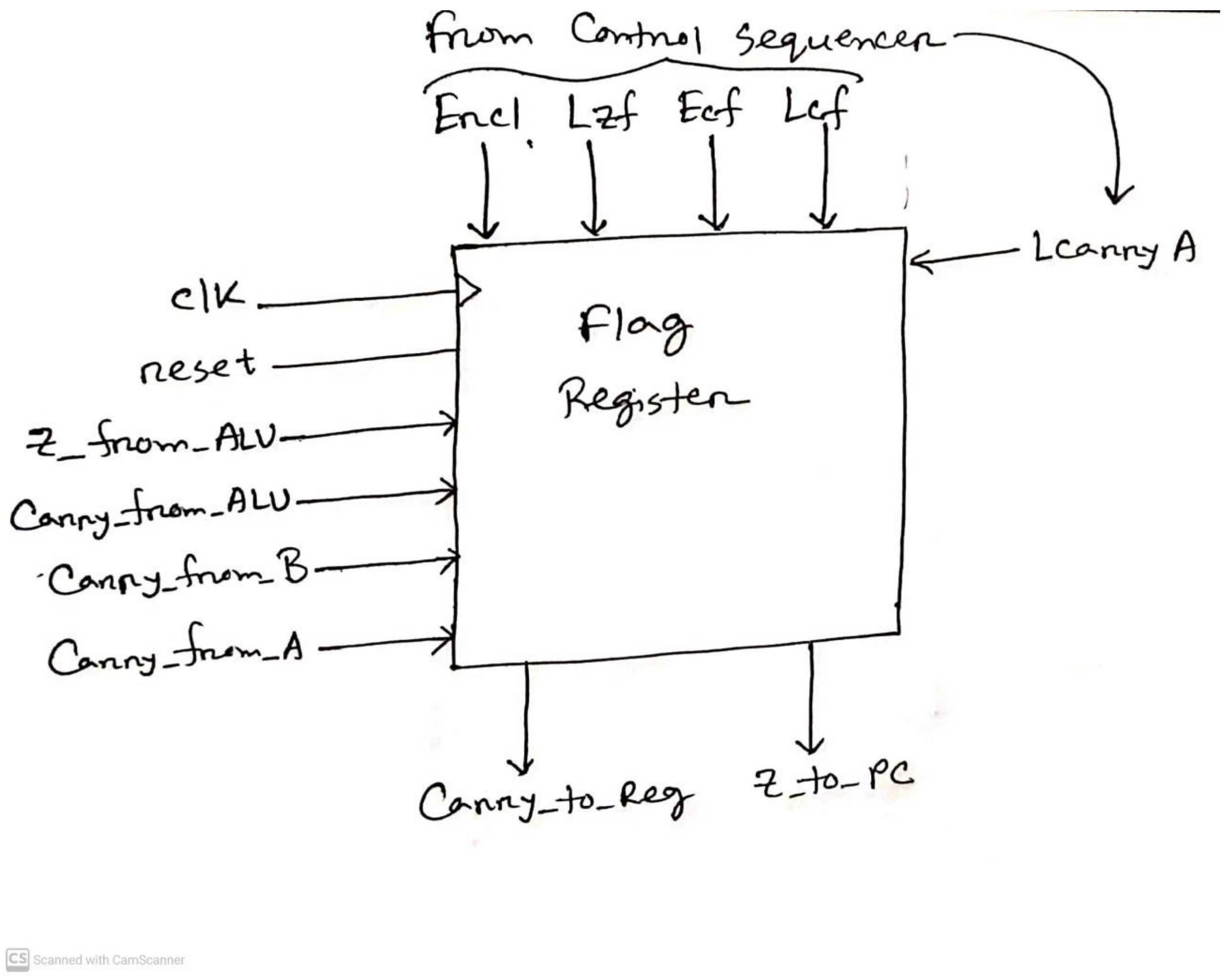


Fig 13. Flag Register

Inputs:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the zero and carry flags are both set to zero.
- L_{zf}: For every ALU operation, this signal is set and the zero flag is updated. This is an asynchronous operation.
- L_{cf}: For every ADD/SUB operation, this signal is set and the carry flag is updated. This is an asynchronous operation.
- Z_from_ALU: Carries the updated zero flag from ALU to F after an ALU instruction is done executing.
- Carry_from_ALU: Carries the updated carry flag from ALU to F after an ADD/SUB instruction is done executing.
- L_{carryA}: For 'SHR A' operation, this signal is set and the updated carry sent from A is loaded to the carry flag register. This is an asynchronous operation.
- Carry_from_A: 1-bit wire connecting A to F and holds the updated carry after 'SHR A'.
- E_{rcl}: For 'RCL B' operation, this signal is set and the updated carry sent from B is loaded to the carry flag register. This is an asynchronous operation.
- Carry_from_B: 1-bit wire connecting B to F and holds the updated carry after 'RCL B'.

Outputs:

- Carry_to_Reg: 1-bit wire carrying the current carry flag from F to B for 'RCL B' operation.
- Z_to_PC: 1-bit wire carrying the current zero flag from F to PC for 'JZ Address' operation.

11. Control Sequencer (CS):

Functionality:

This is the brain of the computer. All control signals are generated here and sent to the different modules. The process of generating control signals is explained below:

a) From the common clock, a ring counter is used inside this module to generate the six states for a machine cycle. The states have a general form such as the following: $T_6 T_5 T_4 T_3 T_2 T_1$, where only one of these is a '1' during a given state. So, for state 1, we have '000001'.
b) Depending on the current state and the type of instruction decoded using the opcode, control signals are set.
c) Once the signals are set, internally the state is changed to the next state by shifting the bit pattern one bit to the left. From '000001', we go to '000010' next and then to '000100' and so on.
d) When the state is '100000', the state resets to '000001'
e) When there is manual reset, the state within the module changes to '000001'.

The ring counter output looks like Fig.15. For the first 3 states, the generated control signals are exactly the same. For the following three states, the control signals change based on the opcode/type of instruction.

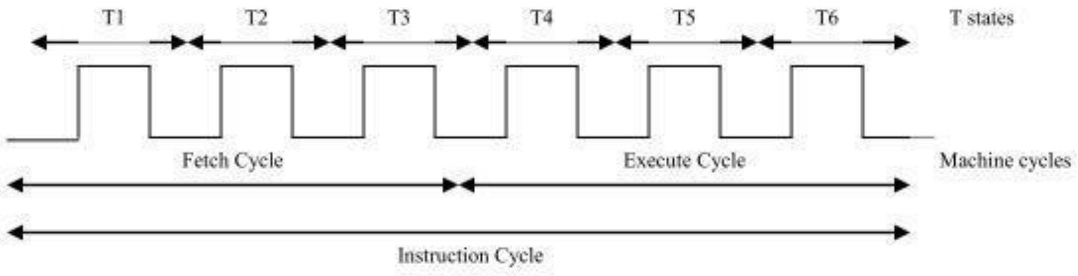


Fig 15. Output of Ring Counter used for different timing states

The connections of the Control Sequencer is shown in Fig.14.

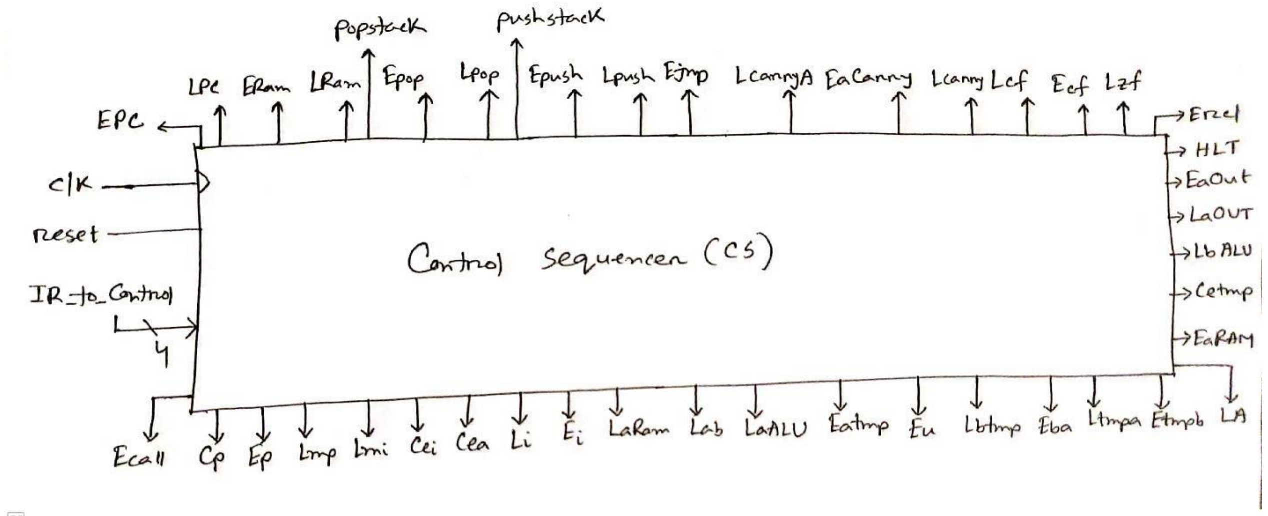


Fig 14. Control Sequencer

<u>Inputs</u>:

- Clk and reset: The common clock and common reset. At the negative edge of reset, the all the control flags are set to zero.
- IR_to_Control: The first 4-bits/opcode coming from the Instruction Register (IR). This denotes the form of instruction that is to be executed next

<u>Outputs</u>:

All control signals are generated by the control sequencer. There are in total **43** of them. And, all of them have been discussed in the description of their respective destination modules.

PART-3: Demonstration of the Computer

In this section, the waveforms of the different instruction sets will be presented. For each instruction, a testbench has been written to fully visualize everything that is happening while that instruction is being executed.

The testbench of the particular instruction is already attached to the submission file in the .zip format. Here I will mainly focus on the significant information such as which control signal is being turned on, and how the different values are changing.

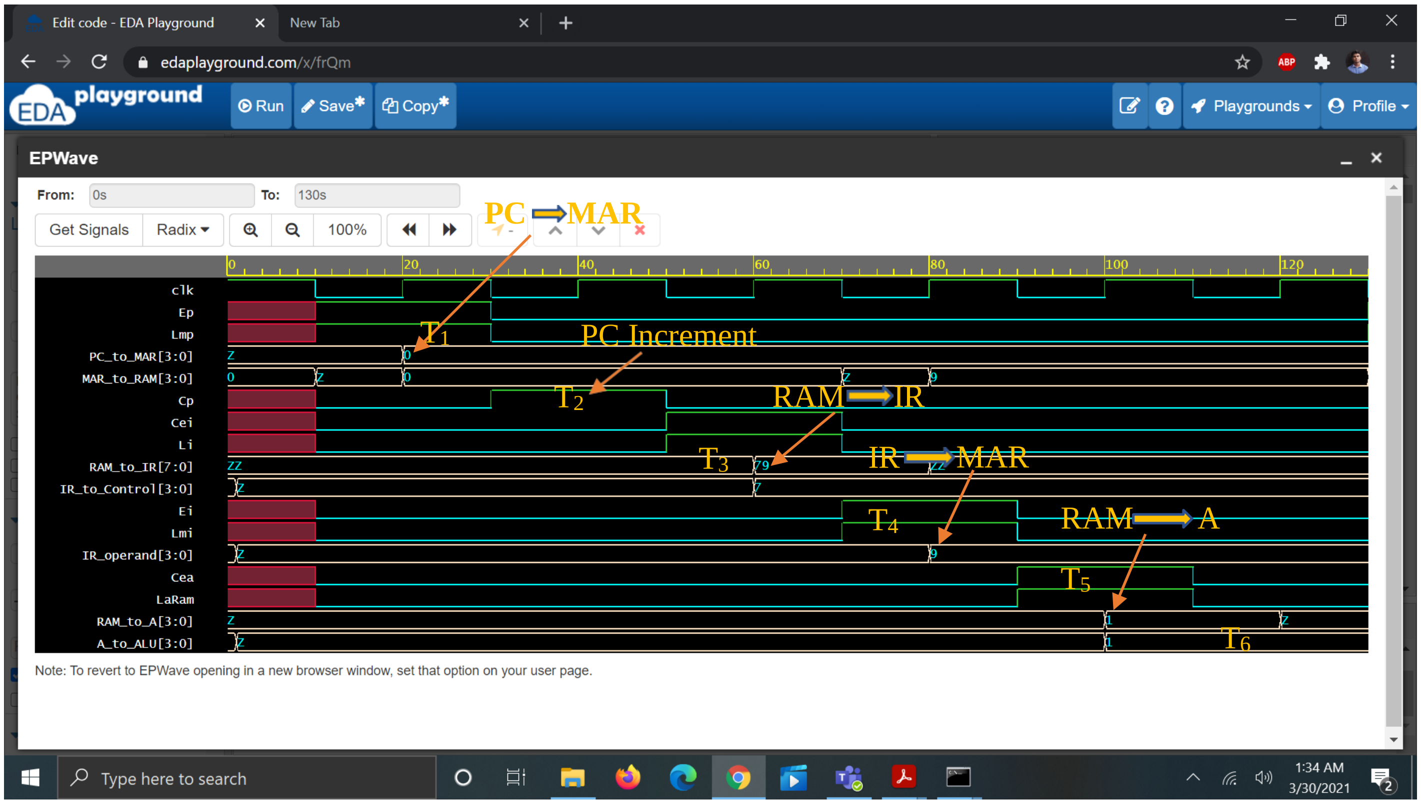
There are six machine states that form a machine cycle. It is also equal to the instruction cycle, so the time to fetch and execute each instruction is kept the same. For any instruction, in the first three states, the control signals generated are the same. They are presented below:

Timing State	Control Signals Set	Purpose
T_1	E_p, L_{mp}	Send the address of the current instruction to the MAR
T_2	C_p (except for HLT in which case this is off)	Increment the program counter by 1
T_3	C_{ei}, L_i	Send the instruction at the addressed location to the instruction register and generate control signals in CS

For each instruction, first the waveform is shown. The waveforms are also edited to simplify understanding of the testbench. After the waveforms, there is a table to demonstrate what happens during each timing state for that instruction.

**For better quality of the waveforms, please check the ‘Results’ folder inside the attached zip file.

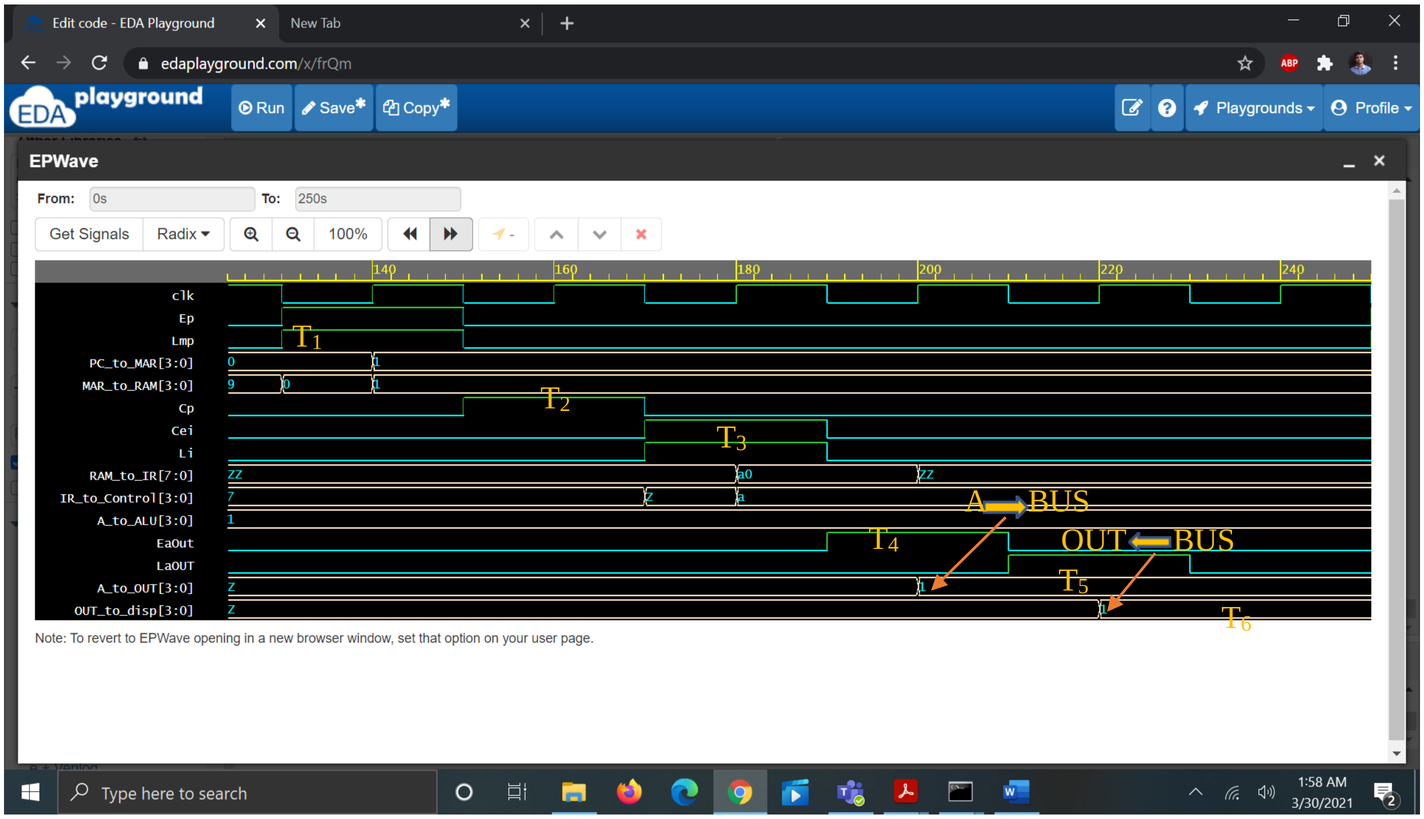
MOV A, Address:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_i, L_{mi}	Send the address location of the data that is to be loaded into A
T_5	C_{ea}, L_{aRam}	Send the data from the addressed location at RAM and load into A
T_6	None	NOP

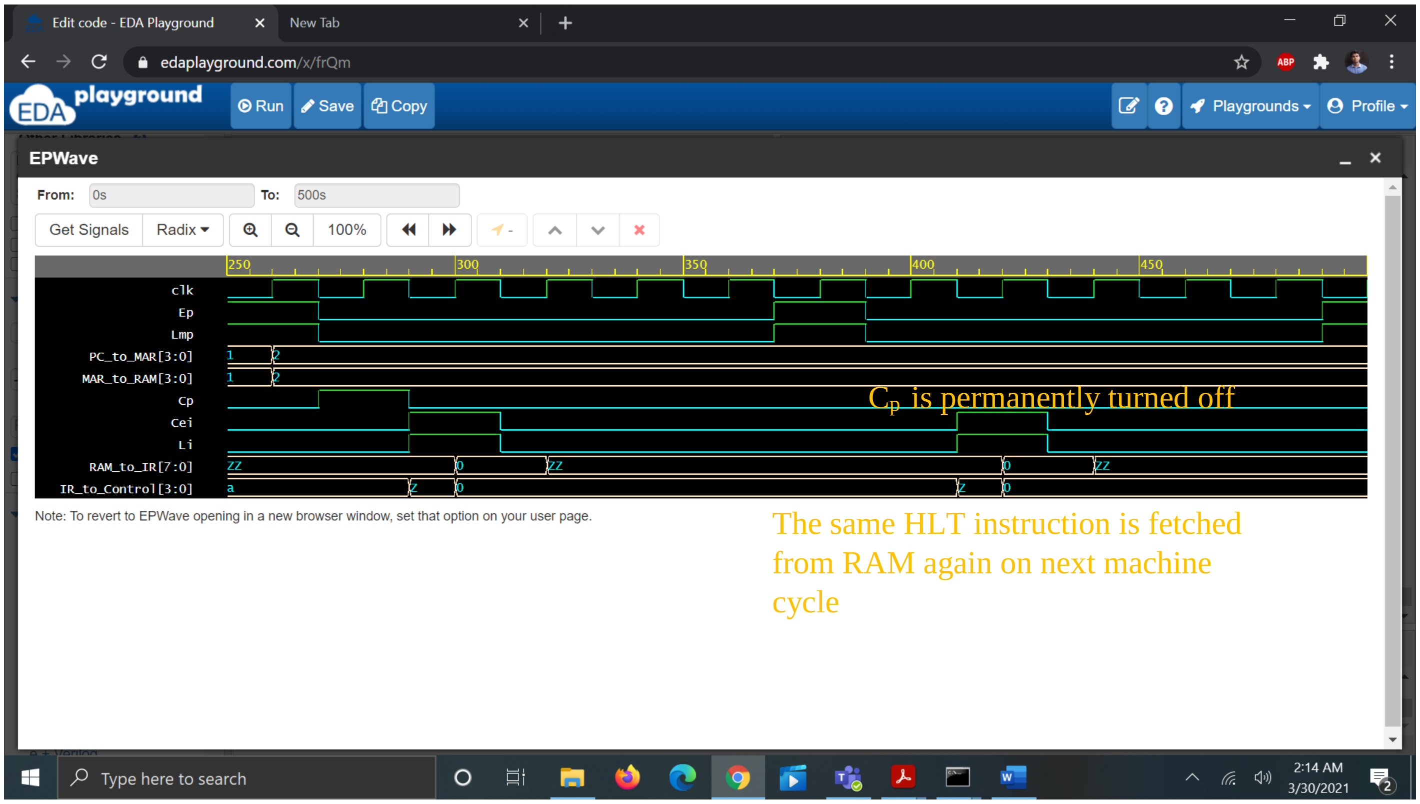
OUT A:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_{aOut}, L_{aOut}	Send the data in A and load output register with that data
T_5	None	NOP
T_6	None	NOP

HLT:



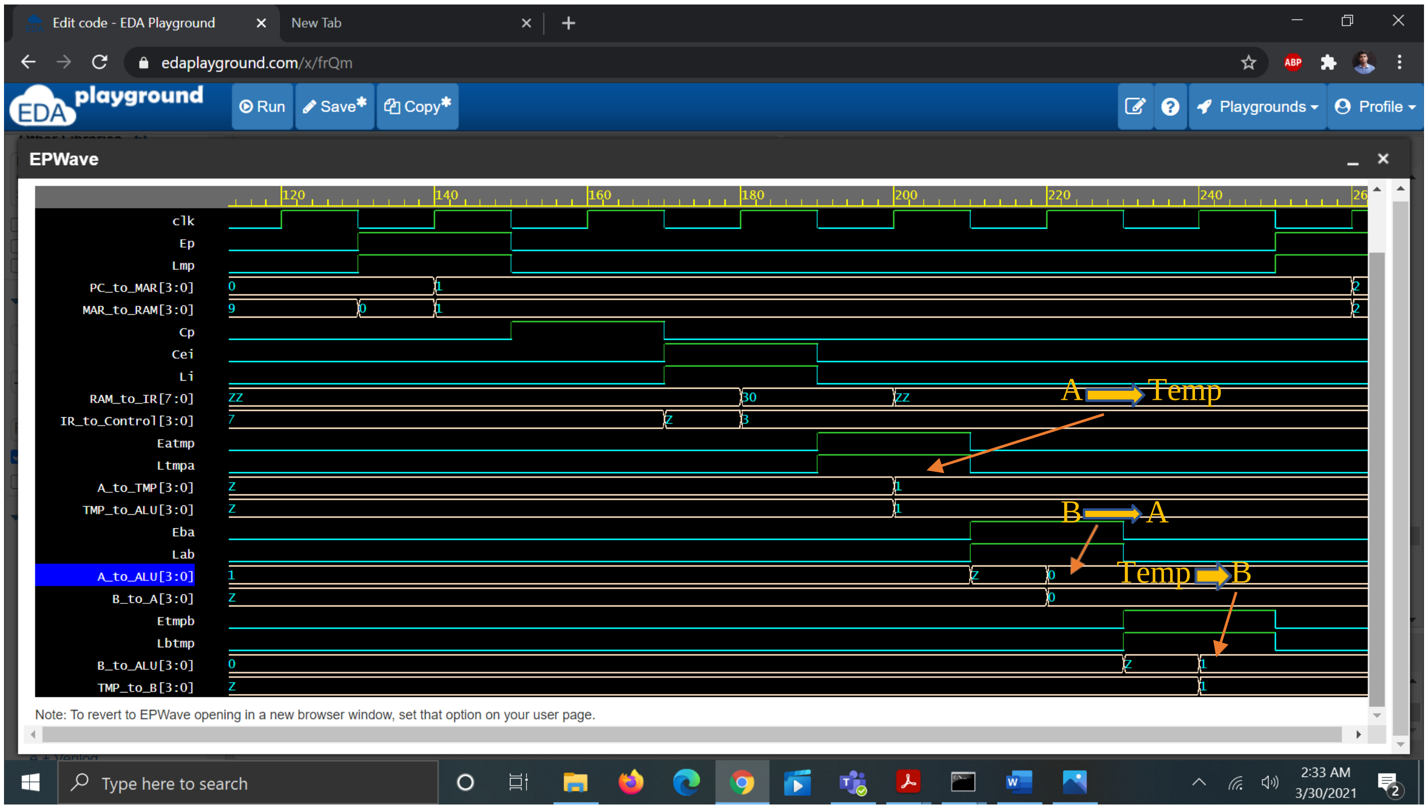
C_p is permanently turned off

Note: To revert to EPWave opening in a new browser window, set that option on your user page.

The same HLT instruction is fetched from RAM again on next machine cycle

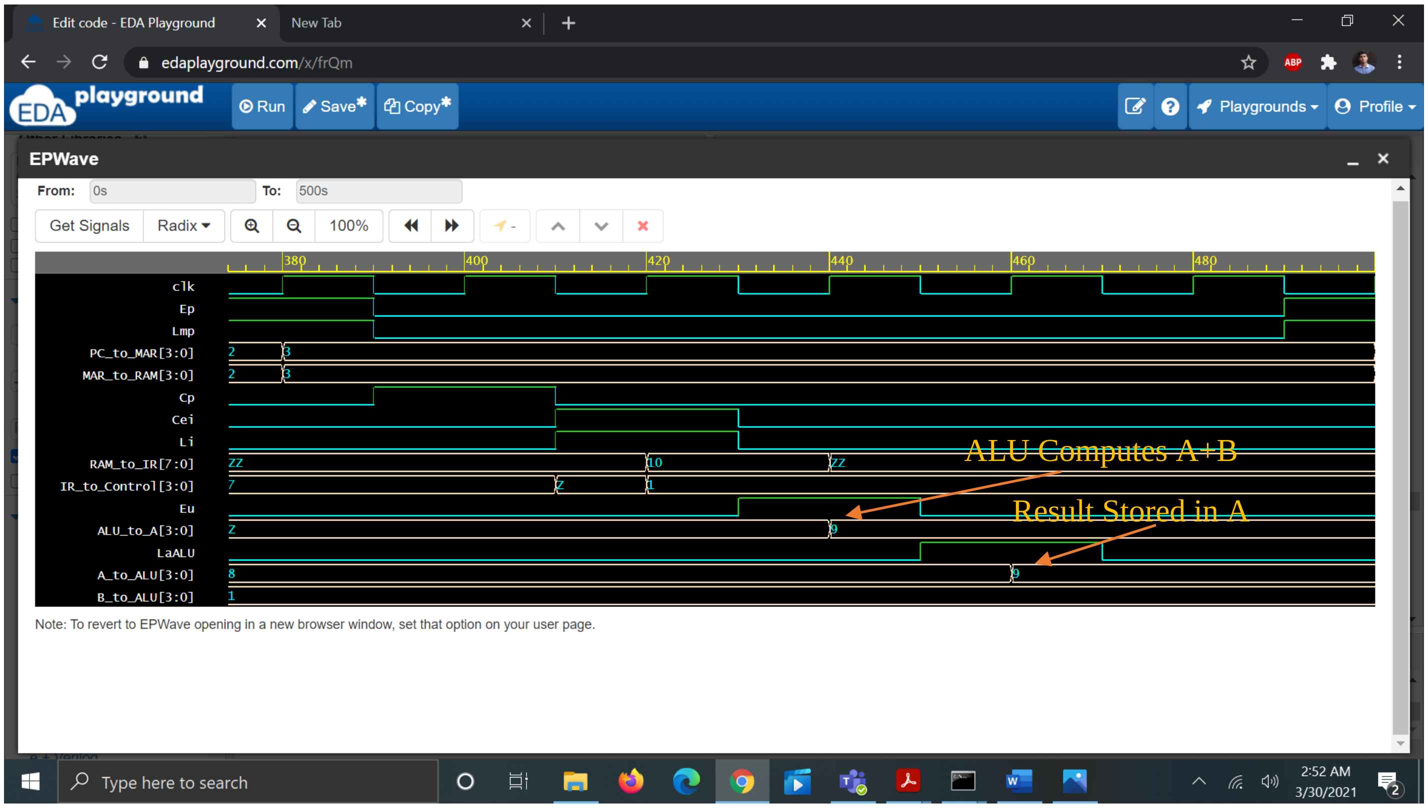
Timing State	Control Signals Set	Purpose
T_4	HLT	The program counter is decremented and for all following instructions, C_p=0
T_5	None	NOP
T_6	None	NOP

XCHG A,B:



Timing State	Control Signals Set	Purpose
T_4	E_{atmp}, L_{tmpa}	Send data in A and load TMP with that data
T_5	E_{ba}, L_{ab}	Send the data in B and load A with that data
T_6	E_{tmpb}, L_{btmp}	Send the data in TMP and load B with that data

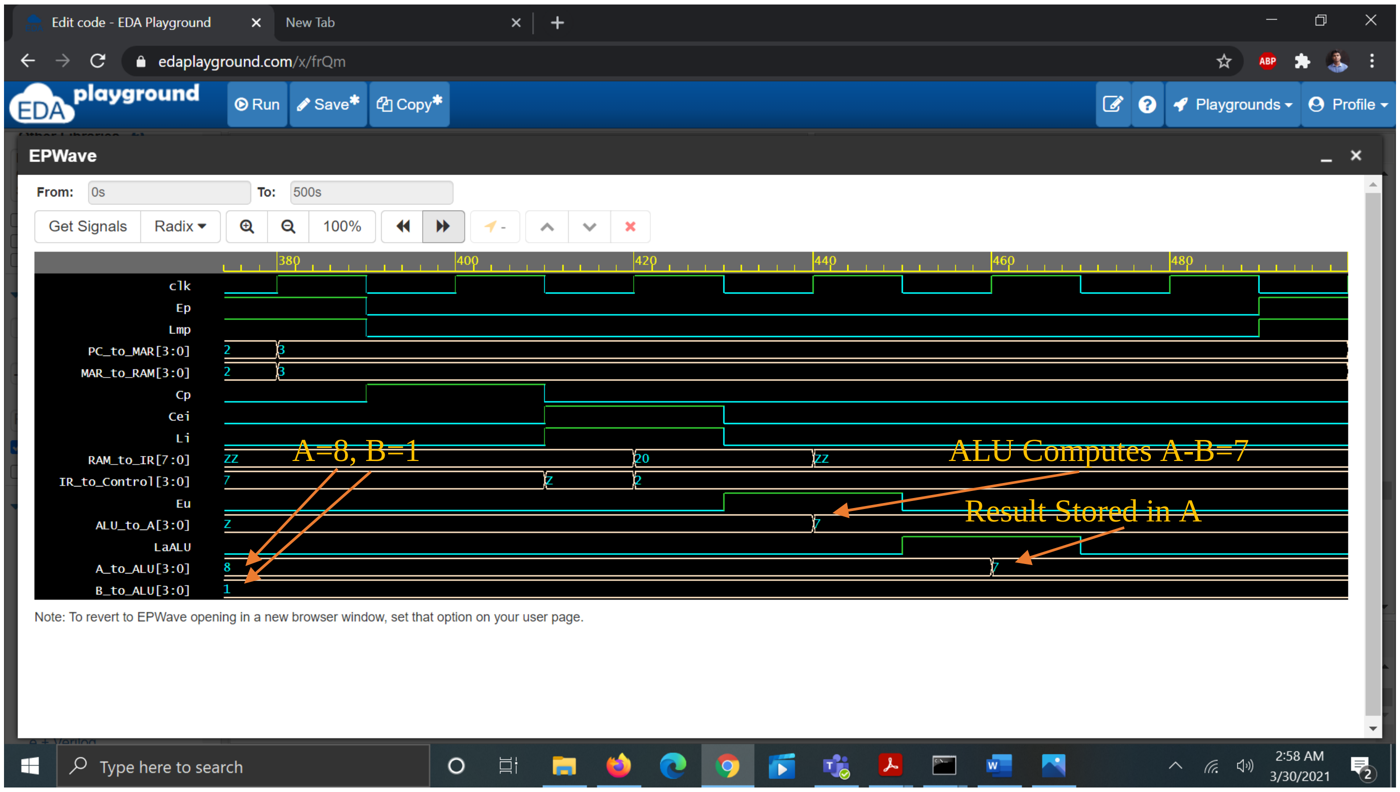
ADD A,B:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_u	Send the result of the ALU after addition and put on the bus to A
T_5	L_{aALU}	Load the result of the ALU from the bus to A
T_6	None	NOP

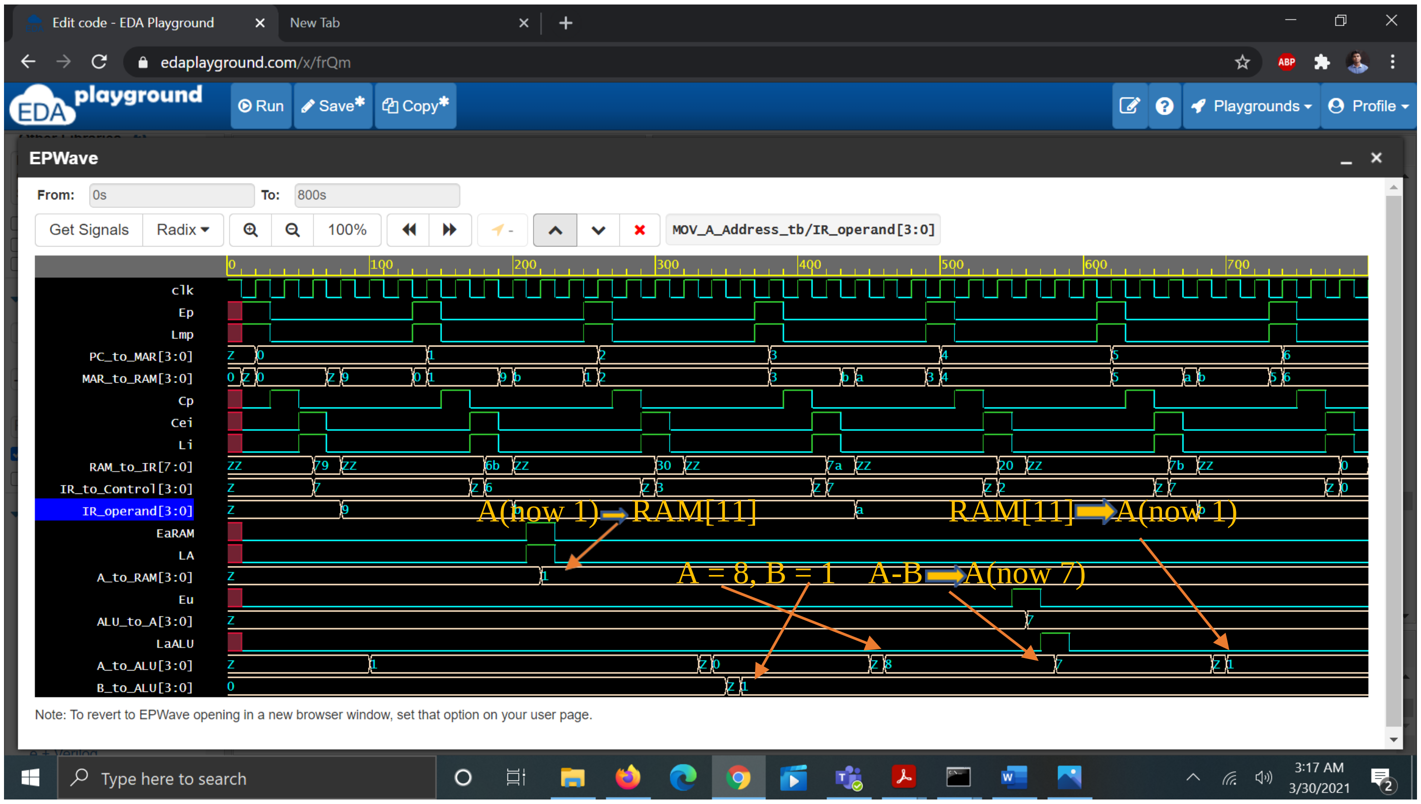
SUB A,B:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_u	Send the result of the ALU after subtraction and put on the bus to A
T_5	L_{aALU}	Load the result of the ALU from the bus to A
T_6	None	NOP

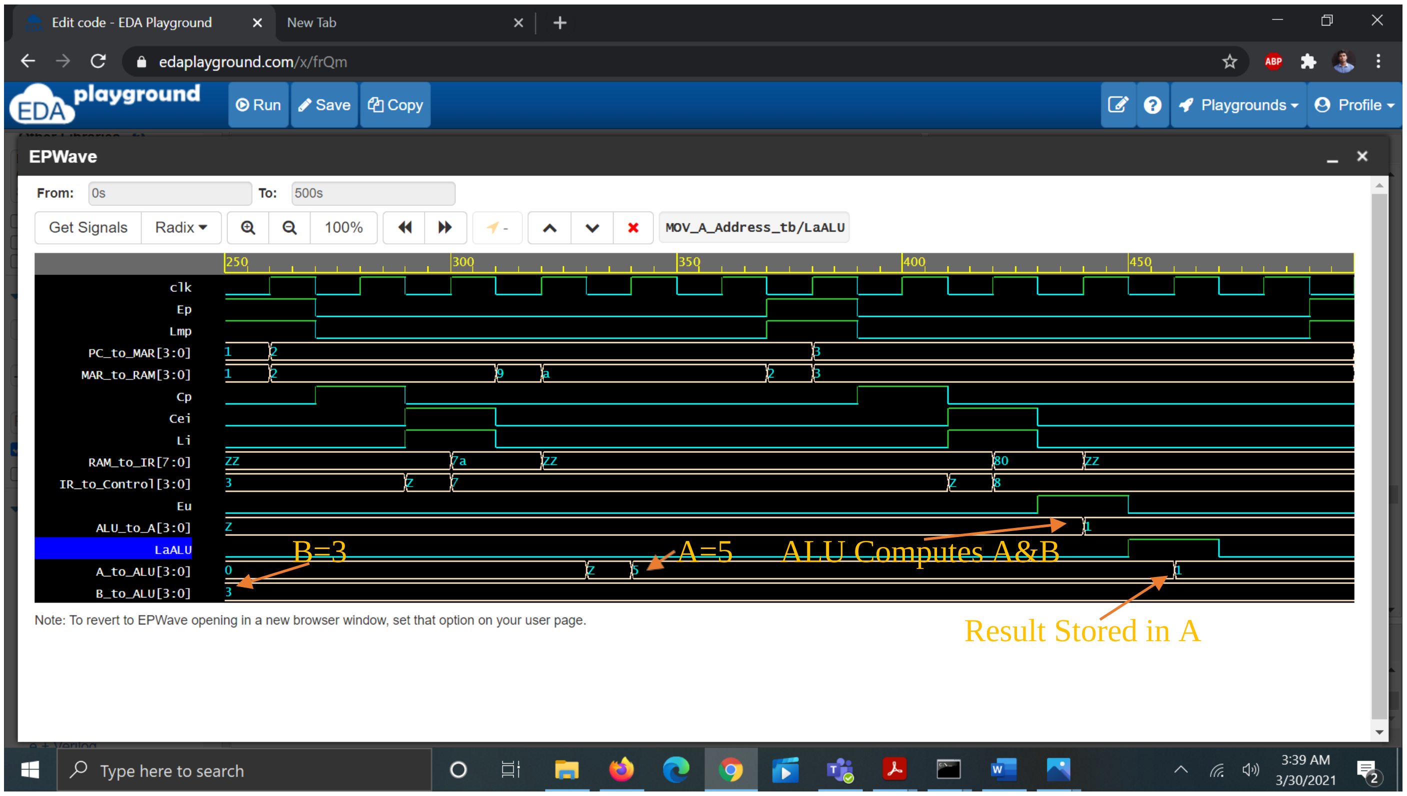
MOV [Address], A



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

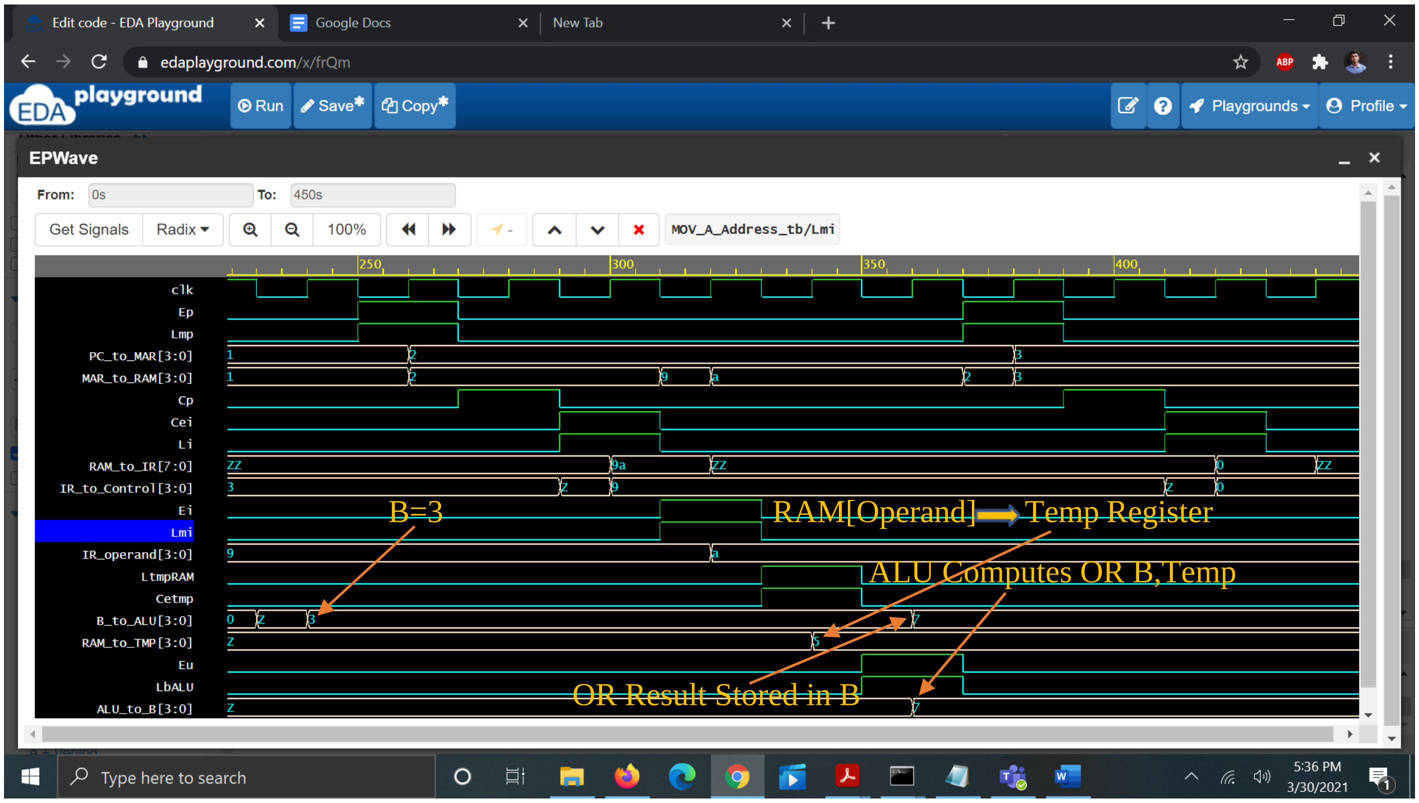
Timing State	Control Signals Set	Purpose
T_4	E_i, L_{mi} (absent in the waveform)	Send the storage address from IR and load MAR with that address
T_5	E_{aRam}, LA	Send the data in A and load the addressed location in RAM with that data
T_6	None	NOP

AND A,B



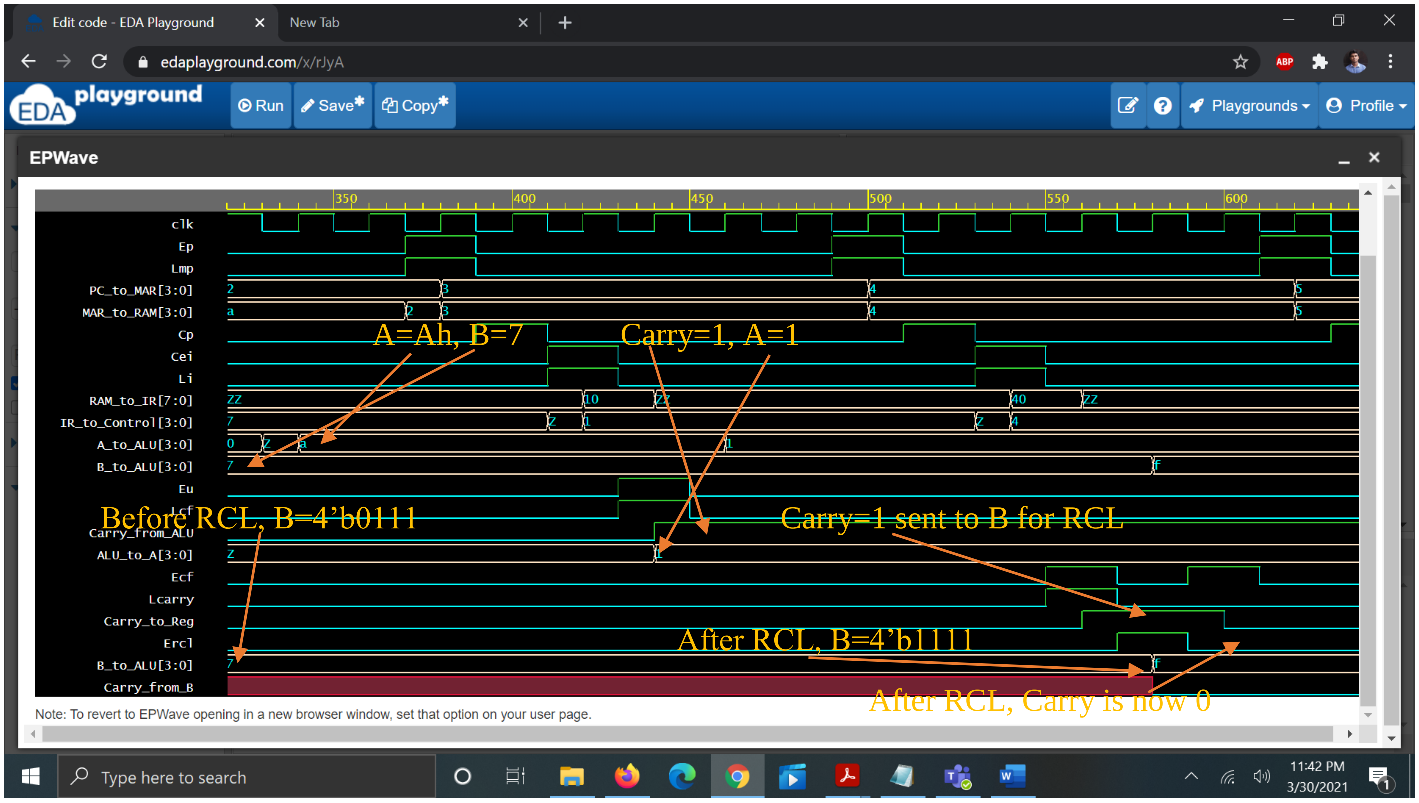
Timing State	Control Signals Set	Purpose
T_4	E_u	Send the result of the ALU after AND and put on the bus to A
T_5	L_{aALU}	Load the result of the ALU from the bus to A
T_6	None	NOP

OR B, [Address]:



Timing State	Control Signals Set	Purpose
T_4	E_i, L_{mi}	Send the address from IR and load MAR with that address
T_5	C_{etmp}, L_{tmpRam}	Send the data in the addressed location at RAM and load TMP with that data
T_6	E_u, L_{bALU}	Send the result of the ALU after OR and load B with that data

RCL B:



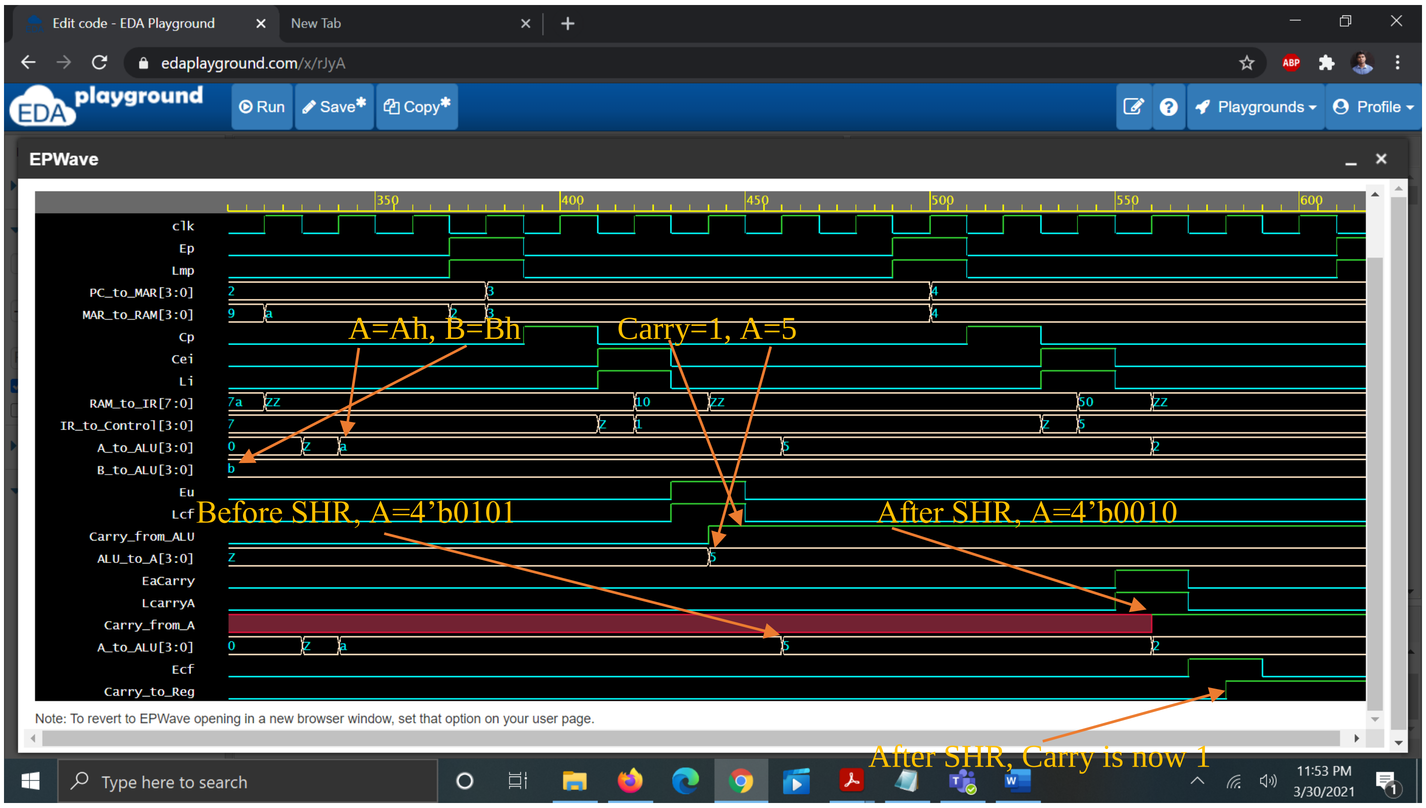
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

On T_4, E_{cf} and L_{carry} are set. So, the carry coming from ALU result is latched onto the carry flag register and appears on the bus to B, because of high E_{cf}. And, inside B, a register stores the value of this incoming carry, because of high L_{cf}.

On T_5, E_{rcl} is set. So, the bits of B is counter-clockwise rotated through the carry. The MSB of B replaces the original carry and the original carry becomes the LSB of B. This ‘new’ carry is stored in B and sent to the flag register. The carry flag register stores this updated carry but it does not appear on the bus to B as yet.

So, on T_6, E_{cf} is set again. Now, the updated carry(MSB of B) appears on the bus to B.

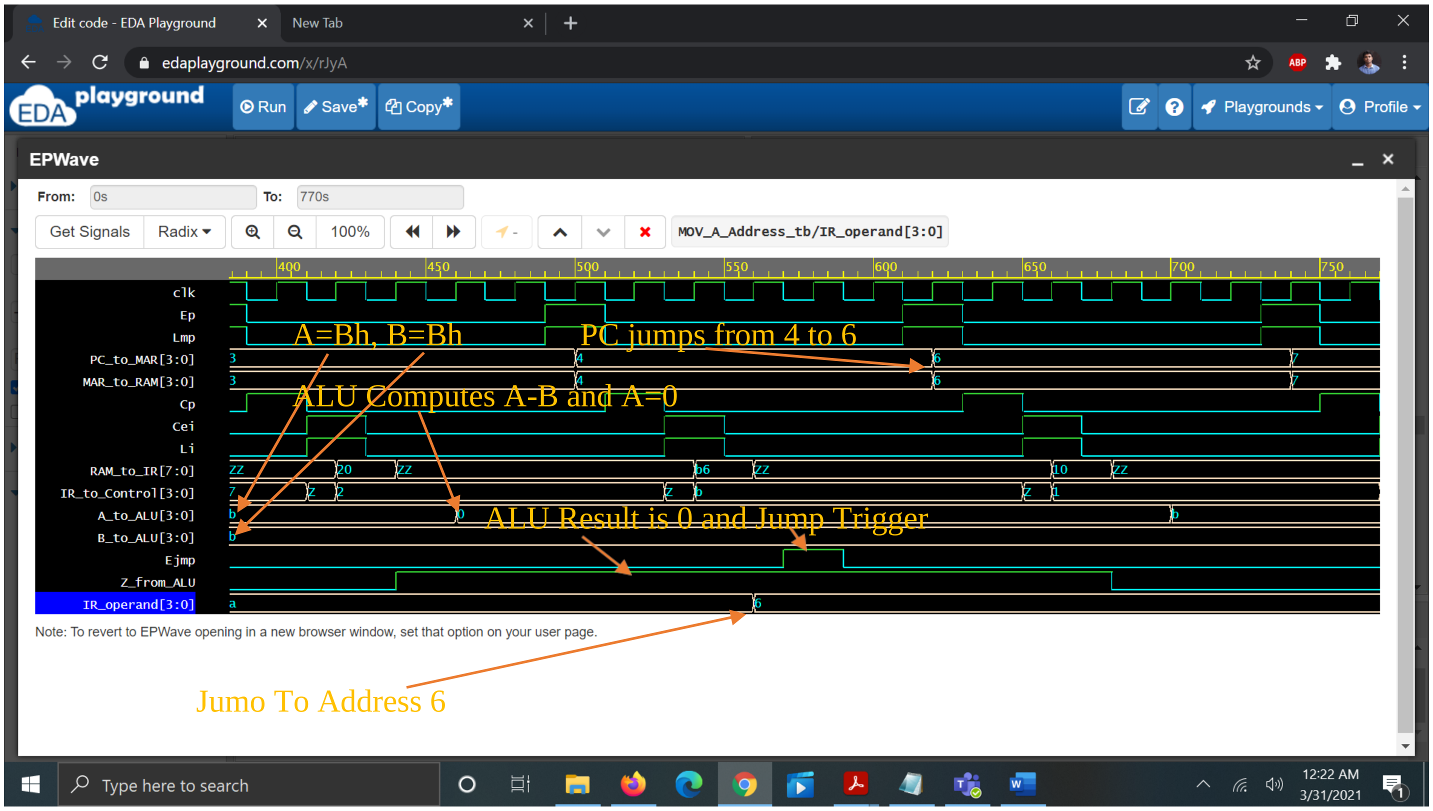
SHR A:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_{aCarry}, L_{carryA}	Perform the shifting operation, send the carry back to F and load F with that carry
T_5	E_{cf}	Put the updated carry on the bus to B (Carry_to_Reg)
T_6	None	NOP

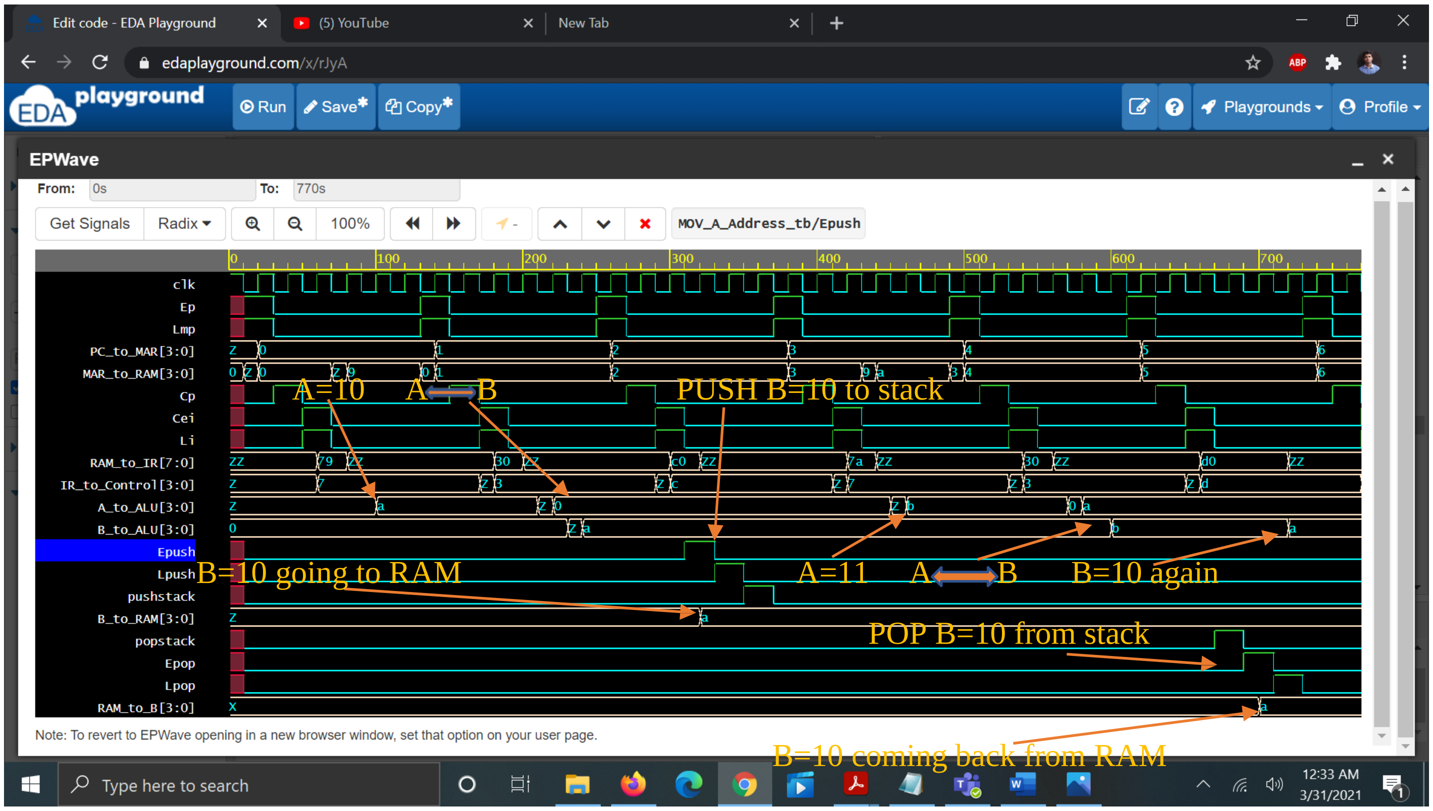
JZ Address:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	Control Signals Set	Purpose
T_4	E_i (absent from the waveform)	Send the address from IR
T_5	E_{jmp}	Load PC with the address sent by IR and Program counter value is replaced by that address
T_6	None	NOP

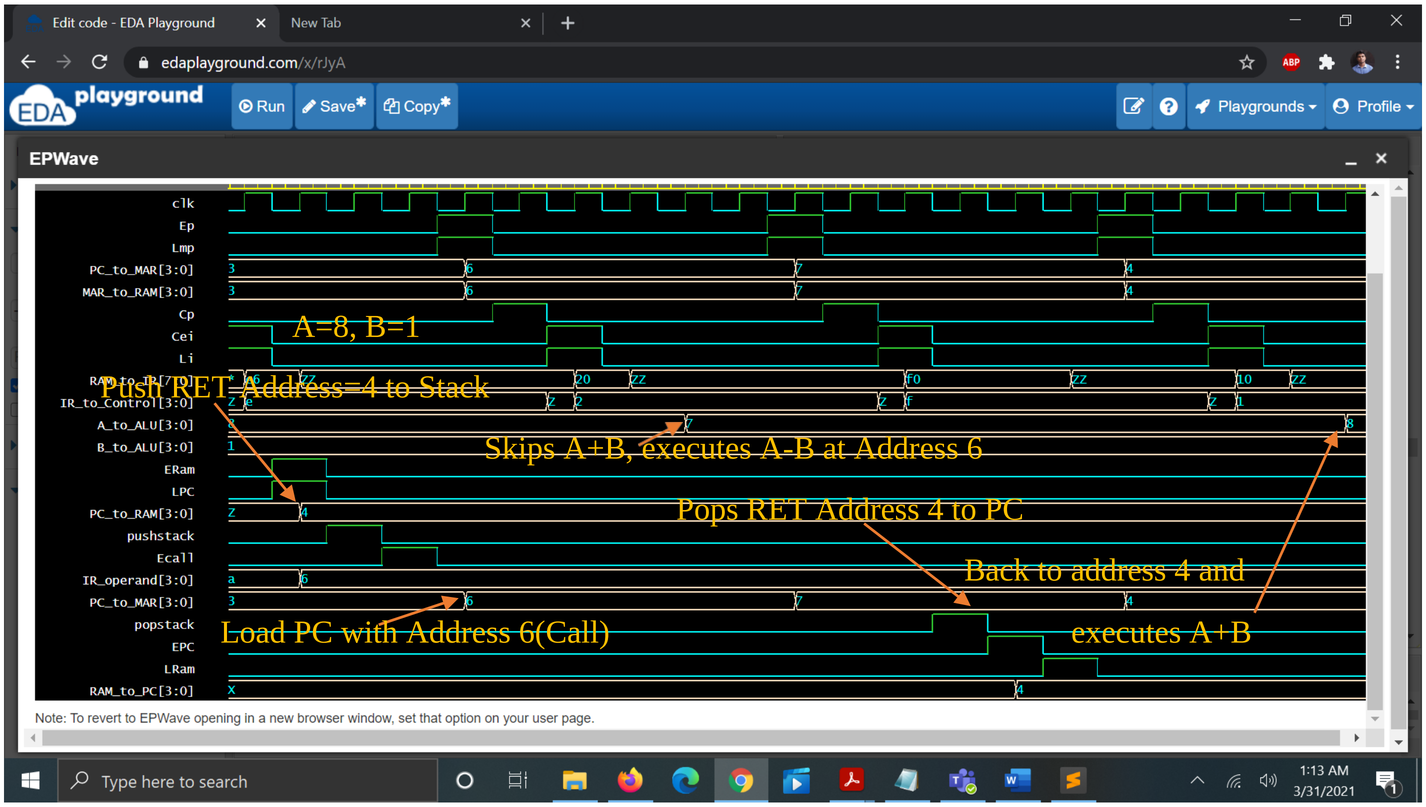
PUSH B, POP B:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	PUSH B		POP B	
	Control Signals Set	Purpose	Control Signals Set	Purpose
T_4	E_{push}	Send the data from B to stack	popstack	Increment stack pointer by 1
T_5	L_{push}	Load stack with data	E_{pop}	Send data from stack to B
T_6	pushstack	Decrement stack pointer by 1	L_{pop}	Load B with that data

CALL ADDRESS, RET:



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Timing State	CALL ADDRESS		RET	
	Control Signals Set	Purpose	Control Signals Set	Purpose
T_4	E_{Ram}, L_{pc}, E_i	Return address is sent from PC and loaded to stack. Send the address from IR and put on the bus to PC	popstack	Increment stack pointer by 1
T_5	pushstack	Decrement stack pointer by 1	E_{PC}	Send return address back to PC
T_6	E_{call}	Load PC with the address to beginning of subroutine	L_{pop}	Load PC with the return address

Assembler

I have written an assembler for my processor which is attached with my submission files.

The idea is pretty simple. First, I read a text file which contains the assembly code. Then, I send each line to a function that converts that line to a machine code.

Inside the function, there are if-else blocks which essentially do some string operation to find out the opcode first. The opcodes are the first 4 bits of the corresponding machine code.

And, then depending on the type of instruction, it checks whether there needs to be a valid operand/address field. There is a valid address field for MOV, JZ Address, and CALL instructions.

If there is not a need for the operand, the last 4 bits of the machine code is padded with zeros.

Sample:

Assembly Code	Machine Code
MOV A,9h	8'b0111_1001
XCHG A,B	8'b0011_0000
MOV A,Ah	8'b0111_1010
SUB A,B	8'b0010_0000
JZ 6h	8'b1011_0110
SHR A	8'b0101_0000
OR B,9h	8'b1001_1001
OUT A	8'b1010_0000
HLT	8'b0000_0000

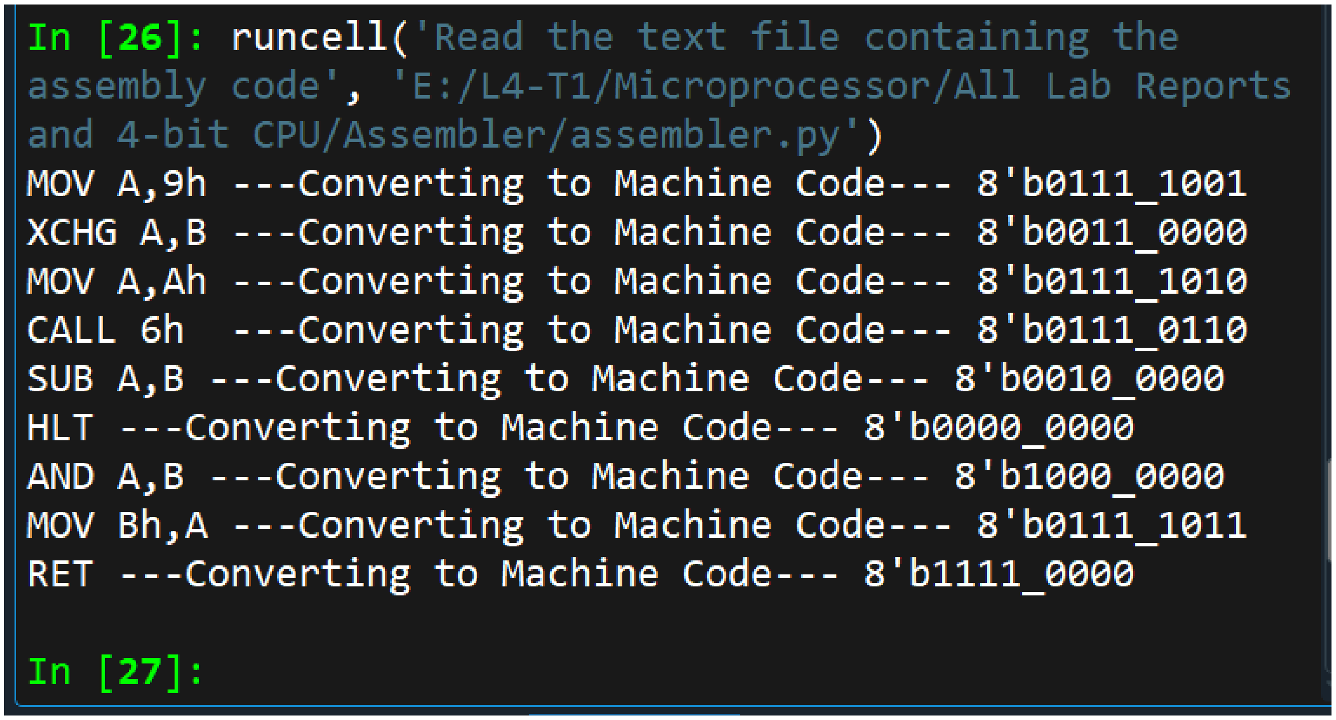
Code:

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 30 15:52:54 2021

@author: Shafin Bin Hamid
"""



#%%Read the text file containing the assembly code
machine_code = []
filetoRead = open('sample_2.txt','r')
fullCode = filetoRead.read()
for line in fullCode.strip().split('\n'):
    m_code = convert_to_machine_code(line)
    machine_code.append(m_code)
    print(line, "---Converting to Machine Code---", m_code)

filetoWrite = open("machine_sample_2.txt", "w")

for i in range(len(machine_code)):
    #print(machine_code[i])
    filetoWrite.write(machine_code[i])
    filetoWrite.write('\n')
```

```
#%%
#Write a function to convert a line of assembly code

def convert_to_machine_code(line):
    #print(line)
    m_code = "8'b"

    if line.startswith('MOV'):
        #print(line)
        if line[4]=='A':
            #print(line)
            #denoting mov a, [address]
            operand = bin(int(line[6], 16))[2:]
            if len(operand)==3:
                operand = '0' + operand
            m_code = m_code+"0111_"+operand
            #m_code = m_code+"0111_"+bin(int(line[6], 16))[2:]
            #print(line, "---Converting to Machine Code---", m_code)

        else:
            #denoting mov [address], a
            operand = bin(int(line[4], 16))[2:]
            if len(operand)==3:
                operand = '0' + operand
            m_code = m_code+"0111_"+operand
            #m_code = m_code+"0110_"+bin(int(line[4], 16))[2:]
            #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('CALL'):
        #denoting call address
        operand = bin(int(line[5], 16))[2:]
        if len(operand)==3:
            operand = '0' + operand
        m_code = m_code+"0111_"+operand
        #m_code = m_code+"1110_"+bin(int(line[5], 16))[2:]
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('JZ'):
        #denoting jz address
        operand = bin(int(line[3], 16))[2:]
        if len(operand)==3:
            operand = '0' + operand
        m_code = m_code+"0111_"+operand
        #m_code = m_code+"1011_"+bin(int(line[3], 16))[2:]
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('OR'):
        #denoting or b, [address]
        operand = bin(int(line[5], 16))[2:]
        if len(operand)==3:
            operand = '0' + operand
        m_code = m_code+"0111_"+operand
        #m_code = m_code+"1001_"+bin(int(line[5], 16))[2:]
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('ADD'):
        m_code = m_code+"0001_0000"
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('SUB'):
        m_code = m_code+"0010_0000"
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('AND'):
        m_code = m_code+"1000_0000"
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('XCHG'):
        m_code = m_code+"0011_0000"
        #print(line, "---Converting to Machine Code---", m_code)

    elif line.startswith('RCL'):
        m_code = m_code+"0100_0000"
        #print(line, "---Converting to Machine Code---", m_code)
```

```
elif line.startswith('SHR'):
    m_code = m_code+"0101_0000"
    #print(line, "---Converting to Machine Code---", m_code)

elif line.startswith('OUT'):
    m_code = m_code+"1010_0000"
    #print(line, "---Converting to Machine Code---", m_code)

elif line.startswith('HLT'):
    m_code = m_code+"0000_0000"
    #print(line, "---Converting to Machine Code---", m_code)

elif line.startswith('PUSH'):
    m_code = m_code+"1100_0000"
    #print(line, "---Converting to Machine Code---", m_code)

elif line.startswith('POP'):
    m_code = m_code+"1101_0000"
    #print(line, "---Converting to Machine Code---", m_code)

elif line.startswith('RET'):
    m_code = m_code+"1111_0000"
    #print(line, "---Converting to Machine Code---", m_code)

return m_code
```

Sample Results:


```
In [26]: runcell('Read the text file containing the
assembly code', 'E:/L4-T1/Microprocessor/All Lab Reports
and 4-bit CPU/Assembler/assembler.py')
MOV A,9h ---Converting to Machine Code--- 8'b0111_1001
XCHG A,B ---Converting to Machine Code--- 8'b0011_0000
MOV A,Ah ---Converting to Machine Code--- 8'b0111_1010
CALL 6h  ---Converting to Machine Code--- 8'b0111_0110
SUB A,B ---Converting to Machine Code--- 8'b0010_0000
HLT ---Converting to Machine Code--- 8'b0000_0000
AND A,B ---Converting to Machine Code--- 8'b1000_0000
MOV Bh,A ---Converting to Machine Code--- 8'b0111_1011
RET ---Converting to Machine Code--- 8'b1111_0000

In [27]:
```