Introduction:

For this assignment, we had to construct a simplified 4-bit computer simulation using proteus.

The said 4-bit PC is consisted of different registers(ACC, MAR, MDR, etc), counters (PC,SP), ROMs, a RAM and an arithmetic logic unit.

Our PC is able to execute any of the 28 instructions that were assigned to us.

The rest of this report contains implementation details of 4-bit PC on proteus.

Assigned Instruction set:

LDA address; STA address; MOV Acc, B; MOV B, Acc;
MOV Acc, immediate; IN; OUT; ADD B; ADC B; SUB B; SBB B;
ADD immediate; SUB immediate; CMP B; NEG; JO address; JNE address;
PUSH; POP; CALL address; RET; JMP; HLT; NOP;
XCHG; CMC; OR [address]; XOR B;

Block diagram of 4-bit PC:

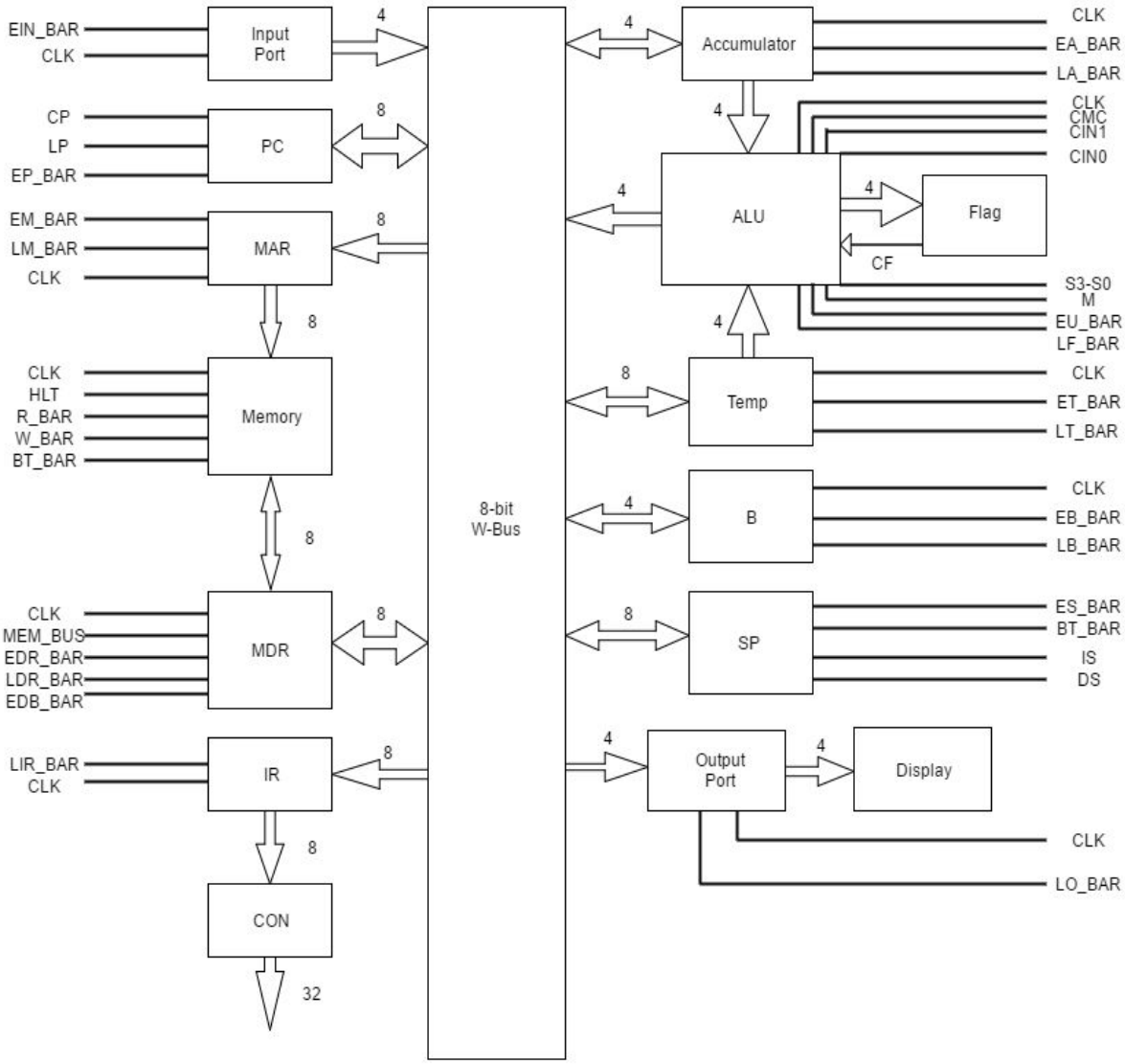
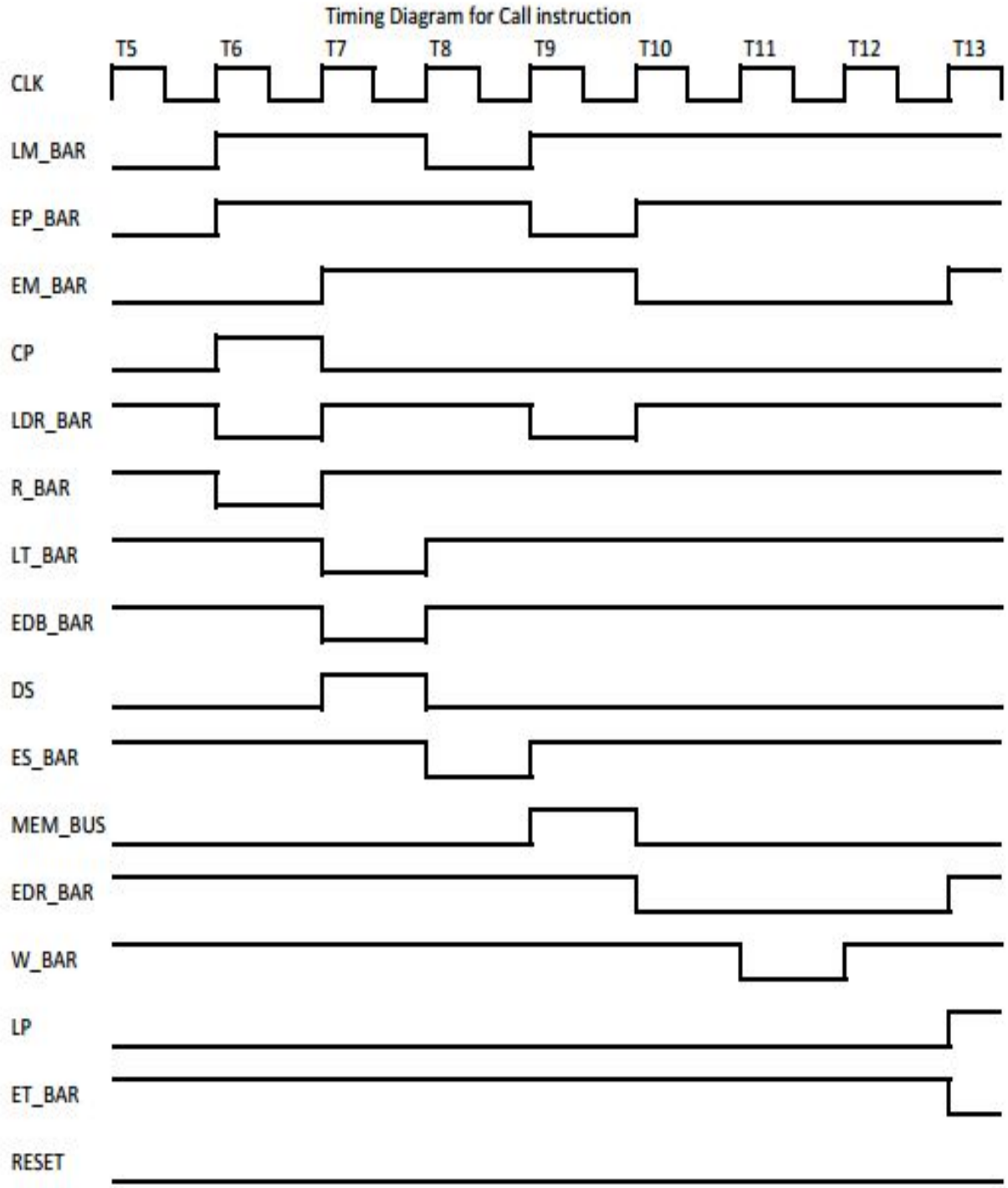


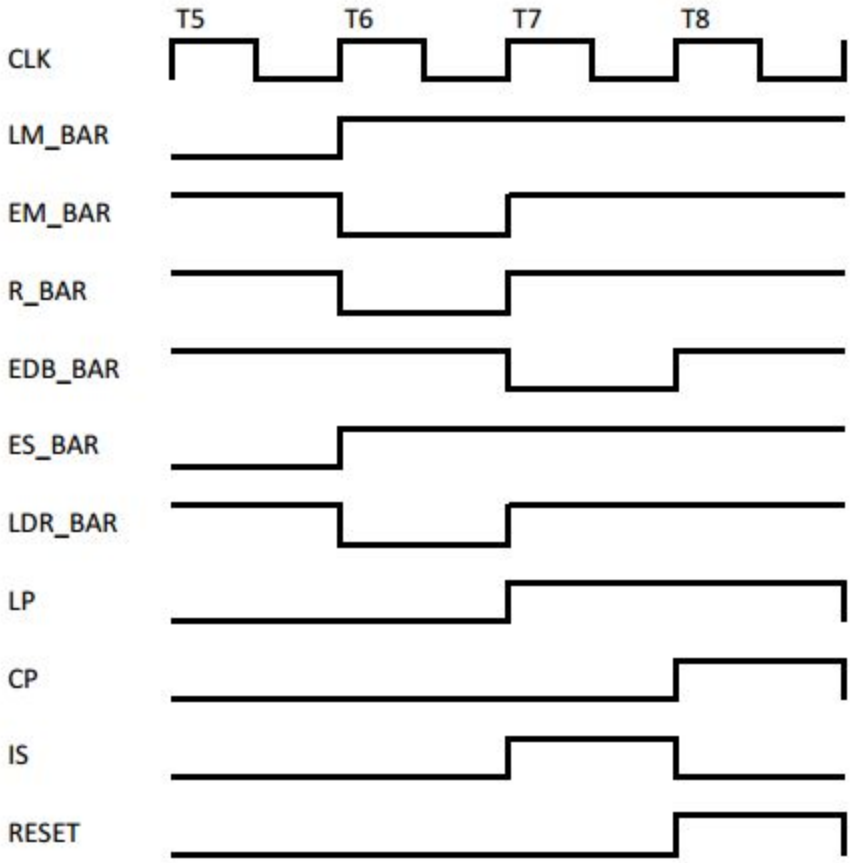
Figure: Block diagram of 4-bit PC

Timing Diagrams:

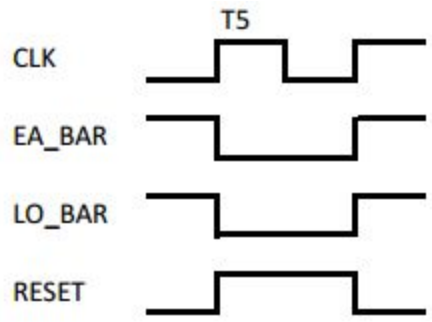
Timing diagram for CALL instruction.



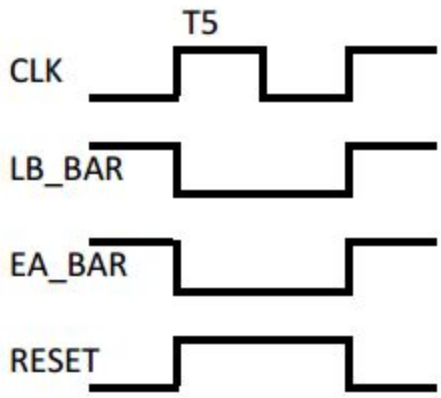
Timing diagram for RET instruction:



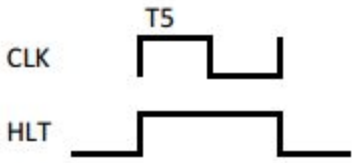
Timing diagram for OUT instruction:



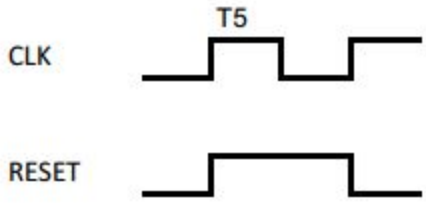
Timing diagram for MOV B, ACC instruction:



Timing diagram for HLT instruction:



Timing diagram for NOP instruction:



Timing diagram for NOP

Explanation of all blocks:

Input Register (IN):

Input register is used to take user input into account. At any moment, the user input is stored in a buffer.

This 4-bit Input register interacts with the BUS.

EIN_BAR is the associated signal. When EIN_BAR gets a low signal, the output of IN register is passed to the W0-W3 lines.

Program Counter (PC):

Program counter, also known as instruction pointer, is a register that contains the address of the instruction being executed at the current time. As each instruction is fetched, PC is incremented to contain the next instruction address. Instructions are usually fetched sequentially from memory, but some control instructions (namely JMP, CALL, RET) change the sequence by placing a new value in the PC.

PC can be only interacted through bus. PC can give the address of the instruction to bus only and the address of control instruction by loading into itself through bus. In our implementation, PC can interact with MAR MDR, TEMP register indirectly through bus.

CP is used to increment PC value after each instruction is fetched. EP_BAR is used to give address to bus and LP is used to load the address of the control instructions to PC. For example, when a CALL function is executed, the next instruction address is stored at SP and operand address (function address) of CALL is stored at TEMP register. After saving next instruction address to SP, data from TEMP register is loaded to PC enabling LP.

Memory Address Register (MAR):

Memory Address Register (MAR) is an 8-bit register. To read or write content from or to Memory at a particular address, we need to point at the address. This purpose is served by this register.

This register is connected with the W-Bus and the Memory. It uses lines W0-W7 to connect with the W-Bus and lines MO0-MO7 to connect with the memory. It gets the address from the W-Bus through lines W0-W7 and passes this address to the memory through lines MO0-MO7. For these reasons it uses 4-bit buffers.

EM_BAR and LM_BAR are the associated control signals both of which are active low. That means if we set EM_BAR to low, we can enable this register. And when we set LM_BAR to low, we can load address into this register.

RAM:

We can write content to and read content from RAM.

During the boot-loading period, instructions (hex-code for opcode and operand) is loaded into RAM. And, during the run of the PC, RAM is frequently accessed for instruction fetch, data store, data load etc.

RAM has the following associated signals.

WR_BAR, R_BAR, CLK.

RAM has the following input lines, MO0-MO7, these lines connect RAM and MAR. These lines are used to provide address to RAM. And, bidirectional data line, RM0-RM7. These lines connect MDR and RAM, using these lines either data

is fed into RAM using the W_BAR signal, or Data is read from RAM using the R_BAR signal.

To write content to RAM at a particular address, we must point at the interested address of RAM using MAR, we provide data in RM0-RM7 lines via MDR,then we control W_BAR signal to write the data. We must apply 1, 0, 1 logic level to W_BAR pin while providing valid data on RM0-RM7 to successfully complete a write operation.

To read content from RAM at a particular address, we point that address using MAR, then we control the R_BAR signal to read content from RAM, later on this data is sent to MDR.

Boot loader:

Boot loader resides inside the MEMORY module. It is responsible for loading program data from a ROM to RAM during boot period.

Boot loader is consisted of two ROMs, and two counters. One of the ROMs contain the instructions and data and a special sentinel value FF, which is used to terminate the boot-loading process. The second ROM contains control words to drive the boot-loading process.

Using one counter, we provide same address to both the program ROM and RAM, so that content in ROM at a particular address can be sent to that exact address in RAM.
Using the second counter, we drive the boot control ROM.

Basically, using the boot control ROM, we execute the write cycle of RAM, and content from ROM to RAM is transferred one byte at a time. When the final line in the program ROM is reached, boot-loading process is terminated and a special

signal BT_BAR is generated to let other key components know that boot has completed and PC can run now.

Memory Data Register (MDR):

MDR is an 8 bit register and it is used to read and write data to and from RAM, and it also relays RAM data to other registers.

MDR has bidirectional data line RM0-RM7. These lines connect MDR and RAM. These lines are used to read data from RAM to MDR and write MDR data to RAM. Given the appropriate signals, MDR can also load data from, and provide data to the BUS.

MEM_BUS, LDR_BAR, EDR_BAR, EDB_BAR are the associated signals with MDR.

The signal MEM_BUS is used in a multiplexer, to decide whether to load data from RAM or load data from BUS.

When MEM_BUS is LOW, data is loaded to MDR from RAM. This is used to read data from RAM.

When MEM_BUS is HIGH, data is loaded to MDR from the BUS.

When the LDR_BAR is LOW data is loaded to MDR either from RAM or BUS, depending on the MEM_BUS signal.

When EDB_BAR is LOW data from the MDR is sent to BUS via W0-W7.
When EDR_BAR is LOW data from the MDR is sent to RAM via RM0-RM7.

Instruction Register (IR):

Instruction Register is used to store the fetched Instruction from physical memory/RAM.

This 8-bit register receives instruction's opcode from memory through MDR register via BUS. This register holds the instruction's opcode and then passes the opcode to the control sequencer.

LIR_BAR is the associated signal. Setting LIR_BAR to low, we can load content from MDR via W0-W7 to this register. Least significant 5 bit of this value is available to the Control register via CON0-CON4 bus.

Controller-Sequencer:

The controller-sequencer unit produces the control words for microinstructions that coordinate and direct the rest of the computer. The control word or microinstruction determines how the registers react to the next positive clock edge.

This supervisor unit contains two types of ROM, namely address ROM and control ROM. The control ROM contains the control word for each micro-instruction in order to execute a macro-instruction. The starting address of execution cycle of each macro-instruction is listed in address ROM. The index of address ROM is the op-code of a macro-instruction. We collect the op-code bits $CON_4CON_3CON_2CON_1CON_0$ from the instruction register. These bits drive the address ROM and starting address of that particular routine is generated. Since our control word is 39-bit length, we need five control ROMs. One control word for any micro-instruction is listed in the same index of those ROMs. The outputs of the control ROMs are the outputs of this block.

We use an internal counter to generate the required indices for control ROM. After getting *BT_BAR* as low, i.e., boot loading is done, this counter generates zero. In the zeroth index, the control word for first micro-instruction of fetch cycle is written. Thus, the corresponding signals are generated and the PC value is transferred to MAR. Similarly, for each count, the rest of the micro-instructions of fetch cycle are executed. After three clocks, the op-code for a macro-instruction is available in the input of the address ROM. At the last micro-instruction of fetch cycle, we generate a special signal named as *"LOAD"*, which loads the content of

Instruction Register, i.e., op-code of that macro-instruction to the counter. On the next clock, the counter starts counting from that address, which is the starting address of that macro-instruction's execution routine. Thus, sequentially the rest of the micro-instructions of that routine are executed, i.e., the control words are generated, which drive the rest of the computer. At the last micro-instruction of execution cycle, we generate a special signal named as *"RESET"*, which resets the counter. Hence, on the next clock the zeroth index's content are generated from the control ROM that means the fetch cycle is started again. Thus, another macro-instruction's execution is started immediately after the first one. Since, we use *RESET* signal to reset the internal counter for every execution routine, we do not need to waste a single clock. Hence, we have variable machine cycle. Thus, we avoid the hardware complexity by micro-programming through the *RESET* and *LOAD* signals.

In order to execute conditional jump such as *JNE* and *JO,* we propagate *Zero Flag* and *Overflow Flag* to the controller. When the op-code of any of them is available in the input of address ROM, we propagate the content of the required flag to the flag input bits, i.e., CON_6 or CON_5. Depending on the content of the required flag bit, the counter jumps or not. We ensure *GND* to another flag input bit by using a simple combination circuit. Besides, except those conditional jumps, we ensure *GND* in those flag input bits by using a multiplexer.

Accumulator Register (ACC):

Accumulator register (ACC) is one of the most used blocks of 4-bit PC. It is a 4-bit register. It is used to perform data related operations. It can store and provide with data when necessary.

This register is connected with the W-Bus using the bidirectional lines W0-W3. It is also connected with the ALU using lines A0-A3. Using the bidirectional lines W0-W3 it can read or send data from or through W-Bus. It can also pass data to the ALU using the lines A0-A3. While taking such actions it uses 4-bit buffers also.

LA_BAR and EA_BAR are the control signals for this register both of which are active low. By setting EA_BAR to low, we can provide register data to the BUS. And when we set LA_BAR to low, we can load data into this register.

Arithmetic Logic Unit (ALU):

This asynchronous unit performs the required arithmetic as well as logical operations of our micro-computer.

We have used *74LS181* IC as an ALU. It has five control bits to determine the arithmetic or logic operation performed on words *A* and *B*. Here, word A comes from Accumulator Register and B comes from Temporary Register. The output of ALU is provisioned to go to the *W-BUS* by enabling *EU_BAR* signal.

We cannot perform ADC and *SBB* instructions easily by mode selector bits and carry in (*CN)* bit, since, it only uses the content of carry flag, not constant logic *one*. Besides, the datasheet of that IC shows that, in order to execute *SUB*, we provide logic *one* to the *CN* bit. In summary, there is no ready-made operation by which *ADC, SUB,* and *SBB* operations can be performed. Hence, we generate the following function table that relates the inputs of the *CN* bit of ALU to the external input signals. We need to provide two signals for controlling the carry such as *CIN1* and *CIN0*. This function table yields a combinational circuit equation, i.e., *CIN1'. CIN0. CF + CIN1. CIN0' + CIN1. CF'.*

CIN1	CIN0	CN (ALU input)	Required Operation
0	0	0	ADD
0	1	Content of carry flag (CF)	ADC
1	0	1	SUB
1	1	Inverted Content of carry flag (CF')	SBB

In order to keep track of a changing condition during a computer run, we use a flip-flop and a register named as flag register. We store carry flag, sign flag, overflow flag, and zero flag. Except overflow flag, rest of the flags are readily available as ALU's output. To determine the overflow flag, we use simple intuition, that is, an overflow can only occur when two numbers added are both positive or both negative. We test the inputs' and output's sign flag for determining the overflow condition.

We know that the contents of flag register are changed only in arithmetic operations. We ensure it by the load signal of flag flip-flop, i.e., *LF_BAR*. Besides, only addition operation can change the carry. We prevent the carry contents from unwanted changes by using another flip-flop. Besides, in order to execute CMC, i.e., complement the carry flag, we use a multiplexer, which is enabled by a *CMC* signal.

Temporary Register (TEMP):

Temporary register (TEMP) can be used to store both data and address whichever is needed in various operations. It is an 8-bit register.

This register interacts with the W-Bus and the ALU. With the bidirectional lines W0-W7 it keeps contact with the W-Bus. It is connected with the ALU with the lines T0-T3. With the lines W0-W7 it can receive or send data or address information through the W-Bus. It passes data to the ALU using lines T0-T3. It also uses 4-bit buffers to perform these actions.

LT_BAR and ET_BAR are the associated control signals which are active low. When ET_BAR is set to LOW, data is provided to the BUS. When LT_BAR is low data or address is loaded into this register.

B Register (B):

B register is used to store data operands for various computations.

This 4-bit register interacts with the BUS. This register can both load data from, and provide data to the BUS, depending on its input control signals.

LB_BAR and EB_BAR are the associated signals. Setting LB_BAR to low, we can load content from W0-W3 to B register and setting EB_BAR to low, we can load data to W0-W3.

Stack Pointer (SP):

Stack pointer is a register that stores the address of the last program request in a stack. A stack is a specialized memory segment that stores data in last in first out manner. The most recently entered request resides at the top of the stack and the program always takes request from the top.

. At the starting of boot loader, we initialize stack pointer with FF to point last address of RAM. When a PUSH instruction is requested, SP is decremented and then this SP is loaded to the MAR to point new memory location. As SP always holds the recent request, when a POP instruction is requested, value of SP is loaded to MAP without increment and decrement.

When a function is called by CALL instruction, then the next instruction address is stored on the SP. After returning from this function by RET instruction, SP value is loaded to MAR to execute the instruction following function CALL instruction.

ES_BAR is used to load the data from SP to bus. BT_BAR is used to initialize SP at the starting of boot loader and it is initialized to FF to point last memory location. IS and DS is used to increment and decrement the value of SP respectively.

Output Register (OUT):

Output register can be used to display results of different computation, for instance by adding a hex-output converter with it or LEDs.

This 4-bit output register interacts with the BUS.

LO_BAR is the associated signal.

When LO_BAR gets a low signal, the output register loads the content of W0-W3.

Control Word:

HLT - Halt
CMC - Complement Carry Flag
LF_BAR - Load Flag

M - Mode selector for ALU
S3 - ALU operation selection
S2 - ALU operation selection
S1 - ALU operation selection

S0 - ALU operation selection
CIN1 - Selector for CIN
CIN0 - Selector for CIN
EF_BAR - Enable Flag

EU_BAR - Enable Accumulator Register
LS - Load Stack
IS - Increment Stack
DS - Decrement Stack

ES_BAR - Enable Stack
LT_BAR - Load Temp Register
ET_BAR - Enable Temp Register
LO_BAR - Load Output Register

EB_BAR - Enable B register
LB_BAR - Load B Register
EA_BAR - Enable Accumulator Register
LA_BAR - Load Accumulator Register

EIN_BAR - Enable Input Port
LIR_BAR - Load Instruction Register
MEM_BUS - Memory Bus
LDR_BAR - Load Data Register

EDR_BAR - Enable Data Register
EDB_BAR - Enable Data Bus
W_BAR - Write RAM
R_BAR - Read RAM

LM_BAR - Load MAR
EM_BAR - Enable MAR
CLEAR
CP - Counter for PC

EP_BAR - Enable PC
LP - Load PC
LOAD
RESET

Explanation of all instructions:

Here, MO stands for Micro operation and HLO stands for High Level Overview.

MO: Micro operations. Each minimal operation(done in one T state) needed to execute an instruction(macro) .

All the instructions share the fetch cycle. The fetch cycle has the following micro operations. Maximum number of Micro operations determine the maximum number of T states needed.

Common micro-operations for all instructions:
Fetch cycle:
1st cycle: load PC value to MAR
2nd cycle: RAM to MDR and PC= PC + 1
3rd cycle: MDR to IR

Micro operations for individual instructions(execution cycle):

1. LDA address
Opcode: 00H
HLO: ACC←Memory[Address]

MO: i) MAR ← Address[operand]
ii) ACC ← MDR
4th cycle:load load PC value to MAR
5th cycle: load RAM value to MDR and PC++
6th cycle: load MDR to MAR
7th cycle: load RAM value to MDR
8th cycle: load MDR data to Accumulator

2. STA address
Opcode: 01H
HLO: Memory[Address] ← ACC

MO: 4) load PC to MAR
5) load RAM to MDR, Increment PC //MDR holds the operand
6) load MDR to MAR
7) load ACC to MDR
8) Write MDR data to RAM

3. MOV Acc, B
Opcode: 05H
HLO: ACC ← B
MO: 4th cycle: load data from B to Accumulator
4. MOV B, Acc
Opcode: 04H
HLO: B← ACC
MO:
4th cycle: load Accumulator to B

5. MOV Acc, immediate
Opcode: 07H
HLO: ACC← immediate
MO:
4th cycle: load PC value to MAR
5th cycle: load RAM to MDR and PC++
6th cycle: load MDR to Accumulator

6. IN
Opcode: 02H
HLO: ACC← Input port.
MO: 4th cycle: load Input port to Accumulator..

7. OUT
Opcode: 03H
HLO: Output port← ACC.
MO: 4th cycle: Load Accumulator to Output Port.

8. ADD B
Opcode: 0AH
HLO: ACC ← ACC+ B
MO:
4th cycle: Load register B to TEMP_REGISTER
5th cycle: Select ALU Operation for ADD
6th cycle: Load ALU output to Accumulator.

9. ADC B

Opcode: 0BH

HLO: ACC← ACC+ B + C (Carry)

MO:

4th cycle: Load register B data to TEMP_REGISTER

5th cycle: Select ALU Operation for ADC

6th cycle: Load ALU output to Accumulator.

10. SUB B

Opcode: 0CH

HLO: ACC← ACC- B

MO:

4th cycle: load B data to TEMP_REGISTER

5th cycle: select ALU operation for SUB

6th cycle: Load ALU output to Accumulator.

11. SBB B

Opcode: 0DH

HLO: ACC← ACC - B -Bo(Borrow)

MO:

4th cycle: load B data to TEMP_REGISTER

5th cycle: select ALU operation for SBB

6th cycle: Load ALU output to Accumulator.

12. ADD immediate

Opcode: 0EH

HLO: ACC← ACC + immediate

MO: Cycle 4: Load PC to MAR

Cycle 5: Write RAM data to MDR ,Increment PC
Cycle 6: Load MDR to TEMP REGISTER
Cycle 7: Select ALU operation for ADD
Cycle 8: Load ALU output to Accumulator.

13. SUB immediate
Opcode: 10H
HLO: ACC← ACC - immediate
MO: Cycle 4: Load PC to MAR
Cycle 5: Write RAM data to MDR ,Increment PC
Cycle 6: Load MDR data to TEMP REGISTER
Cycle 7: Select ALU operation for SUB
Cycle 8: Load ALU output to Accumulator.

14. CMP B
Opcode: 11H
HLO: ACC not change, but flags will be changed according to (ACC-B)
MO:
4th cycle: load B to TEMP_REGISTER
5th cycle: select ALU operation for SUB (no load is needed to Accumulator)

15. NEG
Opcode: 12H
HLO: ACC← -ACC

MO:
4th cycle: Invert Accumulator data and Load this value to Accumulator.
5th cycle: Add 1 with Accumulator data and load this value to Accumulator.

16. JO address
Opcode: 5BH
HLO: Jump to address when overflow flag is 1.
MO: If Overflow occurs,
  4th cycle: load PC to MAR
  5th cycle: load RAM data to MDR and PC+=1
  6th cycle: load MDR to PC

17. JNE address
Opcode: 3AH
HLO: Jump to address when zero flag is 0.
MO: If zero flag is 0,
  4th cycle: load PC to MAR
  5th cycle: load RAM data to MDR and PC+=1
  6th cycle: load MDR data to PC

18. PUSH
Opcode: 15H
HLO: STACK← ACC
MO: 4th cycle: Decrement SP and load accumulator to MDR.
  5th cycle: Load SP to MAR
  6th cycle: Write MDR data to RAM location

19. POP

Opcode: 16H

HLO: ACC← STACK

MO:

4th cycle: load SP to MAR

5th cycle: load RAM value to MDR

6th cycle: increment SP and load MDR to Accumulator

20. CALL address

Opcode: 17H

HLO: Calls a subroutine(at the specified address) unconditionally.

i) Save next_instruction address to stack

ii) Jump to the operand address

MO:

4th cycle: Load PC to MAR

5th cycle: Write RAM data to MDR and increment PC

6th cycle: Load MDR data to TEMP_REGISTER and decrement SP

7th cycle: Load SP to MAR

8th cycle: Load PC to MDR

9th cycle: Write MDR data to RAM

10th cycle: Load TEMP REGISTER to PC

21. RET

Opcode: 18H

HLO: Returns from current subroutine to the caller unconditionally.

MO:

4th cycle: Load SP to MAR

5th cycle: Write RAM data to MDR

6th cycle: Load MDR to PC and Increment SP

22. JMP address

Opcode: 09H

HLO: Jumps unconditionally to address.

MO:

4th cycle: Load PC to MAR

5th cycle: Write RAM data to MDR, increment PC

6th cycle: Load MDR to PC

23. HLT

Opcode: 0FH

HLO: Halts execution.

MO: 4th cycle: activate halt pin to stop the clock.

24. NOP

Opcode: 08H

MO:

4th cycle: Nothing else to do, return to the start of fetch cycle.

25. XCHG

Opcode: 06H

HLO: ACC ←→ B (Exchanges contents of ACC and B)

MO:

4th cycle: Load register B to TEMP_REGISTER

5th cycle: Load Accumulator to B

6th cycle: Load TEMP_REGISTER to Accumulator

26. CMC

Opcode: 19H

HLO: Complements the carry flag.

MO:

4th cycle: Flip the carry flag.

27. OR [address]

Opcode: 13H

HLO: ACC← ACC | Memory[Address]

MO:

4th cycle: Load PC to MAR

5th cycle: Write RAM to MDR, increment PC

6th cycle: Load MDR to MAR

7th cycle: Write RAM to MDR

8th cycle: Load MDR to TEMP_REGISTER

9th cycle: SELECT ALU operation OR and Load ALU output to Accumulator

28. XOR B

Opcode: 14H

HLO: ACC← ACC ⊕ B

MO:

4th cycle: Load B data to TEMP_register

5th cycle: Select ALU operation for XOR and Load ALU output to Accumulator

Explanation of all control signals:

Cycle Description:

Macro-Instruction	Op-Code	Description	Total T-States	T-State	Micro-Operation	Active	CON				
ALL	-	Fetch	4	T1	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T2	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98

				T3	IR ← MDR	EDB_BAR, LIR_BAR	10	18	FF	9B	C8
				T4	LOAD from IR	LOAD	10	18	FF	DF	CA
LDA address	00H	ACC ← RAM[address]	9	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	MAR ← MDR	LM_BAR, EDB_BAR, EM_BAR	10	18	FF	DB	8
				T8	MDR ← RAM[MAR]	LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	88
				T9	ACC ← MDR	LA_BAR, EDB_BAR, RESET	10	18	FE	DB	C9
HALT	0FH	Halts execution	5	T5	HALT	HLT	50	18	FF	DF	C8
IN	02H	ACC ← input_port	5	T5	ACC ← input_port	LA_BAR, EIN_BAR, RESET	10	18	FE	5F	C9
STA address	01H	RAM[address] ← ACC	12	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	MAR ← MDR	LM_BAR, EDB_BAR, EM_BAR	10	18	FF	DB	8
				T8	MDR ← ACC	LDR_BAR, EA_BAR, MEM_BUS, EM_BAR	10	18	FD	EF	88
				T9	RAM[MAR] ← MDR	EDR_BAR, EM_BAR	10	18	FF	D7	88

				T10		W_BAR, EDR_BAR, EM_BAR	10	18	FF	D5	88
				T11		EDR_BAR, EM_BAR	10	18	FF	D7	88
				T12		R_BAR, RESET, EM_BAR	10	18	FF	DE	89
OUT	03H	output_port ← ACC	5	T5	output_port ← ACC	LO_BAR, EA_BAR, RESET	10	18	ED	DF	C9
MOV B, ACC	04H	B ← ACC	5	T5	B ← ACC	LB_BAR, EA_BAR, RESET	10	18	F9	DF	C9
MOV ACC, B	05H	ACC ← B	5	T5	ACC ← B	LA_BAR, EB_BAR, RESET	10	18	F6	DF	C9
XCHG	06H	ACC ↔ B	7	T5	TEMP ← B	LT_BAR, EB_BAR	10	18	B7	DF	C8
				T6	B ← ACC	LB_BAR, EA_BAR	10	18	F9	DF	C8
				T7	ACC ← TEMP	LA_BAR, ET_BAR, RESET	10	18	DE	DF	C9
MOV ACC, immediate	07H	ACC ← immediate	7	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	ACC ← MDR	LA_BAR, EDB_BAR, RESET	10	18	FE	DB	C9
NOP	08H	No Operation	5	T5	NOP	NOP, RESET	10	18	FF	DF	C9
JMP address	09H	Jump to the address	8	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	PC ← MDR	LP, EDB_BAR	10	18	FF	DB	CC

				T8		CP, LP, RESET	10	18	FF	DF	DD
ADD B	0AH	ACC ← ACC + B	7	T5	TEMP ← B	LT_BAR, EB_BAR, S3, S0	14	98	B7	DF	C8
				T6	ACC ← ACC + TEMP	S3, S0	14	98	FF	DF	C8
				T7		LA_BAR, EU_BAR, S3, S0, LF_BAR, RESET	4	90	FE	DF	C9
ADC B	0BH	ACC ← ACC + B + C	7	T5	TEMP ← B	LT_BAR, EB_BAR, S3, S0, CIN0	14	B8	B7	DF	C8
				T6	ACC ←ACC + TEMP + C	S3, S0, CIN0	14	B8	FF	DF	C8
				T7		LA_BAR, EU_BAR, S3, S0, CIN0, LF_BAR, RESET	4	B0	FE	DF	C9
SUB B	0CH	ACC ← ACC - B	7	T5	TEMP ← B	LT_BAR, EB_BAR, S2, S1, CIN1	13	58	B7	DF	C8
				T6	ACC ← ACC - TEMP	S2, S1, CIN1	13	58	FF	DF	C8
				T7		LA_BAR, EU_BAR, S2, S1, CIN1, LF_BAR, RESET	3	50	FE	DF	C9
SBB B	0DH	ACC ← ACC - B - BO	7	T5	TEMP ← B	LT_BAR, EB_BAR, S2, S1, CIN1, CIN0	13	78	B7	DF	C8
				T6	ACC←ACC - TEMP - BO	S2, S1, CIN1, CIN0	13	78	FF	DF	C8
				T7		LA_BAR, EU_BAR, S2, S1, CIN1, CIN0, LF_BAR, RESET	3	70	FE	DF	C9
ADD immediate	0EH	ACC ← ACC + immediate	9	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98

				T7	TEMP ← MDR	LT_BAR, EDB_BAR, S3, S0	14	98	BF	DB	C8
				T8	ACC ← ACC + TEMP	S3, S0	14	98	FF	DF	C8
				T9		LA_BAR, EU_BAR, S3, S0, LF_BAR, RESET	4	90	FE	DF	C9
SUB immediate	10H	ACC ← ACC - immediate	9	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	TEMP ← MDR	LT_BAR, EDB_BAR, S2, S1, CIN1	13	58	BF	DB	C8
				T8	ACC ← ACC - TEMP	S2, S1, CIN1	13	58	FF	DF	C8
				T9		LA_BAR, EU_BAR, S2, S1, CIN1, LF_BAR, RESET	3	50	FE	DF	C9
CMP B	11H	ACC - B	7	T5	TEMP ← B	LT_BAR, EB_BAR, S2, S1, CIN1	13	58	B7	DF	C8
				T6	ACC - TEMP	S2, S1, CIN1	13	58	FF	DF	C8
				T7		S2, S1, CIN1, LF_BAR, RESET	3	58	FF	DF	C9
NEG	12H	ACC ← - ACC	6	T5	ACC ← ACC'	LA_BAR, EU_BAR, M	18	10	FE	DF	C8
				T6	ACC ← ACC' + 1	LA_BAR, EU_BAR, S3, S2, S1, S0, CIN1, RESET	17	D0	FE	DF	C9
OR [address]	13H	ACC ← ACC | RAM[address]	10	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0

				T6	PC⟵ PC+1, MDR ⟵ RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	MAR ⟵ MDR	LM_BAR, EDB_BAR, EM_BAR	10	18	FF	DB	8
				T8	MDR ⟵ RAM[MAR]	LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	88
				T9	TEMP ⟵ MDR	LT_BAR, EDB_BAR	10	18	BF	DB	C8
				T10	ACC ⟵ ACC | TEMP	LA_BAR, EU_BAR, M, S3, S1, S0, RESET	1D	90	FE	DF	C9
XOR B	14H	ACC ⟵ ACC ⊕ B	6	T5	TEMP ⟵ B	LT_BAR, EB_BAR	10	18	B7	DF	C8
				T6	ACC ⟵ ACC ⊕ TEMP	LA_BAR, EU_BAR, M, S3, S0, RESET	1C	90	FE	DF	C9
PUSH	15H	RAM[SP] ⟵ ACC	10	T5	SP ⟵ SP - 1, MDR ⟵ ACC	DS, LDR_BAR, EA_BAR, MEM_BUS, EM_BAR	10	19	FD	EF	88
				T6	MAR ⟵ SP	LM_BAR, ES_BAR, EM_BAR	10	18	7F	DF	8
				T7	RAM[MAR] ⟵ MDR	EDR_BAR, EM_BAR	10	18	FF	D7	88
				T8		W_BAR, EDR_BAR, EM_BAR	10	18	FF	D5	88
				T9		EDR_BAR, EM_BAR	10	18	FF	D7	88
				T10		R_BAR, RESET, EM_BAR	10	18	FF	DE	89
POP	16H	ACC ⟵ RAM[SP]	7	T5	MAR ⟵ SP	LM_BAR, ES_BAR, EM_BAR	10	18	7F	DF	8
				T6	MDR ⟵ RAM[MAR]	LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	88

				T7	ACC ← MDR, SP ← SP + 1	LA_BAR, EDB_BAR, IS, RESET	10	1A	FE	DB	C9
CALL address	17H	Calls a subroutine	14	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	TEMP ← MDR, SP ← SP - 1	LT_BAR, EDB_BAR, DS	10	19	BF	DB	C8
				T8	MAR ← SP	LM_BAR, ES_BAR	10	18	7F	DF	48
				T9	MDR ← PC	LDR_BAR, EP_BAR, MEM_BUS	10	18	FF	EF	C0
				T10	RAM[MAR] ← MDR	EDR_BAR, EM_BAR	10	18	FF	D7	88
				T11		W_BAR, EDR_BAR, EM_BAR	10	18	FF	D5	88
				T12		EDR_BAR, EM_BAR	10	18	FF	D7	88
				T13	PC ← TEMP	LP, ET_BAR	10	18	DF	DF	CC
				T14		CP, LP, RESET	10	18	FF	DF	DD
RET	18H	Returns from current subroutine	8	T5	MAR ← SP	LM_BAR, ES_BAR	10	18	7F	DF	48
				T6	MDR ← RAM[MAR]	LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	88
				T7	PC ← MDR, SP ← SP + 1	LP, EDB_BAR, IS	10	1A	FF	DB	CC
				T8		CP, LP, RESET	10	18	FF	DF	DD
CMC	19H	CF ← CF'	5	T5	CF ← CF'	CMC, LF_BAR, RESET	20	18	FF	DF	C9

JNE address	1AH	Jump if not equal	8	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	PC ← MDR	LP, EDB_BAR	10	18	FF	DB	CC
				T8		CP, LP, RESET	10	18	FF	DF	DD
JO address	1BH	Jump if overflow	8	T5	MAR ← PC	LM_BAR, EP_BAR, EM_BAR	10	18	FF	DF	0
				T6	PC← PC+1, MDR ← RAM[MAR]	CP, LDR_BAR, R_BAR, EM_BAR	10	18	FF	CE	98
				T7	PC ← MDR	LP, EDB_BAR	10	18	FF	DB	CC
				T8		CP, LP, RESET	10	18	FF	DF	DD

How to Write and Execute a program in this computer:

Writing:

Writing a program for this computer can be done in two steps,

i) Writing in mnemonics form of instructions.
ii) Converting the mnemonic form to hex code. This is done by merging the hex op code of the instruction and the hex value of the operand(address or immediate value, if exists).

Each instruction will become a hex value representing 1 or 2 bytes of information, given whether they use address or immediate value operands or not.

Next, we arrange the hex values in a BIN file line by line such that each line has a two digit hex code(1 byte).

Each program must be terminated with the HALT instruction. Which has the hex opcode, F.

To denote the end of the program file, we use a special value of FF, we put this value at the last line of the program BIN file.

A sample program:
Mnemonic form:
IN
STA F3
LDA F3
HLT

Hex code formulation:
IN has opcode 2
STA has opcode 1
LDA has opcode 0,
So the program becomes,
02
01
F3
00
F3
0F

We store these hex codes in a BIN file. We add, FF at the final line to denote end of FILE.

We load this BIN file in the program ROM. When the PC starts, during the boot loader phase, each of these instructions from the program ROM is loaded into the RAM, afterwards during the fetch cycle, OP is fetched from the RAM and it is eventually sent to the instruction register and the execution phase starts.

IC used with Count:

IC Number	IC Name	Number of ICs used
74LS173	4 bit D-type register with 3 state output	15
74LS244	Octal 3 state buffer	37
74LS157	Quad 2-input Multiplexer	6
COUNTER_8	8 bit Binary up/down counter	4
74LS386	Quad 2-input Exclusive OR gate	1

74LS04	NOT gate	24
AND_4	4 input AND gate	2
AND_3	3 input AND gate	3
74LS32	OR gate	1
74LS181	4 bit Arithmetic Logic Unit	1
74LS08	Quad 2-input AND gate	6
OR_3	3 input OR gate	1
2732	EPROM	8
COUNTER_4	4 bit binary up/down counter	1
6116	CMOS Static RAM 16K(2K * 8 bit)	1
74HC21	Dual 4 input AND gate	2

Discussion:

We have completed implementing the simplified 4-bit PC with 28 instructions. This particular assignment from the Digital System Design lab has taken more than 8 days from our lives, has frustrated us at many points during the development cycle, it made us spend many sleepless nights. However it has enriched us with a deeper appreciation of the execution sequences of a computer in general. It was an euphoric experience to observe our PC successfully executing some programs(even with function calls).

During the development of the 4-bit PC, we have encountered some difficulties. Such as, we had made an assumption that given a valid address at ROM or RAM, we immediately get the content, which doesn't reflect reality. There is an invalid period during which even though there is a valid address at address line, we do not get the valid data at that address.

Another problem we faced was during working with memory write operation, by going through a lot of trial and error, we realized that, we need to provide valid data to RAM for three consecutive clock cycles, during which we give logic_high, logic_low, logic_high to the W_BAR signal for a successful write operation.

We faced problems with data contention while establishing communication between two components only to realize that if we contain the register value in buffers and do not pass it to BUS when it's not necessary, we will not have contention.

We have faced race conditions while implementing the ALU circuit, which was resolved using latches.

We have constructed the flag register in such a way that, its value remains unaffected during logical operations.