INTRODUCTION:

In this assignment, we constructed a 4-bit "Simple As Possible" (SAP) computer simulation using the Proteus Design Suite.

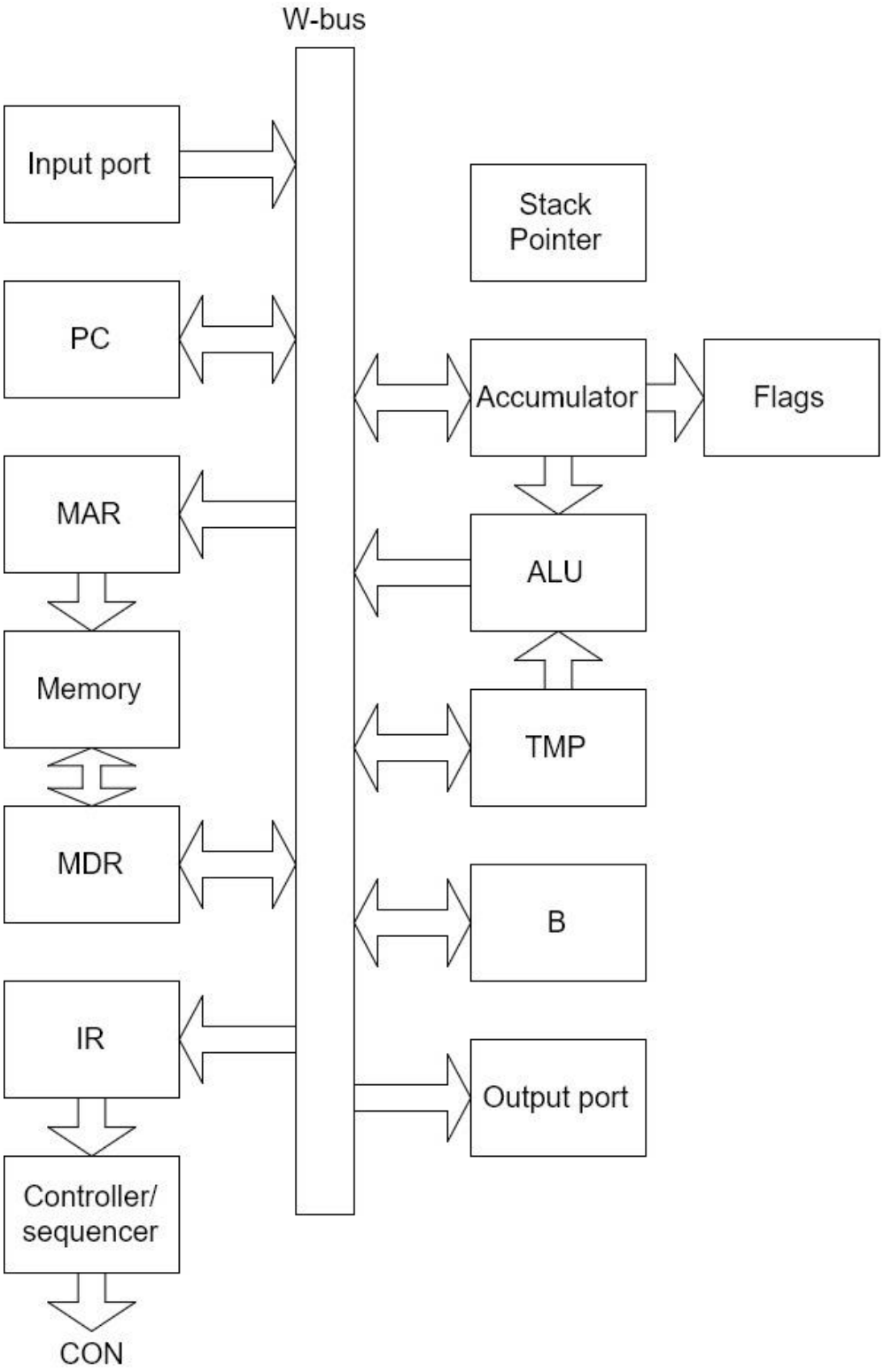
The 4-bit PC consists of a few registers – ACC (accumulator), MAR (Memory Address Register), MDR (Memory Data Register) etc.; counters - PC (Program Counter), SP (Stack Pointer) etc.; ROMs, a RAM and an Arithmetic Logic Unit (ALU).

This 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 the Proteus Design Suite.

INSTRUCTION SET:

MNEMONIC	DESCRIPTION
LDA address	Acc ← Memory [address]
STA address	Memory [address] ← Acc
MOV ACC, B	Acc ← B
MOV B, ACC	B ← Acc
MOV ACC, immediate	Acc ← immediate
IN	Acc ← Input port
OUT	Output port ← Acc
ADD B	Acc ← Acc + B
ADC B	Acc ← Acc + B + C (Contents of Carry Flag)
SUB B	Acc ← Acc – B
SBB B	Acc ← Acc – B – Bo (Contents of Carry Flag)
ADC immediate	Acc ← Acc + immediate + C (Contents of Carry Flag)
SBB immediate	Acc ← Acc – immediate – Bo (Contents of Carry Flag)
CMP B	Acc will be unchanged. Sets Flags according to (Acc – B)
XCHG	Exchanges contents of Accumulator and B
JC address	Jumps to the address if carry flag is set
JE address	Jump if equal
PUSH	Pushes the content of Accumulator to the stack
POP	Pops off stack to Accumulator
CALL address	Calls a subroutine (at the specified address) unconditionally
RET	Returns from current subroutine to the caller unconditionally
JMP	Jumps unconditionally to the address
HLT	Halts execution
NOP	No Operation
STZ	Sets the zero flag
CLZ	Clears the zero flag
AND immediate	Acc ← Acc . immediate
OR [address]	Acc ← Acc | Memory [address]

BLOCK DIAGRAM:

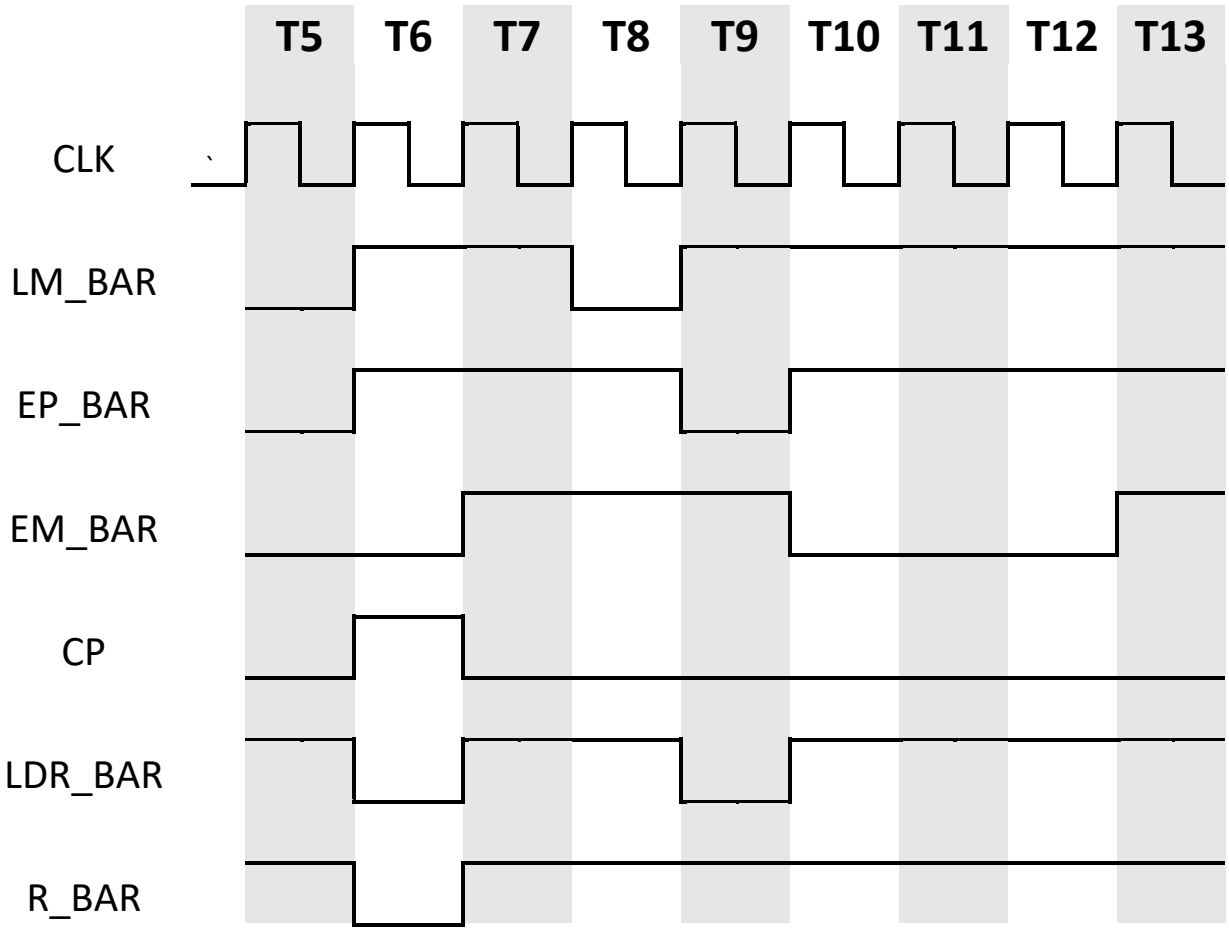


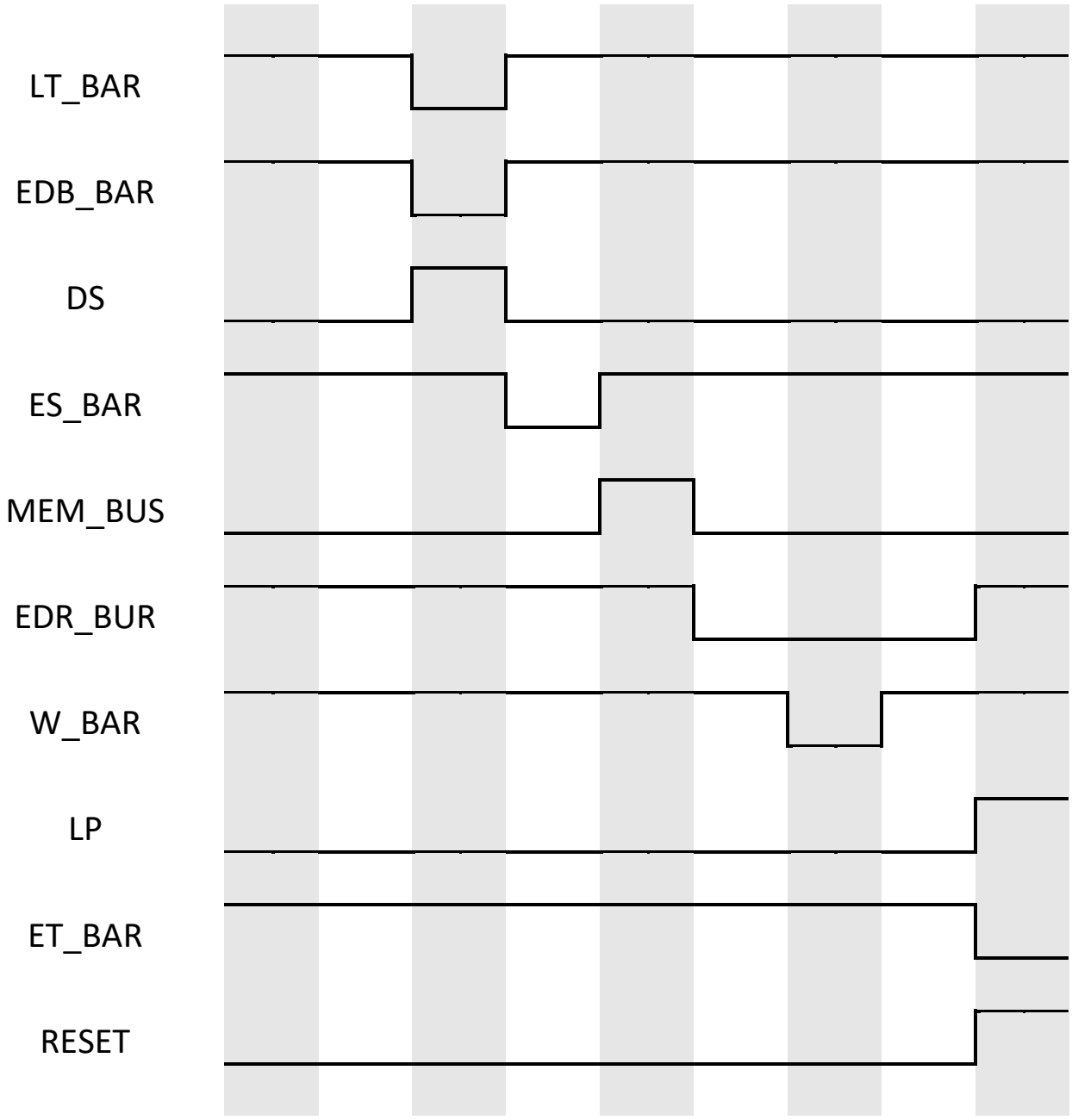
CIRCUIT DIAGRAM:

Complete circuit diagram of the project is printed on drawing paper and attached to this report.

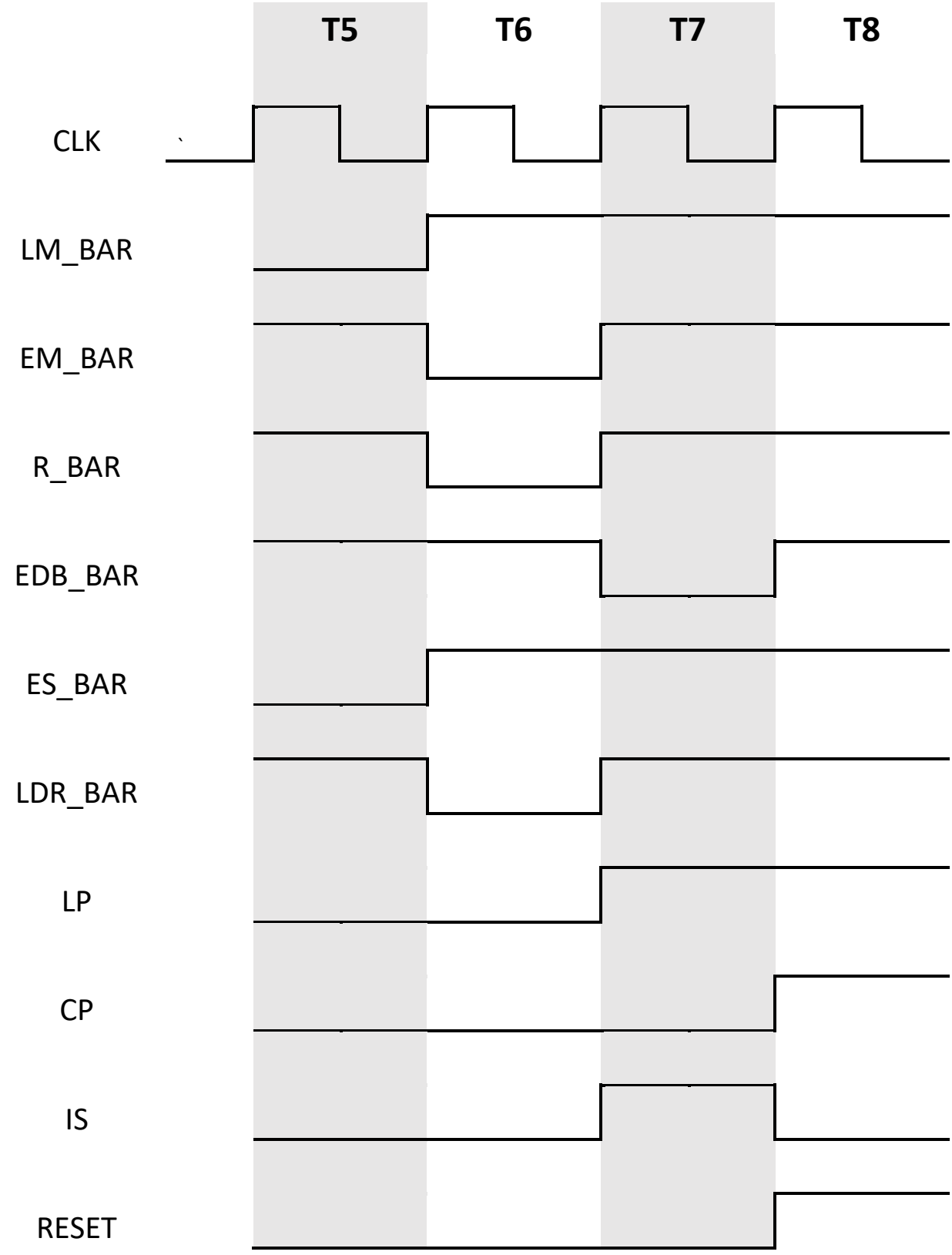
TIMING DIAGRAMS:

CALL INSTRUCTION:

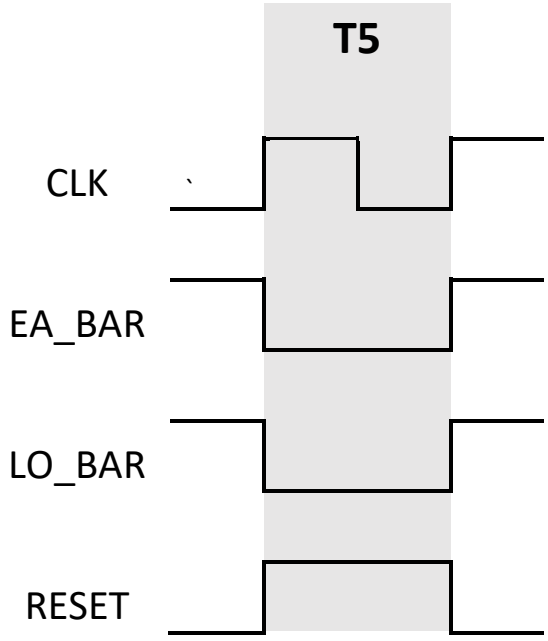



LT_BAR
EDB_BAR
DS
ES_BAR
MEM_BUS
EDR_BUR
W_BAR
LP
ET_BAR
RESET

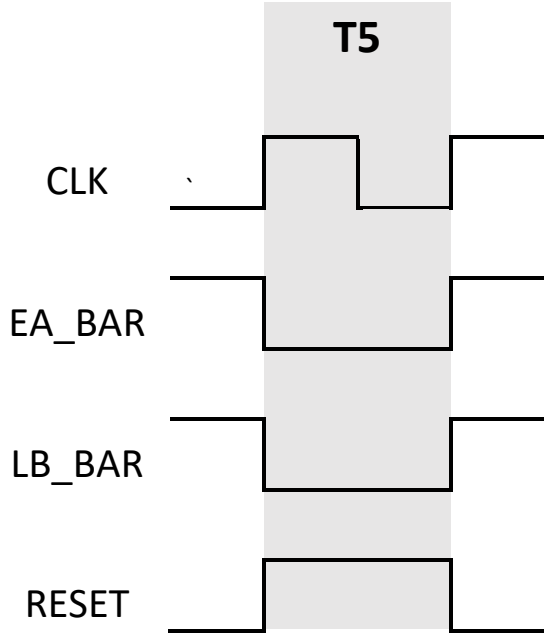
RET INSTRUCTION:



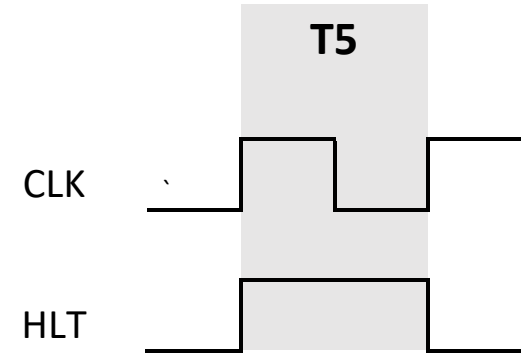
OUT INSTRUCTION:



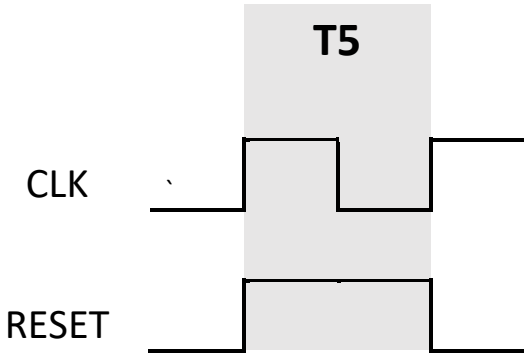
MOV B, Acc INSTRUCTION:



HLT INSTRUCTION:



NOP INSTRUCTION:



DESCRIPTION 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 zero'th 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 zeroeth 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:

CONTROL SIGNAL	ACTION PERFORMED
DATA_Z	Set/Clear the Z flag
HLT	Halt the machine
CHG_Z	Enable CLZ/STZ
LF_BAR	Load the Flags
M	ALU mode selection
S3	ALU operation selection
S2	ALU operation selection
S1	ALU operation selection
S0	ALU operation selection
CIN1	ALU carry selection
CIN0	ALU carry selection
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	Load the controller from IR
RESET	Start new fetch cycle

EXPLANATION OF ALL INSTRUCTIONS:

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.

MACRO-INSTRUCTION	OPCODE	T-STATE	MICRO-INSTRUCTION
FETCH	-	T1	MAR ← PC
		T2	PC← PC+1, MDR ← RAM[MAR]
		T3	IR ← MDR
		T4	LOAD from IR
LDA	0	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	MAR ← MDR
		T8	MDR ← RAM[MAR]
		T9	ACC ← MDR
STA	1	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	MAR ← MDR
		T8	MDR ← ACC
		T9	RAM[MAR] ← MDR
IN	2	T5	ACC ← Input port
OUT	3	T5	Output port ← ACC
MOV B, A	4	T5	B ← ACC
MOV A, B	5	T5	ACC ← B
XCHG	6	T5	TEMP ← B
		T6	B ← ACC
		T7	ACC ← TEMP

MOV A, immediate	7	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	ACC ← MDR
NOP	8	T5	NOP
JMP	9	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	PC ← MDR
ADD B	A	T5	TEMP ← B
		T6	ACC ← ACC + TEMP
ADC B	B	T5	TEMP ← B
		T6	ACC ← ACC + TEMP + C
SUB B	C	T5	TEMP ← B
		T6	ACC ← ACC - TEMP
SBB B	D	T5	TEMP ← B
		T6	ACC ← ACC - TEMP - BO
ADC immediate	E	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	TEMP ← MDR
		T8	ACC ← ACC + TEMP + C
HLT	F	T5	HALT
SBB immediate	10	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	TEMP ← MDR
		T8	ACC ← ACC - TEMP - C
CMP B	11	T5	TEMP ← B
		T6	ACC - TEMP
CLZ	12	T5	Z ← 0

OR [address]	13	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	MAR ← MDR
		T8	MDR ← RAM[MAR]
		T9	TEMP ← MDR
		T10	ACC ← ACC | TEMP
AND immediate	14	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	TEMP ← MDR
		T8	ACC ← ACC & TEMP
PUSH	15	T5	SP ← SP - 1, MDR ← ACC
		T6	MAR ← SP
		T7	RAM[MAR] ← MDR
POP	16	T5	MAR ← SP
		T6	MDR ← RAM[MAR]
		T7	ACC ← MDR, SP ← SP + 1
CALL	17	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	TEMP ← MDR, SP ← SP - 1
		T8	MAR ← SP
		T9	MDR ← PC
		T10	RAM[MAR] ← MDR
		T11	PC ← TEMP
RET	18	T5	MAR ← SP
		T6	MDR ← RAM[MAR]
		T7	PC ← MDR, SP ← SP + 1
STZ	19	T5	Z ← 1

JC address	3A	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	PC ← MDR
JE address	5B	T5	MAR ← PC
		T6	PC← PC+1, MDR ← RAM[MAR]
		T7	PC ← MDR

WRITING AND EXECUTING A PROGRAM:

WRITING:

We write programs in this PC in the following way:

- We convert the mnemonic form to hex code. This is done by merging the hex opcode 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 HLT instruction. Which has the hex opcode 0FH.

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	OPCODE	FINAL MACHINE CODE
		02
IN	IN has opcode 2	01
STA F3	STA has opcode 1	F3
LDA F3	LDA has opcode 0	00
HLT	HLT has opcode F	F3
		0F

EXECUTION:

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:

IC NUMBER	DESCRIPTION	NUMBER OF IC'S 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	22
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:

The 4-bit SAP Computer project, although interesting and educative, was also extremely abstruse, and onerous at the same time. We faced numerous issues – both technical and non-technical. And some of those issues are discussed below.

The first issue that we faced was with the design suite itself. “Proteus” is a premium commercial product, and the university did not provide us with a license to use this product. Since the software is fairly expensive, we attempted to use some open-source software/freeware instead; but none of them had the functionality required for the project. As there was no other option available to us, we were forced to use a patched version of the Proteus Design Suite. We find this act of piracy extremely unethical and regretful. Not only that, the patched version that we initially installed in our workstations was faulty, and caused the corruption and loss of valuable data, not to mention the loss of valuable time.

Also, it took a lot of time to understand the mechanism of the 4-bit computer itself. Many a times we had to start our work all over again because of incorrect actions and assumptions. Since this is not a conventional project, the resources and tutorials, both online and offline, were very scarce, too.

We also made a foolish mistake on the day of presenting the simulated computer. In all the programs to be executed, we forgot to add a one-liner code (namely, FF) to finish bootloading which needs to be appended to last of any program. For this reason, none of our programs executed successfully, resulting in our failure to demonstrate all the hard work we had done. All our sleepless nights’ efforts seemed to go in vain. This incident almost destroyed our morale, though we were given an opportunity of a late-submission. The mistake was very trifling, and the solution to it even more so. But the effect of the mistake was grave. We felt anguish and woe after we found out what a foolish mistake we had made which required virtually no debugging at all.

Nevertheless, this project helped us learn a lot, about the internal mechanism of a computer and about our own limitations, too. Most importantly, the project aroused our curiosity and developed our appreciation of computer hardware.
