Design and Simulation of a Pipelined 4 bit Computer

CSE 404N Digital System Design

A1:5

Bangladesh University of Engineering and Technology
Department of Computer Science and Engineering
Dhaka, Bangladesh

Submitted By :
Section A1 Group 5

Ishtiak Zaman	03 05 011
Md Tanvir Al Amin	03 05 012
Ashic Mahtab	03 05 018
Sukarna Barua	03 05 026
Zinat Wali	03 05 030

Table of Contents

1 Introduction

Our 4 bit computer system can execute 28 instructions. Each instruction requires only 1.4 clock cycles on average. The entire system consists of a 2 stage pipeline – instruction fetch & decode unit and 4 bit execution unit. The execution unit has a shared 4 bit data bus. The memory system is organized as Harvard architecture. That is instruction memory and data memories are separate units. Programs are stored in instruction memory and data in data memory. Data memory also contains the program stack. All instructions are either 1 or 2 bytes long. Some instructions require 1 clock cycle on average and others require 2 clock cycles on average. The computer can also communicate with the external devices through the input and output port registers. The instruction set contains all types of instructions which generalize the computer to execute any kind of program. However the 8 bit address bus restricts the program length to be within 256 Bytes. The 4 bit arithmetic and logic unit can perform addition and subtraction and several logical operations.

2 Block Diagram of Architecture and Control Unit

The entire processor system can be divided into two independent sections.

- Fetch and decode unit
- Execution unit

These two sections organized as a two staged pipeline processes instructions independently. While execution unit executes an instruction; at the same time fetch unit fetches an instruction into the instruction register.

- All data values are 4 bit. Although immediate values take out 1 byte of instruction memory (it is because instruction memory is byte addressable), the higher nibble contains all zeros, lower nibble contains the actual 4 bit data.

- There are separate memory for instructions and data. Program codes are loaded in instruction memory. All data writes are written to the data memory. Stack memory is also data memory.

- The execution unit has a shared 4 bit data bus. All components in the execution unit share this bus for transferring data. All outputs therefore from execution unit registers that are connected to the shared bus are 3 states so that one component does not load the bus while others are using it.

- All the memory address length is 8 bit. However this computer has no explicit address bus. Because we did not need to share this address bus.

Block diagram of 4 bit computer

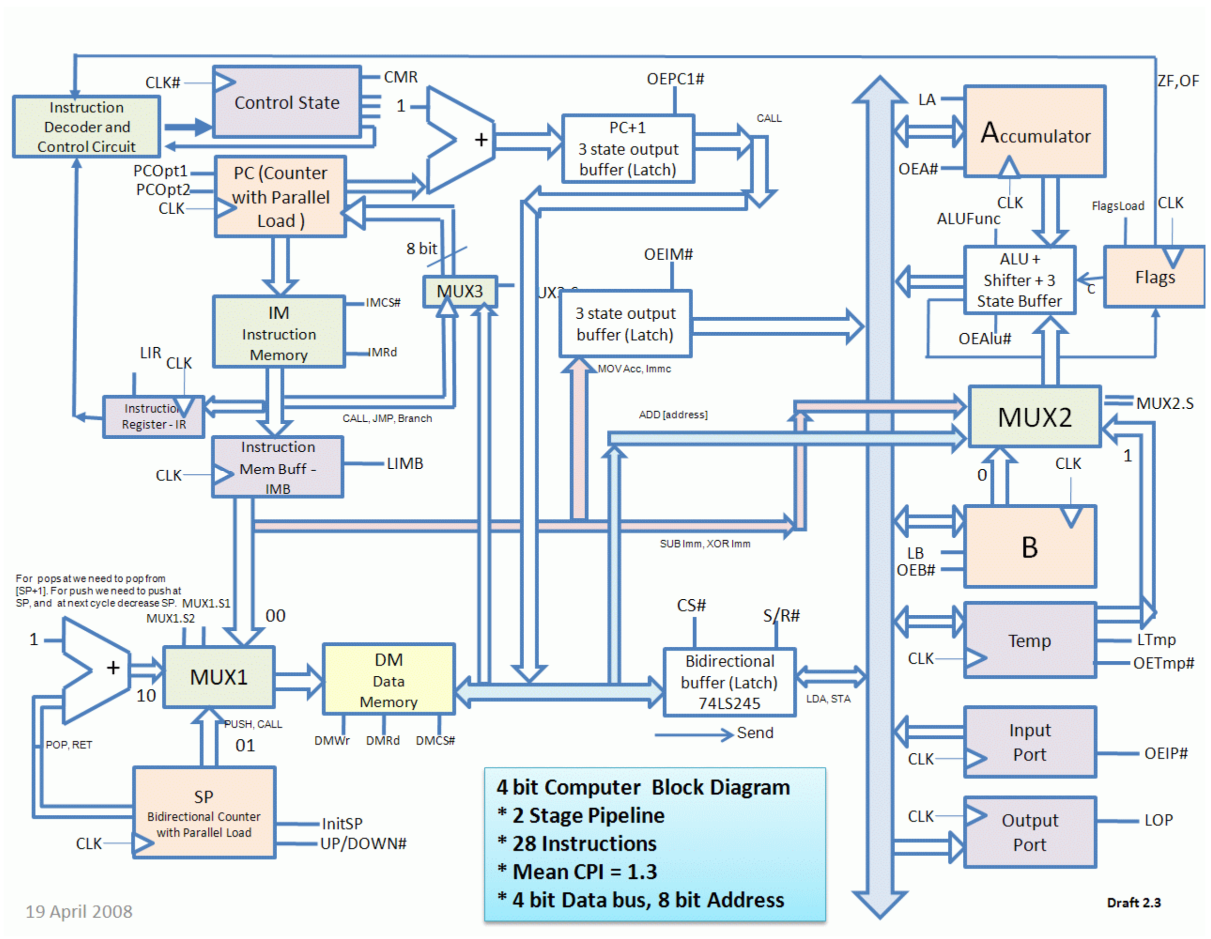


Fig: Block diagram of 4 bit computer

Description of Important Components

2.1 Fetch and decode unit

This unit supplies instructions in program order to the execution unit. It performs the following functions:

- Fetches instructions that will be executed next
- Decodes instruction into micro-instructions
- Generates micro-code for executing microinstructions

2.1.1 Program counter register

The program counter register contains the address of the next instruction to be executed. It is incremented automatically at each clock cycle to contain the next instruction address or operand address. Sometimes it is directly loaded with address values by JMP, CALL, RET, JE and JO instructions. The output of the program counter register is directly fed to the input of the instruction memory.

2.1.2 Instruction memory

Instruction memory is a 256X8 RAM. It has 8 address pins and 8 output pins. So it is byte addressable. Each byte contains the desired instruction. The read signal of instruction memory is always activated. Since address inputs are directly connected with the outputs of program counter register, hence the memory output is always available without requiring any extra clock cycle. As long as program counter outputs are valid, memory outputs are also valid. There is no need to write the instruction memory. Because we used separate memory for data and instruction. Data write and stack writes occurs on the data memory. Since program code is read only, hence write signal of instruction memory is deactivated permanently. Only at the start of the simulation program code needs to be loaded in the instruction RAM.

2.1.3 Instruction register

The instruction register holds the opcode of the instruction that is being executed on the execution unit. This register is 8 bits long. The instruction register content is directly fed to the instruction decoder unit to generate the micro-operations for each instruction.

2.1.4 Instruction memory buffer register

This register is used to hold the content of the second byte for all 2 byte instructions. The first byte which is the opcode byte is stored in the instruction register. During the first clock cycle the opcode byte is fetched into the instruction register. During the second clock cycle the second byte is fetched and stored in the instruction memory buffer register. The instruction register is not loaded during the second clock cycle.

2.1.5 MUX1, Data memory address selector

This MUX controls the input to the data memory address. The possible inputs can be

- SP value, required in PUSH instruction
- SP value + 1, required in POP instruction
- Instruction memory buffer register output, required in indirect data memory access where data address value is given as second byte of the instruction

2.1.6 Adder for computing PC + 1 and output buffer

This adder computes the value of current program counter + 1. It is required in CALL instruction where we need to save the address of next instruction to the stack. The 3 state buffer in front of the adder for PC + 1 output is necessary so that the adder output does not load the data memory output bus while data memory is accessed for other purpose.

2.1.7 Bidirectional transceiver

- In some instructions we require that the data memory output is to be passed to the shared data bus.
- In some other instructions we require that shared bus output is to be loaded to the data memory output bus.
- In other cases the data memory must not load the shared bus while others are using it

To meet all the above purposes we require a bi-directional 3 state buffer. The bus transceiver does the same. It has a chip select pin. If this pin is in active it does no load the bus on both sides. When chip select pin is active, then depending on another pin (data direction pin), it transfers data from any one side to the other side.

2.1.6 Instruction decoder unit

Instruction decoder unit decodes each instruction according to their type. This is a ROM that generates the appropriate micro-operation depending on the value in the instruction register. This decoder actually determines the ROM address at which the instruction is to be handled. The ROM address thus found generates the control signals for handling the execution of the instruction. However at the next clock cycle, the control ROM address might be selected from other sources other than instruction decoder output as determined by the instruction type and length of clock cycle required to execute it. The important thing that it does is that it maps many instructions to the same initial address (all these instructions have same output from the control ROM) of the ROM since all those instructions have the same control signals in their first clock cycle. It reduces number of rows needed for the control ROM thus reducing the ROM size.

2.1.7 Control unit

Control unit consists of a 32X32 ROM and a 2 bit flip flop. The ROM generates the control word for each micro-instruction and the 2 bit flip flop stores the next address selector value which is fed into a MUX selector pins to select control ROM address.

- The full block diagram of control unit is given below:

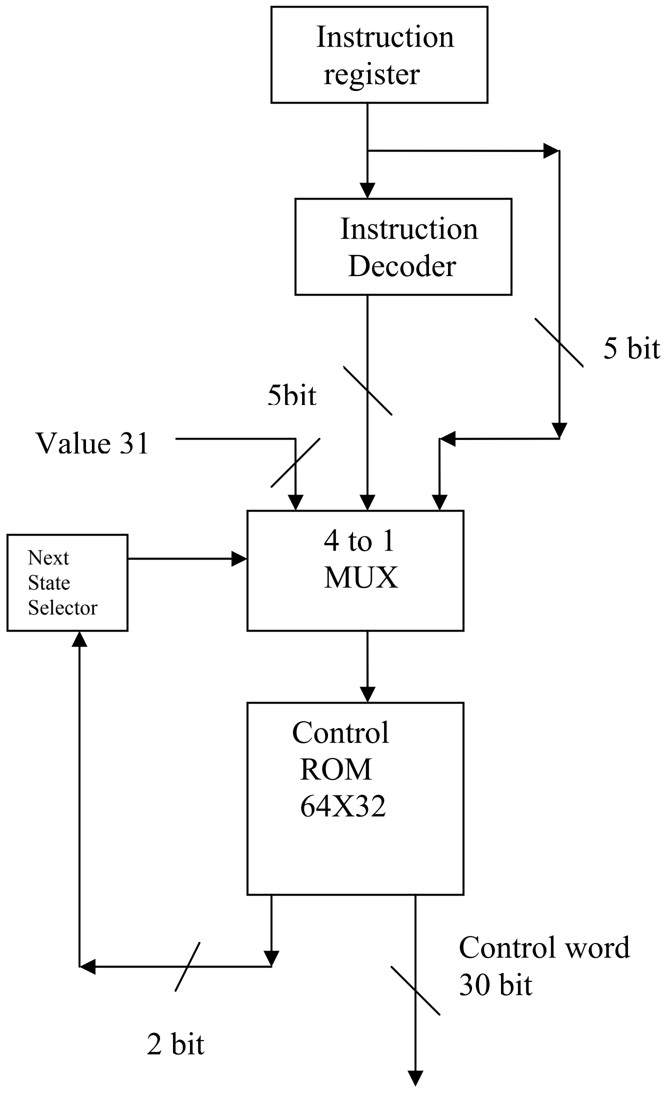


Fig: block diagram for control ROM

- **Control state diagram**

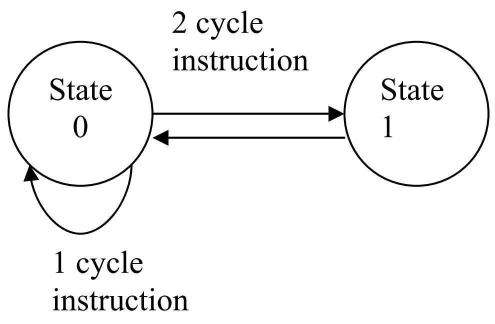


Fig: state diagram for control ROM

The working principle of control unit is as follows:

- For all instructions, at the first clock cycle the control ROM address is selected from instruction decoder output.
- The next address selector input from the control ROM output selects the appropriate next address of the control ROM through the MUX.
- For some 2 cycle instructions, at the second clock cycle the control ROM address input is selected from the instruction register output. These instructions have same control signals in their first clock cycle, however different control signals in their second clock cycle.
- For some other 2 cycle instructions, at the second clock cycle control ROM address input is selected as the constant value 31. This is because all these instruction have same control signals in their second clock cycle although they have different control signals in their first clock cycle.

2.2 Execution unit

The execution unit can perform arithmetic and logic operations, transfer values between general purpose registers and data memory and input and output port registers.

2.2.1 Arithmetic and logic unit

This is a 4-bit arithmetic and logic unit that supports the following operations:

- Arithmetic operations –addition, subtraction, increment, decrement and transfer operation
- Logical operations – logical or, xor, and, compliment operations

The operation of arithmetic logic unit is as determined by the function pins are given below:
One operand of the arithmetic logic unit is the accumulator register. The other operand is selected by MUX2. The two selector pins S1, S0 of MUX2 selects second operand as follows:

S1	S0	Operand
0	0	B register
0	1	1
1	0	Data memory output
1	1	Instruction memory buffer register output

- Immediate operands are selected from instruction memory buffer register output.
- Memory operands are selected from data memory output.
- During NEG instruction value 1 is selected.
- During all other arithmetic and logic instructions B register is the second operand.
-

2.2.2 Shifter unit

The shifter unit can shift operands 1-bit position either to the left or right. The operation of the shifter is determined by the two selector pins S1, S0:

S1	S0	Operation
0	0	Transfer
0	1	Shift right
1	0	Shift left
1	1	Transfer 0

2.2.3 Registers

The entire register set consists of:
- Two general purpose registers Accumulator register and B register

- One program status register or flags register
- One input port register
- One output port register

2.2.3.1 Functions of each register

- **Accumulato**r - For any arithmetic or logic operation one operand is the default accumulator register and the result also goes to this register.
- **B register** - This works as a general purpose temporary storage register. We can move values in and out of this register to the accumulator. This register can also be selected as the second operand for some arithmetic and logic operations.
- **Input port register** – The input port register is used for interfacing with devices outside the computer. This register is used to receive input values from devices like keyboard or any other device.
- **Output port register** – the output port register is used to send data to devices external to the computer. This device can be either a display or anything.
- **Flags register** –Stores status of recent arithmetic or logic operation.

2.2.3.2 The status flags register

The status flags of the 4-bit flags register indicate the results of arithmetic and logic operations such as ADD, SUB, OR, XOR. The status flag functions are:

CF (bit 0)	**Carry flag** – Set if an arithmetic operation generates a carry or a borrow out of the most significant bit of the result; cleared otherwise. This flag indicates an overflow for the unsigned-integer arithmetic. It is also used in shift operations and multiple precision arithmetic.
SF (bit 1)	**Sign flag** – Set equal to the most significant bit of the result which is the sign bit of the signed integer.
ZF (bit 2)	**Zero flag** – Set it the result is zero; cleared otherwise.
VF (bit 3)	**Overflow flag** – Set if the integer result is too large a positive number or too small a negative number (excluding the sign bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow for signed integer arithmetic.

- Arithmetic operations modifies all flags
- CF and VF are not changed during logic operations.
- CF can be changed by shift operations such as SHL, SHR

The status flags allow a single arithmetic or logic operation to produce results for two different data types: unsigned integers and signed integers.

When performing multiple precision arithmetic on integers, the CF flag is used in conjunction with add with carry (ADC) and subtract with borrow (SBB) instruction to propagate a carry or borrow from one computation to the next.

The conditional branch instructions (such as JE, JO) use one or more of the status flags as condition codes and test them for branching.

3 Addressing Mode

3.1 Instruction length

Length of each instruction is either 1 byte or 2 byte.
Each instruction consists of an opcode byte and optionally followed by an operand byte. The opcode byte determines the specific instruction. The operand byte may be any of the following:

- an immediate value
- an address value (used in imp, call instructions)
- an address value for data memory (used in indirect data addressing)

3.2 Operand addressing modes

Each instruction is either a zero or one or two operand instruction. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:

- the instruction itself (an immediate operand)
- a register
- a data memory location
- input port

When an instruction returns data to a destination operand it can be returned to:

- a register
- a data memory location
- output port

3.2.1 Immediate operands

Some instruction use data encoded in the instruction itself as a source operand. These operands are called immediate operands. For example the following instruction subtracts immediate value 04H from the accumulator register:

SUB 04H

There are three instructions available now that use immediate operands:

1. SUB imm
2. MOV ACC,imm
3. XOR imm

3.2.2 Register operands

For move operations source and destination operand can be either accumulator register or B register or any of the input or output port register. But for arithmetic and logic operations one source operand is the default accumulator register and the result is also stored in the accumulator register. So no destination operand need to be specified. It is implied in each instruction as the accumulator. The other source operand can be selected as B register.

3.2.3 Memory operands

Some operands can be directly specified by their address value in data memory where the operand is located. The actual operand is fetched from the data memory addressed by the value given in the instruction. For example the following instruction adds the operand value at data memory location 04H with the accumulator:

ADD [04H]

4 Data Types

4.1 Numeric data types

Each numeric data is a nibble that is four bits long. Only integer data types are supported. Depending on the interpretation this can be either:

- **Signed** – All signed operands are represented using two's compliment representation. The leftmost bit (bit 3) is the sign bit. For positive number this bit is zero (0). For negative number this bit is one (1). The range of numbers includes -8 to +7.
- **Unsigned** – All 4 bits are used to represent the value. The range of numbers includes 0 to +15.

4.2 Pointer data types

All pointers are addresses to data memory locations. They are 8 bits long. Hence the total number of locations that can be addressed is 256K Bytes. The total size of the data segment is therefore also 256K Bytes.

5 Supported Instruction set

Instructions can be divided into the following groups:

- Data transfer instructions
- Arithmetic instructions
- Logical instructions
- Shift instructions
- Control transfer instructions
- I/O instructions
- Stack instructions
- Miscellaneous instructions

5.1 Data transfer instructions

The data transfer instructions move data between:

- data memory and accumulator
- immediate value and accumulator
- between accumulator and B register or input port register or output port register

1. **MOV ACC, B –** moves the contents of B register to the accumulator register.

2. **MOV B, ACC** - moves the contents of accumulator to the B register.

3. **LDA [*addr*]** – loads the accumulator with the value at data memory address specified by *addr*. The *addr* value is 8 bits long so that 256K Bytes of data memory locations can be directly specified by this value.
4. **STA [*addr*] -** stores the contents of accumulator register to the data memory location specified by the *addr* value.

5. **MOV ACC,*imm* –** moves the immediate value to the accumulator register. The immediate value is given with the instruction. The next byte after the instruction byte contains this immediate value.

5.2 Arithmetic instructions

1. **ADD B** – adds the contents of the accumulator with the contents of the B register. The default destination of the result is the accumulator.

2. **ADC B** – adds the contents of the accumulator with the B register and contents of the carry flag.

3. **SUB B –** subtracts the contents of the B register from the accumulator.

4. **SBB B –** subtracts the contents of the B register and contents of the carry flag from the accumulator.

5. **CMP B –** subtracts the contents of the B register from the contents of the accumulator. However the result is not stored in the accumulator. Only the flags are affected.

6. **NEG –** negates the value of the accumulator. The accumulator value changes to the two's compliment of the value previously stored.

7. **SUB *imm* - subtracts** the immediate value from the contents of the accumulator.

8. **ADD [*addr*] -** adds with the accumulator the value at data memory location addressed by *addr.*

Note: All flags are affected during all the above arithmetic instructions.

5.3 Logical instructions

1. **OR B** – performs a bitwise inclusive OR operation between the accumulator and B register and stores the result in the accumulator. Each bit of the result is set to 0 if both corresponding bits of the accumulator and B register are 0; otherwise each bit is set to 1.

2. **XOR *imm*** – performs a bitwise exclusive OR operation between the accumulator and the immediate value specified by *imm.* Each bit of the result is set to 0 if both Corresponding bits are same; otherwise each bit is set to 1.

Note: Carry flag and overflow flags are not affected during these logical operations. The rest of the flags are affected.

5.4 Shift instructions

1. **SHL** – shifts the contents of the accumulator 1 bit position to the left. The most significant bit is saved in the carry flag (thus destroying the previous content of the carry flag). The least significant bit is 0.

2. **SHR** – shifts the contents of the accumulator 1 bit position to the right. The rightmost bit (LSB) is shifted out and saved in the carry flag. The most significant bit is 0.

5.5 Control transfer instructions

1. **JMP *addr*** – transfers program control to a different location in the instruction memory specified by *addr* without recording return value. The *addr* is an 8 bit value so that this instruction can jump to any location in the memory within 256K Bytes of instruction memory.

2. **JO *addr*** – transfers program control to the specified location if the condition is met that is if the contents of the overflow (V) flag is 1. Otherwise next instruction after this instruction will be sequentially executed.

3. **JE *addr*** – transfers program control to the specified location if the contents of the zero flag is 1. Otherwise next instruction after this instruction will be executed sequentially.

4. **CALL *addr* –** transfers program control the location specified by *addr.* But before that the address of the next sequential instruction is stored in the stack.

5. **RET -** loads the program counter with the value from the stack. It first pops the value from the stack and then stores it in the program counter.

5.6 I/O instruction

1. **IN –** this instruction transfers contents of the input port register to the accumulator.

2. **OUT –** this instruction transfers contents of the accumulator to the output port register.

Miscellaneous instructions

1. **NOP -** this instruction performs no operation.

2. **HLT -** this instruction halts the execution of the entire processor.

6 System Functions & Instruction Steps

6.1 Functioning mechanism of the system

After power on the computer system starts execution of the program starting at memory location 0000 of the instruction memory. Hence the first instruction of the program must be loaded at this address. After then an unconditional jump instruction can be used to run programs starting at any other address.

The instruction memory must be loaded with the program to be executed. The data memory should be loaded with initial data values if program requires. The stack pointer (SP register) is initially loaded with the value FFH at start up. Note that all data values and stack pops fetches values from data memory. Stack pointer grows from higher end (address FF) of the data memory to the lower addresses. After start up contents of any other register are undefined.

The entire program must end with the HLT instruction. Otherwise processor will not stop executing instructions and thus no output can be observed.

6.2 Instruction timing diagram

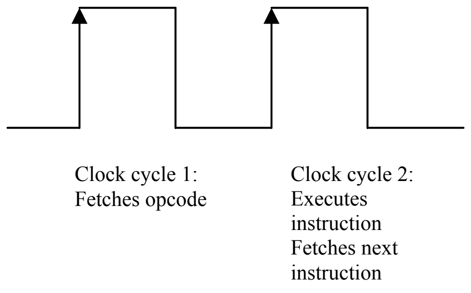
The two stage pipeline architecture allows one instruction in executing while another instruction is fetched from the instruction memory. Some instructions are executed in two clock cycles (containing 1 fetch cycle and 1 execution cycle) and others require three clock cycles (containing 2 fetch cycle and 1 execution cycle).

A two cycle instruction execution requires the following steps:

- Before the first clock cycle occurs, program counter contains the address of the instruction to be executed. This address value is directly connected to the input address of instruction memory. Hence the Opcode value is available at the output of the instruction memory.

- **Clock cycle 1**- At the first clock cycle (low to high transition) the Opcode value is loaded in the instruction register. The output of the instruction register is directly fed to the input of the instruction decoder. Hence decoded output of the decoder is available before the next clock cycle occurs. The decoded output is fed to the input of the control ROM which generates the control world before the next clock cycle occurs.

- **Clock cycle 2** – At this clock cycle the instruction is executed. Since control word was generated before this clock cycle, hence result of arithmetic or logical operation are also computed before this clock cycle and appears in the shared data bus. Only the output of the instruction is loaded at the low to high transition of this clock cycle. Fetch of next instruction also occurs in this clock cycle.



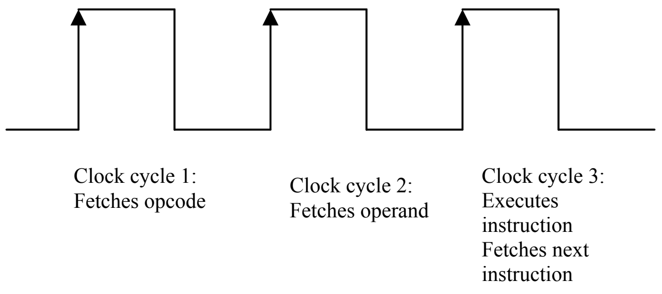
A three cycle instruction execution (except the control transfer instructions) requires the following steps:

- Before the first clock cycle occurs, program counter contains the address of the instruction to be executed. This address value is directly connected to the input address of instruction memory. Hence the Opcode value is available at the output of the instruction memory.

- **Clock cycle 1** - At the first clock cycle (low to high transition) the Opcode value (first byte of the instruction) is loaded in the instruction register. The output of the instruction register is directly fed to the input of the instruction decoder. Hence decoded output of the decoder is available before the next clock cycle occurs. The decoded output is fed to the input of the control ROM which generates the control world before the next clock cycle occurs.

- **Clock cycle 2**- At the second clock cycle (low to high transition) second byte of the instruction is loaded in the instruction memory buffer register (the instruction register remains unchanged and contains the first byte of this instruction that was loaded at the previous clock cycle). This can be either an immediate operand or data memory address operand.

- **Clock cycle 3** – At the third clock cycle the instruction is executed. Since control word was generated before this clock cycle, hence result of arithmetic or logical operation are also computed before this clock cycle and appears in the shared data bus. Only the output of the instruction is loaded at the low to high transition of this clock cycle. In this clock cycle fetch of next instruction also occurs.



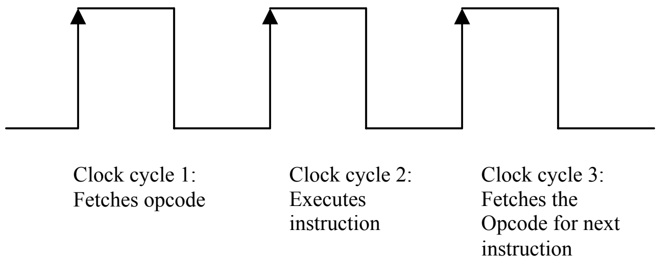
For control transfer instruction (such as JMP, CALL) the following steps occur:

- **Clock cycle 1** – same as others.

- **Clock cycle 2**- At this clock cycle, the program counter is loaded with address value selected by MUX3 from either instruction memory (during JMP, JE, JO, CALL instructions) or data memory (during RET instruction). In case of CALL instruction, the address of the next sequential instruction (current program counter value + 1) is also stored in the stack (data memory) at this clock cycle. The selector pin S0 determines the address value that is loaded to the program counter as follows:

S0	Value
0	Instruction memory output
1	Data memory output

- **Clock cycle 3** – At the third clock cycle no operation is performed. Only the fetch of the next instruction occurs. This can be considered as a pipeline stall.

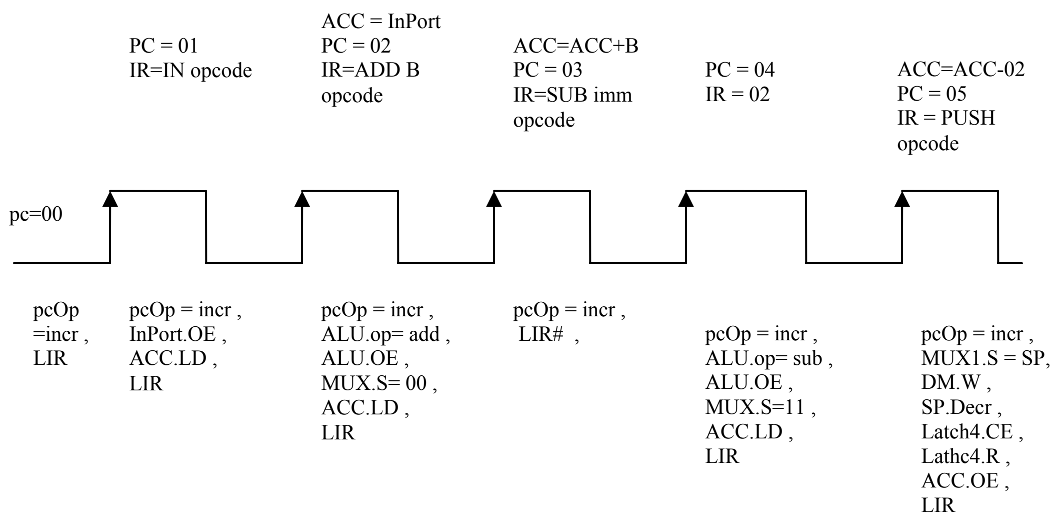


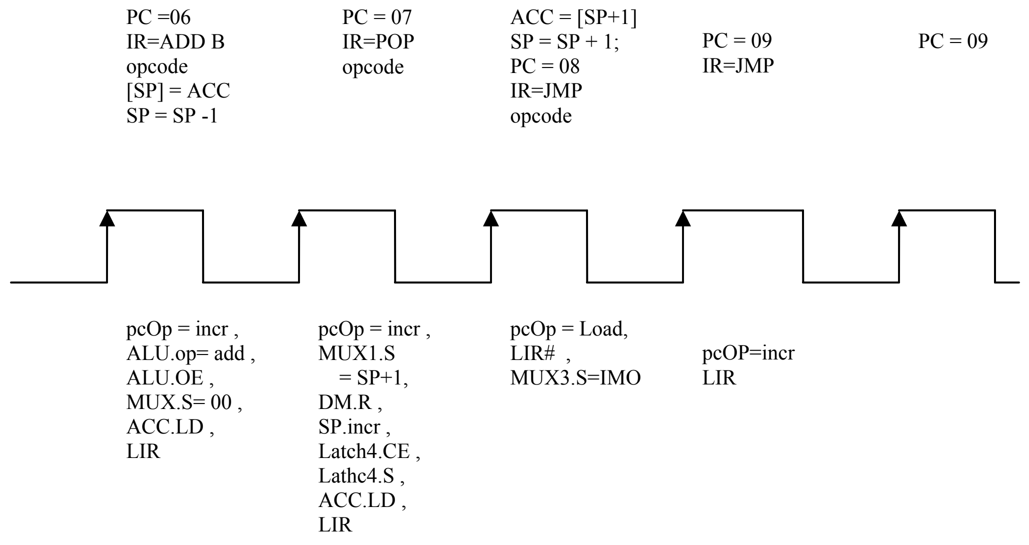
6.2.1 Actual timing diagram for example program with control signals explicitly shown:

Consider the following program segment:

Address	Instruction
00	IN
01	ADD B
02	SUB 02
04	PUSH
05	ADD B
06	POP
07	JMP 09

The actual timing diagram with control signals are shown below. In each clock pulse (a low to high transition of the clock signal), some results are stored in some registers and new control signals are generated according to the fetched instruction. The result of the previous control signals that are actually loaded in this clock cycle are shown above the clock cycle line and the control signals that are being generated for new instruction are shown below the clock cycle line. All control signals are named with their numeric interpretation (not the actual binary value that is generated). Only the effective control signals are shown for each clock cycle.





6.2.2 Time line diagram for example program

Address	Instruction
00	IN
01	ADD B

02	MOV B,ACC
03	SUB 02
05	PUSH
06	ADD B
07	POP

	C1	C2	C3	C4	C5	C6	C7	C8	C9
1. IN	F1	E							
2. ADD B		F1	E						
3. MOV B,ACC			F1	E					
4. SUB 02				F1	F2	E			
5. PUSH						F1	E		
6. ADD B							F1	E	
7. POP								F1	E

F1: Fetch opcode
F2: Fetch operand
E: Execution
C: Clock cycle

Following is another program that has several pipeline stalls due to JMP, CALL and RET instructions.

Address	Instruction
00	IN
01	ADD B
02	CALL 08
08	RET
04	JMP 0A
10	ADD B

	C1	C2	C3	C4	C5	C6	C7	C8	C9	
1. IN	F1	E								
2. ADD B		F1	E							
3. CALL 08			F1	E1	S					
4. RET					F1	E1	S			
5. JMP 0A							F1	E1	S	
6. ADD B									F1	E

F1: Fetch opcode
F2: Fetch operand
E: Execution
E1: Execution and load pc
S: Stall of execution pipeline
C: Clock cycle

6.3 Average CPI

For 2 cycle instructions, due to pipeline architecture clock cycle 2 (of current instruction executing) and clock cycle 1(of next instruction) are performed parallel in only 1 clock cycle. For 3 cycle instructions clock cycle 3 (of current instruction executing) and clock cycle 1(of next instruction) are performed parallel in only 1 clock cycle This reduces the average clock cycle requires for each instruction executed. The following table shows the required number of clock cycles for each instruction (considering pipelined architecture):

Instruction	Clock cycles
ADD B	1
ADC B	1
SUB B	1

SBB B	1
CMP B	1
MOV ACC,B	1
MOV B,ACC	1
NEG	1
OR B	1
IN	1
OUT	1
SHL	1
SHR	1
LDA [*addr*]	2
STA [*addr*]	2
MOV ACC, *imm*	2
SUB *imm*	2
XOR *imm*	2
ADD [*addr*]	2
JMP *addr*	2
JO *addr*	2
JE *addr*	2
CALL *addr*	2
RET	2
PUSH	1
POP	1
NOP	1
HLT	1

Average CPI = (11 * 2 + 17) / 28
= 1.4

7 Opcode and Control Matrix

7.1 Opcode table for all instructions

Instruction	Length (in Byte)	Opcode (in binary)		Opcode(in hex)	
		Byte 1	Byte 2	Byte 1	Byte 2
ADD B	1	0000 0001		01	
ADC B	1	0000 0010		02	
SUB B	1	0000 0011		03	
SBB B	1	0000 0100		04	
CMP B	1	0000 0101		05	
MOV ACC,B	1	0000 0110		06	
MOV B,ACC	1	0000 0111		07	
NEG	1	0000 1000		08	
OR B	1	0000 1001		09	
IN	1	0000 1010		0A	
OUT	1	0000 1011		0B	
SHL	1	0000 1100		0C	
SHR	1	0000 1101		0D	
LDA [*addr*]	2	0000 1110	*addr*	0E	*addr*
STA [*addr*]	2	0000 1111	*addr*	0F	*addr*
MOV ACC, *imm*	2	0001 0000	*imm*	10	*imm*
SUB *imm*	2	0001	*imm*	11	*imm*

		0001			
XOR *imm*	2	0001 0010	*imm*	12	*imm*
ADD [*addr*]	2	0001 0011	*addr*	13	*addr*
JMP *addr*	2	0001 0100	*addr*	14	*addr*
JO *addr*	2	0001 0101	*addr*	15	*addr*
JE *addr*	2	0001 0110	*addr*	16	*addr*
CALL *addr*	2	0001 0111	*addr*	17	*addr*
RET	1	0001 1000		18	
PUSH	1	0001 1001		19	
POP	1	0001 1010		1A	
NOP	1	0001 1011		1B	
HLT	1	0001 1100		1C	

7.3 Control Matrix Sheet

Address(in decimal)	Value(in hex)
0000	00
0001	01
0002	02
0003	03
0004	04
0005	05
0006	06
0007	07
0008	48

0009	09
0010	0A
0011	0B
0012	0C
0013	0D
0014	1E
0015	1E
0016	1E
0017	1E
0018	1E
0019	1E
0020	14
0021	15
0022	16
0023	17
0024	18
0025	19
0026	1A
0027	1B
0028	80
0029	00
0030	00
0031	00

7.4.1 Pin designations of the control ROM

ROM#	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit1	Bit0
1	PCOp. S1	PCOp. S0	MUX3. S	LIR	MUX1. S1	MUX1. S0	SP. U/D#	SP. CEN#
2	Latch3. OE	Latch3. S/R#	ACC. OE	ACC. LD#	DM. R/W#	Latch1. OE#	Latch2. OE	Latch4. OE
3	MUX2. S1	MUX2. S0	B. LD#	B. OE	InPort. OE#	OutPort. LD#	NS. S1	NS.S0
4	ALU. S2	ALU. S1	ALU. S0	ALU. M	CarrySel	SHOp. S1	SHOp. S0	Flags. LD#

- **PCOp.S1:S0** – these two bits selects the pc operation at the next clock cycle according to the following table:

S1	S0	Operation
0	0	Increment Pc
0	1	Load Pc
1	0	Load Pc if V = 1
1	1	Load Pc if Z = 1

- **MUX3.S** – this pin selects the value to be loaded into the program counter. If 1 loads pc with the value from instruction memory output; otherwise loads pc from data memory output.
- **LIR** – Load instruction register. If 1 loads the instruction register with the value of instruction memory at the next clock cycle.
- **MUX1.S1:S0** – these two bit selects the address that will be the input to the data memory according to the following table:

S1	S0	Address input to data memory
0	0	Instruction memory buffer register
0	1	SP
1	0	SP+1
1	1	Unused

- **SP.U/D#** - if 1 increments the stack pointer register at the next clock cycle; otherwise decrements. Note that this pin is active only when SP.CEN = 1.
- **SP.CEN** – count enable signal for SP. This pin enables SP.U/D# signal.
- **Latch3.OE** – this pin enables the bus transceiver.
- **Latch3.S/R#** - if 1 the bus transceiver sends data (from data memory to the data bus); otherwise receives data (from data bus to data memory).
- **ACC.OE** – output enable signal for the accumulator register. If active the accumulator loads the data bus with its contents.
- **ACC.LD#** - load signal for the accumulator register. If active accumulator will be loaded with the value from data bus at the next clock cycle.
- **DM.R/W# -** data memory read/write# signal. If 1 data memory is selected for read operation; otherwise data memory is selected for write operation.
- **Latch1.OE# -** this pin enables the output buffer for PC + 1 output latch. This loads the output of the data memory output bus.
- **Latch2.OE -** this pin enables output buffer for instruction memory buffer register. This loads the data bus with the value of this register.
- **Latch4.OE –** output enable signal for arithmetic logic and shifter unit.
- **MUX2.S1:S0 –** these two bits selects the second operand for the arithmetic and logic operation according to the following table:

S1	S0	Operand
0	0	B register
0	1	1
1	0	Data memory output
1	1	Instruction memory buffer register output

- **B.LD# -** load signal for B register.
- **B.OE –** output enable signal for B register.
- **InPort.OE# -** output enable signal for input port register
- **OutPort.LD# -** load signal for output port register.
- **NS.S1:S0 –** selects next ROM input
- **ALU.S2:S0 –** these three bits selects the appropriate ALU operation to be performed.
- **ALU.M –** selects the mode of ALU operation. If 1 the operation is logic; otherwise operation will be arithmetic
- **CarrySel –** this bit determines the carry in signal for ALU along with some other bits.
- **SHOp.S1:S0 –** selects the shifter operation according to the following table:

S1	S0	Operation
0	0	Transfer
0	1	Shift right
1	0	Shift left
1	1	Transfer 0

- **Flags.LD#** - load signal for flags register.

7.4.1 Contents of the control ROM

address	ROM1 (in hex)	ROM2 (in hex)	ROM3 (in hex)	ROM4 (in hex)
0000	11	9C	2C	00
0001	11	8D	2C	A8
0002	11	8D	2C	A0
0003	11	8D	2C	40
0004	11	8D	2C	48
0005	11	9C	2C	40

0006	11	8C	3C	01
0007	11	BC	0C	01
0008	11	BD	6C	A8
0009	11	BD	2C	D0
0010	11	8C	24	01
0011	11	BC	28	01
0012	11	8D	2C	0C
0013	11	8D	2C	0A
0014	11	4C	2C	01
0015	11	34	2C	01
0016	11	8E	2C	01
0017	11	8D	EC	40
0018	11	8D	EC	50
0019	11	8D	AC	A8
0020	41	9C	2E	01
0021	81	9C	2E	01
0022	C1	9C	2E	01
0023	44	90	2E	01
0024	6A	9C	2E	01
0025	15	34	2C	01
0026	1A	4C	2C	01
0027	00	00	00	00
0028	00	00	00	00
0029	00	00	00	00
0030	01	9C	2D	01
0031	11	9C	2C	01

8 Implementation of components

Program counter and Stack pointer register

Program counter and stack pointer are both implemented by counters. We used here 4 bit bi-directional counter IC no 74LS169. It has the following features that made it special to serve our purpose:

- Up and down capability.
- Parallel load capability.
- Two independent count enable signals
- One full count signal for cascading

Since we required 8 bit counter for both program counter and stack pointer we cascaded two 4 bit counters to make 8 bit counter. The TC (pin 15) output pin of the low nibble counter is connected to the count enable signals of the high nibble counter.

8.1.2 Accumulator register

It is a 4 bit flip flop IC 74LS173. One output of this register is passed to the ALU operand through a MUX. Another is fed into the IC 74LS126. It is a 3 state buffer. The output of this buffer is connected to the shared data bus. The output enable signal of this buffer connects and disconnects the accumulator register output from the shared buffer. It is required so that it does not load the bus when another register is loading the bus with its output.

8.1.3 B register

It is same as accumulator register except that one output of this register is fed into the MUX2.

8.1.4 Arithmetic logic unit

IC no 74LS181 is a 4 bit arithmetic logic unit. The input and output can be considered as either active low or active high. However depending on the active low or active high interpretation of inputs and outputs the carry in signal is just the opposite of what we consider it to be. For example if we want to perform add with carry operation and we consider active high inputs and outputs, then carry in signal must be active low. The 4 function selector pins enables various arithmetic and logic operations. One mode select pin is available to select between arithmetic and logic operations.

8.1.5 Shifter Unit

The shifter unit is implemented by two 74LS153 dual 4-to-1 MUX. This is a combinational shifter and no clock cycle is required to perform shifting.

8.1.6 Input Port Register and Output Port Register

These are both 4 bit flip flop IC 74LS173. The input enable signal and output enable signal are used for input and output enable.

8.1.7 RAM

For simulation of the circuit, we used circuit maker software. In circuit maker software a generic RAM1K is available. It has 10 address pins and 8 output pins. We used it in our simulation for both instruction memory and data memory. Since we required 8 address bits, the highest 2 bits were left at zero voltage (ground).

8.1.8 PROM

In circuit maker only 1 ROM was available. That is 32X8 bit PROM. It has 5 address pins and 8 output pins. We needed 32X32 PROM. So we cascaded 4 PROMs. Address pins of these 4 PROMs are connected together to generate 32 bit output simultaneously.

8.1.8 Bidirectional buffer

Octal 3 state bus transceiver IC 74LS245 was used for connecting the output of the data memory to the shared bus. For some instructions we have to send data from the data memory to the shared bus. For some other instruction we have to receive data from the shared bus to the data memory (during memory write operations). Hence bi-directional buffer was necessary. The S/R# pin (pin 1) of this IC controls the direction of data transfer through the transceiver. If chip select pin is inactive then the device disconnects the memory output from the data bus so that it does not load the data bus while others are using the shared bus.

8.2 Name and number of IC's used

IC#	Name	Pcs
74LS173	Quad 3 state D-type flip flop	9
74LS181	4-bit arithmetic logic unit	1
74LS126	Quad 3-state buffers	5
74LS153	4 to 1 multiplexer	14
74LS169A	4 bit bidirectional counter	4
74LS273	Octal D-flip flop	1
74LS157	Quad 2 to 1 multiplexer	5
74LS245	Octal 3 state transceiver	1
74LS83A	4 bit binary full adder	4
74LS373	Octal 3 state latch	1
74LS10	3 input NAND	1
74LS04	Hex inverter	3
74LS25	Dual 4 input NOR gate	1
74LS27	3 input NOR	1
74LS138	1-of-8 decoder	1
74LS08	Quad 2 input AND	1
74LS32	Quad 2 input OR	1
RAM	Generic RAM 1KX8bit	2
PROM	Generic PROM 32X8 bit	5

9 Problems encountered and discussion

- The main objective of our 4 bit computer design was to reduce number of average clock cycles for each instruction. To achieve this purpose we had to add additional hardware in our design. Our initial design was based on Von-Neumann architecture. Since the instruction set that was supplied to us to implement was as like as Intel's instruction set, we followed their architecture. So we used same memory for instruction and data in our initial design. However, this required some greater number of clock cycles for some instructions. It is because the clock cycle, at which we were fetching data from memory for an instruction, we were unable to fetch Opcode for next instruction at the same clock cycle. This stalled the fetch pipeline. Later we separated our data memory from instruction memory. This reduced number of average clock cycles for those instructions to be nearly equal to one.

- Writing to data memory caused us a serious problem in our design. We needed that the address value at which new data will be written to memory must be stable before the write signal is generated. It is because if the write signal is generated before the address value is stable; it might happen that data will be written to unknown address values. So at first we kept three clock cycles for each memory write operation. The first clock cycle allowed the address value to be stable, the actual write occurred in the second clock cycle and the third clock cycle was for de-activating the write signal before changing the address value. However this introduced two additional clock cycles for every instruction that involved memory write operation such as PUSH, STA. However later we solved this problem using the clock signal for writing. The address value was generated at the low to high transition of the clock cycle. So we decoded the write signal in the following way:

    - when clock is high write signal is inhibited
    - when clock is low write signal is actived

    This ensured half the clock cycle time (after low to high transition and before high to low transition of the clock) for the address values to be stable before write signal begins active. So our problem was solved easily by using only some gates.

- We had to consider each instruction separately to design the computer rather than thinking generally. Because we were supposed to implement only those instructions at lowest clock cycles. So we had the option to think each instruction separately to reduce clock cycle by adding hardware components. Since we had not to implement the actual design we were not afraid of too much chips and wires.

- Some flip flops have no master reset pin. This caused a big problem for us after designing the whole system. Because at the start of each new simulation the flip flop values were unknown. Moreover unnecessary values were being written in data memory. So we wanted that we would zero our all flip flops at the start of each simulation by one control signal. But the unavailability of master reset pin of some flip flops compelled us later to change those. At last we used IC 74LS273 because it had a master reset pin.

- We kept two separate memories for instruction and data like MIPS architecture. However MIPS architecture considers instruction to be of fixed sized. Our instruction size was not fixed. Some instructions were of 1 byte and others are of 2 bytes. This is dissimilarity between the architecture and instruction set.

- At first we had too many control signals (nearly 40) and 6 input lines for the control ROM. This required that the control ROM should be of size 64X40. Circuit maker software supports only one ROM that is of size 32X8. We had to cascade 10 ROMs to generate the all control signals. However later we reduced the input lines from 6 to 5 by considering the following facts:

    - first execution cycle of some instructions was doing the same thing
    - second cycle of some other instructions was doing the same thing

  We used an instruction decoder ROM. For each instruction this generates the address at which the instruction will be executed. In this way we mapped several instructions to the same address of the control ROM, since those required the same control signals. Later we was able to reduce to total number of control signals to 32. We required only 5 ROMs to implement the control circuit.

- NEG instruction required to compute two's complement of the accumulator. Since 4 bit ALU which we used in our design does not support this function, we fell in a trouble to perform it in one clock cycle. We could easily compute it in two clock cycles. In the first cycle we would compute the one's compliment (ALU supports this function). In the second clock cycle we would increment the accumulator. However since all other instructions required at most two clock cycles two execute, implementing NEG in this way would require NEG three clock cycles. So to reduce it later we added a MUX and inverter gates in front of ALU input A from the accumulator to generate the one's compliment of the accumulator and passing it directly to the ALU input without using the ALU. This allowed the NEG to be computed in only one clock cycle.

- At first running a program in the simulator (circuit maker) we required to load the hex value of the program instructions in instruction memory. However converting each instruction mnemonic to their hex value was disgusting. We wrote a C program *assembler.c*. This program takes as input file and generates an output file converting each mnemonic instructions to their hex values. The program is given below:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>


int main(int argc,char * argv[])
{

        if(argc!=3)
        {
                printf("Too many arguments...\n");
                exit(0);
        }

        FILE * in, * out;

        in = fopen(argv[1],"r");
        out = fopen(argv[2],"w");
        if(in==NULL | out==NULL)
        {
                printf("One or more files could not be opened..\n");
                exit(1);
        }



        char str[10];
        int state;


        state = 0;
        int address=0;

        fprintf(out,"addr: value\n");
        fprintf(out,"----  -----\n");
        while(1)
        {
                if( feof(in) )
                {
                        fclose(in);
                        fclose(out);
```

```
		break;
	}

	fscanf(in,"%s",str);
	if( strcmp(str,"ADD")==0 )
	{
		fscanf(in,"%s",str);
		if( strcmp(str,"B")==0 )
		{
			fprintf(out,"%04d: %02X\n",address,1);
			address++;
		}
		else
		{
			fprintf(out,"%04d: %02X\n",address,19);
			address++;

			fprintf(out,"%04d: %s\n",address,str);
			address++;
		}

	}
	else if( strcmp(str,"ADC")==0 )
	{
		fscanf(in,"%s",str);
		if( strcmp(str,"B")==0 )
		{
			fprintf(out,"%04d: %02X\n",address,2);
			address++;
		}
		else
		{
			printf("Invalid operand for ADC..\n");
		}

	}
	else if( strcmp(str,"SUB")==0 )
	{

		fscanf(in,"%s",str);
		if( strcmp(str,"B")==0 )
		{
			fprintf(out,"%04d: %02X\n",address,3);
			address++;
		}
		else
		{
			fprintf(out,"%04d: %02X\n",address,17);
			address++;

			fprintf(out,"%04d: %s\n",address,str);
			address++;
		}

	}else if( strcmp(str,"SBB")==0 )
	{

		fscanf(in,"%s",str);
		if( strcmp(str,"B")==0 )
		{
			fprintf(out,"%04d: %02X\n",address,4);
			address++;
		}
		else
		{
			printf("Invalid operand for SUB..\n");
		}

	}
	else if( strcmp(str,"CMP")==0 )
	{
		fscanf(in,"%s",str);
		if( strcmp(str,"B")==0 )
		{
			fprintf(out,"%04d: %02X\n",address,5);
			address++;
		}
		else
		{
			printf("Invalid operand for CMP..\n");
		}

	}
	else if( strcmp(str,"MOV")==0 )
	{
		fscanf(in,"%[ ]",str);


		fscanf(in,"%[A-Z]",str);



		if( strcmp(str,"ACC")==0 )
		{
			fscanf(in,"%[, ]",str); //discand , and spaces if any


			fscanf(in,"%s",str);


			if( strcmp(str,"B")==0 )
			{

				fprintf(out,"%04d: %02X\n",address,6);
```

```
                                address++;

                        }
                        else
                        {
                                fprintf(out,"%04d: %02X\n",address,16);
                                address++;

                                fprintf(out,"%04d: %s\n",address,str);
                                address++;

                        }
                }
                else if( strcmp(str,"B")==0 )
                {
                        fscanf(in,"%[, ]",str);
                        fscanf(in,"%s",str);


                        if( strcmp(str,"ACC")==0 )
                        {
                                fprintf(out,"%04d: %02X\n",address,7);
                                address++;
                        }
                        else
                        {
                                printf("Invalid operand for MOV B,ACC");
                        }
                }


        }else if( strcmp(str,"NEG")==0 )
        {
                fprintf(out,"%04d: %02X\n",address,8);
                address++;

        }
        else if( strcmp(str,"OR")==0 )
        {

                fscanf(in,"%s",str);
                if( strcmp(str,"B")==0 )
                {
                        fprintf(out,"%04d: %02X\n",address,9);
                        address++;
                }
                else
                {
                        printf("Invalid operand for OR..\n");
                }

        }else if( strcmp(str,"IN")==0 )
```

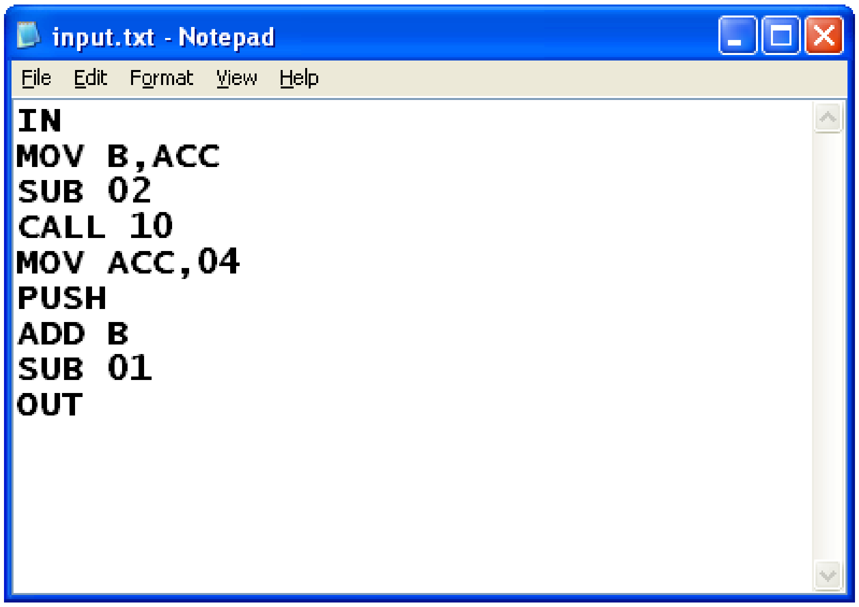
```
{
            fprintf(out,"%04d: %02X\n",address,10);
            address++;
}else if( strcmp(str,"OUT")==0 )
{
            fprintf(out,"%04d: %02X\n",address,11);
            address++;

}else if( strcmp(str,"SHL")==0 )
{
            fprintf(out,"%04d: %02X\n",address,12);
            address++;

}else if( strcmp(str,"SHR")==0 )
{
            fprintf(out,"%04d: %02X\n",address,13);
            address++;

}else if( strcmp(str,"LDA")==0 )
{
            fprintf(out,"%04d: %02X\n",address,14);
            address++;


            fscanf(in,"%s",str);

            fprintf(out,"%04d: %s\n",address,str);
            address++;


}
else if( strcmp(str,"STA")==0 )
{
            fprintf(out,"%04d: %02X\n",address,15);
            address++;


            fscanf(in,"%s",str);

            fprintf(out,"%04d: %s\n",address,str);
            address++;

}else if( strcmp(str,"XOR")==0 )
{
            fprintf(out,"%04d: %02X\n",address,18);
            address++;


            fscanf(in,"%s",str);

            fprintf(out,"%04d: %s\n",address,str);
            address++;
```

```
}else if( strcmp(str,"JMP")==0 )
{
		fprintf(out,"%04d: %02X\n",address,20);
		address++;


		fscanf(in,"%s",str);

		fprintf(out,"%04d: %s\n",address,str);
		address++;
}else if( strcmp(str,"JO")==0 )
{
		fprintf(out,"%04d: %02X\n",address,21);
		address++;


		fscanf(in,"%s",str);

		fprintf(out,"%04d: %s\n",address,str);
		address++;

}else if( strcmp(str,"JE")==0 )
{
		fprintf(out,"%04d: %02X\n",address,22);
		address++;


		fscanf(in,"%s",str);

		fprintf(out,"%04d: %s\n",address,str);
		address++;
}else if( strcmp(str,"CALL")==0 )
{
		fprintf(out,"%04d: %02X\n",address,23);
		address++;


		fscanf(in,"%s",str);

		fprintf(out,"%04d: %s\n",address,str);
		address++;

}else if( strcmp(str,"RET")==0 )
{
		fprintf(out,"%04d: %02X\n",address,24);
		address++;

}else if( strcmp(str,"PUSH")==0 )
{
		fprintf(out,"%04d: %02X\n",address,25);
		address++;
```

```
                }else if( strcmp(str,"POP")==0 )
                {
                        fprintf(out,"%04d: %02X\n",address,26);
                        address++;
                }else if( strcmp(str,"NOP")==0 )
                {
                        fprintf(out,"%04d: %02X\n",address,27);
                        address++;

                }else if( strcmp(str,"HLT")==0 )
                {
                        fprintf(out,"%04d: %02X\n",address,28);
                        address++;
                }
                else
                {
                        printf("Opcode mismatch..\n");
                }

        }

        return 0;
}
```

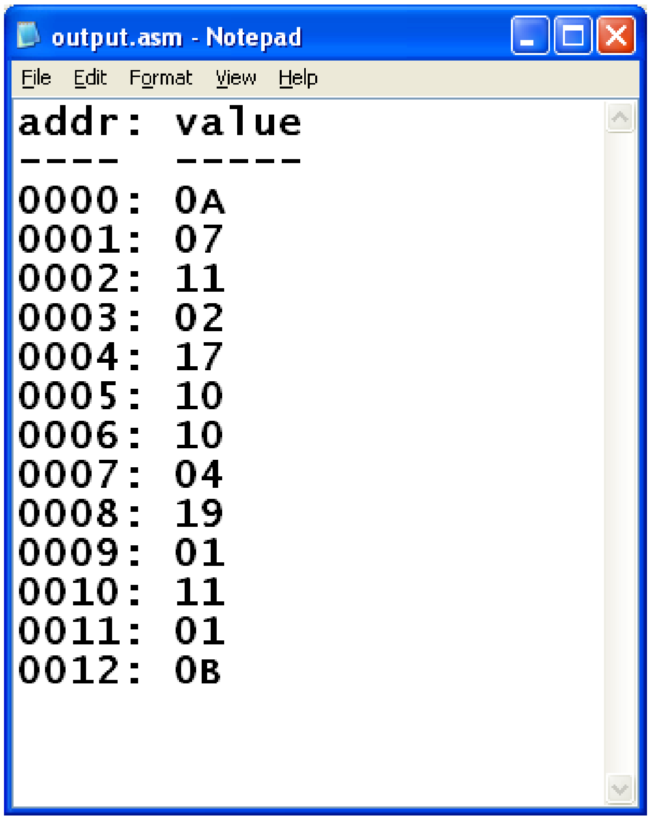
- How to run the assembler program:
- First compile the program and build the program to geneate the binary file assembler.exe
- From the command line write: assembler.exe input.txt output.asm

For example assume the following program instructions are written in input.txt file:



```
IN
MOV B,ACC
SUB 02
CALL 10
MOV ACC,04
PUSH
ADD B
SUB 01
OUT
```

The program generates the following output.asm file:

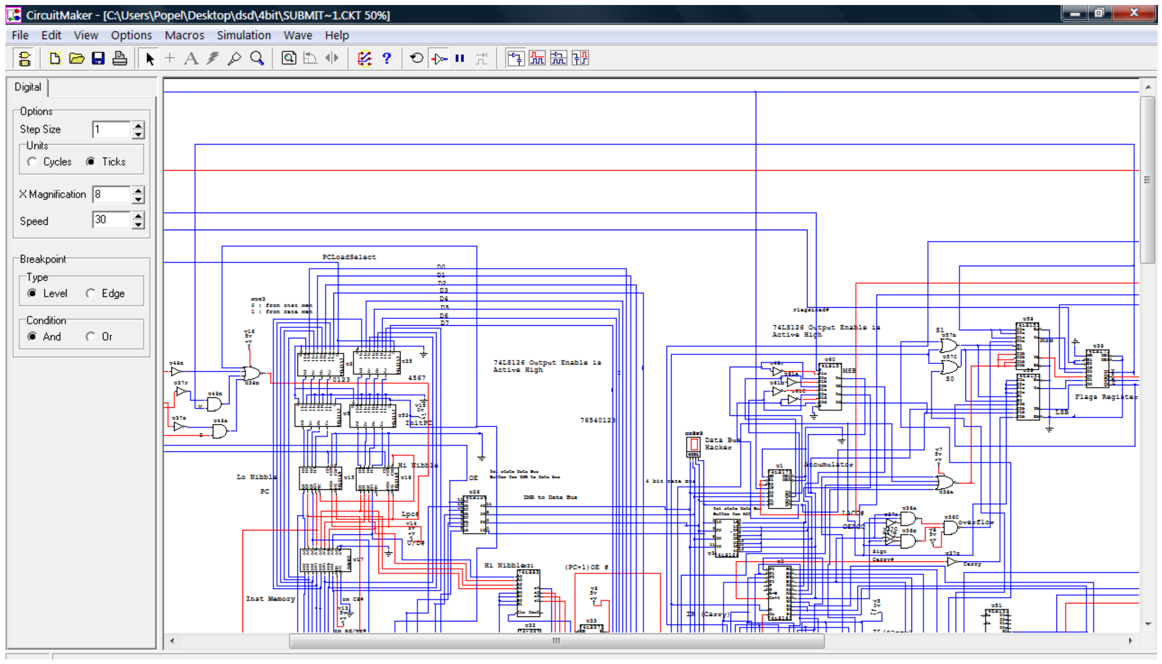


```
addr: value
----  -----
0000: 0A
0001: 07
0002: 11
0003: 02
0004: 17
0005: 10
0006: 10
0007: 04
0008: 19
0009: 01
0010: 11
0011: 01
0012: 0B
```

For each address value in the RAM, it shows the hex value that is to be written.

10 About the simulation tool

We used Circuit maker 2000 for simulating our processor.


CircuitMaker - [C:\Users\Popel\Desktop\dsd\4bit\SUBMIT~1.CKT 50%]
File Edit View Options Macros Simulation Wave Help
Digital
PCLoadSelect
74LS126 Output Enable is Active High
Lo Nibble
PC
Hi Nibble
Inst Memory
Data Bus Hacker
Accumulator
Flags Register
Overflow
Carry
Sign
(PC+1)OE

Options
Step Size 1
Units: ( ) Cycles (●) Ticks
X Magnification 8
Speed 30
Breakpoint
Type: (●) Level ( ) Edge
Condition: (●) And ( ) Or

11 Running a program in Simulator

To run a program in the simulation following steps are to be taken.

>> Write the program in assembly

>> Using the assembler tool provided, make the machine code of it. It will give Address:Code in hex format

>> Before starting the simulation, double click on Instruction Memory, and write the codes in each address as created by the assembler.

12 Conclusion

We enjoyed designing this 4 bit computer. The design was revised in several phases. At first we decided not to use separate data and instruction memory (use Von Neumann architecture). Then we found that we could improve the performance greatly by using Harvard architecture. Thus we ended up with a design, which could execute most of the instructions in one cycle and rest in two. The simulation part was most exciting. Because it let us check our own design in action.