PEN : 200840131012

PRACTICAL-1

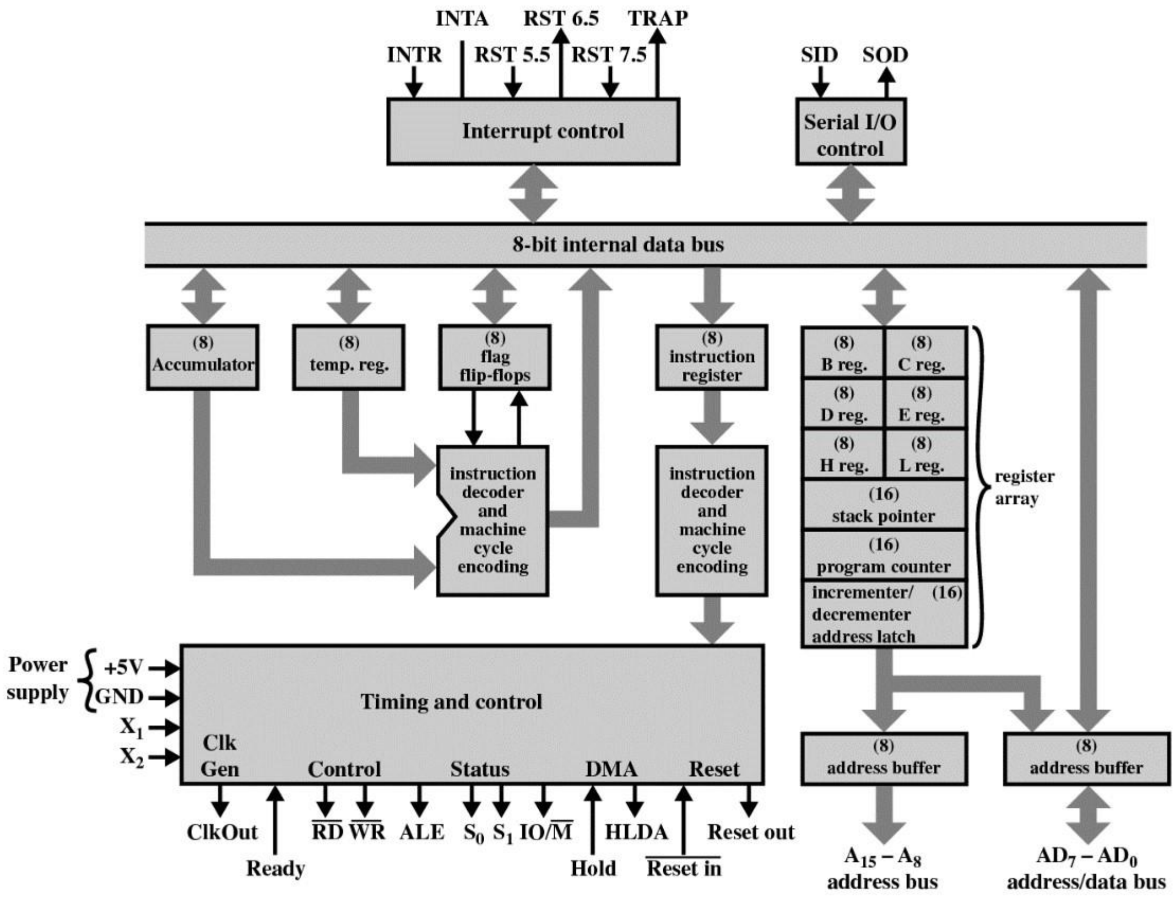
AIM : Write the working of 8085 simulator GNUsim8085 and basic architecture of 8085 along with small introduction.

Introduction:

GNUSim8085 is a graphical simulator, assembler and debugger for the Intel 8085 microprocessor in Linux and Windows. It is among the 20 winners of the FOSS India Awards announced on February, 2008. GNUSim8085 was originally written by Sridhar Ratnakumar in fall 2003 when he realized that no proper simulators existed for Linux. Several patches, bug fixes and software packaging have been contributed by the GNUSim8085 community. GNUSim8085 users are encouraged to contribute to the simulator through coding, documenting, testing, translating and porting the simulator.

Architecture:

The architecture of the 8085 microprocessor mainly includes the timing & control unit, Arithmetic and logic unit, decoder, instruction register, interrupt control, a register array, serial input/output control. The most important part of the microprocessor is the central processing unit.



It can perform operations that are given below:

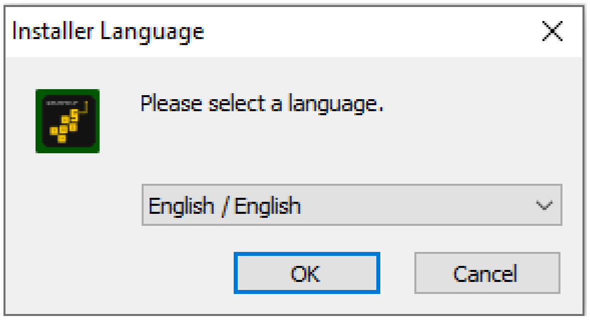
1. Operates on and stores 8-bit data.
2. It executes arithmetic and logic operations.
3. 8085 also sequences the instructions to be executed.
4. Stores data temporarily.

However, in order to perform all such operations, the processor needs a control unit, arithmetic logic unit, registers, buses etc.

Installion Process:

-Go to web browser and search GNUsim8085 download.
-Download GNUsim8085 Link:- https://gnusim8085.srid.ca/download
-After download the gnusim8085-1.4.1-installer.exe file and click the .exe file and following below operation:-

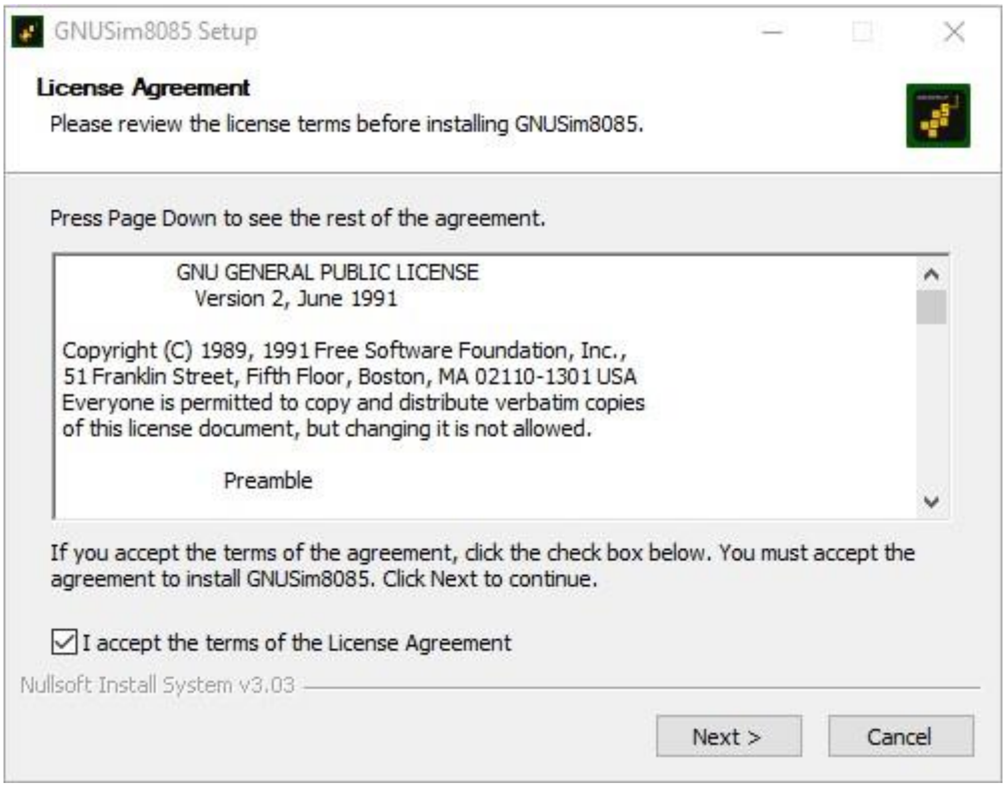
1.) Installion Language Of GNUsim8085.


Installer Language

Please select a language.

English / English

2.) GNUsim8085 License Agreement.
- Accept the terms of the License Agreement And on Next.


GNUSim8085 Setup

License Agreement

Please review the license terms before installing GNUSim8085.

Press Page Down to see the rest of the agreement.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

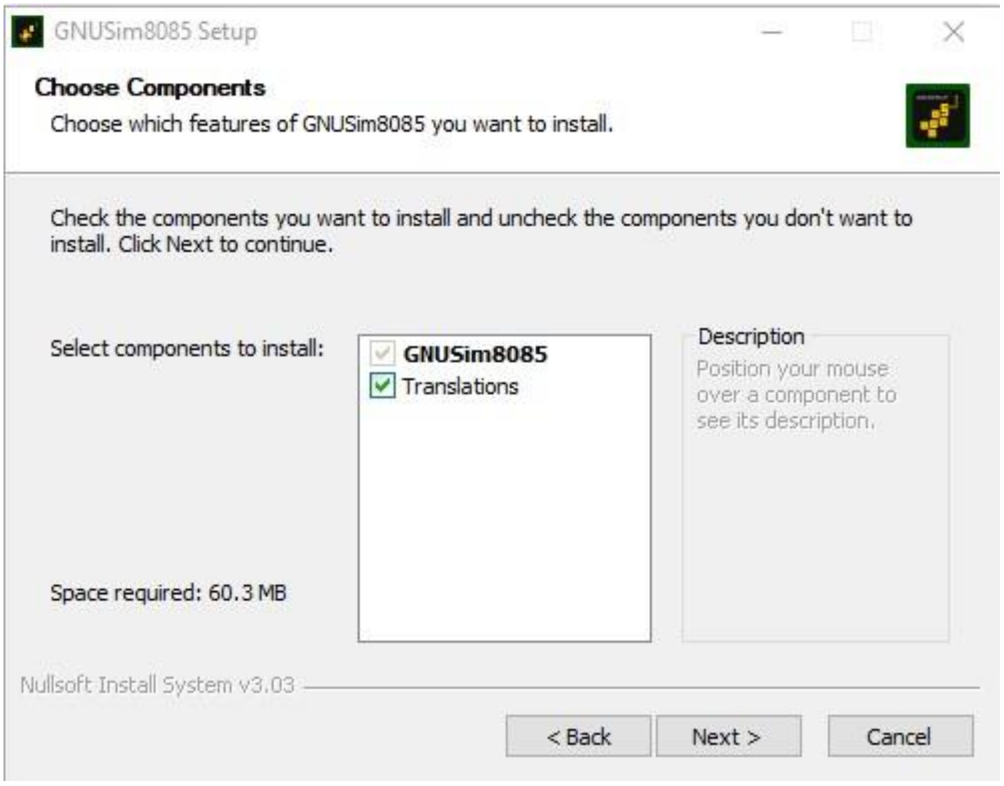
Preamble

If you accept the terms of the agreement, click the check box below. You must accept the agreement to install GNUSim8085. Click Next to continue.

☑ I accept the terms of the License Agreement

Nullsoft Install System v3.03

3.) Choose the Components to install.



GNUSim8085 Setup

Choose Components

Choose which features of GNUSim8085 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

✓ **GNUSim8085**
✓ Translations

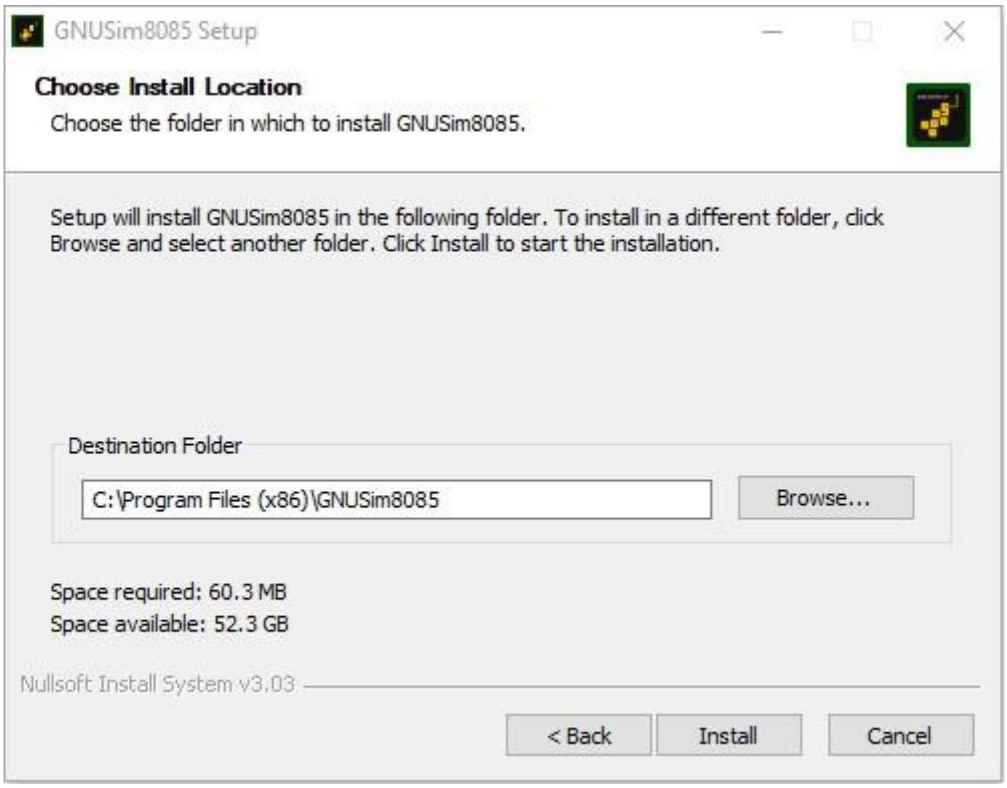
Description

Position your mouse over a component to see its description.

Space required: 60.3 MB

Nullsoft Install System v3.03

4.) Choose the GNUsim8085 Location.



GNUSim8085 Setup

Choose Install Location

Choose the folder in which to install GNUSim8085.

Setup will install GNUSim8085 in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder

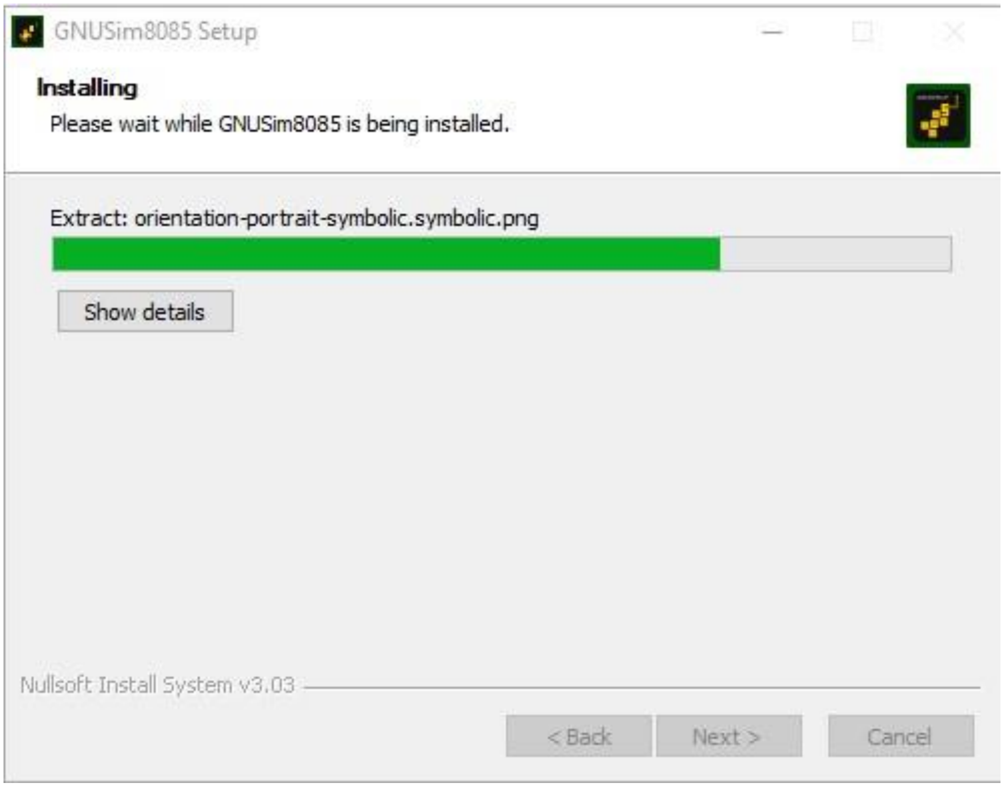
C:\Program Files (x86)\GNUSim8085

Space required: 60.3 MB

Space available: 52.3 GB

Nullsoft Install System v3.03

5.) GNUsim8085 is being installed.

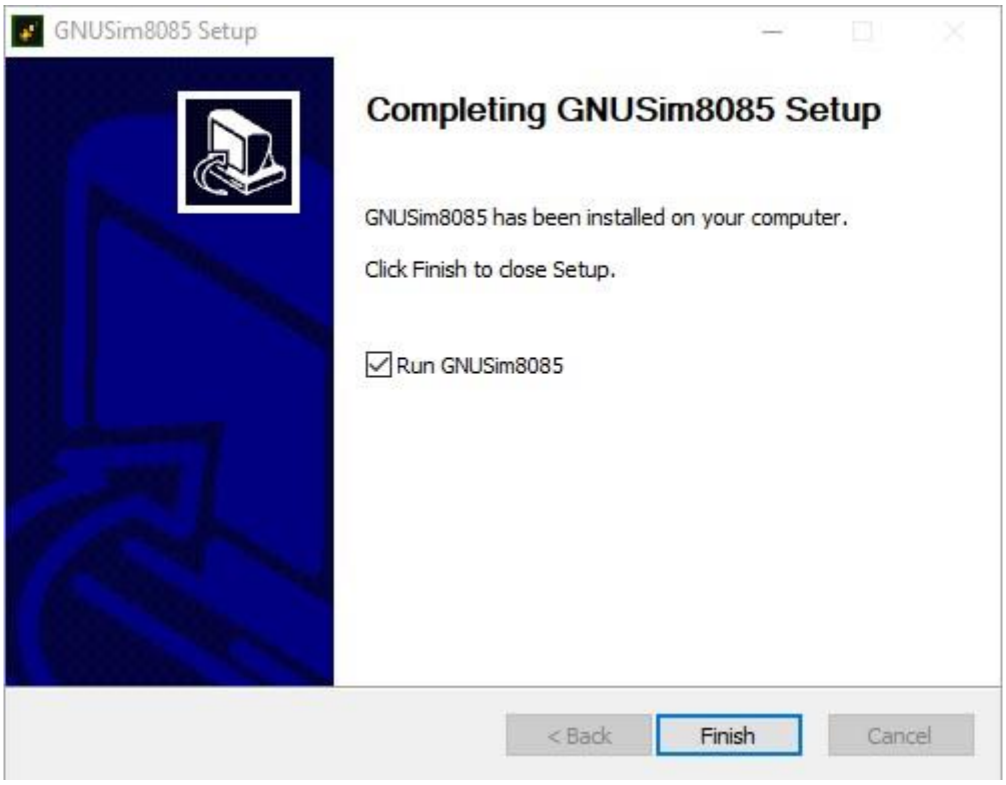

GNUSim8085 Setup

Installing

Please wait while GNUSim8085 is being installed.

Extract: orientation-portrait-symbolic.symbolic.png

6.) Complete Setup of GNUsim8085.
- Click on Finish


GNUSim8085 Setup

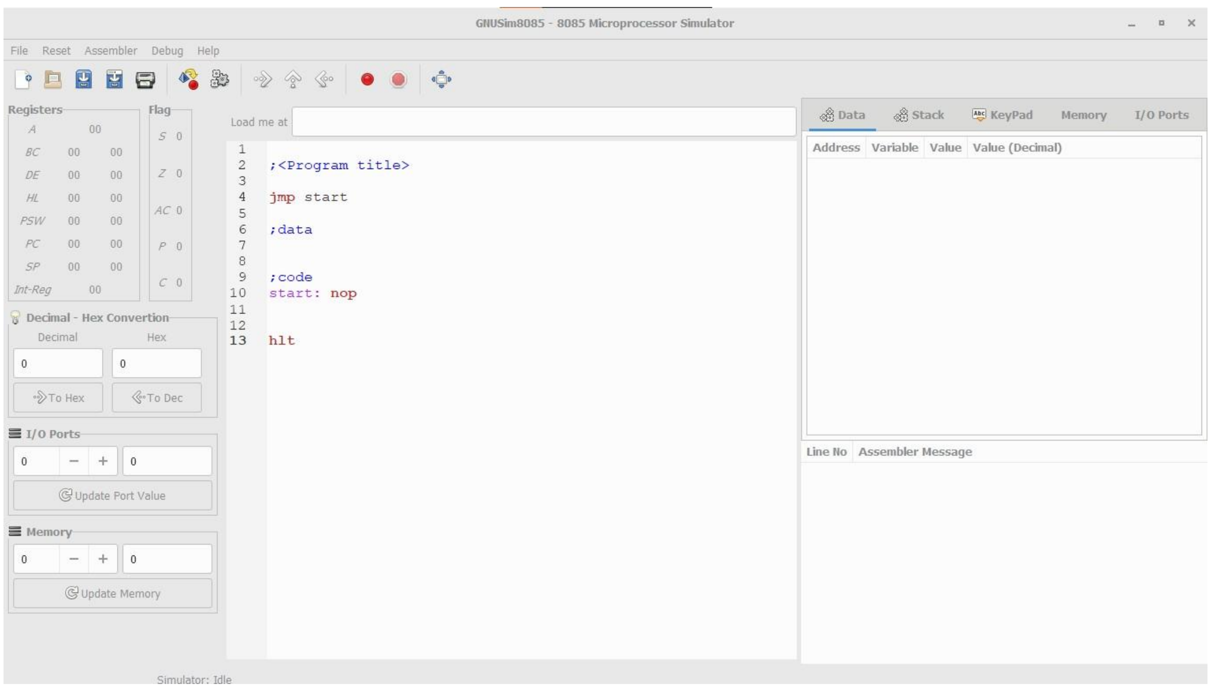
Completing GNUSim8085 Setup

GNUSim8085 has been installed on your computer.

Click Finish to close Setup.

☑ Run GNUSim8085

7.) GNUsim8085 default Screen.


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Registers
Flag
Load me at
Data Stack KeyPad Memory I/O Ports
Address Variable Value Value (Decimal)
Line No Assembler Message
Decimal - Hex Convertion
Decimal Hex
To Hex To Dec
I/O Ports
Update Port Value
Memory
Update Memory
Simulator: Idle

```

;<Program title>

jmp start

;data


;code
start: nop


hlt
```

GNUSim8085 is a 8085 microprocessor simulator with following features.

- A simple editor component with syntax highlighting.
- A keypad to input assembly language instructions with appropriate arguments.
- Easy view of register contents.
- Easy view of flag contents.
- Hexadecimal <--> Decimal converter.
- View of stack, memory and I/O contents.
- Support for breakpoints for programming debugging.
- Stepwise program execution.
- One click conversion of assembly program to opcode listing.
- Printing support (known not to work well on Windows).
- UI translated in various languages

8085 Microprocessor – Functional Units

1. Accumulator

It is an 8-bit register used for general purposes. It also helps in arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU. The data is stored in this register.

2. Arithmetic and logic unit

As the name indicates, it is helpful in arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

3. General purpose register

There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H & L. Each register can hold 8-bit data.These registers can work in pairs in order to hold 16-bit data and their pairing combination looks like B-C, D-E & H-L.

4. Program counter

It is a type of 16-bit register used to store the address of the instructions that is to be executed. Whenever each instruction gets fetched from the program counter its store value is increased by 1.

5. Stack pointer

It is also a 16-bit register works like stack, which is always incremented/ decremented by 2 during push & pop operations.

6. Temporary register

It is an 8-bit register, as the name suggests it holds the temporary data of arithmetic and logical operations.

7. Flag register

It is an 8-bit register which has five 1-bit flip-flops, which contains either 0 or 1 based on the result data that is stored in the accumulator.

These are the set of 5 flip-flops –
Sign (S)
Zero (Z)
Auxiliary Carry (AC)
Parity (P)
Carry (C)
Its bit position is shown in the following table –

D7	D6	D5	D4	D3	D2	D1	D0
S	Z	AC	P	CY			

8. Instruction register and decoder

the instructions that is fetched from the memory. Instruction decoder decodes the information present in the Instruction register.

9. Timing and control unit

It supplies timing and control signal to the microprocessor to perform certain operations. Following are the timing and control signals, which control external and internal circuits –
- Control Signals: READY, RD’, WR’, ALE
- Status Signals: S0, S1, IO/M’
- DMA Signals: HOLD, HLDA
- RESET Signals: RESET IN, RESET OUT

10. Interrupt control

As the name implies it controls the interrupts during a process. When a main program is being executed from the microprocessor some interruptions may arise. So, to avoid this interruption the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control returns back to the main program.
There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

11. Serial Input/output control

It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).

12. Address buffer and address-data buffer

Address buffer is a type of storage that holds the copied data from the memory to make it ready for the execution. In other words it acts like a buffer. It contains the stored content of the stack pointer and program counter in order to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

13. Address bus and data bus

Data bus carries the relevant data that is to be stored. It is bidirectional, whereas address bus signifies the location as to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

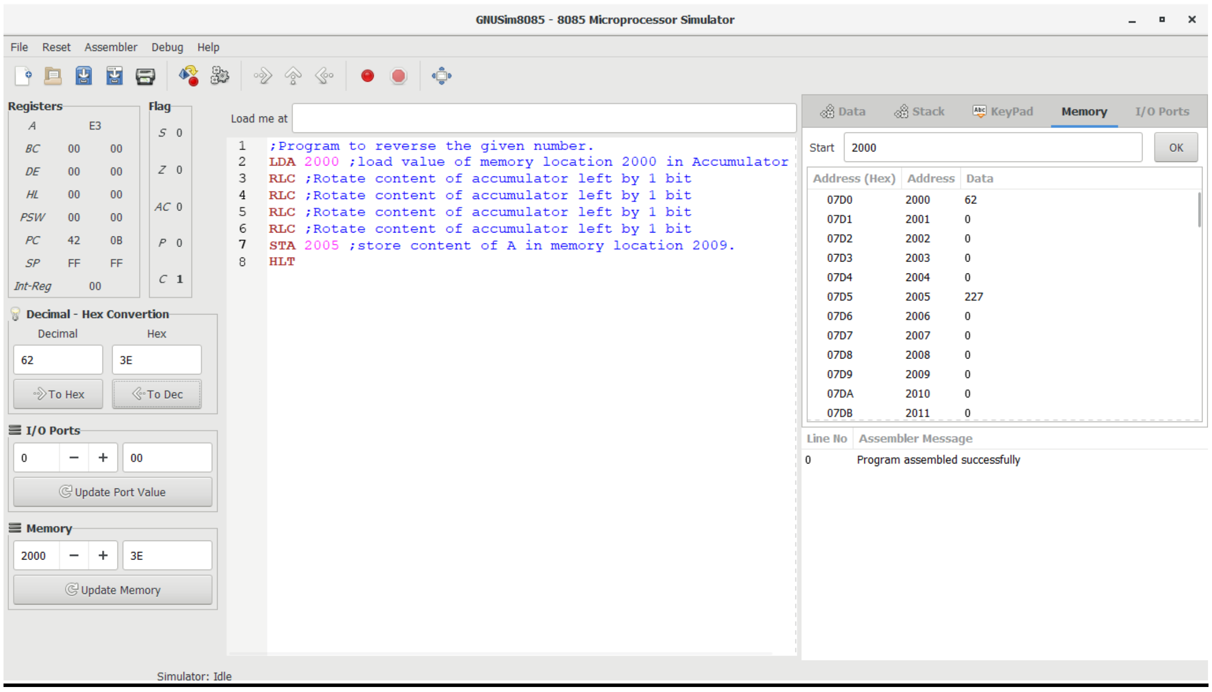
PRACTICAL-2

AIM : Write an assembly language code in GNUsim8085 to store numbers in reverse order in memory location.

PROGRAM:

```
;Program to reverse the given number
LDA 2000H        ;initialize register a with 05
RLC              ;rotate accumulator left
RLC              ;rotate accumulator left
RLC              ;rotate accumulator left
RLC              ;rotate accumulator left
STA 2001H        ;store value of accumulator to 0001H address
HLT
```

OUTPUT:



Registers

A	E3	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0B
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	0
P	0
C	1

Decimal - Hex Convertion

Decimal: 62 Hex: 3E

I/O Ports: 0 00

Memory: 2000 3E

Load me at

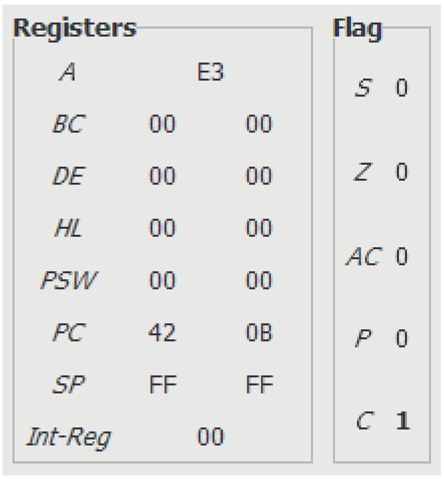
```
;Program to reverse the given number.
LDA 2000 ;load value of memory location 2000 in Accumulator
RLC ;Rotate content of accumulator left by 1 bit
RLC ;Rotate content of accumulator left by 1 bit
RLC ;Rotate content of accumulator left by 1 bit
RLC ;Rotate content of accumulator left by 1 bit
STA 2005 ;store content of A in memory location 2009.
HLT
```

Start 2000

Address (Hex)	Address	Data
07D0	2000	62
07D1	2001	0
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	227
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

Simulator: Idle

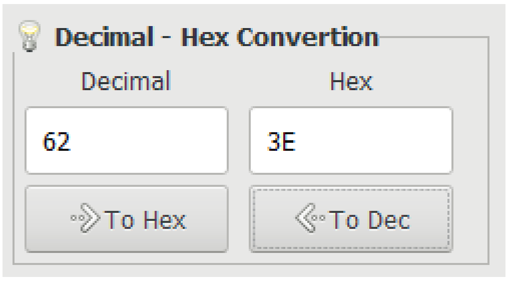


Registers

A	E3	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0B
SP	FF	FF
Int-Reg	00	

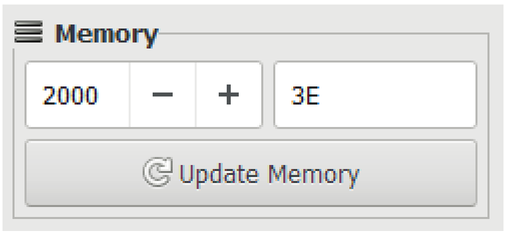
Flag

S	0
Z	0
AC	0
P	0
C	1



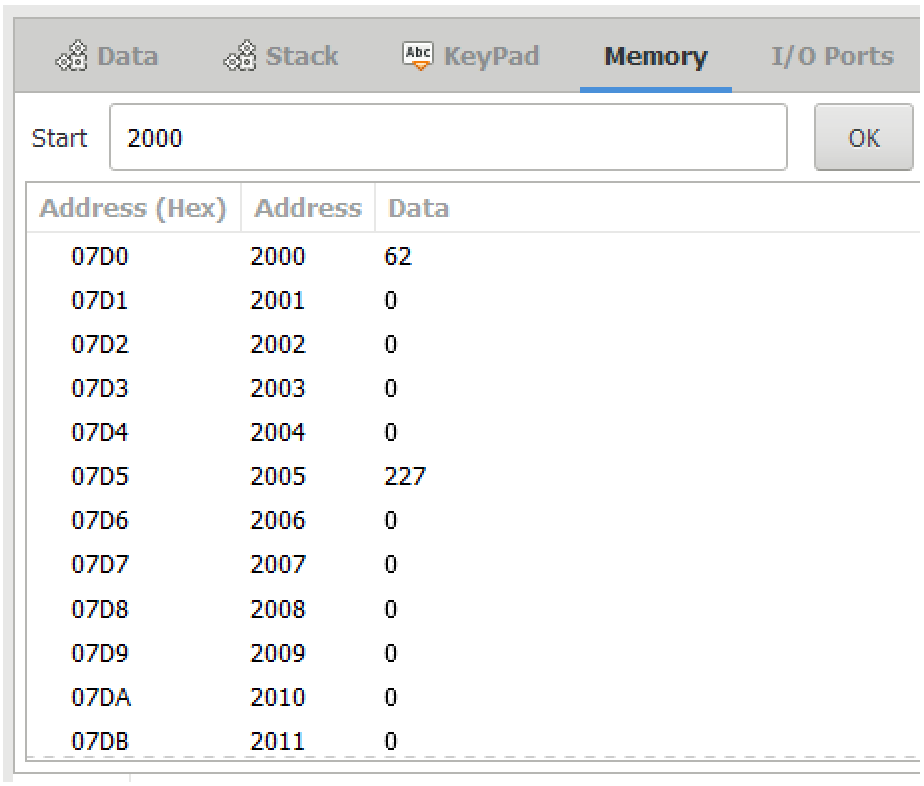
Decimal - Hex Convertion

Decimal	Hex
62	3E



Memory

2000 3E



Start 2000

Address (Hex)	Address	Data
07D0	2000	62
07D1	2001	0
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	227
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

PEN : 200840131012

PRACTICAL-3

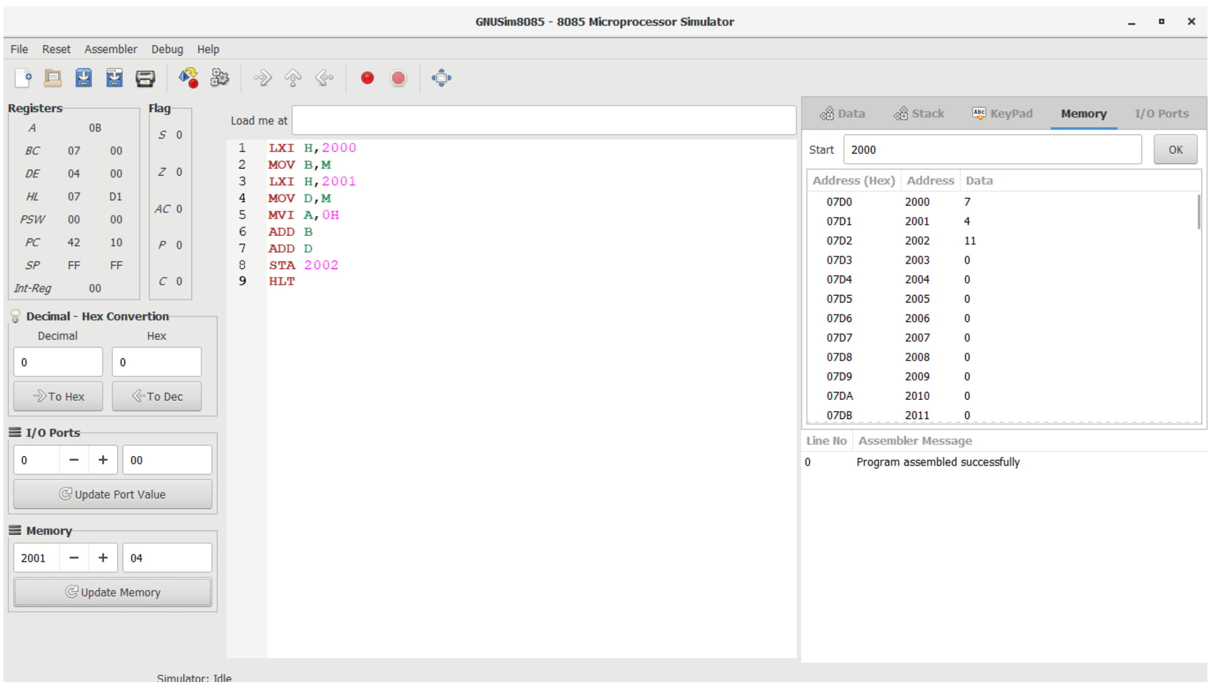
AIM : Write an assembly language code in GNUsim8085 to implement arithmetic instruction. (Addition, Subtraction, Multiplication, Division)

1.) ADDITION:

PROGRAM :

```
LXI H,2000
MOV B,M
LXI H,2001
MOV D,M
MVI A,0H
ADD B
ADD D
STA 2002
HLT
```

OUTPUT :



```
LXI H,2000
MOV B,M
LXI H,2001
MOV D,M
MVI A,0H
ADD B
ADD D
STA 2002
HLT
```

Registers

A	0B	
BC	07	00
DE	04	00
HL	07	D1
PSW	00	00
PC	42	10
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	0
P	0
C	0

Decimal: 0 Hex: 0

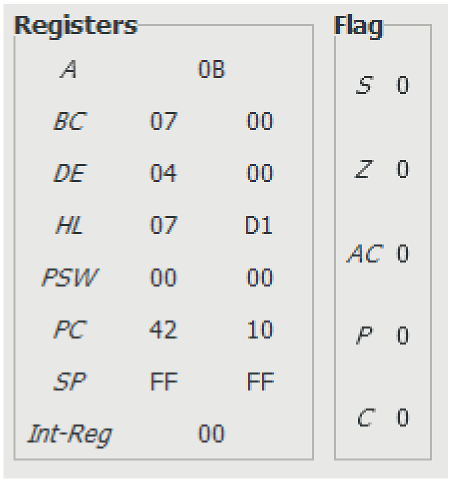
I/O Ports: 0 — 00

Memory: 2001 — 04

Start: 2000

Address (Hex)	Address	Data
07D0	2000	7
07D1	2001	4
07D2	2002	11
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

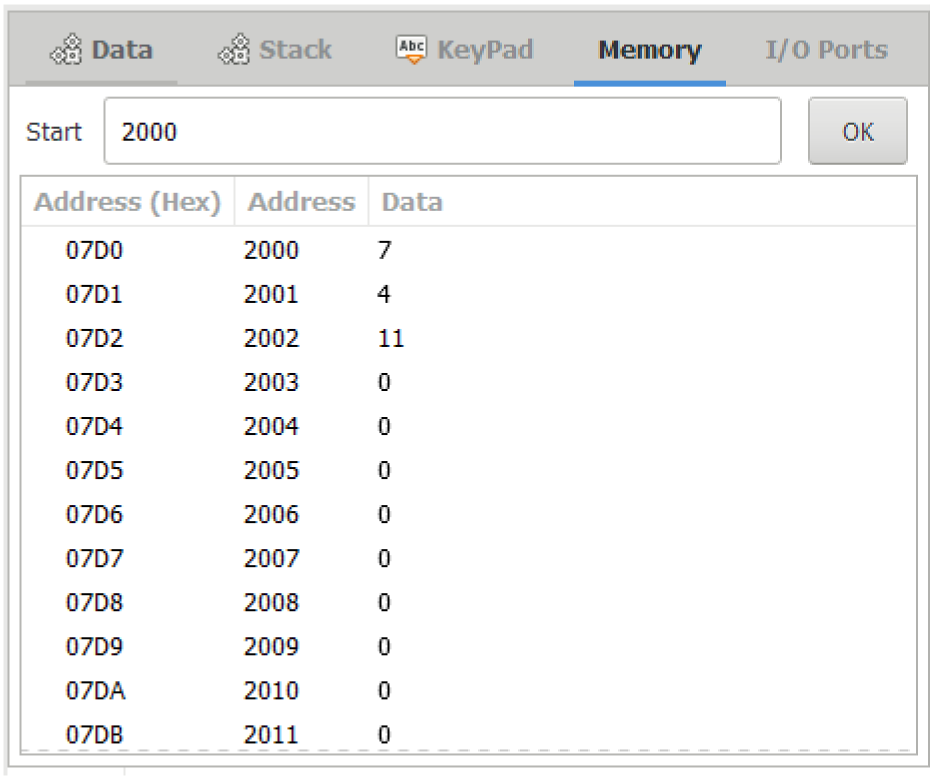


Registers

A	0B	
BC	07	00
DE	04	00
HL	07	D1
PSW	00	00
PC	42	10
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	0
P	0
C	0



Start 2000

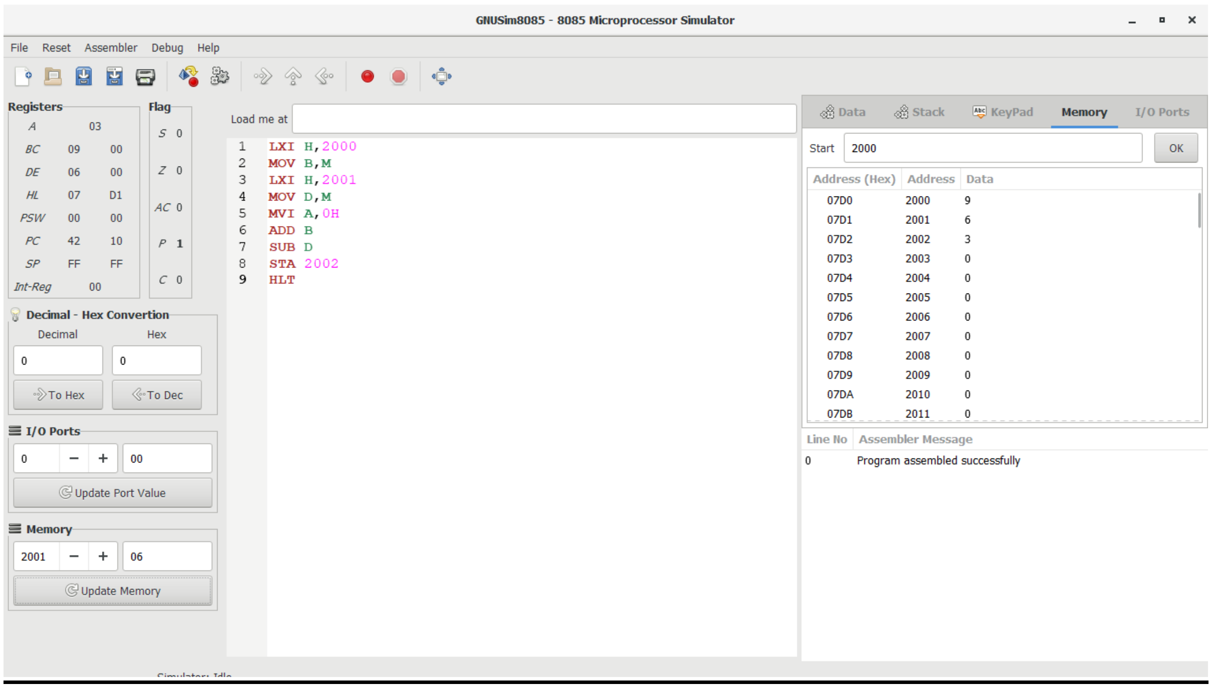
Address (Hex)	Address	Data
07D0	2000	7
07D1	2001	4
07D2	2002	11
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

2.) SUBTRACTION:

PROGRAM:

```
LXI H,2000
MOV B,M
LXI H,2001
MOV D,M
MVI A,0H
ADD B
SUB D
STA 2002
HLT
```

OUTPUT:



Registers

A	03	
BC	09	00
DE	06	00
HL	07	D1
PSW	00	00
PC	42	10
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	0
P	1
C	0

Decimal: 0 Hex: 0

I/O Ports: 0 00

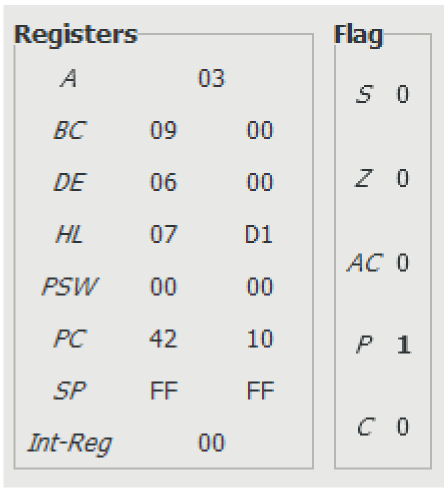
Memory: 2001 06

```
LXI H,2000
MOV B,M
LXI H,2001
MOV D,M
MVI A,0H
ADD B
SUB D
STA 2002
HLT
```

Start: 2000

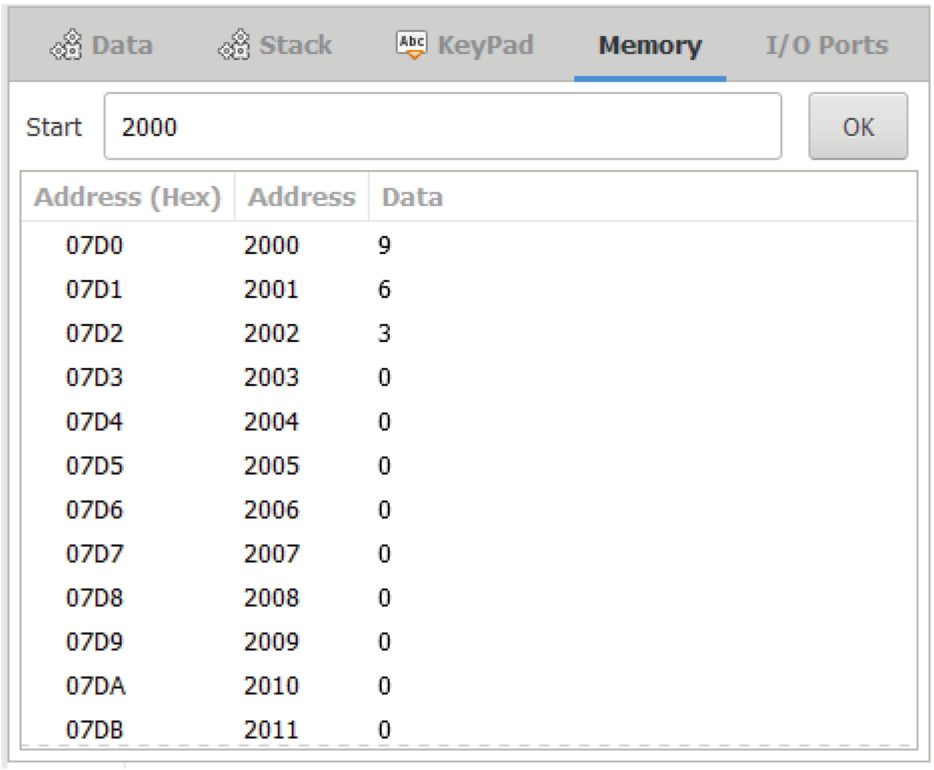
Address (Hex)	Address	Data
07D0	2000	9
07D1	2001	6
07D2	2002	3
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully



Registers		
A	03	
BC	09	00
DE	06	00
HL	07	D1
PSW	00	00
PC	42	10
SP	FF	FF
Int-Reg	00	

Flag	
S	0
Z	0
AC	0
P	1
C	0



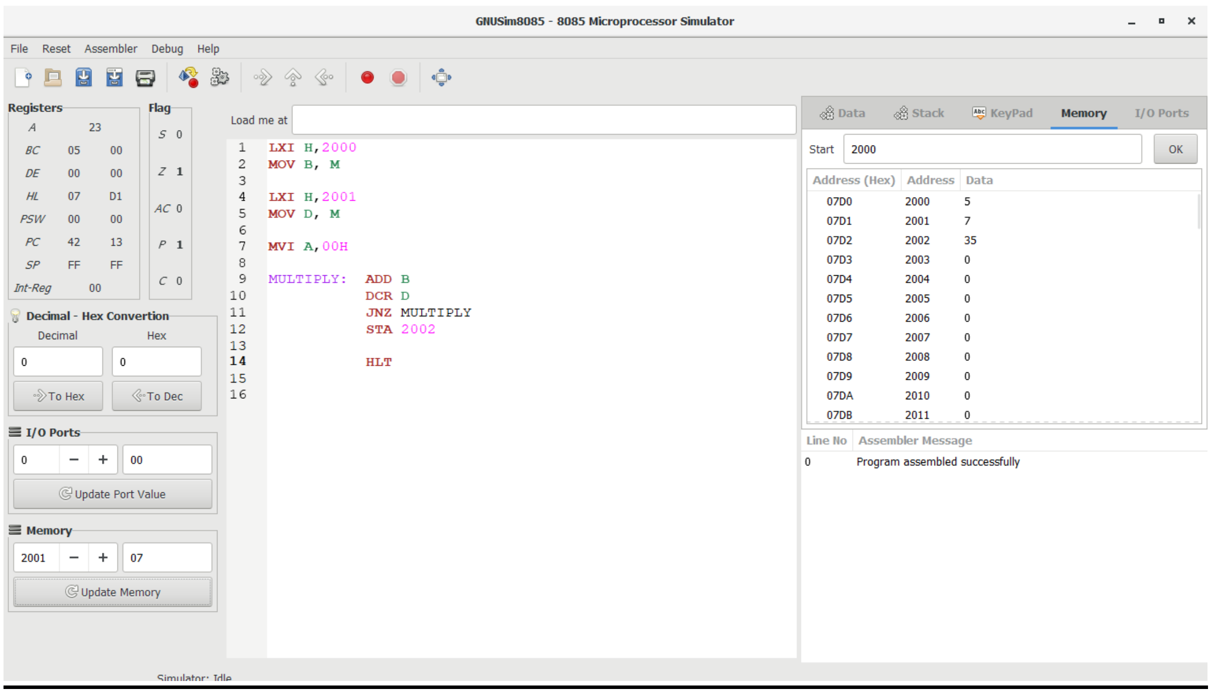
Address (Hex)	Address	Data
07D0	2000	9
07D1	2001	6
07D2	2002	3
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

3.) MULTIPLICATION:

PROGRAM:

```
LXI H,2000
MOV B, M

LXI H,2001
MOV D, M

MVI A,00H

MULTIPLY:  ADD B
           DCR D
           JNZ MULTIPLY
           STA 2002

           HLT
```

OUTPUT:


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Decimal - Hex Convertion
To Hex
To Dec
Update Port Value
Update Memory
Data Stack KeyPad Memory I/O Ports
OK
Simulator: Idle

Registers		
A	23	
BC	05	00
DE	00	00
HL	07	D1
PSW	00	00
PC	42	13
SP	FF	FF
Int-Reg	00	

Flag	
S	0
Z	1
AC	0
P	1
C	0

Decimal: 0 Hex: 0

I/O Ports: 0 00

Memory: 2001 07

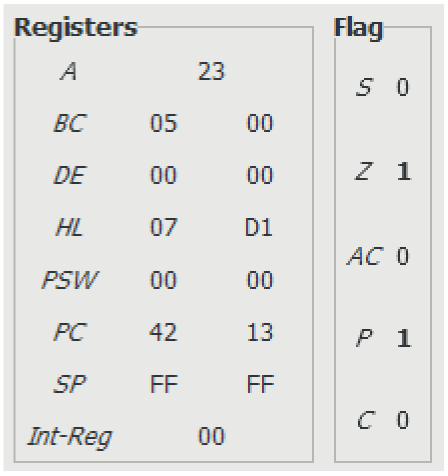
Load me at

```
LXI H,2000
MOV B, M

LXI H,2001
MOV D, M

MVI A,00H

MULTIPLY:  ADD B
           DCR D
           JNZ MULTIPLY
           STA 2002

           HLT
```

Start 2000

Address (Hex)	Address	Data
07D0	2000	5
07D1	2001	7
07D2	2002	35
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

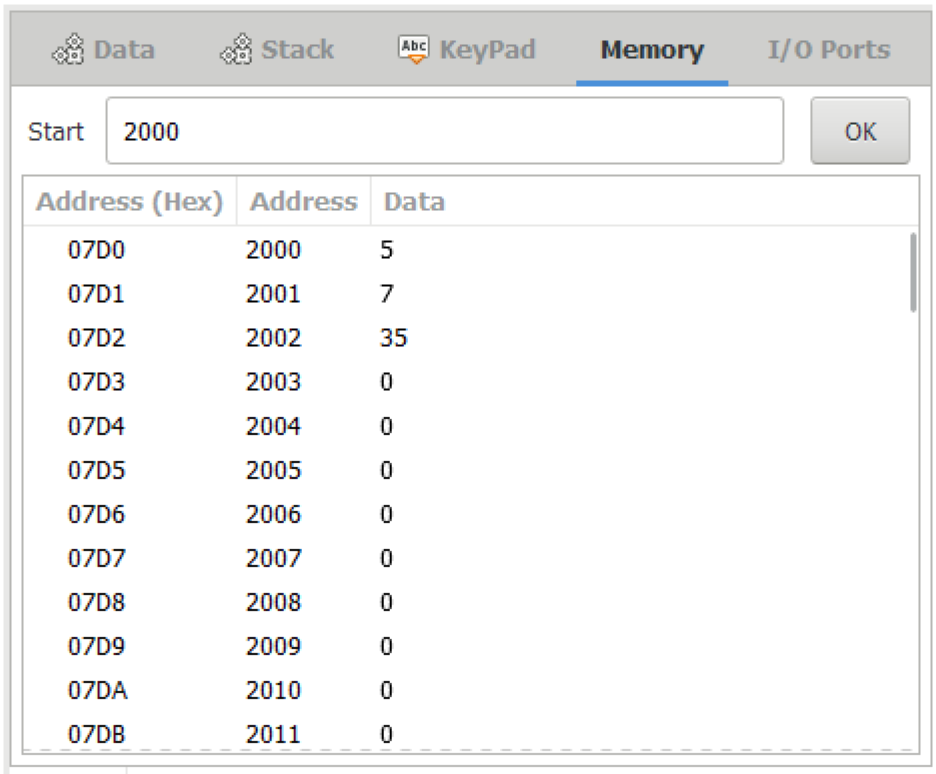


Registers

A	23	
BC	05	00
DE	00	00
HL	07	D1
PSW	00	00
PC	42	13
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0



Start 2000

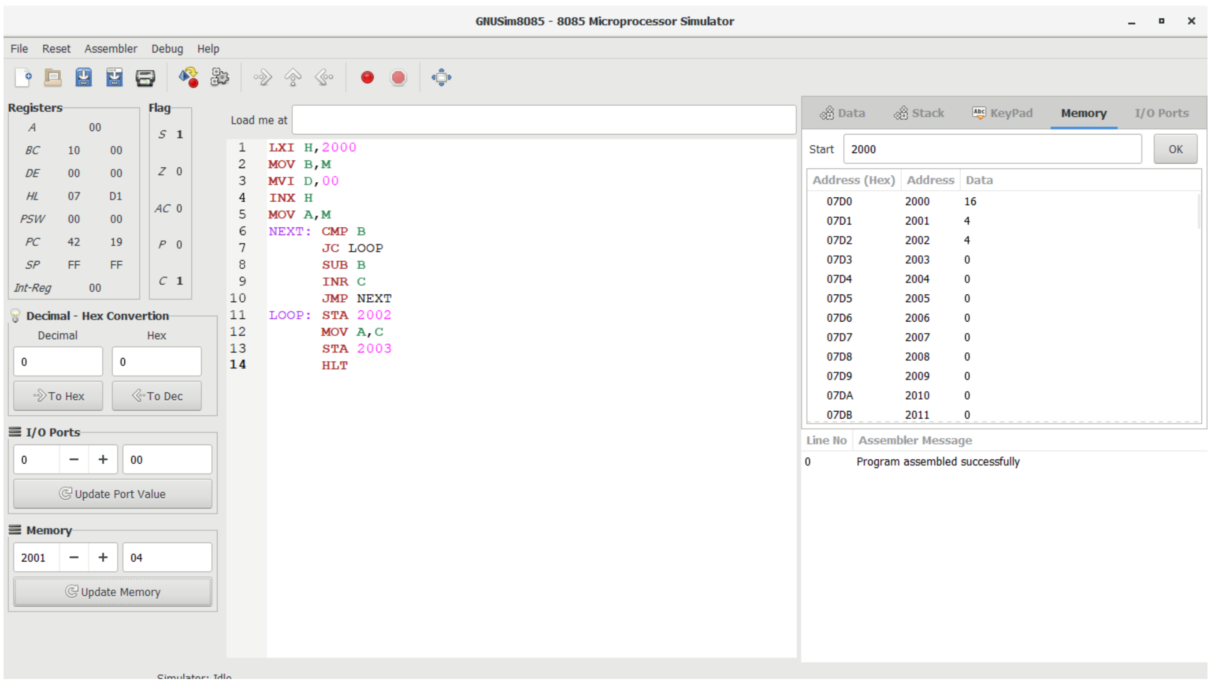
Address (Hex)	Address	Data
07D0	2000	5
07D1	2001	7
07D2	2002	35
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

4.) <u>DIVISION:</u>

<u>PROGRAM:</u>

```
LXI H,2000
MOV B,M
MVI D,00
INX H
MOV A,M
NEXT: CMP B
        JC LOOP
        SUB B
        INR C
        JMP NEXT
LOOP: STA 2002
        MOV A,C
        STA 2003
        HLT
```

<u>OUTPUT:</u>


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Load me at
To Hex
To Dec
Update Port Value
Update Memory
Data Stack KeyPad Memory I/O Ports
OK
Simulator: Idle

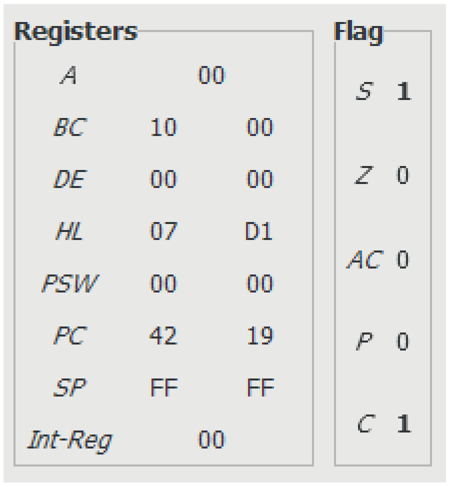
```
Registers
A          00
BC     10     00
DE     00     00
HL     07     D1
PSW    00     00
PC     42     19
SP     FF     FF
Int-Reg    00

Flag
S  1
Z  0
AC 0
P  0
C  1

Decimal - Hex Convertion
Decimal  0    Hex  0

I/O Ports
0   00

Memory
2001   04
```

```
LXI H,2000
MOV B,M
MVI D,00
INX H
MOV A,M
NEXT:  CMP B
       JC LOOP
       SUB B
       INR C
       JMP NEXT
LOOP:  STA 2002
       MOV A,C
       STA 2003
       HLT
```

Start 2000

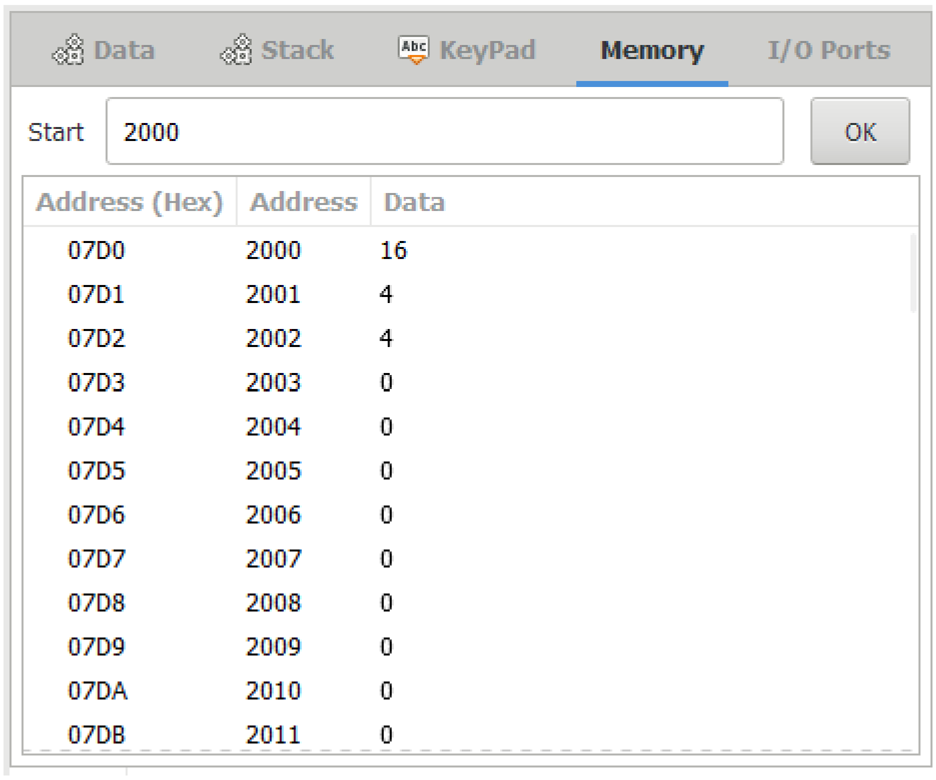
Address (Hex)	Address	Data
07D0	2000	16
07D1	2001	4
07D2	2002	4
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully



Registers		
A	00	
BC	10	00
DE	00	00
HL	07	D1
PSW	00	00
PC	42	19
SP	FF	FF
Int-Reg	00	

Flag	
S	1
Z	0
AC	0
P	0
C	1



Start 2000

Address (Hex)	Address	Data
07D0	2000	16
07D1	2001	4
07D2	2002	4
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

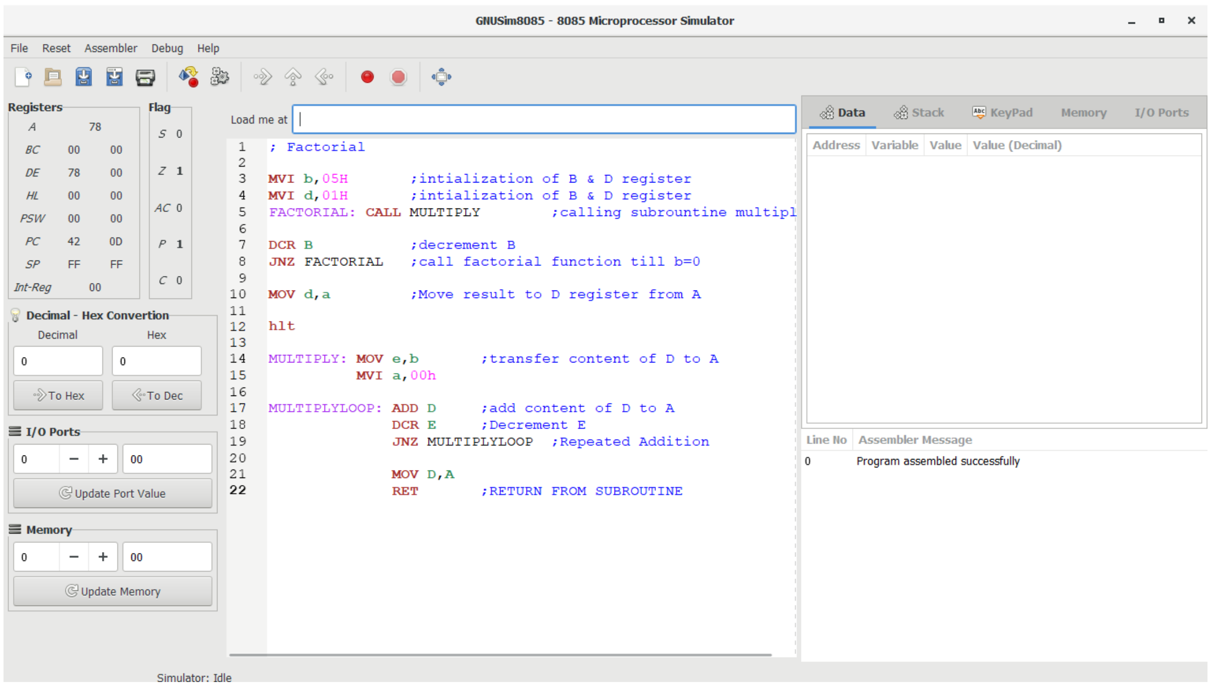
PRACTICAL-4

AIM : Write an assembly language code in GNUsim8085 to find the factorial of a number.

PROGRAM :

```
; Factorial


MVI b,05H                   ;intialization of B & D register
MVI d,01H                   ;intialization of B & D register
FACTORIAL: CALL MULTIPLY    ;calling subrountine multiply


DCR B                       ;decrement B
JNZ FACTORIAL               ;call factorial function till b=0


MOV d,a                     ;Move result to D register from A


hlt


MULTIPLY: MOV e,b           ;transfer content of D to A
          MVI a,00h


MULTIPLYLOOP: ADD D         ;add content of D to A
            DCR E           ;Decrement E
            JNZ MULTIPLYLOOP ;Repeated Addition


            MOV D,A
            RET             ;RETURN FROM SUBROUTINE
```

OUTPUT :



GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Data Stack KeyPad Memory I/O Ports
Address Variable Value Value (Decimal)
To Hex
To Dec
Update Port Value
Update Memory

Registers

A	78	
BC	00	00
DE	78	00
HL	00	00
PSW	00	00
PC	42	0D
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0

Decimal - Hex Convertion

Decimal: 0 Hex: 0

I/O Ports: 0 00

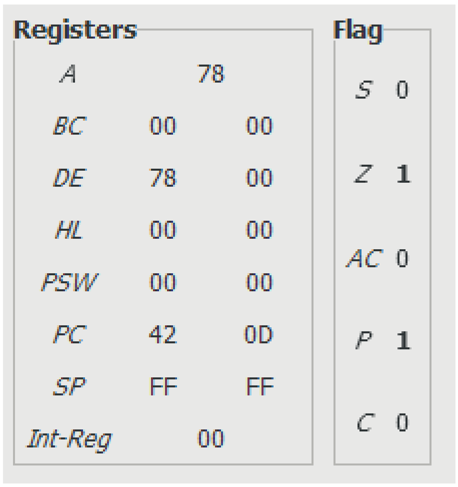
Memory: 0 00

Load me at

```
; Factorial

MVI b,05H        ;intialization of B & D register
MVI d,01H        ;intialization of B & D register
FACTORIAL: CALL MULTIPLY        ;calling subrountine multipl

DCR B            ;decrement B
JNZ FACTORIAL    ;call factorial function till b=0

MOV d,a          ;Move result to D register from A

hlt

MULTIPLY: MOV e,b        ;transfer content of D to A
          MVI a,00h

MULTIPLYLOOP: ADD D      ;add content of D to A
              DCR E      ;Decrement E
              JNZ MULTIPLYLOOP  ;Repeated Addition

              MOV D,A
              RET        ;RETURN FROM SUBROUTINE
```

Line No	Assembler Message
0	Program assembled successfully

Simulator: Idle

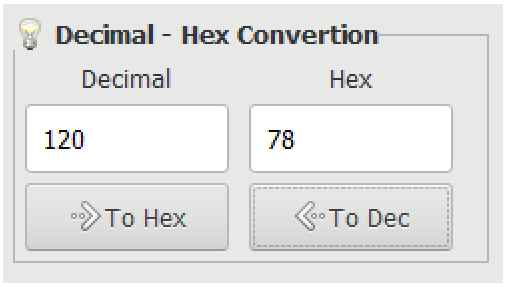


Registers

A	78	
BC	00	00
DE	78	00
HL	00	00
PSW	00	00
PC	42	0D
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0



Decimal - Hex Convertion

Decimal	Hex
120	78

PRACTICAL-5

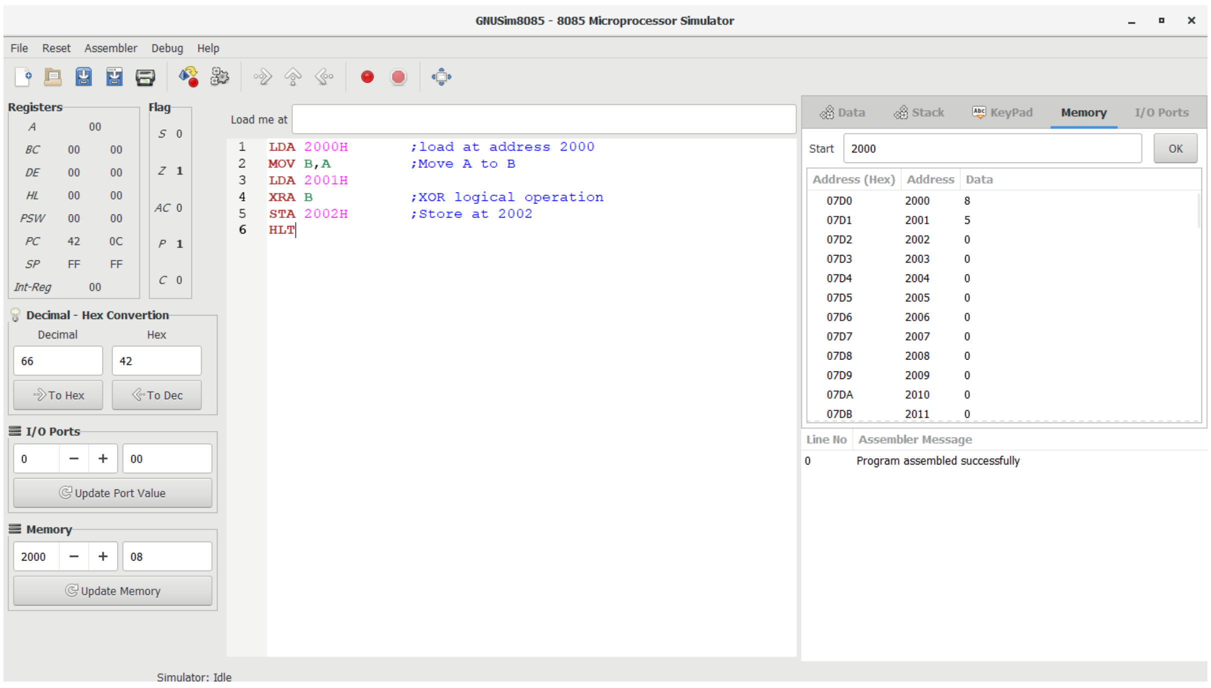
AIM : Write an assembly language code in GNUsim8085 to implement logical instructions.

XOR OPERATION:

PROGRAM:

```
LDA 2000H       ;load at address 2000
MOV B,A         ;Move A to B
LDA 2001H
XRA B           ;XOR logical operation
STA 2002H       ;Store at 2002
HLT
```

OUTPUT:



```
LDA 2000H       ;load at address 2000
MOV B,A         ;Move A to B
LDA 2001H
XRA B           ;XOR logical operation
STA 2002H       ;Store at 2002
HLT
```

Register		
A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag	
S	0
Z	1
AC	0
P	1
C	0

Decimal: 66 Hex: 42

I/O Ports: 0 00

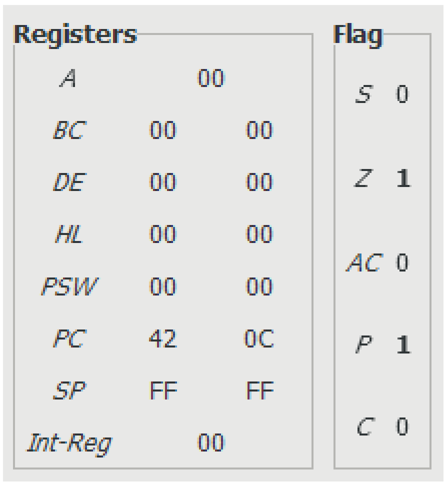
Memory: 2000 08

Start: 2000

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

PEN : 200840131012

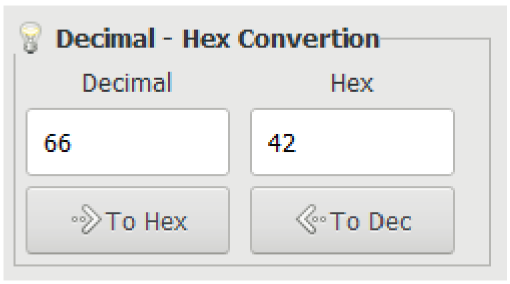


Registers

A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

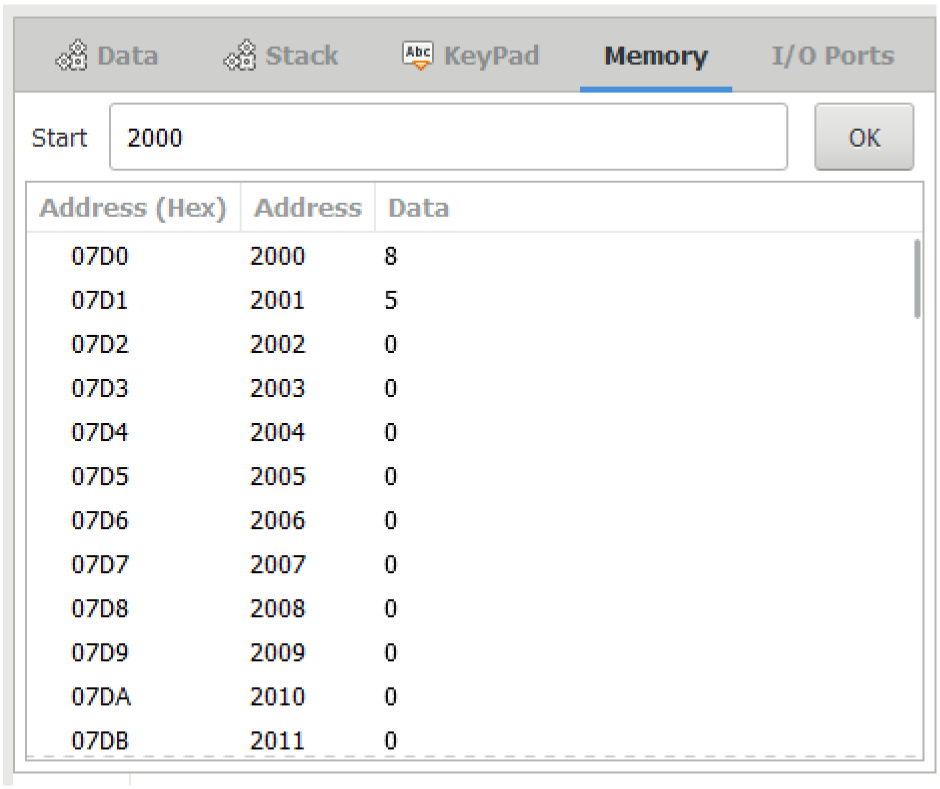
Flag

S	0
Z	1
AC	0
P	1
C	0



Decimal - Hex Convertion

Decimal: 66
Hex: 42



Start: 2000

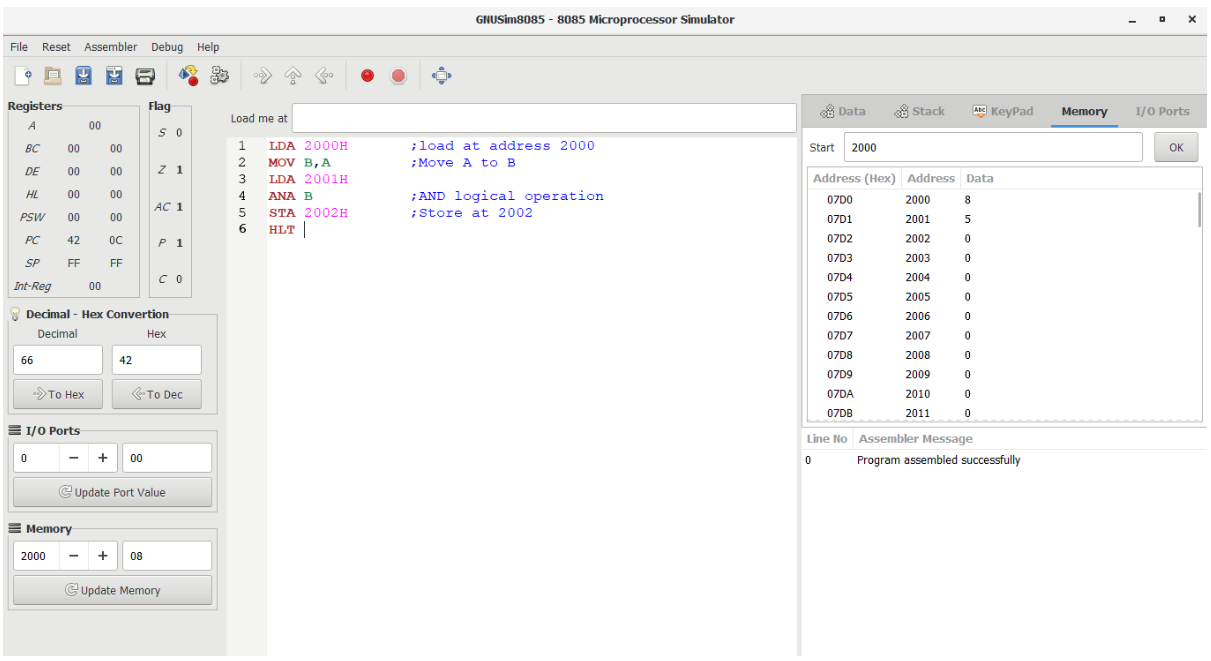
Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

AND OPERATION:

PROGRAM:

```
LDA 2000H       ;load at address 2000
MOV B,A         ;Move A to B
LDA 2001H
ANA B           ;AND logical operation
STA 2002H       ;Store at 2002
HLT
```

OUTPUT:



```
LDA 2000H       ;load at address 2000
MOV B,A         ;Move A to B
LDA 2001H
ANA B           ;AND logical operation
STA 2002H       ;Store at 2002
HLT
```

Registers		
A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag	
S	0
Z	1
AC	1
P	1
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42

I/O Ports: 0 — 00

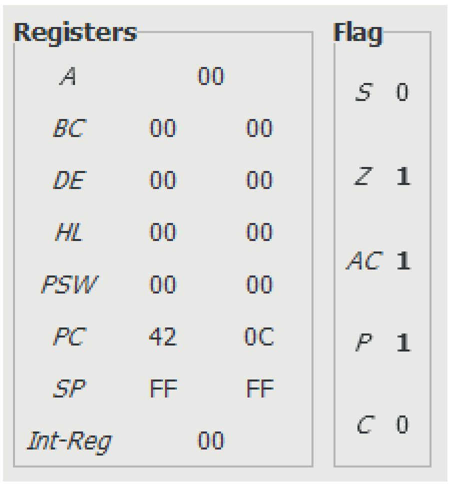
Memory: 2000 — 08

Start 2000

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

PEN : 200840131012

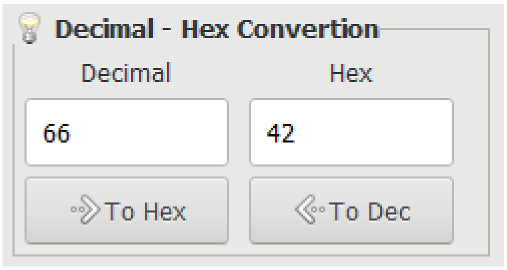


Registers

A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

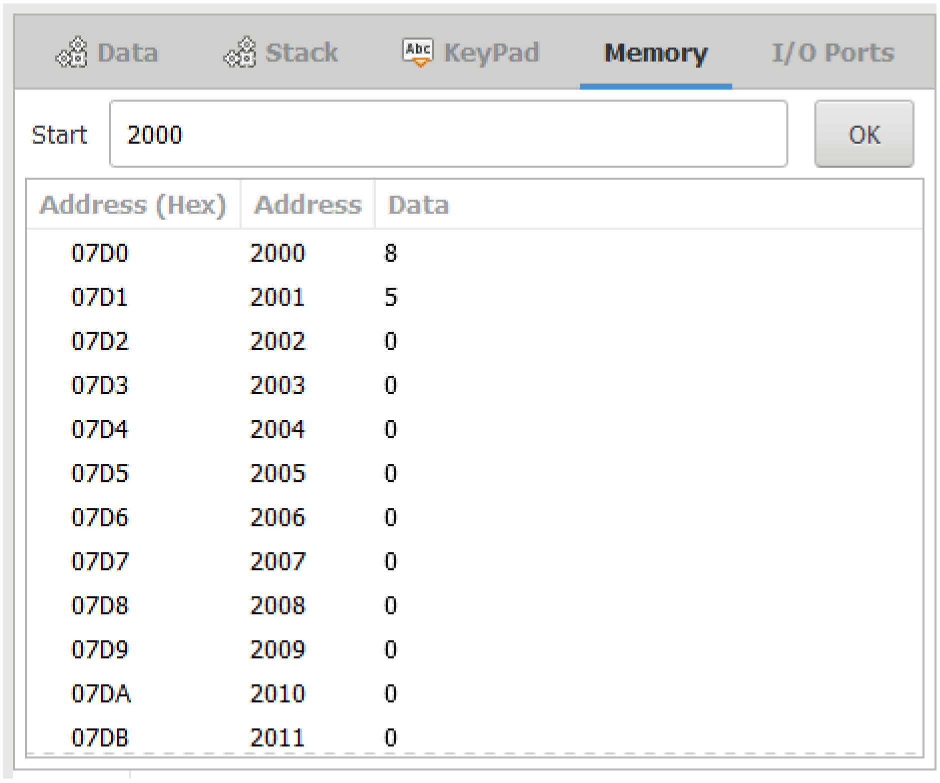
Flag

S	0
Z	1
AC	1
P	1
C	0



Decimal - Hex Convertion

Decimal	Hex
66	42



Start 2000

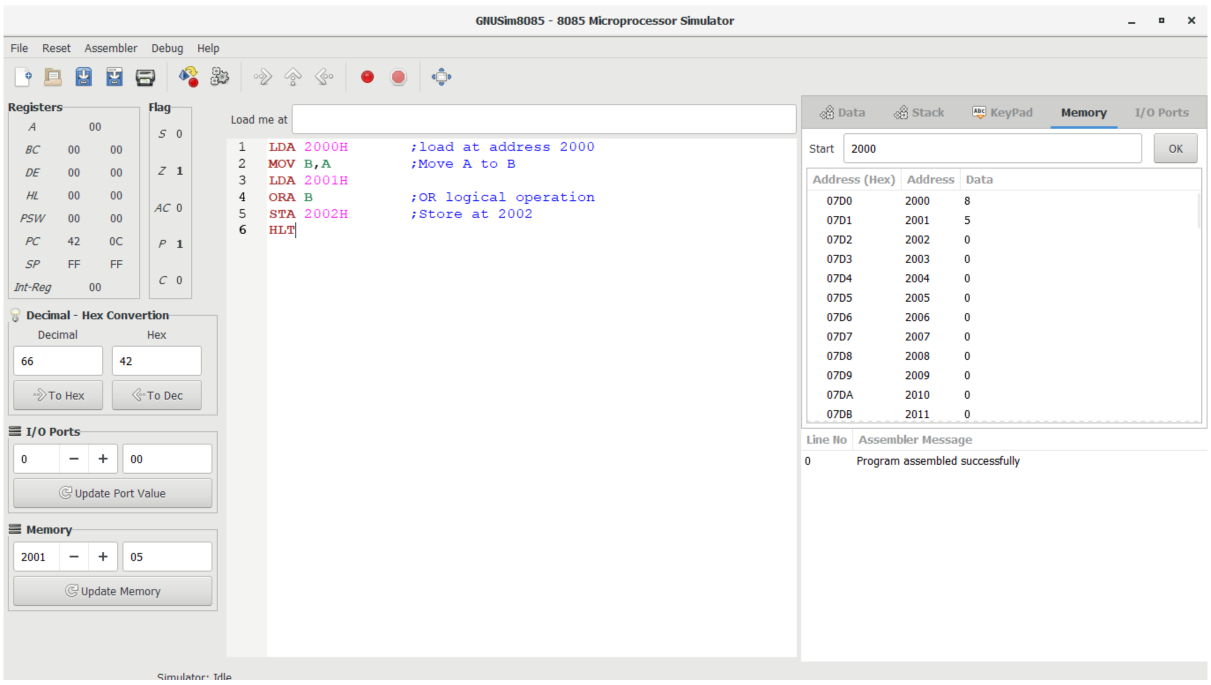
Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

OR OPERATION:

PROGRAM:

```
LDA 2000H        ;load at address 2000
MOV B,A          ;Move A to B
LDA 2001H
ORA B            ;OR logical operation
STA 2002H        ;Store at 2002
HLT
```

OUTPUT:


GNUSim8085 - 8085 Microprocessor Simulator
File Reset Assembler Debug Help
Data Stack KeyPad Memory I/O Ports
Load me at
To Hex To Dec
Update Port Value
Update Memory
OK
Simulator: Idle

```
LDA 2000H        ;load at address 2000
MOV B,A          ;Move A to B
LDA 2001H
ORA B            ;OR logical operation
STA 2002H        ;Store at 2002
HLT
```

Registers

A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42

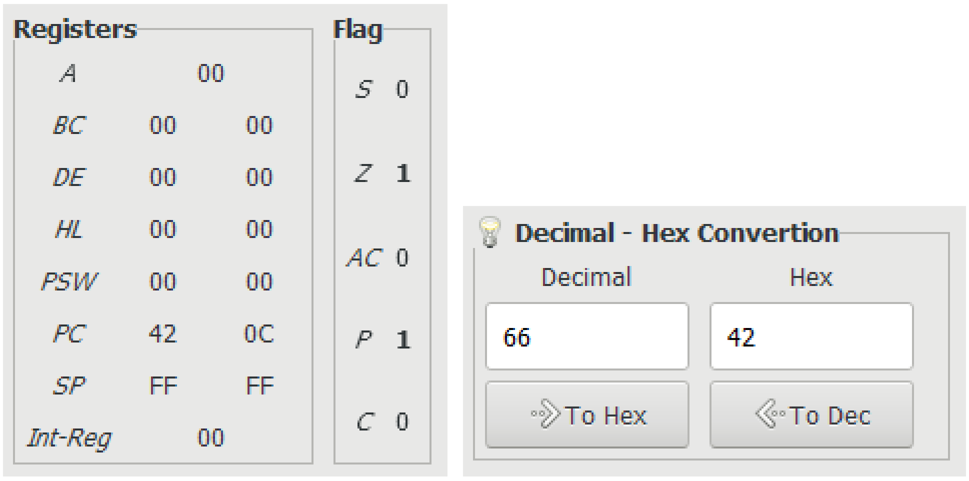
I/O Ports: 0 — 00

Memory: 2001 — 05

Start 2000

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully



Registers

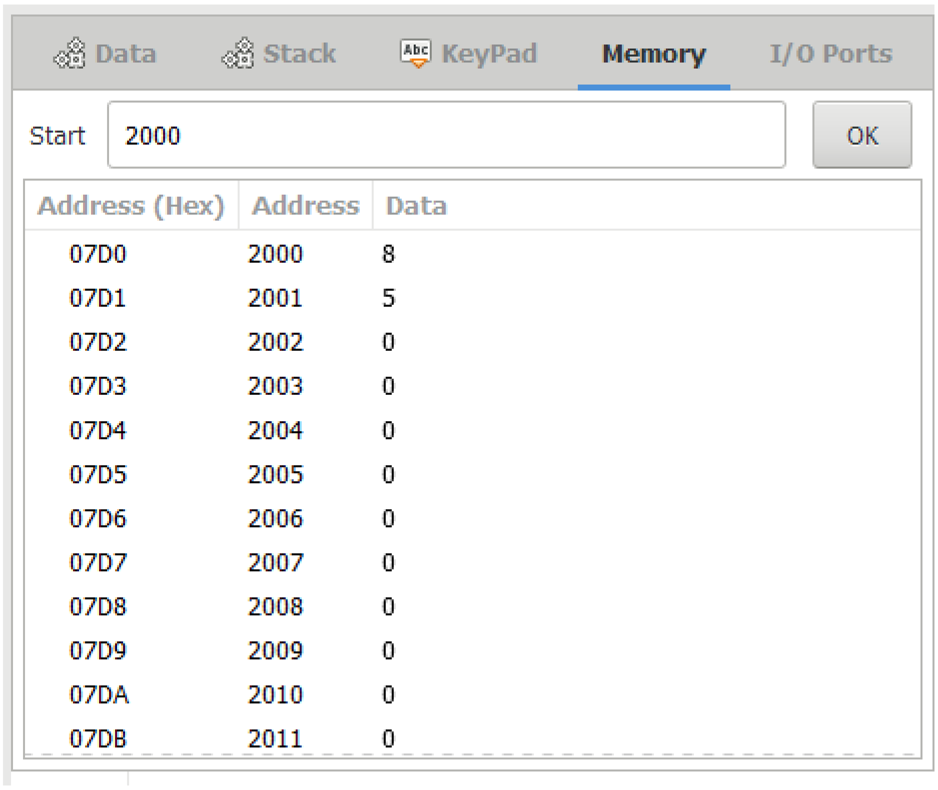
A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42



Start 2000

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

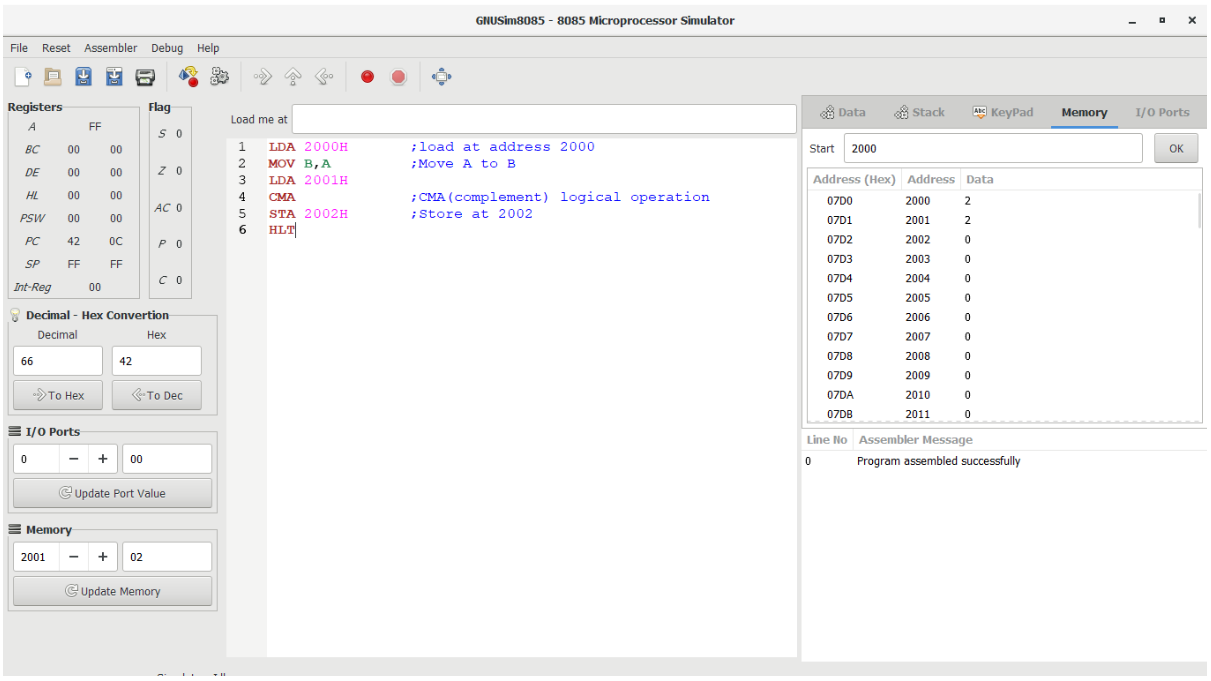
PEN : 200840131012

COMPLEMENT OPERATION:

PROGRAM:

```
LDA 2000H        ;load at address 2000
MOV B,A          ;Move A to B
LDA 2001H
CMA              ;CMA(complement) logical operation
STA 2002H        ;Store at 2002
HLT
```

OUTPUT:



GNUSim8085 - 8085 Microprocessor Simulator

```
LDA 2000H        ;load at address 2000
MOV B,A          ;Move A to B
LDA 2001H
CMA              ;CMA(complement) logical operation
STA 2002H        ;Store at 2002
HLT
```

Registers

A	FF	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	0
AC	0
P	0
C	0

Decimal - Hex Convertion: Decimal 66, Hex 42

I/O Ports: 0, 00

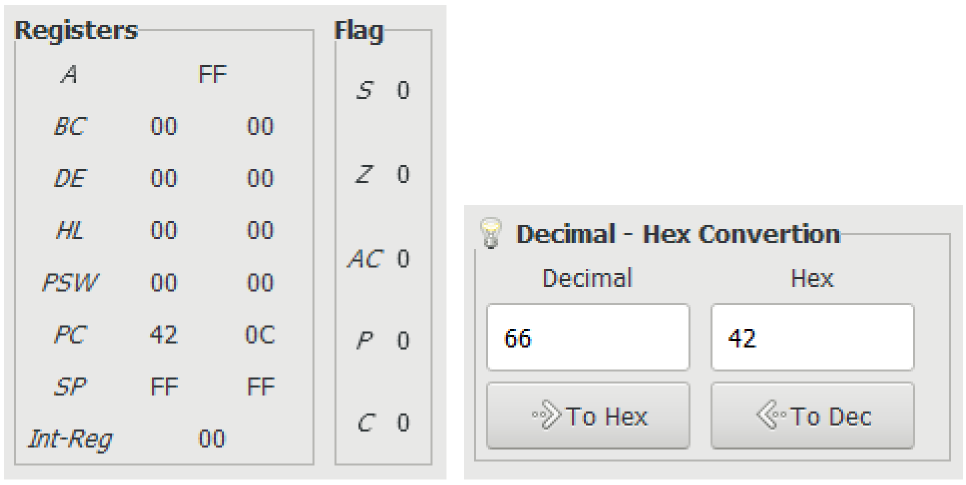
Memory: 2001, 02

Memory — Start 2000

Address (Hex)	Address	Data
07D0	2000	2
07D1	2001	2
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

PEN : 200840131012

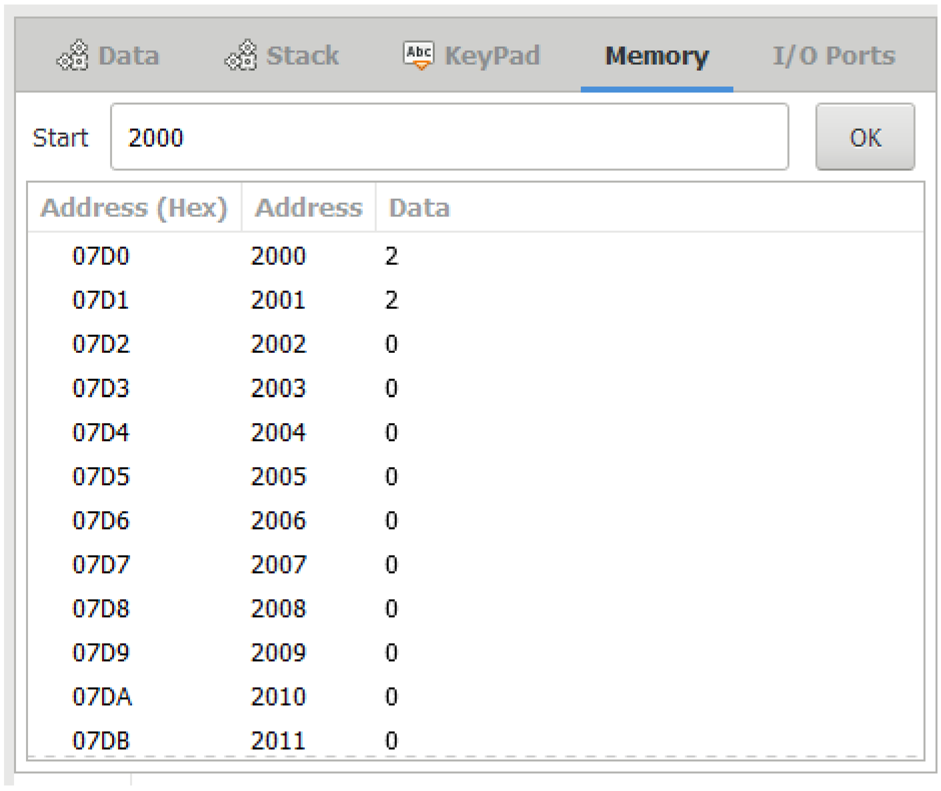


Registers		
A	FF	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag	
S	0
Z	0
AC	0
P	0
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42



Start 2000

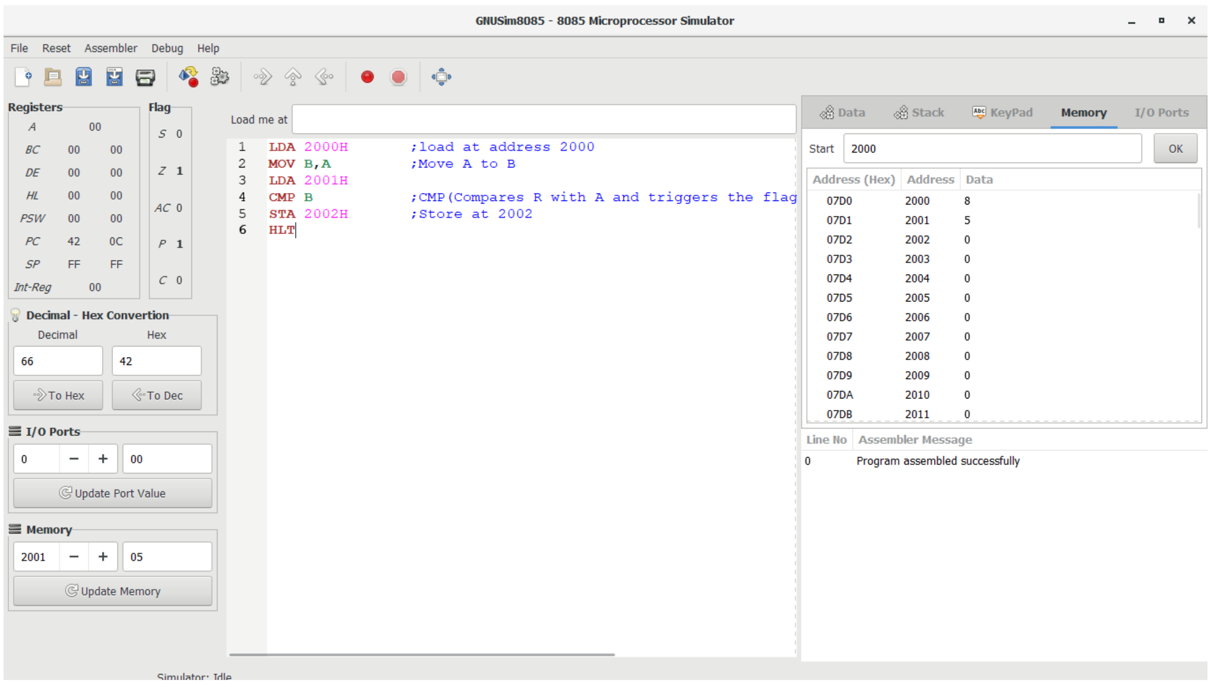
Address (Hex)	Address	Data
07D0	2000	2
07D1	2001	2
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

COMPARISON OPERATION:

PROGRAM:

LDA 2000H	;load at address 2000
MOV B,A	;Move A to B
LDA 2001H	
CMP B	;CMP(Compares R with A and triggers the flag register)logical operation
STA 2002H	;Store at 2002
HLT	

OUTPUT:



```
LDA 2000H        ;load at address 2000
MOV B,A          ;Move A to B
LDA 2001H
CMP B            ;CMP(Compares R with A and triggers the flag
STA 2002H        ;Store at 2002
HLT
```

Registers

A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42

I/O Ports: 0 — 00

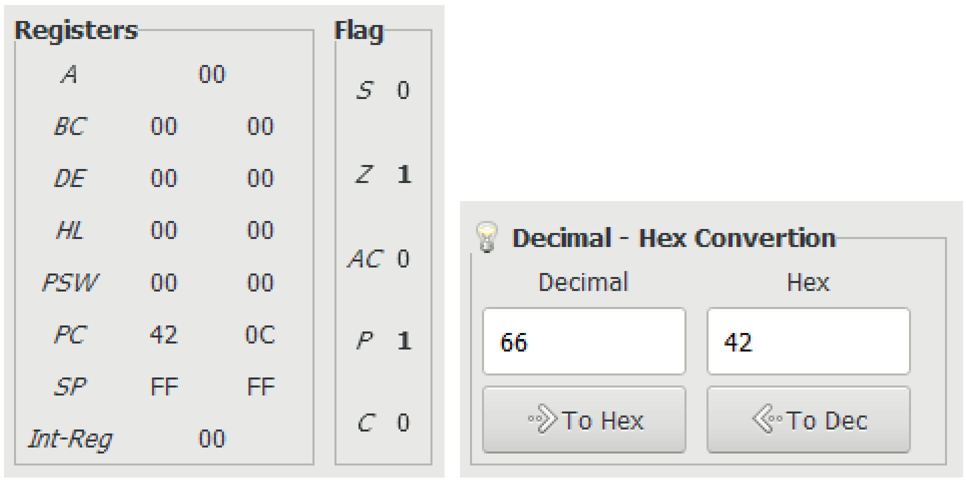
Memory: 2001 — 05

Start 2000 OK

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

Line No	Assembler Message
0	Program assembled successfully

PEN : 200840131012



Registers

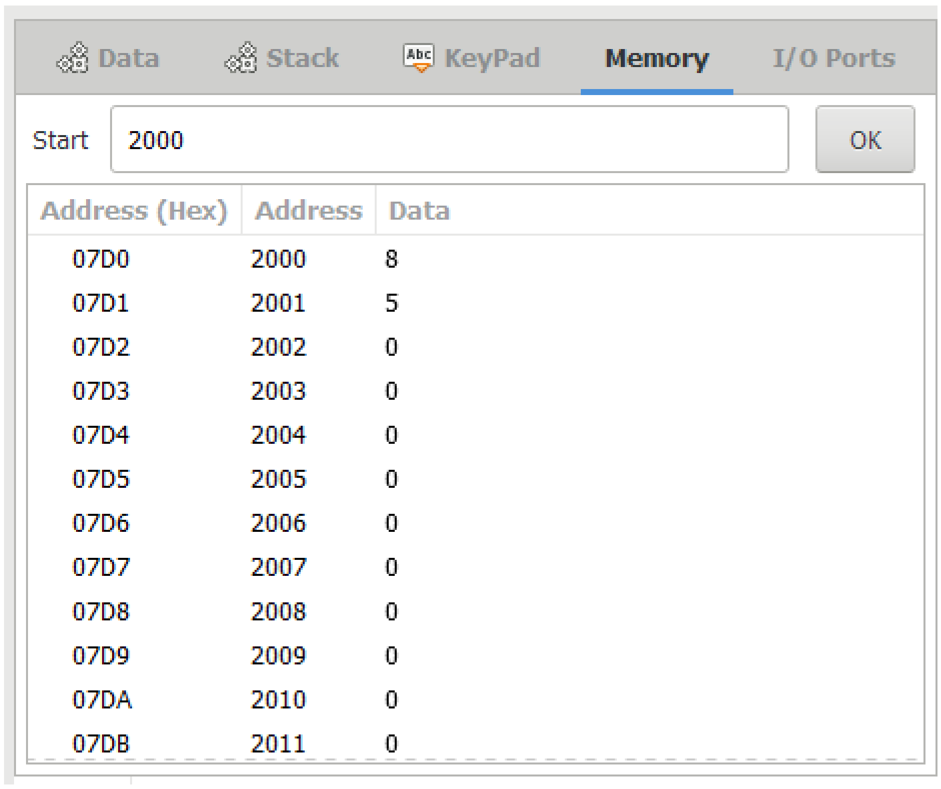
A	00	
BC	00	00
DE	00	00
HL	00	00
PSW	00	00
PC	42	0C
SP	FF	FF
Int-Reg	00	

Flag

S	0
Z	1
AC	0
P	1
C	0

Decimal - Hex Convertion

Decimal	Hex
66	42



Start 2000

Address (Hex)	Address	Data
07D0	2000	8
07D1	2001	5
07D2	2002	0
07D3	2003	0
07D4	2004	0
07D5	2005	0
07D6	2006	0
07D7	2007	0
07D8	2008	0
07D9	2009	0
07DA	2010	0
07DB	2011	0

PRACTICAL-6

AIM : Implement Booth's Algorithm.

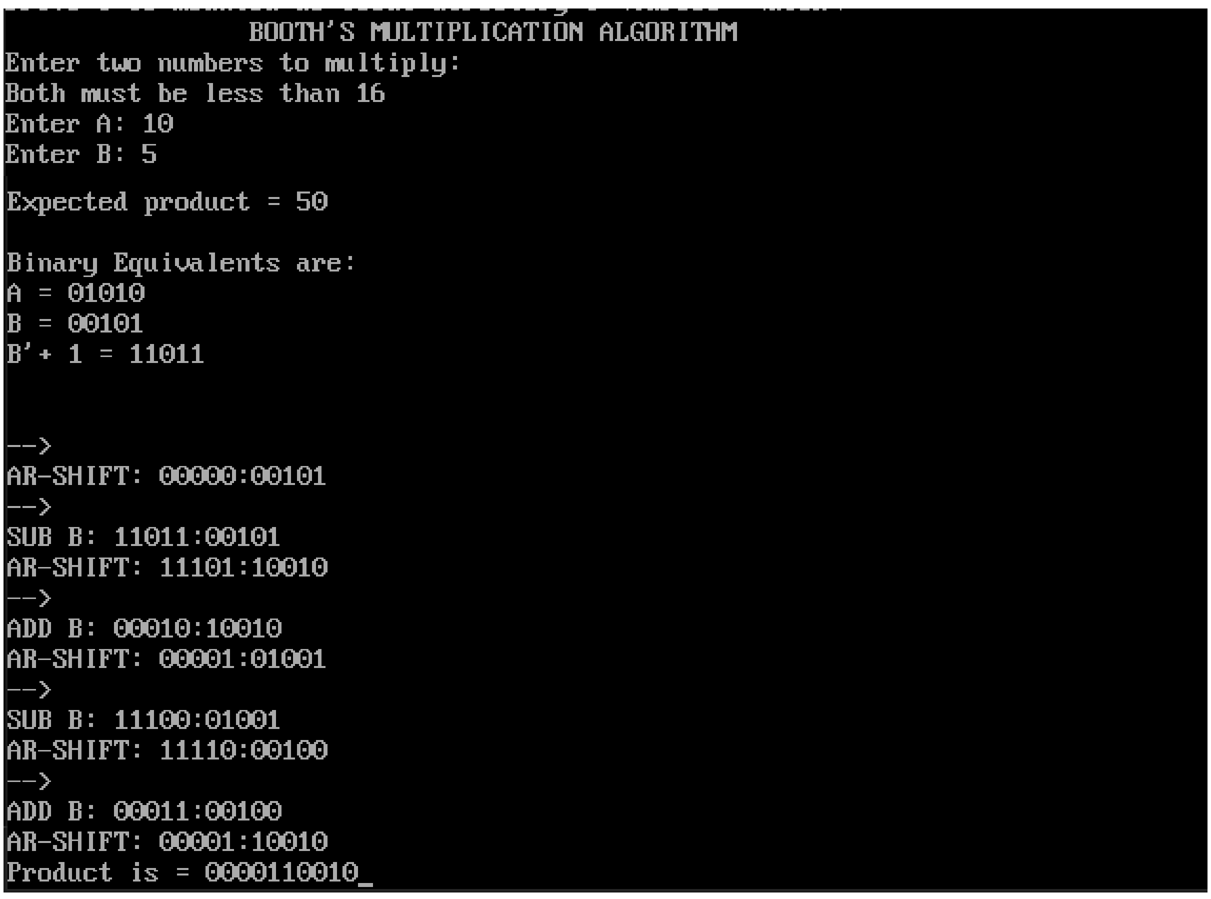
PROGRAM:

```
#include <stdio.h>
#include <math.h>
int a = 0,b = 0, c = 0, a1 = 0, b1 = 0, com[5] = { 1, 0, 0, 0, 0};
int anum[5] = {0}, anumcp[5] = {0}, bnum[5] = {0};
int acomp[5] = {0}, bcomp[5] = {0}, pro[5] = {0}, res[5] = {0};
void binary()
{
        a1 = fabs(a);
        b1 = fabs(b);
        int r, r2, i, temp;
        for (i = 0; i < 5; i++)
        {
                r = a1 % 2;
                a1 = a1 / 2;
                r2 = b1 % 2;
                b1 = b1 / 2;
                anum[i] = r;
                anumcp[i] = r;
                bnum[i] = r2;
                if(r2 == 0)
                {
                        bcomp[i] = 1;
                }
                if(r == 0)
                {
                        acomp[i] =1;
                }
        }
        //part for two's complementing
        c = 0;
        for ( i = 0; i < 5; i++)
        {
                res[i] = com[i]+ bcomp[i] + c;
                if(res[i] >= 2)
                {
                        c = 1;
                }
                else
                c = 0;
                res[i] = res[i] % 2;
        }
        for (i = 4; i >= 0; i--)
        {
                bcomp[i] = res[i];
        }
        //in case of negative inputs
        if (a < 0)
        {
                c = 0;
                for (i = 4; i >= 0; i--)
                {
                        res[i] = 0;
                }
                for ( i = 0; i < 5; i++)
                {
                        res[i] = com[i] + acomp[i] + c;
                        if (res[i] >= 2)
                        {
                                c = 1;
                        }
                        else
                        c = 0;
                        res[i] = res[i]%2;
                }
                for (i = 4; i >= 0; i--)
                {
                        anum[i] = res[i];
                        anumcp[i] = res[i];
                }
        }
        if(b < 0)
        {
                for (i = 0; i < 5; i++)
                {
                        temp = bnum[i];
                        bnum[i] = bcomp[i];
                        bcomp[i] = temp;

                }
        }
}
void add(int num[])
{
        int i;
        c = 0;
        for ( i = 0; i < 5; i++)
        {
                res[i] = pro[i] + num[i] + c;
                if (res[i] >= 2)
                {
                        c = 1;
                }
                else
                {
                        c = 0;
                }
                res[i] = res[i]%2;
        }
        for (i = 4; i >= 0; i--)
        {
                pro[i] = res[i];
                printf("%d",pro[i]);
        }
        printf(":");
        for (i = 4; i >= 0; i--)
        {
                printf("%d", anumcp[i]);
        }
}
void arshift()
{
        //for arithmetic shift right
        int temp = pro[4], temp2 = pro[0], i;
        for (i = 1; i < 5 ; i++)
        {
                //shift the MSB of product
                pro[i-1] = pro[i];
        }
        pro[4] = temp;
        for (i = 1; i < 5 ; i++)
        {
                //shift the LSB of product
                anumcp[i-1] = anumcp[i];
        }
        anumcp[4] = temp2;
        printf("\nAR-SHIFT: ");
        //display together
        for (i = 4; i >= 0; i--)
        {
                printf("%d",pro[i]);
        }
        printf(":");
        for(i = 4; i >= 0; i--)
        {
                printf("%d", anumcp[i]);
        }
}
void main()
{
        int i, q = 0;
        printf("\t\tBOOTH'S MULTIPLICATION ALGORITHM");
        printf("\nEnter two numbers to multiply: ");
        printf("\nBoth must be less than 16");
        //simulating for two numbers each below 16
```

```
do
{
        printf("\nEnter A: ");
        scanf("%d",&a);
        printf("Enter B: ");
        scanf("%d", &b);
}
while(a >=16 || b >=16);
printf("\nExpected product = %d", a * b);
binary();
printf("\n\nBinary Equivalents are: ");
printf("\nA = ");
for (i = 4; i >= 0; i--)
{
        printf("%d", anum[i]);
}
printf("\nB = ");
for (i = 4; i >= 0; i--)
{
        printf("%d", bnum[i]);
}
printf("\nB'+ 1 = ");
for (i = 4; i >= 0; i--)
{
        printf("%d", bcomp[i]);
}
printf("\n\n");
for (i = 0;i < 5; i++)
{
        if (anum[i] == q)
        {
                //just shift for 00 or 11
                printf("\n-->");
                arshift();
                q = anum[i];
        }
        else if(anum[i] == 1 && q == 0)
        {
                //subtract and shift for 10
                printf("\n-->");
                printf("\nSUB B: ");
                add(bcomp);
                //add two's complement to implement subtraction
                arshift();
                q = anum[i];
        }
        else
        {
                //add ans shift for 01
                printf("\n-->");
```

```
                        printf("\nADD B: ");
                        add(bnum);
                        arshift();
                        q = anum[i];
                }
        }
        printf("\nProduct is = ");
        for (i = 4; i >= 0; i--)
        {
                printf("%d", pro[i]);
        }
        for (i = 4; i >= 0; i--)
        {
                printf("%d", anumcp[i]);
        }
}
```

<u>OUTPUT:</u>



```
                BOOTH'S MULTIPLICATION ALGORITHM
Enter two numbers to multiply:
Both must be less than 16
Enter A: 10
Enter B: 5
Expected product = 50

Binary Equivalents are:
A = 01010
B = 00101
B'+ 1 = 11011


-->
AR-SHIFT: 00000:00101
-->
SUB B: 11011:00101
AR-SHIFT: 11101:10010
-->
ADD B: 00010:10010
AR-SHIFT: 00001:01001
-->
SUB B: 11100:01001
AR-SHIFT: 11110:00100
-->
ADD B: 00011:00100
AR-SHIFT: 00001:10010
Product is = 0000110010_
```

PRACTICAL-7

AIM : Design ALU using Logisim.

LOGISIM:

Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes.

Features:

- Logisim is open-source (GPL).
- It runs on *any* machine supporting Java 5 or later; special versions are released for MacOS X and Windows. The cross-platform nature is important for students who have a variety of home/dorm computer systems.
- The drawing interface is based on an intuitive toolbar. Color-coded wires aid in simulating and debugging a circuit.
- The wiring tool draws horizontal and vertical wires, automatically connecting to components and to other wires. It's very easy to draw circuits!
- Completed circuits can be saved into a file, exported to a GIF file, or printed on a printer.
- Circuit layouts can be used as "subcircuits" of other circuits, allowing for hierarchical circuit design.
- Included circuit components include inputs and outputs, gates, multiplexers, arithmetic circuits, flip-flops, and RAM memory.
- The included "combinational analysis" module allows for conversion between circuits, truth tables, and Boolean expressions.

How to Use:

- Logisim is simple, compact and portable. The installation process is easy and startup takes only a few seconds. Logisim opens as a simple workspace with panels on the top and the left. The workspace is dotted. You can drag logic gates, components and wires and connect them inside the workspace to create your circuit.

- Connecting the components together is simple. Simply dragging the mouse between the intended components joins them with a digital wire. The wires can run horizontally and vertically.

- All the elements of the circuit are colour-coded. For example, input ports are represented in green and output ports in blue. Also, active wires during simulation turn bright green, while wires with no electricity remain deeper green by default. Any flawed wiring is indicated in red. All input and output ports have a designated value of either 0 or 1 assigned to them, indicating their active status.

- The upper edge of the software contains two menu bars. The topmost bar contains File, Edit, Project, Simulate, Windows and Help menus. You can perform different actions from these menus, such as open or continue a project, edit a circuit, access tutorials or run a simulation. Project menu contains many sub-menus like Add Circuit, Load or Unload Library, View Simulation Tree, Analyse Circuit and so on. Under Analyse Circuit, you can check values of the input, output and truth table that the logic circuit follows.

- The other toolbar consists of some useful tools and shortcuts. Finger icon lets you toggle between a cursor and a finger, and change values in the circuit. So, if you click an input value of 0 with the finger, it will change to 1, meaning the input is receiving electric flow.

- Cursor is the default mouse pointer. You can use it for interactive actions on the workspace, like selecting and moving components, connecting or deleting wires, resetting component properties and more. The icon for textbox lets you add labels to your circuit for convenience.

- Following the icons are symbols of five of the most used components in an electronic circuit—input, output, as well as NOT, AND OR logic gates. Use the cursor to drag these elements on the toolbar and drop them on the workspace for use in the circuit as desired.

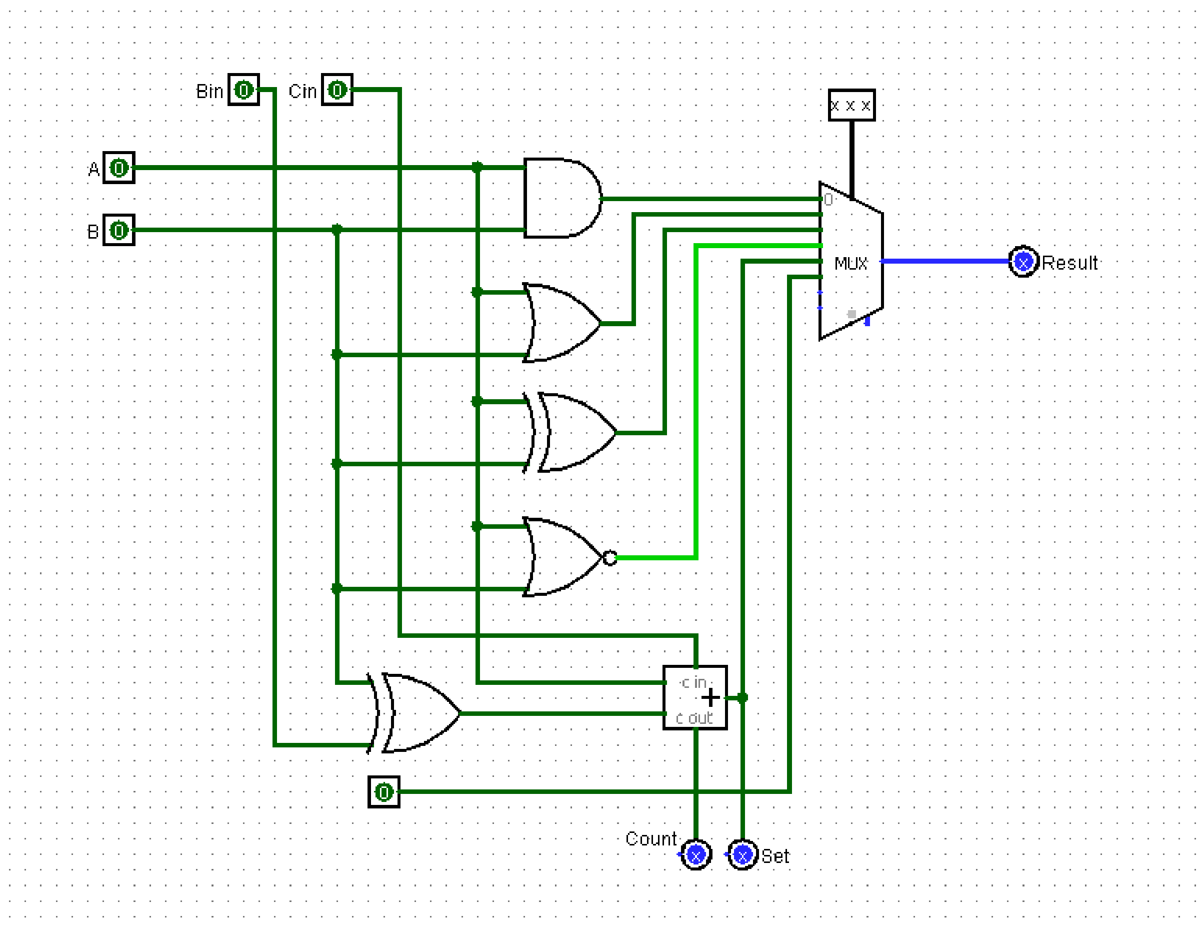
- The left panel of the software consists of libraries and properties windows. The first folder represents the project. It is followed by folders called Base, Gates, Plexers, Arithmetic, Memory and Input/Output. All folders contain components from the library that can be used for building different kinds of circuits and sub-circuits. These components are categorised in folders for easy retrieval.

- For instance, under Gates folder, along with the common three logic gates mentioned earlier, you get NAND, NOR, XOR, XNOR, Buffer, Odd Party, Even Party and so on.

- Below the folders comes Properties window. Here you can edit properties of all components. The window displays circuit name, text label assigned, direction and font of the label, number of input ports of logic gates and so on. To modify any component, just click the component in the circuit with the cursor and do the modifications in this windows. Suppose you dragged in an AND gate that has two input ports by default. If you need four input ports, click the AND gate symbol and in the properties window, click to change the number in Inputs entry field. The logic gate in the circuit will update immediately.

Designing an ALU:

In the computer system, ALU is a main component of the central processing unit, which stands for arithmetic logic unit and performs arithmetic and logic operations. It is also known as an integer unit (IU) that is an integrated circuit within a CPU or GPU, which is the last component to perform calculations in the processor. It has the ability to perform all processes related to arithmetic and logic operations such as addition, subtraction, and shifting operations, including Boolean comparisons (XOR, OR, AND, and NOT operations). Also, binary numbers can accomplish mathematical and bitwise operations. The arithmetic logic unit is split into AU (arithmetic unit) and LU (logic unit). The operands and code used by the ALU tell it which operations have to perform according to input data. When the ALU completes the processing of input, the information is sent to the computer's memory.



- This ALU is capable of the operations AND, OR, XOR, NOR, ADD/SUB.
- [A] & [B] are separately 8-bit inputs.
- The [Sub] input designates whether adding (0) or subtracting (1).
- The [Operation Setter] designates what operation is being performed.
- [R] is the result of the designated operation, with the [Zero Flag] alerting us to a ‘0’ output, such as when a number is subtracted from itself.

PRACTICAL-8

AIM : Implement 16-bit single-cycle MIPS processor in Verilog HDL.

16 bit single-cycle MIPS processor :

INTODUCTION:

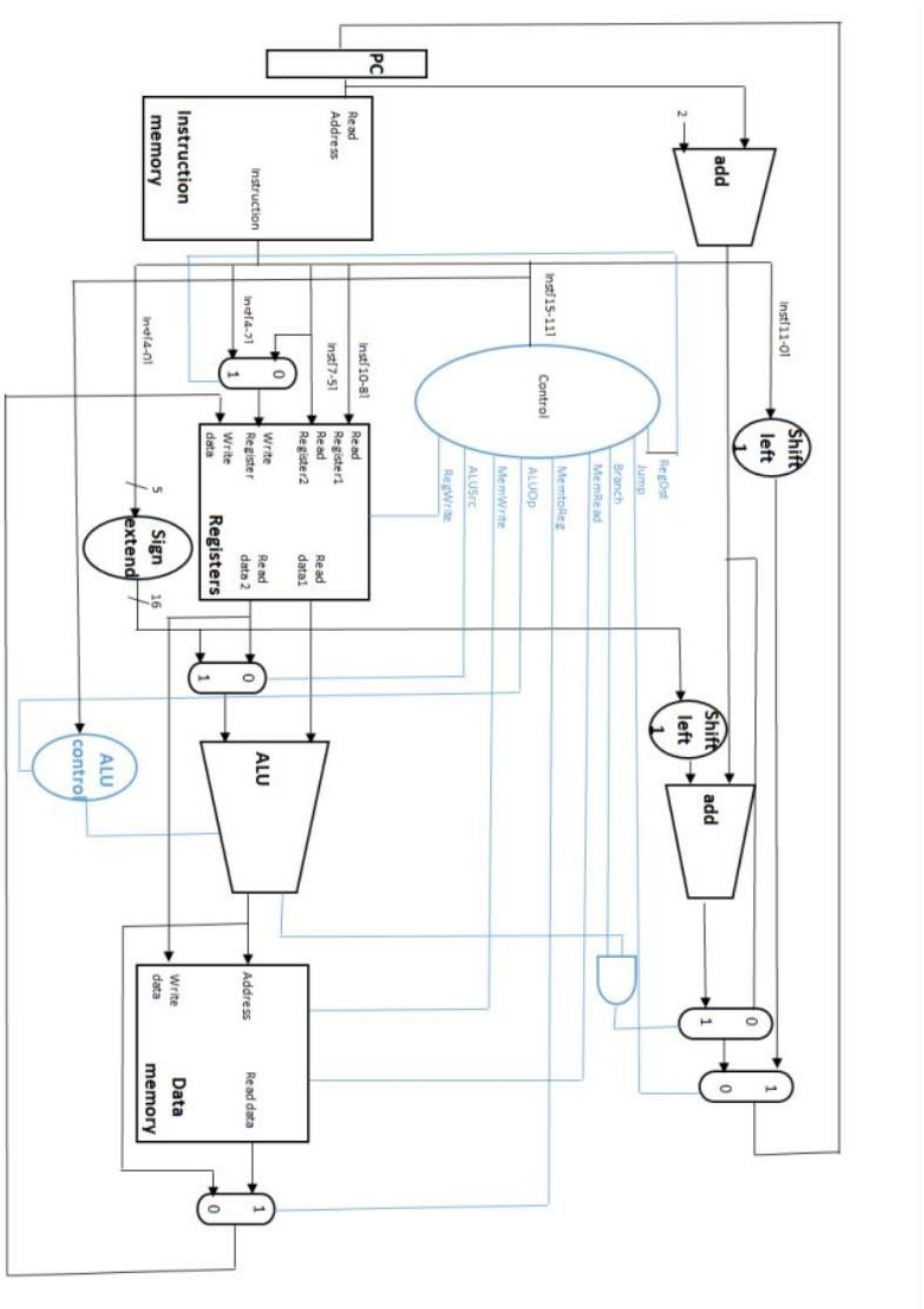
We will be designing the datapath and control for a 16 bit, single-cycle (nonpipelined) processor. This processor will be able to execute a set of instructions given (similar to MIPS). In this report you will find a hardware diagram, a listing of the 10 functions and their op codes, a description of the control unit, and explanations of the process for each of the 10 functions executed.

Our hardware performs as indicated in class and throughout the book, specifically our Arithmetic logic unit (ALU) has 6 functions AND, OR, add, subtract, set on less than and NORwhich are defined by the 4bit output of the ALU control 0000, 0001, 0010, 0110, 0111, and 1100 respectively. For our design we only used the add, subtract, And, and Or functions of theALU The control will output a 2 bit ALUOp code signifying the action needed by the ALU; 00 is addition for load and store, 01 is subtraction for Beq, and when the Op code is 01 the operation is determined by the 5 bit Op code for type 1 instructions. Our sign extension can convert both 5- and 8-bit offsets into a full 16 bits.

HARDWARE DIAGRAM :

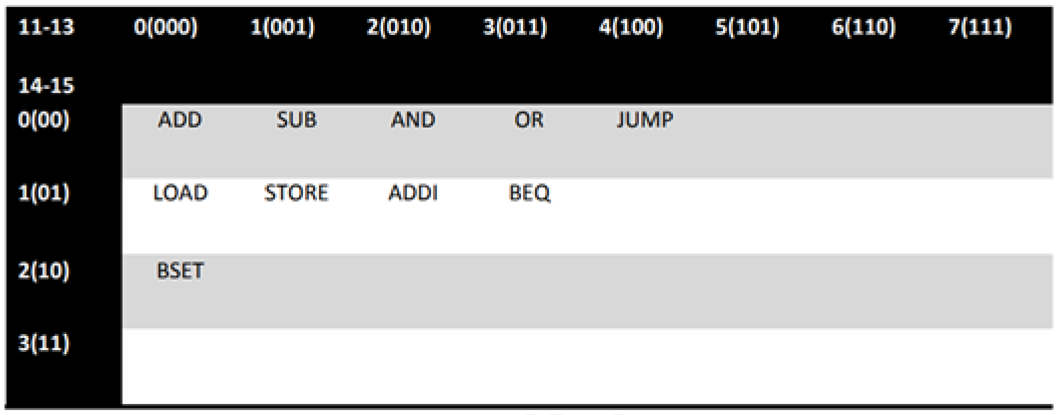
Figure below shows the hardware diagram for our designed processor which includes the ALU, PC, Control, and Memory. Each function of the processor is detailed further later in this report. There are 9 outputs of the control unit which include Register Destination (RegDst), Jump, Branch, Memory Read (MemRead), Memory to Register (MemtoReg), ALU Operation (ALUOp), Memory Write (MemWrite), ALU Source (ALUSrc), and Register Write (RegWrite). Since our Processor is 16 bits we assign instruction bits to certain sections of the datapath. For our processor the register reads aren’t specific to the MIPs R type configurationmeaning any register can be used for each read and write location.

PEN : 200840131012



5-BIT OP CODE:

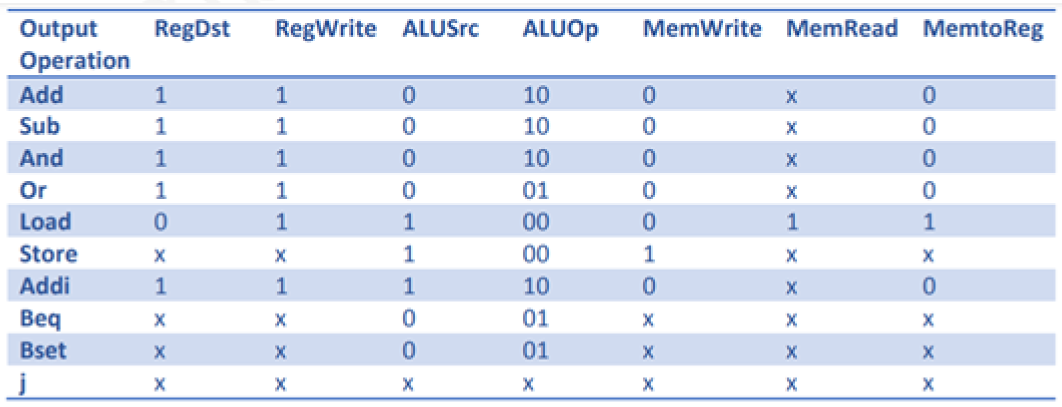
Figure below Shows the given function assigned to with 5 bit codes. The code is indicated by the first two bits of the row and the last 3 bits of the column. For example the Addi function is located in row 1 (01) and column 3 (011), which gives an op code of 01011. We decided to split each function type indicated by bits 14 and 15. Type 1 is referenced by 00, type 2 by 01, and type 3 by 10. There is plenty of design space to add functions as we have room for 8 functions under 4 function types culminating to a total of 32 functions (25).



11-13 14-15	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	ADD	SUB	AND	OR	JUMP			
1(01)	LOAD	STORE	ADDI	BEQ				
2(10)	BSET							
3(11)								

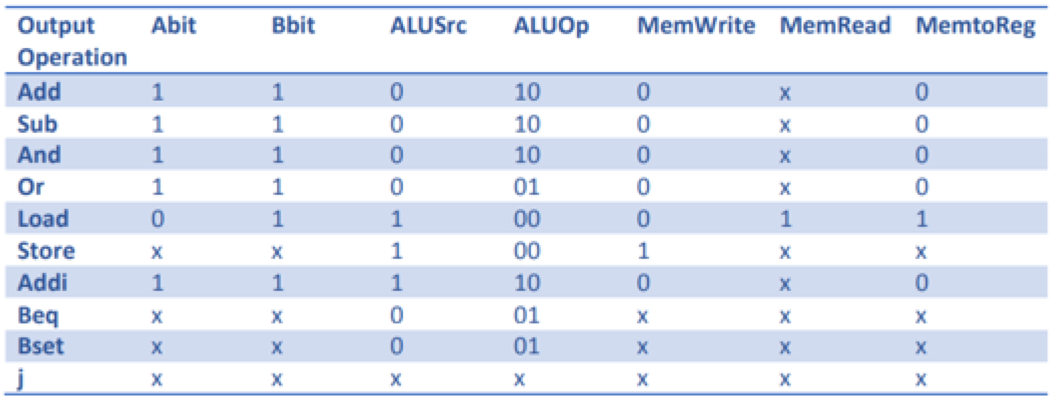
CONTROL SIGNAL :

In Figure below we show the signal output of the control unit. The Jump and Branch/B- set outputs are not shown as outputs due to the fact they are exclusive to their own function. The control unit will send a 1 or a 0 as the output as indicated. The control releases no output for the fields marked with an x.



Output Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemtoReg
Add	1	1	0	10	0	x	0
Sub	1	1	0	10	0	x	0
And	1	1	0	10	0	x	0
Or	1	1	0	01	0	x	0
Load	0	1	1	00	0	1	1
Store	x	x	1	00	1	x	x
Addi	1	1	1	10	0	x	0
Beq	x	x	0	01	x	x	x
Bset	x	x	0	01	x	x	x
j	x	x	x	x	x	x	x

CONTROL SIGNAL TABLE



Output Operation	Abit	Bbit	ALUSrc	ALUOp	MemWrite	MemRead	MemtoReg
Add	1	1	0	10	0	x	0
Sub	1	1	0	10	0	x	0
And	1	1	0	10	0	x	0
Or	1	1	0	01	0	x	0
Load	0	1	1	00	0	1	1
Store	x	x	1	00	1	x	x
Addi	1	1	1	10	0	x	0
Beq	x	x	0	01	x	x	x
Bset	x	x	0	01	x	x	x
j	x	x	x	x	x	x	x

TYPE-1 FUNCTION DESCRIPTIONS :

Each function of the Datapath has a unique process in which the controller and datapath mandate which instruments are utilized for operation. Type 1 functions use 3 registers and arein the format {field (bits)}: Opcode (5), R1 (3), R2 (3), R3 (3), Unused (2). Jump is the exception with only and Opcode (5) and an offset (11). Each of the 5-type 1 processor functionsare detailed below.

Op code 00000 (Add):

The "Add" function utilizes the Op code, and the control unit sets the ALUOp to 10. The value of $r3 [2-4] are read and the value of $r2 [5-7] are used due to the 0 indication fromthe ALUSrc multiplexer. The two register values are fed into the ALU which performs an "add"function dictated by the ALU Control output (0010). Once the addition is performed the resultis written into the 3 bits [8-10] of $r1 because the MemtoReg multiplexer is set to 0.

Op code 00001 (Subtract):

The "Subtract" function utilizes the Op code, and the control unit sets the ALUOp to 10. The value of $r3 [2-4] is read and the value of $r2 [5-7] are used due to the 0 indication from the ALUSrc multiplexer. The two register values are fed into the ALU which performs a"subtract" function dictated by the ALU Control output (0110 due to the negation of the $r3).Once the subtraction is performed the result is written into the 3 bits [8-10] of $r1 because theMemtoReg multiplexer is set to 0.

Op code 00010 (And):

The “And” function utilizes the Op code, and the control unit sets the ALUOp to 10. The 3 bits [2-4] of $r3 are read and the 3 bits [5-7] of $r2 are used due to the 0 indication fromthe ALUSrc multiplexer. The two register values are fed into the ALU which performs an “And” function dictated by the ALUControl output (0000). Once each bit is compared to see if they are both 1 the results of the “And” are written into the 3 bits [8-10] of $r1 because the MemtoReg multiplexer is set to 0.

Op code 00011 (Or):

The “Or” function utilizes the Op code, and the control unit sets the ALUOp to 10. The3 bits [2-4] of $r3 are read and the 3 bits [5-7] of $r2 are used due to the 0 indication from theALUSrc multiplexer. The two register values are fed into the ALU which performs an “Or” function dictated by the ALUControl output (0001). Once each bit is compared to see if eitheris a 1 the results of the “Or” are written into the 3 bits [8-10] of $r1 because the MemtoReg multiplexer is set to 0.

Op code 00100 (Jump):

The jump function is unique in the fact that it only requires the offset [0-10] to be shiftedleft by one thus multiplying by 2. This address is then issued to the PC through the multiplexerto execute the jump.

TYPE-2 FUNCTIONS DESCRIPTIONS:

Each function of the datapath has a unique process in which the controller and datapathmandate which instruments are utilized for operation. Type 2 functions use two registers and are in the format {field (bits)}: Opcode (5), Register1 (3), Register2 (3), Offset/Value (5). Eachof the 4- type 2 processor functions are detailed below.

Op code 01000 (Load):

The “Load” function utilizes the offset, and the control unit sets the ALUOp to 00 which is addition for Load and Store. The offset[0-4] is sign extended and loaded into the ALU,due to the 1 signal of the AluSrc, along with the base address of $r2 [5-7] which is loaded from memory, due to the signal of MemRead. These values are added to compute an address. That address is then read from memory and written back into $r1[8-10].

Op code 01001 (Store):

The "Store" function utilizes the offset, and the control unit sets the ALUOp to 00 which is addition for Load and Store. The offset[0-4] is sign extended and loaded into the ALU,due to the 1 signal of the AluSrc, along with the value of $r2 [5-7]. These values are added to compute an address. The resulting address is then read from memory and written back into $r1[8-10].

Op code 01010 (Addi):

The "Addi" function utilizes the offset, and the control unit sets the ALUOp to 01 similar to the Add function. The value [0-4] is sign extended and loaded into the ALU, due tothe 1 signal of the AluSrc, along with the value of $r2 [5-7]. Once the addition is performed the result is written into the 3 bits [8-10] of $r1 because the MemtoReg multiplexer is set to 0.

Op code 01011 (Beq):

The "Beq" function utilizes the offset, and the control unit sets the ALUOp to 01 which is subtraction for Beq. The offset is sign extended and applied to the adder ALU.The value of $r1 [8-10] is read and the value of $r2 [5-7] is used due to the 0 indication from the ALUSrc multiplexer. For Beq $r2 would be set to $0. The two register values are fed into the ALU which performs a "subtract" function dictated by the ALUControl output (0110 due to the negation of the $r3). If the resulting value is zero the connected "and" gate will result in one (since the branch value is set to one). This will add the value of the offset (already in the adder) to the value of the pc which is also in the adder. In turn the multiplexer will be set to one which will send the new address to the pc to execute. If the two register values aren't equalthen the function acts as a no op.

<u>TYPE-3 FUNCTION DESCRIPTIONS :</u>

Each function of the Datapath has a unique process in which the controller and datapath mandate which instruments are utilized for operation. Type 3 functions use two registers and are in the format {field (bits)}: Opcode (5), Register1 (3), Register2 (3), Offset/Value (5). Bset is the only type 3 function, and it is a Branch instruction so the Branch control output would be set to 1.

Op code 10000 (Bset):

The “Bset” function utilizes the offset, and the control unit sets the ALUOp to 01 which is subtraction for Beq. The offset is sign extended and applied to the ALU. The value of $r1 [8-10] is read and the value of $r2 [5-7] is used due to the 0 indication from the ALUSrc multiplexer. Our design requires the value of $r2 to be 000 in order for the Bset function to beoperational. The two register values are fed into the ALU which performs a “subtract” functiondictated by the ALUControl output (0110 due to the negation of the $r3). If the resulting valueis zero the connected “and” gate will result in one (since the branch value is set to one). This will add the value of the offset (already in the adder) to the value of the pc which is also in theadder. In turn the multiplexer will be set to one which will send the new address to the pc to execute. If the two register values aren’t equal then the function acts as a no op.