# Practical No:1

**AIM: Implement Booth’s Algorithm.**

**Program:**

Booth’s algorithm is a multiplication algorithm that multiplies two signed binary numbers in 2’s compliment notation.  
Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth’s algorithm is of interest in the study of computer architecture. Here’s the implementation of the algorithm.

# // CPP code to implement booth's algorithm

# #include <bits/stdc++.h>

# using namespace std;

# // function to perform adding in the accumulator

# void add(int ac[], int x[], intqrn)

# {

# int i, c = 0;

# for (i = 0; i <qrn; i++)

# {

# // updating accumulator with A = A + BR

# ac[i] = ac[i] + x[i] + c;

# if (ac[i] > 1)

# {

# ac[i] = ac[i] % 2;

# c = 1;

# }

# Else

# c = 0;

# }

# }

# // function to find the number's complement

# void complement(int a[], int n)

# {

# int i;

# int x[8] = { 0};

# x[0] = 1;

# for (i = 0; i < n; i++) {

# a[i] = (a[i] + 1) % 2;

# }

# add(a, x, n);

# }

# // function ro perform right shift

# voidrightShift(int ac[], intqr[], int&qn, intqrn)

# {

# int temp, i;

# temp = ac[0];

# qn = qr[0];

# cout<< "\t\trightShift\t";

# for (i = 0; i <qrn - 1; i++) {

# ac[i] = ac[i + 1];

# qr[i] = qr[i + 1];

# }

# qr[qrn - 1] = temp;

# }

# // function to display oprations

# void display(int ac[], intqr[], intqrn)

# {

# int i;

# // accumulator content

# for (i = qrn - 1; i >= 0; i--)

# cout<< ac[i];

# cout<< "\t";

# // multiplier content

# for (i = qrn - 1; i >= 0; i--)

# cout<<qr[i];

# }

# // Function to implement booth's algo

# voidboothAlgorithm(intbr[], intqr[], intmt[], intqrn, intsc)

# {

# intqn = 0, ac[10] = { 0 };

# int temp = 0;

# cout<< "qn\tq[n+1]\t\tBR\t\tAC\tQR\t\tsc\n";

# cout<< "\t\t\tinitial\t\t";

# display(ac, qr, qrn);

# cout<< "\t\t" <<sc<< "\n";

# while (sc != 0)

# {

# cout<<qr[0] << "\t" <<qn;

# // SECOND CONDITION

# if ((qn + qr[0]) == 1)

# {

# if (temp == 0)

# {

# // subtract BR from accumulator

# add(ac, mt, qrn);

# cout<< "\t\tA = A - BR\t";

# for (int i = qrn - 1; i >= 0; i--)

# cout<< ac[i];

# temp = 1;

# }

# // THIRD CONDITION

# else if (temp == 1)

# {

# // add BR to accumulator

# add(ac, br, qrn);

# cout<< "\t\tA = A + BR\t";

# 

# for (int i = qrn - 1; i >= 0; i--)

# cout<< ac[i];

# temp = 0;

# }

# cout<< "\n\t";

# rightShift(ac, qr, qn, qrn);

# }

# // FIRST CONDITION

# else if (qn - qr[0] == 0)

# rightShift(ac, qr, qn, qrn);

# display(ac, qr, qrn);

# cout<< "\t";

# // decrement counter

# sc--;

# cout<< "\t" <<sc<< "\n";

# }

# }

# // driver code

# int main(intargc, char\*\* arg)

# {

# intmt[10], sc;

# intbrn, qrn;

# // Number of multiplicand bit

# brn = 4;

# // multiplicand

# intbr[] = { 0, 1, 1, 0 };

# 

# // copy multiplier to temp array mt[]

# for (int i = brn - 1; i >= 0; i--)

# mt[i] = br[i];

# reverse(br, br + brn);

# complement(mt, brn);

# // No. of multiplier bit

# qrn = 4;

# // sequence counter

# sc = qrn;

# // multiplier

# intqr[] = { 1, 0, 1, 0 };

# reverse(qr, qr + qrn);

# boothAlgorithm(br, qr, mt, qrn, sc);

# cout<<endl

# << "Result = ";

# for (int i = qrn - 1; i >= 0; i--)

# cout<<qr[i];

# }

**Output:**

Input : 0110, 0010

Output :qn q[n+1] AC QR sc(step count)

initial 0000 0010 4

0 0 rightShift 0000 0001 3

1 0 A = A - BR 1010

rightShift 1101 0000 2

0 1 A = A + BR 0011

rightShift 0001 1000 1

0 0 rightShift 0000 1100 0

Result=1100

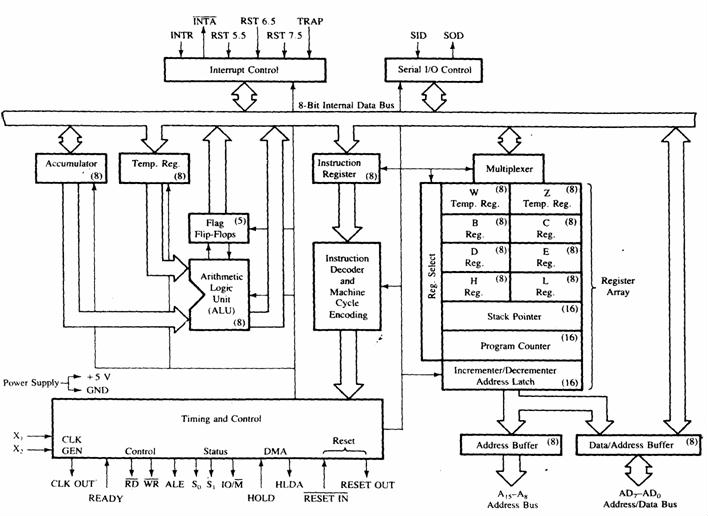
# Practical No:2

**AIM: Write the working of simulator GNUsim8085and basic architecture of 8085 along with small instruction.**

|  |
| --- |
|  |

Computer Organization and architecture lab consist of performing various experiments in GNU Sim (A simulator for 8085 microprocessor).

Before doing the coding on the simulator it’s necessary to study the complete architecture of 8085 microprocessor along with its instruction set.

ARCHITECTURE OF MICROPROCESSOR8085

### List of registers used in 8085 to perform various operations:

**Accumulator**:-It is a 8-bit register which is used to perform arithmetical and logical operation. It stores the output of any operation. It also works as registers for i/o accesses.

It can be one of the operand in the instruction.

**Temporary Register:**-It is a 8-bit register which is used to hold the data on which the accumulator is computing operation. It is also called as operand register because it provides operands to ALU.

**Registers**:-These are general purposes registers. Microprocessor consists 6 general purpose registers of 8-bit each named as B,C,D,E,H and L. Generally theses registers are not used for storing the data permanently. It carries the 8-bits data. These are used only during the execution of the instructions.

These registers can also be used to carry the 16 bits data by making the pair of 2 registers. The valid register pairs available are BC,DE HL. We cannot use other pairs except BC,DEand HL. These registers are programmed by user.

**Flag Registers:-**It consists of 5 flip flop which changes its status according to the result stored in an accumulator. It is also known as status registers. It is connected to the ALU.

There are five flip-flops in the flag register are as follows:

**The bit position of flip top in flag register is :**

All of the three flip flop set and reset according to the stored result in the accumulator.

1. **Sign**- If D7 of the result is 1 then sign flag is set otherwise reset. As we know that a number on the D7 always desides the sign of thenumber.

if D7 is 1: the number is negative. if D7 is 0: the number is positive.

1. **Zeros(Z)**-If the result stored in an accumulator is zero then this flip flop is set as 1 otherwiseit is reset and also if the result of any arithmetic or logical operation is zero its set as 1( The result of this operation can be in anyregisters).
2. **Auxiliary carry(AC)**-If any carry goes from D3 to D4 in the output then it is set otherwise itis reset.
3. **Parity(P)**-If the no of 1's is even in the output stored in the accumulator then it is setotherwise it is reset for theodd.
4. **Carry(C)**-If the result stored in an accumulator generates a carry in its final output then it isset

otherwise it is reset.

**Instruction registers(IR)**:-It is a 8-bit register. When an instruction is fetched from memory then it is stored in this register.

### Description of other components of 8085 to:

**Instruction Decoder: -** Instruction decoder identifies the instructions. It takes the information from instruction register and decodes the instruction to be performed.

**Program Counter:**-It is a 16 bit register used as memory pointer. It stores the memoryaddress of the next instruction to be executed. So we can say that this register is used to sequencing the program. Generally the memory have 16 bit addresses so that it has 16 bitmemory.

The program counter is set to 0000H.

**Stack Pointer:**-It is also a 16 bit register used as memory pointer. It points to the memory location called stack. Generally stack is a reserved portion of memory where information can be stores or taken back together.

**Timing and Control Unit:**-It provides timing and control signal to the microprocessor to perform the various operation. It has three control signals. It controls all external and internal circuits. It operates with reference to clock signal. It synchronizes all the data transfers.

There are three control signal:

1. ALE-Arithmetic Latch Enable, It provides control signal to synchronize the components of microprocessor.
2. RD- This is active low used for reading operation. 3.WR-This is active low used for writingoperation.

There are three status signal used in microprocessor S0, S1 and IO/M. It changes its status according the provided input to these pins.

**Serial Input Output Control**-There are two pins in this unit. This unit is used for serial data communication.

**Interrupt Unit**-There are 6 interrupt pins in this unit. Generally an external hardware is connected to these pins. These pins provide interrupt signal sent by external hardware to microprocessor and microprocessor sends acknowledgement for receiving the interrupt signal. Generally INTA is used for acknowledgement.

### INTRODUCTION TO GNU Simulator8085

8085 simulator is software on which instructions are executed by writing the programs in assembly language.

GNUSim8085 is a 8085 microprocessor simulator with following features.

* + A simple editor component with syntaxhighlighting.
  + A keypad to input assembly language instructions with appropriatearguments.
  + Easy view of registercontents.
  + Easy view of flagcontents.
  + Hexadecimal <--> Decimalconverter.
  + View of stack, memory and I/Ocontents.
  + Support for breakpoints for programmingdebugging.
  + Stepwise programexecution.
  + One click conversion of assembly program to opcodelisting.
  + Printing support (known not to work well onWindows).
  + UI translated in variouslanguages.

### Writing a program in assembly language:-

**Format of the instruction is as follows:-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Label** | **Operation** | **Operands** | **Comments** |
| Its optional | Necessary | Necessary | Its optional |

A basic assembly program consists of 4 parts.

1. Labels
2. Operations :- these operations can be specifiedas

**Machine operations (mnemonics):**- used to define operations in the form of opcode as mention in the instruction set of microprocessor 8085.

**Pseudo operations (like preprocessor in C):-** these are assembly directives.

1. Operands
2. Comments

In addition, you have **constants** in an assembly program. Unless otherwise specified, a constant which is always numeric is in decimal form. If appended with a character h it is assumed to be in hexadecimal form. If a hex constant starts with an alpha-char don't forget to include the

number0 in the beginning, since that will help the assembler to differentiate between a label and a constant.

**Labels**:-

When given to any particular instruction/data in a program, takes the address of that instruction or data as its value. But it has different meaning when given to EQU directive. Then it takes the operand of EQU as its value. Labels must always be placed in the first column and must be followed by an instruction (no empty line). Labels must be followed by a :(colon), to differentiate it from other tokens.

### Operations:-

As mentioned above the operations can be specified in two ways that are **mnemonics** and

### pseudo operation.

Pseudo operations can be defined by using following directives:-

There are only 3 directives currently available in our assembly language.

1. DB - define byte ( 8 bits)
2. DS - define size (no. ofbytes)
3. EQU - like minimalistic #define inC

**DB**is used to define space for an array of values specified by comma separated list. And the label (if given to the beginning of DB) is assigned the address of the first data item.

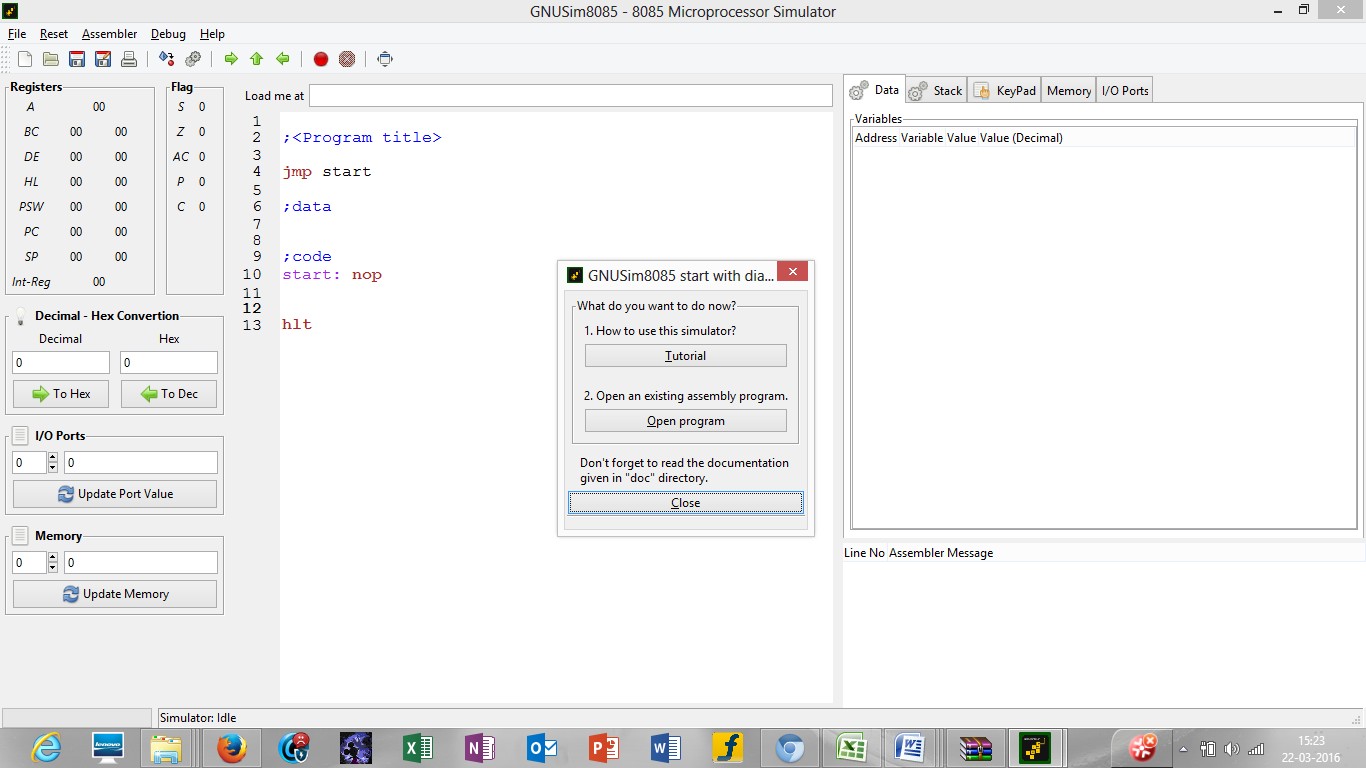
**DS**is used to define the specified number of bytes to be assigned and initialize them to zero. To access each byte you can use the + or -operator along with label.

**EQU**behaves similar to #define in C. But it is simple. It can be used to give names only to numeric constants. Nesting of EQU is not allowed. You can use EQU only in operands for pseudo ops and mnemonics.

### Operands:-

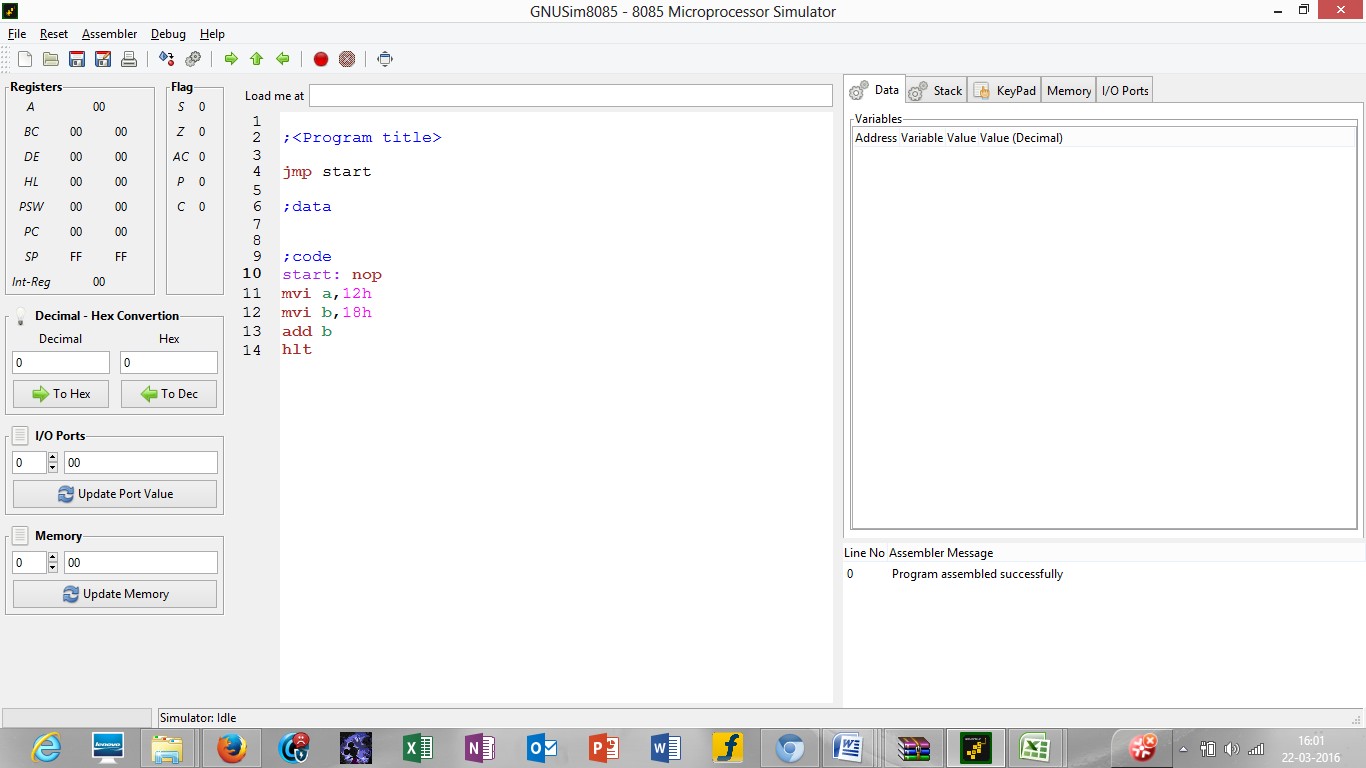
Operands are specified according to the user. The register set specified in the architecture of 8085 (A, B, C, D, H and L) are used to access and store data. These registers are specified as operand. In case of accessing data or storing data in the memory ‘m’ is specified as an operand and the address of this memory location is taken from the HL pair (data in HL pair).

### Working WITH GNU Simulator 8085



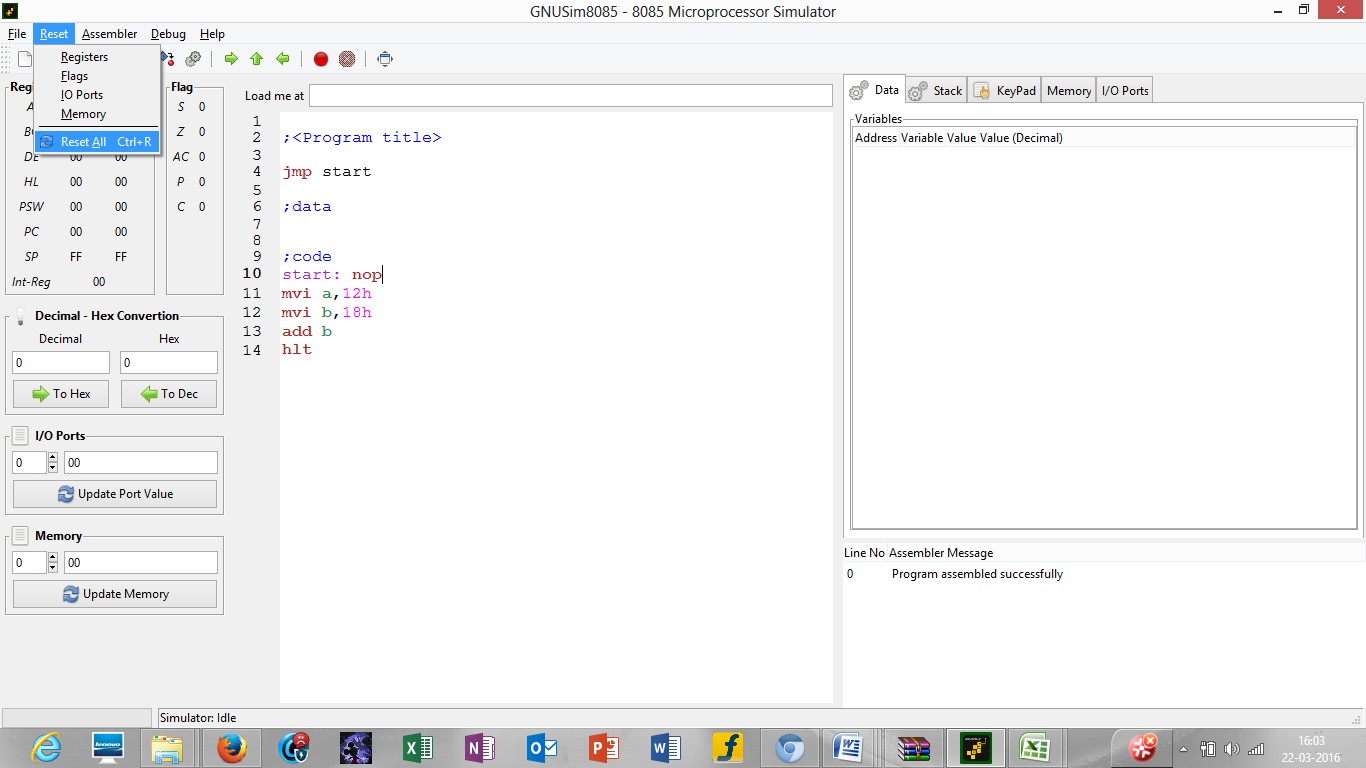
**Picture 1**

**Step1:**open GNU Sim 8085 above window will open. Now click on close button highlighted in the above screen shot.



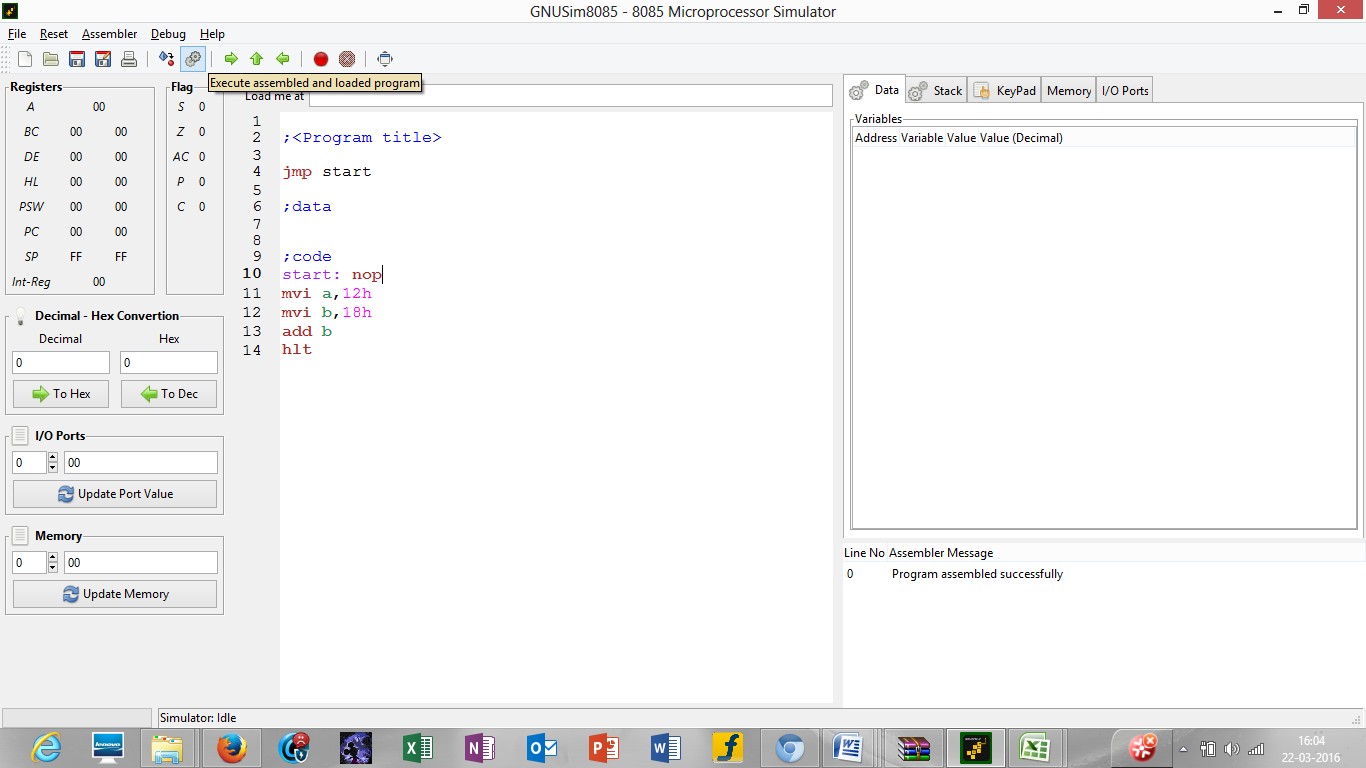
### Picture 2

**Step2:** start writing the code after start: nop in load me at 10 that is at load me at 11.



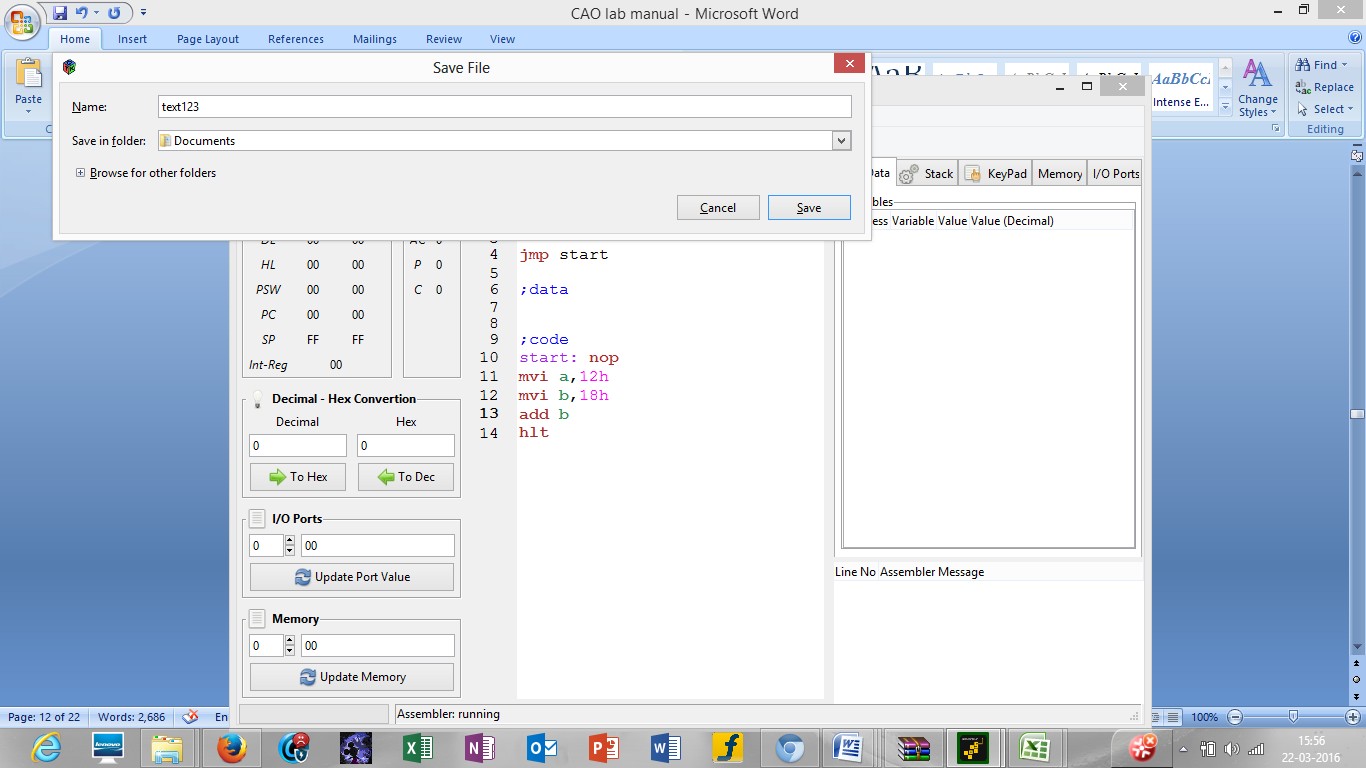
### Picture 3

**Step 3**: click on reset and reset all the registers by clicking on reset all.



**Picture 4**

**Step 4**: click on the highlighted button to execute thecode



### Picture 5

**Step 5**: after you execute the code mention the name your program by writing the name in the name section as mentioned in the screen shot in picture 5 and the drive where you want to save it. After that click on save.

Picture 6

**Step 6**: after this you will see the result of the instructions in the respective registers as seen in the above picture 6.

# Practical No:3

**AIM : Write the assembly language code in GNUsim8085 to store number in reverse order in memory location.**

lxih,var

movc,m;counter

inx h

lxi d,varr4

back: mova,m

stax d

inx h

dcx d

dcr c

jnz back

hlt

var: db 04h

var1: db 03h

var2: db 07h

var3: db 08h

var4: db 05h

varr1: db 00h

varr2: db 00h

varr3: db 00h

varr4: db 00h

**(OR)**

**Algorithm:**

1. Load the accumulator with the first data.
2. Use RLC instruction to shift contain of accumulator by 1 bit without carry. Use this 4 times to reverse the contain of accumulator.
3. Now load the result value in memory location.

To reverse 8 bits number

LDA 2050

RLC

RLC

RLC

RLC

STA 3050

HLT

Output**:**

INPUT:

2050:05

OUTPUT:

3050:50

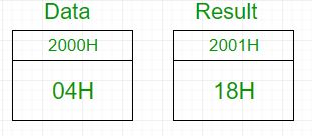
|  |
| --- |
|  |

# Practical No:5

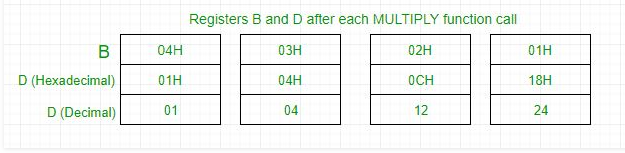
**AIM : Write the assembly language code in GNUsim8085 to find a factorial of a number.**

**Example**

|  |
| --- |
| Input : 04H  Output : 18H  as 04\*03\*02\*01 = 24 in decimal => 18H |

****

In 8085 microprocessor, no direct instruction exists to multiply two numbers, so multiplication is done by repeated addition as 4×3 is equivalent to 4+4+4 (i.e., 3 times).  
Load 04H in D register -> Add 04H 3 times -> D register now contains 0CH -> Add 0CH 2 times -> D register now contains 18H -> Add 18H 1 time -> D register now contains 18H -> Output is 18H



**Algorithm –**

1. Load the data into register B
2. To start multiplication set D to 01H
3. Jump to step 7
4. Decrements B to multiply previous number
5. Jump to step 3 till value of B>0
6. Take memory pointer to next location and store result
7. Load E with contents of B and clear accumulator
8. Repeatedly add contents of D to accumulator E times
9. Store accumulator content to D

10. Go to step 4

| **ESS** | **LABEL** | **MNEMONIC** | **COMMENT** |
| --- | --- | --- | --- |
| 2000H | Data |  | Data Byte |
| 2001H | Result |  | Result of factorial |
| 2002H |  | LXI H, 2000H | Load data from memory |
| 2005H |  | MOV B, M | Load data to B register |
| 2006H |  | MVI D, 01H | Set D register with 1 |
| 2008H | FACTORIAL | CALL MULTIPLY | Subroutine call for multiplication |
| 200BH |  | DCR B | Decrement B |
| 200CH |  | JNZ FACTORIAL | Call factorial till B becomes 0 |
| 200FH |  | INX H | Increment memory |
| 2010H |  | MOV M, D | Store result in memory |
| 2011H |  | HLT | Halt |
| 2100H | MULTIPLY | MOV E, B | Transfer contents of B to C |
| 2101H |  | MVI A, 00H | Clear accumulator to store result |
| 2103H | MULTIPLYLOOP | ADD D | Add contents of D to A |
| 2104H |  | DCR E | Decrement E |
| 2105H |  | JNZ MULTIPLYLOOP | Repeated addition |
| 2108H |  | MOV D, A | Transfer contents of A to D |
| 2109H |  | RET | Return from subroutine |
|  |  |  |  |

**The assembly language code in GNUsim8085 is below**

lxi sp,27ffh

lda var2

cpi 02h

jc last

mvi d,00h

move,a

dcr a

movc,a

call facto

xchg

shldvar

jmp end

last: lxi h,0001h

end: shldvar

hlt

facto: lxi h,0000h

movb,c

back: dad d

dcr b

jnz back

xchg

dcr c

cnz facto

ret

var: db 00h

var2: db03h ;

input the number 3 here, donot give number more than 5

Input : 04H

Output : 18H

as 04\*03\*02\*01 = 24 in decimal => 18H

# Practical No:8

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

**Program:**

## In this project, a 16-bit single-cycle MIPS processor is implemented in Verilog HDL. [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture) is an [RISC processor](https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html), which is widely used by many universities in academic courses related to computer organization and architecture.

#### The Instruction Format and [Instruction Set Architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture) for the 16-bit single-cycle MIPS are as follows:

|  |
| --- |
| Verilog code for 16 bit MIPS CPU |
| ***Instruction set for the MIPS processor*** |

|  |
| --- |
| Verilog code for MIPS processor |
| ***Instruction Set Architecture for the MIPS processor*** |

#### Below is the description for instructions being implemented in Verilog:

1. Add : R[rd] = R[rs] + R[rt]
2. Subtract : R[rd] = R[rs] - R[rt]
3. And: R[rd] = R[rs] & R[rt]
4. Or : R[rd] = R[rs] | R[rt]
5. SLT: R[rd] = 1 if R[rs] <  R[rt] else 0
6. Jr: PC=R[rs]
7. Lw: R[rt] = M[R[rs]+SignExtImm]
8. Sw : M[R[rs]+SignExtImm] = R[rt]
9. Beq : if(R[rs]==R[rt]) PC=PC+1+BranchAddr
10. Addi: R[rt] = R[rs] + SignExtImm
11. J : PC=JumpAddr
12. Jal : R[7]=PC+2;PC=JumpAddr
13. SLTI: R[rt] = 1 if R[rs] < imm else 0

SignExtImm = { 9{immediate[6]}, imm

JumpAddr =    { (PC+1)[15:13], address}

BranchAddr = { 7{immediate[6]}, immediate, 1’b0 }

#### Based on the provided instruction set, the data-path and control unit are designed and implemented.

**Control unit design:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Control signals** | | | | | | | | | |
| **Instruction** | **Reg**  **Dst** | **ALUSrc** | **Memto**  **Reg** | **Reg**  **Write** | **MemRead** | **Mem**  **Write** | **Branch** | **ALUOp** | **Jump** |
| **R-type** | **1** | **0** | **0** | **1** | **0** | **0** | **0** | **00** | **0** |
| **LW** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **11** | **0** |
| **SW** | **0** | **1** | **0** | **0** | **0** | **1** | **0** | **11** | **0** |
| **addi** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **11** | **0** |
| **beq** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **01** | **0** |
| **j** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **00** | **1** |
| **jal** | **2** | **0** | **2** | **1** | **0** | **0** | **0** | **00** | **1** |
| **slti** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **10** | **0** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ALU Control** | | | | |
| **ALU op** | **Function** | **ALUcnt** | **ALU Operation** | **Instruction** |
| **11** | **xxxx** | **000** | **ADD** | **Addi,lw,sw** |
| **01** | **xxxx** | **001** | **SUB** | **BEQ** |
| **00** | **00** | **000** | **ADD** | **R-type: ADD** |
| **00** | **01** | **001** | **SUB** | **R-type: sub** |
| **00** | **02** | **010** | **AND** | **R-type: AND** |
| **00** | **03** | **011** | **OR** | **R-type: OR** |
| **00** | **04** | **100** | **slt** | **R-type: slt** |
| **10** | **xxxxxx** | **100** | **slt** | **i-type: slti** |

|  |
| --- |
| k |
| *Data-path and control unit of the 16-bit MIPS processor* |

#### After completing the design for the MIPS processor, it is easy to write Verilog code for the MIPS processor. The Verilog code for the whole design of the MIPS processor as follows:

[**Verilog code for ALU unit**](https://www.fpga4student.com/2017/01/basic-digital-blocks-in-verilog.html)  
[**Verilog code for register file**](https://www.fpga4student.com/2017/01/basic-digital-blocks-in-verilog.html)  
[**Verilog code for instruction memory**](https://www.fpga4student.com/2017/01/basic-digital-blocks-in-verilog.html)

#### Verilog code for data memory:

//fpga4student.com: [FPGA](https://www.fpga4student.com/2017/08/what-is-fpga-programming.html) projects, Verilog projects, VHDL projects

// Verilog project: Verilog code for 16-bit MIPS Processor

// Submodule: Data memory in Verilog

**module**data\_memory

(

**input**clk,

// address input, shared by read and write port

**input** [**15**:**0**] mem\_access\_addr,

// write port

**input** [**15**:**0**] mem\_write\_data,

**input**mem\_write\_en,

**input**mem\_read,

// read port

**output** [**15**:**0**] mem\_read\_data

);

**integer**i;

**reg** [**15**:**0**] ram [**255**:**0**];

**wire** [**7**:**0**] ram\_addr=mem\_access\_addr[**8**:**1**];

**initialbegin**

**for**(i=**0**;i<**256**;i=i+**1**)

ram[i] <=**16'd0**;

**end**

**always** @(**posedge**clk) **begin**

**if** (mem\_write\_en)

ram[ram\_addr] <=mem\_write\_data;

**end**

**assign**mem\_read\_data= (mem\_read==**1'b1**) ?ram[ram\_addr]:**16'd0**;

**endmodule**

#### Verilog code for ALU Control unit:

//fpga4student.com: [FPGA projects](https://www.fpga4student.com/p/fpga-projects.html), [Verilog projects](https://www.fpga4student.com/p/verilog-project.html), [VHDL projects](https://www.fpga4student.com/p/vhdl-project.html)

// Verilog project: Verilog code for 16-bit MIPS Processor

// Submodule: ALU Control Unit in Verilog

**module**ALUControl(ALU\_Control, ALUOp, Function);

**outputreg**[**2**:**0**] ALU\_Control;

**input** [**1**:**0**] ALUOp;

**input** [**3**:**0**] Function;

**wire** [**5**:**0**] ALUControlIn;

**assign**ALUControlIn= {ALUOp,Function};

**always** @(ALUControlIn)

**casex** (ALUControlIn)

**6'b11xxxx:**ALU\_Control=**3'b000**;

**6'b10xxxx:**ALU\_Control=**3'b100**;

**6'b01xxxx:**ALU\_Control=**3'b001**;

**6'b000000**:ALU\_Control=**3'b000**;

**6'b000001**:ALU\_Control=**3'b001**;

**6'b000010**:ALU\_Control=**3'b010**;

**6'b000011**:ALU\_Control=**3'b011**;

**6'b000100**:ALU\_Control=**3'b100**;

**default**:ALU\_Control=**3'b000**;

**endcase**

**endmodule**

// Verilog code for JR control unit

**module**JR\_Control(**input**[**1**:**0**] alu\_op,

**input** [**3**:**0**] funct,

**output**JRControl

);

**assign**JRControl= ({alu\_op,funct}==**6'b001000**) ?**1'b1**:**1'b0**;

**endmodule**

#### Verilog code for control unit:

//fpga4student.com: FPGA projects, Verilog projects, VHDL projects

// Verilog project: Verilog code for 16-bit MIPS Processor

// Submodule: Control Unit in Verilog

**module** control( **input**[**2**:**0**] opcode,

**input** reset,

**outputreg**[**1**:**0**] reg\_dst,mem\_to\_reg,alu\_op,

**outputreg**jump,branch,mem\_read,mem\_write,alu\_src,reg\_write,sign\_or\_zero

);

**always** @(\*)

**begin**

**if**(reset ==**1'b1**) **begin**

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b00**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b0**;

sign\_or\_zero=**1'b1**;

**end**

**elsebegin**

**case**(opcode)

**3'b000**:**begin**// add

reg\_dst=**2'b01**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b00**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b1**;

**end**

**3'b001**:**begin**// sli

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b10**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b1**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b0**;

**end**

**3'b010**:**begin**// j

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b00**;

jump=**1'b1**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b0**;

sign\_or\_zero=**1'b1**;

**end**

**3'b011**:**begin**// jal

reg\_dst=**2'b10**;

mem\_to\_reg=**2'b10**;

alu\_op=**2'b00**;

jump=**1'b1**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b1**;

**end**

**3'b100**:**begin**// lw

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b01**;

alu\_op=**2'b11**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b1**;

mem\_write=**1'b0**;

alu\_src=**1'b1**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b1**;

**end**

**3'b101**:**begin**// sw

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b11**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b1**;

alu\_src=**1'b1**;

reg\_write=**1'b0**;

sign\_or\_zero=**1'b1**;

**end**

**3'b110**:**begin**// beq

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b01**;

jump=**1'b0**;

branch=**1'b1**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b0**;

sign\_or\_zero=**1'b1**;

**end**

**3'b111**:**begin**// addi

reg\_dst=**2'b00**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b11**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b1**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b1**;

**end**

**default**:**begin**

reg\_dst=**2'b01**;

mem\_to\_reg=**2'b00**;

alu\_op=**2'b00**;

jump=**1'b0**;

branch=**1'b0**;

mem\_read=**1'b0**;

mem\_write=**1'b0**;

alu\_src=**1'b0**;

reg\_write=**1'b1**;

sign\_or\_zero=**1'b1**;

**end**

**endcase**

**end**

**end**

**endmodule**

### Verilog code for the single-cycle [MIPS processor](https://www.fpga4student.com/2017/09/vhdl-code-for-mips-processor.html):

//fpga4student.com: [FPGA](https://www.fpga4student.com/2017/08/what-is-fpga-programming.html) projects, Verilog projects, VHDL projects

// Verilog project: Verilog code for 16-bit MIPS Processor

// Verilog code for 16 bit single cycle MIPS CPU

**module** mips\_16( **input**clk,reset,

**output**[**15**:**0**] pc\_out, alu\_result

//,reg3,reg4

);

**reg**[**15**:**0**] pc\_current;

**wiresigned**[**15**:**0**] pc\_next,pc2;

**wire** [**15**:**0**] instr;

**wire**[**1**:**0**] reg\_dst,mem\_to\_reg,alu\_op;

**wire**jump,branch,mem\_read,mem\_write,alu\_src,reg\_write ;

**wire** [**2**:**0**] reg\_write\_dest;

**wire** [**15**:**0**] reg\_write\_data;

**wire** [**2**:**0**] reg\_read\_addr\_1;

**wire** [**15**:**0**] reg\_read\_data\_1;

**wire** [**2**:**0**] reg\_read\_addr\_2;

**wire** [**15**:**0**] reg\_read\_data\_2;

**wire** [**15**:**0**] sign\_ext\_im,read\_data2,zero\_ext\_im,imm\_ext;

**wire**JRControl;

**wire** [**2**:**0**] ALU\_Control;

**wire** [**15**:**0**] ALU\_out;

**wire**zero\_flag;

**wiresigned**[**15**:**0**] im\_shift\_1, PC\_j, PC\_beq, PC\_4beq,PC\_4beqj,PC\_jr;

**wire**beq\_control;

**wire** [**14**:**0**] jump\_shift\_1;

**wire** [**15**:**0**]mem\_read\_data;

**wire** [**15**:**0**] no\_sign\_ext;

**wire**sign\_or\_zero;

// PC

**always** @(**posedge**clk**orposedge** reset)

**begin**

**if**(reset)

pc\_current<=**16'd0**;

**else**

pc\_current<=pc\_next;

**end**

// PC + 2

**assign** pc2 =pc\_current+**16'd2**;

// instruction memory

instr\_meminstrucion\_memory(.pc(pc\_current),.instruction(instr));

// jump shift left 1

**assign** jump\_shift\_1 = {instr[**13**:**0**],**1'b0**};

// control unit

control control\_unit(.reset(reset),.opcode(instr[**15**:**13**]),.reg\_dst(reg\_dst)

,.mem\_to\_reg(mem\_to\_reg),.alu\_op(alu\_op),.jump(jump),.branch(branch),.mem\_read(mem\_read),

.mem\_write(mem\_write),.alu\_src(alu\_src),.reg\_write(reg\_write),.sign\_or\_zero(sign\_or\_zero));

// multiplexer regdest

**assign**reg\_write\_dest= (reg\_dst==**2'b10**) ?**3'b111**: ((reg\_dst==**2'b01**) ?instr[**6**:**4**] :instr[**9**:**7**]);

// register file

**assign** reg\_read\_addr\_1 =instr[**12**:**10**];

**assign** reg\_read\_addr\_2 =instr[**9**:**7**];

register\_filereg\_file(.clk(clk),.rst(reset),.reg\_write\_en(reg\_write),

.reg\_write\_dest(reg\_write\_dest),

.reg\_write\_data(reg\_write\_data),

.reg\_read\_addr\_1(reg\_read\_addr\_1),

.reg\_read\_data\_1(reg\_read\_data\_1),

.reg\_read\_addr\_2(reg\_read\_addr\_2),

.reg\_read\_data\_2(reg\_read\_data\_2));

//.reg3(reg3),

//.reg4(reg4));

// sign extend

**assign**sign\_ext\_im= {{**9**{instr[**6**]}},instr[**6**:**0**]};

**assign**zero\_ext\_im= {{**9**{**1'b0**}},instr[**6**:**0**]};

**assign**imm\_ext= (sign\_or\_zero==**1'b1**) ?sign\_ext\_im:zero\_ext\_im;

// JR control

JR\_Control JRControl\_unit(.alu\_op(alu\_op),.funct(instr[**3**:**0**]),.JRControl(JRControl));

// ALU control unit

ALUControl ALU\_Control\_unit(.ALUOp(alu\_op),.Function(instr[**3**:**0**]),.ALU\_Control(ALU\_Control));

// multiplexer alu\_src

**assign** read\_data2 = (alu\_src==**1'b1**) ?imm\_ext: reg\_read\_data\_2;

// ALU

alu alu\_unit(.a(reg\_read\_data\_1),.b(read\_data2),.alu\_control(ALU\_Control),.result(ALU\_out),.zero(zero\_flag));

// immediate shift 1

**assign** im\_shift\_1 = {imm\_ext[**14**:**0**],**1'b0**};

//

**assign**no\_sign\_ext=~(im\_shift\_1) +**1'b1**;

// PC beq add

**assign**PC\_beq= (im\_shift\_1[**15**] ==**1'b1**) ? (pc2 -no\_sign\_ext): (pc2 +im\_shift\_1);

// beq control

**assign**beq\_control= branch &zero\_flag;

// PC\_beq

**assign** PC\_4beq = (beq\_control==**1'b1**) ?PC\_beq: pc2;

// PC\_j

**assign**PC\_j= {pc2[**15**],jump\_shift\_1};

// PC\_4beqj

**assign** PC\_4beqj = (jump ==**1'b1**) ?PC\_j: PC\_4beq;

// PC\_jr

**assign**PC\_jr= reg\_read\_data\_1;

// PC\_next

**assign**pc\_next= (JRControl==**1'b1**) ?PC\_jr: PC\_4beqj;

// data memory

data\_memorydatamem(.clk(clk),.mem\_access\_addr(ALU\_out),

.mem\_write\_data(reg\_read\_data\_2),.mem\_write\_en(mem\_write),.mem\_read(mem\_read),

.mem\_read\_data(mem\_read\_data));

// write back

**assign**reg\_write\_data= (mem\_to\_reg==**2'b10**) ?**pc2:**((mem\_to\_reg==**2'b01**)?**mem\_read\_data:**ALU\_out);

// output

**assign**pc\_out=pc\_current;

**assign**alu\_result=ALU\_out;

**endmodule**

### Verilog testbench code for the single-cycle MIPS processor:

**`timescale1**ns /**1**ps

//fpga4student.com: [FPGA](https://www.fpga4student.com/2017/08/what-is-fpga-programming.html) projects, Verilog projects, VHDL projects

// Verilog project: Verilog code for 16-bit MIPS Processor

// Testbench Verilog code for 16 bit single cycle MIPS CPU

**module** tb\_mips16;

// Inputs

**reg**clk;

**reg** reset;

// Outputs

**wire** [**15**:**0**] pc\_out;

**wire** [**15**:**0**] alu\_result;//,reg3,reg4;

// Instantiate the Unit Under Test (UUT)

mips\_16uut (

.clk(clk),

.reset(reset),

.pc\_out(pc\_out),

.alu\_result(alu\_result)

//.reg3(reg3),

// .reg4(reg4)

);

**initialbegin**

clk=**0**;

**forever** #**10**clk=~clk;

**end**

**initialbegin**

// Initialize Inputs

//$monitor ("register 3=%d, register 4=%d", reg3,reg4);

reset=**1**;

// Wait 100 ns for global reset to finish

#**100**;

reset=**0**;

// Add stimulus here

**end**

**endmodule**

It is quite simple to verify the Verilog code for the single-cycle MIPS CPU by doing several simulations on ModelSim or Xilinx ISIM in order to see how the MIPS processor works. To fully verify the MIPS processor, it is needed to modify the instruction memory to simulate all the instructions in the instruction set architecture, and then check simulation waveform and memory to see if the processor works correctly as designed.