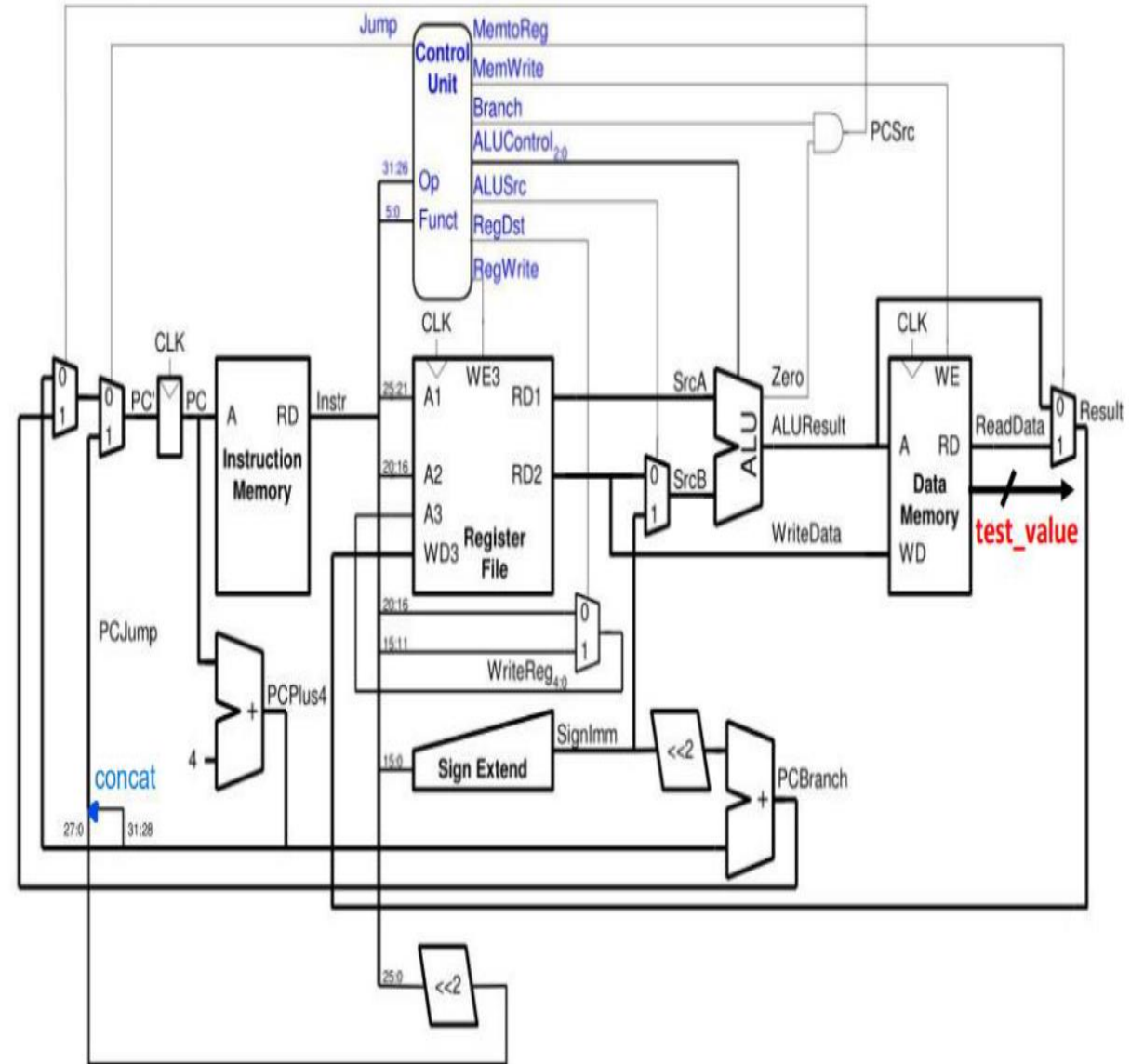


# 32BITS SINGLE-CYCLE MIPS PROCESSOR



# AGENDA

Introduction

MIPS Architecture modules

Codes

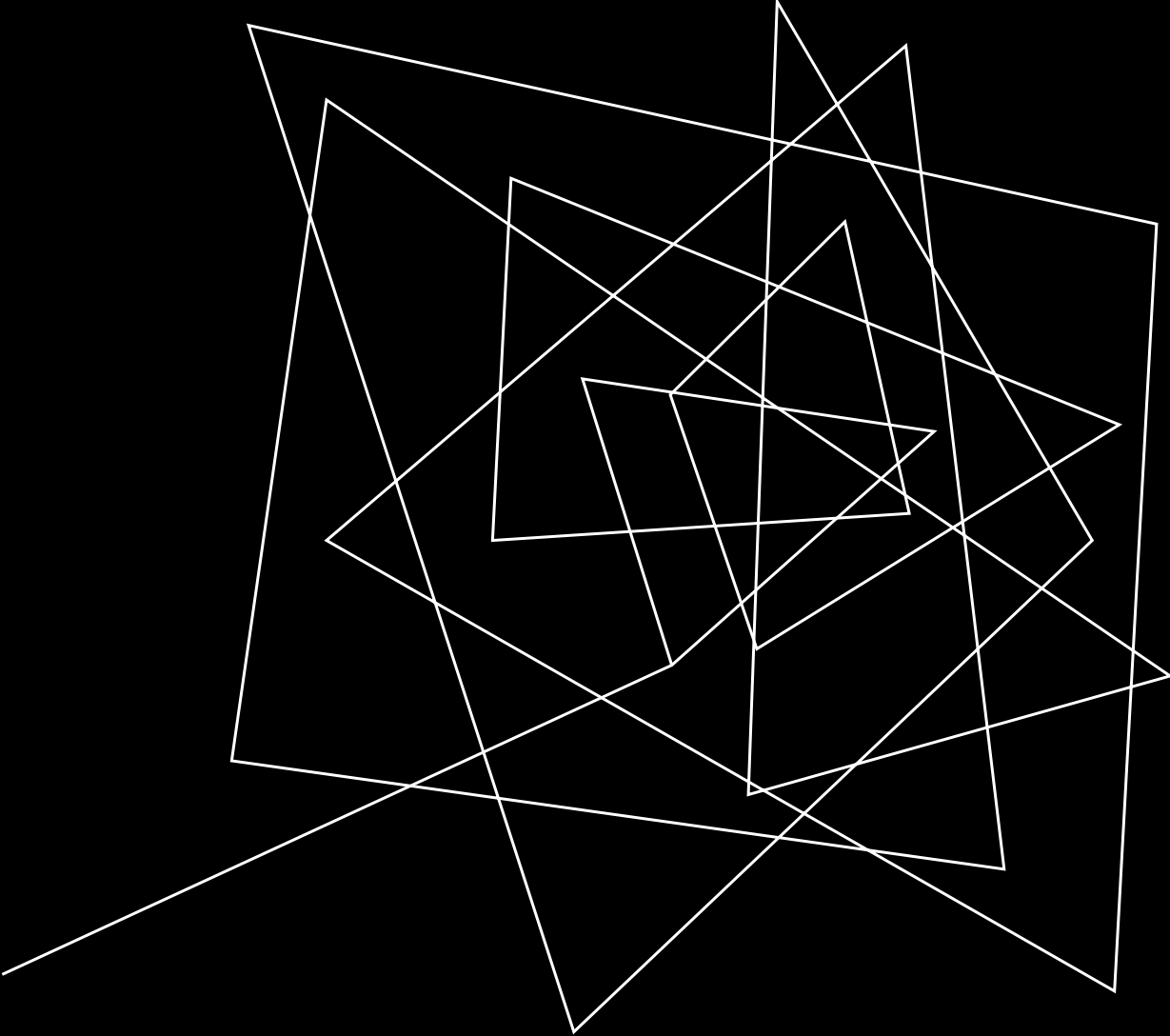
Simulation and test

# INTRODUCTION

-MIPS architecture is a type of the RISC (Reduced Instruction Set Computer) design philosophy, which aims to minimize complexity and optimizing performance. It uses a fixed instruction with a length of 32 bits, making instruction decoding and execution easy. The design allows only a small set of instructions, focusing on simple operations while ignoring complex instructions that are rarely used.

-MIPS architecture is byte-addressed, not word-addressed.

-In the Single Cycle MIPS architecture, each instruction is executed in a single clock cycle. This means that the entire instruction fetch, decode, execute, and write-back stages are completed within a single cycle.



# MIPS ARCHITECTURE COMPONENTS

# MAIN MODULES

1. Add4 (to add 4 to program counter).
2. ALu (to do arithmetic and logic operations).
3. ALUDecoder (to control the ALu module).
4. ControlUnit (generate the control signals which control the processor's modules).
5. DataMem (store the data).
6. instMem (store the instructions of the programs).
7. MainDecoder (generate the control signals and a signal which control ALUDecoder).
8. mu\_x\_2\_1\_5bits.
9. mu\_x\_2\_1\_32bits.
10. PC (program counter).
11. PCBranch (adder).
12. RegFile (store the registers values).
13. sign\_extend (extend the number from 16bits to 32bits while maintaining the sign).
14. SI2 (shift left 2 == multiplication by 4).



# CODES OF MODULES

## RegFile

```
module RegFile
#(parameter width = 32,depth = 32)
(
input clk,
input WE3,
input [4:0] A1,A2,A3,
input [31:0] WD3,
output [31:0] RD1,RD2
);

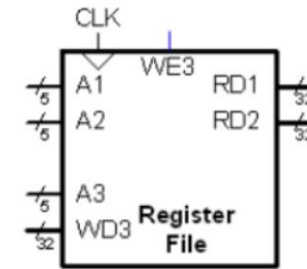
reg [width-1:0] RegFile [0:depth-1];

integer i;
initial
begin
for (i = 0; i < 32; i = i+1)
    RegFile [i] = 0;
end

always @(posedge clk)
begin
if (WE3)
begin
    RegFile[A3] <= WD3; // Synchronous write
    RegFile[0]=32'b0; //always keep the value of $0 equal to zero
end
end

assign RD1 = RegFile[A1]; // Asynchronous read for RD1
assign RD2 = RegFile[A2]; // Asynchronous read for RD2

endmodule
```



# InstMem

```
module InstMem
#(parameter width = 32,depth = 256)
(
input [width-1:0] pc_A,
output reg [width-1:0] instr
);

reg [width-1:0] InstMem [0:depth-1];

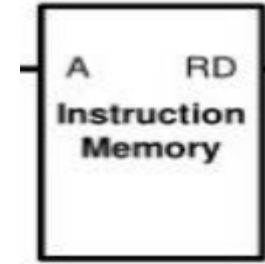
/*****FIND GCD FOR 120 and 180*****/
initial
begin
InstMem[0] = 32'h00008020;
InstMem[1] = 32'h20100078; // change the least two significant decimals with the number
InstMem[2] = 32'h00008820;
InstMem[3] = 32'h201100B4; // change the least two significant decimals with the number.
InstMem[4] = 32'h00009020;
InstMem[5] = 32'h12110006;
InstMem[6] = 32'h0211482A;
InstMem[7] = 32'h11200002;
InstMem[8] = 32'h02308822;
InstMem[9] = 32'h08000005;
InstMem[10] = 32'h02118022;
InstMem[11] = 32'h08000005;
InstMem[12] = 32'h00109020;
InstMem[13] = 32'hAC120000;
end

/*****FIND FACTORIAL OF NUMBER 7*****/
/*initial
begin
InstMem [0] = 32'h00008020; //add $s0, $0, $0
InstMem [1] = 32'h20100007; //addi $s0, $0, 7
InstMem [2] = 32'h00008820; //add $s1, $0, $0
InstMem [3] = 32'h20110001; //addi $s1, $0, 1
InstMem [4] = 32'h12000003; //beq $s1, $0, 3
InstMem [5] = 32'h0230881C; //mul $s1, $s1, $s0
InstMem [6] = 32'h2210FFFF; //addi $s0, $s0, -1
InstMem [7] = 32'h08000004; //j 4
InstMem [8] = 32'hAC110000; //sw $s1, 0($0)
end*/

/*****/

always @(*)
begin
instr = InstMem[pc_A[9:2]]; // Read the instruction from the ROM memory
end

endmodule
```





# DataMem

```
module Data_Mem
#(parameter n = 32)
(
    input clk,
    input WE,
    input [31:0] A,
    input [31:0] WD,
    output [31:0] RD,
    output [15:0] test_value
);

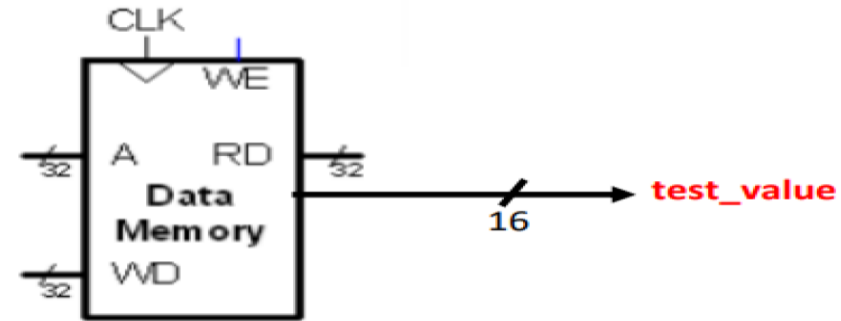
    reg [31:0] Data_Mem [0:255];

    integer i;
    initial
    begin
        for (i = 0; i < 256; i = i+1)
            Data_Mem [i] = 0;
        end

    always @(posedge clk)
    begin
        if (WE)
            Data_Mem[A] <= WD; // synchronous write
        end

    assign RD = Data_Mem[A]; // Asynchronous read
    assign test_value = Data_Mem[32'h0000_0000][15:0]; // Test value read from address 0x0000_0000

endmodule
```

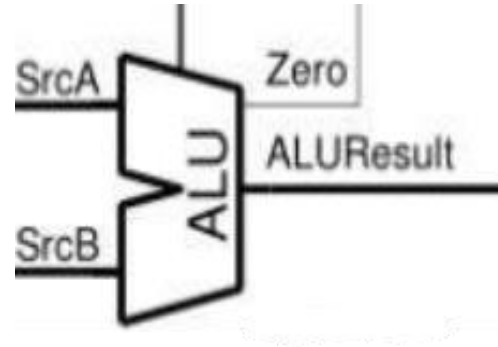


## ALU

```

module ALU
#(parameter n = 32)
(
    input [n-1:0] A,B,
    input [2:0] ALUControl,
    output reg [n-1:0] ALUResult,
    output reg zeroflag
);
always @(*)
begin
    case (ALUControl)
        3'b000:ALUResult <= A & B;
        3'b001:ALUResult <= A | B;
        3'b010:ALUResult <= A + B;
        3'b100:ALUResult <= A - B;
        3'b101:ALUResult <= A * B;
        3'b110:ALUResult <= (A < B) ? 1'b1:1'b0;
        default:ALUResult <= 0;
    endcase
    zeroflag <= (ALUResult == 0); // Set zero flag if result is zero
end
endmodule

```



## ControlUnit

```

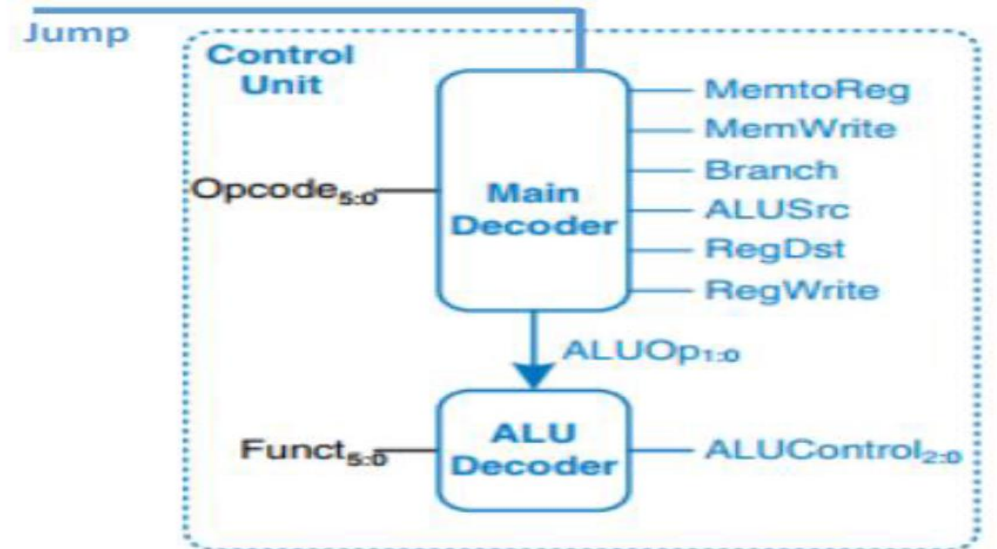
module ControlUnit
(
    input [5:0] OpCode,
    input [5:0] Funct,
    input zero,
    output Jump,
    output MemtoReg,MemWrite,ALUSrc,RegDest,RegWrite,PCSrc,
    output [2:0] ALUControl
);
    wire Branch;
    wire [1:0] ALUOp;

    MainDecoder Block1
    (
        .OpCode(OpCode),
        .Jump(Jump),
        .ALUOp(ALUOp),
        .MemtoReg(MemtoReg),
        .MemWrite(MemWrite),
        .Branch(Branch),
        .ALUSrc(ALUSrc),
        .RegDest(RegDest),
        .RegWrite(RegWrite)
    );

    ALUDecoder Block2
    (
        .ALUOp(ALUOp),
        .Funct(Funct),
        .ALUControl(ALUControl)
    );

    and (PCSrc,zero,Branch);
endmodule

```

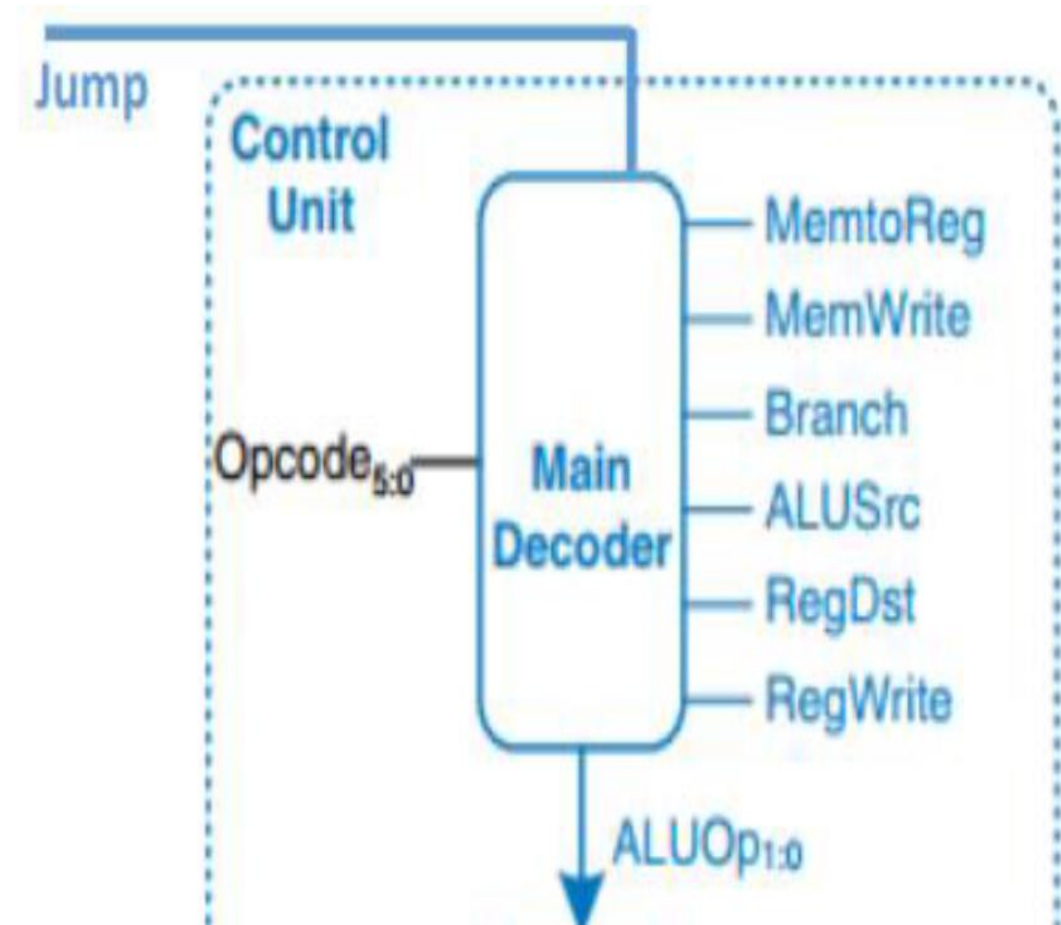


# MainDecoder

```

module MainDecoder
(
    input [5:0] Opcode,
    output reg [1:0] ALUop,
    output reg Jump,
    output reg Memwrite, Regwrite, MemtoReg, RegDest, ALUSrc, Branch
);
always @(*)
begin
    casez(Opcode)
        6'b100011: //lw
        begin
            Jump = 1'b0;
            ALUop = 2'b00;
            Memwrite = 1'b0;
            Regwrite = 1'b1;
            RegDest = 1'b0;
            ALUSrc = 1'b1;
            MemtoReg = 1'b1;
            Branch = 1'b0;
        end
        6'b101011: //sw
        begin
            Jump = 1'b0;
            ALUop = 2'b00;
            Memwrite = 1'b1;
            Regwrite = 1'b0;
            RegDest = 1'b0;
            ALUSrc = 1'b1;
            MemtoReg = 1'b1;
            Branch = 1'b0;
        end
        6'b000000: //R type
        begin
            Jump = 1'b0;
            ALUop = 2'b10;
            Memwrite = 1'b0;
            Regwrite = 1'b1;
            RegDest = 1'b1;
            ALUSrc = 1'b0;
            MemtoReg = 1'b0;
            Branch = 1'b0;
        end
        6'b001000: //addi
        begin
            Jump = 1'b0;
            ALUop = 2'b00;
            Memwrite = 1'b0;
            Regwrite = 1'b1;
            RegDest = 1'b0;
            ALUSrc = 1'b1;
            MemtoReg = 1'b0;
            Branch = 1'b0;
        end
        6'b000100: //beq
        begin
            Jump = 1'b0;
            ALUop = 2'b01;
            Memwrite = 1'b0;
            Regwrite = 1'b0;
            RegDest = 1'b0;
            ALUSrc = 1'b0;
            MemtoReg = 1'b0;
            Branch = 1'b1;
        end
        6'b000010: //jal&j
        begin
            Jump = 1'b1;
            ALUop = 2'b00;
            Memwrite = 1'b0;
            Regwrite = 1'b0;
            RegDest = 1'b0;
            ALUSrc = 1'b0;
            MemtoReg = 1'b0;
            Branch = 1'b0;
        end
        default:
        begin
            Jump = 1'b0;
            ALUop = 2'b00;
            Memwrite = 1'b0;
            Regwrite = 1'b0;
            RegDest = 1'b0;
            ALUSrc = 1'b0;
            MemtoReg = 1'b0;
            Branch = 1'b0;
        end
    endcase
end
endmodule

```

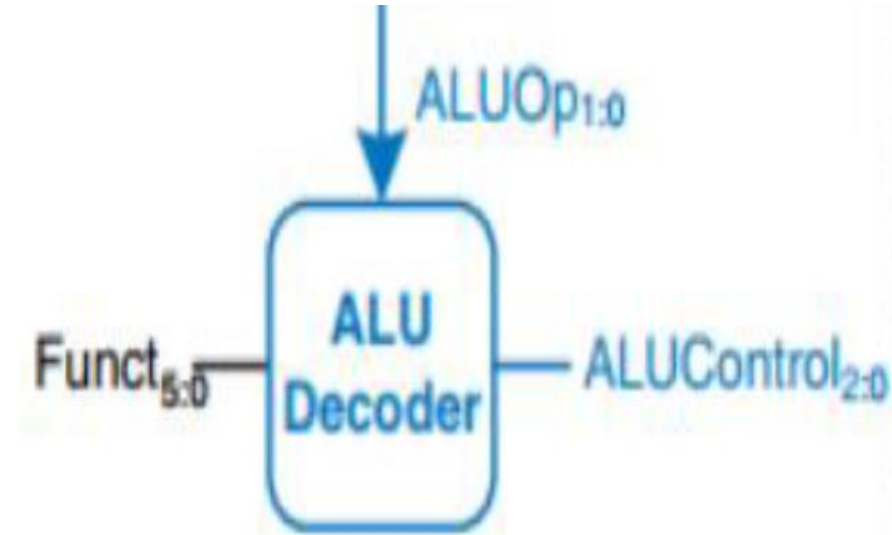


## ALUDecoder

```
module ALUDecoder
(
    input [1:0] ALUOp,
    input [5:0] Funct,
    output reg [2:0] ALUControl
);

always @(*)
begin
    case(ALUOp)
    2'b00: ALUControl = 3'b010;
    2'b01: ALUControl = 3'b100;
    2'b10:
    begin
        casez(Funct)
        6'b100000: ALUControl = 3'b010; //add
        6'b100010: ALUControl = 3'b100; //sub
        6'b101010: ALUControl = 3'b110; //slt
        6'b011100: ALUControl = 3'b101; //mul
        default: ALUControl = 3'b010; //add
        endcase
    end
    default: ALUControl = 3'b010;
    endcase
end

endmodule
```



## PC

```
module PC
#(parameter n = 32)
(
    input clk, reset_n,
    input [n-1:0] pc_bar,
    output [n-1:0] pc
);

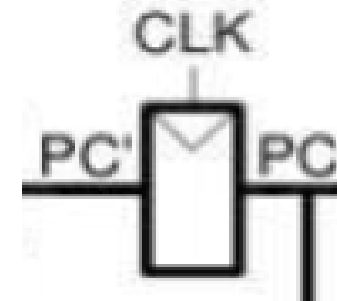
reg [n-1:0] pc_reg, pc_next;

always @(posedge clk, negedge reset_n)
begin
    if (~reset_n)
        pc_reg <= 0;
    else
        pc_reg <= pc_next;
    end
end

always @(*)
begin
    pc_next = pc_bar;
end

assign pc = pc_reg;

endmodule
```



## sign\_extend

```
module sign_extend
(
    input [15:0] imm,
    output reg [31:0] signimm
);

always @(imm)
begin
    signimm = {{16{imm[15]}}},imm};
end

endmodule
```



## sl2

```
module sl2
#(parameter n = 32)
(
    input [n-1:0] signimm,
    output reg [n-1:0] signimmsl2
);

always @(signimm)
begin
    signimmsl2 = signimm << 2;
end

endmodule
```

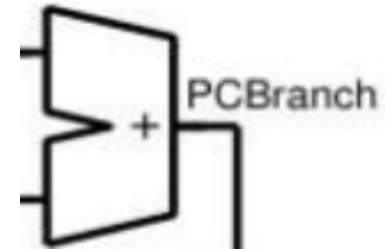


## PCBranch

```
module PCBranch
(
    input [31:0] PCplus4,signimmsl2,
    output reg [31:0] PCBranch
);

always @(*)
begin
    PCBranch = PCplus4 + signimmsl2;
end

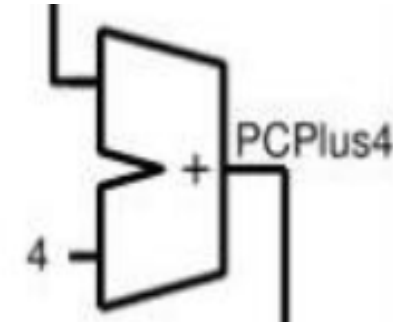
endmodule
```



## add4

```
module add4
  #(parameter n = 32)
  (
    input [n-1:0] pc,
    output reg [n-1:0] PCPlus4
  );

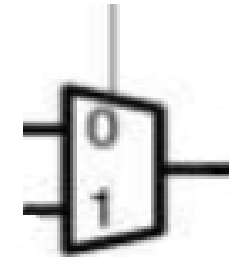
  always @(*)
  begin
    PCPlus4 = pc + 4;
  end
endmodule
```



## mu x 2 1 5bits

```
module mu_x_2_1_5bits
  (
    input s1,
    input [4:0] in_0, in_1,
    output reg [4:0] out
  );

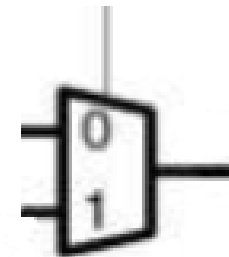
  always @(*)
  begin
    if (1'b1 == s1)
      out = in_1;
    else
      out = in_0;
  end
endmodule
```



## mu x 2 1 32bits

```
module mu_x2_1_32bits
  (
    input s1,
    input [31:0] in_0, in_1,
    output reg [31:0] out
  );

  always @(*)
  begin
    if (1'b1 == s1)
      out = in_1;
    else
      out = in_0;
  end
endmodule
```



# DataPath

```

module DataPath
(
  input clk, reset_n,
  input [31:0] instr,
  input [31:0] ReadData,
  input [2:0] ALUControl,
  input Jump, PCSrc, MemtoReg, ALUSrc, RegWrite, RegDest,
  output zero,
  output [31:0] ALUOut, PC, WriteData
);

  /******************************************************************/
  wire [31:0] PCPlus4;
  wire [31:0] instrS2;
  wire [31:0] SignImm;
  wire [31:0] SignImmS2;
  wire [31:0] PCBranch;
  wire [31:0] SrcA;
  wire [31:0] Out_In_0, Out_IN_0;
  wire [31:0] PCJump = {PCPlus4[31:28], instrS2[27:0]};
  wire [31:0] pc_bar, pc_bar;
  wire [31:0] SrcB;
  wire [31:0] Result;
  wire [4:0] writeReg;
  /******************************************************************/

  PC #(32) B1
  (
    .clk(clk),
    .reset_n(reset_n),
    .pc_bar(pc_bar),
    .pc(PC)
  );

  /******************************************************************/

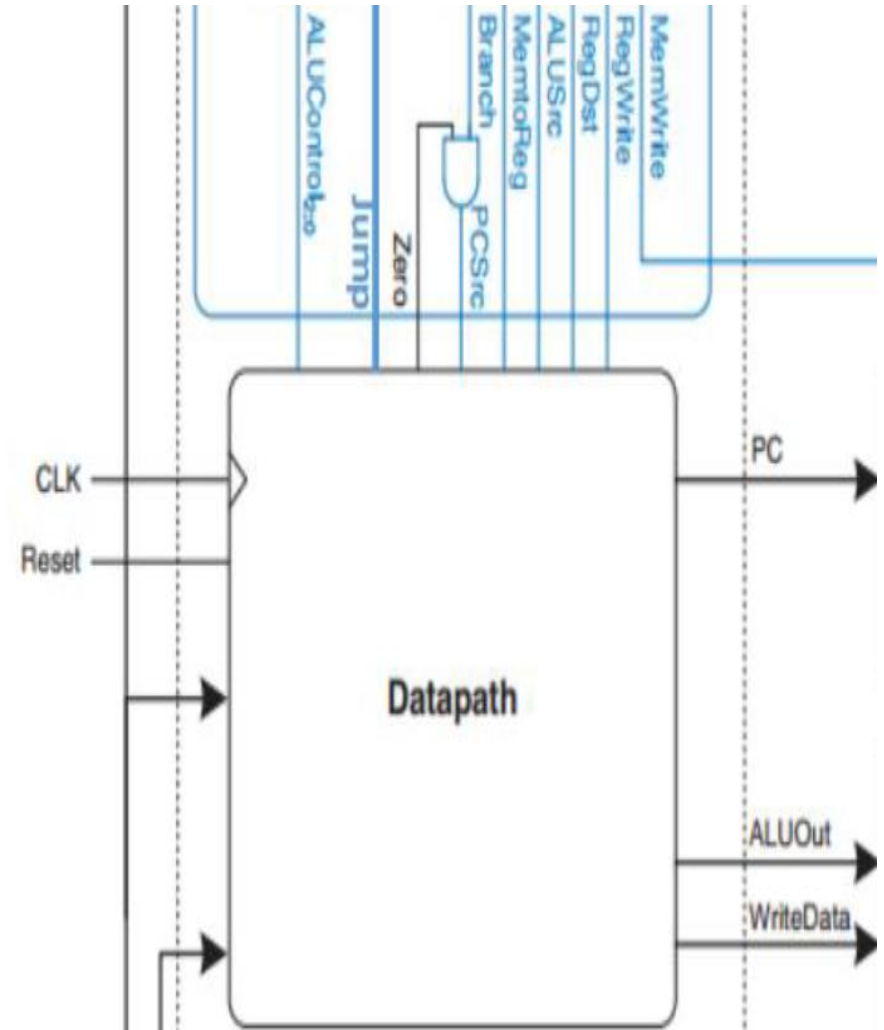
  add4 #(32) B2
  (
    .pc(PC),
    .PCPlus4(PCPlus4)
  );

  /******************************************************************/

  s12 #(32) B3
  (
    .signimm(instr[25:0]),
    .signimms12(instrS2)
  );

  /******************************************************************/

```



```

mu_x_2_1_5bits B4
(
  .s1(RegDest),
  .in_0(instr[20:16]),
  .in_1(instr[15:11]),
  .out(WriteReg)
);

/*****/

sign_extend B5
(
  .imm(instr[15:0]),
  .signimm(SignImm)
);

/*****/

s12 #(.n(32)) B6
(
  .signimm(SignImm),
  .signimms12(SignImms2)
);

/*****/

PCBranch B7
(
  .PCPlus4(PCPlus4),
  .signimms12(SignImms2),
  .PCBranch(PCBranch)
);

/*****/

RegFile #(.width(32), .depth(32)) B8
(
  .clk(clk),
  .WE3(RegWrite),
  .A1(instr[25:21]),
  .A2(instr[20:16]),
  .A3(WriteReg),
  .WD3(Result),
  .RD1(SrcA),
  .RD2(WriteData)
);

/*****/

```



```

mu_x2_1_32bits B9
(
  .sl(PCSrc),
  .in_0(PCPlus4),
  .in_1(PCBranch),
  .out(out_In_0)
);

/*****/

mu_x2_1_32bits B10
(
  .sl(Jump),
  .in_0(out_In_0),
  .in_1(PCJump),
  .out(pc_bar)
);

/*****/

mu_x2_1_32bits B11
(
  .sl(ALUSrc),
  .in_0(writeData),
  .in_1(signImm),
  .out(SrcB)
);

/*****/

ALU #(.n(32)) B12
(
  .A(SrcA),
  .B(SrcB),
  .ALUControl(ALUControl),
  .ALUResult(ALUOut),
  .zeroFlag(zero)
);

/*****/

mu_x2_1_32bits B13
(
  .sl(MemtoReg),
  .in_0(ALUOut),
  .in_1(ReadData),
  .out(Result)
);

```

endmodule

# Processor

```

module DataPath
(
  input clk,reset_n,
  input [31:0] instr,
  input [31:0] ReadData,
  input [2:0] ALUControl,
  input Jump,PCSrc,MemtoReg,ALUSrc,RegWrite,RegDest,
  output zero,
  output [31:0] ALUOut,PC,writeData
);

  /******************************************************************/
  wire [31:0] PCPlus4;
  wire [31:0] instrS2;
  wire [31:0] signImm;
  wire [31:0] signImms2;
  wire [31:0] PCBranch;
  wire [31:0] SrcA;
  wire [31:0] Out_In_0,out_IN_0;
  wire [31:0] PCJump = {PCPlus4[31:28],instrS2[27:0]};
  wire [31:0] pc_bar,pc_bar;
  wire [31:0] SrcB;
  wire [31:0] Result;
  wire [4:0] writeReg;
  /******************************************************************/

  PC #(.(n(32))) B1
  (
    .clk(clk),
    .reset_n(reset_n),
    .pc_bar(pc_bar),
    .pc(PC)
  );

  /******************************************************************/

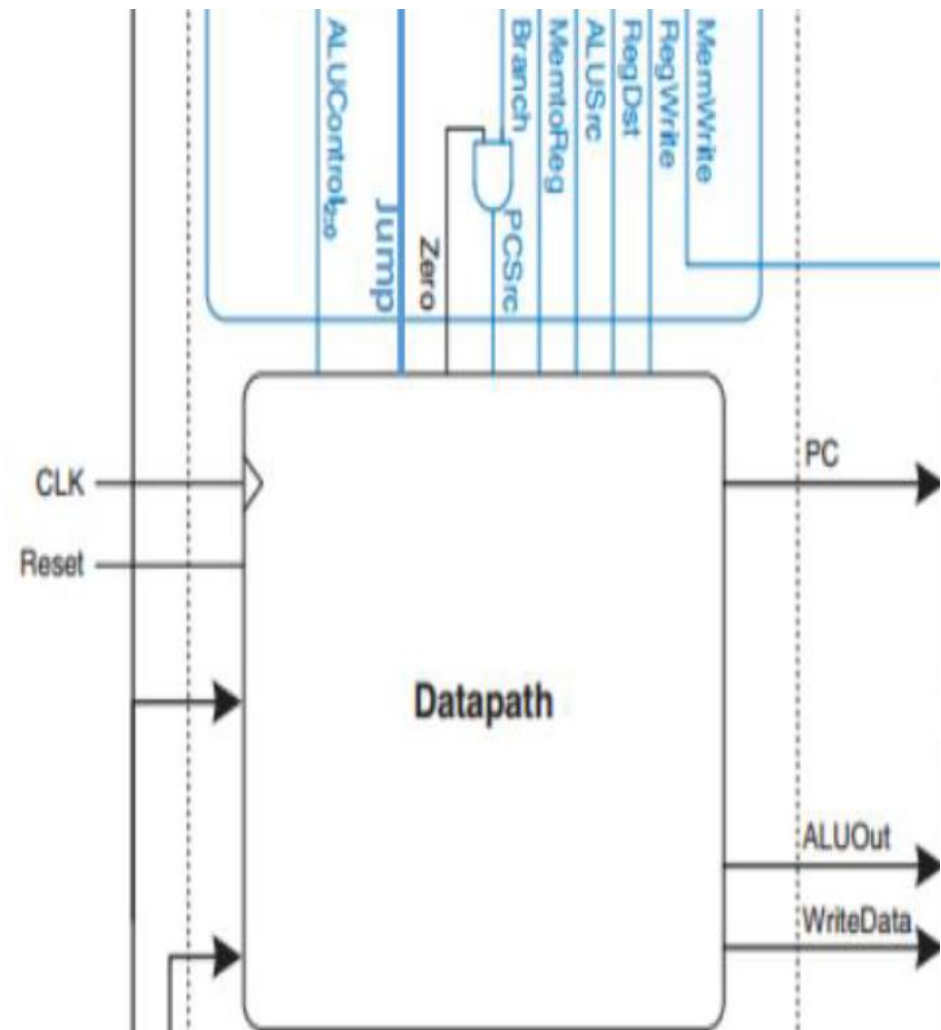
  add4 #(.(n(32))) B2
  (
    .pc(PC),
    .PCPlus4(PCPlus4)
  );

  /******************************************************************/

  s12 #(.(n(32))) B3
  (
    .signimm(instr[25:0]),
    .signimms12(instrS2)
  );

  /******************************************************************/

```



# MIPS32

```

module MIPS32
(
    input clk,reset_n,
    output [15:0] test_value
);

wire MemWrite;
wire [31:0] PC;
wire [31:0] ALUOut_,WriteData,ReadData,instr;

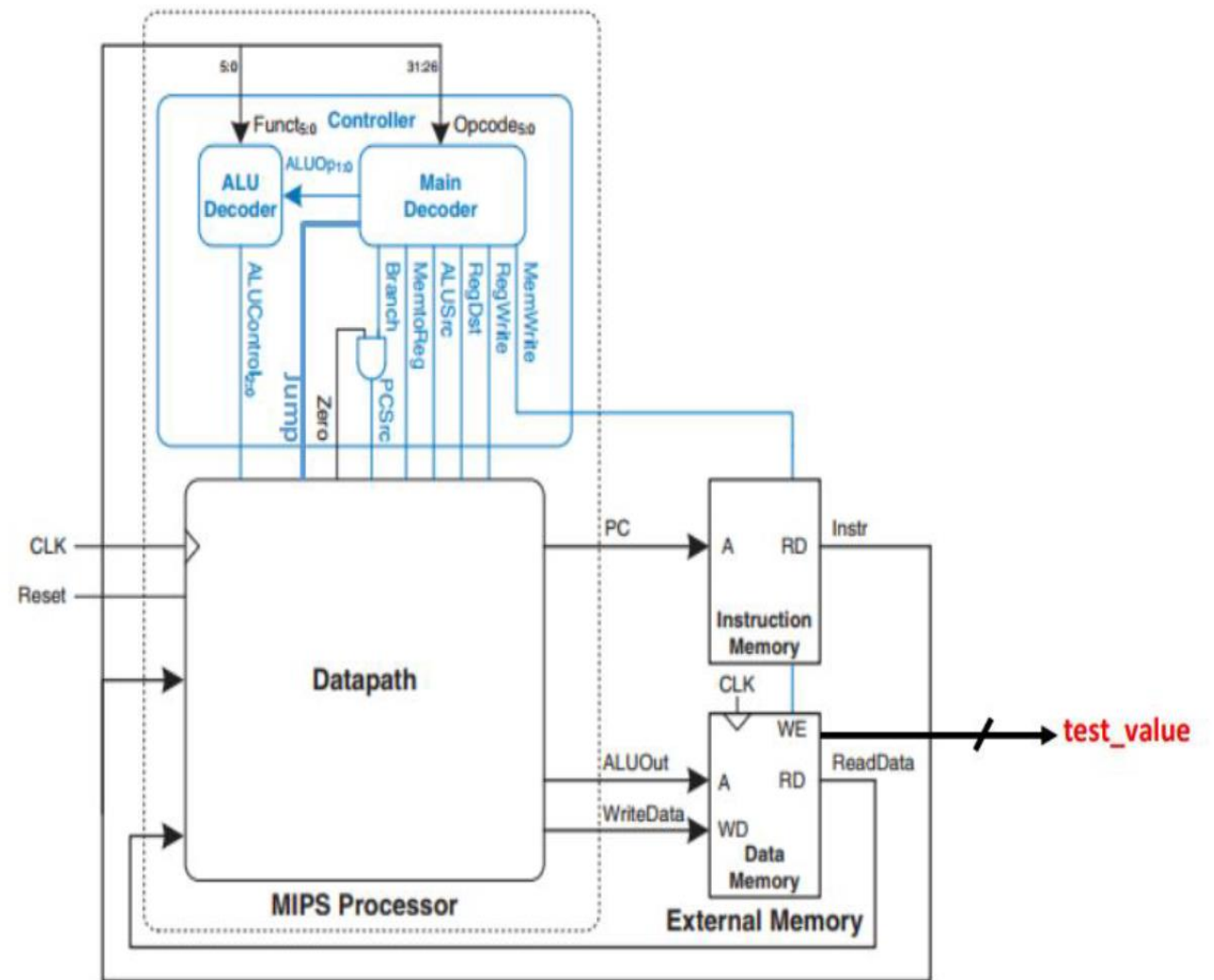
    InstMem #(.width(32),.depth(256)) B1
    (
        .pc_A(PC),
        .instr(instr)
    );

    Processor B2
    (
        .clk(clk),
        .reset_n(reset_n),
        .instr(instr),
        .ReadData(ReadData),
        .PC(PC),
        .ALUOut_(ALUOut_),
        .MemWrite(MemWrite),
        .WriteData(WriteData)
    );

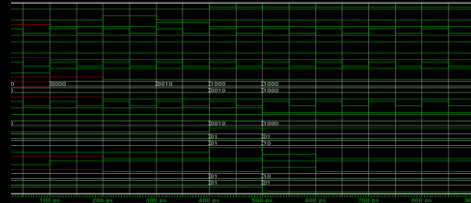
    Data_Mem #(.n(32)) B3
    (
        .clk(clk),
        .WE(MemWrite),
        .A(ALUOut_),
        .WD(WriteData),
        .RD(ReadData),
        .test_value(test_value)
    );

endmodule

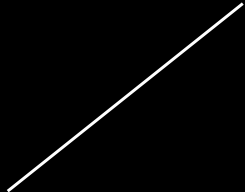
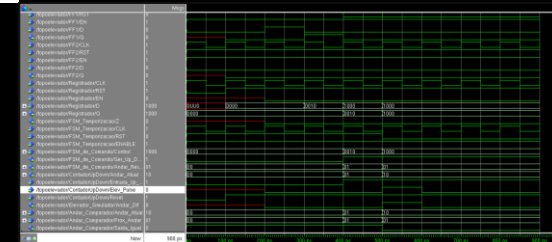
```



```
00008020
20100078      // change the least two significant decimals with the number
00008820
                // change the least two significant decimals with the number,
20110084
08009020
12110006
0211482A
11200002
02308822
08000005
02118022
08000005
00109020
AC120000
```



```
00008020
20100007 // change the least two significant decimals with the number
00008820
20110001
12000003
0230881C
2210FFFF
08000004
AC110000
```



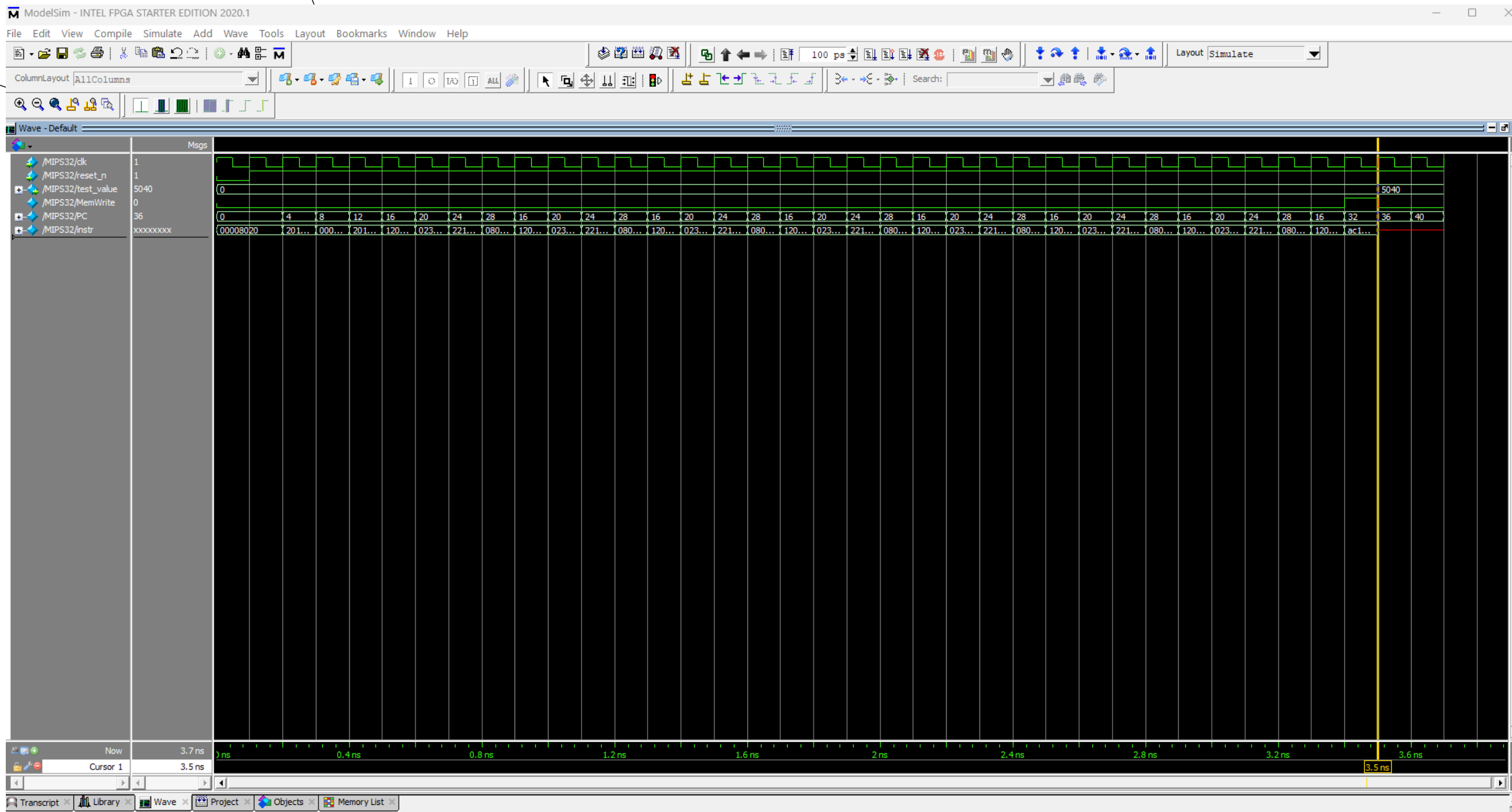
# FACTORIAL OF NUMBER 7

## Machine Code

```
00008020
20100007    // change the least two significant decimals with the number
00008820
20110001
12000003
0230881C
2210FFFF
08000004
AC110000
```

## Data Mem

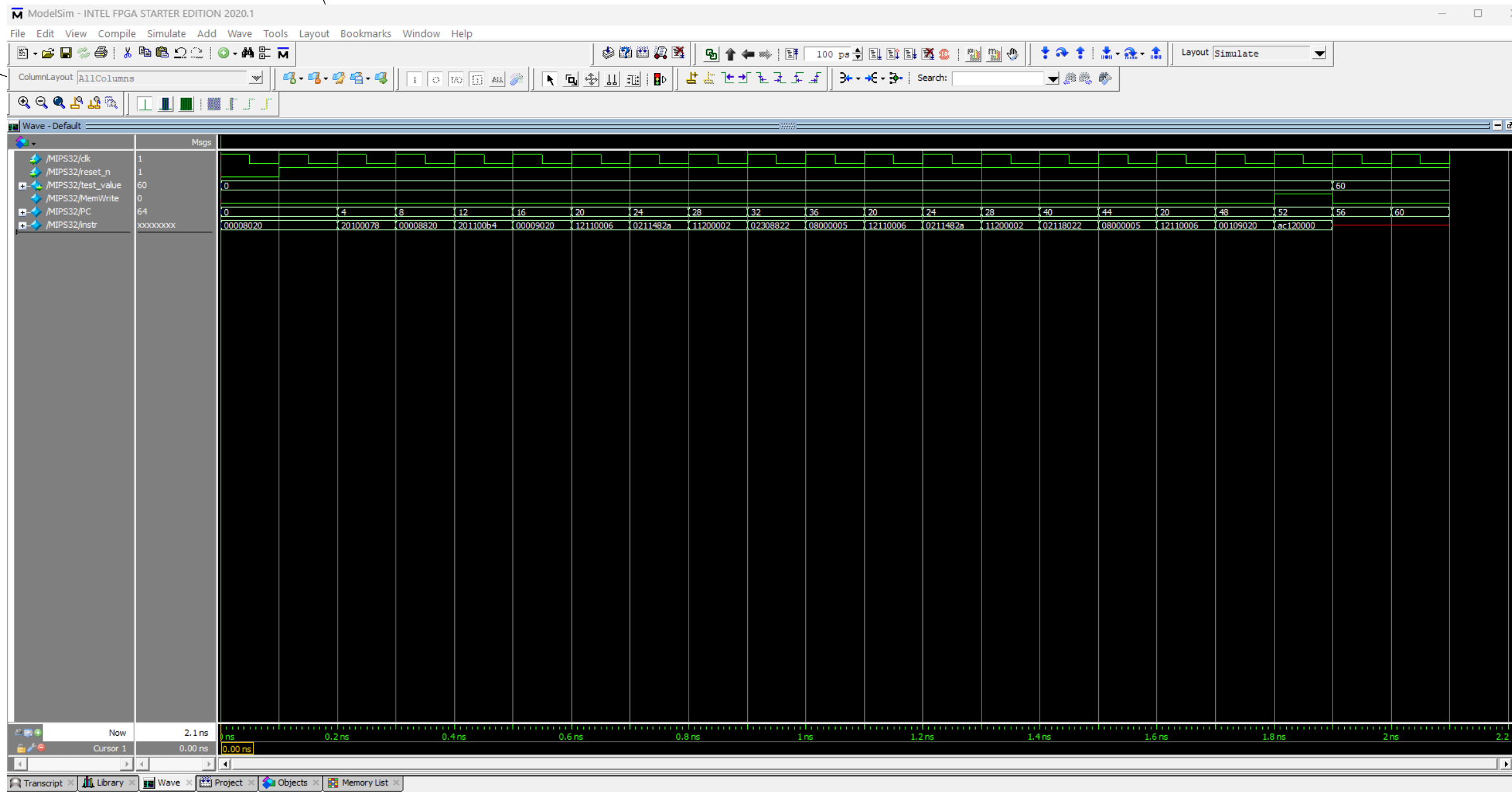
Memory Data - /MIPS32/B3/Data_Mem - Default																															
00000000	000013b0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0000001c	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000038	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000054	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000070	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0000008c	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000000a8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000000c4	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000000e0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000000fc	00000000	00000000	00000000	00000000																											
00000118																															



## Machine Code

```
// change the least two significant decimals with the number
// change the least two significant decimals with the number.
```

[illegible]







# THANK YOU

Mohamed Dawod

E-mail → [muhmddawod@gmail.com](mailto:muhmddawod@gmail.com)

Github repo for this project → [MohamedDawod29/32bits-single-cycle-MIPs \(github.com\)](https://github.com/MohamedDawod29/32bits-single-cycle-MIPs)

Linkedin → <https://www.linkedin.com/in/muhmd-dawod-79198522a>