GEBZE TECHNICAL UNIVERSITY

Computer Engineering Department

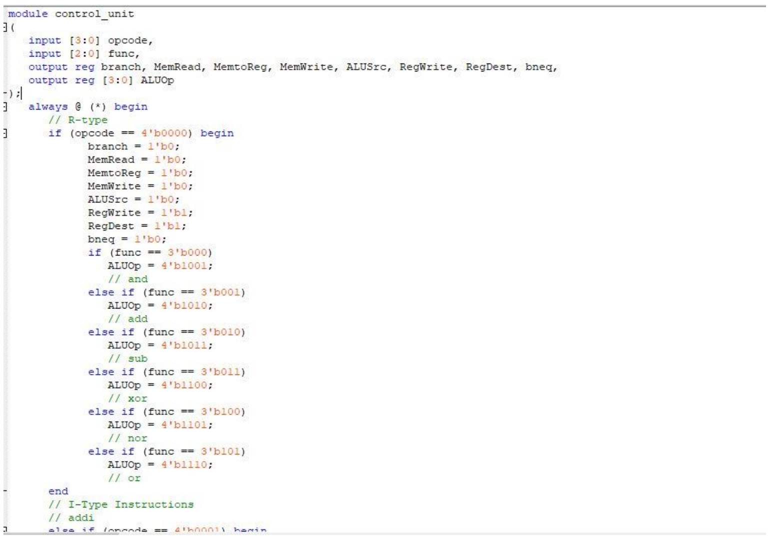
CSE 331/501 FINAL PROJECT – MiniMIPS Design

ERSİN ALÇİN

1801042692

<u>Control_unit:</u>

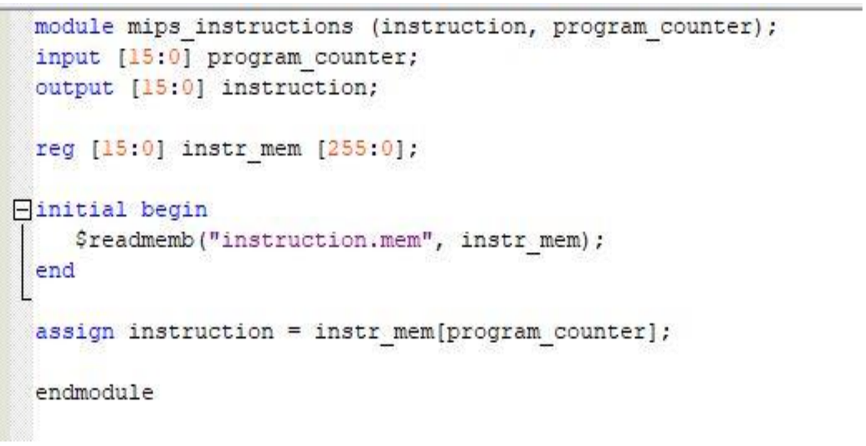
This module takes the opcode and function code as input. According to these inputs, it generates the control signal required for the instruction to work correctly.



```
module control_unit
(
   input [3:0] opcode,
   input [2:0] func,
   output reg branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, RegDest, bneq,
   output reg [3:0] ALUOp
);
   always @ (*) begin
      // R-type
      if (opcode == 4'b0000) begin
            branch = 1'b0;
            MemRead = 1'b0;
            MemtoReg = 1'b0;
            MemWrite = 1'b0;
            ALUSrc = 1'b0;
            RegWrite = 1'b1;
            RegDest = 1'b1;
            bneq = 1'b0;
            if (func == 3'b000)
               ALUOp = 4'b1001;
               // and
            else if (func == 3'b001)
               ALUOp = 4'b1010;
               // add
            else if (func == 3'b010)
               ALUOp = 4'b1011;
               // sub
            else if (func == 3'b011)
               ALUOp = 4'b1100;
               // xor
            else if (func == 3'b100)
               ALUOp = 4'b1101;
               // nor
            else if (func == 3'b101)
               ALUOp = 4'b1110;
               // or
      end
      // I-Type Instructions
      // addi
      else if (opcode == 4'b0001) begin
```

<u>Mips_instructions:</u>

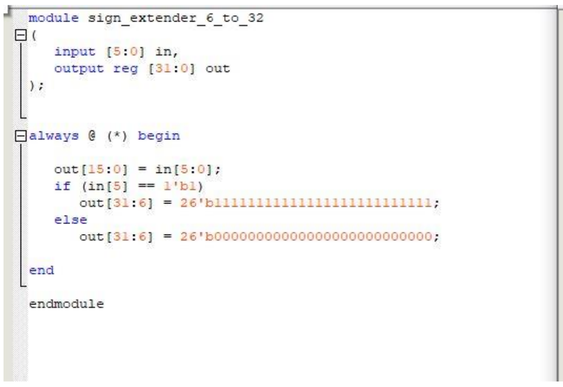
In this module, the instructions in the instruction.mem file are read into the instr_mem two-dimensional array. The instruction value in the program_counter value that comes as input to this module is given to the output of this module.



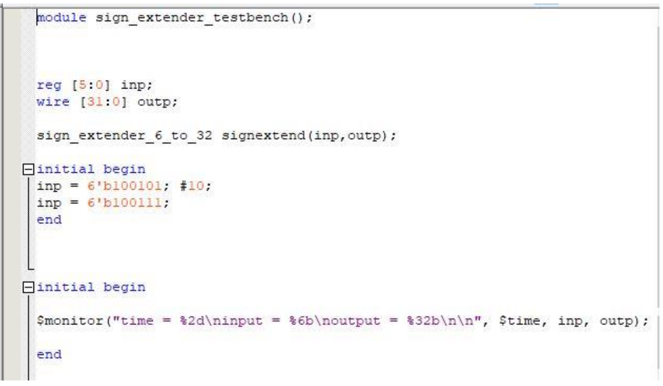
```
module mips_instructions (instruction, program_counter);
input [15:0] program_counter;
output [15:0] instruction;

reg [15:0] instr_mem [255:0];

initial begin
   $readmemb("instruction.mem", instr_mem);
end

assign instruction = instr_mem[program_counter];

endmodule
```

<u>Sign_extender_6_to_32:</u>

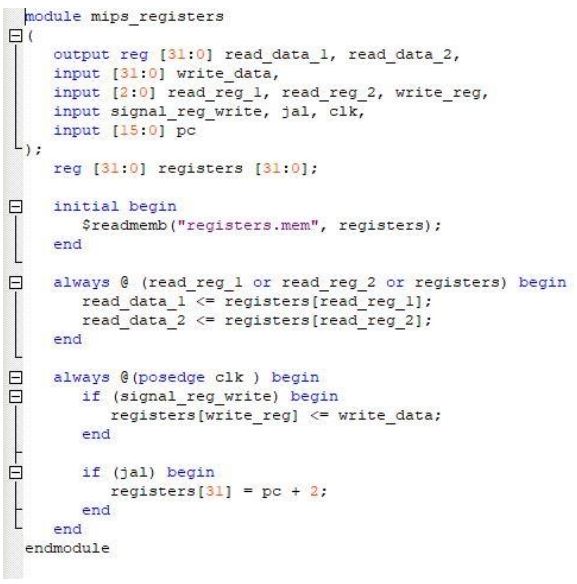
This module is used to extend the 16-bit immediate part of the instruction to 32 bits. I also wrote the testbench code of this modüle



```
module sign_extender_6_to_32
(
   input [5:0] in,
   output reg [31:0] out
);

always @ (*) begin

   out[15:0] = in[5:0];
   if (in[5] == 1'b1)
      out[31:6] = 26'b11111111111111111111111111;
   else
      out[31:6] = 26'b00000000000000000000000000;

end

endmodule
```



```
module sign_extender_testbench();



reg [5:0] inp;
wire [31:0] outp;

sign_extender_6_to_32 signextend(inp,outp);

initial begin
inp = 6'b100101; #10;
inp = 6'b100111;
end


initial begin

$monitor("time = %2d\ninput = %6b\noutput = %32b\n\n", $time, inp, outp);

end
```



```
module mips_registers
(
   output reg [31:0] read_data_1, read_data_2,
   input [31:0] write_data,
   input [2:0] read_reg_1, read_reg_2, write_reg,
   input signal_reg_write, jal, clk,
   input [15:0] pc
);
   reg [31:0] registers [31:0];

   initial begin
      $readmemb("registers.mem", registers);
   end

   always @ (read_reg_1 or read_reg_2 or registers) begin
      read_data_1 <= registers[read_reg_1];
      read_data_2 <= registers[read_reg_2];
   end

   always @(posedge clk ) begin
      if (signal_reg_write) begin
         registers[write_reg] <= write_data;
      end

      if (jal) begin
         registers[31] = pc + 2;
      end
   end
endmodule
```

<u>Mips_registers:</u>

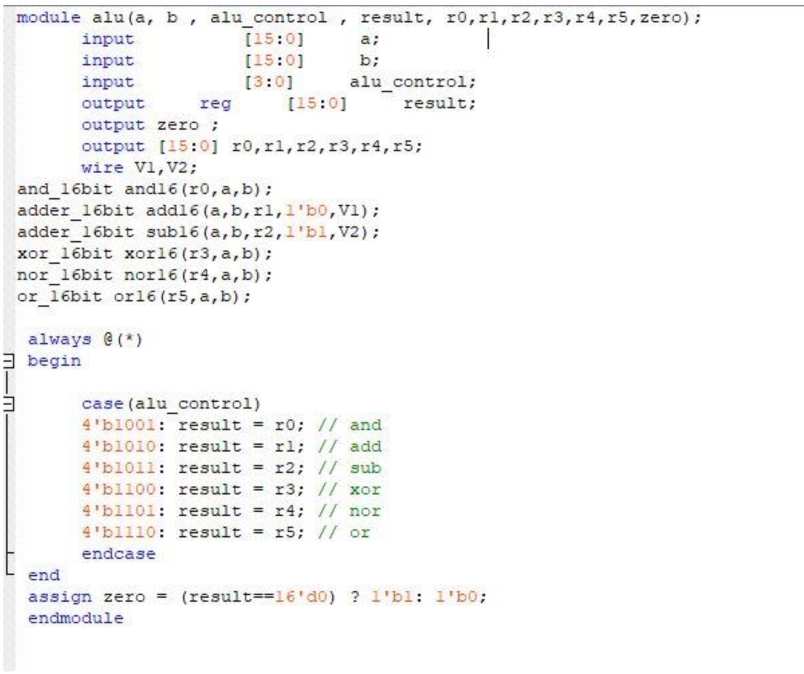
In this module, 32 bits of 32 data registers in registers.mem file are read into a two-dimensional array. The contents of registers rs and rt are read.

<u>Mux_2_1_16bit:</u>

This module is used in two places. The ALUSrc signal comes as an input to select the input of the ALU module. If ALUSrc is 1, the sign extended immediate value is given to the output of this module. If it is 0, the content of the rt value is given to the output of this module. This module is also used to select the data to be written to the register. If the MemToReg signal is 1, the value read from the memory is given to the output of this module. If it is 0, the output of the ALU is given to the output of this module.

ALU:

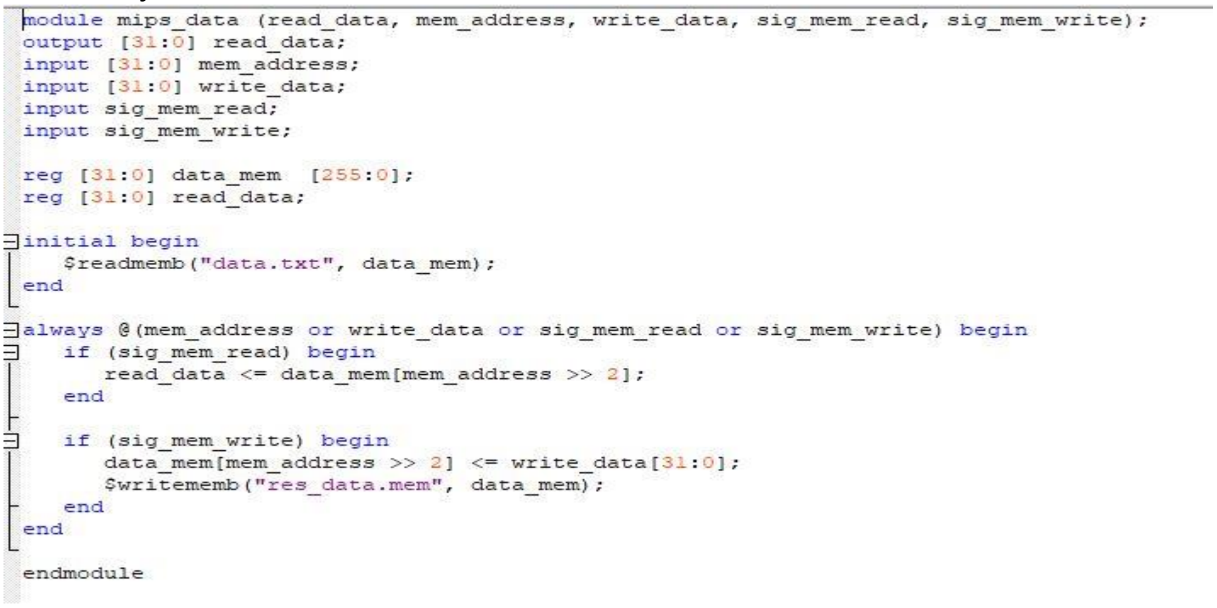
This module is the module used to perform arithmetic operations. According to the ALUOp control that comes as an input to this module, the operation to be done is selected and the result of that operation is given to the output of this module. Also in this module, the zero and overflow bits are given to the output of this module. The zero bit is used to make the branch decision.



```
module alu(a, b , alu_control , result, r0,r1,r2,r3,r4,r5,zero);
        input           [15:0]      a;
        input           [15:0]      b;
        input           [3:0]      alu_control;
        output     reg      [15:0]      result;
        output zero ;
        output [15:0] r0,r1,r2,r3,r4,r5;
        wire V1,V2;
and_16bit and16(r0,a,b);
adder_16bit add16(a,b,r1,1'b0,V1);
adder_16bit sub16(a,b,r2,1'b1,V2);
xor_16bit xor16(r3,a,b);
nor_16bit nor16(r4,a,b);
or_16bit or16(r5,a,b);

 always @(*)
 begin

        case(alu_control)
        4'b1001: result = r0; // and
        4'b1010: result = r1; // add
        4'b1011: result = r2; // sub
        4'b1100: result = r3; // xor
        4'b1101: result = r4; // nor
        4'b1110: result = r5; // or
        endcase
 end
 assign zero = (result==16'd0) ? 1'b1: 1'b0;
 endmodule
```

Mips_data:

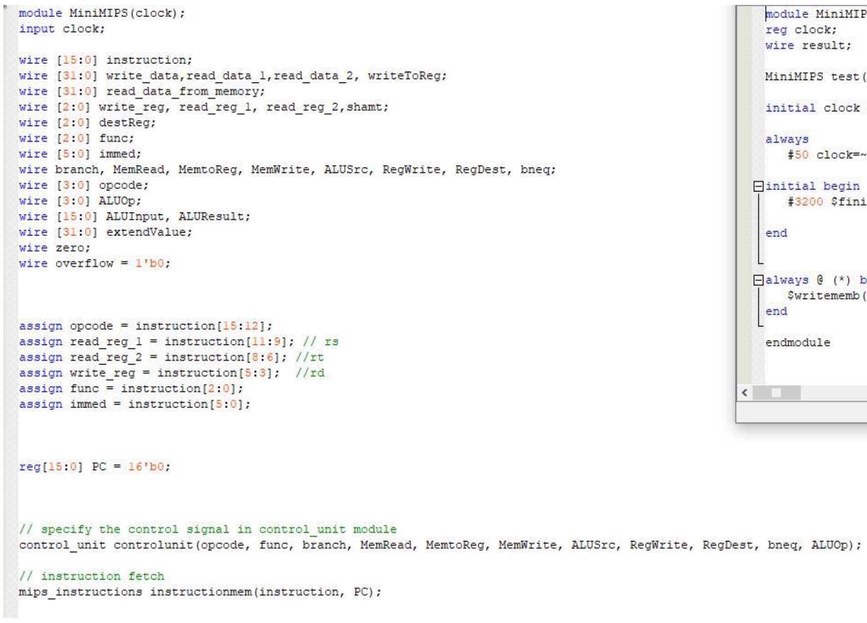
In this module, firstly, 32bit data read from the data.mem file is read into the data_mem two-dimensional array. If the sig_mem_read signal is 1, the reading is made from the address indicated by the mem_address value. At the end of the cycle, if the sig_mem_write signal is 1, the memory is written.



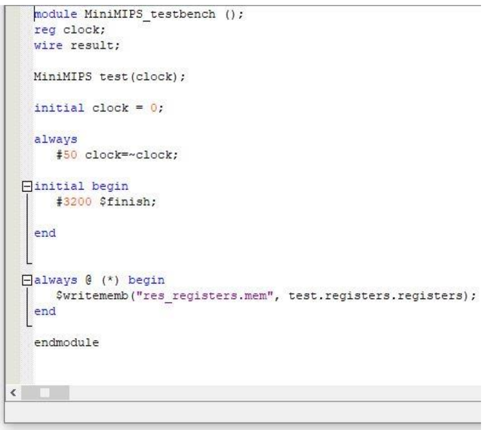
```
module mips_data (read_data, mem_address, write_data, sig_mem_read, sig_mem_write);
output [31:0] read_data;
input [31:0] mem_address;
input [31:0] write_data;
input sig_mem_read;
input sig_mem_write;

reg [31:0] data_mem  [255:0];
reg [31:0] read_data;

initial begin
    $readmemb("data.txt", data_mem);
end

always @(mem_address or write_data or sig_mem_read or sig_mem_write) begin
    if (sig_mem_read) begin
        read_data <= data_mem[mem_address >> 2];
    end

    if (sig_mem_write) begin
        data_mem[mem_address >> 2] <= write_data[31:0];
        $writememb("res_data.mem", data_mem);
    end
end

endmodule
```

<u>MiniMIPS:</u>

This module is top-level module. In this module, all modules are connected to each other. At the end of the cycle, the appropriate value is written to the program counter.



```
module MiniMIPS(clock);
input clock;

wire [15:0] instruction;
wire [31:0] write_data,read_data_1,read_data_2, writeToReg;
wire [31:0] read_data_from_memory;
wire [2:0] write_reg, read_reg_1, read_reg_2,shamt;
wire [2:0] destReg;
wire [2:0] func;
wire [5:0] immed;
wire branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, RegDest, bneq;
wire [3:0] opcode;
wire [3:0] ALUOp;
wire [15:0] ALUInput, ALUResult;
wire [31:0] extendValue;
wire zero;
wire overflow = 1'b0;



assign opcode = instruction[15:12];
assign read_reg_1 = instruction[11:9]; // rs
assign read_reg_2 = instruction[8:6]; //rt
assign write_reg = instruction[5:3];  //rd
assign func = instruction[2:0];
assign immed = instruction[5:0];



reg[15:0] PC = 16'b0;



// specify the control signal in control_unit module
control_unit controlunit(opcode, func, branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, RegDest, bneq, ALUOp);

// instruction fetch
mips_instructions instructionmem(instruction, PC);
```

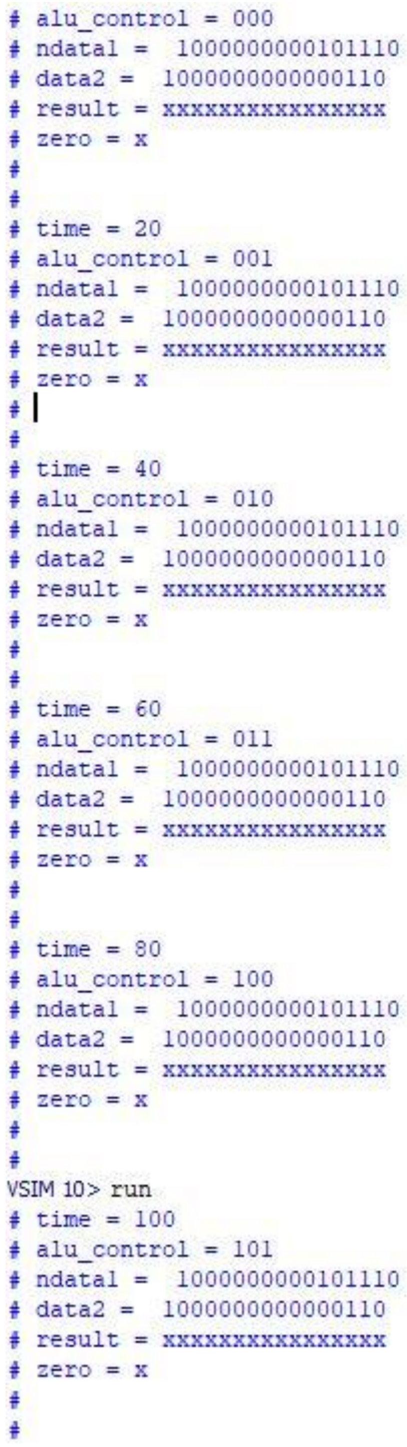


```
module MiniMIPS_testbench ();
reg clock;
wire result;

MiniMIPS test(clock);

initial clock = 0;

always
   #50 clock=~clock;

initial begin
   #3200 $finish;

end


always @ (*) begin
   $writememb("res_registers.mem", test.registers.registers);
end

endmodule
```

RESULTS

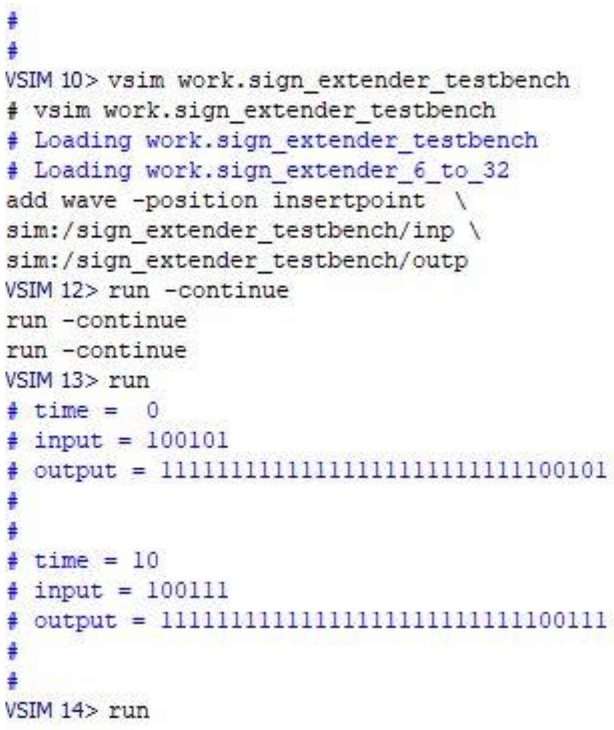
In this project, there is a testbench to test the ball module. In this testbench, the clock is reset from 0. The clock is changed every 100 nanoseconds. Every time the clock is one, it is the end of the cyle.

ALU TESTBENCH



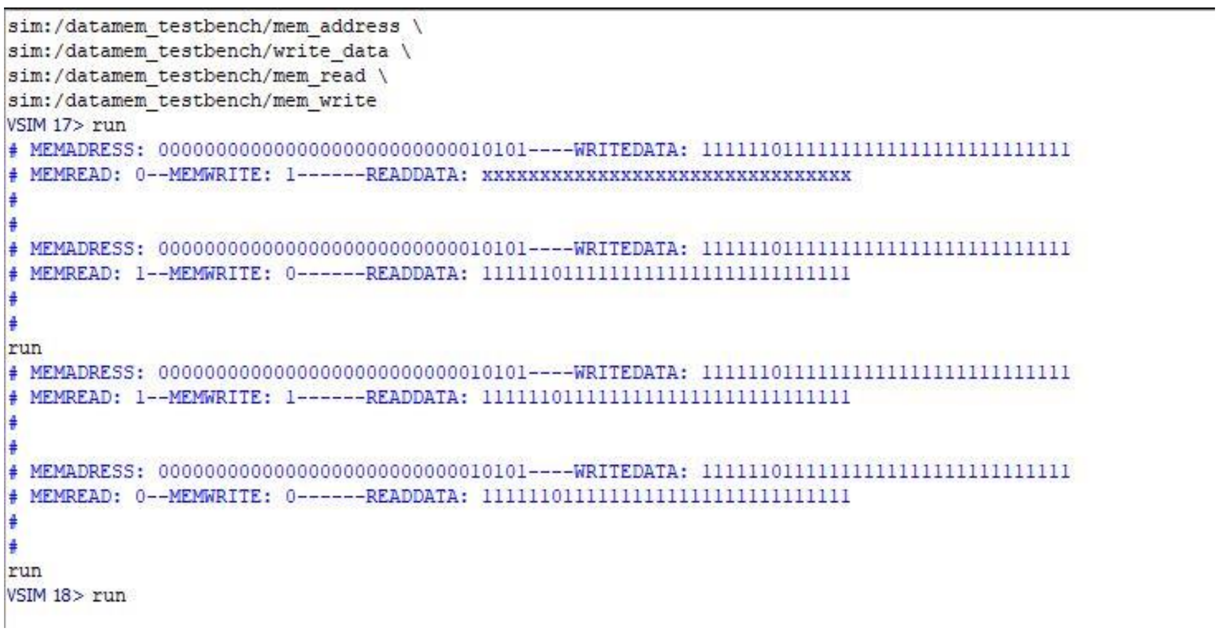
```
# alu_control = 000
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
# time = 20
# alu_control = 001
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
# time = 40
# alu_control = 010
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
# time = 60
# alu_control = 011
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
# time = 80
# alu_control = 100
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
VSIM 10> run
# time = 100
# alu_control = 101
# ndata1 =  1000000000101110
# data2 =  1000000000000110
# result = xxxxxxxxxxxxxxxx
# zero = x
#
#
```

SİGN EXTENDER TESTBENCH



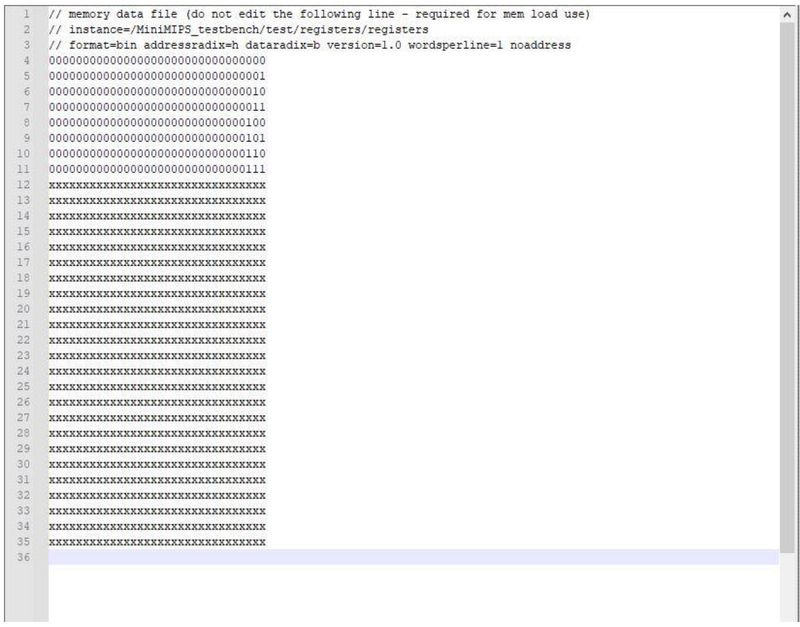
```
#
#
VSIM 10> vsim work.sign_extender_testbench
# vsim work.sign_extender_testbench
# Loading work.sign_extender_testbench
# Loading work.sign_extender_6_to_32
add wave -position insertpoint  \
sim:/sign_extender_testbench/inp \
sim:/sign_extender_testbench/outp
VSIM 12> run -continue
run -continue
run -continue
VSIM 13> run
# time =  0
# input = 100101
# output = 11111111111111111111111111100101
#
#
# time = 10
# input = 100111
# output = 11111111111111111111111111100111
#
#
VSIM 14> run
```

DATA MEMORY TESTBENCH

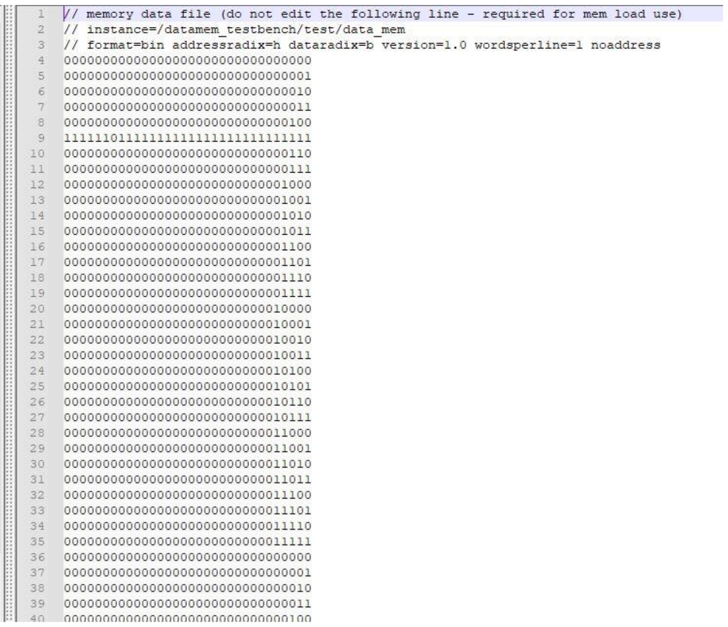


```
sim:/datamem_testbench/mem_address \
sim:/datamem_testbench/write_data \
sim:/datamem_testbench/mem_read \
sim:/datamem_testbench/mem_write
VSIM 17> run
# MEMADRESS: 00000000000000000000000000010101----WRITEDATA: 11111011111111111111111111111111
# MEMREAD: 0--MEMWRITE: 1------READDATA: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
#
#
# MEMADRESS: 00000000000000000000000000010101----WRITEDATA: 11111011111111111111111111111111
# MEMREAD: 1--MEMWRITE: 0------READDATA: 11111011111111111111111111111111
#
#
run
# MEMADRESS: 00000000000000000000000000010101----WRITEDATA: 11111011111111111111111111111111
# MEMREAD: 1--MEMWRITE: 1------READDATA: 11111011111111111111111111111111
#
#
# MEMADRESS: 00000000000000000000000000010101----WRITEDATA: 11111011111111111111111111111111
# MEMREAD: 0--MEMWRITE: 0------READDATA: 11111011111111111111111111111111
#
#
run
VSIM 18> run
```

REGISTER AND DATA OUPUT .MEM



```
// memory data file (do not edit the following line - required for mem load use)
// instance=/MiniMIPS_testbench/test/registers/registers
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=1 noaddress
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```



```
// memory data file (do not edit the following line - required for mem load use)
// instance=/datamem_testbench/test/data_mem
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=1 noaddress
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
11111101111111111111111111111111
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001000
00000000000000000000000000001001
00000000000000000000000000001010
00000000000000000000000000001011
00000000000000000000000000001100
00000000000000000000000000001101
00000000000000000000000000001110
00000000000000000000000000001111
00000000000000000000000000010000
00000000000000000000000000010001
00000000000000000000000000010010
00000000000000000000000000010011
00000000000000000000000000010100
00000000000000000000000000010101
00000000000000000000000000010110
00000000000000000000000000010111
00000000000000000000000000011000
00000000000000000000000000011001
00000000000000000000000000011010
00000000000000000000000000011011
00000000000000000000000000011100
00000000000000000000000000011101
00000000000000000000000000011110
00000000000000000000000000011111
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
```