University of Moratuwa, Sri Lanka

Faculty of Engineering

Department of Electronic and Telecommunication Engineering

Semester 5

EN3021 Digital System Design

Non-pipelined Single Stage (Cycle) CPU Design

G.K.M.I.D.Rajarathna

200500L

Introduction

Non-pipelined Single Cycle CPU Design

This individual project was at its core about crafting a 32-bit non-pipelined RISC-V processor using an FPGA platform. The central thrust of the project lay in gaining a deep understanding of computer architecture, particularly delving into microprogramming and the execution of instructions within the RV32I instruction set. A notable aspect of this project was the implementation of two new instructions, which served as a practical showcase of a solid grasp of micro-architecture and efficient hardware utilization.

In terms of design, the project utilized a combinational approach, also known as a microprogrammed processor. In this approach, instruction decoding was handled through microcode—a sequence of control microinstructions stored in a control store or ROM (Read-Only Memory). This microcode, selected based on the extracted opcode from the instruction, directed the processor's operations.

The primary objective of the project was to develop a fully operational RISC-V processor that could handle essential instruction types such as R, I, S, and SB. Beyond this, the project ambitiously included the integration of custom instructions, MEMCOPY and MUL, designed to augment the processor's functionality.

The project's post-implementation phase involved rigorous testing procedures. This entailed RTL simulations to scrutinize the processor's behavior in a simulated environment and to ensure the precise execution of instructions. Subsequently, the processor was deployed onto an FPGA board for real-world testing, solidifying its practical functionality and robustness in a hardware context.

Processor Implementation overview

In this implementation, a three-bus architecture was utilized. The design incorporated three primary components: the controller, ALU (Arithmetic Logic Unit) controller, and datapath. Several adjustments were made to these components to support the new instructions. The original design was based on a load and store architecture, meaning only load and store instructions could access memory. However, in this design, a new instruction named MEMCOPY was introduced to facilitate memory copying tasks.

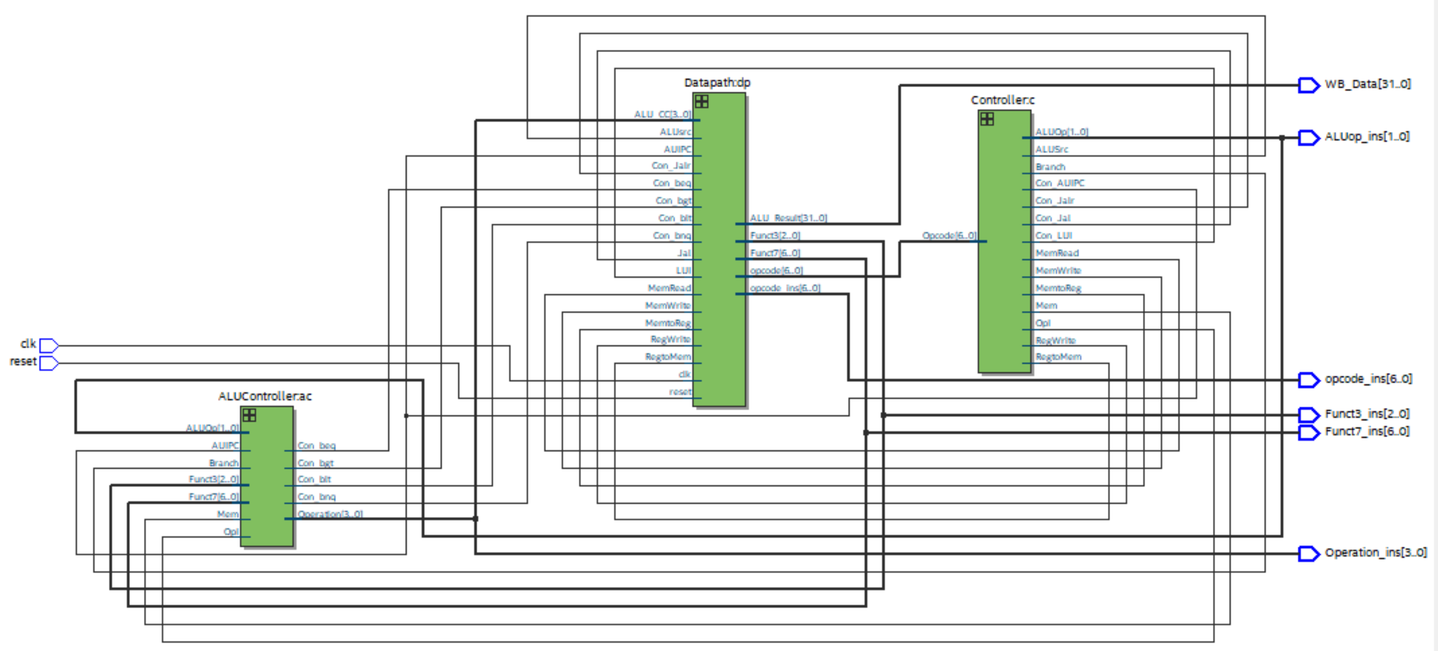
Additionally, another new instruction, MUL, was added to handle multiplication, which was not part of the processor's initial design. The entire architecture is built around 32-bit registers, and for multiplication, it is constrained to 16-bit numbers.

In this process, the instruction is accessed using the program counter, which is used to fetch a specific instruction from the instruction memory. Subsequently, the controller examines the instruction's opcode and generates the required control signals based on the instruction type. Additionally, a separate ALU (Arithmetic Logic Unit) controller is designed to handle specific ALU-related tasks, with guidance from the main controller.

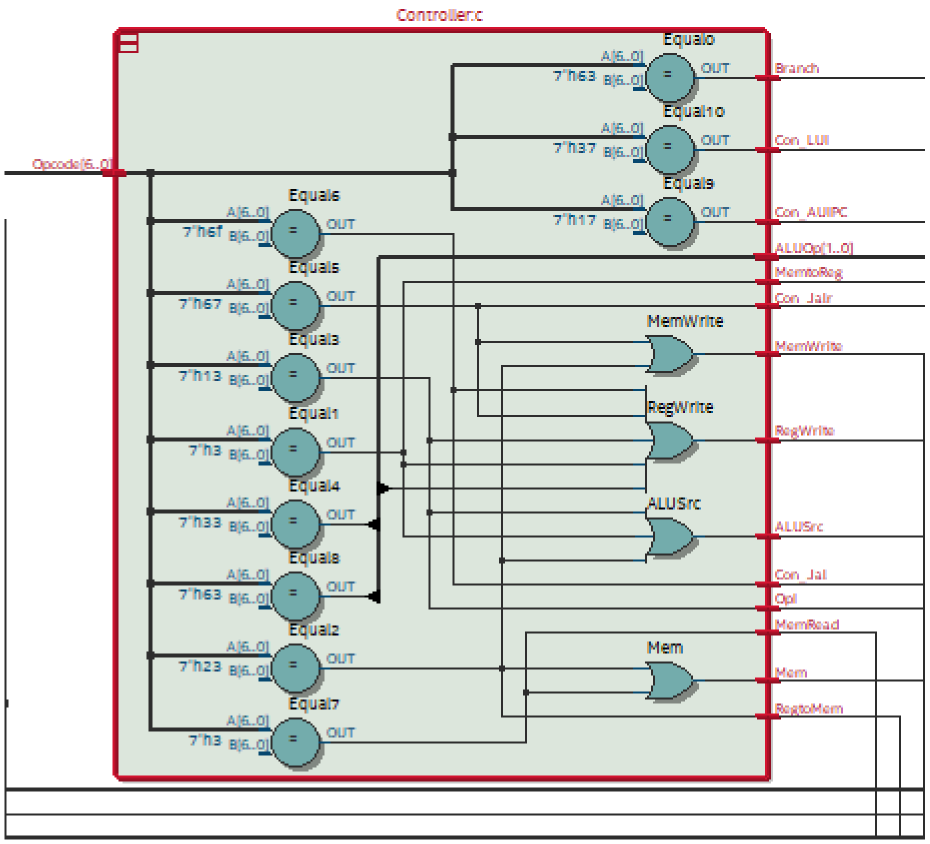
Following this, the data path is established, and it performs dedicated tasks in synchronization with the clock, relying on the control signals generated by the controller.

In the design, inputs include the clock and reset signals, while outputs consist of the ALU results and other outputs for illustrative purposes. When the reset signal is set to a high state, all the values in the registers are reset to zero.

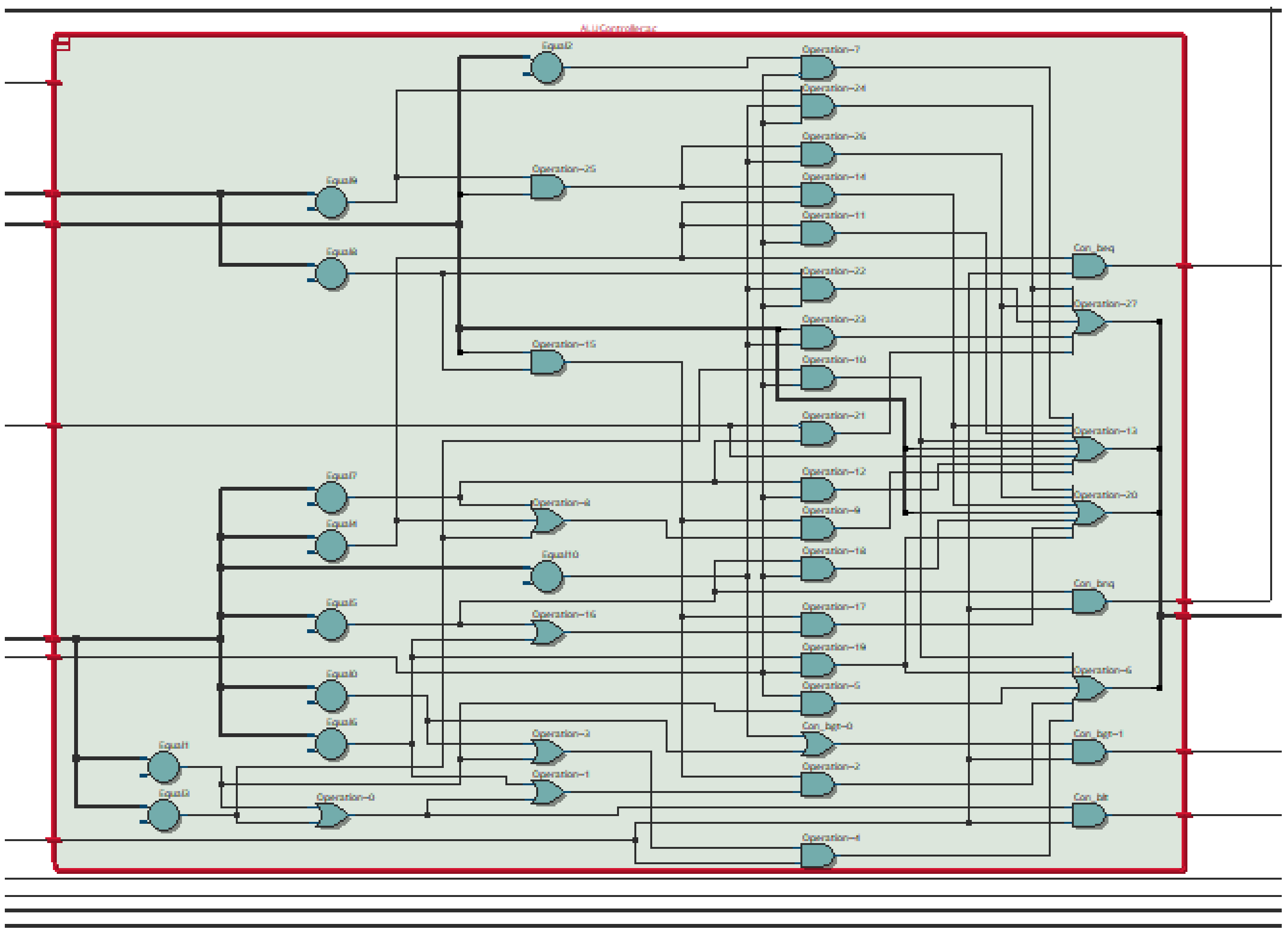
Processor Implementation



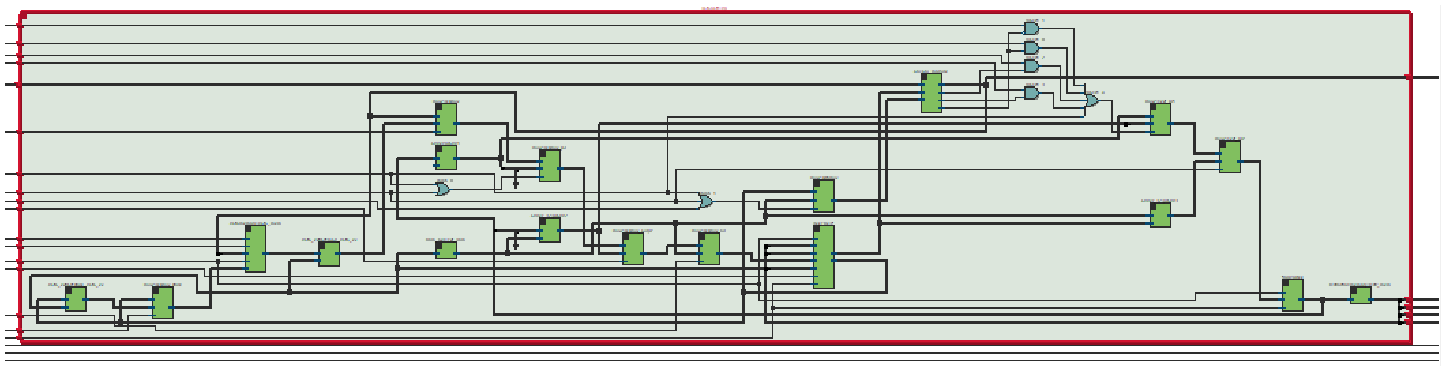
Main controller



Alu controller



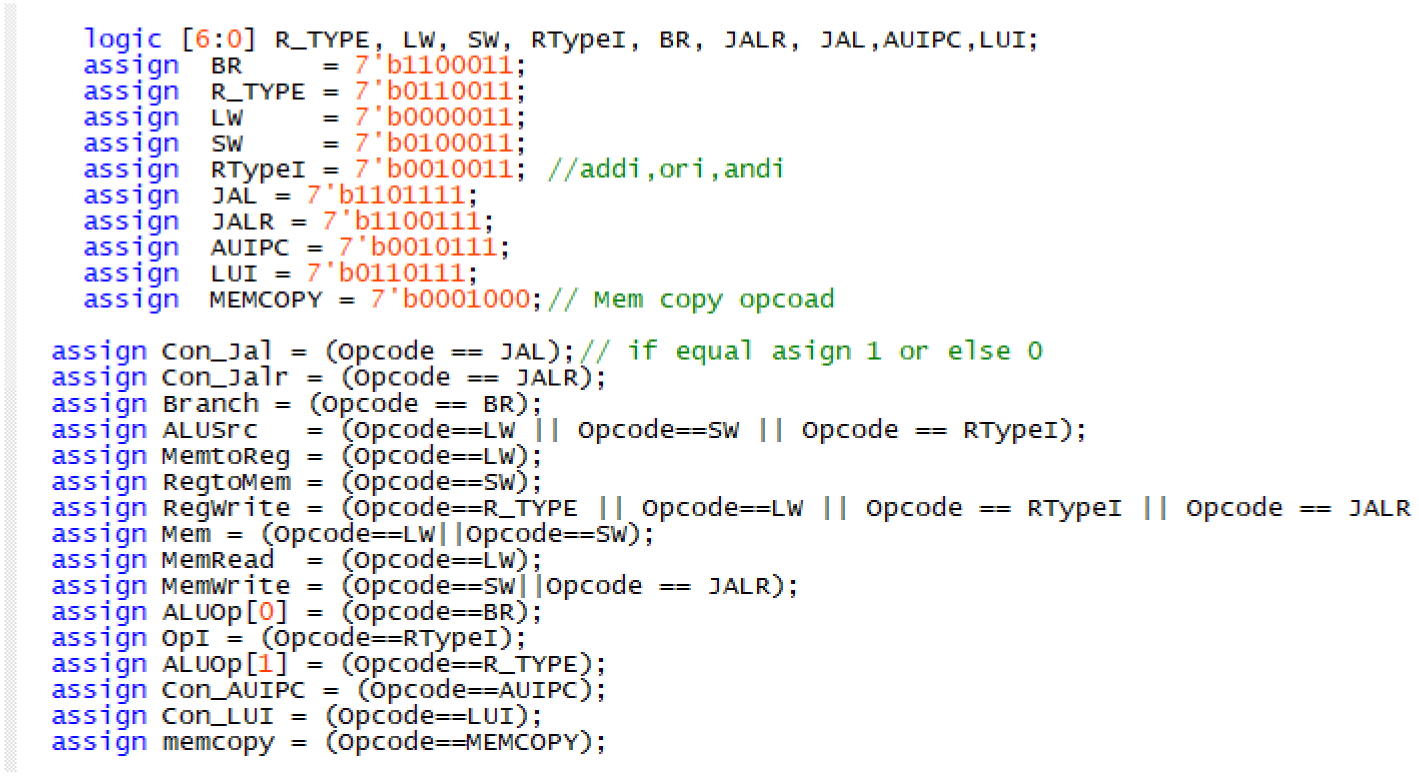
Datapath



In this implementation, all instructions, including those from the RISC-V architecture (R, I, S, SB, etc.), are stored in the instruction memory for testing purposes. Furthermore, new instructions, specifically MUL and MEMCOPY, have been added to the instruction set to expand the capabilities of the processor.

Controller

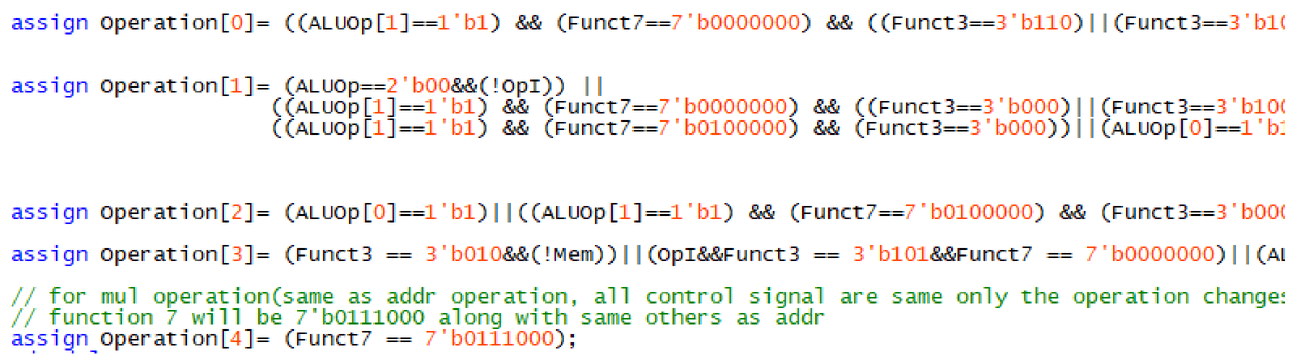
The controller takes the 7-bit opcode of each instruction and uses it to generate the necessary control signals for various modules. It distinguishes between different types of instructions (R, I, S, SB, etc.) and generates the corresponding control signals for each type.



```
    logic [6:0] R_TYPE, LW, SW, RTypeI, BR, JALR, JAL,AUIPC,LUI;
    assign  BR      = 7'b1100011;
    assign  R_TYPE = 7'b0110011;
    assign  LW      = 7'b0000011;
    assign  SW      = 7'b0100011;
    assign  RTypeI = 7'b0010011; //addi,ori,andi
    assign  JAL = 7'b1101111;
    assign  JALR = 7'b1100111;
    assign  AUIPC = 7'b0010111;
    assign  LUI = 7'b0110111;
    assign  MEMCOPY = 7'b0001000;// Mem copy opcoad

  assign Con_Jal = (Opcode == JAL);// if equal asign 1 or else 0
  assign Con_Jalr = (Opcode == JALR);
  assign Branch = (Opcode == BR);
  assign ALUSrc   = (Opcode==LW || Opcode==SW || Opcode == RTypeI);
  assign MemtoReg = (Opcode==LW);
  assign RegtoMem = (Opcode==SW);
  assign RegWrite = (Opcode==R_TYPE || Opcode==LW || Opcode == RTypeI || Opcode == JALR
  assign Mem = (Opcode==LW||Opcode==SW);
  assign MemRead  = (Opcode==LW);
  assign MemWrite = (Opcode==SW||Opcode == JALR);
  assign ALUOp[0] = (Opcode==BR);
  assign OpI = (Opcode==RTypeI);
  assign ALUOp[1] = (Opcode==R_TYPE);
  assign Con_AUIPC = (Opcode==AUIPC);
  assign Con_LUI = (Opcode==LUI);
  assign memcopy = (Opcode==MEMCOPY);
```

ALU – Controller

This particular component receives control signals from the main controller, as well as function_3 and function_7 derived from the instruction. It takes inputs such as the opcode (2 bits from the 7-bit instruction opcode), function7 (bits 25 to 31 of the instruction), and function3 (bits 12 to 14 of the instruction). Additionally, it receives control signals indicating whether the instruction is related to branching, memory, immediate operation (OpI), or AUIPC. Based on these inputs, it assigns a control opcode (4-bit operation) for the ALU.

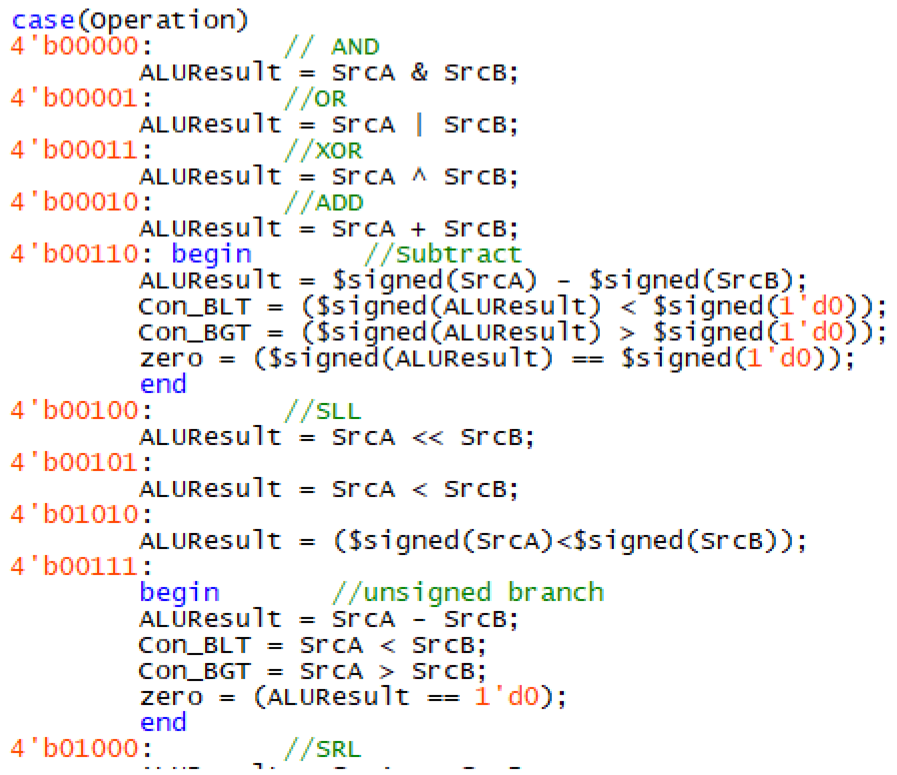


```
assign Con_beq = (Branch)&&(Funct3==3'b000);
assign Con_bnq = (Branch)&&(Funct3==3'b001);
assign Con_blt = (Branch)&&(Funct3==3'b100||Funct3==3'b110);
assign Con_bgt = (Branch)&&(Funct3==3'b101||Funct3==3'b111);

assign Operation[0]= ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b110)||(Funct3==3'b1


assign Operation[1]= (ALUOp==2'b00&&(!OpI)) ||
                     ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b000)||(Funct3==3'b10
                     ((ALUOp[1]==1'b1) && (Funct7==7'b0100000) && (Funct3==3'b000))||(ALUOp[0]==1'b


assign Operation[2]= (ALUOp[0]==1'b1)||((ALUOp[1]==1'b1) && (Funct7==7'b0100000) && (Funct3==3'b00

assign Operation[3]= (Funct3 == 3'b010&&(!Mem))||(OpI&&Funct3 == 3'b101&&Funct7 == 7'b0000000)||(A

// for mul operation(same as addr operation, all control signal are same only the operation change
// function 7 will be 7'b0111000 along with same others as addr
assign Operation[4]= (Funct7 == 7'b0111000);
```

ALU

The ALU (Arithmetic Logic Unit) is responsible for performing calculations. It takes two input data, each consisting of 32 bits (srcA and srcB), as well as a 4-bit opcode (operation). It produces a 32-bit output called Aluresult and several flags, namely con_blt, con_bgt, and zero, each being one bit in size. The ALU's behavior is defined as combinational, meaning it performs calculations immediately without the need for a clock signal.

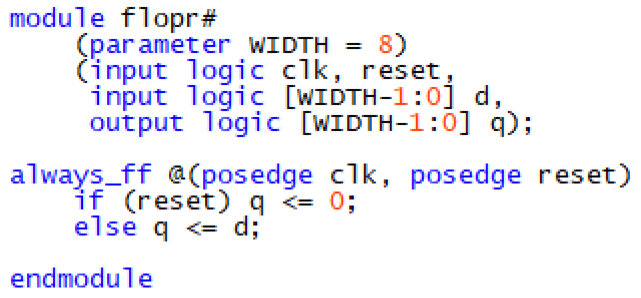
The ALU executes different tasks based on the 4-bit opcode, with a total of 2^4 (16) possible tasks. But to add new instruction (MUL) add additional bit for Alu control signal. So keeping the same architecture we can add new instruction as required. (i.e. for MUL instruction control signal are almost same as for addr instruction)



```
case(Operation)
4'b00000:        // AND
        ALUResult = SrcA & SrcB;
4'b00001:        //OR
        ALUResult = SrcA | SrcB;
4'b00011:        //XOR
        ALUResult = SrcA ^ SrcB;
4'b00010:        //ADD
        ALUResult = SrcA + SrcB;
4'b00110: begin       //Subtract
        ALUResult = $signed(SrcA) - $signed(SrcB);
        Con_BLT = ($signed(ALUResult) < $signed(1'd0));
        Con_BGT = ($signed(ALUResult) > $signed(1'd0));
        zero = ($signed(ALUResult) == $signed(1'd0));
        end
4'b00100:        //SLL
        ALUResult = SrcA << SrcB;
4'b00101:
        ALUResult = SrcA < SrcB;
4'b01010:
        ALUResult = ($signed(SrcA)<$signed(SrcB));
4'b00111:
        begin       //unsigned branch
        ALUResult = SrcA - SrcB;
        Con_BLT = SrcA < SrcB;
        Con_BGT = SrcA > SrcB;
        zero = (ALUResult == 1'd0);
        end
4'b01000:        //SRL
```

Floper

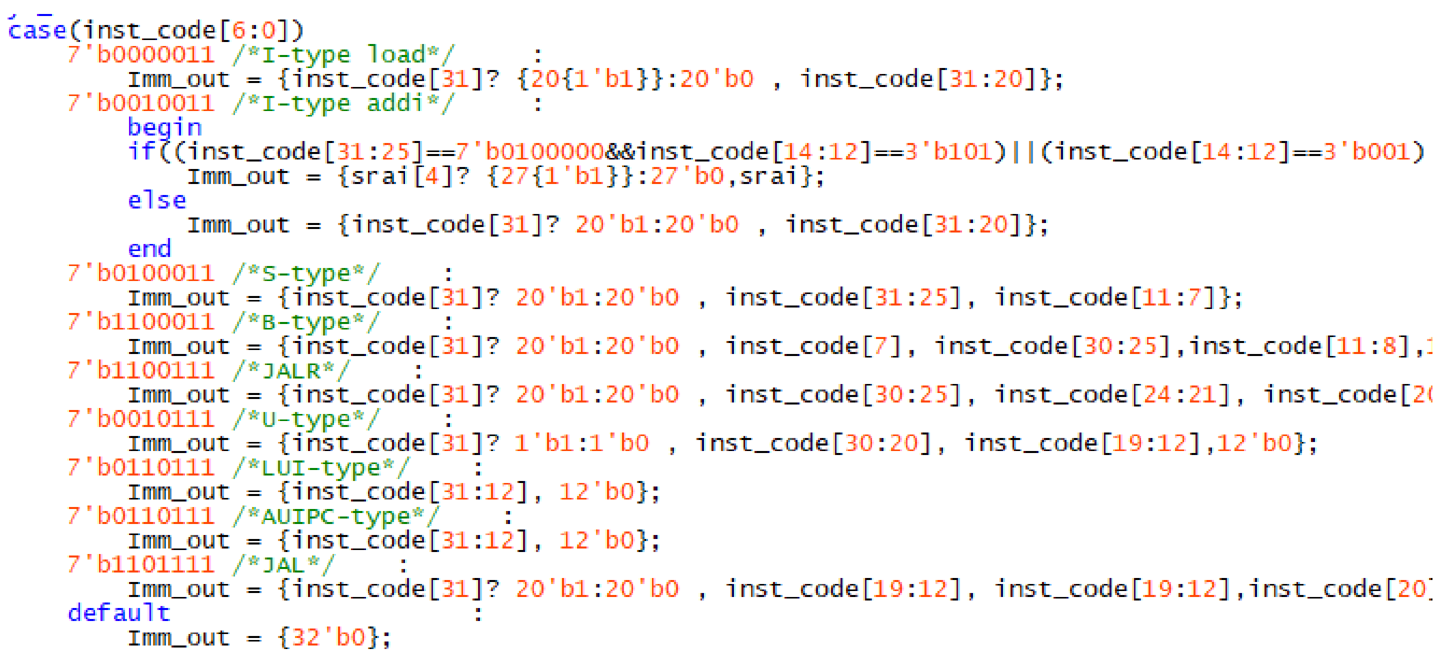
This module serves as a synchronous reset flip-flop, a fundamental component in digital circuit design. It stores data from the "d" input when there's a positive edge on the clock signal (clk), and it resets its state to 0 when the reset signal is activated. This flip-flop used to store and update the program counter.



```
module flopr#
    (parameter WIDTH = 8)
    (input logic clk, reset,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= d;

endmodule
```

IMM Generator

This module is tasked with generating immediate values by extracting data from instructions. It consistently produces a 32-bit value for the immediate data.

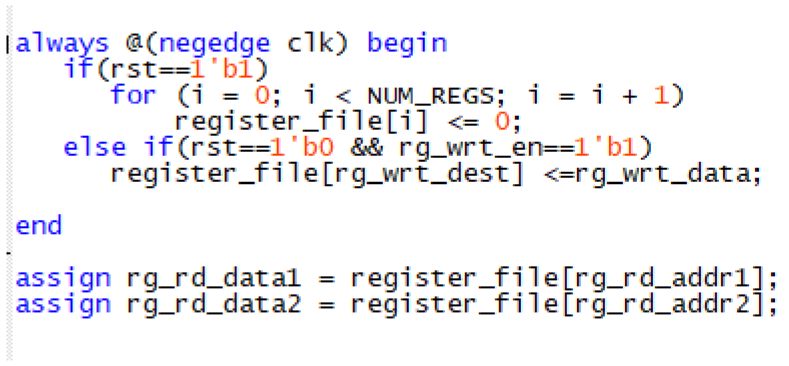


```
case(inst_code[6:0])
    7'b0000011 /*I-type load*/      :
        Imm_out = {inst_code[31]? {20{1'b1}}:20'b0 , inst_code[31:20]};
    7'b0010011 /*I-type addi*/      :
        begin
        if((inst_code[31:25]==7'b0100000&&inst_code[14:12]==3'b101)||(inst_code[14:12]==3'b001)
            Imm_out = {srai[4]? {27{1'b1}}:27'b0,srai};
        else
            Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:20]};
        end
    7'b0100011 /*S-type*/     :
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:25], inst_code[11:7]};
    7'b1100011 /*B-type*/     :
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[7], inst_code[30:25],inst_code[11:8],1
    7'b1100111 /*JALR*/     :
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[30:25], inst_code[24:21], inst_code[2
    7'b0010111 /*U-type*/     :
        Imm_out = {inst_code[31]? 1'b1:1'b0 , inst_code[30:20], inst_code[19:12],12'b0};
    7'b0110111 /*LUI-type*/     :
        Imm_out = {inst_code[31:12], 12'b0};
    7'b0110111 /*AUIPC-type*/     :
        Imm_out = {inst_code[31:12], 12'b0};
    7'b1101111 /*JAL*/     :
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[19:12], inst_code[19:12],inst_code[20
    default                   :
        Imm_out = {32'b0};
```

Registers

In our design, we employ 32-bit long registers with 32 registers in total, necessitating 5 bits for the address inputs. These inputs include clock signals, a reset signal (which resets all registers to 0 when set to 1), a write enable control signal, a 5-bit address to write data, and two 5-bit addresses to read data. The data to be written is 32 bits.

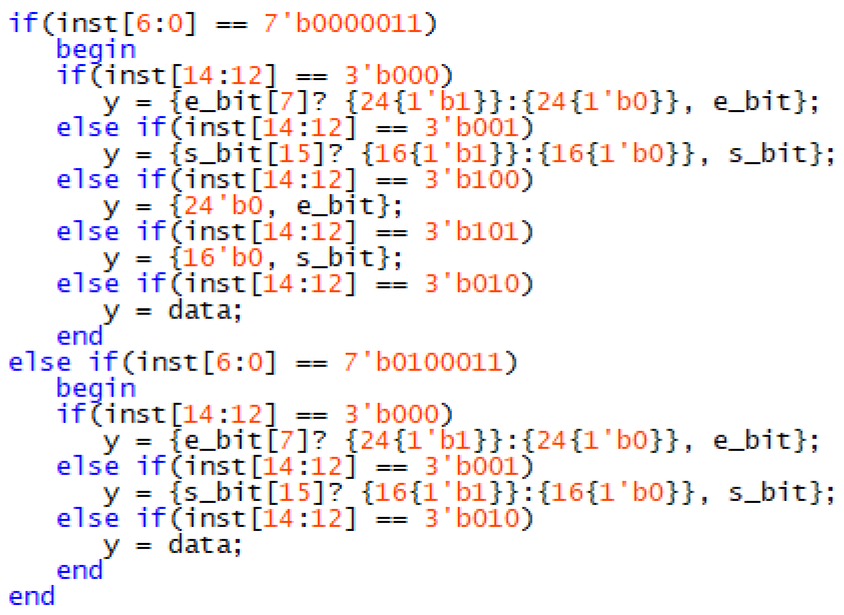
The output comprises two 32-bit data reads. The registers are checked at the negative clock edge. When the reset signal is set to 1, all register values are reset to 0. If the memory read control signal is available at this clock edge, the data is written to the designated register address. However, data read is not synchronized with clock edges. As soon as the data read address is available, the module assigns the data from the specified register to the output register (rd). This module has the capability to read two addresses simultaneously.



```
always @(negedge clk) begin
    if(rst==1'b1)
        for (i = 0; i < NUM_REGS; i = i + 1)
            register_file[i] <= 0;
    else if(rst==1'b0 && rg_wrt_en==1'b1)
        register_file[rg_wrt_dest] <=rg_wrt_data;

end

assign rg_rd_data1 = register_file[rg_rd_addr1];
assign rg_rd_data2 = register_file[rg_rd_addr2];
```

Data Extractor

This module accepts a 32-bit instruction and 32-bit data as inputs. It produces a 32-bit output, denoted as "y," based on the instruction's specifications. To prevent the generation of latches, a default value of 0 is added to "y." The module's primary function is to generate immediate (imm) values for certain instructions and extract the necessary data from the instructions.

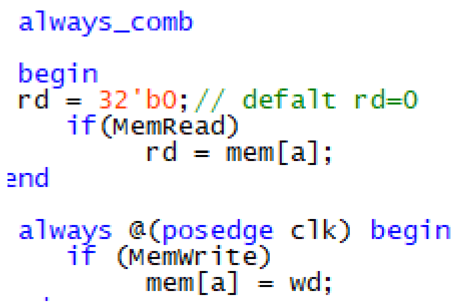


```
if(inst[6:0] == 7'b0000011)
   begin
   if(inst[14:12] == 3'b000)
      y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit};
   else if(inst[14:12] == 3'b001)
      y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit};
   else if(inst[14:12] == 3'b100)
      y = {24'b0, e_bit};
   else if(inst[14:12] == 3'b101)
      y = {16'b0, s_bit};
   else if(inst[14:12] == 3'b010)
      y = data;
   end
else if(inst[6:0] == 7'b0100011)
   begin
   if(inst[14:12] == 3'b000)
      y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit};
   else if(inst[14:12] == 3'b001)
      y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit};
   else if(inst[14:12] == 3'b010)
      y = data;
   end
end
```

Data Memory

This module consists of an array with 2^9 (512) 32-bit memory elements. Initially, it takes a 9-bit address to specify the current memory location and a 32-bit data to write to that specified location. It also provides a 32-bit output data when a memory read operation is requested. Additionally, it relies on various control signals to determine its actions.

To accommodate the new instruction "MEMCOPY," an additional control signal has been introduced. This new control signal is used to execute the MEMCOPY operation and is accompanied by two 9-bit addresses and a 7-bit offset to define a specific memory range for the operation.

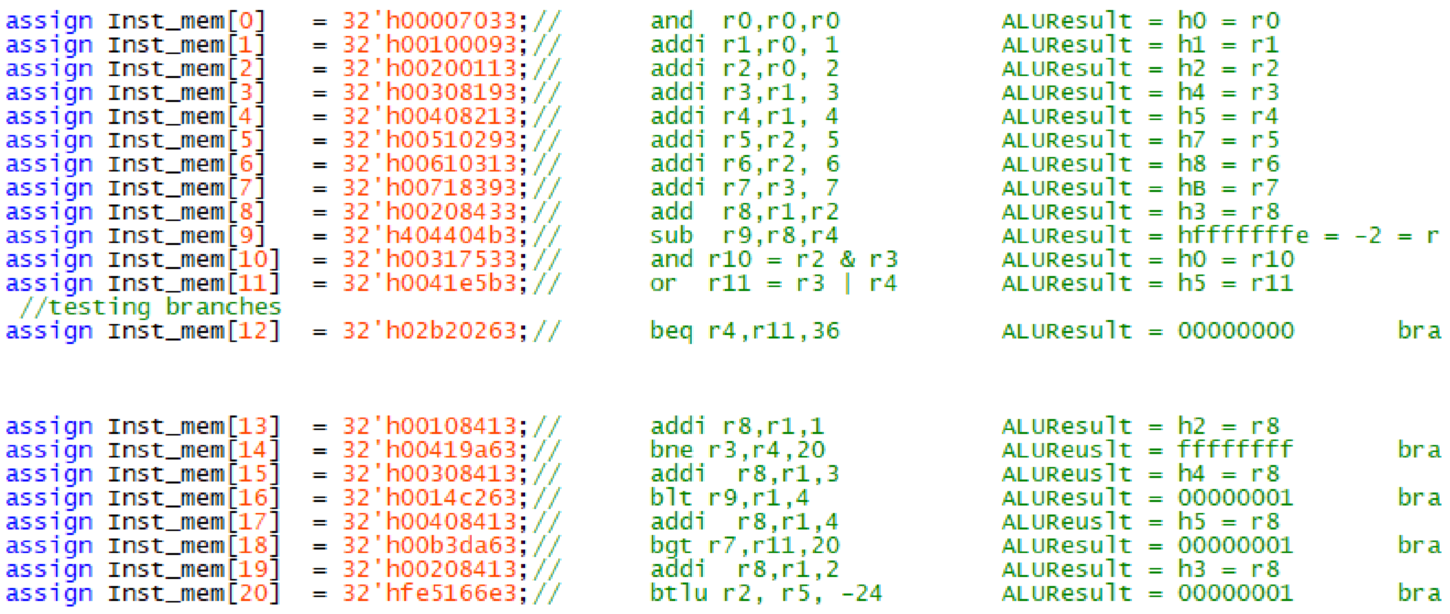


```
always_comb

begin
rd = 32'b0;// defalt rd=0
   if(MemRead)
        rd = mem[a];
end

always @(posedge clk) begin
   if (MemWrite)
        mem[a] = wd;
```

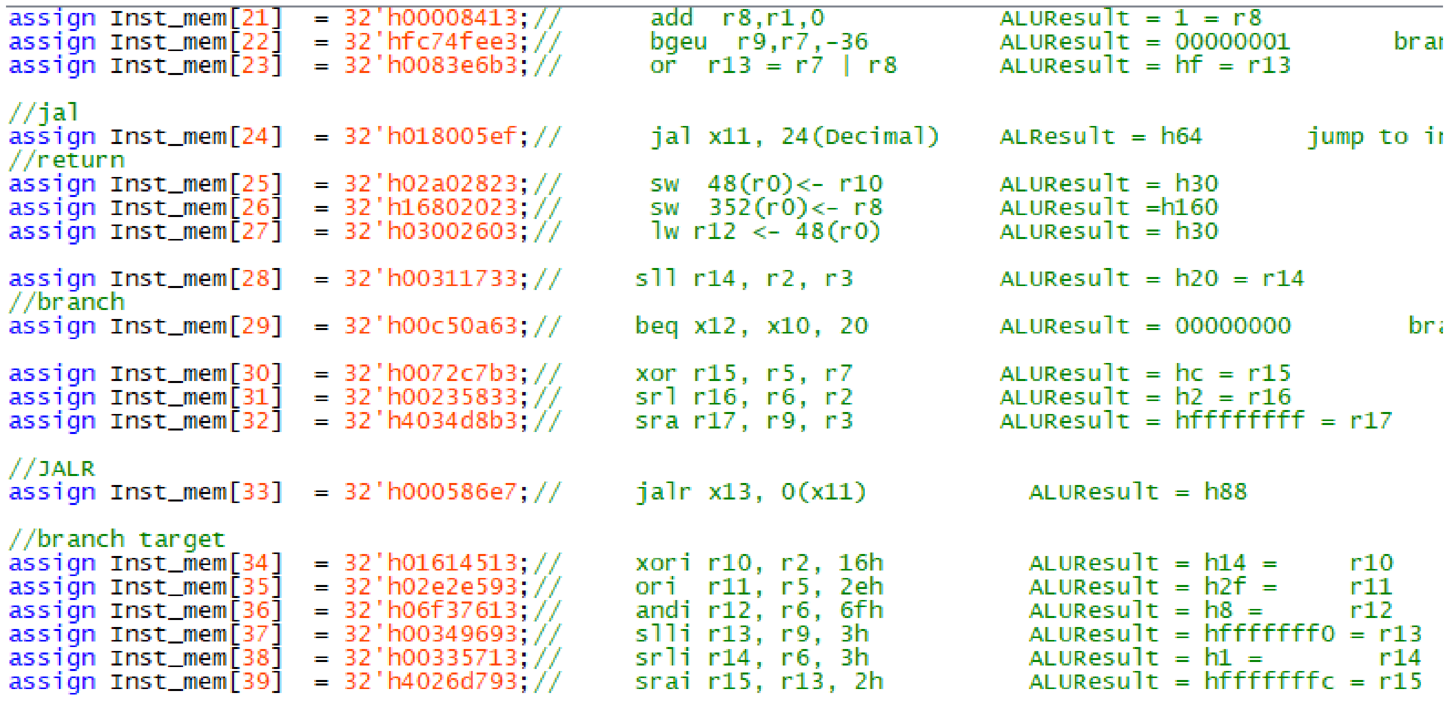
Instruction Memory

This module comprises an array of sub-arrays, with each sub-array being 32 bits long. The width of each instruction is 32 bits, allowing for a total of 2^7 (128) possible instructions, as indicated by the 7-bit address. The input "ra" is a 32-bit data from the program counter (PC). With the PC being 6 bits long (enabling 64 possibilities), there are 56 distinct values for it. Also additional instruction has added for MUL and MEMCOPY.

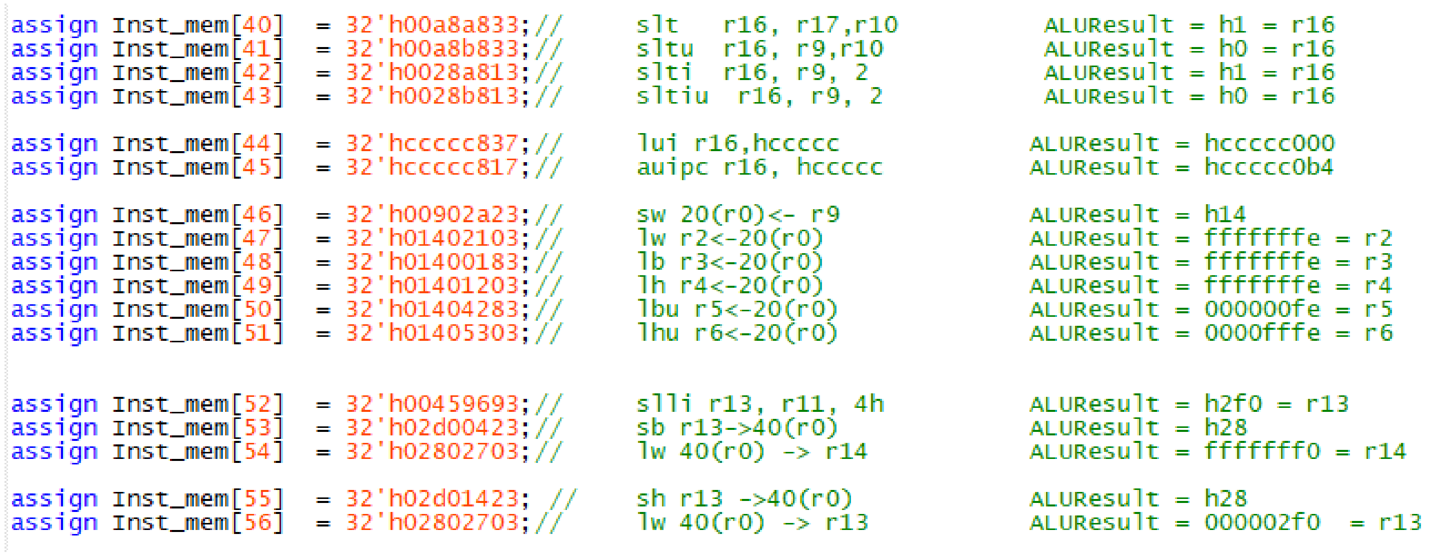
This module reads the 32-bit value from the PC and assigns the corresponding instruction to the designated destination based on the provided address.



```
assign Inst_mem[0]   = 32'h00007033;//      and  r0,r0,r0          ALUResult = h0 = r0
assign Inst_mem[1]   = 32'h00100093;//      addi r1,r0, 1          ALUResult = h1 = r1
assign Inst_mem[2]   = 32'h00200113;//      addi r2,r0, 2          ALUResult = h2 = r2
assign Inst_mem[3]   = 32'h00308193;//      addi r3,r1, 3          ALUResult = h4 = r3
assign Inst_mem[4]   = 32'h00408213;//      addi r4,r1, 4          ALUResult = h5 = r4
assign Inst_mem[5]   = 32'h00510293;//      addi r5,r2, 5          ALUResult = h7 = r5
assign Inst_mem[6]   = 32'h00610313;//      addi r6,r2, 6          ALUResult = h8 = r6
assign Inst_mem[7]   = 32'h00718393;//      addi r7,r3, 7          ALUResult = hB = r7
assign Inst_mem[8]   = 32'h00208433;//      add  r8,r1,r2          ALUResult = h3 = r8
assign Inst_mem[9]   = 32'h404404b3;//      sub  r9,r8,r4          ALUResult = hfffffffe = -2 = r
assign Inst_mem[10]  = 32'h00317533;//      and r10 = r2 & r3      ALUResult = h0 = r10
assign Inst_mem[11]  = 32'h0041e5b3;//      or   r11 = r3 | r4     ALUResult = h5 = r11
 //testing branches
assign Inst_mem[12]  = 32'h02b20263;//      beq r4,r11,36          ALUResult = 00000000       bra


assign Inst_mem[13]  = 32'h00108413;//      addi r8,r1,1           ALUResult = h2 = r8
assign Inst_mem[14]  = 32'h00419a63;//      bne r3,r4,20           ALUReuslt = ffffffff       bra
assign Inst_mem[15]  = 32'h00308413;//      addi  r8,r1,3          ALUReuslt = h4 = r8
assign Inst_mem[16]  = 32'h0014c263;//      blt r9,r1,4            ALUResult = 00000001       bra
assign Inst_mem[17]  = 32'h00408413;//      addi  r8,r1,4          ALUReuslt = h5 = r8
assign Inst_mem[18]  = 32'h00b3da63;//      bgt r7,r11,20          ALUResult = 00000001       bra
assign Inst_mem[19]  = 32'h00208413;//      addi  r8,r1,2          ALUResult = h3 = r8
assign Inst_mem[20]  = 32'hfe5166e3;//      btlu r2, r5, -24       ALUResult = 00000001       bra
```



```
assign Inst_mem[21]  = 32'h00008413;//      add  r8,r1,0           ALUResult = 1 = r8
assign Inst_mem[22]  = 32'hfc74fee3;//      bgeu  r9,r7,-36        ALUResult = 00000001       brai
assign Inst_mem[23]  = 32'h0083e6b3;//      or   r13 = r7 | r8     ALUResult = hf = r13

//jal
assign Inst_mem[24]  = 32'h018005ef;//      jal x11, 24(Decimal)   ALResult = h64       jump to i
//return
assign Inst_mem[25]  = 32'h02a02823;//      sw  48(r0)<- r10       ALUResult = h30
assign Inst_mem[26]  = 32'h16802023;//      sw  352(r0)<- r8       ALUResult =h160
assign Inst_mem[27]  = 32'h03002603;//      lw r12 <- 48(r0)       ALUResult = h30

assign Inst_mem[28]  = 32'h00311733;//     sll r14, r2, r3         ALUResult = h20 = r14
//branch
assign Inst_mem[29]  = 32'h00c50a63;//     beq x12, x10, 20        ALUResult = 00000000        bra

assign Inst_mem[30]  = 32'h0072c7b3;//     xor r15, r5, r7         ALUResult = hc = r15
assign Inst_mem[31]  = 32'h00235833;//     srl r16, r6, r2         ALUResult = h2 = r16
assign Inst_mem[32]  = 32'h4034d8b3;//     sra r17, r9, r3         ALUResult = hffffffff = r17

//JALR
assign Inst_mem[33]  = 32'h000586e7;//     jalr x13, 0(x11)          ALUResult = h88

//branch target
assign Inst_mem[34]  = 32'h01614513;//     xori r10, r2, 16h         ALUResult = h14 =     r10
assign Inst_mem[35]  = 32'h02e2e593;//     ori  r11, r5, 2eh         ALUResult = h2f =     r11
assign Inst_mem[36]  = 32'h06f37613;//     andi r12, r6, 6fh         ALUResult = h8 =      r12
assign Inst_mem[37]  = 32'h00349693;//     slli r13, r9, 3h          ALUResult = hfffffff0 = r13
assign Inst_mem[38]  = 32'h00335713;//     srli r14, r6, 3h          ALUResult = h1 =       r14
assign Inst_mem[39]  = 32'h4026d793;//     srai r15, r13, 2h         ALUResult = hfffffffc = r15
```



```
assign Inst_mem[40]  = 32'h00a8a833;//     slt   r16, r17,r10          ALUResult = h1 = r16
assign Inst_mem[41]  = 32'h00a8b833;//     sltu  r16, r9,r10           ALUResult = h0 = r16
assign Inst_mem[42]  = 32'h0028a813;//     slti  r16, r9, 2            ALUResult = h1 = r16
assign Inst_mem[43]  = 32'h0028b813;//     sltiu  r16, r9, 2           ALUResult = h0 = r16

assign Inst_mem[44]  = 32'hccccc837;//     lui r16,hccccc             ALUResult = hccccc000
assign Inst_mem[45]  = 32'hccccc817;//     auipc r16, hccccc          ALUResult = hccccc0b4

assign Inst_mem[46]  = 32'h00902a23;//     sw 20(r0)<- r9             ALUResult = h14
assign Inst_mem[47]  = 32'h01402103;//     lw r2<-20(r0)              ALUResult = fffffffe = r2
assign Inst_mem[48]  = 32'h01400183;//     lb r3<-20(r0)              ALUResult = fffffffe = r3
assign Inst_mem[49]  = 32'h01401203;//     lh r4<-20(r0)              ALUResult = fffffffe = r4
assign Inst_mem[50]  = 32'h01404283;//     lbu r5<-20(r0)             ALUResult = 000000fe = r5
assign Inst_mem[51]  = 32'h01405303;//     lhu r6<-20(r0)             ALUResult = 0000fffe = r6


assign Inst_mem[52]  = 32'h00459693;//     slli r13, r11, 4h          ALUResult = h2f0 = r13
assign Inst_mem[53]  = 32'h02d00423;//     sb r13->40(r0)             ALUResult = h28
assign Inst_mem[54]  = 32'h02802703;//     lw 40(r0) -> r14           ALUResult = fffffff0 = r14

assign Inst_mem[55]  = 32'h02d01423;  //   sh r13 ->40(r0)            ALUResult = h28
assign Inst_mem[56]  = 32'h02802703;//     lw 40(r0) -> r13           ALUResult = 000002f0  = r13
```

Adders

- 32-Bit Adder (Word Adder)
  This module takes two 32-bit inputs, 'a' and 'b,' and produces a 32-bit output, which is the result of adding 'a' and 'b.' It's important to note that this adder operates without any clock signals.

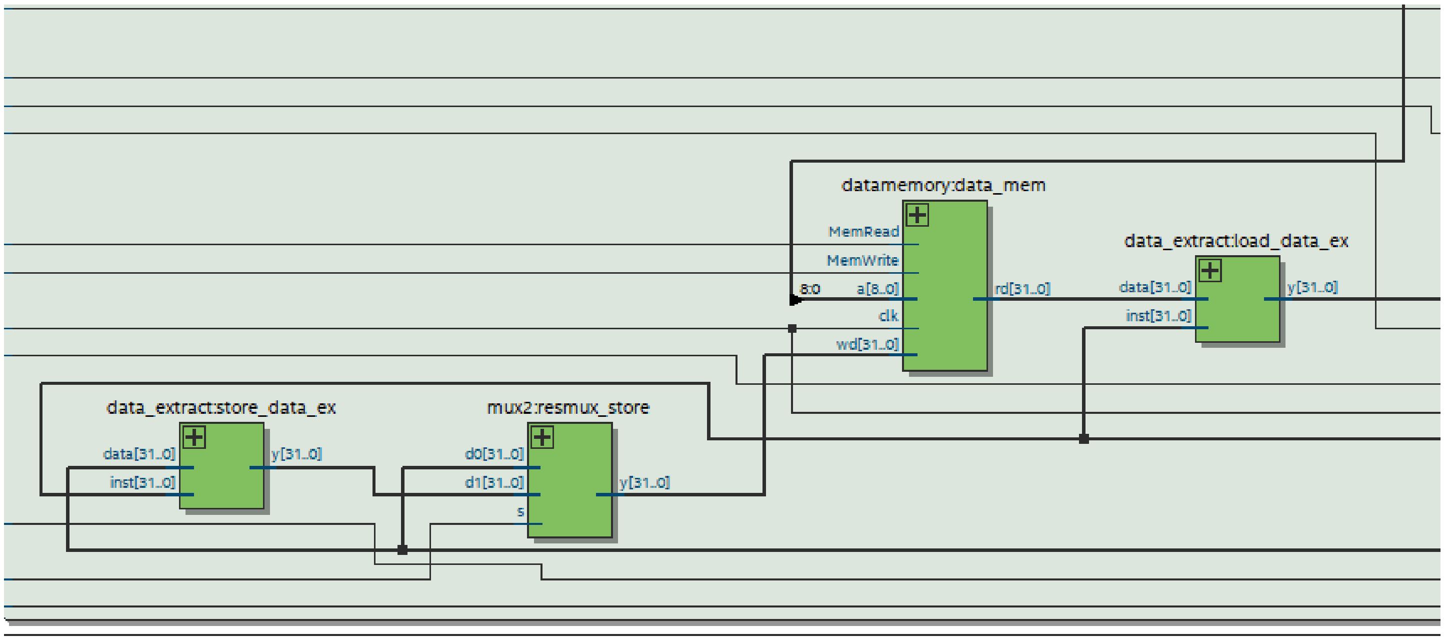
- 8-Bit Adder (Byte Adder)
  Similar to the 32-bit adder, this module takes two 8-bit inputs and produces an 8-bit output, which represents the sum of the inputs. Like the 32-bit adder, the 8-bit adder operates without clock signals.

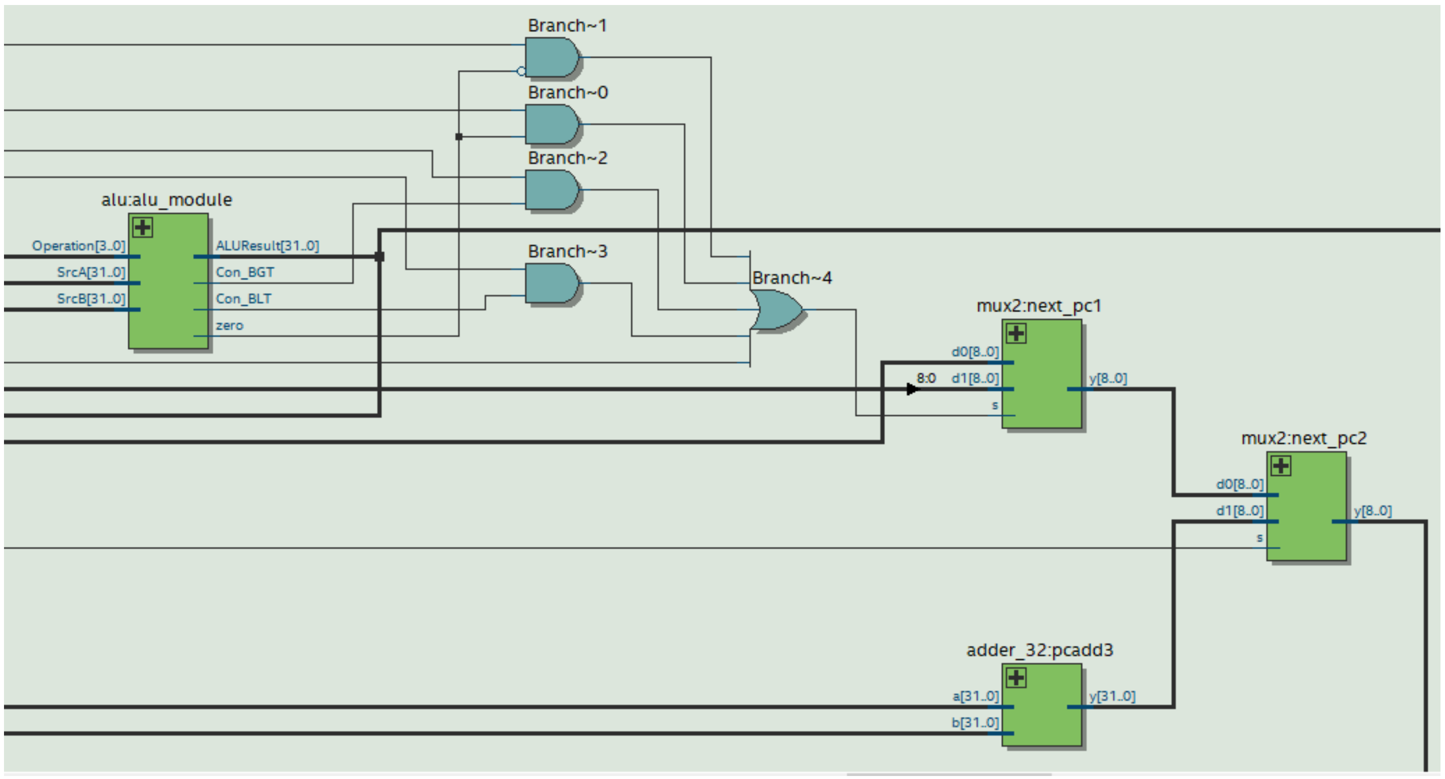
Mux

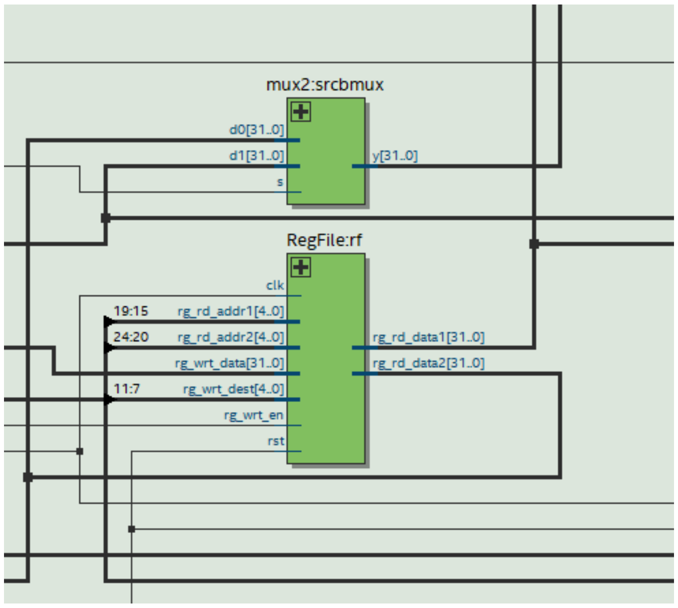
- 3-to-1 Multiplexer (MUX3)
  This multiplexer has three inputs, each with 9 bits (d0, d1, d2), and it uses a 2-bit selection signal (s[0] and s[1]). The output 'y' (9 bits) is determined based on the combinations of s[0] and s[1]. If (s[1] is true and s[0] is false), 'y' is set to d1. If (s[1] is false and s[0] is true), 'y' is set to d2. Otherwise, 'y' is set to d0.

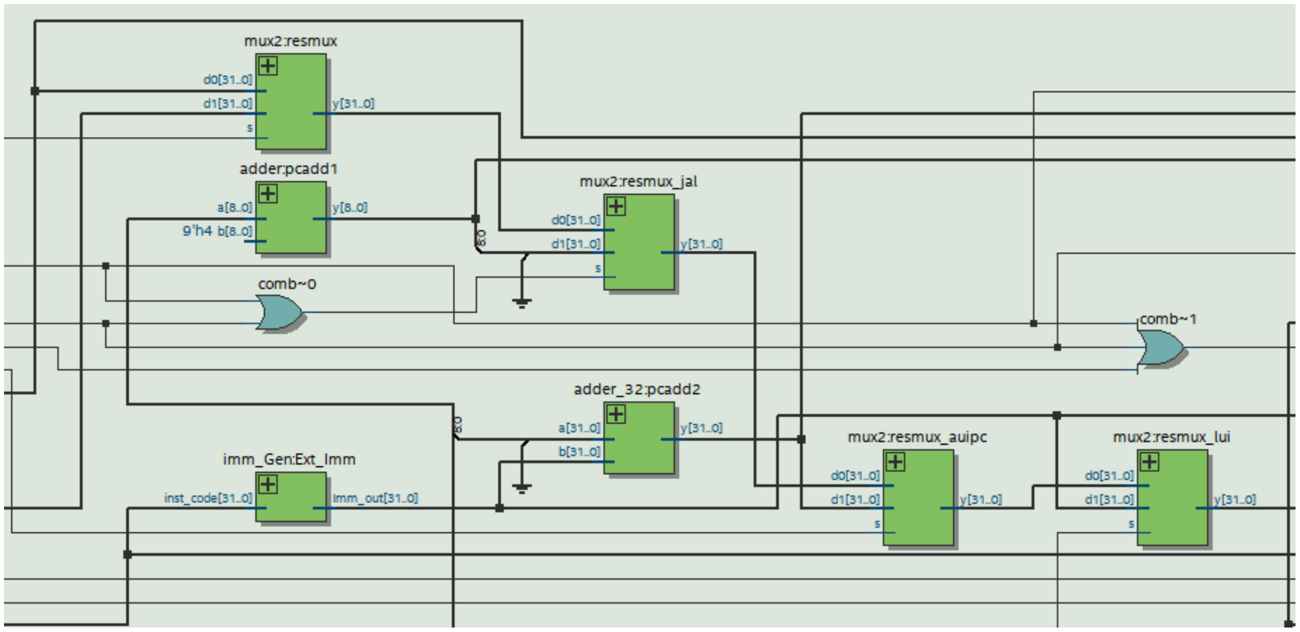
- 2-to-1 Multiplexer (MUX2)
  This multiplexer has two inputs, each with 9 bits (d0, d1), and it uses a 1-bit selection signal (s). The output 'y' (9 bits) is determined by the value of 's.' If 's' is true, 'y' is set to d1; otherwise, 'y' is set to d0.

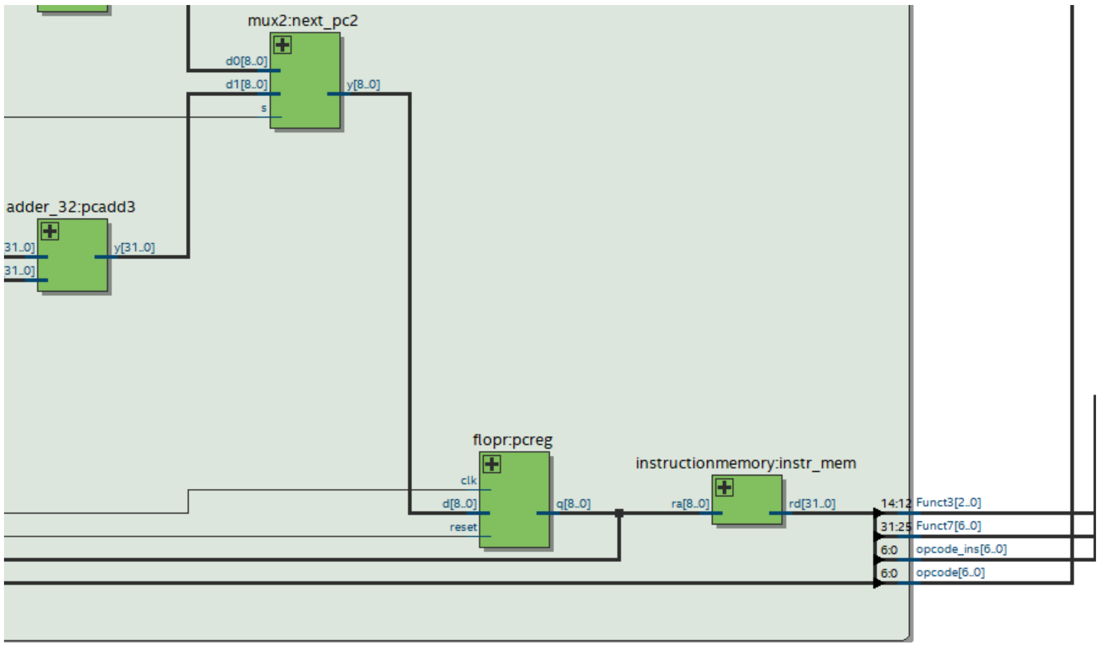
Implemented Modules in Datapath






mux2:srcbmux
d0[31..0]
d1[31..0]
s
y[31..0]
RegFile:rf
clk
19:15 rg_rd_addr1[4..0]
24:20 rg_rd_addr2[4..0]
rg_wrt_data[31..0]
11:7 rg_wrt_dest[4..0]
rg_wrt_en
rst
rg_rd_data1[31..0]
rg_rd_data2[31..0]


mux2:resmux
d0[31..0]
d1[31..0]
s
y[31..0]
adder:pcadd1
a[8..0]
9'h4 b[8..0]
y[8..0]
mux2:resmux_jal
d0[31..0]
d1[31..0]
s
y[31..0]
comb~0
comb~1
adder_32:pcadd2
a[31..0]
b[31..0]
y[31..0]
imm_Gen:Ext_Imm
inst_code[31..0]
Imm_out[31..0]
mux2:resmux_auipc
d0[31..0]
d1[31..0]
s
y[31..0]
mux2:resmux_lui
d0[31..0]
d1[31..0]
s
y[31..0]


mux2:next_pc2
d0[8..0]
d1[8..0]
s
y[8..0]
adder_32:pcadd3
31..0]
31..0]
y[31..0]
flopr:pcreg
clk
d[8..0]
reset
q[8..0]
instructionmemory:instr_mem
ra[8..0]
rd[31..0]
14:12 Funct3[2..0]
31:25 Funct7[6..0]
6:0 opcode_ins[6..0]
6:0 opcode[6..0]

Microprogramming

Based on the opcode controller identify the instruction type and will generate the required control signals
Here are the descriptions of the various control signals,

- **ALUSrc:** This signal allows for the selection between immediate values and the rd2 register, which is essential for data processing.
- **MemtoReg**: It governs the transfer of data from memory to the registers, ensuring accurate data handling.
- **RegWrite**: This signal authorizes the writing of data to the registers, a fundamental operation in the execution of instructions.
- **MemRead**: It initiates the retrieval of data from the data memory, which is crucial for memory access instructions.
- **MemWrite**: This signal triggers the storing of data into the data memory, which is essential for saving values in memory.
- **Branch**: It involves a comparison in the ALU and adds the immediate value to the Program Counter (PC) value, controlling conditional branches in the instruction flow.
- **Jump**: This signal enables the addition of an offset to the PC value, facilitating unconditional jumps in the instruction flow.
- **ALUOp**: It determines the appropriate ALU operation based on the type of instruction, ensuring that the correct arithmetic and logic calculations are performed.
- **Memcopy** : decide the memory copy operation based on the opcode
- **Mul** : decide the multiplication operation based on the function_7 value in the instruction.

Instructions

In this project, there are four main types of instructions that have been implemented, which belong to the basic instruction set of the RISC-V architecture. Each instruction will be 32 bits because this processor is based on a 32-bit architecture. There are predefined blocks of instructions that can be used to define different functionality. The RISC-V architecture is little-endian. So the final seven bits will be the opcode for every instruction. Based on these 7 bits, there are mainly four types of instructions: R-type, S-type, I-type, and SB-type. Additionally, there are function3 (a 3-bit pattern) and function7 (a 7-bit pattern) that can be used to define different instructions within one type of instruction.

Two additional instructions, MEMCOPY and MUL, have been introduced in addition to the four main types. The MEMCOPY instruction defines a new instruction type that differs from the basic four types. The MUL instruction is similar to the R-type as it multiplies two register values and stores the result in a given destination. The only difference between this instruction and the R-type add instruction lies in the function7 bits.

R-type

R-type instructions are employed for arithmetic and logic operations between two registers. These instructions execute operations such as addition, subtraction, AND, OR, XOR, and shifting. In these instructions, the source registers (rs1 and rs2) provide input, and the destination register (rd) stores the result of the operation. For example, the ADD instruction adds the values of rs1 and rs2 and stores the result in rd. based on the opcode ALUSrc , Regwrite control signals and Alu controller output ‘Operation’ will change as required.

I-type

These instructions contain immediate data. Depending on the type of the instruction, immediate values and sizes vary. However, by utilizing the "immgenerator," you can extract the immediate value and generate a suitable output value. These instructions are similar to the R-type in that they take values from registers and also incorporate immediate values from instructions to perform specific tasks.based on the opcode ALUSrc , Regwrite control signals and Alu controller output ‘Operation’ will change as required.

For example, the ADDI instruction adds an immediate value to the content of a register and stores the result back in the destination.

S-type

These are primarily load and store instructions, which means they handle the loading of data from memory and the storing of data from registers in memory. Depending on the specific operation, the control signals such as ALUSrc, MemtoReg, RegtoMem, RegWrite, MemRead, Mem, and MemWrite will change as needed.

SB-type

SB-type instructions are responsible for managing control flow operations, specifically conditional branching. These instructions involve comparing two registers and initiating a branch when a particular condition is met. If the result of the comparison aligns with the specified condition, it leads to an update in the program counter, thereby altering the flow of code execution. The specific branch instruction in use dictates the adjustment of control signals such as Branch, Con_Jalr, and Con_Jal.

Implementation of new instructions (MEMCOPY and MUL)

To enhance the instruction set, two new instructions have been introduced. Firstly, the MEMCOPY instruction allows for the efficient copying of arrays, specifically those with a size greater than one, from one memory location to another.

Secondly, the MUL instruction has been added to enable unsigned multiplication, an operation not originally included in the RV32I instruction set. These additions expand the capabilities of the processor and provide support for a wider range of operations.

MEMCOPY

For the memory copy operation, a special type of instruction with the opcode "0001000" has been designed. Given that the memory comprises 2^9 memory addresses, this instruction includes two blocks for specifying the starting address of the copying process and the destination address where the content is to be copied. The remaining 7 bits are utilized to specify the offset, representing the size of the object being copied from one location to another.

Instruction format

Offset	Copying address	Starting address	Opcode
7 bits	9 bits	9 bits	7 bits

The controller generates the 'memcopy' control signal based on the opcode, and with this control signal and the two specified addresses, the data memory executes the memory copy operation. To perform memory copy operations for sizes greater than 32 bits, multiple clock cycles are required, making this instruction a multicycle instruction.

In the instruction, 25 bits are required to specify the type of instruction and the necessary address. This means that only the remaining 7 bits can be used to specify the offset. As a result**, the maximum size of the array** that can be addressed using this instruction is **2^7 bits**.

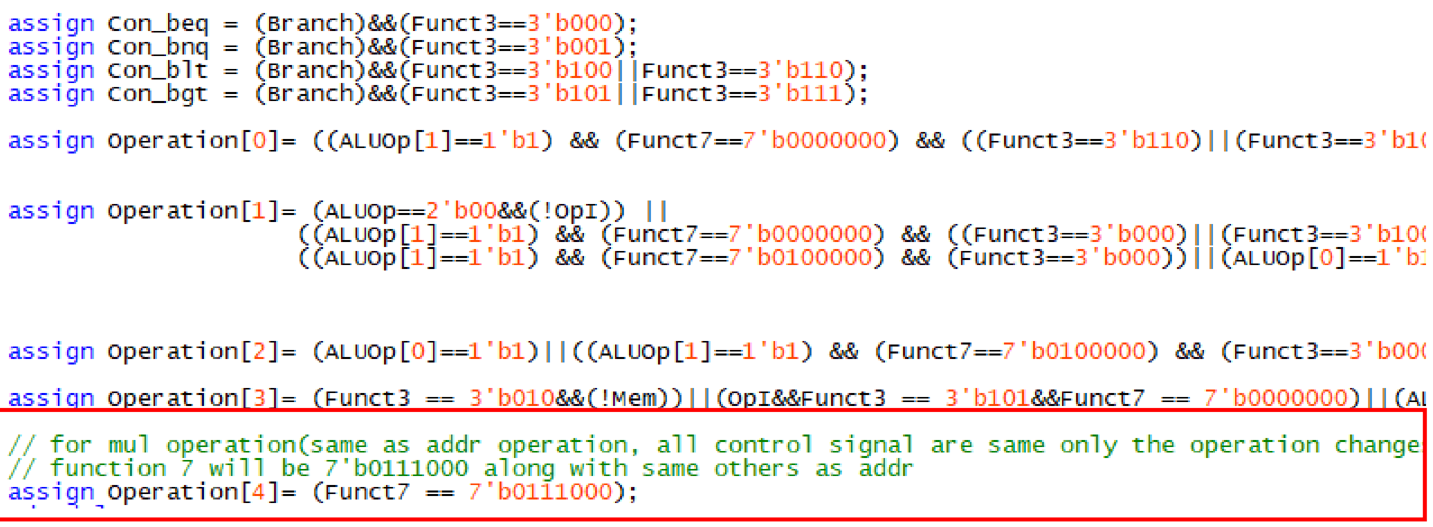
MUL

In this operation, the task involves multiplying two given register values and storing the results in the designated destination register. To carry out this operation, a similar instruction to the R-type (add) is used, which specifies the source registers and the destination register. The key distinction lies in the function7 field: for the add instruction, the function7 value is '0000000,' while in this particular instruction, it is '0111000.'

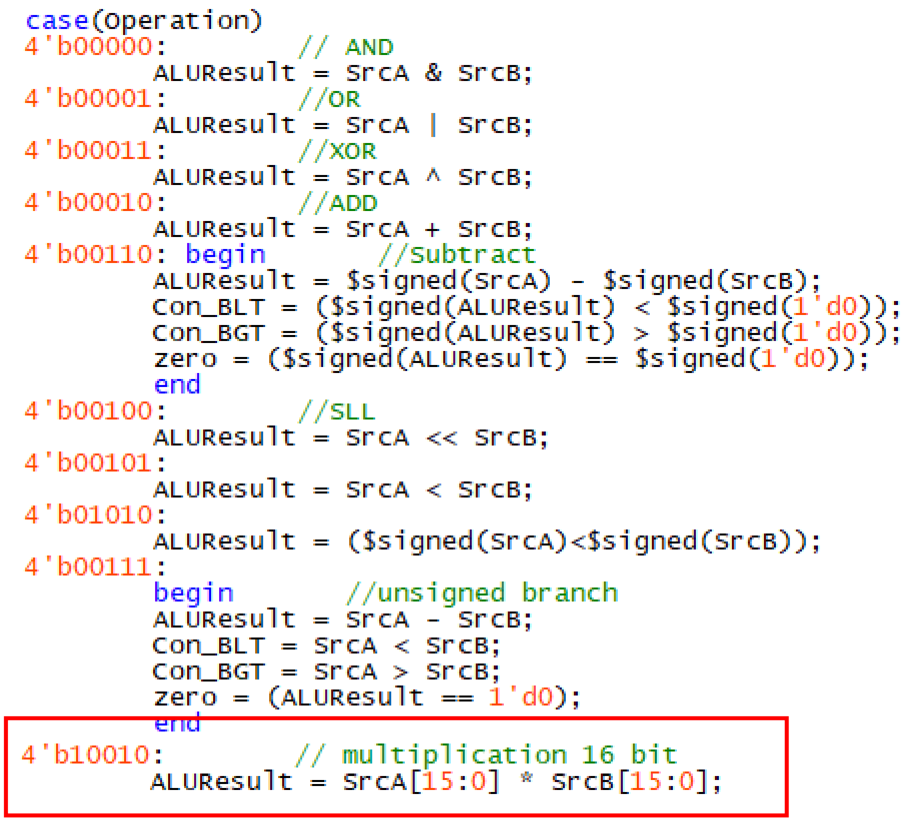
Instruction format

Function7	Rs2	Rs1	Function3	Rd	Opcode
0111000	xxxxx	xxxxx	000	xxxxx	0110011

All the control signals remain the same in the main controller as they are for the 'add' instruction. However, due to the different function7 value, the ALU controller generates a distinct 'operation' control signal compared to the 'add' instruction. Consequently, the ALU will perform multiplication using the two specified registers instead of addition and store the result in the designated register.



```
assign Con_beq = (Branch)&&(Funct3==3'b000);
assign Con_bnq = (Branch)&&(Funct3==3'b001);
assign Con_blt = (Branch)&&(Funct3==3'b100||Funct3==3'b110);
assign Con_bgt = (Branch)&&(Funct3==3'b101||Funct3==3'b111);

assign Operation[0]= ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b110)||(Funct3==3'b1


assign Operation[1]= (ALUOp==2'b00&&(!OpI)) ||
                     ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b000)||(Funct3==3'b10
                     ((ALUOp[1]==1'b1) && (Funct7==7'b0100000) && (Funct3==3'b000))||(ALUOp[0]==1'b


assign Operation[2]= (ALUOp[0]==1'b1)||((ALUOp[1]==1'b1) && (Funct7==7'b0100000) && (Funct3==3'b00

assign Operation[3]= (Funct3 == 3'b010&&(!Mem))||(OpI&&Funct3 == 3'b101&&Funct7 == 7'b0000000)||(Al

// for mul operation(same as addr operation, all control signal are same only the operation change
// function 7 will be 7'b0111000 along with same others as addr
assign Operation[4]= (Funct7 == 7'b0111000);
```

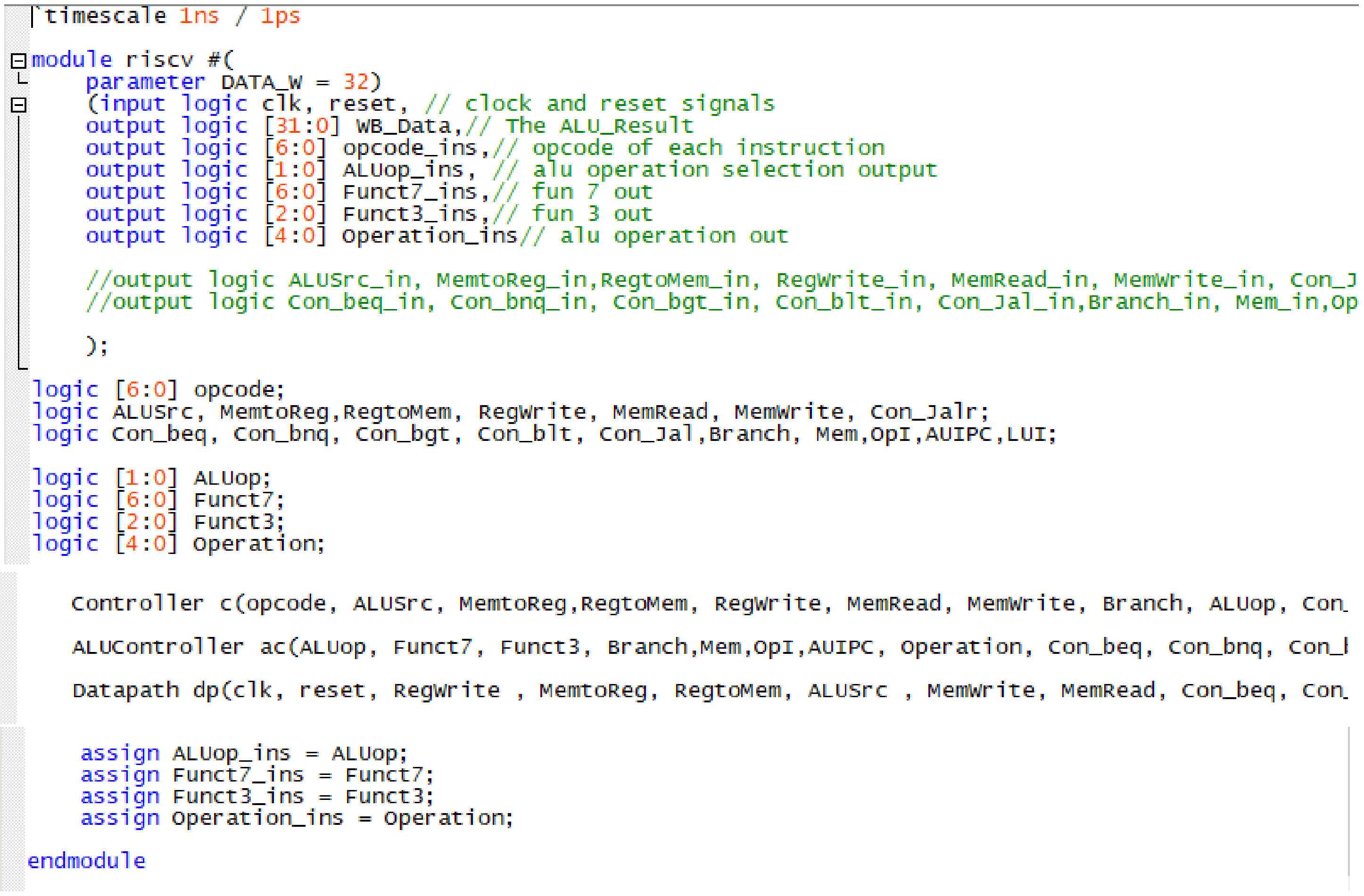


```
case(Operation)
4'b00000:          // AND
        ALUResult = SrcA & SrcB;
4'b00001:          //OR
        ALUResult = SrcA | SrcB;
4'b00011:          //XOR
        ALUResult = SrcA ^ SrcB;
4'b00010:          //ADD
        ALUResult = SrcA + SrcB;
4'b00110: begin         //Subtract
        ALUResult = $signed(SrcA) - $signed(SrcB);
        Con_BLT = ($signed(ALUResult) < $signed(1'd0));
        Con_BGT = ($signed(ALUResult) > $signed(1'd0));
        zero = ($signed(ALUResult) == $signed(1'd0));
        end
4'b00100:          //SLL
        ALUResult = SrcA << SrcB;
4'b00101:
        ALUResult = SrcA < SrcB;
4'b01010:
        ALUResult = ($signed(SrcA)<$signed(SrcB));
4'b00111:
        begin       //unsigned branch
        ALUResult = SrcA - SrcB;
        Con_BLT = SrcA < SrcB;
        Con_BGT = SrcA > SrcB;
        zero = (ALUResult == 1'd0);
        end
4'b10010:          // multiplication 16 bit
        ALUResult = SrcA[15:0] * SrcB[15:0];
```

Constraints in MUL instruction.

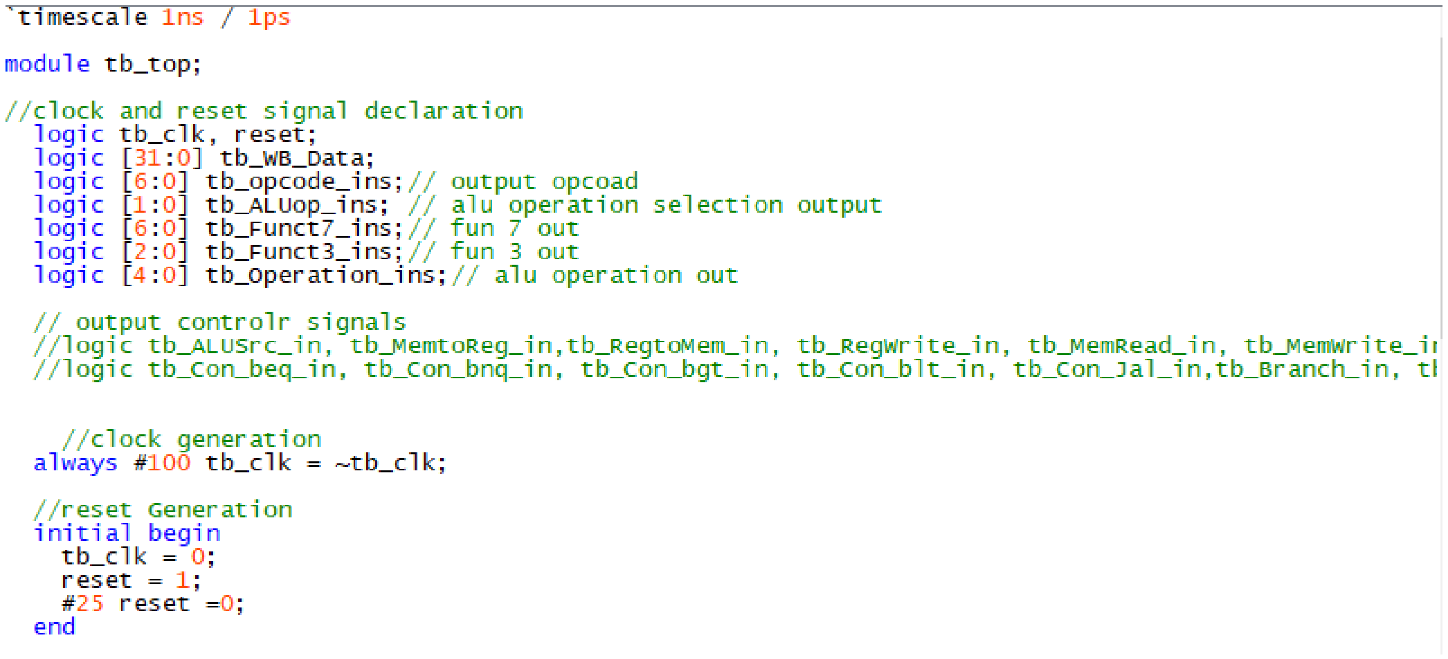
Given that this is a 32-bit architecture design with all registers being 32 bits, multiplying two 32-bit numbers may result in a 64-bit value, which could potentially lead to overflow. Therefore, in this design, multiplication is restricted to 16-bit numbers to prevent overflow issues.

Processor implementation



```
`timescale 1ns / 1ps

module riscv #(
    parameter DATA_W = 32)
    (input logic clk, reset, // clock and reset signals
    output logic [31:0] WB_Data,// The ALU_Result
    output logic [6:0] opcode_ins,// opcode of each instruction
    output logic [1:0] ALUop_ins, // alu operation selection output
    output logic [6:0] Funct7_ins,// fun 7 out
    output logic [2:0] Funct3_ins,// fun 3 out
    output logic [4:0] Operation_ins// alu operation out

    //output logic ALUSrc_in, MemtoReg_in,RegtoMem_in, RegWrite_in, MemRead_in, MemWrite_in, Con_J
    //output logic Con_beq_in, Con_bnq_in, Con_bgt_in, Con_blt_in, Con_Jal_in,Branch_in, Mem_in,Op

    );
logic [6:0] opcode;
logic ALUSrc, MemtoReg,RegtoMem, RegWrite, MemRead, MemWrite, Con_Jalr;
logic Con_beq, Con_bnq, Con_bgt, Con_blt, Con_Jal,Branch, Mem,OpI,AUIPC,LUI;

logic [1:0] ALUop;
logic [6:0] Funct7;
logic [2:0] Funct3;
logic [4:0] Operation;

    Controller c(opcode, ALUSrc, MemtoReg,RegtoMem, RegWrite, MemRead, MemWrite, Branch, ALUop, Con

    ALUController ac(ALUop, Funct7, Funct3, Branch,Mem,OpI,AUIPC, Operation, Con_beq, Con_bnq, Con_

    Datapath dp(clk, reset, RegWrite , MemtoReg, RegtoMem, ALUSrc , MemWrite, MemRead, Con_beq, Con

     assign ALUop_ins = ALUop;
     assign Funct7_ins = Funct7;
     assign Funct3_ins = Funct3;
     assign Operation_ins = Operation;

endmodule
```

Testbench for Testing

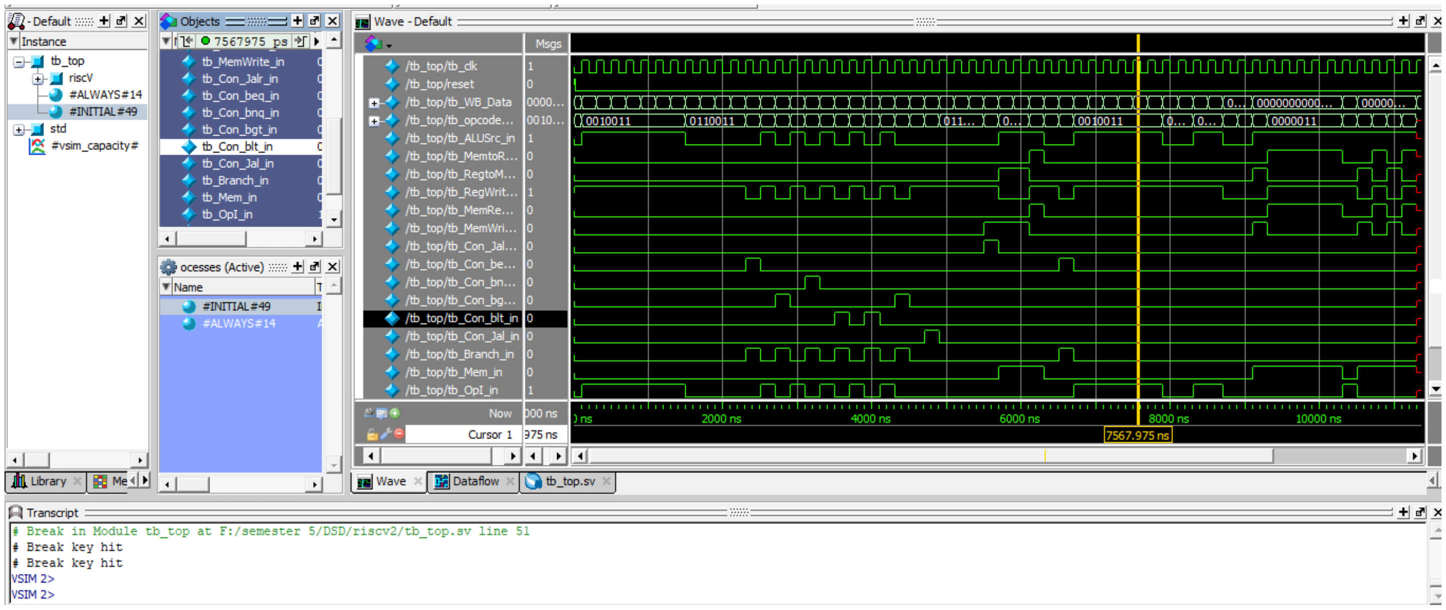


```
`timescale 1ns / 1ps

module tb_top;

//clock and reset signal declaration
  logic tb_clk, reset;
  logic [31:0] tb_WB_Data;
  logic [6:0] tb_opcode_ins;// output opcoad
  logic [1:0] tb_ALUop_ins; // alu operation selection output
  logic [6:0] tb_Funct7_ins;// fun 7 out
  logic [2:0] tb_Funct3_ins;// fun 3 out
  logic [4:0] tb_Operation_ins;// alu operation out

  // output controlr signals
  //logic tb_ALUSrc_in, tb_MemtoReg_in,tb_RegtoMem_in, tb_RegWrite_in, tb_MemRead_in, tb_MemWrite_i
  //logic tb_Con_beq_in, tb_Con_bnq_in, tb_Con_bgt_in, tb_Con_blt_in, tb_Con_Jal_in,tb_Branch_in, t


    //clock generation
  always #100 tb_clk = ~tb_clk;

  //reset Generation
  initial begin
    tb_clk = 0;
    reset = 1;
    #25 reset =0;
  end
```

```
riscv riscV(
    .clk(tb_clk),
    .reset(reset),
    .WB_Data(tb_WB_Data),
    .opcode_ins(tb_opcode_ins),// opcoade output
    .ALUop_ins(tb_ALUop_ins),// aluop output
    .Funct7_ins(tb_Funct7_ins),// fun7 output
    .Funct3_ins(tb_Funct3_ins),// fun3 output
    .Operation_ins(tb_Operation_ins)// operation output

  initial begin
    #13000;
    $finish;
   end
endmodule
```

RTL Simulations and Testing

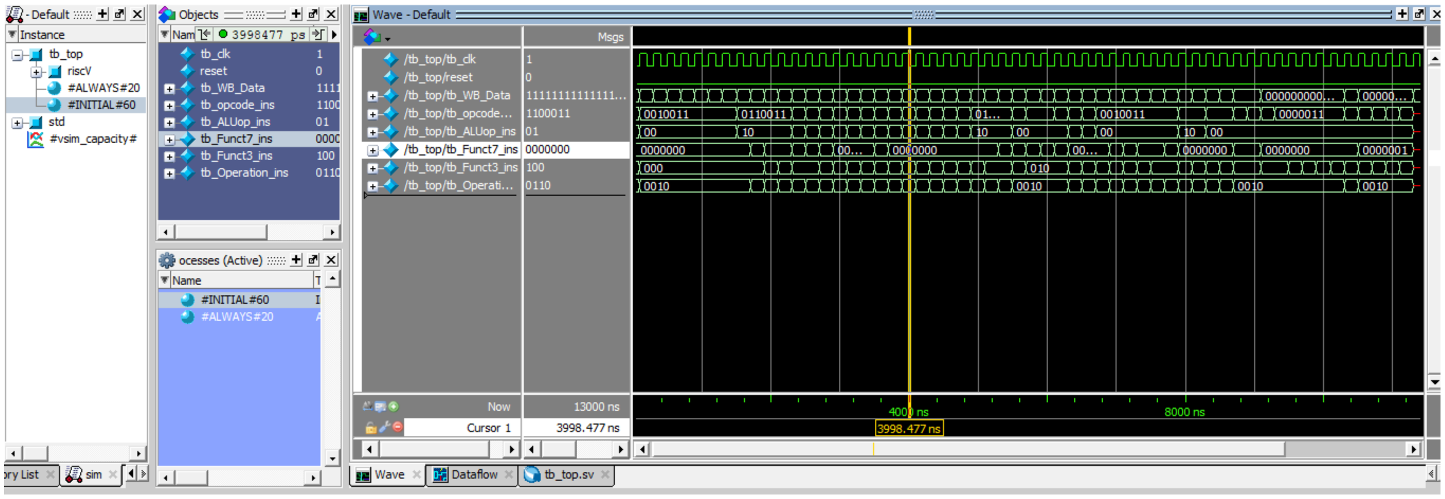
Control Signals for all Type of Instructions



/tb_top/tb_clk	1
/tb_top/reset	0
/tb_top/tb_WB_Data	0000...
/tb_top/tb_opcode...	0010...
/tb_top/tb_ALUSrc_in	1
/tb_top/tb_MemtoR...	0
/tb_top/tb_RegtoM...	0
/tb_top/tb_RegWrit...	1
/tb_top/tb_MemRe...	0
/tb_top/tb_MemWri...	0
/tb_top/tb_Con_Jal...	0
/tb_top/tb_Con_be...	0
/tb_top/tb_Con_bn...	0
/tb_top/tb_Con_bg...	0
/tb_top/tb_Con_blt_in	0
/tb_top/tb_Con_Jal_in	0
/tb_top/tb_Branch_in	0
/tb_top/tb_Mem_in	0
/tb_top/tb_OpI_in	1

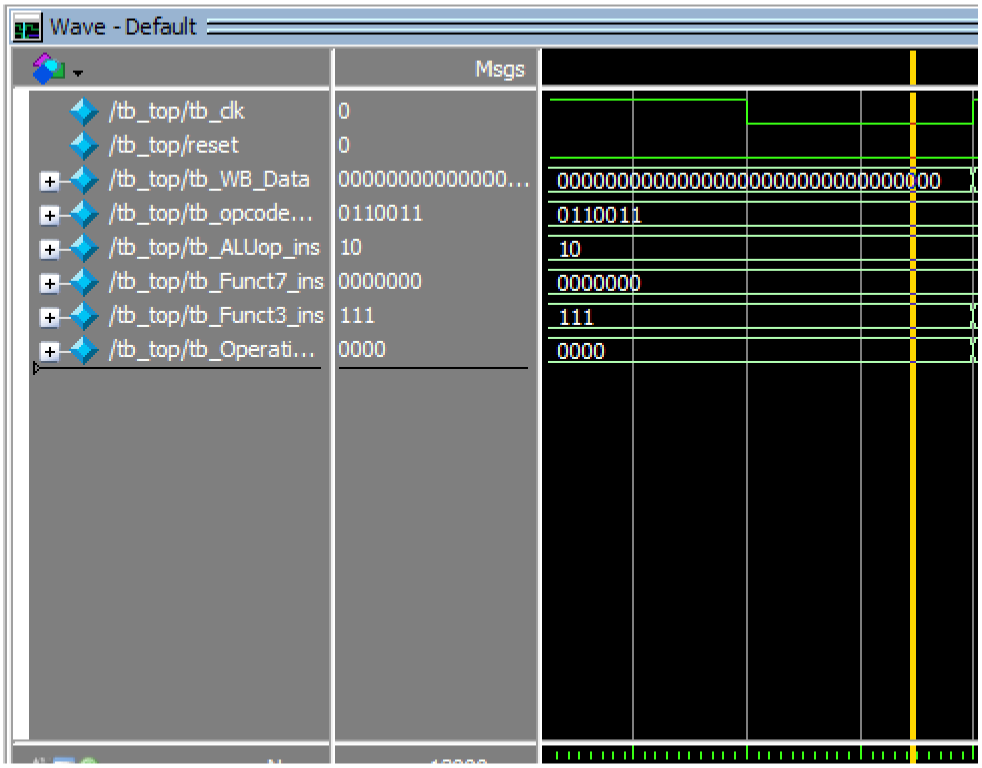
```
# Break in Module tb_top at F:/semester 5/DSD/riscv2/tb_top.sv line 51
# Break key hit
# Break key hit
VSIM 2>
VSIM 2>
```

Function 7 and Function 3 with ALU Result for all Instructions



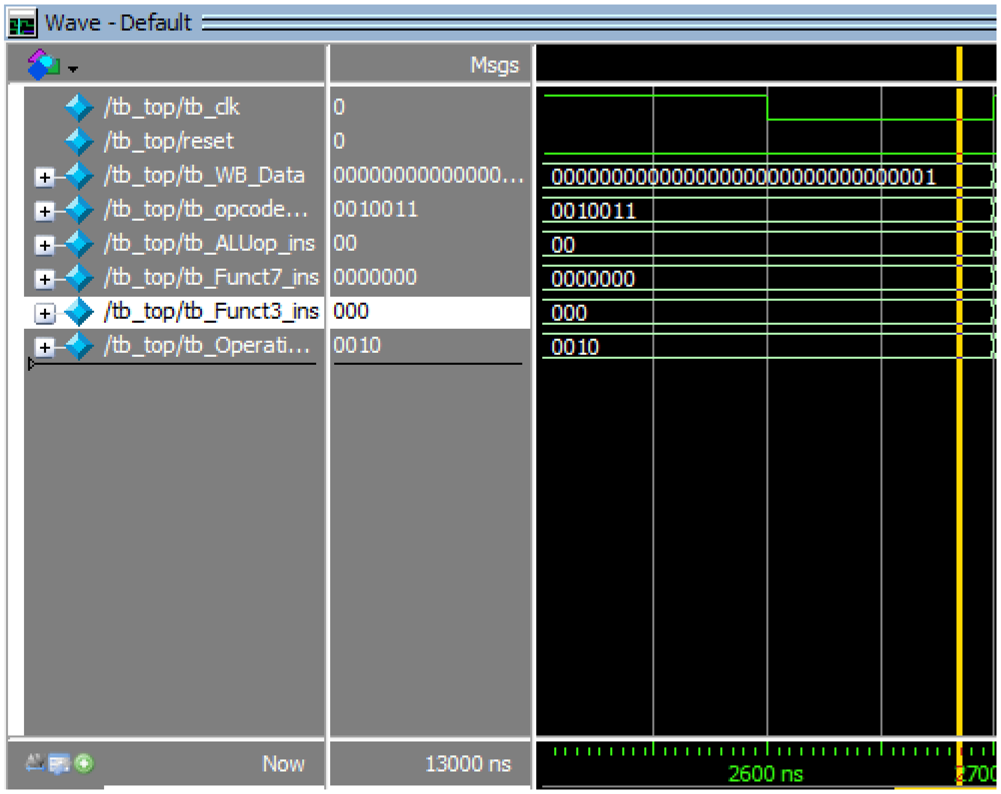
```
/tb_top/tb_clk          1
/tb_top/reset           0
/tb_top/tb_WB_Data      11111111111111...
/tb_top/tb_opcode...    1100011
/tb_top/tb_ALUop_ins    01
/tb_top/tb_Funct7_ins   0000000
/tb_top/tb_Funct3_ins   100
/tb_top/tb_Operati...   0110

Now       13000 ns
Cursor 1  3998.477 ns
```

R-type



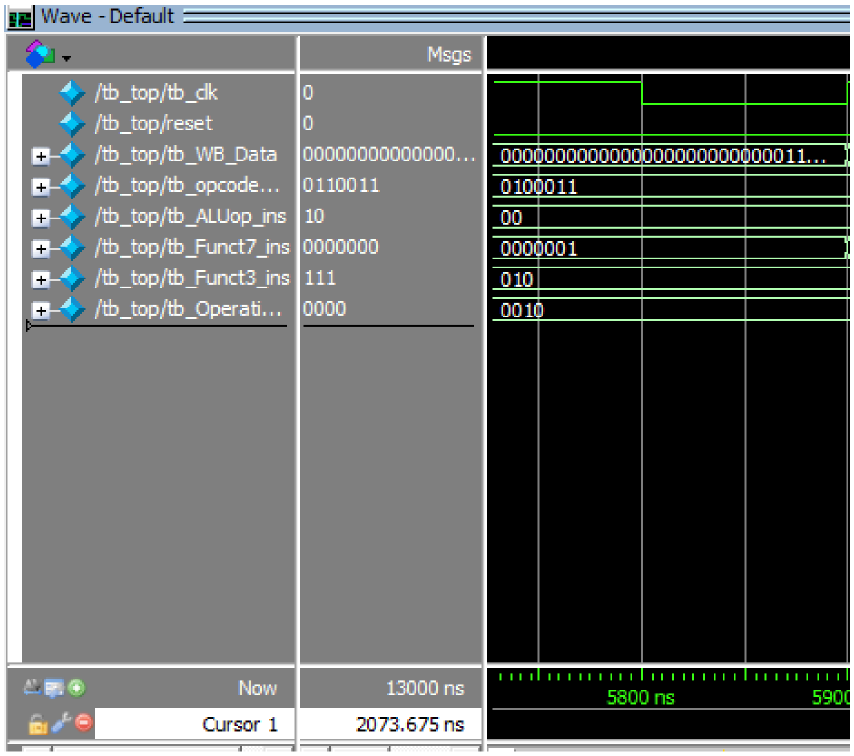
```
/tb_top/tb_clk          0
/tb_top/reset           0
/tb_top/tb_WB_Data      00000000000000...
/tb_top/tb_opcode...    0110011
/tb_top/tb_ALUop_ins    10
/tb_top/tb_Funct7_ins   0000000
/tb_top/tb_Funct3_ins   111
/tb_top/tb_Operati...   0000
```

I-type



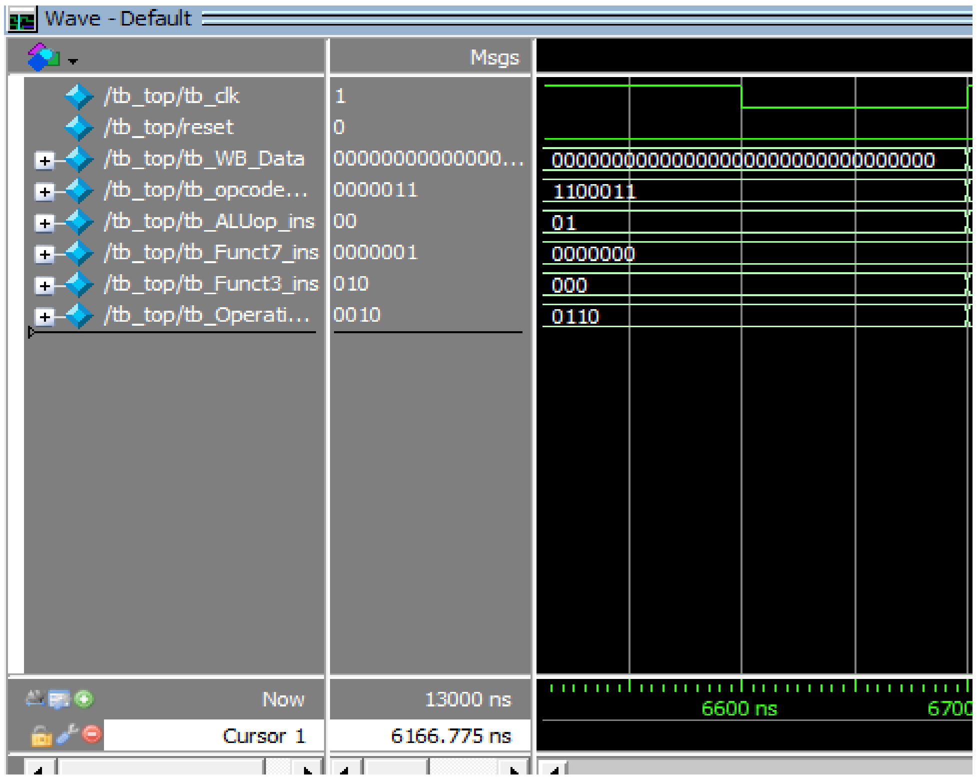
```
Wave - Default
                              Msgs
/tb_top/tb_clk                0
/tb_top/reset                 0
/tb_top/tb_WB_Data            00000000000000...   00000000000000000000000000000001
/tb_top/tb_opcode...          0010011             0010011
/tb_top/tb_ALUop_ins          00                  00
/tb_top/tb_Funct7_ins         0000000             0000000
/tb_top/tb_Funct3_ins         000                 000
/tb_top/tb_Operati...         0010                0010
Now                           13000 ns            2600 ns     2700
```

S-type



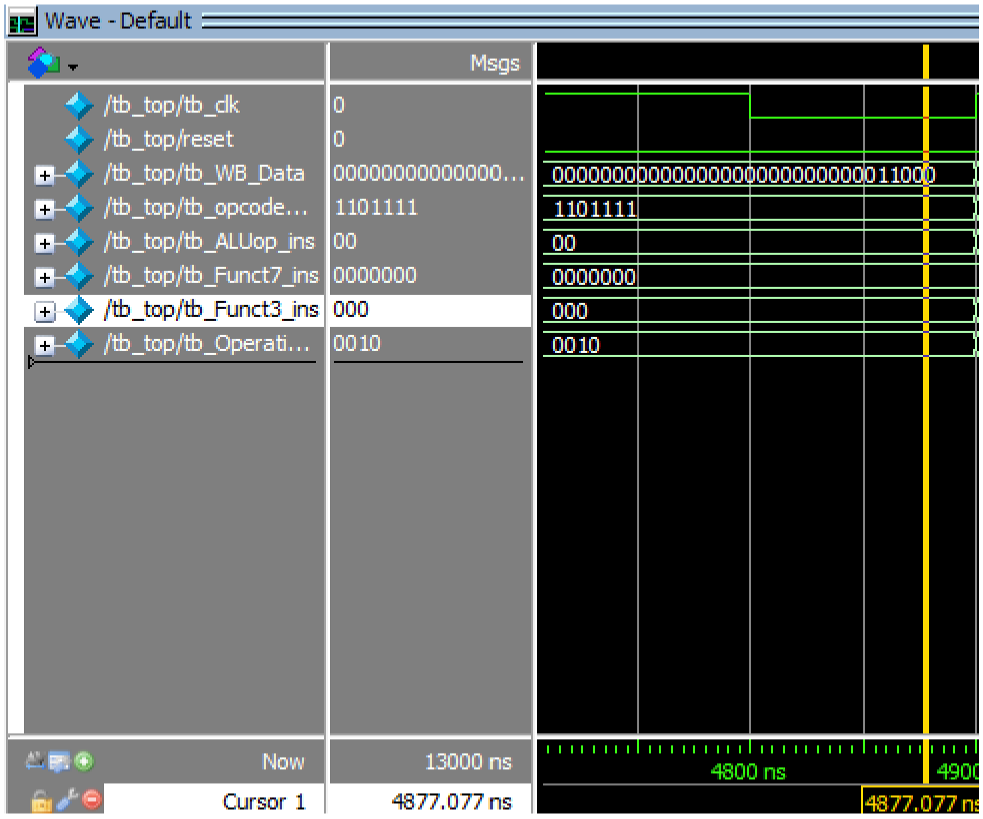
```
Wave - Default
                              Msgs
/tb_top/tb_clk                0
/tb_top/reset                 0
/tb_top/tb_WB_Data            00000000000000...   0000000000000000000000000011...
/tb_top/tb_opcode...          0110011             0100011
/tb_top/tb_ALUop_ins          10                  00
/tb_top/tb_Funct7_ins         0000000             0000001
/tb_top/tb_Funct3_ins         111                 010
/tb_top/tb_Operati...         0000                0010
Now                           13000 ns            5800 ns     5900
Cursor 1                      2073.675 ns
```

SB-type



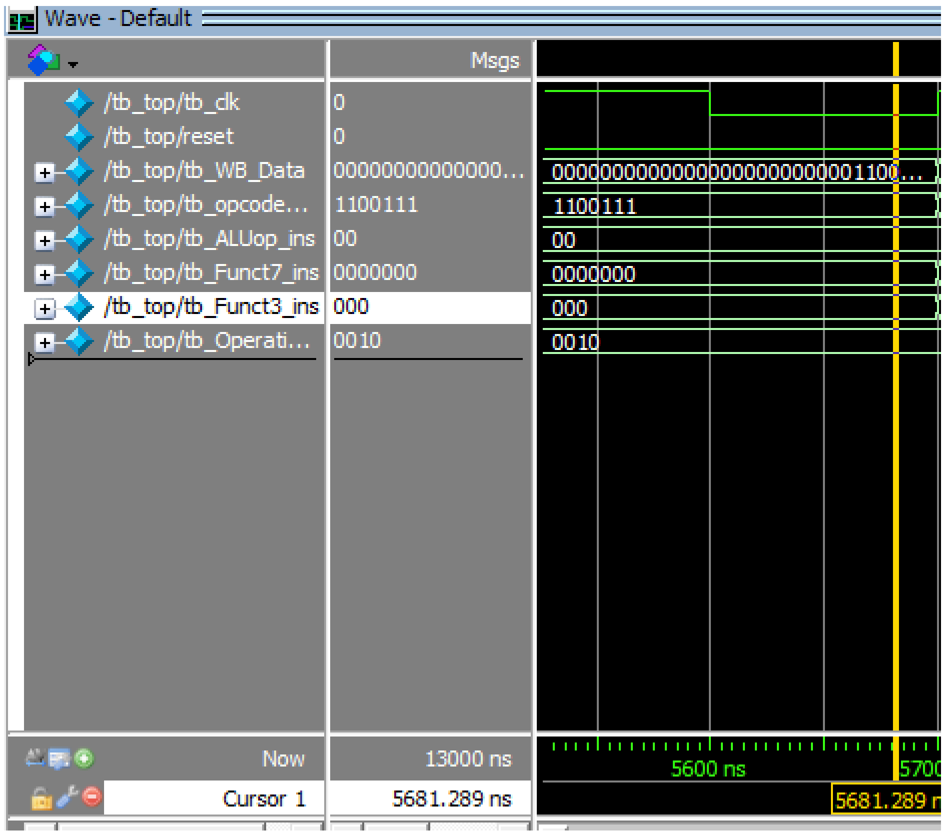
```
Wave - Default
                          Msgs
/tb_top/tb_clk            1
/tb_top/reset             0
/tb_top/tb_WB_Data        00000000000000...   00000000000000000000000000000000
/tb_top/tb_opcode...      0000011             1100011
/tb_top/tb_ALUop_ins      00                  01
/tb_top/tb_Funct7_ins     0000001             0000000
/tb_top/tb_Funct3_ins     010                 000
/tb_top/tb_Operati...     0010                0110

Now                       13000 ns            6600 ns        6700
Cursor 1                  6166.775 ns
```

JAL-type



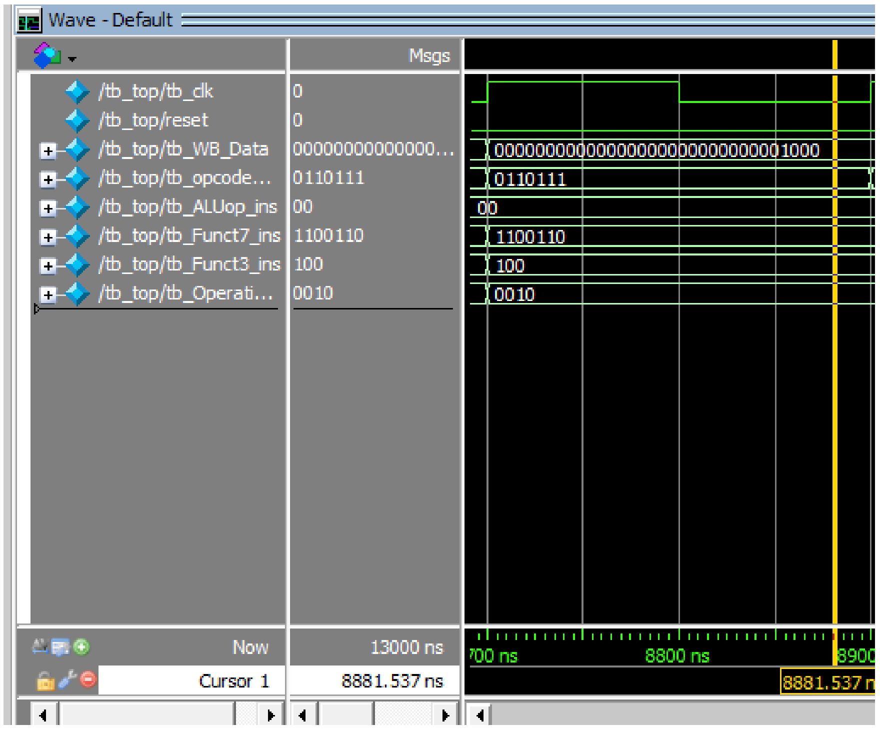
```
Wave - Default
                          Msgs
/tb_top/tb_clk            0
/tb_top/reset             0
/tb_top/tb_WB_Data        00000000000000...   00000000000000000000000000011000
/tb_top/tb_opcode...      1101111             1101111
/tb_top/tb_ALUop_ins      00                  00
/tb_top/tb_Funct7_ins     0000000             0000000
/tb_top/tb_Funct3_ins     000                 000
/tb_top/tb_Operati...     0010                0010

Now                       13000 ns            4800 ns        4900
Cursor 1                  4877.077 ns         4877.077 ns
```

JALR-type



```
/tb_top/tb_clk          0
/tb_top/reset           0
/tb_top/tb_WB_Data      00000000000000...   000000000000000000000000001100...
/tb_top/tb_opcode...    1100111             1100111
/tb_top/tb_ALUop_ins    00                  00
/tb_top/tb_Funct7_ins   0000000             0000000
/tb_top/tb_Funct3_ins   000                 000
/tb_top/tb_Operati...   0010                0010

Now        13000 ns
Cursor 1   5681.289 ns
```

LUI-type



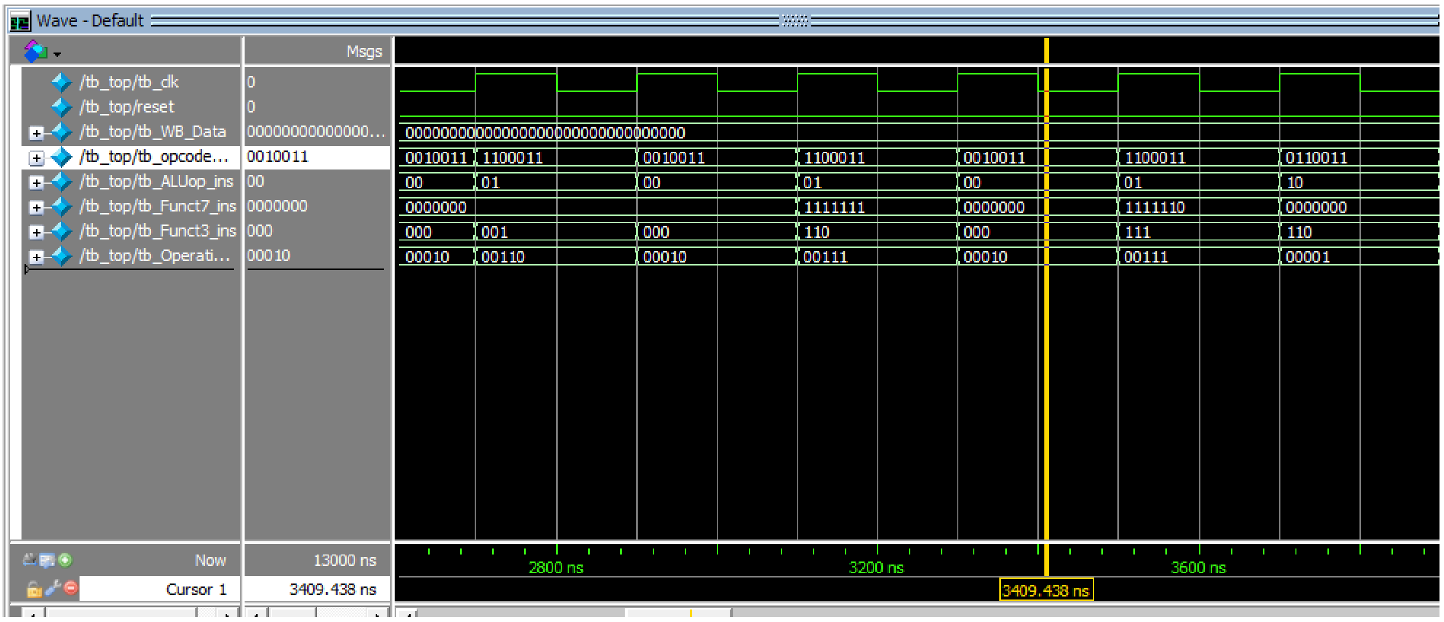
```
/tb_top/tb_clk          0
/tb_top/reset           0
/tb_top/tb_WB_Data      00000000000000...   00000000000000000000000000001000
/tb_top/tb_opcode...    0110111             0110111
/tb_top/tb_ALUop_ins    00                  00
/tb_top/tb_Funct7_ins   1100110             1100110
/tb_top/tb_Funct3_ins   100                 100
/tb_top/tb_Operati...   0010                0010

Now        13000 ns
Cursor 1   8881.537 ns
```

MEMCOPY and MUL

Instructions

```
// mem mul testing (32'b0111000 00101 00100 000 00111 0110011;)- new instruction
assign Inst_mem[57]   = 32'b00000000010000001000001000010011;//add 100 (4 in decimal) to register
assign Inst_mem[58]   = 32'b00000000010100010000001010010011;//add 101 (5 in decimal) to register
assign Inst_mem[59]   = 32'b01110000010100100000001110110011;//multiply regidter 0100 and 0101

// memcopy testing instructionv format-(offset)(copy to)(copy from-9)(opcode-7)
//                                  (0000000)(000000111)(000000101)(0001000)
assign Inst_mem[60]   = 32'b01110000010100100000001110001000;
assign Inst_mem[61]   = 32'b00000000010000001000001000001000;
```

Rtl simulations



```
/tb_top/tb_clk          0
/tb_top/reset           0
/tb_top/tb_WB_Data      00000000000000...   00000000000000000000000000000000
/tb_top/tb_opcode...    0010011   0010011 1100011 0010011 1100011 0010011 1100011 0110011
/tb_top/tb_ALUop_ins    00        00 01 00 01 00 01 10
/tb_top/tb_Funct7_ins   0000000   0000000 1111111 0000000 1111110 0000000
/tb_top/tb_Funct3_ins   000       000 001 000 110 000 111 110
/tb_top/tb_Operati...   00010     00010 00110 00010 00111 00010 00111 00001
```