MAPÚA MALAYAN COLLEGES MINDANAO

SAP (SIMPLE-AS-POSSIBLE) COMPUTER DESIGN AND DEVELOPMENT

A Modular Assessment submitted
in Partial Fulfillment of the Requirements for the Course
CPE131L-1 Computer Architecture and Organization (Laboratory)

by:

Ralph Edcel R. Fabian

Bryxter N. Bulisig

Florencio N. Pulido III

Louis Raphael Q. Lagare

Peter Anthony Miles O. Laporre

Bachelor of Science in Computer Engineering

Course Professor

Marlon V. Maddara

College of Engineering and Architecture

November 2024

TABLE OF CONTENTS

OVERVIEW OF THE APPLICATION

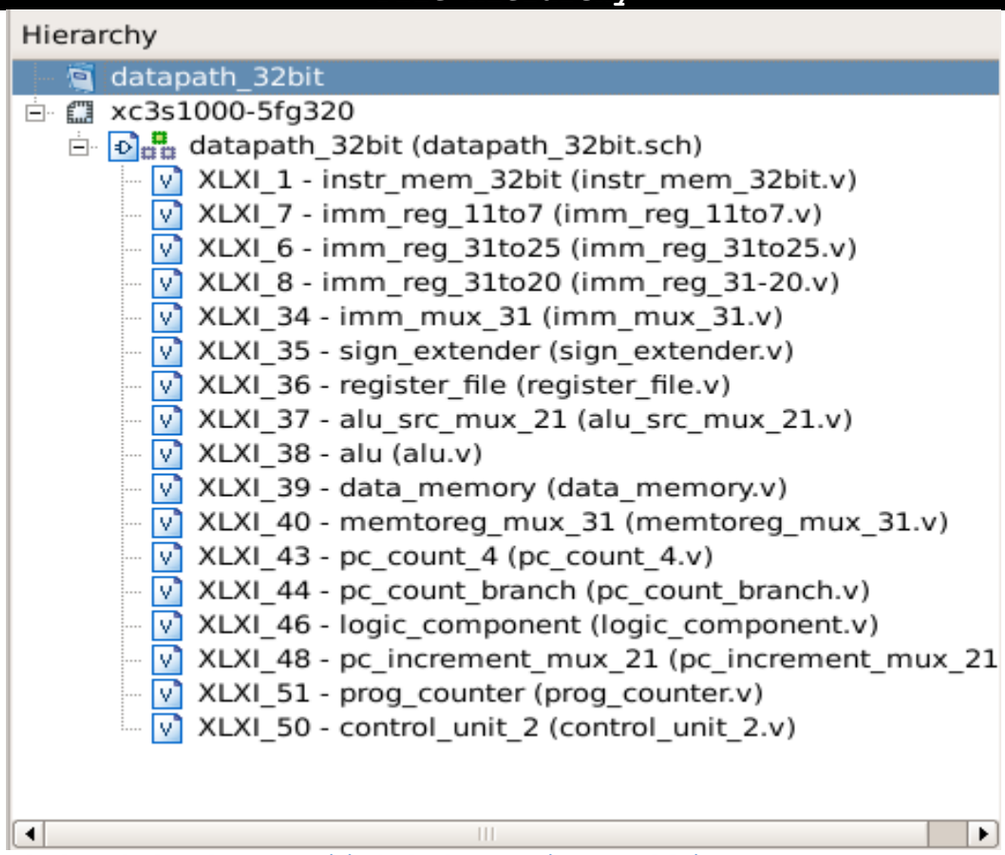
The Simple as Possible (SAP) computer developed using Xilinx AMD [1] Software is designed to implement fundamental computer architecture concepts. The SAP computer focuses on the basics of instruction processing, memory management, and arithmetic operations. Its simplicity makes it an excellent tool for educational purposes, providing an introduction to key concepts such as instruction execution, memory addressing, control units, and arithmetic logic units (ALUs) [2]. The project involves designing key components such as the program counter, instruction memory, register file, ALU, data memory, and control unit, all of which work together to execute instructions following the RISC-V architecture.

The SAP computer operates in a controlled environment where the processing of instructions follows a cycle that includes fetching from memory, decoding, executing, and storing the results [3]. The instructions used in this project such as load, store, branch, arithmetic, and logical operations form the core of the instruction set architecture. This project focuses only on the use of R-type, I-type, S-type, and B-type instructions, leveraging simple control signals to facilitate smooth execution of operations.

Through this project, fundamental aspects of computer architecture were explored and implemented which provides hands-on experience in designing and optimizing a working computing system. The focus on simplicity ensures that the system can be easily understood and further expanded by students or beginners looking to delve into the intricacies of computer design.

COMPLETE SOURCE CODE

File Hierarchy



Github Link: https://github.com/EDCEL02/RISC-V-SAP-

Instruction Memory

```
module instr_mem_32bit(
    input [31:0] count,             // Address provided by
the program counter
    input rst,                    // Reset signal
    output [31:0] instr_code         // 32-bit instruction
output
);
      reg [7:0] mem [100:0];       // Memory to hold larger
space using 16-bit addressing

    assign   instr_code   =   {mem[count+3],   mem[count+2],
mem[count+1], mem[count]};

          always @(rst)
          begin
               if(rst == 1)
                    begin
                         //  Little-endian  format  (least
significant byte first)

                         //sw  x20,  -1(x30)  ||  x20  =  0xB;
x30 = 0xA
                         mem[16'h0000]  =  8'hA3;   //  LSB
(least significant byte)
                         mem[16'h0001] = 8'h2F;
                         mem[16'h0002] = 8'h4F;
```

```
                          mem[16'h0003]  =  8'hFF;    //  MSB
(most significant byte)

                          //sw  x21,  -2(x31)  ||  x21  =  0x5;
x31 = 0x9
                          mem[16'h0004] = 8'h23;
                          mem[16'h0005] = 8'hAF;
                          mem[16'h0006] = 8'h5F;
                          mem[16'h0007] = 8'hFF;

                          //lw x22, -1(x30)
                        mem[16'h0008] = 8'h03;
                          mem[16'h0009] = 8'h2B;
                          mem[16'h000A] = 8'hFF;
                          mem[16'h000B] = 8'hFF;

                          //lw x23, -2(x31)
                          mem[16'h000C] = 8'h83;
                          mem[16'h000D] = 8'hAB;
                          mem[16'h000E] = 8'hEF;
                          mem[16'h000F] = 8'hFF;

                          //xor x24, x22, x23
                          mem[16'h0010] = 8'h33;
                          mem[16'h0011] = 8'h4C;
                          mem[16'h0012] = 8'h7B;
                          mem[16'h0013] = 8'h01;

                          //beq x24, x1, 8
                          mem[16'h0014] = 8'h63;
                          mem[16'h0015] = 8'h08;
                          mem[16'h0016] = 8'h1C;
                          mem[16'h0017] = 8'h00;


                          //skipped
                        mem[16'h0018] = 8'hA3;
                          mem[16'h0019]=  8'h12;
                          mem[16'h001A] = 8'h7B;
                          mem[16'h001B] = 8'h01;

                          mem[16'h001C] = 8'h33;
                          mem[16'h001D]=  8'h1C;
                          mem[16'h001E] = 8'h3B;
                          mem[16'h001F] = 8'h41;

                          mem[16'h0020] = 8'h53;
                          mem[16'h0021]=  8'h6C;
                          mem[16'h0022] = 8'h8B;
                          mem[16'h0023] = 8'h91;

                          //branched
                          //addi x25, x24,-2
                          mem[16'h0024] = 8'h93;
                          mem[16'h0025] = 8'h0C;
                          mem[16'h0026] = 8'hEC;
                          mem[16'h0027] = 8'hFF;
```

```
                        end
            end

endmodule
```

Immediate Register [11:7]

```
module imm_reg_11to7(
    input [4:0] imm_in,         // 5-bit input (immediate
field)
    output [4:0] imm_out       // 5-bit output (immediate
field)
);

    // Assign the input directly to the output
    assign imm_out = imm_in;

endmodule
```

Immediate Register [31:25]

```
module imm_reg_31to25(
    input [6:0] imm_in,         // 7-bit input (immediate
field)
    output [6:0] imm_out       // 7-bit output (immediate
field)
);

    // Assign the input directly to the output
    assign imm_out = imm_in;

endmodule
```

Immediate Register [31:20]

```
module imm_reg_31to20(
    input [11:0] imm_in,        // 12-bit input (immediate
field)
    output [11:0] imm_out      // 12-bit output (immediate
field)
);

    // Assign the input directly to the output
    assign imm_out = imm_in;

endmodule
```

Immediate Multiplexer

```
module imm_mux_31(
    input [11:0] imm_31to20,    // Immediate from bits 31:20
(12 bits)
    input [6:0] imm_31to25,     // Immediate from bits 31:25
(7 bits)
```

```
    input [4:0] imm_11to7,      // Immediate from bits 11:7
(5 bits)
    input [1:0] instruction_type, // Control signal from
the control unit
    output reg [11:0] imm_out   // 12-bit output immediate
);

  always @(*) begin
        case (instruction_type)
            2'b00: begin
                // R-type: Do nothing (output should remain
unchanged)
                imm_out = 12'b0;   // Output zero as a
default for R-type
            end

            2'b01: begin
                // I-type: Output immediate from bits 31:20
(12-bit value)
                imm_out = imm_31to20;
            end

                     2'b10: begin
                // S-type: Combine 31:25 (7 bits) and 11:7
(5 bits)
                imm_out = {imm_31to25, imm_11to7};
            end

            2'b11: begin
                //   B-type:   Combine   imm[31],   imm[7],
imm[30:25], imm[11:8]
                imm_out = {imm_31to25[6], imm_11to7[0],
imm_31to25[5:0], imm_11to7[4:1]};
            end

                    default: begin
                imm_out = 12'b0; // Default to 0 in case of
an invalid instruction type
            end

           endcase
    end
endmodule
```

Sign Extender

```
module sign_extender(
    input [11:0] imm_in,     // 12-bit input immediate
    output [31:0] imm_out     // 32-bit output after sign
extension
);

    // If the 12th bit (MSB) is 1, pad with 1's, else pad
with 0's
      // just copy the 12th bit and replicate
    assign imm_out = {{20{imm_in[11]}}, imm_in};

endmodule
```

Register File

```
module register_file(

          input [4:0] RR_1, //read register 1
          input [4:0] RR_2, //read register 2
          input [4:0] WA,   //Write Address
          input [31:0] WD, //Write Data


          output [31:0] RD_1, //Read Data 1
          output [31:0] RD_2, //Read Data 2

          input reg_wr, //register write from control unit
          input clk,
          input rst

    );

          integer i = 0;

          reg [31:0] reg_mem [31:0];

        // Reset logic
          always @(posedge clk) begin
               if (rst == 1) begin
                         // Initialize registers
                      reg_mem[0]   <= 32'h00000000;  //Keep
this 0
                      reg_mem[1]  <= 32'h0000000E;
                      reg_mem[2]  <= 32'h00000000;
                      reg_mem[3]  <= 32'h00000000;
                      reg_mem[4]  <= 32'h00000000;
                      reg_mem[5]  <= 32'h00000000;
                      reg_mem[6]  <= 32'h00000000;
                      reg_mem[7]  <= 32'h00000000;
                      reg_mem[8]  <= 32'h00000000;
                      reg_mem[9]  <= 32'h00000000;
                      reg_mem[10] <= 32'h00000000;
                      reg_mem[11] <= 32'h00000000;
                      reg_mem[12] <= 32'h00000000;
                      reg_mem[13] <= 32'h00000000;
                      reg_mem[14] <= 32'h00000000;
                      reg_mem[15] <= 32'h00000000;
                      reg_mem[16] <= 32'h00000000;
                      reg_mem[17] <= 32'h00000000;
                      reg_mem[18] <= 32'h00000000;
                      reg_mem[19] <= 32'h00000000;
                      reg_mem[20] <= 32'h0000000B;  //used
in first instruction
                      reg_mem[21] <= 32'h00000005;  //used
in second instruction
                      reg_mem[22] <= 32'h00000000;
                      reg_mem[23] <= 32'h00000000;
                      reg_mem[24] <= 32'h00000000;
                      reg_mem[25] <= 32'h00000000;
                      reg_mem[26] <= 32'h00000000;
                      reg_mem[27] <= 32'h00000000;
                      reg_mem[28] <= 32'h00000001;
```

```
                        reg_mem[29] <= 32'h00000000;
                        reg_mem[30] <= 32'h0000000A;  // used
in first instruction
                        reg_mem[31] <= 32'h00000009;  // used
in second instruction


            // Add more register initialization if needed
        end
            else if (reg_wr && WA != 5'b00000) begin
            // Write to the register if reg_wr is enabled
and WA is not register x0
            reg_mem[WA] <= WD;
        end
          end

          // Read from registers (combinational logic)
          assign RD_1 = reg_mem[RR_1]; // Read data from
register RR_1
          assign RD_2 = reg_mem[RR_2]; // Read data from
register RR_2

          //assign RD_1 = 32'h2;
          //assign RD_2 = 32'h3;
endmodule
```

ALU Source Multiplexer

```
module alu_src_mux_21(
    input [31:0] RD2,         // 32-bit input from register
RD2
    input [31:0] imm,          // 32-bit immediate input
    input alu_src,            // Control signal: 0 = RD2, 1
= imm
    output reg [31:0] alu_in  // 32-bit output to the ALU
);
      always @(*) begin
        if (alu_src == 1'b0)
            alu_in = RD2;     // Select RD2 if alu_src is 0
        else
            alu_in = imm;    // Select immediate if alu_src
is 1
    end


endmodule
```

Arithmetic Logical Unit (ALU)

```
module alu(
    input [31:0] in1,       // Unsigned 32-bit input 1 from
register file
    input [31:0] in2,       // Unsigned 32-bit input 2 from
register file
    input [4:0] alu_ctrl,   // ALU control signal
    output reg [31:0] alu_res,  // 32-bit ALU result (signed
result)
```

```
    output reg zero_flag       // Zero flag for conditional
branches
);

   // Internal signed versions of inputs
   reg signed [31:0] signed_in1;
   reg signed [31:0] signed_in2;


     always @(*)begin
          // Sign-extend in1 and in2
      signed_in1 = (in1[31] == 1'b1) ? {1'b1, in1[30:0]} :
in1;  // Treat in1 as signed if MSB is 1
      signed_in2 = (in2[31] == 1'b1) ? {1'b1, in2[30:0]} :
in2;  // Treat in2 as signed if MSB is 1

          case (alu_ctrl)
               5'b00000: alu_res = signed_in1 + signed_in2;
// Perform signed addition
         5'b00001: alu_res = signed_in1 - signed_in2;   //
Perform signed subtraction
         5'b00010: alu_res = signed_in1 & signed_in2;   //
Perform AND
         5'b00011: alu_res = signed_in1 | signed_in2;   //
Perform OR
         5'b00100: alu_res = signed_in1 ^ signed_in2;   //
Perform XOR

               5'b00101:   alu_res   =   ~(signed_in1   ^
signed_in2); // Perform XNOR
         5'b00110: alu_res = signed_in1 * signed_in2;   //
Perform signed multiplication
         //5'b00111: alu_res = signed_in1 / signed_in2;  //
Perform signed division
         5'b01000: alu_res = signed_in1 << in2[4:0];    //
Perform logical shift left
         5'b01001: alu_res = signed_in1 >> in2[4:0];    //
Perform logical shift right

               5'b01010: alu_res = signed_in1 >>> in2[4:0];
// Perform arithmetic shift right
         5'b01011: alu_res = signed_in1 < signed_in2 ? 1 :
0;  // Set less than (signed)
         5'b01100: alu_res = in1 < in2 ? 1 : 0;  // Set less
than (unsigned)
         //5'b01101: alu_res = signed_in1 % signed_in2;  //
Perform signed modulus (remainder)
      endcase

          if (alu_res == 0)
               zero_flag = 1'b1;
          else
               zero_flag = 1'b0;
     end



endmodule
```

Data Memory

```
module data_memory(
    input clk,                  // Clock signal
    input reset,                // Reset signal
    input write,                // Write enable signal (for
store operations)
    input read,                  // Read enable signal (for
load operations)
    input [31:0] rd_add,        // 32-bit read address input
    input [31:0] wr_data,       // 32-bit write data input
    output reg [31:0] rd_data   // 32-bit read data output
);

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

      // Memory initialization upon reset
    always @(posedge clk) begin
        if (reset) begin
            //  Manual  initialization  of  memory  with
hexadecimal values
                    mem[32'h0]    <=  32'h00000000;    //
Register 0
            mem[32'h1]  <= 32'h00000000;  // Register 1
            mem[32'h2]  <= 32'h00000000;  // Register 2
            mem[32'h3]  <= 32'h00000000;  // Register 3
            mem[32'h4]  <= 32'h00000000;  // Register 4
            mem[32'h5]  <= 32'h00000000;  // Register 5
            mem[32'h6]  <= 32'h00000000;  // Register 6
            mem[32'h7]  <= 32'h00000000;  // Register 7
                    mem[32'h8]    <=  32'h00000000;    //
Register 8
            mem[32'h9]  <= 32'h00000000;  // Register 9
            mem[32'hA]  <= 32'h00000000;  // Register 10
            mem[32'hB]  <= 32'h00000000;  // Register 11
            mem[32'hC]  <= 32'h00000000;  // Register 12
            mem[32'hD]  <= 32'h00000000;  // Register 13
            mem[32'hE]  <= 32'h00000000;  // Register 14
            mem[32'hF]  <= 32'h00000000;  // Register 15
               end

               else if (write) begin
            // Write to memory when write enable is active
            mem[rd_add] <= wr_data;
               end
          end


    // Read from memory
    always @(negedge clk) begin
        if (read == 1) begin
            rd_data <= mem[rd_add];
        end
    end


endmodule

     always @(*)begin
```

```
          // Sign-extend in1 and in2
      signed_in1 = (in1[31] == 1'b1) ? {1'b1, in1[30:0]} :
in1;  // Treat in1 as signed if MSB is 1
      signed_in2 = (in2[31] == 1'b1) ? {1'b1, in2[30:0]} :
in2;  // Treat in2 as signed if MSB is 1

          case (alu_ctrl)
               5'b00000: alu_res = signed_in1 + signed_in2;
// Perform signed addition
         5'b00001: alu_res = signed_in1 - signed_in2;   //
Perform signed subtraction
         5'b00010: alu_res = signed_in1 & signed_in2;   //
Perform AND
         5'b00011: alu_res = signed_in1 | signed_in2;   //
Perform OR
         5'b00100: alu_res = signed_in1 ^ signed_in2;   //
Perform XOR

               5'b00101:   alu_res   =   ~(signed_in1   ^
signed_in2); // Perform XNOR
         5'b00110: alu_res = signed_in1 * signed_in2;   //
Perform signed multiplication
         //5'b00111: alu_res = signed_in1 / signed_in2;  //
Perform signed division
         5'b01000: alu_res = signed_in1 << in2[4:0];    //
Perform logical shift left
         5'b01001: alu_res = signed_in1 >> in2[4:0];    //
Perform logical shift right

               5'b01010: alu_res = signed_in1 >>> in2[4:0];
// Perform arithmetic shift right
         5'b01011: alu_res = signed_in1 < signed_in2 ? 1 :
0;  // Set less than (signed)
         5'b01100: alu_res = in1 < in2 ? 1 : 0;  // Set less
than (unsigned)
         //5'b01101: alu_res = signed_in1 % signed_in2;  //
Perform signed modulus (remainder)
      endcase

          if (alu_res == 0)
               zero_flag = 1'b1;
          else
               zero_flag = 1'b0;
     end



endmodule
```

Control Unit

```
module control_unit_2(
    input [6:0] op_code,      // Opcode from the instruction
    input [6:0] funct7,          // funct7 field from the
instruction
    input [2:0] funct3,          // funct3 field from the
instruction


    output reg alu_src,         // ALU source (immediate or
register)
    output reg [4:0] alu_ctrl, // ALU control signal
    output reg [1:0] inst_type,// Instruction type (R, I,
S, B, etc.)
    output reg memtoreg,        // Memory to register (for
load instructions)
    output reg dm_write,       // Data memory write enable
    output reg dm_read,        // Data memory read enable
    output reg reg_wr          // Register write enable


    );

          always @(*) begin
        // Default values

        alu_src = 1'b0;
        alu_ctrl = 2'b00;
        inst_type = 2'b00;
        memtoreg = 1'b0;
        dm_write = 0;
        dm_read = 0;
        reg_wr = 1'b0;

              // Decode instruction based on opcode
        case(op_code)

/////////////////////////////////////////////////////R
TYPE//////////////////////////////////////////////////////

            7'b0110011: begin // R-type instructions (ADD,
SUB, etc.)

                          inst_type = 2'b00; //R-Type
                          reg_wr  =  1'b1;        //  Enable
writing to the register
                          alu_src = 1'b0;      //RD2   for
alu_in 2


                          case (funct3)
                    3'b000: begin
                       // ADD or SUB depending on funct7
                       if (funct7 == 7'b0000000)
                          alu_ctrl = 5'b00000; // ADD
                       else if (funct7 == 7'b0100000)
                          alu_ctrl = 5'b00001; // SUB
                    end
```

```
                                3'b001: alu_ctrl = 5'b01000;
// SLL (Shift left logical)
                                3'b011: alu_ctrl = 5'b01011;
// SLT (Set less than, signed comparison)
                                3'b010: alu_ctrl = 5'b01100;
// SLTU (Set less than, unsigned comparison)

                                3'b100: alu_ctrl = 5'b00100;
// XOR

                                3'b101: begin
                                      //     Shift     right
(logical or arithmetic) based on funct7
                                      if      (funct7     ==
7'b0000000)
                                               alu_ctrl     =
5'b01001;  // SRL (Shift right logical)
                                      else  if  (funct7  ==
7'b0100000)
                                               alu_ctrl     =
5'b01010;  // SRA (Shift right arithmetic)
                                end

                                3'b110: alu_ctrl = 5'b00011;
// OR
                                3'b111: alu_ctrl = 5'b00010;
// AND
                                default:     alu_ctrl     =
5'b00000;  // Default to ADD if no match
                          endcase
            end


/////////////////////////////////////////////////////////I
TYPE//////////////////////////////////////////////


                    7'b0010011:   begin   //   I-type   ALU
instructions
                          inst_type  =  2'b01;    //  I-type
instruction
                          alu_src = 1'b1;
                          reg_wr = 1'b1;
                          memtoreg = 1'b0;


                          case (funct3)
                 3'b000: alu_ctrl = 5'b00000;  // ADDI (Add
Immediate)
                 3'b001: alu_ctrl = 5'b01000;  // SLLI (Shift
left logical immediate)
                 3'b010: alu_ctrl = 5'b01011;  // SLTI (Set
less than immediate, signed)
                 3'b011: alu_ctrl = 5'b01100;  // SLTIU (Set
less than immediate, unsigned)
                 3'b100: alu_ctrl = 5'b00100;  // XORI (XOR
immediate)
```

```
                3'b101: begin
                         //  Shift  right  (logical  or
arithmetic) based on funct7
                    if (funct7 == 7'b0000000)
                        alu_ctrl  =  5'b01001;    //  SRLI
(Shift right logical immediate)
                    else if (funct7 == 7'b0100000)
                        alu_ctrl  =  5'b01010;    //  SRAI
(Shift right arithmetic immediate)
                end
                         3'b110: alu_ctrl = 5'b00011;  //
ORI (OR immediate)
                         3'b111: alu_ctrl = 5'b00010;  //
ANDI (AND immediate)
                         default: alu_ctrl = 5'b00000; //
Default to ADDI if no match
                         endcase
                    end

                    7'b0000011:  begin  //  I-type  Load
instructions (LB, LH, LW, LBU, LHU)
                        inst_type  =  2'b01;    //  I-type
instruction
                        alu_src = 1'b1;      // Immediate
used for address offset calculation
                        memtoreg = 1'b1;    // Data memory
output goes to the register
                        dm_read  =  1'b1;        //  Enable
memory read
                        reg_wr  =  1'b1;         //  Enable
register write

                        case (funct3)
                              3'b000: alu_ctrl = 5'b00000;
// LB (Load Byte, sign-extended)
                              3'b001: alu_ctrl = 5'b00000;
// LH (Load Halfword, sign-extended)
                              3'b010: alu_ctrl = 5'b00000;
// LW (Load Word)
                              3'b100: alu_ctrl = 5'b00000;
// LBU (Load Byte Unsigned)
                              3'b101: alu_ctrl = 5'b00000;
// LHU (Load Halfword Unsigned)
                              default:     alu_ctrl     =
5'b00000; // Default to ADD (address calculation)
                        endcase
                    end


/////////////////////////////////////////////////////////S
TYPE//////////////////////////////////////////////////////




                  7'b0100011: begin // S-type instructions
(SB, SH, SW, SD)
```

```
                          inst_type  =  2'b10;    //  S-type
instruction
                          alu_src = 1'b1;
                          reg_wr = 1'b0;
                          memtoreg = 1'b0;
                          dm_write = 1'b1;
                          dm_read = 1'b0;


                           case (funct3)
                                3'b000: alu_ctrl = 5'b00000;
// SB (Store Byte)
                                3'b001: alu_ctrl = 5'b00000;
// SH (Store Halfword)
                                3'b010: alu_ctrl = 5'b00000;
// SW (Store Word)
                                3'b011: alu_ctrl = 5'b00000;
// SD (Store Double Word)
                                default:     alu_ctrl     =
5'b00000; // Default to address calculation (ADD)
                          endcase
                     end



/////////////////////////////////////////////////////B
TYPE//////////////////////////////////////////////////



                     7'b1100011:  begin  //  B-type  branch
instructions
                          inst_type  =  2'b11;    //  B-type
instruction
                          alu_src = 1'b0;

                          case (funct3)
                               3'b000: begin // BEQ (Branch
if equal)
                                    alu_ctrl = 5'b00001;  //
Perform subtraction
                               end

                               3'b001: begin // BNE (Branch
if not equal)
                                    alu_ctrl = 5'b00001;  //
Perform subtraction
                               end

                               3'b100: begin // BLT (Branch
if less than)
                     alu_ctrl  =  5'b00001;    //  Perform
subtraction
                               end

                               3'b101: begin // BGE (Branch
if greater than or equal)
```

```
                    alu_ctrl   =   5'b00001;     //   Perform
subtraction
                              end

                              3'b110: begin // BLTU (Branch
if less than, unsigned)
                    alu_ctrl   =   5'b00001;     //   Perform
subtraction (unsigned)
                              end

                              3'b111: begin // BGEU (Branch
if greater than or equal, unsigned)
                    alu_ctrl   =   5'b00001;     //   Perform
subtraction (unsigned)
                              end

                              default: alu_ctrl = 5'b00000;
// Default to ADD

                         endcase
                    end









                    default: begin
                         //    Default    case:    Disable
everything for unknown opcode
                         alu_src = 1'b1;
                         alu_ctrl = 5'b00000;
                         inst_type = 2'b00;
                         memtoreg = 1'b0;
                         dm_write = 1'b0;
                         dm_read = 1'b0;
                         reg_wr = 1'b0;
                    end
               endcase
          end





endmodule
```

Memory to Register Multiplexer

```
module memtoreg_mux_31(
    input memtoreg,              // Control signal: 0 = ALU
output, 1 = data memory output
    input [31:0] alu_out,        // 32-bit ALU output
    input  [31:0]  data_mem_out,     //  32-bit  data  memory
output
    output reg [31:0] result     // 32-bit output
);

    always @(*) begin
        if (memtoreg == 1'b0)
            result = alu_out;         // Select ALU output
if memtoreg == 0
        else
            result = data_mem_out;   // Select data memory
output if memtoreg == 1
    end

endmodule
```

Program Counter Count + 4

```
module pc_count_4(

    input [31:0] pc_count,
    output reg[31:0] pc_count_4
);
     always @(*) begin
          pc_count_4 = pc_count + 16'd4;
     end

endmodule
```

Program Counter Count + BTA

```
module pc_count_branch(
    input [31:0] pc_count,
    input [31:0] immediate,
    output reg [31:0] pc_count_branch
);

      always @(*) begin
          pc_count_branch        =        pc_count        +
$signed($signed(immediate) * 2);
      end

endmodule
```

Program Counter Logic Component

```
module logic_component(

    input[31:0] alu_res,
      input[2:0] funct_3,

    output reg branch_enable
);
```

```
    always @(*) begin
        case (funct_3)
            3'b000: begin // BEQ (Branch if equal)
                branch_enable = (alu_res == 0) ? 1'b1 :
1'b0;
            end

                    3'b001: begin // BNE (Branch if not
equal)
                branch_enable = (alu_res != 0) ? 1'b1 :
1'b0;
            end

            3'b100: begin // BLT (Branch if less than)
                branch_enable = (alu_res[31] == 1'b1) ?
1'b1 : 1'b0; // Negative result indicates rs1 < rs2
            end

                  3'b101: begin // BGE (Branch if greater
than or equal)
                branch_enable = (alu_res[31] == 1'b0 ||
alu_res == 0) ? 1'b1 : 1'b0; // Non-negative result or zero
            end

            3'b110: begin // BLTU (Branch if less than,
unsigned)
                branch_enable = (alu_res[31] == 1'b1 &&
alu_res != 0) ? 1'b1 : 1'b0; // Unsigned comparison
            end

                  3'b111: begin // BGEU (Branch if greater
than or equal, unsigned)
                branch_enable = (alu_res[31] == 1'b0 ||
alu_res == 0) ? 1'b1 : 1'b0; // Unsigned comparison
            end

            default: branch_enable = 1'b0; // Default to no
branch
        endcase
    end


endmodule
```

Program Counter Multiplexer

```
module pc_increment_mux_21(
    input branch_en,
    input [31:0] pc_4,
    input [31:0] pc_branch,
    output reg [31:0] pc_count_out
);

        always @(*) begin
                 if (branch_en == 1)
                        pc_count_out   =   pc_branch;   //
Select branch address
                 else
```

```
                            pc_count_out = pc_4;     // Select
incremented address
            end


endmodule
```

Program Counter

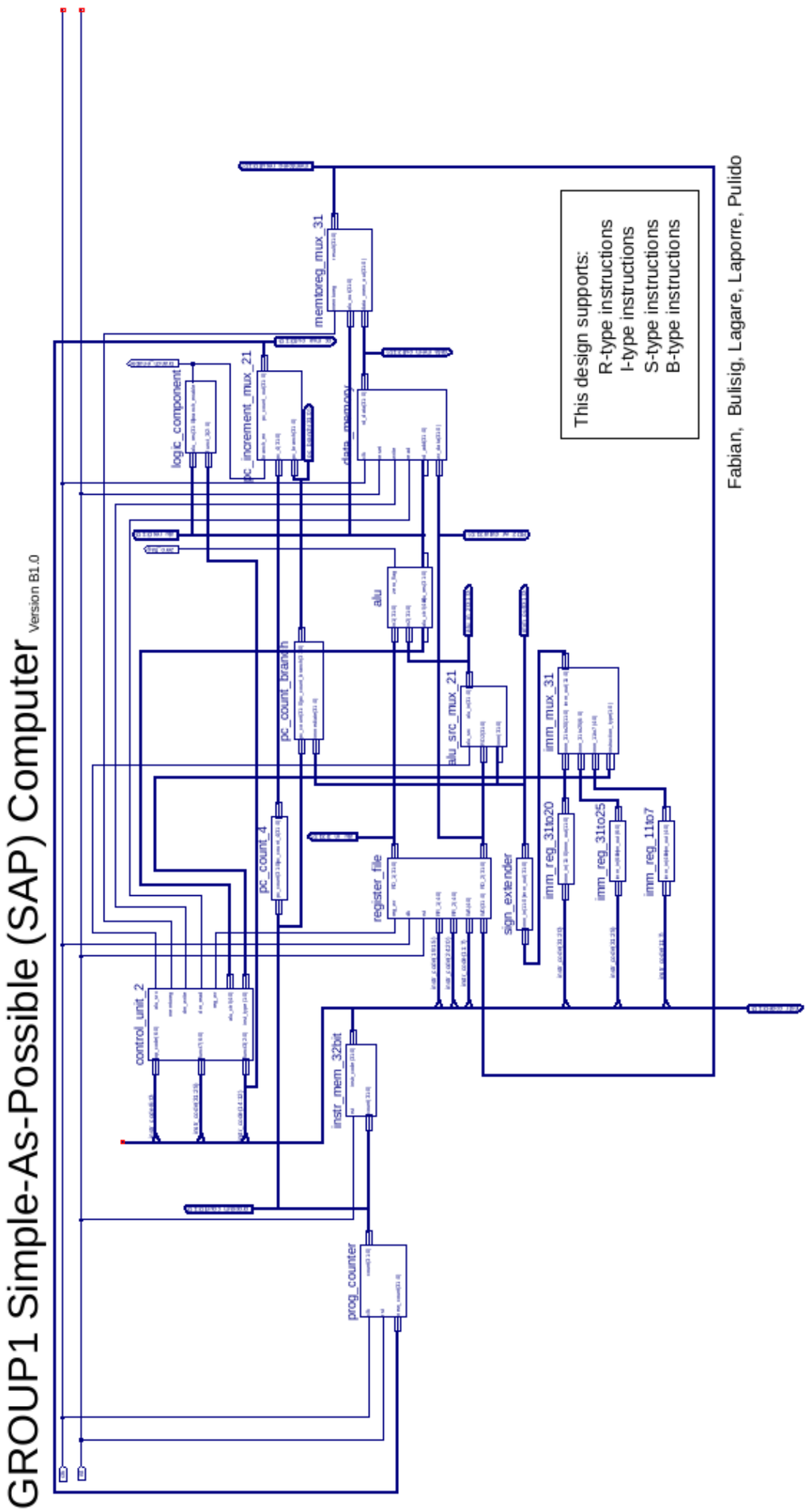
```
module prog_counter(
    input clk,
    input rst,
    input [31:0] new_count,
    output reg [31:0] count
);
    always @(posedge clk) begin
        if (rst) begin
            count = 32'h0;     // Reset the program counter
to 0
        end
        else begin
            count  =  new_count;  //  Update  count  with
new_count, zero-extended to 32 bits
        end
    end
endmodule
```

Test Fixure

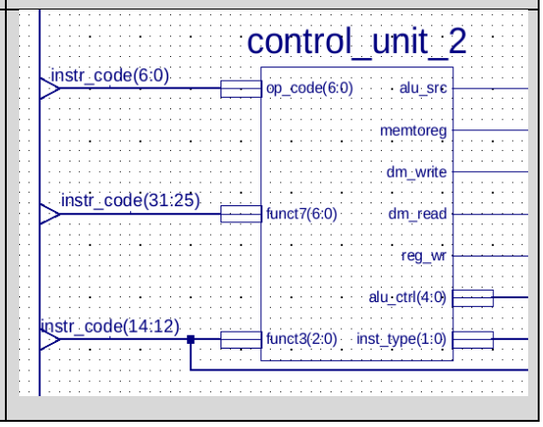
```
// Verilog test fixture created from schematic /home/ise/SAP
Project/Alpha            Version/v8_branch            with
complete/datapath_32bit/datapath_32bit.sch  -  Wed  Nov   6
10:10:29 2024

`timescale 1ns / 1ps

module datapath_32bit_datapath_32bit_sch_tb();

// Inputs
   reg clk;
   reg rst;

// Output
   wire [31:0] instr_code;
   wire [31:0] alu_in_1;
   wire [31:0] alu_in_2;
   wire [31:0] data_mem_out;
   wire [31:0] memtoreg_result;
   wire zero_flag;
   wire [31:0] RD2_wr_data;
   wire [31:0] alu_res;
   wire [31:0] imm_out;
   wire branch_enable;
   wire [31:0] pc_mux_out;
   wire [31:0] program_count;
   wire [31:0] pc_branch;

// Bidirs
```

```
// Instantiate the UUT
   datapath_32bit UUT (
          .instr_code(instr_code),
          .alu_in_1(alu_in_1),
          .alu_in_2(alu_in_2),
          .data_mem_out(data_mem_out),
          .memtoreg_result(memtoreg_result),
          .clk(clk),
          .rst(rst),
          .zero_flag(zero_flag),
          .RD2_wr_data(RD2_wr_data),
          .alu_res(alu_res),
          .imm_out(imm_out),
          .branch_enable(branch_enable),
          .pc_mux_out(pc_mux_out),
          .program_count(program_count),
          .pc_branch(pc_branch)
   );
// Initialize Inputs
          initial
          begin
               clk = 1'b1;
               forever
                    #25 clk = ~clk;
               end

          initial
          begin
               rst = 1'b1;
               #0 rst = 1'b0;
               #500 $stop;
          end
endmodule
```

PROGRAM OUTPUT

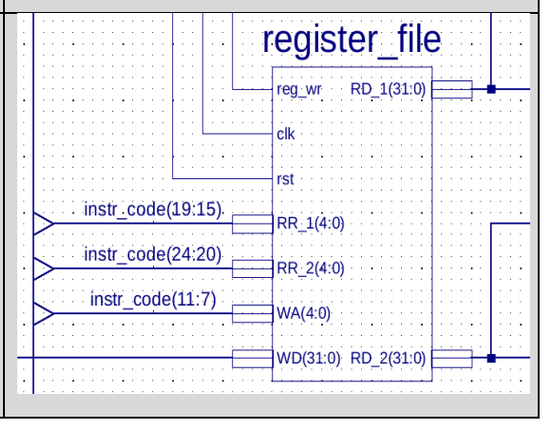


Control Unit

Component Description	Component Photo
The Control Unit sends control signals to the datapath components. This design explicitly extracts the data from the instruction rather than the whole.	

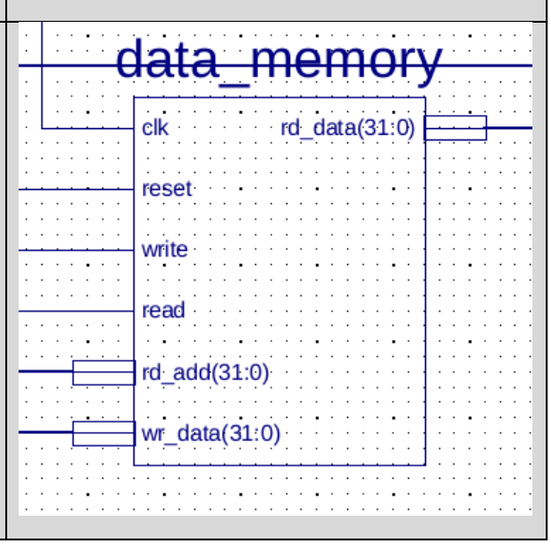
31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type

Register File

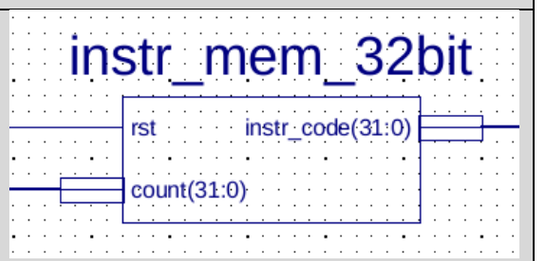
Component Description	Component Photo
The register file stores the data to be feed to ALU or Data Memory. This design explicitly extracts the data from the instruction rather than the whole.	

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type

Data Memory

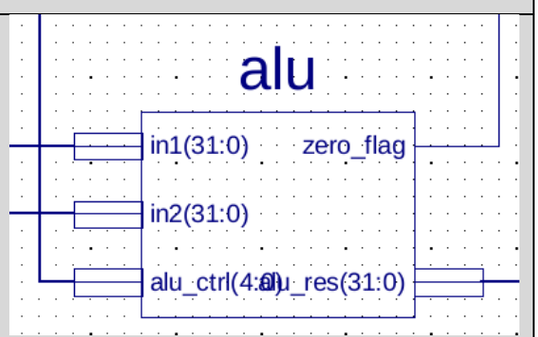
Component Description	Component Photo
The data memory stores the data from Register File's RD2 to the effective address calculated by the ALU.	

Instruction Memory

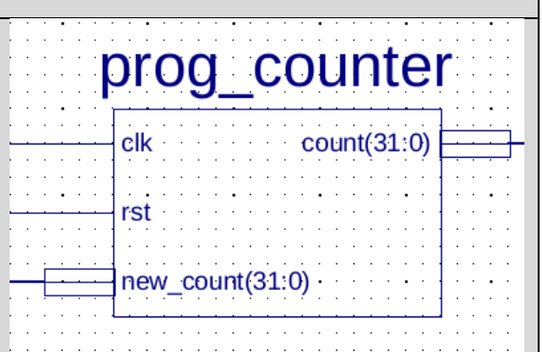
Component Description	Component Photo
The instruction memory stores the instruction. This SAP design doesn't support manual input of instructions due to Xilinx capability. Thus, all instructions are predefined. It uses Little Endian (LSB to MSB)	


```
// Little-endian format (least significant byte first)

//sw x20, -1(x30) || x20 = 0xB; x30 = 0xA
mem[16'h0000] = 8'hA3;  // LSB (least significant byte)
mem[16'h0001] = 8'h2F;
mem[16'h0002] = 8'h4F;
mem[16'h0003] = 8'hFF;  // MSB (most significant byte)
```

Arithmetic Logical Unit (ALU)

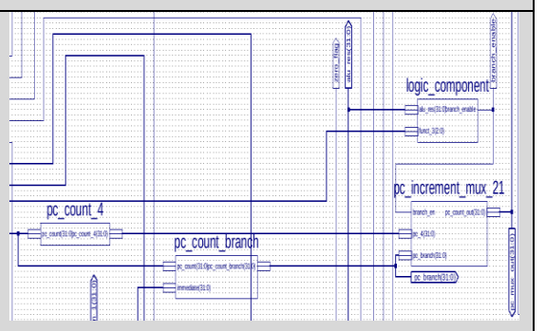
Component Description	Component Photo
The component responsible for the calculation of the two inputs. The calculation differs from the alu_ctrl signal of the Control Unit. Examples are addittion, shift, and subtraction.	

Program Counter

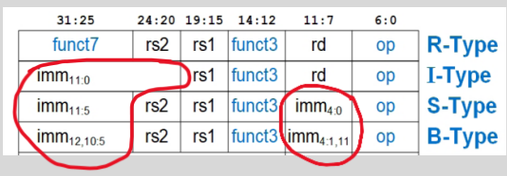
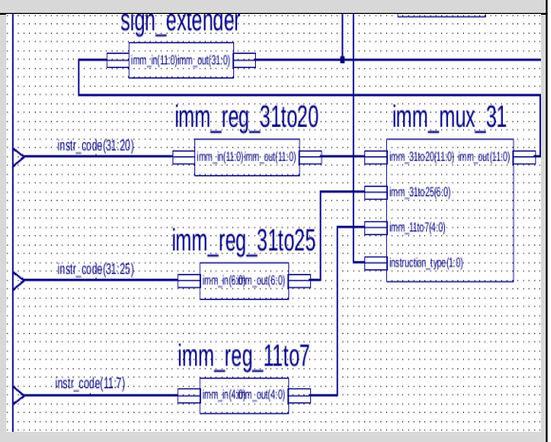
Component Description	Component Photo
The program counter outputs the "new_count input" every posedge clk.	

```
always @(posedge clk) begin
    if (rst) begin
        count = 32'h0;
    end
    else begin
        count = new_count;
    end
end
```

Program Counter Counting Components
(Program Count + 4, Program Count Branch, Logic Component, 2 to 1 Multiplexer)

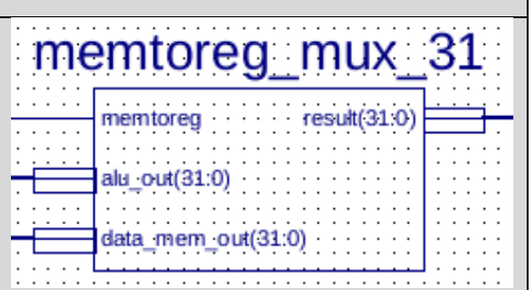
Component Description	Component Photo
This SAP follow the datapath design for branch instructions. The Logic Component outputs the “branch enable” signal based on the ALU result and funct3. If enabled, the multiplexer will choose the branch address calculation; increments by 4 otherwise.	

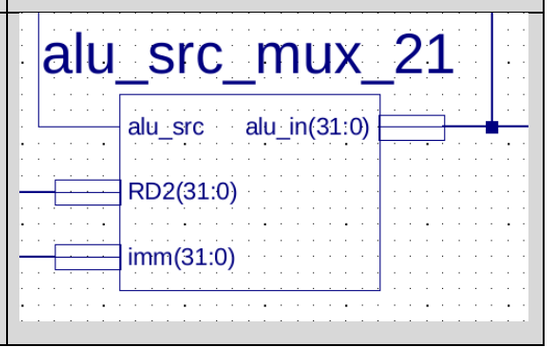
Immediate Related Components
(Sign Extender, Immediate Multiplexer, and immediate registers)

Component Description	Component Photo
This SAP design extracts the immediate values from different index. The immediate multiplexer decides what is the sequence of the bits based on the instruction type (signal from control unit). The sign extender pads the immediate value up to 32nd bit. The 12th bit is the padded bit. 	

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**

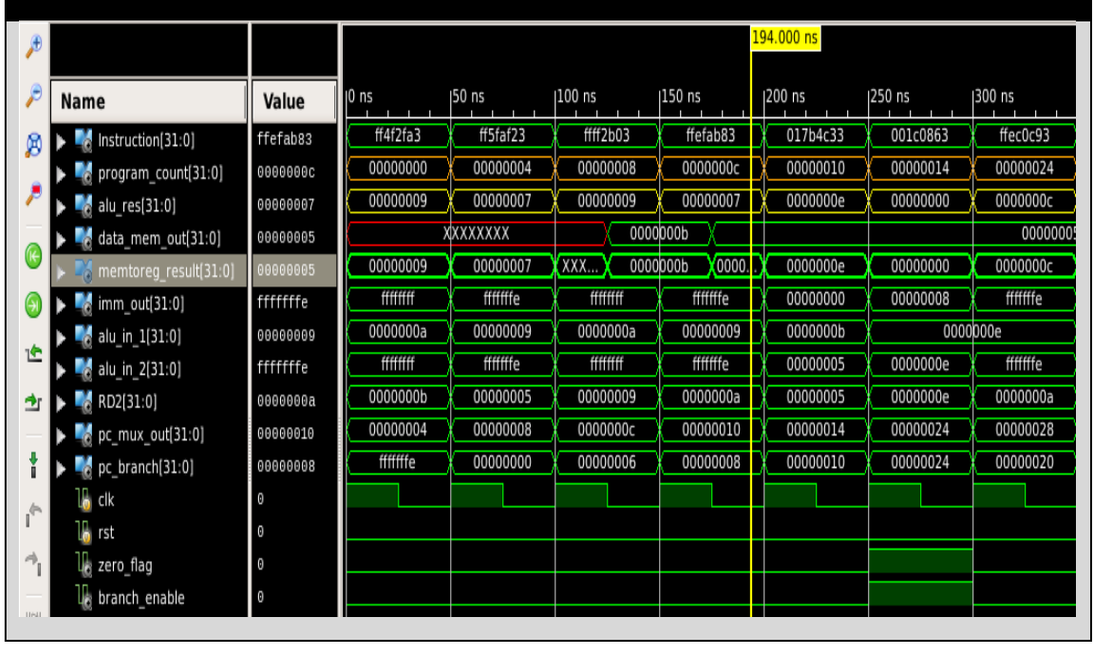
MemtoReg Multiplexer

Component Description	Component Photo
This multiplexer decides what values it will feed to the destination register on the register file.	

ALUSource Multiplexer	
Component Description	Component Photo
This multiplexer decides what values it will feed to the 2nd input of the Arithmetic Logic Unit (ALU).	

THE INSTRUCTIONS TABLE			
ASSUMED VALUES: **x30** = 0xA; **x20** = 0xB; **x31** = 0x9; **x21** = 0x5; **x1** = 0xE			
Count	**Abstract Code**	**Instruction**	**Interpretation**
0x0000	sw x20, -1(x30)	FF 4F 2F A3	@Data Memory: (0x9 = 0xB)
0x0004	sw x21, -2(x31)	FF 5F AF 23	@Data Memory: (0x7 = 0x5)
0x0008	lw x22, -1(x30)	FF FF 2B 03	@Register File: (x22 = 0xB)
0x000C	lw x23, -2(x31)	FF EF AB 83	@Register File: (x23 = 0x5)
0x0010	xor x24, x22, x23	01 7B 4C 33	1011_2 xor 0101_2 = 1110_2 = 0xE
0x0014	beq x24, x1, 8	00 1C 08 63	BTA = 0x0014 + (8_{10} * 2) = 0x0024
Skips (0x0018 – 0x0023)			
0x0024	addi x25, x24, -2	FF EC 0C 93	0xE + (-2_{10}) = 0xC

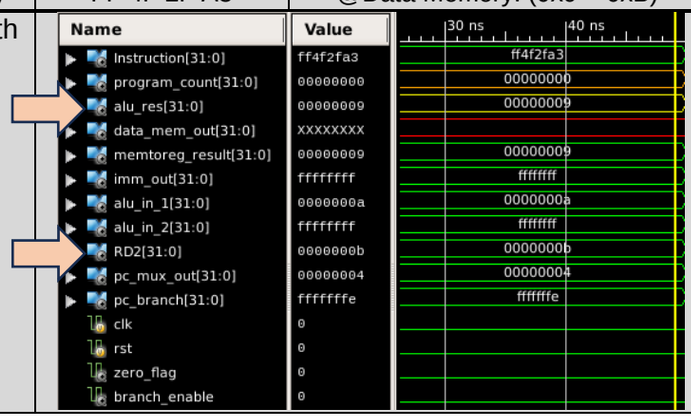
Test Bench Result



Count	Abstract Code	Instruction	Interpretation
0x0000	sw x20, -1(x30)	FF 4F 2F A3	@Data Memory: (0x9 = 0xB)

The count starts from 0x0 with instruction code FF4F2FA3.

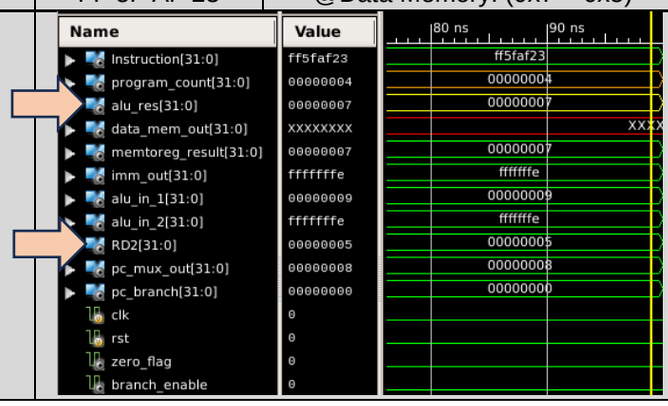
The calculated effective address of the ALU is stored with 0xB.



Count	Abstract Code	Instruction	Interpretation
0x0004	sw x21, -2(x31)	FF 5F AF 23	@Data Memory: (0x7 = 0x5)

The count now is 0x4 with instruction code FF5FAF23.

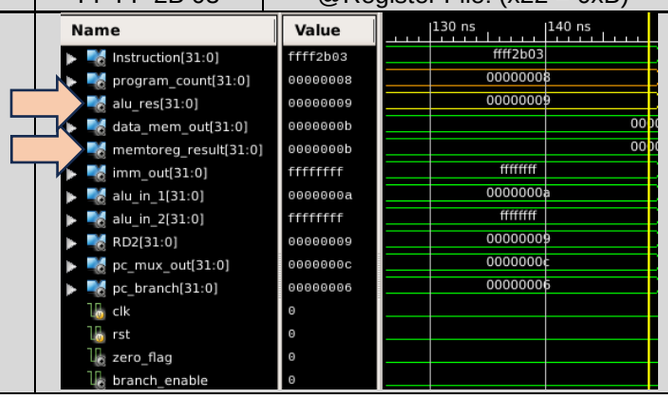
The calculated effective address of the ALU is stored with 0x5.



Count	Abstract Code	Instruction	Interpretation
0x0008	lw x22, -1(x30)	FF FF 2B 03	@Register File: (x22 = 0xB)

The count now is 0x8 with instruction code FFFF2B03.

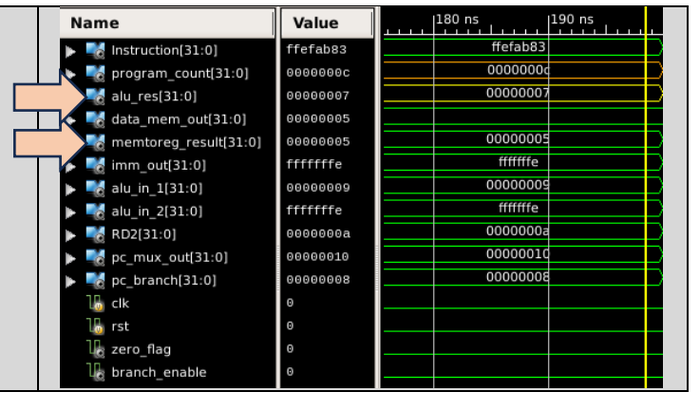
The value of the calculated effective address (0x9) in the data memory is loaded to the register file (x22 = 0xB).



Count	Abstract Code	Instruction	Interpretation
0x000C	lw x23, -2(x31)	FF EF AB 83	@Register File: (x23 = 0x5)

The count now is 0xC with instruction code FFEFAB83.

The value of the calculated effective address (0x7) in the data memory is loaded to the register file (0x7 = 0x5).



Count	Abstract Code	Instruction	Interpretation
0x0010	xor x24, x22, x23	01 7B 4C 33	1011_2 xor 0101_2 = 1110_2 = 0xE

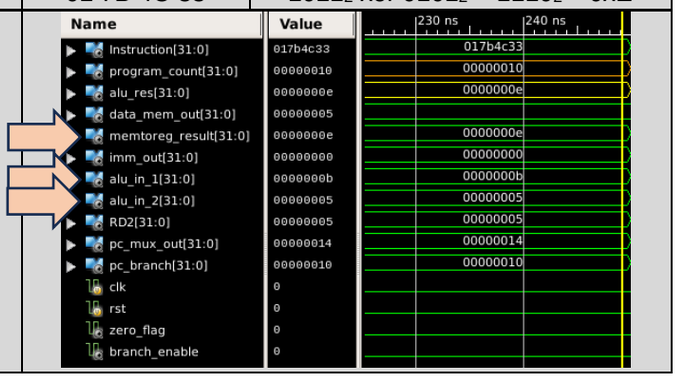
The count now is 0x10 with instruction code 017B4C33.

The value of the of x22 and x23 are with xored.

xor x24, x22, x23

0xE = 0xB xor 0x5

$$\begin{array}{r} 1011 \\ 0101 \\ \hline 1110_2 \end{array}$$

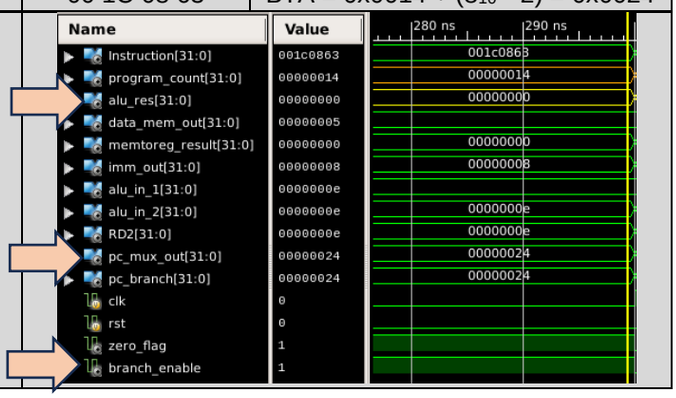


Count	Abstract Code	Instruction	Interpretation
0x0014	beq x24, x1, 8	00 1C 08 63	BTA = 0x0014 + (8_{10} * 2) = 0x0024

The count now is 0x14 with instruction code 001C0863.

The Branch is enabled. Program Counter Multiplexer outputs 0x24 as the next Program Count

BTA = 0x0014 + (810 * 2) = 0x0024

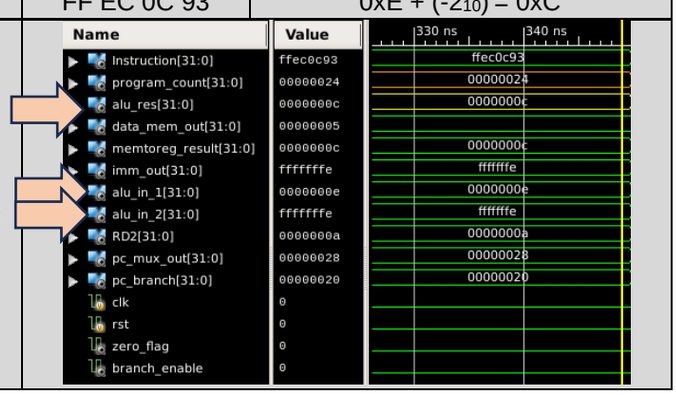


Count	Abstract Code	Instruction	Interpretation
0x0024	addi x25, x24, -2	FF EC 0C 93	0xE + (-2_{10}) = 0xC

The count now is 0x24 after branching with instruction code FFEC0C93

The x25 of register file is stored with the addition of stored value in x24 and the immediate -2.

0xE + (-2_{10}) = 0xC



Findings, Observations, and Comments

1. **Verilog Code to Schematic Approach**
   One of the key learning points during this project was the decision to use Verilog code to generate the schematic rather than traditional schematic development. Utilizing Xilinx's ability to convert Verilog code into schematics made the design process more efficient and reduced the complexity of manually creating connections between components.

2. **System First, Control Unit Later**
   A significant learning moment was realizing the value of first developing the system without the control unit. By initially focusing on building and testing the core components with a test bench using basic control signals, we could verify that the essential data flow and logic worked as expected. This incremental approach allowed us to avoid issues that might arise from having an incomplete or incorrect control unit, thus saving time in troubleshooting later on.

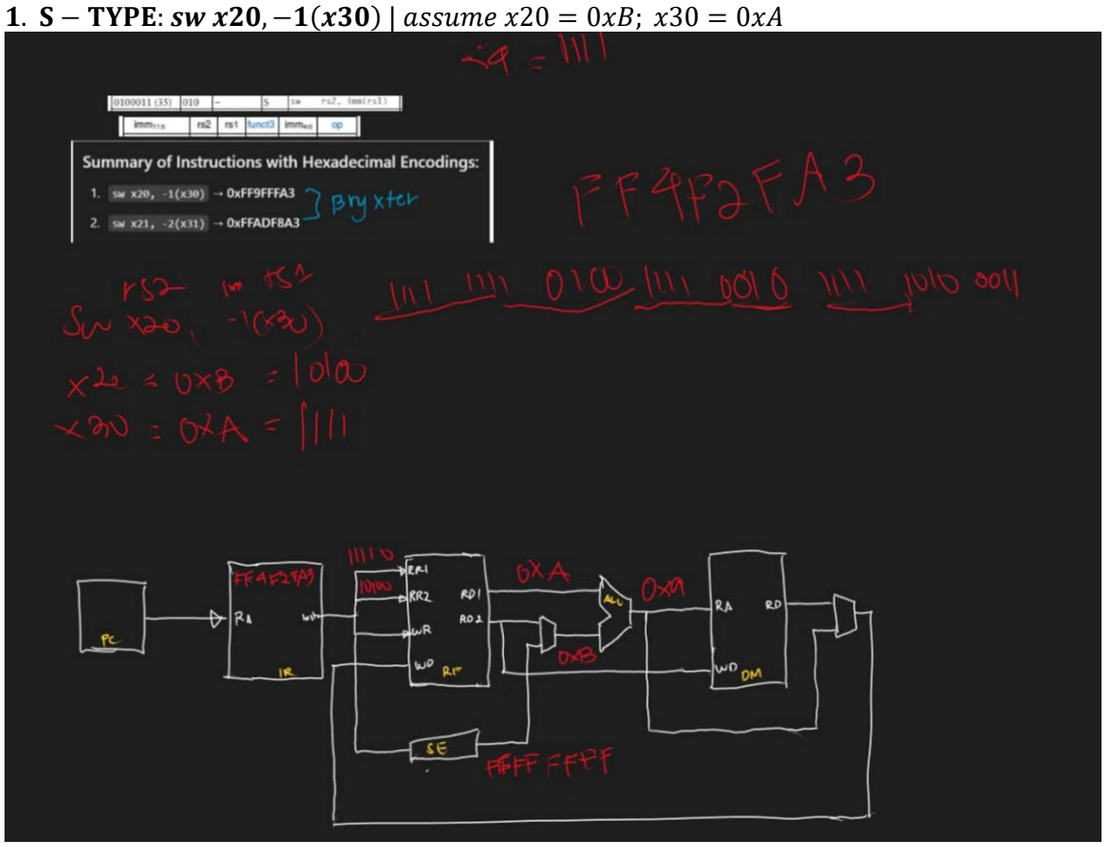
3. **Xilinx Virtual Machine Memory Constraints**
   While using Xilinx on a virtual machine, we encountered several memory limitations. When making changes, the group learned that it's crucial to regenerate the schematic symbol, close the application, and reopen it to ensure the updates are properly reflected. This workaround ensured that the system could handle memory-intensive tasks while working within the constraints of the virtual environment. Proper memory management in Xilinx became an essential step especially when dealing with larger projects like the SAP computer.

4. **Learning Concepts Along the Way**
   One of the more interesting takeaways from the project was how many concepts become clear as the project progresses. Concepts like instruction types, branching, ALU operations, and control signals were better understood through hands-on implementation. This kind of learning, which occurs while working directly with code and hardware descriptions, deepens the understanding of how the SAP computer operates and how instructions flow through the system.

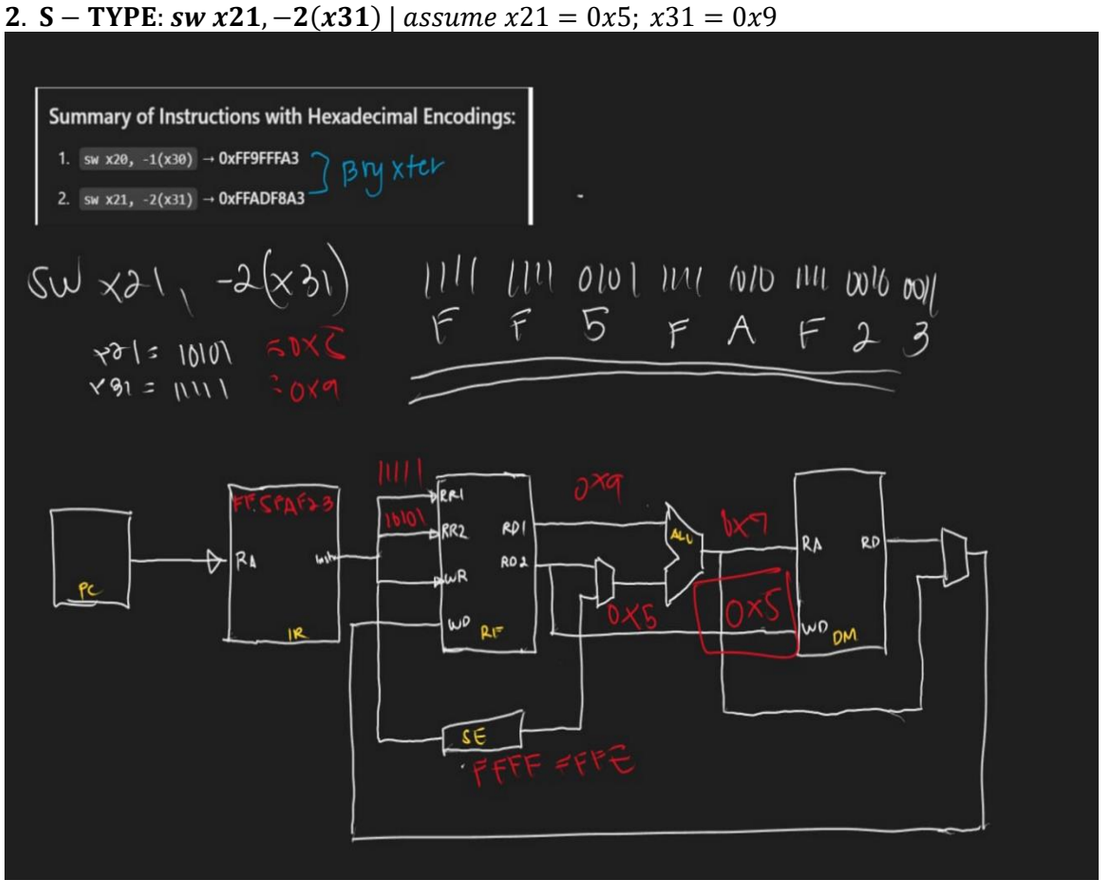
5. **RISB Type Instruction Support**
   The project was limited to supporting R, I, S, and B instruction types from the RISC-V architecture. This gave us a focused view on how these instructions interact with the SAP computer and how they are decoded and processed. It also provided a solid foundation for expanding support to other instruction types in future iterations of the project. Through this, we also understood how branching instructions behave in relation to the program counter and how immediate values affect execution.

APPENDICES

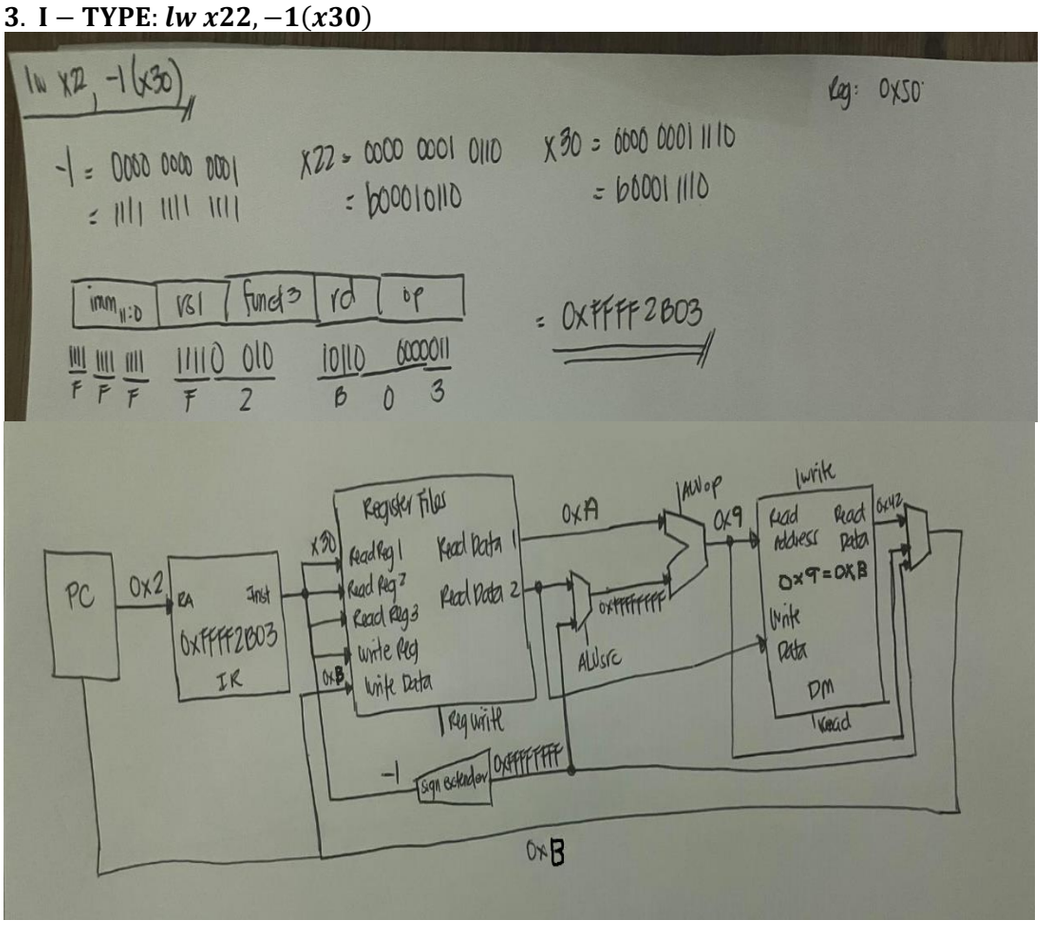
1. S − TYPE: *sw x20, −1(x30)* | *assume* $x20 = 0xB;\ x30 = 0xA$



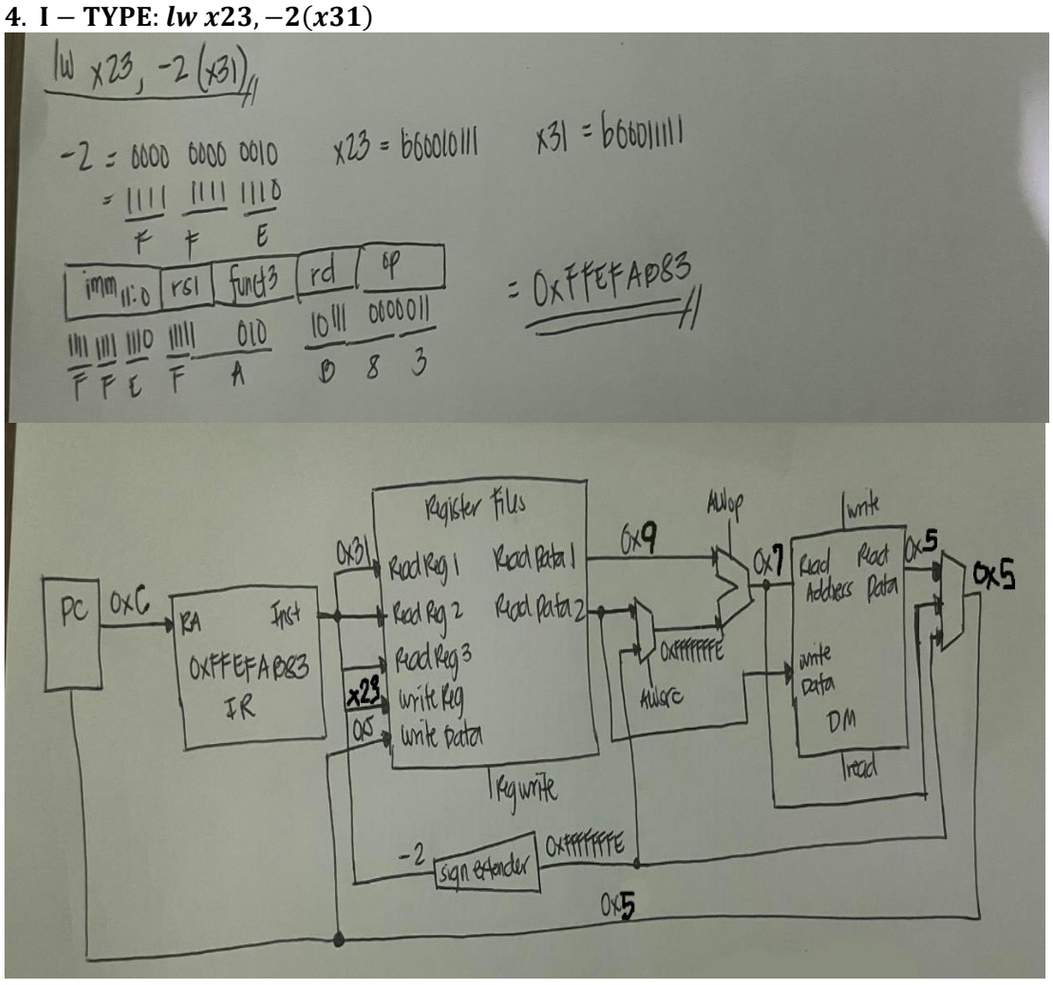
2. S − TYPE: *sw x21, −2(x31)* | *assume* $x21 = 0x5;\ x31 = 0x9$



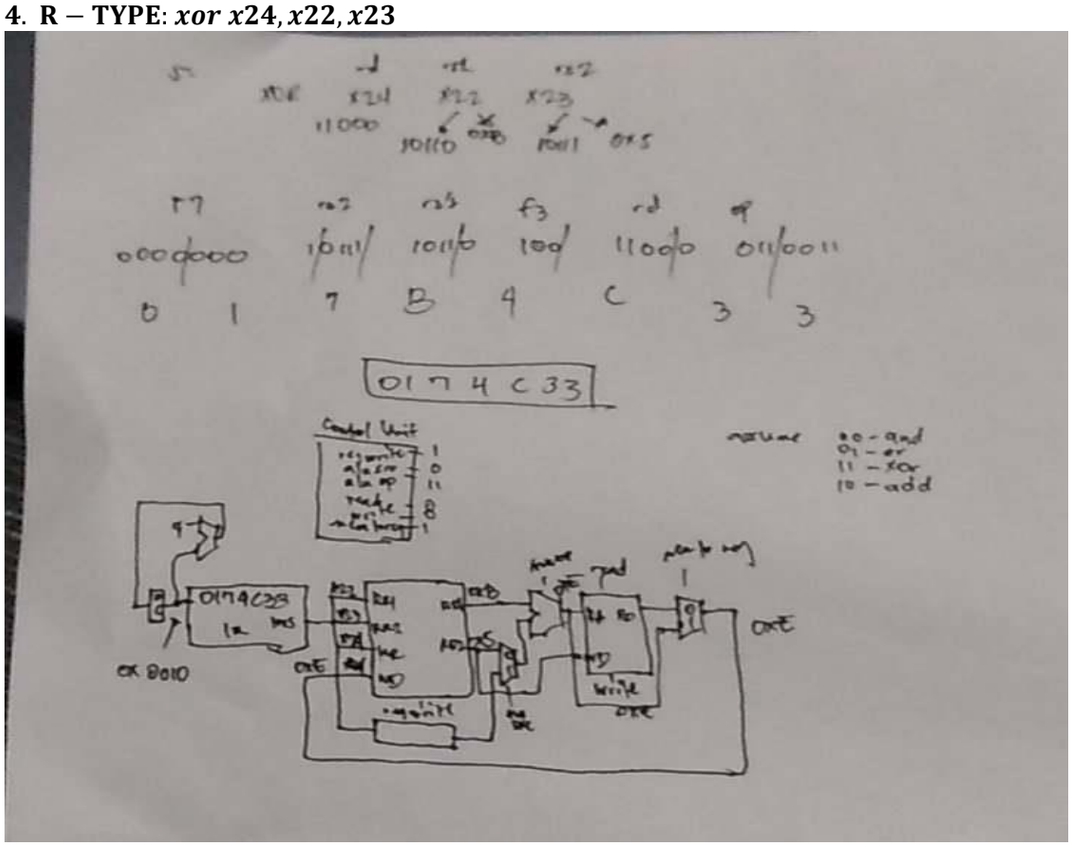
3. I – TYPE: $lw\ x22, -1(x30)$



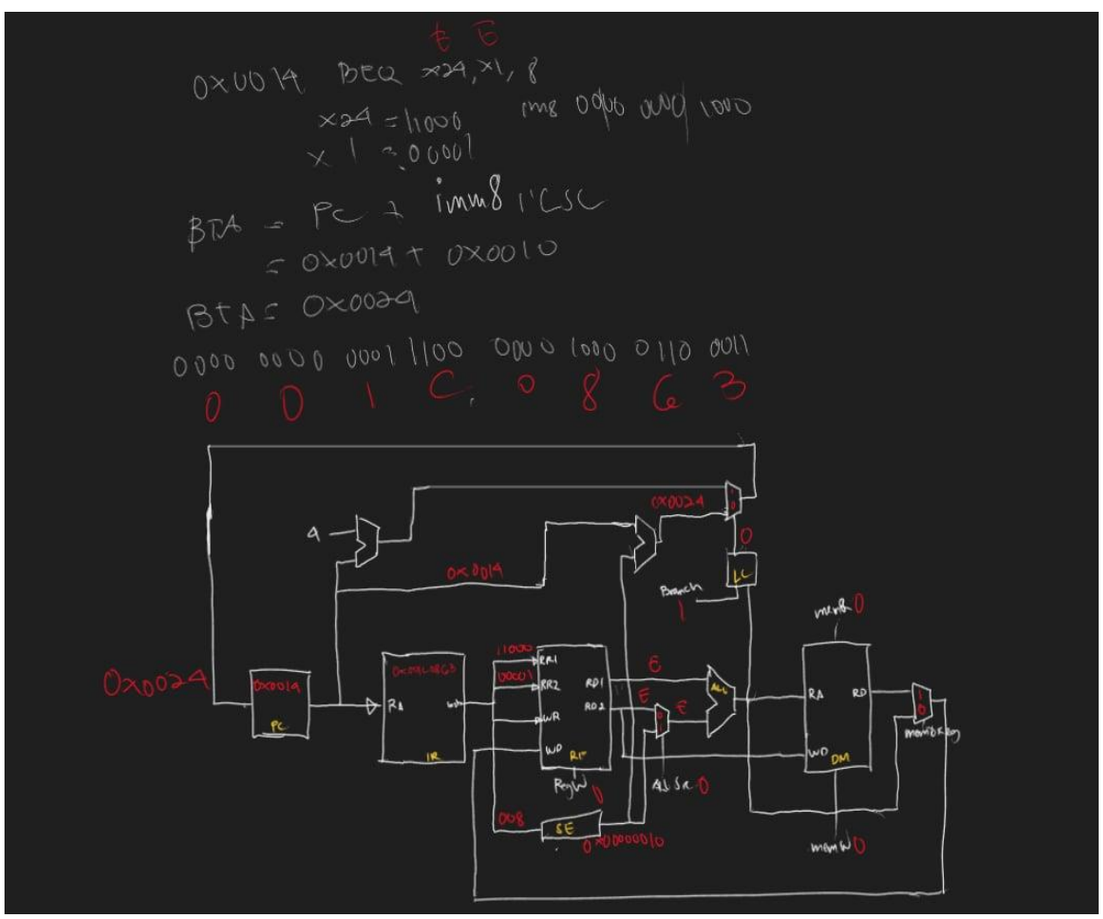
4. I – TYPE: $lw\ x23, -2(x31)$



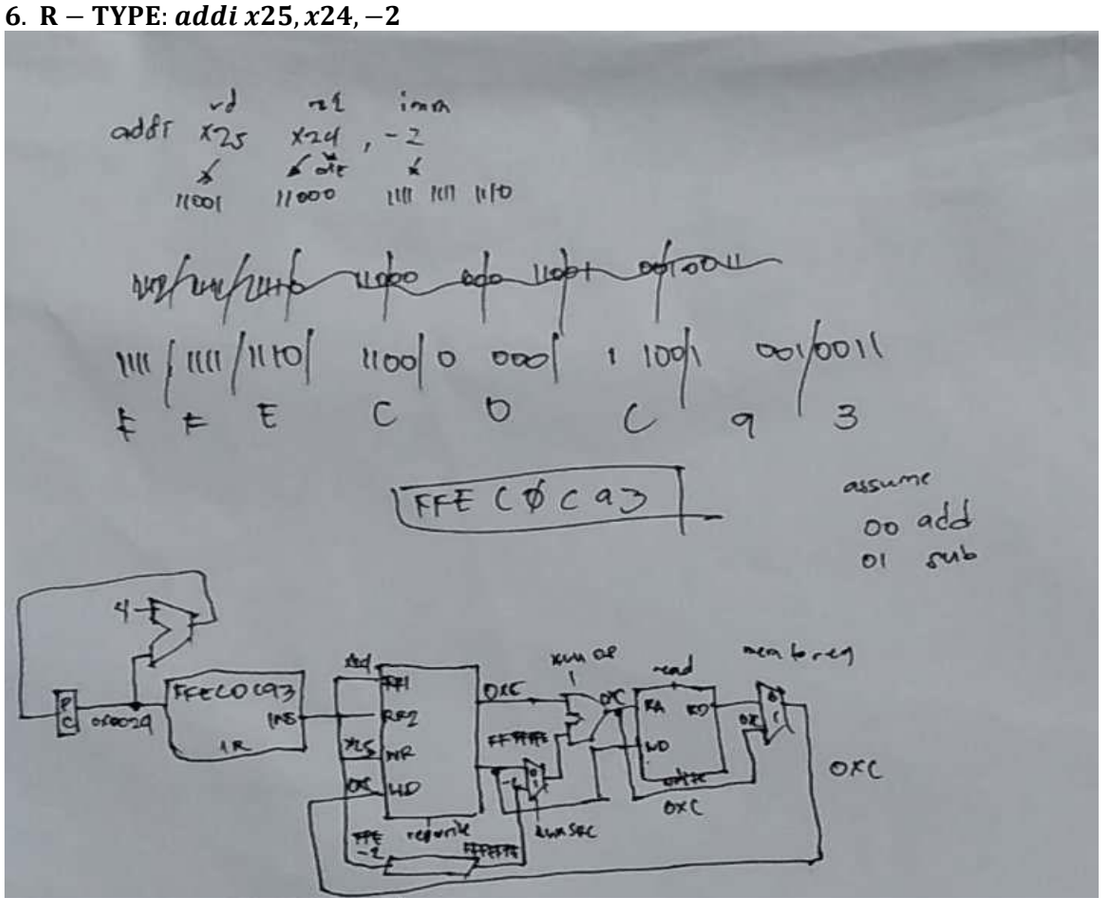
4. R – TYPE: $xor\ x24, x22, x23$



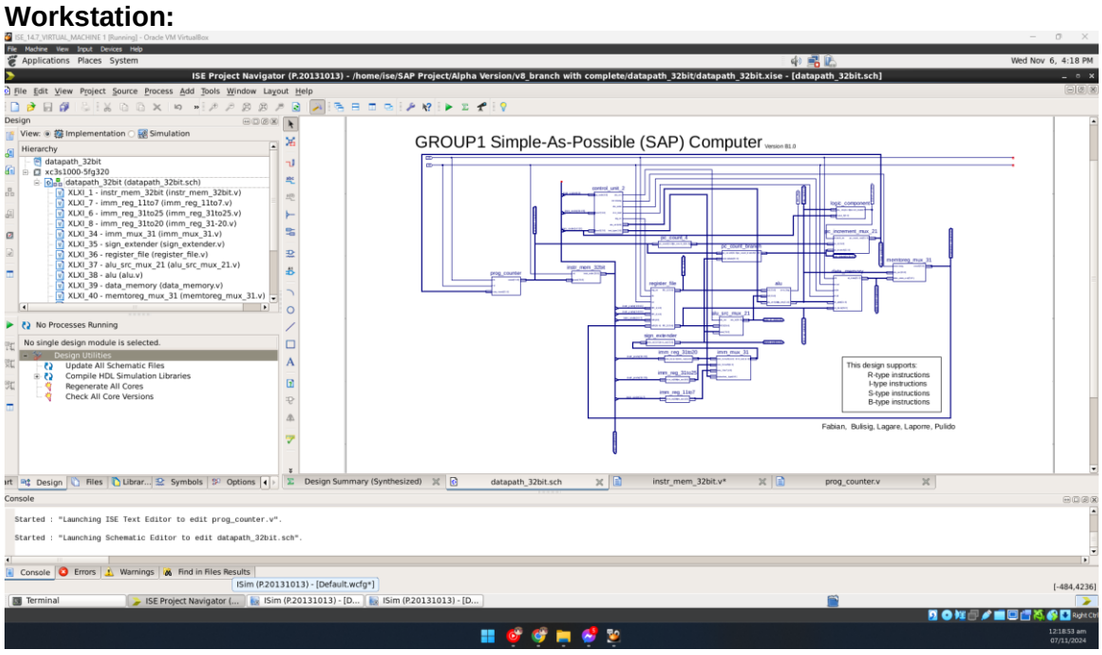
5. B – TYPE: $beq\ x24, x1, 8$



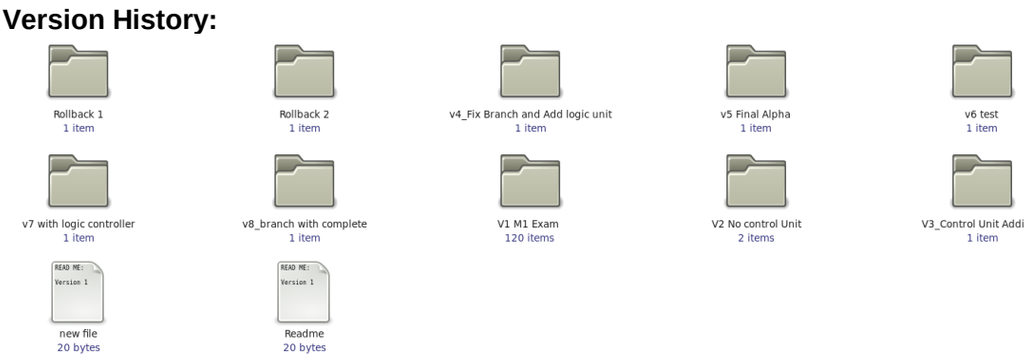
6. R − TYPE: $addi\ x25, x24, -2$



Workstation:



Version History:



REFERENCES

[1] “AMD-Downloads,” *AMD*, 2024. https://www.xilinx.com/support/download.html (accessed Nov. 07, 2024).

[2] “Designing and Implementing a SAP-1 Computer,” *SAP-1-Computer*, 2024. https://karenok.github.io/SAP-1-Computer/ (accessed Nov. 07, 2024).

[3] Codecademy, “The Instruction Cycle,” *Codecademy*, 2024. https://www.codecademy.com/article/the-instruction-cycle (accessed Nov. 07, 2024).