Lab 4 – RISC-V Processor Design

As discussed, a RISC V instruction normally goes through different phases starting with the instruction fetch phase. We will implement the necessary building blocks to perform the actions required at each phase.

Instruction Memory

Instruction memory can be viewed as a read-only memory buffer with address inputs and data outputs. Since we are following the RV 32I instruction set, all the instructions are encoded using 32-bit machine codes. As a result the data bus width will be 32-bits. The address bus size depends on the size of the memory, however the addresses generated by the processor ALU will be 32-bits.

The following code example illustrates the instantiation of instruction memory and its initialization using the user provided file.

```
// Instruction memory instantiation and initialization XLEN is 32
logic   [`XLEN-1:0]          inst_memory[`IMEM_SIZE];

initial
begin
     // Reading the contents of imem.txt file to memory variable
     $readmemh("imem.txt", inst_memory);
end
```

Listing 4.1. Instruction memory instantiation and initialization.

Task: The above implementation makes the instruction memory word-addressable. How would you modify it to byte-addressable memory? What will be the impact on the memory size?

Fetch Phase (F)

An instruction is fetched by providing an appropriate address to the instruction memory. The sample code that can perform this task is listed below.

```
// Asynchronous read operation of instruction memory
assign inst_machine_code = inst_memory[address];
```

Listing 4.2. Reading instruction memory.

Decode Phase (D)

The three key operations performed at this stage are:

1) Generating the control signals
2) Preparing the immediate values
3) Fetching the operands from the register file

Generating the Control Signals – The Decoder

The decoder is responsible to decode the machine code of an instruction and

```
case (instr_opcode)
    OPCODE_ARITH_INST  : begin

        id2exe_ctrl.alu_opr1_sel     = ALU_OPR1_REG;
        id2exe_ctrl.alu_opr2_sel     = ALU_OPR2_REG;

                      ...

        case (funct7_opcode)
            7'b0000000 : begin
                case (funct3_opcode)
                   3'b000 : id2exe_ctrl.alu_ops = ALU_OPS_ADD; // ADD
                   3'b001 : id2exe_ctrl.alu_ops = ALU_OPS_SLL; // Shift left logical
                        ...
```

Listing 4.3. Reading instruction memory.

Register File

A register file consists of thirty-two 32-bit registers which can be read and written by supplying the address of the registers. The register file is going to have two read ports and one write port. So the register file is going to take three addresses i.e two registers to be read and one write register along with the control signal and the data to be written for the write operation at the input and it is going to provide the value of the read operation at the output.

Following figure shows the block diagram of the register file.

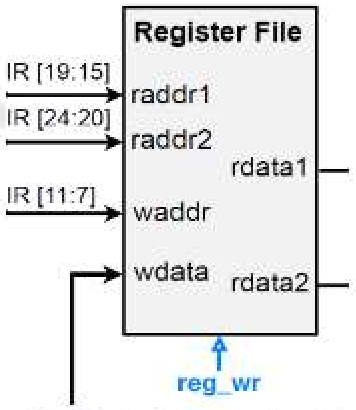

Figure 4.1. Block diagram of register file.

For designing the register file first of all we are going to create the header file which is going to define the parameters for the size of the register file, the width of the register file and the size of the registers inside the register file.

```
// UETRV_PCore_defs.svh
`ifndef UETRV_PCORE_SVH
`define UETRV_PCORE_SVH
```

```
//============================ CORE PARAMETERS =======================//

// Width of main registers and buses
`define XLEN               32

`define RF_AWIDTH           5
`define RF_SIZE            32

`endif // UETRV_PCORE_SVH
```

Listing 4.4. Header File Defining the core parameters

The register file is going to be instantiated as a multidimensional array along with the local signals which are going to check the validity of the input addresses for the read operation and the write operations. Asynchronous read operation for two register operands. The write operation is going to be performed based on the valid signal for the write operation.

```
// register file instantiation
logic   [`XLEN-1:0]          register_file[`RF_SIZE];

// control signals for validity of register file read/write operations
assign  rs1_addr_valid   = |id2rf_rs1_addr_i;
assign  rs2_addr_valid   = |id2rf_rs2_addr_i;
assign  rf_wr_valid      = |id2rf_rd_addr_i & id2rf_rd_wr_req_i;

// asynchronous read operation for two register operands
assign  rf2id_rs1_data_o = ( rs1_addr_valid )
                         ? register_file[id2rf_rs1_addr_i]
                         : '0;
assign  rf2id_rs2_data_o = (rs2_addr_valid)
                         ? register_file[id2rf_rs2_addr_i]
                         : '0;
```

Listing 4.5. Register file instantiation and asynchronous read operation.

```
// write operation
always_ff @( posedge clk) begin
    if ( rst_n ) begin
        register_file <= '{default: '0};
    end else if ( rf_wr_valid ) begin
        register_file[id2rf_rd_addr_i] <= id2rf_rd_data_i;
    end
end

endmodule : reg_file
```

Listing 4.6. Register file synchronous write operation.

Tasks

- Perform the verification of the register file code shared in the manual by initializing the register file with a memory file using the `$readmemh` command. Use of `$readmemh` has been explained in the context of instruction memory.

Execution Phase (E) – The ALU

The required operation by the instruction is performed in the execution phase. The first step is to prepare the operands followed by the implementation of the operation to be performed by the ALU. The Listing 4.7 illustrates the preparation of the operands, Listing 4.8 shows the ALU operation implementation.

```
// Prepare the two operands
always_comb begin
   alu_operand_1 = (id2exe_ctrl.alu_opr1_sel  ==   ALU_OPR1_PC)
                 ? (id2exe_data.pc)                     // Operand 1 is PC
                 : (id2exe_data.rs1_data);              // Operand 1 is register

   alu_operand_2 = (id2exe_ctrl.alu_opr2_sel  ==   ALU_OPR2_IMM)
                 ? (id2exe_data.imm)                    // Operand 2 is immediate
                 : (id2exe_data.rs2_data);              // Operand 2 is register
end
                     ...
```

Listing 4.7. Preparing the operands for the ALU.

```
always_comb begin
   exe2mem_alu_result   =  '0;
   case (alu_operator)
      ALU_OPS_AND : begin
         exe2mem_alu_result = alu_operand_1 & alu_operand_2;
      end
                           ...
```

Listing 4.8. Implementing the ALU operations.

Instruction	Operation
add rd, rs1, rs2	rd = rs1 + rs2
sub rd, rs1, rs2	rd = rs1 - rs2
sll rd, rs1, rs2	rd = rs1 << rs2[4:0]
slt rd, rs1, rs2	rd = $signed(rs1) < $signed(rs2)
sltu rd, rs1, rs2	rd = rs1 < rs2
xor rd, rs1, rs2	rd = rs1 ^ rs2
srl rd, rs1, rs2	rd = rs1 >> rs2[4:0]

sra rd, rs1, rs2	rd = rs1 >>> rs2[4:0]
or rd, rs1, rs2	rd = rs1 | rs2
and rd, rs1, rs2	rd = rs1 & rs2

Table 4.1. R-Type Instructions and operations.

Data Memory Access Phase (M)

For R-type instructions no memory access is required and as a result no activity is performed in this phase. We will discuss the data memory access phase and the required interface for this purpose in the next lab handout.

Writeback Phase (W)

During the write back phase the result of execution is written back to the destination register. The following code illustrates this step.

```
// Writeback MUX for output signal selection
always_comb begin
      wb2id_rd_data  = '0;
       case (mem2wb_ctrl.rd_wb_sel)
          RD_WB_ALU : begin
              wb2id_rd_data  =  mem2wb_data.alu_result;
          End

                    ...

          RD_WB_MEM : begin
              wb2id_rd_data  =  mem2wb_data.dmem_rdata;
          end
                                ...

          default :    wb2id_rd_data  = '0; // default case
      endcase
 end
```

Listing 4.9. The writeback operation to register file.

Tasks

- Implement the R-type instructions using the following datapath.

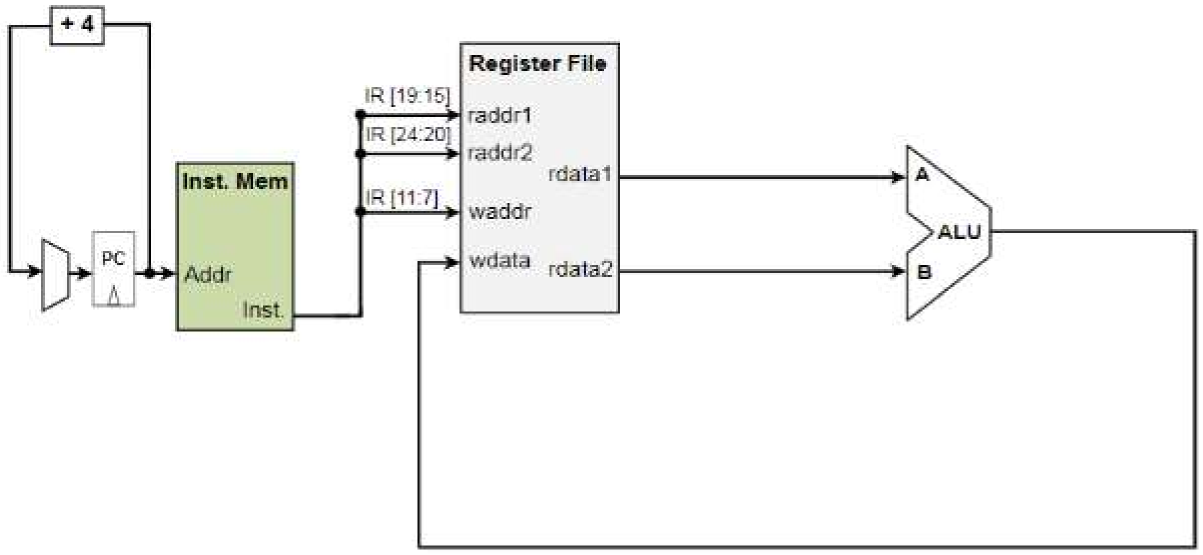

Figure 4.2. R-Type Instruction Data Path.

- Initialize the register file registers using $readmemh and write a simple assembly program that can test and verify the working of the implemented R-type instructions.

Lab 5 – Load/Store Operation and Data Memory Interface

The load/store operations are performed for data transfer from/to data memory. To implement these operations we first need to connect data memory with the processor data path at the data memory access phase. A simple tightly coupled data memory interface is shown in figure below.

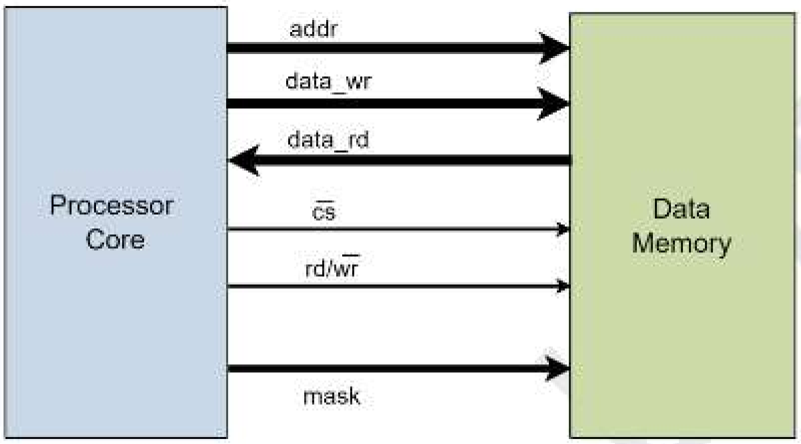

Figure 5.1. Data memory interface.

The data write operation is performed by setting up the address (**addr**), data to be written (**data_wr**) and **mask** followed by the **cs** and **wr** signals. The memory write operation is synchronous. The *mask* signal is used to inform the memory 1) the size of the data to be written, which can be of size byte, half-word or word and, 2) the location of the byte/half-word on a bus width equal to word size.

Load Operation – Memory Access Phase (M)

The code segment below illustrates setting up the signals during the data memory access phase when performing a load or store operation.

```
// Prepare the signals to perform load/store operations
assign st_ops        = |exe2mem_ctrl.mem_st_ops;
assign mem2dmem.addr = exe2mem_data.alu_result;
assign mem2dmem.cs   = ~(st_ops | (|exe2mem_ctrl.mem_ld_ops)); // cs is active low
assign mem2dmem.wr   = ~st_ops;  // Memory write/store is active low
```

Listing 5.1. Setting up the signals in the data memory access phase for load/store operation.

The data load operation corresponds to memory read and here for single cycle implementation we have used asynchronous memory read. The mask signal is of no significance because the load operation always receives 32-bit data from data memory as can be observed from the following code segment.

```
// Asynchronous read operation
```

```
assign dmem2mem.data_rd = ((~mem2dmem.cs) & (~mem2dmem.wr))
                          ? data_memory[addr_ff]
                          : '0;
```

Listing 5.2. Asynchronous data memory read for load operation.

The received 32-bit data is processed during the data memory access phase for the requested size and the corresponding address location.

```
// Extract the right size from the read data
always_comb begin
   dmem_rdata_byte  = '0;
   dmem_rdata_hword = '0;
   dmem_rdata_word  = '0;

   case (exe2mem_ctrl.mem_ld_ops)
      MEM_LD_OPS_LB,
      MEM_LD_OPS_LBU : begin
         case (mem2dmem.addr[1:0])
            2'b00 : begin
               dmem_rdata_byte = dmem2mem.data_rd[7:0];
            end
            2'b01 : begin
               dmem_rdata_byte = dmem2mem.data_rd[15:8];
            end
                          ...
            default : begin
            end
         endcase
      end // MEM_LD_OPS_LB, MEM_LD_OPS_LBU
      MEM_LD_OPS_LH,
      MEM_LD_OPS_LHU : begin
         case (mem2dmem.addr[1])
            1'b0 : begin
               dmem_rdata_hword = dmem2mem.data_rd[15:0];
            end
                    ...
         endcase
      end // MEM_LD_OPS_LH, MEM_LD_OPS_LHU
      MEM_LD_OPS_LW : begin
         dmem_rdata_word = dmem2mem.data_rd;
      end
      default : begin
      end
   endcase // mem_ld_ops
end
```

Listing 5.3. Asynchronous data memory read for load operation.

Next we need to perform either sign- or zero-extension of the received data, before it is put into the destination register, as illustrated below.

```
// Extend the load data for sign/zero
always_comb begin
    case (exe2mem_ctrl.mem_ld_ops)
        MEM_LD_OPS_LB  : mem2wb_data.dmem_rdata = {{24{dmem_rdata_byte[7]}},  dmem_rdata_byte};
        MEM_LD_OPS_LBU : mem2wb_data.dmem_rdata = { 24'b0,                    dmem_rdata_byte};
        MEM_LD_OPS_LH  : mem2wb_data.dmem_rdata = {{16{dmem_rdata_byte[15]}}, dmem_rdata_hword};
        MEM_LD_OPS_LHU : mem2wb_data.dmem_rdata = { 16'b0,                    dmem_rdata_hword};
        MEM_LD_OPS_LW  : mem2wb_data.dmem_rdata = {                           dmem_rdata_word};
        default        : mem2wb_data.dmem_rdata = '0;
    endcase // mem_ld_ops
end
```

Listing 5.4. Sign- or zero-extension of the loaded data.

Store Operation – Memory Access Phase (M)

Listing 5.1 illustrates setting up the signals for data memory access, which are equally applicable for store operation. In addition, the data that is to be stored and the corresponding mask are also prepared for the store operation, which resolves the data size as well as its address location. The code segment below illustrates this aspect of the data memory access.

```
// Prepare the write data and mask for store
always_comb begin
   mem2dmem.data_wr = '0;
   mem2dmem.mask    = '0;

   case (exe2mem_ctrl.mem_st_ops)
      MEM_ST_OPS_SB : begin
         case (mem2dmem.addr[1:0])
            2'b00 : begin
               mem2dmem.data_wr[7:0]   = exe2mem_data.rs2_data[7:0];
               mem2dmem.mask = 4'b0001;
            end
                        ...
            2'b11 : begin
               mem2dmem.data_wr[31:24] = exe2mem_data.rs2_data[31:24];
               mem2dmem.mask = 4'b1000;
            end
            default : begin
            end
         endcase
      end // MEM_ST_OPS_SB
      MEM_ST_OPS_SH : begin
         case (mem2dmem.addr[1])
            1'b0 : begin
```

```
                mem2dmem.data_wr[15:0]  = exe2mem_data.rs2_data[15:0];
                mem2dmem.mask = 4'b0011;
             end
                             ...
          endcase
       end // MEM_ST_OPS_SH
       MEM_ST_OPS_SW : begin
          mem2dmem.data_wr = exe2mem_data.rs2_data;
          mem2dmem.mask = 4'b1111;
       end
       default : begin
          mem2dmem.data_wr = '0;
       end
    endcase // mem_st_ops
end
```

Listing 5.5. Data and mask preparation for store operation.

Finally the data prepared during the memory phase for store operation is sent to the data memory for store operation and is illustrated in the below code segment.

```
// Memory store operation
always_ff @(negedge clk)
begin
   if ( !cs_ff && !wr_ff ) begin
        if (mask_ff[0])
                data_memory[addr_ff][7:0] = data_wr_ff[7:0];
        if (mask_ff[1])
                data_memory[addr_ff][15:8] = data_wr_ff[15:8];
        if (mask_ff[2])
                data_memory[addr_ff][23:16] = data_wr_ff[23:16];
        if (mask_ff[3])
                data_memory[addr_ff][31:24] = data_wr_ff[31:24];
   end
end
```

Listing 5.6. Storing the data synchronously to the data memory.

Load/Store Operation – Decode Phase

The decode operation for load and store instructions use **func3** bit-field of the instruction machine code to define the size and type of the data variable that is to be exchanged with the data memory. Recall that load instructions are I-type while store operations follow S-type encoding format. The memory address for load/store operations is constructed using register-offset memory addressing mode. The code listings implementing the load and store operations at instruction decode phase are given in Listing 5.7 and 5.8, respectively.

```
// Load operations
OPCODE_LOAD_INST : begin
    id2exe_ctrl.rd_wb_sel     = RD_WB_MEM;
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
    id2exe_ctrl.rd_wr_req    = 1'b1;
     case (funct3_opcode)
              3'b000  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LB;    // Load byte signed
              3'b001  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LH;    // Load halfword signed
              3'b010  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LW;    // Load word
              3'b100  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LBU;   // Load byte unsigned
              3'b101  : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_LHU;   // Load halfword unsigned
              default : id2exe_ctrl.mem_ld_ops = MEM_LD_OPS_NONE;  // Load type unknown
    endcase // funct3_opcode

end // OPCODE_LOAD_INST
```

Listing 5.7. Instruction decoding for load operation.

```
// Store operations
OPCODE_STORE_INST : begin
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
    id2exe_data.imm          = {{21{instr_codeword[31]}}, instr_codeword[30:25],
                                    instr_codeword[11:7]};
     case (funct3_opcode)
              3'b000  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SB;    // Store byte signed
              3'b001  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SH;    // Store halfword signed
              3'b010  : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_SW;    // Store word
              default : id2exe_ctrl.mem_st_ops = MEM_ST_OPS_NONE;  // Store word
    endcase // funct3_opcode

end // OPCODE_STORE_INST
```

Listing 5.8. Instruction decoding for store operation.

Load/Store Operation – Execute Phase

The address generation for load/store operations during the instruction execution phase simply uses the ADD operation for register-immediate offset addressing.

Lab 6 – Flow Control Instructions

The branch and jump instructions are used to transfer the flow of control from one part of the instruction memory to another part. The branch instructions transfer control based upon the comparison of the two register operands and if the condition is true then the control will be transferred to another part of the instruction. The jump instructions provide unconditional control transfer to the target. To implement these operations we are required to modify our datapath in such a way that we can use our ALU to perform the calculation of the address of the target to which the control will be transferred. This will be performed by adding the immediate value to the current value of the program counter. The immediate value will be calculated during the decode stage and the calculation of the address along with the comparison for the branch operation will be performed in the execute stage.

Branch/ Jump Operation – Fetch Phase

The value of PC for the next instruction will be changed in case the branch condition becomes true or the jump instruction is executed. Listing 6.1 demonstrates that the PC value is updated with the result of the ALU when the branch or jump is taken.

```
assign pc_next  = exe2if_fb.jump_br_taken ? exe2if_fb.alu_pc
                : imem_update_addr        ? (pc_ff + 32'd4)
                : pc_ff;
```

Listing 6.1. PC value updated for Branch/Jump operation.

Branch/ Jump Operation – Decode Phase

The decode operation for branch instructions uses the **func3** bit-field of the instruction machine code to select the operands, comparison operation to be performed, ALU operation to be performed, request for branch instruction and immediate value which are going to be used inside the execution stage. However, these selections for jump instructions are based on the opcode of the instruction. There are two jump instructions which include the jump and link (jal) instruction and the jump and link register (jalr) instructions. Recall that branch instructions are B-type while jump operations follow J-type encoding format. The target address for branch and jal operations is constructed using pc-offset addressing. However, for the jalr instruction register-offset addressing is used. The code listings implementing the load and store operations at instruction decode phase are given in Listing 6.2 and 6.3, respectively.

```
OPCODE_BRANCH_INST : begin
   id2exe_ctrl.alu_opr1_sel     = ALU_OPR1_PC;
   id2exe_ctrl.alu_opr2_sel     = ALU_OPR2_IMM;
   id2exe_ctrl.alu_cmp_opr2_sel = ALU_CMP_OPR2_REG;
   id2exe_ctrl.alu_ops          = ALU_OPS_ADD;
   id2exe_ctrl.branch_req       = 1'b1;
   id2exe_data.imm              = {{20{instr_codeword[31]}}, instr_codeword[7],
                                  instr_codeword[30:25], instr_codeword[11:8], 1'b0};

   case (funct3_opcode)
       3'b000  : id2exe_ctrl.branch_ops = BR_EQ;      // Branch equal
       3'b001  : id2exe_ctrl.branch_ops = BR_NE;      // Branch not equal
       3'b100  : id2exe_ctrl.branch_ops = BR_LT;      // Branch less than
       3'b101  : id2exe_ctrl.branch_ops = BR_GE;      // Branch greater than or equal signed
```

```
        3'b110  : id2exe_ctrl.branch_ops = BR_LTU;     // Branch less than unsigned
        3'b111  : id2exe_ctrl.branch_ops = BR_GEU;     // Branch greater than or equal unsigned
        default : id2exe_ctrl.branch_ops = BR_NONE;    // No branch
    endcase // funct3_opcode
    end // OPCODE_BRANCH_INST
```

Listing 6.2. Instruction decoding for Branch operation.

```
OPCODE_JALR_INST : begin
    id2exe_ctrl.rd_wb_sel    = RD_WB_INC_PC;
    id2exe_ctrl.alu_opr1_sel = ALU_OPR1_REG;
    id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
    id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
    id2exe_ctrl.rd_wr_req    = 1'b1;
    id2exe_ctrl.jump_req     = 1'b1;
    id2exe_data.imm           =   {{12{instr_codeword[31]}}, instr_codeword[19:12],
                                  instr_codeword[20], instr_codeword[30:21], 1'b0};

end // OPCODE_JALR_INST

// JAL operation
OPCODE_JAL_INST : begin
   id2exe_ctrl.rd_wb_sel    = RD_WB_INC_PC;
   id2exe_ctrl.alu_opr1_sel = ALU_OPR1_PC;
   id2exe_ctrl.alu_opr2_sel = ALU_OPR2_IMM;
   id2exe_ctrl.alu_ops      = ALU_OPS_ADD;
   id2exe_ctrl.rd_wr_req    = 1'b1;
   id2exe_ctrl.jump_req     = 1'b1;
   id2exe_data.imm          = {{12{instr_codeword[31]}}, instr_codeword[19:12],
                              instr_codeword[20], instr_codeword[30:21], 1'b0};

end // OPCODE_JAL_INST
```

Listing 6.3. Instruction decoding for Jump operation.

Branch/Jump Operation – Execute Phase

Branch instruction requires comparison between two operands. Listing 6.4 illustrates comparison based on the subtract operation which can be used to utilize different flags based on the result of the subtraction between two operations.

```
// Difference calculation for comparison operations (branch and set-less-then operations)
assign cmp_operand_1 = id2exu_data.rs1_data;
assign cmp_operand_2 = (id2exu_ctrl.alu_cmp_opr2_sel == ALU_CMP_OPR2_IMM)
                      ? id2exu_data.imm
                      : id2exu_data.rs2_data;

assign cmp_output    = {1'b0, cmp_operand_1} - {1'b0, cmp_operand_2};
assign cmp_not_zero  = |cmp_output[`XLEN-1:0];
assign cmp_neg       = cmp_output[`XLEN-1];
assign cmp_overflow  = (cmp_neg & ~cmp_operand_1[`XLEN-1] & cmp_operand_2[`XLEN-1]) |
                       (~cmp_neg & cmp_operand_1[`XLEN-1] & ~cmp_operand_2[`XLEN-1]);
```

Listing 6.4. Comparison performed for branch operation.

There are different branch operations available in the RV 32I and based on the opcode of the type of branch the execution unit will use the comparison flags for checking different branch conditions. Listing 6.5 illustrates the use of comparison flags for different branch instructions.

```
// Evaluate the branch comparison result
always_comb begin
  // set comparison by default
  branch_res = 1'b0;
  case (id2exu_ctrl.branch_ops)
     BR_EQ:   branch_res = ~cmp_not_zero;
     BR_NE:   branch_res = cmp_not_zero;
     BR_LT:   branch_res = (cmp_neg ^ cmp_overflow);
     BR_LTU:  branch_res = cmp_output[`XLEN];          // Check if the carry-flag bit is set
     BR_GE:   branch_res = ~(cmp_neg ^ cmp_overflow);
     BR_GEU:  branch_res = ~cmp_output[`XLEN];         // Carry flag bit is cleared
     default: branch_res = 1'b0;
  endcase
end
```

Listing 6.5. Comparison flags used for branch condition check.

The jump instructions write their return address to the destination register. Listing 6.6 shows that the execute stage will update the value of PC in the fetch stage when the branch instruction is executed and the branch condition is true or the jump instruction is executed.

```
// Feedback signals from EXE to IF stage
assign exe2if_fb.jump_br_taken   = id2exu_ctrl.jump_req || (id2exu_ctrl.branch_req &
                                   branch_res);
assign exe2if_fb.alu_pc          = exe2mem_alu_result;
assign exe2if_fb_o               = exe2if_fb;
```

Listing 6.6. Feedback signals from execution to fetch stage.