Department of Electronic and Telecommunication Engineering
University of Moratuwa

Stage 1 - Individual Project

Non-pipelined Single Stage (Cycle) CPU Design

THILAKARATHNE D.L.J. **200650U**

This report is submitted as a partial fulfilment of module EN3021 - Digital System Design

October 16, 2023

Abstract

This project involves the design and implementation of a 32-bit non-pipelined RISC-V processor using Micro-programming with a 3-bus structure, specifically focusing on the RV32I instruction set. The task entails the inclusion of three classes of instructions: computational (R and I types), memory access (I and S types), and control flow (SB type). Additionally, two new instructions, MEMCOPY and MUL, are to be integrated, with MEMCOPY capable of copying an array of size N (with a constraint of N ¿ 1), and MUL intended for unsigned multiplication. This is a comprehensive report encompassing resource utilization.

1 Background

The RISC-V architecture is a reduced instruction set computer (RISC) instruction set architecture (ISA). RISC-V ISAs are based on the principle that a small number of simple instructions can be combined to perform any computing task. This approach makes RISC-V processors simpler to design and implement, and it can also lead to performance improvements.

Micro-programming is a technique for implementing a computer's ISA using a smaller set of more basic instructions. This is done by storing the micro-instructions in a control memory, which is read by the processor to determine what to do next. Micro-programming allows for greater flexibility in the design of the processor, but it can also lead to a decrease in performance.

The 3-bus structure is a common bus architecture for micro-programmed processors. In this architecture, there are three separate buses: the address bus, the data bus, and the control bus. The address bus is used to address memory locations, the data bus is used to transfer data between the processor and memory, and the control bus is used to transmit control signals between the processor and other components of the system.

The RISC-V architecture is an open-source ISA, which means that anyone can design and implement a RISC-V processor without having to pay royalties. This has led to a growing ecosystem of RISC-V hardware and software.

RISC-V processors are being used in a wide range of applications, from embedded systems to supercomputers. RISC-V is also being used in research projects to develop new types of computer architectures.

2 Design Overview

The 32-bit non-pipelined RISC-V processor was developed with a primary focus on achieving efficient execution and simplicity in design. The architecture leveraged a Micro-programming approach, enabling a systematic and structured implementation of the RV32I instruction set. The processor was equipped with a 3-bus structure to facilitate efficient data transfer and processing. Initial efforts were dedicated to the successful integration of the computational instructions, covering both the R and I types, along with the essential memory access instructions, encompassing the I and S types. The control flow instructions, specifically of the SB type, were seamlessly incorporated to ensure robust program control within the processor.

The design emphasized a balance between computational power and resource utilization, ensuring a streamlined execution of instructions while maintaining optimal hardware usage. Leveraging the foundational elements of the RV32I instruction set, the processor was able to perform fundamental arithmetic and logical operations efficiently. The instruction decoding and execution were meticulously structured to minimize latency and maximize throughput, thereby enabling swift processing of complex instructions.

In the subsequent phase of the design, emphasis was placed on integrating two new instructions, MEMCOPY and MUL, to enhance the processor's functionality. MEMCOPY was tailored to efficiently copy arrays of variable sizes while adhering to predefined constraints to optimize memory usage. On the other hand, the MUL instruction was introduced to facilitate unsigned multiplication, thereby expanding the processor's arithmetic capabilities beyond the limitations of the original RV32I instruction set. This design modification aimed to enhance the processor's versatility and utility, catering to a broader range of computational requirements.

3 Implementation

The implementation phase involved the utilization of System Verilog, a hardware description language (HDL), to construct the various modules and components of the processor design. Initially, Intel Quartus Prime was employed as the primary software tool for RTL development and synthesis. However, to leverage more advanced features and achieve a streamlined development process, the decision was made to transition to Vivado.

Vivado's user-friendly interface and intelligent design capabilities significantly expedited the implementation process, allowing for smoother integration of the new instructions and efficient debugging of any potential issues. The tool's robust synthesis and optimization features facilitated the effective utilization of hardware resources, ensuring optimal performance of the final processor design. Through the systematic use of Vivado, the processor's architecture was meticulously translated into synthesizable RTL files, ready for further analysis and verification.

3.1 Modelling

In the initial stages of the project, the focus was on the fundamental structural modelling of the system, starting with the implementation of a Prefix Adder, deemed the optimal choice for a 32-bit scenario. However, during the course of the implementation process, it became evident that Vivado, with its intelligent features, could automatically select the most suitable option if behavioral modelling was adopted. As a result, the decision was made to shift to behavioral modelling to leverage Vivado's advanced capabilities and streamline the implementation process(ML based logic optimization). This transition allowed for a more efficient and optimized design approach, ultimately contributing to the enhanced performance and functionality of the processor.

4 Testbenches

5 MUL

Overflow can occur when multiplying two binary, 32-bit numbers even if the output is also a 32-bit number. In binary arithmetic, overflow happens when the result of an operation cannot be represented within the allotted number of bits. In your case, both the input numbers and the output are 32-bit, but it's still possible for overflow to occur if the product of the two numbers exceeds the maximum positive or minimum negative value that can be represented in a 32-bit signed binary representation.

Here's how overflow can occur during multiplication:

Magnitude Overflow: This occurs when the magnitude of the product exceeds the maximum positive value that can be represented in a 32-bit signed integer. In a 32-bit signed representation, the maximum positive value is $2^31 - 1$, and the minimum negative value is -2^31.

For example, if you multiply two 32-bit binary numbers and the result is a binary number that has a magnitude greater than 2^31-1 or less than -2^31, you have an overflow.

Carry Overflow: Overflow can also occur when there's a carry bit generated during multiplication that cannot be accommodated in the 32-bit output. This can happen when the most significant bits of the multiplication result produce a carry into a bit position that is beyond the 32-bit limit.

For example, consider the multiplication of two 32-bit binary numbers where both inputs are 32-bit maximum positive values (i.e., $2 \cdot 31-1$). When you multiply them, the result will be approximately

$$(2^31 - 1) * (2^31 - 1)$$

, which will be much larger than $2^31 - 1$, and this will cause a magnitude overflow.

It's important to note that in some computer architectures, the overflow condition is checked, and the result may be treated as undefined or wrapped around (result modulo 2^32) or generate an exception, depending on the hardware and software implementation.

To prevent overflow, you should always be aware of the maximum and minimum values that can be represented in the given number of bits and ensure that the result of multiplication stays within these limits. If overflow is a possibility, you may need to use a larger data type or implement additional checks and error-handling mechanisms.

6 Resource Utilization

```
Copyright 1986-2022 Xilinx, Inc. All Rights Reserved. Copyright 2022-2023 Advanced Micro Devices, Inc. All
-----------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2023.1 (win64) Build 3865809 Sun May  7 15:05:29 MDT 2023
| Date         : Mon Oct 16 21:39:34 2023
| Host         : DESKTOP-G3EKPRB running 64-bit major release  (build 9200)
| Command      : report_utilization -file processor_utilization_synth.rpt -pb processor_utilization_synth.p
| Design       : processor
| Device       : xc7a35tcpg236-1
| Speed File   : -1
| Design State : Synthesized
-----------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |    0 |     0 |          0 |     20800 |  0.00 |
|   LUT as Logic          |    0 |     0 |          0 |     20800 |  0.00 |
|   LUT as Memory         |    0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         |    0 |     0 |          0 |     41600 |  0.00 |
|   Register as Flip Flop |    0 |     0 |          0 |     41600 |  0.00 |
|   Register as Latch     |    0 |     0 |          0 |     41600 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     16300 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |      8150 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower.
Warning! LUT value is adjusted to account for LUT combining.


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
| 0     |          Yes |           - |        Reset |
| 0      |           Yes |         Set |            - |
| 0      |           Yes |       Reset |            - |
+--------+--------------+-------------+--------------+


2. Memory
---------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| Block RAM Tile |    0 |     0 |          0 |        50 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |          0 |        50 |  0.00 |
|   RAMB18       |    0 |     0 |          0 |       100 |  0.00 |
+----------------+------+-------+------------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36


3. DSP
------

+-----------+------+-------+------------+-----------+-------+
| Site Type | Used | Fixed | Prohibited | Available | Util% |
+-----------+------+-------+------------+-----------+-------+
| DSPs      |    0 |     0 |          0 |        90 |  0.00 |
+-----------+------+-------+------------+-----------+-------+


4. IO and GT Specific
---------------------

+-----------------------------+------+-------+------------+-----------+-------+
|          Site Type          | Used | Fixed | Prohibited | Available | Util% |
+-----------------------------+------+-------+------------+-----------+-------+
| Bonded IOB                  |   32 |     0 |          0 |       106 | 30.19 |
| Bonded IPADs                |    0 |     0 |          0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |          0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |          0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |          0 |         5 |  0.00 |
| OUT_FIFO                    |    0 |     0 |          0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |          0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |          0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |          0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |          0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |          0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |          0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |          0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |          0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |          0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |          0 |       106 |  0.00 |
+-----------------------------+------+-------+------------+-----------+-------+


5. Clocking
-----------

+------------+------+-------+------------+-----------+-------+
|  Site Type | Used | Fixed | Prohibited | Available | Util% |
+------------+------+-------+------------+-----------+-------+
| BUFGCTRL   |    0 |     0 |          0 |        32 |  0.00 |
| BUFIO      |    0 |     0 |          0 |        20 |  0.00 |
| MMCME2_ADV |    0 |     0 |          0 |         5 |  0.00 |
| PLLE2_ADV  |    0 |     0 |          0 |         5 |  0.00 |
| BUFMRCE    |    0 |     0 |          0 |        10 |  0.00 |
| BUFHCE     |    0 |     0 |          0 |        72 |  0.00 |
| BUFR       |    0 |     0 |          0 |        20 |  0.00 |
+------------+------+-------+------------+-----------+-------+


6. Specific Feature
-------------------

+-------------+------+-------+------------+-----------+-------+
|  Site Type  | Used | Fixed | Prohibited | Available | Util% |
+-------------+------+-------+------------+-----------+-------+
| BSCANE2     |    0 |     0 |          0 |         4 |  0.00 |
| CAPTUREE2   |    0 |     0 |          0 |         1 |  0.00 |
| DNA_PORT    |    0 |     0 |          0 |         1 |  0.00 |
| EFUSE_USR   |    0 |     0 |          0 |         1 |  0.00 |
| FRAME_ECCE2 |    0 |     0 |          0 |         1 |  0.00 |
| ICAPE2      |    0 |     0 |          0 |         2 |  0.00 |
| PCIE_2_1    |    0 |     0 |          0 |         1 |  0.00 |
| STARTUPE2   |    0 |     0 |          0 |         1 |  0.00 |
| XADC        |    0 |     0 |          0 |         1 |  0.00 |
+-------------+------+-------+------------+-----------+-------+


7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| OBUFT    |   32 |                  IO |
+----------+------+---------------------+


8. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+


9. Instantiated Netlists
------------------------

+----------+------+
| Ref Name | Used |
+----------+------+
```

7 Appendices

7.1 Top Module - Processor

```
`timescale 1ns / 1ps

// Defining the processor module with a parameter for data width
module processor #(
   data_width = 32) // Data width parameter set to 32 bits
   (
   input logic clk, // Clock input
   reset, // Reset signal input
```

```
    output logic [31:0] data_out // 32-bit data output
);

    // Internal signal declarations for various control signals
    logic [6:0] opcode; // 7-bit opcode signal
    logic [3:0] ALU_operation; // 4-bit ALU operation signal
    logic [6:0] Funct7; // 7-bit Funct7 signal
    logic [2:0] Funct3; // 3-bit Funct3 signal

    // Instantiating the control unit module
    control_unit control_unit(); // Control unit submodule

    // Instantiating the datapath module
    datapath datapath(); // Datapath submodule

endmodule
```

7.2 Control Unit

```
module control_unit (
    input logic [6:0] opcode, // Input opcode field
    input logic [6:0] funct7, // Input funct7 field
    input logic [2:0] funct3, // Input funct3 field

    output logic ALUSrc, // Output for ALU source selection
    output logic MemtoReg, RegtoMem, // Output flags for memory-to-register and register-to-memory
    output logic RegWrite, // Output for register write control
    output logic MemRead, MemWrite, // Output flags for memory read and write control

    output logic [3:0] ALUOp, // Output for ALU operation code
    output logic Con_Jalr, Con_beq, Con_blt // Output control signals for specific instructions
);

    logic [5:0] address; // 6-bit logic signal for address calculation
    logic [16:0] micro_instruction; // 17-bit logic signal for microinstruction

    logic [6:0] R_type = 7'b0110011; // Define opcode values for R-type instructions
    logic [6:0] I_type = 7'b0010011; // Define opcode values for I-type instructions
    logic [6:0] LW = 7'b0000011; // Define opcode values for LW-type instructions
    logic [6:0] SW = 7'b0100011; // Define opcode values for SW-type instructions
    logic [6:0] BR = 7'b1100011; // Define opcode values for branch instructions
    logic [6:0] JALR = 7'b1100111; // Define opcode values for JALR instruction
    logic [6:0] MEMCOPY = 7'b111110; // Define opcode values for MEMCOPY instruction

    always_comb begin
        if (funct3 == 3'b000 && (opcode == R_type || opcode == I_type))
            address = 6'd0; // Set address to 6'd0 for specific R-type and I-type instructions

        if (opcode == MEMCOPY)
            address = 6'd32; // Set address to 6'd32 for the MEMCOPY instruction
    end

    micro_instruction_memory micro_instruction_memory (.address(address), .data(micro_instruction));

    assign ALUSrc = micro_instruction[0]; // Set ALUSrc based on micro_instruction bit 0
    assign MemtoReg = micro_instruction[1]; // Set MemtoReg based on micro_instruction bit 1
    assign RegtoMem = micro_instruction[2]; // Set RegtoMem based on micro_instruction bit 2
    assign RegWrite = micro_instruction[3]; // Set RegWrite based on micro_instruction bit 3
    assign MemRead = micro_instruction[4]; // Set MemRead based on micro_instruction bit 4
    assign MemWrite = micro_instruction[5]; // Set MemWrite based on micro_instruction bit 5
    assign ALUOp = micro_instruction[9:6]; // Set ALUOp based on micro_instruction bits 9 to 6
    assign Con_Jalr = micro_instruction[10]; // Set Con_Jalr based on micro_instruction bit 10
    assign Con_beq = micro_instruction[11]; // Set Con_beq based on micro_instruction bit 11
    assign Con_blt = micro_instruction[12]; // Set Con_blt based on micro_instruction bit 12

endmodule
```

7.3 Micro Instruction Memory

```
'timescale 1ns / 1ps

module micro_instruction_memory
#(parameter WIDTH = 16,
    parameter DEPTH = 6)
(
    input logic [(DEPTH-1):0] address,
    output logic [(WIDTH-1):0] data
);

    logic [(WIDTH-1):0] mem [0:DEPTH-1];

// ALUSrc[0],
// MemtoReg[1],
// RegtoMem[2],
// RegWrite[3],
// MemRead[4],
// MemWrite[5],
// ALUOp[9:6],
// Con_Jalr[10],
// Con_beq[11],
// Con_blt[12],
// MemAcc[15:13],

    //R_type
    assign mem[6'd0] = 16'b000_100_0000_000_000; //add
    assign mem[6'd1] = 16'b000_100_0001_000_000; //sub
    assign mem[6'd2] = 16'b000_100_0101_000_000; //sll
    assign mem[6'd3] = 16'b000_100_0111_000_000; //slt
    assign mem[6'd4] = 16'b000_100_1000_000_000; //sltu
    assign mem[6'd5] = 16'b000_100_0100_000_000; //xor
    assign mem[6'd6] = 16'b000_100_0110_000_000; //srl
    assign mem[6'd7] = 16'b000_100_1001_000_000; //sra
    assign mem[6'd8] = 16'b000_100_0011_000_000; //or
    assign mem[6'd9] = 16'b000_100_0010_000_000; //and

    //I_type
    assign mem[6'd10] = 16'b100_100_0000_000_000; //addi
    assign mem[6'd11] = 16'b100_100_0101_000_000; //slli
    assign mem[6'd12] = 16'b100_100_0111_000_000; //slti
    assign mem[6'd13] = 16'b100_100_1000_000_000; //sltiu
    assign mem[6'd14] = 16'b100_100_0100_000_000; //xori
    assign mem[6'd15] = 16'b100_100_0110_000_000; //srli
    assign mem[6'd16] = 16'b100_100_1001_000_000; //srai
    assign mem[6'd17] = 16'b100_100_0011_000_000; //ori
    assign mem[6'd18] = 16'b100_100_0010_000_000; //andi

    assign mem[6'd19] = 16'b110_110_0000_000_001; //lw
    assign mem[6'd20] = 16'b110_110_0000_000_010; //lh
    assign mem[6'd21] = 16'b110_110_0000_000_011; //lb
    assign mem[6'd22] = 16'b110_110_0000_000_100; //lhu
    assign mem[6'd23] = 16'b110_110_0000_000_101; //lbu

    //S_type
    assign mem[6'd24] = 16'b001_001_0000_000_001; //sw
    assign mem[6'd25] = 16'b001_001_0000_000_010; //sh
    assign mem[6'd26] = 16'b001_001_0000_000_011; //sb

    //B_type
    assign mem[6'd27] = 16'b000_000_1010_010_000; //beq
    assign mem[6'd28] = 16'b000_000_1010_010_000; //bnq
    assign mem[6'd29] = 16'b000_000_0111_001_000; //blt
    assign mem[6'd30] = 16'b000_000_0111_001_000; //bge
    assign mem[6'd30] = 16'b000_000_1000_001_000; //bltu
    assign mem[6'd30] = 16'b000_000_1000_001_000; //bgeu

    assign mem[6'd30] = 16'b000_100_0000_100_000; //jalr

    //MUL
    assign mem[6'd31] = 16'b000_100_1101_000_000;

    //MEMCOPY
    assign mem[6'd32] = 16'b000_011_0000_000_111;

    always_comb begin
        data = mem[address];
    end

endmodule
```

7.4 Datapath

```
module datapath #(
    parameter PC_W = 9, // Width of the program counter
    parameter INS_W = 32, // Width of the instructions
    parameter RF_ADDRESS = 5, // Width of the register file address
    parameter DATA_W = 32, // Width of the data
    parameter DM_ADDRESS = 9, // Width of the data memory address
    parameter ALU_CC_W = 4 // Width of the ALU condition code
)(
    input logic clk, reset, RegWrite, MemtoReg, RegtoMem, ALUsrc, MemWrite, MemRead, // Input signals
    input logic [2:0] MemAcc, // Memory access
    input logic Con_beq, // Control signal for branch equal
    input logic Con_blt, // Control signal for branch less than
    input logic Con_Jalr, // Control signal for jump and link register
    input logic [ALU_CC_W - 1:0] ALU_CC, // ALU condition code
    output logic [6:0] opcode, // Output opcode
    output logic [6:0] Funct7, // Output Funct7
    output logic [2:0] Funct3, // Output Funct3
    output logic [31:0] ALU_Result // Output of the ALU result
);

    wire [8:0] PCPlus4; // Wire for the incremented program counter
    wire [8:0] PCPlus4_unsign_extend; // Wire for the unsigned extended program counter
    wire [31:0] pc; // Wire for the program counter

    // Instance of the program counter module
    program_counter #(
    .INS_ADDRESS(PC_W),
    .PC_WIDTH(32)
    ) program_counter (
        .clk(clk),
        .reset(reset),
        .next_pc(PCPlus4_unsign_extend),
        .branch(branch), // Branch signal
        .pc(pc) // Program counter
    );

    assign PCPlus4_unsign_extend = {23'b0, PCPlus4}; // Extending the program counter
    assign PCPlus4 = pc + 9'b100; // Incrementing the program counter

    logic [31:0] Instr; // Instruction variable
    instruction_memory instruction_memory (PC, Instr); // Accessing the instruction memory

    assign opcode = Instr[6:0]; // Extracting opcode from instruction
    assign Funct3 = Instr[14:12]; // Extracting Funct3 from instruction
    assign Funct7 = Instr[31:25]; // Extracting Funct7 from instruction

    data_interpreter data_store(Instr, Reg2, ST); // Interpreting data for store operation
    mux_2 #(32) resmux_store(Reg2, ST, RegtoMem, Store_data); // Multiplexer for store operation
        result

    register_file register_file(clk, reset, RegWrite, Instr[11:7], Instr[19:15], Instr[24:20], Result
        , Reg1, Reg2); // Register file instance

    mux_2 #(32) resmux(ALUResult, LD, MemtoReg, Read_Alu_Result); // Multiplexer for memory-to-
        register operation
    mux_2 #(32) resmux_jalr(Read_Alu_Result, {23'b0, PCPlus4}, (Jalr), Jal_test); // Multiplexer for
        jump-and-link operation

    immediate_generator Ext_Imm (Instr,ExtImm); // Generating immediate value

    mux_2 #(32) srcbmux(Reg2, ExtImm, (ALUsrc||Jalr), SrcB); // Multiplexer for source B selection
    ALU_32bit ALU_32bit(Reg1, SrcB, ALU_CC, ALUResult, zero); // 32-bit ALU operation

    assign ALU_Result = Result; // Assigning the ALU result

    data_interpreter data_load(Instr, ReadData, LD); // Interpreting data for load operation

    logic [31:0] temp_arr = ALUResult; // Temporary storage for ALU result
    data_memory data_memory(clk, MemRead, MemWrite, MemAcc, temp_arr[8:0], Store_data, ReadData); //
        Data memory operation

endmodule
```

7.5 ALU

```
`timescale 1ns / 1ps

module ALU_32bit(
    input [31:0] operandA,
    input [31:0] operandB,
    input [3:0] ALU_control,
    output reg [31:0] result,
    output reg zero_flag
);

    always @* begin
        case (ALU_control)
            4'b0000: result = operandA + operandB; // ADD
            4'b0001: result = operandA - operandB; // SUB
            4'b0010: result = operandA & operandB; // AND
            4'b0011: result = operandA | operandB; // OR
            4'b0100: result = operandA ^ operandB; // XOR
            4'b0101: result = operandA << operandB; // SLL
            4'b0110: result = operandA >> operandB; // SRL
            4'b0111: result = ($signed(operandA) < $signed(operandB)) ? 32'h1 : 32'h0; // SLT
            4'b1000: result = (operandA < operandB) ? 32'h1 : 32'h0; // SLTU
            4'b1001: result = operandA >>> operandB; // SRA
            4'b1010: result = (operandA == operandB) ? 32'h1 : 32'h0; // EQ
            4'b1011: result = (operandA != operandB) ? 32'h1 : 32'h0; // NQ
            4'b1100: result = (operandA >= operandB) ? 32'h1 : 32'h0; // SGEU
            4'b1101: result = operandA * operandB; //MUL
            default: result = 32'h0;
        endcase

        // Calculate the zero_flag
        if (result == 32'h0) begin
            zero_flag = 1'b1;
        end else begin
            zero_flag = 1'b0;
        end
    end
endmodule
```

7.6 Program Counter

```
`timescale 1ns / 1ps

module data_memory#(
    parameter DM_ADDRESS = 9 ,
    parameter DATA_W = 32
)(
    input logic clk,
    input logic MemRead , // comes from control unit
    input logic MemWrite , // Comes from control unit
    input logic [2:0] MemAcc,
    input logic [DM_ADDRESS -1:0] a , // Read / Write address - 9 LSB bits of the ALU output
    input logic [DATA_W -1:0] wd , // Write Data
    output logic [DATA_W -1:0] rd // Read Data
);

    logic [DATA_W-1:0] mem [(2**DM_ADDRESS)-1:0];

    // Initialize mem to zero
    initial begin
        integer i;
        for (i = 0; i < (2**DM_ADDRESS); i = i + 1) begin
            mem[i] = 32'b0; // Assuming 32-bit wide data
        end
    end

    always_comb begin
        if(MemRead)
            case(MemAcc)
                3'b001: rd = mem[a]; //lw
                3'b010: rd = {mem[a][15] ? {16{1'b1}}: {16{1'b0}}, mem[a][15:0]}; //lh
                3'b011: rd = {mem[a][7] ? {24{1'b1}}: {24{1'b0}}, mem[a][7:0]}; //lb
                3'b100: rd = {16'b0, mem[a][15:0]}; //lhu
                3'b101: rd = {24'b0, mem[a][7:0]}; //lbu
            endcase
    end

    always @(posedge clk) begin
        if (MemWrite)
            case(MemAcc)
                3'b001: mem[a] = wd; //sw
                3'b010: mem[a] = {wd[15] ? {16{1'b1}}: {16{1'b0}}, wd[15:0]}; //sh
                3'b011: mem[a] = {wd[7] ? {24{1'b1}}: {24{1'b0}}, wd[7:0]}; //sb
            endcase
    end
```

```

endmodule
```

7.7 Register File

```
module register_file (
    input clk, // Clock input
    input rst, // Reset input
    input en, // Enable input
    input [4:0] rs1, rs2, rd, // Register indices
    input [31:0] wdata, // Data to be written
    output [31:0] rdata1, rdata2 // Data read from registers
);

    reg [31:0] regs [31:0]; // 32 32-bit registers

    always @(posedge clk or posedge rst) begin
        if (rst) begin // If reset is active
            for (int i = 0; i < 32; i++) begin // Initialize all registers to 0
                regs[i] <= 32'h0; // Set each register to 0
            end
        end else if (en) begin // If enable signal is active
            if (rd != 0) begin // Check if the destination register is not 0
                regs[rd] <= wdata; // Write the data to the specified register
            end
        end
    end

    assign rdata1 = regs[rs1]; // Output the data from rs1
    assign rdata2 = regs[rs2]; // Output the data from rs2

endmodule
```

7.8 Data Memory

```
`timescale 1ns / 1ps

module data_memory#(
    parameter DM_ADDRESS = 9 ,
    parameter DATA_W = 32
)(
    input logic clk, // Clock signal
    input logic MemRead , // Signal from control unit for memory read
    input logic MemWrite , // Signal from control unit for memory write
    input logic [2:0] MemAcc, // Memory access signal
    input logic [DM_ADDRESS -1:0] a , // Read / Write address - 9 LSB bits of the ALU output
    input logic [DATA_W -1:0] wd , // Write Data
    output logic [DATA_W -1:0] rd // Read Data
);

    logic [DATA_W-1:0] mem [(2**DM_ADDRESS)-1:0]; // Memory array

    // Initialize mem to zero
    initial begin
        integer i;
        for (i = 0; i < (2**DM_ADDRESS); i = i + 1) begin
            mem[i] = 32'b0; // Assuming 32-bit wide data
        end
    end

    always_comb begin
```

```
        if(MemRead)
            case(MemAcc) // Memory access cases
                3'b001: rd = mem[a]; // Load word
                3'b010: rd = {mem[a][15] ? {16{1'b1}}: {16{1'b0}}, mem[a][15:0]}; // Load half word
                3'b011: rd = {mem[a][7] ? {24{1'b1}}: {24{1'b0}}, mem[a][7:0]}; // Load byte
                3'b100: rd = {16'b0, mem[a][15:0]}; // Load half word unsigned
                3'b101: rd = {24'b0, mem[a][7:0]}; // Load byte unsigned
            endcase
    end

    always @(posedge clk) begin
        if (MemWrite)
            case(MemAcc) // Memory access cases
                3'b001: mem[a] = wd; // Store word
                3'b010: mem[a] = {wd[15] ? {16{1'b1}}: {16{1'b0}}, wd[15:0]}; // Store half word
                3'b011: mem[a] = {wd[7] ? {24{1'b1}}: {24{1'b0}}, wd[7:0]}; // Store byte
            endcase
    end

endmodule
```

7.9 MUX

```
`timescale 1ns / 1ps

module mux_2
    #(parameter WIDTH = 9)
    (input logic [WIDTH-1:0] d0, d1,
    input logic select,
    output logic [WIDTH-1:0] outcome);

    assign outcome = select ? d1 : d0;

endmodule
```

7.10 Data Interpreter

```
`timescale 1ns / 1ps

module data_interpreter
    #(parameter WIDTH = 32)
    (input logic [WIDTH-1:0] inst, // Input instruction
    input logic [WIDTH-1:0] data, // Input data
    output logic [WIDTH-1:0] y); // Output data
    logic [31:0] Imm_out; // Intermediate variable for immediate values
    logic [15:0] s_bit; // Extracting lower bits of 'data' for manipulation
    logic [7:0] e_bit; // Extracting bits from 'data' for manipulation
    assign s_bit = data[15:0]; // Assigning lower bits of 'data' to 's_bit'
    assign e_bit = data[7:0]; // Assigning bits of 'data' to 'e_bit'

    // Combinational logic block to interpret data
    always_comb
    begin
        Imm_out = {inst[31]? {20{1'b1}}:{20{1'b0}}, inst[31:20]}; // Creating immediate values based
            on instruction
        if(inst[6:0] == 7'b0000011) // Checking for specific instruction type
            begin
                if(inst[14:12] == 3'b000) // Checking for specific function
                    y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit}; // Performing specific operation on
                        'e_bit'
                else if(inst[14:12] == 3'b001) // Checking for specific function
```

```
                    y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit}; // Performing specific operation on 's_bit'
                else if(inst[14:12] == 3'b100) // Checking for specific function
                    y = {24'b0, e_bit}; // Performing specific operation on 'e_bit'
                else if(inst[14:12] == 3'b101) // Checking for specific function
                    y = {16'b0, s_bit}; // Performing specific operation on 's_bit'
                else if(inst[14:12] == 3'b010) // Checking for specific function
                    y = data; // Directly assigning 'data' to 'y'
            end
        else if(inst[6:0] == 7'b0100011) // Checking for specific instruction type
        begin
            if(inst[14:12] == 3'b000) // Checking for specific function
                y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit}; // Performing specific operation on 'e_bit'
            else if(inst[14:12] == 3'b001) // Checking for specific function
                y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit}; // Performing specific operation on 's_bit'
            else if(inst[14:12] == 3'b010) // Checking for specific function
                y = data; // Directly assigning 'data' to 'y'
        end
    end
endmodule
```

7.11 Immediate Generator

```
`timescale 1ns / 1ps

module immediate_generator(
    input logic [31:0] inst_code, // Input instruction code
    output logic [31:0] Imm_out); // Output immediate value

    logic [4:0] srai; // Declaring srai variable for later use
    assign srai = inst_code[24:20]; // Assigning bits 24 to 20 of inst_code to srai

    always_comb
    case(inst_code[6:0]) // Checking the opcode bits of the instruction
        7'b0000011: // If opcode corresponds to '0000011'
        Imm_out = {inst_code[31]? {20{1'b1}}:20'b0 , inst_code[31:20]}; // Generate immediate value for load instructions
        7'b0010011: // If opcode corresponds to '0010011'
        begin
            if((inst_code[31:25]==7'b0100000&&inst_code[14:12]==3'b101)||(inst_code[14:12]==3'b001)||inst_code[14:12]==3'b101) // Checking for specific conditions
                Imm_out = {srai[4]? {27{1'b1}}:27'b0,srai}; // Generate immediate value for arithmetic right shift instructions
            else
                Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:20]}; // Generate immediate value for other instructions
        end
        7'b0100011: // If opcode corresponds to '0100011'
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:25], inst_code[11:7]}; // Generate immediate value for store instructions
        7'b1100011: // If opcode corresponds to '1100011'
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[7], inst_code[30:25],inst_code[11:8],1'b0}; // Generate immediate value for branch instructions
        7'b1100111: // If opcode corresponds to '1100111'
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[30:25], inst_code[24:21], inst_code[20]}; // Generate immediate value for jump and link instructions
        7'b0010111: // If opcode corresponds to '0010111'
        Imm_out = {inst_code[31]? 1'b1:1'b0 , inst_code[30:20], inst_code[19:12],12'b0}; // Generate immediate value for upper immediate instructions
        7'b0110111: // If opcode corresponds to '0110111'
        Imm_out = {inst_code[31:12], 12'b0}; // Generate immediate value for upper immediate
            instructions
        7'b0110111: // If opcode corresponds to '0110111'
        Imm_out = {inst_code[31:12], 12'b0}; // Generate immediate value for upper immediate
            instructions
        7'b1101111: // If opcode corresponds to '1101111'
        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[19:12], inst_code[19:12],inst_code[20],
            inst_code[30:25],inst_code[24:21],1'b0}; // Generate immediate value for jump and link
            register instructions
        default :
        Imm_out = {32'b0}; // Default case, assign immediate value as 0
    endcase

endmodule
```

8 References

- Processor Design #2: Introduction to RISC-V - Simon Southwell
- RISC-V Instruction Encoder/Decoder
- RV32I, RV64I Instructions