Faculty of Engineering & Technology

Electrical & Computer Engineering Department

COMPUTER ARCHITECTURE - ENCS4370

Project # 2: Simple RISC processor

Prepared by:

Abdulghafer Qeel 1210408

Ismail Tarteer 1211243

Mosa Sbeih 1211250

Instructors: Dr.Aziz Qaroush

Sections: 1

Date: 22/6/2024

Abstract:

This paper summarizes the design and realization of a simple, 5-stage pipelined RISC processor using Verilog for the Computer Architecture course (ENCS4370) conducted in the second semester of 2023/2024. The project's objective is to design, simulate, and verify a 16-bit processor with an ISA that includes four instruction types: R-type, I-type, J-type, and S-type. It has eight general-purpose registers and uses separate memories for data and instructions. The memory implemented is byte-addressable and uses little-endian ordering. To enhance instruction throughput, it is constructed using a five-stage pipeline (fetch, decode, ALU, memory access, and write back). The report documents the following:

Design and Implementation: Explain the various Datapath and control path components in detail, the generation of control signals, and how these are combined. Block diagrams and implementation choices are provided to justify the design. The core is simulated using a testbench in order to validate various code sequences written in the ISA. Documentation of simulation results, such as snapshots and expected results, is done.

Teamwork: The project was done in a group, and every member contributed to designing, implementing, testing, writing reports, and demonstrating the project. The project shows single components' correctness and overall design with surety for completeness and accuracy in executing the reduced but functional instruction set. Verification results confirm that the processor can handle data and control hazards effectively, showcasing a robust multicycle design. The report concludes with project findings and an outlook for future improvements.

Table of Contents

Table of Figures

RTL Notations for given instructions

- AND R-Type: `Rd ← Rs1 & Rs2`
- ADD R-Type: `Rd ← Rs1 + Rs2`
- SUB R-Type: `Rd ← Rs1 - Rs2`
- ADDI I-Type: `Rd ← Rs1 + Imm`
- ANDI I-Type: `Rd ← Rs1 & Imm`
- LW I-Type: `Rd ← Mem(Rs1 + Imm)`
- LBu I-Type: `Rd ← Mem(Rs1 + Imm)`
- LBs I-Type: `Rd ← Mem(Rs1 + Imm)`
- SW I-Type: `Mem(Rs1 + Imm) ← Rd`
- BGT I-Type:
```
if (Rd > Rs1) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BGTZ I-Type:
```
if (Rd > R0) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BLT I-Type:
```
if (Rd < Rs1) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BLTZ I-Type:
```
if (Rd < R0) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BEQ I-Type:
```
if (Rd == Rs1) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BEQZ I-Type:
```
if (Rd == R0) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BNE I-Type:
```
if (Rd != Rs1) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- BNEZ I-Type:
```
if (Rd != Rs1) then Next PC ← PC + sign_extend(Imm)
else PC ← PC + 2
```
- JMP J-Type: `Next PC ← {PC[15:10], Immediate}`

- CALL J-Type: `Next PC ← {PC[15:10], Immediate}, R7 ← PC + 4`
- RET J-Type: `Next PC ← R7`
- Sv S-Type: `M[Rs] ← Imm`

Design Description:

This is a design of a five stage multi-cycle processor: fetch, decode, execute, memory access, and write back. The processor follows the provided architecture and incorporates the following components:

Data Path:

1. Program Counter (PC):

PC is a special-purpose register that holds the address of the next instruction to be fetched in the instruction memory. The PC gets updated at the end of each instruction execution to point to the next instruction. It plays a crucial role in controlling the program flow and ensuring the sequential execution of instructions.

2. Instruction Memory:

Instruction Memory is a component that stores the program instructions in the processor. It is accessed by the PC to fetch the next instruction for execution, the Instruction Memory holds the machine code instructions, each represented by a fixed number of bits (e.g., 32 bits).

3. Register File:

Register File is a component in the processor that stores a set of general-purpose registers. It provides fast access to the registers, allowing data to be read from and written to them during instruction execution, the Register File is typically organized as an array of registers, each capable of holding a fixed-size data value (e.g., 16 bits).

4. ALU (Arithmetic Logic Unit):

ALU is a fundamental component in a processor responsible for performing arithmetic and logical operations. It operates on binary data, executing operations such as addition, subtraction, AND, OR, and more. The ALU takes input operands, performs the specified operation, and produces a result along with status flags such as zero and negative flags.

5. Data Memory

Data Memory is a component in the processor that stores data values during program execution. It is used to read from and write data during memory access instructions, such as load and store operations. The Data Memory is typically organized as a separate memory module, distinct from the instruction memory, and provides storage for variables, arrays, and other data used by the program.

6. Some Multiplexers:

Some Multiplexers are a component we used to control the information that must be enter to the inputs for each of the components in the stages.

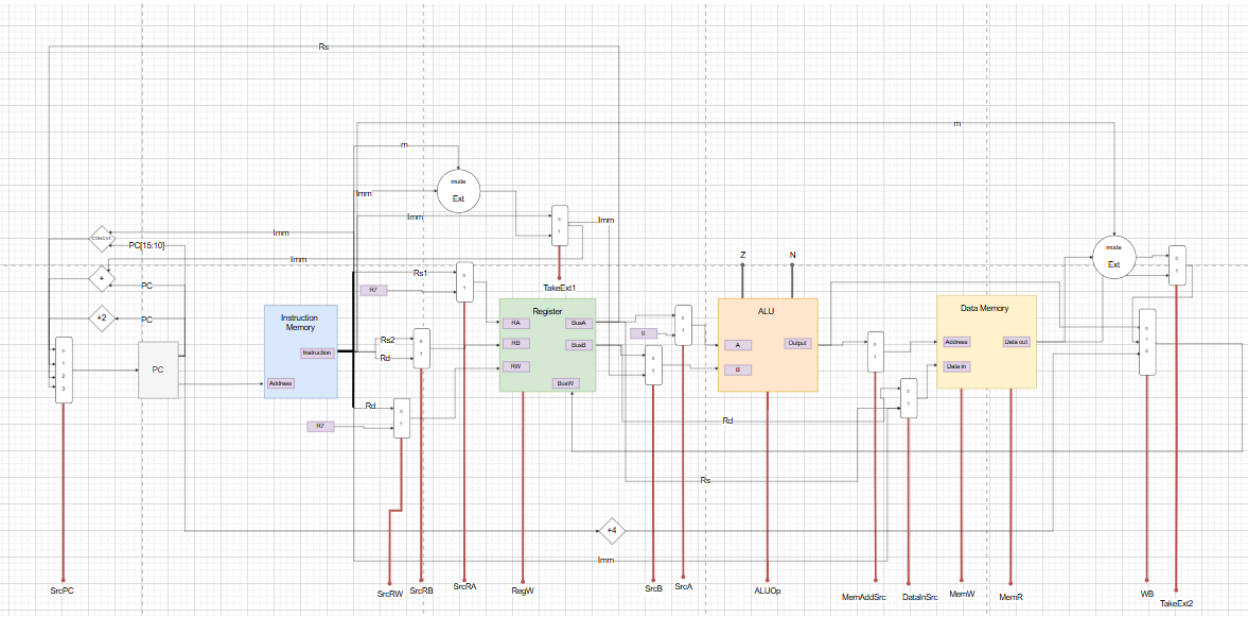

Figure 1: Data Path Of Multi-cycle RISC processor

Control Unit:

The Control Unit in our design is responsible for generating the control signals that manage the operation of various components within the processor. It interprets the opcode, mode, and flag inputs (Zero Flag and Negative Flag) to produce the necessary control signals for executing instructions. Depending on the opcode, it sets signals for selecting the source of the Program Counter (PC), read/write operations for registers and memory, ALU operation codes, memory addressing, data sources, and handling branch instructions. For instance, it distinguishes between arithmetic operations like ADD and SUB, immediate operations like ANDI and ADDI, memory operations like LW and SW, and control flow instructions like BGT, BLT, BEQ, BNE, JMP, CALL, RET, and others. By doing so, the Control Unit ensures that each instruction is executed correctly by coordinating the activities of the ALU, registers, memory units, and the PC, thus maintaining the proper execution flow of the program.

How data path works:

OPCODE	opcode	m	byte_en	Type	SrcPc	SrcRW	SrcRB	SrcRA	RegW	SrcB	SrcA	ALUOp	MemAddSrc	DataInSrc	MemW	MemR	WB	TakeExt1	TakeExt2
AND	0		x	R-Type	0	0	0	0	1	0	0	2	x	x	0	0	0	x	x
ADD	1		x	R-Type	0	0	0	0	1	0	0	0	x	x	0	0	0	x	x
SUB	2		x	R-Type	0	0	0	0	1	0	0	1	x	x	0	0	0	x	x
ANDI	3		x	I-Type	0	0	x	0	1	1	0	2	x	x	0	0	0	0	x
ADDI	4		x	I-Type	0	0	x	0	1	1	0	0	x	x	0	0	0	0	x
LW	5		0	I-Type	0	0	x	0	1	1	0	0	0	x	0	1	1	0	0
LBs	6	1	1	I-Type	0	0	x	0	1	1	0	0	0	x	0	1	1	0	1
Lbu	6	0	1	I-Type	0	0	x	0	1	1	0	0	0	x	0	1	1	0	1
SW	7		0	I-Type	0	x	1	0	0	1	0	0	0	0	1	0	x	0	x
BGT	8	0	x	I-Type	if taken 1 else 0	x	1	0	0	0	0	1	x	x	0	0	x	1	x
BGTZ	8	1	x	I-Type	if taken 1 else 0	x	1	0	0	0	1	1	x	x	0	0	x	1	x
BLT	9	0	x	I-Type	if taken 1 else 0	x	1	0	0	0	0	1	x	x	0	0	x	1	x
BLTZ	9	1	x	I-Type	if taken 1 else 0	x	1	0	0	0	1	1	x	x	0	0	x	1	x
BEQ	10	0	x	I-Type	if taken 1 else 0	x	1	0	0	0	0	1	x	x	0	0	x	1	x
BEQZ	10	1	x	I-Type	if taken 1 else 0	x	1	0	0	0	1	1	x	x	0	0	x	1	x
BNE	11	0	x	I-Type	if taken 1 else 0	x	1	0	0	0	0	1	x	x	0	0	x	1	x
BNEZ	11	1	x	I-Type	if taken 1 else 0	x	1	0	0	0	1	1	x	x	0	0	x	1	x
JMP	12		x	J-Type	2	x	x	x	0	x	x	x	x	x	0	0	x	x	x
CALL	13		x	J-Type	2	1	x	x	1	x	x	x	x	x	0	0	2	x	x
RET	14		x	J-Type	3	x	x	1	0	x	x	x	x	x	0	0	x	x	x
Sv	15		1	S-Type	0	x	x	0	0	x	x	x	1	1	1	0	x	x	x

Boolean Expressions:

1. **RegW**:

   RegW =

   (opcode=0)+(opcode=1)+(opcode=2)+(opcode=3)+(opcode=4)+
   (opcode=5)+(opcode=6)+(opcode=13)

2. **MemW**:

   MemW= (opcode=7)+(opcode=15)

3. **MemR**:

   MemR= (opcode=5)+(opcode=6)

4. **WB**:

   WB[0] : (opcode=0)+(opcode=1)+(opcode=2)+(opcode=3)+(opcode=4)

   WB[1]: (opcode=5)+(opcode=6)

   WB[2]: (opcode=13)

5. **ALUOp**:

   ALUOp[0]=(opcode=1)+(opcode=4) +(opcode=5) +(opcode=6) +(opcode=7)

   ALUOp[1]=(opcode=2)+(opcode=8)+(opcode=9)+(opcode=10)+(opcode=11)

   ALUOp[1]=(opcode=0)+(opcode=3)

6. **SrcRW**:

   SrcRW= (opcode=13)

7. **SrcRB**:

   SrcRB= (opcode=7)+ (opcode=8)+(opcode=9)+(opcode=10)+(opcode=11)

8. **SrcRA**:

   SrcRA=(opcode=14)

9. **SrcB**:

   SrcB=(opcode=3)+(opcode=4)+(opcode=5)+(opcode=6)+(opcode=7)

10. **SrcA**:

    SrcA=((opcode=8)+(opcode=9)+(opcode=10)+(opcode=11)) and (m=1)

11. **MemAddSrc**:

    MemAddSrc=(opcode=15)

12. **DataInSrc**:

    DataInSrc= (opcode=15)

13. **TakeExt1**:

    TakeExt1= (opcode=8)+ (opcode=9)+(opcode=10)+(opcode=11)

14. **TakeExt2**:

    TakeExt2=(opcode=6)+(opcode=7)

15. **byte_en**:

    byte_en=(opcode=6 and m=1)

Implementation Details and Design Choices:

1. **Instruction Size:** Each instruction is 16 bits in size, accommodating the opcode, register addresses, immediate values, and other necessary fields.
2. **Register File:** The processor includes a 8 x 16 register file for efficient data storage and retrieval. Each register can hold a 16-bit value.
3. **Program Counter (PC):** The PC keeps track of the address of the current instruction being executed. It gets updated at the end of each instruction to fetch the next instruction.
4. **ALU and Zero, Carry, Negative Signals**: The ALU in the processor is designed to perform arithmetic and logical operations. In addition to the "zero" signal, which indicates if the result of the last operation is zero, the ALU also generates the "negative" signal. The "negative" signal indicates if the result is negative. These signals are important for subsequent instructions or conditional branching that rely on the ALU operation results. Including these signals provides a more comprehensive and versatile ALU functionality.
5. **Separate Data and Instruction Memories:** The processor has separate memories for data and instructions. This separation allows independent access to data and instructions, improving overall performance.
6. **Five-Stage:** The processor follows a five-stage to enable concurrent instruction execution and maximize throughput.
7. **Multi-Cycle Processor:** The processor is designed as a multi-cycle processor, where each instruction takes multiple cycles to complete. This design allows for simpler control logic and more flexibility in optimizing the performance.

Notable Features:

1. **Instruction Types:** The processor supports four instruction types (R-Type, I-Type, J-Type, and S-Type) to provide a wide range of operations, enabling flexible programming.

2. **ALU Zero Signal:** Enables efficient branching and conditional execution based on ALU results.

3. **Negative Signal:** Indicates if the result of an operation is negative, supporting signed arithmetic.

Correctness of the individual components

We implemented these instructions to illustrate the work of the component.

```
memory[0]  = 16'b0000001000110100; // AND R1, R2, R3
memory[1]  = 16'b0001001000110101; // ADD R1, R2, R3
memory[2]  = 16'b0010001000110110; // SUB R1, R2, R3
memory[3]  = 16'b0011000100100001; // ADDI R1, R1, 1
memory[4]  = 16'b0100000100100010; // ANDI R1, R1, 2
memory[5]  = 16'b0101000100100100; // LW R1, 4(R2)
memory[6]  = 16'b0110000100100100; // LBu R1, 4(R2)
memory[7]  = 16'b0110010100100100; // LBs R1, 4(R2)
memory[8]  = 16'b0111000101000101; // SW 5(R2), R1
memory[9]  = 16'b1000000100110101; // BGT R1, R3, 5
memory[10] = 16'b1000010100100101; // BGTZ R1, 5
memory[11] = 16'b1001000100110101; // BLT R1, R3, 5
memory[12] = 16'b1001010100100101; // BLTZ R1, 5
memory[13] = 16'b1010000100110101; // BEQ R1, R3, 5
memory[14] = 16'b1010010100100101; // BEQZ R1, 5
memory[15] = 16'b1011000100110101; // BNE R1, R3, 5
memory[16] = 16'b1011010100100101; // BNEZ R1, 5
memory[17] = 16'b1100000000000101; // JMP 5
memory[18] = 16'b1101000000000101; // CALL 5
memory[19] = 16'b1110000000000000; // RET
memory[20] = 16'b1111001000000101; // Sv R2, 5
// Data Hazard Testing
memory[21] = 16'b0011000100100001; // ADDI R1, R1, 1 (initialize R1 to 1)
memory[22] = 16'b0001000110000100; // ADD R3, R1, R0 (use R1 in next cycle)
memory[23] = 16'b0011000110000010; // ADDI R3, R3, 2 (use R3 immediately)
memory[24] = 16'b0001001000110010; // ADD R4, R3, R1 (use R3 and R1)
// Control Hazard Testing
memory[25] = 16'b0011000100100001; // ADDI R1, R1, 1 (initialize R1 to 1)
memory[26] = 16'b1000010100100001; // BGTZ R1, 1 (branch if R1 > 0)
memory[27] = 16'b1100000000000100; // JMP 4 (jump to address 4 if branch taken)
```

```
memory[28] = 16'b0001001000100100; // ADD R4, R1, R1 (should be skipped if branch is taken)
memory[29] = 16'b0011001001000001; // ADDI R4, R4, 1 (address 4: execute if branch taken)
```

and these values was initialized on the registers

```
registers[0] = 16'b0;       // Register R0 is always zero
registers[1] = 16'h1234;    // Register R1
registers[2] = 16'h5678;    // Register R2
registers[3] = 16'h9ABC;    // Register R3
registers[4] = 16'hDEF0;    // Register R4
registers[5] = 16'h1357;    // Register R5
registers[6] = 16'h2468;    // Register R6
registers[7] = 16'hACE0;    // Register R7
```

1. Program Counter (PC):

As shown in the following Figure 3, there is a block for the Program Counter (PC) register, which has one 16 bit input and one 16 bit output, so that the address is stored for the next instruction to execute, and it is running at the positive edge of the clk.

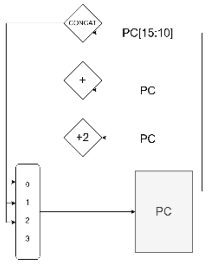

Figure 2:Program Counter (PC) Block

```
module pc_module (
    input wire clock,       // Clock signal
    input wire reset,       // Reset signal to initialize PC
    input wire enable,      // Enable signal for updating PC
    input wire [15:0] next_pc, // Next PC value to be loaded
    output reg [15:0] PC    // Current PC value
);

    // Always block triggered on the rising edge of the clock or reset signal
    always @(posedge clock or posedge reset) begin
        if (reset) begin
            PC <= 16'b0;  // If reset is high, initialize PC to 0
        end else if (enable) begin
            PC <= next_pc; // If enable is high, update PC with next_pc value
        end else begin
            PC <= PC;     // If neither reset nor enable is high, maintain the current PC value
        end
    end

endmodule
```

Figure 3:Program Counter (PC) Code

2. Instruction Memory:

As shown in the following Figure, there is a block for Instruction Memory, which has one 16 bit input and one 16-bit output, so that the instruction address enters it and the instruction exits from it.

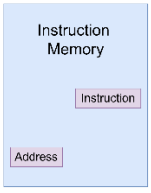

Figure 4:Instruction Memory Block

```
module instructionMemory (clock, address,im_enable, instruction);
    input wire clock;
    input wire [15:0] address;
    input im_enable;
    output reg [15:0] instruction;
    reg [15:0] memory [0:255];
    always @(*) begin
        if (im_enable == 1'b1) begin
        instruction <= memory[address];
        end
    end
    initial begin
        memory[0]  = 16'b0000001000110100; // AND R1, R2, R3


    end
endmodule
```

Figure 5:Instruction Memory Code

3. Registers File:

As shown in the following figure, there is a block for the Registers File, which contains three 5-bit entries, an entry to control writing to it, a 16-bit entry to store it in, and two 16-bit outputs, so that it has 8 16-bit registers in which the data is stored, and it is running at the positive edge of the clk.

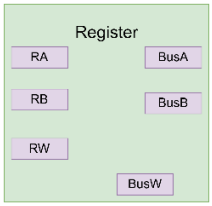

Figure 6:Registers File Block

```
module registerFile (
    input clk,                    // Clock signal
    input rst,                    // Reset signal
    input [2:0] RA,          // First read register address
    input [2:0] RB,          // Second read register address
    input [2:0] RW,          // Write register address
    input [15:0] WD,        // Data to be written
    input RegWrite,                // Write enable signal
    input rf_enable,
    output reg [15:0] BusA,      // First read data
    output reg [15:0] BusB,       // Second read data
    output reg [15:0] registers [7:0]    // for debugging purpos
);

    // Initialize registers
    integer i;
    initial begin
        for (i = 0; i < 8; i = i + 1) begin
            registers[i] = 16'b0;
        end
    end

    // Read data from registers
    always @(*) begin
        if (rf_enable == 1'b1) begin
            assign BusA = (RA == 3'b000) ? 16'b0 : registers[RA];
            assign BusB = (RB == 3'b000) ? 16'b0 : registers[RB];
        end
    end

    // Write data to register on the positive edge of the clock
    always @(*) begin
        if (rst) begin
            // Reset all registers to 0
            for (i = 0; i < 8; i = i + 1) begin
                registers[i] = 16'b0;
            end
        end else if (RegWrite && RW != 3'b000 && rf_enable == 1'b1) begin
            // Write data to the register if write enable is high and not writing to R0
            registers[RW] <= WD;
        end
    end
endmodule
```

Figure 7:Registers File Code

4. ALU:

As shown in the figure, there is the ALU Block, whose function is to carry out arithmetic operations and other tasks, so that two inputs of 16 bits size are entered into it, and also the input for selecting the operation to be executed is called ALU op, and these operations result in what results in the effect on the flags, and the other on the ALU out result.

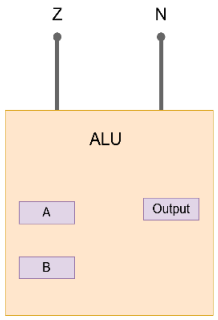

Figure 8:ALU Block

```
module ALU (
    input clk,
    input [15:0] A, B,          // ALU inputs
    input [1:0] ALU_op,         // ALU operation code
    output reg [15:0] result,   // ALU result
    output Z,                   // Zero flag
    output N                    // Negative flag
);

    reg carry_out;

    always @(posedge clk) begin
        case (ALU_op)
            2'b00: {carry_out, result} = A + B;  // ADD
            2'b01: {carry_out, result} = A - B;  // SUB
            2'b10: result = A & B;               // AND
            2'b11: result = A | B;               // OR
            // Add other operations as needed
            default: result = 16'b0;
        endcase
    end

    // Generate condition flags
    assign Z = (result == 16'b0);
    assign N = result[15];

endmodule
```

Figure 9::ALU Code

5. Data Memory:

As shown in the figure, there is the Data Memory Block, whose function is to store the data, so that two inputs of 16 bits size enter it, one is an address inside the memory and the other is data for storage, and also two inputs for choosing the operation to be executed in writing or execution called Mem rw, Mem rd so that Data is stored or read from memory, and it is running at the positive edge of the clk.

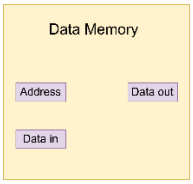

Figure 10:Data Memory Block

```
module DataMemory (
    input clk,                  // Clock signal
    input reg dm_enable,
    input [15:0] address,       // Memory address
    input [15:0] Data_in,  // Data to write
    input MemWrite,             // Memory write enable signal
    input MemRead,              // Memory read enable signal
    input byte_enable,          // Byte enable signal (1 for byte, 0 for word)
    output reg [15:0] Data_out, // Data read from memory
    output reg [7:0] memory [511:0]       // for debugging purpos
);

    // Define a memory array with 512 locations, each 8 bits wide (256 words of 16 bits)

    // Memory read operation
    always @(posedge clk) begin
        if (MemRead && dm_enable == 1'b1) begin
            if (byte_enable) begin
                // Byte access
                Data_out = {8'b0, memory[address]}; // Zero-extend to 16 bits
            end else begin
                // Word access (two consecutive bytes)
                Data_out = {memory[address + 1], memory[address]};
            end
        end else begin
            Data_out = 16'b0;
        end
    end

    // Memory write operation
    always @(posedge clk) begin
        if (MemWrite && dm_enable == 1'b1) begin
            if (byte_enable) begin
                // Byte access
                memory[address] <= Data_in[7:0];
            end else begin
                // Word access (two consecutive bytes)
                memory[address] <= Data_in[7:0];
                memory[address + 1] <= Data_in[15:8];
            end
        end
    end

endmodule
```

Figure 11:Data Memory Code

6. Extenders:

As shown in the following figure, there are Extenders Blocks, which extend the bits until they are in the 32-bit format in order to work on them correctly.

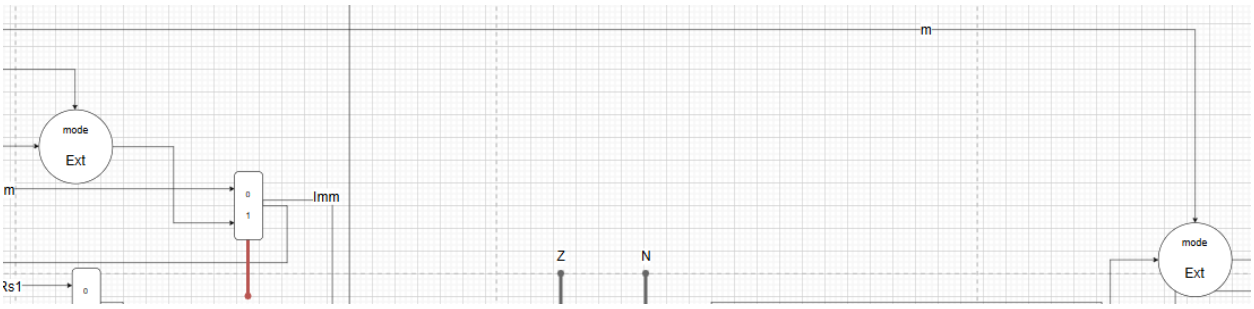

Figure 12:Extenders Blocks

The extenders and muxes wasn't built or used as separate modules, instead it was replaced by a dataflow codes in the main module CPU like this:

```
B <= (SrcB == 1'b0) ? BusB : {{11{imm[4]}}, imm};
```

Simulation and Testing

We have implemented a set of instructions that we wrote ourselves, which are the following:

```
memory[0] = 16'b0100000100100001; // ADDI R1, R1, 1
memory[1] = 16'b0100000100100001; // ADDI R1, R1, 1
memory[2] = 16'b0100000100100001; // ADDI R1, R1, 1
memory[3] = 16'b0100000100100001; // ADD R1, R1, R1
memory[4] = 16'b0001001001001000; // ADD R1, R1, R1
memory[5] = 16'b0111000100100000; // SW R1, R1
memory[6] = 16'b0110100110100000; // LBs R1,R5
memory[7] = 16'b1101000000001000; // Call 8
memory[8] = 16'b1000100100011111; // BGTZ R1
```

The goal of the program is to test different instruction functionality(arithmetic and logic operations, and branching) and here are the results:

```
# KERNEL: Time: 70, PC:     1, Instruction: 0100000100100001, R1: 0000000000000000, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 140, PC:     2, Instruction: 0100000100100001, R1: 0000000000000001, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 210, PC:     3, Instruction: 0100000100100001, R1: 0000000000000010, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 280, PC:     4, Instruction: 0001001001001000, R1: 0000000000000011, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 350, PC:     5, Instruction: 0111000100100000, R1: 0000000000000110, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 420, PC:     6, Instruction: 0110100110100000, R1: 0000000000000110, R7: 1010110011100000, M[6]: 00000110, Rd: 1, SrcPc: 0
# KERNEL: Time: 490, PC:     7, Instruction: 1101000000001000, R1: 0000000000000110, R7: 0000000000001011, M[6]: 00000110, Rd: 0, SrcPc: 2
# KERNEL: Time: 560, PC:     7, Instruction: 1101000000001000, R1: 0000000000000110, R7: 0000000000001011, M[6]: 00000110, Rd: 1, SrcPc: 1
# KERNEL: Time: 630, PC:     8, Instruction: 1000100100011111, R1: 0000000000000110, R7: 0000000000001011, M[6]: 00000110, Rd: 0, SrcPc: 2
```

Also here is another set of instruction to test:

```
memory[0] = 16'b0000001000110100; // AND R1, R2, R3
memory[1] = 16'b0001001000110101; // ADD R1, R2, R3
memory[2] = 16'b0010001000110110; // SUB R1, R2, R3
memory[3] = 16'b0011000100100001; // ADDI R1, R1, 1
memory[4] = 16'b0100000100100010; // ANDI R1, R1, 2
memory[5] = 16'b0101000100100100; // LW R1, 4(R2)
memory[6] = 16'b0110000100100100; // LBu R1, 4(R2)
memory[7] = 16'b0110010100100100; // LBs R1, 4(R2)
memory[8] = 16'b0111001001000100; // SW 4(R2), R1
```

```
memory[9]  = 16'b1000000100110101; // BGT R1, R3, 5
//memory[10] = 16'b1000010100100101; // BGTZ R1, 5
memory[11] = 16'b1001000100110101; // BLT R1, R3, 5
memory[12] = 16'b1001010100100101; // BLTZ R1, 5
memory[13] = 16'b1010000100110101; // BEQ R1, R3, 5
memory[14] = 16'b1010010100100101; // BEQZ R1, 5
memory[15] = 16'b1011000100110101; // BNE R1, R3, 5
memory[16] = 16'b1011010100100101; // BNEZ R1, 5
memory[17] = 16'b1100000000000101; // JMP 5
memory[18] = 16'b1101000000000101; // CALL 5
memory[19] = 16'b1110000000000000; // RET
memory[20] = 16'b1111010000000101; // Sv R2, 5
// Data Hazard Testing
memory[21] = 16'b0011000100100001; // ADDI R1, R1, 1 (initialize R1 to 1)
memory[22] = 16'b0001000110000100; // ADD R3, R1, R0 (use R1 in next cycle)
memory[23] = 16'b0011000110000010; // ADDI R3, R3, 2 (use R3 immediately)
memory[24] = 16'b0001001000110010; // ADD R4, R3, R1 (use R3 and R1)
// Control Hazard Testing
memory[25] = 16'b0011000100100001; // ADDI R1, R1, 1 (initialize R1 to 1)
memory[26] = 16'b1000010100100001; // BGTZ R1, 1 (branch if R1 > 0)
memory[27] = 16'b1100000000000100; // JMP 4 (jump to address 4 if branch taken)
memory[28] = 16'b0001001000100100; // ADD R4, R1, R1 (should be skipped if branch is taken)
memory[29] = 16'b0011001001000001; // ADDI R4, R4, 1 (address 4: execute if branch taken)
```

made to comment so the pc keeps going since the value becomes out of the initialized instructions range

and here are the console results:

```
# KERNEL: Time: 70, PC:     1, Instruction: 0001001000110101, R1: 0000000000000000, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 140, PC:    2, Instruction: 0010001000110110, R1: 0010010001101000, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 210, PC:    3, Instruction: 0011000100100001, R1: 1101101110011000, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 280, PC:    4, Instruction: 0100000100100010, R1: 0000000000000000, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 350, PC:    5, Instruction: 0101000100100100, R1: 0000000000000010, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 420, PC:    6, Instruction: 0110000100100100, R1: xxxxxxxxxxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 490, PC:    7, Instruction: 0110010100100100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0
# KERNEL: Time: 560, PC:    8, Instruction: 0111001001000100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 2, SrcPc: 0
# KERNEL: Time: 630, PC:    9, Instruction: xxxxxxxxxxxxxxxx, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: x, SrcPc: 0
# KERNEL: Time: 700, PC:   11, Instruction: 1001000100110101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0
# KERNEL: Time: 770, PC:   12, Instruction: 1001010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0
# KERNEL: Time: 840, PC:   13, Instruction: 1010000100110101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
```

KERNEL: Time: 910, PC: 14, Instruction: 1010010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

KERNEL: Time: 980, PC: 15, Instruction: 1011000100110101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1050, PC: 16, Instruction: 1011010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

KERNEL: Time: 1120, PC: 5, Instruction: 0101000100100100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 0, SrcPc: 2

KERNEL: Time: 1190, PC: 6, Instruction: 0110000100100100, R1: xxxxxxxxxxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1260, PC: 7, Instruction: 0110010100100100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1330, PC: 8, Instruction: 0111001001000100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

KERNEL: Time: 1400, PC: 9, Instruction: xxxxxxxxxxxxxxxx, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 2, SrcPc: 0

KERNEL: Time: 1470, PC: 10, Instruction: 1000010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: x, SrcPc: 0

KERNEL: Time: 1540, PC: 11, Instruction: 1001000100110101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1610, PC: 12, Instruction: 1001010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

KERNEL: Time: 1680, PC: 13, Instruction: 1010000100110101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1750, PC: 14, Instruction: 1010010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

KERNEL: Time: 1820, PC: 16, Instruction: 1011010100100101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 1890, PC: 17, Instruction: 1100000000000101, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 0, SrcPc: 0

KERNEL: Time: 1960, PC: 5, Instruction: 0101000100100100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 2030, PC: 6, Instruction: 0110000100100100, R1: xxxxxxxxxxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0

KERNEL: Time: 2100, PC: 7, Instruction: 0110010100100100, R1: 00000000xxxxxxxx, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 5, SrcPc: 0

Notice that the branches and the jump instructions prevented the program from reaching the the other instructions since it works properly so we will make some edits.

New instructions:

```
// Data Hazard Testing
    memory[0] = 16'b0011000100000001; // ADDI R1, R0, 1 (initialize R1 to 1)
    memory[1] = 16'b0001111001000100; // ADD R7, R1, R0 (use R1 in next cycle)
    memory[2] = 16'b0011011111100010; // ADDI R7, R7, 2 (use R7 immediately)
    memory[3] = 16'b0001111111001000; // ADD R7, R7, R1 (use R7 and R1)

    // Control Hazard Testing
    memory[4] = 16'b0011000100000001; // ADDI R1, R0, 1 (initialize R1 to 1)
    memory[5] = 16'b1000100100001001; // BGTZ R1, 9 (branch if R1 > 0)
    memory[6] = 16'b1100000000001000; // JMP 8 (jump to address 8 if branch taken)
    memory[7] = 16'b0001001000100100; // ADD R4, R1, R1 (should be skipped if branch is taken)
    memory[8] = 16'b0011001001000001; // ADDI R4, R4, 1 (address 8: execute if branch taken)
```

and here are the outputs on the

KERNEL: Time: 70, PC: 0, Instruction: 0011000100100001, R1: 0000000000000001, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: x, SrcPc: x

KERNEL: Time: 140, PC: 1, Instruction: 0001111001000100, R1: 0000000000000001, R7: 1010110011100000, M[6]: xxxxxxxx, Rd: 7, SrcPc: 0

```
# KERNEL: Time: 210, PC:    2, Instruction: 0011011111100010, R1: 0000000000000001, R7: 0000000000000001, M[6]: xxxxxxxx, Rd: 7, SrcPc: 0
# KERNEL: Time: 280, PC:    3, Instruction: 0001111111001000, R1: 0000000000000001, R7: 0000000000000011, M[6]: xxxxxxxx, Rd: 7, SrcPc: 0
# KERNEL: Time: 350, PC:    4, Instruction: 0011000100000001, R1: 0000000000000001, R7: 0000000000000111, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
# KERNEL: Time: 420, PC:    5, Instruction: 1000100100001001, R1: 0000000000000001, R7: 0000000000000111, M[6]: xxxxxxxx, Rd: 1, SrcPc: 1
# KERNEL: Time: 490, PC:   15, Instruction: xxxxxxxxxxxxxxx, R1: 0000000000000001, R7: 0000000000000000, M[6]: xxxxxxxx, Rd: 1, SrcPc: 0
```

Test Bench:

We have a testbench for every module used in the project which will be provided in the project files.

There is no testbench for the main module CPU since it's a testbench itself.

References

- Github.com
- Getting Started with Verilog - GeeksforGeeks
- cpu architecture - Multicycle implementation of MIPS ISA - Stack Overflow