Computer Organization and Architecture Laboratory
CS39001
Design and Synthesize of a 32-bit processor using Verilog

Group - 08
Bratin Mondal (21CS10016)
Somya Kumar (21CS30050)

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

1 Introduction

This report details the design of a 32-bit RISC-like processor using Verilog, featuring a register bank with special-purpose registers, byte-addressable memory, and various addressing modes, including immediate and base addressing, tailored for efficient computing and robust functionality.

2 Instruction Set Design

The instruction set architecture we propose has four types of Instructions. The details of the instruction type and their encoding are given below

2.1 A-Type

This type of instruction takes as input 3 register numbers of 5 bits each. 6 bit Opcode for all of them is 0 and a 10 bit function code is provided : The opcode and function code for

31-26	**25-21**	**20-16**	**15-11**	**10-0**
Opcode	rs (Register Source)	rt (Register Operand)	rd (Register Destination)	Function

Table 1: A-type Instruction Format

different types of operations are shown below :

Operation	**Opcode**	**Function**	**Instruction**
ADD	000000	0000000001	add $rd, $rs, $rt
SUB	000000	0000000010	sub $rd, $rs, $rt
AND	000000	0000000100	and $rd, $rs, $rt
OR	000000	0000001000	or $rd, $rs, $rt
XOR	000000	0000010000	xor $rd, $rs, $rt
NOT	000000	0000100000	not $rd, $rs
SLA	000000	0001000000	sla $rd, $rs, $rt
SRA	000000	0010000000	sra $rd, $rs, $rt
SRL	000000	0100000000	srl $rd, $rs, $rt
MOVE	000000	1000000000	move $rd, $rs

Table 2: A-type Instructions

2.2 B-Type

This type of instruction takes as input register(s) and immediate value. We further subdivide it in 2 category:

2.2.1 B1-Type

This type of instruction takes as input 1 register number of 5 bits each. 6 bit Opcode for all of them is specified and a 21 bit immediate value is provided

31-26	25-21	20-0
Opcode	rs (Register Source)	Immediate

Table 3: B1-type Instruction Format

The opcode for different types of operations are shown below:

Operation	Opcode	Instruction
ADDI	000001	addi $rs, imm
SUBI	000010	subi $rs, imm
ANDI	000011	andi $rs, imm
ORI	000100	ori $rs, imm
XORI	000101	xori $rs, imm
NOTI	000110	noti $rs, imm
SLAI	000111	slai $rs, imm
SRAI	001000	srai $rs, imm
SRLI	001001	srli $rs, imm
BMI	001010	bmi $rs, imm
BPL	001011	bpl $rs, imm
BZ	001100	bz $rs, imm
LDSP	001101	ldsp sp, imm, $rs
STSP	001110	stsp sp, imm, $rs
PUSH	001111	push $rs
POP	010000	pop $rs

Table 4: B1-type Instructions

2.2.2 B2-Type

This type of instruction takes as input 2 register numbers of 5 bits each. 6 bit Opcode for all of them is specified and a 16 bit immediate value is provided The opcode for different

31-26	25-21	20-16	15-0
Opcode	rs (Register Source)	rt (Register Operand)	Immediate

Table 5: B2-type Instruction Format

types of operations are shown below:

Operation	Opcode	Instruction
LD	010001	ld $rs, imm, $rt
ST	010010	st $rs, imm, $rt

Table 6: B2-type Instructions

2.3 C-Type

This type of instruction takes as input 6 bit Opcode and a 26 bit immediate value is provided:

31-26	25-0
Opcode	Immediate

Table 7: Instruction Format with Opcode and Immediate

The opcode for different types of operations are shown below:

Operation	Opcode	Instruction
BR	010011	br imm
CALL	010100	call imm

Table 8: Branch and Call Instructions

2.4 D-type

This type of instruction takes as input 6 bit Opcode and rest 26 bit is ignored:

31-26	25-0
Opcode	Don't Care

Table 9: Instruction Format with Opcode and Don't Care Field

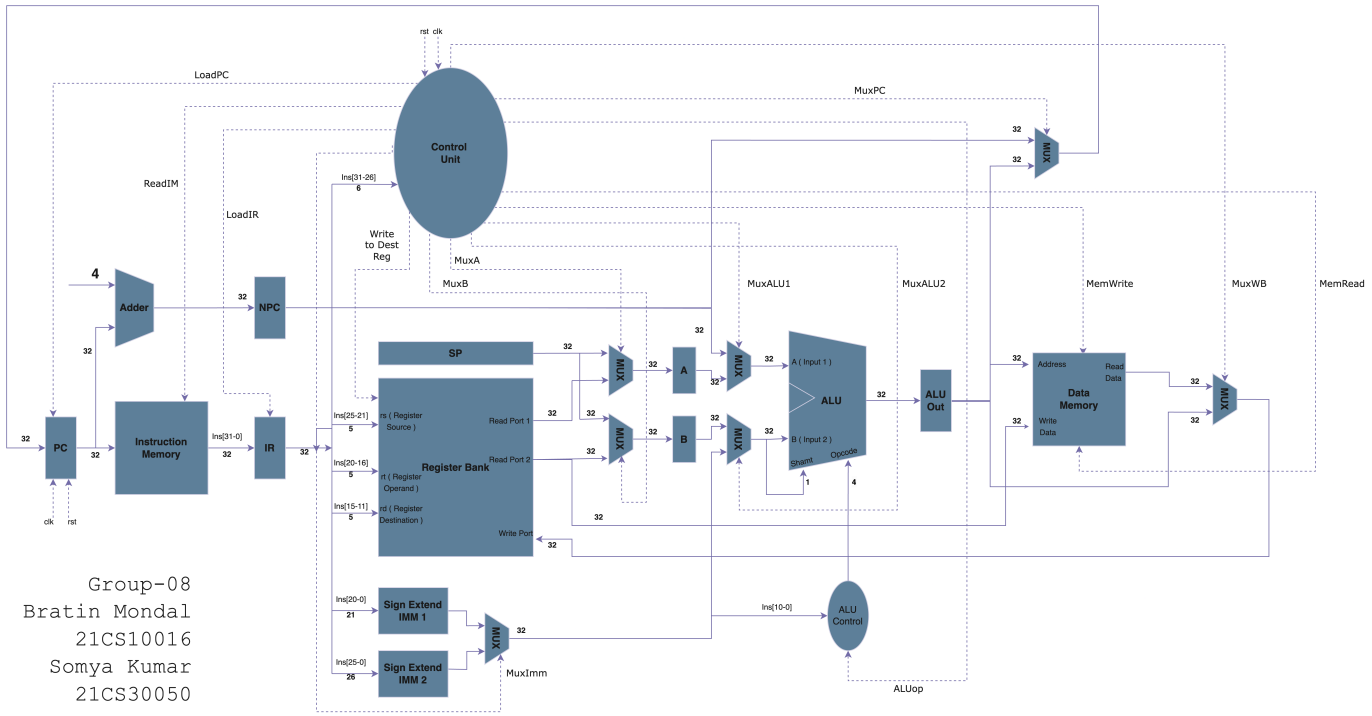
The opcode for different types of operations are shown below:

Operation	Opcode	Instruction
RET	010101	ret
HALT	010110	halt
NOP	010111	nop

Table 10: Control Instructions

3 Datapath

The datapath diagram for our design is shown below:



Click Here

4 Code Design and Implementation:

We have the following Modules in our Code:

4.1 alu.v

The ALU (Arithmetic Logic Unit) module, defined in Verilog, performs arithmetic and logical operations based on the control signals provided. It takes two 32-bit signed inputs and a shift amount (`shamt`), controlled by a 5-bit signal (`control`). The output is a 32-bit result (`out`) along with a 3-bit flag (`flags`) that indicates zero, negative, or carry out status of the operation. The ALU supports addition, subtraction, bitwise AND, OR, XOR, NOT, and shift operations (both logical and arithmetic). The operation is determined by the lower four bits of the control signal, while the most significant bit decides the shift amount source.

4.2 alu_control.v

The alu_control module in Verilog serves as a decoder to generate the control signals for the Arithmetic Logic Unit (ALU) based on the current operation code (`alu_op`) and the

function code (`func_code`). It outputs a 5-bit control signal (`alu_control_signal`) that determines the ALU's behavior. The decoding process involves a combination of nested case statements that select the appropriate output for each combination of `alu_op` and `func_code`. This mechanism allows for a comprehensive set of operations to be encoded, encompassing basic arithmetic and logical operations. A default signal is provided to ensure stability for undefined operation codes.

4.3 branching_mechanism.v

The branching_mechanism module is a crucial component of the processor's control flow, responsible for determining the next program counter (`pc_out`) value based on branching conditions. It takes the current program counter (`pc_in`), a destination address (`dest_addr`), a register value (`reg1`), a branch control signal, an instruction function code (`ins_func_code`), ALU flags (`alu_flag`), a reset signal (`rst`), and the clock (`clk`). On reset, the program counter is set to zero, and the ALU flags are cleared. The branching decision is made based on the branch control signal and function code: it either increments the program counter for sequential execution or updates it to a branch address for conditional branching. The module supports various branch conditions such as negative, non-negative, and zero register values.

4.4 write_data_selector.v

The mem_reg_pc_selector module is designed to select between memory data, register data, and the program counter value for write-back operations in a processor. It uses a 2-bit input (`mem_reg_pc`) to control the selection: when set to '01', memory input (`mem_in`) is selected; '10' selects the program counter input (`pc_in`); and any other value defaults to the register input (`reg_in`). The chosen value is then outputted to `write_data`, which is used for updating the register file or other stateful components.

4.5 register_file.v

The register_file module serves as the central storage for the processor's registers, facilitating read and write operations. It is comprised of seventeen 32-bit registers, where register 16 is designated as the stack pointer, initialized to 1024. The module accepts two read addresses (`reg1Addr` and `reg2Addr`), one write address (`regWriteAddr`), a write control signal (`writeReg`), and the data to be written (`writeData`). On reset (`rst`), all registers are cleared except for the stack pointer. The write control signal determines which register is to be written to during the write operation. The module outputs two register values (`reg1Out` and `reg2Out`) based on the read addresses and also provides the value of register 1 (`retReg`) for quick access.

4.6 data_memory.v

The 'data_memory' module interfaces with a memory unit, referred to as DataRAM, to perform read and write operations. It maps a 32-bit input address to a 12-bit address space

within the memory. The module is clocked and only accesses memory on a clock edge if enabled. When the write enable signal is asserted, the module writes the 32-bit input data to the memory at the specified address. Conversely, it reads data from the memory into a 32-bit output on each clock cycle when write enable is not asserted. The DataRAM is accessed with an inverted clock signal to align with its active edge requirements.

4.7 instruction_fetcher.v

The instruction_fetcher module is responsible for retrieving instructions from the instruction memory, known as InstrROM, using the program counter (`pc`) as an address reference. It takes a 32-bit `pc` as input, which is clocked by the `clk` signal. The module outputs a 32-bit instruction (`instr`), which corresponds to the instruction at the memory location pointed to by the program counter. The lower 12 bits of the `pc` are used as the address, aligning with the addressing capabilities of the InstrROM.

4.8 program_counter.v

The program_counter module is a fundamental component in the processor's control flow mechanism, maintaining the address of the next instruction to be executed. The module updates its output, `pc_out`, on the rising edge of the clock signal (`clk`). If the reset signal (`rst`) is asserted, `pc_out` is set to zero, effectively resetting the program counter. Otherwise, the program counter is loaded with the value from `pc_in`, allowing for both sequential instruction processing and controlled jumps. The simplicity of this module is key to its fast and reliable operation within the CPU.

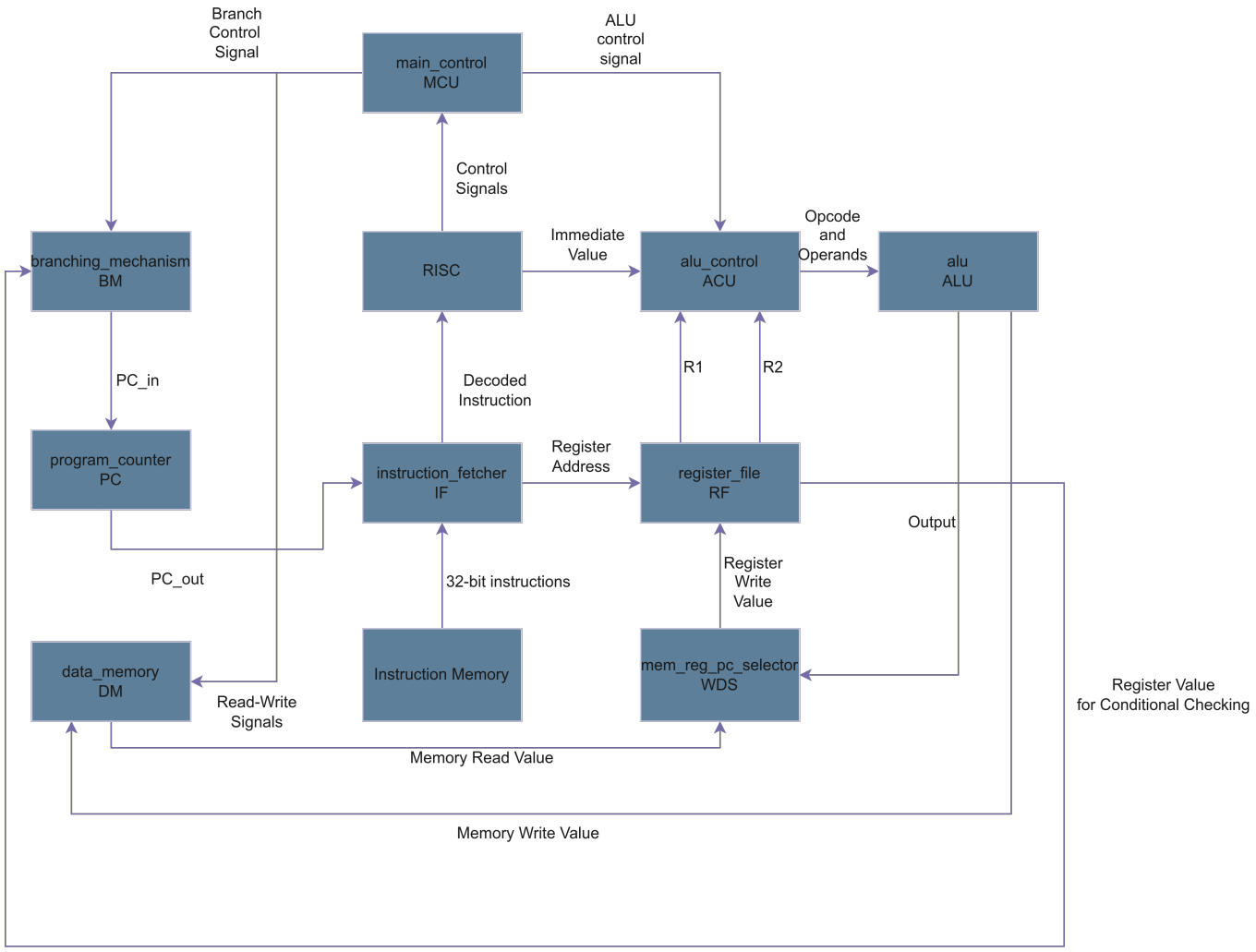
4.9 main_control.v

The main_control module in Verilog acts as the central nervous system of a processor's control unit. It decodes the 6-bit opcode from the instruction and sets the control signals for other components of the processor accordingly. These control signals include a branch flag, memory read and write operations, ALU operation codes, ALU source selection, and register write operations. The control logic ensures the correct execution path for a given instruction, whether it be a computation, memory access, or a control flow modification. The module encapsulates complex control logic within a simple case statement structure, which greatly simplifies the design and operation of the processor.

4.10 RISC.v

The RISC module encapsulates a simplified model of a RISC processor in Verilog, integrating various sub-modules that collectively execute instructions. It includes a program counter (PC), an instruction fetcher (IF), an ALU control unit (ACU), an arithmetic logic unit (ALU), a main control unit (MCU), a memory register/PC selector (WDS), a data memory (DM), and a register file (RF). The processor reads instructions, decodes them, performs arithmetic and logical operations, handles branching, and accesses data memory, all orchestrated by a clock signal (`clk`) and reset (`rst`). The module's design demonstrates

interconnectivity and data flow between components, forming a cohesive unit capable of executing a sequence of instructions.



Click Here

5 Memory

Two types of memory are implemented using BRAM

5.1 Data Memory

The Verilog instantiation of the 'DataRAM' module within a larger memory system utilizes an inverted clock signal and several control signals to manage memory operations. It is parameterized by an enable signal, a write enable signal, an address input, data input for write operations, and provides a data output. The inversion of the clock signal (`clka(~`

clk)) ensures that memory operations are synchronized with the negative edge of the clock cycle, potentially aligning with the specific timing requirements of the memory component.

5.2 Instruction Memory

Within the digital design, the 'InstrROM' is instantiated as a read-only memory module, serving as the instruction memory of the system. It is clocked by a signal clk, which dictates the timing of instruction fetch cycles. The address from which to fetch the instruction is specified by the lower 12 bits of the program counter pc[11:0], and the fetched instruction is outputted on instr. This setup ensures that instructions are sequentially read from memory locations determined by the program counter, facilitating the execution flow of the processor. This needs to be initialized by file with .coe type extension

6 Testbench

The RISC_tb module is designed to verify the functionality of the RISC processor model by simulating its behavior. Within this testbench, the clock (clk) signal is driven by an always block that toggles its state every 2 nanoseconds, creating a 1 GHz clock frequency. The reset (rst) is initially set high to reset the system, then taken low after 10 nanoseconds to begin normal operation. The testbench monitors the return register (retReg) of the RISC processor for changes, which could indicate successful execution of instructions or highlight potential issues. The instantiation of the RISC module within the testbench allows for direct observation and verification of its output signals in response to the input stimuli.

7 Code Parser

The input code needs to be parsed and turned into a **.coe** file to be loaded into the instruction memory

7.1 instruc.json

This JSON object represents an instruction set architecture (ISA) for a processor, specifying various operations such as arithmetic, logical, and branching instructions, along with their corresponding opcode and operand fields. Each key represents an instruction mnemonic, and the array associated with it represents different parts of the instruction encoding, such as opcode, register identifiers, shift amounts, and function codes.

7.2 registers.json

The JSON object defines a set of general-purpose registers, each with a unique identifier and corresponding numerical value. The register mapping is as follows: $0 is always 0, signifying a constant value register. Registers $1 through $15 are used for general purposes and hold values from 1 to 15, respectively. Lastly, $sp is the stack pointer with a designated value of

16, which is used to point to the current top of the stack in memory, playing a critical role in function call management and local variable storage.

7.3 main.py

The Python script serves as an assembler for a custom RISC processor. It reads assembly instructions, converts them to binary representation, and writes the output to a text file. The script handles two's complement conversion for immediate values, comment stripping, and error checking for instruction validity. It supports various instruction formats, including arithmetic, logical, load, store, and branch instructions. Register names are mapped to binary codes using a JSON file, enabling the script to convert mnemonic opcodes to their binary equivalents. The output is formatted to be compatible with memory initialization files used in hardware simulations.

8 FPGA Dumping

The RISC.v file is modified by adding the following section to dump on FPGA board

```
module INTERACT(
    input rst,
    input clk,
    output [15:0] result
);
    wire [31:0] return_reg;
    assign result = return_reg[15:0];
    RISC uut(
        .clk(clk),
        .rst(rst),
        .retReg(return_reg)
    );
    endmodule
```

Listing 1: INTERACT module

8.1 master.xdc

Following pin planning is used for dumping it to FPGA

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];


## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { result[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { result[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { result[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { result[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { result[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { result[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { result[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { result[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { result[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { result[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { result[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { result[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { result[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { result[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { result[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { result[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]


##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { rst }]; #IO_L9P_T1_DQS_14 Sch=btnc
```

Listing 2: master.xdc

9 RTL Micro-operations

The RTL Micro-operations for different types of instructions are shown below

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, ReadRegPort2, MuxA=0, MuxB=0, LoadA, LoadB
ALUfunc = add, MuxALU1=0, MuxALU2=0, LoadALUOut
LoadPC
MuxWB, WriteReg

Table 11: Instructions for ADD operation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, LoadA, LoadImm, MuxA=0, MuxB=1, MuxImm=0
ALUfunc = add, MuxALU1=0, MuxALU2=1, LoadALUOut
LoadPC
MuxWB, WriteReg

Table 12: Instructions for ADDI operation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, LoadA, LoadIMM, MuxA=0, MuxB=1, MuxImm=0
ALUfunc = add, MuxALU1=0, MuxALU2=1, LoadALUOut
LoadPC, MemRead, LoadLMD
MuxWB=0, WriteReg

Table 13: Instructions for LOAD operation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, LoadA, LoadB, LoadIMM, MuxA=0, MuxB=0, MuxImm=0
ALUfunc = add, MuxALU1=0, MuxALU2=1, LoadALUOut
LoadPC, MemWrite

Table 14: Instructions for STORE operation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, LoadA, MuxA=0, MuxB=1
ALUfunc=add, MuxALU1=0, MuxALU2=0, LoadALUOut
LoadPC, MemWrite
MuxA=1, MuxB=1, MuxImm=1, LoadA, LoadB
MuxALU1=0, MuxALU2=1, ALUfunc=add, LoadALUOut, SPin

Table 15: Instructions for PUSH operation with stack pointer manipulation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, LoadImm, MuxA=0, LoadA
ALUfunc = sub, MuxALU1=0, MuxALU2=1, MuxPC=0, LoadALUOut
LoadPC

Table 16: Instructions for BMI operation

Instruction
ReadIM, LoadIR, LoadNPC
ReadRegPort1, ReadRegPort2, MuxA=0, MuxB=0, LoadA, LoadB
MuxALU1=0, LoadALUOut
LoadPC
MuxWB=1, WriteReg

Table 17: Instructions for MOVE operation

10 Conclusion

In conclusion, the development of a 32-bit RISC-like processor in Verilog has been successfully outlined in this report. The designed processor architecture demonstrates efficient computing capabilities through its register bank inclusive of special-purpose registers, a byte-addressable memory interface, and support for multiple addressing modes. These features ensure that the processor is not only adept at handling a variety of computational tasks but also robust in its functionality. The use of Verilog allowed for a detailed and flexible design process, resulting in a processor that can be readily synthesized for FPGA implementation or further extended to accommodate additional features or instruction sets. Future work could explore optimizations for performance, power efficiency, and the integration of more complex instructions to expand the processor's capabilities.