EE3043 Computer Architecture

Semester 241

Design of a Single Cycle RISC-V Processor

Student: Nguyen Nhut Khanh - 2151097

Doan Duy Quan - 2151136

Nguyen Thanh Yen Thy - 2151034

1. Introduction

In Milestone 2, the focus is on deepening the understanding of SystemVerilog and the RV32I instruction set architecture by designing a single-cycle RV32I processor. This involves reviewing the concept and features of SystemVerilog, and the RV32I instruction set, which provides the foundational operations for RISC-V processors. The goal is to construct a soft-core processor that executes RV32I instructions within a single clock cycle. Furthermore, the project involves in interfacing this custom processor with external peripherals, such as 7-segment LEDs, LCD, switches and buttons.

2. Design Strategy

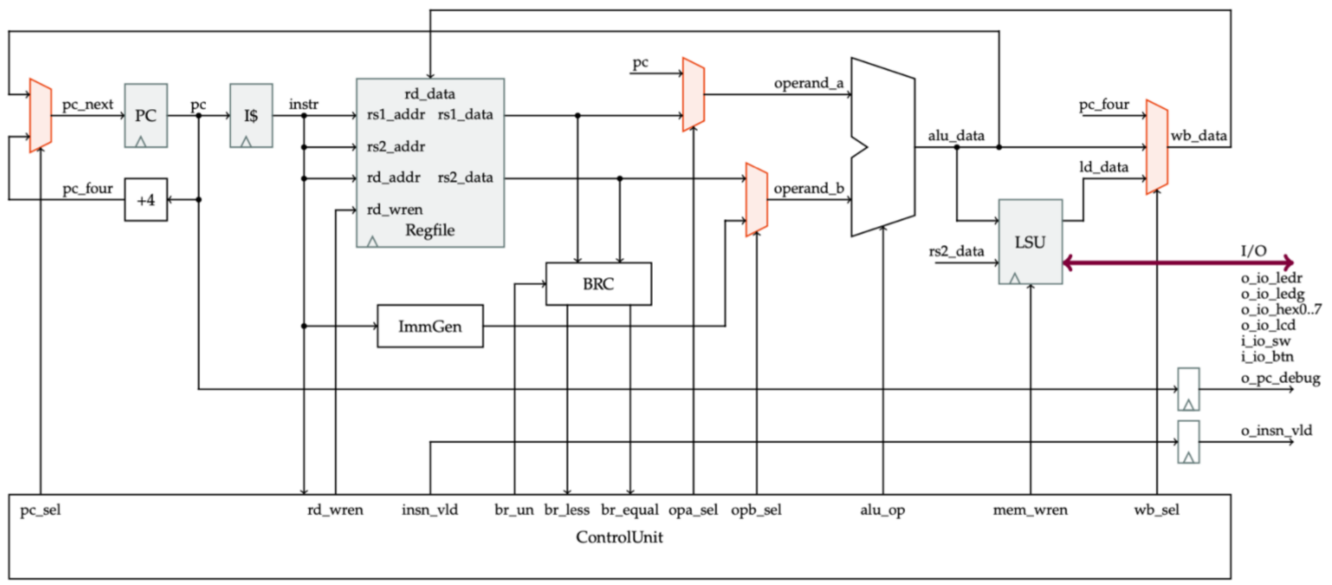
2.1. Overview

The design strategy for developing the single-cycle RV32I processor focuses on creating a processor that executes each instruction in a single clock cycle. In order to implement the single-cycle processor, the strategy involves in building the datapath that incorporates components, which are program counter, instruction memory, register file, Arithmetic Logic Unit (ALU) and memory to guarantee the smooth instruction flow.

The control unit is designed to generate the essential signals and manage the datapath's operation. Mechanisms for efficient reading and writing from memory and the register file are integrated to support memory operations within the single-cycle design. Interfaces for external peripherals are implemented, allowing the processor to communicate with I/O devices.

The top module is **singlecycle**. It includes all the sub-modules and ensures the connectivity between all of them to make the instruction operate correctly.

Signal Name	**Width**	**Direction**	**Description**
`i_clk`	1	input	Global clock, active on the rising edge.
`i_rst_n`	1	input	Global low active reset.
`o_pc_debug`	32	output	Debug program counter.
`o_insn_vld`	1	output	Instruction valid.
`o_io_ledr`	32	output	Output for driving red LEDs.
`o_io_ledg`	32	output	Output for driving green LEDs.
`o_io_hex0..7`	7	output	Output for driving 7-segment LED displays.
`o_io_lcd`	32	output	Output for driving the LCD register.
`i_io_sw`	32	input	Input for switches.
`i_io_btn`	4	input	Input for buttons.



Single Cycle Processor Block Diagram

2.2. Arithmetic Logical Unit (ALU)

The **alu** module must be able to execute a variety of logical and arithmetic operations in RV321 required instruction set.

alu_op	**opcode**	**Description (R-type)**	**Description (I-type)**
ADD	0000	$rd \leftarrow rs1 + rs2$	$rd \leftarrow rs1 + imm$
SUB	1000	$rd \leftarrow rs1 - rs2$	*n/a*
XOR	0100	$rd \leftarrow rs1 \oplus rs2$	$rd \leftarrow rs1 \oplus imm$
OR	0110	$rd \leftarrow rs1 \vee rs2$	$rd \leftarrow rs1 \vee imm$
AND	0111	$rd \leftarrow rs1 \wedge rs2$	$rd \leftarrow rs1 \wedge imm$
SLL	0001	$rd \leftarrow rs1 << rs2[4:0]$	$rd \leftarrow rs1 << imm[4:0]$
SRL	0101	$rd \leftarrow rs1 >> rs2[4:0]$	$rd \leftarrow rs1 >> imm[4:0]$
SRA	1101	$rd \leftarrow rs1 <<< rs2[4:0]$	$rd \leftarrow rs1 <<< imm[4:0]$
SLT	0010	$rd \leftarrow (rs1 < rs2)?\ 1 : 0$	$rd \leftarrow (rs1 < imm)?\ 1 : 0$

SLTU	0011	*rd ← (rs1 < rs2)? 1 : 0*	*rd ← (rs1 < imm)? 1 : 0*

2.2.1. *Specification*

Signal	**Width**	**Direction**	**Description**
`i_operand_a`	32	input	First operand for ALU operations.
`i_operand_b`	32	input	Second operand for ALU operations.
`i_alu_op`	4	input	The operation to be performed.
`o_alu_data`	32	output	Result of the ALU operation.

2.2.2. *Explanation*

The **alu** module performs a variety of arithmetic and logical operations based on the RV32I instruction set, with defined opcode shown on the above table. The module takes in two operands (i_operand_a and i_operand_b) and a 4-bit operation code (i_alu_op), which determines the operation to perform. The output (o_alu_data) holds the result of the operation, while (o_insn_vld) is an output signal indicating the validity of the instruction.

Moreover, built-in SystemVerilog operators for subtraction (−), comparison (<, >), or shifting (<<, >>, and >>>) are not allowed to use. Therefore, we built two more add-on modules: **add_sub**, **barrel_shifter** and **ksa_adder**.

The ALU supports the following operations:

- Addition and Subtraction (A_ADD, A_SUB): These operations are executed by an **add_sub** module, which performs addition or subtraction based on the Cin control signal which is defined by the 3rd bit of A_ADD and A_SUB opcode (i_alu_op[3] signal).

- Set If Less Than (A_SLT, A_SLTU): The brc module performs comparisons, setting the (blarger) signal if the comparison conditions are met. The least significant bit of o_alu_data will be 1 and other bits will be 0.
- Bitwise Logical (A_XOR, A_OR, A_AND): These operations are performed directly using bitwise logic.
- Shift (A_SLL, A_SRL, A_SRA): The **barrel_shifter** module executes these shift operations, with the shift amount determined by the lower 5 bits of i_operand_b. The barrel shifter includes a series of multiplexers, arranged in multiple stages where each stage shifts by an increasing power of two and performs several types of shifters by setting selection signal.

2.3. Branch Comparison Unit (BRC)

The **brc** module is responsible for comparing two registers to determine the outcome of branch instructions. This unit should be capable of handling both signed and unsigned comparisons.

2.3.1. Specification

Signal	**Width**	**Direction**	**Description**
`i_rs1_data`	32	input	Data from the first register.
`i_rs2_data`	32	input	Data from the second register.
`i_brc_un`	1	input	Comparison mode (1 if signed, 0 if unsigned).
`o_brc_less`	1	output	Output is 1 if *rs1* < *rs2*.

`o_brc_equal`	1	output	Output is 1 if *rs1* = *rs2*.

2.3.2. *Explanation*

Two signals (i_rs1_data) and (i_rs2_data) are the two binary numbers to be compared. The brc module supports both signed and unsigned comparison based on (i_brc_un) signal. Since built-in comparison operators (<, >) are not allowed to use, the module uses an external **comparator** module to perform the actual comparison.

In comparator module, the data is divided into upper and lower halves, then compared. The **comparator** module is recursive, meaning it divides the comparison process into smaller pieces until it reaches a base case of 1-bit comparisons.

The output (o_brc_less) and (o_brc_equal) will be determined based on the output (less), (equal) of the **comparator** module.

2.4. Regfile

The requirement is to implement a register file with 32 registers, each 32-bit wide. The register file must have two read ports and one write port, with register 0 always reading as zero.

2.4.1. *Specification*

Signal	**Width**	**Direction**	**Description**
`i_clk`	1	input	Global clock.
`i_rst_n`	1	input	Global active reset.
`i_rs1_addr`	5	input	Address of the first source register.
`i_rs2_addr`	5	input	Address of the second source register.
`o_rs1_data`	32	output	Data from the first source register.

`o_rs2_data`	32	output	Data from the second source register.
`i_rd_addr`	5	input	Address of the destination register.
`i_rd_data`	32	input	Data to write to the destination register.
`i_rd_wren`	1	input	Write enable for the destination register.

2.4.2. *Explanation*

The **regfile** module contains a series of 32 registers with register 0 is always zero. It has two 5-bit input addresses (i_rst1_addr) and (i_rst2_addr) to specify source registers for reading, and a 5-bit address (i_rd_addr) and 32-bit data input (i_rd_data) for writing to a destination register, controlled by a write-enable signal (i_wr_en).

When the reset signal (i_rst_n) is active low, all the registers are set to zero. Whereas, on the positive clock edge, when the signal (i_wr_en) is high, the data in (i_rd_data) is written to the register by the specific address determined by (i_rd_addr) signal, except for the register 0, which is always set to zero.

The data at addresses (i_rst1_addr) and (i_rst2_addr) are output to (o_rst1_data) and (o_rst2_data), respectively. The assign statement ensures that the values from the registers are continuously provided on the outputs without requiring a clock edge to update.

2.5. Load-Store Unit (LSU)

2.5.1. *I/O System and Memory Mapping*

In real-world applications, processors interact with peripheral devices through an Input/Output (I/O) System to send and receive data. Common peripherals, such as LEDs, LCDs, and switches, are treated as special forms of “memory” or “registers”.

Memory mapping is an essential strategy for organizing memory layout, designating regions for specific functions.

2.5.2. Requirements

Boundary Address	**Mapping**
0x7820 -- 0xFFFF	(Reserved)
0x7810 -- 0x781F	Buttons
0x7800 -- 0x780F	Switches *(required)*
0x7040 -- 0x70FF	(Reserved)
0x7030 -- 0x703F	LCD Control Registers
0x7020 -- 0x7027	Seven-segment LEDs
0x7010 -- 0x701F	Green LEDs *(required)*
0x7000 -- 0x700F	Red LEDs *(required)*
0x4000 -- 0x6FFF	(Reserved)
0x2000 -- 0x3FFF	Data Memory (8KiB using SDRAM) *(required)*
0x0000 -- 0x1FFF	Instruction Memory (8KiB) *(required)*

Table of LSU memory mapping specifications

Signal Name	**Width**	**Direction**	**Description**
`i_clk`	1	input	Global clock, active on the rising edge.
`i_rst_n`	1	input	Global active reset.
`i_lsu_addr`	32	input	Address for data read/write.
`i_st_data`	32	input	Data to be stored.
`i_lsu_wren`	1	input	Write enable signal (1 if writing).
`instr`	7	input	Instruction type of load and store.

`i_io_sw`	32	input	Inputs from switches.
`i_io_btn`	4	input	Inputs from buttons.
`i_lsu_op`	2	input	Type of load operations.
`o_ld_data`	32	output	Data read from memory.
`o_io_ledr`	32	output	Output for red LEDs.
`o_io_ledg`	32	output	Output for green LEDs.
`o_io_hex0..7`	7	output	Output for 7-segment displays.

Table of LSU module's signals

2.5.3. Explanation

The **lsu** module divides the address space into regions, enabling it to distinguish between different types of memory. Based on the (i_lsu_addr), it can determine to read or write data to the respective peripheral to the input_buffer and output_buffer sub-modules.

The (o_ld_data) output is assigned based on (i_lsu_op), which specifies the type of load operation: LB (load byte), LH (load half-word), LW (load word), LBU (load byte unsigned), and LHU (load half-word unsigned).

The (new_data_in) signal is assigned based on (i_lsu_op), which specifies the type of store operation: SB (store byte), SH (store half-word), SW (store word).

The module interfaces with external SRAM through an instantiated sram_IS61WV25616_controller_32b_3lr controller module.

2.6. Immediate Generator

This immgen module interprets different instruction formats in the RISC-V architecture and extracts the appropriate immediate values from each.

2.6.1. Requirements

Signal Name	Width	Direction	Description
`i_instr`	32	input	Input instruction.
`o_imm`	output	input	Immediate value extracted from the instruction

2.6.2. Explanation

Local parameters define different instruction formats based on specific opcode patterns in instr_i[6:2], allowing the module to identify the instruction type:

- I_FORMAT_CALCULATE and I_FORMAT_LOAD: Instructions with I-format (e.g., arithmetic with immediates, loads).
- S_FORMAT: Instructions with S-format (e.g., stores).
- B_FORMAT: Instructions with B-format (e.g., conditional branches).
- U_FORMAT_LUI and U_FORMAT_AUIPC: U-format instructions for LUI and AUIPC.
- J_FORMAT_JAL and J_FORMAT_JALR: J-format instructions for JAL and JALR.

2.7. Control Unit

The **ctrl_unit** module interprets the current instruction (instr) and generates control signals for the processor's datapath, including selecting program counter updates, enabling writes to the register file, and determining ALU operations.

2.7.1. *Requirements*

Signal Name	**Width**	**Direction**	**Description**
`instr`	32	input	The current instruction.
`br_less`	1	input	The "less" result from BRC.
`br_equal`	1	input	The "equal" result from BRC.
`pc_sel`	1	output	Select signal between PC+4 and ALU.
`br_un`	1	output	Set to 1 if two operands are unsigned.
`rd_wren`	1	output	Set to 1 if writing to regfile, otherwise 0.
`opa_sel`	1	output	Choose between rs1_data or PC.
`opb_sel`	1	output	Choose between rs2_data of imm.
`alu_op`	4	output	The operation that ALU must perform.
`mem_wren`	1	output	Set to 1 if writing to LSU, otherwise 0
`wb_sel`	2	output	Choose between alu_data(0), ld_data(1) or pc_four(2) to write back to rd_data of regfile
`insn_vld`	1	output	Instruction valid

2.7.2. *Explanation*

The **ctrl_unit** module generates the control signals required for various RISC-V instruction formats. It decodes the instruction, sets ALU operations, determines the source for the ALU operands, enables or disables writes to memory and the register file, and handles program counter updates for branches and jumps. The control unit identifies different instruction formats (IR_FORMAT, I_FORMAT_LOAD, S_FORMAT, B_FORMAT, U_FORMAT, and J_FORMAT) based on the opcode bits instr[6:0].

2.8. Instruction Memory

3. Verification Strategy

In order to test whether all the modules operate accurately, we must create a comprehensive testbench. For each module, directed test cases are developed to validate the functionality, covering all possible operations and edge cases. Finally, generating waveforms to guarantee the correct operations of each module.

4. Application

Input 3 2-D coordinates of A, B, and C. Determine which point, A or B, is closer to C, using LCD as the display.

5. Evaluation

6. Result