Vietnam National University-Ho Chi Minh City
Ho Chi Minh City University of Technology
Faculty of Electrical-Electronic Engineering

EE4423 Computer Organization and Design
Semester 241

Design of a Pipeline RISC-V Processor

Instructor:	PhD. Tran Hoang Linh	
Class:	TT01	
Students:	Tran Quang Nguyen Anh	2151047
	Van Thien Lam	2151110
	Chu Thanh Dong	2151065

Contents

1 Introduction

1.1 Overview

This project details the design and implementation of a pipelined RISC-V processor core using the RV32I instruction set. The core is organized into distinct pipeline stages, including instruction fetch, decode, execute, memory access, and write-back. Our design incorporates a datapath, control unit, and memory interface, optimized for efficient execution of RV32I instructions. To validate functionality, we tested the pipelined core by running various applications, demonstrating its performance and correctness. This report outlines the design process, pipeline optimization strategies, and results, showcasing the core's efficiency and potential for practical applications. Using blocks of milestone 2 again and designing new blocks to design pipeline processor.

1.2 Block diagram

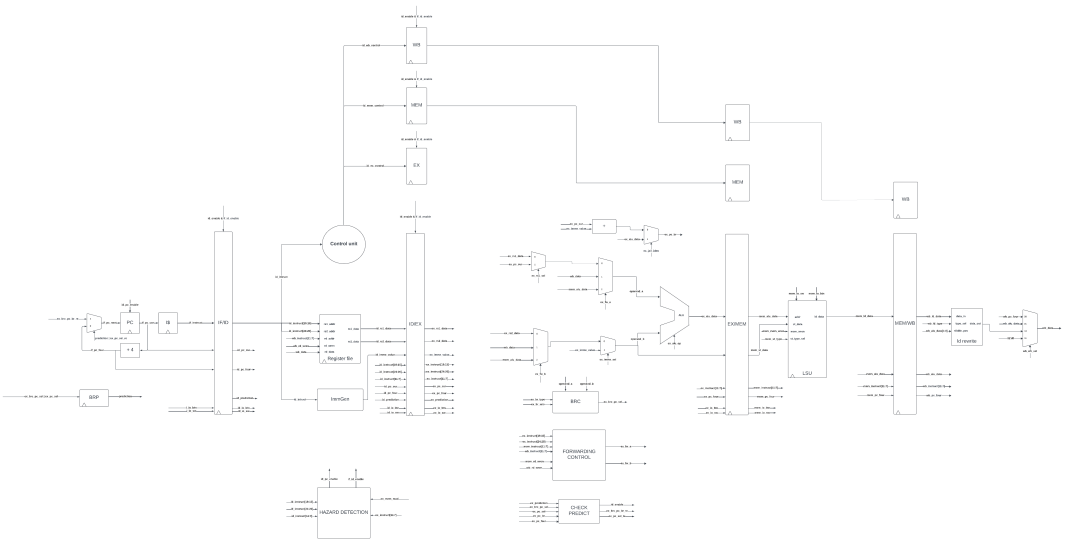

Figure 1: Pipeline's overall architecture

1.3 RV32I

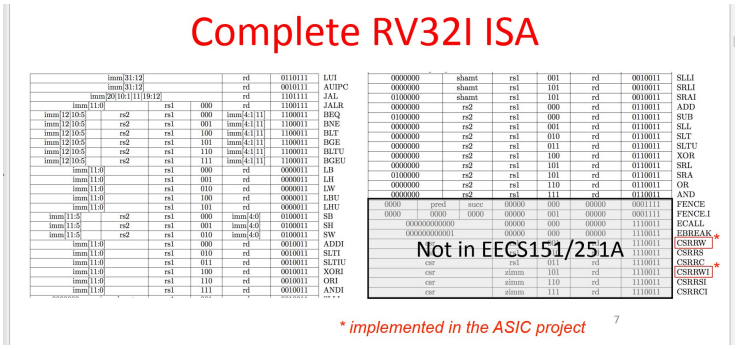

Figure 2: RV32I ISA

1.4 Implementation on board

Implementing on the Altera DE2 board: EP2C325F672C6.

Run assembly programs for testing: Hex to decimal on 7 led segments (16 bits).

2 Design strategy

2.1 Fetch stage (IF)

The Fetch stage is the first step in the pipeline of a processor. Its primary responsibility is to retrieve the next instruction from program memory (Instruction Memory) based on the value of the Program Counter (PC) and prepare it for subsequent pipeline stages.

2.1.1 Program counter (PC)

I/O port:

Signal name	Width	Direction	Description
i_clk	1	Input	Global clock
i_rst	1	Input	Global reset
i_pc_enable	1	Input	PC enable update data
i_pc	32	Input	PC address input
o_pc	32	Output	PC address output

Table 1: PC signals.

Design:

Update pc (instruction address) at each positive edge clock. Furthermore, the next pc is equal to pc + 4 (byte addressing due to the bus in processor is 32 bits).

2.1.2 Instruction memory (I$)

I/O port:

Signal name	Width	Direction	Description
i_addr	32	Input	Instruction memory address
o_data	32	Output	Instruction data

Table 2: Instruction memory signals.

Design:

Create a array of 2048 registers for 8kB instruction memory (32 bits = 4 Byte => 8kB is nearly 4 * 2000).

2.1.3 Branch prediction (BRP)

I/O port:

Signal name	Width	Direction	Description
i_clk	1	Input	Global clock
i_rst	1	Input	Global reset
i_actual_branch_taken	1	Input	Actual branch outcome
o_prediction	1	Output	Predicted outcome

Table 3: BRP signals.

Design:

We use concept 2-bit branch prediction to predict the branch or not branch. In this concept, we have a FSM with 4 states: Strongly not taken, Weakly not taken, Weakly taken and Strongly taken.

At reset is on, our *current_state* always is at Weakly taken, the next state of *current_state* depends on *i_actual_branch_taken*. The state cases (*current_state*):

+ Strongly not taken: if *i_actual_branch_take* = 1, *current_state* = Weakly not taken, else *current_state* = Strongly not taken.

+ Weakly not taken: if *i_actual_branch_take* = 1, *current_state* = Weakly taken, else *current_state* = Strongly not taken.

+ Weakly taken: if *i_actual_branch_take* = 1, *current_state* = Strongly taken, else *current_state* = Weakly not taken.

+ Strongly taken: if *i_actual_branch_take* = 1, *current_state* = Strongly taken, else *current_state* = Weakly taken.

2.2 Decode stage (ID)

The Decode stage is the second step in a processor's pipeline. Its primary function is to interpret the fetched instruction, identify the operation to be performed, and prepare the necessary operands and control signals for the subsequent stages.

2.2.1 Register file (Regfile)

I/O port:

Signal name	Width	Direction	Description
i_clk	1	Input	Global clock
i_rst	1	Input	Global reset
i_rd_wren	1	Input	Enable write task signal
i_rs1_addr	5	Input	Address of first source register
i_rs2_addr	5	Input	Address of second source register
i_rd_addr	5	Input	Address of destination register
i_rd_data	32	Input	Data written to destination register
o_rs1_data	32	Output	Data of first source register
o_rs2_data	32	Output	Data of second source register

Table 4: Register file signals.

Design:

Create 32 purpose registers, each register will update the new value at the positive edge clock (except register *x0*: reserved register) with a signal *rd_wren* for identifying and enabling write tasks.

Furthermore, the read task will be asynchronous, it means that the data outputs: *rs1_data* and *rs2_data*, will be always read data from the registers: *rs1_addr*, *rs2_addr*.

2.2.2 Immediate generator (ImmeGen)

I/O port:

Signal name	Width	Direction	Description
i_instruction	32	Input	Instruction
o_imme_value	32	Output	Immediate value output

Table 5: ImmGen signals.

Design:

This is a signed extended block for instructions that provide an immediate value in 12 or 20 bits. But in our design, the width of the calculation signals are 32 bits.

The mode of this module is corresponding to the *imm* in RV32I, and step of module execution:

+ Arranging the *imm* in instruction to the order of temporary label *temp[19:0]*. With instructions that provide 12-bit *imm*, the rest of the bit is 0.

+ Then check the signed bit of arranged *imm* to extend. (*imm[19]* for JAL,LUI,AUIPC and *imm[11]* for others)

+ Extend the *imm* with all the rest of the bits are the same as the signed bit at the previous step. (12 bits for JAL,LUI,AUIPC and 20 bits for others)

Note: For BR-type instruction the value output from the ImmGen module will be shifted left 1 bit and for LUI and AUIPC will be shifted left 12 bits before going out.

2.2.3 Control unit (CTRL)

I/O port:

Signal name	Width	Direction	Description
i_instruction	32	Input	Instruction
o_pc_sel	1	Output	Selection PC(branch or next)
o_rd_wren	1	Output	Destination register write enable
o_br_uns	1	Output	Branch unsigned identify
o_insn_vld	1	Output	Instruction valid
o_mem_rden	1	Output	Mem read enable
o_mem_wren	1	Output	Mem write enable
o_imme_sel	1	Output	Immediate value selection
o_rs1_sel	1	Output	PC value selection in EX stage
o_jalr_iden	1	Output	Identify JALR & JAL
o_alu_op	4	Output	Mode ALU
o_ld_rewrite	3	Output	Rewrite type for LD instruction
o_br_type	3	Output	BR instruction type
o_st_rewrite	2	Output	Rewrite type for ST instruction
o_wb_sel	2	Output	Choose data write to destination register

Table 6: Control unit signals.

Design:

The control unit module is designed to decode the instruction opcode and generate the necessary control signals for each type of instruction.

The design uses a decoder architecture, which examines the opcode to identify the instruction type, with predefined parameter values for each type. The control signals are generated in an always block with a case statement, where specific values are assigned to control each unit within the processor based on the decoded instruction.

2.2.4 Hazard detection

I/O port:

Signal name	Width	Direction	Description
i_ex_read	1	Input	Read enable signal at EX stage
i_id_rs1	5	Input	Rs1's address at ID stage
i_id_rs2	5	Input	Rs2's address at ID stage
i_id_rd	5	Input	Rd's address at ID stage
i_ex_rd	5	Input	Rd's address at EX stage
o_pc_enable	1	Output	PC enable
o_if_id_register_enable	1	Output	IF_EX register enable

Table 7: Hazard detection signals.

Design:

The hazard detection stalls the processor for one clock to wait for the data which comes to WB stage to forward to EX stage. In this block, we only stall processor in case: Write/Use after Read.

We will check the *RD'address* at EX stage to *RS1's address* and *RS2's address* at ID stage to check that the after LOAD instructions, the destination register is needed for calculating at next instruction.

2.3 Execute stage (EX)

The Execute stage is the third stage in a processor's pipeline. Its primary role is to perform the actual computation or operation specified by the decoded instruction, such as arithmetic, logic, or address calculation.

2.3.1 Branch comparator (BRC)

I/O port:

Signal name	Width	Direction	Description
i_br_uns	1	Input	Identify unsigned BR instruction
i_br_type	3	Input	Branch instruction type
i_rs1_data	32	Input	First operand for computing
i_rs2_data	32	Input	Second operand for computing
o_pc_sel	1	Output	PC address selection

Table 8: BRC signals.

Design:

This module compares the *operand_a* and *operand_b* to get conditions for the processor that can jump to another address.

Instruction	i_br_unsigned	o_br_less	o_br_equal	Description
BEQ	0	x	1|0	$(rs1{=}rs2)$? 1 : 0
BNE	0	x	0	$(rs1{!=}\ rs2)$? 1 : 0
BLT	0	1|0	0	$(rs1{<}rs2)$? 1 : 0
BGE	0	0	x	$(rs1{>=}rs2)$? 1 : 0
BLTU	1	1|0	0	$(rs1{<}rs2)$? 1 : 0
BGEU	1	0	x	$(rs1{>=}rs2)$? 1 : 0

Table 9: Output BRC case.

2.3.2 ALU

I/O port:

Signal name	Width	Direction	Description
i_alu_op	4	Input	Selection mode
i_operand_a	32	Input	First operand for computing
i_operand_b	32	Input	Second operand for computing
o_alu_data	32	Output	Output of computing

Table 10: ALU signals.

ALU operations:

Design:

ALU is a block that execute the computation of instrustion such as add, sub, shift logical and arithmetical,...

Without using '-' operator, we use 2's complement of a number to generater:
$o\_alu\_data = i\_operand\_a + !i\_operand\_b + 1$

Furthermore, for shift operators, we design a special block for executing with a optimized used resources for synthesizing and simulating on board. The algorithm is executed following below steps by steps(take shift logical right as an example):

+ Store the value to a temporary label *temp*.

i_alu_op	Operation	Description
4'h0	ADD	*o_alu_data=i_operand_a+i_operand_b*
4'h1	SUB	*o_alu_data=i_operand_a-i_operand_b*
4'h2	SLT	*o_alu_data=(i_operand_a<i_operand_b)*?1:0
4'h3	SLTU	*o_alu_data=(i_operand_a<i_operand_b)*?1:0
4'h4	XOR	*o_alu_data=i_operand_a⊕i_operand_b*
4'h5	OR	*o_alu_data=i_operand_a|i_operand_b*
4'h6	AND	*o_alu_data=i_operand_a&i_operand_b*
4'h7	SLL	*o_alu_data=(i_operand_a)<<i_operand_b[4:0]*
4'h	SRL	*o_alu_data=(i_operand_a)>>i_operand_b[4:0]*
4'h9	SRA	*o_alu_data=(i_operand_a)>>>i_operand_b*
4'hA	LUI	*o_alu_data=i_operand_b*

Table 11: ALU signals.

+ Check each bit of the *i_operand_b[4:0]*.If any bit of that is 1, the *temp* will be shifted right with *i_operand_b[n]* * 2^n bits by conscast and stored into *temp*. On contrary, unchanged.

+ Repeat the previous step untill checking all 5 bits of *i_operand_b[4:0]*. And output *o_alu_data* = *temp* final.

Comment: With this design, we decrease the number of muxes with case to 5 muxes. Success optimization for synthesizing and increasing the speed of processor. Moreover, we also add a special mode for LUI introduction. The output will equal to the second operand which is shifted in ImmGen.

The differences with single-cycle design are that we have separated PC branch addresses of BR instructions and JAL & JALR instructions to 2 parts. Furthermore, we also add 2 muxes for forwarding before going to ALU block. The orders of 2 operands are in the below figure:

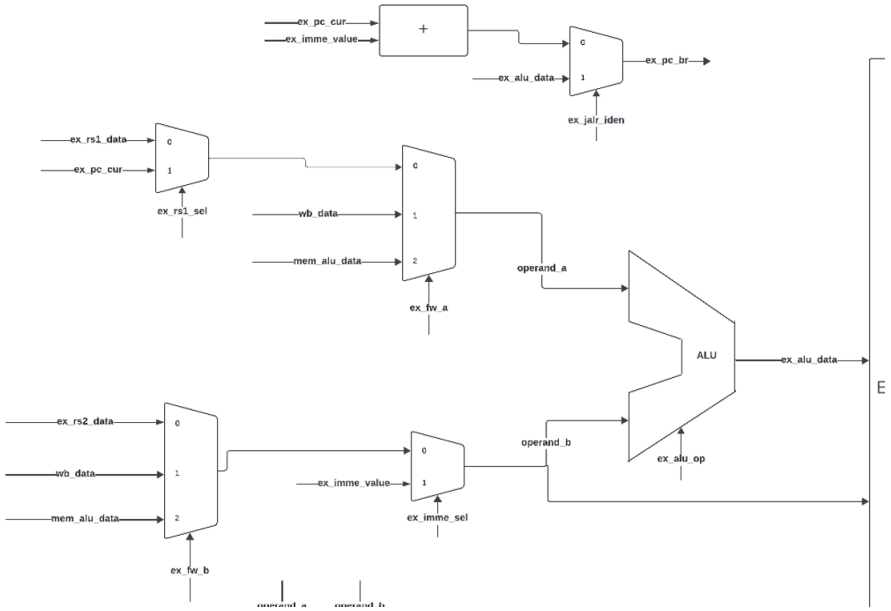

Figure 3: Flows of 2 operands before going to ALU

2.3.3 Forwarding control

I/O port:

Design:

This block controls the data at MEM stage or WB stage that will be forwarded to EX stage (solve the data hazard in pipeline architecture).

Signal name	Width	Direction	Description
i_rd_wren_wb	1	Input	rd_wren signal at WB stage
i_rd_wren_mem	1	Input	rd_wren signal at MEM stage
i_rs1_addr	5	Input	Rs1's address at EX stage
i_rs2_addr	5	Input	Rs2's address at EX stage
i_mem_rd_addr	5	Input	Rd's address at MEM stage
i_wb_rd_addr	5	Input	Rd's address at WB stage
o_forwarding_a	2	Output	Forwarding signal for operand_a
o_forwarding_b	2	Output	Forwarding signal for operand_b

Table 12: Control unit signals.

o_fw_a|b	Forwarding type
2'h0	None forwarding
2'h1	Forward from WB stage
2'h2	Forward from MEM stage

Table 13: Mode forwarding.

Furthermore, to reduce pipeline latency, ensure the validity and timeliness of data and optimize the flow of data, the level of priorities to check that is from *2'h2* to *2'h0*. The condition for each case:

+ None forwarding: *Rs1's address* and *Rs2's address* are not common in *Rd's address* at WB and MEM stage and stored to register file.

+ Forward form WB stage: *Rs1's address* and *Rs2's address* are common in *Rd's address* at WB stage (not the same as MEM stage) and stored to register file.

+ Forward from MEM stage: *Rs1's address* and *Rs2's address* are common in *Rd's address* at MEM stage and stored to register file.

2.3.4 Check predict

I/O port:

Signal name	Width	Direction	Description
i_ex_prediction	1	Input	Prediction at EX stage
i_ex_brc_pc_sel	1	Input	Output of BRC block
i_ex_pc_sel	1	Input	Output PC sel signal from control unit at EX stage
i_ex_pc_four	32	Input	PC+4 at EX stage
i_ex_pc_br	32	Input	PC branch address
o_id_enable	1	Output	Flush signals
o_ex_pc_re	1	Output	Resolve wrong predict signal
o_ex_pc_br_re	32	Output	PC address branch recheck

Table 14: Check predict signals.

Design:

The check predict block plays the role that checks the predict is right or not. In this block, it will compare the *i_ex_prediction* to two signals: *i_ex_brc_pc_sel* and *i_ex_pc_sel* if signal *i_ex_prediction* is common in one of 2 above signals, prediction right. Else, predict is wrong. When wrong, the processor is flush or unenable PC, IF_ID register, ID_EX register. And then set the *pc_next* is always taken the *o_ex_pc_br_re* to jump to the right address.

2.4 Memory stage (MEM)

The Memory stage is the fourth stage in a processor's pipeline. Its primary function is to handle memory-related instructions, such as loading data from memory (load) or storing data to memory (store). For non-memory instructions, the results from the Execute stage are simply forwarded to the next stage.

2.4.1 Load and Store unit (LSU)

I/O port:

Signal name	Width	Direction	Description
i_clk	1	Input	Global clock
i_rst	1	Input	Global reset
i_mem_wren	1	Input	Write mem enable
i_st_rewrite	2	Input	Mode store type
i_io_btn	4	Input	Button's data
i_io_sw	32	Input	Switch's data
i_alu_data	32	Input	Address
i_st_data	32	Input	Data stored
o_io_ledr	32	Output	LED red output
o_io_ledg	32	Output	LED green output
o_hex_data_1	32	Output	Data of HEX[3:0]
o_hex_data_2	32	Output	Data of HEX[7:4]
o_io_lcd	32	Output	LCD control output
o_ld_data	32	Output	LOAD data

Table 15: LSU signals.

Design:

In milestone 3, we decide that we don't use the SRAM instead of Sync-ram. With sync-ram, we trade off latency and used memory bit. No latency, high used memory bit with sync-ram and high latency, no used memory bit with SRAM. We use back the resize memory mapping in milestone 2. So we divide the LSU into 3 parts: input memory (9 registers), output memory (16 registers) and data memory (2048 registers).

Furthermore, to get the output of IO signals, it will be late one clock. It means that after taking data of *output_peripheral[addr_shifted[3:0]]* (with output_peripheral is output memory and addr_shifted is i_alu_data shift right 2 bits), after one clock, the output IO will update data corresponding to correct registers. The reason why we have to update after one clock is that we set the read task is at positive edge and write task is at negative edge to execute STORE functions(read then resize and write). The read task will be at the same as with EX_MEM register enable, so overlapping the address, no data output.

2.4.2 Store rewrite

I/O port:

Signal name	Width	Direction	Description
i_lsu_addr_segment	2	Input	Bite position
i_st_type	2	Input	Mode store type
i_ld_data	32	Input	Load data
i_st_data	32	Input	Store data
o_st_new_data	32	Output	Store data after rewriting

Table 16: ST rewrite signals.

Design:

Store rewrite block rewrites the store data input from EX stage before writing to LSU. This block is designed to solve the problem that is how executing SB, SH instructions. The steps of executing this block are: 1) taking data from memory, 2) rewriting the store data input to data at step 1) and 3) writing to memory back (this is why Read needs to do before Write).

i_st_type	ST type description
2'h0	SB (8 bits write)
2'h1	SH (16 bits write)
2'h2	SW (32 bits write)
2'h3	Reserved value

Table 17: Mode LD type.

2.5 Write back stage (WB)

The Write-Back stage is the final stage in a processor's pipeline. Its primary function is to update the destination register with the result of the operation performed in previous stages, completing the execution of the instruction.

2.5.1 LD rewrite

I/O port:

Signal name	Width	Direction	Description
i_segment_lsu_addr	2	Input	Byte position change
i_rewrite_sel	3	Input	Mode choose type of LD
i_ld_data	32	Input	Old load data from LSU
o_new_ld_data	32	Output	Output new LD data

Table 18: LD rewrite signals.

Mode of LD rewrite:

i_rewrite_sel	LD type description
3'h0	LB (8 bits with signed extended)
3'h1	LH (16 bits with signed extended)
3'h2	LW (32 bits load)
3'h3	LBU (8 bits without signed extended)
3'h4	LHU (16 bits without signed extended)
3'h5	Reserved value

Table 19: Mode LD type.

Design:

Rewrite the load data from LSU to execute LB, LH, LBU, LHU. To identify the byte position that will be loaded, we use the first 2 bits of address.

+ For LB, LBU: value of address[1:0] corresponding to the byte loaded.
+ For LH, LHU: value of address[1] corresponding to 16-bit high (1) or 16-bit low (0).

3 Verification strategy

Following the type of instruction, we decide that the part of verification process of the project is divided into 3 parts: test individual block, test with each type of instruction with a completing design and test with a full assembly program.

With these steps, we can modify what module is error or which part has the wrong connect without recheck all design.

Furthermore, we also use *$display* and *$monitor* to watch the result of signals at a tme or at when the signals change in testbench file.

4 Result

4.1 Quartus result

Figure 4: Compilation report.

The RISC-V single-cycle processor design was successfully synthesized on the Altera Cyclone II (EP2C35F672C6) FPGA using Quartus II version 13.0.1. Key resource utilization statistics are as follows:

+ Total Logic Elements: 5,601 out of 33,216 (17%)
+ Combinational Functions: 4,650 (14%)
+ Dedicated Logic Registers: 2,043 (6%)
+ Total Registers: 2,043
+ Total Pins Used: 344 out of 475 (72%)

The FPGA project "single_cycle" demonstrates efficient resource utilization with 5,601 logic elements, accounting for 17% of the available 33,216 elements. The project leverages 4,650 combinational functions (14%) and 2,043 logic registers (6%), indicating a balanced use of logic and memory resources. Additionally, the design utilizes 344 out of 475 available pins, representing 72% of the total pins, highlighting significant interaction with external components. This efficient allocation of resources ensures the project's scalability and robust performance, while maintaining ample room for future enhancements.

Figure 5: Fmax report.

The timing analysis of the RISC-V single-cycle processor design yielded a maximum operating frequency (Fmax) of 37.67 MHz, with the clock signal named i_clk. This result indicates that the design can achieve a stable operation at speeds up to 37.67 MHz on the Cyclone II FPGA in the slow timing model. This frequency provides a baseline for understanding the processor's performance and can guide further optimizations if a higher clock speed is desired.

4.2 Implement on board result

Link Hex to Decimal on LED segments: Click here

Link stopwatch: Click here