Faculty of Engineering
Electrical Engineering Dep.
Instructor: Dr.Gihan Naguib

CEE210/CEE216
Computer Engineering
2nd year computer/Commination
Spring 2025

Pipelined MIPS Processor Implementation

Objectives

- Designing a Pipelined 32-bit RISC processor with 32-bit instructions
- Using the Logisim simulator to model and test the processor
- *Teamwork*

Instruction Set Architecture

In this project, you will design a RISC Processor that has 31 general purpose 32-bit registers: R1 through R31. Register R0 is hardwired to zero Reading R0 always returns the value 0. Writing R0 has no effect. All instructions are 32-bit long and aligned in memory. Memory is ***word (32-bit) addressable***. The program counter PC is a special-purpose 20-bit register. That can address at most 2^{20} instructions. There are three instruction formats, ***R-type, I-type and SB-Type as shown below:***

R-type format

6-bit opcode (Op), 5-bit destination register number d, and two 5-bit source registers numbers S_1 & S_2 and 11-bit function field F

F^{11}	**$S_2{}^5$**	**$S_1{}^5$**	**d^5**	**OP^6**

I-type format

6-bit opcode (Op), 5-bit destination register number d, and 5-bit source registers number S_1 and16-bit immediate (Imm16)

Imm 16	**$S_1{}^5$**	**d^5**	**OP^6**

SB-type format

6-bit opcode (Op), 5-bit register numbers (**S_1**, and **S_2**) and 16-bit immediate split into ({ImmU (11-bit) and ImmL(5-bit)})

$ImmU^{11}$	**$S_2{}^5$**	**$S_1{}^5$**	**$ImmL^5$**	**OP^6**

- S_1 is the first source register number. This register is always read (never written). **RS_1** is the name and value of register S_1
- S_2 is the second source register number. This register is always read (never written). **RS_2** is the name and value of register S_2
- d is the destination register number. This register is always written (never read). **Rd** is the name and value of destination register d

Instruction Encoding

The R-typc, I-type. And SB-type instructions, meanings, and encodings are shown below:

	Instruction	Meaning	Encoding				
R-Type	SLL	Rd = Shift Left Logical (RS_1, RS_2[4:0])	F= 0	S_2	S_1	d	OP=0
	SRL	Rd= Shift Right Logical (RS_1, RS_2 [4:0])	F=1	S_2	S_1	d	OP=0
	SRA	Rd= Shift Right Arith (RS_1, RS_2 [4:0])	F=2	S_2	S_1	d	OP=0
	ROR	Rd = Rotate Right (RS_1, RS_2 [4:0])	F=3	S_2	S_1	d	OP=0
	ADD	Rd = RS_1+ RS_2	F=4	S_2	S_1	d	OP=0
	SUB	Rd = RS_1 – RS_2	F=5	S_2	S_1	d	OP=0
	SLT	Rd = (RS_1 <$_{\text{signed}}$ RS_2)) ? (result 1or 0)	F=6	S_2	S_1	d	OP=0
	SLTU	Rd = (RS_1 < unsigned RS_2) ? (result 1or 0)	F=7	S_2	S_1	d	OP=0
	SEQ	Rd = (RS_1 == RS_2)? (result 1or 0)	F=8	S_2	S_1	d	OP=0
	XOR	Rd = RS_1^ RS_2	F=9	S_2	S_1	d	OP=0
	OR	Rd= RS_1 | RS_2	F=10	S_2	S_1	d	OP=0
	AND	Rd= RS_1& RS_2	F=11	S_2	S_1	d	OP=0
	NOR	Rd = ~ (RS_1 | RS_2)	F=12	S_2	S_1	d	OP=0
	MUL	Rd = (RS_1* RS_2) [31:00]	F=13	S_2	S_1	d	OP=0
I-Type	SLLI	Rd= Shift Left Logical (RS_1, Sa)	0	Sa	S_1	d	Op=1
	SRLI	Rd = Shift Right Logical(RS_1 , Sa)	0	Sa	S_1	d	Op=2
	SRAI	Rd Shift Righi Arith (RS_1, Sa)	0	Sa	S_1	d	Op=3
	RORI	Rd =Rotate Right (RS_1, Sa)	0	Sa	S_1	d	Op=4
	ADDI	Rd = RS_1+ sign extend (imml6)	Imm 16		S_1	d	Op=5
	SLTI	Rd = (RS_1 <$_{\text{Singed}}$ sign extend (imml6)) ?1or 0	Imm 16		S_1	d	Op=6
	SLTIU	Rd = (RS_1 < unsigned zero extend(imml6)) ?1or 0	Imm 16		S_1	d	Op=7
	SEQI	Rd = (RS_1 == sign extend (imml6))? 1or 0	Imm 16		S_1	d	Op=8
	XORI	Rd = RS_1 ^ zero extend(imml6)	Imm 16		S_1	d	Op=9
	ORI	Rd = RS_1 | zero extend(imml6)	Imm 16		S_1	d	Op=10
	ANDI	Rd = RS_1 & zero extend(immI6)	Imm 16		S_1	d	Op=11
	NORI	Rd = ~ (RS_1 | zero extend(imml6))	Imm 16		S_1	d	Op=12
	SET	Rd = sign extend (Imm16)	Imm 16		0	d	Op=13
	SSET	*Rd* ={*(Rd)*[15:0], Imm16}	Imm 16		0	d	Op=14
	JALR	PC = RS_1+sign extend (Imm 16), Rd = PC+l	Imm 16		S_1	d	Op=15
	LW	Rd = Mem[RS_1+sign extend(imm 16)]	Imm 16		S_1	d	Op=16
SB-Type	SW	Mem[RS_1+sign_extend({imm U, immL})] = RS_2	ImmU	S_2	S_1	ImmL	Op=17
	BEQ	If (RS_1= RS_2) PC = PC+sign extend ({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=18
	BNE	if (Rs1 != Rs2) PC=PC+ sign extends ({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=19

	BLT	if (RS_1 < RS_2)
		PC= PC+sign extend ({imm U, immL}
			ImmU	S_2	S_1	ImmL	Op=20

	BGE	If (RS_1 >= RS_2) PC=PC+sign extend({imm U, immL}	ImmU	S_2	S_1	ImmL	Op=21
	BLTU	if (RS_1 < RS_2) unsigned PC = PC + sign extend({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=22

	BGEU	if (RS_1 >= RS_2) unsigned
		PC = PC+ sign extend ({imm U, immL}
			ImmU	S_2	S_1	ImmL	Op=23

Sa: shift amount (bit 0:4)

Instruction Description

Opcodes 0 is used for R-format ALU instructions. There are 14 instructions in total.

Opcodes 1 through 16 are used for I-format instructions. There are 16 instructions in total.

The I-format ALU instructions **(SLLI** through **NORI)** have identical functionality as their corresponding R-format instructions **(SLL** through **NOR),** except that the second ALU operand is an immediate constant. The "**Imm16"** immediate value is sign extended for all instructions except bitwise logical instructions **(XORI, ORI. ANDI,** and **NORI)** where the constant is zero extended.

The SET instruction (opcode 13) sets destination register Rd with a 16-bit signed constant. The immediate constant is sign-extended to 32 bits before writing in register Rd. The **SSET** instruction (opcode 14) reads and writes register Rd. It shifts the value of register a left 16 bits and sets the lower 16 bits: Rd = {Rd[15:0], Imm16}, where **{} means concatenation**. The SET **and SSET instructions can be used together to form any 32-bit constant**. For example, to initialize register **R1 with constant 0x12345678**, do the following:

- **SET R1, 0x1234 (lower 16-bit)**
- **SSET R1, 0x5678 (higher 16-bit)**

Opcode 15 defines the **JALR** (Jump-And-Link-Register) instruction. It saves the return address in **Rd (Rd = PC+1)** and does an indirect register jump with an offset **(PC = RS_1 + sigextend (Imml6)).**

- If **Rd** is **R0** then the return address **(PC+1)** is not saved, and **JALR becomes a Jump Register (JR) pseudo-instruction**.
- If **RS_1** is **R0,** then **JALR** instruction becomes a Jump and link **(JAL)** pseudo-instruction.
- If both **RS_1** and **Rd** are **R0,** then **JALR** instruction becomes a Jump **(J)** pseudo-instruction.
- To use **JALR** instruction, follow the following syntax:
  - **JALR Rd, RS_l, Imml6.**

Opcode 16 define load word (LW) instruction. This instruction addresses 4-byte words in memory. The effective memory address = **RS_1 + sign_extend(Imml6).**

Opcodes 17 through 23 are used for SB -format instructions

Opcode 17 define store word (SW) instruction. This instruction addresses 4-byte words in memory. The effective memory address = **RS_1 + sign_extend({Imm U, Imm L}).**

There are six branch instructions BEQ to BGEU with **opcodes 18 to 23** and PC-relative addressing. If the branch is taken, then **PC = PC + sign_extend({ImmU, ImmL}). Otherwise, PC = PC + 1. The conditional branch range is (-32768, +32677**).

Memory

Your processor will have separate instruction and data memories. The PC register should be 20 bits. The instruction memory can store 2^{20} instructions, where each instruction occupies four bytes. There are 1048576 (IM) instructions in the instruction memory. The PC contains a word address (not a byte address). Therefore, it is sufficient to increment the PC by 1 (rather than 4) to point to the next instruction in memory

The data memory will also be to 2^{20} words. The data memory **is *word addressable***, since only the LW and SW instructions address memory. Words should be always aligned in memory. The ALU result represent the address for the data memory.

Addressing Modes

PC-relative addressing mode is used for branch and jump instructions.

For taken branches:

- PC = PC + sign extend({Imm U, ImmL})

For JALR:

- PC = RS_1 + sign extend(Imml6)

For LW, displacement addressing is usee:

- Memory address = RS_1 + sign extend(imml6)

For SW, displacement addressing is used:

- Memory address = RS_1+sign extend ({ImmU, ImmL})

Register File

Implement a Register file containing 32 (thirty-two) 32-bit registers R0 to R31 with two read ports and one write port. R0 is a special register that can only be read not written (hardwired to zero).

Arithmetic and Logic Unit (ALU)

Implement a 32-bit ALU to perform all the required operations (shown in table above)

.

Program Execution

The program will be loaded and will start at address 0 in the instruction memory. The data segment will be loaded and will start also at address 0 in the data memory. You can also have a stack segment to support procedures. The stack segment can occupy the upper part of the data memory and can grow backwards towards lower memory addresses. The stack segment is implemented completely in software. You can dedicate register R30 as the stack pointer. To terminate the execution of a program, the last instruction in the program can jump to itself indefinitely (because there is no underlying operating system to terminate the program).

.

Phase 1: Build a Single-Cycle Processor

Start by building the datapath and control of a single-cycle processor and ensure its correctness. You should have sufficient test cases that ensure the correct execution of ALL instructions in the instruction set. You should also write test cases that show the correct execution of complete programs. To verify the correctness of your design, show the values of all registers in the register file (R0 to R31) at the top-level of your design. Provide output pins for all registers R0 through R31 and make their values visible outside the register file.

Phase 2 Build a Pipelined Processor

Once you have succeeded in building a single-cycle processor and verified its correctness, design and implement a pipelined version of your design. Make a copy of your single-cycle design, then convert it and implement a pipelined datapath and its control logic. Add pipeline registers between stages. Design the control logic to detect data dependencies among instructions and implement the forwarding logic. You should handle properly the control hazards of the branch and jump instructions. Also, stall the pipeline after a LW instruction, if it is followed by a dependent instruction.

Bonus:

By integrating a 2-bit dynamic branch predictor into your simulator, you can effectively minimize the frequency of pipeline flushes required for branch and jump instructions. This enhancement leads to a significant reduction in the delay, ultimately achieving a zero-cycle delay for such instructions.

Testing and verification

To demonstrate that your CPU is working, you should do the following:

- ❖ Test all components and sub-circuits independently to ensure their correctness. For example, test the correctness of the ALU, the register file, the control logic separately, before putting your components together.
- ❖ Test each instruction independently to ensure its correct execution.
- ❖ Write a sequence of instructions to verify the correctness of ALL instructions. Use SET and SSET to initialize registers or load their values from memory. Demonstrate the correctness of all ALU R-type and I-type instructions. Demonstrate the correctness of LW and SW instructions. Similarly, you should demonstrate the correctness of all branch and jump instructions
- ❖ Test sequences of dependent instructions to ensure the correctness of the forwarding logic. Also, test a LW (load word) followed by a dependent instruction to ensure stalling the pipeline correctly by one clock cycle.
- ❖ Test the behavior of taken and untaken branch instructions and their effect on stalling the pipeline.
- ❖ Make several copies and versions of your design before making changes, in case you need to go back to an older version.

Test Programs:

1. Write a **test program involving procedures and arrays**. For example, the main procedure initializes array elements with some constant values. It then calls a second procedure after passing the array address and the number of elements as parameters in two registers. The second procedure uses the parameters to compute the sum of the array elements and returns the result in a register.

2. Translate each program into machine instructions and load it into the instruction memory starting at address 0. You can also save the image of the instruction and data memories into files and reload them later for testing purposes.

Project Report

The report document must contain sections highlighting the following:

1. *Design and Implementation*

- Specify clearly the design giving detailed description of the datapath, its components, control, and the implementation details.
- Provide drawings of the component circuits and the overall datapath.
- Provide a description of the control logic and the control signals. **Provide a table giving the control signal values for each instruction.**
- Provide a description of the forwarding logic, the cases that were handled, and the cases that stall the pipeline, and the logic that you have implemented to stall the pipeline

2. *Simulation and Testing*

- Carry out the simulation of the processor developed using Logisim.
- List all the instructions that were tested and work correctly. ***List all the instructions that do not run properly.***
- Document all your test programs and files and include them in the report document
- Describe all the cases that you handled involving **dependences between instructions, forwarding cases, and cases that stall the pipeline**
- Also provide snapshots of the Simulator window with your test program loaded and showing the simulation output results.

3. *Teamwork*

- Two or at most three students can form a group. Write the names of all the group members on the project report title page.
- Group members are required to coordinate their work among themselves, so that everyone is involved in design, implementation, simulation, and testing.
- Show the work done by each group member using a chart.

Submission Guidelines

- ❖ The single-cycle processor design should be completed at **28-4-2025**.
    - ➢ It should be fully operational. You should have sufficient test cases ready to prove that your CPU is fully functional.
- ❖ The pipelined processor design should be completed at **9-5-2025** .
    - ➢ It should be fully operational. You should have sufficient test cases ready to prove that your pipelined CPU is fully functional.
- ❖ **If your CPU is not fully operational then identify which instructions do not work properly, or which hazards are not handled properly to avoid the loss of many marks.**
- ❖ All submission will be done through both emails:
    - ➢ **gaa11@fayoum.edu.eg**
    - ➢ **gna00@Fayoum.edu.eg**
- ❖ The project should be submitted **on the due date by midnight.**
    - ❖ Attach one zip file containing:
        - ▪ All the design circuits and sub-circuits,
        - ▪ The test programs, their source code and binary instruction files that you have used to test your design, their test data, as well as the report document.
        - ▪ Project report
        - ▪ ***Video demonstrating your work in details.***

Grading policy

The grade will be divided according to the following components:

- ➢ **Correctness: whether your implementation is working**
- ➢ **Completeness and testing: whether all instructions and cases have been implemented, handled, and tested properly**
- ➢ **Report document**
- ➢ **Project presentation and discussion**