COE 301 – Computer Organization

Term 251 – Fall 2025

Project: Pipelined Processor Design

Objectives:

- Designing a Pipelined 32-bit processor with 16-bit instructions
- Using the Logisim Evolution Simulator to model and test the processor
- Teamwork

Instruction Set Architecture

In this project, you will design a RISC processor that has seven (7) 32-bit registers: R1 to R7. Register R0 is hardwired to zero. Hence, reading R0 always returns the value 0, and writing R0 has no effect, and the value written to R0 is discarded.

All instructions are 16-bit long and aligned in memory. The PC register contains the instruction address. All instruction addresses are even, and therefore the lower bit of the PC register is hardwired to 0. There are three instruction formats, R-format, I-format, and J-format as shown below:

R-format

5-bit opcode (Op), 3-bit register numbers (*a*, *b*, and *d*), and 2-bit function field *f*

Op^5	a^3	b^3	d^3	f^2

I-format

5-bit opcode (Op), 3-bit register numbers (*a* and *b*), and 5-bit Immediate

Op^5	a^3	b^3	Imm5

J-format

5-bit opcode (Op) and11-bit Immediate

Op^5	Imm11

Register Use

Register *a* is the first source register number. *Ra* is the name and value of register *a*. It is always read and never written.

Register *b* is the second source register number. It is always read in the R-format but can be read and written in the I-format. *Rb* is the name and value of register *b*.

Register *d* is the destination register number for the R-format only. It is always written and never read. *Rd* is the name and value of destination register *d*.

Instruction Encoding

The R-type, I-type, and J-type instructions, meanings, and encodings are shown below:

Instruction	Meaning	Encoding				
AND	Rd = Ra & Rb	Op = 0	a	b	d	f = 0
CAND	Rd = ~Ra & Rb	Op = 0	a	b	d	f = 1
OR	Rd = Ra | Rb	Op = 0	a	b	d	f = 2
XOR	Rd = Ra ^ Rb	Op = 0	a	b	d	f = 3
ADD	Rd = Ra + Rb	Op = 1	a	b	d	f = 0
NADD	Rd = –Ra + Rb	Op = 1	a	b	d	f = 1
SEQ	Rd = (Ra == Rb) (result is 0 or 1)	Op = 1	a	b	d	f = 2
SLT	Rd = (Ra < Rb) (result is 0 or 1)	Op = 1	a	b	d	f = 3
ANDI	Rb = Ra & Imm	Op = 4	a	b	Imm5	
CANDI	Rb = ~Ra & Imm	Op = 5	a	b	Imm5	
ORI	Rb = Ra | Imm	Op = 6	a	b	Imm5	
XORI	Rb = Ra ^ Imm	Op = 7	a	b	Imm5	
ADDI	Rb = Ra + Imm	Op = 8	a	b	Imm5	
NADDI	Rd = –Ra + Imm	Op = 9	a	b	Imm5	
SEQI	Rb = (Ra == Imm) (result is 0 or 1)	Op = 10	a	b	Imm5	
SLTI	Rb = (Ra < Imm) (result is 0 or 1)	Op = 11	a	b	Imm5	
SLL	Rb = Shift_Left_Logical(Ra, Imm5)	Op = 12	a	b	Imm5	
SRL	Rb = Shift_Right_Logical(Ra, Imm5)	Op = 13	a	b	Imm5	
SRA	Rb = Shift_Right_Arithmetic(Ra, Imm5)	Op = 14	a	b	Imm5	
ROR	Rb = Rotate_Right(Ra, Imm5)	Op = 15	a	b	Imm5	
BEQ	Branch if (Ra == Rb) (PC += Imm<<1)	Op = 16	a	b	Imm5	
BNE	Branch if (Ra != Rb) (PC += Imm<<1)	Op = 17	a	b	Imm5	
BLT	Branch if (Ra < Rb) (PC += Imm<<1)	Op = 18	a	b	Imm5	
BGE	Branch if (Ra >= Rb) (PC += Imm<<1)	Op = 19	a	b	Imm5	
LW	Rb ←4byte MEM[Ra + Imm]	Op = 20	a	b	Imm5	
SW	MEM[Ra + Imm] ←4byte Rb	Op = 21	a	b	Imm5	
JALR	Rb = PC+2; PC = Ra	Op = 27	a	b	0	
J	PC = PC + signed(Imm11<<1)	Op = 28	Imm11			
JAL	R7 = PC + 2; PC=PC+signed(Imm11<<1)	Op = 29	Imm11			
IMM	11-bit Immediate extension	Op = 30	Imm11			

Instruction Description

Opcodes 0 and 1 are used for R-format ALU instructions. There are 8 instructions in total.

Opcodes 4 through 15 are used for I-format ALU instructions. Register *b* is the destination register. The immediate constant replaces the second ALU operand.

The I-format ALU instructions **ANDI** through **SLTI** have identical functionality as their corresponding R-format instructions (**AND** through **SLT**), except that the second ALU operand is an immediate constant and the destination register is **Rb** (not **Rd**).

The I-format encodes a 5-bit signed immediate **Imm5** with range -16 to +15. However, sometimes a longer 16-bit constant might be required. Instead of loading a 16-bit constant from memory, it is better to encode it in the instruction sequence. To achieve this, the **IMM** opcode is defined. The **IMM** opcode (30) carries an 11-bit immediate **Imm11**. There are two cases:

Case 1: If an I-format instruction is not followed by an **IMM** extension, **Imm5** is **sign-extended** to **32 bits**. The instruction fetch unit must prefetch the next instruction to ensure that it is different from **IMM**.

Imm = sign_extend(Imm5).

Case 2: If an I-format instruction is followed by an **IMM** extension, then **Imm11** is concatenated with **Imm5** to form a **16-bit immediate**, which is **sign-extended to 32 bits**.

Imm = sign_extend(Imm11:Imm5), where **:** means the concatenation of bits.

For example, **ADDI R2 = R1, 0x5678** is encoded with an **IMM** extension as follows:

ADDI R2 = R1, 0x5678	**Op = ADDI**	**a = 2**	**b = 1**	**Imm5 = 0x18**
	Op = IMM	**Imm11 = 0x2B3**		

The **IMM** extension enables the use of 16-bit constants in any I-format instruction. It simplifies programming. You should translate the instruction properly depending on the value of the immediate constant. If the constant is in the range -16 to +15 then there is no need for the **IMM** extension (case 1). If the constant is longer than 5 bits but does not exceed 16 bits then there the **IMM** extension should be used (case 2). If the constant is longer than 16 bits then it should be loaded from memory using the **LW** instruction, or encoded as a sequence of I-type instructions.

At most one **IMM** extension can appear after an I-format instruction. The decode stage should generate the immediate according to the opcode sequence. If more than one **IMM** extension is used after an instruction, then the decode stage should ignore and skip the additional extensions and handle them as if they were **NOP** instructions.

There are four shift and rotate instructions: **SLL** to **ROR** with opcodes 12 to 15. The shift or rotate amount is the 5-bit immediate **Imm5** with values 0 to 31. Shift and rotate instructions do not use the **IMM** extension.

There are four branch instructions **BEQ** to **BGE** with opcodes 16 to 19 and PC-relative addressing. If the branch is taken then **PC = PC + Imm<<1**. Otherwise, **PC = PC + 2**. As with ALU I-format instructions, Branch instructions can use the **IMM** extension to increase the branch target address range.

Opcodes 20 and 21 define the load word (**LW**) and store word (**SW**) instructions. These two instructions address 4-byte words in memory. Displacement addressing is used. The effective memory address = **Ra + Imm**. The **IMM** extension can be used to increase the immediate range. Register *b* is a destination register for **LW**, but a source for **SW**. Loading and storing a byte or a half word is not defined to simplify the project.

Opcode 27 defines the **JALR** (Jump-And-Link-Register) instruction. It saves the return address in **Rb** (**Rb = PC+2**) and does an indirect register jump (**PC = Ra**). If **Rb** is **R0** then the return address (**PC+2**) is not saved and **JALR** becomes a Jump Register (**JR**) pseudo-instruction.

Opcodes 28 and 29 define the Jump (**J**) and Jump-and-Link (**JAL**) instructions. PC-relative addressing is used to compute the jump target address: **PC = PC + sign_extend(Imm11<<1)**. In addition, the **JAL** instruction saves the return address in **R7** (**R7 = PC + 2**).

The **IMM** opcode 30 is not an executable instruction. It is used as an extension to I-format ALU, branch, load, and store instructions to extend the immediate from 5 bits to 16 bits. However, shift and rotate instructions do not use the **IMM** extension because they use a 5-bit shift amount only.

Programming Notes:

NADD	**R1 = R2, R3**	computes R1 = R3 – R2	(Rd = R1, Ra = R2, and Rb = R3)
SUB	**R1 = R2, R3**	is a pseudo-instruction ➔	**NADD R1 = R3, R2**
ADDI	**R1 = R2, –1**	computes R1 = R2 – 1	(Rb = R1, Ra = R2, and Imm5 = –1)
NADDI	**R1 = R2, 1**	computes R1 = 1 – R2	(Rb = R1, Ra = R2, and Imm5 = 1)
LI	**R1 = 5**	is a pseudo-instruction ➔	**ORI R1 = R0, 5**

ADDI **R1 = R2, 0x5678** is a sequence of **ADDI** and **IMM** opcodes (4 bytes in total)
(**ADDI Imm5 = 0x18** and **IMM Imm11 = 0x2B3**)

PC Register and Instruction Memory

The program counter (PC register) is a special-purpose 20-bit register. It can address an **instruction memory** having 2^{20} bytes = 2^{19} instructions. The least-significant bit of the PC register is hardwired to 0. All instruction addresses are multiple of 2 bytes.

Data Memory

The LW and SW instructions address a separate **data memory**. The data memory will be also restricted to 2^{20} bytes = 2^{18} words. Words should be always aligned in memory. The least-significant two bits of the word address must be zeros (ignored in the hardware implementation).

Although the architecture is 32 bits, the size of the instruction and data memories will be restricted to 2^{20} bytes. This is because the *Logisim* tool supports only small memories.

Addressing Modes

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

For taken branches: **PC = PC + sign_extend(Imm<<1)**

For jumps (**J** and **JAL**): **PC = PC + sign_extend(Imm11<<1)**, where **Imm11** is only 11 bits.

For **JALR**: **PC = Ra**

For **LW** and **SW**, displacement addressing is used: **Address = Ra + sign_extend(Imm)**

Register File

Implement a Register file containing Seven 32-bit registers R1 to R7 with two read ports and one write port. R0 is hardwired to 0. R0 is always read as 0. R0 cannot be written.

Arithmetic and Logic Unit (ALU)

Implement a 32-bit ALU to perform all the required operations:

AND, CAND, OR, XOR, ADD, NADD, SEQ, SLT, SLL, SRL, SRA, and ROR.

Instruction Decode Stage

Implement a special instruction decode stage that detects a sequence of an I-format instruction followed by an **IMM** extension. The instruction decode stage should concatenate and sign-extend the immediate to produce the 32-bit immediate. If there is more than one **IMM** extension then the additional ones should be ignored (become NOPs). If an **IMM** extension appears after an R-format or J-format instruction then it should also be ignored.

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 **R6** 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.

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 (R1 to R7) at the top-level of your design. Provide output pins for registers R1 through R7 and make their values visible at the top level of your design to simplify testing and verification.

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 data hazards when there are data dependencies between instructions and stall the pipeline when needed. You should handle properly the control hazards of the branch and jump instructions.

Testing and Verification

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

1. Write a sequence of instructions to verify the correctness of ALL instructions. Demonstrate the correctness of all ALU R-format and I-format instructions. Demonstrate the correctness of LW and SW instructions. Similarly, demonstrate the correctness of all branch and jump instructions.
2. Write short programs and loops to verify the correctness of a sequence of instructions. Make sure that your pipelined CPU can handle data hazards and control hazards properly.
3. Write a sort procedure of your choice (selection sort, bubble sort, etc.). Write a main function to call the sort procedure and sort an array of integers in the data memory.

Document all your test programs and files and include them in the report document.

Project Report

The report document must contain sections highlighting the following:

1 – Design and Implementation

- Provide drawings of the various components and the overall datapath.
- Provide a complete description of the control logic and the control signals. Provide a table giving the control signal values for each instruction.
- Provide a complete description of the forwarding logic, the cases that were handled, and the logic you have implemented to handle the control hazards.

2 – Simulation and Testing

- Describe the test programs that you used to test your design with sufficient comments describing the programs, their input, and expected output. List all the instructions that were tested and work correctly. List all the instructions that do not run properly.
- Describe all the cases that you handled involving data dependences between instructions, data forwarding, and stalling the pipeline.
- Provide snapshots showing test programs and their 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.

PROJECT DEADLINES

The single-cycle processor design should be completed during week 12 of the semester. It should be fully operational and will be evaluated by your lab instructor in the LAB starting week 12 of the semester. You should have sufficient test cases ready to prove that your CPU is fully functional.

The pipelined processor design should be completed during week 14 of the semester. It should be fully operational and demonstrated in the LAB during week 15 of the semester. You should have sufficient test cases ready to prove that your pipelined CPU is fully functional.

Submit a hard copy of the project report document to your LAB instructor during week 15 of the semester.

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.

Submit a zip file containing the logisim design circuits, the test programs, and the project report document on Blackboard during week 15 of the semester.