COE 301 – Computer Architecture & Assembly Language

Term 242 – Spring 2025

Project: 16-bit Pipelined Processor Implementation

Objectives:

- Using the Logisim-evolution simulator
- Designing and testing a Pipelined 16-bit processor
- Teamwork

Instruction Set Architecture

In this project, you will design a simple 16-bit RISC processor with seven 16-bit general-purpose registers: R1 through R7. R0 is hardwired to zero and cannot be written, so we are left with seven registers. There is also one special-purpose 16-bit register, which is the program counter (PC). There are three instruction formats, R-type, I-type, and J-type and all instructions are 16 bits, with the different fields as shown below.

R-type format

5-bit opcode (Op), 2-bit function field (f), and 3-bit register numbers (Rd, Rs, and Rt)

Op^5	f^2	Rd^3	Rs^3	Rt^3

I-type format

5-bit opcode (Op), 5-bit immediate constant and 3-bit register number (Rs and Rt)

Op^5	Immediate^5	Rs^3	Rt^3

J-type format

5-bit opcode (Op) and 11-bit immediate constant

Op^5	Immediate^{11}

For **R-type** instructions, Rs and Rt specify the two source register numbers, and Rd specifies the destination register number. The function field can specify at most four (4) functions for a given opcode. Several opcodes can be reserved for the R-type instructions.

For **I-type** instructions, Rs specifies a source register number, and Rt can be a second source or a destination register number. The immediate constant is only 5 bits because of the fixed-size nature of the instruction. The size of the immediate constant is suitable for the purpose of the design uses. The 5-bit immediate constant can be signed or unsigned, depending on the opcode.

For **J-type** instructions, an 11-bit immediate constant is used for LUI (load upper immediate), J (jump), and JAL (jump-and-link) instructions.

Instruction Encoding

Nine R-type instructions, twelve I-type instructions, and three J-type instructions are defined. These instructions, their meaning, and their encoding are shown below:

Instr.	**Meaning**	**Encoding**				
ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 00000	f = 00	Rd	Rs	Rt
SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 00000	f = 01	Rd	Rs	Rt
SLT	Reg(Rd) = Reg(Rs) signed < Reg(Rt)	Op = 00000	f = 10	Rd	Rs	Rt
SLTU	Reg(Rd) = Reg(Rs) unsigned < Reg(Rt)	Op = 00000	f = 11	Rd	Rs	Rt
AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 00001	f = 00	Rd	Rs	Rt
OR	Reg(Rd) = Reg(Rs) | Reg(Rt)	Op = 00001	f = 01	Rd	Rs	Rt
XOR	Reg(Rd) = Reg(Rs) ^ Reg(Rt)	Op = 00001	f = 10	Rd	Rs	Rt
NOR	Reg(Rd) = ~(Reg(Rs) | Reg(Rt))	Op = 00001	f = 11	Rd	Rs	Rt
SLL	Reg(Rd) = Reg(Rs) << Reg(Rt)	Op = 00010	f = 00	Rd	Rs	Rt
SRL	Reg(Rd) = Reg(Rs) zero>> Reg(Rt)	Op = 00010	f = 01	Rd	Rs	Rt
SRA	Reg(Rd) = Reg(Rs) sign>> Reg(Rt)	Op = 00010	f = 10	Rd	Rs	Rt
ROR	Reg(Rd) = Reg(Rs) rot>> Reg(Rt)	Op = 00010	f = 11	Rd	Rs	Rt
REVL	Reg(Rd[b7, …, b0]) = Reg(Rs[b0, …, b7])	Op = 00011	f = 00	Rd	Rs	000
REVH	Reg(Rd[b7, …, b0]) = Reg(Rs[15, …, b8])	Op = 00011	f = 01	Rd	Rs	000
REVA	Reg(Rd[b15, …, b0])= Reg(Rs[b0, …, b15])	Op = 00011	f = 10	Rd	Rs	000
JR	PC = Reg(Rs)	Op = 00011	f = 11	000	Rs	000
ADDI	Reg(Rt) = Reg(Rs) + Immediate^{5}	Op = 01000	Immediate^{5}		Rs	Rt
ANDI	Reg(Rt) = Reg(Rs) & Immediate^{5}	Op = 01001	Immediate^{5}		Rs	Rt
ORI	Reg(Rt) = Reg(Rs) | Immediate^{5}	Op = 01010	Immediate^{5}		Rs	Rt
XORI	Reg(Rt) = Reg(Rs) ^ Immediate^{5}	Op = 01011	Immediate^{5}		Rs	Rt
LW	Reg(Rt) = Mem(Reg(Rs) + Imm^{5})	Op = 01100	Immediate^{5}		Rs	Rt
SW	Mem(Reg(Rs) + Imm^{5}) = Reg(Rt)	Op = 01101	Immediate^{5}		Rs	Rt
BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 01110	Immediate^{5}		Rs	Rt
BNE	Branch if (Reg(Rs) != Reg(Rt))	Op = 01111	Immediate^{5}		Rs	Rt
J	**PC = (Zero-Extend)Immediate^{11}**	Op = 10000	Immediate^{11}			
JAL	R7 = PC + 1, **PC = (Zero-Extend)Immediate^{11}**	Op = 10001	Immediate^{11}			
LUI	R1 = Immediate^{11} << 5	Op = 11111	Immediate^{11}			

Opcodes 0, 1, 2, and 3 are used for R-type instructions. There are three shift and one rotate instruction. For the shift and rotate instructions, the lower 4 bits of register Rt are used as the shift/rotate amount. Opcode 3 is used for the JR (jump register) instruction and the bit-reversal instructions. Opcodes 8 through 15 are used for I-type instructions. The 5-bit immediate constant is zero-extended for ANDI, ORI, and XORI. It is sign-extended for the remaining instructions. The J-type instructions have an 11-bit immediate constant. The Load Upper Immediate (LUI) is of the J-type to have an 11-bit immediate constant loaded into the upper 11 bits of register R1. The LUI can be combined with ORI to load any 16- bit constant into a register. Although the instruction set is reduced, it is still rich enough to write useful programs. Procedure calls and returns are implemented using the JAL and JR instructions.

Note about the Reverse Instructions:
The instructions REVL, REVH and REVA are used to reverse the contents of the source register (Rs) and save the result in the destination register (Rd). Details of those instructions are given in the table below.

Instruction	Syntax	Effect	Result
REVA	REVA Rd, Rs	Reverses the contents of Rs and saves the result in Rd.	Reg(Rd[b15, …, b0]) 🡐 Reg(Rs[b0, …, b15])
REVL	REVL Rd, Rs	Reverses the contents of the lower byte of Rs and saves the result in the lower byte of Rd.	Reg(Rd[b7, …, b0]) 🡐 Reg(Rs[b0, …, b7])
REVH	REVH Rd, Rs	Reverses the contents of the higher byte of Rs and saves the result in the lower byte of Rd.	Reg(Rd[b7, …, b0]) 🡐 Reg(Rs[b8, …, b15])

Memory

The processor will have separate instruction and data memories with 2^{16} words each. Each word is 16 bits or 2 bytes and the Memory is *word addressable*. Only words (not bytes) can be read and written to memory, and each address is a word address. The purpose of this addressing is to simplify the implementation. The PC contains a word address (not a byte address); therefore, it is sufficient to increment the PC by 1 (rather than 2) to point to the next instruction in memory. Also, the Load and Store instructions can only load and store words. There is no instruction to load or store a byte in memory.

Register File

Implement a Register file containing Seven 16-bit registers R1 to R7 with two read ports and one write port. R0 is hardwired to zero.

Arithmetic and Logical Unit (ALU)

Implement a 16-bit ALU to perform all the required operations:
ADD, SUB, SLT, SLTU, OR, AND, XOR, NOR, SLL, SRL, SRA, ROR, REVA, REVL and REVH.

Addressing Modes

PC-relative addressing mode is used in the CPU for branch and jump instructions.
For branching (BEQ, BNE), the branch target address is computed as follows:
PC = PC + sign-extend(Imm5). Add the contents of PC to the sign-extended 5-bit Immediate. For jumps (J and JAL): PC = PC + sign-extend(Imm11).
For LW and SW base-displacement addressing mode is used. The base address is obtained from register (Rs) and added to the sign-extended 5-bit immediate to compute the memory address.

Program Execution

The program is loaded at address 0x0000 in the instruction memory, and starts execution at the same address. The data segment starts at address 0x0000 in the data memory. A stack segment may also be added to support procedures. The stack segment can occupy the upper part of the data memory and can grow backwards towards lower addresses. The stack segment is completely implemented in software. To **terminate** the execution of a program, the last instruction in the program can jump or branch to itself indefinitely.

Building a Pipelined Processor

It is recommended that you start by building the datapath and control of a single-cycle processor and ensure its correctness. Once you have succeeded in doing it, test it to verify its proper functioning. Then, convert the design and implement a pipelined-datapath and related control logic. A five-stage pipeline should be constructed similar to the pipeline given in class lectures. Add the necessary pipeline registers between the different stages. Design the control logic to detect data dependencies among instructions and implement the forwarding logic. For branch and jump instructions, reduce the delay to one cycle only. Stall the pipeline for one clock cycle after a jump or a taken branch instruction. If the branch is not taken, then there is no need to stall the pipeline. Also, stall the pipeline after a LW instruction, if it is followed by a dependent instruction.

WARNING

Although Logisim is stable, it might crash from time to time. Therefore, it is best to save your work often. Make backup copies and versions of your design before making changes, in case you need to go back to an older version.

Testing

- 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.
- 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.
- Write a sample program that adds an array of integers. Two procedures are required. The main procedure initializes the array elements with some constant values. It then calls the 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. Convert the program into machine instructions by hand and load it into the instruction memory starting at address 0x0000. Having two procedures, you will be able to test the JAL and JR instructions.
- Write additional programs as necessary for further testing, translate them by hand, and save them into files. These files can be loaded into the instruction memory and executed. Their data can be saved as well in files and loaded into 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

- Specify clearly the design giving detailed description of the datapath, its components, control, and the implementation details (highlighting the design choices you made and why, and any notable features that your processor might have.)
- Provide drawings of the component circuits 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 the logic equations for each control signal.
- Provide a complete 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.
- Provide list of sources for any parts of your design that are not entirely yours (if any).
- Carry out the design and implementation with the following aspects in mind:

- Correctness of the individual components

- Correctness of the overall design when wiring the components together

- Completeness: all instructions were implemented properly, detecting dependences and forwarding was handled properly, and stalling the pipeline was handled properly for all cases.

2 – Simulation and Testing

- Carry out the simulation of the processor developed using Logisim.
- Describe the test programs that you used to test your design with enough comments describing the program, its inputs, and its 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 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. Make sure to write the names of all the group members on the project report title page.
- Group members are required to coordinate the work equally among themselves so that everyone is involved in all the following activities:

- Design and Implementation

- Simulation and Testing

- Clearly show the work done by each group member using a chart and prepare an execution plan showing the time frame for completing the subtasks of the project. You can also mention how many meetings were conducted between the group members to discuss the design, implementation, and testing.

Submission Guidelines

All submissions will be done through WebCT.

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. Submit also a hard copy of the report during the class lecture.

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
- Participation and contribution to the project
- Report document

Late policy

The project should be submitted on the due date by midnight. Late projects are accepted for a maximum of 3 late days. Projects submitted after 3 late days will not be accepted. The maximum late penalty is 10%.