COE 301 – Computer Organization

Term 222 – Spring 2023

Project: Pipelined Processor Design

Objectives:

- Designing a Pipelined 16-bit RISC processor with 16-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 7 general-purpose 16-bit registers: R1 to R7. Register R0 is hardwired to zero. Reading R0 always returns the value 0. Writing R0 has no effect. The value written to R0 is discarded.

All instructions are 16-bit long and aligned in memory. Memory is word (16-bit) addressable. The PC register (16-bit wide) contains the instruction address. There are three instruction formats:

R-type, I-type, and J-type as shown below:

R-type

5-bit opcode (Op), 3-bit register numbers (*rs*, *rt*, and *rd*), and 2-bit function field (*f*)

f^2	rd^3	rt^3	rs^3	Op^5
15 14	13 11	10 8	7 5	4 0

I-type

5-bit opcode (Op), 3-bit register numbers (*rs* and *rt*), and 5-bit Immediate (*imm5*)

imm^5	rt^3	rs^3	Op^5
15 11	10 8	7 5	4 0

J-type

5-bit opcode (Op) and 11-bit Immediate (imm11)

imm^{11}	Op^5
15	4 0

rs is the first source register number. This register is always read (never written). Ra is the name and value of register rs1.
rt is the second source register number. This register is read for R-type instructions and written for some I-type instructions. Rb is the name and value of register rt.
rd is the destination register number. This register is always written (never read). Rd is the name and value of destination register rd.

Instruction Encoding

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

Instruction	Meaning	Encoding				
ADD	Rd=Ra + Rb	f=0	rd	rt	rs	Op=0
SUB	Rd=Ra – Rb	f=1	rd	rt	rs	Op=0
SLT	Rd=(Ra < Rb) signed	f=2	rd	rt	rs	Op=0
SLTU	Rd=(Ra < Rb) unsigned	f=3	rd	rt	rs	Op=0
XOR	Rd=Ra ^ Rb	f=0	rd	rt	rs	Op=1
OR	Rd=Ra | Rb	f=1	rd	rt	rs	Op=1
AND	Rd=Ra & Rb	f=2	rd	rt	rs	Op=1
NOR	Rd = ~(Ra | Rb)	f=3	rd	rt	rs	Op=1
SLL	Rd=ShiftLeftLogical(Ra, Rb[3:0])	f=0	rd	rt	rs	Op=2
SRL	Rd=ShiftRightLogical(Ra, Rb[3:0])	f=1	rd	rt	rs	Op=2
SRA	Rd=ShiftRightArith(Ra, Rb[3:0])	f=2	rd	rt	rs	Op=2
ROR	Rd=RotateRight(Ra, Rb[3:0])	f=3	rd	rt	rs	Op=2
ADDI	Rb=Ra + sign_extend(imm5)	imm5		rt	rs	Op=4
SLTI	Rb=(Ra < sign_extend(imm5)) signed	imm5		rt	rs	Op=6
SLTIU	Rb=(Ra < sign_extend(imm5)) unsigned	imm5		rt	rs	Op=7
XORI	Rb=Ra ^ zero_extend(imm5)	imm5		rt	rs	Op=8
ORI	Rb=Ra | zero_extend(imm5)	imm5		rt	rs	Op=9
ANDI	Rb=Ra & zero_extend(imm5)	imm5		rt	rs	Op=10
NORI	Rb = ~(Ra | zero_extend(imm5))	imm5		rt	rs	Op=11
SLLI	Rb=ShiftLeftLogical(Ra, sa)	0	sa	rt	rs	Op=12
SRLI	Rb=ShiftRightLogical(Ra, sa)	0	sa	rt	rs	Op=13
SRAI	Rb=ShiftRightArith(Ra, sa)	0	sa	rt	rs	Op=14
RORI	Rb=RotateRight(Ra, sa)	0	sa	rt	rs	Op=15
LW	Rb=Mem[Ra+sign_extend(imm5)]	imm5		rt	rs	Op=16
SW	Mem[Ra+sign_extend(imm5)] = Rb	imm5		rt	rs	Op=17
BEQ	if (Ra == Rb) PC = PC + sign_extend(imm5)	imm5		rt	rs	Op=18
BNE	if (Ra != Rb) PC = PC + sign_extend(imm5)	imm5		rt	rs	Op=19
BLT	if (Ra < Rb) PC = PC + sign_extend(imm5)	imm5		rt	rs	Op=20
BGE	if (Ra >= Rb) PC = PC + sign_extend(imm5)	imm5		rt	rs	Op=21
JALR	PC=Ra+sign_extend(imm5), Rb=PC+1	imm5		rt	rs	Op=22
LUI	R1=imm11 << 5	imm11				Op=23
J	PC=PC+sign_extend(imm11)	imm11				Op=24
JAL	PC=PC+sign_extend(imm11), R7=PC+1	imm11				Op=25

Instruction Description

Opcodes 0 through 2 are used for R-format ALU instructions. There are 12 instructions in total.

Opcodes 4 through 22 are used for I-format instructions. There are 18 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 **imm5** immediate value is sign extended for all instructions except bitwise logical instructions (**XORI**, **ORI**, **ANDI**, and **NORI**) where the constant is zero extended.

Opcode 16 define load word (**LW**) instruction. This instruction addresses 2-byte words in memory. The effective memory address = **Ra + sign_extend(imm5)**.

Opcode 17 define store word (**SW**) instruction. This instruction addresses 2-byte words in memory. The effective memory address = **Ra + sign_extend(imm5)**.

There are four branch instructions **BEQ** to **BGE** with opcodes 18 to 21 and PC-relative addressing. If the branch is taken, then **PC = PC + sign_extend(imm5)**. Otherwise, **PC = PC + 1**. The conditional branch range is **[-32, +31]**.

Opcode 22 defines the **JALR** (Jump-And-Link-Register) instruction. It saves the return address in **Rb** (**Rb=PC+1**) and does an indirect register jump with an offset (**PC = Ra + sign_extend(imm5)**). If **Rb** is **R0** then the return address (**PC+1**) is not saved, and **JALR** becomes a Jump Register (**JR**) pseudo-instruction. To use **JALR** instruction, follow the following syntax: **JALR rt, rs, imm5**.

Opcode 23 define load upper immediate (**LUI**) instruction. **LUI** is used to build 16-bit constants and uses the J-type format. **LUI** places the **imm11** value in the upper 11 bits of the destination register ***R1***, filling the lowest 5 bits with zeros.

Opcode 24 define the Jump (**J**) instruction. PC-relative addressing is used to compute the jump target address: **PC = PC + imm11**.

Opcode 25 define Jump-and-Link (**JAL**) instruction. PC-relative addressing is used to compute the jump target address: **PC = PC + imm11**. In addition, the **JAL** instruction saves the return address in **R7** (**R7 = PC + 1**).

Memory

Your processor will have separate instruction and data memories. The PC register should be 16 bits. The instruction memory can store 2^{16} instructions, where each instruction occupies two bytes. There are 65536 (64K) instructions in the instruction memory.

The data memory will also be to 2^{16} words = 128 Ki Bytes. The data memory is *word (16-bits) 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(imm5)

For J and JAL: PC = PC + sign_extend(imm11)

For JALR: PC = Ra + sign_extend(imm5)

For LW, displacement addressing is used: Memory address = Ra + sign_extend(imm5)

For SW, displacement addressing is used: Memory address = Ra+sign_extend (imm5)

Register File

Implement a Register file containing 8 (eight) 16-bit registers R0 to R7 with two read ports and one write port. R0 is a special register that can only be read not written (hardwired to zero).

Register rs is always read by all instructions (never written).

Register rt is read by R-type and written by I-type instructions.

Register rd is written by R-type instructions.

Arithmetic and Logic Unit (ALU)

Implement a 16-bit ALU to perform all the required operations:

ADD, SUB, SLT, SLTU, XOR, OR, AND, NOR, SLL, SRL, SRA, ROR

In addition, you should have special support for the LUI instruction.

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.

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 R6) at the top-level of your design. Provide output pins for all registers R0 through R6 and make their values visible outside the register file.

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.

Design Alternatives

When designing the datapath and control unit, explore alternative design options and justify why a given design alternative is chosen. For example, when designing the control unit consider implementing it using a decoder and a set of OR/NOR gates, versus using a ROM to store the control signals, versus optimizing the equation of each control signal separately. When designing the ALU and the shifter unit, consider alternative designs and justify why a design alternative is chosen. The same should be applied for all design decisions in your CPU, such as handling control and data hazards in the pipeline.

Testing

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. Use LUI and ORI to initialize registers. 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.
2. Write a simple program that counts the number of 1's in a 16-bit register.
3. 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.
4. Write a sort procedure (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

- Highlight the design choices you made and why, and any notable features that your processor has.
- 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.

Submission Guidelines

All submissions will be done through Blackboard.

Attach one zip file containing all the design circuits, the test programs source code and binary instruction files that you have used to test your design, their test data, as well as the report document.

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. It should be noted that being in a group that implemented the project correctly does not qualify you to get full mark. Your mark will depend on your contribution in the project.
- Report organization and clarity.

Submission Deadlines:

Task	**Deadline**
Single-cycle CPU Design Demo	Week 13
Pipelined CPU Design Demo	Week 15
Final Report Submission	Week 15