COE 301 – Computer Organization

Term 201 – Fall 2020

Project: Pipelined Processor Design

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 to R31. 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 32-bit long and aligned in memory. The PC register (26-bit wide) contains the instruction address. The lower two bits of the PC register are hardwired to 0. All instruction addresses are multiple of 4. There are four instruction formats: R-type, I-type, SB-type, and J-type as shown below:

R-type

7-bit opcode (Op), 5-bit register numbers (*rs1*, *rs2*, and *rd*), and 10-bit function field (*f*)

f^{10}	$rs2^5$	$rs1^5$	rd^5	Op^7

I-type

7-bit opcode (Op), 5-bit register numbers (*rs1* and *rd*) and 15-bit Immediate

imm15	$rs1^5$	rd^5	Op^7

SB-type

7-bit opcode (Op), 5-bit register numbers (*rs1* and *rs2*) and 15-bit Immediate

imm10	$rs2^5$	$rs1^5$	$imm5^5$	Op^7

J-type

7-bit opcode (Op), 5-bit destination register number (*rd*), and 20-bit Immediate

imm20	rd^5	Op^7

Rs1 is the first source register number. This register is always read (never written). Ra is the name and value of register rs1.
Rs2 is the second source register number. This register is always read (never written). Rb is the name and value of register rs2.
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, SB-type, and J-type instructions, meanings, and encodings are shown below:

Instruction	Meaning	Encoding				
ADD	Rd=Ra + Rb	f=0	rs2	rs1	rd	Op=51
SUB	Rd=Ra – Rb	f=1	rs2	rs1	rd	Op=51
SLT	Rd=(Ra < Rb) signed	f=2	rs2	rs1	rd	Op=51
SLTU	Rd=(Ra < Rb) unsigned	f=3	rs2	rs1	rd	Op=51
XOR	Rd=Ra ^ Rb	f=4	rs2	rs1	rd	Op=51
OR	Rd=Ra | Rb	f=5	rs2	rs1	rd	Op=51
AND	Rd=Ra & Rb	f=6	rs2	rs1	rd	Op=51
SLL	Rd=ShiftLeftLogical(Ra, Rb[4:0])	f=7	rs2	rs1	rd	Op=51
SRL	Rd=ShiftRightLogical(Ra, Rb[4:0])	f=8	rs2	rs1	rd	Op=51
SRA	Rd=ShiftRightArith(Ra, Rb[4:0])	f=9	rs2	rs1	rd	Op=51
ADDI	Rd=Ra + sign_extend(imm15)	imm15		rs1	rd	Op=19
SLTI	Rd=(Ra < sign_extend(imm15)) signed	imm15		rs1	rd	Op=20
SLTIU	Rd=(Ra < sign_extend(imm15)) unsigned	imm15		rs1	rd	Op=21
XORI	Rd=Ra ^ zero_extend(imm15)	imm15		rs1	rd	Op=22
ORI	Rd=Ra | zero_extend(imm15)	imm15		rs1	rd	Op=23
ANDI	Rd=Ra & zero_extend(imm15)	imm15		rs1	rd	Op=24
SLLI	Rd=ShiftLeftLogical(Ra, sa)	0	sa	rs1	rd	Op=25
SRLI	Rd=ShiftRightLogical(Ra, sa)	0	sa	rs1	rd	Op=26
SRAI	Rd=ShiftRightArith(Ra, sa)	0	sa	rs1	rd	Op=27
LW	Rd=Mem[Ra+imm15]	imm15		rs1	rd	Op=28
SW	Mem[Ra+{imm10, imm5}]=Rb	imm10	rs2	rs1	imm5	Op=35
BEQ	if (Ra == Rb) PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=100
BNE	if (Ra != Rb) PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=101
BLT	if (Ra < Rb) PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=102
BGE	if (Ra >= Rb) PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=103
BLTU	if (Ra < Rb) unsigned PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=104
BGEU	if (Ra >= Rb) unsigned PC=PC+sign_extend({imm10, imm5}<<2)	imm10	rs2	rs1	imm5	Op=105
LUI	Rd=imm20 << 12	imm20			rd	Op=55
JAL	PC=PC+sign_extend(imm20<<2), Rd=PC+4	imm20			rd	Op=110
JALR	PC=Ra+sign_extend(imm15<<2), Rd=PC+4	imm15		rs1	rd	Op=111

Instruction Description

Opcodes 51 is used for R-format ALU instructions. There are 10 instructions in total.

Opcodes 19 through 28 are used for I-format instructions. There are 10 instructions in total.

The I-format ALU instructions (**ADDI** through **SRAI**) have identical functionality as their corresponding R-format instructions (**ADD** through **SRA**), except that the second ALU operand is an immediate constant. The **imm15** immediate value is sign extended for **ADDI**, **SLTI**, and **SLTIU**, while it is zero extended for **XORI**, **ORI**, and **ANDI**.

Opcode 28 define load word (**LW**) instruction. This instruction addresses 4-byte words in memory. The effective memory address = **Ra + sign_extend(imm15)**.

Opcode 35 define store word (**SW**) instruction. This instruction addresses 4-byte words in memory. The effective memory address = **Ra + sign_extend({imm10, imm5})**..

There are six branch instructions **BEQ** to **BGEU** with opcodes 100 to 105 and PC-relative addressing. If the branch is taken, then **PC = PC + sign_extend({imm10, imm5}<< 2)**.. Otherwise, **PC = PC + 4**. The conditional branch range is ± 4 KiB.

Opcode 55 define load upper immediate (**LUI**) instruction. **LUI** is used to build 32-bit constants and uses the J-type format. **LUI** places the **imm20** value in the upper 20 bits of the destination register ***rd***, filling the lowest 12 bits with zeros.

Opcodes 110 define the Jump-and-Link (**JAL**) instruction. PC-relative addressing is used to compute the jump target address: **PC = PC + imm20 << 2**. In addition, the **JAL** instruction saves the return address in **Rd** (**Rd = PC + 4**). If **Rd** is **R0** then the return address (**PC+4**) is not saved, and **JAL** becomes a Jump (**J**) pseudo-instruction. To use **JAL** instruction, follow the following syntax: **JAL $10, label**.

Opcode 111 defines the **JALR** (Jump-And-Link-Register) instruction. It saves the return address in **Rd** (**Rd = PC+4**) and does an indirect register jump with an offset (**PC = Ra + imm15 << 2**). If **Rd** is **R0** then the return address (**PC+4**) is not saved, and **JALR** becomes a Jump Register (**JR**) pseudo-instruction. To use **JALR** instruction, follow the following syntax: **JALR $10, $9, imm15**.

Memory

Although the architecture is 32 bits, the size of the instruction and data memories will be restricted. This is because the *Logisim* tool only supports small size memories.

Your processor will have separate instruction and data memories. The PC register should be restricted to 26 bits. The upper 24 bits address instructions in the instruction memory. The instruction memory can store 2^{24} instructions, where each instruction occupies four bytes. There are 16 million instructions in the instruction memory.

The data memory will be also restricted to 2^{24} words = 64 MiBytes. The data memory can be made *word addressable*, since only the LW and SW instructions address memory. Words should be always aligned in memory. The least-significant two bits of the address must be zeros, or simply ignored in the hardware implementation. Bits [25:2] of the ALU result address the data memory.

Addressing Modes

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

For taken branches: PC = PC + sign_extend(imm15<<2)

For JAL: PC = PC + sign_extend(imm20<<2)

For JALR: PC = Ra + sign_extend(imm15<<2)

For LW and SW, displacement addressing is used: Memory address = Ra + sign_extend(imm15)

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).

Register Rs1 is always read by all instructions (never written).

Register Rs2 is always read by R-type and SB-type instructions.

Register Rd is written by R-type and I-type instructions.

Arithmetic and Logic Unit (ALU)

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

ADD, SUB, SLT, SLTU, XOR, OR, AND, SLL, SRL, SRA

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 R30 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 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.

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 32-bit register.
3. 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