COE 301 – Computer Organization

Term 181 – Fall 2018

Project: Pipelined Processor Design

Objectives:

- Designing a Pipelined 32-bit 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 32-bit RISC processor with eight 32-bit general-purpose registers: R0 through R7. R0 is a normal register, NOT hardwired to zero. The program counter PC is a special-purpose 20-bit register. All instructions are only 16 bits. There are four instruction formats, R-type, I-type, B-type and J-type as shown below:

R-type 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-type format

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

Op^5	a^3	b^3	Imm5

B-type format

5-bit opcode (Op), 3-bit register number *a*, and 8-bit Immediate

Op^5	a^3	Imm8

J-type format

5-bit opcode (Op) and11-bit Immediate

Op^5	Imm11

Register Use

Register *a* can be either the first source register or destination register. It can be read and written. *Ra* is the value of register *a*. For R-type, register *a* is always the first source register. However for the I-type and B-type, register *a* can be the first source or destination register.

Register *b* is the second source register number, which is always read and never written. *Rb* is the value of source register *b*.

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

Instruction Encoding

Eight R-type, twelve I-type, ten B-type, and two J-type instructions are defined in the following table. The instructions, their meaning, and encoding are shown below:

Instruction	Meaning	Encoding				
AND	Rd = Ra & Rb	Op = 0	a	b	d	f = 0
OR	Rd = Ra | Rb	Op = 0	a	b	d	f = 1
XOR	Rd = Ra ^ Rb	Op = 0	a	b	d	f = 2
NOR	Rd = ~(Ra | Rb)	Op = 0	a	b	d	f = 3
ADD	Rd = Ra + Rb	Op = 1	a	b	d	f = 0
SUB	Rd = Ra – Rb	Op = 1	a	b	d	f = 1
SLT	Rd = Ra < Rb (Signed)	Op = 1	a	b	d	f = 2
SEQ	Rd = (Ra == Rb)	Op = 1	a	b	d	f = 3
ANDI	Ra = signed(Imm5) & Rb	Op = 4	a	b	Imm5	
ORI	Ra = signed(Imm5) | Rb	Op = 5	a	b	Imm5	
XORI	Ra = signed(Imm5) ^ Rb	Op = 6	a	b	Imm5	
NORI	Ra = ~(signed(Imm5) | Rb)	Op = 7	a	b	Imm5	
ADDI	Ra = signed(Imm5) + Rb	Op = 8	a	b	Imm5	
SUBI	Ra = signed(Imm5) – Rb	Op = 9	a	b	Imm5	
SLTI	Ra = signed(Imm5) < Rb (Signed)	Op = 10	a	b	Imm5	
SEQI	Ra = (signed(Imm5) == Rb)	Op = 11	a	b	Imm5	
SLL	Ra = Rb << unsigned(Imm5)	Op = 12	a	b	Imm5	
SRL	Ra = Rb >> unsigned(Imm5)	Op = 13	a	b	Imm5	
LW	Ra ← MEM[signed(Imm5) + Rb]	Op = 16	a	b	Imm5	
SW	MEM[signed(Imm5) + Rb] ← Ra	Op = 17	a	b	Imm5	
BEQZ	Branch if (Ra == 0)	Op = 20	a	Imm8		
BNEZ	Branch if (Ra != 0)	Op = 21	a	Imm8		
BLTZ	Branch if (Ra < 0)	Op = 22	a	Imm8		
BGEZ	Branch if (Ra >= 0)	Op = 23	a	Imm8		
BGTZ	Branch if (Ra > 0)	Op = 24	a	Imm8		
BLEZ	Branch if (Ra <= 0)	Op = 25	a	Imm8		
JR	PC = Ra + signed(Imm8 << 1)	Op = 26	a	Imm8		
JALR	R7 = PC+2; PC = Ra + signed(Imm8<<1)	Op = 27	a	Imm8		
SET	Ra = signed(Imm8)	Op = 28	a	Imm8		
SSET	Ra = {Ra[23:0], Imm8}	Op = 29	a	Imm8		
J	PC = PC + signed(Imm11<<1)	Op = 30	Imm11			
JAL	R7 = PC + 2; PC=PC+signed(Imm11<<1)	Op = 31	Imm11			

Instruction Description

Opcodes 0 and 1 are used for R-type ALU instructions. (Opcodes 2 and 3 are not used)

Opcodes 4 through 13 are used for I-type ALU instructions. Register *a* is the destination register. The 5-bit immediate constant replaces the first ALU operand, not the second one. It is always sign-extended for all I-type instructions. *Rb* is the second ALU operand.

The I-type ALU instructions (ANDI through SEQI) have identical functionality as their corresponding R-type instructions (AND through SEQ), except that the first ALU operand is a signed immediate (Imm5 with range -16 to +15), and the destination register is Ra (not Rd).

Programming Notes:

R1 = R2 – 1 is translated into ADDI R1, –1, R2 (Ra = R1, Rb = R2, and Imm5 = –1).

R1 = 1 – R2 is translated into SUBI R1, 1, R2 (Ra = R1, Rb = R2, and Imm5 = 1).

R1 = –R2 is translated into SUBI, R1, 0, R2.

R1 = R2 > 5 is translated into SLTI R1, 5, R2 (R1 = 5 < R2).

R1 = R2 ≥ 5 is equivalent to SLTI R1, 4, R2 (R1 = 4 < R2).

There are only two shift instructions: SLL and SRL with opcodes 12 and 13, respectively, with unsigned 5-bit immediate (shift amount) with values 0 to 31.

Opcodes 16 and 17 define the load word (LW) and store word (SW) instructions. These two instructions address 32-bit words in memory. Displacement addressing is used. The effective memory address = sign_extend(Imm5) + Rb. Register *a* is a destination register for LW, but a source for SW. Loading/storing a byte or half word are not defined to simplify the project.

Opcodes 20 through 25 define six branch instructions. Register *a* is source register for all branch instructions. The 32-bit value *Ra* is read and compared against zero. PC-relative addressing is used to define the target of a branch instruction. If the branch is taken, the 8-bit immediate is sign-extended and added to PC as follows:

If (branch is taken) PC = PC + sign_extend(Imm8<<1) else PC = PC + 2.

The PC register stores the address of a 16-bit instruction in memory. The address is always multiple of 2 (least-significant bit of PC register is always 0).

The JR (Jump-Register) instruction does a register-indirect jump, where Imm8 is left-shifted 1 bit and sign-extended: PC = Ra + sign_extend(Imm8<<1). The JALR (Jump-And-Link-Register) instruction saves the return address (PC+2) in R7.

The SET instruction (opcode 28) sets destination register *a* with an 8-bit signed constant. The immediate constant is sign-extended to 32 bits before writing register *a*. The SSET instruction (opcode 29) reads and writes register *a*. It shifts the value of register *a* left 8 bits and sets the lower 8 bits: Ra = {Ra[23:0], Imm8}, 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, 0x12 (first byte)
SSET R1, 0x34 (second byte)
SSET R1, 0x56 (third byte)
SSET R1, 0x78 (fourth byte)

Opcodes 30 and 31 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 writes the return address (PC + 2) in register R7.

Although the instruction set is reduced, it is still rich enough to write useful programs.

Memory

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

Your processor will have separate instruction and data memories. The PC register should be restricted to 20 bits. The instruction memory can store 2^{19} instructions, where each instruction occupies two bytes.

The data memory will be also restricted to 2^{20} bytes. 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 data address must be zeros, or simply ignored in the hardware implementation.

Addressing Modes

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

For taken branches: PC = PC + sign_extend(Imm8<<1)

For jumps: PC = PC + sign_extend(Imm11<<1)

For JR and JALR: PC = Rs + sign_extend(Imm8<<1)

To save the return address: R7 = PC + 2 (address of next instruction)

For LW and SW, displacement addressing is used: Address = Rs + sign_extend(Imm5)

Register File

Implement a Register file containing Eight 32-bit registers R0 to R7 with two read ports and one write port. R0 is a normal register that can be read and written (NOT hardwired to zero).

Arithmetic and Logic Unit (ALU)

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

AND, OR, XOR, NOR, ADD, SUB, SLT, SEQ, SLL, SRL

In addition, you should have special support for the SET and SSET 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 (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 (R0 to R7) at the top-level of your design. Provide output pins for registers R0 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 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, 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 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. 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.
2. Write a simple program that counts the number of 1's in a 32-bit register.
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

- 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.

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 during 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.