COE 301 / ICS 233 – Computer Organization

Term 172 – Spring 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 (*s*, *t*, and *d*), and 2-bit function field *f*

Op^5	s^3	t^3	d^3	f^2

I-type format

5-bit opcode (Op), 3-bit register numbers (*s* and *t*), and 5-bit Immediate

Op^5	s^3	t^3	Imm5

B-type format

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

Op^5	s^3	Imm8

J-type format

5-bit opcode (Op) and11-bit Immediate

Op^5	Imm11

For R-type instructions, *s* and *t* are the 3-bit source register numbers, and *d* is the 3-bit destination register number. *Rs* and *Rt* are the 32-bit values of source registers *s* and *t*, and *Rd* is the 32-bit value of the destination register. The 2-bit function *f* specifies four functions.

For I-type instructions, the 5-bit signed immediate (Imm5) is a second source operand with range -16 to +15. The source register is *s* and the destination register is *t*.

The B-type format is used by branch and jump-register instructions, where *s* is a source register. The 8-bit signed immediate is used for PC-relative and register-indirect addressing.

The J-type format is used by J (jump), JAL (jump-and-link), and for constant formation (SET and SSET). The 11-bit immediate is used for PC-relative addressing and constant formation.

Instruction Encoding

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

Instruction	Meaning	Encoding				
AND	Rd = Rs & Rt	Op = 0	s	t	d	f = 0
CAND	Rd = ~Rs & Rt (Complement Rs, AND)	Op = 0	s	t	d	f = 1
OR	Rd = Rs | Rt	Op = 0	s	t	d	f = 2
XOR	Rd = Rs ^ Rt	Op = 0	s	t	d	f = 3
ADD	Rd = Rs + Rt	Op = 1	s	t	d	f = 0
NADD	Rd = –Rs + Rt (Negate Rs, ADD)	Op = 1	s	t	d	f = 1
SLT	Rd = Rs signed< Rt	Op = 1	s	t	d	f = 2
SLTU	Rd = Rs unsigned< Rt	Op = 1	s	t	d	f = 3
ANDI	Rt = Rs & sign_extend(Imm5)	Op = 4	s	t	Imm5	
CANDI	Rt = ~Rs & sign_extend(Imm5)	Op = 5	s	t	Imm5	
ORI	Rt = Rs | sign_extend(Imm5)	Op = 6	s	t	Imm5	
XORI	Rt = Rs ^ sign_extend(Imm5)	Op = 7	s	t	Imm5	
ADDI	Rt = Rs + sign_extend(Imm5)	Op = 8	s	t	Imm5	
NADDI	Rt = –Rs + sign_extend(Imm5)	Op = 9	s	t	Imm5	
SLTI	Rt = Rs signed< sign_extend(Imm5)	Op = 10	s	t	Imm5	
SLTUI	Rt = Rs unsigned< sign_extend(Imm5)	Op = 11	s	t	Imm5	
SLL	Rt = Rs << Imm5	Op = 12	s	t	Imm5	
SRL	Rt = Rs zero>> Imm5	Op = 13	s	t	Imm5	
SRA	Rt = Rs sign>> Imm5	Op = 14	s	t	Imm5	
ROR	Rt = Rs rotate>> Imm5	Op = 15	s	t	Imm5	
LW	Rt ← MEM[Rs + Imm5]	Op = 16	s	t	Imm5	
SW	MEM[Rs + Imm5] ← Rt	Op = 17	s	t	Imm5	
BEQZ	Branch if (Rs == 0) (PC = PC + Imm8)	Op = 20	s	Imm8		
BNEZ	Branch if (Rs != 0) (PC = PC + Imm8)	Op = 21	s	Imm8		
BLTZ	Branch if (Rs < 0) (PC = PC + Imm8)	Op = 22	s	Imm8		
BGEZ	Branch if (Rs >= 0) (PC = PC + Imm8)	Op = 23	s	Imm8		
BGTZ	Branch if (Rs > 0) (PC = PC + Imm8)	Op = 24	s	Imm8		
BLEZ	Branch if (Rs <= 0) (PC = PC + Imm8)	Op = 25	s	Imm8		
JR	PC = Rs + Imm8	Op = 26	s	Imm8		
JALR	R7 = PC+1; PC = Rs + Imm8	Op = 27	s	Imm8		
SET	R0 = sign_extend(Imm11)	Op = 28	Imm11			
SSET	R0 = {R0<<11, Imm11}	Op = 29	Imm11			
J	PC = PC + Imm11	Op = 30	Imm11			
JAL	R7 = PC + 1; PC = PC + Imm11	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 15 are used for I-type ALU instructions. The 5-bit immediate constant is sign-extended for all I-type instructions.

The R-type ALU instructions (AND through SLTU) have identical functionality as their corresponding I-type instructions, except that the second source operand is a 5-bit signed immediate, and the destination register is Rt (not Rd).

The CAND and CANDI instructions complement the 32-bit value Rs.

The NADD instruction computes: Rd = –Rs + Rt, in which the 32-bit value Rs is negated and added to the 32-bit value Rt. Subtaction: SUB Rd, Rt, Rs is defined as a pseudo-instruction of NADD Rd, Rs, Rt, where the order of Rs and Rt is reversed.

The NADDI instruction computes: Rt = –Rs + Imm5. It was defined this way to be more useful with an immediate. NEG (negate) is a pseudo-instruction that is equivalent to NADDI Rt, Rs, 0.

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 = Rs + sign_extend(imm5). Rt is a destination register for LW, but a source register for SW.

Opcodes 20 through 25 define six branch instructions. The 32-bit value Rs is 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.

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

The PC register stores the address of a 16-bit instruction in memory. Since all instructions are aligned in memory and 2-byte long, there is no need to store the least-significant zero bit. Therefore, the PC register points to an instruction in memory, NOT a byte address. The address of the next instruction in memory is (PC+1).

The JR (Jump-Register) instruction does a register-indirect jump: PC = Rs+sign_extend(Imm8), where Imm8 is a signed displacement. The JALR (Jump-And-Link-Register) instruction saves the return address (PC+1) in R7.

The SET instruction (opcode 28) sets destination register R0 with an 11-bit constant. The 11-bit immediate constant is sign-extended to 32 bits before writing register R0. Register R0 is always the destination register for SET.

The SSET instruction (opcode 29) reads and writes R0. It shifts register R0 left 11 bits and sets the lower 11 bits: R0 = {R0<<11, Imm11}, where {} means concatenation. The SSET instruction can be used three times to form any 32-bit constant in register R0. SET and SSET can also be used together for constant formation.

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). In addition, the JAL instruction writes the return address (PC + 1) 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. It addresses instructions (not bytes) in the instruction memory. The instruction memory can store 2^{20} instructions, where each instruction occupies two bytes.

The data memory will be also restricted to 2^{20} words = 4 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.

Addressing Modes

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

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

For jumps: PC = PC + sign_extend(Imm11)

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

To save the return address: R7 = PC + 1 (address of next instruction)

For LW and SW, displacement addressing is used: Memory address = Rs + sign_extend(Imm8)

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). It is used by SET and SSET for constant formation.

Register Rs is always read by all instructions (never written)

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

Register Rd is written by R-type instructions

In addition, SET and SSET write register R0 implicitly, and JAL writes register R7 implicitly.

Arithmetic and Logic Unit (ALU)

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

ADD, NADD, SLT, SLTU, AND, CAND, OR, XOR, SLL, SRL, SRA, ROR

In addition, you should have special support for the 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.

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 R7) at the top-level of your design. Provide output pins for all registers R0 through R7, 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 SET and SSET 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.

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.