COE 308: Computer Architecture – Fall 2011

Project: Pipelined Processor Implementation

Objectives:

- Using the Logisim simulator
- Designing and testing a Pipelined 32-bit processor
- Teamwork

Instruction Set Architecture

In this project, you will design a simple 32-bit RISC processor with sixteen 32-bit general-purpose registers: R0 through R15. R0 is hardwired to zero and cannot be written, so we are left with fifteen registers. There is also one special-purpose 24-bit register, which is the program counter (PC), which can address at most 2^{24} instructions. All instructions are 32 bits. There are three instruction formats, R-type, I-type, and J-type as shown below:

R-type format

8-bit opcode (Op), and 4-bit register numbers (Rs, Rt, and Rd)

Op^8	Rs^4	Rt^4	Rd^4	Unused^{12}

I-type format

8-bit opcode (Op), 4-bit register number (Rs and Rt), and 16-bit immediate constant

Op^8	Rs^4	Rt^4	Immediate^{16}

J-type format

8-bit opcode (Op) and 24-bit immediate constant

Op^8	Immediate^{24}

For R-type instructions, Rs and Rt specify the two source register numbers, and Rd specifies the destination register number.

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 16 bits as in the MIPS architecture. The 16-bit immediate constant can be signed or unsigned depending on the opcode.

For J-type, the 24-bit immediate constant is used for J (jump) and JAL (jump-and-link) instructions.

Instruction Encoding

Thirteen R-type instructions, nine I-type instructions, and two J-type instructions are defined. These instructions, their meaning, and their encoding are shown below:

Instr	Meaning	Encoding				
AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 0x00	Rs	Rt	Rd	Unused
OR	Reg(Rd) = Reg(Rs) | Reg(Rt)	Op = 0x01	Rs	Rt	Rd	Unused
XOR	Reg(Rd) = Reg(Rs) ^ Reg(Rt)	Op = 0x02	Rs	Rt	Rd	Unused
NOR	Reg(Rd) = ~(Reg(Rs) | Reg(Rt))	Op = 0x03	Rs	Rt	Rd	Unused
ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 0x04	Rs	Rt	Rd	Unused
SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 0x05	Rs	Rt	Rd	Unused
SLT	Reg(Rd) = Reg(Rs) < Reg(Rt)	Op = 0x06	Rs	Rt	Rd	Unused
SLTU	Reg(Rd) = Reg(Rs) $<_{\text{unsigned}}$ Reg(Rt)	Op = 0x07	Rs	Rt	Rd	Unused
SLL	Reg(Rd) = Reg(Rs) << Reg(Rt)	Op = 0x08	Rs	Rt	Rd	Unused
SRL	Reg(Rd) = Reg(Rs) zero>> Reg(Rt)	Op = 0x09	Rs	Rt	Rd	Unused
SRA	Reg(Rd) = Reg(Rs) sign>> Reg(Rt)	Op = 0x0A	Rs	Rt	Rd	Unused
ROR	Reg(Rd) = Reg(Rs) rot>> Reg(Rt)	Op = 0x0B	Rs	Rt	Rd	Unused
JR	PC = Reg(Rs)	Op = 0x10	Rs	0	0	Unused
ANDI	Reg(Rt) = Reg(Rs) & Immediate^{16}	Op = 0x20	Rs	Rt	Immediate^{16}	
ORI	Reg(Rt) = Reg(Rs) | Immediate^{16}	Op = 0x21	Rs	Rt	Immediate^{16}	
XORI	Reg(Rt) = Reg(Rs) ^ Immediate^{16}	Op = 0x22	Rs	Rt	Immediate^{16}	
ADDI	Reg(Rt) = Reg(Rs) + Immediate^{16}	Op = 0x24	Rs	Rt	Immediate^{16}	
LUI	Reg(Rt) = Immediate^{16} << 16	Op = 0x28	0	Rt	Immediate^{16}	
LW	Reg(Rt) = Mem(Reg(Rs) + Imm^{16})	Op = 0x30	Rs	Rt	Immediate^{16}	
SW	Mem(Reg(Rs) + Imm^{16}) = Reg(Rt)	Op = 0x38	Rs	Rt	Immediate^{16}	
BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 0x40	Rs	Rt	Immediate^{16}	
BNE	Branch if (Reg(Rs) != Reg(Rt))	Op = 0x41	Rs	Rt	Immediate^{16}	
J	PC = Immediate^{24}	Op = 0x50	Immediate^{24}			
JAL	R15 = PC + 1, PC = Immediate^{24}	Op = 0x51	Immediate^{24}			

Opcodes 0x00 thru 0x0B are used for R-type arithmetic and bitwise instructions. There are three shift and one rotate instruction. To shift or rotate, use the lower 5 bits of register Rt as the shift/rotate amount. Opcode 0x10 is used for the JR (jump register) instruction. Opcodes 0x20 thru 0x41 are used for I-type instructions. The 16-bit immediate constant is zero-extended for ANDI, ORI, and XORI. It is sign-extended for all remaining instructions. The Load Upper Immediate (LUI) shifts the immediate constant left by 16 bits to load it into the upper 16 bits of register Rt. LUI can be combined with ORI to load any 32-bit constant into a register. The J-type instructions have a 24-bit immediate constant. Although the instruction set is reduced, it is still rich enough to write useful programs. We can have procedure calls and returns using the JAL and JR instructions.

Memory

Your processor will have separate instruction and data memories with 2^{24} words each. Each word is 32 bits or 4 bytes. Memory is *word addressable*. Only words (not bytes) can be read and written to memory, and each address is a word address. This will 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 4) 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 fifteen 32-bit registers R1 to R15 with two read ports and one write port. R0 is hardwired to zero.

Arithmetic and Logical Unit (ALU)

Implement a 32-bit ALU to perform all the required operations:
ADD, SUB, SLT, SLTU, OR, AND, XOR, NOR, SLL, SRL, SRA, ROR

Addressing Modes

PC-relative addressing mode is used for branch instructions.
For branching (BEQ, BNE), the branch target address is computed as follows:
PC = PC + sign-extend(Imm16). Add the contents of PC to the sign-extended 16-bit Immediate.
Direct addressing mode is used for jumps (J and JAL): PC = immediate24.
For LW and SW base-displacement addressing mode is used. The base address is obtained from register(Rs) and added to the sign-extended 16-bit immediate to compute the effective memory address. Only the lower 24 bits of the address are used to address the data memory.

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 may also have a stack segment if you want 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 can be implemented completely 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 correctness. Then, convert your design and implement a pipelined-datapath and its control logic. A five-stage pipeline should be constructed similar to the pipeline presented in the class lectures. Add pipeline registers between stages. Design the control logic to detect data dependencies among instructions and implement the forwarding logic. For jump instructions, reduce the jump delay to just one cycle. Stall the pipeline for one clock cycle after a jump instruction. For branch, assume the branch is never taken. There is no need to stall the pipeline if a branch instruction is not taken. However, if a branch is taken then convert the wrongly fetched instructions into bubbles. Also, stall the pipeline after a LW instruction, if it is followed by a dependent instruction.

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 different test programs to verify the correctness of the pipeline implementation. Write a test program involving procedures and arrays. For example, the main procedure initializes array elements with some constant values. It then calls a 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. Translate the program into machine instructions by hand and load it into the instruction memory starting at address 0. You can also save the image of the instruction and data memories into files and reload them later for testing purposes.
- Document all your test programs and files and include them in the report document.
- Make several copies and versions of your design before making changes, in case you need to go back to an older version.

Project Poster and Presentation

Prepare a poster for your project that includes the group members, a brief description of the project, diagrams of the datapath and control, test cases used to verify the correctness of the pipelined implementation, and snapshots of the simulator. You will use this poster to present your project. You will also prepare a live demo for your pipelined processor. Submit an electronic copy of the poster. Some of the best project posters will be preserved as demos for future students, and will be posted on the COE 308 website.

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.
- 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.

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 Blackboard.

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
- ■ Poster Presentation
- ■ Report document

Late policy

The project should be submitted on the due date by midnight. Late projects are accepted for a maximum of 2 late days. Projects submitted after 2 late days will not be accepted. The maximum late penalty is 10%.