PORTLAND STATE UNIVERSITY

ECE506 PROJECT - RISC-V IN SYSTEMVERILOG

FALL 2019

GENOVESE, R. IGNACIO

Introduction

RISC-V is an open-specification Reduced Instruction Set Architecture, first developed in 2010 at the University of California, Berkeley, that enables designers to easily extend it. By this, engineers don't need to find a processor that fit their needs from different vendors (such as Intel, AMD or ARM), but instead they can do a design themselves. In order to keep the ISA coherent, the RISC-V specifications provide a set of guidelines for the standard extension of the ISA.

RISC-V is actually a family of related ISAs, of which there are currently four base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, which provide 32-bit or 64-bit address spaces respectively. RV32E is a subset variant of the RV32I base instruction set, which has been added to support small microcontrollers, which has half the number of integer registers. RV128I is variant of the base integer instruction set supporting a flat 128-bit address space.

Proposal

The proposal for this project is to implement, in SystemVerilog, a pipelined version of a reduced version of the RV32I RISC-V 32-bit ISA that could be used for educational purposes, in general and for Electrical and Computer Engineering students at Portland State University, in particular. The project should include a basic testbench for verifying the correct implementation of each instruction. The testbench shall generate instructions to fill an instruction memory, read and executed by the processor.

The proposed instructions to implement are 43:

- Integer computational Instructions:
    - Integer Register-Immediate Instructions: ADDI, SLTI, SLTIU, MV, ANDI, ORI, XORI, SEQZ, NOT, SLLI, SRLI, SRAI, LUI, AUIPC.
    - Integer Register-Register Instructions: ADD, SLT, SLTU, SNEZ, AND, OR, XOR, SLL, SRL, SUB, SRA, NOP.
- Control Transfer Instructions: JAL, J, JALR, BEQ, BNE, BLT, BLTU, BGE, BGEU.
- Load/Store Instructions: LW, LH, LHU, LB, LBU, SW, SH, SB.

Instructions from the RV32I ISA that will not be implemented are:

- FENCE (memory ordering) instructions.
- SYSTEM (ECALL, EBREAK) instructions.
- HINT instructions.

Design

As stated in the proposal, the project was implemented in SystemVerilog, using constructs and methodologies learnt while taking ECE571 - SystemVerilog and ECE586 - Computer Architecture classes. QuestaSim was used to debug the design and run simulations, using the MCECS servers.

Following the architecture proposed in chapter 4 of “Computer Organization and Design”, by Patterson and Henessy, the pipeline consists of five stages:

- Instruction Fetch: in this stage instructions are read from an instruction memory, using a program counter.
- Instruction Decode: instructions are decoded, according to the RISC-V R32VI ISA specifications, control signals are generated and operands are obtained.
- Execute: this stage performs arithmetic/logic operations.
- Memory: in this stage data memory is read and written.
- Writeback: results are written to the Register File, if necessary.

The following figure shows the proposed architecture:

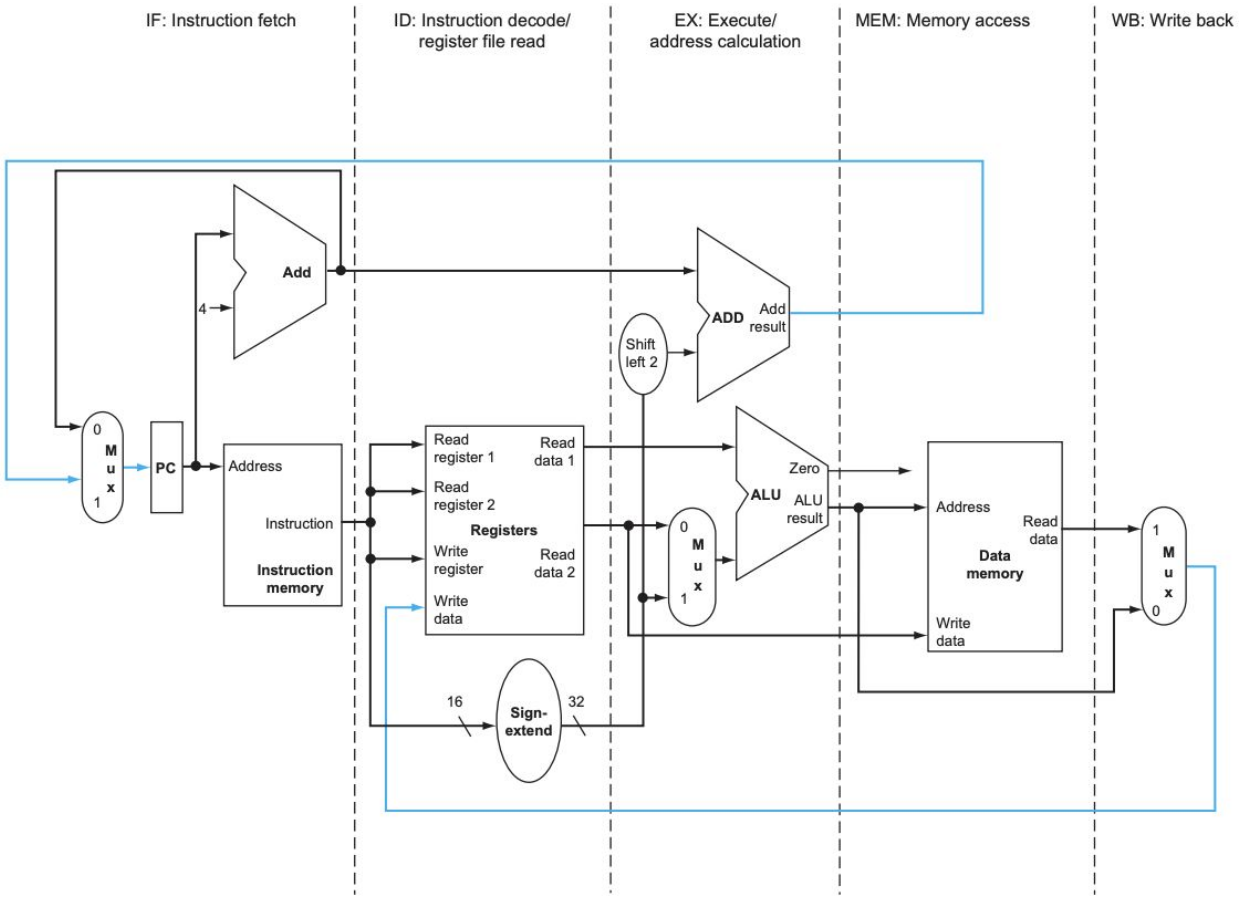

Figure 1 - 5-stage pipeline.

Logic in each stage is combinational (except in the instruction fetch stage) and results from each stage are stored in intermediate registers. These registers allow pipelining the design and having different instructions processed in each stage at the same time. This pipelining of the design produces the emergence of the following hazards:

- Structural hazards: emerge when two instructions want to use the same resource at the same time. Overcoming these hazards, it's possible to execute without conflicts almost every combination of instructions. In order to to this, the following actions were taken:
    - Separate instruction and data memories, to avoid the overlap of Instruction Fetch and Memory stages.

    - Writing registers in the first half of the clock cycle and reading them on the other half.
    - Duplication ALU hardware, for the overlapping of Instruction Fetch and Execute stages when calculating the next program counter and other ALU operations.
- Data hazards: emerge when an instruction wants to use data produced by a previous instruction that has not finished yet. To overcome these hazards the following actions were taken:
    - Forwarding unit: forward results from one stage to another, before they are written to the Register File.
    - Hazard detection unit: there's one case that can not be solved by the forwarding unit and needs to stall the pipeline, that is when a load instruction is followed by an instruction that needs its result.
- Control hazards: emerge when trying to solve a branch instruction over a condition that's not yet ready to be evaluated. As the condition evaluation and the calculation of the next program counter is done in the Execution stage, the two following instructions would already be in the pipeline. To avoid this, the detection of the branch/jump instruction, as well as the condition evaluation and the calculation of the next program counter, is done in the Instruction Decode stage. If the branch is taken, or if the instruction is a jump, the instruction already in the Instruction Fetch stage needs to be flushed.

Implementation

We began by defining parameters, typedefs and enumerations, following the ISA specifications, for the six different types of instructions (shown below). These can be found in the *riscv_defs.sv* file, inside the **riscv_defs** package.

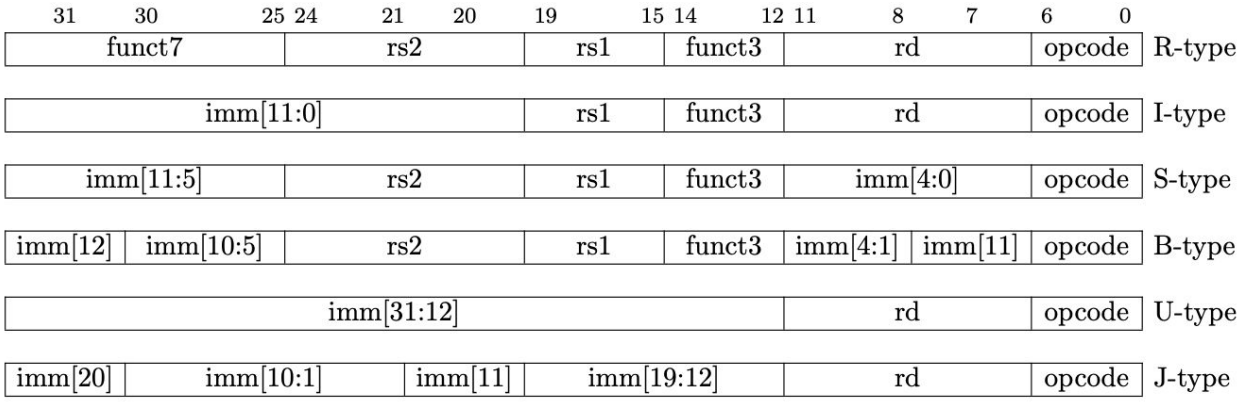

Figure 2 - Instruction Types

We proceed by defining the data and instruction memories and their interfaces (*dmem_if.sv* and *imem_if.sv*, respectively). Then, we implemented each pipeline stage.

Instruction Fetch

This stage fetchs an instruction from the instruction memory, making use of the *imem_if* interface and its corresponding modport. It also calculates the next cycle program counter, if a load hazard was not detected. The next cycle program counter is obtained by adding 4 to the current program counter (as the instruction memory is byte addressed and each instruction is 4 bytes long) if the current instruction is not a taken branch. Otherwise, the next cycle program counter is the branch/jump target address calculated in the decode stage.

Instruction Decode

This stage contains the following modules:

- Branch/Jump unit: detects branch/jump instructions, calculates the target address and evaluates the condition of branch instructions. It also produces the branch taken and flush signals. For jump instructions, it drives signals to indicate the decode unit to store the return address in the Register File.
- Hazard detection unit: detects load hazards by comparing the destination register of the instruction in the execution stage with the source registers of the instruction in the decode stage, if the instruction in the execution stage is a load. It also drives the branch taken signal as a branch hazard.
- Decode unit: contains the Register File, writes it (according to input signals from the branch/jump unit and the Writeback stage) and obtains source operands for the instruction in the decode stage (by reading the register file, the immediate value in the current instruction or forwarded values).
- Control unit: reads the instruction and drives the following control signals, making use of a user-defined structure called *control_bus_t* (defined in **riscv_defs** package):
    - alu_src1: used by the Execution stage, selects source 1 between program counter or source 1 driven by the decode unit.
    - Alu_src2: used by the Execution stage, selects source 2 between the immediate value or source 2 driven by the decode unit.;
    - alu_op: used by the Execution stage, controls operation done by ALU.

- arith_logic: used by the Execution stage, controls whether the current operation is arithmetic or logic.
- rd: used by the Execution stage, specifies the destination register.
- dmem_rd: used by the Memory stage, indicates if the instruction is a load.
- dmem_wr: used by the Memory stage, indicates if the instruction is a store.
- ld_st_funct3: used by the Memory stage, indicates the load/store size (byte, half-word or word)
- rf_wr: used by the Writeback stage, indicates if the instruction writes its result to a register.
- wb_to_rf: used by the Writeback stage, indicates if the result to write to Register File comes from the data memory or the ALU.

Finally, it's important to highlight that if a hazard is detected in this stage or the previous instruction produced a flush, the registered control bus is driven a zero value, to avoid flushed instructions to modify the state of the processor.

Execution Stage

The execution unit contains the arithmetic/logic unit (ALU) to perform these operations. Also, in this unit the source operands are selected between the driven operands by the decode unit, the immediate value, the program counter and forwarded results from other stages, according to the signals driven by the control unit.

Memory Stage

The memory unit communicates with the data memory (using the *dmem_if* interface and its corresponding modport) to read/write values, according to signals driven by the control unit. These values can be entire words, half-words or bytes, according to the *funct3* value of the instruction.

Writeback Stage

As this stage is very simple, it was not necessary to implement it in a module. Instead, it's implemented in the processor's top module. It drives signals to write the result from the ALU or values obtained from the data memory to the Register File in the Decode stage.

Forwarding Unit

Finally, the forwarding unit was implemented. This is not a stage in the pipeline. It drives signals to forward results from the Execution, Memory and Writeback stages to source operands of the Decode and Execution stages. In particular:

- If the instruction in the Execution stage writes the Register File, and its destination register is not register 0 (whose value is always 0), the unit compares this destination register with the source registers of the instruction in the Decode stage. If they match, then the unit drives a signal to forward the result of the Execution stage to the operands of the Decode stage.
- If the instruction in the Memory stage writes the Register File, and its destination register is not register 0, the unit compares this destination register with the source registers of the instruction in the Decode stage. If they match, then the unit drives a signal to forward the result of the Memory stage to the operands of the Decode stage.
- If the instruction in the Memory stage writes the Register File, and its destination register is not register 0, the unit compares this destination register with the source registers of the instruction in the Execution stage. If they match, then the unit drives a signal to forward the result of the Memory stage to the operands of the Execution stage.
- If the instruction in the Writeback stage writes the Register File, and its destination register is not register 0, the unit compares this destination register with the source registers of the instruction in the Execution stage. If they match, then the unit drives a signal to forward the result of the Writeback stage to the operands of the Execution stage.

Top Module

Lastly, these modules were instantiated and connected in the *riscv_top.sv* top module, which also implements the intermediate registers between stages.

Debug and Verification

In order to verify the correct behavior of the design, a testbench was developed. This testbench read assembled programs from files and populates the instruction memory with them. To produce the machine code from assembly code an assembler program was used (https://github.com/metastableB/RISCV-RV32I-Assembler). For each file/testcase, it runs the processor a certain amount of cycles (enough to execute every instruction) and compares the final state of the Register File with an expected one (read from another file). This Register File represents the desired architectural state of the processor at the end of the execution of the program. If these differ then the testcase is said to have failed, otherwise, it's said to have passed.

The following testcases were defined:

- test: contains just one addi instruction, used to begin debugging the design.
- typer_i_s: implements and tests every R, I, S and U instructions (except jumps).
- jump: implements and tests every jump instruction.
- branch: implements and tests every branch instruction.
- forward: tests every forwarding condition.
- hazard: tests every hazard condition.

To run these testcases, a Makefile is provided, along with a wave source file for debugging.

Project Structure

The project was developed using a Git repository for version control (https://github.com/igenovese/ece506_riscv) and it has the following structure:

├── docs	Contains project documentation
├── sim	Contains project Makefile and wave source file
└── src	
├── rtl	Contains SystemVerilog source code
└── tb	Contains testbench
└── testcases	Contains assembly source code for the testcases
├── assembly	Contains machine source code for each testcase
└── rf_results	Contains the expected Register File for each testcase