Technical University of Denmark

02114 Design of a RISC-V microprocessor

Group 15

Søren Peter Loff Christensen - s235483

Bertram Fink-Jakobsen - S235474

Victor Ding - S235468

Contributions

We have worked in collaboration, and everyone have taken part of every aspect of the assignment. Everything implemented was discussed to make sure everyone was in agreement and understood all the aspects of the design.

Large Language models such as Chat GPT o1 were used for parts of the assignment, everything has been reviewed and edited by a human.

23. januar 2025

1 Introduction

This report presents the design and implementation of the "Food"Processor, a 5-stage pipelined hardware processor designed to execute the RISC-V RV32I instruction set. The processor is structured to simulate the behavior of a processing pipeline, integrating key stages: fetch, decode, execute, memory, and writeback. These stages work together seamlessly to perform operations such as instruction fetching, decoding, execution, computation of results, and memory operations including store and load.
Particular focus was placed on achieving hardware synthesis compatibility and addressing errors encountered during elaboration and testing.
This work highlights the challenges and solutions involved in designing a robust digital processor. Objectives include ensuring functional correctness, optimizing inter-component communication within the pipeline, and resolving synthesis compatibility issues.This project demonstrates a structured and scalable approach to processor design.

2 Design Overview

2.1 Processor Design Overview

This course's Food Processor is a classical 5-stage pipelined processor implementing the RISC-V `RV32I` instruction set. The design focuses on efficient execution through pipelining, handling hazards using techniques such as forwarding, stalling, and flushing.

Data Flow and Control Flow

The processor pipeline is designed to effectively handle different types of hazards (data hazards and control hazards) through:

- **Forwarding**: Bypassing the result of an operation directly to the subsequent instruction that needs it.
- **Stalling**: Delaying instruction execution in earlier pipeline stages until necessary data is available.
- **Flushing**: Discarding partially completed instructions when a control hazard (e.g., branch) alters the normal flow.
- **Branching**: Branching will happen over 2 clock cycles since it happens under execute stage.

1. Fetch (IF) Stage

The Fetch stage retrieves the instruction from the instruction memory using the Program Counter (PC) and computes the address of the next instruction. If the instruction is a branch or jump, the PC is updated accordingly to ensure that the correct instruction is fetched next.

2. Decode (ID) Stage

In the Decode stage, the 32-bit instruction is parsed to identify the opcode, destination register (`RD`), source registers (`RS1` and `RS2`), and any immediate values required. The control unit also determines which control signals will be needed in subsequent stages. If a load instruction is followed by one that uses the loaded register immediately, a stall is introduced to allow the data to become available.
In the decoder we also do the NOP instruction for when a forwarding happens

3. Execute (EX) Stage

During the Execute stage, the required operands (from registers or forwarded results) are fed into the Arithmetic Logic Unit (ALU). The ALU performs arithmetic, logical, or shift operations based on the opcode. Branch decisions are resolved here; if a branch is taken, the pipeline flushes instructions that are no longer valid. The correct control signals for memory access are also passed to the next stage as needed.
Branching and flushing also happen in this stage. When a case of flushing is to appear, we will output the value of rd to be 0. Since x0 is hardcoded to be 0, nothing will happen. A counter will also happen to know how many times it should flush. THis module can be seen in the block diagram right above ALU
In case of branching or jump and link, we will send a signal to the fetcher so it knows where to start the indexing for out instruction.

4. Memory (MEM) Stage

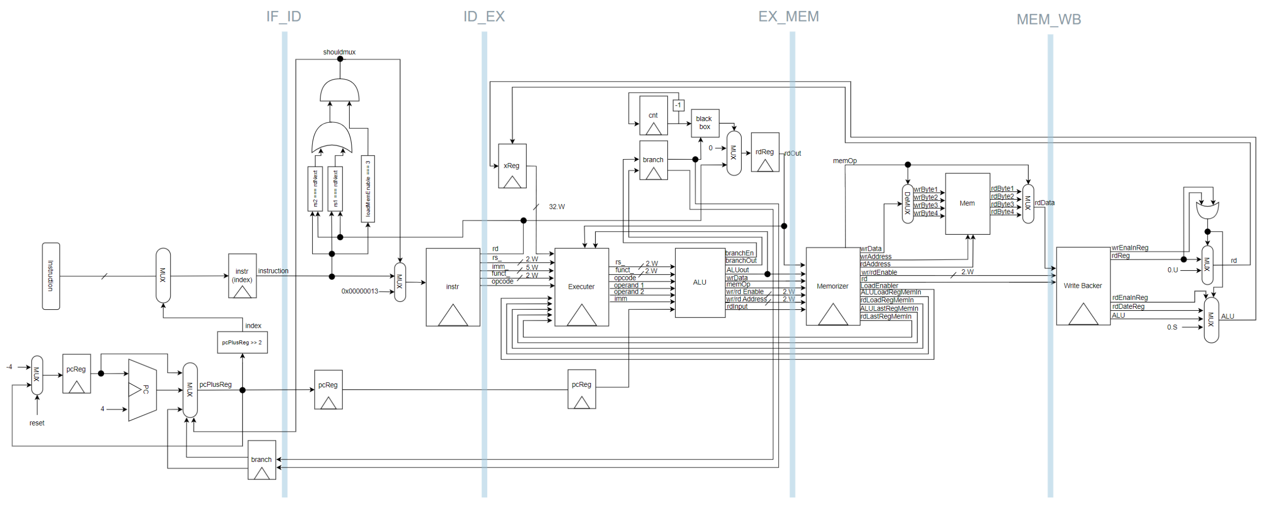
In the Memory stage, data memory is accessed for load and store operations. Load instructions retrieve data from memory and forward it for subsequent usage, while store instructions write data to the appropriate location in memory. Our Memory is written into 4 different modules dependent on offset and memory operation. The reading is also affected by these factors, and thus it concatenates the bytes it should read, and send it to write back.

5. Writeback (WB) Stage

Finally, the Writeback stage writes the result of an ALU operation or memory load back into the designated destination register (`RD`). This completes the instruction's execution.
Key Features:

- Classic 5-stage pipeline (IF, ID, EX, MEM, WB).
- Implements RISC-V `RV32I` instruction set.
- Hazards handled via forwarding, stalling, and flushing.
- Separate control unit for branch resolution and pipeline management.
- Dedicated memory stage for load/store instructions.

2.2 Block Diagram



Figur 1: Block diagram for i6 Food Processor

3 Implementation Details

3.1 Chisel Modules

For our code we followed a modular approach. This approach helped us keep the code organized and somewhat clean. A top-level module is used to connect all the stages and handle pipeline registers.

Food.scala:

Fetcher.scala:

In the fetcher module, we control when the PC counter should jump based on what happens in `Executer`. We feed the PC counter based on the branch jump value, and if the branch has been enabled. This can be seen in the code below:

```
val pcPlusReg = Mux(io.shouldMux, pcReg, Mux(io.branchEnable, io.branchIn, pcReg + 4.S))
```

Decoder.scala

Here we slice the instruction into different operands, and pass everything into the execute stage

```
    io.opcode := instr(6, 0)
  io.imm_S := ((instr(31, 25) << 5) | instr(11, 7)).asSInt
  io.imm_B := ((instr(31) << 12) | (instr(7) << 11) | (instr(30, 25) << 5) | (instr(11, 8)
      << 1)).asSInt
  io.imm_U := instr(31, 12)
  io.imm_J := (((instr(31) << 20) | (instr(19, 12) << 12) | (instr(20) << 11) | (instr(30,
      21) << 1))).asSInt
```

Executer.scala

The main purpose of the executer.scala module is to take in all of the data coming from the decode stage and sort it, in order to only sent the needed operands further to the ALU module. here we also check if we need to use a forwarded operand. We do that in the following bit of code:

```
    val rs1Wire = Mux(lastExBool1, aluLastRegEx, Mux(lastMemBool1, Mux(loadMemBool1, io.
        aluLoadRegMemIn, io.aluLastRegMemIn), io.x(rs1Reg)))
```

ALU.scala

In the following snippet is it shown how the ALU switches for group, which is a control signal from executer, and performs operations using the parsed operands.

```
    switch(group) {
    is(1.U) { // R and I types
      switch(funct3) { // does all the simple R and I type instructions
        is(0x0.U) {
          io.ALUout := operand1.asSInt + operand2
          switch(funct7) {
            is(0x20.U) {
              io.ALUout := operand1.asSInt - operand2
            }
        }
      }
```

Memorizer.scala

The first part concatinates the read data from a lw operation and sends it to the write back. The second part writes the correct byte value into each memory bank.

```
io.rdData := Cat(readByte(3), readByte(2), readByte(1), readByte(0)).asSInt

when(wrEnaReg) {
    for (i <- 0 until 4) {
        mem(i).write(writeAddress, writeByte(i))
    }
}
```

Write Backer.scala

The first part controls wether to read a value from the register or read a value from the memory, dependent on read enable.
The second part is to control wether or not to save into a register. If write is enabled or rd is 0, then it will not save to the register file. This is used for flushing and stalling, where the rd is set to 0, and thereby the data is not saved to the register file.

```
  io.ALUoutput := Mux(rdEnaInReg, rdDataReg, ALUreg)

  when(wrEnaInReg || rdReg === 0.U) {
    io.ALUoutput := 0.S
    io.rdOut := 0.U
  }
```

4 Testing and Evaluation

Most of our test were done in simulation by running our scala test. this printed out the stored values of 6 registers, x1-x6, every clock cycle. from this we could run instructions and read through the cycles to check if all the values stored are correct. In addition to this we also used GTK Wave in order to follow signals and debug when our scala test presented issues.
In the last couple of days of the project we managed to get the design working on our FPGA board and developed a snooping mechanism which displays data stored, on a register chosen by the switches, on the seven segment display. This made us able to test different instructions on the board and check the correctness of our design.
Throughout the project we spent alot of time debugging and getting our modules to work together correctly. the two main struggles were: integrating a redesigned ALU late in the project which took a lot of debugging. and redoing our memory stage, we had written the stage and simulated it in chisel just fine, although in order for our design to work on FPGA, we had to redo it.