MTH410E – RISC-V Architecture and Processor Design

Final Assignment: Superscalar RiscV Core in SystemVerilog

Deniz Bashgoren 040180902

1. Introduction

This project is the implements a RiscV processor with the following properties:

- Processor implements the RV32I subset of the ISA.
- Processor has a 4-stage pipeline: fetch, decode & issue, execute and writeback.
- Processor is in-order (I4).
- Processor has two paths: (1) ALU + BranchComparator, (2) ALU + DataMem.
- Processor assumes an instruction memory with 2 outputs: mem[pc] and mem[pc+4].
- Processor assumes a data memory with a single port used for reading and writing.
- Processor implements a register file with 4 reading ports and 2 writing ports.
- Processor doesn't support exceptions, traps, interrupts, CSRs.
- Processor has no GPIO ports, and no peripherals.
- Processor uses two seperate memory spaces, one for fetching instructions, one for data.
- Both memory spaces can be set to be in Little Endian or Big Endian. Big Endian works on instructions too because all instructions have a fixed 4-byte size.
- Processor works fine on unnatural alignment. Addresses (in both address spaces) wrap around the end. That is, accessing `(word) mem[LEN-1]` will give `mem[LEN-1] through `mem[2]`.

2. Source Code Organization

The main folder contains the following:

- **src/** : The folder with the source code. The top module is riscv_superscalar.sv.
- **tb/** : The testbench modules. Has riscv_superscalar_tb.sv only.
- **samples/** : The folder containing simple C programs to test the core. Each sample consists of a C source file (like prime.c), an elf executable file, compiled by GCC (prime.c.elf), an disassembler output file (prime.c.dump), and hex files ready to be imported by SystemVerilog, extracted from the elf file (prime.c.elf.imem and prime.c.elf.dmem).
- **utils/** : A script written in javascript that generates *.imem and *.dmem files from a given *.elf file. To parse the elf file it uses the library node-elf-file [1].
- **makefile** : The intended way to build the project.
- **compilation.ld** : A linker script used to reorder the memory mapping of the sections.
- **Report.pdf** : This file.

3. Required Software

Here it's assumed that the user runs a GNU/Linux machine that has access to basic tools like make. The software needed to run the core:

- **GNU Compiler Collection for RiscV** : This is used to compile the sample C files to RiscV binary files. Run the **`riscv64-linux-gnu-gcc`** command to make sure it's installed.
- **NodeJS** : This is a javascript runtime that is used to run the scripts to prepare the *.dmem and *.imem files. Run the **`node`** command to make sure it's installed.
- **Icarus Verilog** : The SystemVerilog simulator used for this project.

4. How to Run

The procedure is the same as in 2nd and 3rd assignments. To run the provided tests:

1. `cd` to the project's root directory.
2. Edit the file `src/riscv_superscalar.sv` and update DMemInitFile, IMemInitFile, INITIAL_PC_ADDRESS.
3. Run `make cpu`.
4. Done. The results are in `logFile` and `riscv_superscalar.vcd`

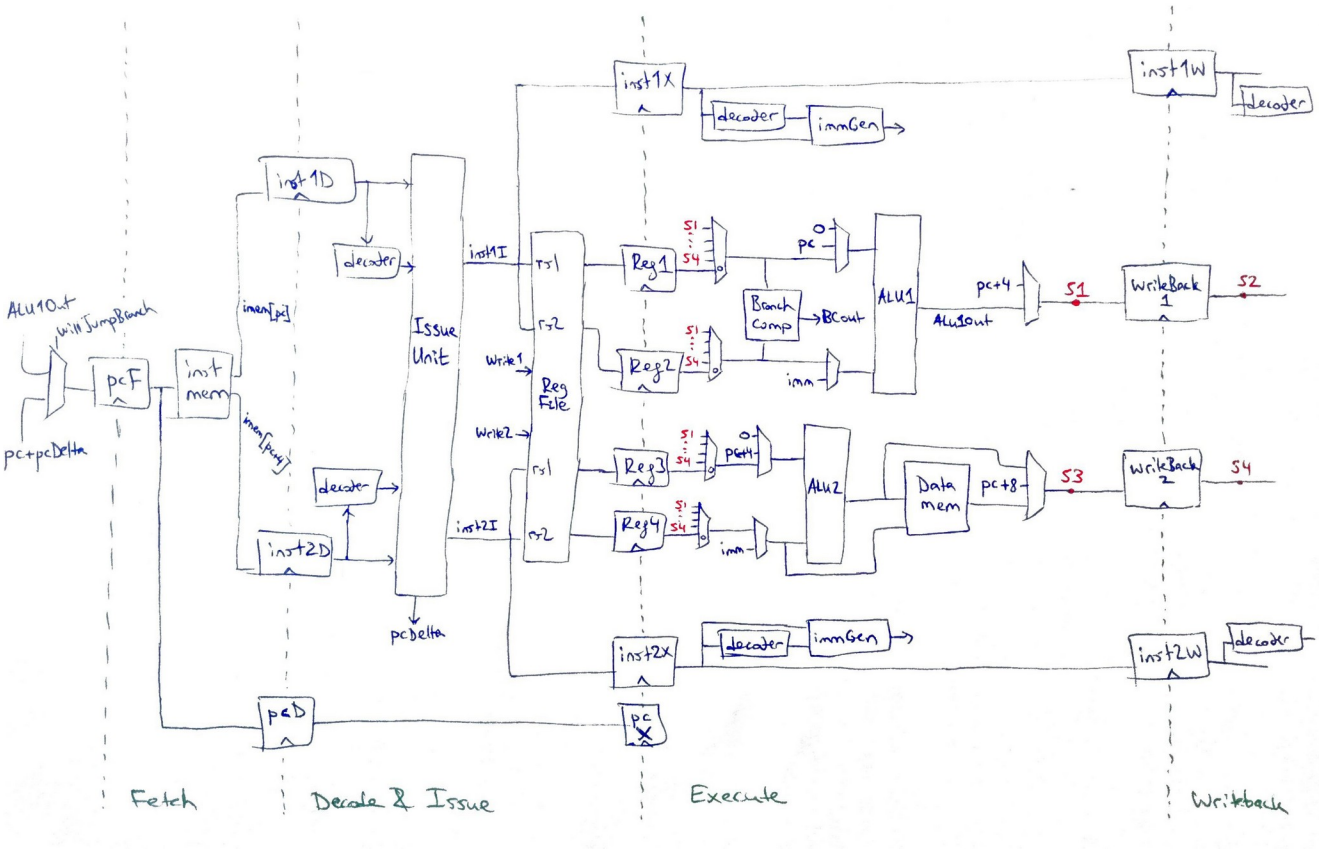
To create a new C or Asm test:

1. `cd` to the project's root directory.
2. Add your source code to the samples/ folder.
   Note: For C programs, the entry point is `void _start(void) {}`. Use `asm("ebreak")` to halt the simulation. For Asm programs, the entry point is `_start:`. Use `ebreak` to halt the simulation. The entry point doesn't have to be at the very top.
3. Run `make prepare_c SRC=samples/filename.c` or `make prepare_asm SRC=samples/filename.s` depending on the language. C++ can be added by adding an entry to the makefile.
4. Make sure the `imem` and optionally `dmem` files are created. Note down the initial PC address and minimum memory sizes.
5. Edit the file `src/riscv_superscalar.sv` and update the fields according to the values logged out.
6. Run `make cpu`.
7. Done. The results are in `logFile` and `riscv_superscalar.vcd`

Note that for the bubblesort test program INITIAL_PC=112 and for the other three INITIAL_PC=36.

5. Changes from the Pipelined Processor

- Regfile now has 4 input ports and 2 output ports.
- Instmem has 2 output ports and they give imem[pc], imem[pc+4].
- There are 2 paths now: (1) ALU + BranchComparator, (2) ALU + DataMem.
- The M stage is removed from the pipeline so that the two paths execute in 1 cycle as required.
- Forwarding logic is put in a seperate module: `src/ForwardingUnit.sv`
- Since the processor is dual issue, the issue logic got more complicated and is put in a seperate module: `src/IssueUnit.sv`



The schematic above illustrates the entire processor. Blue colored parts are paths an modules, red S1-S4 are points from where forwarding can be done. Control words, selects of multiplexers, forwarding paths, stalling and flushing mechanisms are not shown.

6. Performance

There are 4 test programs provided in the samples folder, all written in C. All of the test programs are tested and give correct results. Special assembly instruction combinations that would result in hazards are also tested and all give correct results. These are not provided in the samples folder. The results are as following. The same program is fed to singlecycle (of assignment 2) and pipelined processor (assignment 3). Assuming that the singlecycle processor has CPI=1, the CPI of the other ones is calculated by scaling the completion time.

Program	singlecycle cpu exec time	CPI	pipeline cpu exec time	CPI	superscalar cpu exec time	CPI
Bubblesort: 4 numbers	t=709	1	t=1057	1.49	t=907	1.28
Prime: 6th prime	t=3763	1	t=5645	1.5	t=5585	1.48
Findmax: 17 numbers	t=595	1	t=859	1.44	t=719	1.21
Strlen: 12 characters	t=243	1	t=377	1.55	t=273	1.12

The superscalar CPU has a high CPI due to the issue unit never reordering the instructions. Since the data memory is on path 2, the instruction that comes after it cannot be put on path 1, as that would count as reordering. Likewise, since there's a single branch comparator, and since it's on path 1, the instruction that comes before it has to be executed seperately. The issue unit places bubbles in such cases, which decreases the performance.
Nonetheless, a forwarding unit is implemented that resolves all RAW hazards. If the bottom instruction depends on the top one, value is forwarded from top to bottom. Likewise, if a value depends on the 2 instructions on the previous cycle, the value is forwarded from the W stage to X stage.

7. References

[1] https://github.com/k13-engineering/node-elf-file