Indian Institute of Technology, Roorkee
CSN-221 : Computer Architecture and Microprocessors
Course Project

Roopam Taneja

November 8, 2023

GitHub Link : https://github.com/RoopamTaneja/CSN-221-Comp-Arch-Processor

RISC-V Pipelined Processor Simulator

This project was made as part of course CSN-221 : Computer Architecture and Microprocessors in Autumn Semester 2023-24.

Contents of Processor Folder

- Three sample programs written in RISC-V assembly language :
  - `1prime.s` : Check if a number stored in memory is prime or not.
  - `2factorial.s` : Calculate factorial of a number stored in memory.
  - `3gcd_lcm.s` : Calculate GCD and LCM of two numbers stored in memory.

  The assembly programs are self-explanatory and should run fine on any RISC-V simulator. The user can modify *ecall* and other auxiliary statements as per need.
- An assembler written in C++, `assembler.cpp` which converts RISC-V assembly codes to binary and hex encodings.
- A non-pipelined simulator written in C++, `simulator.cpp` that executes instructions encoded in binary format and writes back into a .txt file it uses as memory.
- A simulator for a 5-stage pipeline with **interlocks** written in C++, `stall_pipeline.cpp` that executes instructions and also writes stage description for each cycle and also prints execution statistics like number of stalls and flushes.
- A simulator for a 5-stage pipeline with **operand forwarding** written in C++, `op_forward_stall_pipeline.cpp` that executes instructions and also writes stage description for each cycle and also prints execution statistics like number of forwards, stalls and flushes.

  NOTE: load-use hazards have also been taken care of.

  The file truth_table.xlsx tabulates the signals which make up the control word, their meaning and the few design choices made to simplify the design.

Assembler Description and Usage Guidelines

The assembler will take as arguments 1 .txt file containing your RISC-V assembly program and 2 empty .txt files, one for outputting binary encoding and other for hex encoding. The hex encoding file is only meant for readability and includes instruction starting addresses as well.

Commands:

After compiling `assembler.cpp`, run:

```
./<file_name>.exe <assembly_file>.txt <binary_file>.txt <hex_file>.txt
```

You can create your own .txt files or use the sample `asm.txt`, `bin.txt` and `hex.txt` present in the repository. `asm.txt` also contains three sample assembly programs ready to be fed to the assembler (ofcourse, use one at a time).

Guidelines:

1. Only integer instructions supported.
2. Supported instructions:
   - R-type: `add, sub, xor, or, and, sll, sra, rem, mul, div`
   - I-type: `addi, xori, ori, andi, slli, srai`
   - Load-Store: `lw, sw`
   - B-Type: `beq, blt`
   - Jump: `jal, jalr`
   - U-Type: `lui, auipc`
   - **Pseudo:** `li`
3. Use ABI names for register operands in assembly code.
4. Positive immediates can be written in either decimal or hexadecimal formats but only decimal format for negative immediates (to avoid confusion).
5. Types of statements allowed:
   - Labels - `<label_name><:><space><comments(optional)>`

     No instruction should come after label in the same line
   - Instructions - maybe followed by comments

     eg: `addi r1, r2, r3` ***(1 whitespace only, whitespace is important, commas not so much)***
   - Only comments

     Everything after # is treated as comment.
   - Blank lines
6. Labels are supported for B-type and J-type instructions (not U-type or JALR). Label name must NOT start with a digit. Provide either labels or numeric offset from PC.

NOTE: `hex_dict.h, nf7_type.h, opcode.h, registers.h, r_type.h` are header files used by `assembler.cpp` and all of them should be kept in the same folder.

Simulator Description and Usage Guidelines

The simulator will take as arguments one .txt file containing your binary encoding of instructions and another .txt acting as memory.

Commands:

After compiling `simulator.cpp`, run:

```
./<file_name>.exe <binary_file>.txt <memory_file>.txt
```

You can create your own .txt files or use the sample `bin.txt` and `data.txt` present in the repository.

Guidelines:

1. In your final binary file, don't have anything other than pure 0s and 1s (not even blank lines).
2. Data file format :

   0x0000: <data>

   0x0004: <data>...

   Address should from 0 and be in multiples of 4. The range of address is not restricted. You can use the `createEmptyDataFile.cpp` to make the memory file instantly. Compile and run it with the name of .txt file you are using as memory. Also don't have any other data or unnecessary blank lines.
3. The data should only be DECIMAL values.

Pipeline with Interlocks Simulator Description and Usage Guidelines

The simulator will take as arguments one .txt file containing your binary encoding of instructions, one .txt acting as memory and one .txt for outputting stage description of each cycle.

Commands:

After compiling `stall_pipeline.cpp`, run:

```
./<file_name>.exe <binary_file>.txt <memory_file>.txt <cycle_file>.txt
```

You can create your own .txt files or use the sample `bin.txt`, `data.txt` and `cycle.txt` present in the repository.

Please note:

- All ALU instructions take single cycle for execution stage in this simulator.
- All branches are assumed *not taken*. Pipeline is flushed incase of a branch penalty.

Guidelines are same as that for the simulator described earlier.

Pipeline with Operand Forwarding Simulator Description and Usage Guidelines

The simulator will take as arguments one .txt file containing your binary encoding of instructions, one .txt acting as memory and one .txt for outputting stage description of each cycle.

Commands:

After compiling `op_forward_stall_pipeline.cpp`, run:

```
./<file_name>.exe <binary_file>.txt <memory_file>.txt <cycle_file>.txt
```

You can create your own .txt files or use the sample `bin.txt`, `data.txt` and `cycle.txt` present in the repository.

Please note:

- Types of operand forwarding paths:
  1. EXMO to EX stage
  2. MOWB to EX stage
  3. MOWB to MO stage

     NOTE: Store instructions receive their second operand exclusively via MOWB to MO path. No other instruction can use that path.
- Load-use hazards have also been taken care of.

- All ALU instructions take single cycle for execution stage in this simulator. - All branches are assumed *not taken.* Pipeline is flushed incase of a branch penalty.

Guidelines are same as that for the simulator described earlier.

Comparative Study

Identifying if a given no is prime

Input : 2099 ; Output : 1 (True)

Pipeline Type	# Stalls (Data Hazards)	# Forwards (# Load-Use Hazards)	# Flushes (Branch Penalties)	Total Cycles	I	S	CPI (k=5)
Non-pipelined	NA	NA	NA	10494	10494	NA	1
With stalls	4203	NA	4195	23091	10494	12593	2.2004
With operand forwarding	NA	2102 (0)	4195	18888	10494	8390	1.79989

Finding factorial of a given no

Input : 12 ; Output : 479001600

Pipeline Type	# Stalls (Data Hazards)	# Forwards (# Load-Use Hazards)	# Flushes (Branch Penalties)	Total Cycles	I	S	CPI (k=5)
Non-pipelined	NA	NA	NA	55	55	NA	1
With stalls	6	NA	13	91	55	32	1.65455
With operand forwarding	NA	4 (0)	13	85	55	26	1.54545

Finding GCD and LCM of two given nos

Input : 9305 & 4300 ; Output : GCD = 5 & LCM = 8002300

Pipeline Type	# Stalls (Data Hazards)	# Forwards (# Load-Use Hazards)	# Flushes (Branch Penalties)	Total Cycles	I	S	CPI (k=5)
Non-pipelined	NA	NA	NA	38673	38673	NA	1
With stalls	38672	NA	4297	85943	38673	47266	2.2223
With operand forwarding	NA	21488 (2)	4297	47273	38673	8596	1.22238

References

The references can be found in the References folder of the repository.