EAST TENNESSEE STATE UNIVERSITY

CSCI-3160: Computer Systems

Hardware Project

Goal

You will write a simulator in C that accepts, as input, a stream of 32-bit RISC-V instructions, and outputs a description of when each of those instructions passes through different stages of an in-order processor's pipeline.

Specifications

Keep in mind: these specifications are the "bare minimum" I expect for this assignment, i.e., a complete and correct implementation of the specified pipeline simulation, are what I would expect a 'C'-level effort to look like. Students with more ambitious designs and/or extensions of this specification may be able to use that additional effort and work as evidence for a higher grade at the end of the course.

Instruction Set

We use a small subset of the canonical 32-bit RISC-V instruction set architecture for the baseline simulation:

- Data transfers: `lw, sw`
- Arithmetic: `add, addi, sub`
- Control: `<nop is a pseudo-instruction for addi x0, x0, 0>, bne, beq`
- Register file: `x0-x31`

Pipeline Design

At a minimum: your project will simulate the RISC-V instruction subset specified above in a five-stage pipeline:

1. Fetch instruction
2. Decode instruction / read register file
3. Execute instruction / calculate address
4. Access memory
5. Write result to register file

Assumptions and Latencies

Not all instructions take the same amount of time to execute. However, for the purposes of this (minimum) specification, I'm making some assumptions about latencies of these instructions. These assumptions may be relaxed/altered for your own simulation (especially for those of you pursuing an "above-and-beyond" implementation of this simulation).

- At most, one instruction can be issued (fetched) per cycle.
- At most, one data memory access (reading/writing memory) can occur per cycle.
  - Memory accesses take one cycle ("magic" L1 cache with 100% hit rate)
- Loads and stores require one cycle to calculate the effective address
- The register file may **not** accomplish a read **and** a write of the **same** register in the same cycle
  - For simplicity's sake (not particularly realistic)
- **Only one instruction may be executing (in the ALU) at a time:**
  - Integer arithmetic (including address calculations) complete in one cycle.

Since this base specification only simulates the pipeline (and not the actual instruction effects), I assume that addresses given in different registers (along with offsets) will not ever collide, e.g., adding an offset to register x3 will **not** lead to the same base address held in x2.

Finally: if I encounter any structural hazards, data hazards (due to a RAW, WAR, or WAW dependency), or branching hazards (due to not knowing if the branch will be taken or fallen through) in the processor pipeline, I stall (insert a bubble into the pipeline).

Input and Output

I will provide a basic starter code that handles basic input of a stream of instructions and outputs a reasonably-well formatted table of instructions, when those instructions are handled in different functional units, and basic statistics, as well as a Makefile for building this starter. As part of your deliverable, you will (likely) need to modify the output code (and potentially the input code for "above-and-beyond" simulations) to correctly implement your simulation, as well as update the Makefile for any additional files you may include.

Other Details

Anything not specified above is left to your discretion—however, be prepared to justify your decisions. See me if you want feedback on your ideas before implementing them.

Deliverables

You will submit your simulation source codes, your Makefile to build your simulation, and a README.txt file that outlines 1) how to run your simulation, 2) how many hours you spent on your simulation, and 3) what your simulation actually does. Submit these deliverables to the posted D2L Dropbox by the posted date and time.

Grading

This project will not receive an analytic mark-up: instead, this is an **effort-based** project that you will partially **self-evaluate** your own efforts and results on. That means I will provide holistic feedback for your submission (what I think—does this meet expectations, exceed expectations, any observations on your simulation and/or your deliverables) and you will incorporate that feedback, along with your own perspective, into self-evaluating how you did on this project and tying it back to the learning outcomes. You will this self-evaluation in your final grade persuasive essay (due at the end of the semester).

Extensions

This project has a lot of room for some interesting "above-and-beyond" extensions—primarily, forwarding via pipeline registers (instead of stalling) on data hazards, static branch prediction paired with flushing (instead of stalling) for control hazards, implementing control logic considerations (pairs well with forwarding), creating a listing of both the assembly and machine language, and building a GUI datapath for your pipeline to illustrate what's happening at each cycle.

These are just a handful of ideas you could implement. I am open to other project extensions (like, say, static multi-issue) or implementing a dynamically-scheduled pipeline (which opens the door to the really interesting instruction-level parallelism features and interesting floating-point instruction delays).

Example Inputs and Outputs

Config

Notes on this file: I use a configuration file format across several of my simulations—this is **not** a hard-and-fast format but merely an example.

For my buffers section: if I only have a total of one arithmetic buffer, I assume a "magic" ALU/FPU combo.

```
buffers

fetch: 1
memory: 1
ints: 1
fp adds: 0
fp muls: 0

latencies

memory: 3
ints: 1
fp_add: 2
fp_mul: 3
fp_div: 10
```

Trace.dat (Input instructions)

Notes on this file: I assume no data races in main memory on address resolution (i.e., no collisions).

```
flw f6,32(x2)
flw f2,48(x3)
fmul.s f0,f2,f4
fsub.s f8,f6,f2
fdiv.s f10,f0,f6
fadd.s f6,f8,f2
```

Output

```
Configuration
-------------
buffers:

fetch: 1
memory: 1
ints: 1
fp adds: 0
fp muls: 0

***NOTE*** combining ALU with FPU

Latencies:

memory: 3
ints: 1
fp_add: 2
fp_mul: 3
fp_div: 10
```

```
                   Static Pipeline Simulation
------------------------------------------------------------------

                    Inst.    Decode   Execute  Access   Write
     Instruction    Fetch    Read Reg Calc Adr Memory   Register
-------------------- -------- -------- -------- -------- --------
flw f6, 32(x2)             1        2        3    4 - 6        7
flw f2, 48(x3)             2        3    4 - 6    7 - 9       10
fmul.s f0, f2, f4          3   4 - 11  12 - 16                17
fsub.s f8, f6, f2     4 - 11  12 - 16  17 - 18                19
fdiv.s f10, f0, f6   12 - 16  17 - 18  19 - 28                29
fadd.s f6, f8, f2    17 - 18  19 - 28  29 - 30                31

Hazard Cycles
-------------
structural: 7
data: 2
control: 0

Dependencies
------------
read-after-write: 2
write-after-read: 0
write-after-write: 0

Cycles Stalled
--------------
instruction fetch: 12
inst. decode / read register: 21
execute / calculate address: 2
read / write memory: 0
write register: 0
```

Notes on this output: the hazards statistics is counting the number of times my simulation inserted bubbles into the pipeline—some of these overlap (like the decode and read register step for the floating-point multiply instruction—I had a structural hazard due to the stall in the execution stage **and** a data hazard due to the read-after-write dependency on `f2`).

The dependencies are only those that result in an actual hazard—there's 7 occurrences of a register being read after a write, but only 2 of those occurrences would overlap temporally to cause a stale read. In addition, there's 1 occurrence of a write (`f6`) after a read, but by the time the floating-point addition would write its result to `f6`, the reads of `f6` are complete (thus, no WAR dependency).