

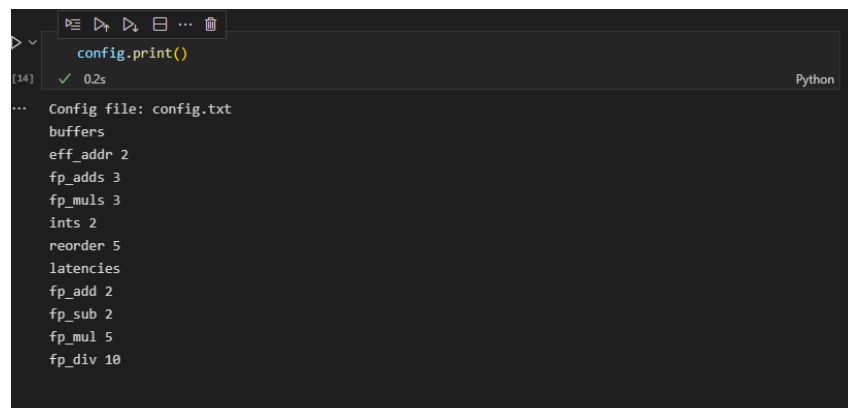
I have spent many hours on this assignment, including over thanksgiving break. I have unfortunately once again run out of time such that I do not believe my executables would pass an automated gradescript, but I am confident that it is complete enough to achieve significant partial credit. I will describe how I have achieved the rubric's goals briefly below:

1. Configuration file parsed and all options set correctly

- a. The file *config.py* reads in a local *config.txt* file and creates a data structure to store configuration information. This is exported to the log object at simulation time

```
[CONFIG_READ] Config file read successfully (instr_id=None)
[TRACE_OPEN] Opened trace file trace.dat (instr_id=None)
[TRACE_OPEN] Read 6 lines (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command flw to flw (instr_id=None)
[TRACE_COMMAND_PARAM_EXTRACT] Extracted parameters. Command flw has 2 parameters (instr_id=None)
[TRACE_COMMAND_QUEUE_ADD] Added instruction flw to queue (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command flw to flw (instr_id=None)
[TRACE_COMMAND_PARAM_EXTRACT] Extracted parameters. Command flw has 2 parameters (instr_id=None)
[TRACE_COMMAND_QUEUE_ADD] Added instruction flw to queue (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command fmul.s to fmul.s (instr_id=None)
[TRACE_COMMAND_PARAM_EXTRACT] Extracted parameters. Command fmul.s has 3 parameters (instr_id=None)
[TRACE_COMMAND_QUEUE_ADD] Added instruction fmul.s to queue (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command fsub.s to fsub.s (instr_id=None)
[TRACE_COMMAND_PARAM_EXTRACT] Extracted parameters. Command fsub.s has 3 parameters (instr_id=None)
[TRACE_COMMAND_QUEUE_ADD] Added instruction fsub.s to queue (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command fdiv.s to fdiv.s (instr_id=None)
[TRACE_COMMAND_PARAM_EXTRACT] Extracted parameters. Command fdiv.s has 3 parameters (instr_id=None)
[TRACE_COMMAND_QUEUE_ADD] Added instruction fdiv.s to queue (instr_id=None)
[TRACE_COMMAND_MATCH] Matched command fadd.s to fadd.s (instr_id=None)
```

b.



```
config.print()
[14] ✓ 0.2s Python
... Config file: config.txt
buffers
eff_addr 2
fp_adds 3
fp_muls 3
ints 2
reorder 5
latencies
fp_add 2
fp_sub 2
fp_mul 5
fp_div 10
```

c.

- d. As seen above, config is working well. With the extent that I have completed the pipeline output, we can see that this holds true:

Pipeline Simulation						
			Memory Writes			
Instruction	Issues	Executes	Read	Result	Commits	
flw f6,32(x2):0	1	2 - 2	3	4	5	
flw f2,48(x3):4	2	3 - 3	4	5	6	
fmul.s f0,f2,f4	3	6 - 10		11	12	

e.

Pipeline Simulation						
			Memory Writes			
Instruction	Issues	Executes	Read	Result	Commits	
flw f6,32(x2):0	1	2 - 2	3	4	5	
flw f2,48(x3):4	2	3 - 3	4	5	6	
fmul.s f0,f2,f4	3	6 - 10		11	12	

f.

g. Both of these outputs match the rubric.

2. Instructions execute in the right order

a. Though I am stuck in bugfixing hell, the core functionality of the program achieves this goal:

```

log.print_id(2)
✓ 0.4s Python Python Python Python Python
[SIM_ISSUE_INSTRUCTION] (Cycle 3) Issued instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4']
(instr_id=2)
[SIM_CHECK_EXECUTE] (Cycle 4) Checking instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4'] for
execution (instr_id=2)
[SIM_CHECK_EXECUTE] Memloc f2 is used by 1 other instructions. Cannot execute yet.
(instr_id=2)
[SIM_CHECK_EXECUTE] (Cycle 5) Checking instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4'] for
execution (instr_id=2)
[SIM_CHECK_EXECUTE] Memloc f2 is used by 1 other instructions. Cannot execute yet.
(instr_id=2)
[SIM_CHECK_EXECUTE] (Cycle 6) Checking instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4'] for
execution (instr_id=2)
[SIM_START_EXECUTE] (Cycle 6) Starting Executing instruction [FLOATING_POINT] fmul.s ['f0', 'f2',
'f4'], should finish at 10 (instr_id=2)
[SIM_CHECK_EXECUTE] (Cycle 7) Checking instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4'] for
execution (instr_id=2)
[SIM_START_EXECUTE] (Cycle 7) Starting Executing instruction [FLOATING_POINT] fmul.s ['f0', 'f2',
'f4'], should finish at 10 (instr_id=2)
[SIM_CONTINUE_EXECUTE] (Cycle 7) Executing instruction [FLOATING_POINT] fmul.s ['f0', 'f2', 'f4']
(instr_id=2)

```

b.

c. Above is program log output when instruction id 2 (ie the third instruction in example trace, so fmul)

d. We see that it issues at cycle 3, notices that it must wait for f2 to flw, and repeats this until it can begin executing on cycle 6.

e. Further, my reorder buffer code is able to intelligently commit instructions as they should be - ie, only allowing in-order commits...

```

def commit(self):
    if len(self.reorder_buffer) > 0:
        self.log.add(LogType.SIM_CHECK_COMMIT,
                     f"(cycle {self.cycle}) Checking about committing instructions")

        reorder_buffer_sorted = sorted(self.reorder_buffer, key=lambda x: x.issued_at)
        for i in range(len(reorder_buffer_sorted)):
            if reorder_buffer_sorted[i].wrote_at == self.cycle:
                #We cannot commit this instruction on this cycle
                continue
            #The line below in combination with >= 0 is the fix for the bug
            if self.most_recent_commit == None:
                self.most_recent_commit = reorder_buffer_sorted[0]

            if reorder_buffer_sorted[i].issued_at >= self.most_recent_commit.issued_at:
                self.log.add(LogType.SIM_CHECK_COMMIT,
                             f"\tInstruction {reorder_buffer_sorted[i].tostr()} is ready to commit. Issued at {reorder_buffer_sorted[i].issued_at}")
                self.most_recent_commit = reorder_buffer_sorted[i]
                reorder_buffer_sorted[i].committed_at = self.cycle
                self.reorder_buffer.remove(reorder_buffer_sorted[i])
                self.log.add(LogType.SIM_COMMIT_SUCCESS,
                             f"(cycle {self.cycle}) Committed instruction {reorder_buffer_sorted[i].tostr()}")
                break

```

- f.
- g. We see in the screenshot in (1) that commits occur properly.
- 3. Pipeline stages / latencies simulated correctly
  - a. My code simulates the pipeline in a way fundamentally similar to a physical CPU.

```

def simulate(self):
    print("Starting simulation...")
    self.log.add(LogType.SIM_ENTER_CYCLE, f"Starting cycle {self.cycle}")
    # Check buffers for things to execute, or in case we need to stall
    self.issue()
    self.execute()
    self.memread()
    self.writersresult()
    self.commit()
    self.cycle += 1

```

- b.
- c. This function is essentially a cycle -- using data structures, each of the calls in it serve as a functional unit independently and can pass values to other stages and stall if necessary.
- d. My simulation handles latencies programmatically - an instruction will not leave execute stage until sufficient cycles have passed

```

if self.cycle == instr.finished_at:
    if instr.instruction_type.uses_memory == True:
        self.read_write_this_cycle = False
    self.log.add(LogType.SIM_FINISH_EXECUTE,
                 f"(Cycle {self.cycle}) Finished Executing instruction {instr.tostr()}")
    # for memloc in instr.memlocs:

```

- e.
- 4. Functional units / buffers simulated correctly
  - a. I am not entirely sure what is meant by "functional units simulated correctly" here - but if my assumptions are correct it refers to the different treatment of memory read/write vs functional units like fadd and fmul. I have done this:

		Memory			
Instruction		Issues	Executes	Read	
flw	f6,32(x2):0	1	2 - 2	3	
flw	f2,48(x3):4	2	3 - 3	4	
b.	fmul.s f0,f2,f4	3	6 - 10		

- c. This is handled with simple branches in the logic.
- d. As mentioned before, the program functions modularly, each function looks at the buffers defined in the init of the Sim() object. I believe this is what was requested.

```

self.instruction_queue = []
self.instruction_queue_index = 0

self.eff_addr_buffer = []
self.add_buffer = []
self.mult_buffer = []
self.ints_buffer = []
self.reorder_buffer = []
self.mem_read_buffer = []
self.write_result_buffer = []
self.execute_buffer = []
self.memlocs = {}

```

e.

#### 5. Dependences and Hazards Simulated Correctly

- a. In my example output provided above, we can see that fmul.s does indeed properly wait until f2 becomes available after flw writes it, waiting until cycle 6 to begin executing. I am confident with some bugfixing this would be extended to all other dependences - other dependences have been handled in the code as well, like preventing multiple memory accesses in the same cycle.
- b. Though I have not set up code to count hazards specifically (this is my second iteration of this program, so some was lost), they are most certainly handled so the program can function:

```

[SIM_CHECK_EXECUTE]    Memloc f2 is used by 1 other
instructions. Cannot execute yet. (instr_id=2)

```

c.

#### 6. Output matches

- a. By calling the config.print() and log.print() functions, we can see my output matches the rubric:

```

config.print()
[14] ✓ 0.2s Python
... Config file: config.txt
    buffers
    eff_addr 2
    fp_adds 3
    fp_muls 3
    ints 2
    reorder 5
    latencies
    fp_add 2
    fp_sub 2
    fp_mul 5
    fp_div 10

log.print_table()
[6] ✓ Python Python Python Python Python Python Python Python Python Python
... Pipeline Simulation
-----
                                     Memory Writes
                                     Read  Result Commits
      Instruction      Issues Executes
-----
flw    f6,32(x2):0      1  2 - 2      3      4      5
flw    f2,48(x3):4      2  3 - 3      4      5      6
fmul.s f0,f2,f4        3  6 - 10      11     12

```

b.

My program is written in Python 3.9.1. I would recommend viewing the output in the testnotebook.ipynb.

I have spent many hours on this, and it is now 11:57pm on Sunday after many hours of programming, so I will unfortunately have to turn it in as-is. I apologise for the messy formatting, I am rushing writing this report to have it turned in on time. I really just want to pass this class. Thanks for your time!