ENEE 446 - Project 1 Report

By Calvin Leung

Contents

Implementation 2
Changes to the state struct: 2
Fetch 2
Decode 2
Decoding and instruction 2
Detecting DATA Hazard 2
Detecting RAW hazard 3
Detecting WAW hazard 4
Detecting Structural hazard 4
Detecting Control hazard 4
Execute 4
Advance functional units 4
Decrement stall counter 4
Halt activation 5
Structural lock 5
Writeback 5
Perform Write 5
Result and Benchmark 6
simple.s 6
newton.s 6
vect.s 6
cos.s 6
Observed Issues 7
Minor inaccuracy in cycle counting for more complex simulations. 7
Decrement R1 Issue 7
How I could improve the simulation 7

Implementation

Changes to the state struct:

I've added 6 variables to the state struct that are useful to determine different types of hazards:

control_stall, stalled_inst, stall_counter, structural_lock, halt_activated and end_simulation.

These variables are used as flags and conditions for the simulation to run.

Fetch

In the fetch function, it follows the sequence of checking whether the state has been fetch-locked by the preceding decode function. If fetch lock is not enabled, the function would fetch the instruction from memory and pass that instruction to the if_id struct. It would then increment the PC.

If a control stall has been triggered by the preceding decode function, the function would still fetch the instruction after the branch and pass it to the if_id struct. Additionally, the function would enable fetch lock for the next couple of cycles until the branch instruction has been resolved.

Decode

The decode function is the most elaborate function in my entire simulation, so I'll break it down to multiple sections.

Decoding and instruction

I used the function provided in fu.c "decode_instr" to get the information about the instruction in the if_id struct.

Detecting DATA Hazard

In order to check whether a data hazard exist, we must check whether the instruction is referring to an integer register or a floating-point register.

Since using the FIELD_RX(instr) function does not specify if it is a floating point register or an integer register. I created two function: source_extract(instr) and destination_extract(instr).

source_extract(instr):

This function returns the type of where source register the instruction has:

INT_SOURCE_ONE - integer source register at Field R1

INT_SOURCE_TWO - integer source register at Field R1 and Field R2

FP_SOURCE_ONE - floating point source register at Field R1

FP_SOURCE_ONE - floating point source register at Field R1 and Field R2

BOTH_SOURCE - integer source register at Field R1 and floating-point source register at Field 2 (for s.s instruction)

destination_extract(instr):

This function returns what type of destination register the instruction has.

It can be of the following types:

INT_DEST_AT_2 - integer destination register at Field R2.

INT_DEST_AT_3 - integer destination register at Field R3.

FP_DEST_AT_2 - floating point destination register at Field R2.

FP_DEST_AT_3 - floating point destination register at Field R3.

Detecting RAW hazard

To detect RAW hazard, I created 3 function to do the task: RAW_finder(), RAW_finder_loop(), RAW_dependancy().

RAW_finder():

The function is called in the decode function.

It loops through all the instructions in the integer functional units.

It calls the RAW_finder_loop() three times to loops through all the instructions in each (ADD, MULT, DIV) floating point functional units.

Since the pipeline is in "reverse order, the function also checks the instruction which has just been sent to the WB register.

RAW_finder_loop():

The function is called 3 times in the RAW_finder() for each floating point functional unit.

It is implemented to reduce code duplication.

It loops through all the instruction in the specified floating point functional units.

RAW_dependancy():

This function is called in each iteration of the loops described above.

It checks whether the source register(s) of the current instruction is the same as the destination register of the instructions that are in the functional units.

This is done by calling the source_extract(instr) on the current instruction and destination_extract(instr) on the instruction in the functional units.

This function returns TRUE for RAW detected, FALSE otherwise.

Detecting WAW hazard

To detect WAW hazard, I created 3 function to do the task: WAW_finder(), WAW_finder_loop(), WAW_dependancy().

The implementation is very similar to the one for WAW, but instead of checking the source destination of the current instruction, the function checks the destination address of bot the current instruction and the instructions in the functional units and write back struct.

After if data hazard exists, the decode function will not issue any instruction until the hazard is resolved

Detecting Structural hazard

If no data hazard is present, I will issue the instruction according to its group number. As I issue the instruction, if the issue_fu_int() or issue_fu_fp() returns -1, the decode function enable the structural_lock variable. As the structural_lock is enabled, fetch_lock is enabled in which the next instruction will not be fetched.

Detecting Control hazard

Control hazard is only applicable to the control instructions. So aside from checking the structural hazard of the instruction, control instructions will also enable the control stall.

When control stall is enabled, the decode function will not immediately enable fetch_lock for the fact that after we issue the instruction to the functional units, we need to fetch the next instruction into the if_id struct and determine whether to flush the instruction or not when the control instruction reaches write back. Therefore, as the control_stall in enabled, the fetch instruction will detect the flag and enable fetch_lock after fetching the next instruction.

Execute

The execute function in my simulation plays two major roles: advancing functional units and resolving hazard.

Advance functional units

The execute function will advance all functional unit from the get-go.

Decrement stall counter

If the following condition is met: fetch_lock enabled and none of the other flags are enabled.

This means that only data hazard is occurring, the execute would then check the stall counter. If the stall counter is greater than 1, this means that the processor must stall for a longer time. Otherwise, this means that the stall is over and fetch_lock can be disabled.

Halt activation

If fetch_lock and halt activation is both enabled, this means that a HALT instruction is detected. Only when all the functional units returns true with fu_int_done() or fu_fp_done() and all the write-back struct is empty, the end_simulation flag will be enabled. As end_simulation flag is enable, this for loop in main will end.

Structural lock

If fetch_lock and structural_lock is enabled, this means that a structural hazard is occurring, I will simple disable the fetch_lock and structural_lock. This is because if the structural hazard persist, the pipeline will enable these flags again in the next cycle.

Writeback

The writeback() function takes the instruction in the int_wb and fp_wb struct and pass them into the perform_write() function, which is my own version of perform_operation().

The writeback function also increment the executed instruction counter after completing its operation. Then the function would clear int_wb and fp_wb in order to complete the process.

Perform Write

My perform_write() function is a variation to the provided perform_operation(). The reason I created my own perform_write() function is because that the perform_operation() function does not have the processor state as a parameter. I think it is more intuitive for the function to directly change the state of the processor instead of using a convoluted way of using the operand_struct.

With that said, my perform_write() function is a massive nested switch statement that has the code that perform the operation of each instruction listed in Figure 1.

Result and Benchmark

simple.s

	Executed Instructions	Total Cycles	CPI
My simulation	6	15	2.5
Provided output	7	15	2.14

Simulating simple.s shows good results. The results in memory and register are consistent. There is strange occurrence that the number of executed instructions here is 1 less than the expected output while more complex programs are accurate in that regard.

newton.s

	Executed Instructions	Total Cycles	CPI
My simulation	51	172	3.37
Provided output	51	172	3.37

Simulating newton.s shows the most promising results. As the executed instructions and total cycles are the same as the provided output. The values in memory as well as the floating point seems to be the same (for the fact that the endianness is different)

vect.s

	Executed Instructions	Total Cycles	CPI
My simulation	123	287	2.33
Provided output	123	285	2.32

Simulating vect.s also yield shows good results. The number of executed instruction and total cycles are very close as well as the values in memory and floating-point registers are consistent to the provided output.

cos.s

	Executed Instructions	Total Cycles	CPI
My simulation	111	429	3.86
Provided output	111	430	3.87

Simulating cos.s shows the least promising results among all four. Even though the number of executed instructions and total cycles are very close, the values found in the registers are not accurate. I suspect that it is caused by some instructions getting executed in the incorrect order due to failure in stalling.

Observed Issues

Minor inaccuracy in cycle counting for more complex simulations.

Beside simulating newton.s, simulations for the other 3 programs have minor inaccuracy in terms of counting the total cycles or the number of executed instruction. For simple.s, I am still trying to figure out why the number of executed instructions is 1 less than expected. I suggest that there could be a bug somewhere that did not increment the counter correctly. Fortunately, the number of executed instructions for the other 3 simulations are spot on. In regard to the total cycles, only vect.s and cos.s have minor error. Though not significant, a margin of 2 cycles, I suspect that it has nothing to do with instructions in the loop since the error would have compounded if it is.

Decrement R1 Issue

In the simulations, R1 is used as a counter. By the end of the simulations, R1 seems to have the value of (-1), which made me realized that the R1 must have been decremented one more time unnecessarily, but since this didn't affect the loop, I do not think this is a major issue.

How I could improve the simulation

Besides on debugging the issues that I have listed above, I think that I could improve the simulation by implementing a better WAW detection. Since the number of stalled cycle is not as intuitive as the stalled cycle for RAW, there is a high chance that my WAW detection does not have an accurate number of stalled cycle.