Project Report ECE-563

1. Stage Integer Pipeline

Current Status:

Testcase1:Passed-->Older revision submitted
Testcase2---->Testcase6-->Code compiles but Segmentation Errors in the pipeline register

Overview:

The integer pipeline MIPS simulation is based on the 5 stage pipeline we mainly get to see in RISC-V microprocessors.Before I go on to the Stages of the pipeline Instructions please do note the Integer pipeline Simulation writes in the first half of the clock cycle and reads in the second half of clock cycle.

Implementation:

Data_Structures---> Special Purpose register::
The Special Purpose register s Implemented as Structures or can be thought of as classes as I have implemented them in c++.The 5 Special Purpose Registers are:

1. IF_ID_PIPELINE_COLUMN
   1.1 Sp_registers NPC
   1.2 Instruction register IR

2. ID_EX_PIPELINE_COLUMN
   2.1 Sp_registers NPC
   2.2 Sp_Registers A
   2.3 Sp_registers B
   2.4 Sp_registers IMM(sign extended)
   2.5 Instruction_register IR

3. EXE_MEM_PIPELINE_COLUMN
   3.1 Sp_registers COND
   3.2 Sp_registers ALU_OUTPUT
   3.3 Sp_registers B
   3.4 Instruction_register IR
4. MEM_WB COLUMN
   4.1 Sp-resisters LMD
   4.2 Sp_registers ALU_OUTPUT
   4.3 Instruction_register IR

General purpose registers are implemented as simple 1D array as integer_registers of size 32.

data memory is an pointer to array which is initialized to 10 int the beginning

Sim_pipe Behaviour @Func_

Instruction_Fetch()-->The main job of this function is to fetch the Instruction from Memory and pass it ot the ID_EX IR and also update the PC-Adder.

Instruction_Decode()--->Decodes the instruction from the GP registers and gets the special purpose register A,B and IMM ready for execute stage. Also checks for Control hazards and Data Hazards.

Instruction Execute()-Executes the instruction according to the op-codes provided using the helper function alu().It transfers the IR to the Memory stage and also updates the special purpose register B.

Instruction_memory()-->The memory stage of the Pipeline register is mainly used to update the LMD special purpose register for load operations .The alu output is written to its special purpose register.

Instruction_Writeback()-->The Writeback() stage of the Pipeline register is mainly used to write to the Integer register.

Hazard ::------> Data Hazard was the easy to understand and solve but i got confused on where to put the flag I removed the flags of the data hazards as it was giving me segmentation faults in all the testcases but I did cross check the implementation is correct:

@Detect_Data_Hazard():: It matches the source and the destination register of the ID_EX pipeline column and MEM_STAGE _PIPELINE COLUMN and checks whether the register matches or not.
if It does it makes pipe_requires_stalling==True and data_hazard=true and return true.The challenge here I was debugging this for quite a along time and later realized t he Store instruction was handled differently and so its hazard must also be detected differently and handled differently however while trying it to detect it at the decode stage it was giving segmentation error.I downgraded my revision so Detect_Data_Hazard() is still not implemented in the decode stage of the program .

Control_hazard Func()::

@Detect_Control_Hazards()---->It is only called when there is a branch operation and control_hazards becomes true;

Structural_hazard::-->

Bool Structural_hazard-As my data hazard and Control hazard testcases were not passing and giving segmentation faults the structural hazard was not implemented properly in the memory stage.But the concept is, if we get a structural hazard in the memory stage than we just increase the count_stalls and prevent the IR from passing down from decode stage unless memory stage has finished

@Handle_hazards()-->.The reason fro the segmentation faults is here it returns bool and it gets a feedback from Detect_Data_Hazards() and Detect_Control_ Hazards() and converts the op-codes to NOP meaning do nothing and send the IR one level up.(The logic is somewhat failing here)

@Detect_Branch()--->The function detects whether there is BQ* or JUMPS opcode and reruns True.

@run-Implements a while loop which runs for required no of clock cycles and an run-cycle which runs the program to completion.WB Stage is called first and the Instruction fetch is called at the end is called at the end.

Helper Functions::
1.@ Write_memory was used to write to register
2.@char2int was used to read from register
3.@get_IPC was used to get the Instruction_per_Clock cycle
4.@get_clock cycle returns my clock cycle counter
5.@get_stalls() returns my stall counter
6.@reset::every sp_register ,data_memory and gp_registers to Undefined and the initial state
7.@get_sp_registers->gets each sp registers from stages by calling each sp register in the struct using switch_cases
8.@create_Integer_Point_registers()-Own implementation of the helper function where each integer point register is filled with Undefined value and is called in the set_gp_registers
9.@get_gp_registers->return the integer point registers
10.@set_gp_registers-If integer registers are not empty call 8 else return integer registers of size 32

2. Floating point Pipeline::
Current_status-implementation still under progress
Testcases-0-5
Status-failed

As my Integer pipeline was giving me a lot of faults at every debugging steps I didn't have the time to look forward to floating point Pipeline implementation .However the basic structure of the pipeline remains the same Integer Point pipeline,which was included, only the changes are implementing floating point registers and introducing latency for Division and Mult operations.So all the data structures till now that is implemented is exactly the same as integer point pipeline

3. **Running the codes;**;I have not made any changes to the makefile and I tested oout in local machine on CLion and on the Linux command line the code was compiling

For safety do an rm -rf of object files with make clean:::
1.cd ${Project root}
2.make clean
3.make --all or make individual test-cases
4../bin/testcase1(or each individual test-cases on may want to run)

Conclusion::
The project actually made me go deeper with 5 stage pipe-lining.Debugging ,segmentation faults and memory leaks gave me nightmares.I kept on revisiting the concepts when pipeline operation of data hazard failed which turned to be wrong memory allocation and also the other way around when the pipeline output was not matching I was checking for wrong code implementation which later turned out to be wrong understanding of the core concepts.
The problem with debugging is most of the time I was getting confused which instruction set was present in which IR and at which cycle, later even after making multiple handwritten charts I was not able to track it properly ,especially at the run_cycle stage I placed checks after checks before test case 1 passed.Also in the debugging section the as the number of

variables increased,my debugging watch increased to 60 variables at a time which was a massive task and sometimes I was not able to even detect a very minor increment bug for hours,which ate up a lot of my time and productivity in this project.

*@SideNote::*I will continue revisiting this project even after submission.This project can be made easier to debug and track if we attach GUI to it.After completing the floating point implementations(which I know I will),I may spend time converting this whole code to Java and using J-Swing library to attach a GUI to to each instruction register and to each stage and track each Instructions visually as they move from one IR to another IR through every stage until their life-cycle has finished ,(or I may remain in C++ and implement the GUI with Qt.).Attaching a GUI to this project will make things much easier to track visually and to understand the core concepts much more clearly from a higher as well as lower level.