Lab Project: A Pipelined VHDL Calculator

This Lab is optional and is due 11:59PM May 20th. Your lab report and source code must be submitted by the deadline. If you choose to do this optional project, your final course grade will be computed as "your normal grade (all labs, homework and exams) + 33%*(grade of this lab project)".

This lab is to be done in teams. The required format for lab reports can be found on the resource page.

Objective: The objective of this laboratory is to implement an 8-bit pipelined calculator on VHDL simulator.

Specification of the calculator: The pipelined calculator is based on the calculator you build for the previous lab. The main difference is the handling of the structural hazard on the two registers and the data hazard.

All the instructions should be divided into 3 stages, ID, EXE and WB. Each stage uses one cycle. Table 1 describes what each instruction does at every stage. **Your implementation must show that the three instructions are completed in three steps. In particular, it is unacceptable to finish all work in one cycle and then just wait for the displaying of result.**

Instruction	ID	EXE	WB
Load instruction	Read/decode the immediate operands.	Do nothing	Store the result to registers. Display the loaded value to the display.
Add/Sub	Read two operands	Calculate operand1 +/-operand0	Store the result to registers. Display the result to the display.
Branch	Read two operands	Calculate the branch condition	Display the condition value to the display, for example, displaying "B0" and "B1".
Display	Read operand(s)	Do nothing	Display the value.
Nop	Do nothing	Do nothing	Display "NOP" on the display.

All numbers are displayed **in decimal format** as in the previous lab, even though instructions and data are read as binary numbers.

You should adapt your implementation of the adder/subtractor from the previous lab in this work. You can make necessary and reasonable changes to your code, but the base line is that you cannot directly use the “+” and “-“ operators in VHDL to do the adding or the subtracting. The subtractor might produce negative results, which should be shown on the Seven Segment Display as such.

Your main design task is to handle the data hazards between instructions, for example if the instruction “load 4 r0” is immediately followed by “add r0 r1 r2”, since r0 is only available after the WB stage of the load instruction, you should insert NOP(s) before the add instruction to make sure the “add” reads the right r0 value.

Task 1: Multiplex the registers in one cycle

In the previous lab, both the reading and writing of registers happen on the rising edge of cycle. Your first task is to change that into writing on rising edges and reading on dropping edges.

Choose either Task 2 or Task 3. You don’t need to do both.

Task 2: Insertion of bubbles into pipeline

A calculator instruction might be data dependent on its preceding instruction. For example, if “load 4 r0” is immediately followed by “add r0 r1 r2”, the second instruction should wait one cycle because r0 is not ready yet, i.e., the correct values being written in the next cycle. One way to handle it is to insert a bubble. At the beginning of the ID stage, decide whether to proceed with the current instruction, or to hold off the firing of the current instruction and replace it with a NOP. Your implementation should be based on the product from Task 1, so that at most one bubble needs to be inserted. (Think if we don’t have Task 1, how many bubbles we might need to insert?) The main implementation challenge here is to figure out the logic for when to insert bubble.

Task 3: Data forwarding

Another way to handle the data dependency is to introducing a controlled data forwarding path from the ALU output to the ALU input, i.e., the end of the EXE stage to the beginning of the EXE stage. In our 3-stage calculator, we just need either bubble insertion or the data forwarding to handle data dependency (Why?), not both. So your implementation will be based the product of Task 1. The main implementation challenges are how many data forwarding paths we need (Is it one or two?), how they merge with existing wires, and when to activate the data forwarding paths.

Expected output:

The expected result is that at the end every cycle, except the first two, we expect a new result appearing on the display. You should design testing sequences that cover different scenarios, for example, needing bubble insertion/forwarding or not.

With bubble insertion:

# of cycle	Instruction	The display
0	Load 4 r0	* * * *
1	Add r0 r1 r2	* * * *
2	* * * *	“4”
3	* * * *	“N O P” (bubble inserted)
4	* * * *	The sum of “4+r1”

With data forwarding

# of cycle	Instruction	The display
0	Load 4 r0	* * *
1	Add r0 r1 r2	* * *
2	* * * *	“4”
3	* * * *	The sum of “4+r1”

What to Turn In

For this lab project, turn in all of your source code and the code that tests your implementations. For each task, all the source files and the project configuration files must be packed as a single zip file. The goal is that I can unpack your zip file and directly compile your project. In your project report, please also describe your testing methodology.

Peer evaluation: Please email your peer evaluation (guideline is on the resource page) of your teammates, including scores and justification, directly to the TA.