COMP 4300 Computer Architecture

Project 4: A Pipeline with a Scoreboard

Points Possible: 100

- Turned in via Canvas

No collaboration among groups. Students in one group should NOT share any project code with another group. Collaborations in any form among groups will be treated as a serious violation of the University's academic integrity code.

Requirements:

(1) Each group should **independently** accomplish this project assignment. You are allowed to discuss any design issue with your friends in any group to solve the coding problems.

(2) You must submit your partially assembled code, the source code of your simulator, a README file, and the values of total clock cycles, total instructions executed, and number of NOPs.

(3) Your simulated machine must have an initial fetch / issue stage - you can decide on the size of the fetch buffer.

(4) The issue stage dispatches an instruction to one of four functional units:

  a) a two-stage pipelined integer ALU,
  b) a two-stage pipelined floating point adder (the adder can also execute floating point subtraction),
  c) a six-stage pipelined floating point multiplier, and
  d) a memory (load-store) "functional unit" that implements a two cycle delay.

(5) All of this is to be controlled by a scoreboard.

(6) Your README file must contain compilation instructions, and instructions on how to use your program.

(7) You must test (i.e., compile and run) your simulator on a Linux machine. Specific compilers you have to use to test and compile your source code are given below:

  a) g++ for C++
  b) gcc for C

1. Introduction

In this project, we will be making the simulated machine slightly more complex, and add scoreboard-based control.

You will evaluate both the case where the scoreboard reserves an entire functional unit for each incoming instruction, as well as the case where the scoreboard reserves just the first stage of a functional unit. You can satisfy this requirement by using two separate programs, or you can write one program that takes a parameter that causes it to switch modes. In the case where the scoreboard reserves only the first stage in a pipeline, instructions already in execution in the pipeline will continue to proceed, even if the instruction at the entry to the pipeline stalls waiting for an operand. However, instructions must leave a functional unit in the same order in which they enter it - if an instruction at the exit of a functional unit stalls, all upstream instructions must stall.

You are not required to implement branch delay slots, but you will note that load and store delays are implicit in the description of the machine above. You must perform each part as described, compare the performance of the two flavours of scoreboarding on the DTMF loop (see below), submit your programs.

Please refer to Section 1.1 of Project 1's description for instructions on how to make your code readable. Please refer to Section 1.2 of Project 1's description for some general tips for writing good code.

2. A Diverse Machine

It is assumed that you have built a simulated five-stage MIPS machine. This part of the lab is similar as the previous one. Rather than managing a single pipeline, you will be running four independent functional units.

First, think about the tasks of fetch and issue: fetch is pretty simple (for the MIPS ISA). Issue occurs at the final position in the fetch buffer, and departure of an instruction from the issue stage is controlled by the scoreboard.

One strategy for implementing the simulator in steps would be to first create a machine with only fetch/issue and a single functional unit, and then add the remaining functional units after that simpler machine has been tested. To create the first machine, you could split your five-stage pipeline from the previous lab into two parts: the front-end containing fetch, and the (integer) functional unit containing decode and execute. You could then add and test the scoreboard-based control. As long as there are no loads or stores in the instruction stream, you don't need MEM, and WB becomes a function of the scoreboard.

Whatever the implementation approach you take, be sure to read the textbook, which gives a detailed description of the actions in each step of an instruction. Please note that issue sends the instruction to the functional unit before the instruction obtains its operands. The functional unit reads the operands for the incoming instruction when the scoreboard tells the functional unit the operands are ready. Thus, the "operand fetch" part of decode has been moved to the functional unit, under the scoreboard control. Once the operands are available, the instruction can proceed. On completion (when the instruction reaches the final stage of the functional unit), the scoreboard checks to see whether there is a WAR hazard relative to an earlier instruction (in issue order).

For the scoreboard, you should be using a data structure similar to that shown in A-8 of the textbook. There is a paragraph that appears along with the title for the figure giving the detailed rules for how the individual fields are maintained. For a more dynamic example, refer to the scoreboard example in the **sco_tom.PDF** file.

You could think of the scoreboard-based machine as something like a Tomasulo architecture where there is only a single reservation station in front of each functional unit. However, it differs in that all results are written to registers, all operands are read from registers, and there is no tagged common data bus.

To run the example code we will work with, you will need **S.D** (store double, to store a floating point value), **L.D** (load double), **FADD** (floating point add),**FSUB** (floating point subtract) and **FMUL** (floating point multiply) in addition to the integer instructions you have implemented in the previous lab assignments. You will also need a floating-point register file (with registers **$f0** through **$f15**) in addition to the integer register file.

3. Forwarding

The only forwarding in this machine is through the register file.

4. Branches and Loads

4.1 Suggestions

Once you add the load/store unit, there will be a two-cycle delay between the time a load is issued, and the time its result is written into the register file. This load/store functional unit handles both integer and floating point loads and stores. The scoreboard will cause any instruction subsequent to a load that needs the result from the load to stall at the entry to its functional unit, waiting for its operand. This can be straightforwardly implemented when you correctly implement the scoreboard. You can implement it in the following way: when the load issues, the register result status section of the scoreboard will be updated to indicate that the load/store (you could call it memory) unit will be writing to a particular register. Subsequent instructions that refer to that register will stall until their operand arrives from memory.

We will still leave branches alone. Similar to the previous lab assignments, we are going to use **NOP**s to help make everything work out. However, it is a little bit different this time. First, the integer pipeline that is used to resolve the branch condition has a latency of two cycles. Second, if you have a deep fetch buffer, you may need to flush it if the branch is taken. So you will need at least two **NOP**s following each branch. If the branch is taken, it should write the **PC** when it exits the integer unit, and the fetch buffer should be flushed. You need to make sure you do all this before the third instruction following the branch is actually issued (this instruction should be entering the issue stage as the branch is exiting the integer unit) .

4.3 Detailed Syntax

```
ADD Rdest, Rsrc1, Rsrc2
ADDI  Rdest, Rsrc1, Imm
B  label
BEQZ  Rsrc1, label
BGE  Rsrc1, Rsrc2, label
BNE  Rsrc1, Rsrc2, label
LA  Rdest, label
LB  Rdest, offset(Rsrc1)
LI  Rdest, Imm
NOP
SUBI  Rdest, Rsrc1, Imm
SYSCALL

FADD Fdest, Fsrc1, Fsrc2
FMUL Fdest, Fsrc1, Fsrc2
FSUB Fdest, Fsrc1, Fsrc2
L.D  Fdest, offset(Rsrc1)
S.D  Fsource, offset(Rsrc1)
```

- **Rsrc1**, **Rsrc2**, and **Rdest** specify one of the general-purpose registers (**$0** through **$31**).
- **Imm** denotes a signed immediate (an integer).
- **label** denotes the address associated with a label in the .text or .data segment. This is encoded as a signed offset relative to the current value of the program counter, which will be one word past the address of the current instruction (i.e. the instruction referencing the label).
- **offset** denotes a signed offset (an immediate in the instruction) which is to be added to the value in the base register, **Rbase**. The result is the memory address from which a value is loaded into **Rdest**, or to which the value in **Rsrc1** is stored.
- **Fsrc1**, **Fsrc2**, and **Fdest** specify one of the floating-point registers (**$f0** through **$f15**).
- **FSUB** does **(Fsrc1** - **Fsrc2)**

5. A Test Case

(1) Please instrument your simulator to keep track of number of clock cycles of execution, total instruction count, and number of **NOPs** executed.

You must use a very simple test case (see **lab4a.s**) to see whether your new machine is working. As for the previous lab assignments, you will need to hand-assemble it for execution.

(2) You should be able to take a simple example program (see **lab4b.s**) and run it. You can invent other variations of this code to test things more thoroughly.

(3) Create the loop for DTMF tones (see **DTMF_loop.pdf**) - this will give us something more realistic (or at least better motivated) than the toy examples in described in the textbook. **Just to be explicit**: the calculations in the loop must be done using floating point, and the "while button pressed" condition should be replaced with a control condition (e.g. a "for") so that your code executes 100 iterations of the loop.

One simulation-related detail: make sure your simulator runs long enough to push the last instruction in the program all the way to the end of its functional unit. The simplest solution, of course, is just to add a bunch of **NOPs** to the end of the code (and that is acceptable), although there are certainly simple ways to do this more elegantly.

Note that your simulator must output total clock cycles, total instructions executed, and number of **NOPs** at the end of the run.

6. Deliverables

Please submit your program through Canvas (no e-mail submission is accepted).

6.1 A Single Compressed File

You must submit a single compressed file (e.g., file_name.tar.gz); the file name should be formatted as:

`“<group ID>_project4.tar.gz”`

Your compressed tarball (e.g., group6_project4.tar.gz) should contain the four (4) items below:

(1) **lab4c.s** - It contains your partially assembled code for the DTMF loop.
(2) **scoSim** - It contains your simulator. **Note:** If you created two separate programs please call the one that reserves the entire functional unit **sco_fullSim** and the one that reserves just the first stage **sco_pipeSim.**
(3) **README** - It should contain compilation instructions, and instructions on how to use your program. In this README file, please indicate your name and your student ID. In addition, a discussion of any design issues you ran into while implementing this project and a description of how the program works (or any parts that don’t work) is also appropriate content for the README file.

(4) Collaboration document: You must briefly describe how the two members in your group collaborate on this project. You also need to summarize the contributions of each member.

Note: If you have used something different than the five stage MIPS pipeline, you will need to provide sufficient hardcopy documentation so that the TA can understand the details of the machine.

6.2 How to name and create your compressed file?
Create a single compressed (.tar.gz) file named "<group ID>_project4.tar.gz". To create a compressed tar.gz file from multiple files or/and folders, we need to run the tar command as follows.

```
tar vfcz <group ID>_project4.tgz <folder_4300_project4_folder>
```

where `<folder_4300_p4_folder>` is a folder that contains the three (3) items described in Section 6.1. `<group ID>_project4.tgz` is the single compressed file to be submitted via Canvas. For example, my single compressed file to be submitted can be created using the following command:

```
tar vfcz <group ID>_project4.tgz ~/comp4300/project4
```

where `./comp4300/project4` is a folder that contains the above seven items.

6.3 Notes:
1) Other format (e.g., pdf, doc, txt) will not be accepted.
2) No e-mail submission is accepted
3) You will **lose points** (at least 5 points and up to 10 points) if you do not submit a single compressed file and name your compressed file in the format described in this Section.
4) You **lose 15 points** if your README file does not contain values of total clock cycles, total instructions executed, and number of **NOPs** at the end of the run (see Section 5).
5) You will **lose 5 points** if you do not use the specific file names.
6) You will **lose 5 points** if you do not indicate your name and your student ID in your README file.
7) **You will lose points if your source code can not be compiled by g++, or gcc on a Linux machine**.

7. Grading Criteria

1) Partially assembled code for the loop example lab4c.s: 10%
2) A simulator scoSim: 50% (**You will lose points if your source code can not be compiled by g++, or gcc on the Linux machine**)
3) Adhering to coding style: 10%

4) Values of total clock cycles, total instructions executed, and number of **NOPs** at the end of the run (see Section 5): 15%.
5) README file: 7.5%
6) Collaboration document: 7.5%

8. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

9. Rebuttal period

- You will be given a period of one week (i.e., 7 days) to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.