15-347, Spring 1998
Lab 3:
Handling Pipeline Hazards

Assigned: Thu., Apr. 2
Due: Thu., Apr. 16

1. Policy

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

2. Logistics

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page.

For this assignment, you will want to retrieve the file:

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/labs/lab3/files.tar
```

You will hand in a hard copy document for this assignment. Formatted text is preferred to hand written.

You should also provide us with a version of your code file `stages.c`. Do this by naming your file *andrewID*`-stages.c`, where *andrewID* is the andrew ID of one of your group members, and copying this file to the directory

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/labs/lab3/handin
```

Your group should do this before the date and time the assignment is due. You have only write and list permission for this directory. Be careful not to overwrite anyone else's solution.

Your implementation of `stages.c` is expected to compile and link with the testing program. Therefore, be careful not to modify other simulator files in such a way that your `stages.c` file depends upon your modifications. If your group's file does not compile, 20 points will be deducted from your group's score.

As in lab2, you should fill in the structure at the beginning of `stages.c` with the names of your group members.

3. Introduction

The purpose of this assignment is to gain a deeper understanding of how pipelined processors are implemented. Our method of doing this will be to create a C program that "simulates" pAlpha, a pipelined implementation of a subset of the Alpha architecture. Although C is not an ideal language for describing and simulating hardware designs, one can get a high level view of how the hardware operates by using an appropriate coding style.

You can pick up the entire set of files by copying and untarring the file `files.tar`. The source files include the following:

`alpha.h` : Macros defining the pAlpha instruction format. These are generally consistent with the Alpha architecture.

`sim.h`, `sim.c` Simulator framework.

`stages.h`, `stages.c` A partial implementation of the pipeline stages. You will need to fix and extend the code in the file `stages.c`.

In addition, there are some files containing utility routines for the simulator, user interface code, plus a subdirectory called `demos` containing some sample machine programs.

The code can be compiled to generate two different user interfaces:

`alpha_tty` A batch-oriented interface that prints trace information as it executes.

`alpha_tk` A graphic-user interface based on Tcl/Tk that lets you watch and control the simulator execution.

The GUI version is far more pleasant to use. The batch interface is provided as the fall-back on systems that do not support Tcl/Tk. Also, the batch version works better for doing systematic testing of your solution.

You may want modify the file `alpha_tk` to include pathnames for the directory in which you install your code.

A correctly working version of the simulator that runs on the class alphas has been installed as:

`/afs/cs.cmu.edu/academic/class/15347-s98/public/sim/solve_tk`

You might find it useful as an aid in debugging your own code.

Refer to the handout for the Lab3 prologue for instructions on how to operate the simulator.

4. pAlpha Instruction Set

Refer to page 35 of the class handout from lectures 11 and 12 for a description of the Alpha instruction formats.

The pAlpha instruction set includes only 7 classes of instructions:

Arithmetic Operations These are `ADDQ`, `SUBQ`, `ADDL`, `SUBL`, `CMPULT`, `CMPLT`, `BIS`, `ORNOT`, `XOR`, `CMOVEQ`, and `CMOVNE`. These instructions can be of either register-register or register-immediate format.

Load The `LDQ` instruction.

Store The `STQ` instruction.

Conditional Branches These are `BEQ` and `BNE`.

Unconditional Branches These are `BR` and `BSR`.

Jump The `JMP` instruction.

Call PAL This instruction is used to halt the simulator.

The intention of the implementation is to use the semantics of the Alpha instruction set. This includes requiring memory references to be aligned, etc. When you complete this lab assignment, all instructions will be fully implemented.

5. pAlpha Implementation

The version of the pipeline provided to you has serious deficiencies. In particular, stalling and forwarding are not used to handle hazards. Therefore both data and control hazards are handled incorrectly. Furthermore, none of the conditional move instructions are fully implemented. Otherwise, the design should be correct. **Please notify us if you find any bugs in the code**.

The handling of 64 bit integers within the simulator may seem odd. The reason is that the simulator is designed to operate on multiple platforms, some of which may not support 64-bit values. Therefore 64-bit integers are represented in the simulator by a structure consisting of two 32-bit values:

```
typedef struct {
  bit32 high;
  bit32 low;
} bit64;
```

Although you will be doing this lab on the class Alphas, you should continue to follow the convention of using the bit64 structure when making modifications.

6. Your Task

For this assignment you have four tasks:

1. Complete implementation of the two conditional move instructions.
2. Implement the proper handling of hazards in *stall mode*.
3. Implement the proper handling of hazards in *forward mode*.
4. Generate test code that will test all instruction types and all hazard possibilities.

All of the "hardware" modifications involve changing only the code in the file `stages.c`. You may change your own versions of the other files, e.g., to print out stuff while debugging, but your simulation should run properly with the original versions, since `stages.c` is the only file you will be allowed to hand in.

6.1. Conditional Move Instructions

Complete the implementation of these two instructions in a manner consistent with the rest of the design. Note that the simulation of the ALU operation in file `sim.c` is insufficient to correctly implement conditional moves for all cases. In the file `stages.c`, you should add additional logic to ensure that these instructions have the correct behavior in all cases. Make sure that you place the additional logic at the location indicated in the comments.

6.2. Hazard Handling in Stall Mode

This mode should be followed when the global variable `sim_mode` is set to `S_STALL`. It should use stalling to correctly handle both data and control hazards. It should cancel instructions only when stalling is insufficient to handle a particular hazard. Excessive use of cancelling in this mode will result in loss of points. Also, performance problems caused by unnecessary stalling will be penalized.

You can implement this version by inserting appropriate code into the procedure `do_stall_check`. At the end of this procedure there is some code that will do the actual stalling for you: you only need to tell it *when* to stall. You can stall in the IF, ID or EX pipeline stages by setting the appropriate variable to 1. The three variables involved are `stall_if`, `stall_id` and `stall_ex`.

When stalling is insufficient to handle a hazard, cancel instructions by calling the procedure `sim_cancel_stage`.

All the information you need to do a successful job is provided to you: you can get the current pipe-register state by means of the global variables `id_in`, `ex_in` etc., and the computed next state is available via arguments to the procedure. Both current and next state will be valid, since all of the stages have already computed their outputs by the time `do_stall_check` is called.

To guide your implementation, we have indicated where your code should be located with comment dividers, named `START STALL MODE` and `END STALL MODE`. **All of your stall-mode code should be located between those two dividers. In addition, you are neither allowed to create any static variables nor to change/create any global variables.** There is no need to create or modify new global state to come up with a correct solution.

6.3. Hazard Handling in Forwarding Mode

For forwarding mode, you will need to use the following mechanisms:

- Use forwarding (i.e., bypassing) whenever possible to handle data hazards without incurring any delays.
- Add stalls to handle any data hazards for which forwarding alone does not suffice.
- Handle branches by assuming they will not be taken, and canceling any instructions that should not have been fetched in the event that the branch is taken. The way to do this is to call the procedure `sim_cancel_stage` from within `do_stall_check`.
- You may handle jumps either in the same way as branches (always mispredicting!) or by stalling.

This mode should be used when the global variable `sim_mode` is set to `S_FORWARD`. You can implement this version by inserting appropriate code into the procedures `do_ex_stage`, `do_mem_stage`, and `do_stall_check`. In the latter procedure, the code will be similar to that of the stalling mode, although you'll need to stall in fewer cases. For the code within the EX and MEM pipeline stages you will implement three "multiplexors" ("muxes"). The A and B muxes in the EX stage can override the `Adata` and `Bdata` operands coming from the ID/EX pipeline register. The S mux in the MEM stage can override the `Adata` operand coming from the EX/MEM pipeline register. Your code should implement the control for these multiplexors by setting the variables `amux`, `bmux` and `smux` appropriately.

To guide your implementation, we have indicated where your code should be located with comment dividers, named `START FORWARD MODE` and `END FORWARD MODE`. **All of your forward-mode code should be located between those dividers. In addition, you are neither allowed to create any static variables nor to change/create any global variables, except for amux, bmux and smux.** Again, there is no need to create new global state to come up with a correct solution.

6.4. Testing

The test cases in the demos subdirectory are a good start, but they are not comprehensive. You should carefully analyze all of the different instruction types, all of their possible pipeline interactions, and generate

test code that will exercise these interactions. Note that you need not consider every possible instruction for all possible data values. Instead you can group instructions into classes and try them for representative data values.

Writing lots of test code by hand and then running them on the GUI interface is not a good idea—it is very time consuming and hard to get reliable results. Instead, you should set up a systematic "test bench," consisting of a semi-automatic way to generate, simulate, and evaluate test cases. Using a scripting language such as Perl or Tcl can be very useful for this task. Refer to the lecture discussion on micro-tests as one possible approach.

For this problem you should hand in the following:

- A description of your testing methodology for conditional move instructions. You should describe the various outcomes of executing each conditional move, as well as how you test that the expected outcomes are achieved.
- The results of running your code against these tests.
- A description of your testing methodology for data hazards. You should document your testing methodology by creating table(s) of all the different hazard types, as well as an indication of how you test that this hazard is handled properly. Look at the lecture slides for the sort of table we expect to see here.
- The results of running your code against these tests.
- A description of your testing methodology for control hazards. You should try to devise a systematic way to tabulate possible control hazards, e.g., instructions that should or should not be executed within some distance of a branch or jump.
- The results of running your code against these tests.

Remember that you must also handle hazards induced by a load instruction followed by an instruction requiring the result of the load. Be especially careful to test hazards involving register 31.

6.5. Additional Guidelines

- It is important that you do not change any global state other than the mux signals, nor create any new global state.
- For the hazard handling, please make sure that *all* of your code is located between the appropriate comment dividers.
- Although it may be tempting, do not rearrange what gets computed in what stage, or add any additional pipeline state.
- Remember that each stage must obey the protocol of taking pipe register current state and computing pipe register next state that will feed into the next stage. With forwarding, you will find that the EX and MEM stages access the current state of multiple pipe registers.
- The stall logic is implemented by a separate procedure `do_stall_check` that is called between the operate phase of the stages and the update phase of the pipe registers. This procedure can therefore make use of both the current and next state of all the pipe registers, but its only effect should be to cancel or stall some of the pipeline stages.
- Your implementation will require writing roughly 100 lines of code.

7. Hand In

Electronic submission:

1. Your version of `stages.c`. Remember to fill in the structure at the beginning of the file with the names of your group members. Also remember to comment your solutions.

Hard-copy submission:

1. A description of how you implement the conditional move instructions.
2. A description of your stall-mode hazard handling mechanism. Document the conditions under which you stall stages. Document the conditions under which you cancel instructions. Don't just show the code!
3. A description of your forward-mode hazard handling mechanisms. Document the conditions under which you stall or cancel stages and forward data.
4. Documentation of your testing methodology (see above).