Pipeline Simulator Exercise #1
SimpleScalar Exercise #2

Due Wednesday, September 11, 2002, 4:30 p.m.

This exercise is meant to help you better understand how pipelines function and to further familiarize you with the SimpleScalar 3.0 toolset. It is the first of two assignments, in which you will build a simulator for a complete, single-issue, in-order processor. In this first phase, you will construct the pipeline stages and hazard detection and interlocking mechanisms.

Assumptions

For the purposes of this exercise, we are assuming the following:

- Perfect instruction cache (no I-cache misses)
- Perfect data cache (no D-cache misses)
- No branch prediction
- No result forwarding
- Split-phase register access for WB stage (writes occur in first half of clock cycle, reads in second half)

This means that for the moment, we will not add cache or branch predictor functionality to the simulator (that is left for phase two). Therefore, for now, no stalling will be required in the fetch stage (IF) for instruction cache misses or in the memory stage (MEM) for data cache misses. When branches occur, stalling will be required until the branch condition and target are resolved, which in our pipeline will be the execute stage (EX).

Pipeline Model

The pipeline we will be modeling is otherwise the 5-stage pipeline described in Hennessy and Patterson, Appendix A (*e.g.*, Fig. A.2), with the main difference being branch resolution in EX rather than ID.

The following is a brief description of the pipeline stage functionality:

- **IF** – instruction fetch from memory based on PC+4 or PC provided by branch resolution.
- **ID** – instruction decoding, hazard detection, and register fetch. Note that in phase one, we are not modeling result forwarding.
- **EX** –ALU and other computation operations, memory effective address calculation for memory instructions, branch resolution.
- **MEM** – perform loads/stores to perfect D-cache using address calculated in EX.

- **WB** – writeback of results to register file and instruction retirement.

NOTES

1. For the **IF** stage we are assuming a perfect instruction cache (icache) which will always have the instruction we are looking for. This assumption is not true in a real processor: if the I-cache doesn't contain the required instructions we may have to go out to memory to fetch the instruction, requiring us to suffer a cache miss penalty. However assuming a perfect I-cache simplifies the assignment significantly. Later we will add caches and account for miss penalties. (The same applies to the data cache which would be accessed during the MEM stage.)

2. **ID**. Decode takes the instruction passed from fetch and decodes it, determining what type of operation it is. Because of the design of SimpleScalar, we will actually completely execute the instruction from the standpoint of *functional* simulation in the decode stage. The EX stage merely exists for accurate modeling of timing. We must still write appropriate information into the latches so that we can model behavior and timing.

3. `wb_finished_s:` An extra pipeline latch is included in the code you will receive. It is not part of the behavioral model, but is merely there to keep information about what actions WB has just completed.

Simulator Skeleton Code

0. Download the code distribution `assign2.tar.gz` from:
   `~cs654/fall2001/assign2`

1. The pipeline simulator you will build is based on sim-safe.c. We have provided you with a slightly modified version of the *sim-safe.c*, now called *sim-pipe.c*, which contains a few hints on how to proceed with the assignment.

2. The basic thing to note is that there are 5 functions corresponding to the 5 stages of the pipeline, and that several structs have been provided to serve as the inter-stage latches (or pipeline registers in Hennessey and Patterson). The main simulator loop simulates as one cycle: **WB, MEM, EX, ID, IF**. (Traversing the stages in backwards order simplifies the instruction flow through the pipeline.)

3. The following is the code for the stage latch:

```
/* naming convention follows H&P latch name convention */
struct stage_latch {
        int busy;          /* latch stage is busy */
        md_inst_t IR;      /* instruction bits */
        md_addr_t PC;      /* PC */
        md_addr_t NPC;     /* the new PC */
        md_addr_t addr;    /* mem address to read or write */
        int out1;          /* output 1 register number */
        int out2;          /* output 2 register number */
```

```
        int in1;          /* input 1 register number */
        int in2;          /* input 2 register number */
        int in3;          /* input 3 register number */
        int ls_size;      /* size of read or write */
        enum md_opcode op;/* decoded op code */
        int will_exit;    /* will this inst force the pgm to
                             exit */
} if_id_s, id_ex_s, ex_mem_s, mem_wb_s, wb_finished_s;
```

This is the information we feel might be necessary for the pipeline we are simulating to have available. You may augment this struct if you feel you need additional information in any of the stages.

`out1-2`, and `in1-3` are provided for hazard detection purposes. `machine.def` names the inputs and outputs for each instruction. The `DEFINST` macro included in `sim-pipe.c` will allow you to gather the necessary input and output register information needed for hazard detection.

For the moment `ls_size` is not needed, but will become necessary when we add caches. This indicates the size of the load or store. For the moment, it can be ignored.

`will_exit` is provided as a measure to prevent cycle miscounts due to the fact that we will actually be executing instructions in the ID stage. `will_exit` is basically a variable that will prevent the exit system call in the program (signalling the end of the program) from being executed until the WB stage. This is a violation of the behavior of our pipeline, but if we allow it to execute in either ID (for SimpleScalar) or EX (for a real pipeline), the program will terminate without allowing the exit instruction to reach the WB stage when it will truly have been "completed." This has an effect on the total cycle count in that if we don't wait until WB to execute the exit, we will have under-counted the total number of cycles to complete the program.

4. FYI, the ISA definition in machine.def is slightly different than shown in class. In the newest version of SimpleScalar, portability considerations have caused the instruction-implementation specification to be moved into a #define preceding the other aspects of instruction specification (`DEFINST`).

Sample Test Code and Sample Output

1. **Assembly Code Programs**. Three small sample assembly code programs: `raw.S, branch.S, and branch2.S` have been provided as sample tests for you to use during your simulator development. To compile these, simply use ~skadron/SimpleScalar/sun/bin/ssbig-na-trix-gcc, with the `–nostdlib` flag. This prevents the C standard library from being compiled into your code, thus limiting your instruction count to the number of instructions in your assembly code file (makes it easier to assess whether your cycle count is correct). An example:

```
~skadron/SimpleScalar/sun/bin/ssbig-na-trix-gcc -o raw
raw.S -nostdlib -O0
```

This takes `raw.S`, compiles it with no optimizations and names the binary `raw`. Feel free to modify these test cases to test other types of hazards and other scenarios, as we will be testing more situations than those given in the sample files. NOTE: if you want to comment your assembly code with c-style comments, your assembly code file needs to end with `.S` as opposed to `.s`

2. **Sample output**. Reference output is provided for `branch.S, branch2.S, raw.S,` and also for `test-fmath` in `tests/bin.big`. They are named `branch.output, branch2.output, raw.output`. For the sake of ease of reading, since `test-fmath` is a relatively long program, the simulator statistics and the assembly code trace are separated into `test-fmath.stats` and `test-fmath.output`. Also, in order to save some space, `test-fmath.output` only contains information printed from the decode stage.

   The reference simulator was run with the `-v` flag set. This provides you with a code "trace", which will allow you to track whether your simulator is doing what our solution simulator does. (You will need to add the code to print statements similar to those shown in the output). General form of the output file is as follows (a slightly modified version of what verbose prints in `sim-safe.c`):

```
Stage    Cycle #  Inst, #                          Address       Assembly Code
fetch:       1        0 [xor: 0x7fff8008] @ 0x004000f0: addiu     r4,r0,0
decod:       2        1 [xor: 0x7fff8008] @ 0x004000f0: addiu     r4,r0,0
```

   Stages which are missing from the output during certain cycles indicates that the stage is not currently doing work during that cycle. I.e., it is stalling. After the trace, the simulation statistics follow, including the number of instructions executed and total number of cycles used during execution.

   (NOTE: for the tests in tests/bin.big, you may not get exactly the same results as those distributed, this is due to differences in execution runs, and in environment variable settings)

Some Modifications Before Starting

1. `Makefile`. You will need to modify `Makefile` to compile your new simulator. Basically, this involves adding the name of your simulator (sim-pipe.c) to the list for the following variables: SRCS, PROGS, adding a line that tells the makefile how to compile sim-pipe, and also a few lines to tell the makefile sim-pipe's dependences. Follow the pattern for sim-safe.

2. `loader.c`. This file loads programs. There is a slight bug where the loader attempts to read the segment even if it is empty (i.e. size==0). Therefore, on lines 504 and 554 of `loader.c`, please alter the line which reads:

```
if (fread(p, shdr.s_size, 1, fobj) < 1)
```

to the following

```
if(shdr.s_size>0 && (fread(p, shdr.s_size, 1, fobj) < 1))
```

This basically short circuits the read attempt if the segment header is size 0.

The Assignment

1. Included in sim-pipe.c are empty functions for each pipeline stage and a prototype for the pipeline registers (a struct called `stage_latch` in the code). You are to write in the functionality of each pipeline stage. I.e., for instruction_fetch(), you will write code that checks if the `if_id_s latch` is ready for you to start writing information to it. If it is, then write the appropriate code to do the instruction fetch. The same process applies for the remaining pipeline stage functions. The functionality for each pipeline stage is briefly described above.

2. Your simulator should do the following:

   - execute program instructions
   - detect data and control hazards
   - stall as appropriate (stall on control and data hazards)

Homework Submission

(NOTE: We will be compiling your simulators in our SimpleScalar installation and running our test cases on them, so make sure that your README file gives clear directions on how to compile your simulator and details any additional files we will need.)

In an email to cs654@cs.virginia.edu,include the following information:

1) Group member names and email ids (no email aliases please).
2) README: This should detail how to compile your simulator and the names of files you have modified.
3) Pointer to the location of the files you have modified to write your simulator. Place all the files that you modified for this assignment, along with the README file in a directory like this: `/home/<your userid>/cs654/assign01`

Approach Suggestions

1. Build in phases and test often. Debugging a lot of changes is hard, debugging a small change is somewhat easier.

2. Work on getting instructions flowing through the pipeline smoothly before worrying about getting hazards detected. When you get that working as you'd like, work on detecting data hazards, then control hazards.

Now you're set to go. Good luck!