Pipeline Simulator Exercise #1
SimpleScalar Exercise #2

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

Purpose: Understand and implement a 5 stage pipeline, including hazard detection, stalls, branch/store/load instruction handling

Simulator Skeleton Code

1. Download the code distribution `assign2.tar.gz` from: `~cs654/fall2003/assign2`
2. In assign2.tar.gz, there is a file called sim-pipe.c. This file is modified from sim-safe.c under the directory SimpleScalar3.0. This assignment will need you to add necessary code into the sim-pipe.c.
3. The basic thing to note is that there are 5 functions corresponding to the 5 stages of the pipeline. The simulator simulates a loop as one iteration of **WB, MEM, EX, ID, IF**. (Traversing the stages in backwards order simplifies the instruction flow through the pipeline.) We have implemented most of IF, and you will do the remaining stages.
4. While you read the following you may wish to refer to sim-pipe.c.

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 data 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, there is no cache or branch predictor functionality in the simulator (that is left for next assignment). This means there is no stalling in the fetch stage (IF) due to instruction cache misses or in the memory stage (MEM) due to data cache misses. Since no branch condition and target can only be resolved after the execute stage (EX), branches will induce stalls.

Pipeline Model

The pipeline we will be modeling is similar to the 5-stage pipeline described in Hennessy and Patterson, Appendix A (*e.g.*, Fig. A.2). But our pipeline resolves branches in the EX stage, while H&P (Fig A.2) resolve the branch at ID stage.

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

- **IF** – fetch instruction from memory based on PC+4 for non-branch instruction, or based on the PC provided by branch resolution for branch instruction No cache misses: perfect cache
- **ID** – decode instruct, detect data/control hazard, read register for operands. Note in this assignment, no data forwarding
- **EX** –execute computation operations (ALU/others), calculate effective memory address for load/store instructions, resolve branches
- **MEM** – load/store data from/to memory address calculated in EX stage. No cache misses: perfect cache
- **WB** – write data back to register file and finish the instruction.

Important notes

1. In the **IF** stage, in this assignment we are assuming a perfect instruction cache (icache) which will always have the instruction we are looking for. We make this assumption only because we want to simplify this assignment. This assumption is not true in a real processor. In a real processor, if the I-cache doesn't contain the required instructions we may have to fetch the instruction from lower level memory, which will introduce a cache miss penalty. (The same assumption applies to the data cache at the MEM stage.)

   The skeleton code that we have given you implements most of this stage, to give you an example of the kind of modifications you need to be making for the other stages. This is only one way to implement the IF stage – feel free to remove our implementation and replace it with yours.

2. In the **ID** stage, the instruction passed from IF stage, will be decoded to determine what type of instruction it is, to detect data/control hazard and to determine any need for stall cycles. But in SimpleScalar, there is some extra work simulated in the ID stage to simplify the simulation.. This extra work is the odd-looking switch statement in sim-pipe.c in the ID stage that looks like the following:

```
/* execute the instruction */
   switch (op)
   {
        #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)\
              case OP:                                         \
                     SYMCAT(OP,_IMPL);                         \
              break;
         #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)          \
              case OP:                                         \
              panic("attempted to execute a linking opcode");
         #define CONNECT(OP)
         #define DECLARE_FAULT(FAULT)                        \
              { fault = (FAULT); break; }
         #include "machine.def"
              default:
                     panic("attempted to execute a bogus opcode");
   }
```

This code segment implements what is referred to as *functional simulation* or *trace generation*. It emulates the function of the instructions in terms of emulating the actual computation performed by each instruction, producing the correct result values, and so forth. This is necessary to determine what instruction sequence will actually be executed, and so functional simulation is a prerequisite for *timing simulation*—modeling the cycle-accurate flow of instructions through the pipeline. In this assignment, we only care about timing simulation. We are not interested in the details of extracting the necessary bits from the appropriate registers, performing the actual computation on the operands, etc. For this reason, the EX stage merely exists for accurate modeling of timing/cycles. The functional simulation clutters up our timing simulator by appearing in the ID stage; it might be more convenient to have this separated out, but the current arrangement simplifies various aspects of the overall simulator.

3. In sim-pipe.c, there is an extra latch called `wb_finished_s`. It is not part of the behavioral model or in the real pipeline, but in the simulator we need it to keep information about what actions WB has just completed.

Important Data Structure/Variables/functions

1. **Data Structure 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 an important data structure to pass information from stage to stage, for example, if you want to pass PC information from ID stage to EX stage, you can add code like: id_ex_s.PC = if_id_s.PC.

`out1-2`, and `in1-3` are provided for hazard detection purposes. The file called `machine.def` under SimpleScalar3.0 names the inputs and outputs registers for each instruction. The `DEFINST` macro, at ID stage inside of the switch(op) function included in `sim-pipe.c`, will allow you to gather the necessary input and output register information needed for hazard detection.

For example, if you want to detect if the new fetched instruction will use the same register as previous instruction at EXE stage for RAW hazard, you can do something like this:

```
if((ex_mem_s.busy == 1 && (
        ((if_id_s.in1==ex_mem_s.out1) && if_id_s.in1!=0)||
        ((if_id_s.in2==ex_mem_s.out1) && if_id_s.in2!=0) ||
        ((if_id_s.in3==ex_mem_s.out1) && if_id_s.in3!=0) ||
        ((if_id_s.in1==ex_mem_s.out2) && if_id_s.in1!=0) ||
        ((if_id_s.in2==ex_mem_s.out2) && if_id_s.in2!=0) ||
        ((if_id_s.in3==ex_mem_s.out2) && if_id_s.in3!=0)))
```

The variable `ls_size` is not needed in this assignment, but will become necessary when we add caches. This indicates the size of the load or store. For this assignment, 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.

2. Memory/ Register Data Structure

All variables/structures suffixed with string "mem" are defined in Memory.c/h files under SimpleScalar3.0, such as structure `MEM_READ_QWORD`.

All variables/structures suffixed with string "md" are defined in Machine.c/h files under SimpleScalar3.0, such as structure `md_inst_t`.
Some other files you may want to have a look are: misc.c/h, options.c/h, syscall.c/h.

3. Important variables/functions

Statistic information variables, such as sim_num_refs, sim_num_insn, sim_cycle, max_insts, sim_elapsed_time, etc. And some statistic functions, such as stat_reg_counter, stat_reg_int, stat_reg_formula, etc. All these variables/function definitions can be found in file stats.c/h under SimpleScalar3.0

Work you need to do

1. ***Add proper code into the five stage functions in sim-pipe.c.*** The five functions are called as following for WB, MEM, EXE, ID, IF stages:

```
static void writeback_stage(void){

};
static void memory_stage(void) {

};
static void execute_stage(void) {

};
static void instruction_decode_stage(void) {

};
static void instruction_fetch_stage(void) {

};
```

Currently, the functions are doing nothing, except for IF. You need to make each function correctly model the timing of instruction flow through the pipeline. For example, for WB stage, it should pass the instruction info from to the `mem_wb_s` latch to the `wb_finished_s` latch, and finish the instruction. And if the instruction is an exit system call, it should exit now. So the sample WB stage function would look like:

```
      if(mem_wb_s.busy == 1)
      {
            /* if we have an exit syscall, do it now */
            if(mem_wb_s.will_exit)
            {
                  sys_syscall(&regs,   mem_access,   mem,   mem_wb_s.IR,
TRUE);
            }

            /* copy instruction info to the
            latch/ pipeline registers which follow */
            wb_finished_s = mem_wb_s;

            /* set appropriate stage latch flags */
            mem_wb_s.busy = 0;
      }
```

In order to have statistics about performance, etc., we want to count various events and print out the statistics. So the final WB stage looks like:

```
       if(mem_wb_s.busy == 1)
      {
            /* if we have an exit syscall, do it now */
            if(mem_wb_s.will_exit)
            {
                  sys_syscall(&regs,   mem_access,   mem,   mem_wb_s.IR,
TRUE);
            }

            /* copy instruction info to the
            latch/ pipeline registers which follow */
            wb_finished_s = mem_wb_s;

            /* set appropriate stage latch flags */
            mem_wb_s.busy = 0;

            if(verbose)
            {
                  myfprintf(stderr, "wb   :"
                  "%10n %10n [xor: 0x%08x] @ 0x%08p: ",
                  sim_cycle, sim_num_insn,
                  md_xor_regs(&regs), mem_wb_s.PC);
                  md_print_insn(mem_wb_s.IR, mem_wb_s.PC, stderr);
                  myfprintf(stderr, "\n");
            }

      }
```

For the rest of four stages, you need to think carefully what the function this stage needs to achieve, then add the appropriate code into each function.

2. Initialize the simulator

In sim-pipe.c, there is a function called sim_init as following:

```
void sim_init(void)
{
      sim_num_refs = 0;
      /* allocate and initialize register file */
      regs_init(&regs);
      /* allocate and initialize memory space */
      mem = mem_create("mem");
      mem_init(mem);

      /* initialize stage latches */
      /* IF/ID */

      /* ID/EX */

      /* EX/MEM */

      /* MEM/WB */

      /* WB_FINISHED */

}
```

You need to add code to initialize the stage latches. You can use function `void init_latch(struct stage_latch * the_latch)` to do this. This function also in sim-pipe.c

3. ***Modify Makefile to enable compile of your sim-pipe.c***

   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. Easiest way to do it is to follow the pattern for sim-safe.

4. ***Modify file 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.

5. ***Your simulator should do the following:***

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

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-sstrix-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-sstrix-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)

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!