Lab 0: SimpleScalar Simulator

1 Purpose and Overview

The purpose of Lab 0 is to familiarize students with the SimpleScalar simulation suite. SimpleScalar is one of the simulation tools you will use throughout the semester. Other tools will be introduced in the lab as appropriate. Students will learn how to compile the simulator, simulate a precompiled benchmark, compile a sample test code, and modify the simulator to extract simple statistics from a dynamic instruction stream.

Although students do not need to hand in this pre-assignment, everyone is expected to go through this document and run through all the examples. You will receive 1 lab mark for attending the first lab, finding a lab partner and being assigned a time-slot. You are encouraged to stay in lab to work through this handout and have your questions answered by the TA. Understanding the walk-through example in Section 6 will be an excellent practice for Lab 1. Investing a little bit of time now will save time when doing the marked assignments.

2 Reading

Please read the document called *The SimpleScalar Toolset, Version 2.0* which is available in PDF format in the Labs section on Blackboard. Although the document describes version 2.0, we will be using Version 3.0 (revision d) of the SimpleScalar toolset. You may skip Section 2 of the document regarding installing and running the simulator, as we have already installed SimpleScalar on the ug machines (e.g., ug150.eecg) and we will cover running SimpleScalar in this lab[1]. Also, note that the version 2.0 document refers to the file `ss.def` which has been renamed to `machine.def` in version 3.0.

You should have also read the lab policy handout available on Blackboard.

3 Introduction

There are three basic components in an architectural simulation suite: 1) the simulator, 2) the compiler, and 3) the executable or source of instructions.

1. Our architectural simulator is SimpleScalar and a gzipped tar file that contains the SimpleScalar simulator is located at `/cad2/ece552f/simplesim-3.0d-ece552f-assign1.tgz`. The simulator is configured to simulate the PISA architecture. The PISA architecture and instruction set were created by SimpleScalar designers to be similar to MIPS.
2. Our compiler, provided with SimpleScalar, is a version of `gcc` that has been modified to generate program files using the PISA instruction set.The compiler is located at `/cad2/ece552f/compiler/bin/ssbig-na-sstrix-gcc`.
3. As a source of instructions, we can use any C code that can be compiled by the gcc compiler. We have some pre-compiled binaries of popular benchmark programs located in the directory `/cad2/ece552f/benchmarks`. The binaries have the ending .pisa-big to indicate they were

[1] You are free to do development on your own computer but you must ensure that your code compiles and runs correctly on the ug machines as these will be used for marking. Furthermore, the TAs are unable to provide support for issues related to environment and setup on non-ug machines.

compiled for the big-endian format of the PISA architecture. The same directory also contains EIO traces.

4 SimpleScalar Suite

As the document *The SimpleScalar Toolset, Version 2.0* describes, there are a several different versions of the SimpleScalar simulator, each with a different purpose. All the SimpleScalar simulators are *execution-driven* in that they actually simulate the execution of each instruction. This is in contrast to *trace-driven* simulators that use a trace of a previous execution to retrace the path followed by the program.

SimpleScalar includes both *functional* simulators and a *performance* simulator. A functional simulator runs a program just like a microprocessor supporting the same instruction set would, by taking program inputs and converting them to program outputs. However, because it does not simulate each individual processor cycle, we cannot precisely predict the speed of the processor. Functional simulators are useful when developing a new instruction set architecture as they are fast. Also, we can use functional simulators to learn about various instruction streams. For example, we may like to find out how often branch instructions occur, or how often dependencies exist between instructions. In addition to being a useful tool for computer architects, the speed of functional simulators allows compiler writers and application developers to test their work without actually first building a microprocessor.

A performance (or timing) simulator measures the performance of a microprocessor design by keeping track of individual clock cycles. Thus we can use performance simulation to find instructions per cycle (IPC), or its inverse (CPI). The drawback of maintaining such detailed timing information is much slower execution time compared to a functional simulator. In the SimpleScalar suite, the fastest functional simulator can simulate instructions 25 times faster than the performance simulator.

We usually prefer to use a functional simulator to make a measurement or perform an experiment. Sometimes, we can use a clever method or accept some inaccuracy in our measurements to avoid the use of a performance simulator while still making useful measurements. We try to leave the performance simulator as a last resort, since simulation time is long. Of course, in some cases, we have no choice but to use a performance simulator. Choosing between a functional and performance simulator and instrumenting them to extract results is part of the art of architectural simulation and design.

In this course we will primarily modify functional simulators to perform experiments and extract measurements.

The basic simulators included with SimpleScalar are:

1. `sim-safe`: A functional simulator with safety checks and a simple built-in debugger.
   We will be using sim-safe in `Labs 1 and 3`.

2. `sim-fast`: The same as `sim-safe` but with no safety checks and no debugger, intended to be as fast as possible.

3. `sim-cache`: A functional simulator used for simulating the effects of the cache configuration.
   We will be using sim-cache in `Lab 5`.

4. `sim-profile`: A functional simulator that generates profiling statistics about the instruction stream.

5. `sim-outorder`: A performance simulator that simulates an out-of-order architecture.

5 SimpleScalar Basics

In this section, students will learn how to compile the SimpleScalar simulator, run a pre-compiled benchmark and an EIO trace, and compile a sample microbenchmark program.

5.1 Compiling Simplescalar

To compile SimpleScalar execute the following commands in your *ug* machine account.

```
cd ~

mkdir ece552

chmod go-rwx ece552

cd ece552

mkdir preassignment

cd preassignment

cp /cad2/ece552f/simplesim-3.0d-ece552f-assign1.tgz .

tar -zxf simplesim-3.0d-ece552f-assign1.tgz

cd simplesim-3.0d-ece552f-assign1

make sim-safe

ls -l sim-safe
```

Assuming the above completed with no errors, you should now have the `sim-safe` executable in the current directory. Now we will run a pre-compiled benchmark and compare the simulator output to the expected output.

5.2 Simulating the go benchmark

Simulate the go benchmark by executing the following:

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

cp /cad2/ece552f/benchmarks/go.pisa-big .

cp /cad2/ece552f/benchmarks/2stone9.in .

sim-safe go.pisa-big 50 9 2stone9.in > go.out
```

This will simulate the program `go.pisa-big` (compiled for big-endian PISA) using `sim-safe` and will redirect the output from the program `go.pisa-big` to the file go.out (by using the '>'

character). Running this simulation will take up to several minutes depending on the load of your ug machine (use the `uptime` command to see the load; `man uptime` will give you the manual for the command). When the simulation completes, you will see some information about the simulation printed to the screen, like the number of instructions, the number of loads and stores, etc. We can redirect the simulator output to a file by using the `-redir:sim` flag of `sim-safe`. We can also eliminate the > (redirect) from the command line by using the `-redir:prog` flag of `sim-safe`. We use these flags and re-run the simulation. (Note that for formatting purposes, the `sim-safe` command below is broken into two lines, but should be entered as a single line. For the rest of this document, we use the convention that indenting the second line of a command indicates that there should be no line break when typing it.)

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

sim-safe -redir:sim go.simout -redir:prog go.progout
     go.pisa-big 50 9 2stone9.in
```

The flags and parameters for `sim-safe` are passed *before* the executable (`go.pisa-big`) and the executable's arguments. In general, we invoke `sim-safe` with: `sim-safe [arguments] executable [executable-arguments]`. On this run, the simulator output was written to the file `go.simout` and the program output of `go.pisa-big` was written to the file `go.progout`. Although, `go.out` (from the previous run) and `go.progout` should be the same, you may notice that in these files the value for `sim_num_insn` (the number of simulated instructions) is different! Small variations between different runs of the same program are possible with SimpleScalar. These variations are limited to a few thousand instructions at most and can be made negligible by simulating at least a hundred thousand instructions. The sources of variation are listed in the SimpleScalar `FAQ` file and copied here in italics:

- *Redirecting output will cause subtle changes in* **printf()** *execution*
- *Calls to* **time()** *and* **getrusage()** *will produce different results*
- *The size of your environment, which is imported into the simulated virtual memory space, affects the starting location of a program's stack pointer*
- *Small variations in floating point across platforms can affect execution*

We can verify that the `go.pisa-big` executable has simulated successfully by comparing the program output in the file `go.progout` to the expected output which is available in the file `/cad2/ece552f/benchmarks/go.out`. Verify that these file are identical by running the commands:

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

diff go.progout /cad2/ece552f/benchmarks/go.out
```

The Unix `diff` command should produce no output if the files are identical.

In some cases, we may not want to wait for the whole program to simulate. This is especially true when we are in the process of modifying the simulator and testing our new code. We can use the `-max:inst` flag of `sim-safe` to specify the maximum number of instructions to simulate. We simulate the first 10^7 instructions of `go.pisa-big` by running:

```
    cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

    sim-safe -max:inst 10000000 -redir:sim go.simout -redir:prog go.progout
         go.pisa-big 50 9 2stone9.in
```

We can verify that `sim-safe` is actually accepting the parameters that we pass by using the `-dumpconfig` flag, which causes `sim-safe` to write out the configuration to a file:

```
    cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

    sim-safe -dumpconfig config.txt -max:inst 10000000
         -redir:sim go.simout -redir:prog go.progout
         go.pisa-big 50 9 2stone9.in
```

You can look at the file `config.txt` to verify that the parameters were passed correctly.

5.3 Simulating an EIO trace

To simplify benchmark execution, we use EIO traces. These are previously generated traces which we feed to the sim-safe simulator. The following command executes an EIO trace from the gcc benchmark.

```
   sim-safe /cad2/ece552f/benchmarks/gcc.eio
```

Running the simulator with an EIO trace eliminates the need for a binary or configuration parameters, and makes execution 100% reproducible. For your information, the benchmark can also be run as follows:

```
  cp /cad2/ece552f/benchmarks/1stmt.i .
  sim-safe /cad2/ece552f/benchmarks/cc1.pisa-big -O 1stmt.i
```

The aforementioned commands first copy the input data file to the working directory and then run the simulator. You can also copy the EIO trace to your working directory and use that path when running the benchmark.

The gcc benchmark runs for about 3×10^8 instructions and takes about 5 minutes to complete on an unloaded ug machine. Recall that the `-max:inst` flag of `sim-safe` limits the number of simulated instructions and is useful for early testing purposes. You can verify the output of the benchmark executable by checking with the Unix `diff` command the output (assembly code) file `1stmt.s` against the expected output `/cad2/ece552f/benchmarks/1stmt.s.ref`:

```
diff 1stmt.s /cad2/ece552f/benchmarks/1stmt.s.ref
```

5.4 Compiling a microbenchmark executable

In this section, we will compile a microbenchmark program and simulate it using `sim-safe`. The `C` code for the microbenchmark program is listed in Figure 1 below. The program takes a number as its argument and returns the sum of 1+2+   up to the argument (e.g., pass it the number 5 and it will return the sum 15). We have a copy of this code available to save typing.

```
#include <stdio.h>

int main (int argc, char *argv[ ])
{
    int i;
    int sum = 0;

    if ( argc != 2 ){
       printf("Usage: %s <count>\n", argv[0]);
       exit(5);
    }

    for (i = 1; i <= atoi(argv[1]); i++){
       sum += i;
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

Figure 1: Microbenchmark program `C` source code

To compile this program for simulating in `sim-safe`, first copy the program to your working directory:

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

cp /cad2/ece552f/testcode/testexec.c .
```

Next we need to add SimpleScalar's `gcc` compiler to the Unix path. The compiler has been configured to generate big-endian code using the PISA instruction set. Run the following command to add the compiler to your path:

```
set path = ( $path /cad2/ece552f/compiler/bin )
```

To execute this command automatically every time you open a new shell, you may want to add the command to the appropriate place in the .cshrc file in your home directory. Now, we are ready to compile our microbenchmark program. We will compile two versions with different levels of optimization:

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

ssbig-na-sstrix-gcc testexec.c -O0 -o testexec0

ssbig-na-sstrix-gcc testexec.c -O2 -o testexec2
```

We can now simulate our microbenchmark programs by running:

```
cd ~/ece552/preassignment/simplesim-3.0d-ece552f-assign1

./sim-safe testexec0 1000

./sim-safe testexec2 1000
```

Look at the dynamic instruction count (`sim_num_insn`) reported by `sim-safe`. Does it vary between the `testexec0` executable and the `testexec2` executable? Why? Try passing different arguments to the executables. Does the dynamic instruction count vary as you expect?

6 Modifying sim-safe

In this section, we will modify `sim-safe` to count the number of load instructions and the number of data-hazards due to loads in a dynamic instruction stream. But first, we will familiarize ourselves with the simulator. So go ahead and open the file `sim-safe.c` in your favourite text editor.

6.1 Understanding sim-safe's main loop

In sim-safe the core of instruction execution occurs in the function `sim_main()`. Find the function `sim_main()`, which is at the end of the file. After a few statements in `sim_main()` you will find an infinite loop starting with the `while (TRUE)` statement. Each iteration of this loop corresponds to the execution of one instruction. The basic operation of the simulator is to read the instruction at address `regs_PC`, and execute it. In the case of functional simulation (i.e., `sim-safe`), this means immediately updating the registers and/or memory locations referenced by the current instruction with the appropriate values before fetching the next instruction to execute. The loop continues until either the instruction stream ends or the maximum instruction count is reached.

The first statement in the loop is (ignore all code that is within the `#ifdef TARGET_ALPHA ... #endif`)

```
regs.regs_R[MD_REF_ZERO] = 0;
```

This statement sets register `$0` to the value `0`. In the PISA (and MIPS) instruction set architecture, register `$0` always has the value `0`. The array `regs.regs_R[]` implements the general purpose register file. Look in `regs.h` and `regs.c` if you wish to learn more. (Feel free to avoid looking at `regs.h` and `regs.c` until you feel it is necessary. Using and modifying existing, complex and lengthy code is a common task for both hardware and software engineers. It is good practice to abstract away parts of unfamiliar code that are not of immediate interest.)

The next statement reads an instruction from memory:

```
MD_FETCH_INST(inst, mem, regs.regs_PC);
```

This macro is defined in `machine.h`, which contains many PISA-specific macros. Recall that in `C`, macros are expanded by the preprocessor and then passed to the compiler. The variable *mem* implements main memory. Again, you may look in `memory.c` if you care for details, or, for now, think of *mem* as simply a very large array (it is almost that, as it is implemented via an array of pointers pointing to 4K chunks of simulated memory. This approach is similar to page tables, as we will learn later in the course). The above statement reads an instruction (which is of size `sizeof(md_inst_t)` from the address `regs.regs_PC`, which is an integer that simulates the program counter.

Next is a statement that keeps a tally of executed instructions along with some bookkeeping statements. The next macro extracts the opcode field (`op`) from the raw instruction (`inst`):

```
    MD_SET_OPCODE (op, inst);
```

The opcode field specifies the actions of the instruction (e.g., add, multiply, load, store, etc.). `MD_SET_OPCODE()` is defined in `machine.h`.

Instruction execution takes places in the switch/case statement that follows. This switch statement uses the extracted opcode (`switch (op)`) to determine what needs to be done. The code within the switch statement probably looks weird because it is created by a series of macro expansions. Don't panic! Just keep on reading and soon it will make sense. The basic idea is that the switch statement contains one `case` statement for each instruction opcode. Inside each of the `case` statements are statements that simulate the instruction execution.

Below is a simplified version of the `switch (op)` statement with safety checking code removed, but with all important functional code retained:

```
    switch (op) {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3) \
    case OP: \
        SYMCAT(OP,_IMPL); \
        break;
#include "machine.def"
    default:
        panic("attempted to execute a bogus opcode");
}
```

We see that first the preprocessor macro called `DEFINST()` is defined over several lines. The "\" character tells the `C` preprocessor that the macro definition continues on the next line. The `DEFINST()` macro is defined as: "`case OP; SYMCAT(OP,_IMPL); break;`". Notice that only the parameter `OP` is used from `DEFINST()` and the parameters `MSK, NAME, OPFORM, RES, FLAGS, O1, O2, I1, I2,` and `I3` are unused. This is because `DEFINST()` is defined very generally for use in many different versions of the simulator. For the purpose of `sim-safe`, only the `OP` parameter is needed. In fact, we will be using some of the other parameters (`O1, O2, I1, I2,` and `I3`) in our modifications.

`SYMCAT(OP,_IMPL)` is a special macro that expands into `OP_IMPL` where `OP` is the first argument passed into `DEFINST()`. For example, the statement `SYMCAT(ADD,_IMPL)` expands into `ADD_IMPL()`. In turn, `ADD_IMPL()` is treated as macro and expanded as per the definition in the file `machine.def`. At this point, we have seen the definition of `DEFINST()` but we still have not seen the macro actually used.

To see the `DEFINST()` macro actually used, look in the file `machine.def`, which is `#include`'d at the end of the code listing above. The file `machine.def` contains a macro definition and a macro call per instruction opcode. The macro definition is of the form `X_IMPL` where `X` is an opcode such as `ADD` or `SUB`. The macro call is a call of `DEFINST(X,...)`.

Returning to `sim_main()` and the switch statement, we can now see what happens when this code passes through the preprocessor. Whenever `DEFINST()` is called it expands into a case element of the switch statement. Thus, each call to `DEFINST()` in the `machine.def` expands into a case statement that implements an opcode.

In summary, every call to `DEFINST()` in `machine.def` expands into the following code:

```
case OP: EXPR; break;
```

Then the preprocessor replaces EXPR, with the appropriate statements as defined in the file machine.def. As a result, after preprocessing, the switch (op) statement expands into a huge switch statement with a case element for every possible opcode. Each case element includes the code for the appropriate instruction (from machine.def). For example, the JUMP instruction expands to the case element (the formatting is quite ugly because of the macro preprocessing):

```
case JUMP :
    { (void)0;
     (regs.regs_NPC = ( ((regs.regs_PC) & 036000000000) |
     ((inst.b & 0x3ffffff) << 2) ));
    };
break;
```

You can see the full version of the expanded code by telling gcc to stop after the preprocessing stage with the command: gcc -E sim-safe.c -o sim-safe.pcc. This produces the file sim-safe.pcc. Using a text editor, search for sim_main() in sim-safe.pcc and then look carefully at the expanded switch statement. You should be able to find the case element for the JUMP instruction that is listed above.

The EXPR macros utilize other macros to access machine state, including the register file and memory. These macros that access state are part of the simulator. The file sim-safe.c includes appropriate definitions for our purposes. For example, to read a register, EXPR uses the GPR(x) macro, while it uses the SET_GPR(x) to write to a register. GPR(x) is defined simply as regs.regs_R[x] in sim-safe.c. That is, to read register x, we just access the x element of the array regs.regs_R[], which implements the register file. There are other macros that deal with floating point registers and memory, which you can find in sim-safe.c.

A more detailed description of how to use the DEFINST() macro is given at the beginning of the file machine.def. For our purposes, it suffices to know that for the current instruction O1 and O2 are the target register numbers, while I1, I2 and I3 are the source register numbers. There are up to 2 target registers and 3 source registers per instruction. In the PISA instruction set, an instruction may write up to 2 registers (e.g., load double, multiply) and read up to 3 registers (e.g., store double). The macro DNA is used to indicate that the current instruction does not use the corresponding register (i.e., if a specific instruction writes only 1 target register then O2 will have the value DNA). DNA stands for Does Not Access.

After the switch statement, we encounter a couple of if statements that detect faults and report statistics. There is also one that checks whether the instruction was a load or a store (if (MD_OP_FLAGS(op) & F_MEM)). This is also used for keeping statistics and has nothing to do with execution. Throughout the simulator, you may notice calls to functions named dlite_.... SimpleScalar implements a simple debugger called DLite. Simply ignore these for the time being. After all these statements, there are two statements that update the PC and a statement that checks whether the maximum count of instructions has been reached.

```
/* GO TO THE NEXT INSTRUCTION */

regs.regs_PC = regs.regs_NPC;
regs.regs_NPC += sizeof(md_inst_t);
```

6.2 Counting Load Instructions

Now that we have a high-level understanding of how the simulator works, we will modify it to count the number of load instructions in a dynamic instruction stream. Specifically, we will be adding some code to the main loop that increments a counter if the current instruction is a load. All necessary modifications are listed below.

1. We add a counter called `sim_num_loads` to the simulator. After the `#include` statements at the top of `sim-safe.c` add the following line:

   ```
   static counter_t sim_num_loads = 0;
   ```

   You may want to remind yourself of the purpose of the `static` keyword in `C`.

2. SimpleScalar provides an elaborate package for collecting statistics. It keeps an internal database of counters and other statistics-related data structures. It also prints out this database at the end of the simulation. We need to register our counter with this database so that it is printed by `sim-safe` at the end of the simulation. To register our counter, go to the `sim_reg_stats()` function and add the following statements to the end of the function:

   ```
   stat_reg_counter(sdb, "sim_num_loads",
        "total number of load instructions",
        &sim_num_loads, sim_num_loads, NULL);

   stat_reg_formula(sdb, "sim_load_ratio",
        "load instruction fraction",
        "sim_num_loads / sim_num_insn", NULL);
   ```

   The first statement registers our counter with the SimpleScalar statistics database. The second statement creates a new statistic, `sim_load_ratio`, and provides an equation that SimpleScalar executes to create it. The `sim_num_insn` counter is pre-defined in SimpleScalar, so we are free to use it when we create our own statistics.

3. Finally, we add the code that will increment the counter every time a load instruction is detected in the dynamic instruction stream. Go the the function `sim_main()` and add the following code after the switch statement (but before the end of the `while (TRUE)` loop):

   ```
   if ( (MD_OP_FLAGS(op) & F_MEM) && (MD_OP_FLAGS(op) & F_LOAD) ) {
        sim_num_loads++;
   }
   ```

   `MD_OP_FLAGS()` is a macro defined in `machine.h` that works on the current instruction's opcode, op, and allows us to compare to predefined constants like `F_MEM` and `F_LOAD` (which are also defined in `machine.h`) to determine the instruction type.

Save the modified version of `sim-safe.c` and compile it with the command `make sim-safe`. It should compile with no errors.

Now that you have modified `sim-safe` to report the number of load instructions in the dynamic instruction stream, re-run the go simulation and the testexec simulation from Sections 5.2, 5.4.

Check to see that the number of loads is less than the total number of instructions. Also, `sim-safe` by default reports the total number of loads and stores as `sim_num_refs`. Make sure that `sim_num_loads` is less than `sim_num_refs`.

Since it is easy to make a coding or a conceptual error when modifying a simulator, we should verify that we are performing the right measurements. To do so, we create a simple contrived example program (microbenchmark) where we can predict the results of our simulation.

6.3 Counting Load-To-Use Hazards

Now we will modify sim-safe to count the number of data hazards due to loads in a dynamic instruction stream. For simplicity, we will call these **load-to-use hazards** for the rest of this assignment. We assume a pipeline with full forwarding and bypassing.

First, we start with an overview of how we will measure the number of load-to-use hazards in `sim-safe`. Every time an instruction is executed, we check whether one of its inputs is the target of an immediately preceding load. If it is, we increment a counter of load-to-use hazards. To do this, we will modify `sim-safe` to collect the source register numbers for each instruction and to identify load instructions and their target registers. We present a step-by-step description of how this is done:

1. We want to measure how often the target register of a load is used by the next instruction. Here is how we are going to measure this. We will keep a record per register indicating when was the last time the register was the target of a load. Although we will call it time, we know that functional simulators count instructions, not cycles, so this time is really just an instruction count. For example, if register `$1` is set by a load instruction at time 1000, then we mark that this value will be ready at time 1002 (recall we need to stall for one cycle for the load value to be available in the pipeline we have studied in class with appropriate forwarding circuitry). A subsequent instruction then checks whether any of its inputs are not yet available by inspecting the corresponding record. In our example, if a subsequent instruction at time 1001 required register `$1` as an input, then we will increment a global counter of load-to-use hazards.

   By modifying the functional simulator `sim-safe` to include some timing information, we get both the speed of a functional simulator and the detail of a performance (timing) simulator.

2. First, we declare an array that will hold the ready time for each register. Go to the beginning of `sim-safe.c`, just after the initial `#include` statements, and add the following lines (where `MD_TOTAL_REGS` is predefined in `machine.h` as the max. number of registers that are available; this includes the general purpose registers, the floating point registers, hi, lo and others).

   ```
   /* ECE552 Pre-Assignment - BEGIN CODE*/
   static counter_t reg_ready[MD_TOTAL_REGS];
   /* ECE552 Pre-Assignment - END CODE*/
   ```

   We identify all changes to the file `sim-safe.c` with the comments `/* ECE552 Pre-Assignment - BEGIN CODE*/` and `/* ECE552 Pre-Assignment - END CODE*/`. Students should follow this convention for all modifications made to `sim-safe.c`. This convention will simplify debugging your code and help TAs identify your modifications.

3. After the above statement, declare and initialize to zero a variable that will hold the count of load-to-use hazards:

```
/* ECE552 Pre-Assignment - BEGIN CODE*/
static counter_t sim_num_lduh = 0;
/* ECE552 Pre-Assignment - END CODE*/
```

4. As mentioned earlier, SimpleScalar provides an elaborate package for collecting statistics. It keeps an internal database of counters and other statistics-related data structures. It also prints out this database at the end of the simulation. We need to register our counter with this database so that it is printed out at the end. To do so, go to the `sim_reg_stats()` function and add the following statements to the end of the function:

```
/* ECE552 Pre-Assignment - BEGIN CODE*/
stat_reg_counter(sdb, "sim_num_lduh",
                 "total number of load use hazards",
                 &sim_num_lduh, sim_num_lduh, NULL);

stat_reg_formula(sdb, "sim_load_use_ratio",
                 "load use fraction",
                 "sim_num_lduh / sim_num_insn", NULL);
/* ECE552 Pre-Assignment - END CODE*/
```

   The second statement registers a formula that reports the load-to-use hazard count as a fraction of all instructions executed in the dynamic instruction stream. The statistics database is implemented in the file `stats.c`.

5. Now, go to the beginning of `sim_main()` and declare the following local variables:

```
/* ECE552 Pre-Assignment - BEGIN CODE*/
int r_out[2], r_in[3];
/* ECE552 Pre-Assignment - END CODE*/
```

   Recall the `DEFINST()` macro's arguments include `O1, O2`, which hold the target register numbers, and `I1, I2`, and `I3`, which hold the source register numbers. The file `machine.def` defines these appropriately for every possible opcode. We will use our newly defined local variables `r_out[]` and `r_in[]` to copy these register numbers and use them for our purposes. Other statistics that you need may be collected in a similar way.

6. Modify the `DEFINST()` macro to extract the source register and target register numbers. If you browse through the file `decode.def` you will see definitions for a number of `D...(N)` macros. These macros map architectural register numbers (e.g., `$0, $r0, $f0`) to numbers, which we will use to index an array. This mapping is necessary, since inside the instruction code, the number 1 may be used to refer to both an integer register and floating point register depending on which instruction uses it. For example, an integer `ADD` instruction uses the number 1 to refer to the general purpose register 1. Whereas a floating point `ADD (FADD)` instruction uses the number 1 to refer to the floating point register 1. Hence the number 1 is used by different instructions to refer to different registers. With the macros defined in `decode.def`, SimpleScalar maps all these register/type combinations into a continuous series of numbers starting with 0. For example, the macro `DGPR(N)` maps the general purpose registers, while `DFPR(N)` maps the double precision floating point registers. As you can see, the `GPRs` are

mapped to 0...31 while FPRs are mapped to 32...63. There are other special-purpose macros for the other registers such as HI, LO, PC, and FCC (floating point condition codes). These D...(N) macros are called from the DEFINST() macros in the file machine.def to set the parameters O1, O2, I1, I2, and I3. Not every flavour of simulator needs these mapping functions; however, for our purposes it is convenient to use them.

Make the following modifications to the DEFINST() macro to copy the source register numbers and target register numbers to our local variables for use outside the switch statement. Go to sim_main() and find where DEFINST() is defined within the switch (op) statement. Change the definition of DEFINST() to the following:

```
/* ECE552 Pre-Assignment - BEGIN CODE*/
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3) \
case OP: \
   r_out[0] = (O1); r_out[1] = (O2); \
   r_in[0] = (I1); r_in[1] = (I2); r_in[2] = (I3); \
   SYMCAT(OP,_IMPL); \
   break;
/* ECE552 Pre-Assignment - END CODE*/
```

Make sure that there are no spaces after the '\' character on every line. The '\' is used to split the macro definition across multiple lines. A space breaks the definition and will cause errors that are difficult to track down. For extra safety, using parentheses around macro arguments (like O1) is a very good practice.

We are getting close! Now we add the code for collecting our statistic. Go after the switch statement, but stay within the while (TRUE) loop. Now include the following code:

```
    /* ECE552 Pre-Assignment - BEGIN CODE*/
{
    int i;
    for (i = 0; i < 3; i++) {
        if (r_in[i] != DNA && reg_ready [r_in [i]] > sim_num_insn) {
            if ((i == 0) && (MD_OP_FLAGS(op) & F_MEM) &&
                (MD_OP_FLAGS(op) & F_STORE)) {
                continue;
            }
            sim_num_lduh++;
            break;
        }
    }
}
    /* ECE552 Pre-Assignment - END CODE*/
```

This loop scans through all input registers and checks whether they are currently available. The DNA constant indicates that the corresponding source register is not really used by the instruction. We break out of the for loop as soon as we find an unavailable register since we only want to count a single load-to-use hazard per instruction, even if more than one register is unavailable.

Corner Cases:

There are two corner cases in the aforementioned code: First, if the dependent instruction is a store, and the only dependency is in the register that holds the store value, then no stalls occur. The reason is that the store value is needed at the beginning of the memory stage, instead of the beginning of the execute stage, and thus can be forwarded on time. Note that it is the first source register (I1) which holds the store value.

The second corner case involves any instruction with a double register (e.g., DGPR_D(N)) such as the dlw (i.e., double load word instruction). Although dlw has two destination registers, only one, the (O1), is "captured". The second destination register (O2) is implicitly computed as (O1 + 1), but appears as empty (DNA) in the instruction DEFINST, as shown below. You can ignore this corner case, and continue to rely on O1, O2, I1, I2 and I3 to identify dependences.

```
  #define DLW_IMPL                                                  \
  {                                                                 \
    word_t _result_hi, _result_lo;                                  \
    ...
    SET_GPR(RT, _result_hi);                                        \
    SET_GPR((RT) + 1, _result_lo);                                  \
  }
DEFINST(DLW,                    0x29,
        "dlw",                  "t,o(b)",
        RdPort,                 F_MEM|F_LOAD|F_DISP,
        DGPR_D(RT), DNA,        DNA, DGPR(BS), DNA)
```

7. We are almost there. The last thing we need to do is update the `reg_ready[]` array on loads. Go after the code we added in the previous step and add the following:

```
/* ECE552 Pre-Assignment - BEGIN CODE*/
if ((MD_OP_FLAGS(op) & F_MEM) && (MD_OP_FLAGS(op) & F_LOAD)) {
    if (r_out[0] != DNA)
        reg_ready[r_out[0]] = sim_num_insn + 2;
    if (r_out[1] != DNA)
        reg_ready[r_out[1]] = sim_num_insn + 2;
}
/* ECE552 Pre-Assignment - END CODE*/
```

This code flags target registers for load instructions indicating that they will not be available for the next instruction to see.

8. The modifications are now complete. Compile the modified version of `sim-safe.c` using the `make` command. It is probably a good idea to go over the above descriptions several times to make sure that you understand them.

7 Questions

Please post clarification questions on the discussion board. Also, bring your questions to the next tutorial session.