Project Part 1:
Simplescalar ISA and Functional Simulator
&
ILP Limit Study

Version 1.0
Due Date: 11:59 PM, Friday, September 7

ECE 721: Advanced Microarchitecture
Fall 2012, Professor Rotenberg

- READ THIS ENTIRE DOCUMENT.
- Late policy: 1 point deduction for each hour the project is late (that's 24 points/day).
- Even if you do not finish this part by the due date, you must build on it for later parts.
- Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy (see the section titled "Academic integrity" in the course syllabus). The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
- A Wolfware message board will be provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message board (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.

1. Introduction

In this course, you will augment a detailed superscalar processor simulator. Unlike the *trace-driven* simulators used in ECE 463/521, our simulator is *execution-driven*.

In trace-driven simulation, the simulated processor reads a trace of the dynamic instruction stream from a file. The trace was generated by a previous run of the program.

In execution-driven simulation, the simulated processor is just like a real processor. A real processor isn't fed a trace. It uses a program counter to fetch instructions from a compiled program binary, reads values from the register file and main memory, executes instructions using these values, and updates the register file, program counter, and main memory with new values. The simulated processor does this too. (In fact, the traces we used in ECE 463/521 were generated by an execution-driven simulator — how else could we produce a trace of the dynamic instruction stream?)

Because values are modeled, it is possible to detect simulator bugs. For example:

- If an instruction executes before its source operands are ready, the instruction will produce a wrong value.
- If you mispredict a branch instruction and do not repair the program counter, you'll continue fetching and executing down an incorrect path in the program.

To validate the simulated processor, we use two simulators concurrently [1,2]. This is shown in Figure 1. The *functional simulator* on the left-hand side of the diagram only produces values. It does not model timing. Therefore, it is quite trivial and has been verified. So we can compare its results, instruction-by-instruction, with the results produced by your simulated processor (shaded box in Figure 1). If your simulated processor does something wrong, you will know it almost immediately because you'll retire an instruction with a bad source or destination value. You'll know it's bad because it won't match the functional simulator.

As shown in Figure 1, the interface between the functional simulator and your timing simulator is called the *debug buffer*. Each entry in the debug buffer contains a single instruction executed by the functional simulator — the instruction opcode, the values it consumed and produced, addresses of loads/stores, etc. The functional simulator always runs a certain number of instructions ahead of your timing simulator. Thus, the debug buffer is not only useful for validating the timing simulator ("asserting" that control flow and data flow of the two simulators match), but also for simulating perfect branch prediction, perfect value prediction, perfect memory disambiguation, etc. (it can be used as an "oracle").

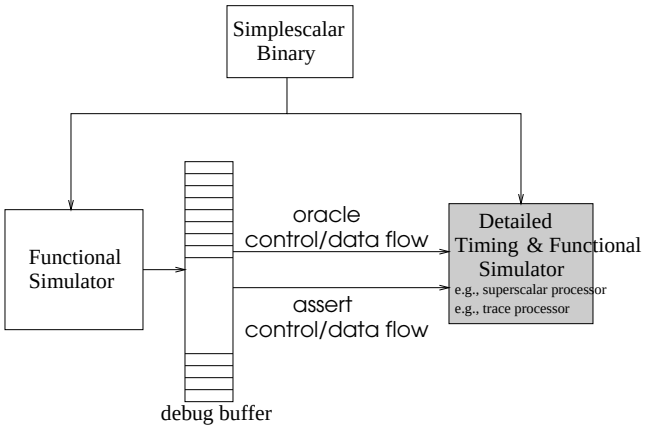

FIGURE 1. High-level view of simulator infrastructure. Results from the functional simulator (left) and timing simulator (right) are compared to validate the timing simulator ("assert control/data flow"). The functional simulator also enables perfect prediction modes ("oracle control/data flow"). Diagram taken from [2].

2. Getting started

2.1 Compiling

Currently, the infrastructure is only supported for LINUX machines.

1. Create a simulator directory under your home directory, I'll call it X for demonstration.
2. Download 721proj.zip from the course web page to your computer's /tmp directory.
3. cd /tmp
4. unzip 721proj.zip
5. mv 721proj ~/X/721proj
6. cd ~/X/721proj
7. You will see two directories: LibSS_smt/ and src1/. These are described in the next section.
8. cd LibSS_smt/lib.src ; make clean ; make
9. (go back to ~/X/721proj/ directory) cd src1 ; make clean ; make

If you have problems with building the bare simulator, send e-mail to ericro@ncsu.edu.

The Makefile in src1/ produces your simulator, called sim_little (linux runs on x86 machines, which are little-endian).

2.2 Simplescalar ISA

We are using a MIPS-like ISA called Simplescalar [3]. The course web page gives a pointer to the Simplescalar technical report. Follow directions on how to access the technical report. Then, you only need to read sections related to the ISA itself. The relevant sections you should read carefully are:

- Section 3 (The Simplescalar Architecture)
- Figures 2 and 3, Table 1
- Appendix A

3. Description of directories and source files

3.1 LibSS_smt/ (libSS_smt_little.a)

This is the modified Simplescalar functional simulator component. I modified it in several ways, mainly (1) addition of the debug buffer interface and (2) support for running multiple programs simultaneously, for multithreading research.

You will be using macros provided by the Simplescalar toolset. For example, to decode the opcode of an instruction, there is a macro you can use. These macros and other functionality will be described as needed (See Section 5.2).

3.2 src1/

This is the directory in which you will develop your detailed timing simulator (eventually). The only source code in the first release (hence the "1" in src1) is:

1. **simulator.cc** : This file contains two important functions. *Trace_consume_init*() is where you will initialize/allocate all of your dynamically created components (e.g., branch predictor), in future parts of the project. *Trace_consume*() replaces the "*main*()" function you are normally used to — *main*() is actually in LibSS_smt/. *Trace_consume*() contains an apparently infinite while loop from where you will call functions needed to process instructions every cycle. Each iteration of the while loop will correspond to a cycle, when you begin modeling timing in future parts of the project.

   In this part of the project, you will make a call to the functional simulator's debug buffer from *trace_consume*(). The function call grabs a single instruction from the debug buffer. You will then partially decode the instruction to collect statistics and also schedule the instruction in the context of an ILP limit study. See Section 5.

2. **input.cc** : This file contains the function *sim_options*(), called before *trace_consume_init*(). *Sim_options*() is where you can parse simulator command-line options, used to set simulation parameters.

   Adding new simulator command-line options works as follows.

   — First, pick a single letter or digit for the new flag (example: x).

   — Add the flag to the string called char **sim_optstring*, located in the file input.cc. If the flag will be followed by a number or any other piece of information, then add a single colon (:) after the flag (example: char *sim_optstring = "… x: …"). If the flag is a simple switch with no additional information after it, then don't include the colon (example: char *sim_optstring = "… x …"). There shouldn't be any spaces in *sim_optstring*.

   — In the switch statement within *sim_options*(), add a case statement for your new flag (example: case x: …). Within the case statement, you can configure the appropriate parameter(s) to the desired value(s). If there is information specified after the flag on the command-line, you can parse the information using the special string called *getopt_arg* (example: sscanf(getopt_arg, "...", …);).

   — *sim_options*() will automatically generate an output filename called "out.<sorted command-line flags>.<timedatestamp>". The first part of the filename is always "out". The second part is the command-line flags specified by the user. If no flags were specified, then "base" is used. The third part is a timestamp, only used if the filename already exists; that way, existing files are never overwritten.

   — The automatically-generated output filename can be overridden with the command-line flag "-f<filename>".

3. **output.cc** : This file contains *sim_stats*(), the function where you print out final output of the simulator. For example, this is where you will print out the final performance measure, IPC.

4. **info.h/cc** : These files provide a macro for printing stuff out. The macro is INFO() — the arguments are processed just like *printf*(). INFO() prints to the automatically generated output filename (see -2- above for how output filenames are generated).

4. Running benchmarks

See the project web page for directions on how to obtain the benchmarks. It points to a directory that contains the SPEC2K benchmarks. Each sub-directory is an individual benchmark, with (1) Simplescalar program binary, (2) input files that the benchmark requires, and (3) a "job" file for running the benchmark (see below). Of course, you cannot run the benchmark without the simulator, because the binary is a Simplescalar binary and not a native linux binary.

Our simulator can run multiple benchmarks simultaneously. A job file is specified on the command-line of the simulator. The job file contains the benchmark commands. There are as many lines in the job file as benchmarks you want to run. Since we will run only one benchmark at a time, the job file contains only a single benchmark command.

EXAMPLE #1:

If you want to simulate the first 1,000,000 dynamic instructions of the *gcc* benchmark:

```
sim_little -z1000000 job
```

Contents of file "job" is the *gcc* benchmark command:

```
gcc.exe expr.i -o expr.s
```

EXAMPLE #2:

If you want to skip the first 10,000,000 dynamic instructions and then simulate the next 1,000,000 dynamic instructions thereafter, for the *gcc* benchmark:

```
sim_little -Z10000000 -z1000000 job
```

Contents of file "job" is the *gcc* benchmark command:

```
gcc.exe expr.i -o expr.s
```

5. Tasks for Project Part 1

In this project, you will (1) compile the initial simulator infrastructure (Figure 1, where the shaded box is basically an empty shell), (2) exercise the debug buffer interface by grabbing instructions from the debug buffer, (3) partially decode the instructions and then collect/print out statistics, and (4) implement a very simple "ILP limit study" and print out a measure of available ILP in each benchmark.

Compiling the infrastructure is described in Section 2.

5.1 Exercise debug buffer

Within *trace_consume*(), add the following line in the while(1) loop.

db_t *db_ptr = THREAD[0]->get_instr();

LibSS_smt allows instantiating multiple programs. THREAD[x] refers to program x. Since you will only be running one program, x = 0. The function call returns a pointer to the next dynamic instruction, of type *db_t* (debug buffer type).

After getting the instruction, you need to check if the instruction is a system call (also called a trap) and service the system call. The following piece of code does this.

```
// Must service system calls (traps).
if (db_ptr->a_flags & F_TRAP) {           // trap instruction
   THREAD[0]->trap_now(db_ptr->a_inst);   // service the trap
   THREAD[0]->trap_resume();              // resume functional simulator
}
```

5.2 Decode instructions

5.2.1 db_t type

The *db_t* type is defined in LibSS_smt/include/debug.h. It contains a bunch of fields, prefixed with “a_”. The field “a_inst” is of type SS_INST_TYPE — a 64-bit instruction (made up of two 32-bit integers, a_inst.a/a_inst.b). See next subsection on how to decode an instruction *db_ptr->a_inst*.

The field “a_flags” is a bit vector. It provides a quick way of finding out the category of an instruction. See next subsection on how to extract useful information from the flags.

You may end up using other fields of the *db_t* type.

5.2.2 Fully and partially decoding instructions

The following Simplescalar macros, provided by LibSS_smt/include/**ss.h**, will be of use in decoding the instruction.

1. SS_OPCODE(<instruction>). <instruction> here is of type SS_INST_TYPE.

   This macro fully decodes the instruction, returning the mnemonic of a particular instruction Example: if (SS_OPCODE(db_ptr->a_inst) == BNE).... This statement checks to see if the instruction is a BNE instruction.

2. A number of #define’s are provided for testing the flags of an instruction. For example: if (db_ptr->a_flags & F_CTRL)…. This statement checks to see if the instruction is a control-type instruction (all branches). F_CTRL is a bit vector signifying control-transfer-instructions. In ss.h, you will find all of the other useful #define’s, like F_CTRL, that will help you detect types of instructions.

3. Many macros assume the local variable SS_INST_TYPE *inst* is defined. If you want these macros to work, you must declare the local variable *inst*, and then copy *db_ptr->a_inst* to *inst*. You must copy the two parts of the instruction individually: inst.a = db_ptr->a_inst.a; inst.b = db_ptr->a_inst.b.

   For example, the macro RS returns the 1st source register of the instruction *inst*. You will need this below to decode return instructions, which use a single source register, register 31.

5.2.3 What decode-related statistics to collect and print out

Your simulator should output the following:

1. Determine the number of control-transfer instructions, of any kind.
2. Determine the number of call instructions. These are jump-and-link (JAL) and jump-and-link-indirect (JALR).
3. Determine the number of return instructions. A return is a jump-indirect (JR) whose source register is register 31 (hint: use the RS macro described earlier).
4. Determine the number of load instructions, of any kind.
5. Determine the number of store instructions, of any kind.
6. Print the program counter (PC) of the 9,900th instruction encountered.
7. Print the memory addresses of the third load instruction and third store instruction encountered.
8. Determine the number of *not-taken* conditional branches. (hint: the PCs of sequential instructions differ by 8 in this ISA)
9. Determine the number of *taken* conditional branches. (hint: the PCs of sequential instructions differ by 8 in this ISA)

5.3 ILP Limit Study

To explore the available ILP in programs, implement an ILP limit study assuming the following very ideal conditions. These conditions make it relatively simple to implement an ILP limit study, because it is equivalent to constructing the data-flow graph of the dynamic instruction stream.

1. Infinite resources. This means an unbounded window (unbounded physical registers / ROB entries and unbounded reservation stations) and unbounded issue width.
2. 1-cycle latency for all instructions.
3. Perfect branch prediction. This eliminates all control dependences. Thus, an instruction only waits for all of its data dependences (registers and/or memory).
4. Oracle memory disambiguation. This means a load only waits for the closest prior store (in program order) to the same address. (It does not wait for even closer prior stores to *different* addresses, whose addresses are not yet computed by the time the load computes its address. This is the oracle aspect, since in reality the load would have to wait for these unknown store addresses, only to find out that they are different than the load's address.)

5. Perfect register renaming. This makes it simple to model the time at which registers are available. For each logical register, you only need to record the time of the last write (in program order) to the logical register.
6. Perfect memory renaming. This makes it simple to model the time at which memory bytes are available. For each byte in memory, you only need to record the time of the last store (in program order) to the byte.

The following subsections outline a basic approach for implementing the ILP limit study.

5.3.1 Register timestamps

The number of logical (architectural) registers in the ISA is TOTAL_ARCH_REGS (see LibSS_smt/include/mt.h).

Maintain a timestamp for each logical register. The timestamp of a logical register reflects the time of the last write (in program order) to the register.

5.3.2 Memory timestamps

Maintain a timestamp for each byte of memory that is ever stored to. The timestamp of a byte reflects the time of the last store (in program order) to the byte. For bytes that are never stored to, their timestamps are assumed to be 0 (available at time 0).

Maintaining memory timestamps is somewhat trickier than maintaining register timestamps for two reasons:

1. You can't represent memory as a statically-allocated array, because the number of unique bytes touched is not known *a priori* and is hypothetically unbounded (virtual memory). So you need to model memory using some form of hash table or other data structure that is indexed like an array but is dynamically expandable. There are STL classes that you may leverage, for example, the *map* class. It's up to you how to implement this functionality, however.

```
// Reference on using the STL map: http://www.sgi.com/tech/stl/Map.h
// For more tips, consult TA or instructor. But only after putting in some effort.
#include <map.h>
using namespace std;
// "key" is the load/store address, "contents" is the timestamp
map<unsigned int, unsigned int> MemoryTimestamps;
```

2. Depending on the load or store opcode, a load or store may access 1, 2, 4, or 8 bytes.

   The following load and store opcodes access 1 byte starting at address (*db_ptr->a_addr)*:
   LB, LB_RR, LBU, LBU_RR, SB, SB_RR

   The following load and store opcodes access 2 bytes starting at the halfword-aligned address (*db_ptr->a_addr & ~(1)*):
   LH, LH_RR, LHU, LHU_RR, SH, SH_RR

   The following load and store opcodes access 4 bytes starting at the word-aligned address (*db_ptr->a_addr & ~(3)*):
   LWL, SWL, LWR, SWR, LW, LW_RR, L_S, L_S_RR, SW, SW_RR, S_S, S_S_RR

The following load and store opcodes access 8 bytes starting at the doubleword-aligned address (*db_ptr->a_addr & ~(7)*):
DLW, DLW_RR, L_D, L_D_RR, DSW, DSW_RR, S_D, S_D_RR, DSZ, DSZ_RR

5.3.3 Scheduling an instruction

You need to determine the earliest time that an instruction can execute (i.e., schedule the instruction). An instruction must wait for all of its source operands (register and memory source operands). The times that register and memory source operands are available, are indicated by the register and memory timestamps (respectively), discussed previously in Section 5.3.1 and Section 5.3.2.

For an instruction, find the maximum time among all its register source operands. There are two types of register source operands, rsrc and rsrcA. You don't care about the distinction for this project, just handle them the same way. *db_ptr->a_num_rsrc* indicates the number of rsrc operands. The actual rsrc operands are indicated by *db_ptr->a_rsrc[...].n* (n is the logical register number, e.g., n=22 means R22). Similarly, *db_ptr->a_num_rsrcA* indicates the number of rsrcA operands. The actual rsrcA operands are indicated by *db_ptr->a_rsrcA[...].n* (n is the logical register number, e.g., n=22 means R22). You should be able to see this in the *db_t* struct.

Load instructions are unique in that they also have memory source operands (in addition to register source operands). They read bytes from memory. Therefore, for loads, find the maximum time among all register source operands and memory source operands. Register source operands were discussed above (rsrc and rsrcA). The memory source operands are all bytes read by the load. You use *db_ptr->a_addr* and the opcode to determine which bytes are read by the load, as discussed in Section 5.3.2.

The maximum time among all register and memory source operands, is the time that an instruction can execute. (Note that this assumes perfect branch prediction, since we only model data dependences and not control dependences.)

5.3.4 Updating register and memory timestamps

An instruction may write into one or more registers or store to memory. Register and memory timestamps must be updated accordingly, to reflect the time of these productions.

For an instruction, update the timestamps of all destination registers written by the instruction. The updated timestamp should be the time that the instruction executes (Section 5.3.3), plus 1 to reflect the 1-cycle latency of all instructions. *db_ptr->a_num_rdst* indicates the number of destination registers. The actual destination registers are indicated by *db_ptr->a_rdst[...].n* (n is the logical register number, e.g., n=22 means R22).

Store instructions are unique in that they write one or more bytes in memory. Therefore, for a store instruction, update the timestamps of all bytes written by the store. The updated timestamp should be the time that the store instruction executes (Section 5.3.3), plus 1 to reflect the 1-cycle latency of all instructions.You use *db_ptr->a_addr* and the opcode to determine which bytes are written by the store, as discussed in Section 5.3.2.

5.3.5 What to print out for your ILP limit study

CYCLES: Output the number of cycles to "execute" the benchmark. This is the maximum time (i.e., time that an instruction executed) among all instructions.

IPC: Output the instructions per cycle (IPC), which reflects the average instruction-level parallelism. This is simply total instructions divided by total cycles.

5.4 Modifying ILP Limit Study to account for branch mispredictions

The ILP Limit Study described in Section 5.3 assumes two very ideal control-flow conditions: (1) unlimited instruction fetch bandwidth and (2) perfect branch prediction. The implication of these assumptions is that all instructions are fetched in cycle 0. Therefore, the cycle in which an instruction executes is determined solely by the latest-available register or memory operand.

This section describes a way to model real branch prediction in the ILP Limit Study, so that the impact of branch mispredictions on available ILP can be evaluated. Essentially, a mispredicted branch delays the fetching of all later instructions. This effect can be modeled with a single "mispredition timestamp", which indicates the cycle when the most recent mispredicted branch executed. It is updated and used as follows:

1. Updating the misprediction timestamp: (a) At the beginning of simulation, the misprediction timestamp is 0. (b) All branches will be predicted using a pre-packaged branch predictor module (details below). If a branch is mispredicted, then the misprediction timestamp is set to the branch's execution cycle plus 1 to reflect the 1-cycle latency of all instructions.

2. Using the misprediction timestamp: The misprediction timestamp will affect when all subsequent instructions execute. That's because subsequent instructions cannot be fetched -- hence executed -- earlier than the misprediction timestamp (reflecting the control dependence on the mispredicted branch). Whereas with perfect branch prediction the execution cycle of an instruction was the maximum of only its register and memory timestamps, with real branch prediction its execution cycle is the maximum of its register and memory timestamps *and* the misprediction timestamp.

5.4.1 Using pre-packaged branch predictor

A pre-packaged branch predictor is included in the simulator distribution and is automatically linked by the Makefile. To predict a branch, all you need to do is call a single wrapper function:

bool bpred_wrapper(db_t *db_ptr)

The wrapper function takes in the pointer to the instruction (which has all the information that the branch predictor needs to decode the branch type and apply the appropriate predictor structures) and returns "true" if the branch is mispredicted or "false" if the branch is not mispredicted.

You must call the bpred_wrapper() function for all branch instructions (of any branch type, since it can predict conditional branches, calls, returns, and direct and indirect jumps), and only branch

instructions. The bpred_wrapper() function will print an error message and exit the simulator if you call it for any instruction which is not a branch.

5.4.2 Support two modes: perfect branch prediction and real branch prediction

Your simulator must support two modes for the ILP Limit Study: perfect branch prediction and real branch prediction. A global variable has already been defined for you in the source files "parameters.cc/h", called PERFECT_BRANCH_PREDICTION, which dynamically configures the simulator for either mode. It is initialized to "false". You need to modify the source file "input.cc" to support a new command-line parameter "-P" that, when specified by the user on the simulator command-line, sets PERFECT_BRANCH_PREDICTION to "true". Section 3.2 explained how to add command-line parameters to "input.cc".

5.5 Debug runs

The project web page lists benchmarks to run and the corresponding output of my or the TA's simulator. You can use this to debug your simulator. Your output may not match exactly: the number of instructions executed can vary slightly depending on your environment. But you should be within 1%.

5.6 What to hand in

Hand in your simulator (directions are posted on the project web page). The TA will compile and run your simulator to verify it. The TA will also inspect the code.

5.7 Grading

- **[30 points]** Decode-related outputs (Section 5.2.3). Three points for each output that matches within 1%. Since there are only nine outputs, there are three free points to reach a total of 30 points.
- **[70 points]** IPC (hence CYCLES) from limit study matches within 1%.

6. References

[1] S. Breach. Design and Evaluation of a Multiscalar Processor. Ph.D. Thesis, University of Wisconsin - Madison, August 1998.

[2] E. Rotenberg. Trace Processors: Exploiting Hierarchy and Speculation. Ph.D. Thesis, University of Wisconsin - Madison, August 1999.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.