The SimpleScalar Tool Set, Version 2.0

Doug Burger

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA

Todd M. Austin

SimpleScalar LLC
2395 Timbercrest Court
Ann Arbor, MI 48105

*Contact: info@simplescalar.com
http://www.simplescalar.com

This report describes release 2.0 of the SimpleScalar tool set, a suite of powerful computer simulation tools that provide both detailed and high-performance simulation of modern microprocessors. The new release offers more tools and capabilities, precompiled binaries, cleaner interfaces, better documentation, easier installation, improved portability, and higher performance. This report contains a complete description of the tool set, including retrieval and installation instructions, a description of how to use the tools, a description of the target SimpleScalar architecture, and many details about the internals of the tools and how to customize them. With this guide, the tool set can be brought up and generating results in under an hour (on supported platforms).

1 Overview

Modern processors are incredibly complex marvels of engineering that are becoming increasingly hard to evaluate. This report describes the SimpleScalar tool set (release 2.0), which performs fast, flexible, and accurate simulation of modern processors that implement the SimpleScalar architecture (a close derivative of the MIPS architecture [4]). The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators. We provide sets of precompiled binaries (including SPEC95), plus a modified version of GNU GCC (with associated utilities) that allows you to compile your own SimpleScalar test binaries from FORTRAN or C code.

The advantages of the SimpleScalar tools are high flexibility, portability, extensibility, and performance. We include five execution-driven processor simulators in the release. They range from an extremely fast functional simulator to a detailed, out-of-order issue, superscalar processor simulator that supports non-blocking caches and speculative execution.

The tool set is portable, requiring only that the GNU tools may be installed on the host system. The tool set has been tested extensively on many platforms (listed in Section 2). The tool set is easily extensible. We designed the instruction set to support easy annotation of instructions, without requiring a retargeted compiler for incremental changes. The instruction definition method, along with the ported GNU tools, makes new simulators easy to write, and the old ones even simpler to extend. Finally, the simulators have been aggressively tuned for performance, and can run codes approaching "real" sizes in tractable amounts of time. On a 200-MHz Pentium Pro, the fastest, least detailed simulator simulates about four million machine cycles per second, whereas the most detailed processor simulator simulates about 150,000 per second.

The current release (version 2.0) of the tools is a major improvement over the previous release. Compared to version 1.0 [2], this release includes better documentation, enhanced performance, compatibility with more platforms, precompiled SPEC95 SimpleScalar binaries, cleaner interfaces, two new processor simulators, option and statistic management packages, a source-level debugger (DLite!) and a tool to trace the out-of-order pipeline.

The rest of this document contains information about obtaining, installing, running, using, and modifying the tool set. In Section 2 we provide a detailed procedure for downloading the release, installing it, and getting it up and running. In Section 3, we describe the SimpleScalar architecture and details about the target (simulated) system. In Section 4, we describe the SimpleScalar processor simulators and discuss their internal workings. In Section 5, we describe two tools that enhance the utility of the tool set: a pipeline tracer and a source-level debugger (for stepping through the program being simulated). In Section 6, we provide the history of the tools' development, describe current and planned efforts to extend the tool set, and conclude. In Appendix A and Appendix B contain detailed definitions of the SimpleScalar instructions and system calls, respectively.

2 Installation and Use

This tool set is distributed "as is" in the hope that it will be useful. The tool set comes with no warranty, and no author or distributor accepts any responsibility for the consequences of its use.

Everyone is granted permission to copy, modify and redistribute this tool set under the following conditions:

- *This tool set is distributed for non-commercial use only. Please contact the maintainer for restrictions applying to commercial use of these tools.*
- *Permission is granted to anyone to make or distribute copies of this tool set, either as received or modified, in any medium, provided that all copyright notices, permission and nonwarranty notices are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this document.*
- *Permission is granted to distribute these tools in compiled or executable form under the same conditions that apply for source code, provided that either: (1) it is accompanied by the corresponding machine-readable source code, or (2) it is accompanied by a written offer, with no time limit, to give anyone a machine-readable copy of the corresponding source code in return for reimbursement of the cost of distribution. This written offer must permit verbatim duplication by anyone, or (3) it is distributed by someone who received only the executable form, and is accompanied by a copy of the written offer of source code that they received concurrently.*

The latest version of the SimpleScalar license can be found at the following web address:

```
http://www.simplescalar.com/license.html
```

2.1 Obtaining the tools

The tools can either be obtained through the World Wide Web, or by conventional ftp. For example, to get the file `simplesim.tar.gz` via the WWW, enter the URL:

```
www.simplescalar.com
```

and to obtain the same file with traditional ftp:

```
ftp ftp.simplescalar.com
user: anonymous
password: enter your e-mail address here
cd pub/simplescalar
get simplesim.tar
```

Note the "tar.gz" suffix: by requesting the file without the ".gz" suffix, the ftp server uncompresses it automatically. To get the compressed version, simply request the file with the ".gz" suffix.

The five distribution files in the directory (which are symbolic links to the files containing the latest version of the tools) are:

- **simplesim.tar.gz** - contains the simulator sources, the instruction set definition macros, and test program source and binaries. The directory is 1 MB compressed and 4 MB uncompressed. When the simulators are built, the directory (including object files) will require 11 MB. This file is required for installation of the tool set.
- **simpleutils.tar.gz** - contains the GNU binutils source (version 2.5.2), retargeted to the SimpleScalar architecture. These utilities are not required to run the simulators themselves, but is required to compile your own SimpleScalar benchmark binaries (e.g. test programs other than the ones we provide). The compressed file is 3 MB, the uncompressed file is 14 MB, and the build requires 52 MB.
- **simpletools.tar.gz** - contains the retargeted GNU compiler and library sources needed to build SimpleScalar benchmark binaries (GCC 2.6.3, glibc 1.0.9, and f2c), as well as pre-built big- and little-endian versions of libc. This file is needed only to build benchmarks, not to compile or run the simulators. The tools are 11 MB compressed, 47 MB uncompressed, and the full installation requires 70 MB.
- **simplebench.big.tar.gz** - contains a set of the SPEC95 benchmark binaries, compiled to the SimpleScalar architecture running on a big-endian host. The binaries take under 5 MB compressed, and are 29 MB when uncompressed.
- **simplebench.little.tar.gz** - same as above, except that the binaries were compiled to the SimpleScalar architecture running on a little-endian host.

Once you have selected the appropriate files, place the downloaded files into the desired target directory. If you obtained the files with the ".gz" suffix, run the GNU decompress utility (gunzip). The files should now have a ".tar" suffix. To remove the directories from the archive:

```
tar xf filename.tar
```

If you download and unpack all files, release, you should have the following subdirectories with following contents:

- **simplesim-2.0** - the sources of the SimpleScalar processor simulators, supporting scripts, and small test benchmarks. It also holds precompiled binaries of the test benchmarks.
- **binutils-2.5.2** - the GNU binary utilities code, ported to the SimpleScalar architecture.
- **ssbig-na-sstrix** - the root directory for the tree in which the big-endian SimpleScalar binary utilities and compiler tools will be installed. The unpacked directories contain header files and a pre-compiled copy of libc and a necessary object file.
- **sslittle-na-sstrix** - same as above, except that this directory holds the little-endian versions of the SimpleScalar utilities.
- **gcc-2.6.3** - the GNU C compiler code, targeted toward the SimpleScalar architecture.
- **glibc-1.09** - the GNU libraries code, ported to the SimpleScalar architecture.
- **f2c-1994.09.27** - the 1994 release of AT&T Bell Labs' FORTRAN to C translator code.
- **spec95-big** - precompiled SimpleScalar SPEC95 benchmark binaries (big-endian version).
- **spec95-little** - precompiled SimpleScalar SPEC95 benchmark binaries (little-endian version)

2.2 Installing and running Simplescalar

We depict a graphical overview of the tool set in Figure 1. Benchmarks written in FORTRAN are converted to C using Bell Labs' f2c converter. Both benchmarks written in C and those converted from FORTRAN are compiled using the SimpleScalar

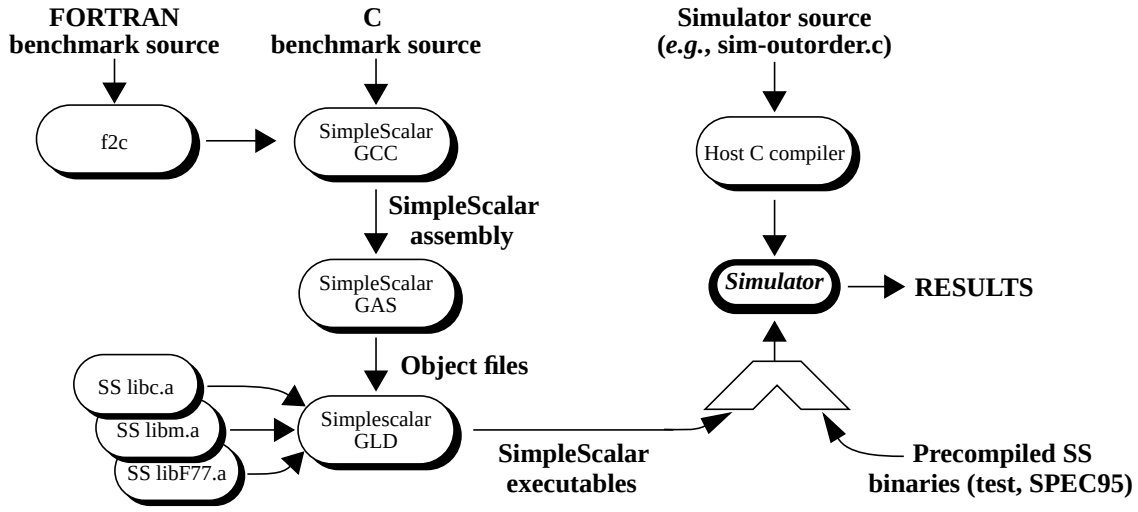

Figure 1. SimpleScalar tool set overview

version of GCC, which generates SimpleScalar assembly. The SimpleScalar assembler and loader, along with the necessary ported libraries, produce SimpleScalar executables that can then be fed directly into one of the provided simulators. (The simulators themselves are compiled with the host platform's native compiler; any ANSI C compiler will do).

If you use the precompiled SPEC95 binaries or the precompiled test programs, all you have to install is the simulator source itself. If you wish to compile your own benchmarks, you will have to install and build the GCC tree and optionally (recommended) the GNU binutils. If you wish to modify the support libraries, you will have to install, modify, and build the glibc source as well.

The SimpleScalar architecture, like the MIPS architecture [4], supports both big-endian and little-endian executables. The tool set supports compilation for either of these targets; the names for the big-endian and little-endian architecture are **ssbig-na-sstrix** and **sslittle-na-sstrix**, respectively. You should use the target endian-ness that matches your host platform; the simulators may not work correctly if you force the compiler to provide cross-endian support. To determine which endian your host uses, run the **endian** program located in the `simplesim-2.0/` directory. For simplicity, the following instructions will assume a big-endian installation. In the following instructions, we will refer to the directory in which you are installing SimpleScalar as `$IDIR/`.

The simulators come equipped with their own loader, and thus you do not need to build the GNU binary utilities to run simulations. However, many of these utilities are useful, and we recommend that you install them. If desired, build the GNU binary utilities[1]:

```
cd $IDIR/binutils-2.5.2
configure --host=$HOST --target=ssbig-na-
     sstrix --with-gnu-as --with-gnu-ld --pre-
     fix=$IDIR
make
make install
```

$HOST here is a "canonical configuration" string that represents your host architecture and system (CPU-COMPANY-SYSTEM). The string for a Sparcstation running SunOS would be sparc-sun-sunos4.1.3, running Solaris: sparc-sun-solaris2, a 386 running Solaris: i386-sun-solaris2.4, etc. A complete list of supported $HOST strings resides in `$IDIR/gcc-2.6.3/INSTALL`.

This installation will create the needed directories in `$IDIR` (these include `bin/`, `lib/`, `include/`, and `man/`). Once the binutils have been built, build the simulators themselves. This is necessary to do before building GCC, since one of the binaries is needed for the cross-compiler build. You should edit `$IDIR/simplesim-2.0/Makefile` to use the desired compile flags (e.g., the correct optimization level). To use the GNU BFD loader instead of the custom loader in the simulators, uncomment -DBFD_LOADER in the Makefile. To build the simulators:

```
cd $IDIR/simplesim-2.0
make
```

If desired, build the compiler:

```
cd $IDIR/gcc-2.6.3
configure --host=$HOST --target=ssbig-na-
     sstrix --with-gnu-as --with-gnu-ld --pre-
     fix=$IDIR
make LANGUAGES=c
../simplesim-2.0/sim-safe ./enquire -f >!
     float.h-cross
make install
```

We provide pre-built copies of the necessary libraries in `ssbig-na-sstrix/lib/`, so you do not need to build the code in **glibc-1.09**, unless you change the library code. Building these libraries is tricky, and we do not recommend it unless you have a specific need to do so. In that event, to build the libraries:

```
cd $IDIR/glibc-1.09
configure --prefix=$IDIR/ssbig-na-sstrix
     ssbig-na-sstrix
```

1. You must have GNU Make to do the majority of installations described in this document. To check if you have the GNU version, execute "make -v" or "gmake -v". The GNU version understands this switch and displays version information.

```
setenv CC $IDIR/bin/ssbig-na-sstrix-gcc
unsetenv TZ
unsetenv MACHINE
make
make install
```

Note that you must have already built the SimpleScalar simulators to build this library, since the glibc build requires a compiled simulator to test target machine-specific parameters such as endian-ness.

If you have FORTRAN benchmarks, you will need to build f2c:

```
cd $IDIR/f2c-1994.09.27
make
make install
```

The entire tool set should now be ready for use. We provide precompiled test binaries (big- and little-endian) and their sources in `$IDIR/simplesim2.0/tests`). To run a test:

```
cd $IDIR/simplesim-2.0
sim-safe tests/bin.big/test-math
```

The test should generate about a page of output, and will run very quickly. The release has been ported to—and should run on—the following systems:

- gcc/AIX 413/RS6000
- xlc/AIX 413/RS6000
- gcc/HPUX/PA-RISC
- gcc/SunOS 4.1.3/SPARC
- gcc/Linux 1.3/x86
- gcc/Solaris 2/SPARC
- gcc/Solaris 2/x86
- gcc/DEC Unix 3.2/Alpha
- c89/DEC Unix 3.2/Alpha
- gcc/FreeBSD 2.2/x86
- gcc/WindowsNT/x86

3 The Simplescalar architecture

The SimpleScalar architecture is derived from the MIPS-IV ISA [4]. The tool suite defines both little-endian and big-endian versions of the architecture to improve portability (the version used on a given host machine is the one that matches the endianness of the host). The semantics of the SimpleScalar ISA are a superset of MIPS with the following notable differences and additions:

- There are no architected delay slots: loads, stores, and control transfers do not execute the succeeding instruction.
- Loads and stores support two addressing modes—for all data types—in addition to those found in the MIPS architecture. These are: indexed (register+register), and auto-increment/decrement.
- A square-root instruction, which implements both single- and double-precision floating point square roots.
- An extended 64-bit instruction encoding.

We list all SimpleScalar instructions in Figure 2. We provide a complete list of the instruction semantics (as implemented in the simulator) in Appendix A. In Table 1, we list the architected registers in the SimpleScalar architecture, their hardware and software names (which are recognized by the assembler), and a description of each. Both the number and the semantics of the registers are identical to those in the MIPS-IV ISA.

In Figure 3, we depict the three instruction encodings of SimpleScalar instructions: *register*, *immediate*, and *jump* formats. All instructions are 64 bits in length.

The register format is used for computational instructions. The immediate format supports the inclusion of a 16-bit constant. The jump format supports specification of 24-bit jump targets. The register fields are all 8 bits, to support extension of the architected registers to 256 integer and floating point registers. Each instruction format has a fixed-location, 16-bit opcode field that facilitates fast instruction decoding.

The *annote* field is a 16-bit field that can be modified post-compile, with annotations to instructions in the assembly files. The annotation interface is useful for synthesizing new instructions without having to change and recompile the assembler. Annotations are attached to the opcode, and come in two flavors: bit and field annotations. A bit annotation is written as follows:

```
lw/a        $r6,4($r7)
```

The annotation in this example is /a. It specifies that the first bit of the annotation field should be set. Bit annotations /a through /p set bits 0 through 15, respectively. Field annotations are written in the form:

```
lw/6:4(7)   $r6,4($r7)
```

This annotation sets the specified 3-bit field (from bit 4 to bit 6 *within* the 16-bit annotation field) to the value 7.

System calls in SimpleScalar are managed by a proxy handler (located in `syscall.c`) that intercepts system calls made by the simulated binary, decodes the system call, copies the system call arguments, makes the corresponding call to the host's operating system, and then copies the results of the call into the simulated program's memory. If you are porting SimpleScalar to a new platform, you will have to code the system call translation from SimpleScalar to your host machine in `syscall.c`. A list of all SimpleScalar system calls is provided in Appendix B.

SimpleScalar uses a 31-bit address space, and its virtual memory is laid out as follows:

```
0x00000000   Unused
0x00400000   Start of text segment
0x10000000   Start of data segment
0x7fffc000   Stack base (grows down)
```

The top of the data segment (which includes init and bss) is held in `mem_brk_point`. The areas below the text segment and above the stack base are unused.

4 Simulator internals

In this section, we describe the functionality of the processor simulators that accompany the tool set. We describe each of the simulators, their functionality, command-line arguments, and internal structures.

The compiler outputs binaries that are compatible with the MIPS ECOFF object format. Library calls are handled with the ported version of GNU GLIBC and POSIX-compliant Unix system calls. The simulators currently execute only user-level code. All SimpleScalar-related extensions to GCC are contained in the `config/ss` subdirectory of the GCC source tree that comes

Control

j - jump
jal - jump and link
jr - jump register
jalr - jump and link register
beq - branch == 0
bne - branch != 0
blez - branch <= 0
bgtz - branch > 0
bltz - branch < 0
bgez - branch >= 0
bct - branch FCC TRUE
bcf - branch FCC FALSE

Load/Store

lb - load byte
lbu - load byte unsigned
lh - load half (short)
lhu - load half (short) unsigned
lw - load word
dlw - load double word
l.s - load single-precision FP
l.d - load double-precision FP
sb - store byte
sbu - store byte unsigned
sh - store half (short)
shu - store half (short) unsigned
sw - store word
dsw - store double word
s.s - store single-precision FP
s.d - store double-precision FP

addressing modes:
(C)
(reg+C) (with pre/post inc/dec)
(reg+reg) (with pre/post inc/dec)

Integer Arithmetic

add - integer add
addu - integer add unsigned
sub - integer subtract
subu - integer subtract unsigned
mult - integer multiply
multu - integer multiply unsigned
div - integer divide
divu - integer divide unsigned
and - logical AND
or - logical OR
xor - logical XOR
nor - logical NOR
sll - shift left logical
srl - shift right logical
sra - shift right arithmetic
slt - set less than
sltu - set less than unsigned

Floating Point Arithmetic

add.s - single-precision (SP) add
add.d - double-precision (DP) add
sub.s - SP subtract
sub.d - DP subtract
mult.s - SP multiply
mult.d - DP multiply
div.s - SP divide
div.d - DP divide
abs.s - SP absolute value
abs.d - DP absolute value
neg.s - SP negation
neg.d - DP negation
sqrt.s - SP square root
sqrt.d - DP square root
cvt - int., single, double conversion
c.s - SP compare
c.d - DP compare

Miscellaneous

nop - no operation
syscall - system call
break - declare program error

Figure 2. Summary of SimpleScalar instructions

Hardware Name	Software Name	Description
$0	$zero	zero-valued source/sink
$1	$at	reserved by assembler
$2-$3	$v0-$v1	fn return result regs
$4-$7	$a0-$a3	fn argument value regs
$8-$15	$t0-$t7	temp regs, caller saved
$16-$23	$s0-$s7	saved regs, callee saved
$25-$25	$t8-$t9	temp regs, caller saved
$26-$27	$k0-$k1	reserved by OS
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$s8	saved regs, callee saved
$31	$ra	return address reg
$hi	$hi	high result register
$lo	$lo	low result register
$f0-$f31	$f0-$f31	floating point registers
$fcc	$fcc	floating point condition code

Table 1: SimpleScalar architecture register definitions

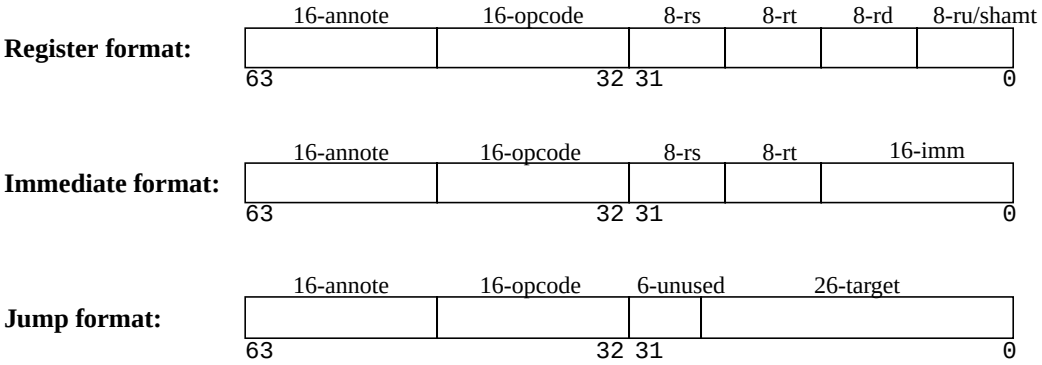

Figure 3. SimpleScalar architecture instruction formats

with the distribution.

The architecture is defined in `ss.def`, which contains a macro definition for each instruction in the instruction set. Each macro defines the opcode, name, flags, operand sources and destinations, and actions to be taken for a particular instruction.

The instruction actions (which appear as macros) that are common to all simulators are defined in `ss.h`. Those actions that require different implementations in different simulators are defined in each simulator code file.

When running a simulator, `main()` (defined in `main.c`) does all the initialization and loads the target binary into memory. The routine then calls `sim_main()`, which is simulator-specific, defined in each simulator code file. `sim_main()` predecodes the entire text segment for faster simulation, and then begins simulation from the target program entry point.

The following command-line arguments are available in all simulators included with the release:

-h	prints the simulator help message.
-d	turn on the debug message.
-i	start execution in the DLite! debugger (see Section 5.2). This option is not supported in the **sim-fast** simulator.
-q	terminate immediately (for use with -dumpconfig).
-dumpconfig <file>	generate a configuration file saving the command-line parameters. Comments are permitted in the config files, and begin with a #.
-config <file>	read in and use a configuration file. These files may reference other config files.

4.1 Functional simulation

The fastest, least detailed simulator (**sim-fast**) resides in `sim-fast.c`. **sim-fast** does no time accounting, only functional simulation—it executes each instruction serially, simulating no instructions in parallel. **sim-fast** is optimized for raw speed, and assumes no cache, instruction checking, and has no support for DLite!.

A separate version of **sim-fast**, called **sim-safe**, also performs functional simulation, but checks for correct alignment and access permissions for each memory reference. Although similar, **sim-fast** and **sim-safe** are split (i.e., protection is not toggled with a command-line argument in a merged simulator) to maximize performance. Neither of the simulators accept any additional command-line arguments. Both versions are very simple: less than 300 lines of code—they therefore make good starting points for understanding the internal workings of the simulators. In addition to the simulator file, both **sim-fast** and **sim-safe** use the following code files (not including header files): `main.c`, `syscall.c`, `memory.c`, `regs.c`, `loader.c`, `ss.c`, `endian.c`, and `misc.c`. **sim-safe** also uses `dlite.c`.

4.2 Cache simulation

The SimpleScalar distribution comes with two functional cache simulators; **sim-cache** and **sim-cheetah**. Both use the file `cache.c`, and they use `sim-cache.c` and `sim-cheetah.c`, respectively. These simulators are ideal for fast simulation of caches if the effect of cache performance on execution time is not needed.

sim-cache accepts the following arguments, in addition to the universal arguments described in Section 4:

-cache:dl1 <config>	configures a level-one data cache.
-cache:dl2 <config>	configures a level-two data cache.
-cache:il1 <config>	configures a level-one instr. cache.
-cache:il2 <config>	configures a level-two instr. cache.
-tlb:dtlb <config>	configures the data TLB.
-tlb:itlb <config>	configures the instruction TLB.
-flush <boolean>	flush all caches on a system call; (<boolean> = 0 | 1 | true | TRUE | false | FALSE).
-icompress	remap SimpleScalar's 64-bit instructions to a 32-bit equivalent in the simulation (i.e., model a machine with 4-word instructions).
-pcstat <stat>	generate a text-based profile, as described in Section 4.3.

The cache configuration (<config>) is formatted as follows:

`<name>:<nsets>:<bsize>:<assoc>:<repl>`

Each of these fields has the following meaning:

<*name*>	cache name, must be unique.
<*nsets*>	number of sets in the cache.
<*bsize*>	block size (for TLBs, use the page size).
<*assoc*>	associativity of the cache (power of two).
<*repl*>	replacement policy (l | f | r), where *l* = LRU, *f* = FIFO, *r* = random replacement.

The cache size is therefore the product of <nsets>, <bsize>, and <assoc>. To have a unified level in the hierarchy, "point" the instruction cache to the name of the data cache in the corresponding level, as in the following example:

```
-cache:il1 il1:128:64:1:l
-cache:il2 dl2
-cache:dl1 dl1:256:32:1:l
-cache:dl2 ul2:1024:64:2:l
```

The defaults used in **sim-cache** are as follows:

L1 instruction cache:	il1:256:32:1:l	(8 KB)
L1 data cache:	dl1:256:32:1:l	(8 KB)
L2 unified cache:	ul2:1024:64:4:l	(256 KB)
instruction TLB:	itlb:16:4096:4:l	(64 entries)
data TLB:	dtlb:32:4096:4:l	(128 entries)

sim-cheetah is based on work performed by Ragin Sugumar and Santosh Abraham while they were at the University of Michigan. It uses their Cheetah cache simulation engine [6] to generate simulation results for multiple cache configurations with a single simulation. The Cheetah engine simulates fully associative caches efficiently, as well as simulating a sometimes-optimal replacement policy. This policy was called MIN by Belady [1], although the simulator refers to it as *opt*. Opt uses future knowledge to select a replacement; it chooses the block that will be referenced the furthest in the future (if at all). This policy is optimal for read-only instruction streams. It is not optimal for write-back caches because it may be more expensive to replace a block referenced further in the future if the block must be written back, as opposed to a clean block referenced slightly less far in the future.

Horwitz et al. [3] formally described an optimal algorithm that includes writes; however, only MIN is implemented in the simulator.

We have included the Cheetah engine as a stand-alone library, which is built and resides in the `libcheetah/` directory. **sim-cheetah** accepts the following command-line arguments, in addition to those listed at the beginning of Section 4:

-refs [inst | data | unified]	specify which reference stream to analyze.
-C [fa | sa | dm]	fully associative, set associative, or direct-mapped cache.
-R [lru | opt]	replacement policy.
-a <sets>	log base 2 minimum bound on number of sets to simulate simultaneously.
-b <sets>	log base 2 maximum bound on set number.
-l <line>	cache line size (in bytes).
-n <assoc>	maximum associativity to analyze (in log base 2).
-in <interval>	cache size interval to report when simulating fully associative caches.
-M <size>	maximum cache size of interest.
-C <size>	cache size for direct-mapped analyses.

Both of these simulators are ideal for performing high-level cache studies that do not take access time of the caches into account (e.g., studies that are concerned only with miss rates). To measure the effect of cache organization upon the execution time of real programs, however, the timing simulator described in Section 4.4 must be used.

4.3 Profiling

The distribution comes with a functional simulator that produces voluminous and varied profile information. **sim-profile** can generate detailed profiles on instruction classes and addresses, text symbols, memory accesses, branches, and data segment symbols.

sim-profile takes the following command-line arguments, which toggle the various profiling features:

-iclass	instruction class profiling (e.g. ALU, branch).
-iprof	instruction profiling (e.g., bnez, addi).
-brprof	branch class profiling (e.g., direct, calls, conditional).
-amprof	addr. mode profiling (e.g., displaced, R+R).
-segprof	load/store segment profiling (e.g., data, heap).
-tsymprof	execution profile by text symbol (functions).
-dsymprof	reference profile by data segment symbol.
-taddrprof	execution profile by text address.
-all	turn on all profiling listed above.

Three of the simulators (**sim-profile**, **sim-cache**, and **sim-outorder**) support text segment profiles for statistical integer counters. The supported counters include any added by users, so long as they are correctly "registered" with the SimpleScalar stats package included with the simulator code (see Section 4.5). To use the counter profiles, simply add the command-line flag:

-pcstat <stat>	where <stat> is the integer counter that you wish to profile by text address.

To generate the statistics for the profile, follow the following example:

```
sim-profile -pcstat sim_num_insn test-math >&!
   test-math.out
objdump -dl test-math >! test-math.dis
textprof.pl test-math.dis test-math.out
   sim_num_insn_by_pc
```

We show a segment of the text profile output in Figure 4. Make sure that "objdump" is the version created when compiling the binutils. Also, the first line of `textprof.pl` must be changed to reflect your system's path to Perl (which must be installed on your system for you to use this script). As an aside, note that "-taddrprof" is equivalent to "-pcstat sim_num_insn".

4.4 Out-of-order processor timing simulation

The most complicated and detailed simulator in the distribution, by far, is **sim-outorder** (the main code file for which is `sim-outorder.c`—about 3500 lines long). This simulator supports out-of-order issue and execution, based on the Register Update Unit [5]. The RUU scheme uses a reorder buffer to automatically rename registers and hold the results of pending instructions. Each cycle the reorder buffer retires completed instructions in program order to the architected register file.

The processor's memory system employs a load/store queue. Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. Loads may be satisfied either by the memory system or by an earlier store value residing in the queue, if their addresses match. Speculative loads may generate cache misses, but speculative TLB misses stall the pipeline until the branch condition is known.

We depict the simulated pipeline of **sim-outorder** in Figure 5. The main loop of the simulator, located in `sim_main()`, is structured as follows:

```
ruu_init();
for (;;) {
  ruu_commit();
  ruu_writeback();
  lsq_refresh();
  ruu_issue();
  ruu_dispatch();
  ruu_fetch();
}
```

This loop is executed once for each target (simulated) machine cycle. By walking the pipeline in reverse, inter-stage latch synchronization can be handled correctly with only one pass through each stage. When the target program terminates with an `exit()` system call, the simulator performs a `longjmp()` to `main()` to generate the statistics.

The fetch stage of the pipeline is implemented in `ruu_fetch()`. The fetch unit models the machine instruction bandwidth, and takes the following inputs: the program counter, the predictor state, and misprediction detection from the branch execution unit(s). Each cycle, it fetches instructions from only one I-cache line (and it blocks on an I-cache miss until the miss

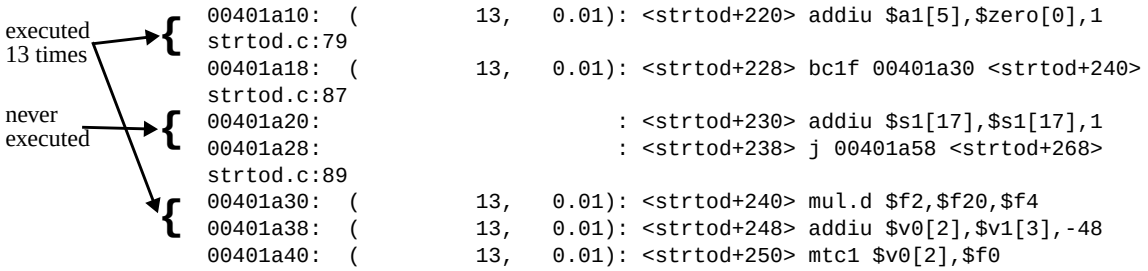

Figure 4. Sample output from text segment statistical profile

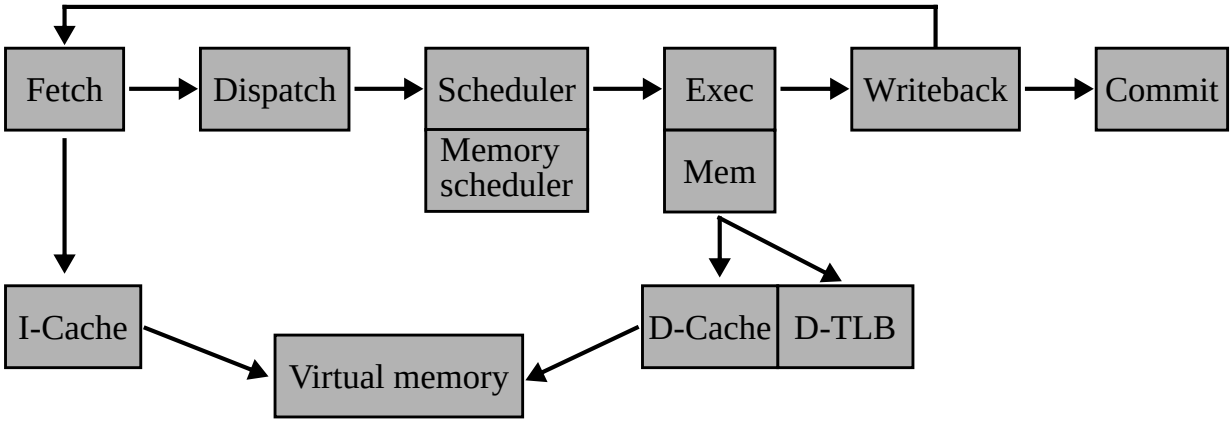

Figure 5. Pipeline for sim-outorder

completes). After fetching the instructions, it places them in the dispatch queue, and probes the line predictor to obtain the correct cache line to access in the next cycle.

The code for the dispatch stage of the pipeline resides in `ruu_dispatch()`. This routine is where instruction decoding and register renaming is performed. The function uses the instructions in the input queue filled by the fetch stage, a pointer to the active RUU, and the rename table. Once per cycle, the dispatcher takes as many instructions as possible (up to the dispatch width of the target machine) from the fetch queue and places them in the scheduler queue. This routine is the one in which branch mispredictions are noted. (When a misprediction occurs, the simulator uses speculative state buffers, which are managed with a copy-on-write policy). The dispatch routine enters and links instructions into the RUU and the load/store queue (LSQ), as well as splitting memory operations into two separate instructions (the addition to compute the effective address and the memory operation itself).

The issue stage of the pipeline is contained in `ruu_issue()` and `lsq_refresh()`. These routines model instruction wakeup and issue to the functional units, tracking register and memory dependences. Each cycle, the scheduling routines locate the instructions for which the register inputs are all ready. The issue of ready loads is stalled if there is an earlier store with an unresolved effective address in the load/store queue. If the address of the earlier store matches that of the waiting load, the store value is forwarded to the load. Otherwise, the load is sent to the memory system.

The execute stage is also handled in `ruu_issue()`. Each cycle, the routine gets as many ready instructions as possible from the scheduler queue (up to the issue width). The functional units' availability is also checked, and if they have available access ports, the instructions are issued. Finally, the routine schedules writeback events using the latency of the functional units (memory operations probe the data cache to obtain the correct latency of the operation). Data TLB misses stall the issue of the memory operation, are serviced in the commit stage of the pipeline, and currently assume a fixed latency. The functional units' latencies are hardcoded in the definition of `fu_config[]` in `sim-outorder.c`.

The writeback stage resides in `ruu_writeback()`. Each cycle it scans the event queue for instruction completions. When it finds a completed instruction, it walks the dependence chain of instruction outputs to mark instructions that are dependent on the completed instruction. If a dependent instruction is waiting only for that completion, the routine marks it as ready to be issued. The writeback stage also detects branch mispredictions; when it determines that a branch misprediction has occurred, it rolls the state back to the checkpoint, discarding the erroneously issued instructions.

`ruu_commit()` handles the instructions from the writeback stage that are ready to commit. This routine does in-order committing of instructions, updating of the data caches (or memory) with store values, and data TLB miss handling. The routine keeps

retiring instructions at the head of the RUU that are ready to commit until the head instruction is one that is not ready. When an instruction is committed, its result is placed into the architected register file, and the RUU/LSQ resources devoted to that instruction are reclaimed.

sim-outorder runs about an order of magnitude slower than **sim-fast**. In addition to the arguments listed at the beginning of Section 4, **sim-outorder** uses the following command-line arguments:

Specifying the processor core

-fetch:ifqsize <size>
: set the fetch width to be <size> instructions. Must be a power of two. The default is 4.

-fetch:speed <ratio>
: set the ratio of the front end speed relative to the execution core (allowing <ratio> times as many instructions to be fetched as decoded per cycle).

-fetch:mplat <cycles>
: set the branch misprediction latency. The default is 3 cycles.

-decode:width <insts>
: set the decode width to be <insts>, which must be a power of two. The default is 4.

-issue:width <insts>
: set the maximum issue width in a given cycle. Must be a power of two. The default is 4.

-issue:inorder
: force the simulator to use in-order issue. The default is false.

-issue:wrongpath
: allow instructions to issue after a misspeculation. The default is true.

-ruu:size <insts>
: capacity of the RUU (in instructions). The default is 16.

-lsq:size <insts>
: capacity of the load/store queue (in instructions). The default is 8.

-res:ialu <num>
: specify number of integer ALUs. The default is 4.

-res:imult <num>
: specify number of integer multipliers/dividers. The default is 1.

-res:memports <num>
: specify number of L1 cache ports. The default is 2.

-res:fpalu <num>
: specify number of floating point ALUs. The default is 4.

-res: fpmult <num>
: specify number of floating point multipliers/dividers. The default is 1.

Specifying the memory hierarchy

All of the cache arguments and formats used in **sim-cache** (listed at the beginning of Section 4.2) are also used in **sim-outorder**, with the following additions:

-cache:dl1lat <cycles>
: specify the hit latency of the L1 data cache. The default is 1 cycle.

-cache:d12lat <cycles>
: specify the hit latency of the L2 data cache. The default is 6 cycles.

-cache:il1lat <cycles>
: specify the hit latency of the L1 instruction cache. The default is 1 cycle.

-cache:il2lat <cycles>
: specify the hit latency of the L2 instruction cache. The default is 6 cycles.

-mem:lat <1st> <next>
: specify main memory access latency (first, rest). The defaults are 18 cycles and 2 cycles.

-mem:width <bytes>
: specify width of memory bus in bytes. The default is 8 bytes.

-tlb:lat <cycles>
: specify latency (in cycles) to service a TLB miss. The default is 30 cycles.

Specifying the branch predictor

Branch prediction is specified by choosing the following flag with one of the six subsequent arguments. The default is a bimodal predictor with 2048 entries.

-bpred <type>

nottaken	always predict not taken.
taken	always predict taken.
perfect	perfect predictor.
bimod	bimodal predictor, using a branch target buffer (BTB) with 2-bit counters.
2lev	2-level adaptive predictor.
comb	combined predictor (bimodal and 2-level adaptive).

The predictor-specific arguments are listed below:

-bpred:bimod <size>
: set the bimodal predictor table size to be <size> entries.

-bpred:2lev <l1size> <l2size> <hist_size> <xor>
: specify the 2-level adaptive predictor. <l1size> specifies the number of entries in the first-level table, <l2size> specifies the number of entries in the second-level table, <hist_size> specifies the history width, and <xor> allows you to xor the history and the address in the second level of the predictor. This organization is depicted in Figure 6. In Table 2 we show how these parameters correspond to modern prediction schemes. The default settings for the four parameters are 1, 1024, 8, and 0, respectively.

-bpred:comb <size>
: set the meta-table size of the combined predictor to be <size> entries. The default is 1024.

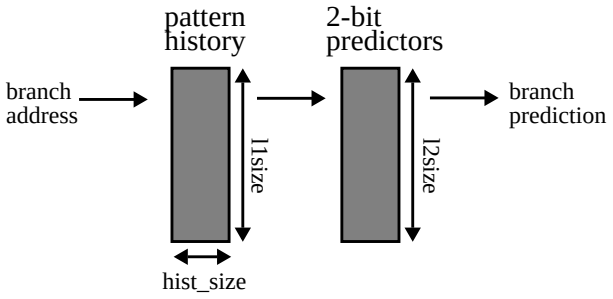

Figure 6. 2-level adaptive predictor structure

predictor	l1_size	hist_size	l2_size	xor
GAg	1	W	2^W	0
GAp	1	W	$>2^W$	0
PAg	N	W	2^W	0
PAp	N	W	2^{N+W}	0
gshare	1	W	2^W	1

Table 2: Branch predictor parameters

-bpred:ras <size>

set the return stack size to <size> (0 entries means to return stack). The default is 8. entries.

-bpred:btb <sets> <assoc>

configure the BTB to have <sets> sets and an associativity of <assoc>. The defaults are 512 sets and an associativity of 4.

-bpred:spec_update <stage>

allow speculative updates of the branch predictor in the decode or writeback stages (<stage> = [ID|WB]). The default is non-speculative updates in the commit stage.

Visualization

-pcstat <stat>

record statistic <stat> by text address; described in Section 4.3.

-ptrace <file> <range>

pipeline tracing, described in Section 5.

4.5 Simulator code file descriptions

The following list describes the functionality of the C code files in the `simplesim-2.0/` directory, which are used by all of the simulators.

- `bitmap.h`: Contains support macros for performing bitmap manipulation.
- `bpred.[c,h]`: Handles the creation, functionality, and updates of the branch predictors. `bpred_create()`, `bpred_lookup()`, and `bpred_update()` are the key interface functions.
- `cache.[c,h]`: Contains general functions to support multiple cache types (e.g., TLBs, instruction and data caches). Uses a linked-list for tag comparisons in caches of low associativity (less than or equal to four), and a hash table for tag comparisons in higher-associativity caches. The important interfaces are `cache_create()`, `cache_access()`, `cache_probe()`, `cache_flush()`, and `cache_flush_addr()`.
- `dlite.[c,h]`: Contains the code for DLite!, the source-level target program debugger.
- `endian.[c,h]`: Defines a few simple functions to determine byte- and word-order on the host and target platforms.
- `eval.[c,h]`: Contains code to evaluate expressions, used in DLite!.
- `eventq.[c,h]`: Defines functions and macros to handle ordered event queues (used for ordering writebacks). The important interface functions are `eventq_queue()` and `eventq_service_events()`.
- `loader.[c,h]`: Loads the target program into memory, sets up the segment sizes and addresses, sets up the initial call stack, and obtains the target program entry point. The interface is `ld_load_prog()`.
- `main.c`: Performs all initialization and launches the main simulator function. The key functions are `sim_options()`, `sim_config()`, `sim_main()`, and `sim_stats()`.
- `memory.[c,h]`: Contains functions for reading from, writing to, initializing, and dumping the contents of the target main memory. Memory is implemented as a large flat space, each portion of which is allocated on demand. `mem_access()` is the important interface function.
- `misc.[c,h]`: Contains numerous useful support functions, such as `fatal()`, `panic()`, `warn()`, `info()`, `debug()`, `getcore()`, and `elapsed_time()`.
- `options.[c,h]`: Contains the SimpleScalar options package code, used to process command-line arguments and/or option specifications from config files. Options are registered with an option database (see the functions called `opt_reg_*()`). `opt_print_help`() generates a help listing, and `opt_print_options()` prints the current options' state.
- `ptrace.[c,h]`: Contains code to collect and produce pipeline traces from **sim-outorder**.
- `range.[c,h]`: Holds code that interprets program range commands used in DLite!.
- `regs.[c,h]`: Contains functions to initialize the register files and dump their contents.
- `resource.[c,h]`: Contains code to manage functional unit resources, divided up into classes. The three defined functions create the resource pools and busy tables (`res_create_pool()`), return a resource from the specified pool if available (`reg_get()`), and dump the contents of a pool (`res_dump()`).
- `sim.h`: Contains a few extern variable declarations and function prototypes.
- `stats.[c,h]`: Contains routines to handle statistics measuring target program behavior. As with the options pack-

age, counters are "registered" by type with an internal database. The `stat_reg_*()` routines register counters of various types, and `stat_reg_formula()` allows you to register expressions constructed of other statistics. `stat_print_stats()` prints all registered statistics. The statistics package also has facilities to measure distributions; `stat_reg_dist()` creates an array distribution, `stat_reg_sdist()` creates a sparse array distribution, and `stat_add_sample()` updates a distribution.

- `ss.[c,h]`: Defines macros to expedite the processing of instructions, numerous constants needed across simulators, and a function to print out individual instructions in a readable format.
- `ss.def`: Holds a list of macro calls (the macros are defined in the simulators and `ss.h` and `ss.c`), each of which defines an instruction. The macro calls accept as arguments the opcode, name of the instruction, sources, destinations, actions to execute, and other information. This file serves as the definition of the instruction set.
- `symbol.[c,h]`: Holds routines to handle program symbol and line information (used in DLite!).
- `syscall.[c,h]`: Contains code that acts as the interface between the SimpleScalar system calls (which are POSIX-compliant) and the system calls on the host machine.
- `sysprobe.c`: Determines byte and word order on the host platform, and generates appropriate compiler flags.
- `version.h`: Defines the version number and release date of the distribution.

5 Utilities

In this section we describe the utilities that accompany the SimpleScalar tool set; pipeline tracing and a source-level debugger.

5.1 Out-of-order pipeline tracing

The tool set provides the ability to extract and view traces of the out-of-order pipeline. Using the "-ptrace" option, a detailed history of all instructions executed in a range may be saved to a file. The information saved includes instruction fetch, retirement, and stage transitions. The syntax of this command is as follows:

-ptrace <file> <start>:<end>

<file> is the file to which the trace will be saved. <start> and <end> are the instruction numbers at which the trace will be started and stopped. If they are left blank, the trace will start at the beginning and/or stop at the end of the program, respectively.

For example:

-ptrace FOO.trc 100:500

trace from instructions 100 to 500, store the trace in file FOO.src.

-ptrace FOO.trc :10000

trace from program beginning to instruction 10000.

-ptrace FOO.trc :

trace the entire program execution.

The traces may be viewed with the `pipeview.pl` Perl script, which is provided in the simplesim-2.0 directory. (You will have to update the first line of `pipeview.pl` to have the correct path to your local Perl binary, and you must have Perl installed on your system).

```
pipeview.pl <ptrace_file>
```

We depict sample output from the pipetracer in Figure 7.

5.2 The DLite! debugger

Release 2.0 of SimpleScalar includes a lightweight symbolic debugger called DLite!, which runs with all simulators except for **sim-fast**. DLite! allows you to step through the *benchmark target code*, not the simulator code. The debugger can be incorporated into a simulator by adding only four function calls (which have already been added to all simulators in the distribution). The needed four function prototypes are in `dlite.h`.

To use the debugger in a simulation, add the "-i" option (which stands for interactive) to the simulator command line. Below we list the set of commands that DLite! accepts.

Getting help and getting out:

help [string]	print command reference.
version	print DLite! version information.
quit	exit simulator.
terminate	generate statistics and exit simulator.

Running and setting breakpoints:

step	execute next instruction and break.
cont [addr]	continue execution (optionally continuing starting at <addr>).
break <addr>	set breakpoint at <addr>, returns <id> of breakpoint.
dbreak <addr> [r,w,x]	set data breakpoint at <addr> for (r)ead, (w)rite, and/or e(x)ecute, returns <id> of breakpoint.
rbreak <range> [r,w,x]	set breakpoint at <range> for (r)ead, (w)rite, and/or e(x)ecute, returns <id> of breakpoint.
breaks	list active code and data breakpoints.
delete <id>	delete breakpoint <id>.
clear	clear all breakpoints (code and data).

Printing information:

print [modifiers] <expr>	print the value of <expr> using optional modifiers.
display [modifiers] <expr>	display the value of <expr> using optional modifiers.
option <string>	print the value of option <string>.
options	print the values of all options.
stat <string>	print the value of a statistical variable.
stats	print the values of all statistical variables.
whatis <expr>	print the type of <expr>.
regs	print all register contents.
iregs	print all instruction register contents.

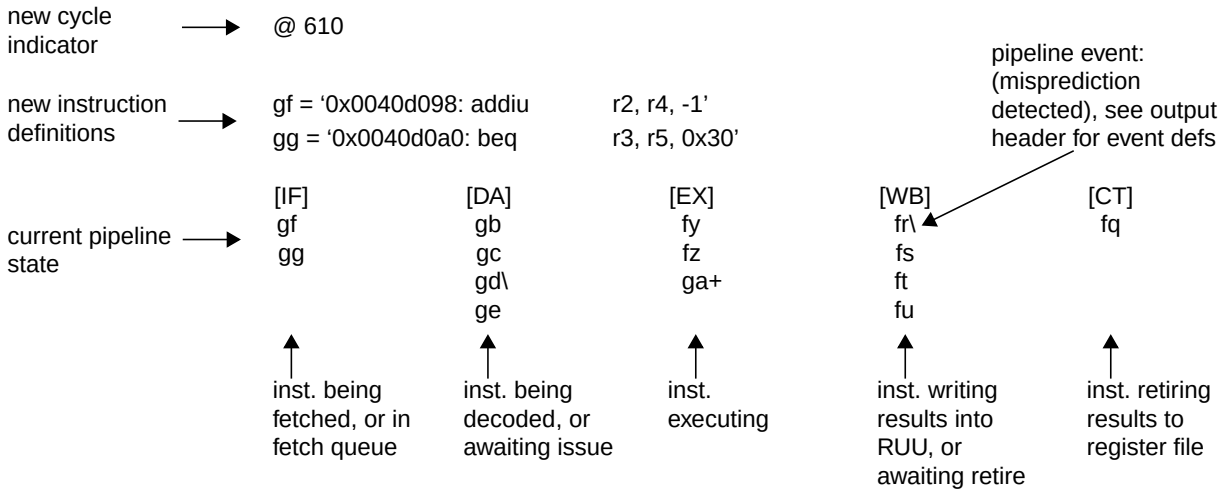

Figure 7. Example of sim-outorder pipetrace

fpregs print all floating point register contents.

mstate [string] print machine-specific state.

dump <addr> [count]
dump memory at <addr> (optionally for <count> words).

dis <addr> [count]
disassemble instructions at <addr> (optionally for <count> instructions).

symbols print the value of all program symbols.

tsymbols print the value of all program text symbols.

dsymbols print the value of all program data symbols.

symbol <string>
print the value of symbol <string>.

Legal arguments:

Arguments <addr>, <cnt>, <expr>, and <id> are any legal expression:

<expr> ← <factor> +|- <expr>
<factor> ← <term> *|/ <factor>
<term> ← (<expr>)
| - <term> | <const> | <symbol> | <file:loc>
<symbol> ← <literal> | <function name> | <register>
<literal> ← [0-9]+ | 0x[0-9,a-f]+ | 0[0-7]+
<register> ← $r[0-31] | $f[0-31] | $pc | $fcc | $hi | $lo

Legal ranges:

<range> ← <address> | <instruction> | <cycle>
<address> ← @<function name>:{+<literal>}
<instruction>← {<literal>}:{<literal>}
<cycle> ← #{<literal>}:{<literal>}

Omitting optional arguments to the left of the colon will default to the smallest value permitted in that range. Omitting an optional argument at the right of the colon will default to the largest value permitted in that range.

Legal command modifiers:

b print a byte
h print a half (short)
w print a word (default)
t print in decimal format (default)
o print in octal format
x print in hex format
1 print in binary format
f print float
d print double
c print character
s print string

Examples of legal commands:

```
break main+8
break 0x400148
dbreak stdin w
dbreak sys_count wr
rbreak @main:+279
rbreak 2000:3500
rbreak #:100        cycle 0 to cycle 100
rbreak :            entire execution
```

6 Summary

The SimpleScalar tool set was written by Todd Austin over about one and a half years, between 1994 and 1996. He continues to add improvements and updates. The ancestors of the tool set date back to the mid to late 1980s, to tools written by Manoj Franklin. At the time the tools were developed, both individuals were research assistants at the University of Wisconsin-Madison Computer Sciences Department, supervised by Professor Guri Sohi. Scott Breach provided valuable assistance with the implementation of the proxy system calls. The first release was assembled, debugged, and documented by Doug Burger, also a research assistant at Wisconsin, who is the maintainer of the second release as well. Kevin Skadron, currently at Princeton, implemented many of the more recent branch prediction mechanisms.

Many exciting extensions to SimpleScalar are both underway and planned. Efforts have begun to extend the processor simula-

tors to simulate multithreaded processors and multiprocessors. A Linux port to SimpleScalar (enabling simulation of the OS on a kernel with publicly available sources) is planned, using device-level emulation and a user-level file system. Other plans include extending the tool set to simulate ISAs other than SimpleScalar and MIPS (Alpha and SPARC ISA support will be the first additions).

As they stand now, these tools provide researchers with a simulation infrastructure that is fast, flexible, and efficient. Changes in both the target hardware and software may be made with minimal effort. We hope that you find these tools useful, and encourage you to contact us with ways that we can improve the release, documentation, and the tools themselves.

References

[1] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[2] Doug Burger, Todd M. Austin, and Steven Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. Technical Report 1308, Computer Sciences Department, University of Wisconsin, Madison, WI, July 1996.

[3] L. P. Horwitz, R. M. Karp, R. E. Miller, and A. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[4] Charles Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, January 1995.

[5] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[6] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, May 1993.

A Instruction set definition

This appendix lists all SimpleScalar instructions with their opcode, assembler format, and semantics. The semantics are expressed as a C-style expression that uses the extended operators and operands described in Table 3. Operands that are not listed in Table 3 refer to actual instruction fields described in Figure 3. For each instruction, the next PC value (NPC) defaults to the current PC value plus eight (CPC+8) unless otherwise specified.

A.1 Control instructions

J: Jump to absolute address.
- Opcode: 0x01
- Format: J target
- Semantics: SET_NPC((CPC&0xf0000000)|(TARGET<<2)))

JAL: Jump to absolute address and link.
- Opcode: 0x02
- Format: JAL target
- Semantics: SET_NPC((CPC\&0xf0000000) | (TARGET<<2))
  SET_GPR(31, CPC + 8))

JR: Jump to register address.
- Opcode: 0x03
- Format: JR rs
- Semantics: TALIGN(GPR(RS))
  SET_NPC(GPR(RS))

JALR: Jump to register address and link.
- Opcode: 0x04
- Format: JALR rs
- Semantics: TALIGN(GPR(RS))
  SET_GPR(RD, CPC + 8)
  SET_NPC(GPR(RS))

BEQ: Branch if equal.
- Opcode: 0x05
- Format: BEQ rs,rt,offset
- Semantics:
```
if (GPR(RS) == GPR(RT))
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BNE: Branch if not equal.
- Opcode: 0x06
- Format: BEQ rs,rt,offset
- Semantics:
```
if (GPR(RS) != GPR(RT))
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BLEZ: Branch if less than or equal to zero.
- Opcode: 0x07
- Format: BLEZ rs,offset
- Semantics:
```
if (GPR(RS) <= 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BGTZ: Branch if greater than zero.
- Opcode: 0x08
- Format: BGTZ rs,offset
- Semantics:
```
if (GPR(RS) > 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BLTZ: Branch if less than zero.
- Opcode: 0x09
- Format: BLTZ rs,offset
- Semantics:
```
if (GPR(RS) < 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BGEZ: Branch if greater than or equal to zero.
- Opcode: 0x0a
- Format: BGEZ rs,offset
- Semantics:
```
if (GPR(RS) >= 0)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BC1F: Branch on floating point compare false.

Operator/operand	Semantics
FS	same as field RS
FT	same as field RT
FD	same as field RD
UIMM	IMM field unsigned-extended to word value
IMM	IMM field sign-extended to word value
OFFSET	IMM field sign-extended to word value
CPC	PC value of executing instruction
NPC	next PC value
SET_NPC(V)	Set next PC to value V
GPR(N)	General purpose register N
SET_GPR(N,V)	Set general purpose register N to value V
FPR_F(N)	Floating point register N single-precision value
SET_FPR_F(N,V)	Set floating point register N to single-precision value V
FPR_D(N)	Floating point register N double-precision value
SET_FPR_D(N,V)	Set floating point register N to double-precision value V
FPR_L(N)	Floating point register N literal word value
SET_FPR_L(N,V)	Set floating point register N to literal word value V
HI	High result register value
SET_HI(V)	Set high result register to value V
LO	Low result register value
SET_LO(V)	Set low result register to value V
READ_SIGNED_BYTE(A)	Read signed byte from address A
READ_UNSIGNED_BYTE(A)	Read unsigned byte from address A
WRITE_BYTE(V,A)	Write byte value V at address A
READ_SIGNED_HALF(A)	Read signed half from address A
READ_UNSIGNED_HALF(A)	Read unsigned half from address A
WRITE_HALF(V,A)	Write half value V at address A
READ_WORD(A)	Read word from address A
WRITE_WORD(V,A)	Write word value V at address A
TALIGN(T)	Check target T is aligned to 8 byte boundary
FPALIGN(N)	Check register N is wholly divisible by 2
OVER(X,Y)	Check for overflow when adding X to Y
UNDER(X,Y)	Check for overflow when subtraction Y from X
DIV0(V)	Check for division by zero error with divisor V

Table 3: Operator/operand semantics

Opcode: 0x0b
Format: BC1F offset
Semantics:
```
if (!FCC)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

BC1T: Branch on floating point compare true.
Opcode: 0x0c
Format: BC1T offset
Semantics:
```
if (FCC)
  SET_NPC(CPC + 8 + (OFFSET << 2))
else
  SET_NPC(CPC + 8)
```

A.2 Load/store instructions

LB: Load byte signed, displaced addressing.
Opcode: 0x20
Format: LB rt,offset(rs) inc_dec
Semantics: SET_GPR(RT,READ_SIGNED_BYTE(GPR(RS) + OFFSET))

LB: Load byte signed, indexed addressing.
Opcode: 0xc0
Format: LB rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_SIGNED_BYTE(GPR(RS)+GPR(RD)))

LBU: Load byte unsigned, displaced addressing.
Opcode: 0x22
Format: LBU rt,offset(rs) inc_dec
Semantics: SET_GPR(RT, READ_UNSIGNED_BYTE(GPR(RS)+OFFSET))

LBU: Load byte unsigned, indexed addressing.
Opcode: 0xc1
Format: LBU rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_UNSIGNED_BYTE(GPR(RS)+GPR(RD)))

LH: Load half signed, displaced addressing.
Opcode: 0x24
Format: LH rt,offset(rs) inc_dec
Semantics: SET_GPR(RT, READ_SIGNED_HALF(GPR(RS)+OFFSET))

LH: Load half signed, indexed addressing.
Opcode: 0xc2

Format: LH rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_SIGNED_HALF(GPR(RS)+GPR(RD)))

LHU: Load half unsigned, displaced addressing.
Opcode: 0x26
Format: LHU rt,offset(rs) inc_dec
Semantics: SET_GPR(RT, READ_UNSIGNED_HALF(GPR(RS)+OFFSET))

LHU: Load half unsigned, indexed addressing.
Opcode: 0xc3
Format: LHU rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_UNSIGNED_HALF(GPR(RS)+GPR(RD)))

LW: Load word, displaced addressing.
Opcode: 0x28
Format: LW rt,offset(rs) inc_dec
Semantics: SET_GPR(RT, READ_WORD(GPR(RS)+OFFSET))

LW: Load word, indexed addressing.
Opcode: 0xc4
Format: LW rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_WORD(GPR(RS)+GPR(RD)))

DLW: Double load word, displaced addressing.
Opcode: 0x29
Format: DLW rt,offset(rs) inc_dec
Semantics: SET_GPR(RT, READ_WORD(GPR(RS)+OFFSET))
SET_GPR(RT+1, READ_WORD(GPR(RS)+OFFSET+4))

DLW: Double load word, indexed addressing.
Opcode: 0xce
Format: DLW rt,(rs+rd) inc_dec
Semantics: SET_GPR(RT, READ_WORD(GPR(RS)+GPR(RD)))
SET_GPR(RT+1, READ_WORD(GPR(RS)+GPR(RD)+4))

L.S: Load word into floating point register file, displaced addressing.
Opcode: 0x2a
Format: L.S ft,offset(rs) inc_dec
Semantics: SET_FPR_L(FT, READ_WORD(GPR(RS)+OFFSET))

L.S: Load word into floating point register file, indexed addressing.
Opcode: 0xc5
Format: L.S ft,(rs+rd) inc_dec
Semantics: SET_FPR_L(RT, READ_WORD(GPR(RS)+GPR(RD)))

L.D: Load double word into floating point register file, displaced addressing.
Opcode: 0x2b
Format: L.D ft,offset(rs) inc_dec
Semantics: SET_FPR_L(FT, READ_WORD(GPR(RS)+OFFSET))
SET_FPR_L(FT+1, READ_WORD(GPR(RS)+OFFSET+4))

L.D: Load double word into floating point register file, indexed addressing.
Opcode: 0xcf
Format: L.D ft,(rs+rd) inc_dec
Semantics: SET_FPR_L(RT, READ_WORD(GPR(RS)+GPR(RD)))
SET_FPR_L(RT+1, READ_WORD(GPR(RS)+GPR(RD)+4))

LWL: Load word left, displaced addressing.
Opcode: 0x2c
Format: LWL offset(rs)
Semantics: See `ss.def` for a detailed description of this instruction's semantics. NOTE: LWL does not support pre-/post- inc/dec.

LWR: Load word right, displaced addressing.
Opcode: 0x2d
Format: LWR offset(rs)
Semantics: See `ss.def` for a detailed description of this instruction's semantics. NOTE: LWR does not support pre-/post- inc/dec.

SB: Store byte, displaced addressing.
Opcode: 0x30
Format: SB rt,offset(rs) inc_dec
Semantics: WRITE_BYTE(GPR(RT), GPR(RS)+OFFSET)

SB: Store byte, indexed addressing.
Opcode: 0xc6
Format: SB rt,(rs+rd) inc_dec
Semantics: WRITE_BYTE(GPR(RT), GPR(RS)+GPR(RD))

SH: Store half, displaced addressing.
Opcode: 0x32
Format: SH rt,offset(rs) inc_dec
Semantics: WRITE_HALF(GPR(RT), GPR(RS)+OFFSET)

SH: Store half, indexed addressing.
Opcode: 0xc7
Format: SH rt,(rs+rd) inc_dec
Semantics: WRITE_HALF(GPR(RT), GPR(RS)+GPR(RD))

SW: Store word, displaced addressing.
Opcode: 0x34
Format: SW rt,offset(rs) inc_dec
Semantics: WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)

SW: Store word, indexed addressing.
Opcode: 0xc8
Format: SW rt,(rs+rd) inc_dec
Semantics: WRITE_WORD(GPR(RT), GPR(RS)+GPR(RD))

DSW: Double store word, displaced addressing.
Opcode: 0x35
Format: DSW rt,offset(rs) inc_dec
Semantics: WRITE_WORD(GPR(RT), GPR(RS)+OFFSET)

WRITE_WORD(GPR(RT+1), GPR(RS)+OFF-SET+4)

DSW: Double store word, indexed addressing.
Opcode: 0xd0
Format: DSW rt,(rs+rd) inc_dec
Semantics: WRITE_WORD(GPR(RT), GPR(RS)+GPR(RD))
WRITE_WORD(GPR(RT+1), GPR(RS)+GPR(RD)+4)

DSZ: Double store zero, displaced addressing.
Opcode: 0x38
Format: DSW rt,offset(rs) inc_dec
Semantics: WRITE_WORD(0, GPR(RS)+OFFSET)
WRITE_WORD(0, GPR(RS)+OFFSET+4)

DSZ: Double store zero, indexed addressing.
Opcode: 0xd1
Format: DSW rt,(rs+rd) inc_dec
Semantics: WRITE_WORD(0, GPR(RS)+GPR(RD))
WRITE_WORD(0, GPR(RS)+GPR(RD)+4)

S.S: Store word from floating point register file, displaced addressing.
Opcode: 0x36
Format: S.S ft,offset(rs) inc_dec
Semantics: WRITE_WORD(FPR_L(FT), GPR(RS)+OFF-SET)

S.S: Store word from floating point register file, indexed addressing.
Opcode: 0xc9
Format: S.S ft,(rs+rd) inc_dec
Semantics: WRITE_WORD(FPR_L(FT), GPR(RS)+GPR(RD))

S.D: Store double word from floating point register file, displaced addressing.
Opcode: 0x37
Format: S.D ft,offset(rs) inc_dec
Semantics: WRITE_WORD(FPR_L(FT), GPR(RS)+OFF-SET)
WRITE_WORD(FPR_L(FT+1), GPR(RS)+OFF-SET+4)

S.D: Store double word from floating point register file, indexed addressing.
Opcode: 0xd2
Format: S.D ft,(rs+rd) inc_dec
Semantics: WRITE_WORD(FPR_L(FT), GPR(RS)+GPR(RD))
WRITE_WORD(FPR_L(FT+1), GPR(RS)+GPR(RD)+4)

SWL: Store word left, displaced addressing.
Opcode: 0x39
Format: SWL rt,offset(rs)
Semantics: See `ss.def` for a detailed description of this instruction's semantics. NOTE: SWL does not support pre-/post- inc/dec.

SWR: Store word right, displaced addressing.
Opcode: 0x3a
Format: SWR rt,offset(rs)
Semantics: See `ss.def` for a detailed description of this instruction's semantics. NOTE: SWR does not support pre-/post- inc/dec.

A.3 Integer instructions

ADD: Add signed (with overflow check).
Opcode: 0x40
Format: ADD rd,rs,rt
Semantics: OVER(GPR(RT),GPR(RT))
SET_GPR(RD, GPR(RS) + GPR(RT))

ADDI: Add immediate signed (with overflow check).
Opcode: 0x41
Format: ADDI rd,rs,rt
Semantics: OVER(GPR(RS),IMM)
SET_GPR(RT, GPR(RS) + IMM)

ADDU: Add unsigned (no overflow check).
Opcode: 0x42
Format: ADDU rd,rs,rt
Semantics: SET_GPR(RD, GPR(RS) + GPR(RT))

ADDIU: Add immediate unsigned (no overflow check).
Opcode: 0x43
Format: ADDIU rd,rs,rt
Semantics: SET_GPR(RT, GPR(RS) + IMM)

SUB: Subtract signed (with underflow check).
Opcode: 0x44
Format: SUB rd,rs,rt
Semantics: UNDER(GPR(RS),GPR(RT))
SET_GPR(RD, GPR(RS) - GPR(RT))

SUBU: Subtract unsigned (without underflow check).
Opcode: 0x45
Format: SUBU rd,rs,rt
Semantics: SET_GPR(RD, GPR(RS) - GPR(RT))

MULT: Multiply signed.
Opcode: 0x46
Format: MULT rs,rt
Semantics: SET_HI((RS * RT) / (1<<32))
SET_LO((RS * RT) % (1<<32))

MULTU: Multiply unsigned.
Opcode: 0x47
Format: MULTU rs,rt
Semantics: SET_HI(((unsigned)RS * (unsigned)RT)/(1<<32))
SET_LO(((unsigned)RS*(unsigned)RT) % (1<<32))

DIV: Divide signed.
Opcode: 0x48
Format: DIV rs,rt
Semantics: DIV0(GPR(RT))
SET_LO(GPR(RS) / GPR(RT))
SET_HI(GPR(RS) % GPR(RT))

DIVU Divide unsigned.
Opcode: 0x49
Format: DIVU rs,rt
Semantics: DIV0(GPR(RT))
SET_LO((unsigned)GPR(RS)/(unsigned)GPR(RT))
SET_HI((unsigned)GPR(RS)%(unsigned)GPR(RT))

MFHI: Move from HI register.
Opcode: 0x4a
Format: MFHI rd
Semantics: SET_GPR(RD, HI)

MTHI: Move to HI register.
Opcode: 0x4b
Format: MTHI rs
Semantics: SET_HI(GPR(RS))

MFLO: Move from LO register.
Opcode: 0x4c
Format: MFLO rd
Semantics: SET_GPR(RD, LO)

MTLO: Move to LO register.
Opcode: 0x4d
Format: MTLO rs
Semantics: SET_LO(GPR(RS))

AND: Logical AND.
Opcode: 0x4e
Format: AND rd,rs,rt
Semantics: SET_GPR(RD, GPR(RS) & GPR(RT))

ANDI: Logical AND immediate.
Opcode: 0x4f
Format: ANDI rd,rt,imm
Semantics: SET_GPR(RT, GPR(RS) & UIMM)

OR: Logical OR.
Opcode: 0x50
Format: OR rd,rs,rt
Semantics: SET_GPR(RD, GPR(RS) | GPR(RT))

ORI: Logical OR immediate.
Opcode: 0x51
Format: ORI rd,rt,imm
Semantics: SET_GPR(RT, GPR(RS) | UIMM)

XOR: Logical XOR.
Opcode: 0x52
Format: XOR rd,rs,rt
Semantics: SET_GPR(RD, GPR(RS) ^ GPR(RT))

XORI: Logical XOR immediate.
Opcode: 0x53
Format: ORI rd,rt,uimm
Semantics: SET_GPR(RT, GPR(RS) ^ UIMM)

NOR: Logical NOR.
Opcode: 0x54
Format: NOR rd,rs,rt
Semantics: SET_GPR(RD, ~(GPR(RS) | GPR(RT)))

SLL: Shift left logical.
Opcode: 0x55
Format: SLL rd,rt,shamt
Semantics: SET_GPR(RD, GPR(RT) << SHAMT)

SLLV: Shift left logical variable.
Opcode: 0x56
Format: SLLV rd,rt,rs
Semantics: SET_GPR(RD, GPR(RT) << (GPR(RS) & 0x1f))

SRL: Shift right logical.
Opcode: 0x57
Format: SRL rd,rt,shamt
Semantics: SET_GPR(RD, GPR(RT) >> SHAMT)

SRLV: Shift right logical variable.
Opcode: 0x58
Format: SRLV rd,rt,rs
Semantics: SET_GPR(RD, GPR(RT) << (GPR(RS) & 0x1f))

SRA: Shift right arithmetic.
Opcode: 0x59
Format: SRA rd,rt,shamt
Semantics: SET_GPR(RD, SEX(GPR(RT) >> SHAMT, 31 - SHAMT))

SRAV: Shift right arithmetic variable.
Opcode: 0x59
Format: SRAV rd,rt,rs
Semantics: SET_GPR(RD, SEX(GPR(RT) >> SHAMT, 31 - (GPR(RD) & 0x1f)))

SLT: Set register if less than.
Opcode: 0x5b
Format: SLT rd,rs,rt
Semantics: SET_GPR(RD, (GPR(RS) < GPR(RT)) ? 1 : 0)

SLTI: Set register if less than immediate.
Opcode: 0x5c
Format: SLTI rd,rs,imm
Semantics: SET_GPR(RD, (GPR(RS) < IMM) ? 1 : 0)

SLTU: Set register if less than unsigned.
Opcode: 0x5d
Format: SLTU rd,rs,rt
Semantics: SET_GPR(RD, ((unsigned)GPR(RS)<(unsigned)GPR(RT)) ? 1 : 0)

SLTIU: Set register if less than unsigned immediate.
Opcode: 0x5d
Format: SLTIU rd,rs,imm
Semantics: SET_GPR(RD, ((unsigned)GPR(RS)<(unsigned)GPR(RT)) ? 1 : 0)

A.4 Floating-point instructions

ADD.S: Add floating point, single precision.
Opcode: 0x70
Format: ADD.S fd,fs,ft
Semantics: FPALIGN(FD)

FPALIGN(FS)
FPALIGN(FT)
SET_FPR_F(FD, FPR_F(FS) + FPR_F(FT)))

ADD.D: Add floating point, double-precision.
Opcode: 0x71
Format: ADD.D fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_D(FD, FPR_D(FS) + FPR_D(FT)))

SUB.S: Subtract floating point, single precision.
Opcode: 0x72
Format: SUB.S fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_F(FD, FPR_F(FS) - FPR_F(FT)))

SUB.D: Subtract floating point, double precision.
Opcode: 0x73
Format: SUB.D fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_D(FD, FPR_D(FS) - FPR_D(FT)))

MUL.S: Multiply floating point, single precision.
Opcode: 0x74
Format: MUL.S fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_F(FD,FPR_F(FS)*FPR_F(FT)))

MUL.D: Multiply floating point, double precision.
Opcode: 0x75
Format: MUL.D fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
SET_FPR_D(FD, FPR_D(FS) * FPR_D(FT)))

DIV.S: Divide floating point, single precision.
Opcode: 0x76
Format: DIV.S fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
DIV0(FPR_F(FT))
SET_FPR_F(FD, FPR_F(FS) / FPR_F(FT)))

DIV.D: Divide floating point, double precision.
Opcode: 0x77
Format: DIV.D fd,fs,ft
Semantics: FPALIGN(FD)
FPALIGN(FS)
FPALIGN(FT)
DIV0(FPR_D(FT))
SET_FPR_D(FD, FPR_D(FS) / FPR_D(FT)))

ABS.S: Absolute value, single precision.
Opcode: 0x78
Format: ABS.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, fabs((double)FPR_F(FS))))

ABS.D: Absolute value, double precision.
Opcode: 0x79
Format: ABS.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, fabs(FPR_D(FS))))

MOV.S: Move floating point value, single precision.
Opcode: 0x7a
Format: MOV.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, FPR_F(FS))

MOV.D: Move floating point value, double precision.
Opcode: 0x7b
Format: MOV.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, FPR_D(FS))

NEG.S: Negate floating point value, single precision.
Opcode: 0x7c
Format: NEG.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, -FPR_F(FS))

NEG.D: Negate floating point value, double precision.
Opcode: 0x7d
Format: NEG.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, -FPR_D(FS))

CVT.S.D: Convert double precision to single precision.
Opcode: 0x80
Format: CVT.S.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, -FPR_D(FS))

CVT.S.W: Convert integer to single precision.
Opcode: 0x81
Format: CVT.S.W fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD, (float)FPR_L(FS))

CVT.D.S: Convert single precision to double precision.
Opcode: 0x82
Format: CVT.D.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD,(double)FPR_F(FS))

CVT.D.W: Convert integer to double precision.
Opcode: 0x83
Format: CVT.D.W fd,fs

Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD,(double)FPR_L(FS))

CVT.W.S: Convert single precision to integer.
Opcode: 0x84
Format: CVT.W.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_L(FD, (long)FPR_F(FS))

CVT.W.D: Convert double precision to integer.
Opcode: 0x85
Format: CVT.W.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_L(FD, (long)FPR_D(FS))

C.EQ.S: Test if equal, single precision.
Opcode: 0x90
Format: C.EQ.S fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_F(FS) == FPR_F(FT))

C.EQ.D: Test if equal, double precision.
Opcode: 0x91
Format: C.EQ.D fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_D(FS) == FPR_D(FT))

C.LT.S: Test if less than, single precision.
Opcode: 0x92
Format: C.LT.S fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_F(FS) < FPR_F(FT))

C.LT.D: Test if less than, double precision.
Opcode: 0x93
Format: C.LT.D fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_D(FS) < FPR_D(FT))

C.LE.S: Test if less than or equal, single precision.
Opcode: 0x94
Format: C.LE.S fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_F(FS) <= FPR_F(FT))

C.LE.D: Test if less than or equal, double precision.
Opcode: 0x95
Format: C.LE.D fs,ft
Semantics: FPALIGN(FS)
FPALIGN(FT)
SET_FCC(FPR_D(FS) <= FPR_D(FT))

SQRT.S: Square root, single precision.
Opcode: 0x96
Format: SQRT.S fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_F(FD,sqrt((double)FPR_F(FS)))

SQRT.D: Square root, double precision.
Opcode: 0x97
Format: SQRT.D fd,fs
Semantics: FPALIGN(FD)
FPALIGN(FS)
SET_FPR_D(FD, sqrt(FPR_D(FS)))

A.5 Miscellaneous instructions

NOP: No operation.
Opcode: 0x00
Format: NOP
Semantics: None

SYSCALL: System call.
Opcode: 0xa0
Format: SYSCALL
Semantics: See Appendix B for details

BREAK: Declare a program error.
Opcode: 0xa1
Format: BREAK uimm
Semantics: Actions are simulator-dependent. Typically, an error message is printed and `abort()` is called.

LUI: Load upper immediate.
Opcode: 0xa2
Format: LUI uimm
Semantics: SET_GPR(RT, UIMM << 16)

MFC1: Move from floating point to integer register file.
Opcode: 0xa3
Format: MFC1 rt,fs
Semantics: SET_GPR(RT, FPR_L(FS))

MTC1: Move from integer to floating point register file.
Opcode: 0xa5
Format: MTC1 rt,fs
Semantics: SET_FPR_L(FS, GPR(RT))

B System call definitions

This appendix lists all system calls supported by the simulators with their system call code (syscode), interface specification, and appropriate POSIX Unix reference. Systems calls are initiated with the SYSCALL instruction. Prior to execution of a SYSCALL instruction, register $*v*0 should be loaded with the system call code. The arguments of the system call interface prototype should be loaded into registers $*a*0 - $*a*3 in the order specified by the system call interface prototype, *e.g.*, for:

```
read(int fd, char *buf, int nbyte),
```

0x03 is loaded into $*v*0, fd is loaded into $*a*0, buf into $*a*1, and nbyte into $*a*2.

EXIT: Exit process.

Syscode: 0x01
Interface: void exit(int status);
Semantics: See `exit(2)`.

READ: Read from file to buffer.
Syscode: 0x03
Interface: int read(int fd, char *buf, int nbyte);
Semantics: See `read(2)`.

WRITE: Write from a buffer to a file.
Syscode: 0x04
Interface: int write(int fd, char *buf, int nbyte);
Semantics: See `write(2)`.

OPEN: Open a file.
Syscode: 0x05
Interface: int open(char *fname, int flags, int mode);
Semantics: See `open(2)`.

CLOSE: Close a file.
Syscode: 0x06
Interface: int close(int fd);
Semantics: See `close(2)`.

CREAT: Create a file.
Syscode: 0x08
Interface: int creat(char *fname, int mode);
Semantics: See `creat(2)`.

UNLINK: Delete a file.
Syscode: 0x0a
Interface: int unlink(char *fname);
Semantics: See `unlink(2)`.

CHDIR: Change process directory.
Syscode: 0x0c
Interface: int chdir(char *path);
Semantics: See `chdir(2)`.

CHMOD: Change file permissions.
Syscode: 0x0f
Interface: int chmod(int *fname, int mode);
Semantics: See `chmod(2)`.

CHOWN: Change file owner and group.
Syscode: 0x10
Interface: int chown(char *fname, int owner, int group);
Semantics: See `chown(2)`.

BRK: Change process break address.
Syscode: 0x11
Interface: int brk(long addr);
Semantics: See `brk(2)`.

LSEEK: Move file pointer.
Syscode: 0x13
Interface: long lseek(int fd, long offset, int whence);
Semantics: See `lseek(2)`.

GETPID: Get process identifier.
Syscode: 0x14
Interface: int getpid(void);
Semantics: See `getpid(2)`.

GETUID: Get user identifier.
Syscode: 0x18
Interface: int getuid(void);
Semantics: See `getuid(2)`.

ACCESS: Determine accessibility of a file.
Syscode: 0x21
Interface: int access(char *fname, int mode);
Semantics: See `access(2)`.

STAT: Get file status.
Syscode: 0x26
Interface:
```
struct stat
{
  short          st_dev;
  long           st_ino;
  unsigned short st_mode;
  short          st_nlink;
  short          st_uid;
  short          st_gid;
  short          st_rdev;
  int            st_size;
  int            st_atime;
  int            st_spare1;
  int            st_mtime;
  int            st_spare2;
  int            st_ctime;
  int            st_spare3;
  long           st_blksize;
  long           st_blocks;
  long           st_gennum;
  long           st_spare4;
};
int stat(char *fname, struct stat *buf);
```
Semantics: See `stat(2)`.

LSTAT: Get file status (and don't dereference links).
Syscode: 0x28
Interface: int lstat(char *fname, struct stat *buf);
Semantics: See `lstat(2)`.

DUP: Duplicate a file descriptor.
Syscode: 0x29
Interface: int dup(int fd);
Semantics: See `dup(2)`.

PIPE: Create an interprocess comm. channel.
Syscode: 0x2a
Interface: int pipe(int fd[2]);
Semantics: See `pipe(2)`.

GETGID: Get group identifier.
Syscode: 0x2f
Interface: int getgid(void);
Semantics: See `getgid(2)`.

IOCTL: Device control interface.
Syscode: 0x36
Interface: int ioctl(int fd, int request, char *arg);
Semantics: See `ioctl(2)`.

FSTAT: Get file descriptor status.
Syscode: 0x3e
Interface: int fstat(int fd, struct stat *buf);
Semantics: See `fstat(2)`.

GETPAGESIZE: Get page size.
Syscode: 0x40
Interface: int getpagesize(void);
Semantics: See `getpagesize(2)`.

GETDTABLESIZE: Get file descriptor table size.
Syscode: 0x59
Interface: int getdtablesize(void);
Semantics: See `getdtablesize(2)`.

DUP2: Duplicate a file descriptor.
Syscode: 0x5a
Interface: int dup2(int fd1, int fd2);
Semantics: See `dup2(2)`.

FCNTL: File control.
Syscode: 0x5c
Interface: int fcntl(int fd, int cmd, int arg);
Semantics: See `fcntl(2)`.

SELECT: Synchronous I/O multiplexing.
Syscode: 0x5d
Interface: int select (int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
Semantics: See `select(2)`.

GETTIMEOFDAY: Get the date and time.
Syscode: 0x74
Interface:
```
struct timeval {
  long tv_sec;
  long tv_usec;
};
struct int {
  timezone tz_minuteswest;
  int tz_dsttime;
};
int gettimeofday(struct timeval *tp,
  struct timezone *tzp);
```
Semantics: See `gettimeofday(2)`.

WRITEV: Write output, vectored.
Syscode: 0x79
Interface: int writev(int fd, struct iovec *iov, int cnt);
Semantics: See `writev(2)`.

UTIMES: Set file times.
Syscode: 0x8a
Interface: int utimes(char *file, struct timeval *tvp);
Semantics: See `utimes(2)`.

GETRLIMIT: Get maximum resource consumption.
Syscode: 0x90
Interface: int getrlimit(int res, struct rlimit *rlp);
Semantics: See `getrlimit(2)`.

SETRLIMIT: Set maximum resource consumption.
Syscode: 0x91
Interface: int setrlimit(int res, struct rlimit *rlp);
Semantics: See `setrlimit(2)`.