Evaluating Future Microprocessors: the SimpleScalar Tool Set

Doug Burger*

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

Todd M. Austin† and Steve Bennett‡

†MicroComputer Research Labs, JF3-359
‡Measurement Architecture Planning, JF1-91
Intel Corporation, 2111 NE 25th Avenue
Hillsboro, OR 97124 USA

*Contact: dburger@cs.wisc.edu
http://www.cs.wisc.edu/~mscalar/simplescalar.html

Abstract[1]

This document describes the SimpleScalar tool set, a collection of publicly-available simulation tools that use detailed execution-driven to simulate modern processor architectures. In this report, we give an overview of the tool set, show how to obtain, install and use it. We also discuss details about the tools' internals, and document the SimpleScalar architecture.

1 Overview

Modern processors are extremely complex pieces of engineering. Researching aspects of processor and system design with these complicated beasts requires excellent simulation tools. Prototyping processors in hardware is expensive and time-consuming, particularly during the initial phases of a research project. Mathematical performance models of current-generation processors can be inaccurate, given the aggressive use of caches, out-of-order execution, and speculation in these processors.

However, the researcher does not always want to simulate at the same level of detail. Initial studies, or isolation of one component of the system, require a fast simulator that abstracts away unnecessary detail. Measuring the effect of a design change on overall processor or system performance requires a detailed simulator that captures the interactions of all the different processor components. Furthermore, the pace of processor improvements mandates a simulation environment that is easily extensible and flexible—lest the intrepid graduate student finish coding a simulator only to find it out of date.

The SimpleScalar tool set, documented in this report, addresses the above concerns. The tool set provides a GCC-based compiler and associated utilities that produce object code targeted toward the SimpleScalar architecture, which is itself a derivation of the MIPS architecture [1].

The advantages of the SimpleScalar tools are high flexibility, portability, extensibility, and performance. Their flexibility is demonstrated by the inclusion of five processor simulators in the release. These five are execution-driven processor simulators for the SimpleScalar architecture, which range from an extremely fast functional simulator to a detailed out-of-order issue 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 both Sparc SunOS and Solaris platforms. The tool set is easily extensible—due mostly to the way in which we define the instruction set. 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 easier to extend. Finally, the simulators have been aggressively tuned for performance, and can run codes approaching "real" sizes in tractable amounts of time.

In addition to the tools based on the SimpleScalar architecture, we also provide a tool that uses the SimpleScalar design philosophy to simulate binaries compiled for a target Linux/x86 system. This tool, called SimpleScalar x86, currently runs only on a Sparc SunOS host, but is not prohibitively difficult to port to other platforms (in particular, a port to a Linux/x86 host would be trivial).

The rest of this document contains information about obtaining, installing, running, using, and modifying the simulators. In Section 2 we provide a detailed procedure for downloading the release, installing it, and getting it up and running. We provide such instructions for both the main SimpleScalar release and SimpleScalar x86. In Section 3, we discuss the SimpleScalar architecture itself in detail. In Section 4, we discuss the internal details of the SimpleScalar processor simulators. In Section 5, we discuss some details about the SimpleScalar x86 internals. In Section 6, we provide the history of the tools' development and conclude. Appendices A and B contain complete definitions of the SimpleScalar instruction set and system calls, respectively.

2 Installation and Use

The only restrictions on using and distributing the tool set are that (1) the copyright notice must accompany all re-releases of the tool set, and (2) third parties (i.e., you) are forbidden to place any additional distribution restrictions on extensions to the tool set that you release. The copyright notice can be found in the distribution directory as well as at the head of all simulator source files. We have included the copyright here as well:

1. This research has been supported by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, a donation from Intel Corp., and by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346

- *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.*

In other words, you are welcome to use, share and improve these tools. You are forbidden to forbid anyone else to use, share and improve what you give them.

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 "release.tar.gz" via the WWW, enter the URL:

```
ftp://ftp.cs.wisc.edu/sohi/Code/simplescalar/
    release.tar
```

and to obtain the same file with traditional ftp:

```
ftp ftp.cs.wisc.edu
user: anonymous
password: enter your e-mail address here
cd sohi/Code/simplescalar
get release.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 two distribution files in the directory are:

- **release.tar.gz** - contains the full-blown SimpleScalar release. This file contains the GCC source, utilities, simulator sources, essentially everything you will need to port the tool suite to your system. This file is quite large—63 MB uncompressed.
- **x86.tar.gz** - contains the SimpleScalar x86 tool set, for simulating x86 Linux binaries. Includes the x86 simulator, but does not include the GNU tools or Linux sources. Fully installed (with GNU and Linux), it requires 170 MB (not including tar files). Simplescalar x86 currently runs only under SunOS, although ports to other systems are not be prohibitively hard.

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

```
tar xf filename.tar
```

If you are downloading the full release, you will have the following subdirectories, which have the following contents:

- **simplesim-0.1** - holds code for five SimpleScalar processor simulators and all supporting code files.
- **gcc-2.6.3** - holds the GNU C compiler code, targeted toward the SimpleScalar architecture.
- **binutils-2.5.2** - contains the GNU binary utilities code, ported to the SimpleScalar architecture
- **glibc-1.09** - contains the GNU libraries code, ported to the SimpleScalar architecture.
- **f2c-1994.09.27** - contains the 1994 release of AT&T Bell Labs' FORTRAN to C translator code.
- **test-progs** - contains a battery of benchmarks that can be used to test the simulators
- **ss-bootstrap** - target directory for the ported cross-compiler, compiled GNU binary utilities, and libraries that are targeted to the SimpleScalar architecture.
- **bin**, **include**, **info**, **lib**, **man** - target directories where the compiled GNU tools and support files will be installed.

The SimpleScalar x86 release contains the code files for the simulator and two subdirectories, **include** and **tests**. See Section 2.3 for a description of how to obtain the other files needed for running SimpleScalar x86.

2.2 Installing and running Simplescalar

We depict a graphical overview of the tool set[1] 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 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).

The SimpleScalar architecture, like the MIPS architecture [1], 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 endianness that matches your host platform; the simulators will generate numerous warnings and 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-0.1 directory. The following instructions will assume a big-endian installation for simplicity.

To install the full release, first make a symbolic link to whichever target architecture you desire in the **ss-bootstrap** directory: $IDIR will represent the directory in which you are installing the tools.

```
cd $IDIR
ln -s ss-bootstrap/ssbig-na-sstrix
```

Next, build the GNU binary utilities[2]:

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

1. Figure 1 applies to the main SimpleScalar release but not SimpleScalar x86

2. 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.

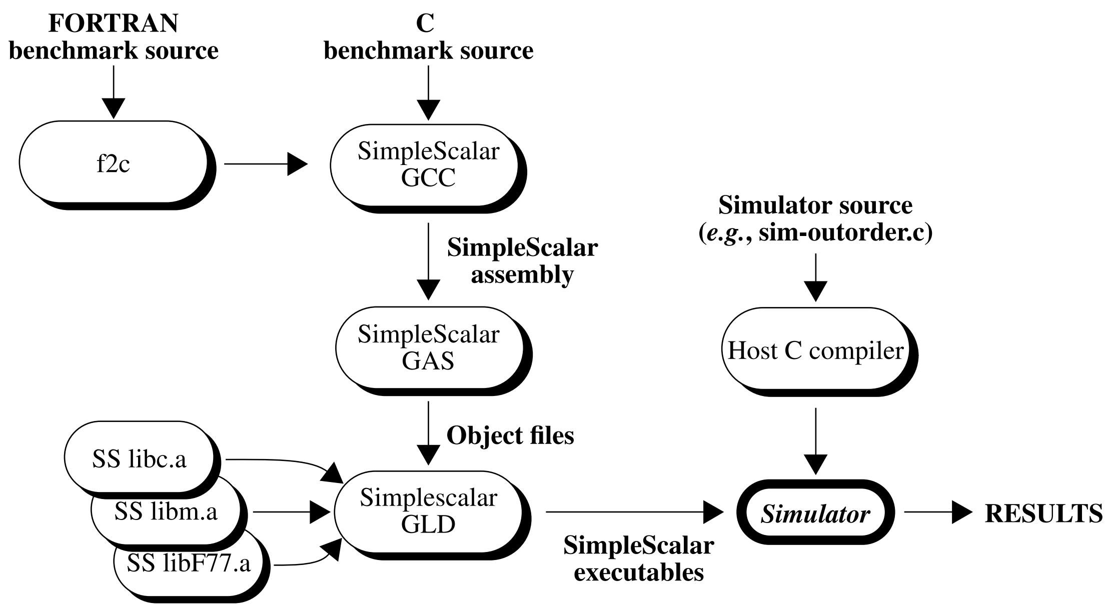

Figure 1. SimpleScalar tool set overview

```
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.

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-0.1/Makefile to use the desired compile flags (e.g., the correct optimization level) To build the simulators:

```
cd $IDIR/simplesim-0.1
make
```

Now, build the compiler itself:

```
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-0.1/sim-safe ./enquire -f >!
   float.h-cross
make install
```

We provide pre-built copies of the necessary libraries in ss-bootstrap/ssbig-na-sstrix/lib, so you do not need to build the code in **glibc-1.09**, unless you change the library code. In that event, to build the libraries:

```
cd $IDIR/glibc-1.09
configure --prefix=$IDIR/ssbig-na-sstrix ssbig-
     na-sstrix
setenv CC $IDIR/bin/ssbig-na-sstrix-gcc
unsetenv TZ
unsetenv MACHINE
make
make install
```

Note that you must have already installed the SimpleScalar tools 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 the f2c tool:

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

Finally, build the test benchmarks:

```
cd $IDIR/test-progs
make
```

The tool set should now be ready for use. To run a test:

```
cd $IDIR/simplesim-0.1
sim-inorder ../test_progs/test-math
```

The test should generate about a page of output, and will run very quickly.

2.3 Installing Simplescalar x86

Decide where to install the GNU tools for x86, the x86 simulator and the Linux sources. The tools require about 170MB of space when installed and built (discounting the tar files). Set the environment variables GNUROOT, X86ROOT and LNXROOT to be the above three directories, respectively. Then create the directory structures needed for installation:

```
mkdir $GNUROOT/src
mkdir $GNUROOT/i486-linux
mkdir $GNUROOT/i486-linux/bin
mkdir $GNUROOT/i486-linux/lib
mkdir $GNUROOT/i486-linux/include
mkdir $GNUROOT/bin
mkdir $GNUROOT/lib
mkdir $GNUROOT/lib/gcc-lib
mkdir $GNUROOT/lib/gcc-lib/i486-linux
mkdir $GNUROOT/lib/gcc-lib/i486-linux/2.7.2
mkdir $LNXROOT/src
```

Place **x86.tar** in $X86ROOT, and obtain the GNU and Linux sources from one of the many gnu/linux software mirrors[1]. Download the following files from the FTP site (the paths to the files may be different if you use a different software mirror):

```
systems/linux/sunsite/GCC/binutils-
    2.6.0.14.tar.gz
systems/gnu/gcc-2.7.2.tar.gz
systems/linux/sunsite/GCC/libc-5.0.9.tar.gz
systems/linux/sunsite/kernel/v1.2/linux-
    1.2.13.tar.gz
systems/linux/sunsite/GCC/libc-5.0.9.bin.tar.gz
```

Now move the files from their downloaded directory to their correct directories:

```
cp binutils-2.6.0.14.tar.gz $GNUROOT/src
cp gcc-2.7.2.tar.gz $GNUROOT/src
cp libc-5.0.9.tar.gz $LNXROOT/src
cp linux-1.2.13.tar.gz $LNXROOT
cp libc-5.0.9.bin.tar.gz $LNXROOT
```

Unpack the downloaded files:

```
cd $LNXROOT
gunzip linux-1.2.13.tar.gz
tar xf linux-1.2.13.tar
gunzip libc-5.0.9.bin.tar
tar xf libc-5.0.9.bin.tar
cd src
gunzip libc-5.0.9.tar.gz
tar xf libc-5.0.9.tar
cd $GNUROOT/src
gunzip binutils-2.6.0.14.tar.gz
tar xf binutils-2.6.0.14.tar
gunzip gcc.2.7.2.tar.gz
tar xf gcc-2.7.2.tar
cd $X86ROOT
tar xf x86.tar
```

You may want to remove the tar files at this point; they occupy a great deal of disk space and are not needed after this step.

The version.h include file is built when the Linux kernel is compiled. Since we are not compiling the kernel here, we need to fake this include file.

```
cp $X86ROOT/include/version.h $LNXROOT/linux/
    include/linux
```

2.3.1 Build binutils:

Build the various utility packages for use with gcc, including the GNU loader and assembler[2].

```
cd $GNUROOT/src/binutils-2.6.0.14
configure --target=i486-linux
cd bfd
make CC=gcc headers
cd ..
make CC=gcc
```

Now create links to the utilities where gcc and make files will look for them:

```
cd $GNUROOT/i486-linux/bin
setenv $BINROOT $GNUROOT/src/binutils-2.6.0.14
ln -s $BINROOT/binutils/objdump objdump
ln -s $BINROOT/binutils/size    size
ln -s $BINROOT/binutils/nm.new  nm
ln -s $BINROOT/binutils/ar      ar
ln -s $BINROOT/binutils/ranlib  ranlib
ln -s $BINROOT/ld/ld.new        ld
ln -s $BINROOT/gas/as.new       as
cd $GNUROOT/bin
ln -s $BINROOT/binutils/objdump objdump-i486
ln -s $BINROOT/binutils/size    size-i486
ln -s $BINROOT/binutils/nm.new  nm-i486
ln -s $BINROOT/binutils/ar      ar-i486
ln -s $BINROOT/binutils/ranlib  ranlib-i486
ln -s $BINROOT/ld/ld.new        ld-i486
ln -s $BINROOT/gas/as.new       as-i486
```

2.3.2 Build gcc-2.7.2

Now build gcc itself. Copy the include files into $GNUROOT/i486-linux/include

```
cd $GNUROOT/i486-linux/include
cp -R $LNXROOT/linux/include/linux .
cp -R $LNXROOT/linux/include/asm-i386 .
ln -s asm-i386 asm
cp -R $LNXROOT/usr/include/* .
cp $X86ROOT/include/float.h .
```

Set up library and include links:

```
cd $GNUROOT/lib/gcc-lib/i486-linux/2.7.2
ln -s $GNUROOT/i486-linux/include include
ln -s $GNUROOT/i486-linux/lib     lib
```

Configure the gcc Makefile:

```
cd $GNUROOT/src/gcc-2.7.2
configure --target=i486-linux --with-gnu-as --
    with-gnu-ld --prefix=$GNUROOT
```

Edit the gcc Makefile to use gcc to build the cross compiler:

```
change "CC = cc" to "CC = gcc"
```

Edit the gcc Makefile to avoid squashing the float.h include file by commenting out these three lines:

```
# rm -f include/float.h
# cp gfloat.h include/float.h
# chmod a+r include/float.h
```

Touch the following library files to prevent gcc from trying to create them (and thus failing):

```
touch libgcc.cross libgcc1.a
```

Build and install gcc.

```
make LANGUAGES=c
make LANGUAGES=c install
```

The compilation should end with output that looks approximately like this:

```
[snip]
gcc -DCROSS_COMPILE -DIN_GCC   -g   obstack.o ` case
    "gcc" in "cc") echo "" ;; esac `   -o c++filt \
  cxxmain.o underscore.o getopt.o getopt1.o
```

When this is done, there should be an executable version of gcc in the $GNUROOT/bin/i486-linux directory. Add this directory into your search path. All the x86 tools are here.

Verify that $GNUROOT/bin/i486-linux/gcc is executable. We experienced problems with a version of 'install' on the suns. If you run into problems during the "make install" step, copy the gcc-cross driver (or xgcc) to $GNUROOT/i486-linux/bin/gcc and $GNUROOT/bin/i486-linux-gcc. Some versions of install put a copy of the driver into directories of the same name. You may try

1. We used ftp://wuarchive.wustl.edu; you can finger fsf@prep.ai.mit.edu for more information.

2. You must also use GNU make here to build these files.

the following steps if this problem arises:

```
cd $GNUROOT/i486-linux/bin
mv gcc gcc.install
ln -s gcc.install/gcc-cross gcc
cd $GNUROOT/bin
mv i486-linux-gcc i486-linux-gcc.install
ln -s i486-linux-gcc.install/gcc-cross i486-
    linux-gcc
```

2.3.3 Build GNU libc

Run the configure program.

```
cd $LNXROOT/src/libc
configure
```

Give the configure program the following information (NOTE: GNUROOT is the actual path without a trailing '/', *not* an environment variable):

```
Build 386, 486 or m68k library code (486
    default) 4/3/m [4] ? 4
The target platform [i486-linux] ? i486-linux
The target OS [linux] ?           linux
Build targets (static/shared) s/a [a] ?s
Root path to i486-linux related files [] ?
    GNUROOT
Bin path to gcc [] ?GNUROOT/i486-linux/bin
The gcc version [2.6.2] ?         2.7.2
Fast build/save space (fast default) f/s [f] ?f
GNU `make' executable [gmake] ?   make
Root path to installation dirs ?  GNUROOT/test
Build a NYS libc from nys y/n [n] ?n
```

Build the libraries:

```
make clean
make depend
unsetenv MACHINE
make
```

Copy the libraries to the Linux library directory:

```
cd $GNUROOT/i486-linux/lib
cp $LNXROOT/libc/elfstatic/lib*.a .
cp $LNXROOT/libc/elfshared/crt* .
```

2.3.4 Building and testing SimpleScalar x86

Finally, we are ready to build the simulator itself:

```
cd $X86SIMROOT/xsim
```

If you have set the GNUROOT environment variable as described above, no modifications of the Makefile are necessary. Otherwise, you must modify the Makefile so that BINUTILDIR points to $GNUROOT/src/binutils-2.6.0.14. Now build the simulator:

```
make depend
make sim-func
```

Build the test executables:

```
cd $X86ROOT/tests
make sun
make intel
```

To run a sample test program:.

```
cd $X86ROOT
sim-func -Wsim tests/hello-i486
```

The result should print "hello world", surrounded by simulator comments.

3 The Simplescalar architecture

The SimpleScalar architecture is derived from the MIPS-IV ISA [1]. The semantics are a superset 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—additional 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

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 or each. Both the number and the semantics of the registers are identical to those in the MIPS-IV ISA.

In Figure 2, 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        $4,4($5)
```

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)   $4,4($5)
```

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

To measure instruction cache performance with architectures that have 32-bit instruction formats, the simulators may be run with instruction cache blocks twice as large as the blocks on the 32-bit target machine. This trick will produce statistics that are consistent with a target that uses a 32-bit instruction encoding. Since the unextended SimpleScalar architecture may be encoded into a 32-bit instruction, doubling the cache block size yields a valid result. The two timing simulators (sim-inorder and sim-outorder, discussed in the next section) currently automatically double instruction cache block sizes. The cache module will need to be changed for simulation of unified caches or accurate bus contention on target machines with 32-bit instructions, however.

4 Tool internals

In this section we discuss the code files of the simulators provided with the release. Each simulator has one main code file, and shares all other support files with the other simulators. Section 4.1

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

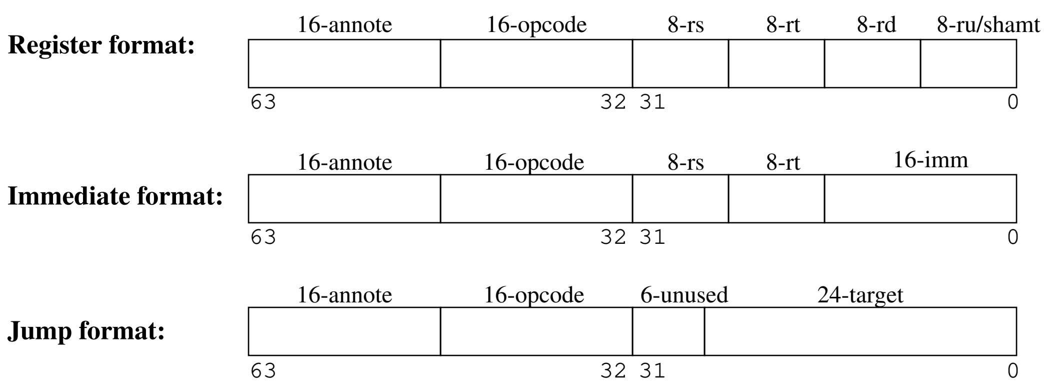

Figure 2. SimpleScalar architecture instruction formats

through Section 4.4 contain descriptions of the simulator files, from the fastest and least detailed to the slowest and most detailed.

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. Plans exist at Wisconsin to eventually extend the tool set for simulation of kernel code.

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()` pre-decodes the entire text segment for faster simulation, and then begins simulation from the target program entry point.

4.1 Functional simulation

The fastest, least detailed simulator (**sim-fast**) resides in sim-fast.c. sim-fast does no time accounting; it executes each instruction serially, performing no instructions in parallel. sim-fast assumes no cache.

A separate version of sim-fast, **sim-safe**, also performs functional simulation, but checks for correct alignment and access permissions for each memory reference. The two (safe and unchecked memory references) simulators are split (e.g., protection is not toggled with a command-line argument) to maximize performance. Neither of the simulators accept any command-line arguments at all. 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**.

4.2 Fast functional simulation with cache

The sim-cache simulator (the main file of which is **sim-cache.c**) takes the fast functional simulation (with unchecked memory accesses) and adds the capability to simulate one level of cache and/or a TLB. The cache code is located in **cache.c**. The simulator supports simulation of split level-one instruction and data caches, or just an instruction or just a data cache, but not a unified I/D level-one cache (although this is a trivial change to the simulator). The command-line arguments that it accepts are:

-d*name:sets:blocksize:assoc:repl* - Simulate a level-one data cache, called *name* in the statistics file, with *sets* number of sets, blocks of *blocksize* bytes, *assoc* set-associativity, and a replacement policy of *repl*, where *repl* is either *l*, *r*, or *f* (for LRU, random, and FIFO, respectively). The cache size will be $\text{sets} \times \text{blocksize} \times \text{assoc}$ bytes in size. A two-way set associative, 64-Kbyte, 32-byte block, LRU data cache would thus have the parameter:

-d*L1dcache:1024:32:2:l*

-f Flush caches on system calls

-i*name:sets:blocksize:assoc:repl* - Simulate an instruction cache, with the same parameter format as the data cache example above.

-t*name:sets:blocksize:assoc:repl* - Simulate a TLB, using the same parameter format as the instruction and data caches above.

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

4.3 Simulating in-order issue execution

The simulator found in **sim-inorder.c** models an in-order issue processor, including timing of functional units, memory latencies, and thus gives cycle counts for programs' executions. In addition to **cache.c** and the files used for functional simulation, sim-inorder also uses **bpred.c**, **eventq.c**, and **resource.c**.

This simulator assumes a four-stage pipelined processor. The four stages are fetch, decode, execute, and writeback. Each of these stages are handled by a different function: `ifetch()`, `idecode()`, and `execute()`, in **sim-inorder.c** and `eventq_service_events()` in **eventq.c**. The in-order issue pipeline supports out-of-order completion, but stalls the pipeline upon detection of a data hazard.

Both sim-inorder and sim-outorder (discussed in Section 4.4) perform speculative execution—they execute down a speculative path until they detect a fault, a TLB miss, or a branch misprediction. Both simulators support dynamic and static branch prediction. The dynamic prediction uses a branch target buffer with 2-bit saturating counters.

For timing purposes, both simulators assume the following functional unit latencies (which may easily be changed). The latencies are presented as (cycles for one operation)/(initiation rate). The latencies are: Integer ALU: 1/1, load/store unit: 2/1, integer multiply: 3/1, integer division: 12/12, floating-point addition: 2/1, floating-point multiplication: 4/1, floating-point division: 12/12.

sim-inorder accepts a superset of the command-line arguments that sim-cache allows[1]. In addition to those of sim-cache, sim-inorder accepts the following arguments:

-b*size* Use a branch prediction table for branch speculation, with *size* entries. *size* must be a power of two.

-j*penalty* Set the branch misprediction penalty to *penalty* cycles. The default is 2.

-k Run with a blocking cache. This option currently has no effect.

-m*latency* Defines main memory access time to be *latency* cycles. This option must precede cache definition arguments, if any, on the command line. The default is 6 cycles.

-p Run with infinite bandwidth in terms of memory instruction issue (allow issue of multiple load/store instructions in the same cycle).

-s*choice* Use static branch prediction, where *choice* is either the string "taken" or "nottaken".

-w*width* Sets the issue width of the processor to be *width*. The argument must be a power of two (and greater than zero).

-y If this option is declared, instruction fetch will not continue (e.g., stop filling the decode stage up to the issue width) on branches.

4.4 Simulating out-of-order issue execution

By far the most complicated and detailed simulator is in **sim-outorder.c**. This simulator supports out-of-order issue and execution, based on the Register Update Unit [2]. This 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 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 by either the memory system or an earlier store value sitting 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.

The sim-outorder simulator file is over 2200 lines long, and runs about an order of magnitude slower than sim-fast (150,000 cycles per second compared to about 2.5 million per second, on a Sparc SS-10).

TLBs are not currently activated in the simulator; the code is there but #defined out. The branch misprediction penalty has a default of 3 cycles (as opposed to 2 in sim-inorder). The following arguments accepted by sim-inorder are not supported in sim-outorder: '-f', '-k', '-p', '-y'. The arguments unique to sim-outorder are as follows:

-D*width* Sets the decode width to be *width*, which much be a power of two. The default is 4.

-L*size* Sets the number of entries in the load/store queue to be *size*. *size* must be a power of two, and is set to a default of 4.

-R*entries* Sets the number of slots in the reorder buffer to be *entries*. *entries* must be a power of two. The default is 8.

-W*width* Sets the issue width to be *width*, which much be a power of two. The default is 4.

-0 Prevents the simulator from issuing mis-speculated instructions.

-1 Forces the simulator to use in-order issue.

1. The arguments of sim-inorder are a superset of those of sim-cache, except for the 't' argument; TLBs are not yet fully supported in sim-inorder or sim-outorder.

-9 Run with perfect branch prediction.

4.5 Common support files

The following list describes the functionality of the C code files in the simplesim-0.1 directory, that are shared 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 dynamic branch prediction buffer.
- *cache.[c,h]*: Contains general functions to support multiple caches (e.g., TLB, instruction and data cache, BRB). 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.
- *endian.[c,h]*: Defines a few simple functions to determine byte- and word-order on the host and target platforms.
- *eventq.[c,h]*: Defines ten functions and two macros to handle ordered event queues that control when writebacks occur.
- *loader.[c,h]*: Loads the target program into memory, sets up the segment sizes and addresses, and obtains the target program entry point.
- *main.c*: Performs all initialization and launches the main simulator function (`sim_main()`).
- *memory.[c,h]*: Contains functions for reading from, writing to, initializing, and dumping the contents of the target main memory.
- *misc.[c,h]*: Contains support functions, most notably argument string parsing and string manipulation functions.
- *regs.[c,h]*: Allocates space for the register files, and contains functions to initialize them 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, return a resource (if any are available) from a given pool, and dump the contents of a pool.
- *sim.h*: Contains a few extern variable declarations and function prototypes.
- *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.
- *syscall.[c,h]*: This file functions 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 host platform, and generates appropriate compiler flags.
- *version.h*: Defines the version number and release date of the distribution.

5 Details of SimpleScalar x86

In this section we describe the command-line arguments for SimpleScalar x86, discuss each of the source files, and close with a brief description of our experiences running the simulator with the Spec92 benchmark suite [3].

5.1 SimpleScalar x86 command-line arguments.

Currently, only a functional simulator is available for SimpleScalar x86. The simulator is functionally and structurally similar to that described in Section 4.1, but the code is quite different. The command line for the simulator is:

sim-func-x86 -W*baseName* **[switches]** *executable* **[arguments] < [input to target program] > [output of target program]**

The only argument required to run the simulator is as follows:

-W*baseName*	This switch sets the base name in the simulator. It is used to name result files. It is required.

The optional arguments that the simulator will accept are the following:

-v*instAddr*	When an instruction at address *instAddr* is encountered, become verbose (dump trace information to stderr)
-vt*time*	After executing *time* cycles, become verbose. This option currently has no effect, since we have only implemented a functional simulator (in which cycle counts are meaningless).
-vi*instCount*	After executing *instCount* instructions, become verbose. If instCount==1, the simulator will be verbose from the first instruction.
-V	Be verbose during system calls.
-m*count*	Execute only *count* instructions, then terminate.
-?	Display usage information

5.2 Simulator code description

Below we list the code files for the simulator with a high-level description of their purpose:

- *main.c:* The main driver for the simulator.
- *func.[hc]*: Implements the high-level functionality of a simple functional simulator.
- *ix86.def*: Captures the functionality of the instruction set. This file is similar to **ss.def** in the main SimpleScalar release. The complex decoding required for x86 instructions made this structure quite convoluted. Although messy, this macro strategy avoids both storing a huge lookup table in simulator memory to parse instructions, and building an unwieldy nest of case statements that would parse the instruction stream.
- *non-spec.<un>defines*: The *non-spec.defines* file contains the macros called in ix86.def that actually execute the instructions. *non-spec.undefines* undefines these macros. The pair of files was needed because these files change the state of the machine, and can not be undone. Eventually we plan to have a corresponding pair of files for speculative execution, that will be used after branch prediction, so that the simulator may recover from a branch mispredictions.
- *translate.[hc]*: Parses the instruction byte stream, using the

structure in ix86.def. This file uses a lookup table that uses only certain bits from the instructions to reduce the size of the tables. This module also contains a "decoded instruction cache" that speeds simulation.

- *operands.[hc]*: Implements operand fetch and store functionality. The "ops" text file in $X86ROOT contains descriptions of every operand type.
- *helper.[hc]*: Contains simple, x86-specific functions such as shift, rotate and flag manipulation.
- *syscall.[hc]*: Describes the operating system emulation. Each target operating system call is mapped to either an equivalent host operating system call, or a series of helper routines that duplicate the functionality of the target system call. Currently this file is only compatible with SunOS calls.
- *memory.[hc]*: Similar to the corresponding SimpleScalar files.
- misc.[hc]: Contains simple functions such as sign extension and MIN/MAX error routines.

5.3 Simulator details

Using GCC and Linux (instead of DOS or Windows, for example) eliminated many difficulties, including:

- segment register manipulation
- segment register overrides
- 16 bit addressing modes
- self-modifying code
- kernel instructions (tlb, cache, control register, etc.)

Some of the files contain code to count micro operations (delimited by #ifdef MICRO_OPS ... #endif). This functionality is only partially implemented, but should provide a start if you want to extend the code to handle μops explicitly (the code currently handles the CISC x86 instructions correctly). If you use the micro operations code, the simulator creates 2 output files: *baseName*.ops-dist and *baseName*.ops_file. The former file contains the distribution of number of μops per x86 instruction. The latter file contains the number of occurrences of each x86 instruction.

To validate this functional simulator, we attempted to simulate all of the SPEC92 benchmarks. Table 2 lists the benchmarks that we ran with their corresponding inputs. Below the table we list specific problems that occurred when we simulated these benchmarks.them

The integer benchmarks were simulated much more successfully than the floating point benchmarks. Most of the problems with the floating-point codes involved the different formats of x86 and SPARC floating-point numbers (80 bits versus 64 bits). The simulator currently does not support the ability of the programmer or compiler on x86 machines to write 80-bit values to memory. When the compiler spills the 80-bit floating point numbers, therefore, the simulator does not function correctly (implementing this feature should not be prohibitively difficult).

Another difficulty that we experienced involved the *curses* library when simulating *sc*: the functionality of the Linux libc varies from the SunOS implementation. We emphasize that the objdump-i486 utility was invaluable in debugging the simulator. Finally, we note that the Intel 486 Programmers Reference Manual from which we worked was riddled with bugs.

Program	Input Tested
cc1	1recog.i
compress	in (100k and 1MB versions)
elvis	unix.c
eqntott	input.short/int_pri_3.eqn
espresso	opa.in, and others
grep	input.txt and various others
perl	tests.pl
yacr2	input2
xlisp	8queens
alvinn	10 iterations
doduc[a]	doducin.tiny
ear[b]	short.m22
fpppp[a]	8 atoms
hydro2d[c]	short
mdljdp2[d]	built in
mdljsp2	built in
spice[e]	short.in
su2cor[a]	built in
swm256[a]	built in
tomcatv	100 iterations

Table 2: Test programs and inputs

a. Program executes an unimplemented instruction (fstp80)
b. Results are off slightly from the sun result, but matches when the ascii output files are compared with spiff -r0.005
c. Completes without error, but result file does not match sun results
d. Runs 3.7 billion instructions and then stops on a CUBLOW statement (in the mdljdp2 source).
e. Errors during scanning phase of program

6 Summary

The SimpleScalar tool set was written by Todd Austin over about one and a half years, between 1994 and 1996. The ancestors of the tool set date back to the mid to late 1980s, to tools written by Manoj Franklin. Steve Bennett wrote SimpleScalar x86 during the summer of 1995. 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 release was assembled, debugged, and documented by Doug Burger, also a research assistant at Wisconsin. Much of the SimpleScalar x86 documentation in this report was only slightly modified from the report written entirely by Steve Bennett.

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 tools themselves.

References

[1] Charles Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, California, January

1995.
[2] 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.
[3] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, Virginia, December 1991.

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 2. 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.
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.

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: 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 or [Kane:92] 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 or [Kane:92] 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)+OFFSET+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)+OFFSET)

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)+OFFSET) WRITE_WORD(FPR_L(FT+1), GPR(RS)+OFFSET+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` or [Kane:92] 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` or [Kane:92] 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:

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 $\$v0$ should be loaded with the system call code. The arguments of the system call interface prototype should be loaded into registers $\$a0$ - $\$a3$ in the order specified by the system call interface prototype, *e.g.*, for:

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

0x03 is loaded into $\$v0$, fd is loaded into $\$a0$, buf into $\$a1$, and nbyte into $\$a2$.

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).