Getting Started with the SimpleScalar Tool Set Version 2.0

Introduction

This document contains everything that you need to know to work the SimpleScalar problems in the textbook. Section 1 is a summary that will familiarize you with SimpleScalar and the use of simulation tools in computer architecture research. Section 2 tells you how to get the tool set from the Web and set up the part that is needed for the textbook problems. Section 3 describes each of the simulators, including the information supplied by each of them. Section 4 describes the SPEC95 benchmark binaries that can be used with the simulators. Section 5 consists of a complete example of the use of one of the simulators. Section 6 briefly describes the complete tool set for people interested in going beyond the problems in the textbook.

1 SimpleScalar and Simulation in Computer Architecture

When computer architecture researchers work to improve the performance of a computer system, they often use an existing system to simulate a proposed system. Although the intent is not always to measure raw performance (estimating power consumption is one alternative), performance estimation is one of the most important results obtained by simulation. The SimpleScalar tool set is designed to measure the performance of several parts of a superscalar processor and its memory hierarchy. This document describes the SimpleScalar simulators. Other simulation systems may be similar or very different.

Overview of SimpleScalar Simulation

The SimpleScalar tool set includes a compiler that creates binaries for a non-existent processor. The binaries can be executed on one of several simulators that are included in the tool set. This section describes the goals of processor simulation.

The execution of a processor can be modeled as a series of known states and the time (or other costs, i.e., power) required to make the transition between each pair of states. The state information may include all or a subset of:

- The values stored in all memory locations.
- The values stored in and the status of all cache memories.
- The values stored in and the status of the translation-lookaside buffer (TLB).
- The values stored in and the status of the branch prediction table(s) or branch target buffer (BTB).
- All processor state (i.e., the pipeline, execution units (integer ALU, load/store unit, etc.), register file, register update unit (RUU), etc.)

A good way to evaluate the performance of a program on a proposed processor architecture is to simulate the state of the architecture during the execution of the program. By simulating the states through which the processor will pass during the execution of a program and estimating the time (or other measurement) necessary for each state transition, the amount of time that the simulated processor will need to execute the program can be estimated.

The more state that is simulated, the longer a simulation will take. Complex simulations can execute 100s of times slower than a real processor. Therefore, simulating the execution of a program that would take an hour of CPU time on an existing processor can take a week on a complex simulator. For this reason, it is important to evaluate what measurements are desired and limit the simulation to only the state that is necessary to properly estimate those measurements. This is the reason for the inclusion of several different simulators in the SimpleScalar tool set.

Profiling

In addition to estimating the execution time of a program on the simulated processor, profile information may be of use to computer architects. Profile information is a count of the number or frequency of events that occur during the execution of a program. One common example of profile data is a count of how often each type of instruction (i.e., branch, load, store, ALU operation, etc.) is executed during the running of a program.

Profile information can be used to gauge the relative importance of each part of a processor's implementation in determining its performance when executing the profiled program.

The SimpleScalar Base Processor

The SimpleScalar tool set is based on the MIPS R2000 processor's instruction set architecture (ISA). The processor is described in MIPS RISC Architecture by Gerry Kane, published by Prentice Hall, 1988. The ISA describes the instructions that the processor is capable of executing—and therefore the instructions that a compiler can generate—but it does not describe how the instructions are implemented. The implementation is what computer architects change in order to improve the performance of a processor.

An existing processor can be chosen as a base processor for several reasons. These may include:

- The architecture of the processor is well known and documented.
- The architecture of the processor is state-of-the-art and therefore it is likely to be useful as a base for the study of future processors.
- The architecture of the processor has been implemented as a real processor, allowing simulations to be compared to executions on a real, physical processor.

An important consideration in the choice of the MIPS architecture for the SimpleScalar tool set was the fact that the GNU GCC compiler was available in source-code form, and could compile to the MIPS architecture. This allowed the use of this public-domain software as part of the SimpleScalar tool set.

2 Setting-Up the Simulators

The SimpleScalar simulators are part of the complete SimpleScalar tool set that is described in Section 6. The simulators are the easiest part of the tool set to set up, and they can be used to simulate existing SimpleScalar binaries without the need for any of the other parts of the tool set.

The simulators are available in a file that is available on the *SimpleScalar Tools Home Page* at

http://www.cs.wisc.edu/~mscalar/simplescalar.html

on the Web. The file is called `simplesim.tar`. You can get it directly from the Web page or via ftp (the ftp file is named `simplesim.tar.gz`) as described on the Web page. At the time that this document was written, the current release of SimpleScalar was version 2.0.

Using standard Unix commands, you can unpack the archive. All that remains is to build the simulators by moving to the directory containing them (probably `simplesim-2.0`) and building them by typing `make`. Note that the default optimization is `-O`. The simulators will run much faster (i.e., 10 times) if you modify the make file (`Makefile`) section “Choose your optimization level here” so that the “for optimization” flags are used as the `OFLAGS`. If you have built the simulators and want to rebuild them with different optimizations, type `make clean` and then `make`.

Note: Building the most complex simulator, `sim-outorder`, may require 64MB of memory when it is compiled with all optimizations. If the building process is stalled for several minutes on the line

that compiles `sim-outorder`, you may want to try changing the optimizations. You may also want to simply wait for the build to finish, as the simulator will run much faster with the optimizations included.

After you have built the simulators, either move the executables to a directory in your execution path or add the directory in which you built the executables to your execution path. The simulator executable files include:

- `sim-bpred`
- `sim-cache`
- `sim-cheetah`
- `sim-fast`
- `sim-outorder`
- `sim-profile`
- `sim-safe`

Each of these is described in the next section.

3 Description of the Simulators

Each of the simulators appears here in a separate section. Each section includes the name of the simulator and the output that is generated when you ask for the simulator's help screen by executing it with no input file or parameters.

These help screens—written by the SimpleScalar authors—include a short description of the function of the simulator, a short description of why you would want to use the particular simulator, and a description of the input parameters for the simulator.

sim-bpred

```
sim-bpred: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-bpred {-options} executable {arguments}

sim-bpred: This simulator implements a branch predictor analyzer.

#
# -option       <args>          #    <default> # description
#
-config         <string>        #       <null> # load configuration from a file
-dumpconfig     <string>        #       <null> # dump configuration to a file
-h              <true|false>    #        false # print help message
-v              <true|false>    #        false # verbose operation
-d              <true|false>    #        false # enable debug message
-i              <true|false>    #        false # start in Dlite debugger
-seed           <int>           #            1 # random number generator seed (0 for timer seed)
-q              <true|false>    #        false # initialize and terminate immediately
-bpred          <string>        #        bimod # branch predictor type {nottaken|taken|bimod|2lev|comb}
-bpred:bimod    <int>           # 2048 # bimodal predictor config (<table size>)
-bpred:2lev     <int list...>   # 1 1024 8 0 # 2-level predictor config (<l1size> <l2size> <hist_size>
<xor>)
-bpred:comb     <int>           # 1024 # combining predictor config (<meta_table_size>)
-bpred:ras      <int>           #            8 # return address stack size (0 for no return stack)
-bpred:btb      <int list...>   # 512 4 # BTB config (<num_sets> <associativity>)

  Branch predictor configuration examples for 2-level predictor:
    Configurations:   N, M, W, X
      N   # entries in first level (# of shift register(s))
      W   width of shift register(s)
      M   # entries in 2nd level (# of counters, or other FSM)
      X   (yes-1/no-0) xor history and address for 2nd level index
    Sample predictors:
      GAg     : 1, W, 2^W, 0
      GAp     : 1, W, M (M > 2^W), 0
      PAg     : N, W, 2^W, 0
      PAp     : N, W, M (M == 2^(N+W)), 0
      gshare  : 1, W, 2^W, 1
  Predictor `comb' combines a bimodal and a 2-level predictor.
```

sim-cache

```
sim-cache: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-cache {-options} executable {arguments}

sim-cache: This simulator implements a functional cache simulator.  Cache
statistics are generated for a user-selected cache and TLB configuration,
which may include up to two levels of instruction and data cache (with any
levels unified), and one level of instruction and data TLBs.  No timing
information is generated.

#
# -option       <args>          #    <default> # description
#
-config         <string>        #       <null> # load configuration from a file
-dumpconfig     <string>        #       <null> # dump configuration to a file
-h              <true|false>    #        false # print help message
-v              <true|false>    #        false # verbose operation
-d              <true|false>    #        false # enable debug message
-i              <true|false>    #        false # start in Dlite debugger
-seed           <int>           #            1 # random number generator seed (0 for timer seed)
-q              <true|false>    #        false # initialize and terminate immediately
-cache:dl1      <string>        # dl1:256:32:1:l # l1 data cache config, i.e., {<config>|none}
-cache:dl2      <string>        # ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:il1      <string>        # il1:256:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il2      <string>        #          dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-tlb:itlb       <string>        # itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb       <string>        # dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-flush          <true|false>    #        false # flush caches on system calls
-icompress      <true|false>    #        false # convert 64-bit inst addresses to 32-bit inst equivalents
-pcstat         <string list...> #       <null> # profile stat(s) against text addr's (mult uses ok)

  The cache config parameter <config> has the following format:

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

    <name>   - name of the cache being defined
    <nsets>  - number of sets in the cache
    <bsize>  - block size of the cache
    <assoc>  - associativity of the cache
    <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

    Examples:   -cache:dl1 dl1:4096:32:1:l
                -dtlb dtlb:128:4096:32:r

  Cache levels can be unified by pointing a level of the instruction cache
  hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
  configuration arguments.  Most sensible combinations are supported, e.g.,

    A unified l2 cache (il2 is pointed at dl2):
      -cache:il1 il1:128:64:1:l -cache:il2 dl2
      -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

    Or, a fully unified cache hierarchy (il1 pointed at dl1):
      -cache:il1 dl1
      -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```

sim-cheetah

```
sim-cheetah: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Portions Copyright (C) 1989-1993 by Rabin A. Sugumar and Santosh G. Abraham.
Usage: sim-cheetah {-options} executable {arguments}

sim-cheetah: This program implements a functional simulator driver for
Cheetah.  Cheetah is a cache simulation package written by Rabin Sugumar
and Santosh Abraham which can efficiently simulate multiple cache
configurations in a single run of a program.  Specifically, Cheetah can
simulate ranges of single level set-associative and fully-associative
caches.  See the directory libcheetah/ for more details on Cheetah.

#
# -option        <args>          #    <default> # description
#
-config          <string>        #       <null> # load configuration from a file
-dumpconfig      <string>        #       <null> # dump configuration to a file
-h               <true|false>    #        false # print help message
-v               <true|false>    #        false # verbose operation
-d               <true|false>    #        false # enable debug message
-i               <true|false>    #        false # start in Dlite debugger
-seed            <int>           #            1 # random number generator seed (0 for timer seed)
-q               <true|false>    #        false # initialize and terminate immediately
-refs            <string>        #         data # reference stream to analyze, i.e., {inst|data|unified}
-R               <string>        #          lru # replacement policy, i.e., lru or opt
-C               <string>        #           sa # cache configuration, i.e., fa, sa, or dm
-a               <int>           #            7 # min number of sets (log base 2, line size for DM)
-b               <int>           #           14 # max number of sets (log base 2, line size for DM)
-l               <int>           #            4 # line size of the caches (log base 2)
-n               <int>           #            1 # max degree of associativity to analyze (log base 2)
-in              <int>           #          512 # cache size intervals at which miss ratio is shown
-M               <int>           #       524288 # maximum cache size of interest
-c               <int>           #           16 # size of cache (log base 2) for DM analysis
```

sim-fast

```
sim-fast: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-fast {-options} executable {arguments}

sim-fast: This simulator implements a very fast functional simulator.  This
functional simulator implementation is much more difficult to digest than
the simpler, cleaner sim-safe functional simulator.  By default, this
simulator performs no instruction error checking, as a result, any
instruction errors will manifest as simulator execution errors, possibly
causing sim-fast to execute incorrectly or dump core.  Such is the
price we pay for speed!!!!

#
# -option        <args>          #    <default> # description
#
-config          <string>        #       <null> # load configuration from a file
-dumpconfig      <string>        #       <null> # dump configuration to a file
-h               <true|false>    #        false # print help message
-v               <true|false>    #        false # verbose operation
-d               <true|false>    #        false # enable debug message
-i               <true|false>    #        false # start in Dlite debugger
-seed            <int>           #            1 # random number generator seed (0 for timer seed)
-q               <true|false>    #        false # initialize and terminate immediately
```

sim-outorder

```
sim-outorder: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-outorder {-options} executable {arguments}

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

#
# -option        <args>           #    <default> # description
#
-config          <string>         #       <null> # load configuration from a file
-dumpconfig      <string>         #       <null> # dump configuration to a file
-h               <true|false>     #        false # print help message
-v               <true|false>     #        false # verbose operation
-d               <true|false>     #        false # enable debug message
-i               <true|false>     #        false # start in Dlite debugger
-seed            <int>            #            1 # random number generator seed (0 for timer seed)
-q               <true|false>     #        false # initialize and terminate immediately
-ptrace          <string list...> #       <null> # generate pipetrace, i.e., <fname|stdout|stderr> <range>
-fetch:ifqsize   <int>            #            4 # instruction fetch queue size (in insts)
-fetch:mplat     <int>            #            3 # extra branch mis-prediction latency
-fetch:speed     <int>            #            1 # speed of front-end of machine relative to execution core
-bpred           <string>         #        bimod # branch predictor type
{nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     <int>            # 2048 # bimodal predictor config (<table size>)
-bpred:2lev      <int list...>    # 1 1024 8 0 # 2-level predictor config (<l1size> <l2size> <hist_size>
<xor>)
-bpred:comb      <int>            # 1024 # combining predictor config (<meta_table_size>)
-bpred:ras       <int>            #            8 # return address stack size (0 for no return stack)
-bpred:btb       <int list...>    # 512 4 # BTB config (<num_sets> <associativity>)
-bpred:spec_update <string>         #       <null> # speculative predictors update in {ID|WB} (default non-
spec)
-decode:width    <int>            #            4 # instruction decode B/W (insts/cycle)
-issue:width     <int>            #            4 # instruction issue B/W (insts/cycle)
-issue:inorder   <true|false>     #        false # run pipeline with in-order issue
-issue:wrongpath <true|false>     #         true # issue instructions down wrong execution paths
-commit:width    <int>            #            4 # instruction commit B/W (insts/cycle)
-ruu:size        <int>            #           16 # register update unit (RUU) size
-lsq:size        <int>            #            8 # load/store queue (LSQ) size
-cache:dl1       <string>         # dl1:128:32:4:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat    <int>            #            1 # l1 data cache hit latency (in cycles)
-cache:dl2       <string>         # ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat    <int>            #            6 # l2 data cache hit latency (in cycles)
-cache:il1       <string>         # il1:512:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat    <int>            #            1 # l1 instruction cache hit latency (in cycles)
-cache:il2       <string>         #          dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2lat    <int>            #            6 # l2 instruction cache hit latency (in cycles)
-cache:flush     <true|false>     #        false # flush caches on system calls
-cache:icompress <true|false>     #        false # convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat         <int list...>    # 18 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width       <int>            #            8 # memory access bus width (in bytes)
-tlb:itlb        <string>         # itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        <string>         # dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat         <int>            #           30 # inst/data TLB miss latency (in cycles)
-res:ialu        <int>            #            4 # total number of integer ALU's available
-res:imult       <int>            #            1 # total number of integer multiplier/dividers available
-res:memport     <int>            #            2 # total number of memory system ports available (to CPU)
-res:fpalu       <int>            #            4 # total number of floating point ALU's available
-res:fpmult      <int>            #            1 # total number of floating point multiplier/dividers
available
-pcstat          <string list...> #       <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat       <true|false>     #        false # operate in backward-compatible bugs mode (for testing only)

  Pipetrace range arguments are formatted as follows:
```

```
  {{@|#}<start>}:{{@|#|+}<end>}

Both ends of the range are optional, if neither are specified, the entire
execution is traced.  Ranges that start with a `@' designate an address
range to be traced, those that start with an `#' designate a cycle count
range.  All other range values represent an instruction count range.  The
second argument, if specified with a `+', indicates a value relative
to the first argument, e.g., 1000:+100 == 1000:1100.  Program symbols may
be used in all contexts.

  Examples:   -ptrace FOO.trc #0:#1000
              -ptrace BAR.trc @2000:
              -ptrace BLAH.trc :1500
              -ptrace UXXE.trc :
              -ptrace FOOBAR.trc @main:+278

Branch predictor configuration examples for 2-level predictor:
  Configurations:   N, M, W, X
    N   # entries in first level (# of shift register(s))
    W   width of shift register(s)
    M   # entries in 2nd level (# of counters, or other FSM)
    X   (yes-1/no-0) xor history and address for 2nd level index
  Sample predictors:
    GAg     : 1, W, 2^W, 0
    GAp     : 1, W, M (M > 2^W), 0
    PAg     : N, W, 2^W, 0
    PAp     : N, W, M (M == 2^(N+W)), 0
    gshare  : 1, W, 2^W, 1
Predictor `comb' combines a bimodal and a 2-level predictor.

The cache config parameter <config> has the following format:

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

  <name>   - name of the cache being defined
  <nsets>  - number of sets in the cache
  <bsize>  - block size of the cache
  <assoc>  - associativity of the cache
  <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

  Examples:   -cache:dl1 dl1:4096:32:1:l
              -dtlb dtlb:128:4096:32:r

Cache levels can be unified by pointing a level of the instruction cache
hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
configuration arguments.  Most sensible combinations are supported, e.g.,

  A unified l2 cache (il2 is pointed at dl2):
    -cache:il1 il1:128:64:1:l -cache:il2 dl2
    -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

  Or, a fully unified cache hierarchy (il1 pointed at dl1):
    -cache:il1 dl1
    -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```

sim-profile

```
sim-profile: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-profile {-options} executable {arguments}

sim-profile: This simulator implements a functional simulator with
profiling support.  Run with the `-h' flag to see profiling options
available.

#
# -option       <args>          #    <default> # description
#
-config         <string>        #       <null> # load configuration from a file
-dumpconfig     <string>        #       <null> # dump configuration to a file
-h              <true|false>    #        false # print help message
-v              <true|false>    #        false # verbose operation
-d              <true|false>    #        false # enable debug message
-i              <true|false>    #        false # start in Dlite debugger
-seed           <int>           #            1 # random number generator seed (0 for timer seed)
-q              <true|false>    #        false # initialize and terminate immediately
-all            <true|false>    #        false # enable all profile options
-iclass         <true|false>    #        false # enable instruction class profiling
-iprof          <true|false>    #        false # enable instruction profiling
-brprof         <true|false>    #        false # enable branch instruction profiling
-amprof         <true|false>    #        false # enable address mode profiling
-segprof        <true|false>    #        false # enable load/store address segment profiling
-tsymprof       <true|false>    #        false # enable text symbol profiling
-taddrprof      <true|false>    #        false # enable text address profiling
-dsymprof       <true|false>    #        false # enable data symbol profiling
-internal       <true|false>    #        false # include compiler-internal symbols during symbol profiling
-pcstat         <string list...> #       <null> # profile stat(s) against text addr's (mult uses ok)
```

sim-safe

```
sim-safe: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.

Usage: sim-safe {-options} executable {arguments}

sim-safe: This simulator implements a functional simulator.  This
functional simulator is the simplest, most user-friendly simulator in the
simplescalar tool set.  Unlike sim-fast, this functional simulator checks
for all instruction errors, and the implementation is crafted for clarity
rather than speed.

#
# -option        <args>          #    <default> # description
#
-config          <string>        #       <null> # load configuration from a file
-dumpconfig      <string>        #       <null> # dump configuration to a file
-h               <true|false>    #        false # print help message
-v               <true|false>    #        false # verbose operation
-d               <true|false>    #        false # enable debug message
-i               <true|false>    #        false # start in Dlite debugger
-seed            <int>           #            1 # random number generator seed (0 for timer seed)
-q               <true|false>    #        false # initialize and terminate immediately
```

4 The SPEC95 Benchmark Binaries

The Standard Performance Evaluation Corporation (SPEC) provides (for a fee) a set of benchmarks that are used to evaluate processors. At the time this document was written the current release of these benchmarks was called SPEC95. It consists of two parts: (i) CINT95 is a set of integer benchmarks and (ii) CFP95 is a set of floating point benchmarks. You can find information about these benchmarks on the Web at

http://www.spec.org/osg/cpu95/.

Using the SPEC95 Benchmarks with the SimpleScalar Simulators

The Standard Performance Evaluation Corporation has allowed the authors of the SimpleScalar tool set to make a set of SimpleScalar binaries for the SPEC95 benchmarks available on the Web and via ftp. These binaries can be run on the simulators as provided with no compilation or other manipulation. Because SPEC does not want the inputs used for their standard benchmark suite to be distributed, you will have to create appropriate command-line options and/or input files for each of the benchmarks. The inputs have been provided in the written problems that use the SPEC95 benchmarks. Additional information can be found by taking advantage of the fact that many of the SPEC95 benchmarks are derived from programs that are available free-of-charge on the Web.

Installing the SPEC95 Benchmarks

In order to install the SPEC95 benchmarks, you will have to determine the endianness of the processor on which you will be running the simulators. The endianness of a processor is defined as the part (byte) of a multi-byte number that appears first in a word, when the word is viewed as consisting of bytes ordered by their physical order in memory. Thus, a little-endian processor will have the least-significant (e.g., the “little”) byte first and big-endian will have the most-significant (e.g., the “big”) byte first. The ordering of the first two 4-byte words of memory for each endianness is shown in Table 1.

Processor Endianness	First Word Byte Locations				Second Word Byte Locations			
Little-endian	3	2	1	0	7	6	5	4
Big-endian	0	1	2	3	4	5	6	7

Table 1: The difference between big- and little-endian.

Intel x86 processors are little-endian. Most other processors (Alpha, SPARC, MIPS, etc.) are big-endian. When you get the SPEC95 benchmark binaries, get the file that corresponds to the endianness of your processor. If you are not certain which endianness you processor uses, you can find out by executing the program `sysprobe` that is built in the same directory as the simulators (See Section 2.)

The SPEC95 SimpleScalar binaries are available on the Web at

http://www.cs.wisc.edu/~mscalar/simplescalar.html

or via ftp (the ftp files will have a `.gz` suffix) as described on that page. The file for little-endian processors is called `simplebench.little.tar` and the file for big-endian processors is called `simplebench.big.tar`. Using standard Unix commands, you can unpack the archive. The result will be a directory that contains one file for each SPEC95 benchmark. The files have a `.ss` extension to distinguish them as SimpleScalar binary files. See the next section to test your installation.

5 An Example

You can test your installation by executing the SPEC95 `go` benchmark using the fastest simulator. After you have installed the simulators, set up the execution path, and set up the proper SPEC95 SimpleScalar binaries as described above, move to the directory that contains the SPEC95 SimpleScalar binaries. You should then create an empty file called `go.in` using the Unix command `touch go.in`. You can now run the `go` benchmark by typing the command `sim-fast go.ss 2 8 go.in`. If the simulation works, you can watch the two computer players as they present their moves. The game (and therefore the simulation) will end at move number 40.

Here is the output you will see when you run the example:

```
sim-fast: Version 2.0 of July, 1997.
Copyright (c) 1994-1997 by Todd M. Austin.  All Rights Reserved.


sim: simulation started @ Fri May 28 08:30:59 1999, options follow:

sim-fast: This simulator implements a very fast functional simulator.  This
functional simulator implementation is much more difficult to digest than
the simpler, cleaner sim-safe functional simulator.  By default, this
simulator performs no instruction error checking, as a result, any
instruction errors will manifest as simulator execution errors, possibly
causing sim-fast to execute incorrectly or dump core.  Such is the
price we pay for speed!!!!

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately

sim: ** starting *fast* functional simulation **
Initial position from go.in
  1 B D6
  2 W C4
  3 B E3
  4 W B6
  5 B C2
  6 W C7
  7 B D7
  8 W E4
  9 B D4
 10 W D8
 11 B E6
 12 W F8
 13 B B3
 14 W F7
 15 B F6
 16 W D5
 17 B G4
 18 W E7
 19 B C8
 20 W E8
 21 B B7
 22 W C6
 23 B G7
 24 W B8
 25 B G6
 26 W A7
 27 B F3
 28 W D3
 29 B D2
 30 W B4
 31 B E2
```

```
 32 W A3
 33 B A2
 34 W A4
 35 B G8
 36 W C8
 37 B C3
 38 W D4
 39 B pass
 40 W pass
Game over

sim: ** simulation statistics **
sim_elapsed_time                8 # total simulation time in seconds
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                621600 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                578004 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
ld_environ_base          0x7fff8000 # program environment base address address
ld_target_big_endian            0 # target executable endian-ness, non-zero if big endian
mem_brk_point            0x10094000 # data segment break point
mem_stack_min            0x7fff7500 # lowest address accessed in stack segment
mem_total_data                565k # total bytes used in init/uninit data segment
mem_total_heap                 28k # total bytes used in program heap segment
mem_total_stack                19k # total bytes used in stack segment
mem_total_mem                 612k # total bytes used in data, heap, and stack segments
```

6 The Complete Tool Set

In addition to the simulators, the SimpleScalar tool set includes several other packages that allow you to create SimpleScalar executable files. These are described in detail in the document *The SimpleScalar Tool Set, Version 2.0*, written by Doug Burger and Todd M. Austin while they were at the University of Wisconsin. The document includes installation instructions for each of the packages. You can find a copy of the document at

http://www.cs.wisc.edu/~mscalar/simplescalar.html

on the Web. The file is called `TR_1342.ps`.

The complete tool set consists of these packages:

- `simplesim.tar` (`simplesim.tar.gz` via ftp) is the simulators as mentioned above.
- `simpleutils.tar` (`simpleutils.tar.gz` via ftp) is the GNU binutils version 2.5.2 modified to compile to the SimpleScalar architecture. These are needed if you are going to use the SimpleScalar compiler.
- `simpletools.tar` (`simpletools.tar.gz` via ftp) is the SimpleScalar-targeted GNU compiler version 2.6.3. With this and `simpleutils.tar` you can compile your own C programs to run on the SimpleScalar simulators. If you plan to do this, you should read the document *The SimpleScalar Tool Set, Version 2.0* that is mentioned at the beginning of this section.