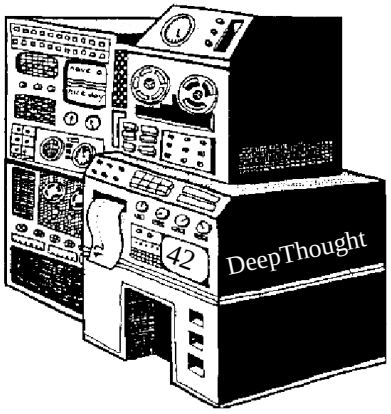


Title: **SimpleScalar CPU Simulator**

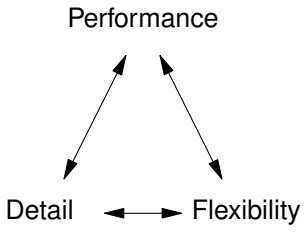
Author: Rob Williams 10/05, 10/06
Grouping: Individual or pairs
Prerequisites: Knowledge of basic CPU architecture
Languages: a bit of C
Time: 2 hours.
Break-Points:
Courses: BSc CRTS/CSE yr4

Requirements: Access to the SimpleScalar simulator software and the gcc compiler configured for the simplescalar architecture.

Summary:
This worksheet uses the SimpleScalar tool set, developed originally by Doug Burger & Todd Austin at Wisconsin Uni. Further work has now been carried out at Michigan and several other universities. It comprises a suite of powerful computer simulators which can provide both detailed and high-performance simulation for the study of modern CPU architectures.

Learning Objectives:
To be able to use the SimpleScalar tool set to test out a range of architectural parameters, such as cache size and configuration, relative fetch/issue rates, inorder/out-of-order strategies, and load/store buffering strategies. Also to gain experience with setting up special benchmark trials to profile the performance of a new CPU. Finally, to experiment with dramatically different architectures, pipeline lengths, and number.

Commercial relevance:

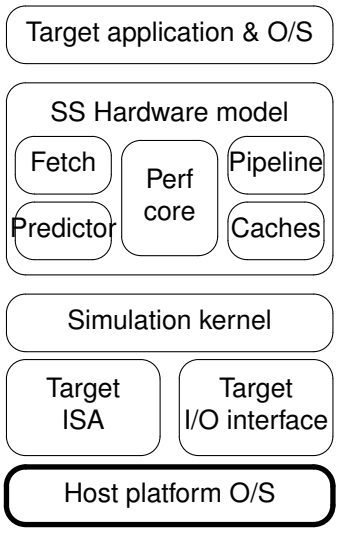


When developing a simulator the three opposing dimensions, Performance, Flexibility and Detail need to be reconciled for the needs of the user. The `sim-outorder` simulator is the most *detailed* CPU simulator provided with the SimpleScalar tool set and suffering some reduction in *performance*. It was originally based on the MIPS-4 instruction set architecture and models a very modern superscalar microprocessor with 10 execution units (pipelines). Newer ARM and x86 versions have also been developed by other research labs. It attempts to maximally exploit ILP (Instruction Level Parallelism) and keep all the execution units busy by using out-of-order instruction execution. In many ways it realistically models the current processors found in the latest workstations/PCs.
`Sim-fast` is a functional simulator, with a single, serial instruction stream, no caching and no command line switches. Also it does not capture timing information, unlike `sim-outorder`. An alternative, slower but more detailed simulation can be carried out by `sim-safe`.

SimpleScalar is widely used in academic research as well as commercial product development. The performance is good, on a P4/1.6 GHz host, sim-fast will emulate 10 Mips, while sim-outorder achieves 350 kips. The ability to skillfully use a contemporary simulator tool set provides a strength to your professional CV. Commercial ASIC or FPGA developments frequently commence with a phase of pre-HDL simulation using special tool sets, such as SimpleScalar, or bespoke C++ programs.

What you do:

Read the SimpleScalar Tool Set Report, written by Burger & Austin. There are lots of details contained that you will need to run the various simulators. The simultators are: sim-fast, sim-safe, sim-outorder, sim-cache and sim-cheetah. Note that the simulator code is compiled to run on the little-endian Pentium. But the simulators actually emulate the action of other CPUs, so to produce test programs to be run by a simulator, the test code has to be generated by a cross-compiler using a special version of gcc, simplesim-gcc, which produces the correct machine binary for the current simulator configuration. The simulators and the simplesim-gcc cross-compiler have aready been down-loaded and installed on kenny at:




```
/usr/local/simplesim/
```

Take a look. All the simulators can be operated immediately sometimes using command line flag options. The basic set are:

```
    -h   print out the help screen
    -d   turn on debug messages
    -i   run the DLite! debugger (not for sim-fast)
    -q   terminate immediately
    -dumpconfig  generate a config file from command line parameters
    -config file.cfg  use the flag options in config file
```

To carry out a test, the chosen simulator will have to "execute" a program. This will have to be presented in a valid executable format for the SimpleScalar machine architecture which is based on the MIPS 4 CPU. It is normal to use standard benchmark programs for these test runs, and several have already been prepared for use, and stored in `/usr/local/simplesim/simplesim-3.0/tests/bin.little`

```
test-fmath  test-lswlr  test-printf anagram  test-llong  test-math
```

To run a simulation session with the test-math benchmark try the following:

```
> cd /usr/local/simplesim/simplesim-3.0
> ./sim-safe test/bin.little/test-math

   sim-safe: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
   Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
   All Rights Reserved. This version of SimpleScalar is licensed for academic
   non-commercial use.  No portion of this work may be used by any commercial
   entity, or for any commercial purpose, without the prior written permission
   of SimpleScalar, LLC (info@simplescalar.com).

   sim: command line: ./sim-safe tests/bin.little/test-math

   sim: simulation started @ Sat Oct 22 22:34:39 2005, options follow:

   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.

   # -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
   # -chkpt               <null> # restore EIO trace execution from <fname>
   # -redir:sim           <null> # redirect simulator output to file (non-interacti
   ve only)
   # -redir:prog          <null> # redirect simulated program output to file
```

```
-nice                        0 # simulator scheduling priority
-max:inst                    0 # maximum number of inst's to execute

sim: ** starting functional simulation **
pow(12.0, 2.0) == 144.000000
pow(10.0, 3.0) == 1000.000000
pow(10.0, -3.0) == 0.001000
str: 123.456
x: 123.000000
str: 123.456
x: 123.456000
str: 123.456
x: 123.456000
123.456 123.456000 123 1000
sinh(2.0) = 3.62686
sinh(3.0) = 10.01787
h=3.60555
atan2(3,2) = 0.98279
pow(3.60555,4.0) = 169
169 / exp(0.98279 * 5) = 1.24102
3.93117 + 5*log(3.60555) = 10.34355
cos(10.34355) = -0.6068,  sin(10.34355) = -0.79486
x     0.5x
x0.5     x
x   0.5x
-1e-17 != -1e-17 Worked!

sim: ** simulation statistics **
sim_num_insn                213688 # total number of instructions executed
sim_num_refs                 56897 # total number of loads and stores executed
sim_elapsed_time                 1 # total simulation time in seconds
sim_inst_rate           213688.0000 # simulation speed (in insts/sec)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 91744 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 13028 # program init'ed '.data' and uninit'ed '.bs
s' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest addres
s 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.page_count                  33 # total number of pages allocated
mem.page_mem                  132k # total size of memory pages allocated
mem.ptab_misses                 34 # total first level page table misses
mem.ptab_accesses           1547345 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
>
```

Producing test programmes

Consider the following short program, testit.c:

```
// testit.c

int a[100];
int b[100];
unsigned int c = 255;
unsigned int i;

main () {
for (i = 0; i < 100; i++)
                a[i] = b[i] + c;
}
```

Before you start using the SimpleScalar version of the gcc compiler, you may want to use the bash shell `alias` facility to reduce the typing and the space taken up on your command line. You may place these instructions in your `.bashrc` file so that it gets setup with each new shell:

```
> alias ssgcc='/usr/local/simplesim/bin/sslittle-na-sstrix-gcc'
> alias sim-safe='/usr/local/simplesim/simplesim-3.0/sim-safe'
> alias sim-ooo='/usr/local/simplesim/simplesim-3.0/sim-outorder'
```

Alternatively you could insert /usr/local/simplesim/ into the PATH environment variable, but there would still remain the need to type complex names. ssssing aliases, you can simply type in ssgcc in place of /usr/local/simplesim/bin/sslittle-na-sstrix-gcc, clearly a good thing because you can compile a test program with:

```
> ssgcc -g -O testit.c -o testit
```

or to get n assembler file to look at:

```
> ssgcc -S testit.c
  cat testit.s
        .file   1 "testit.c"

 # GNU C 2.6.3 [AL 1.1, MM 40, tma 0.1] SimpleScalar running sstrix compiled by GNU C
 # Cc1 defaults:
 # -mgas -mgpOPT
 # Cc1 arguments (-G value = 8, Cpu = default, ISA = 1):
 # -quiet -dumpbase -o

gcc2_compiled.:
__gnu_compiled_c:
        .globl  c
        .sdata
        .align  2
c:
        .word   255
        .text
        .align  2
        .globl  main

        .comm   i,4
        .comm   a,400
        .comm   b,400

        .text
```

```
        .loc    1 9
        .ent    main
main:
        .frame  $fp,24,$31              # vars= 0, regs= 2/0, args= 16, extra= 0
        .mask   0xc0000000,-4
        .fmask  0x00000000,0
        subu    $sp,$sp,24
        sw      $31,20($sp)
        sw      $fp,16($sp)
        move    $fp,$sp
        jal     __main
        sw      $0,i
$L2:    lw      $2,i
        sltu    $3,$2,100
        bne     $3,$0,$L5
        j       $L3
$L5:    lw      $2,i
        move    $3,$2
        sll     $2,$3,2
        la      $4,a
        addu    $3,$2,$4
        move    $2,$3
        lw      $3,i
        move    $4,$3
        sll     $3,$4,2
        la      $4,b
        addu    $3,$3,$4
        move    $4,$3
        lw      $3,0($4)
        lw      $4,c
        addu    $3,$3,$4
        sw      $3,0($2)
$L4:    lw      $3,i
        addu    $2,$3,1
        move    $3,$2
        sw      $3,i
        j       $L2
$L3:
$L1:    move    $sp,$fp                 # sp not trusted here
        lw      $31,20($sp)
        lw      $fp,16($sp)
        addu    $sp,$sp,24
        j       $31
        .end    main
>
```

a) Highlight the block of 5 instructions which builds a pointer to the `a[]` array.

b) Highlight a similar 5 instruction block which builds a pointer to the `b[]` array.

c) Highlight the 4 instruction block which adds the constant (255) into the element of `a[]` and placesthe result in `b[]`.

d) How would you write Pentium assembler to initialize a register with 255? Now, would you do the same in MIPS assembler?

e) Where does the stack frame get set up? What doe the `subu $sp,$sp,24` instruction achieve? What is being stored in $31? How can you confirm your idea?

f) How are conditional branch/jumps achieved in MIPS?

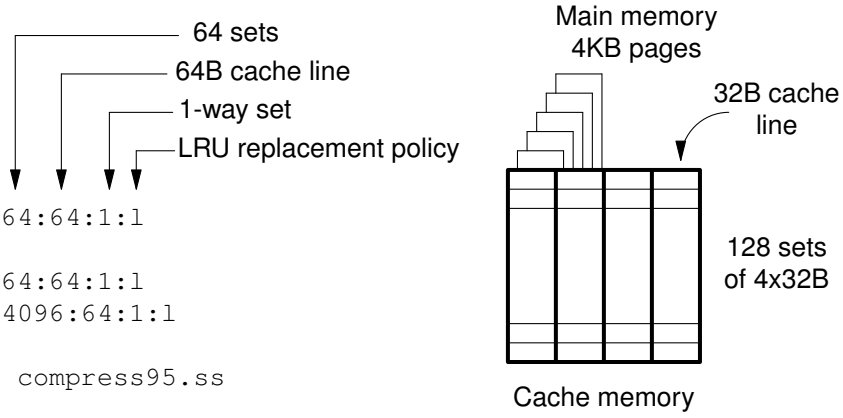
g) Is there a CPU Status Flags Register?

Trace out the way this code runs. Assume that `a` and `b` are arrays of words and the base address of `a` is held in register `$0` and the base address of `b` is held in register `$1`. Register `$2` is associated with variable `i` and register `$3` with `c`.

Using the C-code as a template, hand write your own version of the assembly code. Use the compiler/assembler (gcc/gas) provided with the SimpleScalar tool kit translate the assembly code to executable code forthe simulator.

Cache Configuration

The two separate L1 caches are set for default sizes of 8kB with a single unified L2 of 256 kB. The TLBs have 64 and 128 entries. Different values can be set as follows:




```
sim-cache -cache:il1 il1:64:64:1:l
          -cache:il2 dl2
          -cache:dl1 dl1:64:64:1:l
          -cache:dl2 dl2:4096:64:1:l
          -tlb:itlb none
          -tlb:dtlb none  compress95.ss
```

or

```
          -itlb:16:4096:4:1      (64 x 4-way set gives 64 entries)
          -dtlb:32:4096:4:1      (32 x 4-way set gives 128 entries)
```

So 64 x 64 x 1 = 4kB cache size. A single 8k unified cache can be specified instead of separate data and instruction caches by the following:

```
sim-cache -cache:il1 dl1
          -cache:dl1 dl1:128:64:1:l
          -cache:dl2 none
          -cache:il2 none
          -tlb:itlb none
          -tlb:dtlb none  compress95.ss
```

or more convenenently, using a script file to hold all the flag options:

```
sim-cache -config test_cache.cfg compress95.ss
```

The cache size parameters are arranged as:

Name : #sets : cache_line_size : associative# : replacement_policy (l, f, r) (l-LRU, f-FIFO, r-random)

Simulation

Do a full compile and build on the testit.c code and then use the
sim-safe simulator to run the output code, collecting the monitored data in a log file:

```
> sim-safe testit >& /tmp/testit.out
> more
  sim-safe: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
  Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
  All Rights Reserved. This version of SimpleScalar is licensed for academic
  non-commercial use.  No portion of this work may be used by any commercial
  entity, or for any commercial purpose, without the prior written permission
  of SimpleScalar, LLC (info@simplescalar.com).

  sim: command line: /usr/local/simplesim/simplesim-3.0/sim-safe testit

  sim: simulation started @ Sat Sep 30 12:53:38 2006, options follow:

  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.

  # -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
  # -chkpt                <null> # restore EIO trace execution from <fname>
  # -redir:sim            <null> # redirect simulator output to file (non-interactive only)
  # -redir:prog           <null> # redirect simulated program output to file
  -nice                     0 # simulator scheduling priority
  -max:inst                 0 # maximum number of inst's to execute

  sim: ** starting functional simulation **

  sim: ** simulation statistics **
  sim_num_insn                7136 # total number of instructions executed
  sim_num_refs                4024 # total number of loads and stores executed
  sim_elapsed_time               1 # total simulation time in seconds
  sim_inst_rate          7136.0000 # simulation speed (in insts/sec)
  ld_text_base          0x00400000 # program text (code) segment base
  ld_text_size               22960 # program text (code) size in bytes
  ld_data_base          0x10000000 # program initialized data segment base
  ld_data_size                4096 # program init'ed '.data' and uninit'ed '.bss' size
  ld_stack_base         0x7fffc000 # program stack segment base (highest address in sta
  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 end
  mem.page_count                12 # total number of pages allocated
  mem.page_mem                  48k # total size of memory pages allocated
  mem.ptab_misses               12 # total first level page table misses
  mem.ptab_accesses         182004 # total page table accesses
  mem.ptab_miss_rate        0.0001 # first level page table miss rate
```

a) How many machine instructions are executed during the running of the code?

b) How many memory data references are made during its execution?

c) Now use the gcc compiler provided with the SimpleScalar tool kit to compile the C code equivalent into MIPS binary code. Use the sim-safe simulator to run your code.

What are the principal differences between the generated results from the hand coded assembler and C code? If there is any difference, why are there different instructions, or more specificaly, different memory references?

d) Now try downloading source code for the game **go** from a GNU ftp site, unzip and extract the program from the tar ball (gnugo-3.4.tar.gz):

```
> ftp ftp.gnu.org

ftp> cd gnu/gnugo
ftp> get gnugo-3.4.tar.gz /tmp/gnugo.tgz
ftp> quit
> cd /tmp
> tar -zvxf gnugo.tgz
>
```

e) The sim-outorder module simulates, in a cycle-by-cycle mode, a pipelined, speculative, out of order (OoO) CPU with a full memory hierarchy (L1, L2 and main memory), and computes a wide range of information on the execution time and performance of each section of the processor. Sim-outorder is an execution driven simulator, which means that it *executes* the program that it is given, step-by-step, reading and writing data as it goes through the program. he alternative scheme of a trace driven simulator, uses a list of all executed instructions taken from a previous run, as a trace, and determines performance statistics from that. Trace driven simulators are unable to track mispredicted branches, cache misses, and other execution-time events because the instruction trace does not include them. Simplescalar is an execution-driven simulator and so it gives a more complete picture of the new CPU design as it executes.

Running sim-outorder without arguments will display a few pages of parameters and flags that can be passed to the simulator to modify its behavior. The parameters will scroll by too quickly for mere mortals to read, so try redirecting its output to a file using:`>&` and then browsing that file:

```
> sim-outorder >& /tmp/sim.out
> more /tmp/sim_out
>
```

As you become more familiar with Simplescalar and with computer architectures in general, more of the options will make sense.

The parameters you pass to sim-outorder will depend on the type of data you want to generate. For example, say you would like to compare how the execution time of a program will change if the L1 cache latency is doubled. In this case, you need to run two simulations, one with the baseline L1 cache latency, and a second with twice the latency of the first. Cache configuration options will be discussed in detail later; for now, know that we can obtain the data we need by running the following two simulations:

```
> sim-outorder -cache:il1lat1 -cache:dl1lat1 testit >& testit_1.out
>
> sim-outorder -cache:il1lat2 -cache:dl1lat2 testit >& testit_2.out
```

The final argument to sim-outorder (before the redirection) is the test program that you would like to run on the simulated chip. Typically, you will run a few programs chosen from known benchmark suites such as SPECint and SPECfp, which will be provided. If you need to run other benchmarks which do not have precompiled Simplescalar binaries, the program must be compiled from source for the Simplescalar ISA

using the Simplescalar targeted version of gcc (which is called ss-little-na-sstrix for the Pentium host).

Some other useful sim-outorder command line options are:

-nice X Sets the simulator priority to X (for running in the background).

-max:inst n Executes upto n instructions, then stops.

-fastfwd n Does not calculate simulation stats for the first n instructions.

-cache:xly <config> Specifies the cache configuration.

-cache:xlylat N Specifies the cache latency in number of cycles

-mem:lat N M Specifies the DRAM latency (N: first byte, M: next bytes).

Always remember to set the DRAM latency to a reasonable value (e.g. M: 100, N:1).
A config file facility is available to hold all the configuration data. See the current status of a simulator by:

```
> sim-safe -dumpconfig -
  # load configuration from a file
  # -config
  # dump configuration to a file
  # -dumpconfig
  # print help message
  # -h                          false
  # verbose operation
  # -v                          false
  # enable debug message
  # -d                          false
  # start in Dlite debugger
  # -i                          false
  # random number generator seed (0 for timer seed)
  -seed                           1
  # initialize and terminate immediately
  # -q                          false
  # restore EIO trace execution from <fname>
  # -chkpt                     <null>
  # redirect simulator output to file (non-interactive only)
  # -redir:sim                 <null>
  # redirect simulated program output to file
  # -redir:prog                <null>
  # simulator scheduling priority
  -nice                           0
  # maximum number of inst's to execute
  -max:inst                       0

  sim-safe: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
  Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
  All Rights Reserved. This version of SimpleScalar is licensed for academic
  non-commercial use.  No portion of this work may be used by any commercial
  entity, or for any commercial purpose, without the prior written permission
  of SimpleScalar, LLC (info@simplescalar.com).

  error: no executable specified
  Usage: /usr/local/simplesim/simplesim-3.0/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 t
  -q               <true|false>     #        false # initialize and terminate immediately
  -chkpt           <string>         #       <null> # restore EIO trace execution from <fna
  -redir:sim       <string>         #       <null> # redirect simulator output to file (no
  -redir:prog      <string>         #       <null> # redirect simulated program output to
  -nice            <int>            #            0 # simulator scheduling priority
  -max:inst        <uint>           #            0 # maximum number of inst's to execute
>
```

To load new configuration data, either save a config file:

```
> sim-safe -dumpconfig sim-safe.config

Then edit the contents and reload it for the next simulation run:

>>
> sim-safe -config sim-safe_2.config testit >& testit.run-data
>
```

Now try to configure Simplescalar to run using a 128 4-way set, 32kB split (16kB data/16kB instruction) L1 cache with a 3 cycle latency and 32 byte line size (block size), and a 4-way, 512kB unified L2 cache with a 27 cycle latency and 64 byte block size. Assume both caches are use LRU replacement.

The solution is:

```
-cache:dl1 dl1:128:32:4:l -cache:dl1lat 3 -cache:il1 il1:128:32:4:l
-cache:il1lat 3 -cache:il2 dl2 -cache:il2lat 27 -cache:dl2 dl2:2048:64:4:l
-cache:dl2lat 27
```

Changing the SimpleScalar Architecture

The machine instruction definitions are held in a single file in the following format;

```
DEINST (ADDI,                    ;
        0x41,                    ; opcode value
        "addi",                  ; opcode mnemonic
        "t, s, i",               ; fields
        IntALU,                  ; FU requirements
        F_ICOMP | F_IMM,         ; Instruction flags
        GPR(RT),                 ; Output dependences
        NA,
        GPR(RS),                 ; Input dependences
        NA,
        NA,
        SET_GPR(RT, GPR(RS)+IMM) ; semantics
        )
```

The standard components comprising the simulator are:

bpred.h .c	branch prediction
cache.h .c	cache operation module
eventq.h .c	event queue management
libcheetah	Cheetah cache simulation library
ptrace.h .c	pipetrace module
res.h .c	resource manager module
sim.h	main code interface definitions
textprof.pl	test segment profile view (Perl)
pipeview.pl	pipetrace view (Perl)

dlite.h .c	lightweight debugger
eio.h .c	external I/O tracing module
loader.h .c	program loader
memory.h .c	flat memory space module
regs.h .c	register module
machine.h .c	target and ISA-dependent routines
machine.def	SimpleScalar ISA definition
symbol.h .c	symbol table module
syscall.h .c	proxy system call implementation

eval.h .c	generic expression evaluator
libexo	EXO(-skeletal) persistent data structure library
misc.h .c	everything miscellaneous
options.h .c	options package
range.h .c	range expression package
stats.h .c	statistics package

Then the gcc compiler suite will need reconfiguring for the new ISA.

Read:

http://wwww.simplescalar.com

MIPS Instruction Reference

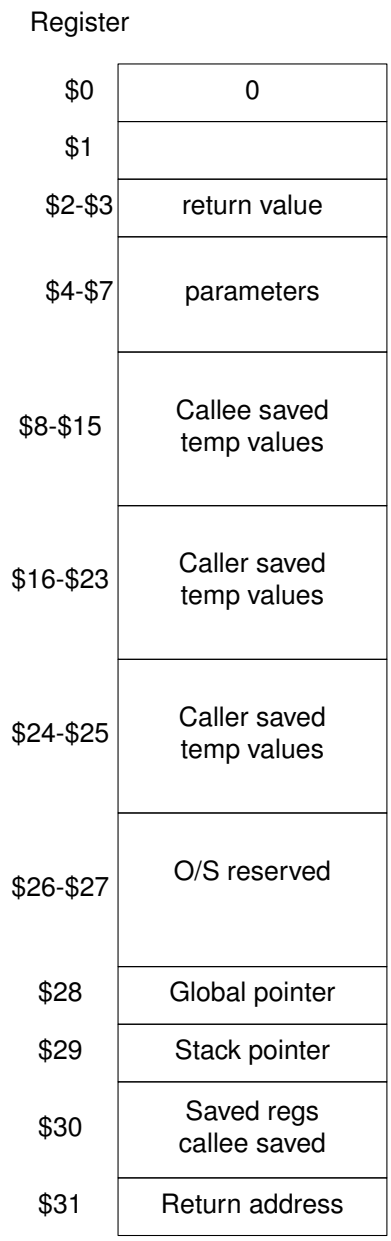
Here is a descriptive listing of the MIPS registers and instructions, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens indicate that those bits don't get decoded for that instruction. General purpose registers (GPRs) are indicated with a dollar sign ($). The words SWORD and UWORD refer to 32-bit signed and 32-bit unsigned data types, respectively. The manner in which the processor executes an instruction and advances its program counters is as follows:

1. execute the instruction at PC
2. copy nPC to PC
3. add 4 or the branch offset to nPC

This behavior is indicated in the instruction specifications below. For brevity, the function advance_pc (int) is used in many of the instruction descriptions. This function is defined as follows:

```
void advance_pc (SWORD offset)
{
                    PC  =  nPC;
                   nPC  += offset;
}
```

Note: ALL immediate values should be sign extended. After that, you treat them as signed or unsigned 32 bit numbers. For the non-immediate instructions, the only difference between signed and unsigned instructions is that signed instructions can generate an overflow.



The instruction descriptions are given below:

ADD -- /Add/

Description: Adds two registers and stores the result in a register
Operation: $d = $s + $t; advance_pc (4);
Syntax: add $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0000|

ADDI -- /Add immediate/

Description: Adds a register and a signed immediate value and stores the result in a register
Operation: $t = $s + imm; advance_pc (4);
Syntax: addi $t, $s, imm
Encoding: |0010 00ss ssst tttt iiii iiii iiii iiii|

ADDIU -- /Add immediate unsigned/

Description: Adds a register and an unsigned immediate value and stores the result in a register
Operation: $t = $s + imm; advance_pc (4);
Syntax: addiu $t, $s, imm
Encoding: |0010 01ss ssst tttt iiii iiii iiii iiii|

ADDU -- /Add unsigned/

Description: Adds two registers and stores the result in a register
Operation: $d = $s + $t; advance_pc (4);
Syntax: addu $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0001|

AND -- /Bitwise and/

Description: Bitwise ands two registers and stores the result in a register
Operation: $d = $s & $t; advance_pc (4);
Syntax: and $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0100|

ANDI -- /Bitwise and immediate/

Description: Bitwise ands a register and an immediate value and stores the result in a register
Operation: $t = $s & imm; advance_pc (4);
Syntax: andi $t, $s, imm
Encoding: |0011 00ss ssst tttt iiii iiii iiii iiii|

BEQ -- /Branch on equal/

Description: Branches if the two registers are equal
Operation: if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Syntax: beq $s, $t, offset
Encoding: |0001 00ss ssst tttt iiii iiii iiii iiii|

BGEZ -- /Branch on greater than or equal to zero/

Description: Branches if the register is greater than or equal to zero
Operation: if $s >= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax: bgez $s, offset
Encoding: |0000 01ss sss0 0001 iiii iiii iiii iiii|

BGEZAL -- /Branch on greater than or equal to zero and link/

Description: Branches if the register is greater than or equal to zero and saves the return address in $31
Operation: if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax: bgezal $s, offset
Encoding: |0000 01ss sss1 0001 iiii iiii iiii iiii|

BGTZ -- /Branch on greater than zero/

Description: Branches if the register is greater than zero
Operation: if $s > 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax: bgtz $s, offset
Encoding: |0001 11ss sss0 0000 iiii iiii iiii iiii|

BLEZ -- /Branch on less than or equal to zero/

Description: Branches if the register is less than or equal to zero
Operation: if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax: blez $s, offset
Encoding: |0001 10ss sss0 0000 iiii iiii iiii iiii|

BLTZ -- /Branch on less than zero/

Description: Branches if the register is less than zero
Operation: if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax: bltz $s, offset
Encoding: |0000 01ss sss0 0000 iiii iiii iiii iiii|

BLTZAL -- /Branch on less than zero and link/

Description: Branches if the register is less than zero and saves the return address in $31
Operation: if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax: bltzal $s, offset
Encoding: |0000 01ss sss1 0000 iiii iiii iiii iiii|

BNE -- /Branch on not equal/

Description: Branches if the two registers are not equal
Operation: if $s != $t advance_pc (offset << 2)); else advance_pc (4);
Syntax: bne $s, $t, offset
Encoding: |0001 01ss ssst tttt iiii iiii iiii iiii|

DIV -- /Divide/

Description: Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation: $LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax: div $s, $t
Encoding: |0000 00ss ssst tttt 0000 0000 0001 1010|

DIVU -- /Divide unsigned/

Description: Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation: $LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax: divu $s, $t
Encoding: |0000 00ss ssst tttt 0000 0000 0001 1011|

J -- /Jump/

Description: Jumps to the calculated address
Operation: PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax: j target
Encoding: |0000 10ii iiii iiii iiii iiii iiii iiii|

JAL -- /Jump and link/

Description: Jumps to the calculated address and stores the return address in $31
Operation: $31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000)

| (target << 2);
Syntax: jal target
Encoding: |0000 11ii iiii iiii iiii iiii iiii iiii|

JR -- /Jump register/

Description: Jump to the address contained in register $s
Operation: PC = nPC; nPC = $s;
Syntax: jr $s
Encoding: |0000 00ss sss0 0000 0000 0000 0000 1000|

LB -- /Load byte/

Description: A byte is loaded into a register from the specified address.
Operation: $t = MEM[$s + offset]; advance_pc (4);
Syntax: lb $t, offset($s)
Encoding: |1000 00ss ssst tttt iiii iiii iiii iiii|

LUI -- /Load upper immediate/

Description: The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation: $t = (imm << 16); advance_pc (4);
Syntax: lui $t, imm
Encoding: |0011 11-- ---t tttt iiii iiii iiii iiii|

LW -- /Load word/

Description: A word is loaded into a register from the specified address.
Operation: $t = MEM[$s + offset]; advance_pc (4);
Syntax: lw $t, offset($s)
Encoding: |1000 11ss ssst tttt iiii iiii iiii iiii|

MFHI -- /Move from HI/

Description: The contents of register HI are moved to the specified register.
Operation: $d = $HI; advance_pc (4);
Syntax: mfhi $d
Encoding: |0000 0000 0000 0000 dddd d000 0001 0000|

MFLO -- /Move from LO/

Description: The contents of register LO are moved to the specified register.
Operation: $d = $LO; advance_pc (4);
Syntax: mflo $d
Encoding: |0000 0000 0000 0000 dddd d000 0001 0010|

MULT -- /Multiply/

Description: Multiplies $s by $t and stores the result in $LO.
Operation: $LO = $s * $t; advance_pc (4);
Syntax: mult $s, $t
Encoding: |0000 00ss ssst tttt 0000 0000 0001 1000|

MULTU -- /Multiply unsigned/

Description: Multiplies $s by $t and stores the result in $LO.

Operation: $LO = $s * $t; advance_pc (4);
Syntax: multu $s, $t
Encoding: |0000 00ss ssst tttt 0000 0000 0001 1001|

NOOP -- /no operation, SLL $0, $0, 0/

Description: Performs no operation.
Operation: advance_pc (4);
Syntax: noop
Encoding: |0000 0000 0000 0000 0000 0000 0000 0000|

OR -- /Bitwise or/

Description: Bitwise logical ors two registers and stores the result in a register
Operation: $d = $s | $t; advance_pc (4);
Syntax: or $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0101|

ORI -- /Bitwise or immediate/

Description: Bitwise ors a register and an immediate value and stores the result in a register
Operation: $t = $s | imm; advance_pc (4);
Syntax: ori $t, $s, imm
Encoding: |0011 01ss ssst tttt iiii iiii iiii iiii|

SB -- /Store byte/

Description: The least significant byte of $t is stored at the specified address.
Operation: MEM[$s + offset] = (0xff & $t); advance_pc (4);
Syntax: sb $t, offset($s)
Encoding: |1010 00ss ssst tttt iiii iiii iiii iiii|

SLL -- /Shift left logical /

Description: Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation: $d = $t << h; advance_pc (4);
Syntax: sll $d, $t, h
Encoding: |0000 00ss ssst tttt dddd dhhh hh00 0000|

SLLV -- /Shift left logical variable/

Description: Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation: $d = $t << $s; advance_pc (4);
Syntax: sllv $d, $t, $s
Encoding: |0000 00ss ssst tttt dddd d--- --00 0100|

SLT -- /Set on less than (signed)/

Description: If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation: if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
Syntax: slt $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 1010|

SLTI -- /Set on less than immediate (signed)/

Description: If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation: if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
Syntax: slti $t, $s, imm
Encoding: |0010 10ss ssst tttt iiii iiii iiii iiii|

SLTIU -- /Set on less than immediate unsigned/

Description: If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation: if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
Syntax: sltiu $t, $s, imm
Encoding: |0010 11ss ssst tttt iiii iiii iiii iiii|

SLTU -- /Set on less than unsigned/

Description: If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation: if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
Syntax: sltu $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 1011|

SRA -- /Shift right arithmetic/

Description: Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation: $d = $t >> h; advance_pc (4);
Syntax: sra $d, $t, h
Encoding: |0000 00-- ---t tttt dddd dhhh hh00 0011|

SRL -- /Shift right logical/

Description: Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation: $d = $t >> h; advance_pc (4);
Syntax: srl $d, $t, h
Encoding: |0000 00-- ---t tttt dddd dhhh hh00 0010|

SRLV -- /Shift right logical variable/

Description: Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation: $d = $t >> $s; advance_pc (4);
Syntax: srlv $d, $t, $s
Encoding: |0000 00ss ssst tttt dddd d000 0000 0110|

SUB -- /Subtract/

Description: Subtracts two registers and stores the result in a register
Operation: $d = $s - $t; advance_pc (4);
Syntax: sub $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0010|

UBU -- /Subtract unsigned/

Description: Subtracts two registers and stores the result in a register
Operation: $d = $s - $t; advance_pc (4);
Syntax: subu $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d000 0010 0011|

SW -- /Store word/

Description: The contents of $t is stored at the specified address.
Operation: MEM[$s + offset] = $t; advance_pc (4);
Syntax: sw $t, offset($s)
Encoding: |1010 11ss ssst tttt iiii iiii iiii iiii|

SYSCALL -- /System call/

Description: Generates a software interrupt.
Operation: advance_pc (4);
Syntax: syscall
Encoding: |0000 00-- ---- ---- ---- ---- --00 1100|

XOR -- /Bitwise exclusive or/

Description: Exclusive ors two registers and stores the result in a register
Operation: $d = $s ^ $t; advance_pc (4);
Syntax: xor $d, $s, $t
Encoding: |0000 00ss ssst tttt dddd d--- --10 0110|

XORI -- /Bitwise exclusive or immediate/

Description: Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation: $t = $s ^ imm; advance_pc (4);
Syntax: xori $t, $s, imm
Encoding: |0011 10ss ssst tttt iiii iiii iiii iiii|