Sunflower Emulator Manual

University of Cambridge
Department of Engineering
Physical Computation Laboratory

Phillip Stanley-Marbell

COMPUTER PROGRAMS ARE FORMULATED IN A PROGRAMMING LANGUAGE AND SPECIFY CLASSES OF COMPUTING PROCESSES.

COMPUTERS, HOWEVER, INTERPRET SEQUENCES OF PARTICULAR INSTRUCTIONS, BUT NOT PROGRAM TEXTS.

—NIKLAUS WIRTH, *COMPILER CONSTRUCTION*, ADDISON-WESLEY, 1996.

PHILLIP STANLEY-MARBELL

SUNFLOWER EMULATOR MANUAL

SUNFLOWER-1.1
PHILLIP.STANLEY-MARBELL@ENG.CAM.AC.UK

First printing, 1999.

Contents

Overview and Installation 15
- 0.1 Overview . . . 16
- 0.2 Licensing . . . 18
- 0.3 Conventions . . . 18

1 Installation 21
- 1.1 Obtaining the Sources . . . 21
- 1.2 Obtaining the Source via Git . . . 22
- 1.3 Building the Emulator and Cross-Compilers . . . 22
- 1.4 Compiling the Emulator . . . 23
- 1.5 Compiling the Compiler . . . 24

2 Getting Started 25
- 2.1 Compiling Applications . . . 25
- 2.2 Running the Emulator . . . 25
- 2.3 Emulator Command Language . . . 27
- 2.4 The Emulator User Interface . . . 30

3 Loading and Running a Single Application 33
- 3.1 Simple Example: A C-Language Bubble Sort Implementation . 33
- 3.2 Running the Compiled `bubblesort` Application . . . 34

4 Modeling Processor Cores 41
- 4.1 Processor Cores . . . 41
- 4.2 Processing Nodes . . . 43
- 4.3 Built-in Assembler . . . 43

5 Power Estimation, Electrochemical Cells, and Voltage Regulators 45
- 5.1 Computation Power Estimation . . . 45
- 5.2 Non-ideal Power Sources and Voltage Regulators . . . 47

6 Interconnect and Network Modeling 51

6.1 Instantiating Network Media: NETNEWSEG 52
6.2 Instantiating Network Interfaces: **netnodenewifc** 52
6.3 Saving and Loading Network Traces: **netseg2file** and **file2netseg** 53
6.4 Configuring Network Media Signal Propagation Properties . . 53
6.5 Example 54

7 Fault Modeling **57**

8 Environment Models **59**
8.1 Node Location, Orientation, and Trajectory Definition 60
8.2 Defining Signal Sources and Signal Interactions/Interference . 60

9 Extended Example **61**
9.1 A Software-Defined Radio Application 61
9.2 Interaction Between Applications and Low-Level Machine State 62
9.3 Implementation of Software Radio Application 68
9.4 System Architecture Setup for Software Radio Application . . . 70

10 Non-Uniform Memory Accesses Latencies, Memory Remapping, and Memory Tracing **75**
10.1 The **mmap** Command 75
10.2 The **numa*** Commands 75

11 Stochastic Processes in Emulation and Simulation Experiments **77**
11.1 Generating Random Variates from Different Distributions . . . 77
11.2 User-Defined Discrete Distributions 78
11.3 Configuration Constants and Implementation Variables 78

12 Input and Output File Formats **79**
12.1 The Configuration File **conf/setup.conf** 79
12.2 The Configuration File **sim/config.h** 81
12.3 The Configuration Files **sim/config.$OSTYPE** 81
12.4 Architecture Specification Files 82
12.5 The Output Log File **sunflower.out** 82
12.6 Signal Source Sample Values File 82
12.7 Signal Source Trajectory File 82
12.8 Node Location Trajectory File 83
12.9 Network Trace Log File 83

13 Cross-Compilation Toolchain **85**
13.1 Miscellaneous Notes and Pointers 85

14 Benchmarks **87**
14.1 The SPEC CPU 2000 Benchmarks 87

14.2 The CMU Sphynx3 Speech Recognition Benchmark 87
14.3 The MPEG2 Encoder Benchmark 88
14.4 The MPEG Decoder Benchmark 88
14.5 The MiBench Benchmark Suite 88
14.6 The ALPBench Suite 88
14.7 A Sensor Network Benchmark Suite 88
14.8 The Software-Defined Radio Benchmark 88
14.9 The Sunflower Pthreads Subset Implementation and Software-Defined Radio Pthreads implementation 88

15 Utilities **89**
15.1 `logmarkparse` 89

Appendices **91**

Appendix A Frequently answered questions **93**
A.1 Defining complex memory maps with different memory access latencies 93
A.2 Extracting the archives downloaded from the web page 93
A.3 General problems compiling the tools 93
A.4 Behavior of Sunflower — “nothing happens” 94
A.5 Crashing benchmarks — function calls in interrupt handler . . 94
A.6 Relation between **`CLK`** and **`ICLK`** 94
A.7 Adding new memory-mapped registers to modeled machine . 95
A.8 Compiling the SPEC CPU 2000 benchmarks 96
A.9 What is **`NIC_OUI`**? 96
A.10 Adding new memory-mapped registers to the Hitachi SH architecture 97
A.11 Changing voltage/frequency from within applications running over simulator 97
A.12 Why does the swradio benchmark stop after 1024 samples ? . . 98
A.13 Errors opening .sr files in the software-defined radio example . 98
A.14 Simulation stopped with a “FATAL” message 98
A.15 Voltage and frequency scaling model 100
A.16 Implementing real-time applications, dynamic voltage scaling (DVS) and low-power idling 101
A.17 Porting new benchmarks to the simulator 102
A.18 Modeled costs of voltage and frequency scaling 102
A.19 Energy model 103
A.20 The setquantum command 103
A.21 Getting the current program counter (PC), frequency and supply voltage 104

A.22 Instruction latencies 104
A.23 Application using timer peripheral on Hitachi SH sleeps forever 104
A.24 Non-interactive simulation 105
A.25 Calculating instructions per cycle (IPC) 105
A.26 Accessing the arguments supplied to the run command from applications 106
A.27 Adapting the simulator's dynamic voltage scaling (DVS) latency modeling 106
A.28 Adding new commands to simulator command language . . . 107
A.29 Setting the different simulation modes — **fast functional**, *cycle-accurate*, **bit-flip analysis** and so on 107
A.30 Modeling custom hardware blocks 108
A.31 Configuring on-chip communication topologies 109
A.32 Bus arbitration when modeling on-chip networks 109
A.33 Functional, versus instruction-level, versus cycle-accuracy . . . 109
A.34 Application partitioning 110
A.35 Multiple applications on one processor core 110

Appendix B Implementation Overview **111**
B.1 **LICENSE.txt** 111
B.2 **sf.h** 112
B.3 **arch-Inferno.c** 112
B.4 **arch-OpenBSD.c** 112
B.5 **arch-darwin.c** 112
B.6 **arch-linux.c** 112
B.7 **arch-solaris.c** 112
B.8 **utils/batt-test.c** 112
B.9 **batt.c** 113
B.10 **batt.h** 113
B.11 **battmodels/** 113
B.12 **big-endian-hitachi-sh.h**, **little-endian-hitachi-sh.h** . . . 113
B.13 **bit.h** 113
B.14 **bit-utils.c** 113
B.15 **cache-hitachi-sh.c** 114
B.16 **cache-hitachi-sh.h** 114
B.17 **decode-hitachi-sh.c** 114
B.18 **decode-hitachi-sh.h** 114
B.19 **decode-ti-msp430.h** 114
B.20 **dev7708.c** 114
B.21 **dev7708.h** 114
B.22 **dev430x1xxx.c** 115
B.23 **dev430x1xxx.h** 115

B.24 **devsim7708.c** . . . 115
B.25 **devsim7708.h** . . . 115
B.26 **devsunflower.c** . . . 115
B.27 **endian-hitachi-sh.h** . . . 115
B.28 **fault.c** . . . 115
B.29 **fault.h** . . . 116
B.30 **fdr.c** . . . 116
B.31 **fdr.h** . . . 116
B.32 **mfns.h** . . . 116
B.33 **instr-hitachi-sh.h** . . . 116
B.34 **interrupts-hitachi-sh.h** . . . 116
B.35 **interrupts-ti-msp430.h** . . . 116
B.36 **lex.c** . . . 116
B.37 **machine-hitachi-sh.c**, **machine-hitachi-sh.h** . . . 117
B.38 **machine-ti-msp430.c**, **machine-ti-msp430.h** . . . 117
B.39 **main.c** . . . 117
B.40 **main.h** . . . 118
B.41 **mkhelp** . . . 118
B.42 **mkmantex** . . . 118
B.43 **mkopstr-hitachi-sh** . . . 118
B.44 **mkopstr-ti-msp430** . . . 118
B.45 **network-hitachi-sh.c** . . . 118
B.46 **network-hitachi-sh.h** . . . 119
B.47 **op-hitachi-sh.c** . . . 119
B.48 **op-hitachi-sh.h** . . . 119
B.49 **op-ti-msp430.c** . . . 119
B.50 **op-ti-msp430.h** . . . 119
B.51 **pipeline-hitachi-sh.c** . . . 119
B.52 **pipeline-hitachi-sh.h** . . . 119
B.53 **pipeline-ti-msp430.c** . . . 120
B.54 **pipeline-ti-msp430.h** . . . 120
B.55 **power.c** . . . 120
B.56 **randgen.c** . . . 120
B.57 **randgen.h** . . . 120
B.58 **regaccess-hitachi-sh.c** . . . 120
B.59 **regaccess-ti-msp430.c** . . . 120
B.60 **pau.c** . . . 120
B.61 **pau.h** . . . 120
B.62 **pic.c** . . . 121
B.63 **pic.h** . . . 121
B.64 **pipeline-hitachi-sh.c** . . . 121

B.65 `pipeline-hitachi-sh.h` . . . 121
B.66 `power.h` . . . 121
B.67 `regs-hitachi-sh.h` . . . 121
B.68 `regs-ti-msp430.h` . . . 121
B.69 `sf.y` . . . 121
B.70 `syscalls.c`, `syscalls.h`, `syscalls-Inferno.c` . . . 122
B.71 `tag.c` . . . 122
B.72 `tokenhandling.c` . . . 122

Appendix C Sunflower Commands **123**
C.1 `ADDVALUETRACE` . . . 123
C.2 `BATTALERTFRAC` . . . 123
C.3 `BATTCF` . . . 123
C.4 `BATTETALUT` . . . 123
C.5 `BATTETALUTNENTRIES` . . . 123
C.6 `BATTILEAK` . . . 124
C.7 `BATTINOMINAL` . . . 124
C.8 `BATTNODEATTACH` . . . 124
C.9 `BATTRF` . . . 124
C.10 `BATTSTATS` . . . 124
C.11 `BATTVBATTLUT` . . . 124
C.12 `BATTVBATTLUTNENTRIES` . . . 124
C.13 `BATTVLOSTLUT` . . . 124
C.14 `BATTVLOSTLUTNENTRIES` . . . 125
C.15 `BPT` . . . 125
C.16 `BPTDEL` . . . 125
C.17 `BPTLS` . . . 125
C.18 `C` . . . 125
C.19 `CA` . . . 125
C.20 `CACHEINIT` . . . 125
C.21 `CACHEOFF` . . . 125
C.22 `CACHESTATS` . . . 126
C.23 `CD` . . . 126
C.24 `CLOCKINTR` . . . 126
C.25 `CONT` . . . 126
C.26 `D` . . . 126
C.27 `DEFNDIST` . . . 126
C.28 `DELVALUETRACE` . . . 126
C.29 `DUMPALL` . . . 126
C.30 `DUMPMEM` . . . 127
C.31 `DUMPPIPE` . . . 127
C.32 `DUMPREGS` . . . 127

C.33 DUMPSYSREGS 127
C.34 DUMPTLB 127
C.35 DYNINSTR 127
C.36 EBATTINTR 127
C.37 EFAULTS 127
C.38 FF 128
C.39 FILE2NETSEG 128
C.40 FLTTHRESH 128
C.41 FORCEAVGPWR 128
C.42 GETRANDOMSEED 128
C.43 HELP 128
C.44 HWSEEREG 128
C.45 IGN 128
C.46 INITRANDTABLE 129
C.47 INITSEESTATE 129
C.48 L 129
C.49 LISTRVARS 129
C.50 LOAD 129
C.51 LOCSTATS 129
C.52 MALLOCDEBUG 129
C.53 MAN 129
C.54 MMAP 130
C.55 N 130
C.56 NANOPAUSE 130
C.57 ND 130
C.58 NETCORREL 130
C.59 NETDEBUG 130
C.60 NETNEWSEG 130
C.61 NETNODENEWIFC 131
C.62 NETSEG2FILE 131
C.63 NETSEGDELETE 131
C.64 NETSEGFAILDURMAX 131
C.65 NETSEGFAILPROB 131
C.66 NETSEGFAILPROBFN 131
C.67 NETSEGNICATTACH 131
C.68 NETSEGPROPMODEL 132
C.69 NEWBATT 132
C.70 NEWNODE 132
C.71 NI 132
C.72 NODEFAILDURMAX 132
C.73 NODEFAILPROB 132

C.74 NODEFAILPROBFN . . . 132
C.75 NUMAREGION . . . 133
C.76 NUMASETMAPID . . . 133
C.77 NUMASTATS . . . 133
C.78 NUMASTATSALL . . . 133
C.79 OFF . . . 133
C.80 ON . . . 133
C.81 PARSEOBJDUMP . . . 133
C.82 PAUINFO . . . 134
C.83 PAUSE . . . 134
C.84 PCBT . . . 134
C.85 PD . . . 134
C.86 PE . . . 134
C.87 PF . . . 134
C.88 PFUN . . . 134
C.89 PI . . . 134
C.90 POWERSTATS . . . 135
C.91 POWERTOTAL . . . 135
C.92 PS . . . 135
C.93 PWD . . . 135
C.94 Q . . . 135
C.95 QUIT . . . 135
C.96 R . . . 135
C.97 RANDPRINT . . . 135
C.98 RATIO . . . 136
C.99 REGISTERRVAR . . . 136
C.100 REGISTERSTABS . . . 136
C.101 RENUMBERNODES . . . 136
C.102 RESETALLCTRS . . . 136
C.103 RESETCPU . . . 136
C.104 RESETNODECTRS . . . 136
C.105 RETRYALG . . . 137
C.106 RUN . . . 137
C.107 SAVE . . . 137
C.108 SENSORSDEBUG . . . 137
C.109 SETBASENODEID . . . 137
C.110 SETBATT . . . 137
C.111 SETBATTFEEDPERIOD . . . 137
C.112 SETDUMPPWRPERIOD . . . 137
C.113 SETFAULTPERIOD . . . 138
C.114 SETFLASHRLATENCY . . . 138

C.115 SETFLASHWLATENCY . . . 138
C.116 SETFREQ . . . 138
C.117 SETIFCOUI . . . 138
C.118 SETLOC . . . 138
C.119 SETMEMRLATENCY . . . 138
C.120 SETMEMWLATENCY . . . 138
C.121 SETNETPERIOD . . . 139
C.122 SETNODE . . . 139
C.123 SETPC . . . 139
C.124 SETPHYSICSPERIOD . . . 139
C.125 SETQUANTUM . . . 139
C.126 SETRANDOMSEED . . . 139
C.127 SETSCALEALPHA . . . 139
C.128 SETSCALEK . . . 139
C.129 SETSCALEVT . . . 140
C.130 SETSCHEDRANDOM . . . 140
C.131 SETSCHEDROUNDROBIN . . . 140
C.132 SETTIMERDELAY . . . 140
C.133 SETVDD . . . 140
C.134 SFATAL . . . 140
C.135 SHAREBUS . . . 140
C.136 SHOWCLK . . . 140
C.137 SHOWPIPE . . . 141
C.138 SIGSRC . . . 141
C.139 SIGSUBSCRIBE . . . 141
C.140 SIZEMEM . . . 141
C.141 SPLIT . . . 141
C.142 SRECL . . . 141
C.143 STOP . . . 141
C.144 THROTTLE . . . 141
C.145 THROTTLEWIN . . . 142
C.146 TRACE . . . 142
C.147 V . . . 142
C.148 VALUESTATS . . . 142
C.149 VERBOSE . . . 142
C.150 VERSION . . . 142
C.151 NODETACH . . . 142
C.152 SIZEPAU . . . 142

Index **145**

Overview and Installation

Sunflower is an execution-driven full-system hardware emulator for networked embedded systems. It is intended for use in the modeling and study of single and multi-processor embedded systems and the environments in which they are deployed. Examples of systems that can be modeled with the emulator are illustrated in Figure 1.

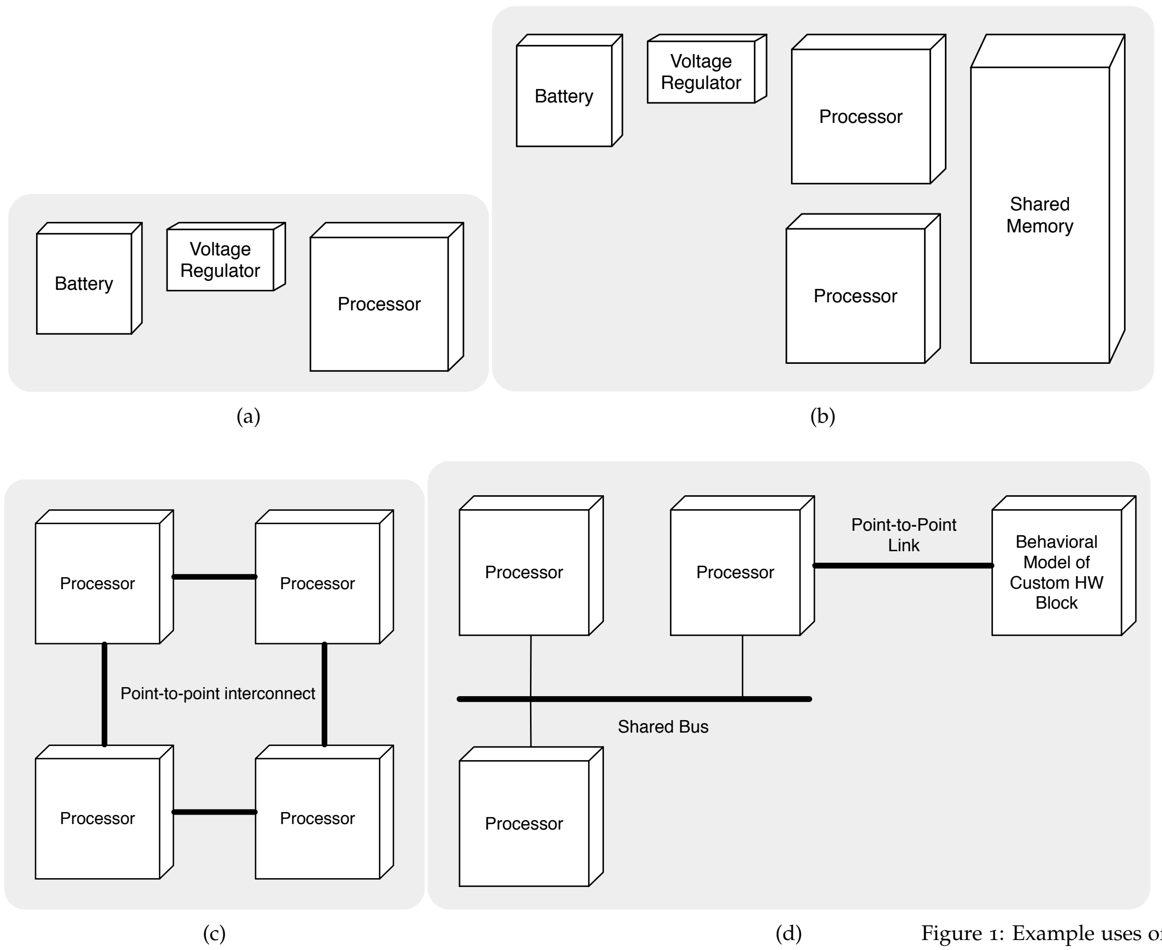


Figure 1: Example uses of the Sunflower emulator.

0.1 Overview

The possible components of a model to be emulated by Sunflower are illustrated in Figure 2. A *system architecture description file (ADF)* defines the components that make up the system, and the interconnections between them, such as the components in Figure 1. A simple system might define a single processor, a battery and a voltage regulator, in its system architecture description file; at the minimum, a system will contain at least one processing element (i.e., a processor or microcontroller) as all emulations are execution driven, and thus central to the evolution of time is the passage of clock cycles on one or more processors. Multiple processors may be instantiated in a given modeled system, and these processors may be linked together using either shared memory or explicit communication over interconnect links (message passing). Associated with each instantiated processor is an executable program to be loaded into the memory of the processing element. The programs are the output of compilation and linking with an appropriate cross-compiler tool-chain for the processor architectures modeled by the emulator (Chapter 13).

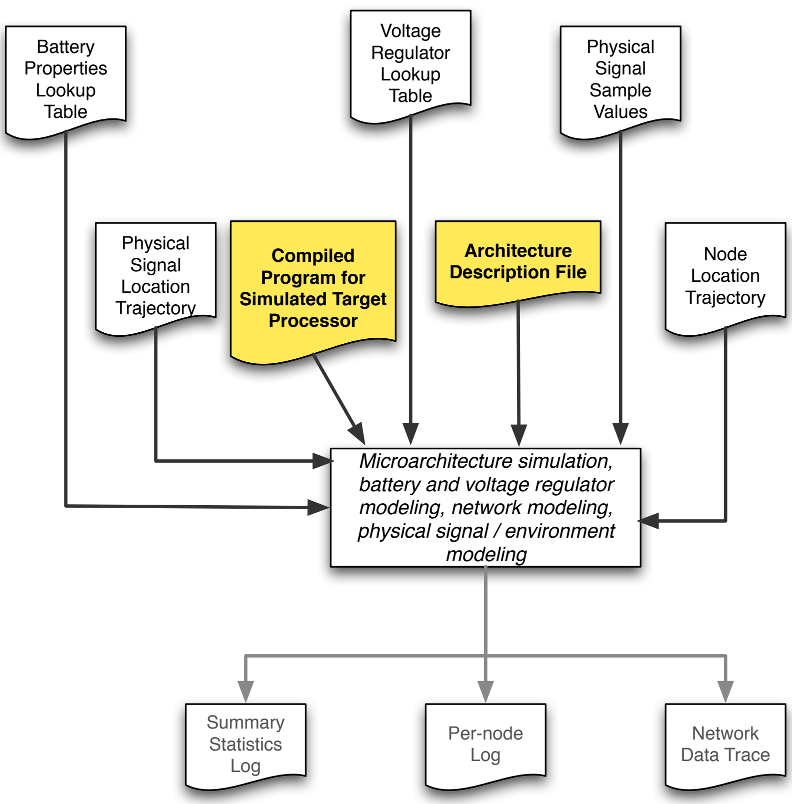


Figure 2: Inputs and outputs to the emulator.

The Sunflower emulator is intended for microarchitectural and system architecture exploration of embedded computing systems. An important aspect of embedded computing systems is the environments in which they are *embedded*, and the signals and phenomena that evolve over time in these environments. These may be electromagnetic modulated signals, sounds, temperature, or even the collective changes in these resulting from the motion of the system under study. The Sunflower emulator enables the definition and simulation of signals in the environs of modeled systems, the evolution in time, motion and interactions (constructive and destructive interference) between these signals, as well as the motion and directional orientation of the systems themselves. Modeling of signals in the environment of systems is achieved through the input of a signal sample value file (SVF) as well as a signal trajectory file (STF). These inputs to the emulator enable the definition of arbitrary signals and their evolution in time and location. A similar input file, the node location input file (LIF), defines the location, directional orientation and motion of nodes. The detailed specifications of these input files are discussed further in Chapter 12.

Together, the aforementioned inputs and configuration files define a system to be modeled by the Sunflower framework. The simulation of such a system proceeds by the cycle-by-cycle modeling of the instantiated processors within the systems, alongside the modeling of the evolution of signals in their environments, communications between systems, and so forth. The "output" of a simulation is dependent on the intent of the user of the system. At a minimum, each simulation results in a summary of machine state that is logged to a simulation output file (SOF), whose format and contents are detailed in Chapter 12. Other values that may be of interest may include captured network traffic traces (the file format for which is discussed in Chapter 12), traces of values taken on by source-level variables in the programs being executed over the modeled processors, and the statistics of values taken on by various internal counters in the emulator.

The Sunflower simulator is part of a larger suite of hardware and software tools intended for the design and exploration of networks of resource-constrained and failure-prone systems. The suite includes hardware platforms ranging from a energy-scavenging embedded system platform, to a 24-processor embedded multiprocessor, and a handheld portable computing device. These hardware platforms may be modeled within the emulator, and measurements taken on the platforms may similarly be used to calibrate properties of the simulation.

This manual is intended to provide an overview of the usage, as well as the design and implementation of the Sunflower emulator. The next chapter details the installation of the emulator, including binary-only GUI and command-line interfaces, as well as compiling the implementation of the simulator from its source.

0.2 Licensing

The simulator is distributed under a modified BSD license, which permits, in summary, the free copying of the source, for both commercial and non-commercial purposes, as long as the authors are credited, and the license terms are maintained. More information on the terms of the BSD license can be obtained from `http://www.opensource.org/licenses/bsd-license.php`.

0.3 Conventions

Input at a shell command prompt, absolute and relative paths and file names, are typeset in a `typewriter` typeface.

Simulator commands are shown in a shaded text region, (with the name of the relevant manual section in parenthesis, hyper-linked, to the relevant section in the manual page appendix in this document) and with "keyboard icon", such as **off** (C.79) and **help** (C.43) for the commands to turn the simulator off and to obtain on-line help directly from the simulator. Aggregates of commands and their parameters, such as the command to issue to obtain information on all commands beginning with the prefix "net", **man net*** are similarly displayed but not hyper-linked. Commands which are specific to a given processor architecture, i.e., assembly language commands, are shown in bold upper case, e.g., **MOV.L**. In-line references to variables, and data structures from the simulator implementation are shown shaded with an icon of a "paper stack", such as **Engine** to refer to the `Engine` data structure in the simulator implementation. Likewise, inline references to the source implementation of the benchmarks supplied with the simulator are shown shaded with a "weight-lifter" icon, such as **startup()** referring to the `startup()` function that most of the benchmarks implement. References to the simulator configuration parameters are similarly shown shaded, with a single "sheet" icon, such as **SF_SIMLOG** for the simulator logging configuration file parameter. Actual references to files in the simulator distribution (simulator source implementation files, configuration files, or benchmark files) are shown shaded with a "paper folder" icon, such as **sim/Makefile** for the Makefile in the directory `sim` from the root of the simulator source tree. The references are hyper-linked to the online source repository of the last revision of the simulator distribution for which the manual is valid. Commands to be issued at an operating system shell are shown shaded with a "blinking letter" icon, e.g., `make` for a reference to typing `make` at a shell command prompt.

Blocks of text relevant to the above categories are shown shaded, with the same icon scheme. Thus, for example, a snippet of a shell session transcript is shown as

```
[precision:~] pip% pwd
/Users/pip
[precision:~] pip%
```

Important information is shown with an exclamation mark in the margin, and should not be ignored!

Simulator command names: The names of the simulator commands often share a common prefix, denoting the type of command. Thus, for example, commands relating the configuring networks generally begin with **net**, such as the `netnewseg` (C.60) , `netcorrel` (C.58) and `netdebug` (C.59) commands. Thus, to obtain a list of commands related to a given topic, one may enter `man net*` at the simulator command prompt.

1
Installation

The Sunflower emulator can be obtained as pre-compiled binaries, or in source form. This chapter describes installation from the source, as the pre-compiled binaries need no further configuration.

1.1 Obtaining the Sources

The source archive for the emulator can be obtained from:

```
http://www.sflr.org
```

via the “Simulator/Hg Source Repository” section of the web page. This download is approximately 60 MB, and includes the source for the emulator, benchmark suites, and pre-compiled benchmarks. The source for the GCC cross-compiler and its associated packages (Binutils, Newlib) are not included, but instructions are provided for the specific steps to perform to download the necessary archives from the web.

For example, to uncompress and extract the archive bzipped version of the archive:

```
bunzip2 sunflower-1.0-release-source-beta.3.tar.bz2
tar -xvf sunflower-1.0-release-source-beta.3.tar
```

Some web browsers or download clients will automatically uncompress the archive upon download, and this might result in a file with an extension such as ".tar.bz2.tar", which is already uncompressed and can be extracted with the `tar` utility. Please consult your system manuals or system administrator if you have trouble figuring out how to uncompress the archive. Uncompressing the archive should create a directory, `sunflower-1.0-release-source-beta.3/`. All paths to files and directories in this manual will be specified relative to the root of the distribution, unless the relative location is deemed obvious from the context.

1.2 Obtaining the Source via Git

The emulator can also be obtained via anonymous access to a Git repository:

```
git clone git@github.com:phillipstanleymarbell/sunflower-simulator.git
```

The directory `tools/source` contains template directories into which the appropriate versions of the tools should be unpacked. For example, at the time of writing, the cross compilation tool sources required are:

```
shell$ ls sunflowersim/tools/source/

binutils-2.16.1      gcc-4.1.1   newlib-1.9.0

shell$
```

The `README.md` file at root of the emulator distribution details the steps needed to download the cross-compiler sources, for population of the template directories. The cross-compiler tools are not included in the emulator distribution due to their large size.

All the appropriate Makefiles and build steps to build the cross-compilers from these particular sources are already in place, and no further configuration other than extracting the sources for the packages into the appropriate directories is necessary.

1.3 Building the Emulator and Cross-Compilers

The emulator, compiler build and applications, all rely on a single configuration file, `conf/setup.conf`. You will need to modify the first line of this file to reflect your installation location. For example, if the emulator source is unpacked into the directory `/home/luser/sunflower-1.0-release-source-beta.3`, and your host operating system is OpenBSD 3.1 running on an Intel system, then the first few lines of `conf/setup.conf` will look like the following:

```
##
##      You will want to change the following to suit your setup:
##
SUNFLOWERROOT            = /home/luser/sunflower-1.0-release-source-beta.3

HOST            = i686-unknown-openbsd3.1
TARGET          = superH
TARGET-ARCH     = sh-coff
TARGET-ARCH-FLAGS = -DeEK32

##
##      You do not necessarily need to change this stuff:
```

```
##
GCCINCLUDEDIR   = $(SUNFLOWERROOT)/tools/source/gcc-4.1.1/gcc/ginclude/
PREFIX          = $(TOOLS)/$(TARGET)
```

The configuration string for the **HOST** field is easiest obtained by executing **gcc -v**, and is a string in the format *machine_architecture-vendor_name-operating_system*, e.g., **i686-pc-linux-gnu** (generic Linux) or **i686-unknown-openbsd3.1** (OpenBSD) or **ppc-unknown-darwin** (MacOS X on a PowerPC processor).

1.4 *Compiling the Emulator*

Once you have correctly edited the **SUNFLOWERROOT** and **HOST** fields of the configuration file, you should be able to build the emulator. The emulator source resides in the directory `sim/` from the root of the distribution. For OpenBSD, Darwin/OSX, Linux and Solaris, you should be able to compile the emulator by just typing `make OSTYPE=xyz MACHTYPE=abc`, where `xyz` is one of `darwin`, `OpenBSD`, `linux` or `solaris`, and **abc** is one of `i386`, `ppc`, `sparc`, for the eponymous systems. On many systems, the environment variables `OSTYPE` and `MACHTYPE` are already set, and the above steps may be redundant. For other host platforms, copy the file `config.posix` to a file whose name is `config.OSTYPE-MACHTYPE`, where **OSTYPE** is the value of the environment variable `$OSTYPE`, or an appropriately chosen system type if the environment variable is not set, likewise **MACHTYPE**. You might need to edit the `config.$OSTYPE-$MACHTYPE`.; the format and fields of the `config.$OSTYPE-$MACHTYPE` file are detailed in Chapter 12 (file formats), in Section 12.3. Experienced Unix users should find any necessary changes to the configuration file straightforward.

For performance reasons, the emulator implementation uses a few techniques which depend on the byte-order of the host machine (i.e. little- or big-endian). There is a flag in the `config.OSTYPE` file which must be set to reflect the architecture of the host machine. For little-endian host architectures (e.g., Intel x86 processors), the flag is `SF_L_ENDIAN` and for big-endian machines such as SPARC the flag should be set to `SF_B_ENDIAN`.

Portions of the source and headers for the emulator build are generated by a set of shell scripts: `mkhelp`, `mkmantex`, `mkopstr-hitachi-sh` and `mkopstr-ti-msp430`. These scripts depend on the presence of an installation of the Gnu version of awk (`gawk`). Gnu awk will likely be present on most systems. On systems where it is not, it should be easy to install. For example, on MacOS, it can be installed via MacPorts. The path to the Gnu awk is one of the variables in the `config.$OSTYPE-$MACHTYPE` file.

1.5 *Compiling the Compiler*

Once you have the emulator built, you may now proceed to compiling the cross-compiler. In order for you to use the compiler (GCC) to generate code for the target architectures (Hitachi SH and TI MSP430), you must compile GCC, configured to generate code for the appropriate target. Such a version of GCC is referred to as a *cross compiler*, as it runs on one target architecture (e.g. OpenBSD x86) and generates code for another (e.g. Hitachi SH, no OS).

Building a cross compiler can be a tedious process, however, a significant amount of work has been done already for you, so building GCC from the sources provided is simple. From the root of the Sunflower distribution, just type `make cross`. This will build the cross-compiler for the default target architecture (Hitachi SH), as defined in the **`conf/setup.conf`** file. Building the cross compiler for the MSP430 architecture is currently not integrated into the distribution's Makefiles, as it requires a patched version of GCC.

The build process for building the cross compiler assumes you have access to the gnu version of Make (`gmake`) in your path. On systems where this is not present, it can be easily installed. The necessary Makefiles have already been put in place to configure and build Binutils (the binary utility tool-suite that GCC depends on for assembling and linking), then GCC itself, and finally to use the freshly compiled GCC to build the standard libraries against which your programs will be linked (Newlib).

The compilation process will take a while, on the order of 30 minutes. Once it completes, you should have several files in the automatically created **`tools/bin`** directory of the Sunflower root:

```
devilbunny /tmp/sunflower-1.0-release-source-beta.3> ls
Makefile  conf  sim  sys  tools  tools-lib

devilbunny /tmp/sunflower-1.0-release-source-beta.3> ls tools/bin
sh-coff-addr2line  sh-coff-g77    sh-coff-objcopy   sh-coff-strings
sh-coff-ar         sh-coff-gasp   sh-coff-objdump   sh-coff-strip
sh-coff-as         sh-coff-gcc    sh-coff-protoize  sh-coff-unprotoize
sh-coff-c++        sh-coff-gprof  sh-coff-ranlib
sh-coff-c++filt    sh-coff-ld     sh-coff-readelf
sh-coff-g++        sh-coff-nm     sh-coff-size
```

The central configuration file previously described references these binaries for building applications to run over the emulator, so for the most part, you do not have to remember where they are or reference them directly for that matter.

2

Getting Started

A few example applications are provided with the simulator, and these reside in `benchmarks/source/`.

2.1 Compiling Applications

The directory `benchmarks/source/bubblesort` contains the source for the *bubblesort* example presented in Chapter 3.

Each example application under `benchmarks/source/` contains a Makefile. To construct applications of your own, it you might want to copy the entire contents of one of these directories to a new one, and make modifications as necessary.

2.2 Running the Emulator

When the simulator builds successfully, a binary, '`sf`', should be produced. You should be able to run it by typing `./sf`. The simulator can be scripted by providing it a *simulator command file* or *architecture specification file* as standard input or as its sole argument.

The simulator has an interactive interface. Starting the simulator instantiates a single processor, and attaches the interactive interface to it (see Figure 2.1). Commands typed at the interface are with respect to the currently attached processor. From the command interface, a user will typically issue commands to create new processors, new network interconnection links, load compiled binaries into the memory of instantiated processors, switch on or off a processor, etc. Rather than type in all the commands needed to setup a typical simulation from the command interface, a user may place all the necessary commands in a file and use the **`load`** (C.50) command to load it in.

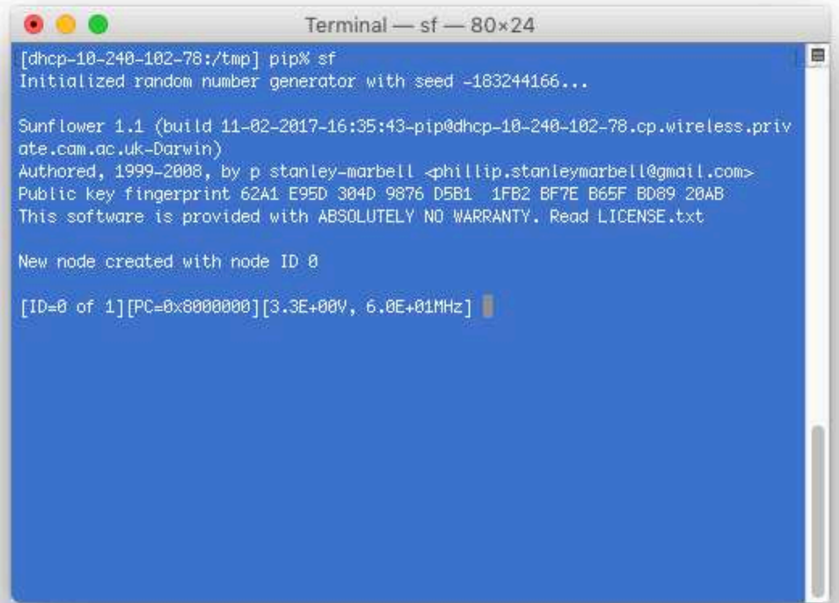


Figure 2.1: The interactive command interface for the Sunflower simulator.

2.3 *Emulator Command Language*

Commands entered at the simulator command prompt are generally used to setup and control simulations, probe the state of simulated processors and interconnection links etc. For example, given a C- language program compiled for one of the target architectures, the binary can be loaded into a processor node using the **srecl** ⌨ (C.142) command. Once the binary has been loaded into memory, the **run** ⌨ (C.106) command is issued to activate the processor to which it was loaded, and the **on** ⌨ (C.80) command issued to set the simulator running. At any time, the **off** ⌨ (C.79) command may be issued to pause the simulation. Other commands of common interest include **ni** ⌨ (C.71) for querying the number of instructions executed to date, **showclk** ⌨ (C.136) for seeing the current number of elapsed clock cycles and current global simulation time and **c** ⌨ (C.18) for seeing the current cache access statistics if a cache has been instantiated.

The command interface executes as a separate thread from the simulation engine. Thus entering any command at the command prompt brings you directly back to the prompt, while the command executes. !

Central to the use of the command interface is the concept of *attachment to a processor*. Multiple interconnected or independent processors may be instantiated at the command interface (using the **newnode** ⌨ (C.70) command). At any given moment, the command interface is *associated* with a particular processor instance. Thus, for example, you can initiate execution on an instantiated processor, and then issue commands to probe the state of the processor while it executes (in the background). Commands for probing machine state include the **dumpregs** ⌨ (C.32) command for displaying the contents of the register file. !

In addition to such commands for controlling execution, the command interface also acts as an assembler for the architecture of the processor instance to which it is connected, thus any valid assembler mnemonic may be entered at the command line. For example, entering `MOV #4, R5` at the command interface attached to a Hitachi SH processor instance, will set the contents of register `R5` of currently attached processor to the value 4.

Example simulation configuration files are included with the most of the benchmarks, e.g., `benchmarks/source/swradio/swr.m` 🗁. By convention, simulator configuration files have the suffix ".m". To get a quick feel for the command language, browse through such simulator configuration files, and match the commands therein to entries in the appendix. The **help** ⌨ (C.43) command lists all available commands (see Figure 2.2, and entering **man commandname** ⌨ will provide a brief summary of the action of the command, as illustrated in

```
[ID=0 of 1][PC=0x8000000][3.3E+00V, 6.0E+01MHz] help
ADDVALUETRACE            BATTALERTFRAC            BATTCF
        BATTETALUT               BATTETALUTNENTRIES       BATTILEAK
        BATTINOMINAL             BATTNODEATTACH           BATTRF
        BATTSTATS                BATTVBATTLUT             BATTVBATTLUTNENTRIES
        BATTVLOSTLUT             BATTVLOSTLUTNENTRIES     BPT
        BPTDEL                   BPTLS                    C
        CA                       CACHEINIT                CACHEOFF
        CACHESTATS               CD                       CLOCKINTR
        CONT                     D                        DEFNDIST
        DELVALUETRACE            DUMPALL                  DUMPMEM
        DUMPPIPE                 DUMPREGS                 DUMPSYSREGS
        DUMPTLB                  DYNINSTR                 EBATTINTR
        EFAULTS                  FF                       FILE2NETSEG
        FLTTHRESH                FORCEAVGPWR              GETRANDOMSEED
        HELP                     HWSEEREG                 IGN
        INITRANDTABLE            INITSEESTATE             L
        LISTRVARS                LOAD                     LOCSTATS
        MALLOCDEBUG              MAN                      MMAP
        N                        NANOPAUSE                ND
        NETCORREL                NETDEBUG                 NETNEWSEG
        NETNODENEWIFC            NETSEG2FILE              NETSEGDELETE
        NETSEGFAILDURMAX         NETSEGFAILPROB           NETSEGFAILPROBFN
        NETSEGNICATTACH          NETSEGPROPMODEL          NEWBATT
        NEWNODE                  NI                       NODEFAILDURMAX
        NODEFAILPROB             NODEFAILPROBFN           NUMAREGION
        NUMASETMAPID             NUMASTATS                NUMASTATSALL
        OFF                      ON                       PARSEOBJDUMP
        PAUINFO                  PAUSE                    PCBT
        PD                       PE                       PF
        PFUN                     PI                       POWERSTATS
        POWERTOTAL               PS                       PWD
        Q                        QUIT                     R
        RANDPRINT                RATIO                    REGISTERRVAR
        REGISTERSTABS            RENUMBERNODES            RESETALLCTRS
        RESETCPU                 RESETNODECTRS            RETRYALG
        RUN                      SAVE                     SENSORSDEBUG
        SETBASENODEID            SETBATT                  SETBATTFEEDPERIOD
        SETDUMPPWRPERIOD         SETFAULTPERIOD           SETFLASHRLATENCY
        SETFLASHWLATENCY         SETFREQ                  SETIFCOUI
        SETLOC                   SETMEMRLATENCY           SETMEMWLATENCY
        SETNETPERIOD             SETNODE                  SETPC
        SETPHYSICSPERIOD         SETQUANTUM               SETRANDOMSEED
        SETSCALEALPHA            SETSCALEK                SETSCALEVT
        SETSCHEDRANDOM           SETSCHEDROUNDROBIN       SETTIMERDELAY
        SETVDD                   SFATAL                   SHAREBUS
        SHOWCLK                  SHOWPIPE                 SIGSRC
        SIGSUBSCRIBE             SIZEMEM                  SPLIT
        SRECL                    STOP                     THROTTLE
        THROTTLEWIN              TRACE                    V
        VALUESTATS               VERBOSE                  VERSION
        NODETACH                 SIZEPAU
There are 139 commands and 13 aliases:
Type "man <command>" for help on a particular command.
You can also type "man <cmd>*" to get a list of commands matching "cmd".
[ID=0 of 1][PC=0x8000000][3.3E+00V, 6.0E+01MHz]
```

Figure 2.2: The `help` command lists all the available commands. More information on a particular command may be obtained with the `man` command.

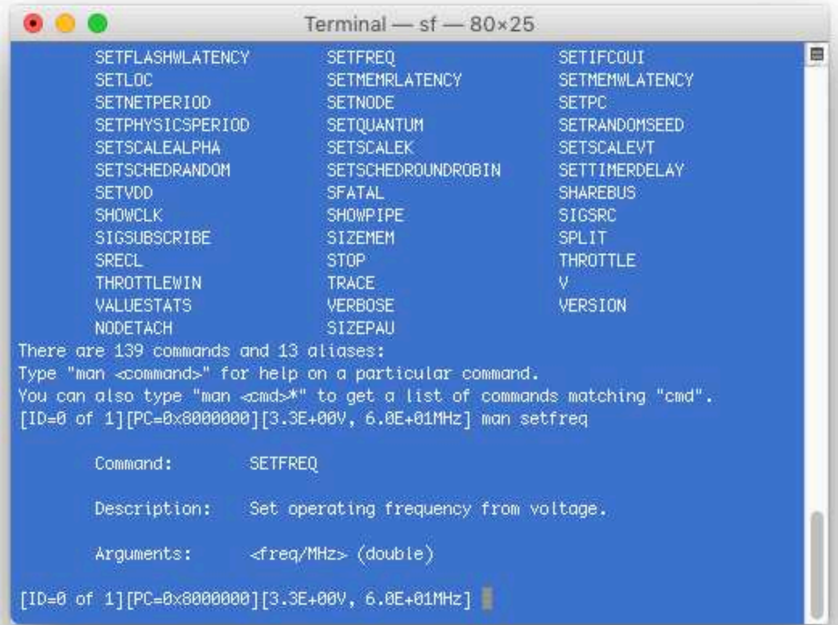


Figure 2.3: Using the built-in manual pages. Shown here is the manual entry for the `setfreq` command.

Figure 2.3

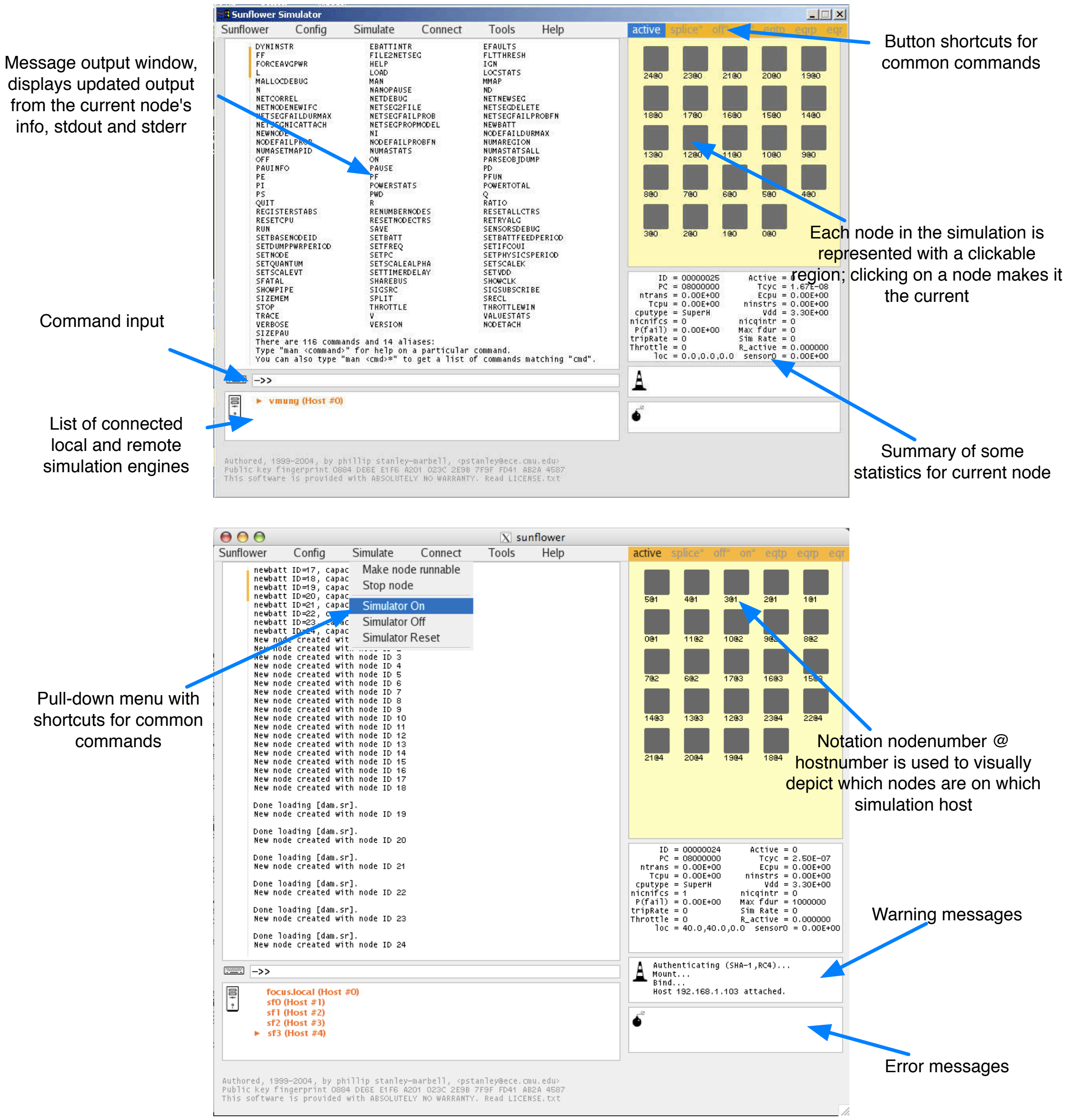


Figure 2.4: Illustration of the emulator GUI, on Windows (top) and MacOS (bottom).

2.4 *The Emulator User Interface*

The emulator provides two interfaces — an interactive text-based command interface, and a graphical user interface (GUI), illustrated in Figure 2.4. In addition to the facilities provided by the text-based interface, the graphical interface serves as the glue-logic for implementing facilities for distributing simulations over multiple host workstations.[1]

Both the text-based and graphical interface provide extensive on-line help facilities for all the built-in commands. Sets of commands, e.g., for setting up a processor network and its environment models, may be placed in files and loaded into the simulator at runtime.

[1] Stanley-Marbell 2006.

3
Loading and Running a Single Application

The emulator distribution includes the source and pre-compiled binaries for, among other examples, a simple *bubblesort* implementation, in the directory `benchmarks/source/bubblesort/`.

3.1 *Simple Example: A C-Language Bubble Sort Implementation*

The implementation of the bubble sort is in the file `benchmarks/source/bubblesort/bsort.c`, and the input to be sorted is included from the file `benchmarks/source/bubblesort/bsort-input.h`. This latter file contains a C array definition, containing the characters of a small passage of text and was generated from the file `benchmarks/source/bubblesort/input.txt`. The Makefile, which directs the compilation of the source files, compiles the C source, along with an assembly language stub (in Hitachi SH assembler) for initializing the processor, since the application will be executed in the absence of an operating system, directly over the modeled processor. The assembly language stub initializes the processor, sets up the stack pointer, and then jumps to the C code. The bsort application makes calls to a routine `print`, which is a minimal implementation of the `printf` routine from the standard C library.

The Makefile in the bubblesort build directory defines a variable `TREEROOT`, which specifies the root of the emulator installation directory, and is used to reference the emulator installation configuration file, `conf/setup.conf`. This is used to obtain various configuration information, such as which target architecture to compile for by default, and so on.

To compile the bubblesort application, given that the compilation tools have been correctly installed, change directory to `benchmarks/source/bubblesort/` and type `make`. This will build the bubblesort application from the C language source, and generate, among other things, a binary in S-RECORD format, `bsort.sr`. Binaries to be run over the emulator are in Motorola S-RECORD format and end in the suffix *.sr*. The bubble sort application is supplied pre-compiled, so even prior to building the cross-compiler, the built binaries necessary for loading into the emulator (i.e., `bsort.sr`), will already be present.

3.2 *Running the Compiled* **`bubblesort`** *Application*

After starting up the emulator, a binary may be loaded into the simulated processor's memory using the **srecl** command. To load a single binary into the emulator for simulation, type **srecl** *filename*. To run the program, entering the **`run`** (C.106) command marks the processor to which the command console is currently attached as "runnable", *and this must be followed by* the **`on`** (C.80) command, which actually initiates simulation.

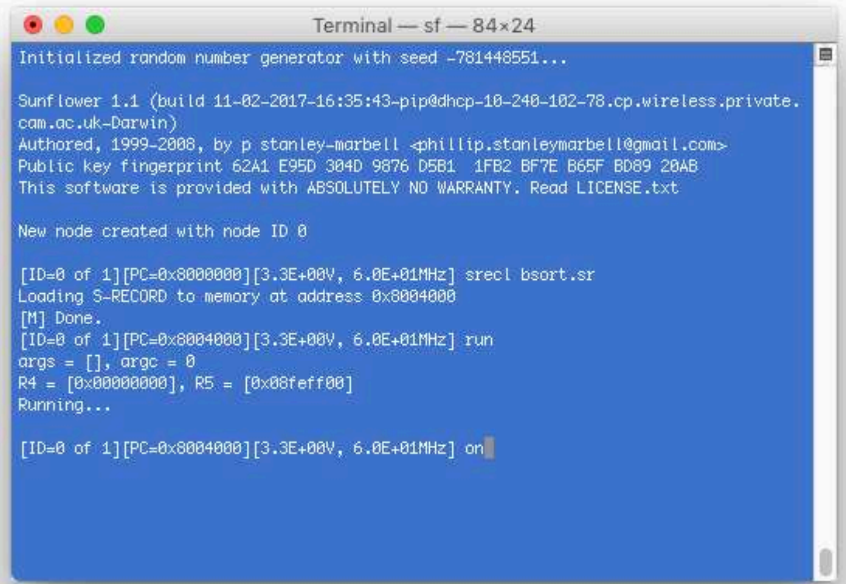


Figure 3.1: Loading and running a single binary on the emulator.

Figure 3.1 shows a screen capture of a session where a user starts up the emulator (`./sf`), creates a battery (**`newbatt 0 1.0`**) and attaches the current processor node to it (**battnodeattach 0**), loads the bubblesort binary into the simulated machine's memory (**`srecl bsort.sr`**), and runs it (**`run`** (C.106) and then **`on`** (C.80)). The **`run`** (C.106) command marks the current processor node (node 0, as shown in the leftmost side of the emulator prompt) as *runnable*. The **`on`** (C.80) command acts as the "Big Switch" to turn the emulator off or on. After the user enters the `on` (C.80) command, the emulator begins the execution of the instructions that were previously loaded into the simulated machine's memory.

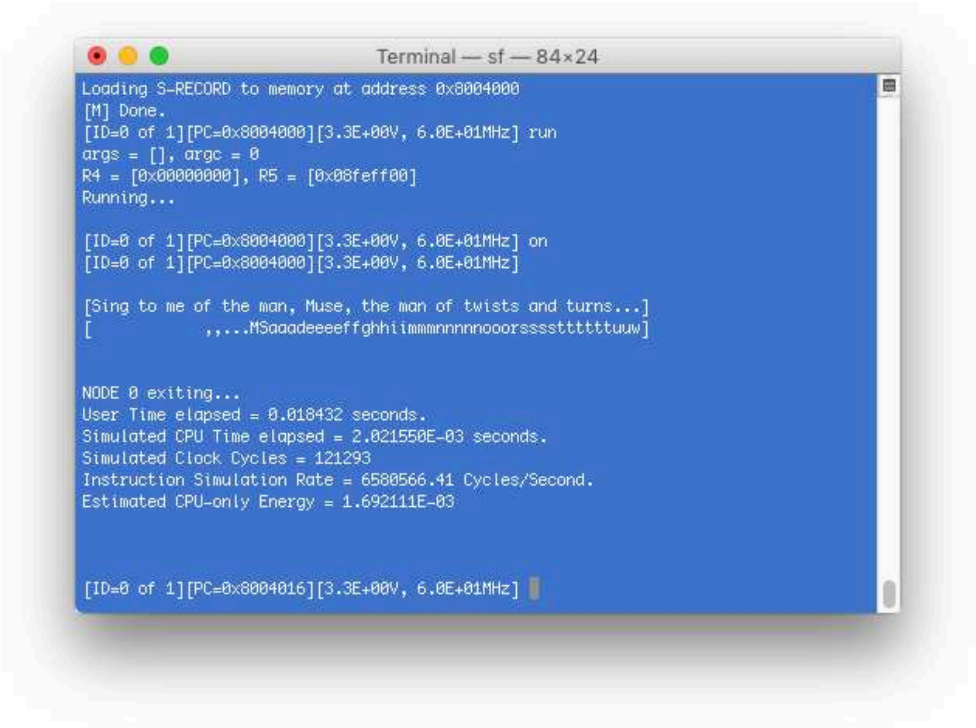


Figure 3.2: Sample output from the end of a simulated application. The emulator halts the simulation and prints some statistics when the application executes a `exit()` system call, which is eventually seen by the emulator as an exception.

Figure 3.2 shows the command console at the end of simulation of the bubble sort application. The emulator halts the simulation and prints some statistics when the application executes a `exit()` system call, which is eventually seen by the emulator as an exception. In the figure, this simulation took 0.05 seconds on the host machine that was running the emulator. The simulated time elapsed, from the point of view of the simulated processor is 6.6135E-4 seconds which corresponds to the simulated processor taking 39,681 clock cycles to execute the bubblesort application. These 39,681 clock cycles correspond to 6.6135E-4 seconds since the processor is assumed[1] to have a cycle time of 16.6667 ns, corresponding to an operating frequency of 60 MHz. Given the number of processor cycles simulated, and the time taken to perform this simulation on the host machine, the emulator reports a *simulation rate* of 793.62 K Cycles/Second. The energy consumed by the processor in executing the bubblesort program is reported as 5.448111E-04 Joules, and is also obtainable by entering the `ps` (C.92) command at the command line. In Figure 3.2, The output is

```
[Sing to me of the man, Muse, the man of twists and turns...]
```

[1] This is actually not an assumption, but the actual speed at which the modeled processor runs, with respect to the empirical power measurements that are integrated into the emulator. However, in terms of functional simulation, only the number of clock cycles simulated have any real significance.

```
[            ,,...MSaaadeeeeffghhiimmmnnnnnooorssssttttttuuw]
```

At any point during, or at the end of the simulation, the user may enter commands to probe the state of the system, as the command line operates asynchronously from the simulations.

The are numerous commands that users may use to probe or modify the state of the simulated machine. Figure 3.3 shows the output of the **dumpregs** (C.32) command, which displays the contents of the machine's general purpose registers.

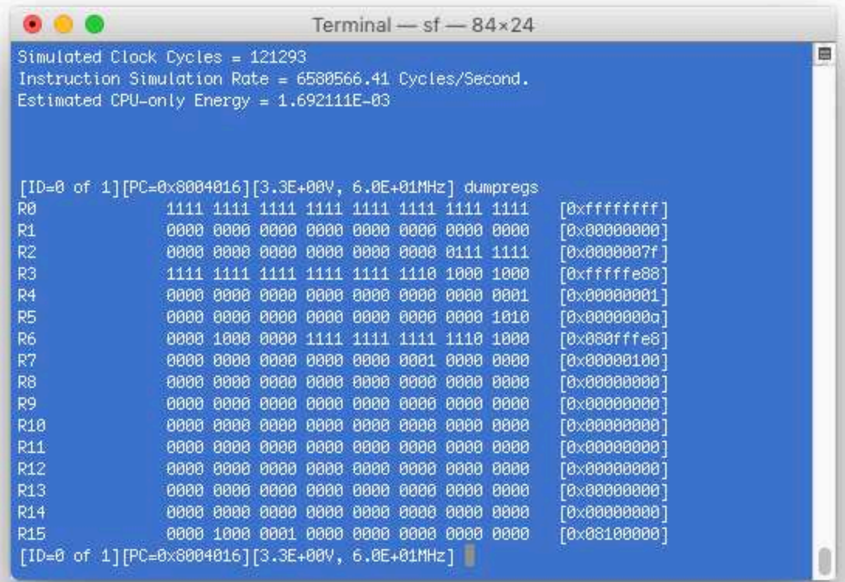


Figure 3.3: Dumping the contents of the active processor core's general-purpose registers using the **dumpregs** command.

A user may modify the machine state arbitrarily, since the *entire instruction set of the simulated machine is available to the user as commands*. For example, given the state of the machine's register file as displayed in Figure 3.3, to copy the contents of the register R2 to the register R7, a user could do this by issuing the appropriate Hitachi SH instruction, the **MOV** instruction, from the command line. Prior to doing this however, the emulator's modeling of the pipeline must be disabled, in order for the instruction to be executed as soon as it is issued, using the **pd** (C.85) command, as illustrated in Figure 3.4

The **dumpregs** (C.32) command is now issued again, and it shows that registers R2 and R7 now have the same value of 0x00fffffe88, as shown

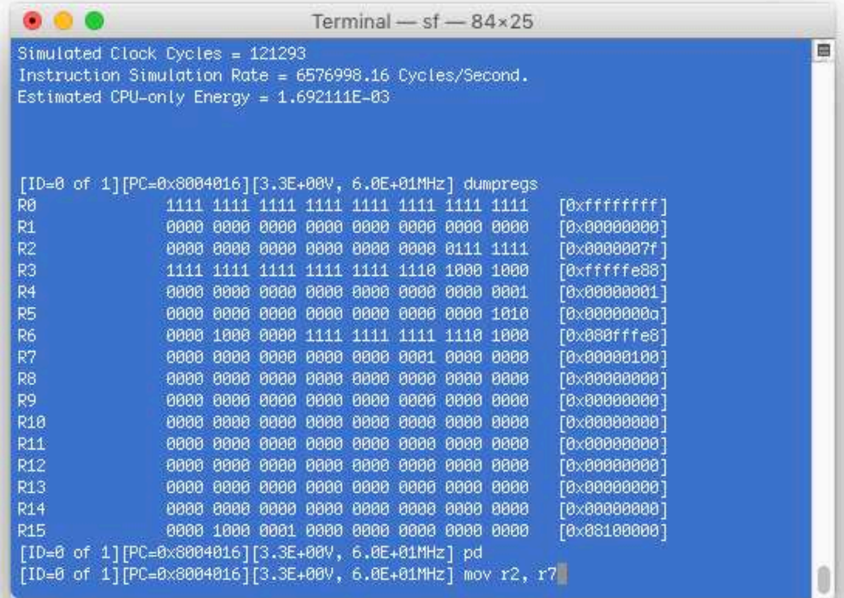


Figure 3.4: Using the emulator console interface's built-in assembler. We first disable the modeling of the pipeline using the **pd** command so that assembly code types into the emulator prompt will get executed as soon as it is assembled rather than only being assembled and stored in memory. We then issue the Hitachi SH assembly instructions **mov, r2, r7** to copy the contents of register **R2** into register **R7** of the emulated processor.

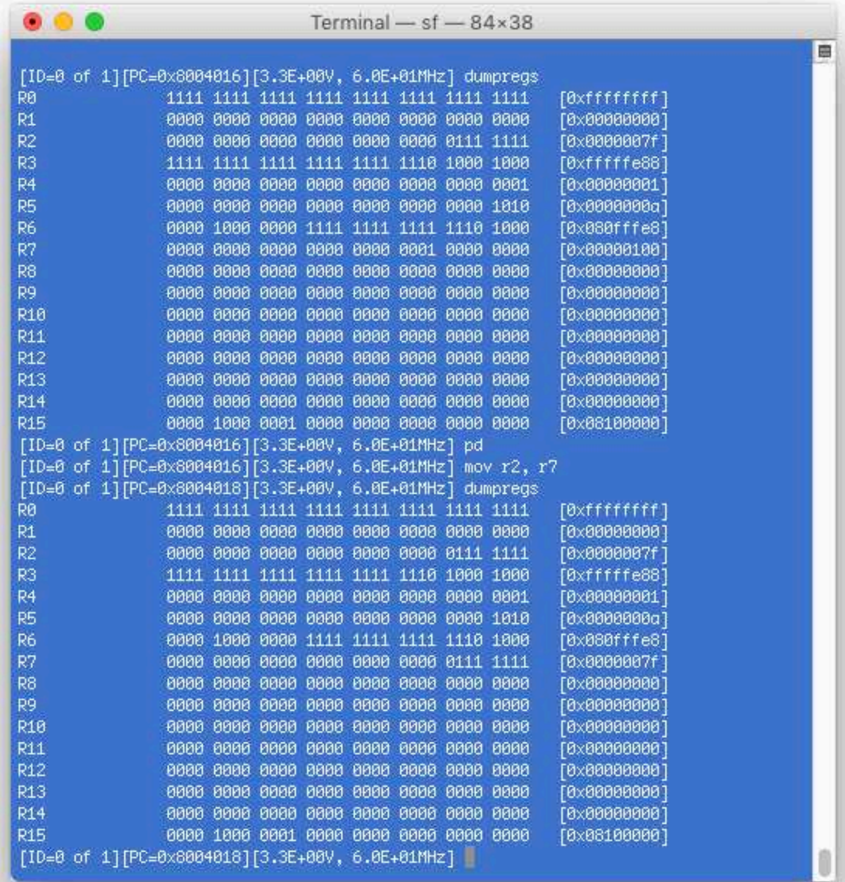


Figure 3.5: After issuing the **`dumpregs`** command again, we see the effect of the **`mov, r2, r7`** instruction on the emulated processor core state.

below in Figure 3.5.

4
Modeling Processor Cores

Emulating instruction execution is the functionality of Sunflower. Emulating applications at the level of detail of the emulating the execution of their compiled code makes it possible to employ the emulator as a debugging platform for hardware prototypes. Sunflower's combination of processor emulation and physics simulation makes it possible to determine important interactions between the requirements of computation, communication and reliability, and the effects of these constraints on power consumption.

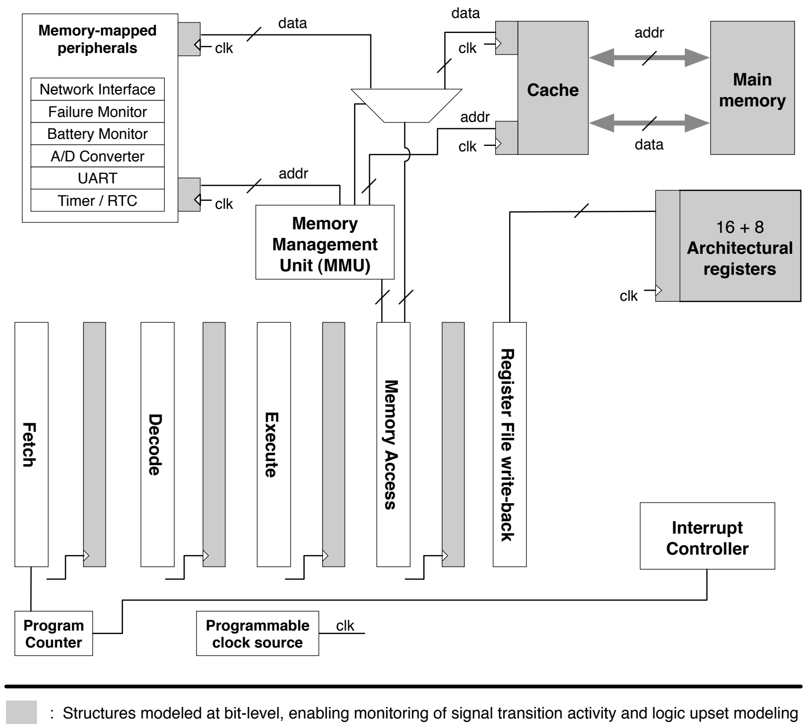
4.1 Processor Cores

The emulator includes two different architectural implementations, one for the Hitachi SH architecture, based on the Hitachi SH3 SH7708 (Figure 4.1(a)), and the other of the TI MSP430 architecture(Figure 4.1(b)). Support for new architectures requires primarily the addition of code for implementing instruction decode and execution. The modeling of on-chip structures such as interrupt generation, caches, memory interfaces and some standard peripherals such as a network interface is shared across the different architectures.

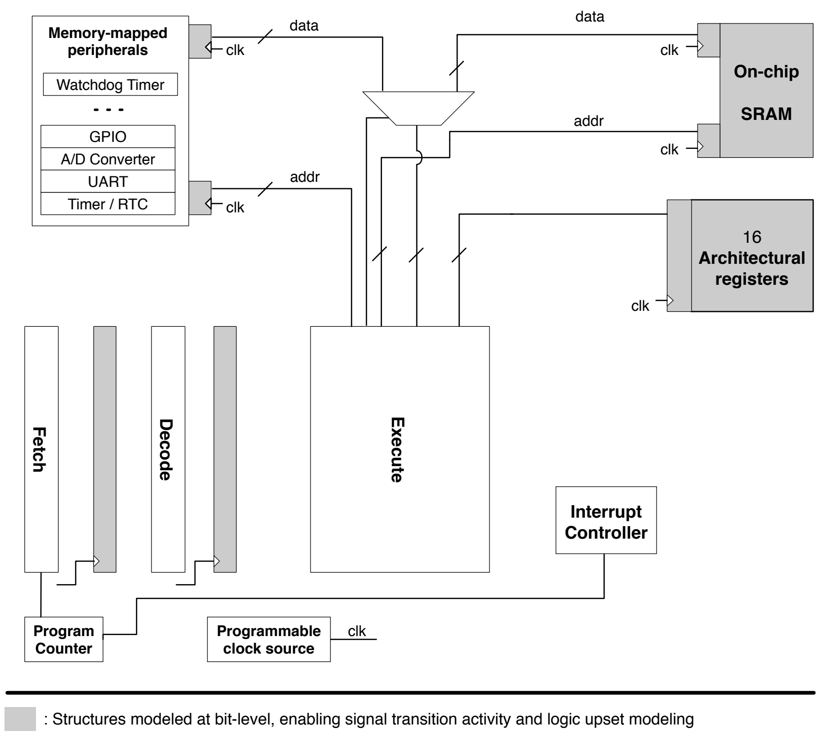
The Hitachi SH3 model includes detailed modeling of the CPU core, on-chip cache and on-chip peripherals such as an RS-232 UART. It incorporates multiple complementary means of estimating the energy cost of application software, including an empirical instruction level power model and circuit activity estimation. The instruction-level power model functions by assigning to each instruction executed, an energy dissipation based on empirically measured values, scaled if necessary for a given operating voltage and frequency, as the model supports dynamic scaling of both operating voltage and frequency. Employing this simple energy estimation scheme enables fast simulation, which is critical since the framework is often used to simulate such platforms consisting of tens of processing devices. Although simple, the employed instruction level power estimation has been shown to be within 6.5% of measured values for the hardware it models.[1] The instruction-level power

[1] Stanley-Marbell and Hsiao 2001.

Figure 4.1: Default configuration of the modeled microarchitectures.



(a) Default configuration of the modeled, 32-bit architecture, employing the Renesas/Hitachi SuperH (SH) ISA.



(b) Default configuration of the modeled 16-bit architecture, employing the TI MSP430 ISA.

model can be augmented with a circuit transition activity estimation, which reports, for each simulation cycle, the signal transition activity on the address and data buses, in the register file, the program counter and pipeline registers. The SH3 core model provides 6 levels of detailed simulation, enabling a tradeoff between power estimation accuracy and simulation speed.[2] The energy estimation facilities, as well as the modeling of batteries and voltage regulators, is described in more detail in Chapter 5.

[2] Stanley-Marbell and Hsiao 2001.

The TI MSP430 architecture model provides functional simulation of the processor and its peripherals for the MSP430F11 series of microcontrollers. Unlike the SH3 model, it currently provides only functional modeling of the modeled microcontroller, to enable applications compiled for a prototype system to be modeled and debugged in the emulator. The implementation of the MSP430 model is currently not fully integrated into the public source distribution.

4.2 *Processing Nodes*

The emulator uses the term *processing node* to refer to a combination of a CPU core, on-chip cache, various on-chip peripherals, off-chip memory, RS-232 serial communications interface and a network interface controller. Each processing node may further have several network interfaces instantiated, and each of these connected to an interconnection link. The processing nodes may be configured to run at different operating voltages (and hence frequencies), main memory size, cache size etc., and may also be configured for different probabilities of random failure.

4.3 *Built-in Assembler*

The emulator includes built-in assemblers for the Hitachi SH. In addition to the standard assembler mnemonics, a small number of assembler directives in the Gnu assembler format are supported:

1. **`.org`** location counter set.
2. **`.align 2`** 2-byte boundary alignment.
3. **`.align 4`** 2-byte boundary alignment.

Not supported:

1. **`.data.w`** for setting integer word data.
2. **`.data.l`** for setting integer longword data.
3. **`.sdata`** for setting string data.

4. **`.arepeat 16`** 16-repeat expansion.
5. **`.arepeat 32`** 32-repeat expansion.
6. **`.aendr`** end of repeat expansion of specified number.

5

Power Estimation, Electrochemical Cells, and Voltage Regulators

Energy consumption, average power dissipation and battery lifetime play an increasingly important role as *metrics* of system performance, in addition to traditional metric objectives such as various interpretations of timeliness (communication and computation throughput, per-operation and end-to-end latency, and so on). Energy, power, and battery lifetime are not always related in simple ways (knowing one does not always imply the other).

In a modeling framework targeted at application domains where these metrics are of importance, it is thus desirable to enable their accurate modeling. The Sunflower simulator enables the estimation of instantaneous power dissipation of computation (processors) and communication (network interfaces), as well as the modeling of the behavior of battery subsystems.

5.1 Computation Power Estimation

The simulator incorporates three complementary means of estimating energy cost of application software — an empirical instruction-level power model similar to,[1] circuit activity estimation, and a coarse-grained mode-based power model.

[1] Tiwari, Malik, and Wolfe 1994.

The instruction-level power model employs a table of measured average current drains for each instruction in the ISA, using this lookup table during simulation to estimate the average power dissipation during each clock cycle, given the present (possibly-scaled) operating voltage and frequency. When either the operating voltage or frequency is changed via the **`setvdd`** ⌨ (C.133) or **`setfreq`** ⌨ (C.116) commands, the other is updated based on the CMOS gate delay equation:

$$delay = (k \cdot Vdd)/(Vdd - Vt)^{\alpha}, \tag{5.1}$$

where the operating frequency is the reciprocal of $delay$, and Vdd is the operating voltage. The variables k, Vt, and α can be set via the commands **setscalek** ⌨ (C.128) **setscalevt** ⌨ (C.129) and **setscalealpha** ⌨ (C.127). *The default values are set to enforce a linear relation between operating frequency and operating voltage.* If such behavior is not desired, the values of the delay equation variables should be set appropriately by the user of the simulator. Due to the non-algebraic relation between Vdd and $delay$, while the delay is easily calculated for a given choice of operating voltage, the solution of Vdd for given values of $delay$ (i.e., setting the operating voltage given a requested setting of operating frequency), is not straightforward. The approach taken in the simulator implementation is to restrict the values of $\alpha = 0.5, 0.6, \ldots, 1.9, 2.0$, and the simulator only permits using those pre-determined values of α when scaling frequency.

The second alternative means of estimating (dynamic) power dissipation for a given execution window is through the use of circuit activity estimation. The simulator models several structural aspects of the processor architecture, such as the pipeline latches, register file read and write ports, address and data buses. The structures for both modeled ISAs which are modeled structurally were shown previously in Figure 4.1(a) and Figure 4.1(b). Monitoring the number of signal transitions on these structures enable qualitative comparison between the expected dynamic power dissipation while executing different applications, or while employing different system architecture configurations. This modeling facility however does not provide a direct readout of power dissipation, as the simulation framework does not incorporate any notion of the design- and fabrication-technology dependent capacitances. The output of the **ps** ⌨ (C.92) command reports the dynamic signal transition count to-date, and it is also reported in the simulator output log file (`sunflower.out`), generated at the completion of simulation or at any point via the **dumpall** ⌨ (C.29) (alias **d** ⌨ (C.26)) command.

While most of the facilities of the simulator are enabled via commands at runtime, facilities which may slow down simulation and may not be needed by casual users must be enabled at compile time in the `sim/config.h` 🗁 *configuration file, whose format is described in more detail in Section 12.2. For example, the flag* `SF_BATT` *therein enables modeling of the battery, while the flag* `SF_BITFLIP_ANALYSIS` *enables the circuit activity estimation modeling.* !

The third facility for power estimation is a coarse-grained mode-based power estimation facility, which uses configuration-specified fixed power dissipations for the processor active and idle modes. These mode power dissipations are set via the **forceavgpwr** ⌨ (C.41) command, which takes two arguments, the active and idle mode power dissipations. Using this power estimation facility bypasses the instruction-level power estimation, but it may

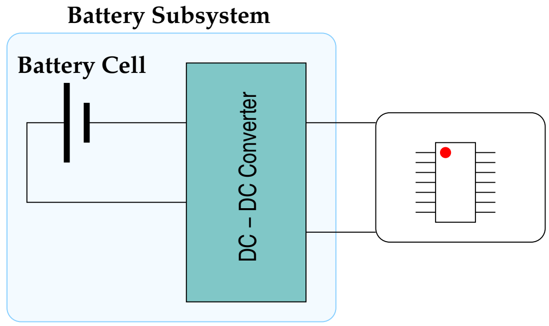


Figure 5.1: Organization of battery subsystem. The voltage regulator (DC-DC converter) is required to obtain a constant voltage to power electronics, due to dependence of battery cell terminal voltage on battery state of charge.

be used in conjunction with the circuit activity estimation.

5.2 *Non-ideal Power Sources and Voltage Regulators*

Each processing node must be attached to a source of energy. The first order effects of discharge rate, voltage regulator efficiency, etc., are modeled, and battery dependent characteristics such as the dependence of the battery terminal voltage on state of charge, and the DC-DC converter efficiency curve may be supplied by the user. The default battery parameters are for a Panasonic CGR18 family Lithium Ion battery. The default voltage regulator characteristics are those for a Dallas Semiconductor/MAXIM MAX1653.

5.2.1 *Battery Subsystem*

The simulator includes a detailed discrete-time battery modeling engine based on.[2] In brief, the model takes into account properties of battery cells, such as dependence of battery terminal voltage on the *state of charge (SOC)* of a battery, dependence of usable capacity on discharge rate, and dependence on the rate of change of current over time. In order to provide a constant voltage to the powered electronics in the face of variation in battery terminal voltage over time, a *voltage regulator* (DC-DC converter) provides voltage stabilization, at the cost of a loss due to inherent inefficiencies in the conversion. A simple organization of a battery powered system is shown in Figure 5.1 to illustrate this further.

[2] Benini et al. 2000.

In order to model different types and sizes of batteries and voltage regulators, the model (and its implementation in the simulator) uses lookup tables (LUTs) and additional constants to capture empirical characteristics of specific batteries. The default battery characteristics employed in our implementation, are those for a lithium ion cell from the Panasonic CGR18 family. The supplied models in the simulator distribution, which may be loaded during a given simulation configuration, currently include models for the Panasonic CGP345010, Panasonic CGP345010g, Panasonic CGR17500 and

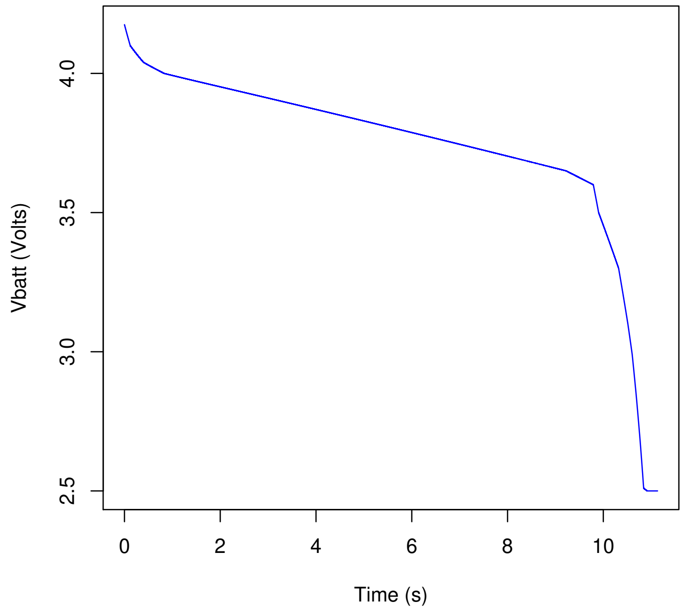


Figure 5.2: Variation of battery cell terminal voltage over time for a nominal current draw of 150 mA from outside the battery subsystem.

Panasonic CGR18650HM. The voltage regulator characteristics employed are those for a Dallas Semiconductor/Maxim MAX1653 device. Additional supplied models that may be loaded at simulation time are for the TI TPS61070, TI TPS61071, TI TPS6110x and TI TPS6113x voltage regulators. User lookup tables may be loaded into the simulator to mimic other device's characteristics, for both the battery cell or other kinds of energy storage devices such as supercapacitors, and other voltage regulators.

Figure 5.2 shows the dependence of battery terminal voltage with time for a nominal discharge rate of 150 mA. The data in Figure 5.2 also includes the effect of voltage regulation, and depicts the lumped behavior of the battery cell if the entire battery subsystem were attached to electronics that had a constant current draw of 150 mA.

Battery self-discharge is modeled by specifying a battery leakage current, which can be changed from its defaults via the **`battileak`** ⌨ (C.6) command. The other components of the battery properties are illustrated in Figure 5.3. The parameters of interest in this work are V_r, a measure of the rate of discharge, V_{rate}, a low-pass filtered version of V_r, and V_{lost}, which models the dependence of battery terminal voltage on the magnitude of V_{rate} for a particular battery type (from a lookup table). Lastly, V_C models the instantaneous state of charge, taking in to consideration V_{lost}.

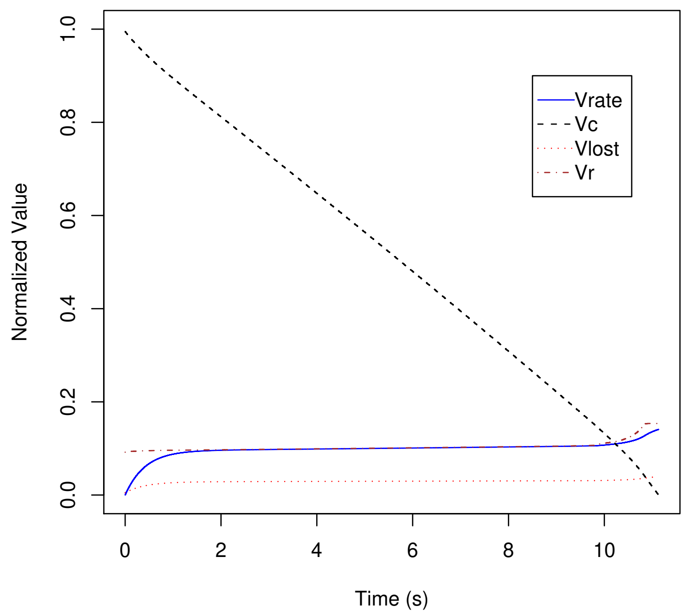


Figure 5.3: Variation of components of battery model with time for a nominal current draw of 150 mA from outside the battery subsystem.

The battery low-pass filter capacitance and resistance can be set via the **battcf** ⌨ (C.3) and **battrf** ⌨ (C.9) commands. The points on the battery discharge profile lookup table can be set via **battvbattlut** ⌨ (C.11) and **battvbattlutnentries** ⌨ (C.12) commands, while the points on the voltage regulator efficiency curve can be set via the **battetalut** ⌨ (C.4) and **battetalutnentries** ⌨ (C.5) commands. The battery voltage sag as a function of drain current is specified via the **battvlostlut** ⌨ (C.13) and **battvlostlutnentries** ⌨ (C.14) commands. The battery nominal current draw associated with these lookup tables can be set by the **battinominal** ⌨ (C.7) command.

6
Interconnect and Network Modeling

Flexible modeling of interconnect networks in the Sunflower simulator is facilitated by an interconnect architecture made up of two components — *network interfaces* and *communication media* (also henceforth referred to as *network media*, *network segments*, *network links* or *communication links*). The communication media are the models of the actual interconnect links, and have properties such as the ability to permit single- or multi-access communication, communication bit rates, signal deterioration along the length (for wires) or area (for wireless channels) of the communication link, and so on. Separate from these communication media models, are models of the interfaces between the modeled processors and the communication medium. In the Hitachi SH processor model, a new standard hardware peripheral has been added to the system architecture, a multi-channel network interface, of which multiple communication interfaces may be instantiated. Each such communication interface on a single processor may be connected to a different communication medium, enabling the creation of arbitrary interconnect topologies between modeled computing systems.

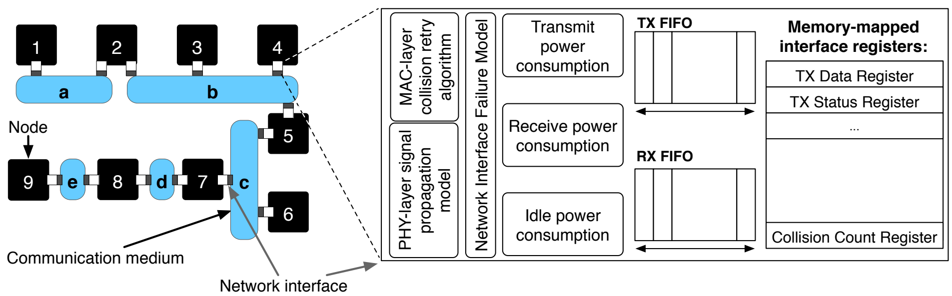


Figure 6.1: Modeling communication networks is separated into the modeling of *network interfaces*, connected to *communication media* to form communication topologies.

Figure 6.1 illustrates the organization of an example network. The figure depicts an example network comprising nine nodes, connected in a topology consisting of 5 disjoint communication media. Some of the nodes, e.g., 2, 5, 7, and 8 have multiple network interfaces, and are attached to multiple media through these interfaces. Some media, e.g., **b** and **c** are "multi-drop" or shared links, while others (**a**, **d** and **e**), are point-to-point.

Like other groups of commands related to the same functionality, commands related to the modeling of interconnect networks generally begin with the same prefix, in this case, **net**. The commands related to interconnect networks include `netcorrel` ⌨ (C.58), `netdebug` ⌨ (C.59), `netnewseg` ⌨ (C.60), `netnodenewifc` ⌨ (C.61), `netseg2file` ⌨ (C.62), `netsegdelete` ⌨ (C.63), `netsegfaildurmax` ⌨ (C.64), `netsegnicattach` ⌨ (C.67), `netsegpropmodel` ⌨ (C.68) and `file2netseg` ⌨ (C.39).

6.1 *Instantiating Network Media:* NETNEWSEG

Interconnect links are instantiated with the `netnewseg` ⌨ (C.60) command. Each communication link may be configured for the following properties:

- *Frame size* — data is transmitted on a communication link in groups of bytes referred to as a "frame".
- *Propagation speed* — the propagation delay specifies the speed at which a signal travels in the communication medium, over the communication link. When modeling wired communication, this is taken to be the speed of light. Nodes in the simulation can have associated with them a location in 3-dimensional space, which will then be used in conjunction with the propagation speed to determine the propagation delay. For most simulation scenarios however, this parameter can be ignored.
- *Transmission speed* — the transmission speed specifies the number of bits that are modulated per second, or the bit-rate of the communication medium.
- *Maximum simultaneous accesses* — specifying a maximum number of simultaneous accesses permits a medium to be configured to behave like either a CSMA medium, such as Ethernet, or as a CDMA medium,
- *Failure probability and maximum failure duration* — These are discussed further in the description of the failure modeling in Chapter 7.

6.2 *Instantiating Network Interfaces:* `netnodenewifc`

The interface between applications executing over the microarchitectural simulation, and the modeled networks, is the *network interface*. In the Hitachi SH processor model, the original processor architecture was extended with a flexible network interface peripheral that permits the dynamic instantiation of multiple network interfaces; in the TI MSP430 model, the USCI (Universal Serial Communication Interface) serves as the network interface, and the number of network interfaces is thus fixed, and the `netnodenewifc` (C.61) command is not relevant.

The `netnodenewifc` (C.61) command takes as arguments the transmit, receive, idle and idle listening power dissipation settings, among other things. In order to ensure network interfaces are always compatible with the networks to which they are attached, network interfaces inherit all other properties (e.g., communication bit rate, failure configuration, etc.) from the interconnect link to which they attached. The transmission and receive power consumption of a network interface may however be configured independently of the properties of the link with which it is associated. The simulation of data transmission and receipt is kept cycle-accurate with respect to computation. The granularity at which data is transferred from one device to another is determined by the smallest cycle time of all the modeled processing devices.

6.3 *Saving and Loading Network Traces:* `netseg2file` *and* `file2netseg`

It is often desirable to be able to save a trace of the network traffic transpiring over an interconnect network, including both the data being communicated as well as sufficient information to re-create such traffic. The `netseg2file` (C.62) command takes as parameter a file name, and saves all data traffic transpiring over the network to this file. The file format of such *network trace files* is detail in Section 12.9. These files can subsequently be loaded into a simulation via the `file2netseg` (C.39) command. Naturally, such traces may also be created by other means, whether artificially or via capturing trace data from actual deployed networks, converted to the tracefile format, and loaded into simulations via the `file2netseg` (C.39) command.

6.4 *Configuring Network Media Signal Propagation Properties*

Interconnect links are seldom ideal carriers of bits from source to destination. Data to be transmitted is modulated over a carrier medium, and is in principle always subject to a variety of sources of signal degradation or other forms of interruptions. Such transmitted signal interactions, interference and degradation over distance is particularly relevant in the study of wireless communication links.

The Sunflower simulator enables the modeling of many of these signal propagation aspects of interconnection links, by harnessing the emulator's existing facilities for modeling the propagation of signals in environments, and their interactions with each other. An instantiated interconnect link within a simulation can be associated with a signal propagation model via the **netsegpropmodel** (C.68) command. This command takes as arguments the identifier of the interconnect link, that of the signal propagation model (i.e., one out of members of a *signal group* as described in Chapter 8), and a minimum signal to noise ratio (SNR) specification. During data transmission, the strength of the associated signal at the location of the destination of the communication, relative to the net strength of other signals within the signal group, is used to calculate an instantaneous SNR at the destination. If this SNR is smaller than the minimum SNR specified in the **netsegpropmodel** (C.68) command, then bit errors are introduced in the transmitted data. Since this approach harnesses the full implementation of signal propagation, interference and interaction models in the emulator, arbitrarily complex signal propagation models can be associated with interconnect links.

6.5 Example

The following except from a simulation configuration file illustrates the ideas in the foregoing discussion.

```
...

--
--      Signal source "A"
--      Due to the proximity of the sensors in the original experiment @ PARC,
--      we use a Ricean model for the RF propagation, w/ received Pr = K/d^n, n = 2
--      We set ambient RF noise at 1/100 of the Peak radio power.
--
sigsrc                  1 "Radio propagation model"     0.0 0.0         1.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0         -2.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
        " "             0 0.0 0.0 0.0 1 " " 0 0.0 1

--      Signal source "B"
--      We set ambient RF sig strength to equal 1/10 strength of 89.1mW
--      radio at sqrt(10^2+10^2)=14.14 units, and set minsnr to 9 (9 < 10)
sigsrc                  1 "Ambient RF noise"            0.0 0.0         1.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0         0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
" "             0 0.0 0.0 0.0 1 " " 0 0.00004455 1


--
--      Because PARC experiment uses broadcasts, it makes sense to do collision
--      detection, hence we configure medium to be CSMA (width 1)
--
netnewseg               0 1024 300000000 38400 999 0 0 0 0 0 0 0 0
netsegfailprob          0 0.0
netsegfaildurmax        0 1000000
-- netseg2file          0 netseg0log
```

```
--
--      The SNR is tuned for the spatial layout (below), so that only
--      immediate neighbors in the topology get valid transmissions
--
netsegpropmodel         0 3 9.0


...


--
-- Node instantiation, creation of a network interface, and attachment to
-- an interconnect link
--
newnode superH       0 0 0 0 0
netnodenewifc           0 0.0891 0.0330 0.0000033 0.0330 0 0 0 0 256 256
netsegnicattach         0 0
retryalg                0 "none"
```

7
Fault Modeling

Device failures in computing systems may take a variety of forms, ranging from bit-level errors within a system microarchitecture, to whole-system failures. The consideration of failures of different kinds is of increasing relevance in computing system and computer architecture research, as trends in semiconductor device technology (smaller device feature sizes, migration to new gate, gate oxide and interconnect materials, lower operating voltages, smaller margins between operating and threshold voltages) while enabling increased performance, may result in increased susceptibility to fault sources such as high energy particles, ground bounce, and electromagnetic interference. The falling costs of semiconductor devices have also spurred many new applications of computing systems, and many of these new application domains are in environments where devices might be subject to non-ideal operating conditions (forests, deserts, car engines) and furthermore, may be difficult to reach to diagnose in the case of a hardware fault leading to a system failure. It is therefore of interest to consider the modeling of these diverse types of faults in system evaluation frameworks such as Sunflower.

Modeling Failures: The Sunflower emulator models failures in both processing devices and communication links. Failures in processing devices can be configured to manifest as intermittent stalls of the entire processing device, for the duration of the failure, or as bit-level data value inversion in the portions of the microarchitecture that are modeled structurally, such as the pipeline latches, register files, buses, and so on, shown shaded grey in Figure 4.1(a) and Figure 4.1(b). Failures in communication links manifest as intermittent loss of carrier for the duration of the failure, and may also be introduced implicitly when modeling wireless networks with radio propagation profiles, as detailed in Chapter 6. For both failures in devices and communication links, the failure rate and maximum failure duration are configurable. Correlated failures between processing devices and communication links can be modeled by specifying appropriate correlation coefficients for a given node-link pair.

The failure probabilities of interconnect links are specified with the **`netsegfailprob`** ⌨ (C.65) command, while that of nodes is specified with the command **`nodefailprob`** ⌨ (C.73) . Correlation coefficients between node and network failures are specified using the **`netcorrel`** ⌨ (C.58) command.

8
Environment Models

Signals in the environment, such as light, sound or electromagnetic waves, drive the computation occurring in many embedded systems. This is particularly true in application domains such as wireless sensor networks, where the sole role of deployed computing systems is often to monitor and react to such signals in the environment. The presence or absence of a signal at a given location in space, its strength, rate of (amplitude) variation with time, etc., may all affect the occurrence of computation, in systems monitoring the phenomenon, and may even affect the *performance* of such computation. For example, signal processing applications processing values from sensors need to sample the (band-limited) signal at twice the maximum frequency component to prevent aliasing (Nyquist's criterion), and thus the amount of data needed to be processed by such a signal processing system, as well as the rate at which it must perform such computation, is dependent on the properties of the signal it is monitoring.

The environment in which a system is deployed may also have more indirect effects on computation. For example, temperature in a system will affect its leakage power dissipation, as well as the drift in any crystal driven oscillators. In a networked embedded system for example, such clock drift may then lead to the need for the implementation of a time-synchronization protocol, which may add additional computation and latency overhead, and so on.

The Sunflower simulator provides facilities for modeling the location, motion and time-evolution of signals in the environment of computing systems, and synchronizes this modeling with the low-level architectural simulation it performs. Instantiated processors are assigned a location and bearing (direction) in three dimensional space, and computation executing on processors may read from sensors tied to signals in the environment, as well as driving actuators tied to the environment.

8.1 Node Location, Orientation, and Trajectory Definition

Node locations and their direction/orientation relative to a common reference "north" and "horizon" are specified when creating new nodes via the **newnode** ⌨ (C.70) command; the location is specified as an x, y, z triplet Cartesian coordinate in an arbitrary reference frame, while the orientation is specified as a ρ, θ, ϕ polar coordinate. The **newnode** ⌨ (C.70) command also permits the specification of a node location trajectory file (format detailed in Section 12.8), specifying any variation in the position and direction of the node with time. Node locations may also be changed dynamically using the **setloc** ⌨ (C.118) command, and a node's current location can be queried with the **locstats** ⌨ (C.51) command.

Node locations are used in determining the strength of signals sensed by a node, as signal definitions, described in Section 8.2, are associated with signal attenuation profiles.

8.2 Defining Signal Sources and Signal Interactions/Interference

A signal in an environment is defined using the **sigsrc** ⌨ (C.138) command, and multiple signals being *subscribed to* (via **sigsubscribe** ⌨ (C.139)) by a sensor are termed a *signal group*. Each component in this group of signal sources has a defined signal propagation speed in space (relevant to changes in value), a signal attenuation profile equation , a signal trajectory specification file (format described in Section 12.7), and a signal sample value specification file (format described in Section 12.6), among other things.

The attenuation of signals with radial distance, r, is modeled by providing coefficients to the expression:

$$Amplitude(r) = S \cdot (A \cdot r^i + B \cdot r^j + C \cdot r^k + D \cdot r^l + E \cdot K^{(F \cdot r^m + G \cdot r^n + H \cdot r^o + I \cdot r^p)}). \quad (8.1)$$

Arbitrary signals can thus be modeled by regression, with reasonable accuracy. This approach lets a user choose the coefficients of r in the above equation to provide a good fit for many functions that are likely to be of interest, while enabling efficient simulation. For example, a perfect fit for an attenuation function that has the shape of a standard normal distribution can be obtained as follows: set the coefficients S, E, F, K and q to 1, 1, e, -0.5 and 2 respectively, all other coefficients to zero.

Signal sources can have positive or negative amplitudes. Signal sources within a given group are summed to yield the final signal result, thus arbitrarily complex signal spatial distributions with properties such as directionality and non-radial profile can be created by defining appropriate members of a signal source group.

9
Extended Example

The previous chapter illustrated the basics of loading and executing an application over the simulator. This chapter carries the basic concepts introduced further, and presents the implementation and simulation of a larger application that executes over a network of multiple processors. The majority of the material presented in this section is specific to the target *architecture* used in the examples here — the Hitachi SH processor model.

9.1 A Software-Defined Radio Application

As the illustrative example in this chapter, we will employ a software-defined radio or *software radio*, application, partitioned for execution over a network of processors. The software radio application (henceforth, *swradio*), is partitioned into 5 components—*Source*, *LPF*, *Demod*, *EQ* and *Sink*— as shown in Figure 9.1(a). Each of these components is implemented as a stand-alone application, which executes on a single processor, and communicates with the other components over an interconnect.

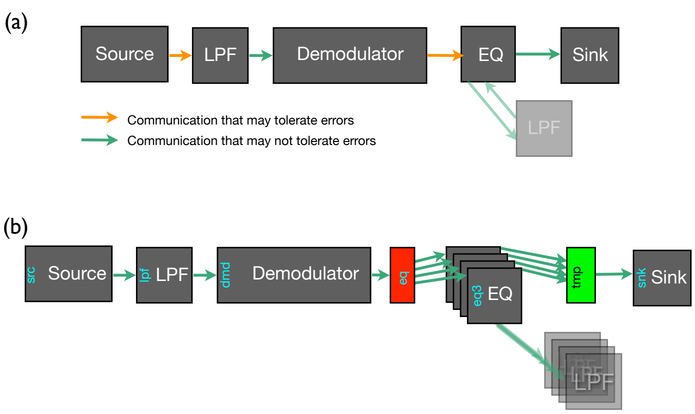


Figure 9.1: Software radio application, showing computation stages ((a), top), with further partitioning of the EQ stage, ((b), bottom).

The *Source* stage generates samples at a fixed rate, which it send to the *LPF* stage over the network, and so on. Due to the mismatch between the computational requirements of the different stages, the throughput of the application might be limited by the slowest or most compute-intensive stage, which happens in this case to be the *EQ* stage. In other words, the fraction of time spent idle for the different processors on which the stages of the application run will be mismatched. In order to provide a better balance of CPU utilization therefore (and also to improve throughput), the *EQ* stage is further partitioned into 8 copies (Figure 9.1(b)), which receive (and process) samples round-robin. This breaking up of the *EQ* stage is essentially a high-granularity implementation of the well-known software pipelining technique.

Thus, rather than the *Demod* stage sending all its data to a single *EQ* stage, it sends the data, round-robin, to each of the 8 different instances of the *EQ* stage, running on 8 different processors. In the steady state, one of these 8 *EQ* stages will produce a processed sample each period, though their processing of samples will overlap in time.

The implementation of the *swradio* application resides in **`benchmarks/source/swradio/`**. Common routines used by each of the stages is in `swradio-common/`, at the root of this directory. The implementations for each application (recall that these will each be compiled to run stand-alone on a single processor) reside in separate directories, named appropriately. Each of these components is structures in a manner similar to approach described in Section 9.2.3, and executes directly over the processor, in the absence of an operating system.

The file **`benchmarks/source/swradio/swr.m`** is an *architectural specification file (ASF)* for the software radio simulation. It defines the hardware architecture that is modeled by the emulator — the instantiated processors, their properties (memory size, clock speeds, and so on), the interconnect linking the processors and its properties, and so on.

The top-level directory of the swradio application contains a Makefile. Executing a `make` in this directory builds all the components of the partitioned application, and `make install` copies the resulting individual binaries to be loaded to the various processors (i.e., the `.sr` files), into the top level directory.

If you change the swradio application source and recompile, remember to copy *.sr from the subdirectories into the top-level directory containing the architectural specification file, for the changes to have any effect in the simulation. !

9.2 *Interaction Between Applications and Low-Level Machine State*

For those familiar with writing software for embedded microcontrollers, implementing, or porting operating systems for general purpose microcomputer systems, most of the topics of this section may be skipped. If familiar with ideas such as *memory maps* and *memory mapped I/O*, this section may be skipped to go directly to Section 9.2.5.

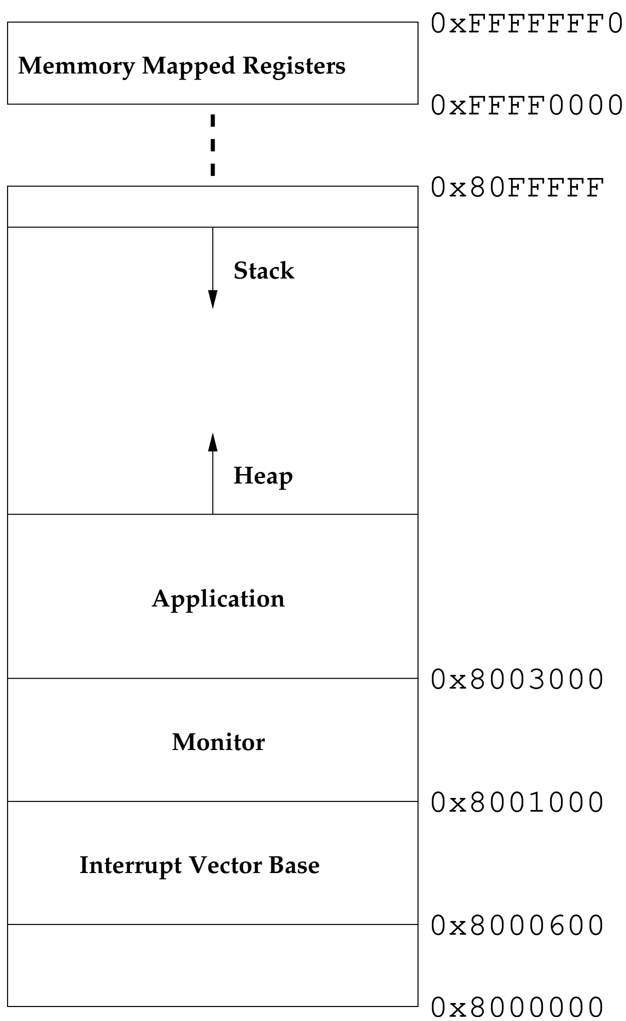


Figure 9.2: Memory Map of the Hitachi SH machine model in the Sunflower simulator.

9.2.1 Memory Map

The *memory map* of a system specifies the organization of the physical[1] address space seen by the processor.

Figure 9.2 illustrates the memory map of the Hitachi SH version of the architecture modeled by the Sunflower simulator. The base of the address space is at memory address `0x8000000`. The region of memory beginning at address `0x8000600` contains the *interrupt vector base*. On the occurrence of an *interrupt* (a hardware generated exceptional condition) or *exception* (a

[1] The discussion in this section sidesteps discussions of virtual memory organizations, as it is not of relevance here.

software generated exceptional condition), execution vectors to this address, and code in this region of memory is executed. Code at the interrupt vector base address must perform necessary saving of register state, determine the actual cause of the exceptional condition (i.e., the type of interrupt or exception raised) and call the appropriate routines to handle the condition. The type of interrupt or exception is determined by reading the `EXCP_INTEVT` or `EXCP_EXPEVT` *memory mapped registers* respectively.

Memory mapped registers are mapped to a separate region of memory starting at `0xFFFF0000` and ending at `0xFFFFFFF0`. The manner in which such memory mapped registers are accessed is described in Section 9.2.2. The mapping of registers to particular memory addresses are listed in **`sim/devsim7708.h`** in the simulator source distribution. In the simulator's implementation of the Hitachi SH architecture, in addition to architecture specified registers, several new registers have been added to provide interfaces to new facilities such as network interfaces, pseudo-random number generators, and the like.

The region of memory from `0x8001000` upwards is used to as application memory. The upper limit is bounded by how much memory is configured for a simulation. For example, for a configured memory size of 1 MB, as in Figure 9.2, the memory space spans `0x8000000` to `0x80FFFFF`. The default memory size is defined in the simulator source file **`sim/main.h`**; the size of modeled memory can be adjusted with the **`sizemem`** (C.140) simulator command.

In applications currently distributed with the simulator, the lower region of memory (from `0x8001000` to `0x8003000`), is typically used exclusively for a *monitor* or *firmware* application. The region of memory above `0x8003000` is used to hold general application code, followed by the application heap (growing upwards from the end of the application code) and the stack (for both the monitor and ordinary applications) growing downwards from the top of memory. The region of memory occupied by an application or the monitor, is further broken down into regions for code (`text`), initialized data (`data`) and uninitialized data or `bss`[2].

[2] The term `bss` is a historical vestige from UNIX. It stands for *Block Started by Symbol*.

9.2.2 *Memory Mapped I/O*

In the Hitachi SH architecture, several of the processor status facilities are implemented as *memory mapped registers*. These are essentially words in the memory space which when read, yield the value of a hardware system register. For example, the `EXCP_INTEVT` memory mapped register mentioned in Section 9.2.1 is a hardware register which is accessed by reading from memory address `0xFFFFFFD4`. Some memory mapped registers are byte addressed, others are word (16-bit) addressed, and yet others are long-word (32-bit) addressed. The header file `sys/kern/superH/sh7708.h` defines macros to enable easy access to all the memory mapped registers in the modeled Hitachi SH architecture.

In practice, applications executing over the simulator do not need to be concerned with these memory mapped registers, unless they wish to interact directly with built-in peripherals, or peripheral extensions to the architecture created by the user. Routines for simplifying the access to many of the peripheral devices are already implemented and provided with the simulator distribution. These routines can be found in `benchmarks/source/port/`, and are described in more detail in the following sections.

9.2.3 *Considerations for applications executing in absence of an operating system*

The emulator provides many facilities for executing off-the-shelf applications, including traditional computer architecture benchmark suites such as SPEC, MiBench and ALPBench, typically intended for execution over an operating system. In some applications however, it is desirable to expose more details of the underlying system architecture to applications. This is desirable when, e.g., implementing applications which interact with peripherals such as timers or network interfaces. In such applications which interact with hardware peripherals, application developers have two options — to employ an operating system (OS), or to interface applications directly to hardware. In many system evaluations, it is desirable to take the latter approach, removing from consideration any additional behaviors that may be introduced by an OS. This section details the interface to hardware seen by such applications. It is also of relevance to developers intending to port an operating system implementation to the simulation platform.

Applications executing in the absence of an operating system are generally constructed as a main event loop, with interrupts handled asynchronously by an interrupt handler. The primary challenge here is to ensure that data structures which are modified asynchronously do not adversely affect the execution of the main event loop. In the absence of an operating system, it is not possible to perform operations like sleeping on signals or scheduling events to be executed at a later time, unless a state machine of some sort is added to the application implementation. It is therefore necessary to use global variables to exchange information in both ways, between the main event loop and the interrupt handler. An important rule to follow is the following : *always declare variables to be used to exchange information between the main event loop and the interrupt handler as* `volatile`. This ensures that the C compiler will generate code that ensures that variable updates always occur, even when the compiler thinks such updates can be optimized away. This is important because, if the main event loop is something like the following,

```
int   flag;

flag = 0;
while (flag)
{
   print("hello");
}
```

then the compiler might think that since the variable `flag` is never updated in the body of the loop, it can decide not to generate code for the `while` loop in its *dead code elimination phase*. If the variable `flag` is modified by the interrupt handler, this will however be an incorrect optimization to make. To tell a C compiler that a variable might be changed asynchronously, such a variable must be marked as `volatile`. For example, the following is a corrected implementation of the above:

```
volatile int    flag;

flag = 0;
while (flag)
{
        print("hello");
}
```

9.2.4 *Register calling conventions on the Hitachi SH*

On the Hitachi SH, the first four *words* of arguments to a function are passed in registers R4 to R7, with subsequent arguments pushed on the stack, in reverse order, such that the first argument not passed in a register will be lowest in the stack;[3] arguments that are multi-word will take up multiple of these registers, and arguments may even partly reside in registers (R7) with the remainder on the stack. Function return values are passed in in R0.

[3] "Cygnus GnuPro Documentation".

9.2.5 *Interrupts generated by Sunflower*

The simulator generates many types of interrupts which can be disabled or must otherwise be handled by applications. Every millisecond, if enabled, a *clock interrupt* is generated, and on such an interrupt, the memory mapped interrupt code register, `EXCP_INTEVT` will have the value `TMU0_TUNI0_EXCP_CODE`. Similarly, *network interface interrupts* and *battery low interrupts* have the interrupt codes `NIC_RX_EXCP_CODE` `BATT_LOW_EXCP_CODE` respectively.

9.2.6 *Utility routines :* **devnet_xmit()**, **udelay()**

There are several utility functions, to interface to the peripherals modeled by the simulator. These utilities typically have the name `devXXX_YYY`, for example **`devnet_xmit()`** ϒ,
benchsrcdevnet_recv(); The
benchsrcudelay() routine provides a calibrated busy microsecond delay. Table 9.1 lists the currently available helper routines, the location of their implementation in the source tree, and the necessary header files that must be included to use them. These routines are currently not compiled into a library, but rather, must be complied together with applications that need them.

Routine	**Description**	**Source**	**Headers**
`int devexcp_getintevt(void)`	Get Interrupt event #	`benchmarks/misc/port/`	`"devexcp.h"`
`int devloc_getorbit(void)`	Get orbit	`benchmarks/misc/port/`	`"devloc.h"`
`int devloc_getvelocity(void)`	Get velocity	`benchmarks/misc/port/`	`"devloc.h"`
`int devloc_getxloc(void)`	Get x-location	`benchmarks/misc/port/`	`"devloc.h"`
`int devloc_getyloc(void)`	Get y-location	`benchmarks/misc/port/`	`"devloc.h"`
`int devloc_getzloc(void)`	Get z-location	`benchmarks/misc/port/`	`"devloc.h"`
`void devlog_ctl(uchar *cmd)`	Rabbit hole	`benchmarks/misc/port/`	`"devlog.h"`
`int devnet_xmit(uchar *dst, int proto,`	Transmit `data` `dst`	`benchmarks/misc/port/`	`"devnet.h"`
`uchar *data, int nbytes, int whichifc)`			
`void devnet_recv(uchar *recvbuf,`	Get data in RX buf.	`benchmarks/misc/port/`	`"devnet.h"`
`int nbytes, int whichifc)`			
`ulong devnet_getfsz(void)`	Get frame size	`benchmarks/misc/port/`	`"devnet.h"`
`ulong devnet_getncr(void)`	Get NIC status	`benchmarks/misc/port/`	`"devnet.h"`
`ulong devnet_getspeed(void)`	Get link speed	`benchmarks/misc/port/`	`"devnet.h"`
`int devnet_ctl(int cmd, int val)`	Configure NIC	`benchmarks/misc/port/`	`"devnet.h"`
`void devnet_framedelay(int nframes)`	Determine latency	`benchmarks/misc/port/`	`"devnet.h"`
`ulong devnet_getncolls(void)`	Get # collisions	`benchmarks/misc/port/`	`"devnet.h"`
`ulong devnet_getncsense(void)`	Get # carrier errs.	`benchmarks/misc/port/`	`"devnet.h"`
`ulong devrand_getrand(void)`	Get a random #	`benchmarks/misc/port/`	`"devrand.h"`
`void devrand_seed(ulong seed)`	Seed the rand. gen.	`benchmarks/misc/port/`	`"devrand.h"`
`ulong devrtc_getusecs(void)`	Get time in μs	`benchmarks/misc/port/`	`"devrtc.h"`
`void devtag_write(int which, Tag *t)`	Write Tag	`benchmarks/misc/port/`	`"devtag.h"`
`Tag devtag_read(int which)`	Read Tag	`benchmarks/misc/port/`	`"devtag.h"`
`ulong devtag_rttl(int which)`	Read Tag TTL	`benchmarks/misc/port/`	`"devtag.h"`
`void devtag_wttl(int which, ulong age)`	Set Tag TTL	`benchmarks/misc/port/`	`"devtag.h"`

Table 9.1: Helper routines often used within applications. These routines take out some of the drudgery of accessing modeled peripherals. For example, `devnet_xmit()` takes care of writing the supplied data to the NIC transmit register, word at a time.

9.3 *Implementation of Software Radio Application*

The directory `benchmarks/source/swradio/` contains the implementation of the swradio application. The top-level directory contains the subdirectories `swradio-demod`, `swradio-eq`, `swradio-lpf`, `swradio-sink` and `swradio-source`, corresponding to the implementations of the demodulator, equalizer, lowpass filter, sink and source stages of the application, as described previously in Section 9.1. The following sections describe the various components that go into the final compiled application.

9.3.1 *The* `Makefile`

The Makefile in the top-level directory drives the execution of the Makefiles in the subdirectories corresponding to each stage of the swradio application.

The Makefile in each subdirectory determines which source files are compiled into a given binary, their dependencies and the tools necessary for their compilation. Like most of the Makefiles for benchmarks and applications for execution over the simulated hardware, which are provided in the simulator distribution, each swradio stage's Makefile contains a variable, `PROGRAM`, set to the name of the primary C source file of the application. This makes it possible to copy over the Makefile for most of the examples, change the variable name, and add the appropriate new C source file, and just type *make* to build a new application. The variable `OBJS` specifies the list of object files that will be linked into the final binary, and the remainder of the Makefile provides rules for building these object files. One important point to note is that the object file, `init.o` should be the first in the object file list. This is because it is the assembled startup assembly code that must reside at the bottom of the final compiled binaries memory map.

9.3.2 *Startup code:* `init.S`

The `init.S` file contains the assembly startup code. It sets up the stack by setting register `R15` of the machine to contain the highest address in memory, then calls the C code, `startup()`. Note that the "initialization" or "main" routine in the C source must therefore be called `startup()`. The reason for not calling it `main()` has to do with the special treatment of the symbol `main` by C compilers, and is beyond the scope of this manual.

```
        .align  2
start:
        /*      Clear Status Reg        */
        AND     #0, r0
        LDC     r0, sr

        /*      Go !    */
```

```

        MOVL    stack_addr, r15
        MOVL    start_addr, r0
        JSR     @r0
        NOP

        /*      SYSCALL SYS_exit        */
        mov     #1, r4
        trapa   #34

   /*                      */
   /* Main body of code in l.S is not shown for brevity  */
   /*                      */

        .align  2
stack_addr:
        .long   (0x8000000 + (1 << 20))
start_addr:
        .long   _startup
```

9.3.3 *Implementations of the swradio stages — example:* `swradio-demod/swradiodemod.c`

The signal processing stages of the swradio application are implemented as self-contained applications, each executing over a single processor, which communicate by exchanging packets over interconnection links. The demodulator stage of the swradio pipeline is implemented in `swradio-demod/swradiodemod.c`.

It includes several header files and their dependencies, for interfacing with peripherals such as the network interface (**`sim/devnet-hitachi-sh.h`**) and the management of interrupts and exceptions (**`sim/devexcpt.h`**). Also included are header files containing macros for interacting with the hardware peripherals via memory mapped registers (**`sim/devsim7708.h`**), a header file defining abbreviations for type names, such as `uint`, `uchar` and so on (**`sim/sf-types.h`**), as well as some header files which are part of the simulator source (**`sim/network-hitachi-sh.h`** , **`sim/interrupts-hitachi-sh.h`**), which define various constants needed for interaction with the peripherals.

The entry into the demodulator implementation is via the function **`startup()`** . This function first installs the interrupt handlers (via **`hdlr_install()`** , which copies the assembly instruction defined between **`_vec_stub_begin`** and **`_vec_stub_end`** in `init.S` to the interrupt vector base), then proceeds through two phases of perpetually waiting for incoming packets, and processing them.

The incoming packets trigger the execution of the interrupt handler (**`intr_hdlr()`**), which determines the source of the interrupt using the facilities of the **`devexcp_getintevt()`** routine, whose declaration was included from `devexcp.h`, and whose implementation resides in **`benchmarks/source/port/`** . For network interrupts, this routine calls the network interrupt handler `nic_hdlr()`, which retrieves the oldest received packets from the receive FIFOs via the helper routine `devnet_recv()`, implemented along with the other helper "device drivers" in the directory `benchmarks/source/port/`.

9.4 *System Architecture Setup for Software Radio Application*

The system architecture of the simulated hardware platform for the swradio application comprises 12 processors connected in the topology previously illustrated in Figure 9.1. There are a total of 4 interconnect links, with two of them configured as point-to-point links (between nodes 0 and 1, and nodes 1 and 2), and the other two as shared media.

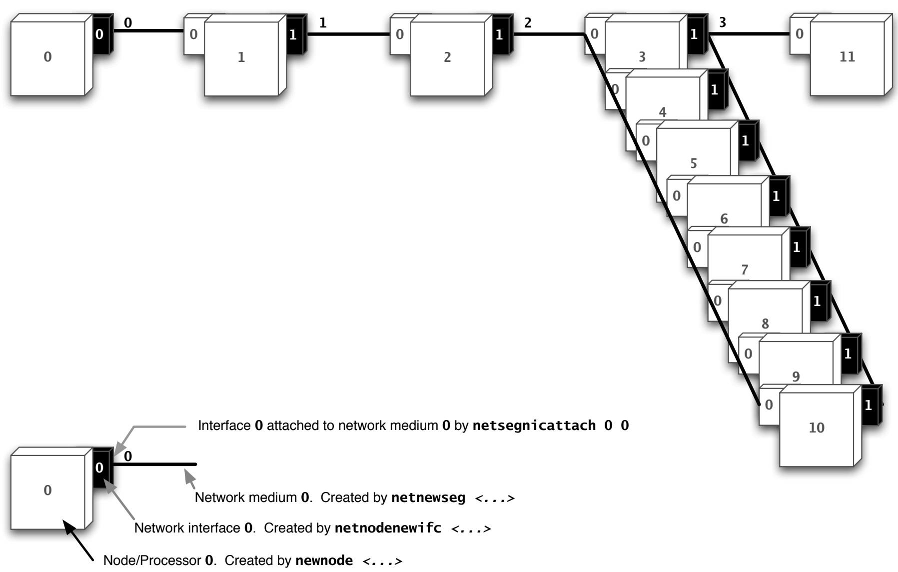


Figure 9.3: Organization of simulation components (processors, network interfaces, interconnects) for the swradio application.

The simulation interconnect topology for the swradio application is shown in Figure 9.3, and the system architecture configuration file which defines the hardware instances and interconnection links for the swradio application is shown below:

```
netnewseg       0 8192 300000000 100000000 0 0 0 0 0 0 0 0 0
netnewseg       1 8192 300000000 100000000 0 0 0 0 0 0 0 0 0
netnewseg       2 8192 300000000 100000000 0 0 0 0 0 0 0 0 0
netnewseg       3 8192 300000000 100000000 0 0 0 0 0 0 0 0 0


clockintr       1
cacheoff
ff
netnodenewifc      0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 0
sizemem         3000000
srecl        swradiosource.sr
```

```
run


newnode         superH 0 0 0 0 0
clockintr       1
cacheoff
ff
netnodenewifc      0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       0 0
netnodenewifc      1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       1 1
sizemem         3000000
srecl           swradiolpf.sr
run


newnode         superH 0 0 0 0 0
clockintr       1
cacheoff
ff
netnodenewifc      0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       0 1
netnodenewifc      1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       1 2
sizemem         3000000
srecl           swradiodemod.sr
run


newnode         superH 0 0 0 0 0
clockintr       1
cacheoff
ff
netnodenewifc      0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       0 2
netnodenewifc      1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       1 3
sizemem         3000000
srecl           swradioeq.sr
run


newnode         superH 0 0 0 0 0
clockintr       1
cacheoff
ff
netnodenewifc      0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       0 2
netnodenewifc      1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach       1 3
sizemem         3000000
srecl           swradioeq.sr
run


newnode         superH 0 0 0 0 0
clockintr       1
cacheoff
```

```
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem       3000000
srecl         swradioeq.sr
run


newnode       superH 0 0 0 0 0
clockintr     1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem       3000000
srecl         swradioeq.sr
run


newnode       superH 0 0 0 0 0
clockintr     1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem       3000000
srecl         swradioeq.sr
run


newnode       superH 0 0 0 0 0
clockintr     1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem       3000000
srecl         swradioeq.sr
run


newnode       superH 0 0 0 0 0
clockintr     1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem       3000000
```

```
srecl          swradioeq.sr
run


newnode        superH 0 0 0 0 0
clockintr      1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 2
netnodenewifc     1 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      1 3
sizemem        3000000
srecl          swradioeq.sr
run


newnode        superH 0 0 0 0 0
clockintr      1
cacheoff
ff
netnodenewifc     0 0.250 0.250 0 0 0 0 0 1024 1024
netsegnicattach      0 3
sizemem        3000000
srecl          swradiosink.sr
run
```

The architectural specification file shown above first instantiates several interconnect links, via the **netnewseg** (C.60) command in the appendices details the arguments to the command).

Many of the commands in the simulator's command language are *modal*. This means that, they act within a given context, more specifically, within the context of the given *current processor/node*. The commands following the group of **netnewseg** (C.60) commands (**clockintr** (C.24) , **cacheoff** (C.21) and so on) act on the default instantiated processor; subsequent processors are instantiated with the **newnode** (C.70) command.

The first node in the swradio pipeline (the *source* node) has only one network interface instantiated (via a **netnodenewifc** (C.61) command), while subsequently instantiated nodes have two network interfaces. The **netsegnicattach** (C.67) command is used to connect instantiated network interfaces to instantiated interconnect segments. The memory for each instantiated node is resized (via **sizemem** (C.140)) to match the memory map expected by the compiled swradio application, and the appropriate binary is loaded into the memory of the simulated processor (via **srecl** (C.142)).

In general, the step necessary for creating a simulation architecture definition for a network of processors simulation involves:

1. Creating the necessary network links with the **netnewseg** (C.60) command. Properties of the link such as frame size, link speed (transmission

delay), propagation delay, failure probability, mean failure duration may supplied as arguments to the instantiation.

2. Creating the necessary nodes with the **newnode** (C.70) command. One may specify various parameters for each node such as its operating voltage, frequency, cache size and configuration, failure probability, etc.

3. Instantiating network interfaces on the nodes. A node may have multiple network interfaces.

4. Connecting each network interface on each node to a particular instantiated link. This step determines, in essence, the topology; By using different connections, one can model a shared bus, point to point links, a torus, hypercube, mesh, etc.

10

Non-Uniform Memory Accesses Latencies, Memory Remapping, and Memory Tracing

With multiple processors, by default, each processor has its own local memory and local cache.

10.1 *The* **mmap** *Command*

The **mmap** (C.54) command creates a shared memory region across two processors, but there is no mechanism to ensure cache coherence *per se*. However, because of the way the cache is modeled[1], there will not actually ever be a coherence problem in a running application.

[1] Sunflower only keeps track of the cache's tag array and data is not really stored in the cache.

10.2 *The* **numa*** *Commands*

Another thing that might be of use to you is the modeling of *non-uniform memory access regions* in the simulator (see the commands **numaregion** (C.75), **numasetmapid** (C.76), **numastats** (C.77), and **numastall** (??)).

You can use the **numaregion** (C.75) command to tell the emulator that all memory accesses to some range of addresses on processor A, should be mapped to the memory of processor B, at a given offset from where the address range actually is, and with specific latencies when read/written locally, and when read/written remotely.

Since you can have applications running over the simulator issue commands to the simulator (see, e.g., `benchmarks/source/libsfpthread/spthr_simcmd.c`) you can have your application, which is running over the simulator, map a particular data structure (or even a single variable) to a remote memory:

```
snprintf(buf, buflen, "numaregion \"%.128s\" 0x%lx 0x%lx %d %d %d %d %d 0 %d\n",
             name, &mystruct, &mystruct+sizeof(mystruct), lrlat, lwlat, rrlat,
  rwlat, id, private);

spthr_simcmd(buf);
```

The above will make all subsequent accesses to the data structure **`mystruct`** have memory read and write latencies **`lrlat`** and **`lwlat`** on the local processor, and **`rrlat/rwlat`** on the remote processor which has identifier **`id`**.

11
Stochastic Processes in Emulation and Simulation Experiments

Underlying both the modeling of bit-level and whole-system failures in the Sunflower simulator, is the generation of random events. For simulations which have long duration, it is desirable for any pseudo-random sequences they employ to not repeat. While useful for some simple applications, the standard C library pseudo-random number generation routines provided with most operating systems do not provide sufficiently high periodicity when simulating billions of random events. Fortunately, the research literature contains better solutions for generating pseudo-random numbers, and we have incorporated one of these[1] in the simulator.

[1] Nishimura 2000.

11.1 Generating Random Variates from Different Distributions

Pseudo-random number generators typically generate values uniformly distributed on some support set. However, during systems evaluation, it is often desirable to use random numbers drawn from some other distribution, such as, e.g., a Gaussian, χ^2, or a heavy-tailed distribution like the Pareto distribution. Standard textbook methods[2] facilitate transforming random variates drawn from one distribution (e.g., uniform) to obtain random variates drawn on a different distribution (e.g., exponential). The Sunflower simulation environment implements the generation of random variates from over twenty different distributions, shown in Figure 11.1. These distributions (with appropriate parameters,) can be used by a system architect to define the distribution of time between failures, duration of failures, or locations of logic upsets. *The current implementation of these random value generators is unfortunately rather slow for all but uniform and exponential distributions.*

[2] Ross 2001.

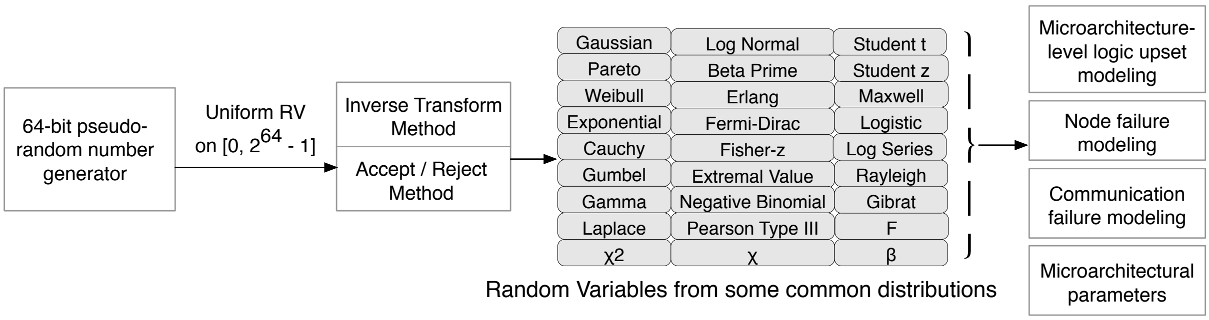


Figure 11.1: A high periodicity 64-bit pseudo random number generator is used as the basis for generating random variates from a large set of distributions. In addition to the distributions listed here, a distribution having a "bathtub" shaped hazard function is also provided.

11.2 *User-Defined Discrete Distributions*

The emulator also permits the runtime (from the command line) definition of discrete distributions based on the built-in distributions, by specifying a range of basis values, the inter-basis-point distance, and the built-in distribution for assigning probabilities to these basis points. Such distributions are defined using the `initrandtable` ⌨ (C.46) command. Furthermore, arbitrary discrete distributions can be defined by defining a set of basis points and associated probabilities, using the `defndist` ⌨ (C.27) command.

11.3 *Stochastic Configuration Constants and Emulator Configuration Variables as Random Variates*

While not currently activated for all parsed command arguments in the simulator, some simulator commands that semantically require a floating point value can instead take a specification for a random constant drawn from a specified distribution. Examples of this can be see in the `T_NEWNODE` grammar production in `sim/sf.y`.

A further extension of this idea is the ability to instruct the simulator to randomly change the values of internal simulation state, with specified distributions for the delay between updates and the values supplied. This is implemented in the `registerrvar` ⌨ (C.99) command. The implementation of this facility has a lot of room for improvement, and has not yet been extensively tested.

12

Input and Output File Formats

A variety of file formats are used as input and generated as output by the emulator. All the file formats are plain text, and are intended to be both easily human-readable, as well as easily processed by machine.

12.1 *The Configuration File* `conf/setup.conf`

The configuration file `conf/setup.conf` (relative to the root of the simulator source tree) defines forms the backbone of the simulator installation configuration. It defines the location of the installation for all utilities that need this information (in the variable `SUNFLOWERROOT`).

The host machine architecture is defined in the variable `HOST`, to elide the need for guessing it in the build process. The variable `TARGET` defines a *general name* for the default target architecture, while the variable `TARGET-ARCH` defines the specific target architecture and binary format configuration name as used by Gnu tools such as GCC and Binutils. Similarly, the lists `SUPPORTED-TARGETS` and `SUPPORTED-TARGET-ARCHS` define the list of cross-compiler configurations that can be automatically built using the setup provided by the simulation infrastructure.

```
##
## You will want to change the following to suit your setup:
##
SUNFLOWERROOT  =  /tmp/sunflower-1.0-release-source-beta.3

HOST     = powerpc-apple-darwin9
TARGET      = superH
TARGET-ARCH = sh-coff
TARGET-ARCH-FLAGS = -DM32

##
## You do not necessarily need to change this stuff:
```

```
##
GCCINCLUDEDIR   = $(SUNFLOWERROOT)/tools/source/gcc-3.2.3/gcc/ginclude/
PREFIX      = $(TOOLS)/$(TARGET)

TOOLS    = $(SUNFLOWERROOT)/tools
TOOLSBIN = $(TOOLS)/bin
TOOLSLIB = $(SUNFLOWERROOT)/tools-lib
APPS     = $(SUNFLOWERROOT)/apps

CC       = $(TOOLSBIN)/$(TARGET-ARCH)-gcc
CXX      = $(TOOLSBIN)/$(TARGET-ARCH)-g++
F77      = $(TOOLSBIN)/$(TARGET-ARCH)-g77
PROLACC     = /usr/local/bin/prolacc
LD    = $(TOOLSBIN)/$(TARGET-ARCH)-ld
AR    = $(TOOLSBIN)/$(TARGET-ARCH)-ar
OBJCOPY     = $(TOOLSBIN)/$(TARGET-ARCH)-objcopy
OBJDUMP     = $(TOOLSBIN)/$(TARGET-ARCH)-objdump
AS    = $(TOOLSBIN)/$(TARGET-ARCH)-as
SIZE     = $(TOOLSBIN)/$(TARGET-ARCH)-size
GCCLIB      = gcc
MAKE     = make
RM    = rm -rf
DEL      = rm -rf
LOADER      = $(SUNFLOWERROOT)/loaders/superHload/shload


SUPPORTED-TARGETS=\
      msp430\
      superH\
      ppc\
      arm\
      sparclite\
      mcore\
      v850\
      coldfire\
      h8\
      avr\
      x86\

SUPPORTED-TARGET-ARCHS =\
         msp430\
         sh-coff\
         powerpc-eabi\
         sparclite-coff\
         arm-elf\
         mcore-pe\
         v850-coff\
         h8300-hitachi-hms\
         avr\
         m68k-coff\
         i386-aout\
```

12.2 *The Configuration File* **sim/config.h**

This file contains a set of compile-time flags for enabling various facilities that may not be needed by casual users, but which may have a large effect on simulator performance:

```
#define M32

#define SF_AUTO_QUANTUM         0
#define SF_CHATTY               0
#define SF_PHYSICS              1
#define SF_DEBUG                0
#define SF_NETWORK              1
#define SF_MOBILITY             0
#define SF_SIMLOG               1
#define SF_PAU_DEFINED          0
#define SF_BITFLIP_ANALYSIS     0
#define SF_POWER_ANALYSIS       1
#define SF_MEMTRACE             0
#define SF_BATT                 1
#define SF_BATTLOG              0
#define SF_FAULT                0
#define SF_DUMPPWR              0
#define SF_VALUETRACE_ANALYSIS  0
#define SF_FT_TANDEM            0
#define SF_BPTS                 1
#define SF_TRAJECTORIES         0
```

12.3 *The Configuration Files* **sim/config.$OSTYPE**

The simulator build process uses the file **config.***ostype.machtype*, where *ostype* is one of **OpenBSD**, **darwin**, **darwin9.0**, **linux**, **posix** and **solaris**, and *machtype* is one of **i386**, **ppc** and sparc, to determine platform-specific configuration for compiling the simulator. This configuration file also defines any platform-specific flags required in the build process, as well as possible platform-specific optimization flags. The contents of the `sim/config.darwin-ppc` file are shown below:

```
CC                          = gcc
GAWK                        = gawk
LINT                        = echo
LD                          = ld
CC                          = gcc
BISON                       = bison
ENDIAN                      = SF_B_ENDIAN
PLATFORM_CFLAGS             = -no-cpp-precomp -arch ppc -Wno-long-double
```

```
        -Wmost -Wno-four-char-constants -Wno-unknown-pragmas
        -pipe -multiply_defined suppress -malign-natural -D$(ENDIAN)
PLATFORM_LFLAGS         = -lpthread
PLATFORM_OPTFLAGS       = -fast -mcpu=7450
```

12.4 Architecture Specification Files

The *architectural specification files (ASFs)* or *simulator command files* contain lists of simulator commands, and are typically used in defining a system configuration for simulation. The only formatting constraint on ASFs is that they can contain only valid simulator commands, at most one command per line. Comments are introduced with two minus characters, "**-**", and continue until the next newline.

12.5 The Output Log File **sunflower.out**

The simulator log file, **sunflower.out** is written to disk whenever the simulator exists. It is a plain text file consisting of four tab-separated columns. The first column is a whitespace-free string of the form **Node%d**, where **%d** denotes an integer node ID; the file contains summary statistics for all modeled processors, and this column is used to distinguish between the information for the various processors. The second column is a string identifying the statistic in question, and may contain whitespace. The third column contains the character **=**, and the fourth column is the value of the summary statistic.

12.6 Signal Source Sample Values File

The signal source samples file is a plain text file which specifies a number of samples of a modeled signal, as floating point values. It contains as its first line the number of sample values in the file, and the remainder contains the sample values. The rate at which these sample values are used to update a modeled signal, as well as the option of whether or not the values are looped in simulation, is specified as the **samplerate** parameter to the **sigsrc** (C.138) command which references the signal source samples file.

12.7 Signal Source Trajectory File

The signal source trajectory file is a plain text file which specifies a list of way points (x-, y- and z- location) of a modeled signal. It contains as its first line the number of location values in the file, and the remainder contains the location coordinates. The rate at which the locations are used to update a signal model, i.e., the rate of motion of the signal source, is defined in the `sigsrc` (C.138) command which references the signal source trajectory file as the **`trajectoryrate`** parameter.

12.8 Node Location Trajectory File

The node location trajectory file specifies the motion and change in heading of a system in its environment. The first line of the node trajectory file specifies the number of samples within the file, and the remainder of the file contains a list of tuples of x-, y-, z-location and heading (in degrees, with 0 degrees being a heading to a common "north" reference).

12.9 Network Trace Log File

The network trace log file contains a dump of the traffic traversing a network. It may also contain additional markers to enable the correlation of the data frames represented within the trace, with operating occurring within the simulated processor, such as with state within simulated applications.

Each dumped data frame within the trace consists of nine field: a timestamp, the actual frame data, an indicator of the frame size, indicators of the source and destination nodes, a broadcasts indicator, information about which of the senders possibly-multiple network interfaces generated the data frame, and an indicator as to whether the frame originated from a device being simulated on another simulation host (relevant only to distributed simulations. All other data in the trace file is preceded with a comment indicator ("--"), An example snippet from a trace log file showing a captured data frame, as well as a marker containing various statistics about the state of the sending node prior to the generation of the frame, is shown below:

```
--Tag NODE3_NETTRACEMARK_TAG_4{
--Node3     "ICLK"   =  16529673
--Node3     "CLK" =  1030616
--Node3     "TIME"   =  4.132418E+00
--Node3     "dyncnt" =  1030616
--} Tag NODE3_NETTRACEMARK_TAG_4.

Timestamp: 4.133721E+00
Data: 33 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3A 3A 31 00 2F E6 4F 22
7F C4 6E F3 61 E3 71 FC 00 00 00 36 00 00 3F 0D 54 33 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 33 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 41 6C 76 8D 41 6C 76 8D 41 6C 76 8D 00 3F 0D 54 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 .
Bits left: 0x00000400
Src node: 0x00000003
Dst node: 0xFFFFFFFE
Bcast flag: 0x00000001
```

```
Src ifc: 0x00000000
Parent netseg ID: 0x00000000
from_remote flag: 0x00000000
```

13 Cross-Compilation Toolchain

Benchmarks executing over the emulator are compiled with GCC and linked against relevant libraries (usually, Newlib, the embedded C library). All the necessary configuration is in place to enable you to build the GCC cross-compiler by issuing the command `make cross` from the root of the simulator source tree, after having performed the requisite editing of the simulator configuration file **`conf/setup.conf`** as described in Chapter 1.

13.1 Miscellaneous Notes and Pointers

This section contains various notes and observations made in getting the cross-compilers to build on various platforms. All of the observations made here have already been integrated into the source tree. This information is provided here as it might aid users who run into similar issues on new platforms.

The cross-compiler tools are currently setup by default to only build the C compiler, `gcc`, and not the C++ compiler, `g++`. The cross compiler can be used to generate a C++ compiler, and indeed it has been used in that manner in the past, with `g++` and libstdc++. The current **`tools/Makefile`** contains the necessary rules to pursue this path. Additional changes required include adding "c++" to the "languages=" option, and possibly to employ the rule command **`make all-gcc`** within **`tools/Makefile`** for building gcc rather than just `make`.

Initial attempts to build Binutils 2.16.1 on MacOS 10.4 (Intel) failed, because the MacOS `make` has an implicit rule for handling ".m" files (the MIME type for Objective-C files in MacOS), and this is not what is needed for one of the rules in building gprof. See `http://sources.redhat.com/ml/binutils/2005-12/msg00085.html` for a discussion. This problem was solved by adding the "-r" flag to the build of Binutils in the Makefile, which causes make to ignore implicit rules.

The `-disable-nls` flag was added to the Binutils configure since we don't need internationalization

In the past, the final stage in the build of gcc-4.1.1 broke due to something related to libssp. As libssp is not needed, it has also been disabled in the configure flags.

Other items to disable in the configure flags in the future, include the building of the man pages for the cross compiler.

14
Benchmarks

The Sunflower emulator is supplied with a number of benchmark applications, in source and binary form. The benchmarks come from a variety of application domains of relevance to the architectures that can be modeled using Sunflower — traditional high-performance workstation benchmarks (SPEC CPU 2000, Section 14.1), high performance embedded applications (Sphynx speech recognition (Section 14.2), MPEG encoder/decoder (Sections 14.3 and 14.4), MiBench (Section 14.5) and ALPBench (Section 14.6) benchmark suites), multiprocessor systems-on-chip (multi-processor partitioned software-defined radio benchmark (Section 14.8), multi-processor Pthreads library (Section 14.9)) and resource-constrained wired and wireless networked embedded systems (sensor network benchmarks, Section 14.7), as well as a trivial example, bubblesort (used as a running example in Chapter 3).

14.1 The SPEC CPU 2000 Benchmarks

A subset of the SPEC CPU 2000 benchmarks (*ammp*, *art*, *bzip2*, *cc1*, *equake*, *gzip*, *mcf*, *parser*, *vortex*, *vpr*) are provided in pre-compiled binary form, along with their reference and *reduced inputs* (where available), and appropriate default simulator configuration files. Unlike all the other benchmark suites supplied with the Sunflower simulator, the SPEC benchmarks are not provided in source form in the standard distribution, due to license restrictions. Holders of source licenses for SPEC CPU 2000 can however obtain a copy of an appropriately configured SPEC 2000 source distribution for compilation for the simulator. The SPEC benchmark binaries, their inputs, and the simulator configuration files needed to run them reside in `benchmarks/dist/SPEC2000` 🗁.

14.2 The CMU Sphynx3 Speech Recognition Benchmark

The source and input files for the CMU Sphynx3 benchmark implementation reside in `benchmarks/source/sphynx3/` 🗁.

14.3 *The MPEG2 Encoder Benchmark*

The source and input files for the MPEG2 encoder benchmark implementation reside in `benchmarks/source/mpegencoder/` 🗁.

14.4 *The MPEG Decoder Benchmark*

The source and input files for the MPEG2 decoder benchmark implementation reside in `benchmarks/source/mpeg2dec/` 🗁.

14.5 *The MiBench Benchmark Suite*

The source and input files for the MiBench benchmark suite implementation reside in `benchmarks/source/MiBench/` 🗁.

14.6 *The ALPBench Suite*

The source and input files for the ALPBench benchmark suite implementation reside in `benchmarks/source/ALPBench/` 🗁.

14.7 *A Sensor Network Benchmark Suite*

The source and input files for a collection of wireless sensor network applications reside in `benchmarks/source/sbench/` 🗁.

14.8 *The Software-Defined Radio Benchmark*

(See source tree.)

14.9 *The Sunflower Pthreads Subset Implementation and Software-Defined Radio Pthreads implementation*

(See source tree.)

15 Utilities

15.1 logmarkparse

The `logmarkparse` utility can be found in **`utils/logmarkparse/`**.
Usage:

```
logmarkparse <tagstub> <start tag> <end tag> <statistic name> <stub min suffix> <stub max suffix> <netflag>
```

For example, to capture "TIME" values for entries for "NODE0" through "NODE24", which are not network logs:

```
logmarkparse    NODE    _LOGMARK_TAG_2    _LOGMARK_TAG_3    TIME    0    24    0    sunflower.out
```

Appendices

A
Frequently answered questions

A.1 Defining complex memory maps with different memory access latencies

Can I define complex memory maps, with, say multiple memories with different latencies and power consumption ?

Yes. This can be achieved using the `numaregion` (C.75) command to alter the local/remote read/write latencies and read/write power consumption properties of a range of memory addresses.

A.2 Extracting the archives downloaded from the web page

I am having trouble extracting the archive from the web page. I tried **`tar -zxvf *.tgz`**, *but the error message is as follows:*

```
gzip: stdin: not in gzip format
tar: Child returned status 1
tar: Error exit delayed from previous errors
```

The problem you are facing is that you are trying to uncompress a bzip2 archive using the gzip filter in the tar utility. Obviously will not work. Instead, try:

```
bunzip2 -c sunflower-release-source-beta-3.tar.bz2 | tar xvf -
```

and if you are reading this particular FAQ entry, then maybe it should be pointed out to you that the trailing "-" in the above is not a typo.

A.3 General problems compiling the tools

I'm having trouble compiling the simulator. It is making me really sad.

If having problems compiling the tools, always first check to make sure that your simulator distribution configuration file is correctly setup:

- **Check to make sure the `SUNFLOWERROOT` in the `conf/setup.conf` configuration file is set to point to the location of the source tree.**

- **Check any lines you edited in `conf/setup.conf` to make sure no extra whitespace was introduced at the ends of the lines.**

A.4 Behavior of Sunflower — "nothing happens"

When I load my program in Sunflower, it is not doing anything: it just print some text into beginning, and then it waits ... and when I press a key ends

All commands issued at the simulator's command interface return immediately even the initiation of a simulation — they *do not* block until the command completes. The simulator is implemented using two threads: (1) the interactive command line interface and (2) the simulation engine. While you are running a simulation you can still type commands at the simulator prompt — the simulation is running in the background. It is likely that you believe your simulation is over since you press a key and you get the command line back ? That is not the case; your program is still running in the simulator. Unless the simulator prints messages about "Stopping Simulation" or some similar message about simulation completion, then your benchmark is still running — in the background.

A.5 Crashing benchmarks — function calls in interrupt handler

My benchmark crashes when I put function calls in the interrupt handler.

Not all functions in the standard C library are reentrant, so calling functions such as `printf()` from within the interrupt handler, either directly or indirectly, is pushing your luck (this is not specific to the simulator). Consider a case where the main body of the benchmark is inside `strtok()` (a C library routine that keeps state between calls), and the interrupt handler also calls `strtok()`. Much confusion will ensue.

A.6 Relation between `CLK` and `ICLK`

What is the relationship between `CLK` *and* `ICLK` *that are output by the* ***`showclk`*** *(C.136) command?* `CLK` *always seems to be the same as the dynamic instruction count. The* `ICLK` *is always larger, but by a different factor for each node. How can I use either of these to determine the overall performance (runtime) of a benchmark?*

`CLK` is the number of cycles for which the processor is actively executing instructions or stalled on a cache miss, but not including when processor

is idle upon executing a SLEEP instruction (Hitachi SH). `ICLK` includes all clock cycles, including cycles during sleep.

A.7 Adding new memory-mapped registers to modeled machine

How do I add new memory-mapped registers to the modeled machine, to let me implement, say, performance counters?

Adding registers to the Hitachi SH machine (not for the modeled TI MSP430 machine):

1. Edit `devsim7708.h`, and add a new entry in the enumeration. The easiest to add is an 8 bit memory mapped register, e.g., like `SUPERH_NIC_NMR`. If you want to add a multi-byte register, you'll need to add two entries for the start and end byte addresses of the register. (e.g., see `SUPERH_USECS_*`)

   NOTE: make sure you add the entries to the bottom of the enumeration, and not somewhere in the middle, as that would change where the other registers are mapped in memory.

2. How do you want your new memory-mapped register to be accessed ? With byte-, word- or longword-accesses ?

   You will now have to add code to `devsim7708.c`, to handle cases where your new memory-mapped register is being written to or read, by byte-, word- and longword memory accesses.

3. If you've done the above two, then applications running over the simulator can now write and read from the memory mapped register.

   Sample application code to read/write from a memory mapped register:

the following function `devlog_ctl()`, which would be running *above* the simulator, takes a string argument (`uchar *cmd`) and writes the individual characters of the string into the `SUPERH_SIMCMD_DATA` memory-mapped register, then writes '0' into `SIMCMD_CTL`.

```
#include "e-types.h"
#include "tag.h"
#include "devsim7708.h"
#include "sh7708.h"

/*                          */
/*          Simulator control       */
/*                          */
#define SIMCMD_DATA     ((volatile unsigned char *) SUPERH_SIMCMD_DATA)
#define SIMCMD_CTL      ((volatile unsigned char *) SUPERH_SIMCMD_CTL)

```

```
void
devlog_ctl(uchar *cmd)
{
   int   i, cmdlen;

   cmdlen = strlen(cmd);
   for (i = 0; i < cmdlen; i++)
   {
      *SIMCMD_DATA = *(cmd + i);

      if (*(cmd + i) == '\n')
      {
         break;
      }
   }
   *SIMCMD_CTL = 0;

   return;
}
```

The details of how applications are compiled for execution over the simulator is covered in the main text of the manual.

A.8 Compiling the SPEC CPU 2000 benchmarks

How do I compile the SPEC benchmarks for the emulator?

To compile the SPEC benchmarks, you will need the sources. Only the SPEC CPU 2000 benchmarks have been built for the simulator, though newer versions of the suite might work as well. If you can prove you have a SPEC CPU 2000 source license, you can obtain the contents of the directory `benchmarks/source/SPEC2000/` from the maintainers of the Sunflower simulator. To build the SPEC benchmarks, after building the cross-compilers as described elsewhere in the manual, change directory to `benchmarks/source/SPEC2000/` and perform `make TREEROOT = full-path-to-simulator-distribution`, where `full-path-to-simulator-distribution` is the directory where the simulator is installed.

A.9 What is `NIC_OUI`?

What is `NIC_OUI` *? I saw in the sample code that you used it to get the ID. But you decrement it by '0'. Is* `NIC_OUI` *an integer or a char? If it is a char, does that mean that the range of IDs is limited?*

`NIC_OUI` is a 16-byte (128 bit) per-node address (*Organizationally Unique Identifier* or OUI, the term often used for MAC addresses e.g., Ethernet MAC addresses). What is done in the code (e.g., `my_id` in the `benchmarks/source/swradio/` examples), is that we convert this 16 byte address into an integer. `NIC_OUI` is a memory-mapped register, so to read the full 16 bytes you would do:

```
... = *(NIC_OUI+0);
... = *(NIC_OUI+2);
...
... = *(NIC_OUI+15);
```

to get all 16 bytes. The 16 byte OUI in many benchmarks is the string representation of the decimal node ID, i.e., there are a maximum of 10^{16} possible node IDs for those benchmarks that use this translation between node ID and OUI.

What the 'for' loop to calculate `my_id` in the `benchmarks/source/swradio/` directory does is, it converts from a string representation of a decimal, to a decimal. To convert, each character of the string has the ASCII value of '0' subtracted from it. That is, if you have the string `char *mystring[] = {"165"}`, you can convert it to an integer by:

```
my_int = (my_string[0]-'0')*100 + (my_string[1]-'0')*10 + (my_string[2]-'0')*1;
```

A.10 Adding new memory-mapped registers to the Hitachi SH architecture

From what I gather, we can add registers to the simulator which will then be visible to programs running over the simulator?

Yes, that is correct.

A.11 Changing voltage/frequency from within applications running over simulator

How can we change the voltage/frequency until the next timer interrupt?

Programs running over the simulator can issue any command that is available from the command line (type `help` (C.43) at the simulator command prompt to get the complete list). They do so by writing to the `SUPERH_SIMCMD_CTL` and `SUPERH_SIMCMD_DATA` memory-mapped registers. You can figure out this address by looking at the enumeration in `devsim7708.h`.

The default configuration of the simulator will scale frequency linearly with operating voltage, i.e., the Vt, K and α of the delay equation are 0.0, 5.5E-8 and 2 respectively, to get linear scaling for an operating voltage (Vdd) of 3.3 V and frequency of 60 MHz. You can set the Vt, K and α from the command prompt; "man setscale*" from within the simulator.

All the extant memory mapped registers are listed in the enumeration in `devsim7708.h`. You can figure out the actual addresses from the enumeration.

A.12 *Why does the swradio benchmark stop after 1024 samples ?*

Why does the swradio benchmark stop after 1024 samples ?

The benchmark that is being simulated is a streaming application, so technically, the benchmark will continue running forever. The benchmark was therefore setup so that after it processes 1024 samples, it signals the simulator to stop, using the `devlog_ctl()` interface and the `quit` (C.95) command.

A.13 *Errors opening .sr files in the software-defined radio example*

I get error messages such as " **`Open of "swradio*.sr" failed...`** *"*

```
./mconsole-linux-2.4-suse
load swradio/ece743HW4.m (in Sunflower)

[ID=0 of 1][PC=0x8000000][3.3E+00V, 6.0E+01MHz] load swradio/swradio.m
[M] Loading swradio/swradio.m...
[M] Cache deactivated
[M] Set memory size to 2929 Kilobytes
[M] Open of "swradiosource.sr" failed...

[M] args = [], argc = 0
[M] R4 = [0x00000000], R5 = [0x082cc5c0]
[M] Running...

[M] New node created with node ID 1
[M] Cache initialized with zero size
[M] Cache deactivated
[M] Set memory size to 2929 Kilobytes
[M] Open of "swradiolpf.sr" failed...
```

The simulator could not find the binaries for the code to be simulated on each node. In this particular example, the binaries to be loaded in the memory of each processor are given as relative paths, and needed to reside in the same directory as that from which the simulator was invoked (or the otherwise current directory, changed via the `cd` (C.23) command within the simulator).

A.14 *Simulation stopped with a "FATAL" message*

The simulation halted with a message of the form **`Sunflower FATAL (node 0)`**, *followed by a page of binary and hexadecimal numbers. Did the simulator crash ?*

No, the simulator did not crash. What you are observing is that your application, which is executing *o*ver the simulator, has performed an illegal operation (e.g., accessed an invalid memory address); the simulator has therefore printed out relevant state of the simulated machine to help you in debugging your application, and has halted. You can probe the state of the simulated machine by issuing the appropriate commands from the simulator console. An example of this output is shown below:

```
[ID=0 of 1][PC=0x8004000][3.3E+00V, 6.0E+01MHz]
Byte access (read) at address 0x0
Sunflower FATAL (node 0) : <Invalid byte access.>


FATAL (node 0): P.EX=[MOVBP]
R0              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R1              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R2              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R3              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R4              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R5              0000 1000 0000 1110 1111 1111 0000 0000   [0x080eff00]
R6              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R7              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R8              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R9              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R10             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R11             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R12             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R13             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R14             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R15             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_0        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_1        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_2        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_3        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_4        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_5        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_6        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
R_BANK_7        0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
SR              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
SSR             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
GBR             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
MACH            0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
MACL            0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
PR              0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
VBR             0000 1000 0000 0000 0000 0000 0000 0000   [0x08000000]
PC              0000 1000 0000 0000 0100 0000 0000 1110   [0x0800400e]
SPC             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
TTB             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
TEA             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
MMUCR           0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
PTEH            0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
PTEL            0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
TRA             0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
EXPEVT          0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
```

```
INTEVT          0000 0000 0000 0000 0000 0000 0000 0000   [0x00000000]
SLEEP = [NO]
Stopping execution on node 0 and pausing simulation...


[ID=0 of 1][PC=0x800400e][3.3E+00V, 6.0E+01MHz]
```

The output, shown above, is a dump of the contents of the simulated machine's register file, as well as the contents of various relevant system registers (shown above for the Hitachi SH architecture). To determine the cause of your program's untimely demise, there are a few items from this dump that are helpful:

- The first few lines of the dump often indicate the cause. In this case, the first text of the dump indicates `Byte access (read) at address 0x0`, i.e., the application was attempting to access memory at address `0`.
- In the case of the Hitachi SH architecture, the first things to check in the case of an illegal memory access are the stack pointer (`R15`) and the frame pointer (`R14`). These should point to values near the top of the address space. In the above example, they are both zero. This likely means the stack was not setup by an appropriate assembly language initialization before C code begun execution.
- Again for the Hitachi SH architecture, if the stack and frame pointers look sensible, the next items to check are the `INTEVT` and `EXPEVT` registers. These indicate the status of any interrupts or exceptions. A possible cause of failure might be that you have enabled the generation of interrupts of some kind or the other, but do not have interrupt handling code for them.

A.15 Voltage and frequency scaling model

I noticed that the simulator employs a linear model for both voltage and frequency changes. (3.3 V / 60 MHz; 4.4 V / 80 MHz). Is it reasonable?

The simulator does include a realistic voltage scaling model:

$$delay = (k \cdot Vdd)/(Vdd - Vt)^{\alpha}$$

The default values for k, α and Vt are set so that voltage scales linearly. You can change the values for Vt, k and α with the commands:

- `setscalevt` (C.129)
- `setscalek` (C.128)

- `setscalealpha` (C.127)

respectively. They each take a floating-point argument. The default values of the internal simulator parameters which these commands update are set to give linear scaling.

When scaling the operating voltage (Vdd), then the frequency calculation is easy. If scaling frequency, then calculating the appropriate Vdd given the delay, Vt and α is tricky, since Vdd, and delay are related in a non-algebraic manner. For the simulator implementation, the delay equation is solved for specific values of $\alpha = 0.5, 0.6, \ldots, 1.9, 2.0$, and the simulator only permits using those pre-determined values of α when scaling frequency.

A.16 Implementing real-time applications, dynamic voltage scaling (DVS) and low-power idling

Is there a easy way to write a real-time application (somehow to make the application to wait until a deadline, and during the wait to put the processor in an idle mode with low energy consumption)?

The simulator models among other things, a timer peripheral, which will generate timer interrupts every 1 ms unless you tell it not to. The software-defined radio application (`benchmarks/source/swradio/` installs an interrupt handler for timer interrupts, and uses timer interrupts and an interface to the real-time clock. The low-level assembly language initialization code (`init.S`), included with the example, includes the bottom part of the interrupt handler that saves registers and restores them on completion of the handler. It also contains a few utility routines, like `sleep()` , which issues a Hitachi SH sleep instruction. This puts the CPU in an idle mode (stops fetching instructions) until the next interrupt.

There are two ways you can implement application-controlled voltage scaling. The simple way is as follows: benchmarks running over the simulator can issue any of the commands you type in at the simulator console, through a "simulation control memory mapped register". See `benchmarks/source/swradio/swradio-sink/swradiosink.c` for an example of using this interface to turn the simulator off (see the line `devlog_ctl("off");` . You can therefore use this interface to perform a `setvdd` (C.133) or `setfreq` (C.116) or any other of the simulator commands.

To keep track of the passage of time in your application performing voltage-scaling, you might be interested in getting the current time with the routine `devrtc_getusecs()` which is implemented in `benchmarks/source/port/` . See `benchmarks/source/swradio/swradio-source/swradiosource.c` for an example of its use. The Makefile for `swradiosource.c` compiles and links in the necessary files to use this routine. There are other similar "device driver" routines in the `benchmarks/source/port` directory.

A.17 Porting new benchmarks to the simulator

I have legacy C/C++ applications that I would like to run over the simulator. How do I go about getting these to run?

For general purpose applications and computer architecture evaluation, it is easiest to get these benchmarks running on the Hitachi SH architecture model. The basic principles of the tasks you need to perform are (1) link in an assembly language stub to setup the stack and setup the processor before jumping to C code; (2) setup the Makefiles to appropriately link in the Newlib C library. Since the simulator's Hitachi SH model intercepts domain-crossing exceptions raised by the Newlib library due to system calls, it permits the emulation of a POSIX-like operating system, passing system calls performed by the simulated application down to the host, and managing the delivery of the return values (if any) of those calls, and so on; (3) employ an appropriate *linker script file* — the topic of linker script files as used, for example by GCC and the Gnu Linker is beyond the scope of this document, however much relevant information can be found on the web. If using the directory `benchmarks/source/swradio/swradio-source/` as a template, the linker script file is `superh.ld` (found in the same directory). For reasons that will not be elaborated here, using the above linker script file requires the entry point of the application to be a function with a name other then the traditional `main()`, and in the above case, the entry function is named `startup()`, and its this function that is jumped to by the assembly language initialization.

If it is desired to employ any of the interrupt sources in the ported application, the application must install interrupt handlers immediately after beginning execution. The functions `hdlr_install()` and `intr_hdlr()` can be copied from one of the provided example applications, e.g., `benchmarks/source/swradio/swradio-source/swra`
Lastly, the interrupt handlers need a memory location to which to save registers, essentially performing a context switch. This is achieved in the supplied examples with the appropriately sized global array `REGSAVESTACK` which is referenced by the low-level assembly language context switch routines in `init.S`.

A.18 Modeled costs of voltage and frequency scaling

How expensive in time (micro or milliseconds?) and energy is it to change the processor frequency / voltage (I need an order of magnitude, for let's say, the worst case).

If you change the operating voltage / frequency from the command line, it happens instantaneously — this is because what you are doing when you issue a `setvdd` ⌨ (C.133) or `setfreq` ⌨ (C.116) command from the simulator console, is you are changing a *static* machine configuration.

However, if you want to model microarchitecture-based voltage scaling, you will need to modify the simulator to add an appropriate penalty. One example of such a modification is what is done in `pau.c` 🗁 The files `pau.c` 🗁 and `pau.h` 🗁 are an outdated implementation[1] of a hardware-controlled dynamic voltage scaling scheme. The implementation is a bit outdated relative to the architecture of the rest of the simulator, but it might give you some ideas about how to perform dynamic scaling. Alternatively, when you use `devlog_ctl()` device interface to pass commands to the simulator from within your application, since this is getting executed *within* the simulation, there is some overhead there, of the order of 100's of cycles.

[1] Stanley-Marbell, Hsiao, and Kremer 2002.

A.19 Energy model

Could you give me a reference to the energy model used in the simulator (paper)? (I've only seen the paper about fast simulation).

There are essentially three models for power estimation in the simulator. An instruction-level energy model is based on actual measurements performed on a Hitachi SH3 SH7708 integrated circuit; that is described in the paper about fast simulation for the predecessor of Sunflower, which was named "Myrmigki".[2] The average power consumption of each instruction type in the ISA was characterized by measuring the current drawn by processor and memory system and incorporated that data into the simulator (`ilpa.h` 🗁). The second model isn't really a power model per se, but the simulator can report the amount of switching activity in the pipeline latches, register file, memory read/write ports and buses. You can use this to perform comparative dynamic power studies. The third model is relevant when you want to concentrate on active versus sleep mode power, estimates power consumption based on the state of the processor being in one of two states — active or idle. The estimates used for this last coarse-grained estimation mode are obtained from a data sheet. See the `forceavgpwr` ⌨ (C.41) command (Section C.41) for more information.

[2] Stanley-Marbell and Hsiao 2001.

A.20 The setquantum command

Just curious: what is "setquantum 1000000000" doing? I read the manual, but I didn't get what command quantum means.

The `setquantum` ⌨ (C.125) command is a command to speed up sim-

ulation when you are modeling multiple processors, or when modeling system components such as the network, batteries and external analog signals. What it does, in essence, is that rather than simulating each processor in a multiprocessor for one clock cycle round-robin to ensure fine-grained coherence of the passage of time, it simulates each processor for a large quantum of cycles, corresponding to the argument to the `setquantum` ⌨ (C.125) command. This can lead to significant simulation speedups, even in single-processor simulation, as it "tightens" the inner loop of the simulator. The tradeoff lies in the reduced time-coherence between modeled multiple processors, or between a single processor and the modeling of the environment, batteries, etc.

A.21 Getting the current program counter (PC), frequency and supply voltage

Which is the right command to obtain the current PC, processor frequency and supply voltage from the simulator?

The current program counter (PC) and operating voltage (Vdd) and processor clock frequency are displayed as part of the command line (recall, the simulation engine is running in the background). So just hit the "enter" key to see the updated PC, operating voltage and frequency.

A.22 Instruction latencies

Where can I find in the source code which instruction corresponds to each line from the `R0000` *array from* `ilpa.h` 🗁*. Does this array cover the entire ISA? I would like to use the number of cycles for each instruction for worst-case execution time (WCET) computation.*

The third column in `sim/utils/ilpa.orig.h` 🗁 is the number of clock cycles the instruction takes. Also see `sim/decode-hitachi-sh.c` 🗁 for the same instruction latency information.

The file `sim/ilpa.h` 🗁 (and hence `sim/utils/ilpa.orig.h` 🗁) cover the entire instruction set, except the `TRAPA` instruction: to perform the instruction-level power analysis, each instruction was put in a loop of 100 instruction, run and measured. Well, TRAPA is a software trap / software exception instruction, so you cannot do that.

A.23 Application using timer peripheral on Hitachi SH sleeps forever

I tried to use the timer. The application remains forever in the sleep. Do you know why?

You will also need to add the following in the simulator configuration file to enable interrupts (disabled by default):

sflr

```
clockintr 1
```

and, if you want to change the default time between timer interrupts:

sflr

```
settimerdelay <time in usecs>
```

To debug anything related to interrupts, use the `dumpsysregs` (C.33) command to inspect the contents of the `EXPEVT` and `INTEVT` registers. If their values are both `0x00000000`, then no interrupt or exception has occurred. Additionally, in the simcmddumpsysregs output, if `sleep = [yes]`, then the processor is currently idle after executing a sleep instruction. If the simulator is *not* running in FF mode, you should be able to use the `dumppipe` (C.31) command to see the contents of the pipeline; you should see a `SLEEP` instruction in the execution stage.

A.24 Non-interactive simulation

Is there any way to simulate in a non-interactive way: just giving the commands in a .m file, and collecting all the output (generated by the simulator and the program together) in a log file. I would like to have something like a "printf" instruction which tell me in which cycle the write action was executed.

One option is to supply the architecture specification file as a command line argument to the simulator. You will also need to put the `nodetach` (C.151) command at the top of the architecture configuration file, and the `quit` (C.95) at the end, to force the commands to be executed sequentially, quitting on completion; see Section C.151 for more information on `nodetach` (C.151) . The simulator automatically exists when all configured batteries are depleted. You could also use the `devlog_ctl()` interface to notify the simulator when the benchmark is ready to quit, even prior to its completion or depletion of batteries.

A.25 Calculating instructions per cycle (IPC)

How can I find the IPC (executed instruction per cycle)? Should I add a command to the simulator, or is it already there? Of course, I could compute it, first executing *`ni` (C.71), and then `showclk` (C.136), but I would prefer to obtain it directly.*

You can calculate IPC from the ratio of the counter `NINSTR` (number of instructions) to the counter `ICLK` (clock cycles). You can obtain `NINSTR` from the command line by command `ni` (C.71). Likewise `ICLK` by `showclk` (C.136). You can get both also from the simulator output file (`sunflower.out`), at the end of a simulation, or forced via the `dumpall` (C.29) (alias `d` (C.26)) command. You can cause the simulator to dump statistics to file "somefile" by either typing "d somefile" or, if you like, from your application running *over* the simulator, use `devlog_ctl()` to cause it to dump a new checkpoint of statistics.

A.26 Accessing the arguments supplied to the run command from applications

How can I access the arguments supplied to the `run` (C.106) command which appear to be passed as the application `argc` and `argv`, from the startup function?

The simulator sets up the necessary registers and stack space so that the C entry function such as `main(int argc, char *argv[])` or `startup(int argc, char *argv[])`, can access the arguments supplied to the simulator `run` (C.106) command via its `argc` and `argv` arguments.

A.27 Adapting the simulator's dynamic voltage scaling (DVS) latency modeling

Regarding DVS: I got the impression that the `pau.c` implements a hardware-driven DVS algorithm. Is it true? I am interested more in application-driven DVS. I think that the only thing that I have to do, in addition to the current simulator facilities, is to add a way to introduce the right cycle and energy penalty into simulator, when the frequency and voltage are changed.

Yes. That is trivial to do. In `pipeline-hitachi-sh.c`, either:

```
S->superH->P.fetch_stall_cycles += latency;
```

or

```
S->superH->P.EX.cycles += latency;
```

will stall either the fetch stage of the pipeline or the EX stage. both will have the same net effect, except that stalling EX will leave the pipeline full while you stall, while stalling fetch will introduce bubbles in the pipeline for the duration of stall.

A.28 *Adding new commands to simulator command language*

How do I add a new command?

To add a new command:

1. add a new production to the Yacc grammar in `sf.y`
2. add an new entry (and comment, the comment becomes the help line) to `lex.c`
3. implement the function to do what you want or put the code directly in `sf.y`

A.29 *Setting the different simulation modes —* `fast functional`*, cycle-accurate,* `bit-flip analysis` *and so on*

How do I configure the simulator for the different simulation modes, such as the fast functional mode, or the cycle-accurate pipeline simulation?

While the simulation of the motion of instructions through the pipeline can be enabled or disabled at the simulator command interface via the `pd` (C.85) command, enabling support for power estimation and signal transition counting must be performed when the simulator is compiled:

```
#define SF_BITFLIP_ANALYSIS     0      <-- enable/disable TC (0 = disable)
#define SF_POWER_ANALYSIS       1      <-- enable/disable ILPA (1 = enabled)
```

Coarse-grained power estimation, wherein the simulator only monitors the state of the processor (active, or in idle/sleep mode) is enabled by the `forceavgpwr` (C.41) command, detailed in Section C.41. Detailed motion of instructions through pipeline is enabled in the *cycle accurate (CA)* mode, activated by the `ca` (C.19) command (Section C.19). The fast execution mode (fetch and execute instruction without modeling their motion through pipeline), the *fast functional simulation (FF)* mode is enabled with the `ff` (C.38) command (Section C.38). Naturally, if you compiled-in support for signal transition activity counting, then you want to run in CA mode because otherwise you are missing all pipeline signal transition activity, etc.

A.30 Modeling custom hardware blocks

Does Sunflower let you add custom hardware modules — say, a hardware accelerator?

The general philosophy in the simulator implementation has been to try to structure things so that users of the framework can achieve all they want without modifying the simulator implementation, but rather just set up an appropriate simulator configuration. To implement models of custom hardware blocks, you could of course model your hardware in C, as an added peripheral to the processor, in much the same manner that Sunflower extends the Hitachi SH architecture with the network interface peripheral, for example (see `sim/network-hitachi-sh.c` and `sim/devsim7708.c` for the relevant implementations).

An easier method that enables you to cleanly decouple your hardware accelerator implementation from the simulation implementation however exists. You could instantiate a processor within the simulation to act as the hardware block, and configure that core's processor speed to give you the performance you would expect from a hardware block. For example, if you wanted to model a system with 2 general purpose cores, one hardware cryptographic engine, and, say, one hardware compression engine, you would:

- **instantiate two cores, and configure them to run at the intended clock speed for the general-purpose processors (say, 400 MHz).**
- **compile the software portion of your application to run on these 400 MHz cores**
- **instantiate another core each for the cryptographic hardware accelerator and the compression accelerator, and you would then need a software implementation of the cryptographic and compression algorithms to run over these cores. You would configure the cores to run at, say, 1GHz or whatever gives the execution of the "hardware cores" the performance you expect (e.g., number of ciphered bytes per second, say) compared to the 400 MHz cores. You can also then set the power consumed by these two instantiated processors to what you would expect from a hardware implementation, via the `forceavgpwr` (C.41) command.**

The nice thing about this approach is that timing issues, memory interfacing, etc., are all taken care of by the simulation engine. The main disadvantage is that the speed of simulation is reduced, as compared to, say, compiling the hardware blocks into the simulator.

A.31 Configuring on-chip communication topologies

Can the links behave like a simple on-chip communication bus? From the manual it seemed like these can be more closely modeled as wired or wireless channels?

Communication links are modeled as generalized channels which can be either single or multi-access. There are two main entities of interest: communication interfaces and communication channels. They are described in more detail in Chapter 6.

One or more communication interfaces are instantiated on each processor, and these are then separately connected to communication channels. By instantiating multiple interfaces per node, and multiple point-to-point or shared channels, you can create arbitrary topologies.

A channel is *single-access* **if its "width" is 1, and is** *multi-access* **if the width is > 1. A packet placed in the channel is addressed to either a specific node or is a broadcast. Once in transmission, for a single access channel, the channel is "busy" until the packet has been emptied into the recipients receive first-in-first-out (FIFO) queues, the time this takes being based on the size of the data and the configured network bit rate. Besides these communication facilities, applications may also communicate through shared memory. See the `mmap` ⌨ (C.54) command (Section C.54).**

A.32 Bus arbitration when modeling on-chip networks

Suppose I instantiate multiple cores, how is the bus access / arbitration etc. handled? Is it possible to simulate various communication architectures using Sunflower?

There is no pre-defined global arbiter; to implement an arbiter, one approach is to implement the arbitration algorithm as an application that runs over the simulator, and run that in an instantiated core, setting that core's clock frequency to give you the performance you desire from the arbiter hardware (via the `setfreq` ⌨ (C.116) command), and setting its power consumption to your estimated hardware arbiter power cost (via the `forceavgpwr` ⌨ (C.41) command). You would then instantiate channels from the processor cores to the arbiter, the equivalent of bus request lines, and have a separate channel to which all cores are connected, which is the actual bus. Each of these channels can be configured to give you the performance of a single wire, packet-based network, or multi-bit wide bus. All the above are things you would define in the simulator configuration file file, and you don't need to modify the simulator sources at all!

A.33 Functional, versus instruction-level, versus cycle-accuracy

Will the overall simulation be functionally-accurate, instruction-level-accurate or cycle-accurate?

The overall simulation will certainly be functionally accurate, as you will run code compiled with GCC for the target processor, same as you would do on real hardware. It is instruction-level accurate if your target system has the same ISA as the ISA modeled in the simulator. For a target general purpose RISC core, I'd say the instruction performance will be "close", though that is not a precise statement. You can however do some analysis to quantify how different instruction performance will be from your target ISA. On the other hand, if you are using the simulator to simulate a DSP, the instruction level performance might be markedly different, especially if you are simulating signal processing applications, as the simulated ISA does not have the multiply-accumulate instructions that are typical of DSPs, neither does the micro- and system-architecture have the circular buffers that make DSPs so great for signal processing kernels like filters. So, further then, one would not be able to expect cycle-level accuracy if the cores have different ISAs from that modeled in the simulator, etc., strictly speaking. But then strictly speaking, you would only have cycle-accuracy if you had the whole MPSoC modeled at the RTL, in, say, VHDL, Verilog, SystemVerilog, SystemC, BlueSpec, etc.

A.34 Application partitioning

Can I split one application to run over multiple cores, and also have multiple applications to run on one core?

You can split up a single application across multiple cores (you will have to do this manually of course, or if it is a Pthreads application, you can use the Pthreads library developed for the simulator, to enable initiation of new threads on new cores[3]). One example of a manually partitioned application is the swradio example supplied with the simulator (`benchmarks/source/swradio/` 🗁), which was originally one application (from the MIT Scale RAW benchmarks), and was partitioned to run in a pipeline with 12 cores.

[3] Stanley-Marbell, Lahiri, and Raghunathan 2006.

A.35 Multiple applications on one processor core

Are any OS functionalities available? Or would i have to write a Linux-like task scheduler?

You currently cannot easily run two applications on one core, as there is no officially supported OS. You could easily write your own simple scheduler. The sensor network benchmark applications in benchmarks/source/sbench include the starting points you need for operations like context switching, processor initialization and links to interrupt handlers.

B

Implementation Overview

This appendix provides brief descriptions for all the files in the main simulator implementation directory (`sim/`).

The simulator models each processing element with a structure, the `State` structure, defined in `sim/main.h` . All the components of the simulator that change machine state take as a parameter a pointer to an instance of a **`State`** structure, as well as a pointer to an instance of a *simulation engine*, which holds all global simulation state, in a `Engine` data structure. All the instantiated processors in the simulation are accessible through the global `Engine *E->sp` , which is an array of pointers to all the instantiated processors. This is utilized, for example, by routines that must perform some operation on all the processors. For example, the battery model must sum up the recorded current consumption for all modeled processors each cycle, and does this by scanning through `E->sp` for the current simulation engine. On the other hand, some routines only need access to the state of a single, specific processor. For example the cache access routines act on a single (specific) `State *` reference.

Each `State` structure contains pointers to functions which implement, e.g., actions to be performed each cycle (e.g., (`(State *)S)->step()` is called each clock cycle and exercises the pipeline, controlling instruction execution). These routines might in turn invoke other routines defined in the **`State`** structure. For example, on a given clock cycle, **`(State *)S)->step()`** will be invoked, and the instruction executed that cycle might cause a memory access, which might lead to a pipeline stall, for which `(State *)S)->stallaction()` will be invoked to perform any particular actions that are done on a cache miss (e.g., the implementation of the PAU structure[1] uses this).

[1] Stanley-Marbell, Hsiao, and Kremer 2002.

B.1 `LICENSE.txt`

The file `LICENSE.txt` contains the terms of distribution for the simulator.

B.2 sf.h

Almost all the source files include the header file `sim/sf.h` 🗁. Although it may be considered a bad idea (in some circles) to have header files which include other header files, there are several dependent structures defined in the various header files which would make it necessary for all C source files to include a large number of headers. Instead, they just include `sim/sf.h` 🗁 and any other specific needs.

B.3 arch-Inferno.c

The file `sim/arch-Inferno.c` 🗁 implements the host-platform dependent system calls for when the simulator is being used as part of the Inferno emulator (for the GUI).

B.4 arch-OpenBSD.c

The file `sim/arch-OpenBSD.c` 🗁 implements the host-platform dependent system calls for OpenBSD.

B.5 arch-darwin.c

The file `sim/arch-darwin.c` 🗁 implements the host-platform dependent system calls for Mac OS X.

B.6 arch-linux.c

The file `sim/arch-linux.c` 🗁 implements the host-platform dependent system calls for Linux.

B.7 arch-solaris.c

The file `sim/arch-solaris.c` 🗁 implements the host-platform dependent system calls for Solaris.

B.8 utils/batt-test.c

The file `sim/utils/batt-test.c` 🗁 is a small driver application that drives the battery model with a constant current profile. It can be used to generate nominal discharge characteristics for a given battery model. It calls routines implemented in **batt.c** (**newbatt()** to instantiate a new battery, **battery_feed()** to exercise the battery model update, and **battery_debug()** to generate its output). The parameter supplied to **battery_feed()** is the constant current that will be drawn from outside the battery system.

B.9 **batt.c**

The file `sim/batt.c` implements a discrete-time battery model based on.[2] Each simulation quantum, **`battery_feed()`** is called, and it sums up the current drawn from all the devices attached to each battery, and updates their modeled state. The granularity at which this battery update is performed is determined by, e.g., whether **`battery_feed()`** is called every clock cycle or not. This is determined in the simulators main event loop, in the function `schedule()`, in **`main.c`**.

[2] Benini et al. 2000.

B.10 **batt.h**

The file `sim/batt.h` defines the various structures and constants used by the battery model.

B.11 **battmodels/**

The directory `sim/battmodels/` contains the battery models provided with the simulator.

B.12 **big-endian-hitachi-sh.h, little-endian-hitachi-sh.h**

The simulator's instruction encoding and decoding uses C structure bit-fields on 2-byte structures. Although bit-fields are derided by certain bigots and purists, this technique does make the implementation easier, easier to correlate to the machine instruction layout specification, and faster. For simulations which often take days or a whole week (or more), even a mere 50% speedup is a big deal. The files `sim/little-endian-hitachi-sh.h` and `sim/big-endian-hitachi-sh.h` define different versions of the structures for Big-endian and Little-endian host machines, respectively.

B.13 **bit.h**

The file `sim/bit.h` defines constants that make dealing with binary masks easier.

B.14 **bit-utils.c**

The file `sim/bit-utils.c` implements routines for performing fast bit counting, as well as some bit display routines. Its actual implementation lies in **`sys/include/bit-utils.inc`**.

B.15 `cache-hitachi-sh.c`

The file `sim/cache-hitachi-sh.c` 🗁 implements a cache, whose size, block size and set-associativity is determined in the call to **`cacheinit()`**. The cache has a fixed write-back behavior, and block replacement is LRU. The implementation of the cache also does signal transition activity accounting (at the cache read and write ports) for use in the transition counting power analysis.

B.16 `cache-hitachi-sh.h`

The file `sim/cache-hitachi-sh.h` 🗁 defines the structures relating to the cache, such as the **`Cache`** structure, which in turn uses the `Block` structure.

B.17 `decode-hitachi-sh.c`

The file `sim/decode-hitachi-sh.c` 🗁 implements instruction decoding for the Hitachi SH ISA. The implementation uses symbolic names such as **`B0001`** or `B1111` to represent the binary values **`1`** and **`15`** respectively. This makes it easy to compare the constants appearing in different parts of the instruction encoding to the corresponding bit vectors defined in the manufacturer's data sheets. The constants are also used in various other places in the implementation. They are all defined in **`sys/include/bit.h`**.

B.18 `decode-hitachi-sh.h`

The file `sim/decode-hitachi-sh.h` 🗁 some definitions used by the instruction decode implementation for the Hitachi SH.

B.19 `decode-ti-msp430.h`

The file `sim/decode-ti-msp430.h` 🗁 contains instruction decode definitions for TI MSP430.

B.20 `dev7708.c`

The file `sim/dev7708.c` 🗁 implements all the memory-mapped registers for the Hitachi SH3 SH7708, along with other new memory-mapped registers, for, e.g., the modeled network interface, and permitting applications to access the simulator command set.

B.21 `dev7708.h`

The file `sim/dev7708.h` 🗁 contains relevant definitions for the implementation of the memory-mapped registers in Hitachi SH3 SH7708.

B.22 `dev430x1xxx.c`

The file `sim/dev430x1xxx.h` implements all the memory-mapped registers for the TI MSP430 F11X. Not distributed / empty in the distribution. This is in the process of being implemented.

B.23 `dev430x1xxx.h`

The file `sim/dev430x1xxx.h` contains relevant definitions for the implementation of the memory-mapped registers for the TI MSP430 F11X. Not distributed / empty in the distribution. This is in the process of being implemented.

B.24 `devsim7708.c`

The file `sim/devsim7708.c` implements extensions to the Hitachi SH architecture specific to the simulator.

B.25 `devsim7708.h`

The file `sim/devsim7708.c` defines relevant constants and data structures for the extensions to the Hitachi SH architecture specific to the simulator.

B.26 `devsunflower.c`

The file `sim/devsunflower.c` implements the device driver interface to the simulation engine. It is only compiled into the Inferno emulator.

B.27 `endian-hitachi-sh.h`

The file `sim/endian-hitachi-sh.h` includes the appropriate headers based on the host machine endianness defined by **`SF_X_ENDIAN`** in the **`Makefile`**, where 'X' is either 'L' for Little-endian host machines (e.g., all Linux/BSD on Intel x86 machines), or 'B' for Big-endian hosts (e.g., BSD/Linux/MacOS X on PowerPC, Solaris/BSD/Linux on SPARC).

B.28 `fault.c`

The file `sim/fault.c` implements the failure modeling for the processing devices and network segments. On each simulator cycle, based on granularity determined in the **`scheduler()`** loop in **`main.c`**, the function **`fault_feed()`** is called, which basically "kicks the dog", not that I—or any of the organizations with which I am affiliated—advocate the kicking of dogs.

B.29 `fault.h`

The file `sim/fault.h` 🗁 includes relevant definitions for the fault modeling in **`fault.c()`**.

B.30 `fdr.c`

The file `sim/fdr.c` 🗁 implements the "flight data recorder" — facilities for obtaining traces of register and memory contents associated with program source-level variables.

B.31 `fdr.h`

The file `sim/fdr.h` 🗁 defines relevant constants and data structures for the "flight data recorder" facilities.

B.32 `mfns.h`

The file `sim/mfns.h` 🗁 contains all the function prototype definitions for all the functions defined in the various parts of the simulator implementation. It is one of the things included from **`sf.h`**.

B.33 `instr-hitachi-sh.h`

The file `sim/instr-hitachi-sh.h` 🗁 contains instruction format definitions for the Hitachi SH3.

B.34 `interrupts-hitachi-sh.h`

The file `sim/interrupts-hitachi-sh.h` 🗁 defines relevant constants and data structures for the modeling of interrupts and exceptions on the Hitachi SH.

B.35 `interrupts-ti-msp430.h`

The file `sim/interrupts-hitachi-sh.h` 🗁 defines relevant constants and data structures for the modeling of interrupts and exceptions on the TI MSP430.

B.36 `lex.c`

The file `sim/lex.c` 🗁 is part of the lexical analyzer implementation, used by the simulator's built in assembler (which should accept any assembler generated by GCC for the Hitachi SH3), as well as the interactive simulator specific commands. The `TokenTab token_table[]` array defines the various commands accepted at the simulators command interface. The comments associated with each of the array entries are in a special format, and begin with **"/*+"**. The comments are parsed by the script **mkhelp** to generate the file **help.c** and also, by the script `mkmantex`, to generate LaTeX source for inclusion in, e.g., the appendix of this manual. New commands added with comments in this format immediately become visible in the online help and documentation, after recompilation. The comments must have the form **"/*+ description : parameters for command */"**. The description string must not contain any newlines or any of the characters '*', '_', '}', '{', '+', ',', ':' or '"'. The description string may end in a period (**"."**), and should be followed by a colon (**":"**), and the arguments taken by the command which the description describes.

B.37 `machine-hitachi-sh.c`, `machine-hitachi-sh.h`

The files `sim/machine-hitachi-sh.c` and `sim/machine-hitachi-sh.h` contains all the parts of the simulator implementation which are not part of instruction decode/execution, but which are specific to the Hitachi SH architecture and ISA. It is mostly the Hitachi SH specific versions of functions for which there are function pointers in the **State** structure.

B.38 `machine-ti-msp430.c`, `machine-ti-msp430.h`

The files `sim/machine-ti-msp430.c` and `sim/machine-ti-msp430.h` contain relevant machine-specific definitions for the TI MSP430.

B.39 `main.c`

The file `sim/main.c` is the main "glue" for the simulator. It contains the definitions of all global structures (such as the `SIM_STATE_PTRS[]` array mentioned previously), and the simulator's main event loop. The simulator operates as 2 threads. The command interface event loop is one thread, and the simulation engine is a separate thread. This is done with POSIX threads or *pthreads*, but it might just as easily be done with some variant of *fork()* such as the Plan 9 *rfork()*.

The main simulation event loop is implemented in the function **scheduler()** defined in **main.c**. It's sole function is to increment the global simulation clock, **SIM_GLOBAL_CLOCK**, and call all of the routines which need to be exercised each clock cycle, once. Thus, it calls **network_clock()** which makes the network simulation code "do its thing" for that clock cycle, calls the routine **fault_feed()** which makes the fault modeling implementation "do its thing", and most importantly, calls the **step()** routine of each modeled processor, which exercises the instruction pipeline for one clock tick. This main loop also checks to see if any modeled processor should be delivered an interrupt, for various reasons.

The **main.c** file also contains various helper routines, such as routines for decoding binaries and loading them into memory.

B.40 main.h

sim/main.h contains the definitions for many constants and structures used throughout the simulator that are not specific to any one structure. Most importantly, it contains the definition of of the **State** structure, which contains all the state for a modeled processor, and pointers to routines to be called, for example, to exercise its pipeline each clock cycle.

B.41 mkhelp

The **sim/mkhelp** script parses the file **sim/lex.c** (as hinted at previously) to generate a C array definition, which goes into **help.h**. This array is indexed to provide the online help.

B.42 mkmantex

The script **sim/mkmantex** parses the file **lex.c** (as hinted at previously) to generate LaTeX source for inclusion in documentation such as this manual.

B.43 mkopstr-hitachi-sh

The script **sim/mkopstr-hitachi-sh** parses the file **decode-hitachi-sh.h** to generate the file **opstr-hitachi-sh.h** which is used to provide decoded instruction information, for example, when displaying the contents of the instruction pipeline via the **DUMPPIPE** command.

B.44 mkopstr-ti-msp430

The script **sim/mkopstr-ti-msp430** parses the file **decode-ti-msp430.h** to generate the file **opstr-ti-msp430.h** which is used to provide decoded instruction information, for example, when displaying the contents of the instruction pipeline via the **DUMPPIPE** command.

B.45 network-hitachi-sh.c

The file **sim/network-hitachi-sh.c** contains the implementation of the network modeling. Most importantly, it contains the function **network_clock()**

which is called each simulation cycle from the function `scheduler()`, to exercise the network modeling, such as moving the right amount of bits from a network into a processor's network interface receive buffer, for the amount of time elapsed during a clock cycle, and appropriately related to the simulated network speed.

B.46 network-hitachi-sh.h

The file `sim/network-hitachi-sh.h` contains all the necessary structure and constant definitions for the network modeling. It contains definitions for the `Ifc`, **`Segbuf`** and **`Netsegment`** structures. These define the simulated network interface, network segment storage (i.e., when the bits are "on the wire", they are stored in a **`Segbuf`**) and network segment, respectively.

B.47 op-hitachi-sh.c

`sim/op-hitachi-sh.c` implements the hard work of instruction execution for the Hitachi SH architecture.

B.48 op-hitachi-sh.h

The file `sim/op-hitachi-sh.h` provides all the definitions specific to **`op-hitachi-sh.c`** are here.

B.49 op-ti-msp430.c

`sim/op-ti-msp430.c` implements the hard work of instruction execution for the TI MSP430 architecture.

B.50 op-ti-msp430.h

The file `sim/op-hitachi-sh.h` provides all the definitions specific to **`op-ti-msp430.c`** are here.

B.51 pipeline-hitachi-sh.c

The file `sim/pipeline-hitachi-sh.c` implements the Hitachi SH pipeline.

B.52 pipeline-hitachi-sh.h

The file `sim/pipeline-hitachi-sh.h` defines relevant constants and data structures for the Hitachi SH pipeline.

B.53 `pipeline-ti-msp430.c`

The file `sim/pipeline-ti-msp430.c` implements the TI MSP430 pipeline.

B.54 `pipeline-ti-msp430.h`

The file `sim/pipeline-ti-msp430.h` defines relevant constants and data structures for the Hitachi SH pipeline.

B.55 `power.c`

The file `sim/power.c` functions relating to power estimation and frequency / voltage scaling.

B.56 `randgen.c`

The file `sim/randgen.c` implements the random number generation.

B.57 `randgen.h`

The file `sim/randgen.h` defines relevant constants and data structures for the random number generation.

B.58 `regaccess-hitachi-sh.c`

The file `sim/regaccess-hitachi-sh.c` implements the Hitachi SH register access functions.

B.59 `regaccess-ti-msp430.c`

The file `sim/regaccess-ti-msp430.c` implements the TI MSP430 register access functions.

B.60 `pau.c`

The file `sim/pau.c` implements the Power Adaptation Unit (PAU),[3] which exploits the mismatch between CPU and memory system performance to reduce energy dissipation, via dynamic voltage scaling.

[3] Stanley-Marbell, Hsiao, and Kremer 2002.

B.61 `pau.h`

The file `sim/pau.h` defines definitions needed by **`pau.c`** are here.

B.62 `pic.c`

The file `sim/pic.c` 🗁 implements queued interrupts. The idea is that it is a form of a programmable interrupt controller.

B.63 `pic.h`

The file `sim/pic.h` 🗁 provides definitions needed by **`pic.c`** are here.

B.64 `pipeline-hitachi-sh.c`

The file `sim/pipeline-hitachi-sh.c` 🗁 implements the modeling of the Hitachi SH's pipeline. It defines the routine **`step()`** which is called for each modeled processor during each simulation step, to move instruction one more step along in their execution.

B.65 `pipeline-hitachi-sh.h`

The file `sim/pipeline-hitachi-sh.h` 🗁 defines the structures and constants used by **`pipeline-hitachi-sh.c`**, such as the **`Pipe`** and **`Pipestage`** structures.

B.66 `power.h`

The file `sim/power.h` 🗁 defines all the structures supporting the simulators power modeling.

B.67 `regs-hitachi-sh.h`

The file `sim/regs-hitachi-sh.h` 🗁 provides various definitions pertinent to the modeling of machine registers on the Hitachi SH.

B.68 `regs-ti-msp430.h`

The file `sim/regs-ti-msp430.h` 🗁 provides various definitions pertinent to the modeling of machine registers on the TI MSP430.

B.69 `sf.y`

The file `sim/sf.y` 🗁 is the YACC grammar for the command interface and assembler parser. The command interface parser is defined by `shasm.y` and **`lex.c`**. The file **`lex.c`** contains a hand-written lexer (included from **`sys/include/lex.inc`**). You might find it useful to look in lex.c if you are curious about the commands accepted by the simulator.

B.70 `syscalls.c`, `syscalls.h`, `syscalls-Inferno.c`

The files `sim/syscalls.c` 🗁, `sim/syscalls.h` 🗁 and `sim/syscalls-Inferno.c` 🗁 implement functions and provide definitions for the system call trap values. They implement, e.g., the handling of system calls by the simulator. This is where, e.g., TRAPA #34 instruction passes system calls to the host operating system etc.

B.71 `tag.c`

The file `sim/tag.c` 🗁 implements simulator implements per-processor "tag memory", *a la* Smart Messages.[4]

[4] Stanley-Marbell et al. 2000.

B.72 `tokenhandling.c`

The file `sim/tokenhandling.c` 🗁 functions relating to parsing of input.

C
Sunflower Commands

C.1 ADDVALUETRACE

Description: Install an address monitor to track data values.

Synopsis:

```
ADDVALUETRACE   <name string> <base addr> <size> <onstack> <pcstart> <
  frameoffset>
```

C.2 BATTALERTFRAC

Description: Set battery alert level fraction.

Synopsis:

```
BATTALERTFRAC
```

C.3 BATTCF

Description: Set Battery Vrate lowpass filter capacitance.

Synopsis:

```
BATTCF   <Capacitance in Farads>
```

C.4 BATTETALUT

Description: Set Battery etaLUT value.

Synopsis:

```
BATTETALUT   <LUT index> <value>
```

C.5 BATTETALUTNENTRIES

Description: Set number of etaLUT entries.

Synopsis:

```
BATTETALUTNENTRIES   <number of entries>
```

C.6 BATTILEAK

Description: Set Battery self-discharge current.

Synopsis:
```
BATTILEAK   <Current in Amperes>
```

C.7 BATTINOMINAL

Description: Set Battery Inominal.

Synopsis:
```
BATTINOMINAL   <Inominal in Amperes>
```

C.8 BATTNODEATTACH

Description: Attach current node to a specified battery.

Synopsis:
```
BATTNODEATTACH   <which battery>
```

C.9 BATTRF

Description: Set Battery Vrate lowpass filter resistance.

Synopsis:
```
BATTRF   <Resistance in Ohms>
```

C.10 BATTSTATS

Description: Get battery statistics.

Synopsis:
```
BATTSTATS   <which battery>
```

C.11 BATTVBATTLUT

Description: Set Battery VbattLUT value.

Synopsis:
```
BATTVBATTLUT   <index> <value>
```

C.12 BATTVBATTLUTNENTRIES

Description: Set number of VbattLUT entries.

Synopsis:
```
BATTVBATTLUTNENTRIES   <number of entries>
```

C.13 BATTVLOSTLUT

Description: Set Battery VlostLUT value.

Synopsis:
```
BATTVLOSTLUT   <index> <value>
```

C.14 BATTVLOSTLUTNENTRIES

Description: Set number of VlostLUT entries.

Synopsis:
```
BATTVLOSTLUTNENTRIES   <number of entries>
```

C.15 BPT

Description: Set breakpoint.

Synopsis:
```
BPT   <CYCLES> <ncycles on current node> | <INSTRS> <ninstrs on current node>| <
  SENSORREADING> <which sensor> <float value> | <GLOBALTIME> <global time in
  picoseconds>
```

C.16 BPTDEL

Description: Delete breakpoint.

Synopsis:
```
BPTDEL   <breakpoint ID>
```

C.17 BPTLS

Description: List breakpoints and their IDs.

Synopsis:
```
BPTLS   <>
```

C.18 C

Description: Synonym for CACHESTATS.

Synopsis:
```
C
```

C.19 CA

Description: Set simulator in cycle-accurate mode.

Synopsis:
```
CA
```

C.20 CACHEINIT

Description: Initialise cache.

Synopsis:
```
CACHEINIT   <cache size> <block size> <set associativity>
```

C.21 CACHEOFF

Description: Deactivate cache.

Synopsis:
```
CACHEOFF
```

C.22 CACHESTATS

Description: Retrieve cache access statistics.

Synopsis:
```
CACHESTATS
```

C.23 CD

Description: Change current working directory.

Synopsis:
```
CD   <path>
```

C.24 CLOCKINTR

Description: Toggle enabling clock interrupts.

Synopsis:
```
CLOCKINTR   <0/1>
```

C.25 CONT

Description: Continue execution while PC is not equal to specified PC.

Synopsis:
```
CONT   <until PC>
```

C.26 D

Description: Synonym for DUMPALL.

Synopsis:
```
D   <filename> <tag> <prefix>
```

C.27 DEFNDIST

Description: Define a discrete probability measure as a set of badis value probability tuples.

Synopsis:
```
DEFNDIST   <list of basis value> <list of probabilities>
```

C.28 DELVALUETRACE

Description: Delete an installed address monitor for tracking data values.

Synopsis:
```
DELVALUETRACE   <name string> <base addr> <size> <onstack> <pcstart><frameoffset
  >
```

C.29 DUMPALL

Description: Dump the State structure info for all nodes to the file using given tag and prefix.

Synopsis:
```
DUMPALL   <filename> <tag> <prefix>
```

C.30 DUMPMEM

Description: Show contents of memory.

Synopsis:
```
DUMPMEM  <start mem address> <end mem address>
```

C.31 DUMPPIPE

Description: Show the contents of the pipeline stages.

Synopsis:
```
DUMPPIPE
```

C.32 DUMPREGS

Description: Show the contents of the general purpose registers.

Synopsis:
```
DUMPREGS
```

C.33 DUMPSYSREGS

Description: Show the contents of the system registers.

Synopsis:
```
DUMPSYSREGS
```

C.34 DUMPTLB

Description: Display all TLB entries.

Synopsis:
```
DUMPTLB
```

C.35 DYNINSTR

Description: Display number of instructions executed.

Synopsis:
```
DYNINSTR
```

C.36 EBATTINTR

Description: Toggle enable low battery level interrupts.

Synopsis:
```
EBATTINTR  <0/1>
```

C.37 EFAULTS

Description: Enable interuppt when too many faults occur.

Synopsis:
```
EFAULTS
```

C.38 FF

Description: Set simulator in fast functional mode.

Synopsis:
```
FF
```

C.39 FILE2NETSEG

Description: Connect file to netseg.

Synopsis:
```
FILE2NETSEG   <file> <netseg>
```

C.40 FLTTHRESH

Description: Set threashold for EFAULTS.

Synopsis:
```
FLTTHRESH   <threshold>
```

C.41 FORCEAVGPWR

Description: Bypass ILPA analysis and set avg pwr consumption.

Synopsis:
```
FORCEAVGPWR   <avg pwr in Watts> <sleep pwr in Watts>
```

C.42 GETRANDOMSEED

Description: Query seed used to initialize random number generation system useful for reinitializing generator to same seed for reproducibility.

Synopsis:
```
GETRANDOMSEED
```

C.43 HELP

Description: Print list of commands.

Synopsis:
```
HELP
```

C.44 HWSEEREG

Description: Register a hardware structure or part thereof for inducement of SEEs.

Synopsis:
```
HWSEEREG   <structure name> <actual bits> <logical bits> <bit offset>
```

C.45 IGN

Description: Ignore node fatalities and continue sim without pausing.

Synopsis:
```
IGN   <0 or 1>
```

C.46 INITRANDTABLE

Description: Set or change node location.

Synopsis:
```
INITRANDTABLE  <distname> <pfun name> <basis min> <basis max> <granularity> <p1
  > <p2> <p3> <p4>
```

C.47 INITSEESTATE

Description: Initialize SEE function and parameter state.

Synopsis:
```
INITSEESTATE   <loc pfun> <loc p1> <loc p2> <loc p3> <loc p4> <bit pfun> <bit
  p1> <bit p2> <bit p3> <bit p4> <duration pfun> <dur p1> <dur p2> <dur p3> <dur
   p4>
```

C.48 L

Description: Synonym for LOCSTATS.

Synopsis:
```
L
```

C.49 LISTRVARS

Description: List all structures that can be treated as rvars.

Synopsis:
```
LISTRVARS
```

C.50 LOAD

Description: Load a script file.

Synopsis:
```
LOAD  <filename>
```

C.51 LOCSTATS

Description: Show node's current location in three-dimentional space.

Synopsis:
```
LOCSTATS
```

C.52 MALLOCDEBUG

Description: Display malloc stats.

Synopsis:
```
MALLOCDEBUG
```

C.53 MAN

Description: Print synopsis for command usage.

Synopsis:
```
MAN  <command name>
```

C.54 MMAP

Description: Map memory of one simulated node into another.

Synopsis:
```
MMAP   <source> <destination>
```

C.55 N

Description: Step through simulation for a number (default 1) of cycles.

Synopsis:
```
N   [# cycles]
```

C.56 NANOPAUSE

Description: Pause the simulation for arg nanoseconds.

Synopsis:
```
NANOPAUSE   <duration of pause in nanoseconds>
```

C.57 ND

Description: Synonym for NETDEBUG.

Synopsis:
```
ND
```

C.58 NETCORREL

Description: Specify correlation coefficient between failure of a network segment and failure of an IFC on a node **NOTE that it is not using the current node so we can specify in a matrix-like form**.

Synopsis:
```
NETCORREL   <which seg> <which node>   <coefficient>
```

C.59 NETDEBUG

Description: Show debugging information about the simulated network interface.

Synopsis:
```
NETDEBUG
```

C.60 NETNEWSEG

Description: Add a new network segment to simulation.

Synopsis:
```
NETNEWSEG   <which (if exists)> <frame bits> <propagation speed> <bitrate> <
  medium width> <link failure probability distribution> <link failure
  distribution mu> <link failure probability distribution sigma> <link failure
  probability distribution lambda> <link failure duration distribution> <link
  failure duration distribution mu> <link failure duration distribution sigma> <
  link failure duration distribution lambda>
```

C.61 NETNODENEWIFC

Description: Add a new IFC to current node frame bits and segno are set at attach time.

Synopsis:
```
NETNODENEWIFC   <ifc num (if valid)> <tx pwr (watts)> <rx pwr (watts)> <idle pwr
    (watts)> <listen pwr (watts)> <fail distribution> <fail mu> <fail sigma> <
   fail lambda> <transmit FIFO size> <receive FIFO size>
```

C.62 NETSEG2FILE

Description: Connect netseg to file.

Synopsis:
```
NETSEG2FILE   <netseg> <file>
```

C.63 NETSEGDELETE

Description: Disable a specified network segment.

Synopsis:
```
NETSEGDELETE   <which segment>
```

C.64 NETSEGFAILDURMAX

Description: Set maximum network segment failure duration in clock cycles though actual failure duration is determined by probability distribution.

Synopsis:
```
NETSEGFAILDURMAX   <duration>
```

C.65 NETSEGFAILPROB

Description: Set probability of failure for a setseg.

Synopsis:
```
NETSEGFAILPROB   <which segment> <probability>
```

C.66 NETSEGFAILPROBFN

Description: Specify Netseg failure Probability Distribution Function (fxn of time).

Synopsis:
```
NETSEGFAILPROBFN   <expression in terms of constants and 'pow(a
```

C.67 NETSEGNICATTACH

Description: Attach a current node's IFC to a network segment.

Synopsis:
```
NETSEGNICATTACH   <which IFC> <which segment>
```

C.68 NETSEGPROPMODEL

Description: Associate a network segment with a signal propagation model.

Synopsis:

```
NETSEGPROPMODEL   <netseg ID> <sigsrc ID> <minimum SNR>
```

C.69 NEWBATT

Description: New battery

Synopsis:

```
NEWBATT   <ID> <capacity in mAh>
```

C.70 NEWNODE

Description: Create a new node (simulated system).

Synopsis:

```
NEWNODE   <type=superH|msp430> [<x location> <y location> <z location>] [<
  trajectory file name> <loopsamples> <picoseconds per trajectory sample>]
```

C.71 NI

Description: Synonym for DYNINSTR.

Synopsis:

```
NI
```

C.72 NODEFAILDURMAX

Description: Set maximum node failure duration in clock cycles though actual failure duration is determined by probability distribution.

Synopsis:

```
NODEFAILDURMAX    <duration>
```

C.73 NODEFAILPROB

Description: Set probability of failure for current node.

Synopsis:

```
NODEFAILPROB   <probability>
```

C.74 NODEFAILPROBFN

Description: Specify Node failure Probability Distribution Function (fxn of time).

Synopsis:

```
NODEFAILPROBFN   <expression in terms of constants and ’pow(a
```

C.75 NUMAREGION

Description: Specify a memory access latency and a node mapping (can only map into destination RAM) for an address range for a private mapping.

Synopsis:

```
NUMAREGION  <name string> <start address (inclusive)> <end address (non-
  inclusive)> <local read latency in cycles> <local write latency in cycles> <
  remote read latency in cycles> <remote write latency in cycles> <Map ID> <Map
  offset> <private flag>
```

C.76 NUMASETMAPID

Description: Change the mapid for nth map table entry on all nodes to i.

Synopsis:

```
NUMASETMAPID  <n> <i>
```

C.77 NUMASTATS

Description: Display access statistics for all NUMA regions for current node.

Synopsis:

```
NUMASTATS
```

C.78 NUMASTATSALL

Description: Display access statistics for all NUMA regions for all nodes.

Synopsis:

```
NUMASTATSALL
```

C.79 OFF

Description: Turn the simulator off.

Synopsis:

```
OFF
```

C.80 ON

Description: Turn the simulator on.

Synopsis:

```
ON
```

C.81 PARSEOBJDUMP

Description: Parse a GNU objdump file and load into memory.

Synopsis:

```
PARSEOBJDUMP  <objdump file path>
```

C.82 PAUINFO

Description: Show information about all valid PAU entries.

Synopsis:
```
PAUINFO
```

C.83 PAUSE

Description: Pause the simulation for arg seconds.

Synopsis:
```
PAUSE   <duration of pause in seconds>
```

C.84 PCBT

Description: Dump PC backtrace.

Synopsis:
```
PCBT
```

C.85 PD

Description: Disable simulation of processor's pipeline.

Synopsis:
```
PD
```

C.86 PE

Description: Enable simulation of processor's pipeline.

Synopsis:
```
PE
```

C.87 PF

Description: Flush the pipeline.

Synopsis:
```
PF
```

C.88 PFUN

Description: Change probability distrib fxn (default is uniform).

Synopsis:
```
PFUN
```

C.89 PI

Description: Synonym for PAUINFO.

Synopsis:
```
PI
```

C.90 POWERSTATS

Description: Show estimated energy and circuit activity.

Synopsis:
```
POWERSTATS
```

C.91 POWERTOTAL

Description: Print total power accross all node.

Synopsis:
```
POWERTOTAL
```

C.92 PS

Description: Synonym for POWERSTATS.

Synopsis:
```
PS
```

C.93 PWD

Description: Get current working directory.

Synopsis:
```
PWD
```

C.94 Q

Description: Synonym for QUIT.

Synopsis:
```
Q
```

C.95 QUIT

Description: Exit the simulator.

Synopsis:
```
QUIT
```

C.96 R

Description: Synonym for RATIO.

Synopsis:
```
R  <>
```

C.97 RANDPRINT

Description: Print a random value from the selected distribution with given parameters.

Synopsis:
```
RANDPRINT  <distribution name> <min> <max> <p1> <p2> <p3> <p4>
```

C.98 RATIO

Description: Print ratio of cycles spent active to those spent sleeping.

Synopsis:
```
RATIO
```

C.99 REGISTERRVAR

Description: Register a simulator internal implementation variable or structure for periodic updates either overwriting values or summing determined by the mode parameter.

Synopsis:
```
REGISTERRVAR    <sim var name> <index for array structures> <value dist name> <
  value dist p1> <value dist p2> <value dist p3> <value dist p4> <duration dist
  name> <duration dist p1> <duration dist p2> <duration dist p3> <duration dist
  p4> <mode>
```

C.100 REGISTERSTABS

Description: Register variables in a STABS file with value tracing framework.

Synopsis:
```
REGISTERSTABS   <STABS filename>
```

C.101 RENUMBERNODES

Description: Renumber nodes based on base node ID.

Synopsis:
```
RENUMBERNODES
```

C.102 RESETALLCTRS

Description: Reset simulation rate measurement trip counters for all nodes.

Synopsis:
```
RESETALLCTRS
```

C.103 RESETCPU

Description: Reset entire simulated CPU state.

Synopsis:
```
RESETCPU
```

C.104 RESETNODECTRS

Description: Reset simulation rate measurement trip counters for current node only.

Synopsis:
```
RESETNODECTRS
```

C.105 RETRYALG

Description: set NIC retransmission backoff algorithm.

Synopsis:
```
RETRYALG  <ifc #> <algname>
```

C.106 RUN

Description: Mark a node as runnable.

Synopsis:
```
RUN
```

C.107 SAVE

Description: Dump memory region to disk.

Synopsis:
```
SAVE  <start mem addr> <end mem addr> <filename>
```

C.108 SENSORSDEBUG

Description: Display various statistics on sensors and signals.

Synopsis:
```
SENSORSDEBUG
```

C.109 SETBASENODEID

Description: Set ID of first node from which all node IDs will be offset.

Synopsis:
```
SETBASENODEID  <integer>
```

C.110 SETBATT

Description: Set current battery.

Synopsis:
```
SETBATT  <Battery ID>
```

C.111 SETBATTFEEDPERIOD

Description: Set update periodicity for battery simulation.

Synopsis:
```
SETBATTFEEDPERIOD  <period in picoseconds>
```

C.112 SETDUMPPWRPERIOD

Description: Set periodicity power logging to simlog.

Synopsis:
```
SETDUMPPWRPERIOD  <period in picoseconds>
```

C.113 SETFAULTPERIOD

Description: Set period for activating fault scheduling.

Synopsis:
```
SETFAULTPERIOD   <period in picoseconds>
```

C.114 SETFLASHRLATENCY

Description: Set flash read latency.

Synopsis:
```
SETFLASHRLATENCY    <latency in clock cycles>
```

C.115 SETFLASHWLATENCY

Description: Set flash write latency.

Synopsis:
```
SETFLASHWLATENCY    <latency in clock cycles>
```

C.116 SETFREQ

Description: Set operating frequency from voltage.

Synopsis:
```
SETFREQ   <freq/MHz> (double)
```

C.117 SETIFCOUI

Description: Set OUI for current IFC.

Synopsis:
```
SETIFCOUI   <which IFC> <new OUI>
```

C.118 SETLOC

Description: Set or change node location.

Synopsis:
```
SETLOC   <xloc> <yloc> <zloc>
```

C.119 SETMEMRLATENCY

Description: Set memory read latency.

Synopsis:
```
SETMEMRLATENCY    <latency in clock cycles>
```

C.120 SETMEMWLATENCY

Description: Set memory write latency.

Synopsis:
```
SETMEMWLATENCY    <latency in clock cycles>
```

C.121 SETNETPERIOD

Description: Set period for activting network scheduling.

Synopsis:
```
SETNETPERIOD   <period in picoseconds>
```

C.122 SETNODE

Description: Set the current simulated node.

Synopsis:
```
SETNODE   <node id>
```

C.123 SETPC

Description: Set the value of the program counter.

Synopsis:
```
SETPC   <PC value>
```

C.124 SETPHYSICSPERIOD

Description: Set update periodicity for physical phenomenon simulation.

Synopsis:
```
SETPHYSICSPERIOD   <period in picoseconds>
```

C.125 SETQUANTUM

Description: Set simulation instruction group quantum.

Synopsis:
```
SETQUANTUM   <integer>
```

C.126 SETRANDOMSEED

Description: Reinitialize random number generation system with a specific seed useful in conjunction with GETRANDOMSEED for reproducing same pseudorandom state.

Synopsis:
```
SETRANDOMSEED   <seed value negative one to use current time>
```

C.127 SETSCALEALPHA

Description: Set technology alpha parameter for use in voltage scaling.

Synopsis:
```
SETSCALEALPHA   <double>
```

C.128 SETSCALEK

Description: Set technology K parameter for use in voltage scaling.

Synopsis:
```
SETSCALEK   <double>
```

C.129 SETSCALEVT

Description: Set technology Vt for use in voltage scaling.

Synopsis:
```
SETSCALEVT   <double>
```

C.130 SETSCHEDRANDOM

Description: Use a different random order for node simulation every cycle.

Synopsis:
```
SETSCHEDRANDOM   <>
```

C.131 SETSCHEDROUNDROBIN

Description: Use a round-robin order for node simulation.

Synopsis:
```
SETSCHEDROUNDROBIN   <>
```

C.132 SETTIMERDELAY

Description: Change granularity of timer intrs.

Synopsis:
```
SETTIMERDELAY   <granularity in microseconds>
```

C.133 SETVDD

Description: Set operating voltage from frequency.

Synopsis:
```
SETVDD   <Vdd/volts> (double)
```

C.134 SFATAL

Description: Induce a node death and state dump.

Synopsis:
```
SFATAL   <suicide note>
```

C.135 SHAREBUS

Description: Share bus structure with ther named node.

Synopsis:
```
SHAREBUS   <Bus donor nodeid>
```

C.136 SHOWCLK

Description: Show the number of clock cycles simulated since processor reset.

Synopsis:
```
SHOWCLK
```

C.137 SHOWPIPE

Description: Show contents of the processor pipeline.

Synopsis:
```
SHOWPIPE
```

C.138 SIGSRC

Description: Create a physical phenomenon signal source.

Synopsis:
```
SIGSRC  <type> <description> <tau> <propagationspeed> <A> <B> <C> <D> <E> <F> <
  G> <H> <I> <K> <m> <n> <o> <p> <q> <r> <s> <t> <x> <y> <z> <trajectoryfile> <
  trajectoryrate> <looptrajectory> <samplesfile> <samplerate> <fixedsampleval> <
  loopsamples>
```

C.139 SIGSUBSCRIBE

Description: Subscribe sensor X on the current node to a signal source Y.

Synopsis:
```
SIGSUBSCRIBE  <X> <Y>
```

C.140 SIZEMEM

Description: Set the size of memory.

Synopsis:
```
SIZEMEM  <size of memory in bytes>
```

C.141 SPLIT

Description: Split current CPU to execute from a new PC and stack.

Synopsis:
```
SPLIT  <newpc> <newstackaddr> <argaddr> <newcpuidstr>
```

C.142 SRECL

Description: Load a binary program in Motorola S-Record format.

Synopsis:
```
SRECL
```

C.143 STOP

Description: Mark the current node as unrunnable.

Synopsis:
```
STOP
```

C.144 THROTTLE

Description: Set the throttling delay in nanoseconds.

Synopsis:
```
THROTTLE  <throttle delay in nanoseconds>
```

C.145 THROTTLEWIN

Description: Set the throttling window — main simulation loop sleeps for throttlensecs x throttlewin nanosecs every throttlewin simulation cycles

Synopsis:
```
THROTTLEWIN    for an average of throttlensecs sleep per simulation cycle.
```

C.146 TRACE

Description: Toggle Tracing.

Synopsis:
```
TRACE
```

C.147 V

Description: Synonym for VERBOSE.

Synopsis:
```
V
```

C.148 VALUESTATS

Description: Print data value tracking statistics.

Synopsis:
```
VALUESTATS
```

C.149 VERBOSE

Description: Enable the various prints.

Synopsis:
```
VERBOSE
```

C.150 VERSION

Description: Display the simulator version and build.

Synopsis:
```
VERSION
```

C.151 NODETACH

Description: Set whether new thread should be spawned on a ON command.

Synopsis:
```
NODETACH   <0 or 1>
```

C.152 SIZEPAU

Description: Set the size of the PAU.

Synopsis:
```
SIZEPAU   <size of PAU in number of entries>
```

Bibliography

1. Benini, L. et al. (2000). "A discrete-time battery model for high-level power estimation". In: *Proceedings of the conference on Design, automation and test in Europe*, pp. 35–39 (cit. on pp. 47, 113).

2. "Cygnus GnuPro Documentation" (cit. on p. 66).

3. Nishimura, Takuji (2000). "Tables of 64-bit Mersenne twisters". In: *ACM Trans. Model. Comput. Simul.* 10.4, pp. 348–357. ISSN: 1049-3301. DOI: http://doi.acm.org/10.1145/369534.369540 (cit. on p. 77).

4. Ross, Sheldon M. (2001). *Simulation*. San Diego, CA: Academic Press (cit. on p. 77).

5. Stanley-Marbell, P. and M. Hsiao (2001). "Fast, Flexible, Cycle-Accurxate Energy Estimation". In: *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 141–146 (cit. on pp. 41, 43, 103).

6. Stanley-Marbell, P., M. S. Hsiao, and U. Kremer (2002). "A Hardware Architecture for Dynamic Performance and Energy Adaptation". In: *Lecture Notes in Computer Science, Springer-Verlag* 2325.1, pp. 33–52 (cit. on pp. 103, 111, 120).

7. Stanley-Marbell, P. et al. (2000). "Smart Messages : A System Architecture for Large Networks of Embedded Systems". In: *8th Workshop on Hot Topics in Operating Systems, HOTOS-VIII*, p. 153 (cit. on p. 122).

8. Stanley-Marbell, Phillip (2006). "Implementation of a Distributed Full-System Simulation Framework as a Filesystem Server". In: *Proceedings of the First International Workshop on Plan 9*. ISBN: 84-690-2787-5 (cit. on p. 31).

9. Stanley-Marbell, Phillip, Kanishka Lahiri, and Anand Raghunathan (2006). "Adaptive data placement in an embedded multiprocessor thread library". In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. Munich, Germany: European Design and Automation Association, pp. 698–699. ISBN: 3-9810801-0-6 (cit. on p. 110).

10. Tiwari, V., S. Malik, and A. Wolfe (1994). "Power Analysis of Embedded Software: A first Step Towards Software Power Estimation". In: *IEEE/ACM International Conference on Computer-Aided Design*, pp. 384–390 (cit. on p. 45).

Index

SF_B_ENDIAN, 23
SF_L_ENDIAN, 23
Block, 114
Ifc, 119
MOV, 27
Makefile, 68
OBJS, 68
PROGRAM, 68
$(OBJS), 68
$(PROGRAM), 68
devXXX_YYY, 67
devexcp_getintevt(), 67
devloc_getorbit(), 67
devloc_getvelocity(), 67
devloc_getxloc(), 67
devloc_getyloc(), 67
devloc_getzloc(), 67
devlog_ctl(), 67
devnet_ctl(), 67
devnet_framedelay(), 67
devnet_getfsz(), 67
devnet_getncolls(), 67
devnet_getncr(), 67
devnet_getncsense(), 67
devnet_getspeed(), 67
devnet_recv(), 67
devnet_xmit(), 67
devrand_getrand(), 67
devrand_seed(), 67
devrtc_getusecs(), 67
devtag_read(), 67
devtag_rttl(), 67
devtag_write(), 67
devtag_wttl(), 67
init.S, 68
init.o, 68
mkmantex, 117
schedule(), 113
scheduler(), 119
shasm.y, 121
udelay(), 67
volatile, 65
A Hardware Architecture for Dynamic Performance and Energy Adaptation, 103, 111, 120, 143
A discrete-time battery model for high-level power estimation, 47, 113, 143
Cygnus GnuPro Documentation, 66, 143
Power Analysis of Embedded Software: A first Step Towards Software Power Estimation, 45, 144
Smart Messages : A System Architecture for Large Networks of Embedded Systems, 122, 143
CLK, 94
ICLK, 94, 106
NINSTR, 106
(State *)S)->step(), 111
B*nnnn*, 114
Cache, 114
DUMPPIPE, 118
Makefile, 115
Netsegment, 119
Pipestage, 121
Pipe, 121
SF_X_ENDIAN, 115
SIM_GLOBAL_CLOCK, 117
Segbuf, 119
State, 117, 118
State structure, 111
battery_debug(), 112
battery_feed(), 112, 113
decode-hitachi-sh.h, 118
decode-ti-msp430.h, 118
help.c, 117
help.h, 118
lex.c, 118, 121
main.c, 113, 115
mkhelp, 117
network_clock(), 118
newbatt(), 112
opstr-hitachi-sh.h, 118
opstr-ti-msp430.h, 118
scheduler(), 115, 117
sf.h, 116
step(), 121
sys/include/bit.h, 114
sys/include/lex.inc, 121
token_table, 116

Acronym,LIF, 17
Acronyms,ADF, 16
Acronyms,ASF, 62, 82
Acronyms,SNR, 54
Acronyms,SOF, 17
Acronyms,STF, 17
Acronyms,SVF, 17
Adaptive data placement in an embedded multiprocessor thread library, 110, 143
Architecture Specification File, 25

battery lifetime, 45
battery low interrupts, 66
bearing, 59
benchmark implementation,**nic_hdlr()**, 70
benchmark implementation,**_vec_stub_begin**, 69
benchmark implementation,**_vec_stub_end**, 69
benchmark implementation,**argc**, 106
benchmark implementation,**argv**, 106
benchmark implementation,**devexcp_getintevt()**, 69
benchmark implementation,**devlog_ctl()**, 95, 98, 103, 105, 106
benchmark implementation,**devlog_ctl(off);**, 101
benchmark implementation,**devnet_recv()**, 70
benchmark implementation,**devnet_xmit()**, 67
benchmark implementation,**devrtc_getusecs()**, 101
benchmark implementation,**exit()**, 35
benchmark implementation,**hdlr_install()**, 69, 102
benchmark implementation,**intr_hdlr()**, 69, 102
benchmark implementation,**main()**, 68, 102
benchmark implementation,**main(int argc, char *argv[])**, 106
benchmark implementation,**main**, 68
benchmark implementation,**my_id**, 96, 97
benchmark implementation,**NIC_OUI**, 96
benchmark implementation,**printf()**, 94
benchmark implementation,**printf**, 33
benchmark implementation,**print**, 33
benchmark implementation,**REGSAVESTACK**, 102
benchmark implementation,**sleep()**, 101
benchmark implementation,**startup()**, 18, 68, 69, 102
benchmark implementation,**startup(int argc, char *argv[])**, 106
Benini, L., 47, 113, 143
Big Endian, 23
Big-endian, 115
Binutils, 24
BSD, 115
bss, 64
bzip2, 93
Cache, 43, 114
cache miss, 94
cache size, 43
circuit activity estimation, 45
clock interrupt, 66
Command File, 27
Commands,**battcf**, 49
Commands,battetalutnentries, 49
Commands,battetalut, 49
Commands,battileak, 48
Commands,battinominal, 49
Commands,battrf, 49
Commands,battvbattlutnentries, 49
Commands,battvbattlut, 49
Commands,battvlostlutnentries, 49
Commands,battvlostlut, 49
Commands,cacheoff, 73
Commands,ca, 107
Commands,cd, 98
Commands,clockintr, 73
Commands,c, 27
Commands,defndist, 78
Commands,dumpall, 46, 106
Commands,dumppipe, 105
Commands,dumpregs, 27, 36
Commands,dumpsysregs, 105
Commands,d, 46, 106
Commands,ff, 107
Commands,file2netseg, 52, 53
Commands,forceavgpwr, 46, 103, 107-109
Commands,help, 18, 27, 97
Commands,initrandtable, 78
Commands,load, 25
Commands,locstats, 60
Commands,mmap, 75, 109
Commands,netcorrel, 19, 52, 58
Commands,netdebug, 19, 52
Commands,netnewseg, 19, 52, 73
Commands,netnodenewifc, 52, 53, 73
Commands,netseg2file, 52, 53
Commands,netsegdelete, 52
Commands,netsegfaildurmax, 52
Commands,netsegfailprob, 58
Commands,netsegnicattach, 52, 73
Commands,netsegpropmodel, 52, 54
Commands,newnode, 27, 60, 73, 74
Commands,ni, 27, 106
Commands,nodefailprob, 58
Commands,nodetach, 105
Commands,numaregion, 75, 93
Commands,numasetmapid, 75
Commands,numastall, 75
Commands,numastats, 75
Commands,off, 18, 27
Commands,on, 27, 34, 35
Commands,pd, 36, 107
Commands,ps, 35, 46
Commands,quit, 98, 105
Commands,registervar, 78
Commands,run, 27, 34, 106
Commands,setfreq, 45, 101, 103, 109
Commands,setloc, 60
Commands,setquantum, 103, 104
Commands,setscalealpha, 46, 101
Commands,setscalek, 46, 100
Commands,setscalevt, 46, 100
Commands,setvdd, 45, 101, 103
Commands,showclk, 27, 94, 106
Commands,sigsrc, 60, 82, 83
Commands,sigsubscribe, 60
Commands,sizemem, 64, 73
Commands,srecl, 27, 73
Commands,Hitachi SH Assembler,

SLEEP, 101
Commands,Hitachi SH Assembler,MOV, 95
Commands,Hitachi SH Assembler,SLEEP, 105
Commands,Hitachi SH Assembler,TRAPA, 104
Commands,Hitachi SH Assembler,MOV, 36
Commands,modal commands, 73
Commands,battnodeattach, 34
Commands,newbatt, 34
Commands,on, 34, 35
Commands,run, 34
Commands,srecl, 34
Commands,addvaluetrace, 123
Commands,battalertfrac, 123
Commands,battcf, 123
Commands,battetalutnentries, 123
Commands,battetalut, 123
Commands,battileak, 124
Commands,battinominal, 124
Commands,battnodeattach, 124
Commands,battrf, 124
Commands,battstats, 124
Commands,battvbattlutnentries, 124
Commands,battvbattlut, 124
Commands,battvlostlutnentries, 125
Commands,battvlostlut, 124
Commands,bptdel, 125
Commands,bptls, 125
Commands,bpt, 125
Commands,cacheinit, 125
Commands,cacheoff, 125
Commands,cachestats, 126
Commands,ca, 125
Commands,cd, 126
Commands,clockintr, 126
Commands,cont, 126
Commands,c, 125
Commands,defndist, 126
Commands,delvaluetrace, 126
Commands,dumpall, 126
Commands,dumpmem, 127
Commands,dumppipe, 127
Commands,dumpregs, 127
Commands,dumpsysregs, 127
Commands,dumptlb, 127
Commands,dyninstr, 127
Commands,d, 126
Commands,ebattintr, 127
Commands,efaults, 127
Commands,ff, 128
Commands,file2netseg, 128
Commands,fltthresh, 128
Commands,forceavgpwr, 128
Commands,getrandomseed, 128
Commands,help, 128
Commands,hwseereg, 128
Commands,ign, 128
Commands,initrandtable, 129
Commands,initseestate, 129
Commands,listrvars, 129
Commands,load, 129
Commands,locstats, 129
Commands,l, 129
Commands,mallocdebug, 129
Commands,man, 129
Commands,mmap, 130
Commands,nanopause, 130
Commands,nd, 130
Commands,netcorrel, 130
Commands,netdebug, 130
Commands,netnewseg, 130
Commands,netnodenewifc, 131
Commands,netseg2file, 131
Commands,netsegdelete, 131
Commands,netsegfaildurmax, 131
Commands,netsegfailprobfn, 131
Commands,netsegfailprob, 131
Commands,netsegnicattach, 131
Commands,netsegpropmodel, 132
Commands,newbatt, 132
Commands,newnode, 132
Commands,ni, 132
Commands,nodefaildurmax, 132
Commands,nodefailprobfn, 132
Commands,nodefailprob, 132
Commands,nodetach, 142
Commands,numaregion, 133
Commands,numasetmapid, 133
Commands,numastatsall, 133
Commands,numastats, 133
Commands,n, 130
Commands,off, 133
Commands,on, 133
Commands,parseobjdump, 133
Commands,pauinfo, 134
Commands,pause, 134
Commands,pcbt, 134
Commands,pd, 134
Commands,pe, 134
Commands,pfun, 134
Commands,pf, 134
Commands,pi, 134
Commands,powerstats, 135
Commands,powertotal, 135
Commands,ps, 135
Commands,pwd, 135
Commands,quit, 135
Commands,q, 135
Commands,randprint, 135
Commands,ratio, 136
Commands,registerrvar, 136
Commands,registerstabs, 136
Commands,renumbernodes, 136
Commands,resetallctrs, 136
Commands,resetcpu, 136
Commands,resetnodectrs, 136
Commands,retryalg, 137
Commands,run, 137
Commands,r, 135
Commands,save, 137
Commands,sensorsdebug, 137
Commands,setbasenodeid, 137
Commands,setbattfeedperiod, 137
Commands,setbatt, 137
Commands,setdumppwrperiod, 137
Commands,setfaultperiod, 138
Commands,setflashrlatency, 138
Commands,setflashwlatency, 138
Commands,setfreq, 138
Commands,setifcoui, 138
Commands,setloc, 138
Commands,setmemrlatency, 138
Commands,setmemwlatency, 138
Commands,setnetperiod, 139
Commands,setnode, 139
Commands,setpc, 139
Commands,setphysicsperiod, 139
Commands,setquantum, 139

Commands,setrandomseed, 139
Commands,setscalealpha, 139
Commands,setscalek, 139
Commands,setscalevt, 140
Commands,setschedrandom, 140
Commands,setschedroundrobin, 140
Commands,settimerdelay, 140
Commands,setvdd, 140
Commands,sfatal, 140
Commands,sharebus, 140
Commands,showclk, 140
Commands,showpipe, 141
Commands,sigsrc, 141
Commands,sigsubscribe, 141
Commands,sizemem, 141
Commands,sizepau, 142
Commands,split, 141
Commands,srecl, 141
Commands,stop, 141
Commands,throttlewin, 142
Commands,throttle, 141
Commands,trace, 142
Commands,valuestats, 142
Commands,verbose, 142
Commands,version, 142
Commands,v, 142
communication channel,single-access, 106, 109
communication channels,multi-access, 104, 109
communication links, 51
communication media, 51

data, 64
dead code elimination phase, 66
direction, 59
distributed simulation, 83
dynamic instruction count, 94

embedded systems, 15, 59
emulator configuration,HOST, 79
emulator configuration,make all-gcc, 85
emulator configuration,SF_SIMLOG, 18
emulator configuration,SUNFLOWERROOT, 79, 94
emulator configuration,SUPPORTED-TARGET-ARCHS, 79
emulator configuration,SUPPORTED-TARGETS, 79
emulator configuration,TARGET-ARCH, 79
emulator configuration,TARGET, 79
emulator configuration,TREEROOT, 33
emulator implementation,(State *)S)->step(), 111
emulator implementation,BATT_LOW_EXCP_CODE, 66
emulator implementation,E->sp, 111
emulator implementation,Engine *E->sp, 111
emulator implementation,Engine, 18, 111
emulator implementation,EXCP_INTEVT, 66
emulator implementation,lex.c, 107
emulator implementation,NIC_RX_EXCP_CODE, 66
emulator implementation,pipeline-hitachi-sh.c,
emulator implementation,R0000,
emulator implementation,sf.y, 107
emulator implementation,SF_BATT, 46
emulator implementation,SF_BITFLIP_ANALYSIS, 46
emulator implementation,SIMCMD_CTL, 95
emulator implementation,State *, 111
emulator implementation,State, 111
emulator implementation,SUPERH_NIC_NMR, 95
emulator implementation,SUPERH_SIMCMD_CTL, 97
emulator implementation,SUPERH_SIMCMD_DATA, 95, 97
emulator implementation,SUPERH_USECS_*, 95
emulator implementation,T_NEWNODE, 78
emulator implementation,TMU0_TUNI0_EXCP_CODE, 66
emulator implementation,(State *)S)->stallaction(), 111
estimating energy, 45
exception, 63
EXCP_EXPEVT, 64
EXCP_INTEVT, 64
execution-driven, 15

failure, 43
Fast, Flexible, Cycle-Accurxate Energy Estimation, 41, 43, 103, 143
FATAL, 98
FF, 105
file formats,architectural specification file, 82
file formats,simulator configuration file, 81
file formats,simulator log file, 82
file formats,simulator platform-specific config file, 81
file formats,system configuration file, 79
file formats,conf/setup.conf, 79
files and folders,benchmarks/source/swradio/swr.m
27
files and folders,benchmarks/dist/SPEC2000, 87
files and folders,benchmarks/source/ALPBench/, 88
files and folders,benchmarks/source/bubblesort/bs
33
files and folders,benchmarks/source/bubblesort/bs
33
files and folders,benchmarks/source/bubblesort/in
33
files and folders,benchmarks/source/bubblesort/, 33
files and folders,benchmarks/source/bubblesort,

25
files and folders,benchmarks/source/[illegible]
75
files and folders,benchmarks/source/MiBench/,
88
files and folders,benchmarks/source/mpeg2dec/,
88
files and folders,benchmarks/source/mpeg2enc/,
88
files and folders,benchmarks/source/port/,
65, 69, 70, 101
files and folders,benchmarks/source/port,
102
files and folders,benchmarks/source/sbench/,
88
files and folders,benchmarks/source/SPEC2000/,
96
files and folders,benchmarks/source/sphynx3/,
87
files and folders,benchmarks/source/sw-radio/[illegible]
62
files and folders,benchmarks/source/sw-radio/[illegible]
101
files and folders,benchmarks/source/sw-radio/[illegible]
101, 102
files and folders,benchmarks/source/sw-radio/[illegible]
102
files and folders,benchmarks/source/sw-radio/,
62, 68, 96, 97, 101, 110
files and folders,benchmarks/source/,
25
files and folders,conf/setup.conf,
22, 24, 33, 79, 85, 94
files and folders,devsim7708.c,
95
files and folders,devsim7708.h,
95, 97, 98
files and folders,ilpa.h, 103,
104
files and folders,init.S, 101,
102
files and folders,LICENSE.txt,
111
files and folders,pau.c, 103,
106
files and folders,pau.h, 103
files and folders,README.md, 22
files and folders,sim/arch-darwin.c,
112
files and folders,sim/arch-Inferno.c,
112
files and folders,sim/arch-linux.c,
112
files and folders,sim/arch-OpenBSD.c,
112
files and folders,sim/arch-solaris.c,
112
files and folders,sim/batt.c,
113
files and folders,sim/batt.h,
113
files and folders,sim/battmodels/,
113
files and folders,sim/big-endian-hitachi-sh.h,
113
files and folders,sim/bit-utils.c,
113
files and folders,sim/bit.h,
113
files and folders,sim/cache-hitachi-sh.c,
114
files and folders,sim/cache-hitachi-sh.h,
114
files and folders,sim/config.darwin-ppc,
81
files and folders,sim/config.h,
46
files and folders,sim/decode-hitachi-sh.c,
104, 114
files and folders,sim/decode-hitachi-sh.h,
114
files and folders,sim/decode-ti-msp430.h,
114
files and folders,sim/dev430x1xxx.h,
115
files and folders,sim/dev7708.c,
114
files and folders,sim/dev7708.h,
114
files and folders,sim/devexcpt.h,
69
files and folders,sim/devnet-hitachi-sh.h,
69
files and folders,sim/devsim7708.c,
108, 115
files and folders,sim/devsim7708.h,
64, 69
files and folders,sim/devsunflower.c,
115
files and folders,sim/endian-hitachi-sh.h,
115
files and folders,sim/fault.c,
115
files and folders,sim/fault.h,
116
files and folders,sim/fdr.c,
116
files and folders,sim/fdr.h,
116
files and folders,sim/ilpa.h,
104
files and folders,sim/instr-hitachi-sh.h,
116
files and folders,sim/interrupts-hitachi-sh
69, 116
files and folders,sim/lex.c,
116, 118
files and folders,sim/little-endian-hitachi
113
files and folders,sim/machine-hitachi-sh.c,
117
files and folders,sim/machine-hitachi-sh.h,
117
files and folders,sim/machine-ti-msp430.c,
117
files and folders,sim/machine-ti-msp430.h,
117
files and folders,sim/main.c,
117
files and folders,sim/main.h,
64, 111, 118
files and folders,sim/Makefile,
18
files and folders,sim/mfns.h,
116
files and folders,sim/mkhelp,
118
files and folders,sim/mkmantex,
118
files and folders,sim/mkopstr-hitachi-sh,
118
files and folders,sim/mkopstr-ti-msp430,
118
files and folders,sim/network-hitachi-sh.c,

108, 118
files and folders,sim/network-hitachi-sh.h, 69, 119
files and folders,sim/op-hitachi-sh.c, 119
files and folders,sim/op-hitachi-sh.h, 119
files and folders,sim/op-ti-msp430.c, 119
files and folders,sim/pau.c, 120
files and folders,sim/pau.h, 120
files and folders,sim/pic.c, 121
files and folders,sim/pic.h, 121
files and folders,sim/pipeline-hitachi-sh.c, 119, 121
files and folders,sim/pipeline-hitachi-sh.h, 119, 121
files and folders,sim/pipeline-ti-msp430.c, 120
files and folders,sim/pipeline-ti-msp430.h, 120
files and folders,sim/power.c, 120
files and folders,sim/power.h, 121
files and folders,sim/randgen.c, 120
files and folders,sim/randgen.h, 120
files and folders,sim/regaccess-hitachi-sh.c, 120
files and folders,sim/regaccess-ti-msp430.c, 120
files and folders,sim/regs-hitachi-sh.h, 121
files and folders,sim/regs-ti-msp430.h, 121
files and folders,sim/sf-types.h, 69
files and folders,sim/sf.h, 112
files and folders,sim/sf.y, 78, 121
files and folders,sim/syscalls-Inferno.c, 122
files and folders,sim/syscalls.c, 122
files and folders,sim/syscalls.h, 122
files and folders,sim/tag.c, 122
files and folders,sim/tokenhandling.c, 96
files and folders,sim/utils/batt-test.c, 112
files and folders,sim/utils/ilpa.c, 104
files and folders,sim/, 23, 111
files and folders,sunflower.out, 106
files and folders,superh.ld, 102
files and folders,swradiosource.c, 101
files and folders,sys/kern/superH/sh7708.h, 65
files and folders,tools/Makefile, 85
files and folders,tools/source, 22
files and folders,utils/logmarkparse/, 89
Files,architectural specification file, 62
firmware, 64
frame size, 73
full-system hardware emulator for networked embedded systems, 15

GCC, 24
gzip, 93

Hitachi SH, 117, 119, 121
Hitachi SH ISA, 114
Hitachi SH3, 116
Hitachi SH3 SH7708, 114
Host OS and shell commands,g++, 85
Host OS and shell commands,gcc, 85
Host OS and shell commands,logmarkparse, 89
Host OS and shell commands,make cross, 24, 85
Host OS and shell commands,make install, 62
Host OS and shell commands,make TREEROOT = full-path-to-simulator-distribution, 96
Host OS and shell commands,make, 18, 33, 62, 85
Hsiao, M., 41, 43, 103, 143
Hsiao, M. S., 103, 111, 120, 143

Implementation of a Distributed Full-System Simulation Framework as a Filesystem Server, 31, 143
Installation, 21
Installation,Compilation, 23
Installation,Compilation,Applications, 25
Installation,Compilation,GCC, 24
Installation,Compilation,Sunflower, 23
Installation,Obtaining Sources, 21
Installation,Setup, 22
instruction-level power model, 45
Intel x86, 23, 115
interactive interface, 25
interrupt, 63
Interrupts, 66
interrupts,interrupt vector base, 63, 69
interrupts,timer, 101

Kremer, U., 103, 111, 120, 143

Lahiri, Kanishka, 110, 143
License, 111
link speed, 73
Linux, 115
Lithium Ion battery, 47
Little Endian, 23
little-endian, 115
location, 59

MacOS X, 115
Malik, S., 45, 144
MAXIM MAX1653, 47
Memory, 43
Memory Map, 63
memory map, 73
memory mapped I/O, 64
memory mapped registers, 64
memory size, 43
memory-mapped registers, 95, 97
message passing, 16
metrics, 45
metrics,battery lifetime, 45
metrics,end-to-end latency, 45
metrics,performance, 45
metrics,power dissipation, 45
metrics,throughput, 45
metrics,timeliness, 45
monitor, 64
multi-processor, 15
Myrmigki, 103

Network Interface, 43
network interface, 52
network interface interrupts, 66
network interfaces, 45, 51
network links, 51
network media, 51
network segments, 51
network trace files, 53
network traffic traces, 17
Nishimura, Takuji, 77, 143
node location input file (LIF), 17
node location trajectory file, 60

online help, 118
Operating system, 65
Operating Voltage, 43
OS, 65
Overview, 15

Panasonic CGR18 family, 47
PAU, 120
Peripherals, 43
power dissipation, 45
power modeling, 121
PowerPC, 115
processing element, 16
processing node, 43
processors, 45
program counter, 104
propagation delay, 74

Raghunathan, Anand, 110, 143
registers,memory mapped registers, 64
registers,EXPEVT, 100, 105
registers,INTEVT, 100, 105
registers,R14, 100
registers,R15, 100
Ross, Sheldon M., 77, 143
RS-232, 43
runnable, 34
Running Sunflower, 25

shared memory, 16
signal attenuation profile, 60
signal group, 54
signal sample value file (SVF), 17
signal trajectory file (STF), 17
Simulation, 77, 143
simulation, 17
simulation modes,cycle-accurate (CA), 107
simulation modes,fast functional (FF), 107
simulation output file (SOF), 17
Simulator Command File, 25
Simulator Command Language, 27
sleep, 95
Smart Messages, 122
Solaris, 115
SPARC, 23, 115
Stanley-Marbell, P., 41, 43, 103, 111, 120, 122, 143
Stanley-Marbell, Phillip, 31, 110, 143
system architecture description file, 16
system calls, 122

Tables of 64-bit Mersenne twisters, 77, 143
tag memory, 122
tar, 93
text, 64
TI MSP430, 114, 117, 119, 121
TI MSP430 F11X, 115
Tiwari, V., 45, 144
transmission delay, 74
TRAPA, 122

voltage regulator, 47

Wolfe, A., 45, 144

YACC, 121

$\left(10_2\right)$

University of Cambridge
Department of Engineering

Physical
Computation
Laboratory