Sunflower Simulator Manual

sunflower-1.0-release-source-beta.3

This is a draft; it needs a lot of updating. Be patient.

Phillip Stanley-Marbell

Energy Aware Computing Research Group

Department of Electrical and Computer Engineering

Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Contents

1 Overview and Installation

Computers are useless. They can only give you answers.

—Pablo Picasso

1.1 OVERVIEW

Sunflower is an execution driven architectural simulator and energy estimation framework, which models an embedded system based on the Hitachi SH3 architecture [2]. Although it currently only simulates the Hitachi SH architecture family, the simulator can easily be extended to model other processor families, due to its modular construction. The SH3 is the third generation of the Hitachi SuperH RISC architecture [2]. It is a 32-bit RISC architecture, with a 5-stage pipeline, most instructions completing in a single cycle. Instructions are 16-bit for high code density. Figure 1.1 shows the functional layout of the SH3 architecture.

The simulator models:

- *Processing nodes*: CPU core, on-chip cache, various on-chip peripherals, off-chip memory, and 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.
- *Interconnection links connecting the processing nodes*: Interconnection links may be instantiated as necessary to create networks, and the network interfaces of processing nodes are

attached to links. The links may be configured for variable transmission delay (link speed), link frame size, link failure probability (bit error rate), and link failure mode (e.g. constant rate intermittent failures or exponentially worsening failures).

- *Batteries and DC-DC converters*: Each processing node must be attached to a source of energy. The first order effects of discharge rate, DC-DC converter 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 DC-DC converter characteristics are those for a Dallas Semiconductor/MAXIM MAX1653.

- *Failures*: The failure rate, average failure duration and failure mode of both processing nodes and interconnection links may be specified. Correlated failures between nodes and links may also be enabled, by specifying appropriate correlation coefficients.

In addition to an accurate functional model, it includes two complementary means of estimating energy cost of application software — An empirical instruction level power model similar to [6] and circuit activity estimation. Also modeled are features not present on the target hardware, such as dynamic voltage scaling, clock speed setting and a broad range of on-chip cache configurations, to permit the investigation of architectural tradeoffs for energy efficiency, and the investigation of software and hardware architectures for dynamic voltage scaling and clock speed setting.

1.2 SIMULATION INFRASTRUCTURE

Processing Devices

At the core of the simulation framework is the modeling of instruction execution. Modeling applications at the level of detail of the simulation of the execution of their compiled code, makes it possible to employ the simulation framework as a debugging platform for actual prototypes. It also makes it possible to determine important interactions between the requirements of computation, communication and reliability, and the effects of these constraints on power consumption.

The simulation framework includes two different architectural models, one for the Hitachi SH architecture, based on the Hitachi SH3 SH7708 (Figure 1.1), and the other of the TI MSP430 architecture(Figure 1.2). Support for new architectures is easily added, and 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 two complementary means of estimating the energy cost of application software—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

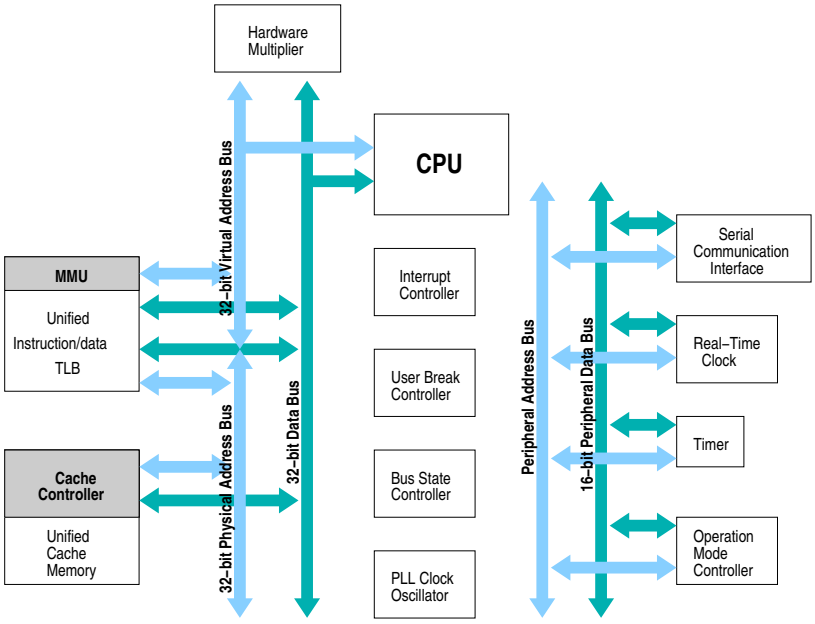

Fig. 1.1 Hitachi SH3 architecture

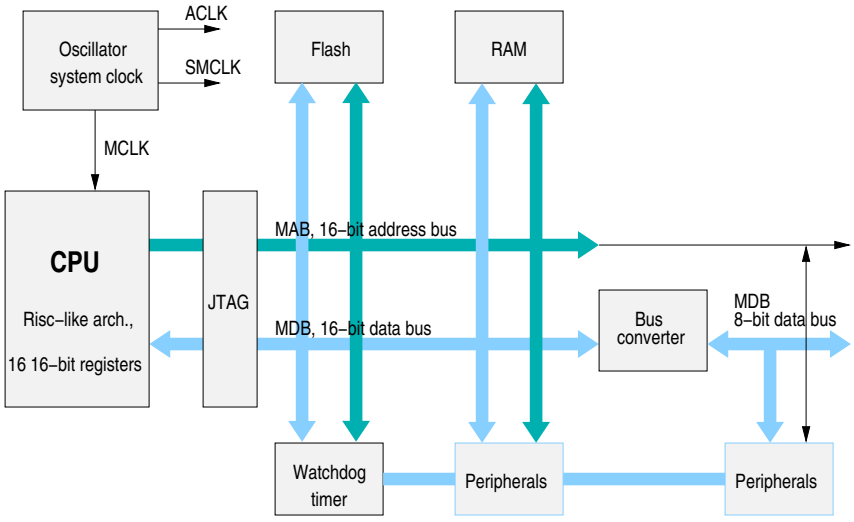

Fig. 1.2 TI MSP430 architecture

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 [4]. The instruction level power 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 [4].

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 simulation framework.

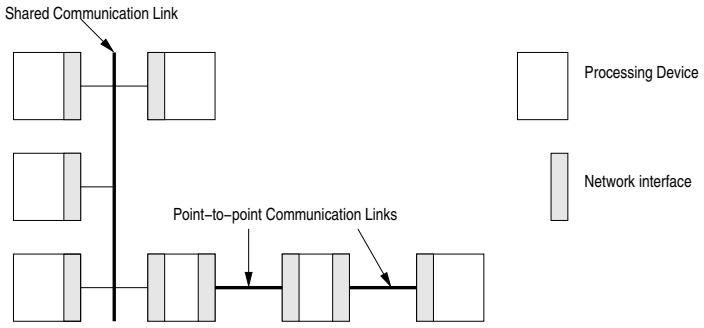

Fig. 1.3 An example communication topology

Communication

The communication modeling architecture in the simulation framework enables the construction of a variety of communication topologies, such as that illustrated in Figure 1.3. Processing elements communicate with each other through their *network interfaces*, which are connected to *communication links*. The behavior of the network interface modeling is independent of the processing core model chosen for a given device. The properties of a modeled communication links and interfaces are flexible and parameterizable, enabling them to be configured to model the properties of media ranging from one with properties like RS-232, to one that behave like Ethernet.

Each communication interface on a device must be associated with a communication link. Each *communication link* or *network segment* may be configured for the following specific properties:

- *Frame size* — Data is transmitted on a communication link in groups of bytes referred to as a "frame".

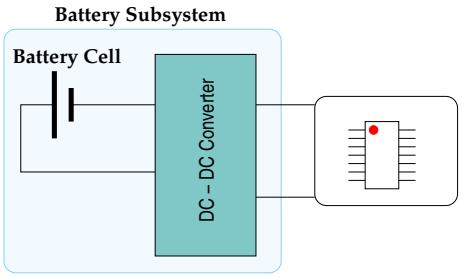

Fig. 1.4 Organization of battery subsystem. The DC-DC converter is required to obtain a constant voltage to power electronics, due to dependence of battery cell terminal voltage on battery state of charge.

- *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 below.

In order to ensure network interfaces are always compatible with the networks to which they are attached, network interfaces inherit the aforementioned properties from a network segment to which they attached. The transmission and receive power consumption of a network interface specific 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.

Battery Subsystem

The simulator includes a detailed discrete-time battery modeling engine based on [1]. 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 *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 1.4 to illustrate this further.

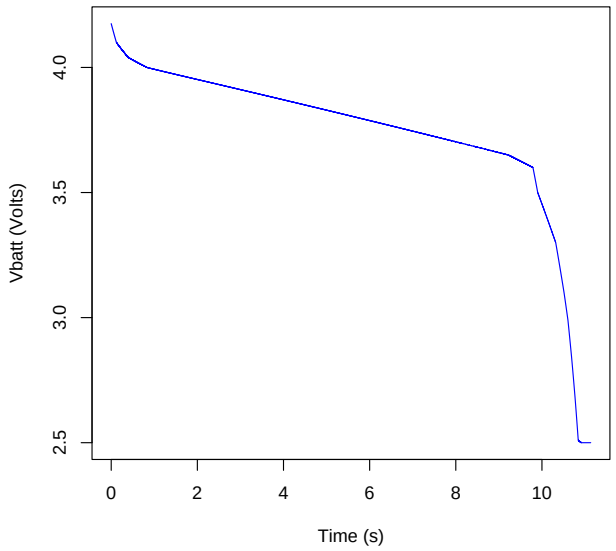

Fig. 1.5 Variation of battery cell terminal voltage over time for a nominal current draw of 150mA from outside the battery subsystem.

In order to model different types and sizes of batteries and DC-DC converters, the model (and its implementation in the simulator employed in this work) uses lookup tables (LUTs) and additional fixed parameters to store the characteristics of specific batteries. The default battery characteristics employed in our implementation, are those for a Li Ion cell from the Panasonic CGR18 family. The DC-DC converter characteristics employed are those for a Dallas Semiconductor/Maxim MAX1653 device. User lookup tables may be loaded into the simulator to mimic other devices characteristics, for both the battery cell and DC-DC converter. Figure 1.5 shows the dependence of battery terminal voltage with time for a nominal discharge rate of 150mA. The data in Figure 1.5, although showing the voltage at the terminals of the battery cell, also includes the effect of DC-DC conversion, and depicts the lumped behavior of the battery cell if the battery subsystem were attached to electronics that had a constant current draw of 150mA.

The components of the battery properties are illustrated in Figure 1.6. The parameters of interest in this work are V_r, a measure of the rate of discharge, V_{rate}, a *time-sluggish* (i.e. low-pass filtered) version of V_r, V_{lost}, which models the dependence of battery terminal voltage on the magnitude of V_{rate} for a particular battery type (from a LUT). Lastly, V_C models the instantaneous state of charge, taking in to consideration V_{lost}.

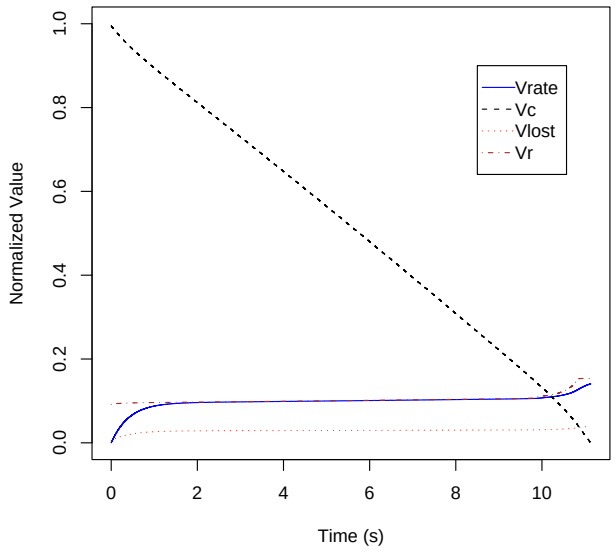

Fig. 1.6 Variation of components of battery model with time for a nominal current draw of 150mA from outside the battery subsystem.

Modeling Failures

The simulation framework models failures in both processing devices and communication links. Failures in processing devices manifest as intermittent stalls of the entire processing device, for the duration of the failure. Failures in communication links manifest as intermittent loss of carrier for the duration of the failure. Failures such as bit errors introduced into the communication stream and into device computation are not currently supported, but are planned. 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.

1.3 INSTALLATION

The simulator is currently distributed as a source archive, and can be obtained from:

```
http://www.sunflowersim.org
```

To uncompress the archive and build the simulator, you will need *at least* 300MB of space. This is because the simulator is also distributed with the source for the compiler and associated

tools that will be needed to build applications to run *over* the simulator. To uncompress and extract the archive:

```
tar -zxvf sunflower-1.0-release-source-beta.3.tgz
```

The above will create a directory, `sunflower-1.0-release-source-beta.3/`. The root of this directory will henceforth be referred to as `$ROOT`.

!!! NOTE !!! **The simulator, compiler build and applications, all rely on a single configuration file, `$ROOT/conf/setup.conf`. You will need to modify the first line of this file to reflect your installation location.**

`$ROOT/conf/setup.conf` will should look like the following:

```
## Filename: $ROOT/conf/setup.conf

##
##      You will want to change the following to suit your setup:
##
ROOT            = /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   = $(ROOT)/tools/source/gcc-2.95.3/gcc/ginclude/
PREFIX          = $(TOOLS)/$(TARGET)
...
```

1.4 COMPILING THE SIMULATOR

Once you have correctly edited the `ROOT` and `HOST`[1] fields of the configuration file, you should be able to build the simulator. The simulator source resides in the directory `sim/superH` from the root of the distribution. For Intel architecture UNIX machines (Linux-x86, OpenBSD-x86, etc), You should be able to build the simulator by simply typing `make`. For other architectures (e.g. Sun SPARC Solaris), you will need to change one flag in the simulator makefile. In the simulator directory, there are several alternate makefiles, such as `Makefile.linux`. You may want to copy the appropriate Makefile to overwrite the existing default for the platforms with customized makefiles available.

For performance reasons, the simulator 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 simulator makefile 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`. The example below illustrates the first few lines of a simulator makefile that has the byte-order flag `SF_B_ENDIAN` set:

```
## Filename: $ROOT/sim/superH/Makefile

include ../../conf/setup.conf

TARGET  = sf
BIN     = $(HOME)/bin

LINT    = echo
LD      = ld
CC      = gcc
BISON   = bison
BFLAGS  = -t -v
...
```

With that solitary change to the makefile (where appropriate), you should be able to build the simulator. If you encounter *any* issues at this point, send mail to `pstanley@ece.cmu.edu` *with a copy of the encountered errors*.

1.5 COMPILING THE COMPILER

Once you have the simulator built, you may now proceed to compiling the compiler — In order for you to use the compiler (GCC) to generate code for the target architecture (Hitachi SH), you must compile GCC, configured to generate code for the Hitachi SH. 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).

[1]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).

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`. 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) and then GCC itself.

The compilation process will take a while, on the order of 30 minutes. Once it completes, you should have several files in the `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 simulator, so for the most part, you do not have to remember that they are they or reference them directly for that matter.

1.6 COMPILING APPLICATIONS

A few example applications are provided with the simulator, and these reside in `BENCHMARKS/c/` from `$ROOT/sim/superH/`.

Each example application under `$ROOT/sim/superH/BENCHMARKS/c/` 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.

1.7 RUNNING THE SIMULATOR

When the simulator builds successfully, a binary, '`sf`', should be produced. You should be able to run it by typing `./sf`. The simulator takes no command line arguments, though it can be scripted by providing it a simulator command file as standard input.

The simulator has an interactive interface. Starting the simulator instantiates a single processor, and attaches the interactive interface to it (see Figure 1.7). 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.

Fig. 1.7 The interactive command interface for the Sunflower simulator.

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` command to load it in.

1.7.1 Simulator 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 the modeled architecture, the binary can be loaded into a processor node using the `srecl` command. Once the binary has been loaded into memory, the `run` command is issued to activate the processor to which it was loaded, and the `on` command issued to set the simulator running. At any time, the `off` command may be issued to pause the simulation. The state of the execution may be probed, for example the register file of a processor may be examined using the `dumpregs` command.

In addition to such commands for controlling execution, the command interface also acts as an assembler for the Hitachi SH ISA, thus any valid assembler mnemonic may be entered at the command line. For example, entering `MOV #4, R5` at the command interface will set the contents of register `R5` of currently attached processor to the value 4. The entire command language is listed in the appendix to this document, and consists of upward of 180 commands.

An example simulation configuration file is included with the simulator in the directory `sim/superH/BENCHMARKS/c/swradio/`. By convention, simulator configuration files have the suffix ".m". To get a quick feel for the command language, browse through `sim/superH/BENCHMARKS/c/swradio/test.m` and match the commands therein to entries

```
done with resetcpu...
Priming Decode Cache...done.
Initialising random number generator with seed 73650...


[ID=0 of 1][PC=0x8000000][3.30E+00V, 6.00E+01MHz] help
There are 182 commands and 16 aliases:
.ALIGN          .COMM           .DATA           .FILE
.GLOBAL         .LONG           .ORG            .TEXT
ADD             ADD             ADDC            ADDV
AND             ANDB            BATTALERTFRAC   BATTNODEATTACH
BATTSTATS       BF              BF.S            BF/S
BRA             BRAF            BSR             BSRF
BT              BT.S            BT/S            C
CA              CACHEINIT       CACHEOFF        CACHESTATS
CLOCKINTR       CLRMAC          CLRS            CLRT
CMP/EQ          CMP/GE          CMP/GT          CMP/HI
CMP/HS          CMP/PL          CMP/PZ          CMP/STR
CONT            DB              DC              DIV0S
DIV0U           DIV1            DMULS.L         DMULU.L
DP              DT              DT              DUMPBATTSTATS
DUMPCACHE       DUMPMEM         DUMPPIPE        DUMPPWR
DUMPREGS        DUMPSYSREGS     DUMPTIME        DYNINSTR
EBATTINTR       EFAULTS         EXTS.B          EXTS.W
```

Fig. 1.8 The `help` command lists all the available commands. More information on a particular command may be obtained with the `man` command.

in the appendix. The `help` command lists all available commands (see Figure 1.8, and entering '`man` *command name*' will provide a brief summary of the action of the command, as illustrated in Figure 1.9

Fig. 1.9 Using the built-in manual pages. Shown here is the manual entry for the `setfreq` command.

2 Simple Example – Loading and running a single application

I'm allowed to put this radio under your bed

—Anonymous

2.1 RUNNING A SIMPLE EXAMPLE ON THE SIMULATOR

The simplest example to run is the *bubblesort* application. Binaries to be run over the simulator are in Motorola S-RECORD format and end in the suffix *.sr*.

To compile the bubblesort application, given that the compilation tools have been correctly installed, change directory to `sim/superH/BENCHMARKS/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`.

Start up the simulator by typing `./sf`. To load a single binary into the simulator for simulation, type `srecl` *filename*. This will load the binary to be run over the simulator into the simulated machine memory. To run the program, you need to type `RUN` *and then* `ON`.

Figure 2.1 shows a screen capture of a session where a user starts up the simulator (`./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` and then `ON`). The `RUN` command marks the current processor node (node 0, as shown in the leftmost side of the simulator prompt) as *runnable*. The `ON` command acts as the "Big Switch" to turn the simulator off or on.

Figure 2.2 shows sample output from the end of a simulated application. The simulator halts the simulation and prints some statistics when the application executes a `exit()` system call,

Fig. 2.1 Loading and running a single binary on the simulator

which is eventually seen by the simulator as an exception. After the user enters the `ON` command, the simulator begins the execution of the instructions that were previously loaded into the simulated machine's memory. In the figure, this simulation took 0.05 seconds on the host machine that was running the simulator. The simulated time elapsed, from the point of view 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 16.6667ns, corresponding to an operating frequency of 60MHz. Given the number of processor cycles simulated, and the time taken to perform this simulation on the host machine, the simulator reports a *simulation rate* of 793.62K 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` command at the command line.

The output of *simulated* applications is always printed out by the simulator in colored text on a blue background. In Figure 2.2, The output is

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

[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 simulator. However, in terms of functional simulation, only the number of clock cycles simulated have any real significance.

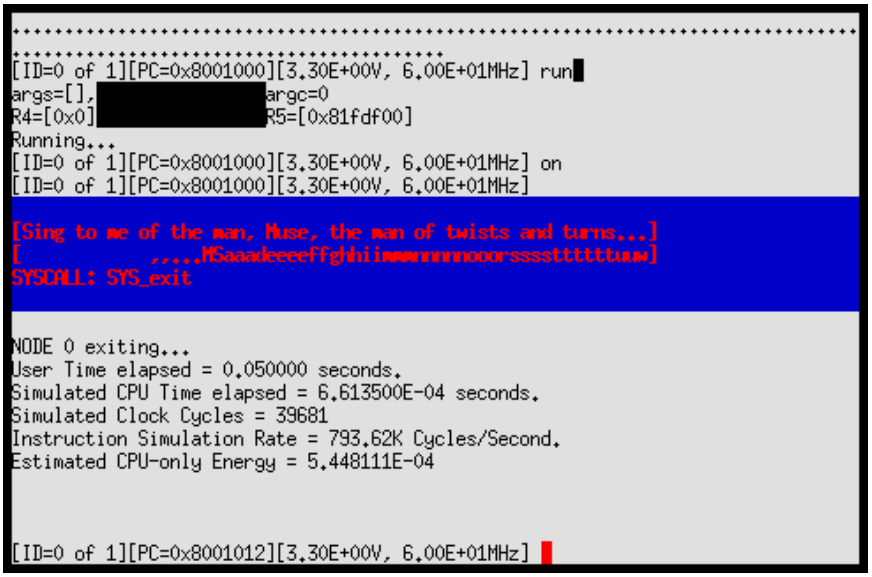

Fig. 2.2 Sample output from the end of a simulated application. The simulator halts the simulation and prints some statistics when the application executes a `exit()` system call, which is eventually seen by the simulator as an exception.

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 2.3 shows the output of the `DUMPREGS` command, which displays the contents of the machine's general purpose registers.

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 2.3, to copy the contents of the register `R2` to the register `R7`, a user could do this by issuing the Hitachi SH instruction to do this, the `MOV` instruction, from the command line. Prior to doing this however, the simulator'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` command, as illustrated in Figure 2.4

The `DUMPREGS` command is now issued again, and it shows that registers `R2` and `R7` now have the same value of `0x00fffffe88`, as shown below in Figure 2.5.

```
Simulated Clock Cycles = 39681
Instruction Simulation Rate = 793.62K Cycles/Second.
Estimated CPU-only Energy = 5.448111E-04



[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz] dumpregs
R0              11111111111111111111111111111111  [0x00ffffffff]
R1              11111111111111111111111010001000  [0x00fffffe88]
R2              11111111111111111111111010001000  [0x00fffffe88]
R3              00000000000000000000000000000000  [0x0000000000]
R4              00000000000000000000000000000001  [0x0000000001]
R5              00000000000000000000000000001010  [0x000000000a]
R6              00000000000000000000000000000000  [0x0000000000]
R7              00000000000000000000000100001000  [0x0000000108]
R8              00000000000000000000000000000000  [0x0000000000]
R9              00000000000000000000000000000000  [0x0000000000]
R10             00000000000000000000000000000000  [0x0000000000]
R11             00000000000000000000000000000000  [0x0000000000]
R12             00000000000000000000000000000000  [0x0000000000]
R13             00000000000000000000000000000000  [0x0000000000]
R14             00000000000000000000000000000000  [0x0000000000]
R15             00001000000100000000000000000000  [0x0008100000]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz]
```

Fig. 2.3 Loading and running a single binary on the simulator

```
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz] pd
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz] dumpregs
R0              11111111111111111111111111111111  [0x00ffffffff]
R1              11111111111111111111111010001000  [0x00fffffe88]
R2              11111111111111111111111010001000  [0x00fffffe88]
R3              00000000000000000000000000000000  [0x0000000000]
R4              00000000000000000000000000000001  [0x0000000001]
R5              00000000000000000000000000001010  [0x000000000a]
R6              00000000000000000000000000000000  [0x0000000000]
R7              00000000000000000000000100001000  [0x0000000108]
R8              00000000000000000000000000000000  [0x0000000000]
R9              00000000000000000000000000000000  [0x0000000000]
R10             00000000000000000000000000000000  [0x0000000000]
R11             00000000000000000000000000000000  [0x0000000000]
R12             00000000000000000000000000000000  [0x0000000000]
R13             00000000000000000000000000000000  [0x0000000000]
R14             00000000000000000000000000000000  [0x0000000000]
R15             00001000000100000000000000000000  [0x0008100000]
[ID=0 of 1][PC=0x8001012][3.30E+00V, 6.00E+01MHz] mov r2, r7
[ID=0 of 1][PC=0x8001014][3.30E+00V, 6.00E+01MHz]
```

Fig. 2.4 Loading and running a single binary on the simulator

Fig. 2.5 Loading and running a single binary on the simulator

3 Extended Example – Beamforming in wired network

Date: Thu Nov 14, 2002 01:55:06 PM US/Eastern
Subject: Re: [9fans] how to avoid a memset() optimization
...
Jarring chord. The door flies open and Cardinal Ximinez of Spain enters flanked by two junior cardinals. Cardinal Biggles has goggles pushed over his forehead. Cardinal Fang is just Cardinal Fang.
Ximinez:
Nobody expects the Volatile Inquisition. Our chief weapon is surprise that your code doesn't work ... surprise and fear that you forgot an optimisation parameter... fear and surprise ... our two weapons are fear and surprise ... and ruthless efficiency instead of correctness. Our THREE weapons are fear, surprise and ruthless efficiency and an almost fanatical devotion to RMS ... Our FOUR ... no ... AMONGST our weaponry are such elements as fear, surprise ... I'll come in again.
(exit and exeunt)

—jmk@plan9.bell-labs.com, on 9fans

3.1 OVERVIEW

In this chapter, we'll step through the motions of setting up a simulation configuration and compiling the source for, a complete application. This will serve as a means of exposition of the most commonly necessary commands in the simulator, the compilation process for applications

to be run over the simulator, and the constructs often used in implementing such applications in the C programming language.

3.2 WHIRLWIND TOUR OF EMBEDDED PROGRAMMING

For those familiar with writing software for embedded microcontrollers, or writing or porting operating systems for general purpose microcomputer systems, most of the topics of this section should be intuitively obvious. If familiar with ideas such as *Memory Maps* and *Memory Mapped IO* this section may be skipped to go directly to §3.3.

3.2.1 Memory Map

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

Figure 3.1 illustrates the memory map of the system 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 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. 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 §3.2.2. The mapping of registers to particular memory addresses are listed in `sim/superH/dev7708.h` in the simulator source distribution.

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 1MB, as in Figure 3.1, the memory space spans `0x8000000` to `0x80FFFFF`. The default memory size is defined in the simulator source file `sim/superH/mem.h`, however, it is never necessary to change that — the size of modeled memory can be adjusted with the `SIZEMEM` 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].

[1]The discussion in this section sidesteps discussions of virtual memory organizations, as it is not of relevance here.
[2]The term `bss` is a historical vestige from UNIX. It stands for *Block Started by Symbol*.

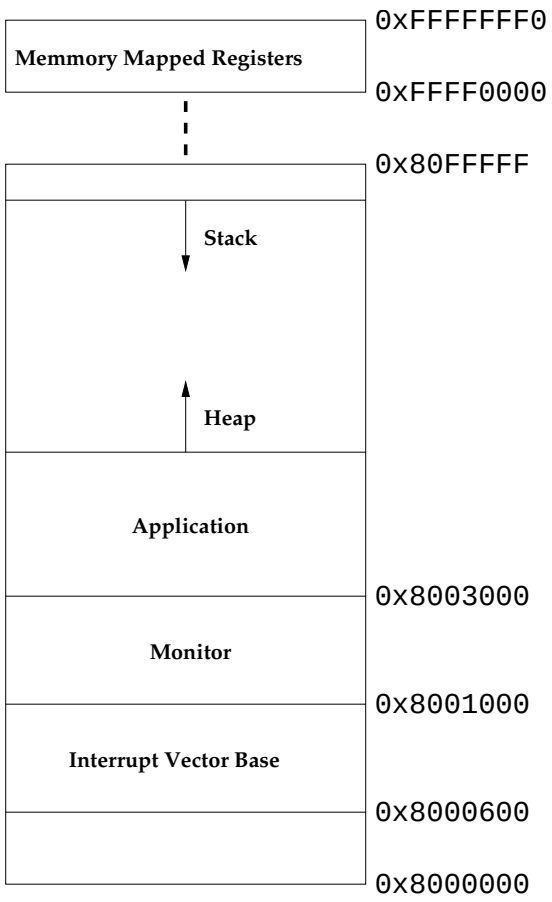

Fig. 3.1 Memory Map of the system modeled by the Sunflower simulator.

3.2.2 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 §3.2.1 is a hardware register which is accessed by reading from memory address `0xFFFFFFD4`.

3.2.3 "`main` + `intr_handler()`" — Life without an OS

Due to the level of detail modeled by the simulator (i.e., the level of detail of actual hardware instruction execution, interrupts etc.) constructing applications for the simulator is no less trivial that it is to, say, write an application to run on an embedded microcontroller system. Most programmers are probably very comfortable writing applications to run over an operating system,

with the operating system handling interrupts and providing primitives that abstract away the details of the underlying architecture.

Although it is possible to boot commodity or custom operating systems over the simulator, it is generally used in conjunction with applications that run in the absence of an operating system. Such applications 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. 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. The reason this is important is that, 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");
}
```

3.3 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.

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

Table 3.1 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.

3.4 UTILITY ROUTINES : DEVNET_XMIT(), UDELAY() AND FRIENDS

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()`, `devnet_recv()`; The `udelay()` routine provides a calibrated busy microsecond delay. Table 3.1 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. There is no real justification for not placing them in a library.

3.5 EXAMPLE : BENCHMARKS/C/BEAMSLAVE

The directory `BENCHMARKS/c/beamslave` contains an implementation of a slave node which takes part in a partitioned beamforming or *radar* application. The following sections describe the various components that go into the final complied application.

3.5.1 The `Makefile`

The makefile determines which source files are compiled into a given binary, their dependencies and the tools necessary for their compilation. Each makefile contains a variable, `PROGRAM`, which is usually 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 is essentially 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.

3.5.2 Startup code: `init.S`

The `init.S` file contains the assembly startup code. It just 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
```

3.5.3 The main body : `beamslave.c`

The main body of `beamslave.c` is the routine `startup()`. Typically, applications to be run over the simulator are written as one main event loop, in the routine `startup()`, with a fair amount of the actual work residing in the interrupt handlers.

3.5.4 Frame format and transmitting frames

The layout of frames sent over the network is shown in Figure 3.2. The data received contains a 34 byte header : source address (16 bytes) destination address (16 bytes) and frame length (2 bytes). There is some code in the applications that decodes this header, then sets the variable `tdataptr` to point to the remainder of the frame (i.e. after the header).

The routine `devlog_ctl()` is used to pass strings to the simulator to be interpreted as commands, as well as printing its second argument to a log file. This means that any command that you can type at the command interface (i.e. all 200 of them :) can also be called by applications running *over* the simulator. Usually, you will do:

```
        devlog_ctl(" dumptime BLAHBLAH");
```

to cause an entry to be written to the logfile which contains the current clock cycle at which the call was made, and followed by the string BLAHBLAH

For each node created in the simulation, a logfile, simlog::NNN::, where NNN is the node ID is created. The call to devlog_ctl will cause the message to be written to the logfile for the particular node executing the call.

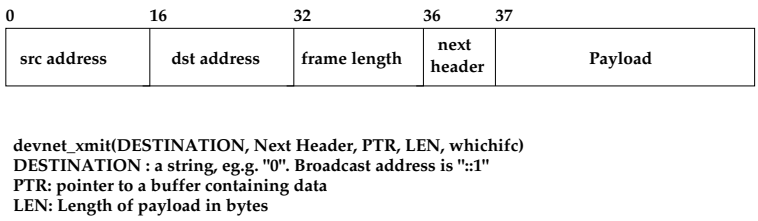

Fig. 3.2 Frame layout for data transmitted in example applications

3.6 RECEIVING FRAMES IN THE INTERRUPT HANDLER

The 16 byte OUI is (NIC_OUI is a 16-byte (128 bit) per-node address) currently the string representation of the decimal node ID, i.e. there are a maximum of 10^{16} - 1 possible node IDs. OUI stands for "Organizationally Unique Identifier", and that is the official name people use for MAC addresses e.g. Ethernet MAC addresses.

What the 'for' loop to calculate `my_id` does is, it basically 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. I.e., if you have the string char *mystring[] = "165", you can convert it to an `int` by doing:

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

3.7 SETTING UP A NETWORK OF NODES RUNNING BEAMSLAVE

To set up a networked simulation the steps involved are

1. Create the necessary network links with the `NETNEWSEG` command. You can specify properties of the link such as frame size, link speed (transmission delay), propagation delay, failure probability, mean failure duration, and more.
2. Create the necessary nodes with the `NEWNODE` command. You can specify various parameters for each node such as its operating voltage, frequency, cache size and configuration, failure probability, and more.
3. Instantiate network interfaces (IFCs) on the nodes. A node may have multiple network interfaces.
4. Connect each network interface each node to a particular instantiated link. This step determines, in essence, the topology; By using different connections, you can model a shared bus, point to point links, a torus, hypercube, mesh, etc.
5. Create batteries.
6. Attach the nodes to batteries. You can attach several nodes to batteries, provided they do not draw more than 1500mA of current.

Whenever a message is being sent from one node to the other, and the simulator is handling the delivery of the data, it prints out such a message, so that is how you know if nodes are communicating.

3.7.1 Instantiating nodes

Instantiate nodes with the with the `NEWNODE` command. The following creates a node, configures its failure probability, attaches it to battery number 1, sets the threshold for battery interrupts to 0.8, enables the generation of low battery interrupts, disables cache simulation (thus

perfect cache behavior) and employs the simulators fast functional mode. It then enables clock interrupts and creates a few interfaces each with transmit and receive power consumption of 0.1W and attaches them to links. Finally, the code to be run on the node is loaded into the memory of the simulated node and the node is marked as runnable.

```
newnode                 0 0 0 0 0
nodefailprob            0.0
nodefaildurmax          1000000
battnodeattach          1
battalertfrac           0.8
ebattintr
cacheoff
ff
clockintr
netnodenewifc           0 0.1 0.1 0 0 0 0 0 1024 1024
netsegnicattach         0 1
netnodenewifc           1 0.1 0.1 0 0 0 0 0 1024 1024
netsegnicattach         1 2
netnodenewifc           2 0.1 0.1 0 0 0 0 0 1024 1024
netsegnicattach         2 3
netnodenewifc           3 0.1 0.1 0 0 0 0 0 1024 1024
netsegnicattach         3 4
srecl                   beamslave.sr
run
```

3.7.2 Creating an interconnection network

Create new links with the `NETNEWSEG` command.

```
netnewseg               0 256 300000000 1600000 0 0 0 0 0 0 0 0 0
```

The above creates a new segment with a frame size of 256 bytes, propagation delay corresponding to the speed of light, and a transmission delay of 1.6Mb/s.

3.7.3 Modeling batteries

Create new batteries with the `NEWBATT` command. The following creates a battery with ID 0, and a capacity of 0.1mAh.

```
newbatt                 0 1.0
```

4 Extended Example — Software Radio

This chapter provides another extended example of running an application over the simulator. The application in this case is a software defined radio or *software radio*, which is not too trivially partitioned to execute over multiple devices.

4.1 OVERVIEW

The software radio application (henceforth, *swradio*), is partitioned into 5 components—*Source*, *LPF*, *Demod*, *EQ* and *Sink*— as shown in Figure 4.1(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.

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 4.1(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 send it, round-robin, to each of the 8 different instances of the *EQ* stage, running on 8 different processors. In

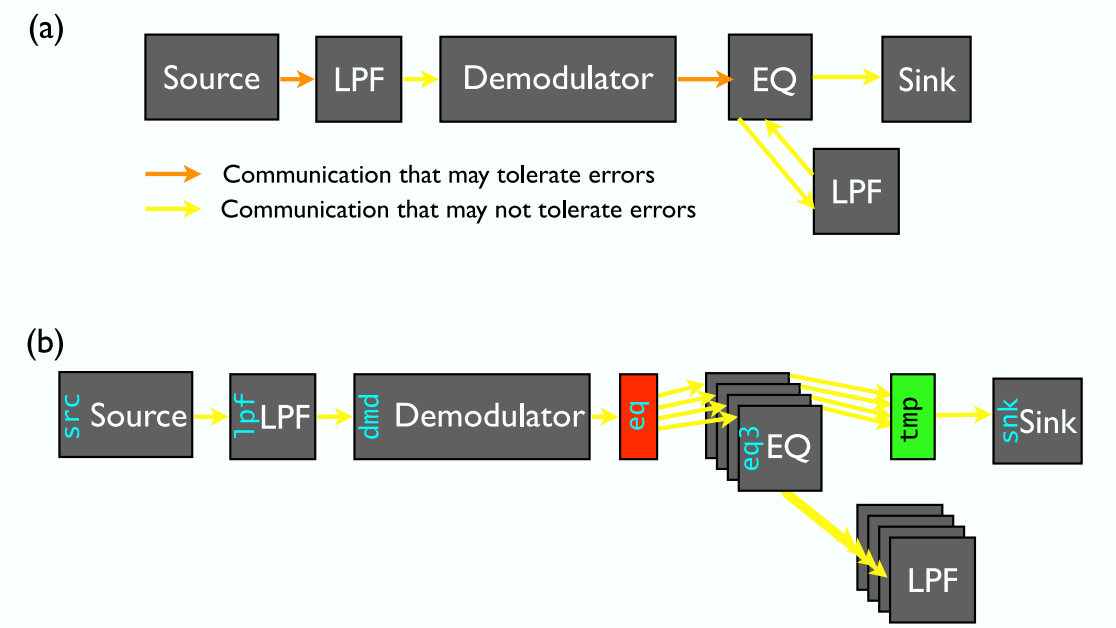

Fig. 4.1 Software radio application, showing computation stages ((a), top), with further partitioning of the EQ stage, ((b), bottom)

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 `sim/superH/BENCHMARKS/c/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 §3.2.3.

4.2 QUICKSTART GUIDE

A quick sequence of steps to get you up and running (once you have everything compiled):

```
cd $ROOT/benchmarks/source/misc/swradio
../../../../sim/sf    # start simulator
```

At the simulator command prompt, type

```
load test.m
on
```

(note that there are several .sr files in the swradio directory, which are copied from the subdirectories. If you change the swradio application and recompile, remember to copy *.sr from the subdirectories into the directory containing test.m)

You should see a few things printed out in color. After a pause of maybe 3 minutes or so, you will see several messages after the processors in the system start exchanging messages. You can hit 'return' to get the simulator command prompt back, with the simulation happeneing in the background.

A good thing to try out when the simulator starts is:

```
help
```

which will list a subset of the simulator commands (see manual appendix B for full list). You can type:

```
man xyz
```

to get more information on a particular command (named xyz, for example).

While the simulator is running, you can pause it by typing:

```
off
```

You can do things like probe machine state etc. (you can actually do this when the simulation is running, but the output will get intermingled with the simulation informatory messages.)

Appendix A Source files

A.1 IMPLEMENTATION OVERVIEW

This appendix provides brief descriptions for all the files in the main simulator implementation directory (`sim/superH/`). Unless otherwise stated, all file names in this appendix are relative to the `sim/superH/` directory.

The simulator models each processing element with a structure, the `State` structure, defined in `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. All the instantiated processors in the simulation are accessible through the global `SIM_STATE_PTRS[]`, 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 `SIM_STATE_PTRS[]`. 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 [5] uses this).

A.1.1 LICENSE.txt

This contains the terms of distribution for the simulator.

A.1.2 all.h

Almost all the source files include this. Although it is generally a bad idea 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 `all.h` and any other specific needs.

A.1.3 batt-test.c

This 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.

A.1.4 batt.c

This implements a discrete-time battery model based on [1]. 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`.

A.1.5 batt.h

This defines the various structures and constants used by the battery model.

A.1.6 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. These two files define different versions of the structures for Big-endian and Little-endian host machines, respectively.

A.1.7 bit-utils.c

This implements routines for doing fast bit counting, as well as some bit display routines. Its actual implementation lies in `sys/include/bit-utils.inc`.

A.1.8 cache.c

This 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.

A.1.9 cache.h

This defines the structures relating to the cache, such as the `Cache` structure, which in turn uses the `Block` structure.

A.1.10 decode-hitachi-sh.c

This 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`.

A.1.11 decode-hitachi-sh.h

Contains some definitions used by the instruction decode implementation for the Hitachi SH.

A.1.12 decode-ti-msp430.h

Instruction decode definitions for TI MSP430. Not distributed / empty in the distribution. This is in the process of being implemented.

A.1.13 dev7708.c

This 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.

A.1.14 dev7708.h

Relevant definitions for the implementation of the memory-mapped registers in Hitachi SH3 SH7708.

A.1.15 devF11x.c

This 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.

A.1.16 devF11x.h

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.

A.1.17 dynsched.h

Header files for the out-of-order machine implementation. This is a vestige of an abandoned attempt. Disregard.

A.1.18 endian-hitachi-sh.h

This includes the appropriate headers based on the host machine endianness defined by `MYRMIGKI_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).

A.1.19 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.

A.1.20 fault.h

Relevant definitions for the fault modeling in `fault.c()`.

A.1.21 fns.h

This 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 `all.h`.

A.1.22 instr-hitachi-sh.h

Instruction format definitions for the Hitachi SH3.

A.1.23 lex.c

This 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.

A.1.24 log.h

This contains a solitary constant definition.

A.1.25 machine-hitachi-sh.c, 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.

A.1.26 machine-ti-msp430.h

Relevant machine-specific definitions for the TI MSP430. Not distributed/empty. This is in the process of being implemented.

A.1.27 main.c

This 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.

A.1.28 main.h

This 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.

A.1.29 mem.h

This contains various constant definitions related to the the size and modeling of the memory system. It contains the definition of the default memory size, `DEFLT_MEMSIZE`.

A.1.30 mkhelp

This script parses the file `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.

A.1.31 mkmantex

This script parses the file `lex.c` (as hinted at previously) to generate LaTeX source for inclusion in documentation such as this manual.

A.1.32 mkopstr-hitachi-sh

This script 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.

A.1.33 mkopstr-ti-msp430

This script 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.

A.1.34 network.c

This 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.

A.1.35 network.h

This 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.

A.1.36 op-hitachi-sh.c

This does the hard work of instruction execution for the Hitachi SH architecture.

A.1.37 op-hitachi-sh.h

All the definitions specific to `op-hitachi-sh.c` are here.

A.1.38 op-ti-msp430.c

This does the hard work of instruction execution for the TI MSP430 architecture.

A.1.39 op-ti-msp430.h

All the definitions specific to `op-ti-msp430.c` are here.

A.1.40 pau.c

This implements the Power Adaptation Unit (PAU) [5], which exploits the mismatch between CPU and memory system performance to reduce energy dissipation, via dynamic voltage scaling.

A.1.41 pau.h

The definitions needed by `pau.c` are here.

A.1.42 pic.c

This implements queued interrupts. The idea is that it is a form of a programmable interrupt controller.

A.1.43 pic.h

The definitions needed by `pic.c` are here.

A.1.44 pipeline-hitachi-sh.c

This 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.

A.1.45 pipeline-hitachi-sh.h

This defines the structures and constants used by `pipeline-hitachi-sh.c`, such as the `Pipe` and `Pipestage` structures.

A.1.46 power.h

This defines all the structures supporting the simulators power modeling.

A.1.47 regs-hitachi-sh.h

Various definitions pertinent to the modeling of machine registers on the Hitachi SH.

A.1.48 regs-ti-msp430.h

Various definitions pertinent to the modeling of machine registers on the TI MSP430.

A.1.49 shasm.y

This 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.

A.1.50 syscalls.c

This implements 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.

A.1.51 syscalls.h

Definitions for the system call trap values.

A.1.52 tag.h

The simulator implements per-processor "tag memory", *a la* Smart Messages [3].

A.1.53 termcolor.c

Colored text on the terminal.

A.1.54 termcolor.h

Support definitions for colored text on the terminal.

A.1.55 timer-hitachi-sh.h

Defines various constants related to the modeling of the Hitachi SH timer unit.

A.1.56 topology.h

Various definitions related to the modeling of space/distance.

Appendix B Commands

B.1 SIMULATOR COMMAND SET

B.1.1 ADDVALUETRACE

Description: Install an address monitor to track data values.

Synopsis:

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

B.1.2 BATTALERTFRAC

Description: Set battery alert level fraction.

Synopsis:

```
BATTALERTFRAC
```

B.1.3 BATTCF

Description: Set Battery Vrate lowpass filter capacitance.

Synopsis:

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

B.1.4 BATTETALUT

Description: Set Battery etaLUT value.

Synopsis:

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

B.1.5 BATTETALUTNENTRIES

Description: Set number of etaLUT entries.

Synopsis:

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

B.1.6 BATTINOMINAL

Description: Set Battery Inominal.

Synopsis:

```
BATTINOMINAL   <Inominal in Amperes>
```

B.1.7 BATTLEAKCURRENT

Description: Set battery leakage current (default is 1uA).

Synopsis:

```
BATTLEAKCURRENT
```

B.1.8 BATTNODEATTACH

Description: Attach current node to a specified battery.

Synopsis:

```
BATTNODEATTACH   <which battery>
```

B.1.9 BATTRF

Description: Set Battery Vrate lowpass filter resistance.

Synopsis:

```
BATTRF   <Resistance in Ohms>
```

B.1.10 BATTSTATS

Description: Get battery statistics.

Synopsis:

```
BATTSTATS   <which battery>
```

B.1.11 BATTVBATTLUT

Description: Set Battery VbattLUT value.

Synopsis:

```
BATTVBATTLUT   <index> <value>
```

B.1.12 BATTVBATTLUTNENTRIES

Description: Set number of VbattLUT entries.

Synopsis:

```
BATTVBATTLUTNENTRIES   <number of entries>
```

B.1.13 BATTVLOSTLUT

Description: Set Battery VlostLUT value.

Synopsis:

```
BATTVLOSTLUT   <index> <value>
```

B.1.14 BATTVLOSTLUTNENTRIES

Description: Set number of VlostLUT entries.

Synopsis:

```
BATTVLOSTLUTNENTRIES   <number of entries>
```

B.1.15 C

Description: Synonym for CACHESTATS.

Synopsis:

```
C
```

B.1.16 CA

Description: Set simulator in cycle-accurate mode.

Synopsis:

```
CA
```

B.1.17 CACHEINIT

Description: Initialise cache.

Synopsis:

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

B.1.18 CACHEOFF

Description: Deactivate cache.

Synopsis:

```
CACHEOFF
```

B.1.19 CACHESTATS

Description: Retrieve cache access statistics.

Synopsis:

```
CACHESTATS
```

B.1.20 CD

Description: Change current working directory.

Synopsis:

```
CD   <path>
```

B.1.21 CLOCKINTR

Description: Toggle enabling clock interrupts.

Synopsis:

```
CLOCKINTR   <0/1>
```

B.1.22 CONT

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

Synopsis:

```
CONT   <until PC>
```

B.1.23 D

Description: Synonym for DUMPALL.

Synopsis:

```
D   <filename>
```

B.1.24 DELVALUETRACE

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

Synopsis:

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

B.1.25 DUMPALL

Description: Dump the State structure info for all nodes to the file.

Synopsis:

```
DUMPALL   <filename>
```

B.1.26 DUMPMEM

Description: Show contents of memory.

Synopsis:

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

B.1.27 DUMPPIPE

Description: Show the contents of the pipeline stages.

Synopsis:

```
DUMPPIPE
```

B.1.28 DUMPREGS

Description: Show the contents of the general purpose registers.

Synopsis:

```
DUMPREGS
```

B.1.29 DUMPSYSREGS

Description: Show the contents of the system registers.

Synopsis:

```
DUMPSYSREGS
```

B.1.30 DUMPTLB

Description: Display all TLB entries.

Synopsis:

```
DUMPTLB
```

B.1.31 DYNINSTR

Description: Display number of instructions executed.

Synopsis:

```
DYNINSTR
```

B.1.32 EBATTINTR

Description: Toggle enable low battery level interrupts.

Synopsis:

```
EBATTINTR   <0/1>
```

B.1.33 EFAULTS

Description: Enable interuppt when too many faults occur.

Synopsis:

```
EFAULTS
```

B.1.34 FF

Description: Set simulator in fast functional mode.

Synopsis:

```
FF
```

B.1.35 FILE2NETSEG

Description: Connect file to netseg.

Synopsis:

```
FILE2NETSEG   <file><netseg>
```

B.1.36 FLTTHRESH

Description: Set threashold for EFAULTS.

Synopsis:

```
FLTTHRESH   <threshold>
```

B.1.37 FORCEAVGPWR

Description: Bypass ILPA analysis and set avg pwr consumption.

Synopsis:

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

B.1.38 HELP

Description: Print list of commands.

Synopsis:

```
HELP
```

B.1.39 IGN

Description: Ignore node fatalities and continue sim without pausing.

Synopsis:

```
IGN   <0 or 1>
```

B.1.40 L

Description: Synonym for LOCSTATS.

Synopsis:

```
L
```

B.1.41 LOAD

Description: Load a script file.

Synopsis:

```
LOAD   <filename>
```

B.1.42 LOCSTATS

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

Synopsis:

```
LOCSTATS
```

B.1.43 MALLOCDEBUG

Description: Display malloc stats.

Synopsis:

```
MALLOCDEBUG
```

B.1.44 MAN

Description: Print synopsis for command usage.

Synopsis:

```
MAN   <command name>
```

B.1.45 MMAP

Description: Map memory of one simulated node into another.

Synopsis:

```
MMAP   <source> <destination>
```

B.1.46 N

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

Synopsis:

```
N   [# cycles]
```

B.1.47 NANOPAUSE

Description: Pause the simulation for arg nanoseconds.

Synopsis:

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

B.1.48 ND

Description: Synonym for NETDEBUG.

Synopsis:

```
ND
```

B.1.49 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>
```

B.1.50 NETDEBUG

Description: Show debugging information about the simulated network interface.

Synopsis:

```
NETDEBUG
```

B.1.51 NETNEWSEG

Description: Add a new network segment to simulation.

Synopsis:

```
NETNEWSEG   <which (if exists)><frame bits><propagation speed><bitrate><medium width><link failure probability distributio
```

B.1.52 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)><fail distribution><fail mu><fail si
```

B.1.53 NETSEG2FILE

Description: Connect netseg to file.

Synopsis:

```
NETSEG2FILE   <netseg> <file>
```

B.1.54 NETSEGDELETE

Description: Disable a specified network segment.

Synopsis:

```
NETSEGDELETE   <which segment>
```

B.1.55 NETSEGFAILDURMAX

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

Synopsis:

```
NETSEGFAILDURMAX   <duration>
```

B.1.56 NETSEGFAILPROB

Description: Set probability of failure for a setseg.

Synopsis:

```
NETSEGFAILPROB   <which segment> <probability>
```

B.1.57 NETSEGFAILPROBFN

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

Synopsis:

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

B.1.58 NETSEGNICATTACH

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

Synopsis:

```
NETSEGNICATTACH   <which IFC><which segment>
```

B.1.59 NETSEGPROPMODEL

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

Synopsis:

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

B.1.60 NEWBATT

Description: New battery

Synopsis:

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

B.1.61 NEWNODE

Description: Create a new node (Simulated system).

Synopsis:

```
NEWNODE   <type=superH|msp430> <x> <y> <z> <speed> <orbit type>
```

B.1.62 NI

Description: Synonym for DYNINSTR.

Synopsis:

```
NI
```

B.1.63 NODEFAILDURMAX

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

Synopsis:

```
NODEFAILDURMAX    <duration>
```

B.1.64 NODEFAILPROB

Description: Set probability of failure for current node.

Synopsis:

```
NODEFAILPROB   <probability>
```

B.1.65 NODEFAILPROBFN

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

Synopsis:

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

B.1.66 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
```

B.1.67 NUMASETMAPID

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

Synopsis:

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

B.1.68 NUMASTATS

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

Synopsis:

```
NUMASTATS
```

B.1.69 NUMASTATSALL

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

Synopsis:

```
NUMASTATSALL
```

B.1.70 OFF

Description: Turn the simulator off.

Synopsis:

```
OFF
```

B.1.71 ON

Description: Turn the simulator on.

Synopsis:

```
ON
```

B.1.72 PARSEOBJDUMP

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

Synopsis:

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

B.1.73 PAUINFO

Description: Show information about all valid PAU entries.

Synopsis:

```
PAUINFO
```

B.1.74 PAUSE

Description: Pause the simulation for arg seconds.

Synopsis:

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

B.1.75 PD

Description: Disable simulation of processor's pipeline.

Synopsis:

```
PD
```

B.1.76 PE

Description: Enable simulation of processor's pipeline.

Synopsis:

```
PE
```

B.1.77 PF

Description: Flush the pipeline.

Synopsis:

```
PF
```

B.1.78 PFUN

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

Synopsis:

```
PFUN
```

B.1.79 PI

Description: Synonym for PAUINFO.

Synopsis:

```
PI
```

B.1.80 POWERSTATS

Description: Show estimated energy and circuit activity.

Synopsis:

```
POWERSTATS
```

B.1.81 POWERTOTAL

Description: Print total power accross all node.

Synopsis:

```
POWERTOTAL
```

B.1.82 PS

Description: Synonym for POWERSTATS.

Synopsis:

```
PS
```

B.1.83 PWD

Description: Get current working directory.

Synopsis:

```
PWD
```

B.1.84 Q

Description: Synonym for QUIT.

Synopsis:

```
Q
```

B.1.85 QUIT

Description: Exit the simulator.

Synopsis:

```
QUIT
```

B.1.86 R

Description: Synonym for RATIO.

Synopsis:

```
R   <>
```

B.1.87 RATIO

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

Synopsis:

```
RATIO
```

B.1.88 REGISTERSTABS

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

Synopsis:

```
REGISTERSTABS   <STABS filename>
```

B.1.89 RENUMBERNODES

Description: Renumber nodes based on base node ID.

Synopsis:

```
RENUMBERNODES
```

B.1.90 RESETALLCTRS

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

Synopsis:

```
RESETALLCTRS
```

B.1.91 RESETCPU

Description: Reset entire simulated CPU state.

Synopsis:

```
RESETCPU
```

B.1.92 RESETNODECTRS

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

Synopsis:

```
RESETNODECTRS
```

B.1.93 RETRYALG

Description: set NIC retransmission backoff algorithm.

Synopsis:

```
RETRYALG   <ifc #> <algname>
```

B.1.94 RUN

Description: Mark a node as runnable.

Synopsis:

```
RUN
```

B.1.95 SAVE

Description: Dump memory region to disk.

Synopsis:

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

B.1.96 SENSORSDEBUG

Description: Display various statistics on sensors and signals.

Synopsis:

```
SENSORSDEBUG
```

B.1.97 SETBASENODEID

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

Synopsis:

```
SETBASENODEID   <integer>
```

B.1.98 SETBATT

Description: Set current battery.

Synopsis:

```
SETBATT   <Battery ID>
```

B.1.99 SETBATTFEEDPERIOD

Description: Set update periodicity for battery simulation.

Synopsis:

```
SETBATTFEEDPERIOD   <period in microseconds>
```

B.1.100 SETDUMPPWRPERIOD

Description: Set periodicity power logging to simlog.

Synopsis:

```
SETDUMPPWRPERIOD   <period in microseconds>
```

B.1.101 SETFREQ

Description: Set operating frequency from voltage.

Synopsis:

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

B.1.102 SETIFCOUI

Description: Set OUI for current IFC.

Synopsis:

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

B.1.103 SETNODE

Description: Set the current simulated node.

Synopsis:

```
SETNODE   <node id>
```

B.1.104 SETPC

Description: Set the value of the program counter.

Synopsis:

```
SETPC   <PC value>
```

B.1.105 SETPHYSICSPERIOD

Description: Set update periodicity for physical phenomenon simulation.

Synopsis:

```
SETPHYSICSPERIOD   <period in microseconds>
```

B.1.106 SETQUANTUM

Description: Set simulation instruction group quantum.

Synopsis:

```
SETQUANTUM   <integer>
```

B.1.107 SETSCALEALPHA

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

Synopsis:

```
SETSCALEALPHA   <double>
```

B.1.108 SETSCALEK

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

Synopsis:

```
SETSCALEK   <double>
```

B.1.109 SETSCALEVT

Description: Set technology Vt for use in voltage scaling.

Synopsis:

```
SETSCALEVT   <double>
```

B.1.110 SETTIMERDELAY

Description: Change granularity of timer intrs.

Synopsis:

```
SETTIMERDELAY   <granularity in microseconds>
```

B.1.111 SETVDD

Description: Set operating voltage from frequency.

Synopsis:

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

B.1.112 SFATAL

Description: Induce a node death and state dump.

Synopsis:

```
SFATAL   <suicide note>
```

B.1.113 SHAREBUS

Description: Share bus structure with ther named node.

Synopsis:

```
SHAREBUS   <Bus donor nodeid>
```

B.1.114 SHOWCLK

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

Synopsis:

```
SHOWCLK
```

B.1.115 SHOWPIPE

Description: Show contents of the processor pipeline.

Synopsis:

```
SHOWPIPE
```

B.1.116 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>
```

B.1.117 SIGSUBSCRIBE

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

Synopsis:

```
SIGSUBSCRIBE   <X> <Y>
```

B.1.118 SIZEMEM

Description: Set the size of memory.

Synopsis:

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

B.1.119 SPLIT

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

Synopsis:

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

B.1.120 SRECL

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

Synopsis:

```
SRECL
```

B.1.121 STOP

Description: Mark the current node as unrunnable.

Synopsis:

```
STOP
```

B.1.122 THROTTLE

Description: Set the throttling delay in usecs.

Synopsis:

```
THROTTLE   <throttle delay in usecs>
```

B.1.123 THROTTLEWIN

Description: Set the throttling window clock cycles.

Synopsis:

```
THROTTLEWIN   <throttle window period in clock cycles>
```

B.1.124 TRACE

Description: Toggle Tracing.

Synopsis:

```
TRACE
```

B.1.125 V

Description: Synonym for VERBOSE.

Synopsis:

```
V
```

B.1.126 VALUESTATS

Description: Print data value tracking statistics.

Synopsis:

```
VALUESTATS
```

B.1.127 VERBOSE

Description: Enable the various prints.

Synopsis:

```
VERBOSE
```

B.1.128 VERSION

Description: Display the simulator version and build.

Synopsis:

```
VERSION
```

B.1.129 NODETACH

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

Synopsis:

```
NODETACH   <0 or 1>
```

B.1.130 SIZEPAU

Description: Set the size of the PAU.

Synopsis:

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

Index

References

1. L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. In *Proceedings of the conference on Design, automation and test in Europe*, pages 35–39, January 2000.

2. A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. SH3: High Code Density, Low Power. *IEEE Micro*, 15(6):11–19, December 1995.

3. P. Stanley-Marbell, C. Borcea, K. Nagaraja, and L. Iftode. Smart Messages : A System Architecture for Large Networks of Embedded Systems. In *8th Workshop on Hot Topics in Operating Systems, HOTOS-VIII*, page 153, May 2000.

4. P. Stanley-Marbell and M. Hsiao. Fast, flexible, cycle-accurxate energy estimation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 141–146, August 2001.

5. P. Stanley-Marbell, M. S. Hsiao, and U. Kremer. A Hardware Architecture for Dynamic Performance and Energy Adaptation. *Lecture Notes in Computer Science, Springer-Verlag*, 2325(1):33–52, 2002.

6. V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A first Step Towards Software Power Estimation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 384–390, August 1994.