University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

SEMi: A Simulator for EARTH, MANNA, and i860 (Version 0.23)

Kevin B. Theobald

CAPSL Technical Memo 27
March 1, 1999

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

SEMi (Simulator for EARTH, MANNA and the i860) is a modular package designed to simulate various machines based on the Intel i860. SEMi simulates cycle-by-cycle, making it highly accurate yet fast (for a simulator). The current configuration includes modules for accurate simulation of MANNA-based external hardware (main memory, I/O, link and crossbar chips), enabling real applications written for the MANNA or i860 to be run on SEMi. A rich set of command line options can be used to modify processor characteristics (e.g., change cache sizes), enabling the study of the effects of these parameters. Many runtime commands are implemented, including a powerful and versatile breakpoint system which allows complex commands to be triggered by the code, yet executed "outside" of the code.

This manual is a user's guide to SEMi. It begins with a description of the major characteristics of the SEMi model, their features and limitations. Several simple tutorials allow the reader to become familiar with the user interface to SEMi. The remaining sections list and describe SEMi's options and user commands in detail.

NOTE: this user's manual is intended to cover general aspects of SEMi. There may also be a "Local Guide" which describes SEMi-related information which is specific to one site, such as the location of SEMi and compilers which generate SEMi code.

Contents

List of Figures

List of Tables

SEMi: Half as good as the real thing... at a fraction of the cost!

1 Introduction

SEMi is a modularized simulator package designed to simulate various i860-based machines. It was originally inspired by DLXsim [3], and uses a similar interface. There are some major differences, however:

1. SEMi simulates the instruction set of a real machine, the Intel i860XP processor [6, 8], and can read regular executable files. This means that it is possible to run code generated by existing compilers (code which includes run-time libraries). Furthermore, the simulator can be run side-by-side with a real machine and the results compared. Therefore, the simulator can be used to study the behavior of the real machine.
2. SEMi carefully models actual processor behavior in terms of cycle counts, and accounts for resource hazards and load stalls. The i860 core simulator is augmented by a bus simulator to produce accurate cycle counts.
3. SEMi is modularized, separating the behavior of internal and external components so that new configurations can be made easily.

SEMi performs simulation cycle-by-cycle. During each cycle, SEMi simulates the CPU for one cycle, and then simulates the bus and memory system for one cycle. While this method of simulation is slower, it leads to more accurate timing results.

SEMi has an interactive command interface. This allows the user to step through the code while examining or modifying state information. SEMi automatically collects statistics on instruction counts and stalls. In addition, a versatile breakpoint feature coupled with meta-instructions enables the user to program customized statistics-gathering routines which do not interfere with the timing of the simulated code. (In contrast, gathering statistics in a real machine would require either adding extra code to the original program, which would affect the timing behavior of the code, or would require special external hardware.)

This document has five main sections. The next section describes the execution model assumed by SEMi, and discusses some of the limitations of SEMi. Section 3 is a tutorial introducing the basic commands of SEMi using sample sessions. Section 4 describes the command for invoking SEMi and the format of the command line. Section 5 covers interaction with SEMi once it has been started. These three sections cover general features that are common to all versions of SEMi. The final section describes SEMiMANNA, a version of SEMi configured for a multiprocessor with dual-CPU nodes. This section only covers the additional features in SEMiMANNA, so the reader should first read the previous sections.

2 Execution Model

SEMi assumes a machine with a single i860XP processor [6, 8] with on-chip caches and a contiguous block of external physical memory. There are special memory-mapped addresses for I/O and external control hardware. SEMi also provides the user with "meta-memory," which is not part of the regular memory but can be accessed through special SEMi commands.

	Core operation	Memory operation
Stage 1	Instruction fetch	Instruction fetch
2	Instruction decode	Instruction decode
3	Operation execution	Address calculation
4	Result writeback	Cache access
5		Result writeback

Figure 1: Execution Pipeline of the i860

2.1 Simulating the i860XP Processor

The core of SEMi is a simulator for the i860XP microprocessor, built to achieve 1) correct behavior of any legal program run on the simulator, 2) accurate processor statistics, and 3) high speed. To achieve the last goal, only those elements considered essential to the first two goals are simulated.

2.1.1 Execution Pipeline

The i860 execution pipeline [8] has 4 stages (5 for loads and stores) as shown in Figure 1. Since SEMi is intended primarily as a "functional" simulator, the real behavior of individual stages of the pipeline are not simulated. SEMi only simulates the details needed to model the interactions between stages that affect timing. For instance, SEMi models the fact that instructions are fetched 2 cycles before they are executed. This is necessary for the correct modeling of delay slots and bus interactions. The order of operations in a pipeline affects the latter because bus cycles can be generated from both the fetch stage (I-cache miss) and the cache load stage (D-cache miss).

SEMi can display three stages of the "pipeline" to the user: the **M**emory stage, e**X**ecute stage, and **D**ecode stage. The M stage contains the effective address of a load or store, computed during the execute stage. The X stage contains the operation about to be performed. The D stage contains the instruction that follows. The write-back stage isn't modeled since it doesn't affect the timing behavior (due to bypass logic). A stall in one stage blocks all stages behind it in the list, but not ahead.

2.1.2 Registers

SEMi supports all the standard registers in the i860: core registers (r0–r31), floating point registers (f0–f31), special registers (fir, psr, dirbase, db, fsr, epsr, bear, ccr, p0–p3), and the KI, KR, T and MERGE registers. Many of the special registers are read-only. Some are irrelevant; for instance, the bear (bus error address) is always 0 because bus errors never occur in the simulator. However, they are included so that programs which access them will still execute. Others have only partial functionality.

In addition, SEMi has some *meta*-registers. These are provided to give the user access to internal processor states, though they can't be directly accessed by regular code. The meta-registers are listed in Table 1.

The A, M, and G registers give the user access to the stages of the pipelines. They are only meant for pipelined ops and won't show anything meaningful during scalar FP ops. When SEMi simulates a pipelined operation, it computes the result immediately and places it in stage 1 of the pipeline, where it propagates unchanged down the pipeline until being written from the output stage to a destination register. The output stages are A3 (adder), M3 (multiplier), and G1 (graphics). Thus, the user can see

Mnemonic	Description
PC	Program Counter
NPC	Next Program Counter
CLK	Clock (no. of cycles executed)
NID	Node ID
PID	Processor ID
A1–A3	Floating-point adder pipeline
M1–M3	Floating-point multiplier pipeline
G1	Floating-point graphics pipeline
L$m.n$	n^{th} word of m^{th} stage of load pipeline ($1 \leq m \leq 3, 0 \leq n \leq 3$)
*reg*HI	High 32-bits of A, M, G and CLK registers

Table 1: Meta-Registers in the i860

which operation is in each stage of the pipeline, but SEMi will not display the intermediate results that would exist in a real CPU. Stage M2 is bypassed by double-precision multiplies.

L registers show the state of loads from the PFLD op. A PFLD always directs its data to L1, but later PFLD ops shift this destination to later stages. A PFLD always loads its destination register(s) from L3. If an L stage is displayed, the contents displayed are valid only if the load has completed.

All pipeline meta-registers, as well as the K, T and MERGE registers, can hold 64-bit values. For each, the mnemonic can refer to either the entire 64-bit contents or just the lower 32 bits, depending on the access mode. It is also possible to refer to the upper 32 bits separately by appending "HI" to the register name (e.g., KRHI, A2HI). L registers can hold up to 4 words, which can be referenced individually using extensions from 0–3, e.g., L1.2, L3.0. SEMi keeps track of how many words in a pipeline stage are actually used, and when it displays pipeline stages, it will ignore words not currently being used.

SEMi measures time in cycle counts, and this time is kept in the CLK register. One can convert this to real time by dividing it by the clock speed, i.e., 50 million. CLK is a double register, so CLKHI refers explicitly to the high 32 bits.

The Procesor ID and Node ID are read-only and hardwired to values unique to each CPU in the simulation. Both are 0 on uniprocessor configurations, but may take on other values for multiprocessor simulations.

2.1.3 Virtual Memory and Cache

The i860's MMU is partially supported. The first-level page table (whose physical address is given in the dirbase register) can contain entries for 4-megabyte pages, but not 4-kilobyte pages. Only the WT (Write-Through) and CD (Cache Disable) bits are relevant; the others are ignored. There is no support for disk memory; all valid virtual addresses should be mapped to valid physical addresses.

Aliasing between virtual and physical tags is supported (to allow for such situations as two processes sharing a buffer), as is the MESI protocol (relevant mainly with multiprocessor configurations of SEMi). The default cache writeback policy (the policy used if the WT bit in the page table entry is not set) is selectable in the command line. System code that modifies the page table and/or dirbase is supposed to flush or invalidate the caches to prevent inconsistent cache states. Code which fails to do this properly may not behave in SEMi the same as in a real i860.

	CATEGORY	CAUSE
Pipeline bubbles	Instruction fetch	Waiting to fetch instruction after I-cache miss
	Multi-cycle operation	Extra cycles required by core ops such as BRI
	Delay slot	Bubble due to non-executed delay slot
Dependences	Core dependence	Flow dependence between core registers
	FP dependence	Like core dependence stalls, but for FP
	Miss	Stalls waiting for loads from external memory
Hazards	FP hazard	Waiting for previous scalar FP op to finish
	Cache hazard	Contention for D-cache due to miss load in progress
	Bus read	Stall because load buffer is full
	Bus write	Stall because store buffer is full

Table 2: Stall Classifications

2.1.4 Hazards and Stalls

SEMi models hazards that prevent execution of an instruction, including data hazards and resource conflicts. If SEMi executes an instruction that requires more than one cycle (such as a scalar FP op or a load from memory), it will perform the operation immediately and write the result to the destination register. However, it will remember that the operation will not "really" be finished until later, and any operation that reads that register too early will stall. Thus, if the user steps through the code cycle-by-cycle, it will appear that all operations complete in one cycle, but certain operations will stall the pipeline. (Reads that go off the chip are an exception; the value is not stored in the destination register until the bus access is completed.)

For every cycle in which the CPU stalls, the stall is classified and tallied. Every stall falls into one of the categories shown in Table 2.

In general, there are three types of stalls: *pipeline bubbles, dependences,* and *hazards.* Pipeline bubbles affect the execution pipe. Instruction fetch stalls are cycles lost while waiting to fetch an instruction that missed the I-cache. Multi-cycle stalls are caused by certain ops that require more than one cycle, such as BRI. Delay slot stalls are bubbles caused by the instructions BC, BNC, BTE and BTNE, which do not have true delay slots, but take additional cycles in some cases. Since at least one of these extra cycles is due to flushing the next instruction from the pipeline, SEMi counts the stall as a "delay slot" stall rather than a "multi-cycle" stall. Instructions BC.T and BNC.T cause a delay slot if they don't take the branch, since they then cancel the instruction in the delay slot. Both BTE and BTNE take two extra cycles; one is counted as a "delay slot" stall and the other as a "multi-cycle" stall.

Dependences are stalls in which one operation needs an operand which has not yet been produced. (Unlike pipeline bubbles, dependences can theoretically be avoided by rearranging code.) Core dependence stalls occur when one op reads a core register produced by a previous operation which has not completed (e.g., a conditional branch following ADDU or ADDS). This includes the 1-cycle stall between a load which hits the cache and an operation which uses that register in the next instruction, but does *not* count stalls due to cache misses, which are counted separately. FP dependence stalls are caused by analogous dependences between FP registers. Miss stalls occur when a register is used in an operation but the operation stalls because the register is being loaded from external memory (following a D-cache miss, or because the load is from I/O).

A hazard occurs when one operation needs resources being used by another operation, even if it doesn't need *data* produced by the other. FP hazards count contention for the floating point pipeline, e.g., when one scalar FP op follows another. A cache hazard means a load or store operation can't check the D-cache because a previous operation is still using it (e.g., a cache miss load). Bus read and write stalls occur when a load, store or fetch can't put an address onto the bus because the bus pipeline is full.

In some cases, the CPU can stall for more than one reason. For example, the instruction **st.l f3, 0(r1)** could stall if either f3 is in the floating point pipeline or if r1 is being loaded from the bus, and both could occur at the same time. In such cases, SEMi will classify the stall in one category only. Thus, the total number of stalls counted by SEMi should obey the equation

(ops − dual-instruction-mode ops) + stalls = total cycles

2.1.5 Limitations

The current simulator has limitations in functionality. The following lists some of the main limitations.

Operations: The following operations are not simulated: pst.d, intovr, ldio, stio, ldint, scyc, pfmul3.dd. The last is for dealing with FP traps, which are not generated in SEMi.

Software traps (the trap instruction) do not cause execution to jump to the trap handler. Certain traps are hardwired into SEMi, as discussed in Section 2.3.

Hardware Traps and Illegal Code: Currently, hardware traps are not modeled. However, many operations which would cause a hardware trap in the real machine will cause the simulation to halt with an error message. Many status bits which are used by trap handlers in order to restore the processor state following an exception are consequently not used.

Certain combinations of instructions are illegal and will cause hardware traps. In some cases, there is a sensible interpretation of the combination, and it is only for implementation reasons that it is disallowed. In such a case, SEMi will generally perform the operation anyway, rather than checking for legality. In other cases, the combination can't be sensibly interpreted, in which case the simulator generally halts with a warning.

The db (data breakpoint) register, which is used for setting a breakpoint in the hardware, is not implemented in SEMi, though the register can be read and written using ld.c and st.c. SEMi has its own breakpoint system, described in Section 2.4.

Virtual Memory: As mentioned previously, address translation only supports 4-megabyte pages. Currently, no TLB behavior is modeled. It is assumed that there are no TLB misses.

User and Supervisor Modes: SEMi does not distinguish between User and Supervisor modes, and all instructions can work in either mode.

2.2 External Memory and I/O

Physical memory is assumed to be a contiguous block of memory whose size (in bytes) is a power of 2. Its starting address must be on a 4K boundary (the lower 12 bits are 0), but does not have to be aligned with the total memory size. For instance, one could specify a 1-megabyte (2^{20}-byte) block starting at address

r31	System call	Returns	Argument types
1	exit	void	(int)
3	read	int	(int, char *, int)
4	write	int	(int, char *, int)
5	open	int	(char *, int, int)
6	close	int	(int)
8	creat	int	(char *, int)
9	link	int	(char *, char *)
10	unlink	int	(char *)
12	chdir	int	(char *)
14	mknod	int	(char *, int, int)
15	chmod	int	(char *, int)
16	chown	int	(char *, int, int)
19	lseek	int	(int, int, int)
20	getpid	int	(void)

r31	System call	Returns	Argument types
22	unmount	int	(char *)
24	getuid	int	(void)
33	access	int	(char *, int)
36	sync	void	(void)
37	kill	int	(int, int)
39	setpgrp	int	(int)
41	dup	int	(int)
47	getegid	int	(void)
51	acct	int	(char *)
60	umask	int	(int)
61	chroot	int	(char *)
79	rmdir	int	(char *)
80	mkdir	int	(char *, int)
81	getdents	int	(int, char *, int)

Table 3: Software Trap System Calls

0xFFFF0000 (i.e., an address range of 0xFFFF0000–0x000EFFFF). The size and base address can be specified in the command line. All addresses outside the physical range are assumed to be potential I/O addresses. See Appendix A for a list of the valid I/O addresses and their functions.

Any access to an address which is not in physical memory and not a valid I/O address, and any access to a misaligned address, will halt simulation with a warning message. Warnings will also occur for writing to an address which was previously executed (the i860 doesn't like self-modifying code), or reading from a page of memory which has not yet been initialized (the last is not, strictly speaking, illegal, but probably indicates a bug in the code).

2.3 Software Traps and POSIX Support

Software traps (the trap instruction) do not cause execution to jump to the trap handler, as they do in the real processor. The instruction **trap r0,r0,r0** causes simulation to halt and control to return to the SEMi user interface.

Disk memory is not simulated in SEMi. However, users can run programs that access files, because SEMi provides support for file I/O through POSIX system calls, which are accessed through trap instructions. If a trap instruction has r31 for the first register, the trap is assumed to be a system call, with the contents of r31 used to identify the specific call. System calls currently supported by SEMi are listed in Table 3 (refer to a POSIX manual or man pages for the meaning of each call and its arguments; these calls conform to SunOS 4.1). The trap handler assumes that integer and pointer arguments are in registers starting with register r16, and returns its result (if any) in r16 (the standard i860 procedure calling conventions).

SEMi does not count cycles during system calls, i.e., these calls take only a single cycle, the cycle to execute the trap instruction.

2.4 Breakpoints and Meta-Memory

SEMi allows the user to set breakpoints at specific memory addresses. When the CPU tries to access an address containing a breakpoint, the CPU will execute one or more commands which are associated with that breakpoint. Such commands can include commands to change or view the state of the machine, or to halt the simulation. The breakpoint can be set up so that only certain types of accesses will trigger the breakpoint: read (reading with a load instruction, not an instruction fetch), write, and/or execute. Breakpoints triggered by an instruction fetch are triggered *before* the instruction is executed; those caused by a read or write are triggered *after* the load or store instruction (but before the instruction gets to the M stage of the pipeline).

To make breakpoints useful for gathering statistics, SEMi provides the user with auxiliary memory called "meta-memory." This memory is not part of the normal architecture, and can only be read or written using user-level commands. The number of bytes can be specified on the command line, but is always a power of 2. The base address is always 0. Meta-memory is accessed in a user-level command by preceding the address with a carat ($\wedge$).

Meta-memory is most useful when used in conjunction with breakpoints, because it can be used to retain run-time information which is placed there at certain breakpoints. For instance, one can program breakpoints to count the number of times a procedure or basic block is entered. The 4-byte word at meta-address 0 is sometimes used by certain SEMi commands to return values, so addresses 0–3 should never be used by the user for storing data.

3 A Brief Tutorial on SEMi

(This tutorial is based on a tutorial created by Louis Hutchinson.)

This tutorial has two parts. The first part starts with an empty machine and inserts and runs a simple assembly language program. The second part loads a compiled COFF executable. (COFF files are an Intel standard for executables.)

3.1 Tutorial 1

The goals for this brief tutorial are:

- To learn to use the following commands: =, get, view, cycle, and run;
- To learn to set and delete breakpoints;
- To learn to change the display format;
- To learn how to view processor statistics.

To start SEMi, type '`semiuni -mb 0 -ms 16`' at the shell prompt. The switch `-mb 0` sets the memory base address to zero. The switch `-ms 16` sets the amount of memory which SEMi will simulate to 2^{16} bytes. Using these two switches will make it easier to put a program into memory and find it later. Now the user should be at the SEMi prompt, which looks like the following:

```
SEMi [t=0]>
```

Once in SEMi, the first step to be taken is to insert a program into memory. The command which does this is the `=` command. This command assigns value to registers or memory locations. The values may be instructions, pointers, numbers, or characters. The following command will insert a simple program into memory. Type

```
= 0 {or 0,r0,r1} {or 10,r0,r2} {addu r1,r2,r1} {adds -1,r2,r2} {bnc 8} {trap r0,r0,r0}
```

This program fragment is written in i860 assembly language. When run, it will add all of the numbers from 10 down to 1. Notice that the assembly instructions are entered in curly braces. This is necessary in SEMi whenever an argument to a command contains whitespace. Also notice the 0 preceeding the other arguments. This is the memory address to which the first instruction in the program will be stored. The remaining instructions will be stored in consecutive words in memory.

In general, use '`=` ***address value ...***' to put one or more values into a series of memory locations or registers starting from *address*. You should only use this method of storing items in memory during debugging, as accurate timing isn't guaranteed when the machine state is changed in this way.

Now that the program is stored in memory, type '`= pc 0`' to set the program counter to the beginning of the program. The program counter is actually a register. The = command can also store values in registers, if a register name is used instead of a memory address. Also, remember that the program was inserted at 0 in memory. A memory address can always be entered as an absolute address, or as a label. A good example of a label would be `main` in a program (which often is turned into `_main` by compilers).

To see that the program is properly loaded into memory and that the program counter is pointing to it, use the `get` command. Type '`get pc 6i`'. This command can only be used for viewing memory, not registers. In such a case, the pointer is followed to the location to which it points. Following 'pc' is the format 6i. The format 6i tells get to display six consecutive instructions. (Six consecutive double floating point numbers can be viewed using 6df instead, provided that double floats are stored at the specified address.) Now the following should appear on the screen:

```
SEMi [t=0]> get pc 6i
0x00000000:   (e4010000) or 0, r0, r1
0x00000004:   (e402000a) or 10, r0, r2
0x00000008:   (80410800) addu r1, r2, r1
0x0000000c:   (9442ffff) adds -1, r2, r2
0x00000010:   (7bfffffd) bnc 0x8
0x00000014:   (44000000) trap r0, r0, r0
SEMi [t=0]>
```

Notice that each line begins with an address, a hexadecimal representation of an instruction, and the instruction mnemonic. In general, use '`get` ***address [format]***'.

Now the program is ready to run. It is possible to run one cycle of a program at a time by typing `cycle`, or a series of cycles by typing '`cycle` ***number-of-cycles***'. Just typing the command without an argument advances the time one cycle:

```
SEMi [t=0]> cycle
SEMi [t=1]>
```

To see the progress of the computation at this point, type either the `view` or `get` command *without any arguments*. Either one will show the state of the execution pipeline:

```
X| 0x00000000: ----------------
D| 0x00000000: FETCH IN PROGRESS
Pending bus transactions: 1
    Line to I-cache from          0 at time 0 (on bus)
```

SEMi shows the eXecute and Decode stages. The X stage is stalled, and the CPU tried to fetch from the current PC address. However, SEMi always starts with a cold cache, so the fetch missed and had to go to external memory at the end of cycle 0.

It is also possible to check on the progress of the program using '`get pc 10i`' to see where the program counter is pointing, although this won't say anything about the state of the execution pipeline.

It takes 9 more cycles for the load to finish, and another cycle to get through the pipeline. Type '`cycle 10`' to get to this point, then type `view`. Observe that the second or instrution is to be executed next in the pipeline.

```
X| 0x00000000: (e4010000) or 0, r0, r1
D| 0x00000004: (e402000a) or 10, r0, r2
Pending bus transactions: 1
    Line to I-cache from          0 at time 0 (on bus)
```

Note the cache line is still being loaded. That's because the line is read in a 4-cycle burst, but instructions could start executing as soon as the first cycle finished. A few more cycles should take care of this. This is a good time to point out that SEMi commands can be abbreviated to the shortest unique prefix, which is usually just the first letter.

```
SEMi [t=11]> c 4
SEMi [t=13]> g
X| 0x00000010: (7bfffffd) bnc +0x8
D| 0x00000014: (44000000) trap r0, r0, r0
Pending bus transactions: 0
```

(From now on, sample commands in this documentation will use full names, but abbreviations may be used. The `?` command prints a full list of commands.)

In general, use '`view` ***value [format]***' for viewing the state of the processor or memory, or evaluating expressions. The value may be an expression, address pointer, register, constant, or mnemonic. Type the following to display the contents of register r1. It was just set to 10 by the addu instruction at address 8.

```
SEMi [t=15]> view r1 n
r1:               10
```

The 'n' format displays the data as a decimal **n**umber. It can be used in combination with 'd' to specify a double long (64-bit) integer (in conjunction with the next register), with 'h' to specify a short (**h**alfword) integer, or with 'c' to specify a byte (**c**har).

The default format is 'x' (hex), which is why 'n' must be typed to see a decimal number. The default can be changed with `format`. Type '`format n`' to set the default to signed decimal numbers, as an alternative to using the format switches in the view command.

```
SEMi [t=15]> view r1
r1:            0xa
SEMi [t=15]> format n
SEMi [t=15]> view r1
r1:              10
```

Set a breakpoint by typing '`break 8`'. Now SEMi will halt every time the program counter reaches the instruction at memory address 8. The `run` command will tell SEMi to simulate indefinitely until it executes a `trap r0,r0,r0` instruction (which is recognized as a halt command by SEMi, although it doesn't have this meaning in the real i860), encounters a breakpoint, or some kind of error occurs. Type `run` and watch the program stop at the breakpoint.

```
SEMi [t=15]> break 8
SEMi [t=15]> run
Halting at breakpoint 1 (executing)
SEMi [t=18]> get
X| 0x00000008: (80410800) addu r1, r2, r1
D| 0x0000000c: (9442ffff) adds -1, r2, r2
Pending bus transactions: 0
SEMi [t=18]> view r1
r1:              10
SEMi [t=18]> run
Halting at breakpoint 1 (executing)
SEMi [t=23]> view r1
r1:              19
```

After one pass through the loop, the sum (accumulated in r1) is 10. After another pass, 9 has been added.

Notice the word "executing" in parentheses. Breakpoints can be set to trigger when executing an instruction at a specific location in memory as it is in this case. Breakpoints can also be set to trigger when reading data from or writing data to a location in memory. This is done using the switches available to the `break` command.

The switches are d, l, r, w, and x. An execution breakpoint is set by x. (This is the default if no switches are used.) A read breakpoint is set by r, and a write by w. The d switch deletes a breakpoint, while the l switch lists a breakpoint. In general, use

`break` ***[-switches] value [command]***

to set breakpoints and

`break` ***-switches value [value]***

to list or delete them. In this second form, giving two values instead of one specifies a range over which the list or deletion applies. A break command with no arguments will list all current breakpoints, while `break` with just a value argument will set an execution breakpoint at the address specified by the value.

In the first form, the optional *command* argument tells SEMi to perform some task when it reaches the specified breakpoint. For instance, instead of manually printing the contents of r1 each time we hit the breakpoint at the beginning of the loop, we could tell SEMi to do this for us. First, we must delete the old breakpoint at address 8:

```
SEMi [t=23]> break -d 8
```

Another way to delete a breakpoint is to use the `delete` command. This command deletes a breakpoint by specifying its number. Note that the breakpoint was called "breakpoint 1" in the output above. So the breakpoint at address 8 could have been deleted with '`delete 1`' as well.[1] Breakpoint numbers appear when breakpoints are listed.

Now set the new breakpoint with a command. The command contains whitespace, so it must be in curly braces. Also, if we want SEMi to halt at address 8, we need to add a `halt` command, which is normally the default action if none is specified. Multiple commands are separated with a semicolon.

```
SEMi [t=23]> break 8 {view r1; halt}
SEMi [t=23]> run
r1:             27
Halting at breakpoint 2 (executing)
SEMi [t=28]>
```

If we redo the breakpoint and leave out the halt command, then typing `run` will finish running the program. The program will halt when it finally exits the loop and reaches the `trap r0,r0,r0` instruction.

```
SEMi [t=28]> delete 2
SEMi [t=28]> break 8 {view r1}
SEMi [t=28]> run
r1:             34
r1:             40
r1:             45
r1:             49
r1:             52
r1:             54
r1:             55
TRAP #0 received
SEMi [t=68]>
```

Type `stats` to receive a summary of the processor statistics collected up to this point. This is a complete summary. Individual tables can be selected with letter arguments. For instance, `stats s` summarizes processor stalls:

```
SEMi [t=68]> stats s
PROCESSOR STATISTICS:
```

[1] Users of gdb may notice a similarity here. There is even a `tbreak` command for temporary breakpoints.

```
STALLS
------
Multi-cycle ops:               0
Core dependences:             11
FP hazards:                    0
FP dependences:                0
Delay slots:                  10
Instruction fetches:          11
D-Cache hazards:               0
D-Cache misses:                0
Bus reads:                     0
Bus writes:                    0
Total:                        32
```

The 11 "instruction fetch" stalls were from the cache miss at the beginning. The loop was executed a total of 11 times. Each time, there was a stall cycle between the add instruction at address 0xc and the conditional branch that followed. These were counted as "core dependences." For all but the last iteration, the branch was taken (back to the start of the loop). The delay slot was skipped, and a bubble put in its place in the pipeline, and this caused the 10 "delay slot" stalls.

To check the branches taken and not taken use `stats b`. Cache hits and misses are listed by following the `stats` with a c. An o calls for a count of all of the operations used. Finally, follow `stats` with a t to view the totals, a shorter summary.

Type `quit` to exit SEMi.

3.2 Tutorial 2

If one wants to run a precompiled COFF file, add the name of the executable, along with any desired command line arguments, to the end of the semiuni command. Include the switch -r to make sure the command line arguments are passed to the executable (so that the main() function can read them with argc and argv). For instance, to run an executable compiled from the following source code:[2]

```
#include <stdio.h>

main (int argc, char **argv) {
printf("Hello %s!\n", argv[1]);
}
```

type the command `semiuni -r hello world`. This loads in the COFF file `hello` and passes the string "world" to it. The starting address is automatically extracted from the COFF file. Be sure to set up the memory so that there is valid memory at that address (in this case, using the defaults is sufficient.)

The COFF file's symbol table is also read. These symbols can be used in SEMi commands. For instance:

[2] In this case, compiled with the command `etcc -O2 -target manna -semi hello.c`.

```
SEMi [t=0]> get _main 5i
0xfe020500:   (8442ffc0) addu -64, r2, r2
0xfe020504:   (1c400801) st.l r1, 0(r2)
0xfe020508:   (1c408835) st.l r17, 52(r2)
0xfe02050c:   (145c0035) ld.l 52(r2), r28
0xfe020510:   (17910005) ld.l 4(r28), r17
```

Addresses can be displayed symbolically as well. For instance, the format command can change the address format to 'e' (external symbol).

```
SEMi [t=0]> format e
SEMi [t=0]> get _main 3i
                          _main:   (8442ffc0) addu -64, r2, r2
                      _main+0x4:   (1c400801) st.l r1, 0(r2)
                      _main+0x8:   (1c408835) st.l r17, 52(r2)
```

There are actually five default formats at any one time in SEMi, or more correctly, one default format from each of five groups. The first group is simply the number of items to be displayed, which should be familiar from using the `get` command to view the program in memory. All other format codes are single letters. The second group indicates the size of the items to be displayed. The formats in this group are c (char – 1 byte), h (halfword – 2 bytes), w (word – 4 bytes), and d (double – 8 bytes). The third group describes the representation of the item in memory. In this group, the formats are a(scii), b(inary), f(loating point), i(nstruction), n(umber – signed decimal), o(ctal), s(ymbol), u(nsigned decimal), and x (hexadecimal). The fourth group indicates the format by which instruction addresses will be displayed. In this group, v and p display absolute addresses, either virtual or physical (yes, SEMi handles address translation). The l and e formats generate symbolic forms, relative to the nearest label available, or restricted to external labels (e.g., procedure names). The final group decides whether the destination address of a branch instruction is displayed as a global address (g) or a relative offset (r).

The run command can take an address argument. In this case, it sets a temporary breakpoint at the address and starts executing:

```
SEMi [t=0]> run _main
Halting at breakpoint 1 (executing)
SEMi [t=4840]> run _printf
Halting at breakpoint 2 (executing)
SEMi [t=4867]>
```

(It took a while to get to the user main() function because of all the setting up required by the user process, e.g., flushing the data caches.)

At this point, assuming the code is compiled with the normal i860 calling conventions, we can view the arguments as follows:

```
SEMi [t=4867]> view r16 3
r16:    0xfe02f020
r17:    0xfe020036
r18:    0x3045b250
SEMi [t=4867]> get r16 3a
```

```
              __EARTH_SIM+0x1c:   Hell
              __EARTH_SIM+0x20:   o %s
              __EARTH_SIM+0x24:   !\n\000\000
SEMi [t=4867]> get r17 6ac
                   +0xfe020036:   w
                   +0xfe020037:   o
                   +0xfe020038:   r
                   +0xfe020039:   l
                   +0xfe02003a:   d
                   +0xfe02003b:   \000
```

Of course, we need to know that r17 (the second argument to printf) is a pointer to a string in order to choose a sensible format. Note the difference the 'c' format code makes. Without it, the default size (word) is used. In the second case, the 'c' is mandatory because the address is not aligned for reading words and an error would result otherwise.[3]

The call instruction puts the return address in r1, so normally, if we're at the beginning of a function, we can get out of it by running to that address:

```
SEMi [t=4867]> run r1
Hello world!
Halting at breakpoint 3 (executing)
SEMi [t=9472]>
```

The first line after the command is the output of the program itself, rather than SEMi. On machines supporting X windows, one can create a separate xterm window for program output by including the switch -xf in the call to SEMi.

4 Invoking SEMi

The uniprocessor version of SEMi is invoked with the `semiuni` command. The command-line format is:

`semiuni` *switches [executable arg arg ...]*

The optional *executable* is the name of a COFF (Common Object File Format) executable file. This file can be generated by compiling C code using the pgcc C compiler or assembly code using the as860 assembler. SEMi will load the .text (program) and .data segments from the COFF file into its simulated memory, and set all uninitialized globals to 0 as specified in the .bss segment. If the file has a symbol table, SEMi extracts the symbols, which can then be used to specify addresses in SEMi commands, and may be used by SEMi's display routines. SEMi also extracts the starting address from the file and sets the program counter to that address. All other registers are initialized to 0, except for certain read-only bits in special registers. Any arguments that follow the *executable* are passed to the program, unless the `-nr` switch (see below) is used.

All switches for the simulator must come before *executable*. SEMi has a lot of switches, listed in Table 4, but, as with other tools such as gcc, most of the switches are specialized and can be safely ignored until the user has learned the basics. A switch is preceded with a hyphen (`-`). A switch can

[3] Actually, 'h' would work as well because the address is even.

be negated by adding `n` after the hyphen. Some switches can be followed by one or more integers or a string (there must be white space between the switch and value). Switches are evaluated left-to-right, so if two switches conflict, the one to the right prevails.

There are four types of switches:

- On/off switches turn a feature on or off. Each feature has a default initial setting (on or off). A feature can be turned on with the switch; the negated form ('n' between the hyphen and switch name) turns the feature off. These switches don't take values.
- One-of-n switches control features which have exactly one of several mutually-exclusive settings, and each setting is represented by a different switch. If two switches controlling the same feature are used, the one to the right prevails. Each feature has a default setting which is used if none of the relevant switches are used. Negating any switch in the set is equivalent to setting the feature to the default. These switches don't take values.
- Value switches may be followed by an integer, a string or filename. For such switches, there are two default values, one to use if the switch is not used at all, or if the switch is negated, and one to use if the affirmative form of the switch is used, but without a value. If the table shows only one default, then both are the same. For most filename switches, a value (valid filename) is required. For integer switches, there is a range of legal values, shown in the table.
- Set switches may be followed by a set of unsigned integers separated by commas and no white space. There is a range of legal values for each member of the set. If a particular member of the set is absent, the default value for that element is used. For instance, the -cd switch sets N, K, and L (respectively) for the D-cache. If the default setting is 7,2,5 then the switch '-cd ,,3' would set the values to N=7, K=2 and L=3; the first two fields are missing so the default values are used.

Note that if a switch that takes an argument is the *last* switch before *executable*, then it *must* have a value after the switch, or else SEMi will think *executable* is the switch value. Also note that the name of the executable can't start with a hyphen.

The following subsections describe the switches, which are grouped according to whether they control the parameters of the simulated machine, or the behavior of SEMi itself.

4.1 Simulated Machine Configuration

This group may be further subdivided into switches which parameterize the hardware external to the i860 processor (e.g., memory and I/O), and those which allow parameters within the i860 to be modified. Switches in the first category, therefore, present options which could be realized in hardware using off-the-shelf processors. Switches in the second group change the CPU itself, and therefore represent unrealizable modifications (unless you want to build your own i860) but can be used for hypothetical studies. In this group, the default settings always conform to the standard i860 configuration.

4.1.1 External Hardware

The `-ms` and `-mb` switches configure the physical memory. It is assumed that the number of bytes in physical memory is a power of 2; this power is specified with the `-ms` switch. The base address of physical memory is given with the `-mb` switch. This base needs to be aligned with SEMi's page size (4K).

The `-m` switch sets the timing parameters of the physical memory. The five parameters are:

Switch name	Type	Range (inclusive)	Default if no... Switch	Value	Purpose
-as	int	12 — 24	12		Aux. (meta) memory size ($\log_2$)
-b	int	300 — 2^{31}–1	∞	9600	Baud rate of serial port
-bb	on/off	—	on	—	Block on BRDY
-c2	int×4	0—20,0—20,4—10,0—255	none		L2 cache N,K,L,access
-cb	on/off	—	on	—	Cache blocks on load
-cd	int×3	0—20,0—20,4—10	7,2,5		Data cache N,K,L ($\log_2$)
-ci	int×3	0—20,0—20,4—10	7,2,5		Instruction cache N,K,L ($\log_2$)
-e	file	—	none	-of file	File stderr redirection
-fh	on/off	—	on	—	Scalar FP hazard resolution
-fl	on/off	—	on	—	FP loads don't stall
-i	file	—	none	required	-in and -is together
-ib	int	0 — 2^{32}–1	0xf2000000		I/O address base
-if	file	—	none	required	File stdin redirection
-is	file	—	none	required	Serial port input redirection
-m	int×5	0—255,0—255,0—15, 0—15,0—255	3,1,1,1,2		Main memory parameters
-mb	int	0 — 2^{32}–1	0xfe000000		Main memory base
-ms	int	12 — 32	25		Main memory size ($\log_2$)
-o	file	—	none	required	-of and -os together
-of	file	—	none	required	File stdout redirection
-os	file	—	none	required	Serial port output redirection
-r	on/off	—	on	—	Set up for runtime system
-rb	int	2 — 30	16		First reg. for parameter passing
-s	on/off	—	off	—	Superscalar mode
-v	on/off	—	off	—	Print versions
-wb	1-of-n	—	-wb		Write-back mode
-we	1-of-n	—	-wb		Write-exclusive mode
-wo	1-of-n	—	-wb		Write-once mode
-wt	1-of-n	—	-wb		Write-through mode
-xf	1-of-n	—	off	—	Xterm for file I/O (buffered)
-xfr	1-of-n	—	off	—	Xterm for file I/O (raw)
-xs	1-of-n	—	off	—	Xterm for serial I/O (buffered)
-xsr	1-of-n	—	off	—	Xterm for serial I/O (raw)

Table 4: Switches to **semiuni**

1. Page-mode access time
2. New-page access time minus page-mode access time
3. Page-mode recovery time
4. New-page recovery time minus page-mode recovery time
5. Delay before switching out of page mode

Each parameter is in clock cycles. The page-mode access time (read or write) applies if the new access is on the same page as the previous access and the page is still in the DRAM's output buffer. Otherwise, the new-mode access time applies. The page-mode recovery time is the number of cycles the memory waits between successive accesses on the same page. The new-page recovery time is the number of cycles the memory waits when accessing a new page. The final parameter is the number of idle cycles before the memory writes the DRAM buffer back to the memory plane, which means that the next access automatically is on a new page.

Since the new-page access and recovery times must be at least as large as their page-mode counterparts, these values are given in the command line relative to the page-mode values. For instance, to get a page-mode access of 3 cycles and a new-page access of 4 cycles, the switch '-m 3,1,...' would be used (4=3+1).

Note that the actual times for the CPU to access data are larger than the parameters given to the -m switch because these parameters don't include the time to access the bus.

The `-c2` switch enables a second-level (L2) cache in the main memory. It is assumed the L2 cache is between the bus and the main memory. This switch takes four values. The first three are N, K and L (each is the log of the actual value). N is the number of sets in the cache, K is the associativity (number of lines per set), and L is the line size (bytes per cache line). L must be at least as large as the largest line size of the on-chip caches. The last value is the number of cycles to access the L2 cache in the case of a hit. (In the case of a miss, the parameters in the -m switch apply instead.) Normally, this value would be less than the first parameter in the -m switch, or else there would be no reason to have an L2 cache!

The `-ib` switch sets the base address of the memory-mapped I/O devices. Currently, the assignment of addresses to I/O functions is fixed, but the base of the entire range can be modified in the command line. Details of the I/O device can be found in the appendix.

The `-b` switch sets the "baud rate" of the serial port (in bits per second). This should only be used when one wants to model accurately the timing of an I/O-intensive program, since it slows down the simulation considerably. (During serial writes, the CPU typically sits in a busy loop waiting for the port to become ready, but the simulator has no way to know this for sure, so it simulates every cycle.) The default is for infinite speed, meaning the serial port is always ready to receive/send data.

Switches `-wb`, `-we`, `-wo`, and `wt`, of which only one may be used, set the default cache writeback policy. This is the policy used unless overridden by the WT bit in a page table entry.

- With write-through (wt), writes always go out to main memory even if they hit the cache. Cache lines are either S(hared) or I(nvalid), never E(xclusive) or M(odified).
- With write-exclusive (we), if a CPU reads a new line into its cache, the line is in E state. Writes do not go out to memory, but cause the cache line to go to M state. Such a line will not go out to memory until displaced by another incoming line, a flush operation, or an inquiry from another CPU (in multiprocessor configurations). If another CPU reads the same memory line, the inquiry will invalidate the first CPU's copy, even if it is in the E state; S state is not allowed.
- Write-back (wb) is similar to write-exclusive, except that sharing is allowed. If a CPU has a line in E state and another CPU reads the line, the cache line in both CPU's go to S state. (WB and WE are equivalent if there is only one CPU on the bus, since nothing would ever cause a line to become shared.)
- With write-once, if a CPU writes to a line which is in its local cache in the S state, the write goes out to main memory and also updates the cache line, which changes state to E. The policy for that line reverts to write-back.

The `-rb` switch gives the number of the first integer register used for parameter passing in software traps (see Section 2.3). For instance, if `-rb 10` is used, then when the write() system function is called through a software trap, SEMi expects the 3 arguments to be in registers r10–r12.

4.1.2 Non-Standard i860 Options

Any timing results obtained with these switches won't accurately reflect the performance of a real i860. They are intended for hypothetical experimentation only.

The `-cd` switch can be used to change the data cache parameters in the i860. This switch takes 3 values: N, K, and L (each is the log of the actual value). N is the number of sets in the cache, K is the associativity (number of lines per set), and L is the line size (bytes per cache line). L must be at least 4 (16 bytes); this floor guarantees that quad loads and stores are correctly simulated.

The `-ci` switch controls the same settings for the instruction cache.

Changing cache parameters can lead to behavioral changes in the code if not done carefully. If the code being simulated flushes the data cache, it will need to iterate over the sets. Such code should read the cache size from bits 18–21 of the EPSR. This 4-bit field is added to 12 to give the log (base 2) of the number of bytes in the cache (if the cache is smaller than 4K, the field is still 0). However, this assumes K and L are the default values. If K changes, SEMi must adjust for the different associativity, since the RB field in the dirbase register (used to select one item of a set to flush) only has 2 bits (for an associativity of 4). If $K < 2$, the lowest K bits of the RB field are used. If $K > 2$, all lines whose lower 2 bits match the RB field are flushed by a single flush instruction.

The next four switches optimize certain features of the i860 hardware. The 'on' setting is the default, corresponding to a real i860. One can explicitly turn the switch off to try optimizing the hardware.

If a load misses the internal data cache and the i860 tries loading a cache line, the cache will be blocked from further access until the load is complete. This default behavior can be overridden with `-ncb`. This will allow the cache to be probed by other loads and stores while a cache load is in progress. The cache is still blocked if the bus is locked and if a store writes to the line which is being loaded.

The i860 has a simple mechanism for preventing read-after-write hazards; any scalar FP op will block the entire FP unit until it is finished, even if the next FP op uses independent registers. The `-nfh` switch tells SEMi to use the same register-based scoreboarding technique for the FP unit that it uses for the integer unit. This will reduce the number of FP hazards significantly, but not eliminate them entirely, because SEMi still counts the extra stalls caused by pipelined double-precision multiplies, which would not be eliminated by a scoreboard. This will make FP-intensive code run faster (about 5%).

The `-nfl` switch tells SEMi not to stall FP loads. The i860 normally waits until a load finishes, even if it misses the cache and there are non-load-store instructions following the fP load.

Certain instructions, such as CALL, stall until the next BRDY (bus ready) signal if there is a bus load in progress. The `-nbb` switch turns this off and allows such ops to continue while the cache is being loaded.

The final feature `-s` gives the processor the ability to perform in-order-only superscalar execution. The CPU executes code according to the same 1-instruction-per-cycle semantics (or 2-IPC semantics in the case of Dual Instruction Mode), but tries to execute as many instructions as it can in one cycle, only stopping when a hazard occurs or the instruction is a load or store. Instructions can only be issued in order; a hazard will block all instructions that follow even if those other instructions could be executed by an out-of-order superscalar.

4.2 SEMi Options

The `-as` switch is similar to `-ms`, but indicates the log of the size of the *auxiliary* memory. There is no switch corresponding to `-mb` because auxiliary memory must start at $\wedge 0$.

Switches beginning with e, i, o or x (11 total) control redirection of simulated ports. This can be used to separate interaction with the simulated machine from interaction with SEMi itself, for SEMi by default uses the same stdin/out for both.

SEMi simulates a serial I/O port (see Appendix A), indicated with the letter `s`. Switch `-is` causes data that is read from the serial port to come from the specified file. (Note that using a redirection command '<' in the command line won't work, as this will affect the input to SEMi itself rather than the input to the simulated serial port.) Similarly, `-os` causes output to the serial port to go to the given file instead of the screen. On systems supporting X-Windows, the switch `-xs` causes SEMi to create an xterm and connect its output (what the user types within the window) to the simulated serial input port, and its input to the simulated serial output port. (The xterm command must be in the user's PATH.) Input to the port will undergo normal input line processing (e.g., nothing will go to the port until RETURN is entered), but if `-xsr` is used instead, the input will be read "raw" (all characters go immediately to the input port, with no echo). The `-is` and `-os` commands override `-xs`; for instance, the command

```
semiuni -is foo -xs ...
```

will create an xterm, and output to the serial port will be displayed on this xterm, but anything typed in the xterm will be ignored, and input from the serial port will come from the file foo.

Commands containing the letter `f` can be used for file descriptors 0–2, which, like all simulated file descriptors, are used in software traps (see Section 2.3) used to simulate POSIX read() and write() commands. Switches `-if`, `-of`, `-xf` and `-xfr` are analogous to the corresponding 's' switches, but they control file descriptors 0 and 1 rather than the serial input and output. The `-e` switch causes output to simulated file descriptor 2 to be redirected to the specified file (no 'f' is needed because there is no error serial output port). Unlike the other switches, the filename argument to `-e` is optional. If one isn't used, the simulated stderr output is redirected to the same file as the simulated stdout output.

Switches `i` and `o` are shortcuts setting both serial port and file port redirection simultaneously. For instance, `-i foo` is equivalent to `-if foo -is foo`.

The `-r` switch tells SEMi to assume the standard EARTH-MANNA program interface is being used (this is also the default mode). Node information and command line arguments (anything following the *executable* filename) is packed into memory just after the starting address, in a pre-defined structure which the EARTH-MANNA library start-up routine knows how to access. If *executable* was compiled from a C program, then within main(), argv[0] will point to *executable* (the string), argv[1] will point to the first arg after *executable*, etc. Arguments are limited to a total of 976 characters (including 1 character between successive arguments). The actual program (beginning with `_main`) will start about 500 cycles after the start of the runtime system. The `-nr` switch turns off the `-r` switch.

The `-v` switch causes SEMi to list the version number of each module in the program and then quit, ignoring all other command line arguments.

5 Using SEMi

SEMi has an interactive command interface. SEMi displays a prompt (which includes the current cycle count), then the user types a command, and SEMi returns with its response (if any). Most commands merely cause SEMi to display or modify some portion of the CPU or memory state. Exceptions are the `cycle` and `run` commands, which cause SEMi to simulate normal operation until one of the following occurs:

- The specified number of cycles (in the `cycle` command) is completed;
- An illegal operation occurs during simulated execution or during execution of a breakpoint;
- A breakpoint executes a halt command;
- A `trap r0,r0,r0` instruction is executed.

If SEMi halts prematurely, it will always complete the cycle cleanly. If the halt occurs while simulating the CPU, SEMi will simulate the bus for one cycle before returning to the user. This allows the user to restart normal execution without timing discrepancies.

A SEMi command consists of a command name followed by zero or more arguments. Command names are case-insensitive, and can be abbreviated if the abbreviation is unique (in most cases, a single letter is sufficient). More than one command may be typed on one line if they are separated by semicolons.

5.1 Expressions in SEMi

Most SEMi commands take various numerical arguments, and general expressions can be used where such arguments are expected. Expressions have a C-like syntax, with the addition of some special operators unique to SEMi. In SEMi, an expression consists of one or more atoms and zero or more operators, with no white space.

Atoms are the smallest units of an expression, and consist of numerical constants, symbols (e.g., from the COFF file), and register names. Each atom has a specific type, either *int* (signed long), *float* (single-precision floating point), or *double* (double-precision floating point). Atoms are evaluated according to the following rules:

1. A register name (see Section 2.1.2 and Table 1) in upper or lower case is replaced by the value stored in the register at the time the expression is evaluated.
   - If the register name is preceded by a single period, then the atom's type is float.
   - If an even register name is preceded by two periods, then the register and its odd partner are read together, and the atom's type is double.
   - If the register name is not preceded by any periods, then the atom's type is int, even if the register is a floating point register.
2. An integer constant has type int. The integer is assumed decimal unless its base is specified with a leading `0x` (hex), `0b` (binary), or `0` (octal). A leading sign is optional.

3. A floating point constant (with optional exponent, decimal point and leading sign) has type double, unless there is a trailing `f` or `F`, which makes it a float (as in C). (Note: the constant *must* have an exponent, decimal point or trailing `f` or `F`. Without at least one of these three things, SEMi has no way to distinguish this from an integer constant, and will give it the type int.)

4. A name in the symbol table (e.g., from the COFF file) is replaced by the value, and has type int, provided none of the rules above apply first.

5. A valid hex integer without a leading 0x is evaluated and given type int, provided none of the rules above apply first. For example, if FD is in the symbol table, the atom `FD` is evaluated according to the value in the table, rather than as the constant `0xfd`.

6. A character surrounded by single quotes has type int and value given by the ASCII code for the character. Unprintable characters may be generated using the standard C backslash conventions [4, page 181] [5, page 38]. Examples of valid character constants are: `'a'`, `'\n'`, `'\177'`, `'"'`, `'\''`.

Operators include the usual arithmetic, logical and bitwise operators from C with the standard precedence rules [4, page 49] [5, page 53]. Sub-expressions, like atoms, have types; a sub-expression's type is determined by its operator. Except where noted, binary operators expect both operands to be the same type, and will "promote" the lower type to the higher type if they are different (int is lowest; double is highest).

1. The arithmetic operators ({+, −, *, /, %}) can operate on any type, producing a result of the same type.

2. Comparison operators ({==, !=, <, <=, >=, >}) also operate on any type, but always yield ints (0 or 1).

3. Bitwise operators ({<<, >>, |, &, ∧, ∼}) require int operands and yield ints.

4. Logical operators ({||, &&, !}) convert operands of any type to true (non-zero) or false (zero), and produce ints (0 or 1).

5. The ternary op "?:" yields either the second or third operand, *without performing any conversion,* depending on whether the predicate is non-zero or zero.

6. A * can be used as a unary op to dereference an address. Its operand must be int, and it produces an int. Normally, this will fetch a 32-bit value from memory. However, if the effective address is misaligned, the largest value consistent with the alignment is fetched and sign-extended. (For an odd address, a byte is fetched; for an even but non-quad-aligned address, a halfword is fetched.) To fetch a single byte from an even address or a single halfword from a quad-aligned address, use explicit bitwise operations. (For instance, use `*r2&0xff` to fetch an unsigned byte from the address stored in r2 and `*r2<<24>>24` to fetch a signed byte from the same place.)

   If the address dereferenced is in the local cache, the cache copy is used in case of inconsistency with main memory. This reading is non-intrusive. If the address is not in the cache, the corresponding physical location in main memory is read, but is not loaded into the cache.

The following operators are special to SEMi:

1. The unary dereferencing operator (*) normally fetches an int. However, as with register references, one can control the resulting type with optional periods. The operator `*.` fetches a 32-bit value from the specified address, but the result has type float. Similarly, `*..` fetches a 64-bit value and produces a double. The address must have proper alignment.

   It is important to understand how these operators work. Suppose address 8 contains the single-precision floating point value 5.0. The expression `*8+1` has type int and the value 0x40a00001,[4] probably not what was expected! The reason for this is that memory is not typed, so when SEMi evaluates `*8`, all it sees is bits in memory. The floating point value 5.0 is represented with the bit pattern 0x40a00000, which is then incremented. On the other hand, the expression `*.8+1` yields the expected result of 6.0 (type float), because `*.8` yields 5.0 with type float, and the int 1 is promoted.

2. The C dereferencing operator (*) fetches from regular memory. SEMi has an equivalent operator (∧) for fetching from meta-memory (see Section 2.4). Both dereferencers have the same precedence, as defined by C. SEMi also has operators `∧.` and `∧..`, analogous to `*.` and `*..` (defined above).

3. One additional non-standard-C operator in SEMi is juxtaposition. The form *expr1*(*expr2*) is equivalent to (*expr1*+(*expr2*)) (*expr2* must be in parentheses in order for the juxtaposition to be recognized). This form is consistent with the way indexed indirection is written in i860 assembly language, and is provided as an easy way to copy an expression directly from an operand field in an assembly instruction. For consistency with assembly language, the precedence of juxtaposition is lower than unary minus, so that `-4(r2)` is equivalent to `(r2-4)` rather than `-(r2+4)`. However, its precedence is higher than unary dereferencing, so that `*r1(r2)` is equivalent to `*(r1+r2)` rather than `((*r1)+r2)`.

   Note that SEMi is not a source-level debugger. In an expression such as `*r2(4)` (fetch from address r2+4), the offset 4 is in bytes, regardless of the size of the data type at the source-code level.

5.2 Display Formats

When SEMi displays data or reads data from input, it uses a specific format, unit size and unit count. A complete format specifier is a quintuple:

(*size,display,address,offset,count*)

where *size* is the number of bytes per datum, *display* is the format used to represent each datum, *address* and *offset* affect the display of memory addresses and branch offsets, and *count,* a positive integer, is the number of items. Table 5 gives a list of all the choices for the first four categories.

If SEMi prints a value that it knows is an address, it will use the address format code to determine how to print the address. Addresses are either numerical (raw hex values) or symbolic (relative to labels in the symbol table). Numerical addresses can be either the virtual address or the physical address (following MMU translation). Symbolic addresses have the form *symbol+offset*, where *symbol* is the nearest valid symbol which is not at a higher address than the address displayed. This symbol is either the nearest external symbol (`e` code) or the nearest symbol of any kind, local or external (`l` code). Using the `e` code can help to determine which subroutine contains the address.

Most control transfer instructions printed using the `i` format are affected by one other category, the offset format. These instructions encode the destination of the jump as an offset from the current PC.

[4]This assumes the simulating machine's native FP format is the same as the i860's.

Category	Letter	Description
Size (bytes per datum)	c	Characters (1 byte)
	h	Halfwords (2 bytes)
	w	Words (4 bytes)
	d	Double words (8 bytes)
Display format	a	ASCII
	b	Binary
	f	Floating point (single or double)
	i	Instruction
	n	Numerical (signed decimal)
	o	Octal
	s	Symbol (assume datum is an address; look in symbol table)
	u	Unsigned decimal
	x	Hexadecimal
Address format	e	Relative to nearest external symbol
	l	Relative to nearest local (static) or external symbol
	p	Physical address (numerical)
	v	Virtual address (numerical)
Offset format	g	Compute and show global destination address
	r	Show relative offset only

Table 5: Format specifiers

One can choose to view the relative offset (`r` code) or the resulting global address (`g` code). However, the `g` code only works when displaying instructions stored in memory, since only then is the base of a relative jump defined.

In general, the first four categories are orthogonal; any of the $4 \times 9 \times 3 \times 2$ combinations are allowed. Certain combinations may be meaningless; for example, the *offset* format is meaningless (and hence ignored) unless the *display* format is `i`. Also, in some cases, one format will override other specifiers:

- If the *display* format is `s`, then data are printed in 4-byte units regardless of the *size* format.
- If the *display* format is `f` or `i`, and the *size* format is smaller than 4 bytes, then data are printed in 4-byte units. (Properly-aligned doubles may be printed with `i` format, in which case two instructions will be displayed on the same line. However, this doesn't imply that the two will execute in the same cycle; that depends on the setting of DIM bits.)

Specifier `a` may be used with any size. If `a` is used with halfwords, words or doubles, characters will be printed out in groups of 2, 4, or 8. These will be in proper character order. For instance, 'abcd' would be printed out as '64636261' with format `(w,x,*,*,*)` (because the i860 is little-endian) but 'abcd' with format `(w,a,*,*,*)`. Non-printable characters are displayed using standard C conventions, e.g., special forms for common control characters such as "\n" and "\0" with \ooo (3-digit octal number) used for all others [4, page 181] [5, page 38]. A backslash character is displayed as "\\".

A format quintuple may be specified within a relevant command. However, for ease of use, SEMi maintains a "default format" quintuple. Initially, this default is `(w,x,p,g,1)`. The default can be changed using the `format` command (see Section 5.4), and may be selectively overridden componentwise by a specific command.

5.3 Argument Types

Several types of arguments can occur in SEMi commands. If an argument contains white space or semicolons, then it must be enclosed in curly braces ({}). The main argument types are:

Address: This is an expression whose result has type int and corresponds to a valid address in virtual memory.

Command: A command is a string containing a complete SEMi command which is part of a larger compound statement, such as an `if` command or a breakpoint. It is not parsed at the time the outer command is executed, but is saved, and then parsed and executed at some later point. Any variable expression within a command is not evaluated until the command is executed. If the string contains spaces or semicolons, it must be enclosed in curly braces. Curly braces may be nested if the command itself contains a command as an argument.

Filename: A filename argument must specify a full pathname or a path relative to the current directory. The user must have the required access to the file (which depends on the command).

Format: Table 5 lists a letter with each option. Because no letter appears twice, a format can be specified with any combination of legal letters in any order, with no spaces in between. Letters are scanned from left to right. If two or more letters from the same category appear, the rightmost letter prevails. In some cases, a format specifier may contain a count (positive integer). This may be interspersed with the format letters as long as the integer itself is contiguous. Therefore, for instance, '10xc' and 'c10x' are equivalent.

Int: This is an expression whose result has type int.

Mnemonic: This is a string representing a single instruction in assembly-language format. If the string requires white space between the opcode and the operands (as most do), it must be enclosed in curly braces. Labels may be used in expressions provided they have been previously defined (e.g., read from the COFF file). At the top level of an expression, the constructs %l, %h and %ha (which extract 16 bits of a 32-bit value) may be used. No label definitions, assembler directives, or macros requiring conversion to more than one instruction are allowed.

A mnemonic is assembled and converted to a value of type int. This can be used, in conjunction with the `=` command, to "patch" binary code within SEMi itself.

Pointer: Some SEMi commands use references to registers or memory locations. When such an argument is allowed, the following types of expressions are turned into pointers:

1. An expression which consists of a single register name, without periods, becomes a pointer to that register.
2. An expression whose *topmost* operator is `*` becomes a pointer to the address in main memory given by the rest of the expression. For instance, `*(r2+1)` produces a pointer to the byte after

the address stored in r2. Note that `*r2+1` is *not* turned into a pointer, because `+` is the topmost operator (it has lower precedence than `*`). Operators `*.` and `*..` are equivalent to `*`; the periods don't change the pointer in any way.

3. Similarly, an expression whose *topmost* operator is $\wedge$ (or $\wedge$`.` or $\wedge$`..`) becomes a pointer to the specified meta-memory location, provided it exists.

String: This is an arbitrary string without braces. If it has spaces or tabs, then it should be enclosed in braces (*not* quotes).

Switches: This is a set of one or more letters, in arbitrary order, from a specific set determined by the command.

Value: This is an expression producing a result of any type (int, float or double).

5.4 Commands

In the following list, optional arguments are in square brackets, and non-keywords are in italics. An underlined section followed by ellipses (...) denotes a sequence of one or more arguments that may be repeated (if the underlined section has more than one argument, then all must be repeated as a group). Unless otherwise indicated, an expression can contain register names and references to memory; the expression is evaluated by replacing all variable references with their current values and performing the arithmetic.

?

This prints a list of available commands.

= [-*switches*] *pointer*|*address* <u>*value*|*mnemonic*</u> ...

This is the SEMi assignment operator. The first argument (after optional switches) represents the start of a series of register or memory locations which are to be filled with the values that follow. If this argument is an address (int expression) rather than a pointer, it is converted to a pointer to the specified address in main memory.

The command can optionally begin with a switch consisting of a hyphen followed by a single letter, one of {c,h,w,d}. In this case, the unit size is set (to 1, 2, 4, or 8 bytes respectively); otherwise the *size* component from the default format is used. If the pointer is a register name, the unit size must be 4 or 8 bytes.

The remaining arguments are evaluated and stored in successive register or memory locations in increments of the unit size. The format of the value is determined solely by the value's type, and not by the default format or how the pointer was specified. For instance, "`= *100 5`" will write an integer 5, not a floating point 5.0, to address 100 even if the *display* property of the default format is float. The same is true for "`= *.100 5`"; the '.' makes no difference. However, some conversions are done based on size:

1. If the value is type int, and the unit size is 8, the upper bits are padded with 0. If the unit size is 1 or 2, the value is truncated.
2. If the value is type float, the unit size must be 4 or 8. If the latter, the float value is converted to a double before being stored.
3. If the value is type double, the unit size must be 4 or 8. If the former, the double value is converted to a float before being stored.

Writing to registers r0, f0 and f1 are disallowed. Writing to the pc is allowed, but may cause a momentary bubble in the pipeline. Writing to npc causes a delayed branch.

Writing to regular memory can cause cache and/or main memory state changes. If the location being written is not in the local cache, then only main memory is updated. If the location is in the local cache in either a modified (M) or exclusive (E) state, then the cache is updated and changed to M state, and the main memory is not touched. If the location is in the local cache in a shared (S) state, both cache and memory are updated and the cache stays in S state. In multiprocessor configurations, if main memory is written, all other CPU's on the bus are inquired; any modified lines are written to memory *before* the assignment operation is done, and all other cache copies are invalidated. These transactions will occur instantaneously and won't take any bus cycles.

NOTE: Changing the CPU or memory state is risky and difficult to model, due to the complex pipelining modeled by SEMi. Every effort has been made to ensure correct operation that is consistent with the changes made, but could lead to errors in timing. Assignments to anything other than meta memory should primarily be used for debugging a program.

break [[***-switches***] ***address*** [***command***]]
break ***-switches address*** [***address***]

This command creates, list and/or deletes breakpoints, according to how it is used. If the first argument starts with a hyphen, it is a set of switches; all remaining characters must be a combination of the following (case insensitive):

d	Delete specified breakpoints
l	List specified breakpoints
r	Reading data from location triggers breakpoint
w	Writing data to location triggers breakpoint
x	Executing instruction at location triggers breakpoint

The next argument is evaluated in the CPU's context, producing a value treaded as an address. The last argument, if any, is either another address (if the l and/or d switches are used) or a command.

If switches l and/or d are used, then the second form of the command is used. The command lists and/or deletes breakpoints at the specified address (if there is only 1 address argument) or in the specified range of addresses (if there are 2 addresses). If no other switches are used, all breakpoints in the range are listed or deleted. The r, w, and x switches can be included to limit listing or deletion to those breakpoints which are triggered by the accesses specified by the switches.

If neither l nor d is used, the first form of the command is used, and the break command creates a new breakpoint at the specified address. The second argument (or third argument if there are switches) contains the command executed when the breakpoint is triggered. If there is no command, the default action is to halt simulation when the breakpoint is triggered. Switches r, w, and x may be used to select the types of accesses which will trigger a breakpoint; if no switches are used, the default is x (execute) only. If a breakpoint is to be triggered by execution, it must be word-aligned.

Breakpoints may execute any commands except cycle, next, run, and the 'm' option of the stats command. Multiple commands may be entered by using a semicolon. For instance, the following sets a breakpoint which measures how many cycles are spent in the printf() routine.

```
break _printf {= ^20 clk; break r1 {= ^24 ^24+(clk-^20); break -dx pc}}
```

When printf() is called, the breakpoint saves the clock value in meta-memory location 20, then sets a breakpoint at r1 (which should contain the return address according to the usual i860 calling conventions). The second breakpoint, when triggered, subtracts the saved clock from the new clock and adds the result to meta-memory location 24 (which should have been initialized to 0). It then deletes itself (at address given by the pc. i.e., the current address).

Each breakpoint created is assigned a unique positive integer, which is shown when breakpoints are listed, and can be used to specify a breakpoint to delete with the delete command (see below). When the break command is used to create a breakpoint, it stores the value of this integer in a 4-byte word at meta-address 0. This value can be saved and used in a delete command (see below).

There can be at most one breakpoint per address. SEMi will return an error message if one tries to set a breakpoint at an address that already has one.

If there are no arguments to the break command, then SEMi lists all breakpoints.

cycle [*int*]

SEMi evaluates *int*, which must be positive, and executes that many cycles, unless halted prematurely by an error or breakpoint halt. If *int* is missing, 1 cycle is simulated.

delete ***int***| *****

If the argument is an asterisk (*), SEMi removes all breakpoints on the CPU. Otherwise, the argument must produce a breakpoint number which specifies a particular breakpoint to delete.

execute ***filename*** [***value***|***mnemonic*** **...**]

SEMi execute all commands in the named file, as if they had been typed at the keyboard interactively. Commands may be placed on separate lines of the file and/or separated by semicolons.

Before the file is executed, SEMi evaluates the arguments (if any) and stores their values into 8-byte chunks of meta-memory at addresses ∧0, ∧8, ∧16, etc. If an argument evaluates as a 4-byte value (int or float), the lower 4 bytes of the chunk get the value and the upper 4 bytes are filled with zeroes.

This serves as a kind of convenient parameter passing to "subroutines" stored in separate macro files, though there is no automatic type checking.

Files may be executed in breakpoints, but they can only contain commands allowed within breakpoints (no cycle, format, next, or run commands, and no stats with the 'm' option).

format ***format***

This command modifies the default format quintuple with the specifiers in *format*, updating the quintuple componentwise rather than replacing the whole thing. For instance, suppose the default format is `(w,i,e,g,1)` (print single instructions with external symbols and global destinations). If the command "format l2" is typed, the 'l' will override the `e` and the 2 will override the count of 1, but the other parts will remain in effect, making the new default `(w,i,l,g,2)`.

If *format* is bad, the entire command is ignored and the previous default remains.

get [***address*** [***format***]]

This is similar to the `view` command (see description below for general information), but always retrieves values from regular memory. The first argument is evaluated and used as the effective address of the first location in memory. Thus, "`get` *exp form*" is just a shorthand for "`view` *(*exp*) *form*". See `view` below for details.

A `get` command with no arguments is identical to a `view` command with no arguments.

halt [***string*** [***value*** **...**]]

If there are no arguments, this command simply halts execution. If there are arguments, the halt command behaves like the `printf` command (see below), printing the values according to the format specified by *string*, before halting execution. This is not very interesting when typed as a user-level command, so it is mostly used for breakpoints. See `printf` below for details.

if ***value command*** **...** [***command***]

This is equivalent to a general IF-THEN-ELSEIF-ELSE statement. Arguments form pairs of alternating values and commands (there must be at least one pair). The first arg of each pair (the value) is evaluated in turn until one returns a non-zero value, at which time the corresponding command is executed. If the number of arguments is odd, the last arg is treated as a final ELSE clause, which is executed if none of the values are non-zero. If the number of arguments is even and no values are non-zero, nothing happens. The following shows the equivalence between SEMi and C IF-statements:

SEMi command	Equivalent C statement
if E_1 C_1	if (E_1) C_1
if E_1 C_1 C_2	if (E_1) C_1 else C_2
if E_1 C_1 E_2 C_2	if (E_1) C_1 elseif (E_2) C_2
if E_1 C_1 E_2 C_2 C_3	if (E_1) C_1 elseif (E_2) C_2 else C_3
...	...

If an error occurs while evaluating an value, no commands are executed, and SEMi returns to the user.

next [***int***]

SEMi evaluates *int*, which must be positive, and executes that many instructions, unless halted prematurely by an error or breakpoint halt. Simulation halts after the instruction that follows the simulated instruction(s) has been fetched. Delay slots count as part of the *int* instructions even if they are skipped.

If *int* is missing, 1 cycle is simulated.

printf ***string*** [***value|pointer*** **...**]

This command works much like the standard C printf() function [4, pages 145–147] [5, pages 243–245]. The first argument is a format string (but enclosed in curly braces rather than quotes) which may contain non-printable characters (using backslashes as in C [4, page 181][5, page 38]) and format specifiers beginning with %. The remaining arguments are evaluated and matched to the format specifiers. Width and precision modifiers have the same meaning as in the regular printf. For instance, the command

```
printf {I'm executing address %x with f16=%f\n} pc .f16
```

will print the values of the PC and f16 in the formatted string. Note the period before f16, which tells SEMi to read f16 as a float rather than a long.

The SEMi version of printf does a little more argument checking than its C counterpart. For instance, the command above fails if the period before f16 is removed, because SEMi detects a mismatch between a floating point format code and an integer argument.

The format codes %d, %i, %o, %x, %X, %u, %c and %p, and size specifiers (*), take integer arguments. The format codes %f, %e, %E, %g, and %G take float or double arguments. The %s format takes a pointer to either main memory or auxiliary memory (but not a register). If the argument corresponding to this format is an integer, the integer is converted to a pointer to main memory, as in the `=` command.

The printf SEMi command does not support the %n format. However, it offers two formatting commands not available in the regular printf. Each takes an int value, and each interprets precision and field width modifiers in the same way as the %s (string) format.

- %a converts the corresponding int value, which must be a valid address in main memory, to a string containing a symbolic label. This label consists of an offset from the nearest external symbol from the symbol table, and corresponds to the `e` address format (see Section 5.2). The resulting string may be affected by width and precision specifiers in the %a command itself.
- %A is the same as %a, except that the nearest label of any type (local or external) is used.

put [-***switches***] ***pointer|address*** ***value|mnemonic*** **...**

"Put" is just an alternate name for the "=" command (described earlier).

run [***address***]

Executes until an exception (halting breakpoint or error) occurs. If *address* is present, it is evaluated and a temporary breakpoint of access type "-x" is set at the resulting address. Thus, the meaning is "run until the CPU reaches *address*." For example, "run _printf" will run until _printf() is entered. At that point, "run r1" will continue until this function is exited (assuming the normal i860 calling convention of keeping the return address in r1).

Note that the program could halt before reaching the specified address (due to an error or another breakpoint). In that case, the temporary breakpoint will still be there.

Letter	Description
b	Branches taken and not taken
c	Cache hits and misses
h	Hardware configuration (no dynamic statistics)
m	Memory bus usage
o	Op counts
r	Reset all counters
s	Stalls (see Section 2.1.4)
t	Totals (a summary)

Table 6: Switches for the `stats` command

sprintf ***pointer*|*address string*** [***value*|*pointer* ...**]

This command is like the SEMi `printf` command, but writes to memory rather than stdout, like the C sprintf() function. The arguments are the same as the SEMi `printf` command, except for an additional argument immediately after the command name. This argument is a pointer to main or auxiliary memory, or an int value corresponding to an address in main memory (as in the `=` command).

stats [***switches***]

Show statistics gathered since the last reset. Letters indicate specific statistics to show, as shown in Table 6. The default (if there are no args) is to show everything.

In the lists of instructions, the pseudo-ops nop, fnop, mov and fmov appear. The simulator recognizes the special pseudo-ops and represents them internally using special forms for the pseudo-ops (improving speed). These are counted separately in the statistics. [p]fiadd.ss(dd) r_i,r0,r_j is converted to [p]fmov.ss(dd) r_i,r_j. However, famov.sd and famov.ds are identical to the pseudo-op versions, and are not converted.

The 't' switch summarizes total statistics. Integer and floating point operations, and all derivative totals, have both RAW and ADJUSTED counts. The adjusted counts are an attempt to avoid inflating counts with instructions that don't do actual calculations. The adjusted integer op count excludes nop instructions. The adjusted FP op count excludes fnop and any pipelined FP op (excluding dual FP ops) in which both args are f0 (which means their only purpose is to advance the FP pipe without performing useful computation).

The 'r' switch will cause all counters to be reset to 0.

tbreak [[***-switches***] ***address*** [***command***]]
tbreak ***-switches address*** [***address***]

This is identical to the `break` command, except that the breakpoint is temporary and will disappear as soon as the CPU encounters it.

When used to list breakpoints, *tbreak* and *break* are identical.

view [***value*|*mnemonic*|*pointer*** [***format***]]

This is used for viewing part of the state of the processor or memory and for evaluating expressions. If the argument is a value or mnemonic, it is evaluated (or assembled) and the resulting single value is printed. Regardless of the type of the resulting value, the value is displayed according to the default format. Thus, `view` can be used as a simple desk calculator that can do such things as look up symbols, convert bases, or show how a particular bit pattern looks as a floating-point number.

The optional *format* argument allows one to change the format from the default componentwise. The new argument only affects the format during the current command and doesn't change the default for future commands (only the `format` command does that).

If the first argument is a pointer, then values are fetched from registers or from regular or meta memory, beginning at the location specified in *value.* For instance, "`view *fe000000 10c`" will display 10 bytes in memory from 0xfe000000 to 0xfe000009, "`view f0 16fw`" will display the first 16 FP registers (as single-precision floats), "`view fir 12w`" will display all the special registers, and "`view ∧0 3dx`" will show the first 3 double-words in meta-memory. All register values are shown as 32-bit words unless d (double-precision) format is used.

If memory contents are viewed, the value in each row is preceded by its address, which is written according to the address format (see Section 5.2). The address is followed by a colon and two characters, which describe the status of this location in the CPU's internal caches. The first character is I if the address is in the instruction cache, and the second character is M, E, or S if the address is in the data cache with the corresponding MESI state. A blank in one of these positions means the address is not in the corresponding cache. If a character is lowercase, it means the address hits the physical tags but misses the virtual tags. In either case, if the cache copy is inconsistent with main memory, the cache contents are shown. If the location is not in the cache, the main memory contents are shown but not loaded into the cache.

If there are no arguments to the `view` (or `get`) command, SEMi will display the state of the execution pipeline, showing the instructions in the eXecute and Decode stages using the default *address* and *offset* formats.

while *value command*

SEMi evaluates *value*, and if true (non-zero), executes *command.* This is repeated until *value* evaluates to 0, one of the commands halts, or an error occurs.

6 Multi-Processor Nodes and Multi-Node Machines

The preceding sections describe commands and features common to all variants of SEMi. This section describes additional information needed to run SEMiMANNA, which is a version of SEMi configured to emulate a full MANNA machine [1, 2]. Such a machine may have more than one node, each of which may have more than one i860XP processor. The nodes are connected by a network consisting of packet-switched crossbars. A node is a single point in the network, consisting of a bus with private memory, I/O devices, a connection to the network and one or more processors. Each node has the same configuration (number of processors, memory addresses, etc.) and components on each node can only address memory on that node.

Each CPU has a specific node number and processor number. The node number ranges from 0 to one less than the number of nodes, and the processor number ranges from 0 to one less than the number

Switch name	Type	Range (inclusive)	Default		Purpose
			No switch	No value	
-ag	int	$0 — 2^{20}–1$	half		Aux memory global space
-lb	int	$0 — 2^{32}–1$	0xf0000000		Link chip address base
-ld	int	1 — 15	9		Link/crossbar input delay
-n	int	1 — *max*	*max*		Number of nodes
-p	int	1 — 16	2		Number of processors on each node
-qb	int	$0 — 2^{32}–2$	No queues	0xf1000000	Queues address base
-qd	int	3 — 16	10		Log of queue depth
-t	file	—	—	required	Topology file
-wd	int	$50 — 2^{31}–1$	50,000		Watchdog timer
-ws	on/off	—	off	—	Write-through snooping

Table 7: Extra Switches to SEMiMANNA

Topology number	# of nodes	# of crossbars	Ports per crossbar	Nodes per crossbar	Comments
1	1	0	—	—	Link active but unconnected
2	2	0	—	—	Links interconnected directly
3	16	1	16	16	
4	20	2	16	10	2 links between crossbars
5	40	4	16	10	Some pairs double-linked
6	80	8	16	10	"
7	120	12	16	10	"

Table 8: Built-in Topology Options

of processors per node. These values are hardwired into the NID and PID pseudo-registers, respectively, of each CPU (see Section 2.1.2), and may be read by a breakpoint command to determine its location.

6.1 Invoking SEMiMANNA

The command line for invoking the MANNA version is similar to the uniprocessor version, with the addition of a few arguments and switches. All switches in Table 4 (see Section 4) are valid and are used in the same way except where noted below. Additional switches are listed in Table 7. The SEMiMANNA command line is:

semimanna [*topology*] *switches* [*filename arg arg ...*]

The *topology* argument and the -n switch determine how many nodes are simulated and how they are connected to each other. If the *topology* argument is used, it must be the first argument. The available topologies and their configurations are listed in Table 8. If the argument is missing, then SEMiMANNA will check the environment variable MANNA_IDX for the value. If this variable is undefined, a single-node MANNA is created by default.

If none of the topologies in Table 8 are suitable, a custom table can be specified using the -t switch. The -t switch takes a mandatory filename, which specifies a text file containing one or more topology descriptions. Each of these has a number; the *topology* argument or MANNA_IDX value selects which one is used. The format of this topology file is described in Apendix B.

The -n switch allows one to simulate a subset of the specified topology. The *count* must be between

1 and the number of nodes normally associated with the selected topology. The simulator will simulate only the requested number of nodes, and enough crossbars to connect them. The variable NUM_NODES (normally accessed by both user and system software) is set to the *count*.

Note that -n may reduce the number of nodes, but not the interconnection. The command "semimanna 3 -n 2 ..." does *not* produce the same machine as "semimanna 2 ..."! Both have 2 nodes, but in the former case, there is a crossbar between the two links, so communications between the two nodes will be a little slower.

The -p switch controls the number of processors on each node. All processors on each node share the same bus with the same memory and other devices. Each processor has its own cache, and the MESI protocol is used to maintain consistency between different caches and between caches and memory. A bus arbiter determines which CPU has access to the bus, and operates in a round-robin fashion.

SEMiMANNA, like SEMiUni, provides auxiliary memory accessible only by the user and breakpoint commands. Because there are more than one node, it is useful to have some of this memory be private, so that the same breakpoint commands can run on all nodes and not interfere with each other. It is also useful to have some of this memory be shared, so that statistics from each node can be combined. Therefore, the -ag switch can be used to indicate the address of the beginning of "global" auxiliary memory. If this switch is followed by a value, the value is rounded down to the nearest page boundary. Addresses below this boundary are private, i.e., a separate copy exists on every node. Addresses on and above this boundary are shared, i.e., every node has access. If no value is given or the -ag switch is not used, then the default is to divide the auxiliary address space (whose size is specified with -as) in half, with the higher addresses being shared. The default value for the -as switch is 13 (note that it is 12 in SEMiUni), so if no switches are given, the auxiliary address space will be private from ∧0 to ∧4095 and shared from ∧4096 to ∧8191. If '-as 12' is specified, the single page of auxiliary memory will be shared.

The MANNA has additional devices not included in the uniprocessor version. The link chip and hardware queues, like the I/O module, are separate devices accessed through ranges of memory addresses. Like the I/O module, the specific arrangement of addresses within each module is fixed (see the appendix for details), but the base may be selected in the command line, using -lb and -qb. The hardware queue module may also have its depth (number of words stored) specified in the command line with -qd. The link chips and crossbar chips have input delays which can be selected using the -ld switch, which sets the input delay in cycles.

The -wd switch is used for setting the "polling watchdog" [7]. This feature exists in SEMiMANNA only and does not exist in the real MANNA. Basically, if a message from the network is received by a link, the link starts a timer, which is set to expire after *wdog* cycles. If the link's receive queue is read, the timer is reset. If the receive queue is completely *emptied*, the timer is turned off. If the timer expires (a message is pending and the link has not been read for *wdog* cycles), the simulator generates an interrupt on the AP. If this feature is used, The user code must include an interrupt handler at the address 0xffffff00 (this is a standard i860 requirement). If the -wd switch has no value, the default value 50,000 is used (corresponding to a timeout value of 1 millisecond).

The -ws (write-snoop) switch allows one to change the cache policy in the i860, though, like the switches in Section 4.1.2, this is not supported in the real processor. The i860 uses a write-invalidate policy, which means that if one CPU writes to memory, all other CPU's must invalidate their copies of that line. Setting the -ws switch allows certain writes to update these copies rather than invalidate them. Such updates are allowed if

1. The writing CPU does not have a copy in its own cache, or

2. The writing CPU has a copy in its own cache in S(hared) state, and a write-through policy applies (either because the -wt switch was used or the WT bit is set in the Page Table Entry corresponding to that line).

These restrictions are necessary to prevent cache lines from being in states which are inconsistent with the meanings of those states.

6.2 Using SEMiMANNA

SEMiMANNA's user interface is virtually identical to SEMiUni's. There are a few additional commands and command line switches, reflecting the fact that SEMiMANNA simulates a machine with multiple nodes.

Most SEMi commands refer to specific register or memory contents. In SEMiUni, there is only one place from which these can come. Since there may be more than one CPU in SEMiMANNA, there must be some way to indicate which CPU's state is requested. Therefore, the simulated state includes the notion of a "default CPU." For instance, if one examines the PC with a view command, the PC displayed will be from the default CPU.

A SEMiMANNA command can be preceded by one or more specifiers, which can change these two components before the command is executed. The new default remains in effect in the current and all future commands until changed by new specifiers. A '#' followed by a node number (with no space between) changes the default node to the given number. If there is no number, the default becomes the current node number plus 1 (modulo the number of nodes). A '@' followed by a processor number selects the appropriate CPU on the default node. A '@' by itself does a module increment similar to #. These two specifiers can go in either order, but must be separated by white space. A CPU specifier without a node specifier changes the CPU but stays on the same node. If there is a node specifier without a CPU specifier, the default CPU becomes the corresponding CPU on the new node.

For instance, the command

```
#2 @0 if r3==0 {#3 view r4}
```

will cause SEMi to check the value of r3 on CPU 0 of node 2, and if r3 is 0, then SEMi will display the value of r4 on CPU 0 of node 3. (Note that the command in the second argument to the if command changes the default node, but not the CPU, so the default remains the AP even though the node changed.)

If an asterisk (*) is used as a node or CPU specifier, then the command is performed on all nodes or on all CPU's. For instance, the command

```
#* @* view r4}
```

will view register r4 on every CPU in the machine. If a wildcard is used for the node (CPU), the default node (CPU) won't change, but the default will not apply to this command. Wildcards may not be used for the cycle, next or run commands.

CPU and node specifiers are not allowed in commands that are executed from breakpoints. Thus, a command executed within a breakpoint only has access to the CPU and memory where the breakpoint occured. Breakpoints are local to a node, but not a CPU (since they are associated with a memory location). If one wants only a particular CPU to trigger a breakpoint, then the breakpoint should contain an 'if' command which tests the PID meta-register.

Another extension to SEMiMANNA is the addition of a **link** command, which displays the state of the link chip on the default node.

The **format** command only affects a single node, but affects all processors in that node, and may affect the display of other components, such as the link.

SEMi tries to optimize for speed by simulating multiple cycles on one component (a CPU, memory or link) before switching to another.[5] In the uniprocessor configuration, this is transparent to the user. The multiprocessor version tries to be consistent; for instance, the command `cycle 10` tries to simulate 10 cycles on every component. However, if one CPU halts execution prematurely, because of an error or a breakpoint, SEMi may end with CPU's at different times. The SEMi prompt will display the earliest time which has been simulated on all components, but clock meta-registers on some CPU's may indicate a later time. However, correct global simulation is guaranteed; when the simulation continues, the later CPU's will be suspended until the other clocks catch up.

References

[1] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[2] Gesellschaft für Mathematik und Datenverarbeitung mbH. *MANNA Hardware Reference Manual.* Berlin, Germany, 1993.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 2nd edition, 1988.

[6] Les Kohn and Neal Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 13(4):15–30, August 1989.

[7] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[8] Neal Margulis. *i860 Microprocessor Architecture.* McGraw-Hill, Inc., Berkeley, California, 1990.

[5]This improves cache locality – meaning the caches on the simulating machine, and not the caches in the target machine!

Offset	R/W	Purpose
8	read	Returns the status of the serial ports: A 1 in bit 0 means there is data at the input port A 1 in bit 1 means the output port is ready to receive data
18	read	Data waiting at the input port
	write	Write to output port
203000	read	Least significant byte of clock (from buffer after loading it)
203008	read	2nd byte of clock (from buffer)
203010	read	3rd byte of clock (from buffer)
203018	read	Most significant byte of clock (from buffer)
306000	write	Set interrupt of AP
306008	write	Clear interrupt of AP
307000	write	Set interrupt of CP
307008	write	Clear interrupt of CP

Table 9: I/O Device Addresses and Functions

A MANNA-Based External Devices

The following section describes devices simulated by various versions of SEMi, other than the processors, memory and bus. These devices are addressed using loads and stores, but have addresses outside of the memory range. The devices described here are based on the MANNA hardware. This section is intended for programmers who need to program their simulations in order to do I/O and other operations properly (e.g., programmers may need to know where the I/O addresses are and how they are used.)

Each module takes a block of addresses, distinct from all other devices, in the physical address space. Each address range has a base address, which is selectable by the user in the command line. Each function within a module is given a specific offset within the address block, hardwired into the module, so the user can change the addresses of all functions of one type only by changing the base address.

A.1 I/O Module

Table 9 lists the I/O registers and their functions. Each register is a single byte; a load or store of anything larger than a byte has undefined results. All offsets are listed in hexadecimal. In configurations with multiple CPU's on a bus, it is recommended that these registers only be read when the bus is locked.

The serial ports (input and output) are double-buffered, and their speed is controlled by the -ib switch (see Section 4.1.1). Each should be accessed only when the appropriate bit in the status register indicates it is ready. Otherwise, results are undefined.

The I/O module has a clock running at one-half the speed of the main clock. At any time, the value is therefore half of the CLK meta-register (modulo 2^{32}). The contents can only be read 1 byte at a time, and each byte has a separate address. Reading the least-significant byte loads the clock into an internal buffer; subsequent reads of the other three bytes read from this buffer, so that assembling the four bytes yields a consistent clock reading.

The CPU's can be interrupted (forcing a jump to the interrupt handler) by writing to certain addresses, as shown in the table. Writing to such an address sets the condition forcing the interrupt, which is serviced as soon as it is allowed. A second address is written to clear the interrupt (either by the interrupt handler, or by the code that set the interrupt if it changes its mind.

Offset	R/W	Purpose
0	read	Read data
	write	Write data
10	read	Read command
	write	Write command
20	read	Read either
	write	Write either
70	read	Read status of FIFO's

Table 10: Link Device Addresses and Functions

Command	Name	Description
12 (0xc)	Close	Disconnect
14 (0xe)	Route	Connect to port i
15 (0xf)	Filler	Ignored

Table 11: Crossbar Commands

A.2 Link Module

In multinode configurations, link chips connect a node with the network. The link consists primarily of a pair of FIFO's, one for input and one for output. One end of each is connected to the main bus, and is accessed through normal loads and stores. The other end of each is connected to a bidirectional 9-bit port (1 byte plus a tag bit to identify a byte as either a data byte or control byte).

Each FIFO is 64 bits wide, the same width as the main bus. Data words going out are transmitted as 8 consecutive bytes. Incoming bytes are gathered 8 at a time and put into the FIFO.

Table 10 lists the link registers and their functions. All offsets are listed in hexadecimal.

The FIFO distinguishes between commands and data, since bytes going over the network are tagged one or the other. The distinction is made at the bus side by selecting different addresses. The processor identifies the contents on the data bus as being data or a command by writing to one address or the other. When the processor reads from the link module, it must read from the data address or command address. If the processor reads from one address and the head of the incoming FIFO has the other type, results are undefined; thus, the protocol must guarantee that the processor knows what is coming next. The processor can choose a third address, which reads either one, but there is no way to tell whether the values read are data or commands.

A.3 Crossbar Module

Data and commands written to the link chip end up going to the nearest crossbar. Each crossbar port takes 1 byte of input every cycle, with an extra bit tagging the byte as a command or a data byte. A command byte has two nibbles; the most significant nibble is a command, one of those in Table 11, and the least significant nibble is an argument. The argument is meaningful only for the route command (0xe), in which case it specifies the output port.

A message sent over the crossbars looks like Figure 2. The application-dependent payload message is preceded by zero or more ROUTE commands. When the message gets to the first crossbar, the crossbar reads the first ROUTE command, and the lower nibble specifies a particular output port. The entire

ROUTE_1	ROUTE_2	$\cdots$	ROUTE_n	PAYLOAD	CLOSE

Figure 2: Message Format for Crossbar Transfer

message, except for the first ROUTE command, is sent to that port, which sends it to another crossbar or the final destination node. Each intermediate crossbar in turn reads one of the ROUTE commands and deletes it from the stream, so that the destination node only sees the payload message terminated by a CLOSE command. (This assumes correct routing, which is the responsibility of the network driver software simulated on the nodes.)

B Format of a Topology File

This appendix describes the format of a topology for those who wish to create non-standard multiprocessor configurations, and can be skipped by those using only the built-in topologies listed in Table 8.

A topology file consists of a series of consecutive blocks separated by blank lines. Each block has the following format:

- The first line begins with the keyword **IDX** (no leading space) followed by a space. Text following this space is scanned for an instance of the selected topology number (the *topology* argument of the semimanna command, or the MANNA_IDX environment variable), with at least one space before and after. A match means this block is selected. Anything else on the line is ignored, so comments can be inserted as long as they don't have integers separated by spaces.
- The second line has three integers separated by white space:
  1. The number of nodes
  2. The number of crossbar switches
  3. The number of ports per crossbar

  If there is more than one crossbar, then the third number should be more than the first divided by the second, because extra ports are needed to connect one crossbar to another! These three numbers may optionally be followed by one or more expressions, whose role is explained below.
- All remaining lines specify some kind of routing. The general form of each line is one of

  crossbar-range port-range delimiter expression [*expression*]
  node-range delimiter expression [*expression*]

In the routing lines, the part before the delimiter indicates one or more ports, which may be the output ports of crossbars (first form) or the outputs of nodes (second form). In the first form, the two components are separated by white space. Node, port and crossbar numbers range from 0 to the maximum, one less than the corresponding parameter in the second row of the block. Each component may be a single number, or a pair of numbers separated by a hyphen and no space; the latter form indicates a range bounded by the two numbers. In the first form, both components can be ranges, in which case the specifier represents a two-dimensional sub-range.

```
IDX 10 6-D hypercube
64 64 7
0-63 : p 6
0-63 0-5 . x^(1<<p) p

IDX 11 16x16 baseline (2x2 crossbars)
16 32 2 1<<(2-(x>>3)) r1*2
0-15 . p>>1 p&1
0-23 0-1 . r2*(x/r2)+((x%r2)>>1)+r1*p+8 x&1
24-31 0-1 . (x-24)*2+p
```

Figure 3: Topology Description for Hypercube and Baseline

The expressions specify a destination. If there is one expression, the destination is the input port of the node whose number is given by the value of the expression. If there are two expressions, separated by white space, the destination is port number P of crossbar number C, where C is the value of the first expression and P is the value of the second. The delimiter also is surrounded by white space. A period (.) indicates a unidirectional connection from the source port (left of the period) to the destination port. A colon (:) indicates a bidirectional connection, i.e., after connecting source to destination, SEMi makes another connection from destination to source.

Destination expressions are in the same form as other expressions in SEMi (see Section 5.1, with the following restrictions:

- No floats, doubles, or memory references are allowed.
- Only the following symbols are recognized:
  - `X` and `x` refer to the number of the source crossbar, or -1 if the source is a node;
  - `P` and `p` are the source crossbar port number, or the node number if the source is a node.

  These refer to the source on the left side of the delimiter. They are mostly useful when ranges are used for the source, because the expressions are recalculated for each source in the range.
- Only registers from r1 to rk are allowed, where k is the number of optional expressions following the three mandatory integers in the second line (the line immediately following the IDX line). For each destination derived, these expressions are recalculated with the new crossbar and port/node numbers, and are thus most useful for computing common subexpressions containing node, port and crossbar variables. The $(i+1)^{st}$ expression in the second line may use registers from r1 to ri to refer to expressions already computed to the left of the $(i+1)^{st}$ expression.

Figure 3 gives an example of a file with two topologies. The first is a six-dimensional hypercube. Each of the 64 nodes is connected to port 6 of a 7-way crossbar. Each of the other 6 ports of a given crossbar are connected to the corresponding ports of its neighbors, each of which is computed by inverting one bit of the crossbar number. The second topology (11) is a baseline network (a form of the banyan or delta network) with 2×2 crossbars.

Both topologies are mathematically regular, and can be easily modified for different numbers of nodes. Topologies which are more complex may require use of many subexpressions or the use of conditional operators (? and :) to produce single expressions.