

# **Liberty Simulation Environment User's Manual**

**The Liberty Research Group**

# **Liberty Simulation Environment User's Manual**

by The Liberty Research Group

Version 1.0 Edition

# Table of Contents

Preface .....	i
Typographical conventions used in this book .....	i
<b>1. Using emulators .....</b>	<b>1</b>
Concepts .....	1
The <i>emulation</i> interface vs. the <i>emulator</i> interface .....	1
Capabilities .....	1
Instructions .....	2
Contexts .....	3
State spaces .....	3
Using the emulation interface .....	3
Declaring the emulator in <b>lss</b> .....	4
Datatypes .....	4
Dealing with multiple emulator instances .....	5
Things to watch out for .....	6
The most basic tasks .....	6
Assigning an execution context .....	6
Finding the first instruction to execute .....	7
Creating a dynamic instruction instance .....	7
Executing an instruction (simple form) .....	8
Finding the next instruction .....	8
Determining when a context is finished .....	8
Putting it all together .....	9
Other basic tasks .....	10
Disassembling instructions .....	10
Accessing instruction information .....	10
Decoding instruction classes .....	10
Determining branch targets and direction .....	11
Comparing the age of instructions .....	12
Obtaining state space information .....	12
Detecting register-carried data dependencies .....	13
Obtaining memory access information .....	14
Detecting memory-carried data dependencies .....	15
Advanced context handling .....	15
Handling context switches .....	16
Controlling context mapping .....	16
Creating and destroying software contexts .....	16
Creating and destroying hardware contexts .....	17
An example of context mapping .....	17
Sharing state .....	18
More complex tasks .....	18
Executing an instruction (detailed form) .....	18
Manipulating operand values .....	19
Source operands .....	19
Destination operands .....	20
Interactions with the <i>speculation</i> capability .....	21

Intermediate operands .....	21
Other considerations .....	21
Handling speculation .....	21
Issues with imprecise speculation recovery .....	22
Creating proper instruction ordering .....	23
Modeling just timing.....	23
Fully-interlocked scheduling schemes .....	24
Relaxing WAR dependencies.....	24
Relaxing WAW dependencies.....	25
Relaxing RAW dependencies.....	25
Software-scheduled pipelines .....	25
Emulators with internal support for out-of-order execution .....	26
Instruction ordering and memory-carried dependencies.....	26
<b>A. LSS Reference .....</b>	<b>27</b>
Basic Syntax.....	27
Basic Data Types .....	27
int .....	28
float.....	28
boolean .....	28
char.....	28
string.....	28
literal .....	29
type.....	29
enumerations .....	29
arrays.....	29
structures.....	30
functions.....	30
external Types.....	31
Comments.....	31
Variable Declaration .....	31
Expressions and Operators .....	32
Unary Operator Expressions .....	33
Binary Operators and Expressions.....	33
The Ternary Operator.....	36
Assignment Operators.....	36
Indexing Expressions .....	37
Subfield Expressions.....	37
Function Invocation Expression.....	37
Data Initialization Check Expression.....	38
Expression Substitution via $\$ \{ \}$ .....	38
Statements.....	39
Control Flow .....	40
The <code>if</code> Statement .....	40
Loops .....	41
The <code>return</code> statement .....	41
Including Other Source Files .....	41
Declarations .....	42

Variables .....	42
Types .....	42
Functions .....	43
Conditional Assignment .....	43
Built-In Functions.....	43
Machine Construction Constructs.....	44
Module Instances.....	44
Creating Module Instances .....	45
Parameterizing Module Instance.....	45
Using Parameters.....	45
Code-Valued Parameters .....	46
System Defined Instance Parameters .....	46
Runtime Parameters .....	47
Module Instance Connections .....	48
Syntax and Semantics .....	48
Port Types and Connections.....	49
Polymorphic Types.....	49
Type Variables.....	49
The Or-Type.....	50
Constraining Port Types with Connections .....	50
Utility Functions .....	51
Augmenting Instance State.....	51
structadds.....	51
Runtime Variables.....	51
Modules.....	52
Module Declaration Syntax.....	52
Ports.....	52
Parameters .....	53
Leaf Modules.....	54
Module Attributes .....	54
Port Attributes.....	55
Methods and Queries .....	56
Events.....	56
Type Exports .....	57
Hierarchical Modules .....	57
Data Collectors.....	58
Packages.....	59
Using packages.....	59
Usage overview .....	60
Packages, Subpackages and Naming .....	60
Building Packages .....	61
Domains .....	62
Creating a Domain Class.....	62
Domain Types.....	62
Using Domains .....	63

# List of Tables

- 1-1. Standard instruction class names..... 11
- 1-2. Memory access flags ..... 15
- A-1. Binary Operators ..... 33
- A-2. System-Defined Instance Parameters ..... 39
- A-3. System-Defined Instance Parameters ..... 47
- A-4. Parameter Modifiers ..... 54
- A-5. Leaf Module Attributes ..... 54
- A-6. Port Attributes on Leaf Modules ..... 55
- A-7. Collector Sections..... 58

# List of Examples

- A-1. Several Variable Declarations..... 32
- A-2. Pre- and Post- Increment ..... 37
- A-3. Use of the `initialized` Expression ..... 38
- A-4. A Simple `for` loop ..... 41
- A-5. Userpoint Declaration and Use..... 46
- A-6. Incorrect Port Indexing..... 49
- A-7. Corrected Port Indexing ..... 49

# Preface

This book describes how to use LSE. It includes information on LSS, writing configurations, using the various APIs available to code points, using the emulator and debugging.

## Typographical conventions used in this book

The following typefaces are used in this book:

- Normal text
- *Emphasized text*
- The name of a program variable
- The name of a constant
- **The name of an LSE module**
- **The name of a package**
- *The name of an domain class*
- **The name of an domain implementation**
- **The name of an attribute in a domain implementation description file**
- **The name of an emulator**
- *The name of an emulator capability*
- *The name of a module parameter*
- **The name of a module port**
- Literal text
- *Text the user replaces*
- The name of a file
- The name of an environment variable
- *The first occurrence of a term*

# Chapter 1. Using emulators

The Liberty Simulation Environment provides the ability to link *emulators* into a simulation. Emulators are *abstractions* of the architectural state and the semantics of instructions. This chapter describes how to use emulators in the Liberty Simulation Environment. The APIs, data types, and structures used with emulators are called the *emulation interface*.

The chapter begins with an explanation of general concepts about emulators. It tells a few things you need to know to use the interface successfully. It then describes how to accomplish common tasks with the emulation interface. For all the details of the emulation interface, see the chapter entitled *Emulation Interface* in *The Liberty Simulation Environment API Reference Manual*.

## Concepts

### The *emulation* interface vs. the *emulator* interface

The emulation interface described in this chapter is a "front-end" interface; this is the interface used when writing modules and configurations. It is *not* the interface presented by the emulators themselves; that interface (the *emulator interface*) is not called directly from modules or configurations (thus it is a "back-end" interface) and is described in *The Liberty Simulation Environment Developer's Manual*. LSE provides a translation layer between the front-end (emulation) interface and the back-end (emulator) interface. This translation layer provides a level of indirection and function renaming and allows multiple emulators to be supported in the same simulation.

## Capabilities

Emulators are not all alike; LSE is able to support emulators with differing services and levels of detail. For example:

- The level of control of instruction execution can vary. For example, some emulators may only provide an interface which executes the instruction atomically. Others may provide interfaces allowing different parts of the instruction to be executed at different times.
- The amount of information provided by the emulator can vary. For example, some emulators will provide detailed information about all state read and written by an instruction; others will not.
- An emulator need not be complete. Some emulators may leave difficult microarchitecture-dependent semantics (e.g. register windowing) up to the microarchitecture simulator. In such cases, the configuration must include modules and code which can "fill-in" the behavior.

Because there are so many variations to the services provided by an emulator, the functionality of emulators is broken up into units called *capabilities*. A capability is simply a name for a specific piece of functionality; its presence indicates that a particular set of datatypes, data structure fields, and API functions is available for use. An essential part of any emulator's documentation is a listing of what

capabilities it supports. This chapter also indicates which datatypes, fields, and API functions are provided by each capability.

## Instructions

The basic unit of abstraction of semantics in emulators is the *instruction*. Almost all emulation API calls include a reference to a structure holding information about an instruction. The exact definition of an instruction is intentionally vague; it can be understood in the traditional sense of "an individual command"<sup>1</sup> or as a set of state updates that are related.

Instructions usually pass through several common steps:

fetch	get the instruction from instruction memory
decode	determine instruction characteristics
opfeth	fetch instruction source operands (input state)
evaluate	determine results (values to place in output state) of instruction
writeback	make results visible to later instructions (update output state at least temporarily)
commit	make results permanent (make output state updates permanent)

This simple linear order is not appropriate for all architectures; for example, some architectures may require some input state to be read before decoding due to features such as mode bits or rotating registers. Each emulator may merge, subdivide, or reorder the steps as necessary for the ISA being emulated. However, all emulators are required to provide a division of these steps into a "frontend" corresponding roughly to "fetch and decode", a "backend" corresponding to roughly to "operand fetch, evaluate, and writeback", and a "commit" step.

**Update of memory state:** The way in which emulators handle update of memory state can be confusing. Emulators often update this state as part of the writeback step, *not* the commit step. This contrasts with the normal practice of microprocessors, which is to delay writeback of memory (i.e. stores) until the commit stage. This confusion occurs in part because of the use of the same names for emulation steps and microprocessor pipeline stages when the semantics are slightly different. For an emulator, writeback is "making results visible to later instructions" and commit is "making results permanent", which are two distinct operations. Microprocessors rarely can separate the two operations for stores (because they usually can't have speculative memory state) and thus usually delay the writeback operation until it can be done with the commit. Note, however, that microprocessors often do make stores visible *to later loads in the same processor* through use of a store buffer, which is a form of speculative memory state.

So what does this mean practically? If you are performing a simulation where there is no sharing of data between processors and there is bypassing from the store queue, and the emulator supports speculation (see the Section called *Handling speculation*), you should be able to just writeback your memory operands at writeback and all will be well. If these conditions do not apply, you will need to skip the writeback step and use the *operandval* capability to writeback memory result operands when they actually need to become visible. If the time when a memory result needs to become visible varies (e.g., you have both internal store bypassing and external sharing of data), you will need to make it visible at the later time and supply the value explicitly to instructions which would see it before that time by appropriate manipulation of the operand pointers in instruction information structures. (See the Section called *Manipulating operand values*.)

## Contexts

Instructions operate within some *execution context*. A context is simply a name for the set of state available for an instruction to operate upon. Some emulation API calls include explicit references to a context, but generally once an instruction instance has been created, the context is implicit in the instruction reference.

There are two kinds of contexts in LSE. The first kind of context, a *hardware context*, corresponds roughly to the actual state elements in the simulation model. A uniprocessor simulation would have a single hardware context; multiprocessor simulations would typically have as many hardware contexts as there are processors. Shared state is represented by hardware contexts which have some state in common.

The second kind of context is a *software context*. These contexts exist because emulators may emulate operating-system functionality. Such functionality often includes virtualization of the processor and other resources. When this functionality is present, emulators can maintain more contexts than simply the hardware contexts; these software contexts correspond to processes or threads.

LSE maintains a mapping of software contexts to hardware contexts. References to hardware contexts are dereferenced to access the mapped-in software context. The mappings are usually modified by emulators, but the mappings can also be manipulated by modules or configurations. Mappings can change during the course of the simulation (this is called a *context switch*), but the mapping for a given dynamic instruction instance is set at the time that the instruction instance is created.

Context mappings are also used to determine when to terminate simulation. Simulation normally terminates when all hardware contexts have no software context mapped to them.

### TO DO

Explain better how termination works with emulators and other domains as well...

## State spaces

The state available in a context is divided into a set of *state spaces*. These state spaces provide names for pieces of state as well as differing semantics for different kinds of state (e.g. registers vs. memory). Sharing of state happens at a state space granularity. Emulators are responsible for defining the state spaces available to a context.

## Using the emulation interface

### Declaring the emulator in lss

Emulators are a particular kind of domain class and, as such, are declared to lss in the same way as other domain classes. The domain class name for emulators is *LSE\_emu*.

An emulator implementation is generally named for the ISA which it supports. Thus, the emulator supplied with LSE for the Intel IA64 architecture is **LSE\_IA64**.

To apply an emulator to a simulation, put the following code at the head of your configuration file:

```
using LSE_emu;                                ❶
var emu = LSE_emu::create("inst0", "LSE_IA64",  ❷
                          "command argument list")
    : domain ref;
add_to_domain_searchpath(emu);                ❸
```

- ❶ Bring the *LSE\_emu* domain class into scope.
- ❷ Create an emulator instance named *inst0* with implementation **LSE\_IA64**. The final argument gives command-line arguments for the emulator which will be presented to it at run-time; allowing a configuration to set "default" command-line arguments for the final simulator.

**Note:** The use of the final argument is not yet implemented.

- ❸ Add this emulator instance to the domain search path for all module instances below this instance (in this example, the top-level).

References to emulator types can be made using the LSS package syntax, e.g.,

`LSE_emu::SIM_emu_addr_t`. References to a particular emulator instance's implementation of an emulator type can be made using a function-call like syntax: `LSE_emu::SIM_emu_addr_t(emu)`.

### Warning

Multiple emulators can be instantiated by repeated use of the above syntax. However, support for multiple emulator instances is not yet implemented and will result in link-time errors.

### Datatypes

The emulation interface provides several datatypes to represent common datatypes in ISAs or information about instructions. Some of these datatypes (such as the datatype for target addresses) are specified by the underlying emulator. Others of the datatypes are constructed based upon the capabilities

of the underlying emulator; for example, emulators which do not provide information about branch targets do not have a field to record that information in their instruction information structure.

The following is a list of the most useful datatypes provided by the emulation interface. For a complete list, including information about what capabilities are required for a certain type or structure field to be present, see the chapter entitled *Emulation API* in *The Liberty Simulation Environment API Reference Manual*.

- **LSE\_emu\_addr\_t** is an address in the ISA.
- **LSE\_emu\_contextno\_t** is a global context number type. Context numbers greater than zero are hardware contexts; context numbers less than zero are software contexts. Zero indicates "no context".
- **LSE\_dynid\_t** is the ubiquitous dynamic message identifier type. This structure should only be accessed using accessor functions (e.g. `LSE_dynid_get`).
- **LSE\_emu\_instr\_info\_t** contains information for a dynamic instance of an instruction. This information includes the the address, decode information, operand information, address of the next instruction to execute, operand values, and results of the instruction (potentially including intermediate results). This structure should only be accessed using accessor functions (e.g. `LSE_emu_instr_info_get`).
- **LSE\_emu\_instrstep\_name\_t** is an enumerated type whose values are the evaluation step names for an emulator. For example, if there is an instruction step named "readmem", there is an value `LSE_emu_instrstep_name_readmem`.
- **LSE\_emu\_operand\_info\_t** contains information about instruction operands. This information includes whether the operand is needed for the instruction, whether it is an immediate, the state space identifier and address for the operand and the starting location and ending location within the register. Accessor macros are not needed for this structure.
- **LSE\_emu\_operand\_name\_t** is an enumerated type whose values are the operand names for an emulator. For example, if there is an operand named "left", there is an value `LSE_emu_operand_name_left`.
- **LSE\_emu\_operand\_val\_t** contains information about instruction operand values. This information includes whether the operand value is valid and its value. Accessor macros are not needed for this structure.
- **LSE\_emu\_spaceaddr\_t** is a union type which can hold addresses within state spaces. The fields have the names of the state spaces for the particular emulator. There is also a default field named `LSE`.
- **LSE\_emu\_spaceid\_t** is an enumerated type whose values are the state space identifiers for an emulator. The names of the values are the names of the state spaces. For example, if there is a state space named GR, there is a value `LSE_emu_spaceid_GR`.
- **LSE\_emu\_spacetype\_t** is an enumerated type defining possible state space types. Its possible values are:
  - `LSE_emu_spacetype_creg`
  - `LSE_emu_spacetype_mapping`
  - `LSE_emu_spacetype_mem`
  - `LSE_emu_spacetype_reg`

## Dealing with multiple emulator instances

Datatypes depend upon the underlying emulator instance. For example, **LSE\_emu\_addr\_t** represents

addresses in a target ISA. For a 32-bit ISA, it would be a 32-bit integer, but for a 64-bit ISA it would be a 64-bit integer. When there is more than one emulator instance in a particular simulator, (e.g. when simulating a multiprocessing system with heterogenous processors), you cannot simply use a type name such as `LSE_emu_addr_t`; to which emulator's address type does it refer?

LSE attempts to infer the emulator instance you wish to use; the normal algorithm is to use the domain search path (naturally, as emulators are a domain class). What this means is that the domain search path is searched for domain instances which define the identifier in question. The domain search path is inherited from the parent module in the module instance hierarchy, but can be prepended to by any particular module. And, of course, code inside `lss` triple-angle-brackets is evaluated with the search path of the final module in which it is placed.

When you do not wish to use the domain search path, use the domain instance notation. For types and constants, this is:

```
LSE_emu_addr_t([emulator instance name])
```

The square brackets are required. The *emulator instance name* must be a literal parameter.

Similarly, API functions can be qualified with the emulator instance name and must be qualified if LSE cannot infer the emulator to use. You use the same syntax as for types:

```
function_name([emulator instance name])(arguments)
```

Using an emulator instance name when one is not allowed will result in odd errors at code generation or code compilation time.

## Things to watch out for

### Open Issue

- Emulation interface calls often have side effects on state; care is needed w.r.t. multiple invocations of modules/control points calling them.
- Cross-instruction semantics rules

## The most basic tasks

### Assigning an execution context

All instruction execution takes place within a context. In order to create an instruction instance, modules must have a hardware context number. This context number may be provided by a parameter or a constant; parameters are recommended.

Hardware context numbers are positive integers starting with 1. Hardware contexts must be created before they are used. Hardware context numbers should be assigned without "skipping" in the numbering scheme for best performance.

Hardware contexts will normally have a software context mapped to them, if possible, as they are created. Ways of preventing this and controlling the process in much finer detail are described in the Section called *Advanced context handling*.

Software context numbers are negative integers starting at -1. 0 is "no context". Software context numbers should also be assigned without "skipping" in the numbering scheme for best performance.

Note that both software and hardware context numbers use the same datatype, but have non-overlapping values. This unusual numbering scheme allows emulation API functions to accept either software or context numbers as parameters.

Not all modules will need to know the context. If a module simply operates on an instruction that it received upon an input port, the information about the context is already present in the instruction information structure. In general, only modules which create new dynamic message IDs or use emulation API calls which do not have a dynamic message ID as an input parameter will need to determine the execution context.

## Finding the first instruction to execute

Once it has a hardware context number, a module may need to find the first instruction to execute in this context. This is done by calling `LSE_emu_get_start_addr` in the following fashion:

```
LSE_emu_addr_t addr;
LSE_emu_contextno_t cno;
...
addr = LSE_emu_get_start_addr(cno);
```

This function need not return the same value after API function calls which cause parts of an instruction to execute, as emulators may use the context's starting address to track some internal concept of "current" instruction.

## Creating a dynamic instruction instance

When you create a dynamic instruction instance, there are two things you must do:

```
LSE_dynid_t d;
LSE_emu_contextno_t cno;
LSE_emu_addr_t a;

... // get the context number (cno) and address (a)

d=LSE_dynid_create();           ❶
LSE_emu_init_instr(d,cno,a);    ❷
```

- ❶ Create the dynamic ID structure.

- ② Notify the emulator, setting hardware context number and address. The mapping from hardware to software context numbers becomes fixed for this instruction at this time.

## Executing an instruction (simple form)

As described before, instructions conceptually pass through six steps which may be merged, split, or reordered by the emulator. The emulator is required to provide "frontend", "backend", and "commit" groupings of these steps so that it becomes possible to perform the "frontend" steps followed by the "backend" steps followed by the "commit" steps and get correct execution of the instruction.

The emulator interface provides three API functions for performing the "frontend", "backend", and "commit" steps: `LSE_emu_dofront`, `LSE_emu_doback`, and `LSE_emu_docommit`. These APIs are called the *simple form* of execution. Emulators are encouraged to make the break between the frontend and backend occur after instruction fetch and decode but before operand fetch, if possible. Emulator documentation describes where the break actually occurs and what data fields are valid at the break. The break between the backend and commit has to do with speculation (see the Section called *Handling speculation*).

Thus, to fully execute an instruction, you need only use:

```
LSE_dynid_t d;
...
LSE_emu_dofront(d);
LSE_emu_doback(d);
LSE_emu_commit(d);
```

**Note:** Not all emulators will work with just this simple interface because some emulators require notification of "time" passing between instructions or may require the microarchitectural model to manage some state. You must consult the documentation for each emulator to determine whether the simple form of execution is sufficient.

## Finding the next instruction

To execute a program, some module or modules will have to generate dynamic IDs for the entire instruction stream over the course of the simulation. To do this, they may need to know how to find the next instruction in the stream. The necessary API functions have already been introduced; simply use `LSE_emu_dynid_get` to get field `next_pc` of the instruction information to get the next address after an instruction has been evaluated, and then create a dynamic ID for the new instruction.

## Determining when a context is finished

A hardware context is finished when it no longer has a software context mapped to it. This can be determined by calling `LSE_emu_get_context_mapping`; when this function returns the special context number 0, there is no software context mapped to the hardware context.

### TO DO

Some other part of the manual should cover termination conditions for the simulation. For now, we'll just say that simulation terminates when either `LSE_sim_terminate_now` is non-zero or `LSE_sim_terminate_count` is zero or the simulator has no more scheduled time steps.

## Putting it all together

The following code snippet should be able to execute code with many simple emulators.

```
LSE_dynid_t d;
LSE_emu_addr_t addr;
LSE_emu_contextno_t cno;

// mystically determine what hardware context to use

addr = LSE_emu_get_start_addr(cno);

while (LSE_emu_get_context_mapping(cno)) {
    d=LSE_dynid_create();
    LSE_emu_init_instr(d,cno,addr);
    LSE_emu_dofront(d);
    LSE_emu_doback(d);
    LSE_emu_docommit(d);
    addr = LSE_emu_dynid_get(d,next_pc);
    LSE_dynid_cancel(d);
}
```

The above example actually has a subtle problem. The `LSE_dynid_cancel` function does not immediately release memory taken up by a dynid. As a result, the above loop will likely run out of memory. In situations where you have a potentially unbounded number of instructions like this, you should not cancel the dynid and create a new one, but instead use `LSE_dynid_recreate` to "clean out" the old dynid structure:

```
LSE_dynid_t d;
LSE_emu_addr_t addr;
LSE_emu_contextno_t cno;

// mystically determine what hardware context to use
```

```

addr = LSE_emu_get_start_addr(cno);
d = LSE_dynid_create();

while (LSE_emu_get_context_mapping(cno)) {
    d=LSE_dynid_recreate();
    LSE_emu_init_instr(d,cno,addr);
    LSE_emu_dofront(d);
    LSE_emu_doback(d);
    LSE_emu_docommit(d);
    addr = LSE_emu_dynid_get(d,next_pc);
}
LSE_dynid_cancel(d);

```

Be aware also that the above example only works for emulators which do not attempt to context switch. See the Section called *Handling context switches* for further information.

## Other basic tasks

### Disassembling instructions

Emulators with the *disassemble* can display the disassembly of instructions. This capability can be accessed by calling `LSE_emu_disassemble`. You must have a dynamic ID for the instruction, but need not have fetched or decoded the instruction.

### Accessing instruction information

Information for the instruction is placed in the instruction information structure. It is accessed using the `LSE_emu_dynid_get` macro. The different fields of the instruction typically become available at different steps of execution of the instruction; the emulator documents when they become available.

It is possible to update the instruction information structure using the `LSE_emu_dynid_set` macro. The emulator may or may not use this updated information depending upon the information and what steps of execution have already been performed. The emulator documentation should make clear what happens when instruction information is updated.

### Decoding instruction classes

Emulators offer a means of classifying instructions. This classification is stored in the instruction information structure and can be accessed via the `LSE_emu_instr_info_is` and `LSE_emu_dynid_is` function calls. An instruction may belong to more than one class.

Unfortunately, emulators do not all provide all possible classes. Indeed, some classes would make very little sense for some emulators. Furthermore, defining all "reasonable" classes is a task requiring a crystal ball. Therefore, emulators are allowed to provide any set of classes which they desire. Emulator writers are encouraged to use standard class names, which are listed below, but only the `sideeffect` class is

required. You can learn whether an instruction class is supported by an emulator by calling `LSE_emu_has_instr_class`.

**Table 1-1. Standard instruction class names**

Class name	Purpose
<code>cti</code>	The next instruction might not be the next one inline.
<code>indirect_cti</code>	The next instruction might not be the next one inline and the target is unknown from just the instruction itself and its address.
<code>load</code>	The instruction loads from memory.
<code>store</code>	The instruction stores to memory.
<code>sideeffect</code>	The instruction has a side effect which cannot be accounted for as a described state manipulation. This class is required.
<code>unconditional_cti</code>	The next instruction is definitely not the next one inline. Should also be classified as <code>cti</code> .

An example of the use of these APIs is:

```

boolean is_a_cti;
LSE_dynid_t t;
...
#if LSE_emu_has_instr_class(cti)
    is_a_cti = LSE_emu_dynid_is(t,cti);
#else
#error Attempted to use instruction class cti that emulator does not provide
#endif

```

## Determining branch targets and direction

In many situations, knowing more than just the next instruction is useful; it may be useful to know potential branch targets, inline addresses, and the direction of a branch (taken or not-taken). Emulators with the *branchinfo* capability can provide this information. The step of evaluation at which it is produced depends upon the emulator; this must be documented by the emulator. Many emulators make inline addresses and direct branch targets available as part of the decode step.

All the branch information can be obtained by using `LSE_emu_dynid_get`; the relevant fields are *branch\_dir* and *branch\_targets*. Field *branch\_num\_targets* gives the actual number of targets. The inline (not-taken) address is counted as a target and is always *branch\_targets*[0]. Unconditional branches still treat the non-taken address as target number 0; the "unconditionality" is reflected in *branch\_dir* for these instructions being always greater than 0. The maximum number of branch targets is the constant `LSE_emu_max_branch_targets`.

**TO DO**

Example

The `next_pc` field is normally the branch target in the branch direction. However, it does not have to be; when the expression

```
LSE_dynid_t id;
...
(LSE_emu_dynid_get(id,
    branch_targets[LSE_emu_dynid_get(id,branch_dir)])
    != LSE_emu_dynid_get(id,next_pc))
```

is true after the instruction has been completely evaluated, there has been a *discontinuity* in instruction execution. Emulators can cause discontinuities due to signal handlers, callbacks, or "longjmp" like instructions or OS calls. If an emulator can have such behavior, the microarchitectural model must use the above expression to detect it and respond accordingly. Emulator documentation states whether a discontinuity can occur.

## Comparing the age of instructions

Many schemes for detecting dependencies between instructions rely upon comparing older instructions vs. newer instructions, where age is the position in "program order". While this information is often implicit in "where" in a microarchitectural structure an instruction is (e.g., older instructions are closer to the tail of queues), it can be useful to simply compare the age of two instructions. This is done by comparing the `idno` fields of the dynamic message identifier:

```
LSE_dynid_t a, b;
boolean a_olderthan_b;
...
a_olderthan_b = LSE_dynid_get(a,idno) < LSE_dynid_get(b,idno);
```

## Obtaining state space information

There are several API functions which return information about state spaces as provided by the emulator's description file. They are:

- `LSE_emu_get_statespace_name` - returns a string with the state space name.
- `LSE_emu_get_statespace_size` - returns the number of locations in the state space, if it is less than  $2^{31} - 1$ .
- `LSE_emu_get_statespace_bitsize` - returns the number of bits needed to address the state space.
- `LSE_emu_get_statespace_type` - returns the state space type.
- `LSE_emu_get_statespace_width` - returns the width of locations in the state space.
- `LSE_emu_statespace_has_capability` - Does the statespace have a particular capability?

There is also a constant named `LSE_emu_num_statespaces` which is the number of state spaces in the emulator.

**TO DO**

Write an example.

**Detecting register-carried data dependencies**

Register-carried data dependencies between instructions can be detected when the emulator implements the *operandinfo* capability. This capability indicates that the emulator provides information about the source and destination operands of an instruction. These operands are typically the register and immediate operands. They do not generally include memory operands, unless the address is known statically.

Operand information is generally provided when the decode step is performed. It contains only information about which state is accessed, but *not* operand values. The operand information is stored in arrays of type `LSE_emu_operand_info_t` within the `LSE_emu_instr_info_t` structure; the field names and formats are described later.

Emulators provide "names" for the entries in the operand information arrays; these names describe what the operands are intended for. For example, a simple DLX-style architecture might have source operands named "Left" and "Right" and a single destination operand named "Result". The choice of names is left to the author of the emulator; there is no enforced standardization of names.

Operand names are provided as values of the enumerated `LSE_emu_operand_name_t` and have the form `LSE_emu_operand_name_emulator-supplied-name`. For example, in the simple DLX-style architecture mentioned above, the names would be `LSE_emu_operand_name_Left`, `LSE_emu_operand_name_Right`, and `LSE_emu_operand_name_Result`.

The operand information is supplied in two fields added to `LSE_emu_instr_info_t`:

- `operand_src[LSE_emu_max_operand_src]` - array of source operand information. These are operands which are read by the instruction.
- `operand_dest[LSE_emu_max_operand_dest]` - array of destination operand information. These are operands which are written by the instruction.

The information for each operand is a `LSE_emu_operand_info_t` structure. This structure has fields for the state space number (*spaceid*), address within the state space (*spaceaddr*), and operand usage information (*used*). The usage information is a union; for register accesses, the relevant field is (*uses.reg.bits*).

Not all operand names need refer to registers; memory operands, immediate operands, and unused operands (i.e. this instruction uses less than the maximum number of operands) may all be present. Register accesses can be distinguished from memory accesses either through emulator-specific convention about how operand names are used or through the `LSE_emu_get_statespace_type` function.

Unused operand names for a particular instruction are marked with *spaceid* equal to 0 and *spaceaddr.LSE* equal to 0. Immediate operand names are marked with *spaceid* equal to 0 and *spaceaddr.LSE* not equal to 0. Some destination operands may also not be registers or memory

accesses. These are marked as immediates with *spaceid* equal to 0 and *spaceaddr.LSE* not equal to 0.

There are three additional API function calls which may be of use. The first, `LSE_emu_spaceref_eq`, compares two state addresses *already determined to be in the same space* to see whether they are equal. This function must be used for equality testing because the state space addresses can have varying numbers of bits or can even be strings. The second, `LSE_emu_spaceref_is_constant`, returns whether a particular register is a constant, as general register 0 is in many ISAs. The third, `LSE_emu_spaceref_to_int`, maps a state space address to an integer.

The following code segment compares two dynamic instructions to see whether there are any read-after-write (RAW) or write-after-write (WAW) register dependencies between them, ignoring the exact bits involved and dependencies after writing a constant register:

```
int i,j;
LSE_dynid_t firsti, secondi;
LSE_emu_operand_info_t firstop, secondop;
...
// find RAW and WAW dependencies
for (i=0 ; i < LSE_emu_max_operand_dest ; i++) {

    firstop = LSE_emu_dynid_get(firsti,operand_dest[i]);

    // immediates, irrelevant, and constant registers do not form dependencies
    if (firstop.spaceid <= 0 ||
        LSE_emu_get_statespace_type(firstop.spaceid)!=LSE_emu_spacetype_reg ||
        LSE_emu_spaceref_is_constant(firstop.spaceid,firstop.spaceaddr))
        continue;

    // look for RAW
    for (j=0 ; j < LSE_emu_max_operand_src ; j++) {
        secondop = LSE_emu_dynid_get(secondi,operand_src[j]);
        if (LSE_emu_spaceref_eq(firstop.spaceid, firstop.spaceaddr,
                               secondop.spaceid,secondop.spaceaddr)) {
            ... ; // process RAW
        }
    }

    // look for WAW
    for (j=0 ; j < LSE_emu_max_operand_dest ; j++) {
        secondop = LSE_emu_dynid_get(secondi,operand_dest[j]);
        if (LSE_emu_spaceref_eq(firstop.spaceid, firstop.spaceaddr,
                               secondop.spaceid,secondop.spaceaddr)) {
            ... ; // process WAW
        }
    }
}
}
```

## Obtaining memory access information

You may wish to find out details about data memory accesses performed by an instruction. These details can include the effect address of the access, the size of the access, and flags indicating the type of access and attributes such as atomicity. Emulators with the *operandinfo* capability may provide this information, but are not required to. The information is stored within the `LSE_emu_operand_info_t` structure. The exact offset within this structure is emulator-dependent.

The address of the access is stored in the *spaceaddr* field of the operand. Access size and flags describing the access appear in the *uses* field of the operand in sub-fields named *mem.size* and *mem.flags*, respectively. There are a few pre-defined flag values; additional values are emulator-dependent.

The pre-defined flag values are:

**Table 1-2. Memory access flags**

Flag name	meaning
<code>LSE_emu_memaccess_read</code>	The access is a read. This can usually also be implied by whether the access is reported in the source or destination operands of the instructions.
<code>LSE_emu_memaccess_write</code>	The access is a write. This can usually also be implied by whether the access is reported in the source or destination operands of the instructions.
<code>LSE_emu_memaccess_atomic</code>	The access is atomic with respect to some other access in the instruction.
<code>LSE_emu_memaccess_noaccess</code>	No actual access is required; prefetches and probe instructions might set this flag.

You may wish to obtain the access information without actually performing the accesses. For example, you may be simulating a multi-processor, and the exact timing of accesses will affect the data values seen. This can only be accomplished if the emulator has broken the evaluation step (for loads) and writeback or commit step (for stores) into a part that computes access information and another which performs the access.

## Detecting memory-carried data dependencies

Memory-carried data dependencies (i.e. data dependencies between load and store instructions) can be discovered when the emulator supplies memory access information as discussed in the previous section.

## Advanced context handling

### Handling context switches

Some emulators may perform context switches — changes of the software-to-hardware context mappings. A context switch can be detected by comparing the software context (field *swcontextno*) of a particular instruction with the current mapping:

```
LSE_emu_contextno_t hcno; // hardware context number
LSE_dynid_t tid;
boolean contextswitched;
... // tid created with hardware context hcno.
contextswitched = (LSE_emu_get_context_mapping(hcno) !=
                  LSE_emu_dynid_get(tid, swcontext));
```

Emulators attempt to update the starting address of a context when it is switched out so that later calls to `LSE_emu_get_start_addr` for the old context will return the next instruction to be executed in that context. Usually, the assumption is that if an instruction X caused the context to be switched out, the instruction after X should be the next instruction in the context. The assumption made depends upon the emulator.

Emulators cannot in general successfully update the starting address if a context is switched out because of some instruction executing in another context or some microarchitectural event. This is because emulators do not have any idea of the semantics of microarchitectural events with respect to "outstanding" vs. "completed" instructions (if it even tracks such state, which most do not). In such cases, the microarchitectural model must update the starting address by using `LSE_emu_set_start_addr`. A software context number should be used as an input parameter to this function rather than a hardware context number if the software-to-hardware mapping has already been changed when the function is called.

### Controlling context mapping

Mapping between software and hardware contexts normally occurs automatically as contexts are created. It is possible to prevent this mapping for a particular context by setting a flag called the *automap flag* for the context to `FALSE`. The initial value for this flag is provided as a parameter when the context is created. It can be changed by calling `LSE_emu_set_context_automap`. Note that when the automap flag is changed from `FALSE` to `TRUE` for a context, the underlying emulator may immediately map the context.

The mapping itself can be directly set by calling `LSE_emu_map_context`. This function takes a software context number and a hardware context number as parameters and maps them to each other.

### Creating and destroying software contexts

Software contexts are normally created by the initialization code of the simulator main program (which knows about command-line arguments) or by instructions in the emulators. However, they can be created by modules.

Software contexts are created by calling `LSE_emu_create_context`. This function takes two parameters: a context number and the initial automap flag for the context. context participates in automatic context mapping. To create a software context, a software context number should be used. If the context number exists, the function works like `LSE_emu_set_context_automap`; existing state values in the context are not lost. An unused context number can be found by calling `LSE_emu_get_contextno`.

Programs can be loaded into software contexts by calling `LSE_emu_load_context`. This function also sets the starting address of the software context; the starting address can be set explicitly with `LSE_emu_set_start_addr`.

Software contexts cannot currently be destroyed.

## Creating and destroying hardware contexts

Hardware contexts are also normally created by the initialization code of the simulator main program. They can also be created by modules.

Like software contexts, hardware contexts are created by calling `LSE_emu_create_context`, but the context number supplied as a parameter needs to be a hardware context number. If the context number exists, the function works like `LSE_emu_set_context_automap`; existing state values in the context are not lost. An unused context number can be found by calling `LSE_emu_get_contextno`.

Hardware contexts cannot be destroyed.

## An example of context mapping

This example creates two hardware contexts and a software context and maps the software context to the second hardware context, leaving the first hardware context free to participate in automatic mapping.

```
int argc;
char *argv[], **envp;
...
LSE_emu_create_context(1, TRUE);           ❶
LSE_emu_create_context(2, FALSE);        ❷

LSE_emu_create_context(-1, FALSE);        ❸
LSE_emu_load_context(-1, argc, argv, envp);  ❹

LSE_emu_map_context(2, -1);               ❺
```

- ❶ Create a hardware context with automatic mapping.
- ❷ Create a hardware context without automatic mapping.
- ❸ Create a software context without automatic mapping.
- ❹ Load a program into the software context.
- ❺ Map the first software context to the second hardware context.

## Sharing state

### Open Issue

While it's easy enough to specify an API for sharing a state space, I'm not sure how to go about specifying sharing of portions of an state space or doing things like having two contexts with shared memory with different addresses. Implementation-wise, I've some ideas how to do it, but what's a reasonable API for this? In the meanwhile, emulators can certainly share state in whatever form they have implemented.

## More complex tasks

### TO DO

- Exceptions

## Executing an instruction (detailed form)

An earlier section presented the simple form of instruction execution. In the simple form, execution was split into "front end", "back end", and "commit" steps. This section introduces the more complex form, which allows finer-grained steps to be executed.

Emulators divide up execution into whatever number of steps (at least two, however) the emulator writer desires. These steps are each given names. The enumerated type `LSE_emu_instrstep_name_t` has values which correspond to these names. The values have the form `LSE_emu_instrstep_name_step`. For example, if there is a step named "memread", there would be a value `LSE_emu_instrstep_name_memread`. There is also a constant `LSE_emu_max_instrstep` which is the maximum instruction step name value plus one.

An example set of names might be: `fetch`, `decode`, `opfetch`, `alu`, `memaccess`, `writeback`, `commit`.

The exact meanings of the steps are left up to the emulator, but typically correspond to stages of instruction execution. Not all instructions may pass through all steps; attempting to execute a step which is not defined for an instruction is legal and the emulator just ignores the attempt. Some step names may be aliases for one another for convenience in describing different instructions. Executing all distinct step numbers in ascending numerical order results in correct execution for emulators which are able to correctly and completely execute instructions one at a time.

There may be data dependencies between different steps of execution. If these data dependencies are violated, the behavior of the emulator is undefined and may include crashing, though emulators are encouraged to provide a "debug" mode where data dependencies are checked. To make the data dependencies manageable by "generic" code, the value assignments for step names must be such that performing the steps in order by value is a valid execution.

The API function which performs a step is `LSE_emu_do_instrstep`.

The following code snippet should give correct execution for all emulators which do not have cross-instruction dependencies and which have implemented complete instruction behavior:

```
LSE_dynid_t instr;
int i;
...
for (i=0 ; i < LSE_emu_max_instrstep ; i++) {
    LSE_emu_do_instrstep(instr,i);
}
```

The following code snippet performs the execution step named "readmem":

```
LSE_dynid_t instr;
...
LSE_emu_do_instrstep(instr,LSE_emu_instrstep_name_readmem);
```

## Manipulating operand values

It is often useful to be able to both inspect and change individual operand values which the emulator uses. When the *operandval* capability is present, this can be done.

When the *operandval* capability is present, there is an additional type for operand values. This type is called `LSE_emu_operand_val_t`. It contains two fields. The first field, named *valid* is simply a "valid" flag indicating that the operand value is valid. The second field, named *data*, is nearly always a union type of the different kinds of operand values possible in the emulator.

How operands are manipulated is best understood by considering source, destination, and intermediate operands separately.

### Source operands

Source operands are those that read from state. The instruction information structure has a field called *operand\_val\_src* which is an array of source operand value structures (of type `LSE_emu_operand_val_t`). These values are filled in (fetched) during the steps of operand execution. After a value has been filled, later steps of execution use the value from the operand value array.

You may read and modify the operand value in the instruction information structure using the accessor macros for instruction information.

Individual operands can be fetched into the operand value array using the `LSE_emu_fetch_operand` API function. The following code snippet fetches only the source operands named "reg1" and "reg2":

```
LSE_dynid_t instr;
...
```

```
LSE_emu_fetch_operand(instr,LSE_emu_operand_name_reg1);
LSE_emu_fetch_operand(instr,LSE_emu_operand_name_reg2);
```

Fetching individual operands does not prevent instruction steps from fetching them again at a later time.

Another API function, `LSE_emu_fetch_remaining`, fetches all source operands which have not yet been fetched (i.e., those whose `valid` flags are `FALSE` for an instruction).

Some emulators may require that certain operands be fetched before others (for example, a rotating register base must be fetched before fetching source registers that can rotate). Such cases are documented by the emulators; violating these dependencies causes undefined results.

This description has assumed that all operands can be manipulated in this fashion. This is rarely the case; emulator writers choose which source operands to make visible or modifiable. For operands which are not reported in this fashion, the values in this array will never become valid (though the `valid` flag may be set). For operands which are not modifiable, any changes to the reported values will be ignored.

## Destination operands

Destination operands are those that write to state. The instruction information structure has a field called `operand_ptr_dest` which is an array of destination operand value structures (of type `LSE_emu_operand_val_t`). All instruction steps which calculate a destination operand value place the value in this array.

You may read and modify the operand value in the instruction information structure using the accessor macros for instruction information.

Operands can be individually "written back" to state using the `LSE_emu_writeback_operand` API call. This function makes later read accesses to the state referenced by the named operand return the new value. It can thus be seen as updating "current" state. This update may or may not be permanent if speculation is supported by the emulator; see the Section called *Handling speculation*.

There is also a field in the instruction information structure called `operand_written_dest`. This field is an array of flags indicating that a particular destination operand has been written back. The `LSE_emu_writeback_operand` function sets the flag to true as a side effect of the writeback. Another API function, `LSE_emu_writeback_remaining`, writes back all destination operands for which this flag is not set.

A common use of individual control of writeback is to write back registers at the "writeback" stage of a pipeline, while delaying writeback of memory to the "commit" stage. The following code snippet writes back all register operands:

```
LSE_dynid_t instr;
LSE_emu_operand_info_t opinfo;
...
for (i = 0; i < LSE_emu_max_operand_dest; i++) {
    opinfo = LSE_emu_dynid_get(instr,operand_dest[i]);
    // assume it is a register when a destination has a spaceid; could actually
    // check the space type
    if (opinfo.spaceid>0 &&
        LSE_emu_get_statespace_type(opinfo.spaceid)==LSE_emu_spacetype_reg) {
        LSE_emu_writeback_operand(instr,i);
    }
}
```

}

This description has assumed that all operands can be manipulated in this fashion. This is rarely the case; emulator writers choose which destination operands to make visible or modifiable. For operands which are not reported in this fashion, the values in this array will never become valid (though the *valid* flag may be set). For operands which are not modifiable, any changes to the values will be ignored.

#### *Interactions with the speculation capability*

See the Section called *Handling speculation* for information about this.

## Intermediate operands

Intermediate operands are those that are transitory, existing only for the duration of instruction execution. Intermediate operand values are not accessed using pointers; instead, the values themselves are stored in the instruction information structure. The field used is called *operand\_val\_int* and is an array of *LSE\_emu\_operand\_val\_t* of width *LSE\_emu\_max\_oper\_int*.

You may look at and modify internal operands at any time, bearing in mind the instruction steps in which they are calculated and used.

## Other considerations

It is important to bear in mind that manipulation of operand values is heavily dependent upon the emulator. You must understand when the values become available and how they are used. *Always* consult the emulator documentation before using this capability.

## Handling speculation

Speculation is another very important technique. For the purposes of this section, speculation is performing any step of an instruction's execution that modifies *emulator* state when that instruction might not commit. Modification of microarchitectural state (such as cache contents) in the presence of speculation is up to the microarchitectural model to manage.

There are many sources of speculation. The most obvious one is control speculation, where instructions modify state before a branch resolves. Another is data speculation, where instructions modify state using operands that are not certain to be correct. Another important source we call exception speculation; this is modifying state while a previous instruction could still signal a precise exception.

The key issue for handling speculation is how to get a *consistent* state after mis-speculation has occurred. The definition of consistent varies from architecture to architecture and even from condition to condition. For example, an architecture may require precise recovery from branch misprediction (that is very normal), but an imprecise recovery from floating point exceptions. Here precise means that the state of the machine after the mis-speculation is handled is as if all instructions before some instruction have committed and all instructions after it have not been executed at all.

LSE's view of speculation recovery is that there is some notion of a "current" state and a "permanent" state. LSE always assumes that any instruction operates on the current state, but updates the current and permanent state at different times. Emulators may use different methods to maintain this separation of state; the exact method is not relevant to the use of the emulator. If an emulator can support speculation recovery, it has the *speculation* capability.

To "undo" the effects of speculation, use the API call `LSE_emu_rollback_dynid`. Because the standard LSE modules often use messages of type `LSE_resolution_t` to indicate mis-speculation, there is also a function `LSE_emu_rollback_resolution` which rolls back in bulk the instructions dependent upon the mis-speculated instruction.

When rolling back many instructions, you should roll back in a reverse data dependency order (i.e. the youngest dependent instructions first). Reverse program order should always be safe.

`LSE_emu_rollback_resolution` rolls back its instruction list in reverse order. It is possible to selectively rollback instructions when imprecise handling of mis-speculation is preferred.

After `LSE_emu_docommit` has been called (when using the simple form of execution) or any of the "commit" steps has been performed (when using the detailed form of execution), it is no longer legal to call rollback API functions; undefined results will occur if you do so. Until one of these functions has been called, it is possible to call a rollback API function, unless the instruction is marked (from decode) as having side effects. Rollback APIs can never be called for instructions which have side effects. Some emulators may have further restrictions on the order in which rollbacks or commit can occur; emulator documentation describes these restrictions. Also, some emulators may not require the commit steps; again, emulator documentation states this.

### Warning

Emulators do not always provide speculation abilities for all the state they modify. For example, it is very difficult (when it's not impossible!) to speculatively perform I/O operations. Emulators always mark instructions with such problems as belonging to the `sideeffect` instruction class. In your microarchitectural model you must ensure that such instructions are not executed speculatively.

Store instructions executed speculatively can also cause problems when an emulated I/O device is memory-mapped. The emulator may not be able to classify such stores as having side effects. The configuration must check the effective address if this is possible and not speculatively execute such stores. Of course, in real hardware such stores should not be executed speculatively either.

### Issues with imprecise speculation recovery

While imprecise recovery may be allowed, there are some situations in which it may be extremely difficult to model correctly (or even build correctly). The basic problem is encountered when multiple writers of some state are allowed to be "in flight" and an earlier one (in program order) is cancelled while a later one is not, and the later one has already executed. In such a case, the current state should *not* be rolled back.

One common case in which this occurs is with "sticky" bits in status registers, such as those mandated by IEEE-754. Emulators may choose to not distinguish between current and permanent state for such bits, but this means that *no* recovery of speculative updates to this state is possible for such emulators.

Another case arises when register renaming is part of the microarchitecture. Suppose that you have two writes to register r4 in a machine that performs register renaming. Both writes are able to proceed because of renaming. Suppose now that the first instruction is cancelled without cancelling the second instruction. The cancellation restores an old value of r4, but the intervening write to r4 has made the rollback obsolete.

### Open Issue

What should we do about this problem?

A linked-list of values would do the trick where rollback deletes an old value from inside the list (and commit would be required as well, clipping off earlier values), but would be very expensive in execution time and doesn't scale well to memory instead of registers.

Actually, we can solve both problems by having the microarchitecture maintain the register state itself and fill in operands appropriately, thus never making any rollback records in the emulator, but this is terribly heavy-duty work for the configurer. Is there a better way?

Perhaps we need finer-grain access to rollback records?

## Creating proper instruction ordering

Relaxing of instruction ordering is a key microarchitectural technique. The level of modeling detail you can achieve will depend upon the interaction between the capabilities of the emulator and the microarchitecture to be modeled. The key issue is here is the fact that emulators generally maintain only one "current" copy of architectural state for a given context, but various microarchitectural techniques can have multiple in-flight values for any given state. This is closely related to the issues described previously for speculation, but there the question was a question of difference between "current" and "permanent" state. This section describes how to perform modeling of various types of ordering schemes with different levels of detail.

Some ordering schemes require use of both the detailed form of execution and the *operandval* capability; others require only the simple form of execution and the base emulation interface. It is always possible to use the more detailed forms of execution and the *operandval* capability in situations when the simpler method applies. You might wish to do this to make your model "more realistic", even though the simulation results are the same, at the cost of more complexity.

## Modeling just timing

The timing of many in-order or out-of-order machines can be modeled with just the base emulator interface and the simple form of execution. The way to accomplish this is to ensure that `LSE_emu_doback` is called in program order. This is most easily done at the decode stage. (Note that this technique has been used in other simulation systems, most notably SimpleScalar.) You probably cannot do this at the commit stage (the other place where instructions are in program order), because most microarchitectural models need some instruction results (such as branch resolutions) before commit.

The timing which you create in the microarchitectural model can take into account structural hazards through proper routing and flow control. Scheduling of writeback and bypass paths can be part of this,

even though the paths themselves are not carrying "real" data. Instruction dependencies can be used when the *operandinfo* capability is provided by the emulator.

This technique will not produce accurate timing results if instruction results depend upon timing. For example, the value returned by a load in a multiprocessor can depend upon the timing if some other processor could write to that location. Execution of the load at decode could give different results than execution of the load at the proper time. This could then lead to a different execution path for the program.

Correct execution of a model with this technique should not be taken as strong evidence that the scheduling algorithms modeled are working correctly, as errors in scheduling of instructions will not affect the results of the target program. The only errors that can be found are those where the scheduling logic is never notified of the completion of an instruction (in which case dependent instructions never issue and eventually the instruction window fills).

### Fully-interlocked scheduling schemes

It is possible to perform the backend stages of an instruction in the execution stages of a pipeline using just the base emulator interface and the simple form of execution if the microarchitecture includes interlocks to prevent execution of an instruction if it has data dependencies with previously-issued but not yet committed instructions. These data dependencies are the standard ones: read-after-write, write-after-read, and write-after-write. Many simple in-order pipelines will satisfy these requirements. A simple scoreboard arrangement meets these requirements; the improved scoreboard scheme, Tomasulo's algorithm, and register renaming schemes do not, as they either read available operands early to relax WAR hazards or rename operands to relax WAW hazards. A microarchitectural model which does meet the requirements can call `LSE_emu_doback` at the time that instructions are to execute.

Care must be taken if the microarchitectural model includes bypasses to ensure that `LSE_emu_doback` is called for the instruction at the source of a bypass before it is called for the instruction at the destination of that bypass. Typically this requires explicit modeling of the potential bypass paths (i.e., there are connections from the writeback stage to the execute units which must not be "unknown" before the execute units exist, though data need not be carried), or port queries in the input control functions of the execute units.

The timing which you create in the microarchitectural model can take into account structural hazards through proper routing and flow control. Scheduling of writeback and bypass paths can be part of this, even though the paths themselves are not carrying "real" data. Instruction dependencies can be used when the *operandinfo* capability is provided by the emulator.

The example given in the previous section where load results depend upon timing does not cause difficulties when this technique is used, as the load can be executed at the correct time. However, correct ordering of memory references may require that the emulator provide information about memory accesses so that the microarchitectural model knows about address conflicts.

Correct execution of a model using this technique is fairly good evidence that the scheduling algorithm modeled is working correctly; instructions executed at the wrong time will see incorrect operand values or write results in an incorrect order.

## Relaxing WAR dependencies

Microarchitectures often relax WAR dependencies by reading and storing source operands which are not in flight in structures such as reservation stations, allowing later instructions which might write to the source operand to proceed. There are two ways to use an emulator with such a scheme:

1. Explicitly control operand read timing. This requires that the emulator provide the *operandval* capability; such an emulator must also break up execution into separate operand fetch, evaluate, and writeback stages. Read the operands at appropriate times using `LSE_emu_fetch_operand`. Evaluate and writeback at appropriate times using `LSE_emu_doinstrstep`. Note that you do not need to manipulate the operand values; the emulator will have the correct values in the instruction information structure if the timing has been done correctly.
2. Explicitly control the operand values. This method also requires the *operandval* capability and execution broken into separate operand fetch, evaluate, and writeback steps, but you must manipulate values directly. Call the operand fetch step first to get non-in-flight source operand values into the instruction information structure (in-flight operands will also be set, but with stale values). Then update in-flight source operand values in the instruction information structure as they become available. Evaluate after operands have become available.

## Relaxing WAW dependencies

When the microarchitecture attempts to relax WAW dependencies, the base emulator interface and the simple form of execution no longer allow correct execution, as earlier instructions could write a value which a later instruction has already written. The interface provided by *operandval* and detailed execution must be used.

**TO DO**

Explain this better

## Relaxing RAW dependencies

RAW dependencies may be violated speculatively.

**TO DO**

Figure out how this works

## Software-scheduled pipelines

When the software is supposed to guarantee proper scheduling, as with many VLIW architectures, the microarchitectural model need not include interlocks for anything which the software is supposed to take care of, unless you wish to model the behavior of the machine with respect to mis-scheduled code. Any

interlocks which are present should be modeled in accordance with the style of interlocking as described above.

### Emulators with internal support for out-of-order execution

Some emulators, most notably those provided as part of Simics, attempt to support out-of-order execution by renaming registers as part of their decode operation. This behavior does not have a capability name, as LSE does provide any special support for this behavior.

**TO DO**

describe how to use them.

Emulators in this class may not work so simply with microarchitectures which perform unusual actions as part of renaming. Instruction reuse might be one such microarchitectural technique that would present difficulties. However, it seems likely that these could be overcome by overriding the operands of the instruction as described in the Section called *Manipulating operand values*.

### Instruction ordering and memory-carried dependencies

**TO DO**

Stuff to say here

## Notes

1. Patterson, David A. & Hennessy, John L. *Computer Organization & Design: The Hardware/Software Interface*, 1998, p. 5.

# Appendix A. LSS Reference

The Liberty Structural Specification (LSS) language is a language designed to describe hardware structure. It allows for concise specification of hardware systems by leveraging imperative programming constructs for instantiating, customizing, and connecting blocks. This appendix is a reference for LSS's syntax, semantics, and type system.

The programs have no input (other than the program itself), and the output, which is generated through side-effecting statements, is a netlist of structural components, their customization, and their interconnectivity. Since the programs have no inputs, programs written in this language are run-once, interpreted programs. This appendix will serve as a reference to help guide a programmer through the various syntactic and semantic elements of LSS.

**TO DO**

external types in LSS\_builtins

## Basic Syntax

In this section, the basic LSS syntax will be outlined. This will include the basic data types, data literals, variable declaration, control flow, and function definition and invocation. Side-effecting statements which create the programs output will be discussed later in the Section called *Machine Construction Constructs*.

### Basic Data Types

The LSS language is a strongly typed programming language. Thus, all values in the language have an associated data type. This section will describe the basic LSS data types and constants for these data types.

The following data types will be described in this section:

- `int`
- `float`
- `boolean`
- `char`
- `string`
- `literal`
- `type`
- `enumerations`

- arrays
- structures
- functions
- external types

### **int**

The `int` data type is used for integer data. Values of this type are 64 bit signed integers. Thus their values can range from  $-2^{63}-1$  to  $2^{63}$ . Integer value constants can be specified in binary, octal, decimal, and hexadecimal. Octal, decimal, and hexadecimal constants share the same syntax as C and Java. Decimal constants are specified using decimal digits (e.g. 341), octal constants are specified using the digits 0 through 7 and prefixing the constant with a 0 (e.g. 0525), and hexadecimal constants are specified using the digits 0 through 9 and a (or A) through f (or F) and prefixing with 0x (e.g. 0x155). Binary constants are specified using the digits 0 and 1 and prefixing the constant with 0b (e.g. 0b101010101). Negative numbers are specified by prefixing the constant with a - (e.g. -5, -0x5, -05, or -0b101).

### **float**

The `float` data type is used for floating point (real) numbers. These values are signed and their specific precision is undefined. Constant values for floating point numbers can be specified in standard decimal notation (e.g. 134.703) or using scientific notation (e.g. 6.022e23 or 6.022E23). The exponent in the scientific notation can be positive or negative. If no sign is specified it is assumed to be positive. For example, the following numbers are equivalent: 50, 5e1, and 5e+1. The following numbers are also equivalent: .001 and 1e-2.

### **boolean**

The `boolean` data type is used to represent boolean values. `booleans` can take on one of two values: `TRUE` or `FALSE`. For compatibility with other languages (such as Java), the literals `true` and `false` are also acceptable.

### **char**

The `char` data type is used for ASCII character data. Character literals are specified by placing the desired character between single quotes('). In addition, certain escape sequences are also legal: '\\\' for the backslash character, '\n\' for the newline character, '\t\' for the tab character, and '\r\' for the carriage return character. Only printable ASCII characters are permitted. This includes characters in between ' ' (ASCII 0x20) and '~' (ASCII 0x7E) as well as tab (ASCII 0x09), newline (ASCII 0x0A), and carriage return (ASCII 0x0D).

**string**

The `string` data type is used to hold string data. String literals are specified by enclosing sequences of characters between open `"` and close `"`, open `"""` and close `"""`, or open `<<<` and close `>>>`. For example, `"foo"`, `"""foo"""`, and `<<<foo>>>` all represent the string `foo`. Within a string literal, you can use the escape sequences `\r`, `\n`, and `\t` in addition to `\c` where `c` is any single character. Strings can span multiple lines with no special punctuation unless they are enclosed with open `"` and close `"`. Such strings cannot span multiple lines. Strings enclosed with open `<<<` and close `>>>` may contain fragments of the form `${expr}`. Such fragments will be replaced with the value of the LSS expression `expr`. The uses and exact semantics of such a replacement will be described in the Section called *Expression Substitution via \${}*.

**literal**

The `literal` data type is similar to `string` data type. It is used for storing strings that, eventually, will be output without any surrounding quotation marks. The details of why the `literal` type exists will be explained in the Section called *Parameters*. There are no constants that have type `literal`. However, `string` is a subtype of `literal` and thus `string` literals can be used whenever data of type `literal` is needed.

**type**

In the LSS language types are also values. This is useful, for example, when defining functions which want to create ports, parameters, or connections of a user specified type. In such cases, a function could be defined which accepts the type as an argument. The `type` data type is the type of all types including itself. The literal constants for this type include all the types discussed above (including this one) as well as any other syntactic construct which creates a type. For example, in addition to being a data type, `int` is a value whose type is `type`.

**enumerations**

Strictly speaking, in LSS there is no enumeration data type, but rather the `enum` keyword is a type constructor. The syntax:

```
enum { ident1, ident2, ... , identn }
```

will create a new anonymous data type whose constant values are given by `ident1, ident2, ... , identn`. Unlike enumerations in C, LSS enumerations are strongly typed. Thus, anything which expects data of a particular enumerated data type will *not* accept an integer as a substitute.

**arrays**

Arrays in LSS are similar to Java arrays. Unlike Java, LSS supports both bounded length and unbounded length arrays. Array data types let you define bounded or unbounded lists of a common data type. The syntax `type[expr]` creates a bounded array data type of `type` items with a length of `expr`. `expr` is

any LSS expression whose type is `int`. Alternatively, the syntax: `type[ ]` creates an unbounded array data type of `type` items.

Array literal constants are constructed using the syntax:

```
{ expr1, expr2, ..., exprn }
```

where `expr1`, `expr2`, ..., `exprn` must all have the same data type. This will create an array of size `n` of type given by the common type of `expr1`, `expr2`, ..., `exprn`.

In addition to the data values in the list, array values also have a length attribute which identifies the number of elements in the array. For example if `arr` is an array with type `int[10]`, then `arr.length` would have the value 10.

## structures

Structures in LSS are similar to C structures. Structure data types let you aggregate multiple pieces of data into a single data value. Just like enumerations, the `struct` keyword is a type constructor. The syntax:

```
struct {
  ident1 : type1;
  ident2 : type2;
  :
  identn : typen;
}
```

will create an anonymous aggregate data type with fields identified by `ident1`, `ident2`, ..., `identn`. The fields `ident1`, `ident2`, ..., `identn` have data types `type1`, `type2`, ..., `typen` respectively.

Structure literal constants are constructed using the syntax:

```
{ ident1 = expr1, ident2 = expr2, ..., identn = exprn }
```

where `expr1`, `expr2`, ..., `exprn` are LSS expressions used to initialize the fields `ident1`, `ident2`, ..., `identn` respectively

For example, the following structure could represent a point on a plane:

```
struct {
  x : float;
  y : float;
}
```

and the following structure literal constant would represent the origin of the plane:

```
{ x = 0.0, y = 0.0 }
```

## functions

Functions are used, as in other programming languages, however, in LSS they are first class values. The syntax for a function type is as follows:

```
fun (type1, type2, ..., typen) => typeret
```

This will define a function type which accepts  $n$  arguments with types  $type_1, type_2, \dots, type_n$ . The return type of the function is given by  $type_{ret}$ . More details on defining and using functions is in the Section called *Functions*.

## external Types

Some types have no LSS definition but are useful as types on ports and connections. These types, in particular, often arise in domain classes and implementations. The `external` constructor lets you create types which reference types in the underlying simulation language (currently stylized C). The syntax for the type constructor is:

```
external(expr)
```

`expr` must evaluate to a `string`-typed value and its value must be a legitimate type in the underlying simulation language. No values of this type can be defined in LSS

## Comments

LSS borrows the syntax from C++ for its comments. Multiline comments are delimited with `/*` and `*/`. Nesting comments of that type is not permitted. Single line comments are introduced with `//` and continue until the end of the line.

Just as in C++, comments are treated like whitespace by the LSS interpreter.

## Variable Declaration

This section will describe how to declare variables to store values during the execution of an LSS program. This section will make use the data types and value literals described in the Section called *Basic Data Types*. Variable declaration is the first LSS statement described in this reference. More information on statements can be found in the Section called *Statements*.

Like C, C++, or Java, the LSS language requires that all variables be declared before they are used. Within a given scope, two symbols cannot share a name. However, a variable defined in a new scope will mask all symbols from outer scopes that share its name. All LSS variables have lifetime equal to their lexical scope. Therefore, once a variable goes out of scope, its value is lost. Furthermore, it is illegal (a checked error) to read from an uninitialized variable.

The syntax for variable declaration is very simple and is similar to the style used in the PASCAL programming language. The following syntax:

```
var ident1[ = expr1 ],
```

```

    ident2[ = expr2],
    :
    identn[ = exprn] : [const] type;

```

will declare  $n$  variables which are named  $ident_1, ident_2, \dots, ident_n$ . Each variable will have data type given by  $type$ .  $type$  can be any LSS data type. The syntax from the Section called *Basic Data Types* should be used to create a variable with one of the basic data types. If the optional expressions are provided, they will be used to initialize the corresponding newly created variable. Recall that LSS is a strongly typed language, so the type of the initializing expression must match the type of the declared variable. Note that variables can be defined anywhere within a block. There is no restriction (as in C) that variables be declared at the top of a block.

If the optional type modifier `const` is given, the value of the variable cannot be changed after it is declared. Thus, it only makes sense to use the modifier if the initializing expression is also used.

Refer to Example A-1 to see several examples of variable declaration.

### Example A-1. Several Variable Declarations

```

var x : int; // declare an integer called x and
             // leave it uninitialized
var truth = false : boolean; // declare an boolean called truth and
                              // initialize it to false
var origin = {x = 0.0, y = 0.0} :
  struct {
    x : float;
    y : float;
  }; // declare a structure and initialize it
var i, j = 0, k = 1 : int; // declare several variables at once
var point = struct {
  x : float;
  y : float;
} : const type; // declare a variable of type type and initialize
                // to hold a structure type
var coord = {x = 10.5, y = -3.3} : point; // use the newly created type

```

## Expressions and Operators

This section will describe the basic LSS operators and expressions. Data values and variables connected with operators form expressions which will, in turn, be used as parts of LSS statements to build an LSS program. These expressions will create, combine, and transform data of the various types discussed in the Section called *Basic Data Types* and will prove extremely useful in machine construction.

Since LSS is a strongly typed language, all LSS expressions have a type. The types may not necessarily be known statically, but dynamically, all expressions will be type checked and any type errors will be reported and will cause the program's execution to abort.

The simplest LSS expression is a literal constant as described in the Section called *Basic Data Types*. The type of the expression is the same as the type of the value. Variable identifiers are also LSS expressions and once again, the type of the expression is equal to the type of the value held in the variable.

Any LSS expression can be enclosed in parentheses to form another LSS expression. Thus the syntax:

$(expr)$

is an LSS expression. The type of this expression is equal to the type of the expression  $expr$ . Expressions are evaluated according to operator precedence from left to right. Placing an expression in parentheses will cause the expression to be evaluated with high precedence.

## Unary Operator Expressions

There are three unary operators in LSS. These operators are  $-$ ,  $+$ , and  $!$ . Any expression with a numeric type (`int` and `float` types) may be negated by placing a  $-$  in front of it. Thus the syntax:

$-expr$

is an LSS expression whose value is the additive inverse of  $expr$ . To complement the unary negation operator, the  $+$  operator may similarly be applied to any numeric LSS expression and its value will be equal to the original expression's value.

For `boolean` typed expressions, the  $!$  operator will calculate the boolean complement. Therefore the expression,

$!expr$

would evaluate to the boolean complement of the expression  $expr$ .

## Binary Operators and Expressions

The LSS language supports a number of binary operators in addition the unary operators described in the previous section. All expressions formed with binary operators have the syntax:

$expr_1 \ op \ expr_2$

where  $op$  is the binary operator being used. Table A-1 summarizes the LSS operators, the valid expression types, the result type, and the operators semantics.

**Table A-1. Binary Operators**

Operator	$expr_1$ Type	$expr_2$ Type	Binary Operation Expression Type	Operator Semantics

Operator	<i>expr<sub>1</sub></i> Type	<i>expr<sub>2</sub></i> Type	Binary Operation Expression Type	Operator Semantics
+	float, int, string, literal, function types, functions	float, int, string, literal, function types, functions	float, int, string, literal, function types, functions	This operator will add its operands. For numeric types, this is common arithmetic addition. For <code>string</code> and <code>literal</code> operands, this operator performs string concatenation. For function types, this operator will produce an overloaded function type. The added function types must have a common return type and different numbers of arguments. For functions, this operator will produce an overloaded function. For this sum to be legal, the sum of the function types must be legal.
-	float, int	float, int	float, int	This operator will calculate the arithmetic difference of its operands.
*	float, int	float, int	float, int	This operator will calculate the arithmetic product of its operands.

Operator	<i>expr<sub>1</sub></i> Type	<i>expr<sub>2</sub></i> Type	Binary Operation Expression Type	Operator Semantics
/	float, int	float, int	float, int	This operator will calculate the arithmetic quotient of its operands. If the operands are ints, then the result will also be an int and it will have the fractional part of the quotient truncated.
%	int	int	int	This operator will calculate the remainder when of the arithmetic division of <i>expr<sub>1</sub></i> and <i>expr<sub>2</sub></i> . This is the modulo division operator
<<	int	int	int	This operator will left shift the bitwise representation of the value of <i>expr<sub>1</sub></i> by the number of bits specified by <i>expr<sub>2</sub></i> .
>>	int	int	int	This operator will perform an arithmetic right shift of the bitwise representation of the value of <i>expr<sub>1</sub></i> by the number of bits specified by <i>expr<sub>2</sub></i> .
== and !=	any	any	boolean	These operators will compare two values for equality and inequality respectively

Operator	$expr_1$ Type	$expr_2$ Type	Binary Operation Expression Type	Operator Semantics
<, <=, >, >=	int,float,string	int,float,string	boolean	These operators compare the two values provided. For strings the comparison is a lexicographic comparison.
&&	boolean	boolean	boolean	This operator calculates the logical AND of the two operands
	boolean	boolean	boolean	This operator calculates the logical OR of the two operands
	type	type	type	This operator concatenates two types to produce a polymorphic or-type

## The Ternary Operator

LSS supports the C-style ternary operator. This operator has the following syntax:

$$expr_{cond} ? expr_1 : expr_2$$

In this expression,  $expr_{cond}$  must evaluate to a `boolean`. If it evaluates to `true` then the whole expression evaluates to the value of  $expr_1$ , otherwise it evaluates to the value of  $expr_2$ .

## Assignment Operators

The `=` operator is used in LSS to assign a value to a variable or other lvalue. Similar to C, assignment in LSS is an expression. The expression  $expr_1 = expr_2$  will evaluate to the value of  $expr_2$  and simultaneously update the value of  $expr_1$  if it is an lvalue. It is a checked error for  $expr_1$  to not be an lvalue.

In addition to basic assignment, LSS also supports C style combination assignment operators: `+=`, `-=`, `*=`, `/=`, and `%=`. These operators are simply shorthand. The following two expressions are equivalent.

```
a = a + b
a += b
```

Similar equivalences hold for the other operators.

Finally, LSS also supports pre- and post- increment and decrement operators. The operators `++` and `--` can be placed before or after any `int` lvalue. The lvalue will be incremented or decremented respectively. If the operator comes before the lvalue, then the increment(decrement) expression will evaluate to the incremented(decremented) value. Otherwise, the expression will evaluate to the lvalue's previous value. This is the same behavior as in C.

### Example A-2. Pre- and Post- Increment

```
var x,y,z : int;
x = 3;
y = x++;

x = 3;
z = ++x;
```

Example A-2 should clarify any ambiguity. After this example runs, the variable `x` will have the value 4, the variable `y` will have the value 3, and the variable `z` will have the value 4.

## Indexing Expressions

Several LSS entities represent lists of items. Arrays, which were discussed in the Section called *Basic Data Types*, are one such example. Index expressions extract one item from such a list. The syntax for index expressions is as follows:

$$expr_{list}[expr_{index}]$$

. The expression  $expr_{index}$  must evaluate to an `int` and identifies which element from the list should be extracted. The expression  $expr_{list}$  must evaluate to some data type which is indexable. This expression identifies which list the item should be extracted from.

If  $expr_{list}$  is an lvalue, then this expression is also a legal lvalue and thus can be used to set items in a list in addition to extracting them.

## Subfield Expressions

Several LSS entities represent aggregates of items. Structures, which were discussed in the Section called *Basic Data Types* are one such example. Subfield expressions extract an item from an aggregate. The syntax for subfield expressions is as follows:

$$expr_{agg}.fieldname$$

The expression  $expr_{agg}$  must evaluate to some aggregate data type which has a field named *fieldname*.

If  $expr_{agg}$  is an lvalue, then this expression is also a legal lvalue and thus can be used to set items in an aggregate in addition to extracting them.

## Function Invocation Expression

The syntax for function invocation is identical to C and Java. An expression which evaluates to a function is followed by a parenthesized, comma-separated list of the actual arguments. Each actual argument is an LSS expression which evaluates to the type of the corresponding formal argument. The type of the function call expression is the type of the return value of the function. For example, to call a function named `func` with type `fun (int, bool)=>int`, the expression would be `func(3, FALSE)` and that expression would evaluate to a value of type `int`.

## Data Initialization Check Expression

It is illegal in LSS to reference a variable or parameter which has not yet been set. However, sometimes it is convenient (especially with parameters) to be able to check to see if a value has already been set. This expression allows one to check whether or not an expression contains any references to uninitialized variables or parameters.

The syntax for the expression is as follows:

```
initialized(expr)
```

The semantics of the expression are simple. *expr* is evaluated. If during the evaluation, any uninitialized entities are found, then this expression evaluates to `FALSE`. Otherwise it evaluates to `TRUE`. Note that if *expr* contains side-effects, they *may* occur. However, if an uninitialized value is found before reaching the side-effecting sub-expression, the side-effect *may not* occur also! Thus, it is discouraged from using any side-effecting expression within this expression.

### Example A-3. Use of the `initialized` Expression

```
var x : int;
if(initialized(x)) {
  print("Hello World\n");
} else {
  print("Goodbye World\n");
}
```

Example A-3 illustrates the use of this expression. This program will print:

```
Goodbye World
```

since the variable `x` is not initialized.

## Expression Substitution via `${}`

Any legal LSS expression can be embedded into a `string` using a special notation. When embedded inside of a `string`, the expression is evaluated and the resulting value is translated into a text which is appropriate for the underlying simulation language.

In order to embed an expression inside of a string, the `<<<>>>` quote characters must be used. Within a string quoted in that fashion, an LSS expression can be enclosed in `#{ }`. This expression will be embedded in the `string`. For example, the following code:

```
<<<#{3+7}>>>
```

would evaluate to the string "10". Table A-2 describes how values are translated when placed inside of `#{ }`.

**Table A-2. System-Defined Instance Parameters**

Type	Translation
<code>string</code>	The string's value is printed unquoted.
<code>type</code>	The type is converted to a type that is suitable for use in the underlying runtime language. Most types offer a straightforward conversion. One exception is arrays. An LSS array gets wrapped into a C structure with one field named <code>elements</code> . The field <code>elements</code> is an array with appropriate type and length.
<code>runtime_var ref</code>	The value is emitted as a variable accessible in the underlying simulation language
others	The conversion is straightforward and omitted for brevity

## Statements

An LSS program is sequence of statements. Statements exist to wrap expressions, group statements together, handle control flow, and include other files. An LSS program is evaluated by processing each statement in the sequence in order while following directions from certain statements which affect control flow. The next few paragraphs and sections will describe basic LSS statements and how they are executed.

The simplest kind of LSS statement is the expression statement. Following any LSS expression with a `;` forms an LSS statement. This statement causes the expression to be evaluated, including any side-effects, and then proceeds to the next statement in the sequence.

The next most simple kind of LSS statement is the *compound statement*. A compound statement has the following syntax:

```
{
  stmt1
  stmt1
  :
  stmtn
}
```

This statement serves to group together the statements inside of it. Execution of this statement simply amounts to execution of the statements inside of it in sequence order.

## Control Flow

This section outlines the features of LSS that allow users to specify control flow. The LSS language has a syntax very similar to C for control flow and the following few sections will describe what control flow statements exist and how they work.

### *The if Statement*

The `if`-construct in LSS is similar to the one in C with a few exceptions. The syntax for an `if` statement is as follows:

```
if (exprcond)
    cmpd_stmt
```

The first thing to notice is that the body of the `if` statement is a *compound statement*. This means that the body of the `if` statement *must* be enclosed in `{ }`. This is different from how `if` statements work in C.

To clarify this point, examine the following code listings. The following is illegal in LSS:

```
if (x == 3)
    x++;
else
    x--;
```

The correct LSS syntax is:

```
if (x == 3) {
    x++;
} else {
    x--;
}
```

While the above syntax prevents programming errors when adding code to an existing LSS specification, it makes chains of `if-else-if` blocks nest too deep. To alleviate this, LSS supports the `elsif` construct which can be used in place of the `else` clause. The following two programs are equivalent:

```
/* Program 1 */
if(x==3) {
    x++
} elsif
    if(x==2) {
        x--;
    }
}

/* Program 2 */
if(x==3) {
    x++
```

```

} elseif(x==2) {
    x--;
}

```

In addition to the required { } around the body of an if statements, since LSS is strongly-typed, the condition expression,  $expr_{cond}$ , provided in the if statement must evaluate to a boolean value.

### Loops

LSS currently only supports the for loop. The syntax for this loop is very similar to the syntax in C. The syntax for is:

```

for( $expr_{init}$ ;  $expr_{cond}$ ;  $expr_{inc}$ )
     $cmpd\_stmt$ 

```

Just like the if statement, notice that the body of a for loop is a compound statement. This means, unlike C, the body of the loop must be enclosed in { }. Also notice that the initialization clause of the loop,  $expr_{init}$ , is an expression, so it *cannot* include a variable declaration as can be done in Java or C++. Finally, it is mandatory for the type of  $expr_{cond}$  to be boolean. Example A-4 shows an example of a for loop.

#### Example A-4. A Simple for loop

```

var i,sum : int;
sum = 0;
for(i = 0; i < 10; i++) {
    sum += i;
}

```

A loop can be terminated early using the break statement. The syntax of the statement is simply the token break followed by a semicolon. Execution of this statement causes the innermost loop to terminate immediately.

### The return statement

The return statement allows the flow of execution to leave the body of a function early and also allows returning a value from a function. The syntax for the return statement is identical to C. A return statement is either the keyword return followed by a semicolon or the keyword return followed by an expression followed by a semicolon. In the first form, no value is returned from the function. In the second form, the given expression will be evaluated and its value will be the function's return value. Note, that the type of the expression must match the return type of the function. Further note, that it is illegal to use the first form of the return statement in any function whose return type is not void. Finally, note that the return statement may *only* appear in the body of a function. Any other use is illegal.

## Including Other Source Files

In order to allow a machine description to span more than one file, LSS offers two mechanisms to pull in other source files. The first, the `include` statement amounts to simple textual replacement of the named file inline where the `include` statement appeared. The following example illustrates the syntax for the `include` statement.

```
include "other.lss";
```

Note that only `string` literals can be used in `include` statements, not expressions which evaluate to `strings`. If the specified file name is absolute, LSS will include it directly, otherwise, LSS will search the module search path in order to find the file.

**Note:** The use of `include` statements is generally discouraged due to the potential namespace collisions that can occur. This is especially true for any reusable code that is being put into an LSS file. Use of the package system is recommended.

In addition to the `include` mechanism, LSS supports a package system for grouping together code in a unique namespace. The system is described in more detail in the Section called *Packages*. That section will describe the `import`, `using`, and `subpackage` statements.

## Declarations

This section will cover a few statements used to declare LSS types, variables, and functions.

### *Variables*

Variable declaration is discussed earlier in the Section called *Variable Declaration*. Look there for details.

### *Types*

In order to ease the use of complex data types, new data types can be assigned names through the use of the `typedef` declaration. The syntax for the statement is as follows:

```
typedef ident : type;
```

This syntax will associate the identifier *ident* with the type *type*. In reality, this syntax is shorthand for:

```
var ident = type : const type;
```

For example, the following two pieces of code are equivalent.

```
/* Program 1 */
typedef point : struct { x : int; y : int; };

/* Program 1 */
var point = struct { x : int; y : int; } : const type;
```

Since the `typedef` statement is shorthand for a variable declaration, all the same scoping rules apply.

### Functions

Functions in LSS are similar to functions in C and C++ and methods in Java. Each function is piece of code which accepts arguments and produces a return value. The type signature of the function determines the types of the arguments and return values. Once defined, a function can be invoked and the body of the function will be executed using the arguments passed to the function to produce a return value and cause any side-effects.

As was mentioned in the Section called *Basic Data Types*, functions are first class values in LSS. The data type constructor for functions was discussed in that section, however, no syntax for function literals was given. In LSS it is *impossible* to create an anonymous function literal (a  $\lambda$ -expression). Instead, named functions can be declared and then they can be assigned to other variables of the appropriate function type.

The syntax for declaring a function is as follows:

```
fun ident(ident1 : type1, ident2 : type2, ..., identn : typen) => typeret =
    compd_stmt;
```

$ident_1, ident_2, \dots, ident_n$  are the formal arguments of the function and have types  $type_1, type_2, \dots, type_n$  respectively. The return type of the function is  $type_{ret}$ . If the return type of the function is *not* `void` then the body of the function *must* contain a `return` statement which returns a value of the appropriate type.

If, within the same scope, two functions with the same name, same return type, and different numbers of arguments are defined, the function will become an *overloaded* function. The correct function will be dispatched during invocation based on the number of parameters.

## Conditional Assignment

As a parallel to the `initialized` expression, there is a statement which acts as shorthand for a common idiom when dealing with hierarchical modules, parameters, and default values. It is often desirable to not set a parameter on an sub-instance, if a parameter on your own instance is unset. This behavior could be achieved with an `if` statement and the `initialized` expression, however, this statement is shorthand for that composition. The statement:

```
exprlvalue ?= expr
```

will cause the lvalue to which  $expr_{lvalue}$  evaluates to be assigned the value to which  $expr$  evaluates only if `initialized(expr)` would evaluate to `TRUE`. Note that in processing this statement,  $expr$  is only evaluated once.

## Built-In Functions

The following list summarizes some built-in LSS functions.

```
print(str : string) => void
```

This function prints the given string to standard out

```
punt(str : string) => void
```

This function prints the given string prefixed with `Punt:` to standard error. It also aborts the LSS program, thus terminating simulator construction

```
warn(str : string) => void
```

This function prints the given string prefixed with `Warning:` to standard error.

```
to_string(val : any-type) => string
```

This function converts any value to its `string` representation

```
to_literal(val : any-type) => literal
```

This function converts any value to its `literal` representation

```
LSS_ipow(base:int, exponent:int) => int
```

This function computes  $\text{base}^{\text{exponent}}$  and returns it.

```
LSS_log2down(val:int) => int
```

This function computes  $\lfloor \log_2(\text{val}) \rfloor$  and returns it.

```
LSS_log2up(val:int) => int
```

This function computes  $\lceil \log_2(\text{val}) \rceil$  and returns it.

## Machine Construction Constructs

This section discusses all the primitive operations supported by LSS to create objects for use in simulator construction. The declarations, expressions, and statements seen in the Section called *Basic Syntax* were used to control the flow of the LSS program or to store variables during its execution. Conversely, the declarations, expressions and statements that will be seen in this section will cause *side-effects* that create or customize objects that are part of the programs netlist output. This distinction is important and should be remembered when reading this section.

### Module Instances

Module instances are the most fundamental components of an LSS program. Creating a module instance in LSS creates a component in the generated runtime simulator. In the generated simulator, this component will be responsible for reading input values from its input ports, maintaining internal state, and producing output values on its output ports. Each module instance is created from a parameterizable template called a *module*. More details on modules will be covered in the Section called *Modules*, however, this section will cover their instantiation and parameterization.

## Creating Module Instances

New instances are created with the `new instance` expression. The syntax for this expression is as follows:

```
new instance(instance-name, module-name)
```

*instance-name* is an expression that must evaluate to be a `string` which gives a name to this newly created instance. *module-name* is an identifier for a module declared within the current namespace or a package-scoped identifier for a module declared within a package. The expression returns a value of type `instance ref` which is a reference to the newly created instance. Values of type `instance ref` are aggregate data structures and subfields of the structure can be accessed using subfield expressions.

Since it is often desirable to create arrays of instances from the same module, there is another `new instance` expression which will do just that. The syntax:

```
new instance[exprsize](instance-name-base, module-name)
```

will return an array of `instance refs`. The size of the array is determined by the *expr*<sub>size</sub> expression. This expression must evaluate to a value of type `int`. The newly created instances can be accessed from the returned array of references and will be named *instance-name-base*0, *instance-name-base*1, ..., *instance-name-base*N where N+1 is the value to which the expression *expr*<sub>size</sub> evaluates.

The most common usage pattern for the `new instance` expression is:

```
var instance-name = new instance("instance-name", module-name) : const instance ref;
```

and thus LSS provides a shorthand syntax for this operation with the `instance` declaration statement.

The following `instance` declaration statement is equivalent to the above module instantiation:

```
instance instance-name:module-name;
```

## Parameterizing Module Instance

Parameters are used to customize a module instance's functionality, timing and interface to obtain a specialized component for the runtime system. Each module from which an instance is instantiated may define parameters which will affect the behavior of an instance. These parameters are free to change simulator runtime properties (e.g. size of a cache, etc.) as well as instance interface properties (e.g. names of ports, presence of other parameters, etc.).

### Using Parameters

To set a parameter on an instance, the subfield expression is used. For example, if `inst` is an `instance ref` variable referring to an instance of module **mod** and module **mod** has a parameter named *parm*, then

this parameter can be referenced with `inst.parm`. To set the parameter's value one would use the following syntax:

```
inst.parm = expr;
```

where `expr` evaluates to a value whose type is compatible with the type of the parameter `parm`.

Some parameters on a module have default values, while others do not. Those parameters without default values *must* be filled in on any instance instantiated from that module. The other parameters are optional.

### Code-Valued Parameters

In addition to the types discussed in the Section called *Basic Data Types*, parameters (and variables) may contain source code which implements particular parts of a components functionality. These code-typed values will also prove useful when defining data collectors. Several data types are used to hold code-valued data including the `string` type which has already been introduced.

An internal (not user accessible) type exists to represent control points on ports. Parameters of the `controlpoint` type can be assigned `string` values and the value will be coerced into the `controlpoint` type.

A user-visible type constructor, `userpoint`, is used to define algorithmic parameters on modules. The syntax for using the type constructor is as follows:

```
userpoint(exprargs => exprret)
```

This syntax will create a new `userpoint` type. The expressions `exprargs` and `exprret` must evaluate to a `string`-typed values. `exprargs` declares a formal argument list to the code that will fill in the parameter which uses this type. `exprret` declares the type of the data returned by the code that will fill in the parameter. Note that the types and syntax of these strings is that of the backing simulation language (currently stylized C). Just like the `controlpoint` type, a parameter with a `userpoint` type can be assigned a `string`-typed value. Example A-5 illustrates the declaration of a `userpoint` parameter and assigning it a value.

### Example A-5. Userpoint Declaration and Use

```
parameter comparison : userpoint(<<<int x, int y>>>) => <<<int>>>;

comparison = <<<
  if(x < y) return -1;
  else if(x > y) return 1;
  else return 0;
>>>;
```

**Note:** For all code-typed parameters, you should refer to the *Liberty Simulation Environment API Reference Manual* to see what API calls are available.

### System Defined Instance Parameters

Instances have several parameters that are defined by the system. These are listed in Table A-3 along with their type and purpose.

**Table A-3. System-Defined Instance Parameters**

Name	Type	Purpose
<code>funcheader</code>	string (code)	Include header files and define common functions for use in control functions
<code>init</code>	<code>userpoint(&lt;&lt;&lt;void&gt;&gt;&gt; =&gt; &lt;&lt;&lt;void&gt;&gt;&gt;)</code>	Code run at simulator startup
<code>start_of_timestep</code>	<code>userpoint(&lt;&lt;&lt;LSE_time_num skipped&gt;&gt;&gt; =&gt; &lt;&lt;&lt;void&gt;&gt;&gt;)</code>	Code run at the start of every simulation timestep. The argument <code>skipped</code> indicates how many timesteps have been skipped since the last simulated timestep.
<code>end_of_timestep</code>	<code>userpoint(&lt;&lt;&lt;void&gt;&gt;&gt; =&gt; &lt;&lt;&lt;void&gt;&gt;&gt;)</code>	Code run at the end of every simulation timestep.
<code>finish</code>	<code>userpoint(&lt;&lt;&lt;void&gt;&gt;&gt; =&gt; &lt;&lt;&lt;void&gt;&gt;&gt;)</code>	Code run at simulator finish
<code>port-name.control</code>	controlpoint	Code run whenever a signal on the port named <code>port-name</code> changes. This code is used to filter the signal values entering or leaving a module instance.
<code>port-name.width</code>	int	Setting the width field will fix the port at the given width. It is an error then if there is a connection to the port with index larger than or equal to the width value. If fewer connections are made, the unconnected port instances will <i>still</i> exist.

### Runtime Parameters

While many parameters on modules will be fixed in a specification at design time, it is convenient to allow some parameters to be set at runtime. If a module exports a parameter as `runtimeable`, then the parameter may be exported such that it can be set at runtime. To do this, one must create a `runtime_parm` value using the following syntax:

```
new runtime_parm(exprtype, exprdefault-value, exproption-name, exproption-desc)
```

where *expr*<sub>type</sub> evaluates to the type of the parameter, *expr*<sub>default-value</sub> evaluates to the default value

of the parameter (used when no runtime value is specified), `exproption-name` evaluates to a `string` which is the option name exported to the simulator command-line processor, and `exproption-desc` evaluates to a `string` which is exported to the command-line processor as help for this command-line option.

## Module Instance Connections

While module instances are fundamental for creating a simulator specification, they have little value without the ability to connect module instances together. Module instance connections allow a user to specify the interconnectivity of the machine being modeled. Connections are discussed in this section, however, ports are only covered in as much detail as is needed to discuss connections. A more thorough discussion of ports and operations on ports is in the Section called *Modules*.

### Syntax and Semantics

The data type used to represent port objects is the `port_ref` data type. If `p1` and `p2` are `port_refs`, a connection is made between the two ports using the `->` operator as follows:

```
p1 -> p2;
```

If `inst` is an `instance_ref` referencing an instance with a port `p`, `i.p` is also a `port_ref` and thus we can write:

```
p1 -> inst.p;
```

Each port in LSE is actually an indexed series of ports called a multiport. Connections can be made explicitly between multiport instances by using the indexing operator to specify the port index. This is shown below:

```
p1[0] -> i.p[2];
```

A connection is always made between a pair of port instances. In fact, each port instance can only appear in a *single* connection. This means that all connections are point-to-point, and there is no built-in notion of fanout.

In certain situations, the specific port instance number is not relevant (e.g. the specific output multiport instance `c` on an instance of the `tee` with one input connection). In such cases, rather than requiring specification of port instance numbers, LSS will automatically assign port indexes when the connection operator is used. The syntax for this is actually shown in the earlier examples, the port index is just omitted. In a given connection statement, one or both port indexes may be omitted and the omitted index will be automatically assigned by the LSS interpreter to the next available index. Connections will be assigned to port indexes in the order in which the connections are seen. To avoid confusion, the LSS interpreter will flag an error if a particular port is used in connection statements with both explicit

indexing and implicit automatically generated indexes. Example A-6 shows an illegal mix of explicit and implicit port indexing.

#### Example A-6. Incorrect Port Indexing

```
p1[0] ❶ -> i.p
```

```
p1 ❷ -> p2
```

- ❶ Port **p1** is used with explicit port index 0
- ❷ Port **p1** is used *without* an explicit port index

The code shown in the example would be rejected by the interpreter since the **p1** is both explicitly indexed and implicitly indexed. Example A-7 shows the corrected code.

#### Example A-7. Corrected Port Indexing

```
p1[0] -> i.p
```

```
p1[1] -> p2
```

## Port Types and Connections

Each port on a module instance is typed and a connection can only be made between two ports with compatible types. For non-polymorphic types, the compatibility relation is equality. That is to say, only two ports with equal types can be connected. However, in order to allow modules to be more flexible, the types on a module's ports can be polymorphic. To handle this polymorphism, the LSS interpreter includes a type inference engine which will resolve the polymorphism on an instantiated system. To aide this inference process, connections can and sometimes *must* include typing constraints. This section will discuss the types of polymorphism and how connections can constrain the set of possible instantiations of the polymorphism.

### *Polymorphic Types*

The LSS system has two fundamental polymorphic type constructs. From these constructs, complex polymorphic types can be built. A polymorphic type can be used in any type constructor where a type is expected.

### *Type Variables*

The first such construct is the type variable. The syntax for a type variable is:

```
'ident
```

This syntax is a use of the type variable named by *ident* and its first use also serves as its definition. A type variable stands for any LSS type and the value of the type variable is resolved by type inference.

To support array types with polymorphic length (as opposed to unbounded length), LSS also supports another syntax for array length type variables. The syntax:

```
#ident
```

is a type variable that can be used as the size of an array when using the array type constructor. For example:

```
int[#len]
```

defines an array of integers with polymorphic length. The actual length of the array will be resolved during type inference.

The type a type variable may take is initially unconstrained. As will be described in the next section, port connections constrain the legal values of type variables. A shorthand notation exists for creating anonymous type variables (i.e. type variables that will not be explicitly referenced elsewhere). The symbol \*, each time it is used, will create a new anonymous type variable. The symbol was selected because, in essence, the type is a wild card.

A specification where there are multiple values for a type variable that satisfy all constraints is an *under-constrained* system. A system for which *no* type exists which satisfies all the constraints is an *over-constrained* system.

### The Or-Type

In the Section called *Binary Operators and Expressions* the | operator was introduced to create or-types. During type inference, any entity which has an or-type will be resolved to one of the types listed in the disjunction.

### Constraining Port Types with Connections

Each time a connection is made between two ports, the two ports are constrained to have the same type. The user can further constrain what this type may be by placing a constraint expression after the connection operator. The syntax for this is shown below:

```
p1 ->[exprconstraint] p2
```

Legal constraint expressions include any expression which evaluates to a type *type*. The following are several examples of connections with additional constraints:

```
1 p1 ->:int p2
2 p1 ->:[int | boolean] p2
3 p1 ->:'a p2
4 p3 ->:'a p4
```

The first line constrains **p1** and **p2** to have type *int*. The second line constrains port **p1** and **p2** to have either *int* or *boolean* as their types. The last two lines constrain **p1**, **p2**, **p3**, and **p4** to all have the same type, specifically the value of type variable 'a.

## Utility Functions

Since it is common to connect port instances in buses of connections, a utility function has been defined to achieve this. The function, `LSS_connect_bus` will make  $N$  connections on port indexes  $0 \dots N-1$  between two ports. The function is overloaded. In its first form it takes three arguments: a `port ref` for the source of the connection, a `port ref` for the destination of the connection, and finally an `int` for the width of the bus. In its second form, it has an additional fourth argument which is a type constraint to be applied to the connections.

## Augmenting Instance State

LSE offers two mechanisms by which to augment the state kept by a module. The first mechanism adds fields to common runtime structures. The second mechanism allows users to define arbitrary variables for use in control and user functions.

### **structaddS**

LSS defines a builtin function to augment some simulation time data structures with additional per-instance fields. Presently, `LSE_dynid_t` and `LSE_resolution_t` can be augmented. In order to augment the data structures, one may call the `structadd` function. The function's signature is:

```
structadd(inst : instance ref, data_struct : string,
          field_type : string, field_name : string) => void
```

The first argument to the function indicates for which instance you wish to augment the data structure. The second argument is a `string` which identifies which data structure you wish to augment. The legitimate values for the second parameter are `"LSE_dynid_t"` or `"LSE_resolution_t"`. The third argument is a `string` containing the type of the field you wish to add. This type should be a type in the underlying simulation language. Finally, the last argument is a `string` which names the field. For a given module instance, the field name's must be unique. The following is an example of a `structadd` call:

```
structadd(inst, "LSE_dynid_t", "int", "counter");
```

## Runtime Variables

The other mechanism for augmenting instance runtime state is to create a runtime variable. To create a runtime variable, use the following syntax:

```
new runtime_var(expr_name, expr_type)
```

This expression will return a value of type `runtime_var ref`. You can reference this variable inside of strings using `${}`. This reference will be the runtime variable name. So you can treat this reference just

as if it were a variable in the underlying simulation language. For example, the following piece of code would update a round-robin counter at the end of each cycle:

```
var round_robin_counter : runtime_var ref;
round_robin_counter = new runtime_var("rr_counter", int);

inst.end_of_timestep = <<<
  ${round_robin_counter} = (${round_robin_counter} + 1) % 5;
>>>;
```

Note that the name of runtime variables need not be unique, but unique names are encouraged to promote faster incremental build times.

## Modules

Modules are the building blocks for simulator specifications. Modules are instantiated to form the runtime components of a simulation system. As has been described earlier, instances can be customized through parameters which must be defined by modules. Further, instances can be interconnected via ports which must also be defined by the module from which the instance was instantiated. In this section, the syntax for defining modules will be discussed. Since LSS supports two kinds of modules, *leaf modules* and *hierarchical modules*, this section will discuss the syntax common for both types of modules and then the syntax that is specific to each type of module.

### Module Declaration Syntax

To declare a module, leaf or hierarchical, one uses the `module` keyword followed by the name of the module, followed by a compound statement that will be run when an instance of this module is defined, and finally a trailing semicolon. This syntax is shown below:

```
module module_name {
  ...
};
```

Within a module body any statements are permissible, with certain exceptions to be noted below, and they have the same effect as if invoked at the top-level of the description. There are however, several types of statements that are for use within module declarations only. These are port declarations and parameter declarations, and, for leaf modules, query and method declarations, event declarations, and type exports.

### Ports

Ports define the interface of a module. To declare a port in LSS one uses the `inport` and `outport` keyword for input ports and output ports respectively. The following module declaration declares a

module with an input port `in` and output port `out`. Both ports have data type `int`.

```
module foo {
  inport in:int;
  outport out:int;
};
```

In general the syntax for declaring a port is

```
inport portname:exprtype;
outport portname:exprtype;
```

The syntax will add a port named `portname` to the instance being processed as well as create a symbol of type `port ref` in the current scope named `portname`. Recall that the type on a port can be a polymorphic type.

In addition to being defined statically, ports may also be defined dynamically using the new `inport` or the new `outport` expressions. These expressions have the following syntax:

```
new inport(exprname, exprtype)
new outport(exprname, exprtype)
```

The expressions evaluate to values of type `port ref` and these references may be stored in variables for further connection and manipulation. The created port will have name and type given by the `string` value to which `exprname` evaluates and the `type` value to which `exprtype` evaluates respectively.

There are several attributes (accessed via the subfield expression) on ports that may be read or written to control the specific behavior of the system in relation to the module. Most of these fields are only relevant for leaf modules and are discussed there. However, the fields `width`, `connected`, and `control` are available on both leaf and hierarchical modules. The `width` and `connected` fields are both read-only fields for any port on the current module being evaluated. The `width` field is an `int` whose value is one more than the largest index connected on the port. The `connected` field is a `boolean` that is `TRUE` if there are any connections to the port. The `control` field defines some code that is run whenever a signal on the port changes. See Table A-3 for more details on the `width` and `control` attributes. Note that any assignment to the `control` attribute is a default value assignment that can be overridden by the user.

## Parameters

Parameters are used in the module declaration and definition to create a highly flexible module. Functionality, timing and interface can be made flexible by using parameters. Parameters behave very similarly to variables, and in fact their syntax is quite similar too, however it is important to understand the *significant* differences.

The syntax for declaring a parameter is as follows:

```
[parameter-modifier] parameter parmname:exprtype;
```

Just like instances and ports, there is a dynamic syntax as well. This syntax is:

```
new [parameter-modifier] parameter(exprname, exprtype)
```

The first syntax creates a parameter named *parmname* and a local variable of type `parameter ref` named *parmname*. The second syntax is an expression that evaluates to type `parameter ref` and creates a parameter whose name and type are the *string* value to which *expr<sub>name</sub>* evaluates and the *type* value to which *expr<sub>type</sub>* evaluates respectively. Table A-4 describes the legal values for *parameter-modifier* and what they mean.

The first difference to note, between parameters and variables, is what assigning to them means and how they influence the runtime behavior of the specification. Assignment to parameters within the body of a module is a *default* value assignment. Users of the module can override this value by assigning to the parameter when instantiating the module. Therefore, the assignment is relevant when the user *does not* assign to the parameter. Because of this property, it is desirable to ensure that a consistent view of parameters is maintained. Therefore, although multiple default assignments to the same parameter are legal, no assignment may be made to the parameter once the value has been read (i.e. used as an rvalue). Finally for leaf modules, the value of the parameter will be available in the code which implements the behavior of the module (unless an appropriate *parameter-modifier* is used).

**Table A-4. Parameter Modifiers**

Modifier	Meaning
local	User's <i>cannot</i> override default values
internal	Parameter not exported to the behavioral code
runtimeable	This parameter can be set at runtime

## Leaf Modules

Leaf modules are modules whose behavior is *not* defined in LSS, but rather in a behavior specification language (currently stylized C). Thus, their description consists of two pieces:

1. The module declaration consisting of the port declarations, parameter declarations, structadds, queries, methods, and events. This is specified in LSS.
2. The module definition which is a behavioral description of the module's timing and functionality. This is specified in a separate file (.clm file) in a stylized C language.

## Module Attributes

Leaf modules possess certain basic attributes that can be set within the module. Table A-5 summarizes the names, types, and meanings of these attributes.

**Table A-5. Leaf Module Attributes**

Name	Required	Type	Purpose
------	----------	------	---------

Name	Required	Type	Purpose
tar_file	yes	string	This attribute specifies the .tar file which contains all the .clm code for the module.
phase_start	yes	boolean	Indicates whether or not this module has a phase_start function
phase	yes	boolean	Indicates whether or not this module has a phase function
phase_end	yes	boolean	Indicates whether or not this module has a phase_end function
reactive	yes	boolean	Indicates whether or not this module has internal state or if it reacts only to its inputs
port_dataflow	no	string	This string is a Python list of tuples. Each tuple has the form: ( <i>source-signal</i> , <i>dest-signal</i> , <i>condition</i> ). Each one of the replaceable terms is a Python string. The first two have the format: <i>port-name</i> . <i>signal-name</i> where <i>signal-name</i> is data, en, or ack. The <i>port-name</i> can be an actual port name or the wildcard character, *. The <i>condition</i> is a Python boolean expression for when this data dependence exists. It may use the variables <i>isporti</i> and <i>osporti</i> which are the input and output port instance numbers respectively. By default, the system assumes dependence amongst all ports and signals, so the tuple ('*', '*', '0') is typically the first element in the list.

## Port Attributes

Ports have various attributes which affect how the module's behavioral description handles information arriving on a specified port. Table A-6 describes the attributes, their type, and meaning.

**Table A-6. Port Attributes on Leaf Modules**

Name	Required	Type	Purpose
------	----------	------	---------

Name	Required	Type	Purpose
independent	yes	boolean	If this attribute is true, then changes to the status of this port will not cause this module to be activated. The data however will be buffered until after phase_end so that it may be used to update state at the end of the cycle. If this attribute is false, port status changes will cause module activation. However, data on this port is not buffered. Therefore it is not available for use during phase_end. The module must manually buffer any data it wishes to use during phase_end.
handler	yes	boolean	This attribute specifies whether or not a handler processes port status changes for this port. If the parameter is false, the modules phase function is activated on port status change. However, if the module does not have a phase function, the module will not be activated. Also, if the port has been marked independent, this attribute has no purpose and is ignored.

## Methods and Queries

A method can define methods and queries for other code to invoke. A method is a function which does not affect scheduling. A query on the other hand is a method which can return an undetermined value but cause reinvocation later in the schedule.

The syntax for declaring a query is as follows:

```
query name : (stringargs => stringret);
```

*string<sub>args</sub>* is a string literal which defines the argument list. *string<sub>ret</sub>* is a string literal that defines the return type.

The syntax for declaring a method is as follows:

```
[locked] method name : (stringargs => stringret);
```

*string<sub>args</sub>* is a string literal which defines the argument list. *string<sub>ret</sub>* is a string literal that defines the return type. If the optional `locked` token is used then the method may only be invoked from the instance on which it is defined.

## Events

A module may emit events which can be processed by data collectors to allow for simulator instrumentation. Each event comes from a particular instance of the module and can carry with it

information which describes what occurred. The syntax for defining events is as follows:

```
event name {
  field1 : type1;
  field2 : type2;
  ⋮
  fieldn : typen;
};
```

$field_1, \dots, field_n$  are identifiers labeling the pieces of data that the event will emit.  $type_1; \dots, type_n;$  are string literals which identify the type of the data in the underlying simulation language.  $name$  is the name of the event.

Events may be declared anywhere, but it is common to define them in packages or inside of a module body. Declaring an event does not state that a module will generate that event. The `emits` statement is used to indicate that a module will emit an event. The syntax of the `emits` statement is as follows:

```
emits event-name;
emits event-declaration;
```

The two alternative syntaxes give two ways to declare that a module emits an event. The first references an already declared event. The second simultaneously declares an event and asserts that this module emits that event.

## Type Exports

If the code that implements a leaf module wishes to use an LSS type, the module declaration can export the type to the behavioral code. The syntax for exporting the type is:

```
export exprtype as ident;
```

This statement will cause the `type` to which the expression  $expr_{type}$  evaluates to be accessible as  $ident$  in the behavioral code.

## Hierarchical Modules

Unlike leaf modules, hierarchical modules specify their behavior by instantiating other modules and interconnecting them. Thus all the syntax discussed in the Section called *Machine Construction Constructs* can be used inside of a hierarchical module to define its behavior.

One important thing to note is that connections made to ports of this module have inverted direction sense. That is to say, an output port of this module can be connected to an output port of one of the child instances. The child instance is feeding this module's output. Similarly, an input port on this module can be connected to an input port of a child instance. The input port of the module is feeding the child instance. These direction senses are inverted from the more familiar connections between output ports and input ports.

**Note:** In hierarchical modules, parameters are often propagated down to child instances. It is desirable to have no default value for a such parameters and simply not override the default value of a child parameter if the user did not set the value of the hierarchical parameter. To accomplish this a conditional assignment operator is defined.

```
parameter propagate_me : int;
instance child : foo;

foo.parm = propagate_me;</pre

```

Notice how no default value was given to the parameter `propagate_me` and how the `=</code was used in the assignment. This operator assigns only if propagate_me has been assigned a value.`

## Data Collectors

In order to instrument a simulator for data collection, a specification must capture events using data collectors. The syntax for defining data collectors is as follows:

```
collector event-name on exprinst {
  [decl = decl-string;]
  [init = init-string;]
  [record = record-string;]
  [report = report-string;]
};
```

*event-name* is the name of the event that you wish to collect data from. *expr<sub>inst</sub>* is an expression which should evaluate to a *string*. The value of the *string* should be the name of an instance relative to the current instance (or fully qualified if at the top-level). All of the values inside the `{}` are *string* literals of code that will run during the simulation. The meanings of the various sections is defined in Table A-7.

**Table A-7. Collector Sections**

Field	Meaning
decl	Include declarations that are useful for your collector in this section. This should also include statements to include any necessary header files.
init	This section is run once at simulator initialization time. Initialize variables that need to be initialized here.
record	This section gets run each time the event is triggered. Include any code to aggregate statistics or print debugging information here.
report	This section gets called once at the end of simulation. Include code in this section to report any statistics aggregated during simulation.

In addition to events defined explicitly by modules. Several implicit system events exist. First, there are two toplevel events. The declaration for these two events is as follows:

```
event start_of_timestep {
    skipper : "LSE_time_numticks_t";
};

event end_of_timestep {
};
```

These events are generated at the beginning and end of each timestep.

Each port also has an implicit events defined on it. The signature for these events are:

```
event portname.resolved {
    porti : "int";
    status : "LSE_signal_t";
    prevstatus : "LSE_signal_t";
    id : "LSE_dynid_t";
    datap : "LSE_port_type(portname) *";
};

event portname.localresolved {
    porti : "int";
    status : "LSE_signal_t";
    prevstatus : "LSE_signal_t";
    id : "LSE_dynid_t";
    datap : "LSE_port_type(portname) *";
};
```

These events get fired whenever signal values change either outside the control function or inside the control function. The fields of the event are mostly self-explanatory. `porti` is the port instance which had the signal change. `status` is the status of the port signals. `prevstatus` is the status of the port signals the last time the event was fired. `id` is the dynamic identifier of the message sent on the port and `datap` is the data that was sent.

## Packages

LSS provides a system by which items may be placed in a separate namespace. These separate namespaces, called packages, provide a mechanism to bundle related modules, functions, variables, and types. Users can import a package loading its contents for use, and can also import all the items in the package into the current namespace.

## Using packages

### Usage overview

To load a package, the `import` statement is used. The syntax of the `import` statement is shown below.

```
import package_name
```

To use elements inside this package, the `::` operator is used to qualify an identifier with a namespace. Thus, to access the **rename\_table** module inside a package call `dlxlib`, one would do the following.

```
import dlxlib;
instance x:dlxlib::rename_table;
```

To make all the symbols defined in package accessible without qualification, the `using` statement is used.

```
using package_name;
```

The `using` statement will additionally import the named package if it has not already been imported. The same use of **rename\_table** above, but with `using` instead of `import` is shown below.

```
using dlxlib;
instance x:rename_table;
```

Note that the `using` statement does not actually place the names into the current name space, but instead adds the specified package (or subpackage) to a package search list. Thus, symbols from packages that were included with the `using` statement earlier are chosen in preference to those that were included later. The package search list itself is scoped like any other variable.

## Packages, Subpackages and Naming

Package names consist of a list of identifiers separated by dots. For example, `corelib`, `LSE_emu`, and `corelib.tee` are all valid package names.

LSS supports two kinds of packages: package and subpackages. The difference between the two is subtle, but important. Packages can be directly imported, while subpackages can only be imported as a side-effect of importing another package. `corelib` for example is a package, while `corelib.tee` is a subpackage.

Because of this difference, the `using` statement cannot be used with a subpackage unless it has already been imported. Conversely, the `using` statement will automatically import a package that has not already been imported.

Within a package, symbols can be accessed using a relative name (i.e. a symbol name that is not qualified with the `::` operator) even if no `using` statement has been used. In fact, an error will be generated if any

attempt is made within a package to import the package that is being defined. Such circular references are illegal.

Symbols from other packages or subpackages can be accessed using qualification. Either the full package name can be used, or the package name itself can be relative. By default, the package name is assumed to be fully qualified. If the package does not exist, the current package name is prepended to the given package name and this fully qualified name is searched for. If it doesn't exist, then an error is emitted.

Relative symbol references are made by omitting the `::` and everything before it. Relative names are relative to the current package and if the symbol is not found, it is then relative to the various packages on the package search list. The first package where a match is found is used. If no match is found, an error is emitted.

## Building Packages

Packages are defined within a file that begins with the package statement. The syntax is shown below.

```
package package_name;
```

The toplevel file associated with a package must conform to a particular naming convention. To understand this convention, the method used to search packages must be understood. This process is described below. Assume the command in question is:

```
import foo.bar.baz;
```

The module path is searched looking for the file which defines the package `foo.bar.baz`. We search each directory in the `module_path` looking for `foo/bar/baz.lss`, then `foo/bar.baz.lss`, and finally `foo.bar.baz.lss`. The different file names are iterated over on a directory by directory basis. Therefore if `foo.bar.baz.lss` is located in the first directory in the module path, it will be selected in preference to `foo/bar/baz.lss` off the second directory in the module path. The found file must begin with a package statement that declares that it is, in fact, the definition of the package. If it is missing the package declaration an error will be emitted. Note that in the above example, `baz.lss` must contain the line:

```
package foo.bar.baz;
```

Subpackages are declared by using the `subpackage` statement. The syntax for the statement is as follows:

```
subpackage package-name {
    ...
}
```

The name of the subpackage will be the name of the current package concatenated with a dot concatenated with the given package name. Note that subpackages *cannot* be imported directly. They will automatically imported when their parent package gets imported.

It is recommended that one create a subpackage for each module in a package that will define globally visible types that are specific to the module (especially `enum` types). It is probably a good idea to define module local events in this subpackage also.

A common paradigm for implementing packages is to have a single file which includes, *not imports* other `.lss` files which contain the actual definitions of interesting things.

## Domains

A domain is an extension library to LSE. Domains are split into domain classes, domain implementations, and domain instances. In LSS domain classes are typically implemented as an LSS package. Domain instances are created by calling a function inside the domain class package. An argument is passed to that function which indicates which domain implementation to use.

### Creating a Domain Class

A domain class is a package in LSS. This package includes a function which allows the creation of domain instances of this class. To create this function, one uses the `new domain` expression. The syntax for the expression is:

```
new domain(exprname)
```

where *expr<sub>name</sub>* evaluates to a `string` which identifies this domain class. The type that this expression evaluates to is `LSE_domain_constructor` whose type definition is:

```
typedef LSE_domain_constructor : fun (string, string, string) => domain ref;
```

This function takes three `string` arguments. The first argument is the name of the domain instance. The second is the name of the domain implementation. The third argument is an argument list to pass to the domain implementation. An example domain class looks like the following:

```
package LSE_emu;

var class_name = "LSE_emu" : const string;
var create = new domain(class_name) : const LSE_domain_constructor;

...
```

### Domain Types

A domain type is one whose specific definition is determined by a particular domain instance. A domain type, represented by the type `LSE_domain_type` is actually an overloaded function from a `domain`

`ref` to a `type` and from no arguments to a `type`. The actual type is built from the `external` type constructor. As a convenience, a function is provided which will help define domain types. This function is `LSE_domain_type_create`. This function takes two arguments and returns a `LSE_domain_type`. The first argument is a string which is the domain class to which this type belongs. The second is the name of the underlying external type. Continuing the above example, the following code defines a domain type from the `LSE_emu`:

```
var LSE_emu_addr_t =
    LSE_domain_type_create(class_name, "LSE_emu_addr_t") :
    const LSE_domain_type;
```

## Using Domains

To use a domain class, one must create an instance by calling the appropriate function. In the running example, this function is the `create` function defined with the `new domain` expression. This function returns a `domain ref` which is the handle to the domain instance. It is most commonly used with domain types and inside of `${}` expressions to call LSE APIs.

As was mentioned previously, `LSE_domain_type` is an overloaded function. The noary (version with no arguments) of this function obtains the appropriate domain instance from a per-instance context of *default* domain instances. If not explicitly set, the first domain instance of a particular class becomes the default for that class. This default is automatically propagated down the instance hierarchy. It can be overridden by setting the appropriate field of the `LSE_domain` parameter on instances. The fields are named based on the defined domain classes. For example, you could set the default implementation of the `LSE_emu` domain class with the following code:

```
inst.LSE_domain.LSE_emu = expr;
```

provided that `expr` evaluated to a domain instance that was part of the `LSE_emu` domain class.

Similarly, to allow for default instances when calling LSE APIs, there is something known as the domain search path. A module which uses a particular domain, should add the default instance from the `LSE_domain` parameter to the domain search path so that the behavioral code for leaf modules and API calls inside of code typed parameters can work without explicit reference to a particular domain instance. The function that allows this is called `add_to_domain_searchpath` and it takes one argument. The argument is a `domain ref` and this domain instance will be added to the search path.