Liberty Simulation Environment
Developer’s Manual

The Liberty Research Group
# Table of Contents

Preface ........................................................................................................................................i

Typographical conventions used in this book .............................................................................i

I. Modules .....................................................................................................................................i

   1. Things I want to say but haven’t organized yet .................................................................1

II. Domains and other extensions ...............................................................................................1

   2. Extending LSE through domains ......................................................................................2
       General concepts ..................................................................................................................2
       Describing a domain class .................................................................................................2
       Structure of the files ..........................................................................................................3
       Structure of the Python module ........................................................................................3
       Structure of the m4 macro file ..........................................................................................4
       Options for implementing identifiers .............................................................................5
       How to define identifiers ....................................................................................................7
       Declaring user-visible identifiers in Python .....................................................................7
       Implementing identifiers in the m4 file .............................................................................8
       Additional attributes .........................................................................................................9
       Including header files ......................................................................................................9
       Structure attributes ..........................................................................................................9
       Declaring the back-end interface ....................................................................................10
       Other domain attributes ..................................................................................................10
       Hooks ................................................................................................................................10
       Other considerations .........................................................................................................12
       Passing variables or function pointers to domain implementations .............................12
       An example domain class ..............................................................................................12
       Implementing a domain ....................................................................................................12

3. The Command-Line Processor .............................................................................................14

       General concepts .............................................................................................................14
       The standard command line processor ............................................................................14
       Interface the command-line processor must provide ....................................................15
       Interface provided to the command line processor .........................................................15
       Datatypes and variables ....................................................................................................16
       APIs for argument parsing ...............................................................................................16
       APIs for initialization and finalization .............................................................................17
       APIs for simulator control ...............................................................................................17

III. Emulators .............................................................................................................................19

4. The LSE/Emulator Interface ...............................................................................................20

       General concepts .............................................................................................................20
       What is an emulator? ........................................................................................................20
       Interface goals ..................................................................................................................20
       How are emulators interfaced? .......................................................................................21

       State and Contexts .........................................................................................................21
       State within a single context ..........................................................................................21
       Context management ......................................................................................................22
       Sharing state between contexts ......................................................................................23
5. LSE/Emulator API details ................................................................. 30
   Preparing an emulator for use with LSE ............................................. 30
   The emulator description file ........................................................ 30
   The base emulator interface .......................................................... 33
      Datatypes, variables, and functions made available to emulators .......... 33
      Functions an emulator must supply .............................................. 35
   Other requirements ........................................................................... 37
      Context handling ........................................................................... 37
      State spaces ................................................................................ 37
      Decoding and instruction classes ................................................... 39
      Predecoding information ............................................................... 39
      Instruction steps ........................................................................... 39
      Exiting and signal handlers ........................................................... 40
      Error reporting ............................................................................. 41
      Extra functions ............................................................................ 41
      Header files .................................................................................. 41
      Library names .............................................................................. 41
   State-space capability definitions ...................................................... 42
   General capability definitions ........................................................... 42
      The \textit{branchinfo} capability ......................................................... 42
      The \textit{commandline} capability .................................................... 43
      The \textit{disassemble} capability ....................................................... 43
      The \textit{operandinfo} capability ....................................................... 43
      The \textit{operandval} capability ....................................................... 45
      The \textit{speculation} capability ....................................................... 46
      The \textit{threadsafe} capability ......................................................... 47
   Capabilities to be implemented ......................................................... 47
      State-space capabilities ................................................................... 48
         The \textit{extaccess} capability ......................................................... 48
         The \textit{externalize} capability ...................................................... 48
         The \textit{multiinst} capability ......................................................... 48
         The \textit{shareable} capability ....................................................... 48
      Information capabilities ................................................................... 48
         The \textit{itokenrefs} capability ....................................................... 48
      Instruction flow capabilities .......................................................... 49
         The \textit{needtime} capability ....................................................... 49
      Miscellaneous capabilities ............................................................ 49
         The \textit{except} capability .......................................................... 49
   Documenting the emulator ............................................................... 49
   Stuff to incorporate into text ............................................................ 50
State update rules

Emulator-supplied APIs allowed to update internal state

Emulator-supplied APIs not allowed to update internal state
List of Tables

2-1. Class/instance attributes .................................................................................................................. 4
5-1. Description file contents .................................................................................................................. 32
5-2. State space types ............................................................................................................................. 38
Preface

This book is stuff you need to know if you are going to develop modules or domains to use in the Liberty Simulation Environment.

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
I. Modules
Chapter 1. Things I want to say but haven’t organized yet

This part documents how to develop modules, but for now I’m just putting my random thoughts about modules here that I do not want to forget.

**TO DO**

Get the following stuff into chapters

- Rules about when dynids/resolutions/etc. are reclaimed
- Do not use assert inside modules
- Do not use state-updating libc calls (like rand())
- Do not use LSEm4_warn or print statements for debugging inside modules. Definitely do not create debugging parameters to print things out. All of this should be done using events and stat libraries. *If it’s interesting enough to print while debugging, it’s interesting enough to be an event.*
- When making a makefile for a module, be sure to include targets clean and all. .clm files should depend upon a file named remaker (used for forcing rebuild with incremental rebuild)

UI decisions:
-------------

_LIBERTY_SIM_USER_PATH:_ points to directory for module development

-~/modulelib contains tar balls
-~/<module>/Makefile
  /<module>.clm
  /<module>.xml

ls-create-module <name>
- create module under _LIBERTY_SIM_USER_PATH_ (first item)
II. Domains and other extensions
Chapter 2. Extending LSE through domains

This chapter describes an extension mechanism for the Liberty Simulation Environment. This mechanism is called the domain. The chapter provides an explanation of what a domain is and how it should be specified and implemented.

General concepts

Domains are LSE’s principal extension mechanism. A domain (or more properly, a domain class) is a template for an interface, in the “object-oriented” sense of the word interface; a domain class defines types, constants, variables, and methods (API calls) which are to be made available to the writers of modules and configurations. The types, variables, and method signatures are polymorphic. For example, the LSE_emu domain class defines the interface which an emulator presents to the user. The types (such as LSE_emu_addr_t) are polymorphic: different emulators may have different definitions of these types.

A domain implementation is an implementation of a domain class; it indicates how to resolve the polymorphism of the types and implement the methods. For example, the LSE_IA64 emulator is an implementation of the LSE_emu domain class. This emulator defines LSE_emu_addr_t to be uint64_t. Note that a domain implementation is not quite a “class” in the conventional sense of the word. While the implementation says “how” to resolve the polymorphism, it is not actually resolved until the domain is instantiated.

A domain instance is an instantiation of a domain implementation. Each instantiation has separate data (and potentially code). All types are fully resolved by the domain instance. For example, if there are two LSE_IA64 emulators specified in a configuration, there will be a domain instance for each. Similarly, underlying implementation code may be copied for each domain instance.

Domain instances cannot share data, and their types have different names in the system. For example, the LSE_emu_addr_t types of two instantiations of the LSE_IA64 implementation of the LSE_emu domain class are not the same type.

Describing a domain class

To describe a domain class, follow these steps:

1. Create a basic domain Python module and m4 file for the class. These files must have the same base name; if the domain class name is foo, the file names are foo.py and foo.m4.
2. Determine the identifiers which are to be visible to the user. These identifiers are the new API elements added by the domain.
3. Determine any additional internal identifiers you will need.
4. Determine how to implement each identifier.
5. Define each identifier within the Python module and/or an optional m4 macro file for the class.
6. Define additional class and instance attributes (such as hooks)
7. Create an LSS package file for the class. This file must be named foo.lss for a domain class named foo.
8. Install all three domain class files in install_dir/domains.

Structure of the files

Structure of the Python module
The Python module must import the LSE_domain Python module. This module is installed in install_dir/domains. The file must define a Python class named LSE_DomainObject which is a subclass of LSE_domain.LSE_BaseDomainObject. This class is instantiated to create objects describing domain instances. The attributes of the class and of objects of that class inform LSE about constants, types, variables, and methods which the domain class implements, as described in later sections.

The class must contain an attribute className, which is a string indicating the name of the domain class. It must also contain a __init__ method with the following arguments:

- self - a reference to the new class instance
- instname - a string with the name of the domain class instance.
- implfile - a string with the name of a Python file describing the domain implementation. The implementation name (which is used only for documentation and error reporting) is derived from the name of this file.

The __init__ method must begin by calling the __init__ method for the superclass. The superclass sets the instance implName and instName attributes from its arguments. Also, if the instname parameter is empty, it must return after calling the superclass’s init method and should not create any per-instance attributes.

A minimal domain class Python module for a domain class named foo is given below:

```python
import LSE_domain

class LSE_DomainObject(LSE_domain.LSE_BaseDomainObject):
    className = "foo"

    # class attributes go here

    def __init__(self, instname, params):
        LSE_domain.LSE_BaseDomainObject.__init__(self, instname, implfile)

        # Is this a class object we are creating?
        if not instname: return

        # here we assign per-instance attributes
```
Class and instance attributes are used to describe identifiers and the domain class or implementations; a list of these attributes is given here, with additional description to follow in later sections.

Table 2-1. Class/instance attributes

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>changesTerminateCount</td>
<td>integer</td>
<td>Does the domain class manipulate the termination count for the simulator?</td>
</tr>
<tr>
<td>classCompileFlags</td>
<td>list of strings</td>
<td>C compilation flags for the domain class, usually include paths for special header files (such as glib).</td>
</tr>
<tr>
<td>classHeaders</td>
<td>list of strings</td>
<td>C header files used by the domain class</td>
</tr>
<tr>
<td>classHooks</td>
<td>list of strings</td>
<td>framework hooks used by the domain class</td>
</tr>
<tr>
<td>classIdentifiers</td>
<td>list of identifier definitions</td>
<td>per-class identifiers to be made available to the user</td>
</tr>
<tr>
<td>className</td>
<td>string</td>
<td>Name of the domain class</td>
</tr>
<tr>
<td>mergedIdentifiers</td>
<td>list of identifier definitions</td>
<td>per-class identifiers to be made available to the user, but which combine information from all instances</td>
</tr>
<tr>
<td>backendFuncs</td>
<td>list of interface definitions</td>
<td>backend functions for the implementation</td>
</tr>
<tr>
<td>implFile</td>
<td>string</td>
<td>Name of the domain instance implementation file (set automatically)</td>
</tr>
<tr>
<td>implName</td>
<td>string</td>
<td>Name of the domain instance implementation (set automatically)</td>
</tr>
<tr>
<td>instCompileFlags</td>
<td>list of strings</td>
<td>C compilation flags for the domain instance, usually include paths for special header files (such as glib).</td>
</tr>
<tr>
<td>instHeaders</td>
<td>list of strings</td>
<td>C header files used by the domain instance</td>
</tr>
<tr>
<td>instHooks</td>
<td>list of strings</td>
<td>framework hooks used by the domain instance</td>
</tr>
<tr>
<td>instIdentifiers</td>
<td>list of identifier definitions</td>
<td>per-instance identifiers to be made available to the user</td>
</tr>
<tr>
<td>instName</td>
<td>string</td>
<td>Name of the domain instance (set automatically)</td>
</tr>
</tbody>
</table>

If any errors are encountered during execution of code in this file, a LSE_domain.LSE_DomainException exception should be raised. The argument to this exception is a string describing the error. Do not raise SystemExit.

Structure of the m4 macro file

The m4 macro file is divided into "pieces". Everything placed within a "piece" has a particular purpose and is placed into the generated code in specific places. The pieces are:

- *header* - this piece contains constant, type, external variable, external function, and C macro declarations. It may also include static function definitions. The text in this piece is placed in a header
Chapter 2. Extending LSE through domains

file which is included by all code files in the generated simulator. It should not include non-static function definitions. This piece begins with the word `LSE_domain_header`.

• **code** - this piece contains variable and function definitions. The text in this piece is placed in a code file. All the declarations in the header piece are available to text in this piece. This piece begins with the word `LSE_domain_code`.

• **macros** - this piece contains m4 macro definitions. The text in this piece is placed in the header file, the code file, and every module’s code file. It should not contain anything but m4 macros. This piece begins with the word `LSE_domain_macros`.

There is also a distinction between per-class and per-instance pieces. Per-class pieces are inserted once for the domain class. Per-instance pieces are inserted once for each instance of the domain class. A per-class piece begins with the word `LSE_domain_class_defs` while a per-instance piece begins with the word `LSE_domain_inst_defs`.

Switching between pieces is accomplished by placing the word for the piece as defined above into the file. Both the per-class-instance and "kind of piece" word are necessary. An example outline of a domain class m4 macro file is given below:

```c
LSE_domain_class_defs
LSE_domain_macros
/* Per-class macro text goes here */
LSE_domain_header
/* Per-class header text goes here */
LSE_domain_code
/* Per-class code text goes here */
LSE_domain_inst_defs
LSE_domain_macros
/* Per-instance macro text goes here */
LSE_domain_header
/* Per-instance header text goes here */
LSE_domain_code
/* Per-instance code text goes here */
```
Chapter 2. Extending LSE through domains

Options for implementing identifiers

Probably the hardest part of creating a domain class is determining how and where identifiers are to be defined.

Decision 1: per-class or per-instance?

The first decision is easy: is the identifier one that should be defined on a per-class or per-instance basis? For example, in the LSE_emu domain class, a variable for counting the total number of emulation contexts of any instance is a per-class variable, while a count of how many context a particular domain instance has instantiated is a per-instance variable.

A user-visible per-class identifier must be declared in either the classIdentifiers attribute or the mergedIdentifiers attribute of the Python class for the domain class. A user-visible per-instance identifier must be declared in the instIdentifiers attribute of the Python object for the domain implementation.

Decision 2: defined by LSE or defined by you?

The next decision is whether to allow LSE to create the identifier definition from a specification you supply in the Python file or whether to make the definition yourself in the m4 file. The distinction is best explained using variables: you may specify the variable as you declare its identifier by giving its type and initial value; or, you can place a C statement declaring the variable in the appropriate portion of the m4 file.

Internal identifiers must be defined in the m4 file. User-visible identifiers may be defined in either fashion, though we recommend that you allow LSE to create as many definitions as possible. There are some identifiers which have additional restrictions:

- User-visible types must be defined in the Python file.
- Tokenizer macros must be defined in the Python file.
- C functions must be defined in the m4 file.

At this point there are no additional decisions to make for constants, types, and variables. However, for functions there are two additional decisions.

Decision 3: generated code or a library?

The first additional decision is whether the function is to be completely implemented by generated code or whether a library for the domain instance is to be called. This is not an all-or-nothing decision; it is possible to split the functionality between generated code and library code. If library code is used, the interface between generated code and library code is called a back-end interface. We expect that domain classes will often use back-ends; the LSE_emu domain class is an example of such a domain class.

All functions included in the back-end interface must be declared in the backendFunc attribute of the Python domain instance object. This is an instance attribute so that different instances can support different back-end functions.
Chapter 2. Extending LSE through domains

Open Issue

At present, name clashes between identifiers in back end libraries are not resolved and will result in link-time errors. As a result, it is generally not possible to create multiple instances of a domain class using a backend. This will be resolved in future releases, but may involve the creation of dynamically loadable modules for the backends.

Decision 4: function or macro?

The second additional decision is whether the function really should be a function or whether it should be a macro. Clearly, this only applies if some portion of the function is implemented by generated code. Macros cause more code expansion than functions, but generally improve code optimization. A sub-decision is what kind of macro it should be. There are three options to choose from: a C macro, an m4 macro, or an internal "tokenizer" macro. Use of "tokenizer" macros is not described here as it is somewhat involved and subject to change, though it is the method by which many core APIs are defined. You should avoid the use of m4 macros unless you are well-versed in m4.

How to define identifiers

Declaring user-visible identifiers in Python

User-visible identifiers are declared in the classIdentifiers, instIdentifiers, or mergedIdentifiers attributes of the domain class or instance Python objects. These three attributes are lists of 3-tuples, one tuple per identifier.

The first element of each declaration tuple is the name of the identifier, expressed as a string. The second element of the tuple is the type of identifier. Possible identifier types are listed later. The third element is the implementation of the identifier, where None indicates that LSE should not create the implementation because it is supplied elsewhere, either in m4 file or an enumerated type definition.

The possible identifier types and the formats of the implementation elements are:

- LSE_domain.LSE_domainID_const - a constant; the implementation element is its value. Implementations given in the tuple should be for integer-valued constants only.
- LSE_domain.LSE_domainID_type - a type; the implementation element is a string containing the C code defining the type.
- LSE_domain.LSE_domainID_var - a variable; the implementation element a 2-tuple. The first element of this tuple is a string containing the type and the second is the initial value. If the initial value is None, no initial value is generated.
- LSE_domain.LSE_domainID_func - a C function or macro; the implementation element must be None.
- LSE_domain.LSE_domainID_m4macro - an m4 macro; the implementation element must be None.

There is a special kind of per-class identifier called merged identifiers. These are identifiers, typically constants or structure or union types, which use information from all instances of a domain. The attribute
which describes these is `mergedIdentifiers`; it is a class attribute. This attribute should be set by a method of the domain class called `createMergedInfo` which has two parameters:

- `self` - the module class object
- `objlist` - the list of module instances for this class.

All the identifiers defined through the Python file are made available with domain instance notation to the end user. Thus, if a domain class defines a per-instance type `foo_t`, the user may use `foo_t(domaininst)` to distinguish between different instances’ implementations of `foo_t`.

### Implementing identifiers in the m4 file

Any identifier which does not have an implementation in the domain class Python file must have an implementation in the m4 file. The nature of the implementation depends upon the nature of the identifier:

- Constants should be implemented as C or m4 macros. Define C macros in the header piece of the file, and m4 macros in the macros piece.
- Types should be implemented as C types. Define types in the header piece of the file.
- Variables should be implemented as C variables. Declare the variables as "extern" in the header piece of the file and put their definitions in the code piece of the file.
- C functions which are to be shared among module instances should be declared "extern" in the header piece of the file with their definitions in the code piece of the file. Functions to not be shared (and presumably to be inlined) should be declared "static" in the header piece of the file and have their definition there.
- C macros should be declared in the header piece of the file.
- m4 macros should be defined in the macros piece. These definitions cannot be made through the normal m4_define macro; you must use `LSE_domain_class_define` and `LSE_domain_inst_define` for per-class and per-instance macro definitions, respectively. Both these macros take the same arguments that `m4_define` does. When the newly defined macro is expanded and it is a per-instance macro, its argument is shifted "right" by one; the new first argument is the domain instance name.

All identifiers in the file with scope outside of a function must be either user-visible identifiers or must be "wrapped" with a macro call that gives the identifier a unique name in the presence of multiple domain instances. The macros which are used to wrap these names are `CLASSID` for per-class identifiers and `INSTID` for per-instance identifiers. These macros may not be used to wrap the names of user-visible identifiers.

If you find it necessary to use m4 quotes, they are set to control characters by LSE when parsing the m4 macrofile. The open quote character is `Control-__` (\037), while the close-quote character is `Control-^` (\036).

It is also possible to embed Python code in the m4 macro file using a macro called `m4_pythonfile`. Any output of the embedded Python to `sys.stdout` is inserted into the m4 text buffer and reparsed.

While in a per-class piece, the macro `LSE_domain_class_name` gives the class name in text; the variable `LSE_domain_class` points to a special domain instance object in Python used to represent the
Warning

The CLASSID, INSTID, LSE_domain_class, and LSE_domain_inst macros as well as the LSE_domain_class and LSE_domain_inst variables are only available while the m4 file is being processed; they are not available while macros defined in the m4 file are expanded in other user code. This can cause some surprises when defining m4 macros. The correct way to deal with this is to expand these macros (by coming out of quotes) while defining the new macro.

Additional attributes

Including header files

If the domain class requires special headers on a per-class or per-instance basis, these can be declared with the classHeaders and instHeaders attributes for the domain class or instance. Each of these is a list of strings; each string is a header file name. These headers will be included by a header file generated from all domains.

Structure attributes

A domain class can add attributes to some LSE simulator structures. The structure types which can be modified are LSE_dynid_t and LSE_resolution_t. Structure attributes are added by assigning a value to the attributes attribute of the domain class. This attribute is a Python mapping; the keys are the structure name and the values are the structure definitions. For example,

    attributes = { "LSE_resolution_t" : "int foo;" }

adds an attribute to LSE_resolution_t which is an integer named foo.

Domain classes which add attributes must also define a method checkAttribute in their domain class. This function must return a string which is the C-code accessing a given attribute must be made or None if the attribute is not valid. The parameters of this method are:

- self - the module class object
- struct - a string giving the simulator structure name referenced (e.g. LSE_dynid_t)
- attrname - a string giving the attribute name referenced.

Continuing the previous example, class LSE_DomainObject should have a method:

```python
def checkAttribute(self, struct, attrname):
    if struct == "LSE_resolution_t":
        if attrname == "foo": return "foo"
```
Declaring the back-end interface

The back-end interface must be declared by setting the `backendFuncs` attribute. This attribute is a list of three-tuples. The first element is a string giving the name of the backend function. The second is a string giving the return type. The third is a string containing the argument list.

Calls to back-end functions must use the `LSE_domain_invoke` macro. The first argument is the back-end function name, while the remaining arguments are the arguments of the back-end function.

Other domain attributes

Some domains affect the termination conditions of the simulator. They do this by modifying the simulator variable `LSE_sim_terminate_count`. Such domains must set the class attribute `changesTerminateCount` to a non-zero value.

Hooks

Hooks are functions supplied by a domain class which are called by the framework to perform functions such as initialization, argument parsing, finalization, etc. Hooks may apply to an entire class (a class hook) or on a per instance basis (an instance hook).

It is important to understand that hooks are supplied by a domain class, not an implementation. The code for hooks is placed by the framework into the generated simulator code; it is not part of a domain implementation library. Hooks may call functions in an implementation library, but themselves remain outside of it. Many of the hooks will essentially be "wrappers" for implementation functions.

Hooks which are implemented must be declared in the domain class Python file as two attributes: `classHooks` and `instHooks`. The format of each attribute is a list of strings; each string is a hook name. The first attribute lists hooks which will be called once for the domain class. The second attribute lists hooks which will be called once per domain instance of that domain class. The class hooks are always called before the per-instance hooks. It is possible to list the same hook in both attributes; in such a case there is both a class hook and a per-instance hook.

The implementation of the hooks must be provided in the domain's `m4` file in the code pieces, with hooks appropriately placed in per-class and per-instance definition sections. Hook implementations are simply C functions where the function name is the hook name wrapped by the `LSE_domain_hook` macro, as shown in the hook list given below.

The hooks which can be supplied by a domain class or instance are:

```c
void LSE_domain_hook(dynid_dump) (LSE_dynid_t d);

   Called when a debug message for a dynid is being printed. Should print attributes of the dynid believed to be helpful in identifying it during debugging to LSE_stderr.

void LSE_domain_hook(dynid_reclaim) (LSE_dynid_t d);
```
Chapter 2. Extending LSE through domains

Called when a dynid is reclaimed (moved to the free list).

```c
void LSE_domain_hook(end_of_timestep) (void);
```

Called at the end of a simulation timestep, after module end of timestep functions are called.

```c
int LSE_domain_hook(finalize) (void);
```

Finalize the domain class or instance. Return a non-zero value on error.

```c
int LSE_domain_hook(finish) (void);
```

Called when a simulation run finishes. Return a non-zero value on error.

```c
int LSE_domain_hook(init) (void);
```

Initialize the domain class or instance and prepare to parse arguments. Return a non-zero value on error.

```c
int LSE_domain_hook(parse_arg) (int argc, char *arg, char *argv[]);
```

Parse a single command-line argument `arg`, which may have additional following arguments in `argv`. `argc` is the length of `argv` plus 1 (for `arg`). Must return the number of arguments used, including `arg`; 0 for an error. Error messages should be printed to `LSE_stderr`. If `arg` is not valid for this domain class or instance, it should be considered as a user error and reported as such.

```c
int LSE_domain_hook(parse_leftovers) (int argc, char *argv[], char **envp);
```

Parse any remaining command-line arguments which were not parsed by specific domains or the simulator. The number of arguments remaining is `argc` and these arguments are in `argv`. The environment to use for any target program execution is also provided in `envp`. Must return the number of arguments accepted; return a negative number to report an error. Error messages should be printed to `LSE_stderr`.

```c
void LSE_domain_hook(resolution_reclaim) (LSE_resolution_t r);
```

Called when a `LSE_resolution_t` structure is reclaimed (moved to the free list).

```c
bool LSE_domain_hook(should_skip) (LSE_time_numticks_t skipper);
```

Called to determine whether the simulator should skip time; if `FALSE` is returned, the timestep will not be skipped. The parameter `skipper` indicates how many time steps are to be skipped. This hook is only called when the simulator is considering skipping time; it is not guaranteed to be called on every time step. It may be called multiple times per time step, in which case `skipper` will monotonically increase each time it is called within the same timestep.

```c
int LSE_domain_hook(start) (void);
```
Chapter 2. Extending LSE through domains

Called when a simulation run is about to start, before the simulation module instances are initialized. Return a non-zero value on error.

```c
void LSE_domain_hook(start_of_timestep)(LSE_time_numticks_t skipper);
```

Called at the beginning of a simulation timestep, before module start of timestep functions are called. The number of timesteps skipped (if any) is given by `skipper`.

```c
void LSE_domain_hook(usage)(void);
```

Print usage for the domain class or instance to `LSE_stderr`.

Note that the name of the hook is wrapped in a macro call; this will generate the proper unique function name (much like a `FUNC` macro does within a module). No hooks are ever required; if a particular domain class has nothing to place in a particular hook, it merely leaves the hook out of the appropriate list.

**Other considerations**

**Passing variables or function pointers to domain implementations**

Domain implementations should not attempt to directly access simulator or domain variables or functions. All such accesses should be made indirectly; this eases use of dynamic libraries to implement domains. The back-end initialize method for the domain implementation should be passed a structure containing pointers to the variables and functions to be accessed by a domain implementation as part of the per-instance initialization hook defined by the domain class.

**An example domain class**

```
TO DO
```

For now, look at `LSE_emu`.

**Implementing a domain**
Chapter 2. Extending LSE through domains

TO DO

Write this down
The APIs and variables available to you (none so that dynamic library implementations are reasonable)
The APIs you must provide (depends upon the domain)
Chapter 3. The Command-Line Processor

This chapter describes requirements for the command-line processing and main function of a front end for a Liberty Simulation Environment simulator.

General concepts

An important goal of the LSE software structure is to allow LSE to be integrated with other tools. The domain concept described previously allows LSE to embed other components as libraries. LSE itself can also be embedded within other tools. Furthermore, LSE should also be able to have different front ends of its own, e.g. a text-based front end, stand-alone front ends, or a graphical front end.

To support these goals, a final simulator binary has three components which are linked together: the command-line parser, the built simulator, and domain libraries (e.g. emulators). The command-line parser is the front-end or "other tools" which embed LSE.

The command-line parser contains the main function and is responsible for passing command-line arguments to the simulator, calling initialization and finalization routines, catching signals, and calling the simulation main loop. It may also have a command-line interface allowing interactive control of the simulator.

The built simulator simulates components, performing actions at the proper time. Domain libraries may be called upon by the simulator to perform further actions.

This chapter gives specifications for the command-line processor (CLP) used to control the final built simulator. While it is described in the context of an interactive, text-based environment, any user interface or embedding system must meet these specifications.

The standard command line processor

The command line which the standard CLP provides is:

```
Xsim [-sim:arg | -dom: [name]:arg | otherargs]...
[binary_name [emulated_prog_args]]
```

Simulator arguments are prepended with `-sim:`.

Domain arguments are prepended with `-dom:`. If `name` is present, it is the name of the domain instance or class. The name (but not the second colon) can be left out when there is a single domain instance.

`otherargs` can be:

- `-script:filename` - script file to run
- `-i` - use interactive interface (after any scripts)

A binary name and emulated program arguments should only be supplied on the command line when
there is a single emulator instance; when there is more than one emulator instance, all emulated program arguments have to be supplied through scripts or the interactive interface. For a compiled-code emulator, the binary name is only used to supply the program name (argv[0]).

Also, neither a binary nor emulated program arguments should be supplied when -script or -i are used. In these cases, either the script or the user (interactively) must supply the binary name (if needed) and emulated program arguments.

Non-LSE-supplied CLP implementations are free to change these arguments (or indeed, provide them in a totally different fashion), but need to remember to have some way to distinguish between simulator and domain arguments.

**Interface the command-line processor must provide**

The CLP must provide a main routine to LSE which must perform (either directly or through some function it calls) the following steps in the order given:

1. Assign a valid file pointer to the variable LSE_stderr. This file pointer will be used by the simulator to report errors. It must remain valid until LSE_sim_finalize is called. It should be an unbuffered file (as stderr(3) normally is); this may require a setbuf(3) call to accomplish.
2. Call an API (LSE_sim_initialize) to initialize the simulator and domains. This prepares the simulator and domains to accept command-line arguments.
3. Parse the command line, asking the simulator and domains about the arguments. Separate API calls (LSE_sim_parse_arg and LSE_domain_parse_arg) must be called for simulator and domain arguments. All arguments after the first unrecognized argument without a leading – are passed to the simulator as left over arguments using (LSE_sim_parse_leftovers).
4. Call an API function (LSE_sim_start) to begin simulation.
5. Enter the simulator main loop. Run until the simulator exits. This may be done one timestep at a time or all at once.
6. Call an API function (LSE_sim_finish) to end simulation.
7. Finalize the simulator by calling the API function (LSE_sim_finalize).
8. Return the exit status provided by the simulator (in LSE_sim_exit_status).

The steps after step 4 may be performed interactively; if so, the CLP should include appropriate checks to see that steps are not skipped.

**Interface provided to the command line processor**

The interface visible to the CLP allows the CLP to parse the command line, control the simulator, and determine when simulation should terminate. The interface consists of several groups of API calls as well as datatypes and variables. Interface definitions are found in LSE_clp_interface.h which is installed in LIBERTY_LSE/include/simulator.
Chapter 3. The Command-Line Processor

Open Issue

How do we create an interactive interface? What interface is available to a CLP for domain manipulation, hierarchy manipulation, parameter manipulation, and domain and/or module-specific commands?

Datatypes and variables

A boolean data type boolean and constants TRUE and FALSE are supplied to the CLP if the CLP is not written in C++.

The following variables are supplied to the CLP:

- int LSE_sim_exit_status is the value which should be returned as the exit status from the simulator when simulation terminates.
- int LSE_sim_terminate_count is a counter; a zero value indicates that no domain class or instance has any further work to do. The variable is initialized to zero if any domain classes or instances can report this; it is initialized to 1 otherwise.
- int LSE_sim_terminate_now is a flag; a non-zero value indicates that a module or domain instance has requested termination of the simulation at the end of the timestep. A negative value indicates that the termination is due to an error. Negative values greater than -100 are reserved for use by LSE.
- FILE *LSE_stderr is a file pointer used by the simulator for reporting errors.

APIs for argument parsing

int LSE_domain_parse_arg(char *domain_inst_name, int argc, char *arg, char *argv[]);

Incrementally parse command-line arguments looking for domain options. The specified domain instance or class name (if any) should be pointed to by domain_inst_name. The first argument to parse should be pointed to by arg while the rest should be pointed to by the elements of argv. This is done so that the CLP may more easily remove a prefix from the first argument. argc is the length of argv plus 1. LSE will parse a single argument with parameters and return the number of command-line arguments used by the argument and its parameters. 0 is returned on error.

int LSE_sim_parse_arg(int argc, char *arg, char *argv[]);

Incrementally parse command-line arguments looking for simulator options. The first argument to parse should be pointed to by arg while the rest should be pointed to by the elements of argv. This is done so that the CLP may more easily remove a prefix from the first argument. argc is the length of argv plus 1. LSE will parse a single argument with parameters and return the number of command-line arguments used by the argument and its parameters. 0 is returned on error.
int LSE_sim_parse_leftovers(int argc, char *argv[], char **envp);
    Parse the left-over command-line options and the environment in which the simulator runs. Returns non-zero if there is an error; if the return value is negative, the CLP should print a usage message.

void LSE_sim_print_usage(void);
    Print the simulator usage message to LSE_stderr.

APIs for initialization and finalization

int LSE_sim_initialize(void);
    Initialize the simulator and domain instances sufficiently to parse command-line arguments. Returns non-zero on error.

int LSE_sim_start(void);
    Initialize the simulator and domain instances (after command-line arguments have been read) to their initial simulation state. This routine can be called multiple times (if LSE_sim_finish is called in between). Returns non-zero on error.

int LSE_sim_finish(boolean dostats);
    Finish simulation. Print statistics reports if dostats is true. Release memory allocated in LSE_sim_start. Returns non-zero on error.

int LSE_sim_finalize(void);
    Finalize the simulator and domain instances. Returns non-zero on error.

APIs for simulator control

int LSE_sim_engine(void);
    Run the simulator to termination. This function is not interruptable by the CLP. Returns a negative number if some sort of error occurred in simulation.

int LSE_sim_do_timestep(void);
    Do a single time step of the simulator; used when CLP wants to control execution at a fine granularity. Returns a non-zero number when the timestep did not occur because the simulation had terminated; the number is negative when the simulation terminated due to an error (such as a lack of
scheduled timesteps) and is positive when termination is due to a normal condition. The CLP should report negative return values to the user.

Open Issue

What if the timestep is a time-skipped one? Do I run it anyway or do I skip it? At present, I run the first non-skipped timestep.

The "known" error status values returned from LSE_sim_engine and LSE_sim_do_timestep are listed below. Individual modules or domain implementations may return other error codes.

- -99 - call to LSE_report_err
- -1 - out of timesteps
- -2 - dynid/resolution limit exceeded
- -3 - unknown port status
III. Emulators
Chapter 4. The LSE/Emulator Interface

This chapter describes the interface between emulators and the Liberty Simulation Environment. This interface is called the emulator interface. The chapter provides an explanation of the important concepts used in the interface and then provides a high-level description of what each portion of the interface does. Programming details of the emulator interface and commands to use to prepare an emulator are given in Chapter 5.

General concepts

What is an emulator?

For LSE, an emulator is a software library which transforms "architectural" state such as register files or memories according to the semantics of some instruction-set architecture (ISA). An emulator declares such state, and often instantiates and maintains it as well. It then provides an interface to simulator modules; this interface transforms the state according to the semantics of the ISA to be emulated. Some emulators may be designed to "stand alone" without simulators by using a simple driver program.

The exact mechanisms by which the emulator transforms the state are not constrained by LSE. An emulator is often an interpreter, but it could be a JIT, a binary translator, an assembly-language pre-processor, or some other system.

No emulator is required by LSE; you could write custom modules or fill code points to perform all ISA-dependent behaviors. Thus the emulator is really an abstraction of architectural state and ISA behavior. Such an abstraction is convenient; it allows ISA behavior to be reused in different microarchitectures and allows the same microarchitecture to be used for multiple ISAs.

The abstraction concept also allows great flexibility in the behavior provided by an emulator. An abstraction need not be complete. For example, when an ISA does something odd that depends upon microarchitectural state, the emulator need not perform that behavior completely, but can "punt" it to the microarchitectural model. Of course, such an emulator imposes constraints upon the microarchitectural models which can be used with the ISA, much as a real ISA imposes constraints upon microarchitectures. As an extreme example, an emulator could provide only the ISA-dependent type definitions, leaving all behavior up to the microarchitectural model.

Interface goals

One primary goal of the emulator interface is to allow generic structural microarchitectural modules to be used with a variety of different ISAs. The performance of the emulator interface itself is only a secondary consideration, as we have found that ISA emulation consumes only a tiny fraction of the execution time of a detailed microarchitectural simulation. One result of these priorities is that emulators must communicate detailed information about the instructions to the LSE at runtime.
The other primary goal of the emulator interface is to support emulators stemming from a variety of sources. Emulators may be hand-generated or they may be machine-generated. They may be simple or complex and may support different degrees of granularity of control of the emulation process and provide differing amounts of information about instruction execution. They will often come from non-Liberty sources. This requirement leads to the introduction of capabilities, as defined in the next section.

### How are emulators interfaced?

An emulator is a software library; the interface between an emulator and LSE consists of a number of function calls (APIs) and datatype definitions. However, to accommodate the wide variety of emulators available, the interface is partitioned into small increments of functionality called capabilities. For example, providing detailed information about memory accesses is a capability. Emulators must support a fixed base interface, but all capabilities are optional. Of course, the more capabilities an emulator supports, the more useful it is for microarchitectural modeling.

The emulator interface is a "back-end" interface; it is not the interface which LSE modules or code functions see. (That interface is called the emulation interface.) The interaction between LSE modules or code functions and the emulator is mediated by LSE, which must translate "front-end" emulation interface API calls by modules and code functions using dynamic instruction IDs into "back-end" emulator interface calls to the appropriate (potentially multiple) emulators. The emulation interface is described in the chapter entitled *Using emulators* in *The Liberty Simulation Environment User’s Manual*.

An emulator is an example of a domain implementation and the concept of "emulators" is an example of a domain. See Chapter 2 for more information about domains.

The basic process for preparing an emulator to work with LSE is simple: you determine which capabilities the emulator supports, and then write "wrappers" around the emulator's functions to provide the API calls and data structures that those capabilities imply. (Of course, if you are starting from scratch or generating code, no wrappers are necessary; you just directly implement the API calls.) You must also write an *emulator description file*. This file lists the capabilities provided by the emulator and defines basic data types. You then compile the code and place the object files in a library.

### State and Contexts

Emulators define and manipulate architectural state. The set of architectural state available for an instruction to operate upon is called an *execution context*. All state definition and manipulation must take place within some execution context.

#### State within a single context

Emulators declare to LSE (through an emulator description file) what *names* are available for accessing architectural state and what size and kind of state are implied by those names. Declared architectural state consists of a set of *state spaces*. A state space has a name, a type, a number of locations, a location width, and a list of capabilities which the emulator provides for the state space.
State spaces can be either \textit{internal} or \textit{external}. Internal state spaces are allocated by the emulator and accessed directly within the emulator. External state spaces are allocated by LSE and accessed through LSE API calls.

The state space type determines how external accesses are performed on the state space. For example, accesses to register spaces which are larger than the register are illegal, while accesses to memory spaces which are larger than the addressing unit are considered as atomic consecutive accesses. The type is also used by some of the information capabilities to determine how data dependencies through the space are classified. It can also affect how external state spaces are instantiated and managed.

The state space capabilities are emulator capabilities which can be applied on a per-state-space basis. The \texttt{extaccess} capability indicates that an internal space can be accessed externally through an API call. The \texttt{externalize} capability indicates than a space which has been declared internal can become an external space at runtime (this is important for sharing state between emulators). The \texttt{multiinst} capability indicates that an internal state space is not automatically shared by all contexts. The \texttt{shareable} capability indicates that an internal state space can be shared between contexts.

Finally, note that an emulator is not required to cooperate with LSE in this fashion. An emulator could declare no state spaces and may completely deal with all state handling within its instruction semantics. However, such an emulator will not be as useful as one that does declare state and provide additional capabilities.

\section*{Context management}

Architectural state within an emulator is itself an abstraction of physical state; the state involved might be spread across multiple physical structures and encoded in a variety of fashions in the simulated microarchitecture. Furthermore, architectural state may be virtualized; multiple execution "threads" might have differing sets of architectural state which "take turns" being mapped to physical state in a real system. As a result, emulators and LSE must cooperate closely to manage the execution contexts and there must be a way of virtualizing the state.

To allow virtualization, LSE defines two kinds of contexts. The first kind of context, a \textit{hardware context}, corresponds to the physical state elements. A uniprocessor simulation would have a single hardware context; multiprocessor simulations would have one context per processor. The second kind of context, a \textit{software context}, represents the virtualization of the physical context and corresponds to processes or threads in an operating system.

LSE maintains mappings between software contexts and hardware contexts. Emulators usually determine what the mappings should be, performing \textit{context switches} (changes of the mapping) as desired, and unmapping a hardware context when there is no code to be run.

Emulators normally operate upon software contexts when executing instructions. Hardware contexts are nothing more than a reference to a software context; when the microarchitectural model references a hardware context, the reference is translated to a reference to the software context mapped to that hardware context. As a result, emulators which don’t actually context switch need not concern themselves with hardware contexts.

Note that LSE’s contexts are simply names for specific sets of instances of architectural state spaces. LSE does \textit{not} have any notion of relationships between contexts, such as parent to child. Such relationships are the responsibility of OS emulation. For example, when a parent software context finishes, the emulator should unmap child software contexts (if those are the OS semantics).
When there are emulators present, simulation normally terminates when no hardware context has a software context mapped onto it. In addition, simulation can terminate because some code in a module or domain instance or the LSE framework raises an error flag.

**Sharing state between contexts**

A context is a name for a specific set of instances of architectural state spaces. Contexts may overlap; for example, two different user-level threads in the same process typically overlap in memory spaces and virtual-to-physical translations, but do not overlap in register spaces. When contexts overlap, state is shared between the contexts.

**Open Issue**

How do we share pieces of a state space? How about the same state having different names in different contexts (as in many shared memory situations)?

State sharing is implemented through mapping. All execution of instructions and state access takes place within a particular single context. LSE maintains context identifiers which are available while emulator functions are executing. These identifiers contain a mapping of state space names to state space instances. LSE automatically performs this mapping for external state spaces. Emulators must perform the mapping for internal state spaces. Internal state spaces which can be shared between contexts within the same emulator have the *shareable* capability.

State sharing between different emulators can happen in one of two ways: either the shared state space is external to all emulators or it is internal to only one emulator and external to all others. The *externalize* capability allows LSE at runtime to make a state space external to an emulator to which it would normally be internal. If all emulators which are to share the space have this capability for the space, LSE can choose one emulator to treat the state space internally and have all others treat it externally. In general, new emulators should either declare state spaces to be external or provide the *externalize* capability to allow maximum state space sharing between emulators.

Some emulators may have internal state spaces but no *multiinst* capability for those spaces. These spaces are shared among all contexts within the emulator. Many common single-context emulators will be in this category. These emulators are still useful. Such emulators can be used for single-context simulations or simulations where the internal state spaces are to be shared among all contexts. These emulators may also be useful for simulations where the internal state spaces are not to be shared if LSE creates multiple instances of the emulator code.

The mappings between contexts and state spaces are created as the contexts are created. How the mappings are specified depends upon who is creating the contexts:

- Emulators can create new contexts during the course of execution (e.g. by emulating a system call). The overlap between the contexts is part of the execution semantics of the ISA or OS being emulated. The emulator instantiates any non-shared internal state spaces and updates its mappings for all internal state spaces. It informs LSE of the new context (through API calls) and then asks LSE to instantiate non-shared external state spaces and update mappings for shared external state spaces.

- When LSE creates new contexts, they are created with new state spaces; the only shared ones will be those which are shared among all contexts in the emulator instance. LSE then (under direction of the
user, a module, or a configuration) asks the emulator to copy individual internal state spaces (for sharing with copy-on-write semantics), externalize individual internal state spaces (if they have the externalize capability), or make the mappings for individual internal state spaces point to some other context.

Mappings between contexts and state spaces are destroyed as contexts are destroyed. Mappings can be changed for both internal and external state spaces; this is how the initialization of sharing is done, though it is not clear of what utility this may be at later points in time. Most dynamic behavior should be handled as a translation state space.

**Note:** Hardware contexts do not share state directly; they share state if the software contexts mapped to them share state.

### LSE-defined state spaces

LSE can declare state spaces and add state spaces to contexts either implicitly (such as when LSE determines that a state space is external to all emulators) or explicitly due to the configuration or module API calls. These state spaces are always external, and often are invisible to the emulators. One common situation in which this will occur is when translation state spaces are used.

A translation state space is a state space which translates references from one state space to another state space. For example, a translation state space can be used to remap logical to physical registers. The way in which this would be used is:

1. The configuration declares a physical register state space.
2. The configuration declares a translation state space from the general register state space to the physical state space.
3. The configuration declares a translation function for the state space. This function looks up logical to physical translations assigned for the dynamic instruction instance (because different in-flight dynamic instructions may have different translations for the same logical register).
4. The configuration declares that the general register state space of the emulator (which must be external) is actually the translation state space.
5. At runtime, as contexts are created, LSE creates new translation state spaces and physical state spaces.
6. At runtime, when an instruction is decoded, the rename table is read by a simulator module which stores the translation information in the dynamic instruction ID for the translation to use.
7. At runtime, as the instruction is executed, the emulator calls the LSE APIs for external state space access. These APIs call the translation function, which looks up the physical register number for the dynamic instruction instance and calls the physical state space access function.
State and the model of computation

The LSE model of computation allows code blocks in the microarchitectural simulator to be executed multiple times in a single time step. This may result in multiple calls to emulator APIs. We do not want the emulator author to have to be deeply concerned with the model of computation. Therefore, LSE module writers and configurers must prevent multiple calls to emulator APIs which update architectural state. The only burden placed upon the emulator author is to not update state in APIs which are not allowed to update state; the Section called State update rules in Chapter 5 states which APIs may or may not update state.

Furthermore, emulator interface data structures are outside of the model of computation; they may change values more than once within a time step. They are carried with (or at least associated with) a dynamic instruction ID structure. Care must be taken by the configurer to prevent this data from being used improperly.

Instruction semantics

The basic behavioral unit of abstraction of an emulator is the instruction. The exact definition of an instruction is intentionally vague; it can be understood in the traditional sense of "an individual command" or as a set of state updates that are related. The semantics of instructions are defined by the emulators; they can be very simple, or somewhat complex.

An instruction’s semantics do not need to be complete. An emulator may choose to not abstract all the instruction behavior, leaving some of it to the microarchitectural model. Of course, such an emulator cannot be used without microarchitectural models that supply the appropriate behavior.

Lifecycle of an instruction

An instruction is thought of as passing through six classic steps:

- fetch: get the instruction from instruction memory
- decode: determine instruction characteristics
- opfetch: 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 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.

Any step may be affected by architectural or, in some cases, microarchitectural state. Timing of the steps is supplied wholly by the simulator.

**Exception semantics**

TO DO

This needs to be reviewed to rethink whether it is the way we wish to do it.

Exceptions require special handling because they can create behavior at any point in the instruction’s life cycle. For example, fetch could be to an illegal address or decode could find an illegal instruction. Exceptions can even create behavior that is semantically "between" instructions, as interrupts do in many architectures. The kind of behavior discussed here are things like storing the PC, making interrupt stack frames, changing status registers, and jumping to an exception handler. For some emulators, the behavior may be to abort simulation. Several issues must be dealt with.

The first issue is who generates the exceptions. There can be exceptions generated by the emulator and exceptions generated by the microarchitecture (LSE). Exceptions generated by the emulator can easily have the appropriate state updates "built in" to the emulator. Microarchitecture-generated exceptions do not. Emulators which support exceptions must provide an API for the microarchitecture to request exception state manipulation through the exception capability.

The next issue is how exceptions are reported to LSE. This is fairly easy; all emulator API calls return an exception status type which indicates the presence of an exception.

The third issue is when or if exception behavior should occur. This should be under control of the microarchitectural model; there will be many cases where the microarchitecture is attempting to execute something speculatively and any exceptions will be ignored in the end. Furthermore, the behavior itself will likely include state space accesses (including memory accesses) needing detailed timing which may themselves generate exceptions. To allow detailed execution of exception semantics, an emulator may provide an exceptprogram capability which allows LSE to see the semantics of the exception handling, much as the iprogram capability allows LSE to see the semantics of an instruction. If this capability is not present, the behavior is considered atomic and is triggered by a call to the API defined by the exception capability.

The final issue is that of classifying exceptions. The emulator description file allows symbolic names and codes for exception types to be defined.

**Cross-instruction semantics**

Some ISAs do not fit well the model of instructions executing in-order independently of each other. These ISAs define cross-instruction semantics; classic examples of such semantics are delayed branches, annulled branches, and register read/write ordering for VLIWs. There are several ways in which such semantics can be dealt with by an emulator.
Chapter 4. The LSE/Emulator Interface

The first way to deal with cross-instruction semantics is not to deal with them; the emulator need not reflect all of these semantics directly for simulation purposes. For example, two parallel instructions in a VLIW "packet" may read and write the same register, but the write is guaranteed to take place after the read even if the writing instruction is "earlier" in instruction memory. While it would be possible to define the instruction as being the entire VLIW packet, it is generally more convenient to treat each instruction in the packet as a separate instruction which simply reads and writes its operands. In such a case, the semantics of the ISA are only partially provided by the emulator; the simulation model must ask the emulator to read and write operands at the proper time to ensure that the cross-instruction semantics are maintained.

Another way of dealing with cross-instruction semantics is through auxiliary state. For example, delayed branches can be dealt with by setting a flag indicating that a branch must take place after the "next" instruction. However, all instructions might need to have an appropriate "epilogue" using such flags added to their semantics.

A third way is to include the semantics in the emulator with some notion of time and require the simulator to notify the emulator when time advances. Note that what "time" means depends upon the ISA semantics; some ISAs may define time as cycles, though a definition in terms of "instructions" is far more likely. Emulators which do this define the needtime capability. It is up to the configurer to ensure that the advance time function is called appropriately.

Finally, a fourth way would be to define the cross-instruction semantics to LSE. However, because the semantics can be so varied, it is difficult to conceive of a way in which LSE modules would use this information. However, as ways to use the information become apparent in the future, capabilities for emulators to define specific kinds of cross-instruction semantics can be defined. One fairly likely future capability would be one which allows the emulator to report about delayed and annulled branches so that the standard module library’s instruction sequencer modules can work with ISAs with delayed and annulled branches.

Capabilities

The following is a list of the capabilities an emulator can support. They are grouped by nature.

State-space capabilities.

- **extaccess** allows external access to the state space
- **externalize** allows LSE to treat the state space as external instead of internal
- **multiinst** internal state space is not the same in all contexts
- **shareable** internal state space can be shared between contexts

Information capabilities.

- **branchinfo** provides branching information
- **operandinfo** provides operand information
### Chapter 4. The LSE/Emulator Interface

**itokenrefs** requires notification when instruction information is no longer useful

**Instruction flow capabilities.**

- **operandval** provides operand value information and provides control of operands
- **speculation** supports recovery from mis-speculation
- **needtime** requires notification of time advancing

**Miscellaneous capabilities.**

- **commandline** has command-line options
- **disassemble** provides a disassembler
- **except** can report exceptions or receive exception reports
- **threadsafe** emulator itself is threadsafe

Each of these capabilities is discussed in detail in Chapter 5.

### Other considerations

This section collects random additional thoughts about emulators:

- As an ISA might be used with more than one OS or it might be used to do detailed simulation of an OS, it might be desirable to separate OS from ISA emulation. LSE does not attempt to facilitate this; we see this as part of the emulator generation problem: how does one obtain an emulator in the first place? One could envision an "emulator generator" that is able to put ISA emulators together with OS emulators, but that is an effort apart from LSE.

- OS code may have callbacks to program code, e.g. signal handlers or the Posix qsort function. How can the code in the callbacks be emulated? There seem to be two options:
  1. The first option is that the emulator can enter some sort of special mode where it emulates multiple instructions until the system call causing the callback returns. This option may mess up some microarchitectures because registers will be changing without going through renaming, though if the system call itself has flushed the pipeline, this probably does not cause a problem. Emulators which have left some behavioral details to the microarchitecture will have difficulties using this option.
  2. The second option is to treat the system call that triggered the callback as a control transfer instruction, rather like a longjmp. The next instruction is the address of the callback. The problem here is how to return from the callback; whatever calling convention is used by the ISA/OS must be faked to allow the return instruction of the callback routine to return to a pseudo-system call that resumes the previous one.
Some emulators may have additional functions which might be of use to a microarchitectural model. For example, one such function might calculate whether a given address is a valid virtual address or not. Emulators may declare additional functions to export to LSE in their emulator description files.

Notes

Chapter 5. LSE/Emulator API details

This chapter describes the details of the interface between emulators and the Liberty Simulation Environment. It describes data structures and function calls needed. It also describes the procedure you must use to prepare an emulation library for use with Liberty. The interface description is organized by capability.

Preparing an emulator for use with LSE

An emulator is an implementation of the LSE_emu domain class. As such, the process for creating the implementation is that of creating a domain implementation. However, for ease of reference, the instructions are repeated here and specialized for emulators.

Preparing an emulator for use with LSE requires the following steps:

1. Pick a name for your emulator. This name should be globally unique. A combination of the ISA name and your project name would make a good name. The name must consist only of characters valid in a C identifier, must not contain a double underscore (___), and must not begin with LSE, EMU, or m4.

2. Determine the capabilities which the emulator will support.

3. Write an emulator description file named emulator_name.dsc. The format of this file is described in the Section called The emulator description file.

4. Generate a header file with all the datatypes and prototypes for the emulator interface. This is done by running:

   ls-make-domain-header LSE_emu {description_file (w/o .dsc)} {header_file}

   You may name the header file anything you choose. The Liberty environment variables must be set when you run the script.

   Whenever you modify the description file you must repeat the preparation procedure starting at this step (not throwing away old emulator source code, of course).

5. Write/modify the emulator source code or wrappers.

6. Compile the emulator, placing the object code into a library.

7. Install the library into $LIBERTY/lib/domains. Install the description file into $LIBERTY/share/domains/LSE_emu.

8. Document your emulator as described in the Section called Documenting the emulator.

You should write and build your emulator source code outside of the normal Liberty directory structures because your emulator is not part of the Liberty distribution.
The emulator description file

The emulator description file defines attributes and capabilities which the emulator has. The file is a Python script, but you do not need any knowledge of Python to write an emulator description. The syntax rules are very simple:

1. Definitions have the form attribute = value.
2. Definitions must begin in the first column; white space is legal between any token after this.
3. Comments begin with a number sign (#); they must also begin in the first column unless there is text before them on the line.
4. Simple strings are enclosed in either single or double-quotes; strings with newlines are enclosed in triple double-quotes.
5. Lists are made using square brackets and commas, i.e.: ['item1','item2']
6. Tuples are made using parenthesis and commas, i.e.: (1,2)
7. Blank lines must be completely blank with no invisible spaces or tab characters.

An portion of a description file illustrating the syntax is given below:

```python
# Emulator name
name="LibertySample"

value=3

# Interface capabilities supported
capabilities=[
    "branchinfo", # provides branch information
    "fork", # can fork new contexts
]

# Private static info (C-style structure)
privatestatic=""
struct {
    uint32_t target_addr;
    void *(host_addr)();
}

# A random attribute
a = (3, 4)
""
```

1. A comment
2. A string attribute definition
3. An integer attribute definition
4. A list of strings attribute definition
5. A multi-line string attribute definition
A tuple attribute definition

The following table lists all the possible attributes; details of how they are used can be found in the corresponding sections for the capabilities which require the attribute. Attributes without a default value must be assigned a value if their corresponding capability is present in the emulator. If no capability is given, the attribute applies to all emulators. Further descriptions of what the attributes are used for are given as required in later sections.

Table 5-1. Description file contents

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Default value</th>
<th>Capability</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>addrtype</td>
<td>string</td>
<td>—</td>
<td>—</td>
<td>C-type for addresses in ISA</td>
</tr>
<tr>
<td>addrtype_print_format</td>
<td>string</td>
<td>—</td>
<td>—</td>
<td>C-format-specifier for printing addrtype</td>
</tr>
<tr>
<td>capabilities</td>
<td>list of strings</td>
<td>[]</td>
<td>—</td>
<td>Capabilities provided by emulator</td>
</tr>
<tr>
<td>compileFlags</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>—</td>
<td>Flags to use for compilation of simulators using this emulator; usually specifies include paths for header files</td>
</tr>
<tr>
<td>compiled</td>
<td>int</td>
<td>0</td>
<td>—</td>
<td>Does the emulator do compiled-code emulation?</td>
</tr>
<tr>
<td>ctokentype</td>
<td>string</td>
<td>—</td>
<td>—</td>
<td>C-type for context token</td>
</tr>
<tr>
<td>extrafields</td>
<td>string</td>
<td>empty</td>
<td>—</td>
<td>Extra fields for LSE_emu_instr_info_t</td>
</tr>
<tr>
<td>extrafuncs</td>
<td>special</td>
<td>[]</td>
<td>—</td>
<td>Extra functions to export to the simulator. See the Section called Extra functions.</td>
</tr>
<tr>
<td>headers</td>
<td>list of strings</td>
<td>[]</td>
<td>—</td>
<td>A list of header files which should be included by simulators using this emulator.</td>
</tr>
<tr>
<td>iclasses</td>
<td>list of strings</td>
<td>[]</td>
<td>—</td>
<td>Instruction classes decoded by emulator</td>
</tr>
<tr>
<td>libraries</td>
<td>string</td>
<td>—</td>
<td>—</td>
<td>Library file name</td>
</tr>
<tr>
<td>max_branch_targets</td>
<td>int</td>
<td>—</td>
<td>branchinfo</td>
<td>Maximum number of potential next instructions</td>
</tr>
<tr>
<td>max_operandDest</td>
<td>int</td>
<td>—</td>
<td>operandinfo</td>
<td>Number of potential destination operands</td>
</tr>
<tr>
<td>max_operand_int</td>
<td>int</td>
<td>0</td>
<td>operandval</td>
<td>Number of potential intermediate operands</td>
</tr>
<tr>
<td>max_operand_src</td>
<td>int</td>
<td>—</td>
<td>operandinfo</td>
<td>Number of potential source operands</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>—</td>
<td>—</td>
<td>Name of emulator</td>
</tr>
<tr>
<td>operand_names</td>
<td>list of tuples</td>
<td>[]</td>
<td>operandinfo</td>
<td>Operand names and associated values</td>
</tr>
<tr>
<td>operandvaltype</td>
<td>string</td>
<td>—</td>
<td>operandval</td>
<td>Operand data value type</td>
</tr>
</tbody>
</table>
Chapter 5. LSE/Emulator API details

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Default value</th>
<th>Capability</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecodefields</td>
<td>list of strings</td>
<td>[]</td>
<td>—</td>
<td>Names of fields of LSE_emu_instr_info_t which are to be moved to LSE_emu_predecode_info_t</td>
</tr>
<tr>
<td>privatefields</td>
<td>string</td>
<td>empty</td>
<td>—</td>
<td>Extra fields not visible to LSE for LSE_emu_instr_info_t</td>
</tr>
<tr>
<td>statespaces</td>
<td>special</td>
<td>[]</td>
<td>—</td>
<td>State space descriptions. See the Section called State spaces.</td>
</tr>
<tr>
<td>step_names</td>
<td>list of tuples</td>
<td>—</td>
<td>—</td>
<td>Execution step names, classification, and associated values</td>
</tr>
</tbody>
</table>

**The base emulator interface**

The base emulator interface does not have a capability name. It provides initialization routines and an simple instruction lifetime interface suitable for coarse simulations. This interface is simply a "frontend" function that normally performs fetch and decode, a "backend" function that normally performs evaluate, writeback, and commit, and a "commit" function that normally performs commit. Note that not all ISAs will function properly with just the base emulator interface because of cross-instruction semantics (e.g. classic VLIW).

**Datatypes, variables, and functions made available to emulators**

**Datatypes**

The datatypes listed below are provided to the emulator. They equal the corresponding datatypes in the emulation interface, but emulator manipulates the fields of structures directly rather than through accessor macros.

- **LSE_emu_addr_t** is the address type defined in the addrtype attribute in the emulator description file.
- **LSE_emu_contextno_t** is the global context number type.
- **LSE_emu_ctoken_t** is the emulator context token type defined by the ctokentype attribute in the emulator description file.
- **LSE_emu_instr_info_t** contains instruction information for a dynamic instruction instance. It has fields:
  - LSE_emu_addr_t addr; - address of the instruction.
  - LSE_emu_contextno_t contextno; - global context number of the instruction.
  - LSE_emu_ctoken_t contexttok; - emulator context token of the instruction’s context.
  - struct {...} iclasses; - instruction classes.
Chapter 5. LSE/Emulator API details

The structure is filled with definitions of the form: boolean is_class; for each instruction class in the iclasses attribute in the description file. The order of the definitions is the order listed in the description file.

- LSE_emu_addr_t next_pc; - address of the next instruction which should be executed.
- LSE_emu_predecode_info_t *pre_info; - pointer to predecoded information. Only exists if LSE_emu_predecode_info_t is not empty.
- privatefields privatef; - fields defined by privatefields attribute in description file, if the attribute is not empty.
- extrafields extra; - fields defined by extrafields attribute in description file, if the attribute is not empty.
- LSE_emu_addr_t size; - size of the instruction.

- LSE_emu_instrstep_name_t is an enumerated type whose values are the evaluation step names for an emulator. The values have the form LSE_emu_instrstep_name_stepname. For example, if there is an instruction step named "readmem", there is a value LSE_emu_instrstep_name_readmem. Names are taken from the step_names attribute in the description file.
- LSE_emu_interface_t is a structure which contains pointers to variables and APIs available for use by the emulator.
- LSE_emu_predecode_info_t is a structure which contains fields which have been identified in the predecodefields attribute in the description file as predecoded fields. If the structure would be empty, the type does not exist.
- LSE_emu_spaceaddr_t is a union type which defines the address types for each state space. The fields of the union have the same names as the state spaces. The type of the field depends upon how the number of locations in the state space are specified in the description file. For integer-defined spaces, the type of the field is int. For spaces defined by a number of address bits, the type of the field is the smaller of a 32-bit integer, a 64-bit integer, or a string of bytes with sufficient bits. For spaces defined by a number of characters, the type of the field is an array of characters. There is always a member of the union with type int named LSE.
- LSE_emu_spaceid_t is an enumerated type which defines the state space identifiers. The names of the values are of the form: LSE_emu_spaceid_spacename, where spacename is the name of the corresponding state space as defined in the description file.
- LSE_emu_spacetype_t is an enumerated type which defines the possible state space types. The values are listed in Table 5-2.

Variables and APIs

Variables and APIs are not provided directly to the emulator; pointers to variables and APIs are provided instead. This is done so that dynamically linked emulators may more easily be supported. The pointers are supplied through a structure of type LSE_emu_interface_t passed into the emulator initialization function (EMU_init). This structure has fields containing pointers to each of the following variables and APIs; the fields have the same name as the corresponding variable or API.

The variables accessed through the LSE_emu_interface_t structure are:
Chapter 5. LSE/Emulator API details

- `int LSE_sim_exit_status;` - the exit value which LSE might use when exiting the simulator (the standard CLP uses it, but others might not). This exit status might be used for emulator errors, simulator errors, or even the return status of the target application. No attempt is made by LSE to arbitrate between these uses.

- `FILE *LSE_stderr;` - a stream pointer to be used for reporting errors.

The APIs accessed through the `LSE_emu_interface_t` structure are:

```c
LSE_emu_contextno_t LSE_emu_announce_context (LSE_emu_ctoken_t ctoken, boolean automap);
```

Informs LSE that a new software context with token `ctoken` has been created by the emulator. The automap flag is set to `automap`. The return value must be associated with this context by the emulator for later use when calling the emulator interface.

```c
int LSE_emu_update_context_map (LSE_emu_contextno_t hwcno, LSE_emu_contextno_t swcno);
```

Informs LSE that software context `swcno` is now mapped to hardware context `hwcno`. If `hwcno` is 0 ("no context"), whatever hardware context is mapped to `swcno` is also unmapped.

### Functions an emulator must supply

```c
int EMU_context_create (LSE_emu_ctoken_t *ctokenp, LSE_emu_contextno_t cno);
```

Create a new software context and place its token into the location pointed to by `ctokenp`. The `cno` parameter must be associated with this context by the emulator for later use when calling the emulator interface. Do not immediately map this software context to a hardware context and do not call `LSE_emu_announce_context` for the new context. Return zero on successful creation; non-zero on error, though exiting is allowed on error.

```c
int EMU_context_load (LSE_emu_ctoken_t ctoken, int argc, char *argv[], char **envp);
```

Load a program into the context given by `ctoken`. The program has arguments `argc` and `argv` and environment `envp`. The binary name is `argv[0]`. Set up all initial architectural state for the context. If the context is ready, this function must call `LSE_emu_set_context_state` to indicate this. Return zero on successful completion; non-zero on error.

```c
void EMU_context_map_notify (LSE_emu_contextno_t hwcno, LSE_emu_contextno_t swcno, LSE_emu_ctoken_t ctoken);
```

Note that software context `ctoken` is mapped to hardware context `hwcno`. The parameter `swcno` is included so that unmapping can be detected (as LSE does not know what a "null" value for
void **EMU_do_step** (LSE_emu_instr_info_t *ii, LSE_emu_instrstep_name_t sname);

Perform the execution step named *sname* for instruction *ii*. Instruction information which is used or updated, state that is read or updated, and side effects caused by each step should be documented.

---

**Open Issue**

**Error reporting and exceptions**

---

void **EMU_finish** (void);

Finalize the emulator instance. This function must not call any emulator APIs.

LSE_emu_addr_t **EMU_get_start_addr** (LSE_emu_ctoken_t ctoken);

Return the starting address of the context *ctoken*. The address need not be guaranteed to remain the same after an API which implies execution within the same context is called. This function will not be called until after a program is loaded into the context or the address has been set with **EMU_set_start_addr**.

int **EMU_get_statespace_size** (LSE_emu_ctoken_t ctoken, LSE_emu_spaceid_t sid);

Return the size of state space *sid* in context *ctoken*. This function is only called for state spaces for which the number of locations is not set until runtime. This function is *not* required if no state spaces have string addresses.

void **EMU_hwcontext_automap_changed** (LSE_emu_contextno_t cno, boolean automap);

If *automap* is TRUE, add the hardware context *cno* to the list of hardware contexts which can be automatically mapped by this emulator instance. If *automap* is FALSE, remove the context from the list. When the hardware context is added, if it is not already mapped, try to map it and call **LSE_emu_update_context_map** if successful.

void **EMU_init** (LSE_emu_interface_t cno);

Initialize the emulator instance. After this function is called, the emulator must be ready to create contexts or parse command-line options (if the **commandline** capability is present).
Chapter 5. LSE/Emulator API details

void EMU_init_instr(LSE_emu_instr_info_t *ii);

Initialize any fields in $ii$ which need initialization before an instruction can be executed.

void EMU_set_start_addr(LSE_emu_ctoken_t ctoken, LSE_emu_addr_t addr);

Set the starting address of the context $ctoken$ to $addr$. The address need not be guaranteed to remain the same after an API which implies execution within the same context is called.

void EMU_swcontext_automap_changed(LSE_emu_ctoken_t ctoken, boolean automap);

If $automap$ is TRUE, add the software context $ctoken$ to the list of software contexts which can be automatically mapped by this emulator instance. If $automap$ is FALSE, remove the context from the list. When the software context is added, if it is not already mapped, try to map it and call LSE_emu_update_context_map if successful.

Other requirements

Context handling

The emulator is required to notify LSE whenever it creates software contexts or changes mappings between software and hardware contexts. This is done by calling LSE_emu_announce_context in the former case and LSE_emu_update_context_map in the latter. The only exceptions are when LSE has called the emulator to create the new context or to notify the emulator of a particular mapping.

When the emulator context switches, it normally does not change the mapping for any context which is not on the list of contexts subject to automatic mapping. However, it always changes mappings when a context finishes.

Open Issue

- Destruction of contexts

State spaces

The description file includes information about the state upon which instructions operate. This information is important for many "advanced" capabilities, but is not required if the emulator does not support these capabilities. However, it is simple to describe and we encourage you to provide it for all emulators.

The information is put into the statespaces attribute as a list of tuples. Tuples are formed by using parenthesis and commas, and have the following (ordered) elements:
1. State space name. This is a string and must be unique within the emulator. It must be a valid C identifier and must not contain two underscores in a row.

2. Space type. The possible space types are:

<table>
<thead>
<tr>
<th>Space type</th>
<th>Meaning</th>
<th>Unit for size</th>
<th>Special semantics in the standard module library</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_spacetype_reg</td>
<td>Simple registers</td>
<td>bits</td>
<td>Data dependencies detected</td>
</tr>
<tr>
<td>LSE_emu_spacetype_creg</td>
<td>Control registers</td>
<td>bits</td>
<td>Data dependencies detected</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mem</td>
<td>Memory</td>
<td>bytes</td>
<td>—</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mapping</td>
<td>Mapping</td>
<td>undefined</td>
<td>—</td>
</tr>
</tbody>
</table>

The space type names are also available as constants to the emulator.

3. Number of locations in the state space. The number of locations can be specified in one of three ways:
   - As an integer between 0 and $2^{31} - 1$, inclusive. If the value is less than 0, the number of locations is not fixed until run time. Not fixing the number of locations allows compilers for ISAs without fixed instruction encodings, like Lcode, to use different numbers of registers for different target programs. A state space without a fixed number of locations cannot have more than $2^{31} - 1$ locations.
   - As a string of the form "numberb". The number of locations is $2^{number}$.
   - As a string of the form "numberrc". The number of locations is not fixed until run time, and the addresses of locations are strings with at most number characters (not including a null byte at the end, as in C).
   - As a string "s". The number of locations is not fixed until run time, and the addresses of locations are constant strings in the emulator.

4. Size of an element (in bits or bytes depending on the space type)

5. List of state-space capabilities supported for that state space.

An example of state space definitions is:

```python
statespaces = [
    ("GR", LSE_emu_spacetype_reg, 32, 64, ["externaccesss"]),
    ("SR", LSE_emu_spacetype_creg, "3c", 64, []),
    ("MEM", LSE_emu_spacetype_mem, "64b", 1, [])
]
```

This information is useful in three principal ways:
• It defines the possible identifiers for pieces of state. A state identifier always consists of two numbers: the state space number and the address within the state space. The state space numbers are derived from the `statespaces` attribute; state spaces are numbered starting from zero in the order they are defined by the attribute. These state identifiers are used when describing the semantics or the data dependencies of an instruction.

• It defines how large a state space is and the semantics of access to it. When some capabilities are present, LSE can perform allocation of and access to the state space on behalf of the emulator, thus simplifying state sharing; in such cases, LSE uses the number and size of elements declared.

**TO DO**

Define the exact semantics of state space types

**Decoding and instruction classes**

Emulators must classify instructions and place this information in the instruction information structure. This information typically is provided at some "decode" step. The exact classes which an emulator provides are left up to the discretion of the emulator writer, but every effort should be made to give the classes names and meanings that match the "standard" names as described in the section entitled *Decoding instruction classes* in the chapter *Using emulators* in *The Liberty Simulation Environment Users’ Manual*.

The classes actually provided by the emulator are listed in the `iclasses` attribute in the description file. All emulators must provide the `sideeffect` class.

The results of classification during decode must be placed into the static information structure (`LSE_static_info_t`) into the fields named `iclasses.is_class`.

**Predecoded information**

Some emulators may wish to pre-decode instructions to improve emulation speed. Such emulators can use the `predecodefields` attribute in the description file to indicate that fields of `LSE_emu_instr_info_t` are to be moved from this type to another type named `LSE_emu_predecode_info_t`. This latter type should be the type the emulator uses for storing predecoded information. If the type is not empty, there is a field named `pre_info` added to `LSE_emu_instr_info_t` which is a pointer to predecode information. This pointer must be set by some step of instruction execution, and will be used by LSE for accesses to the fields which have been moved between the types.

Any field but `addr`, `contextno`, and `contexttok` can be pre-decoded in this way. Fields are arranged in `LSE_emu_predecode_info_t` in the order in which they are listed in the `predecodefields` attribute.

**Instruction steps**

The emulator must divide instruction execution into at least two steps. Each step must be given a name and a non-negative integer step number. Two steps may have the same step number if they are simply
aliases of each other. Step numbers should start with 0. They must be assigned so that execution of all step numbers from 0 to the maximum step number, inclusive, will result in complete, correct execution of the instruction.

Each step number must be assigned to one of three groups of steps: "front end", "back end", and "commit". These correspond roughly to "fetch and decode", "operand fetch, execute, and writeback", and "commit". The exact boundaries are up to the emulator, but the assignment must be such that executing the three groups in "front", "back", "commit" sequence does not violate the correct execution order of the steps.

The steps must be described in the step_names attribute of the description file. This attribute is a list of tuples of three elements of the form (name, step number, group). The encoding of groups is 0 for "front", 1 for "back", and 2 for "commit".

A potential division of and description of steps is:

```python
step_names = [
    ("fetch", 0, 1),
    ("decode", 1, 1),
    ("opfetch", 2, 2),
    ("alu", 3, 2),
    ("memread", 4, 2),
    ("longalu", 4, 2),
    ("writeback", 5, 2),
    ("memwrite", 6, 3),
    ("commit", 7, 3),
]
```

The last step may release memory allocated by the emulator for the instruction for private or extra fields, but the emulator must document which fields thus become invalid.

It is not necessary to assign any steps of execution to the "commit" group if the method the emulator uses to deal with mis-speculation recovery (if any) does not require notification about commits.

As steps are executed, if any data dependencies between steps (or between operand fetches and steps when the operandval capability is present) are violated, the emulator behavior is undefined; it may perform missing steps, report an error, compute incorrect results, or crash. We recommend that a debug mode be implemented which tests for the violation of data dependencies and reports an error and terminates simulation in such cases.

If a particular step number does not apply to an instruction, the emulator should simply do nothing; it should not report this to be an error.

### Exiting and signal handlers

The emulator must not register signal handlers to catch error conditions unless it is going to catch and continue after these errors when an instruction is speculative (which in general it does not know).

**TO DO**

Say something intelligent about host machine exceptions.
Every context must have an EXIT instruction which jumps to itself and implements the context exit semantics (as described in the next paragraph). This EXIT instruction must act like a system call and be a jump to itself. The return address (if there is such a concept in the ISA's Application Binary Interface) must be set to point to the EXIT instruction at context creation or loading.

When a context exits in the emulator, the emulator does not exit the simulation by calling `exit(3)` (or its relatives) or `longjmp(3)`. Instead, it context switches out the software context (even if the context is not subject to automapping). If no new context can be switched in or the hardware context is not on the list of automatically mapped contexts, the hardware context is mapped to "no context" (context number equals 0).

**Error reporting**

The emulator should report errors it encounters using writes to `LSE_stderr`. A pointer to this variable is given in the `LSE_emu_interface_t` structure passed to `EMU_init`. The redirection of `LSE_stderr` to specific files is the responsibility of the command-line processor and/or scripts; the emulator must not do this.

**Extra functions**

The emulator can declare extra functions to be available to the simulator. These functions can provide extra capabilities which do not fit within a standard capability definition. For example, the `BLiSSAlpha` emulator provides a function to check whether an address falls within the text segment of a program. All such functions must be declared in the `extrafuncs` attribute. This attribute is a list of tuples. Tuples are formed by using parenthesis and commas, with elements: `(return_type, function_name, parameter_list)`. An example of an `extrafuncs` attribute with two functions is:

```plaintext
extrafuncs = [  
    ("boolean", "EMUEXT_is_in_range", "LSE_emu_addr_t") ,  
    ("int", "EMUEXT_print_product", "int a, int b")  
]
```

The functions may have any name, but for consistency with other API function names we recommend beginning them with "EMUEXT_". The return type and parameters must be either a well-known C type, a glib type, or one of the types made available by LSE to the emulator.

**Header files**

A list of header files to include in simulators using this emulator is provided by the `headers` attribute. This attribute can only contain header file names.

Some header files may require include paths to be added to the compilation command line. Specify the additional compiler flags using the `compileFlags` attribute. This text will be all passed literally to the compiler command line (in constrast to the text passed to the linker as described below).

41
Library names

A list of libraries to link with is provided by the libraries attribute. This attribute can contain linker options, linker search paths (-L), libraries to be searched (-l), and text to be passed literally to the linker. Each word of literal text must begin with a # character. An example of a libraries attribute with back-tick execution of a command (done using literal text) is:

```
libraries = "mylib.a #'glib-config #--libs"
```

In this example, the command glib-config --libs would be run by the shell performing the link.

Note that it is not possible to pass specific whitespace characters onto the linker command line; the libraries attribute is broken into words at whitespace boundaries and is then processed word-by-word.

State-space capability definitions

Capabilities are listed here in alphabetical order.

General capability definitions

Capabilities are listed here in alphabetical order.

The branchinfo capability

The branchinfo capability indicates that the emulator calculates inline addresses, branch targets, and branch direction and store them in standard locations in interface structures. The step at which the emulator calculates these fields is left to the emulator and may vary for different types of branches. In particular, direct and indirect branches are likely to compute targets at different steps while branch direction and target are also likely to be computed at different steps. The emulator should document the step at which different elements of branch information become available.

When the branchinfo capability is present, the description file must contain an attribute named max_branch_targets. This attribute indicates the maximum number of potential "next" instructions after any instruction. The number includes the "inline" instruction, so this attribute must always be greater than 1. The attribute appears in header files as a constant LSE_emu_max_branch_targets.

**Note:** The inline instruction is always target number 0. Unconditional branches must still treat the "inline" instruction as target number 0; their "unconditionality" is reflected by always setting branch_dir to a value greater than zero.

The following fields are added to LSE_emu_instr_info_t:

- int branch_dir; - which potential next instruction is to be executed; 0 indicates the inline instruction.
The **commandline** capability

The commandline capability indicates that the emulator provides functions to parse command-line arguments and print out a portion of a usage message. The functions are:

```c
int EMU_parse_arg(int argc, char *arg, char *argv[]);
```

Parse a single command-line argument `arg`, which may have additional following arguments in `argv`. `argc` is the length of `argv` plus 1 (for `arg`). Must return the number of arguments used, including `arg`; 0 for an error. Error messages should be printed to `LSE_stderr` or `stderr(3)`.

```c
void EMU_print_usage(void);
```

Print usage for the emulator to `LSE_stderr` or `stderr(3)`.

The **disassemble** capability

The disassemble capability provides a function that the simulator can call to get the disassembly of an instruction. The function is given an address to fetch and disassemble, but when the splitfront capability is present, there must also be a function which disassembles from a given instruction word.

The functions the emulator must provide are:

```c
void EMU_disassemble_addr(LSE_emu_ctoken_t ctoken, LSE_emu_addr_t addr, FILE *outfile);
```

Fetch and disassemble the instruction at `addr` in context `ctoken`, outputting the text to `outfile`.

```c
void EMU_disassemble_instr(LSE_emu_instr_info_t *ii, FILE *outfile);
```

Disassemble instruction `ii`, outputting the text to `outfile`.

The **operandinfo** capability

The operandinfo capability indicates that the emulator will provide information about what state is used or modified as source and destination operands of each instruction. The emulator must do this by filling in proper fields in the interface structures during the decode operation. Operands report their state references as addresses within state spaces of the emulator and may provide bit-level access information.
Chapter 5. LSE/Emulator API details

There are two primary purposes for the the operand information. The first is to allow the microarchitectural model to discover register-carried data dependencies. The second is to provide the ability to manipulate operand values at different times when the operandval capability is also present. To meet these purposes properly, emulators should represent all register operands in the operand information. Immediate source operands may be included as well; this is particularly appropriate when the operandval capability is also present, as it will allow microarchitectural models to access the immediate value. Note that immediate destination operands are possible; these are often used to indicate state updates that are not normal registers (e.g. memory) and imply that "normal" register-carried data dependency checking should not happen on them.

**Note:** Reported operands should include registers which are implicitly used as well as the more obvious ones encoded explicitly into the instruction. A common example of an implicit register is a carry flag.

Operand information is placed into an array of information structures. The location of a particular operation in the array can be used to denote the purpose of the operand. To do this, the emulator defines a set of "names" which map to offsets in the array. For example, a simple DLX-style architecture might define names "Left" and "Right" with values 0 and 1 for the name mappings. All "left" operands would go into the 0th element of the information array while all "right" operands would go into the 1st element of the array. An emulator is not required to provide a set of names (it can be left empty), nor is it required (though it is very strongly encouraged) to make them particularly useful. There are emphasis no standard names which must be supported.

When this capability is present, the description file must contain four attributes. The first three are named max_operand_src, max_operand_dest, and max_operand_int, which indicate the number of source, destination, and intermediate operands, respectively. These attributes’ values appear in header files as constants LSE_emu_max_operand_src, LSE_emu_max_operand_dest, and LSE_emu_max_operand_int. The final attribute is operand_names, which is a list of (name, value) tuples, e.g:

```plaintext
operand_names = [ ("Left", 0), ("Right", 1) ]
```

Two types become available with this capability. The first type, LSE_emu_operand_name_t is an enumerated type with the values being the operand names defined in the operand_names attribute. Individual names have the form: LSE_emu_operand_name_name. The other type, LSE_emu_operand_info_t, is a structure with fields:

- LSE_emu_spaceaddr_t spaceaddr; - The address of the register within its state space.
- LSE_emu_spaceid_t spaceid; - The state space of the register.
- union { ... } uses; - provides information about how the operand is used. The exact structure is:

```c
union {
    struct {
        uint64_t bits[];
    } reg;
    struct {
        unsigned int size;
```
uses.reg.bits contains the bits used in the register access; bit number x’s flag is
uses.reg.bits[x/64] & (1LL<<(x%64)). A set bit indicates that the corresponding bit is
accessed. This field is valid only for register state spaces.

uses.mem.size and uses.mem.flags contain the size of the access (in bytes) and flags
indicating things such as direction (read vs. write), atomicity, and ordering. These fields are valid only
for memory state spaces. There are standard flag values (LSE_emu_memaccess_*) for common
information, but emulators may use additional values.

The following fields are added to LSE_emu_instr_info_t:

• LSE_emu_operand_info_t operand_dest[LSE_emu_max_operand_dest]; - information
  about destination operands.
• LSE_emu_operand_info_t operand_src[LSE_emu_max_operand_src]; - information about
  source operands.

Not all instructions will require all of the operands; some instructions may use immediates instead of
registers for some operands. These cases can be encoded in the operand information. An unused operand
has a spaceid which is zero and a spaceaddr.LSE which is zero. An immediate operand has a
spaceid which is zero and a spaceaddr.LSE which is not zero. The uses field is undefined in
these cases.

Note: Remember that operand information is only information about what state is accessed by the
operands. The values of the operands (particularly immediates) are not carried in the operand
information structure.

Two additional functions must be supplied by the emulator when this capability is present:

boolean EMU_spaceaddr_is_constant(LSE_emu_ctoken_t ctoken,
LSE_emu_spaceid_t sid, LSE_emu_spaceaddr_t *addr);

Return TRUE if address addr in state space sid in context ctoken is a constant, FALSE
otherwise.

int EMU_spaceaddr_to_int(LSE_emu_ctoken_t ctoken, LSE_emu_spaceid_t
sid, LSE_emu_spaceaddr_t *addr);

Return a translation of addr in state space sid in context ctoken into an integer. The integer
may not equal or exceed the number of elements in the state space. This function will not be called
until after a program is loaded into the context and is only called for state spaces which are defined
with string addresses. This function is not required if no state spaces have string addresses.
Chapter 5. LSE/Emulator API details

The **operandval** capability

The *operandval* capability indicates that the emulator makes operand values available in the instruction information structure as they are fetched or computed and uses the values stored in the structures at later steps. This makes it possible for microarchitectural models to override operand values. It also allows operands to be individually fetched and written back. The *operandval* capability requires the *operandinfo* capability.

When the *operandval* capability is present, the description file must contain an attribute named *operandvaltype* which describes the type of operand values. This is usually a union type.

The following fields are added to *LSE_emu_instr_info_t*:

- *LSE_emu_operand_val_t operand_dest[LSE_emu_max_operand_dest]*; - destination operand values.
- *LSE_emu_operand_val_t operand_int[LSE_emu_max_operand_int]*; - intermediate operand values. Only appears if *LSE_emu_max_operand_int* is greater than 0.
- *LSE_emu_operand_val_t operand_src[LSE_emu_max_operand_src]*; - source operand values.
- *boolean operand_written_dest[LSE_emu_max_operand_dest]*; - flags indicating whether each destination operand has been written back. These flags should be cleared in *EMU_init_instr*.

It is not required to make all operands available, though we strongly encourage you to do so. It is also desirable to make certain that no operand is both written and read in the same step of execution to ensure that modifications to the operand can have an effect.

When the *operandval* capability is present, the emulator must also provide two functions:

```c
void EMU_fetch_operand(LSE_emu_instr_info_t *ii, LSE_emu_operand_name_t oname);

Fetch (read the state for) the source operand named *oname* for instruction *ii*. The value must be placed in the *operand_val_src[oname].data* field. The *valid* flag must be set to TRUE.
```

```c
void EMU_writeback_operand(LSE_emu_instr_info_t *ii, LSE_emu_operand_name_t oname);

Write back (set the state for) the destination, intermediate, or memory destination operand named *oname* for instruction *ii*. The value is taken from the *operand_val_dest[oname].data* field. The field *operand_written_dest[oname]* must be set to TRUE. Other operand values may be used to determine the state to be updated (e.g., an effective address, or a rotating register base).
```

The **speculation** capability

The *speculation* capability indicates that the emulator supports mis-speculation recovery by providing a way to "undo" the effects of emulation. It is not necessary to be able to undo the effects for all
instructions, but any instruction which has some state change which cannot be undone must be be classified as a side-effecting instruction.

Some side effects cannot be known at the time that instructions are normally classified (normally a "decode" step). An example would be a state space which can emulate a hardware device. Because the presence of side effects can depend upon the effective address, the instruction cannot be classified as side-effecting during decode. In such situations, it will be up to the configurer to ensure that the instruction does not change the same state multiple times. It might be helpful in some cases if the emulator were to provide an extra function indicating whether an effective address has a side effect.

When the *speculation* capability is present, the emulator must supply one additional API function:

```c
void EMU_rollback(LSE_emu_instr_info_t *ii);
Roll back any state modified by the instruction given by *ii*.
```

**Warning**

Be very careful when implementing this capability as writeback steps could be repeated and cannot be bounded a priori. This is particularly an issue when the *operandval* capability is also present. The *operand_written_dest* field cannot be used as a flag indicating that the old state value has already been saved because the microarchitectural model may clear this flag to indicate to itself that a value needs to be written back again.

**Open Issue**

How do we deal with writes to external spaces? Does the emulator add the recovery information or does the simulator? Rollback and cross-instruction semantics... how do they interoperate?

The **threadsafe** capability

The *threadsafe* capability indicates that calls to all APIs provided by the emulator are thread-safe. This capability is not used at present, but will be used when multi-threaded simulation is implemented; a thread-safe emulator would not need to have a lock in LSE to control access to its API.

There are no additional datatypes or functions defined by this capability.

**Open Issue**

What about use of LSE_error? How does this affect thread-safe properties?
Capabilities to be implemented

This information will be expanded and moved into the above main section as each capability is implemented.

State-space capabilities

The *extaccess* capability

The *extaccess* capability indicates that the emulator provides APIs which allow this state-space to be written to and read from.

The *externalize* capability

The *externalize* capability indicates that the emulator will treat the state space as an external state space instead of an internal state space for all functionality when LSE indicates that it must do so. This capability is used by LSE to determine which emulator allocates a particular state space instance.

This capability can also be used to implement automatic remapping of state space accesses. Suppose that a general register state space has the externalize capability. The configuration can define a physical register state space and a translation state space which translates from general register numbers to register numbers in the physical register state space. If the configuration externalizes the general register state space instance and maps it to the translation state space, then all general register accesses will be directed to the physical register file.

An attribute will imply whether internal or not by default. Obviously, if this capability is not present, will be internal.

The *multiinst* capability

The *multiinst* capability indicates that the state space is not shared among all contexts.

The *shareable* capability

The *shareable* capability indicates that the state space can be shared between multiple contexts.

Information capabilities

The *itokenrefs* capability

The *itokenrefs* capability indicates that both LSE and the emulator maintain dynamic instruction instance information. The emulator provides an API for LSE to call when LSE is about to deallocate the dynamic instruction. The emulator must not deallocate the instruction before this time (or must be able to deal with a repeated call for an already-deallocated instruction).
Instruction flow capabilities

The needtime capability

The needtime capability indicates that LSE must call an emulator API whenever "time" advances. "Time" is an abstract notion; it may refer to clock cycles or instructions or instruction groups. The configurer must ensure that this API gets called at the appropriate occasions.

Miscellaneous capabilities

The except capability

The except capability indicates that the emulator may report exceptions and have exceptions reported to it.

Documenting the emulator
Chapter 5. LSE/Emulator API details

Open Issue

What must be documented?

• must state capabilities
• must state when base interface doesn’t work
• What fields become valid after each step of execution, what fields are required to be valid, and where the frontend/backend break is
• When each field must be updated to actually change emulator behavior.
• Any limitations on rollback/commit. Whether commit is needed.
• must state what operands are not identified or what immediates are in operandinfo.
• State when earlybranchinfo doesn’t supply all normal info.
• When detailed evaluation cross-pollinates branch and mem access structures.
• How evaluation of steps deals with missing data
• Ordering requirements of operand fetch
• Are commits required in order?
• Any fields deallocated by commit
• Any steps which are required of all or particular classes of instructions
• What happens to starting address on context switch
• Any emulator discontinuities
• Is memory access information stored in operand info structures? Where? Any additional flags? What step?

Stuff to incorporate into text

Open Issue

Should the emulator declare that base doesn’t work so that we can check for its use?
When do we need to have the syscall stuff valid?
Tables giving the order in which things appear within the static and dynamic info structures.

------------------------- API definitions ---------------------------

--- "itokenrefs" capability ---

This capability indicates that instruction tokens are references to some allocated instruction information which must be properly freed. The token is allocated when the instruction is decoded. It is
deallocated when the appropriate function is called.

void EMU_release_itoken(<itokentype> t)

    Release the token.

===========================================================================

Unresolved details
==================

- Emulators needing "instruction instances" to operate need to have a clear
  start and finish point for the instructions... somehow needs to be linked
  with dynids.

- Reference counting for contexts: how do we know when a context is safe
  to remove entirely? Need some sort of interface....

State update rules

Open Issue

Which Simulator-supplied APIs update internal state and which cannot be rolled back?

Emulator-supplied APIs allowed to update internal state

hi bye

Emulator-supplied APIs not allowed to update internal state

LSE_emu_emulator_decode bye