

Liberty Simulation Environment Core Module Library Reference Manual

The Liberty Research Group

Liberty Simulation Environment Core Module Library Reference Manual
by The Liberty Research Group
Version 1.0p1 Edition

Table of Contents

| | |
|--|----------|
| Preface | i |
| Typographical conventions used in this book | i |
| 1. Overview | 1 |
| 2. Conventions for Modules Described in this Chapter..... | 2 |
| The Module That is Described in This Section..... | 2 |
| Short Description..... | 2 |
| Detailed Description..... | 2 |
| Types..... | 2 |
| Parameters | 2 |
| Functions | 2 |
| Additional Conventions..... | 2 |
| 3. Modules..... | 3 |
| The aligner Module | 3 |
| Short Description..... | 3 |
| Detailed Description..... | 3 |
| Types..... | 3 |
| Parameters | 3 |
| Events | 3 |
| Ports..... | 3 |
| Functions | 4 |
| The arbiter module | 4 |
| Short Description..... | 4 |
| Detailed Description..... | 4 |
| Types..... | 4 |
| Parameters | 5 |
| Events | 5 |
| Ports..... | 5 |
| Functions | 5 |
| The combiner Module..... | 6 |
| Short Description..... | 6 |
| Detailed Description..... | 6 |
| Types..... | 6 |
| Parameters | 6 |
| Events | 8 |
| Ports..... | 8 |
| Functions | 8 |
| The converter Module..... | 8 |
| Short Description..... | 8 |
| Detailed Description..... | 8 |
| Types..... | 9 |
| Parameters | 9 |
| Events | 9 |
| Ports..... | 9 |
| Functions | 9 |

| | |
|--------------------------------|----|
| The delay Module | 9 |
| Short Description | 10 |
| Detailed Description | 10 |
| Types | 10 |
| Parameters | 10 |
| Events | 11 |
| Ports | 11 |
| Functions | 12 |
| The demux Module | 12 |
| Short Description | 12 |
| Detailed Description | 12 |
| Types | 12 |
| Parameters | 12 |
| Events | 13 |
| Ports | 13 |
| Functions | 13 |
| The gate Module | 13 |
| Short Description | 13 |
| Detailed Description | 13 |
| Types | 14 |
| Parameters | 14 |
| Events | 15 |
| Ports | 15 |
| Functions | 16 |
| The mqueue Module | 16 |
| Short description | 16 |
| Detailed description | 16 |
| Types | 16 |
| Parameters | 17 |
| Events | 18 |
| Ports | 18 |
| Functions | 18 |
| The pipe Module | 19 |
| Short description | 19 |
| Detailed description | 19 |
| Types | 20 |
| Parameters | 20 |
| Events | 21 |
| Ports | 21 |
| Functions | 21 |
| The router Module | 21 |
| Short Description | 21 |
| Detailed Description | 21 |
| Types | 22 |
| Parameters | 22 |
| Events | 23 |
| Ports | 23 |
| Functions | 23 |

| | |
|------------------------------------|----|
| The selector Module | 23 |
| Short Description | 23 |
| Detailed Description | 23 |
| Types | 24 |
| Parameters | 24 |
| Events | 24 |
| Ports | 24 |
| Functions | 24 |
| The serializer Module | 24 |
| Short Description | 25 |
| Detailed Description | 25 |
| Types | 25 |
| Parameters | 25 |
| Events | 26 |
| Ports | 26 |
| Functions | 26 |
| The sink Module | 26 |
| Short Description | 26 |
| Detailed Description | 26 |
| Types | 26 |
| Parameters | 26 |
| Events | 27 |
| Ports | 27 |
| Functions | 27 |
| The source Module | 27 |
| Short Description | 27 |
| Detailed Description | 27 |
| Types | 28 |
| Parameters | 28 |
| Events | 28 |
| Ports | 28 |
| Functions | 28 |
| The tee Module | 29 |
| Short Description | 29 |
| Detailed Description | 29 |
| Types | 29 |
| Parameters | 29 |
| Events | 29 |
| Ports | 30 |
| Functions | 30 |
| The wire Module..... | 31 |
| Short description..... | 31 |
| Detailed Description | 31 |
| Types..... | 31 |
| Parameters | 31 |
| Events | 31 |
| Ports | 31 |
| Functions | 32 |

Preface

This book describes the core Liberty Simulation Environment library of modules. To use these modules, please import the package **corelib** using either the `using` or `import` keyword.

Typographical conventions used in this book

The following typefaces are used in this book:

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

Chapter 1. Overview

This book describes the modules contained in the Liberty Simulation Environment Core Module Library. These basic modules are can be used for fanout of data, data routing and arbitration, queuing structures, unit depth delay or pipelines, variable depth pipelines, ALU like structures, and many other structures. These modules are contained in the **corelib** package.

Chapter 2. Conventions for Modules Described in this Chapter

The Module That is Described in This Section

Short Description

A short description of the module, usually one line.

Detailed Description

A more detailed description of the module.

Types

The types defined by the module. Naming is relative to the corelib package. So a type, **foo::bar** has fully qualified name **corelib.foo::bar**.

Parameters

Parameters are listed in this section. User points are written using C function prototype notation (although the declaration syntax in LSS is different).

Functions

In this section any LSS functions that can be used to customize the instance of a module is listed and described. Naming is relative to the corelib package. So a function, **foo::bar** has fully qualified name **corelib.foo::bar**.

Additional Conventions

If **foo** is a port name, then **foo[i]** refers to the *i*th port instance of the **foo** port. **foo[i].data**, **foo[i].enable**, and **foo[i].ack** refer to the data, enable, and ack signals, respectively, of **foo[i]**. **foo[i].data_value** refers to the actual data value, and **foo[i].data_id** refers to the dynamic identifier that is part of the data signal.

Chapter 3. Modules

Below are the descriptions for the modules in the **corelib** package. The conventions used to describe modules in this chapter are described above.

Note: For the modules in this chapter, whenever possible, unless otherwise noted, the module instances will treat the dynamic identifier and data values as a unit. Thus, when the documentation says that that data is routed from port **in** to **out**, the id is routed as well.

The aligner Module

Short Description

The **aligner** is a special case of an arbiter.

Detailed Description

The aligner module implements a special case of the arbiter that has been optimized for simply "squashing" out data signals of value nothing. An aligner will output on its out port instances the first out.width data values on the **in** port instances that are `LSE_signal_something`. The output enable signal is set based on the input enable signal that was routed to the output. The ack works in a similar fashion, but in the reverse direction.

Types

none

Parameters

none

Events

none

Ports

in:'a

The input port.

out:'a

The output port with aligned data.

Functions

none

The arbiter module

Short Description

Instances of this module perform many to many arbitration via a sorting algorithm.

Detailed Description

Instances of this module arbitrate among the port instances of the **in** port and send the data corresponding to the winners of the arbitration to the out port instances.

The arbitration occurs via a sorting algorithm. The instance will sort the data on the instance of the **in** port according to a user point defined comparison function. The first **out**.width elements of the sorted list are considered the winners of the arbitration and the out port instances get the corresponding dynamic ids and data. **out[0]** will get the first element in the sorted list, **out[1]** the second, and so on.

The **arbiter** will also output on its **in_map** output port the indexes of the winners of the arbitration. Each instance of the **in_map** port corresponds to identically numbered instance of the **out** port. if **out[i]** is outputting the data on **in[j]** then **in_map[i]** gets the value j. **in_map** can be connected to the route_info ports of a router instance to allow the **arbiter** to control the routing of 2 or more distinct signals.

The output enable signal is the same as the input enable signal of the data that was routed to the output. The ack is the same, but in reverse. The **in_map** port ignores acks and outputs sets things enabled.

The in and out port types must match. **in_map** has type **int**.

Note: The **arbiter** module is conservative in its routing of enable and acks. That means that all routing decisions must be made before enable and ack are propagated.

Types

none

Parameters

```
boolean comparison_func(LSE_signal_t status1, LSE_dynid_t id1,
LSE_port_type(out) *data1, int port1, LSE_signal_t status2,
LSE_dynid_t id2, LSE_port_type(out) *data2, int port2);
```

Function used to compare 2 elements to see which has higher priority for arbitration. Returns whether item 2 should be closer to the front than item 1. In other words, return >0 if there is an inversion, 0 if there is not, and -1 if the answer is not yet known. There is no default value for this parameter, it must be set by the instantiator.

Events

none

Ports

in:'a

The input port.

out:'a

The output port which outputs selected winners.

Functions

```
arbiter::round_robin_funcheader(state:literal) => string
```

Function that returns a value for an **arbiter**'s funcheader parameter to implement round robin arbitration. state is a **literal** that is the name of a variable that will be used at runtime to store the state of the **arbiter**. You must use the round_robin_end_of_timestep function to create the code that will update the state.

```
arbiter::round_robin_end_of_timestep(state:literal) => string
```

Function that returns a value for an **arbiter**'s `end_of_timestep` parameter to implement round robin arbitration. `state` is a **literal** that is the name of a variable that will be used at runtime to store the state of the **arbiter**. You must use the `round_robin_funcheader` function to create the state.

```
arbiter::round_robin_comparison_func(state:literal) => string
```

Function that returns a value for the `comparison_func` user point that will perform round robin arbitration. `state` contains a variable name that will be used to store the state of the round robin **arbiter**. You must use `round_robin_funcheader` and `round_robin_end_of_timestep` to create and maintain the state.

The combiner Module

Short Description

A very flexible module whose instances can be parameterized to combine multiple inputs into multiple outputs based on a number of user points and parameters.

Detailed Description

Instances of this module have input ports with the names given in elements of the `inputs` parameter. Outputs are determined by the `outputs` parameter. By default, the `combine` user point is called for every port instance, `i`, to determine how to set the outputs. The time when the user point is called, as well as the default behavior when it is not called is subject to the setting of several parameters. See the `parameters` section below.

Behavior of this module is purely sliced, that is, input port instances number `i` can only affect output port instances numbered `i`. Thus, the input port widths must match the output port widths.

This is a very general module and can be used to synchronize inputs, behave as an ALU, etc. Note, however that the input and output ports have no type constraints. Please wrap this module in another module that enforces the type constraints you desire.

Types

none

Parameters

```
void combine(variable);
```

This user point determines how to compute the outputs from the inputs.

The user function has a set of arguments, *portname_data*, *portname_status*, and *portname_id* for every input and output port, and an argument, *porti* that determines the slice being worked on. For input ports, the *portname_status* variable gives the input status of *portname*[*porti*], *portname_id* is the dynamic identifier received, and *portname_data* is a pointer to the data.

For output ports, *portname_status* is a pointer to the output status which contains the current output ack status and must be set with the proper data and enable status (i.e. *LSE_signal_something*, *LSE_signal_enabled*, etc.). *portname_id* is a pointer to a **LSE_dynid_t** which should be set to the output dynamic identifier. *portname_data* is a pointer to a cell in which the output data should be stored.

***wait_for_data*:bool**

Determines if all data signals on input port instances numbered *porti* must be something other than *LSE_signal_unknown* before the combine user point is called.

When **TRUE**, the combine function is only called when all the inputs in slice *porti* have data signal values that are known, otherwise all outputs on slice *porti* get *LSE_signal_nothing*, *LSE_signal_disabled*.

The default value for this parameter is **TRUE**.

***require_all_data_present*:bool**

When **TRUE**, the combine function is only called when all the inputs in slice *porti* have data signal values that are *LSE_signal_something*. If some input has data signal *LSE_signal_nothing* and another input has data signal *LSE_signal_something* an error message will be displayed and simulation will abort.

The default value is **FALSE**

***propagate_nothing*:bool**

When **TRUE**, send nothing on the output if any input has nothing.

The default value is **TRUE**

***NACK_nothing*:bool**

Determines whether nothings on the inputs are sent *LSE_signal_ack* or *LSE_signal_nack*.

The default value is **FALSE**.

***combine_calculates_enable*:bool**

When `FALSE`, the output enables for output port instances numbered `porti` are computing by anding the enable signals of input port instances numbered `porti`.

When `TRUE`, the combine function **MUST** explicitly set the enable signal if it is called.

The default value is `FALSE`

***combine_uses_ack*:bool**

When `FALSE`, the output acknowledge signals are not supplied to the combine user point.

When `TRUE`, the output acknowledge signals are supplied to the combine user point.

The default value is `FALSE`.

Events

none

Ports

Parameter dependent. See above.

Functions

none

The converter Module

Short Description

This module takes data, modifies in according to a user point then outputs it in the same cycle.

Detailed Description

This module takes any data received on port `in[i]` and calls the `convert_func` user point to modify the data. The `convert_func` returns the modified data and the data is output on port `out[i]`. `in[i].enable` is passed through to `out[i].enable`. `out[i].ack` is passed through to `in[i].ack`

The types of the **in** and **out** port may be different and thus this module can perform type conversion. **in.width** and **out.width** must match.

Types

none

Parameters

```
LSE_port_type(out) convert_func(LSE_dynid_t id, LSE_dynid_t *newidp,  
LSE_port_type(in) data, int porti);
```

This user point determines how the incoming data is modified. The port on which the data to be converted was received is stored in `porti`. The dynamic identifier received is in `id` and the data is in `data`. The return value of this function is the new data. The dynamic identifier received on the input port is normally placed on the output port. However, the user point can indicate a different dynamic identifier by changing the value pointed to by `newidp`.

Events

none

Ports

in:'a

The input port.

out:'b

The output port with converted data.

Functions

none

The delay Module

Short Description

This module delays its input by at least one cycle.

Detailed Description

Instance of this module takes enabled data on an input port instance, **in[i]**, and store the data, outputting it each cycle thereafter on port instance **out[i]**, until an ack is received on **out[i]**. The enable signal on port **out[i]** is equal to the ack signal in that cycle if *pass_acks_to_enable* is `TRUE`; otherwise it is equal to the data signal.

If the *pass_acks_when_full* parameter is `TRUE` (the default), then **in[i]** will get the value of the ack signal on **out[i]**. Otherwise, **in[i]** is acked when the delay element is not holding any data to be output on port **out[i]**.

The delay element starts out storing no data unless there is an *initial_state* user point definition override. See below for details.

The input and output ports must have the same type. The input and output ports must have the same width.

Types

none

Parameters

pass_acks_when_full:**bool**

Determines when space is freed in the delay element.

The default value is `TRUE`

pass_acks_to_enable:**bool**

When `TRUE`, the enable signal on an instance of the **out** port is driven to match the ack signal received on that port instance. When `FALSE`, the enable signal is driven to match the data signal on that port instance.

The default value is `TRUE`

```
boolean drop_func(int porti, LSE_dynid_t id, LSE_port_type(out)
*data);
```

This user point is used to drop elements from the delay element at the end of a timestep. The default value is

```
<<<return FALSE>>>
```

```
boolean initial_state(int porti, LSE_dynid_t *init_id,
LSE_port_type(out) *init_value);
```

This user point allows setting of the initial state of the delay element. This function should create a dynamic identifier and output data for the initial value of the delay element (for output on port instance **out[*porti*]**) and return a pointer to the id in *init_id* and store the initial value in the location pointed to by *init_value*. If the function returns **FALSE**, the delay element slice *porti* starts out empty. If it returns **TRUE**, the values in *init_value* and *init_id* are stored in the delay element and output on port **out[*porti*]**.

The default value is:

```
<<< return FALSE; >>>
```

Events

STORED_DATA

This event passes the following data to a collectors record code every time a item of data is stored in the delay element:

```
porti : <<<int>>>
id : <<<int>>>
datap : <<<LSE_port_type(in) *>>>
```

porti is the port instance on which the stored data was received, *id* is the dynamic id stored, and *datap* is a pointer to the data stored. It is illegal to modify the data pointed to by *datap*.

Ports

in:'a

The input port.

out:'a

The output port.

Functions

```
delay::init(value:literal) => string
```

This function returns a user point that will set the initial value of the delay element for all port instances to the value in the value argument. The **literal** in the value argument is treated as a literal of the type of the input and output ports.

The demux Module

Short Description

Instances of this module take data from an input port instance and send it to one of several output port instances based on a of a user point.

Detailed Description

An instance of this module forwards data from port instance **in[i]** to output port instance **out[(n/m)*i+k]** where k is the return value of the *choose_logic* user point, n is **out.width**, and m is **in.width**. k must be between 0 and (n/m)-1.

If the *choose_logic* user point returns -1 the selection is deferred. Any rescheduling of the module must be explicitly in the user point code, either through use of a query or an explicit scheduler API call.

The ack signal on port **out[(n/m)*i+k]** is routed to **in[i]** where the variables have the same definition as above.

The input and output port must have the same type. **out.width/in.width** must be an integer.

Types

none

Parameters

```
int choose_logic(int porti, LSE_dynid_t id, LSE_port_type(in) *data);
```

User point to determining how to route the input

Events

none

Ports

in:'a

The input port.

out:'a

The output port.

Functions

none

The gate Module

Short Description

The gate module provides a way to filter/block out signals based on a user point and a polymorphic control port. The user may independently define whether data, enable, and ack get filtered/blocked.

Detailed Description

The gate module provides a way to filter/block out signals based on a user point and a polymorphic control port. The user may independently define whether data, enable, and ack get filtered/blocked.

The gate has 3 ports, an **in** input port, an **out** output port and a **control** input port. For each **in/out** slice, the gate will call the *gate_control* user point. The return value of the user point determines whether the data, enable, and ack signals on this slice will pass unmodified through the gate or if the signals will be blocked/filtered.

The *gate_control* user point has passed, as a parameter, an array of the control port statuses and data values (which may potentially be unknown). Based on the particular input port instance status and data value and the control port data and statuses, the code returns a value indicating what the gate should do.

If the user point returns 1 then the data, enable and ack are passed through. If the code returns 0 then the output data and enable signal are set to `LSE_signal_nothing` and `LSE_signal_disabled` respectively. The value of the ack signal is discussed below. If the code returns -1 the decision is delayed until additional control signals resolve.

If the gate is configured as a blocker, the `gate_style` parameter is set to `gate::blocker`, then the gate will block, that is, send a `LSE_signal_nack`, when the `gate_control` function returns 0. Otherwise, when the `gate_style` parameter is set to `gate::filter`, the gate will return `LSE_signal_ack` when the `gate_control` function returns 0.

Users can control which signals are controlled by the `gate_control` function via the `gate_ack`, `gate_enable`, and `gate_data` parameters. If any of these parameters is false, the gate simply passes the corresponding signal straight through the module from `in[i]` to `out[i]` (or the reverse direction for the ack signal).

If the parameter `wait_for_data` is set to `TRUE` then the `gate_control` user function is only called when the data signal on `in[i]` is `LSE_signal_something`. If the signal resolves to `LSE_signal_nothing`, then the data, enable, and ack signals just pass through the gate. If the parameter `wait_for_data` is set to `FALSE`, then the `gate_control` user function is called without waiting for the data signal on `in[i]`. This means that the `gate_control` user function can be called when the data signal has value `LSE_signal_unknown`.

The control port can have any type. The `in` and `out` ports must have the same type.

Types

```
gate::gate_style_t:enum {filter, blocker}
```

Type to enumerate `gate_style` values

Parameters

`gate_data:bool`

This parameter determines whether the `gate_control` function blocks/filters the data signal.

The default value is `TRUE`

`gate_enable:bool`

This parameter determines whether the `gate_control` function blocks/filters the enable signal.

The default value is `TRUE`

gate_ack:bool

This parameter determines whether the `gate_control` function blocks/filters the ack signal.

The default value is `TRUE`

wait_for_data:bool

This parameter determines if the `gate_control` function is called only when the `in[i]` has data on it.

The default value is `TRUE`

```
int gate_control(int porti, LSE_signal_t status, LSE_dynid_t id,
LSE_port_type(in) *data, LSE_signal_t *cstatus, LSE_dynid_t *cid,
LSE_port_type(control) **cdata, int cwidth);
```

User point that determines if data is passed or not.

The default value is

```
<<<return 1;>>>
```

```
void control_handler(int porti, LSE_signal_t cstatus, LSE_dynid_t
cid, LSE_port_type(control) *cdata);
```

Utility function that is called every time the control port status changes.

Events

none

Ports

in:'a

The input port.

out:'a

The output port.

control:*

The control port whose values can be used to determine if data should pass or be blocked/filtered.

Functions

none

The mqueue Module**Short description**

Instances of this module model multiple input multiple output queues.

Detailed description

Instances of the mqueue module are queues which can add more than one item to the tail and remove more than one item from the head in a single time step. Items must be in the queue for at least one time step. They do *not* need to be removed in order, though this can be controlled via a parameter. The maximum size of the queue is defined by the *size* parameter.

Inputs on the **in** port instances are added onto the tail queue (with priority given to lower port instances) if **in[i].enable** == LSE_signal_enabled. Values presented on the **out[i]** port instance is removed from the head of the queue if the **out[i].ack** == LSE_signal_ack.

The **in** port instances are not arbitrated; what this means is that if there are 4 inputs, but only two spaces remaining in the queue and **in[0]** sends nothing or does not enable what it sends then this does not mean that **in[2]** gets to go; **in[2].ack** already is set to LSE_signal_nack set. If you want arbitration, put an arbiter in front of the mqueue.

The *pass_acks_when_full* parameter, when TRUE, allows space freed by an ack on an **out** port instance this cycle to be filled this cycle by new data. This is not the case when FALSE.

The *in_order_acks* parameter only allows elements to be removed from the queue in-order.

The **mqueue** also has several other user points to customize its behavior. The contents of the queue may be sorted at the end of each time step. The comparison function is user-defined in the *comparison_func* user point.

At the end of a timestep, it is possible to control whether an element will be presented on the output of the mqueue for removal via the *present_func*. The *drop_func* can be used to drop elements from the queue at the end of timestep. The queue may be initialized with data at the start of simulation via the *init_func*.

The **in** and **out** port types must match.

Types

none

Parameters

size:**int**

The capacity of the queue.

pass_acks_when_full:**bool**

When TRUE, space freed by an ack may be filled by new incoming items in the same time step.

pass_acks_to_enable:**bool**

When TRUE, the enable signal on an instance of the **out** port is driven to match the ack signal received on that port instance. When FALSE, the enable signal is driven to match the data signal on that port instance.

The default value is TRUE

in_order_acks:**bool**

When TRUE, receives are only recognized in-order.

```
boolean comparison_func(LSE_dynid_t id1, LSE_port_type(in) *data1,
LSE_dynid_t id2, LSE_port_type(in) *data2);
```

Provides a comparison function for sorting the queue contents. The return value indicates whether item 2 should be closer to the front than item 1. In other words, return TRUE if there is an inversion.

```
boolean present_func(LSE_dynid_t id, LSE_port_type(in) *data);
```

Tells whether the value should be presented in the next timestep. The user point is invoked at phase end. Return TRUE if the value should be presented, FALSE otherwise. The default value is

```
<<< return FALSE; >>>
```

This function works just like a filter; it does not make unpresented data not take up space on the output port. For that, what you really want is an arbiter on the output. What makes it different from just filtering in the output control function is that it allows you to use "state" which may change during the cycle, such as user-defined fields of dynid structures, without worrying about whether you've got the currently computed value or not (this could be handled with appropriate port queries in the control function, but....)

Note: An ack received on a port which is not presented is ignored; we do not want to ack like the module on the other side is prescient.

```
boolean drop_func(LSE_dynid_t id, LSE_port_type(in) *data);
```

Tells whether the value should be dropped in the next timestep. The user point is invoked at phase end. Return TRUE if the value should be dropped, FALSE otherwise. The default value is

```
<<< return FALSE; >>>
```

Note: If the queue was full and you still try to insert things, you will successfully insert them if they are dropped. Also, anything new that is inserted is checked by this function.

```
int init_func(LSE_dynid_t *dynids, LSE_port_type(in) *data, int size);
```

Initializes the **mqueue** with data. The return value indicates the number of elements initialized. The *size* parameter gives the maximum size of the mqueue. Initial dynamic identifiers and data should be stored in the *dynids* and *data* arrays. The first element goes in slot 0, the second in slot 1, etc.

Events

none

Ports

in:'a

The input port.

out:'a

The output port.

Functions

none

The pipe Module

Short description

Instances of this module implement a variable-latency pipeline.

Detailed description

Instances of this module implement a variable-latency pipeline. The depth of the pipeline is always at least one stage, but it is possible to vary the depth on a per-item basis. However, items are always dealt with in a FIFO order; you cannot insert an item in front of an item already in the pipeline.

Data is inserted into the pipeline via the **in** port and extracted via the **out** port. Multiple port instances of the **in** port create multiple parallel pipelines. A single pipeline is always single input/single output. That is to say that if an element arrives on **in[i]** then it will emerge on **out[i]** sometime in the future, provided that it is not dropped.

The maximum depth of all pipelines is given by the *depth* parameter. The stage at which a new element is added to the pipeline is controlled by the *delay_for_send* user function. When input data appears on the **in** port, the *delay_for_send* user function is invoked and its return value specifies how many pipeline stages the input element must go through before emerging on the out port. For example, if a new element arrives on **in[i]** and *delay_for_send* returns 3, in the absence of any stall conditions, the element will appear on **out[i]** 3 cycles in the future. Items are only inserted in the pipeline if their corresponding enable signal is `LSE_signal_enabled`.

The *space_available* user point is called to determine the value of the **in** port instances' ack signal. The function is passed the input data, the data currently being output, how many stages are occupied, and how many real elements (non-bubbles) are in the pipeline (thus `non_bubble_count <= curr_fullness`). The return value of the *space_available* user point is either `pipe::ret_yes`, `pipe::ret_no`, `pipe::ret_unknown`, or `pipe::ret_ifoutack` which will ack the input if the output is acked (this return value is the same as `pipe::ret_no` if the *pass_acks_when_full* parameter is `FALSE`). Realize that any scheduling needed to resolve an unknown value must be set up by the user (either through use of a scheduled query, which will be handled automatically, or via a explicit scheduling API call). Also, beware that *space_available* is ONLY used to calculate the output ack. If *delay_for_send* requests and illegal insertion (i.e. reordering of elements in the pipeline) or if `curr_fullness == depth`, the element may be acked but *not* inserted into the pipeline.

Bubbles are squashed at a rate of 1 consecutive bubble per cycle.

A simple n-stage pipeline can be created by retaining the default values for all user functions and simply setting the *depth* parameter to the depth of the pipeline.

There is an additional user point called *drop_func* which is called at the end of cycle on any element that will be output. If it returns `TRUE`, the element is dropped and not output.

The **in** and **out** ports must have the same type.

Types

```
pipe::space_available_return_t:enum {ret_yes, ret_no, ret_ifoutack,ret_unknown}
```

Return type for the *space_available* user point.

Parameters

```
pass_acks_to_enable:bool
```

When **TRUE**, the enable signal on an instance of the **out** port is driven to match the ack signal received on that port instance. When **FALSE**, the enable signal is driven to match the data signal on that port instance.

The default value is **TRUE**

```
int delay_for_send(int porti, LSE_dynid_t id, LSE_port_type(out)
*data);
```

This user point determines latency of a newly added element. The default value is

```
<<<return PARM(depth);>>>
```

```
space_available_return_t space_available(int porti, LSE_signal_t
in_status, LSE_dynid_t in_id, LSE_port_type(in) *in_data,
LSE_signal_t out_status, LSE_dynid_t out_id, LSE_port_type(in)
*out_data, int curr_fullness, int non_bubble_count);
```

This user point computes whether space is available in the pipe and set the ack on the **in** port appropriately. The default value can be found by examining *pipe.lss*, since it uses an unsupported LSS function.

A warning regarding the *space_available* user point

This user point only sets the ack of the **in** port. Just because it says to ack does *not* mean that the element will be inserted. Space must be available in the pipe and the return value of *delay_for_send* must not cause a reordering of elements.

```
boolean drop_func(int porti, LSE_dynid_t id, LSE_port_type(out)
*data);
```

This user point is used to drop elements from the pipeline instead of emitting them. The default value is

```
<<<return FALSE>>>
```

Events

none

Ports

in:'a

The input port.

out:'a

The output port.

Functions

none

Note: See `pipe.lss` for an unsupported, but perhaps useful, `delay_for_ack` LSS function.

The router Module

Short Description

Instances of this module routes the values from the in ports to the out ports.

Detailed Description

Instance of this module routes the values from the **in** port instances to the **out** port instances based on the routing information coming in via the **route_info** port instances. The routing information will be a destination-to-source mapping. (If only source-to-destination mapping is available, use a **converter** module to convert this into a destination-to-source mapping). In the usual case, being loose with notation, **out[i].data** gets **in[route_info[i].data_value].data**, **out[i].enable** gets **in[route_info[i].data_value].enable**, **in[i].ack** gets **out[route_info[i].data_value].ack**, **in[i].data_value** goes to **out[route_info[i].data_value].data_value**, and **in[i].id** goes to **out[route_info[i].data_value].id**.

If **in.width** > **out.width**, then for all **in** port instances which do not take part in the routing, an instance of this module sends **LSE_signal_ack** for the **in** port instances which have **LSE_signal_nothing** for data and **LSE_signal_nack** for the in port instances which have **LSE_signal_something** for data and did not take part in routing. For the **in** port instances which did take part in the routing, the acks are propagated from the corresponding **out** port instances as described above.

If **out.width** > **in.width**, then for all **out** port instances which do not take part in the routing, an instance of the **router** module simply send **LSE_signal_nothing** and **LSE_signal_disabled** while ignoring the acks coming in via these ports.

The user can specify a default routing via the *default_routing_function* user point, that will be used if any of the **route_info** port instances brings in an **LSE_signal_nothing**. The default user function will be used to fill the routing table with invalid values so that no routing is done for ports for which the routing information is not specified.

route_info.width must be equal to **out.width**. **route_info** must have type **int**. **in** and **out** must have the same type.

Note: The current implementation of the router is conservative and waits for all **route_info** inputs before computing any output values.

Types

none

Parameters

```
void default_routing_function(int in_width, int out_width, int
route_table[], int invalid_val, int nothing_val);
```

This user point determines routing information not available via the **route_info** ports. The default value is:

```
<<< {
    int i;
```

```

    for(i=0;i<out_width;i++)
        route_table[i] = invalid_val;
} >>>

```

The *invalid_val* argument is placed in the routing table to indicate no route for a particular input. The *nothing_val* should be placed in the route table if no input is routed to the given output. `route_table[i]` is used to select an **in** instance for **out[i]**.

Events

none

Ports

in:'a

The input port.

out:'a

The output port to which inputs get routed.

route_info:int

The port on which routing information is recieved.

Functions

none

The selector Module

Short Description

The selector is a special case of an aligner/router pair.

Detailed Description

Selector instances function like an aligner instance followed by a special router instance. The signals on the **in** port instances are aligned and then routed to instances of the **out** port instances that have the ack signal set to `LSE_signal_ack`.

Note: The behavior of this module breaks the standard control flow paradigm. The datapath connected to the **out** port instances must generate the ack signal independently of their input data and enable signals.

The **in** and **out** port must have the same type.

Types

none

Parameters

none

Events

none

Ports

in:'a

The input port.

out:'a

The output port with selected input values.

Functions

none

The serializer Module

Short Description

The serializer module serializes its inputs. See the detailed description.

Detailed Description

The serializer module is useful for ensuring that a set of signals that arrive in-order. For example, if there is an instruction window module with 4 output port instances (meaning that four instructions can be ready per cycle) and we want to insure that instructions are processed in order (with earlier instruction output by the window on lower port instances), then we would connect the output of the instruction window unit to the serializer in the straightforward fashion.

The serializer module takes an a set of input signals, call them `input[i]`, $i=1..n$ and produces an output signal `output[i]` if and only if for all $j<i$ `input[j]` is present. The input signals are the data and enable signals on the **in** port instances and the ack signal on the **out** port instances. The output signals are the data and enable signals on the **out** port instances and the ack signals on the **in** port instances. Inputs are indexed in port instance order. For example, for the `in[i].data` signal, `out[i].data` would get `in[i].data` if and only if `in[j].data` for $j<i$ was `LSE_signal_something`. (The corresponding values for enable and ack are `LSE_signal_enabled` and `LSE_signal_ack`).

The data, enable and ack signals are serialized independently if the corresponding `serialize_data`, `serialize_enable`, and `serialize_ack` parameters are set to `TRUE` (which is the default). If the parameter is `FALSE`, the data is simply passed through the module to the corresponding output on the other port instance (`out[i]` for the data and enable signal and `in[i]` for the ack signal).

The **in** and **out** port types must match, and the input port width must be non-zero and must match the output port width.

Types

none

Parameters

serialize_ack:**bool**

Determines if acks are serialized or passed through. The default value is `TRUE`

serialize_data:**bool**

Determines if data is serialized or passed through. The default value is `TRUE`

serialize_enable:**bool**

Determines if enables are serialized or passed through. The default value is `TRUE`

Events

none

Ports

in: 'a

The input port.

out: 'a

The output port with serialized signals.

Functions

none

The sink Module

Short Description

Instances of the sink module simply ack any data on the in port and throw it away.

Detailed Description

Instances of the sink module simply ack any data on any **in** port instance and then throw it away. The instance will call the `sink_func` on every port instance at the end of cycle with the **in** port status, **in** port data, dynamic identifier and port instance number.

Types

none

Parameters

```
void sink_func(LSE_signal_t status, LSE_dynid_t id, LSE_port_type(in)
*data, int porti);
```

Called on every **in** port instance at the end of timestep. The default value is:

```
<<< ; >>>
```

Events

none

Ports

in:'a

The input port. All data recieved on this port is acknowledged and thrown out.

Functions

none

The source Module

Short Description

Instances of the source module emits user defined data on its output port and ignores the ack signal.

Detailed Description

Instances of the source module will output a data signal and an enable signal on its **out** port instances depending on the value of the *create_data* user point. The user point passes has three arguments, *porti*, *id*, and *data*. The user point code should put into the cell referenced by the *data* pointer a value that should be output. This data will be output on **out[porti]** with dynamic identifier *id*. The status of the

out[porti] port instances' data and enable signals is the value of these fields in the status returned by the user point. By default instances output `LSE_signal_nothing` and `LSE_signal_disabled`.

Types

none

Parameters

```
LSE_signal_t create_data(int porti, LSE_dynid_t id,
LSE_port_type(out) *data);
```

Determines how and if output data is created. *porti* gives the **out** port instance number, *id* is the dynid that will be output, and *data* is a pointer to a data element that should be filled in with the data value.

The default value is:

```
<<< return LSE_signal_nothing | LSE_signal_disabled >>>
```

Events

none

Ports

out:'a

The output port on which created data is output.

Functions

```
source::create_constant(x : literal) => string
```

This function returns a value for the *create_data* user point that will create a data element with the value passed in *x*. *x* is a **literal**, and thus is treated like a constant literal of the datatype of the **out**

port.

The tee Module

Short Description

Instance of the tee module fan out an input signal to multiple output signals

Detailed Description

An instance of the tee module fans out input on its **in** port instances to multiple output port instances on the **out** port. If there are *m* instances on the **in** port, and *n* instances of the **out** port, the input data and enable signals on port **in**[*i*] are forwarded to **out**[(*n/m*)**i*+*j*] for *j*=1..*n/m*. (*n* must be evenly divisible by *m*).

Instances also propagate ack signals on the **out** port back to the **in** ports. If the *control_flow_style* parameter is set to `tee::and_acks`, the ack signals from port instances **out**[(*n/m*)**i*+*j*], *j*=1..*n/m*, are anded together and the result is forwarded to input port **in**[*i*] (with `LSE_signal_ack` being 1 and `LSE_signal_nack` being 0). For any ack signal on **out**[(*n/m*)**i*+*j*], if there is an unknown ack value but a known ack signal is `LSE_signal_nack`, then the **in**[*i*]'s ack signal is set to `LSE_signal_nack`. If the *control_flow_style* parameter is set to `tee::or_acks`, the ack signal on input port **in**[*i*] is computed by oring the ack signals on **out**[(*n/m*)**i*+*j*] in a similar fashion, with a known ack signal with value `LSE_signal_ack` forcing **in**[*i*]'s ack signal to be `LSE_signal_ack`. The default setting is to and ack signals.

Types

```
tee::control_flow_style_t:enum {and_acks, or_acks}
```

Type for the *control_flow_style* parameter

Parameters

```
control_flow_style:control_flow_style_t
```

Determines whether acks are anded or ored to generate ack for the **in** port instances. The default value is `tee::and_acks`

Events

none

Ports

in:'a

The input port.

out:'a

The output port.

Functions

`tee::connect_bus_from_tee_output (from : port ref, to : port ref, width : int, ratio : int, tee_out_num : int) => void`

This function will connect port instances from $[\text{ratio} * i + \text{tee_out_num}]$ to port $\text{to}[i]$; This is useful when fanning out a bus of signals. For example to expand the following connection pattern:

```
using corelib;

instance bar:source;
instance hole0:sink;
instance hole1:sink;
instance hole2:sink;
instance fo:tee;

bar.out ->:int fo.in;
fo.out[0] -> hole0.in;
fo.out[1] -> hole1.in;
fo.out[2] -> hole2.in;
```

to a five wide bus one would write:

```
using corelib;

instance bar:source;
instance hole0:sink;
instance hole1:sink;
instance hole2:sink;
instance fo:tee;

LSS_connect_bus(bar.out, fo.in, 5, int);
tee::connect_bus_from_tee_output(fo.out, hole0.in, 5, 3, 0);
tee::connect_bus_from_tee_output(fo.out, hole1.in, 5, 3, 1);
```

```
tee::connect_bus_from_tee_output(fo.out, hole2.in, 5, 3, 2);
```

ratio is the input bus width versus the output of output busses, tee_out_num is which output set to connect to. (this number should be between 0 and ratio-1).

The wire Module

Short description

Instances of this module forwards the input data and enable signals to the output and vice versa for the ack signal.

Detailed Description

Instances of this module will forward the data and enable signals on port instance **in[i]** to the data and enable signals of port **out[i]**.

The **in** and **out** ports must have the same type and width.

Types

none

Parameters

none

Events

none

Ports

in:'a

The input port

out:'a

The output port

Functions

none