March 10, 2013

Copyright © 2013 D. E. Shaw Research

All Rights Reserved

Programmer's Guide

Cascade Programmer's Guide

Notice

The Cascade Programmer's Guide and the information it contains is offered solely for educational purposes, as a service to users. It is subject to change without notice, as is the software it describes. D. E. Shaw Research assumes no responsibility or liability regarding the correctness or completeness of the information provided herein, nor for damages or loss suffered as a result of actions taken in accordance with said information. No part of this guide may be reproduced, displayed, transmitted, or otherwise copied in any form without written authorization from D. E. Shaw Research. The software described in this guide is copyrighted and licensed by D. E. Shaw Research under separate agreement. This software may be used only according to the terms and conditions of such agreement.

Copyright

© 2013 by D. E. Shaw Research. All rights reserved.

Trademarks

All trademarks are the property of their respective owners.

Contents

1.0 Overview 6

1.1 Running a simulation 6
1.2 A “hello world” simulation 7
1.3 Cascade parameters 9

2.0 Components 10

2.1 Subcomponents and component names 10
2.2 Customizing component names 11
2.3 Templates and derived components 12
2.4 Cycle-based behavioral implementation 12
2.5 Modeling reset 14
2.6 Application-specific Component extentions 15

3.0 Ports 16

3.1 Port value types 17
3.2 Port response types 17
3.3 Valid flags and port names 18
3.4 Accessors 19
3.5 Ports and update functions 20
3.6 Combinational connections 23
3.7 Synchronous connections 25
3.8 Fifo ports 27
3.9 Port arrays 31
3.10 Reset 33

4.0 Clock Domains 35

4.1 Defining clocks 35
4.2 Timing of rising clock edges 37
4.3 Multiple clocks 37
4.4 Manual clocks 38
4.5 Assigning components, ports and update functions to clock domains 39
4.6 Connections and clock domains 40
4.7 Helper functions 40
4.8 Custom rising clock edge behaviour 41
4.9 Manually scheduled events 41

5.0 Interfaces 43

5.1 Implementing an interface 44
5.2 Instantiating an interface implementation 45
5.3 Extending interfaces 45
5.4 Custom constructors 45
5.5 Connecting interfaces 46
5.6 Reads/writes declarations 47
5.7 Retrieving the component pointer 47
5.8 Reset order 47

6.0 Arrays 48

6.1 Named Arrays 48
6.2 Element names 49
6.3 Arrays with custom allocators 49
6.4 Dynamic allocation of interfaces 51
6.5 Calling a function on all array elements 51

7.0 Activation and Triggers 52

7.1 Multiple update functions 53
7.2 Triggers 53

8.0 Bit Vectors 56
8.1 Indexing bit vectors........57
8.2 Bit vector slices........57
8.3 Concatenation........57
8.4 Reduction operators........58
8.5 Functions........58
8.6 Assignments to signed bit vectors........58
8.7 Bit vector traits........59
8.8 Using bit vectors as ports........59
8.9 String conversion........60
8.10 Bit vector references........60

9.0 Logging and Tracing 62
9.1 Component tracing........62
9.2 Tracing from an interface........62
9.3 Controlling tracing based on time........62
9.4 Trace specifier format........62

10.0 Archiving 64
10.1 Checkpoints........64
10.2 Archiving custom data........65
10.3 Archiving component/interface pointers........66

11.0 Multithreading 67
11.1 Subdividing clock domains........67
11.2 Controlling threading........67
11.3 Avoiding race conditions........67

12.0 Generating VCD Files 68
12.1 Functions controlling waves generation........68
12.2 Clocks and timing model........68
12.3 Fifo ports........70
12.4 Invalid signals........71
12.5 Explicit Signals........71
12.6 Data Representation........71
12.7 Controlling waves generation from the command line........72

13.0 Cascade/Verilog Co-simulation 73
13.1 Interface binding........74
13.2 Data marshalling........75
13.3 Port types........76
13.4 Instantiating Cascade models within Verilog........77
13.5 Instantiating Verilog modules within Cascade........82
13.6 Additional functions........85

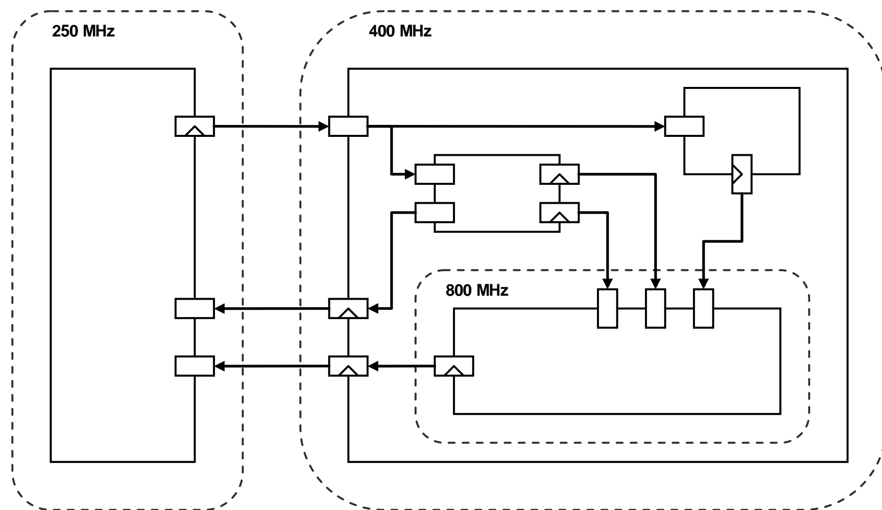
14.0 Example: Conway’s Game of Life 87
14.1 Life chip design........87
14.2 ROM source code........88
14.3 Cell source code........89
14.4 Controller source code........90
14.5 Life chip source code........91
14.6 Main program source code........92

15.0 Reference 93
15.1 Component functions........93
15.2 Interface functions........94
15.3 Port functions........94
15.4 Fifo port functions........96
15.5 Parameters........97
15.6 Rising clock edge behaviour........99
15.7 Recommended coding standards........100

1.0 Overview

Cascade is a C++ framework for creating cycle-based hardware simulations. It has been designed to provide fast, modular simulations with a minimal amount of programmer effort. When writing a simulation, the hardware is broken down into a hierarchy of *components* in the same way as when using Verilog or VHDL. Each component is implemented as a separate C++ class which has input ports, output ports, and internal state. Communication between components is effected by connecting inputs and outputs before the simulation begins. The simulation is divided into one or more *clock domains*, each of which has a distinct clock driver. Figure 1 illustrates a component hierarchy with inputs, outputs and clock domains.

Figure 1. Illustration of components, connected IOs and clock domains in Cascade.



1.1 Running a simulation

A simulation is divided into three primary phases: *construction*, *initialization*, and *simulation*. A fourth *teardown* phase also implicitly exists at the end of a simulation.

1.1.1 Construction

During the construction phase the C++ objects representing all components, ports and clock domains are constructed, and connections are established between components.

1.1.2 Initialization

The initialization phase consists of a call to the global function `Sim::init()`. All construction and connections must occur prior to this call. During this phase the internal simulation state is prepared and optimized, and the simulation is reset. It is generally not necessary to call `Sim::init()` explicitly because Cascade will call it automatically when the simulation begins.

1.1.3 Simulation

After initialization, the simulation will be at time zero. Two global functions are provided to advance the simulation time:

```
void Sim::run (uint64 runTime = 0)
void Sim::runUntil (uint64 endTime)
```

`Sim::run()` runs the simulation for the specified amount of time measured in picoseconds; any rising clock edges scheduled for exactly `runTime` in the future will not be evaluated. If `runTime` is zero (default), the next rising clock edge is evaluated and the simulation time is advanced to the subsequent rising clock edge (which is not evaluated). `Sim::runUntil()` runs the simulation until the time reaches the specified `endTime` (also measured in picoseconds); again, any rising clock edges scheduled for exactly `endTime` will not be evaluated. As the simulation advances, the current simulation time in picoseconds is stored in the global variable `Sim::simTime`.

Within the simulation phase, execution is driven by the sequence of rising clock edges. Each rising clock edge causes synchronous state to be updated, followed immediately by combinational evaluation, which occurs in zero time. The update of synchronous state is generally automated within the Cascade infrastructure, while the programmer is responsible for implementing combinational evaluation.

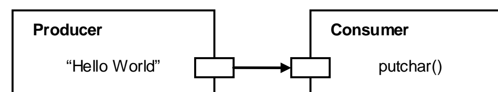
1.1.4 Teardown

The simulation infrastructure will automatically tear itself down and clean up when all components are destroyed (so be careful not to forget to delete any dynamically allocated components). Once teardown is complete, a new simulation may be created (by again constructing all components and running the simulation) without restarting the executable.

1.2 A “hello world” simulation

Let’s take a quick look at a very simple example involving two components: a producer that generates a stream of characters, and a consumer that prints these characters on `stdout` (Figure 2).

Figure 2. Simple producer-consumer design.



The code for this example is as follows:

```
#include <Cascade.hpp>

class Producer : public Component
{
    DECLARE_COMPONENT(Producer);
public:
    Producer (COMPONENT_CTOR) {}

    //---------------------------------
    // Interface
    //---------------------------------
    Output(char, out_ch);

    //---------------------------------
    // Simulation
    //---------------------------------
    void reset ()
    {
        m_ch = "Hello World\n";
    }

    void update ()
    {
        out_ch = *m_ch;
        if (*m_ch)
            m_ch++;
    }

private:
    const char *m_ch;
};

class Consumer : public Component
{
    DECLARE_COMPONENT(Consumer);
public:
    Consumer (COMPONENT_CTOR) {}

    //---------------------------------
    // Interface
    //---------------------------------
    Input(char, in_ch);

    //---------------------------------
    // Simulation
    //---------------------------------
    void update ()
    {
        if (in_ch)
            putchar(in_ch);
    };
};

void main ()
{
    Producer producer;
    Consumer consumer;
    consumer.in_ch << producer.out_ch;
    Sim::run(100000);
    Sim::reset();
    Sim::run(100000);
}
```

Each component is a C++ class (Section 2.0) and has a single port (Section 3.0). The cycle-based behavior is defined by the `update()` functions (Section 2.4), and the reset behavior of `Producer` is defined by its `reset()` function (Section 2.5). The first three lines of `main()` construct the simulation by instantiating the components and establishing a connection between their ports (Section 3.6). The first call to `Sim::run()` automatically initializes and resets the simulation, following which the simulation runs for 100,000 ps. The model is then reset, and the simulation is run for an additional 100,000 ps, producing the following output:

```
Hello World
Hello World
```

1.3 Cascade parameters

Many aspects of Cascade are controlled by a set of parameters defined in the header file `Params.hpp` (refer to "The descore Library" for a complete description of parameters). These parameters are all in the parameter group "cascade". Cascade parameters can be accessed by name (e.g. on the command line) with the "`cascade.`" prefix, and they can be accessed within code as members of the structure `Cascade::params`. For example, `cascade.ClockRounding` can be set on the command line via

```
-cascade.ClockRounding=0
```

and it can be set from within code as follows:

```
Cascade::params.ClockRounding = 0;
```

Within this manual we will refer to parameters using the "`cascade.`" command-line prefix, with the understanding that this must be replaced by the prefix `Cascade::params` in order to access parameters from the code.

2.0 Components

A component is a C++ class that obeys the following three rules:

1. It must inherit from `Component`.
2. It must contain the macro `DECLARE_COMPONENT(class [, name])`.
3. The argument list of each constructor declaration must end with the macro `COMPONENT_CTOR`; the argument list of each constructor implementation must end with the macro `IMPL_CTOR`.

Example

```
class BlackBox : public Component
{
    DECLARE_COMPONENT(BlackBox, BB);
public:
    BlackBox (COMPONENT_CTOR) {}
    …
};
```

If a component inherits from multiple base classes, `Component` (or a class derived from `Component`) must appear first in the inheritance list. In the `DECLARE_COMPONENT` macro, the first macro argument must be the exact class name, and the second argument is an optional human-readable component name, which is used to automatically construct fully-qualified instance names. If the second argument is omitted, the human-readable name defaults to the class name. The `COMPONENT_CTOR` and `IMPL_CTOR` macros do not effectively modify the constructor argument lists and so can be ignored when components are constructed.

Example

```
class SizedBox : public Component
{
    DECLARE_COMPONENT(SizedBox, SB);
public:
    SizedBox (int size, COMPONENT_CTOR);
    …
};

SizedBox::SizedBox (int size, IMPL_CTOR) : m_size(size)
{
    …
}

SizedBox myBox(4);
```

2.1 Subcomponents and component names

Components can contain other components and in this manner form a hierarchy. A subcomponent can either be contained directly or declared as a pointer and dynamically created during construction. For example, we can define a component which consists of two BlackBoxes as follows:

```
class TwoBlackBoxes : public Component
{
    DECLARE_COMPONENT(TwoBlackBoxes, TwoBB);
public:
    TwoBlackBoxes (COMPONENT_CTOR) : m_pBox(new BlackBox) {}

protected:
    BlackBox m_box;
    BlackBox *m_pBox;
};
```

We refer to the containing component as the *parent component* of the subcomponents. Each component in the hierarchy is automatically given a unique name of the form

```
<parent name>.<child name><id>
```

where <*parent name*> is the full name of the parent component, <*child name*> is the human-readable name of the child component, and <*id*> is a non-negative integer which is chosen to make the name unique, and is omitted if there are no sibling components with the same name. For example, if a simulation is created which contains a single `TwoBlackBoxes` component, then three components are created with names

```
TwoBB
TwoBB.BB0
TwoBB.BB1
```

The order in which ids are assigned to subcomponents with the same name is the order in which the components are constructed by the C++ code. A component's name can be retrieved as a `strbuff` (see "The descore Library") using the `getName()` member function:

```
strbuff getName () const
```

To prevent a component name from appearing in the naming hierarchy, specify 0 as the second argument to `DECLARE_COMPONENT`, e.g.

```
DECLARE_COMPONENT(BlackBox, 0);
```

2.2 Customizing component names

In most cases the automatically-generated component names are sufficiently descriptive and relieve the programmer from having to explicitly name every component instance. More descriptive per-instance names can be provided by adding the member variable `m_componentName` with type `char *`, `const char *` or `string`. Cascade detects the presence of this member variable and uses its contents in place of the human-readable component name.

Example

```
class BlackBox : public Component
{
    DECLARE_COMPONENT(BlackBox);
public:
    BlackBox (const string &name, COMPONENT_CTOR) : m_componentName(name) {}
    …
    string m_componentName;
};

class TwoBlackBoxes : public Component
{
    DECLARE_COMPONENT(TwoBlackBoxes, TwoBB);
public:
    TwoBlackBoxes (COMPONENT_CTOR) : m_box1("primary"), m_box2("secondary") {}
    …
    BlackBox m_box1;
    BlackBox m_box2;
};
```

In the above example, which modifies `BlackBox` and `TwoBlackBoxes`, if a simulation is created which contains a single `TwoBlackBoxes` component, then three components are created with names

```
TwoBB
TwoBB.primary
TwoBB.secondary
```

2.3 Templates and derived components

Both templates and inheritance may be used to create components. A templated component is defined and instantiated in exactly the same manner as a templated class, as shown in the following example:

```
template <class T>
class FIFO : public Component
{
    DECLARE_COMPONENT(FIFO);
public:
    FIFO (COMPONENT_CTOR) {}
    …
protected:
    T m_values[8];
};

FIFO<int> fifo;
```

Derived components are defined by inheriting from a base component class instead of inheriting directly from `Component`:

```
class NewBlackBox : public BlackBox
{
    DECLARE_COMPONENT(NewBlackBox, NewBB);
public:
    NewBlackBox (COMPONENT_CTOR) {}
    …
};
```

Multiple inheritance of components is not supported, although a component may inherit from multiple classes so long as the first class in the inheritance list (and only the first) is a component.

2.4 Cycle-based behavioral implementation

Combinational evaluation within a clock cycle is modeled using *update functions*. Update functions are component member functions which are automatically called by the Cascade infrastructure on every clock cycle. Any member function with no arguments or return value can be an update function. To facilitate the common case in which a component has a single update function, a member function named '`update`' is automatically recognized by the Cascade infrastructure as a default update function:

```
void update ();  // Cascade automatically recognizes and calls this function
```

When `update()` is implemented, it will be called on every clock cycle; no additional programmer action is required. Other update functions must be explicitly registered in the component constructor using the macro `UPDATE(function)`. The following example shows a component with two update functions; the default `update()` function and a second `updateOutputs()` function that is explicitly registered.

```
class TwoUpdates : public Component
{
     DECLARE_COMPONENT(TwoUpdates);
public:
     TwoUpdates (COMPONENT_CTOR)
     {
          UPDATE(updateOutputs);
     }

     void update ()
     {
          …
     }

     void updateOutputs ()
     {
          …
     }
};
```

The relative order in which update functions are invoked is, by default, unspecified. In order to correctly handle combinational dependencies between update functions, additional information must be supplied to the infrastructure to ensure that updates are ordered accordingly. The mechanism for specifying this information will be discussed in Section 3.5.

The default `update()` function is non-virtual. If both a base class and a derived class define `update()` and the base class function is not explicitly registered, then only the derived class' function will be called. The programmer can cause the base class `update()` function to be invoked in one of two ways:

1. by registering it with Cascade using the `UPDATE` macro, or
2. by calling it directly from the derived class' `update()` function.

In all cases, an explicitly registered non-virtual update function will be called on every cycle irrespective of whether or not a base component/derived component has an update function with the same name. Virtual functions which are registered as update functions maintain virtual function semantics: only the most derived update function is ever invoked automatically.

In addition to combinational evaluation, a component can specify custom rising clock edge behavior by implementing the function `tick()`:

```
void tick ()
```

This function is automatically recognized by Cascade and called on each rising clock edge. The following example shows a component with a member variable that is reset to zero on every rising clock edge.

```
class Tickable : public Component
{
     DECLARE_COMPONENT(Tickable);
public:
     Tickable (COMPONENT_CTOR) : m_count(0) {}

     …
     void tick ()
     {
          m_count = 0;
     }
private:
     int m_count;
};
```

2.5 Modeling reset

Reset behavior is modeled within a component by implementing one of the following functions:

```
void reset ();
void reset (int level);
```

The second form allows different reset levels to be implemented; Cascade predefines constants for warm and cold reset:

```
const int RESET_COLD = 0;
const int RESET_WARM = 1;
```

Cascade automatically calls these functions on all components when the simulation is initialized. If the functions are not implemented, then a default implementation is invoked which does nothing. Cascade does not know which version of `reset()` a given component implements, so it calls both, supplying the argument `RESET_COLD` to the latter. The assumption is that at least one of the calls will be directed to the default null implementation, but if for some reason a component implements both versions of `reset()` then both will be called.

The `reset()` functions are intended to supplement component constructors by initializing all component state. Member variables, which in other software would normally be initialized within the constructor, should instead be initialized within `reset()`. This ensures correct behavior if the simulated hardware is dynamically reset during the simulation. Ports should also be reset within this function, as will be discussed in Section 3.10. The `reset()` functions have modified constructor semantics: if a derived component inherits from a base component, then `reset()` is called on the base component first, on the derived component second, then on member components of the derived component (if any) third. Thus, it is never necessary to explicitly call `reset()` for base or member components.

For a number of reasons, a componet's `reset()` function may be called multiple times. This does not present a problem for straightforward initialization of state via assignment, however if additional code is placed within `reset()`, then care must be taken to ensure that this code is idempotent (i.e. that it can be executed multiple times without additional side effects). So, for example, if part of a component's state consists of a dynamically allocated array, then the memory for the array should be allocated within the component constructor, but the contents of the array should be initialized within `reset()`.

Another restriction on reset functions is that they should avoid calling virtual functions, because Cascade temporarily replaces the component virtual function table in order to ensure that reset is called for base classes. Calling a virtual function can therefore produce unexpected results, including a run-time error due to a pure virtual function call.

Example

```
class Randomizer : public Component
{
    DECLARE_COMPONENT(Randomizer);
public:
    Randomizer (COMPONENT_CTOR) {}

    virtual int initialSeed () = 0;

    void reset ()
    {
        m_randval = initialSeed();  // WRONG!!!  Will result in pure virtual call
    }

    int m_randval;
};
```

At any point during a simulation, some or all of the component hierarchy can be reset by calling one of the following two global functions:

```
void Sim::reset (int level = RESET_COLD);
void Sim::reset (Component *component, int level = RESET_COLD);
```

The first of these functions resets the entire simulation; the second resets the specified component and all of its sub-components. Again, both versions of `reset()` will be called for affected components, using the supplied `level` argument for the second form. These functions will also reset the state of all Fifos (Section 3.8) connected to a component being reset.

2.6 Application-specific Component extentions

All components inherit from the base class `ComponentExtensions`, defined in `ComponentExtensions.hpp`. This class serves as a placeholder for any application-specific customizations to components, i.e. member variables or virtual functions that should be shared by all components in the application. The files `ComponentExtensions.hpp` and `ComponentExtensions.cpp` are the only Cascade files intended to be modified to customize an application.

Example

```
struct ComponentExtensions : public Cascade::InterfaceBase
{
    // Component name
    strbuff getName ();

    //============================================================
    // Application-specific customizations

    // Statistics
    virtual void recordStats () {}
};
```

A function with no arguments or return value defined in this base class can be called across all components or a subset of components using the helper function

```
void Sim::doComponents (void (Component::*f) (), const char *wildcardName)
```

The first argument is a pointer to the function, and the second argument is a wildcard string: the function will be invoked on any component whose name matches the wildcard string, with parent components tested before child components.

Example

```
Sim::doComponents(&Component::recordStats);
```

3.0 Ports

Ports define the interface to components and allow signals to propagate through the design. Cascade defines three basic types of ports: Input, Output and InOut. Each port conceptually holds a single value of an arbitrary type. Macros with the corresponding names are provided to declare ports:

```
Input(<type>, <name>)
Output(<type>, <name>)
InOut(<type>, <name>)
```

<*type*> is the C++ type of the port, and <*name*> is the name of the port. The following example shows a component with three ports:

```
class Adder : public Component
{
    DECLARE_COMPONENT(Adder);
public:
    Adder (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in_a);
    Input(int,  in_b);
    Output(int, out_sum);

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void reset ()
    {
        out_sum.reset(0);
    }
    void update ()
    {
        out_sum = in_a + in_b;
    }
};
```

In this example the (default) combinational update function reads the two inputs and sets the output to their sum. Each port is implemented as a templated class which holds a single pointer to the specified type. The port class names are the same as the macro names:

```
template <class T> class Input;
template <class T> class Output;
template <class T> class InOut;
```

In general it is recommended that the port macros be used as they supply Cascade with the port names (see Section 3.3). The ports in the above example could, however, be equivalently declared as follows:

```
class Adder : public Component
{
    DECLARE_COMPONENT(Adder);
public:
    Adder (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input<int>  in_a;
    Input<int>  in_b;
    Output<int> out_sum;
    …
```

The one restriction on the port macros is that they must appear within a public section of the class declaration. Otherwise a gcc build will fail with an error message of the form:

```
error: 'struct Adder::_Counter<60, gcc_hack>' redeclared with different access
```

3.1 Port value types

Port values are internally copied as raw bits. As such, shared pointers and other objects with custom copy constructors or assignment operators should not be used as port values; these restrictions are enforced by static assertions. To allow objects with custom copy constructors or assignment operators to be used as port values when it is known that they can be safely copied as raw bits, use the macro

```
ALLOW_PORT_TYPE(<type>)
```

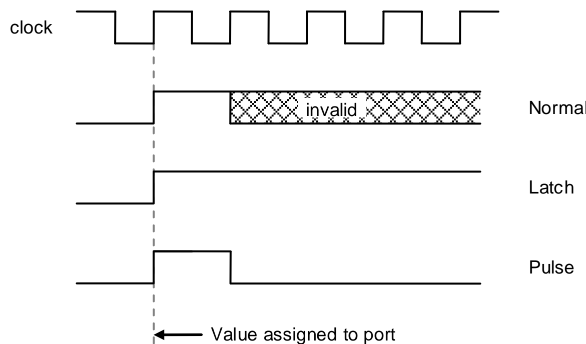
Example

```
struct vec2
{
    vec2 () {}
    vec2 (const vec2 &rhs) : a(rhs.a), b(rhs.b) {}
    vec2 &operator= (const vec2 &rhs) { a = rhs.a; b = rhs.b; return *this; }
    int a, b;
};

// This is required to avoid a static assertion failure when vec2 is used as a
// port type because vec2 has a custom copy constructor and assignment operator.
ALLOW_PORT_TYPE(vec2);
```

3.2 Port response types

Cascade supports three different port response types that define the behavior over time when a value is assigned to a port. *Normal* ports are valid from the time of the assignment to the next rising clock edge, and invalid thereafter. *Latch* ports maintain their state across rising clock edges. *Pulse* ports are reset to zero at each rising clock edge. These port response types are illustrated in Figure 3.

Figure 3. Port response types.



The default port response type is Normal. To change the response type of a port, use the `setType(<type>)` member function where <*type*> is one of

```
PORT_NORMAL
PORT_LATCH
PORT_PULSE
```

The `setType()` function must be called during the construction phase, as shown in the following example:

```
class Channel : public Component
{
    DECLARE_COMPONENT(Channel);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Output(bit, out_enabled);
    Output(bit, out_valid);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Channel (COMPONENT_CTOR)
    {
        out_enabled.setType(PORT_LATCH);
        out_valid.setType(PORT_PULSE);
    }
    …
};
```

3.3 Valid flags and port names

In debug builds, *valid flags* are associated with all port values. When a port is written the valid flag is set, and when a port is read the valid flag is inspected to ensure that the value is not stale. Finally, all normal port valid flags within a clock domain are cleared on each rising clock edge. This automatically detects a variety of common errors:

Invalid Outputs – A component's update function(s) may neglect to set one or more outputs, resulting in garbage values.

Uninitialized Outputs – An output which is not initialized at the start of the simulation will have a garbage value on the first cycle (this is only an issue if another port has a synchronous connection to the output; see Section 3.7).

Bad Update Order – If a producer component has a combinational output which feeds a consumer component, then the components must be updated in the correct order. If the consumer is updated first then it will read a value from the previous cycle.

Missing Connections – One or more required connections between ports may be omitted at construction time.

Incorrect Handshaking – Certain bits may be valid only on certain cycles depending on an interface's protocol. A producer may neglect to set these bits when it is supposed to, or a consumer may attempt to read these bits when it shouldn't.

Without valid flags, these errors tend to produce silent failures which are difficult to diagnose and correct. Valid flags allow Cascade to immediately and automatically catch these errors at run time. In order to provide the developer with the exact location of the failure, each port is given the name

```
<component name>.<port name>
```

where <*component name*> is the name of the component containing the port (as described in Section 2.1), and <*port name*> is the name of the port as specified in the port declaration macro. Ports which are declared directly using the templated port classes rather than the corresponding macros are given the generic names

"`Input`", "`Output`", "`InOut`", with a non-negative id appended to distinguish siblings (which, unlike subcomponents, is included even when there are no siblings of the same port type). In the Adder example of the previous section, the ports would be given the names

```
Adder.in_a
Adder.in_b
Adder.out_sum
```

As with components, a port's name can be retrieved using the `getName()` member function:

```
strbuff getName () const
```

3.3.1 Setting the valid flag manually

The `setValid()` member function can be used to set a port's valid flag when the value of the port has not changed from the previous cycle. This is less expensive than re-assigning the same value to the port. In debug builds this function sets the valid flag; in release builds it does nothing. This function cannot be used for read-only ports as it is intended to be used in place of assignment. The `dontCare()` member function is similar to `setValid()`, but is provided for clarity to handle the separate situation where an invalid value may be read speculatively but then discarded. In debug builds, `dontCare()` causes the port value to be filled with garbage.

3.4 Accessors

As illustrated in the adder example, ports can, for the most part, be accessed and assigned in exactly the same manner as ordinary variables of the same type. The following is the full set of accessors defined for reading, writing and modifying a port (named "io") of type T:

```
io = val           // [WRITE] Set valid flag and value, return value
io.nonConstPtr()  // [WRITE] Set valid flag, return non-const pointer to value
(T) io            // [READ] Check valid flag, return value
io()              // [READ] Check valid flag, return value
*io               // [READ] Check valid flag, return value
io.constPtr()     // [READ] Check valid flag, return const pointer to value
io->member        // [READ] Check valid flag, return member of structure T
```

In most cases the compiler is able to automatically cast a port of type T to a value of type T. However, there are two instances in which it is unable to do so and an explicit typecast must be performed (either with an actual cast or with the overloaded function call or dereference operators):

1. When the port is passed to a function with a variable number of arguments.

Example

```
Output(int, out);
…
printf("%d\n", (int) out);  // explicit typecast
printf("%d\n", *out);       // typecast using dereference operator
```

2. When the port is a structure and a member needs to be accessed

Example

```
struct TwoInts
{
     int a;
     int b;
};

Input(TwoInts, in);
…
int sum = in().a + in().b;  // typecast using function call operator
int sum = in->a + in->b;    // access using pointer-to-member operator
```

The two pointer accessors, `io.nonConstPtr()` and `io.ConstPtr()`, return pointers directly to the port's value. The former should be used when the port is being written; it sets the valid flag and returns the pointer. The latter should be used when the port is being read; it checks the valid flag and returns a const pointer to the value.

Note that all discussion of valid flags applies to debug builds only: in release builds all valid flag operations are optimized away, and no storage is allocated for valid flags.

3.5 Ports and update functions

When one update function writes a value to a port that is read by another update function, correct simulation depends on the update functions being invoked in the appropriate order. Cascade cannot deduce this order entirely on its own; it requires additional programmer input to specify which ports are read and written by a given update function. The syntax for doing so is to append the declarations `.reads(<ports>)` and/or `.writes(<ports>)` to an update function declaration. Multiple `reads` and `writes` declarations can be appended to a single update function declaration, prefixing each with a period. <*ports*> is a list of up to 8 ports; longer lists must be divided into multiple `reads`/`writes` declarations. The following constructor shows how to declare the ports which are read/written by the adder example update function:

```
Adder (COMPONENT_CTOR)
{
     UPDATE(update).reads(in_a, in_b).writes(out_sum);
}
```

The order in which the `reads`/`writes` declarations are provided is unimportant, as is the order of the ports. As an illustration of the use of multiple declarations, as well as a syntactically legal formatting style that may be preferable for longer lists, the above code is equivalent to the following:

```
Adder (COMPONENT_CTOR)
{
     UPDATE(update)
          .reads(in_a)
          .reads(in_b)
          .writes(out_sum);
}
```

It is also permitted to use multiple `UPDATE` declarations for the same update function, in which case the read/write lists are taken to be the union of the lists specified with each declaration. This is particularly useful for programmatically specifying the read/write lists. For example, if an update function reads every other input in an array and writes every other output in an array, then the read/write lists can be declared as follows:

```
for (int i = 0 ; i < NUM_PORTS ; i += 2)
    UPDATE(update_event).reads(in[i]).writes(out[i]);
```

In the common case, a component has a single update function which reads all input ports and writes all output ports. To simplify this case for the programmer, if the (default) `update()` function is defined but not explicitly declared, then when it is automatically registered it is made a reader of any `Input` ports with no other reader, and it is made a writer of any unconnected `Output` or `InOut` ports with no other writer. Thus, the sample Adder constructors, which illustrate read/write list declarations, are unnecessary: if the explicit declaration of `update()` is omitted then Cascade will assume that `update()` reads `in_a`, `in_b` and writes `out_sum`. Note that these assumptions are *only* made when there is no declaration for `update()`: the programmer can control the exact set of ports read/written by this function by declaring it explicitly and providing exact read/write lists.

The following simple example shows how read/write lists can be defined for multiple update functions:

```
class BidirectionalLink : public Component
{
    DECLARE_COMPONENT(BidirectionalLink, Link);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in_data1);
    Input(bit,  in_enable1);
    Input(int,  in_data2);
    Input(bit,  in_enable2);
    Output(int, out_data1);
    Output(int, out_data2);

    //----------------------------------------
    // Construction
    //----------------------------------------
    BidirectionalLink (COMPONENT_CTOR)
    {
        UPDATE(update1).reads(in_enable1, in_data1).writes(out_data1);
        UPDATE(update2).reads(in_enable2, in_data2).writes(out_data2);
    }

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update1 ()
    {
        out_data1 = in_enable1 ? in_data1 : 0;
    }
    void update2 ()
    {
        out_data2 = in_enable2 ? in_data2 : 0;
    }
};
```

It may seem that, in the above example, the explicit update declarations can be eliminated and the two update functions merged as follows:

```
void update ()
{
    out_data1 = in_enable1 ? in_data1 : 0;
    out_data2 = in_enable2 ? in_data2 : 0;
}
```

However, multiple update functions are sometimes necessary for Cascade to be able to find a permissible ordering. For example, if two `BidirectionalLink` components were connected to each other (see next section), then a single update function would fail as in this case each component would need to be updated be-

fore the other. Splitting the update function into two separate functions avoids this cyclic dependency. Another case in which multiple update functions are necessary is when a wrapper component sets the inputs and reads the outputs of a combinational subcomponent.

Example

```
class Wrapper : public Component
{
    DECLARE_COMPONENT(Wrapper);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in_a);
    Input(int,  in_b);
    Input(bit,  in_neg);
    Output(int, out);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Wrapper (COMPONENT_CTOR)
    {
        UPDATE(updateIn)
            .reads(in_a, in_b, in_neg)
            .writes(m_bb.in_a, m_bb.in_b);
        UPDATE(updateOut)
            .reads(m_bb.out)
            .writes(out);
    }

    //----------------------------------------
    // Simulation
    //----------------------------------------
    updateIn ()
    {
        m_bb.in_a = in_neg ? -in_a : in_a;
        m_bb.in_b = in_neg ? -in_b : in_b;
    }
    updateOut ()
    {
        out = in_neg ? -m_bb.out : m_bb.out;
    }

protected:
    BlackBox m_bb;
};
```

Within a reads/writes declaration, the special syntax `Inputs(<component>)`, `Outputs(<component>)`, `InOuts(<component>)` can be used to specify all Input, Output or InOut ports (respectively) of the specified component, where <*component*> is either a pointer or a reference to a component. For example, the above `Wrapper` constructor could be equivalently expressed as follows:

```
Wrapper (COMPONENT_CTOR)
{
    UPDATE(updateIn).reads(Inputs(this)).writes(Inputs(m_bb));
    UPDATE(updateOut).reads(Outputs(m_bb)).writes(Outputs(*this));
}
```

This example shows the use of both component pointers (`this`) and component references (`m_bb`, `*this`).

Both individual ports and port sets (above) can be specified within a single `reads`/ `writes` declaration, up to a maximum of 8 total arguments.

3.6 Combinational connections

Ports may be directly connected to one another at construction time using the overloaded left-shift operator. When two ports are connected they become synonyms: writes to one are immediately visible to reads from the other. Inputs and Outputs can be connected to any other port and become read-only, so that attempts to assign a connected Input/Output in debug builds will generate an error. InOuts can only be connected to other InOuts, and after a connection is made both InOuts can still be read or written. These restrictions reflect the various ways of connecting ports of two separate components or of a parent and sub-component.

The following example illustrates the use of connections to create a three input adder:

```
class Adder3 : public Component
{
    DECLARE_COMPONENT(Adder3);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in_a);
    Input(int,  in_b);
    Input(int,  in_c);
    Output(int, out_sum);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Adder3 (COMPONENT_CTOR)
    {
        m_adder1.in_a << in_a;
        m_adder1.in_b << in_b;
        m_adder2.in_a << m_adder1.out_sum;
        m_adder2.in_b << in_c;
        out_sum << m_adder2.out_sum;
    }

protected:
    //----------------------------------------
    // Subcomponents
    //----------------------------------------
    Adder m_adder1;
    Adder m_adder2;
};
```

Note that the order of sub-component updates is important as one of the inputs of `m_adder2` is connected to the output of `m_adder1`: Cascade uses the connection information in conjunction with read/write lists to ensure that the update functions are invoked in the correct order.

When ports are connected, the left shift operator points in the direction of signal propagation, so "*a* << *b*" is read as "*a* gets its input from *b*" or equivalently "*a* is connected to *b*". The order is important when connecting Inputs and Outputs as the port on the *left* becomes read-only. The order is also important when connections are chained, for example the statements

```
in_b << in_c;
out_a << in_b;
```

connect `in_b` to `in_c` and also `out_a` to `in_b`. Hence, all three ports will be part of the same netlist, but `in_b` and `out_a` will be read-only. Multiple connections may be specified on a single line: the above example can be equivalently expressed as

```
out_a << in_b << in_c;
```

3.6.1 Connecting ports to variables and constants

Ports may also be wired directly to variables or constants of the same type using the `wireTo()` and `wireToConst()` member functions; ports that are wired to variables or constants become read-only. For example, an increment component could be defined as follows:

```
class Increment : public Component
{
    DECLARE_COMPONENT(Increment);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in);
    Output(int, out);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Increment (COMPONENT_CTOR)
    {
        m_adder.in_a << in;
        m_adder.in_b.wireToConst(1);
        out << m_adder.out_sum;
    }

protected:
    Adder m_adder;
};
```

A fixed-delay FIFO could be defined as follows (this is not an efficient or recommended implementation, but it illustrates connecting an output to a variable):

```
template <typename T>
class FIFO3 : public Component
{
    DECLARE_COMPONENT(FIFO3, Q);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(T,  in);
    Output(T, out);

    //----------------------------------------
    // Construction
    //----------------------------------------
    FIFO3 (COMPONENT_CTOR)
    {
        out.wireTo(m_queue[2]);
    }

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ()
    {
        m_queue[2] = m_queue[1];
        m_queue[1] = m_queue[0];
        m_queue[0] = in;
    }

protected:
    T m_queue[3];
};
```

3.6.2 Restrictions on connections

The order in which connections are established is unimportant. The only restrictions are that an Input or Output can only be connected once, and a set of connected ports can be wired to at most one constant or variable. Thus, the following code would generate an error at construction time:

```
int m_size;
io_a.wireToConst(6);
io_b.wireTo(m_size);
io_c << io_a;        // OK
io_c << io_b;        // Error!! This would connect io_c to both m_size and 6.
```

3.7 Synchronous connections

A second connection operator, the overloaded less-than-or-equal-to operator, can be used to establish a synchronous connection between two ports. For this type of connection the producer and consumer ports are *not* synonyms: a value written to the producer port does not become visible to the consumer until the rising clock edge of the consumer's clock domain. In other words, a synchronous connection models a single stage of flip-flops between the producer and consumer ports, where the flip-flops are controlled by the consumer's clock. Internal registers can be modeled with the `Register` pseudo-port: combinational or synchronous connections may be established from a `Register` to an `Input`, `Output` or `InOut`, and synchronous connections may be established from a `Register`, `Input` or `Output` to another `Register`. The following example shows how synchronous connections can be used to construct a pipelined 3-input adder:

```
class PipelinedAdder3 : public Component
{
    DECLARE_COMPONENT(Adder3, PipedAdder);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(int,  in_a);
    Input(int,  in_b);
    Input(int,  in_c);
    Output(int, out_sum);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Adder3 (COMPONENT_CTOR)
    {
        m_adder1.in_a << in_a;
        m_adder1.in_b << in_b;
        m_adder2.in_a <= m_adder1.out_sum;
        m_adder2.in_b <= m_reg << in_c;
        out_sum << m_adder2.out_sum;
    }

protected:
    Adder m_adder1;
    Adder m_adder2;
    Register<int> m_reg;
};
```

The explicit register is unnecessary in this example as `m_adder2.in_b` could be synchronously connected directly to `in_c`, but it has been included for the purpose of illustration. As with ports, registers can be declared using either the macro `Register(type, name)` or the templated type `Register<type>`, and the macro can only be used within public sections of the class declaration.

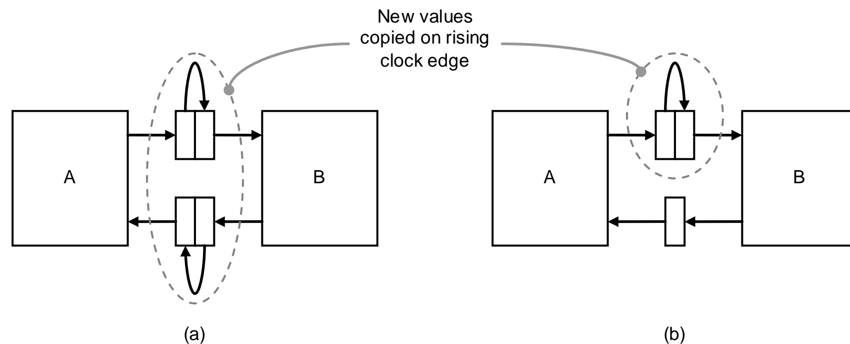
When a default `update()` function is not explicitly declared and is automatically recognized by Cascade, it is made a reader of any Registers with no other readers, and a writer of any unconnected Registers with no other writers.

As shown in the above example, multiple connections on the same line can include a mix of combinational and synchronous connections.

3.7.1 Register elimination

There is a small amount of overhead in the implementation of synchronous connections due to the need to maintain separate storage locations for the old and new values and to copy data between them (Figure 4a), but in some cases this overhead can be avoided. Consider two components A and B which are interconnected in a purely synchronous manner. If, in simulation, component A is always updated before component B, then only one storage location is needed for the synchronous signals from B to A since they are always read by A before being written by B (Figure 4b).

Figure 4. (a) Purely synchronous connections between A and B. (b) Register elimination from B to A; requires that `A::update()` is always called before `B::update()`.



More generally, a register can be eliminated whenever Cascade can prove that the first write of the D port always occurs *after* the last read of the Q port. For the most part, this is completely transparent to the programmer. However, because Cascade relies on read/write lists to determine whether or not the write-after-read condition holds, the programmer must take care to properly declare these lists as described in Section 3.5. Otherwise, the simulation may be incorrect, as indicated by port validation errors in debug builds.

3.7.2 Register merging

If two ports are in the same clock domain and have synchronous connections (or a chain of synchronous connections with the same depth) to a common source port, then Cascade will automatically merge these ports, using the same storage location for both of them. This reduces both storage and copy overhead, and is almost entirely transparent to the simulation. The one case where this optimization has a visible effect is at reset time: if the two ports are reset with different values (Section 3.10), then one of them will silently overwrite the other. Thus, the programmer should take care to ensure that ports in the same clock domain with synchronous connections to a common source port have the same reset value (if one is supplied at all).

3.7.3 Synchronous delay

A synchronous connection has the effect of placing a single register between the producer and consumer ports, so that values written to the producer port are visible to the consumer after exactly one consumer clock cycle. Larger delays can be modeled by calling the consumer's `setDelay()` member function:

```
void setDelay (int delay)
```

This has the effect of placing `delay` registers between the producer and consumer ports, so that values written to the producer port are visible to the consumer after exactly `delay` clock cycles in the consumer's clock domain. `setDelay()` can be called either before or after the synchronous connection is established; the delay argument must be ≥ 0 and the function call has no effect if delay is 0.

3.8 Fifo ports

The communication between two hardware components is often governed by some form of flow control. *Fifo ports* encapsulate this flow control, along with any associated queuing, so that the programmer does not need to explicitly implement the communication protocol at the lowest level. A fifo port models a fifo queue with delay: values are pushed onto the queue by a producer component and popped off the queue by a consumer component. Fifo ports can also be used to implement connections between clock domains.

There are two type of fifo ports: `FifoInput` and `FifoOutput`. Macros with the corresponding names are provided to declare fifo ports. As with standard ports, the fifo port classes can also be used directly when necessary (at the expense of losing the port names from debug messages); the class names are the same as the macro names. Also as with standard ports, the macros can only be used within public sections of the class declaration. The following example shows both methods of declaring a fifo port.

```
class Fetcher : public Component
{
    DECLARE_COMPONENT(Fetcher);
public:

    FifoInput(uint16,  in_addr);
    FifoOutput<uint32> out_data;
    …
```

3.8.1 Reading and writing fifo ports

Unlike standard ports, which can be directly assigned or implicitly converted to the appropriate type, fifo ports must be manipulated using a set of queue accessor functions. Flow control is implemented using four member functions which return the currently visible state of the queue:

`bool full () const`

Returns true if the queue is full, false otherwise. Called from the producer.

`bool empty () const`

Returns true if the queue is empty, false otherwise. Called from the consumer.

`int freeCount () const`

Returns the number of free queue entries visible to the producer.

`int popCount () const`

Returns the number of elements in the queue that are visible to the consumer.

The `full()` and `freeCount()` functions are used by a producer to determine whether or not a queue can be written; the `empty()` and `popCount()` functions are used by a consumer to determine whether or not a queue can be read. Note that these functions return the *visible* state of the queue which, for a queue with

delay (Section 3.8.3), will lag some number of cycles behind the actual state. For example, when a producer pushes a value onto an empty queue with N cycles of delay, the consumer's calls to `empty()` will continue to return `true` until N cycles in the future.

Communication across a fifo is implemented using the following three member functions (where T is the templated queue data type):

`void push (const T &data)`

Push a value onto the queue.

`const T &pop ()`

Pop a value from the head of queue. The reference is guaranteed to be valid until at least the next call to `push()`.

`const T &peek () const`

Retrieve the value at the head of the queue without popping it. The reference is guaranteed to be valid until at least the next call to `push()`.

An assertion failure is generated if `push()` is called when the queue is full, or if `pop()`/`peek()` are called when the queue is empty.

Example

```
class Fetcher : public Component
{
    DECLARE_COMPONENT(Fetcher);
public:
    Fetcher (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    FifoInput(uint16, in_addr);
    FifoOutput(int,   out_pos);
    FifoOutput(int,   out_neg);

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ()
    {
        if (!in_addr.empty())
        {
            uint16 addr = in_addr.peek();
            if (m_mem[addr] >= 0)
            {
                if (!out_pos.full())
                {
                    out_pos.push(m_mem[addr]);
                    in_addr.pop();
                }
            }
            else if (!out_neg.full())
            {
                out_neg.push(m_mem[addr]);
                in_addr.pop();
            }
        }
    }
};
```

An additional function can be used to return the maximum population count over the lifetime of the queue:

```
int highWaterMark () const
```

3.8.2 Connecting fifo ports

Fifo ports may be connected to other fifo ports using the same combinational and synchronous connection operators that are used for standard ports:

```
consumer.in << producer.out;  // Combinational connection (no delay)
consumer.in <= producer.out;  // Synchronous connection (one cycle of delay)
```

Fifo ports cannot be connected to standard ports (or vice versa), and fanout is not supported (so at most one connection can be made to any fifo port). However, multiple fifo ports can be connected in a chain:

```
consumer.in << repeater.out;
repeater.out <= repeater.in;
repeater.in << producer.out;
```

Such a chain of connections is modeled as a single queue. The head of the queue (`consumer.in`) is read-only, the tail of the queue (`producer.out`) is write-only, and intermediate fifo ports cannot be read or written. As with regular ports, chains of connections can also be specified with a single statement:

```
consumer.in << repeater.out <= repeater.in << producer.out;
```

3.8.3 Queue size and delay

Fifo queues are modeled as having both finite size and finite delay. As with an actual hardware queue, the size is the maximum number of data values that can be stored in the queue, and the delay is the amount of time (measured in cycles) that it takes for an action on one end of the queue to be visible on the other. Note that, if a queue is intended to support a sustained throughput of one data value per cycle, then the size of the queue must be at least twice the delay plus one. Otherwise, the queue will have insufficient buffering to support the full round trip delay, and it will experience stalls as a result. If a queue has size 0 (i.e. its size has not been specified), then Cascade automatically sizes the queue to twice the delay plus one. If the queue size has been specified but is less than this amount, then Cascade prints a warning message indicating that the queue size is too small to achieve full throughput. This warning can be disabled by setting the parameter `cascade.FifoSizeWarnings` to false.

Each fifo port constructor takes two optional arguments which can be used to specify the port size and the delay in cycles:

```
FifoPort (int size = 0, int delay = 0)
```

During the construction phase, the size and delay can also be set individually using two fifo port member functions:

```
void setSize (int size)
void setDelay (int delay)
```

If no delay is explicitly assigned (by one of the above two methods) but the fifo port is synchronously connected, then it has delay 1.

The following example illustrates the various ways in which fifo port sizes and delays can be specified.

```
class FifoExample : public Component
{
    DECLARE_COMPONENT(FifoExample, Fifos);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    FifoInput(int,  in1);
    FifoInput(int,  in2);
    FifoInput(int,  in3);
    FifoOutput(int, out1);
    FifoOutput(int, out2);

    //----------------------------------------
    // Construction
    //----------------------------------------
    FifoExample (COMPONENT_CTOR) : in1(2), in2(0, 3), out1(3, 4)
    {
        in1.setDelay(6);
        out2.setSize(3);
    }
};
```

When multiple fifo ports are connected to form a single queue, the size of the queue is the sum of the sizes specified for the individual ports. If no size has been specified for any of the ports, then the queue size is set to the minimum value required for full throughput. Similarly, the delay of the queue is the sum of the individual port delays. A port with delay *d* contributes *d* cycles of delay, measured in terms of the clock domain which contains the port. The resulting total delay in picoseconds is converted to a delay in consumer port clock cycles, rounding up if necessary.

3.8.4 Fifo port readers and writers

The rules for determining which update functions read/write which fifo ports are exactly the same as those described in Section 3.5 for standard ports. It is also possible to explicitly specify that a fifo port has no reader/writer using the member functions

```
void wireToZero ()
void sendToBitBucket ()
```

A fifo port that has been wired to zero has no writer and may not be connected to another fifo port. The resulting queue will always be empty: calls to `empty()` will always return `true`, and calls to `popCount()` will always return zero. Similarly, a fifo port that has been sent to the bit bucket has no reader, and other fifo ports cannot connect to it. Calls to `full()` will always return `false`, calls to `freeCount()` will always return 65535, and any data pushed onto the queue will vanish into the ether.

Unless a fifo has been wired to zero or sent to the bit bucket, it must have exactly one reader and exactly one writer. This is enforced during construction and initialization, and an error will be generated for any fifo that does not satisfy this condition.

3.8.5 Disabling flow control

Sometimes it is desirable to implement flow control explicitly, for example if multiple virtual channels share the same physical fifo. In these cases fifo ports still provide a convenient mechanism for modeling data paired with a valid signal or clock-domain boundary crossings, but their implicit flow control should be disabled during the construction phase by calling the member function

```
void disableFlowControl()
```

This function call has two effects:

1. The producer is no longer allowed to access the flow-control member functions: in Debug builds, calls to `full()` or `freeCount()` will generate an assertion failure.

2. Instead of requiring the size of the fifo to be twice the delay plus one, Cascade only requires the size to be at least the delay plus one. If the size is unspecified then it will automatically be set to this amount. If the size is specified but is too small, then an assertion failure is generated since there will be no way to prevent the queue from overflowing.

3.9 Port arrays

Statically sized single-dimensional port arrays can be declared using the port macros by specifying '<*name*>[<*size*>]' as the second argument. For example, an array of 64 input ports can be declared as follows:

```
Input(bit, in_enable[64]);
```

These ports will be given the names "in_enable[0]", "in_enable[1]", etc.

Dynamically sized single-dimensional arrays of ports or registers may be defined using the macros

```
InputArray(<type>, <name> [, <size>])
OutputArray(<type>, <name> [, <size>])
InOutArray(<type>, <name> [, <size>])
RegisterArray(<type>, <name> [, <size>])
FifoInputArray(<type>, <name> [, <size>])
FifoOutputArray(<type>, <name> [, <size>])
```

These macros must appear within a public section of the class declaration. As with the standard port macros, <*type*> and <*name*> specify the port type and name respectively. The third optional <*size*> argument specifies the size of the array. If supplied, this argument must be an expression which evaluates to an integer in a static class scope. That is, it is not required to be a compile-time constant, but it cannot involve non-static class member variables or component constructor arguments. The array size can be omitted from the array declaration and specified instead as the single constructor argument for the array; this allows the size to depend on component constructor arguments. Exactly one of these methods must be used to specify array size, so in particular if the size is specified within the declaration macro then the default (no arguments) constructor must be used for the array. The following example shows a component with two port arrays and illustrates both mechanisms for specifying array size.

```
class Router : public Component
{
    DECLARE_COMPONENT(Router);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    InputArray(int,  in_data,  NUM_INPUT_PORTS);
    OutputArray(int, out_data);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Router (int numOutputPorts, COMPONENT_CTOR) : out_data(numOutputPorts) {}
    …
};
```

Array elements are accessed in the standard manner using square brackets. In addition, the `size()` member function returns the size of the array.

Dynamically allocated ports are not allowed, so an array of ports *must* be created either using one of the above methods or by declaring a static array of the appropriate port class, as shown in the following example.

```
class ThreeArrays : public Component
{
    DECLARE_COMPONENT(ThreeArrays);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input<int> *in1;         // WRONG!!
    Input<int> in2[4];       // Correct
    InputArray(int, in3, 4); // Correct

    //----------------------------------------
    // Construction
    //----------------------------------------
    ThreeArrays (COMPONENT_CTOR) : in1(new Input<int>[4]) {}  // WRONG!!
};
```

Port arrays declared using the port or port array macros can be supplied to update function read/write lists, which is equivalent to supplying each port in the array individually.

Example

```
class FourArrays : public Commponent
{
    DECLARE_COMPONENT(FourArrays);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input      (int, in1[5]);
    InputArray (int, in2, 5);
    Output     (int, out1[5]);
    OutputArray(out, out2, 5);

    //----------------------------------------
    // Construction
    //----------------------------------------
    FourArrays (COMPNENT_CTOR)
    {
        // This UPDATE declaration is equivalent to:
        // for (int i = 0 ; i < 5 ; i++)
        //     UPDATE(update_arrays).reads(in1[i], in2[i]).writes(out1[i], out2[i]);
        UPDATE(update_arrays).reads(in1, in2).writes(out1, out2);

    }

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update_arrays ()
    {
        ...
    }
};
```

Multidimensional port arrays are not supported by the above macros, however they can still be declared directly as static multidimensional arrays of the appropriate port class with no port name information.

Example

```
Output<bit> out_request[4][6];
```

3.10 Reset

Output ports should generally be initialized within the component `reset()` function by calling the port `reset()` member function, passing the reset value as the single argument. There are two differences between the port `reset()` function and assignment. First, if a port has been wired to a constant then the `reset()` function does nothing, whereas assignment will silently overwrite the constant value in release builds producing very unexpected results (in debug builds, assignment to a constant will generate an assertion failure). Second, read-only ports (ports connected to another port or wired to a variable) can be safely initialized using the `reset()` function, whereas assignment would generate assertion failures in debug builds.

The consumer-side port in a synchronous connection does not need to be reset so long as the producer-side port is reset, because the initial rising clock edge will copy the producer's value to the consumer before any update function attempts to read the consumer. The producer-side port, on the other hand, should generally be reset for this reason.

Example

```
class DelayLines : public Component
{
     DECLARE_COMPONENT(DelayLines);
public:
     //----------------------------------------
     // Interface
     //----------------------------------------
     Input (int, in);
     Output(int, out0);
     Output(int, out1);
     Output(int, out2);

     //----------------------------------------
     // Construction
     //----------------------------------------
     DelayLines (COMPONENT_CTOR)
     {
          out0 << in;   // Don’t need to reset out0
          out1 <= in;   // Don’t need to reset out1
          reg  <= in;
          out2 <= reg;  // Need to reset reg using reset()
     }
     Register(int, reg);

     //----------------------------------------
     // Simulation
     //----------------------------------------
     void reset ()
     {
          reg.reset(0);
     }
};
```

3.10.1 Iterative reset

The reset value of one or more output ports may depend on the values of some input ports, which in turn obtain their value from another component's reset function. To ensure that all reset values are fully and properly propagated during reset, Cascade monitors the output ports of all components and iteratively resets the simulation until all output ports have quiesced.

The operation of this iterative reset is controlled by the integer parameter `cascade.MaxResetIterations`. If the output ports fail to quiesce after the specified maximum number of reset iterations, then an assertion failure is generated. This prevents an infinite loop if the reset behavior contains an unstable

feedback loop. In a simulation with a long chain of reset dependencies this parameter may need to be increased from its default value of 10.

In general, at least two iterations are required since the first iteration resets output ports (changing their values), necessitating a second iteration to detect quiescence. If `cascade.MaxResetIterations` is less than 2 then iterative reset is disabled. Iterative reset should only be disabled if the reset operation is particularly time consuming and it can be proven that a single iteration will produce correct results under the reset order described in Section 2.5.

Example

```
class Double : public Component
{
    DECLARE_COMPONENT(Double);
public:
    Double (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input<int>  in_x;
    Output<int> out_2x;

    //----------------------------------------
    // Reset
    //----------------------------------------
    void reset ()
    {
        out_2x = 2 * in_x;
    }
};

Double d1, d2;
d1.in_x.wireToConst(10);
d2.in_x << d1.out_2x;
Sim::init();
```

In the above example, the reset value of `d2.out_2x` will be 40 after either two or three reset iterations, depending on which of d1, d2 is reset first.

4.0 Clock Domains

A simulation is organized into disjoint *clock domains*, each of which has a clock whose period (time between rising clock edges) and *offset* (time of first rising clock edge) are measured in picoseconds. As time advances, the simulation is driven by the rising edges of the various clock domains. On a rising clock edge the following two actions are performed in order:

1. The actual rising clock edge is simulated; register values are copied from D ports to Q ports.
2. Combinational evaluation is simulated in zero time within the domain.

If multiple clock domains have a rising clock edge at the same time then Cascade first simulates the rising clock edge for each of these domains and then simulates combinational evaluation for each domain.

4.1 Defining clocks

Clock domains are associated with clocks. The simulation contains a single top-level implicit clock whose offset is zero and whose period is defined by the parameter `cascade.DefaultClockPeriod`. Components with no explicitly declared clocks are driven by this top-level clock. Explicit clocks are declared using the `Clock` pseudo-port macro:

```
class ComponentWithClock : public Component
{
     DECLARE_COMPONENT(ComponentWithClock);
public:
     ComponentWithClock (COMPONENT_CTOR) {}

     Clock(clk);   // Defines the clock for this component

     Input(int, in_enable);
     …
```

The Clock macro supplies the clock name to the Cascade infrastructure. As with ports, it is recommended that this macro be used, and it must appear within a public section of the class declaration. A clock can also be declared directly:

```
Clock clk;
```

No programmer action is required to associate a clock with a component beyond including the clock as a member; Cascade performs this association automatically. Clocks can be connected using the combinational connection operator << to form clock nets; these clock nets then define the clock domains of the simulation.

Example

```
class TwoComponents : public Component
{
     DECLARE_COMPONENT(TwoComponents);
public:
     TwoComponents (COMPONENT_CTOR)
     {
          m_comp1.clk << clk;
          m_comp2.clk << clk;
     }

     Clock(clk);

private:
     ComponentWithClock m_comp1;
     ComponentWithClock m_comp2;
};
```

Clocks cannot be connected to ports. Unlike ports, a Clock can appear at the top level and does not need to appear within a component, which allows the top-level clocks to be defined separately from the hardware being modeled. Top-level clocks must be declared directly; the `Clock()` macro can only be used within components.

Each clock net must have a single driver. A clock is declared as a driver using one of the following three member functions:

`void generateClock (int period = <default>, int offset = 0)`

Sets the clock period and offset (both in picoseconds) directly. If the period is not supplied then the parameter `cascade.DefaultClockPeriod` is used. If *offset* is non-negative, then it directly specifies the time of the first rising clock edge. If it is negative, then the first rising clock edge will be at time $offset + k \times period$, where k is the smallest integer for which this expression is non-negative.

`void divideClock (Clock &rhs, float ratio, int offset = 0)`

Sets the clock period and offset indirectly by multiplying the clock period of `rhs` by `ratio`, and adding `offset` to the clock offset of `rhs`. This results in a divided (slower) clock when `ratio` is greater than 1, and a multiplied (faster) clock when `ratio` is less than 1.

`void offsetClock (Clock &rhs, int offset)`

Sets the clock period and offset indirectly by using the clock period of `rhs` and adding `offset` to the clock offset of `rhs`. It has the same effect as `divideClock()` with `ratio` = 1.

When a clock is defined using `divideClock()` or `offsetClock()`, we refer to it as a *derived clock*.

A clock net can also be permanently disabled at construction time (preventing any calls to its components' update functions) using the member function

`void disable ()`

The following example shows a top-level 1GHz clock which is multiplied to 2GHz for a subcomponent.

```
class TwoComponents : public Component
{
    DECLARE_COMPONENT(TwoComponents);
public:
    TwoComponents (COMPONENT_CTOR)
    {
        m_comp1GHz.clk << clk;
        m_comp2GHz.clk.divideClock(clk, 0.5);
    }

    Clock(clk);

private:
    ComponentWithClock m_comp1GHz;
    ComponentWithClock m_comp2GHz;
};
…
Clock clk;
clk.generateClock(1000);
TwoComponents twoComponents;
twoComponents.clk << clk;
```

4.2 Timing of rising clock edges

As described above, the first simulated rising clock edge of a clock domain occurs at time *offset* + $k \times$ *period*, where k is the smallest non-negative integer such that this expression is non-negative. In particular, no rising clock edges are simulated at negative times, and if *offset* is larger than *period* then the effect will be a clock that is initially quiescent and becomes active at time *offset*.

During the simulation, rising clock edge times that are close to (by default, within 5 picoseconds of) an even number of nanoseconds will be rounded to this time. For example, a 1.5GHz clock can be modeled by using a period of 667 picoseconds. The rising clock edges for this clock will occur at the following simulation times (in picoseconds):

0, 667, 1334, 2000, 2667, 3334, 4000, …

The rising edge that would have occurred at 2001 picoseconds is rounded to 2 nanoseconds, and the subsequent edge occurs at the rounded time plus the clock period. Thus, for this example, there will be exactly 3 rising clock edges every 2 nanoseconds of simulated time.

The amount of clock rounding is controlled by the `cascade.ClockRounding` parameter, whose default value is 5. Setting this parameter to zero disables clock rounding; with clock rounding disabled the rising clock edges for the above example would occur at the following simulation times (in picoseconds):

0, 667, 1334, 2001, 2668, 3335, 4002, …

When a simulation is run until time T, all rising clock edges that occur before T are simulated. Rising clock edges that occur exactly at T are not simulated.

Irrespective of clock rounding, when a clock domain C_1 is derived from another domain C_0 (using `divideClock()` or `offsetClock()`) with a rational clock period ratio a/b, then every b^{th} rising edge of C_1 will be synchronized with every a^{th} rising edge of C_0 (including an appropriate fixed offset if necessary).

4.3 Multiple clocks

If a component contains multiple clocks, then Cascade is unable to automatically determine which clock should drive that component's update functions. This ambiguity can be resolved in one of two ways. The first method is to call the following clock member function for one of the clocks from the component constructor:

```
void setAsDefault ()
```

The selected clock then becomes the default clock for the containing component. It is an error to call this function for two clocks contained in the same component. The second method is to explicitly associate one of the clocks with an update function by appending `.clock(`*clock*`)` to the update function declaration. A combination of these two methods can be used within a single component, as shown in the following example:

```
class MultipleClocks : public Component
{
    DECLARE_COMPONENT(MultipleClocks);
public:
    MultipleClocks (COMPONENT_CTOR)
    {
        clk400.setAsDefault();
        UPDATE(update800)
            .reads(in800)
            .writes(out800)
            .clock(clk800);
    }

    Clock(clk400);
    Clock(clk800);

    void update400 ();
    void update800 ();
    …
};
```

4.4 Manual clocks

While the typical usage of a clock is to rely on its period to automatically generate a sequence of evenly-spaced rising clock edges, it is also possible to create a manually-controlled clock by calling the clock's `setManual()` member function at construction time. Rising clock edges can then be simulated at any point by calling the clock's `tick()` member function in between calls to `Sim::run()`, or from a component's `tick()` function (but not from an update function). `setManual()` is mutually exclusive with the functions `generateClock()`, `divideClock()`, `offsetClock()`, and `disable()`.

Example

```
class ManualClock : public Component
{
    DECLARE_COMPONENT(ManualClock);
public:
    ManualClock (COMPONENT_CTOR)
    {
        clk.setManual();
    }

    Clock(clk);

    void update ();
};
…
ManualClock mc;
for (int i = 0 ; i < 20 ; i++)
{
    mc.clk.tick();
    Sim::run(1000);
}
```

The above example is almost identical to calling `clk.setPeriod(1000)` and then running for 20 ns, with one small difference. If multiple automatic clocks have a simultaneous rising clock edge, then these edges are combined, i.e. the rising clock edge is simulated for each of the clock domains followed by combinational evaluation for each of the domains. A manual rising clock edge, on the other hand, is never combined with any other rising clock edges that are not derived from the manual clock.

A manual clock can be supplied as the argument to either `divideClock()` or `offsetClock()`. Cascade guarantees the appropriate ratio of rising clock edges between the manual clock and the derived clock by automatically simulating zero or more derived clock edges on each call to the manual clock's `tick()` function.

The global variable `Sim::simTime` is set to the appropriate time for each of these rising clock edges, with the result that the actual order of clock edges between manual and automatic clock domains may not match the chronological order implied by the simulation time. For example, consider the following:

```
Clock clk, clk_div, clk_manual, clk_manual_div;
clk.generateClock(1000);
clk_manual.setManual();
clk_div.divideClock(clk, 0.333);
clk_manual_div.divideClock(clk_manual, 0.333);

for (i = 0 ; i < 20 ; i++)
{
    clk_manual.tick();
    Sim::run(1000);
}
```

Both `clk` and `clk_manual` are 1GHz clocks with rising clock edges at simulation times 0 ps, 1000 ps, 2000 ps, 3000 ps, etc. Both `clk_div` and `clk_manual_div` are 3GHz clocks with rising clock edges at simulation times 0 ps, 333 ps, 666 ps, 1000 ps, etc. However, the rising clock edges for `clk_manual_div` only occur as a result of calls to `clk_manual.tick()`, so the actual order of simulated rising clock edges is as follows:

```
clk_manual_tick();
    // 0ps:    rising clock edge for clk_manual and clk_manual_div
Sim::run(1000);
    // 0ps:    rising clock edge for clk and clk_div
    // 333ps:  rising clock edge for clk_div
    // 666ps:  rising clock edge for clk_div
clk_manual_tick();
    // 333ps:  rising clock edge for clk_manual_div
    // 666ps:  rising clock edge for clk_manual_div
    // 1000ps: rising clock edge for clk_manual and clk_manual_div
Sim::run(1000);
    // 1000ps: rising clock edge for clk and clk_div
    // 1333ps: rising clock edge for clk_div
    // 1666ps: rising clock edge for clk_div
```

When a derived clock D is generated from a manual clock C using ratio r and offset m, then the following rules are used to generate the rising clock edges for D when C is ticked for the nth time at simulation time t:

1. If $n = 1$, then the absolute offsets m_C and m_D of C and D (respectively) are set to $m_C = t$ and $m_D = t + m$. If $0 \le m_D \le t$ then a rising clock edge of D is simulated at time m_D.

2. If $n > 1$, then the period p_C of C is assumed to be $p_C = (t - m_C)/(n - 1)$, and the period of p_D of D is assumed to be $r \times p_C$. If D has already had n_D rising clock edges, then additional rising clock edges are simulated at times

   $$n_D \times p_D + m_D,\quad (n_D + 1) \times p_D + m_D,\quad (n_D + 2) \times p_D + m_{D, \ldots}$$

   starting with the first non-negative time in this sequence and ending with the last time in the sequence less than or equal to t.

4.5 Assigning components, ports and update functions to clock domains

If a component has no clocks then it inherits the default clock (if any) of its parent; top-level components with no clocks are assigned to the top-level implicit clock domain. If a component has a single clock, than that clock becomes the component's default clock. If a component has multiple clocks, then one the these clocks can be specified as the default clock by calling its `setAsDefault()` member

function within the component constructor. Finally, if a component has multiple clocks and does not call `setAsDefault()` for any of them, then that component has no default clock.

If an update function is not explicitly assigned to a clock via `.clock(`*clock*`)`, then it is assigned to the default clock of the containing component. An error is generated at initialization time if the containing component has no default clock (as determined by the rules in the previous paragraph).

If a port is synchronously connected and has one or more readers (as defined in Section 3.5), then these readers must all be in the same clock domain, and the port is assigned to that clock domain. If a port is not synchronously connected and has one or more writers, then these writers must all be in the same clock domain, and the port is assigned to that clock domain. In all other cases the port is assigned to the default clock domain of the containing component; an error is generated at initialization time if the containing component has no default clock.

4.6 Connections and clock domains

Within a single clock domain it is possible to properly model a combinational path from one component to another by controlling the order of the updates. Between clock domains, however, it is not possible to similarly control the update order when the clock domains have a simultaneous rising edge. Thus, pure combinational connections are not allowed between clock domains that have any simultaneous rising edges (with the exception of ports wired to a constant).

The determination of whether or not two clock domains have any simultaneous rising edges is complicated by both clock rounding (Section 4.2) and manual clocks (Section 4.4). A manual clock domain is assumed to potentially have a simultaneous rising edge with any other domain, so cross-domain combinational connections are never allowed when one of the domains is a manual clock domain. For two automatic clock domains, Cascade uses a conservative heuristic to decide if they might have a simultaneous rising clock edge. First, “rounded GCD” of the two clock periods is computed using the following modified GCD algorithm:

```
int rounded_gcd (int a, int b)
{
    if (a > b) return rounded_gcd(b, a);
    if (a <= cascade.ClockRounding) return b;
    return rounded_gcd(a, b-a);
}
```

Next, a “minimal relative offset” between the domains is computed by taking the difference of their offsets, then adding (or subtracting) a multiple of the GCD to obtain a number in (-GCD/2, GCD/2]. Finally, the two clock domains are assumed to have a simultaneous rising clock edge if the absolute value of this minimal relative offset is less than or equal to `cascade.ClockRounding`.

4.7 Helper functions

Two component member functions can be called from update functions:

`int getClockPeriod () const`

Returns the clock period of the component’s clock domain in picoseconds.

`int getTickCount () const`

Returns the number of rising clock edges that have occurred in the component's clock domain, including the rising clock edge that just occurred.

4.8 Custom rising clock edge behaviour

Custom behavior on a rising clock edge can be implemented using the component `tick()` function. A clock domain calls `tick()` for all of its active components that define this function before synchronous state is updated, and before any calls to component update functions. `tick()` is not called for deactivated components. A `tick()` function is allowed to read ports and registers, but it should *not* set any normal or pulse ports/registers since these are invalidated or reset before the component update functions are called.

Example

```
class DFlipFlop : public Component
{
    DECLARE_COMPONENT(DFlipFlop);
public:
    DFlipFlop (COMPONENT_CTOR) {}

    // Interface
    Input(bit, D);
    Output(bit, Q);

    // Simulation
    void tick ()
    {
        m_temp = D;
    }
    void update ()
    {
        Q = m_temp;
    }

private:
    bit m_temp;
};
```

4.9 Manually scheduled events

Arbitrary events can be scheduled for some number of clock cycles in the future by calling a component's `scheduleEvent()` member function from an update function, from `reset()`, or from `tick()`:

```
void scheduleEvent (int delay, fn_t fn)
void scheduleEvent (int delay, fn_t fn, A1 a1)
void scheduleEvent (int delay, fn_t fn, A1 a1, A2 a2)
void scheduleEvent (int delay, fn_t fn, A1 a1, A2 a2, A3 a3)
void scheduleEvent (int delay, fn_t fn, A1 a1, A2 a2, A3 a3, A4 a4)
```

`delay` specifies the delay until the event is executed, measured in rising clock edges of the current clock domain. `delay` must be greater than zero; if it is 1 then the event is executed on the next rising clock edge of the current clock domain.

`fn` specifies the function to call when the event is executed. `fn` must be a member function of the component for which `scheduleEvent()` is called, and it can take up to four arguments.

The remaining optional arguments (`a1`, `a2`, `a3`, `a4`) specify up to four arguments that are passed to the function `fn` when the event is executed.

Scheduled events are executed on the specified rising clock edge after synchronous state has been updated (i.e. register values have been copied from D ports to Q ports) and before any component update functions are called.

An event callback function must be explicitly declared within the component constructor using the `DECLARE_EVENT(fn)` macro. Furthermore, any ports written by the callback function must be declared using one or more `.writes()` declarations following `DECLARE_EVENT()`; these declarations have the same format as for update functions as described in Section 3.5. Failure to include the appropriate `.writes()` declarations could result in one or more registers being erroneously eliminated.

The callback function can read ports that are synchronously connected or wired to constants, but it cannot read any combinationally connected ports since the component update functions have not yet been called so these ports will be invalid.

Important Note: The delay of a scheduled event is always measured in rising clock edges of the *current* clock domain. Thus, if a component A has a pointer to a component B in a different clock domain and it uses this pointer within `A::update()` to schedule an event within B, the delay will be measured in rising clock edges of A's clock domain.

Example

```
class RemoteMemory : public Component
{
    DECLARE_COMPONENT(RemoteMemory);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(bit,     in_read);
    Input(uint32,  in_addr);
    Output(bit,    out_resp);
    Output(uint32, out_data);

    //----------------------------------------
    // Construction
    //----------------------------------------
    Client (COMPONENT_CTOR)
    {
        DECLARE_EVENT(readResponse).writes(out_resp, out_data);
        out_valid.setType(PORT_PULSE);
    }

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ();
    void readResponse (uint32 data);

    …
};

void RemoteMemory::update ()
{
    if (in_read)
    {
        uint32 data = readData(in_addr);
        scheduleEvent(REMOTE_READ_LATENCY, &RemoteMemory::readResponse, data);
    }
}

void RemoteMemory::readResponse (uint32 data)
{
    out_resp = 1;
    out_data = data;
}
```

5.0 Interfaces

A C++ interface allows object implementations to be dynamically chosen at run-time by defining a set of pure virtual functions that form the contract between clients and implementations. Similarly, a hardware interface permits *component* implementations to be selected at run-time by defining the inputs and outputs of the component. This allows the system to make connections to the component at construction time without any knowledge of the actual component implementation, which could be a high-level functional implementation, a detailed cycle-based implementation, or even a wrapper around an RTL implementation. Interfaces also provide a convenient mechanism for grouping and reusing related sets of ports.

A Cascade interface is a structure which contains definitions for inputs and outputs. It must inherit (directly or indirectly) from `Interface`, and it must begin with the `DECLARE_INTERFACE(interface [, name])` macro. This macro is similar to `DECLARE_COMPONENT`; the first argument is the structure name and the second argument is an optional human-readable name. An interface should implement `reset()` to initialize outputs, and if two interfaces are designed to be directly connected then it may be convenient for one of them to define a connection operator. As with components, the `reset()` function can optionally take an integer argument specifying the reset level, and it is called automatically by Cascade. The following example shows sample interfaces for an SRAM and its client:

```
// SRAM interface
struct ISRAM : public Interface
{
     DECLARE_INTERFACE(ISRAM);

     Input(bit,  in_valid);
     Input(bit,  in_write);
     Input(u12,  in_address);
     Input(u32,  in_data);

     Output(bit, out_valid);
     Output(u32, out_data);

     void reset () { out_valid = 0; }
};

// SRAM client interface
struct ISRAMClient : public Interface
{
     DECLARE_INTERFACE(ISRAMClient, IClient);

     Output(bit, out_valid);
     Output(bit, out_write);
     Output(u12, out_address);
     Output(u32, out_data;

     Input(bit,  in_valid);
     Input(u32,  in_data);

     void reset () { out_valid = 0; }

     void operator<< (struct ISRAM &sram)
     {
          // Submit the SRAM request on the same cycle
          sram.in_valid << out_valid;
          sram.in_write << out_write;
          sram.in_address << out_address;
          sram.in_data << out_data;

          // ... and get the results on the following cycle
          in_valid <= sram.out_valid;
          in_data <= sram.out_data;
     }
};
```

5.1 Implementing an interface

A component can implement an interface through either *inheritance* or *containment*. Implementation via inheritance is slightly easier to work with as a pointer to the component can be cast directly to a pointer to the interface, and since the individual inputs/outputs are direct members of the component it is not necessary to qualify them with a structure name. A component can inherit from multiple distinct interfaces. Implementation via containment has the advantages of explicitly declaring the interface and avoiding naming collisions (for example, if a component needs to expose two instances of the same interface, then containment is the only option). Both components and interfaces can inherit from multiple interfaces, but `Component` (or some other component base class) must always appear first in the list of a component's base classes.

The following example shows an `SRAM` component that implements `ISRAM` via inheritance, and a `Processor` component that contains two `ISRAMClient` interfaces:

```
//----------------------------------------
// SRAM
//----------------------------------------
class SRAM : public Component, public ISRAM
{
    DECLARE_COMPONENT(SRAM);
public:
    SRAM (COMPONENT_CTOR) {}

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ()
    {
        if (in_valid)
        {
            if (in_write)
                m_data[in_address] = in_data;
            else
                out_data = m_data[in_address];
        }
        out_valid = in_valid && !in_write;
    }

protected:
    u32 m_data[0x1000];
};

//----------------------------------------
// Processor
//----------------------------------------
class Processor : public Component
{
    DECLARE_COMPONENT(Processor, Proc);
public:
    Processor (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    ISRAMClient io_sram[2];
    …
};
```

Ports names are scoped by the interface(s) in which they are contained (there may be a chain of interfaces if some interfaces contain or inherit from others). The following are some sample port names from the above example:

```
SRAM.ISRAM.in_valid
SRAM.ISRAM.out_data
Proc.IClient0.out_address
Proc.IClient1.in_data
```

5.2 Instantiating an interface implementation

When there is only a single implementation of a component, parent components can contain the implementing class directly and make connections to its inputs and outputs. In order to support the ability to dynamically choose the implementing class at run time, the parent component should instead contain a pointer to the appropriate interface, which it can use to establish connections. Again, this exactly parallels the standard technique of using pointers to C++ interfaces and calling virtual functions.

The following example shows a `ProcMemNode` component which contains a Processor and two generic SRAMs. The function `ConstructSRAM()` is assumed to create a component which implements `ISRAM` and returns an `ISRAM` pointer (for example, it could create the `SRAM` component from the previous example).

```
class ProcMemNode : public Component
{
    DECLARE_COMPONENT(ProcMemNode, Node);
public:
    //----------------------------------------
    // Construction
    //----------------------------------------
    ProcMemNode (COMPONENT_CTOR)
    {
        // Create the SRAM components – can create anything
        // that supplies the appropriate interface
        m_sram[0] = ConstructSRAM();
        m_sram[1] = ConstructSRAM();

        // Connect the components using interface pointers
        m_processor.io_sram[0] << *isram[0];
        m_processor.io_sram[1] << *isram[1];
    }

    ~ProcMemNode()
    {
        delete m_sram[0];
        delete m_sram[1];
    }

protected:
    Processor m_processor;
    ISRAM    *m_sram[2];
    …
};
```

5.3 Extending interfaces

As with components, an interface can both inherit from and contain multiple interfaces.

Example

```
struct INetworkPort : public IDebugPort, IScanPort
{
    DECLARE_INTERFACE(INetworkPort);

    INetworkInput  net_in;
    INetworkOutput net_out;
};
```

5.4 Custom constructors

Interfaces typically do not require custom constructors, so for convenience one is supplied by the `DECLARE_INTERFACE` macro. In order to specify one or more custom constructors for an interface, use `DECLARE_INTERFACE_WITH_CTOR` instead of `DECLARE_INTERFACE`; this macro takes the same two arguments but

does not supply a default constructor. Similar to components, the last argument of every constructor must be INTERFACE_CTOR. Where a constructor is declared and implemented separately, the declaration should use the macro INTERFACE_CTOR while the implementation should use the macro IMPL_CTOR.

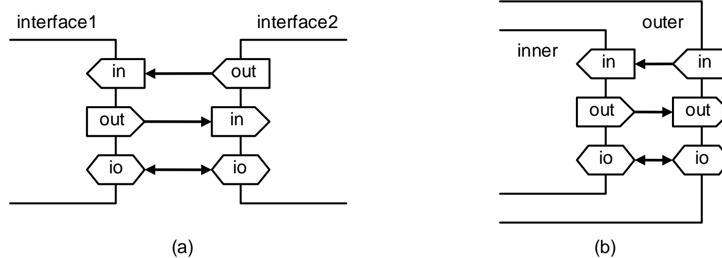
Example

```
struct ICreditOutput : public Interface
{
    DECLARE_INTERFACE_WITH_CTOR(ICreditOutput, ICreditOut);

    Output(int, out_initial_credits);
    Output(bit, out_credit);

    //----------------------------------------
    // Construction
    //----------------------------------------
    ICreditOutput (int initialCredits = 8, INTERFACE_CTOR);
};

ICreditOutput::ICreditOutput (int initialCredits, IMPL_CTOR)
{
    out_initial_credits.wireToConst(initialCredits);
}
```

5.5 Connecting interfaces

As a convenience, functions are provided for connecting two entire interfaces rather than having to connect each pair of ports individually. Two complementary interfaces can be *connected*, which connects the inputs of one to the outputs of the other; two identical interfaces can be *chained*, which connects the inputs of an inner interface to the inputs of an outer interface and/or the outputs of an outer interface to the outputs of an inner interface. These connections are illustrated in Figure 5.

Figure 5. (a) Connected interfaces. (b) Chained interfaces.



The functions used to connect interfaces are as follows:

```
void connect     (I1 &i1, I2 &i2)
void syncConnect (I1 &i1, I2 &i2, int delay = 1)
void chain       (I1 &inner, I2 &outer)
void syncChain   (I1 &inner, I2 &outer, int delay = 1)
```

The connect() and chain() functions establish combinational connections between pairs of ports; the syncConnect() and syncChain() functions establish synchronous connections with an arbitrary delay. Specifying a delay greater than 1 has the same effect as calling setDelay(<delay>) for each consumer port (Section 3.7.3).

The ports of the interfaces being connected must match exactly, i.e. they must appear in the same order and have the same types. Clocks are not connected by

these functions, and InOuts are ignored by the two synchronous connection functions. Note that the order of the interface arguments does not matter for `connect()` and `syncConnect()` but it *does* matter for both `chain()` and `syncChain()`, since the direction of the connections depends on which interface is the inner interface and which is the outer interface.

To connect only the inputs of an interface (`Input` and `FifoInput` ports), pass `Inputs(<interface>)` as an argument to one of the above connection functions, where *<interface>* is a pointer or a reference to an interface. Similarly, pass `Outputs(<interface>)` or `InOuts(<interface>)` to connect only the outputs (`Output` and `FifoOutput` ports) or `InOuts` respectively. For example, the `ISRAMClient` connection operator could be defined as follows:

```
void operator<< (struct ISRAM &sram)
{
    connect(Inputs(sram), Outputs(this));
    syncConnect(Input(this), Outputs(sram));
}
```

Components can also be used in place of interfaces in the above functions.

5.6 Reads/writes declarations

The `Inputs()`, `Outputs()` and `InOuts()` expressions can also be passed as arguments to `reads()` or `writes()` when declaring update function read/write lists. Again, the *interface* argument can be a pointer or reference to an interface or component.

5.7 Retrieving the component pointer

Every interface belongs to a single component. A pointer to this component can be obtained by calling the interface's `getComponent()` function:

```
Component *getComponent () const
```

5.8 Reset order

Both components and interfaces may inherit from and contain multiple interfaces, each of which may define their own reset function. The order in which these functions are called when the simulation is reset is as follows:

1. For a component that inherits from a base component, the base component is reset first.
2. Base interfaces are reset in the order that they appear in the inheritance list.
3. The current component/interface is reset.
4. Member components (for a component only) and member interfaces are reset in the order that they are declared.

6.0 Arrays

Cascade provides a templated `Array` class which allows arrays of components and interfaces to be named and accessed in a natural manner. Arrays are declared without dimensions as shown in the following examples:

```
Array<ISRAMClient> io_sram;
Array<Adder>       m_adders;
```

The dimensions of an array are supplied to the array's constructor. The following `Array` constructor can be used for arrays of components and interfaces with default constructors:

```
Array (int sizeX [, int sizeY [, int sizeZ]])
```

This declares an *anonymous* array. Array elements in an anonymous array have names of the form:

```
<parent name>.<component/interface name>(<index>[,<index>[,<index>]])
```

For a one dimensional array the `sizeY` and `sizeZ` arguments are omitted; for a two dimensional array the `sizeZ` argument is omitted.

Array elements can be accessed using the overloaded function call operator with array indices supplied as arguments, e.g.

```
io_sram.out_data(1,2) = m_adders(0,1).out_sum;
```

The elements can also be accessed using square brackets with a single index ranging from `0` to `N-1`, where `N = sizeX * sizeY * sizeZ` is the total number of elements in the array and is returned by the `size()` member function. The linear index of element (x, y, z) in the array is

$$(z * sizeY + y) * sizeX + x$$

6.1 Named Arrays

A second form of the `Array` constructor is used to declare a *named array*:

```
Array (const char *name, int sizeX [, int sizeY [, int sizeZ ]])
```

The first constructor argument specifies the array name. The elements of a named array have names of the form:

```
<parent name>.<array name>(<index>[,<index>[,<index>]])
```

Named arrays are useful for disambiguating two different arrays of the same type within the same component. The following simple example shows a component with two named interface arrays:

```
class Processor : public Component
{
    DECLARE_COMPONENT(Processor);
public:
    Array<ISRAM> io_iram;
    Array<ISRAM> io_dram;

    Processor (COMPONENT_CTOR) :
        io_iram("iram", NUM_BANKS),
        io_dram("dram", NUM_BANKS)
    {}
};
```

6.2 Element names

A third naming option also exists, which is to give each element of the array a distinct name. The following constructor is provided for this purpose:

```
Array (const char **names, int sizeX, int sizeY = 1, int sizeZ = 1)
```

The `names` argument points to an array of names whose size must match that of the interface array. The array elements are given names of the form

```
<parent name>.<element name>
```

where, for the kth element, `<element name>` is `names[k]`.

Example

```
const char *clientNames[4] = {"proc0", "proc1", "mem0", "mem1"};

class DataBus : public Component
{
    DECLARE_COMPONENT(DataBus);
public:
    DataBus (COMPONENT_CTOR) : clients(clientNames, 4) {}

    Array<IBusClient> clients;
};
```

6.3 Arrays with custom allocators

For components and interfaces with a non-default constructor, an `Array` has the following templated constructors for one, two and three dimensional anonymous and named arrays (`Allocator` is a template type):

```
Array (int sizeX, Allocator *allocator)
Array (const char *name, int sizeX, Allocator *allocator)
Array (const char **names, int sizeX, Allocator *allocator)
Array (int sizeX, int sizeY, Allocator *allocator)
Array (const char *name, int sizeX, int sizeY, Allocator *allocator)
Array (const char **names, int sizeX, int sizeY, Allocator *allocator)
Array (int sizeX, int sizeY, int sizeZ, Allocator *allocator)
Array (const char *name, int sizeX, int sizeY, int sizeZ, Allocator *allocator)
Array (const char **names, int sizeX, int sizeY, int sizeZ, Allocator *allocator)
```

The `Allocator` object must implement one of the following three functions, depending on the dimension of the array, where `T` is the templated array type:

```
void allocateElement (T *&element, int x)
void allocateElement (T *&element, int x, int y)
void allocateElement (T *&element, int x, int y, int z)
```

The `allocateElement()` function will be called to allocate each element in the array, and is supplied with the element's coordinates. This function must assign the element to `element`. The allocator object must contain any additional data required to construct the array elements (and may choose to ignore the coordinates).

Two helper macros are provided to handle the common cases where the element constructors take either the element's coordinates or a single constant argument. To construct each element by passing the coordinates to its constructor, use the macro `ARRAY_INDEX_ALLOCATOR` for the allocator argument.

Example

```
class Node : public Component
{
     DECLARE_COMPONENT(Node);
public:
     Node (int x, int y, COMPONENT_CTOR) : m_x(x), m_y(y) {}
     ...
private:
     int m_x;
     int m_y;
};

class Grid : public Component
{
     DECLARE_COMPONENT(Grid);
public:
     Grid (int xsize, int ysize) : m_nodes(xsize, ysize, ARRAY_INDEX_ALLOCATOR) {}
     ...
private:
     Array<Node> m_nodes;
};
```

To construct the elements by passing the same fixed argument to each element constructor, use the macro

```
ARRAY_ARG_ALLOCATOR(<type>, <value>)
```

where *<type>* is the type of the argument and *<value>* is its value.

Example

```
class Fifo : public Component
{
     DECLARE_COMPONENT(Fifo);
public:
     Fifo (int depth, COMPONENT_CTOR)
     ...
};

class Router : public Component
{
     DECLARE_COMPONENT(Router);
public:
     Router (COMPONENT_CTOR)
     : m_inputFifos(NUM_PORTS, ARRAY_ARG_ALLOCATOR(int, ROUTER_INPUT_FIFO_DEPTH))
     ...

private:
     Array<Fifo> m_inputFifos;
};
```

6.4 Dynamic allocation of interfaces

Cascade does not allow dynamic allocation of interfaces, so for example the `ISend` array in the following example is incorrect and will generate an error at construction time. The `IRecv` array is correct; it uses an `Array` to create the array of interfaces.

```
class Master : public Component
{
    DECLARE_COMPONENT(Master);
public:
    //----------------------------------------
    // Interface
    //----------------------------------------
    ISend *io_send;                         // WRONG!!
    Array<IRecv> io_recv;

    //----------------------------------------
    // Construction
    //----------------------------------------
    Master(int numClients, COMPONENT_CTOR) :
        io_send(new ISend[numClients]),  // WRONG!!
        io_recv(numClients),             // Correct
        m_numClients(numClients)
    {}
    …
};
```

This restriction does *not* apply to components, so arrays of subcomponents can be dynamically allocated if none of the `Array` features are desired.

Example

```
class ManyAdders : public Component
{
    DECLARE_COMPONENT(ManyAdders);
public:
    ManyAdders (COMPONENT_CTOR) : m_adders(new Adder[NUM_ADDERS]) {}

private:
    Adder *m_adders;
};
```

6.5 Calling a function on all array elements

For convenience, the `Array` class provides a `doAcross()` function for calling the same function on all array elements:

```
void doAcross (void (T::*func) (<args>) [, <args>])
```

The function `func` that is called can take 0–4 arguments; if it takes arguments then they must be supplied to `doAcross()`, and the same arguments will be used for each function call.

Example

```
m_adders.doAcross(&Adder::clearOverflow);
m_adders.doAcross(&Adder::setCarryBit, 0);
```

7.0 Activation and Triggers

By default, each update function is called on every clock cycle. This can be a significant source of inefficiency for simulations containing components that are only active a small fraction of the total time. In order to improve the performance of such simulations, Cascade provides the programmer with a mechanism for dynamically activating and deactivating components. The following component member functions are used to change or query the active/inactive state of a component:

```
void activate ()
void deactivate ()
bool isActive () const
```

When a component is inactive, its update function(s) (and its `tick()` function, if defined) are not called. In addition to the above functions, two mechanisms are provided to automatically activate a component when input data becomes available. First, if the component has a fifo input, then the component is activated whenever data pushed onto the fifo becomes visible. Second, any port can be declared as activating any component using the port member function

```
void activates (Component *target,
                PortActivationType activationType = ACTIVE_HIGH)
```

This function, which must be called during the construction phase, takes two arguments. The first is the component to activate (note that a single port can activate multiple components). The second optionally indicates the activation type. By default, the port will activate the component whenever it has a non-zero value. If the activation type is specified as `ACTIVE_LOW`, then the port will activate the component whenever it is zero.

The following example shows a simple ALU that uses activation:

```
class ALU : public Component
{
    DECLARE_COMPONENT(ALU);
public:
    enum { OP_NONE, OP_AND, OP_OR, OP_ADD, OP_SUB };

    ALU (COMPONENT_CTOR)
    {
        in_op.activates(this);
    }
    void reset ()
    {
        deactivate(); // Start in the deactivated state
    }

    Input(byte, in_op);
    Input(int,  in_a);
    Input(int,  in_b);
    Output(int, out);

    void update ()
    {
        out = (in_op == OP_AND) ? (in_a & in_b) :
              (in_op == OP_OR)  ? (in_a | in_b) :
              (in_op == OP_ADD) ? (in_a + in_b) :
                                  (in_a - in_b);

        // Everything happens in a single cycle, so deactivate immediately
        deactivate();
    }
};
```

As shown in the example, a component can be made initially inactive by calling `deactivate()` from within `reset()`. In this example the component unconditionally deactivates on every clock cycle; in general the programmer must ensure that the component is truly idle before calling `deactivate()`. In particular, this includes verifying that all input fifos are empty. Otherwise a situation could arise where some input data is never consumed because the component was prematurely deactivated; Cascade will not activate the component if additional data is pushed onto the fifo since, as an optimization, components are only activated when data is pushed onto an *empty* fifo. While Cascade is not able to prevent such situations, it does detect them by periodically checking for inactive components with non-empty input fifos. If it finds any then the simulation is aborted with the following error message:

```
Error: Top level, simTime = 48359868750
Error: Deadlock detected!
     Test1.Consumer.in_data is non-empty
     but Test1.Consumer is inactive
```

7.1 Multiple update functions

Normally the call to `deactivate()` occurs within a component's `update()` function. However, if a component has multiple update functions then this can cause problems because the relative order of the updates is unknown: a call to `deactivate()` within one update function could prevent another update function from being called on a cycle during which it should have been (e.g. because it services an input fifo that just received data). In this case, the call to `deactivate()`, as well as the test to determine whether or not the component is active, should occur within the component's `tick()` function. This is always safe because Cascade calls `tick()` before processing synchronous events that could activate the component.

7.2 Triggers

A second technique that can be used to optimize a simulation by avoiding the need to poll certain inputs and/or avoiding the need to activate components is the use of *triggers*. A trigger is an action, associated with a port, that is invoked whenever the value of the port is non-zero (active-high triggers) or zero (active-low triggers). The action of a trigger is defined by an implementation of the `ITrigger` interface which specifies a single function templated on the type of the port:

```
template <typename T>
struct ITrigger
{
     virtual void trigger (const T &data) = 0;
};
```

A trigger action is associated with a port using the member function

```
void addTrigger (ITrigger<T> *trigger,
                 PortActivationType activationType = ACTIVE_HIGH)
```

All regular ports support activation and triggers, but Cascade is optimized for single-byte active-high triggers, so the best performance will be obtained with active-high triggers associated with ports of type `bit` (Section 8.0), `byte`, or bit vectors (Section 8.0) of width at most 8. Multiple triggers and activations may be associated with a single port.

When the `trigger()` function is called, the value of the port is passed as the single argument. The `trigger()` function is always called before any update function that would ordinarily read the input port. If the `trigger()` function is a component member function then it can read any valid input port, but it cannot write output ports: only update functions and scheduled events are allowed to write output ports.

Example

```
class Counter : public Component, public ITrigger<bit>
{
    DECLARE_COMPONENT(Counter);
public:
    Counter (COMPONENT_CTOR) : m_count(0)
    {
        in_valid.addTrigger(this);
    }

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(bit, in_valid);
    Input(int, in_amount);
    Output(int, out_count);

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ()
    {
        out_count = m_count; // correct
    }
    void trigger (const bit &)
    {
        m_count += in_valid; // correct (can read ports from trigger function)
        out_count = m_count; // WRONG!!! (cannot write ports)
    }
};
```

Trigger actions can also be associated with fifo ports, but since a fifo port is not allowed to have fanout, at most one trigger action can be associated with a given fifo port, and this action *replaces* the default action of activating the consumer component. A trigger action is associated with a fifo port using the member function

```
void setTrigger (ITrigger<T> *trigger)
```

Note that a fifo port trigger is not defined as active-high or active-low; the trigger action is invoked as soon as data arrives at the consumer. If the fifo has no delay, then the action is immediately invoked when the producer pushes data onto the fifo, and takes place before the call to `push()` returns. If the fifo has a delay, then the action is invoked on the appropriate rising clock edge. Again, the value in the fifo is passed as the single argument to `trigger()`, and is automatically popped from the fifo.

Setting a fifo trigger is incompatible with sending the fifo to the bit bucket, connecting another fifo port to the fifo, or explicitly reading from the fifo: a fifo with a trigger action may not be accessed via `pop()`, `peek()`, `empty()` or `popcount()`. Attempts to use these functions will generate assertion failures in debug builds. However, a zero-delay fifo with a trigger action is still required to have exactly one reader as explained in Section 3.8.4. This is so that Cascade can schedule the update function that writes to the fifo before the update function that sees the effects of the `trigger()` function.

The following example shows how triggers can be used to create an efficient implementation of credit-based virtual channel flow control. The `trigger()` action, implemented in `VCCreditCounter`, increments the per-virtual channel credit counts. This is more efficient than an implementation in which the `NetworkOutput` update function polls `in_vc_credit` on every cycle.

```
class VCCreditCounter : public Component, public ITrigger<byte>
{
     DECLARE_COMPONENT(VCCreditCounter);
public:
     VCCreditCounter (COMPONENT_CTOR) {}

     //----------------------------------------
     // Accessors
     //----------------------------------------
     inline bool hasCredit (byte vc) const
     {
          assert(vc < NUM_VCS);
          return m_credits[vc] != 0;
     }
     inline void useCredit (byte vc)
     {
          assert(vc < NUM_VCS);
          m_credits[vc]--;
     }

     //----------------------------------------
     // Simulation
     //----------------------------------------
     void reset ()
     {
          for (unsigned i = 0 ; i < NUM_VCS ; i++)
               m_credits[i] = VC_DEPTH;
     }
     void trigger (byte vc)
     {
          assert(vc < NUM_VCS);
          m_credits[vc]++;
     }

private:
     byte m_credits[NUM_VCS];
};

class NetworkOutput : public Component
{
     DECLARE_COMPONENT(NetworkOutput);
public:
     NetworkOutput (COMPONENT_CTOR)
     {
          in_credit_vc.setTrigger(&m_creditCounter);
     }

     //----------------------------------------
     // Interface
     //----------------------------------------
     FifoOutput(bit, out_valid);
     FifoOutput(int, out_data);
     FifoOutput(byte, out_vc);
     FifoInput(byte, in_credit_vc);
     …
private:
     VCCreditCounter m_creditCounter;
};
```

8.0 Bit Vectors

Cascade defines a templated type for signed or unsigned bit vectors of arbitrary width. For $n \geq 0$, the type `bitvec<n>` stores an unsigned n-bit vector, and the type `bitvec<-n>` stores a signed n-bit vector. For convenience, for $n \leq 128$ and $n = 256$, `bitvec<n>` is typedefed to `un`, and `bitvec<-n>` is typedefed to `sn`. Additionally, the single-bit type `bitvec<1>` is typedefed to "`bit`".

Example

```
bit b;              // single bit
u5 v5;              // 5 bit unsigned bit vector
s19 v19;            // 19 bit signed bit vector
bitvec<196> v196;   // 196 bit unsigned bit vector
bitvec<-228> v228;  // 228 bit signed bit vector
```

For $n \leq 64$, unsigned n-bit vectors can be constructed from or converted to unsigned integers, and signed n-bit vectors can be constructed from or converted to signed integers. Thus, small bit vectors can be treated as integers and support all standard operations; the compiler will automatically supply the necessary type conversions.

Internally, small bit vectors are stored using the smallest possible integral type, so in release builds there is no storage or performance penalty associated with using bit vectors instead of integers. For example, a `u4` is implemented as a `uint8` and occupies one byte of storage, and an `s13` is implemented as an `int16` and occupies two bytes. In debug builds, assertions are added to ensure that the bit vector's value lies within the legal range of representable values.

For large bit vectors ($n > 64$), only the Boolean operators (&, |, ^, ~, &=, |=, ^=) and shifting operators (<<, >>, <<=, >>=) are supported. For large signed bit vectors, the right shift operator copies the high bit (i.e. the bit vector is treated as a large signed integer). A large unsigned bit vector can be constructed (or assigned to) from a `uint64`, and a large signed bit vector can be constructed (or assigned to) from an `int64`. In the latter case, the bit vector is sign extended appropriately. Large bitvectors *cannot* be converted to integers, and cannot be used directly as the predicate in an `if` or `while` statement. Equality comparisons to 64-bit integers are supported (==, !=), so large bitvectors can be used as predicates by comparing to zero, e.g. "`if (val != 0)`". Internally, large bit vectors are stored using an array of 64-bit integers.

For convenience, additional constructors are supplied for large bit vectors which take up to eight 64 bit integers, allowing the lower 512 bits to be set directly. The constructors take their arguments in the order of high bits to low bits, with the last argument corresponding to bits [63:0]. Thus, in the following example:

```
uint64 val2, val1, val0;
...
u192 v192(val2, val1, val0);
```

`v192` is constructed with `val2` in bits [191:128], `val1` in bits [127:64], and `val2` in bits [63:0].

8.1 Indexing bit vectors

Bit vectors support the standard array indexing operator to read or set individual bits. For a bitvector of width n, the legal index values range from 0 (low bit) to n–1 (high bit).

Example

```
u17 v17 = 0x1a34e;  // v17 = 11010001101001110
v17[10] = 1;        // v17 = 11010011101001110
v17[2] = v17[4];    // v17 = 11010011101001010
bit b = v17[8];     // b = 1
```

8.2 Bit vector slices

The function call operator *bits*(a, b) can be used to read or set the slice [a:b] of the bit vector *bits* as a single $a - b + 1$ bit entity (note that the slice arguments appear in standard high, low order). A slice can appear on either the left hand side or right hand side of an assignment or comparison, however the widths of the two sides must match exactly. Mismatched widths will generate assertion failures in debug builds. Additionally, slices of ≤ 64 bits can be assigned from or converted to unsigned integers.

Example

```
u17 v17 = 0x1a34e;              // v17 = 11010001101001110
u8 v8 = v17(15,8);              // v8  =  10100111
v17(11,3) = 0x1a4;              // v17 = 10111110100100110
v8(5,2) = v17(13,10);           // v8  =  10111111
if (v17(10,8) == v17(16,14))    // true (101 == 101)
    v17(8,1) = v8;              // v17 = 10111111010111110
```

When compile-time constants are used for the slice operator, the compiler is able to optimize it heavily and in release mode it is just as fast (and uses the exact same assembly instructions) as manually manipulating the bits of an unsigned integer. A slice from a bit vector can be assigned to another slice from the same bit vector, e.g.:

```
v17(15,11) = v17(8,4);
```

The results are undefined, however, if the two slices overlap.

8.3 Concatenation

Bit vectors, indexed bits and slices can be concatenated using the comma operator to form larger compound bit vectors. The same usage rules that apply to slices apply to compound bit vectors: they can appear on the left hand side or right hand side of any assignment or comparison, the widths of the two sides must match exactly or an assertion failure will be generated in debug builds, and compound bitvectors with ≤ 64 bits can be assigned from or converted to unsigned integers.

Example

```
(v2, v4, v8) = 0x2cfe;  // v2 = 0x2,  v4 = 0xc,  v8 = 0xfe
v27 = (v13, v14);
(v13, v14) = v27;
(v5[2], v7, v18(10, 2)) = ((v3, u5(0)), (v19(16, 9), v6[3]));
```

Concatenation takes the form (*high bits*, *low bits*), so in the first example the result of the assignment is as shown in the comment, while in the second example `v13` is assigned to the upper 13 bits of `v27` and `v14` is assigned to the lower 14 bits. A

constant cannot appear directly within a compound bit vector as C++ constants do not have bit widths. However, as shown in the last example, a constant can be cast to a bit vector which has the desired effect. As with slices, if a bit vector appears on both the left hand side and the right hand side of an assignment, the bits being read on the right hand side must be distinct from those being set on the left hand side, or the results are undefined. Similarly, if the same bit within a bit vector appears twice on the left hand side of an assignment then the results are undefined.

8.4 Reduction operators

Cascade supports three reduction operators:

```
bit reduce_and (a [, b, …])
bit reduce_or  (a [, b, …])
bit reduce_xor (a [, b, …])
```

Each of these operators reduces the operands to a single bit using the indicated boolean reduction. The arguments can be bit vectors, indexed bits, bit slices or compound bit vectors. Each operator takes an arbitrary number of arguments. For bit vectors, bit vector slices and compound bit vectors, the `!` operator is supported which returns true if all of the bits are zero, e.g.:

```
if (!v7(6,2) || !(v6[2], v9))
    ...
```

This is effectively a fourth reduction operator equivalent to `!reduce_or(`*bits*`)`.

8.5 Functions

The following functions are supported on bit vectors, indexed bits, bit slices and compound bit vectors:

`int popcount (x)`

Returns the number of bits that are set in the argument.

`int lsb (x)`

Returns the zero-based index of the least significant bit that is set in the argument. If the argument is zero, returns the size of the argument in bits.

8.6 Assignments to signed bit vectors

When a signed bit vector is assigned from an unsigned bit vector, the compiler automatically provides intermediate conversions to/from integers. So in the following example:

```
u11 a11 = 0x7ea;
s11 b11 = a11;        // ERROR!!!  Value out of range.
s11 b11 = a11(10,0);  // OK (raw bit assignment)
```

on the second line the unsigned bit vector `a11` is not directly assigned to `b11`; it is first converted to the integer 0x7ea and then assigned to `b11`, generating an error in debug builds because this positive value is outside the legal range of values for `b11`. Taking the slice (10,0) of `a11` (third line) forces the assignment to be treated as raw bits, after which b11 is appropriately sign extended.

8.7 Bit vector traits

Each bit vector type contains a 'traits' type which in turn contains a number of constants and typedefs related to the bit vector type. This can be useful for types that are typedefed to a bit vector whose width is subject to design changes, as in the following example:

```
typedef u4 CreditCount;
...
CreditCount cnt = CreditCount::traits::maxval;
```

In this example, the variable cnt will always be initialized with the largest legal value; it is not necessary to change this code if at some point in the design CreditCount changes from a u4 to a u3. The full set of typedef and constant traits for bit vectors is shown in Table 1.

Table 1. Bit vector traits.

Name	Type	Description
bv_t	*typedef*	Integer type used to represent bit vector: (int|uint)(8|16|32|64)
u_t	*typedef*	Unsigned integer type of same width as bv_t: uint(8|16|32|64)
const_t	*typedef*	Integer type used in assignment operator: (int|uint)(32|64)
width	unsigned	Width of bit vector in bits
usize	unsigned	Size in bits of bv_t
arraylen	unsigned	Length of array of bv_t required to hold bit vector (len = 1 if width ≤ 64)
mask	bv_t	Bitmask of valid bitvector bits in highest word in array of bv_t
umask	u_t	Same as mask, but unsigned type
msb	bv_t	Most significant bit of highest word in array (sign bit for signed bit vectors)
maxval	bv_t	When width ≤ 64, largest integer that can be represented by bit vector
minval	bv_t	When width ≤ 64, smallest integer that can be represented by bit vector

8.8 Using bit vectors as ports

The most common reason to use a bit vector type for a port is to specify the exact port width; it is still convenient to access the actual value directly as an integer. The implicit conversion read operators for bit vector ports therefore return signed or unsigned integers of the appropriate size, allowing the ports to be accessed within integer expressions without requiring the overloaded function call operator to explicitly cast the value. In order to access the port value as bit vector, one of the explicit read accessors must be used.

Example

```
Input(s23,  in_a);
Input(s23,  in_b);
Output(s23, out_sum);
Output(u46, out_concat);
...
void update ()
{
    out_sum = in_a + in_b;
    out_concat = (*in_b, *in_a);
}
```

8.9 String conversion

Functions are provide to convert a bit vector, indexed bit, bit slice or compound bit vector to or from a hexadecimal or binary string.

`string str (x)`

Converts the argument to a fixed-width hexadecimal string prefixed with "`0x`". If the width of the argument is N then the number of hexadecimal digits in the returned string is $\lfloor (N+3)/4 \rfloor$ (so in particular leading zeros are included in the output). The last digit of the output contains the least significant bits. There is no difference in output between signed and unsigned bit vectors; the output string is based on the raw bit values contained in the argument.

`string str_bits (x)`

Converts the argument to a fixed-width binary string with no prefix. An N-digit string of 1's and 0's is generated for an N-bit argument, with the least significant bit at the end of the string.

`fromString (T &, const string &s)`

Parses a hexadecimal string into one of the supported types. Whitespace at the start of the string is ignored, and the string may optionally contain the "`0x`" prefix. If the string is not a valid hexadecimal string or if it specifies a value that is too wide for the width of the argument, then a `strcast_error` exception is thrown.

`fromBitString (T &, const string &s)`

Parses a binary string into one of the supported types. Whitespace at the start of the string is ignored. If the string is not a valid binary string or if it specifies a value that is too wide for the argument, a `strcast_error` exception is thrown.

8.10 Bit vector references

A bit vector reference allows arbitrary data to be treated as a bit vector, so that the data can be manipulated using bit vector operations such as indexing, slicing and concatenation. A bit vector reference has type `bitvecref` and is declared in the same manner as a bit vector (and takes the same template parameter specifying size and sign), but the constructor requires a pointer to the data. A second optional constructor parameter specifies an offset into the data, measured in bits. A bit vector reference behaves in every respect like a bit vector, the only difference being that the actual bits are stored elsewhere.

Example

```
uint32 data = 0x12345678;
bitvecref<16> u16(&data);   // Treat data[15:0] as an unsigned bit vector
u16(15,4) = 0xabc;          // data = 0x1234abc8
bitvecref<-8> s8(&data,16); // Treat data[23:16] as a signed bit vector
s8(3,0) = 0xe;              // data = 0x123eabc8
```

8.10.1 Const bit vector references

As shown in the above example, the `bitvecref` type allows the bits to be modified and must therefore be constructed using a non-const pointer to the data. To treat constant data as a bit vector, use the `const_bitvecref` type instead. This type behaves in every respect like a `const bitvector`.

Example

```
const uint32 data = 0x12345678;
bitvecref<24> u24(&data,4);
int x = u24(19,4);   // x = 0x3456
```

8.10.2 Improving bit vector reference performance

In order to ensure correctness when compiling with strict aliasing rules, a bit vector reference internally stores a byte pointer, and all operations performed on the bit vector reference are performed using bytes.[1] This provides convenience (the constructor takes a `void *` pointer so there is no need to type-cast the argument) and safety (it eliminates potential errors due to aliasing) at the cost of performance, since if the underlying data type is a larger integer then operations on that larger integer type would be more efficient. In cases where performance is of concern, the pointer type of the bit vector reference (and the type used for all operations) can be specified by supplying one of `BITVECREF_16BIT_PTR`, `BITVECREF_32BIT_PTR` or `BITVECREF_64BIT_PTR` as a second template parameter. In this case, the pointer type supplied to the bit vector reference constructor must be a pointer to an integer (signed or unsigned) of the specified width.

Example

```
uint32 data = 0x12345678;
bitvecref<16, BITVECREF_32BIT_PTR> u16(&data);

uint16 values[3] = { 0x1234, 0x5678, 0x9abc };
const_bitvecref<24, BITVECREF_16BIT_PTR> u24(&values, 12);
```

[1] With strict aliasing rules, the compiler may assume that pointers to unrelated types point to different data unless one of the pointer types is a signed or unsigned `char`. Thus, for example, if a `uint16` pointer is used to modify a `uint32` which is then used elsewhere, the compiler may reorder the instructions such that the `uint32` value is used before it is modified, producing unexpected results.

9.0 Logging and Tracing

Cascade uses the descore logging and tracing mechanisms, and in particular `log()` is intended to be used in place of `printf()`. Trace output is automatically prefixed with the current simulation time in nanoseconds.

9.1 Component tracing

A component's trace context is its name; refer to the descore documentation for tracing syntax, how to define trace keys, and how to enable/disable tracing within specific contexts. As a convenience, two functions are provided to directly control tracing within a component:

`void Component::setTrace (const char *key = "")`
`void Component::unsetTrace (const char *key = "")`

Enable/disable any matching traces within the component, where `key` is a wildcard string. Equivalent to calling the functions `descore::setTrace()` or `descore::UnsetTrace()` and providing the component name as the first argument.

9.2 Tracing from an interface

Tracing is also supported from within interface member functions. An interface inherits the trace context of its containing component.

9.3 Controlling tracing based on time

Trace output can be globally restricted to a specified time interval by setting two parameters:

```
cascade.TraceStartTime
cascade.TraceStopTime
```

Each of these is an unsigned integer parameter that specifies a time in nanoseconds. Trace output will be enabled when the global simulation time is in the specified interval (inclusive on both ends), and disabled otherwise.

9.4 Trace specifier format

Here we review the trace specifier format described in "The descore Library", which can be used to enable tracing from a specifiers string by:

- calling `descore::setTraces()`, or
- specifying `-trace <specifiers>` on the command line if the application calls `descore::ParseTraces()`, or
- enabling tracing within a Verilog co-simulation using one of the methods described in Section 13.6, or
- setting the string parameter `cascade.Traces`

First, the specifiers string is parsed into multiple independent specifier strings according to the following rules:

- Top-level semicolons delimit separate strings
- Any substring contained in curly braces is recursively expanded into a list of strings, then the curly braces are replaced by each string in this list

Next, each string within the list is parsed as a specifier of the form

```
component[/keyname][:filename]
```

where `component` is a wildcard string matched against component names, and `keyname` is an optional wildcard string specifying one or more trace keys, and `filename` is an optional wildcard string used to further restrict the set of trace keys to those defined in matching files. The following examples illustrate the use of trace specifiers on the command line:

```
// Enable anonymous tracing for all components whose name ends in “Multiplier”
-trace “*Multiplier”

// Enable anonymous tracing for Adder and trace keys starting with “packet” for Router
-trace “Adder;Router/packet*”

// Within Decoders 0 and 2, enable the named trace “input”, and anonymous
// tracing for all of their Filter subcomponents
-trace “Decoder{0,2}{/input;.Filter*}
```

10.0 Archiving

The archiving functionality in Cascade is designed to facilitate saving and loading the state of an entire simulation. Archiving makes use of the descore `Archive` class. Cascade provides the following global static functions for archiving a simulation:

```
void SimArchive::archiveSimulation (Archive &ar)
void SimArchive::loadSimulation (const char *filename)
void SimArchive::saveSimulation (const char *filename, bool safeMode = false)
```

These functions must be called *after* the simulation has been initialized. The first function uses an existing `Archive` object, which allows the programmer to archive additional data before or after the simulation. The other two functions are convenience wrappers that create an archive and can `archiveSimulation()`, for example `SimArchive::loadSimulation()` is equivalent to

```
Archive ar(filename, Archive::Load);
SimArchive::archiveSimulation(ar);
```

The parameter `safeMode` indicates that the archive should be created in safe mode, which inserts check bytes as a debugging tool to detect mismatches between code for saving and loading state.

Cascade automatically archives the current simulation state including all clock domains and ports. Additionally, each component's `archive()` function is called; the programmer is responsible for implementing these functions to archive any component internal state. The default `Component::archive()` implementation generates an assertion failure, so this function *must* be implemented for all components in the simulation, including components with no internal state that needs to be archived.

Successfully restoring the state of a simulation depends on archiving all components, ports and clock domains in a consistent order; in particular it is not valid to attempt to load a simulation from an archive that was created from a different hardware configuration. To protect against this, the first entry in the simulation archive is a hardware checksum which is a 32-bit CRC of the current configuration. When a simulation is loaded the checksum in the file is compared against the current hardware configuration; an assertion failure is generated if they do not match.

Simulation archives are compatible across debug/release builds so, for example, the state of a simulation can be saved by a release build executable and reloaded by a debug build executable provided that the same underlying hardware is being simulated.

10.1 Checkpoints

Cascade provides support for creating periodic checkpoints and restoring a simulation from a checkpoint. This functionality is controlled by five parameters:

```
cascade.CheckpointInterval      // integer
cascade.CheckpointName          // string
cascade.RestoreFromCheckpoint   // string
cascade.SafeCheckpoint          // boolean
cascade.ValidateCheckpoint      // string
```

cascade.CheckpointInterval specifies the interval, measured in nanoseconds, between simulation checkpoints. Note that simulation time is measured in picoseconds, but the checkpoint interval is measured in nanoseconds. If the checkpoint interval is zero then no checkpoints will be generated. If it is positive, then checkpoint files will be generated at regular intervals and will be named

```
<name>_<time>.ckp
```

<name> is the checkpoint name as specified by the cascade.CheckpointName parameter, which can also include the checkpoint directory. *<time>* is the current simulation time in ns.

The cascade.RestoreFromCheckpoint parameter can be used to automatically restore the simulation from the specified checkpoint file when the simulation is initialized. In particular,

```
Sim::init();
SimArchive::loadSimulation("sim_300.ckp");
```

is equivalent to

```
Cascade::params.RestoreFromCheckpoint = "sim_300.ckp";
Sim::init();
```

The latter form may be more convenient as it allows the decision to restore from a checkpoint to be separated from the code that initializes a simulation.

The cascade.SafeCheckpoint parameter controls whether or not the checkpoint is created in safe mode; when a checkpoint is created this parameter is passed as the second argument to SimArchive::saveSimulation().

Finally, cascade.ValidateCheckpoint can be used when restoring the simulation from a checkpoint to validate the state against a second checkpoint file. For example, suppose two runs that should be identical both produce a checkpoint file at 1000ns, but due to a non-determinism bug the simulations diverge and the checkpoint files differ. Restoring from one of these checkpoint files (by setting cascade.RestoreFromCheckpoint to its filename) and validating against the other checkpoint file (by setting cascade.ValidateCheckpoint to its filename) will identify the locations within the component hierarchy at which the two simulation states differ. An error message will be displayed for every component whose state is different in the two archives. Setting both parameters to the same file can also elucidate certain types of bugs in the component archive() functions themselves.

10.2 Archiving custom data

With checkpointing, the Archive object used to save the simulation state is created by Cascade. To allow the programmer to store additional custom data with each checkpoint, archive callback functions may be registered:

```
typedef void (*SimArchive::Callback) (Archive &)
void SimArchive::registerCallback (SimArchive::Callback callback)
void SimArchive::unregisterCallback (SimArchive::Callback callback)
```

An archive callback function takes a single parameter: the current Archive object. Multiple callback functions may be registered (or unregistered, using the second function). When a simulation is archived, either manually or as the result of a checkpoint, the callback functions are invoked in the order in which they were

registered *before* the simulation state is archived. This mechanism can be used, for example, to archive simulation state that is not contained in any component.

10.3 Archiving component/interface pointers

Two templated functions are provided for archiving pointers to components and interfaces:

```
void archiveComponentPointer (Archive &ar, T *&component);
void archiveInterfacePointer (Archive &ar, T *&interface);
```

The pointers are encoded within the archive in a manner that allows them to be correctly reconstructed when the archive is loaded. A static assertion failure is generated if `archiveComponentPointer()` is called with a non-component pointer or if `archiveInterfacePointer()` is called with a non-interface pointer.

Example

```
Adder *current_adder;
ISRAM *current_sram;
...
archiveComponentPointer(ar, current_adder);
archiveInterfacePointer(ar, current_sram);
```

11.0 Multithreading

Cascade's simulation engine supports multithreading, allowing simulations to run faster when multiple cores are available. The multithreading strategy is fairly simple: when multiple clock domains have simultaneous rising clock edges, the clock domains are partitioned among the available threads. Thus, to make effective use of multithreading, the programmer may need to artificially subdivide clock domains into two or more domains with identical clocks.

11.1 Subdividing clock domains

The only restriction when subdiving a clock domain into multiple domains is that connections between components in different domains must be synchronous. A clock domain can be subdivided either by explicitly defining multiple clocks with the same period and offset, or by deriving one clock from another with no offset and ratio 1. For example, consider the following simulation:

```
Receiver receiver;
Decoder decoder;
Clock clk;
syncConnect(receiver.data_out, decoder.data_in);
receiver.clk << clk;
decoder.clk << clk;
clk.generateClock();
```

This simulation contains only a single clock domain and so will not benefit from multithreading. We can make use of two threads by placing each component in its own clock domain:

```
Receiver receiver;
Decoder decoder;
syncConnect(receiver.data_out, decoder.data_in);
receiver.clk.generateClock();
decoder.clk.generateClock();
```

Subdiving clock domains in this manner has no effect on the simulation (other than supporting the use of multiple threads).

11.2 Controlling threading

Multithreading is controlled using the parameter `cascade.NumThreads`. The number of threads can be at most the number of cores; if the value of this parameter is too large then Cascade will use one thread per core. Additionally, if this parameter is set to -1 then Cascade will use the maximum number of threads (one thread per core).

Example

```
$ run_simulation -cascade.NumThreads=4
```

11.3 Avoiding race conditions

As long as all communication between components is through connected ports, Cascade ensures that multithreading does not introduce any race conditions by requiring all inter-clock-domain connections to be synchronous. However, race conditions can be introduced if a component in one clock domain magically calls a member function of a component in another clock domain, and the effect of the function call is immediately visible. Such function calls should therefore be avoided to ensure deterministic multithreaded simulation results.

12.0 Generating VCD Files

The cycle-by-cycle state of a simulation can be captured in a Value Change Dump (VCD) file, which we refer to as "generating waves". The resulting waves can be inspected using a viewer such as DVE. Cascade can automatically dump the port and register values for any specified subset of the simulation hierarchy. Additional state can be added to the dump file by explicitly declaring signals.

The VCD filename is controlled by the parameter `cascade.WavesFilename`, and defaults to "`sim.vcd`". The timescale string within the VCD file is controlled by the parameter `cascade.WavesTimescale`, and defaults to "`1 ps`".

12.1 Functions controlling waves generation

Several variants of the global function `Sim::dumpWaves()` are provided to control the generation of waves:

```
void Sim::dumpWaves (const char *component, int level = 0)
void Sim::dumpWaves (const char *component, const char *signals, int level = 0)
void Sim::dumpWaves (Component *component,  int level = 0)
void Sim::dumpWaves (Component *component,  const char *signals, int level = 0)
```

These functions *must* be called during the construction phase; once the simulation has been constructed the set of signals being dumped cannot be modified. Each of these functions specifies a subset of signals within a subtree; the VCD file will contain the union of all these subsets.

The `component` argument specifies the root of the subtree. It can be specified as either a pointer or a wildcard string; in the latter case every component whose name matches the wildcard string defines the root of a subtree. In particular, specifying "`*`" will enable signal dumping for all components.

The optional `signals` argument specifies the set of signals to dump within the subtree as a wildcard string. Any port or register or explicitly declared signal within the subtree whose name matches the wildcard string will be included in the VCD file. If this argument is omitted then it defaults to "`*`".

Finally, the optional `level` argument specifies the depth of the subtree. The default value of 0 includes the entire subtree. If `level` is positive then it specifies the number of levels in the hierarchy to include starting with the root of the subtree, so if `level` is 1 then only the root component of each subtree is included.

12.2 Clocks and timing model

Clocks will appear in the waves in the expected manner, with evenly spaced rising and falling edges. Signals within a clock domain will transition on the rising clock edges. Manual clocks present a challenge, because value change events are added to the VCD file in the order in which they occur during simulation, but as explained in Section 4.4 the simulated times of these events are not necessarily monotonically increasing in the presence of derived manual clocks. However, the timestamps within the VCD file must always be monotonically increasing.

Cascade deals with this issue by defining a minimum time increment for the VCD file, measured in ps, which is controlled by the parameter `cascade.WavesDT` and defaults to 10 ps. When the simulation time changes, the current time in the VCD file is updated, if possible by simply setting it to the current simulation time.

However, if this would increase the VCD file time by less than the minimum increment (or if it would decrease the time), then the VCD file time is instead increased by the minimum increment. This can cause events within a derived manual clock domain to appear at a later time, or clumped together in the waves separated by this minimum increment.

As an example, consider the following simulation of three instances of a counter component:

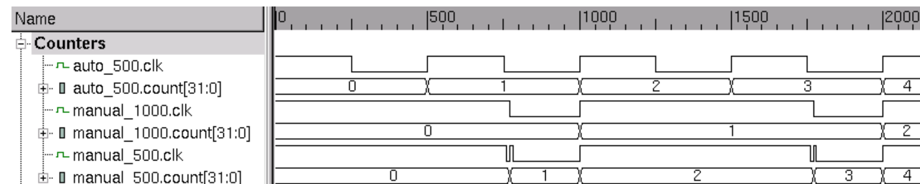
```
class Counter : public Component
{
    DECLARE_COMPONENT(Counter);
public:
    Counter (const char *name, COMPONENT_CTOR) : m_componentName(name) {}

    //----------------------------------
    // Interface
    //----------------------------------
    Clock(clk);
    Output(int, count);

    //----------------------------------
    // Simulation
    //----------------------------------
    void reset ()
    {
        m_count = 0;
    }
    void update ()
    {
        count = m_count++;
    }

private:
    int m_count;
    const char *m_componentName;
};

int main ()
{
    Counter auto_500("auto_500");
    auto_500.clk.generateClock(500);
    Counter manual_1000("manual_1000");
    manual_1000.clk.setManual();
    Counter manual_500("manual_500");
    manual_500.clk.divideClock(manual_1000.clk, 0.5);

    for (int i = 0 ; i < 5 ; i++)
    {
        manual_1000.clk.tick();
        Sim::run(1000);
    }
}
```

The `Counter` component simply increments its output count on each rising clock edge. The `auto_500` instance has an automatic clock with a 500ps period. The `manual_1000` instance has a manual clock with a 1000ps period (effected by calling its `tick()` function every 1000ps). Finally, the `manual_500` instance has a clock period of 500ps, obtained by dividing the clock of `manual_1000`.

Logically, `auto_500` and `manual_500` should behave identically, with rising clock edges at 0ps, 500ps, 1000ps, … and falling clock edges at 250ps, 750ps, 1250ps, … while manual_1000 should have rising clock edges at 0ps, 1000ps, … and falling clock edges at 500ps, 1500ps, etc. However, by the second time that `manual_1000.clk.tick()` is called at 1000ps, the VCD timestamp has already reached 750ps due to the second falling edge of `auto_500.clk`. As a result, the first falling edge of `manual_500.clk`, which logically occurs at 250ps, is placed at 760ps within the waves. Then the simultaneous rising edge of this clock and

the falling edge of `manual_1000.clk`, logically at 500ps, are placed at 770ps. Finally, the second falling edge of `manual_500.clk`, logically at 750ps, is placed at 780ps. This pattern then repeats every 1000ps (Figure 6).

Figure 6. Waves generated by Counter example. Events always appear in the waves in simulation order rather than chronological order. As a result, when the simulation contains derived manual clocks, certain events may appear in the waves later than their actual logical time.



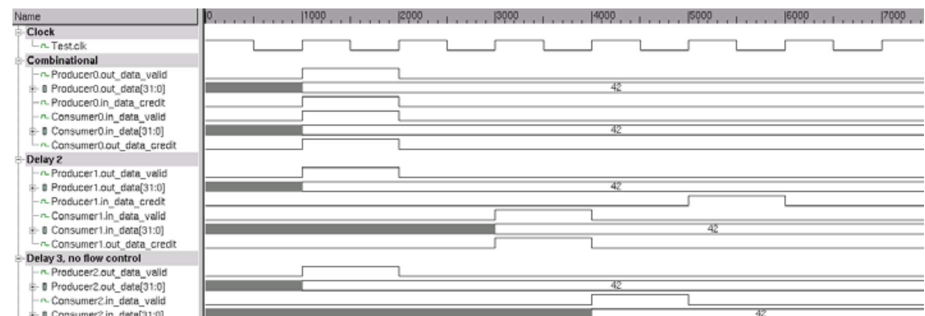
12.3 Fifo ports

When a fifo port is added to the waves, it is automatically converted to a credit-based interface: a 'valid' signal is added in the forward direction, and a 'credit' signal is added in the reverse direction. The 'valid' signal pulses high at the producer during the cycle that data is pushed onto the fifo, and it pulses high at the consumer on the cycle that the data becomes visible (as determined by the fifo's delay). At both the producer and consumer, the data signal is shown as invalid (X) before the first push; it becomes valid when the 'valid' signal goes high and remains valid thereafter, always displaying the most recent value that was pushed.

The 'credit' signal pulses high at the consumer when data is popped from the fifo, and it pulses high at the producer when the available fifo entry becomes visible (again, as determined by the fifo's delay). If the fifo's flow control has been disabled then only the 'valid' signal is added, and there is no 'credit' signal.

Figure 7 shows waves for three consumers connected to three producers by 32-bit fifos. The first producer/consumer pair uses a combinational fifo. The second pair uses a fifo with delay 2, so the valid signal is delayed by two clock cycles in the forward direction and the credit return signal is delayed by two clock cycles in the reverse direction. The third pair uses a fifo with delay 3 and disabled flow control, so there is no credit return signal.

Figure 7. Three producer/consumer pairs connected by 32-bit fifos. The first fifo is combinational, the second has delay 2, and the third has delay 3 with flow control disabled.



Note that in a behavioral simulation, multiple values can be pushed onto a fifo during the same cycle; in this case only the last one will be visible in the waves. Similarly, multiple values can be popped from a fifo during the same cycle; in this case only a single pulse on the credit return signal will appear in the waves.

12.4 Invalid signals

In debug builds, the port valid flags (Section 3.3) are used (when available) to determine whether or not a port value is valid. Invalid ports are shown as invalid (X) in the waves. In release builds there are no valid flags, so all ports are shown as valid in the waves, with the exception of fifo data ports before the first push as described in the previous section.

12.5 Explicit Signals

Additional component member variables can be added to the waves by explicitly declaring them as signals using the `Signal` macro:

```
Signal(<type>, <name>)
```

This macro creates a normal member variable with the specified type and name, and it registers the member variable with Cascade's reflection infrastructure so that it can be automatically added to the waves. The variable can be either a single variable or a single-dimensional statically sized array; in the latter case the second argument to the macro is *name*[*size*]. As with the port macros, the `Signal` macro must appear within a public section of the class declaration.

Example

```
class CreditCounter : public Component
{
    DECLARE_COMPONENT(Counter);
public:
    Counter (COMPONENT_CTOR) {}

    //---------------------------------
    // Interface
    //---------------------------------
    Clock(clk);
    Input(u4, in_initial_credits);
    Input(bit, in_use_credit);
    Input(bit, in_return_credit);
    Output(bit, out_has_credits);

    //---------------------------------
    // Simulation
    //---------------------------------
    void reset ()
    {
        credit_count = in_initial_credits;
        out_has_credits.reset(0);
    }
    void update ()
    {
        credit_count += in_return_credit - in_use_credit;
        out_has_credits = creditCount ? 1 : 0;
    }

    Signal(u4, credit_count);  // must be public
};
```

12.6 Data Representation

By default, the values that appear in the waves are the raw bits of the ports and signals being dumped. One can also arbitrarily transform these values on a per-

type basis, modifying the number of bits, the values themselves, or both. The methods for doing so are the same as for marshalling data between Cascade and a Verilog simulation, and are described in Sections 13.1 and 13.2.

12.7 Controlling waves generation from the command line

A helper function is provided to parse wave dumping directives from the command line and remove these directives from the command line:

```
void Sim::parseDumps (int &csz, char *rgsz[])
```

Each dump directive is of the form

```
-dump <signals>
```

where `<signals>` is a string specifying a set of signals to dump to the waves file. The string is first expanded into a list of strings using the following rules:

- Top-level semicolons delimit separate strings
- Any substring contained in curly braces is recursively expanded into a list of strings, then the curly braces are replaced by each string in this list

For example, the string `"Decoder/{in*;out*}"` expands to the list of string `["Decoder/in*", "Decoder/out*"]`. Each individual string in the list should then be of the form '*component*[:*level*][/*signals*]' where *component*, *level* and *signals* have the same meanings and defaults as the arguments to `Sim::dumpWaves()`, described in Section 12.1.

The set of signals to dump can also be specified by setting the string parameter `cascade.DumpSignals`. In particular, if an application supports both command-line dump specifiers and command-line parameter specifiers, then

```
-dump dumpstring1 -dump dumpstring2 … -dump dumpstringN
```

is equivalent to

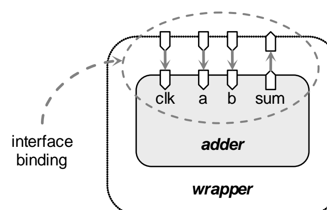
```
-cascade.DumpSignals="dumpstring1;dumpstring2;…;dumpstringN"
```

13.0 Cascade/Verilog Co-simulation

Cascade supports mixed C++/RTL simulations where some components are modeled using Cascade and others are modeled using Verilog. This is primarily intended to support design verification: Cascade golden models can be instantiated within Verilog testbenches, and Verilog modules can be tested in a "natural" environment by instantiating them within Cascade simulations. Co-simulation with a Cascade component inside a Verilog simulation can be implemented using either the VPI or the DPI interfaces; co-simulation with a Verilog component inside a Cascade simulation requires the VPI interface.

The key mechanism used to support co-simulation is *interface binding*: if a Verilog module and a Cascade component have the same ports in the same order, Cascade can automatically establish a binding between corresponding ports so that data is transparently marshaled between Verilog and C++. To instantiate a Cascade component within a Verilog simulation, an empty Verilog module is created with the same interface as the C++ model. The C++ model is then created and interface binding is used to associate its ports with those of the Verilog wrapper (Figure 8). The wrapper can be used within the RTL simulation in the same manner as any other Verilog component. Instantiating a Verilog module within a C++ simulation is similar: a Cascade interface is used as a wrapper around the Verilog implementation.

Figure 8. Interface binding is used to connect the ports of a C++ implementation to those of a Verilog wrapper (or vice-versa).



The following example shows how a Cascade implementation of the adder depicted in Figure 8 can be instantiated within a Verilog wrapper.

```
// Cascade component
class Adder : public Component
{
     DECLARE_COMPONENT(Adder);
public:
     Adder (COMPONENT_CTOR) {}

     Clock(clk);
     Input(u16,  in_a);
     Input(u16,  in_b);
     Output(u17, out_sum);

     void update ()
     {
          out_sum = in_a + in_b;
     }
};
DECLARE_CMODULE(adder, new Adder);
```

```
// Verilog wrapper
module CAdder
(
     input  logic clk,
     input  logic [15:0] i_a,
     input  logic [15:0] i_b,
     output logic [16:0] o_sum
);

     initial $create_cmodule(“adder”);

endmodule
```

13.1 Interface binding

Cascade's basic method of interface binding is to simultaneously iterate over the ports of a Cascade component and the ports of a Verilog module in order, establishing a binding between corresponding ports. All of the Verilog ports must be bound to a C++ port, however, the Cascade component is permitted to have additional ports beyond the end of the matching ports; these ports are not bound. Additionally, individual C++ ports may be omitted from the binding by calling their `noVerilog()` function from the component constructor.

The order of ports within a Verilog module is simply the linear order in which they appear. For a Cascade component, the order of ports is the order in which the C++ port objects are constructed, so base component/interface ports appear first followed by member interfaces/ports.

In establishing bindings, Cascade compares the C++ and Verilog port types, widths and names in an attempt to detect binding mismatches (which can occur, for example, if the Cascade component and Verilog module list their ports in different orders). In particular, the following rules are applied:

- Cascade Clock and Reset ports (Section 13.4.2) must be bound to single-bit Verilog ports with the exact same names (in particular, this requires the macro variant of the port definitions so that their names are explicitly specified).
- For data ports, the direction must be consistent: input ports must be bound to input ports, output ports must be bound to output ports, and inout ports must be bound to inout ports.
- For data ports, the widths must be consistent. For a C++ type T, the macro `DECLARE_PORT_SIZE(T, size)` can be used to specify the exact bit width for ports of type T. Additionally, the width of a bitvector port is the exact size of the bitvector. In all other cases, `sizeof(T)` is used to validate the width of a port according to the following rules:
  - if sizeof(T) = 1 then the port width must be ≤ 8
  - else if sizeof(T) = 2 then the port width must be ≥ 9 and ≤ 16
  - else if sizeof(T) ≤ 4 then the port width must be ≥ 17 and ≤ 32
  - else 8 * sizeof(T) must be equal to the port width rounded up to a multiple of 32.
- For data ports, the names must be consistent. If the Boolean parameter `cascade.ExactPortNames` is false, then the names must be "similar": they must contain a common substring that is at least two characters long and is surrounded by non-letters. If the parameter is true, then the port names must match exactly with the following exceptions:
  - '.' in Cascade matches '_' in Verilog
  - The prefix 'i_' or 'o_' must appear at the start of the Verilog port name; the matching 'i_', 'o_', 'in_' or 'out_' can appear either at the start of the Cascade name or after a period

If any of these rules are violated then an assertion failure is generated whose message contains the names of the mismatched ports.

13.1.1 Interface binding by name (VPI only)

For VPI simulations only, Cascade makes an initial pass through the Verilog ports looking for exact name matches with C++ ports (using the rules for exact name matching described in the previous section). Any Verilog port whose name exactly matches one and only one C++ port is immediately bound to that C++ port. Once this initial pass is complete, a second pass is made where the remaining ports are bound in order as described in the previous section.

Example

```
// Cascade component
class Adapter : public Component
{
    …
    Clock(clk);

    Input(bit, i_qff_valid);
    Input(u27, i_qff_data);
    Output(bit o_qff_accept);

    Output(bit, o_sfa_valid);
    Output(u9,  o_sfa_data[3]);
    Output(u3,  o_sfa_parity);
    …
};
```

```
// Verilog wrapper
module CAdapter
(
    input logic clk,

    output logic       o_to_sfa_valid,
    output logic [8:0] o_to_sfa_data0,
    output logic [8:0] o_to_sfa_data1,
    output logic [8:0] o_to_sfa_data2,
    output logic [2:0] o_to_sfa_parity,

    output logic        o_qff_accept,
    input  logic [26:0] i_qff_data,
    input  logic        i_qff_valid
);
```

In this example, when using the VPI interface, the clock and `*qff*` ports are bound first because their names match exactly, and they do not need to appear in the same order. The remaining `*sfa*` ports are bound second and must appear in the same order because their names are similar but do not match exactly.

13.2 Data marshalling

By default, data is copied between bound ports as a "bag of bits", where the number of bits is specified by the width of the Verilog port. In particular, if the C++ type of a Cascade port is larger than the width of the Verilog port to which it is bound, then the extra upper bits of the C++ port are not copied to Verilog and are not modified when data is copied from Verilog to C++. This can be desirable, for example, to support C++-only metadata or debug fields after the portion of a structure shared with the Verilog code. Note that increasing the size of the C++ type can violate the port width consistency check described in the previous section, in which case it is necessary to use the `DECLARE_PORT_SIZE` macro to artificially reduce the effective with for ports of this type as follows:

```
struct Flit
{
    uint32 type : 4;
    uint32 addr : 16;
    uint32 data : 12;

    uint32 debug_id;
};

DECLARE_PORT_SIZE(Flit, 32);
```

It is also possible to explicitly specify the manner in which data is copied between bound ports of a given C++ type by creating a *bitmap* for that type. A bitmap is a struct or class which inherits from `IBitmap` and implements its two virtual functions, `mapCtoV` and `mapVtoC`:

```
struct IBitmap
{
    virtual void mapCtoV (IWriteWordArray &dst, const byte *src) = 0;
    virtual void mapVtoC (byte *dst, IReadWordArray &src) = 0;
};
```

`mapCtoV()` is responsible for copying a C++ port value (represented as an array of bytes) to the Verilog port value (represented as an array of 32-bit words). Similarly, `mapVtoC()` copies a Verilog port value to a C++ port value. The interfaces to the array of 32-bit words representing the Verilog port value are defined as follows:

```
struct IReadWordArray
{
    virtual uint32 getWord (int i) = 0;
};
struct IWriteWordArray
{
    virtual void setWord (int i, uint32 w) = 0;
};
```

A bitmap type `bitmap_t` is associated with a C++ port type T using the macro `DECLARE_PORT_BITMAP(T, bitmap_t)`. Bitmaps support arbitrary mapping of data values between C++ and Verilog, as shown in the following example:

```
// C++ defines a NodeId as three integers for simplicity, but Verilog uses a packed
// 18-bit representation with x in bits [5:0], y in bits [11:6], and z in bits [17:12]
struct NodeId
{
     int x;
     int y;
     int z;
};

struct NodeIdBitmap : public IBitmap
{
    void mapCtoV (IWriteWordArray &dst, const byte *src)
    {
        NodeId cid = *(NodeId*)src;
        uint32 vid = (cid.z << 12) | (cid.y << 6) | cid.x;
        dst.setWord(0, vid);
    }
    void mapVtoC (byte *dst, IReadWordArray &src)
    {
        uint32 vid = src.getWord(0);
        NodeId cid = { vid & 0x3f, (vid >> 6) & 0x3f, (vid >> 12) & 0x3f };
        *(NodeId*)dst = cid;
    }
};

DECLARE_PORT_BITMAP(NodeId, NodeIdBitmap);
```

13.3 Port types

By default, a port of type T stores values of type T. It is possible to override this default and explicitly specify a different type for the port values, so that the port template type T is simply a tag used to determine the actual port value type. This is accomplished using the macro `DECLARE_PORT_TYPE(T, value_t)`, where T is the template type and `value_t` is the port value type (i.e. the type that can be assigned to the port and that is returned by the port's read accessors).

This level of indirection between the declared port type and the actual value type allows different port traits to be associated with ports that have the same value type (using the `DECLARE_PORT_SIZE` and `DECLARE_PORT_BITMAP` macros). For example, suppose a 'tag' port is being modeled as an integer, but the actual Verilog port width is specified by a compile-time constant. A dummy type can be declared along with the appropriate traits:

```
struct Tag {};

DECLARE_PORT_SIZE(Tag, TAG_WIDTH);
DECLARE_PORT_TYPE(Tag, int);
```

Ports can then be declared using the dummy `Tag` type:

```
Input(Tag, in_tag);
```

The `in_tag` port above is then modeled as an integer, but has the (exact) width `TAG_WIDTH` for the purpose of Verilog interface binding.

13.4 Instantiating Cascade models within Verilog

Three steps are required to instantiate a Cascade model within a Verilog simulations:

1. Implement the Cascade model.
2. Expose the model to Verilog using the `DECLARE_CMODULE()` macro.
3. Create a Verilog wrapper module with the same interface that instantiates the model using `$create_cmodule()` (VPI) or `createCModule` (DPI).

13.4.1 Clocks and timing model

The Cascade portion of a mixed simulation is driven by rising clock edges, so in particular the Cascade model must have at least one clock (and may have many). A clock bound to a Verilog clock port has the semantics of a manual clock (Section 4.4); on each rising Verilog clock edge the following sequence of events occurs:

1. Input port values are copied from Verilog to Cascade.
2. The clock's `tick()` function is called, causing `update()` functions to be invoked.
3. Output port values are copied from Cascade to Verilog

Due to this behavior, signals in the Cascade→Verilog direction are inherently combinational, whereas signals in the Verilog→Cascade direction are inherently synchronous. The port values copied from Cascade to Verilog are visible to the Verilog simulation on the delta cycle following the rising clock edge. Changes to Verilog signals, however, are not visible to Cascade until the next rising clock edge.

It is still possible to model purely combinational paths by artificially supplying one or more delayed clocks to the Cascade component; this technique explicitly controls when, during the clock cycle, the component `update()` functions are called.

13.4.2 Modeling reset

Reset of a Cascade model within a Verilog simulation is effected by creating one or more *reset ports* and binding them to Verilog reset ports. A reset port is declared as follows:

```
Reset(<level>, <name>)
```

where *<level>* is the reset level as described in Section 2.5, and *<name>* is the name of the reset port. This macro must appear within a public section of the class declaration. When a reset port is bound to a Verilog port and, on a rising clock edge, that Verilog port is high, then instead of calling the component update functions the component is reset. The behavior is as though a call were made to

```
Sim::reset(component, <level>)
```

where *<level>* is as specified in the reset port declaration. If multiple reset ports are high, then `Sim::reset()` is called multiple times: once for each of these ports, supplying the appropriate level as the second argument to `reset()`.

Because `reset()` is called *instead* of the `update()` functions, it is not possible to run a simulation while holding one of the resets high. Also note that if a reset is high for multiple clock cycles, then `Sim::reset()` will be called on each of these clock cycles.

13.4.3 Creating a CModule

When a Cascade component is instantiated within a Verilog testbench, we refer to the component as a *CModule*. The macro `DECLARE_CMODULE` is used to allow a component to be instantiated as a CModule:

```
DECLARE_CMODULE(<name>, <construct>)
```

The first argument is an alphanumeric string (without quotes) specifying the name, which will be used within Verilog code to instantiate the CModule. The second argument is a snippet of C++ code used to create the component; the code must evaluate to a pointer to a new component of the appropriate type. The example following Figure 8 illustrates basic usage of this macro.

In order to support parameterized CModules, Verilog can supply named integer parameters by calling either `$set_cmodule_param()` (VPI) or `setCModuleParam()` (DPI) before the CModule is created. The CModule construction code can then retrieve these parameters using the `param()` function:

```
int param (const char *name);
int param (const char *name, int defaultValue);
```

The first form of the `param()` function is used for required parameters; the second form is used for optional parameters and includes the default value to use if the parameter is not specified from Verilog.

When using VPI, parameters can also be supplied as arguments to `$create_cmodule()`. If `name` is of the form "`k|<actual name>`", where *k* is a non-negative integer, then the parameter can either be specified using `$set_cmodule_param()` or as the kth argument following the CModule name in `$create_cmodule()`.

The following example shows the different ways to create a CModule with parameters using the VPI interface.

```
// AdderN.hpp
class AdderN : public Component
{
    DECLARE_COMPONENT(AdderN);
public:
    AdderN (int numArgs, COMPONENT_CTOR) : in_data(numArgs) {}

    Clock(clk);
    Reset(RESET_COLD, rst);
    InputArray(uint16, in_data);
    Output    (uint16, out_sum);

    void reset ()
    {
        out_sum.reset(0);
    }
    void update ()
    {
        uint16 sum = 0;
        for (int i = 0 ; i < in_data.size() ; i++)
            sum += in_data[i];
        out_sum = sum;
    }
};

DECLARE_CMODULE(addern, new AdderN(param("0|nargs", 2)));

// in Adder2.v
initial $create_cmodule("addern");       // Use default nargs (2)

// in Adder3.v
initial begin
    $set_cmodule_parameter("nargs", 3); // Specify nargs by name
    $create_cmodule("addern");          // Create with nargs = 3
end

// in Adder4.v
initial $create_cmodule("addern", 4); // Specify nargs = 4 as additional argument
```

13.4.4 Determining if a component is a CModule

When a component is intended to be used both as a regular component as well as a CModule and there are behavioral differences between the two, the following component member function can be used to determine the context for a specific instance:

bool isVerilogModule () const

This function returns true if the current instance is a CModule, and false if it is a regular Cascade component.

Example

```
class Router : public Component
{
    DECLARE_COMPONENT(Router);
public:
    Router (COMPONENT_CTOR)
    {
        i_metadata.noVerilog();
    }
    …
    Input(bit,      i_valid);
    Input(Flit,     i_flit);
    Input(Metadata, i_metadata);
    …
    void update ()
    {
        if (i_valid && !isVerilogModule())
            trace("Received packet id %d\n", i_metadata->id);
        …
    }
};
DECLARE_CMODULE(router, new Router);
```

13.4.5 Instantiating a CModule using VPI

CModules can be created using either the VPI or the DPI interfaces. The VPI interface is much easier to use, whereas the DPI interface may provide better performance. To use the VPI interface, create a Verilog shell module with the same ports in the same order as the Cascade component, then call $create_cmodule() from an “initial” block within the module as in the following example:

```
module Adder3
(
     input  logic        clk,
     input  logic        rst,
     input  logic [15:0] i_data0,
     input  logic [15:0] i_data1,
     input  logic [15:0] i_data2,
     output logic [15:0] o_sum
);

initial $create_cmodule(“addern”, 3);

endmodule
```

Parameters should be specified as described in Section 13.4.3; any calls to $set_cmodule_parameter() should appear within the same “initial” block as (and precede) the call to $create_component().

To compile the Verilog design, add ‘-P <path to Cascade>/vpi.tab’ to the VCS command line, and link to descore, Cascade, and the library containing the CModule component.

13.4.6 Instantiating a CModule using DPI

The DPI interface is more complicated than the VPI interface because the rising clock edge must be driven manually, and port values must be explicitly copied back and forth between Verilog and C++.

To use the DPI interface, create a Verilog shell module with the same ports as the Cascade component (this step is the same as for the VPI interface, but in this case the order of the ports is unimportant since they will be explicitly ordered by a sequence of PUSH/POP macros). The module file should include “dpi.svh” (this file is located in the Cascade source directory), and the module body should be based on the following template:

```
chandle cmodule;

initial begin
     string s;
     $sformat(s, “%m”);
     cmodule = createCModule(<name>, s);
end

always_ff @(posedge clk) begin

     // Copy each input port from Verilog to C++
     `PUSH_TO_C(<i_port>);
     `PUSH_TO_C(<i_port>);
     ...

     // Rising clock edge
     `CLOCK_CMODULE(clk);

     // Copy each output port from C++ to Verilog
     `POP_FROM_C(<o_port>);
     `POP_FROM_C(<o_port>);
     ...
end
```

On each rising clock edge, the `always_ff` block performs the following actions:

1. The `` `PUSH_TO_C `` macro is used to copy input port values to C++. Each macro takes a single input port as its argument. The order of these macros must match the order of the input ports in the Cascade module.

2. The `` `CLOCK_CMODULE `` macro is used to drive the rising clock edge. Its single argument is the clock to drive, and must exactly match the name of the clock in the Cascade module.

3. The `` `POP_FROM_C `` macro is used to copy output port values from C++. Each macro takes a single output port as its argument. The order of these macros must match the order of the output ports in the Cascade module.

When using the DPI interface, CModule parameters cannot be specified as extra arguments to `createCModule()`; they must be specified by name using the DPI function `setCModuleParam()` before calling `createCModule()`.

The following code shows the DPI version of the example from the previous section:

```
`include “dpi.svh”

module Adder3
(
     input  logic        clk,
     input  logic        rst,
     input  logic [15:0] i_data0,
     input  logic [15:0] i_data1,
     input  logic [15:0] i_data2,
     output logic [15:0] o_sum
);

chandle cmodule;

initial begin
     string s;
     $sformat(s, “%m”);
     setCModuleParam(“nargs”, 3);
     cmodule = createCModule(“addern”, s);
end

always_ff @(posedge clk) begin
     `PUSH_TO_C(i_data0);
     `PUSH_TO_C(i_data1);
     `PUSH_TO_C(i_data2);
     `CLOCK_CMODULE(clk);
     `POP_FROM_C(o_sum);
end

endmodule
```

13.4.7 Using DPI with multiple clocks

When the DPI interface is used for a CModule with multiple clocks, then multiple `always_ff` blocks are required (one per clock). In the most straightforward implementation, each `always_ff` contains the same set of `` `PUSH_TO_C `` and `` `POP_FROM_C `` macros, and the blocks differ only in the argument supplied to `` `CLOCK_CMODULE ``. This generally results in more data copying than necessary, since most of the input/output ports will be specific to a single clock. If performance is critical, the amount of data copying can be reduced in two ways:

1. The `` `PUSH_TO_C `` macros following the last relevant input can be omitted; similarly the `` `POP_FROM_C `` macros following the last relevant output can be omitted.

2. Of the remaining macros, `` `PUSH_TO_C `` can be replaced with `` `IGNORE_TO_C `` for unneeded inputs, and similarly `` `POP_FROM_C `` can be replaced with `` `IGNORE_FROM_C `` for unneeded outputs.

Example

```
module cdc_fifo
(
     input logic clk_400,
     input logic clk_800,

     input  logic        i_push_400,
     input  logic [15:0] i_data_400,
     output logic        o_full_400,

     input  logic        i_pop_800,
     output logic [15:0] o_data_800,
     output logic        o_empty_800
);

chandle cmodule;

initial begin
     string s;
     $sformat(s, “%m”);
     cmodule = createCModule(“cross_domain_fifo”, s);
end

always_ff @(posedge clk_400) begin
     `PUSH_TO_C(i_push_400);
     `PUSH_TO_C(i_data_400);
     `CLOCK_CMODULE(clk_400);
     `POP_FROM_C(o_full_400);
end

always_ff @(posedge clk_800) begin
     `IGNORE_TO_C(i_push_400);
     `IGNORE_TO_C(i_data_400);
     `PUSH_TO_C(i_pop_800);
     `CLOCK_CMODULE(clk_800);
     `IGNORE_FROM_C(o_full_400);
     `POP_FROM_C(o_data_800);
     `POP_FROM_C(o_empty_800);
end
```

13.5 Instantiating Verilog modules within Cascade

The top level of any co-simulation is always a Verilog module, so instantiating a Verilog module within a Cascade simulation requires some additional steps: in addition to embedding the Verilog module with Cascade, the Cascade simulation as a whole must also be instantiated from a top-level Verilog module. The following steps are required to instantiate a Verilog module within a Cascade simulation (which is only supported via the VPI interface):

1. Implement the Verilog module.

2. Instatiate the module within Verilog, and register the instance by calling `$register_module`, supplying a string name for the instance.

3. Create a Cascade interface that matches the module’s interface.

4. Instantiate a `VerilogComponent` with this interface, passing the string name of the registered module instance to the constructor.

5. Implement the Cascade simulation within a top-level component.

6. Expose the top-level component to Verilog using `DECLARE_CMODULE()`.

7. Run the simulation from the top-level Verilog module by calling `$run_simulation`.

The following example illustrates these steps. A synchronous adder module is instantiated within a Cascade testbench, which is in turn created from a top-level Verilog module.

```
// Verilog module
module adder
(
     input  logic clk,
     input  logic [15:0] i_a,
     input  logic [15:0] i_b,
     output logic [16:0] o_sum
);

always_ff @(posedge clk)
     o_sum <= i_a + i_b;

endmodule

// Top-level module
module top;

  // Instantiate the adder
  adder adder0;

  // Register the adder and run the
  // simulation
  initial begin
    $register_module(adder0, “a0”);
    $run_simulation(“test_add”);
  end

endmodule
```

```
// Cascade interface
struct IAdder : public Interface
{
    DECLARE_INTERFACE(IAdder);

    Clock(clk);
    Input(u16, in_a);
    Input(u16, in_b);
    Output(u17, out_sum);
};

// Testbench
class TestAdder : public Component
{
    DECLARE_COMPONENT(TestAdder);
public:
    TestAdder (COMPONENT_CTOR)
        : m_adder(“a0”)
    {
        clk.generateClock();
        m_adder.clk << clk;
    }

    Clock clk;
    …
private:
    VerilogComponent<IAdder> m_adder;
};
DECLARE_CMODULE(test_add, new TestAdder);
```

13.5.1 Clocks and timing model

On each rising *or* falling clock edge, the following sequence of events occurs:

1. Output port values are copied from Verilog to Cascade.
2. For all *rising* clock edges, Cascade simulates synchronous events followed by combinational evaluation within those clock domains.
3. Input port values are copied from Cascade to Verilog (including the clock).

The Verilog module is driven by the rising and falling clock edges as the value of the clock is copied from Cascade to Verilog. Again, signals in the Cascade→Verilog direction are inherently combinational, whereas signals in the Verilog→Cascade direction are inherently synchronous.

The rules for assigning a `VerilogComponent`’s ports to clock domains are modified from those described in Section 4.5 in two ways. First, output ports are always assigned to their *reader’s* clock domain. Second, the first clock in the `VerilogComponent`’s interface is always taken to be its default clock, and the corresponding clock domain is used for output ports with no known readers and input ports with no known writers.

13.5.2 Modelling reset

No specialized reset support is required when instantiating a Verilog module within a Cascade simulation: the `VerilogComponent` interface simply includes single-bit input ports corresponding to the Verilog module's reset inputs, and the Cascade simulation is responsible for explicitly driving these signals high for the appropriate number of cycles. The update function that drives the reset signal can use `getTickCount()` to determine when the reset port(s) should be deasserted.

Example

```
struct IRegFile : public Interface
{
    Clock(clk);
    Input(bit, rst);
    …
};

class TestRegfile : public Component
{
    DECLARE_COMPONENT(TestRegfile);
public:
    TestRegfile (COMPONENT_CTOR) : m_regfile("regfile")
    {
        clk.generateClock();
        m_regfile.clk << clk;
        m_regfile.rst.setType(PORT_LATCH);
    }

    void reset ()
    {
        m_regfile.rst.reset(1);
    }

    void update ()
    {
        if (getTickCount() > 5)
            m_regfile.rst = 0;
        …
    }
    VerilogComponent<IRegFile> m_regfile;
};
```

13.5.3 Module registration and instantiation

A Verilog module that is to be included within a Cascade simulation must be instantiated within Verilog and then registered using the system task

```
$register_module(<module>, <name>)
```

The first argument references the module instantiation, and the second argument is the string name that will be used within Cascade to retrieve the module. There is no restriction on the number of modules that may be included within a Cascade simulation, but the string names supplied to `$register_module` must all be unique.

Within the Cascade simulation, the templated `VerilogComponent` class is used to instantiate a Verilog module. The single template argument is the Cascade interface that matches the interface of the Verilog module. The single constructor argument is the string name that was supplied to `$register_module`. An assertion failure is generated if no such Verilog module has been registered.

`VerilogComponent` has two static helper functions that allow a simulation to be constructed when the Verilog module may or may not have been registered:

bool VerilogComponent<T>::isModuleRegistered (const char *name)

Returns true if a Verilog module has been registered with the specified name, false otherwise.

T *VerilogComponent<T>::create (const char *name)

If a Verilog module has been registered with the specified name, creates a VerilogComponent with the appropriate port bindings and returns a pointer to the interface. Otherwise, returns NULL.

Example

```
class TestAdder : public Component
{
    DECLARE_COMPONENT(TestAdder);
public:
    TestAdder (COMPONENT_CTOR)
    {
        if (VerilogComponent<IAdder>::isModuleRegistered("a0"))
            m_adder = new VerilogComponent<IAdder>("a0");
        else
            m_adder = new Adder;
    }
    …
private:
    IAdder *m_adder;
};
```

13.5.4 Running the simulation

The top-level simulation must be encapsulated in a Cascade component and instantiated within Verilog in the same manner as described in Section 13.4. The only difference is the use of `$run_simulation` instead of `$create_cmodule` which, in addition to instantiating the Cascade component, starts the Cascade clocks so that they may drive the simulation. The simulation can be parameterized as described in Section 13.4.3, passing parameters via `$set_parameter` or as additional arguments to `$run_simulation`. As shown in the Section 13.5 example, the top-level Cascade component (and the top level Verilog module that instantiates it) need not have any ports, but if they do have ports then the usual interface binding will take place.

The simulation can be terminated either from Verilog, by calling `$finish`, or from Cascade, by calling `tf_dofinish()`.

13.6 Additional functions

The following additional functions are available via both VPI and DPI. The VPI functions require `vpi.tab` on the VCS command line as described in Section 13.4.4; the DPI functions require `` `include "dpi.svh" ``.

VPI: $set_traces (string specifiers)
DPI: setCModuleTraces (string specifiers)

Enable Cascade tracing based on a trace specifiers string. The format of the specifiers string is as described in Section 9.4.

VPI: $dump_cvars (string specifiers)
DPI: dumpCModuleVars (string specifiers)

Enable waves dumping based on a dump specifiers string. The format of the specifiers string is as described in Section 11.0. These functions must be called before the first rising clock edge.

VPI: $disable_assertion (string message)
DPI: disableCAssertion (string message)

Disable one or more C++ assertions. An assertion failure is ignored if any substring of the error output text matches the supplied message string, which may contain the ? and * wildcard characters. So in particular, `$disable_assertion("*")` disables all assertions.

VPI: $set_parameter (string name, string value)
DPI: setCParameter (string name, string value)

Set one of the application parameters (see descore Parameter documentation) where the value is specified as a string.

13.6.1 Command-line interface

When the file `plusargs.v` in the Cascade source directory is included in the Verilog build, the functions described in the previous section also become available from the command line via plusargs. The module within `plusargs.v` parses the command line within an `initial` block, and recognizes the following plusargs:

+trace+<specifiers>

Enable Cascade tracing based on a trace specifiers string. The format of the specifiers string is as described in Section 9.4.

+dump_cvars+<specifiers>

Enable waves dumping based on a dump specifiers string. The format of the specifiers string is as described in Section 12.7.

+disable_assertions+message1+message2+…

Disable one or more C++ assertions. One or more message strings may be supplied delimited by '+'. Equivalent to calling `$disable_assertion` with each individual message string in turn.

+setparams+param1=value1+param2=value2+…

Set one or more application parameters (see descore Parameter documentation), where each setting is of the form `param=value`, and the settings are delimited by '+'. Equivalent to calling `$set_parameter` with each individual parameter setting in turn.

Each of these plusargs may appear at most once on the command line. To include special characters (space, comma, semicolon, etc), either escape them with a backslash, or enclose them in quotes, e.g.

```
+setparams+Subboxing="(1,2,4)"
```

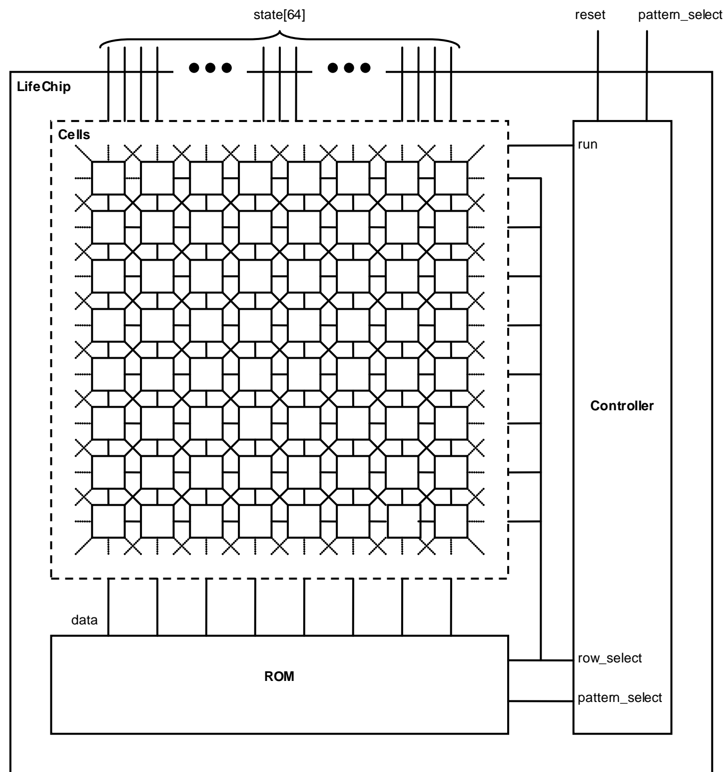
14.0 Example: Conway's Game of Life

Conway's game of Life is a cellular automata invented by John Conway. The game consists of a two dimensional array of cells, usually either infinite in both directions or on a finite torus. Each cell is in one of two states: alive or dead. At each timestep, a dead cell comes to life if it has exactly three live neighbors (counting all eight neighbors horizontally, vertically and diagonally), and a live cell dies if it has fewer than two or greater than three live neighbors. In this section we present the design and complete simulation source code for a simple Life chip which plays the game of life on an 8×8 torus.

14.1 Life chip design

Figure 9 shows the components of the life chip. An 8×8 array of cells performs the actual computation; each cell is connected to its eight neighbors (including wraparound connections at the edge of the torus). A 256 bit ROM holds four initialization patterns and sends one row (8 bits) at a time to the cell array. Finally, a controller directs initialization and computation with a simple state machine. When the external *reset* signal is asserted, the controller latches *pattern_select* and initializes one row of cells per cycle. On each initialization cycle a row of data is read from the ROM using *row_select* and *pattern_select*; the *row_select* signal is also decoded and used to cause a row of cells to store the data being sent from the ROM. During initialization the *run* signal is low; once initialization completes the *run* signal is asserted which causes the cells to stop storing data and starting running the game of life. The life chip has 64 single-bit outputs, one for each cell state.

Figure 9. Life chip block diagram.



14.2 ROM source code

The ROM is a purely combinational component which always outputs 8 bits based on the *row_select* and *pattern_select* inputs. The source code for the ROM is as follows:

```
class ROM : public Component
{
    DECLARE_COMPONENT(ROM);
public:
    ROM (COMPONENT_CTOR) {}

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(u2, in_pattern_select);
    Input(u3, in_row_select);

    Output(bit, out_data[8]);

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void update ()
    {
        for (int i = 0 ; i < 8 ; i++)
            out_data[i] = s_pattern[in_pattern_select][8*in_row_select + i];
    }

    void archive (Archive &) {}

private:
    static u64 s_pattern[4];
};

u64 ROM::s_pattern[4] =
{
    // glider
    0x000008101c000000LL,

    // spaceship
    0x0000784440240000LL,

    // random 1
    0x1bded76ace9c0f32LL,

    // random 2
    0x59a0203ce90a21caLL
};
```

Two input bits are required to select one of four patterns and 3 input bits are required to select one of eight rows. Each pattern is stored as a 64-bit bitvector; the `update()` function extracts the appropriate bits (taking advantage of the bit vector indexing operator) and sets the output data.

14.3 Cell source code

Each cell is a systolic component with a single bit output (its state). A cell has four different inputs: an *initVal* input used to set the cell's state from the ROM output, an *initialize* signal which indicates that the cell should store *initVal*, a *run* signal which indicates that the cell should update its state, and eight single-bit *neighbour* inputs. The source code for the `Cell` component is as follows, including a sample trace statement:

```
class Cell : public Component
{
    DECLARE_COMPONENT(Cell);
public:
    Cell (COMPONENT_CTOR);

    //---------------------------------------
    // Interface
    //---------------------------------------
    Input(bit, in_initialize);
    Input(bit, in_initVal);
    Input(bit, in_run);
    Input(bit, in_neighbour[8]);
    Output(bit, out_state);

    //---------------------------------------
    // Simulation
    //---------------------------------------
    void update ();
    void archive (Archive &ar) {}
    void reset ()
    {
        out_state.reset(0);
    }
};

////////////////////////////////////////////////////////////////////////
//
// Cell::Cell()
//
////////////////////////////////////////////////////////////////////////
Cell::Cell (IMPL_CTOR)
{
    out_state.setType(PORT_LATCH);
}

////////////////////////////////////////////////////////////////////////
//
// Cell::update()
//
////////////////////////////////////////////////////////////////////////
void Cell::update ()
{
    if (in_initialize)
    {
        out_state = in_initVal;
    }
    else if (in_run)
    {
        int count = 0;
        for (int i = 0 ; i < 8 ; i++)
            count += in_neighbour[i];
        trace("state = %d, count = %d\n", (int) out_state, count);
        out_state = ((count | out_state) == 3) ? 1 : 0;
    }
}
```

14.4 Controller source code

The controller processes the external inputs and generates control signals for the ROM and Cell components. It has three outputs used during array initialization (*initialize, row_select* and *pattern_select*) and a single-bit *run* signal which is asserted after initialization. The controller source code is as follows:

```
class Controller : public Component
{
    DECLARE_COMPONENT(Controller);
public:
    Controller (COMPONENT_CTOR);

    //---------------------------------------
    // Interface
    //---------------------------------------
    Input(u2,  in_pattern_select);

    Output(u3,  out_rom_row_select);   // row select index
    Output(bit, out_cell_row_init[8]); // 1-hot row select
    Output(u2,  out_pattern_select);
    Output(bit, out_run);

    //---------------------------------------
    // Simulation
    //---------------------------------------
    void update ();
    void archive (Archive &ar) {}
    void reset ()
    {
        m_cycleCount = 0;
        out_run.reset(0);
        out_pattern_select.reset(*in_pattern_select);
        out_rom_row_select.reset(7);
    }

private:
    int m_cycleCount;
};

////////////////////////////////////////////////////////////////////////
//
// Controller::Controller()
//
////////////////////////////////////////////////////////////////////////
Controller::Controller (IMPL_CTOR)
{
    out_rom_row_select.setType(PORT_LATCH);
    out_pattern_select.setType(PORT_LATCH);
}

////////////////////////////////////////////////////////////////////////
//
// Controller::update()
//
////////////////////////////////////////////////////////////////////////
void Controller::update ()
{
    out_run = (m_cycleCount >= 8);
    out_rom_row_select = m_cycleCount & 7;
    for (int i = 0 ; i < 8 ; i++)
        out_cell_row_init[i] = (m_cycleCount == i) ? 1 : 0;
    m_cycleCount++;
}
```

14.5 Life chip source code

The LifeChip component is the top-level component which represents the entire chip and contains the ROM, Cell and Controller components. It is declared as follows:

```
class LifeChip : public Component
{
    DECLARE_COMPONENT(LifeChip, Chip);
public:
    LifeChip (COMPONENT_CTOR);

    //----------------------------------------
    // Interface
    //----------------------------------------
    Input(u2, in_pattern_select);

    Output<bit> out_state[8][8];

    //----------------------------------------
    // Simulation
    //----------------------------------------
    void archive (Archive &ar) {}

private:
    ROM m_rom;
    Array<Cell> m_cells;
    Controller m_controller;
};
```

The LifeChip component has the most complicated constructor because it is responsible for establishing all on-chip connections. Since there is no functionality at the top level, no update() or reset() functions are required. The source code for the LifeChip constructor is as follows:

```
////////////////////////////////////////////////////////////////////////
//
// LifeChip::LifeChip()
//
////////////////////////////////////////////////////////////////////////
LifeChip::LifeChip (IMPL_CTOR) : m_cells(8, 8)
{
    static int dx[8] = {1, 1, 0, -1, -1, -1, 0, 1};
    static int dy[8] = {0, 1, 1, 1, 0, -1, -1, -1};

    // Connect controller inputs
    m_controller.in_pattern_select << in_pattern_select;

    // Connect ROM inputs
    m_rom.in_pattern_select << m_controller.out_pattern_select;
    m_rom.in_row_select << m_controller.out_rom_row_select;

    // Connect Cell inputs and LifeChip outputs
    for (int x = 0 ; x < 8 ; x++)
    {
        for (int y = 0 ; y < 8 ; y++)
        {
            out_state[x][y] <= m_cells(x, y).out_state;

            m_cells(x, y).in_initialize << m_controller.out_cell_row_init[y];
            m_cells(x, y).in_initVal << m_rom.out_data[x];
            m_cells(x, y).in_run << m_controller.out_run;

            // neighbours
            for (int i = 0 ; i < 8 ; i++)
                m_cells(x, y).in_neighbour[i] <=
                        m_cells((x+dx[i])&7, (y+dy[i])&7).out_state;
        }
    }
}
```

14.6 Main program source code

The `main()` function starts the simulation then enters a command prompt loop. The source code is as follows:

```
int main (int csz, char *rgsz[])
{
    descore::parseTraces(csz, rgsz);
    Parameter::parseCommandLine(csz, rgsz);

    LifeChip chip;
    Sim::init();
    chip.in_pattern_select = 0;

    while (1)
    {
        char buff[64];
        printf("> ");
        fgets(buff, 64, stdin);
        if (*buff == 'q')
            return 0;
        if (*buff == 'l')
        {
            SimArchive::loadSimulation("life.dat");
            printf("Simulation restored from life.dat\n");
        }
        else if (*buff == 's')
        {
            SimArchive::saveSimulation("life.dat");
            printf("Simulation saved to life.dat\n");
            continue;
        }
        else
        {
            if (*buff >= '0' && *buff <= '3')
            {
                chip.in_pattern_select = *buff - '0';
                Sim::reset();
            }
            Sim::run();
        }
        for (int y = 7 ; y >= 0 ; y--)
        {
            for (int x = 0 ; x < 8 ; x++)
                printf("%c", chip.out_state[x][y] ? 'o' : '.');
            printf("\n");
        }
    }
}
```

The first two lines parse the arguments to `main()` so that the command line can be used to specific traces and parameter settings.

After the `LifeChip` is created, `Sim::init()` is called explicitly so that the input port `in_pattern_select` may be assigned directly. The command prompt loop is then entered which accepts simple single-character commands: ‘q’ to quit, ‘l’ to load the simulation from “life.dat”, ‘s’ to save the simulation to “life.dat”, ‘0’–‘3’ to reset the LifeChip with the specified pattern, and <enter> to advance the simulation by a single clock tick. The state of the simulation is displayed after loading and after each clock tick as an 8×8 array of characters with dead cells represented by ‘.’ and live cells represented by ‘o’.

15.0 Reference

15.1 Component functions

`void activate ()`

Activates the component, enabling subsequent calls to update functions.

`void deactivate ()`

Deactivates the component, disabling subsequent calls to update functions.

`bool isActive ()`

Returns true if the component is active, false otherwise.

`strbuff getName ()`

Returns the full hierarchical component name.

`strbuff getLocalName ()`

Returns the component name without prepending the names of its parents.

`void setTrace (const char *traceName = NULL)`

Enable anonymous tracing or the specified named trace.

`void unsetTrace (const char *traceName = NULL)`

Disable anonymous tracing or the specified named trace.

`int getClockPeriod () const`

Return the clock period in picoseconds. Must be called from an update function to ensure correctness (always returns the clock period of the currently active clock).

`int getTickCount () const`

Returns the number of rising clock edges, including the current one. Must be called from an update function to ensure correctness (always returns the tick count for the currently active clock).

`bool isVerilogModule () const`

Returns true if this component has been instantiated as a CModule.

```
void scheduleEvent (int delay, fn_t fn)
void scheduleEvent (int delay, fn_t fn, A a1)
void scheduleEvent (int delay, fn_t fn, A a1, A a2)
void scheduleEvent (int delay, fn_t fn, A a1, A a2, A a3)
void scheduleEvent (int delay, fn_t fn, A a1, A a2, A a3, A a4)
```

Schedule an event within the component for `delay` rising clock edges in the future. `delay` must be postivie; a delay of 1 schedules the event for the next rising clock edge. `fn` can be an arbitrary member function with 0–4 arguments; the arguments `a`*n* will be passed to `fn` when it is called.

The following additional functions are called automatically by Cascade.

`void update ()`

Default update function that is automatically registered and called after every rising clock edge when the component is active. Additional update functions may be explicitly registered using the `UPDATE()` macro (Section 2.4).

`void tick ()`

If present, this function is automatically called on every rising clock edge when the component is active.

`virtual void archive (Archive &ar)`

Archive the component state; called automatically when a simulation is archived. The default implementation generates an assertion failure, so all components must implement this function in a simulation that supports archiving.

`void reset ()`
`void reset (int level)`

Called automatically when the simulation is reset. Either form may be used; the current reset level is passed as an argument to the second form. If both forms are present then both will be called. `reset()` may be called multiple times, so all implementations must be idempotent (i.e. multiple calls must have exactly the same effect as a single call).

15.2 Interface functions

`void reset ()`
`void reset (int level)`

Called automatically when the simulation is reset. Same as component reset function; must be idempotent.

`Component *getComponent ()`

Returns the parent component of the interface (the component that inherits from or contains this interface).

`strbuff getName ()`

Returns the full hierarchical interface name.

15.3 Port functions

`void wireTo (const T &data)`

Wire the port directly to a variable of the same type. The port becomes read-only. Can only be called during construction.

`void wireToConst (T data)`

Wire the port directly to a constant of the same type. The port becomes read-only. Can only be called during construction.

```
void setType (PortType type)
```

Set the port type; `type` must be one of `PORT_NORMAL`, `PORT_LATCH` or `PORT_PULSE`. Can only be called during construction.

```
PortType getType () const
```

Returns the current port type. Can only be called during construction.

```
void setDelay (int delay)
```

Set the delay in clock cycles for synchronously-connected ports. If delay is zero this function has no effect. Can only be called during construction.

```
void activates (Component *target,
                PortActivationType activationType = ACTIVE_HIGH)
```

Specify a component that should be activated whenever this port is non-zero (`activationType = ACTIVE_HIGH`) or zero (`activationType = ACTIVE_LOW`). Can only be called during construction.

```
void addTrigger (ITrigger<T> *trigger,
                 PortActivationType activationType = ACTIVE_HIGH)
```

Specify a trigger function that should be called whenever this port is non-zero (`activationType = ACTIVE_HIGH`) or zero (`activationType = ACTIVE_LOW`). Can only be called during construction.

```
void noVerilog ()
```

Exclude this port when binding to a Verilog interface. Can only be called during construction.

```
void reset (const T &data)
```

Reset a port. Use this function instead of assignment within component/interface `reset()` functions.

```
T operator= (const T &data)
```

Assign a value to the port. In debug builds, sets the port valid flag.

```
T *nonConstPtr ()
```

Return a pointer to the port value. In debug builds, sets the port valid flag.

```
operator T () const
```

Automatic conversion to a value of the appropriate type. In debug builds, verifies the valid flag.

```
T &operator() () const
T &operator* () const
```

Explicit conversion to a value of the appropriate type. In debug builds, verifies the valid flag.

```
const T *constPtr () const
```

Return a const pointer to the port value. In debug builds, verifies the valid flag.

`const T *operator-> () const`

For ports whose value is a structure, access one of the structure elements using the arrow operator. In debug builds, verifies the valid flag.

```
T operator+= (const T &data)
T operator-= (const T &data)
T operator*= (const T &data)
T operator/= (const T &data)
T operator%= (const T &data)
T operator&= (const T &data)
T operator|= (const T &data)
T operator^= (const T &data)
T operator>>= (int shift)
T operator<<= (int shift)
```

Read-modify-write accessors. In debug builds, verifies the valid flag.

`void setValid ()`

In debug builds, sets the valid flag without modifying the port value. Does nothing in release builds.

`void dontCare ()`

In debug builds, sets the valid flag and fills the port value with garbage. Used for ports whose values will be read but discarded. Does nothing in release builds.

`strbuff getName () const`

Returns the full hierarchical name of the port. This function is slow due to the need to locate the port within the component hierarchy.

15.4 Fifo port functions

`void setSize (int size)`

Set the fifo capacity. Can only be called during construction.

`void setDelay (int delay)`

Set the fifo delay in clock cycles. Can only be called during construction.

`void sendToBitBucket ()`

Indicate that the fifo has no consumer. Calls to `full()` will always return false; any data pushed onto the fifo will be silently dropped. Can only be called during construction.

`void wireToZero ()`

Indicate that the fifo has no producer. Calls to `empty()` will always return true. Can only be called during construction.

`void disableFlowControl ()`

Disable the simulated credit return path. Calls to `full()` and `freeCount()` are not allowed when flow control has been disabled. Can only be called during construction.

`void setTrigger (ITrigger<T> *trigger)`

Specify a trigger function that should be called when data becomes available at the consumer. Calls to `empty()`, `popCount()` and `pop()` are not allowed if a trigger has been specified. Only one trigger may be specified per fifo. Can only be called during construction.

`bool empty () const`

Returns true if there is no data available at the consumer, false otherwise.

`bool full () const`

Returns true if the fifo appears full to the producer, false otherwise.

`int popCount () const`

Returns the number of data elements currently available at the consumer.

`int freeCount () const`

Returns the number of free fifo entries currently visible to the producer.

`void push (const T &data)`

Push a value onto the fifo.

`const T &pop ()`

Pop a value off of the fifo. The reference is guaranteed to be valid until at least the next push.

`const T &peek () const`

Peek at the next value without popping it. The reference is guaranteed to be valid until at least the next push.

`int highWaterMark () const`

Return the maximum number of elements that have been in the fifo at once since the fifo was last reset.

`strbuff getName () const`

Returns the full hierarchical name of the port. This function is slow due to the need to locate the fifo port within the component hierarchy.

15.5 Parameters

Each of the following parameters, which are listed with their types and default values, can be accessed from the code as members of the structure `Cascade::params` or from the command line using the prefix "`cascade.`".

unsigned CheckpointInterval = 0

Simulated time (ns) between archive checkpoints, or 0 to disable checkpoints.

string CheckpointName = "sim"

Base name of checkpoint files (full name is `<name>_<time>.ckp`).

unsigned ClockRounding = 5

Rising clock edges within this many picoseconds of an even number of nanoseconds will be rounded to the even number of nanoseconds.

unsigned DefaultClockPeriod = 1000

Default clock period in picoseconds.

string DumpSignals = ""

Specify a set of signals to dump using the same format as the `-dump` command-line directive

bool ExactPortNames = false

When binding to a Verilog port, require that the port name (appropriately translated) matches exactly.

bool FifoSizeWarnings = true

Print a warning message if a fifo size is too small to sustain full throughput.

unsigned Finish = 0

If non-zero, end the simulation and exit at the specified time (in ns).

int MaxResetIterations = 10

If greater than one, calls to `reset()` will iterate until all output ports quiesce, up to the specified maximum number of iterations. An error is generated if the output ports are still changing after the maximum number of iterations.

int NumThreads = 1

Number of threads to use for the simulation. Set to -1 to use the maximum number of threads.

string RestoreFromCheckpoint = ""

Checkpoint file from which simulation should be restored after initialization.

bool SafeCheckpoint = false

Create archive checkpoints in safe mode.

unsigned Timeout = 0

If non-zero, abort the simulation with an error at the specified timeout (in ns).

`unsigned TraceStartTime = 0`

Tracing is disabled before this time, measured in ns.

`unsigned TraceStopTime = 0xffffffff`

Tracing is disabled after this time, measured in ns.

`string Traces = ""`

Specify a set of traces using the same format as the `-trace` command-line directive.

`string ValidateCheckpoint = ""`

Secondary checkpoint file against which first checkpoint file should be validated.

`bool Verbose = false`

Display additional information during initialization.

`unsigned WavesDT = 10`

Minimum time increment (ps) between succcessive times in the VCD file.

`string WavesFilename = "sim.vcd"`

Filename of VCD file (used for generating waves).

`string WavesTimescale = "1 ps"`

Timescale string for VCD file.

15.6 Rising clock edge behaviour

On a rising clock edge, the following actions are performed in order:

1. Simulation time is advanced to the time of the rising clock edge.
2. A checkpoint file is created if the simulation time has reached the next checkpoint time.
3. Once every 10 seconds, Cascade traverses the entire hierarchy searching for deactivated components with non-empty input fifos. The simulation is aborted if any are found.
4. `tick()` is called for any component that implements this function.
5. Register values are copied.
6. In debug builds, port valid flags are updated.
7. Pulse ports are reset to zero.
8. Events scheduled for this cycle are executed. This includes:
    a. trigger functions for fifos whose data becomes visible on the rising clock edge,

   b. trigger functions associated with synchronous ports that become active on this cycle, and

   c. events that were manually scheduled for this cycle.

9. Combinational evaluation is simulated by calling `update()` functions.

10. The VCD file is updated if waves are being generated.

15.7 Recommended coding standards

There are two primary coding standards recommendations that will improve the legibility of simulation code and help to prevent certain errors: structured header files and a naming convention for inputs/outputs.

15.7.1 Component header files

All component header files should have the following sections clearly marked:

1. **Construction** – Any non-trivial constructors/destructors/helpers.

2. **Interface** – Explicitly declared inputs and outputs.

3. **Simulation** – `reset()`, `archive()`, `tick()` and update/helper functions.

These sections have been illustrated in examples throughout this document.

15.7.2 Naming of inputs/outputs

All Inputs, Outputs and InOuts should be given prefixes which clearly indicate port direction: ‘`in_`’ or ‘`i_`’ for Inputs, ‘`out_`’ or ‘`o_`’ for outputs, and ‘`io_`’ for InOuts. This coding standard has also been illustrated in examples throughout this document.