BSV: A Simple Configuration Bus

March 3, 2009

1 Introduction

This tutorial explores the use of Bluespec SystemVerilog^{TM} (BSV) in developing a simple configuration bus capability. The BSV library includes a powerful full-featured configuration bus capability (the `LBus` package). Through the use of a simplified example, this tutorial aims to explain the BSV features used in that package so that BSV users may make better use of that package, personalize that package based on their specific needs, or develop other similar capabilities. Portions of this document have been copied or otherwise derived from the documentation of the `LBus` package. The examples presented in this tutorial demonstrate use of the `ModuleCollect` library package and explore issues related to rule and method conflicts and how to resolve them.

2 Control Status Registers

Complex designs typically include a back door mechanism for reading and writing the values of certain *control status* registers contained in that design. Each register that is to be accessed in this way has a unique associated address. It is then possible to access and modify the value of these registers in much the same way that a memory subsystem is accessed (via addressed read and write operations).

When a design is specified at the RTL level, the design is often structured such that all the control status registers are instantiated inside dedicated modules. This allows the required logic for configuration bus address decoding and such to be separated from the main design. In contrast, BSV's `ModuleCollect` mechanism allows the *plumbing* associated with the configuration bus to be automatically added without any need to specify it directly in the main design. There is thus no need to change the natural structure of a design in order to support a configuration bus capability.

3 The ModuleCollect Package

The simple configuration bus package presented in this tutorial is called the `CBus` package. This package as well as the more full-featured `LBus` package both make use of the `ModuleCollect` library package. As mentioned in the previous section, use of this package allows the configuration bus connections to be collected together in a way which does not clutter up the main design. A brief description of the `ModuleCollect` mechanism follows.

An ordinary `BSV` module, when instantiated, adds its own items to the growing accumulation of (1) state elements, and (2) rules, which are used in later stages of the compilation process. The `ModuleCollect` package allows other items to be accumulated as well. This capability is just what is required in order to automatically collect up the interfaces of any control status

registers included in a module and add the associated logic and ports required to allow them to be accessed via a configuration bus.

There are two mechanisms provided by the `ModuleCollect` package that allow this collecting of items to be accomplished. The first is a function that is used to add a given item to the current *collection*. In the course of evaluating a module body during its instantiation, an item `x` may be added to the collection by the call

```
addToCollection(x);
```

Once a set of items has been collected, those items must be exposed and processed. In the case of a configuration bus implementation, the items being collected are the interfaces of control status registers and the processing to be done involves adding the logic such that all the registers collected in a given module can be accessed via a single read/write interface. In order to do this processing, the collected interfaces must first be *exposed*. When such processing is to be performed, the collection may be brought into the open by the following module instantiation:

```
IWithCollection#(IfcType, ItemType) ifc();
exposeCollection#(mkDut) the_dut(ifc);
```

Here `IfcType` is the interface type for the user's design; `ItemType` is the type of item being collected. `ifc` is the interface provided by `the_dut`, the instantiation of the `exposeCollection` module. It has two sub-interfaces: `ifc.device` is the interface of the `mkDut` module (of type `IfcType`) as designed by the user, and `ifc.collection` is a list of the collected items, available for further handling. These concepts will become more clear when we look at them being used in the implementation of the `CBus` package.

A module which is participating in the accumulation of a special collection has a special type (the exact type depends on the type of objects being collected). An ordinary module, not collecting anything special, has the "vanilla" type, `Module`. But for a module accumulating a collection, the type must be explicitly given, and it is supplied in square brackets, immediately after the `module` keyword, in the following way:

```
module [MyModuleType] mkSubDesign#(x, y) (IfcType);
   ...
endmodule
```

It should be noted that modules with a special type (i.e. not of type `Module`) are not synthesizable. This is not surprising since the compiler will not know by default how to process the items being collected. The module type of `exposeCollection` is `Module`, however; so once the collection processing has been handled, the design is available for synthesis as usual. With the `CBus` package presented in this tutorial (as well as with the `LBus` package), all direct use of the `ModuleCollect` capabilities are contained within the package itself (either `CBus` or `LBus`). Users of of these packages need not deal with the `ModuleCollect` package directly.

4 The CBus Package

This section describes the features and implementation details of the `CBus` package. The source code for this package can be found in the example files that accompany this tutorial.

At its essence, implementing a configuration bus capability involves creating a hierarchical tree structure to provide access to all the control and status registers in a given module hierarchy. Clearly, there are many different ways to implement this structure. The `LBus` package implements a sophisticated bus protocol which provides different configuration bus interfaces at different levels of the hierarchy. For instance, the interface provided by a sub-module including many control status registers is different than the interface provided by a single control status register.

Since the `CBus` package is intended as a learning aide, the implementation strategy used in this package is much more straightforward. More specifically, a single configuration bus interface (the `CBus` interface) is used at all levels of the hierarchy. The `CBus` interface provides `read` and `write` methods to access control status registers. It is polymorphic in terms of the size of the address bus (`sa`) and the size of the data bus (`sd`).

```
interface CBus#(type sa, type sd);
   method Action           write(Bit#(sa) addr, Bit#(sd) data);
   method Maybe#(Bit#(sd)) read(Bit#(sa) addr);
endinterface
```

In order to allow this interface to be used at all levels of hierarchy, the implementation strategy is as follows.

At the leaf level (i.e. at the level of a single control status register), the behavior of the `write` method is it to write the `data` value to the register if and only if the value of `addr` matches the address of the register. Similarly, the `read` method returns the value of the associated register if and only if `addr` matches the register address. In all other cases, the `read` method returns an `Invalid` value;

At higher levels of the hierarchy, the `write` method broadcasts the write Action to all included sub-modules (and indirectly to all the included control status registers). Note however that the `write` method will only have an effect on the one register that has the appropriate address. The `read` method collects the returned values from all the sub modules, returning the appropriate `Valid` value (if there is one).

The only other interface defined in the package is the `IWithCBus` interface. This interface is used to couple the `CBus` interface with a *normal* module interface. It is defined as a structured interface with two sub-interfaces. `cbus_ifc` is the associated configuration bus interface while `device_ifc` is the associated normal interface. It is polymorphic in terms of the type of the configuration bus interface and the type of the normal design interface.

```
interface IWithCBus#(type cbus_IFC, type device_IFC);
```

```
   interface cbus_IFC cbus_ifc;
   interface device_IFC device_ifc;
endinterface
```

The package also defines a type for the items to be collected, `CBusItem` (which in this case is simply a synonym for the `CBus` interface type).

```
typedef CBus#(sa, sd) CBusItem #(type sa, type sd);
```

The module type `ModWithCBus` is also defined (a type for modules collecting `CBusItem`s).

```
typedef ModuleCollect#(CBusItem#(sa, sd), i)
        ModWithCBus#(type sa, type sd, type i);
```

Now that we have all the types and interfaces defined, we need to provide the functionality to collect `CBusItem`s as well as to process them in order to provide an `IWithCBus` interface. In the `CBus` package, the `collectCBusIFC` module wrapper is used to add a `CBusItem` to the current collection.

```
module [ModWithCBus#(sa,sd)]
       collectCBusIFC#(Module#(IWithCBus#(CBus#(sa, sd), i)) m)
                      (i);

   IWithCBus#(CBus#(sa, sd), i) double_ifc();
   liftModule#(m) _temp(double_ifc);

   addToCollection(double_ifc.cbus_ifc);

   return(double_ifc.device_ifc);
endmodule
```

This module takes as an argument a module with an `IWithCBus` interface, adds the associated `CBus` interface to the current collection (using the `addToCollection` function), and returns a module with just the normal interface. Note that `collectCBusIFC` is of module type `ModWithCBus` (which it must be in order to add an interface to the current collection).

Similarly, the `exposeCBusIFC` module is used to create an `IWithCBus` interface given a module with a normal interface and an associated collection of `CBusItem`s.

```
module [Module] exposeCBusIFC#(ModWithCBus#(sa, sd, i) sm)
                               (IWithCBus#(CBus#(sa, sd), i));
    ...
endmodule
```

This module takes as an argument a module (of type `ModWithCBus`) and provides an interface of type `IWithCBus`. In more concrete terms, it is the `exposeCBusIFC` module that exposes the collected `CBusItem`s, processes them, and provides a new combined interface. If we look

at the implementation of this module, we see that it first uses `exposeCollection` (from the `ModuleCollect` package) to provide a handle on the collected `CBus` interfaces.

```
IWithCollection#(CBusItem#(sa, sd), i) collection_ifc();
exposeCollection#(sm) _temp(collection_ifc);

let item_list = collection_ifc.collection;
```

Next, the functionality of the `read` and `write` methods of the provided `CBus` interface is defined. The action of the write method involves simply mapping the write method invocation to all all the collected interfaces

```
method Action write(Bit#(sa) addr, Bit#(sd) data);

    function ifc_write(item_ifc);
       action
          item_ifc.write(addr, data);
       endaction
    endfunction

    /// write to all collected interfaces
    joinActions(map(ifc_write, item_list));
endmethod
```

Similarly, the `read` method applies a `read` method invocation to all of the the collected interfaces and then *fold*s together the results.

```
method Maybe#(Bit#(sd)) read(Bit#(sa) addr);

    function ifc_read(item_ifc);
       return item_ifc.read(addr);
    endfunction

    /// fold together the read values for all the collected interfaces
    let vs = map(ifc_read, item_list);
    return(foldt(fold_maybes, Invalid, vs));
 endmethod
```

Finally in the `CBus` package, there is a definition for one flavor of control status register. The register is defined in two parts, first there is a module definition (`regRW`) with an `IWithCBus` interface.

```
module regRW#(Bit#(sa) reg_addr, r reset)(IWithCBus#(CBus#(sa, sd), Reg#(r)))
   provisos (Bits#(r, sr), Add#(k, sr, sd));

   Reg#(r) x();
```

```
   mkReg#(reset) inner_reg(x);

/// Interface and method code

endmodule
```

This module implements the basic functionality of the register (both in terms of the `CBus` interface and the `Reg` interface). There is also an associated module wrapper (`mkCBRegRW`) which uses `collectCBusIFC` and provides a normal register interface.

```
module [ModWithCBus#(sa, sd)] mkCBRegRW#(Bit#(sa) reg_addr, r x)(Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, sd));
   let ifc();
   collectCBusIFC#(regRW(reg_addr, x)) _temp(ifc);
   return(ifc);
endmodule
```

The `mkCBRegRW` module is instantiated inside user designs just as other registers would be. Note that when instantiated, the module takes two arguments, an associated configuration bus address (`reg_addr`) as well as a reset value. The `LBus` package defines many different flavors of control status registers (with *read only* functionality, *write one on clear* etc.). For the purposes of this tutorial, a single register type will suffice.

5 An Example Using The CBus

In this section, we present an example in which we add configuration bus capabilities to a design. The BSV code presented here can be found in the various `CBusExample` packages that accompany this tutorial. The configuration bus definitions in the `CBus` package are polymorphic and thus we start by defining some type definitions for a configuration bus with particular address and data sizes.

```
typedef 10 CBADDRSIZE; //size of configuration address bus to decode
typedef 32 CBDATASIZE; //size of configuration data bus

typedef ModWithCBus#(CBADDRSIZE, CBDATASIZE, i) MyModWithCBus#(type i);
typedef CBus#(CBADDRSIZE, CBDATASIZE) MyCBus;
```

The example design is a simple counter module. The `Counter` interface definition is given below.

```
interface Counter#(type size_t);
   method Bool   isZero();
   method Action decrement();
   method Action load(Bit#(size_t) newval);
```

```
endinterface
```

The interface includes a `isZero` value method which returns `True` when the counter reaches zero. There are also three Action methods (`decrement` and `load`) which are used to modify the current value of the counter. The basic structure of the counter module (without configuration bus registers) is given below.

```
module mkCounter (Counter#(size_t));
   Reg#(Bit#(size_t)) counter <- mkReg(0);
   // code for methods
endmodule
```

A modified version of `mkCounter` which includes a configuration register is given below. In addition to changing the register instantiation to a configuration register (`mkCBRegRW`), note that the module type has been changed to `MyModWithCReg`.

```
module [MyModWithCReg] mkCounter(Counter#(size_t))
   provisos(Add#(size_t, k, CBDATASIZE)); // this provisos ensures the register
                                          // size is no larger than the
                                          // data size of the configuration bus
   Reg#(Bit#(size_t)) counter <- mkCBRegRW(13, 0); // instantiate a configuration
                                                   // register with address 13.
   // code for methods
endmodule
```

In order to access the configuration bus (and to make the module synthesizable), we use `exposeCBusIFC` to create a module (of type `Module`) which provides an `IWithCbus` interface.

```
(* synthesize *)
module mkCounterSynth(IWithCBus#(MyCBus, Counter#(8)));
   let ifc();
   exposeCBusIFC#(mkCounter) _temp(ifc);
   return (ifc);
endmodule
```

If we look at the synthesized Verilog for this module, we see that the port list includes the signals required for the `Counter` interface as well as those for the `CBus` interface. Finally, the example includes a simple testbench module, `mkCBusExample`.

`mkCBusExample` includes an instantiation of a counter module

```
let counter_ifc();
mkCounterSynth the_counter(counter_ifc);
```

as well as a rule to display the value of the configuration register inside the counter.

```
rule display_value (True);
   let read_value = fromMaybe(0, counter_ifc.cbus_ifc.read(13));
```

```
   $display ("Current Value %2d at time:", read_value, $time);
endrule
```

Recall that the configuration register was instantiated with an associated address of `13`. Next, we define a set of rules to first load a value of `4` into the counter and then to successively decrement the counter until it reaches a value of `1` at which time the simulation stops.

```
rule init_counter (counter_ifc.device_ifc.isZero());
   counter_ifc.device_ifc.load(4);
endrule

rule decrement (!counter_ifc.device_ifc.isZero());
   counter_ifc.device_ifc.decrement();
endrule

rule done (True);
   let read_value = fromMaybe(0, counter_ifc.cbus_ifc.read(13));
   if (read_value == 1) $finish();
endrule
```

If we simulate the `mkCBusExample` module, we see that via the configuration bus we are able to access the value of the `counter` register inside the counter module and that it has the expected values.

```
Current Value  0 at time:                 5
Current Value  4 at time:                15
Current Value  3 at time:                25
Current Value  2 at time:                35
$finish at simulation time                 45
```

The configuration bus can also be used to modify the values of control status registers. Let's change the design such that the `init_counter` rule now uses the configuration bus interface instead of the `Counter` interface `load` method.

```
rule init_counter (counter_ifc.device_ifc.isZero());
   counter_ifc.cbus_ifc.write(13,5);
endrule
```

The simulation results for this modified design show that the internal value of the `counter` register has indeed been modified via the configuration bus, and that the behavior is exactly as before except that for this design, the counter initializes to a value of `5` instead of `4`.

```
Current Value  0 at time:                 5
Current Value  5 at time:                15
Current Value  4 at time:                25
Current Value  3 at time:                35
```

```
Current Value  2 at time:                  45
$finish at simulation time                  55
```

6 CBus Method Conflicts

Given the ability to modify control status registers via a configuration bus, lets now investigate the behavior that results when design interface methods and configuration bus interface methods both attempt to modify the the value of a register at the same time. Consider a new testbench module which includes the same rules as before except that there are now two counter initialization rules.

```
rule init_counter_via_cbus (counter_ifc.device_ifc.isZero());
   counter_ifc.cbus_ifc.write(13,5);
endrule

rule init_counter_via_load (counter_ifc.device_ifc.isZero());
   counter_ifc.device_ifc.load(4);
endrule
```

Note that both rules *can fire* when the counter value is zero, and that both rules attempt to modify the value of the `counter` register (one via the `load` method of the `Counter` interface and one through the `write` method of the configuration bus). If we compile the code for this example, we get the following warning messages.

```
"CBusExample.bsv", line 79, column 8: (G0010) Warning:
  Rule "init_counter_via_load" was treated as more urgent than
  "init_counter_via_cbus". Conflicts:
    "init_counter_via_load" vs. "init_counter_via_cbus":
      calls to the_counter.device_ifc_load vs. the_counter.device_ifc_isZero
    "init_counter_via_cbus" vs. "init_counter_via_load":
      calls to the_counter.cbus_ifc_write vs. the_counter.device_ifc_isZero
"CBusExample.bsv", line 89, column 9: (G0021) Warning:
  According to the generated schedule, rule "init_counter_via_cbus" can never
  fire.
```

The first warning (`G0010`) tells us that there is a conflict between the two initialization rules (`init_counter_via_load` and `init_counter_via_cbus`) and that when they both *can fire* the compiler is giving priority to the `init_counter_via_load` rule. The second warning (`G0021`) tells us that the rule `init_counter_via_cbus` will never fire. This warning is a direct consequence of the first. Since the compiler fixes rule *urgencies* at compile time, and since these two rules have identical *can fire* conditions, the rule `init_counter_via_load` will always be selected over the rule `init_counter_via_cbus`. The rule `init_counter_via_cbus` will thus never fire. If we simulate this design, we get the following results.

```
Current Value  0 at time:                  5
Current Value  4 at time:                 15
Current Value  3 at time:                 25
Current Value  2 at time:                 35
$finish at simulation time                  45
```

We can see that the `init_counter_via_load` rule was indeed given priority, initializing the counter to a value of `4`.

7 Managing CBus Method Conflicts

In the example given in the previous section, the default behavior of the compiler is to give the rule `init_counter_via_load` priority over the rule `init_counter_via_cbus`. More precisely, the rule `init_counter_via_load` is given a higher urgency than `init_counter_via_cbus`. Thus when a conflict occurs such that only one of the rules can fire, `init_counter_via_load` is always the rule chosen to fire. What do we do if this is not the behavior we want?

One option is to direct the compiler to prioritize the rules differently by adding a *descending_urgency* attribute to the module definition. More specifically, if we add the following line to the definition of `mkCBusExample`

```
(* descending_urgency="init_counter_via_cbus, init_counter_via_load" *)
```

and then recompile, we'll see from the changed warnings that that the relative urgencies of the two rules have been changed and the `init_counter_via_cbus` rule will now fire instead of `init_counter_via_load`.

This example is somewhat artificial however in that there is only a single rule conflict to manage. A more realistic situation is one in which the configuration bus provides access to hundreds or thousands of registers and in which you'd like to have a consistent behavior in which the configuration bus `write` method always takes precedence over the `Reg` interface write (or visa versa).

Imagine for instance that we want the `CBus` interface `write` method and the `Reg` interface write to be conflict-free and that when both methods try to write simultaneously, the value from the `CBus` write takes precedence. We can modify the definition of the `regRW` to get this behavior. First we add two `RWire`s to the module, one for each write method.

```
RWire#(r) cbus_value();
mkRWire  _the_cbus_value(cbus_value);

RWire#(r) value();
mkRWire  _the_value(value);
```

Next we modify the write methods so as to update the added RWires instead of updating register x directly. The CBus interface write method now updates the cbus_value RWire

```
method Action write(addr, data);
   if (addr == reg_addr)
      begin
         cbus_value.wset(unpack(truncate(data)));
      end
endmethod
```

and the Reg interface write now updates the value RWire.

```
method Action _write(y);
   value.wset(y);
endmethod
```

Since these methods are no longer updating the same register, they are now conflict free. Finally we add a couple rules to update the value of the x register appropriately.

```
rule update_cbus_value (isValid(cbus_value.wget()));
   x <= validValue(cbus_value.wget());
endrule

rule update_value (isValid(value.wget()) && !isValid(cbus_value.wget()));
   x <= validValue(value.wget());
endrule
```

Note that the update_cbus_value rule will fire whenever a CBus interface write occurs (as indicated by cbus_value.wget() being valid). In contrast, the update_value rule will only fire when a Reg interface write occurs and a CBus interface write does not occur. The value written via the CBus interface will thus take precedence when simultaneous writes occur.

If we use this modified register in the example from Section 6, the rule conflicts disappear and the behavior is as desired, with the CBus interface write taking precedence. The simulation results are then as follows

```
Current Value  0 at time:                  5
Current Value  5 at time:                 15
Current Value  4 at time:                 25
Current Value  3 at time:                 35
Current Value  2 at time:                 45
$finish at simulation time                  55
```

with a counter initialization value of 5.