BSV 101: DESIGNING A COUNTER

November 24, 2008

1 Introduction

This tutorial aims to explore the basics of design in Bluespec SystemVerilog™ (BSV). We assume previous hardware design experince in Verilog or VHDL, and some familiarity with the Unix command line, and hope that, after you've completed this tutorial, you will be able to design, synthesize, and debug simple circuits and testbenches in BSV.

2 Counter

Let us start with a simple counter, perhaps in a file `Counter.bsv`.

First, we need to know how the counter will interact with the outside world. To start with, the user will be able to increment it, read the value, and set it to a custom value. In BSV parlance, this means we must define an interface:

```
interface Counter;
    method Bit#(8) read();
    method Action load(Bit#(8) newval);
    method Action increment();
endinterface
```

(we'll make an eight-bit counter first, and later generalize it to any bit width).

Our `Counter`[1] interface contains two kinds of methods: a *value* method (`read()`) and two *action* methods (`increment()` and `load()`). Only action methods, distinguished by a return type of `Action`, may modify state (such as register contents) in a module; the typechecker ensures that side effects are not permitted in value methods. Both kinds of methods may take arguments, as `load()` does above.

Now, we need a module, perhaps called `mkCounter()`, to implement (or provide) the `Counter` interface:

```
(* synthesize *)
module mkCounter(Counter);
    // module body goes here
endmodule
```

The `(* synthesize *)` attribute directs `bsc` to generate a separate Verilog module for `mkCounter()`—otherwise, `mkCounter()` would have been inlined in the instantiating module.[2]

Our counter will need to store its current value in a register, which we instantiate like any other module instance:

[1] in BSV, interface and type names (such as `Counter` or `Bit` above) begin with a capital letter, whereas module, variable, function, and method names (such as `read()` or `newval`) start with a lowercase letter.

[2] in general, we prefer to let `bsc` inline instantiated modules; we employ `(* synthesize *)` when the blocks must be separately handled by downstream tools (e.g., RTL synthesis).

```
Reg#(Bit#(8)) value <- mkReg(0);
```

(the 0 argument to mkReg is the reset value of the register).[3]

State by itself, of course, is rather useless: we must allow the user to change and read it via the methods of our `Counter` interface. The `read()` method is the easiest: it just relays the contents of `value`:

```
method Bit#(8) read();
    return value;
endmethod
```

Now, `load()` overwrites `value` with its argument,

```
method Action load(Bit#(8) newval);
    value <= newval;
endmethod
```

and `increment()` adds one to `value` and writes it back:

```
method Action increment();
    value <= value + 1;
endmethod
```

Putting the register instance and the methods together inside the `module ... endmodule` block, we obtain the complete counter module. We can now synthesize our BSV into Verilog by running

```
bsc -u -verilog Counter.bsv
```

Barring errors caused by typos, the output should resemble

```
checking package dependencies
compiling Counter.bsv
code generation for mkCounter starts
Verilog file created: mkCounter.v
packages up-to-date
```

Note that, because of the `-u` flag, `bsc` only recompiles files which have changed, so re-running the `bsc` command above will not cause `mkCounter.v` to be regenerated.[4]

[3] `mkReg()` creates a register with synchronous reset. An asynchronously reset register can be made with `mkRegA()`, and a register without reset with `mkRegU()`.

[4] we can force recompilation by removing the relevant `.bi` and `.bo` files (intermediate files created by `bsc`) or `touch`ing the source file.

3 Testbench

Now, how do we test this counter? Out of the many possible approaches, we shall focus on a simple finite state machine which performs predefined actions at certain times and checks that the counter output matches expectations.

First, we'll write the testbench module skeleton in a file `TbCounter.bsv`:

```
import Counter::*;

(* synthesize *)
module mkTbCounter();

endmodule
```

You may be surprised that, unlike `mkCounter()`, `mkTbCounter()` leaves out the name of the interface it implements. Because our testbench doesn't need to interact with any external modules, its interface contains no methods; `bsc` assumes such an interface when a module declaration omits the interface type.

Our testbench will instantiate a counter to test and a state register (containing, say, sixteen bits) to keep track of which phase it's in:

```
Counter counter <- mkCounter();
Reg#(Bit#(16)) state <- mkReg(0);
```

Now all we need to do is to add behavior triggered by certain values of `state`. In `mkCounter()`, all behavior was contained in methods and carried out when the methods were invoked by the instantiating module, but `mkTbCounter()` doesn't have any methods! For this purpose, we employ rules—self-triggered methods of sorts:

```
rule step0(state == 0);
    counter.load(42);
    state <= 1;
endrule

rule step1(state == 1);
    if (counter.read() != 42) $display("FAIL: counter.load(42)");
    state <= 2;
endrule
```

We'll also add a rule to end the simulation at a specific time:

```
rule done(state == 2);
    $display("TESTS FINISHED");
    $finish(0);
endrule
```

which completes our testbench. We can now synthesize everything to RTL by running

```
bsc -u -verilog TbCounter.bsv
```

which should produce `mkCounter.v` and `mkTbCounter.v`.

4 Simulation

We can employ Bluespec's native simulator Bluesim to simulate the design and generate a VCD. To do this, we compile the sources to Bluesim objects instead of Verilog RTL,

```
bsc -u -sim TbCounter.bsv
```

and create and run the simulator executable:[5]

```
bsc -o sim -e mkTbCounter mkTbCounter.o
./sim -V dump.vcd
```

Alternately, we can simulate the generated Verilog RTL using a Verilog simulator. Because clock and reset signals are by default implicit in BSV, we must use a wrapper which instantiates the module to be simulated and flips the clock every so often. Conveniently, the file `main.v` included in the BSV distribution does just that: it instantiates a module identified by preprocessor macro `TOP` and toggles the clock every five simulation cycles. `main.v` also dumps all signals to a VCD file whenever it is passed the `+bscvcd` flag. All we have to do is define `TOP` to be `mkTbCounter` and compile `main.v` along with the generated RTL.

For example, using VCS,

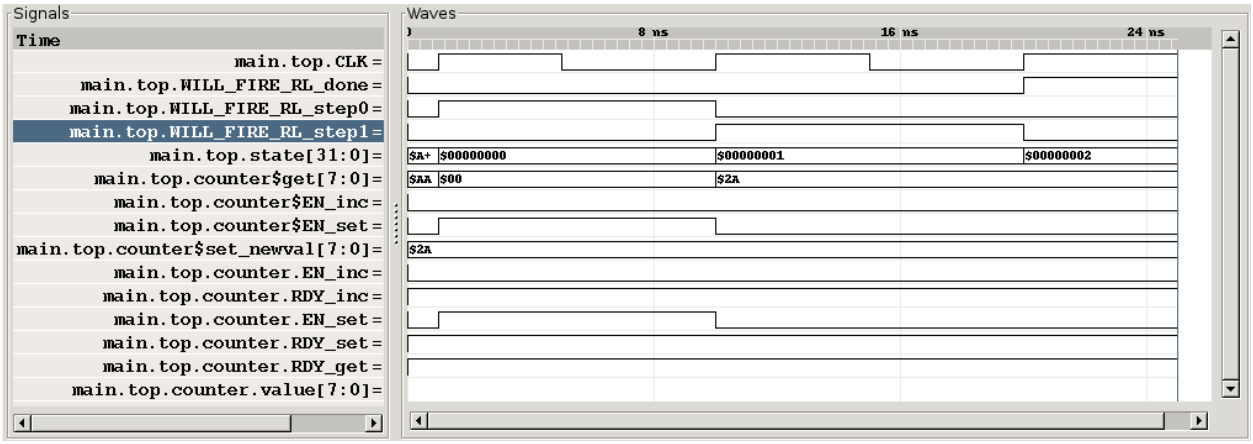
```
bsc -verilog -e mkTbCounter -o simv -vsim vcs
mkTbCounter.v mkCounter.v
```

or, using NCVerilog,

```
bsc -verilog -e mkTbCounter -o simv -vsim ncverilog
mkTbCounter.v mkCounter.v
```

Either way, we should see "`TESTS FINISHED`" as the output, and obtain a waveform dump in `dump.vcd`. Let us examine the VCD:
You'll notice that the VCD has several signals which do not precisely correspond to registers or wires in the BSV source. The signals named `WILL_FIRE_RL_`*`rulename`* are asserted when the relevant rule is being executed, and make it easy to tell which operations our testbench is performing during each clock cycle. Special signals are also generated for each method: in addition to the outputs of *value* methods (e.g., `read[7:0]`) and method arguments (e.g., `load_newval[7:0]`), we see a "ready" signal for each method (e.g., `RDY_increment`), and an

[5] Bluesim simulators have a number of other useful features; run the simulator with the `-h` flag for more information



"enable" signal for each *action* method (e.g., `EN_increment`). Just as the rules' `WILL_FIRE` signals, these tell us when a given method is ready to be called[6] and (for action methods) actually invoked, and are invaluable in debugging.

5 Decrementing

Let us nows extend our counter by adding a `decrement()` method. We'll have to extend our interface by adding a declaration for `decrement()` to the `interface ... endinterface` block:

```
method Action decrement();
```

and implement the method in the module block:

```
method Action decrement();
    value <= value - 1;
endmethod
```

That's all! Note that we don't need to modify the logic in any of the other methods—`bsc` will analyze any concurrency conflicts we may have introduced and automatically insert interlocks, so each rule and method is guaranteed to do exactly what it says regardless of what other rules and methods exist. What's more, the existing tests in the testbench do not need to be modified to account for the new addition.

While we're on the subject of concurrency, we might well ask what happens when `increment()` and `decrement()` are invoked at the same time (we'd *like* them to cancel each other out). Let us experiment: let's change rule `step1` to call both methods:

[6] in our `mkCounter()`, all methods are always ready. While this need not always be the case—a method may wait for a computation to complete, for example—in cases where ready signals are always high, the ports can be removed by adding "`always_ready`" to the module's `(* synthesize *)` attribute (`bsc` will report an error unless it can prove that the method is in fact always ready). There is a dual "`always_enabled`" attribute which removes the enable signals and ensures that the method is always called.

```
rule step1(state == 1);
    counter.increment();
    counter.decrement();
    state <= 2;
endrule
```

and recompile. But now `bsc` reports an error:

```
"TbCounter.bsv", line 13, column 10: (G0004) Error:
  Rule RL_step1 uses write methods that conflict in parallel:
    counter.increment
  and
    counter.decrement
```

It turns out that `bsc` does not know much about algebra, and does not permit `increment()` and `decrement()` to be called concurrently—but putting both calls in the same rule requests just that.

If we can't put the calls in one rule, can we put them in two different rules? Let's try:

```
rule step1a(state == 1);
    counter.increment();
endrule
rule step1b(state == 1);
    counter.decrement();
endrule
rule step1c(state == 1);
    state <= 2;
endrule
```

Now, compilation succeeds, but with two warnings:

```
"TbCounter1.bsv", line 4, column 8: (G0010) Warning:
  Rule "step1b" was treated as more urgent than "step1a". Conflicts:
    "step1b" vs. "step1a":
      calls to
 counter.decrement vs. counter.increment
    "step1a" vs. "step1b":
      calls to
 counter.increment vs. counter.decrement
"TbCounter1.bsv", line 13, column 14: (G0021) Warning:
  According to the generated schedule, rule "step1a" can never fire.
```

Now, because `increment()` and `decrement()` could not be called in the same clock cycle, the scheduler chose to fire rule `step1b` whenever `state == 1`, thus ruling out `step1a`.

But let us return to the goal of making `increment()` and `decrement()` safe to call concurrently. Implementing our desired behavior will require some more BSV machinery.

Conceptually, we would like to specify special behavior when both methods are called. But, because methods cannot communicate among themselves except through instantiated modules, this is not possible directly.

Instead, we'll use a predefined element called `PulseWire`. A `PulseWire` can be set via its `send()` method and read by mentioning its name; because it contains no clocked state, the pulse vanishes at the next clock edge. We'll need a `PulseWire` for each method, which we'll set whenever that method is called; we will then carry out the arithmetic in rules which read the `PulseWire`s. In the `module ... endmodule` block, we instantiate the pulse wires:

```
PulseWire increment_called <- mkPulseWire();
PulseWire decrement_called <- mkPulseWire();
```

and change each method to do nothing but send a pulse down the relevant wire:

```
method Action increment();
    increment_called.send();
endmethod

method Action decrement();
    decrement_called.send();
endmethod
```

All that remains is to implement rules which increment and decrement `value` whenever one of the `PulseWire`s is set:

```
rule do_increment(increment_called && !decrement_called);
    value <= value + 1;
endrule

rule do_decrement(!increment_called && decrement_called);
    value <= value - 1;
endrule
```

Since the value does not change when `increment()` and `decrement()` are invoked simultaneously, no rule is required to cover that case.

6 Better Testbench

Adding tests to our testbench is a bit of a pain: each test requires two new rules, triggered on a unique value of the "current state," which wiggle the device under test, update the state, check that the device has responded correctly, and update the state again, as well as a change to the `done` rule to match the new "last" state. Fortunately, BSV has a library for describing finite state machines more concisely, which allows us to write a shorter and more maintainable testbench.

Briefly, the FSM library works by capturing a sequence of actions and instantiating a module which automatically builds the FSM to run this sequence one step per clock cycle. The sequence of `action`s is defined using a `seq` ... `endseq` block and has type `Stmt`. An FSM is then created by passing the sequence to `mkAutoFSM()`.

To add this to our testbench, we need to import the FSM library in addition to our device under test:

```
import StmtFSM::*;
import Counter::*;
```

We then define a statement sequence `test_seq` to be executed by the FSM—perhaps set the counter to a value and check that it returns that value in the next cycle:

```
Stmt test_seq = seq
         counter.load(42);
         if (counter.read() != 42) $display("FAIL: counter != 42");
         $display("TESTS FINISHED");
     endseq;
```

and instantiate an FSM to sequentially execute the statements in the `seq` ... `endseq` block, one per clock cycle:[7]

```
mkAutoFSM(test_seq);
```

Under the covers, the `mkAutoFSM()` instantiation creates a new module with an automatically sized state counter, adds a rule for every `action` ... `endaction` block, much like we did by hand in the original testbench; the resulting FSM runs through the sequence once and exits the simulation.

Our new testbench now does exactly what our previous one did, but it's much easier to create new tests: we only need to add lines to the `test_seq` sequence.

There is another way we can simplify adding new tests. Most of our tests will have a stage checking that the counter is set to a specific value—which we achieved above with

```
if (counter.read() != 42) $display("FAIL: counter != 42");
```

Since all of this (other than the value 42) will remain the same in each test, we can save ourselves some typing by defining, in the module body, a function which takes the expected value as an argument and reports a failure when the counter does not match it:

```
function check(expected_val);
    action
        if (counter.read() != expected_val)
            $display("FAIL: counter != %0d", expected_val);
    endaction
endfunction
```

[7] to execute multiple statements in the same cycle, place them in an `action` ... `endaction` block.

We can then rewrite `test_seq` in terms of `check`:

```
Stmt test_seq = seq
                    counter.load(42);
                    check(42);
                    $display("TESTS FINISHED");
                endseq;
```

and revel in the ease of adding tests.

7 Generalized Counter

There is no good reason why our counter couldn't work with any bit width: save for the number of wires, the arithmetic is the same. To achieve this, we should only need to replace all occurrences of 8 in the program with a type variable, and everything should work.

First, the interface: we'll make all of the methods parameterized on the bit-width, a type variable we'll call `size_t`:

```
interface Counter#(type size_t);
    method Bit#(size_t) read();
    method Action increment();
    method Action decrement();
    method Action load(Bit#(size_t) newval);
endinterface
```

Note that it's really the *interface*, not the *methods*, that is parameterized, so `size_t` remains the same in all of the methods.

We'll also need to change the type of the interface implemented by `mkCounter()`. Changing the module header to

```
module mkCounter(Counter#(8));
```

would work, but would again create an eight-bit counter; instead, we choose to parameterize the module as well:

```
module mkCounter(Counter#(size_t));
```

(from the lowercase letter starting `size_t`, `bsc` knows that it's a type variable, not a concrete type). `size_t` will be replaced with a concrete size when `mkCounter()` is instantiated.

Inside the module, the type of the register contents changes,

```
Reg#(Bit#(size_t)) value <- mkReg(0);
```

and so does the return type of `read()`,

```
method Bit#(size_t) read();
```

and the argument type of `load()`,

```
method Action load(Bit#(size_t) newval);
```

Finally, in our testbench, we must add the size parameter to the `mkCounter()` instantiation:

```
Counter#(8) counter <- mkCounter();
```

And that's all: `bsc` will fix `size_t` to 8 and automatically adjust related operations and assignments to the correct bit width.

Compiling our new counter, however, with

```
bsc -u -verilog TbCounter.bsv
```

results in an error:

```
Error: "Counter.bsv", line 10, column 8: (T0043)
  Cannot synthesize ‘mkCounter’: Its interface is polymorphic.
```

Why? It turns out that `bsc` does not produce parameterized RTL or simulators; because BSV parameterization mechanisms are more powerful than those of the RTL, this is often not even possible. So, instead of generating a separate Verilog module for `mkCounter.v`, we'll remove the `(* synthesize *)` attribute on `mkCounter()` and let `bsc` inline it in the design which instantiates it. Now the compilation works and the circuit simulates as previously.

8 Advanced Generalization

(Caveat: this is an advanced topic!)

But, really, the counter could work for any type... well, *almost*: any type on which addition is defined, and which can be converted to bits (to be stored in a register).[8] This kind of constraint is expressed in BSV using a *proviso*: intuitively, a module can have a certain interface type *provided* that some requirement is satisfied.

Since the interface says nothing about how the implementation behind it should work, no provisos are necessary, and we just replace `Bit#(size_t)` with a type variable, say, `count_t`:

```
interface Counter#(type count_t);
    method count_t read();
    method Action increment();
    method Action decrement();
    method Action load(count_t newval);
endinterface
```

[8] values of type `Bool`, for example, cannot be added, and interfaces—such as `Counter`—cannot be converted to a fixed number of bits.

The module, however, requires a proviso, because our implementation requires addition and conversion to bits. The former is expressed as membership in the `Arith` typeclass, and the latter, as membership in the `Bits` typeclass:

```
module mkCounter(Counter#(count_t))
        provisos(Arith#(count_t), Bits#(count_t, count_t_sz));
```

(the `count_t_sz` is computed from `count_t` by `bsc` and available inside the module). We continue the replacement with `value`,

```
Reg#(count_t) value <- mkReg(0);
```

the `read()` method header,

```
method count_t read();
```

the `load()` method header,

```
method Action load(count_t newval);
```

and, finally, the counter instance in the testbench:

```
Counter#(Bit#(8)) counter <- mkCounter();
```

which propagates `Bit#(8)` to the relevant uses of `count_t`.

Now, after compilation, we have the same eight-bit counter as before. So what have we gained? We can now vary more than just the bit-width: we can easily change our design to count, say, twenty-bit signed integers,

```
Counter#(Int#(20)) counter <- mkCounter();
```

or any user-defined type for which addition is defined (for example, a complex number type or a floating-point number type).[9]

9 Summary

We started by developing a simple eight-bit counter with a simple testbench. Next, we learned how to compile and simulate BSV designs, and how to examine the waveforms. We then extended the behavior of the counter by adding a `decrement()` method—while trivially keeping all previous behavior and without having to alter caller modules (such as our testbench)—or other rules. Later, we made an improved—more extendible and maintainable—testbench using the FSM library and abstracting behavior in a function, and parameterized the counter design to any bit length and then to any possible type. While these examples have, by necessity, employed only some of the power of BSV, we hope they have succeeded as a brief introduction.

[9] see the *Reference Guide* for information on defining addition on custom types.

10 Exercises

Exercise 1 Add tests for `increment()` and `decrement()` to both versions of the testbench.

Exercise 2 In the `increment()`-only version of the counter, change `increment()` to take as an argument the amount by which to increment the counter.

Exercise 3 Write a "cycle" counter which only has `read()` and `load()` methods, and increments automatically on every cycle during which `load()` is not called.

Exercise 4 In the counter with simultaneous `increment()` and `decrement()`, change `increment()` and `decrement()` to take the change amount as an argument, as above. You might find either `mkWire()` or `mkRWire()` useful (both are defined in Prelude and always available).

Appendix A Chapter 2 code

```
// simple counter from Chapter 2

interface Counter;
    method Bit#(8) read();
    method Action load(Bit#(8) newval);
    method Action increment();
endinterface

(* synthesize *)
module mkCounter(Counter);
    Reg#(Bit#(8)) value <- mkReg(0);

    method Bit#(8) read();
        return value;
    endmethod

    method Action load(Bit#(8) newval);
        value <= newval;
    endmethod

    method Action increment();
        value <= value + 1;
    endmethod
endmodule
```

Appendix B Chapter 3 code

```
import Ch02_Counter::*;

(* synthesize *)
module mkTbCounter();
    Counter counter <- mkCounter();
    Reg#(Bit#(16)) state <- mkReg(0);

    rule step0(state == 0);
        counter.load(42);
        state <= 1;
    endrule

    rule step1(state == 1);
```

```
        if (counter.read() != 42) $display("FAIL: counter.load(42)");
        state <= 2;
    endrule

    rule done(state == 2);
        $display("TESTS FINISHED");
        $finish(0);
    endrule
endmodule
```

Appendix C Chapter 5 code

```
// counter + decrement from Chapter 5

interface Counter;
    method Bit#(8) read();
    method Action load(Bit#(8) newval);
    method Action increment();
    method Action decrement();
endinterface

(* synthesize *)
module mkCounter(Counter);
    Reg#(Bit#(8)) value <- mkReg(0);

    PulseWire increment_called <- mkPulseWire();
    PulseWire decrement_called <- mkPulseWire();

    rule do_increment(increment_called && !decrement_called);
        value <= value + 1;
    endrule

    rule do_decrement(!increment_called && decrement_called);
        value <= value - 1;
    endrule

    method Bit#(8) read();
        return value;
    endmethod

    method Action load(Bit#(8) newval);
        value <= newval;
```

```
    endmethod

    method Action increment();
        increment_called.send();
    endmethod

    method Action decrement();
        decrement_called.send();
    endmethod
endmodule
```

Appendix D Chapter 6 code

```
import StmtFSM::*;
import Ch05_Counter::*;

(* synthesize *)
module mkTbCounter();
    Counter counter <- mkCounter();
    Reg#(Bit#(16)) state <- mkReg(0);

    // check that the counter matches an expected value
    function check(expected_val);
        action
            if (counter.read() != expected_val)
            $display("FAIL: counter != %0d", expected_val);
        endaction
    endfunction

    Stmt test_seq = seq
            counter.load(42);
            check(42);
            $display("TESTS FINISHED");
        endseq;
    mkAutoFSM(test_seq);
endmodule
```

Appendix E Chapter 7 code

```
// generalized counter from Chapter 7
```

```
interface Counter#(type size_t);
    method Bit#(size_t) read();
    method Action load(Bit#(size_t) newval);
    method Action increment();
    method Action decrement();
endinterface

module mkCounter(Counter#(size_t));
    Reg#(Bit#(size_t)) value <- mkReg(0);

    PulseWire increment_called <- mkPulseWire();
    PulseWire decrement_called <- mkPulseWire();

    rule do_increment(increment_called && !decrement_called);
        value <= value + 1;
    endrule

    rule do_decrement(!increment_called && decrement_called);
        value <= value - 1;
    endrule

    method Bit#(size_t) read();
        return value;
    endmethod

    method Action load(Bit#(size_t) newval);
        value <= newval;
    endmethod

    method Action increment();
        increment_called.send();
    endmethod

    method Action decrement();
        decrement_called.send();
    endmethod
endmodule




import StmtFSM::*;
import Ch07_Counter::*;
```

```
(* synthesize *)
module mkTbCounter();
    Counter#(8) counter <- mkCounter();
    Reg#(Bit#(16)) state <- mkReg(0);

    // check that the counter matches an expected value
    function check(expected_val);
        action
            if (counter.read() != expected_val)
            $display("FAIL: counter != %0d", expected_val);
        endaction
    endfunction

    Stmt test_seq = seq
            counter.load(42);
            check(42);
            $display("TESTS FINISHED");
        endseq;
    mkAutoFSM(test_seq);
endmodule
```

Appendix F Chapter 8 code

```
// fully generalized counter from Chapter 8

interface Counter#(type count_t);
    method count_t read();
    method Action load(count_t newval);
    method Action increment();
    method Action decrement();
endinterface

module mkCounter(Counter#(count_t))
        provisos(Arith#(count_t), Bits#(count_t, count_t_sz));
    Reg#(count_t) value <- mkReg(0);

    PulseWire increment_called <- mkPulseWire();
    PulseWire decrement_called <- mkPulseWire();

    rule do_increment(increment_called && !decrement_called);
        value <= value + 1;
```

```
    endrule

    rule do_decrement(!increment_called && decrement_called);
        value <= value - 1;
    endrule

    method count_t read();
        return value;
    endmethod

    method Action load(count_t newval);
        value <= newval;
    endmethod

    method Action increment();
        increment_called.send();
    endmethod

    method Action decrement();
        decrement_called.send();
    endmethod
endmodule




import StmtFSM::*;
import Ch08_Counter::*;

(* synthesize *)
module mkTbCounter();
    Counter#(Bit#(8)) counter <- mkCounter();
    Reg#(Bit#(16)) state <- mkReg(0);

    // check that the counter matches an expected value
    function check(expected_val);
        action
            if (counter.read() != expected_val)
            $display("FAIL: counter != %0d", expected_val);
        endaction
    endfunction

    Stmt test_seq = seq
```

```
            counter.load(42);
            check(42);
            $display("TESTS FINISHED");
        endseq;
    mkAutoFSM(test_seq);
endmodule
```