Übung zur Vorlesung Einführung in Computer Microsystems

Prof. Dr-Ing. A. Koch
Jaco Hofmann, MSc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 15

Übungsblatt 2 - Lösungsvorschlag

Aufgabe 2.1 FSM in Bluespec

Finite State Machine (FSM) werden in Hardware häufig gebraucht im sequentielle Abläufe zu Modellieren. Dementsprechend werden FSM auch häufig in Bluespec gebraucht. In Übung 1 wurde eine FSM zur Eingabe von Stimuli in das zu testende Modul benutzt.

Um die Nutzung von FSM in Bluespec zu vereinfachen existiert in der AzureIP Bibliothek das Packet `StmtFSM` (Einbinden des Moduls mit `import` nicht vergessen). Die darin definierte Sprache `Stmt` ermöglicht das einfache erstellen von FSM. `Stmt` ist dabei folgendermaßen Definiert:

```
1     exprPrimary ::= seqFsmStmt | parFsmStmt
2     fsmStmt     ::= exprFsmStmt
3                   | seqFsmStmt
4                   | parFsmStmt
5                   | ifFsmStmt
6                   | whileFsmStmt
7                   | repeatFsmStmt
8                   | forFsmStmt
9                   | returnFsmStmt
10    exprFsmStmt ::= regWrite ;
11                  | expression ;
12    seqFsmStmt  ::= seq fsmStmt { fsmStmt } endseq
13    parFsmStmt  ::= par fsmStmt { fsmStmt } endpar
14    ifFsmStmt   ::= if expression fsmStmt
15                    [ else fsmStmt ]
16    whileFsmStmt    ::= while ( expression )
17                        loopBodyFsmStmt
18    forFsmStmt      ::= for ( fsmStmt ; expression ; fsmStmt )
19                        loopBodyFsmStmt
20    returnFsmStmt   ::= return ;
21    repeatFsmStmt   ::= repeat ( expression )
22                        loopBodyFsmStmt
23    loopBodyFsmStmt ::= fsmStmt
24                      | break ;
25                      | continue ;
```

In Bluespec lässt sich dementsprechend ein Objekt vom Typ `Stmt` folgendermaßen erzeugen:

```
1  Stmt myFirstFSM = {
2    seq
3      action
4        $display("Hello World.");
5      endaction
6    endseq
7  };
```

Zur Nutzung in `Stmt` sind zusätzlich einige Funktionen definiert.

```
function Action await(Bool cond);
function Stmt delay(a_type value);
```

Die Funktion `await` wartet dabei mit der Fortsetzung der Ausführung der FSM bis die Bedingung (die als Parameter Übergeben wurde) wahr ist. Die Funktion `delay` verzögert die Ausführung der FSM um die als Parameter angegebenen Takte.

Dieses `Stmt` Objekt kann als Parameter zur Erzeugung einer `FSM` Instanz genutzt werden. Für das Interface `FSM` sind dabei drei verschiedene Module definiert:

```
module mkFSM#( Stmt seq_stmt ) ( FSM );
module mkFSMWithPred#( Stmt seq_stmt, Bool pred ) ( FSM );
module mkAutoFSM#( seq_stmt ) ();
```

Aufgabe 2.1.1 Eine erste FSM

Nutzen Sie `mkAutoFSM` und `delay` dazu eine FSM zu erstellen die 100 Taktzyklen wartet und danach „Hello World“ sowie die aktuelle Systemzeit (`$time`) ausgibt.

```
package FSMTests;

  import StmtFSM :: *;
  module mkFirstFSM(Empty);
    Stmt firstStmt = {
      seq
          delay(100);
          action
              $display("(%0d) Hello World!", $time);
          endaction
      endseq
    };
    mkAutoFSM(firstStmt);
endmodule
endpackage
```

Was stellen Sie fest wenn Sie die Zeit der Ausgabe der Nachricht betrachten?

Aufgabe 2.1.2 Parallel Ausführung in FSM

Neben der sequentiellen Ausführung von Aktionen mit `seq` können diese auch Parallel ausgeführt werden mit `par`.

Erstellen Sie eine FSM mit zwei parallel ausgeführten sequentiellen Teilen. Der erste Teil soll dabei eine Nachricht ausgeben (Denken Sie daran die Systemzeit mit auszugeben) und nach 100 Taktzyklen ein Bool-Register auf `True` setzen.

Der zweite Parallele Teil soll mit Hilfe von `repeat` 10 mal eine Nachricht ausgeben und danach auf den ersten Teil warten.

Am Ende sollen beide sequentiellen Teile gleichzeitig eine Nachricht ausgeben.

Was fällt Ihnen beim Betrachten der beiden Schlussnachrichten auf?

`await` scheint einen zusätzlichen Taktzyklus zu brauchen um nach setzen der Synchronisationsvariable in den nächsten State zu springen.

Aufgabe 2.1.3 FSM Ausführung steuern

Häufig möchte man nicht, dass die eingesetzten FSM mit dem Systemtakt angesteuert wird. Eine Möglichkeit die FSM mit einem beliebigen (aber langsameren als dem Systemtakt) Takt anzusteuern ist die Verwendung von `mkFSMWithPred`.

Erstellen Sie eine FSM die mit $\frac{1}{100}$ des Systemtakts vorwärts läuft. Verwenden Sie dafür einen Zähler und ein `PulseWire` mit folgender Definition:

```
interface PulseWire;
  method Action send();
  method Bool _read();
endinterface
```

Die FSM soll dabei 20 mal eine Nachricht ausgeben und in der Nachricht die Zählvariable beinhalten. Nutzen Sie dafür eine `for` Schleife.

```
module mkThirdFSM(Empty);
  Reg#(UInt#(12)) counter <- mkReg(0);
  PulseWire pw <- mkPulseWire();
  Reg#(UInt#(12)) i <- mkReg(0);

  rule count (counter < 99);
    counter <= counter + 1;
  endrule

  rule resetCount (counter == 99);
    counter <= 0;
    pw.send();
  endrule

  Stmt thirdStmt = {
    seq
      for(i <= 0; i < 20; i <= i + 1) seq
        $display("(%0d) Iteration %d.",$time,i);
      endseq
      $finish();
    endseq
  };
  FSM myFSM <- mkFSMWithPred(thirdStmt, pw);
  rule startFSM (myFSM.done());
    myFSM.start();
  endrule
endmodule
```

Was fällt Ihnen auf wenn Sie die Zeitpunkte der Ausgaben betrachten? Was können Sie daraus im Bezug auf zeitkritische Anwendungen schließen?

Die Schleife benötigt einen extra Takt zur Verwaltung der Schleifenvariable sowie zur Überprüfung der Abbruchbedingung. Wenn die eigentliche Aktion nur einen Takt benötigt ergibt sich daraus ein Overhead von 100 %.

Aufgabe 2.1.4 FSM als Testbench

Das Modul `mkAutoFSM` eignet sich hervorragend zur Erstellung von Testbenches.

Schreiben Sie eine Testbench für das Modul `mkHelloALU` aus der ersten Übung. Nutzen Sie dabei einen Vektor der alle Testdaten beinhaltet. Lagern Sie häufig genutzte Teile (Operanden eingeben und Ergebnis überprüfen) in eine extra FSM aus indem Sie `mkAutoFSM` und `mkFSM` kombinieren. Einen Vektor können Sie folgendermaßen erzeugen:

```
typedef struct {
  Int#(32) opA;
  Int#(32) opB;
  AluOps    operator;
  Int#(32) expectedResult;
} TestData deriving (Eq, Bits);
...
Vector#(20, TestData) myVector;
myVector[0] = TestData {opA: 2, opB: 4, operator: Add, expectedResult: 6};
```

```
...
myVector[19] = TestData {opA: 4, opB: 2, operator: Div, expectedResult: 2};
```

Aufgabe 2.2 Tagged Unions

Tagged Unions sind ein zusammengesetzter Typ der im Gegensatz zur `struct` immer genau einen seiner Member enthält. Eine `tagged union` wird dabei wie eine `struct` erstellt.

```
typedef union tagged {UInt#(32) Unsigned; Int#(32) Signed;} SignedOrUnsigned deriving(Bits, Eq);
```

Das jeweilige Wert kann dabei mit `pattern matching` extrahiert werden. In einer Guard würde das folgendermaßen aussehen:

```
rule someRule (unionReg matches tagged Signed .v);
  $display("%d", v);
endrule;
rule anotherRule (unionReg matches tagged Unsigned .v);
  $display("%u", v);
endrule
```

Weitere Möglichkeiten für `pattern matching` finden Sie ab Seite 82 in der Bluespec Referenz.

Aufgabe 2.2.1 Flexible ALU

Erweitern Sie die ALU aus der vorherigen Übung um die Möglichkeit `UInt` Werte zu verarbeiten. Fügen Sie dabei keine weitere `Action` hinzu sondern verwenden Sie die oben definierte `tagged union SignedOrUnsigned`.

```
package Alu;
    typedef enum {Mul,Div,Add,Sub,And,Or,Pow} AluOps deriving (Eq, Bits);
    typedef union tagged {UInt#(32) Unsigned; Int#(32) Signed;} SignedOrUnsigned deriving(Bits, Eq);

    interface Power#(type t);
        method Action   setOperands(t a, t b);
        method t getResult();
    endinterface

    module mkPower(Power#(t))
        provisos(Bits#(t, t_sz),
                 Ord#(t),
                 Arith#(t),
                 Eq#(t));
        Reg#(Bool) resultValid <- mkReg(False);

        Reg#(t) opA    <- mkReg(0);
        Reg#(t) opB    <- mkReg(0);
        Reg#(t) result <- mkReg(1);

        rule calc (opB > 0);
            opB <= opB - 1;
            result <= result * opA;
        endrule

        rule calcDone (opB == 0 && !resultValid);
            resultValid <= True;
        endrule

```

```
30          method Action setOperands(t a, t b);
31              result <= 1;
32              opA    <= a;
33              opB    <= b;
34              resultValid <= False;
35          endmethod
36
37          method t getResult() if(resultValid);
38              return result;
39          endmethod
40      endmodule
41
42      interface HelloALU;
43          method Action setupCalculation(AluOps op, SignedOrUnsigned a, SignedOrUnsigned b);
44          method ActionValue#(SignedOrUnsigned)  getResult();
45      endinterface
46
47      module mkHelloALU(HelloALU);
48          Reg#(Bool) newOperands <- mkReg(False);
49          Reg#(Bool) resultValid <- mkReg(False);
50          Reg#(AluOps) operation <- mkReg(Mul);
51          Reg#(SignedOrUnsigned) opA    <- mkReg(tagged Signed 0);
52          Reg#(SignedOrUnsigned) opB    <- mkReg(tagged Signed 0);
53          Reg#(SignedOrUnsigned) result <- mkReg(tagged Signed 0);
54
55          Power#(UInt#(32)) powUInt <- mkPower();
56          Power#(Int#(32))  powInt  <- mkPower();
57
58          rule calculateSigned (opA matches tagged Signed .va &&& opB matches tagged Signed .vb &&& newOperand
59                  Int#(32) rTmp = 0;
60                  case(operation)
61                      Mul: rTmp = va * vb;
62                      Div: rTmp = va / vb;
63                      Add: rTmp = va + vb;
64                      Sub: rTmp = va - vb;
65                      And: rTmp = va & vb;
66                      Or:  rTmp = va | vb;
67                      Pow: rTmp = powInt.getResult();
68                  endcase
69                  result <= tagged Signed rTmp;
70                  newOperands <= False;
71                  resultValid <= True;
72          endrule
73
74          rule calculateUnsigned (opA matches tagged Unsigned .va &&& opB matches tagged Unsigned .vb &&& newO
75                  UInt#(32) rTmp = 0;
76                  case(operation)
77                      Mul: rTmp = va * vb;
78                      Div: rTmp = va / vb;
79                      Add: rTmp = va + vb;
80                      Sub: rTmp = va - vb;
81                      And: rTmp = va & vb;
82                      Or:  rTmp = va | vb;
83                      Pow: rTmp = powUInt.getResult();
84                  endcase
```

```
85                 result <= tagged Unsigned rTmp;
86                 newOperands <= False;
87                 resultValid <= True;
88         endrule
89
90         function Bool isUnsigned(SignedOrUnsigned v);
91             if(v matches tagged Unsigned .va) return True;
92             else return False;
93         endfunction
94
95         rule dumpInvalid (newOperands && isUnsigned(opA) != isUnsigned(opB));
96             $display("Invalid combination of Signed and Unsigned Operands");
97             newOperands <= False;
98             resultValid <= False;
99         endrule
100
101        method Action setupCalculation(AluOps op, SignedOrUnsigned a, SignedOrUnsigned b) if(!newOperands);
102            opA <= a;
103            opB <= b;
104            operation <= op;
105            newOperands <= True;
106            resultValid <= False;
107            if(op == Pow) begin
108                if(a matches tagged Signed .va &&& b matches tagged Signed .vb) powInt.setOperands(va,vb);
109                else if(a matches tagged Unsigned .va &&& b matches tagged Unsigned .vb) powUInt.setOperands(va
110                else $display("Mixed signs not supported.");
111            end
112        endmethod
113
114        method ActionValue#(SignedOrUnsigned) getResult() if(resultValid);
115            resultValid <= False;
116            return result;
117        endmethod
118    endmodule
119
120    module mkALUTestbench(Empty);
121        HelloALU uut             <- mkHelloALU();
122        Reg#(UInt#(8)) testState <- mkReg(0);
123
124        rule checkMul (testState == 0);
125            uut.setupCalculation(Mul, tagged Unsigned 4, tagged Unsigned 5);
126            testState <= testState + 1;
127        endrule
128
129        rule checkDiv (testState == 2);
130            uut.setupCalculation(Div, tagged Unsigned 12,tagged Unsigned 4);
131            testState <= testState + 1;
132        endrule
133
134        rule checkAdd (testState == 4);
135            uut.setupCalculation(Add, tagged Unsigned 12, tagged Unsigned 4);
136            testState <= testState + 1;
137        endrule
138
139        rule checkSub (testState == 6);
```

```
            uut.setupCalculation(Sub, tagged Unsigned 12, tagged Unsigned 4);
            testState <= testState + 1;
        endrule

        rule checkAnd (testState == 8);
            uut.setupCalculation(And, tagged Unsigned 32'hA, tagged Unsigned 32'hA);
            testState <= testState + 1;
        endrule

        rule checkOr (testState == 10);
            uut.setupCalculation(Or, tagged Unsigned 32'hA, tagged Unsigned 32'hA);
            testState <= testState + 1;
        endrule

        rule checkPow (testState == 12);
            uut.setupCalculation(Pow, tagged Unsigned 2, tagged Unsigned 12);
            testState <= testState + 1;
        endrule

       rule printResults (unpack(pack(testState)[0]));
           let res <- uut.getResult();
           if(res matches tagged Unsigned .v)
               $display("Result: %d", v);
           else if(res matches tagged Signed .v)
               $display("Result: %d", v);
           testState <= testState + 1;
       endrule

        rule endSim (testState == 14);
            $finish();
        endrule
    endmodule
endpackage
```

Aufgabe 2.2.2 Maybe

Die `Tagged Union` `Maybe` ist im Prelude von Bluespec enthalten:

```
typedef union tagged {
  void Invalid;
  data_t Valid;
} Maybe #(type data_t) deriving (Eq, Bits);
```

Nutzen Sie `Maybe` um einen Zähler zu erstellen. Der Zähler hat dabei folgendes Interface:

```
interface SimpleCounter;
  method Action incr(UInt#(32) v);
  method Action decr(UInt#(32) v);
  method UInt#(32) counterValue();
endinterface
```

Die beiden Methoden `incr` und `decr` sollen dabei gleichzeitig Ausführbar sein. Nutzen Sie dafür zwei RWire die in einer gemeinsamen Rule abgefragt und in den entsprechenden Methoden gesetzt werden:

```
interface RWire#(type element_type) ;
  method Action wset(element_type datain) ;
  method Maybe#(element_type) wget() ;
endinterface: RWire
```

Vergessen Sie nicht Ihr Modul zu testen.

```
1  module mkSimpleCounter(SimpleCounter);
2    RWire#(UInt#(32)) incrWire <- mkRWire();
3    RWire#(UInt#(32)) decrWire <- mkRWire();
4
5    Reg#(UInt#(32)) cntr <- mkReg(0);
6
7    rule count;
8      let counterVal = cntr;
9      Maybe#(UInt#(32)) maybeIncr = incrWire.wget();
10     Maybe#(UInt#(32)) maybeDecr = decrWire.wget();
11
12     UInt#(32) incrVal = 0;
13     UInt#(32) decrVal = 0;
14
15     if(isValid(maybeIncr)) begin
16       incrVal = fromMaybe(?, maybeIncr);
17     end
18     if(isValid(maybeDecr)) begin
19       decrVal = fromMaybe(?, maybeDecr);
20     end
21
22     cntr <= cntr + incrVal - decrVal;
23   endrule
24
25   method Action incr(UInt#(32) v);
26     incrWire.wset(v);
27   endmethod
28
29   method Action decr(UInt#(32) v);
30     decrWire.wset(v);
31   endmethod
32
33   method UInt#(32) counterValue();
34     return cntr;
35   endmethod
36 endmodule
37
38 module mkCounterTest(Empty);
39   SimpleCounter uut <- mkSimpleCounter();
40   Stmt testbench = {
41     seq
42       action
43         uut.incr(5);
44       endaction
45       action
46         $display("%d", uut.counterValue());
47         uut.incr(5);
48         uut.decr(6);
49       endaction
50       action
51         $display("%d", uut.counterValue());
52         uut.decr(4);
53       endaction
54       action
```

```
55          $display("%d", uut.counterValue());
56        endaction
57      endseq
58    };
59
60    mkAutoFSM(testbench);
61  endmodule
```

Aufgabe 2.2.3 Maybe 2

Erweitern Sie das Interface um eine Methode `load` mit der man den Zählerstand setzen kann. Diese Methode soll Zeitgleich mit `incr` und `decr` aufrufbar sein.

```
1  interface SimpleCounter;
2    method Action incr(UInt#(32) i);
3    method Action decr(UInt#(32) d);
4    method Action load(UInt#(32) l);
5    method UInt#(32) counterValue();
6  endinterface
7
8  module mkSimpleCounter(SimpleCounter);
9    RWire#(UInt#(32)) incrWire <- mkRWire();
10   RWire#(UInt#(32)) decrWire <- mkRWire();
11   RWire#(UInt#(32)) loadWire <- mkRWire();
12
13   Reg#(UInt#(32)) cntr <- mkReg(0);
14
15   rule count;
16       let counterVal = cntr;
17       Maybe#(UInt#(32)) maybeIncr = incrWire.wget();
18       Maybe#(UInt#(32)) maybeDecr = decrWire.wget();
19       Maybe#(UInt#(32)) maybeLoad = loadWire.wget();
20
21       UInt#(32) incrVal = fromMaybe(0, maybeIncr);
22       UInt#(32) decrVal = fromMaybe(0, maybeDecr);
23       UInt#(32) baseVal = fromMaybe(cntr, maybeLoad);
24
25       cntr <= baseVal + incrVal - decrVal;
26   endrule
27
28   method Action incr(UInt#(32) v);
29     incrWire.wset(v);
30   endmethod
31
32   method Action decr(UInt#(32) v);
33     decrWire.wset(v);
34   endmethod
35
36   method Action load(UInt#(32) v);
37     loadWire.wset(v);
38   endmethod
39
40   method UInt#(32) counterValue();
41     return cntr;
42   endmethod
```

```
endmodule

module mkCounterTest(Empty);
  SimpleCounter uut <- mkSimpleCounter();
  Stmt testbench = {
    seq
      action
        uut.incr(5);
      endaction
      action
        $display("%d", uut.counterValue());
        uut.incr(5);
        uut.decr(6);
      endaction
      action
        $display("%d", uut.counterValue());
        uut.decr(4);
      endaction
      action
        uut.load(1024);
        uut.incr(42);
        uut.decr(48);
        $display("%d", uut.counterValue());
      endaction
      action
        $display("%d", uut.counterValue());
      endaction
    endseq
  };

  mkAutoFSM(testbench);
endmodule
```

Aufgabe 2.3 Nested Interfaces

In Bluespec kann man Interfaces beliebig Schachteln. Dies kann zum Beispiel dazu genutzt werden bestimmte Teile eines Interfaces wiederzuverwenden.

Führen Sie die Berechnung $((((x + a) \times b) \times c)/4) + 128$ in einer Pipeline aus. Die Parameter a, b und c sollen dabei zur Laufzeit veränderbar sein. Nutzen Sie das folgende Interface:

```
interface CalcUnit;
  method Action put(Int#(32) v);
  method ActionValue#(Int#(32)) result;
endinterface

interface CalcUnitChangeable;
  interface CalcUnit calc;
  method Action setParameter(Int#(32) param);
endinterface
```

Schalten Sie dabei zwischen die jeweiligen Stufen der Pipeline eine einelementige `FIFO`. Das kombinierende Modul soll auch das `CalcUnit` Interface implementieren. Nutzen Sie zum Speichern der Interfaces folgenden Vektor:

```
Vector#(5,CalcUnit) calcUnits;
```

```
import FIFO :: *;

```

```
module mkChangeableUnit#(function Int#(32) f(Int#(32) a, Int#(32) b))(CalcUnitChangeable);
  Reg#(Int#(32)) p <- mkReg(0);
  Wire#(Int#(32)) a <- mkWire();
  FIFO#(Int#(32)) r <- mkFIFO();

  rule doCalc;
    r.enq(f(a, p));
  endrule

  method Action setParameter(Int#(32) param);
    p <= param;
  endmethod

  interface CalcUnit calc;
    method Action put(Int#(32) v);
      a <= v;
    endmethod

    method ActionValue#(Int#(32)) result;
      r.deq();
      return r.first();
    endmethod
  endinterface
endmodule

module mkCalcUnit#(function Int#(32) f(Int#(32) a))(CalcUnit);
  Wire#(Int#(32)) a <- mkWire();
  FIFO#(Int#(32)) r <- mkFIFO();

  rule calc;
    r.enq(f(a));
  endrule

  method Action put(Int#(32) v);
    a <= v;
  endmethod

  method ActionValue#(Int#(32)) result;
    r.deq();
    return r.first();
  endmethod
endmodule

//$((((x + a) * b) * c) / 4) + 128$

module mkSomeCalculation(CalcUnit);
  Reg#(Int#(32)) a <- mkReg(42);
  Reg#(Int#(32)) b <- mkReg(2);
  Reg#(Int#(32)) c <- mkReg(4);
  function addFun(x,y) = x + y;
  function timesFun(x,y) = x * y;
  function divBy4Fun(x) = x / 4;
  function add128Fun(x) = x + 128;

  CalcUnitChangeable addA   <- mkChangeableUnit(addFun);
```

```
58    CalcUnitChangeable timesB <- mkChangeableUnit(timesFun);
59    CalcUnitChangeable timesC <- mkChangeableUnit(timesFun);
60    Vector#(5,CalcUnit) calcUnits;
61    calcUnits[0] = addA.calc;
62    calcUnits[1] = timesB.calc;
63    calcUnits[2] = timesC.calc;
64    calcUnits[3] <- mkCalcUnit(divBy4Fun);
65    calcUnits[4] <- mkCalcUnit(add128Fun);
66
67    Reg#(Bool) initialised <- mkReg(False);
68    rule initialise (!initialised);
69      initialised <= True;
70      addA.setParameter(a);
71      timesB.setParameter(b);
72      timesC.setParameter(c);
73    endrule
74
75    FIFO#(Int#(32)) inFIFO  <- mkFIFO();
76    FIFO#(Int#(32)) outFIFO <- mkFIFO();
77
78    for(Integer i = 1; i < 5; i = i + 1) begin
79      rule calc;
80        let t <- calcUnits[i - 1].result();
81        calcUnits[i].put(t);
82      endrule
83    end
84
85    rule setupCalc;
86      calcUnits[0].put(inFIFO.first());
87      inFIFO.deq();
88    endrule
89
90    rule outputResult;
91      let result <- calcUnits[4].result();
92      outFIFO.enq(result);
93    endrule
94
95    method Action put(Int#(32) v);
96      inFIFO.enq(v);
97    endmethod
98
99    method ActionValue#(Int#(32)) result;
100     outFIFO.deq();
101     return outFIFO.first();
102   endmethod
103 endmodule
104
105 module testCalculations(Empty);
106   CalcUnit uut <- mkSomeCalculation();
107   Reg#(Int#(32)) cntr <- mkReg(0);
108
109   rule printResult;
110     $display("(%0d) Result: %d", $time, uut.result());
111   endrule
112
```

```
113     rule putData;
114       $display("(%0d) Put: %d", $time, cntr);
115       uut.put(cntr);
116     endrule
117
118     rule countUp;
119       cntr <= cntr + 1;
120     endrule
121
122     rule endIt (cntr == 40);
123       $finish();
124     endrule
125   endmodule
```