Übung zur Vorlesung Architekturen und Entwurf von Rechnersystemen

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 16/17
Übungsblatt 2 - Lösungsvorschlag

Aufgabe 2.1 FSM in Bluespec

Finite State Machine (FSM) werden in Hardware häufig gebraucht, um 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:

```
exprPrimary ::= seqFsmStmt | parFsmStmt
fsmStmt     ::= exprFsmStmt
              | seqFsmStmt
              | parFsmStmt
              | ifFsmStmt
              | whileFsmStmt
              | repeatFsmStmt
              | forFsmStmt
              | returnFsmStmt
exprFsmStmt ::= regWrite ;
              | expression ;
seqFsmStmt  ::= seq fsmStmt { fsmStmt } endseq
parFsmStmt  ::= par fsmStmt { fsmStmt } endpar
ifFsmStmt   ::= if expression fsmStmt
              [ else fsmStmt ]
whileFsmStmt    ::= while ( expression )
                    loopBodyFsmStmt
forFsmStmt      ::= for ( fsmStmt ; expression ; fsmStmt )
                    loopBodyFsmStmt
returnFsmStmt   ::= return ;
repeatFsmStmt   ::= repeat ( expression )
                    loopBodyFsmStmt
loopBodyFsmStmt ::= fsmStmt
                  | break ;
                  | continue ;
```

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

```
Stmt myFirstFSM = {
  seq
    action
      $display("Hello World.");
    endaction
  endseq
};
```

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.

```
module mkSecondFSM(Empty);
Reg#(Bool) syncVar <- mkReg(False);
Stmt secondStmt = {
  par
    seq
      $display("(%0d) Part one starts.", $time);
      delay(100);
      syncVar <= True;
      $display("(%0d) Part one done.", $time);
    endseq
    seq
      repeat(10) $display("(%0d) Print this 10 times.", $time);
```

```
13        await(syncVar);
14        $display("(%0d) Part two done.", $time);
15      endseq
16    endpar
17  };
18  mkAutoFSM(secondStmt);
19  endmodule
```

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 werden. 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:

```
1  interface PulseWire;
2    method Action send();
3    method Bool _read();
4  endinterface
```

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

```
1  module mkThirdFSM(Empty);
2    Reg#(UInt#(12)) counter <- mkReg(0);
3    PulseWire pw <- mkPulseWire();
4    Reg#(UInt#(12)) i <- mkReg(0);
5
6    rule count (counter < 99);
7      counter <= counter + 1;
8    endrule
9
10   rule resetCount (counter == 99);
11     counter <= 0;
12     pw.send();
13   endrule
14
15   Stmt thirdStmt = {
16     seq
17       for(i <= 0; i < 20; i <= i + 1) seq
18         $display("(%0d) Iteration %d.",$time,i);
19       endseq
20       $finish();
21     endseq
22   };
23   FSM myFSM <- mkFSMWithPred(thirdStmt, pw);
24   rule startFSM (myFSM.done());
25     myFSM.start();
26   endrule
27 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};
```

```
import Vector::*;

typedef enum {Mul,Div,Add,Sub,And,Or,Pow} AluOps deriving (Eq, Bits, FShow);

typedef struct {
  Int#(32) opA;
  Int#(32) opB;
  AluOps    operator;
  Int#(32) expectedResult;
} TestData deriving (Eq, Bits);


module mkAluFSMTB(Empty);
  Vector#(5, TestData) myVector;
  myVector[0] = TestData {opA: 2, opB: 4, operator: Add, expectedResult: 6};
  myVector[1] = TestData {opA: 2, opB: 4, operator: Mul, expectedResult: 8};
  myVector[2] = TestData {opA: 4, opB: 2, operator: Div, expectedResult: 2};
  myVector[3] = TestData {opA: 4, opB: 0, operator: Pow, expectedResult: 1};
  myVector[4] = TestData {opA: 4, opB: 4, operator: Pow, expectedResult: 256};

  Reg#(UInt#(32)) dataPtr <- mkReg(0);

  HelloALU uut <- mkHelloALU();

  Stmt checkStmt = {
    seq
      action
        let currentData = myVector[dataPtr];
        uut.setupCalculation(currentData.operator, currentData.opA, currentData.opB);
      endaction
      action
        let currentData = myVector[dataPtr];
        let result <- uut.getResult();
        let print  = $format("Calculation: %d ", currentData.opA) + fshow(currentData.operator) +
↪  $format("%d", currentData.opB);
        $display(print);
        if(result == currentData.expectedResult) begin
          $display("Result correct: %d", result);
```

```
38          end else begin
39            $display("Result incorrect: %d != ", result, currentData.expectedResult);
40          end
41        endaction
42      endseq
43    };
44
45    FSM checkFSM <- mkFSM(checkStmt);
46
47    Stmt mainFSM = {
48      seq
49        for(dataPtr <= 0; dataPtr < 5; dataPtr <= dataPtr + 1) seq
50          checkFSM.start();
51          checkFSM.waitTillDone();
52        endseq
53      endseq
54    };
55    mkAutoFSM(mainFSM);
56  endmodule
```

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.

```
1  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:

```
1  rule someRule (unionReg matches tagged Signed .v);
2    $display("%d", v);
3  endrule;
4  rule anotherRule (unionReg matches tagged Unsigned .v);
5    $display("%u", v);
6  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`.

```
1  package Alu;
2      typedef enum {Mul,Div,Add,Sub,And,Or,Pow} AluOps deriving (Eq, Bits, FShow);
3      typedef union tagged {UInt#(32) Unsigned; Int#(32) Signed;} SignedOrUnsigned deriving(Bits,
   ↪  Eq);
4
5      interface Power#(type t);
6          method Action   setOperands(t a, t b);
7          method t getResult();
8      endinterface
9
10     module mkPower(Power#(t))
11         provisos(Bits#(t, t_sz),
```

```
12                  Ord#(t),
13                  Arith#(t),
14                  Eq#(t));
15          Reg#(Bool) resultValid <- mkReg(False);
16
17          Reg#(t) opA    <- mkReg(0);
18          Reg#(t) opB    <- mkReg(0);
19          Reg#(t) result <- mkReg(1);
20
21          rule calc (opB > 0);
22              opB <= opB - 1;
23              result <= result * opA;
24          endrule
25
26          rule calcDone (opB == 0 && !resultValid);
27              resultValid <= True;
28          endrule
29
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 &&&
    ↪ newOperands);
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 &&& newOperands);
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(opA matches tagged Signed .va &&& opB matches tagged Signed .vb)
    ↪  powInt.setOperands(va,vb);
109                 else if(opA matches tagged Unsigned .va &&& opB matches tagged Unsigned .vb)
    ↪  powUInt.setOperands(va,vb);
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);
140             uut.setupCalculation(Sub, tagged Unsigned 12, tagged Unsigned 4);
141             testState <= testState + 1;
142         endrule
143
144         rule checkAnd (testState == 8);
145             uut.setupCalculation(And, tagged Unsigned 32'hA, tagged Unsigned 32'hA);
146             testState <= testState + 1;
147         endrule
148
149         rule checkOr (testState == 10);
150             uut.setupCalculation(Or, tagged Unsigned 32'hA, tagged Unsigned 32'hA);
151             testState <= testState + 1;
152         endrule
153
154         rule checkPow (testState == 12);
155             uut.setupCalculation(Pow, tagged Unsigned 2, tagged Unsigned 12);
156             testState <= testState + 1;
157         endrule
158
159         rule printResults (unpack(pack(testState)[0]));
160             $display("Result: %d", uut.getResult());
161             testState <= testState + 1;
162         endrule
163
164         rule endSim (testState == 14);
165             $finish();
166         endrule
167     endmodule
168 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.

```
module mkSimpleCounter(SimpleCounter);
  RWire#(UInt#(32)) incrWire <- mkRWire();
  RWire#(UInt#(32)) decrWire <- mkRWire();

  Reg#(UInt#(32)) cntr <- mkReg(0);

  rule count;
    let counterVal = cntr;
    Maybe#(UInt#(32)) maybeIncr = incrWire.wget();
    Maybe#(UInt#(32)) maybeDecr = decrWire.wget();

    UInt#(32) incrVal = 0;
    UInt#(32) decrVal = 0;

    if(isValid(maybeIncr)) begin
      incrVal = fromMaybe(?, maybeIncr);
    end
    if(isValid(maybeDecr)) begin
      decrVal = fromMaybe(?, maybeDecr);
    end

    cntr <= cntr + incrVal - decrVal;
  endrule

  method Action incr(UInt#(32) v);
    incrWire.wset(v);
  endmethod

  method Action decr(UInt#(32) v);
    decrWire.wset(v);
  endmethod

  method UInt#(32) counterValue();
    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
43    endmodule
44
45    module mkCounterTest(Empty);
46      SimpleCounter uut <- mkSimpleCounter();
47      Stmt testbench = {
48        seq
49          action
50            uut.incr(5);
51          endaction
52          action
53            $display("%d", uut.counterValue());
54            uut.incr(5);
55            uut.decr(6);
56          endaction
57          action
58            $display("%d", uut.counterValue());
59            uut.decr(4);
60          endaction
61          action
62            uut.load(1024);
63            uut.incr(42);
64            uut.decr(48);
65            $display("%d", uut.counterValue());
66          endaction
67          action
68            $display("%d", uut.counterValue());
69          endaction
70        endseq
71      };
72
73      mkAutoFSM(testbench);
74    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

```

```
36    method Action put(Int#(32) v);
37      a <= v;
38    endmethod
39
40    method ActionValue#(Int#(32)) result;
41      r.deq();
42      return r.first();
43    endmethod
44  endmodule
45
46  //$((((x + a) * b) * c) / 4) + 128$
47
48  module mkSomeCalculation(CalcUnit);
49    Reg#(Int#(32)) a <- mkReg(42);
50    Reg#(Int#(32)) b <- mkReg(2);
51    Reg#(Int#(32)) c <- mkReg(4);
52    function addFun(x,y) = x + y;
53    function timesFun(x,y) = x * y;
54    function divBy4Fun(x) = x / 4;
55    function add128Fun(x) = x + 128;
56
57    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
```