Vorlesung Architekturen und Entwurf von Rechnersystemen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2018/2019
Übungsblatt 2
Prof. Andreas Koch, Jaco Hofmann

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:

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

Aufgabe 2.1.2 Parallele 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?

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:

```
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.

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

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`.

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.

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.

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;
```