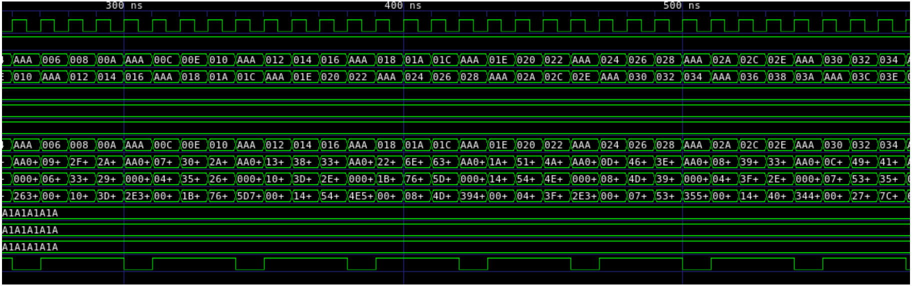
Architekturen und Entwurf von Rechnersystemen
Wintersemester 2016/2017

Hörsaalübung 3: Noch mehr Bluespec Grundlagen



Übung 2

Übung 2

- Vertiefung der Inhalte aus Hörsaalübung 1
- Verschiedene Einsatzzwecke von FSM
- Tagged Unions im Einsatz
- Neues Thema: Nested Interfaces

Aufgabe 2.1.1

Eine erste FSM

- Eine FSM erstellen die nach 100 Taktzyklen eine Nachricht ausgibt:
  - Vertraut werden mit der Syntax von StmtFSM
  - Eigenschaften der Ausführungszeit betrachten

Eine erste FSM

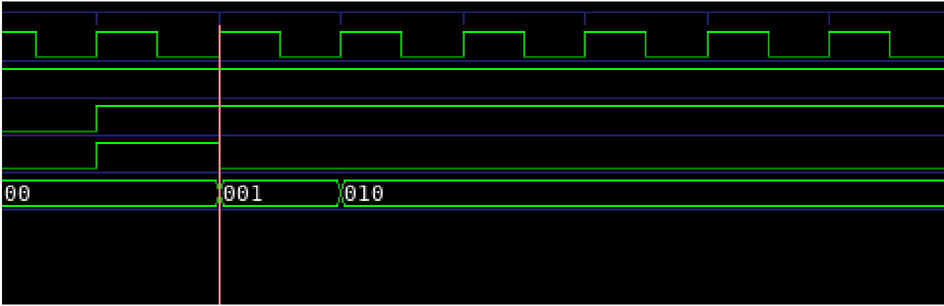
TECHNISCHE UNIVERSITÄT DARMSTADT

```
package FSMTests;

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

Eine erste FSM

```
> ./out
(1020) Hello World!
```

- 10 ns Taktlänge → 102 Takte



Aufgabe 2.1.2

Parallele Ausführung in FSM

- StmtFSM ermöglicht nicht nur sequentielle Ausführung
  - `par`-Umgebung
- Alle Anweisungen in einer `par` Umgebung laufen parallel ab
- FSM läuft erst weiter wenn alle Teile der Umgebung fertig sind
- Erstellen Sie eine FSM mit paralleler Ausführung:
  1. Zähle 100 Taktzyklen und setze Synchronisationsregister
  2. Gib eine Nachricht 10 mal aus und wartet dann auf Synchronisationsregister

Parallele Ausführung in FSM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkSecondFSM(Empty);
Reg#(Bool) syncVar <- mkReg(False);
Stmt secondStmt = {
  seq
  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);
      await(syncVar);
      $display("(%0d) Part two done.", $time);
    endseq
  endpar
  $display("(%0d) Everything is done.", $time);
  endseq
};
mkAutoFSM(secondStmt);
endmodule
```

Parallele Ausführung in FSM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# (15) Part one starts.
# (15) Print this 10 times.
# (25) Print this 10 times.
# (35) Print this 10 times.
# (45) Print this 10 times.
# (55) Print this 10 times.
# (65) Print this 10 times.
# (75) Print this 10 times.
# (85) Print this 10 times.
# (95) Print this 10 times.
# (105) Print this 10 times.
# (1035) Part one done.
# (1045) Part two done.
# (1055) Everything is done.
```

- `par` Block wartet bis alle Teile fertig sind.

Aufgabe 2.1.3

FSM Ausführung steuern

- Häufig ist der Systemtakt zu schnell für bestimmte Aufgaben
- Ausführung der FSM kann gesteuert werden: Zusätzliche Guard
- Aufgabe:
  1. `mkFSMWithPred`: FSM mit $\frac{1}{100}$ des Systemtakts ansteuern
  2. `PulseWire` zur Synchronisation nutzen

FSM Ausführung steuern

TECHNISCHE UNIVERSITÄT DARMSTADT

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

FSM Ausführung steuern

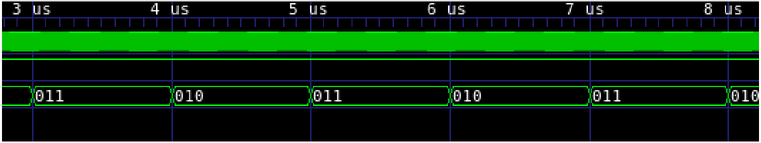
```
rule count (counter < 99);
  counter <= counter + 1;
endrule

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

FSM Ausführung steuern

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# (1995) Iteration    0.
# (3995) Iteration    1.
# (5995) Iteration    2.
# (7995) Iteration    3.
# (9995) Iteration    4.
  .....
```

- Jede Iteration braucht zwei Zyklen. Warum?
- Waveform ansehen.



Aufgabe 2.1.5

FSM als Testbench

- FSM hervorragend für Testbenches geeignet
- StmtFSM bietet einige Hilfen dafür
- Häufig genutzte Teile können ausgelagert werden

FSM als Testbench


```
import Vector::*;

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 Declarations
endmodule
```

FSM als Testbench

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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);
      end else begin
        $display("Result incorrect: %d != ", result,
                                currentData.expectedResult);
      end
    endaction
  endseq
};
```

FSM als Testbench

TECHNISCHE UNIVERSITÄT DARMSTADT

```
FSM checkFSM <- mkFSM(checkStmt);
Stmt mainFSM = {
  seq
    for(dataPtr <= 0; dataPtr < 5; dataPtr <= dataPtr + 1) seq
      checkFSM.start();
      checkFSM.waitTillDone();
    endseq
  endseq
};
mkAutoFSM(mainFSM);
```

FSM als Testbench

```
# Calculation:           2 Add          4
# Result correct:          6
# Calculation:           2 Mul          4
# Result correct:          8
# Calculation:           4 Div          2
# Result correct:          2
# Calculation:           4 Pow          0
# Result correct:          1
# Calculation:           8 Div          4
# Result incorrect:          2 !=          42
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Aufgabe 2.2.1

Flexible ALU

TECHNISCHE UNIVERSITÄT DARMSTADT

- ALU Kompatibel mit `UInt` und `Int` machen.
- Einsatz von `Tagged Union`

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

Flexible ALU: Power generisch machen

TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
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);
    ...
```

Flexible ALU: Welche Provisos sind nötig?

```
The following additional provisos are needed:
  Eq#(t)
    Introduced at the following locations:
"Alu.bsv", line 26, column 28
  Ord#(t)
    Introduced at the following locations:
"Alu.bsv", line 21, column 24
  Bits#(t, a__)
    Introduced at the following locations:
"Alu.bsv", line 19, column 27
"Alu.bsv", line 18, column 27
"Alu.bsv", line 17, column 27
  Arith#(t)
    Introduced at the following locations:
"Alu.bsv", line 23, column 30
"Alu.bsv", line 22, column 24
```

Flexible ALU

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface HelloALU;
  method Action setupCalculation(AluOps op, SignedOrUnsigned a,
                                             SignedOrUnsigned b);
  method ActionValue#(SignedOrUnsigned)  getResult();
endinterface
```

Flexible ALU

```
module mkHelloALU(HelloALU);
    ...
    Reg#(SignedOrUnsigned) opA    <- mkReg(tagged Signed 0);
    Reg#(SignedOrUnsigned) opB    <- mkReg(tagged Signed 0);
    Reg#(SignedOrUnsigned) result <- mkReg(tagged Signed 0);
    Power#(UInt#(32)) powUInt <- mkPower();
    Power#(Int#(32))  powInt  <- mkPower();
    rule calculateSigned (opA matches tagged Signed .va
                      &&& opB matches tagged Signed .vb
                      &&& newOperands);
            Int#(32) rTmp = 0;
            case(operation)
                Mul: rTmp = va * vb;
            ....
    endrule
    rule calculateUnsigned (opA matches tagged Unsigned .va
                        &&& opB matches tagged Unsigned .vb
                        &&& newOperands);
            UInt#(32) rTmp = 0;
            case(operation)
                Mul: rTmp = va * vb;
            ...
```

Flexible ALU: Ungültige Eingaben verwerfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
function Bool isUnsigned(SignedOrUnsigned v);
    if(v matches tagged Unsigned .va) return True;
    else return False;
endfunction

rule dumpInvalid (newOperands && isUnsigned(opA) != isUnsigned(opB));
    $display("Invalid combination of Signed and Unsigned Operands");
    newOperands <= False;
    resultValid <= False;
endrule
```

Aufgabe 2.2.2

Maybe?

TECHNISCHE UNIVERSITÄT DARMSTADT

- `Maybe` bereits aus der Vorlesung bekannt
- Entweder `Invalid` oder `Valid t`
- Wird hier genutzt um einen Zähler zu Implementieren
- Zähler hat `incr` und `decr` Methoden → Im gleichen Takt
- Dafür RWire benutzen

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

Maybe?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkSimpleCounter(SimpleCounter);
  RWire#(UInt#(32)) incrWire <- mkRWire();
  RWire#(UInt#(32)) decrWire <- mkRWire();
  Reg#(UInt#(32)) cntr <- mkReg(0);
  // Rules...
  ////////
  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;
  endmethod
endmodule
```

Maybe?

```
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
```

Aufgabe 2.2.3

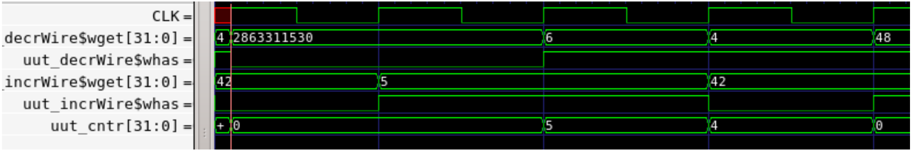
Maybe!

- `load` Methode hinzufügen
- Gleichzeitig Vereinfachung der Methode

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

  UInt#(32) incrVal = fromMaybe(0, maybeIncr);
  UInt#(32) decrVal = fromMaybe(0, maybeDecr);
  UInt#(32) baseVal = fromMaybe(cntr, maybeLoad);

  cntr <= baseVal + incrVal - decrVal;
endrule
```

Maybe!

- cntr + 5
- cntr + 5 - 6
- cntr - 4
- 1024 + 42 - 48



Aufgabe 2.3

Nested Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

- Möglichkeit zur Wiederverwendung von Interfaces
- Standardisierte Schnittstellen
- Aufgabe: Berechnung von $((((x + a) \times b) \times c)/4) + 128$
- Parameter a, b und c können zur Laufzeit verändert werden

```
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
```

Nested Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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
```

Nested Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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
```

Nested Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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);
  CalcUnitChangeable timesB <- mkChangeableUnit(timesFun);
  CalcUnitChangeable timesC <- mkChangeableUnit(timesFun);
  Vector#(5,CalcUnit) calcUnits;
  calcUnits[0] = addA.calc;
  calcUnits[1] = timesB.calc;
  calcUnits[2] = timesC.calc;
  calcUnits[3] <- mkCalcUnit(divBy4Fun);
  calcUnits[4] <- mkCalcUnit(add128Fun);
```

Nested Interfaces

```
Reg#(Bool) initialised <- mkReg(False);
rule initialise (!initialised);
  initialised <= True;
  addA.setParameter(a);
  timesB.setParameter(b);
  timesC.setParameter(c);
endrule

rule setupCalc;
  calcUnits[0].put(inFIFO.first());
  inFIFO.deq();
endrule

rule outputResult;
  let result <- calcUnits[4].result();
  outFIFO.enq(result);
endrule
```

Nested Interfaces

```
for(Integer i = 1; i < 5; i = i + 1) begin
  rule calc;
    let t <- calcUnits[i - 1].result();
    calcUnits[i].put(t);
  endrule
end
```

Übung 3

Übung 3

TECHNISCHE UNIVERSITÄT DARMSTADT

- BlueCheck: Automatisiertes Testen
- Basiert auf QuickCheck
- Kann sehr effizient sein:
  - Cache-System eines Prozessors
  - BSD erfolgreich gebootet
  - BlueCheck findet Fehler in 10 Methodenaufrufen

Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module [BlueCheck] mkArithSpec ();
  function Bool addComm(Int#(4) x, Int#(4) y) =
    x + y == y + x;

  function Bool addAssoc(Int#(4) x, Int#(4) y, Int#(4) z) =
    x + (y + z) == (x + y) + z;

  function Bool subComm(Int#(4) x, Int#(4) y) =
    x - y == y - x;

  prop("addComm"  , addComm);
  prop("addAssoc" , addAssoc);
  prop("subComm" , subComm);
endmodule
```

Golden Sample

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module [BlueCheck] checkStack ();
  /* Specification instance */
  Stack#(8, Bit#(4)) spec <- mkStackSpec();

  /* Implmentation instance */
  Stack#(8, Bit#(4)) imp <- mkBRAMStack();

  equiv("pop"    , spec.pop    , imp.pop);
  equiv("push"   , spec.push   , imp.push);
  equiv("isEmpty", spec.isEmpty, imp.isEmpty);
  equiv("top"    , spec.top    , imp.top);
endmodule
```