Architekturen und Entwurf von Rechnersystemen Wintersemester 2016/17

Hörsaalübung 2: Übung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE UNIVERSITÄT DARMSTADT

Überblick

Was erwartet Sie?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiellösung Übung 1
- Nachtrag zur letzten Vorlesung
- Vorschau auf Übung 2

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Übung 1: Bluespec

Hello Bluespec

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kompilieren eines ersten Bluespec Moduls
- Primitives erstes Programm in Bluespec

```
1  package HelloBluespec;
2    module mkHelloBluespec(Empty);
3      rule helloDisplay;
4        $display("(%0d) Hello World!", $time);
5      endrule
6    endmodule
7  endpackage
```

- Gab es Probleme mit der Bluespec Simulation/Einrichtung?

Blinky

TECHNISCHE UNIVERSITÄT DARMSTADT

- Erstes einfaches Hardwaremodul
- 100 MHz Takt auf ca. 3 Hz Takt runterbrechen
- Mit langsamen Takt LED ansteuern
- Was ist zu tun?
  - Langsamen Takt mit Zähler generieren
  - Interface mit LED Ausgang erstellen
  - Das neue Modul testen
  - Erzeugte Hardware Taktgenau analysieren

Rahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   package HelloBluespec;
2     module mkHelloBluespec(Empty);
3
4
5
6
7
8
9     endmodule
10  endpackage
```

- Typischer Rahmen eines BSV Packages

Zähler

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   package HelloBluespec;
2     module mkHelloBluespec(Empty);
3       Reg#(UInt#(25)) counter <- mkReg(0);
4
5
6
7
8
9       rule count;
10        counter <= counter + 1; // Überlauf ausnutzen
11      endrule
12    endmodule
13  endpackage
```

- Zähler von Typ `UInt#(25)` in Register
- `rule` die Zählerregister erhöht
- Automatisches zurücksetzen durch Wahl der Bitbreite des Registers

Zähler

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  package HelloBluespec;
2    module mkHelloBluespec(Empty);
3      Reg#(UInt#(25)) counter <- mkReg(0);
4
5      rule helloDisplay (counter == 0);
6        $display("(%0d) Hello World!", $time);
7      endrule
8
9      rule count;
10       counter <= counter + 1;
11     endrule
12   endmodule
13 endpackage
```

- Ausgabe wenn Zählerregister resettet wurde

LED hinzufügen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  interface HelloBluespec;
2    (* always_enabled, always_ready *) method Bool led();
3  endinterface
```

- Neues Sprachkonstrukt `Attribute`
- Weist den Compiler an nachfolgende Objekte anders zu verarbeiten
- In diesem Fall wird der Compiler angewiesen:
  - Methode `led` benötigt kein `enable` Signal
  - Methode `led` benötigt kein `ready` Signal
  - $\rightarrow$ Methode `led` benötigt keinen Handshake
- Compiler überprüft die Einhaltung der Bedingung

LED hinzufügen: Handshake

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   BSV: method Bool led();
2   Verilog:
3   module mkHelloBluespec(CLK,
4              RST_N,
5
6              led,
7              RDY_led);
8     input  CLK;
9     input  RST_N;
10
11    // value method led
12    output led;
13    output RDY_led;
14  ...
```

```
1   BSV: (* always_enabled, always_ready *)
2                       method Bool led();
3   Verilog:
4   module mkHelloBluespec(CLK,
5              RST_N,
6
7              led);
8     input  CLK;
9     input  RST_N;
10
11    // value method led
12    output led;
```

LED hinzufügen: Das Modul

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkHelloBluespec(HelloBluespec);
2       Reg#(Bool) ledStatus <- mkReg(False);
3       Reg#(UInt#(25)) counter <- mkReg(0);
4
5       rule helloDisplay (counter == 0);
6           $display("(%0d) Hello World!", $time);
7           ledStatus <= !ledStatus;
8       endrule
9
10      rule count;
11        counter <= counter + 1;
12      endrule
13
14      method Bool led();
15         return ledStatus;
16      endmethod
17  endmodule
```

Blinky testen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Counter zählt 2 s ab (Zählen bis 200 000 000 bei 100 MHz).
- `rule` die Simulation beendet

```
1   module mkHelloTestbench(Empty);
2     Reg#(UInt#(32)) counter <- mkReg(0);
3
4     HelloBluespec uut <- mkHelloBluespec();
5
6      rule endSimulation (counter == 200000000);
7         $finish();
8      endrule
9
10     rule counterIncr;
11        counter <= counter + 1;
12     endrule
13  endmodule
```

Blinky testen: LED Statusänderung

TECHNISCHE UNIVERSITÄT DARMSTADT

- LED Statusänderung detektieren und ausgeben
- LED wird angeschaltet wenn LED vorher aus war und nun an ist
- LED wird ausgeschaltet wenn LED vorher an war und nun aus ist

```
1  Reg#(Bool) ledLastCycle <- mkReg(False);
2
3  rule checkLedStatus;
4    ledLastCycle <= uut.led();
5    if(ledLastCycle == True && uut.led() == False) $display("LED aus.");
6    else if(ledLastCycle == False && uut.led() == True) $display("LED an.");
7  endrule
```

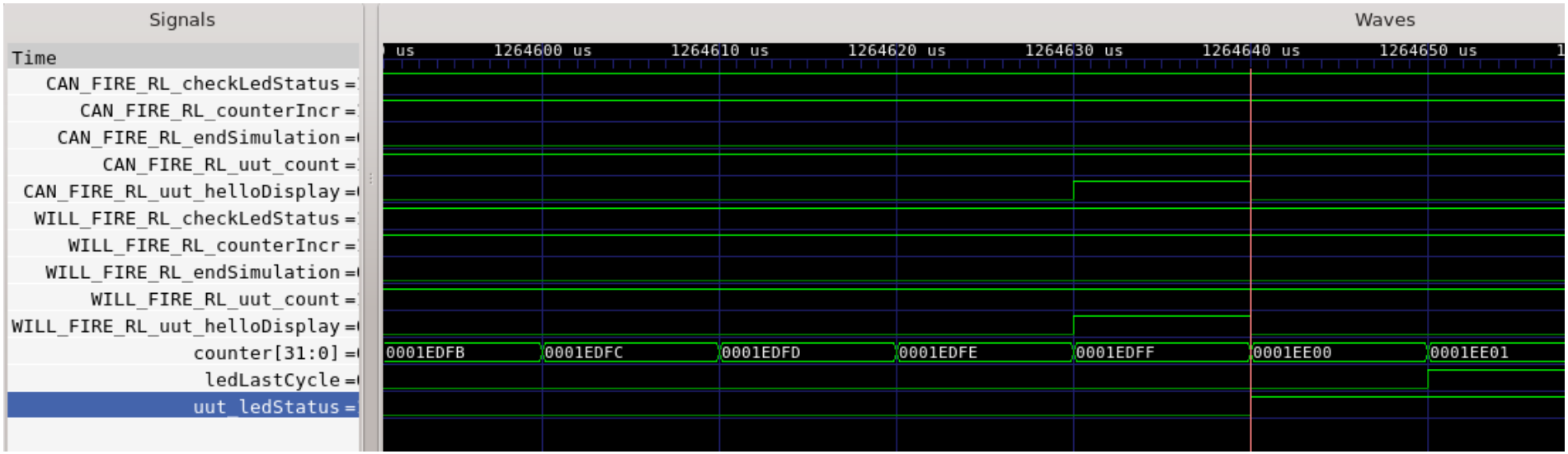
Blinky Taktgenau verfolgen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bluesim kann Waveforms erzeugen
- Taktgenaue Verfolgung der Ausführung
- Hilfssignale (`can_fire`, `will_fire`) mit `-keep-fires`.
- Bluesim wird dadurch um vielfaches langsamer
- Mögliche Optimierungen fallen weg

Blinky Waveform

TECHNISCHE UNIVERSITÄT DARMSTADT



ALU

TECHNISCHE UNIVERSITÄT DARMSTADT

- Arithmetic Logic Unit mit
  - Multiplizieren
  - Dividieren
  - Addieren
  - Subtrahieren
  - Logisches Und
  - Logisches Oder

ALU: Das Interface

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  interface HelloALU;
2      method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b);
3      method ActionValue#(Int#(32))  getResult();
4  endinterface
```

- Methode `setupCalculation` blockiert so lange eine Berechnung aktiv ist
- Methode `getResult` blockiert so lange bis ein Ergebnis vorliegt

ALU: Enum AluOps

TECHNISCHE UNIVERSITÄT DARMSTADT

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

- `AluOps` ist ein `enum` (bekannt aus C/C++/Java etc.)
- `deriving` für Nutzung von Typklassen (später mehr dazu)
  - `Eq` erlaubt Vergleich von Werten des Typen mit logischen Operatoren == und ! =. Einfachste Implementierung der Operatoren.
  - `Bits` erlaubt die Nutzung der Funktionen `pack` ($t \rightarrow Bit$) und `unpack` ($Bit \rightarrow t$). Wichtig für Hardwaresynthese.

ALU: Implementierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkHelloALU(HelloALU);
2       Reg#(Bool) newOperands <- mkReg(False);
3       Reg#(Bool) resultValid <- mkReg(False);
4       Reg#(AluOps) operation <- mkReg(Mul);
5       Reg#(Int#(32)) opA     <- mkReg(0);
6       Reg#(Int#(32)) opB     <- mkReg(0);
7       Reg#(Int#(32)) result  <- mkReg(0);
8
9       method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b)
10                                                          if(!newOperands);
11          opA <= a;
12          opB <= b;
13          operation <= op;
14          newOperands <= True;
15          resultValid <= False;
16      endmethod
17
18      method ActionValue#(Int#(32)) getResult() if(resultValid);
19          resultValid <= False;
20          return result;
21      endmethod
22  endmodule
```

ALU: Calculate Rule

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1       rule calculate (newOperands);
2           Int#(32) rTmp = 0;
3           case(operation)
4               Mul: rTmp = opA * opB;
5               Div: rTmp = opA / opB;
6               Add: rTmp = opA + opB;
7               Sub: rTmp = opA - opB;
8               And: rTmp = opA & opB;
9               Or:  rTmp = opA | opB;
10          endcase
11          result <= rTmp;
12          newOperands <= False;
13          resultValid <= True;
14      endrule
```

- Was ist wenn eine der Berechnungen länger als ein Takt braucht?

ALU: Potenzberechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die ALU wird um eine Potenzberechnung erweitert.

```
1  int pow(int a, int b) {
2    int tmp = 1;
3    for(int i = 0; i < b; ++i) {
4      tmp *= a;
5    }
6    return tmp;
7  }
```

```
1  interface Power;
2      method Action   setOperands(Int#(32) a, Int#(32) b);
3      method Int#(32) getResult();
4  endinterface
```

ALU: Potenzberechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1         Reg#(Bool) resultValid <- mkReg(False);
2         Reg#(Int#(32)) opA     <- mkReg(0);
3         Reg#(Int#(32)) opB     <- mkReg(0);
4         Reg#(Int#(32)) result  <- mkReg(1);
5     // Rules
6         ...
7     /////////
8         method Action setOperands(Int#(32) a, Int#(32) b);
9             result <= 1;
10            opA    <= a;
11            opB    <= b;
12            resultValid <= False;
13        endmethod
14
15        method Int#(32) getResult() if(resultValid);
16            return result;
17        endmethod
```

ALU: Potenzberechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1         rule calc (opB > 0);
2             opB <= opB - 1;
3             result <= result * opA;
4         endrule
5
6         rule calcDone (opB == 0 && !resultValid);
7             resultValid <= True;
8         endrule
```

ALU: Zusammenfügen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkHelloALU(HelloALU);
2       ...
3       Power pow               <- mkPower();
4       rule calculate (newOperands);
5           ...
6           case(operation)
7               ...
8               Pow: rTmp = pow.getResult();
9           endcase
10      endrule
11      ...
12      method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b)
13                                                      if(!newOperands);
14          ...
15          if(op == Pow) pow.setOperands(a,b);
16      endmethod
17      ...
18  endmodule
```

ALU: Testbench

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkALUTestbench(Empty);
2       HelloALU uut              <- mkHelloALU();
3       Reg#(UInt#(8)) testState <- mkReg(0);
4
5       rule checkMul (testState == 0);
6           uut.setupCalculation(Mul, 4,5);
7           testState <= testState + 1;
8       endrule
9       rule checkDiv (testState == 2);
10          uut.setupCalculation(Div, 12,4);
11          testState <= testState + 1;
12      endrule
13      ...
14      rule printResults (unpack(pack(testState)[0]));
15          $display("Result: %d", uut.getResult());
16          testState <= testState + 1;
17      endrule
18      rule endSim (testState == 14);
19          $finish();
20      endrule
21  endmodule
```

Teil 2: Nachtrag zur Vorlesung

Konflikte zwischen verschiedenen Modulen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Können `x.foo` und `y.foo` in Konflikt stehen?

Konflikte zwischen verschiedenen Modulen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Können `x.foo` und `y.foo` in Konflikt stehen?
- Antwort von Dienstag: Nein
- Dies ist nach dem Wissen aus der Vorlesung auch korrekt
- Allerdings: Bluespec hat sehr viele Features

Parameter in Bluespec

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1    interface TestIfc;
2        method Action foo();
3    endinterface
4
5    module mkTest1#(Reg#(Bool) testreg, Bool val)(TestIfc);
6        method Action foo();
7            testreg <= val;
8        endmethod
9    endmodule
```

- Parameter `testreg` und `val` werden an Modul übergeben.

Parameter in Bluespec: Testbench

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkTest(Empty);
2       Reg#(Bool) val <- mkReg(False);
3
4       TestIfc x <- mkTest1(val, True);
5       TestIfc y <- mkTest1(val, False);
6
7       rule writeX;
8           x.foo();
9       endrule
10
11      rule writeY;
12          y.foo();
13      endrule
14
15  endmodule
```

- Konflikt zwischen `x.foo` und `y.foo`.
- Zugriff auf gemeinsames Register `val`.

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Teil 3: Erste Einblicke in AzureIP

FSM in Bluespec

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkALUTestbench(Empty);
2       HelloALU uut                <- mkHelloALU();
3       Reg#(UInt#(8)) testState <- mkReg(0);
4       rule checkMul (testState == 0);
5           uut.setupCalculation(Mul, 4,5);
6           testState <= testState + 1;
7       endrule
8       rule checkDiv (testState == 2);
9           uut.setupCalculation(Div, 12,4);
10          testState <= testState + 1;
11      endrule
12      ...
13  endmodule
```

- FSM mit rules: Unübersichtlich und wenig Abstrakt
- Extra Zählvariable nur für den State
- State muss verwaltet werden

FSM in Bluespec: STMTFsm

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil der AzureIP Bibliothek
- Eigene Sprache für FSM
- Übersichtliche Darstellung von FSM
- Darstellung von sequentiellen Aktionen (for-/while-loop etc.)

FSM in Bluespec: STMT

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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 ]
```

FSM in Bluespec: STMT Part 2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1        whileFsmStmt     ::= while ( expression )
2                              loopBodyFsmStmt
3        forFsmStmt       ::= for ( fsmStmt ; expression ; fsmStmt )
4                              loopBodyFsmStmt
5        returnFsmStmt    ::= return ;
6        repeatFsmStmt    ::= repeat ( expression )
7                              loopBodyFsmStmt
8        loopBodyFsmStmt  ::= fsmStmt
9                            | break ;
10                           | continue ;
```

FSM in Bluespec: Testbench

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   import StmtFSM::*;
2   Stmt aluTestbench = {
3                       seq
4                         action
5                           uut.setupCalculation(Mul, 4, 5)
6                         endaction
7                         action
8                           $display("Result: %d", uut.getResult());
9                         endaction
10                        ...
11                      endseq
12  };
13  mkAutoFSM(aluTestbench);
```

- Wesentlich einfacher zu lesen und zu erweitern
- Verschiedene FSM Typen
  - `mkAutoFSM`: Führt FSM ein mal aus und beendet dann die Simulation
  - `mkFSM`: Ausführung startet wenn `start()` Methode aufgerufen wird
  - `mkFSMWithPred`: FSM mit extra Prädikat. Wird genutzt um Konflikte zwischen FSM zu vermeiden.

FSM in Bluespec: FSM wiederverwenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   Reg#(UInt#(32)) a         <- mkRegU;
2   Reg#(UInt#(32)) b         <- mkRegU;
3   Reg#(AluOp)     aluOp     <- mkRegU;
4   Reg#(UInt#(3))  activeFSM <- mkReg(0);
5   Stmt aluTestbench = {
6                       seq
7                         action
8                            activeFSM <= 1;
9                            a <= 4;
10                           b <= 5;
11                           aluOp <= Mul;
12                           testFSM.start();
13                        endaction
14                        await(testFSM.done());
15                        ...
16                        $finish();
17                      endseq
18  };
19  FSM testbenchFSM <- mkFSMWithPred(aluTestbench, activeFSM == 0);
20  rule startFSM (!testbenchFSM.done());
21    testbenchFSM.start();
22  endrule
```

FSM in Bluespec: FSM wiederverwenden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   Stmt testFSMStmt = {
2    seq
3      action
4        uut.setupCalculation(aluOp, a, b);
5      endaction
6      action
7        $display("%d %d %d = %d", a, aluOp, b, uut.getResult());
8        activeFSM <= 0;
9      endaction
10   endseq
11  };
12  FSM testFSM <- mkFSMWithPred(testFSMStmt, activeFSM == 1);
```

FSM in Bluespec: Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- StmtFSM ist ein mächtiges Werkzeug um FSM umzusetzen
- Kompakte Umsetzung von Schleifen, sequentieller Ausführung, paralleler Ausführung etc.
- Schachtelung von FSM um komplexere System umzusetzen
- Nicht immer geeignet $\rightarrow$ Übung

Bluespec Tagged Unions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Typ der verschiedene Typen zu einem zusammenfasst
- Variable entspricht dem entsprechenden Typ und einem Tag
- Tag kann durch Pattern Matching gefiltert werden
- Bekannt aus der Vorlesung `Maybe` $\rightarrow$ `Valid t` oder `Invalid`
- Anderes Beispiel: Lese- oder Speicheranfrage an RAM

```
1     interface RAM;
2       method Action    putRequest(RAMRequest);
3       method ActionValue#(Bit#(32)) getReadResult();
4     endinterface;
```

Bluespec Tagged Unions: RAM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1     typedef struct {
2                      Bit#(32) addr;
3                     } ReadRequest  deriving(Bits, Eq);
4     typedef struct {
5                      Bit#(32) addr; Bit#(32) data;
6                     } WriteRequest deriving(Bits, Eq);
7     typedef union tagged {
8                            ReadRequest Read; WriteRequest Write;
9                           } RAMRequest deriving(Bits, Eq);
```

- Deklaration mit `typedef`
- Nutzung von Typklassen mit `deriving`

Bluespec Tagged Unions: Pattern Matching

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   module mkRAM(RAM);
2     Reg#(RAMRequest) currentRequest <- mkRegU;
3     Reg#(Bool)       requestValid <- mkReg(False);
4     RegFile#(Bit#(32), Bit#(32)) ram <- mkRegFileFull();
5     Reg#(Maybe#(Bit#(32))) readData;
6     // Rules
7     ...
8     ////////
9     method Action    putRequest(RAMRequest r) if(!requestValid);
10      currentRequest <= r;
11      requestValid   <= True;
12    endmethod
13
14    method ActionValue#(Bit#(32)) getReadResult()
15                                  if(readData matches tagged Valid .v);
16      readData <= Invalid;
17      return v;
18    endmethod
19  endmodule;
```

Bluespec Tagged Unions: RAM Rules

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   rule handleRequests (requestValid);
2     case (currentRequest) matches
3       tagged Read .r: begin
4         let data <- ram.sub(r.addr);
5         readData <= tagged Valid data;
6       end
7       tagged Write .r: begin
8         ram.upd(r.addr, r.data);
9       end
10    endcase
11    requestValid <= False;
12  endrule;
```

Bluespec Tagged Unions: RAM Read Modify Write

TECHNISCHE UNIVERSITÄT DARMSTADT

- Durch Tagged Unions einfache Erweiterung des Moduls möglich
- Beispiel: Hinzufügen von Read-Modify-Write

```
1     typedef struct {
2                      Bit#(32) addr; Bit#(32) data; Bit#(32) mask;
3                     } RMWRequest deriving(Bits, Eq);
4     typedef union tagged {
5                            ReadRequest Read; WriteRequest Write; RMWRequest RMW;
6                           } RAMRequest deriving(Bits, Eq);
```

Bluespec Tagged Unions: RAM Read Modify Write

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1     rule handleRequests (requestValid);
2       case (currentRequest) matches
3         tagged Read .r: begin
4           let data <- ram.sub(r.addr);
5           readData <= tagged Valid data;
6         end
7         tagged Write .r: begin
8           ram.upd(r.addr, r.data);
9         end
10        tagged RMW .r: begin
11          let data  <- ram.sub(r.addr);
12          let masked = data & (~r.mask);
13          ram.upd(r.addr, r.data | modified);
14        end
15      endcase
16      requestValid <= False;
17    endrule;
```