Einführung in Computer Microsystems

1. Teil: Die Hardware-Beschreibungssprache Bluespec

Andreas Koch
FG Eingebettete Systeme und ihre Anwendungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Embedded Systems & Applications

Material

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorlesungsfolien basieren auf Material von Rishiyur Nikhil
  - Ehemals Professor am MIT
  - Nun CTO von Bluespec Inc.
- Eine englische Fassung auch auf RBG Rechnern installiert

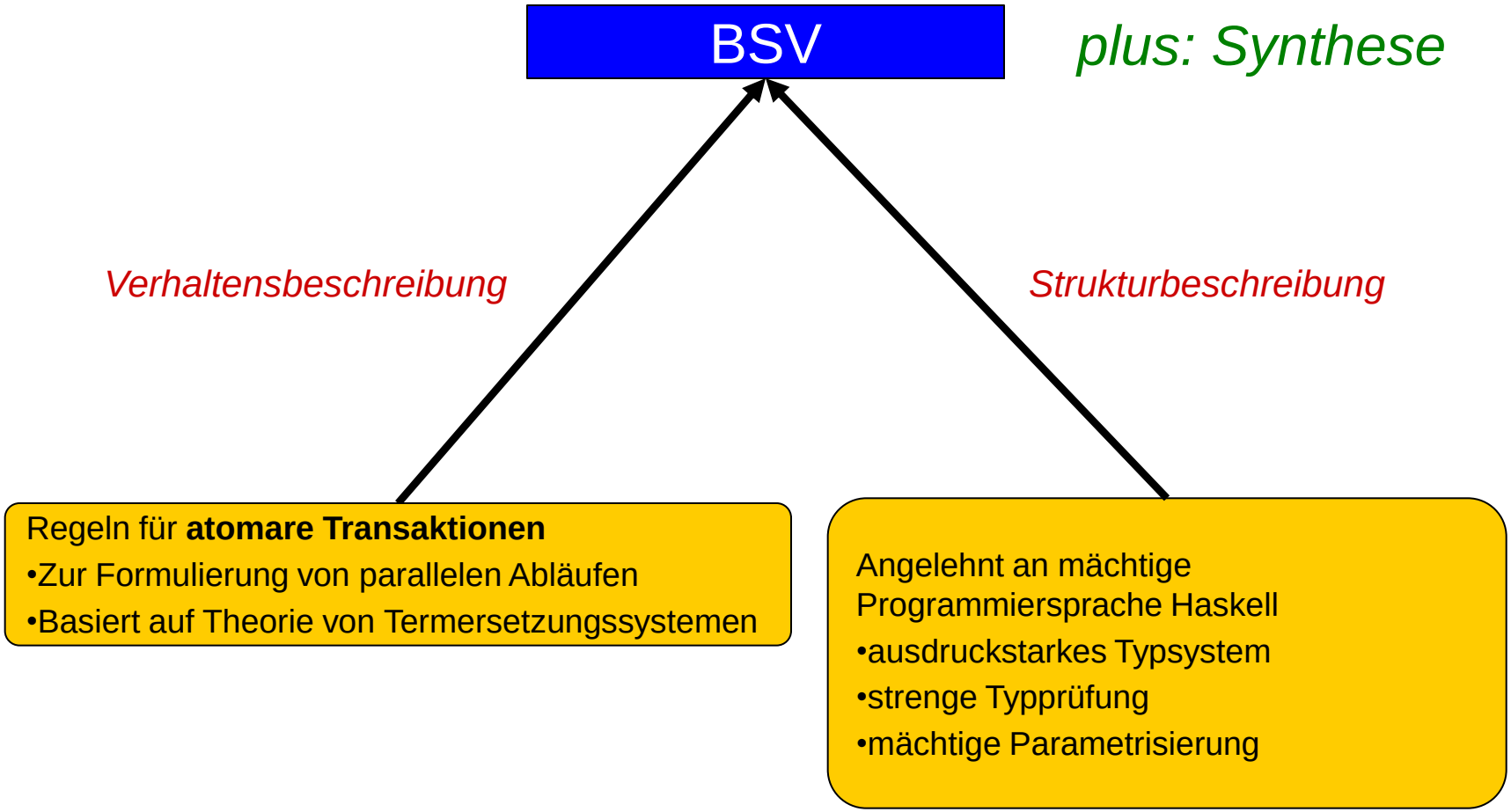
TECHNISCHE UNIVERSITÄT DARMSTADT

EINFÜHRUNG

Grundlegende Elemente von BSV

Bluespec System Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT



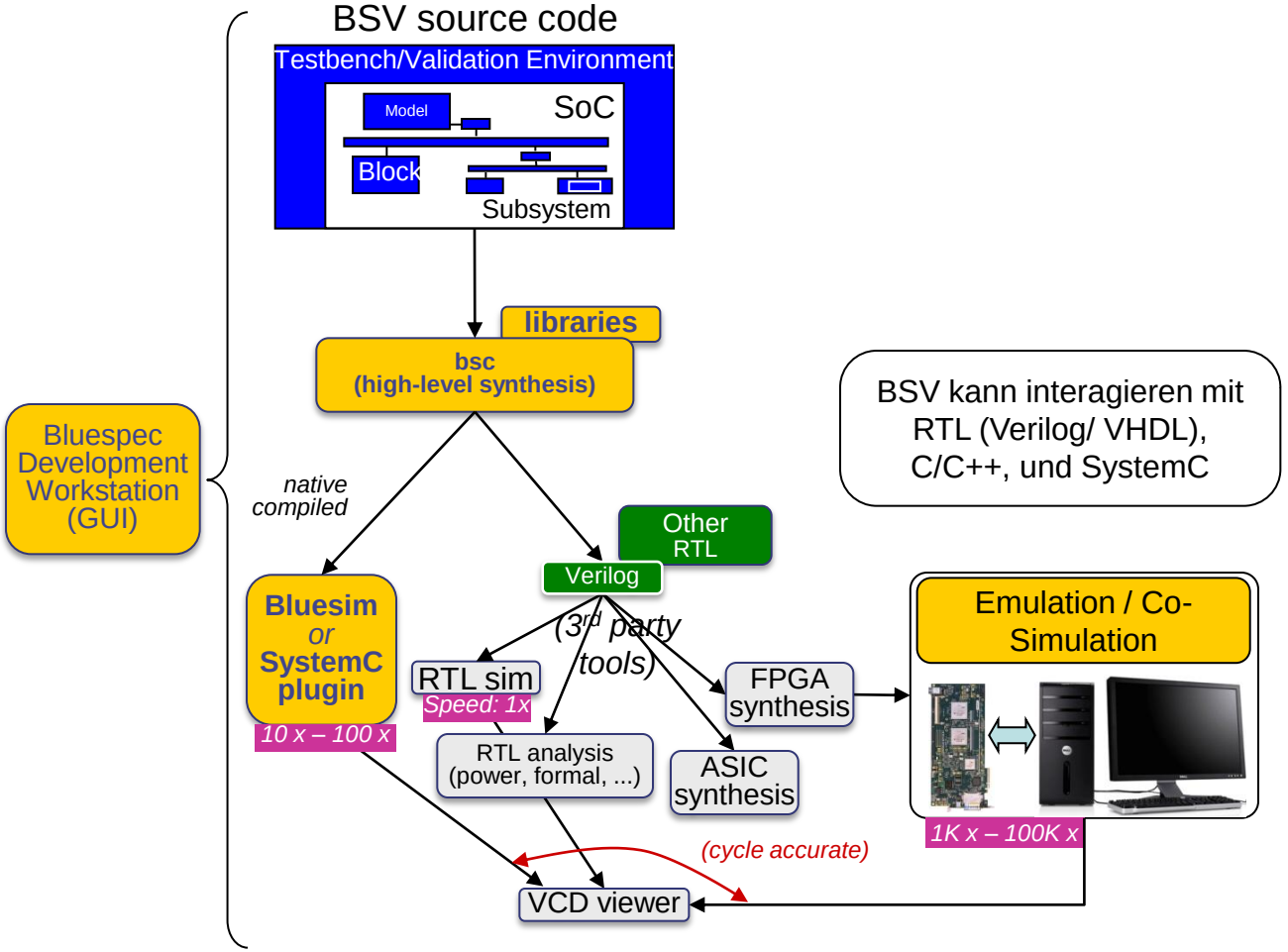
Vergleich von Hardware-Beschreibungssprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

Verhaltens-beschreibung	BSV	**Synthetisierbare** RTL HDL (Verilog, VHDL, SystemVerilog, SystemC)	High-Level **Synthese** (C/C++/Matlab)
Verhalten	Atomare Regeln	Synchrone Schaltungen	Sequentielle Programmierung
Schnittstellen	Atomare Methoden	Im wesentlichen Drähte (kaum Abstraktion)	I.d.R. nur auf oberster Hierarchieebene

Struktur	BSV	Synthetisierbare RTL HDL (Verilog, VHDL, SystemVerilog, SystemC)	High-Level Synthese (C/C++/Matlab)
Direkter Einfluss auf HW-Architektur	Stark	Stark	Schwach
Typprüfungen	Stark	Schwach-Mittel (VHDL)	Mittel
Typen	Mächtig, auch benutzerdefiniert	Bits, schwach benutzerdefiniert	Primitive, schwach benutzerdefiniert
Parametrisierung	Mächtig	Schwach	Schwach

BSV Werkzeugfluß



Weitere Informationen: RBG Pool

$BLUESPEC_HOME = /usr/local/bluespec

TECHNISCHE UNIVERSITÄT DARMSTADT

- Englische Folien: **`$BLUESPEC_HOME/training/BSV/slides/`**
  - Weiteres Material in training/BSV: Beispiele, Übungen, Veröffentlichungen
- Sprachspezifikation: **`$BLUESPEC_HOME/doc/BSV/reference-guide.pdf`**
  - Komplette Beschreibung von BSV, einschließlich Standardbibliotheken etc.
- Benutzerhandbuch: **`$BLUESPEC_HOME/doc/BSV/user-guide.pdf`**
  - Verwendung der Werkzeuge (Kommandozeile und GUI)
- Lehrbuch: **`$BLUESPEC_HOME/doc/BSV/bsv_by_example.pdf`**
  - Bluespec by Example
  - Ca. 60 Beispiele
  - Quellcode zum Ausprobieren: $BLUESPEC_HOME/doc/BSV/bsv_by_example_appendix.tar.gz

Einfache Multiplikation

```
  1001        // x = 4'd9
x 0101        // y = 4'd5
------
  1001        // x << 0      (da y[0] == 1)
 0000         // 0 << 1      (da y[1] == 0)
1001          // x << 2      (da y[2] == 1)
0000          // 0 << 3      (da y[3] == 0)
-------
0101101       // Produkt = 45
```

Schiebe y jeweils ein Bit nach rechts und teste lsb

Multiplizierer in BlueSpec
Externe Schnittstelle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface Mult_ifc;
   method Action                 put_x (int xx);
   method Action                 put_y (int yy);
   method ActionValue   #(int)   get_w ();
endinterface: Mult_ifc
```

Multiplizierer in BlueSpec
Benutzung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkTestbench (Empty);
   Mult_ifc m <- mkMult;

   rule gen_x;
      m.put_x (9);
   endrule

   rule gen_y;
      m.put_y (5);
   endrule

   rule drain;
      let w <- m.get_w ();
      $display ("Product = %d", w);
      $finish ();
   endrule
endmodule: mkTestbench
```

Multiplizierer in BlueSpec
Verhalten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkMult (Mult_ifc);
   Reg #(int)  w     <- mkRegU;
   Reg #(int)  x     <- mkRegU;
   Reg #(int)  y     <- mkRegU;
   Reg #(Bool) got_x <- mkReg (False);
   Reg #(Bool) got_y <- mkReg (False);

   rule compute ((y != 0) && got_x && got_y) ;
      if (lsb(y) == 1) w <= w + x;
      x <= x << 1;
      y <= y >> 1;
   endrule

   method Action put_x (int xx) if (! got_x);
      x <= xx; w <= 0; got_x <= True;
   endmethod

   method Action put_y (int yy) if (! got_y);
      y <= yy; got_y <= True;
   endmethod

   method ActionValue #(int) get_w () if ((y == 0)
                                          && got_x
                                          && got_y);
      got_x <= False; got_y <= False;
      return w;
   endmethod
endmodule: mkMult
```


Embedded Systems & Applications

Beispiel ausführen

Aufteilen auf Dateien und Packages

TECHNISCHE UNIVERSITÄT DARMSTADT

Testbench.bsv

```
package Testbench;

import Mult :: *;  // alles aus Package Mult importieren

module mkTestbench (Empty);
  Mult_ifc m <- mkMult;

  rule gen_x;
    m.put_x (9);
  endrule

  rule gen_y;
    m.put_y (5);
  endrule

  rule drain;
    let w <- m.get_w ();
    $display ("Produkt = %d", w);
    $finish ();
  endrule
endmodule: mkTestbench

endpackage: Testbench
```

Mult.bsv

```
package Mult;

interface Mult_ifc;
  method Action put_x (int xx);
  method Action put_y (int yy);
  method ActionValue #(int) get_w ();   // w = xx * yy
endinterface: Mult_ifc

module mkMult (Mult_ifc);
  Reg #(int)  w     <- mkRegU;
  Reg #(int)  x     <- mkRegU;
  Reg #(int)  y     <- mkRegU;
  Reg #(Bool) got_x <- mkReg (False);
  Reg #(Bool) got_y <- mkReg (False);

  rule compute ((y != 0) && got_x && got_y) ;
    if (lsb(y) == 1) w <= w + x;
    x <= x << 1;
    y <= y >> 1;
  endrule

  method Action put_x (int xx) if (! got_x);
    x <= xx; w <= 0; got_x <= True;
  endmethod

  method Action put_y (int yy) if (! got_y);
    y <= yy; got_y <= True;
  endmethod

  method ActionValue #(int) get_w () if ((y == 0) && got_x && got_y);
    got_x <= False; got_y <= False;
    return w;
  endmethod
endmodule: mkMult

endpackage: Mult
```

Beispiel ausführen

Compilieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- RBG Lizenz initialisieren (nur einmal pro Sitzung)

```
$ export LM_LICENSE_FILE=27001@licence.rbg.informatik.tu-darmstadt.de
```

- Compilieren

```
$ bsc -sim -g mkTestbench -u Testbench.bsv
```

  - Erzeuge Code für BlueSpec-internen Simulator (recht schnell)
  - Global oberstes Modul ist mkTestbench
    - In einer Datei können mehrere Module enthalten sein
  - Übersetze auch alle Untermodule
    - Dateiname = Packagename
    - Testbench importiert alles aus Mult, compiliere Mult.bsv

Beispiel ausführen
Linken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Linken

```
bsc -sim -e mkTestbench -o myFirstModel
```

  - Einsprungpunkt ist das Modul `mkTestbench`
  - Ausgabedatei für Simulationsmodell ist `myFirstModel`
    - Ohne –o wird Dateiname `a.out` verwendet

Beispiel ausführen

Simulation starten

TECHNISCHE UNIVERSITÄT DARMSTADT

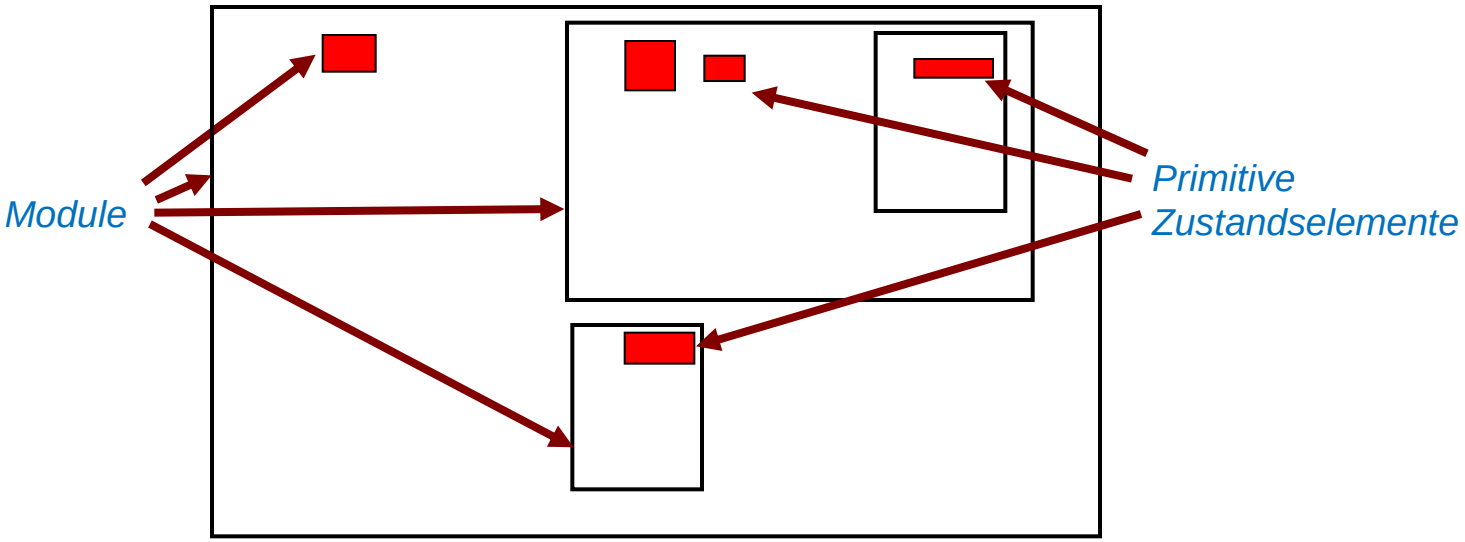
- Simulationsmodell ist **ausführbare Datei**
  - Wird erzeugt beim Linken
- Starten der Simulation: Datei auf Kommandozeile ausführen

```
$ ./myFirstModel
Produkt =          45
$
```

- Diese Schritte sind das übliche Vorgehen
  - Automatisieren der Einzelschritte via **`make`**
  - Oder mittels Bluespec IDE (nicht Bestandteil der Vorlesung)
    - Start mit **`bluespec`**
- Mehr Hinweise zu **`bsc`** und **`bluespec`** im Bluespec User Guide

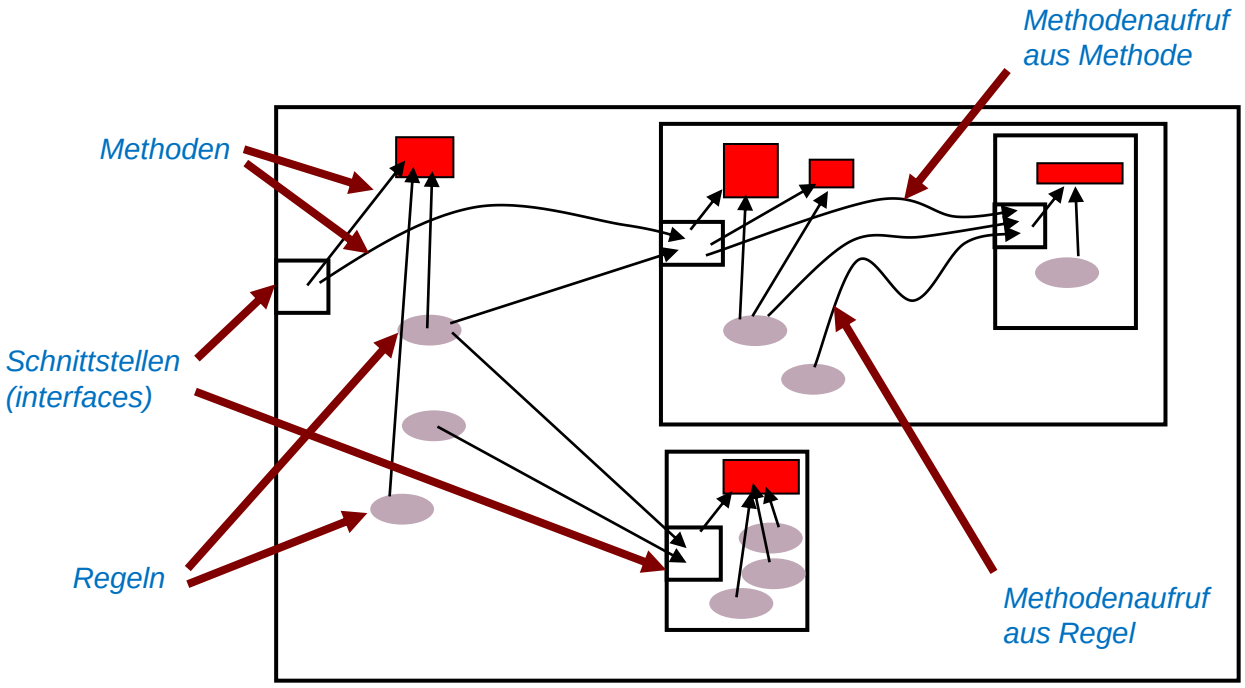
Entwurfshierarchie

- Bluespec besteht aus **Zerlegungshierarchie** von Modulen
  - Wie Verilog, SystemVerilog und SystemC
- Blätter der Hierarchie sind primitive Zustandselemente
  - Register, Warteschlangen (FIFOs), …
- Unterschied zu Verilog: Auch Register sind Module!



Regeln und Schnittstellenmethoden

- Module stellen Schnittstellen durch **Schnittstellenmethoden** bereit
- Module enthalten **Regeln**, die Methoden anderer Module aufrufen
- Methoden können auch Methoden anderer Module aufrufen



Selbst Register sind Module

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schnittstelle zu **Register-Modulen**

```
interface Reg #(type t);
   method Action _write (t v);
   method t      _read  ();
endinterface: Reg
```

`Reg` ist ein **generischer** Typ
Typparameter "t" ist z.B.
`int` oder `Bool` oder `Bit#(16)`

- Zugriff auf Register über **Methodenaufrufe**

```
x._write (x._read () << 1);
```

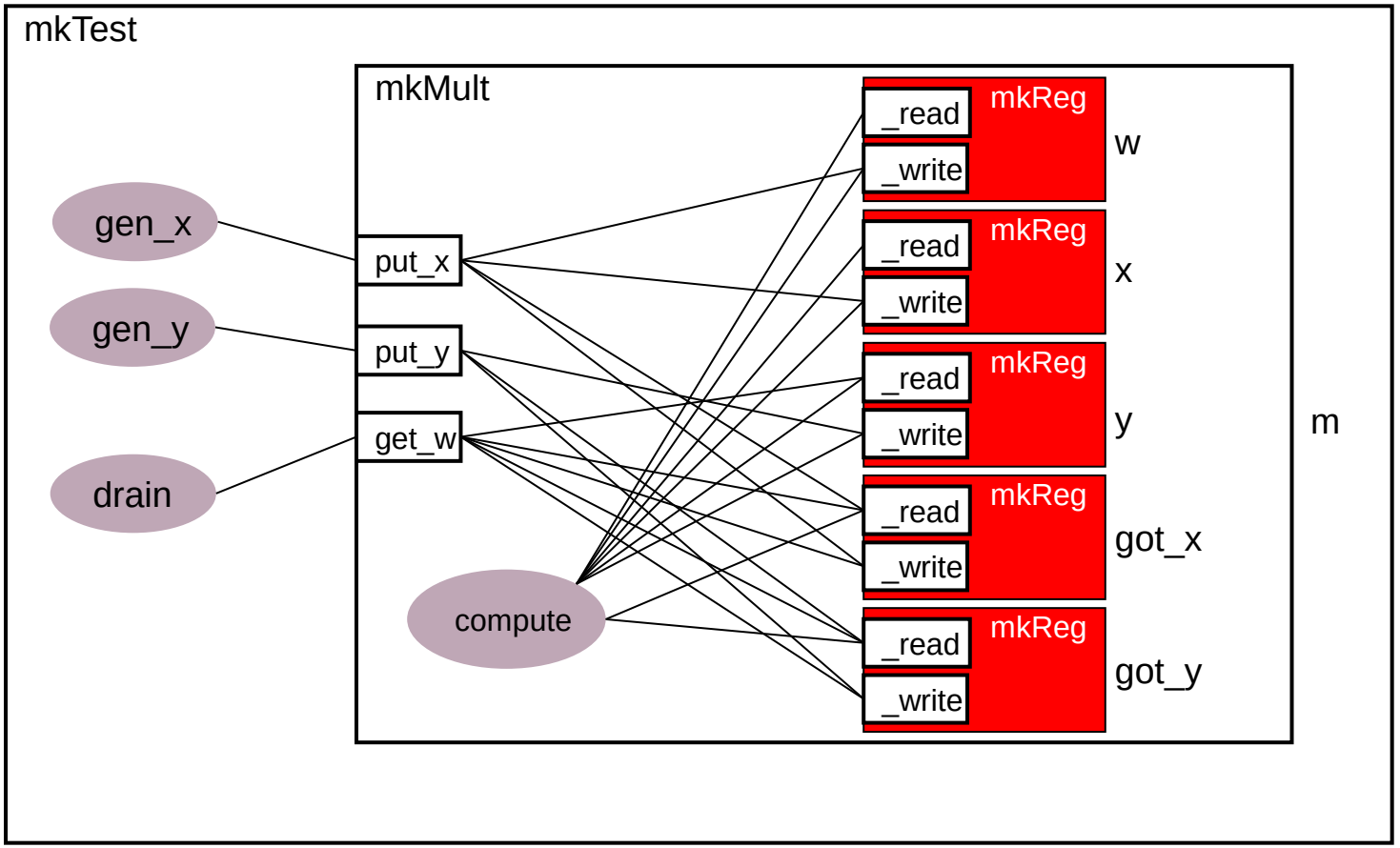
- Kurzform zum **Lesen**

```
x._write (x << 1);
```

- Kurzform zum **Schreiben**

```
x <= x << 1;
```

Beispiel: Multiplizierer



Modulinstanziierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Syntax

```
interface_type   instance_name  <-  module_name ( module_parameters );
```

- "(" *module_parameters* ")" kann bei parameterlosen Modulen entfallen

- **Beispiele**

```
Mult_ifc m <- mkMult;
```

```
Reg#(int)  w     <- mkRegU;
Reg#(Bool) got_x <- mkReg (False);
```

mkRegU ist Modul ohne Parameter;
Intialwert des Registers ist undefiniert
mkReg ist Modul mit Initialwert des Registers als Parameter

Grundlegende Syntaxelemente 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- BSV orientiert sich an (System)Verilog Syntax
- Übliche Regeln für Geltungsbereiche
- Bezeichner
  - Unterscheiden zwischen Groß-/Kleinschreibung
  - **Erstes Zeichen ist relevant**
    - Variablen, Typvariablen und Methoden beginnen mit Kleinbuchstaben
      - `mkMult, x, y, t1`
    - Konstanten und Typen beginnen mit Großbuchstaben
      - `Int, Uint, Bool, True, False`
- Ausnahmen
  - Aus Kompatibilität zu (System)Verilog: `int` und `bit`
  - Sind Kurzformen von `Int#(32)` und `Bit#(1)`

Grundlegende Syntaxelemente 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konvention für Modulnamen
  - Traditioneller Präfix ist “**`mk`**”
  - Gelesen “make”
    - **`mkMultiply`**, **`mkALU`**, …

Methodendeklaration

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zwei wesentliche Arten von Methoden

- Wert-Methoden (*value methods*)
  - Entsprechen mathematischen Funktionen
  - Können Zustand der Schaltung nicht ändern
  - Können lokale Zwischenwerte berechnen (=)
  - Haben einen Rückgabewert an Aufrufer

```
method int foo (int x, int y, int z);
  let sum = x + y;
  return sum + z;
endmethod
```

- Aktions-Methoden (*action methods*)
  - Können Zustand der Schaltung ändern (<=)
  - Haben keinen Rückgabewert

```
Reg#(int) sum <- mkReg(0);
method Action inc(int x);
  sum <= sum + x;
endmethod
```

- Aktionswert-Methoden (*action value meth.*)
  - Können Zustand der Schaltung ändern (<=)
  - Und haben einen Rückgabewert an Aufrufer

```
Reg#(int) sum <- mkReg(0);
method ActionValue(#int) inc2(int x);
  sum <= sum + x;
  return sum*2; // benutzt alten Wert
endmethod
```

Beispiel: Methoden

Deklaration und Aufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface Ifc_Dut;
   method int                read    ();
   method Action             doinc   (int x);
   method ActionValue#(int)  doincrd2(int x);
endinterface

module mkDut(Ifc_Dut);

   Reg#(int) sum <- mkReg(0);

   function Action incsum(int x);
      return
         action
            sum <= sum + x;
         endaction;
   endfunction

   method int read();
      return sum;
   endmethod

   method Action doinc(int x);
      incsum(x);
   endmethod

   method ActionValue#(int) doincrd2(int x);
      let sum2 = 2*sum;
      incsum(x);
      return sum2;
   endmethod

endmodule: mkDut
```

```
module mkTb(Empty);

      Ifc_Dut dut <- mkDut;

      Reg#(int) state <- mkReg(0);

      rule s1 (state == 1);
         $display("s1");
         $display("sum=%0d", dut.read);
         state <= state + 1;
      endrule

      rule s2 (state == 2);
         $display("s2");
         dut.doinc(23);
         $display("sum=%0d", dut.read);
         state <= state + 1;
      endrule

      rule s3 (state == 3);
         $display("s3");
         let newval <- dut.doincrd2(43);
         $display("sum=%0d, newval=%0d", dut.read, newval);
         state <= state + 1;
      endrule

      rule s4 (state == 4);
         $display("s4");
         $display("sum=%0d", dut.read);
         $finish(0);
      endrule

      rule s0 (state == 0);
         state <= 1;
      endrule
   endmodule: mkTb
```


Embedded Systems & Applications

Beispiel: Methoden

Mit Ausgaben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface Ifc_Dut;
   method int                read     ();
   method Action             doinc    (int x);
   method ActionValue#(int)  doincrd2(int x);
endinterface

module mkDut(Ifc_Dut);

   Reg#(int) sum <- mkReg(0);

   function Action incsum(int x);
      return
         action
            sum <= sum + x;
         endaction;
   endfunction

   method int read();
      return sum;
   endmethod

   method Action doinc(int x);
      incsum(x);
   endmethod

   method ActionValue#(int) doincrd2(int x);
      let sum2 = 2*sum;
      incsum(x);
      return sum2;
   endmethod

endmodule: mkDut
```

```
module mkTb(Empty);

      Ifc_Dut dut <- mkDut;

      Reg#(int) state <- mkReg(0);

      rule s1 (state == 1);
         $display("s1");
         $display("sum=%0d", dut.read); // sum=0
         state <= state + 1;
      endrule

      rule s2 (state == 2);
         $display("s2");
         dut.doinc(23);
         $display("sum=%0d", dut.read); // sum=0
         state <= state + 1;
      endrule

      rule s3 (state == 3);
         $display("s3");
         let newval <- dut.doincrd2(43); // sum=23, newval=46
         $display("sum=%0d, newval=%0d", dut.read, newval);
         state <= state + 1;
      endrule

      rule s4 (state == 4);
         $display("s4");
         $display("sum=%0d", dut.read); // sum=66
         $finish(0);
      endrule ...
```

Bedingungen an Methoden und Regeln

TECHNISCHE UNIVERSITÄT DARMSTADT

Bedingungen entscheiden über **Bereitschaft** von Regeln und Methoden zur Ausführung. Default: True

Methoden werden (ggf. indirekt) aus Regeln aufgerufen. Regel ist nur bereit, wenn **alle aufgerufenen** Methoden ebenfalls bereit sind (zusätzlich zur Bedingung an Regel)

Bereitschaft der Regel: **CAN_FIRE**
Konjunktion (AND) von
- Regelbedingung
- Bedingungen an allen aufgerufenen Methoden

```
module mkMult (Mult_ifc);
   Reg #(int)  w     <- mkRegU;
   Reg #(int)  x     <- mkRegU;
   Reg #(int)  y     <- mkRegU;
   Reg #(Bool) got_x <- mkReg (False);
   Reg #(Bool) got_y <- mkReg (False);

   rule compute ((y != 0) && got_x && got_y) ;
      if (lsb(y) == 1) w <= w + x;
      x <= x << 1;
      y <= y >> 1;
   endrule

   method Action put_x (int xx) if (! got_x);
      x <= xx; w <= 0; got_x <= True;
   endmethod

   method Action put_y (int yy) if (! got_y);
      y <= yy; got_y <= True;
   endmethod

   method ActionValue #(int) get_w () if ((y == 0)
                                          && got_x
                                          && got_y);
      got_x <= False; got_y <= False;
      return w;
   endmethod
endmodule: mkMult
```


Embedded Systems & Applications

Bedingungen an Methoden und Regeln

Beispiel: Multiplizierer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkTestbench (Empty);
   Mult_ifc m <- mkMult;

   rule gen_x;
      m.put_x (9);
   endrule

   rule gen_y;
      m.put_y (5);
   endrule

   rule drain;
      let w <- m.get_w ();
      $display ("Product = %d", w);
      $finish ();
   endrule
endmodule: mkTestbench
```

CAN_FIRE_gen_x
`(! got_x)`

CAN_FIRE_gen_y
`(! got_y)`

CAN_FIRE_drain
`((y==0) && got_x && got_y)`

CAN_FIRE_compute
`((y!=0) && got_x && got_y)`

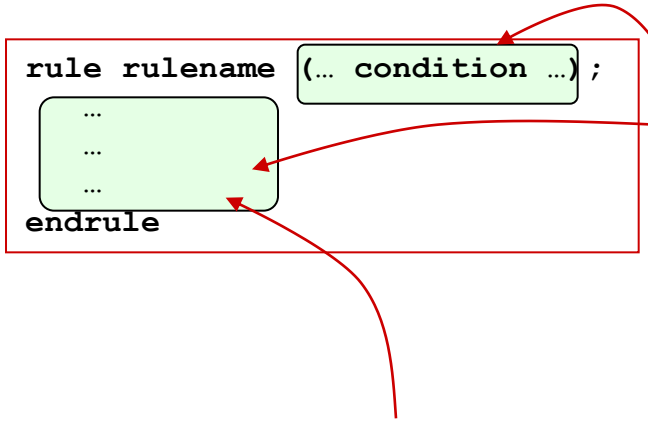
```
module mkMult (Mult_ifc);
   Reg #(int)  w     <- mkRegU;
   Reg #(int)  x     <- mkRegU;
   Reg #(int)  y     <- mkRegU;
   Reg #(Bool) got_x <- mkReg (False);
   Reg #(Bool) got_y <- mkReg (False);

   rule compute ((y != 0) && got_x && got_y) ;
      if (lsb(y) == 1) w <= w + x;
      x <= x << 1;
      y <= y >> 1;
   endrule

   method Action put_x (int xx) if (! got_x);
      x <= xx; w <= 0; got_x <= True;
   endmethod

   method Action put_y (int yy) if (! got_y);
      y <= yy; got_y <= True;
   endmethod

   method ActionValue #(int) get_w () if ((y == 0)
                                          && got_x
                                          && got_y);
      got_x <= False; got_y <= False;
      return w;
   endmethod
endmodule: mkMult
```


Embedded Systems & Applications

Ausführung von Regeln

Vereinfachte Erklärung

Jede Regel hat einen Namen und zwei für die Semantik relevante Teile



CAN_FIRE Bedingung, Konjunktion von

- Expliziter Regelbedingung
- Methodenbedingungen von in Regelbedingung aufgerufener Methoden
- Methodenbedingungen vom im Regelkörper aufgerufener Methoden

Alle im Körper der Regel (*rule body*) ausgeführten **Aktionen**

Vereinfachte Ausführungssemantik

Wann immer das **CAN_FIRE** einer Regel wahr ist
führe die **Aktionen** im Regelkörper aus

Ausführungssequenz von Regeln

Beispiel: Multiplizierer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkTestbench (Empty);
   Mult_ifc m <- mkMult;

   rule gen_x;
      m.put_x (9);
   endrule

   rule gen_y;
      m.put_y (5);
   endrule

   rule drain;
      let w <- m.get_w ();
      $display ("Product = %d", w);
      $finish ();
   endrule
endmodule: mkTestbench
```

gen_x gen_y oder **gen_y gen_x**

oder **gen_x gen_y**

compute
compute
…
compute
drain

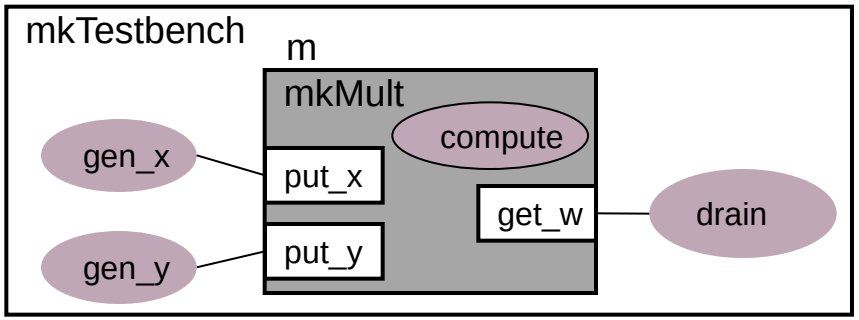
```
module mkMult (Mult_ifc);
   Reg #(int)  w     <- mkRegU;
   Reg #(int)  x     <- mkRegU;
   Reg #(int)  y     <- mkRegU;
   Reg #(Bool) got_x <- mkReg (False);
   Reg #(Bool) got_y <- mkReg (False);

   rule compute ((y != 0) && got_x && got_y) ;
      if (lsb(y) == 1) w <= w + x;
      x <= x << 1;
      y <= y >> 1;
   endrule

   method Action put_x (int xx) if (! got_x);
      x <= xx; w <= 0; got_x <= True;
   endmethod

   method Action put_y (int yy) if (! got_y);
      y <= yy; got_y <= True;
   endmethod

   method ActionValue #(int) get_w () if ((y == 0)
                                          && got_x
                                          && got_y);
      got_x <= False; got_y <= False;
      return w;
   endmethod
endmodule: mkMult
```


Embedded Systems & Applications

Kaskade aus zwei Multiplizierern

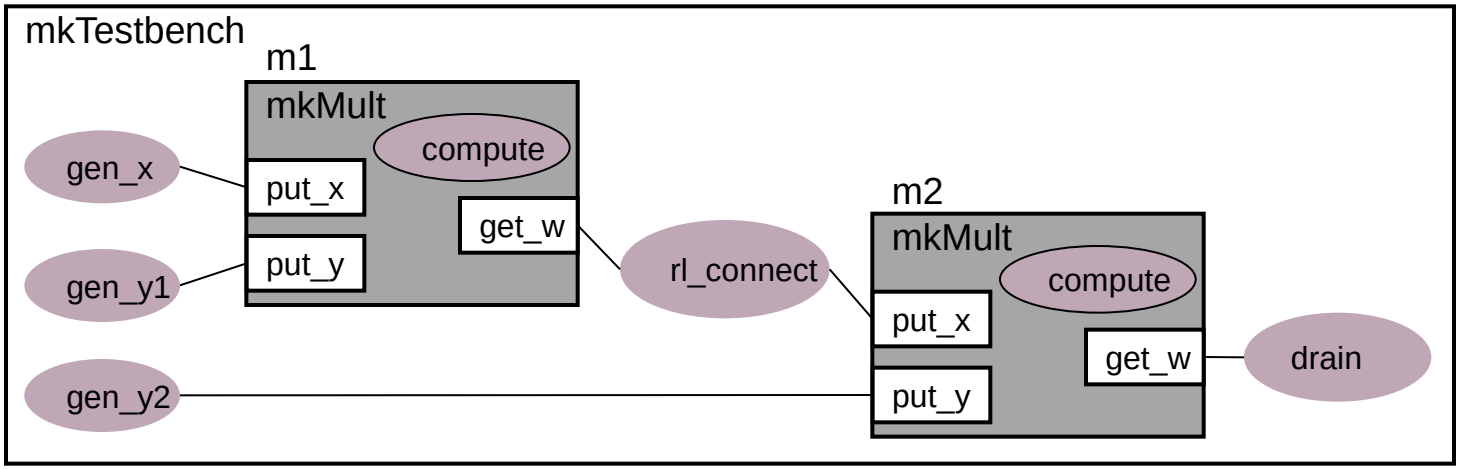
TECHNISCHE UNIVERSITÄT DARMSTADT

Ursprüngliches Beispiel



`mkMult` *bleibt unverändert, nur* **`mkTestbench`** *wird angepasst*

Erweitertes Beispiel

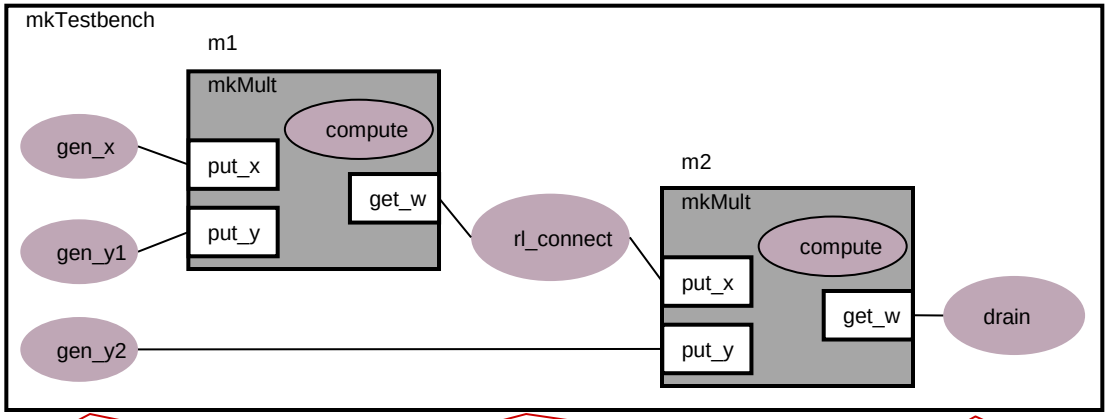


Embedded Systems & Applications

mkTestBench

BSV Code für Kaskade

```
module mkTestbench (Empty);
   Mult_ifc m1 <- mkMult;
   Mult_ifc m2 <- mkMult;

   Reg #(int) rg_x  <- mkReg (1);
   Reg #(int) rg_y1 <- mkReg (1);
   Reg #(int) rg_y2 <- mkReg (1);

   rule gen_x;
      m1.put_x (rg_x);
      rg_x <= rg_x + 1;    // Wertfolge
   endrule

   rule gen_y1;
      m1.put_y (rg_y1);
      rg_y1 <= rg_y1 + 2;  // Wertfolge
   endrule

   rule gen_y2;
      m2.put_y (rg_y2);
      rg_y2 <= rg_y2 + 3;  // Wertfolge
   endrule
```




```
   rule rl_connect;
      let x2 <- m1.get_w ();   // deklariert Zwischenwert
      m2.put_x (x2);           // und führt Aktionswert-Methodenaufruf aus
   endrule

   Reg #(int) rg_j <- mkReg (0);

   rule drain;
      let w2 <- m2.get_w ();
      $display ("Product [%0d]: %0d x %0d x %0d = %0d",
                rg_j, rg_j+1, rg_j*2+1, rg_j*3+1, w2);
      if (rg_j == 10) $finish ();
      rg_j <= rg_j + 1;
   endrule
endmodule: mkTestbench
```

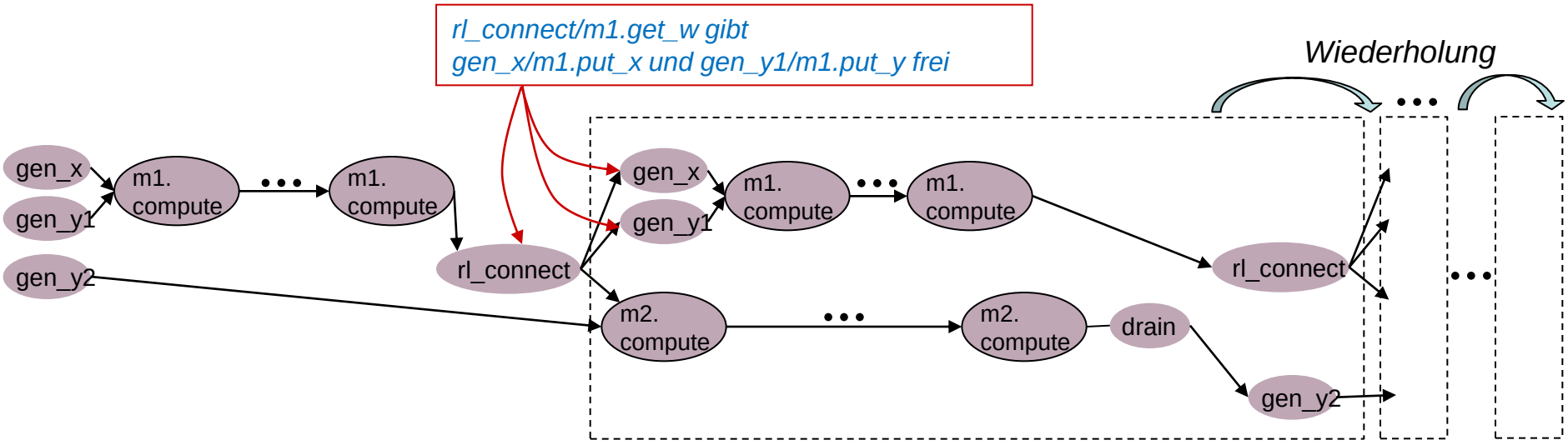
Ausführen von `mkTestbench`

TECHNISCHE UNIVERSITÄT DARMSTADT

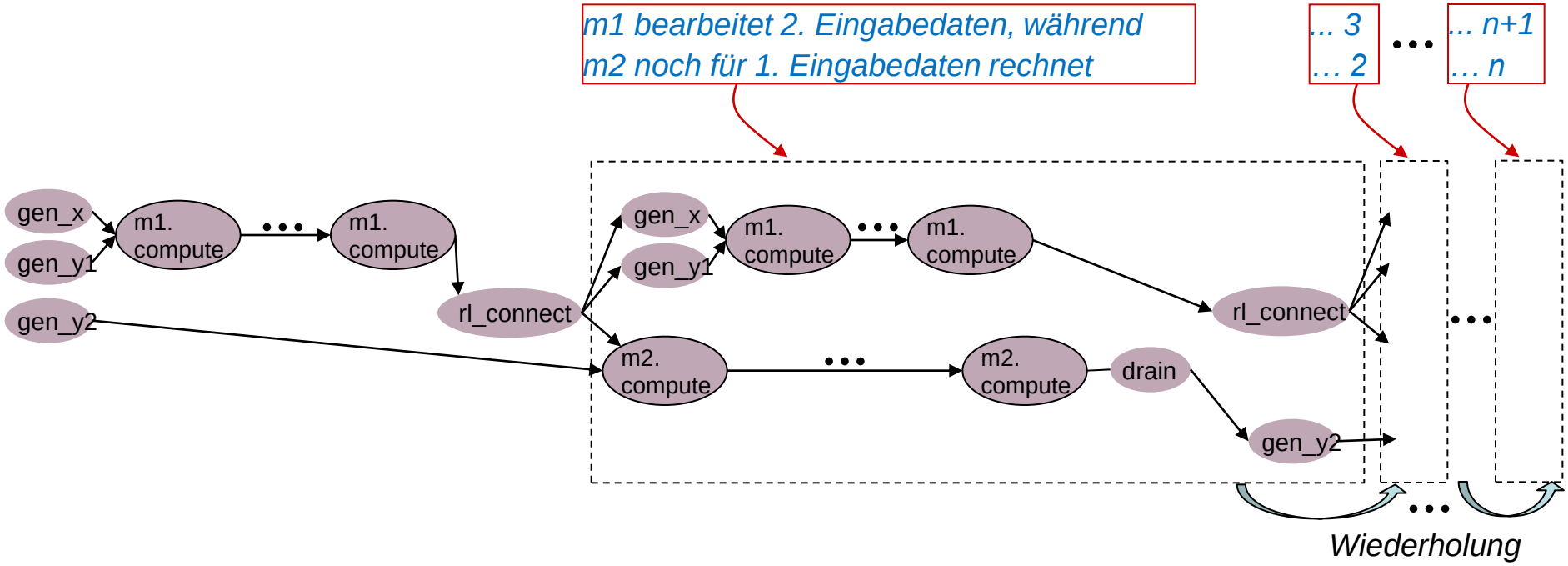
```
$ ./a.out
Product [0]: 1 x 1 x 1 = 1
Product [1]: 2 x 3 x 4 = 24
Product [2]: 3 x 5 x 7 = 105
Product [3]: 4 x 7 x 10 = 280
Product [4]: 5 x 9 x 13 = 585
Product [5]: 6 x 11 x 16 = 1056
Product [6]: 7 x 13 x 19 = 1729
Product [7]: 8 x 15 x 22 = 2640
Product [8]: 9 x 17 x 25 = 3825
Product [9]: 10 x 19 x 28 = 5320
Product [10]: 11 x 21 x 31 = 7161
$
```

Mögliche Ausführungsreihenfolgen

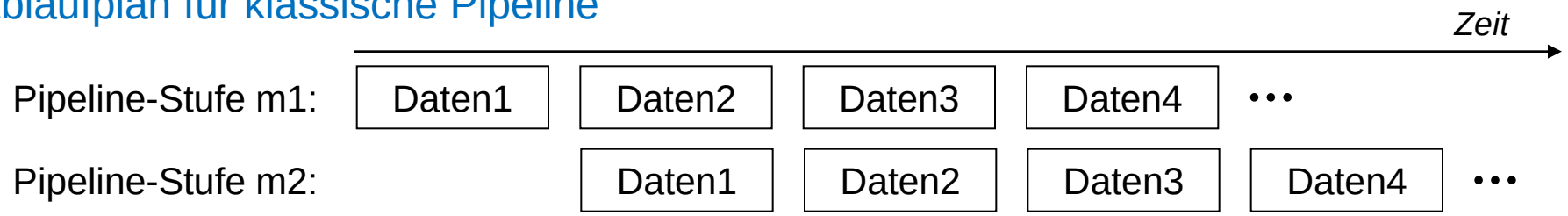
- **Präzedenzrelation** zwischen Regeln/Methoden
  - Quellknoten gibt (Teil)bedingung an Zielknoten frei
- Impliziert eine **Halbordnung**
  - Eine Regel can erst ausgeführt werden (feuern),
    - … wenn alle (auch transitiven) Bereitschaften gesetzt sind
  - Zwei zueinander (auch transitiv) ungeordnete Regeln
    - … können in beliebiger Reihenfolge feuern



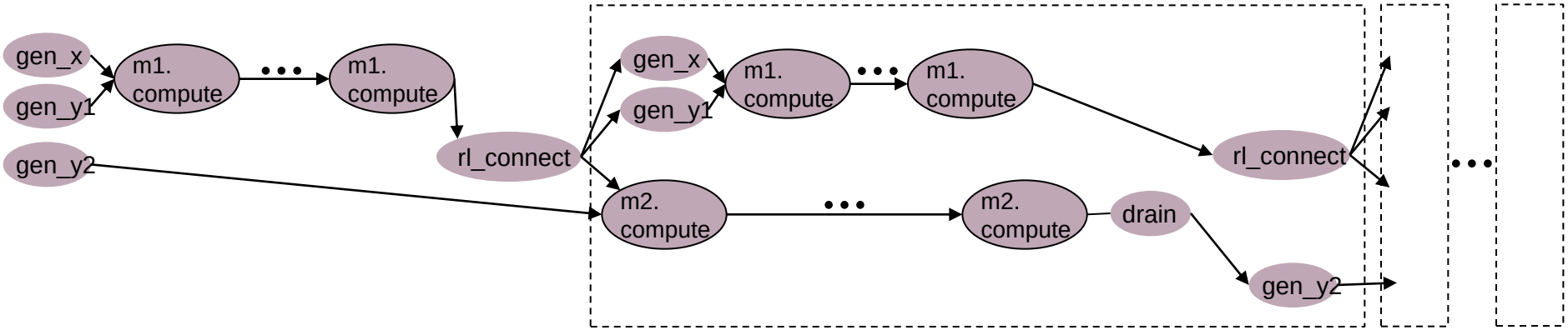
Kaskade erlaubt Pipelineausführung



Ablaufplan für klassische Pipeline



Aber keine “normale” Pipeline



- Pipeline ist **dynamisch**
  - Latenz ist nun datenabhängig variabel (je nach Anzahl 1-Bits im Multiplikator)
  - Gegenbeispiel MIPS: immer 5 Takte für Fetch-Decode-Execute-Mem-Writeback
    - War statische Pipeline
- Pipeline ist **elastisch**
  - Daten bewegen sich mit unterschiedlichem Fortschritt durch Pipeline
    - Hier ohne Balancing Register: Funktioniert, hat aber reduzierten Durchsatz
  - Gegenbeispiel MIPS: alle Daten im Gleichschritt
    - War inelastische (oder starre) Pipeline

Weiteres Vorgehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiele ausprobieren
  - Kommandozeile reicht
- Zur Vertiefung im Buch “Bluespec by Example” lesen
  - Kapitel 2,3,4

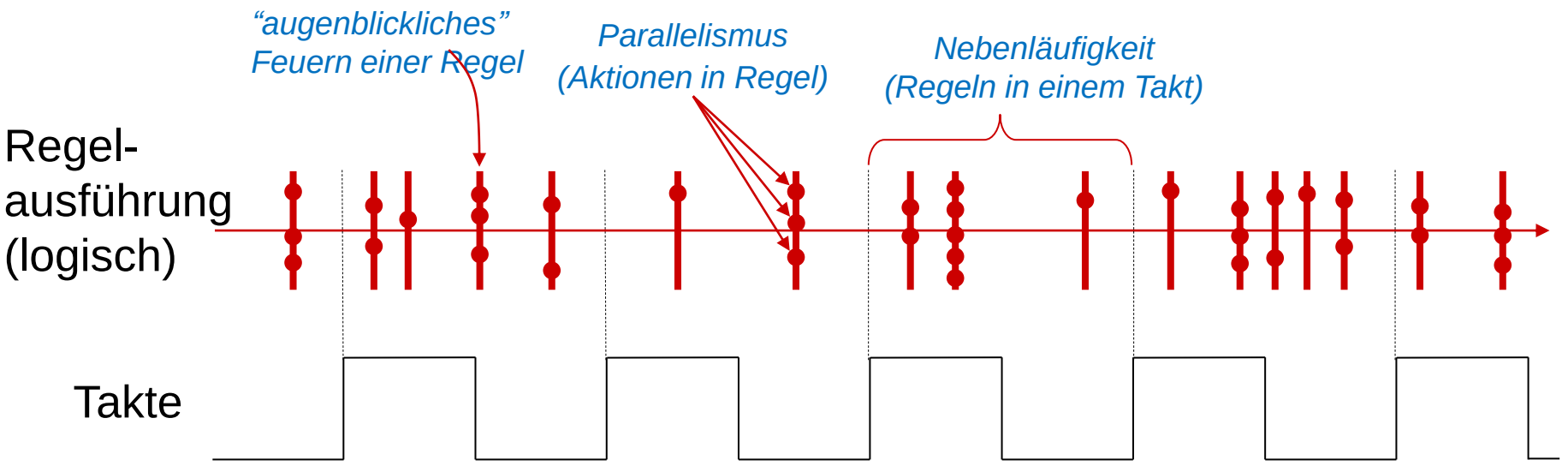
AUSFÜHRUNGSSEMANTIK

Zweistufige Erklärung

Nun genauer als in Einführung

TECHNISCHE UNIVERSITÄT DARMSTADT

1) Semantik einzelner Regeln
   - **Parallele** Ausführung von Aktionen innerhalb einer Regel
2) Zusammenspiel mehrerer Regeln
   - **Nebenläufige** Ausführung mehrerer Regeln in einem Taktzyklus



Embedded Systems & Applications

Parallelismus 1

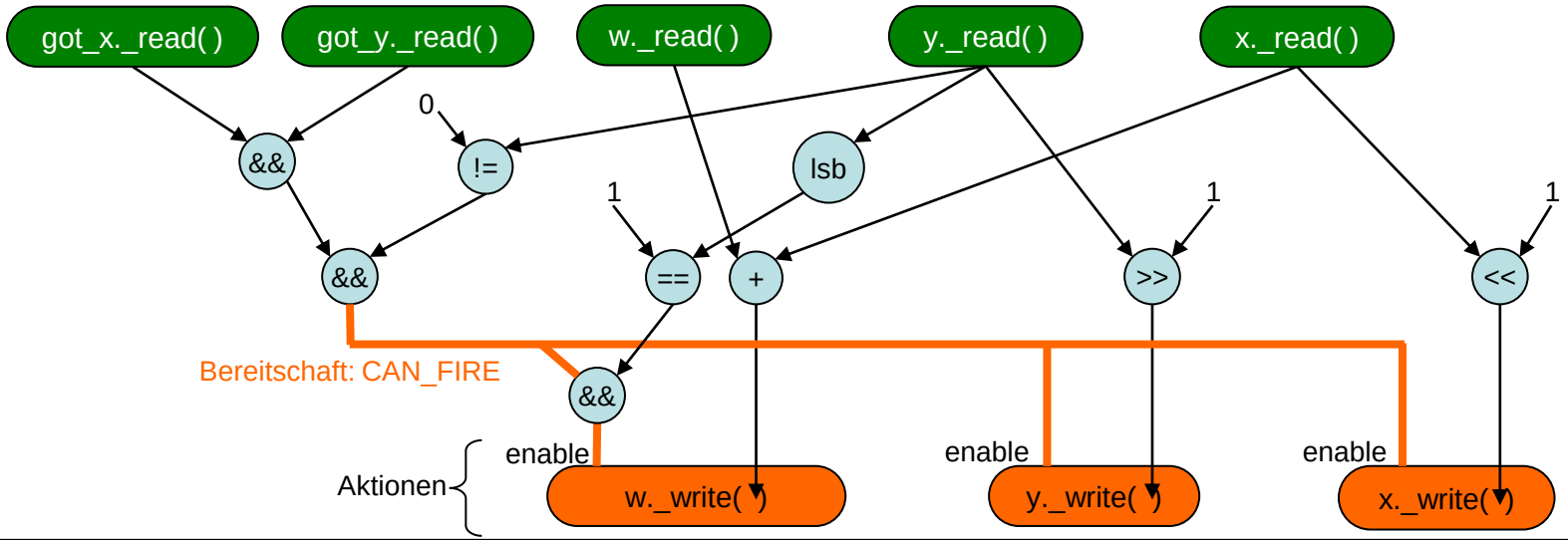
TECHNISCHE UNIVERSITÄT DARMSTADT

- Ausführungssematik für Aktionen innerhalb einer Regel
  - **Gleichzeitig**
  - **Augenblicklich** (Ablauf in “Nullzeit”)

Parallelismus 2

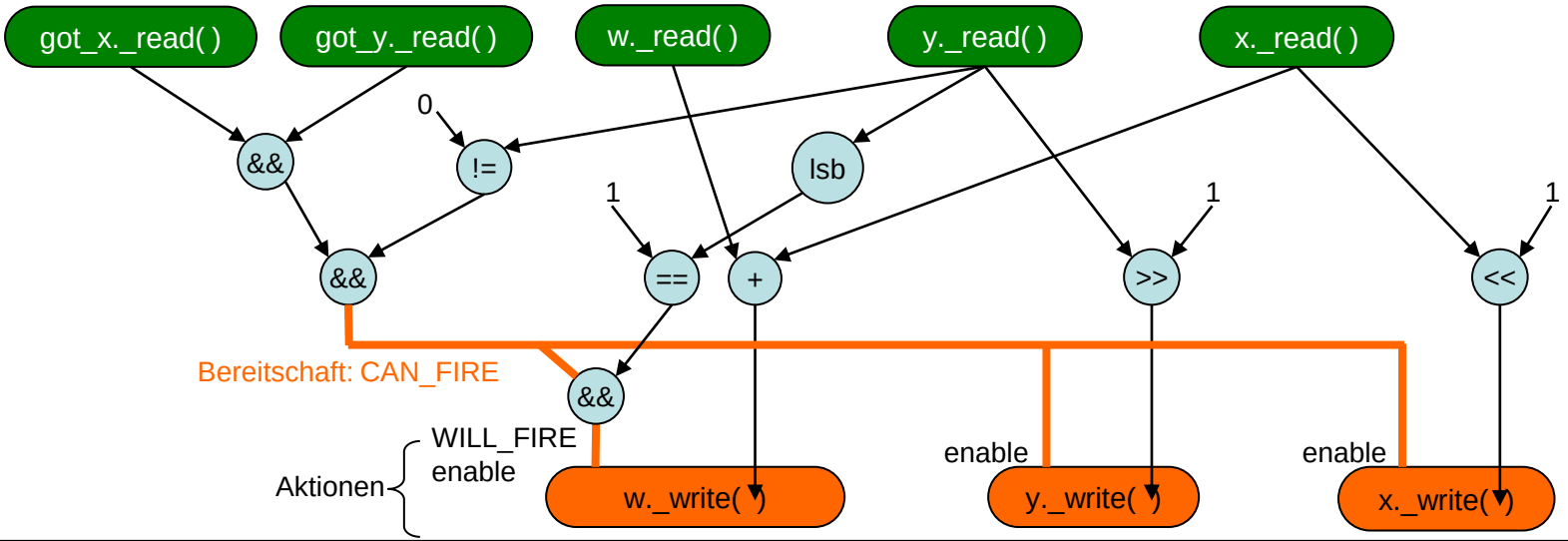
- Betrachte jede Regel als **Datenfluß** von
  - … Konstanten
  - Ergebnissen von reinen Funktionen (in BlueSpec: Wertmethoden)
- Hin zu
  - Parametern für Aktionsmethoden
    - (Indirekte) Veränderung von Zustandselementen

```
rule compute ((y != 0) && got_x && got_y) ;
   if (lsb(y) == 1) w <= w + x;
   x <= x << 1;
   y <= y >> 1;
endrule
```



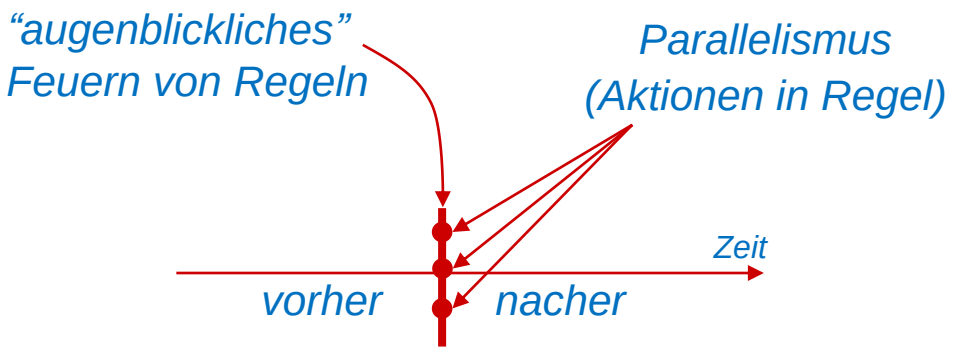
Parallelismus 3

- **Ausführung** von Aktionsmethoden (kurz: Aktionen), kurz: Feuern
  - Ausgelöst durch Freigabesignal **Enable** der Aktion
  - Bedingung für das Feuern berechnet als **WILL_FIRE**
  - **Wichtig**: Unterschied zwischen CAN_FIRE (Bereitschaft) und WILL_FIRE
- Alle Aktionen einer Regel feuern **gleichzeitig**



Parallelismus 4

- Reihenfolge von Aktionen im BSV Quelltext ist **irrelevant**
  - Alle Aktionen feuern immer **gleichzeitig**
- Selbst bereite Aktionen (CAN_FIRE=1) können durch Bedingungen **innerhalb der Regel** am Feuern gehindert werden (WILL_FIRE=0)
- Gelesene Werte spiegeln Zustand **vor** Feuern wieder
- Schreiben von neuen Werten erst **nach** Feuern aller Aktionen
- Effekt: **Atomares** Aktualisieren des Zustandes

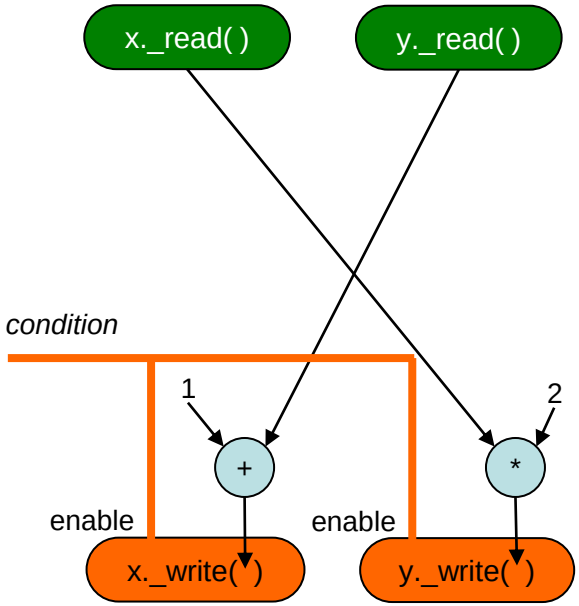


Gleichzeitige Ausführung von Aktionen
Beispiel

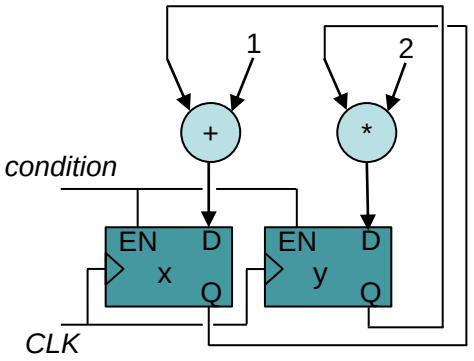
```
rule rule2a (condition);
   x <= y + 1;
   y <= x * 2;
endrule : rule2a
```

Textuelle Reihenfolge der Aktionen irrelevant

```
rule rule2b (condition);
   y <= x * 2;
   x <= y + 1;
endrule : rule2b
```



Beschriebene Hardware



- Damit z.B. auch Austausch von Werten ohne Zwischenvariable
  - Semantik ähnlich nicht-blockender Zuweisung in Verilog (<=)

Parallele Aktionen müssen tatsächlich gleichzeitig ausführbar sein

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rule rule3 (…);
  valuea <= expr1;
  valuea <= expr2;
  …
endrule : rule3
```

Register kann nicht gleichzeitig mit zwei Werten geschrieben werden

```
rule rule4 (…);
  fifo.enq (23);
  fifo.enq (34);
  …
endrule : rule4
```

FIFO kann nicht gleichzeitig mit zwei Werten beschickt werden

```
rule rule5 (…);
  let x = regFile.read (5);
  let y = regFile.read (7);
  …
endrule : rule5
```

Registerfeld kann nicht gleichzeitig zwei Werte aus dem gleichen Port lesen

- Probleme werden durch **bsc Compiler** entdeckt
  - “Cannot compose certain actions in parallel”
- Spezielle Register/FIFOs/Registerfelder **mit mehr Ports** sind möglich
  - Diese können dann auch innerhalb einer Regel mehrere Zugriffe erlauben

Nebenläufigkeit 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- *Concurrency*
- Ausführungssemantik für mehrere Regeln innerhalb desselben Taktes
- Unter Beibehalten der gleichen logischen Ausführungsreihenfolge
  - Wird noch genauer erklärt …

Nebenläufigkeit 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Stelle **Ablaufplan** (*schedule*) auf
  - Zeitlich sequentielle Ausführungsreihenfolge von Regeln im Programm
    r1 r2 r3 … rN
  - Erster Ansatz: Wähle beliebige Reihenfolge

- Daraus nun korrekte **nebenläufige Ausführung** herleiten
  - Für jeden Taktzyklus

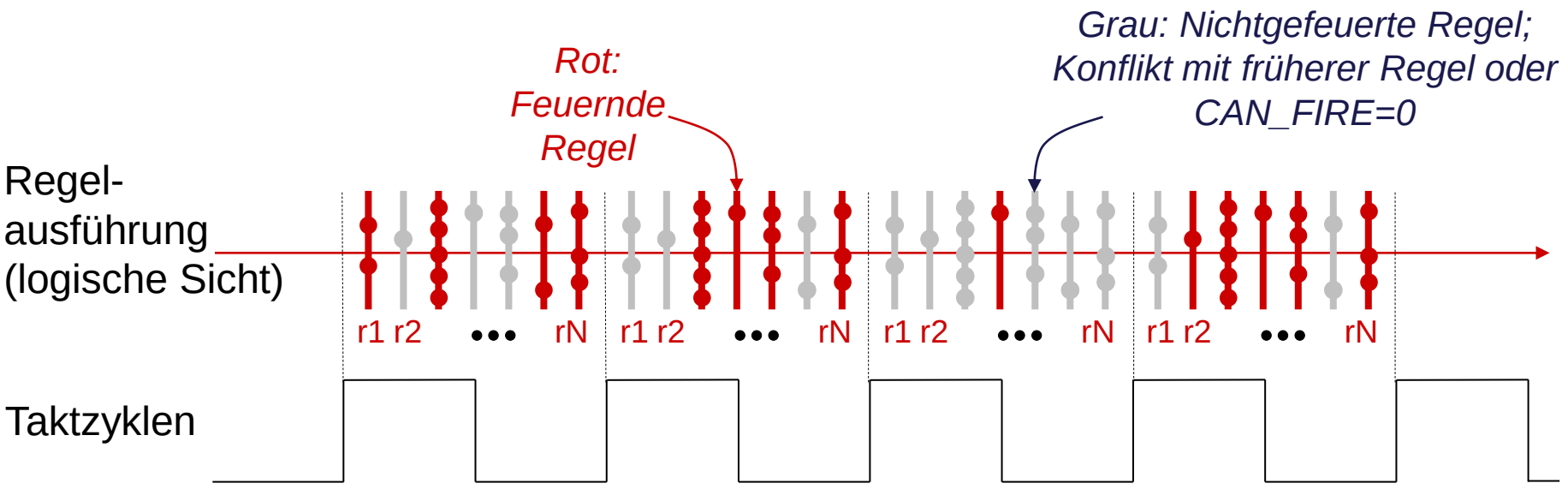
Untersuche jede Regel *rJ* von *r1* bis *rN*
Falls *rJ* **konfliktfrei** zu allen vorhergehenden Regeln *r1 … rJ-1* ist
Führe *rJ* gemäß der Ausführungssemantik für Einzelregeln aus

- Diskussion von **Konflikten**: kommt noch …

Nebenläufigkeit 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Untersuche jede Regel *rJ* von *r1* bis *rN*
Falls *rJ* **konfliktfrei** zu allen vorhergehenden Regeln *r1 … rJ-1* ist
Führe *rJ* gemäß der Ausführungssemantik für Einzelregeln aus



Embedded Systems & Applications

Konflikte zwischen Methodenaufrufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ursache: **Einschränkungen** bei Ausführungsreihenfolge
  - Zwischen Paaren von Methoden
  - Untersucht für alle Methoden eines Modules
- Beeinflussen Ausführungsreihenfolge der aufrufenden **Regeln**

Einschränkung	Bedeutung: Regeln mit Aufrufen von mA und mB können …
mA *konfliktfrei* mB	… nebenläufig feuern (beliebige Reihenfolge), abk.: mA CF mB
mA < mB	… nebenläufig feuern, falls die mA aufrufende Regel logisch eher ausgeführt, wird als die mB aufrufende
mB < mA	… nebenläufig feuern, falls die mB aufrufende Regel logisch eher ausgeführt, wird als die mA aufrufende
mA *konflikt* mB	… nicht nebenläufig feuern (in keiner Reihenfolge), abk.: mA C mB

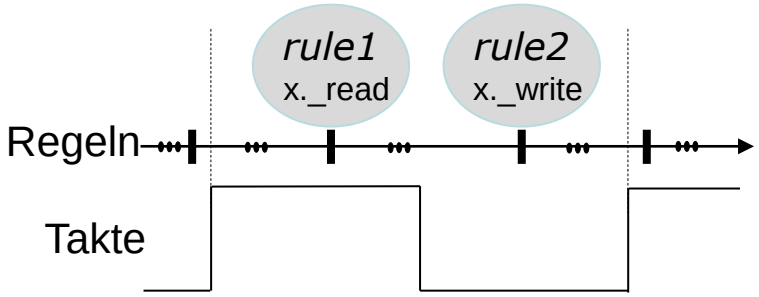
- **Vordefinierte** Einschränkungen für Methoden primitiver BSV-Module
- Daraus **Herleitung** der Einschränkungen der benutzenden Module

Beispiel: Methoden von Registern

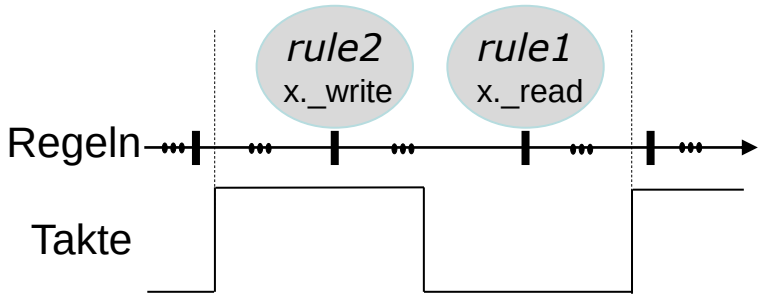
Einschränkungen der Ausführungsreihenfolge

- Primitive **`mkReg`** definiert: **`_read < _write`**
  - Idee: In einem Taktzyklus werden erst alle Werte gelesen, erst danach geschrieben
  - Ergebnis: Neue Werte werden erst nach der steigenden Taktflanke sichtbar

Kein Konflikt: *Ablaufplan rule1,rule2 respektiert Einschränkung; rule1 und rule2 können nebenläufig feuern*



Konflikt: *Ablaufplan rule2,rule1 verletzt Einschränkung; rule2 und rule1 können **nicht** nebenläufig feuern*

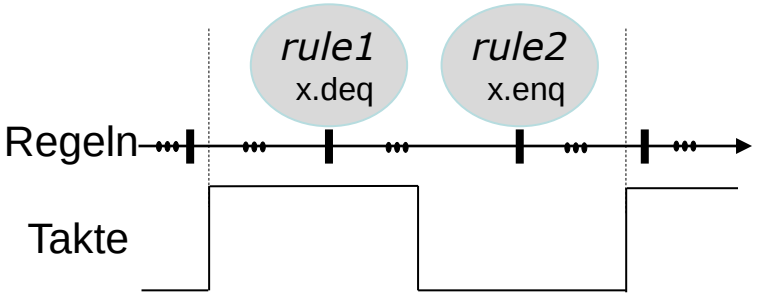


Beispiel: Methoden von FIFOs 1

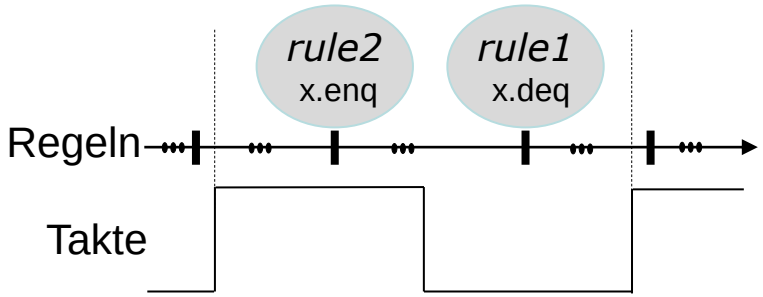
Einschränkungen der Ausführungsreihenfolge

- Primitive **`mkFIFO`** definiert (u.a.): { **`deq`**, **`first`** } *konfliktfrei* **`enq`**
  - Idee: In einem Taktzyklus können **gleichzeitig**
    - … neue Elemente in die Warteschlange (FIFO) eingetragen werden (**`enq`**)
    - … und gleichzeitig das erste Ausgabeelement gelesen/entfernt werden (**`first`**/**`deq`**)

***Kein Konflikt**: Ablaufplan respektiert Einschränkung*



***Kein Konflikt**: Ablaufplan respektiert Einschränkung*

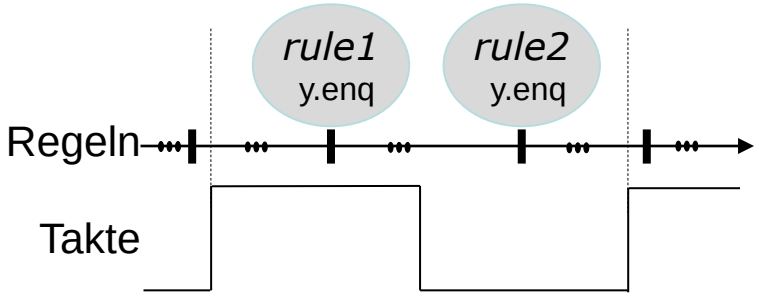


Beispiel: Methoden von FIFOs 2

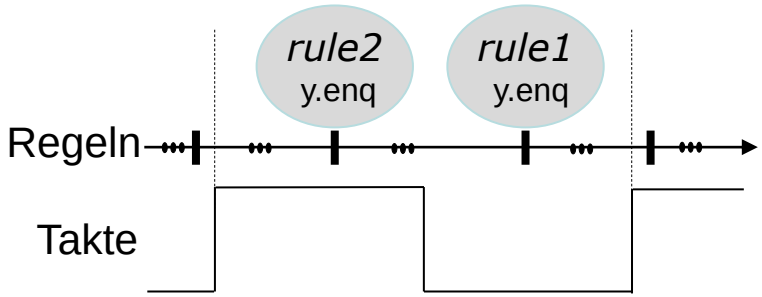
Einschränkungen der Ausführungsreihenfolge

- Primitive **`mkFIFO`** definiert (u.a.): **`enq`** *konflikt* **`enq`**
  - Idee: In einem Taktzyklus können **nicht gleichzeitig**
    - … **zwei** neue Elemente in die Warteschlange (FIFO) eingetragen werden (**`enq`**)
    - Gängige Einschränkung von Hardware: Nur einen Schreib-Port in FIFO

Konflikt: *Ablaufplan verletzt Einschränkung*



Konflikt: *Ablaufplan verletzt Einschränkung*



Verfeinerung: `enq`/`deq` bei FIFOs

First-In First-Out

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Warteschlangen** (FIFOs) sind wesentliches Konstruktionsmittel in BSV

- Einfache FIFOs: **`mkFIFO`**
  - Wenn FIFO **leer**, kein **`deq`** möglich
    - Selbst, wenn gleichzeitig ein enq stattfindet
  - Wenn FIFO **voll**, kein **`enq`** möglich
    - Selbst, wenn gleichzeitig ein deq stattfindet

`enq` und `deq` nur gleichzeitig, wenn FIFO weder voll noch leer ist

- Pipeline FIFOs: **`mkPipelineFIFO`**
  - Auch wenn FIFO **voll**, **`enq`** möglich
    - … wenn gleichzeitig ein deq stattfindet: **`first`** liefert noch alten Wert (vor **`enq`**!)

- Bypass FIFOs: **`mkBypassFIFO`**
  - Auch wenn FIFO **leer**, deq möglich
    - … wenn gleichzeitig ein enq stattfindet: **`first`** liefert schon neuen Wert (nach **`enq`**!)

Beispiel: Pipeline FIFOs

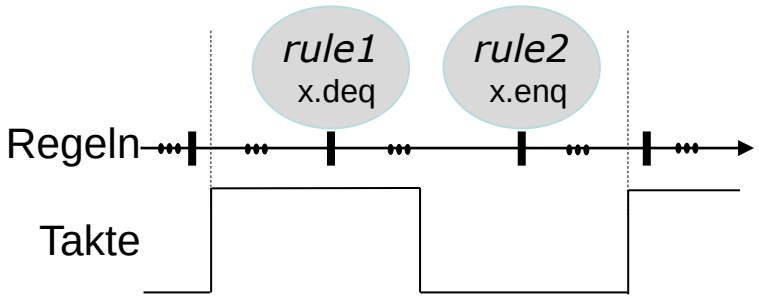
Einschränkungen der Ausführungsreihenfolge

TECHNISCHE UNIVERSITÄT DARMSTADT

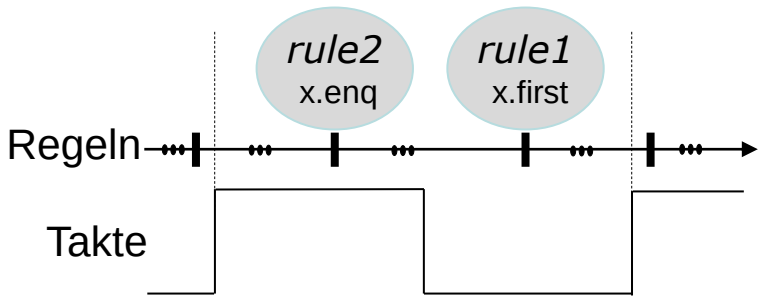
- Pipeline FIFOs: **`mkPipelineFIFO`**
  - Auch wenn FIFO **voll**, **`enq`** möglich
    - … wenn gleichzeitig ein deq stattfindet: **`first`** liefert noch alten Wert (vor **`enq`**!)

$$\{\,\texttt{deq}\,,\,\texttt{first}\,\} < \texttt{enq}$$

***Kein Konflikt**: Ablaufplan respektiert Einschränkung*



***Konflikt:** Ablaufplan verletzt Einschränkung*



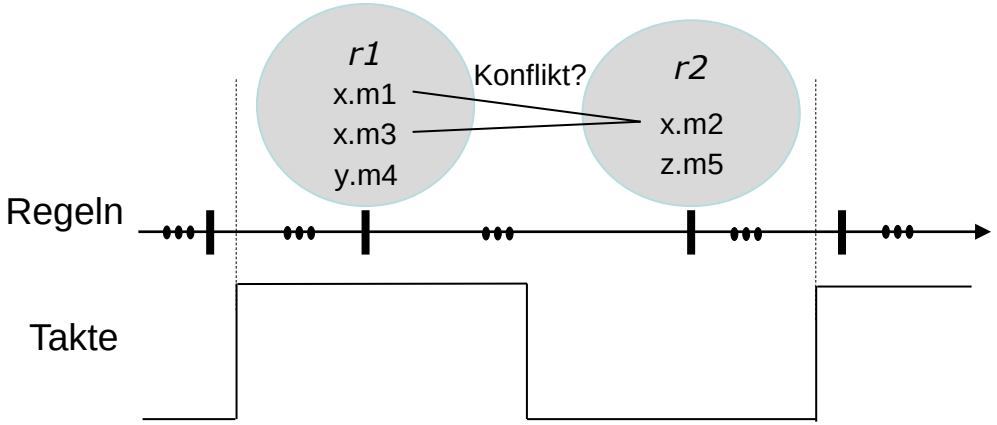
Ausführungsreihenfolge

Von einzelnen Methoden hin zu Regeln

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bisher: Fokus auf jeweils ein **Paar** von Methoden in zwei Regeln
- In der Praxis: **Mehrere** Methoden in jeder Regel
  - Oft auch noch aus verschiedenen Untermodulen

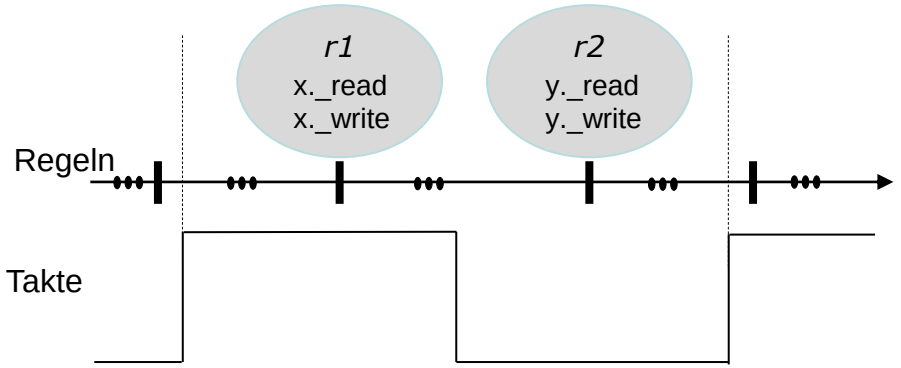
- Definition: Ein **Konflikt** besteht zwischen zwei Regeln *r1* und *r2* genau dann, wenn in einer Instanz *x* zwischen irgendeinem Paar von Methoden *x.m1* in *r1* und *x.m2* in *r2* ein Konflikt besteht

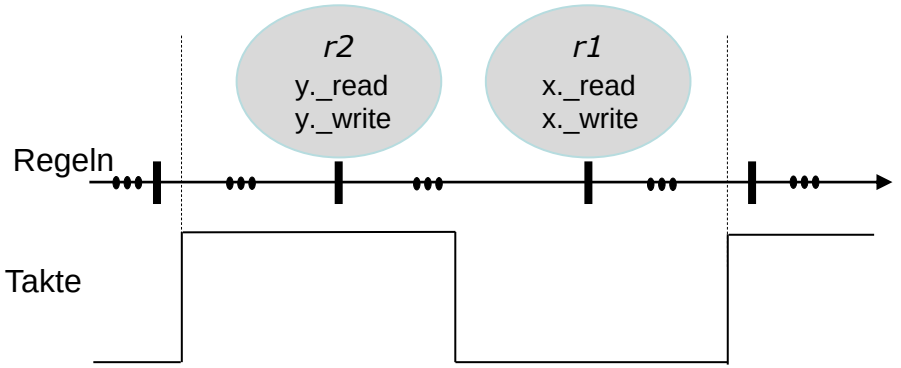


  - Wichtig: Konflikte sind nur zwischen Methoden der **gleichen** Instanz möglich!

Beispiel: Kein Regelkonflikt

```
rule r1;
   x <= x + 1;
endrule

rule r2;
   y <= y + 2;
endrule
```





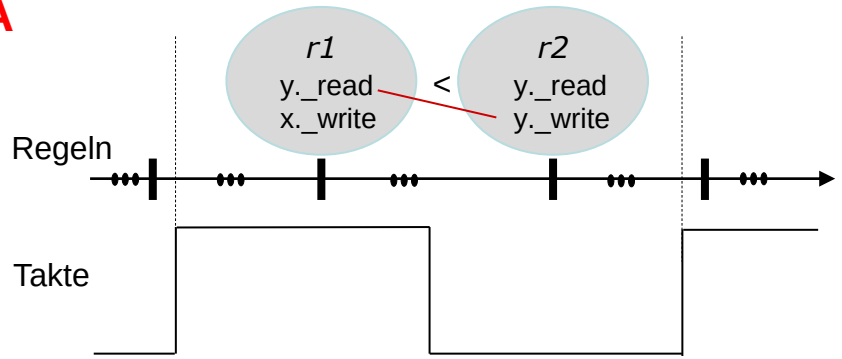
- Zugriff auf unterschiedliche Instanzen
- Register-Instanz **`x`** in **`r1`**
- Register-Instanz **`y`** in **`r2`**
- Alle Ablaufpläne konfliktfrei
- bsc kann Hardware mit nebenläufigen **`r1`** und **`r2`** erzeugen
- Ausführung in einem Takt

Beispiel: Kein Regelkonflikt

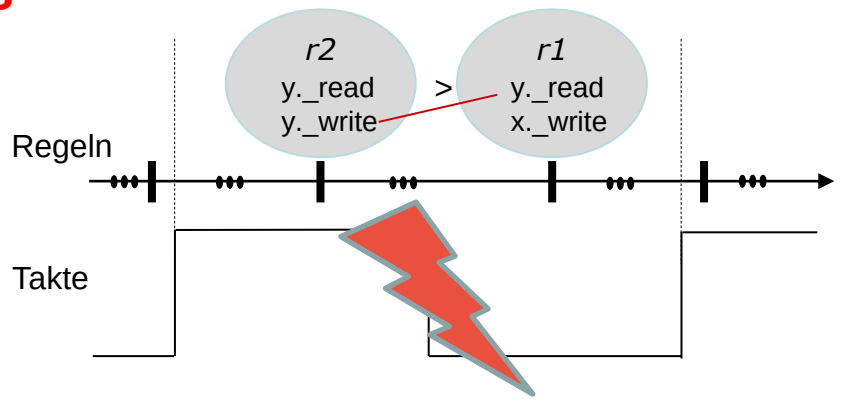
```
rule r1;
   x <= y + 1;
endrule

rule r2;
   y <= y + 2;
endrule
```

- Teilweise Zugriff auf gleiche Instanzen
- Register-Instanz **x** und **y** in **r1**
- Register-Instanz **y** in **r2**
- Einschränkung: **y._read < y._write**
- Ablaufplan A respektiert Einschränkung
- Ablaufplan B verletzt Einschränkung
- bsc wählt A, da dadurch Hardware mit Nebenläufigkeit möglich

A

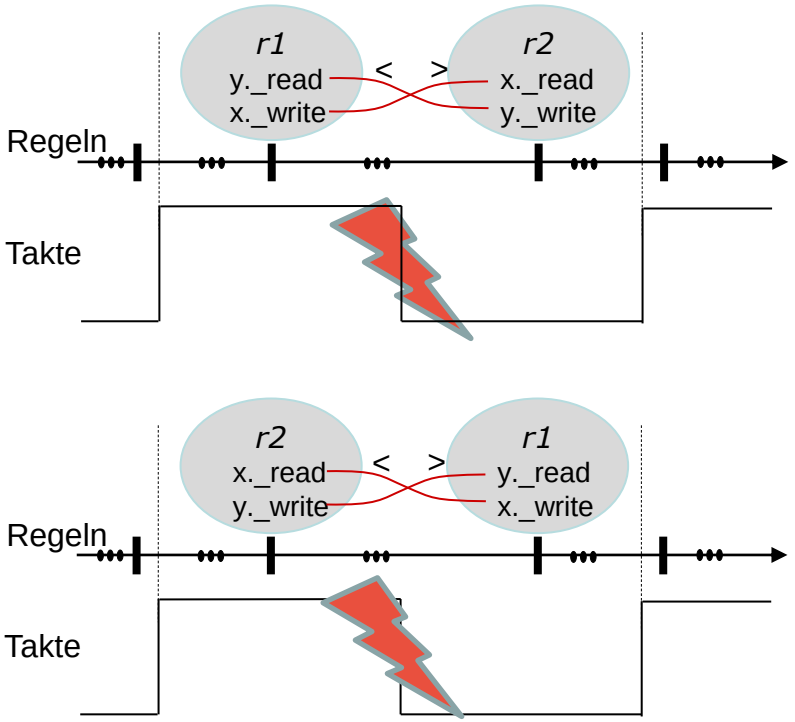


B



Beispiel: Regelkonflikt

```
rule r1;
   x <= y + 1;
endrule

rule r2;
   y <= x * 2;
endrule
```



- Immer Zugriff auf gleiche Instanzen
- Register-Instanz `x` und `y` in **`r1`**
- Register-Instanz `x` und `y` in **`r2`**
- **`{x,y}._read < {x,y}._write`**
- Alle Ablaufpläne verletzten Einschränkung
- bsc erzeugt Hardware für sequentielle Ausführung
- Nur so wird Semantik korrekt abgebildet

Parallelität und Nebenläufigkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

```
rule rule2a (condition);
   x <= y + 1;
   y <= x * 2;
endrule : rule2a
```

Reihenfolge im Quelltext irrelevant

```
rule rule2b (condition);
   y <= x * 2;
   x <= y + 1;
endrule : rule2b
```

- Echte Parallelität von Aktionen innerhalb einer Regel

```
rule r1;
   x <= y + 1;
endrule

rule r2;
   y <= x * 2;
endrule
```

- Verschiedene Regeln laufen logisch sequentiell ab
- Nebenläufig nur, wenn konfliktfrei

Ausführungssemantik für Regeln 1

Logische Sicht, ohne Hardware-Aspekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Regeln laufen **logisch** betrachtet ab …

- … **instantan**
  - Alle Aktionen innerhalb der Regel finden zum selben Zeitpunkt statt
- … **vollständig**
  - Nach Feuern der Regel werden alle enthaltenen Aktionen ausgeführt
- … **geordnet**
  - Eine Regel wird entweder vor **oder** nach anderen Regeln ausgeführt, niemals gleichzeitig

- Zusammengefasst: Bezeichnet als **atomare Ausführung**
  - Im Sinne von “unteilbar”
  - Ähnlich zu atomaren Transaktionen bei Datenbanken

Ausführungssemantik für Regeln 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Abbildung der logischen Sicht auf **synchrone** Schaltungen

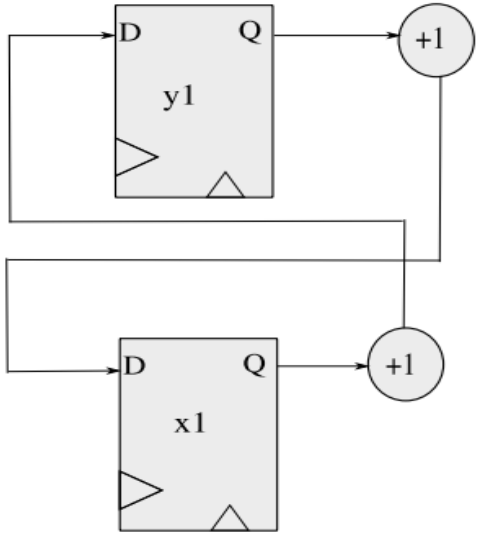
- Wähle eine freigegebene Regel aus und führe sie aus (feuern)
- Jede Regel feuert maximal einmal während eines Taktes
- Konfliktbehaftete Regeln können nicht im gleichen Takt feuern

- Zwei häufige Ursachen für Konflikte
  - Zustandselemente können nur einmal je Takt umschalten
    - Lesen eines geänderten Zustandes im selben Takt ist nicht (ohne weiteres) möglich
    - *Rule ordering conflict*
  - Hardware-Ressourcen (z.B. Drähte) können nur einmal je Takt benutzt werden
    - *Rule resource conflict*

- Auflösung des Konflikts: Wähle (zunächst) willkürlich eine Regel zum Feuern
  - … und hebe die Freigabe der damit in Konflikt stehenden Regeln auf
  - Compiler gibt aber Warnung aus

Ausführungssemantik für Regeln 3

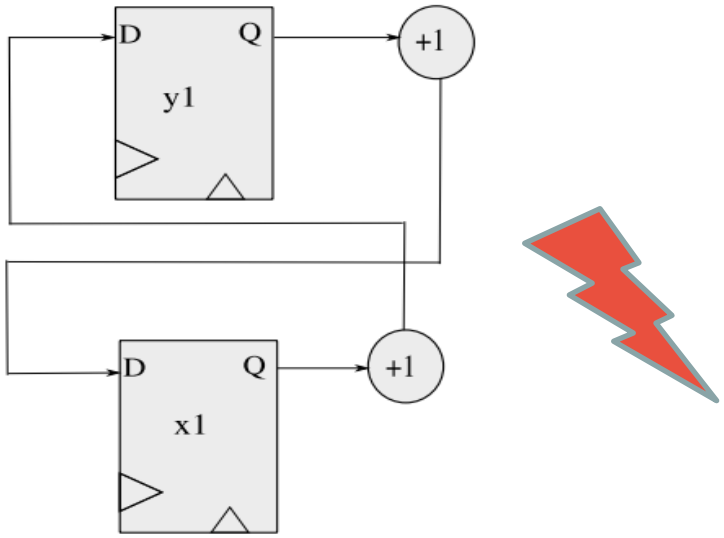
```
rule r1;
  $display ("swap");
  x1 <= y1 + 1;
  y1 <= x1 + 1;
endrule
```



- Register **x1** und **y1** werden vor Ausführung der Regel gelesen
- Beide Register werden nach Ausführung der Regel geschrieben
  - **x1** mit (dem alten Wert von **x1**) + 1
  - **y1** mit (dem alten Wert von **y1**) + 1
- Parallele Komposition von Aktionen innerhalb einer Regel

Ausführungssemantik für Regeln 4

```
rule r2a;
  $display ("r2a");
  x1 <= y1 + 1;
endrule

rule r2b;
  $display ("r2b");
  y1 <= x1 + 1;
endrule
```



- Mögliche logische Abläufe bei x1=10 und y1=100
  - r2a vor r2b: x1=101, y1=102
  - r2b vor r2a: x1=12, y1=11
- Vorgeschlagene Hardware würde aber liefern: x1=101, y1=11
  - Nicht konsistent mit logischem Ausführungsmodell

Ausführungssemantik für Regeln 5

TECHNISCHE UNIVERSITÄT DARMSTADT

- Problem wird vom Compiler bemerkt und unterbunden
  - … aber auf willkürliche Art und Weise

```
Warning: "Tb.bsv", line 16, column 8: (G0010)
  Rule "r2b" was treated as more urgent than "r2a". Conflicts:
    "r2b" cannot fire before "r2a": calls to y1.write vs. y1.read
    "r2a" cannot fire before "r2b": calls to x1.write vs. x1.read
Warning: "Tb.bsv", line 30, column 9: (G0021)
  According to the generated schedule, rule "r2a" can never fire.
```

- Compiler entscheidet sich willkürlich für Ausführung von **`r2b`**
  - … unterbindet dann konsequenterweise die Ausführung von **`r2a`** vollständig

- Benutzereingriff in Auswahl ist möglich
  - *scheduling attributes*: Benutzen wir aber erstmal nicht
  - Stattdessen Konflikte schon beim Verfassen der Regeln vermeiden

Praktische Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- *N*! mögliche Ablaufpläne für die Regeln *r1* bis *rN*

- `bsc` wählt Ablaufplan mit höchstem Grad an Nebenläufigkeit aus
  - Möglichst viele Regeln feuern in einem Takt
  - Idee: Möglichst wenige Takte zur Ausführung des Algorithmus

- Behandlung von Konflikten
  - Falls immer auftretende Konflikte erkannt werden
    - Auswahl einer Untermenge von konfliktfreien Regeln
    - Entfernen aller konfliktbehafteten Regeln (führen nicht zu Hardware)
  - Falls Konflikte nicht immer auftreten
    - Führe Untermenge von konfliktfreien Regeln aus
    - Erzeuge Hardware, um Ausführung noch konfliktbehafteter Regeln in diesem Fall zu unterbinden

„Programmierersicht“

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beim Entwickeln in Bluespec SystemVerilog zunächst

logische Sicht

verwenden

  - Atomare Ausführung von Regeln
  - Unabhängig von Zieltechnologie
    - Bluesim für BSV
    - Simulator für nach Verilog compiliertes BSV
    - Hardware-Synthese für nach Verilog compiliertes BSV

- Erst wenn BSV Modell funktioniert
  - Optimierung der Abbildung auf Taktzyklen
  - Feinabstimmung für bestimmte Zieltechnologie

Weiteres Vorgehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiele ausprobieren
  - Kommandozeile reicht
- Zur Vertiefung im Buch “Bluespec by Example” lesen
  - Kapitel 5 und 7

EINFACHE PIPELINES

Exkurs: Unterschnittstellen

sub interfaces

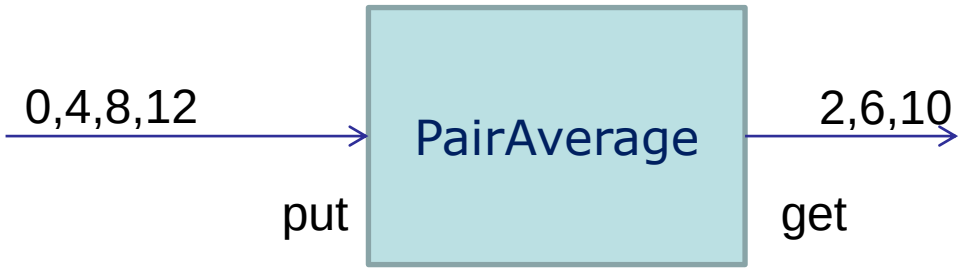
TECHNISCHE UNIVERSITÄT DARMSTADT

- Komplizierte Interfaces
  - Einmal definieren
  - Häufig wiederverwenden

- Bluespec Bibliothek hat eine Sammlung von wiederverwendbaren Interfaces
  - Die dann in eigenen Implementierungen mit Leben gefüllt werden können

Beispiel für Standardschnittstelle

Durchschnitt von aufeinanderfolgenden Werten in Folge



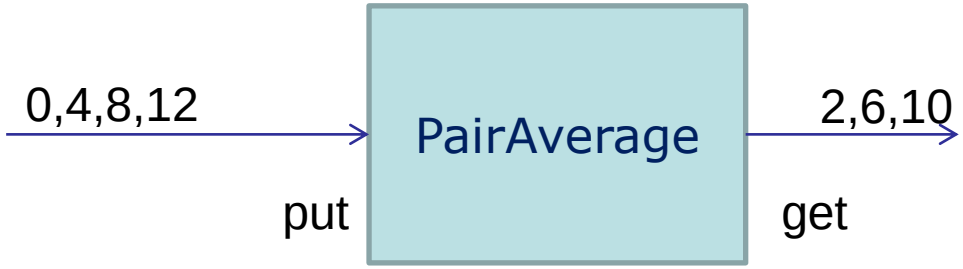
- **Put**: Gebe einen Wert ein (kann blocken!)
  - Hier aber nicht (verwerfe nicht abgerufene Durchschnitte)
- **Get**: Rufe einen Wert ab (kann blocken!)
  - Blockt, wenn noch keine zwei Werte eingegeben wurden
- **GetPut**: generisches Interface für solche Operationen
  - Hier: Für Integer verwendet

Beispiel: PairAverage 1

Schnittstelle und Moduldefinition

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import GetPut :: * ; // aus Bibliothek

interface PairAverage;

  interface Put#(int) data_in;
  interface Get#(int) pair_average;

endinterface

module mkPairAverage (PairAverage);

  Reg#(int)  oldval  <- mkReg(0);
  Reg#(int)  newval  <- mkReg(0);
  Reg#(Bool) got_old <- mkReg(False);
  Reg#(Bool) got_new <- mkReg(False);

  interface Put data_in;
    method Action put(int val);
      if (got_new) begin
        oldval  <= newval;
        got_old <= True;
      end
      newval  <= val;
      got_new <= True;
    endmethod
  endinterface
```




```
  interface Get pair_average;
    method ActionValue#(int) get() if (got_new && got_old);
      return (oldval+newval)/2;
    endmethod
  endinterface
endmodule
```


Embedded Systems & Applications

Beispiel: PairAverage 2

Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module top (Empty);

  Reg#(int)          invalue <- mkReg(0);
  PairAverage        pa      <- mkPairAverage;

  rule average; // kann blocken
    $display("Average of last two items: %d", pa.pair_average.get());
  endrule

  rule counter;
    invalue <= invalue + 4;
    pa.data_in.put(invalue);
    $display("Entered %d", invalue);
    if (invalue == 32)
      $finish;
  endrule
endmodule
```

```
Entered         0
Entered         4
Average of last two items:      2
Entered         8
Average of last two items:      6
Entered         12
Average of last two items:      10
Entered         16
Average of last two items:      14
Entered         20
Average of last two items:      18
Entered         24
Average of last two items:      22
Entered         28
Average of last two items:      26
Entered         32
```

- Auch kompliziertere Schnittstellen in Standardbibliothek (→ Doku)
- Nützlich: Typkonvertierung hin zu Standardschnittstellen
  - Benutzen wir gleich …

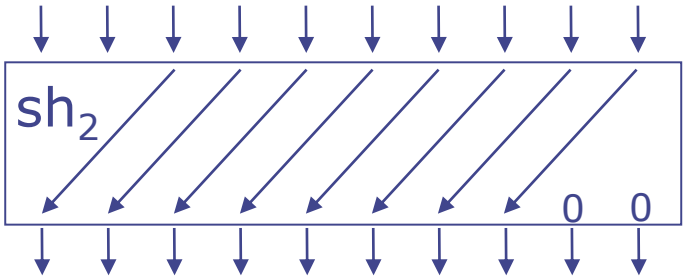
Shifter mit variabler Distanz

- Kann um variable Anzahl von Stellen schieben
- Im Beispiel: Schieben nach links x << y
- Vorgehensweise:

x << y = x << 1 (↓) << 2 (↓) << 4

falls y[0]==1
falls y[1]==1
falls y[2]==1

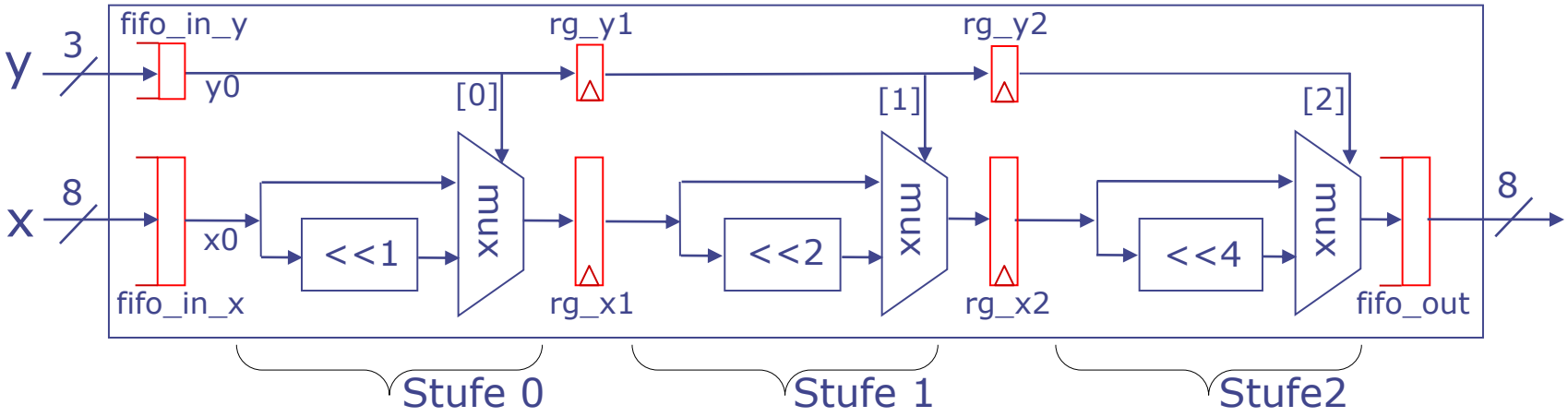
- Allgemein: Verschieben um konstante Distanz ist trivial
  - Realisiert durch Ändern der Anschlüsse mit reiner Verdrahtung



x << 2

1. Realisierung des Shifters

Als synchrone statische starre Pipeline




```
rule rl_all_together;
   // Stufe 0
   let x0 = fifo_in_x.first; fifo_in_x.deq;
   let y0 = fifo_in_y.first; fifo_in_y.deq;
   rg_x1 <= ((y0 [0] == 0) ? x0 : (x0 << 1));
   rg_y1 <= y0;

   // Stufe 1
   rg_x2 <= ((rg_y1 [1] == 0) ? rg_x1 : (rg_x1 << 2));
   rg_y2 <= rg_y1;

   // Stufe 2
   fifo_out_z.enq (((rg_y2 [2] == 0) ? rg_x2 : (rg_x2 << 4)));
endrule
```

1. Realisierung des Shifters

Als synchrone statische starre Pipeline

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface Shifter_Ifc;
   interface Put #(Bit #(8)) put_x;
   interface Put #(Bit #(3)) put_y;
   interface Get #(Bit #(8)) get_z;
endinterface
```

Hier Get/Put-Standardschnittstelle benutzt

```
module mkShifter (Shifter_Ifc);
   FIFOF #(Bit #(8))  fifo_in_x  <- mkFIFOF;
   FIFOF #(Bit #(3))  fifo_in_y  <- mkFIFOF;
   FIFOF #(Bit #(8))  fifo_out_z <- mkFIFOF;

   Reg #(Bit #(8)) rg_x1 <- mkRegU;
   Reg #(Bit #(3)) rg_y1 <- mkRegU;

   Reg #(Bit #(8)) rg_x2 <- mkRegU;
   Reg #(Bit #(3)) rg_y2 <- mkRegU;

   rule rl_all_together;
      …
      … auf voriger Folie gezeigt…
      …
   endrule

   interface put_x = toPut (fifo_in_x);
   interface put_y = toPut (fifo_in_y);
   interface get_z = toGet (fifo_out_z);
endmodule
```

Konvertierung zwischen Standardschnittstellen, hier von FIFOF nach Get/Put

Beispiele für Standardschnittstellen

```
interface Put #(type t);
   method Action put (t x);
endinterface

interface Get #(type t);
   method ActionValue #(t) get ();
endinterface

interface FIFOF #(type t);
   method Action enq (t x);
   method t      first ();
   method Action deq ();
   method Bool   notFull;
   method Bool   notEmpty;
   method Action clear ();
endinterface
```

1. Realisierung des Shifters

Testrahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkTestbench (Empty);
   Shifter_Ifc  shifter <- mkShifter;

   Reg #(Bit #(4)) rg_y <- mkReg (0); // 4b für Schleifenabbruch

   rule rl_gen (rg_y < 8);
      shifter.put_x.put (8'h01);
      shifter.put_y.put (truncate (rg_y));  // oder: rg_y[2:0]
      rg_y <= rg_y + 1;
   endrule

   rule rl_drain;
      let z <- shifter.get_z.get ();
      $display ("Output = %8b", z);
      if (z == 8'h80) $finish ();           // 8'b10000000
   endrule
endmodule: mkTestbench
```

rl_gen *erzeugt diese Eingaben:*

00000001 0
00000001 1
00000001 2
…
00000001 7

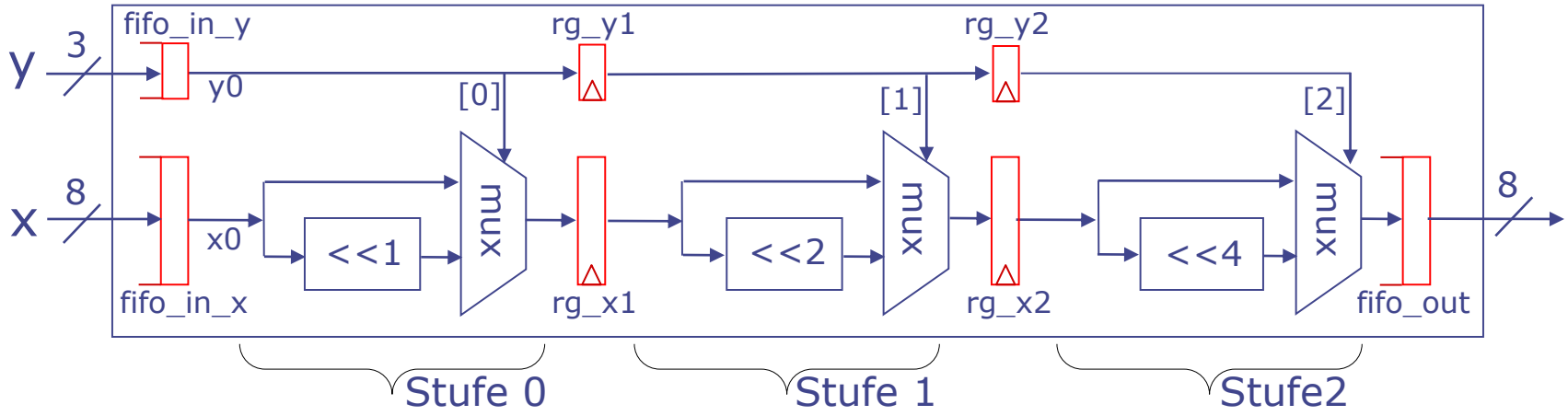
rl_drain *sollte diese Ausgaben beobachten:*

00000001
00000010
00000100
…
10000000

Embedded Systems & Applications

1. Realisierung des Shifters

Problem



Tatsächliche Ausgabe

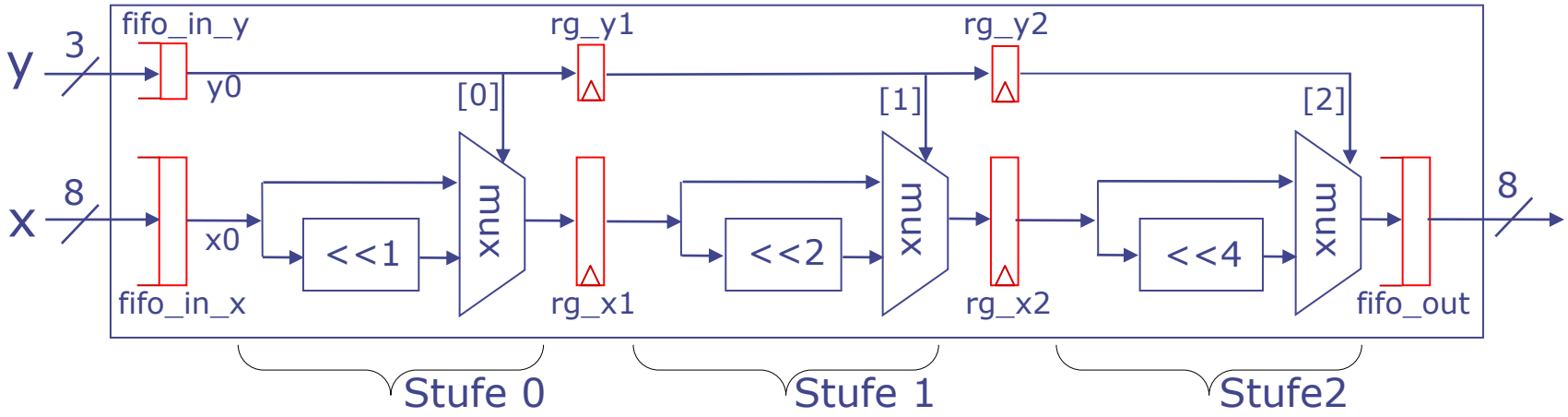
```
01010101
10101000
00000001
00000010
…
00010000
00100000
```

Warum?

Danach “hängt” Simulation

1. Realisierung des Shifters

Problem



Tatsächliche Ausgabe

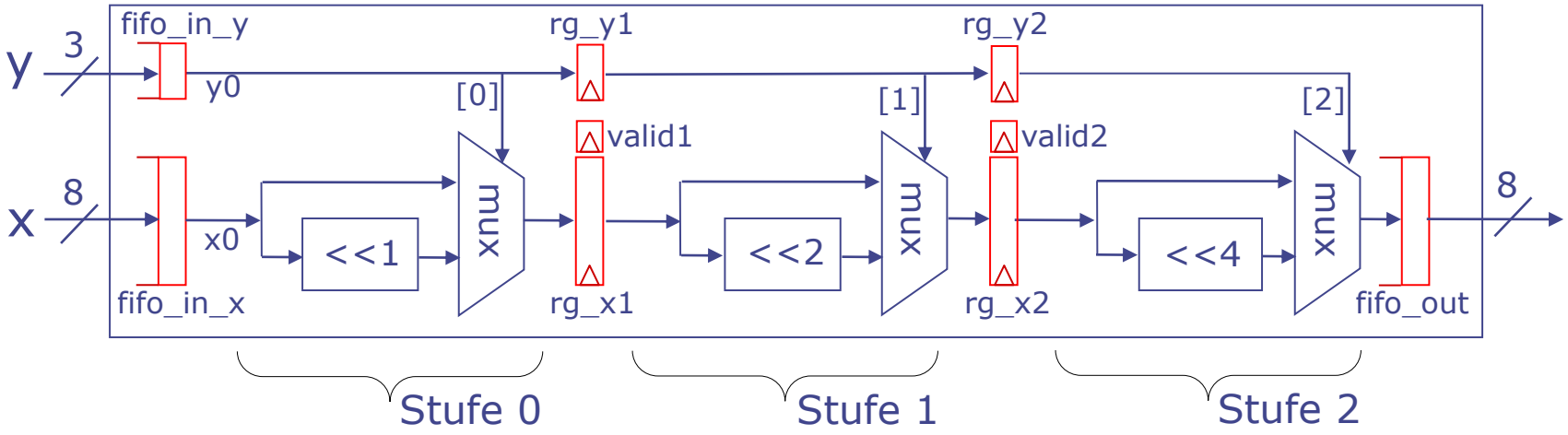
```
01010101
10101000
00000001
00000010
…
00010000
00100000
```

Danach "hängt" Simulation

- Anfang: "Undefinierte Werte" in **`rg_{x,y}{1,2}`**
  - Werden durch Pipeline geschoben
- Ende: Abriss des Datenstromes
  - **`rl_gen`** beschickt Eingabe-FIFOs nicht mehr
  - **`rl_all_together`** blockt, da **`.first`** blockt
  - Keine neuen Einträge mehr in Ausgabe-FIFO
  - **`rl_drain`** blockt
  - Letzte beide Werte hängen nun in Pipeline fest

2. Realisierung des Shifters

Noch synchrone statische starre Pipeline, Fehler behoben



- Führe Buch über **gültige Daten** in Pipeline
  - Register **`valid{1,2}`**
- Erkenne nun
  - Wenn Eingabe FIFOs **leer** sind
  - Wenn Zwischenregister **`rg_{x,y}{1,2}`** **leer** sind
- Passe dann **Verhalten** an
- Könnte direkt in BSV formuliert werden, geht aber besser …

Exkurs: Datentyp `Maybe`

Grundlage

TECHNISCHE UNIVERSITÄT DARMSTADT

- Basiert auf varianten Verbundtypen (*tagged unions*)

Beispiel in Pascal (NICHT Bluespec!)

```
type shapeKind = (square, rectangle, circle);
     shape = record
                 centerx : integer;
                 centery : integer;
                 case kind : shapeKind of (* type tag *)
                    square : (side : integer);
                    rectangle : (length, height : integer);
                    circle : (radius : integer);
             end;
```

Semantik: Zu jedem Zeitpunkt existiert genau ein Satz Attribute **`side`**, (**`length`**, **`height`**), **`radius`**: Auswahl der Variante mit *type tag*

Heutzutage eher mit Vererbung realisiert

Exkurs: Datentyp `Maybe`

Bluespec

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Zu jedem Zeitpunkt existiert genau eine der beiden Komponenten: Festgelegt über *type tag*

Werte des Typs können auf Gleichheit geprüft und als Bits dargestellt werden

Erzeuge neuen Maybe-Wert

```
tagged Invalid
```

Maybe-Wert mit type tag Invalid, kein weiterer Wert

```
tagged Valid expression
```

Maybe-Wert mit type tag Valid, speichert Wert von *expression*

Prüfe Maybe-Wert auf Gültigkeit und verwende ggf. gespeicherten Wert

```
if (value matches tagged Valid .x)
    ... hier ist x definiert, enthält den gültigen Wert...
else
    ... hier den Fall "ungültiger" Wert behandeln...
```

2. Realisierung des Shifters

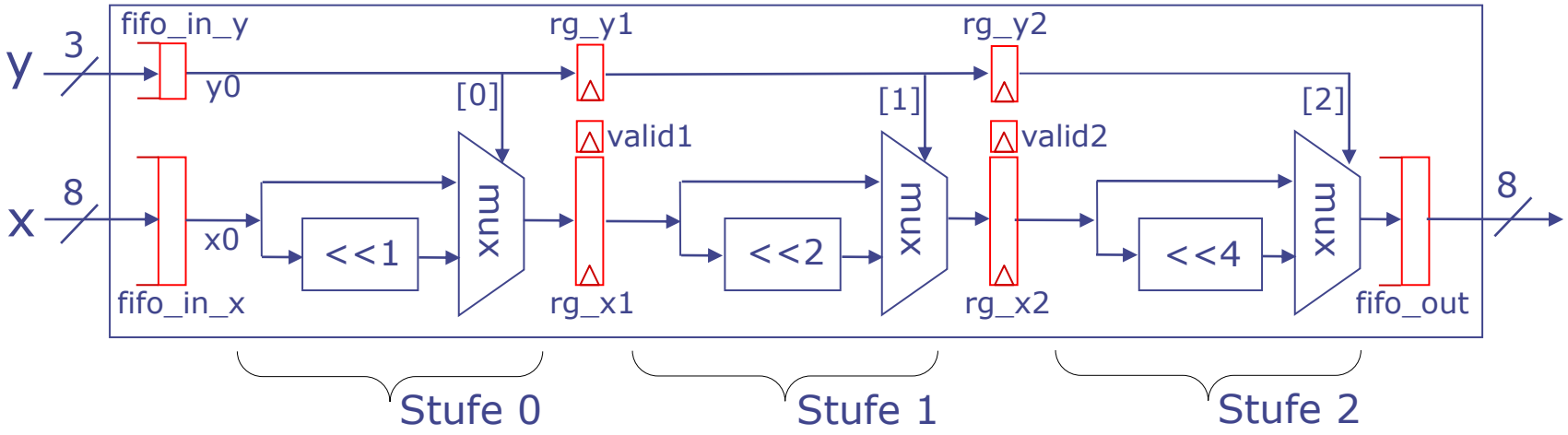
Synchrone statische starre Pipeline mit Gültigkeitsangaben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkShifter (Shifter_Ifc);
   …
   Reg #(Maybe #(Bit #(8))) rg_x1 <- mkReg (tagged Invalid);
   Reg #(Bit #(3))          rg_y1 <- mkRegU;

   Reg #(Maybe #(Bit #(8))) rg_x2 <- mkReg (tagged Invalid);
   Reg #(Bit #(3))          rg_y2 <- mkRegU;

   rule rl_all_together;
      // Stage 0
      Bit #(3) y0 = ?;
      if (fifo_in_x.notEmpty) begin
         let x0 = fifo_in_x.first; fifo_in_x.deq;
             y0 = fifo_in_y.first; fifo_in_y.deq;
         rg_x1 <= tagged Valid ((y0 [0] == 0) ? x0 : (x0 << 1));
      end else
         rg_x1 <= tagged Invalid;
      rg_y1 <= y0;

      // Stage 1
      if (rg_x1 matches tagged Valid .x1)
         rg_x2 <= tagged Valid ((rg_y1 [1] == 0) ? x1 : (x1 << 2));
      else
         rg_x2 <= tagged Invalid;
      rg_y2 <= rg_y1;

      // Stage 2
      if (rg_x2 matches tagged Valid .x2)
         fifo_out_z.enq (((rg_y2 [2] == 0) ? x2 : (x2 << 4)));
   endrule
   …
endmodule
```

2. Realisierung des Shifters

Erprobung



`rl_gen` erzeugt folgende Eingaben:

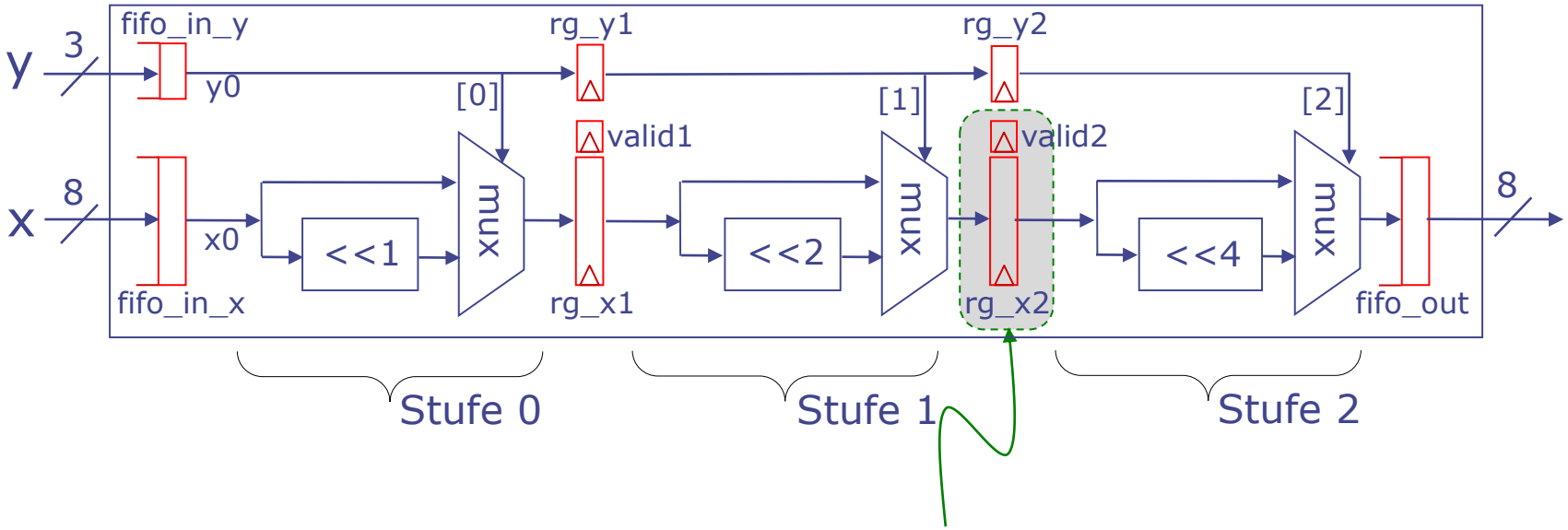
```
00000001   0
00000001   1
00000001   2
...
00000001   7
```

`rl_drain` beobachtet folgende Ausgaben:

```
00000001
00000010
00000100
...
10000000
```

Shifter-Simulationen verhalten sich wie erwartet!

Von Gültigkeitsbits zu FIFOs



- Register mit Gültigkeitsbit kann als 1-Element FIFO angesehen werden
  - Gültig=Wahr: FIFO ist voll
  - Gültig=Falsch: FIFO ist leer
  - Speichern von Werten in Register und markieren als Gültig: Einreihen in FIFO
  - Lesen von Werten aus Register und markieren als Ungültig: Entnehmen aus FIFO
- Damit einfachere Formulierung als statische elastische Pipeline

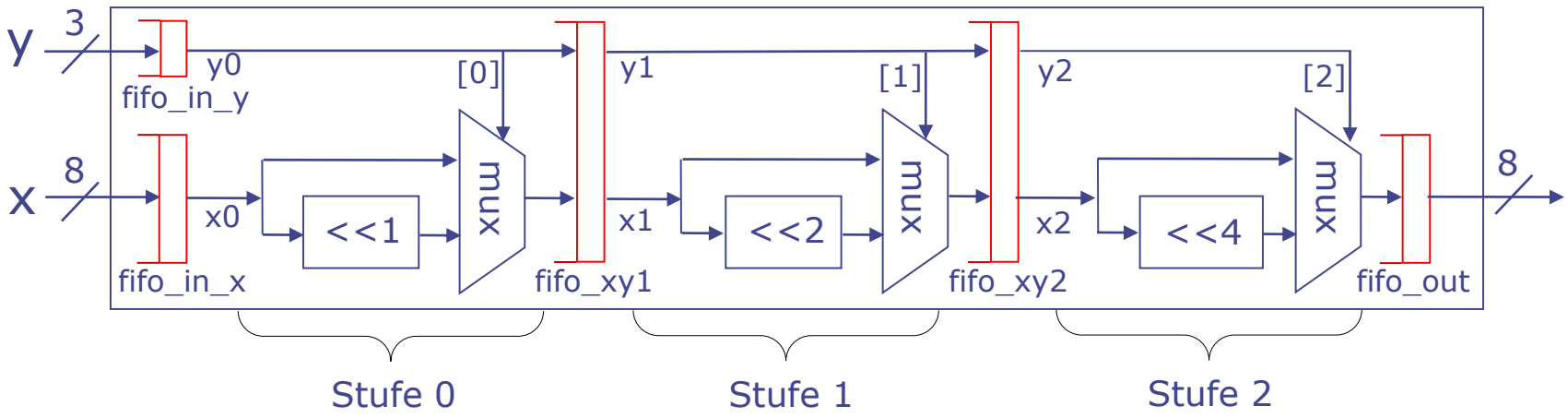
Exkurs: Tupel-Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zusammenfügen von mehreren Einzelwerten zu einem zusammengesetzten Wert
  - Sehr nützlich, Bluespec definiert Typen für Tupel von 2…8 Elementen
- Beispiel: **`Tuple2 #(t1, t2)`**
  - Wertepaar mit erstem Wert von Typ ***`t1`***, zweitem vom Typ ***`t2`***
- Erzeugen von Werten: **`tuple2 (ausdruck1, ausdruck2)`**
- Lesen von Komponenten
  - Mit Funktion **`tpl_j`**: **`tpl_1(ausdruck)`**, **`tpl_2(ausdruck)`**, …
  - Über “Mustervorlage”: **`match { .x, .y } = ausdruck`**
    - Deklariert temporäre Variablen **`x`** und **`y`** mit den Werten der beiden Komponenten

3. Realisierung des Shifters

Statische elastische Pipeline, Aufbau



3. Realisierung des Shifters

Statische elastische Pipeline, Aufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkShifter (Shifter_Ifc);
   …
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy1 <- mkFIFOF;
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy2 <- mkFIFOF;

   rule rl_stage0;
      let x0 = fifo_in_x.first;  fifo_in_x.deq;
      let y0 = fifo_in_y.first;  fifo_in_y.deq;
      fifo_xy1.enq (tuple2 (((y0 [0] == 0) ? x0 : (x0 << 1)), y0));
   endrule

   rule rl_stage1;
      match { .x1, .y1 } = fifo_xy1.first;  fifo_xy1.deq;
      fifo_xy2.enq (tuple2 (((y1 [1] == 0) ? x1 : (x1 << 2)), y1));
   endrule

   rule rl_stage2;
      match { .x2, .y2 } = fifo_xy2.first;  fifo_xy2.deq;
      fifo_out_z.enq ((y2 [2] == 0) ? x2 : (x2 << 4));
   endrule
   …
endmodule
```

3. Realisierung des Shifters

Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Statisch** (Gegenteil wäre dynamisch)
  - Konstante Latenz von der Eingabe zur Ausgabe eines Datums
- **Elastisch** (Gegenteil wäre starr)
  - Daten in unterschiedlichen Stufen schreiten mit unterschiedlichem Fortschritt durch Pipeline voran
- Jede Regel kann nun unabhängig von den anderen feuern
- Viel aufwendige Steuerlogik ist **entfallen**
  - Manuelle Verwaltung der Gültigkeitsbits
- Wird nun **automatisch** aus FIFO-Bedingungen hergeleitet
  - … und in Regelbedingungen **hochgezogen**

Ausdrucksfähigkeit von BSV

TECHNISCHE UNIVERSITÄT DARMSTADT

- Von Verilog-artigen synchronen starren Strukturen
- … hin zu elastischen Strukturen mit komplexer Flußkontrolle
  - Mit weitgehend automatischer Herleitung der Steuerlogik

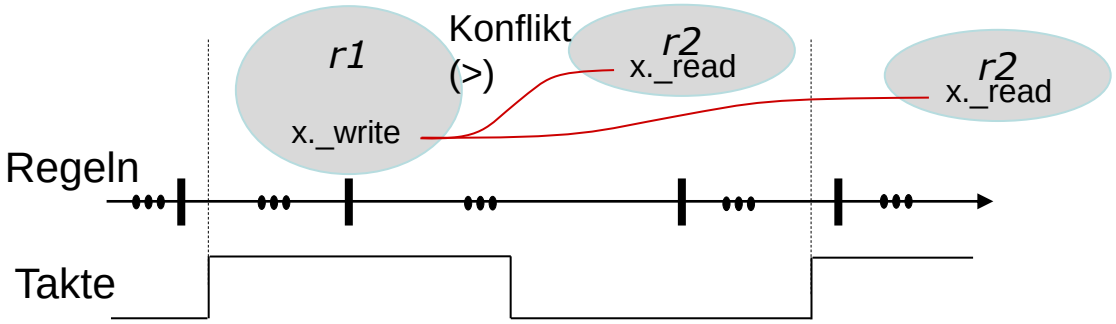
- Starre Strukturen haben oft Skalierungsprobleme
  - Bei größeren Schaltungen: Schwer zu entwerfen und zu debuggen

- Elastische Strukturen passen gut zu modernem **GALS** Berechnungsmodell
  - *Globally asynchronous, locally synchronous*
  - “Inseln” betrieben mit synchroner Logik
  - Kommunizieren untereinander asynchron
  - Vermeidet Problem, nur einen Takt global auf ganzem System verteilen zu müssen

- Vertiefung: Kapitel 5 von BSV-by-Example

MEHR NEBENLÄUFIGKEIT

Ausführung in weniger Taktzyklen



- Verschiedene Regeln kommunizieren über Register
- Änderungen werden erst einen Takt später sichtbar
  - `_read` < `_write`
  - Kann damit zu längeren Ausführungszeiten führen
- "Schnellere" Kommunikation
  - Änderungen sollen innerhalb eines Taktes sichtbar werden
  - … aber immer noch mit exakt spezifizierter Nebenläufigkeit

Nebenläufige Register

***Concurrent Registers* (ehemals *Ephemeral History Regs*.)**

TECHNISCHE UNIVERSITÄT DARMSTADT

- Können neben der Datenhaltung von Takt zu Takt …
  - Übliche Betriebsart von Registern
- … auch eine **Historie** von Werten innerhalb eines Taktes führen
  - Ist **vergänglich** (*ephemeral*), gilt nur innerhalb des Taktes
- Relativ neue Entwicklung in Bluespec
  - Bisher nicht in BSV-by-Example beschrieben, aber im Reference Guide
  - Aber schon in Standard-Bluespec Distribution aufgenommen
- Ersetzt mittlerweile in vielen Fällen ältere Mechanismen
  - **`RWire`**
    - Potentiell fehleranfälliger

Beispiel I für CRegs

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zähler
  - Vorzeichenbehaftete 4b breite ganze Zahlen
  - Saturierende Arithmetik
    - Einschränkung auf Wertebereich -8 … +7
    - Kein Überlauf/Unterlauf
  - Variable Schrittweite: 4b vorzeichenbehaftete ganze Zahlen
  - Zwei Ports
    - Sollen zwei gleichzeitige Aktualisierungen des Zählerstandes ermöglichen

```
interface UpDownSatCounter_Ifc;
   method ActionValue #(Int #(4)) countA (Int #(4) delta);
   method ActionValue #(Int #(4)) countB (Int #(4) delta);
endinterface
```

- Vorsicht: Operationen sind nicht mehr kommutativ

1. Versuch

Implementierung mit konventionellen Registern

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
   Reg #(Int #(4)) ctr <- mkReg (0);

   function ActionValue #(Int #(4)) fn_count (Int #(4) delta);
      actionvalue
         // Erhöhen der Wortbreite zum Vermeiden von Über/Unterlauf
         Int #(5) new_val = extend (ctr) + extend (delta);
         if (new_val > 7) ctr <= 7;
         else if (new_val < -8) ctr <= -8;
         else ctr <= truncate (new_val);

         return ctr;   // Beachte: gibt _alten_ Wert zurück
      endactionvalue
   endfunction

   method countA (Int #(4) deltaA) = fn_count (deltaA);
   method countB (Int #(4) deltaB) = fn_count (deltaB);
endmodule
```

- **`extend(e)`** : Erweiterung von Wert *e* auf breitere Darstellung (sign/zero extension)
- **`truncate(e)`** : Verkürze *e* auf schmalere Darstellung (weglassen vom msb her)

1. Versuch

Testrahmen

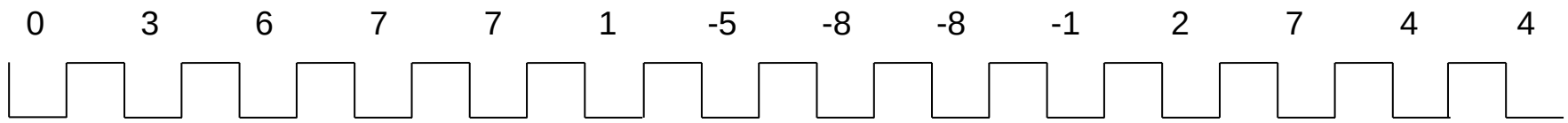
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkTest (Empty);
   UpDownSatCounter_Ifc  ctr <- mkUpDownSatCounter;
   Reg #(int) step <- mkReg (0);
   Reg #(Bool) flag0 <- mkReg (False); Reg #(Bool) flag1 <- mkReg (False);

   function Action count_show (Integer rulenum, Bool a_not_b, Int #(4) delta);
      action
         let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
         let cur_time <- $stime;
         $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_time/10, rulenum, x, delta);
      endaction
   endfunction

   // Regeln 0..9 sind sequentiell, testen countA und countB einzeln
   rule r0  (step == 0);  count_show (0, True,   3); step <= 1;  endrule
   rule r1  (step == 1);  count_show (1, True,   3); step <= 2;  endrule
                  ... etc, erzeugte Eingabedatenfolge: 3,3, -6,-6,-6,-6, 7, 3,
   // Potentiell nebenläufige Ausführung
   rule r10 (step == 10 && !flag0); count_show (10,True,   6); flag0 <= True; endrule
   rule r11 (step == 10 && !flag1); count_show (11,False, -3); flag1 <= True; endrule

   // Abschlusswert des Zählers ausgeben (Zähler bleibt konstant)
   rule r12 (step == 10 && flag0 && flag1); count_show (12,True, 0); $finish; endrule
endmodule: mkTest
```

1. Versuch

Erwartete Ausgaben bei Simulation

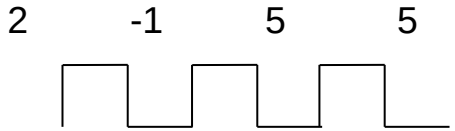
Falls **`r10`** im Ablaufplan **vor** **`r11`** liegt:



Regel	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	**r10**	**r11**	r12
delta	3	3	3	3	-6	-6	-6	-6	7	3	6	-3	0
Ausgabe	0	3	6	7	7	1	-5	-8	-8	-1	2	**7**	**4**

Falls **`r11`** im Ablaufplan **vor** **`r10`** liegt:

● ● ● *Identisch für **`r0`** bis **`r9`*** ● ● ●



r11	**r10**	r12
-3	6	0
2	**-1**	**5**

1. Versuch

Compilierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Warning: "Test.bsv", line 16, column 8: (G0010)
  Rule "r10" was treated as more urgent than "r11". Conflicts:
    "r10" cannot fire before "r11": calls to ctr.countA vs. ctr.countB
    "r11" cannot fire before "r10": calls to ctr.countB vs. ctr.countA
```

- **Konflikt** zwischen **`r10`** und **`r11`**
  - Keine Ausführung im gleichen Takt möglich
  - **`countA`** und **`countB`** greifen auf gleiche Registerinstanz **`ctr`** zu
  - Verletzen **`_read`** < **`_write`**

- Bsc trifft hier **willkürliche** Entscheidung: **`r10`** im Ablaufplan **vor** **`r11`**
  - Falls beide bereit in einem Takt (CAN_FIRE): Nur **`r10`** wird feuern (WILL_FIRE)
  - **`r11`** wird nur betrachtet, wenn **`r10`** nicht bereit war

- Ablaufplanung könnte durch Benutzer **beeinflusst** werden
  - **`(* descending_urgency = "r11, r10" *)`**
  - siehe Bluespec Reference Guide, Abschnitt 13.3.3

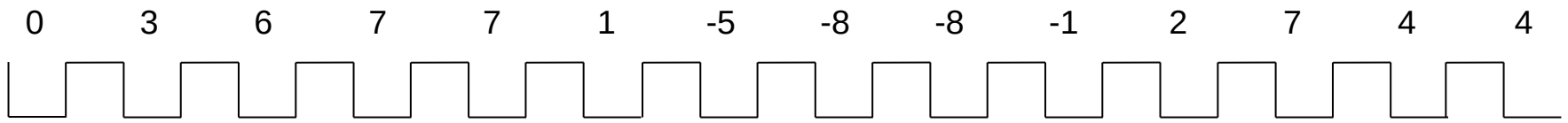
1. Versuch

Tatsächliche Ausgabe

TECHNISCHE UNIVERSITÄT DARMSTADT

- Entspricht 1. Ablaufplan
  - **`r10`** vor **`r11`**

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 12, r11: is 7, count (-3)
cycle 13, r12: is 4, count (0)
```



Regel	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	**r10**	**r11**	r12
delta	3	3	3	3	-6	-6	-6	-6	7	3	6	-3	0
Ausgabe	0	3	6	7	7	1	-5	-8	-8	-1	2	**7**	**4**

1. Versuch: Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ist **funktional** zwar korrekt …
- … aber nicht wirklich ein 2-Port Zähler

- Charakterisierung "*n*-Port" beschreibt üblicherweise *n* **gleichzeitige** Zugriffe je Takt

- Kann hier nicht klappen
  - Es gibt nur ein gemeinsames Register **`ctr`** für **`countA`** und **`countB`**
  - Kann nur einmal pro Takt aktualisiert werden

- Echter 2-Port Betrieb benötigt andere **Art** von Zustandselement
  - Muss mehrere Schreib/Leseoperationen je Takt zulassen
  - Problem: Welche Schreibwerte sieht man beim Lesen?

  → CReg (ehemals EHR), definiert Schreib/Lese-Sichtbarkeiten exakt

Semantik des 2-Port-Betriebs

Nachdenken vor Festlegen von Implementierungsdetails!

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn **`countA`** und **`countB`** beide im selben Takt feuern …
  - … was soll Ergebnis des Zählers sein?
  - … welchen Rückgabewert des Zählerstands sollen die Methoden liefern?
- Kein einzelnes offensichtlich korrektes Verhalten
  - Entwickler muß ad-hoc Festlegung treffen
- Gleiche Vorgehensweise wie bei RTL Entwurf in Verilog/VHDL
  - Dort muß Festlegung aber …
  - … manuell implementiert werden (i.d.R. keine Unterstützung durch Sprache)
  - … klar dokumentiert werden
  - … ggf. in weiteren Richtlinien für die Benutzung der Hardware beachtet werden
- In der Praxis geht oftmals wenigstens einer dieser Punkte schief …

Semantik des 2-Port-Betriebs

Unterstützung bei der Semantikdefinition durch Sprache

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nebenläufigkeit ist direkter Bestandteil der Modellierung in Bluespec
  - Präzedenzrelation zwischen Methoden beeinflusst Reihenfolge
    - Auch innerhalb eines Taktes
  - Kann für automatische Überprüfungen bei Compilierung verwendet werden

- Einfache Beschreibung von Präzedenzrelationen durch CRegs

Verhalten eines CRegs

TECHNISCHE UNIVERSITÄT DARMSTADT

- CReg bietet einen **Array** aus Reg-Schnitstellen
- Können untereinander **nebenläufig** betrieben werden

```
module mkCReg#(parameter Integer n,
               parameter a_type resetval)
              (Reg#(a_type) ifc[])
  provisos (Bits#(a_type, sizea));
```

- **Präzedenzrelation**

```
Reg#(Bool) ports[N] <- mkCReg(N, False);
```

ports [0]._read < ports [0]._write <

ports[1]._read < ports[1]._write <

ports[2]._read < ports[2]._write <

… …

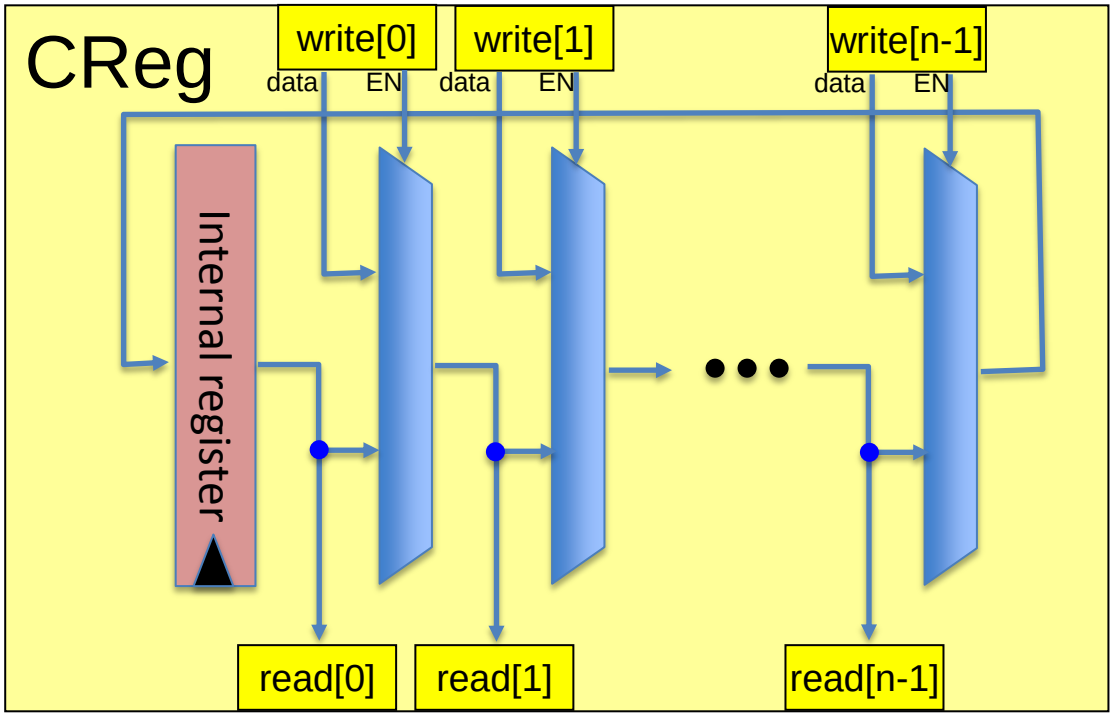
ports[N-1]._read < ports[N-1]._write

Übliche Präzedenzrelation zwischen _read/_write auf gleichen Port

*Lesen von Ports mit höherer Nummer sieht Schreiben auf Ports mit niedrigerer Nummer **noch im selben Takt***

Gesehen wird der Wert geschrieben von Schreiboperation mit größter Portnummer echt kleiner als Lese-Portnummer

Mögliche CReg Implementierung



- Nur als **Beispiel** für Hardware-Realisierbarkeit
- Bluespec trennt
  - Logische Semantik (beschrieben durch Präzedenzrelation)
  - Implementierung (black box)

2. Versuch

Saturierender 2-Port Zähler mit CReg

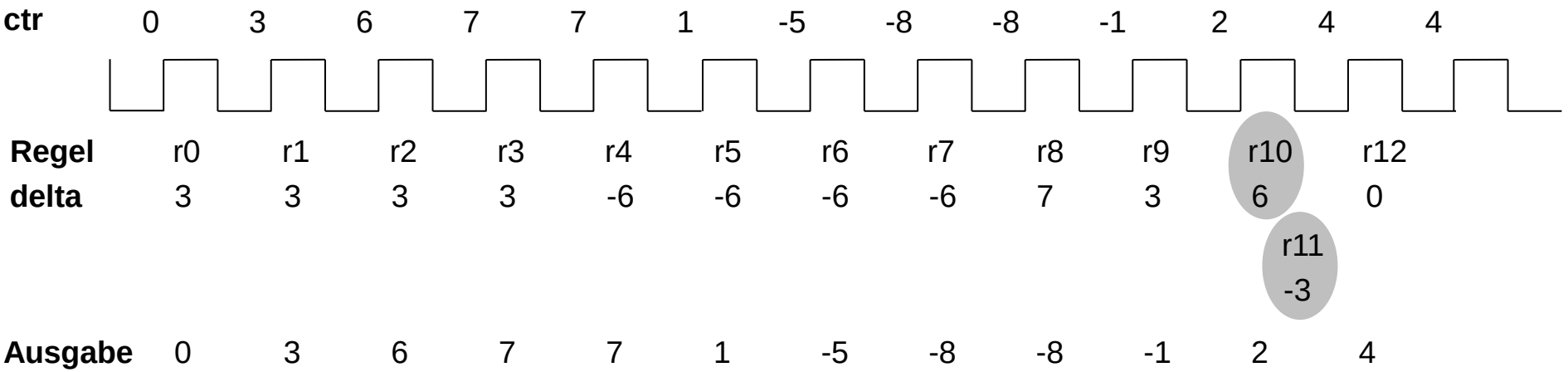
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
   Int#(4) ctr[2] <- mkCReg(2, 0);  // Lege CReg mit 2 Ports an

   function ActionValue #(Int #(4)) fn_count (Integer p, Int #(4) delta);
      actionvalue
         // Erhöhen der Wortbreite zum Vermeiden von Über/Unterlauf
         Int #(5) new_val = extend (ctr[p]) + extend (delta);
         if (new_val > 7) ctr[p] <= 7;
         else if (new_val < -8) ctr[p] <= -8;
         else ctr[p] <= truncate (new_val);

         return ctr[p];    // Beachte: gibt _alten_ Wert zurück
      endactionvalue
   endfunction

   method countA (Int #(4) delta) = fn_count (0, delta);
   method countB (Int #(4) delta) = fn_count (1, delta);
endmodule
```

- **ctr** ist nun 2-Port **CReg** anstatt **Reg**
- **fn_count** nun parametrisiert mit CReg Port-Nummer
- **countA** und **countB** benutzen verschiedene Ports in CReg
  - Port 0 in **countA**, Port 1 in **countB** → **countA < countB**

2. Versuch

Ablauf und Simulationsausgabe



- **`r10`** und **`r11`** nun nebenläufig
  - Können im gleichen Takt ablaufen
  - Benutzen unterschiedliche Ports des CRegs
  - Port-Nummern definieren Präzedenzrelation
    - Erst **`r10`**: 2 + 6 = 8, saturiert = 7
    - Dann **`r11`**: 7 – 3 = 4

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 11, r11: is 7, count (-3)
cycle 12, r12: is 4, count (0)
```

gleicher Takt

2. Versuch

Beispiel für alternative Präzedenzrelation

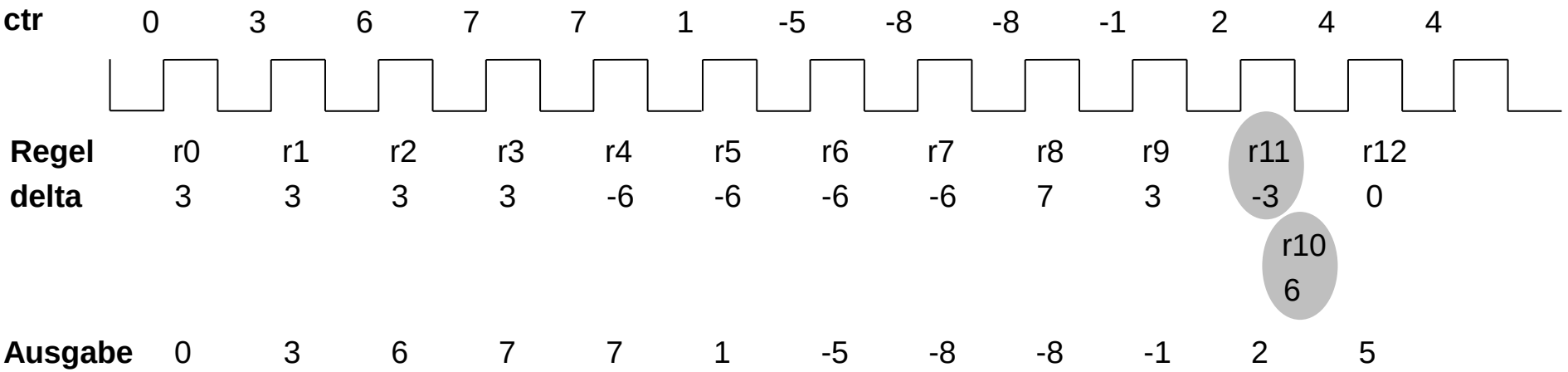
TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
   Int#(4) ctr[2] <- mkCReg(2, 0);    // Lege CReg mit 2 Ports an

   function ActionValue #(Int #(4)) fn_count (Integer p, Int #(4) delta);
      actionvalue
         // Erhöhen der Wortbreite zum Vermeiden von Über/Unterlauf
         Int #(5) new_val = extend (ctr[p]) + extend (delta);
         if (new_val > 7) ctr[p] <= 7;
         else if (new_val < -8) ctr[p] <= -8;
         else ctr[p] <= truncate (new_val);

         return ctr[p];    // Beachte: gibt _alten_ Wert zurück
      endactionvalue
   endfunction

   method countA (Int #(4) delta) = fn_count (1, delta);
   method countB (Int #(4) delta) = fn_count (0, delta);
endmodule
```

- Damit nun **countB < countA**

2. Versuch

Ablauf und Simulationsausgabe bei geänderter Präzedenz



- Nun **`r11`** nebenläufig vor **`r10`**
- Damit anderer Ablauf
  - Erst **`r11`**: 2 - 3 = -1
  - Dann **`r10`**: -1 + 6 = 5

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r11: is 2, count (-3)
cycle 11, r10: is -1, count (6)
cycle 12, r12: is 5, count (0)
```

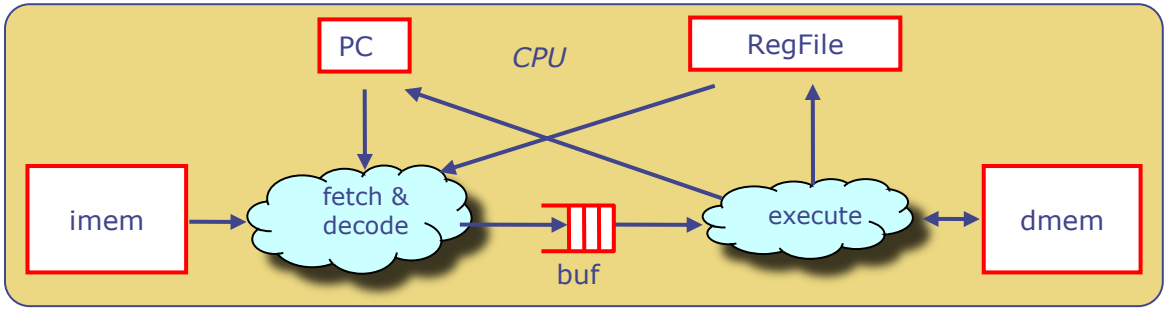
gleicher Takt

Beispiel II für CRegs

Einfache zweistufige Prozessor-Pipeline

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vereinfacht: **Kombiniert** Fetch/Decode und Execute/Memory/Writeback

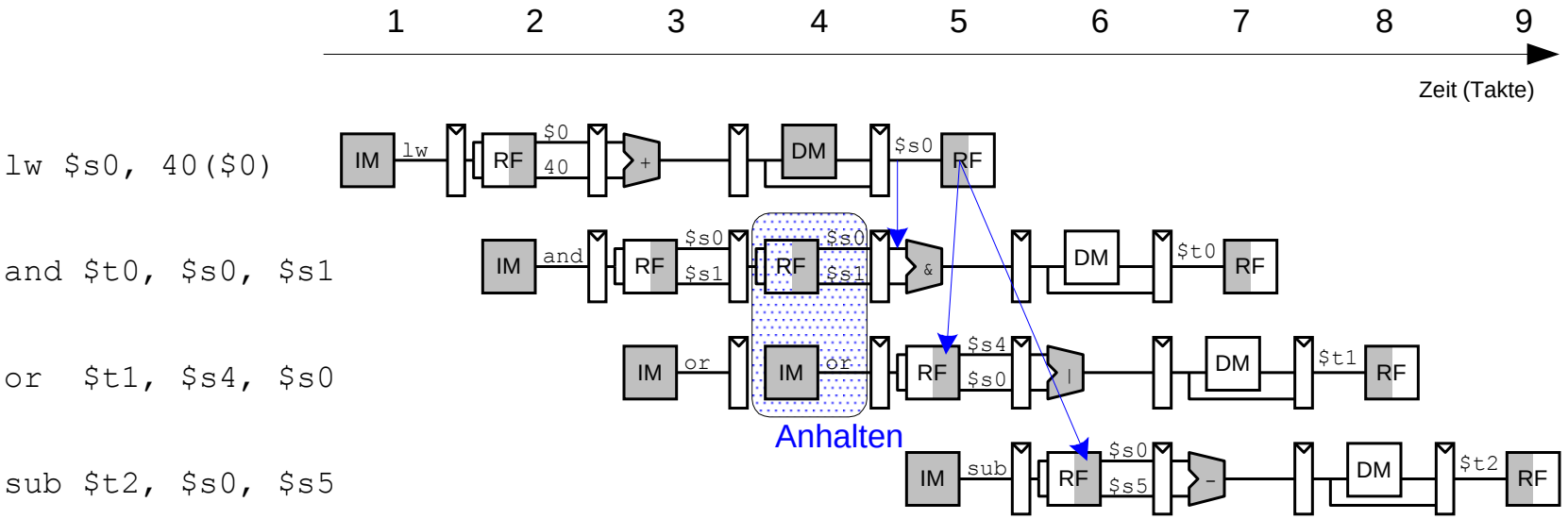


- Hier relevant: **Kommunikation** zwischen FD und EMW-Stufen

- Häufig realisiert als **1-elementige Warteschlange** (FIFO)
- Genauer: Pipeline Register mit Interlock
  - Interlock ist Gültigkeitsstatus (*valid bit*)
  - Kann z.B. FD anhalten (*stall*), wenn EMW noch beschäftigt ist
  - Kann EMW anhalten, wenn FD noch keinen neuen Befehl parat hat

TGDI: Beispiel für Stalling

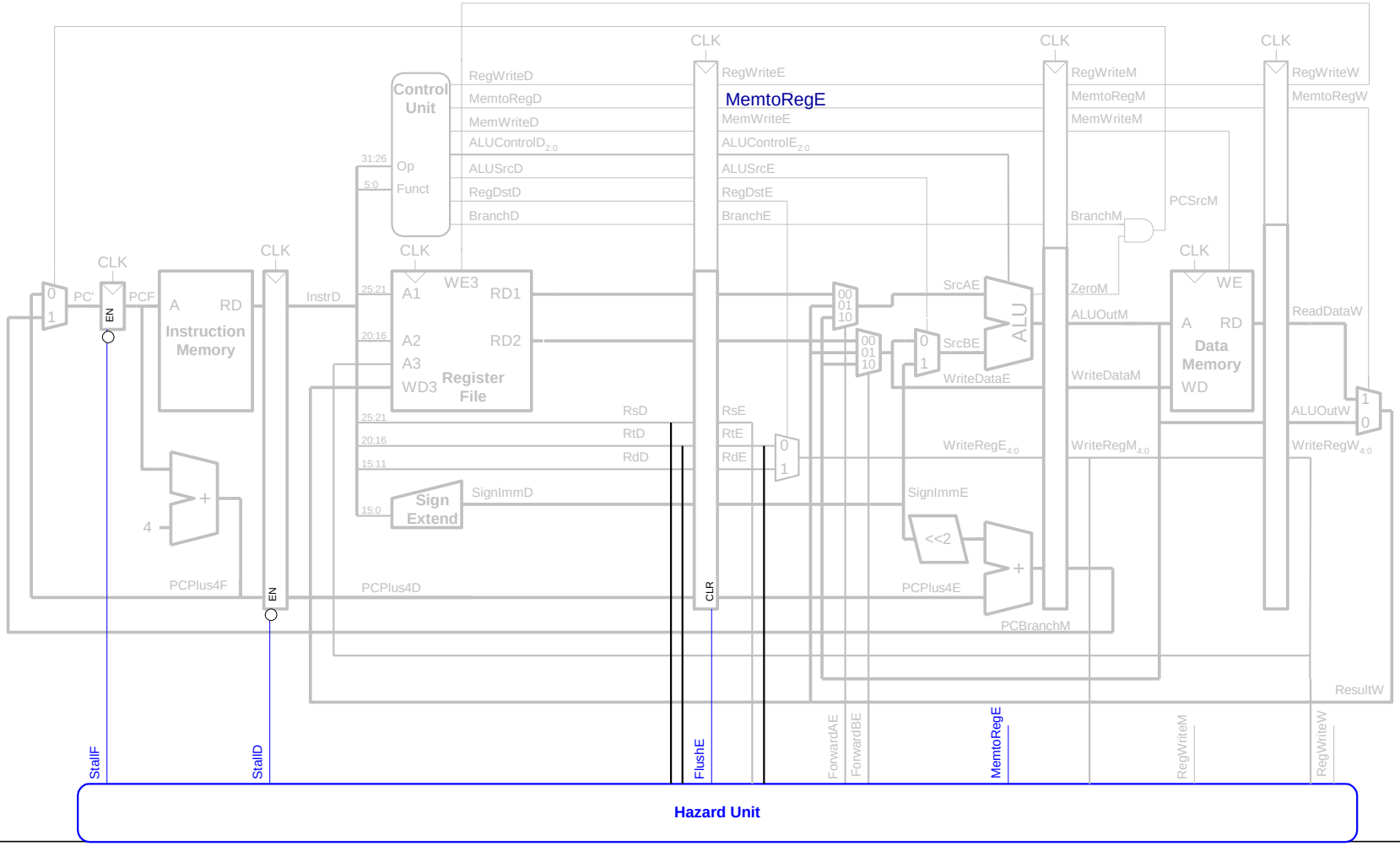
EMW ist noch beschäftigt, FD muß angehalten werden



TGDI: Realisierung im MIPS

Manipulation der CE-Eingänge der FD Pipeline-Register

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

Wie diese 1-FIFO aufbauen?

Aus normalen Registern?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkFIFOF1 (FIFOF #(t));
   Reg #(t)          rg       <- mkRegU;     // Datenhaltung
   Reg #(Bit #(1))   rg_count <- mkReg (0);  // Anzahl Elemente in FIFO (0 or 1)

   method Bool notEmpty = (rg_count == 1);   // Abfrage des Füllstandes
   method Bool notFull  = (rg_count == 0);   // -- "" --

   method Action enq (t x) if (rg_count == 0);  // neue Daten eintragen, wenn nicht voll
      rg <= x;
      rg_count <= 1;
   endmethod

   method t first ()  if (rg_count == 1);    // alten Wert lesen, wenn nicht leer
      return rg;
   endmethod

   method Action deq () if (rg_count == 1);  // alten Wert entfernen, wenn nicht leer
      rg_count <= 0;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

Wie diese 1-FIFO aufbauen?

Aus normalen Registern?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkFIFOF1 (FIFOF #(t));
   Reg #(t)          rg       <- mkRegU;     // Datenhaltung
   Reg #(Bit #(1))   rg_count <- mkReg (0);  // Anzahl Elemente in FIFO (0 or 1)

   method Bool notEmpty = (rg_count == 1);   // Abfrage des Füllstandes
   method Bool notFull  = (rg_count == 0);   // -- "" --

   method Action enq (t x) if (rg_count == 0);  // neue Daten eintragen, wenn nicht voll
      rg <= x;
      rg_count <= 1;
   endmethod

   method t first ()  if (rg_count == 1);    // alten Wert lesen, wenn nicht leer
      return rg;
   endmethod

   method Action deq () if (rg_count == 1);  // alten Wert enfernen, wenn nicht leer
      rg_count <= 0;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

- **Problem**: **`enq`** und {**`first,deq`**} niemals nebenläufig (**`rg_count`** == 0/1)
- FD und EMW niemals in gleichem Takt: Das ist keine Pipeline!

Anderer Ansatz erforderlich

Vor Implementierung über Semantik nachdenken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Häufig verwendete Verhalten
  - Zwischen Pipeline-Stufen (z.B. im MIPS zwischen F/D/E/M/W)
    - Werte können gleichzeitig gelesen und geschrieben werden
    - Lesen liefert **alten** Wert
  - Manchmal auch benötigt (im MIPS: Forwarding via Hazard Unit)
    - Werte können gleichzeitig gelesen und geschrieben werden
    - Lesen liefert bereits **neuen** Wert (z.B. von W direkt nach E)
- Damit nun präzise Formulierung als **Präzedenzrelation** zwischen Bluespec-Methoden möglich

Pipeline FIFO

TECHNISCHE UNIVERSITÄT DARMSTADT

PipelineFIFOs:

- Falls leer: Nur **`enq`** ist bereit
- Falls voll: **`enq`**, **`first`** und **`deq`** sind bereit mit: {**`first,deq`**} < **`enq`**
  d.h., falls alle Methoden bereit sind, wird logisch erst {**`first,deq`**} gefolgt von **`enq`** ausgeführt
  d.h., altes Datum aus FIFO wird erst gelesen, bevor neues eingetragen wird

Bypass FIFO

TECHNISCHE UNIVERSITÄT DARMSTADT

BypassFIFOs:

- Falls voll: nur {**`first,deq`**} sind bereit
- Falls leer: **`enq`**, **`first`** und **`deq`** sind bereit mit: **`enq`** < {**`first,deq`**}
  d.h., falls alle Methoden bereit sind, wird logisch erst **`enq`** gefolgt von {**`first,deq`**} ausgeführt
  d.h., ein neu eingetragener Wert wird sofort zu {**`first,deq`**} durchgeleitet (*bypassed*)

Implementierung mittels CRegs

Präzedenzrelation für Pipeline FIFO

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkPipelineFIFOF (FIFOF #(t));
   t       creg[3]       <- mkCReg(3, ?); // Datenhaltung
   Bit#(1) creg_count[3] <- mkCReg(3, 0); // Anzahl in FIFO (0…1)

   method Bool notEmpty = (creg_count[0] == 1);
   method Bool notFull  = (creg_count[1] == 0);

   method Action enq (t x) if (creg_count[1] == 0);
      creg[1]       <= x;
      creg_count[1] <= 1;
   endmethod

   method t first ()  if (creg_count[0] == 1);
      return creg[0];
   endmethod

   method Action deq () if (creg_count[0] == 1);
      creg_count[0] <= 0;
   endmethod

   method Action clear;
      creg_count[2] <= 0;
   endmethod
endmodule
```

Nur minimale Änderung relativ zu erstem Versuch

- **`notEmpty`**, **`first`** und **`deq`** benutzen CReg Port 0
- **`notFull`** und **`enq`** benutzen CReg Port 1
- **`clear`** benutzt CReg Port 2

Implementierung mittels CRegs

Präzedenzrelation für Bypass FIFO

TECHNISCHE UNIVERSITÄT DARMSTADT

```
module mkBypassFIFOF (FIFOF #(t));
   t       creg[3]       <- mkCReg(3, ?); // Datenhaltung
   Bit#(1) creg_count[3] <- mkCReg(3, 0); // Anzahl in FIFO (0…1)

   method Bool notEmpty = (creg_count[1] == 1);
   method Bool notFull  = (creg_count[0] == 0);

   method Action enq (t x) if (creg_count[0] == 0);
      creg[0]       <= x;
      creg_count[0] <= 1;
   endmethod

   method t first ()  if (creg_count[1] == 1);
      return creg[1];
   endmethod

   method Action deq () if (creg_count[1] == 1);
      creg_count[1] <= 0;
   endmethod

   method Action clear;
      creg_count[2] <= 0;
   endmethod
endmodule
```

Nur minimale Änderung relativ zu erstem Versuch

- **`notFull`** und **`enq`** benutzen CReg Port 0
- **`notEmpty`**, **`first`** und **`deq`** benutzen CReg Port 1
- **`clear`** benutzt CReg Port 2

Zusammenfassung CRegs

TECHNISCHE UNIVERSITÄT DARMSTADT

- Primitive für **kontrolliert nebenläufige** Ausführung
  - Mehrere Methoden können innerhalb eines Taktes ausgeführt werden
  - Wohldefinierte logische Ausführungsreihenfolge
- Verwende CRegs, um **nebenläufige Ausführung** von Regeln zu erreichen
  - Als erstes benötigte Semantik definieren
  - Erst danach mittels CRegs implementieren
- Korrektheit: CRegs funktionieren mit **beliebigen** Ablaufplänen
  - Falls maximal eine Regel je Takt ausgeführt wird: CReg == Reg
- In der Praxis: **Vorgefertigte** Elemente aus Bibliothek verwenden
  - PipelineFIFO, BypassFIFO, …
  - CRegs nur benutzen, um noch nicht vorhandene Funktionalität zu realisieren
    - Vorsicht: Viele Ports führen in der Regel zu langen kombinatorische Pfaden!

BEEINFLUSSEN DER ABLAUFPLANUNG

Ablaufplanung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundlage der Ausführungsreihenfolge ist **Ablaufplan**
- Einmal festgelegte **Reihenfolge** von Regeln: rA rB rC … rZ
- Falls Regeln ausgeführt werden
  - … werden Sie immer in dieser Reihenfolge ausgeführt
- Regeln müssen aber **nicht immer** ausgeführt werden
  - Regeln werden unterdrückt, um Konflikte zu vermeiden
- **Statisch**: Schon zur Compile-Zeit
  - Regeln werden dauerhaft an Ausführung gehindert
- **Dynamisch**: Hier Prüfung zur Laufzeit
  - Regeln werden nur unter bestimmten Umständen an Ausführung gehindert

Ablaufplanung 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei *N* Regeln: *N*! **verschiedene** Ablaufpläne
- `bsc` wählt einen mit der **maximalen Nebenläufigkeit** aus
  - Vermeidet Konflikte
- Entwickler kann **Einfluss** auf die Auswahl von `bsc` nehmen
- BSV **Attribute** zur Ablaufplanung

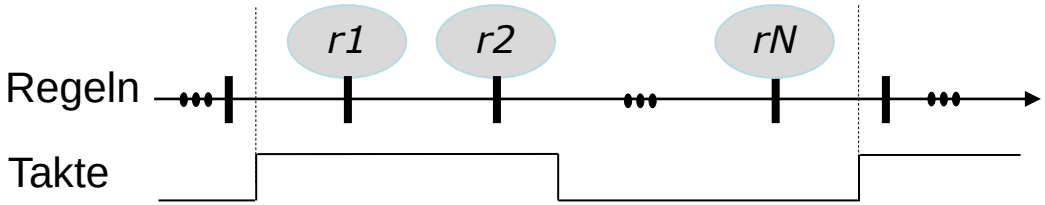
  (attribute = "Regel- und Methodennamen" *)*
  - Syntax wie in SystemVerilog
  - Stehen üblicherweise in einem Modul genau vor den betroffenen Regeln
  - Können aber auch Methoden betreffen
    - Methoden werden in diesem Zusammenhang als Fragmente von Regeln interpretiert
    - Sichtweise: Methodenkörper als in aufrufende Regeln einkopiert betrachten (*inlining*)

Verfeinerung der Nebenläufigkeit

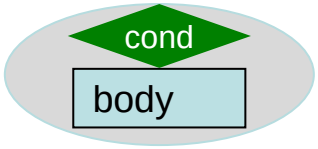
Gentrennte Betrachtung von Regelbedingung und -körper

TECHNISCHE UNIVERSITÄT DARMSTADT

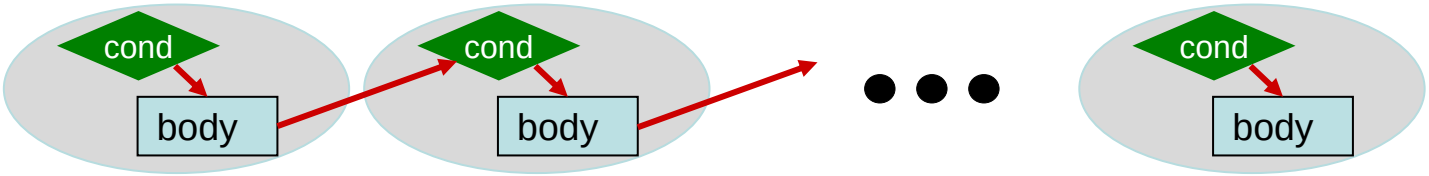
- Nebenläufige Ausführung von Regeln



- Genauer betrachtet:
  - Auswertung der **Regelbedingung** *cond*
  - Auswertung des **Regelköpers** *body*



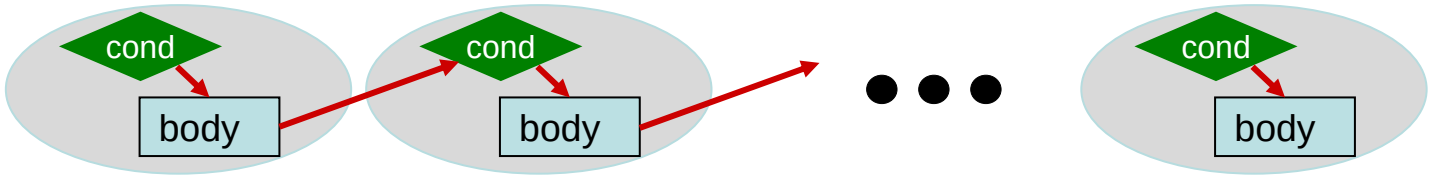
- Damit **verschränkter** Ablauf



Embedded Systems & Applications

Getrennte Ausführung

Dringlichkeit und Frühzeitigkeit



- Regelbedingungen r*N*.cond sind **boole'sche Ausdrücke**
  - Ohne Seiteneffekte
  - Auswertung von r*A*.cond hat keinen Einfluss auf r*B*.cond und r*B*.body
- Häufig hat r*A*.body auch keinen Einfluss auf r*B*.cond
- Damit möglich
  - Umsortieren der Auswertungen von .cond und .body
    - Solange tatsächliche Abhängigkeiten zwischen .body und .cond erhalten bleiben
- Terminologie
  - Dringlichkeit (*urgency*): Reihenfolge der .cond Auswertungen
  - Frühzeitigkeit (*earliness*): Reihenfolge der .body Auswertungen

Festlegen der Dringlichkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

- Reihenfolge/Priorität der Berechnung der **WILL_FIRE** Bedingungen

```
(* descending_urgency = "r1, r2" *)

rule r1 (c1);        // nur ein enq
   fifo.enq (e1);    // pro Takt
endrule

rule r2 (c2);        // nur ein enq
   fifo.enq (e2);    // pro Takt
endrule
```

- **r1** und **r2** in **Konflikt**
  - Nur ein **fifo.enq()** je Takt
- Falls **c1** und **c2** **nicht statisch** berechnet werden können
  - Erzeugung von Hardware zum Unterbinden der Ausführung der jeweils anderen Regel
  - **bsc** legt willkürlich Ablaufplan fest (nicht-deterministisch)
- Attribut **descending_urgency** **bestimmt** Reihenfolge der .cond Prüfungen
  - Ablaufplanung nun komplett deterministisch

Festlegen der Frühzeitigkeit

```
(* execution_order = "r1, r2" * )

rule r1;
   x <= 5;
endrule

rule r2;
   y <= 6;
endrule
```

- Legt **logische Ausführungsreihenfolge** der Regelköper fest
- Im Beispiel wird `r1 < r2` festgelegt
- Angaben zur Dringlichkeit wären hier **sinnlos**
  - Beide Regeln sind immer bereit
  - Aktionen im Körper stehen nicht in Konflikt

Dringlichkeit ≠ Frühzeitigkeit 1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
(* descending_urgency="enq_item, enq_bubble" *)
rule enq_item;
  outfifo.enq(infifo.first); infifo.deq;
  bubbles <= 0;
endrule

rule enq_bubble;
  outfifo.enq(dummy_value);
  max_bubbles <= max (max_bubbles, bubbles);
endrule

rule inc_bubbles;
  bubbles <= bubbles + 1;
endrule
```

- **Übertrage** Datum von **`infifo`** nach **`outfifo`**
  - Falls Datum verfügbar, sonst übertrage Leerwert **`dummy_value`**
  - Zähle grösste Anzahl von aufeinanderfolgenden Leerwerten
    - Sogenannte Blasen (*bubbles*) in Pipeline

Dringlichkeit ≠ Frühzeitigkeit 2

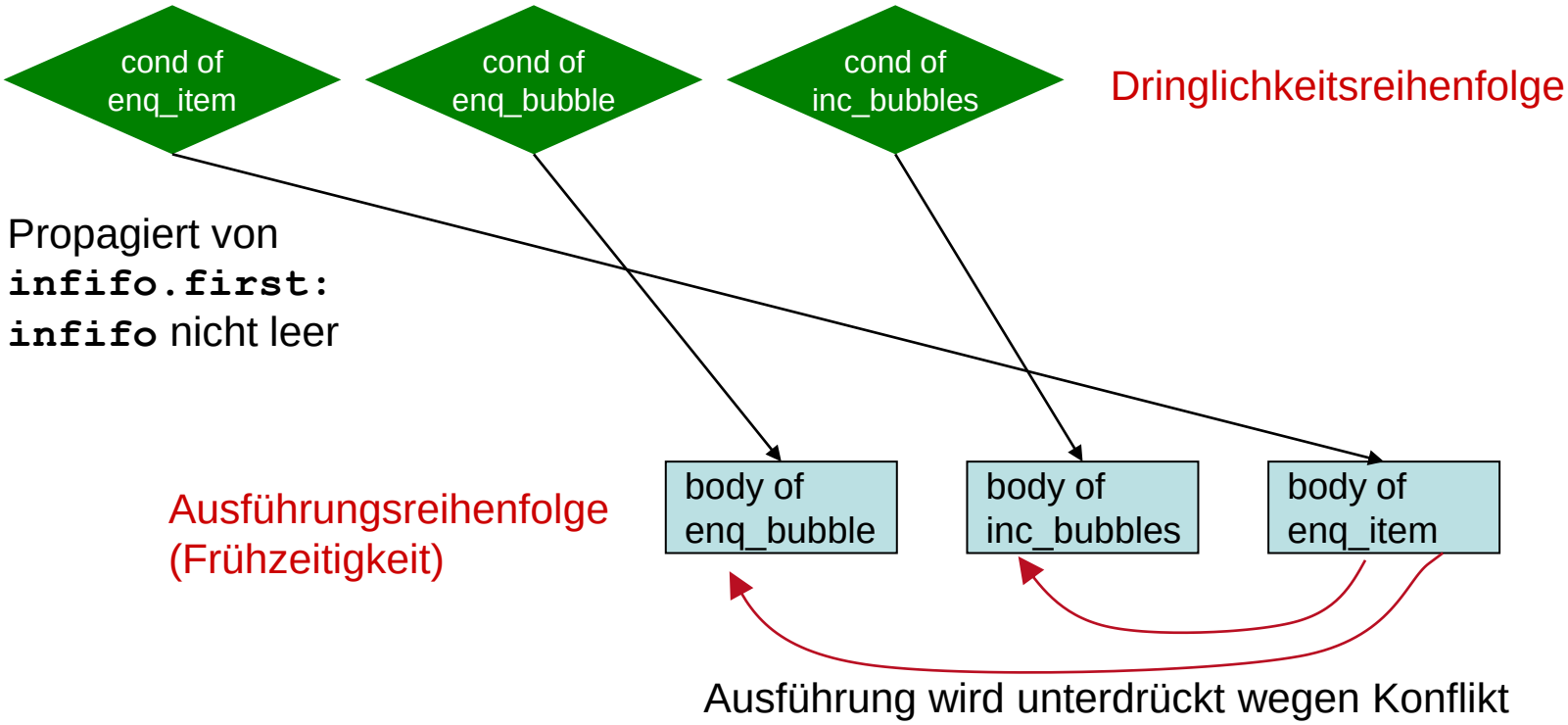
TECHNISCHE UNIVERSITÄT DARMSTADT

```
(* descending_urgency="enq_item, enq_bubble" *)
rule enq_item;
  outfifo.enq(infifo.first); infifo.deq;
  bubbles <= 0;
endrule

rule enq_bubble;
  outfifo.enq(dummy_value);
  max_bubbles <= max (max_bubbles, bubbles);
endrule

rule inc_bubbles;
  bubbles <= bubbles + 1;
endrule
```

- **Ausführungsreihenfolge** (Frühzeitigkeit)
  - **`enq_bubble < inc_bubbles < enq_item`**
  - Lesen von **`bubble`** muss vor Schreiben von **`bubble`** liegen (**`_read < _write`**)
- **Dringlichkeit** aber festgelegt auf: **`enq_item < enq_bubble`**
  - Falls neue Daten vorliegen, diese einreihen (und keine Bubbles!)

Dringlichkeit ≠ Frühzeitigkeit 3

Graphische Darstellung



Bevorrechtigung von Regeln

Preemption

TECHNISCHE UNIVERSITÄT DARMSTADT

```
(* preempts = "r1, r2" *)

rule r1 (upA);
   x <= x + 3;
endrule

rule r2;
   y <= y + 1;
endrule
```

- Erlaubt einer gefeuerten Regel
  - … das Feuern einer anderen Regel zu unterdrücken
  - Auch, wenn kein Konflikt vorliegt

- Beispiel: Falls **`r1`** feuert (**`upA == TRUE`**)
  - … wird **`r2`** an Ausführung gehindert
  - Obwohl **`r2`**.cond immer **`TRUE`** ist
  - Effekt hier: **`r2`** zählt Leerzyklen von **`r1`** (in denen **`r1`** nicht feuert)

Sich wechselseitig auschließende Regeln 1 *mutual exclusion*

TECHNISCHE UNIVERSITÄT DARMSTADT

```
(* mutually_exclusive = "updateBit0, updateBit1" *)

rule updateBit0 (oneHotNumber[0] == 1);
  x[0] <= 1;
endrule

rule updateBit1 (oneHotNumber[1] == 1);
  x[1] <= 1;
endrule
```

- **Zusicherung** an Compiler
  - … dass zwei Regelbedingungen niemals gleichzeitig wahr sind
- Compiler bemüht sich zwar, das automatisch zu ermitteln
- Ist aber im allgemeinen Fall nicht entscheidbar, Beispiele:
  - Externe Schaltungseingänge tauchen in Bedingung auf
  - Wechselseitiger Ausschluss basiert auf anwendungsspezifischem Wissen
    - Z.B. *one-hot* Kodierung von Signalen (→ TGDI)

Sich wechselseitig auschließende Regeln 2 *mutual exclusion*

TECHNISCHE UNIVERSITÄT DARMSTADT

```
(* mutually_exclusive = "updateBit0, updateBit1" *)

rule updateBit0 (oneHotNumber[0] == 1);
  x[0] <= 1;
endrule

rule updateBit1 (oneHotNumber[1] == 1);
  x[1] <= 1;
endrule
```

- Zusicherung wird genutzt …
  - … um effizientere Hardware zu erzeugen
    - Einfache Multiplexer statt Prioritätsmultiplexer
  - … um gegenseitigen Ausschluss aktiv während der Simulation zu überwachen
    - Ausgabe von Fehlermeldung, falls Zusicherung als verletzt erkannt wird

- Mehr dazu: BSV-by-Example, Kapitel 7

VON BSV ZU VERILOG

Zusammenhang BSV-Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

- Aus Sicht von Bluespec verhält sich
  - Verilog zu BSV wie
  - Assembler zu C/C++, Java
- Für ein umfassendes Verständnis ist es hilfreich, die Art der Abbildung
  - von der hohen Ebene (BSV, C/++, Java)
  - auf die niedrige Ebene (Verilog, Assembler)

  zu kennen

Module in Verilog und BSV

Gemeinsamkeiten und Unterschiede

TECHNISCHE UNIVERSITÄT DARMSTADT

Verilog Parameter sind üblicherweise skalare Zahlen.
Verilog Schnittstellen sind Listen von Ports für Signale.

BSV Parameter können beliebige Typen haben
(einschl. Funktionen, Interfaces, Module, …)
BSV Schnittstellen sind Interface-Typen
(definieren Methoden zur Interaktion mit Modul)

```
module m #(params) (ports)

  input …
  output …
  wire …

  reg x;
  reg y;

  module m1 #(params) p (port connections);
  module m1 #(params) q (port connections);
  module m2 #(params) r (port connections);

  assign w = 10 + wire from instance q
  assign …

  always @(posedge clk) …

  always @(posedge clk) …

endmodule
```

wire Deklarationen
Einziger Typ ist 'bits'

'reg' ist kein Modul.
'reg' ist mglw. kein Register.
'reg' enthält nur Bits.

Modul-instanziierung

"Verhalten"

```
module m #(params) (interface type);

  Reg #(t1) x <- mkReg (0);
  Reg #(t2) y <- mkReg (12);

  Ifc_m1 p <- mkM1a (params);
  Ifc_m1 q <- mkM1b (params);
  Ifc_m2 r <- mkM2 (params);

  int w = 10 + q.method();

endmodule
```

Register sind Module und werden instanziiert und typgeprüft

Regeln

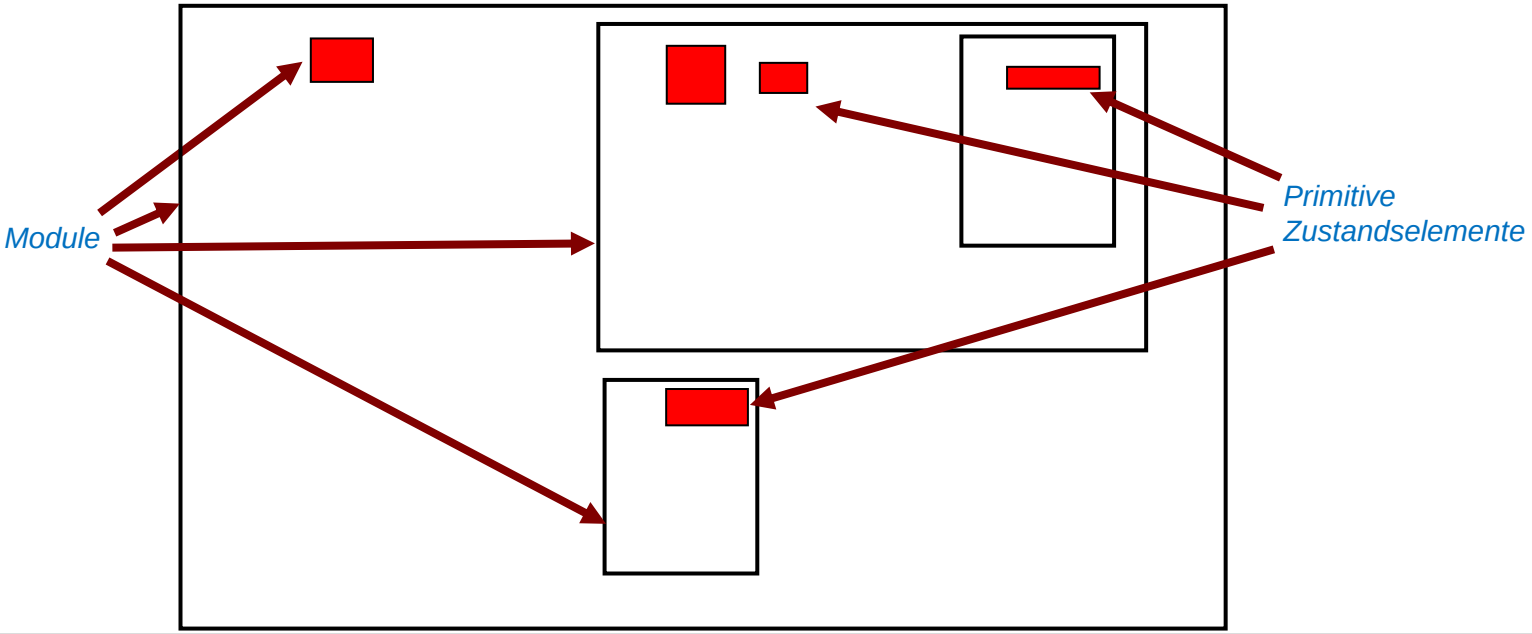
Methoden

Typisierte Variablen-deklarationen

Embedded Systems & Applications

Modulhierarchie und Zustand

- Identisch in BSV und Verilog (und SystemVerilog, SystemC und VHDL)
- Blätter der Modulhierarchie sind primitive Zustandselemente
  - Register, FIFOs, …
  - Neue Primitive können leicht in Verilog definiert und nach BSV importiert werden



Regeln und Schnittstellenmethoden

- Module bieten Schnittstellen bestehend aus Methoden an
- Module enthalten Regeln, die Methoden anderer Instanzen aufrufen
  - Einzige Möglichkeit für Inter-Instanz-Kommunikation (ähnl. OO-Sprachen)
- Methoden können Methoden anderer Instanzen aufrufen

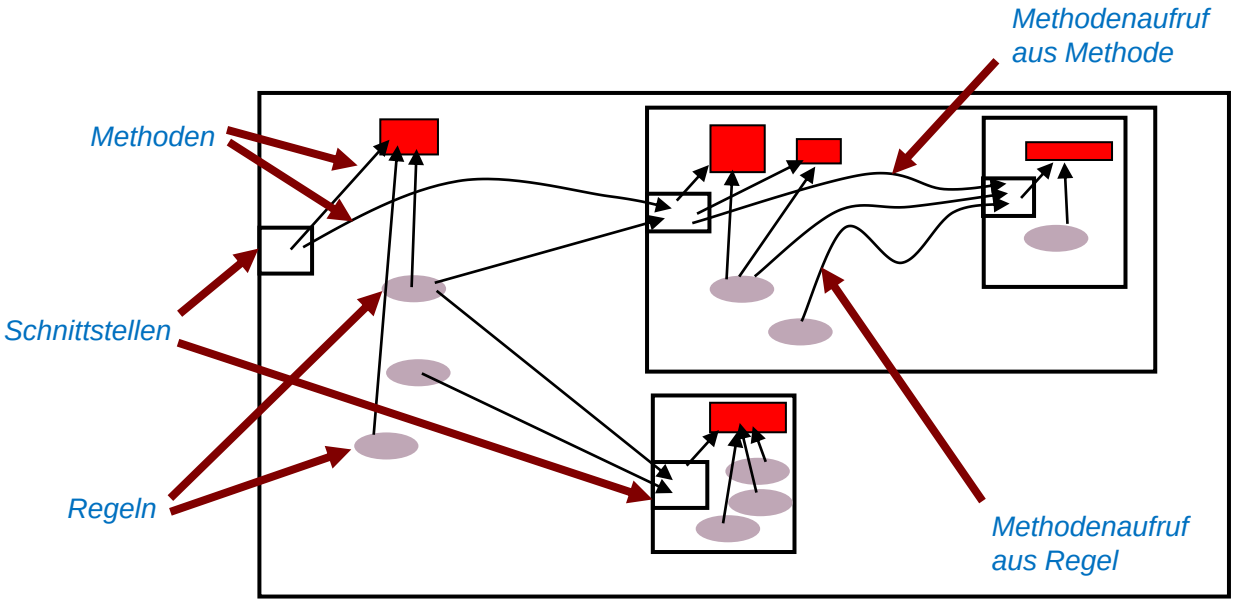


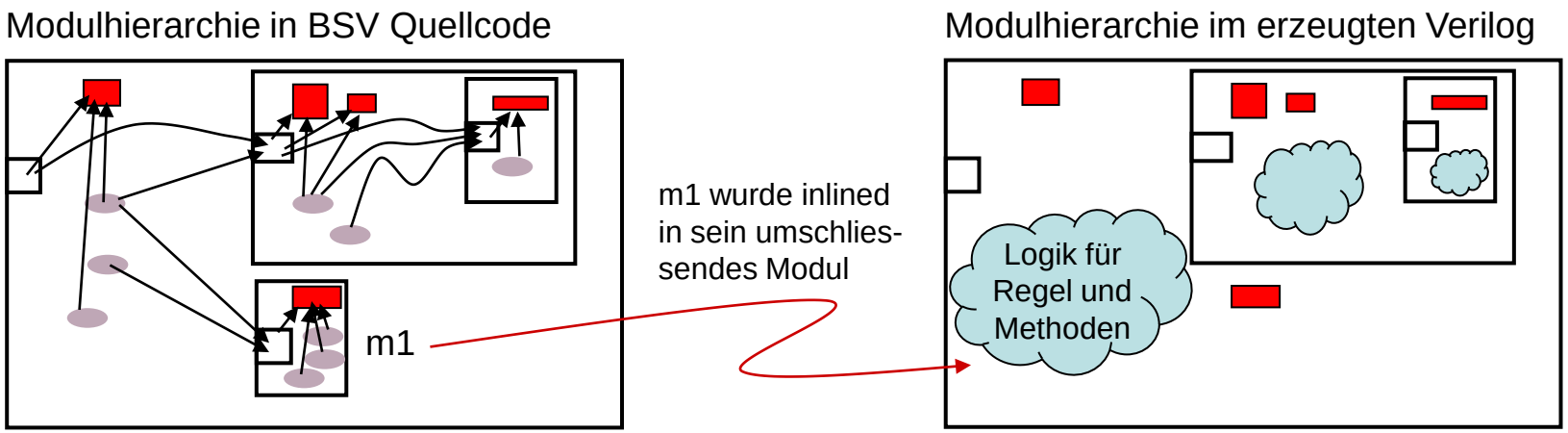
Abbildung der Modulhierarchie 1

Erhalten ./. Auflösen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Grundsätzlich**
  - BSV Hierarchie kann im erzeugten Verilog erhalten werden
    - BSV-Modul **`mkM`** wird zu Verilog-Modul **`mkM`** (in Datei **`mkM.v`**)
  - Wenn BSV-Modul **`mkM1`** ein BSV-Modul **`mkM2`** instanziiert
    - Instanziiert Verilog-Modul **`mkM1`** auch ein Verilog-Modul **`mkM2`**

- **Aber**: Aus Effizienzgründen werden BSV-Module im Verilog oftmals aufgelöst (*inlined*)



Embedded Systems & Applications

Abbildung der Modulhierarchie 2

Beeinflussen mittels des Attributs `synthesize`

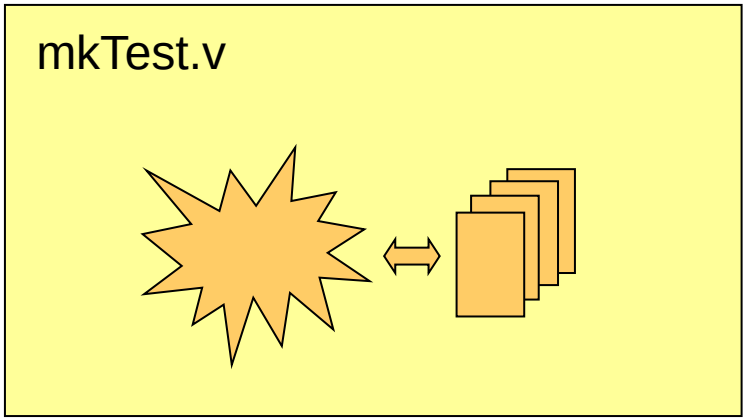
TECHNISCHE UNIVERSITÄT DARMSTADT

BSV Quellcode

```
(* synthesize *)
module mkTest (Empty);
   Mult_ifc m <- mkMult;
   …
endmodule: mkTest
```

```
module mkMult (Mult_ifc);
  …
endmodule: mkMult
```

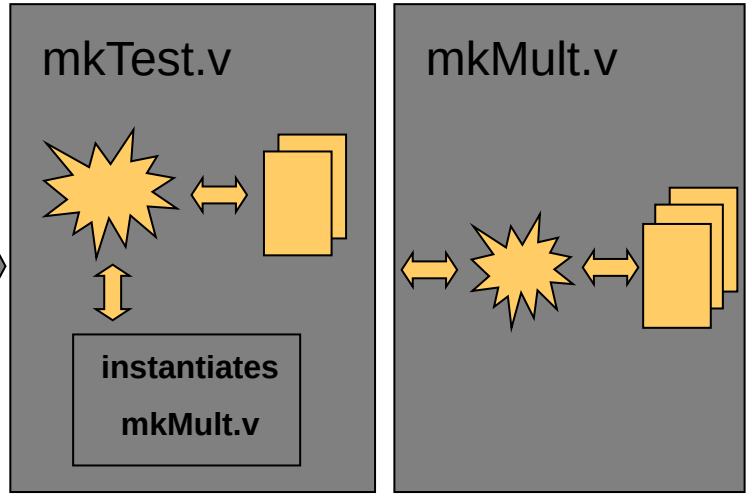
Erzeugtes Verilog



Nur **`mkTest`** markiert. Nur **`mkTest`** bleibt erhalten. Untermodule werden inlined.

```
(* synthesize *)
module mkTest (Empty);
   Mult_ifc m <- mkMult;
   …
endmodule: mkTest
```

```
(* synthesize *)
module mkMult (Mult_ifc);
  …
endmodule: mkMult
```



`mkTest` und **`mkMult`** markiert. Beide Module bleiben erhalten.

Gleicher Effekt: Auf `bsc`-Kommandozeile

```
-g mkTest –g mkMult
```


Embedded Systems & Applications

Einschränkungen von `synthesize`

- **`synthesize`** markiert die Grenzen von separaten Verilog-Modulen
- Darf nur auftauchen genau vor **`module mkFoo (…)`** Kopfzeile

- Darf nur vor bestimmten Modulen auftauchen
  - Da Verilog weniger mächtig ist als BSV
  - Schnittstelle nach aussen darf nur bestehen aus Bits, Skalaren und Bit-Vektoren
  - Anderes kann in Verilog nicht dargestellt werden!

- Aber: Beliebige Schnittstellen zwischen den inlined BSV-Modulen innerhalb der Verilog-Module
  - Der gesamte Sprachumfang von BSV kann in Hardware abgebildet werden

- Einschränkung gilt nur für separat nach Verilog kompilierte BSV-Module

Von BSV Schnittstellen zu Verilog Ports

TECHNISCHE UNIVERSITÄT DARMSTADT

- Interface-Methoden werden auf Verilog Ports abgebildet
- Formale Methodenparameter → **`input`** Ports
- Methodenergebnisse → **`output`** Ports
- Ausführungsbereitschaft einer Methode → **`output`** Port namens RDY_xxx
  - RDY_xxx == TRUE: Methode ist bereit (Bedingung ist erfüllt)
- Ausführen von Action und ActionValue-Methoden → **`input`** Port namens EN_xxx
  - EN_xxx == TRUE: Führe Aktionen in Methode aus

BSV Schnittstellen in Hardware

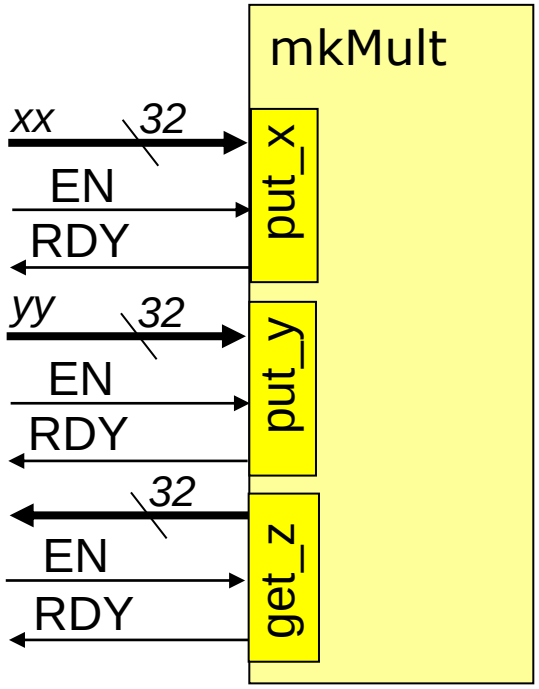
Beispiel 1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface Mult_ifc;
   method Action put_x (int xx);
   method Action put_y (int yy);
   method ActionValue #(int) get_z ();
endinterface: Mult_ifc
```

- RDY = Methode bereit (Bedingung wahr)
- EN = Aktionen in Methode ausführen
- Formale Parameter: separate **`input`** Ports
- Ergebnisse: separate **`output`** Ports

- Optimierung möglich
  - Eliminiere RDY, wenn Methode immer bereit
  - Eliminiere EN, wenn Methode jeden Takt ablaufen soll



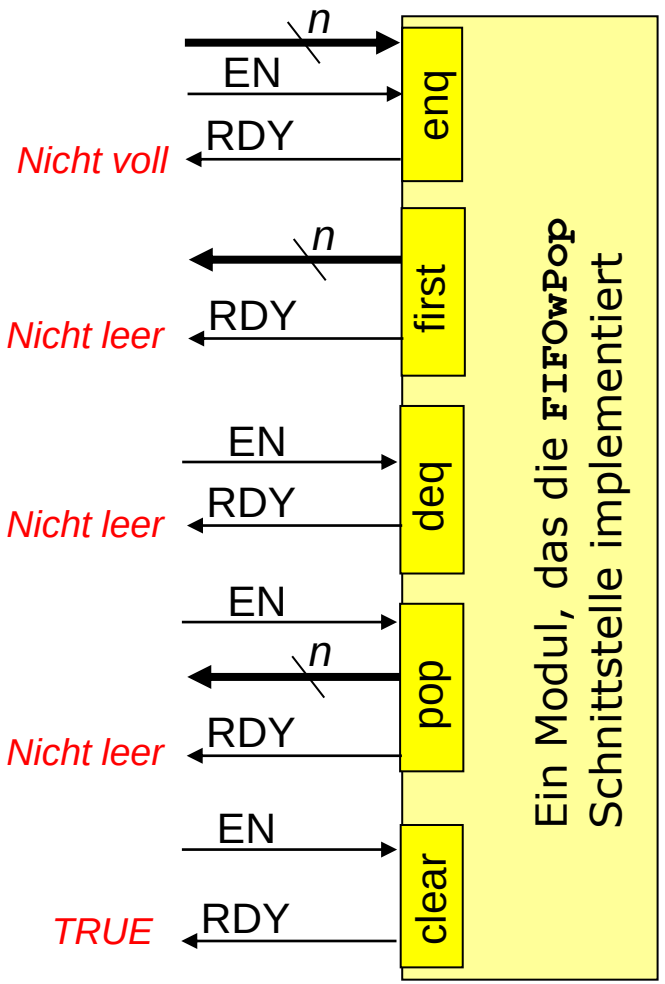
Embedded Systems & Applications

BSV Schnittstellen in Hardware

Beispiel 2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
interface FIFOwPop #(type t);
  method Action           enq (t x);
  method t                first;
  method Action           deq;
  method ActionValue#(t)  pop;
  method Action           clear;
endinterface
```



Embedded Systems & Applications

Gemeinsame Nutzung von Hardware

Zwei Regeln rufen Methode auf gleicher Instanz auf

TECHNISCHE UNIVERSITÄT DARMSTADT

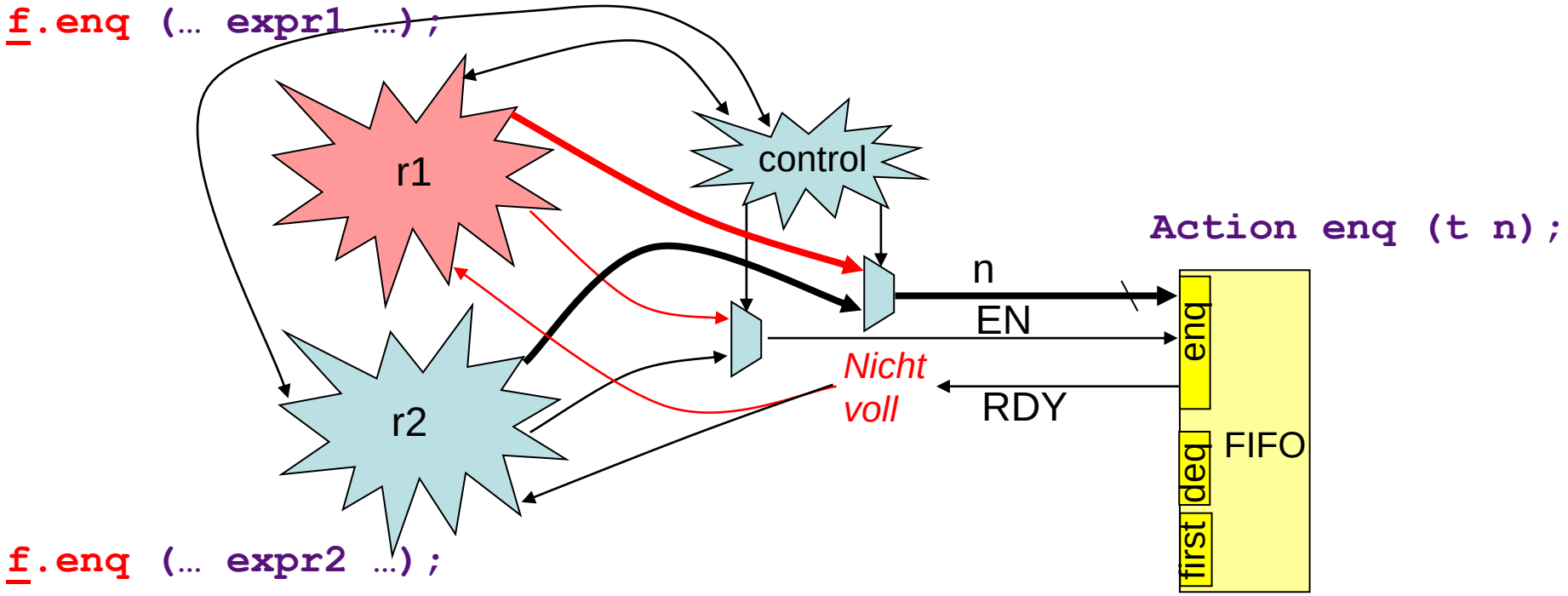
```
module mkTest (…);
  …
  FIFO#(int) f <- mkFIFO;
  …
  rule r1 (… cond1 …);
    …
    f.enq (… expr1 …);
    …
  endrule

  rule r2 (… cond2 …);
    …
    f.enq (… expr2 …);
    …
  endrule
endmodule: mkTest
```

```
interface FIFO#(type t);
  Action enq (t n);
  …
endinterface

module mkFIFO (…);
  …
  method enq(x) if (…notFull…);
    …
  endmethod
  …
endmodule: mkFIFO
```


Embedded Systems & Applications

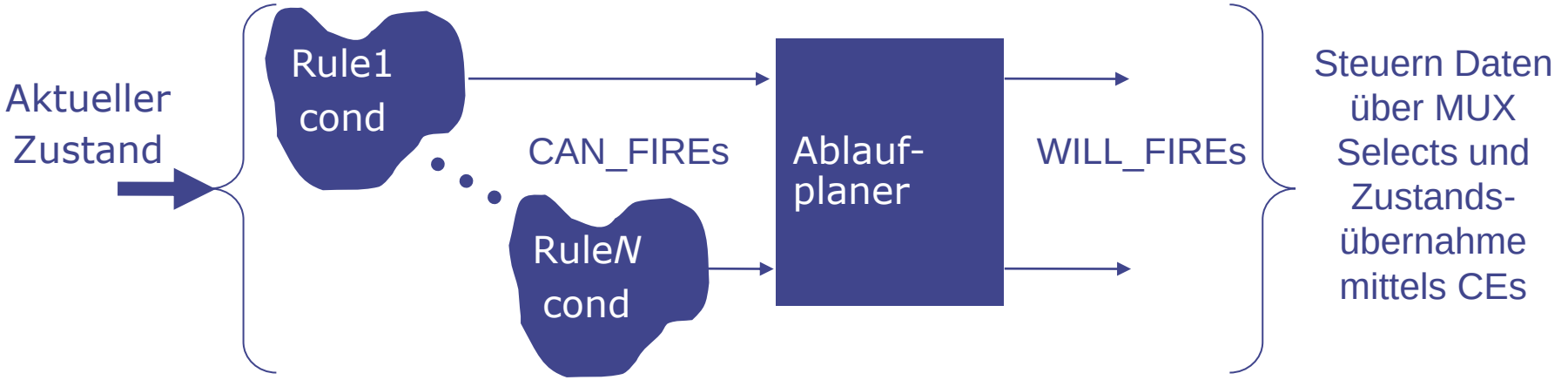
Gemeinsame Nutzung von Hardware

Automatisch erzeugte Logik



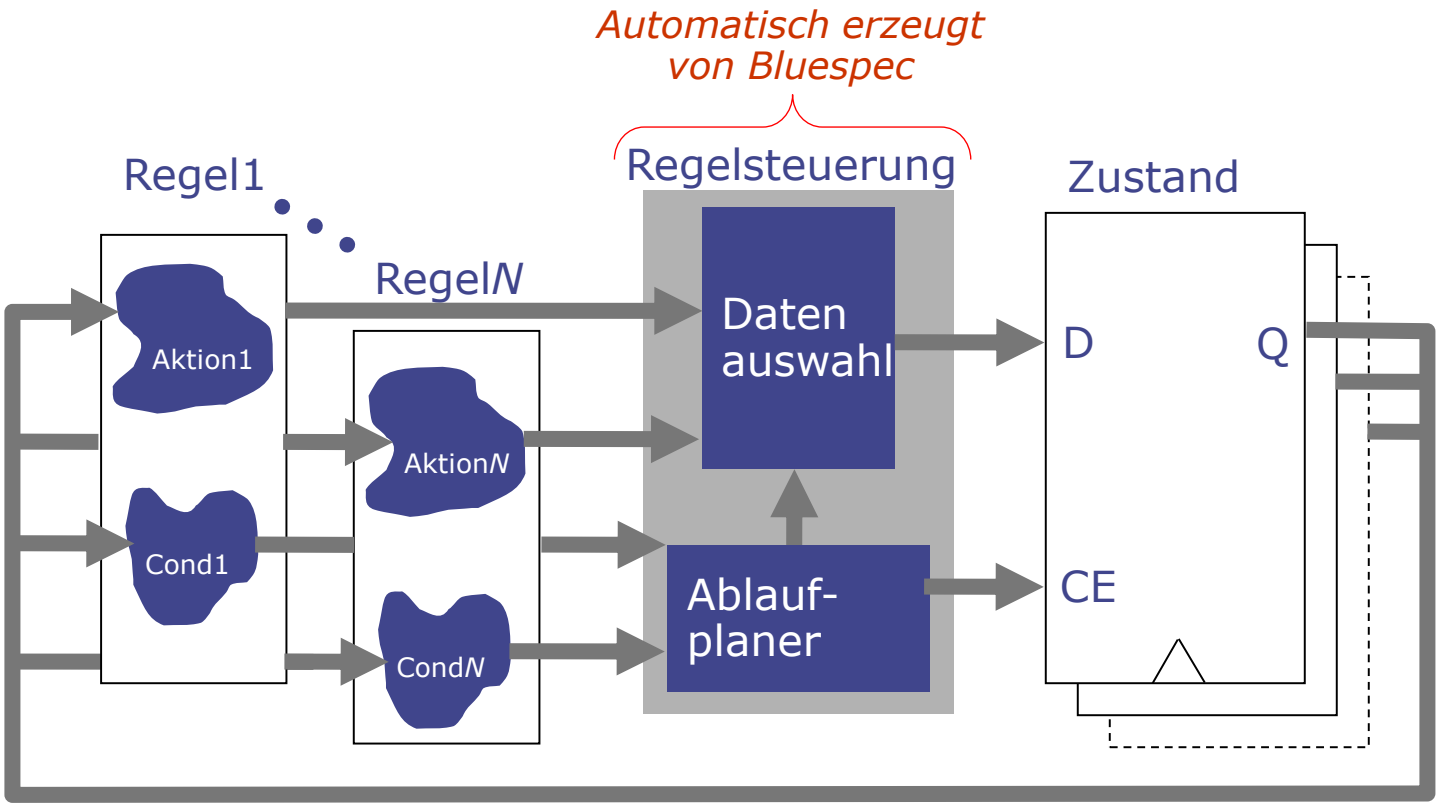
- Jeder **`input`** Port kann Ressource-Konflikt verursachen
  - Kann nur von einer Regel je Takt getrieben werden
- Folge: Nur Wertmethoden ohne formale Parameter können nie Ressource-Konflikte haben

CAN_FIRE und WILL_FIRE in Hardware



- **bsc** analysiert Konflikte und erzeugt dynamischen Ablaufplaner in HW
  - Als rein kombinatorische Schaltung, verteilt über mehrere Module
- CAN_FIRE Signal einer Regel gibt Bereitschaft an
  - Regelbedingung und Bedingungen von aufgerufenen Methoden (transitiv)
- WILL_FIRE Signal einer Regel löst Ausführung aus (feuern)
  - CAN_FIRE && (!WILL_FIRE aller vorhergehenden Regeln mit Konflikten zu dieser)

Übersicht über erzeugte HW



- Regelsteuerung in HDLs von Hand beschrieben
  - Oftmals fehlerbehaftet, hier *correct-by-construction* durch Bluespec-Semantik

Beispiel für Hardware-Erzeugung

Eingaberegeln

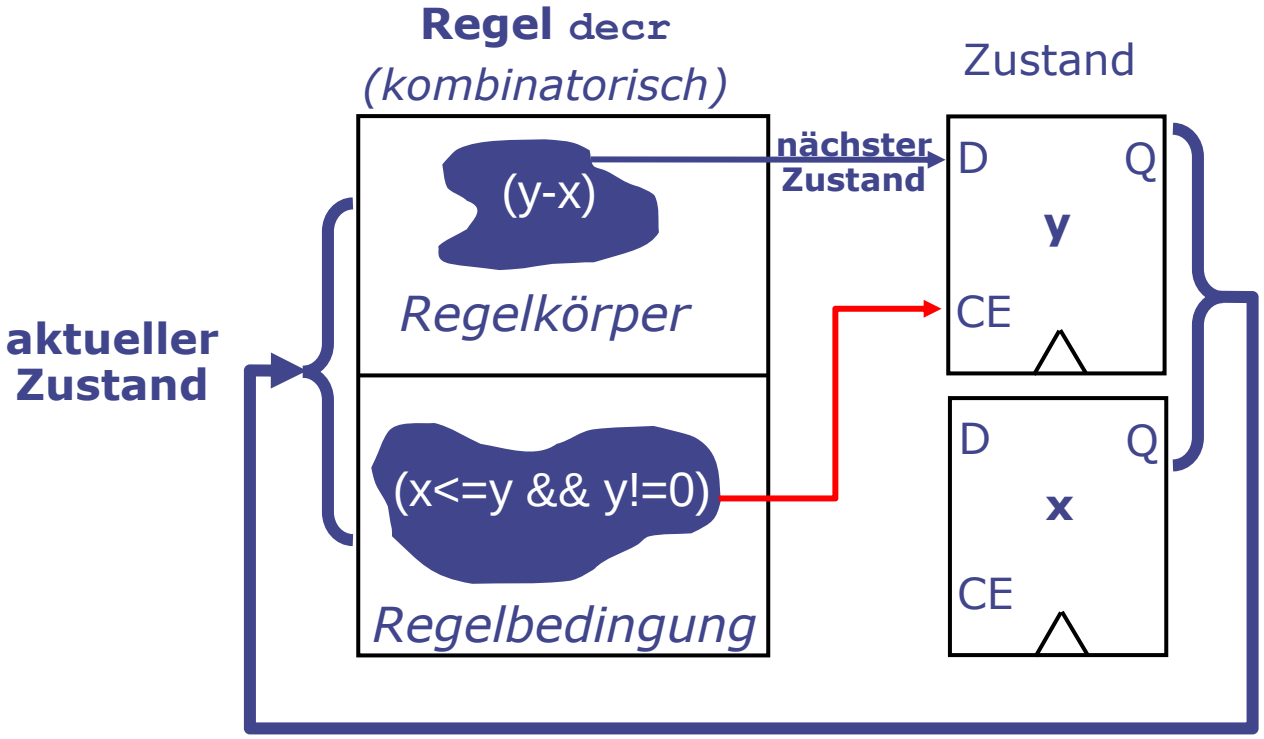
TECHNISCHE UNIVERSITÄT DARMSTADT

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr

rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```

- Frage am Rande: Was mögen diese Regeln berechnen?

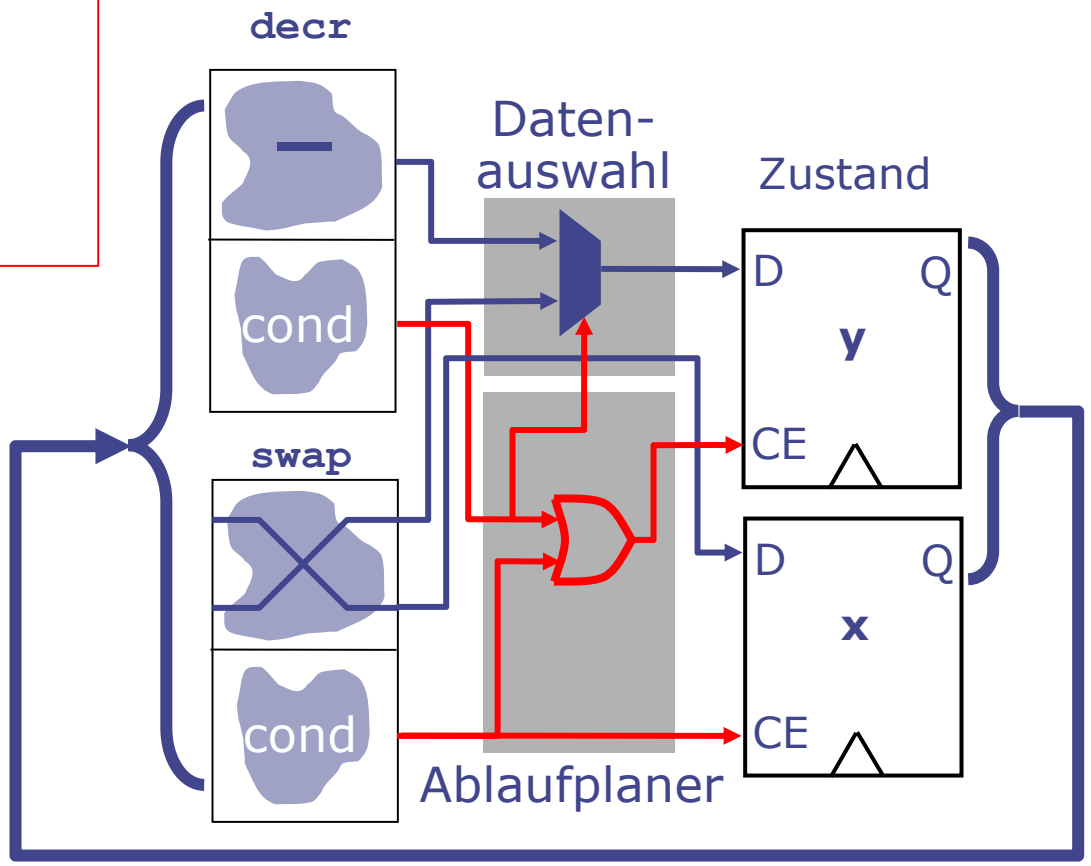
Hardware für eine der Regeln

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr
```



HW für sich ausschliessende Regeln

```
rule decr ( x <= y && y != 0 );
    y <= y - x;
endrule : decr

rule swap (x > y && y != 0);
    x <= y; y <= x;
endrule: swap
```



Clock- und Reset-Signale

TECHNISCHE UNIVERSITÄT DARMSTADT

- BSV-Beschreibungen haben bei uns nur einen Takt und einen Reset
  - Standard, wenn nicht explizit anders angegeben [machen wir nicht]
- Clock und Reset tauchen nicht im BSV Quellcode auf
- Werden im Verilog automatisch erzeugt
  - Jedes Modul hat einen CLK und RST_N Eingang
- Werden im Verilog-Testrahmen aus Bluespec Bibliothek getrieben
  - `$BLUESPECDIR/lib/Verilog/main.v`
- Können aber wenn nötig sehr fein konfiguriert werden
  - Positiv/Negativ, Sync/Async, Clock-Synchronizer, …
  - Machen wir alles nicht

Experimente mit BSV und Verilog

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anderer Aufruf von **`bsc`** zum Kompilieren: Statt **`-sim`** nun

```
bsc -verilog -g top -g dut … -u myfile.bsv
```

  - -g *Namen* von zu nach Verilog kompilierenden Modulen
    - Erzeugt Dateien *`top.v`* und *`dut.v`*, andere Bluespec-Module werden inlined

- Linken zum Erstellen des Simulationmodells aus Verilog nun mittels

```
bsc -verilog -e top
```

  - -e Namen des obersten Moduls für Simulation
    - Erzeugt bei uns mittels Icarus Verilog (iverilog) Simulationsmodell als Datei **`a.out`**

- Simulationsmodell ausführen durch

```
./a.out
```

Ende der Bluespec-Einführung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Alle grundlegenden Konzepte sind nun erklärt
- Weiterführende Konzepte
  - … werden bei Bedarf eingeführt (auch in den Übungen)
  - … oder können selbständig in der Dokumentation nachgelesen warden
    - Tipp: Typen und Typklassen können für komplexere Entwürfe sehr nützlich sein