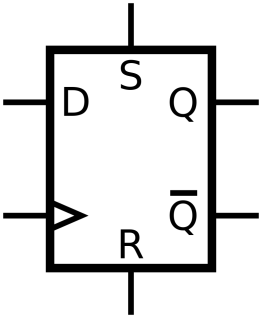
Architekturen und Entwurf von Rechnersystemen
Besprechung Theorieblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023

Johannes Wirth
Fachgebiet Eingebettete Systeme und ihre Anwendungen



Theorieübungsblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Register, Wires und CRegs
- Scheduling
  - Ausführungsreihenfolge
  - Dringlichkeit
- Feuerbereitschaft

Aufgabe 1.1.1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welches Verhalten von Logikelementen wird durch die Präzedenzrelation _read $<$ _write abgebildet?

Aufgabe 1.1.1

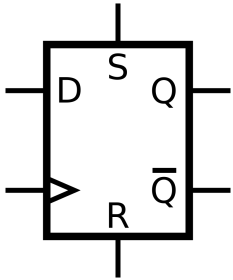
TECHNISCHE UNIVERSITÄT DARMSTADT

- Welches Verhalten von Logikelementen wird durch die Präzedenzrelation _read $<$ _write abgebildet?
- Erklärung zum Beispiel Anhand von D-Flip-Flop.

Aufgabe 1.1.1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welches Verhalten von Logikelementen wird durch die Präzedenzrelation _read $<$ _write abgebildet?
- Erklärung zum Beispiel Anhand von D-Flip-Flop.
- _read gibt den Wert von `Q` zurück. _write setzt zur nächsten Taktflanke einen neuen Wert an `D`.



Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wie Unterscheiden sich **`Reg`**, **`Wire`** und **`CReg`**?

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wie Unterscheiden sich **`Reg`**, **`Wire`** und **`CReg`**?
  - **Reg**: _read $<$ _write

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wie Unterscheiden sich **`Reg`**, **`Wire`** und **`CReg`**?
  - **`Reg`**: _read $<$ _write
  - **`CReg`**: _read[0] $<$ _write[0] $<$ _read[1] $<$ _write[1] . . .

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wie Unterscheiden sich **`Reg`**, **`Wire`** und **`CReg`**?
  - **`Reg`**: _read $<$ _write
  - **`CReg`**: _read[0] $<$ _write[0] $<$ _read[1] $<$ _write[1] . . .
  - **`Wire`**: _write $<$ _read

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wie Unterscheiden sich **`Reg`**, **`Wire`** und **`CReg`**?
  - **`Reg`**: _read $<$ _write
  - **`CReg`**: _read[0] $<$ _write[0] $<$ _read[1] $<$ _write[1] . . .
  - **`Wire`**: _write $<$ _read
- Gerne verschenkte Punkte...

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welche verschiedenen Wires sind in Bluespec verfügbar?

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welche verschiedenen Wires sind in Bluespec verfügbar?
  - **`mkWire`**: _write $<$ _read. Müssen aus verschiedenen Rules aufgerufen werden.

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welche verschiedenen Wires sind in Bluespec verfügbar?
  - **`mkWire`**: _write $<$ _read. Müssen aus verschiedenen Rules aufgerufen werden.
  - **`mkBypassWire`**: _write ist **`always_enabled`** $\rightarrow$ _read hat keine impliziten Bedingungen.

Aufgabe 1.1.2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Welche verschiedenen Wires sind in Bluespec verfügbar?
  - **`mkWire`**: _write $<$ _read. Müssen aus verschiedenen Rules aufgerufen werden.
  - **`mkBypassWire`**: _write ist **`always_enabled`** $\rightarrow$ _read hat keine impliziten Bedingungen.
  - **`mkDWire`**: _read ist **`always_ready`** da **`mkDWire`** einen Standardwert zurückliefert wenn _write nicht im selben Takt aufgerufen wurde.

Aufgabe 1.2.1: Ausführungsreihenfolge von r1, r2, r3

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  module mkFoo(Foo);
2    Reg#(Int#(32)) x <- mkReg(0);
3    Reg#(Int#(32)) y <- mkReg(0);
4    Wire#(Bool) even <- mkDWire(True);
5    Wire#(Int#(32)) x2 <- mkDWire(0);
6    rule r1;
7      even <= pack(x)[0] == 0;
8    endrule
9    rule r2;
10     if(even) y <= y + x;
11     else y <= y - x2;
12   endrule
13   rule r3;
14     x2 <= even ? (x + 42) : (x - 42);
15   endrule
16   method Action setX(Int#(32) px);
17     x <= px;
18   endmethod
19   method Int#(32) getY();
20     return y;
21   endmethod
22 endmodule
```

Aufgabe 1.2.1: Ausführungsreihenfolge von r1, r2, r3, setX und getY

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  module mkFoo(Foo);
2    Reg#(Int#(32)) x <- mkReg(0);
3    Reg#(Int#(32)) y <- mkReg(0);
4    Wire#(Bool) even <- mkDWire(True);
5    Wire#(Int#(32)) x2 <- mkDWire(0);
6    rule r1;
7      even <= pack(x)[0] == 0;
8    endrule
9    rule r2;
10     if(even) y <= y + x;
11     else y <= y - x2;
12   endrule
13   rule r3;
14     x2 <= even ? (x + 42) : (x - 42);
15   endrule
16   method Action setX(Int#(32) px);
17     x <= px;
18   endmethod
19   method Int#(32) getY();
20     return y;
21   endmethod
22 endmodule
```

Aufgabe 1.2.2: Dringlichkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule calc_m2;
2      m.enq(x / b);
3      m_done <= True;
4  endrule
5
6  rule calc_m1(fire);
7      m.enq(b * 1337);
8      m_done <= True;
9  endrule
10
11 rule flip;
12     fire <= !fire;
13 endrule
```

- Welches Scheduling-Problem gibt es in obigem Code?

Aufgabe 1.2.2: Dringlichkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule calc_m2;
2      m.enq(x / b);
3      m_done <= True;
4  endrule
5
6  rule calc_m1(fire);
7      m.enq(b * 1337);
8      m_done <= True;
9  endrule
10
11 rule flip;
12     fire <= !fire;
13 endrule
```

- Welches Scheduling-Problem gibt es in obigem Code?
- Wie reagiert der BSC auf dieses Problem?

Aufgabe 1.2.2: Dringlichkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule calc_m2;
2      m.enq(x / b);
3      m_done <= True;
4  endrule
5
6  rule calc_m1(fire);
7      m.enq(b * 1337);
8      m_done <= True;
9  endrule
10
11 rule flip;
12     fire <= !fire;
13 endrule
```

- Welches Scheduling-Problem gibt es in obigem Code?
- Wie reagiert der BSC auf dieses Problem?
- Wie kann sichergestellt werden das **`calc_m1`** zur Ausführung kommen kann?

Aufgabe 1.3.1: CAN_FIRE und WILL_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule crazy_stuff;
2    counter <= const1 + const2;
3    const2 <= const2 « 1;
4    const1 <= const1 - 1;
5  endrule
6
7  rule blinky (const1 >= 0);
8    if(pack(counter)[0] == 0) on <= True;
9    var1 <= var1 » 2;
10 endrule
```

- Wie sehen die CAN_FIRE Bedingungen der beiden Regeln aus?

Aufgabe 1.3.1: CAN_FIRE und WILL_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule crazy_stuff;
2    counter <= const1 + const2;
3    const2 <= const2 « 1;
4    const1 <= const1 - 1;
5  endrule
6
7  rule blinky (const1 >= 0);
8    if(pack(counter)[0] == 0) on <= True;
9    var1 <= var1 » 2;
10 endrule
```

- Wie sehen die CAN_FIRE Bedingungen der beiden Regeln aus?
  - **`crazy_stuff`**: **`True`**

Aufgabe 1.3.1: CAN_FIRE und WILL_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule crazy_stuff;
2    counter <= const1 + const2;
3    const2 <= const2 « 1;
4    const1 <= const1 - 1;
5  endrule
6
7  rule blinky (const1 >= 0);
8    if(pack(counter)[0] == 0) on <= True;
9    var1 <= var1 » 2;
10 endrule
```

- Wie sehen die CAN_FIRE Bedingungen der beiden Regeln aus?
  - **`crazy_stuff`**: **`True`**
  - **`blinky`**: **`const1 >= 0`**

Embedded Systems & Applications

Aufgabe 1.3.1: CAN_FIRE und WILL_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  rule crazy_stuff;
2    counter <= const1 + const2;
3    const2 <= const2 « 1;
4    const1 <= const1 - 1;
5  endrule
6
7  rule blinky (const1 >= 0);
8    if(pack(counter)[0] == 0) on <= True;
9    var1 <= var1 » 2;
10 endrule
```

- Wie sehen die CAN_FIRE Bedingungen der beiden Regeln aus?
  - **`crazy_stuff`**: **`True`**
  - **`blinky`**: **`const1 >= 0`**
- Wie sehen die WILL_FIRE Bedingungen der beiden Regeln aus?

Aufgabe 1.3.2: CAN_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 rule calc;
2   let c = fifos[switch].first();
3   fifos[switch].deq();
4   if(c[7] == 1) out.enq(255);
5   else out.enq(0);
6 endrule
```

- Wie sieht die CAN_FIRE Bedingung der Regel aus?

Aufgabe 1.3.2: CAN_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 rule calc;
2   let c = fifos[switch].first();
3   fifos[switch].deq();
4   if(c[7] == 1) out.enq(255);
5   else out.enq(0);
6 endrule
```

- Wie sieht die CAN_FIRE Bedingung der Regel aus?
  - **`out.i_notFull &&`**

Aufgabe 1.3.2: CAN_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 rule calc;
2   let c = fifos[switch].first();
3   fifos[switch].deq();
4   if(c[7] == 1) out.enq(255);
5   else out.enq(0);
6 endrule
```

- Wie sieht die CAN_FIRE Bedingung der Regel aus?
  - **`out.i_notFull &&`**
  - **`fifos_0.i_notEmpty`**

Aufgabe 1.3.2: CAN_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 rule calc;
2   let c = fifos[switch].first();
3   fifos[switch].deq();
4   if(c[7] == 1) out.enq(255);
5   else out.enq(0);
6 endrule
```

- Wie sieht die CAN_FIRE Bedingung der Regel aus?
  - **out.i_notFull &&**
  - **fifos_0.i_notEmpty**
  - **&& fifos_1.i_notEmpty && fifos_2.i_notEmpty && fifos_3.i_notEmpty**

Aufgabe 1.3.2: CAN_FIRE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 rule calc;
2   let c = fifos[switch].first();
3   fifos[switch].deq();
4   if(c[7] == 1) out.enq(255);
5   else out.enq(0);
6 endrule
```

- Wie sieht die CAN_FIRE Bedingung der Regel aus?
  - **`out.i_notFull &&`**
  - **`fifos_0.i_notEmpty`**
  - **`&& fifos_1.i_notEmpty && fifos_2.i_notEmpty && fifos_3.i_notEmpty`**
- Führt häufig zu Deadlocks

Aufgabe 1.3.2: Mögliche Lösung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jeder Wert von **`switch`** bekommt seine eigene Regel:

```
1 for(Integer i = 0; i < 4; i = i + 1) begin
2   rule calc (switch == fromInteger(i));
3     let c = fifos[i].first();
4     fifos[i].deq();
5     if(c[7] == 1) out.enq(255);
6     else out.enq(0);
7   endrule
8 end
```

Ende Theorieblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Sie können jetzt:

Ende Theorieblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Sie können jetzt:

- Scheduling-Eigenschaften verschiedener Hardware-Komponenten einordnen

Ende Theorieblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Sie können jetzt:

- Scheduling-Eigenschaften verschiedener Hardware-Komponenten einordnen
- Verstehen wie Konflikte auftreten und diese beheben

Ende Theorieblatt 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Sie können jetzt:

- Scheduling-Eigenschaften verschiedener Hardware-Komponenten einordnen
- Verstehen wie Konflikte auftreten und diese beheben
- Explizite Guards, implizite Guards, CAN_FIRE und WILL_FIRE unterscheiden

Fragerunde

TECHNISCHE UNIVERSITÄT DARMSTADT

Fragen zur Vorlesung oder zur Übung?