Vorlesung Architekturen und Entwurf von Rechnersystemen

Prof. Andreas Koch, Yannick Lavan, Johannes Wirth, Mihaela Damian

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023
Übungsblatt 1

Diese Übung dient der Vertiefung von Hintergrundwissen zu Bluespec SystemVerilog.

1.1 Register, Wires und CRegs

Bluespec verfügt über verschiedene Elemente, welche das Lesen und Schreiben von Werten ermöglichen. Die gängigsten Elemente sind hierbei Register, Wires und CRegs. In dieser Aufgabe werden diese drei Komponenten genauer betrachtet.

1.1.1 Register

Aus der Vorlesung sollten Sie wissen, dass die Methoden _read und _write des Reg-Interfaces der Präzedenzrelation _read < _write folgen, Lesezugriffe also vor Schreibzugriffen durchgeführt werden. Erläutern Sie kurz, welches Verhalten von Logikelementen auf diese Präzedenzrelation abgebildet wird. Sie dürfen hierbei auch eine Skizze zur Verdeutlichung verwenden.

1.1.2 Unterschiedliche Komponenten

Erläutern Sie die wesentlichen Unterschiede zwischen Registern, Wires und CRegs. Gehen Sie hierbei besonders auf die Präzedenzrelationen der Lese-/Schreibmethoden und den Zustand der Schaltung am Ende des Taktes ein.
Bonusfrage: Welche verschiedenen Wires sind in Bluespec verfügbar und was unterscheidet sie?

1.2 Scheduling

Aus der vorherigen Aufgabe sollten nun die wichtigsten Präzedenzrelationen bekannt sein. In dieser Aufgabe wird die Auswirkung der Präzedenzen auf das Scheduling einzelner Regeln betrachtet. Verwenden Sie zur Lösung dieser Aufgabe ***nicht*** den Bluespec Compiler.

1.2.1 Ausführungsreihenfolge

Gegeben sei der folgende BSV Code:

```
1 interface Foo;
2         method Action setX(Int#(32) px);
3         method Int#(32) getY();
4 endinterface
5
6 module mkFoo(Foo);
```

```
        Reg#(Int#(32)) x <- mkReg(0);
        Reg#(Int#(32)) y <- mkReg(0);
        Wire#(Bool) even <- mkDWire(True);
        Wire#(Int#(32)) x2 <- mkDWire(0);
        rule r1;
                even <= pack(x)[0] == 0;
        endrule

        rule r2;
                if(even) y <= y + x;
                else y <= y - x2;
        endrule

        rule r3;
                x2 <= even ? (x + 42) : (x - 42);
        endrule

        method Action setX(Int#(32) px);
                x <= px;
        endmethod

        method Int#(32) getY();
                return y;
        endmethod
endmodule
```

Die Regeln verfügen über keine impliziten Bedingungen (Guards) und können alle in jedem Takt feuern.
a) In welcher Reihenfolge feuern die Regeln r1, r2 und r3? Durch welche Präzedenzen wird dieses Verhalten hervorgerufen?
b) Nehmen Sie nun die Methoden setX und getY in den Schedule auf. In welcher Reihenfolge feuern die Regeln und Methoden? Warum ist das so?

1.2.2 Dringlichkeit

Gegeben sei der folgende BSV Code:

```
interface Bar;
        method Action putX(Int#(32) px);
        method Action putB(Int#(32) pb);
        method ActionValue#(Int#(32)) getY();
endinterface

module mkBar(Bar);
        Reg#(Int#(32)) x <- mkReg(0);
        Reg#(Int#(32)) y <- mkReg(0);
        Reg#(Int#(32)) b <- mkReg(0);
        FIFO#(Int#(32)) m <- mkFIFO();

        Reg#(Bool) y_done <- mkReg(False);
        Reg#(Bool) fire <- mkReg(False);
        Reg#(Bool) m_done <- mkReg(False);

        rule calc_m2;
                m.enq(x / b);
```

```
                m_done <= True;
        endrule

        rule calc_m1(fire);
                m.enq(b * 1337);
                m_done <= True;
        endrule

        rule calc_y (!y_done && m_done);
                y <= m.first() * x + b;
                m.deq();
                y_done <= True;
        endrule

        rule flip;
                fire <= !fire;
        endrule

        method Action putX(Int#(32) px);
                x <= px;
        endmethod

        method Action putB(Int#(32) pb);
                b <= pb;
        endmethod

        method ActionValue#(Int#(32)) getY() if(y_done);
                y_done <= False;
                return y;
        endmethod
endmodule

```

a) Welche Bedingung muss erfüllt sein, damit die Regel calc_y feuert?
b) Geben Sie eine mögliche Ausführungsreihenfolge innerhalb eines Taktes für die Regeln calc_m1, flip und calc_y an und begründen Sie diese Reihenfolge kurz.
c) In dem gegebenen Code-Beispiel gibt es ein Problem. Erläutern Sie kurz das Problem und wie es entsteht.
d) Wie lässt sich das Problem aus (c) lösen, um die Regel calc_m1 jeden zweiten Takt feuern zu lassen?

1.3 Feuerbereitschaft

In dieser Aufgabe werden die Bedingungen zur Ausführung von Regeln und Aktionen betrachtet.

1.3.1 CAN_FIRE und WILL_FIRE

Erläutern Sie anhand der Regel blinky des folgenden Code-Beispiels den Unterschied zwischen CAN_FIRE und WILL_FIRE.

```
interface FooBar;
        method Bool pulse();
endinterface

module mkFooBar(FooBar);
```

```
        Wire#(Bool) on <- mkDWire(False);
        Reg#(Int#(32)) counter <- mkReg(0);
        Reg#(Int#(32)) const1 <- mkReg(42);
        Reg#(Int#(32)) const2 <- mkReg(1337);
        Reg#(Int#(16)) var1 <- mkReg(1);

        rule crazy_stuff;
                counter <= const1 + const2;
                const2 <= const2 << 1;
                const1 <= const1 - 1;
        endrule

        rule blinky (const1 >= 0);
                if(pack(counter)[0] == 0) on <= True;
                var1 <= var1 >> 2;
        endrule

        method Bool pulse();
                return on;
        endmethod
endmodule
```

1.3.2 Implizite Guards

Gegeben sei der folgende Bluespec Code:

```
package FunkyFIFO;
        import FIFO::*;
        import ClientServer::*;
        import Vector::*;
        import GetPut::*;

        interface Bla;
                interface Server#(Bit#(8), Bit#(8)) server;
        endinterface

        module mkBinarizer(Bla);
                FIFO#(Bit#(8)) in <- mkFIFO();
                FIFO#(Bit#(8)) out <- mkFIFO();
                Reg#(UInt#(2)) switch <- mkReg(0);
                Vector#(4, FIFO#(Bit#(8))) fifos <- replicateM(mkSizedFIFO(1));

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

                rule queue;
                        let c = in.first();
                        in.deq();
                        fifos[switch].enq(c);
                endrule

```

```
30                  rule turn;
31                          switch <= switch + 1;
32                  endrule
33
34                  interface Server server;
35                          interface Get response = toGet(out);
36                          interface Put request = toPut(in);
37                  endinterface
38          endmodule
39  endpackage
```

a) Nennen Sie die kompletten CAN_FIRE Bedingungen der Regeln calc und queue. Wie kommen diese Guards zu stande?
b) Nehmen Sie an, dass jeden Takt versucht wird einen neuen Request in die FIFO `in` zu schreiben und wenn möglich eine Response aus der FIFO `out` zu entnehmen. Wie oft feuern die Regeln calc und queue? Warum?
c) Wie könnte man die Regeln öfter feuern lassen?