Vorlesung Architekturen und Entwurf von Rechnersystemen

Lösungsvorschlag

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

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.

Lösungsvorschlag:

Register werden in Hardware durch Flip-Flops realisiert. Ein D-Flip-Flop besteht aus zwei in Reihe geschalteten D-Latches, wobei der Clock-Eingang des hinteren Latches invertiert ist. Dadurch kann in der ersten Takthälfte der Wert des vorherigen Taktes gelesen werden (_read), der neue Wert wird hierbei vom vorderen D-Latch gehalten. In der zweiten Takthälfte übernimmt der hintere Latch den Wert, welcher vom vorderen Latch gehalten wurde, wodurch dieser im folgenden Takt lesbar ist (_write).

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?

Lösungsvorschlag:

Für Register und CRegs gilt grundsätzlich die Präzedenzrelation _read < _write, wobei bei CRegs noch die Portnummer für die Präzedenz relevant ist. Es gilt z.B. _read[0] < _write[0], aber _write[0] < _read[1]. Für Wires gilt grundsätzlich die Präzedenzrelation _write < _read.
Register behalten einen Wert, der geschrieben wurde für spätere Takte, CRegs halten den in einem Takt zuletzt geschriebenen Wert für spätere Takte und Wires halten Werte nur für die Dauer des Taktes, in dem sie geschrieben wurden.

Im folgenden werden Bluespecs wichtigste Wires kurz erläutert. Weitere Informationen finden Sie im BSV Reference Guide Kapitel B.4.4 und folgende oder in BSV by Example.
mkWire: Wire, bei dem Aufrufe zu _read und _write in unterschiedlichen Rules erfolgen müssen.
mkBypassWire: _write ist always_enabled, wodurch die _read Methode keine impliziten Bedingungen hat, da das Wire in jedem Takt geschrieben werden muss und damit in jedem Takt gültig ist.
mkDWire: Das DWire wird mit einem Standardwert initialisiert, wodurch _read always_ready ist. _read liefert diesen Standardwert als Rückgabe, wenn die Methode in einem Takt verwendet wird, in dem das DWire nicht geschrieben wurde.

An dieser Stelle wurden nur Wires beschrieben, die auch das Interface Wire implementieren. Es gibt noch weitere Wires mit verschiedenen Interfaces. Diese finden Sie im BSV Reference Guide.

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:

```
interface Foo;
        method Action setX(Int#(32) px);
        method Int#(32) getY();
endinterface

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?

Lösungsvorschlag:

a) Die Regeln feuern in der Reihenfolge r1, r3, r2. Das Wire even folgt der Präzedenz _write < _read. Da r1 even schreibt und r2 und r3 even lesen muss r1 vor den beiden anderen Rules feuern. x2 ist ebenfalls ein Wire und muss damit zuerst beschrieben werden, bevor es im gleichen Takt gelesen werden kann. Daher muss r3 vor r2 feuern.
b) Die Reihenfolge ist getY, r1, r3, r2, setX. Die Reihenfolge der Rules ändert sich nicht. Jede Rule liest den Wert von x, weshalb die Methode setX, die auf x schreibt, als letztes feuert. getY kann vor r1, r2 oder r3 feuern, da y von der Methode nur gelesen wird. Beide Lösungen sind in diesem Fall gültig. Die genannte Lösung ist die, welche vom bsc erzeugt wird.

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;
```

```
35          endrule
36
37          method Action putX(Int#(32) px);
38                  x <= px;
39          endmethod
40
41          method Action putB(Int#(32) pb);
42                  b <= pb;
43          endmethod
44
45          method ActionValue#(Int#(32)) getY() if(y_done);
46                  y_done <= False;
47                  return y;
48          endmethod
49  endmodule
50
```

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?

Lösungsvorschlag:

a) Die Bedingung ergibt sich aus der Konjunktion der expliziten und aller impliziten Guards. Die explizite Guard ist in diesem Beispiel (!y_done && m_done) und die einzige implizite Guard ist m.i_notEmpty. Damit ergibt sich die Bedingung (!y_done && m_done && m.i_notEmpty).

b) Reihenfolge: calc_y, calc_m1, flip. In der Guard von calc_y wird m_done gelesen, welches in calc_m1 geschrieben wird. Damit muss calc_y vor calc_m1 feuern. calc_m1 liest in seiner Guard fire, was in flip geschrieben wird. Dadurch wird calc_m1 vor flip feuern.

c) Die Regeln calc_m1 und calc_m2 stehen im Konflikt zueinander, sofern sie im gleichen Takt feuern können. Dieser Konflikt entsteht, weil beide Regeln auf die FIFO m die Methode enq aufrufen sollen, welche nur einmal pro Takt ausführbar ist. Durch diesen Konflikt wählt der Compiler willkürlich (aber deterministisch) eine Regel, die im Konfliktfall nicht feuert, was in diesem Fall calc_m1 ist. In diesem Fall würde die Regel calc_m1 nie feuern.

d) Man kann das Compiler Attribut descending_urgency verwenden, um die Reihenfolge, in der der Compiler die Regeln für das Scheduling betrachtet, zu ändern. In diesem Fall würde also (* descending_urgency=”calc_m1, calc_m2”*) das Problem lösen.

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
```

Lösungsvorschlag:

Die CAN_FIRE berechnet sich aus der Konjunktion aller expliziten und impliziten Guards einer Regel und bezieht sich auf die Feuerbereitschaft der Regel. Im Beispiel der Regel blinky existieren keine impliziten Guards, weshalb die CAN_FIRE Bedingung (const1 >= 0) lautet. Die WILL_FIRE Bedingung beschreibt, ob die Regel auch tatsächlich feuert.
Dies kann zum Beispiel nicht der Fall sein, falls ein Konflikt mit einer anderen Regel besteht und diese vom Compiler als dringlicher eingeschätzt wurde. Neben der WILL_FIRE Bedingung einer Regel gibt es noch die WILL_FIRE für Aktionen innerhalb der Regel.
Diese beschreibt, ob die jeweilige Action tatsächlich feuert. Da in diesem Beispiel kein Konflikt besteht ist die WILL_FIRE von blinky gleich zur CAN_FIRE von blinky. In der Regel blinky feuert die Action “var1 <= var1 » 2“ immer, wenn die Regel auch feuert. Ihre WILL_FIRE Bedingung ist also gleich der WILL_FIRE der rule blinky.
Die WILL_FIRE Bedingung der Action “on <= True“ hängt noch zusätzlich vom Prädikat (pack(counter)[0] == 0) ab.

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

                rule turn;
                        switch <= switch + 1;
                endrule

                interface Server server;
                        interface Get response = toGet(out);
                        interface Put request = toPut(in);
                endinterface
        endmodule
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?

Lösungsvorschlag:

---

a) CAN_FIRE für calc: (out.i_notFull && fifos_0.i_notEmpty && fifos_1.i_notEmpty && fifos_2.i_notEmpty && fifos_3.i_notEmpty).
CAN_FIRE für queue: (in.i_notEmpty && fifos_0.i_notFull && fifos_1.i_notFull && fifos_2.i_notFull && fifos_3.i_notFull).
Zunächst darf die FIFO out nicht voll sein, wenn calc feuern soll. Weiterhin müssen alle FIFOs im Vector fifos mindestens einen Wert enthalten, da die Konjunktion aller möglichen impliziten Guards gebildet wird. Analoge Begründung für die Regel queue.

b) Die Regel queue feuert exakt ein mal, da die implizite Guard fifos_1.i_notFull danach nicht mehr erfüllt ist und die Regel somit nie wieder feuern kann. Die Regel calc feuert gar nicht, da durch das einmalige feuern von queue nicht alle FIFOs befüllt werden können.

c) Man könnte jeweils für jede FIFO in fifos eine eigene Regel verwenden und in deren Guard den Wert des Registers switch abfragen (vgl. folgender Code).

```
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));

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

        for(Integer i = 0; i < 4; i = i + 1) begin
                rule queue (switch == fromInteger(i));
                        let c = in.first();
                        in.deq();
                        fifos[i].enq(c);
                endrule
        end

        rule turn;
                switch <= switch + 1;
        endrule

        interface Server server;
                interface Get response = toGet(out);
                interface Put request = toPut(in);
        endinterface
endmodule
```