Einführung in
Computer Microsystems
Sommersemester 2010

Wolfgang Heenes

TECHNISCHE UNIVERSITÄT DARMSTADT

5. Aufgabenblatt mit Lösungsvorschlag

19.05.2010

Aufgabe 1: Logik, Latch, Register

Geben Sie für alle folgenden `reg`*-Variablen an, ob sie bei der Synthese in Latches, Flip-Flops oder kombinatorische Logik übersetzt werden. Begründen Sie Ihre Antworten mit den Kriterien für potenzielle Register.*

a) *Listing 1:*

```
module a (input CLOCK, I, output reg O);
  reg T;
  always @(CLOCK, I)
    if (CLOCK) begin
      T = I;
      O = T;
    end
endmodule
```

Lösungsvorschlag:

O wird zu einem Latch da nicht zeitlich lokal und nicht vollständig. T wird in kombinatorische Logik übersetzt, da zeitlich lokal.

b) *Listing 2:*

```
module b (
  input wire       A,
  input wire [2:0] I,
  output reg [7:0] OCTAL);

always @(posedge A)
  case (I)
    3'h0: OCTAL = 8'b00000001;
    3'h1: OCTAL = 8'b00000010;
    3'h2: OCTAL = 8'b00000100;
    3'h3: OCTAL = 8'b00001000;
    3'h4: OCTAL = 8'b00010000;
    3'h5: OCTAL = 8'b00100000;
    3'h6: OCTAL = 8'b01000000;
    3'h7: OCTAL = 8'b10000000;
  endcase
endmodule
```

Lösungsvorschlag:

OCTAL wird Register/FF da *posedge* in der Aktivierungsliste und OCTAL nicht zeitlich lokal. Vollständigkeit ist irrelevant.

c) *Listing 3:*

```
module b (
  input wire [3:0] I,
  output reg [7:0] OCTAL);

always @(I)
  case (I)
    4'h0: OCTAL = 8'b00000001;
    4'h1: OCTAL = 8'b00000010;
    4'h2: OCTAL = 8'b00000100;
    4'h3: OCTAL = 8'b00001000;
    4'h4: OCTAL = 8'b00010000;
    4'h5: OCTAL = 8'b00100000;
    4'h6: OCTAL = 8'b01000000;
    4'h7: OCTAL = 8'b10000000;
  endcase
endmodule
```

Lösungsvorschlag:

OCTAL wird Latch, da nicht zeitlich lokal und unvollständig.

Aufgabe 2: BCD nach Binär Konverter

Implementieren Sie den folgenden Pseudo-Code für einen BCD nach Binär Konverter als Verhaltensbeschreibung in Verilog HDL. Die Länge der BCD-Zahl sei 24 Bit. Achten Sie darauf, dass bei der Synthese keine Latches entstehen. Testen Sie die Funktion ihres Moduls mit ISE.

Pseudo-Code für BCD nach Binär Konverter:

1. `bcd` ← `bcd_data_input`
2. `bin` ← *0 (gleiche Bitbreite wie* `bcd`*)*
3. *Für* `count` ← *1 bis Bitbreite von* `bcd` *iteriere:*
   - 3.1. `{bcd, bin}` ← `{bcd, bin}` >> *1*
   - 3.2. *Für jede 4-Bit Folge (3...0, 7...4, ...) in* `bcd` *iteriere*
     Wenn die 4-Bit Folge größer 7, dann ziehe 3 von dieser Folge ab
4. `bin` *enthält die konvertierte Zahl*

Lösungsvorschlag:

1. Möglichkeit: `for`-Schleife über `count` und fester Bitbreite.

```
module bcd_to_bin(
    input [23:0] BCD,
    output [23:0] BIN
);

    reg [2*24-1:0] bcd_concat_bin;
    integer count;

    assign BIN = bcd_concat_bin[23:0];
```

```
    always @(BCD) begin
        bcd_concat_bin = {BCD, 24'b0};
        // Schieben und subtrahieren
        for (count = 1; count <= 24; count = count + 1) begin
            bcd_concat_bin = bcd_concat_bin >> 1;
            // 4-Bit Folgen
            if(bcd_concat_bin[3+24] == 1)   // größer als 7 (MSB = 1)
                bcd_concat_bin[3+24:0+24] = bcd_concat_bin[3+24:0+24] - 3;
            if(bcd_concat_bin[7+24] == 1)   // größer als 7 (MSB = 1)
                bcd_concat_bin[7+24:4+24] = bcd_concat_bin[7+24:4+24] - 3;
            if(bcd_concat_bin[11+24] == 1)  // größer als 7 (MSB = 1)
                bcd_concat_bin[11+24:8+24] = bcd_concat_bin[11+24:8+24] - 3;
            if(bcd_concat_bin[15+24] == 1)  // größer als 7 (MSB = 1)
                bcd_concat_bin[15+24:12+24] = bcd_concat_bin[15+24:12+24] - 3;
            if(bcd_concat_bin[19+24] == 1)  // größer als 7 (MSB = 1)
                bcd_concat_bin[19+24:16+24] = bcd_concat_bin[19+24:16+24] - 3;
            if(bcd_concat_bin[23+24] == 1)  // größer als 7 (MSB = 1)
                bcd_concat_bin[23+24:20+24] = bcd_concat_bin[23+24:20+24] - 3;
        end
    end
endmodule
```

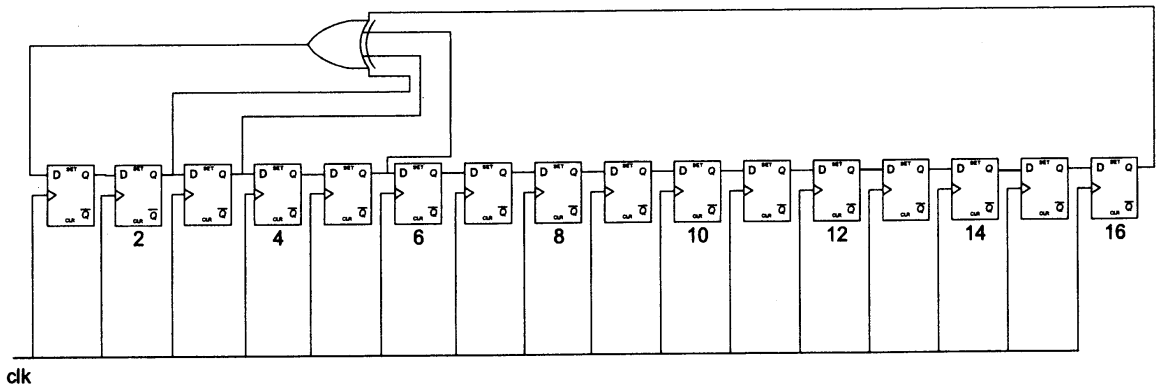
2. Möglichkeit: Mit definierbarer Bitbreite (`parameter`) und inneren `for`-Schleifen:

```
module bcd_to_bin(BCD, BIN);    // input und output definition kann wegen parameter erst im Modul erfolgen.

    parameter LENGTH = 24;

    input [LENGTH-1:0] BCD;
    output [LENGTH-1:0] BIN;

    reg [2*LENGTH-1:0] bcd_concat_bin;
    reg [3:0] temp;
    integer count, j, k;

    assign BIN = bcd_concat_bin[LENGTH-1:0];

    always @(BCD) begin
        bcd_concat_bin = {BCD, {LENGTH{1'b0}} };
        // Schieben und subtrahieren
        for (count = 1; count <= LENGTH; count = count + 1) begin
            bcd_concat_bin = bcd_concat_bin >> 1;
            // 4-Bit Folgen
            for (j = 0; j < LENGTH/4; j = j + 1) begin
                // Extrahiere 4 Einzelbits
                for (k = 0; k < 4; k = k + 1)
                    temp[k] = bcd_concat_bin[LENGTH + j*4 + k];
                if (temp[3] == 1) // größer als 7 (MSB = 1)
                    temp = temp - 3;
                // Und wieder bitweise zurück
                for (k = 0; k < 4; k = k + 1)
                    bcd_concat_bin[LENGTH + j*4 + k] = temp[k];
            end
        end
    end
endmodule
```

Aufgabe 3: Linear rückgekoppeltes Schieberegister

Die folgende Abbildung zeigt ein 16-stufiges linear rückgekoppeltes Schieberegister[1].

[1] engl. LFSR, Linear Feedback Shift Register



Beschreiben Sie das Schieberegister in Verilog HDL. Für das Flip-Flop können Sie die Primitive von Verilog HDL benutzen. Weisen Sie die korrekte Funktionsweise durch Simulation nach.

Lösungsvorschlag:

LFSRs können zur Realisierung von Pseudozufallszahlen genutzt werden. Der Initialzustand bestimmt dabei an welcher Stelle der Zahlenfolge begonnen wird. Sind alle Register auf 0 gesetzt, so ändert sich der Zustand des Schieberegisters nicht mehr, weshalb dies ein ungeeigneter Startzustand ist. Der komplette mögliche Zahlenraum wird nicht bei jeder beliebigen Rückkopplungsstruktur durchlaufen. Geeignete Rückkopplungen findet man mit Hilfe der Mathematik. Dabei wird jedem LFSR ein Polynom zugeordnet, dass die Rückkopplungsstruktur repräsentiert.[2] Bei dem LRSR aus dieser Aufgabe ist es das Polynom $x^{16} + x^5 + x^3 + x^2 + 1 = 0$.

Die folgenden Listings zeigen zwei Möglichkeiten das LRSR zu implementieren.

1. Möglichkeit: Mit selbst definiertem D-FF-Module `dff` und 16 explizit deklarierten Instanzen. Über den Reset- und den Preset-Eingang kann das LFSR zurückgesetzt werden. Hierbei sind nur die beiden Startzustände möglich, bei denen alle D-FFs gleich belegt sind.

```
module lfsr(
    input clk,
    input reset,
    input preset,
    output [15:0] out
    );

    wire xor_out;       // Ausgang XOR

    wire [0:15] d_out;  // Ausgänge D-FFs

    // XOR zur Realisierung des Polynoms x16 + x5 + x3 + x2 + 1 = 0
    xor(xor_out, d_out[1], d_out[2], d_out[4], d_out[15]);

    dff ins0 (clk, xor_out,   reset, preset, d_out[0] );
    dff ins1 (clk, d_out[0],  reset, preset, d_out[1] );
    dff ins2 (clk, d_out[1],  reset, preset, d_out[2] );
    dff ins3 (clk, d_out[2],  reset, preset, d_out[3] );
    dff ins4 (clk, d_out[3],  reset, preset, d_out[4] );
    dff ins5 (clk, d_out[4],  reset, preset, d_out[5] );
    dff ins6 (clk, d_out[5],  reset, preset, d_out[6] );
    dff ins7 (clk, d_out[6],  reset, preset, d_out[7] );
    dff ins8 (clk, d_out[7],  reset, preset, d_out[8] );
    dff ins9 (clk, d_out[8],  reset, preset, d_out[9] );
    dff ins10(clk, d_out[9],  reset, preset, d_out[10]);
    dff ins11(clk, d_out[10], reset, preset, d_out[11]);
```

[2] Genauere Informationen finden sich in der Literatur, z. B. Miron Abramovici, Melvin A. Breuer, Arthur D. Friedman, *Digital Systems testing and testable Design*, IEEE Press, New York, 1990

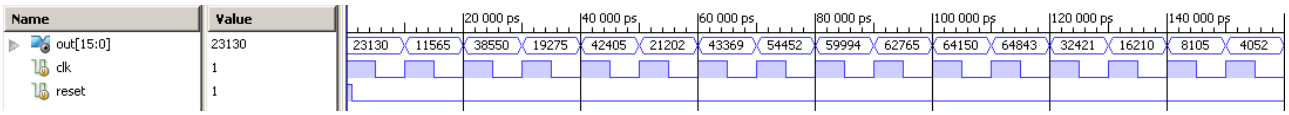
```
    dff ins12(clk, d_out[11], reset, preset, d_out[12]);
    dff ins13(clk, d_out[12], reset, preset, d_out[13]);
    dff ins14(clk, d_out[13], reset, preset, d_out[14]);
    dff ins15(clk, d_out[14], reset, preset, d_out[15]);

    assign out = d_out;     // FFs nach außen führen
endmodule

module dff(
    input clk,
    input d,
    input reset,
    input preset,
    output reg q
    );

    always @( posedge clk, posedge reset, posedge preset) begin
        if ( reset )
            q <= 0;
        else if ( preset )
            q <= 1;
        else
            q <= d;
    end
endmodule
```

2. Möglichkeit: Mit `for`-Schleife und Benutzung von `reg`. Bei dieser Implementierung kann ein Parameter übergeben werden, der den Initialzustand des Schieberegisters festlegt.

```
module lfsr(
    input clk,
    input reset,
    output [15:0] out
    );

parameter init = 16'b0101_1010_0101_1010;

wire xor_out;       // Ausgang XOR

reg [0:15] dff;     // 16 Register, werden zu D-FFs synthetisiert

// XOR zur Realisierung des Polynoms x16 + x5 + x3 + x2 + 1 = 0
assign xor_out = ^{dff[1], dff[2], dff[4], dff[15]};

integer i;  // Laufvariable
always @(posedge clk, posedge reset) begin
    if ( reset )
        dff <= init;    // Reset
    else begin
        dff[0] <= xor_out;      // ersten Eingang verdrahten
        for ( i=1; i<16; i=i+1 ) begin
            dff[i] <= dff[i-1]; // die anderen 15 automatisch generieren
        end
    end
end
assign out = dff;       // Register nach außen führen

endmodule
```

Beide Varianten liefern fast das gleiche Syntheseergebnis. Bei der zweiten Variante haben, um die initiale Bitfolge setzen zu können, einige D-FFs einen Reset-, die anderen einen Preset-Eingang, während beim ersten Modul alle D-FF sowohl Reset- als auch Preset-Eingänge haben.

Da die Zahlenfolge des LFSR $2^{16} - 1$ Elemente hat kann sie hier nicht komplett überprüft werden. Das folgende Bild zeigt die ersten Zahlen des zweiten Moduls, initialisiert mit dem Parameter init.


Name
Value
out[15:0]
23130
clk
1
reset
1
20 000 ps
40 000 ps
60 000 ps
80 000 ps
100 000 ps
120 000 ps
140 000 ps
23130
11565
38550
19275
42405
21202
43369
54452
59994
62765
64150
64843
32421
16210
8105
4052