Übung zur Vorlesung Einführung in Computer Microsystems

Prof. Dr. A. Koch
Thorsten Wink

Sommersemester 11
Übungsblatt 1 - Lösungsvorschlag

Die folgenden Aufgaben sollen in der HDL Verilog bearbeitet werden. Zur Simulation können Sie XILINX ISE verwenden. Es ist als WebPack-Edition frei verfügbar und auch auf den Poolrechnern der RBG installiert. Dort kann es einfach mit dem Befehl *ise* gestartet werden. Ein Tutorial zur Installation und Benutzung finden Sie auf unserer Webseite. Wir empfehlen, Version 11 zu verwenden.

Aufgabe 1.1 Zähler in Verilog

a) Beschreiben Sie einen 4-Bit-Zähler in Verilog HDL. Der Zähler hat einen Eingang für den Takt (mit `clk` bezeichnet). Der Zählerstand wird mit dem Ausgang `count` ausgegeben.

```
module counter(
  input clk,
  output reg[3:0] count
  );

  initial count = 0;

  always @(posedge clk)
    count <= count + 1
endmodule
```

b) Der Zähler soll um einen Eingang für ein *enable*-Signal erweitert werden. Es wird nur gezählt, wenn *enable* 1 ist.

```
module counter(
  input clk,
  output reg[3:0] count,
  input enable
  );

  initial count = 0;

  always @(posedge clk)
    if (enable)                 //nur wenn enable=1 ist hochzählen
      count <= count + 1;
endmodule
```

c) Der Zähler soll mit einem synchronen Reset (*sreset*) erweitert werden, so dass mit steigende Taktflanke der Zähler auf 0 zurückgesetzt wird wenn *sreset*=1.

```
module counter(
  input clk,
  output reg[3:0] count,
  input enable,
```

```
    input sreset
    );

    //initial count = 0;              //nun unnötig, da mit dem Reset-Signal zurückgesetzt wird


    always @(posedge clk)
      if (sreset)                     //wenn sreset gesetzt ist, count auf 0 setzen
        count <= 0;
      else if (enable)                //sonst wie bisher
        count <= count + 1;
  endmodule
```

d) Der Zähler soll mit einem asynchronen Reset (*areset*) erweitert werden, so dass unabhängig vom Takt der Zähler auf 0 zurückgesetzt wird.

```
  module counter(
    input clk,
    output reg[3:0] count,
    input enable,
    input sreset,
    input areset
    );

    always @(posedge clk or posedge areset)       //asynchroner Reset, muss in die sensitivity-list
                                                  //aufgenommen werden
      if (areset)                                 //asynchroner Reset
        count <=0;
      else if (sreset)                            //synchroner Reset
        count <= 0;
      else if(enable)                             //wie bisher
        count <= count + 1;
  endmodule
```

e) Über eine Leitung *set* und einen 4-Bit-Dateneingang *value* soll der Zähler synchron auf den Wert von *value* gesetzt wird, sobald *set* 1 ist.

```
  module counter(
    input clk,
    output reg[3:0] count,
    input enable,
    input sreset,
    input areset,
    input set,
    input [3:0] value
    );

    always @(posedge clk or posedge areset)
      if (areset)
        count <=0;
      else if (sreset)
        count <= 0;
```

```
    else if (set)                       //set-Vorgang
      count <= value;                   //value-Wert übernehmen
    else if (enable)
      count <= count + 1;
endmodule
```

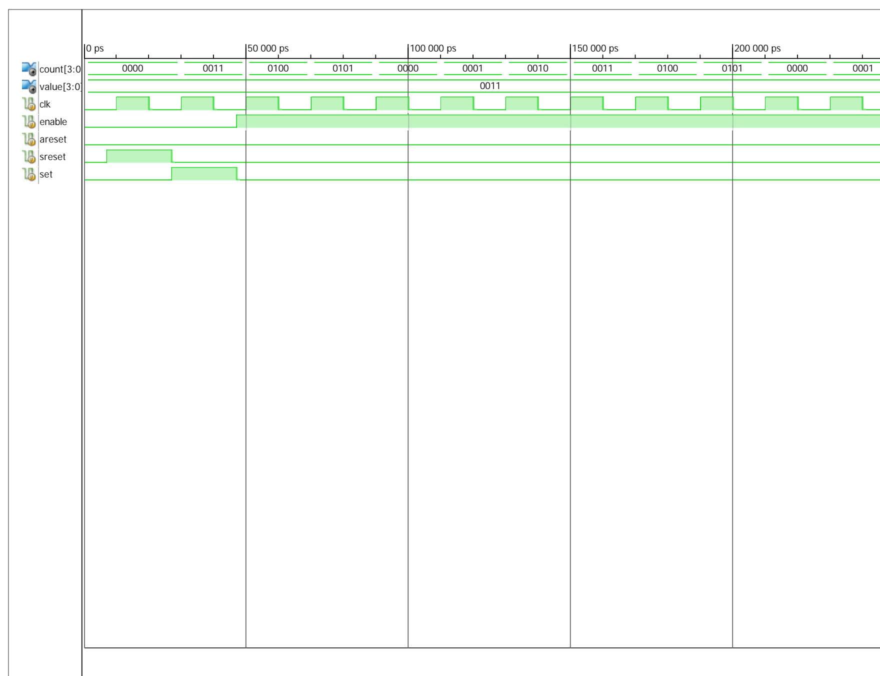
f) Der Zähler soll nur bis zu einem Wert *max* zählen, der über einen zu definierenden Parameter gesetzt werden kann. Ist kein Parameter beim Modulaufruf angegeben, soll wie bisher ohne einen Schwellwert gezählt werden.

```
module counter(
  input clk,
  output reg[3:0] count,
  input enable,
  input sreset,
  input areset,
  input set,
  input [3:0] value
  );
  parameter max = 15;                   //Standardwert 15, maximaler Wert bei 4 Bit

  always @(posedge clk or posedge areset)
    if (areset)
      count <=0;
    else if (sreset)
      count <= 0;
    else if (set)
      count <= value;
    else if (enable)
      if (count == max)                 //Schwellenwert erreicht
        count <= 0;
      else
        count <= count + 1;
endmodule
```

g) Schreiben Sie einen Testrahmen für die letzte Teilaufgabe, so dass $max = 5$. Zu Beginn sollen alle Eingangssignal auf 0 liegen. Nach 7 ns soll ein synchroner Reset erfolgen, danach soll der Startwert 3 gesetzt werden und der Zähler gestartet werden. Geben Sie ein Timing-Diagramm an, bei dem die Werte für *clk* und *count* zu sehen sind.

```
module countertest();

  reg clk;
  wire [3:0] count;
  reg enable,areset,sreset,set;
  wire [3:0] value = 4'b0011;

  //Instanziierung eines Counter mit den gewuenschten Eingaengen
  counter #(5) mycounter(
                        .clk(clk),
                        .count(count),
                        .enable(enable),
                        .areset(areset),
                        .sreset(sreset),
                        .set(set),
                        .value(value)
                        );
```

```
    //Taktgenerierung
    initial clk = 0;
    always
      #5 clk = ~clk;

    //Stimulus
    initial begin
      enable = 0;
      areset = 0;
      sreset = 0;
      set = 0;
      #7;
      sreset = 1;
      #20;
      sreset = 0;
      set = 1;
      #20;
      set = 0;
      enable = 1;
    end
  endmodule
```



Aufgabe 1.2 Paritätsbit

Schreiben Sie ein Verilog-Modul, das zu einem übergebenen Bitstring von n Bits ein Paritätsbit hinten anhängt, welches 1 ist, wenn die Anzahl der Einsen im Bitstring ungerade ist, und das 0 ist, wenn die Anzahl der Einsen im Bitstring gerade ist. Der so entstandene neue Bitstring soll der Ausgang des Moduls sein. Wird kein Parameter angegeben, so soll die Bitbreite des Ausgangs 9 Bit betragen.

```
module parity
  #(parameter n = 8)  //Parameterdeklaration im Modulkopf, damit er schon in der Port-Liste
                      //verwendet werden kann
```

```
    (
      input [n-1:0] in,
      output[n:0]   out
     );

  wire parity;
  assign parity = ^in;        //Reduktion mit xor
  assign out = {in, parity};  //Ausgang = Eingang mit angehaengtem Paritaetsbit
endmodule
```

Aufgabe 1.3 Fragen

a) Wie können Werte an Wire-Variablen zugewiesen werden? Geben Sie ein Beispiel an. Können Wires Werte speichern?

```
wire s; assign s=1;
wire s = a & b;
```

Wires können Werte nur transportieren, nicht speichern. Sie werden zur Verbindung von Modulen und Gattern verwendet.

b) Wie können Zuweisungen an Signale verzögert werden?
Verzögerungen können z. B. mit `assign #15 wire1 = x & y` angegeben werden.

c) Wie unterscheiden sich `initial` und `always`?
Ein `Initial Statement` wird nur einmalig ausgeführt. `Always` Statements werden in einer Endlosschleife ausgeführt.

Aufgabe 1.4 Addierer/Subtrahierer

Gegeben sind folgende Verilog HDL Module:

```
module HalfAdder(
  input A, B,
  output Sum, Carry
  );

  assign Sum = A ^ B;
  assign Carry = A & B;
endmodule

module FullAdder(
  input A, B, CarryIn,
  output Sum, CarryOut
  );

  wire sum1, carry1, carry2;

  HalfAdder ha1(.A(A), .B(B), .Sum(sum1), .Carry(carry1));
  HalfAdder ha2(.A(CarryIn), .B(sum1), .Sum(Sum), .Carry(carry2));

  assign CarryOut = carry1 | carry2;
endmodule
```

a) Konstruieren Sie mit Hilfe der beiden obigen Module einen 4-Bit Ripple-Carry Addierer mit den Eingängen `A` und `B` und dem Ausgang `Sum`. Schreiben Sie eine Testbench, welche die Additionsfunktion testet und führen Sie eine Verhaltenssimulation durch.

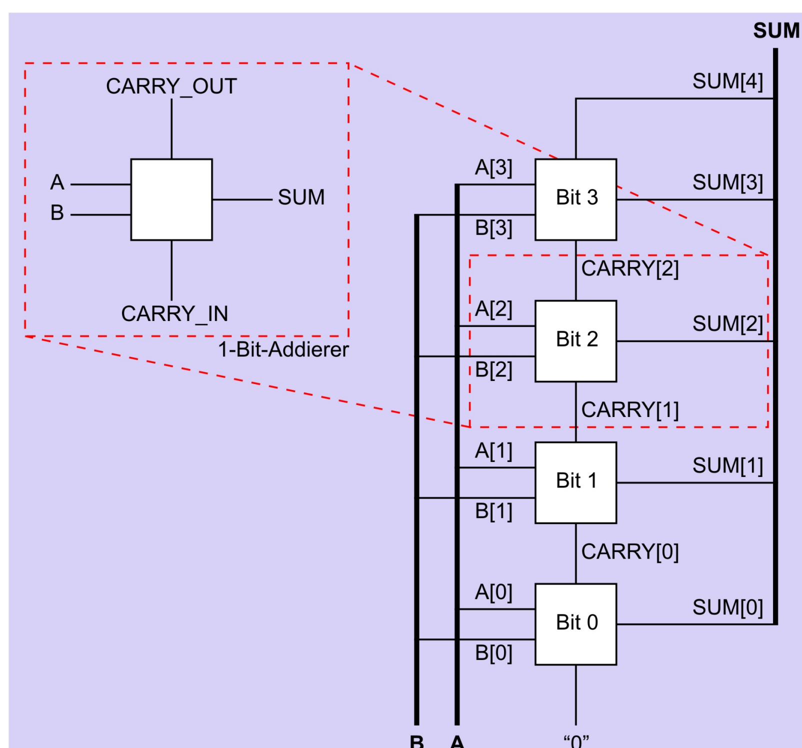
4-Bit Ripple-Carry Addierer:

```
module FourBitAdder(A, B, Sum);
    input [3:0] A;
    input [3:0] B;
    output [4:0] Sum;

    wire [2:0] carry;

    FullAdder Bit0 (.A(A[0]), .B(B[0]), .CarryIn(1'b0),
                    .Sum(Sum[0]), .CarryOut(carry[0]));

    FullAdder Bit1 (.A(A[1]), .B(B[1]), .CarryIn(carry[0]),
                    .Sum(Sum[1]), .CarryOut(carry[1]));

    FullAdder Bit2 (.A(A[2]), .B(B[2]), .CarryIn(carry[1]),
                    .Sum(Sum[2]), .CarryOut(carry[2]));

    FullAdder Bit3 (.A(A[3]), .B(B[3]), .CarryIn(carry[2]),
                    .Sum(Sum[3]), .CarryOut(Sum[4]));
endmodule
```

Hinweise:

Die Instanz *Bit0* kann auch durch einen Halbaddierer realisiert werden. Weil kein Carry auftritt, ist der Carry-Eingang auf Null gesetzt.

Aus dem Modul *HalfAdder* wird durch die Erzeugung von zwei Instanzen das Modul *FullAdder* erzeugt. Aus vier Modulen *FullAdder* wird der 4-Bit Ripple-Carry Addierer zusammengesetzt. Die folgende Abbildung verdeutlicht nochmal die Hierarchie.



Testbench:

```
'timescale 1ns / 1ps
```

```
module tb;
  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  // Outputs
  wire [4:0] Sum;

  //Instanz des Addierers
  FourBitAdder uut (
    .A(A),
    .B(B),
    .Sum(Sum)
  );

  //Stimulus
  initial begin
    A = 0;
    B = 0;
    # 5;
    //Testdaten
    A = 7; #10;
    B = 4; #10;
    B = 13; #10;
  end
endmodule
```

b) Erweitern Sie den 4-Bit Addierer aus a) um eine Subtraktionsfunktion. Ein zusätzliches Eingangssignal `Sub` soll von Addition auf Subtraktion umschalten. Der „-”-Operator darf dazu *nicht* verwendet werden. Erweitern Sie Ihre Testbench aus a) um den zusätzlichen Test der Subtraktion. Testen Sie auch negative Differenzen.

Das um den Sub-Eingang erweiterte Modul des 4-Bit Addierers:

```
module FourBitAddSub(
  input [3:0] A,
  input [3:0] B,
  input       Sub,
  output [4:0] Sum
  );

  wire [2:0] carry;
  wire       sum_4;
  wire [3:0] b_neg = B ^ {4 {Sub}}; // Replikation {n{m}}

  assign Sum[4] = sum_4 ^ Sub; // Vorzeichen

  FullAdder Bit0 (.A(A[0]), .B(b_neg[0]), .CarryIn(Sub),
                  .Sum(Sum[0]), .CarryOut(carry[0]));

  FullAdder Bit1 (.A(A[1]), .B(b_neg[1]), .CarryIn(carry[0]),
                  .Sum(Sum[1]), .CarryOut(carry[1]));

  FullAdder Bit2 (.A(A[2]), .B(b_neg[2]), .CarryIn(carry[1]),
                  .Sum(Sum[2]), .CarryOut(carry[2]));

  FullAdder Bit3 (.A(A[3]), .B(b_neg[3]), .CarryIn(carry[2]),
                  .Sum(Sum[3]), .CarryOut(sum_4));
```

```
endmodule
```

Zur Erklärung: Die Subtraktion wird auf die Addition mit dem Zweier-Komplement zurückgeführt.

Testbench:

```
‘timescale 1ns / 1ps
module tb_FourBitAddSub_v;
   // Inputs
   reg [3:0] A;
   reg [3:0] B;
   reg       Sub;

   // Outputs
   wire [4:0] Sum;

   // Instantiate the Unit Under Test (UUT)
   FourBitAddSub uut (
      .A(A),
      .B(B),
      .Sub(Sub),
      .Sum(Sum)
   );
  //Stimulus
   initial begin
      A = 0;
      B = 0;
      Sub = 0;
      //Testdaten
      A = 7; #10;
      B = 4; #10;
      B = 13; #10;
      Sub = 1; B = 0; A = 7; #10;
      B = 4; #10;
      B = 13; #10;
   end
endmodule
```

Aufgabe 1.5 Tageberechnung

Schreiben Sie ein Verilog-Modul, das als Eingabe den Monat erwartet (binär codiert) und als Ausgabe das Ergebnis liefert, ob der Monat 31 Tage hat oder nicht.

```
module month31days(
  input [3:0] month,     //4Bit nötig
  output reg  y);        //1Bit Ausgang

  always @ ( * )         //rein kombinatorische Zuweisung
    case (month)
      1:       y = 1’b1;
      2:       y = 1’b0;
      3:       y = 1’b1;
      4:       y = 1’b0;
      5:       y = 1’b1;
      6:       y = 1’b0;
```

```
        7:       y = 1'b1;
        8:       y = 1'b1;
        9:       y = 1'b0;
        10:      y = 1'b1;
        11:      y = 1'b0;
        12:      y = 1'b1;
        default: y = 1'b0;
    endcase
endmodule
```

Durch Bearbeitung und Abgabe der Hausaufgaben können Sie einen Notenbonus für die Prüfung CMS erwerben. Es gelten folgende Regeln:

- Die Bearbeitung ist in Gruppen zu 2 Studierenden möglich.
- Gruppenübergreifendes Erarbeiten von Lösungsideen ist erlaubt. Es muss jedoch jede Gruppe eine eigene Lösung abgeben. Wir werden dies kontrollieren, bei Plagiaten werden ALLE Bonuspunkte ALLER beteiligten Gruppen aberkannt!
- Es kann maximal ein Notenbonus von 3 Notenschritten erreicht werden.
- Der Bonus wird nur angerechnet, wenn die Klausur auch ohne den Bonus bestanden ist!!!
- Um den Bonus zu erhalten, muss die Abgabe über das Moodle-System bis zum Abgabetermin hochgeladen werden. Genaue Infos dazu werden rechtzeitig auf der Webseite und im Forum bekannt gegeben. Quellcode muss als Textdatei abgegeben werden, schriftliche Ausarbeitungen müssen im pdf-Format eingereicht werden. Bei Gruppen muss nur eine Abgabe erfolgen, alle Dokumente müssen die Namen und Matrikelnummern BEIDER Gruppenteilnehmer enthalten.

Diese Hausaufgaben müssen bis 6.5.11, 18:00 über das Moodle-System abgegeben werden.

Hausaufgabe 1.1 Mittelwert (5 Punkte)

a) Schreiben Sie ein Verilog-Modul, welches aus vier 16-Bit breiten Zahlen in Zweierkomplement-Darstellung den Mittelwert berechnet. Achten Sie auf eine möglichst effiziente Implementierung!

b) Schreiben Sie eine Testbench für Ihr Modul. Testen Sie darin alle Fälle, die Ihrer Meinung nach nötig sind, um einen fehlerfreien Betrieb zu gewährleisten. Begründen Sie jeden Eingabedatensatz mit erwartetem Ergebnis und dem Grund für diesen Testfall.

Hausaufgabe 1.2 Paralleler Multiplizierer (5 Punkte)

Konstruieren Sie aus den Modulen von Aufgabe 1.4 einen voll parallelen 4-Bit Multiplizierer für *positive* Zahlen. Der „*"-Operator darf dazu *nicht* verwendet werden. Wieviele Bits werden für das Ergebnis benötigt? Testen Sie die Funktion des Multiplizierers durch Simulation mit einer Testbench.

Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weitere Infos unter www.informatik.tu-darmstadt.de/plagiarism