Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Prof. Dr. Andreas Koch
Holger Lange

07.05.2007

Einführung in Computer Microsystems
2. Aufgabenblatt
Sommersemester 2007

Aufgabe 1: Addierer/Subtrahierer

Gegeben sind folgende Verilog-Module:

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

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

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

  wire sum1, carry1, carry2;

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

Lösung:

4-Bit Ripple-Carry Addierer:

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

Testbench:

```
`timescale 1ns / 1ps
module tb_FourBitAdder_v;

   // Inputs
   reg [3:0] A;
   reg [3:0] B;

   // Outputs
   wire [4:0] Sum;

   // Instantiate the Unit Under Test (UUT)
   FourBitAdder uut (
      .A(A),
      .B(B),
      .Sum(Sum)
   );

   initial begin
      // Initialize Inputs
      A = 0;
      B = 0;

      // Wait 100 ns for global reset to finish
      #100;

      // Add stimulus here
      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 unschalten. Der „–”-Operator darf dazu *nicht* verwendet werden. Erweitern Sie Ihre Testbench aus a) um den zusätzlichen Test der Subtraktion. Testen Sie auch negative Differenzen.

Lösung:

Aus **`FourBitAdder`** erweitertes Modul:

```
`timescale 1ns / 1ps
module FourBitAddSub(A, B, Sub, Sum);
    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}};

    assign Sum[4] = sum_4 ^ Sub;

    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
```

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)
   );

   initial begin
```

```
        // Initialize Inputs
        A = 0;
        B = 0;
        Sub = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        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 2: Paralleler Multiplizierer

Konstruieren Sie aus den Modulen von Aufgabe 1 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.

Lösung:

1. Möglichkeit mit 4-Bit Addierern:

```
`timescale 1ns / 1ps
module FourBitMult(A, B, Product);
    input [3:0] A;
    input [3:0] B;
    output [7:0] Product;

    wire [4:0] sum_1, sum_2, sum_3, sum_4;

    assign Product = {sum_4, sum_3[0], sum_2[0], sum_1[0]};
    assign sum_1 = A & { 4 {B[0]} };

    FourBitAdder Slice_2 (A & {4 {B[1]}}, sum_1[4:1], sum_2);
    FourBitAdder Slice_3 (A & {4 {B[2]}}, sum_2[4:1], sum_3);
    FourBitAdder Slice_4 (A & {4 {B[3]}}, sum_3[4:1], sum_4);
endmodule
```

2. Möglichkeit mit Halb- und Volladdierern:

```
`timescale 1ns / 1ps
module FourBitMultHAVA(A, B, Product);
    input [3:0] A;
    input [3:0] B;
    output [7:0] Product;

    wire [4:0] sum_1, sum_2, sum_3, sum_4;
```

```
    wire [2:0] carry_2, carry_3, carry_4;

    assign Product = {sum_4, sum_3[0], sum_2[0], sum_1[0]};
    assign sum_1 = A & { 4 {B[0]} };

    HalfAdder Slice_2_Bit_0 (A[0] & B[1], sum_1[1],
                                          sum_2[0], carry_2[0]);
    FullAdder Slice_2_Bit_1 (A[1] & B[1], sum_1[2], carry_2[0],
                                          sum_2[1], carry_2[1]);
    FullAdder Slice_2_Bit_2 (A[2] & B[1], sum_1[3], carry_2[1],
                                          sum_2[2], carry_2[2]);
    FullAdder Slice_2_Bit_3 (A[3] & B[1], sum_1[4], carry_2[2],
                                          sum_2[3], sum_2[4]);

    HalfAdder Slice_3_Bit_0 (A[0] & B[2], sum_2[1],
                                          sum_3[0], carry_3[0]);
    FullAdder Slice_3_Bit_1 (A[1] & B[2], sum_2[2], carry_3[0],
                                          sum_3[1], carry_3[1]);
    FullAdder Slice_3_Bit_2 (A[2] & B[2], sum_2[3], carry_3[1],
                                          sum_3[2], carry_3[2]);
    FullAdder Slice_3_Bit_3 (A[3] & B[2], sum_2[4], carry_3[2],
                                          sum_3[3], sum_3[4]);

    HalfAdder Slice_4_Bit_0 (A[0] & B[3], sum_3[1],
                                          sum_4[0], carry_4[0]);
    FullAdder Slice_4_Bit_1 (A[1] & B[3], sum_3[2], carry_4[0],
                                          sum_4[1], carry_4[1]);
    FullAdder Slice_4_Bit_2 (A[2] & B[3], sum_3[3], carry_4[1],
                                          sum_4[2], carry_4[2]);
    FullAdder Slice_4_Bit_3 (A[3] & B[3], sum_3[4], carry_4[2],
                                          sum_4[3], sum_4[4]);
endmodule
```

Gemeinsame Testbench für beide Möglichkeiten (nur Anpassung der UUT-Instanzierung erforderlich):

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

   // Outputs
   wire [7:0] Product;

   // Instantiate the Unit Under Test (UUT)
   FourBitMult uut ( // FourBitMultHAVA uut (
      .A(A),
      .B(B),
      .Product(Product)
   );

   initial begin
      // Initialize Inputs
```

```
        A = 0;
        B = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        A = 7;  B = 6;  #10;
        A = 15; B = 15; #10;
        A = 0; #10;
        A = 2; B = 11; #10;
    end

endmodule
```