Einführung in
Computer Microsystems

Sommersemester 2010

Wolfgang Heenes

2. Aufgabenblatt mit Lösungsvorschlag

28.04.2010

Aufgabe 1: Installation Xilinx ISE

Als erstes muss die Entwicklungsumgebung ISE installiert werden. Die Software ist für Windows und Linux verfügbar.[1] *Als Programmpaket wird das* **ISE WebPACK** *verwendet, um es herunterzuladen muss man bei Xilinx registriert sein. Die URL zum Download lautet:*

- *http://www.xilinx.com/ise/logic_design_prod/webpack.htm*

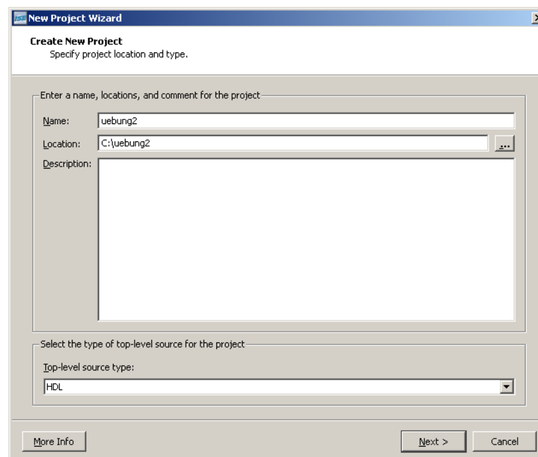
Fertig installiert nimmt das Tool etwa 4 GB Speicherplatz ein!

Weitere Informationen und Tutorials unter:

1. *http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/irn.pdf*
2. *http://www.xilinx.com/support/documentation/dt_ise11-1.htm*

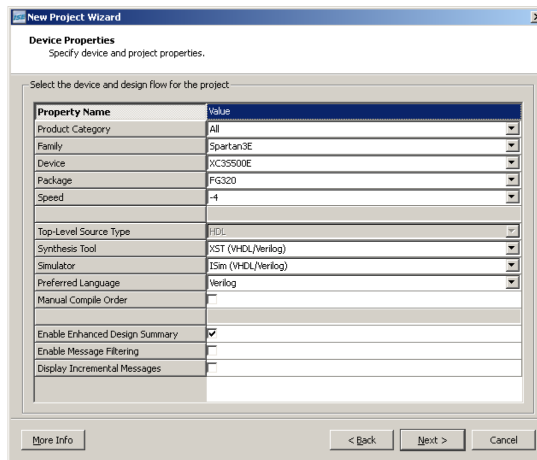
Aufgabe 2: Erstellung und Simulation eines Projektes

Legen Sie mit Xilinx ISE ein neues Projekt an. Nutzen Sie dazu den New Project Wizard (File → New Project. . .). Als Projektname geben Sie z. B. **uebung2** *ein. Außerdem können Sie ein Arbeitsverzeichnis wählen.*

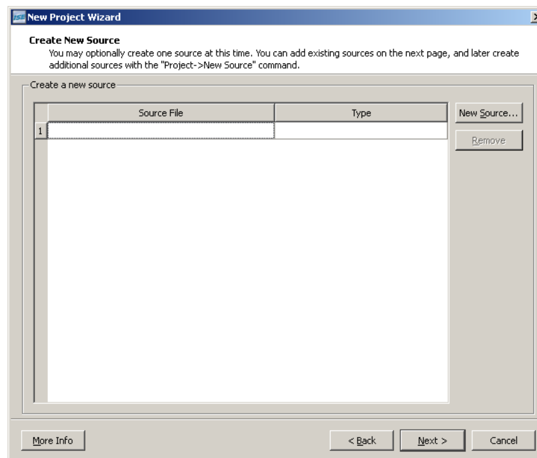


Geben Sie als Family **Spartan3E**, *als Device* **XC3S500E**, *als Package* **FG320** *und für Speed* **-4** *an. Wichtig ist, dass Sie als Simulator* **ISIM (VHDL/Verilog)** *auswählen. Die folgende Abbildung zeigt das Auswahlfenster.*

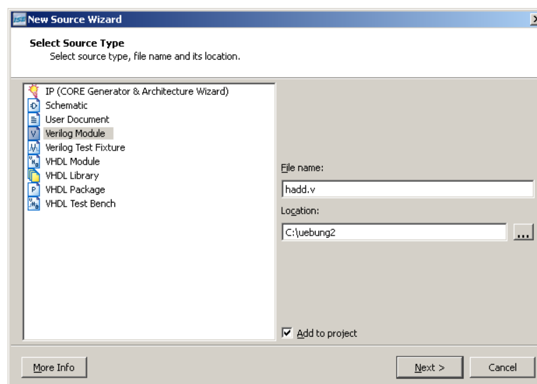
[1] Studierende mit anderen Betriebssystemen können die Übungen im Pool durchführen. Auf den Rechnern der RBG ist ebenfalls eine Installation vorhanden. Mit **ise** wird das Programm gestartet.



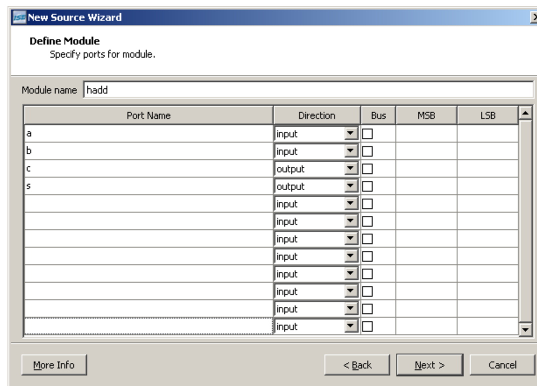
Das nächste Fenster erlaubt es neue Quellcodedateien zu erstellen. Klicken Sie auf New Source. . . .



Geben Sie als Dateinamen **hadd.v** *an und wählen Sie als Typ* **Verilog Module**.



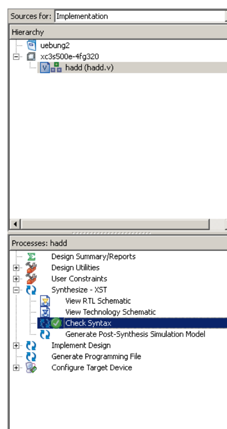
*Der New Source Wizard erlaubt die Eingabe der Eingänge und Ausgänge. Erstellen sie zwei Eingänge (***a** *und* **b***) sowie zwei Ausgänge (***c** *und* **s***).*



In den folgenden Fenstern klicken Sie Next bzw. Finish. Danach geht ein Fenster zur Eingabe des Quellcodes auf. Sie können nun folgendes Beispiel eines Halbaddierers vervollständigen.

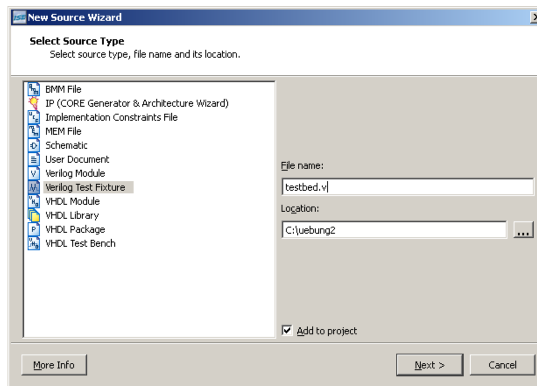
```
module hadd(
    input a,
    input b,
    output c,
    output s
    );
assign s = a ^ b; // ^ exklusiv-oder
assign c = a & b; // & und
endmodule
```

Nach der Eingabe soll die Syntax des Programms überprüft werden. Oben links bei Sources for muss Implementation ausgewählt und das Programm hadd.v markiert sein. Unter dem Punkt Synthesize-XST kann die Syntaxüberprüfung durch Check Syntax gestartet werden.



Erzeugen Sie nun eine Testbench[2]*. Dazu starten sie den New Source Wizard (Project → New Source). Wählen Sie diesmal Verilog Test Fixture und als Dateinamen* **testbed.v**.

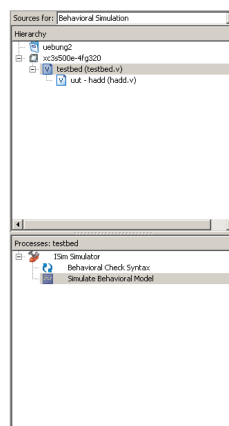
[2] Simulationsfile, Stimuli



Die nächsten Fenster werden mit Next und Finish bestätigt. Es wird wieder ein Verilog HDL File vorgegeben, welches entsprechend folgendem Beispiel zu ergänzen ist. Natürlich können auch andere Zeiten angeben werden.

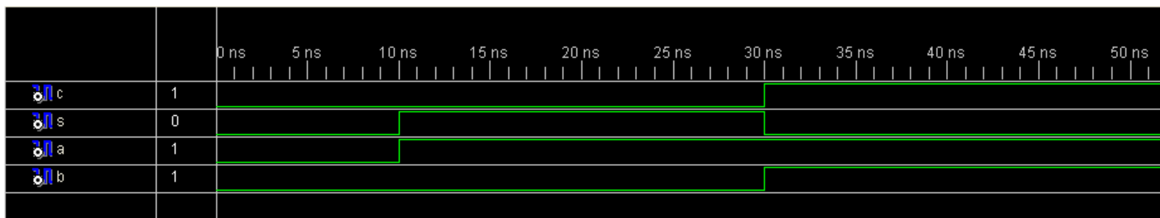
```
module testbed;
    // Inputs
    reg a;
    reg b;
    // Outputs
    wire c;
    wire s;

    // Instantiate the Unit Under Test (UUT)
    hadd uut (
        .a(a),
        .b(b),
        .c(c),
        .s(s)
    );

    initial begin
        // Initialize Inputs
        a = 0; b = 0;

        // Simulate
        #10 a = 1; b = 0;
        #20 a = 1; b = 1;
    end
endmodule
```

Für die Simulation muss im Fenster oben links bei Sources for jetzt Behavioral Simulation ausgewählt werden. Nach Markieren von testbed kann unter ISim Simulator mit Simulate Behavioral Model die Simulation gestartet werden.



Lösungsvorschlag:

Das Simulationsergebnis sollte etwa so aussehen:



Aufgabe 3: Simulation eines 4-Bit Zählers

Gegeben ist folgendes Verilog HDL Programm:

```
module zaehler(
    input clock,
    input up_down,
    output reg [3:0] qa
    );
initial qa = 4'b0;
integer direction;
always @(posedge clock) begin
    if (up_down) begin
        direction = 1;
    end else begin
        direction = -1;
    end
    qa <= qa + direction;
end // end always
endmodule
```

Simulieren Sie den Zähler und weisen Sie über die Simulation die korrekte Funktionsweise nach.

Lösungsvorschlag:

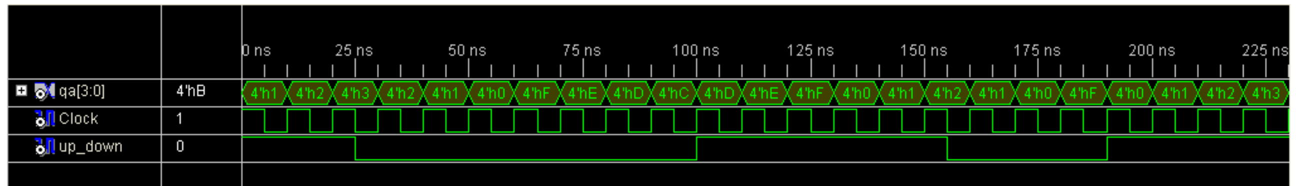
Wie ein Projekt erstellt und simuliert wird, ist in Aufgabe 2 beschrieben.
Zum Testen des Zählers kann folgende Testbench verwendet werden:

```
module testbed;
    // Inputs
    reg clock;
    reg up_down;
    // Outputs
    wire [3:0] qa;
    // Instantiate the Unit Under Test (UUT)
    zaehler uut (
        .clock(clock),
        .up_down(up_down),
        .qa(qa)
    );

    initial begin
        // clock erzeugen
        clock = 1;
        forever #5 clock = ~clock;
    end

    initial begin
        // Richtung initialisieren
        up_down = 1; // Hochzählen

        // Eingaben Simulieren
        #25 up_down = 0;
        #75 up_down = 1;
        #55 up_down = 0;
        #35 up_down = 1;
        #65 up_down = 0;
```

```
    end
endmodule
```

Sie simuliert einen Takt und wechselt in unregelmäßigen Abständen die Zählrichtung.

Das Simulationsergebnis sieht dann so aus:



Aufgabe 4: Addierer/Subtrahierer

Gegeben sind folgende Verilog HDL 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.*

Lösungsvorschlag:

4-Bit Ripple-Carry Addierer:

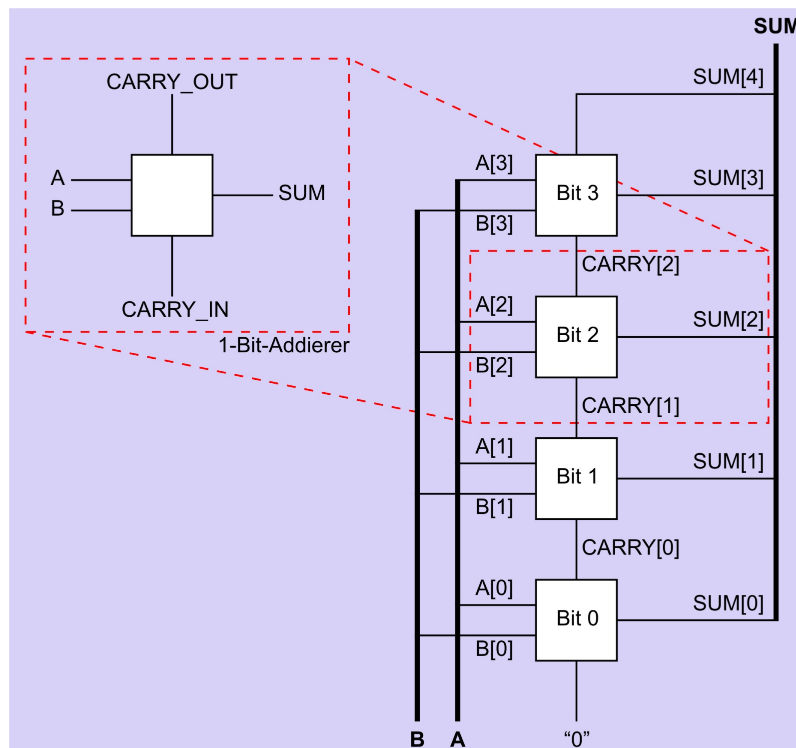
```
module FourBitAdder(A, B, Sum);
    input [3:0] A;
    input [3:0] B;
    output [4:0] Sum;

    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: Zur Angabe der Port-Verbindungen gibt es zwei Möglichkeiten. Neben der Verbindung durch die *Position* gibt es die Verbindung durch *Namen*. Die *Port-Verbindungen* in dem Programm *FourBitAdder* sind durch die *formalen Namen* (z. B. .A) und durch den aktuellen Namen (z. B. A[0]) angegeben. Die Angabe des formalen Namens erhöht die Verständlichkeit der Strukturbeschreibung.

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_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;
      # 5;
      // 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 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.*

Lösungsvorschlag:

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

```
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
```

Zur Erklärung: Die Subtraktion wird auf die Addition mit dem Zwei-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)
   );

   initial begin
      // Initialize Inputs
      A = 0;
      B = 0;
      Sub = 0;
      // Add stimulus here
      A = 7; #10;
      B = 4; #10;
      B = 13; #10;
      Sub = 1; B = 0; A = 7; #10;
      B = 4; #10;
```

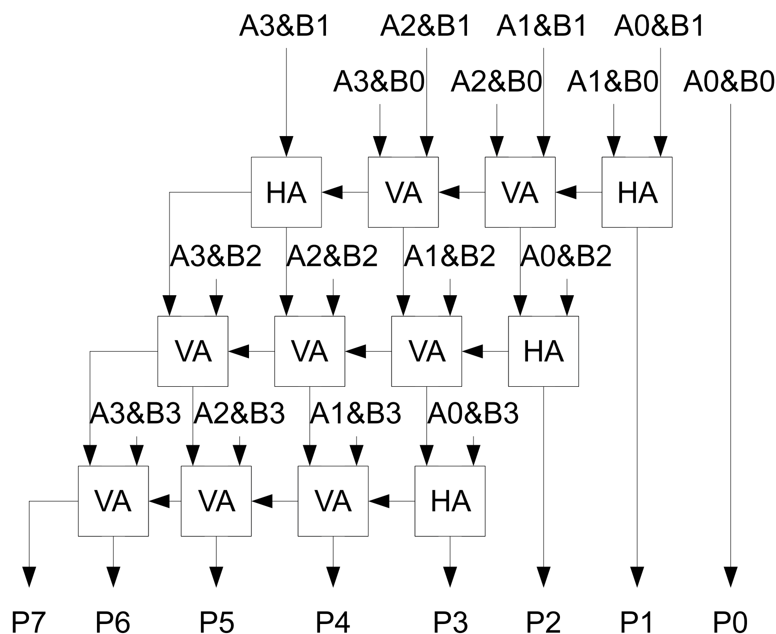
```
        B = 13; #10;
    end
endmodule
```

Aufgabe 5: Paralleler Multiplizierer

Konstruieren Sie aus den Modulen von Aufgabe 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.*

Lösungsvorschlag:

Zur Verdeutlichung wird folgende Abbildung betrachtet.



Das folgende Listing zeigt die Implementierung der Multiplikation.

```
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]);
    HalfAdder Slice_2_Bit_3 (A[3] & B[1], 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
```

Die Testbench sieht z. B. so aus:

```
`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)
   FourBitMultHAVA uut (
      .A(A),
      .B(B),
      .Product(Product)
   );

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

endmodule
```