Übung zur Vorlesung Einführung in Computer Microsystems

Prof. Dr. A. Koch
Thorsten Wink

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 11
Übungsblatt 4 - Lösungsvorschlag

Aufgabe 4.1 Synthese 1

Beschreiben Sie den Ablauf der Synthese. Siehe Folien 4-63ff.

Aufgabe 4.2 Synthese 2

Warum sind die Syntheseziele minimaler Flächenverbrauch und maximale Taktfrequenz (in der Regel) nicht gleichzeitig opimierbar? Siehe Folien 4-63ff.

Aufgabe 4.3 potenzielle Register

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

```
module potential_regs(A, S1, S2, Y, SUM0, SUM1, SUM2, SUM3);
  input [3:0] A;
  input [7:0] S1, S2;
  output reg Y;
  output reg [7:0] SUM0, SUM1, SUM2, SUM3;

  reg [31:0] I;
  reg C0, C1 = 0, C2, C3 = 0;

  always @(*) begin
    C0 = 0;
    for (I = 0; I < 8; I = I + 1) begin
      {C0, SUM0[I]} = S1[I] + S2[I] + C0;
      if (A)
        {C1, SUM1[I]} = S1[I] + S2[I] + C1;
    end
    if (C1 == 1) C1 = 0;
  end

  always @(posedge C0) begin
    C2 = 0;
    for (I = 0; I < 8; I = I + 1) begin
      {C2, SUM2[I]} = S1[I] + S2[I] + C2;
      if (A)
        {C3, SUM3[I]} = S1[I] + S2[I] + C3;
    end
    if (C3 == 1) C3 = 0;
  end

  always @(A) begin
    Y = 0;
    if (A[0] == 1) Y = 1;
    else if (A[1] == 1) Y = 1;
    else if (A[2] == 1) Y = 1;
  end
endmodule
```

Y	Kombinatorische Logik, da vollständig wegen Zuweisung `Y = 0;` (nicht lokal). Eingang A[3] ist unbenutzt.
SUM0	Kombinatorische Logik, da vollständig (nicht lokal)
SUM1	unvollständig, nicht lokal, ergibt Latch
SUM2, SUM3	Flip-Flops, da flankengesteuert und nicht lokal
I	Nicht lokal, aber vollständig, temporäre Variable deren Wert nirgends benutzt wird, entfällt
C0	Vollständig wegen `C0 = 0;` (nicht lokal), wird zu Verdrahtung (= komb. Logik)
C1	Vollständig (triviales `if (C1 == 1) C1 = 0;` wird zu `C1 = 0;`. C1 ist dann auch lokal, weil C1 am Ende des `always`-Blocks immer 0 ist und es wegen seiner räumlichen Lokalität nirgendwo anders geschrieben wird, also auch am Anfang einer neuen Iteration des Blocks immer 0 ist), wird zu Verdrahtung (= komb. Logik)
C2	Wird zu Verdrahtung da lokal und vollständig, Flankensteuerung ist in diesem Fall irrelevant, da der letzte Wert von C2 nirgends benutzt wird (anderenfalls würde für C2 sowohl Verdrahtung als auch ein Flip-Flop erzeugt)
C3	Vollständig (triviales `if (C3 == 1) C3 = 0;` wird zu `C3 = 0;`. C3 ist dann auch lokal, weil C3 am Ende des `always`-Blocks immer 0 ist und es wegen seiner räumlichen Lokalität nirgendwo anders geschrieben wird, also auch am Anfang einer neuen Iteration des Blocks immer 0 ist), wird zu Verdrahtung (= komb. Logik); Flankensteuerung ist in diesem Fall irrelevant, da der letzte Wert von C3 nirgends benutzt wird (anderenfalls würde für C3 sowohl Verdrahtung als auch ein Flip-Flop erzeugt)

Es sei darauf hingewiesen, dass der Verilog-Code in der Aufgabenstellung *keinen* guten Modellierungsstil darstellt. Er dient vielmehr als abschreckendes Beispiel, wie ein auf den ersten Blick relativ harmlos und einfach wirkender, aber *schlecht* entworfener Verilog-Code zu völlig unübersichtlichen Syntheseergebnissen und damit kaum vorauszusagender Hardware-Implementierung mit potenziellen Fehlfunktionen führt. Daher der Rat, nicht unnötig von den in der Vorlesung vorgeschlagenen Modellierungsweisen für kombinatorische und synchrone Logik abzuweichen.

Aufgabe 4.4 Fehlersuche

Gegeben ist folgendes Verilog-Modul, das einige Fehler enthält:

```
module faulty(A, B, Y, Z);
    input A;
    input [1:0] B;
    output reg Y;
    output [1:0] Z;

    reg A;

    always @(A)
      Z = A & B[0];

    always @(B)
      Z = B[1] & Y;

    always @(A)
      Y = B | A;

endmodule
```

Finden und beheben Sie die Fehler in der Verilog-Verhaltensbeschreibung. Vergleichen Sie dazu auch die Funktion des Moduls bei der Verhaltenssimulation mit der Post-Layout Simulation und den RTL-Schematics. Machen Sie Verbesserungsvorschläge und begründen Sie diese.

a) Zeile 7 muss `wire` statt `reg` sein, da `A` ein Eingang ist. Die Zeile kann auch ganz wegfallen, da Wires implizit deklariert sind, wenn nichts anderes angegeben wird.

b) Die Deklaration in Zeile 5 legt `Z` implizit als `wire` fest. Da `Z` aber ein Wert in einem prozeduralen Block zugewiesen wird, muss es als `reg` deklariert werden.

c) Dem Register `Z` wird im ersten *und* zweiten `always`-Block ein Wert zugewiesen. In der Simulation werden die Zuweisungen abhängig von den Änderungen von entweder A oder B ausgeführt. Ändern sich A und B gleichzeitig,

ist die Ausführungsreihenfolge unbestimmt. Bei einer Hardware-Synthese wären für diese Funktionalität Latches und flankengesteuerte Multiplexer notwendig, diese funktionieren im Allgemeinen unzuverlässig (Hazards!) und sind auf FPGAs gar nicht vorhanden. Folglich lässt sich keine Hardware für das beschriebene Verhalten generieren.

d) Bei allen `always`-Blöcken ist die Sensitivity-Liste nicht komplett. Die Simulation führt die Blöcke nur dann aus, wenn sich ein Signal in der Liste ändert (exakt so wie im Code), wohingegen das Hardware-Synthesewerkzeug von ISE immer das Vorhandensein einer vollständigen Liste annimmt. Anderenfalls müssten in der Hardware Latches erzeugt werden, um die alten Werte zwischenzuspeichern, wenn sich ein nicht in der Sensitivity-Liste aufgeführter Wert ändert. Simulation und Hardware stimmen somit ungewollt *nicht* überein.

e) Z[1] ist immer 0, kann daher wegfallen, falls nicht anderweitig verwendet.

Wieviele Register (Flip-Flops) werden bei der Übersetzung des Moduls in Hardware erzeugt?
Es werden *keine* Flip-Flops erzeugt. Ob bei der Synthese ein Flip-Flop generiert wird, hängt nicht davon ab, ob in Verilog ein `reg` deklariert wurde. Es kommt auf die Art der Verwendung an: Wird einem `reg` in einem *synchronen* Block (`always @(posedge CLK) ...`) ein Wert zugewiesen, so entsteht ein Flip-Flop in Hardware. Erfolgt die Zuweisung in einem *kombinatorischen* Block (`always @(A, B, ...) ...`) ohne Clock, so wird nur kombinatorische Logik generiert. Ein `wire` hingegen erzeugt immer rein kombinatorische Logik.

Aufgabe 4.5 generate

Was bewirkt der folgende Code?

```
module generate_example(

  imput  wire        read,write,
  input  wire [31:0] data_in,
  input  wire  [3:0] address,
  output wire [31:0] data_out
);
genvar i;

generate
  for (i=0; i < 4; i=i+1) begin : MEM
      memory U (read, write,
                data_in[(i*8)+7:(i*8)],
                address,data_out[(i*8)+7:(i*8)]);
  end
endgenerate
endmodule
```

Es werden 4 Instanzen des Moduls `memory` an einen 32Bit breiten Bus angeschlossen. So kann ein 32Bit breiter Speicher aus vier Speichermodulen mit je 8Bit zusammengebaut werden.

Diese Hausaufgaben müssen bis 17.6.11, 18:00 über das Moodle-System abgegeben werden.

Hausaufgabe 4.1 Pipeline (10 Punkte)

Hier nochmal die Pipeline-Aufgabe aus der Klausur.

$result = (2 * A + B) * 13 - C$

a) Zeichnen Sie ein Übersichtsplan der Pipeline. Hieraus soll ersichtlich sein, welche Operationen in welchem Takt durchgeführt werden. Zeichnen Sie hierzu alle Register, Funktionen und Module ein.

b) Beschreiben Sie das Modul in Verilog. Dabei gelten folgende Bedingungen:

- Alle Datenleitungen sollen 32 Bit breit sein
- Die Multiplikation kann durch ein bereitgestelltes Modul mit der folgenden Schnittstelle durchgeführt werden:

```
module mult(
  input  wire        clk,          //Takt
  input  wire        reset,        //synchroner Reset
  input  wire [31:0] A,            //Eingang A
  input  wire [31:0] B,            //   "    B
  output wire [31:0] result        //Ergebnis
);
endmodule
```

Dieses Modul benötigt 2 Taktzyklen, bis ein gültiges Ergebnis anliegt.

- Die Addition kann durch den + Operator durchgeführt werden. Sie benötigt 1 Takt. Diese Addition kann nur 2 Eingänge verarbeiten.
- Die Subtraktion kann durch den - Operator durchgeführt werden. Sie benötigt 1 Takt.
- Der $\star$ Operator darf NICHT verwendet werden.
- Achten Sie auf eine effiziente Implementierung

Der Code muss synthetisierbar sein. Kommentieren Sie Ihren Code.

```
module top(
  input  wire        clk,          //Takt
  input  wire        reset,        //synchroner Reset
  input  wire [31:0] A,            //Eingang A
  input  wire [31:0] B,            //   "    B
  input  wire [31:0] C,            //   "    C
  input  wire        input_valid,  //=1: es liegen gültige Eingänge an

  output wire [31:0] result,       //Ergebnis
  output wire        result_ready  //=1: das Ergebnis ist gültig
);

//Hier den eigenen Code einfügen

endmodule
```

Hier die Implementierung von `mult` (Diese Modul darf nicht verändert werden):

```
module mult(
  input  wire        clk,          //Takt
  input  wire        reset,        //synchroner Reset
  input  wire [31:0] A,            //Eingang A
  input  wire [31:0] B,            //   "    B
  output wire [31:0] result        //Ergebnis
);
reg [31:0] r1, r2;

always@(posedge clk) begin
  if(reset)begin
    r1 <= 0;
    r2 <= 0;
  end
  else begin
    r1 <= A * B;
    r2 <= r1;
  end
end

assign result = r2;

endmodule
```

c) Schreiben Sie eine Testbench für die Pipeline. Legen Sie dazu mindestens zwei verschiedene Eingabedatenpaare (ungleich 0)an und überprüfen Sie, ob das Ergebnis korrekt ist. Schreiben Sie das erwartete Ergebnis an der entsprechenden Stelle als Kommentar in den Code.

```
module tb()

  // Inputs
  reg clk;
  reg reset;
  reg [31:0] A;
  reg [31:0] B;
  reg [31:0] C;
  reg input_valid;

  // Outputs
  wire [31:0] result;
  wire result_ready;

  // Instantiate the Unit Under Test (UUT)
  top uut (
    .clk(clk),
    .reset(reset),
    .A(A),
    .B(B),
    .C(C),
    .input_valid(input_valid),
    .result(result),
    .result_ready(result_ready)
  );

  //Stimulus

endmodule
```