Einführung in

# Computer Microsystems

Sommersemester 2010

Wolfgang Heenes

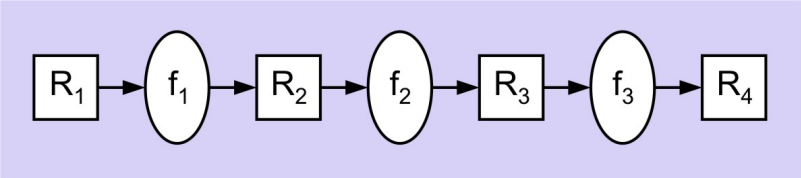
TECHNISCHE UNIVERSITÄT DARMSTADT

# 4. Aufgabenblatt mit Lösungsvorschlag

12.05.2010

## Aufgabe 1: Register-Transfer-Logik

*Gegeben ist folgende Pipeline in Register-Transfer-Logik:*




*Die kombinatorische Logik zwischen den Registern soll folgende Funktionen realisieren.*

- $f_1$: *verdoppeln*
- $f_2$: *plus 5*
- $f_3$: *quadrieren*

*Die Pipeline berechnet also* $R_4 = (2R_1 + 5)^2$.

a) *Beschreiben Sie die Pipeline in Verilog HDL. Die Funktionen sollen in Verilog HDL als* `function` *realisiert werden.*

*Ein Beispiel*[1] *für eine Funktion in Verilog HDL ist im Folgenden angegeben:*

```
module arithmetic_unit (result, operand_1, operand_2);
  output[3:0] result;
  input [3:0] operand_1, operand_2;

  assign result = largest_operand (operand_1, operand_2);

  function [3:0] largest_operand;
  input [3:0] operand_1, operand_2;
   largest_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
  endfunction
endmodule
```

*Die Funktion largest_operand bestimmt den größeren Operanden.*

*Die Register R1, R2, R3 und R4 sind jeweils 8-Bit breit. Überlaufen der Register kann vernachlässigt werden.*

*Das Register R1 soll mit einem Wert geladen werden können. Wenn ein Steuersignal* `ldreg1` *gesetzt ist, soll über den Port* `in` *von außen ein Wert in das Register R1 geladen werden.*

*Das Ergebnis der Berechnung soll über den Port* `out` *nach außen geführt werden.*

*Weisen Sie die korrekte Funktionsweise der Pipeline durch Simulation nach.*

[1] vgl. auch Ciletti, Michael D.: Advanced Digital Design with the Verilog HDL. Prentice Hall, 2003. Seite 190

## Lösungsvorschlag:

Die Pipeline kann entsprechend dem folgenden Listing implementiert werden.

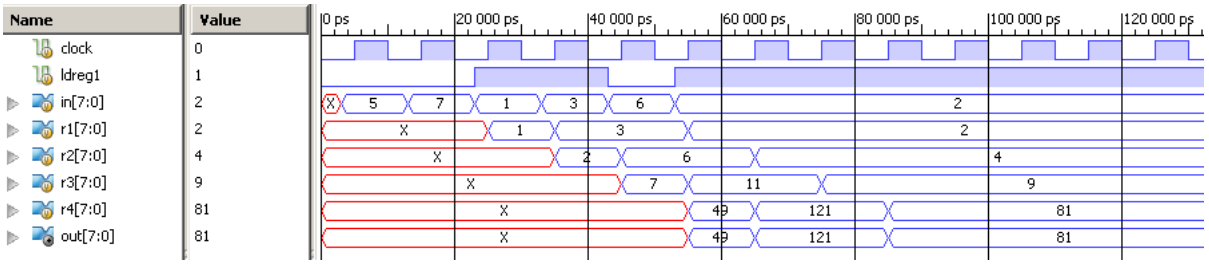
```
module pipeline_a(
     input clock,
     input ldreg1,
     input [7:0] in,
     output [7:0] out
     );

reg [7:0] r1, r2, r3, r4;

// erste Funktion: verdoppeln
function [7:0] f1;
    input [7:0] arg;
    begin
        f1 = arg*8'h02;
    end
endfunction

// zweite Funktion: plus 5
function [7:0] f2;
    input [7:0] arg;
    begin
        f2 = arg+8'h05;
    end
endfunction

// dritte Funktion: quadrieren
function [7:0] f3;
    input [7:0] arg;
    begin
        f3 = arg*arg;
    end
endfunction

// Die Pipeline
always @(posedge clock)
begin
    if (ldreg1)
        r1 <= in;
    else
        r1 <= r1;

    r2 <= f1(r1);
    r3 <= f2(r2);
    r4 <= f3(r3);
end

// Ausgang zuweisen
assign out = r4;
endmodule
```

Um die Funktionalität nachzuweisen wird folgende Testbench verwendet.

```
module testbench_a;
    // Inputs
    reg clock;
    reg ldreg1;
    reg [7:0] in;
    // Outputs
    wire [7:0] out;

    // Instantiate the Unit Under Test (UUT)
    pipeline_a uut (
        .clock(clock),
        .ldreg1(ldreg1),
        .in(in),
        .out(out)
```

```
    );

    // Simulation
    initial begin
        ldreg1 = 0;
        #3; // vor der ersten positiven Taktflanke
        in = 5;
        #10; // 1 Takt warten
        in = 7;
        #10; // 1 Takt warten
        ldreg1 = 1;     // Wert laden
        in = 1;
        #10; // 1 Takt warten
        in = 3;
        #10; // 1 Takt warten
        ldreg1 = 0;     // Keinen Wert laden
        in = 6;
        #10; // 1 Takt warten
        ldreg1 = 1;     // Wert laden
        in = 2;
    end

    // Takt
    initial begin
        clock = 1'b0;
        forever #5 clock = !clock;
    end
endmodule
```

Die Testbench setzt `in` und `ldreg1` auf verschiedene Werte damit überprüft werden kann ob das Ergebnis korrekt ist und nur bei gesetztem `ldreg1` ein Wert geladen wird. Das Simulationsergebnis ist in der nächsten Abbildung zu sehen. Man sieht wie die Zwischenergebnisse durch die vier Register der Pipeline "wandern".




b) *Die Pipeline aus Aufgabenteil a) soll um einen asynchronen Reset-Eingang* `areset` *erweitert werden, mit dem die Pipeline zurückgesetzt werden kann. Wird* `areset` *gesetzt, soll die Berechnung sofort gestoppt werden und solange keine neue Berechnung begonnen werden, bis* `areset` *wieder den Wert 0 annimmt.*

## Lösungsvorschlag:

Das Modul kann wie folgt geändert und erweitert werden:

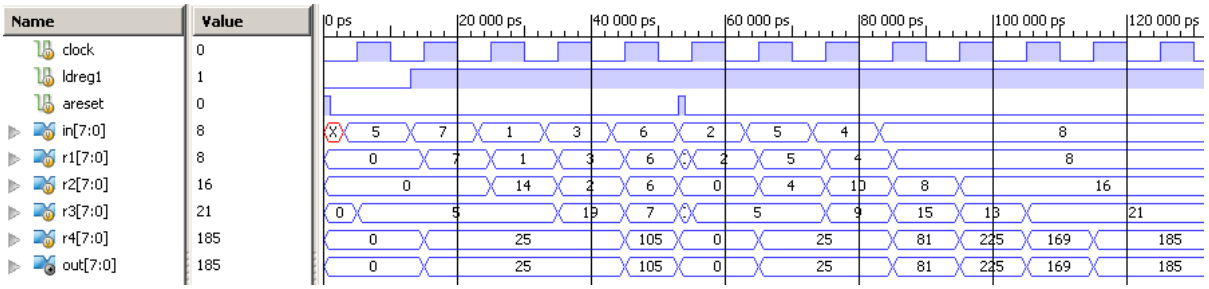
```
module pipeline_b(
     input clock,
     input areset,
     input ldreg1,
     input [7:0] in,
     output [7:0] out
     );

     ...

// Die Pipeline
always @(posedge clock or posedge areset)
if(areset) begin
    r1 <= 8'h00;
    r2 <= 8'h00;
    r3 <= 8'h00;
```

```
    r4 <= 8'h00;
end else begin
    if (ldreg1)
        r1 <= in;
    else
        r1 <= r1;

    r2 <= f1(r1);
    r3 <= f2(r2);
    r4 <= f3(r3);
end

     ...
```

In der Testbench muss `areset` ergänzt werden und der für die Simulation zuständige Initial-Block mit entsprechenden Testfällen erweitert werden.

```
module testbench_b;

    ... // Input, Output und UUT

    // Simulation
    initial begin
        areset = 0;
        ldreg1 = 0;
        areset = 1'b1;  // Pipeline am Anfang zurücksetzen
        #1; // kurz halten
        areset = 1'b0;  // Signal zurücknehmen
        #2; // vor der ersten positiven Taktflanke
        in = 5;
        #10; // 1 Takt warten
        ldreg1 = 1; // Wert laden
        in = 7;
        #10; // 1 Takt warten
        in = 1;
        #10; // 1 Takt warten
        in = 3;
        #10; // 1 Takt warten
        in = 6;
        #10; // 1 Takt warten
        in = 2;

        areset = 1'b1;  // Pipeline zurücksetzen
        #1; // kurz halten
        areset = 1'b0;  // Signal zurücknehmen

        #9; // rest vom Takt warten
        in = 5;
        #10; // 1 Takt warten
        in = 4;
        #10; // 1 Takt warten
        in = 8;
        // usw...
    end

    ...
```

Die Simulation sieht dann so aus:




c) *Da die Pipeline einige Takte benötigt um das Ergebnis zu berechnen, liegen am Ausgang zwischenzeitlich falsche Werte an. Das Module aus Aufgabenteil b) soll deshalb um ein Signal* `valid` *erweitert werden.* `valid` *soll genau dann den Wert 1 haben, wenn der Wert an* `out` *ein korrektes Ergebnis eines geladenen Wertes ist.*

## Lösungsvorschlag:

Das folgende Listing zeigt die Änderungen für das valid-Signal

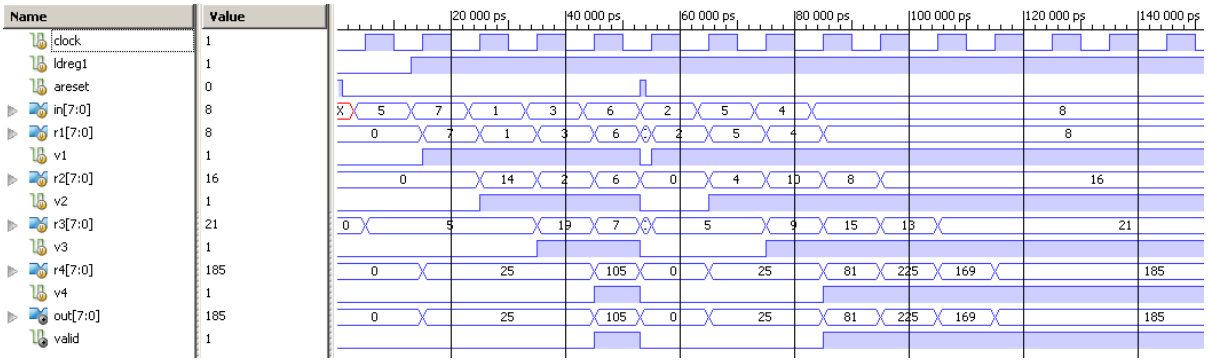
```
module pipeline_c(
     input clock,
     input areset,
     input ldreg1,
     input [7:0] in,
     output valid,
     output [7:0] out
     );

reg [7:0] r1, r2, r3, r4;
reg v1, v2, v3, v4;

    ...

// Die Pipeline
always @(posedge clock or posedge areset)
if(areset) begin
    r1 <= 8'h00;
    r2 <= 8'h00;
    r3 <= 8'h00;
    r4 <= 8'h00;

    v1 <= 1'b0;
    v2 <= 1'b0;
    v3 <= 1'b0;
    v4 <= 1'b0;
end else begin
    if (ldreg1)
        r1 <= in;
    else
        r1 <= r1;

    r2 <= f1(r1);
    r3 <= f2(r2);
    r4 <= f3(r3);

    v1 <= ldreg1;
    v2 <= v1;
    v3 <= v2;
    v4 <= v3;
end

// Ausgang zuweisen
assign out = r4;
assign valid = v4;
endmodule
```

Für jede Stufe der Pipeline wird ein Register eingeführt (`v1` – `v4`), dass das valid-Signal zu dem Wert im entsprechenden Register (`r1` – `r4`) enthält. So "wandert" das valid-Bit mit dem (Zwischen-)Ergebnis durch die Pipeline.

Die Testbench kann im Wesentlichen aus Aufgabenteil b) übernommen werden. Dort muss nur ein `wire` für das valid-Signal eingepflegt werden.

```
module testbench_c;

    ...

    // Outputs
    wire valid;

    ...

    // Instantiate the Unit Under Test (UUT)
    pipeline_c uut (
        .clock(clock),
        .areset(areset),
        .ldreg1(ldreg1),
        .in(in),
        .valid(valid),
        .out(out)
    );

    ...
```

Der folgenden Abbildung kann das Simulationsergebnis entnommen werden.




## Aufgabe 2: Zeitbehaftete Simulation

*Betrachtet wird der 4-Bit Ripple-Carry Addierer aus dem 2. Aufgabenblatt, Aufgabe 4. Die technische Realisierung, z. B. auf einem FPGA, führt dazu, dass Signale sich nicht sofort ändern, da Leitungen und Lock-Up-Tabellen Verzögerungen hervorrufen. Diese Verzögerungen sollen im Folgenden an dem 4-Bit Ripple-Carry Addierer gezeigt werden. Als FPGA ist das* **Spartan3E**-*Device (vgl. 2. Aufgabenblatt, Aufgabe 2) zu verwenden.*

```
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]),
```

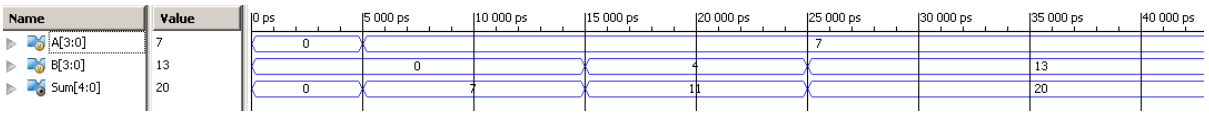
```
                .Sum(Sum[3]), .CarryOut(Sum[4]));
endmodule
```

*Neben der bekannten Verhaltenssimulation kann man mit der Entwicklungsumgebung auch zeitbehaftete Simulationen durchführen. Dazu wird in dem linken oberen Fenster unter Sources for der Punkt* **Post-Route Simulation** *ausgewählt. Unter Processes kann nun durch Auswahl von* **Simulate Post-Place & Route Model** *die zeitbehaftete Simulation gestartet werden.*

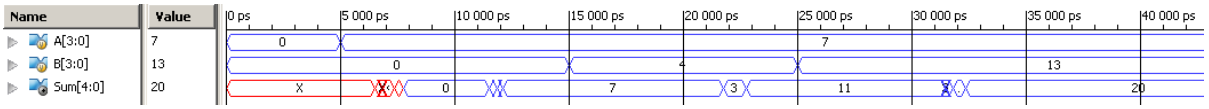
a) *Vergleichen Sie die Ergebnisse der Verhaltenssimulation mit den Ergebnissen der zeitbehafteten Simulation.*

## Lösungsvorschlag:

Zur Simulation wird die gleiche Testbench wie im 2. Aufgabenblatt verwendet. Mit dieser wird in der *Verhaltenssimulation* folgendes Ergebnis erzielt.




Für die *Post-Route Simulation* wird ebenfalls diese Testbench verwendet. Das Ergebnis unterscheidet sich jedoch deutlich.

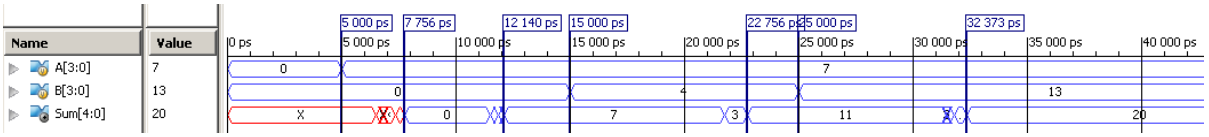



Man erkennt, dass in der *Verhaltenssimulation* die Ergebnisse der Addition sofort korrekt am Ausgang anliegen. Bei der *Post-Route Simulation* hingegen dauert es eine gewisse Zeit bis sich der Ausgang ändert. Der anliegende Wert ist i. d. R. zunächst falsch und wechselt bevor er stabil und richtig wird.

b) *Wie lange dauert es, bis eine Änderung am Eingang eine Änderung am Ausgang bewirkt?*

## Lösungsvorschlag:

In folgender Abbildung sind Marker an einigen Wertänderungen eingefügt.




Bei den hier durchgeführten Transitionen tritt ein Wechsel am Ausgang mit 7,14 – 7,37 ns Verzögerung auf. Da die verwendete Testbench nicht alle möglichen Transitionen prüft, sind auch kürzere oder längere Verzögerungen möglich.