

# Digitaltechnik

## Kapitel 8, VHDL-Vertiefung

**Prof. Dr.-Ing. M. Winzker**

*Nutzung nur für Studierende der Hochschule Bonn-Rhein-Sieg gestattet.  
(Stand: 21.03.2022)*

# 8.1 Kurzwiederholung VHDL und Top-Down Entwurf

- Der Entwurf von digitalen Systemen erfolgt üblicherweise nach dem **Top-Down Prinzip**
  - Ausgehend von der Spezifikation wird das Gesamtsystem in Teilschaltungen aufgeteilt
    - ➔ Bezeichnung auch: Untermodul
  - Die Untermodule werden wiederum in weitere Untermodule aufgeteilt
  - Die Untermodule werden dann einzeln entworfen und **Bottom-Up** bis zur Gesamtschaltung zusammengesetzt

**Beispiel:** Display-Controller als Steuereinheit eines Daten- und Video-Projektors („Beamer“)

- Signalverarbeitung
  - Skalierung der Eingangsbilder
  - Deinterlacing für Video-Signale
  - Freeze: Einfrieren des Bildes
- Steuerung des gesamten Geräts
  - Reaktion auf Tastendrücke, IR-Fernbedienung, ...

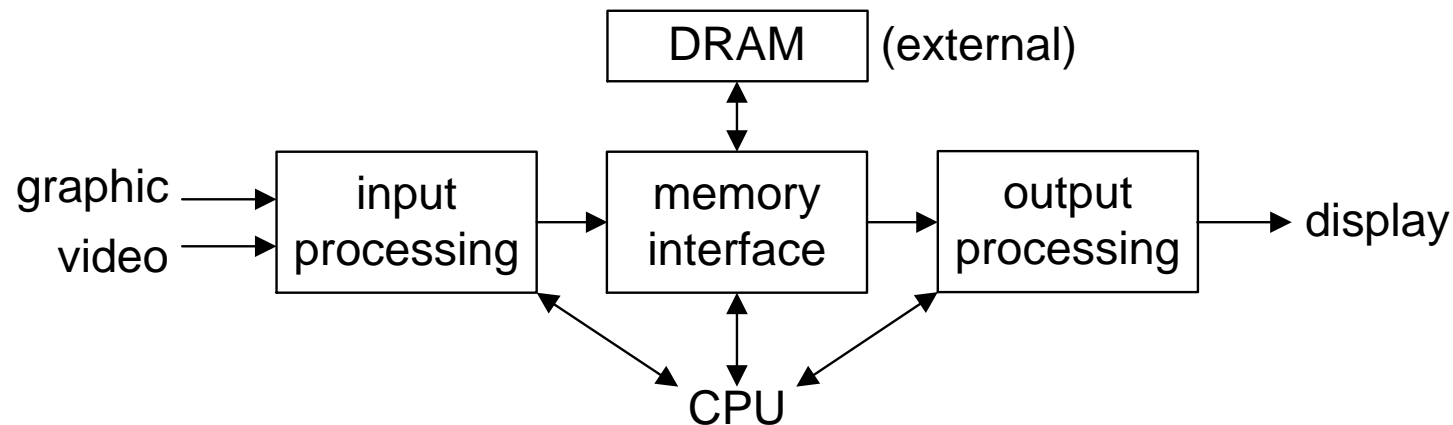
(image: <http://www.liesegang.de>)



# Top-Down Entwurf: Display-Controller für Beamer

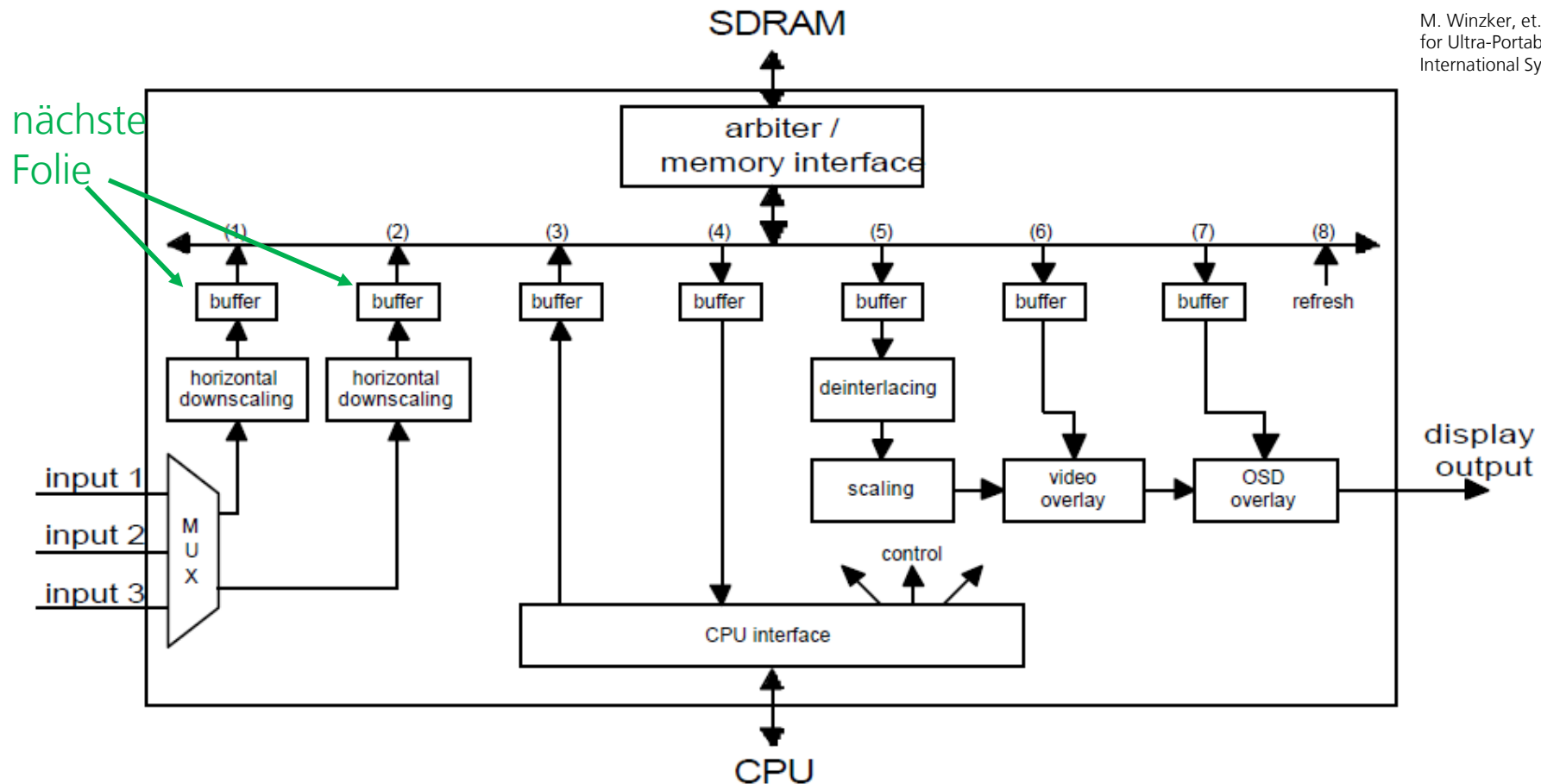
Aufteilung in zwei Untermodule

- CPU-System für Steuerung: Flexibel programmierbar, geringe Rechenleistung
- Signalverarbeitung: Vorgegebene Algorithmen, hohe Rechenleistung
  - ➔ Weitere Struktur der Signalverarbeitung
    - Filter am Eingang und/oder Ausgang für Skalierung, Deinterlacing
    - Bildspeicher für Einfrieren des Bildes
    - Bildspeicher benötigt 2 Bilder (Wechselpuffer), 1280x1024 Pixel, 3 Farben, 8 Bit pro Farbe
      - ➔ Externer Bildspeicher: DRAM



# Top-Down Entwurf: Signalverarbeitung (II)

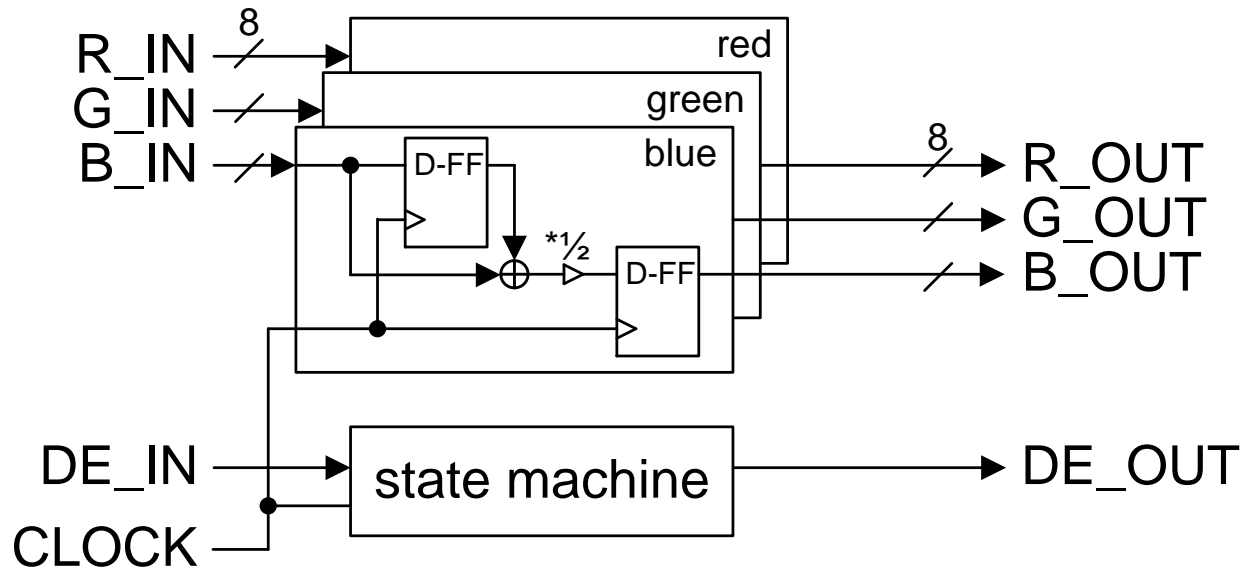
- Weitere Konkretisierung der Teilschaltungen
- Entwurf der Untermodule, z.B. Skalierung als Filter



M. Winzker, et.al, "Integrated Display Architecture for Ultra-Portable Multimedia Projectors," SID International Symposium, 2000.

# Top-Down Entwurf: Horizontal Downsampling

- Extrem große Bilder sollen vor der Speicherung um dem Faktor zwei herunterskaliert werden
  - Besser das Bild wird mit schlechter Qualität angezeigt, als gar nicht
- Horizontal: Mittelwert aus zwei benachbarten Bildpunkten
- Vertikal: Jede zweite Zeile wird ausgelassen
  - ➔ Speicher für Bildzeilen wird gespart
- Automat („state machine“) lässt „data enable“ für jedes zweite Pixel und jede zweite Zeile aus



# VHDL-Beschreibung: Downscale

- Ein Modul wird in VHDL in zwei Teilen beschrieben
  - **Entity:** Die **äußere Schnittstelle** eines Moduls, also Eingangssignale und Ausgangssignale
  - **Architecture:** Die **Funktion** eines Moduls

```
entity downscale is
  port ( clk      : in  std_logic;
         reset    : in  std_logic;
         de_in    : in  std_logic;
         r_in     : in  std_logic_vector(7 downto 0);
         g_in     : in  std_logic_vector(7 downto 0);
         b_in     : in  std_logic_vector(7 downto 0);
         de_out   : out std_logic;
         r_out    : out std_logic_vector(7 downto 0);
         g_out    : out std_logic_vector(7 downto 0);
         b_out    : out std_logic_vector(7 downto 0));
end downscale;
```

# Architecture: Definition der internen Signale

## Signaltypen

- **std\_logic**: Digitale Signal (einzelne „Leitung“ oder „Bit“)
  - 9 Werte: ‚0‘ und ‚1‘, sowie hochohmig (‚Z‘), unbekannt (‚X‘), ...
- **std\_logic\_vector**: Bus aus mehreren std\_logic (siehe entity)
- **integer**: Gut für Arithmetik geeignet
  - Soll hier im Praktikum verwendet werden
  - Durch Angabe des Wertebereichs wird Wortbreite bei Synthese bestimmt
    - In der Simulation wird eine Bereichsüberschreitung gemeldet
    - In der Schaltung bleibt Bereichsüberschreitung **unentdeckt** 💣\*

```
architecture behave of downscale is

    signal r,    g,    b           : integer range 0 to 255;
    signal r_1,  g_1,  b_1         : integer range 0 to 255;
    signal r_2,  g_2,  b_2         : integer range 0 to 255;
    signal de_q, de_pixel, de_line : std_logic;

begin
    [...]
```

# Architecture: Umwandlung der Datentypen

- Das Interface der Schaltung ist std\_logic und std\_logic\_vector
- Die Arithmetik erfolgt in integer
  - ➔ Umwandlung am Beginn und Ende der Architecture
- Paket für Arithmetik: „use IEEE.NUMERIC\_STD.ALL;“
- Umwandlung in zwei Schritten, um zu definieren, dass std\_logic\_vector kein Vorzeichen enthält (Dualzahl)

Hier zur Verdeutlichung:

**integer range 0 to 255**

**std\_logic\_vector(7 downto 0)**

```
architecture behave of downscale is
[...]
begin

r <= to_integer(unsigned(r_in));
g <= to_integer(unsigned(g_in));
b <= to_integer(unsigned(b_in));

[Arithmetik mit integer]

r_out <= std_logic_vector(to_unsigned(r_2, 8));
g_out <= std_logic_vector(to_unsigned(g_2, 8));
b_out <= std_logic_vector(to_unsigned(b_2, 8));

end behave;
```



# Architecture: Arithmetik

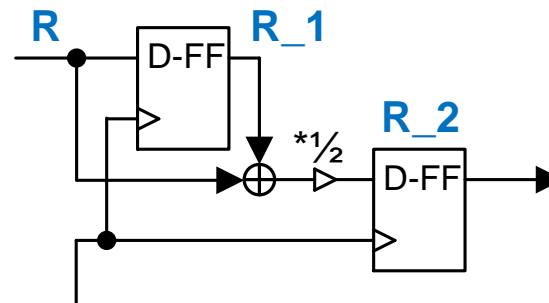
- Mit den Integer-Werten können die Rechenoperationen durchgeführt werden
  - Addition und Subtraktion: Standardoperationen
  - Multiplikation: Möglich aber aufwändig
  - Division: Möglich aber sehr aufwändig
    - Synthese der Division nicht bei allen Synthese-Programmen
- Besonderheit: Multiplikation und Division mit festen Werten
  - Multiplikation und Division mit Zweierpotenzen (2, 4, 8, 16, ...) sehr einfach
    - Entspricht Verschieben des Vektors
  - Multiplikation mit festem Wert durch Addition möglich
    - $5*A = 4*A + A$

## Downscaler

- Mittelwert aus zwei Werten berechnen

$$r\_2 \leq (r + r\_1) / 2;$$

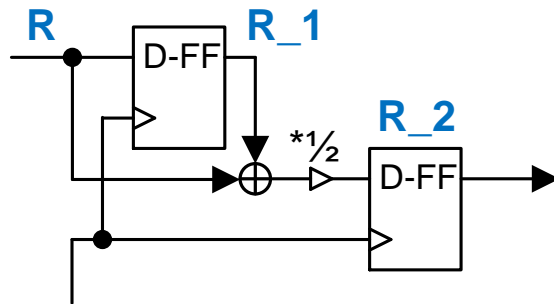
(Beschreibung der FFs folgt)



# Architecture: Prozess

- Synchrone Verarbeitung (mit Flip-Flops) wird in getaktetem Prozess beschrieben
- Jede Signalzuweisung erzeugt ein oder mehrere Flip-Flops
- **Wichtige Besonderheit:** 💣
  - Datenübernahme erst zum Ende des Prozesses
  - Bis dahin hat ein Signal noch den vorherigen Wert
  - Siehe: „r“ und „r\_1“

➔ Erforderlich für  
Parallelverarbeitung der Prozesse



```
architecture behave of downscale is
[...]  
begin  
  
process  
begin  
    wait until rising_edge(clk);  
  
    -- delay FFs  
    r_1 <= r;  
    g_1 <= g;  
    b_1 <= b;  
  
    -- arithmetic  
    r_2 <= (r + r_1)/2;  
    g_2 <= (g + g_1)/2;  
    b_2 <= (b + b_1)/2;  
  
[...]  
end process;  
[...]  
end behave;
```

# Architecture: Steuersignale

- Steuersignale können verknüpft werden durch
  - if-Abfrage
  - case-Statement

## Downscaler

- Verwendung für „state machine“
- Data Enable wird für Zeile und Spalte abwechselnd geschaltet
- Berechnung durch zwei Signale „de\_line“ und „de\_pixel“
  - de\_pixel wird invertiert, wenn de aktiv ist
  - de\_line wird invertiert, am Beginn jeder Zeile (de ist ,1‘ und war im letzten Takt ,0‘)
- Reset aller Signale bei Start

```
[...]
process
begin
    wait until rising_edge(clk);
[...]
```

```
    -- state machine
    if (reset = '1') then
        de_q      <= '0';
        de_line   <= '0';
        de_pixel  <= '0';
        de_out    <= '0';
    else
        de_q <= de_in;
        if (de_in = '1') then
            de_pixel <= not de_pixel;
        end if;
        if ( (de_in = '1') and
              (de_q = '0') ) then
            de_line <= not de_line;
        end if;
        de_out <= de_pixel and de_line;
    end if; -- reset
[...]
```

# Downscaler

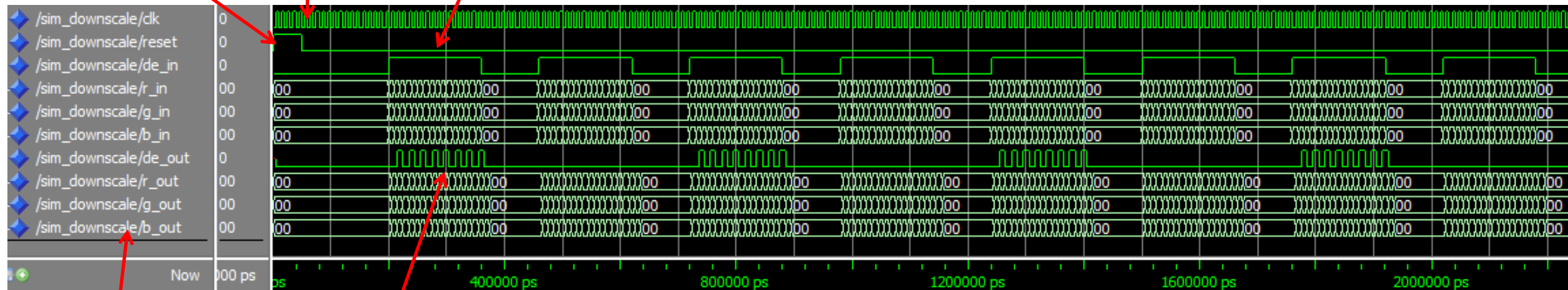
- Simulation zeigt Verhalten des Downscaler
- Simulation von 8 Zeilen mit jeweils 16 Bildpunkten
  - Dateien für Modul und Testbench auf LEA, Ordner „Zusatzmaterial“

## Screenshot des Simulationsfensters

Reset

Takt

Eingang: Data Enable für 8 Zeilen

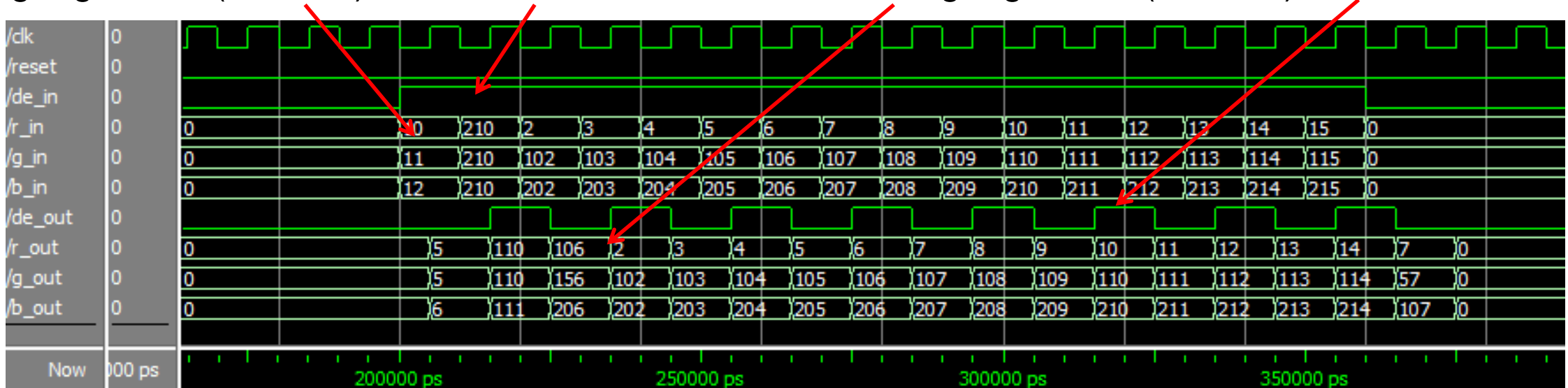


# Downscaler (II)

- Zoom in Simulation der ersten Zeile
  - Werte für R, G, B werden in Testbench vorgegeben
  - Bei Inaktivität setzt Testbench Datenwerte auf 0
- Erste Datenwerte für RGB sind 10, 11, 12 dann 210 (jeweils Dezimalzahlen)
  - Zunächst ungültige Ausgabe von 5 bzw. 6 durch Mittelung mit 0
  - Dann korrekte Ausgabe von 110 bzw. 111
  - Data Enable ist bei korrektem Wert auf '1', d.h. Timing stimmt

Eingangswerte (dezimal) und Data Enable

Ausgangswerte (dezimal) und Data Enable



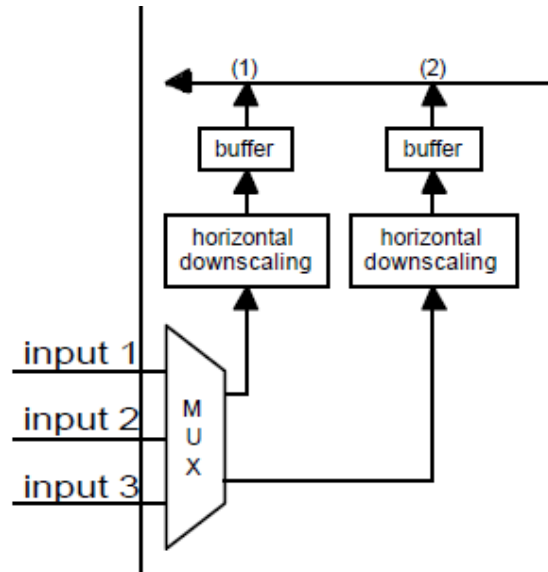
# Bottom-Up-Entwurf

- Synthese des Untermoduls
  - Syntaxcheck und Flächenabschätzung
    - Downscale benötigt 52 FFs und 31 LUTs
  - Überprüfung der Laufzeit möglich
    - Hier unkritisch, da wenig Logik zwischen Flip-Flops
    - Delay unter 2 ns (kann sich erhöhen, falls weitere Logik auf FPGA)
  - Keine Implementierung auf realem FPGA
    - Keine Pinzuweisung erforderlich
- Die Untermodule werden **Bottom-Up** bis zur Gesamtschaltung zusammengesetzt
  - Aufruf als Modul in VHDL

# Bottom-Up-Entwurf (II)

## Beispiel Display-Controller

- Zweifacher Aufruf
  - Mux-Ausgang A nach Buffer 1
  - Mux-Ausgang B nach Buffer 2



M. Winzker, et.al, "Integrated Display Architecture for Ultra-Portable Multimedia Projectors," SID International Symposium, 2000.

```
[...]
I_DOWN_A: entity work.downscale
    port map (      clk      => clk_a,
                    reset    => reset,
                    de_in    => de_mux_a,
                    r_in     => r_mux_a,
                    g_in     => g_mux_a,
                    b_in     => b_mux_a,
                    de_out   => de_buffer_1,
                    r_out    => r_buffer_1,
                    g_out    => g_buffer_1,
                    b_out    => b_buffer_1);
```

```
I_DOWN_B: entity work.downscale
    port map (      clk      => clk_b,
                    reset    => reset,
                    de_in    => de_mux_b,
                    r_in     => r_mux_b,
                    g_in     => g_mux_b,
                    b_in     => b_mux_b,
                    de_out   => de_buffer_2,
                    r_out    => r_buffer_2,
                    g_out    => g_buffer_2,
                    b_out    => b_buffer_2);
```

```
[...]
```

# Erweiterung Downscaler

- Eigentlich fehlt noch ein Steuereingang „active“, mit dem die Downscalierung ein/ausgeschaltet werden kann

## Übungsaufgabe

- Erweitern Sie das Modul, so dass bei
  - active=,1‘ die oben gezeigte Downskalierung erfolgt
  - active=,0‘ das Bildsignal unverändert weitergegeben wird
- Das Bildsignal darf bei active=,0‘ verzögert sein, aber die Zuordnung von Bildpunkten und „de\_out“ muss natürlich passen
- Der Ausgang des Modul sollte weiterhin aus einem getakteten Prozess erfolgen
- Gehen Sie davon aus, dass sich das Steuersignal „active“ nur in der vertikalen Austastlücke zwischen zwei Bildern ändert
- Verifizieren Sie das Verhalten für beide Werte von active
  - Erweitern Sie die Testbench so, dass nacheinander beide Fälle simuliert werden



## 8.2 Weitere Sprachelemente von VHDL

- VHDL bietet drei Möglichkeiten zur Darstellung von Werten:
  - Signal        `signal a : std_logic;`
  - Variable     `variable b : std_logic;`
  - Konstante    `constant c : std_logic := '0';`
- Für jede dieser Darstellungsarten können verschiedenen Datentypen gewählt werden, wie `std_logic`, `std_logic_vector`, `integer`, ...
- Eine **Variable** ist ähnlich einem Signal, unterscheidet sich jedoch in folgenden Eigenschaften:
  - Eine Variable ist nur innerhalb von Prozessen gültig
  - Eine Variable übernimmt einen zugewiesenen Wert sofort
    - Ein Signal wird erst am Ende einer Prozessschleife aktualisiert
  - ➔ Eine Variable kann wie ein Zwischenspeicher aufgefasst werden
- Zur Unterscheidung erfolgt die **Wertzuweisung** an Variable mit dem Symbol „:=“

# Variable

## Beispiel:

Addition mit Überlaufbegrenzung

- Bei der Verwendung von Signalen waren zwei Prozesse erforderlich  
(Auch die nebenläufige Signalzuweisung ist ein Prozess)
- Die Variable kann sofort nach der Signalzuweisung ausgewertet werden
- Durch die Benutzung von Variablen werden insbesondere komplexe Abläufe übersichtlicher

```
sum_9_i <= a_i + b_i;  
  
process(sum_9_i)  
begin  
    if (sum_9_i < 256) then  
        overflow <= '0';  
        sum <= std_logic_vector(  
            to_unsigned(sum_9_i,8));  
    else  
        overflow <= '1';  
        sum <= std_logic_vector(  
            to_unsigned(255,8));  
    end if;  
end process;
```

Deklaration der Variablen

```
process(a_i, b_i)  
    variable sum_9_i : integer range 0 to 511;  
begin  
    sum_9_i := a_i + b_i;  
    if (sum_9_i < 256) then  
        overflow <= '0';  
        sum <= std_logic_vector(  
            to_unsigned(sum_9_i,8));  
    else  
        overflow <= '1';  
        sum <= std_logic_vector(  
            to_unsigned(255,8));  
    end if;  
end process;
```

Wertzuweisung  
mit „:=“

# Vergleich von Variable und Signal

- Oft können sowohl Signale als auch Variablen verwendet werden
- Allerdings müssen die Eigenschaften der Signaldarstellungen beachtet werden
- Ähnlicher Code kann zu anderem Verhalten führen

**Signal:** Wertzuweisung wird am Ende der Prozessschleife gültig

```
signal data_sig : integer
           range 0 to 3;

process
begin
    wait until rising_edge(clk);
    data_sig <= data_sig + 1;
    if (data_sig = 3) then
        data_sig <= 0;
    end if;
end process;
```

clk 


data\_sig 

**Variable:** Wertzuweisung wird sofort gültig

```
process
    variable data_var : integer
           range 0 to 3;

begin
    wait until rising_edge(clk);
    data_var := data_var + 1;
    if (data_var = 3) then
        data_var := 0;
    end if;
end process;
```

clk 

data\_var 

# Bedeutung von Variablen

- Variablen werden für reale VHDL-Beschreibungen häufig verwendet
- Durch Variablen kann innerhalb von Prozessen ein sequentielles Verhalten erzielt werden
  - Dies ähnelt gebräuchlichen Programmiersprachen, z.B. ,C‘
  - Beispiel: Addition mit Überlaufbegrenzung (siehe oben)
    - Zunächst Addition
    - Danach Abfrage auf Überlauf
- Bei einfachen Beispielen werden die Vorteile von Variablen nicht unbedingt deutlich
  - ➔ Schauen Sie sich, wenn Sie intensiv mit VHDL arbeiten, die Möglichkeiten von Variablen erneut an



# Übungsaufgaben

## Aufgabe 8-1-a

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich das **Signal** „count“?

```
signal count : integer range 99 downto 0;
```

```
...
```

```
process
```

```
begin
```

```
    wait until rising_edge(clk);
```

```
    if count = 7 then
```

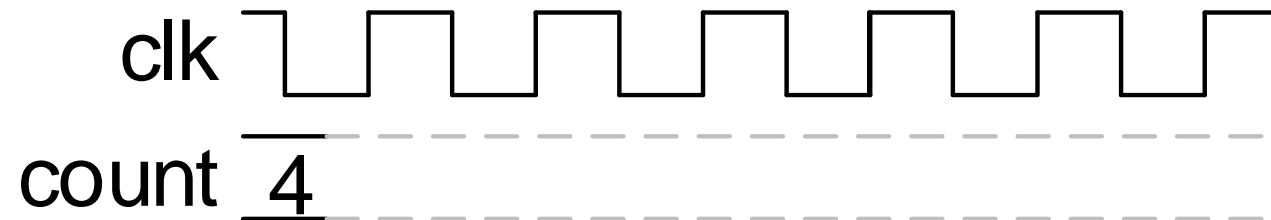
```
        count <= 0;
```

```
    else
```

```
        count <= count + 1;
```

```
    end if;
```

```
end process;
```

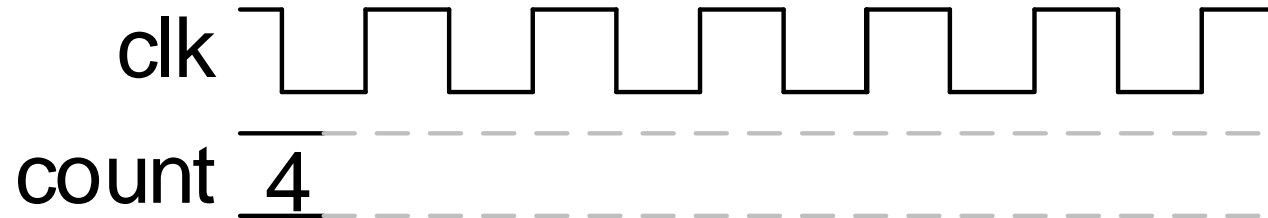


# Übungsaufgaben

## Aufgabe 8-1-b

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich die **Variable** „count“? Vergleichen Sie das Ergebnis mit Aufgabe 8-1-a.

```
process
  variable count : integer range 99 downto 0;
begin
  wait until rising_edge(clk);
  if count = 7 then
    count := 0;
  else
    count := count + 1;
  end if;
  ...
end process;
```



# Übungsaufgaben

## Aufgabe 8-2-a

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich das **Signal** „count“?

```
signal count : integer range 99 downto 0;
```

```
...
```

```
process
```

```
begin
```

```
    wait until rising_edge(clk);
```

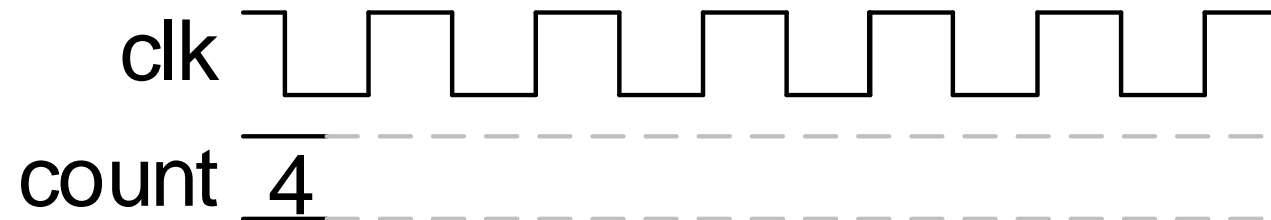
```
    count <= count + 1;
```

```
    if count = 7 then
```

```
        count <= 0;
```

```
    end if;
```

```
end process;
```

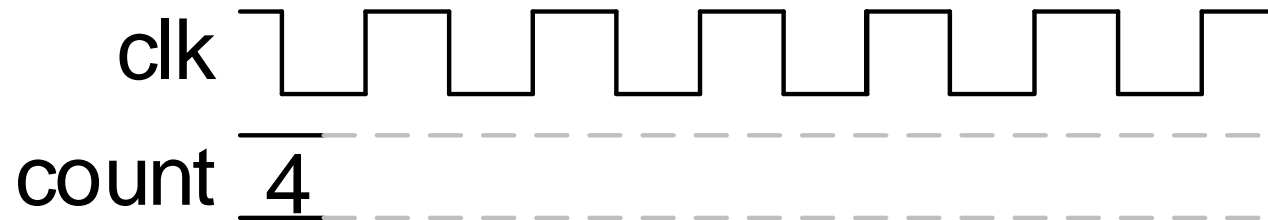


# Übungsaufgaben

## Aufgabe 8-2-b

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich die **Variable** „count“? Vergleichen Sie das Ergebnis mit Aufgabe 8-2-a.

```
process
  variable count : integer range 99 downto 0;
begin
  wait until rising_edge(clk);
  count := count + 1;
  if count = 7 then
    count := 0;
  end if;
  ...
end process;
```





# Übungsaufgaben

## Aufgabe 8-3-a

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich das **Signal** „count“?

```
signal count : integer range 99 downto 0;
```

```
...
```

```
process
```

```
begin
```

```
    wait until rising_edge(clk);
```

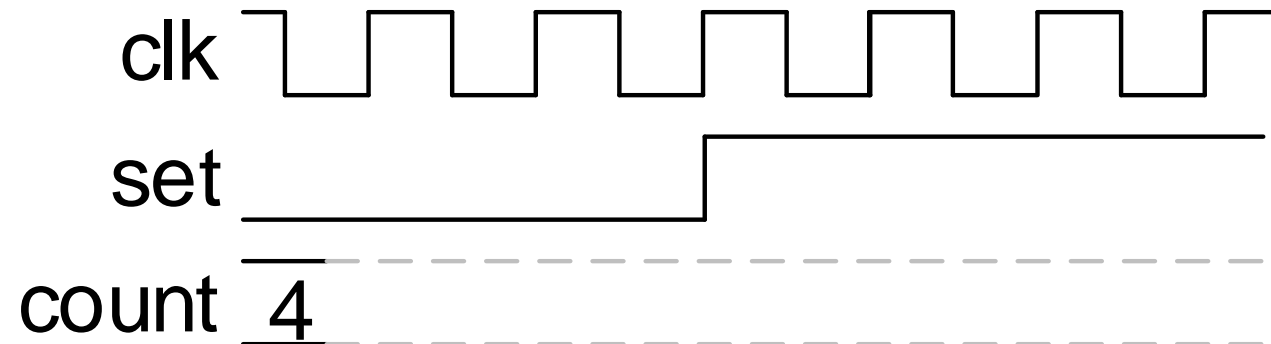
```
    if set = '1' then
```

```
        count <= 0;
```

```
    end if;
```

```
    count <= count + 1;
```

```
end process;
```

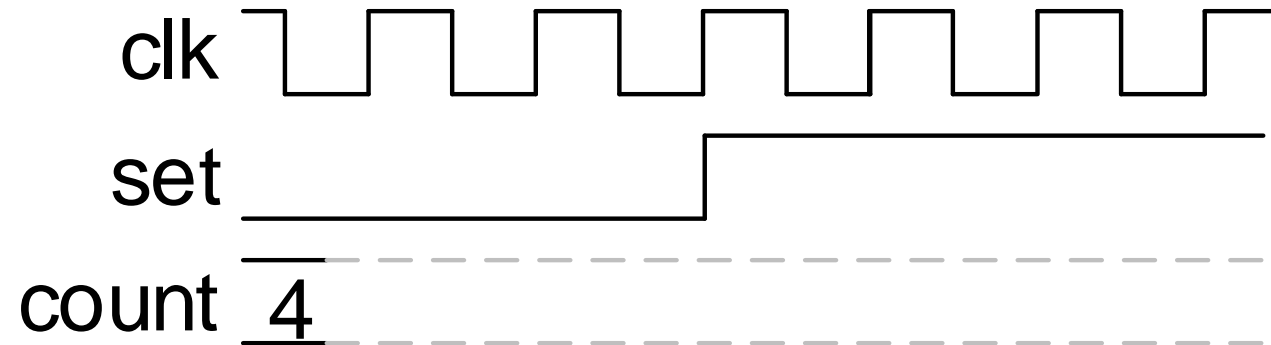


# Übungsaufgaben

## Aufgabe 8-3-b

Betrachten Sie den unten stehenden VHDL-Code. Wie verhält sich die **Variable** „count“? Vergleichen Sie das Ergebnis mit Aufgabe 8-3-a.

```
process
  variable count : integer range 99 downto 0;
begin
  wait until rising_edge(clk);
  if set = '1' then
    count := 0;
  end if;
  count := count + 1;
  ...
end process;
```



# Übungsaufgaben

## Aufgabe 8-4

Betrachten Sie den unten stehenden VHDL-Code. Wie verhalten sich die **Signale**?

```
signal a, a_new, a_old, pulse: std_logic;
```

```
...
```

```
process
```

```
begin
```

```
    wait until rising_edge(clk);
```

```
    a_new <= a;
```

```
    a_old <= a_new;
```

```
    if ((a_old='0') and  
        (a_new='1')) then
```

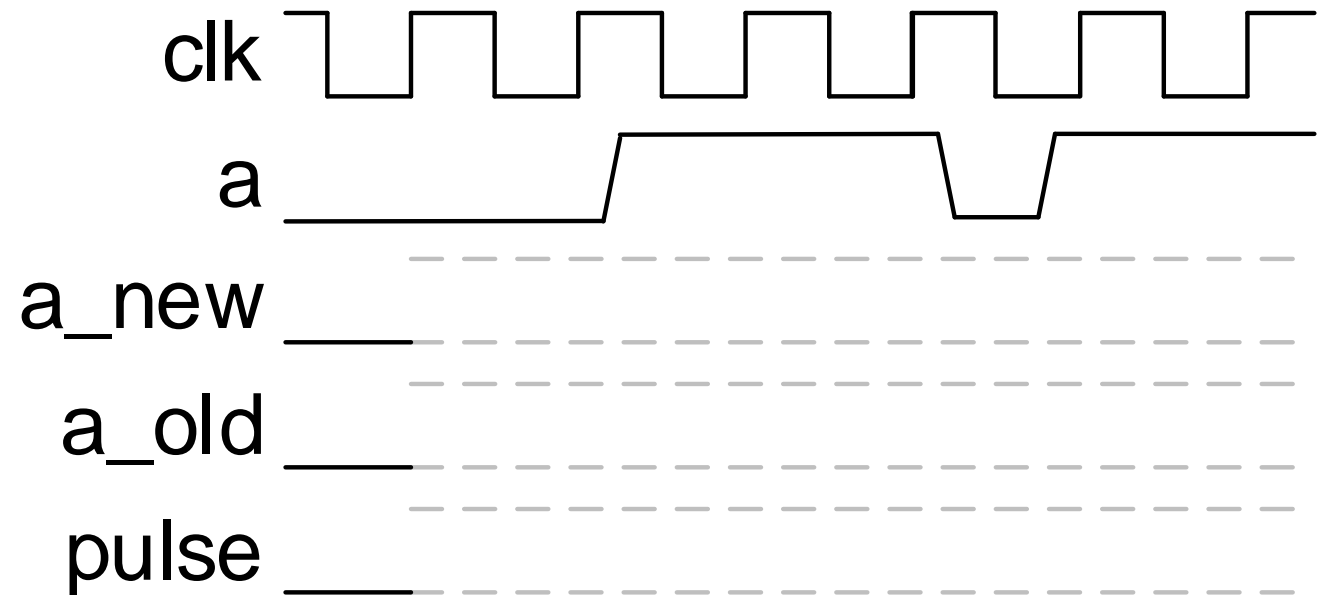
```
        pulse <= '1';
```

```
    else
```

```
        pulse <= '0';
```

```
end if;
```

```
end process;
```



# Function und Procedure

- Die Strukturierung einer Schaltung kann über **Untermodule** erfolgen
- Zusätzlich gibt es die beiden Möglichkeiten
  - **Function**: Aufruf mit mehreren Parametern, ein Rückgabewert
  - **Procedure**: Mehrere Rückgabewerte möglich, Parameter sind „in“ oder „out“
- Die Definition erfolgt **„lokal“** in einer Architecture oder **„global“** in einem Package
  - Package sinnvoll, wenn mehrfach in einem Projekt benötigt

## Beispiel aus Praktikum „Lane Detection“: lane\_g\_matrix.vhd

- RGB Daten als 24 Bit std\_logic\_vector
- Berechnung der Luminanz Y, integer 12 Bit

```
function rgb2y (vec : std_logic_vector(23 downto 0))  
  return integer is  
  variable result : integer range 0 to 4095;  
begin  
  result := 5*to_integer(unsigned(vec(23 downto 16)))  
           + 9*to_integer(unsigned(vec(15 downto 8)))  
           + 2*to_integer(unsigned(vec(7 downto 0)));  
  return result;  
end function;
```

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$0.299 \cdot 16 = 4.78 \approx 5$$

$$0.587 \cdot 16 = 9.39 \approx 9$$

$$0.114 \cdot 16 = 1.82 \approx 2$$

$$\rightarrow Y = 5 \cdot R + 9 \cdot G + 2 \cdot B$$

## 8.3 Verwendung spezieller Funktionsblöcke

- Aktuelle FPGAs enthalten spezielle Funktionsblöcke, die für die Schaltungsentwicklung genutzt werden können, z.B.:
  - Speicher
  - Multiplizierer
  - Taktaufbereitung
- Informationen zu diesen Funktionsblöcken finden sich in den Datenblättern und „Application Notes“ der Hersteller

Die Funktionsblöcke können eingebunden werden:

- Durch „**Inferring**“, d.h. das CAD-Tool erkennt aus der VHDL-Beschreibung, dass ein Funktionsblock sinnvoll ist
  - Beispiel Multiplizierer:  $c \leq a * b;$
  - **Vorteil:** Der Code ist portabel (zu anderen FPGAs und zu ASICs)
- Durch **Aufruf** eines speziellen **Untermoduls**
  - Für Funktionsblöcke, die in VHDL schwierig zu beschreiben sind, z.B. Taktverdopplung
  - **Vorteil:** Die gewünschte Schaltung kann genau spezifiziert werden

# RAM-„Inferring“

- RAM mit 256 Worten zu 16 bit
- Datentyp „array“ wird zu RAM synthetisiert

## Hinweise:

- Zum Zugriff auf das Array wird die Adresse nach Integer konvertiert
- Es ist nicht garantiert, dass jedes Synthese-Programm den Code gleich umsetzt
  - Z.B. Umsetzung mit FFs und Gattern
- Aber die **Funktion** der Schaltung sollte stets gleich sein
- Altera-Dokument: „Recommended HDL Coding Styles“

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram_256x16 is
    port( clk      : in  std_logic;
          data_in   : in  std_logic_vector(15 downto 0);
          addr      : in  std_logic_vector( 7 downto 0);
          write_en  : in  std_logic;
          data_out  : out std_logic_vector(15 downto 0));
end ram_256x16;

architecture behave of ram_256x16 is
    type ram_array is array (0 to 255) of
        std_logic_vector(15 downto 0);
    signal ram : ram_array;
begin

    process
        variable addr_integer : integer range 255 to 0;
    begin
        wait until rising_edge(clk);
        addr_integer := to_integer(addr);
        if (write_en = '1') then
            ram(addr_integer) <= data_in;
        end if;
        data_out <= ram(addr_integer);
    end process;

end behave;
```



# Beispiel: Intel/Altera MegaWizard

MegaWizard Plug-In Manager [page 4 of 12]

## RAM: 2-PORT

About Documentation

1 Parameter Settings 2 EDA 3 Summary

General > Widths/Blk Type > Clks/Rd, Byte En > Regs/Clocks/Adrs > Output1 > Mem Init >

**Altera\_RAM**

Block Type: AUTO

How many 8-bit words of memory? 256

☐ Use different data widths on different ports

Read/Write Ports

How wide should the 'q\_a' output bus be? 8

How wide should the 'data\_a' input bus be? 8

How wide should the 'q' output bus be? 8

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

☒ Auto ☐ MLAB ☐ M9K

☐ M144K ☐ LCs Options...

Set the maximum block depth to Auto words

Resource Usage

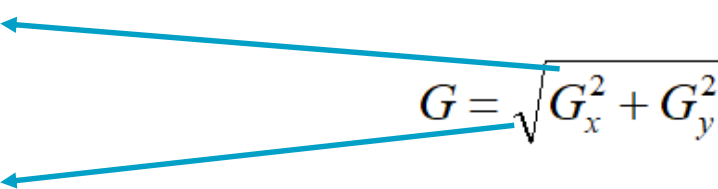
...

Cancel < Back Next > Finish

(Screenshot: Intel/Altera MegaWizard)

# Spezielle Funktionsblöcke im Praktikum

- Speicher in **lane\_linemem.vhd**
  - RAM-Infering, ähnlich wie zwei Folien zuvor
  - Kein Adresseingang, sondern zyklischer Speicher mit 1280 Speicherplätzen
  - Bezeichnung: FIFO-Speicher, First-In-First-Out
- Multiplizierer zum Quadrieren in **lane\_g\_matrix.vhd**
  - Inferring: `data_out <= sum*sum;`
- ROM in **lane\_sobel.vhd**
  - Quadratwurzel für Sobel-Filter
  - **Aufruf** des speziellen **Untermoduls** **lane\_g\_root\_IP.vhd**
  - IP-Modul „altsyncram“ des Intel/Altera Megawizard
    - IP = Intellectual Property
  - Modul für Cyclone IV und V geeignet
    - Prinzipiell wären verschiedene Module für verschiedene FPGAs erforderlich
  - ROM-Inhalt spezifiziert durch **lane\_g\_root.mif**


$$G = \sqrt{G_x^2 + G_y^2}$$



# Spezielle Funktionsblöcke im Praktikum (II)

## Berechnung des ROM-Inhalts

- Zusammenfassung von drei Operationen
  - Wurzel
  - Begrenzung auf 8 bit
  - Invertierung (damit Kanten dunkel auf hellem Grund erscheinen)
- Berechnung in Tabellenkalkulation
  - Wortbreite des C-Programms muss nachgebildet werden
  - Wurzel SQRT
  - Abrunden  
ROUNDDOWN
  - Invertierung:  $255 - X$
  - Begrenzung: MAX
- Header mit Konfiguration des ROM

C7							=MAX(0;255-ROUNDDOWN(SQRT(A7*32)/2;0))
	A	B	C	D	E	F	
1	DEPTH = 8192;			-- The size of memory in words			
2	WIDTH = 8;			-- The size of data in bits			
3	ADDRESS_RADIX = DEC;			-- The radix for address values			
4	DATA_RADIX = DEC;			-- The radix for data values			
5	CONTENT			-- start of (address : data pairs)			
6	BEGIN						
7	0 :		255;				
8	1 :		253;				
9	2 :		251;				
10	3 :		251;				