Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

HARDWARE / SOFTWARE-SCHNITTSTELLEN

Hardwareentwurf mit VHDL

23. Juni 2014
(Revision: 1341)

Prof. Dr. Steffen Reith

Theoretische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**

BASISSCHALTKREISE

Notizen

Notizen

KOMBINATORISCHE SCHALTKREISE

Kombinatorische Schaltkreise enthalten **keine Speicherelemente**. Deshalb hängt die Ausgabe ausschließlich von der Eingabe ab und sie liefern bei **gleicher Eingabe** die **gleiche Ausgabe**.

Ein kombinatorischer Schaltkreis berechnet eine **Boolesche Funktion** f vom Typ $f\colon \{0,1\}^n \to \{0,1\}^m$.

In einem kombinatorischen Schaltkreis gibt es **keine** (zeitverzögerten) **Rückkopplungen**, d.h. fasst man die Schaltung als Graph auf, so ist dieser **kreisfrei**.

Notizen

MULTIPLEXER

Will man aus einer Menge von Signalen genau eines **auswählen** und dann weitergeben, so verwendet man einen Multiplexer:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity MUX4to1 is
5    generic (W : integer := 8);
6
7    port (i0, i1, i2, i3: in std_logic_vector(W - 1 downto 0);
8          sel : in std_logic_vector(1 downto 0);
9          o : out std_logic_vector(W - 1 downto 0));
10 end entity;
11 architecture Behavioral of MUX4to1 is
12 begin
13   with sel select
14     o <= i0 when "00",
15          i1 when "01",
16          i2 when "10",
17          i3 when others; -- Warum reicht "11" nicht?
18 end architecture;
```


Notizen

PRIORITÄTSENCODER

Ein **Prioritätsencoder** ermittelt die **Nummer** des aktiven Eingangs mit der **höchsten Priorität**. Eine Anwendung könnte die **Auswahl eines Interrupts** sein.

Implementiert werden soll ein Prioritätsencoder mit vier Request-Eingängen, wobei die Nummer des Eingangs die Priorität angibt:

```
1  entity pEncoder4 is
2  port (req : in std_logic_vector(4 downto 1);
3        idx : out std_logic_vector(2 downto 0));
4  end pEncoder4;
5
6  architecture Behavioral of pEncoder4 is
7  begin
8    idx <= "100" when (req(4)='1') else
9           "011" when (req(3)='1') else
10          "010" when (req(2)='1') else
11          "001" when (req(1)='1') else
12          "000";
13 end architecture;
```


Notizen

DECODER

Will man, abhängig von einer Binärzahl, genau ein Signal aktivieren, so wird dieser Schaltkreis **Decoder** bezeichnet.

Eine typische Anwendung ist die Implementierung von **Memory-Mapped I/O** oder die Erzeugung von **enable**-Signalen für eine gemultiplexte Ansteuerung der Siebensegmentanzeigen des Nexys-Boards.

```
1  entity Decoder4 is
2  port (idx : in std_logic_vector(2 downto 0);
3          o : out std_logic_vector(7 downto 0));
4  end Decoder4;
5
6  architecture Behavioral of Decoder4 is
7  begin
8    o <= "00000000" when "000" | "001" | "010" | "011",
9         "01000000" when "100",
10        "10000000" when "101",
11        "11000000" when others; -- inkl. "111"
12 end architecture;
```


Notizen

SYNCHRONES DESIGN

Sequentielle Schaltkreise benutzen **internen Speicher**, d.h. die Ausgabe hängt **nicht nur** von der Eingabe ab.

Um die Entwicklung von Schaltkreisen zu **vereinfachen** (z.B. den Clock Skew zu entschärfen, Vermeidung von Glitches), verwendet man das **synchrone Design**.

Bei der synchronen Methode werden **alle Speicherelemente** durch einen **globalen Takt kontrolliert** / **synchronisiert**. Alle **Berechnungen** werden an der steigenden (und/oder) fallenden **Flanke des Taktes** vorgenommen.

Das synchrone Design ermöglicht den Entwurf, Test und die Synthese von **großen** Schaltkreisen mit marktüblichen Tools. Aus diesem Grund wird in der Vorlesung diese Designmethode verwendet werden.

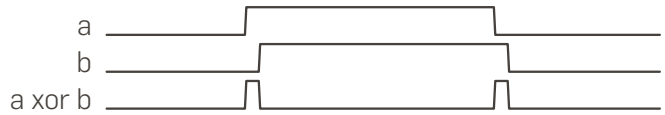
Notizen

SYNCHRONES DESIGN (II)

Unter **Clock Skew** versteht man den Versatz / Verzögerung von Taktsignalen durch **Laufzeitunterschiede**. Mit Hilfe von speziellen **Taktverteilungsnetzwerken** wird in einem FPGA (oder anderen synchronen Schaltkreisen) der Clock Skew **minimiert**.

Aus diesem Grund sollte keine (kombinatorische) Logik im Taktpfad sein, da dies zu ungleichmäßigen Laufzeiten führen kann.

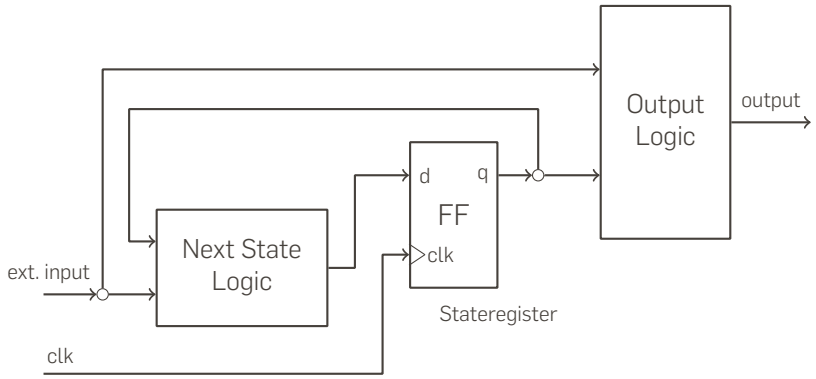
Aufgrund unterschiedlicher Laufzeiten in den Signalpfaden und/oder Gattern können Signale **zeitlich verschoben** sein. Dabei kann es zu **ungewollten / ungültigen** sehr kurzen Signaländerungen kommen (engl. **Glitches**):



Notizen

SYNCHRONE SCHALTKREISE

Die Struktur von synchronen Schaltkreisen ist idealisiert wie folgt aufgebaut:



Notizen

BEISPIEL: FLIP-FLOP

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity ff is
5   port (  clk : in std_logic;
6           d : in std_logic;
7           q : out std_logic);
8  end entity;
9  architecture Behavioral of FF is
10 begin
11   ff : process (clk)
12   begin
13   if (rising_edge(clk)) then -- Warte steigende Flanke
14     q <= d;  -- Eingang übernehmen
15   end if;
16  end process;
17 end architecture;
```

Hier ist die ›Next State Logic‹ und die ›Output Logic‹ trivial und es bleibt das nackte FlipFlop eines synchronen Systems. Da d nur an der Flanke übernommen wird, ist nur `clk` in der Sensitivitätsliste.

Notizen

REGISTER & COUNTER

Gruppiert man **mehrere** Flip-Flops zu einen Speicherelement, so erhält man ein **Register**.

Mit einem entsprechend breiten Register kann man einen **free-running counter** bauen, der die Basis für viele andere Schaltkreise bildet und einfach die Anzahl der **steigenden Flanken des Taktsignals zählt**.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity freeBinCounter is
6
7    generic(N : integer := 8);
8
9    port (clk   : in  std_logic;
10         reset : in  std_logic;
11         value : out std_logic_vector(N - 1 downto 0));
12
13 end entity;
```


Notizen

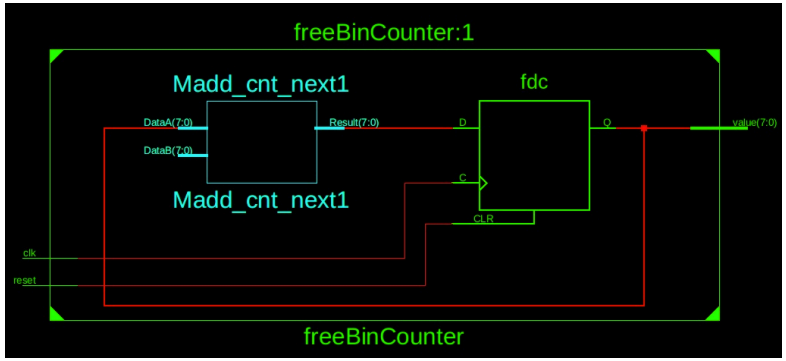
FREE-RUNNING COUNTER

```
1  architecture Behavioral of freeBinCounter is
2
3  signal cnt_reg  : std_logic_vector(N - 1 downto 0);
4  signal cnt_next : std_logic_vector(N - 1 downto 0);
5
6  begin
7   process (clk, reset)
8   begin
9     if (reset = '1') then
10      cnt_reg <= (others => '0');
11    elsif (rising_edge(clk)) then
12      cnt_reg <= cnt_next; -- Change state on rising edge
13    end if;
14  end process;
15
16  -- Next state logic
17  cnt_next <= std_logic_vector(unsigned(cnt_reg) + 1);
18
19  -- Output logic
20  value <= cnt_reg;
21 end architecture;
```


Notizen

DAS ERGEBNIS DER SYNTHESE

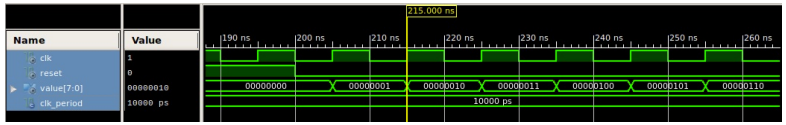
Synthetisiert man die Beschreibung des ›Free-running counter‹ mit Hilfe der Xilinx-Tools, so ergibt sich eine bekannte Struktur:



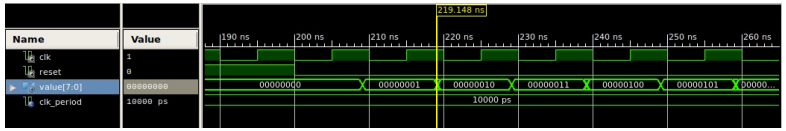
Notizen

THEORIE VS. PRAXIS

Eine einfache Simulation zeigt das erwartete Ergebnis:



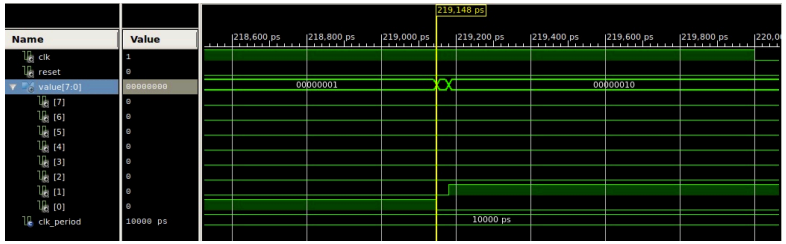
Die wirkliche (Post-Map) Welt sieht ein wenig anders aus



Notizen

THEORIE VS. PRAXIS (II)

Der gleiche Simulationslauf mit höherer Zeitauflösung:



Notizen

EIN UNIVERSELLER COUNTER

Die parallele Anweisung (mit dem entsprechenden Port)

```
1 maxTick <= '1' when (cnt_reg = (others => '1'))
2                else '0';
```

liefert einen Counter, der im letzten Takt vor dem **Überlauf** ein Flag liefert, das einen Takt lange 1 ist.

Baut man eine etwas kompliziertere ›Next State Logic‹ ein, so erhält man einen universellen Zähler, der **vorwärts** und **rückwärts zählen**, den man **anhalten** und **laden** kann.

```
1  cnt_next <= std_logic_vector(unsigned(cnt_reg) + 1)
2                when ((enable = '1') and (up = '1')) else
3
4              std_logic_vector(unsigned(cnt_reg) - 1)
5                when ((enable = '1') and (up = '0')) else
6
7              data when (load = '1') else
8
9              cnt_reg;
```


Notizen

EIN MODULO-COUNTER

Interessant ist ein Zählerbaustein, der **immer wieder** bis zu einem **vorgegebenen Wert** zählt. Ein Anwendung hierfür könnte die Erzeugung eines HSYNC-Signals einer VGA-Schnittstelle sein.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity modCnt is
6
7    generic(W : integer := 10; -- Width
8            M : integer := 800; -- Modulo
9            hsMin : integer := 656;
10           hsMax : integer := 751);
11
12   port(clk : in std_logic;
13        reset : in std_logic;
14
15        sync : out std_logic;
16
17        q : out std_logic_vector(W - 1 downto 0));
18 end modCnt;
```


Notizen

EIN MODULO-COUNTER (II)

```
1  architecture Behavioural of modCnt is
2    signal cnt_r : unsigned(W - 1 downto 0);
3    signal cnt_n : unsigned(W - 1 downto 0);
4  begin
5
6    process (clk, reset)
7    begin
8      if (reset = '1') then
9        cnt_r <= (others => '0');
10     elsif (rising_edge(clk)) then
11       cnt_r <= cnt_n; -- Change state on rising edge
12     end if;
13   end process;
14
15   -- Next state logic
16   cnt_n <= (others => '0') when (cnt_r = M - 1) else cnt_r + 1;
17
18   -- Output logic
19   q <= std_logic_vector(cnt_r);
20   sync <= '1' when ((cnt_r >= hsMin) and (cnt_r <= hsMax))
21               else '0';
22
23 end architecture;
```


Notizen

EIN SCHIEBEREGISTER

Oft sollen Daten **serialisiert**/**parallelisiert** werden. Eine Standardanwendung ist ein UART - Universal Asynchronous Receiver Transmitter (vgl. „serielle Schnittstelle“).

Hierzu verwendet man ein Schieberegister:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity shiftReg is
5
6    generic(N : integer := 8);
7
8    port(clk : in std_logic;
9         reset : in std_logic;
10
11        mode : in std_logic_vector(1 downto 0);
12
13        d : in  std_logic_vector(N - 1 downto 0);
14        q : out std_logic_vector(N - 1 downto 0));
15
16 end shiftReg;
```


Notizen

EIN SCHIEBEREGISTER (II)

```
1  architecture Behavioural of shiftReg is
2  signal dat_r, dat_n : std_logic_vector(N - 1 downto 0);
3  begin
4    process (clk, reset)
5    begin
6      if (reset = '1') then
7        dat_r <= (others => '0');
8      elsif (rising_edge(clk)) then
9        dat_r <= dat_n; -- Change state on rising edge
10     end if;
11   end process;
12
13   with mode select -- Next state logic
14     dat_n <= dat_reg when "00" -- NOP
15              d(0) & dat_r(N - 2 downto 0) when "01" -- Shift right
16              dat_r(N - 1 downto 1) & d(0) when "10" -- Shift left
17              d when "11" -- Load register
18
19   q <= data_r;  -- Output logic
20 end architecture;
```

Ein Aufzählungstyp für `mode` erhöht die Lesbarkeit!

Notizen

EINE ANWENDUNG VON SCHIEBEREGISTERN

In der **Praxis prellen mechanische Kontakte**, d.h. die Eingabe wechselt sehr schnell zwischen '1' und '0' und erst nach einiger Zeit bleibt die Eingabe stabil. Mit Schieberegistern kann man dieses Verhalten unterdrücken:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity debounce is
5    generic (preScale : natural := 10000);
6    port (clk    : in  std_logic;
7          input  : in  std_logic;
8          output : out std_logic;
9          alarm  : out std_logic);
10 end debounce;
11
12 architecture Behavioral of debounce is
13   signal preCnt : integer range 0 to preScale := 0;
14   signal shReg : std_logic_vector(3 downto 0) := (others => '0');
```


Notizen

EINE ANWENDUNG VON SCHIEBEREGISTERN (II)

```
1  begin
2    process (clk)
3    begin
4      if (rising_edge(clk)) then
5        if (preCnt = 0) then
6          preCnt <= preScale;
7          if (shReg = "0000") then output <= '0'; end if;
8          if (shReg = "1111") then output <= '1'; end if;
9          if ((shReg = "1000") or (shReg = "0111")) then
10           alarm <= '1';
11         else
12           alarm <= '0';
13         end if;
14         shReg <= shReg(2 downto 0) & input;
15       else
16         preCnt <= preCnt - 1;
17     end if;
18   end if;
19 end process;
20 end architecture;
```


Notizen

EIN GENERISCHES REGISTERFILE

In einem Prozessor oder anderen komplexen Schaltkreisen müssen oft mehrere Werte zwischengespeichert werden. Hierfür kann ein Registerfile verwendet werden.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity registerFile is
6
7    generic(wordSize := 32; -- number of bits in a register
8            adrSize  := 5); -- number of address bits
9
10   port(clk, reset : in std_logic;
11        writeEnable : in std_logic;
12        wAdr  : in std_logic_vector(adrSize - 1 downto 0);
13        wData : in std_logic_vector(wordSize - 1 downto 0);
14        rAdr  : in  std_logic_vector(adrSize - 1 downto 0);
15        rData : out std_logic_vector(wordSize - 1 downto 0));
16
17 end registerFile;
```


Notizen

EIN GENERISCHES REGISTERFILE (II)

```
1  architecture Behavior of registerFile is
2  type regFile_t is array (2**adrSize - 1 downto 0) of
3                    std_logic_vector(wordSize - 1 downto 0);
4  signal regFile : regFile_t;
5  begin
6    process (clk,reset)
7    begin
8      if (reset = '1') then
9        regFile <= (others => (others => '0')); -- Init the file
10     elsif (rising_edge(clk)) then
11       if (writeEnable = '1') then
12         regFile(to_integer(unsigned(wAdr))) <= wData;
13       end if;
14     end if;
15   end process
16
17   -- read port
18   rData <= regFile(to_integer(unsigned(rAdr)));
19 end architecture;
```

Leicht können weitere Read/Write-Ports hinzu gefügt werden.

Notizen

ENTPRELLUNG

Das Betätigen eines mechanischen Schalters / Tasters führt zu **ungewolltem kurzzeitigen Öffnen und Schließen** des Schaltkontakts.

Dieses Verhalten wird **prellen** genannt und kann zu einer falschen Erfassung von Schaltereignissen führen.

Der folgende Schaltkreis verwendet einen **binären Zähler**, um ein Eingabesignal erst dann weiter zu leiten, wenn klar ist, dass es sich um keine kurzzeitige Signaländerung handelt.

Dazu wird die folgende Schnittstelle verwendet:

```
1  entity debouncer is
2    -- Approx 10ms@100Mhz
3    generic (cntWidth : integer := 20);
4    port(clk    : in  std_logic;
5         sigRaw : in  std_logic;
6         sigDeb : out std_logic);
7  end debouncer;
```


Notizen

IMPLEMENTIERUNG: ENTPRELLUNG

```
1  architecture Behavioral of debouncer is
2    constant HIGH : std_logic_vector(w-1 downto 0) := (others => '1');
3
4    signal dCnt_r : std_logic_vector(w-1 downto 0) := (others => '0');
5    signal dCnt_n : std_logic_vector(w-1 downto 0);
6
7    signal deb_r : std_logic := '0';
8    signal deb_n : std_logic;
9
10 begin
11   state_logic : process(clk)
12   begin
13     -- Test auf steigende Flanke
14     if (rising_edge(clk)) then
15
16       -- Setze Zaehlerregister
17       dCnt_r <= dCnt_n;
18
19       -- Setze Register fuer entprelltes Signal
20       deb_r <= deb_n;
21     end if;
22   end process;
```


Notizen

IMPLEMENTIERUNG: ENTPRELLUNG (II)

```
1   next_state_logic : process(dCnt_r, sigRaw, deb_r)
2   begin
3
4     -- Teste entprelltes Signal und Eingabe auf Gleichheit
5     if (deb_r = sigRaw) then
6
7       -- Reset des Wartezaehlers
8       dCnt_n <= (others => '0');
9
10    else
11
12      -- Zähler erhoehen (prellen vermeiden)
13      dCnt_n <= std_logic_vector(unsigned(dCnt_r) + 1);
14
15    end if;
16
17  end process;
```


Notizen

IMPLEMENTIERUNG: ENTPRELLUNG (III)

```
1   debouncer : process(dCnt_r, sigRaw, deb_r)
2   begin
3
4     -- Test auf groessten Zaehlerwert
5     if (dCnt_r = HIGH) then
6
7       -- Eingabe merken
8       deb_n <= sigRaw;
9
10    else
11
12      -- Aktulles entprelltes Signal nicht aendern
13      deb_n <= deb_r;
14
15    end if;
16
17  end process;
18
19  -- Entprelltes Signal ausgeben
20  sigDeb <= deb_r;
21
22 end architecture;
```


Notizen

LATCHES IN SYNCHRONEN SCHALTKREISEN

Die `process`-Anweisung führt bei **unsachgemäßer Benutzung** zu Problemen, denn es können **unbeabsichtigt Speicherelemente** eingebaut werden („**Latches**“).

Der **Grund** hierfür ist, dass der VHDL-Standard garantiert, dass Signale die **keine Zuweisung** erfahren ihren **alten Wert** behalten.

Latches sind nicht flankengesteuert, sondern pegelabhängig. Dies kann zu **undefinierten Zuständen** führen. Kurz: **Latches** in synchronen Schaltkreisen immer **vermeiden**!

Kochrezept zur Vermeidung von Latches:

- → Alle (relevanten) Eingabesignale werden in der Sensitivitätsliste aufgeführt
- → Alle `if`-Anweisungen haben auch `else`-Zweige
- → Jedem Signal wird in jedem Zweig ein Wert zugewiesen (oder man arbeitet mit Defaultwerten)

Notizen

BEISPIEL: LATCHES

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity LL is
5    port(a : in std_logic;
6         b : in std_logic;
7
8         eq : out std_logic;
9         gt : out std_logic);
10 end LL;
11
12 architecture Behavioral of LL is
13 begin
14   process (a) -- b fehlt in der Sensitivitätsliste
15   begin
16     if (a > b) then -- eq bekommt keinen Wert
17       gt <= '1';
18     elsif (a = b) then -- gt bekommt keinen Wert
19       eq <= '1';
20     end if; -- Es gibt keinen else-Zweig
21   end process;
22 end architecture;
```


Notizen

BEISPIEL: LATCHES (II)

Bei der Synthese des letzten Beispiels erzeugen die Xilinx-Tools (ISE 14.7) die **folgenden Warnungen**:

xst	⚠	HDLCompiler:92 - "/home/streit/src/VHDL/Latch/LL.vhd" Line 28: b should be on the sensitivity list of the process	New
xst	⚠	HDLCompiler:92 - "/home/streit/src/VHDL/Latch/LL.vhd" Line 32: b should be on the sensitivity list of the process	New
xst	⚠	Xst:737 - Found 1-bit latch for signal <eq>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.	New
xst	⚠	Xst:1710 - FF/Latch <eq> (without init value) has a constant value of 1 in block <LL>. This FF/Latch will be trimmed during the optimization process.	New
xst	⚠	Xst:1710 - FF/Latch <eq> (without init value) has a constant value of 1 in block <LL>. This FF/Latch will be trimmed during the optimization process.	New

Notizen

Notizen