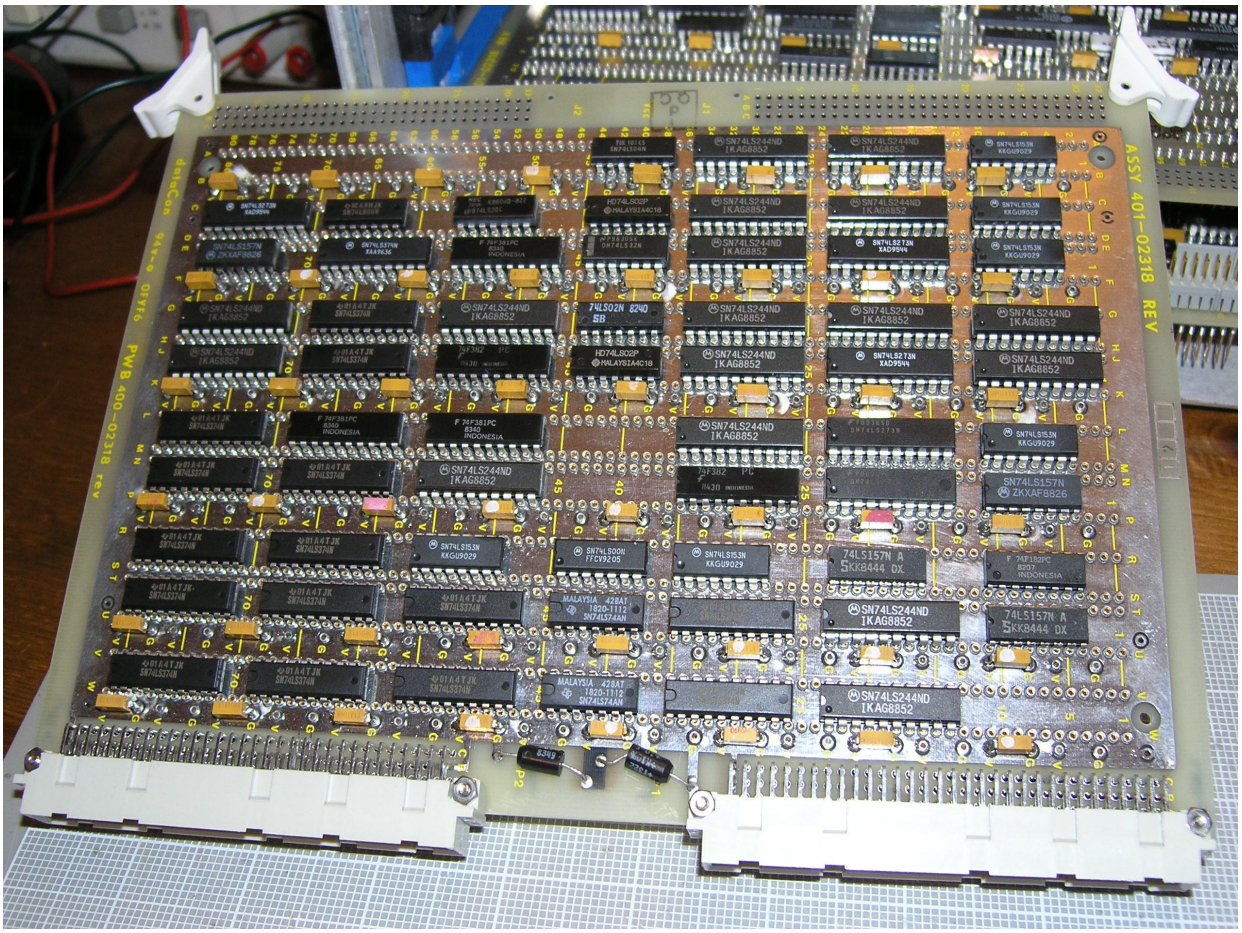
Entwurf digitaler Systeme



www.homebrewcpu.com – ALU in TTL-Logik realisiert

Auszug Lehrplan

Die Schülerinnen und Schüler können im III. Jahrgang, 6. Semester im Bereich Entwurf digitaler Systeme

Schaltwerke entwerfen und in programmierbaren Logikbausteinen implementieren.

1 Prozesse

Alles, was wir bislang beschrieben haben, wurde nebenläufig (sprich: parallel) umgesetzt. Das wird einen Schaltungsentwurf früher oder später einschränken… mit Sicherheit, wenn wir Schaltwerke (Automaten) entwerfen wollen/müssen/dürfen.

Mit VHDL können „Prozesse“ beschrieben werden – und das Besondere dabei ist, dass innerhalb eines Prozesses eine sequentielle Abarbeitung erfolgt. Somit stehen auch sequentielle Anweisungen wie Verzweigungs- und Schleifen-Konstrukte zur Verfügung. Man spricht hier von *sequential statements* innerhalb eines Prozesses. D.h. wie bei einer „herkömmlichen“ Programmiersprache werden diese Anweisungen sequentiell –d.h. nacheinander– abgearbeitet – und bei der Synthese wird dies auch HW-technisch so umgesetzt(!).

Ein Prozess beginnt mit dem VHDL Schlüsselwort **PROCESS**. Alle Prozesse einer ***ARCHITECTURE*** werden wiederum nebenläufig (d.h. parallel) abgearbeitet (hier ändert sich nichts). Die Kommunikation zwischen verschiedenen Prozessen einer ***ARCHITECTURE*** erfolgt durch die Verwendung von lokalen Signalen.

Damit sind wir in der Lage, einfache Funktionselemente mit zeitlicher Abfolge (wie zum Beispiel einem Zähler) oder aber auch umfangreichere und komplexere Digitalentwürfe umzusetzen.

Deklaration von Prozessen:

```
[<Prozessname>:] process [(<Empfindlichkeitsliste>)]
      <Deklarationsteil>
  begin
      {<sequentielle Anweisungen>}
  end process [<Prozessname>];
```

`Prozessname`: der Name des Prozesses. Dieser kann auch weggelassen werden, was aber nicht empfohlen wird, da dies unter Umständen die Fehlersuche nicht gerade erleichtert.

`Empfindlichkeitsliste`: ist eine, in runde Klammern gesetzte Liste von Signalen; diese werden durch ein Komma getrennt. Eine Veränderung eines dieser Signale startet die Bearbeitung des Prozesses. Die Empfindlichkeitsliste (auch *Liste sensitiver Signale*) ist optional.

`Deklarationsteil`: optionale Variablen und Konstantendeklaration

Es gibt zwei -sich gegenseitig ausschließende- Möglichkeiten, wie man Prozesse starten und stoppen kann:

- mit einer Angabe sensitiver Signale in der Empfindlichkeitsliste
  Prozesse dieser Art werden einmal bei der Modell-Initialisierung komplett durchlaufen und zu späteren Zeitpunkten erst wieder aktiviert, wenn sich eines der Signale in der Empfindlichkeitsliste ändert.
- durch ***WAIT***-Anweisung
  Hier wird der Prozess bei der Modell-Initialisierung bis zum ersten ***WAIT***-Statement durchlaufen und so lange angehalten, bis die ***WAIT***-Bedingung erfüllt wird.

Prozesse ohne ***WAIT***-Anweisung und ohne Empfindlichkeitsliste sind üblicherweise nicht sinnvoll, da solche Prozesse beim Simulationsstart aufgerufen und dann ständig zyklisch durchlaufen werden (→ Endlosschleife).

Mit Prozessen lassen sich aber nicht nur (takt)synchrone Schaltwerke realisieren, sondern auch speicherlose Schaltnetze. Enthält die Empfindlichkeitsliste alle Eingangssignale und werden alle

eingehenden/gelesenen Signale auf andere Signale abgebildet, dann wird kein Zustandsspeicher benötigt und im Zuge der Synthese wird ein Schaltnetz erzeugt.

Zusammengefasst:

VHDL Synthese von taktsynchronen Schaltwerken:

- mit Empfindlichkeitsliste: enthält ausschließlich das Taktsignal. Eine **`IF`**-Abfrage auf eine der Taktflanken umhüllt alle weiteren Anweisungen eines Prozesses, wobei dann keine weiteren Abfragen zum Takt oder zur Taktflanke mehr erfolgen soll (darf)!
- ohne Empfindlichkeitsliste: Als guter Beschreibungs-Stil gilt (und die IEEE möchte dies sogar zur Vorschrift machen): ein Prozess darf mit **`WAIT UNTIL`** die Taktflanke nur ein Mal abfragen. Dies muss die erste Anweisung nach **`BEGINN`** sein; danach sind keine weiteren Abfragen des Taktes oder der Taktflanke mehr zulässig.

VHDL Synthese von Schaltnetzen:

- mit Empfindlichkeitsliste: hier werden alle Eingangssignale in die Empfindlichkeitsliste aufgenommen (d.h. alle Signale rechts der Zuweisungen und Abfragen wie **`IF`** und **`CASE`**). Werden alle eingehenden Signale auf andere Signale abgebildet, so wird kein Zustandsspeicher benötigt und ein kombinatorisches Schaltnetz wird erzeugt (Wiederholung von oben).
- ohne Empfindlichkeitsliste: macht keinen Sinn

Auf eine steigende Taktflanke wird mit der Anweisung

```
CLK='1' and CLK'event
```

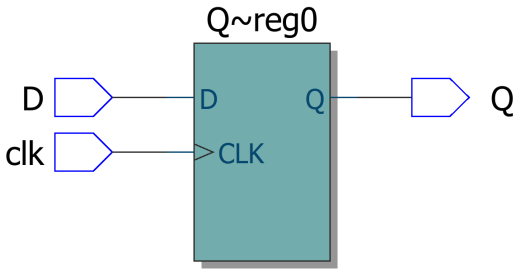
geprüft – auf die fallende Flanke mit

```
CLK='0' and CLK'event
```

Das Signalattribut `'event` ist Bestandteil der Sprache VHDL und bezeichnet einen beliebigen Signalwechsel (ausgesprochen: Tick-Event).

Beispiel für ein positiv taktflankengesteuertes D-FF

```
D_FF: process
  begin
      wait until CLK='1' and CLK'event;
         Q <= D;
  end process D_FF;
```



Innerhalb von Prozessen können Variablen unterschiedlicher Typen verwendet werden. Diese sind lokal und temporär, d.h. sie existieren nur während der „Laufzeit“ des Prozesses. Auf diese Variablen kann erstmals von außerhalb nicht zugegriffen werden. Wenn dies jedoch gewollt ist, muss der Wert der Variable (noch im Prozess) einem Signal zugewiesen werden. Jedoch gibt es hier eine wichtige zeitliche Komponente. Während Variablenwerte sofort bei der Abarbeitung der Anweisung zugewiesen werden, werden die neuen Signalwerte erst vorgemerkt und erst nach Abarbeitung des Prozesses zugewiesen.
Auch wichtig... wird der Prozess neu aktiviert, so besitzen die Variablen wieder die „alten“ Werte

(also die Werte, bevor der Prozess abgearbeitet/beendet wurde).
Im Zuge der Synthese werden nicht immer alle Variablen in real existierende Hardwarekomponenten umgesetzt, sie können auch „weg-optimiert“ werden.

Beispiel für ein zustandsgesteuertes D-FF (Latch)

```
D_FF: process(D, CLK)
  begin
      if CLK='1' then
         Q <= D;
      end if;
  end process D_FF;
```

→ *Latches sollten IMMER vermieden werden !!!*

Eine Liste von Signalen bewirkt, dass so lange gewartet wird, bis sich mindestens eines der Signale ändert. Eine Liste mit Signalen als Argument einer am Ende stehenden **WAIT**-Anweisung entspricht daher einem Prozess mit Empfindlichkeitsliste. Ist ein Signal ein Vektor (Bus), so bewirkt bereits die Änderung eines einzigen Elements den Start eines Prozesses.

Die if-elsif-else-Anweisung

Bedingte Verzweigungen in sequentiellen Anweisungsteilen können mit der **IF-ELSIF-ELSE**-Anweisung folgendermaßen realisiert werden:

```
  if <Bedingung_1_erfuellt> then
      ...
      ...   -- sequentielle Anweisung
      ...
  elsif <Bedingung_2_erfuellt> then
      ...
      ...   -- sequentielle Anweisung
      ...
  elsif <Bedingung_n_erfuellt> then
      ...
      ...   -- sequentielle Anweisung
      ...
  else
      ...
      ...   -- sequentielle Anweisung
      ...
  end if;
```

Zwingend erforderlich ist nur die erste Bedingungsabfrage und die Kennzeichnung des Endes der Struktur mit **END IF;** . Die **ELSIF** und **ELSE**-Teile sind optional, wobei der erstere mehrfach auftreten kann.

Als Bedingungsabfrage sind boolsche Ausdrücke erlaubt, also Signale vom Typ **BOOLEAN** oder logische Vergleichsaussagen. Am einfachsten wird dies durch ein kurzes Beispiel klar...

```
architecture if_elsif_else of Demo is
 signal S1: std_ulogic;
 signal B1: boolean;

begin
PROC: process(S1, B1)
```

```
  begin
    if S1 then
    ...                    -- FEHLER: kein boolean oder keine log. Aussage;
    elsif S1='0' then
    ...                    -- korrekt
    elsif B1 then
    ...                    -- korrekt
    elsif B1 and (S1='1') then
    ...                    -- korrekt
    end if;                -- else fehlt -> kein guter Stil
  end process PROC;
end if_elseif_else;
```

→ ***Übung:*** Entwerfe einen 4-zu-1 Multiplexer unter Verwendung eines Prozesses und if – Abfragen.

Achtung: der folgende VHDL-Code zeigt eine sog. „incomplete branch“ und sollte auf jeden Fall vermieden werden:

```
process(A,B)
begin
    if (A=B) then
      X <= '1';            -- beide Eingangssignale sind gleich → Ausgang log. 1
    end if;
end process;
```

Hinweis-Meldungen während der Kompilierung:

```
Warning (10631): VHDL Process Statement warning at Incomplete_Branch.vhd(13): inferring latch(es)
for signal or variable "X", which holds its previous value in one or more paths through the process

Info (10041): Inferred latch for "X" at Incomplete_Branch.vhd(13)
```

… da die oberen Codezeilen gleichbedeutend sind mit

```
process(A,B)
begin
    if (A=B) then
      X <= '1';            -- beide Eingangssignale sind gleich → Ausgang log. 1
    else
      X <= X;
    end if;
end process;
```

… und dafür ein Speicherelement benötigt wird (→ closed feedback loop). Ein möglicher, korrekte VHDL-Code ist:

```
process(A,B)
begin
    if (A=B) then
      X <= '1';            -- beide Eingangssignale sind gleich → Ausgang log. 1
    else
      X <= '0';            -- ansonsten log. 0
    end if;
end process;
```

Die case-Anweisung

Mit der ***CASE*** – Anweisung lassen sich sehr komfortabel Wertetabellen in VHDL beschreiben. Dies

stellt insbesondere für den FPGA Entwurf eine sehr elegante und effiziente Beschreibungsform dar, da kleinere Wertetabellen praktisch unverändert in die SRAM LUT geschrieben werden.

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

entity Segmentdecoder is
  port ( bcd:     in std_ulogic_vector(3 downto 0);
         sseg:    out std_ulogic_vector(6 downto 0)
       );
end Segmentdecoder;

architecture case_implementierung of Segmentdecoder is
begin
DEC:process(bcd)
   variable Temp_Sieben_Seg : std_ulogic_vector(6 downto 0);
   begin
      case bcd is
         -- 7-Segment-Ansteuerung: gfedcba
         when "0000" => Temp_Sieben_Seg := "0111111";-- Ziffer 0
         when "0001" => Temp_Sieben_Seg := "0000110";-- Ziffer 1
         when "0010" => Temp_Sieben_Seg := "1011011";-- Ziffer 2
         when "0011" => Temp_Sieben_Seg := "1001111";-- Ziffer 3
         when "0100" => Temp_Sieben_Seg := "1100110";-- Ziffer 4
         when "0101" => Temp_Sieben_Seg := "1101101";-- Ziffer 5
         when "0110" => Temp_Sieben_Seg := "1111101";-- Ziffer 6
         when "0111" => Temp_Sieben_Seg := "0100111";-- Ziffer 7
         when "1000" => Temp_Sieben_Seg := "1111111";-- Ziffer 8
         when "1001" => Temp_Sieben_Seg := "1101111";-- Ziffer 9
         when others => Temp_Sieben_Seg := "0000000";-- Anzeige AUS
      end case;

      sseg <= not Temp_Sieben_Seg;  -- DE0: 7Seg. LOW aktiv

   end process DEC;
end case_implementierung;
```

→ ***Übung:*** In diesem Beispiel wurde eine Variable `Temp_Sieben_Seg` verwendet. Modifiziere den Code so, dass anstelle der Variablen mit einem Signal gearbeitet wird. Was ändert sich syntaktisch?

→ ***Übung:*** Entwerfe einen Codewandler, der einen 4bit Eingangs-Gray-Code in einen 4-bit Ausgangs-BCD-Code umsetzt.

2 Schaltwerke, Register & Taktteiler

Schaltwerke -d.h. Logikschaltungen mit Flip-Flop-Elementen- werden oft benötigt. Darunter fallen auch Zähler, die auch als Taktteiler verwendet werden können.
Als Einstiegsbeispiel soll ein positiv taktflankengesteuertes D-FF mit asynchronem Reset-Eingang sowie einem Enable-Eingang (Taktfreigabe) entworfen werden.
Dazu verwenden wir einen VHDL-Prozess und if-Abfragen, der Code kann wie folgt aussehen:

```
entity D_FF_mit_Reset_und_Enable is
  port ( CLK, E, RST, D:   in std_ulogic;
         Q:                out std_ulogic
```

```
      );
end D_FF_mit_Reset_und_Enable;

architecture Verhalten of D_FF_mit_Reset_und_Enable is
begin
FF: process(CLK, RST)
    begin
      if RST = '1' then
         -- asynchroner Reset
         Q <= '0';

      elsif CLK = '1' and CLK'event then
         if E = '1' then  -- haben wir die Freigabe erhalten?
           Q <= D;        -- JA: dann speichere Zustand der Datenleitung D
         end if;
      end if;
    end process FF;
end Verhalten;
```

Die Abfrage auf die steigende Taktflanke wird mit

```
      if CLK = '1' and CLK'event then
```

durchgeführt. Eine log. Verknüpfung mit anderen Boolschen Ausdrücken ist dabei NICHT möglich; d.h. die folgende Anweisung ist nicht synthetisierbar:

```
      if CLK = '1' and CLK'event and E='1' then
```

Ein D-FF mit Enable soll den Eingangswert D nur bei aktivem Enable-Signal übernehmen – dies realisierten wir mit der zweiten if-Abfrage. Ein asynchroner Reset ist immer unabhängig von einem Takt; deshalb wurde in der Empfindlichkeitsliste das Reset-Signal aufgenommen und vor der Taktflankenabfrage auf die Reset-Bedingung abgeprüft.

Werden mehrere D-FFs parallel geschaltet, so entsteht ein Register. Jedes dieser D-FFs besitzt einen eigenen Dateneingang und einen eigenen Datenausgang – lediglich die CLK-, Reset- sowie ggf. Enable-Leitungen teilen sich alle D-FFs innerhalb eines Registers. Zweckmäßigerweise arbeitet man in einem solchen Fall mit Vektoren.

→ ***Übung:*** Baue das vorherige, taktflankengesteuerte D-FF zu einem 8-bit Register aus. Wie groß ist der Änderungsaufwand?

In VHDL gibt es das Schlüsselwort **GENERIC**. Parameterwerte innerhalb einer ***ENTITY*** werden dabei wie Konstanten behandelt. Damit können Hardwareelemente modelliert werden, deren Dimensionierung variabel gestaltet werden kann, da ein **GENERIC**-Wert auch von einer höheren Hierarchieebene überschrieben werden kann.

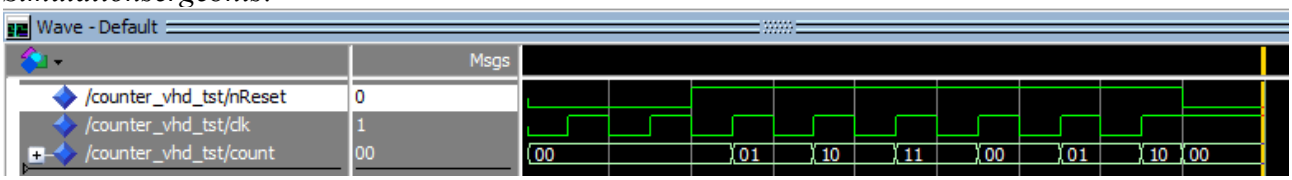
Beispiel:

```
entity Register is
  generic(N: integer:=16);
  port ( CLK, E, RST:   in std_ulogic;
         D:             in std_ulogic_vector(N-1 downto 0);
         Q:             out std_ulogic_vector(N-1 downto 0)
      );
end Register;
```

→ ***Übung:*** Zugegen, die folgende Aufgabe ist nicht ganz so einfach...
Entwerfe einen 2bit Binärzähler, der bei jeder positiven Taktflanke um eine Stelle weiter zählt. Über

einen Low-aktiven Reset-Eingang ist der Zähler zurückzusetzen. Für die Umsetzung sind nur Sprachkonstrukte zu verwenden, die bis zu diesem Punkt besprochen wurden... d.h. es darf keine arithmetische Bibliothek verwendet werden!

Simulationsergebnis:

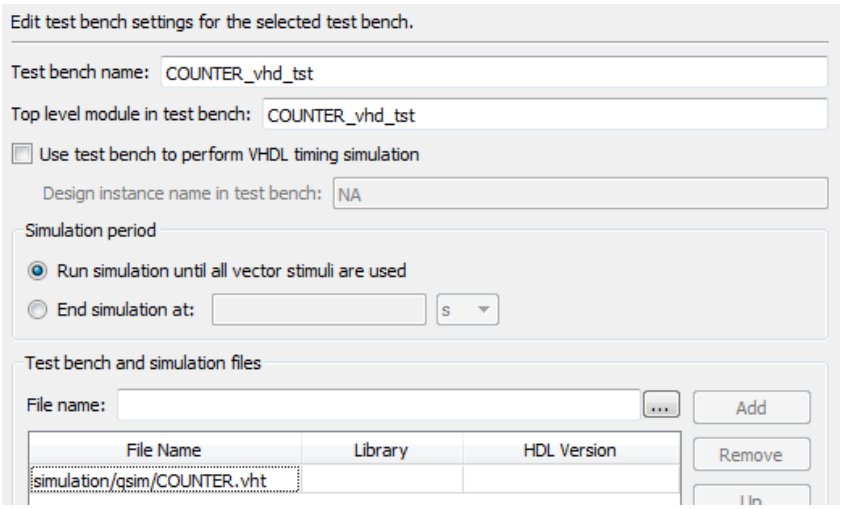


Erstellen der Testbench:

1) Assignements → Settings → EDA Tool Settings: ModelSim-Altera , VHDL
2) Processing → Start → Start Testbench Template Writer
   → auf Basis der TopLevel Entity wird ein Testbench-Template erstellt, zu finden unter .\simulation\ qsim mit der Dateiendung .vht
3) das Testbench-Template ist unter Files hinzuzufügen
4) Doppelklick auf vht öffnet das VHDL-Testbenchfile im Editor
5) Ändern von `std_logic` auf `std_ulogic` (→ Unterschied: siehe Kapitel 2.6)
6) Hinzufügen der folgenden Codezeilen, wobei dieser Code NICHT optimal ist!!!!

```
-- code executes for every event on sensitivity list
-- test 1: nReset = 0, clk
nReset <= '0';
clk <= '0';
wait for 50ns;
clk <= '1';
wait for 50ns;
clk <= '0';
wait for 50ns;
clk <= '1';
wait for 50ns;

-- test 2: nReset = 1, clk
nReset <= '1';
clk <= '0';
wait for 50ns;
clk <= '1';
wait for 50ns;
clk <= '0';
wait for 50ns;

… weitere / mehrmalige Wdhlg 0/1 Taktwechsel …

clk <= '0';
wait for 50ns;
clk <= '1';
wait for 50ns;

-- test 3: Umschalten auf nReset = 0, OHNE CLK
nReset <= '0';
wait for 100ns;
```

7) Anpassen der Simulationseinstellung:
Assignement → Settings → EDA Tool Settings → Simulation → Compile test bench …



8) Start der RTL Simulation...

Das „manuelle“ Erzeugen des Simulationstaktes ist natürlich nicht optimal… viele eleganter ist die Verwendung eines eigenen Prozesses, das den Takt generiert.

```
END PROCESS init;

PROC_CLK: process              -- Taktgenerator, eigener Prozess!!!
begin
      clk <= '0';
      wait for 50ns;
      clk <= '1';
      wait for 50ns;
end process PROC_CLK;

always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
      -- code executes for every event on sensitivity list
      -- test 1: nReset = 0
      nReset <= '0';
      wait for 200ns;

      -- test 2: nReset = 1
      nReset <= '1';
      wait for 12*50ns;

      -- test 3: Umschalten auf nReset = 0
      wait for 20ns;   -- Umschalten asynchron zur Flanke
      nReset <= '0';
      wait for 100ns;

      -- terminate simulation
      --WAIT;
      assert false
            report "Simulation Completed"
      severity failure;

END PROCESS always;
END COUNTER_arch;
```

Taktflankensteuerung

Zur Modellierung einer Taktflanke wurde das Attribut `event` eingeführt. Auf eine steigende Taktflanke kann mit

```
CLK='1' and CLK'event
```

abgeprüft werden. Streng genommen ist diese Formulierung jedoch nicht ganz korrekt/vollständig. Warum?

In der Regel meint man bei einer steigenden Flanken (bei fallenden Flanken gilt nachfolgendes entsprechend) Übergängen von `0` auf `1`. Verwendet man jedoch `std_logic`, so gibt es weitere Signalzustände – und ein Wechsel von `X` nach `1` würden wir mit dieser Beschreibung ebenfalls erfassen; was man allerdings in vielen Fällen gar nicht möchte.

Man sollte daher vorher überprüfen, ob das vorherige Taktsignal `0` war. Dazu gibt es das Attribut `last_value`.

Beispiel für ein positiv taktflankengesteuertes D-FF

```
D_FF: process
  begin
      wait until CLK='1' and CLK'event and CLK'last_value='0';
         Q <= D;
  end process D_FF;
```

Da das Erkennen von steigenden oder fallenden Flanken eines Signals häufig benötigt wird, wurden die zwei Funktionen `rising_edge` und `falling_edge` in das IEEE 1164-Package aufgenommen. Hier wird ein Signal vom Typ ***`BOOLEAN`*** zurückgegeben, das ***`TRUE`*** ist, wenn eine steigende bzw. fallende Flanke erkannt wurde.

```
FUNCTION rising_edge  (SIGNAL s : std_ulogic)
         RETURN boolean;
FUNCTION falling_edge (SIGNAL s : std_ulogic)
         RETURN boolean;
```

Schleifen

Zum Sprachumfang der meisten prozeduralen Sprachen gehören auch Schleifen, sog. loops. Da diese Schleifen sequentiell abgearbeitet werden, können diese bei VHDL nur innerhalb eines Prozesses verwendet werden. VHDL kennt dabei die folgenden 3 Schleifen:

```
[Loop_label :] for Bereichsdefinition loop
      ...
      ...    -- sequentielle Anweisungen
      ...
end loop [Loop_label] ;

[Loop_label :] while Bedingung loop
      ...
      ...    -- sequentielle Anweisungen
      ...
end loop [Loop_label] ;

[Loop_label :] loop
      ...
      ...    -- sequentielle Anweisungen
      ...
end loop [Loop_label] ;
```

Als Bereichsdefinition sind die folgenden zwei Varianten zulässig:

```
      <Schleifenindex> in <Untere_Grenze> to <Obere_Grenze>
```

```
<Schleifenindex> in <Obere_Grenze> downto <Untere_Grenze>
```

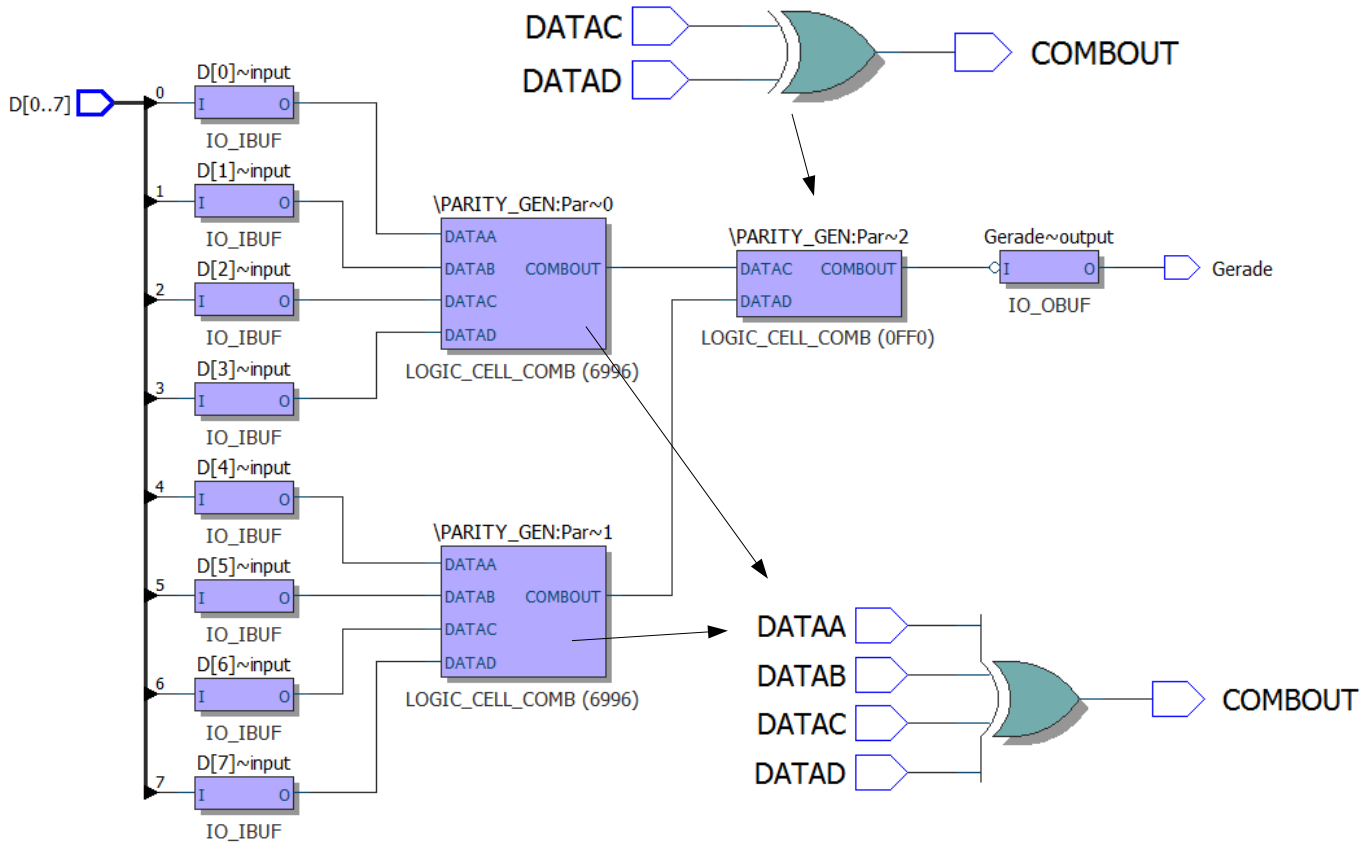
Der Schleifenindex muss nicht deklariert werden, dies wird bereits implizit durch die Verwendung des for-Schleifenkonstrukts durchgeführt.

Beispiel für eine for-Schleife:

```
entity Parity_Checker is
  generic(BITS: integer:=8);
  port ( D:         in std_ulogic_vector(BITS-1 downto 0);
         Gerade:    out std_ulogic
  );
end Parity_Checker;

architecture Verhalten of Parity_Checker is
begin
PARITY_GEN: process(D)
    variable Par : boolean;
    begin
    Par := false;
    for I in BITS-1 downto 0 loop
      if D(I) = '1' then
        Par := not Par;
      end if ;
    end loop ;
    if Par then
       Gerade <= '0';   -- ungerade Anzahl von 1en
    else
       Gerade <= '1';
    end if;
end process PARITY_GEN;
end Verhalten;
```

`BITS` gibt die Bitlänge des Datenvektors an, für den die Paritätsberechnung (auf gerade Parität) erfolgt. Im Prozess wird die Variabel `Par` verwendet, die in der **FOR**-Schleife bei jeder auftretenden `'1'` negiert wird.

Synthese-Ergebnis (Tools → Netlist Viewers → Technology Map Viewer (Post Mapping)):



3 Rechnen in VHDL

Es gibt verschiedene Möglichkeiten, in VHDL synthetisierbare Berechnungen zu beschreiben.

Bei der Konzeption der Beschreibungssprache VHDL wurde auf eine strenge Typisierung der Datentypen und Typdeklarationen Wert gelegt, da sich dadurch „Programmierfehler“ meist im Vorfeld vermeiden lassen. In VHDL sollte daher unbedingt festgelegt werden, welche Werte ein Objekt annehmen kann (zum Beispiel „Signal w kann einen ganzzahligen Wert zwischen 1 und 6 annehmen“). Erfolgt dies nicht, so liefern viele Compiler / Synthese-Werkzeuge Fehler; andere wiederum führen alle Berechnungen mit einer Wortbreite von 32bit durch. VHDL besitzt nativ nur sehr wenige Datentypen wie ***INTEGER***, ***NATURAL*** (kann nur 0 oder ganzzahlige positive Werte annehmen; Subtyp von integer) oder ***REAL***, bietet aber dafür umfangreiche Möglichkeiten, benutzerdefinierte Datentypen zu definieren.

Operatoren und Datentypen nach VHDL 93 und mit IEEE std_logic_1164:

Operation	Beschreibung	Datentyp der Operanden	Datentyp des Ergebnisses	numeric_std
a ** b	a^b	Integer	Integer	
a * b	Multiplikation	(unsigned/signed) Integer	Integer	√
a / b	Division	Integer	Integer	
a + b	Addition	(unsigned/signed) Integer	Integer	√
a - b	Subtraktion	(unsigned/signed) Integer	Integer	√

a = b	Gleichheit	beliebig	Boolean	√
a /= b	Ungleichheit	beliebig	Boolean	√
a < b	kleiner	beliebig	Boolean	√
a <= b	kleiner o. gleich	beliebig	Boolean	√
a > b	größer	beliebig	Boolean	√
a >= b	größer o. gleich	beliebig	Boolean	√

Schiebeoperationen, Addition und Subtraktion sind meist nie ein Problem. Multiplikationen werden -so weit wie möglich- erstmals unter Verwendung eventuell vorhandener HW-Multiplizierer gelöst. Divisionen und Modulo-Berechnungen werden in der Regel jedoch nicht oder nur für Zweierpotenzen unterstützt – d.h. hier muss in den Applikation-Notes des jeweiligen FPGA Herstellers nachgeschlagen werden. Erfolgt keine direkte Unterstützung, so muss der Algorithmus „von Hand“ implementiert werden, oder man greift ggf. auf einen vorgefertigten Block (IP-Core) zurück.

Für die Rechnung mit Ganzzahlen gibt es prinzipiell zwei Möglichkeiten... entweder verwendet man direkt Integer-Typen oder man konvertiert einen `std_ulogic_vector` in einen ***INTEGER***. Wichtig ist dabei, den Wertebereich des Integertypen anzugeben, damit auch wirklich nur mit der benötigten Bitbreite gearbeitet wird.

Beispiel:

```
variable X: integer range 0 to 10;
...
X := X + 1;
...
```

Integer werden gerne in internen Berechnungen verwendet – zum Beispiel bei einem Zähler, der hoch zählt und beim Erreichen eines bestimmten Zählerstandes einen Ausgang setzt. Wenn man allerdings auf Basis von externen Vektoren Berechnungen durchführen muss, so ist es empfehlenswert, dies direkt auf Vektor-Basis vorzunehmen.

Die nicht IEEE-standardisierten Bibliothekspakete `std_logic_unsigned` und `std_logic_signed` erlauben es, direkt mit `std_logic_vector` zu rechnen. Je nach dem, welches der beiden Pakete eingebunden wurde, werden alle (!) std_logic-Vektoren entweder als *signed* oder *unsigned* interpretiert und auf dieser Basis alle Berechnungen durchgeführt... was natürlich in vielen Fällen schlecht und fehleranfällig ist.
Die Berechnungen selber erfolgen mit Hilfe des Paketes `std_logic_arith`. Da dieses Paket jedoch unter dem Copyright von Synopsys steht, kann dies unter Umständen problematisch sein.

Die IEEE empfiehlt die IEEE-eigene Bibliothek `numeric_std` einzubinden. Dabei werden auch die Typen `signed` und `unsigned` als Array von `std_logic` definiert, so dass auch keine speziellen *signed*/*unsigned* Pakete mehr eingebunden werden müssen.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
...
signal S: unsigned(7 downto 0);     -- Zahlenbereich: 0 bis 2**8-1
signal C: signed(7 downto 0);       -- Zahlenbereich: -2^7 bis 2^7-1
...
C <= C + 1;
```

```
S <= C + S;
```

Da VHDL eine stark typisierende Sprache ist, werden `std_logic_vector`, `unsigned` und `signed` als unterschiedliche Datentypen behandelt. Das mag überraschen, bedenkt man, dass alle diese Datentypen auf Arrays von Elementen des Typs `std_logic` basieren. Man benötigt daher auch Typ-Umwandlungsfunktionen, um Signale verschiedener Datentypen zu konvertieren. Die folgende Tabelle gibt eine Zusammenfassung dieser Funktionen:

aktueller Datentyp von a	Ziel-Datentyp	Konvertierungsfunktion
unsigned, signed	std_ulogic_vector	std_ulogic_vector(a)
signed, std_ulogic_vector	unsigned	unsigned(a)
unsigned, std_ulogic_vector	signed	signed(a)
unsigned, signed	Integer	to_integer(a)
natural	unsigned	to_unsigned(a, Länge)
Integer	signed	to_signed(a, Länge)

D.h. möchte man ***INTEGER***-Variablen einem *signed* / *unsigned* Signal zuweisen, so ist eine Konvertierung durchzuführen:

```
X <= to_unsigned(123, X'length);
Y <= to_signed(123, Y'length);
```

Mit dem zweiten Parameter wird die Vektor-Länge angegeben, mit der die Integerzahl konvertiert werden soll. Es kann ja sein, dass der Vektor länger sein muss als die zu konvertierende Zahl Bitstellen benötigt (denke hier an eine 0 oder 1 – zum Beispiel!).

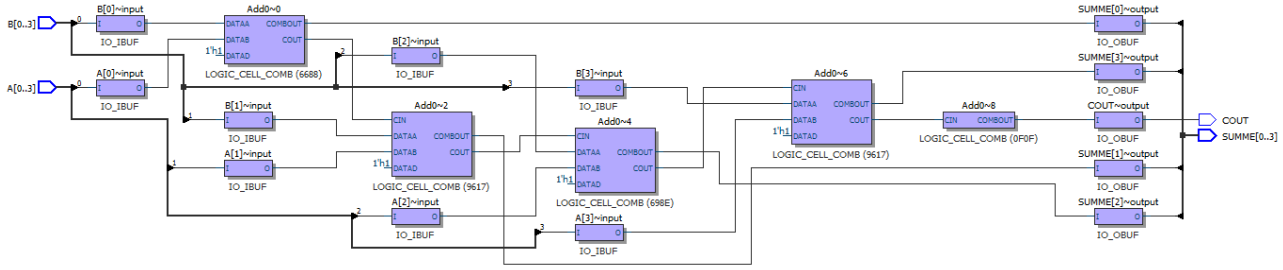
In VHDL kann man auch mit Festkommazahlen (Fixed Point) sowie Gleitkommazahlen (Floating Point; Type real) rechnen. Wenn man allerdings die Beschreibung in eine HW umsetzen möchte (Synthese), so können Gleitkommaberechnungen sehr schnell sehr umfangreich und komplex werden, da die Zahlen in jedem Fall zuerst in eine `std_ulogic_vector` – Darstellung gebracht werden müssen.

Beispiel eines N-Bit-Addierers mit Carry_out:

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

entity Add_mit_Carry is
  generic(N: integer:=4);                    -- Breite der Eingangssignale
  port ( A, B:        in std_ulogic_vector(N-1 downto 0); --Eingangssignale
         COUT:        out std_ulogic;                     --Carry OUT
         SUMME:       out std_ulogic_vector(N-1 downto 0) --Additionssumme
  );
end Add_mit_Carry;

architecture Verhalten of Add_mit_Carry is
  signal A_INT, B_INT, SUM_INT: unsigned(N downto 0);
begin
  A_INT <= unsigned ('0' & A);
  B_INT <= unsigned ('0' & B);
  SUM_INT <= A_INT + B_INT;                                  -- Addition
```

```
    SUMME <= std_ulogic_vector(SUM_INT(N-1 downto 0));      -- Typenumwandlung
    COUT <= SUM_INT(N);                                     -- MSB ist Uebertrag
end Verhalten;
```

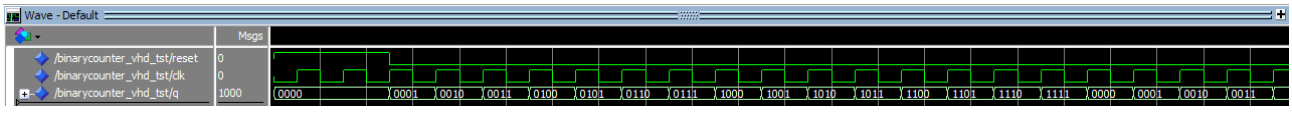
Interessant ist hier, was der Compiler aus diesem VHDL Code macht. Die RTL-Darstellung ist dabei nicht besonders hilfreich, da ein Addierer ein Grundbaustein der Register-Transfer-Ebene darstellt. Man sollte hier aber unbedingt einen Blick in die „Technology Map“ - Darstellung werfen (Tools → Netlist Viewers → Technology Map Viewer)... und wie man sieht, setzt sich der Addierer aus vielen Halbaddierern (Add0~0) / Volladdierern (z.B. Add0~2) zusammen (rechter Mausklick auf Block → Properties)



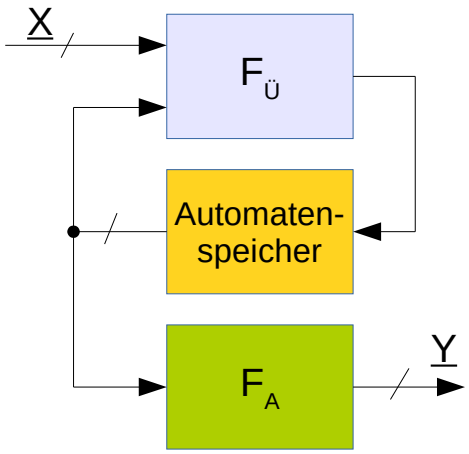
Beispiel eines 4-bit Binärzählers:

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

entity BinaryCounter is
  port ( CLK, RESET:  in std_ulogic;
         Q:           out std_ulogic_vector(3 downto 0)
  );
end BinaryCounter;

architecture Verhalten of BinaryCounter is
  signal cnt: unsigned(3 downto 0);      -- Q als out nicht lesbar (< VHDL2008)

begin
CNT_BIN: process(CLK, RESET)
  begin
  if (RESET = '1') then
    cnt <= (others => '0');              -- Reset: Zähler auf 0

  elsif rising_edge(CLK) then            -- pos. Taktflanke
     cnt <= cnt + 1;
  end if;
end process CNT_BIN;

Q <= std_ulogic_vector(cnt);             -- Zuweisung Ausgangssignal

end Verhalten;
```

Simulationsergebnis:



Ein Zähler kann auch als autonomer Automat realisiert werden – zur Erinnerung Moore-Struktur:



… wobei der Automatenspeicher ein getaktetes Register ist und in $F_Ü$ die Addition um +1 erfolgt:

```
architecture Automat of BinaryCounter is
signal reg: unsigned(3 downto 0);             -- Ausgang Automatenspeicher
signal reg_next: unsigned(3 downto 0);        -- Ausgang F_UE

begin

-- Automatenspeicher
AS: process(CLK, RESET)
  begin
  if (RESET = '1') then
    reg <= (others => '0');

  elsif rising_edge(CLK) then
     reg <= reg_next;
  end if;
end process AS;

-- F_UE
reg_next <= reg + 1;

-- F_A
Q <= std_ulogic_vector(reg);
end Automat;
```

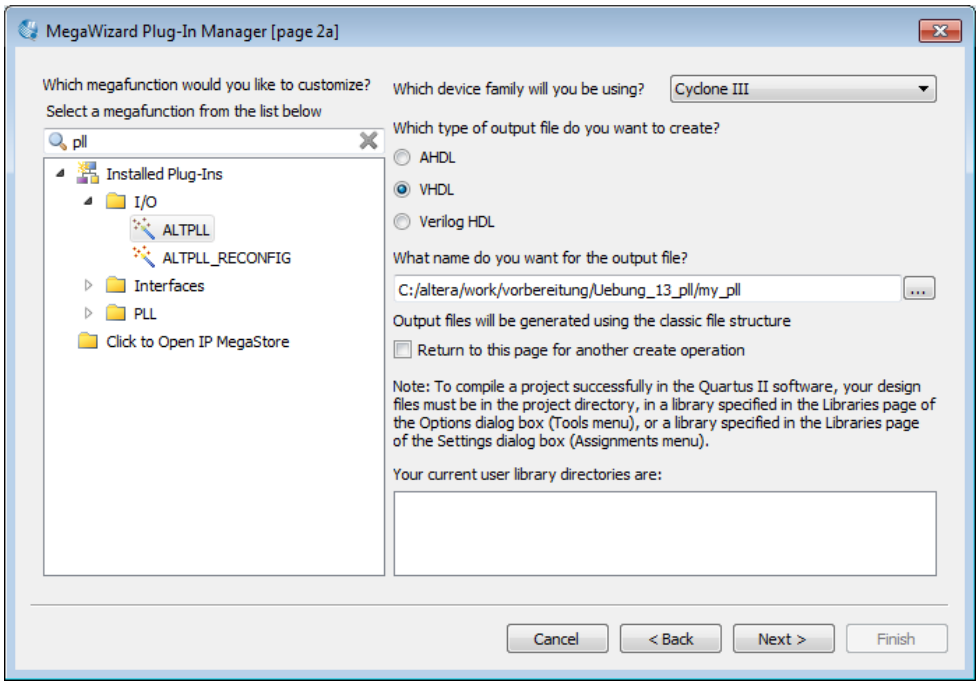
→ ***Übung:*** Baue den binären Zähler zu einem dekadischen Zähler (Zählumfang 0...9) um.

Mittels Zähler lassen sich auch Taktteiler realisieren. So befindet sich zum Beispiel auf dem DE0-Board ein 25MHz (Quarz-) Oszillator und daraus könnte man z.B. ein 100Hz Signal ableiten. Das Herunterteilen eines Taktes benötigt jedoch Ressourcen (FFs) - der Cyclone III FPGA besitzt vier interne PLLs, die sich dafür besser eignen. Die PLL unterstützt nicht beliebige Teilungsfaktoren – so dass eine Kombination aus PLL und Taktteiler (FFs) sinnvoller ist als eine reine FF-Teilerkette.

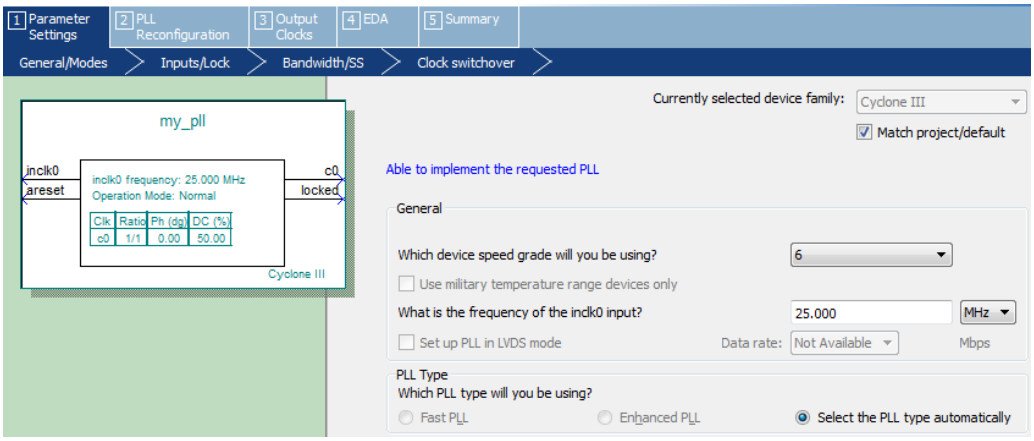
Die PLL ist Teil des IP-Umfanges, das sind mitgelieferter Bibliotheken von Altera:

Tools → MegaWizard Plug-In Manager → Create a new custom megafunction variation →

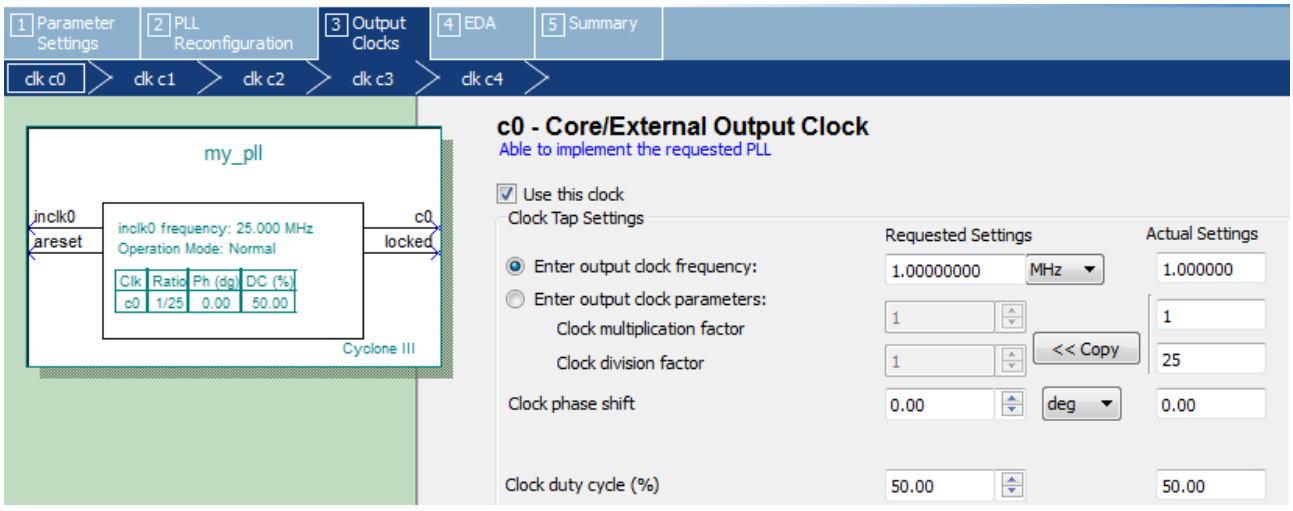
ausgewählt wird `ALTPLL`, `VHDL` und als Name der neuen Komponente wird `my_pll` eingetragen:



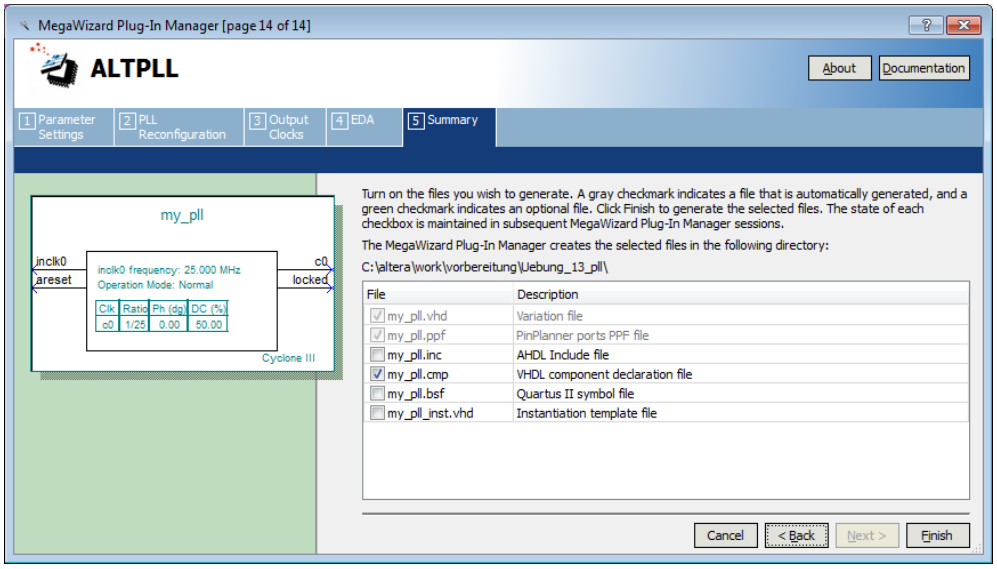
Im nächsten Konfigurationsfenster erfolgt die Angabe der Eingangsfrequenz (DE0: 25MHz)



… und alle weiteren Konfigurationsseiten bleiben unverändert bis zur ersten Seite von Abschnitt 3 – Output Clocks. Hier gibt man die benötige Ausgangsfrequenz an (die auch größer als die Eingangsfrequenz sein kann) – in unserem Fall 1MHz:



Auf der letzten Seite erhält man noch eine Auflistung mit Vorselektion zu generierender Dateien – hier belassen wir die Standardeinstellung.

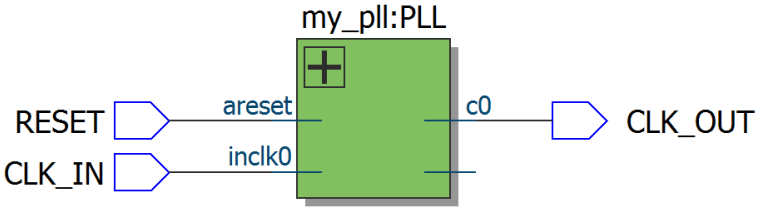


Nach Drücken von Finish erhält man auch einen Hinweis, ob man die neue Komponente dem aktuellen Projekt hinzufügen will oder nicht… was wir natürlich wollen.

VHDL-Beispielcode zur Verwendung unserer PLL-Komponente:

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

entity PLL_DEMO is
  port ( RESET, CLK_IN:     in std_ulogic;
         CLK_OUT:           out std_ulogic
  );
end PLL_DEMO;

architecture Einsatz_PLL of PLL_DEMO is
begin
-- Instanzierung unserer PLL
PLL: entity work.my_pll port map (
    areset => RESET,
    inclk0 => CLK_IN,
    c0 => CLK_OUT,
    locked => open          -- Locked-Ausgang der PLL soll offen bleiben
);

end Einsatz_PLL;
```

Umsetzung:



4 Unterschied std_logic und std_ulogic

Signale vom Typ `std_ulogic` dürfen nur einen Signaltreiber besitzen; Signale vom Typ `std_logic` hingegen mehrere. Dies ist vergleichbar mit dem Verbinden mehrerer Gatterausgänge, wobei es hier zu Konflikten kommen kann, die durch Auflösungsfunktionen gelöst werden.

Auflösungstabelle

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	-

Beispiele:

- liefert ein Signaltreiber ein log. 0 und ein zweiter ein log.1, so ergibt dies das undefinierte Signal X
- wenn ein Signaltreiber hochohmig (Z) ist, so wird das Ergebnis vom anderen Signaltreiber bestimmt
- wenn der Ausgang eines Signaltreibers undefiniert (U) ist, so muss das Ergebnis auch undefiniert sein

5 Projekt: Stoppuhr

Alles bisher vermittelte sollte zu einem Abschlussprojekt -einer Stoppuhr- zusammengefasst werden. Bei Drücken auf „Button1“ ist die Uhr zu starten – bei Drücken auf „Button2“ zu stoppen. Die Anzeige erfolgt mittels den vier 7-Segment-Anzeigen in Hundertstel, Zehntel und einer zweistelligen Sekundenanzeige. Mit Button0 ist die Uhr wieder auf 00.00 zurückzustellen.

Vorgabe:

```
entity StopWatch is
  port(
   -- 50MHz Takt
   CLOCK_50 :     in std_ulogic;

   -- Buttons zum Start-/Stop-/Reset
   BUTTON   :     in std_ulogic_vector(3 downto 0);

   -- 7-Segment-Anzeigen...
   HEX0_D   :     out std_ulogic_vector(6 downto 0);  -- gfedcba
```

```
    HEX0_DP  :      out std_ulogic;
    HEX1_D   :      out std_ulogic_vector(6 downto 0);
    HEX1_DP  :      out std_ulogic;
    HEX2_D   :      out std_ulogic_vector(6 downto 0);
    HEX2_DP  :      out std_ulogic;
    HEX3_D   :      out std_ulogic_vector(6 downto 0);
    HEX3_DP  :      out std_ulogic
  );
end StopWatch;
```

… damit kann die Pinzuordnung mittels vorbereiteter „Pin_DE0_TOP.csv“ Datei erfolgen:

Assignments → Import Assignments → Auswahl Pin_DE0_TOP.csv → OK

Mögliche TopLevel-Struktur: