Rechnerstrukturen im SS2008

Hardwareentwurf

Dr. Rainer Buchty
buchty@ira.uka.de

Universität Karlsruhe (TH) – Forschungsuniversität
Institut für Technische Informatik (ITEC)
Lehrstuhl für Rechnerarchitektur

24.04.2008

Organisatorisches

Vorlesungsfolien, Unterlagen zur Übung, Ergänzungen, Errata, Termine, Ankündigungen, etc. unter

http://itec.uka.de/capp/teaching/rs

Zusammenhänge

- **Leistungsverbrauch**
  - gesamt: $P_{switching} + P_{shortcircuit} + P_{static} + P_{leakage}$
  - Schaltleistung: $P_{switching} \approx C_{eff} * U^2 * f$
- **Zusammenhänge**
  - Miniaturisierung steigert Einfluß des Leckstroms
  - Temperaturerhöhung steigert Einfluß der Leckstroms
  - $P \approx U^2 * f$: Abhängigkeiten, Steigerungsraten

Aufgabe

- Die Kernspannung von Prozessoren ist seit den 1980er-Jahren von 5V auf 0.8V gesenkt worden. Im gleichen Zeitraum stieg die Frequenz von 1MHz auf 1GHz. Was bedeutet dies für die aufgenommene elektrische Leistung?

Aufgabe

- Die Kernspannung von Prozessoren ist seit den 1980er-Jahren von 5V auf 0.8V gesenkt worden. Im gleichen Zeitraum stieg die Frequenz von 1MHz auf 1GHz. Was bedeutet dies für die aufgenommene elektrische Leistung?

- $5V \rightarrow 0.8V \leftrightarrow U^2 : 25 \rightarrow 0.64$ (Faktor 39.1)
- $1MHz \rightarrow 1GHz \leftrightarrow$Faktor 1000
- Aus $P \approx U^2 * f$ resultiert eine Zunahme der elektrischen Leistung um den Faktor 25.6

Aufgabe

- Zum Übertakten von Prozessoren wird die Kernspannung erhöht. Warum ist dies so? Wie fließt die Kernspannungserhöhung in die Leistungsaufnahme ein und was bedeutet dies?

Aufgabe

- Zum Übertakten von Prozessoren wird die Kernspannung erhöht. Warum ist dies so? Wie fließt die Kernspannungserhöhung in die Leistungsaufnahme ein und was bedeutet dies?

- $P_{switching} = C_{eff} * U^2 * f$
- Schnelleres Laden von C_{eff} und damit steilere Flanken
- Nachteil: Spannungsbeitrag wird quadratisch errechnet

- **Statistische Verfahren** (Schaltwahrscheinlichkeiten) im Low-Power-Bereich von Bedeutung

Schaltungsentwurf

Gegeben Sei ein UND-Gatter mit zwei Eingängen. Die Eingabewerte (0,1) seien gleichverteilt.

- **Statistische Verfahren** (Schaltwahrscheinlichkeiten) im Low-Power-Bereich von Bedeutung

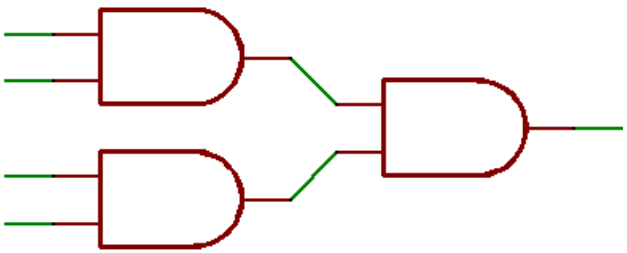
Schaltungsentwurf

Gegeben Sei ein UND-Gatter mit zwei Eingängen. Die Eingabewerte (0,1) seien gleichverteilt.

- UND: 1 wenn beide Eingänge 1, sonst 0
- 4 Zustände (00,01,10,11): somit $p_1 = \frac{1}{4}, \quad p_0 = \frac{3}{4}$

Low-Power-Entwurf (forts.)

- UND-basierte Funktion mit 4 Eingängen:
  Auswirkung von Schaltwahrscheinlichkeiten



Schaltfunktion:

- $f_{UND}(x,y) = \begin{cases} 1 & \forall x = y = 1 \\ 0 & sonst \end{cases}$

Ausgangssituation rechtes Gatter:

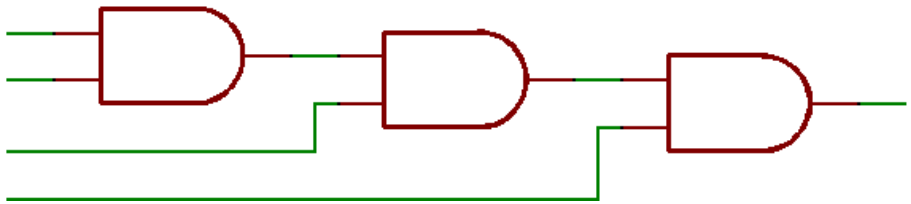
- $p_1 = \frac{1}{2} * \frac{1}{2} = \frac{1}{4}$
- $p_0 = 1 - p_1 = \frac{3}{4}$

Für das rechte Gatter gilt daher:

- $p_{1'} = p_1 * p_1 = \frac{1}{16}$
- $p_{0'} = 1 - p_{1'} = \frac{15}{16}$
- Schaltwahrscheinlichkeit somit $\sum_{p_1} = 2 * \frac{1}{4} + \frac{1}{16} = 0,5625$

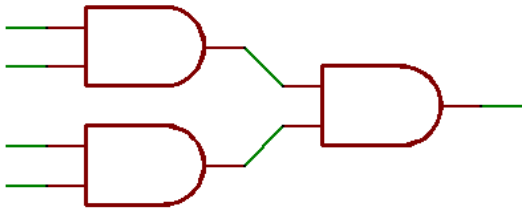
- UND-basierte Funktion mit 4 Eingängen:
  Auswirkung von Schaltwahrscheinlichkeiten

- Hinter erstem Gatter:
  $p_1 = \frac{1}{4}, \quad p_0 = \frac{3}{4}$
- Hinter zweitem Gatter:
  $p_{1'} = p_1 * \frac{1}{2} = \frac{1}{4} * \frac{1}{2} = \frac{1}{8}$
  $p_{0'} = 1 - p_{1'} = \frac{7}{8}$
- Hinter drittem Gatter:
  $p_{1''} = p_{1'} * \frac{1}{2} = \frac{1}{8} * \frac{1}{2} = \frac{1}{16}$
  $p_{0''} = 1 - p_{1''} = \frac{15}{16}$
- Schaltwahrscheinlichkeit:
  $\sum_{p_1} = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0,4375$

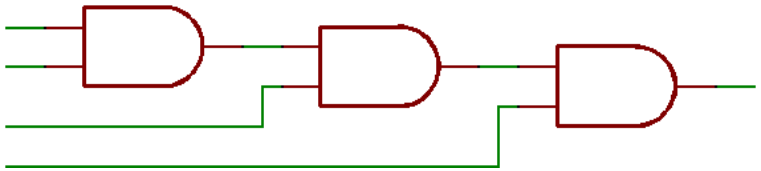


Low-Power-Entwurf (forts.)

- UND-basierte Funktion mit 4 Eingängen:
  Auswirkung von Schaltwahrscheinlichkeiten



- $p_{gesamt} = 0,5625$
- Höherer Leistungsverbrauch
- Geringere Durchlaufzeit



- $p_{gesamt} = 0,4375$
- Geringerer Leistungsverbrauch
- Höhere Durchlaufzeit

Die HW-Beschreibungssprache VHDL

- **VHDL** = VHSIC HDL
- **VHSIC** = Very High Speed Integrated Circuits
- VHDL hervorgegangen aus dem VHSIC-Programm der USA
- Standardisierte Hardware-Beschreibungssprache
- Seit 1987 IEEE-Standard, mittlerweile überarbeitet
- Kann verschiedene Beschreibungen des gesamten Entwurfsablauf darstellen
  - Algorithmische Spezifikation (behavioral)
  - Realisierungsnahe Strukturen (RTL)
  - Simulation
- Vorsicht: **Nicht alle Sprachkonstrukte in jedem Entwurfsschritt nutzbar!**

Die Entwurfssprache VHDL

- Ursprünglich als Modellierungssprache nur für die Simulation konzipiert
- Zunehmender Einsatz als Sprache für Synthese und Verifikation
- heute: VHDL, Verilog, SystemC
- Einsatz zum ASIC- und FPGA-Entwurf, mit Erweiterungen auch Analog-Entwurf möglich
- VHDL umfaßt alle Elemente einer klassischen Programmiersprache (ADA)
- Erweitert um Konstrukte für Schaltungsentwurf

VHDL–Fallstricke

- **Mächtigkeit der Sprache** problematisch:
- Nicht alle Sprachkonstrukte können in Hardware umgesetzt werden
- Trennung zwischen simulierbar und synthetisierbar

- Abstraktion erzeugt eine – im Vergleich zu niederen Modellierungssprachen (ABEL, PALASM) – gewissen **Mehraufwand in der Beschreibung**
- Methoden der Abstraktion für Ein- und Umsteiger ggf. gewöhnungsbedürftig (z.B. Registerbeschreibung)

Grundlage: Spezifikation der Schaltung

- Schnittstellen (Zahl und Art der Ein-/Ausgänge)
- Gewünschtes Verhalten
- Eventuelle weitere Vorgaben bezüglich Geschwindigkeit, Kosten, Fläche, Leistungsverbrauch etc.

- Nur Schnittstellen und gewünschtes Verhalten werden typischerweise in VHDL formuliert
- Geschwindigkeitsvorgaben nur simulierbar (Einhalten von Zeitfenstern)
- Weitere Parameter Domäne des Synthesewerkzeugs

Entwurfsschritte

- **Verhaltensverfeinerung**
  - Detailliertes Ausarbeiten der gewünschten Funktionalität
  - Ersetzen von Black-Boxes
- **Strukturverfeinerung**
  - Realisierung einer spezifizierten Funktion durch Verschaltung von Komponenten mit einfacherer Funktionalität
- **Datenverfeinerung**
  - Realisierung abstrakter Datentypen durch einfachere Datentypen
  - Synthese: Binärer Datentyp bzw. erweiterter binärer Datentyp (incl. Tri-state-Zustand, `std_logic`)

→ **Schaltungsbeschreibung**

Erstellen einer Beschreibung

Schnittstellendefinition (entity)

- **Entity** beschreibt Ein-/Ausgabeschnittstelle
- Pro Modul nur eine Entity erlaubt

Verhaltensbeschreibung (architecture)

- Mehrere **Architectures** pro Modul möglich, z.B. zur Unterscheidung von Verhaltens- oder Synthesebeschreibung
- Keine begriffliche Verwandtschaft zu (Mikro)Architektur bei Prozessoren

Konfiguration und Zuordnung (configuration)

- Festlegung verwendeter Architecture
- Konfiguration, z.B. von Signalbreiten

Signalmodi

Signalmodus bestimmt Datenflußrichtung

- **in** kann nur gelesen werden
- **out** kann nur geschrieben werden
- **inout** bezeichnet bidirektionales Signal
- **buffer** ist ebenfalls bidirektional mit eingeschränkter Zuweisungsmöglichkeit (VHDL-FAQ: *"The use of buffer ports is discouraged"*)
- **linkage** dient zur Verbindung mit externen, nicht-VHDL Modulen; Semantik werkzeug- bzw. herstellerspezifisch
- Typischerweise nur **in, out, inout** in Verwendung
- Modi werden nur in Entity deklariert, nicht bei internen (in der Architecture spezifizierten) Signalen

VHDL-Datentypen

- **boolean**: True, False
- **bit**: 0, 1
- **std_logic**
  - Erweiterung von bit mit zusätzlich
  - **Z**: tristate / hochohmig; Signal mit diesem Wert kann von anderen Signalen mit 0 oder 1 überschrieben werden
  - **X**: unbekannt; Datenwert ist durch unvollständige oder fehlerhafte Berechung / Zusammenschaltung entstanden
  - **U**: undefiniert, d.h. uninitialisiert oder unbekannt
- Skalare Datentypen, Arrays (**_vector**), Integer, Characters
- Fließkomma, Datei, Records, eigene Typdefinitionen
- Synthetisierbarkeit beachten!

Zuweisungsoperatoren

- Zuweisung von **skalaren Werten**
  - a<=’1’;
- Zuweisung von **Vektoren**
  - a<=”1010”;
  - a<=(others=>’0’);
- Zuweisung von **Signalen**
  - a<=b;
- Zuweisung von **Termen**
  - a<=not(b);

Zu beachten:

- Zuweisungsoperator unterscheidet zwischen Signalen <= und Variablen :=
- VHDL führt strikte Typprüfung durch (auch für Operatoren)
- Ggf. Konvertierung (typecasting) notwendig

Operatoren

VHDL-Operatoren

- Logik: AND, OR, NAND, NOR, XOR, XNOR
- Vergleich: $=$, $/=$, $<$, $<=$, $>$, $>=$
- Schieben: SLL, SRL, SLA, SRA, ROL, ROR
- Arithmetik: +, -, *, /, MOD
- Diverse: **, ABS, NOT
- Wichtig: Operatoren innerhalb einer Gruppe haben **gleiche** Präzedenz
- AND/OR haben gleiche Priorität, d.h. gemischte Ausdrücke passend klammern, um Mehrdeutigkeiten zu vermeiden
- Ordnung der Gruppen nach aufsteigender Präzedenz
- Datentypen und Operatoren werden durch Bibliothek definiert; erst durch Benutzung von entsprechenden Bibliotheken werden Datentypen und Operatoren erst nutzbar

Auch hier gilt: nicht zwingend in Hardware abbildbar

Datenobjekte

- **Datenobjekte haben Typ und** – je nach Deklarationsort – **Modus**

Signale (signals)

- Häufigstes Datenobjekt
- Datentransport und Datenspeicherung innerhalb von Architectures und über deren Grenzen (Schnittstellen) hinaus.
- Wertezuweisung nebenläufig, wenn nicht explizit serialisiert (Prozesse)

Konstanten (constants)

- Statische Werte

Variablen (variables)

- Definition innerhalb des **process**-Headers
- Auf einzelnen Process beschränkt, kein Transport über process-Grenzen hinweg
- Wertezuweisung sequentiell und umittelbar in der Reihenfolge der Abarbeitung
- Unterschied zu Signalen!

Entwurfsbeispiel: NAND-Gatter

$$NAND : f(x,y) = \begin{cases} 0 & wenn\ x = 1 \wedge y = 1 \\ 1 & sonst \end{cases}$$

- Datentyp std_logic soll verwendet werden
- Einladen benötigter Bibliotheken

NAND-Gatter: Einladen benötigter Bibliotheken

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Entwurfsbeispiel (forts.)

$$NAND : f(x,y) = \begin{cases} 0 & wenn\ x = 1 \wedge y = 1 \\ 1 & sonst \end{cases}$$

- Zwei Eingänge x/y, ein Ausgang f(x,y)
- Interfacebeschreibung ergibt Entity

NAND-Gatter: Beschreibung der Schnittstellen

```
entity NAND is
  port(
    -- x,y sind Eingänge vom Typ std_logic
    x,y:  in std_logic;
    -- fxy ist Ausgang vom Typ std_logic
    fxy:  out std_logic
  );
end entity;
```

Entwurfsbeispiel (forts.)

$$NAND : f(x,y) = \begin{cases} 0 & wenn\ x = 1 \wedge y = 1 \\ 1 & sonst \end{cases}$$

- Funktionsbeschreibung ergibt Architecture

NAND-Gatter: Beschreibung des Verhaltens

```
architecture arch_NAND of NAND is
begin
    -- funktionale Beschreibung
    fxy<='0' when x='1' and y='1' else '1';
end architecture;
```

Entwurfsbeispiel (forts.)

$$NAND : f(x,y) = \begin{cases} 0 & wenn\ x = 1 \wedge y = 1 \\ 1 & sonst \end{cases}$$

- Funktionsbeschreibung ergibt Architecture

NAND-Gatter: Beschreibung des Verhaltens

```
architecture arch_NAND of NAND is
-- Hilfssignal
signal f_and:  std_logic;
begin
    -- algorithmische Beschreibung
    f_and<=x and y;
    fxy<=not(f_and);
end architeture;
```

Entwurfsbeispiel (forts.)

$$NAND : f(x,y) = \begin{cases} 0 & wenn\ x = 1 \wedge y = 1 \\ 1 & sonst \end{cases}$$

- Weiterverwendung des NAND-Gatters zur Instantiierung in anderen Architectures (**port map**-Statement)
- Komponentenbeschreibung
- Ggf. Zusammenfassung mehrer Komponenten in eigenem **package** zum Aufbau einer Bibliothek (→ **use**-Statement)

NAND-Gatter: Komponentenbeschreibung

```
component NAND is
  port(
    x,y:  in std_logic;
    fxy:  out std_logic
  );
end component;
```

Prozesse und Sequentialität

- Zuweisungen in VHDL grundsätzlich nebenläufig
- Modellierung von Schaltwerken explizit in Prozessen (**process**)
- Beschreibung des Verhaltens über einen sequentiellen Algorithmus
- Simulierte Zeit für die gesamte VHDL-Beschreibung schreitet während Ausführung nicht fort

Aufbau eines Prozesses

- Sensitivity List; Signale, bei deren Änderung der Beschreibungsblock ausgeführt wird
- Rückhalten der Zuweisungen bis Blockende
- Variablendeklaration (**:=**), Zuweisung sofort
- Verhaltensbeschreibung

Prozeß-Sprachkonstrukte

Warten auf Ereignis

- **wait for**
- **wait until**
- Problematisch bei Synthese!

Bedingte Zuweisung

- **if/then/else** (entspricht **when/then**-Konstrukt bei nebenläufiger Zuweisung)
- **case/when**-Zuweisung

Schleifen

- **for/loop**
- **while/loop**
- Auch Zuweisungen innerhalb einer Schleife erfolgen erst zum Ende des Blocks
- Ebenfalls problematisch bei Synthese

Modellierung von Speicherelementen

- VHDL verfügt über **kein explizites Sprachelement** zur Erzeugung von Speicherelementen

Modellierung von Speicherelementen

- Erzeugung über if-Konstrukt in Prozessen
- Asynchrones Speicherelement (Latch, pegelgesteuert)
- Synchrones Speicherelement (Flip-Flop, taktflankengesteuert)
- Spezifikation über Signalattribut (**'event**) und Signalpegel
- Pro **process** und Taktsignal kann nur eine Taktflanke herangezogen werden
- Speicherelementtyp (D, T, RS, JK) mittels Ausformulierung des gewünschten Verhaltens

Speicherelemente (forts.)

- Beispiel: **D-Latch**
- Einspeichern von Daten während low-Pegel des Kontrollsignals möglich
- Verriegelt während high-Pegel

Modellierung eines D-Latchs

```
D_LATCH:
process(write,data)
begin
  if write='0' then
    dlatch<=data;
  end if;
end process;
```

Speicherelemente (forts.)

- Beispiel: **D-Flipflop**
- Übernahme von Daten zur steigenden Taktflanke eines Taktsignals

Modellierung eines D-Flipflops

```
D_FF:
process(clk,data)
begin
  if clk'event and clk='1' then
    dff<=data;
  end if;
end process;
```

Speicherelemente (forts.)

- Beispiel: **D-Flipflop mit asynchronem Rücksetzeingang**
- Übernahme von Daten zur steigenden Taktflanke eines Taktsignals
- Löschen des Inhaltes asynchron, d.h. unabhängig vom Taktsigal, bei low-Pegel des Rücksetzsignals

Modellierung eines D-Flipflops

```
D_FFR:
process(rst,clk,data)
begin
  if rst='0' then
    dffr<=(others=>'0');
  elsif clk'event and clk='1' then
    dffr<=data;
  end if;
end process;
```

Entwurfsbeispiel: Register-File

- **16 Register zu je 8-Bit**
- Schreib- und lesbar
- Steuersignale: Takt, Rücksetzeingang, Chip-Select
- Zugriff via Adreß/Datenbus

Register-File: Interface

```
entity register is
  port(
    clk, rst, cs, rw:  in std_logic;
    addr:  in std_logic_vector(3 downto 0);
    data:  inout std_logic_vector(7 downto 0)
  );
end entity;
```

Entwurfsbeispiel (forts.)

- 16 8-Bit Register → **zweidimensionales Array**
- **In VHDL nicht direkt abbildbar**, d.h. zweistufiges Verfahren
  - Definition eines Subtypen für 8-Bit Arrays
  - Signaldeklaration
- Ausgabe nicht direkt auf Datenbus sondern in internes Signal

Register-File: Deklarationen

```
architecture arch_reg of register is
subtype sdlv8 is std_logic_vector(7 downto 0);
signal register:  sdlv8_vector(15 downto 0);
signal data_out:  std_logic_vector(7 downto 0);

begin
  ...
end architecture;
```

Entwurfsbeispiel (forts.)

- **Prozeßdeklaration**
  - Steuersignale sind Takt, Reset, Chip-Select und R/W
  - Auswahl mittels Adreßleitungen

Register-File: Deklarationen und Reset)

```
process(clk,rst,cs,rw,addr)
begin
 if rst='0' then
   register(0)<=(others=>'0');
   register(1)<=(others=>'0');
   ...
   -- alternativ:  generate-Statement
```

Entwurfsbeispiel (forts.)

- Zugriff synchron zu Takt
- Zugriffsart bestimmt durch R/W
- Gültigkeit des Zugriffs durch Chip-Select
- Schreiben synchron, Lesen asynchron

Register-File: Zugriffe

```
    elsif clk'event and clk='1' then
      if rw='0' and cs='0' then
        -- Datum in Register schreiben
      end if;
    end if;
  end process;
  -- Datum aus Register lesen
end architecture;
```

Entwurfsbeispiel (forts.)

- Register lesen: nebenläufig, nicht taktflankengesteuert
- Ausgabe auf Datenbus gesteuert durch R/W und Chip-Select
- Nebenläufige Zuweisung außerhalb des Prozesses

Register-File: Asynchroner Lesezugriff

```
data<=data_out when rw='1' and cs='0'
 else (others=>'Z');
data_out<=register(0) when addr="0000"
     else register(1) when addr="0001"
     ...
```

Entwurfsbeispiel (forts.)

- Register schreiben: synchron zu Taktflanke
- Plazierung innerhalb des Prozesses

Register-File: Synchroner Schreibzugriff

```
case addr is
  when "0000" => register(0)<=data;
  when "0001" => register(1)<=data;
  ...
  when others => null;
end case;
```

Weiterführende Literatur

Literatur

- VHDL-Kurzanleitung von Georg Acher und Markus Leberecht

  `http://www.lrr.in.tum.de/~acher/tgi/uebung/VHDL-Buch.pdf`

- VHDL-Cookbook von Peter J. Ashenden mit detaillierter Einführung und sehr ausführlichem Entwurfsbeispiel (DP32-Prozessor)

  `http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`

Für Interessierte

- The Hamburg VHDL Archive

  `http://tams-www.informatik.uni-hamburg.de/vhdl/`

- Freie IP-Cores in VHDL und Verilog

  `http://www.opencores.org`

Aufgabe

Eine Zählerschaltung soll entwickelt werden. Diese Schaltung soll folgendes Verhalten aufweisen:

- Ein low-aktives Rücksetzsignal löscht den Zähler.
- Über ein Richtungssignal wird bestimmt, ob der Zähler mit der steigenden Flanke eines Taktsignals aufwärts (=0) oder abwärts (=1) zählt.
- Es wird nur gezählt, wenn der Zähler mit einem high-aktiven Auswahl-Signal freigeschaltet ist.
- Der Zähler soll 64 Zählschritte ausführen können.
- Ein low-aktives Freigabesignal entscheidet, ob der Zählerausgang auf einen gemeinsamen Bus gelegt werden soll; bei nicht erfolgter Freigabe werden die Ausgabeleitungen in den tri-state-Zustand geschaltet.

Aufgabe (forts.) – entity

- Erstellen Sie die zugehörige Schnittstellenbeschreibung in VHDL.

Aufgabe (forts.) – entity

- Erstellen Sie die zugehörige Schnittstellenbeschreibung in VHDL.

Schnittstelle enthält alle nach außen sichtbaren Signale mit Datentyp und Richtung, also:

```
entity counter is
  port(
    rst: in std_logic;   -- Rücksetzsignal
    clk: in std_logic;   -- Taktsignal
    dir: in std_logic;   -- Richtungsumschaltung
    sel: in std_logic;   -- Auswahlsignal
    ena: in std_logic;   -- Freigabesignal

    c_out: out std_logic_vector(5 downto 0)  -- Zählerausgabe
  );
end entity;
```

Es ist alternativ auch möglich, Signale gleichen Typs und Richtung zu gruppieren, z.B. `rst, clk, dir, sel, ena: in std_logic;`.

Aufgabe (forts.) – architecture

- Formulieren Sie die entsprechende Verhaltensbeschreibung in VHDL.

Aufgabe (forts.) – architecture

- Formulieren Sie die entsprechende Verhaltensbeschreibung in VHDL.

Problemfall: `c_out` ist als reines Ausgabesignal deklariert und kann nicht als eigentlicher Zähler verwendet werden. In der Verhaltensbeschreibung muß zusätzlich der eigentliche Zähler als Signal deklariert werden. Die Ausgabe des Zählers erfolgt außerhalb des Prozesses.

Aufgabe (forts.) – architecture

- Formulieren Sie die entsprechende Verhaltensbeschreibung in VHDL.

Problemfall: `c_out` ist als reines Ausgabesignal deklariert und kann nicht als eigentlicher Zähler verwendet werden. In der Verhaltensbeschreibung muß zusätzlich der eigentliche Zähler als Signal deklariert werden. Die Ausgabe des Zählers erfolgt außerhalb des Prozesses.

```
architecture arch_counter of counter is
  signal count: unsigned(5 downto 0);  -- 64 Zustände
begin

  -- Ausgabe
  c_out<=count when ena='0' else (others=>'Z');

end architecture;
```

Aufgabe (forts.) – architecture/process

Prozeß zerfällt in asynchrones Rücksetzen und eigentlichen Zählvorgang.

```
p_counter:
process(rst,clk,dir,sel)
begin

  -- asynchrones Rücksetzen
  if rst='0' then
    count<=(others=>'0');
  -- Zählfunktion
  elsif clk'event and clk='1' then

  -- Zähler selektiert?
    if sel='1' then

  -- Zählrichtung
      if dir='0' then
        count<=count+1;
      else
        count<=count-1;
      end if;

    end if;
  end if;
end process;
```

Aufgabe (forts.)

Der Zähler soll um eine Über-/Unterlaufsfunktion ergänzt werden, d.h. beim Erreichen des Wertes 0 wird für die Dauer eines Taktes ein Anzeigesignal ausgegeben; hierbei soll das Rücksetzsignal nicht fälschlicherweise das Überlaufssignal auslösen. In der Entity sei hierzu das zusätzliche signal `ovl` vom Typ `std_logic` mit Modus `out` deklariert.

- Erweitern Sie die Verhaltensbeschreibung aus Teilaufgabe b) um eine Lösung, bei der das Überlaufssignal rein innerhalb des Prozesses erzeugt wird. Was ist bei dieser Lösung zu beachten und warum? Wozu würde `ovl` in dieser Implementation synthetisiert?
- Erweitern Sie die Verhaltensbeschreibung aus Teilaufgabe b) um eine Lösung, bei der das Überlaufssignal außerhalb des Prozesses erzeugt wird.

→ Lösungsblatt