Digitaltechnik - Zusammenfassung

Patrick Pletscher

9. September 2004

1 VHDL

1.1 VHDL Entities und Architectures

VHDL Modelle bestehen aus mehreren Entities, jede Entity hat eine `entity` Deklaration und ein oder mehrere `architecture` Implementierungen. Implementierungen können keinen Zustand haben wie Objekte.

Einfaches Beispiel

```
entity And2 is
  port (x, y: in bit; z: out bit);
end entity And2;

architecture ex1 of And2 is
begin
  z <= x and y;
end architecture ex1;
```

Erläuterungen

- `bit` einfacher Datentyp mit Konstanten `'0'` und `'1'`. Characters!
- mit `<=` wird zugewiesen
- `in` Eingang, `out` Ausgang, `inout` bidirektionale Ports
- `not` hat höchste Priorität
- `and, or, ...` haben *dieselbe* Priorität

1.2 Bezeichner

Es wird nicht zwischen Gross- und Kleinschreibung unterschieden. Es gibt einen sogenannten Escape-Mechanismus: `\$ein fancy%Bezeichner\`. Zeilenweise Kommentare beginnen mit `--`.

1.3 Kombinatorische RTL Beschreibung

RTL ist eine Abkürzung für register transfer level, der VHDL Code beschreibt also wie Daten von einem Register zum nächsten übergeben werden, die Transformation der Daten wird mit der kombinatorischen Logik zwischen den Registern gemacht, man kann RTL Code aber auch ohne Register benutzen und pure kombinatorische Logik damit abbilden.

```
entity combinational_function is
  port (a, b, c: in bit; z: out bit);
end entity combinational_function;

architecture expression of combinational_function is
begin
  z <= (not a and b) or (a and c);
end architecture expression;
```

1.4 Netzlisten-Implementierung

Implementierung von `combinational_function`.

Beschreibung Bibliotheks-Schaltkreise

```
entity Or2 is
  port (x, y: in bit; z: out bit);
end entity Or2;

architecture ex1 of Or2 is
begin
  z <= x or y;
end architecture ex1;

entity Not1 is
  port (x: in bit; z: out bit);
end entity Not1;

architecture ex1 of Not1 is
begin
  z <= not x;
end architecture ex1;
```

Manuelle Synthese - Übersetzung in Netzliste

```
architecture netlist of combinational_function is
  component And2 is
    port (x, y: in bit; z: out bit);
  end component And2;
  component Or2 is
    port (x, y: in bit; z: out bit);
  end component Or2;
  component Not1 is
    port (x: in bit; z: out bit);
  end component Not1;
  signal p, q, r: bit;
begin
```

```
  g1 : Not1 port map (a, p);
  g2 : And2 port map (p, b, q);
  g3 : And2 port map (a, c, r);
  g4 : Or2 port map (q, r, z);
end architecture netlist;
```

Bemerkung: Das Ganze ist kompilierbar, ohne dass die components definiert sind, auch muss keine Auswahl der Architecture getroffen werden.

Erklärungen

- verwendete externe Komponenten werden mit `component` deklariert
- interne Signale (Drähte) werden durch `signal` erzeugt
- Zusammenstöpseln durch Instanzierung `g1 : ...`
- Verbinden der Ports und Signale durch `map`
- man kann auch eine explizite Signalzuordnung verwenden:
  `g2: And2 port map (z => q, x => p, y => b);`

Instanzierung ohne Komponenten-Deklaration

```
architecture direct of combinational_function is
  signal p, q, r: bit;
begin
  g1 : entity work.Not1(ex1) port map (a, p);
  g2 : entity work.And2(ex1) port map (p, b, q);
  g3 : entity work.And2(ex1) port map (a, c, r);
  g4 : entity work.Or2(ex1) port map (q, r, z);
end architecture direct;
```

`work.Or2(ex1)` ist Pfadangabe für die Entity `Or2` und als Architecture wird `ex1` gewählt.

1.5 Wie wählt man die Architektur aus?

Zuletzt verwendete Architektur wird verwendet.

Man kann die Architektur selber explizit Instanzieren, `work.Or2(ex1)` oder bei Variablendefinitionen usw. `for all: And2 use entity work.And2(ex1);`.

Oder per Konfiguration:

```
configuration use_expr of And2_tb is
  for assignment_only
    for g1 : And2
      use entity work.And2(ex1);
    end for;
  end for;
end configuration use_expr;
```

1.6 Zuweisung von Signalen

```
z <= x and y;
```

Zuweisung mit Delay

```
z <= x after 5 ns;
```

1.7 Generics

Deklaration:

```
entity And2 is
  generic (delay: delay_length);
  port (x,y: in bit; z: out bit);
end entity And2;

architecture ex2 of And2 is
begin
  z <= x and y after delay;
end architecture ex2;
```

Instanz:

```
g2: And2 generic map (delay => 5 ns)
 port map (p, b, q);
```

- Generische Parameter: Konstanten bekannt zur Kompilierzeit.
- Defaultwerte: `generic (delay: delay_length := 5 ns);`. Defaultwerte müssen nicht durch `map` verbunden werden.
- `open` steht für den Defaultwert oder einen unverbunden Ausgang
- Konstanten können an Eingänge gelegt werden

1.8 Testbench

```
entity And2_tb is
end entity And2_tb;

architecture assignment_only of And2_tb is
  signal a, b, o1, o2, check: bit;
  constant period: time := 50 ns;
begin
  g1: entity work.And2(ex1)
   port map (a, b, o1);
  g2: entity work.universal(expr)
   port map (a, b, '0',o2, open);
  check <= o1 xor o2;
  a <= '0', '1' after period,
   '0' after 2*period, '1' after 3*period;
  b <= '0', '1' after 2*period,
   '0' after 4*period;
end architecture assignment only;
```

`check` ist auf 1 gdw. `g1` und `g2` dieselbe Ausgabe produzieren.

Testbench mit Prozessen

Man kann auch einen Prozess zur Erzeugung der Eingabe-Belegungen verwenden:

```
run: process
begin
  a <= '0'; b <= '0'; wait for period;
  a <= '0'; b <= '1'; wait for period;
  a <= '1'; b <= '0'; wait for period;
  a <= '1'; b <= '1'; wait for period;
  a <= '0'; b <= '0'; wait for period;
  wait;
end process run;
```

1.9 Vektoren

```
type bit_vector is array (natural range <>) of bit;
```

`natural` sind die natürlichen Zahlen, `range <>` ist eine undefinierte Index-Menge.

```
entity GrayEncoder is
  port (binary: in  bit_vector (2 downto 0);
        gray  : out bit_vector (2 downto 0));
end entity GrayEncoder;

architecture when_else of GrayEncoder is
begin
  gray <= "000" when binary = "000" else
          --...
          "100";
end architecture when_else;
```

Arrays sind steigende(`to`) oder fallende (`downto`) Indizes. Einschränkung auf Unterbereiche: z.B. `binary (2 downto 1)`.

Man kann das Selbe auch mit `with-select`machen:

```
--..
gray <= "000" when "000",
        --...
        "100" when others;
--..
```

Es müssen alle Fälle explizit behandelt werden.

2 Digitaltechnik

EDA = Electronic Design Automation
PCB = Printed Circuit Boards. Zusammenlöten von kleinen ICs
VLSI = Very Large Scale Integration

2.1 PLA

Jemeils normales und negiertes Eingangssignal, welche in einer UND-Matrix verundet werden können (horizontal) und eine ODER-Matrix, welche vertikal die Signale verodert und auf ein Ausgangssignal gibt.
Man muss also seine logische Formel in eine DNF bringen.

2.2 Disktretisierung

Jede grössere Wertemenge W lässt sich boolesch kodieren durch ein γ:

$$\gamma : W \to 2^n \quad \text{mit } n = \lceil \log_2 |W| \rceil$$

2.3 Einerkomplement

Die $(n+1)$-stellige Binärzahl $s_n d_{n-1} \cdots d_0$ im Einerkompliment kodiert

$$\text{die positive Zahl} \quad \sum_{i=0}^{n-1} d_i \cdot 2^i \quad \text{falls } s_n = 0$$

$$\text{die negative Zahl} \quad -\sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i \quad \text{falls } s_n = 1$$

Trick: Bits Umdrehen, als Integer Negieren

2.4 Zweierkomplement

Die (n+1)-stellige Binärzahl $s_n d_{n-1} \cdots d_0$ im Zweierkomplement kodiert

$$\text{die positive Zahl} \quad \sum_{i=0}^{n-1} d_i \cdot 2^i \quad \text{falls } s_n = 0$$

$$\text{die negative Zahl} \quad -(1 + \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i) \quad \text{falls } s_n = 1$$

Trick: Bits Umdrehen, Eins Hinzuzählen, als Integer Negieren.

2.5 Festkommazahlen

Genauigkeit m der Nachkommastellen wird fixiert, z.B. $m = 2$ Binär-Stellen.

2.6 Fliesskommazahlen für 32 Bit

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| S |    E    |    F    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0   1         9         31
```

S: Vorzeichen
E: Exponent 8 Bits
F: Mantisse 23 Bits

Die Zahl ist:

$$V = (-1)^S \cdot 1.F \cdot 2^{E-127} \quad \text{falls } 0 < E < 255$$

Ausnahmen:

$E = 255, F \neq 0$	$V = NaN$ (not a number)
$E = 255, F = 0, S = 0$	$V = Inf(= \infty)$
$E = 255, F = 0, S = 1$	$V = Inf(= -\infty)$
$E = 0, F \neq 0$	$V = (-1)^s \cdot 0.F \cdot 2^{-126}$ (nicht normalisierte Zahl)
$E = 0, F = 0, S = 1$	$V = -0$
$E = 0, F = 0, S = 0$	$V = 0$

2.7 Aufzählungstypen

One-Hot Encoding

m Werte werden durch m Bits kodiert

Gray-Codes

Aufeinanderfolgende Kombinatioen unterscheiden sich an einer Bitposition.

Ein Gray-Code für p Bits kann in einen Gray-Code für $p+1$ Bits erweitert werden, indem der Code für p Bits gespiegelt angehängt wird. Vor die Codes der ersten Hälfte wird 0 gestellt, vor die der zweiten Hälfte 1.

```
procedure gray(i, p)
begin
  if p = 1 then output "i"
  elsif i < p**(p-1) then
    output "0";
    gray(i, p-1)
  else
    output "1";
    gray((2**p-1)-i, p-1)
  end
end gray
```

3 CMOS

Hierbei wird mit Spannungspegeln gearbeitet.

3.1 Schaltbilder

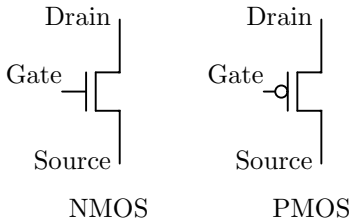


Abbildung 3.1: *CMOS Schaltbilder*

NMOS (= pull-down) muss immer auf GND liegen und PMOS (= pull-up) immer auf VDD. NMOS schaltet das Signal Source nach Drain durch, wenn an Gate eine logische 1 anliegt, welche durch VDD repräsentiert wird. PMOS schaltet das Signal Source nach Drain, wenn an Gate eine logische 0 anliegt, welche durch GND repräsentiert wird.

3.2 Inverter

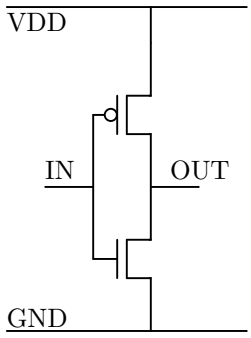


Abbildung 3.2: *CMOS Inverter*

3.3 NOR

Ausgang 1 gdw. beide Eingänge 0.

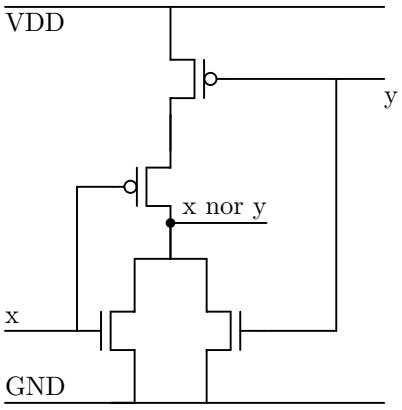


Abbildung 3.3: *CMOS NOR*

3.4 NAND

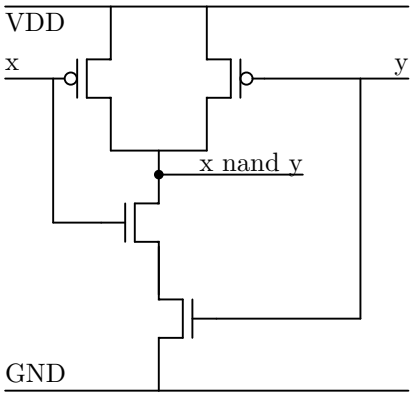


Abbildung 3.4: *CMOS NAND*

3.5 Threestate-Buffer mit Transmission-Gate

3 logische Ausgabewerte: 0, 1, oder gar kein Signal (hochohmig).
in wird negiert durchgeleitet zu out gdw. en auf 1 liegt, sonst ist out hochohmig.

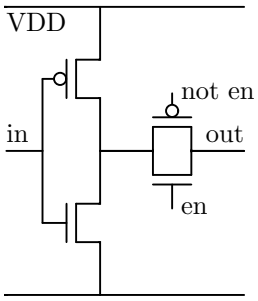


Abbildung 3.5: *Threestate Buffer*

3.6 Multiplexer

`out = if z then y else x` oder $(z \wedge y) \vee (\neg z \wedge x)$.

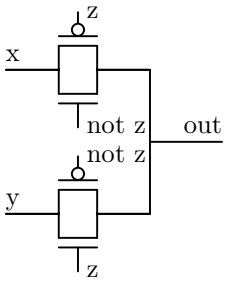


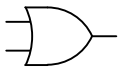
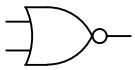
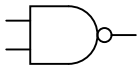
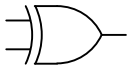
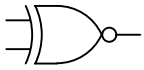
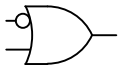
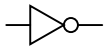
Abbildung 3.6: *Multiplexer*

3.7 Störabstand

Der Transistor garantiert, dass seine Ausgangsspannung sehr nahe bei VDD oder GND ist. Er akzeptiert aber als Eingangsspannungen auch viel grössere Differenzen von GND und VDD, somit können Störungen usw. abgefangen werden.

4 Kombinatorische Schaltungen

4.1 Gatter

Bezeichnung (VHDL)	Boolesche Logik	Algebraische Schreibweise	Traditionelles Schaltbild
Disjunktion (`or`)	$x \vee y$	$x + y$	
Negierte Disjunktion (`nor`)	$\neg(x \vee y)$	$\overline{x + y}$	
Konjunktion (`and`)	$x \wedge y$	$x + y$	
Negierte Konjunktion (`nand`)	$\neg(x \wedge y)$	$\overline{x \cdot y}$	
Exklusiv-Oder (`xor`)	$x \not\leftrightarrow y$	$x \oplus y$	
Äquivalenz (`xnor`)	$x \leftrightarrow y$	$\overline{x \oplus y}$	
Implikation (N/A)	$x \rightarrow y$		
Negation (`not`)	$\neg x$	$\overline{x}$	

4.2 Rechenregeln der Booleschen Algebra

Konjunktive Formulierung	Disjunktive Formulierung
$x \wedge y \equiv y \wedge x$	$x \vee y \equiv y \vee x$
$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$	$x \vee (y \vee z) \equiv (x \vee y) \vee z$
$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
$\neg(x \wedge y) \equiv \neg x \vee \neg y$	$\neg(x \vee y) \equiv \neg x \wedge \neg y$
$x \wedge x \equiv x$	$x \vee x \equiv x$
$x \wedge 0 \equiv 0$	$x \vee 1 \equiv 1$
$x \wedge 1 \equiv x$	$x \vee 0 \equiv x$

$$\neg(\neg x) \equiv x$$

Theorem 4.1 (Shannonsche Expansion). *Seien e, f Boolesche Ausdrücke, x Variable, $e[f/x]$ Substitution von x durch f in e.*

$$e \equiv x \wedge e[1/x] \vee \neg x \wedge e[0/x]$$

Abschwächung $x \cdot y \equiv x$, falls $x \Rightarrow y$ (y schwächer als x)

Verstärkung $x + y \equiv x$, falls $y \Rightarrow x$ (y stärker als x)

Konsensus (Fallunterscheidung und Abschwächung)

$$x \cdot y + \overline{x} \cdot z + y \cdot z \equiv x \cdot y + \overline{x} \cdot z$$

4.3 Disjunkte Normalform (DNF)

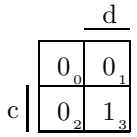
Eine DNF ist eine Disjunktion einer Konjuktion von Literalen.

Ein *Literal* ist eine negierte oder unnegierte Variable, ein *Produktterm* ist eine Konjunktion von Literalen, man bezeichnet einen Produktterm auch als *Monom*, ein *Minterm* ist ein maximaler Produktterm.

4.4 Karnaugh Diagramme

Man wählt eine "Reihenfolge" der Variablen und codiert sie binär.

Beispiel:



Index	c	d	c and d
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

Minimierung von Karnaugh Diagrammen

1. Konstruiere und Bezeichne das Gitter
2. Trage Minterme von der Funktionstabelle ein oder von der DNF
3. Suche und Markiere maximale Blöcke
4. Wähle minimale Listen von nicht-redundanten maximalen Blöcken
5. Generiere Produktterme aus Blöcken

Implikanten und Primimplikanten

Ein *Implikant* einer DNF ist ein implizierter Produktterm, ein Block im Karnaugh-Diagramm. Ein *Primimplikant* ist ein maximaler Implikant, entspricht einem maximalen Block, der in keine Richtung erweitert werden kann. Ein *Kernimplikant* überdeckt ein sonst nicht überdeckten Minterm.

4.5 Abstrakte Minimierungsmethode

1. Generiere alle Primimplikanten durch einfache Konsusregel (diese Terme überdecken am Anfang nur einen Block in einem entsprechenden Karnaugh Diagramm)
2. Gib Kernimplikanten aus, da in jeder minimalen DNF enthalten
3. Entferne redundante Primimplikanten
4. Wähle minimale Überdeckung aus restlichen Primimplikanten

Quine-McClusky-Verfahren

1. Funktionstabelle für alle Kombinationen der Variablen
2. Uns interessieren nur die Zeilen, bei welchen der Funktionswert 1 ist. Diese Zeilen werden nach den Anzahl Einsen (in den Variablen) geordnet
3. Von allen möglichen Paaren von zwei aufeinanderfolgenden Anzahl Einsen nimmt man diese zusammen, für welche man für eine Variable ein "don't care" einfügen kann (Konsensus). Wiederhole solange bis nur noch Primimplikanten vorhanden sind.
4. Man wählt danach alle Kernimplikanten aus und geht von den Grössten Termen (bzgl. abgedeckten Blöcken) zu den kleinsten.

Beispiel *Schritt 1*

Minterme Indizes	a	b	c	d	Anzahl Einsen	Primimpl.
0	0	0	0	0	0	nein
2	0	0	1	0	1	nein
8	1	0	0	0	1	nein
5	0	1	0	1	2	nein
10	1	0	1	0	2	nein
12	1	1	0	0	2	nein
13	1	1	0	1	3	nein
15	1	1	1	1	4	nein

Schritt 2

Minterme Indizes	a	b	c	d	Anzahl Einsen	Primimpl.
0,2	0	0	-	0	0	nein
0,8	-	0	0	0	0	nein
2,10	-	0	1	0	1	nein
8,10	1	0	-	0	1	nein
8,12	1	-	0	0	1	ja
5,13	-	1	0	1	2	ja
12,13	1	1	0	-	2	ja
13,15	1	1	-	1	3	ja

Schritt 3

Minterme Indizes	a	b	c	d	Anzahl Einsen	Primimpl.
0,2,8,10	-	0	-	0	0	ja
0,8,2,10	-	0	-	0	0	redundant

5 Blocks

5.1 Standard Logik IEEE 1164

Neunwertige Logik.

`'U'`	Uninitialisiert
`'X'`	Stark Unbekannt
`'0'`	Starke 0
`'1'`	Starke 1
`'Z'`	Hochohmig
`'W'`	Schwach Unbekannt
`'L'`	Schwache 0
`'H'`	Schwache 1
`'-'`	Don't Care

Bedeutungen

$0 =$ falsch
$1 =$ wahr
$Z =$ inaktiv
$X =$ unbekannt/Widerspruch

Mit Ausnahme von resolve ist das Ergebnis einer Operation mit `'Z'` immer `'X'`.

AND für die 4 wichtigsten

and	X	0	1	Z
X	X	0	X	X
0	0	0	0	0
1	X	0	1	X
Z	X	0	X	X

NOT für die 4 wichtigsten

not X	X
0	1
1	0
Z	X

OR für die 4 wichtigsten

or	X	0	1	Z
X	X	X	1	X
O	X	0	1	X
1	1	1	1	1
Z	X	X	1	X

XOR für die 4 wichtigsten

xor	X	0	1	Z
X	X	X	X	X
O	X	0	1	X
1	X	1	0	X
Z	X	X	X	X

Resolution

```
subtype std_logic is resolved std_ulogic;
```

`std_logic` erlaubt mehrere Treiber, d.h. mehrere Zuweisungen.
`std_ulogic` erlaubt nur einen Treiber, d.h. nur eine Zuweisung.
Konflikte sind in einer Resolutions-Tabelle aufgelöst.

resolve	X	0	1	Z
X	X	X	X	X
O	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

Verwendung des Standards

```
library ieee;
use ieee.std_logic_1164.all;
```

Sollte vor jeder Entity stehen, Deklaration hat keinen File-Scope, gilt nur bis zur nächsten Entity.

Bei `when-else` Konstrukten müssen jetzt alle restlichen Fälle behandelt werden. Meist mit `"XXX"`

Beispiel: Gray Encoder mit Standard Logik

```
library ieee;
use ieee.std_logic_1164.all;
entity GrayEncoder is
  port (binary: in  std_logic_vector (2 downto 0);
        gray  : out std_logic_vector (2 downto 0));
end entity GrayEncoder;

architecture when_else of GrayEncoder is
begin
  gray <= "000" when binary = "000" else
          --...
          "XXX";
end architecture when_else;
```

5.2 Generic Decoder

n Eingabe-Bits werden in 2^n Ausgabe-Bits umgewandelt. Binär Kodierung zu One-Hot-Kodierung.

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
  generic (n: positive);
  port (a: in std_ulogic_vector (n-1 downto 0);
        z: out std_ulogic_vector
         (2**n-1 downto 0));
end entity decoder;

architecture rotate of decoder is
  constant tmp: bit_vector (2**n-2 downto 0)
   := (others => '0');
  constant z_out: bit_vector (2**n-1 downto 0)
   := tmp & '1';
begin
  z <= st_StdULogicVector(z_out sll
   to_integer(unsigned(a)));
end architecture rotate;
```

5.3 std_match und Don't Cares

Mit Hilfe von `std_match` kann man Don't Cares erst ausnützen.

```
Y <= "01" when std_match(a, "001-") else
     --...
```

5.4 Weitere Modellierungen

Drei Möglichkeiten der Modellierung:

1. Strukturell: Instanzierung von Komponenten (z.B. Netzliste)
2. Datenfluss: durch Ausdrücke und `when-else`
3. Sequentiell: mit Prozessen (nicht unbedingt sequentielle Schaltkreise)

Sequentielle Modellierung am Beispiel Priority Encoder

```
library ieee;
use ieee.numeric_std.all;
architecture seq1 of prenc is
begin
  process (a) is
  begin
    valid <= '1';
    if std_match(a, "0001") then
      y <= "00";
    elsif std_match(a, "001-") then
      y <= "01";
    elsif std_match(a, "01--") then
      y <= "10";
    elsif std_match(a, "1---") then
      y <= "11";
    else
      y <= "00";
      valid <= '0';
    end if;
  end process;
end architecture seq1;
```

Der Prozess wird evaluiert, falls a sich ändert.

Variablen in Sequentieller Modellierung

```
-- Deklaration
variable even : std_ulogic;
-- Zuweisung
even := not even;
```

6 Sequentielles System

Sequentiell Zustandsbehaftet, vergangenes Verhalten und Eingaben bestimmen momentanen Zustand.

Kombinatorisch Zustandsfrei, keine Seiteneffekte, Ausgaben sind funktional von den Eingaben abhängig.

6.1 Allgemeines Sequentielles System

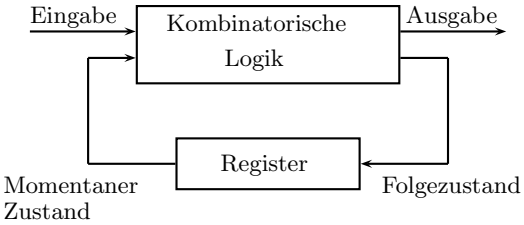


Abbildung 6.7: *Allgemeines Sequentielles System*

6.2 Synchron vs Asynchron

Synchron globales Taktsignal, Zustandsänderung zu fest spezifizierten Zeiten

Asynchron Zustandsänderung zu beliebigen Zeitpunkten

6.3 D-Flipflop

Speichert Eingang als Zustand für einen Taktzyklus. Gibt nach einem Takt die Eingabe aus. Dreieck bedeutet *flankengesteuert*, entweder steigende oder fallende Flanke; meistens wird die steigende Flanke verwendet. Oft gibt es auch einen zusätzlichen asynchronen *reset* oder *set* Eingang.

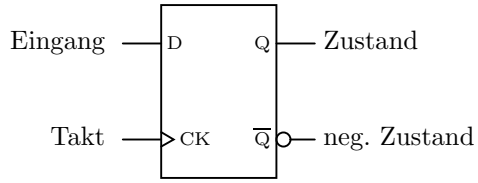


Abbildung 6.8: *D-Flipflop*

D-Flipflop mit positiver Flankensteuerung

D	C	Q'	$\overline{Q}'$
0	$\uparrow$	0	1
1	$\uparrow$	1	0
-	$\downarrow$	Q	$\overline{Q}$
-	0	Q	$\overline{Q}$
-	1	Q	$\overline{Q}$

Tabelle 1: *Übergangstabelle. Strich in Q' bedeutet Wert im nächsten Zustand (keine Negation)*

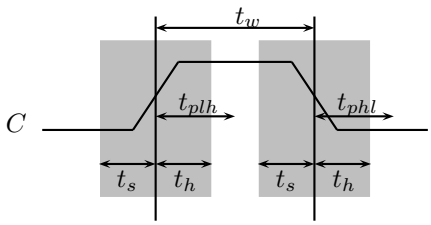


Abbildung 6.9: *Ablauf eines Clockticks*

Eingang stabil während *setup* Phase t_s vor der Flanke. Eingang stabil während *hold* Phase t_h nach der Flanke. Ausgang stabil erst nach *propagation* Phase t_{plh} bzw. t_{phl} nach der Flanke. Minimale Clock-Signallänge t_w.

6.4 Automaten

Determinismus

Nicht-Deterministische Automaten sind Automaten, für die gilt:

1. Deadends: Zustände ohne Nachfolgezustand
2. Zustände mit mehreren möglichen Nachfolgern

6.5 Maschinen

Moore Maschine

Ausgaben hängen nur vom momentanen Zustand ab und ändern mit der Clock-Flanke

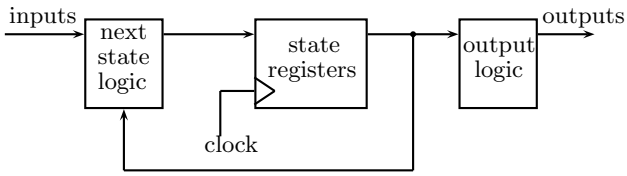


Abbildung 6.10: *Moore Maschine*

Mealy Maschine

Ausgaben hängen vom momentanen Zustand und den aktuellen Eingaben ab.

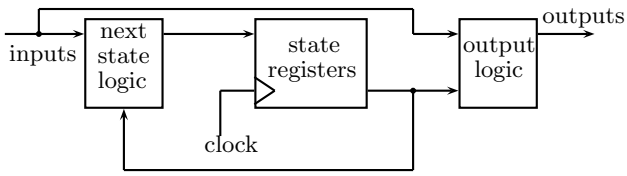


Abbildung 6.11: *Mealy Maschine*

Formale Notation

Menge von Zuständen S, Menge von Eingaben I, Menge von Ausgaben O.

Moore Maschine: $(S, I, O, \delta, \lambda)$ mit $\delta : S \times I \to S$ und $\lambda : S \to O$
Moore Maschine: $(S, I, O, \delta, \lambda)$ mit $\delta : S \times I \to S$ und $\lambda : S \times I \to O$

Booleschse Kodierung: $S = 2^n,\ I = 2^m,\ O = 2^k$

$\delta = (\delta_i)$ mit $\delta_i : 2^{n+m} \to \{0, 1\}$ für $i = 0 \ldots n$
$\lambda = (\lambda_j)$ mit $\lambda_j : 2^n \to \{0, 1\}$ für $j = 0 \ldots k$ (Moore)
$\lambda = (\lambda_j)$ mit $\lambda_j : 2^{n+m} \to \{0, 1\}$ für $j = 0 \ldots k$ (Moore)

Komposition von Maschinen

Sequentielle Komposition von Moore-Maschine

Sequentielle Komposition von Moore-Maschinen erzeugt Verzögerung; das berechnete Ergebnis wird erst im nächsten Takt weitergegeben. Dadurch ist aber ein viel höherer Takt möglich.

Sequentielle Komposition von Mealy-Maschine

Sequentielle Komposition von Mealy-Maschinen erzeugt lange Signalpfade; längste Pfade antiproportional zu erzielbaren Taktfrequenz. In der Zwischenzeit treten Hazards auf, da Eingabe Signal durch alle in Serie geschalteten Gatter gehen muss, dadurch vergeht Zeit, bis stabil.

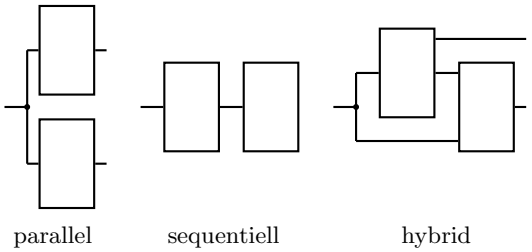


Abbildung 6.12: *Komposition von Maschinen*

Mealy-Maschine als Moore-Maschine

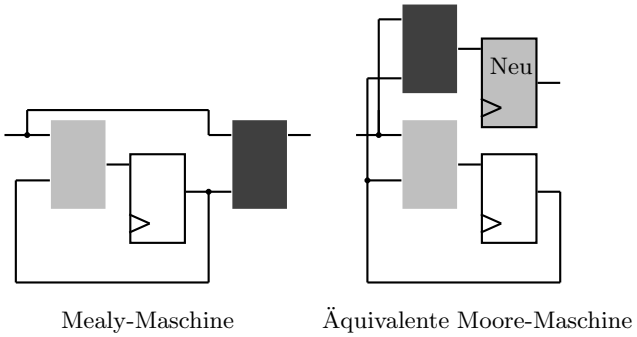


Abbildung 6.13: *Mealy-Maschine als Moore-Maschine*

Moore-Maschine ist um einen Takt zeitverzögert.

7 Algorithmic State Machines

7.1 ASM Elemente

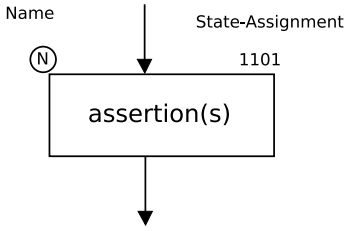


Abbildung 7.14: *Zustand mit Ausgaben*

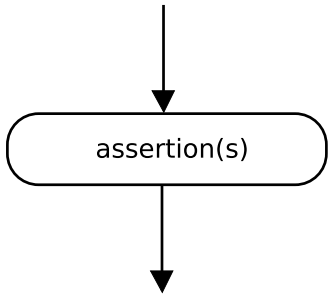


Abbildung 7.15: *Ausgabe gehört zum vorherigen Zustand*

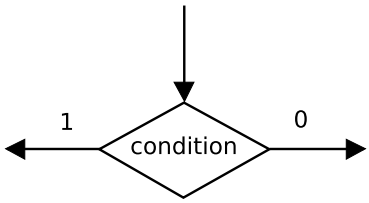


Abbildung 7.16: *Fallunterscheidung*

8 Addierer

8.1 Halb-Addierer (Half Adder)

Addition von zwei Bits mit Übertrag-Generierung

$$\begin{aligned} s &\equiv (a \boxed{+} b) \mod 2 \equiv a \oplus b \\ o &\equiv (a \boxed{+} b) \operatorname{div} 2 \equiv a \cdot b \end{aligned}$$

$\boxed{+}$ bezeichnet die Addition in den natürlichen Zahlen

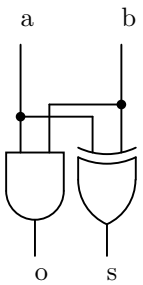


Abbildung 8.17: *Halb-Addierer*

8.2 Voll-Addierer (Full Adder)

Addition von drei Bits mit Übertrag-Generierung

$$\begin{aligned} s &\equiv (a \boxed{+} b \boxed{+} i) \mod 2 \equiv a \oplus b \oplus i \\ o &\equiv (a \boxed{+} b \boxed{+} i) \operatorname{div} 2 \equiv a \cdot b + a \cdot i + b \cdot i \end{aligned}$$

i = carry **i**n
o = carry **o**ut
s = **s**um

Voll-Addierer Whitebox Modell

$$\begin{array}{cccccccc} o & = & (a \cdot b) \oplus ((a \oplus b) \cdot i) & \equiv & (a \cdot b) + ((a \oplus b) \cdot i) & \equiv & ((a+b) \cdot i) + (a \cdot b) \\ s & = & (a \oplus b) \oplus c & \equiv & (a \oplus b) \oplus c & \equiv & (a \oplus b) \oplus c \end{array}$$

8.3 Ripple-Carry Adder

Für n Bits werden n Full Adder in Serie geschaltet, das carry out Bit wird immer an den carry in des nachfolgenden Voll-Addierer weitergegeben.

Platz: $O(n)$
Zeit: $O(n)$

8.4 Kombinierter Addierer Subtrahierer

A, B, S in 2er-Komplement.

Eingänge wie bei Ripple-Carry Adder und zusätzlich ein Eingang, ob Addition oder Subtraktion, falls Subtraktion dann wird jeweils durch einen Mux das B_i invertiert und zu A_i addiert und als S_i ausgegeben. Es kann noch ein XOR des S_n mit dem Carry Out o_n gemacht werden, falls 1, dann trat ein Overflow auf.

8.5 Addierer in VHDL

Addierer Entity

```
library ieee;

use ieee.std_logic_1164.all;

entity adder is
  generic (n: positive);
  port (a,b : in std_ulogic_vector (n-1 downto 0);
        i   : in std_ulogic;
        o   : out std_ulogic;
        s   : out std_ulogic_vector (n-1 downto 0));
end entity adder;
```

Halb-Addierer in VHDL

```
library ieee;
use ieee.std_ulogic_1164.all;

entity half_adder is
  port (a, b: in std_ulogic; o, s: out std_ulogic);
end entity half_adder;

architecture expr of half_adder is
begin
  s <= a xor b;
  o <= a and b;
end architecture expr;
```

Voll-Addierer in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
  port (a, b, i: in std_ulogic;
        o, s   : out std_ulogic);
end entity full_adder;

architecture with_half_adders of full_adder is
  component half_adder
    port(a,b : in std_ulogic;
         o,s : out std_ulogic);
  end component half_adder;
  signal o1, s1, o2 : std_ulogic;
begin
  ha1: half_adder port map (a, b, o1, s1);
  ha2: half_adder port map (s1, i, o2, s);
  ha3: half_adder port map (o1, o2, open, o);
end architecture with_half_adders;
```

Ripple-Carry-Addierer in VHDL

```
architecture ripple_carry of adder is
  component full_adder is
    port (a, b, i : in std_ulogic;
          o, s    : out std_ulogic);
  end component full_adder;
  signal c: std_ulogic_vector (n downto 0);
begin
  g1 : for i in 0 to n-1 generate
    fa : full_adder port map
     (a(i), b(i), c(i), c(i+1), s(i));
  end generate g1;
  c(0) <= i;
  o <= c(n);
end architecture ripple_carry;
```

8.6 Carry Lookahead

Idee: Vorausberechnung des Carry, Faktorisieren von Carry-Propagierung und Generierung.

Faktorisieren

$$c_{k+1} \equiv a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \equiv \underbrace{a_k \cdot b_k}_{g_k} + \underbrace{(a_k + b_k)}_{p_k} \cdot c_k$$

g_k: die Summanden-Bits an der k-ten Stelle **g**enerieren ein Carry
p_k: die Summanden-Bits an der k-ten Stelle **p**ropagieren das Carry

$$\begin{aligned} c_k &\equiv g_{k-1} + p_{k-1} \cdot g_{k-2} + \cdots + (p_{k-1} \cdot p_{k-2} \cdots p_0 \cdot c_0) \\ s_k &\equiv a_k \oplus b_k \oplus (g_{k-1} + p_{k-1} \cdot g_{k-2} + \cdots + (p_{k-1} \cdot p_{k-2} \cdots p_0 \cdot c_0)) \end{aligned}$$

pg Element

$$\begin{aligned} g_k &\equiv a_k \cdot b_k \\ p_k &\equiv a_k + b_k \end{aligned}$$

$$c_{k+1} \equiv g_k + p_k \cdot c_k$$

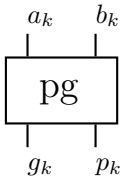


Abbildung 8.18: *pg Element*

Schlussbemerkung

Dieser Ansatz benötigt aber für das letzte ODER n Eingänge, und insgesamt $O(n^2)$ Transistoren, dafür ist aber die Zeit $O(1)$, es gibt eine 4-stufige Logik, welche unabhängig von n ist.

8.7 Optimierter Carry Lookahead Addierer

PG Element

$G_{k,i}$: $[k,i]$ generiert Carry
$P_{k,i}$: $[k,i]$ propagiert Carry

$$\begin{aligned} G_{k,i} &\equiv G_{k,j+1} + P_{k,j+1} \cdot G_{j,i} \\ P_{k,i} &\equiv P_{k,j+1} \cdot P_{j,i} \end{aligned}$$

$$c_{k+1} \equiv G_{k,i} + P_{k,i} \cdot c_i$$

Bemerkungen:
$k-1 > j > i$, $G_{ii} = g_i$, $P_{ii} = p_{ii}$

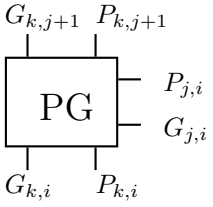


Abbildung 8.19: *PG Element*

Summe und Carry Element

$$\begin{aligned} s_k &\equiv a_k \oplus b_k \oplus c_k \\ c_{k+1} &\equiv G_{k,i} + P_{k,i} \cdot c_i \end{aligned}$$

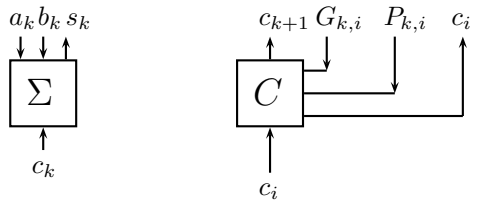


Abbildung 8.20: *Summe und Carry Element*

Zusammenfassung

- 1. Phase: Generierung und Propagierung (down)
- 2. Phase: Carry Erzeugen (up)
- 3. Phase: lokale Summenbildung
- Baum wird zweimal traversiert
- Zeit: $O(\log n)$, Platz: $O(n)$
- Bei ausschliesslicher Verwendung von Gattern mit zwei Eingängen

9 Multiplizierer

9.1 Sequentieller Algorithmus

4 Register: C, P, A, B

1. Obere Hälfte P der partiellen Summe mit 0 initialisieren.
2. Erster Operand ins B Register
3. Zweiter Operand ins A Register (untere Hälfte der partiellen Summe)
4. Für jeden der n Multiplikationsschritte:
   a) LSB von A gleich 1, dann Addiere B zu P (ansonsten 0)
   b) Schiebe (C, P, A) nach rechts (C ist Carry des Addierers)
5. Resultat findet sich in (P, A)

9.2 Probleme mit Vorzeichen

Einfache Multiplikation mit Vorzeichen

1. Konvertierung der beiden Operanden in positive Zahlen
2. Speichern der ursprünglichen Vorzeichen
3. Unsigned Multiplikation der Konvertierten Zahlen
4. Berechnung des Resultat-Vorzeichens aus gespeicherten Vorzeichen (negativ gdw. ursprüngliche Operanden komplementäres Vorzeichen hatten)
5. eventuell Negation des Ergebnisses bei negativem Resultats-Vorzeichen

Booth Recoding

- Verwende Arithmetisches Shift statt Logischem, somit wird jedes Mal, wenn das signed Bit gesetzt ist auch ein 1 hineingeschoben.
- Kein Carry Bit beim MSB, falls Carry, so vergesse ihn einfach.
- Verwende folgende Addition/Subtraktion Regeln ($A_{-1} = 0$)
  1. addiere 0 zu P wenn $A_i = 0$ und $A_{i-1} = 0$
  2. addiere B zu P wenn $A_i = 0$ und $A_{i-1} = 1$
  3. subtrahiere B von P wenn $A_i = 1$ und $A_{i-1} = 0$
  4. addiere 0 zu P wenn $A_i = 1$ und $A_{i-1} = 1$

Hintergrund Booth Multiplikation

Da jedesmal

$$B \cdot (A_{i-1} - A_i)$$

zum partiellen Produkt addiert wird, erhält man die Teleskopsumme

$$\sum_{i=0}^{n-1} B \cdot (A_{i-1} - A_i) \cdot 2^i \equiv$$

$$B \cdot (-A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \cdot + A_1 2 + A_0) + B \cdot A_{-1}$$

9.3 Carry-Save-Adder (CSA)

- Idee: keine Carry Propagierung bei der Multiplikation
- Unabhängige Volladdierer (Full Adder = FA)
- Carry-In wird von vorheriger Berechnung genommen
- Carry-OUt wird gespeichert für nachfolgende Berechnung
- Man spart Zeit für die Propagation
- Nach n Schritten muss noch die Summe und die Carries addiert werden

10 Sequentielle Blocks

Während Flip-Flops flankengesteurte Speicher-Elemente sind, d.h. sie lesen Eingabe, wenn Flanke nach oben, sind Latches pegelgesteuerte Speicher-Elemente, d.h. sie lesen Eingabe solange clock=1.

10.1 RS Latch mit NANDs

S	R	Q'	$\overline{Q}'$
0	0	1	1
0	1	0	1
1	0	1	0
1	1	Q	$\overline{Q}$

Erste Zeile ist unerwünscht, da Q' und $\overline{Q}'$ nicht komplementär.

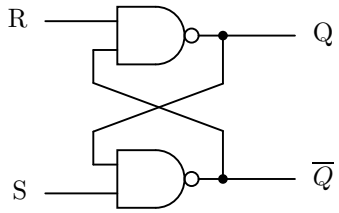


Abbildung 10.21: *RS Latch*

RS Latch in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity rslatch1 is
  port (S, R : in std_ulogic;
        Q, Qbar : buffer std_ulogic);
end entity rslatch1;

architecture dataflow of rslatch1 is
begin
  Q <= '1' when R = '0' else
       '0' when S = '0' else Q;
  Qbar <= '1' when S = '0' else
          '0' when R = '0' else Qbar;
end architecture dataflow;

use ieee.std_logic_1164.all;

entity rslatch2 is
  port (S, R : in std_ulogic;
        Q, Qbar : out std_ulogic);
end entity rslatch2;

architecture dataflow of rslatch2 is
begin
  Q <= '1' when R = '0' else
       '0' when S = '0' else unaffected;
  Qbar <= '1' when S = '0' else
          '0' when R = '0' else unaffected;
end architecture dataflow;
```

10.2 D Latch

D wird zu Q weitergeschaltet, wenn Enable gesetzt ist. Geht Enable zurück, dann behält Q seinen Wert bei. Kontrollsignal C ist durch 1 verknüpft mit der Eingabe D.

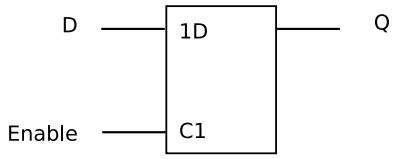


Abbildung 10.22: *D Latch*

```
library ieee;
use ieee.std_logic_1164.all;

entity dlatch is
  port (D, Enable : in std_ulogic;
        Q : out std_ulogic);
end entity dlatch;

architecture behavioural of dlatch is
begin
  process (D, Enable) is
  begin
    if (Enable == '1') then
      Q <= D;
    end if;
  end process;
end architecture behavioural;
```

10.3 D Flip-Flop mit Asynchronem Set/Reset

Reset (Set) auf 0 setzt Q auf 0 (1). Die Zahl 1 veknüpft die Clock mit der Eingabe, R und S sind unabhängig von C (keine 1) also asynchron.

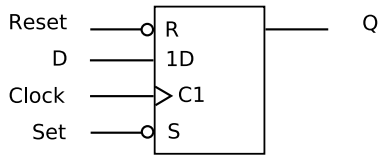


Abbildung 10.23: *D Flip-Flop mit Asynchronem Set/Reset*

```
library ieee;
use ieee.std_logic_1164.all;

entity dffr is
  port (D, Clock, Reset : in std_ulogic;
        Q : out std_ulogic);
end entity dffr;

architecture behaviourial of dffr is
begin
  process (Clock, Reset) is
  begin
    if (Set = '0') then
      Q <= '1';
    elsif (Reset = '0') then
      Q <= '0';
    elsif (rising_edge(Clock)) then
      Q <= D;
    end if;
  end process;
end architecture behavioural
```

10.4 T Flip-Flop

T	Q'	$\overline{Q}'$
0	Q	$\overline{Q}$
1	$\overline{Q}$	Q

Zustand ändert sich nicht bei $T = 0$, sonst wechselt (toggled) der Zustand (0 zu 1, 1 zu 0)

10.5 JK Flip-Flop

J	K	Q'	$\overline{Q}'$
0	0	Q	$\overline{Q}$
0	1	0	1
1	0	1	0
1	1	$\overline{Q}$	Q

ähnlich RS Flip-Flop, aber keine problemtische Eingangs-Belegungen, die 1/1 Kombination toggled den Zustand, die 0/0 Kombination ändert nichts.

10.6 Register

4-Bit Register IEEE Schaltbild

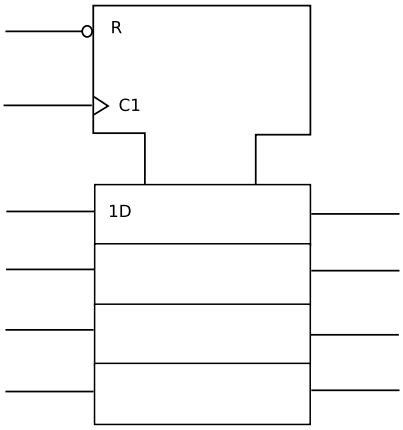


Abbildung 10.24: *4 Bit Register IEEE Schaltbild*

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
  generic (n : natural := 4);
  port (D: in std_ulogic_vector(n-1 downto 0);
        Clock, Reset : in std_ulogic;
        Q: out std_ulogic_vector(n-1 downto 0));
end entity reg;

architecture behavioural of reg is
begin
  process (Clock, Reset) is
  begin
    if (Reset = '0') then
      Q <= (others => '0');
    elsif rising_edge(Clock) then
      Q <= D;
    end if;
  end process;
end architecture behavioural;
```

10.7 Serial In Parallel Out (SIPO) Register

Bit-Sequenz wird Bit für Bit eingelesen, genügend Bits gesammelt: als Wort (z.B. 32 Bit) auslesen. Dazu wird ein Schieberegister verwendet. Gelesene Bits werden um eine Position nach vorne verschoben und neue Bits werden hinten angehängt.

SIPO VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity sipo is
  generic (n: natural := 8);
  port (a: in std_ulogic;
        q: out std_ulogic_vector(n-1 downto 0);
clk: in std_ulogic);
end entity sipo;

architecture rtl of sipo is
begin
  process (clk) is
    variable reg : std_ulogic_vector(n-1 downto 0);
  begin
    if rising_edge(clk) then
      reg := reg(n-2 downto 0) & a;
      q <= reg;
    end if;
  end process;
end architecture rtl;
```

10.8 Universal Shift Register

s(1)	s(0)	Action
0	0	Hold
0	1	Shift right
1	0	Shift left
1	1	Par. Load

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity universal_shift_register is
  generic (n : natural := 8);
  port (a: in std_ulogic_vector(n-1 downto 0);
        lin, rin : in std_ulogic;
        s : in std_ulogic_vector(1 downto 0);
        clk, rst: in std_ulogic;
        q : out std_ulogic_vector(n-1 downto 0));
end entity universal_shift_register;
```

```
architecture rtl of universal_shift_register is
begin
  process (clk, rst) is
    variable reg : std_ulogic_vector(n-1 downto 0);
  begin
    if (rst = '0') then
      reg := (others => '0');
    elsif rising_edge(clk) then
      case s is
        when "11" =>
          reg := a;
        when "10" =>
          reg := reg(n-2 downto 0) & lin;
        when "01" =>
          reg := rin & reg(n-1 downto 1);
        when others =>
          null;
      end case;
    end if;
    q <= reg;
  end process;
end architecture rtl;
```

10.9 Zähler in RTL

```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

entity counter is

end entity counter;

architecture rtl of counter is
begin
  process (clk, rst) is
    variable cnt: unsigned (n-1 downto 0);
  begin
    if rst = '0' then
      cnt := (others => '0');
    elsif rising_edge(clk) then
      cnt := cnt -1;
    end if;
    count <= std_ulogic_vector(cnt);
  end process;
end architecture rtl;
```

10.10 Ripple Zähler mit T-Flip-Flops

Ausgang der T-Flip-Flops steuern Takt folgender T-Flip-Flops. Lineare Verzögerung in der Wortbreite bis MSB stabil.

11 Speicher

ROM read-only memory

RAM random-access memory

SRAM static RAM. Zustand bleibt solange Strom anliegt.

DRAM dynamic RAM. implementiert mit Kondensatoren. Zustand verliert sich ohne regelmässiges Auffrischen.

11.1 SRAM

Ist schreib- und lesbar, also bidirektionaler `inout` Daten-Port. Kontrollsignale CS (Chip Select), WE (Write Enable), OE (Output Enable). CS aktiviert einen einzelnen RAM-Chip. Deaktiviert sind Ausgaben eines RAM-Chip im Tristate (`Z`). Schreiben durch Setzen von $\overline{WE}$, Lesen durch Setzen von $\overline{OE}$. Interface Constraint: $\overline{WE}$ und $\overline{OE}$ sind nie gleichzeitig gesetzt.

Schreibzyklus

$\overline{WE}$ muss schon gewisse Zeit gesetzt sein, bis mit $\overline{CS}$ Chip gewählt werden kann.

Lesezyklus

Auch hier muss $\overline{OE}$ schon gewisse Zeit gesetzt sein, bis mit $\overline{CS}$ Chip gewählt werden kann. Zusätzlich ist Data erst nach einer gewissen Zeit verfügbar.

SRAM Zelle mit zwei Invertern

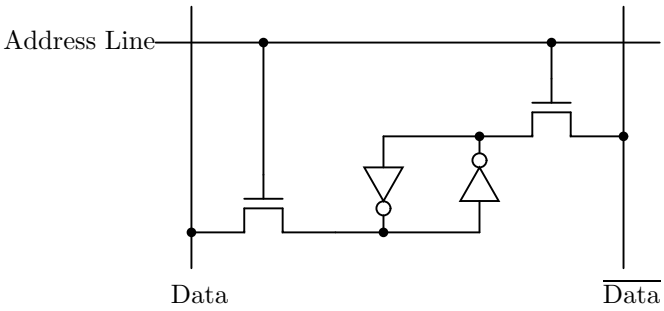


Abbildung 11.25: *SRAM Zelle mit zwei Invertern*

Kaskadierung von SRAM Chips

Mehrere SRAM Chips können kombiniert werden, indem man jeweils die hintersten Bits für die Adressierung der Chips benutzt, also z.Bsp 2 Bits für 4 Chips und die restlichen Adressbits für die Adressierung innerhalb des Chips.

11.2 DRAM

Verwendet eine Kapazität antstatt Transitstor Gatter. Es benötigt immer einen Refresh, da die Daten nach gewisser Zeit verschwinden.

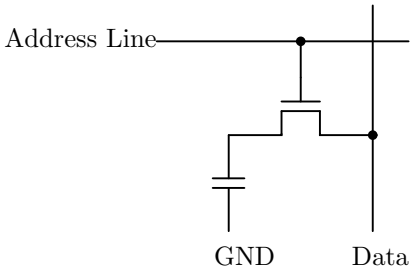


Abbildung 11.26: *DRAM Zelle mit Kapazität*

12 Simulation

Im Normalfall wir Stimulus und ein Checker verwendet. Unter *Funktionale Tests* versteht man korrekte funktionale Implementierung, man testet als ob Eingabe/Ausgabe der Spezifikation entspricht. Unter *Timing Simulation* versteht man korrektes Zeitverhalten, also ob die Zeitschranken eingehalten werden, hier kann man möglicherweise den funktionalen Stimulus wiederverwenden.
Oftmals benutzt man für das *Design under Test (DUT)* RTL (Spezifikation) und Strukturelle (DUT = Eigentliches Testobjekt) "Implementierung" und vergleiche diese gegeneinander. Also im Allgemeinen wird Spezifikation und Implementierung miteinander verglichen.

In VHDL vergleicht man die Spezifikation gegen die Implementierung oft mittels einem Prozess, der mit `assert` arbeitet.

```
wait for 50 ns;
assert sr = sd report "outputs differ" severity failure
```

12.1 Datei-Operationen

```
library ieee;
use ieee.std_logic_1164.all, std.textio.all;
entity neg4bit is
end entity neg4bit;
architecture fileio of neg4bit is
  file input_file, output_file : text;
begin
  process
    variable input_line, output_line : Line;
    variable v : bit_vector(3 downto 0);
  begin
    file_open(input_file, "input.txt", read_mode);
    file_open(output_file, "output.txt", write_mode);
    while not(endfile(input_file)) loop
      readline(input_file, input_line);
      read(input_line, v);
      for i in 0 to 3 loop v(i) := not v(i); end loop;
      write(output_line, v);
      writeline(output_file, output_line);
    end loop;
    file_close(input_file); file_close(output_file);
    wait;
  end process;
end architecture fileio;
```

12.2 Event-Driven Simulation

Event: Signal plus Wert plus Zeitpunkt

Es wird eine Event-Queue mit Delta-Delays benutzt um das Ganze zu simulieren, dabei muss man die Delta-Delays beachten, das sind Verzögerungen zwischen Events, die eigentlich zur gleichen Zeit eintreten.

Single-Pass Even

```
for (i = each event at current_time) {
  update_node (i);
  for (j = each gate on the fanout list of i) {
    update input values of j;
    evaluate (j);
    if (new_value (j) != last_scheduled_value (j)) {
      schedule new_value (j) at (current_time + delay (j));
      last_scheduled_value (j) = new_value (j);
    }
  }
}
```

Delta-Delays

Evaluation eines VHDL-Ausdruckes kostet ein Delta, also wird so z.B. ein Signal, welches durch zwei Inverter geht um ein Delta nachher in Event-Queue behandelt, als eines, welches nur durch einen Inverter geht.

13 Test

Test: häufig eingeschränkt auf Testen von Chips auf Fabrikationsfehler. Unter einem *Funktionalen Test* versteht man das Testen aller Eingabe-Kombinationen. Bei einem *Strukturellen Test* werden alle potentiellen Stuck-at-Faults getestet, unter einem *Stuck-at-fault* versteht man ein Leiter, der immer mit logisch 1 oder 0 verbunden ist.

13.1 ATPG

Automatic Test Pattern Generation (ATPG).

Algorithmus zur Generierung aller Test-Vektoren

1. Generiere Liste von Test-Vektoren für alle potentiellen Stuck-at-Faults (Stuck-at-0 und Stuck-at-1 für alle Signale). Also bei n Gattern $2n$ Testvektoren.
2. Wähle ungebrauchten Test-Vektor und simuliere ihn (für den Fault-Free und den Faulty-Circuit).
3. streiche von der Liste alle zusätzlich abgedeckten Faults (ein Testvektor kann mehrere Faults testen).
4. Coverage definiert als Anteil abgedeckter Faults.
5. Falls Coverage nicht gut genug und Zeit nicht abgelaufen gehe zu 2.

Test-Vektor-Generierung

Controllability: Gibt es ein Eingabe-Vektor, mit $S = \overline{X}$ im Fault-Free Circuit? Suche nach partiellen Eingabe-Vektor im Input-Cone.

Observability: Kann $S = \overline{X}$ im Fault-Free Circuit zu einem Ausgang propagiert werden? Kann $S = X$ im Faulty-Free Circuit zu demselben Ausgang propagiert werden? Suche nach Propagierungs-Pfad im Output-Cone. Miteinbeziehung von Input-Cones der Signale auf dem Propagierungs-Pfad.

Fault Equivalence

Zwei Faults heissen äquivalent gdw. sie die gleichen Tests haben. Also z.B. 0 an Ausgang von AND sowie an einem der Eingänge.

D-Algorithmus

- Repräsentiere Differenz zwischen Fault-Free und Faulty Circuit symbolisch
- Neuer Wahrheitswert "D": Signal 1 im Fault-Free und 0 im Faulty Circuit
- Negation "$\overline{D}$": Signal 0 im Fault-Free und 1 im Faulty Circuit
- Neues Propagierungsziel: D oder $\overline{D}$ muss am Ausgang erscheinen!
- Suche mit Backtracking wie zuvor
- Falls man D od. $\overline{D}$ am Ausgang nicht bekommt, so ist dieser Stuck nicht testbar.

AND	0	1	D	$\overline{D}$	X
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	X
D	0	D	D	0	X
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
X	0	X	X	X	X

Scan-Chain Design

Das ganze Design der Schaltung wird am Ende so modifiziert, dass alle Flip Flops linear angeordnet sind und die eigentliche kombinatorische Schaltung davon getrennt ist. Die FF haben jeweils einen Multiplexer vorgeschaltet.

14 Synthese vs. Simulation

Wenn man von der Architektur nach Gattern synthetisiert, so nennt man dies Behavioural Synthese, wenn es von RTL nach Gatter geschieht, RTL Synthese.

14.1 Abgeleitete Latches

Abdeckung aller Fälle bei einem `process` bei den if Abfragen erzeugt kombinatorische Logik und umgekehrt.

14.2 Formale(re) Analyse von Asynchronen Schaltkreisen

Fundamental Mode
Annahme 1: immer nur ein Eingang ändert sich
Annahme 2: Änderung eines Eingangs erst nach Stabilisierung

Idee: füge bei Feedback-Loops einen *virtuellen* Buffer ein. Weitere Annahme: gesamte Delays nur noch im Buffer, andere Gates haben also Zero-Delay.

15 Repräsentation kombinatorischer Logik / SAG

Verschiedene Möglichkeiten um Boolsche Logik darzustellen: Funktionstabellen, welche kanonisch (= eindeutig) sind aber immer 2^n Platz benötigen, oder z.B. DNF.

15.1 Syntaktische Darstellung

Man kann z.B. ein Parse-DAG (Directed Acylic Graph) für kombinatorische Logik erstellen. Dafür benutzt man Knoten mit einem Typ (VAR, XOR, OR, AND, ...) und 2 Pointer zu Argumenten (ausser bei VAR). Somit kann man das ganze als DAG kompakt darstellen.

15.2 Signed-And-Graphs (SAG)

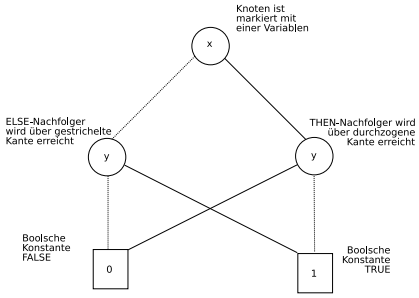
Logische Basis-Operationen: Konjunktion und Negation. DAG-Darstellung: Operations-Knoten sind alles Konjunktionen; Negation/Vorzeichen als Kanten-Attribut (daher *signed*) (kann man platzsparend als LSB im Pointer implementieren)

Syntaktisches Sharen

Invariante: zwei Knoten haben immer unterschiedliche Kinder. Wenn ein neuer Knoten erzeugt wird, wird zuerst nach äquivalentem Knoten gesucht und falls einer vorhanden, so wird dieser benutzt. Doch die SAGs bleiben auch so nicht-kanonisch.

16 Binary Decision Diagrams

Neue dreistellige Basis-Operation ITE (if-then-else), wobei die Bedingung immer eine Variable ist. Es werden meistens ROBDDs verwendet welche bewiesenermasen kononisch sind.



16.1 Semantik

Innere Knoten sind ITE, Blätter sind boolsche Konstanten. Schreibweise $ite(x, f_1, f_0)$ bedeutet, wenn x dann f_1 ansonsten f_0. *eval* produziert booleschen Ausdruck aus einem BDD:

$$eval(0) \equiv 0 \quad (1)$$
$$eval(1) \equiv 1 \quad (2)$$
$$eval(ite(x, f_1, f_0)) \equiv x \cdot eval(f_1) \vee \overline{x} \cdot eval(f_0) \quad (3)$$

16.2 Reduced Ordered BDDs

Es werden folgende Bedingungen an die BDDs gestellt:

- Maximale algebraische und semantische Reduktion
- Variablen von der Wurzel zu den Blättern sind gleich geordnet

Algebraische Reduktions-Regel

Maximales Sharen isomorpher Teil-Graphen. Wenn 2 Knoten die gleichen Kinder haben und auch dieselbe Variable beinhalten so kann man sie zusammennehmen.

Semantische Reduktions-Regel

Knoten, welche für if und else den gleichen Nachfolger haben, können entfernt werden.

16.3 Tautologie, Erfüllbarkeit, Äquivalenz

- ein BDD ist eine Tautologie, gdw. er nur aus dem 1-Blatt besteht
- ein BDD ist erfüllbar, gdw. er nicht aus dem 0-Blatt besteht
- zwei BDDs sind äquivalent, gdw. die BDDs isomorph sind

16.4 Shannonsche Expansionsregel

$$f(x) \equiv x \cdot f(1) \vee \overline{x} f(0)$$

Sei nun x die oberste Variable zweier BDDs f und g:

$$f \equiv ite(x, f_1, f_0) \qquad g \equiv ite(x, g_1, g_0)$$

mit f_i bzw. g_i die Kinder von f und g für $i = 0, 1$. Dann folgt

$$f(0) = 0 \quad g(0) = g_0 \quad f(1) = f_1 \quad g(1) = g_1$$

da nach dem R in ROBDD x nur ganz oben in f und g vorkommt.

$$\begin{aligned}
(f@g)(x) &\equiv x \cdot (f@g)(1) \vee \overline{x}(f@g)(0) && (4)\\
&\equiv x \cdot (f(1)@g(1)) \vee \overline{x} \cdot (f(0)@g(0)) && (5)\\
&\equiv x \cdot (f_1@g_1) \vee \overline{x}(f_0@g_0) && (6)
\end{aligned}$$

Weobi @ eine beliebige zweistellige boolsche Operation ist.