

Elektronik und Digitaltechnik – Digital (EDT)

Schule: HTBLuVA St. Pölten
Abteilung / Zweig: Elektronik / Technische Informatik
Lehrperson: Dipl.-Ing. Gerald Gruber
Jahrgang: 2005 / 06
Klasse: 4AHELI

1 Anmerkung

| Praktische Beispiele sind mit einem Strich auf der Seite gekennzeichnet.
Programmteile sind in dieser Schrift geschrieben.

Prof. Gruber stellt für jedes Kapitel ein eigenes Skriptum zur Verfügung, welches allerdings nur der Übersicht dient.

Da Prof. Gruber schnell spricht, sind die Formulierungen eher schlecht, das Skriptum dafür aber sehr ausführlich.

Notenschlüssel:

Mitarbeiten:	Tests:
+	1
+o	2
o	3
-o	4
-	5
	5- (<= 25%)
	5! (<= 12%)

2 Inhaltsverzeichnis

1	Anmerkung.....	2
2	Inhaltsverzeichnis.....	2
3	Protel	5
3.1	Schematic Bibliothek	5
3.2	PINs.....	5
3.3	ERC (Electrical Rule Check)	7
3.4	Footprint Bibliothek	7
3.5	Layer.....	8
3.6	Bohrplan.....	8
3.7	CAM.....	8
3.8	Masse Flächen.....	9
3.9	weitere Einstellungen.....	9
3.10	Vias – PINs.....	10
3.11	internal Layers.....	10
3.12	Bus.....	10
4	AT2051.....	10
4.1	DATA Bereich	11
4.1.1	BDATA	11
4.2	Stackpointer.....	12
4.3	SFR.....	12
4.4	IDATA	12
4.5	Zahlensysteme in Assembler.....	12
4.6	Codespeicher	13
4.6.1	Assembler Programmierung.....	13
4.7	Externes RAM.....	14

4.8	Assemblerprogrammierung beim 2051	14
4.8.1	Sprung und Unterprogramm.....	14
4.8.2	Operationen	16
4.8.3	Weitere Befehle.....	17
4.8.4	Rotierbefehle	18
4.8.5	Fehlersuche.....	18
4.8.6	Pre Prozessor Befehle.....	18
4.8.7	Definitionen.....	19
4.8.8	Tabellen.....	19
4.9	Registermodell	19
4.9.1	CPU	20
4.9.2	Timer	21
4.10	RS232 (Serielle Schnittstelle)	23
4.10.1	SBUF Register.....	24
4.10.2	Framing	25
4.10.3	Software Timer.....	26
4.11	Handshake	28
4.11.1	Hardware Handshake	28
4.11.2	Software Handshake.....	29
4.12	Tipps zur A51 Programmierung.....	30
5	LPT Centronics (Parallele Schnittstelle)	32
6	Sequencer	34
6.1	ROM Sequencer	35
6.2	JK-Sequencer	36
6.3	Software Sequencer.....	40
7	EasyABEL (PLD Software von DATAIO)	42
7.1	Programm Architektur.....	42
7.2	Aufbau einer ABEL-Datei	43
7.3	Beispiel einer ABEL-Datei	44
7.4	Die ABEL HDL Sprache.....	44
7.5	Beispiel für ein Schrittschaltwerk	47
7.6	Sequencer Beispiel	48
7.7	Beispiel 4 Bit Multiplikator.....	49
7.8	Tristate.....	50
7.9	Wired Logic.....	50
8	Kundenprogrammierbare Logikbausteine	51
8.1	Sicherungen.....	51
8.2	Software	52
8.3	ROM (Read Only Memory)	52
8.4	PAL (Programmable Array Logic)	52
8.5	PLD (Programmable Logic Device)	53
8.6	GAL (Generic Array Logic).....	53
8.7	CPLD (Complex Programmable Logic Device)	55
8.7.1	GLB (Generic Logic Block).....	57
8.8	FPGA (Field Programmable Gate Array)	58
8.8.1	Ein- und Ausgangszelle.....	60
8.8.2	CLB (Complex Logic Block).....	61
8.8.3	Unterschied synchrone / asynchrone Logik	62
8.9	ASIC (Application Specific Integrated Circuit)	63
8.9.1	Sea of Gates.....	64
8.9.2	Standardzelle	65

8.9.3	Makro-Zellen	66
8.9.4	Full Custom	66
8.9.5	Entwurfsregeln	67
8.10	Testbarkeit	68
8.10.1	Nadeladapter (große Stückzahl) U,I,R	68
8.10.2	X-Y-Plotter	68
8.10.3	Boundary – Scan – Interface	69
8.10.4	Multiplexer	70
8.10.5	BIST (Burn in self test)	70
8.10.6	Stromaufnahme	70
8.11	Ausfallsraten	71
8.12	Effekte beim ASIC-Design	72
8.12.1	Signal Delay	72
8.12.2	Jitter	73
8.12.3	Noise	73
8.12.4	Latchup-Effekt	73
8.12.5	Übersprechen	73
8.12.6	Leakage (Leckstrom)	74
8.12.7	Stress	75
8.13	Kosten	76
9	VHDL	77
9.1	VHDL Arbeitsumgebung	78
9.2	Allgemeines	79
9.3	Aufbau eines VHDL Textes	80
9.4	Datentypen und Objekte	81
9.5	Kombinatorische Logik in VHDL	83
9.6	Sequentielle Logik in VHDL	84
9.7	Schrittschaltwerke in VHDL	86
9.8	VHDL Testbench	87

3 Protel

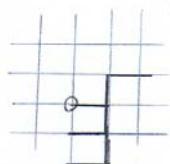
Protel ist ein Programm zum entwickeln von Printlayouts.

3.1 Schematic Bibliothek

*.sch	Schematic
*.lib	library
*.slb	symbol library (in industrie gern so genannt)

In einer Bibliothek sind mehrere Bauteile enthalten. Alle Bauteile eines Projekts werden in einer Bibliothek zusammengefasst. In der Industrie werden in Bibliotheken ähnliche Bauteile zusammengefasst (μ C, Dioden, Widerstände,...)

3.2 PINs



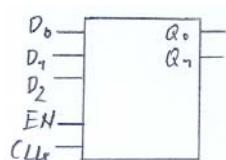
Bauteile haben PINs, bei diesen kann man Drähte zu anderen Bauteilen anschließen. Außerdem müssen Ein- und Ausgänge definiert werden. (Es hat z.B. bei logischen Schaltungen keinen Sinn, Ausgänge parallel zu schalten.) Außerdem gibt es Power Pins, diese haben den Vorteil, dass PINs mit gleichem Namen automatisch verbunden werden. (z.B.: V_{CC}) – auf die gleichen Namen besondere Acht legen. Passive PINs , bezeichnen zum Beispiel mechanische Weiterleitung (Steckverbinder).

Es gibt folgende PINs:

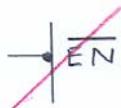
- IN
- OUT
- I/O
- Power Pins
- Passive Pins

PINs können „hidden“ gemacht werden, dann sind sie nachher nicht mehr sichtbar (sehr nützlich bei Spannungsversorgung).

Normalerweise sind bei Bauteilen links die Eingänge und rechts die Ausgänge. Die Zählung von Ein- und Ausgängen beginnt mit 0, also Q₀, Q₁, Q₂, etc. oder A₀, A₁, ... Ähnliche PINs sind visuell zusammenzufassen.



Invertierte Ausgänge werden mit DOTs oder Querstrich gekennzeichnet, nicht mit beidem.



Ein PIN hat einen Namen und eine Nummer:



Widerstände brauchen weder Name noch Nummer. Diese sind unabhängig von einander anzeigbar.

Die Pinnummern beginnen immer mit 1, nicht mit 0! (Die Namen beginnen sehr wohl mit 0) In Protel gibt es Elemente, die weiter vorne oder hinten liegen können. Sollte ein Name also hinter einer gelben Fläche verschwinden, muss man die betreffenden Attribute ändern.

In Protel gibt es so genannte „Parts“. Damit kann man zusammengehörige Symbole splitten und getrennt von einander verschieben (in einem UND-IC sind ja 4 UND-Gatter drin).

Es ist wichtig, dass beim Zeichnen alle Parts korrekt übereinander liegen.

Außerdem muss man den Ursprung (Origin) des Bauteils auf 0/0 liegen. Auf diesem Punkt wird der Bauteil verschoben, gedreht und gespiegelt.

Den Knopf „Add“ beim erstellen von Parts nicht drücken. (neuer Bauteil mit gleicher Bild!)

Beim Erstellen von Parts müssen bei allen Parts Versorgungspins gesetzt werden.

Der Name des Bauteils wird automatisch mit eingeblendet. Der „default designator“ muss handische eingegeben werden (IC1, IC2). Beim Erstellen sollte man sie IC?, R?, etc. nennen, denn so gibt es die Möglichkeit, die Nummern automatisch zu vergeben.

Beim einstellen des „default designators“ muss man auch die Footprints einstellen (bis zu 4).

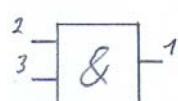
Ein Beispiel dafür wäre DIP16, ein Wort wie HC03 ist keine Gehäusebezeichnung.

Manche Hersteller schreiben dazu, in welchem Package der Bauteil ist, ansonsten muss man den Footprint selbst erstellen.

Als Letztes kann man eine Textbeschreibung hinzufügen. Diese werden erst wichtig, wenn man das Protel-File in eine Datenbank exportiert.

Ein Bauteil kann mehrere Ansichten haben.

- Normal
- Demorgan
- IEEE



Man kann so auch einen kleinen und einen großen Widerstand zeichnen. In einer digitalen Schaltung hat man meist nur Pull-Up Widerstände, diese werden besser klein gezeichnet, in analogen Schaltungen sollten sie aber größer sein. So kann man das Layout wesentlich verschönern.

Anmerkung:

Es gibt einen Button Update-Schematic, manchmal funktioniert er, manchmal nicht. Wenn er nicht funktioniert, muss man den Bauteil aus der Schaltung löschen, die Library neu laden und ihn erneut platzieren.

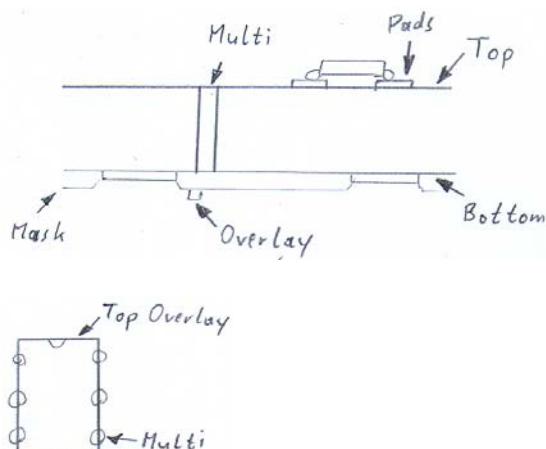
3.3 ERC (Electrical Rule Check)

Dieser Prüft, ob alle Ein- und Ausgänge korrekt zusammen geschalten sind.

3.4 Footprint Bibliothek

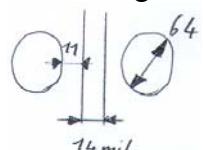
*.LIB

*.PLB



Bei Steckern, Tastern und ähnlichem muss angegeben werden, wo der PIN 1 sitzt. Alle Elemente müssen auf Grund des Overlays eindeutig eingebaut werden können.

In der HTL können wir 20mil verarbeiten. Für Zuleitungen kann man auch 30mil verwenden. Bei Lötaugendurchführungen gibt es eine Ausnahme:



Lötaugen müssen in der HTL einen Durchmesser von 80mil haben.

Achtung:

Protel Libraries haben standardmäßig weniger.

Einen großen Vorteil hat es, wenn man Lötaugen Oval macht.



3.5 Layer

Auf dem Mechanical Layer ist der Printumriss angegeben.

Es kann sein, dass die Fräse den Umriss als Mittelmaß oder als äußere Grenze betrachtet.

Wenn die Fräse den Umriss als Mittelmaß nimmt, fällt an jeder Seite die Hälfte des Bohrdurchmesser weg.

Die Bohrlöcher sitzen auf dem Multi-Layer.

Weiteres wird ein Keep Out Layer benötigt. Außerhalb dieses Layers können keine Bauteile platziert werden.

Montagelöcher nicht vergessen! Wenn möglich an allen vier Ecken, wenn er größer ist auch in der Mitte.

Montagelöcher sind non plated, das heißt, sie haben keine Kupferverbindung. Ansonsten könnte sich Kupfer im Loch ansammeln.

Um in der HTL das Befestigungsloch auch bohren zu können, wird ein Donut auf den Mechanical Layer gelegt, der Außenring sollte aber kleiner sein als der Bohrdurchmesser, damit er auch wirklich verschwindet. Es wäre auch möglich ein eigenes Symbol zu erstellen und es zu platzieren (z.B.: )

Name, Klasse, Seriennummer und Projekt müssen in gespiegelter Schrift auf dem Bottom Layer angebracht werden. (60mil)

Die Maße des Prints müssen beschriftet werden. Die Maße müssen in mm angegeben werden. Die Bemaßung (Mechanical Layer) muss gespiegelt werden. Dazu kann man diese in Teile zerlegen und dann nur die Schrift spiegeln, oder den Namen sehr klein machen, einen neuen dazuschreiben und diesen dann spiegeln.

Am Overlay (Bauteil Aufdruck) sind die Bauteile abgebildet (am Lötstopplack).

3.6 Bohrplan

Drill Draw  ...mehrere Durchmesser

Drill Guide + ...nur ein Symbol

Mit dem „Legend“ Text macht Protel automatisch an diese Stelle eine Legende der Bohrdurchmesser.

Es sollten so wenig verschiedene Bohrdurchmesser als möglich sein.

3.7 CAM

Für die Printerei müssen wir Gerber Files anlegen (im CAM Design).

Diese werden für das automatische Zusammenmontieren mit dem Fotoplotter benötigt.

Die Gerber Files sind textal für jede Maschine lesbar.

Es gibt verschiedene Formate (Vor- und Nachkommastellen).

→ In der Printerei fragen, welches Format verlangt wird.

Mit dem Programm Camtastic kann man sich erstellte Gerber Files Layer für Layer ansehen.

BOM (Bill of Material) ist die Stückliste.

Fürs Laborprotokoll:

Schaltungslibrary
Bauteillibrary
Schaltung
Printplan
Bestückungsplan
Bohrplan
Stückliste
Gerber Files (Nur auf Diskette)

Die Libraries auch auf A4 Blatt ausdrucken (ca 5 Bauteile auf einem A4 Blatt)

Bis auf die Gerber Files muss alles in elektronischer und gedruckter Form vorliegen.

3.8 Masse Flächen

Masseflächen können entweder als durchgehende Fläche, oder als Gitter aufgetragen werden.
HF-Technisch sollte die Gittergröße viel kleiner sein als die Wellenlänge.

Für normale Signale (kHz) ist das kein Problem.

Im Schwalllötverfahren (Industrie) ist die Gittermethode besser. Wenn ein Lötstopplack aufgetragen ist, ist es egal, welche Methode angewendet wird.

Wenn in die Massefläche PINs gesetzt werden, tritt genau dieses Problem auf. Abhilfe wird mit Thermal Boreholes geschaffen (Im Protel unter Power Plain Connection Style):



Vor dem Erstellen der Massefläche, sollte man die Width Constraint auf 40 oder 50mil setzen. Dann hält die Massefläche mehr Abstand von den anderen Leitungen. Danach wieder auf die Standard 20/12mil zurückstellen.

3.9 weitere Einstellungen

Angle Constraint:

ideal wären 90° (wichtig fürs Löten)



Hole Size Constraint

Man kann Protel den minimalen und den maximalen Bohrdurchmesser angeben.

Minimal Annual Ring Constraint

Wie klein darf der Restring minimal sein. (in der HTL 20mil)

Diese Angaben sind nötig, um einen sinnvollen DRC (Design Rule Check) durchzuführen.

3.10 Vias – PINs

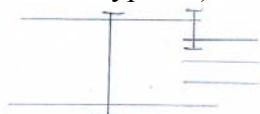
Ein Via ist im Gegensatz zu einem PIN eine Durchkontaktierung. Diese kann wahlweise mit Lötstopplack versehen werden.

In der HTL können keine Vias erzeugt werden. Man muss auf Lötbrücken ausweichen. Lötbrücken müssen gerade sein:



3.11 internal Layers

In der Industrie ist es möglich Prints mit mehreren Schichten zu erstellen (siehe Motherboard und Handyprints).



Die HTL kann keine Leiterbahnen im Inneren des Prints erstellen.

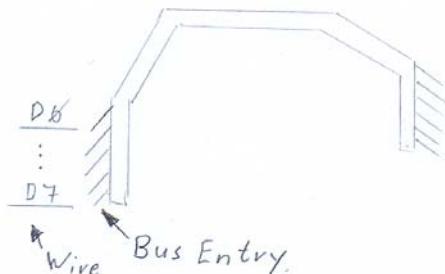
In den Mittellagen liegen meistens V_{CC} und GND (Plains). So braucht man diese oft benötigten Anschlüsse nicht routen (Plains sind komplett Ebenen aus Kupfer).

3.12 Bus

Wo ein Bus entsteht wird erstmal eine dicke blaue Linie gezeichnet.

Zum Bus hin werden mit „Bus Entries“ die Wires angeschlossen.

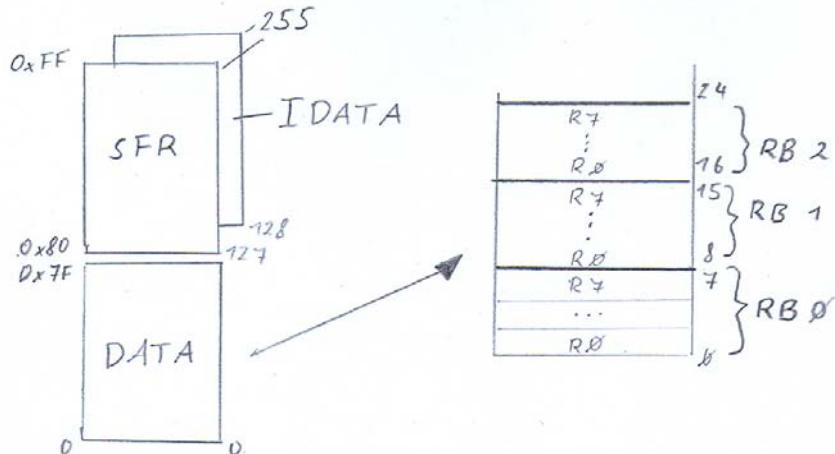
Mit „Net Label“ muss man den Datenleitungsnamen geben.



Dicke Linie mit „Place Bus“ zeichnen.

4 AT2051

Der AT2051 ist ein µC aus der Familie der 8051 der Atmel Familie.



4.1 DATA Bereich

In jeder Registerbank gibt es die Register R0 bis R7.

Wenn man ein Register ansprechen will, muss man auch die Registerbank angeben.

Um die einzelnen Registerbänke einstellen zu können, gibt es die Bits RB1 und RB0 (liegen im PSW).

RB_1	RB_0	Registerbank
0	0	0
0	1	1
1	0	2
1	1	3

Standardmäßig sollte man in die Registerbank 0 schreiben.

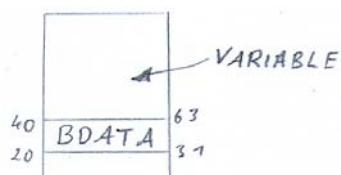
z.B.:

Den Dezimalen Wert 17 in das Register 0 verschieben.

MOV R0,#17

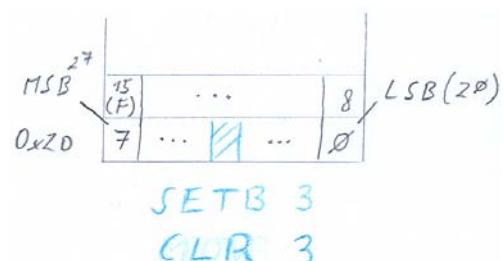
Die Registerbank wurde zuvor eingestellt.

Im Data-Block liegt von Bit 31 bis 63 (20 bis 40 Hex) der BDATA Bereich. In diesem Bereich kann man jedes Bit einzeln ansprechen. Dies hilft beim Sparen von Speicherplatz. Über den BDATA Bereich sollten die normalen Variablen gelegt werden.



Der Data Bereich ist sowohl direkt, als auch indirekt addressierbar.

4.1.1 BDATA



MSB most significant Bit

MSB...most significant Bit
LSB least significant Bit

Der Befehl **MOV 20, #0** würde zum Beispiel alle BITS im Register 20 auf einmal auf 0 setzen.

Es gibt keinen Bitzeiger, man kann Bits also nicht indirekt ansprechen

4.2 Stackpointer

Es ist zu beachten, dass der Stackpointer auf einen leeren Bereich zeigt (sonst überschreibt er die Variablen). Mehr Bereich für den Stackpointer bedeutet, dass man mehr Unterprogramme aufrufen kann, aber gleichzeitig weniger Platz für Variable hat.

Der Stack ist eine indirekte Adressierung.

Man kann den Stack in den IDATA Bereich hineinwachsen lassen. Die Variablen werden dann möglichst weit oben abgelegt.

4.3 SFR

Im SFR (Special Funktion Register) liegen Register mit speziellen Funktionen (Akkumulator, Serielle Schnittstelle, Ports, Timermodi, ...)

Das SFR eignet sich nicht, um etwas abzuspeichern. Es dient dazu, den µC richtig einzustellen.

4.4 IDATA

Der Bereich IDATA ist nur indirekt (über Pointer) adressierbar.

z.B.:

MOV R0,#255

MOV @R0,#17

R0 wird hier als Pointer auf das IDATA Register 255 benutzt.

In C bezeichnet man dieses als Zeiger.

**pcx=17*

Es gibt im 2051 nur zwei 8 Bit Zeiger (R0 und R1).

Eine indirekte Adressierung ist immer langsamer als eine direkte Adressierung.

4.5 Zahlensysteme in Assembler

Assembler:

<i>MOV A,#0</i>	dezimal
<i>MOV A,#0FFh</i>	hex
<i>MOV A,#10100011b</i>	binary

C:

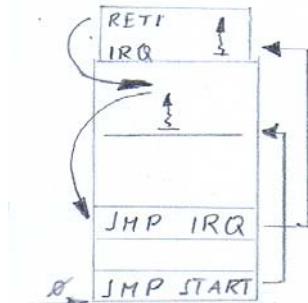
<i>cx = 17</i>	dezimal
<i>cx = 017</i>	octal
<i>cx = 0x17</i>	hex

Achtung:

In Assembler braucht man die Null für Hex-, in C für Octalsystem.

4.6 Codespeicher

Der Codespeicher ist in neueren Modellen als Flash, in älteren als PROM, EPROM oder E²PROM ausgeführt.



Die untere Hälfte des Codespeichers ist die IRQ (Interrupt Tabelle).

Dort stehen nur JMPs auf die oberen Zeilen. Man schreibt nicht die ganze Inerrupt Routine in die Interrupt Tabelle, da man sonst andere Interrupts überschreiben würde.

Ein Programm im µC fängt mit einem Initialisierungsblock an und hat am Ende immer eine Endlosschleife. Denn ansonsten würde er nach dem Programm immer weiter Speicher einlesen und diese unsinnigen „Befehle“ ausführen.

4.6.1 Assembler Programmierung

Hier ein kurzes Beispiel, wie die Assemblerprogrammierung einer solchen Interrupttabelle aussehen könnte.

```
ORG 00h ;ORG schreibt den Befehl an der Stelle 00Hex in den Codespeicher
JMP START
ORG 0Bh
JMP IRQ
ORG 30h
START: ;beschreibt eine Adresse
...
JMP START
END
```

Es gibt in Assembler 3 Jumps:

- SJMP (± 127 Byte rel.)
- AJMP (2k)
- LJMP (0-64k absolut)

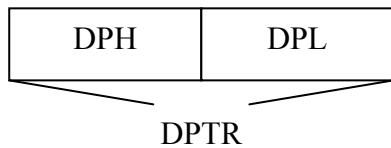
Schreibt man nur JMP, sucht der Assembler in 95% der Fälle den richtigen JMP Befehl.

4.6.1.1 MOVC

Mit dem MOVC Befehl und mit Hilfe des Datapointers DPTR kann man Codespeicher auslesen, um z.B. Tabellen oder Konstante auszulesen.

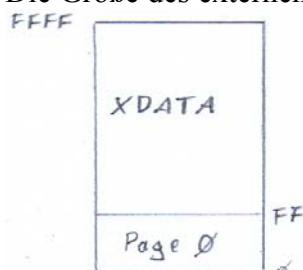
MOVC A,@DPTR ;DPTR ist ein 16-Bit Zeiger

MOV kann aber nur 8Bit bewegen, deshalb wurde der DPTR in DPH (Datapointer High) und DPL (Datapointer Low) aufgeteilt. Diese beiden kann man dann mit 8Bit auslesen.



4.7 Externes RAM

Für einen Externen Speicher wird der Port 0 und der Port 2 benötigt.
Die Größe des externen RAMs wären 64k, was riesengroß ist.



Mit MOVX kann man dann auf den externen Speicher zugreifen (nur indirekt).
z.B.:

*MOVX @DPTR,A
MOVX @R1,A*

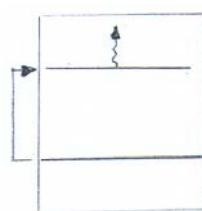
Mit dem DPTR kann man auf die ganzen 64k zugreifen, mit einem 8bit Pointer nur auf die Page 0.

4.8 Assemblerprogrammierung beim 2051

4.8.1 Sprung und Unterprogramm

Sprünge:

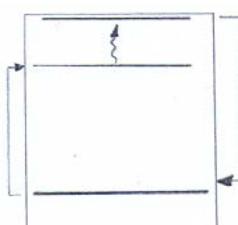
- SJMP (± 127 Byte rel.)
- AJMP (2k)
- LJMP (0-64k absolut)



Bei einem Sprung auf eine bestimmte Adresse wird an dieser Stelle das Programm weiter abgearbeitet.

Unterprogrammaufrufe:

- ACALL (2k)
- LCALL (0-64k absolut)



Bei einem Unterprogrammaufruf wird das Unterprogramm aufgerufen. Ist das Unterprogramm abgearbeitet, so wird das Programm in der Zeile nach dem Aufruf weiter ausgeführt. Die Adresse, zu der zurückgesprungen wird, wird im Stack abgelegt. Zu viele Unterprogrammaufrufe (rekursiv programmiert) können zu viel Speicherplatz am Stack benötigen und so Variable überschreiben, oder aus der Speichergröße hinausreichen (sinnlose Werte).

4.8.1.1 Bedingte Sprünge

- JC (Jump Carry)
- JNC (Jump not Carry)
- JZ (Jump Zero)
- JNZ (Jump not Zero)
- JB (Jump Bit)
- JNB (Jump not Bit)
- CJNE (Compare jump not equal)

Wenn der erste Wert nicht gleich dem zweiten ist, springe.

Bei Ungleichheit wird das Carry 0 oder 1 gesetzt (je nachdem, welcher Wert größer ist). Mit JC und JNC kann < und > realisiert werden.

Das Carry steht im PSW (Überlauf).

Das Zero wird gesetzt, wenn bei der vorherigen Operation Null herausgekommen ist.

Fragt, ob das Bit 0 oder 1 ist.

4.8.1.2 Schleifen

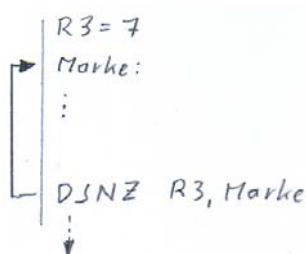
Schleifen sind eine Art von Sprüngen.

DJNZ (Decrement, jump not zero)

Der Befehl zählt eines hinunter. Wenn die Variable nicht 0 ist, springt er zur Marke, ansonsten arbeitet er das Programm weiter ab.

DJNZ R3, SPRUNGMARKE

Zuerst wird dem R3 ein Wert zugewiesen.



4.8.2 Operationen

4.8.2.1 Mathematische Operationen

ADD $Z1 = Z1 + Z2$ wurde ein Überlauf erzeugt, wird das Carry gesetzt

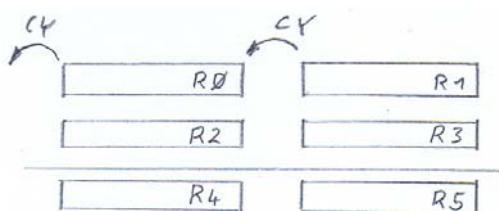
Carry = 0 heißt 0 - 255

Carry = 1 heißt 256 - 510

ADDC $Z1 = Z1 + Z2 + C$

Wird dazu benutzt um z.B. 24Bit Zahlen zu addieren. Dabei muss man dann immer die 8 Bit Zahlen addieren und das Carry (Überlauf vom letzten Mal) mitaddieren.

z.B. Addition zweier 16Bit Zahlen:



ADD R1,R3

MOV R5,R1

ADDC R0,R2

MOV R4,R0

JC?...

CY...Carrybit

SUB $Z1 = Z1 - Z2$

C = 0 bedeutet Zahlen von 0 bis 255

C = 1 bedeutet Zahlen von -1 bis -255

SUBB $Z1 = Z1 - Z2 - C$

B steht für Borrow. Für die Funktionsweise siehe ADDC

MUL AB



DIV AB



Der µC kann nicht mit float Zahlen rechnen, oder zumindest bindet der C-Compiler dann automatisch eine 2k Bibliothek ein, die das kann. 2k sind zuviel dafür.

4.8.2.2 Logische Operationen

ANL AND

ORL OR

XRL XOR

CPL INV

Das L steht für Logic.

Es werden jeweils 8 Bit Zahlen verknüpft.

Syntax:

ANL R0, R1; der Wert wird ins Register 0 geschrieben.

z.B.: 17 & 3 | 8

10001 & 11 | 1000

00010001
00000011
----- UND
00000001
00001000
----- OR
00001001

z.B.: XOR

10101100
00000110

10101010

Der **rote Teil** wurde invertiert.

Man kann mit XOR also Teile einer Zahl invertieren (indem man sie mit 1 XOR verknüpft).

z.B.: UND

10101100
11111001

10101000

Mit UND kann man löschen (mit 0 UND verknüpfen) – siehe **roter Bereich**.

Diese Vorgänge nennt man Maskierung (Bits in einem Byte setzen/löschen).

Bsp.: Überprüfe, ob das dritte Bit in einem Byte gesetzt ist. (Bereich nicht Bit adressierbar)

ANL R0,#00000100b
CJNE R0,#4,_____

Die logischen Verknüpfungen gibt es auch für Bits, dabei wird meistens mit dem Carry verknüpft.

4.8.3 Weitere Befehle

NOP...wartet einen Zyklus

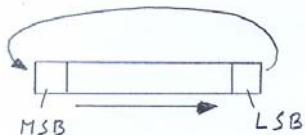
NOP (No Operation) ist ein Zeitvernichtungsbefehl.

z.B.: Wenn man am Port 1 einen kurzen Impuls ausgeben will. Wenn man den Port nur auf 1 setzt und danach wieder auf 0, kann es für die angeschlossene Hardware zu kurz sein.

INC...increment (++)
DEC...decrement (--)

4.8.4 Rotierbefehle

Es gibt Rotate Left ROL, Rotate Right ROR und das ganze mit oder ohne Carry (ROLC, RORC). Bei ROR wird das LSB an die Stelle des MSB geschrieben und alles nachgerückt. Bei Operation mit Carry wird das Carry als MSB verwendet und das LSB ins Carry geschrieben. Das Carry kann danach verglichen werden.



In C gibt es dafür keine Befehle, es gibt aber die Schiebebefehle `>>` und `<<`. Mit den Rotierbefehl kann man in Assembler die Schiebebefehle nachbilden. Dabei wird ins Carry 0 geschrieben, dann rotiert und das neue Carry verworfen.

Die Rotierbefehle rotieren jeweils nur einmal, in C kann man sich aussuchen, wie oft geschoben werden soll.

Mit Carry kann man auch 16 Bit Zahlen rotieren.

4.8.5 Fehlersuche

Man kann den Breakpoint dorthin setzen, wo der Fehler vermutet wird, dadurch braucht man nicht das Programm ewig lange durchgehen lassen.

Dieses Schema kann auch angewendet werden, wenn es Probleme mit einem Timer Interrupt gibt. (Breakpoint in den Interrupt setzen)

Manche Fehler erkennt man erst, wenn man den IC in die Schaltung einbaut.

Dazu kann man am Anfang einer Funktion einen Port PIN setzen und am Ende wieder löschen. So erkennt man, wie lange er in der Funktion braucht, bzw. ob er diese jemals verlässt.

Wenn keinen Ports frei sind, kann man zum Debuggen bereits verwendete benutzen. Die Funktionen, die diese verwenden müssen dazu aber abgeschaltet werden!

Will man sich Signale am I²C Bus mit dem Oszi ansehen, empfiehlt es ich an einem nicht benutzten Port einen kurzen Takt zu senden, bevor das eigentliche Signal kommt. Dann kann man mit dem Oszi auf diesen Takt triggern.

4.8.6 Pre Prozessor Befehle

\$EP Errorprint Fehler werden im Listing mit ausgegeben
\$DB Debug man kann mit Variablennamen debuggen (sonst nur mit Adressen)
\$include (*DateiName*) man kann wie in C Dateien einbinden (oder ein Registermodell)
\$NOMOD51 schaltet das standardmäßige Registermodell vom 2051 aus.

In KeilµVision werden diese Befehle nicht zwingend benötigt, da sie standardmäßig eingeschaltet sind.

Beim Einfügen anderer Registermodelle muss zuerst das Alte ausgeschalten werden. (mit \$NOMOD51)

4.8.7 Definitionen

NAME EQU <i>Text</i>	Textmarke erzeugen
V1 EQU 17	Variable erzeugen (17 = Speicherstelle)
K1 EQU 32	Konstante (Aufruf mit MOV V1,#K1)
K2 EQU (K1+20)	Rechenoperationen

Achtung:

Dem Assembler ist es egal, ob K1 eine Variable oder eine Konstante ist. Erst durch den Einsatz von # wird K1 zu einer Konstanten.

4.8.8 Tabellen

DB...define Byte (8Bit)	
DW...define Word (16Bit)	
DD...define Double Word (32Bit)	
DS...define Segment	Erzeugt eine Tabelle mit X Elementen (z.B.: DS 6)
BIT...define BIT	

Man kann nicht bestimmen, auf welche Adresse die Tabelle gelegt wird.

Die Länge bezieht sich auf die Tabellenbreite (z.B.: 16Bit).

Beispiele für Initialisierung und Zugriff:

NAME DS 8
MOV DPL, Low (Name)

MOVX @DPTR, #12
INC DPTR

4.9 Registermodell

Timer

TCON	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0x88	bitadr.
TMOD	G	C/T	M1	M0	G	C/T	M1	M0	0x89	
Timer 1						Timer 0				
TH0 (0x8C)						TL0 (0x8A)				
TH1 (0x8D)						TL1 (0x8B)				

M1, M0 Modus

0,0 0 8Bit Prescaler (5Bit TH)

0,1 1 16Bit

1,0 2 8Bit Autoreload (8Bit TH)

1,1 3 Spezialmodus: 2x 8Bit, T1 aus, (T1 Steuerung für TH0, T0 Steuerung für TL0)

G=1 Gate enabled (INT0/1 Pin P3.2, P3.3 steuert run/stopp)

C/T=1 Counter enabled (Taktquelle ist Pin P3.4, P3.5)

Serielle Schnittstelle

SCON	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	0x98	bitadr.
SBUF									0x99	
PCON	SMOD								0x87	
ADCON	BD	CLK	ADEX	BSY	ADM	MX2	MX1	MX0	0xD8	bitadr. nicht im 2051

M0, M1 Modus

- 0,0 0 8 Bit Shift Register (fix, synchron), BaudR = 8Bit $f_{osc}/12$
- 0,1 1 8 Bit UART (T1), BaudR = $f_{osc} \cdot 2^{SMOD} / (12 \cdot 32 \cdot (256 - TH1))$
- 1,0 2 9 Bit UART (fix), BaudR = $f_{osc} \cdot 2^{SMOD} / 64$
- 1,1 3 9 Bit UART (T1), BaudR = $f_{osc} \cdot 2^{SMOD} / (12 \cdot 32 \cdot (256 - TH1))$

Interrupt

IE	EA	WDT EADC	ET2	ES	ET1	EX1	ET0	EX0	0x89	bitadr.
IP		WD1 EADC	ET2	ES	ET1	EX1	ET0	EX0	0xA9	
TCON	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0x88	bitadr.

CPU

ACC, A	.7						.0	0xE0	bitadr.	
B	.7						.0	0xF0	bitadr.	
PSW	CY	AC	F0	RS1	RS0	OV	F1	P	0xD0	bitadr.
SP									0x81	
DPH (0x83)					DPL (0x82)					

4.9.1 CPU

ACC, A	Accumulator
PSW	Program Status Word
F0,F1	Benutzerdefiniert
CY	Carry (bei 255)
AC	Auxiliary Carry (von 15 auf 16)
OV	Overflow (bei 127)
RS0, RS1	Registerbankumschaltung
P	Parity
SP	Stackpointer

Zählt man zu 255 eins dazu, wird die Zahl 0 und das Carry gesetzt (CY = 1). Bei einem signed Character (-127 bis + 128) wird bei einem Überlauf das Overflow Bit gesetzt (OV = 1). So wird aus $-127 - 1 = + 128$ (OV = 1). Als Programmierer muss man selbst wissen, welches Bit man nun braucht.

Das AC wird gesetzt, wenn ein Überlauf von den ersten 4 Bit eines Bytes geschieht.
Das wird oft bei BCD-Anzeigen benutzt, da man so zwei 7-Segment-Anzeigen in einer Variablen (8 Bit) speichern kann.

Um eine Zahl in den oberen Bereich der 8 Bit Zahl zu schreiben, kann folgender Syntax verwendet werden:

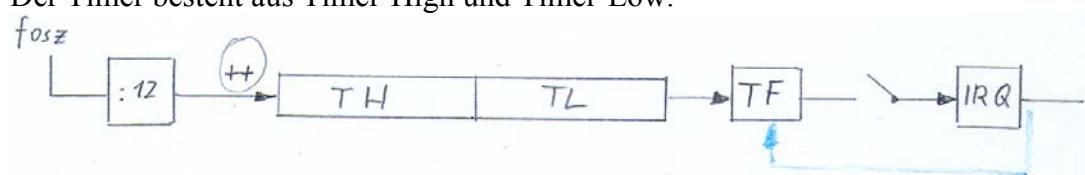
`var &= 0x0F;` oberen Speicher löschen

`var |= (8<<4);` ODER Verknüpfung – Zahl 4 mal geschoben

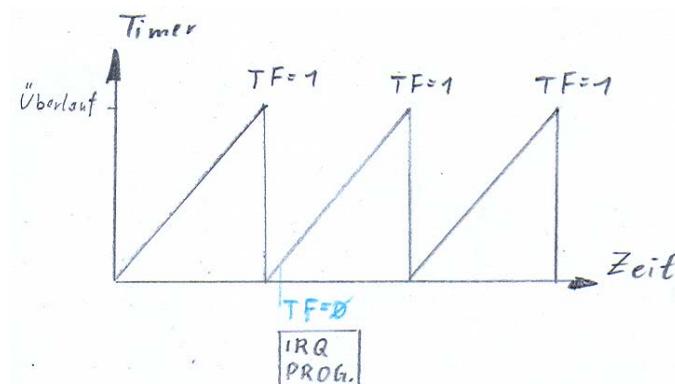
`var |= (8*16);` statt dem 4 mal schieben, kann man auch mit 16 multiplizieren.

4.9.2 Timer

Der Timer besteht aus Timer-High und Timer-Low.



Jedes Mal, wenn der Timer überlauft, wird das Timer Flag (TF) gesetzt, was einen Interrupt auslösen kann.



Kommt ein Interrupt, löscht dieser automatisch das TF.

Das dauert deswegen ein wenig, weil dazwischen der JMP-Befehl ausgeführt werden muss.

Ließt man das TF selbst aus, so ist diese Verzögerungszeit unbestimmt.

TF0...Timer Flag 0

TF1...Timer Flag 1

TR0...Timer Run 0 (ein und ausschalten)

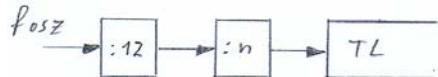
TR1...Timer Run 1

M0, M1... setzen des Timermodus

M0, M1	
0	5 Bit Prescaler
1	16 Bit Timer
2	8 Bit Autoreload
3	macht aus 16Bit Timer 2Bit Timer (selten benötigt)

1 und 2 können je nach Datenblatt unterschiedlich sein.

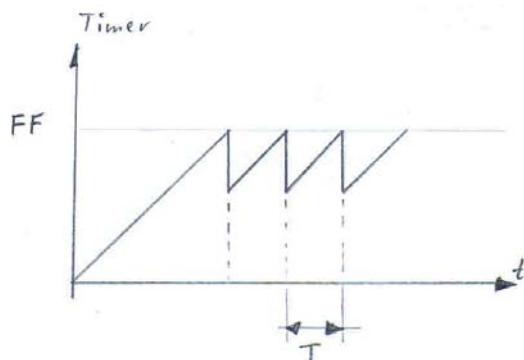
Mit dem Modus 0 kann das Oszillatorsignal zusätzlich verlangsamt werden (/n).



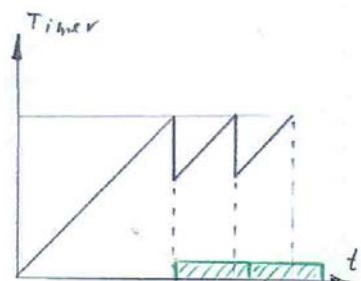
Timer zählt die Zeit, Counter zählt Ereignisse

16 Bit Zähler verwendet man für langsame Sachen (ms Bereich)

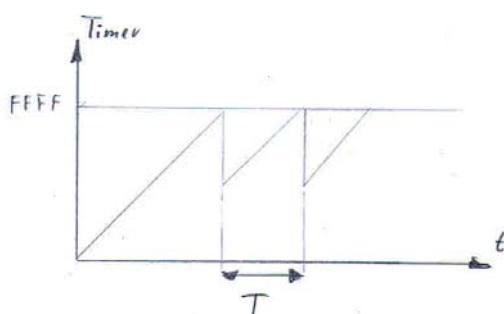
Beim 8Bit Autoreload wird bei einem Timerüberlauf der TH in den TL kopiert. (der TL zählt nach oben). In den TH kann vorher ein beliebiger Wert geschrieben werden. Dadurch ergibt sich immer die selbe Zeit. Das wird für schnelle Zeiten verwendet ($1\mu s - 255\mu s$).



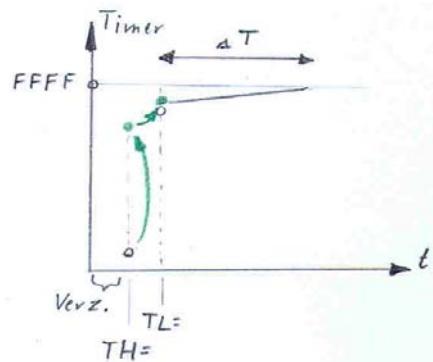
Wenn das Interrupt-Programm allerdings länger dauert als bis zum nächsten Inerrupt, kommt man nie wieder raus (μC hängt sich auf).



Beim 16 Bit Zähler stellt man die Zeit ein, indem man einen Wert in TH und TL schreibt.
(Die Hardware macht das nicht automatisch! Für langsamere Vorgänge)



Die Abarbeitung der Setzen-Befehle braucht Zeit, während dieser Zeit zählt der Timer bereits weiter. Werden genaue Zeiten benötigt (z.B. Uhr) muss die Totzeit berücksichtigt werden.

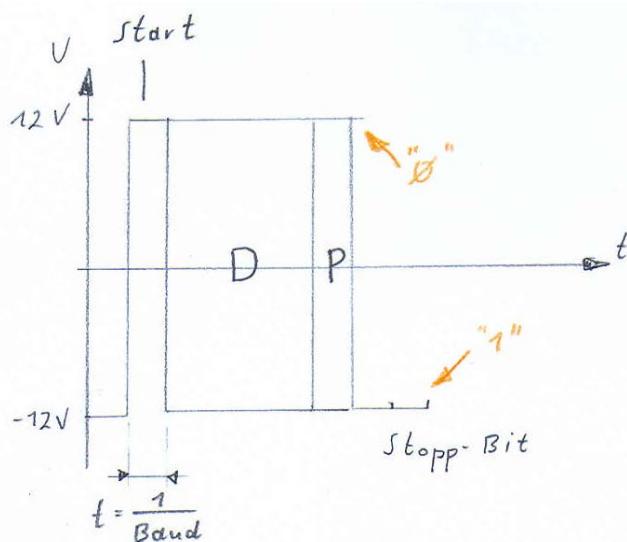


Liegt vorher im Programm sogar eine if-Abfrage, ist die Zeitveränderung nie wieder konstant. (Fehler kann auch nicht berechnet werden).

In der Interruptroutine sollte also immer zuerst TH gesetzt werden und dann TL. Wenn man zuerst den TL setzt, kann das Fehler bis zu 255 Byte verursachen.

Bei manchen Anwendungen ist dieser Fehler vernachlässigbar.

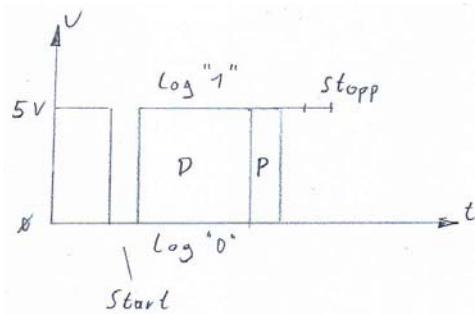
4.10 RS232 (Serielle Schnittstelle)



Beim Sender kann man von 5-18V senden, um noch als high erkannt zu werden (-5 - -18V) für Low.

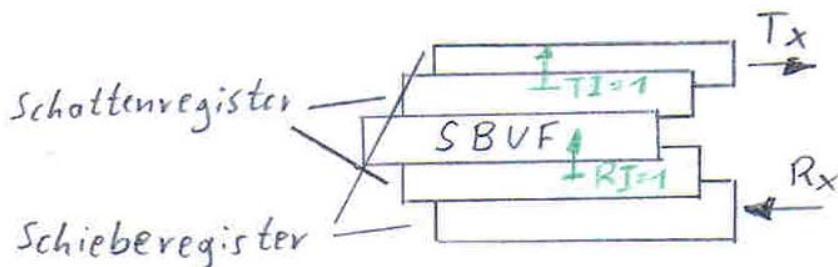
Der Empfänger erkennt das Signal zwischen 3 und 23V.

Zwei μ C kann man einfach ausgekreuzt verbinden (bei RS232), da die Schaltschwelle genau in der Mitte der Pegel liegt.



4.10.1 SBUF Register

Um etwas über Serielle Schnittstelle empfangen bzw. senden zu können, wird das SBUF Register benutzt.



Bei einem Lesevorgang wird aus dem Schattenregister (R_x – Seite) gelesen.

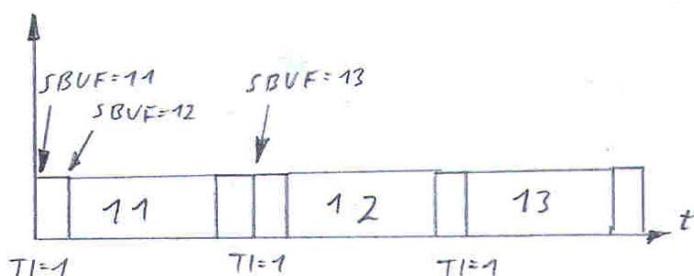
Bei einem Sendevorgang wird in das Schattenregister (T_x – Seite) geschrieben.

Schattenregister...nicht sichtbares (ansteuerbares) Register

Ist das Sendeschreiberegister leer, wird das Schattenregister ins Schreiberegister kopiert, ein „Start“ und ein „Stopp“ Bit hinzugefügt und gesendet.

Bei diesem Kopievorgang wird $T_I = 1$ gesetzt, es muss manuell auf 0 gesetzt werden.

Beispiel:



Nachdem „11“ gesendet wurde ist das Schattenregister leer → nächster Wert („13“) kann gesendet werden.

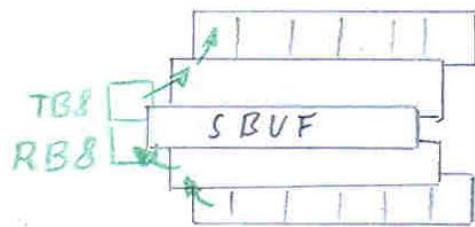
Das Empfangen funktioniert analog, es wird $R_I = 1$ gesetzt.

Bei 9600Baud dauert ein Senden ~1ms (~1000µC Zyklen).

Es hat also keinen Sinn eine Schleife zu machen, um darauf zu warten:

```
while(!TI)
{
}
```

TB8 und RB8 sind die 9. Bits für eine 9 Bit Übertragung.



Dieses Bit wird meistens als Parity-Bit verwendet.

Vor dem **SBUF=** muss das TB8 gesetzt werden.

Das RB8 kann erst gelesen werden, wenn das SBUF abgefragt wurde.

SM0 und SM1 stellen den Modus für die Übertragung per RS232 ein.

Modi:

- | | |
|---|--|
| 0 | 8 Bit reines Schieberegister ohne Start- und Stoppbit (BaudR = $f_{OSC}/12$) |
| 1 | 8 Bit UART (T1) $BaudR = f_{OSC} \cdot 2^{SMOD} / (12 \cdot 32 \cdot (256 - TH1))$ |
| 2 | 9 Bit UART (fix) $BaudR = f_{OSC} \cdot 2^{SMOD} / 64$ |
| 3 | 9 Bit UART (T1) $BaudR = f_{OSC} \cdot 2^{SMOD} / (12 \cdot 32 \cdot (256 - TH1))$ |

UART... Universal, asynchrony, Receiver, Transmitter

Die Taktrate dafür ist der Timer 1 Überlauf (kein Interrupt benötigt).

Modus 1 und 3 haben variable Baud Raten.

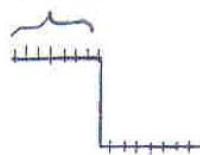
Modus 2 ist die schnellste BaudR, allerdings fix vom Oszillatortakt abhängig (zwischen zwei µC verwendet).

$$\frac{2^{SMOD}}{32} \quad 2^0 = 1 \quad \frac{1}{32}$$

$$2^1 = 2 \quad \frac{1}{16}$$

Wird dazu benötigt, damit die Flanke des Signals steil genug ist.

16 oder 32 Teilschritte



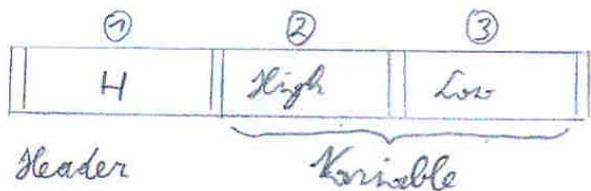
SMOD ist standardmäßig 0.

Das Bit SM2 ist sehr speziell, es schaltet den Multiprozessor-Modus ein (mehrere µC senden über einen Bus).

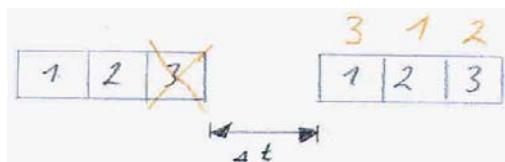
BD...gibt einen zusätzlichen Timer, mit dem man eine fixe BaudR einstellen kann.

4.10.2 Framing

Framing bedeutet, dass man Bytes gleich hintereinander sendet, weil die Anweisung oder Variable länger als 1 Byte ist.



Dabei muss man wissen, ob zuerst der High- oder der Low-Teil gesendet wird, um es im Prozessor wieder zusammensetzen zu können.



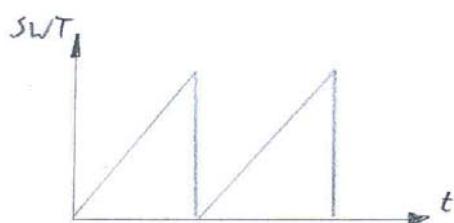
Geht ein Paket verloren, ist die ganze Reihenfolge falsch.

Ein Softwaretimer löst das Problem, er erkennt die Zeit zwischen den Befehlen und merkt so, wann ein neues Element kommt.

So ist nur der erste Teil kaputt, der Rest funktioniert wieder.

4.10.3 Software Timer

Im IRQ zählt man eine Variable dazu:



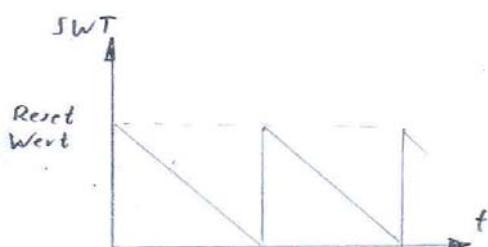
Ein Software-Timer kann bis 256 hinaufzählen.

Dadurch erhält man langsamere Zeitintervalle.

Man kann beliebig viele Software-Counter erstellen.

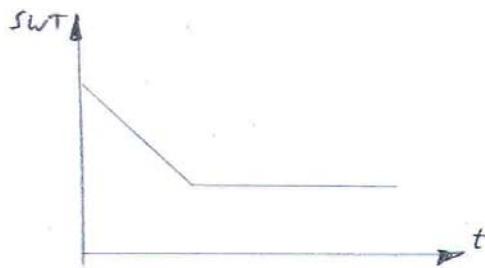
Auch hinunterzählen und Reload-Werte sind möglich.

Der Vorteil am runterzählen ist, dass die Reload-Werte stimmen.



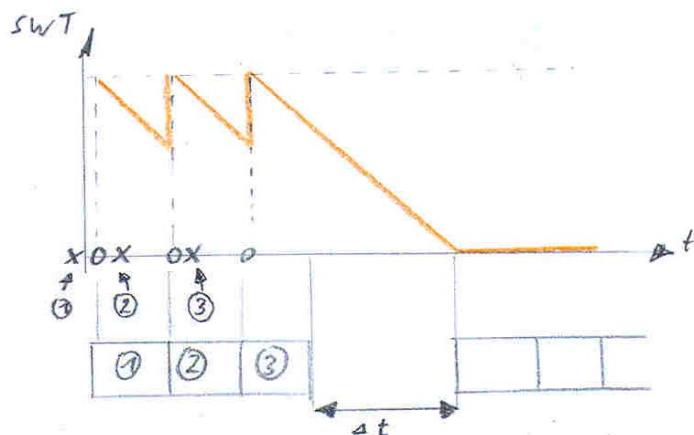
Er kann auch nur einmal laufen und dann auf einem bestimmten Wert bleiben.

```
if(SWT<17)
{
    SWT++;
}
```



Dies wird benutzt, um nach einem Event (Tastendruck) eine gewisse Zeit zu warten (Timer braucht danach nicht mehr zu laufen).

Beim Framing wird diese Methode angewendet.



x...SBUF setzen

o...TI gesetzt (Schattenregister frei)

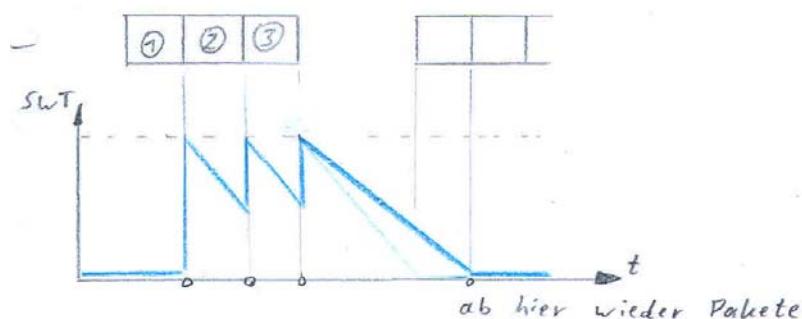
Erst wenn der Software Timer auf 0 ist, kann wieder geschrieben werden.

Wird geschrieben, wird der SWT zurückgesetzt.

Δt muss größer als 1 Byte sein.

Solange der SWT nicht auf 0 ist, muss allen Programmen das Paketsenden verwehrt werden.

Empfangen:



Der Sendetimer muss länger sein als der Empfänger, denn ansonsten kann es sein, dass der Empfänger das Signal wegen ein paar ns nicht annimmt.

Läuft der SWT ab, kann der Index der Pakete auf 1 gesetzt werden.

4.11 Handshake

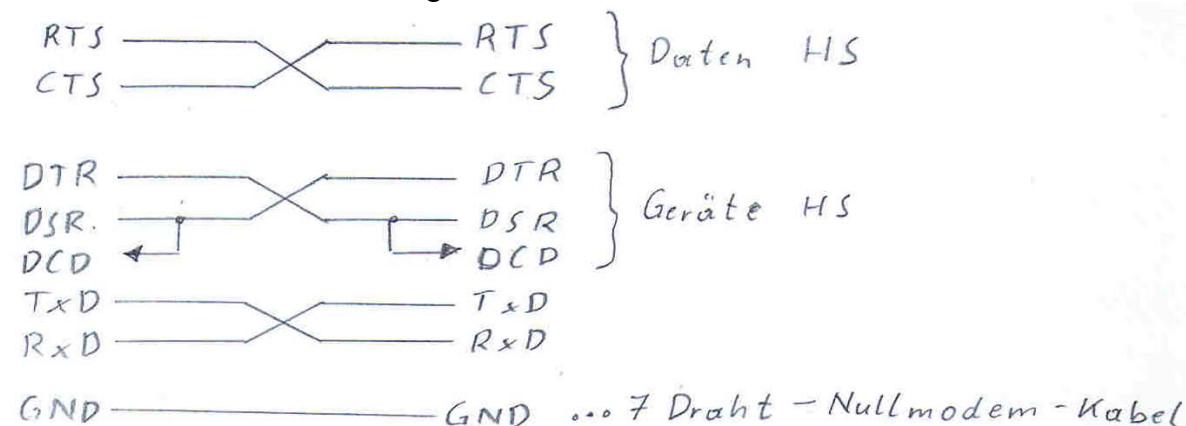
HS...Handshake

4.11.1 Hardware Handshake

Ein Gerät teilt dem anderen Gerät mit, dass es alle Daten korrekt empfangen hat, oder dass es gerade keine Daten verarbeiten kann.

Bei der RS232 sagt ein Gerät dem Anderen, ob es empfangen kann oder nicht.

Bei der RS232 sind das 5 Leitungen:



RTS...Request to send

CTS...Clear to send

DTR...Data terminal ready

DSR...Data set ready

DCD...Data carry detect

Das DCD war dazu gedacht um einen Datenträger zu erkennen und dies dem Gerät mitzuteilen.

Ein 3-Draht-Nullmodem Kabel hat nur TxD, RxD und GND.

Manche Geräte geben Daten-HS aus, manche Geräte-HS, manche gar keine, manche beides.

Um solche Geräte zusammenzuhängen gibt es also viele Möglichkeiten.

Wenn man ein Gerät, das keinen HS ausgibt mit einem zusammenhängen will, das einen braucht, benutzt man folgenden Trick:



Die Leitungen werden entweder am Print oder im Kabel kurzgeschlossen.

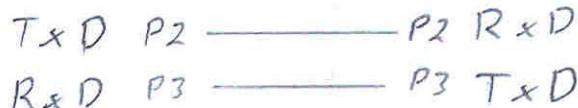
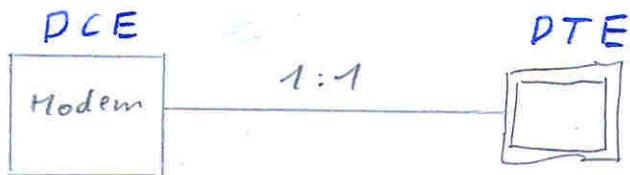
So gibt sich das Gerät mit HS selbst den HS. (Ich kann senden → du kannst empfangen)

Man sollte dies bei Projekten immer vorsehen, um Kompatibilität zu gewährleisten.

Es gibt auch noch 2 Gerätetypen:

DCE...Data Communication Equipment

DTE...Data Terminal Equipment

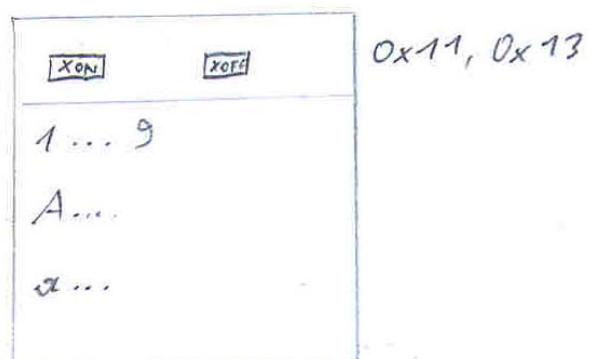


Zwischen zwei DTE (z.B.: 2 PCs) braucht man ausgetauschte Kabel.
Zwischen DTE und DCE braucht man ein 1:1 Kabel.

4.11.2 Software Handshake

Es gibt zwei verschiedene Kommandos für Software HSs.

X_{ON} / X_{OFF}



In den ersten Zeichen des ASCII Tabelle sind Steueranweisungen (z.B.: LF, CR, etc.), so auch X_{ON} und X_{OFF}.

Das Senden von X_{OFF} oder X_{ON} dauert ein Zeichen Sendezeit.

Es können also noch bis zu 2 Zeichen vom Gegenüber gesendet werden. (Schattenregister)

Wenn beim Senden des X_{OFF} ein Bitfehler passiert, gehen Daten verloren.

Schickt man z.B. ein A und es passiert ein Fehler, kann das A zu X_{OFF} werden. Dann schickt das Gegenüber nie wieder.

Deshalb schickt man am Anfang oder Ende einer Kette immer X_{ON} mit.

Achtung:

Beim Senden von binären Daten müssen die Werte für X_{ON} und X_{OFF} ausgelassen werden.

Deshalb schickt man solche Daten meist im ASCII-Code, dass hat außerdem den Vorteil, dass ein Mensch an einem Terminal die Werte ablesen kann (Wenn die Übertragungs geschwindigkeit keine Rolle spielt wird auch oft °C oder ähnliches mit übertragen)

Die Übertragung dauert dadurch länger.

017 wird dann als 3*8 Bit übertragen.

Außerdem ist das Umcodieren zeitaufwändig (Der PC kann es schnell zurückrechnen, aber der µC plagt sich.)

Eine andere Möglichkeiten dieses Problem zu übergehen sind **ESC-Sequenzen**.

X_{ON}	X_{OFF}
17	19
<i>Esc</i>	<i>Z₁</i>
27	11
<i>Z₂</i>	20

Die Zahlen 17 und 19 kann man nicht senden. Aber es wird ein Trick angewandt.

$$17 = \text{ESC} + 18$$

$$19 = \text{ESC} + 20$$

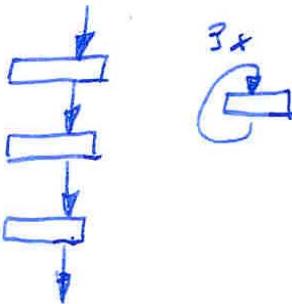
$$\text{ESC} = \text{ESC} + \text{ESC}$$

Die Umwandlung erfolgt schnell, wenn ein ESC kommt, wird das nächste Zeichen mit einer Tabelle umgewandelt bzw. wenn ein Sonderzeichen kommt, dann wird $\text{ESC} + ?$ gesendet. Dieses System findet auch in anderen Bereichen Benutzung, z.B. bei Windows in der Registry, hier ist „\“ ein Steuerzeichen, um ein echtes „\“ zu bekommen, schreibt man „\\“.

4.12 Tipps zur A51 Programmierung

- Keine Schleifen im IRQ
Um eine gewisse Zeit zu bekommen, benutzt man Softwaredimmer.
- IRQ kurz
 - ~ 3-4 C Zeilen / Aufgabe
 - ~ 10-20 A Zeilen / AufgabeBraucht man lange Rechnungen, benutzt man Flagging.
Im IRQ:
 $bF = 1; (\text{SETB } bF)$
Im Main:
 $if(bF==1)$
{
 $bF = 0;$
Aufgabe...
}
Um einen IRQ einem anderen IRQ etwas mitteilen zu lassen wird auch Flagging verwedent.
- Bei SWT immer Bereichsabfragen verwenden.
 $if(CNT>=17)$, nicht aber $if(CNT==17)$
In A51:
 $CJNE$
 JNC
- Felder bei der µC-Programmierung vermeiden.
Wenn Felder notwendig, dann keine unnötig großen (wenig Speicher).

- Mehrere ähnliche Blöcke zusammenfassen. (kein Copy & Paste)



- Interrupts

```
ORG 00
JMP INIT
ORG 0Bh
JMP TIM_IRQ
ORG 23h
JMP RS232_IRQ
```

Man kann IRQs nicht gleich an diese Stelle schreiben, da sonst andere IRQs überschrieben werden.

Ausnahmen sind Assembler IRQs mit wenigen Zeilen bzw. der letzte IRQ.

Mit ORG steht der Code an einer bestimmten Stelle im µC-Codespeicher.

00...Start

0B...T0

1B...T1

23...RS232

- SBUF auslesen

Nach dem ersten Mal lesen ist das SBUF leer.

```
CX = SBUF;
if(CX==...)
if(CX==...)
```

- Stack

Der Anfang des Stacks sollte auf #60 gesetzt werden.

MOV SP,#60

Alles über dem Stackpointer darf nicht als Speicherplatz verwendet werden.

- Beispiele

Ins R0 der RB0 schreiben:

MOV R0,#17

Ins R0 der RB1 schreiben:

SETB RB0

CLR RB1

MOV R0,#18

Ins SBUF schreiben:

MOV SBUF,#20

- Timer für RS232

Wird der Timer 1 verwendet, läuft er meist im Auto Reload Modus, um möglichst schnell zu sein.

Der T1 IRQ muss deaktiviert werden, da er länger als der Reload dauern würde. (Für Zählungen T0 verwenden)

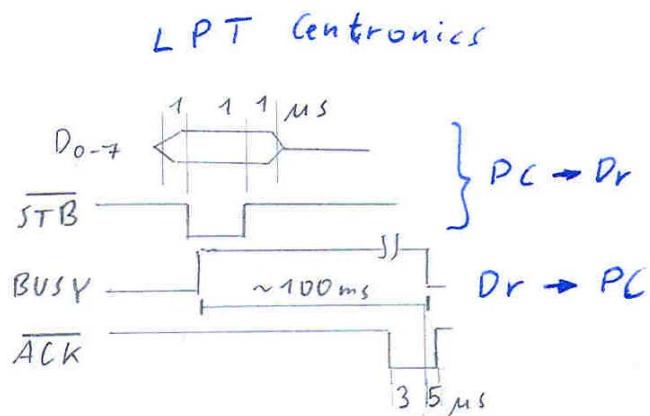
- EQU
Bei EQU immer vor den Namen schreiben, ob es sich um eine Variable oder eine Speicherstelle handelt.
V_NAME EQU 17
K_NAME EQU 17
- Beispiel für DJNZ-Schleife

MOV R0, #MAX
L1:
:
DJNZ R0, L1
↓

- Beispiel für \leq und \geq Abfrage
 $CJNE A, #17, W1$

...
W1: $A \neq 17$
JC MARKE1 $A > 17$
JNC MARKE2 $A < 17$

5 LPT Centronics (Parallele Schnittstelle)



STB...Strobe

ACK...Acknowledge

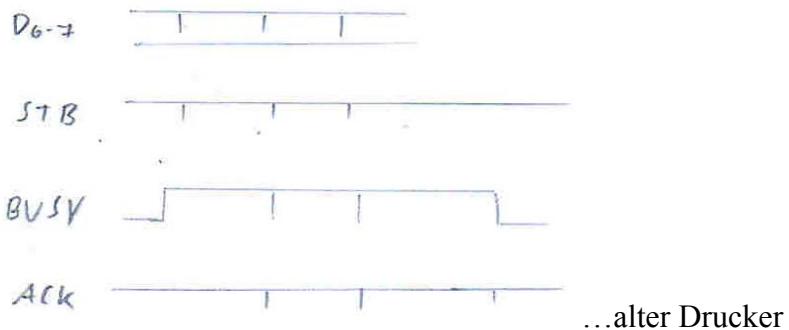
D...Data

Dr...Drucker

Problem:

Das Datensenden dauert μ s, das Verarbeiten (Drucken) ms.

Am Oszilloskop sieht das ungefähr so aus:



Bei einem Drucker mit großem Speicher dauert das Busy auch nur μ s.



Alte Drucker drucken das Zeichen sofort, nachdem sie es empfangen.

Neue Drucker haben Speicher, meist mehrere Seiten.

Drucker können Fehler über die PE (Paper End) und ERROR Leitung melden.

Im Bios können verschiedene Modi ausgewählt werden:

SPP...Standard Parallel Port

EPP...Enhanced Parallel Port

ECP...Enhanced Capability Port

EPP hat auch Leitungen für Fehler wie Toner leer.

Bei ECP hat man in beide Richtungen einen großen Datentransfer.

Früher wurden an die Parallele auch ZIP Laufwerke angeschlossen.

Die Pegel für parallele Datenübertragung sind 0 und 5V.

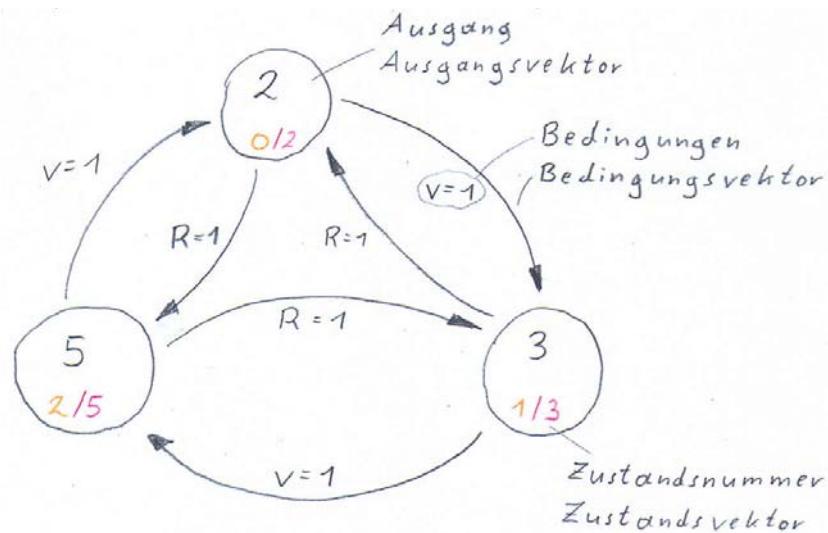
Werden viele gleiche Zeichen gesendet, werden diese als Befehle gesendet, um Zeit zu sparen („drucke 100 weiße Felder“).

Beim Messen mit dem Oszilloskop vergleicht man das Busy mit den anderen Leitungen. Es handelt sich dabei um einmalige Vorgänge, also entsprechend triggern.

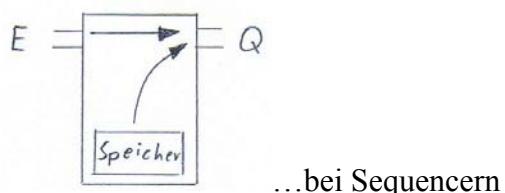
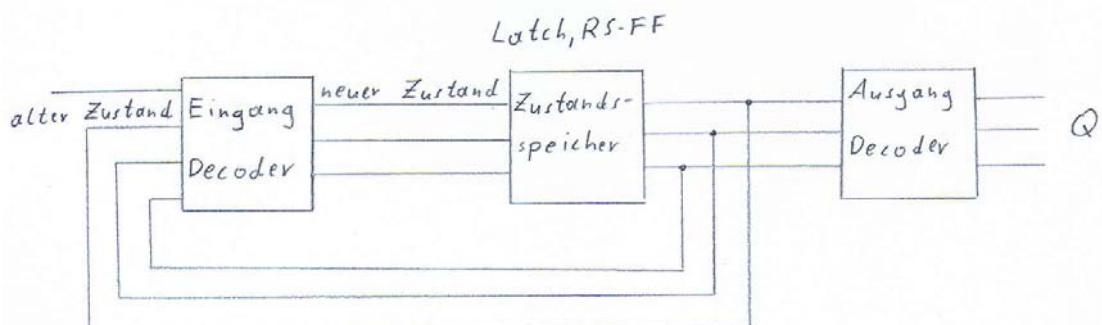
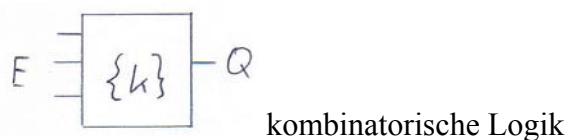
6 Sequencer

Sind Zähler, welche sich den Zustand auch bei einem Stromausfall merken. Sie werden z.B. bei Schrittschaltwerken oder Automaten benutzt.

Bsp.: Zustandsdiagramm eines Primzahlezählers



Dieses Zustandsdiagramm wird im Skriptum als Beispiel für den Aufbau benutzt.



Man kann auch die Zustandsnummer gleich der Ausgangsnummer machen.
typ. Beispiel: Zähler, $V = 1$, $Q = 1,2,3,4,\dots$

Man kann zwischen folgenden Bauarten unterscheiden:

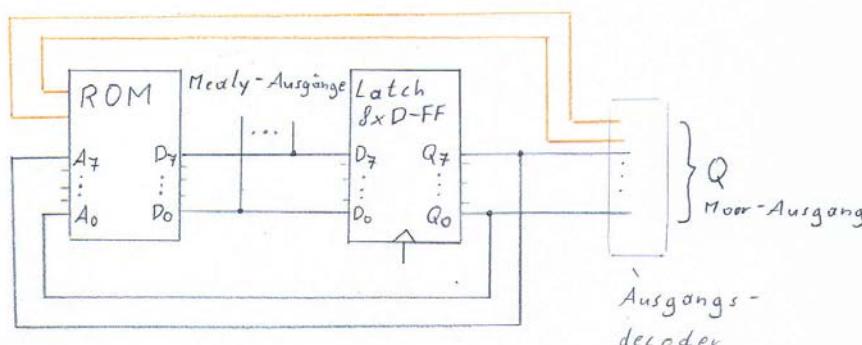
Hardware Sequencer

- ROM Sequencer
- JK – Sequencer

Software Sequencer

Beispiel für HW-Sequencer: Eingang in Modul, das die RS232 simuliert (μ -Controller)

6.1 ROM Sequencer



Die Mealy-Ausgänge sind taktunabhängig, sie ändern sich noch bevor sie im Latch gespeichert werden.

Die Moor-Ausgänge sind vom Takt abhängig.

Auch die Mealy-Ausgänge werden zum Ausgangsdecoder geführt (in Zeichnung nicht vorhanden).

Die orangen Leitungen sind die Steuerleitungen (Vor, Zurück, Ein, Aus, Notaus, ...)

Wahrheitstabelle für das Zustandsdiagramm:

				alter Zustand					neuer Zustand		
A5	A4	A3	A2	A1	A0	D7...D3	D2	D1	D0		
0	0	0	1	0	0	0...0	0	1	0		
0	1	0	1	0	0		1	0	1		
1	0	0	1	0	0		0	1	1		
1	1	0	1	0	0		0	1	0		
0	0	0	1	1	1		0	1	1		
0	1	0	1	1	1		0	1	0		
1	0	0	1	1	1		1	0	1		
1	1	0	1	1	1		0	1	1		
			alle anderen		0...0		0	1	0		

Erklärung:

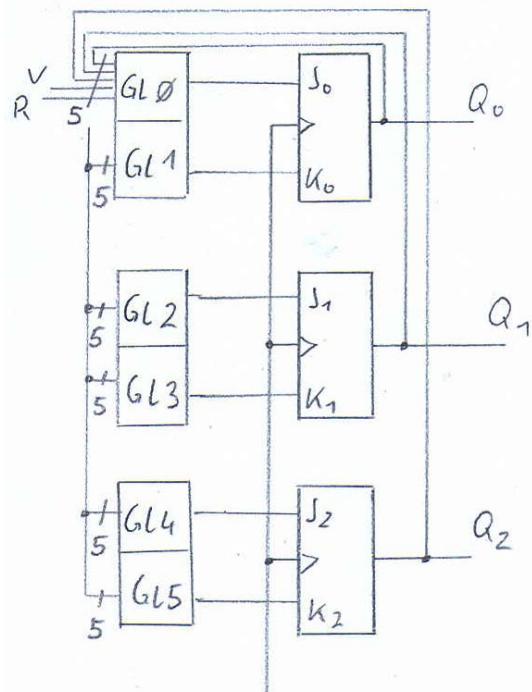
In der ersten Zeile ist der alte Zustand 2, da bei Tasten (v = A4, r = A3) nicht gedrückt sind, ist der neue Zustand auch 2.

In der zweiten Zeile ist der alte Zustand 2, da „r“ gedrückt wurde, ist der neue Zustand 5.

In der dritten Zeile ist der alte Zustand 2, da „v“ gedrückt wurde, ist der neue Zustand 3.

usw.

6.2 JK-Sequencer



Gl...Gleichungssystem

Als Eingänge für die Gls dient ein Bus mit 5 Leitungen (3 Eingänge, V , R)

Das ganze ist eine Moor Maschine (Takt mit JK-FF), um eine Mealy-Maschine daraus zu bauen, bräuchte man einen Decoder.

Jetzt braucht man keinen Decoder, um die Zahlen 2, 3 und 5 darzustellen. Mit Decoder könnte man sich das letzte JK-FF sparen.

Für die Entscheidung, ob man lieber mehr JK-FFs nimmt oder einen Ausgangsdecoder, ist die Größe der Zahlen und die Anzahl der Zustände wichtig.

Bsp.: größte Zahl 1 Million, nur 3 Zustände \rightarrow 20 FFs (um Million darzustellen), oder 2 FFs + Decoder

Die schwarzen Zustände stammen aus dem Zustandsdiagramm.
 Die orangen Zahlen sind die Übersetzung der schwarzen Zahlen mittels der JK-FF-Wahrheitstabelle (lange Wahrheitstabelle mit altem und neuem Q).
 X bedeutet, dass wahlweise 1 oder 0 an diesem Ausgang anliegen kann.

J_0	0	0	1	1	\vee
	0	1	0	1	R
0 1 0	0	1	1	0	
0 1 1	X	X	X	X	
1 0 1	X	X	X	X	
0 0 0	X	X	X	X	
0 0 1	X	X	X	X	
⋮					
1 1 1	X	X	X	X	

KV-Diagramm Vorbereitung

Aus dem vorigen Diagramm werden die Zustände für J_0 ausgelesen. —

Die restlichen Zustände sind egal (x). —

→ Weil sie im Sequencer nicht verwendet werden.

KV-Diagramm mäßig anordnen:

J_0	V	0	0	1	1
	R	0	1	1	0
0 1 0	0	1	0	1	
0 1 1	0	1	0	1	
1 0 1	0	1	0	1	

X als 1 verwenden → nur 1 in der Reihe → Q egal

Päckchen vereinfachen: —

$$S_0 = (R \& \bar{V}) \mid (\bar{R} \& V) = R \text{XOR} V$$

Es muss für jedes J und jedes K ein KV-Diagramm erstellt und ausgewertet werden.
Zur Übung wird noch die Formel für K_0 hergeleitet:

	V	0	0	1	1	V
$K_{0,R}$		0	1	0	1	R
0 1 0	X	X	X	X		
0 1 1	0	1	0	0		
1 0 1	0	0	1	0		

	V	0	0	1	1
$K_{0,R}$		0	-1	1	0
0 1 0	X	X	X	X	
0 1 1	0	1	0	0	
1 0 1	0	0	0	1	

$$K_0 = (R \& \bar{V} \& Q_0 \& Q_1 \& Q_2) | (\bar{R} \& V \& \bar{Q}_0 \& \bar{Q}_1 \& \bar{Q}_2)$$

Vereinfachen:

Man kann die X so wählen, dass sich ein Q-Zustand daraus ergibt —

Für X=1 ergibt sich dann:

$$K_0 = (R \& \bar{V} \& Q_1) | \dots$$

Auch das andere kann vereinfacht werden: —

Für X=0 ergibt sich dann

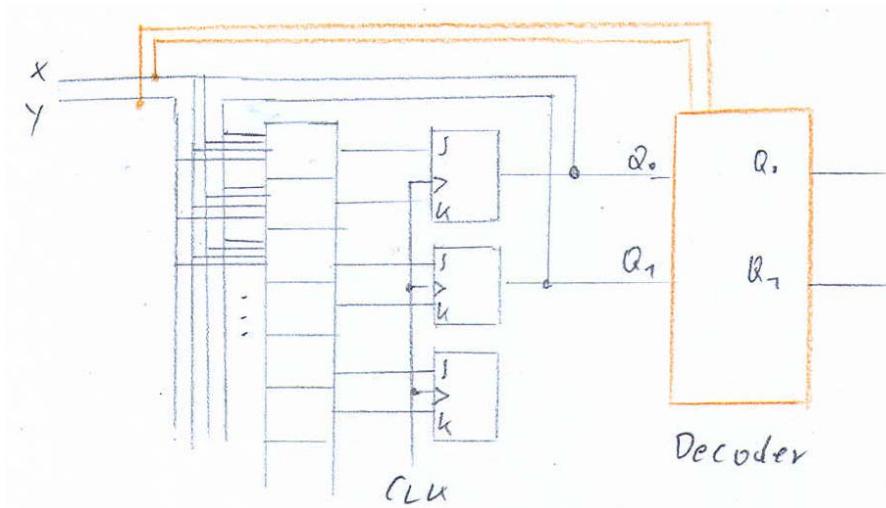
$$K_0 = \dots | (\bar{R} \& V \& Q_2)$$

	S_1	V	0	0	1	1
Q_2		R	0	1	0	1
0 1 0	X	X	X	X		
0 1 1	X	X	X	X		
1 0 1	0	1	1	0		

	J_1	V	0	0	1	1
Q_2		Q_1	0	1	1	0
0 1 0	X	X	X	X		
0 1 1	X	X	X	X		
1 0 1	0	1	1	0		

— X als 1 verwenden → lauter 1 in einer Reihe → Q egal

$$J_1 = R \text{ XOR } V$$



Eine Mealey-Maschine wird mit JK-FFs so realisiert. Der Ausgangsdecoder hat eine Tabelle gespeichert, die die neuen Werte ausgibt.

Man könnte auch die Werte vor den JK-FFs abgreifen, dabei würde aber der Aufwand zu sehr steigen. (Logik-Laufzeiten, nach JK-FFs mit Takt gesetzt)

6.3 Software Sequencer

Werden am leichtesten mit einer globalen Variable gemacht.

Man kann Software Sequencer starten und stoppen.

Das folgende Programm dient als Beispiel (nicht komplett oder korrekt ausprogrammiert)

```
#define ENDE ...
#define ANF ...
unsigned char hState; //Zustandsnummer

void main(void)
{
hState = ANF;
...
while(hState != ENDE)
{
if(hState == 1) { S0(); }
if(hState == 2) { S1(); }

...
/* auch als switch programmierbar
1: S0(); break;
2: S1(); break;

...
Vorteil: Es gibt einen default Zustand.
Beim if müsste man mit else if arbeiten.
Bei der if-Variante gibt es Bereichsabfragen.
→ if bietet die Möglichkeit sehr komplex zu arbeiten.
*/
}
}
```

```

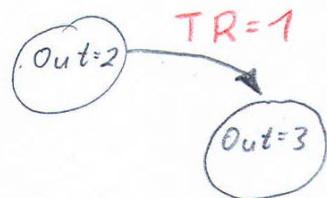
void S0(void)
{
Out = ...; //Port, globale Variable, oder ein anderer Ausgang
if(v == 1) { hState = 1; }
if(r == 1) { hState = 2; } //Jetzt wäre r dominant
}

```

Damit nicht ein Eingang dominant ist, muss man in der if-Abfrage auch die anderen Eingänge abfragen.

→ Der Effekt kann auch bewusst eingesetzt werden. (Not-Aus immer dominant)

Beim Software Sequencer kann man auch beim Übergang in einen anderen Zustand etwas machen.



Dies wird einfach zu der if-Abfrage ins Unterprogramm geschrieben.

Man versucht globale Variable zu vermeiden (lokale Variable und Return-Werte verwenden):

```

uc S0(uc Rx)
{
uc hS1;
hS1 = 0;
if ...
    return (hS1);
}

```

Im Main wird das dann so geändert:

```

main()
{
uc hHP;
uc R;
hHP = ANF;
...
while
{
    if(hHP == 0) { hHP = S0(R); }
    ...
}
}

```

Werden im Unterprogramm Abfragen aus dem main benötigt, nimmt man Übergabewerte. Man könnte auch Zeiger auf Funktionen verwenden.

Das Problem an Software-Sequencern ist, dass sie langsam sind (einige 100MHz werden schon schwierig).

7 EasyABEL (PLD Software von DATAIO)

Mit Abel kann man Gatter so programmieren, wie man will.

Das Programm wird parallel abgearbeitet – nicht sequentiell!

In der devices.txt steht eine Liste mit Geräten und dazugehörige Programmervorschriften.

Aufpassen, dass die definierten Pins mit der Hardware zusammenpassen (I, O, I/O).

Das Verhalten wird mit istype angegeben.

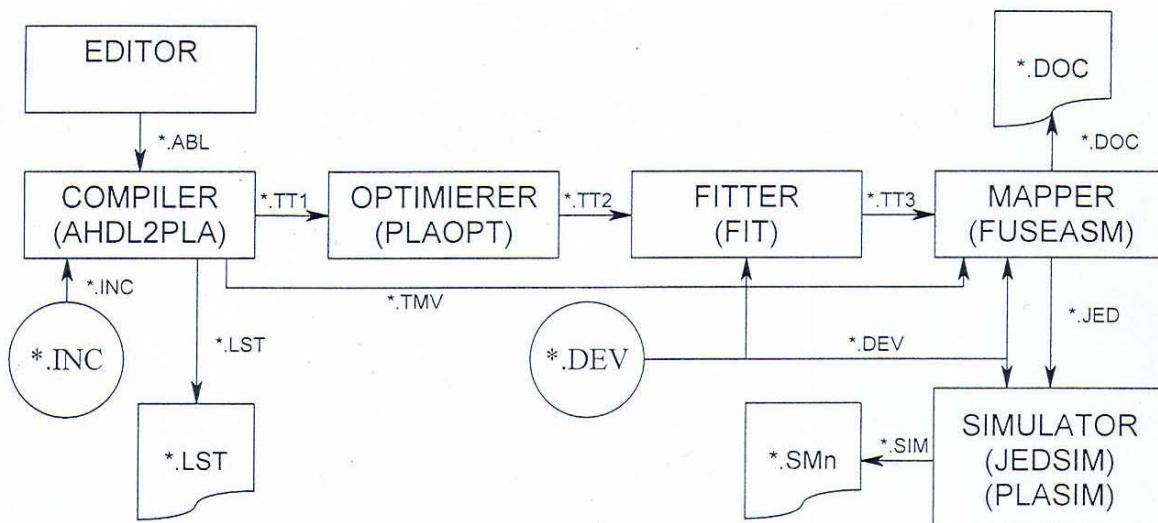
7.1 Programm Architektur

Abel Dateien müssen nach einem bestimmten Schema aufgebaut werden (Siehe Beispiel).

Da das Programm parallel abgearbeitet wird, ist es egal, ob man unter equations das & oder # zuerst schreibt.

Man kann auch eine truth_table verwenden.

Eine ASCII-Textdatei (*.ABL) mit Anweisungen in ABEL-HDL beschreibt die Funktion eines PLD. Diese Quelldatei wird in eine JEDEC-Datei für den PAL-Programmer umgewandelt. Ein einfacher Simulator erlaubt es erste Tests durchzuführen.



Editor...normaler Texteditor

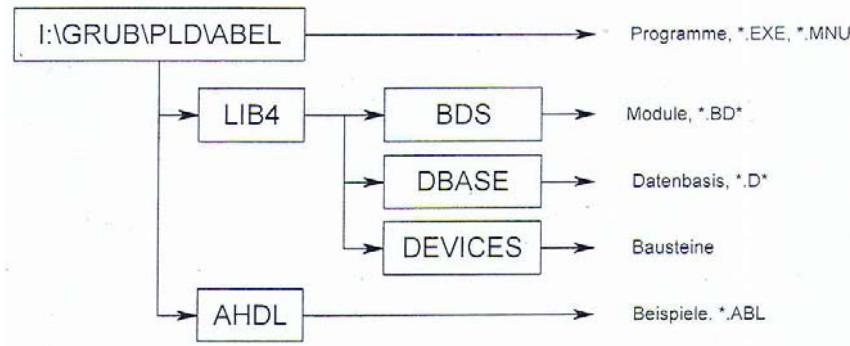
*.lst...vom Compiler erzeugte Fehlermeldung

Optimierer...löscht unnötige Zeilen, übersetzt Variablennamen

Fitter...passt den Code an das jeweilige Gerät an (mit Hilfe des *.dev-Files)

Mapper...wandelt den angepassten Code in die richtigen Dateien um (hex für Gerät, doc für Benutzer)

Die Programme finden sich unter C:\Programme\ABEL (Durch Batch-Datei in Labor erreichbar)



DOS Umgebungsvariable

In DOS müssen die Umgebungsvariable gesetzt werden. Siehe dazu auch z.B. W:\ABEL.BAT. Sollte ABEL unter C:\Program gespeichert sein, sind folgende Aufrufe nötig:

```

@echo off
set ABEL4DEV=C:\Program\Abel\Lib4
set ABEL4DB=C:\Program\Abel\Lib4\Devices
set DB_DICT=C:\Program\Abel\Lib4\Dbase
PATH C:\Program\Abel
echo on
  
```

Im Netzwerk muss beachtet werden, ABEL in dem Verzeichnis die Daten ablegt, aus dem es aufgerufen wird. Es ist daher ein Suchpfad einzurichten, oder über den vollen Pfad I:\Grug\PLD\...\Abel4 aufzurufen. Es ist zu beachten, dass F: nach dem ausloggen automatisch gelöscht wird.

7.2 Aufbau einer ABEL-Datei

Eine ABEL-Quellcodedatei besteht aus vier Blöcken:

1. dem **Dateikopf** mit dem Projektnamen und allgemeinen Bemerkungen
2. dem **Deklarationsteil**, in dem der verwendete Baustein, die Stiftbelegung und die Konstante festgelegt werden
3. der **Funktionsbeschreibung**, in der die interne Funktion des Bausteins definiert wird und
4. den **Testvektoren** zum Prüfen der unter 3. definierten Funktion.
5. Die Testvektoren können auch weggelassen werden.

Das Programm wird mit Hilfe von **Schlüsselwörtern** gesteuert. Diese können groß oder klein geschrieben werden. Alle anderen Bezeichner unterscheiden jedoch Groß- und Kleinschreibung (wie in der Programmiersprache „C“). Jeder Befehl endet mit einem Strichpunkt (;). Kommentare beginnen mit einem doppelten Hochkomma (,,).

Schreibt man einen Ausgang mehrmals in eine Formel, werden diese automatisch ODER-Verknüpft (keine Compilerfehlermeldung).

7.3 Beispiel einer ABEL-Datei

“ Dateikopf:

module *BSP1* “Name des Projektes. Er steht wieder bei ‚end‘
title ’einfaches Beispiel für UND und ODER Verknüpfung mit GAL GAL16V8‘
“Deklarationsteil
declarations “ wie verhält sich der Bauteil
BSP1 **device** ‘P16V8’; “ = GAL16V8 (Chipstruktur festlegen)
I0, I1, I2 **pin** 2,3,4; “ Input Pins
OU, OO **pin** 19, 17; ”Output Pins
Qu, Qo **pin** 15, 14 istype ‘com’; “Output Pins kombinatorische Logik
L, H, X, Z = 0, 1,..X,..Z; “ Definition privater Namen (wie defines in C)
“Funktionsbeschreibung
equations
OU = I0 & I1 & I2; “UND-Verknüpfung
OO = I0 # I1 # I2; “ODER-Verknüpfung
“Ein Beispiel für eine Wahrheitstabelle
truth_table
[I1,I0] -> *[Qu, Qo]* “Kopfzeile, bestimmt welche ein und Ausgänge
“betroffen sind
[0,0] -> *[0, 0];* “Bits reinschreiben
1 -> *[L, H];* “Zahlen als binär vorstellen
[H,L] -> *2;* “H und L wurden oben definiert
3 -> *3;*
“Testvektoren
test_vectors oder nur 1 Ausgang
([I0,I1,I2] -> *[OU,OO]* (*[I0,I1,I2]* -> *OU*) “kein Strichpunkt
[L,L,L] -> *[L,L];* *[L,L,L]* -> *L;*
[L,H,L] -> *[L,H];* *[L,H,L]* -> *L;*
[H,L,H] -> *[L,H];* *[H,L,H]* -> *L;*
[H,H,H] -> *[H,H];* *[H,H,H]* -> *H;*
end *BSP1* “Ende der ABEL-Datei, der Name muss mit dem bei **module** übereinstimmen

Auf **title** folgt ein Titel oder eine kurze, eventuell auch Beschreibung (auch in mehreren Zeilen möglich) der Funktion. Nach **module** kann **options** mit Steueranweisungen für den Compiler folgen. Vor **device** steht der Name der *.JED (JEDEC) Datei, dahinter der Name des PLD Bausteins in der Bibliothek. Es ist empfehlenswert das Projekt und die JEDEC Datei gleich zu benennen. Anstatt, oder ergänzend zu **equations** kann die Funktion des Bausteins auch mit **truth_table** in Form einer Wahrheitstabelle festgelegt werden. Die **test_vectors** dienen zum simulieren und überprüfen von einfachen bis komplexen Gleichungen.

7.4 Die ABEL HDL Sprache

Erlaubte Zahlensysteme sind:

b = binär, h = hexadezimal, o = oktal, d = dezimal.

Die Voreinstellung ist dezimal.

Zeichenketten (Strings) werden durch ‘ ’ eingeschlossen: ’Ein Text‘;
Vektoren werden durch [] beschrieben.

z.B.: Adr4 = [A3,A2,A1,A0]; und Adr4 = [0,1,0,1] oder Adr16 = [A15..A0];
 Zu beachten ist, dass A0 das LSB (Least significant Bit) und A15 das MSB ist.
 Blöcke werden durch {} zusammengefasst.

Statische (kombinatorische) Zuweisungen erfolgen mit =
 Registrierte (taktgesteuerte) Zuweisungen erfolgen mit :=
 Zuweisungen in Wahrheitstabellen und Testvektoren erfolgen mit -> (:-> für taktgesteuert)

Erlaubte Operatoren:

logische Operatoren: ! = NOT, & = AND, # = OR, \$ = XOR, !\$ = XNOR
 mathem. Operatoren: + - * / % >> << und das unäre - = Negation (2er Komplement).
 Vergleichsoperatoren: == != <= >= < >

Einige spezielle Konstante sind vordefiniert:

.X. = don't care, .Z. = tristate, .F. = floating Input, .P. = preload Register,
 .U. = Clock mit pos. Flanke, .D. = Clock mit neg. Flanke, .K.,.C. = Clock mit bel.
 Flanke

Untereinander geschriebene Ausdrücke entsprechen einer ODER-Verknüpfung.
 Durch Attribute sind Signale genauer definierbar. Nach dem Signal Pin folgt das
 Schlüsselwort istype und danach das Attribut als String.

z.B.: q0,q1 pin 14,15 istype 'reg,invert';

Definierte Attribute:

'com'	das Signal ist kombinatorisch
'butter'	Flip-Flop Ausgang geht direkt zum Pin
'invert'	Flip-Flop Ausgang geht über Inverter zum Pin
'neg'	Komplement bilden vor der Verarbeitung
'reg_D'	ein D-Flip-Flop
'reg_JK'	ein JK-Flip-Flop
'reg_SR'	ein RS-Flip-Flop

Zur genaueren Beschreibung von internen Signalen können Punkt-Qualifizierer verwendet werden:

z.B.: [q0,q1].CLK = Clock;

Definierte Qualifizierer:

.CLK, .OE, .PIN, .FB = Clock für Flip-Flop, Output Enable, Feedback Pin, Feedback Register
 .D, .J, .K, .S, .R = die entsprechenden Flip-Flop Eingänge

.Re, .PR, .AR, .AP = synchroner Reset und Preset, asynchroner Reset und Preset

Steuerbefehle:

```

  goto  goto 7;      " gehe zum Zustand 7
  if – then – else
    if (Adr16 < ^h04F3) then State4
    else
      if (Adr16 > ^h10C0.) then State0
      else State 2;
  case – endcase,
    case a == 0 : 4;
    a == 4 : 0;
  endcase

```

when – then – else when B then A = B else A = C;
with – endwith wird an if, case oder goto angehängt, um mit einem bestimmten Ausgangszustand zusätzliche Variable mitzusteuern.
sowie macro. @message, property, trace, node, flag,usw. – siehe Handbuch.
state_diagramm Leitet einen Block für ein Schrittschaltwerk ein, die einzelnen Zustände werden mit state eingeleitet

```
state_diagramm SchiebeReg  
state A : if (Reset) then A else B;  
state B : if (Reset) then A else C;  
state C : if (Reset) then A else A;
```

Ob man die Tabelle mit High oder Low, Zahlen oder Werten aufbaut, ist für die Geschwindigkeit nicht relevant (Compiler rechnet um).

→ Das Beispiel dient nur der Veranschaulichung, in der Praxis nicht die Arten mischen.

Achtung:

Der Compiler muss nachher eine Schaltung aus der Tabelle machen – Alle Zustände müssen eingegeben werden.

Testvektoren müssen nicht geschrieben werden. Sie werden nur zur Fehlersuche angelegt.

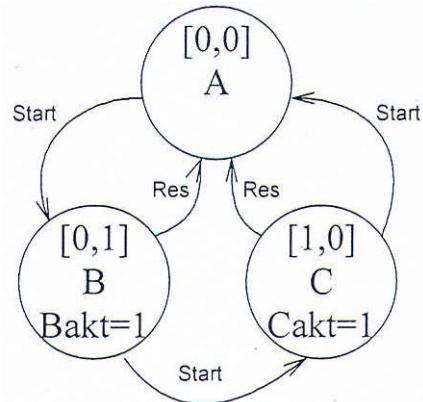
→ Man muss auch nicht alle Ausgänge testen.

Der Name der Datei muss bei **module**, **device** und **end** vorkommen.

Man kann auch Ausgänge invertieren (auch wenn es etwas ungewohnt wirkt): !Q = I0;

7.5 Beispiel für ein Schrittschaltwerk

Ein Sequencer wird meist durch sein Zustandsdiagramm beschrieben:

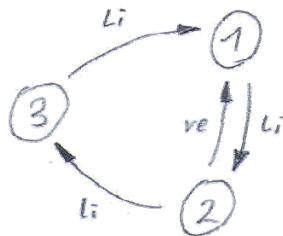


ABEL-Datei:

```

module Automat1 "Name des Moduls"
title 'Sequenzierer mit den Zuständen A=0 -> B=1 -> C=2.
Bei Start = 1 wird mit der Clock zum nächsten Zustand geschaltet A->B->C
->A->...
Reset führt stets zum Zustand A.
Bakt und Cakt melden den entsprechenden Zustand nach außen.'
" Die Zustandsregister, die Zustands- und Ausgangsvektoren sowie die
" Übergangsbedingungen.:
declarations
Automat1 device 'p16r4' " Signetics 16R4
          Q0, Q1 pin 14, 15 istype 'reg'; " Zustandsregister
          clock pin 1; " globaler Takt
          Start pin 4; " Start (Reset) Taste
          Reset pin 3; " Start (Reset) Taste
          Bakt, Cakt pin 12, 13 istype 'com'; " kombinatorische Ausgänge
          Status = [Q1, Q0] " Statusvektor definieren
          A = 0; B = 1; C = 2; " Zustandswerte für Status
equations
          Status.CLK = clock; " Förschaltbedingung für Statusvektor
state_diagram Statusdiagramm
          state A:
          Bakt = 0; Cakt = 0; " Bakt und Cakt sind L in diesem Zustand
          if (Start) then B;
          state B:
          Bakt = 1; " Bakt wird H
          if (Reset) then A " Fortschaltbedingung
          else if (Start) then B
          state C:
          Cakt = 1; " Cakt wird H
          goto A; " Fortschaltbedingung
end Automat1
  
```

7.6 Sequencer Beispiel



```

module BSP1
title 'Abel Sequencer'
declarations
BSP1 device 'P16V8';
LI pin 8; "richtige Nummer aus Datenblatt
RE pin 9;
Q0, Q1 pin 12, 13 istype 'reg'; "reg für sequentielle Logik
Clock pin 16; "Für Moor-Maschine braucht man Takt
equations
Q0.CLOCK = Clock; "Moor ist Clock abhängig
Q1.CLOCK = Clock;
state_diagram [Q0,Q1]
state S1:
Q0 = 0;
Q1 = 1;
if(LI & !RE) then S2;
state S2:
Q0 = 1; "Mit Vektoren könnte man [Q0,Q1] = 2; schreiben
Q1 = 0;
if(LI & !RE) then S3;
if(!LI & RE) then S1;
state S3:
Q0 = 1;
Q1 = 1;
if(LI & !RE) then S1;
end BSP1;
  
```

Es ist vorteilhaft Vektoren zu benutzen, da man so einfach die dezimalen Werte den Ausgängen zuweisen kann.

Werden einige Ausgänge hardwired angelegt (unveränderlich), muss dies in Abel berücksichtigt werden.

	Q4	Q3	Q2	Q1	Im Abel
	X	0	0	X	
0 →	0			0	0 0 → 0
1 →	0			1	0 1 → 1
16 →	1			0	1 0 → 2
17 →	1			1	1 1 → 3
hardwired					

7.7 Beispiel 4 Bit Multiplikator

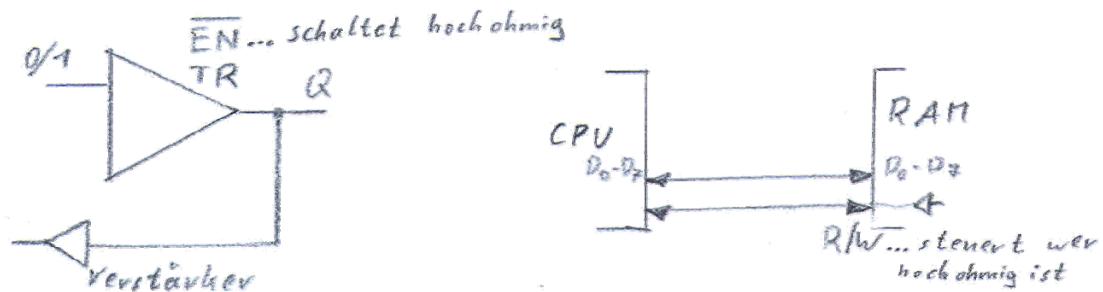
4 Bit Multiplikation → Taktsynchron

```
title...
module BSP_TEST
declarations
BSP_TEST device 'P20V10';
A0, A1, A2, A3 pin ...;
B0, B1, B2, B3 pin ...;
Q0, Q1, ... Q7 pin istype 'reg'...; "reg bedeutet, taktgesteuert
CLK1 pin 1; "laut Datenblatt, meistens ist CLK pin 1
VekA = [A3, A2, A1, A0]; "LSB muss hinten sitzen
VekB = [B3, B2, B1, B0];
VekQ = [Q7...Q0]; "8 Bit reichen für Multiplikation von zwei 4 Bit zahlen
equations
VekQ = VekA * VekB;
VekQ.CLK = CLK1;
end BSP_TEST
```

Bussysteme

Bussysteme können mit Tristate oder Wired Logic realisiert werden.

7.8 Tristate



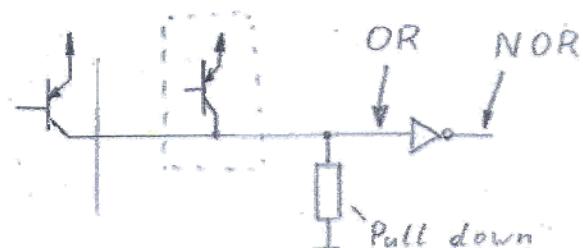
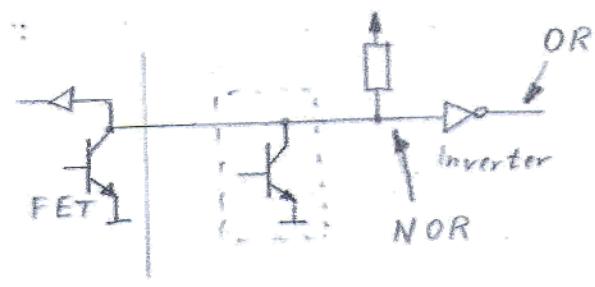
Damit ein Kommunikationspartner (in dem Fall CPU und RAM) schreiben kann, müssen alle anderen hochohmig sein.

Bei einem Bussystem mit Tristate gibt es also 3 Zustände (0, 1, hochohmig).

E	TR;	Q
0	0	0 ... 0
0	1	1 ... 1
X	1	
X	1	Z ... hochohmig

7.9 Wired Logic

Bussysteme können auch mit Wired Logic realisiert werden (z.B. I²C).



Durch Auswechseln von NPN und PNP Transistoren und dem Vorschalten von Invertern können alle Logikbausteine erzeugt werden.

8 Kundenprogrammierbare Logikbausteine

Werden in 2 große Gruppen eingeteilt:

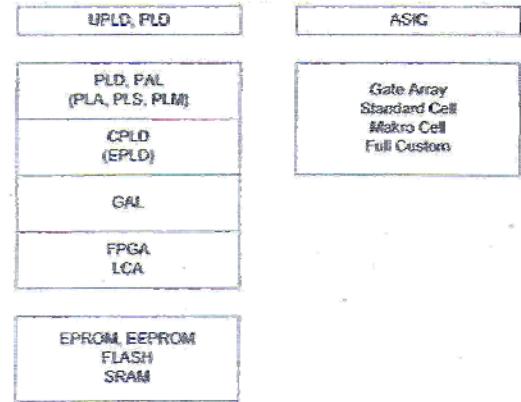
UPLD, PLD (User Programmable Logic, Device)

Werden beim Kunden programmiert. Preis ca. 1-10 € /Stück (Großhandelsmengen),
100 – 20.000 GÄ (Gatter Äquivalent), für den Ersatz von TTL-Gattern,
Kombinatorischer Logik, kleinen digitalen Designs, kaum Initialkosten (nur Software
und Programmer) wird wie ein PROM programmiert. Meist JEDEC (*.jed) Datei.

ASIC (Application Specific Integrated Circuit)

Werden beim Hersteller programmiert. Eigene Firma, ev. Joint Venture, große
Stückzahlen. Sind teuer (vergleichbar mit kleinem LKW).

PAL	Programmable Array Logic
PLD=PLA	Programmable Logic Device, Programmable Logic Array
PLS	Programmable Logic Sequencer
PLM	Programmable Logic Macrocell
GAL	Generic Array Logic
CPLD	Complex Programmable Logic Device
EPLD	Erasable Programmable Logic Device
FPGA	Field Programmable Gate Array
LCA	Logic Cell Array
Memory	Flash-EPROM, EEPROM, ...



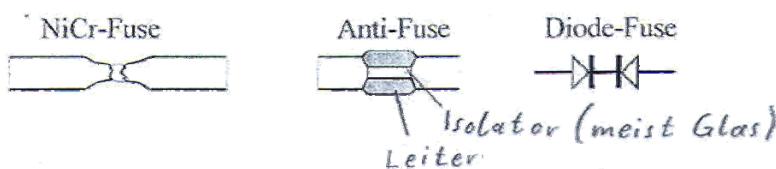
PLDs, PALs und GALs gehören von der Größe her zusammen. Sie brauchen alle viel Strom (50 – 100mA).

PLA, PLS, PLM sind ausgestorben

CPLD, FPGA...an die 32 – 500 Pins (~ 50000 Gatteräquivalente)

8.1 Sicherungen

Es gibt OTP (one time programmable) oder löschbare (EEPROM, Flash).



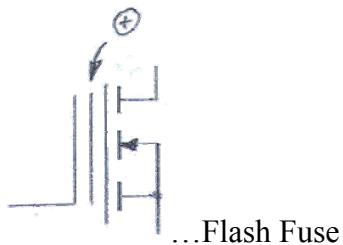
Die Verbindungen im Speicher werden über Sicherungen oder Speicher gemacht.

→ Sicherungen können nicht einmal zerstört werden (OTP).

Anti Fuse werden durchs programmieren geschlossen (Lichtblitz über Isolator macht diesen leitbar (Materialtransport)) → sehr klein.

Bei der Diodensicherung kann man die Leitung auch nur in eine Richtung leitend machen (natürlich auch in beide).

Weil man von Ni-Cr-Fuses ausgegangen ist, baut man auch neue Logiken so auf, dass standardmäßig FF drinnen steht. Durch leitend machen oder unterbrechen wird aus 1 eine 0.



Zwischen Gate und Source werden positive oder negative Teilchen geschossen, es verändert sich die U_{GS} .

→ Entweder durchschaltbar oder U_{GS} zu groß, um durchzuschalten.

Der genaue Aufbau kann variieren (selbstleitend / sperrend, pos./neg. Teilchen).

Zum löschen braucht man nur die Elektronen abziehen.

→ Siehe auch EPROM (mit UV-Licht löschbar).

8.2 Software

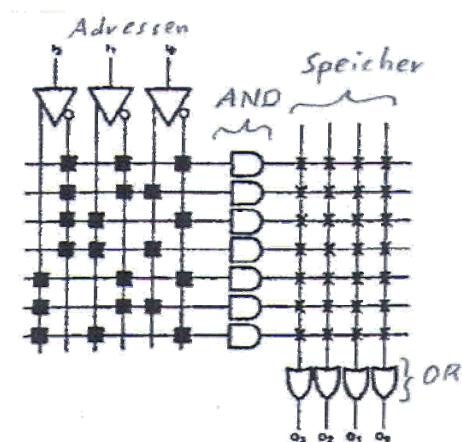
Mit einer Art PLD-Assembler können JEDEC-Dateien erzeugt werden.

Für die Eingabe logischer Gleichungen verwendet man ABEL, CUPL, PALASM, OPAL, ...

$$Y = x_1 \& x_2 \# !x_3$$

Die Eingabe der Funktion kann auch über Schaltungen oder Zustandsdiagramme oder andere Programmiersprachen wie VHDL oder VERILOG erfolgen.

8.3 ROM (Read Only Memory)



Abhängig von der Adresse wird immer ein &-Gatter 1. Die linke Seite wird fix vom Hersteller gemacht (UND-Matrix) → Für jede Adresse muss ein Speicher da sein.

Die Sicherungen rechts sind programmierbar.

Wenn die Verbindung leitet, liegt am Ausgang eine 1.

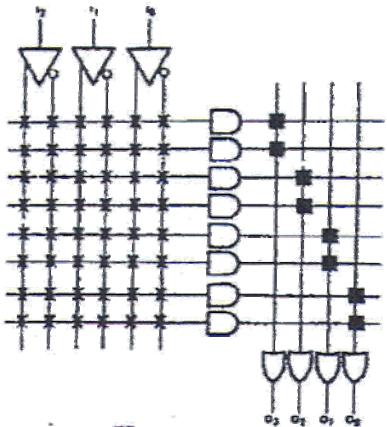
Typisch gibt es 1, 4, 8 oder 16 Speicherleitungen.

Ist eine Sicherung der aktuell angewählten Adresse nicht leitend, folgt eine 0.

Legt man eine Adresse an, erhält man alle 8 (4, 16...) Ausgänge auf einmal.

$[D_0 \dots D_x] = F(Adr)$, man will aber $Q_x = F(\text{Input})$.

8.4 PAL (Programmable Array Logic)

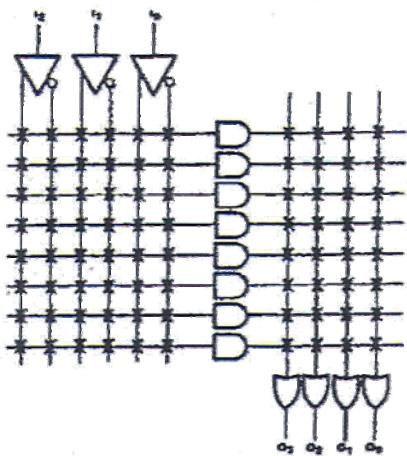


Die linke Seite ist programmierbar. Die Ausgänge sind fix, die dazugehörigen Eingänge kann man programmieren.

2 Ausdrücke sind rechts ODER-verknüpft → nur 2 Terme möglich.

Meistens kann man in Bausteinen 5-8 ODER-Verknüpfungen erstellen.

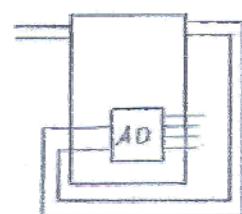
8.5 PLD (Programmable Logic Device)



Beim PLD kann man deshalb alles programmieren (darf aber trotzdem nicht beliebig viele Verknüpfungen benutzen - Speichergröße).

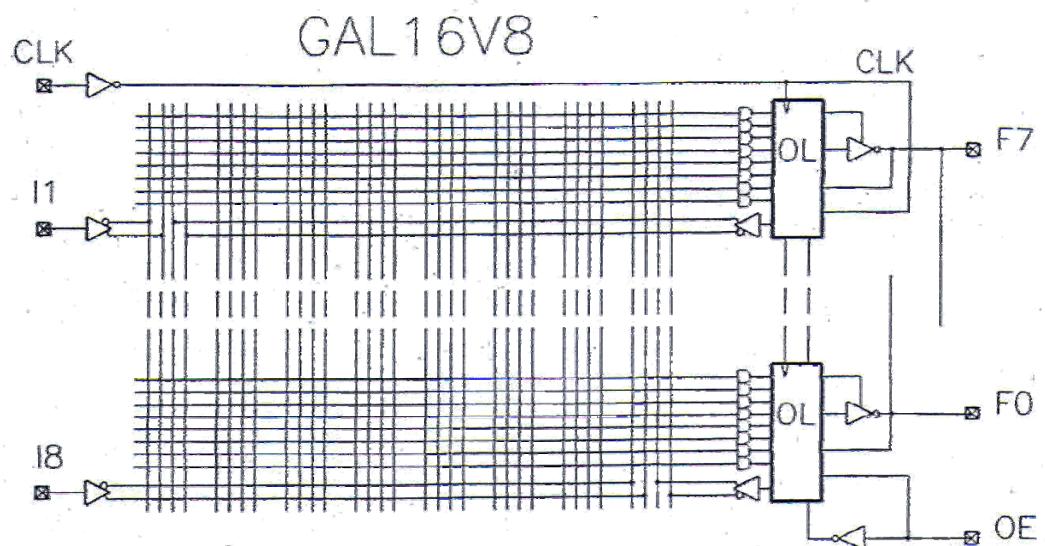
Mealey mit PLD

Für Mealey müssen die Ausgänge nach dem FF (bei PLD ein D-FF) an einen Decoder gehängt werden (siehe Mealey mit JK-FFs), für den Decoder kann gleich die PLD selbst verwendet werden.

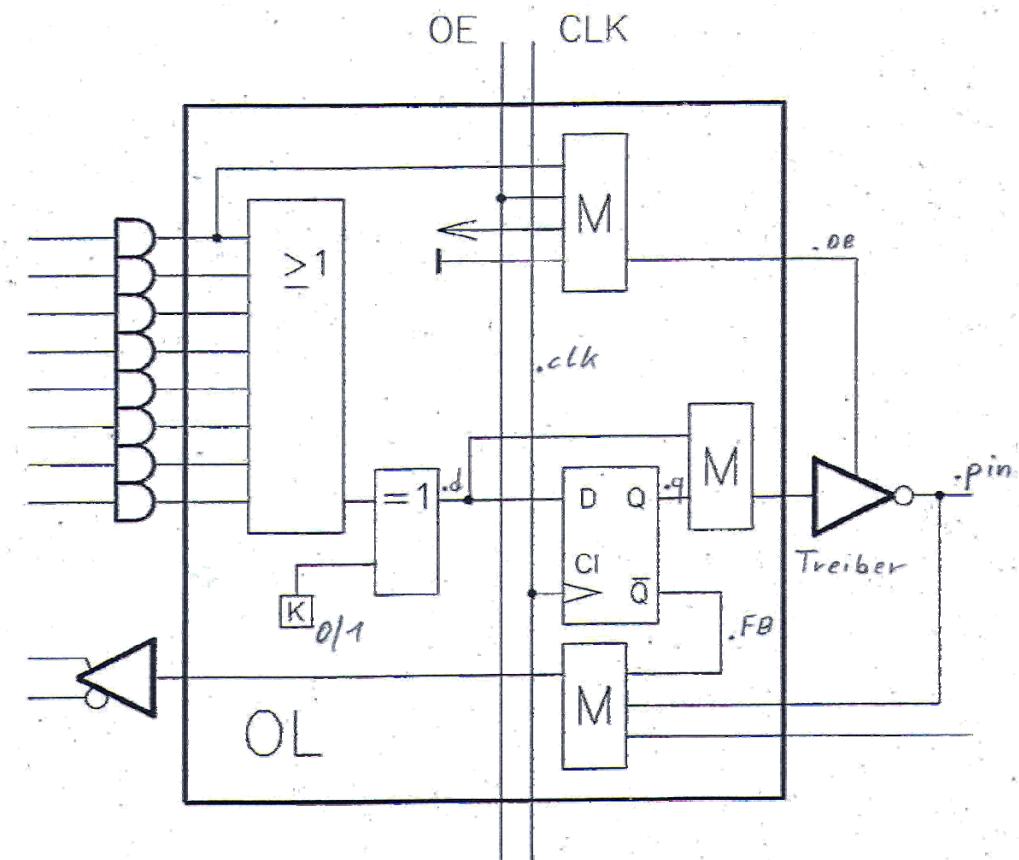


Hat man z.B. 20 Leitungen, könnte man 19 ODER-Verknüpfungen erstellen. Diese können auf beliebig viele Terme aufgeteilt werden. In der Praxis gibt es keine Beschränkung mehr.

8.6 GAL (Generic Array Logic)



Nach den ODER-Steinen wird noch eine Logik geschaltet:



K...Merker (mit 0 oder 1 programmierbar)

Danach kommt ein XOR \rightarrow invertiert Ausgang, wenn K = 1.

K		
0	1	0
0	1	1
1	0	1
1	1	0

Danach kommt ein D-FF und ein Multiplexer. Dieser kann (wieder mit Merker) das D-FF kurzschließen (Umschalter zwischen sequentieller und kombinatorischer Logik).

Anschließend kommt ein Tristate-Treiber (einschaltbar mit oberem Multiplexer). Die Leitung OE ist Output Enable, damit kann man alle Ausgänge ausschalten (ins GAL schreiben).

Der vierte Zustand vom Multiplexer führt zu einer Gleichung.

Die Buchstaben, welche in ABEL benutzt werden sind dazugeschrieben (.d, .oe,...).

Der untere Multiplexer dient dazu, um den Ausgangspin in die Eingangsmatrix einzuspeisen (jeder Ausgangspin kann auch als Eingang benutzt werden).

Um die Flexibilität zu erhöhen, ist an den unteren Multiplexer eine andere Output Logic OL angehängt.

Die Q-nicht Leitung ist deshalb zum unteren Multiplexer geführt, um den Zustand auch rückkoppeln zu können, ohne den Ausgang benutzen zu müssen.

Der Clock schaltet das D-FF. Er ist ein vordefinierter Eingangspin.

Er kann nicht als Gleichung geschrieben werden (ABEL kann es, aber der Baustein nicht).

Achtung: .pin und .fb können nicht gleichzeitig benutzt werden (nur eine Rückleitung).

Nominalgatur des GALs

GAL 16V8 -5

16...Aus und Eingänge

8...davon maximal 8 Ausgänge

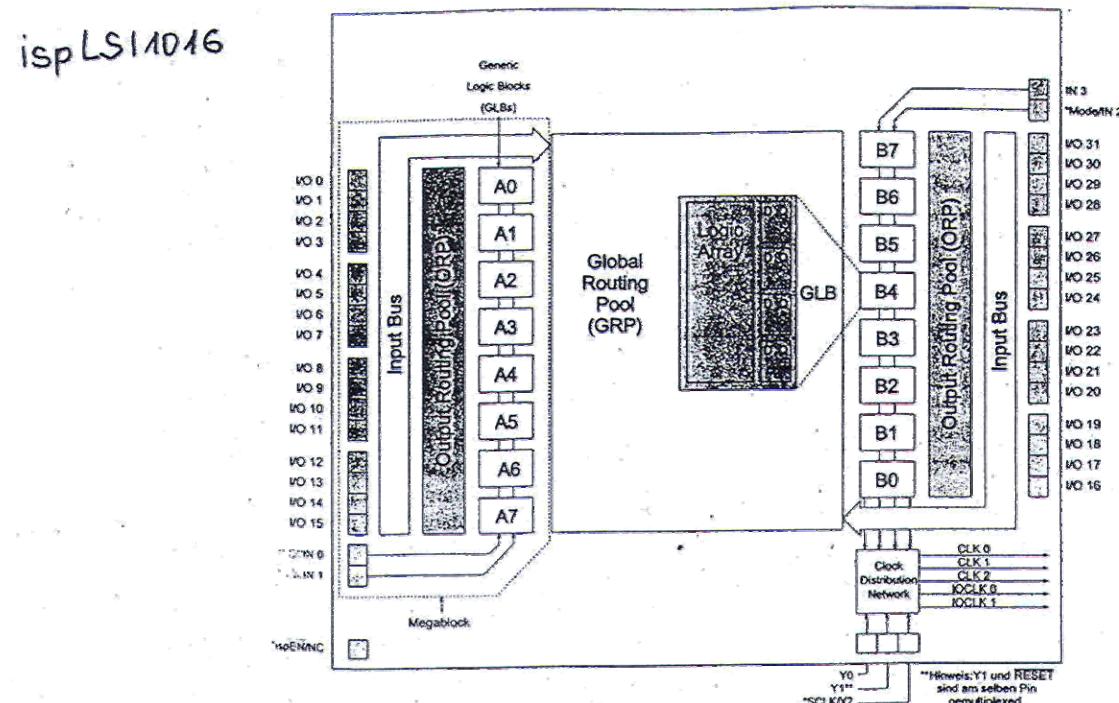
V...Zellenart der OL (Die V wurde in diesem Skript erklärt)

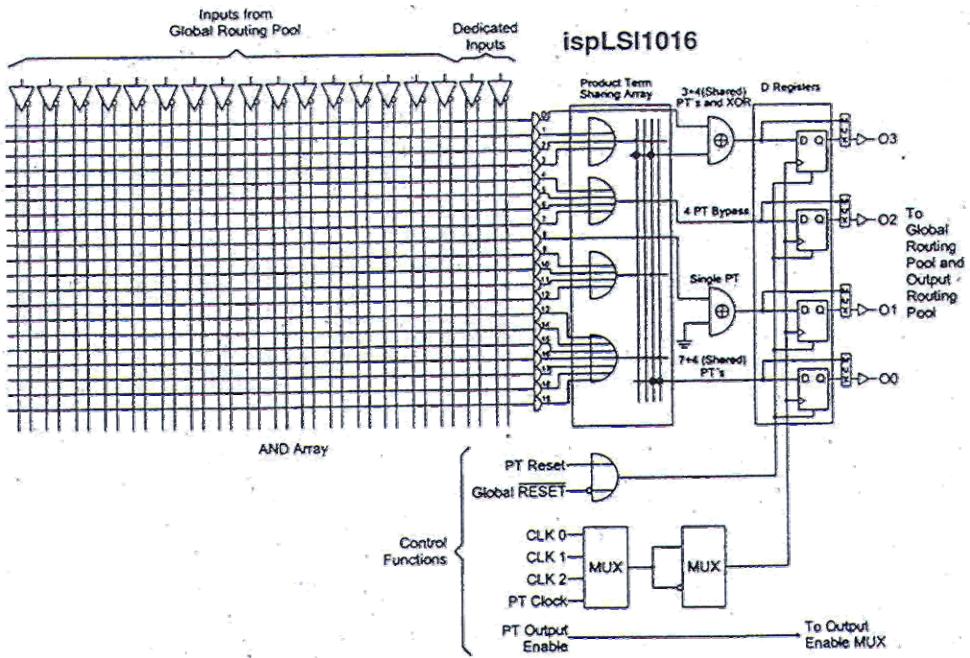
5...Geschwindigkeit vom Ein- zum Ausgang (Angabe in ns oder als Zahl – Datenblatt)

Kosten des Gals liegen im €-Bereich, sie sind wie µCs zu programmieren.

Ausgeführt sind sie bis 32 Pins.

8.7 CPLD (Complex Programmable Logic Device)





Die CPLD leiten sich vom GAL ab, es wird hier am Beispiel des ispLSI1016 besprochen.

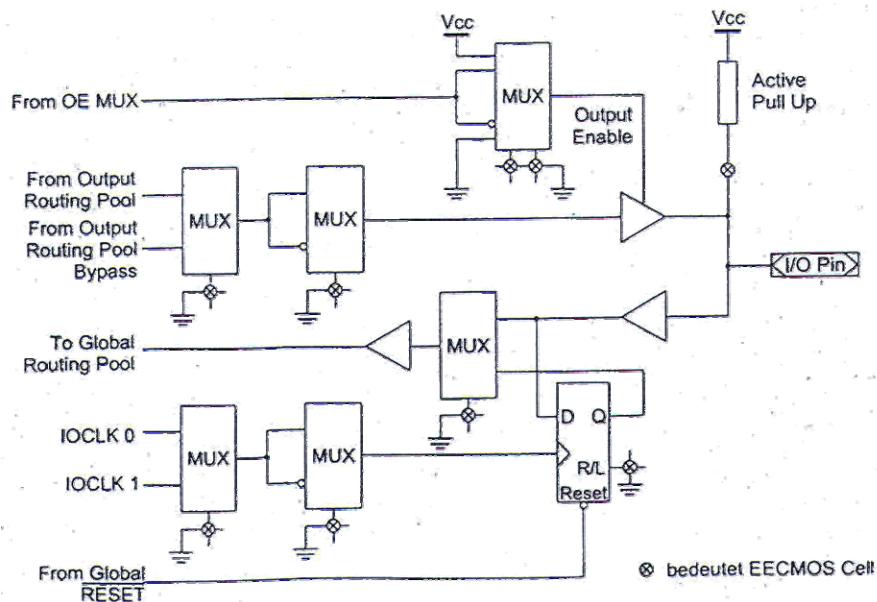
Beim CPLD sind alle Eingänge als Ausgänge verwendbar (nur I/O).

Typisch sind 80 – 500 pins.

Es gibt noch zusätzliche Elemente:

- Ausgangszelle unterteilt
Logik – Output Routing Pool – Logik
Der Output Routing Pool (ORP) wird benötigt, um die Zellen mit dem Global Routing Pool (GRP) zu verbinden
- In den Logiken (unteres Bild) gibt es ODER-Gatter mit 3, 2*4 und 7 Eingängen → max. 3, 4 oder 7 Terme möglich. (Durchschnittliche Termlänge)
- Im ORP werden die Ausgänge mit den ODER-Gattern verbunden, bzw. in den GRP rückgekoppelt.
- Die Elemente mit dem + sind XOR-Gatter (schaltbare Inverter)
- ganz rechts sind D-FFs mit Takt, dahinter wieder Multiplexer zum Ausschalten (kombinatorisch / sequentiell)
- Beim Takt gibt es 4 verschiedene Möglichkeiten (beim GAL nur einen), außerdem gibt es Reset-Leitungen. Mit dem zweiten Multiplexer beim Clock kann man positive oder negative Flanke einstellen.

8.7.1 GLB (Generic Logic Block)



Der Multiplexer bei GND und VCC schaltet den tristate um. Die dritte Leitung (from OE MUX) ist programmierbar.

Der Multiplexer in zweiter Reihe schaltet den Ausgang durch (mit schaltbarem Inverter). Der Bypass kommt direkt vom GRP.

Eingangsseite:

Verstärker ins Innere, D-FF und Multiplexer (überbrückt wieder D-FF).

Die Multiplexer unten hängen Takt 1 oder 2 an (mit Inverter), außerdem gibt es einen Reset.

→ Das FF im Eingang gab es im GAL nicht.

Der Pull-Up mit Kreuz ist programmierbar.

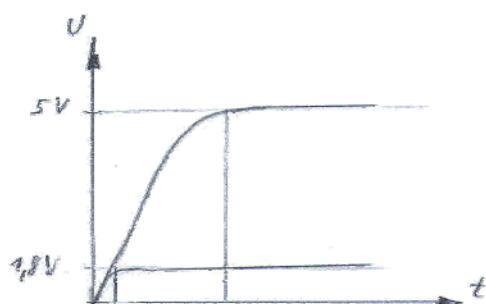
Der Preis von CPLDs liegt bei 10-50€.

Es gibt sie programmierbar oder mit SRAM.

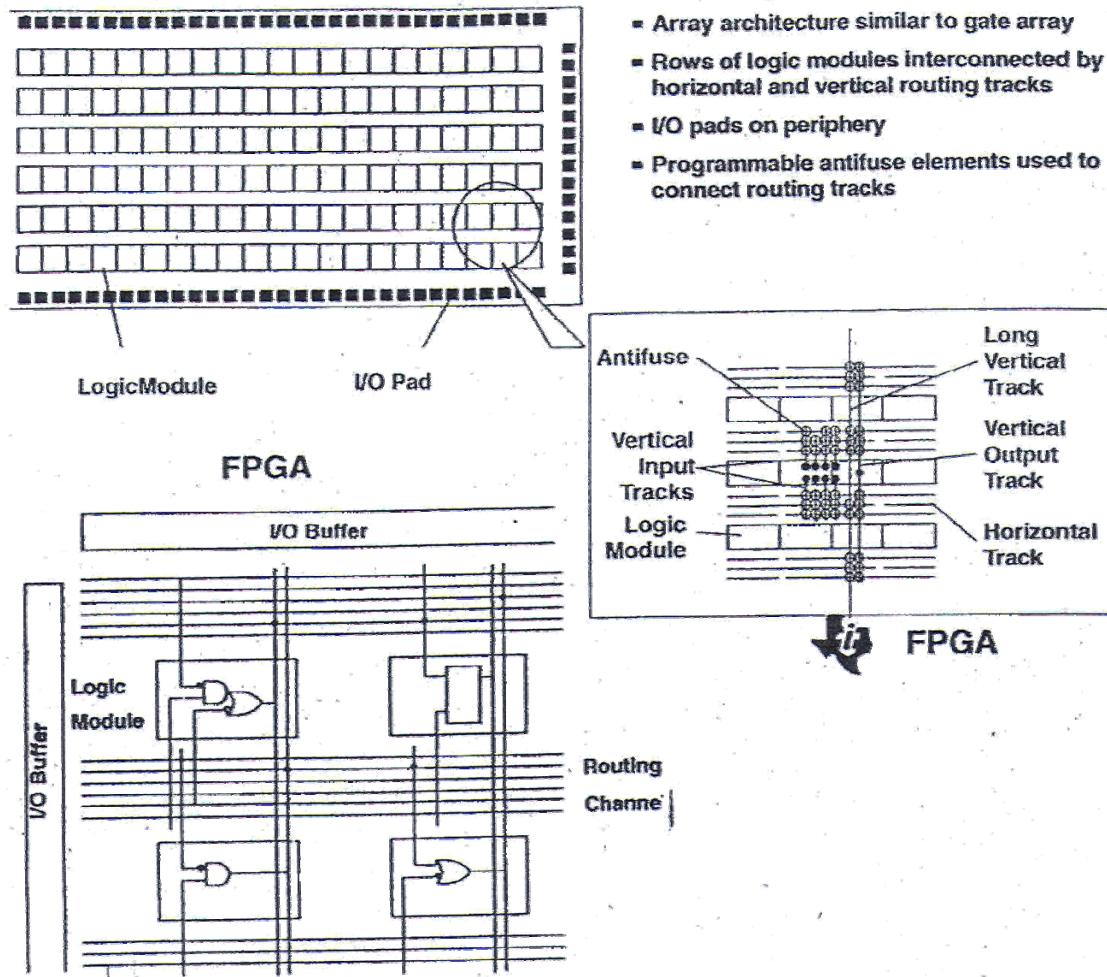
Bei SRAM sitzt neben dem CPLD ein E²PROM, das es beim Einschalten programmiert. → Vorteil: mehr Platz im CPLD.

CPLDs haben meistens 2 Spannungen (eine für den CORE-Bereich und eine für den I/O-Bereich).

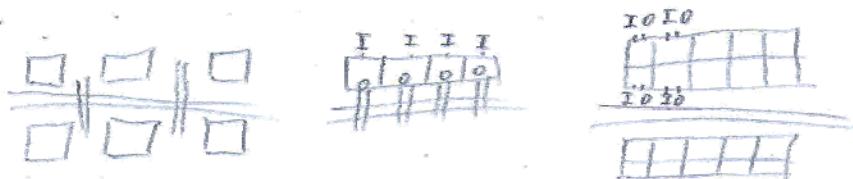
Die Core-Spannung ist niedriger (Strom-, Leistungs- und Wärmeverbrauch, außerdem Geschwindigkeit).



8.8 FPGA (Field Programmable Gate Array)



Die Speicher sind in Zellen aufgeteilt, zwischen denen geroutet wird.
3 Standards:

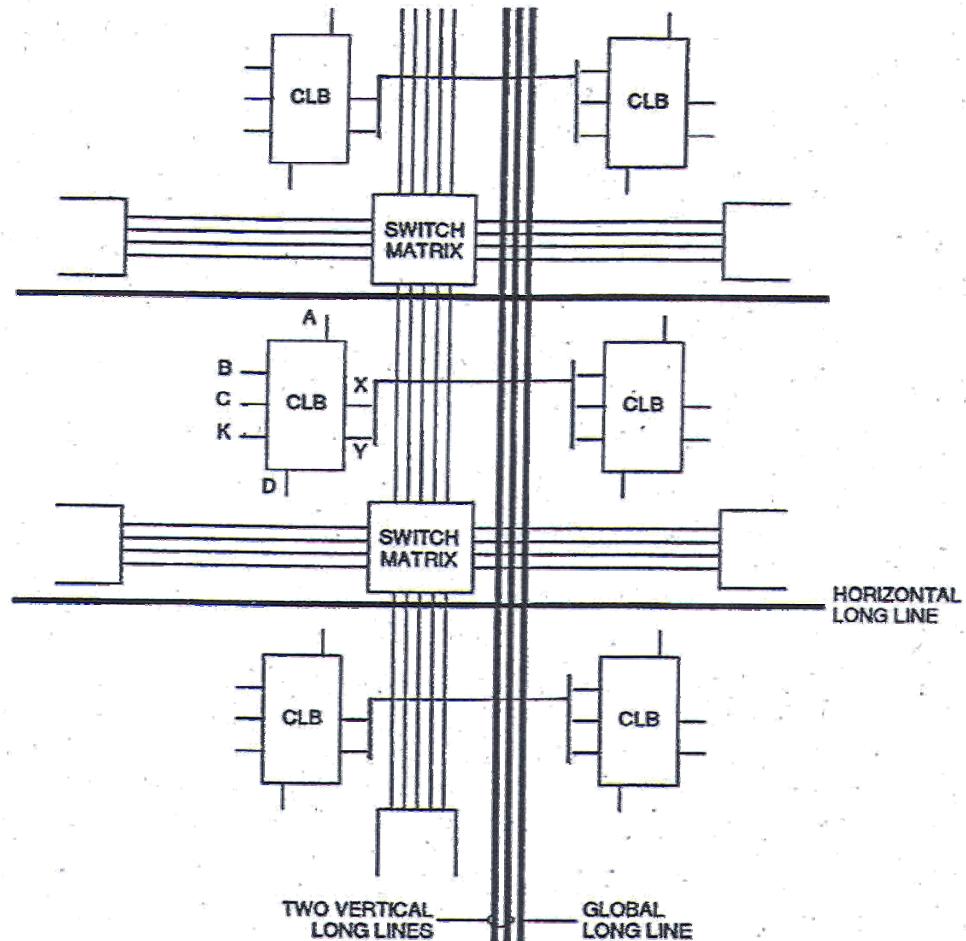


Der Innenaufbau ist für die Benutzung uninteressant.

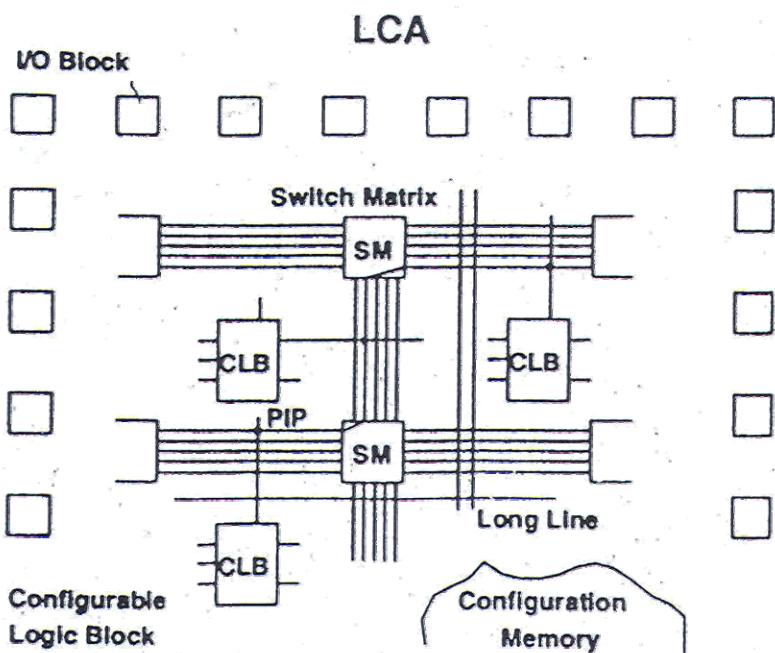
Um nicht alles durch die Zellen lotsen zu müssen, gibt es long lines (liegen über den Zellen). Es gibt dann noch halbe long lines und viertel long lines. Damit kann der Benutzer sehr schnell routen.

Die Hersteller ermitteln den Bedarf der Kunden und entwickeln danach die long lines.

Beim FPGA kann nur die Verzögerungszeit über eine Zelle angegeben werden. Man weiß aber nicht, wie viel Zeit man vom Eingang zum Ausgang braucht (Designabhängig). → Erst nach der Programmierung ersichtlich.



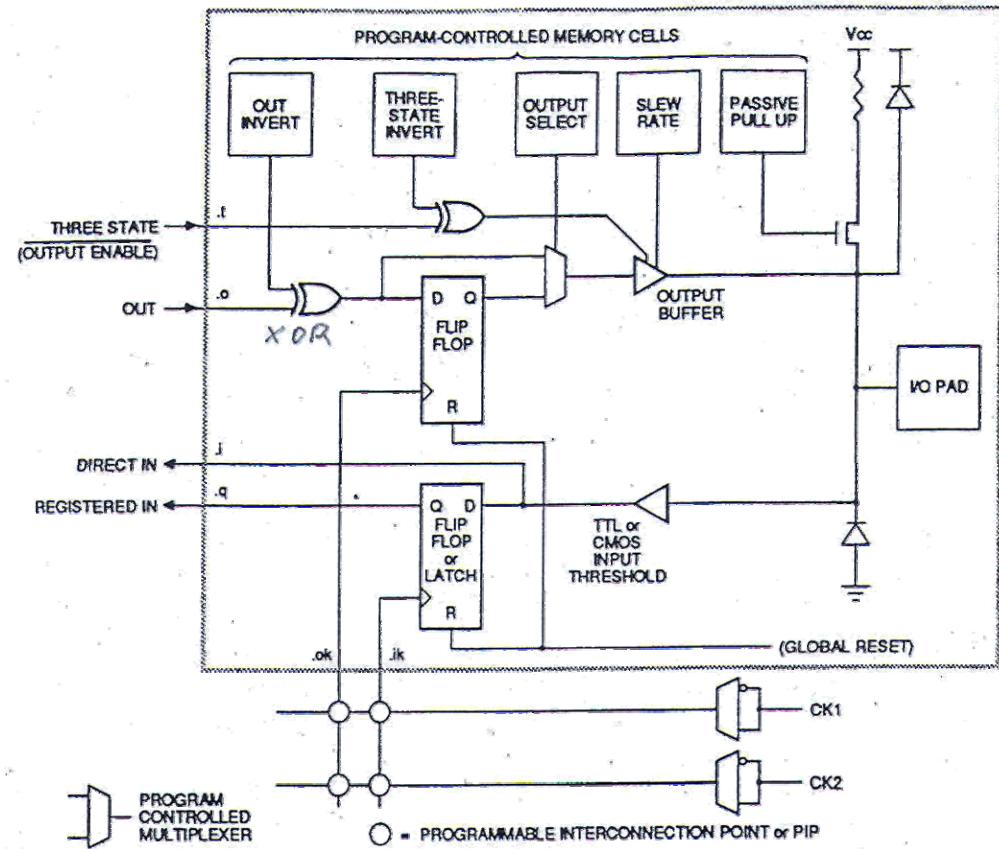
Long Line Interconnect



LCA und FPGA sind dasselbe.

Configuration Memory... RAM für das Design (neben dem Baustein sitzt ein E²PROM, dass ihn beim Start beschreibt)

8.8.1 Ein- und Ausgangszelle



The Input/Output Block includes input and output storage elements and I/O options selected by configuration memory cells.

Es sind wieder Inverter und D-FF-Überbrücker vorhanden.

Für das Tristate gibt es keinen Multiplexer (Laufzeit sparen).
Es wird nachher dann geroutet.

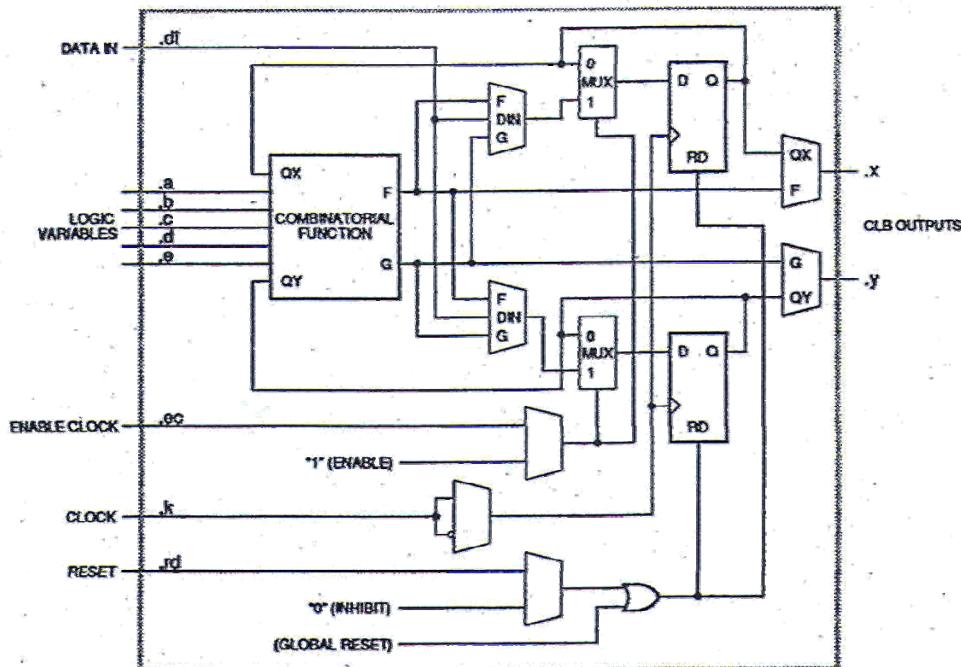
Außerdem kann man die Slew Rate einstellen (Schnelligkeit, EMV, Leistungsverbrauch [mehr Strom um Kondensatoren schneller zu laden]).

Passive Pull Up...ein- und ausschaltbar

2 Dioden gegen Versorgungsspannungsschwankungen → Klemmschaltung zum Schutz des Bausteins (in den meisten inkludiert, aber nicht eingezeichnet).

Die Takteleitungen sind invertierbar.

8.8.2 CLB (Complex Logic Block)



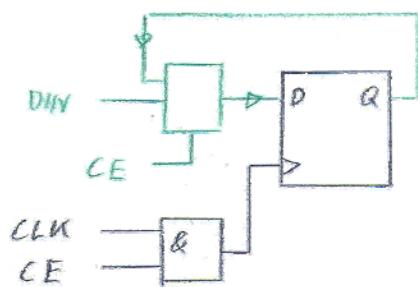
Each Configurable Logic Block includes a combinational logic section, two flip-flops and a program memory controlled multiplexer selection of function.

Der CLB ist eine Speicherzelle.

5 Eingänge, a-e gehen auf ein Mini-PLD (kombinatorische Logik).
Die F und G-Pfade sind symmetrisch aufgebaut.

Der erste Multiplexer geht auf DatenIN, F und G, er kann die Ausgänge des PLDs auswählen, oder es Überbrücken (z.B.: Schieberegister, Addition) → kombinatorische Logik zu langsam. (Fastpath)

Der nächste Multiplexer schaltet den Clock des D-FFs aus.

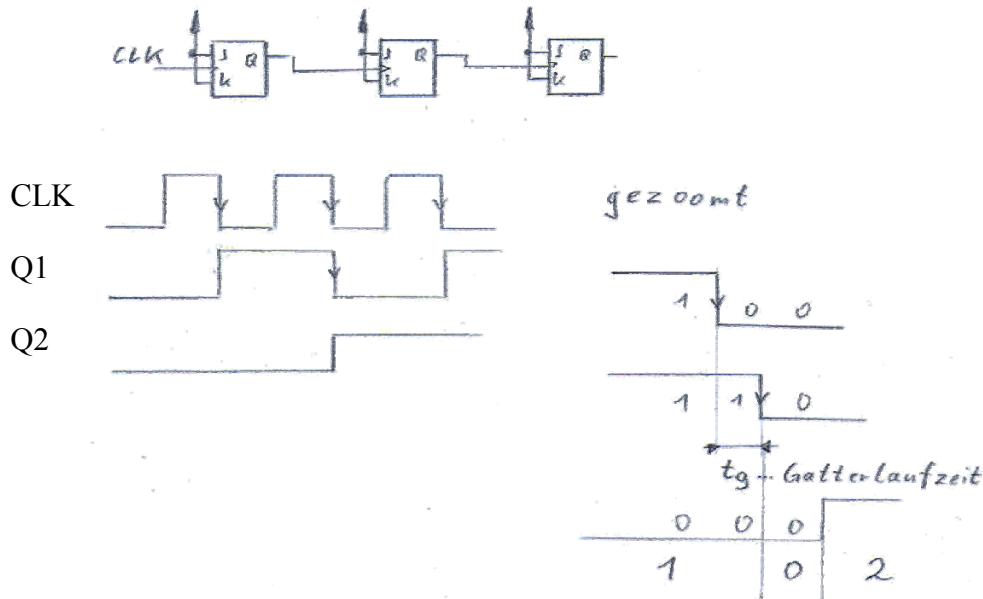


Man will in der Takteleitung keine kombinatorische Logik haben.
→ Einschub über synchrone / asynchrone Logik

8.8.3 Unterschied synchrone / asynchrone Logik

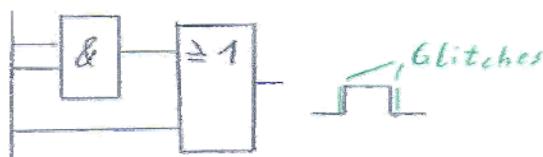
Einen Zähler z.B. kann man synchron und asynchron aufbauen.

asynchron



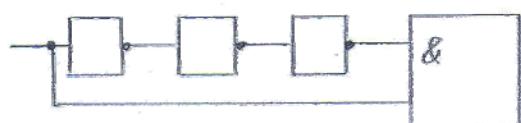
Von 1 auf 2 erhält man eine 0. Diese unerwünschten Zustände nennt man Glitches.

Sie treten bei asynchronem Design auf und wenn die Gatterlaufzeiten nicht gleich sind.



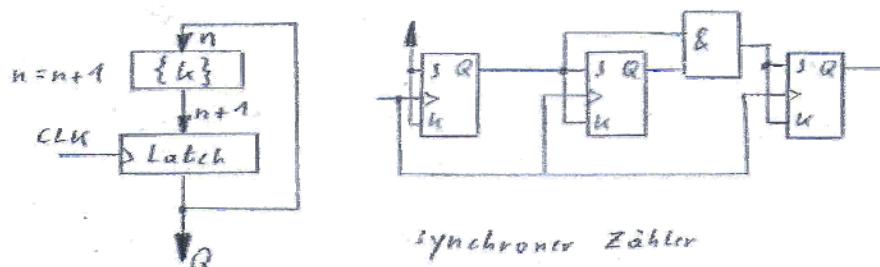
→ Takt nicht mit kombinatorischer Logik erzeugen!

Glitches werden auch bewusst eingesetzt, z.B. bei der Flankenerkennung:



asynchrones Design kann schneller sein (keine FF-Laufzeiten)

synchrone Bauweise



Das erste FF zählt immer, das zweite nur, wenn das erste 1 ist, das dritte nur, wenn beide 1 sind usw.

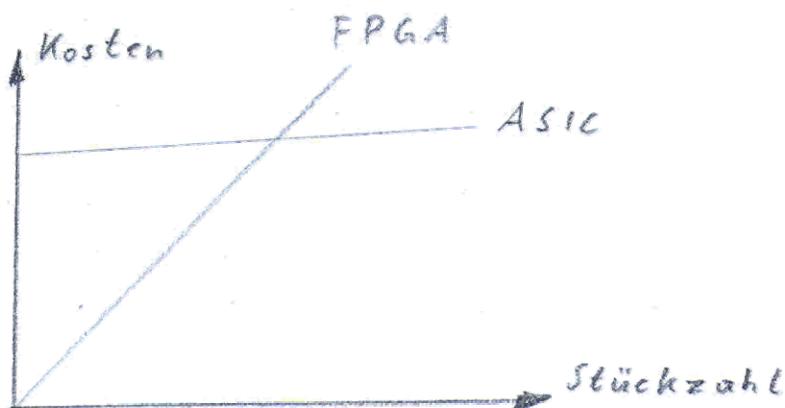
Der Ausgang wird immer synchron mit dem Takt geschalten → Laufzeit des AND-Gatters egal.

FPGAs haben mehr D-FFs (längere Gatterlaufzeit)
CPLD werden für kombinatorische Logik benutzt

8.9 ASIC (Application Specific Integrated Circuit)

ASICS haben eine höhere Gatterdichte als FPGA und CPLD.

Für ASICS setzt man sich mit einer Firma zusammen und entwickelt den Chip – Entwicklung teuer (~ 10000€), Fertigung billig (~ 2-5€).



Ein typisches Beispiel für ASICS sind Grafikkartenchips oder Chips für Fernseher, Taschenrechner, Kinderspielzeug...

Pinanzahl von 4-5 bis hin zu einigen 1000 pins.

ASICS sind nicht so dicht wie Prozessoren (ASICS ~120nm, Prozessor ~60nm).

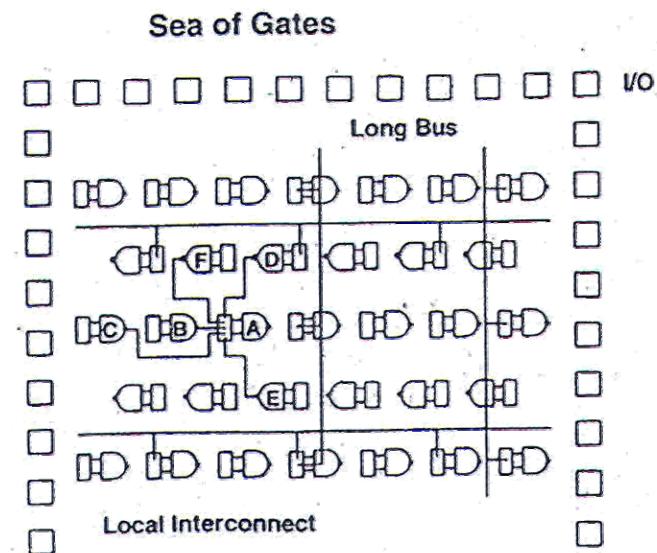
Die Entwicklung dauert ungefähr $\frac{1}{2}$ - 1 Jahr → Es gibt keinen 2. Versuch! Geht es nicht, ist der finanzielle Aufwand groß.

Deshalb sind 10% Designaufwand und 90% Kontrolle (Simulation). → Intel entwickelte einen Chip mit Gleitkommafehler.

8.9.1 Sea of Gates

Das billigste ASIC ist das Sea of Gates.

Hier gibt es vorgefertigte FETs (~10000) im DIE (10x10mm).



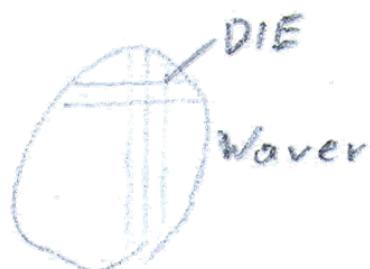
Die FETs werden dann einfach mit Leitungen verbunden.

Durch die Verdrahtung bastelt man sich die Elemente.

Die Integrationsdichte ist nicht besonders groß.

Zur Fertigung:

Es werden Silizium „Wafer“ gebaut, diese bestehen aus den einzelnen DIES.



Sea of Gates wird bei großen Stückzahlen als FPGA-Ersatz geliefert.

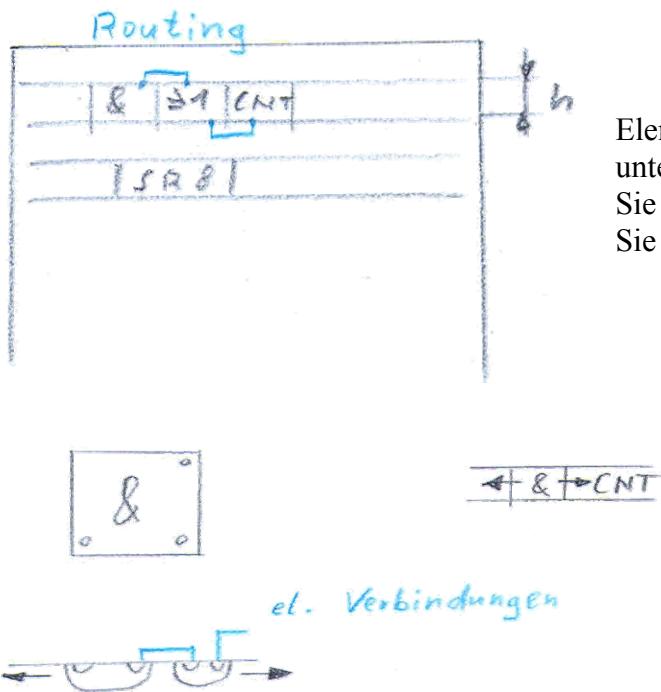
Entwicklungszeit im Monatebereich.

nicht realisiert werden kann:

- μC
- RAM
- Analogschaltungen

8.9.2 Standardzelle

Das DIE wird in Spalten geteilt.



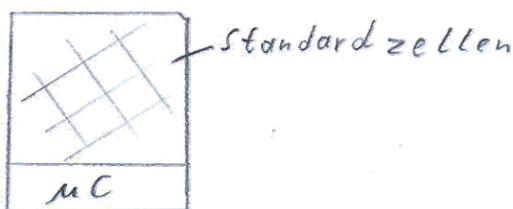
Elemente haben gleiche Höhe, sind aber unterschiedlich breit.
Sie werden aneinander gestückelt.
Sie können horizontal verschoben werden.

Die Elemente sind vordefiniert (als FET-Ausführung). – Maskenbibliothek
Dieses Verfahren ist etwas teurer als Sea of Gates.

wieder kein:

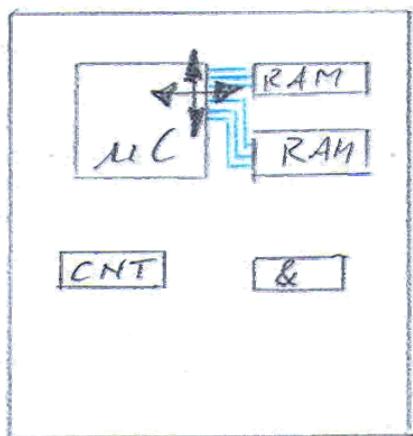
- CPU / μ C
- RAM
- Analog

Allerdings gibt es Firmen, die einen μ C (etc.) inkludieren.



8.9.3 Makro-Zellen

Die Blöcke sind größer und komplett verschiebbar (alle Richtungen).

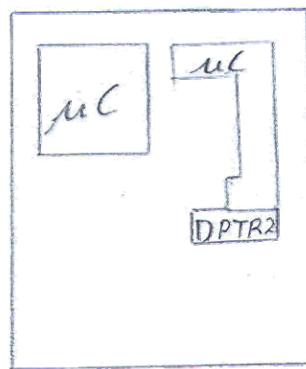


Braucht etwas mehr Zeit → mehr Kosten.

Auch hier gibt es die Bauteile in Bibliotheken (Maskenbibliothek).
→ keine analogen Bauteile.

8.9.4 Full Custom

Jedes Element wird speziell gesetzt (als müsste man einen μC selbst aus Transistoren bauen).



Vorteil:

Flexibilität – Elemente können leicht erweitert werden (2. Datapointer oder Buffer für Serielle)

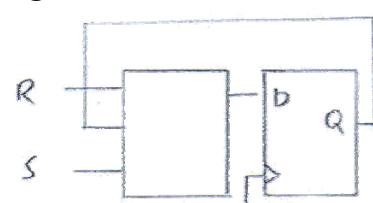
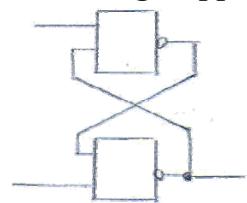
Auch hier gibt es Bibliotheken – Schaltungsbibliotheken ähnlich Protel. Geroutet wird automatisch, der Designer definiert die Bereiche für die Bauelemente oder routet kritische Leitungen. Da es sich hier um reines Halbleitermaterial handelt, kann man auch analoge Schaltungen integrieren (z.B.: OPV).

Dauert dafür lange zum Entwickeln und ist teuer.

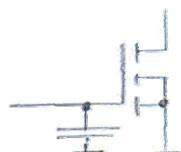
Bei der ASIC-Fertigung kommen auch andere Aspekte hinzu:
Wer darf das Design weiterverkaufen? Patentrechte etc.

8.9.5 Entwurfsregeln

- **Synchrones Design (Glitches!)**
- **Flankengesteuerte D-FFs** (Takt möglichst gleichmäßig verteilen – keine Zustandsabhängigkeit)
Zustandsgesteuerte nur, wenn man den Takt mit kombinatorischer Logik erzeugen müsste.
- **keine rückgekoppelte Logik**

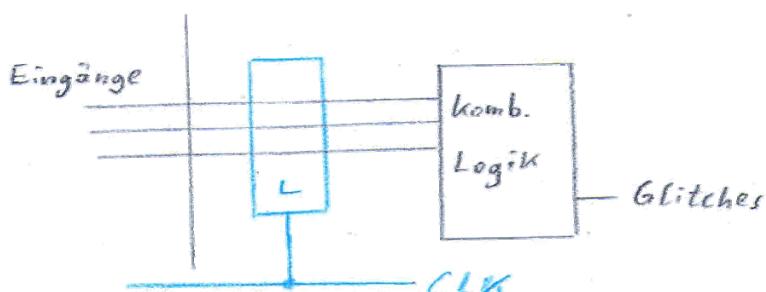


- **CLK nicht durch Logik erzeugen**
- **keine Verzögerungsketten zur Pulserzeugung**



Manchmal benutzt man die parasitären Kapazitäten als Speicher → Vorsicht bei Systemwechsel!

- **Methastabilität meiden**



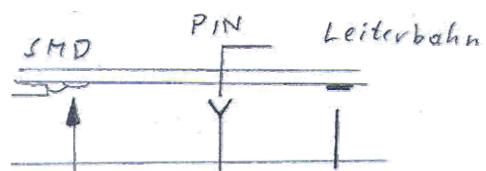
Die unterschiedlichen Laufzeiten am Eingang werden mit einem Latch abgefangen.

8.10 Testbarkeit

Es folgt ein historischer Rückblick aufs Testen.

Man testet um Fehler zu finden, und um den Kunden keine defekten Geräte zu liefern (Imageschaden).

8.10.1 Nadeladapter (große Stückzahl) U,I,R

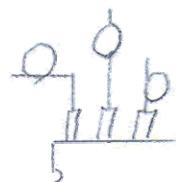


Platine Nadeladapter werden für jeden Print einzeln erstellt (Spezialanfertigung).

Nadeladapter sind teuer, dafür für hohe Stückzahlen konzipiert.

Bis zu 1mm Dichte möglich (Nadeldichte – jede Nadel hat eine Feder).

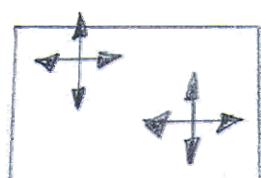
Allerdings wird die Elektronik immer kleiner. Deswegen setzte man eine Zeit lang Testpunkte zu kleinen Bauteilen.



Problem: Da der Testpunkt vom PIN weg ist, kann man nicht die Lötstelle kontrollieren. Manchmal will man auch den Print nach der Printfertigung testen. Nadeladapter würden hier funktionieren, es gibt aber flexiblere Verfahren:

8.10.2 X-Y-Plotter

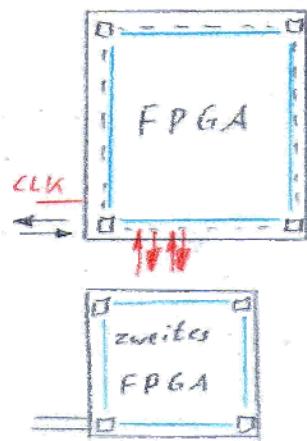
Hier wird mit 2 Nadeln sequentiell (nacheinander) kontrolliert.
Dauert länger, ist flexibler (z.B. für Prototyp).



Die 2 Nadeln können beliebig bewegt werden.

Mit den Gerber Files können so die fehlerhaften Prints gleich aussortiert werden.
Da beim Plotter die Nadeln nicht dicht nebeneinander stehen, können sie dünner ausgeführt werden.

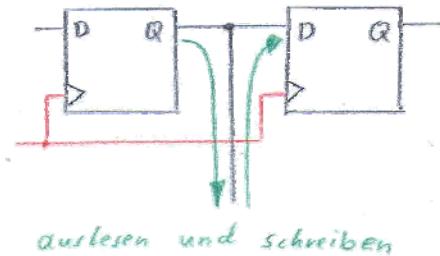
8.10.3 Boundary – Scan – Interface



Man baut in die Ausgangszellen ein **Schieberegister** ein.

Zum FPGA wird ein zweites FPGA gesetzt, dass das erste testet. Die Boundary – Test Hardware wird zusätzlich in das FPGA integriert.

Man schiebt die Daten hinein und schiebt sie im Kreis, dann liest man sie aus. Ist ein IC nicht gut gelötet oder eine Leitung weggeätzt, liefert man das Gerät nicht aus.

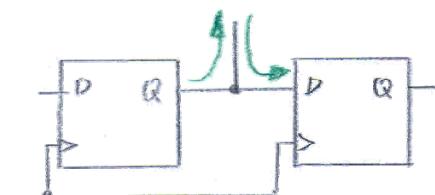
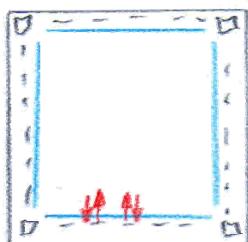


Analoge Schaltungen kann man mit einem Boundary-Scan- Interface nicht testen.

Boundary Scanning wird z.B. bei Handys angewendet.

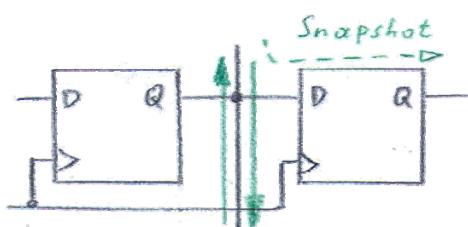
Beim Einkauf muss man auf ein integriertes Boundary Scan System achten.

Diese Methode funktioniert auch nach innen.

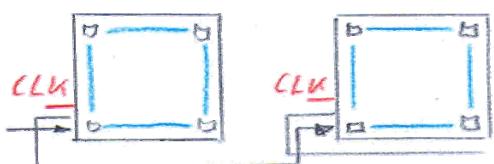


Man testet den Baustein von innen heraus (wird von den Herstellerfirmen vor dem Verkauf gemacht).

Eine dritte Möglichkeit sind Snapshots. Man schreibt die Daten normal nach draußen. Zu einem bestimmten Zeitpunkt werden alle Ausgänge in die FFs gespeichert.

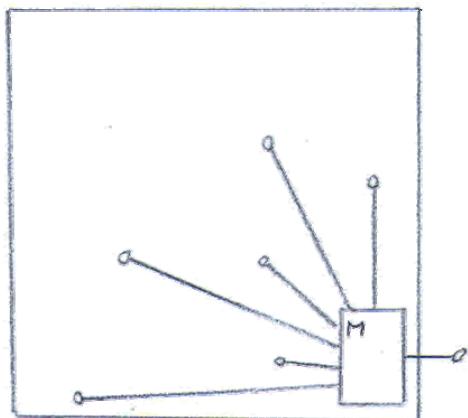


Man kann alle Boundary-Scan Eingänge in Serie schalten. → Man kann viele Chips auf einmal testen.



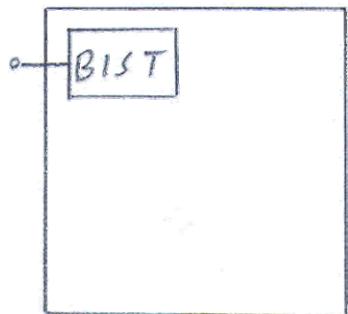
Wird normalerweise mit JTAG-Steckern gemacht.

8.10.4 Multiplexer



Es wird ein Multiplexer ins ASIC integriert, mit dem man bestimmte Signale an den Ausgang schalten kann. → Auch analoge Signale messbar.
Nur bei individuellen Designs (nicht in käufliche ICs integriert).

8.10.5 BIST (Burn in self test)



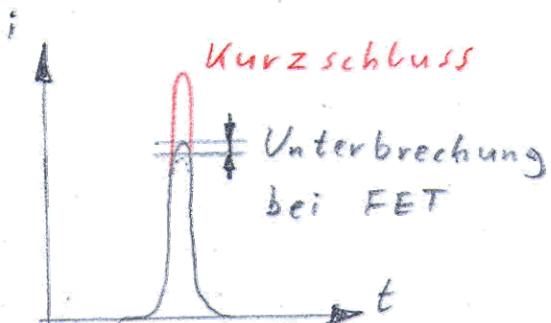
Der BIST meldet am Ausgang durch 0 oder 1, ob der IC funktioniert. → keine externe Hardware notwendig.
Man baut eine kleine Tabelle (komprimiert) mit Werten ein und einen Mini-Sequencer, der alles kontrolliert und dann ein OK ausgibt. Oft hat jeder Logik Baustein einen eigenen Self-Test.

8.10.6 Stromaufnahme

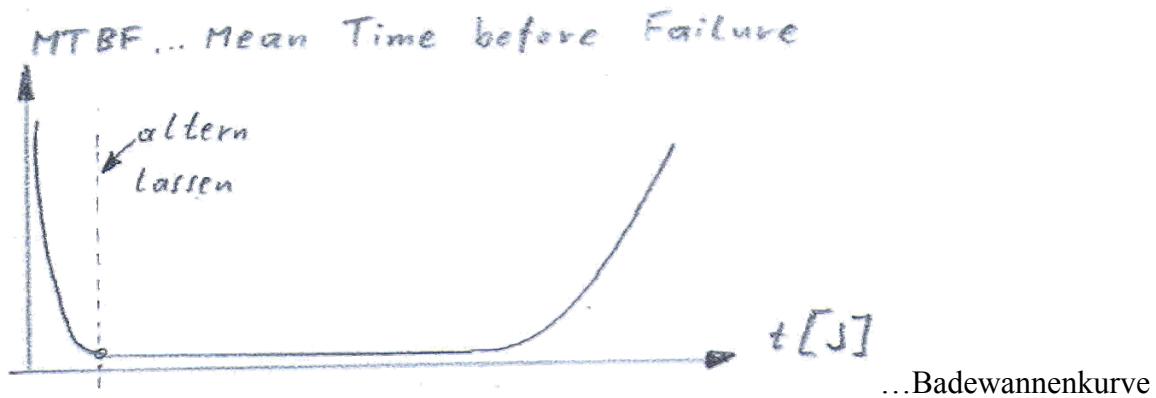


Am Ausgang hat man MOSFET-Endstufen.
Bei einem Schaltvorgang entstehen Peeks. Die Höhe der Peeks ist von der Anzahl der geschalteten Gatter abhängig.

Ein Kurzschluss ist leichter erkennbar. Kaputte ASICs werden weggeworfen.



8.11 Ausfallsraten



Normalerweise hat man ein Gerät weggeworfen, bevor die Bauteile zu alt sind.
Die Fehler am Anfang kommen von Produktionsfehlern (Frühhausfälle).

Problem für Firma:

- Imageschaden
- große Reparationskosten (Versand, Abholung)

Deshalb lässt man die Bauteile altern, bevor man sie testet.
Dieser Prozess heißt „Burnin“. Dabei wird das ASIC bei $\sim 90^\circ$ gelagert, entweder mit oder ohne Spannung.

Durch die hohe Temperatur, wird die Alterung erhöht (von Monaten auf Tage).

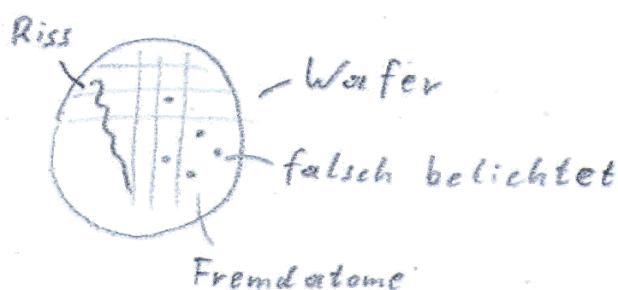
Eine andere wichtige Größe ist der AQL (Acceptable Quality Level). Wie viel Prozent fallen aus von der Produktion?

Man geht davon aus, dass ein Chip, der schleißig gefertigt wurde auch schneller ausfällt (der Ausschuss ist dem Käufer egal).

Man tauscht Kosten gegen Qualität des Fertigungsprozesses.

Bei einem verlangten AQL von 10% z.B. verlässt man sich auf Statistiken der Ausfallsraten bestimmter Fertigungsprozesse.

Fehler:



Je größer ein Chip ist, desto höher ist die Wahrscheinlichkeit von einem Fehler betroffen zu sein. Über $1,2 * 1,5\text{cm}$ wird nichts mehr gefertigt.
CPUs sind nicht billig ($\sim 200\text{€} / \text{Stück}$).

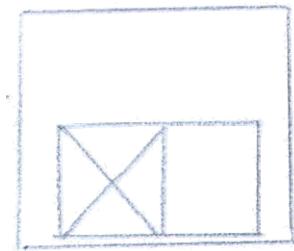
Tricks:

- **Nicht benötigte Einheiten abschalten**

Beispiel:

Level 2 Cache 1M

Ist der Level 2 Cache beim Chip kaputt (Level 2 ist sehr groß), kann man ihn als schlechteren verkaufen (Celeron, Duron).



Kaufen die Leute mehr Celeron als bei Pentium Ausschuss produziert wird, müssen „echte“ Celerons nachproduziert werden.

- **Golden Samples**

Es gibt Hersteller, die Übertaktbarkeit garantieren.

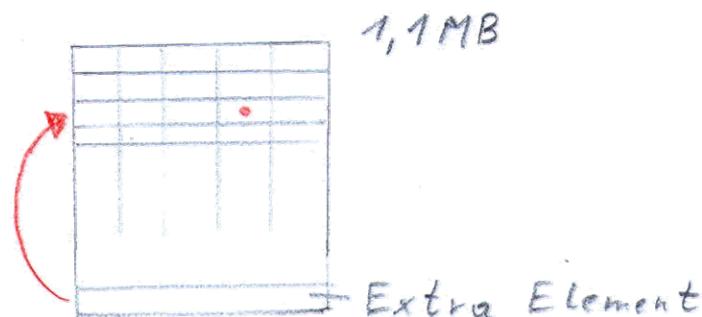
Beim Fertigungsprozess gibt es auch Elemente, die besonders gut sind (spezieller Test).

Ein Beispiel sind High-End Grafikkarten.

Ericson hat auch oft bewusst Golden Samples gekauft (6 statt 4 MHz). Problematisch ist es, wenn die normalen Chips nicht mehr gekauft werden. → Redesign notwendig! Werden zu viele Golden Samples gekauft, drückt man die Nachfrage über den Preis.

- Mehr Elemente vorsehen

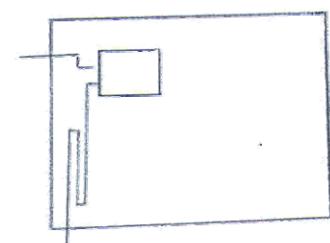
Beispiel: RAM



Fällt nur ein Element aus, wird das extra Element über das kaputte gemappt. Risse über mehrere Elemente können dadurch natürlich nicht repariert werden. Die RAMs müssen bei einem Fehler aber noch speziell behandelt werden, um das extra Element einzubinden.

8.12 Effekte beim ASIC-Design

8.12.1 Signal Delay

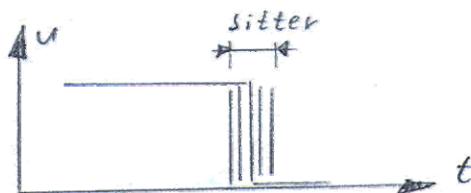


Durch unterschiedlich lange Leitungen verschieben sich die Laufzeiten.

Beim Simulieren wird eingegeben, was am Eingang liegt und was am Ausgang liegen soll.

Auch der Clock-Impuls läuft mit Verzögerungen über die Leitungen (Clock Slew)

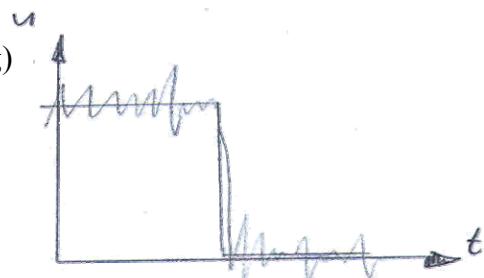
8.12.2 Jitter



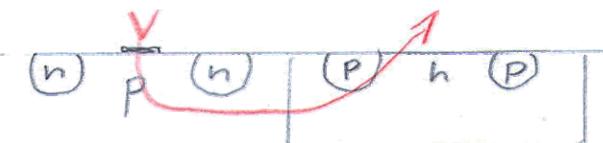
Auch Jitter lassen sich aussimulieren.

8.12.3 Noise

Bei aktuellen μ Cs wird der Abstand zwischen Low und High immer kleiner (geringe Versorgungsspannung)
→ Rauschen wird problematischer.



8.12.4 Latchup-Effekt



Zwei Halbleiterbauteile sitzen nebeneinander.

Passiert einmal ein Kurzschluss, hat man folgenden Übergang: pn-pn (Thyristor). Der Thyristor schaltet nie wieder aus (Haltestrom). Der Zündvorgang kann durch Überspannung (Funke) ausgelöst werden.

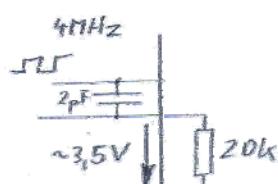
Meistens überlebt es der IC, dann muss er stromlos gemacht werden (abschalten).

Das ist problematisch, wenn er immer laufen muss. (Server, ...)

Man kann beim Design versuchen diesen unerwünschten Thyristor möglichst schlecht zu machen.

Leider bemerkt man oft zu spät, dass ein Latchup-Effekt vorhanden ist. → Hohe Kosten für neues ASIC.

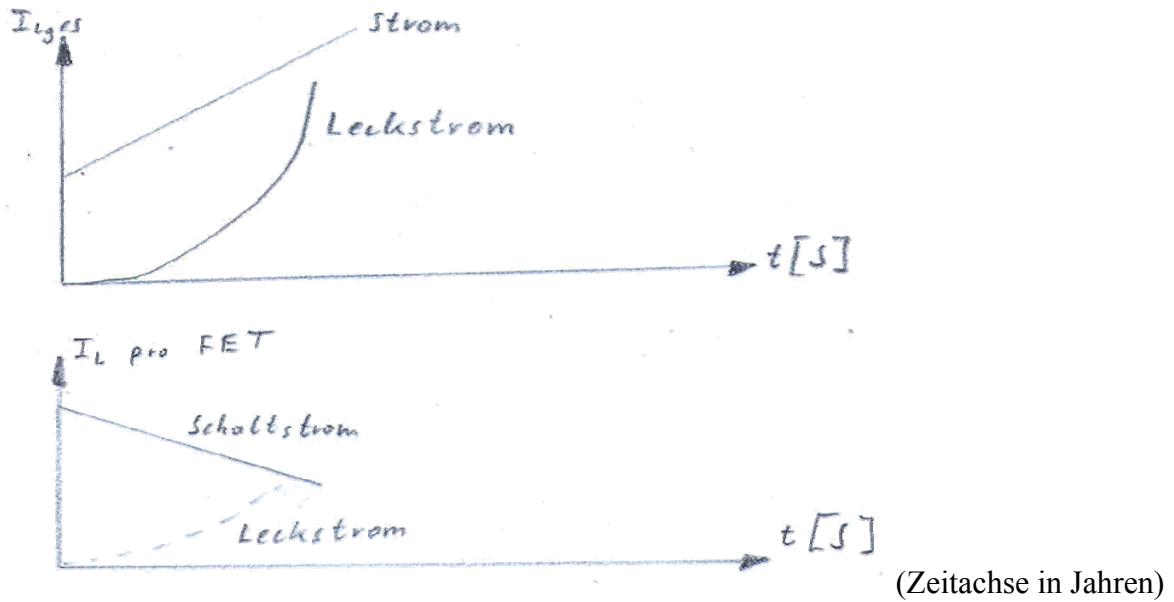
8.12.5 Übersprechen



Über parasitäre Kapazitäten werden Signale übertragen.

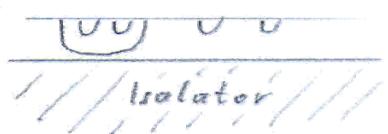
Nicht benutzte Eingänge auf Masse legen!

8.12.6 Leakage (Leckstrom)

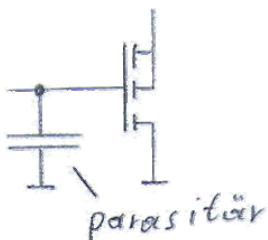


Weil man immer weniger Strom benutzt (immer kleinere Bauweise) und der Leckstrom ansteigt, gibt es Probleme:

SOI (Silicon on Insulator)



Durch den Isolator hat der Leckstrom längere Wege → größerer Widerstand. Früher gab es am Silizium einen Anschluss zum ableiten des Leckstromes.



Mit dem parasitären Kondensator kann man einen Speicher bauen. Der Leckstrom ist aber temperaturabhängig → Man kann das Gerät nicht mit beliebig kleiner Frequenz betreiben. Nicht nur f_{max} , sondern auch f_{min} beim IC. „fully static“ bedeutet, dass solche Effekte nicht ausgenutzt wurden ($f_{min} = 0\text{Hz}$).

Einen Pentium Prozessor kann man z.B. nicht mit 0Hz betreiben.

Bei einem Technologiewechsel beim Fertigungsprozess kann dies nicht mehr funktionieren (f_{min} rutscht über f_{max}).

8.12.7 Stress

Pentium Prozessor: $P = 100\text{W}$

$$A = 1\text{cm}^2$$

$$\rightarrow 100\text{W/cm}^2$$

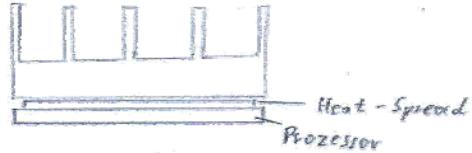
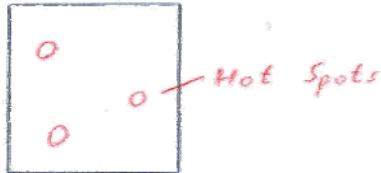
Herdplatte (Vergleich): $d = 30\text{cm}$

$$P = 1\text{kW}$$

$$\rightarrow A = 706\text{cm}^2$$

$$\rightarrow 1,3\text{W/cm}^2$$

Die Leistung am Chip teilt sich nicht gleichmäßig auf.



...beanspruchte Bereiche erwärmen sich stark.

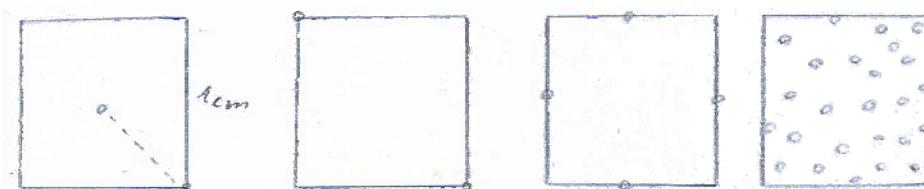
Es wird eine Folie eingebaut, welche die Hitze verteilt.

Problem:

$I_V = 50\text{A} \rightarrow$ mehrere Pins für Versorgungsspannung benötigt.

$$4\mu\Omega_{0,2\mu} \rightarrow 89\Omega_{\text{mm}}$$

Bei einer Leiterbahndicke von $1\mu * 0,2\mu$ fallen auf einem mm 100mV ab.



Deshalb setzt man die Versorgungspins möglichst überall auf dem IC. (Geschichtliche Entwicklung von links nach rechts) \rightarrow Alle VCC-Pins müssen angeschlossen werden.

$$\text{Stromdichte} = 1\text{mA}/0,2\mu\text{m} = 5000\text{A/mm}^2$$

Problem der Ladegeschwindigkeit von Kondensatoren:

$$1_F \quad \Delta t = 4\text{ns} (16\text{Hz})$$

$$4U = 5V$$

$$I = C \cdot \Delta U / t = 5\text{mA} \text{ (pro FET)}$$

Schalten 50 Millionen FETs gleichzeitig \rightarrow BAMM!

Deshalb hält man die Kapazitäten möglichst klein und benutzt keine 5V.

8.13 Kosten

- Designkosten
- Debugkosten
- Ausbildungskosten
- Widerverwertbarkeit

Man baut oft mehrere Funktionen in einen Chip ein (Sender/Empfänger) und benutzt ihn je nach Anwendung anders. (Taschenrechner) → kein Neudesign, größere Stückzahl.

9 VHDL

Der größte Konkurrent zu VHDL ist Verilog.
Von der Art sind sie ähnlich, die Syntax ist unterschiedlich.
VHDL ist genormt.

Die Synthese wird vom Hersteller abhängig.
Für den oberen Entwicklungsteil gibt es viele verschiedene Anbieter.
Fürs FPGA werden JDEC Dateien erzeugt.
Die Simulation ist extrem wichtig und funktioniert in 2 Phasen.
Vor der ASIC Produktion werden Zeiten angenommen und damit das VHDL-Programm getestet.
Funktioniert die Simulation, wird produziert und danach mit echten Zeiten getestet (10% programmieren, 90% simulieren).

Man kann es Simulations-spezifisch schreiben → funktioniert in Simulation oder Hardware-spezifisch mit echten Zeiten.
(Grundeinheit der Simulation sind Femtosekunden!)

In der HTL gibt es 3 Systeme.

Xilinx	Web-Pack	gratis
Altetra	Quatrus, Max II	gratis, mit Lizenz an Laptop gebunden
Altium	DXP	braucht FPGA angeschlossen, Lizenz!

Beim VHDL programmieren gibt es 2 große Blöcke:

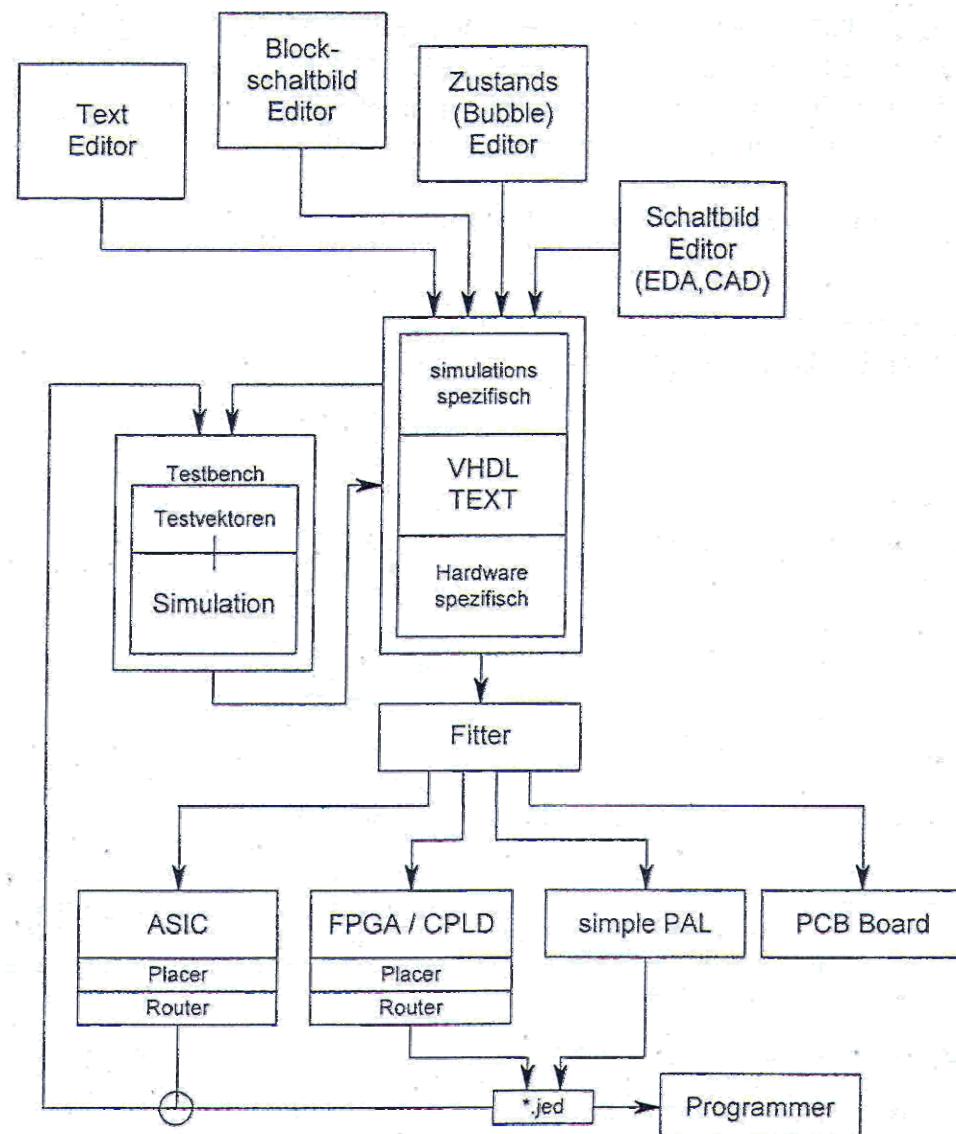
ENTITY

Beschreibt nach außen und innen

ARCHITECTURE

legt fest, was sich innen tut

9.1 VHDL Arbeitsumgebung



Die einzelnen Softwarepakete werden meist von unterschiedlichen, spezialisierten Anbietern geliefert.

Placer, Router und JEDEC-Generator kommen vom Hersteller der entsprechenden Hardware (Altera, Xilinx, TI usw.)

Die Simulatoren und der VHDL-Editor stammen oft aus einer Hand.

Die meisten EDA-Firmen versuchen alles bis zum Simulator und der VHDL-Schnittstelle anzubieten. (PCAD, ORCAD, REDAC, PADS, PROTEL, usw.).

Ein bekannter Fitter ist MINC.

COMPASS und Visual-HDL sind Werkzeuge zum Erstellen und Testen von VHDL-Code.
An der Schule gibt es ein einfaches Demo-Werkzeug: WARP, bestehend aus dem VHDL-Editor GALAXY und dem Simulator NOVA.

9.2 Allgemeines

VHDL = Very high speed integrated circuits and Hardware Description Language

Es handelt sich um eine sehr mächtige Sprache zur Beschreibung digitaler Systeme, Geräte und Bausteine. Neben VHDL ist besonders in den USA die einfachere und kompaktere Beschreibungssprache VERILOG-HDL verbreitet, für einfache Probleme wird manchmal auch ABEL eingesetzt.

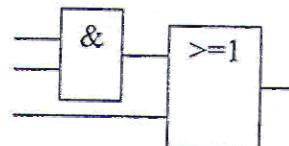
VHDL wurde erstmalig 1987 unter IEEE 1076 spezifiziert und 1993 modifiziert. Dann folgte eine Erweiterung unter IEEE 1164 welche 1996 unter IEEE 1076.3 zur Norm erhoben wurde. VIEWsym unterstützt nur die Version 1987 hat jedoch einen sehr guten Simulator. GALAXY unterstützt IEEE 1164, hat aber nur einen Funktionssimulator (NOVA).

Die wichtigsten Beschreibungsverfahren für digitale Geräte sind:

- **Schaltung:**

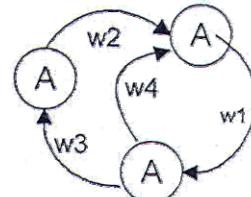
Eine altbekannte, verbreitete Methode.

Wird vor allem für Blockdiagramme verwendet.



- **Zustandsdiagramm**

Ist sehr effizient für Ablaufsteuerungen (Bubblediagramm).



- **Textale Beschreibung (z.B. durch VHDL):**

```
-- inverter : ein Inverter = Überschrift = Kommentar -----
LIBRARY ieee;                                     -- Baut ei l ebi bl i ot hek
USE ieee.std_logic_1164.all;                      -- Abschnitt der Bi bl i ot hek
ENTITY Inverter IS                                -- Beschaltung des Objektes
  GENERIC ( delay : TIME := 10ns);                -- Standardzeit für Simulation
  PORT    ( data_in : IN BIT := '0';              -- Namen und Art der Signale
            ( data_out: OUT BIT);
END Inverter;
ARCHITECTURE behav OF inverter IS               -- Funktion des Objektes
BEGIN
  data_out <= NOT data_in AFTER delay;           -- ein Inverter
END behav;
```

Beschreibungen in VHDL sind unabhängig von der Implementierung. Das Problem wird zuerst in VHDL beschrieben, dann ausgiebig getestet. Erst nachher wird entschieden welche Teile des Designs durch welche Hardware realisiert werden (z.B. als CPLD, FPGA oder als ASIC).

Die VHDL Texte werden mit einem Text-Editor geschrieben (*.VHD), dann mit einem VHDL-Compiler übersetzt (*.JED, *.EDF, ...). In einem Simulator wird nun mit Standardzeiten (eventuell aus der GENERIC Anweisung) das Verhalten simuliert. Ist es zufriedenstellend, wird mit einem firmenspezifischen FITTER ein PLACE & ROUTE Prozess gestartet, der die Schaltung auf einer speziellen Logik realisiert (XILINX, ALTERA, ... usw.). Der Fitter liefert nun reale Zeiten für den Simulator. Es wird wieder ausgiebig simuliert. Dann erst wird das Design freigegeben und tatsächlich gefertigt.

Vorteile von VHDL:

- Es werden weniger Bausteine und es wird weniger Printplatz benötigt.
- die Stromaufnahme wird kleiner und die Signallaufzeiten werden kürzer.
- die Zeit zur Fehlersuche und Fehlerbehebung wird wesentlich verkürzt (Software)
- die Schaltung kann von der Konkurrenz (praktisch) nicht analysiert werden (reengineering)
- die Funktion kann im nachhinein, bei gleicher Pinbelegung leicht modifiziert werden
- wenn es vom Print her gefordert wird, können Pins leicht getauscht werden
- die Anpassung an die geforderte Stückzahl (FPGA oder ASIC) ist leichter durchführbar
- „time to market“ ist kurz
- einmal erstellter Code ist leicht modifizierbar und sehr gut wieder verwendbar
- die Fitter (Synthesewerkzeuge) erzeugen in kurzer Zeit fehlerfreie, gut optimierte Schaltungen

Nachteile von VHDL:

- Hoch qualifiziertes Personal wird benötigt (Ausbildungskosten)
- trotz Optimierung erzeugen die Fitter bei allgemeinen Beschreibungen meist etwas overhead
- die Qualität der Synthese hängt vom Softwarewerkzeug ab

9.3 Aufbau eines VHDL Textes

VHDL kennt mehrere Ebenen der Schaltungsbeschreibung

- a) Verhaltensbeschreibung (= behavioral = hardwareunabhängige Verhaltensbeschreibung)
ARCHITECTURE behavioral OF Comp4 IS
BEGIN
 dout <= '1' when (a_bus = b_bus) else '0';
END behavioral;
oder
ARCHITECTURE behavioral OF Comp4 IS
BEGIN
 comp: PROCESS (a_bus, b_bus)
 BEGIN
 IF a_bus=b_bus THEN dout <='1';
 ELSE dout <='0';
 END IF;
 END PROCESS comp;
END behavioral;
- b) Schaltungsbeschreibung (=structural = hardwarenahe; ähnlich einer Netzliste)
ARCHITECTURE structural OF Comp4 IS -- eine Netzliste mit Gattern
 SIGNAL x : std_logic_vector (0 to 3); -- eine interne Hilfsvariable
BEGIN
 i c0: xnor2 PORT MAP (a(0), b(0), x(0)); -- XNOR aus Bioblock
 i c1: xnor2 PORT MAP (a(1), b(1), x(1)); -- [XNOR (in, in, out)]
 i c2: xnor2 PORT MAP (a(2), b(2), x(2)); -- oder zB. XBlock aus Xilinx
 i c3: xnor2 PORT MAP (a(3), b(3), x(3));
 i c4: and4 PORT MAP (x(0), x(1), x(2), x(3), dout);
END structural;
- c) Diverse Zwischenstufen wie “dataflow” und “register transfer level”

Die Anweisungen in den Modellbeschreibungen werden normalerweise **parallel abgearbeitet** (= nebenläufig = gleichzeitig = concurrent), sie können aber auch sequentiell bearbeitet werden, wenn sie durch PROCESS gekennzeichnet sind.

Eine VHDL Datei besteht entweder aus den Abschnitten **ENTITY** und **ARCHITECTURE** oder aus einer **PACKAGE** Deklaration zur Verdrahtung von Blockschaltbildern.

In ENTITY werden mit GENERIC konstante Größen definiert und mit PORT das Interface nach außen hin beschrieben (= Pinbelegung). Die Konstanten sind oft Busbreiten und Standardwerte für eine Gatterverzögerungszeit. Hier wird auch der Name des Modells festgelegt.

In ARCHITECTURE wird die Funktion des Bausteins beschrieben. Durch den Modellnamen ist die Funktion mit einem bestimmten ENTITY-Eintrag verbunden.

In PACKAGE werden Funktionsblöcke durch COMPONENT (so wie in ENTITY) deklariert und die zugehörigen Anschlüsse nach außen hin sichtbar gemacht. Mit USE können diese Anschlüsse in anderen VHDL-Texten verwendet werden.

Jedem Abschnitt können Funktionen und Prozeduren vorangestellt werden.

Funktionen (FUNCTION) akzeptieren nur Parameter, die Konstante oder Input-Signale sind. Die Parameter sind Werteparameter und können nicht modifiziert werden. Funktionen liefern einen beliebigen Rückgabewert.

Prozeduren (PROCEDURE) akzeptieren als Parameter jeden Typ und können jeden Parameter modifizieren. Sie liefern keinen Rückgabewert.

VHDL-Befehle sind nicht „case sensitive“ (das heißt Groß- und Kleinschreibung wird nicht unterschieden), wohl aber in manchen Versionen die Variablenbezeichner.

Jeder Befehl endet mit einem Strichpunkt (;).

Kommentare beginnen mit zwei Bindestrichen (-- Kommentar).

9.4 Datentypen und Objekte

Grund-Datentypen:

Neben den reinen Zahlenformaten gibt es in VHDL spezielle Datentypen zur Bearbeitung von Signalen.

Von den Zahlentypen wird praktisch nur der Typ Integer als Index in Arrays und als Zählvariable verwendet. Ein **Integer** hat (im Unterschied zu C) 32 Bit.

Der Typ **boolean** mit den Werten TRUE und FALSE wird als Flag verwendet.

Zur Beschreibung von **Signalen** steht der Typ **Bit** bzw. **Bit_Vektor** zur Verfügung. Für den Typ Bit stehen je nach verwendeter Software verschiedene Arten zur Verfügung:

BIT (mit 2 Pegel)

'0' = forcing low -- ieee 1706
'1' = forcing high

STD_LOGIC oder auch STD_ULOGIC (mit 9 Pegel)

'U' = uninitialized
'X' = forcing unknown
'0' = forcing low
'1' = forcing high
'Z' = forcing impedance
'W' = weak unknown (OC oder OE)
'L' = weak low (OE)
'H' = weak high (OC)
'-' = don't care

Dabei kann std_logic mehrere Quellen haben, std_logic jedoch nur eine.
VL_bit in SUSIE hat bis 12 logische Zustände (... diverse unknown Zustände)

Für Busse gibt es Bit-Arrays (Bit-Vektoren), für Testvektoren 2-dimensionale Bit-Arrays.

```
GENERIC ( busbreite: INTEGER := 8 ); -- := Zuweisung für Zahlen
PORT ( SIGNAL counter: OUT STD_LOGIC_VECTOR (busbreite downto 0);
        SIGNAL clock: IN STD_ULOGIC );
```

Für die Zeit gibt es den Datentyp **TIME**, seine Grundeinheit sind fs. Es ist jedoch auch jeder andere genormte Vorsatz wie ms, us, ns definiert.

Sehr häufig werden **benutzerdefinierte Aufzähltypen** (enum) eingesetzt.
TYPE typ_Zustand IS (aus, drucken, warten, laden);

Erweiterungen zu den Datentypen erlauben eine schnellere Simulation:

```
Teilbereichstypen   TYPE typ_index IS RANGE 1 TO 6;
Teiltyp mit Bereich SUBTYPE typ_Adress : integer RANGE 0 TO 63
```

Aliases erlauben es Originaltypen oder Teile davon unter einem eigenen Namen anzusprechen:

```
SIGNAL Adress : STD_LOGIC_VECTOR (31 downto 0);
ALIAS Hi_Adress : STD_LOGIC_VECTOR (31 downto 0) IS Adress (31 downto 28);
```

Modi zu den Signalen helfen bei der Fehlerprüfung

IN	nur Eingang
OUT	nur Ausgang, nicht geeignet für interne Rückführung
BUFFER	Ausgang für interne Rückführungen
INOUT	für bidirektionale Signale, mit oder ohne Rückführung

Attribute liefern zusätzliche Informationen

'LEFT', 'RIGHT', 'HIGH', 'LOW', liefern Teile aus einem Vektor (Bus)
'EVENT' liefert TRUE bei einer Signalflanke

VHDL-Objekte:

Objekte heißen in VHDL auch Klassen (Class) und haben nichts mit den Klassen aus C++ zu tun. Objekte sind Konstante, Variable und Signale. Fehlen die Schlüsselwörter, so versucht der VHDL-Compiler jeder Größendefinition ein Objekt zuzuordnen.

CONSTANT	Konstante haben einen festen Wert CONSTANT BusBreite: INTEGER := 8; CONSTANT delay: TIME := 7ns;
VARIABLE	Variable beinhalten Zahlen (Indices, usw.) Die Zuweisung erfolgt mit := VARIABLE firstTime: BOOLEAN := TRUE; VARIABLE index: INTEGER := 16#FFFF#;
SIGNAL	Signale entsprechen logischen Verbindungen. Jedes Signal besitzt eine Quelle (Treiber) und eine oder mehrere Senken (Reader) in denen es benutzt wird, dazwischen liegt eine Laufzeit. Signale sind zeitlich veränderlich und können nur einen Bit- oder Bit-Vektor-Typ annehmen. Die Zuweisung erfolgt mit <=

```
SIGNAL clock, enable : IN BIT <= '0';
```

Zahlenbasis in VHDL:

Es können die Zahlensysteme Binär, Oktal, Hexadezimal und Dezimal gewählt werden.

Für Variable und Signale wird dazu eine unterschiedliche Syntax verwendet.

Fehlt die Spezifikation der Basis, wird entweder Hexadezimal oder Dezimal als Zahlensystem angenommen.

Für Variable: Basis#Zahl# VARIABLE Hi : INTEGER := 16#3FC7#;

Für Signale: B, O, X

```
SIGNAL DataBus : STD_LOGIC_VECTOR (15 downto 0) <= X "3FC4";
```

9.5 Kombinatorische Logik in VHDL

Logische Gleichungen:

Verknüpfungen können (wie in der PLD-Programmierung üblich) durch logische Gleichungen beschrieben werden (= RTL = Register Transfer Level). Hier z.B. ein Multiplexer:

```
-- mux4a : ein 4/1 Multiplexer mit Log. Gleichungen -----
library ieee;
use ieee.std_logic_1164.all;
entity Mux4a IS
    port ( din : in std_logic_vector(3 downto 0);
           sel : in std_logic_vector(1 downto 0);
           Qout : out std_logic); -- 4 Bit Mux
                           -- Eingangs Bus
                           -- Auswahl Bus
                           -- Ausgang
end Mux4a;
architecture behavior OF Mux4a IS
begin
    Qout <= (din(0) and not(sel(1)) and not(sel(0)))
              or (din(1) and not(sel(1)) and sel(0))
              or (din(2) and sel(1) and not(sel(0)))
              or (din(3) and sel(1) and sel(0));
end behavior;
```

Die logischen Operatoren and, or, nand, nor, xor, xnor und not stehen zur Verfügung. Sie haben keine Rangordnung, Klammern setzen ist daher sehr wichtig.

With – Select – When:

Nach WITH vor SELECT wird das steuernde Signal angegeben. Gleiche Signalwerte für das steuernde Signal dürfen nach WHEN nicht vorkommen. Es ist empfehlenswert am Ende stets eine WHEN OTHERS Klausel anzufügen.

Hier z.B. wieder unser Multiplexer:

```
-- mux4b : ein 4/1 Multiplexer mit WITH, SELECT, WHEN -----
-- !!! LIBRARY und ENTITY genauso wie bei mir vorgenommen Beispiel !!! -----
architecture behavior OF Mux4b IS
begin
    with sel select
        Qout <= din(3) when "11",
                  din(2) when "10",
                  din(1) when "01",
                  din(0) when "00",
                  'X' when others; -- für Signal wie "ZZ" oder "XX"
end behavior;
```

When – Else:

Mit dieser Anweisung können Prioritäten gesetzt werden (Priority Encoder), denn sie wird verlassen, sobald eine aufgelistete Bedingung erfüllt ist.

Hier z.B. wieder unser Multiplexer:

```
architecture behavior OF Mux4c IS
begin
    Qout <= din(3) when (sel ="11") else
              din(2) when (sel ="10") else
              din(1) when (sel ="01") else
              din(0);
end behavior;
```

Manchmal ist die Beschreibung einer Schaltung durch IF THEN ELSE oder CASE WHEN sehr einfach. Diese Befehle dürfen jedoch nur innerhalb von PROCESS stehen. Daher findet man auch in Beschreibungen von kombinatorischer Logik auch Abschnitte mit PROCESS.

Hier ist wieder unser altbekannter Multiplexer mit IF / ELSEIF und Case / WHEN :

If – Elseif (Mux4d):

```
mu xi t : process ( sel )
begin
  if sel = "11" then Qout <= di n( 3 );
  else if sel = "10" then Qout <= di n( 2 );
  else if sel = "01" then Qout <= di n( 1 );
  else Qout <= di n( 0 );
end if;
end process mu xi t;
```

Case When (Mux4e):

```
process ( sel )
begin
  case sel is
    when "11" => Qout <= di n( 3 );
    when "10" => Qout <= di n( 2 );
    when "01" => Qout <= di n( 1 );
    when others => Qout <= di n( 0 );
  end case;
end process;
```

Man kann kombinatorische Logik selbstverständlich auch strukturell (als Netzliste) mit PORT und MAP beschreiben.

9.6 Sequentielle Logik in VHDL

Übliche VHDL Simulatoren können nur synchrone Logik simulieren. Da man asynchrone Designs nur schwer testen kann, sollte man auch aus diesem Grund nur synchrone Schaltwerke entwerfen.

Alle sequentiellen Schaltungen enthalten PROCESS Abschnitte. Auf PROCESS folgt entweder eine „SENSIVITY LIST“ oder nach BEGIN als erstes ein WAIT UNTIL. Wenn eine Liste vorhanden ist, wird der Prozesskörper nur dann ausgeführt, wenn irgendein Signal aus der Liste seinen Pegel ändert. Viele Werkzeuge (Fitter und Simulatoren) ignorieren jedoch die Sensivity List, daher sollte man im Prozesskörper alle Signale nochmals auflisten.

Hier als Beispiel ein flankengesteuertes D-Flip-Flop:

```
-- dfff1 : ein flankengesteuertes D-FF -----
library ieee;
use ieee.std_logic_1164.all;
entity D_FF_f1 IS
  port ( din, clk : in std_logic;          -- Eingänge Data und Clock
         ( Qout      : out std_logic );       -- Ausgang Q
end D_FF_f1;
architecture behaviour OF D_FF_f1 IS
begin
  process (clk)           -- Nur bei einer Flanke von clk
  begin
    if (clk'event and clk='1') then Qout <= din;   -- positive Flanke
  end if;
  end process;
end behaviour;
-----
```

Für ein zustandsgesteuertes D-Flip-Flop sieht der Prozess so aus:

```
process (clk, din)           -- bei Änderung von clk oder din
begin
  if clk='1' then Qout <= din;   -- positive zustandsgesteuert
end if;
end process;
```

Nun erweitern wir unter Zuhilfenahme der Standardfunktion RISING_EDGE und FALLING_EDGE unser D-FF um einen asynchronen RESET und einen synchronen SET.

```

process (clk, reset)           -- bei Änderung von clk oder reset
begin
  if reset='1' then Qout <= '0';    -- asynchroner Reset, dominant
  elsif rising_edge(clk) then
    if preset ='1' then Qout <= '1';    -- synchroner Preset
    else Qout <= din;
    end if;
  end if;
end process;
```

Noch einige Beispiele wie ein JK-FF, ein Zähler und ein Schieberegister

```

-- JK-FF -----
process (clk)
begin
  if (rising_edge(clk)) then
    if ((j='1') and (k='1')) then Qout <= not Qout;
    elsif ((j='1') and (k='0')) then Qout <= '1';
    elsif ((j='0') and (k='1')) then Qout <= '0';
    else Qout <= Qout;
    end if;
  end if;
end process;

-- cout8 : ein 8-Bit Zähler -----
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;          -- überladene Operatoren für Signale
entity Cout8 is
  port ( clk : in std_logic;        -- Clock
         cnt : inout unsigned (7 downto 0));   -- der Zählerausgang cnt darf nicht vom Typ out sein,
                                                -- da er rechts in einer Gleichung verwendet wird !
end Cout8;
architecture behavior of Cout8 is
begin
  process (clk)                  -- Nur bei einer Taktflanke
  begin
    if (rising_edge(clk)) then cnt <= cnt+1;  -- zählen
    end if;
  end process countit
end behavior;
-----
```



```

-- srg8 ein 8-Bit Schieberegister (...nur der process Kern) -----
process (clk)
begin
  if (rising_edge(clk)) then
    for i in MAX downto 1 loop
      sreg(i) <= sreg(i-1);
    end loop;
    sreg(0) <= din;
  end if;
end process;
```

9.7 Schrittschaltwerke in VHDL

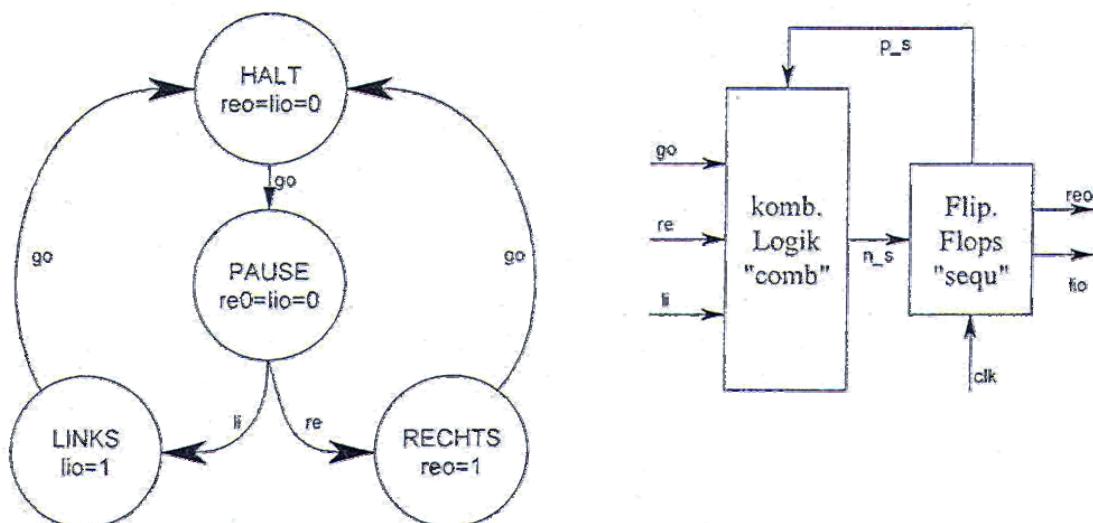
Viele Ablaufsteuerungen lassen sich mit Schrittschaltwerken (= Sequencer = Finite State Machine) lösen. Das Anwendungsspektrum geht vom DMA-Controller bis zum Schweißautomaten. Moore-Maschinen verwenden als Ausgangssignale direkt die Ausgänge der Zustands-FF oder eine kombinatorische Verknüpfung derselben. Bei Mealey-Maschinen hängen die Ausgangssignale noch zusätzlich von den Eingangssignalen ab.

Die Art der Implementierung kann durch den VHDL-Code beeinflusst werden – bei Strukturbeschreibungen wird sie sogar durch den VHDL-Code festgelegt. Meist jedoch schreibt man den VHDL-Code allgemein (problembezogen) und überlässt die Art der Implementierung den Fittern. Die meisten haben dazu Softwareschalter:

- „one hot“ Für jeden Zustand wird nur ein FF aktiv, es werden viele FF benötigt aber nur wenige Produktterme, liefert kurze Laufzeiten, eher für FPGA's.
- „aeramin“ Für minimalen Flächenbedarf, die Ausgänge werden von den Ausgängen der Zustands-FF aus decodiert, gut geeignet für CPLD
- „speedmin“ Für minimale Durchlaufzeit, die Ausgänge werden über eigene (getrennte) FF geführt.

Anweisungen an den Fitter können durch Attribute in der Entity-Sektion transportiert werden:
 ATTRI BUTE part_name of MyGate: ENTITY IS "C371"; -- CPLD Typ festlegen
 ATTRI BUTE pin_numbers of MyGate: ENTITY IS "a(0):5 a(1):3 a(2):2 a(3):6";
 -- PinNummern erzwingen
 ATTRI BUTE state_encoding of MyGate: TYPE IS one_hot_one; -- Impl. Art

Beispiel eines Schrittschaltwerkes (FSM1):



```

-- fsm1 : Finite State Machine in 2-Prozess Form -----
entity FSM1 is
    port ( go, li, re, clk : in bit;
           reo, lio : out bit);
end FSM1;
architecture state_machin of FSM1 is
    type tyState is (HALT, PAUSE, LINKS, RECHTS);
    signal p_s, n_s : tyState;
begin
    comb : process(p_s, go, li, re)
    begin
        case p_s is
            when HALT      => reo<='0'; lio<='0';
            if (go='1') then n_s <= PAUSE;
            else          n_s <= HALT;
            end if;
            when PAUSE     => reo<='0'; lio<='0';
            if (re='1') then n_s <= RECHTS;
            elsif (li='1') n_s <= LINKS;
            else          n_s <= PAUSE;
            end if;
            when RECHTS   => reo<='1'; lio<='0';
            if (go='1') then n_s <= HALT;
            else          n_s <= RECHTS;
            end if;
            when LINKS    => reo<='0'; lio<='1';
            if (go='1') then n_s <= HALT;
            else          n_s <= LINKS;
            end if;
        end case;
    end process comb;
    sequ : process(clk)
    begin
        if (clk'event and clk='1') then
            p_s <= n_s;
        end if;
    end process sequ;
end architecture state_machin;

```

9.8 VHDL Testbench

In der traditionellen Entwurfsmethode kann mit der Überprüfung der Funktionalität erst nach der Implementierung begonnen werden. VHDL-Modelle können schon sehr früh mit dem Simulator getestet werden. Bei komplexen Schaltungen ist es jedoch mühevoll viele Signalsequenzen immer wieder anzulegen bis man jenen Punkt erreicht, den man untersuchen möchte. VHDL erlaubt es „Testbenches“ zu schreiben. Eine Testbench besteht aus einem VHDL-Code, der die Eingangs-Vektoren beschreibt, einem VHDL-Code, der das Gerät beschreibt und einem VHDL-Code, der die erwarteten Ausgangsvektoren beschreibt. Das Ergebnis kann im Simulator betrachtet oder in einer Datei abgelegt werden. Dazu muss der Code für das Gerät mit Package in ein Component verpackt werden. Das sieht für einen 4-Bit Zähler so aus:

```

package MyCount is
    component Count4
        port (clk, rst : in std_logic;
              cnt : out std_logic_vector (3 downto 0));
    end component;
end myCount;

```

Nun kann der Zähler in eine Testbench eingefügt werden. Weil die Testbench keine Leitungen nach außen hin hat, gibt es in ihrer Entity kein port() Statement.

```
entity TestCount is
end TestCount;
```

In Architecture werden die Testsignale als ein Array von konstanten Testvektoren definiert und in einer For-Schleife der Reihe nach an das Gerät angelegt.

```
architecture Mytest of TestCount is
    signal clk, rst, : std_logic;
    ...
    -- zuerst einen Testvektor zusammenstellen
    type TyTestvect is record
        clk : std_logic;
        rst : std_logic;
        cnt : std_logic_vector (3 downto 0);
    end record;
    -- dann die gewünschten Testvektoren der Reihe nach in einem Array ablegen
    type TyTestvectAr is array (natural range <>) of TyTestvect;
    constant TestVector: TyTestvectAr := (
        -- Zähler zurücksetzen
        (clk=>'0', rst=>'1', cnt=>"0000"),
        (clk=>'1', rst=>'1', cnt=>"0000"),
        (clk=>'0', rst=>'1', cnt=>"0000"),
        -- Zähler betreiben
        (clk=>'1', rst=>'0', cnt=>"0001"),
        (clk=>'0', rst=>'0', cnt=>"0001"),
        (clk=>'1', rst=>'0', cnt=>"0010"),
        ...
        (clk=>'1', rst=>'0', cnt=>"0110"),
    );

```

Die Verdrahtung wird mit port map hergestellt.

```
begin
    uut : port map (clk=>clk, rst=>rst, cnt=>cnt); -- Verdrahtung
```

Zum Testen werden die Testvektoren der Reihe nach mit einer For-Schleife an das Gerät angelegt. Mit einem Flag wird überprüft, ob Fehler aufgetreten sind.

```
verify : process
    variable vector : TyTestvect;
    variable flag : boolean := false;
begin
    for i in TestVector'range loop
        vector:=TestVector(i);          -- Testvektor holen
        clk<=vector.clk;              -- Clock und Reset anlegen
        rst<=vector.rst;
        wait for 20 ns;                -- Laufzeit abwarten
        if cnt /= vector.cnt then      -- nachschauen ob alles ok war
            assert false report "FEHLER!";
            flag:=true;
        end if;
    end loop;
end process verify;
```