CMS - Zusammenfassung für Teilklausur I & II (Verilog)

- letzte Änderung: 25.07.11 14:50:18 -

02 - Verilog Überblick

Synthese: Programm in Hardware "bauen". Kann z.B. durch FPGA Board geschehen.
Sumulation: Quelltext in ISE Simulieren
Stimuli / Stimulation: Wenn werde bei Simulation an Kabel bzw. Eingänge angelegt werden.
- *Parallele Blöcke* (always und initial) werden in beliebiger Reihenfolge nacheinander simuliert. Anweisungen innerhalb eines begin/end-Blocks laufen immer in der hingeschriebenen Reihenfolge ab und zwar in der Regel ohne Unterbrechung (atomar).

```
module alu (
  input  wire  [2:0] OPCODE,
  input  wire [31:0] A,
                     B,
  output reg  [31:0] RESULT
);


`define ADD      3'b000  // 0      // nur zur Übung:
`define MUL     'b001    // 1      // Konstanten auf
`define AND      3'o2    // 2      // verschiedene Arten
`define LOGAND 3'h3      // 3
`define MOD      4       // 4
`define SHL      3'b101  // 5


  always @ (OPCODE, A, B)
    case (OPCODE)
      `ADD:     RESULT = A + B;
      `MUL:     RESULT = A * B;
      `AND:     RESULT = A & B;
      `LOGAND: RESULT = A && B;
      `MOD:     RESULT = A % B;
      `SHL:     RESULT = A << B;
      default:  $display ("Unimplemented Opcode: %d!", OPCODE);
    endcase


endmodule
```

- drei inputs mit verschiedenen *Bitbreiten* (3bit und 32bit). Lässt man die Angabe der Bit-Breite weg, so erhält man nur 1-bit-wire

- *define* um Konstanten zu initialisieren

- *always:* "immer wenn sich OPCODE, A oder B ändern, führe folgenden Block aus..." always@(COUNTER): Bie Änderung von COUNTER always@(*): bei alle Leseveriablen eines Blockes

- case(OPCODE): wenn OPCODE = ADD = 3'b000 ist, mache RESULT = A + B;

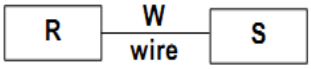
Mit # 10 zahl kann man eine angegebene Zeit (hier 10) Zeiteinheiten warten. # lässt sich jedoch nicht synthetisieren und hat nur Effekte während der Simulation. Dort wird es zur Erzeugung von Testsignalen wie z.B. des clock-signals benutzt. Zeiteinheit von # wird mittels 'timescale am Anfang des Verilog-Modells beschrieben. erster Paramter gibt Maß für 1 Zeiteinheit an, zweiter Parameter gibt die Auflösung in der Simulation an (bsp. 'timescale 1 ns / 1 ns | 'timescale 1ns / 1 ps)

```
if  (CONDITION) begin
  RESULT = 42
end else begin
  RESULT = 23;
end


// gleicher Effekt durch
RESULT = (CONDITION) ? 42 : 23;
```

Jedes bit kann die Werte 0, 1, x (unbekannt) und z (hochohmig) annehmen. ein reg kann Werte speichen (Flip-Flop), während ein wire nur Werte übertragen und nicht speichern kann.
Wird keine Angabe bei der Deklaration getroffen, so wird das angegebene Typ als vorzeichenlos interpretiert. Vorzeichenbehaftete Zahlen (z.B. Zahlen in Zweierkomplementdarstellung) müssen durch das Schlüsselwort *signed* gekennzeichnet werden (bsp.wire signed [7:0] op1). Konstenen durch s vor Kennung für Basis (bsp. 4'she -> 4b breit, signed, hexad., wert -2).
Wenn alle Teile eines Ausdrucks signed sind -> Ergebnis auch signed.
Wenn nur ein Teil unsigned ist -> Ergebis unsigned
Das Ergebnis wird abhängig von seiner Vorzeichenbehaftung auf Breite von Ziel aufgefüllt (unsigned -> mit nullbits, signed -> vorzeichenerweiterung. Stichwort: sign extension)
Implizite Konvertierung in vorzeichenlosen Typ durch Anwendung des Extraktionsoperator [msb:lsb]. Auch wenn man das gesamte Wort angibt (reg signed [7:0] ´data; ... = data[7:0]) ist die rechte Seite vorzeichenlos.

Explizite Konvertierung $signed(V) -> konvertiert v in vorzeichenbehafteten Typ
$unsigned(V) -> konvertiert v in vorzeichenlosen Typ
Was ist mit integer? Nicht für Synthese verwenden! Nur ungenau definiert (-> bitbreite hängt von CAD-Werkzeug ab). Jedoch trotzdem nützlich für Simulation (bsplw. als Schleifenzähler für for-schleife)

Verbindung zwischen Resgitern und Wires: Ein Wire verbindet ein (oder mehrere) Register oder Wires mit irgendetwas.
ständige Zuweisung (bsp. assign W1 = R1): wire wird durch ständige Zuweisung getrieben. ->"Draht W1 wird am Ausgang des Registers R1 festgelötet". Somit spiegelt W1 alle Änderungen von R1 wider.



Bei assign W=R und assign W=S ist W unbestimmt (x) wenn, R und S nicht den gleichen Wert haben und keiner von beiden hochohmig(z) ist

normale Zuweisung (bsp. R2 = W2 oder R2 <= W2): Register ändert Wert nur bei Ausführung der Zuweisung.
Felder von Variablen:

reg A[1:1000] oder reg A[1000:1]	-> Feld von 1000 Variablen, jede 1b breit
reg [15:0] B [1:1000]	->Feld von 1000 Variablen, jede 16b breit
result = A[500][8]	-> gibt des neunten bits der 500 Variablen aus
reg [15:0] B [1:100][1:200][1:300]	-> Feld von 6.000.000 Variablen, jede 16b breit
result = A[99][156][223][7]	-> Ausgabe des achten bits der ... Variablen

Operatoren:

+, -	: Kein Problem
*	: Kann sehr große Schaltungen nach sich ziehen, Abhängig von Zieltechnologie, Bei uns grundsätzlich okay. Datentyp signed beachten!
/, %	: In der Regel nicht synthetisierbar. Ausnahme: Division durch Zweierpotenz. in allen anderen Fällen Modul aus Bibliothek instiieren
==, !=	: Logische Gleichheit / Ungleichheit. Wenn beide Operanden einen Wert von 0 oder 1 haben. Liefere 1'b1 bei Gleichheit/Ungleichheit, sonst 1'b0. Falls ein Operand nicht 0 oder 1 ist, liefere 1'bx
===, !==	: Wörtliche Gleichheit / Ungleichheit. Liefere 1'b1, wenn beide Operanden gleich/ungleich sind, sonst 1'b0. Das gilt auch für Werte x und z. Nicht synthetisierbar, nur in Testrahmen sinnvoll
>, <, =>, <=	: Arithmetische Vergleiche. Falls ein Operand nicht 0 oder 1 ist, liefere 1'bx. Liefere 1'b1 wenn der Vergleich wahr ist, 1'b0 sinst. Beachte korrete Vorzeichenbehaftung der Operanden (signed)
!, &&, ||, ^	: Vergleichbar mit Operanden in Java. Beachte jedoch Hardware-Werte x und z.

Konkatenation: Zusammensetzen von Signalen zu größeren Einheiten {3'b100, 4'bxxzz, 2'ha}
ergibt 100_xxzz_10
Vervielfältigen von Signalen { 3 { 4'b1010 } } ergibt 12'b1010_1010_1010
Kombination der beiden Operatoren ist möglich { 4 { 2'b00, 2'b11} }
ergibt 16'b0011_0011_0011_0011
Logisches Shiften: $display ("%b", 8'b1111_0000 >> 4); 0000_1111
Arithmetisches Shiften: Erhält Vorzeichen beim Rechts-Shift mit >>>, <<< hingegen verhält sich wie <<
$display ("%b", 8'sb1111_0000 >>> 4); 1111_1111
$display ("%b", 8'sb1111_0000 <<< 1); 1110_0000
$display ("%b", 8'sb1111_0000 <<< 4); 0000_0000
Blockende Zuweisung ("=") wird immer zusammenhängend ausgeführt. Auch wenn sie eine Zeitkontrolle #n enthält. Wird zur Erzeugung von Stimuli in Simulation benutzt und in rein kombinatorischen Blöcken (ohne always(...)) in der Synthese Ablauf der blockenden Zuweisung
1. Lese aktuelle Werte von Variablen und werte Ausdruck auf rechter Seite aus
2. Warte evtl. mit # die angegebene Zeit ab
3. Übernehme Wert in Zuweisungsziel auf linker Seite
4. Mache mit nächster Anweisung weiter
Nichtblockende Zuweisung ("<=") wird immer in zwei Phasen getrennt ausgeführt. Wird in allen sequentiellen Blöcken der Synthese benutzt. Ablauf der nichtblockenden Zuweisung:

1. Lese aktuelle Werte von Variablen und werte Ausdruck auf rechter Seite aus,
merke Ergebnis
2. Mache sofort mit nächster Anweisung im Block weiter
3. Am Ende des Blockes
   - Übernehme gemerkte Werte in Zuweisungsziele auf linker Seite
   - Falls Zeitkontrolle: Verzögere obige Zuweisung auch noch (benutzen wir aber nicht!)

Niemals = und <= an eine Variable in einem Block mischen!

Systemfunktionen $display, $write: beide geben Text und formatierte Daten aus.

\n	neue Zeile
\t	Tabulator
\\	das Zeichen \
\"	Anführungszeichen
%%	das Zeichen %
%h, %H	Hexadezimalzahl
%d, %D	Dezimalzahl
%o, %O	Oktalzahl
%b, %B	Binärzahl
%f, %F	reelle Zahl
%c	einzelnes Zeichen
%s	Zeichenkette
%t	Zeit
%m	aktueller Modulname

$display gibt immer Zeilenvorschub am Ende aus
$write nicht
$display("Zur Zeit %t ist das A=%b und B=%d", $time, A, B);

Modulparameter mit parameter und defparam

```
module counter #(
  parameter Width = 8
) (
  input  wire                 CLOCK,
  output reg    [Width-1:0]   COUNT
);

   initial
     COUNT = 0;
  always @(posedge CLOCK)
     COUNT = COUNT + 1;
endmodule // counter

module main;
  defparam Counter1.Width = 3; // Parameter explizit definiert
  wire [Counter1.Width-1:0]  C1;
  wire [3:0]                 C2;
  reg                        CLOCK;
 ...
// Takterzeugung & $display C1, C2 weggelassen
 ...
  counter      Counter1(CLOCK, C1);
  counter #(4) Counter2(CLOCK, C2); // Parameter bei Instanzierung

endmodule // main
```

parameter: bei der Moduldefinition
defparam: bei der Instanziierung

Lesen von Speicherdateien aus Datei mittels $readmemh:

```
module readmemh_demo;

// Speicher
reg [31:0] Mem [0:11];

  // Lese Speicherdaten aus Datei
  initial
    $readmemh("data.txt",Mem);

  // Inhalt des Speichers anzeigen

  initial  begin : a_block
    integer k;
    $display("Inhalt_von_Mem:");
    for  (k=0; k<12; k=k+1)
      $display("%d:%h",k,Mem[k]);
  end

endmodule
```

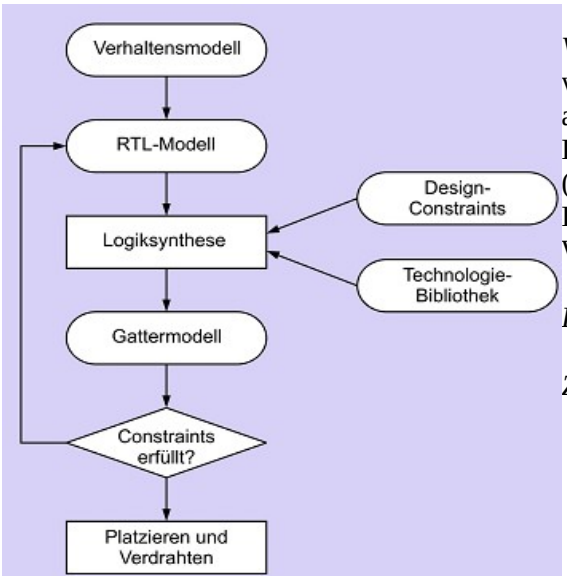
```
Inhalt von Mem:
 0:02328020
 1:02328022
 2:02328024
 3:02328025
 4:8e700002
 5:ae700001
 6:1232fffa
 7:1210fff9
 8:xxxxxxxx
 9:xxxxxxxx
10:xxxxxxxx
11:xxxxxxxx
```

Schreiben in eine Datei muss mit eingener Schleife implementiert werden.

$finish beedenet Simulatin sofort. Bei Xilinx ISE wird auch das Signaldiagramm geschlossen!
$stop schaltet Simulation in interaktiven Modus

04 - Einführung in die Logiksynthese

Logiksynthese: Abbildung von RTL(Register-Transfer-Logik) - Modell auf Gattermodell. Logiksynthese optimiert wesentlich Logik zwischen getakteten Registern, während die *High-Level-Synthese* oberhalb der RTL beginnt und über die Taktgrenzen hinaus optimiert.



Vorteile der Logiksynthese: Kürzere Entwurfszeiten, weniger fehleranfällig, Anforderungen an Zeit und Fläche aufstellbar, Portabilität zwischen verschiedenen Chip-Herstellern, Leichtere Exploration des Entwurfsraumes (Wieviel langsamer, wenn 25% kleiner?), Einheitlicher Entwurfsstil bei Team-Arbeit, Leichtere Wiederverwendung von (Teil-)Entwürfen

Design-Constraints:	Wie schnell? Wie groß? (Wie viel Energie?)
Zieltechnologie:	AND, OR, Addierer, Flip-Flops, Genaue Laufzeiten, genaue Flächenangaben

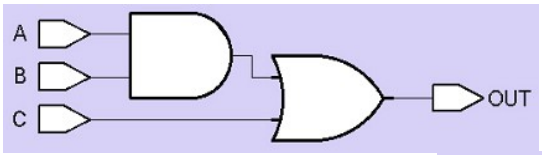
Wichtigste *synthetisierbare Verilog-Konstrukte:*

Signale und Variablen	wire, reg
prozedural	always, begin, end, if, else, case, function, task, =, <=
Struktur	modul, input, inout, output, parameter, assign eingeschränkt: for
arithmetisch	*, /, +, -, %
logisch	!, &&, ||
bit-weise	~, &, |, ^, ^~, ~^
Reduktion	&, ~&, |, ~|, ^, ^~, ~^

Relation	>, <, >=, <=	
Gleichheit	==, !=	**aber kein** === und !== mit x und z
shift	>>, <<<, <<<, >>>	
Konkatenation	{ }	
dedingt	?:	

Einige *nichtsynthetisierbare Verilog-Kontrukte:*
initial: Stattdessen explizites Reset-Verhalten beschreieben.
Zeitkontolle: # und @ innerhalb von Block. Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie:

Syntheseergebnisse:
Zunächst Abbildung auf allgemeines Gattermodell. Noch weitgehend ohne Berücksichtigung der Zieltechnologie, Reine Zwischendarstellung

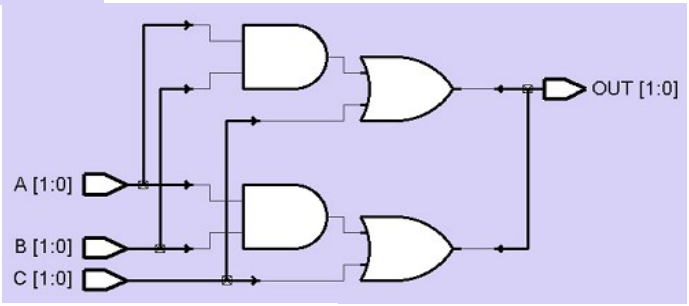


assign OUT = (A & B) | C;

Annahme: Alle Signale 1b breit.

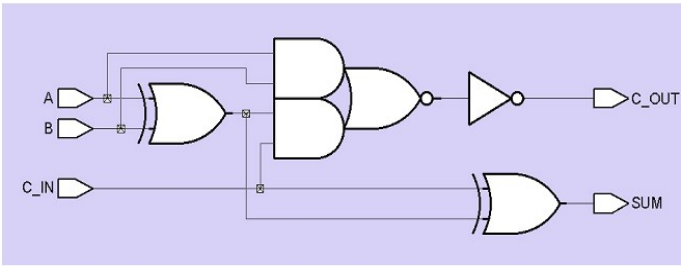
assign OUT = (A & B) | C;

Annahme: Alle Signale 2b breit.




```
assign {C_OUT, SUM} = A + B + C_IN
```

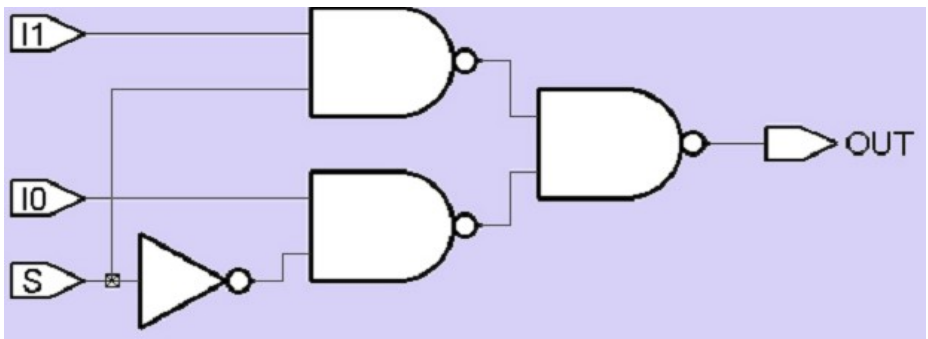
1b-Volladdierer



Merkwürdiges Gatter in der Mitte: AND-OR-INVERT (AOI),
sehr effizient in ASIC-Technologie realisierbar

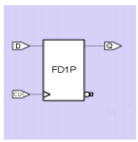
Alle folgenden drei Anweisung ergeben die selbe Gatterlogik:

```
assign OUT = (S) ? I1 : I0
---------------------------------
if(S) OUT = I1;
else  OUT = I0;
---------------------------------
case(S)

  0: OUT = I0;
  1: OUT = I1;
endcase
```

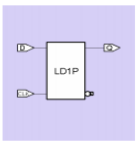


Synthese von Speicherelementen:

```
always @ (posedge CLK)
  Q <= D;
```
Vorderflankengesteuertes Flip-Flop



```
always @ (CLK or D)
  if  (CLK) Q = D;
```
Pegelgesteuertes Latch



Hardware-Register: (flankengesteuertes) Flip-Flop, (pegelgesteuertes) Latch !nicht identisch!
Verilog-Datentyp reg

flankengesteuerte always-Blöcke:
always@(posedge CLK) | always@(posedge CLK, negedge nRESET)
flankenfreie always-Blöcke:
always@(CLK, D) | always@(A,B,C_IN)

Grundlegende wichtige Definitionen:
Potenzielle Register: Eine Variable Q ist ein potenzielles Register (PR), wenn sie in einem always-Block geschrieben wird.
Vollständig: Ein potenzielles Register ist vollständig, falls es bei jedem Durchlauf des always-Blocks nicht-redundant (nicht mit sich selber, also nich Q = Q) geschrieben wird.
räumlich Lokal: Ein potenzielles Register ist räumlich lokal, wenn es nur innerhalb **eines** always-Blocks verwendet wird (lesend oder schreibend). Zu ausserhalb des always-Blocks zähöt auch die modul-Schnittstelle
zeitliche Lokalität: Ein potenzielles Register ist zeitlich lokal (kurz: lokal), falls es räumlich lokal ist und nie vor dem Schreiben gelesen wird.

```
always@(CLK, D)
  if (CLK) Q=D;
```
ergibt ein Latch, da Q unvollständig ist

```
always@(CLK, D)
  if (CLK) Q=D;
  else     Q=0;
```
Latch vermieden da Q nun vollständig

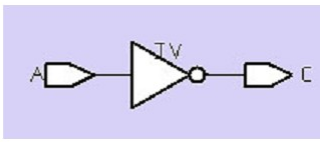
```
always@(A,B)
  if(A) C=B;
  else  C=0;
```
Signalnamen wie CLK irrelevant. Kombinatorische Logik

```
module decoder (
  input  wire [3:0]  I ;
  output reg  [9:0]  DECIMAL);
always @(I)
  case (I)
    4'h0: DECIMAL = 10'b0000000001;
    4'h1: DECIMAL = 10'b0000000010;
    4'h2: DECIMAL = 10'b0000000100;
    4'h3: DECIMAL = 10'b0000001000;
    4'h4: DECIMAL = 10'b0000010000;
    4'h5: DECIMAL = 10'b0000100000;
    4'h6: DECIMAL = 10'b0001000000;
    4'h7: DECIMAL = 10'b0010000000;
    4'h8: DECIMAL = 10'b0100000000;
    4'h9: DECIMAL = 10'b1000000000;
  endcase
endmodule
```

Ein vollständiges potenzielles Register erzeugt zunächst ein Latch. Ist das PR jedoch lokal, wird das Latch aber anschließend wegoptimiert.
Damit PR kein Latch wird: ->PR vollständig beschreiben
-> order PR nur lokal verwenden

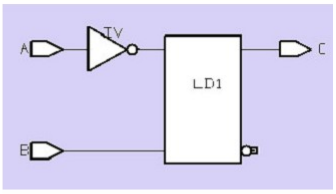
- Latch wegen unvollständigem `case`
- Wie vollständig formulieren?
- Durch Angabe von `default`

```
module z_lokal (
  input  wire A,
  output reg  C);

reg TMP;

always @ (A, TMP)
begin
  TMP = ~A;
  C   = TMP;
end
endmodule
```



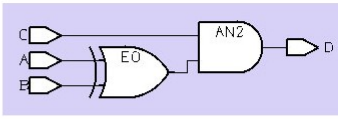
Latch wird vermieden, da TMP vollständig ist (und zeitlich lokal).

```
module lokal (
  input  wire A, B,
  output reg C);

reg TMP;

always @ (A, B, TMP)
begin
  if  (B) TMP = ~A;
  C   = TMP;
end
endmodule
```



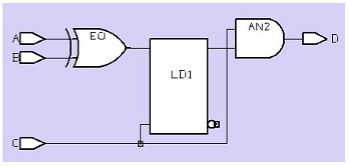
Latch entsteht, da TMP unvollständig ist und nicht zeitlich lokal.

```
module lokal(
  input  wire A, B, C,
  output reg  D);

reg TMP;

always @ (TMP, A, B, C)
  if  (C) begin
    TMP = A + B;
    D = TMP;
  end
  else D = 0;

endmodule
```



Latch wird vermieden, da TMP zwar unvollständig ist, aber räumlich und zeitlich lokal.

```
module lokal (
  input  wire A, B, C,
  output reg  D);

reg TMP;

always @ (TMP, A, B, C)
  if  (C) begin
    D = TMP;
    TMP = A + B;
  end
  else D = 0;

endmodule
```



Latch entsteht, da TMP unvollständig ist und nur räumlich, nicht aber zeitlich lokal ist.

Flankenfreie always-Blöcke
Richtlinien zur Vermeidung von Latches:
- PR vollständig beschreiben
- oder nur lokal verwenden

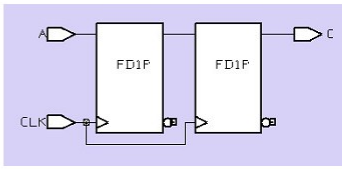
Verlilog-Funktionen ergeben kein Latch
-Sie haben keinen internen Zustand
-Keine globalen oder static-Variablen wie z.B. in Java

In flankenfreien always-Blöcken immer alle Lesevariablen in Aktivierungsliste
-always@(*)

Immer die blockende Zuweisung = verwenden

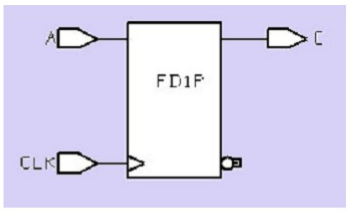
Gatakteter always-Block
Mit posedge oder negedge in der Aktivierungsliste. Jedes nicht-lokale potenzielle Register wird Flip-Flop. Vollständigkeit ist nun irrelevant.

```
module lokal (
  input  wire CLK, A,
  output reg  C);

reg B;

always @ (posedge CLK)
begin
  B <= A;
  C <= B;
end

endmodule
```



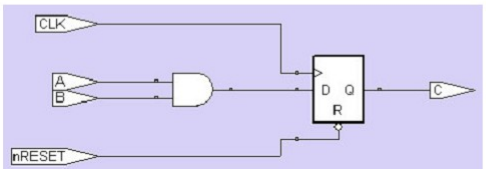
Für B entsteht ein Flip-Flop, da B zwar räumlich, nicht aber zeitlich lokal ist.

```
module lokal (
  input  wire CLK, A,
  output reg  C);

reg B;

always @ (posedge CLK)
begin
  B  = A;
  C <= B;
end

endmodule
```



Für B entsteht kein Flip-Flop, da B räumlich und zeitlich lokal ist.

```
module FF (
  input  wire CLK, nRESET, A, B,
  output reg  C);

always @(  posedge CLK,
           negedge nRESET)
  if  (!nRESET)   C <= 0;
  else            C <= A & B;

endmodule
```



Flip-Flop mit asynchronem Reset

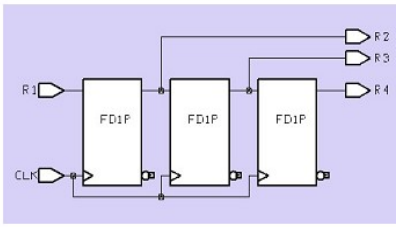
Richtlinien für Zuweisungen:
Für spätere Flip-Flops: nichtblockende Zuweisung
Für kombinatorische Logik, lokale Hilfsvariablen und Latches: blockende Zuweisung
Für die Synthese sind die Zuweisungen unerheblich. Es gelten die Regeln von Vollständigkeit und Lokalität. Trotzdem sollte man die Richtlinien befolgen, da sonst die Prä-Synthese und Post-Synthese Symulation nur schwer vergleichbar sind.

So nicht: Nichtblockende Zuweisung:

```
always @ (A, B)
begin
  C <= 0;
  if  (B) C <= A;
end
```

Trotz <= kombinatorische Logik, da C vollständig beschrieben und lokal verwendet.

```
always @ (posedge CLK)
  R2 = R1;

always @ (posedge CLK)
  R3 = R2;

always @ (posedge CLK)
  R4 = R3;
```



- In Simulation: Zufällige Ausführungsreihenfolge
  - R2=, R3=, R4 oder R4=, R2=, R3= . . .
- In Synthese: Immer Schieberegister aus Flip-Flops
  - R2, R3, R4 nicht nur lokal verwendet

Trennung von kombinatorischer Logik und Register:

1. Möglichkeit: Getrennte always-Blöcke.
2. Möglichkeit: Logik als Funktion
3 .Möglichkeit: Keine saubere Trennung. Kürzer, aber nicht mehr sauber getrennt. Nachteil: Wird bei Erweiterung leicht unübersichtlich. In der Industrie verpönt, dort i.d.R. strikte Trennung

Zusammenfassung PR:

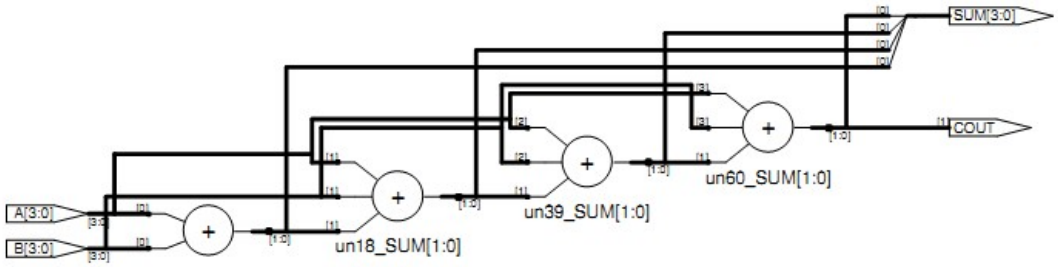
- getakteter always-Block: always @(posedge CLK). . .
- flankenfreier always-Block: always @(CLK, D). . .
- PR ist potenzielles Register, falls PR in einem always-Block geschrieben wird
- PR ist vollständig, wenn es in jedem Durchlauf eines always-Blockes
- geschrieben wird
- PR ist räumlich lokal, wenn es nur innerhalb eines always-Blockes auftritt
- Ein räumlich lokales PR ist zeitlich lokal (kurz: lokal), falls es nie vor dem Schreiben gelesen wird

Getaktete always-Blöcke	Flankenfreie always-Blöcke
- Jedes nicht-lokale PR wird ein getaktetes Flip-Flop - An solche Flip-Flops wird mit <= zugewiesen - An kombinatorische Hilfsvariablen mit = - Spätere Flip-Flops werden an nur einer Stelle geschrieben: Ausgenommen der Reset, der steht extra - Jedes Flip-Flop bekommt bei Reset einen definierten Wert	- Ein vollständiges oder lokales PR wird kombinatorische Logik - Ein nicht-lokales und unvollständiges PR wird ein Latch - Es wird stets blockend mit = zugewiesen - Die Aktivierungsliste enthält alle Lesevariablen des always-Blockes - Ein Takt in der Aktivierungsliste wird in der Synthese wie jede Variable auch behandelt

Synthese von for-Anweisung

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich "ausgerollt"
  - Parallele Abarbeitung

```
module unrolled_for (input      [3:0] A, B,
                     output reg [3:0] SUM,
                     output reg       COUT);
integer I;
reg C;

always @(*) begin
  C = 0;
  for (I = 0; I < 4; I = I + 1) begin
    {C, SUM[I]} = A[I] + B[I] + C;
  end
  COUT = C;
end
```



Besser: generate / genvar-Anweisung verwenden:

```
module generated_arraya_pipeline(data_out,data_in,clk,reset);
 parameter width = 8;
 parameter length = 16;
 output [width−1:0] data_out;
 input  [width−1:0] data_in;
 input clk,reset;
 reg [width−1:0] pipe [0:length−1];
 wire [width−1:0] d_in [0:length−1];
 assign d_in[0] = data_in;
 assign data_out = pipe[length−1];
 generate
  genvar k;
  for (k=1;k<=length−1;k=k+1) begin: W
   assign d_in[k] = pipe[k−1]; end
 endgenerate
 generate
  genvar j;
  for (j=0;j<=length−1;j=j+1)
   begin: stage
    always @(posedge clk or negedge reset) begin
    if (reset == 0) pipe[j] <= 0; else pipe[j] <= d_in[j]; end
   end
 endgenerate
endmodule
```

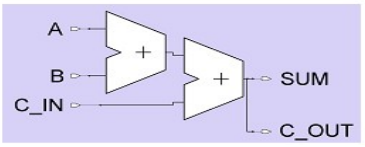
Man kann auch ausserhalb des generate - endgenerate Blockes mittels genvar eine Variable initialisieren. Somit kann man diese in allen folgenden generate - endgenerate Blöcken nutzen

Geht aber noch einfacher :

```
module top_pads3 (pdata, paddr, pctl1, pctl2,   pctl3, pclk);
  input [15:0] pdata;                    // pad data bus
  inout [31:0] paddr;                    // pad addr bus
  wire  [15:0] data;                     // data bus
  wire  [31:0] addr;                     // addr bus
  wire         wr;                       // Schreibsignal (gibt addr auf paddr−Pads aus)
                                         // Array−Instanznamen
                                         // i[15] bis i[0]
  IBUF  i[15:0]  (.O(data), .pI(pdata));
  BIDIR b[31:0] (.N2(addr), .pN1(paddr), .WR(wr));        // Array−Instanznamen
                                                          // b[31] bis b[0]
endmodule
```

Verfeinerter Entwurfsablauf der Synthese

Unoptimierte Zwischendarstellung eines 1b-Addierers



Je nach Zieltechnologie wird aus der unoptimierten Zwischendarstellung eine optimierte Darstellung erstellt.

Design-Contraints:

Zeit: Timing-Analyse, geschätzt nach Synthese (ohne Verdrahtungsverzögerung), Exakt nach Platzieren und Verdrahten

Fläche:

Geschätzt nach Synthese (ohne Verdrahtungsfläche!). Exakt nach Platzieren und Verdrahten

Elektrische Leistungsaufnahme:

Simulation auf Layout-Ebene, Bestimmung von Umschaltfrequenzen von Signalen

Beispiel: 4b-Vergleicher:

- Größenvergleich von zwei 4b breiten Eingabewerten A und B
- Bestimmt Flags für <, >, und =
- Erstes Ziel: Möglichst schnelle Schaltung, Fläche egal

```
// Vergleicher
module mag_comp (
  input  wire [3:0]  A, B,
  output wire        A_GT_B, A_LT_B, A_EQ_B);

assign A_GT_B = (A > B);       // A groesser B
assign A_LT_B = (A < B);       // A kleiner B
assign A_EQ_B = (A == B);      // A gleich B

endmodule
```

Prä-Synthese

```
 0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Post-Synthese

```
 0 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=x
 3 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=0
 6 A=10, B= 9, A_GT_B=x, A_LT_B=0, A_EQ_B=0
 8 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=1, A_LT_B=0, A_EQ_B=0
16 A=14, B=15, A_GT_B=1, A_LT_B=1, A_EQ_B=0
18 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=0
23 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=1
28 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=1
32 A= 8, B=12, A_GT_B=1, A_LT_B=0, A_EQ_B=0
34 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=0
35 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
53 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Annahme: Jedes Gatter hat 1 Zeiteinheit Verzögerung

Diskussion: Prä- ./. Post-Synthese-Simulation

- Unterschiedlich viele Ergebnisse
- Verschiedene Werte
- Unterschiedliche Zeiten
  - vorher gar keine ausser den im Testrahmen
- Manche Ergebnisse schlicht falsch (z.B. bei t=32)
- Interpretation nötig

  “Wenn man lange genug wartet, ist das Ergebnis richtig”!
- Was ist “lange genug”?
- Antwort: Kritischer Pfad (TGDI)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Weitere Verfeinerung der Verifikation

Post-Layout-Simulation schliesst ein
- Gatterverzögerungen
- Leitungsverzögerung
- Kann umfassen: Widerstände, Kapazitäten, Induktivitäten

Synthesebeispiel: Zero-Counter

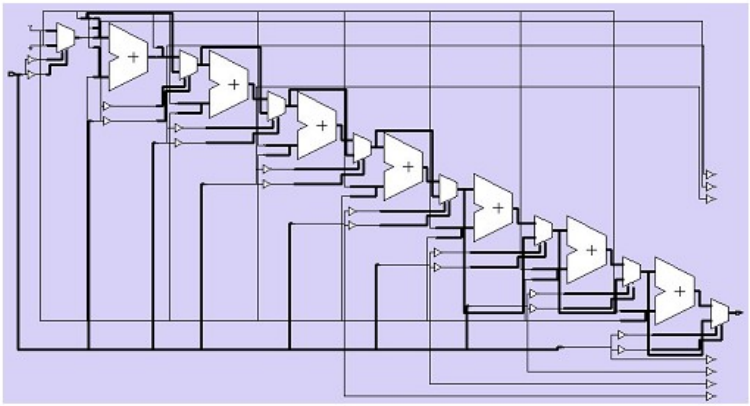
Spezifikation
- Eingabe ist ein 8b Datenwort
- Ausgabe soll sein die Anzahl der Null-Bits in der Eingabe

Genauer betrachtet
- RTL-Modell kann Synthese-Ergebnis direkt beeinflussen
- Wirkung von Design-Constraints

Nicht mehr so relevant
- Konkrete Umsetzung in Gatter-Modell
- Bei größeren Schaltungen oft schlicht zu unübersichtlich

```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird



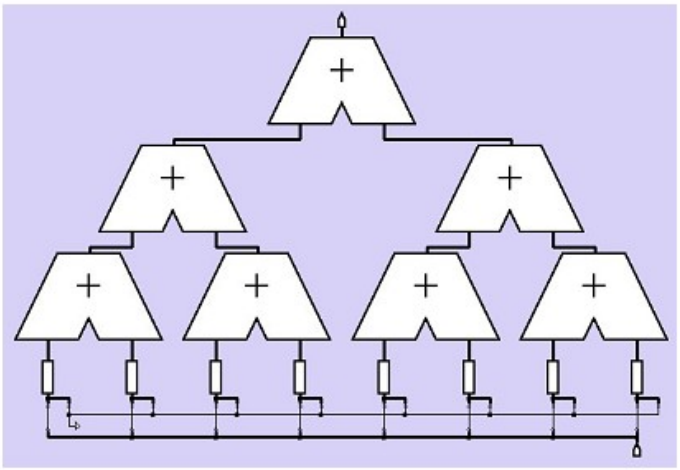
```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

integer      I;

always @(IN) begin
  OUT = ~IN[0];
  for (I=1; I<8; I=I+1)
    OUT = OUT + ~IN[I];
end

endmodule
```

Hierbei entfallen die Multiplexer

Schlaue Lösung, da kritischer Pfad viel kürzer wird:

```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

always @(IN)
  OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))
      + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));

endmodule
```

- Bits werden direkt aufaddiert
- Jetzt aber hierarchische Klammerung
- Damit parallele Berechnung
- Addierer-Baum

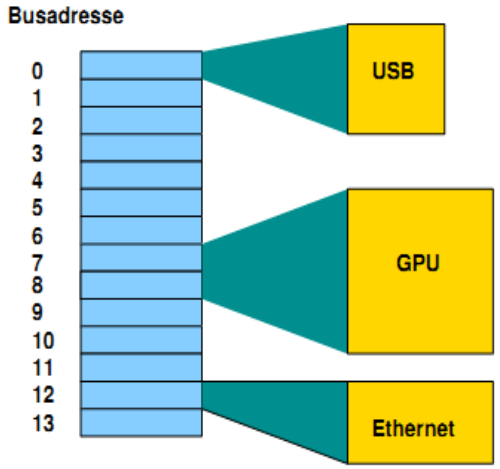


Einfluss von Design-Contrains:
Festlegen unterschiedlicher Optimierungsziele
Üblich: Flächenbedarf, Geschwindigkeit. Diese Optimierungsziele können in den Synthesewekzeugen eingestellt werde.
Seltener: Energieverbrauch und Ausfallsicherheit

Teil II

05 - CRT-Controller, Optimierung und Busse

Kommunikation und Adressierung:



-> Select-Signale werden aus Busadressen erzeugt
-> Gleichzeitig darf max. ein Select-Signal aktiv sein
-> Adressebereiche müssen überlappungsfrei sein
-> Erstelle Adressdekodierlogik durch Aufbaue eines Entscheidungsbaumes
    -> von msb zu lsb der Adressbits
    -> möglichst wenige Adressbits hierbei nutzen
-> Dabei darf derselbe Adressbereich i.d.R. mehrfach auftauschen (aliasing)
-> Er muss aber mindestens an den spezifizierten Adressen erreichbar sein

Startaddresse	Endaddresse	Teilnehmer
0x0000	0x1FFF	RAM 8KB
0x2000	0x23FF	Flash-Speicher 1KB
0xFE00	0xFFFF	Modem 512B

Startaddresse	Endaddresse	Teilnehmer
16'b0000_0000_0000_0000	16'b0001_1111_1111_1111	RAM 8KB
16'b0010_0000_0000_0000	16'b0010_0011_1111_1111	Flash-Speicher 1KB
16'b1111_1110_0000_0000	16'b1111_1111_1111_1111	Modem 512B

```
module decoder2 (
  input [15:0]  ADDR,
  output        SEL_RAM, SEL_FLASH, SEL_MODEM
);

assign SEL_RAM     = ~ADDR[15] & ~ADDR[13];
assign SEL_FLASH   = ~ADDR[15] &  ADDR[13];
assign SEL_MODEM =  ADDR[15];
```

Decoder:
- Ziel: Teilnehmer mit möglichst wenigen Bits identifizieren.
- Beginne mit höchstwertigen Bits

```
module rom1kx8 (
  input         SELECT,
  input  [9:0]  ADDR,
  output [7:0]  DATA
);

  reg [7:0]  MEM [0:1023]

  assign DATA = (SELECT) ? MEM[ADDR] : 8'bz;

  initial  begin          // einige Beispieldaten eintragen
    MEM[0]    = 8'h42;
    MEM[1]    = 8'h23;
     ...
    MEM[1022] = 8'h20;
    MEM[1023] = 8'h07;
  end

endmodule
```

```
module modem (
  input         CLOCK,
  input         SELECT,
  input  [8:0]  ADDR,
  input         WRITE,
  inout  [7:0]  DATA
);

  reg [7:0]  baudrate;
  reg [1:0]  parity ;
  reg [7:0]  inchar, outchar;

  assign DATA =  (~SELECT | WRITE) ? 8'bz :
                 ((ADDR==0) ? baudrate :
                  (ADDR==1) ? {6'b0,parity} :
                  (ADDR==2) ? inchar // <-- ADDR=2 liest Zeichen
                  : 8'h42);          // <-- Default-Wert für Debugging

  always @(posedge CLOCK) begin
    if (WRITE)
      case (ADDR)
        0 : baudrate  <= DATA;
        1 : parity    <= DATA[1:0];
        2 : outchar   <= DATA;          // <-- ADDR=2 schreibt Zeichen
      endcase
  end

endmodule

module mysystem;
 ...

  wire [15:0]  ADDR;
  wire  [7:0]  DATA;
  wire         SEL_RAM, SEL_FLASH, SEL_MODEM;

  // Adressdecoder
  decoder2  DECODER (ADDR, SEL_RAM, SEL_FLASH, SEL_MODEM);

  // Flash-ROM
  rom1kx8   FLASH     (SEL_FLASH, ADDR[9:0], DATA);

 ...
endmodule
```

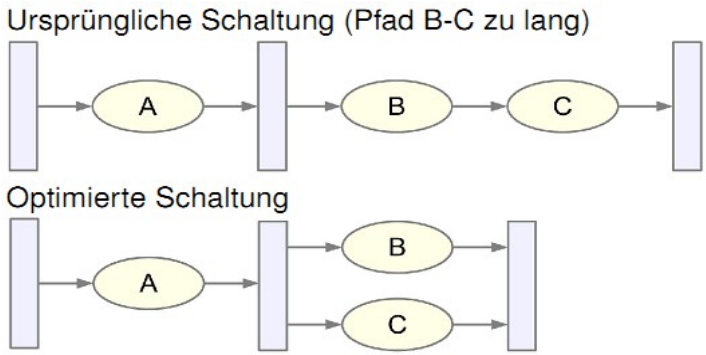
Memory Aliasing:
- Speicherbereich wiederholt sich
- Es sind aber immer die gleichen Daten
- Sichtbarkeit der gleichen lokalen Adressen an unterschiedlichen Busadressen: Aliasing
- Schadet in vielen Fällen nicht

Bsp: Flash von oben:
Adressbereich:
16b'**0**0**1**0_0000_0000_0000 - 16b'**0**0**1**0_0011_1111_1111 entspricht 16h'2000 - 16h'23FF
Select-Signal für Flash gdw. Bit[15] == 0 && Bit[13] == 1

Adresse 16'h2400 erzeugt nun Select-Signal für Flash, da gilt:
16'h2400 -> 16'b**0**0**1**0_0100_0000_0000

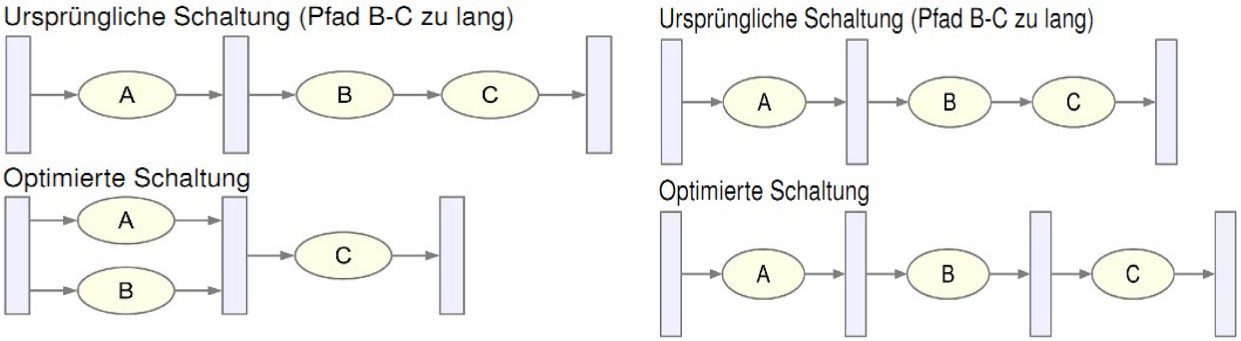
Optimierung:

Möglichkeiten der Logik-Synthese:



-> Parallelisieren von Berechnungen
-> Sequentielles Verhalten bleibt unverändert

Möglichkeiten der High-Level-Synthese



Links: Vorziehen von Berechnungen über Taktgrenzen hinweg. Auch hier: Sequentielles Verhalten unverändert.
Rechts: Aufteilen von langen Berechnungen über mehrere Takte. Nun ist jedoch das sequentielle Verhalten geändert.

-> Welche nehmen? Wir wollen manuell einen möglichst kleinen Eingriff durchführen. Also versuche, sequentielles Verhalten der Signale beizubehalten und Berechnungen zu verschieben.
Lösungsidee: Neue Register einbauen, aber Ergebnise einen Takt im voraus berechnen, damit das von aussen sichtbare sequentielle Verhalten gleich bleibt.

Modellierung von Speicher:

```
module rom16x4 (ROM_data, ROM_addr);
 output [3:0]  ROM_data;
 input  [3:0]  ROM_addr;
 reg    [3:0]  ROM [15:0];

 assign ROM_data = ROM[ROM_addr];

 // für Simulation
 initial  $readmemh("ROM-2b-Adder.txt",ROM,0,15);
endmodule
```

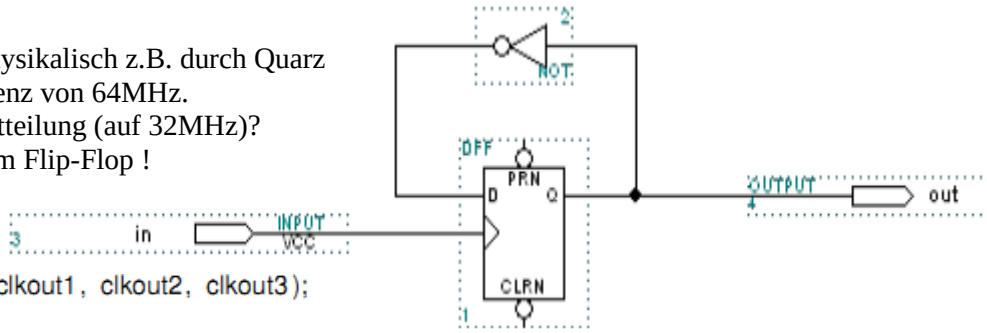
```
module sram16x8 (input [3:0] address,
                 input nCS, nWE, nOE,
                 inout [7:0] data);

 reg [7:0] memory [15:0]; // 16 Zellen, 8 Bit breit

 assign data = (!nCS && !nOE) ? memory[address] : 8'bZ;

 always @(nCS or nWE)
  if (!nCS && !nWE) memory [address] = data;

endmodule
```

```
module dram256x8 (input [3:0] address,
                  input nRAS, nCAS, nWE, nOE,
                  inout [7:0] data);

 reg [7:0] memory [15:0][15:0]; // 16x16 Zellen, 8 Bit breit
 reg [3:0] row, column;

 assign data = (!nOE) ? memory[row][column] : 8'bZ;

 always @(negedge nRAS)
  row <= address;

 always @(negedge nCAS)
  column <= address;

 always @(negedge nWE)
  memory [row][column] <= data;

endmodule
```

Takterzeugung:

- Takterzeugung erfolgt physikalisch z.B. durch Quarz
- Bsp. Quarz hat Takfrequenz von 64MHz.

Wie erreicht man eine Taktteilung (auf 32MHz)?
-> Mittels rückgekoppeltem Flip-Flop !




```
module taktteiler( clkin , clkout1, clkout2, clkout3);

  input clkin ;
  output clkout1, clkout2, clkout3;

  reg [24:0] counter;

  assign clkout1 = counter[24];
  assign clkout2 = counter[23];
  assign clkout3 = counter[18];

  always @(posedge clkin) begin
    counter <= counter + 1;
  end

endmodule
```

Wie erreicht man einen Takt von 125kHz mit einem Quarz, der 64MHz leistet?
Naiv mit neun rückgekoppelten Flip-Flops.
Besser: Zähler mit verschiedenen Abgriffen.
Verschiedene Taktfrequenzen ergeben sich durch unterschiedliche Abgriffe.
Jeweils geteilt durch entsprechende Zweierpotenz.

06 - Systematischer Schaltungsentwurf

Aus Algorithmus Hardware basteln (vgl. Beispiel Fakultätsberechnung):

1. Beschreibe Algorithmus in Pseudo-Code
   -> Wie beim Programmieren von Software
2. Schreibe Pseudo-Code in RTL-Beschreibung um
   -> Keine for, while-Schleifen, Prozeduraufrufe
   -> Aber Sprünge und if/then/else sind zugelassen!
   -> Nur noch Konstrukte vergleichbar synthetisierbarem Verilog
   -> Aber hier noch kein Verilog selbst erforderlich
3. Entwerfe Datenpfad-Struktur
   -> Basierend auf Operationen in RTL-Beschreibung
4. Entwerfe Zustandsmaschine für Steuerwerk
   -> auf Basis der RTL-Beschreibung
5. Realisiere Logik für Zustandsmaschine
   -> Kann von Logiksyntheseübernommen werden
   -> Schauen wir uns hier aber genauer an
   -> Datenpfad und Steuerwerk werden nun in Verilog beschrieben, wobei Datenpfad und Steuerwerk jeweils ein Modul werden (streng getrennt!). Hauptmodul instanziert dann beide anderen Module.

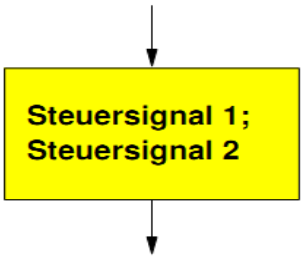
Graphische Beschreibung:

ASM(D)-Charts (Algorithmic State Machine (and Datapath)):

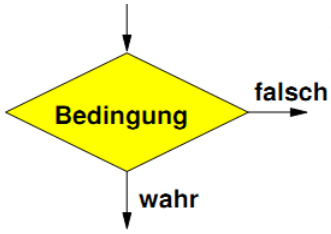
-> ASM-Chart stellt nur das Steuerwerk dar

-> ASMD-Chart enthält zusätzliche zum Steuerwerk auch noch die Datenpfadoperationen.

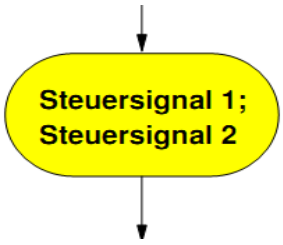
ASM-Chart:



- Zustand
- Zählt aktive Steuersignalee auf
- Nicht aufgeführte Signale sind inaktiv
- Die aufgeführten Signale sind parallel aktiv

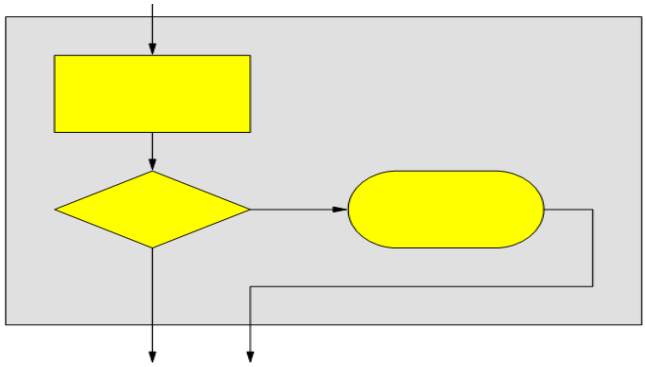


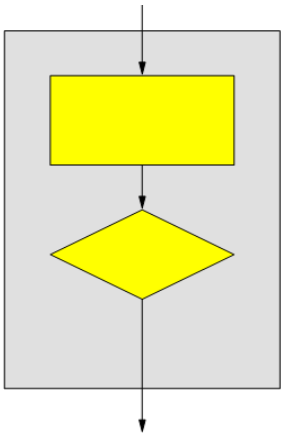
- Enthält logische Bedingung für Zustandsübergang
- Ausgangskanten müssen eindeutig beschriftet sein
  - True, 1, wahr
  - False, 0, falsch



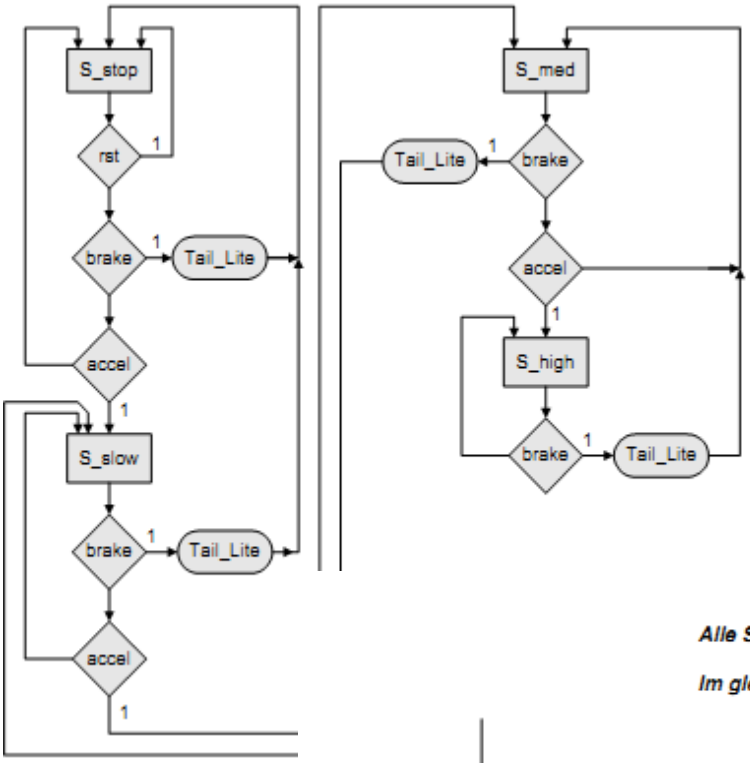
- Kann nur auf einen bedingten Übergang folgen
- Aktiviert Steuersignale abhängig von voriger Bedingung
- Beschreibung der Steuersignale wie bei Zustand

Links Mealy-Block / rechts Moore-Block



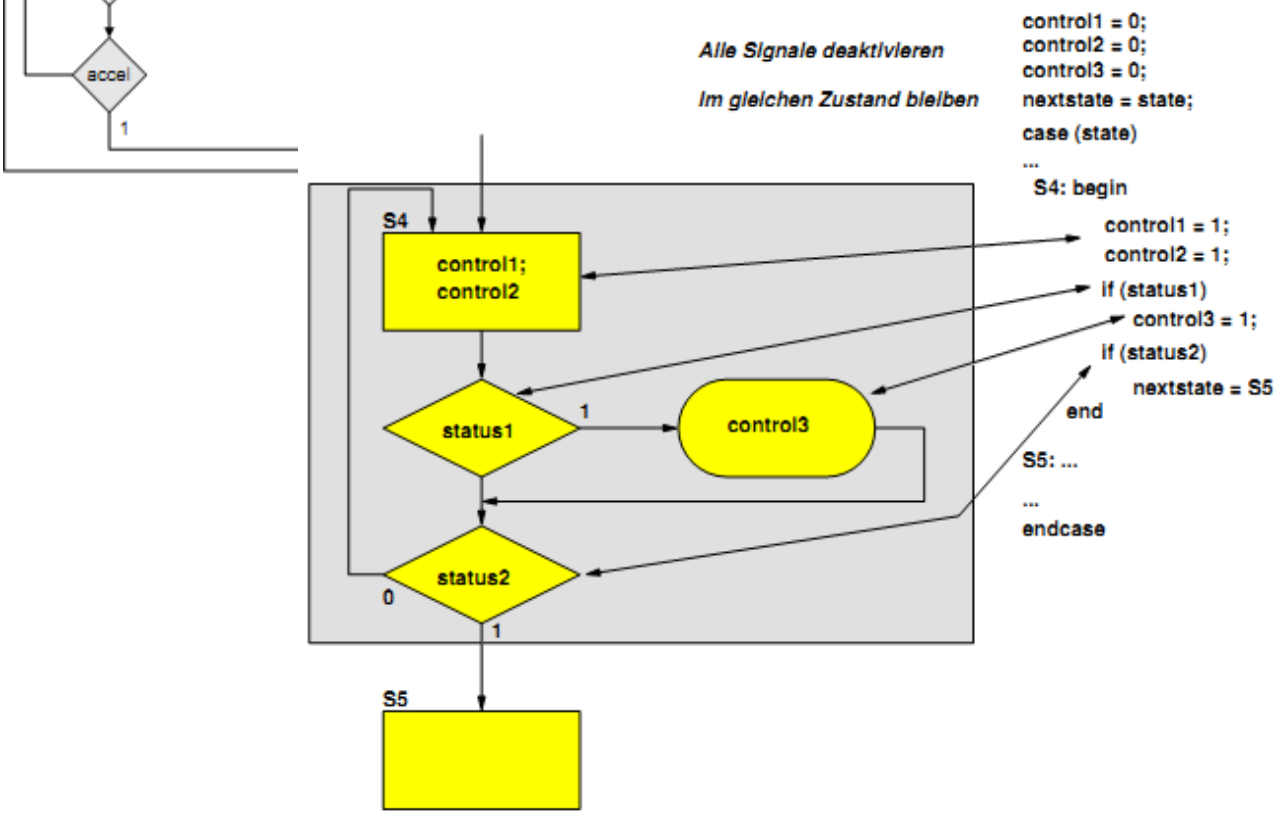


Bsp: Steuergerät von Auto in ASM-Chart:



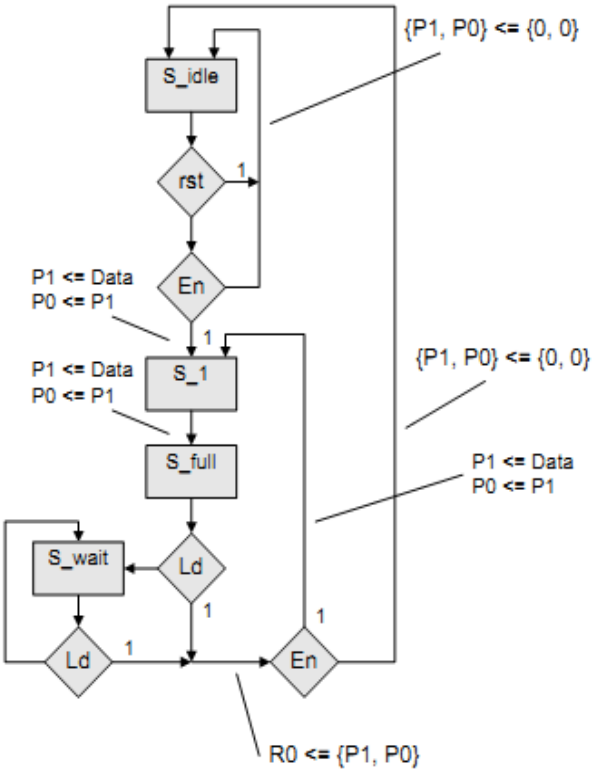
-> Bedingte Übergänge sind priorisiert und werden in der Reihenfolge ihrer Auswertung hingeschrieben (Bremse vor Gas).
-> Konvention: Reset-Übergang nur einmal darstellen. Implizit wird also aus jedem Zustand bei einem auftrettenden Reset in den angegebenen Zustand (hier S-stop) gewechselt.
Vorteil: Sehr ähnlich zu Flussdiagrammen. Wenn Algorithmus leicht als Flussdiagramm darstellbar ist, so ist er auch leicht als ASM-Chart darstellbar.

ASM-Chart -> Verilog:

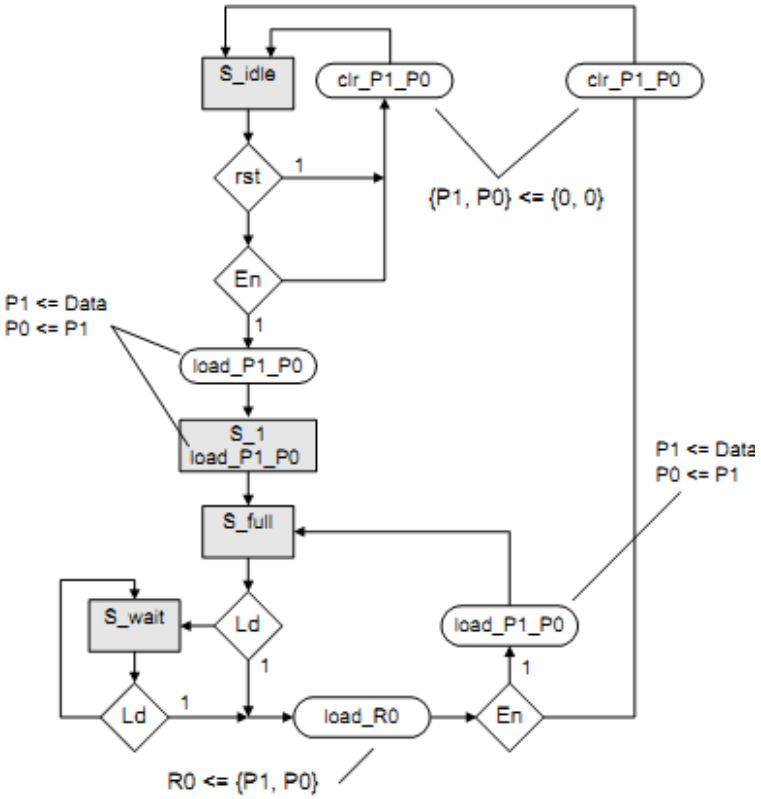


ASM(D)-Chart:

- Kanten werden mit Datenpfadannotation makiert.



ohne Steuersignale



mit Steuersignalen