Kap.1 Hardwareentwurf

Enwurfsautomatisierung
EDA
electronic design automation

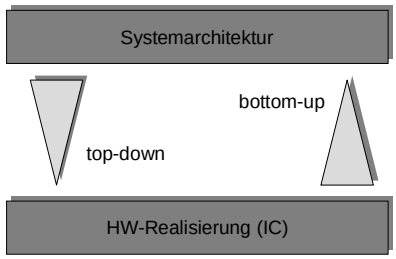
RA Überblick

- Einleitung
- Hardwareentwurf
- Befehlsschnittstelle
- Mikroarchitektur
- Speicherarchitektur
- Ein-/Ausgabe
- Multiprozessorsysteme, ...

1 Hardwareentwurf

- **1.1** Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwareverifikation
- **1.4** Hardwaresynthese
- **1.5** Technologieabbildung

Von der Architektur zur HW



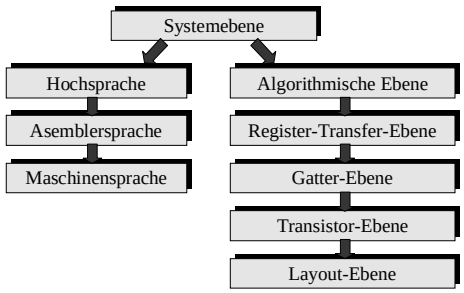
Entwurfsweisen

- top-down
  - abstrakte Beschreibung des Systems wird sukkzessive verfeinert bis zur HW
- bottom-up
  - bereits entworfene Komponenten werden zu komplexeren Einheiten zusammengefügt

Realität:

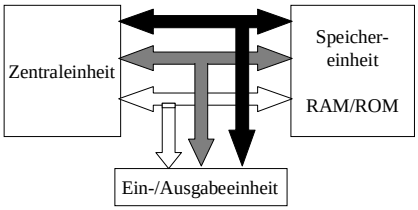
Michung aus beiden Prozessen

Abstraktionsebenen



Systemebene

- Funktionale Einheiten die miteinander kommunizieren



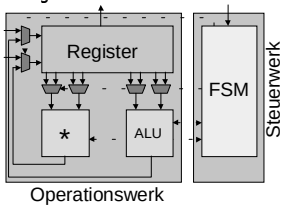
Algorithmische Ebene

- Die Funktionen einzelner Blöcke wird durch Algorithmen in einer HW-Beschreibungssprache spezifiziert

```
int dgl(int x,y,u,dx,a) {
  int x1, y1, u1;
  do
    x1 = x+dx;
    u1 = u-(3*x*u*dx)-(3*y*dx);
    y1 = y+(u * dx);
    x = x1; u = u1; y = y1;
  while (x1<=a);
  return y;
}
```

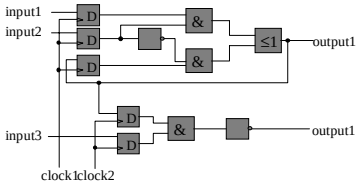

Register-Transfer Ebene

- Darstellung der Funktionalen Einheiten durch Datenpfad- und Kontrollpfad (Daten werden von Register zu Register transferiert und dabei Verarbeitet)



Gatterebene

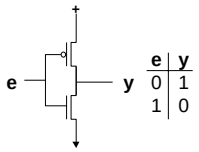
- Es gibt nur noch Boole`sche Signale, Boole`sche Gatter und einfache FlipFlops



Transistorebene

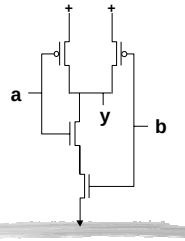
- Realisierung Boolescher Elemente durch Transistoren

Inverter



e	y
0	1
1	0

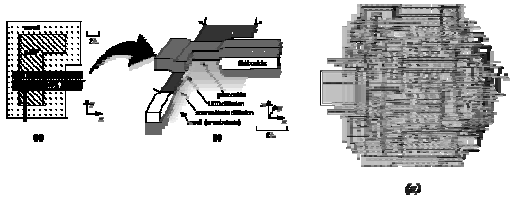
NAND



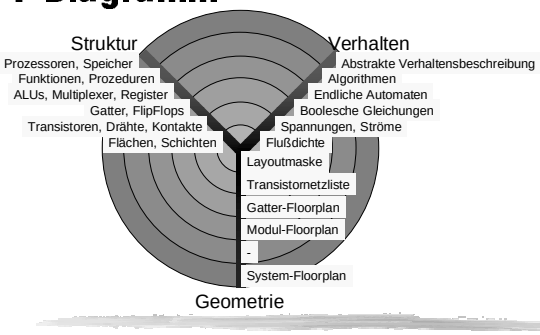
a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

Layoutebene

- Realisierung von Transistoren durch dotierte Bereiche und isolierenden Schichten auf dem IC



Y-Diagramm

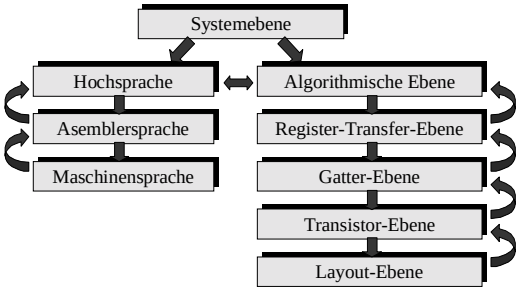


Systementwurf

Ziel des Entwurfsprozesses

- Beschreibungen von einer hohen Abstraktionsebene in eine niedere Abstraktionsebene zu transformieren
  - durch Verfeinerung
  - unter Beibehaltung der Funktionalität (eventuell auch des Zeitverhaltens)
- Verifikation/Validation der Beschreibung auf den unterschiedlichen Ebenen

Systementwurf in der Realität



Automatisierung

- Text-/Grafikeditoren
- Simulation
  - Testbenchgeneratoren
  - Coverage-Analysatoren
- Verifikation
  - Äquivalenzprüfung
  - Modellprüfung
  - linting (design rule checking)
- Timing Analyse
- Synthese
  - Verhaltenssynthese
  - Logiksynthese
- Plazierung und Verdrahtung
- Automatic testpattern generator (ATPG)

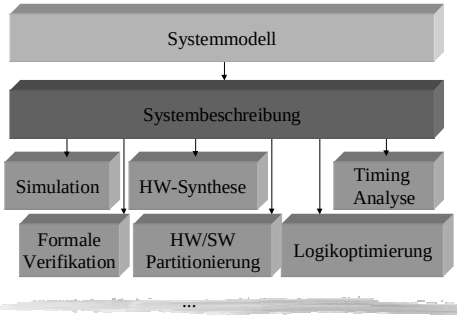
1 Hardwareentwurf

- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung

Literatur

- James M. Lee, Verilog Quickstart! Kluwer Academic Publishers, ISBN 0-7923-8515-2

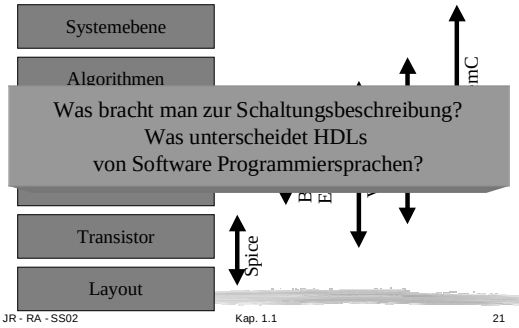
Zusammenhang



Hardware-Beschreibungssprachen

- präzise Spezifikation
- Möglichkeit der Simulation
- Automatisierung
- Dokumentation
- Beschleunigung
- ...

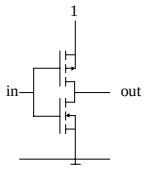
Einordnung verschiedener HDLs



Schaltungsbeschreibungen

Schaltungen bestehen aus

- Komponenten
- und Verbindungen dieser Komponenten

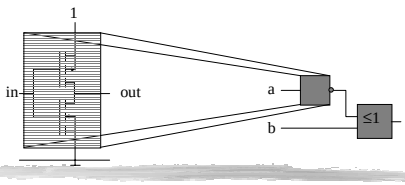


Zu komplex für große Schaltungen

Hierarchiebildung

Zusammenfassung von Teilen zu neuen Modulen

- Gatter, FlipFlops
- ALU, Register, Speicher,
- Prozessor



Algorithmische Beschreibung

Strukturelle Beschreibung ist oft zu komplex für große Entwürfe (mit 20 Millionen Gattern)
⇒ algorithmische Beschreibungen notwendig

Das Verhalten der Module wird durch eine (imperative) Programmiersprache definiert. Diese ist Teil der Hardwarebeschreibungs-sprache

Algorithmische Beschreibung II

Besonderheiten von Hardware:

- Funktionen verbrauchen Zeit
  - ➜ Zeitbegriff
- Funktionen können parallel arbeiten
  - ➜ parallele Tasks
- Kommunikation zwischen Modulen
  - ➜ Signale und Ereignisse (events)
- zweiwertige Logik nicht ausreichend
  - ➜ mehrwertige Logik (0,1,x,z,...)

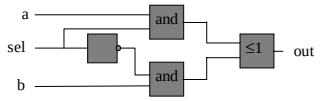
Verilog

- entwickelt von Philip Moorby 1983/1984 bei Gateway Design Automation
- wurde anfangs gemeinsam mit dem Simulator entwickelt
- 1987 Verilog-basiertes Synthesewerkzeug von Synopsys
- 1989 Gateway wurde von Cadence aufgekauft
- Verilog wird public domain um mit VHDL zu konkurrieren

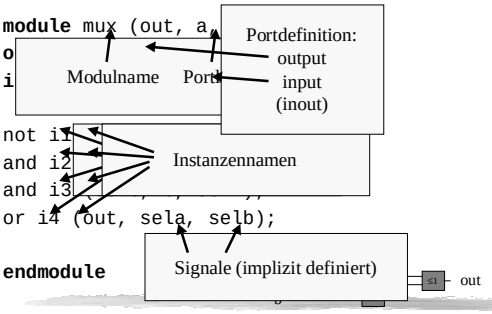
Verilog

- Standard
  - einheitliche Schnittstelle zwischen Werkzeugen und Firmen
- Portabilität
  - Entwürfe können durch verschiedene Synthesewerkzeuge optimiert und durch verschiedene Analysewerkzeuge simuliert werden
  - Technologie- und Firmenunabhängigkeit
    - Wechsel des Technologiepartners möglich

Strukturelle Beschreibung: Multiplexer

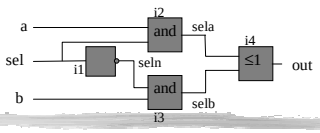


Strukturelle Beschreibung: Multiplexer



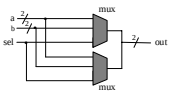
Strukturelle Beschreibung: Multiplexer

```
module mux (out, a, b, sel);
output out;
input a, b, sel;

not i1 (seln, sel);
and i2 (sela, a, sel);
and i3 (selb, b, seln);
or i4 (out, sela, selb);

endmodule
```



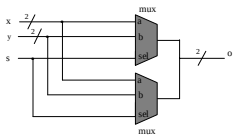
Hierarchie: Multiplexer 2

```
module mux2 (out, a, b, sel);
output [1:0] out;
input [1:0] a, b;
input sel;

mux hi (out[1], a[1], b[1], sel);
mux lo (out[0], a[0], b[0], sel);

endmodule
```



Modulverbindung durch Portnamen

```
module mux2 (o, x, y, s);
...
mux hi (  .out(o[1]),
          .a(x[1]),
          .b(y[1]),
          .sel(s));
mux lo (  .sel(s),
          .b(y[0]),
          .a(x[0]),
          .out(o[0]));
```



Grundlegende Sprachelemente I

Kommentare

// Zeilenkommentar bis zum Zeilenende

/* Bereichskommentar kann über
mehrere Zeilen bis zum schließenden
Kommentarzeichen gehen */

Grundlegende Sprachelemente II

Bezeichner

- Buchstaben (a-z,A-Z), Zahlen (0-9) oder _ $
- beginnt mit Buchstabe oder _
- case sensitive
- maximal 1024 Zeichen lang
- Escaped identifier
  `\hier/kann?jedes:Zeichen.kommen`

Grundlegende Sprachelemente IV

Macros

- Definition
  `` `define opcode_add 33 ``
- Anwendung
  b = `opcode_add

Logikwerte

- 0 : logisch falsch, niedriger Signalpegel
- 1 : logisch wahr, hoher Signalpegel
- x : unbekannt (don't care)
- z : hochohmig (keine Verbindung)

Zahlen

Bits, Bitvektoren

Bitbreite	‘	Basis	Werte

```
8’b11001001
8’hff
16’d12
12’o777
```

Integers, Reals
- •Bitbreite ist maschinenabhängig (z.B. 32 Bit)
- •vorzeichenbehaftete Arithmetik

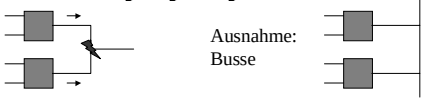
Netze (von Bitvektoren)

- Verbinden Module
- es gibt mehrere Netztypen:
  - wire (tri)
  - wand (triand)
  - wor (trior)
  - tri1, tri0
  - supply1, supply0
- Es können Bitvektoren gebildet werden, z.B.:

```
wire [63:32] high;
```


Resolution

In HW haben Logiksignale genau einen “Treiber”



Verhalten hängt vom Netztyp ab:
- wire: nur ein Treiber erlaubt
- wand: Konjunktion der Treibersignale
- wor: Disjunktion der Treibersignale
- tri0: wie wand mit pulldown (kein Treiber ⇒ Leitung=0)
- tri1:wie wand mit pullup (kein Treiber ⇒ Leitung=1)
- supply0, supply1: kein Treiber erlaubt

Grundlegende Sprachelemente VI

Operatoren:
- arithmetische: +, -, *, /, %
- logische: &, &&, |, ||, ^, ~, !, <<, >>, <<<, >>>
- Reduktion: &, |, ^, ~&, ~|, ~^
- relationale: < , <= , > , >= , == , === , != , !==
- bedingter Operator:
  `cond ? true_exp : false_exp`
- concatenation: { ... }
  ```
  x = { a, b, c };
  {x,y} = 8’b10011101;
  ```


Beschreibung von Schaltnetzen

- Mit “built-in primitives” (siehe MUX)
- Mit “continous assignment”

```
module sn(out, in1, in2);
output out;
input in1, in2;

assign out = in1 & in2;

endmodule
```


Beschreibung von Schaltnetzen

- Mit “built-in primitives” (siehe MUX)
- Mit “continous assignment”

```
module sn(out, in1, in2);
output [32:0] out;
input [31:0] in1, in2;

assign out = in1 + in2;
// hunderte von Gattern
endmodule
```

Beschreibung von Schaltnetzen

- Mit "built-in primitives" (siehe MUX)
- Mit "continous assignment"

```
module sn(out, in1, in2);
output [63:0] out;
input [31:0] in1, in2;

assign out = in1 * in2;
// tausende von Gattern
endmodule
```


Continous Assignment Beispiel

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;

assign out = sel ? in2 : in1;

endmodule;
```


Verhaltensbeschreibung

initial

jeder initial-Block jedes Moduls wird zu Beginn der Simulation genau einmal bis zum Ende ausgeführt.

always

jeder always-Block jedes Moduls wird zu Beginn der Simulation ausgeführt und wird dann zyklisch immer wiederholt.

Alle Blöcke arbeiten parallel

Beispiele

```
module test;

initial $display("init block 1");
initial $display("init block 2");

endmodule
```

Systemtask für Bildschirmausgabe

Was erscheint auf dem Bildschirm?

Beispiele

```
module test;

initial $display("init block 1");
initial $display("init block 2");
always $display("always block");

endmodule
```

Was erscheint auf dem Bildschirm?

Der Zeitoperator

Alle bisher gezeigten Operationen sind (Simulations-) zeitfrei

Mit dem Operator # kann Zeit verbraucht werden

```
module test;
initial #2 $display("init block 1");
initial #5 $display("init block 2");
endmodule
```

Sequentielle Schachtelung

Mehrere Anweisungen können mit begin-end zusammengefaßt werden. Alle eingeschlossenen Anweisungen werden sequentiell abgearbeitet.

```
module test;
initial
  begin
     $display("init block 1");
     $display("init block 2");
  end
endmodule
```


Sequentielle Schachtelung II

Zeit vergeht "sequentiell", d.h. relativ zum letzten Zeitpunkt

```
module test;
initial
  begin
     #2 $display("init block 1");
     #2 $display("init block 2");
  end
endmodule
```


Parallele Schachtelung

Mehrere Anweisungen können mit fork-join zusammengefaßt werden. Alle einge-schlossenen Anweisungen werden parallel abgearbeitet.

```
module test;
initial
  fork
     $display("init block 1");
     $display("init block 2");
  join
endmodule
```


Parallele Schachtelung II

Da alle Anweisungen parallel abgearbeitet werden, wird Zeit immer absolut gemessen

```
module test;
initial
  fork
     #2 $display("i
     #4 $display("i
  join
endmodule
```

Begin-end Blöcke und fork-join Blöcke können beliebig geschachtelt werden

Register

```
reg val [7:0];
```

- Wird in algorithmischen Beschreibungen verwendet
- nur interne Signale und outputs können Register sein
- können auch zur Schaltnetzmodellierung verwendet werden
- Memories:
  `reg [7:0] mem [0:1023];`

Ereignisse

- sind Signalwechsel

- warten auf Ereignisse: `@c`
- abfangen mehrere Ereignisse: `@(e1 or e2)`

Ereignisse Beispiel: DFlipFlop

```
module dff (out, clock, in);
output out;
input clock, in;
reg out;

always @(posedge clock)
  out = in;

endmodule
```


Ereignisse Beispiel: Multiplexer

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;
reg out;

always @(in1 or in2 or sel)
  if (sel) out = in2
  else     out = in1

endmodule
```


Ereignisse Beispiel: FehlerMux

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;
reg out;

always @sel
  if (sel) out = in2
  else     out = in1

endmodule
```


Levelabhängiges Warten

`wait (boolean-expression);`

Falls *boolean-expression* wahr ist, wird direkt mit der nachfolgenden Anweisung im Programmfluß fortgefahren

Falls *boolean-expression* falsch ist, dann wird der Programmfluß solange unterbrochen bis der Ausdruck wahr wird

Kontrollfluß

Bedingung

- **`if`** (cond) statement
- **`if`** (cond) statement1
  `else` statement2
- **`case`** (sel)
  0 : y = a
  1 : y = b
  `default` : y = 2'bxx
  `endcase`
- **`casez`**, **`casex`**

Kontrollfluß II

Schleifen

- **`forever`** statement

```
module ClockGen (clk);
output clk;
reg clk;

initial begin
          clk = 0;
          forever #50 clk = ~clk;
        end
endmodule
```

Kontrollfluß II

Schleifen

- **forever** statement
- **repeat** (num) statement

```
module ClockGen (clk);
initial repeat (5) $display("hallo");
endmodule
```


Kontrollfluß II

Schleifen

- **forever** statement
- **repeat** (num) statement
- **while** (cond) statement
- **for** (init; cond; incr) statement

Zuweisungen in algorithmischen Blöcken

var = expression;

Beispiel

```
initial begin
    x = 3;
    y = 4;
    fork
        x = y;
        y = x;
    join
end
```


Zuweisungen mit intra-assign delay

var = #num expression;

Beispiel

```
initial begin
    x = 3;
    y = 4;
    fork
        x = #1 y;
        y = #1 x;
    join
end
```


Nichtblockierende Zuweisungen

var <= #num expression;

Beispiel

```
initial begin
    x = 3;
    y = 4;
    begin
        x <= #1 y;
        y <= #1 x;
    end
end
```


Zuweisungen Zusammenfassung

Zuweisungstyp	LHS	Wann ausgeführt	Wo im Modul
Procedural assignment	reg	Bei Aufruf	In alg. Blöcken
Continous assignment	net	Bei RHS-Änderung	Im umgebenden Modul

Tasks

- dienen zum „verkapseln" von Verhalten
- werden innerhalb von Modulen definiert
- können auf umgebende Daten zugreifen
- können inputs und outputs haben
- können Verzögerungszeiten beinhalten
- Daten innerhalb eines Tasks gibt es nur einmal auch wenn mehrere identische Tasks laufen

Beispieltask

```
module counter(out,
  clk, reset);
output [7:0] out;
reg [7:0] out;
input clk, reset;

always @clk
  if (reset) out = 0;
  else incr(out);
```

```
task incr;
inout [7:0] x;
  x = x + 1;
endtask
endmodule
```


Funktionen

- dienen zum „verkapseln" von Verhalten
- werden innerhalb von Modulen definiert
- kann auf umgebende Daten zugreifen
- können inputs haben
- liefern immer ein Ergebnis zurück
- dürfen keine Verzögerungszeiten beinhalten
- Daten innerhalb einer Funktion gibt es nur einmal, d.h. es gibt keinen Laufzeitstack

Beispielfunktion

```
module mux (out, a, b,
  c, d, sel);
output out;
input [1:0] sel;
input [7:0] a, b, c,
  d;

assign out =
  muxfunct(sel,a,b,c,d
  );
```

```
function [7:0] muxfunct;
input [1:0] sel;
input [7:0] a,b,c,d;
  case (sel)
    2'b00 : muxfunct = a;
    2'b01 : muxfunct = b;
    2'b10 : muxfunct = c;
    2'b11 : muxfunct = d;
  endcase
endfunction
endmodule
```


Parametrisierte Module

- Dienen zur Beschreibung generischer Module
  - variable Bitbreite
  - variable Verzögerungszeiten
- Parameter müssen vor der Simulationszeit festgelegt werden
- Parameter werden mit dem Schlüsselwort `parameter` definiert
- Parameter sind defaultmäßig vom Typ integer

Beispiel: N-Bit Addierer

```
module nadder(cout, sum, a, b, cin);
parameter size = 32;
parameter delay = 1;
output [size-1:0] sum;
output cout;
input [size-1:0] a, b;
input cin;

assign #delay {cout,sum} = a + b + cin;

endmodule
```

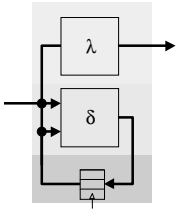
Instanziierung

- Durch den Parameternamen
  ```
  nadder a1 (z,a5,b5,c5,x);
  defparam a1.size = 16;
  defparam a1.delay = 4;
  ```
- Durch die Parameterreihenfolge
  ```
  nadder #(16,4) a1 (z,a5,b5,c5,x);
  ```
- Durch Parameternamensliste
  ```
  nadder #(.size(16), .delay(4))
     a1 (z,a5,b5,c5,x);
  ```


Systemtasks

- Simulationskontrolle
  - $finish, $stop
- Bildschirmausgabe
  - $display, $write, $strobe, $monitor
- Filezugriff
  - Filehandels sind integer (maximal 32)
  - $fopen("file1"), $fclose(i)
  - $fdisplay, $fwrite, $fstrobe, $fmonitor

Endliche Automaten in Verilog



Ausgangsfunktion

Übergangsfunktion

Register (Zustandsspeicher)

Kombinationsmöglichkeiten

Zustandsregister	Übergangsfunktion	Ausgangsfunktion	Typ
separat	separat	separat	1
kombiniert	kombiniert	separat	2
separat	kombiniert	kombiniert	3
kombiniert	separat	kombiniert	4
kombiniert	kombiniert	kombiniert	5

FSMs in Verilog: Typ1

- Jede Einheit in einen separaten always-Block:

```
always @(posedge clk)  state = next_state

always @(state or inp1 or ... or inpn) begin
  next_state = state;
  case (state)
          ...
  endcase

always @(state or inp1 or ... or inpn) begin
   ...
end
```


FSMs in Verilog: Typ1

- Vorteile:
  - übersichtlich
  - einfache Wartbarkeit
  - MOORE and MEALY-Maschinen
- Nachteil
  - längste Darstellung
  - Übergangsfunktion wird bei jeder Eingabe aktualisiert

FSMs in Verilog: Typ2

Register + Übergangsfunktion

- Vorteile
  - kompakter als Typ1
  - effizienter, da Übergangsfunktion nur bei clk aktualisiert wird
  - MOORE and MEALY-Maschinen
- Nachteile
  - nicht so modular wie Typ1

FSMs in Verilog: Typ3

Ausgangsfunktion + Übergangsfunktion

- Vorteile
  - MOORE and MEALY-Maschinen
  - kompakter als Typ1
- Nachteile
  - sehr unübersichtlich
  - Übergangsfunktion wird bei jeder Eingabe aktualisiert-

FSMs in Verilog: Typ4

Register + Ausgangsfunktion

- Vorteile
  - kompakter als Typ1
- Nachteile
  - nur MOORE-Maschinen
  - Übergangsfunktion wird bei jeder Eingabe aktualisiert

FSMs in Verilog: Typ5

Register + Über- und Ausgangsfunktion

- Vorteile
  - kompakter als Typ1 (keine next_state-Variable nötig)
  - Übergangs- und Ausgangsfunktion nur bei clk aktualisiert
- Nachteile
  - nur MOORE-Maschinen

Zustandskodierung

- Boole'sche Kodierung

```
`define IDLE    3'b00
`define CHOOSE  3'b01
`define WATER   3'b10
`define ORANGE  3'b11
```

- Gray Kodierung

```
`define IDLE    3'b00
`define CHOOSE  3'b01
`define WATER   3'b11
`define ORANGE  3'b10
```


Zustandskodierung

- One-Hot

```
`define IDLE    4'b0001
`define CHOOSE  4'b0010
`define WATER   4'b0100
`define ORANGE  4'b1000
```

- Bei MOORE-Maschinen: Ausgänge als Zustandskodierung

```
`define IDLE    4'b000  // (d=0, w=0, o=0)
`define CHOOSE  4'b100  // (d=1, w=0, o=0)
`define WATER   4'b110  // (d=1, w=1, o=0)
`define ORANGE  4'b101  // (d=1, w=0, o=1)
```

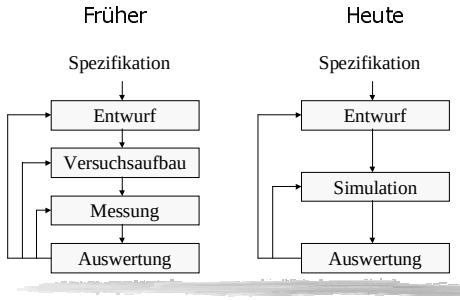
1 Hardwareentwurf

- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
  - Simulation
  - Formale Verifikation
  - Timinganalyse
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung

Literatur

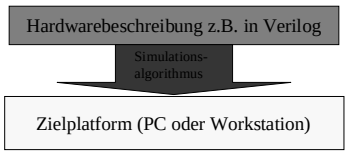
- Thomas Kropf, Introduction to Formal Hardware Verification, Springer Heidelberg, ISBN 3-540-65445-3
- Franz Rammig, Systematischer Entwurf digitaler Systeme, Teubner Stuttgart, ISBN 3-519-02265-6
- Drechsler und Becker, Graphenbasierte Funktionsdarstellung, B. G. Teubner Stuttgart, ISBN 3-519-02149-8vvorlesung
- Shi-Yu Huang, Kwang-Ting Cheng, Formal Equivalence Checking and Design Debugging Kluwer Academic Publishers, ISBN 0-7923-8184-X

Motivation



Simulationstechniken

Aufgabe eines Simulationsalgorithmus:
Abbilden der Hardwarefunktionalität auf eine Zielarchitektur, z.B. von Neumann Rechner



Problem:
hoher Grad an Parallelität muß auf eine sequentielle Maschine „projeziert" werden

Arten der HW-Simulation

- Analogsimulation (Spice, ...)
  - werte- und zeitkontinuierlich
  - nichtlineare Differentialgleichungen
- Digitalsimulation (VSS, modelsim, ...)
  - werte- und zeitdiskret
  - boolesche Algebra
- Mixed-Mode/Simulatorkopplung (Saber,...)
  - analog/digital-Simulation

Logiksimulationsalgorithmen

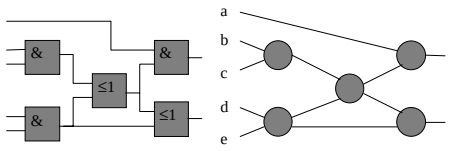
- Streamline Code Simulation
  - Schaltnetze, vollsynchrone Schaltwerke
- Equitemporal Iteration
  - Analogsimulation
- Critical Event Scheduling
  - Schaltungen auf unterschiedlichen Abstraktionsebenen und mit Zeitinformation

Streamline Code Simulation

- auch „Compiled Mode"
- es wird direkt ausführbarer Code auf der Zielplattform generiert
- Einschränkungen
  - kombinatorische oder strikt synchrone Schaltungen
  - keine Zeitinformationen

Streamline Code Simulation II

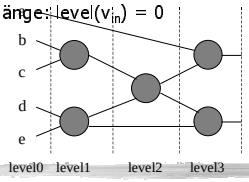
Schaltung wird als gerichteter azyklischer Graph notiert:



Streamline Code Simulation III

Knoten des azyklischen Graphes werden halbgeordnet (levelizing)

- $level(v_i) = 1 + \max \{ level(v_k) \mid$ es gibt eine Kante von v_k nach $v_i \}$
- Primäreingänge: $Level(v_{in}) = 0$



Streamline Code Simulation IV

Eigenschaften des Graphen:

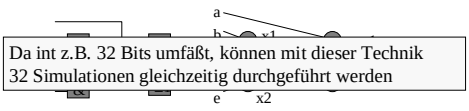
- Knoten auf einer höheren Ebene können Knoten auf niedrigeren Ebenen nicht beeinflussen
- Knoten auf der selben Ebene beeinflussen sich gegenseitig nicht
- Knoten auf niedrigeren Ebenen beeinflussen Knoten auf höheren Ebenen

Streamline Code Simulation V

Codegenerierung:

- Die Level werden nacheinander implementiert
- Die Reihenfolge der Operationen in den Levels ist beliebig
- Die Gatter werden durch Operatoren der Zielmaschine realisiert
- Die Verbindungen (Netze) werden durch Variablen der Zielmaschine implementiert

Streamline Code Simulation VI



Da int z.B. 32 Bits umfäßt, können mit dieser Technik 32 Simulationen gleichzeitig durchgeführt werden

```
int a, b, c, d, e;
int x1, x2, x3;
int o1, o2;

x1 = b & c;    // level 1
x2 = d & e;    // level 1
x3 = x1 | x2;  // level 2
o1 = a & x3;   // level 3
o2 = x2 & x3;  // level 3
```

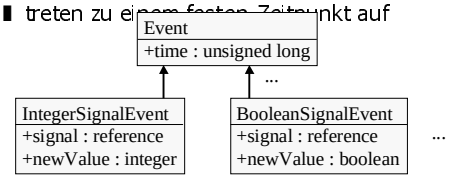
Critical Event Scheduling

- IDEE:
  - Verwaltung algorithmischer Blöcke durch Threads
  - nur die Systemteile werden neu berechnet, deren Eingaben sich verändert haben (anstehende Ereignisse=Events)
  - globale Datenstruktur: EventQueue
  - Ein spezieller Thread übernimmt die Verwaltung: Scheduler

Critical Event Scheduling III

Ereignisse

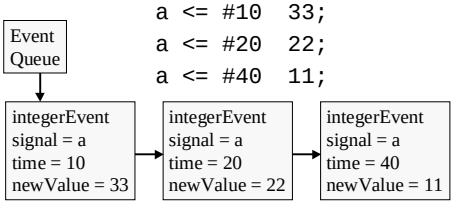
- sind Änderung eines Signalwertes oder werden von Zeitoperatoren (#) generiert
- treten zu einem festen Zeitpunkt auf



Critical Event Scheduling IV

Simulationsalgorithmus speichert eine sortierte Liste der anstehenden Ereignisse

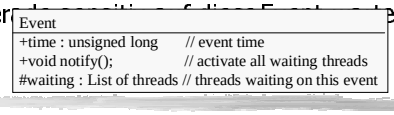
```
a <= #10  33;
a <= #20  22;
a <= #40  11;
```



Critical Event Scheduling V

Um bei Eintreten von Ereignissen die Simulation weitertreibenzu können müssen die Threads gespeichert werden, die gerade auf Events warten

⇒ Zu jedem Event werden die Threads (Blöcke, Tasks etc.) gespeichert, die ger[illegible]en



Critical Event Scheduling VI

```
while (currentTime <= finalTime &&
  !queue.empty()){
  currentEvent = queue.top();
  currentTime = currentEvent.time;
  notify all threads t waiting on
  currentEvent {
      execute t until next wait/#/@ statement
          // hier werden eventuell neue
  Ereignisse
          // in die Queue eingeführt
          // beim Erreichen von  wait trägt
  sich der
          // thread in die „Warteliste“ des
```


Critical Event Scheduling VII

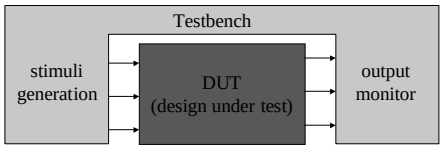
Durch diese ereignisbasierte Simulationstechnik

ergeben sich zwei Zeitachsen:

- Simulationszeit
- Deltazeit (delta delay)

Deltadelays entstehen durch Abarbeitung mehrerer Ereignisse, die zur gleichen Simulationszeit auftreten

Verifikation durch Simulation



- Eingangsstimuli erzeugen
- Ausgangssignale beobachten/verifizieren

Formale Verifikation

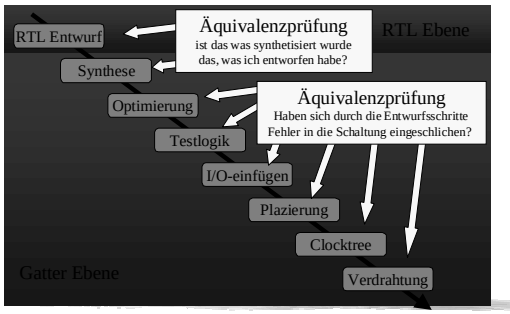
- Äquivalenzprüfung
  - sind zwei Schaltungen äquivalent
- Modellprüfung
  - erfüllt eine Schaltung bestimmte (sequentielle) Eigenschaften
- Theorembeweisen
  - interaktive Methode, basierend auf Logik höherer Ordnung

Was ist Äquivalenzprüfung

Engl. „*equivalence checking*" (EC)

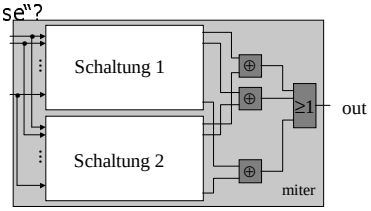
- gegeben: zwei digitale Schaltungen
- gefragt: haben beide die gleiche Funktionalität (keine zeitliches Verhalten)
  - bei kombinatorischen Schaltungen: sind die Ausgänge bei gleichen Eingangsbelegungen gleich?
  - Bei sequentiellen Schaltungen: sind die Ausgänge zu allen Zeitpunkten bei gleichen Eingabefolgen identisch?

Designflow mit Äquivalenzprüfung



Äquivalenzprüfung

- Erzeuge „miter"
- Ist der Ausgang für alle Eingangsbelegungen „false"?



Äquivalenzprüfung von Schaltnetzen

Vorgehen

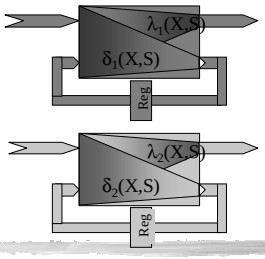
- Transformation der Schaltnetze in eine Normalformdarstellung (z.B. KNF, ROBDDs,...)
- Liegt Äquivalenz der Normalformen vor?

Optimierungen

- Ausnutzen von strukturellen Ähnlichkeiten
- inkrementelle Verfahren

Äquivalenzprüfung für Schaltwerke

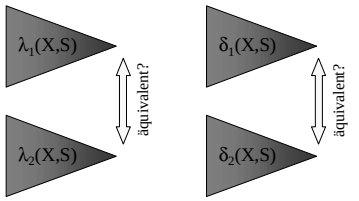
- Schaltungen in Huffman-Normalform



Äquivalenzprüfung für Schaltwerke II

Gleiche Zustandskodierung und eindeutiger Resetzustand: Reduktion auf Äquivalenzprüfung von Schaltnetzen

Für jeden Ausgang und für jedes FlipFlop:



Äquivalenzprüfung für Schaltwerke IV

Was ist, wenn man keine gleiche Zustandskodierung hat?

Oder keine gemeinsame Rücksetzleitung?

⇒ Einsatz von Techniken und Methoden der Automatentheorie

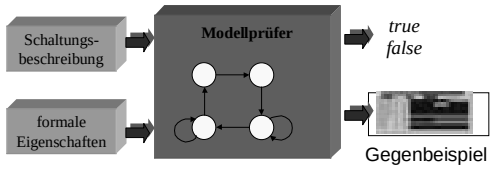
- Produktautomaten bilden mit „miter"-Ausgang
- Durchsuchen des Zusandsraumes nach Zuständen, die „true" am Ausgang erzeugen

Modellprüfung

Erfüllt ein Schaltwerk eine gegebene Spezifikation?

- Modellierung des Schaltwerkes durch endliche Automaten
- Spezifikation der Eigenschaften in temporalen (modalen) Aussagenlogiken

Modellprüfung



Temporale Aussagenlogik (LTL)

- atomare Ausdrücke
  - Signale des Designs
  - Boole`sche Funktionen, z.B. (a<b), OneHot(a,b,c)
- Boole`sche Operatoren
  - z.B. $\neg\varphi$, $\varphi\wedge\psi$, $\varphi\vee\psi$, ...
- temporale Operatoren $m \in N$, $n \in N \cup \{\infty\}$

  $X_{[m]}\ \varphi$ in genau m Zeitschritten gilt φ

  $F_{[m,n]}\ \varphi$ im Intervall [m,n] gilt φ mindestens einmal

  $G_{[m,n]}\ \varphi$ im Intervall [m,n] gilt immer φ

  $\varphi\ U_{[m,n]}\ \psi$ ψ wird im Intervall [m,n] wahr und bis dahin gilt immer φ

Temporale Aussagenlogik (LTL)

Typische Eigenschaften

- Sicherheit:
  a und b werden nie gleichzeitig wahr
  $G\,\neg(a \wedge b)$
- Lebendigkeit
  jede Anforderung wird innerhalb von 5 bis 7 bestätigt
  $G\,(req \rightarrow F_{[5,7]} ack)$
- Fairness:
  Innerhalb von 30 Schritten gilt mindestens einmal a

Semantik von LTL

Definiert über unendlichen Signalfolgen

$Sig = \{ a, b, c, ... \}$ Menge von Bezeichnern für Signale

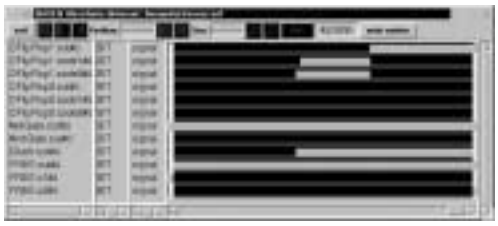
Signalfolge π weist jedem Signal $a \in Sig$ zu jedem Zeitpunkt $t \in N$ einen Boole'schen Wert zu

$\pi: Sig \times N \rightarrow B$

Der Suffix einer Signalfolge ab der Zeit t ist definiert durch

$\pi^t(a,z) = \pi(a,z+t)$

Signalfolgen



Semantik von LTL

ist definiert durch die Modellrelation $\models$

$\pi \models a \quad \Leftrightarrow \quad \pi(a,0)=true$

$\pi \models \neg\varphi \quad \Leftrightarrow \quad \pi \not\models \varphi$

$\pi \models \varphi \wedge \psi \quad \Leftrightarrow \quad \pi \models \varphi$ und $\pi \models \psi$

$\pi \models X_{[m]}\,\varphi \quad \Leftrightarrow \quad \pi^m \models \varphi$

$\pi \models F_{[m,n]}\,\varphi \Leftrightarrow \exists\, m \le t \le n\,.\, \pi^t \models \varphi$

$\pi \models G_{[m,n]}\,\varphi \Leftrightarrow \quad \forall\, m \le t \le n\,.\, \pi^t \models \varphi$

$\pi \models \varphi\, U_{[m,n]}\, \psi \Leftrightarrow \quad \exists\, m \le t \le n\,.\, \pi^t \models \psi$ und $\forall\, k < t\,.\, \pi^k \models \varphi$

Signalabfolgen und Schaltung

- Die Schaltungsbeschreibung enthält freie Eingäng
  - → beliebig viele Signalabfolgen können aus der Schaltung entstehen
  - → alle Signalabfolgen müssen die LTL-Formel erfüllen
  - → Beweis nicht über den Einzelnen Signalabfolgen, sondern direkt auf dem formalen Modell (endlichen Automaten)

Theorembeweisen

- Modellierung der Schaltung in Logik Höherer Ordnung
- Modellierung der Eigenschaften in Logik höherer Ordnung
- interaktiver Beweis durch Anwendung von Theoremen
- oft kombiniert mit
  - Heuristiken zur Automation des Beweises
  - automatischen Entscheidungsprozeduren

Timinganalyse

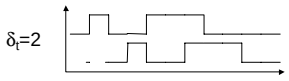
- Wie schnell Propagieren sich Signalwechsel zwischen den Eingängen, den Ausgängen und den Registern
  - → Bestimmung der Taktrate

Zeitmodelle für Gatter

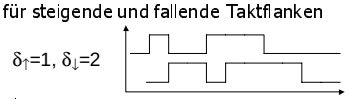
- Totzeit
  - das Eingangssignal erscheint zeitverschoben am Ausgang
- Totzeiten für steigende und fallende Taktflanken
  - steigende und fallende Taktflanken werden unterschiedlich verzögert
- träge Totzeit
  - kurze Impulse werden nicht an den Ausgang propagiert

Zeitmodelle für Gatter

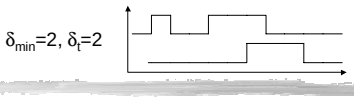
- Totzeit



- Totzeiten für steigende und fallende Taktflanken

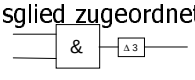


- träge Totzeit

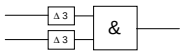


Einfaches Totzeitmodell

- Jedes Gatter arbeitet ohne Verzögerung
- Jedem Gatter ist ein externes Verzögerungsglied zugeordnet



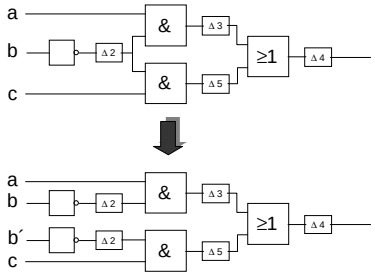
äquivalent zu



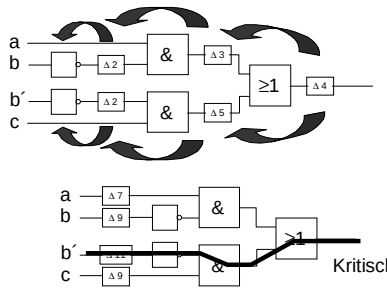
Timinganalyse

- duplizieren von mehrfach benutzen Schaltungsteilen und Eingängen
- verschieben der Verzögerungszeiten zu den Eingängen der Gatter
- addieren von Verzögerungszeiten auf einer Leitung

Duplizieren von Schaltungsteilen



Addieren der Verzögerungszeiten



Sensibilisierbarkeit

Ist der längste Pfad auch sensibilisierbar?

D.h. gibt es ein Eingangssignalwechsel, der sich über den längsten Pfad zum Ausgang durchsetzt?

➔D-Kalkül

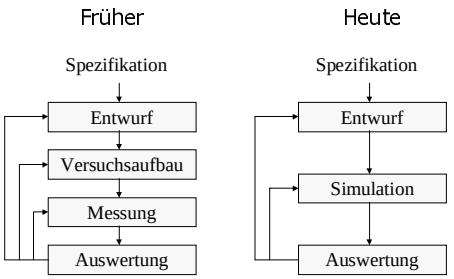
1 Hardwareentwurf

- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
  - Simulation
  - Formale Verifikation
  - Timinganalyse
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung

Literatur

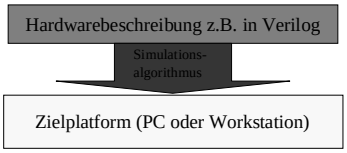
- Thomas Kropf, Introduction to Formal Hardware Verification, Springer Heidelberg, ISBN 3-540-65445-3
- Franz Rammig, Systematischer Entwurf digitaler Systeme, Teubner Stuttgart, ISBN 3-519-02265-6
- Drechsler und Becker, Graphenbasierte Funktionsdarstellung, B. G. Teubner Stuttgart, ISBN 3-519-02149-8vvorlesung
- Shi-Yu Huang, Kwang-Ting Cheng, Formal Equivalence Checking and Design Debugging Kluwer Academic Publishers, ISBN 0-7923-8184-X

Motivation



Simulationstechniken

Aufgabe eines Simulationsalgorithmus:
Abbilden der Hardwarefunktionalität auf eine Zielarchitektur, z.B. von Neumann Rechner



Problem:
hoher Grad an Parallelität muß auf eine sequentielle Maschine „projiziert" werden

Arten der HW-Simulation

- Analogsimulation (Spice, ...)
  - werte- und zeitkontinuierlich
  - nichtlineare Differentialgleichungen
- Digitalsimulation (VSS, modelsim, ...)
  - werte- und zeitdiskret
  - boolesche Algebra
- Mixed-Mode/Simulatorkopplung (Saber,...)
  - analog/digital-Simulation

Logiksimulationsalgorithmen

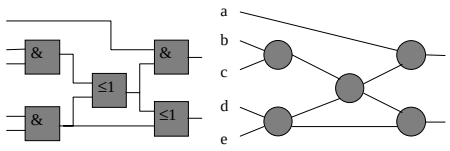
- Streamline Code Simulation
  - Schaltnetze, vollsynchrone Schaltwerke
- Equitemporal Iteration
  - Analogsimulation
- Critical Event Scheduling
  - Schaltungen auf unterschiedlichen Abstraktionsebenen und mit Zeitinformation

Streamline Code Simulation

- auch „Compiled Mode"
- es wird direkt ausführbarer Code auf der Zielplattform generiert
- Einschränkungen
  - kombinatorische oder strikt synchrone Schaltungen
  - keine Zeitinformationen

Streamline Code Simulation II

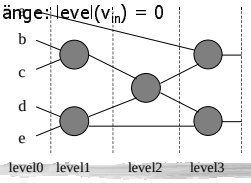
Schaltung wird als gerichteter azyklischer Graph notiert:



Streamline Code Simulation III

Knoten des azyklischen Graphes werden halbgeordnet (levelizing)

- $level(v_i) = 1 + \max \{ level(v_k) \mid$ es gibt eine Kante von v_k nach $v_i \}$
- Primäreingänge: $Level(v_{in}) = 0$



Streamline Code Simulation IV

Eigenschaften des Graphen:

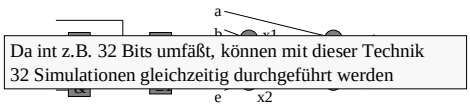
- Knoten auf einer höheren Ebene können Knoten auf niedrigeren Ebenen nicht beeinflussen
- Knoten auf der selben Ebene beeinflussen sich gegenseitig nicht
- Knoten auf niedrigeren Ebenen beeinflussen Knoten auf höheren Ebenen

Streamline Code Simulation V

Codegenerierung:

- Die Level werden nacheinander implementiert
- Die Reihenfolge der Operationen in den Levels ist beliebig
- Die Gatter werden durch Operatoren der Zielmaschine realisiert
- Die Verbindungen (Netze) werden durch Variablen der Zielmaschine implementiert

Streamline Code Simulation VI



Da int z.B. 32 Bits umfäßt, können mit dieser Technik 32 Simulationen gleichzeitig durchgeführt werden

```
int a, b, c, d, e;
int x1, x2, x3;
int o1, o2;

x1 = b & c;    // level 1
x2 = d & e;    // level 1
x3 = x1 | x2;  // level 2
o1 = a & x3;   // level 3
o2 = x2 & x3;  // level 3
```

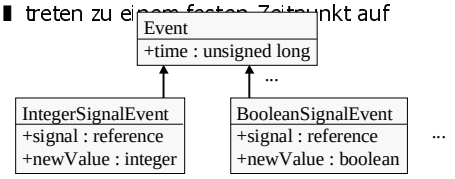

Critical Event Scheduling

- IDEE:
  - Verwaltung algorithmischer Blöcke durch Threads
  - nur die Systemteile werden neu berechnet, deren Eingaben sich verändert haben (anstehende Ereignisse=Events)
  - globale Datenstruktur: EventQueue
  - Ein spezieller Thread übernimmt die Verwaltung: Scheduler

Critical Event Scheduling III

Ereignisse

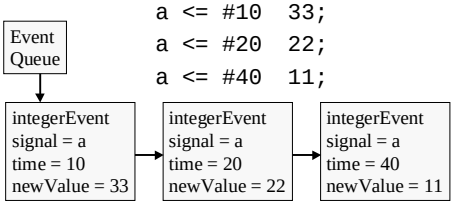
- sind Änderung eines Signalwertes oder werden von Zeitoperatoren (#) generiert
- treten zu einem festen Zeitpunkt auf



Critical Event Scheduling IV

Simulationsalgorithmus speichert eine sortierte Liste der anstehenden Ereignisse

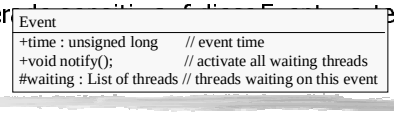
```
a <= #10  33;
a <= #20  22;
a <= #40  11;
```



Critical Event Scheduling V

Um bei Eintreten von Ereignissen die Simulation weitertreibenzu können müssen die Threads gespeichert werden, die gerade auf Events warten

⇒ Zu jedem Event werden die Threads (Blöcke, Tasks etc.) gespeichert, die ger[illegible]



Critical Event Scheduling VI

```
while (currentTime <= finalTime &&
  !queue.empty()){
  currentEvent = queue.top();
  currentTime = currentEvent.time;
  notify all threads t waiting on
  currentEvent {
      execute t until next wait/#/@ statement
          // hier werden eventuell neue
  Ereignisse
          // in die Queue eingeführt
          // beim Erreichen von  wait trägt
  sich der
          // thread in die „Warteliste“ des
```


Critical Event Scheduling VII

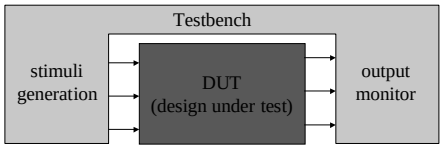
Durch diese ereignisbasierte Simulationstechnik

ergeben sich zwei Zeitachsen:

- Simulationszeit
- Deltazeit (delta delay)

Deltadelays entstehen durch Abarbeitung mehrerer Ereignisse, die zur gleichen Simulationszeit auftreten

Verifikation durch Simulation



- Eingangsstimuli erzeugen
- Ausgangssignale beobachten/verifizieren

Formale Verifikation

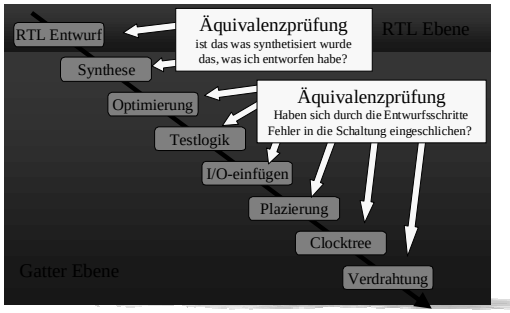
- Äquivalenzprüfung
  - sind zwei Schaltungen äquivalent
- Modellprüfung
  - erfüllt eine Schaltung bestimmte (sequentielle) Eigenschaften
- Theorembeweisen
  - interaktive Methode, basierend auf Logik höherer Ordnung

Was ist Äquivalenzprüfung

Engl. „*equivalence checking*“ (EC)

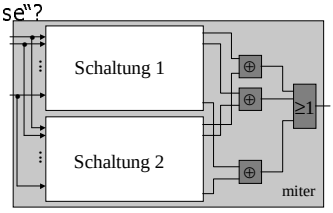
- gegeben: zwei digitale Schaltungen
- gefragt: haben beide die gleiche Funktionalität (keine zeitliches Verhalten)
  - bei kombinatorischen Schaltungen: sind die Ausgänge bei gleichen Eingangsbelegungen gleich?
  - Bei sequentiellen Schaltungen: sind die Ausgänge zu allen Zeitpunkten bei gleichen Eingabefolgen identisch?

Designflow mit Äquivalenzprüfung



Äquivalenzprüfung

- Erzeuge „miter"
- Ist der Ausgang für alle Eingangsbelegungen „false"?



Äquivalenzprüfung von Schaltnetzen

Vorgehen

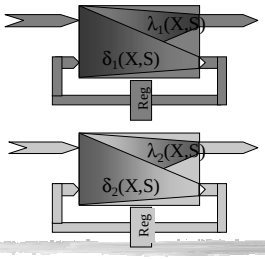
- Transformation der Schaltnetze in eine Normalformdarstellung (z.B. KNF, ROBDDs,...)
- Liegt Äquivalenz der Normalformen vor?

Optimierungen

- Ausnutzen von strukturellen Ähnlichkeiten
- inkrementelle Verfahren

Äquivalenzprüfung für Schaltwerke

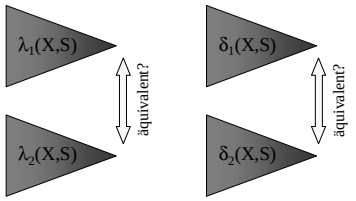
- Schaltungen in Huffman-Normalform



Äquivalenzprüfung für Schaltwerke II

Gleiche Zustandskodierung und eindeutiger Resetzustand: Reduktion auf Äquivalenzprüfung von Schaltnetzen

Für jeden Ausgang und für jedes FlipFlop:



Äquivalenzprüfung für Schaltwerke IV

Was ist, wenn man keine gleiche Zustandskodierung hat?

Oder keine gemeinsame Rücksetzleitung?

⇒ Einsatz von Techniken und Methoden der Automatentheorie

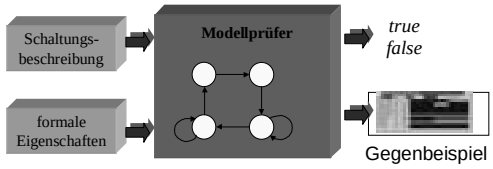
- Produktautomaten bilden mit „miter"-Ausgang
- Durchsuchen des Zusandsraumes nach Zuständen, die „true" am Ausgang erzeugen

Modellprüfung

Erfüllt ein Schaltwerk eine gegebene Spezifikation?

- Modellierung des Schaltwerkes durch endliche Automaten
- Spezifikation der Eigenschaften in temporalen (modalen) Aussagenlogiken

Modellprüfung



Temporale Aussagenlogik (LTL)

- atomare Ausdrücke
  - Signale des Designs
  - Boole`sche Funktionen, z.B. (a<b), OneHot(a,b,c)
- Boole`sche Operatoren
  - z.B. $\neg\varphi$, $\varphi\wedge\psi$, $\varphi\vee\psi$, ...
- temporale Operatoren $m \in N$, $n \in N \cup \{\infty\}$
  - $X_{[m]}\,\varphi$ in genau m Zeitschritten gilt φ
  - $F_{[m,n]}\,\varphi$ im Intervall [m,n] gilt φ mindestens einmal
  - $G_{[m,n]}\,\varphi$ im Intervall [m,n] gilt immer φ
  - $\varphi\,U_{[m,n]}\,\psi$ ψ wird im Intervall [m,n] wahr und bis dahin gilt immer φ

Temporale Aussagenlogik (LTL)

Typische Eigenschaften

- Sicherheit:
  a und b werden nie gleichzeitig wahr
  $G\,\neg(a\wedge b)$
- Lebendigkeit
  jede Anforderung wird innerhalb von 5 bis 7 bestätigt
  $G\,(req \rightarrow F_{[5,7]}ack)$
- Fairness:
  Innerhalb von 30 Schritten gilt mindestens einmal a

Semantik von LTL

Definiert über unendlichen Signalfolgen

$Sig = \{ a, b, c, ... \}$ Menge von Bezeichnern für Signale

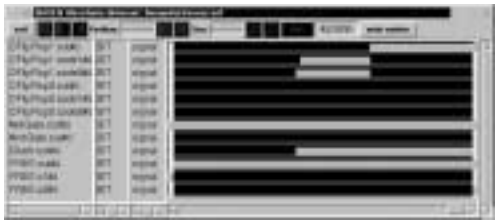
Signalfolge π weist jedem Signal $a \in Sig$ zu jedem Zeitpunkt $t \in N$ einen Boole`schen Wert zu

$\pi: Sig \times N \rightarrow B$

Der Suffix einer Signalfolge ab der Zeit t ist definiert durch

$\pi^t(a,z) = \pi(a,z+t)$

Signalfolgen



Semantik von LTL

ist definiert durch die Modellrelation $\models$

$\pi \models a \quad \Leftrightarrow \quad \pi(a,0)=true$

$\pi \models \neg\varphi \quad \Leftrightarrow \quad \pi \not\models \varphi$

$\pi \models \varphi\wedge\psi \quad \Leftrightarrow \quad \pi \models \varphi$ und $\pi \models \psi$

$\pi \models X_{[m]}\,\varphi \quad \Leftrightarrow \quad \pi^m \models \varphi$

$\pi \models F_{[m,n]}\,\varphi \Leftrightarrow \exists\, m \le t \le n \,.\, \pi^t \models \varphi$

$\pi \models G_{[m,n]}\,\varphi \Leftrightarrow \quad \forall\, m \le t \le n \,.\, \pi^t \models \varphi$

$\pi \models \varphi\,U_{[m,n]}\,\psi \Leftrightarrow \quad \exists\, m \le t \le n \,.\, \pi^t \models \psi$ und $\forall\, k < t \,.\, \pi^k \models \varphi$

Signalabfolgen und Schaltung

- Die Schaltungsbeschreibung enthält freie Eingäng
  - → beliebig viele Signalabfolgen können aus der Schaltung entstehen
  - → alle Signalabfolgen müssen die LTL-Formel erfüllen
  - → Beweis nicht über den Einzelnen Signalabfolgen, sondern direkt auf dem formalen Modell (endlichen Automaten)

Theorembeweisen

- Modellierung der Schaltung in Logik Höherer Ordnung
- Modellierung der Eigenschaften in Logik höherer Ordnung
- interaktiver Beweis durch Anwendung von Theoremen
- oft kombiniert mit
  - Heuristiken zur Automation des Beweises
  - automatischen Entscheidungsprozeduren

Timinganalyse

- Wie schnell Propagieren sich Signalwechsel zwischen den Eingängen, den Ausgängen und den Registern
  - → Bestimmung der Taktrate

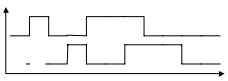
Zeitmodelle für Gatter

- Totzeit
  - das Eingangssignal erscheint zeitverschoben am Ausgang
- Totzeiten für steigende und fallende Taktflanken
  - steigende und fallende Taktflanken werden unterschiedlich verzögert
- träge Totzeit
  - kurze Impulse werden nicht an den Ausgang propagiert

Zeitmodelle für Gatter

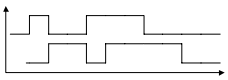
- Totzeit

$\delta_t=2$



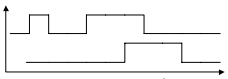
- Totzeiten für steigende und fallende Taktflanken

$\delta_\uparrow=1, \delta_\downarrow=2$



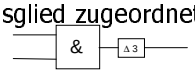
- träge Totzeit

$\delta_{min}=2, \delta_t=2$

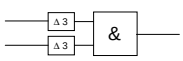


Einfaches Totzeitmodell

- Jedes Gatter arbeitet ohne Verzögerung
- Jedem Gatter ist ein externes Verzögerungsglied zugeordnet



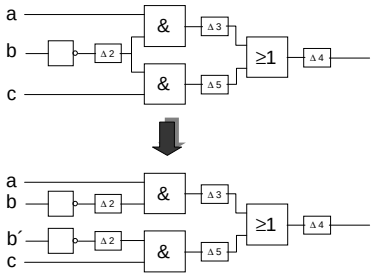
äquivalent zu



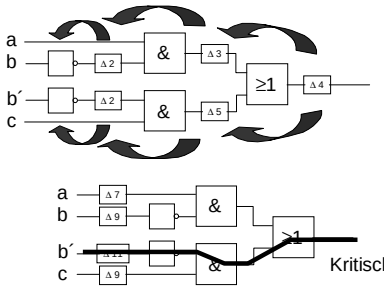
Timinganalyse

- duplizieren von mehrfach benutzen Schaltungsteilen und Eingängen
- verschieben der Verzögerungszeiten zu den Eingängen der Gatter
- addieren von Verzögerungszeiten auf einer Leitung

Duplizieren von Schaltungsteilen



Addieren der Verzögerungszeiten



Sensibilisierbarkeit

Ist der längste Pfad auch sensibilisierbar?

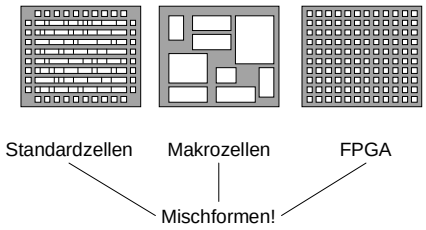
D.h. gibt es ein Eingangssignalwechsel, der sich über den längsten Pfad zum Ausgang durchsetzt?

➔D-Kalkül

1 Hardwareentwurf

- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung

Realisierungsformen



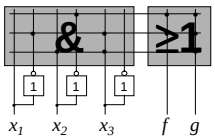
Makrozellen

- Standardisierte Funktionale Einheiten
  - Arithmetische Funktionsblöcke
  - I/O-Interfaces, ...
- PLA
  - programmable logic arrays
  - 2-stufige Logik
- RAM, ROM

Realisierung als PLA

- 2-Stufige Logik läßt sich auf einem ASIC als programmable logic array realisieren

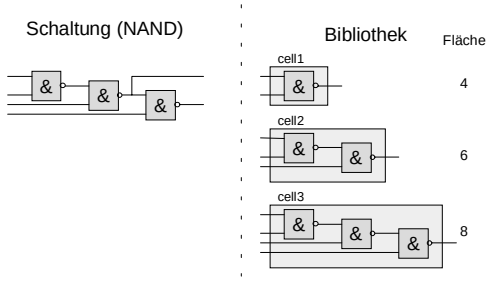
$f = \overline{x_2} \vee \overline{x_1} x_3 \qquad g = \overline{x_2} \vee \overline{x_1} x_3 \vee x_2 x_3$



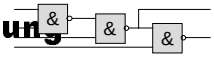
Standardzellen

- Der ASIC-Hersteller kann nicht beliebige Logische Gatter produzieren, deshalb wird vom Hersteller eine Zellbibliothek zur Verfügung gestellt, die alle möglichen Gatter enthält.
- → Abbildung der synthetisierten Gatternetzliste auf die Zellbibliothek
  - → Transformation der Schaltung und der Bibliothek in eine einheitliche Darstellung (z.B. NAND-Gatter)
  - → Finden einer Überdeckung (NP-vollständig)

Bibliotheksabbildung



Bibliotheksabbildung



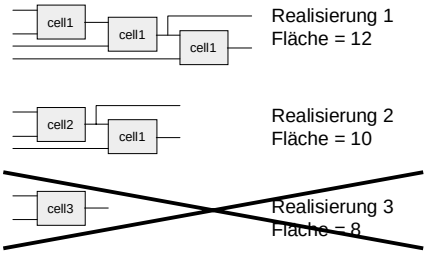
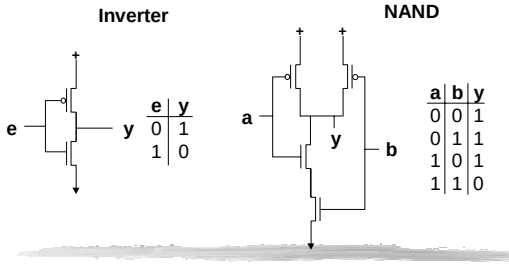
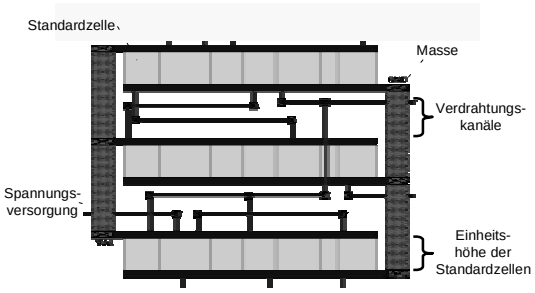


Abbildung auf Technologie

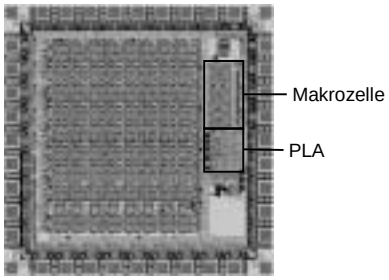
- CMOS



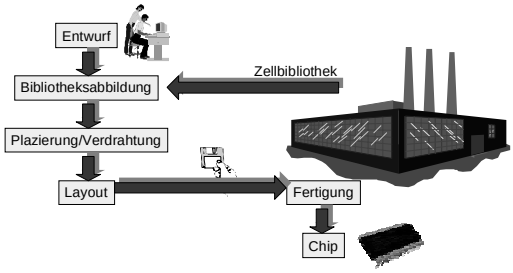
Standardzellen Layout



Standardzellen Layout

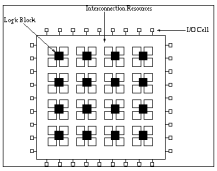


Standardzellen Flow



Realisierung als FPGA

- FPGA (field programmable gate arrays)

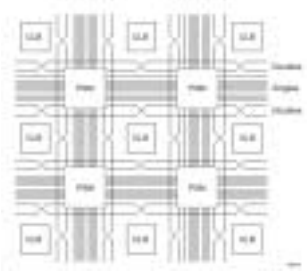


- Verbindungen und Logikblöcke sind programmierbar

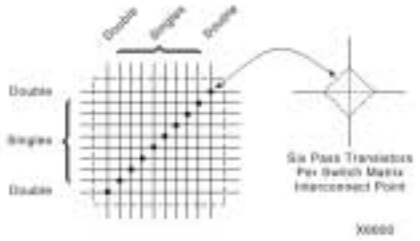
Beispiel: Xilinx 4000E

- CLB = Configurable Logic Block
  - zwei 4-input LUTs (LookUp Tables) und eine 3-input LUT
  - zwei SR D-FlipFlops
  - Bypass-Pfade und carry/cascade-Logic
- PSM = Programmable Switch Matrix
  - 10 Verbindungspunkte pro Matrix
  - Jeder Verbindungspunkt enthält 6 Passtransistoren
    - → jede Verbindung zwischen vier Richtungen möglich

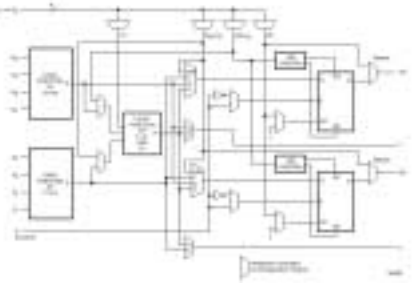
Xilinx 4000E Architektur



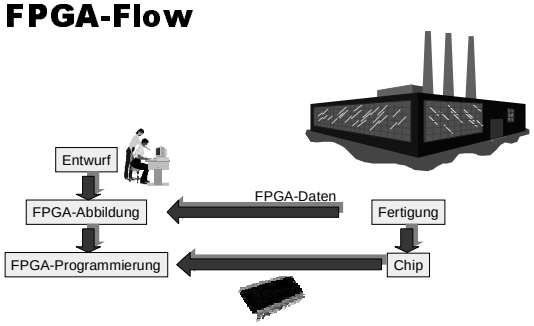
Xilinx 4000E Verbindungsmatrix



Xilinx 4000E CLB



FPGA-Flow



Partitionierung

- umfangreiche Schaltungen sind zu komplex um als „Ganzes" plaziert werden zu können
- ➔ Partitionierung des Problems
  - Zuordnung von n Objekten $O=\{o_1, \ldots, o_n\}$ zu m Partitionen $P=\{p_1, \ldots, p_m\}$, so daß
    $p_1 \cup \ldots \cup p_m = O$
    $p_1 \cap \ldots \cap p_m = \varnothing$
    die Kosten $c(P)$ minimal sind

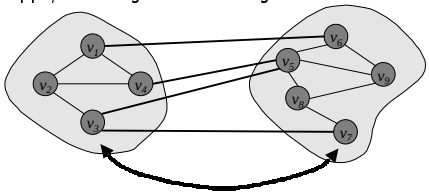
  ➔ das allgemeine Partitionierungsproblem ist NP vollständig

Allgemeine Partitionierungsverfahren

- **exakte Lösungsverfahren**
  - Enumeration der Lösungen
  - Integer Linear Programs (ILP)
- **heuristische Lösungsverfahren**
  - konstruktive Verfahren
    - random mapping
    - hierarchical clustering
  - iterative Verfahren
    - Kernighan-Lin Algorithmus
    - Simulated Annealing
    - evolutionäre Algorithmen

Kernighan-Lin

- Erzeugung von Bipartitionen:
  vertausche diejenigen Objekt in die jeweils andere Gruppe, die den größten Kostengewinn verursachen



Kernighan-Lin - Erweiterung

- Vertausche diejenigen Objekt, die den größten Kostengewinn oder den kleinsten Kostenzuwachs verursachen
- solange eine bessere Partition gefunden wird:
  - vertausche versuchsweise jede Paarung
  - nimm von diesen *(Versuchs-)*Partitionen diejenige mit dem besten Kostenverhältnis und führe die entsprechenden Umgruppierungen durch
  - einmal vertauschte Objekte werden im weiteren Verlauf nicht wieder vertauscht

Kernighan-Lin

- entkommt aus lokalen Minima
- Zeitkomplexität $O(n^3)$
- Partitionierung in m Blöcke: $O(m \cdot n^3)$

Simulated Anealing

- simuliertes Ausglühen
  - Metalle und Glas nehmen beim Abkühlen unter bestimmten Bedingungen einen Zustand minimaler Energie ein:
    - bei jeder Temperatur wird ein thermodynamisches Gleichgewicht erreicht
    - die Temperatur wird beliebig langsam erniedrigt
  - Wahrscheinlichkeit, dass ein Teilchen in einen Zustand höherer Energie springt

$$P(e_i,e_j,T) = e^{\frac{e_j - e_i}{kT}}$$

Simulated Anealing

Anwendung auf kombinatorische Optimierung

- Energie = Kosten der Lösung
- Verringerung der Kosten mit simulierter Temperatur, aber manchmal auch akzeptieren von Kostenerhöhungen

Simulated Anealing

```
temp = temp_start
cost = c(P)
WHILE (Frozen() == FALSE) {
  WHILE (Equilibrium() == FALSE) {
     P' = RandomMove(P)
     cost' = c(P')
     deltacost = cost' - cost
     IF (Accept(deltacost,temp) >
  random[0,1)) {
           P = P'
           cost = cost'
     }
  }
  temp = DecreaseTemp(temp)
}
```

$$min(1, e^{\frac{deltacost}{k*temp}})$$

Simulated Anealing

- Abkühlung: DecreaseTemp(), Frozen()
  - temp_start = 1.0
  - temp = α • temp (typisch: 0.8 = α =0.99)
  - Abbruch bei temp < temp_min oder wenn sich keine Verbesserung mehr ergibt
- Gleichgewicht: Equilibrium()
  - nach bestimmter Anzahl von Iterationen
  - oder wenn sich keine Verbesserung mehr ergibt

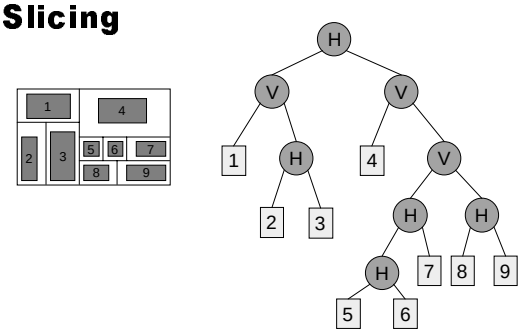
Simulated Anealing

- Zeitkomplexität
  - von exponentiell bis konstant, je nach Implementierung der Funktionen Equilibrium, DecreaseTemp, Frozen
  - je länger die Laufzeit, desto besser die Ergebnisse
  - üblich: Funktionen so konstruiert, dass polynomielle Laufzeit erreicht wird

Plazierung durch Slicing

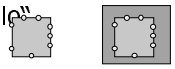
- Unterteile die Chipfläche in zwei Hälften
- Partitioniere die Objekte so in zwei Partitionen, dass
  - beide Partitionen etwa die gleiche Fläche benötigen
  - die Zahl der Verbindungen zwischen beiden Partitionen minimal ist
- wiederhole diese Schritte, bis Partitionen klein genug sind, um sie zu platzieren

Slicing



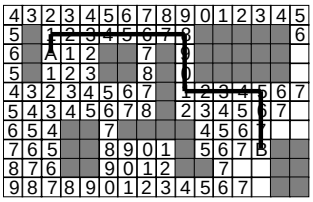
Slicing

- Abschließend werden
  - genaue Position
  - Orientierung
  - Seitenverhältnis festgelegt
- Beim Positionieren muß auch Platz für die Verdrahtung eingerechnte werden, z.B. „halo"



Verdrahtung

- Verfahren nach Lee



Verdrahtung

- High-Tower-Algorithmus
- Channel-Routing zweilagig
- River-Routing einlagig