

# Verilog Crash-Kurs

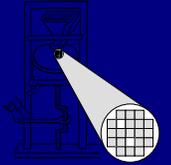
zum Praktikum  
Adaptive Computersysteme

Sommersemester 2006

Holger Lange

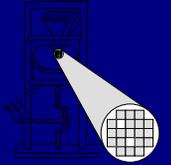
FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt

# Übersicht



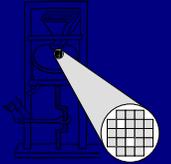
- Verilog zwischen „SDL“ und HDL
- Synthetisierbare Sprachteile
- Kombinatorische Logik
- Sequenzielle Logik
- Beispiel: Pipeline

# Verilog zwischen „SDL“ und HDL



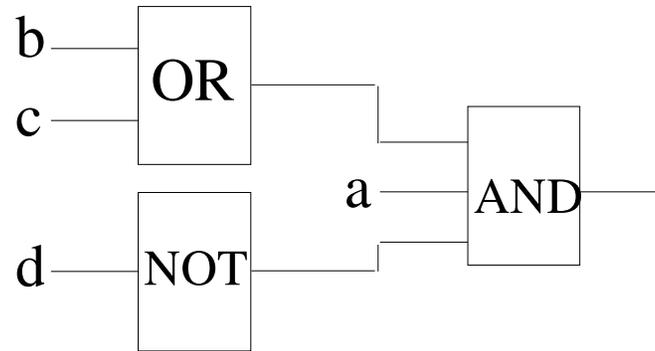
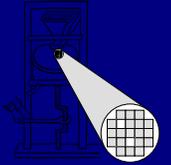
- Verilog ist viel mächtiger als zur reinen Hardware-Beschreibung nötig
    - Wichtig zur Modellierung von Systemumgebungen, „Testbenches“
    - Im Praktikum schon vorgegeben, wird nicht gebraucht
  - Von den reinen HW-Beschreibungselementen nicht alle übersetzbar (Synthese)
    - z.B. Modellierung von Zeit („#10“) in HW nicht garantierbar
- Synthetisierbare Untermenge von Verilog

# Synthetisierbare Sprachteile



- Synthese: Abbildung von Register-Transfer-Logik (RTL) auf Gatternetzlisten
- RTL in Verilog
  - Datentypen *wire* und *reg*
  - Permanente Zuweisung *assign* beschreibt kombinatorische Logik
  - *always* Block beschreibt prozedural kombinatorische **oder** sequenzielle Logik
    - In kombinatorischem Block auch *integer* als Hilfsvariablen zulässig, z.B. Schleifenzähler
  - Hierarchie durch Module und Instanzen
    - *parameter / defparam*
  - Präprozessor *`define NAME / `NAME*

# Kombinatorische Logik I



- Funktion  **$y = a \& (b \mid c) \& (!d)$**
- 1. Möglichkeit: Permanente Zuweisung  
*wire y;*  
*assign y = a && (b || c) && (!d);*
- Oder kürzer (Def. und Zuweisung kombiniert)  
*wire y = a && (b || c) && (!d);*



- 2. Möglichkeit: Kombinatorischer *always*-Block

```
reg y; // hier wird kein Flip-Flop erzeugt  
always @(a or b or c or d) begin // komplett!  
    y = a && (b || c) && (!d);  
end
```

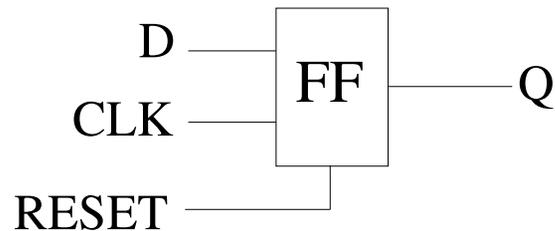
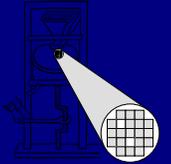
- Schleifenzähler: Beispiel „Reverse“ Leitfaden
- In kombinatorischen *always*-Blöcken **immer** blockende Zuweisung „=“ verwenden
  - Wie in C oder Java ...

# Kombinatorische Logik III



- Wie in C oder Java ...
  - $\&\&$ ,  $\|$ ,  $!$  usw. sind **logische** Operatoren und arbeiten auf den gesamten Ausdrücken
  - $\&$ ,  $/$ ,  $\sim$  usw. sind **bitweise** Operatoren und werten die Ausdrücke Bit für Bit aus
  - Logischer Vergleich:  $==$ ,  $!=$ 
    - **Nicht**  $===$ ,  $!==$  verwenden!
  - Multiplexer: Bedingung ? wahr : falsch
- Vorsicht bei komplexeren Ausdrücken
  - $+$ ,  $-$  erzeugen oft Addierer;  $*$ ,  $/$  Multiplizierer (groß) Dividierer oft gar nicht möglich
  - Besser: 2er Potenzen ausnutzen „ $\ll$ “ „ $\gg$ “

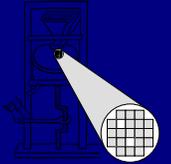
# Sequenzielle Logik I



- D-Flip-Flop mit asynchronem Reset
- Sequenzieller *always*-Block

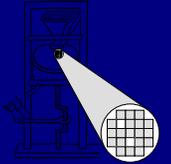
```
reg Q;  
always @(posedge CLK or posedge RESET) begin  
  if (RESET == 1) begin  
    Q <= 0;  
  end  
  else begin  
    Q <= D;  
  end  
end
```

# Sequenzielle Logik II



- In sequenziellen *always*-Blöcken **immer** nicht-blockende Zuweisung „ $\leftarrow$ “ verwenden
  - Wertet in jedem Zeitschritt zunächst alle Ausdrücke auf der rechten Seite aus
  - Transparente Zwischenspeicherung
  - Schließlich Zuweisung an die linke Seite
  - Modelliert die Zeitverzögerung beim Laden eines Flip-Flops
- Sei  $B := 3, A := 2$ 
  - $A = 4; B = A; \rightarrow$  B hat den Wert 4
  - $A \leftarrow 4; B \leftarrow A; \rightarrow$  B hat den Wert 3
    - Im nächsten Zeitschritt ist  $A=4$  und B hat den Wert 2!

# Sequenzielle Logik III



- Zustandsautomaten

- Werte für Zustandsvariable (Typ *reg*) mit *`define* kodieren

- *always @(posedge CLK or posedge RESET) begin*  
*if (RESET) STATE <= `IDLE;*

- else begin*

- case (STATE)*

- `IDLE: STATE <= `GO;*

- `GO: begin*

- STATE <= ...;*

- ...*

- end*

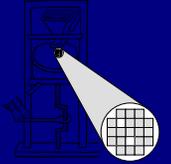
- default: STATE <= 'bx; // darf nicht auftreten***

- endcase*

- end*

- end*

# Beispiel: Pipeline



- Tafel