Universität Siegen – Fachbereich Elektrotechnik und Informatik
Dipl.-Inform. „Benedikt Meurer" : „(Pro)Seminar"
WS 2011/12

VERILOG-HDL

Mehdi Khayati Sarkandi

Inhalt

1 Einleitung

Jede komplexe Softwareprogrammierung braucht eine entsprechende Hardwarestruktur. Neue Anwendungen erfordern neue Hardware. Deshalb stehen die Hardwareprogrammierer vor der Herausforderung, auch die Programme für die Herstellung solcher Hardware entsprechend anzupassen. Zu diesem Zweck wurden die Hardwarebeschreibungs-sprachen (HDL) entwickelt. Verilog ist eine von diesen Sprachen, die beschreibt, wie die logischen Schaltungen aufgebaut und miteinander verknüpft sind.

Die vorliegende Arbeit ist der Versuch einer kompakten, strukturierten Darstellung der Hardwarebeschreibungssprache Verilog-HDL. Nach einem kurzen Überblick über die Hardware-Herstellungswerkzeuge (Kapitel 2) wird auf die Entstehung und Entwicklung von Verilog eingegangen (Kapitel 3). Anschließend werden die besonderen Merkmale von Verilog herausgearbeitet und vielfach mit Programmierungsbeispielen und Schaltbildern illustriert (Kapitel 4).

2. Hardwarebeschreibung (HDL, Hardware Description Language)

Wie schon der Name sagt, ist HDL ein Werkzeug, um komplexe digitale elektronische Schaltungen zu entwerfen, daraufhin das Systemverhalten zu simulieren und jeden Schritt auf korrekte Funktionsfähigkeit zu prüfen (verifizieren).

Die HDL-Sprache bearbeitet - wie herkömmliche prozedurale Sprachen auch - Anweisungen sequentiell und kann zusätzlich Anweisungen zeitlich parallel bearbeiten.

HDL ermöglicht es auch, entworfene Schaltungen zu simulieren und zu verifizieren (auf Korrektheit zu prüfen), damit fehlerhafte Schaltungen vor dem physikalischen Implementieren verbessert werden können. So spart der Halbleiterhersteller enorme Herstellungskosten.

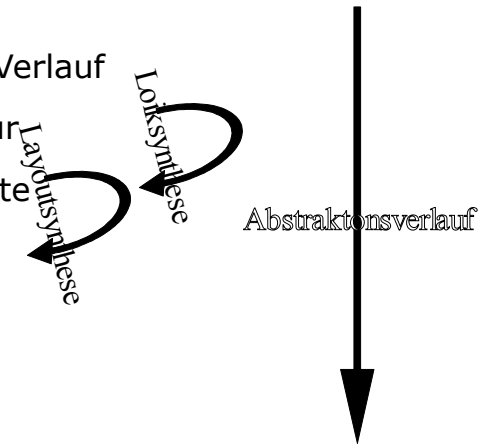
Verilog und VHDL sind die zwei bekanntesten Hardware-Beschreibungs-sprachen (HDL), die in den USA bzw. Europa verbreitet sind. Wegen ihrer leistungsfähigen Simulation, Verifikation und Synthese (bei der Synthese sind allerdings ein paar Regeln zu beachten) sind beide Hardware-beschreibungen Marktführer in der Industrie.

Neben der textuellen Beschreibung gibt es zwar auch grafische Metho-den, doch bei der Darstellung eines komplexen Designsystems verliert man schnell den Überblick. Deshalb hat sich die textuelle Beschreibung mit strukturierter Syntax im Bereich der Hardwarebeschreibung etabliert.

2.1 IC-Herstellung

Das untenstehende vereinfachte Flussdiagramm beschreibt Schritt für Schritt die Herstellung einer integrierten Schaltung (IC). Nach jeder Stufe kann anhand der Rückmeldung, die durch Simulation und Verifikation entsteht, ein sicherer Designablauf gewährleistet werden.

1. Design Idee: Verilog, VHDL
2. Verhaltensmodell: Pseudo-Code, grafischer Verlauf
3. Datenpfad-Design: Bus- und Registerstruktur
4. Logisches Design: FlipFlop und Gatternetzliste
5. Physikalisches Design: Transistor-Layout
6. Herstellung
7. Integrierter CHIP



Die Transformation zwischen dem 3. und 4. Schritt geschieht durch Logiksynthese und zwischen dem 4. und 5. durch Layoutsynthese (place and route).

2.2 Logiksynthese

Bei der Logiksynthese wird die Hardwarebeschreibung anhand eines Synthesewerkzeugs eingelesen und automatisch auf Registertransferebene von Schaltungen in Form von booleschen Gleichungen übersetzt. Der Prozess beinhaltet die Analyse, die Übersetzung in logische Operatoren und Speicher und die anschließende Eliminierung von Redundanzen. Damit entsteht eine Gatternetzliste mit logischen Funktionen der Gatter und deren Signalverzögerung (logische Gatter, FlipFlop etc.).

2.3 Layoutsynthese

In der Layoutsynthese wird mit Hilfe von Synthesewerkzeugen und der Verwendung von Bibliotheks- und Technologie-Informationen eine Gatternetzliste in einem geometrischen Layout auf dem Chip abgebildet. Ein Teil dieser Transformation heißt "Place & Route"; also das Platzieren und Verdrahten von Bauelementen, wobei darauf geachtet wird, die Bauelemente in optimaler Form zu platzieren, um möglichst wenig Platz zu beanspruchen. Die Verbindung ("Route") sollte auch so kurz wie möglich sein, um Signalverzögerungen zu minimieren.

2.4 Gemeinsamkeiten und Unterschiede zwischen VHDL und Verilog

Beide Sprachen dienen zur Beschreibung der Schaltung auf Gatterebene. Für beide Sprachen gilt: Die Richtung der Ports muss deklariert werden (input, output oder inout).

Unterschiede:

- Verilog hat mehrere vorteilhafte Merkmale in der unteren Ebene, die nicht in VHDL enthalten sind, so etwa primitive Module (AND, OR, ...), die schon vordefiniert sind und direkt eingesetzt werden können, sowie benutzerdefinierte Module, die man als Instanzen in andere Module einbinden kann.
- VHDL verlangt eine genaue Typisierung; Konstanten, Variablen und alle Signale bekommen einen festen Datentyp zugewiesen. Wichtige Datentypen in VHDL sind: std_ulogic und std_logic mit neun Zuständen (bei Verilog gibt es dagegen nur vier Zustände) sowie verschiedene vektorfähige Datentypen. Mit VHDL kann man sehr gut Datentypen selbst definieren.
- Bei VHDL wird die Port-Deklaration innerhalb des entity-Blocks und die Beschreibung des Verhaltens von Funktionen innerhalb des architecture-Blocks vorgenommen. Der architecture-Block wird durch den Namen des entity-Blocks verknüpft.
- Dagegen ist die Struktur bei Verilog einfacher: Deklarationen und Statements werden innerhalb eines modules nacheinander aufgelistet.
- VHDL bietet die Möglichkeit zur Einbindung von Bibliotheken, so dass das Schreiben von immer wiederkehrendem Code entfällt. Bei Verilog fehlt dagegen ein Bibliotheksmanagement.
- Deklarationen können in VHDL zu Paketen zusammengefasst werden (= package, wie z. B. Unterprogramme, Typen, Subtypen, Konstanten, Signale, Variablen, Files, Komponenten etc.). Sie werden mit use in den Code eingebunden.
- Ein weiterer bedeutsamer Unterschied sind automatische Typkonversionen. Verilog passt den Datentyp bei der Zuweisung automatisch an. Beispiel: Wenn bei der Zuweisung auf der linken Seite des Operators ein Real-Typ ist, werden die Variablen mit Datentyp Integer entsprechend konvertiert. Dagegen bleiben in VHDL die zuvor deklarierten Datentypen unveränderlich, so dass die Zuweisung unterschiedlicher Datentypen zu einer Fehlermeldung führen würde.
- VHDL verlangt mehr Sorgfalt und Übersicht beim Programmieren. Daher ist der Einstieg schwerer. Dagegen funktioniert Verilog intuitiver.

- Bei der Entscheidung zwischen VHDL und Verilog können auch die Vorkenntnisse der Programmierer eine Rolle spielen, da Verilog Ähnlichkeiten zu C aufweist und VHDL zu Ada und Pascal.
- Verilog ist case-sensitive, VHDL nicht.

Fazit:

- **VHDL** ist etwas höher angesiedelt und geht somit bis zur Systemebene.
- Wiederverwendbarkeit (package)
- Erfordert strengere Deklarationen als Verilog
- **Verilog** ist für eine schnelle simple Umsetzung gut geeignet.
- Bietet auch Einflussmöglichkeiten auf Gatterebene (UDP).

3 Entwicklung von Verilog

Verilog von Phil Moorby, dem Erfinder, wurde zuerst 1984 bei der Firma Gateway Design Automation entwickelt. In den Jahren 1995 und 2001 wurde Verilog als IEEE-Norm standardisiert. Es gibt zahlreiche Ergänzungen in der Version von 2001 gegenüber der von 1995. Und "im Jahre 2005 wurde SystemVerilog (IEEE-Standard 1800-2005) als Verilog-Derivat als eigener Standard veröffentlicht. SystemVerilog bietet gegenüber Verilog viele neue Funktionalitäten und ist nicht nur eine HDL zur Systemmodellierung, sondern auch eine Verifikationssprache." [2]

Verilog wird hauptsächlich in den USA eingesetzt, während in Europa VHDL die meistbenutzte Sprache ist.

Die Modellbildung für Synthese, Simulation und Verifikation ist unter anderem Teilaufgabe von Verilog.

Die Verilog-Sprache modelliert sowohl logische Funktionen auf Schaltkreisebene als auch Systemverhalten auf höherer Abstraktionsebene. Sie ähnelt sehr der Programmiersprache C, daher ist sie für die meisten Programmierer leicht zu lernen.

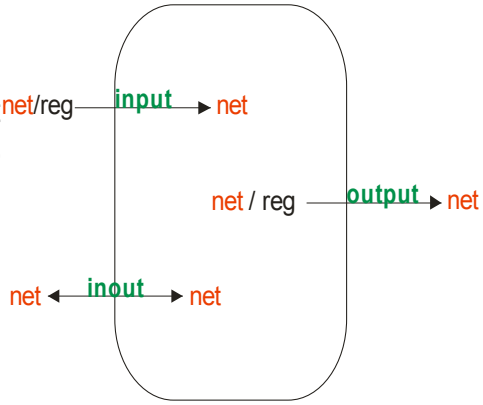
Ein besonderes Merkmal von Verilog im Vergleich zu üblichen Programmiersprachen ist die Spezifikation der parallelen Ausführung von Prozessen.

Die erweiterte Version von Verilog, Verilog-AMS, unterstützt eine Mischung von analog/digitalen Systemen unter anderem für Elektronik, Mechanik usw.

4 Aufbau der Verilog-Sprache

(Schlüsselwörter sind fett ausgedruckt.)

- Jeder Verilog-Code fängt mit dem Schlüsselwort "**module**" an und endet mit "**endmodule**". "Ein Modul kann ein gesamtes System, eine Leiterplatte, einen Chip oder auch nur ein logisches Gatter darstellen." [3]
- In der Parameterliste werden vor dem Module-Namen in Klammern die "Ports", die Ein-/Ausgänge von Modulen, aufgelistet und direkt im 'body' deklariert. Unter Ports sind Typen von Ein-/Ausgängen sowohl als Takt als auch als Variable vorstellbar. Es gibt außer **input** und **output** auch **inout**, ein bidirektionales Bussystem.



- Ports werden auch als Schnittstelle zwischen Modulen oder nach außen betrachtet. Sie haben zum Teil unterschiedliche Datentypen.
- Jeder Zeile im Programm-Code wird mit einem Semikolon "**;**" abgeschlossen.
- Kommentare kann man wie in der C-Programmiersprache auf zwei Arten einfügen, nämlich entweder indem man sie für einzeilige Kommentare hinter zwei Schrägstriche "**//**" stellt oder indem man für mehrzeilige Kommentare "Schrägstrich + Stern" vor den Anfang und "Stern + Schrägstrich" nach dem Ende des Kommentars "**/* .... */**" eingibt.
- Eine einfache Modulbeschreibung sieht folgendermaßen aus.

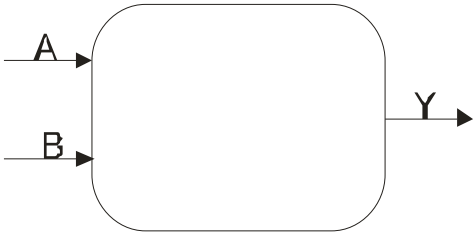
module *module-Name (Liste von Ein- und Ausgängen)*

Programm-Code; // *body*

endmodule

Beispiel:

```
module TestModule (A, B, Y)
    input A, B;
    output Y;
endmodule
```



4.1 Bezeichner

Mit Bezeichner (Identifier) werden Objekte in Verilog definiert. Allerdings muss beachtet werden, dass sie keine Schlüsselwörter sein dürfen. Sie müssen mit einem Buchstaben oder Unterstrich anfangen. Für Bezeichner können außerdem auch die Ziffern von 0 bis 9 benutzt werden. Vorsicht ist geboten bei Klein- und Großschreibung, denn Verilog ist "case sensitive" und unterscheidet zwischen Klein- und Großschreibung, was zu Fehlern führen kann.

Beispiel: nametest, _testname, N52 usw.

4.2 Zahlen

Wie die meisten gewöhnlichen Programmiersprachen erlaubt auch Verilog die Benutzung aller vier Zahlensysteme, nämlich Dezimal-, Binär-, Oktal- und Hexadezimalzahlen.

Zahlen als Wertzuweisung werden in zwei unterschiedlichen Formen darstellbar:

1. in Form von: Size'Basis Wert (z. B.: abc = 4'b0110, d. h. 4-bit breite binäre Zahl), analog dazu werden h, d, o für hexadezimal, dezimal und oktal verwendet;
2. oder als ganze Zahl: abc = 6.

4.3 Operatoren

Wie in der Mathematik bzw. üblichen Programmiersprachen dienen auch hier Operatoren zum Verändern bzw. Erzeugen neuer Werte von Variablen.

Man kann Operatoren allgemein in drei verschiedene Kategorien einordnen:

1. unary operator: Operation für nur einen Operanden (Beispiel siehe unten, Reduktions-Operatoren)
2. binary operator: Operation für zwei Operanden
3. ternary operator: Operation für drei Operanden

Zu den Operatoren gehören:

4.3.1 Arithmetische Operatoren

Addition(+), Subtraktion(-), Division(/), Modulo(%), Multiplikation(*)

4.3.2 Logische Operatoren

Es ist zu beachten, dass bei logischen Operationen bzw. allen Operationen, die zu einem Zustandswechsel führen, vier Zustände existieren:

- logische 1 oder (true)
- logische 0 oder (false)
- x undefiniert (undefined), don't care
- z hochohmige (Tristate)

Zu den logischen Operatoren, die zu (true/false/undefined) als Ergebnis führen, gehören:

1. logisches UND (AND): **&&**
2. logisches ODER (OR): **||**
3. logische Negation (NOT): **!**

4.3.3 Bitweise Operatoren

Genau wie bei den gewöhnlichen Programmiersprachen gibt es hier auch Operatoren, die nur zwischen einzelnen Bits von Operanden agieren und ein Wortbit als Ergebnis ausgeben. Dazu gehören:

1. Negation (~)
2. UND (&)
3. ODER (|)
4. XOR (^)
5. XNOR (^~)

4.3.4 Reduktions-Operatoren

Reduktions-Operatoren führen eine Vektor-Größe von Signalen bitweise zusammen und geben als Ergebnis eine ein-bit-skalare Größe aus.

Beispiel: a= 3'b101; &a ist gleich 1'b0

4.3.5 Bedingte Operatoren

In Verilog stehen folgende Bedingte Operatoren zur Verfügung:

1. **if** (Bedingung==wahr): Anweisung // ausführen, wenn wahr
2. **else**: Anweisung // ausführen, wenn falsch
3. Bedingung **?** Anweisung (wenn wahr) **:** Anweisung (wenn falsch)

4.3.6 Vergleich-Operatoren

Zusätzlich zu den üblichen Vergleich-Operatoren der höheren Programmiersprachen, wie: größer als (>), kleiner als (<), größer als oder gleich groß (>=), kleiner als oder gleich groß (<=), gleich (==), nicht gleich (!=), gibt es in Verilog noch weitere Vergleich-Operatoren, die einen exakten bitweisen Vergleich (Identitätsoperator) durchführen:

bitweise gleich (===) und bitweise nicht gleich (!==)

Wie schon oben erwähnt, gibt es außer "logisch-wahr" (1) und "logisch-falsch" (0) noch (x) undefiniert und (z) hochohmig als logische Signalzustände. Deshalb muss man bei normalen Vergleich-Operatoren darauf achten, dass das Ergebnis auch unbekannt wird, wenn einer der beiden Operanden einen Zustand x oder z beinhaltet. Und beim bitweisen Vergleich wird jedes Bit, falls es x oder z enthält, auf Gleichheit verglichen.

Beispiel: (3'b01x === 3b'01x) liefert wahr,
(3'b01z === 3'b01x) liefert falsch.

4.3.7 Auswahl-Operatoren

funktionieren genauso wie bei herkömmlichen Programmiersprachen *"switch/case/default" indem sie* Alternativen zur Auswahl stellen. In Verilog übernimmt diese Aufgabe der "case"-Konstruktor für zutreffende Signale; für den Fall, dass keine der Auswahlen zutrifft, schreibt man eine "default"-Anweisung.

Beispiel:
```
case (select)
        1'b0: y = x1;
        1'b1: y = x2;
endcase
```

4.3.8 Schleifen-Operatoren

Der "**for**"-Operator unterstützt Verilog im Falle von Schleifen mit der Registertypvariable und wird genauso wie in der C-Programmierung aufgestellt.

Der "**while**"-Operator übernimmt die bedingten Schleifen, Aufstellung analog der C-Programmierung.

Der "**repeat**"-Operator ist für die wiederholte Ausführung einer Anweisung zuständig; dazu muss die genaue Anzahl der Wiederholungen angegeben werden.

Beispiel:

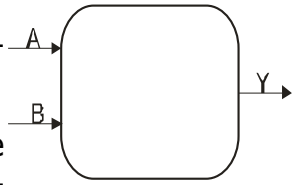
```
reg Konto = 0
repeat (3)
    begin
        konto = konto +2;
    end   // konto == 6
```

4.4 Datentypen

In Verilog existieren zwei Arten von Datentypen, nämlich Variablen, die auch in gewöhnlichen Programmiersprachen verwendet werden, und außerdem der Datentyp Signal. Signale unterscheiden sich in der Art, wie sie ihre Werte zugewiesen bekommen und wie lang sie sie beibehalten.

Die Größe der Signale ist im Vergleich zu gewöhnlichen Variablen nicht nur ein Zahlenwert, sondern zugleich eine von der Zeitbasis abhängige Größe. D. h., deren Werte sind nur so lange gültig, wie deren Zustand kontinuierlich durch ein logisches Gatter oder einen Treiber in Betrieb gehalten wird. Ansonsten leiten sie keine Information mehr durch das Netz und funktionieren wie ein hochohmiger Widerstand, und ihr Wert wird nur dann geändert, wenn ihr Zustand in ein logisches Gatter geändert wird.

Wenn z. B. Y ein Signaltyp ist, dann ist sein Wert während der Betriebszeit abhängig von A und B.



Variablen behalten dagegen ihren Wert, nachdem sie ihn zugewiesen bekommen haben, so lange, bis wieder ein neuer Wert zugewiesen wird. Variablen repräsentieren meistens einen Speicher.

Aus diesem Grund werden die Datentypen in Verilog in zwei Gruppen repräsentiert:

net-Typen und **Register**-Typen.

4.4.1 net-Typen

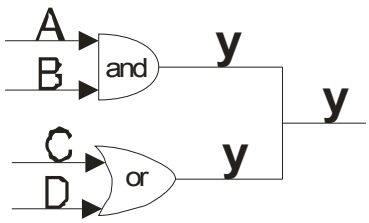
Datentypen **net**, die elektrische Signale zwischen den Komponenten repräsentieren. **net**-Typen werden meist im "Strukturdesign" für die Verbindung zwischen Modulen verwendet.

Für **net**-Datentypen gibt es vordefinierte Netz-Datentypen; einige wichtige Typen sind:

- **wire**: stellt wie ein Draht eine Verbindung zwischen Modulen her und leitet nur Signale, kann sie aber nicht speichern.
- **wor**: stellt den Ausgang als "ODER_Gatter" dar.
- **wand**: stellt den Ausgang als "UND_Gatter" dar.
- **Supply0**: Verbindung mit Masse
- **supply1**: Verbindung mit Power

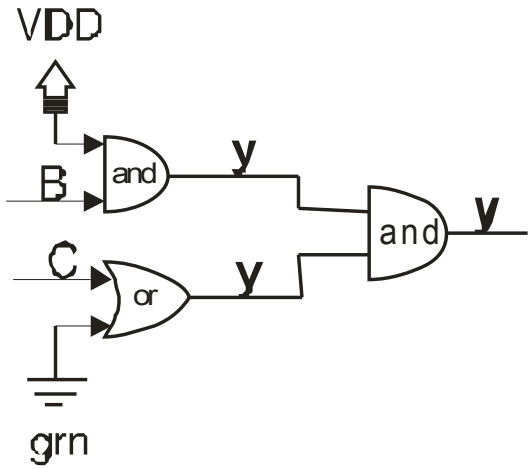
Beispiel für **wire**:

```
module wire_test (A,B,C,D,y)
        input A, B, C, C;
        output y;
        wire y;
        assign y=A and B;        // assign-Konstrukt ist eine dauerhafte
        assign y=C or D;         // Zuweisung für wire-Variablen
endmodule
```



Beispiel für **wand**:

```
module wand_test (B,C,y)
        input  B, C;
        output y;
        wand y;
        supply1 VDD;
        supply0  grn;
        assign y=VDD and B;
        assign y=C or grn;
endmodule
```



Beispiel für **wor**: Analog zu **wand**.

4.4.2 Register-Typen

stellen hauptsächlich einen Speicher in einem Modul dar. Sie behalten ihre vorher zugewiesenen Werte. Register-Typen werden meist im "Verhaltensdesign" verwendet.

Einige vordefinierte Register-Typen sind:

- **reg**: Beim Deklarieren muss man die Wortbreite explizit angeben, sonst bekommt er die Default-Größe ein bit. "**reg"** ist eine vorzeichenbehaftete Größe. Man verwendet sie meistens beim Hardwareregister bzw. Akkumulator ("Register, das für logische und arithmetische Operationen konzipiert ist, gewöhnlich zum Zählen von Elementen und zur Berechnung von Summen" [1]).

  Beispiel: **reg** [7:0] outBus; // Variabler outBus mit 8 bit Breite

- **integer**: eine vorzeichenbehaftete Variable mit Default-Größe 32 bit. Um negative Zahlen darzustellen, wird sie mit 2er-Komplement erweitert. **integer**-Variablen stellen ganze Zahlen für die Berechnung dar und werden meistens als Schleifenindizes oder für die kontinuierliche Berechnung verwendet, selten in elektronischen Schaltungen.
- **time**: für Verzögerungszeit oder Simulationszeit
- **real**: analog zu **integer** mit einer Breite von 64 bit
- **event**: zum Repräsentieren von Ereignissen, wie bei Zustandswechseln von logischen Signalen.

Für die konstanten Variablen gibt es das Konstrukt **parameter**. Mit **parameter** können sowohl ganze Zahlen und physikalische Größen als auch ein String deklariert werden.

Beispiel: **parameter** abc = 3'b100;

4.5 Verilog-Primitive

Bei Primitiven, die schon oben erwähnt wurden, handelt es sich um vordefinierte funktionale Module, die beim "Strukturdesign" eingesetzt werden.

Beispiel: **and** (y, a, b);

Die ersten Elemente in der Portliste stehen für den Ausgangsport und weitere Elemente stehen für die Eingänge. Es gibt auch die Möglichkeit, USER-definierte Module zu erstellen:

Beispiel: **MUX** M1 (y, sel, a, b);

Hier ist M1 eine Instanz aus dem benutzerdefinierten MUX-Modul. Damit spart man das wiederholte Schreiben desselben Code-Moduls.

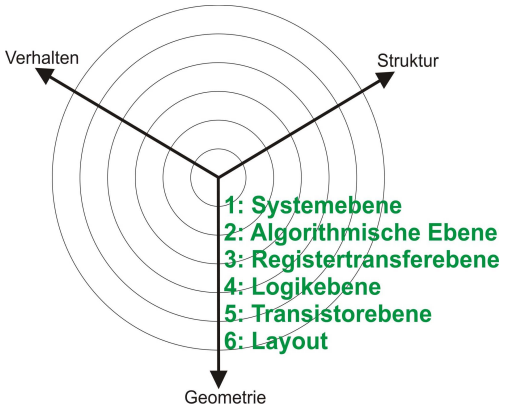
Es gibt 26 Primitive Module bei Verilog. Sie sind in drei Gruppen unterteilt:

1. **Logische** Gatter: and, or, xor, xnor usw. Sie erlauben mehr als zwei Eingänge, aber nur einen Ausgang.
2. **Treiber (Buffer)**: buf, not, pulldown, bufif0, notif0, pullup, bufif1, notif1
3. **Transistoren und Transfergatter (Schaltermodelle)**: nmos, pmos, cmos, rnmos, rpmos, rcmos, rpmos, rcmos, tran, tranif0, tranf1, rtran, rtranif0, rtrabif1

4.6 Designmodelle

Designmodelle (Abstraktionsebene) werden auf mehreren Stufen in drei verschiedenen Sichtweisen beschrieben, die auch als Y-Diagramm bekannt sind. Sie stellen unterschiedliche Abstraktionsebenen dar. Die Verbindung zwischen den Ebenen wird durch „front-end" und „back-end" realisiert. ("Die Begriffe **Front-End** und **Back-End** ... werden in der Informationstechnik an verschiedenen Stellen in Verbindung mit einer Schichteneinteilung verwendet. Dabei ist typischerweise das Front-End näher am Benutzer, das Back-End näher am System." [wikipedia])

- Verhaltensmodelle
- Strukturmodelle
- Physikalische Modelle (Geometrie)



Abstraktionsebene (Y-Diagramm)

	Verhaltensdesign	**Strukturdesign**	**Pysikalisches Design**
1.Systemebene	Systemspezifikation	CPUs, grobe Funktionsblöcke	Partitionierung
2.Algorithmische Ebene	Algorithmen	Subsysteme, Busse, Prozessor	**MCM** [Multi Chip Module]
3.Registertransferebene	Register-Transfers	Module, ALU, MUX, Register	ASICs, FPGAs
4.Logikebene	Boolesche Gleichung	Gatter, Flipflops	Zellen
5.Transistorebene (Schaltkreisebene)	Differentialgleichung, Transistorfunktion	Transistoren, Leitungsstücke	Rasterdiagramm für Transistorlayout
6.Layoutebene	Konkretes Layout mit Maskenebene		

4.6.1 Verhaltensmodell

- Das Verhaltensmodell ist ein Verilog-Modell, das das Verhalten von logischen Schaltnetzen beschreibt, ohne Informationen über die Realisierung bzw. Implementierung der Schaltung zu geben.
- Beim Verhaltensmodell ist keine Information über die Hardware nötig, sondern es geht darum, das Verhalten der Komponente darzustellen.
- Die Anwendung ist ähnlich wie bei herkömmlichen prozeduralen Programmiersprachen. Hier sei darauf hingewiesen, dass es darum geht, das Hardwareverhalten zu beschreiben und keine speziellen Computeralgorithmen zu schreiben.
- Verhaltensmodelle können auf zwei Arten dargestellt werden:
    1. Datenfluss-Modell
    2. algorithmisches Modell

4.6.1.1 Datenfluss-Modell

Das Datenfluss-Modell beschreibt das Schaltnetzverhalten im Register Transformationsebene, und zwar anhand kontinuierlicher Zuweisung (continuous assignment).

Kontinuierliche Zuweisungen werden durch das Konstrukt "**assign**" identifiziert. Die Form des **assign**-Konstrukts sieht wie folgt aus:

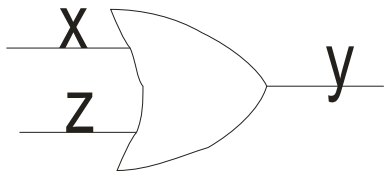
"**assign** Name [Verzögerung] = Ausdruck"

Dem **assign**-Konstrukt folgt der Name des Netzes. Optional kann man eine Verzögerung (delay) vor den Namen einsetzen. Auf die rechte Seite wird eine Variable oder ein Ausdruck gestellt.

Der Datentyp von Variablen auf der linken Seite ist immer ein Netztyp (Skalar oder Vektor), meistens **wire**. Demgegenüber kann auf der rechten Seite des Zuweisungsoperators sowohl ein **net**-Typ als auch ein **reg**-Typ stehen.

Beispiel für ein logisches OR:

```
module ASSIGN_OR(x, y, z);
    reg x;
    wire y, z;
    assign y = x | z;
endmodule
```



"continuous assignment" ist, wie schon der Name sagt, kontinuierlich aktiv. D. h., der Wert von "*y*" wird bei jeder Änderung von "*x*" bzw. "*z*" geändert und bleibt bis zur nächsten Änderung aktiv.

Der Verzögerungsoperator "**#**" (delay) in einem Verilog-Modell erzwingt eine Zeitverzögerung von längeren Zeiteinheiten. Die Zeitlänge hängt von der Zahl vor dem delay-Operator "**#**" ab, die Zeiteinheit wird mit der **timescale**-Anweisung festgelegt.

Die Verzögerung (delay) ist nur im Simulationsmodell gültig und nicht bei der Synthese.

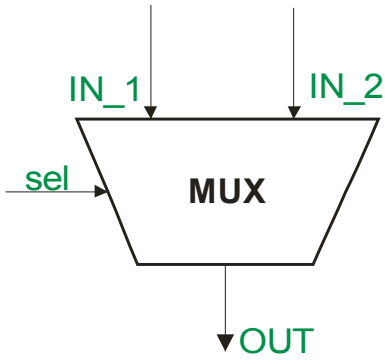
Beispiel: *assign #10 out = In1 & In2;*

Das bedeutet, Ausgang "out" soll erst nach 10 Zeiteinheiten durch logisches UND von "In1" und "In2" zugewiesen werden.

Ein Verilog-Modell kann mehrere "assignments" (kontinuierliche Zuweisungen) beinhalten. Man muss darauf achten, dass in kontinuierlichen Zuweisungen mit dem "**assign**"-Konstrukt nur die Werte von **net**-Typen zugeordnet werden können wie beim ***wire***-Signal.

Ein Beispiel für Multiplexer mit "**assign**"-Konstrukt:

```
module MUX-ASSIGN (IN_1, IN_2, sel, OUT);
        input [2:0] IN_1, IN_2;
        input sel;
        output [2:0] OUT;
        wire OUT;
        assign OUT = (sel) ? IN_1 : IN_2;
endmodule
```



4.6.1.2 Algorithmische Modelle

"beschreiben die Funktionen eines Systems, losgelöst von jeden Hardwarebezogenheiten über algorithmische Transformationen der Eingangssignale in die Ausgangsgrößen. Das System wird als Black-Box modelliert, die zwischen die Eingangs- und Ausgangsports geschaltet ist." [2]

4.6.1.3 Prozeduralblöcke

Bei den kontinuierlichen Anweisungen sind verschiedene Funktionen nicht realisierbar, z. B. wenn eine Funktion erst bei Flanken- oder Pegeländerung der Steuersignale ausgeführt wird oder sie zu komplex ist, als dass man sie mit einfachen booleschen, arithmetischen oder logischen Funk-

tionen realisieren könnte. Für solche Fälle setzt man Prozeduralblöcke ein.

Prozeduralblöcke können wie bei herkömmlichen Programmiersprachen aussehen, d. h., alles, was dort zwischen geschweiften Klammern "**{...}**" steht, wird im Verilog-Code zwischen "**begin**" und "**end**" geschrieben. Die Ausführung des Codes erfolgt sequentiell, nämlich in derselben Reihenfolge, in der sie geschrieben sind. Allerdings unterstützt Verilog zusätzlich die Ausführung von zeitlich parallelen Anweisungen (nicht blockierende prozedurale Zuweisung).

Nebenläufigkeit bzw. Parallelausführung der Anweisungen ist die spezielle Eigenschaft der Verilog-Programmierung, die sie von herkömmlichen Programmiersprachen unterscheidet. Während bei herkömmlichen Programmiersprachen alle Anweisungen nur sequentiell, nämlich nacheinander, ausgeführt werden müssen, gibt es bei der Verilog-Programmierung durch Zuweisungsoperatoren "<=" die Möglichkeit, die Anweisungen zeitlich parallel ausführen zu lassen.

Neben den nicht-blockierenden Zuweisungen, die in einem always-Block zur Parallelisierung von Prozessen eingesetzt werden und die eine grundlegende Bedeutung für die Logiksynthese haben, gibt es auch andere Konstrukte, die man für die Parallelisierung von Prozessen anwenden kann, nämlich **fork** (Gabelung) und **join** (Vereinigung). Diese bewirken, dass zwei oder mehr sequentielle Anweisungen parallel abgearbeitet werden, mit anderen Worten bieten sie die Möglichkeit, nebenläufige Prozesse/Threads parallel gleichzeitig auszuführen.

Beispiel für nicht-blockierende Zuweisung:

Schieberegister:

```
module SchiebeReg (x, y, clk)        // nicht-blockierende Zuweisung
    input [3:0] x;
    input  clk;
    output [3:0] y;
    reg [3:0] a1, a2, a3;
    always @(posedge clk)
    begin
        a1 <= x;
        a2 <= a1;
        a3 <= a2;
        y <= a3;                     // y bekommt den Wert von a3!
    end
endmodule
```

Beispiel für Zuweisung mit Blocking:

```
module BlockReg (x, y, clk)          // Blocking-Zuweisung
    input [3:0] x;
    input  clk;
    output [3:0] y;
    reg [3:0] a1, a2, a3;
    always @(posedge clk)
    begin
        a1 = x;
        a2 = a1;
        a3 = a2;
        y = a3;                      // y bekommt den Wert von x!
    end
endmodule
```

Zwischen den Blöcken dürfen entweder blockierende Zuweisungen "**=**" oder nicht-blockierende Zuweisungen "**<=**" stehen, in die nur Register-Typen "**reg**" zugewiesen werden. Genauer gesagt, die Wert-Zuweisung von "**reg**"-Typen kann nur innerhalb von Prozeduralblöcken stattfinden. Es sei darauf hingewiesen, dass die Verilog-Syntax keine dauerhaften Zuweisungen (assignments) von "**net**"-Typen innerhalb der Prozeduralblöcke erlaubt.

Prozeduralblöcke kann man in zwei unterschiedliche Bereiche einordnen:

1. "**initial**"-Prozeduralblöcke
2. "**always**"-Prozeduralblöcke

   wobei "**always**" wiederum in zwei Blockarten unterteilt wird:

   a. kombinatorische Logik (Schaltnetz)

   b. sequentielle Logik (Schaltwerk)

"**initial**"-Blöcke werden nur einmal am Anfang der Simulation ausgeführt. Damit bekommen Variablen die Anfangswerte zugewiesen und führen sie seqentiell aus. Der Prozess ist beendet, wenn die letzte Anweisung bearbeitet ist. Sie wird hauptsächlich in der Simulation für Testbench angewendet.

Beispiel:

```
module Zähler-initialisieren (y, clk, res);
    input res, clk;
        output y;
    reg [2:0] y;
    initial
        begin
            y = 1;
            clk =0;
        end
    always clk = #10 ~clk; // wechselt alle 10 Zeiteinheiten die Pegel
    always @ ( posedge clk)
        begin
            if(res)
            y <=0;
            else y <= y+1;
        end
endmodule
```

Der "**always**"-Block wird direkt am Anfang des Programmcodes ausgeführt. Die Verilog-Programmiersprache bietet die Möglichkeit, mehrere Blöcke in einem Code anzuwenden. In diesem Fall werden alle Blöcke parallel laufen. "**always**"-Blöcke sind eine endlose Schleife und terminieren während der ganzen Laufzeit nicht.

Wie schon vorher erwähnt, kann in "**always**"-Blöcken nur die Zuweisung von "**reg**"-Typen stattfinden. Das basiert darauf, dass die Änderung abhängig von der Sensitivitätsliste, die aus Ereigniszeit bzw. Ereignisvariable besteht, übernommen wird. Solange die Parameter der Sensitivitätsliste keine Signaländerung bekommen, erfolgt auch keine Änderung im Programmablauf. Deshalb muss der aktuelle Wert bis zur nächsten Aktivierung gespeichert und wieder angewendet werden. Um dies zu realisieren, müssen die Variablen daher als "**reg**"-Typen deklariert sein.

"**always**"-Blöcke kann man als kombinatorische oder auch sequentielle Logik programmieren.

- Kombinatorische Logik (Schaltnetz), wenn deren Werte an den Ausgängen zu irgendeinem Zeitpunkt nur von den Werten der Eingänge zu diesem Zeitpunkt abhängen. Nach Empfehlung erfahrener Programmierer, ist es ratsam, die blockierende Zuweisung "=" nur in kombinatorischer Logik einzusetzen.
- Sequentielle Logik (Schaltwerk), wenn die Ausführung erst bei positivem bzw. negativem Takt stattfinden soll.

Ein "**always**"-Block sieht wie folgt aus:

```
always @ (Ereignis_Ausdrücke liste)
    begin
        Anweisungen_1;          // sequentielle Ausführung
        ...
        Anweisungen_n;
    end
```

4.6.2 Strukturmodell

- Das Strukturmodell besteht aus vordefinierten Modulen, entweder Primitiven-/ oder User-definerten Modulen (siehe oben, Verilog-Primitive), die eine hierarchische Struktur herbeiführen.
- Zum Einbauen von Primitiven werden Module nicht eingefügt, sondern es wird nur eine Instanz davon erstellt. Jede Instanz bekommt einen identifizierbaren Namen, um beim mehrmaligen Aufruf desselben Moduls in einer hohen Ebene eines hierarchischen Moduls die Arbeit zu erleichtern und Fehler zu vermeiden.

Beispiel:

```
module copy ( a, b, c)
    input a, b;
    output c;
    and UND1( c, a, b);
endmodule
```

4.6.3 Physikalisches Modell

beschäftigt sich mit der Darstellung des geometrischen Layouts. Dabei wird die fertige Gatternetzliste anhand von Synthesewerkzeugen (Layoutsynthese) in einem physikalischen Modell dargestellt.

4.7 Verifikation

Die Verifikation in Verilog übernimmt die Überprüfungsaufgabe. Damit wird die Funktionsfähigkeit des Hardwarebeschreibungs-Designs kontrolliert. Gleichzeitig wird überprüft, dass die Ausführungszeit die zeitlichen Randbedingungen nicht überschreitet. Verifikationsmodule kann man nicht für die Anwendung im Syntheseprozess vorgesehen. Sie sind z. B. dazu gedacht, um eine Datei aufzurufen, zu verarbeiten und auf dem Display auszugeben.

Testbench ist Teil der Verifikation; es ist ein einfaches Verilogprogramm, das den Code des Hauptprogramms aufruft, die Eingänge mit Anfangswerten initialisiert und das Ausgangssignal nach der Simulation ausgibt bzw. dokumentiert oder speichert. Durch ein Signalanalyseprogramm kann man den Signalverlauf verfolgen und überprüfen, ob mit dem eingegebenen Eingangssignal das gewünschte Ausgangssignal erreicht wird.

Ein Testbench-Modul kann in allen höheren Abstraktionsebenen eingesetzt werden, sowohl vor als auch nach der Logiksynthese.

Heutzutage ist es bei komplexen digitalen Entwürfen, wie bei Mikroprozessoren mit Hunderten von Signalen, fast unmöglich, jedes einzelne Eingangssignal einzeln auf Korrektheit zu testen.

Während für die Synthese nicht alle Sprachkonstrukte angewendet werden können, ist die Anwendung aller Verilog-Sprachkonstrukte bei der Verifikation erlaubt, und es ist sogar sinnvoll, sie einzusetzen, wie z. B. das Verzögerungskonstrukt "**#**" (delay), "**initial**", "**$finish**" etc.

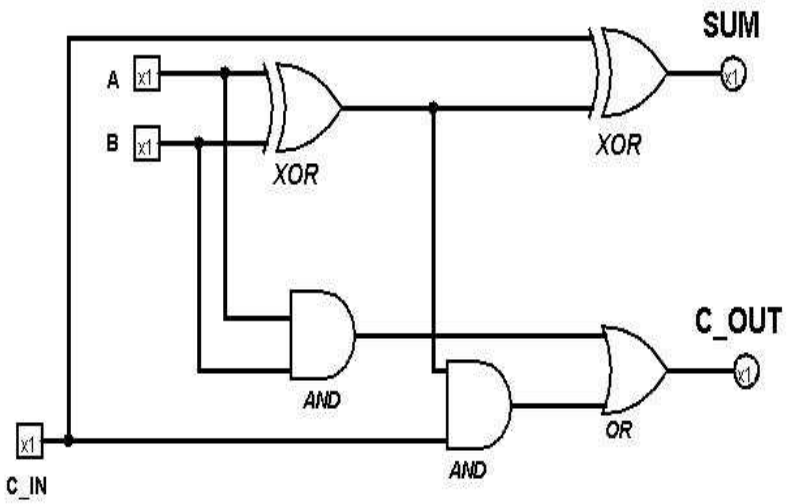
4.8 Compiler-Anweisung (Compiler Directive)

Solche Direktiven steuern den Compiler-Vorgang und wirken sogar über File-Grenzen hinweg, bis sie durch andere Anweisungen überschrieben werden. Die Anweisungen beginnen immer mit dem `-Symbol.

Einige wichtige Compiler-Direktiven sind: **`define**, **`include**, **`ifdef**, **`else**, **`endif**, **`timescale**, **`reset_all**.

Abschließend folgt nun ein Beispiel für ein Verilog-Programm:

```
module Full_ADD (SUM, C_OUT, A, B, C_IN);
        input A, B, C_IN;
        output SUM, C_OUT;
                always @ (A, B, C_IN)
                        begin
                                SUM = A ^ B ^ C_IN;
                                C_OUT = (A & B) | (B & C_IN) | (A & C_IN);
                        end
endmodule
```



Glossar

Prozedur, die; *Subst.* (procedure)

In einem Programm eine benannte Anweisungsfolge - meist mit zugehörigen Konstanten, Datentypen und Variablen - zur Ausführung einer bestimmten Aufgabe. Eine Prozedur lässt sich in der Regel sowohl durch andere Prozeduren als auch durch das Hauptprogramm aufrufen (ausführen). Einige Sprachen unterscheiden zwischen einer Prozedur und einer Funktion, wobei letztere einen Wert zurückgibt. [1]

prozedurale Sprache, die; *Subst.* (procedural language)

Eine Programmiersprache, in der die Prozedur das grundlegende Programmelement darstellt. Unter Prozedur ist in diesem Sinne eine benannte Folge von Anweisungen, z. B. eine Routine, ein Unterprogramm oder eine Funktion zu verstehen. Die allgemein verwendeten Hochsprachen (C, Pascal, Basic, FORTRAN, COBOL, Ada) sind durchgängig prozedurale Sprachen. [1]

Synthese

Unter einem Syntheseprozess versteht man die automatische Transformation einer Beschreibung von einer Abstraktionsebene in die nächsttieferliegende, wobei tieferliegend technologienäher beutet. [2]

verifizieren

1. *(bildungsspr.)* durch Überprüfen die Richtigkeit einer Sache bestätigen [5]

Literaturverzeichnis

[1] Microsoft Press Computer-Lexikon mit Fachwörterbuch (deutsch-englisch/ englisch-deutsch) Ausgabe 2001

[2] "Verilog Modellbildung für Synthese und Verifikation" von Bernhard Hoppe Oldenbourg Verlag 2006.

[3] Einführung in Verilog HDL von Prof. Dr. Bernhard Hoppe, Karthikeyan Balasubramanian, M.Sc.

[4] www.mikrocontroller.net

[5] Duden - Deutsches Universalwörterbuch, 6. Aufl. Mannheim 2006 [CD-ROM].

[6] Verilog Prof. I. Sengupta