Steilkurs Verilog

1) Dient zur Simulation und zur Synthese
   → nicht alles, was man hinschreiben darf, kann auch synthetisiert werden!
   → Man beschränkt sich auf eine synthetisierbare Teilmenge von Verilog (Ausnahme: Testbench)

2) Strukturbeschreibung vs. Verhaltensbeschreibung

Struktur (Instanziierung von Modulen):

```
and g1(c, a, b);    "and": Modulname, "g1" = Instanzname
```

Verhalten (Beschreibung von Berechnungen):

```
assign c = a & b;
```

Üblich ist eine Mischung aus beiden Beschreibungsarten!

3) Es gibt zwei fundamentale Datentypen in Verilog:

   a) Leitungen ("wire"), Bsp.: `wire a;`
      transportieren einen Wert, speichern aber nichts

   b) Register ("reg"), Bsp.: `reg b;`
      speichern einen Wert

Bei beiden kann ein geordnetes Bündel durch einen Vektor beschrieben werden, z.B. `wire [31:0] data;`

Bem.: Die Indexgrenzen können auch anders herum aufgeschrieben werden: `wire [0:31] data;`
Es empfiehlt sich, eine konsistente Schreibweise zu benutzen!
Wir: [x:y] ≙ Index des MSB = x, Index des LSB = y.

Man kann einen Teilvektor aus einem Vektor auswählen: (2)

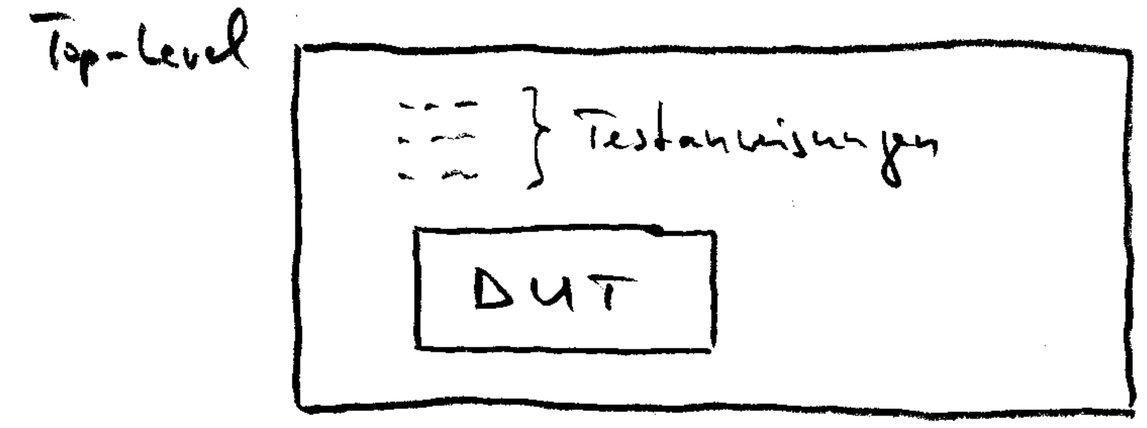
```
wire [31:0] full;
wire [15:0] part;
assign part[15:0] = full[31:16];
```

Man kann Teilvektoren zu einem Vektor zusammenfügen:

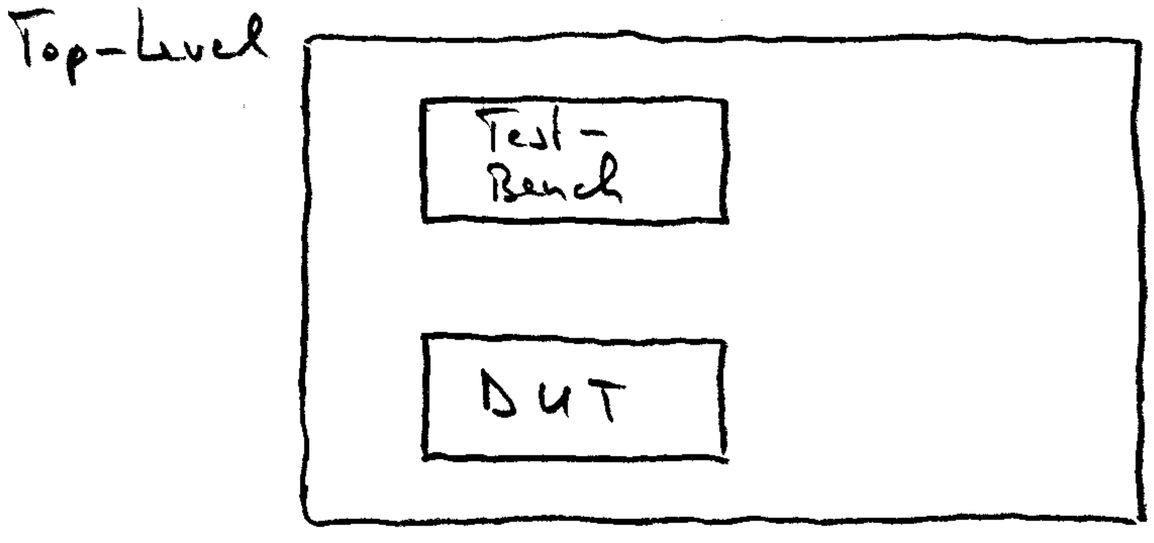
```
wire [15:0] lo;
wire [15:0] hi;
wire [31:0] full;
assign full[31:0] = { hi[15:0], lo[15:0] };
```

Wichtig: Links und rechts der Zuweisung müssen gleich viele Leitungen stehen!

4) "Programmieren" in Verilog ≙ Schreiben von Modulen, die instanziiert werden können. Innerhalb von Modulen können andere Module instanziiert werden. Genau ein Modul hat keine Ein- und Ausgänge und wird implizit genau einmal instanziiert ("Top-Level-Modul"). Das ist häufig die Testbench für die eigentliche Schaltung (oder instanziiert die Testbench und die Schaltung):



DUT:
"Device under test"



Beim Schalten eines Moduls werden seine Ein- und Ausgänge festgelegt. Bsp.: 7-Segment-Dekoder

```
module ssd ( binval, segments);
    input [3:0] binval;
    output [6:0] segments;
      ⋮
endmodule
```

Bem.: Angenommen, ein Ausgang wird von einem Register gespeist:

```
module a(..., out, ...);
    output out;
      ⋮
    reg x;
      ⋮
    assign out = x;
      ⋮
endmodule
```

Dann kann man das zusätzliche Register x durch einen "Ausgang mit Register" einsparen:

```
module a(..., out, ...);
    output reg out;
      ⋮
endmodule
```

5) Konstanten werden dezimal, binär, oder hexadezimal mit Angabe der Bitbreite (!) notiert:

32'hF7129ABC

2'b01

5'd13

Sie können symbolische Namen bekommen:

`define MAGIC 6'd42

Benutzung: `MAGIC

6) Operatoren:

`==`, `!=` } mit der üblichen Bedeutung, nicht aufwendig

`<`, `<=`, `>`, `>=` } mit der üblichen Bedeutung, unsigned, aufwendig, besser vermeiden!

`~`, `&`, `|`, `^` : wie üblich

`+`, `-` : wie üblich, erzeugt normalerweise schnelle Schaltungen

`*`, `/` : kann nicht synthetisiert werden, nicht benutzen!

Für Vektoren gibt's die "Reduktionsoperatoren", die eine Verknüpfung quer über die Bits des Vektors anwenden. Bsp.: Ergebnis soll 1 sein, genau dann, wenn alle Bits von register[4:0] Null sind:

`~|register[4:0]`

7) Beim Instanziieren von Modulen gibt es zwei Möglichkeiten der Zuordnung Argument ⟷ Parameter:

a) durch die Position (wie bei Funktionen in C)

b) durch den Parameternamen

Bsp.:

```
module cpu (clk, rst, ...)

a) cpu cpu1(c, r, ...);

b) cpu cpu1(
       .clk(c),
       .rst(r),
       :
   );
```

⟵ Reihenfolge ist egal!

Bei vielen Parametern (das ist i.d.R. der Fall) ist b) besser!

8) Sie müssen sich jederzeit genau im Klaren sein, ob Sie gerade ein Stück kombinatorische Logik oder ein Schaltwerk schreiben wollen!!

9) Schaltwerke:

ACHTUNG: Wir bauen nur synchrone Schaltwerke !!!

Also schalten alle Flip-Flops nur auf der positiven Flanke der Clock! Es gibt nur eine einzige Clock, und alle Flip-Flops schalten mit dieser Clock! Es befindet sich niemals irgend ein Gatter in der Clockleitung!!
Formulierung in Verilog:

```
always @(posedge clk) begin
    out <= in;
end
```

Benutzen Sie in Flip-Flops nur die nicht-blockierende Zuweisung "<=". Alle Zuweisungen im begin-end-block erfolgen gleichzeitig: zuerst werden alle rechten Seiten berechnet und dann alle gleichzeitig den linken Seiten (das müssen Register sein!) zugewiesen. Achtung: alle always-Blöcke werden gleichzeitig ausgewertet! Deshalb sind folgende zwei Formulierungen gleichwertig:

a)
```
always @(posedge clk) begin
    s <= i;
end
always @(posedge clk) begin
    t <= s;
end
```

b)
```
always @(posedge clk) begin
    s <= i;
    t <= s;
end
```

↔ Beachte: Reihenfolge ist egal!!

Wenn man einen Transfer nur unter einer Bedingung stattfinden lassen möchte, gelt das so:

```
always @(posedge clk) begin
    if (bedingung) begin
        out <= in;
    end
end
```

ACHTUNG: Niemals, unter keiner Kombination von Bedingungen, darf dasselbe Ziel von mehr als einer Zuweisung beschrieben werden!!

Bsp.: Wenn Sie schreiben

```
always @(posedge clk) begin
    if (bedingung1) out <= in1;
    if (bedingung2) out <= in2;
end
```

dann müssen Sie beweisen, daß bedingung1 $\Rightarrow \neg$ bedingung2 gilt!! (Btw, das reicht, denn dann gilt auch bedingung2 $\Rightarrow \neg$ bedingung1)
Einfacher ist es, "else" zu benutzen!

ACHTUNG: Alle Automaten müssen in einen definierten Anfangszustand gelangen!! (Grund: one-hot-Kodierung der Zustände)
Wir benutzen nur einen synchronen Reset!

```
always @(posedge clk) begin
    if (rst) begin
        state <= 12'h0000;
    end else begin
        ⋮
    end
end
```

10) Kombinatorische Schaltungen

a) assign x = a & b; → UND-Gatter

assign x = c ? a : b; → MUX

b) always @(*) begin

:

end

↖ "Sensitivity List": alle Signale, bei deren Änderung der always-Block neu berechnet werden soll. Schaltnetz: alle Signale auf allen rechten Seiten von Zuweisungen ("*").

Häufigste Anwendung:

```
always @ (*) begin
    case (state)
        3'b000:
            begin
                out = in1 & in2;
            end
        3'b001:
            :
        default:
            begin
                out = 32'h0;
            end
    endcase
end
```

ACHTUNG:

- nur die blockierende Zuweisung ("=") verwenden!
- immer einen default-Fall einbauen!
- in jedem case-Fall immer alle Ausgangssignale setzen! (Sonst werden Latches synthetisiert!!)

- die linken Seiten der Zuweisungen müssen formal reg-Variablen sein! ABER: Es werden dafür keine Flip-Flops synthetisiert!

Im Zweifelsfall sollte man sich die generierte Schaltung ansehen!

11) Endliche Automaten (FSMs)

Viele verschiedene Möglichkeiten zur Formulierung!

Häufig: Moore-Automat (Ausgabe hängt nur vom Zustand ab)

```
always @(posedge clk) begin
    if (rst) begin
        state <= 0;
    end else begin
        state <= next_state;
    end
end

always @(*) begin
    case (state)
        `STATE 0 :
            begin
                out = ...
                next_state = ...
            end
        `STATE 1 :
            :
        default :
            :
    endcase
end
```