Einführung in Verilog HDL

Hardware Hacken ohne Lötkolben

Clifford Wolf

ROCK Linux - http://www.rocklinux.org/

Csync2 - http://oss.linbit.com/csync2/

STFL - http://www.clifford.at/stfl/

SPL - http://www.clifford.at/spl/

Einleitung

DISCLAIMER

- Dieser Vortrag deckt naturgemäss nur einen kleinen Teil des Verilog HDL Sprachumfanges ab.
- Die einzelnen Verilog HDL Implementierungen unterscheiden sich zum Teil gravierend darin, wie weit aktuellere Verilog HDL Standards (2001 und 2005) unterstützt werden.
- HDL Design ist nicht Programmieren! Neben der Sprache müssen auch Konzepte erlernt werden. Und dazu ist Übung notwendig..
- Viele der Codebeispiele in den Folien sind ungetestet. Korrekturhinweise bitte an `<clifford@clifford.at>`.

Was sind HDLs?

- HDL = Hardware Description Language

- Computersprachen für die Beschreibung von Schaltungen
- .. anstelle von grafischen Schaltplänen.

- HDLs erfüllen mehrere Zwecke:
  - Dokumentation von Schaltungen
  - Simulation, Test und Verifizierung
  - FPGA und ASIC Schaltungssynthese

- HDLs sind deklarativ statt prozedural!

- Codeteile für Simulation und Test können aber wieder prozedural sein
- ..was dann vor allem für Neulinge besonders verwirrend ist.

Verilog HDL - Hintergrund

- 1983: Projektstart bei Gateway Design Automation.
- 1985: Markteinführung als reiner Simulator.
- 1989: Übernahme durch Cadence Design Systems.
- 1990: OVI (Open Verilog International) wird gegründet.
- 1992: OVI initiert den IEEE Standardisierungsprozess.
- 1995: Verilog is IEEE Standard 1364-1995.
- 2001: Deutliche Verbesserungen in IEEE 1364-2001.
- 2005: Einige weiteren Verbesserungen in IEEE 1364-2005.

- SystemVerilog (IEEE 1800-2005) ist ein nochmal sehr viel mächtigeres Superset von Verilog 2005.

- Verilog-AMS sind Verilog erweiterungen für Analoge und Mixed-Signal Schaltungen.

Verilog HDL - Konzepte (1/4)

- Verilog HDL erlaubt es sehr low-level zu entwickeln ohne sich dabei in Details verlieren zu müssen.
- In dieser Hinsicht ist Verilog HDL vielleicht der Programmiersprache C sehr ähnlich.
- Damit das gut funktioniert ist aber auch etwas Erfahrung notwendig.

- Mit Verilog kann man (digitale) Schaltungen entwerfen.
- Den Schritt, aus Verilog Code die Schaltungen (Netzlisten) zu generieren nennt man **Synthese**.

- Mit Verilog ist es sehr einfach extrem umfangreiche Schaltungen zu entwerfen.

- Dabei sollte man immer aufpassen nicht zu verschwenderisch mit den Resourcen umzugehen.

Verilog HDL - Konzepte (2/4)

- In Verilog HDL kann man sowohl die eigentlichen Schaltungen als auch Testbenches für Simulation schreiben.
- Für die Testbenches gibt es auch einige prozedurale Konstrukte.
- Damit wird zwischen **synthesefähigem** und **nicht synthesefähigem** Verilog Code unterschieden.

- Es gibt auch Sprachkonstrukte die bei der Synthese einfach ignoriert werden.
- Beispiele dafür: Delays sowie Undefined Values.

- Für Simulationen kann Verilog mit der VPI C-API (älter: PLI C-API) erweitert werden.
- Damit lassen sich Verilog Simulationsmodelle leicht z.Bsp. an umfangreiche Architekturmodelle anbinden.

Verilog HDL - Konzepte (3/4)

- Verilog Designs sind hierarchisch in Module aufgeteilt.
- Die atomaren Bausteine dieser Hierarchien (die sog. **Primitives**) sind Gatter, Switches und Speicherzellen.
- Für die Simulation werden die Primitives über Wahrheitstabellen abgebildet.
- Es ist auch möglich eigene Primitives (UDP, **User Defined Primitives**) zu erstellen.
- Den Prozess, Verilog Designs aus Primitives aufzubauen nennt man Gate-Level Modeling.
- Verilog Designs die nur aus Primitives und ihren Verbindungen bestehen nennt man auch **Verilog Netzlisten**.

Verilog HDL - Konzepte (4/4)

- Es ist auch möglich Verknüpfungen zwischen Signalen mit Operatoren zu codieren.
- Diesen Ansatz nennt man **Dataflow Modeling**.

- Und ist es möglich eine Schaltung über ihr Verhalten zu codieren.
- Diesen Ansatz nennt man **Behavioral Modeling**.

- Diese Verfahren sehen rein optisch fast schon wie Programmieren aus.
- Es ist immer wichtig im Hinterkopf zu behalten, dass es aber genau das nicht ist!

Grundlagen und Gate-Level Modeling

Signale

- Signale sind die Verilog-Repräsentation von Leiterbahnen.
- Jedes Signal führt zu jedem Zeitpunkt genau einen Wert (normalerweise `0` oder `1`).
- Ähnlich den Variablen in vielen Programmiersprachen müssen Signale deklariert werden bevor sie verwendet werden können.
- Signale können mit dem `wire` keyword deklariert werden:

```
wire signal1, signal2, signal3;
```

Vektoren

- Vektoren (Busse) sind die Zusammenfassung mehrerer Signale unter einem Namen.
- Die einzelnen Signale eines Vektors sind durchnummeriert.
- Bei der Deklaration eines Vektors muss der Indexbereich der Signale **vor** dem Vektornamen in eckigen Klammern angegeben werden:
  ```
  wire [31:0] bus_of_32_signals;
  ```
- Bei der Verwendung kann man auf Teile des Vektors oder einzelne Signale mit eckigen Klammern **nach** dem Vektornamen zugreifen:
  ```
  bus_of_32_signals[15:0]
  bus_of_32_signals[31]
  ```

Werte und Konstanten

- Signale können folgende Werte annehmen:
  - `0` - Eine logische 0
  - `1` - Eine logische 1
  - `x` - Ein undefinierter Wert (Kollision bzw. Uninitialisiert)
  - `z` - Hochohmig
- Konstanten werden gemäss folgender Syntax notiert: `<bits>'<basis><werte>`
- `<basis>` kann `b` (binär), `d` (dezimal), `o` (oktal) oder `h` (hexadezimal) sein.
- Underscores in Konstanten werden ignoriert.
- Einfache Dezimalzahlen sind 32bit Werte.
- Beispiele: `4b'1010` `12h'abc` `16'bz` `'o777` `42`

Modul Deklaration

- Ein Modul wir eingeleitet mit:
  `module` *modulname*`(`*portliste*`);`
- Und abgeschlossen mit:
  `endmodule`
- Die Portliste ist eine mit Kommata getrennte Liste jener Signale die nach aussen hin sichtbar sein sollen.
- Diese Signale müssen im Modulkörper mit `input` bzw. `output` statt `wire` deklariert werden.

Instanzierungen

- Module können Instanzen anderer Module sowie Primitives beinhalten.
- Die Instanzierung erfolgt mit folgender Syntax:
  `<modul-name> <instanz-name>(<signal-liste>);`
- Dabei beinhaltet die `<signal-liste>` jene Signale mit denen die Ports des Moduls verbunden werden sollen.
- Beispiel:

```
xor my_xor_gate(out, in1, in2);
```

Ports by Name

- Bei der Instanzierung komplexer Module mit vielen Ports ist es besser die Ports lokalen Signalen über die Portnamen als über die Portreihenfolge zuzuweisen.

- Das hilft Flüchtigkeitsfehlern vorzubeugen.

```
RAMB16_S4_S4 dual_port_mem_cell (
  .DIA(wr_data),    // Port A 4-bit Data Input
  .WEA(wr_en),      // Port A Write Enable Input
  .ADDRA(wr_addr),  // Port A 12-bit Address Input
  .CLKA(wr_clk),    // Port A Clock
  .DOB(rd_data),    // Port B 4-bit Data Output
  .ADDRB(rd_addr),  // Port B 12-bit Address Input
  .CLKB(rd_clk)     // Port B Clock
);
```

Dataflow Modeling

Assign Statement

- Im Dataflow Modeling kann man Ausdrücke (Terme) bilden indem man Signale mit Operatoren verknüpft.
- Damit kann man z.Bsp. Rechenwerke extrem schnell aufbauen.
- Man sollte dabei aber nie vergessen, dass jeder verwendete Operator einen weiteren Hardwareaufwand darstellt.

- Das ergebnis eines Ausdrucks kann mit dem `assign` statement einem neuen Signal zugewiesen werden.

```
module add2(sum, a, b);
   output  [2:0]   sum;
   input   [1:0]   a, b;
   assign sum = a + b;
endmodule
```

Operatoren

- Bitweise (infix): `& | ^ ~^`
- Bitweise (prefix): `~`
- Logisch (infix): `&& ||`
- Logisch (prefix): `!`
- Reduktion (prefix): `& ~& | ~| ^ ~^`
- Arithmetrisch (infix): `+ - * / % **`
- Arithmetrisch (prefix): `+ -`
- Relational (infix): `> < >= <= == != === !==`
- Shift (infix): `<< >> <<< >>>`
- Zusammenhängen: `{a, b, c}`
- Wiederholung: `{4{a}}`
- Fallunterscheidung: `q ? a1 : a2`
- Gruppierung: `(...)`

Behavioral Modeling

Einleitung

- Im Behavioral Modeling wird das Verhalten einer Schaltung beschrieben.
- Die Syntheseprodukte solcher Schaltungen vorherzusagen ist nicht immer ganz einfach.
- Daher sollte man soweit es geht hier besonders darauf achten, die Syntheseprodukte händisch zu kontrollieren.
- Behavioral Modeling eignet sich vor allem für synchrone (aka. “getacktete”) Logik.
- Also Logik mit Registern an den Ausgängen.

Register und Always Blöcke

- Behavioral Modeling geschieht immer in `always` blöcken.
- Alle von einem `always` block getriebenen Signale müssen mit `reg` statt `wire` deklariert werden.
- Bei synchroner Logik wird das Synchronisierungsevent mit `@(<event>)` angegeben, ansonsten `@(*)` verwenden.
- Bei mehr als einem Statement im müssen die Keywords `begin` und `end` benutzt werden.

```
always @(posedge clock or posedge reset) begin
   if (reset)
      counter <= 0;
   else
      counter <= counter + 1;
end
```

If und Case

- Für Fallunterscheidungen gibt es ein `if-then-else` Konstrukt sowie `case`, `casex` und `casez` Konstrukte.
- Bei `casez` wird `z` und bei `casex` werden `x` und `z` als don't-care Werte behandelt.
- If und Case Konstrukte werden zu Multiplexern, (Priority-)Dekodern und/oder FF-Enable-Signale synthetisiert.

```
if (reset) counter <= 0;
else counter <= count + 1;

casex (ins)
  8'b1xxx_xxxx: reg <= {1'b0, ins[6:0]};
  8'b01xx_xxxx: reg <= reg + {2'b00, ins[5:0]};
  8'b00xx_xxxx: reg <= reg - {2'b00, ins[5:0]};
endcase
```

Non-Blocking und Blocking Assignments

- Bei non-blocking assignments (<=) gilt der neue Wert erst nach Durchlauf des `always` blocks.

- Bei blocking assignments (=) liegt der neue Wert sofort am Signalnamen an.

```
always @(clock) begin
   flip <= flop;
   flop <= flip;
end

always @(*) begin
   addr = base_addr;
   if (use_offset)
      addr = addr + offset;
end
```

Vermeiden von Hardware Registern

- Nicht alle Signale die von `always` Blöcken getrieben werden müssen als Register ausgeführt werden.
- Oft wird das Synthesetool aber dazu gezwungen ein Register zu erzeugen weil das Signal nicht in allen Pfaden getrieben wird.
- Eine einfache Möglichkeit sich vor Programmierfehlern zu schützen ist es default-Werte zu verwenden.

```
always @(*) begin
   result <= 1'bx;
   if (force_result_to_1)
      result <= 1'b1;
   else
   if (force_result_to_0)
      result <= 1'b0;
end
```

Weiterführende Konzepte

Arrays

- Adressierbarer Speicher kann mit Arrays modelliert werden.
- Bei der deklaration von Arrays wird die Anzahl der Zellen nach dem Signalnamen geschrieben.

```
module mymem(clock, we, addr, in_data, out_data);
   input clock, we;
   input [7:0] addr, in_data;
   output reg [7:0] out_data;
   reg [7:0] memory [0:255];
   always @(posedge clock) begin
      if (we)
         memory[addr] <= in_data;
      else
         out_data <= memory[addr];
   end
endmodule
```

Tristate-Netze

- Ein Signal kann mehrere Treiber haben.
- Ein Signal mit voneinander abweichenden nicht-`z` Treibern nimmt den Wert `x` an.
- Tristate Ports müssen als `inout` deklariert.
- Pulldown und Pullup-Netze können mit `tri0` und `tri1` (statt `wire`) modeliert werden.
- Wired AND Netze können mit `wand` bzw. `triand`, wired OR Netze mit `wor` bzw. `trior` modeliert werden.
- Netze mit kapazitativen “memory” können mit `trireg` modeliert werden.

Parameter

- Module können parametrisiert werden.
- Parameter werden mit `parameter` deklariert.
- Eine Möglichkeit Parameter zu ändern ist: `#(...)`
- Mit `localparam` können “lokale Parameter” deklariert werden, die nicht direkt umparametrisiert werden können.

```
module stepper(clock, reset, value);
   input clock, reset;
   output reg [7:0] value;
   parameter resetval = 0;
   parameter stepsize = 1;
   always @(posedge clock, posedge reset)
      value <= reset ? resetval : value + stepsize;
endmodule

stepper #(.resetval(42), .stepsize(7))
   mystepper (clock, reset, stepval);
```

Funktionen

- Häufig benötigte Logikblöcke können in Funktionen ausgelagert werden.
- Funktionen werden mit `function` deklariert.
- Funktionen können nur rein kombinatorische Schaltungen abbilden. Speicherelemente und non-blocking assignments sind nicht erlaubt. Auch keine delays.

```
function [15:0] inverter;
   input [15:0] word;
   begin
      inverter = ~word;
   end
endfunction

assign x = inverter(y);
```

Tasks

- Neben Funktionen gibt es auch Tasks.
- Diese haben keinen Rückgabewert.
- Dafür dürfen sie auch `output` sowie `inout` Ports haben.
- Und in Tasks sind auch Delays erlaubt.

```
task inverter;
   output [15:0] result;
   input [15:0] word;
   begin
      result = ~word;
   end
endtask

inverter(x, y);
```

Generate-Blöcke

- Mit `generate` Blöcken können Logikblöcke leicht aus Schleifen und Ähnlichem heraus erzeugt werden.
- Dazu notwendige Variablen (Zähler, etc.) müssen mit `genvar` deklariert werden.

```
module foobarchain(input in, output out);
   parameter chainlen = 8;
   wire [chainlen:0] interconn;
   genvar i;
   generate
      for (i=0; i<chainlen; i = i+1) begin:B1
         foobar chainelement(.in(interconn[i]),
                             .out(interconn[i+1]));
      end
   endgenerate
   assign interconn[0] = in;
   assign out = interconn[chainlen];
endmodule
```

Compiler Direktiven

- Compiler Direktiven beginnen immer mit einem `‘` Zeichen.
- Und werden mit dem Zeilenende abgeschlossen.

- Einige Beispiele für Compiler Direktiven:

- Defines können mit `‘define <name> <value>` erzeugt und mit `‘<name>` verwendet werden.

- Defines können mit Konstrukten wie `‘ifdef` abgefragt werden.

- Andere Dateien können mit `‘include` eingebunden werden.

Example: Simplified I2C

Example: Calculator / ALU

Simulation und Test

Integer, Real und Time Register Data Types

- Primär für die Simulation gibt es neben den Signal Registern (`reg`) auch weitere Register Datentypen: `integer`, `real` und `time`

- `time` ist mindestens 64 bit breit.

- Mit `$time` kann ermittelt werden, wie viele Zeiteinheiten seit dem Simulationsstart abgelaufen sind.

Prozedurale Programmierung

- Für die Simulation gibt es prozedurale Programmkonstrukte wie z.Bsp. `while` und `for` Schleifen.
- Neben `always` Blöcken gibt es auch `initial` Blöcke, die nur einmal bei Simulationsstart anlaufen.
- Mit z.Bsp. `#100;` kann man in `always` und `initial` Blöcken 100 Zeiteinheiten warten.

System Tasks

- System Tasks erlauben während der Simulation die Kommunikation mit der Aussenwelt.
- Die Namen aller System Taks beginnen mit einem `$`.
- Mit `$finish` kann die Simulation beendet werden.
- Mit `$display` kann eine Textzeile ausgegeben werden.
- C printf-like Format-Strings sind erlaubt.
- `$monitor` ist wie `$display`, erzeugt aber automatisch eine neue Ausgabezeile wenn einer der Werte sich ändert, mit Ausnahme von `$time`.

```
initial
   $monitor("%t %b", $time, testee.state);
```

VCD Dateien

- Die Werte von Signalen können über den Verlauf einer Simulation hinweg aufgezeichnet werden.
- Dafür gibt es ein standardisiertes Dateiformat: VCD (Value Change Dump)
- Das erzeugen von VCD Dateien wird über System Tasks gesteuert.

```
initial
    $dumpfile("mydumpdata.vcd");
    $dumpvars(0, testee);
```

VPI und PLI

- VPI und PLI sind C-APIs für eigene System Tasks.
- Damit ist es leicht möglich Verilog Simulationen an externe Simulationsumgebungen anzubinden.
- Einige Anwendungsmöglichkeiten sind:
  - GUIs zur einfacheren Interaktion mit der Simulation
  - Datenbanken mit umfangreichen Testdatensätzen
  - Integration mit anderen Simulatoren (z.Bsp. qemu)
- Literatur zu VPI/PLI ist leider extrem teuer.
- Aber die System Task Implementierungen aus den Icarus Verilog Sourcen sind eine gute Referenz.
- Und der IEEE Verilog Standard ist sehr bequem zu lesen.

FPGA Synthese

Constraints

- Für die Synthese brauchen die FPGA Tools zusätzliche Informationen, sogenannte Constraints. Das sind z.Bsp.:
  - Welches Signal liegt auf welchem I/O Pin
  - Wie gehören die I/O Bänke konfiguriert
  - Wie schnell schalten Clocks und andere Signale
- Die exakte Syntax ist herstellerspecifisch.
- Xilinx benutzt externe UCF Dateien sowie Verilog Attribute.
- Dokumentation: Software Manuals - Constraints Guide

```
NET "clk50m"      LOC = "E12"  | IOSTANDARD = LVCMOS33 | PERIOD = 20.000 ;
NET "leds<0>"     LOC = "R20"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
NET "leds<1>"     LOC = "T19"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
NET "leds<2>"     LOC = "U20"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
NET "leds<3>"     LOC = "U19"  | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;
NET "buttons<0>"  LOC = "V8"   | IOSTANDARD = LVCMOS33 ;
NET "buttons<1>"  LOC = "U10"  | IOSTANDARD = LVCMOS33 ;
NET "buttons<2>"  LOC = "U8"   | IOSTANDARD = LVCMOS33 ;
NET "buttons<3>"  LOC = "T9"   | IOSTANDARD = LVCMOS33 ;
```

Design Libraries

- Die Hersteller liefern Bibliotheken mit Modulen für alle möglichen Zwecke.
- Es empfiehlt sich sich mit diesen Bibliotheken vertraut zu machen und sie gegebenenfalls zu Nutzen.
- Besonders wichtig in diesem Zusammenhang sind:
  - RAM und ROM Zellen
  - Komplexe Arithmetik
  - Clock Buffer
- Viele FPGA Entwicklungsumgebungen haben auch Tools zum Generieren speziell angepasster Module.

Referenzen

Literatur

- “Verilog HDL” (Samir Palnitkar)
  ISBN-13: 978-0130449115
- “CMOS VLSI Design” (Neil H.E. Weste)
  `http://www.aw-bc.com/weste/`
  ISBN-13: 978-0321149015
- “Layoutsynthese elektronischer Schaltungen” (Jens Lienig)
  `http://www.ifte.de/lienig/layout/index.html`
  ISBN-13: 978-3540296270
- Verilog HDL auf Wikipedia:
  `http://en.wikipedia.org/wiki/Verilog`
- IEEE 1364-2005 Standard
  `http://ieeexplore.ieee.org/servlet/opac?punumber=10779`

Freie Software

- Icarus Verilog
  `http://www.icarus.com/eda/verilog/`
- GTKWave
  `http://home.nc.rr.com/gtkwave/`
- Qucs
  `http://qucs.sourceforge.net/`

Nicht-Freie Software

- Xilinx ISE (WebPACK)
  `http://www.xilinx.com/ise/`
- Synopsys VCS
  `http://www.synopsys.com/products/simulation/`
- Cadence NC-Verilog
  `http://www.cadence.com/products/functional_ver/nc-verilog/`
- Mentor ModelSim
  `http://www.model.com/`