Vorlesung 4
Verilog Sprache

Verilog - Abstraktionsebenen

- Verilog ist eine Hardware-Beschreibungssprache
- Sie ermöglicht Beschreibung digitaler Schaltungen
- Drei Abstraktionsebenen:
- 1. Behavioural (Beschreibung des Verhaltens mithilfe von high-level Sprachelementen)
  - Beschreibung der Schaltungsfunktion in Form von Algorithmen
  - Befehle werden sequenziell ausgeführt
  - Gut für Modellierung und Simulation
  - nicht hardwarenah und nicht „synthetisierbar“
- 2. Register Transfer Level (RTL) Beschreibung
  - Digitale Schalung besteht in der Regel aus Registern deren Übergänge mit kombinatorischer Logik definiert sind. Hardwarenahe Beschreibung dieser Struktur nennen wir Register Transfer Level (RTL) Beschreibung
  - Moderne Definition: "Any code that is synthesizable is called RTL code“
- 3. Gate Level
  - Eine Beschreibung auf Gatter Ebene ist ebenfalls in Verilog möglich.
  - Beispiel: RTL verwendet if-else…, Gatterebene AND, OR…
  - Es wird nicht im Designprozess verwendet. Gate Level Code wird von synthese-Tools generiert und wird für die Simulation benutzt.
  - Verilog ermöglicht dadurch eine Hardwaresimulation
  - Backend „anntotation“ von Verzögerungen ist möglich

Verilog

- Besonderheiten von Verilog
- Programme in gewöhnlichen Programmiersprachen (z.B. C) enthalten Befehle, die nacheinander (sequenziell) ausgeführt werden
- Verilog ist sowohl eine Programmiersprache als auch eine Schaltungs-Beschreibungssprache
- Die Befehle (Zuweisungen) werden (im Teil der Schaltungen modelliert) gleichzeitig ausgeführt

- Ein Top-Modul enthält oft zwei Teile
- Ein Teil des Codes beschreibt die Schaltung (synthetisierbarer Code).
- Ein Teil enthält sequentielle Befehle und dient zum Testen vom sythetisierbaren Teil

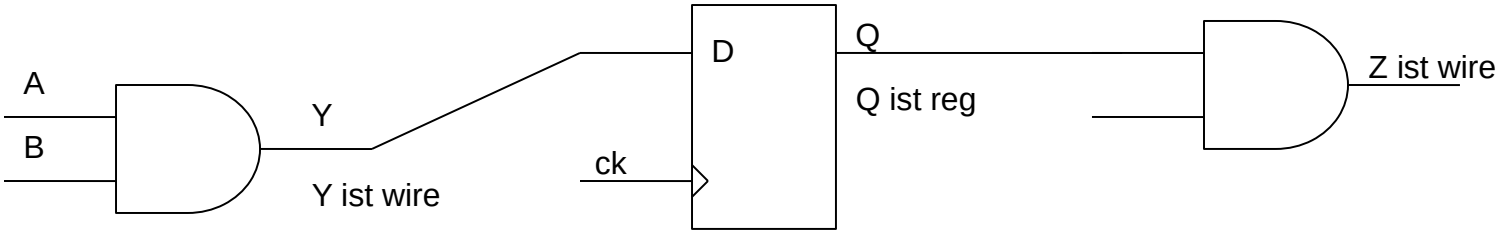
- Verliog ermöglicht Hierarchie zu bilden
- Die wichtigsten Elemente der Hierarchie sind Modul (module) und Instanzen des Moduls
- Das kann mit Klasse und deren Objekt verglichen werden
- Modul hat Eingänge, Ausgänge und Inhalt
- Jedes Programm in Verilog startet mit Wort „module“ module_name <Liste der IOs> und endet mit „endmodule“
- Verilog unterscheidet zwischen klein und grosbuchsstaben
- Kommentar startet mit Zeichen // und endet mit neuer Zeile
- Kommentar mit mehreren Linien startet mit /* und endet mit */

```
module MyChildModule;
<code>
endmodule

module MyModule;
   <code>
   MyChildModule MyChildModuleA();
   MyChildModule MyChildModuleB();
   MyChildModule MyChildModuleC();
endmodule
```

```
module Counter (
   input clock,
   // Note the Bus definition before the name
   output [7:0] value
);
<code>
endmodule
```

- Weitere Besonderheit von Verilog sind die Signale bzw. Driver (Treiber)
- Es gibt zwei Arten von Drivers:
- 1. Drivers die Werte speichern können und 2. Drivers die das nicht können
- Jeder Driver kann ebenfalls mehrere Bits enthalten - Busstruktur
- a[7:0], output [7:0] value
- In Verilog bezeichnet man die Drivers mit Speicherfunktion mit dem Wort „register“ oder „reg“
- Beispiel eines Drivers mit Speicherfunktion ist flip-flop
- Ein Driver ohne Speichermöglichkeit verbindet zwei Knoten, In Verilog verwendet man dafür den Typ „wire“
- Die Driver-Werte können 1, 0, z (hochohmig, tristate), b (0/1) oder x (0/1/z) sein



- Verolog hat folgende Sprachelemente
- Arithmetische Operatoren zwischen Operanden/“Signalen“/Drivers/“Variablen“ (wire, reg) mit mehreren Bits
  - Binäre Operationen: +, -, *, /, % (Modulo-Operator)
  - Mit einem Operand: +, - (wird verwendet um Vorzeichen zu definieren)
  - Division rundet die Zahlen ab
- Bitweise Operatoren
  - Bitweise Operatoren führen eine mathematische Operation zwischen einzelnen Bits von zwei Operanden durch. Wenn ein Operand kleiner ist (weniger Bits hat) wird er mit Nullen erweitert.
  - &, ^, |, !, ~^
- Reduktionsoperatoren
- Diese Operatoren führen Bitweise Operation an jedem Bit eines Operands durch (jedes Bit ist ein Operand) und erzeugen ein Skalar (eine einzelnes Bit) als Ergebnis
- & 4'b1001 = 0

- Vergleichsoperationen
  - a < b
  - a less than b
  - a > b
  - a greater than b
  - a <= b
  - a less than or equal to b
  - a >= b
  - a greater than or equal to b
  - a == b
  - a equal b (Standardvergeliech von zwei Zahlen)
  - a === b
  - a equal b (Case oder Fall, berücksichtigt auch x und z)

- Logische Operatoren
  - &&, ||, !
  - Hier ist das Ergebnis immer ein Skalar
  - ! logic negation
  - && logical and
  - || logical or
- „Shift“ Operator
  - Bei diesen Operationen werden die Binär-Zeichen um eine angegebene Anzahl von Bitpositionen nach links oder rechts verschoben
- Verkettungsoperator
  - X[9:0] = {a, b[3:0], c, 4'b1001}
- Replikationsoperator multipliziert eine Guppe n-mal.
  - {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}
- **Konditionaloperator**
- **cond_expr ? true_expr : false_expr**

- Priorität von Operatoren
- Unary, Multiply, Divide, Modulus
- !, ~, *, /, %
- Add, Subtract, Shift
- +, - , <<, >>
- Relation, Equality
- <,>,<=,>=,==,!=,===,!==
- Reduction
- &, !&,^,^~,|,~|
- Logic
- &&, ||
- Conditional
- ? :

- Weitere Sprachelemente sind die Prozedur Blöcke
- Typ1: „initial“
  - Initial-Block, wird nur einmal am Anfang der Simulation ausgeführt, im Zeitpunkt 0
- Typ2: „always“
  - Always-Block kann eine Sensitivitätsliste haben
  - Always-Block wird immer ausgeführt wenn sich die Signale in der Sensitivitätsliste ändern, oder generell wenn das Ereignis aus der Liste stattfindet

- Sequenzielle Schaltungen werden normalerweise mit always Blöcken beschrieben deren Trigger Signale flanken-sensitiv sind. Normalerweise werden nonblocking Zuweisungen benutzt
- Falls wir mehrere always blöcke im Module haben, werden alle in Parallel ausgeführt. Manchmal führt es zu einer race condition oder zum Fehler wenn ein register in Mehreren Blöcken überschrieben wird

```
always @(posedge clock) begin
    value <= value +1;
end
```

```
always begin
        #5 clk =  ! clk;
end
```

- Prozedurzuweisungen „a=b“ werden in Prozedurblöcken verwendet, sie weisen die Werte den Registern zu. Sie können keine Zuweisung dem Typ wire machen (s. später „assign“)
- Man kann den Wert eines wire-Drivers dem Register zuweisen
- Falls ein Prozedurblock mehrere Zeilen (Ausdrücke) enthält, werden sie in ein begin - end oder ausnahmsweise join – fork block gruppiert. Alle Befehle in einem fork – join werden parallel gemacht
- Prozedurale Zuweisungen können blocking und nonblocking sein.
- Blocking werden in der Reihenfolge ausgeführt wie sie geschrieben wurden. Eine Zeile blockiert die Ausführung der nächsten, deswegen der Name. Blocking Zuweisungen werden mit den Symbol „=„ gemacht
- Nonblocking Zuweisungen werden in Parallel ausgeführt. Sie blockieren sich nicht. Symbol „<=„ wird verwendet

```
always @(posedge clock) begin
    value <= value +1;
end
```

```
initial begin
    clk = 0; // 0 Value
end
```

- Conditional-Statements (bedingte Anweisungen)
- if, else, case
- Prioritätslogik = verkettete if-else Anweisungen
- Wenn wir keine Prioritätslogik brauchen, weil wir wissen dass nur ein Signal in einem Moment aktiv sein kann, verwenden wir nur if
- Logische Implementierung von Prioritätslogik ist komplexer
- Statt viele verkettete if – else Anweisungen zu schreiben, eine für jeden Eingangswert den wir suchen, können wir einen case Befehl verwenden

```
always @(posedge clk or posedge reset) begin

if (reset) begin
      output <= 0;
   end
   else begin
      output <= output + 1;
   end

end
```

- Schleifen
- forever
- repeat
- while
- for

- Kontinuierliche Zuweisungen (Wort „assign“) werden außerhalb von Prozedurblöcken verwendet um kombinatorische Logik zu beschreiben
- Die linke Seite von einem „assign“ muss eine „wire“ sein
- Syntax : assign (strength, strength) #(delay) net = expression;

```
assign y = (a&b) | (c^d);

always…
```

Testbench

- Aufgabe:
- Wir möchten einen Zähler beschreiben und testen
- Wir fangen mit den Spezifikationen an: z.B. Zähler hat clock-, reset-, enable- Eingänge und einen 8-bit Ausgang cnt
- Wir definieren Test Cases (Testfälle, Tests)
- 1. Reset-Test
  - Wir starten mit dem ausgeschalteten Reset, einige Taktperioden später wird
  - reset aktiviert und wieder ausgeschaltet. Wir prüfen ob der Zähler-Ausgang auf null gesetzt wird
- 2. Enable-Test
  - Wir schalten enable ein und aus (nach dem Reset) und schauen ob der Zähler richtig zählt
- 3. Random-Test
  - Wir können sowohl Reset als auch Enable nach dem Zufallsprinzip ein- und ausschalten

Zähler code

- …

```
Module  counter #(parameter  [7:0]  MAXCNT  = 8h'FF)
 (
      Input wire clk,
      input wire reset,
      input wire enable,
      output [7:0] reg count
      );
      wire isMax;
      assign isMax = count == MAXCI

      always@(posedge clk) be
      if(reset) count <= 0;
      else if(enable) begin
            if (isMax) count <= 0;
            else count <= count + 1;
      end//end of no reset
      end//end of always
      endmodule
```

Parameter

Module Definition, Eingänge, Ausgänge

Typen von Hardware-Komponenten: reg, wire

Kein Komma

"Assign" Zuweisung (Kombinatorische Logik)

„Always" Block

nonblocking assignment "<="

Zähler code

- wenn wir eine Instanz vom conter Machen schreiben wir es so

```
wire [7:0] count;
reg clk;
reg reset;
reg enable;

counter #(. MAXCNT (8'hEE)) U0 (
        .clk    (clk),
        .reset  (reset),
        .enable (enable),
        .count  (count)
        );
```

Engang ist im Module Block immer vom „wire“ Type. Bei einer Instanz die Eingänge sind an „reg“ oder „wire“ angeschlossen

Ausgänge können im Module reg oder wire sein, bei Instanz Ausgänge werden immer an wire angeschlossen

Zähler code

- Um Zähler zu Simulieren brauchen wir eine Testbench

- Die einfachste testbench würde so aussehen

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter #(. MAXCNT (8'hEE)) U0 (
        .clk    (clk),
        .reset  (reset),
        .enable (enable),
        .count  (count)
        );
        initial begin
        clk = 0;
        reset = 0;
        enable = 0;
        end
        always begin
                    #5 clk =  ! clk
        end
        endmodule
```

Instanz des Zählers

Anfangsbedingungen

Taktgenerator

- Wir können jetzt entweder einen Simulator mit Waveform-Viewer verwenden und die Signale optisch zu verifizieren oder eine automatische Testroutine schreiben
- Falls wir automatische Testbench machen, brauchen wir ein Model das die Funktionalität von DUT nachahmt. Wir können dafür die high-level Sprachelemente wie Schleifen verwenden. Es wäre in unserem Fall einfach aber bei komplexen DUTs erfordert es Geschick
- Eine Testbench zu entwerfen ist oft ähnlich schwer wie RTL Code zu schreiben. Chipdesigner verbringen die meiste Zeit bei Verifikationen und Simulationen. Auch wenn das bekannt ist, wird die Simulationsarbeit nicht so hoch gewertet

- Schließlich brauchen wir eine „Checker“ Logik die in jedem Moment die Ausgänge von DUT und dem Model vergleicht und prüft ob wir das erwartete Ergebnis bekommen. Wenn ein Fehler passiert die Logik gibt Fehlermeldung aus. Die Logik kann auch die Simulation beenden

count_compare – Ausgang des Modells
count – Ausgang des Zählers

```
always @ (posedge clk)
    if (count_compare  ! = count) begin
    $display ("DUT Error at time %d", $time);
    $display (" Expected value %d, Got Value %d", count_compare, count);
    #5  -> terminate_sim;
    end
```

$display gibt die Signale aus jedes mal wenn er aufgerufen wird

%d Ausgabe von Variable (count_compare) im Dezimalformat

Es ist möglich Ereignisse zu definieren
Ereignisse werden getriggert mithilfe vom -> operator. Trigger aktiviert alle Prozesse die auf das Eregnis warten

Einige Beispiele

- Man kann die kombinatorische Logk auf zwei Weisen definieren
- Kombinatorische Schaltungen mit Always
- Kombinatorische Schaltungen mit Assign

- Definition als Prozeduralblock. Es kann passieren dass wir kombinatorische Logik beschreiben möchten aber wir bekommen ein Latch. Das passiert wenn ein Fall im case fehlt
- Wenn wir kombinatorische Logik mit Prozedur modellieren, der Always-Block muss auf jede Äderung von allen Eingängen reagieren.

```
wire [2:0] in;
reg [7:0] out;

always @ (in) begin
  out = 0;
  case (in)
    3'b001 : out = 8'b0000_0001;
    3'b010 : out = 8'b0000_0010;
    3'b011 : out = 8'b0000_0100;
    3'b100 : out = 8'b0000_1000;
    3'b101 : out = 8'b0001_0000;
    3'b110 : out = 8'b0100_0000;
    3'b111 : out = 8'b1000_0000;
  endcase
end//always
```

- Kombinatorische Schaltungen mit assign

```
input [2:0] in;
output [7:0] out;
wire [7:0] out;
assign out  =
(in == 3'b000 ) ? 8'b0000_0001 :
(in == 3'b001 ) ? 8'b0000_0010 :
(in == 3'b010 ) ? 8'b0000_0100 :
(in == 3'b011 ) ? 8'b0000_1000 :
(in == 3'b100 ) ? 8'b0001_0000 :
(in == 3'b101 ) ? 8'b0010_0000 :
(in == 3'b110 ) ? 8'b0100_0000 :
(in == 3'b111 ) ? 8'b1000_0000 : 8'h00;
```

- Wenn wir flipflop modellieren, benutzen wir ein Prozedurblock. Er ist mit positiver oder negativer Taktflanke getriggert. Wenn das flip-flip asynchrones reset hat, muss der Block sowohl auf Takt als auf Reset reagieren. Alle Zuweisungen werden „nonblocking“ gemacht.

Asynchrones Reset

```
always @ (posedge clk or posedge reset)
      if (reset) begin
      q <= 0;
      end else begin
      q <= d;
      end
```

Snchrones Reset

```
always @ (posedge clk)
      if (reset) begin
      q <= 0;
      else begin
      q <= d;
      end
```

Einige Regeln für guten Code-Stil

- In der SensitivitätsListe eines Always-Blocks sollen die flanken- und pegelsensitive Eingänge nicht gleichzeitig verwendet werden
- Klammer machen die kombinatorischen Formel übersichtlicher
- Kontinuierliche Zuweisung für kombinatorische Logik ist besser
- Es soll nonblocking für sequenzielle- und blocking für kombinatorische Logik verwendet werden
- Es ist besser nur die blocking- oder nur die nonblocking Zuweisungen in einem always Block zu benutzen
- Vorsicht bei Zuweisungen einer Variable an mehreren Stellen
- If ohne else ist schlecht

Advanced

- Advanced
- Generate Block ist manchmal wichtig – z.b. für Demultiplexer

```
generate
    for (i=0; i < 256; i=i+1) begin : DEMUX
        demux[i] <= (input == i);
    end
    endgenerate

generate
    for (i=0; i < 4; i=i+1) begin : MEM
        memory U (read, write,
        data_in[(i*8)+7:(i*8)],
        address,data_out[(i*8)+7:(i*8)]);
    end
    endgenerate
```

- Es ist möglich auf Singale aus den Moduleinstanzen im Top-Module zuzugreifen
- Dafür werden die hierarchische Namen verwendet: Instanzname.Signalname
- Das ist nützlich wenn wir sein Signal aus dem Top überschreiben möchten (z.B. im initial Block) oder wenn wir ein Signal verfolgen möchten
- tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum, tb.U.u3.sum

- Verilog hat auch primitive Gatter (primitive Gates), Schalter (switches, transmission gates). Sie werden selten im RTL Code benutzt aber sin für Modellierung von ASIC Zellen nützlich. Es handelt sich dann um eine Simulation auf Gatter Ebene (Standard Delay Format SDF Simulation)
- not U1(out0,in1);
- and U2(out1,in1,in2,in3,in4);
- xor U3(out2,in1,in2,in3);
- Verilog ermöglicht „Delays“ (Verzögerungen) mit den Gates zu assoziieren
- Rise, Fall and Turn-off delays.
- buf #(1,0)U_rise (rise_delay,in);
- buf #(0,1)U_fall (fall_delay,in);
- buf #1 U_all (all_delay,in);

- Verlilog hat Funktionen und Tasks
- Mithilfe von Funktionen können wir kombinatorische Funktionen definieren
- Tasks modellieren sowohl sequenz- als auch kombinatorischen Schaltungen. Delays sind erlaubt.
- Es gibt Tasks und Funktionen die Ausgaben (und Signale) während Simulationen generieren. Die Namen von diesen Elementen fangen mit $ an. Synthese ignoriert diese Zeilen und sie können auch im synthetisierbaren Teil von Code sein
- Beispiele sind $display und $strobe (geben die Signale aus jedes mal wenn sie aufgerufen werden) und $monitor (gibt Signal aus (oder speichert in einer Datei) immer wenn sich das Signal ändert)
- $random generiert Zufallszahl (Integer)
- Es ist möglich Ereignisse zu definieren
- Ereignisse werden getriggert mithilfe vom „->“ operator. Trigger aktiviert alle Prozesse die auf das Eregnis warten

Einige nützliche Links

- http://ipe-iperic-srv1.ipe.kit.edu/doc/dds/ss18/lecture/hdl/hdl.html
- http://www.asic-world.com/verilog/index.html
- https://stackoverflow.com/
- https://askubuntu.com/