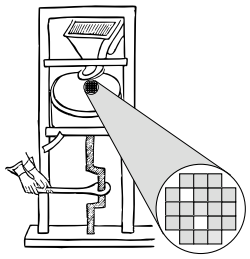
Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Prof. Dr. Andreas Koch
Holger Lange
Mathias Halbach (FG Rechnerarchitektur)



24.04.2006

Verilog Aufgabenblatt Sommersemester 2006

Aufgabe 1: Datentypen

a) Wie können Werte an **`wire`**-Variablen zugewiesen werden? Geben Sie Beispiele an. Können Wires Werte speichern?

Lösung:

```
wire s; assign s=1'b0;

wire s = a & b;
```

Wires können keine Werte speichern, nur transportieren.

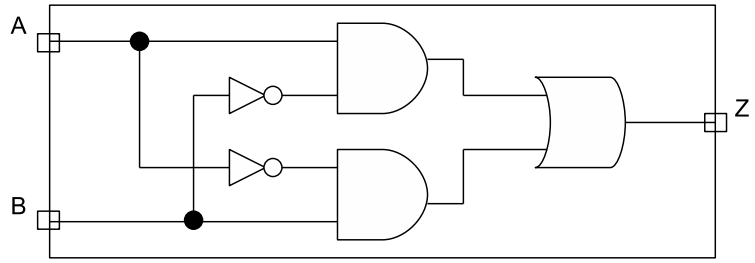
b) In welchem Fall beschreibt eine **`reg`**-Variable nach der Übersetzung in Hardware (Synthese) ein Flip-Flop, wann eine Drahtverbindung? Hinweis: Die Lösung hat etwas mit Aufgabe 2 b) zu tun…

Lösung:

Aus einer **`reg`**-Variablen wird nur ein Flip-Flop erzeugt, wenn ihr in einem sequenziellen **`always`**-Block taktsynchron Werte, z.B. von anderen **`reg`**s oder **`wire`**s, zugewiesen werden. In einem kombinatorischen **`always`**-Block (siehe auch Aufgabe 2 b)) hingegen dient auch eine **`reg`**-Variable nur als temporärer Zwischenspeicher und erzeugt deshalb eine Drahtverbindung in Hardware.

Aufgabe 2: Kombinatorische Logik

a) Schreiben Sie ein Verilog-Modul für die folgende Schaltung.



Lösung:

Es gibt eine Vielzahl möglicher Lösungen. Die folgenden drei zeigen die Konzepte ”permanente Zuweisung”, ”kombinatorischer `always`-Block” und ”Gatterprimitive”.

Permanente Zuweisung:

```
module aufgabe_2_a_1(A, B, Z);
  input A, B;
  output Z;
  wire notA, notB, and1, and2;
  assign notA = ~A;
  assign notB = ~B;
  assign and1 = A & notB;
  assign and2 = B & notA;
  assign Z = and1 | and2;
endmodule
```

Kombinatorischer `always`-Block:

```
module aufgabe_2_a_2(A, B, Z);
  input A, B;
  output Z;
  reg Z;
  always @(A or B) begin
    Z = (A & ~B) | (B & ~A); // alternativ: Z = A ^ B;
  end
endmodule
```

Gatterprimitive:

```
module aufgabe_2_a_3(A, B, Z);
  input A, B;
  output Z;
  xor xor1(Z, A, B);
endmodule
```

b) Auf welche Arten kann kombinatorische Logik in Verilog beschrieben werden?

Lösung:

Die erste Möglichkeit ist die permanente Zuweisung einer `wire`-Variablen mit

- **assign** (siehe **module aufgabe_2_a_1**) oder
- direkt bei ihrer Deklaration (**wire A = ...**).

Als zweite Möglichkeit kann ein kombinatorischer **always**-Block verwendet werden (**module aufgabe_2_a_2**). In diesem Fall wird aus in solchen Blöcken zugewiesenen **reg**-Variablen eine Drahtverbindung erzeugt (vgl. Aufgabe 1 b)). Schließlich können drittens, falls möglich, Gatterprimitive instanziert und geeignet verdrahtet werden (**module aufgabe_2_a_3**).

Aufgabe 3: Sequenzielle Logik

Gegeben ist folgendes Code-Fragment:

```
module foo(CLK, INP, OUT);
  input CLK, INP;
  output OUT;
  reg A, B;
  wire OUT=B;
  always @(posedge CLK) begin
    A <= INP;
    B <= A;
  end
endmodule
```

a) Was ist die Funktion dieses Moduls?

Lösung:

Der Eingang **INP** wird um zwei Takte (**CLK**) verzögert am Ausgang **OUT** ausgegeben.

b) Was passiert, wenn man die Zuweisungen vom Typ "<=" durch welche vom Typ "=" ersetzt? Erklären Sie die Unterschiede der beiden Zuweisungstypen und wann sie jeweils verwendet werden.

Lösung:

Bei der Verwendung der blockenden Zuweisung "=" rutscht der Wert von Signal **INP** im selben Takt schon nach **B** durch, in dem er erst **A** zugewiesen wird, wie bei einer Zuweisungsfolge in den meisten Programmiersprachen. Das Signal **INP** wird nur noch um einen Takt verzögert an **OUT** ausgegeben. Wenn man die beiden Zuweisungen vertauscht, ist das Problem nur teilweise gelöst. In diesem **always**-Block funktioniert alles richtig, aber andere *parallel* (zur gleichen Simulationszeit) ausgeführte **always**-Blöcke können zu früh die neuen Werte in **A** und **B** sehen, wenn sie noch die alten hätten lesen sollen. Die gleichzeitige Ausführung ist nicht mehr gewährleistet, denn nun hängt es davon ab, welcher der beiden Blöcke vom Simulator zuerst berechnet wird. Ein sogenanntes *Race* entsteht. Um dies zu vermeiden, berechnet die nicht-blockende Zuweisung "<=" zunächst alle Ausdrücke auf den rechten Seiten der Zuweisungen, speichert sie in einem transparenten Zwischenspeicher und weist sie im dritten Schritt den linken Seiten zu. Die nicht-blockende Zuweisung ist also keine atomare Operation, die Ausführungsreihenfolge daher egal und die Parallelität gewährleistet. In einem kombinatorischen **always**-Block hingegen (ohne Clock) soll das Ergebnis der Berechnung in der Regel sofort sichtbar sein und weiterverwendet werden können. Daher verwendet man für Berechnungen in

- sequentiellen **`always`**-Blöcken nur *nicht blockende* <=
- kombinatorischen **`always`**-Blöcken nur *blockende* =

Zuweisungen. Races werden dadurch zwar nicht komplett ausgeschlossen, aber können bei Einhaltung dieser Regel in den meisten Fällen vermieden werden.

Aufgabe 4: Ampel

a) Schreiben Sie eine Testumgebung analog zu Folie 31 der Verilog-Einführung für die Ampel-Zustandsmaschine (Folie 34). Die Ampel soll 10 Takte "rot" zeigen, danach einen Takt "rot-gelb", 5 Takte "grün" und schließlich 2 Takte "gelb", bevor sich das Spiel wiederholt.

Lösung:

```
module testbed();
  reg Clock, Schalte;
  wire Rot, Gelb, Gruen;

  ampel test(Clock, Schalte, Rot, Gelb, Gruen);

  initial begin
    Clock = 0;
    Schalte = 0;
  end

  always #1 Clock = ~Clock;

  initial #40 $finish;

  initial begin
    $dumpfile("ampel.vcd");
    $dumpvars(0);
    $monitor("Zeit: %2d Clock:%d Schalte:%d Rot:%d Gelb:%d Gruen:%d",
             $time, Clock, Schalte, Rot, Gelb, Gruen);
  end

  always begin
    #2  Schalte = 0; // Rotphase
    #18 Schalte = 1; // Schalten
    #2  Schalte = 1; // Rot-Gelb und gleich schalten
    #2  Schalte = 0; // Gruen
    #8  Schalte = 1; // Schalten
    #2  Schalte = 0; // Gelb
    #2  Schalte = 1; // Schalten
  end
endmodule
```

Der Systembefehl **`$monitor`** funktioniert ähnlich wie **`$display`**. Während **`$display`** alle Werte zum Zeitpunkt seiner Ausführung ausgibt (wie z.B. printf in C), gibt **`$monitor`** jedesmal alle Werte aus, wenn sich *ein* Wert in der Parameterliste ändert. Dazu braucht **`$monitor`** nicht jedesmal

aufgerufen zu werden, ein Aufruf startet die Überwachung bis zum Simulationsende. Die Systemvariable `$time` liefert die aktuelle Simulationszeit.

b) Simulieren Sie Ihr Ergebnis aus a) mit dem Icarus Verilog-Simulator wie auf den Folien 22 ff. der Verilog-Einführung beschrieben. Erzeugen Sie dabei Textausgaben und Waveforms aller Signale.

Lösung:

Die Textausgabe mit `$monitor` ergibt:

```
Zeit:  0 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  1 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  2 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  3 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  4 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  5 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  6 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  7 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  8 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit:  9 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 10 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 11 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 12 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 13 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 14 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 15 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 16 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 17 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 18 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 19 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 20 Clock:0 Schalte:1 Rot:1 Gelb:0 Gruen:0
Zeit: 21 Clock:1 Schalte:1 Rot:1 Gelb:1 Gruen:0
Zeit: 22 Clock:0 Schalte:1 Rot:1 Gelb:1 Gruen:0
Zeit: 23 Clock:1 Schalte:1 Rot:0 Gelb:0 Gruen:1
Zeit: 24 Clock:0 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 25 Clock:1 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 26 Clock:0 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 27 Clock:1 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 28 Clock:0 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 29 Clock:1 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 30 Clock:0 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 31 Clock:1 Schalte:0 Rot:0 Gelb:0 Gruen:1
Zeit: 32 Clock:0 Schalte:1 Rot:0 Gelb:0 Gruen:1
Zeit: 33 Clock:1 Schalte:1 Rot:0 Gelb:1 Gruen:0
Zeit: 34 Clock:0 Schalte:0 Rot:0 Gelb:1 Gruen:0
Zeit: 35 Clock:1 Schalte:0 Rot:0 Gelb:1 Gruen:0
Zeit: 36 Clock:0 Schalte:1 Rot:0 Gelb:1 Gruen:0
Zeit: 37 Clock:1 Schalte:1 Rot:1 Gelb:0 Gruen:0
Zeit: 38 Clock:0 Schalte:0 Rot:1 Gelb:0 Gruen:0
Zeit: 39 Clock:1 Schalte:0 Rot:1 Gelb:0 Gruen:0
```

Das gleiche Ergebnis als Waveform: