Fehlerkorrektur für Einzelbitfehler

G. Kemnitz*, TU Clausthal, Institut für Informatik

19. Juli 2007

Zusammenfassung

Bei der Speicherung von Daten in hochintegrierten DRAM-Schaltkreisen werden in einem mittleren zeitlichen Abstand von mehreren Stunden bis mehreren Tagen einzelne Bitstellen durch Alpha-Teilchen gelöscht. Alphateilchen entstehen durch radioaktive Zerfallsprozessen u. a. im Schaltkreisgehäuse und im Halbleiter. Zur Verhinderung von Datenverlusten werden Daten unter Nutzung fehlerkorrigierender Codes gespeichert und übertragen.

In der Übung sollen der Coder und die Korrekturschaltung für 4 Bit große Datenobjekte und Einzelbitfehlerkorrektur entwickelt und getestet werden.

1 Das Prinzip fehlerkorrigierender Codes

Ein fehlerkorrigierender Code teilt die Variationen des Codewortes in drei Teilmengen ein:

- Gültige Codeworte
- Ungültige korrigierbare Codeworte: Das sind Codeworte, die mit großer Wahrscheinlichkeit durch Verfälschung eines bestimmten gültigen Codewortes entstanden sind. Bei der Ergebniskorrektur werden sie wieder durch das zugeordnete gültige Codewort ersetzt.
- Ungültige nicht korrigierbare Codeworte: Das sind Codeworte, die mit vergleichbarer Wahrscheinlichkeit aus mehreren unterschiedlichen gültigen Codeworten entstanden sein könnten, so dass das ursprüngliche unverfälschte Codewort nicht mehr bestimmt werden kann.

Wird ein gültiges Codewort verfälscht, sind vier Fälle zu unterscheiden (Abb. 1):

1) Korrigierbare Verfälschung: Der Fehler verfälscht ein gültiges Codewort so, dass ein diesem Codewort zugeordnetes ungültiges Codewort entsteht. Eine solche Verfälschung kann später korrigiert werden, indem das ungültige Codewort wieder durch das ihm zugeordnete gültige Codewort ersetzt wird.
2) Maskierung von Datenfehlern durch Korrektur: Der Fehler verfälscht ein gültiges Codewort so, dass ein ungültiges Codewort entsteht, das einem anderen gültigen Codewort zugeordnet ist. Die Korrektur ersetzt das ungültige Codewort durch ein falsches gültiges Codewort.
3) Erkennen falscher Ergebnisse, ohne dass diese korrigiert werden können: Es entsteht ein ungültiges Codewort, das keinem der gültigen Codeworte zugeordnet ist. Der Datenfehler wird zwar erkannt. Eine Korrektur ist aber nicht möglich.

*Tel. 05323/727116

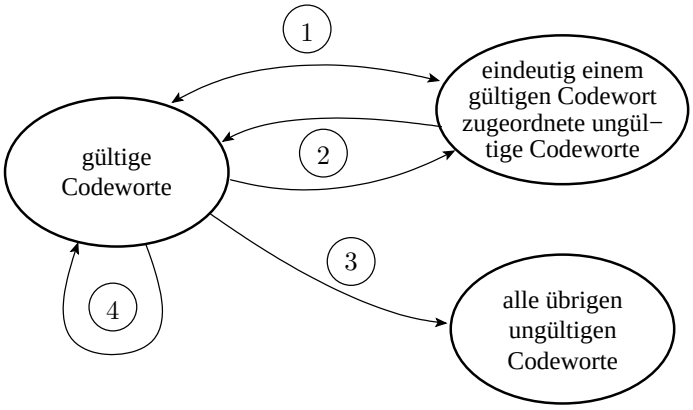

Abbildung 1: Möglichkeiten der Verfälschung eines gültigen Codewortes

4) Direkte Maskierung: Ein gültiges Codewort wird auf ein anderes gültiges Codewort abgebildet. Der Datenfehler wird weder erkannt, noch wird er korrigiert.

Der Trick eines fehlerkorrigierenden Codes besteht darin, die Codeworte so einzuteilen, dass der überwiegende Teil der zu erwartenden Verfälschungen korrigiert und der Rest der Verfälschungen möglichst erkannt wird.

Die Gesamtanzahl aller Codeworte ist durch die Anzahl der Variationen des gesamten Datenobjekts gegeben. Ein N_{Bit} großes Datenobjekt kann $2^{N_{\mathrm{Bit}}}$ Variationen annehmen. Diese Menge muss alle gültigen Codeworte und für jedes gültige Codewort alle ihm zugeordneten ungültigen Codeworte enthalten. Jeder fehlerkorrigierende Code muss folglich die Bedingung erfüllen:

$$N_{\mathrm{CWG}} \cdot \left(N_{\mathrm{CWK/CWG}} + 1\right) \leq 2^{N_{\mathrm{Bit}}} \tag{1}$$

(N_{CWG} – Anzahl der gültigen Codeworte; $N_{\mathrm{CWK/CWG}}$ – Anzahl der jedem gültigen Codewort zugeordneten ungültigen Codeworte; N_{Bit} – Bitanzahl der Codeworte). Die Anzahl der korrigierbaren Verfälschungen eines Codewortes ist dabei stets wesentlich kleiner als die Anzahl der möglichen Variationen für die Verfälschung des N_{Bit} Bit großen Datenobjekts:

$$N_{\mathrm{CWK/CWG}} \leq \frac{2^{N_{\mathrm{Bit}}}}{N_{\mathrm{CWG}}} - 1 \tag{2}$$

Ein fehlerkorrigierender Code kann nur einen kleinen Teil der möglichen Bitverfälschungen korrigieren.

Bei der Übertragung und Speicherung von Daten werden meist nur wenige Bits verfälscht. Oft ist es nur ein einzelnes falsches Bit. Seltener werden zwei oder mehr unabhängige Bits innerhalb eines Datenobjekts verfälscht. Der fehlerkorrigierende Code braucht nur Verfälschungen von bis zu N_{Korr} Bits zu korrigieren. Diese Vorüberlegung führt zu den Hamming-Codes.

Ein N_{Bit} Bit großes Codewort mit maximal N_{Korr} verfälschten Bitstellen kann in

$$N_{\mathrm{CWK/CWG}}\left(N_{\mathrm{Korr}}\right) = \sum_{i=1}^{N_{\mathrm{Korr}}} \left(\begin{array}{c} N_{\mathrm{Bit}} \\ i \end{array} \right) \tag{3}$$

Varianten verfälscht sein. Es muss nach Gl. 1 die Bedingung:

$$N_{\mathrm{CWG}} \cdot \left(\sum_{i=1}^{N_{\mathrm{Korr}}} \left(\begin{array}{c} N_{\mathrm{Bit}} \\ i \end{array} \right) + 1 \right) \leq 2^{N_{\mathrm{Bit}}} \tag{4}$$

erfüllen. Angenommen, es gibt $N_{\mathrm{CWG}} = 2^w$ gültige Codeworte und es ist nur 1 Bit zu korrigieren, dann muss für die Bitanzahl des Codewortes gelten:

$$2^w \cdot (N_{\mathrm{Bit}} + 1) \leq 2^{N_{\mathrm{Bit}}}$$

Ein $w = 11$ Bit großes Datenwort muss, damit alle Einzelbitfehler korrigierbar sind, mindestens $N_{\mathrm{Bit}} = 15$ Bit groß sein. Dazu muss es um eine $r = 4$ Bit große Prüfsumme erweitert werden.

2 Mod2-Prüfbits und -Prüfsummen[1]

Die allgemeine Berechnungsvorschrift eines Mod2-Prüfbit q für ein w Bit großes Datenobjekt X ist das Mod2-Skalarprodukt des als Bitvektor betrachteten Datenobjekts mit einem Koeffizientenvektor $V \neq 0$ gleicher Bitanzahl:

$$\begin{aligned} q &= V \cdot X \\ &= \left(\begin{array}{cccc} v_1 & v_2 & \ldots & v_w \end{array} \right) \cdot \left(\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_w \end{array} \right) \end{aligned} \tag{5}$$

(w – Bitanzahl des Datenobjekts). Jedes Bit des Datenobjekts wird mit dem zugehörigen Bit des Koeffizientenvektors mod2-multipliziert und die entstehenden Teilprodukte $v_i \cdot x_i$ werden mod2-aufsummiert:

$$q = \bigoplus_{i=1}^{w} (v_i \cdot x_i) \tag{6}$$

(X – Datenobjekt ; x_i – Bits i des Datenobjekts; V – Konstante; v_i – Bit i der Konstanten).

Eine Mod2-Prüfsumme ist ein Bitvektor aus mehreren Prüfbits. Jedes Prüfbit q_j ist das Mod2-Skalarprodukt mit einem anderen Koeffizientenvektor $V_j \neq 0$:

$$\left(\begin{array}{c} q_1 \\ q_2 \\ \vdots \\ q_r \end{array} \right) = \left(\begin{array}{c} V_1 \\ V_2 \\ \vdots \\ V_r \end{array} \right) \cdot \left(\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_w \end{array} \right) \tag{7}$$

(r – Anzahl der Prüfbits). Alle Koeffizientenvektoren zusammen bilden die Koeffizientenmatrix A:

$$\left(\begin{array}{c} q_1 \\ q_2 \\ \vdots \\ q_r \end{array} \right) = \underbrace{\left(\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1w} \\ a_{21} & a_{22} & \cdots & a_{2w} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r2} & \cdots & a_{rw} \end{array} \right)}_{A} \cdot \left(\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_w \end{array} \right)$$

[1] mod2 ist die Abkürzung für modulo-2

$$
\begin{aligned}
q_i &= \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_{15} \end{pmatrix} \\
&= x_{15} \oplus x_{11} \oplus x_8 \oplus x_4 \oplus x_3 \oplus x_1
\end{aligned}
$$

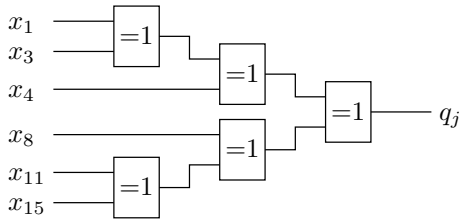

Abbildung 2: Bildung eines Prüfbits mit EXOR-Gattern

(w – Bitanzahl des Datenobjekts; r – Anzahl der Prüfbits). Die Mod2-Prüfsumme Q ist entsprechend das Produkt einer Koeffizientenmatrix mit dem als Bitvektor betrachteten Datenobjekt X:

$$Q = A \cdot X \tag{8}$$

In einer Hardwarerealisierung werden die UND-Verknüpfungen mit Konstanten wegoptimiert. Eine Mod2-Summe mit einem Produkt, dessen einer Faktor immer Null ist, entfällt:

$$x_1 \oplus (0 \cdot x_2) = x_1$$

Eine Mod2-Addition mit einem Produkt, dessen einer Faktor immer Eins ist, wird zu einer einfachen Mod2-Addition:

$$x_1 \oplus (1 \cdot x_2) = x_1 \oplus x_2$$

Aus dem Skalarprodukt eines Datenobjekts mit einem Koeffizientenvektor wird die Mod2-Summe aller Datenbits, deren zugehörige Koeffizientenbits 1 sind:

$$q_i = \bigoplus_{\forall j | a_{i,j}=1} x_j \tag{9}$$

Die Mod2-Summe selbst wird in Hardware durch eine Baumstruktur aus EXOR-Gattern nachgebildet (Abb. 2).

Wenn die Summen zur Berechnung mehrerer Prüfbits gleiche Teilsummen enthalten, im Beispiel

$$
\begin{aligned}
q_1 &= x_1 \oplus x_3 \oplus x_7 \oplus x_{10} \\
q_2 &= x_1 \oplus x_4 \oplus x_8 \oplus x_{10}
\end{aligned}
$$

die Teilsumme $x_{10} \oplus x_1$, lässt sich die Anzahl der Mod2-Additionen durch gemeinsame Berechnung dieser Teilsummen verringern:

$$
\begin{aligned}
z &= x_1 \oplus x_{10} \\
q_1 &= z \oplus x_3 \oplus x_7 \\
q_2 &= z \oplus x_4 \oplus x_8
\end{aligned}
$$

Die zugehörigen Schaltungen zeigt Abb. 3.

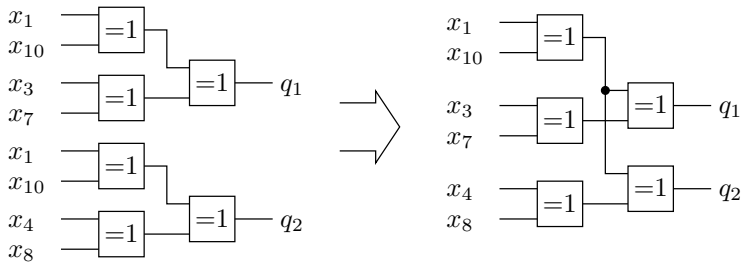

Abbildung 3: Optimierung von EXOR-Schaltungen

2.1 Beispielcode und Korrekturschaltung

Das Grundkonzept lautet, der Codierer erzeugt eine Mod2-Prüfsumme

$$Q_1 = A \cdot X \tag{10}$$

ausreichender Größe für das Datenobjekt, die gemeinsam mit dem Datenobjekt übertragen und gespeichert wird. Nach der Übertragung oder Speicherung wird die Prüfsummen in derselben Weise aus dem potentiell verfälschten Datenobjekt gebildet

$$Q_2 = A \cdot (X \oplus \Delta X) \tag{11}$$

und mit der übertragenen oder gespeicherten Prüfsumme, die auch verfälscht sein kann, verglichen:

$$\Delta Q = Q_1 \oplus \Delta Q_1 \oplus Q_2 \tag{12}$$

Ist die Abweichung $\Delta Q = 0$, gilt das empfangene Datenobjekt als richtig. Bei einer Abweichung ungleich Null wird aus der Abweichung das (die) zu korrigierende(n) Bit(s) bestimmt und korrigiert.

Für die Korrektur von nur einem Bit lässt sich die Prüfsumme so wählen, dass die Abweichung ΔQ in binärer Form die Nummer des zu korrigierenden Bits im Gesamtcodewort B angibt. Das soll am Beispiel eines Codes für $w = 11$ Bit große Datenobjekt und $r = 4$ Bit große Prüfkennzeichen vorgeführt werden. Der Wert $\Delta Q = 0$ ist für den Fall »kein falsches Bit« reserviert. Alle anderen Werte $\Delta Q = i$ bedeuten, dass Bit i, $1 \leq i \leq 15$ verfälscht ist. Bit 0 von ΔQ soll genau dann Eins sein, wenn die Nummer der verfälschten Bitstelle im übertragenen oder gespeicherten Codewort B ungerade ist :

$$\Delta q_0 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \oplus b_{13} \oplus b_{15} \tag{13}$$

Bit Δq_i der Differenz der Prüfkennzeichen soll verfälscht sein, wenn die Nummer des verfälschten Bits den Summanden 2^i enthält:

$$\Delta q_1 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \oplus b_{14} \oplus b_{15} \tag{14}$$

$$\Delta q_2 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \tag{15}$$

$$\Delta q_3 = b_8 \oplus b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \tag{16}$$

Das Prüfkennzeichen ist ein Teil des potentiell verfälschten Gesamtvektors. Ein Bit des Prüfzeichens kann deshalb nur den Bitnummern des Gesamtvektors zugeordnet sein, deren Bits sie aufsummiert. Eine zulässige Zuordnung ist:

$$q_0 \Rightarrow b_1 \;\Big|\; q_1 \Rightarrow b_2 \;\Big|\; q_2 \Rightarrow b_4 \;\Big|\; q_3 \Rightarrow b_8$$

Die übrigen Bits des Gesamtvektors sind für Datenbits frei. Als Zuordnung kann z. B. gewählt werden:

$x_0 \Rightarrow b_3$	$x_3 \Rightarrow b_7$	$x_6 \Rightarrow b_{11}$	$x_9 \Rightarrow b_{14}$
$x_1 \Rightarrow b_5$	$x_4 \Rightarrow b_9$	$x_7 \Rightarrow b_{12}$	$x_{10} \Rightarrow b_{15}$
$x_2 \Rightarrow b_6$	$x_5 \Rightarrow b_{10}$	$x_8 \Rightarrow b_{13}$	

Die Berechnungvorschrift für das Prüfkennzeichen lautet mit dieser Zuordnung:

$$
\begin{aligned}
q_0 &= b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \oplus b_{13} \oplus b_{15} \\
&= x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_{10} && (17) \\
q_1 &= b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \oplus b_{14} \oplus b_{15} \\
&= x_0 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_9 \oplus x_{10} && (18) \\
q_2 &= b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \\
&= x_1 \oplus x_2 \oplus x_3 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{10} && (19) \\
q_3 &= b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \\
&= x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{10} && (20)
\end{aligned}
$$

Abb. 4 zeigt die Schaltung zur Codeerzeugung und zur Datenkorrektur. Vor der potentiellen Datenverfälschung wird die 4-Bit-Prüfsumme berechnet und zum eigentlichen Datenvektor hinzugefügt. Nach der potentiellen Verfälschung wird die Mod2-Prüfsumme nach demselben Algorithmus zum zweiten Male gebildet und zur übertragenen (und auch eventuell verfälschten) Prüfsumme mod2 addiert. Ist die Differenz $\Delta Q \neq 0$, wird das Bit, dessen Nummer gleich der Differenz ist, negiert.

3 Aufgabenstellung

Entwickeln und Testen Sie einen Code, die Codierschaltung und die Korrekturschaltung für 4 Bit große Datenobjekte und Einzelbitfehlerkorrektur. Abb. 5 zeigt einen Vorschlag zum Ausprobieren der gesamten Fehlerkorrekturlösung. Die vier unverfälschten Eingabebits werden von den Tastern als codierter Vektor btn_cod(3 downto 0) bereitgestellt. Die Verfälschung der einzelnen Bits erfolgt durch EXOR-Verknüpfung der einzelnen Daten- und Kontrollbits mit der Eingabe von den Schaltern sw(7 downto 1). Die Eingabe, die verfälschten Bits und die Ausgabe sind auf LEDs geführt.

4 Lösungsschritte

- Aufstellung der Gleichungen für die Berechnung der Prüfsumme
- Anlegen eines neuen Projektes (Einstellungen für Schaltkreis und Designprozess wie im Tutorial)
- Anlegen eines neuen VHDL-Moduls mit dem Namen Fehlerkorrektur.vhd und der Schnittstellenbeschreibung:

```
ENTITY Fehlerkorrektur IS
 PORT( btn_cod: IN std_logic_vector(3 DOWNTO 0);
       sw: IN std_logic_vector(7 DOWNTO 1);
       led: OUT std_logic_vector(15 downto 0));
END Fehlerkorrektur;
```

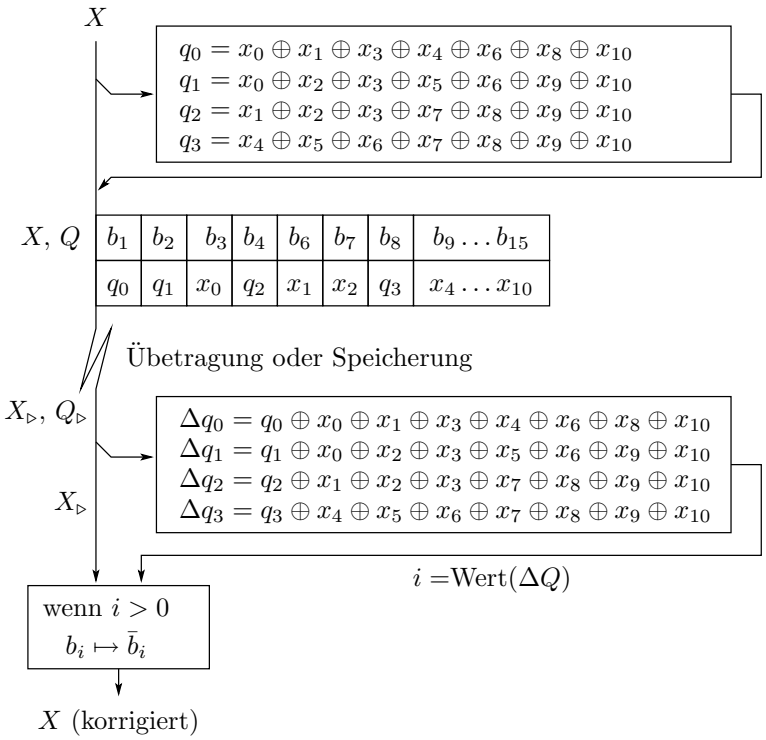

Abbildung 4: Fehlerkorrektur mit Hilfe linearer Summennetzwerke

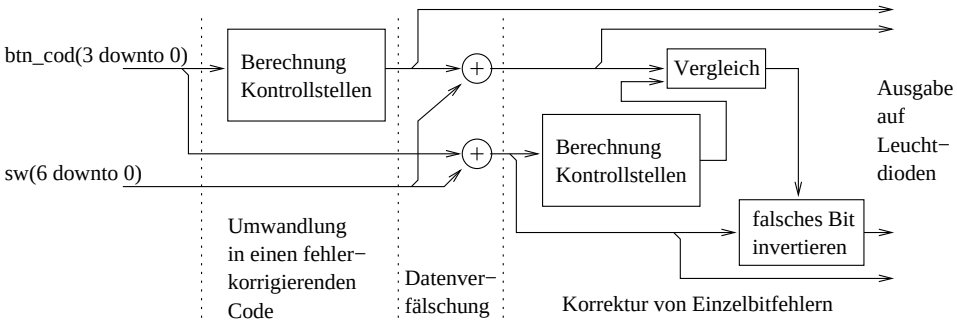

Abbildung 5: Versuchsaufbau zum Test der Fehlerkorrekturschaltung

- Einbinden von Praktikum.ucf aus den anderen Aufgaben und Zuordnung zu Fehlerkorrektur.vhd.
- Anlegen eines neuen VHDL-Moduls "PSum.vhd" mit der Schnittstellenbeschreibung:

```
ENTITY PSum IS
 PORT(x: IN std_logic_vector(3 DOWNTO 0);   -- Eingabebits
      k: OUT std_logic_vector(2 DOWNTO 0);  -- Kontrollstellen
END PSum;
```

- Schreiben der ARCHITECTURE zur Prüfsummenberechnung, Kontrolle durch Syntaxtest und Simulation
- Schreiben der ARCHITECTURE für die Gesamtschaltung und Einbindung der Prüfsummenberechnung als COMPONENT.
- Übersetzen wie im Tutorial mit den Optionen »Allow unused constraints« und »JTAG-Clock«
- Download und Test
- Was passiert, wenn gleichzeitig mehr als ein Bit verfälscht wird? (Protokollieren Sie die entsprechenden Testergebnisse.)

5 Aufräumen

- Über Menüpunkt "Project, Cleanup Project Files" automatisch generierte Design-Files löschen.
- Netzteil zur Spannungsversorgung aus der Steckdose ziehen.
- Modelsim und Projektnavigator beenden.