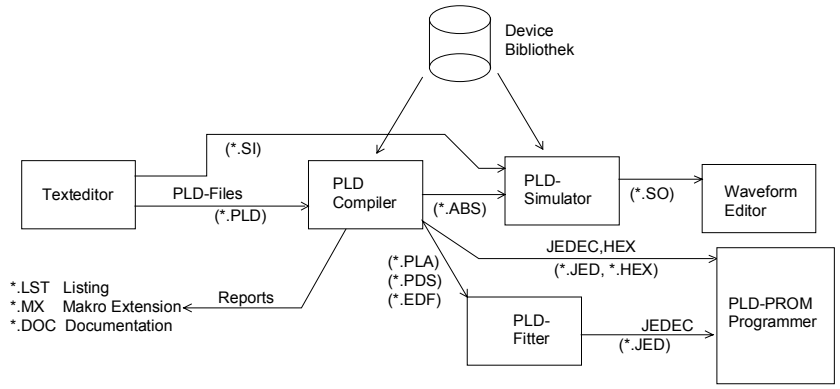
Projektwoche E98/2000: PLD Design mit Protel 98

Umfeld

Zum Protel ECAE-Paket gehört auch das Advanced PLD Modul. Es erlaubt die Synthese und Simulation von PLD Bausteinen.
Die Eingabe erfolgt mit der Sprache CUPL. Dies ist eine einfache Beschreibungssprache für Boolsche Gleichungen und Sequenzabläufe. Einfache Makrofähigkeiten und Konfigurationskommandos runden die Sprachdefinition ab.



Module und Files im Protel98 Advanced PLD.

So gesehen ist das Protel PLD Paket als Gesamtlösung zu sehen. Unterstützt werden gängige PLD Bausteine bis hin zu komplexen XILINX und MACH PLD. Spezialbausteine benötigen einen eigenen herstellerspezifischen Filter, der den PLD Code zur Programmierung aufbereitet.

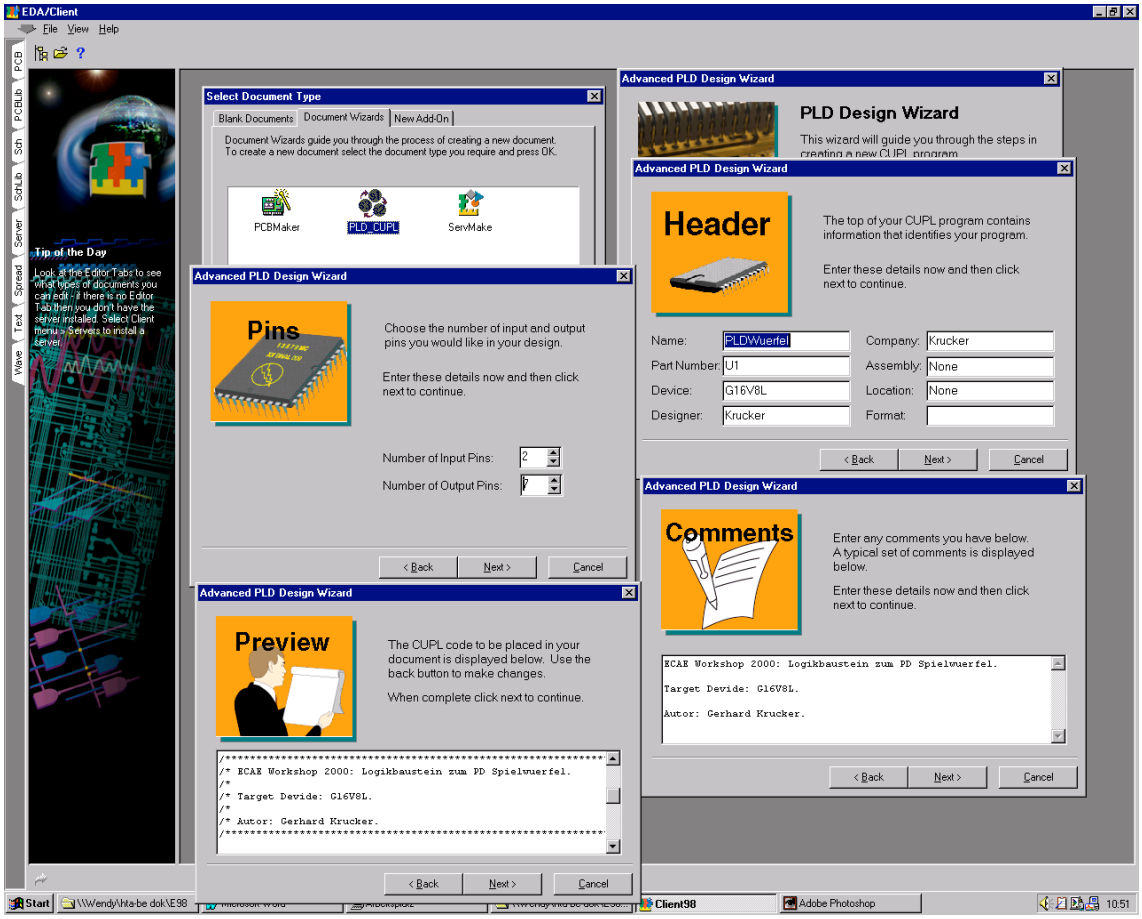
Der PLD-Entwurf erfolgt entweder direkt durch Eingabe in einen Texteditor oder über einen Konfiguration Wizzard, der das Gerüst des PLD Programmes erzeugt.

Von der Leistungsfähigkeit her gesehen, ist das Protel-Produkt eher in der tieferen Mittelklasse anzusehen. Wesentliche Gründe hierfür sind: Ein schwacher PLD Simulator. Der PLD Compiler ist langsam und absolut Fehlerintolerant. Eine Fehlersuche im PLD Code ist äusserst mühsam bis unmöglich. Der Texteditor zur Eingabe weist erhebliche Mängel auf. Vermutlich liegen die Gründe darin, dass das PLD Paket in Teilen von verschiedenen Herstellern zugekauft wurden und nur unvollständig eingebunden worden sind.

Hinweis:
Der PLD Compiler erlaubt keine deutschen Umlaute, weder in Symbolen noch in Kommentaren. Werden deutsche Umlaute benutzt, erscheinen Fehlermeldungen beim Kompilieren: Sie weisen aber nicht direkt auf diese Probleme hin.

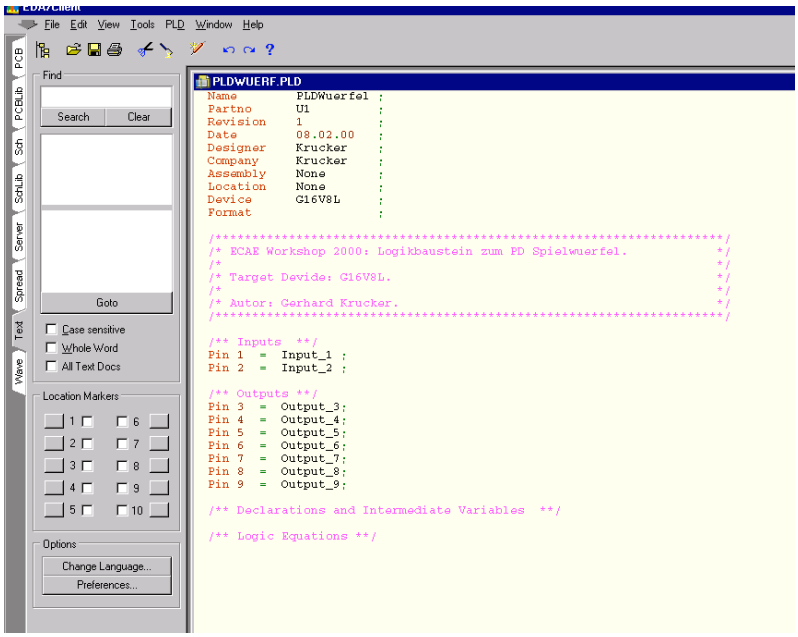
Start des PLD-Paketes

Zweckmässigerweise erstellt man ein neues PLD-Design über den PLD-CUPL Design-Wizzard. Er erstellt über einen Dialog das Grundgerüst des PLD Files. Man kann sich so eine Menge Schreibarbeit ersparen:



CUPL PLD Wizzard zum Dialoggeführten Erstellen eines Codegerüstes.

Nach Abschluss des Dialoges wird das Editorfenster geöffnet und der generierte PLD-Quelltext angezeigt:



Vom Wizzard erzeugtes Codegerüst..

Ein bereits bestehendes PLD-File kann durch normales Öffnen in den Editor geholt werden. An der *.PLD und dem Code erkennt der syntaxsensitive Editor, dass es sich um ein CUPL-PLD-File handelt und zeigt dies mit der syntaxeinfärbung.

Struktur eines CUPL Designs

CUPL ist als einfache Programmiersprache für Logikbausteine zu verstehen. Es gelten daher analoge Regeln für den Quelltext wie für andere Sprachen auch. Die typische Struktur eines CUPL-Files ist:

Programmkopf: Autor, Targetdevice, Datum, Revision, etc.
Textuelle Beschreibung/ Kommentar
Pindefinitionen
Intermediate Variablen (≈Lokalvariablen)
Boolsche Gleichungen / State Machine Definitionen
Technologieanweisungen

CUPL Syntax

`;`	Abschluss eines Statements
`/* ...*/`	Kommentar
`=`	Zuweisung
`[]`	Bereichklammer
`{}`	Indexklammern für Makros/Schleifen
`.`	Selektionsoperator
`:`	Bereichsauswahloperator

Preprozessoranweisungen

`$define`	Symboldefinition
`$IFDEF ..$ELSE...$ENDIF`	Bedingte Kompilation
`$IFNDEF ..$ELSE...$ENDIF`	Bedingte Kompilation
`$REPEAT..$REPEND`	Wiederholung
`$MACRO..$MEND`	Makrodefinition

State Machine Anweisungen

```
SEQUENCE[D,JK,T]
{ PRESENT state
   [IF condition] NEXT state [OUT pin]
}

FUNCTION name ([parameter0,..,parametern])
{...
}
```

Steueranweisungen

`MIN` Minimierungsverfahren (kann selektiv für jeden Ausgang gewählt werden)

0	keine Minimierung
1	Quick
2	Quine-McCluskey
3	Presto
4	Expresso

`FUSE(101,1)` Technologieabhängige Fuse brennen (z.B. Ausleseschutzsicherung)

Logische Operatoren

Operator	Beispiel	Wirkung	Hierarchie
`!`	`!A`	Negation	1
`&`	`A&B`	UND	2
`#`	`A#B`	ODER	3
`$`	`A$B`	EXOR	4

Zahlen

Zahl	Basis	Dezimalwert
`'b'0`	Binär	0
`'B'1101`	Binär	13
`'O'663`	Oktal	435
`'D'92`	Dezimal	92
`'h'BA`	Hexadezimal	186
`'O'[300..477]`	Oktalbereich	192..314
`'H'7FXX`	Hexadezimalbereich	32512..32767

Felder

Zuordnung einer einzelnen Variablen zu einer bestimmten Anzahl Bits.
Bsp:

```
FIELD ADDRESS = [A7,A6,A5,A4,A3,A2,A1,A0];
sel=[A0,A1,A2,A3]:&;
sel=ADDRESS:[0..F];
```

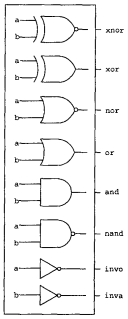
Arithmetische Operatoren und Funktionen (nur in `$MACRO` und `$REPEAT` benutzbar)

Operator	Beispiel	Beschreibung	Hierarchie
`**`	`2**3`	Potenzierung	1
`*`	`2*i`	Multiplikation	2
`/`	`4/2`	Division	2
`%`	`9%8`	Modulus	2
`+`	`2+4`	Addition	3
`-`	`4-i`	Subtraktion	3

`LOG2`	`LOG2(32)`	Zweierlogarithmus (abgerundet auf Ganzes)
`LOG8`	`LOG8(32)`	Logarithmus zur Basis 8
`LOG16`	`LOG16(32)`	Logarithmus zur Basis 16
`LOG`	`LOG(20)`	Logarithmus zur Basis 10

Beispiel 1: Kombinatorische Logik mit Boolschen Gleichungen

```
Name            Gates;
Partno          CA0001;
Revision        04;
Date            9/12/89;
Designer        G. Woolhiser;
Company         Logical Devices, Inc.;
Location        None;
Assembly        None;
Device          G16V8;

/*****************************************************************/
/*                                                               */
/*      This is a example to demonstrate how CUPL                */
/*      compiles simple gates.                                   */
/*                                                               */
/*****************************************************************/
/*      Target Devices: P16L8, P16LD8, P16P8, EP300, and 82S153 */
/*****************************************************************/

/*
 * Inputs:  define inputs to build simple gates from
 */

Pin 1 =  a;
Pin 2 =  b;

/*
 * Outputs:  define outputs as active HI levels
 *
 * Note: For PAL16L8 and PAL16LD8, DeMorgan's Theorem is applied to
 *       invert all outputs due to fixed inverting buffer in the device.
 */

Pin 12 = inva;
Pin 13 = invb;
Pin 14 = and;
Pin 15 = nand;
Pin 16 = or;
Pin 17 = nor;
Pin 18 = xor;
Pin 19 = xnor;

/*
 * Logic:  examples of simple gates expressed in CUPL
 */

Demorgan [nor,xor,xnor] = 2;

inva = !a;              /* inverters */
invb = !b;
and  = a & b;           /* and gate */
nand = !(a & b);        /* nand gate */
or   = a # b;           /* or gate */
nor  = !(a # b);        /* nor gate */
xor  = a $ b;           /* exclusive or gate */
xnor = !(a $ b);        /* exclusive nor gate */
```



Beispiel 2: Ampelsteuerung mit Sequenzmaschine

```
Name       trfk;
Partno     XXXXX;
Date       05/08/90;
Revision   02;
Designer   SDB;
Company    Logical Devices, Inc.;
Assembly   XXXXX;
Location   XXXXX;
Device     p20g10;

/*****************************************************************/
/*                                                               */
/*                     Traffic Controller                        */
/*                                                               */
/*   A Moore state machine design that controls traffic at the   */
/*   intersection of two one-way streets.                        */
/*                                                               */
/*****************************************************************/
/*   Allowable Target Device Types:  20G10 and 22V10             */
/*****************************************************************/

/**  Inputs  **/

Pin     1    = Clock     ;      /*  State machine clock          */
Pin     2    = Sen1      ;      /*  Sensor for direction 1       */
Pin     3    = Sen2      ;      /*  Sensor for direction 2       */
Pin    13    = Enable    ;      /*  Output enable                */
```

```
/**  Outputs  **/

Pin [17..19] = [Q2..0]   ;      /*  State machine registers       */

Pin    14    = !Red1     ;      /*  Red light for direction 1     */
Pin    15    = !Yellow1  ;      /*  Yellow light for direction 1  */
Pin    16    = !Green1   ;      /*  Green light for direction 1   */

Pin    23    = !Red2     ;      /*  Red light for direction 2     */
Pin    22    = !Yellow2  ;      /*  Yellow light for direction 2  */
Pin    21    = !Green2   ;      /*  Green light for direction 2   */

/** Declarations and Intermediate Variable Definitions **/

/* Define the state bits for the individual states */
$define  S0  'b'000
$define  S1  'b'001
$define  S2  'b'010
$define  S3  'b'011
$define  S4  'b'100
$define  S5  'b'101
$define  S6  'b'110
$define  S7  'b'111

/* Define the state bits as a field for the state machine */
Field Traffic = [Q2..0];

/**  Logic Equations  **/

Sequenced Traffic  {

    Present  S0
        if  Sen1 &  Sen2   Next S1;
        if !Sen1 & !Sen2   Next S1;
        if !Sen1 &  Sen2   Next S3;
        if  Sen1 & !Sen2   Next S0;
        Out Green1  Out Red2;

    Present  S1
        Default Next S2;
        Out Green1  Out Red2;

    Present  S2
        Default Next S3;
        Out Green1  Out Red2;

    Present  S3
        Default Next S4;
        Out Yellow1 Out Red2;

    Present  S4
        if  Sen1 &  Sen2   Next S5;
        if !Sen1 & !Sen2   Next S5;
        if !Sen1 &  Sen2   Next S4;
        if  Sen1 & !Sen2   Next S7;
        Out Red1    Out Green2;

    Present  S5
        Default Next S6;
        Out Red1    Out Green2;

    Present  S6
        Default Next S7;
        Out Red1    Out Green2;

    Present  S7
        Default Next S0;
        Out Red1    Out Yellow2;
}

/* Minimize state machine equations */
MIN [Q2..0].d = 4;

/* Enable the outputs by using a Low-True input pin */
[Q2..0].oe = !Enable;
Red1.oe    = !Enable;
Yellow1.oe = !Enable;
Green1.oe  = !Enable;
Red2.oe    = !Enable;
Yellow2.oe = !Enable;
Green2.oe  = !Enable;
```

Editieren des CUPL-PLD-Files

Das Grundgerüst wird nun im Editor erweitert. Dazu gehören primär:

Umbennung der vom Wizzard erzeugten Pindefinitionen
Einbringen der Logik in Form von Boolschen Gleichungen/ Sequenzabläufen

Der PLD Compiler zeigt absolut keine Fehlertoleranz und ist meist nicht in der Lage festzustellen, wo genau Fehler im Quelltext lokalisiert sind. So ist ein Compilerdurchlauf praktisch ein Go/No Go Spiel, das aber mit einer gewissen Sorgfalt doch rasch zum Ziel führt.

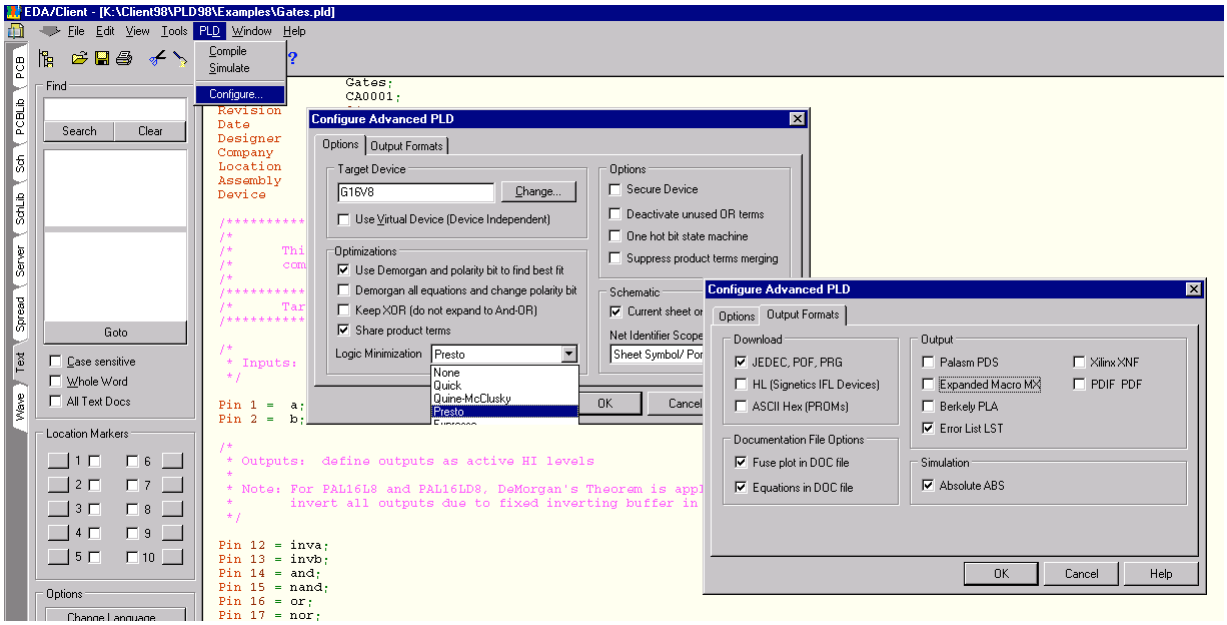
Die Logik selbst wird aus dem Entwurf der Lösung abgeleitet. Dies kann direkt durch Eingabe der Boolschen Gleichungen erfolgen.

Hinweise
Die Eingabe der eckigen und geschweiften Klammern []{} sowie des ODER-Operators # ist im Editor nicht möglich über die Tastatur einzugeben. Workaround: Im Notepad diese Zeichen schreiben und über das Clipboard in den Quelltext kopieren.

Bei groben Fehlern im CUPL-File wird das Quelltextfile vom Compiler nicht mehr geschlossen. Da kann im Editor das Quelltextfile nicht weiter geändert werden, weil es bis zum nächsten Start von Protel98 als *readonly* angesehen wird. In diesem Fall speichert man das File am besten unter einem neuen Namen und compiliert neu.

Compilieren

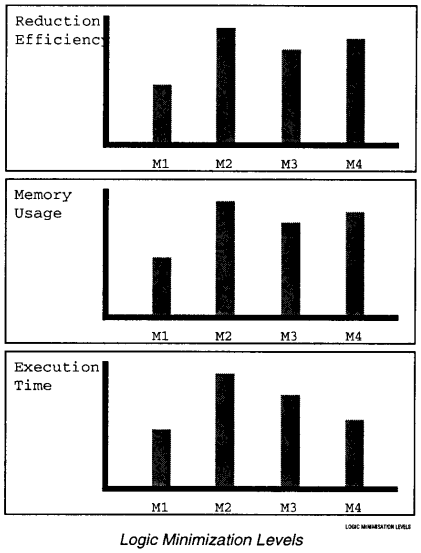
Der Kompilationsdurchlauf erzeugt aus dem CUPL-Quelltext einen downloadbaren Code für das Programmiergerät (JEDEC). Ferner werden verschiedene Listings- und Simulationsfiles erzeugt. Die Ausgaben, sowie Bausteinauswahl werden über den PLD-Konfigurationsdialog im Menü *PLD/Configure* ausgeführt:



PLD Compiler Konfigurationsdialog.

Die Bausteinauswahl im Dialog wirkt sich direkt auf den zugehörigen Eintrag im Quelltext aus. Für erste Entwürfe ist das *Virtual Device* besonders praktisch. Es verkörpert ein ideales PLD, dass alles kann, also keine Einschränkungen bezüglich Makrozellen, Produktterme, etc. aufweist.

Die Optimierungsoptionen bringen fallweise vor allem bei knappen Bausteinressourcen etwas. Bei grösseren Designs können die Optimierungen aber sehr lange dauern. Gemäss Protel-Handbuch wirkt sich die Methode wie folgt aus:

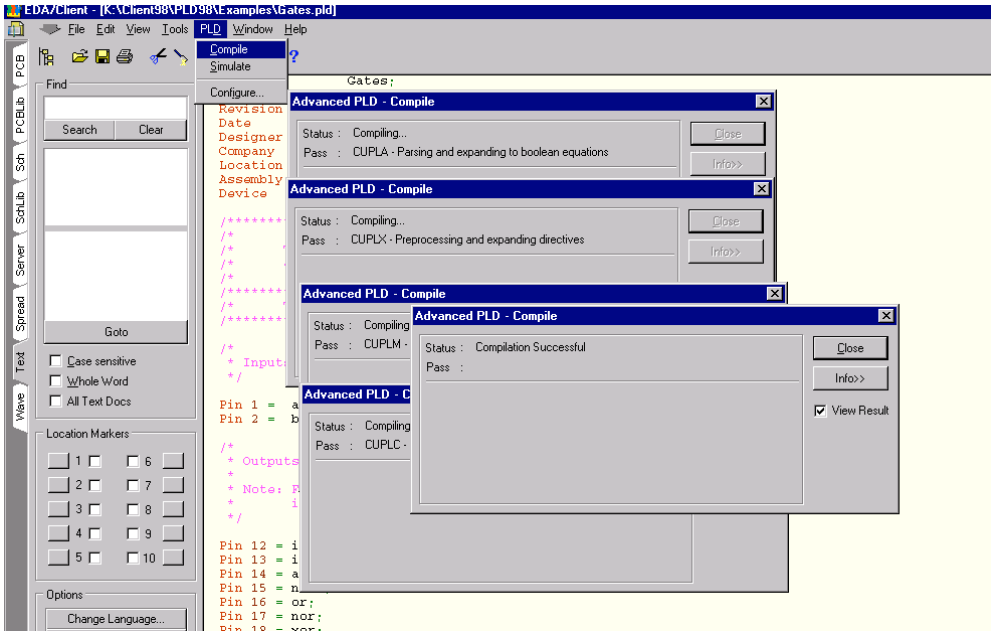


Logic Minimization Levels

Flag	Minimization Description
M1	Quick Minimization
M2	Quine-McCluskey Minimization
M3	Presto Minimization
M4	Expresso Minimization

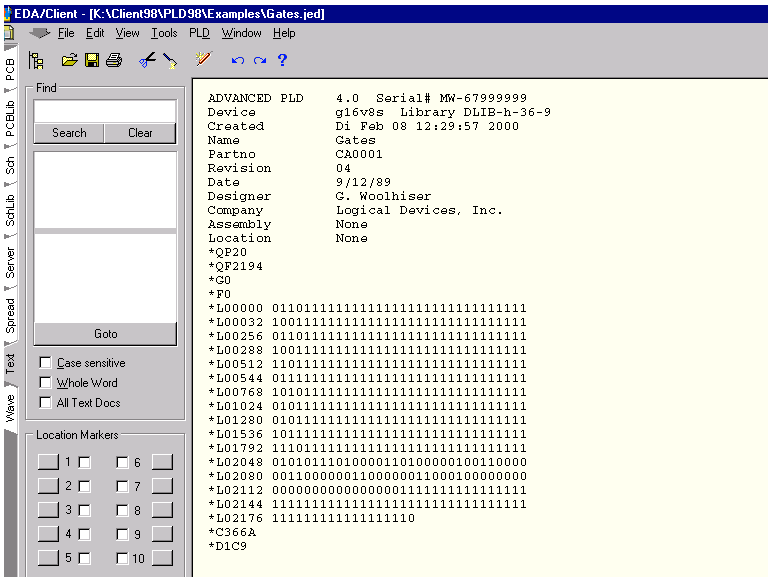
Wirkung und Zeit- / Speicherbedarf der verschiedenen Minimierungsmethoden.

Der Compilerdurchlauf zeigt den Fortschritt über einzelne Messagefenster an. Etwaige Fehler werden nach dem Erkennen mit einer mehr oder weniger aussagekräftigen Fehlermeldung beanstandet.



Informationsfenster zeigen die verschiedenen Durchläufe beim Kompilieren.

Nach erfolgreichem Compilerdurchlauf wird mit *Close* der PLD-Compiler geschlossen und es öffnet sich automatisch ein neues Textfenster mit dem erzeugten JEDEC-Code:



Nach fehlerfreiem Compilerdurchlauf erscheint das Fenster mit dem JEDEC-Code.

Das JEDEC-File kopiert man nun auf eine Floppydiskette und liest es auf dem Rechner beim Programmiergerät im U12 ein.

Programmieren des PLD (ALL07)

Bevor der zu programmierende Baustein eingesetzt wird, ist unbedingt der Bausteintyp und Hersteller einzustellen. Im Access-Menu der Programmersoftware den Eintrag *Device* wählen und Typ einstellen:

PLD-Lattice-GAL16V8D

Ein neues Fenster erscheint und mit *File/Read* das JEDEC-File einlesen. Anschliessend kann der Baustein mit *Program* programmiert werden. Der resultierende Erfolg/Nichterfolg wird angezeigt. Bereits einmal programmierte GALs müssen mit *Erase* gelöscht werden, bevor sie neu programmiert werden können.