

# Advanced PLD 3 ユーザーガイド・言語リファレンス 目次

<b>イントロダクション</b> .....	<b>7</b>
ADVANCED PLD - 統合化 PLD 開発環境 .....	7
Text Expert、テキストエディター .....	7
アドバンスト PLD コンパイラー .....	8
アドバンスト PLD シミュレーター .....	8
シミュレーションの波形の表示 .....	8
CUPL 言語 .....	8
広範囲のデバイスサポート .....	8
レジスタエミュレーション .....	9
ドモルガンの最適化 .....	9
D/T 最適化法 .....	9
データフロー .....	9
システムモジュールの概要 .....	10
Language Preprocessor .....	10
Parser And State Machine Translator .....	11
Device Compatible Checker and DeMorganizer .....	11
Logic Minimizer .....	11
Fitter And Output File Generator .....	11
Simulator .....	13
CUPL.DL .....	14
このガイドの使用法 .....	14
<b>インストレーション</b> .....	<b>15</b>
このガイドについて .....	15
必要なシステム .....	15
最小システム .....	15
推奨システム .....	15
ソフトウェアのインストール .....	15
ソフトウェアを使用可能にする .....	16
ソフトウェアのロックを解除するには .....	16
EDA/CLIENT にアドバンスト PLD を組み込む .....	17
テキストエディターで PLD ツールバーにアクセスするには .....	17
<b>アドバンスト PLD の紹介</b> .....	<b>19</b>
アドバンスト PLD の設定と使用法 .....	19
PLD ツールバー .....	19
アドバンスト PLD コンパイラの設定 .....	20
コンパイラの入力 .....	20
出力フォーマット .....	20
コンパイラの設定 .....	23
ロジック最小化法 .....	24
ブーリアン論理 .....	26

デバイスオプション.....	26
デバイスをコンパイラソースファイルで指定するには.....	26
Target Device ダイアログボックスでデバイスを指定するには.....	27
デバイスライブラリー.....	27
デバイスニーモニック (デバイス名).....	28
LCC と PLCC デバイス.....	28
仮想デバイス (Virtual Device).....	29
アドバンスト PLD によるコンパイル.....	29
アドバンスト PLD によるシミュレーション.....	30
入力.....	30
出力.....	30
シミュレータの起動.....	31
シミュレーション波形の表示.....	32
シミュレーション結果の表示.....	32
Time Base.....	32
Signal Name.....	32
Signal Value.....	33
信号はなぜ違って見えるか?.....	33
タイミングマーク.....	33
バスの作成.....	33
信号の編集.....	33
表示位置の変更.....	34
波形標示 Wave エディターパネル.....	34
<b>PLD の設計プロセス.....</b>	<b>36</b>
ステップ.....	36
コンパイラソースファイルの作成.....	36
CUPL ソースファイルシンタックスの概要.....	37
PLD ソースファイルのコンパイル.....	38
ソースファイルディレクティブ.....	39
言語要素.....	42
言語シンタックス.....	44
簡単なステートマシンの設計.....	51
<b>設計のサンプル.....</b>	<b>53</b>
STEP 1 - 設計作業の確認.....	53
STEP 2 - コンパイラソースファイルの作成.....	55
STEP 3 - 式の公式化.....	57
STEP 4 - ターゲットデバイスの選択.....	58
STEP 5 - ピン割り付け.....	60
STEP 6 - PLD ソースファイルのコンパイル.....	62
STEP 7 - シミュレーションテストベクタファイルの作成.....	69
STEP 8 - デバイスのシミュレーション.....	72
STEP 9 - シミュレーション波形の表示.....	75
概要.....	76

<b>設計例</b> .....	<b>77</b>
例 1 - 簡単なゲート.....	78
例 2 - TTL 設計の PLD への変換.....	85
例 3 - 2 ビットカウンタ.....	90
4 - デカードアップ/ダウンカウンタ.....	93
例 5 - 7 セグメントのディスプレイデコーダ.....	101
例 6 - ロードリセット機能付きの 4 ビットカウンタ.....	104
<b>プログラマブルロジックデバイスの歴史</b> .....	<b>106</b>
イントロダクション.....	106
プログラマブルロジックデバイスの分類.....	108
ジェネリックブーリアンセットインプリメンテーションによる PLD の分類.....	108
PROMs.....	109
PALs.....	110
GALs.....	111
PLAs.....	111
コンプレックス PLD.....	112
FPGAs.....	112
論理アーキテクチャによる PLD の分類.....	113
技術による PLD の分類.....	115
PLD のパッケージング.....	116
論理デバイスのプログラミング.....	117
論理デバイスの機能テスト.....	117
<b>PLD 設計理論</b> .....	<b>118</b>
ブール代数.....	118
ブール代数の仮定と概念.....	118
シーケンシャル回路とステートマシンの設計.....	126
ステートマシンの概念: 2 つの設計例.....	132
論理リダクション: より小さく、より良く.....	137
論理の落とし穴.....	143
ステートマシンの設計.....	146
ステートモデル設計.....	147
ステートマシンシンタクス.....	149
2 ビットカウンタの例.....	151
カウンタの設計.....	160
1 ビットアップカウンタ.....	161
2 ビットカウンタ.....	161
3 ビットカウンタ.....	161
4 ビットカウンタ.....	161
アップ/ダウンを変更できる 4 ビットカウンタ.....	162
アップダウンとロードのできる 4 ビットカウンタ.....	162
ホールド機能の追加.....	163
カウントダウンの実行.....	166
交通官制装置.....	169
ステートマシンのインプリメント.....	169

PLD ソースファイル .....	171
アクティブハイとアクティブロー設計 .....	173
イントロダクション .....	173
アクティブハイ出力 .....	173
アクティブロー出力 .....	175
アクティブロー設計を用いてアクティブハイのシミュレーションを行なう方法 .....	178
<b>CUPL 言語リファレンス .....</b>	<b>180</b>
言語要素 .....	180
変数 .....	180
インデックス付き変数 .....	181
予約語と予約シンボル .....	181
数値 .....	182
コメント .....	183
テンプレートファイル .....	184
ピン宣言命令 .....	187
ブリプロセッサコマンド .....	195
言語シンタックス .....	202
論理演算 .....	202
算術演算 .....	203
算術関数 .....	203
フィードバック拡張子の使用法 .....	207
マルチプレクサ拡張子の使用法 .....	208
拡張子の使用 .....	210
論理式の再検討 .....	227
式 .....	228
論理式 .....	228
APPEND 命令 .....	229
セット演算 .....	230
真理値表 .....	240
フィールド比較演算 "=" .....	241
DECLARE .....	241
PROPERTY .....	242
DEMORGAN .....	243
REGISTER_SELECT .....	244
ステートマシンシンタックス .....	245
ステートマシンモデル .....	245
ステートマシンシンタックス .....	246
条件シンタックス .....	261
ユーザ定義関数 .....	262
<b>設計のシミュレーション .....</b>	<b>265</b>
シミュレータへの入力 .....	265
シミュレータからの出力 .....	265
シミュレーションテストベクタファイルの作成 .....	266
ヘッダー情報 .....	266

デバイスの指定.....	267
コメント .....	267
ステートメント.....	268
ORDER ステートメント.....	268
BASE ステートメント.....	271
VECTORS 命令.....	272
プリロード.....	274
クロック.....	275
非同期ベクタ.....	275
I/O ピンシミュレーション .....	276
変数宣言 (VAR) .....	277
シミュレータディレクティブ .....	278
\$MSG .....	278
\$REPEAT.....	278
\$TRACE.....	279
\$EXIT .....	280
\$SIMOFF.....	280
\$SIMON .....	281
アドバンストシンタクス .....	281
割り付け命令(\$SET) .....	281
算術及び論理演算子(\$COMP).....	281
テストベクタの生成(\$OUT).....	282
条件シミュレーション(\$IF) .....	283
繰り返し命令 .....	283
FOR 命令 .....	283
WHILE 命令.....	284
DO..UNTIL 命令 .....	284
MACRO ステートメントと CALL ステートメント .....	284
仮想シミュレーション .....	290
欠陥のシミュレーション.....	290
<b>デバイスリスト (JAN 96).....</b>	<b>291</b>
ACTEL PLD 2.....	291
ALTERA PLD 42.....	291
AMD/MMI PLD 293.....	291
AMD/MMI PROM 49.....	294
AMI/GOULD PLD 11 .....	294
ATMEL PLD 14.....	294
CYPRESS EPROM 5.....	294
CYPRESS PLD 71 .....	294
EXEL PLD 1 .....	295
FAIRCHILD PLD 14.....	295
HARRIS PLD 14.....	295
HARRIS PROM 46 .....	295
ICT PLD 15.....	296
INTEL PLD 20 .....	296
LATTICE PLD 83.....	296

MOTOROLA PLD 1.....	297
NATIONAL PLD 121.....	297
NATIONAL PROM 39.....	298
PHILIPS PLD 60.....	298
PHILIPS PROM 17.....	298
PLESSY PLD 1.....	299
PLUS LOGIC PLD 2.....	299
PLX TECH. PLD 2.....	299
QUICKLOGIC PLD 4.....	299
RICOH PLD 14.....	299
SAMSUNG PLD 18.....	299
SEEQ PLD 2.....	299
SGS-THOM. PLD 48.....	299
SIGNETICS.....	300
SPRAGUE PLD 6.....	300
TI PLD 88.....	300
TI PROM 18.....	301
TOSHIBA PLD 2.....	301
TRIQUINT PLD 4.....	301
VLSI PLD 10.....	301
XILINX PLD 30.....	301
<b>ファイルフォーマット.....</b>	<b>303</b>
ダウンロードフォーマット.....	303
JEDEC フォーマット.....	303
ASCII-Hex フォーマット.....	306
HL フォーマット.....	306
ドキュメンテーションファイルフォーマット.....	314
PDF ファイルフォーマット.....	320
バークレイ PLA ファイルフォーマット.....	321

# イントロダクション

アドバンスト PLD はプログラマブルロジックデバイス用のロジックを作成するために使用される統合化された、多くの機能を持った強力な開発環境です。

## Advanced PLD - 統合化 PLD 開発環境

アドバンスト PLD により、PLD 設計に統合化の新しい段階がもたらされます。EDA/Client にアドバンスト PLD をインストールすれば、ハードウェア記述言語 CUPL による論理記述ができるようになります。

コンパイルのボタンを押し、アドバンスト PLD の統合化されたエラーレポートを使用しエラーを修正して下さい。コンパイラは、業界標準の JEDEC プログラムファイルを作成し、デバイスプログラマーにダウンロードできます。

アドバンスト PLD により、今日使用できる最大のデバイスライブラリの一つが提供されます。このライブラリは、主な製造元からのプログラマブルロジックはすべてサポートされています。アドバンスト PLD の CUPL 言語は、製造とは独立しておりエンジニアは設計とパッケージングを自由に行なうことができます。

アドバンスト PLD シミュレータを使用して、設計を実装するまえにシミュレーションを実行して下さい。そして、シミュレーションの結果をアドバンスト PLD に付属の Waveform エディタを使用して検証して下さい。

アドバンスト PLD パッケージには、PLD 設計用に特別に仕立てられている 3 種類の EDA/クライアントサーバーがあります。TextExpert：文法チェック用のテキストエディタ、PLD：設計をコンパイルしたりシミュレーションを実行します。Wave：シミュレーションの波形を表示します。

### Text Expert、テキストエディター

ロジックやシミュレーションのリストファイルを記述するために、アドバンスト PLD の文法チェック用のテキストエディタの TextExpert を使用して下さい。アドバンスト PLD には、統合化されたエラーレポート機能が組み込まれています。コンパイルやシミュレーション中にエラーが見つかった、自動的にソースファイルの中でその時のエラーの状態に応じてハイライト表示されます。

カットやコピー、貼り付け、検索、再配置など通常のテキスト編集機能に加えて、Text Expert には、Syntax Highlighting と呼ばれる機能があります。Syntax Highlighting 機能により文法に基づいて異なるワードタイプやシンボル、識別子をそれぞれの色でハイライト表示します。この機能によりドキュメントの編集が便利になります。特に PLD 論理記述ファイルなどで繰り返し作業を行なう時などに非常に便利です。

## アドバンストPLD コンパイラー

アドバンスト PLD コンパイラには、4 段階の最小化でプログラマブル論理回路を最小化する高速で強力なミニマイザがあります。コンパイラは、分配属性やドモルガンの定理を用いてブーリアン表現で簡潔に表現されます。

コンパイラは、業界標準の JEDEC ファイルを作成します。このファイルは、JEDEC フォーマットをサポートする論理プログラマーと互換性のあるダウンロードファイルフォーマットです。アドバンスト PLD は、主な製造元からのプログラマブルロジックはすべてサポートしており、設計とパッケージングを自由に行なうことができます。

## アドバンストPLD シミュレーター

アドバンスト PLD シミュレータを使用して、論理回路が実装される前にシミュレーションを実行して下さい。入力と出力から PLD の機能を予測した記述をシミュレーションファイルに作成します。シミュレータは、予想される値とコンパイラの動作中に計算される実際の値とを比較します。

シミュレータを使用すると、PLD をプログラムする前にロジックを試すことができます。この機能は、デバイスの損傷を防止したりシステムレベルの問題をデバッグする時に役立ちます。シミュレータにより確認されたテストベクタは、ロジックプログラマーにダウンロードされます。

## シミュレーションの波形の表示

シミュレーションの結果は、アドバンスト PLD 波形エディタにより検証することができます。このエディタは、シミュレーションリスティングファイル（ファイル名.SO）を読み込み、スプレッドシートスタイルで波形をエディタウィンドウに表示します。

## CUPL 言語

アドバンスト PLD は、CUPL ハードウェア記述言語を使用します。このガイドの中では、この言語は、CUPL または CUPL 言語と表わします。

設計時間を節約するために、CUPL 言語には、リストやアドレス範囲、ビットフィールドのための式や表記が表現が用意されています。Truth テーブルシンタックスにより、ある種の論理記述を簡潔に表現することができます。

CUPL 言語のリファレンスのセクションは、このマニュアルにあります。

## 広範囲のデバイスサポート

アドバンスト PLD は、プログラマブルロジックの主な製造元のデバイスをすべてサポートしています。また、アドバンスト PLD は 2 つの優位性を持っています。まず一つは、一つの開発環境と一つの言語を習得するだけで使用できることです。アドバンスト PLD を使用すると PAL16L8 を用いて簡単なアドレスデコーダを設計したり、現在お手持ちの設計やパッケージを Xilinx5000 シリーズのデバイスを用いて開発したりできます。2 つめは、同じ機能のロジックを違う部品にパッケージすることができということで



す。これにより、デバイスの製造元の選択に自由度が増します。

## レジスタエミュレーション

この機能により、目的のアーキテクチャを持ったレジスタによらずあらゆる種類のレジスタを使用できます。例えば、このような機能を利用して、JK レジスタで設計してターゲットデバイスにはD レジスタしかないデバイスを使用することが可能になります。

## ドモルガンの最適化

プログラマブル極性を持つデバイスでは、式にドモルガンの定理を適用して回路の効率を上げることができます。設計者は、コンパイラはドモルガンの定理を適用するのに最適であるとしたところに定理を適用させることができます。

## D/T 最適化法

D レジスタとTレジスタの両方を持つデバイスでは、あるレジスタタイプを他のレジスタへ変更することで使用されるプロダクトタームの数を減らすことができます。コンパイラは、どのレジスタを使用すればより最適かを決定し自動的に配置を実行します。

## データフロー

このセクションでは、アドバンスド PLD のデータフローについて説明します。

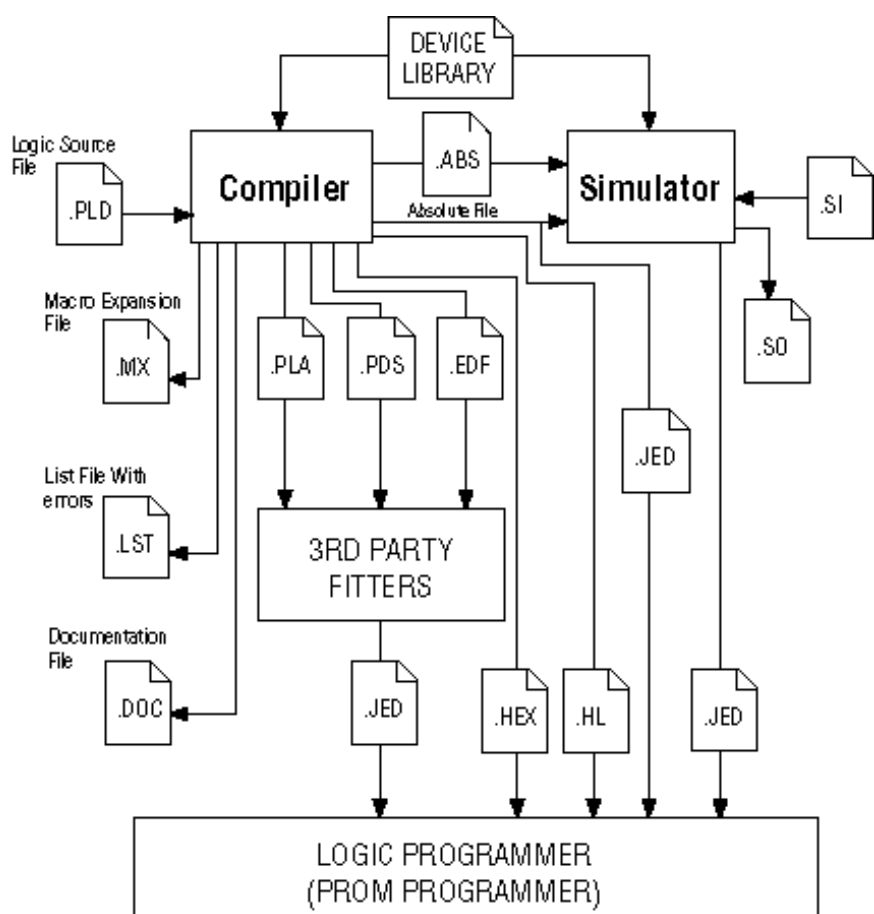
最初に、CUPL 言語を使用して論理記述のソースファイルを作成します。このファイルには、プログラマブルロジックデバイスに割り付けられるロジックが記述されます。

アドバンスド PLD によりソースファイルをコンパイルしデバイスプログラマーへダウンロードするヒューズマップファイルを作成します。コンパイラは、Configure Advanced PLD ダイアログボックスの Absolute ABS オプションをイネーブルにすることでシミュレータで使用される .ABS ファイルを作成します。

そしてアドバンスド PLD シミュレータにより、デバイスのシミュレーションを実行します。シミュレータは、テスト仕様ファイル (.SI) を必要とし論理回路を検証します。シミュレータにより、テスト仕様ファイルで予測される値とコンパイラにより作成されたアブソリュートファイルの実際の値とを比較します。シミュレーションにエラーがなく完了すると、確認済みのテストベクタが、コンパイラにより生成されたダウンロードファイルに追加されます。

この時点で、確認済みのヒューズマップファイルがデバイスプログラマにわたされます。

図 1-1 にアドバンスド PLD のデータフローを示します。



©1994 DataRom

図 1-1 アドバンスド PLD のデータフロー

拡張子	ファイルタイプ
.PLD	論理記述設計ファイル
.SI	シミュレータリスティングファイル（入力）
.SO	エラーの記述されたシミュレータ出力ファイル
.HL	HL ダウンロードファイル
.HEX	Hex ダウンロードファイル
.JED	テストベクタの記述されていない JEDEC ファイル
	テストベクタの記述された JEDEC ファイル

表 9-1 アドバンスド PLD のファイル拡張子

## システムモジュールの概要

アドバンスド PLD コンパイラとシミュレータでは以下のモジュールが使用されます。

### Language Preprocessor

CUPLX モジュールは.PLD (入力) ファイルを検索し、\$DEFINE などのプリプロセッサディレクティブを処理します。このモジュールはすべてのプリプロセッサディレクティブが展開された中間ファイルを作成します。

### **Parser And State Machine Translator**

CUPLA モジュールは CUPLX により生成された中間ファイルを検索し、シンボルテーブルを作成し式を展開します。また、このモジュールは、ステートマシンや真理値表、ユーザ定義関数をブーリアン式に展開します。そして、レンジ命令を処理中にロジックを簡潔にする作業を実行します。

### **Device Compatible Checker and DeMorganizer**

CUPLB モジュールは、デバイスライブラリ(.DL) ファイルの使用ピンアウトに対して PLD ソースファイルで選択されたピンアウトを検証します。また、デバイスのアーキテクチャに基づいて、ソースファイル中の変数が正しく使用されているかを確認します。CUPLB は通常、DeMorganizer として参照されます。すなわち、デバイスのピンの極性と変数の極性とが合っていない場合、自動的にドモルガンの定理が実行されます。これによりアクティブローのデバイスでアクティブハイを実現できます。その逆も同様です。また、コンパイルのこの段階で、最小化の最初の段階が実行されます。最小化により 0 や 1、冗長なターム、他のプロダクトタームに含まれるプロダクトタームが削除されます。CUPLB は、デバイスモデルのリンクや論理設計を表わすビットマップを含む最終的なシンボルテーブルを作成します。

### **Logic Minimizer**

CUPLM モジュールは CUPLB で生成されたロジックのビットマップ表現に論理最小化アルゴリズムを実行します。このモジュールは、最小化が必要な式にだけ実行されます。

ミニマイザは、ソースファイルの式で利用できる最小化アルゴリズムを実行します。最小化アルゴリズムには Quick や Quick-McCluskey、Presto、Espresso があります。PAL アーキテクチャに最適のアルゴリズムは Quine-McCluskey です。そして、PLA アーキテクチャに最適のアルゴリズムは、Espresso (プロダクトタームの分配に強い) です。一般に、Espresso は処理速度が速く Quine-McCluskey とほとんど同じ結果になります。

### **Fitter And Output File Generator**

CUPLC モジュールは、論理回路をデバイスに合わせるモジュールです。このモジュールはビルト - インアルゴリズムまたはサードパーティのフィッタを使用してデバイスのフィットを実行します。CUPLC は、複雑なデバイスのマイクロセルやマルチプレクサを操作してブーリアンロジックをデバイスへフィットさせます。回路がデバイスにフィットできない場合、CUPLC は、回路をフィットできない理由が示されたエラーメッセージを表示します。また、このモジュールは Configure AdvancedPLD ダイアログボックスで指定されるオプションで示されるファイルをすべて作成します。

CUPLC により、回路がターゲットデバイスアーキテクチャにフィットする

かやヒューズマップを作成するかどうかが決まります。ヒューズマップやシンボルテーブルは、ドキュメントや JEDEC ファイルを作成するために使用されます。

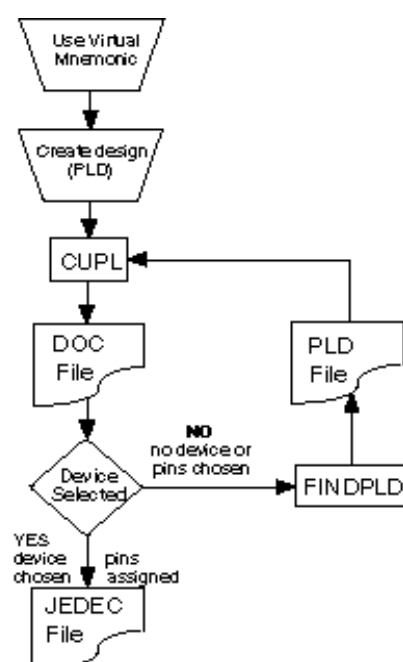


図 1-3 デバイスに独立した設計フロー

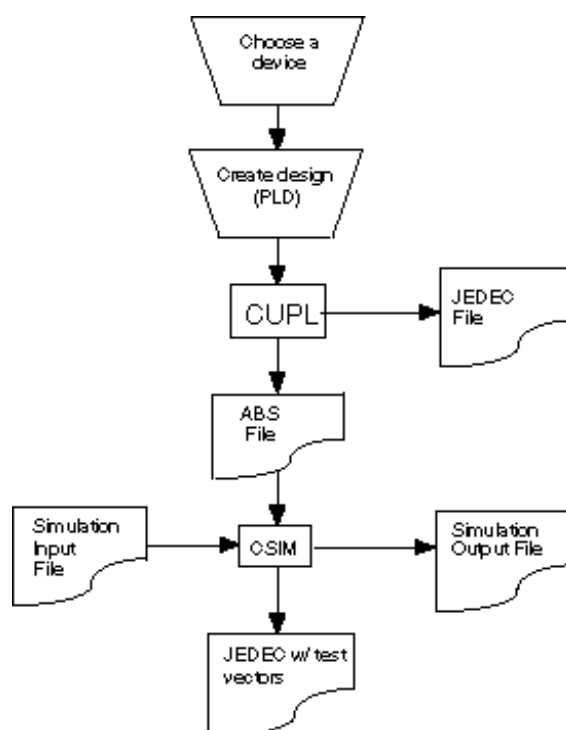


図 1-4 デバイスを特定した設計フロー

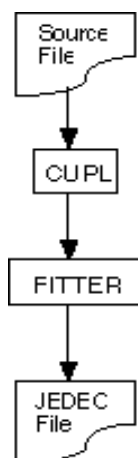


図 1-5 CUPL デバイスフィッティング

外部のフィッターによりサポートされるデバイスでは、アドバンスド PLD は自動的に適当なデバイスフィッターを呼び出します。これらの外部のフィッターは DOS ウィンドウで実行され、フィッターが実行を完了すると閉じられます。DOS ウィンドウが自動的に閉じられないようにしたり、DOS ウィンドウへのメモリの割り付けを変更したい場合、Compile プロセスの PLD のパラメータの設定を変更して下さい。Compile プロセスやパラメータに関する詳細は PLD リファレンスのヘルプファイルを参照して下さい。

現在使用できるフィッタープログラムを以下に示します。

- Generic CUPLC device fitter
- Altera Max
- AMD MACH device fitter
- Atmel High Density EPLDs
- Cypress
- ICT EPLD/FPGA's
- Intel FLEX
- Lattice
- Motorola
- National MAPL device fitter
- Philips PML
- Xilinx EPLDs

### Simulator

シミュレータは、実際のデバイスで回路がどのように動作するかに影響を与えるユニットの遅れをシミュレーションします。あるデバイスの特異性や内部遅れはシミュレーションされません。しかし、このシミュレータは、

デバイスの特徴を表現し、実装に関して何も気にしないで実装できます。

## **CUPL.DL**

デバイスアーキテクチャ情報はすべてデバイスライブラリ CUPL.DL にあります。このライブラリは、ソースファイルのコンパイル中にコンパイラにより使用されます。デバイスライブラリの情報には、拡張できるデバイスアーキテクチャ情報やシミュレーションに必要な情報が含まれます。

## **このガイドの使用方法**

以下に、このガイドの中でアドバンスド PLD のタスクを実行するために必要な情報を識別する方法を説明します。

CAPITALS	ディレクトリまたはファイル名を示します。
Initial Capital	メニューアイテム名やプロセス名、ツール名、ダイアログボックス名、ダイアログボックスオプション名を示します。  ハイライト表示されたワーニングは特別なアドバイス情報を表わすために使用されます。

# インストール

## このガイドについて

このオンラインヘルプではユーザに関して 3 つの仮定をしています。

ユーザは、電気回路設計の原理や用語、シンボルについて精通していること。基本的に、Protel EDA ツールやドキュメントでは標準の電気工学の原理や用語を使用しています。

また、Microsoft Windows のアイコンやメニュー、ウィンドウ、マウスの使用法に精通していることを前提に書かれています。また、Windows がどのようにアプリケーション（プログラムやユーティリティ）やドキュメント（データファイル）を管理し、アプリケーションの実行やドキュメントのオープン、データの保存などのルーチンタスクを行なっているか基本的な理解があることを仮定しています。Windows に不慣れである場合、Microsoft Windows Users Guide を始めに読んで下さい。

これにより、Microsoft DOS やディレクトリの使用法、ファイルの名前の付けかたの慣習などがわかるようになります。

## 必要なシステム

### 最小システム

- IBM PC 互換機で動作する Microsoft Windows 3.1
- i486 プロセッサ
- 16MB の RAM
- 解像度 800 x 600、16 色以上標示可能なディスプレイ
- 33.5MB のハードディスクの空き容量

### 推奨システム

- Pentium プロセッサ
- 16MB 以上の RAM
- 解像度 1024x768 以上、256 色標示可能なディスプレイ
- 50MB のハードディスクの空き容量
- ソフトウェアを使用できるようにするには

## ソフトウェアのインストール

Protel のソフトウェアをインストールするには Windows のプログラムマネ

ージャーのアイコン - ファイル名を指定して実行を選択します。  
(Windows95、NT4.0 以降の場合はスタートボタン - ファイル名を指定して  
実行 を選択します。)ダイアログボックスで以下の様に記入します。

<ドライブ名>:¥setup

<ドライブ名>は CD-ROM のドライブを指定します。

## ソフトウェアを使用可能にする

アドバンスド PLD をインストールしたら、アクセスキーコードを入力して  
パッケージの機能をすべてイネーブルにしてください。Protel ソフトウェアを  
インストールしてアクセスキーコードを入力しないと、ソフトウェアはデ  
モモードでしか動作しません。従って、設計ファイルは保存されません。

ハードロックが必要な場合、ソフトウェアと一緒にハードロックが  
添付されています。ハードロックが必要かどうかは Protel の販売代理  
店に確認して下さい。

ハードロックが必要な場合、コンピュータの平行（プリンタ）ポート  
にインストールして下さい。ハードロックを接続する前にコンピュータの  
電源を切して下さい。ハードロックが正しく接続されているか確認して下  
さい。プリンタが平行ポートに接続されている場合、プリンタケーブ  
ルをハードロックの反対側に接続して下さい。ハードロックはどんなパラ  
レルポートでも接続できます。ソフトウェアは自動的にハードロックを確認  
します。

(日本国内ではハードロックキーバージョンは販売しておりません。)

## ソフトウェアのロックを解除するには

ソフトウェアのいろいろな機能のロックを解除するには適当なアクセスキ  
ーコードを入力して下さい。

Help-About メニューアイテムを選択し Set Access Codes ボタンを押して  
Security Locks ダイアログボックスを表示して下さい。現在使用可能な機能  
がすべてダイアログボックスの Locks セクションに一覧表示されます。ア  
クセスキーコードが入力されない場合、その機能はロックされます。

ある機能のロックを解除するには、その機能を選択し Un-Lock ボタンを押  
して下さい。その機能の Lock ダイアログボックスがポップアップ表示され  
ます。その機能のアクセスキーコードを入力し Test ボタンを押して下さい。  
アクセスキーコードが正しく入力されると、アクセス権が変更されロック  
が解除されたことを示します。OK をクリックすると Locks ダイアログボッ  
クスが閉じられます。機能のロックが正しく解除されると、その機能のド  
アアイコンが開いたドアになります。

同様の作業を繰り返し、アクセスキーコードを持っている機能のロックを  
解除して下さい。



## EDA/Client にアドバンスドPLD を組み込む

アドバンスド PLD は EDA/クライアント環境で動作します。EDA/クライアントを実行しサーバーのインストールや PLD 設計のコンパイルが可能な環境を整備して下さい。

アドバンスド PLD には、2 つのサーバーがあります。まず、PLD サーバーは、設計した論理回路をコンパイルしたりシミュレーションを実行したりします。Wave サーバーは、シミュレーションの結果を検証するために使用する波形エディタです。

インストールプロセスを実行するとサーバーは自動的にインストールされます。以下の説明は、将来手動でサーバーをインストールする必要がでたときのためのものです。

サーバーをインストールするには

- Client メニュー（下向き矢印）へ行き、Server メニューアイテムを選択して下さい。EDA Server ダイアログボックスがポップアップ表示され、現在インストールされているサーバーが表示されます。
- 新しいサーバーをインストールするには、Install ボタンを押して下さい。EDA/Client Server Install ダイアログボックスがポップアップ表示されます。
- サーバーインストールファイル PLD.INS を選択して下さい。
- OK をクリックしてサーバーをインストールして下さい。NotStarted の状態です。サーバーが NotStarted の状態の場合、サーバーはメモリを消費していません。アドバンスド PLD をすぐに起動しない場合、使用する最初の時に自動的に開始されます。
- インストールプロセスを TextExpert と Wave の 2 つのサーバーについても繰返して下さい。これらのサーバーのインストールファイルは Textedit.INS と Wave.INS です。

## テキストエディターでPLD ツールバーにアクセスするには

PLD ロジック記述ファイルは、EDA/クライアントにより提供されるテキスト編集用のサーバーの TextExpert を使用して記述されます。

アドバンスド PLD を使用してコンパイルやシミュレーションを行なうには Tools メニューの PLD オプションか PldTools ツールバーを使用して下さい。このツールバーには、PLD 設計回路を配置したりコンパイルやシミュレーションを行なうボタンがあります。

インストールプロセスにより、TextExpert で使用できる PldTools ツールバーが自動的に作成されます。以下の説明は、将来手動でサーバーをインストールする必要がでたときのためのものです。

PLD ツールバーをテキストエディターでアクセスできるようにするには

- Client メニューへ行き Servers メニューアイテムを選択して下さい。

EDA Server ダイアログボックスがポップアップ表示されます。

- EDA Servers ダイアログボックスで、TextEdit を選択し Configure ボタンを押して下さい。Configure Server ダイアログボックスが表示されます。
- Configure Server ダイアログボックスで、Toolbars ボタンを押して下さい。Resource List Editor ダイアログボックスが表示されます。
- ツールバーにアクセスするには、PldTools を Resources List から選択し Add ボタンを押して下さい。PldTools が Current Available リストに表示されます。
- ダイアログボックスをすべて閉じると、PldTools ツールバーが TextExpert に表示されます。
- サーバーに対してツールバーが使用できるようになると、その表示状態は Customize Resources ダイアログボックスで切替えることができます。Client メニュー - Configure...メニューアイテムを選択し、このダイアログボックスを表示して下さい。

## アドバンストPLD の紹介

このセクションでは、アドバンスト PLD のプログラマブルロジック設計のプロセスを紹介します。まず、アドバンスト PLD の各コンポーネントを紹介し、コンパイラをどのように配置しているかとか、どのように論理記述をコンパイルするか、また、どのようにデバイスのシミュレーションを行ないシミュレーション結果を検証するかを簡単に説明します。

アドバンスト PLD のコンポーネントには以下のものがあります。

- TextExpert (文法チェック機能付きのテキストエディタ)
- アドバンスト PLD ロジックコンパイラ
- アドバンスト PLD デバイスシミュレータ
- 波形編集用 Wave エディター

PLD 開発サイクルの第一段階は、論理記述ソースファイルを作成することです。次にソースファイルは、コンパイルされ実際のデバイスをプログラムするために使用される JEDEC ファイルが作成されます。

コンパイラが処理を始めると、必要な出力ファイルやターゲットデバイスなどの情報が必要になります。このような情報は、Configure Advanced PLD ダイアログボックスで入力されます。

## アドバンストPLD の設定と使用法

プログラマブルロジックの設計の第一段階は、論理記述言語 CUPL で書かれた論理記述ソースファイルを作成することです。EDA/クライアントには、文法チェック機能の付いた ASCII エディタの TextExpert があります。TextExpert を使用して論理記述ソースファイルを作成し、PldTools ツールバーからコンパイラとシミュレータを起動して下さい。

### PLD ツールバー

PLD ツールバーを使用してコンパイラの設定や起動、シミュレータの起動を行なって下さい。PldTools ツールバーには以下のボタンがあります。

- Compile - アドバンスト PLD コンパイラを起動します。これにより、CUPL 言語の論理記述ソースファイルがコンパイルされます。
- Simulate - アドバンスト PLD シミュレータを起動します。このシミュレータは、ユーザが作成したテスト仕様ソースファイルの予想されるテスト値をコンパイル中の論理式から作成される実際の値とを比較します。
- Configure - Configure Advanced PLD ダイアログボックスをオープンします。このダイアログボックスでコンパイラの動作や出力の設定を行います。



TextExpert の中でツールバーが使用できない場合、このガイドのインストレーションセクションを参照して下さい。Configure は Tools メニューの Compile や Simulate プロセスからも起動できます。

## アドバンストPLD コンパイラの設定

このセクションでは、コンパイラの入力とアドバンスト PLD の設定法について説明します。

### コンパイラの入力

論理記述ソースファイル（ファイル名.PLD）はコンパイラへの入力です。このファイルには、ターゲットデバイスで実行したい論理関数が記述されています。ソースファイルは、TextExpert を使用して作成します。

論理記述ソースファイルの作成に関する詳細は、このガイドの PLD 設計プロセスにおける PLD ソースファイルの作成のセクションを参照して下さい。

### 出力フォーマット

アドバンスト PLD は 4 種類の出力ファイルを生成することができます。その 4 種類を以下に示します。

- ダウンロードフォーマット、プログラマへダウンロードするフォーマットです。
- 出力フォーマット、サードパーティのフィッタやエラーリスティングのインタフェース用のフォーマットです。
- ドキュメンテーションフォーマット、ヒューズプロットや式があります。
- シミュレーションフォーマット、仮想デバイスのプログラムです。シミュレータによりテスト仕様ソースファイル（ファイル名.SI）からテストベクタがこの仮想デバイスへ適用されシミュレータリスティングファイル（ファイル名.SO）に結果が記録されます。

コンパイラ出力フォーマットを設定するには、Configure Advanced PLD ダイアログボックスの Output Format タブのチェックボックスを選択して下さい。

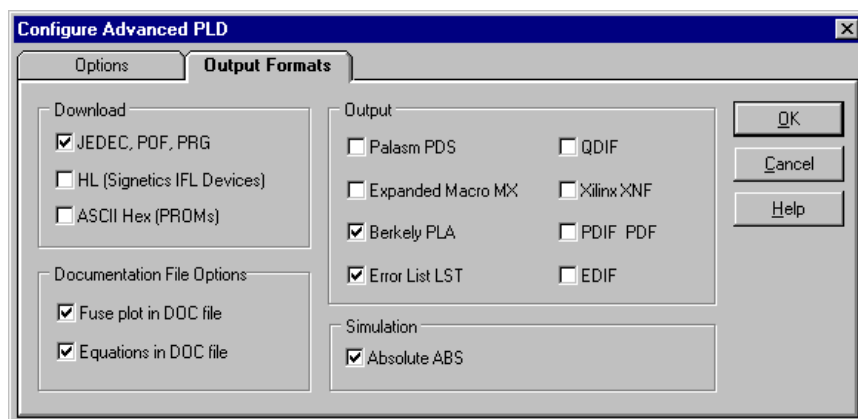


図 3-1 Configure Advanced PLD ダイアログボックス Output Format タブ

表 3-1 個々の出力フォーマットについて

出力フォーマット	説明
JEDEC, POF, PRG	JEDEC 互換の ASCII ダウンロードファイル( ファイル名.JED )を作成します。ファイル名は、コンパイラへの論理記述入力ファイルと異なる名前でも問題ありません。論理記述ファイルのヘッダー情報セクションの NAME ステートメントで、ダウンロードファイルの名前を決定します。
HL	HL ダウンロードファイル( ファイル名.HL )を作成します。このフォーマットは、Signetics IFL デバイスで使用されます。ファイル名は、コンパイラへの論理記述入力ファイルと異なる名前でも問題ありません。論理記述ファイルのヘッダー情報セクションの NAME ステートメントで、ダウンロードファイルの名前を決定します。
ASCII Hex	ASCII-Hex のダウンロードファイル( ファイル名.HEX )を作成します。このフォーマットは、PROMs で使用されます。ファイル名は、コンパイラへの論理記述入力ファイルと異なる名前でも問題ありません。論理記述ファイルのヘッダー情報セクションの NAME ステートメントで、ダウンロードファイルの名前を決定します。
PALASM PDS	PAL ハンドブック( 第 3 版 )の Monolithic Memories による標準設定で、PALASM フォーマットファイル( ファイル名.PDS )を作成します。
Expanded Macro MX	ソースファイルで使用されるマクロがすべて記述されている拡張マクロ定義ファイル( ファイル名.MX )を作成します。REPEAT コマンドを使用する拡張表現も含まれます。

Berkely PLA	PLEASURE やその他の Berkely PLA フォーマットを使用する PLA レイアウトツールなどの Berkely PLA ツールにより使用される Berkely PLA ファイル (ファイル名.PLA) を作成します。
Error list LST	エラーリスティングファイル (ファイル名.PLA) を作成します。元のソースファイルの各行には番号が付けられます。エラーメッセージはファイルの最後に一覧表示され、参照のために行番号が使用されます。
QDIF	QuickLogic デバイス用の QDIF フォーマットファイルです。QuickLogic オプティマイザに対応します。QuickLogic の sPDE ツールを使用して最適化された回路の配置や配線を行います。
Xilinx XNF	他の論理設計ツールや XILINX の PDS2XNF などのゲートアレイフィッタ用の入力ファイルを作成します。
PDIF PDF	PDIFIN プログラムにより、PC-CAPS (P-CAD Schematic Capture) プログラムのシンボルに変換される PDIF (P-CAD Database Interchange Format) ファイル (ファイル名.PDF) を作成します。作成されたシンボルには、PLD のパッケージング情報が含まれます。
EDIF	EDIF フォーマットファイルを作成します。
Equations in DOC File	Sum-of-products フォーマットの論理タームの拡張リストやソースファイルで使用されるすべての変数のシンボルテーブルを含むドキュメンテーションファイル (ファイル名.DOC) を作成します。プロダクトタームの総数や各出力で使用できる値が含まれます。
Fuse Plot in DOC File	ドキュメンテーションファイルの中のヒューズプロットを作成します。PAL デバイスでは、各出力ピンは一覧表示され、関連付けられたプロダクトターム列が JEDEC ヒューズ番号を先頭にして表示されます。現在あるヒューズは x で表わされます。切れたヒューズは、- で表わされます。IFL デバイスでは、HL ダウンロードフォーマットが使用され、H、L、0、- で表わされる入力タームで JEDEC ヒューズ番号が表わされます。
Absolute ABS	アドバンスド PLD 論理シミュレータで使用するアブソリュートファイル (ファイル名.ABS) を作成します。

## コンパイラの設定

Configure Advanced PLD ダイアログボックスの Option Tab により以下の設定を行なうことができます。

- ターゲットデバイスの選択
- 最適化オプションを ON にする
- コンパイルオプションを ON にする
- ロジック最小化法の設定

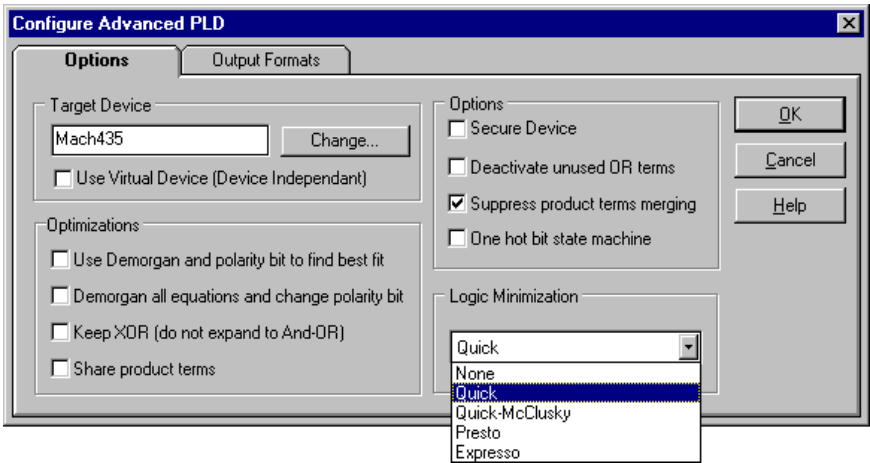


図 3-2 Configue Advanced PLD ダイアログボックス Options タブ

表 3-2 コンパイラーの設定について

設定項目	説明
Use DeMorgan and polarity bit	ピンやピンノード変数のプロダクトタームの使用を最適化します。DEMORGAN ステートメントがソースファイルにある場合、オーバーライドされます。
DeMorgan all equations	ピンやピンノード変数のすべてにドモルガンの定理を適用します。DEMORGAN ステートメントがソースファイルにある場合、オーバーライドされます。
Keep XOR	XOR を AND-OR の式に展開しません。デバイスに独立の設計やフィッタが XOR ゲートをサポートしているデバイスの設計に使用します。
Share Product Terms	最小化を実行中にプロダクトタームを強制的にシェアリングします。これによりグループの削減として参照されます。
Secure Device	The secure device option adds the necessary code in the JEDEC download file to

	<p>automatically allow the device programmer to blow the security fuse when programming. This makes the device non-erasable. Not all programmers support this option.</p>
Deactivate unused OR terms	<p>IFL デバイスでは、OR ゲート出力アレイは AND ゲートプロダクトタームで駆動されます。通常、使用されていない OR ゲートの入力、新しいターム追加されても良いようにプロダクトタームアレイと接続されたままになっています。しかし、このオプションを使用すると、使用されていない OR ゲートの入力、プロダクトタームアレイから取り除かれ (deactivated) ます。その結果、入力から出力への伝達遅れが減少します。</p>
Suppress product terms merging	<p>IFL デバイスでは、AND ゲートからの各プロダクトタームは、数個の OR ゲート出力に分配されます。このオプションによりこの機能が働かなくなり、必要な時にそれぞれのプロダクトタームは、それぞれの出力 OR アレイで作成されます。その結果、入力から出力の伝達遅れが減少します。Quine-McCluskey 最小化法か Espresso 最小化法が選択されると、このオプションによりそれぞれの個別の出力毎に (出力すべてに対して最小化が実行されるのと反対に) 最小化が実行されます。</p>
One-hot-bit State Machines	<p>FPGA を設計する人向けのオプションです。このオプションを使用するとコンパイラはステートマシンイミュレーションを one-hot-bit で作成します。これにより、XILINX デバイスなどのレジスタ - リッチなアーキテクチャでは明確な優位性が確認できます。ファニングが減少し、配線作業が簡単になります。そして、レジスタからレジスタへのフィードバックパスの長さの違いにより発生するタイミング問題が無くなります。このオプションをオン / オフにしてコンパイルを実行し違いを確認して下さい。現在のところ、オプションが使用されると、コンパイラは回路中のステートマシンをすべて one-hot-bit として扱います。</p>

## ロジック最小化法

最小化法はコンパイラにより使用されるロジックを小さくするアルゴリズム



ムです。

4 つの最小化法を使用できます。

- Quick
- Quine-McCluskey
- Presto
- Espresso

Quine-McCluskey 法や Presto 法は IFL デバイスでは、複数の出力の最小化を実行します。これにより、この種のデバイスではプロダクトタームのシェアリング ProductTermSharing を最大にすることができます。

None を ON にすると、コンパイル中の回路の最小化は実行されません。これは PROMs を使用する場合、その中のプロダクトタームが削除されるのを防ぐのに便利です。

図 3-3 に 4 種類の最小化のメモリ消費や速度、効率の比較を示します。

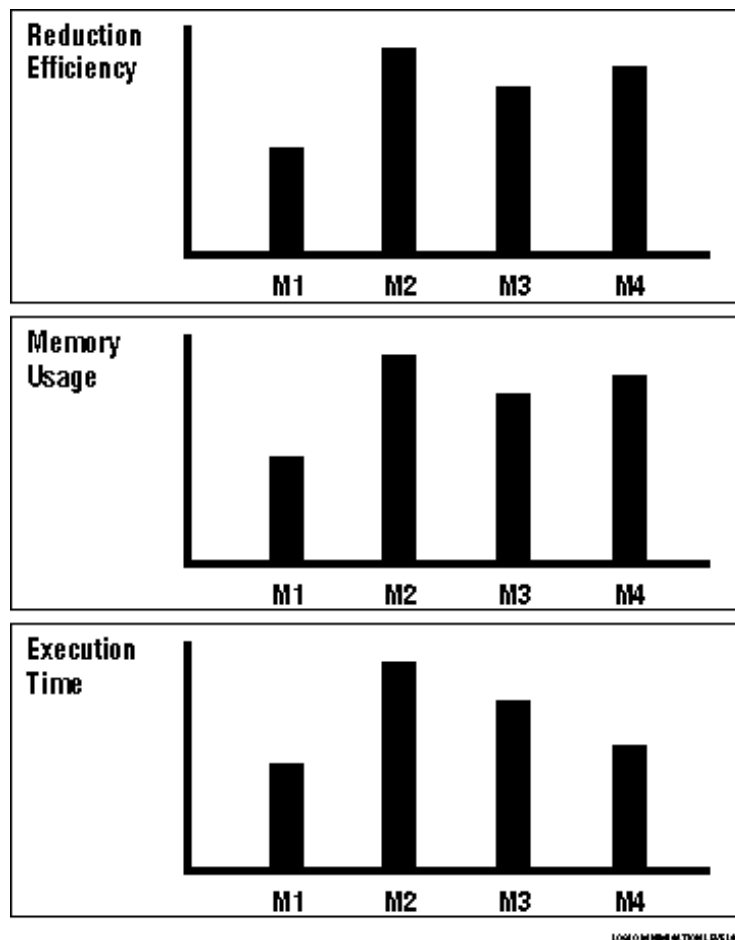


図 3-3 ロジック最小化法の比較

フラグ	最小化法
M1	Quick Minimization
M2	Quine-McCluskey Minimization
M3	Presto Minimization
M4	Expresso Minimization

図 3-3 へのキー

## ブーリアン論理

表 3-3 は、展開された式から不要なプロダクトタームを削除するためのブーリアン論理規則を示します。

表現		結果
!0	=	1
!1	=	0
A & 0	=	0
A & 1	=	A
A & A	=	A
A & !A	=	0
A # 0	=	A
A # 1	=	1
A # A	=	A
A # !A	=	1
A & (A # B)	=	A
A # (A & B)	=	A

表 3-3 ブーリアン論理規則

## デバイスオプション

アドバンスド PLD コンパイラは他のコンパイラと同様に、ソースファイルを処理しターゲットアーキテクチャに基づいて出力を作成します。但し、このコンパイラは、数百種類のターゲットアーキテクチャを持っています。これにより、ソースファイルを変更しなくても、別のアーキテクチャで設計した論理回路を実行することができます。デバイス情報は Device Library に保存されています。

アドバンスド PLD にデバイスを指示する方法は 2 つあります。

- Target Device ダイアログボックスで指定
- PLD ソースファイルで指定

## デバイスをコンパイラソースファイルで指定するには

ターゲットデバイスをコンパイラソースファイルで指定することができます。PLD ファイルのヘッダーセクションの DEVICE フィールドで指定して下さい。PLD ファイルのヘッダーセクションには数種類のフィールドがあります。各フィールドは、キーワードとそれに続く情報から構成されます。

DEVICE フィールドのシンタックスの例を以下に示します。

```
DEVICE P16L8;
```

ターゲットデバイスがソースファイルで指定される場合、Target Device ダイアログボックスで自動的にそのデバイスが選択されます。

### Target Device ダイアログボックスでデバイスを指定するには

Target Device ダイアログボックスにアクセスするには、Configure AdvancedPLD ダイアログボックスの Change ボタンを押して下さい。Target Device ダイアログボックスによりアドバンスド PLD の現在のバージョンでサポートされているターゲットデバイスが表示されます。

ターゲットデバイスを変更するには、Device Type と Device Name を選択し OK ボタンを押して下さい。

Target Device ダイアログボックスでデバイスが指定されると、アドバンスド PLD は PLD ソースファイルの Device 仕様を自動的に更新します。

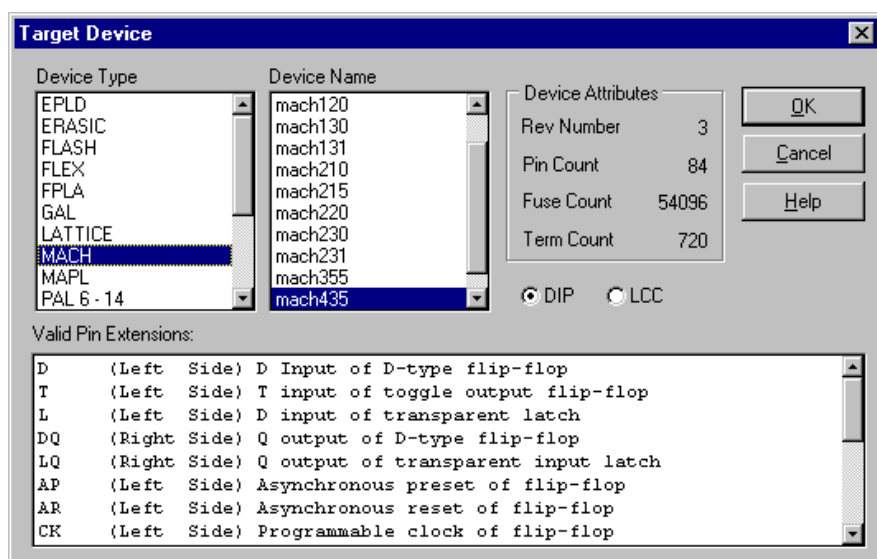


図 3-4 ターゲットデバイスの選択

### デバイスライブラリー

Target Device ダイアログボックスで表示される情報は、デバイスライブラリファイル(CUPL.DL)に保存されています。このファイルには、コンパイラによりサポートされているターゲットデバイスの記述が格納されています。ライブラリには、各デバイスの内部アーキテクチャやピンの数、有効

な入出力ピン等の物理特性や、登録ピンと非登録ピンやプロダクトタームの数、ヒューズマップ情報、ダウンロードフォーマット情報等の論理特性が記述されています。

## デバイスニーモニック(デバイス名)

アドバンスト PLD は、デバイスニーモニックと呼ばれるデバイスアーキテクチャのネーミングシステムを使用します。このネーミングシステムはアーキテクチャに対するものであり、デバイスそのものに対するものではないことに注意して下さい。例えば、AlteraEP312 と Intel5AC312 は両方とも同じアーキテクチャです。従って、これら製造元のこれらのデバイスのどちらかを使用して回路を設計した場合、コンパイラにはターゲットアーキテクチャは EP312 であると伝えられます。

デバイスニーモニックは、複数の製造元を参照することができます。Devices.TXT 中のニーモニック - 製造元クロスリファレンスを参照して下さい。

ニーモニックはデバイスファミリープリフィックスと業界標準のパーツ番号の差フィックスで構成されます。

シンボル	意味
EP	Erasable Programmable Logic Device (EPLD)
G	Generic Array Logic (GAL)
F	Field Programmable Logic Array (FPLA)
F	Field Programmable Gate Array (FPGA)
F	Field Programmable Logic Sequencer (FPLS)
F	Field Programmable Sequence Generator (FPSG)
P	Programmable Logic Array (PAL)
P	Programmable Logic Device (PLD)
P	Programmable Electrically Erasable Logic (PEEL)
PLD	Pseudo Logical Device
RA	Bipolar Programmable Read Only Memory (PROM)

表 3-4 デバイスニーモニックプリフィックスの一覧

例えば、PAL10L8 のデバイスニーモニックは P10L8 です。また、82S100 のデバイスニーモニックは F100 です。バイポーラ PROM ではサフィックスはアレイの大きさを表わします。例えば、1024x8 バイポーラ PROM のデバイスニーモニックは RA10P8 です。すなわち、アドレスピンが 10 本とデータ出力ピンが 8 本です。

## LCC と PLCC デバイス

コンパイラが使用するデフォルトのパッケージタイプは DIP パッケージです。デバイスの中には LCC または PLCC のバージョンを持つものがあります。これらは、Surface Mount Technology または略して SMT と呼ばれることもあります。SMT デバイスのアーキテクチャは DIP のものと同じですが、ピンアウトは異なります。そのために、アドバンスト PLD は、デバイスアーキテクチャの SMT バージョンには異なるニーモニックを使用します。こ

れには、DIP ニーモニックにサフィックスの lcc が付けられます。P16L8 の SMT バージョンは P16L8LCC です。

### 仮想デバイス (Virtual Device)

仮想デバイスオプションにより、ターゲットアーキテクチャに関係なくプログラマブルロジックのデジタル設計を行なうことができます。仮想デバイスはデバイスではありません。コンパイラにより制限が取除かれたデバイスで、プロダクトタームやピンを無制限に使用して回路を設計することができます。仮想デバイスは、設計する回路に必要なリソースを決めるのに有効です。

仮想デバイスを使用すると、コンパイラはピンの定義の中の極性を無視します。ピン番号はそのままです。

仮想デバイスをイネーブルにするには、Configure AdvancedPLD ダイアログボックスの Option Tab の Use Virtual Device チェックボックスを選択するか論理記述ファイル (ファイル名.PLD) のヘッダーセクションのデバイスに VIRTUAL を指定して下さい。

## アドバンストPLDによるコンパイル

コンパイラを起動する前に、アドバンスト PLD はトピックに記述されているように設定されます。アドバンスト PLD コンパイラを設定して.PLD ソースファイルをアクティブドキュメントにします。 .

PLD ファイルをコンパイルの前に保存します。

コンパイルを実行するには、PldTools ツールバーの Compile ボタンを押すか Tools-Compile メニューアイテムを選択して下さい。アドバンスト PLD - Compiling ダイアログボックスがポップアップ表示されコンパイルが実行されます。

コンパイラは Configure AdvancedPLD ダイアログボックスで選択された出力ファイルを作成します。View Result オプションをイネーブルにすると出力ファイルが自動的に開かれます。

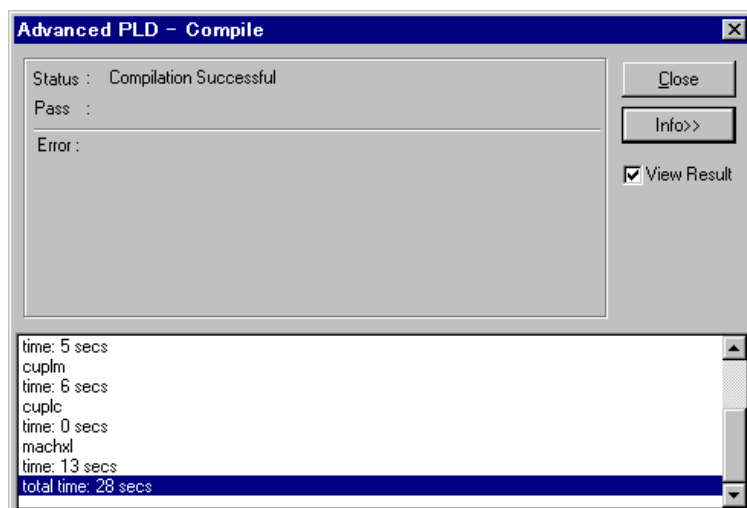


図 3-5 デザインのコンパイル

## アドバンストPLD によるシミュレーション

このセクションではアドバンスト PLD のシミュレーションの概要を説明します。

### 入力

テスト仕様ソースファイル（ファイル名.SI）はシミュレータの入力です。このファイルには回路の中のデバイスに必要な機能の記述が格納されています。

ソースファイルは TextExpert を使用して作成することができます。

シミュレータソースファイルで入力される入力ピンの刺激や出力ピンのテスト値はコンパイラソースファイルで論理式から計算された実際の値と比較されます。これらの計算された値は、アブソリュートファイル（ファイル名.ABS）に書き込まれます。このファイルは Configure AdvancedPLD ダイアログボックスで Absolute ABS オプションをイネーブルにするとコンパイル中に作成されます。

このガイドのサンプルデザインセッションのシミュレーションテストベクタの作成を参照して下さい。

シミュレーションを実行する場合、コンパイラにより作成される JEDEC ダウンロードファイルが必要です。

### 出力

シミュレーションの結果はシミュレーションリスティングファイル（ファイル名.SO）に書き込まれます。また、シミュレータはテストベクタを JEDEC ダウンロードダブルヒューズリンクファイルに追加します。

ヘッダー情報はすべてヘッダーエラーと一緒にリスティングファイルに表

示されます。コンプリートベクタには、番号が割り付けられます。失敗した出力テストには、印が付けられ実際の（シミュレータが決めた）出力値と一緒に表示されます。各変数は、予想される（ユーザが決めた）値と一緒に一覧表示されます。無効なテスト値は、適当なエラーメッセージと一緒に一覧表示されます。

シミュレータはコンパイラのように複数のデバイスファイルをサポートしていません。シミュレータは複数のデバイスファイルの最初のデバイスだけをシミュレートします。

### シミュレータの起動

シミュレーションを実行すると、.PLD ファイルはアクティブドキュメントになります。カレントディレクトリに.SI ファイルが必要です。EDA/クライアントのアクティブドキュメントとして.SI ファイルを用いてシミュレーションを実行しようとししないで下さい。

シミュレーションを実行するには、PldTools ツールバーの Simulate ボタンを押して下さい。Advanced PLD-Simulate ダイアログボックスがポップアップ表示されシミュレーションが実行されます。

シミュレータはシミュレーションリスティングファイル（ファイル名.SO）を作成します。各変数の入力と出力の値は、リストにされます。エラーメッセージは各ベクタに続いてエラーディスプレイの信号名と一緒に示されます。シミュレータにより作成された.SO ファイルはシミュレーションが完了すると EDA/クライアントのカレントドキュメントになります。

View Results チェックボックスをイネーブルにすると Waveform エディタにシミュレーションリスティングファイルが自動的に表示されます。

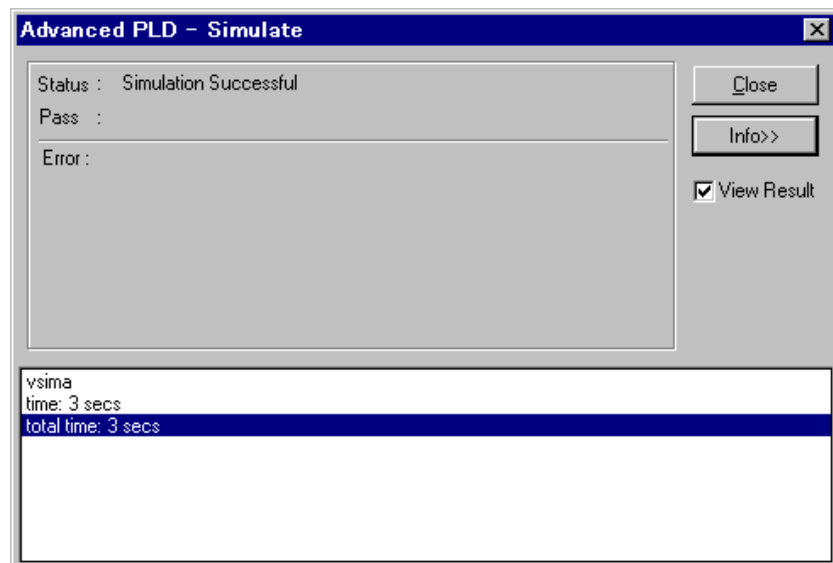


図 3-6 デザインのシミュレーション

## シミュレーション波形の表示

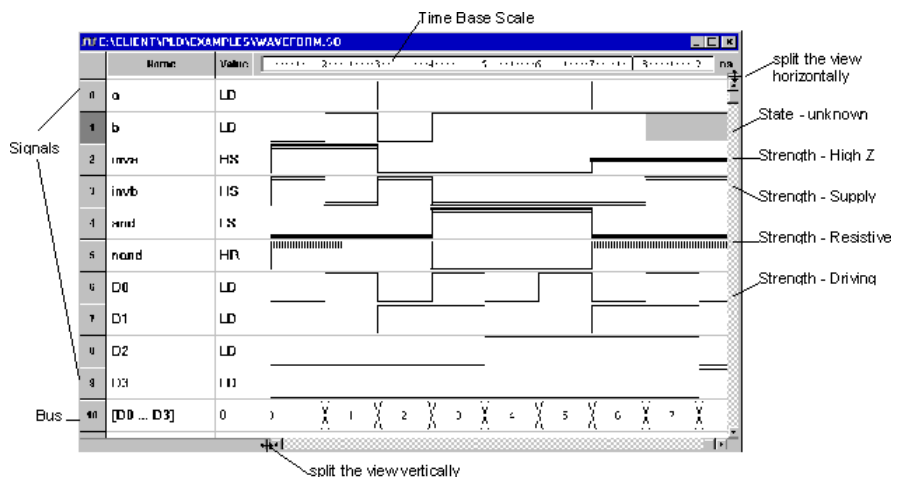
Waveform Editor を使用してシミュレーション結果を波形として表示することができます。シミュレーションリスタンピングファイルには、シミュレーションの結果が保存されています。このファイルは、Text Expert や Waveform Editor で開くことができる ASCII ファイルです。

## シミュレーション結果の表示

シミュレーション結果の標示は Wave エディターで行います。以下に波形標示の方法を説明します。

- File-Open. コマンドを選択して下さい。
- Open Document ダイアログボックスで Editor を Wave に設定して下さい。  
Wave が一覧の中に無い場合、Wave Server がインストールされていません。  
サーバーのインストールに関する説明についてはインストールセクションを参照して下さい。
- Type を PLD シミュレータファイル(\*.SO ) に設定して下さい。
- シミュレーションリスティングファイルを選択し OK をクリックしファイルを開いて下さい。

シミュレーション結果が一連の波形としてスプレッドシート形式の波形ウィンドウに表示されます。



## Time Base

Time Base は波形ウィンドウの一番上のスケールに表示されます。デフォルトでは、シミュレーションリスティングファイルのイベントの変化は波形では 1ns の時間がかかります。

仮想マーカーが Time Base Scale に表示され、現在のカーソル位置を示します。  
正確な時間は Status Line の左端に表示されます。

### Signal Name



信号名はシミュレーションリスティングファイルから抽出されます。名前を編集したりフォントを変更したりできます。編集するには名前をダブルクリックして下さい。

## Signal Value

Value 列は現在の時間位置での各信号の状態を表わします。現在の時間位置は、Time Base Scale の左端です。波形をスクロールすると Value 列の内容が変化することが分かります。

それぞれの値には 2 つの特性があります。以下にその意味を示します。

- L 信号の状態は Low
- H 信号の状態は High
- D 信号の強さは Driving
- S 信号の強さは Supply
- R 信号の強さは Resistive
- xx バスの現在の値を示すバス信号（2 進数または、10 進数、16 進数）

## 信号はなぜ違って見えるか？

信号はその強さに応じて違って表わされます。それぞれの強さがどのように表現されるかについて図 3-7 を参照して下さい。View-Legend メニューアイテムを選択して各信号の強さの例を確認して下さい。

表示される信号の強さはシミュレーションリスティングファイルの信号タイプで決まります。

## タイミングマーク

Waveform Editor にはタイミングマーク 10 個あり任意の位置に置くことができます。Edit-Set Timing Marks サブメニューを選択しタイミングマークを適当な位置に設定して下さい。Edit-Jump サブメニューを使用して、設定したタイミングマークへジャンプできます。（ショートカット：J に続いてタイミングマーク番号）

## バスの作成

近接する信号の任意の組みからバスを作成することができます。バスを作成する方法を説明します。

- バスに組み込む信号をクリック & ドラッグして選択して下さい。（これらの信号は黒でハイライト表示されます。）
- Insert-Bus コマンドを選択して下さい。

バスは、選択した信号の最後に挿入されます。Value 列に現在の時間位置のバスの値が表示されます。バスに含まれる各信号はバスの 1 ビットを表わし、最上位に選択された信号が最下位ビットを表わします。

## 信号の編集

- トランジションをダブルクリックして編集して下さい。
- 信号のトランジションをクリック & ドラッグすると、開始時間をグラフィック画面上で変更できます。

## 表示位置の変更

Waveform Editor には、波形を検証するためのいろいろな機能があります。

### 垂直方向のスクロールと水平方向のスクロール

スクロールバーを使用して波形ウィンドウを後ろや前、上、下へスクロールして下さい。矢印キーを使用してもできます。大きくスクロールアップやスクロールダウンしたい場合、PAGE UP や PAGE DOWN ショートカットキーを使用して下さい。

### ズームインとズームアウト

View-Zoom In ( ショートカット ; + ) または View-Zoom Out ( ショートカット ; - ) メニューアイテムを選択して、Time Base Scale を変更して下さい。Zoom In や Zoom Out プロセスは現在のカーソル位置で行われます。ショートカットキーを押す前のカーソル位置です。

### パンニング

HOME キーを押すと波形の表示を左右にふることができます。このショートカットキーは、現在のカーソル位置で行われます。ショートカットキーを押す前のカーソル位置です。

### マルチ表示にする方法

Waveform Editor はマルチ表示をサポートしており、波形の別の位置を同時に表示することができます。表示は縦に分割 ( 違う時間位置を同時に表示する ) したり横に分割 ( 同時に見る事ができない別の波形を同時に表示する ) したりできます。表示を分割して使用した時のカーソルの位置は図 3-7 を参照して下さい。図のようにカーソルを置いて、クリック & ドラッグすると画面が分割されます。

## 波形標示 Wave エディターパネル

View-Panel メニューアイテムを選択し Waveform Editor Panel を表示したり消したりして下さい。Panel を使用してアクティブ信号のトランジションを進めたり信号の表示 / 非表示を設定して下さい。

### Active Signal

Panel の一番上のボックスは、アクティブ信号の名前を表示します。Jump Transition ボタンにより ( アクティブ信号の ) ジャンプできる位置を以下に示します。

最初のトランジション ( ショートカット ; J,F )  
 前のトランジション ( ショートカット ; J,P )  
 次のトランジション ( ショートカット ; J,N )  
 最後のトランジション ( ショートカット ; J,L )

## 信号の表示と非表示

Waveform Editor により、見たくない信号のサブセットを表示しないようにできます。クリック & ドラッグにより表示したくない信号を選択し、Panel 上の Hide ボタンを押して下さい。

メインツールバーの Select ボタンを使用して信号をすべて選択し、Hide ボタンを押してそれらを表示しないようにして下さい。そして、見たい信号を選択して表示して下さい。パネルの選択を表示するには、シフトとコントロールキーを押しながら信号名をクリックして下さい。

# PLD の設計プロセス

## ステップ

以下の段階に従って、PLD の設計プロセスを説明します。

### 設計作業の検討

設計作業を注意深く見てみると、ステートマシンやブーリアン式、真理値表が設計に使用されます。どのタイプのシンタックスが設計作業に一番合うかを決めて下さい。

### コンパイラソースファイルの作成

与えられたテンプレートファイルを使用し、不要なセクションを取除いて下さい。作成される新しいファイルに影響するヘッダーを編集することを忘れないようにして下さい。

### 式の創出

式は、CUPL シンタックスで記述する必要がある必要論理を記述して下さい。これによりブーリアンやステートマシン、真理値表を記述することができます。

### ターゲットデバイスの選択

入力ピンの数が充分であることを確認して下さい。

レジスタードまたはノンレジスタードの出力ピンの数が設計に対して充分あるか確認して下さい。

必要に応じて、デバイスが three-state 出力コントロールができるかを確認して下さい。

デバイスが必要なプロダクトタームの数を適当に操作できるかを確認して下さい。

Configure AdvancedPLD ダイアログボックスでターゲットデバイスを選択して下さい。

### ピンの割り付け

設計の入力と出力をデバイスのピンに割り付けて下さい。製造元の取り扱い説明書に従ってデバイスが適切に使用されることを確認して下さい。

### コンパイルの準備

ダウンロードやシミュレーションにどのファイルフォーマットが必要かを決めて下さい。4 種類の最小化法を選択できます。設計した論理回路をコンパイルする準備が完了しました。

## コンパイラソースファイルの作成

コンパイラソースファイルは、CUPL ハードウェア記述言語で PLD の設計を記述した論理記述ファイルです。PLD 論理記述ファイルには、設計をコンパイルしたりデバイスプログラマに適切なダウンロードファイルを作成するコンパイラへの入力保存されています。

このセクションでは、PLD ソースファイルを CUPL で作成する方法について説明します。

## CUPL ソースファイルシンタクスの概要

コンパイラソースファイルの作成することは、テンプレートと呼ばれる一般的な輪郭を作成することに付随します。テンプレートファイル (TMPL.PLD) は、ガイドに与えられます。ヘッダー情報は必ずソースファイルの始めにありそれに続いてピン / ノードの定義が記述される必要があります。テンプレートのその他のエリアは、必要に応じて使用され、順番は特に決められていません。以下のセクションでは、このソースファイルのフォーマットについて説明します。

### ヘッダー情報

ヘッダーには、設計に関する基本的な情報が記述されます。記述される情報の種類には、ファイル名やデバイスのパーツナンバー、開始日、設計のリビジョン番号、会社名、設計者名、デバイスのアセンブリ、アセンブリでのデバイスの場所があります。また、ヘッダーは、設計に使用されるデバイスを指定したりコンパイラで作成される出力の型を指定できます。ヘッダー情報の記述順序は任意です。また、情報は name フィールド以外はすべて任意です。ヘッダーアイテムが無い場合、ワーニングメッセージが表示されますがコンパイルは実行されます。

Name	XXXXXXX;
Partno	XXXXXXX;
Date	XX/XX/XX;
Revision	XX;
Designer	XXXXXXX;
Company	XXXXXXX;
Assembly	XXXXXXX;
Location	XXXXXXX;
Format	XXXXXXX;
Device	XXXXXXX;

図 4-1 ヘッダー情報のセクション

### タイトルブロック

タイトルブロックは、設計者に与えられたコメントエリアでタイトルを記入したり設計についての記述をしたりします。実際のデバイス名や製造元をここに書くこともできます。

/*****	
/ *	*/
/ *	*/

```

/*                                                    */
/*****/
/* Allowable Target Device Types:                    */
/*****/

```

図 4-2 タイトルブロックセクション

## ピン/ノードの定義

ピン/ノードセクションには、セクションのコメントラベルを記述したり、設計者が自分の設計のための番号やラベルを記述するピン宣言ステートメントを記述します。ピン宣言の順番は任意です。ただし、使用に応じてピンをグループ分けしたときに理解しやすいようにして下さい。空のコメントは各ピン宣言が終わった後、設計の中でのピンを役目をよく表わした表現が与えられます。

```

/** Inputs **/
Pin      =      ;      /*
Pin      =      ;      /*
Pin      =      ;      /*

/** Outputs **/
Pin      =      ;      /*
Pin      =      ;      /*
Pin      =      ;      /*

Pinnode  =      ;      /*
Pinnode  =      ;      /*

```

図 4-3 ピン/ノード宣言セクション

## 中間変数

中間変数は、ソースファイルのピン宣言セクションでは定義されません。中間変数名は、論理式や追加の中間変数を生成する他の表現中で使用されます。このようなトップダウン形式で論理式を記述することで論理記述ファイルは読みやすくそして理解しやすくなります。

```

/** Declarations and Intermediate Variable Definitions **/

```

図 4-4 中間変数セクション

## 論理式

このセクションは手紙の本文のようなところです。ステートマシンや真理値表、ブーリアン式などの設計の重要な部分はすべてここに記述されます。

```

/** Logic Equations **/

```

図 4-5 論理式セクション

## PLD ソースファイルのコンパイル

このセクションでは、コンパイラがソースファイルに実行することやコンパイラが作成する出力ファイルのタイプを簡単に説明します。

## コンパイラについて

コンパイラは、命令の記述されたテキストファイル进行处理し、ハードウェア用のプリミティブな情報が含まれたファイルを作成します。この情報は、論理関数をプログラマブルロジックデバイスにプログラムするデバイスプログラマーまたは、設計した回路のシミュレーションを実行するシミュレータにより使用されます。

## コンパイラへの入力

ソースファイルは、コンパイラディレクティブやコマンド、コメントで構成されます。コンパイラディレクティブは、コンパイル中に実行されるコンパイラへの命令です。コマンドは、設計そのものを構成する高級命令です。コメントは、設計情報のためだけに設計に書き込まれる注釈で出力に影響しません。

## コンパイラからの出力

コンパイラは、3種類の標準的なダウンロードフォーマット; JEDEC、ASCII Hex、HL を作成することができます。JEDEC 出力は、ダウンロードヒューズマップ用のデバイスプログラマーに必要なフォーマットです。ASCII hex ファイルは PROM デバイスヒューズマップ用のプログラマーに必要なフォーマットです。HL フォーマットは Signetics シーケンサデバイスが選択された場合の通常出力フォーマットです。

## コンパイラオプション

コンパイラオプションは、ソースファイルに組み込まれないコンパイラ命令です。ただし、この命令は、Configure AdvancedPLD ダイアログボックスからコンパイラに与えられます。ダイアログボックスの命令は、ソースファイルにあるコンパイラ命令のすべてをオーバーライドします。また、これらの命令により、コンパイラに作成する出力ファイルを支持します。

## ソースファイルディレクティブ

ほとんどのソースファイルのディレクティブはダラーマーク(\$)で始まります。これらのディレクティブは通常プレプロセッサ命令として参照されます。使用される場合、それらの命令は、ソースファイルのある列から始まります。大文字と小文字の任意の組み合わせを使用してこれらのコマンドを入力することができます。ソースファイルディレクティブのすべてはソースファイルのどの位置に記述してもかまいません。

## シンボル定数の定義と定義の解除

シンボル定数を定義するには `$define` コマンドを使用します。このコマンドは、チャラクタ文字列を他の指定された演算子や数字、シンボルに置き換えます。置き換えは、コンパイラにより処理される前に入力ファイルに行

われます。シンボル定数の定義を解除するには\$undef コマンドを使用して下さい。このコマンドにより\$define コマンドはキャンセルされます。

## 他のファイルのインクルード

他のファイルをソースファイルにインクルードするには\$include コマンドを使用して下さい。ファイルをインクルードすることでよく使う CUPL コードの部分を標準化することができます。また、多くのソースファイルで使用する定数の定義を別のファイルに分離することができます。インクルードされるファイルの中にも\$include コマンドを使用でき、インクルードファイルをネストすることができます。

## 条件付きコンパイル

条件付きコンパイルにより、シンボル定数が存在するか否かに応じてソースコードのセクションをコンパイルすることができます。これらのコマンドは\$ifdef や\$ifndef、\$else、\$endif です。シンボル定数が定義された場合、\$ifdef コマンドに続くソースステートメントは、\$else または\$endif コマンドが現れるまでコンパイルされます。定義がまだされていない場合、\$ifdef コマンドに続くソースステートメントは無視されます。\$ifndef コマンド\$ifdef コマンドと反対の様式で動作します。\$ifdef や\$ifndef コマンドの条件が False の場合、\$else や\$endif コマンドの間のソースステートメントがコンパイルされます。その他の場合は無視されます。\$endif コマンドは\$ifdef や\$ifndef コマンドから始まった条件コンパイルの終わりを表わすために使用されます。条件コンパイルはネストすることができます。\$ifdef や\$ifndef コマンドのネストの各レベルはそれぞれ\$endif で終わる必要があります。

```
$DEFINE Prototype X /* define Prototype*/
$IFDEF Prototype
pin 1 = memreq;      /* memory request on */
                    /* pin 1 of prototype */
pin 2 = ioreq;        /* I/O request on*/
                    /* pin 2 of prototype */
$ELSE
pin 1 = ioreq;        /* I/O request on*/
                    /* pin 1 of PCB */
pin 2 = memreq;       /* memory request on */
                    /* pin 2 of PCB */
$ENDIF
```

図 4-6 条件付きコンパイルの使用例

## 繰り返し命令

一連の CUPL 言語ステートメントを繰り返すには、\$repeat と\$repeat コマンドを使用して下さい。このコマンドは C 言語の FOR 命令や Fortran 言語の DO 命令に似ています。\$repeat と\$repeat の間のステートメントは、\$repeat ステートメントが許す回数だけ繰り返されます。これにより、カウンタに基づくステートマシンの定義が簡単にできます。



## ブリプロセッサ後の CUPL ソースコードの結果

```
FIELD sel = [in2..0] FIELD sel = [in2..0];
$REPEAT i = [0..7]    !out0 = sel:'h'0 & enable;
    !out{i} = sel:'h'{i} & enable; !out1 = sel:'h'1 & enable;
$REPEND    !out2 = sel:'h'2 & enable;
            !out3 = sel:'h'3 & enable;
            !out4 = sel:'h'4 & enable;
            !out5 = sel:'h'5 & enable;
            !out6 = sel:'h'6 & enable;
            !out7 = sel:'h'7 & enable;
```

図 4-7 \$repeat と\$repnd コマンドの使用例

## マクロの使用

マクロはユーザ定義のコマンドで一連のコマンドを一語で置き換えることができます。マクロは\$macro と\$mend コマンドを使って使用します。\$macro と\$mend コマンドの間のステートメントは、マクロ名が呼ばれるまでコンパイルされません。マクロはソースファイルにマクロ名を記述することで呼び出され、パラメータをマクロに渡します。マクロはデコーダやカウンタなどのライブラリを作成するために使用されます。マクロ展開ファイル(.MX)を作成できるようになるとブリプロセッサがマクロ定義をどのように処理するかがわかります。

```
$MACRO decoder bits MY_X MY_Y MY_enable;
    FIELD select = [MY_Y{bits-1}..0];
    $REPEAT i = [0..{2*(bits-1)}]
        !MY_X{i} = select:'h'{i} & MY_enable;
    $REPEND
$mend
_/* Other CUPL statements */
decoder(3, out, in, enable); /*macro function call*/
```

図 4-8 \$macro コマンドと\$mend コマンドの使用例

## 出力の最小化

MIN 命令を使用するとピン毎に出力を最小化することができます。MIN 命令の最小化レベルは 0 から 4 までで、それらは Configure AdvancedPLD ダイアログボックスのオプションに対応します。この命令により同じ回路内の異なる出力に異なる最小化レベルを指定することができます。

表 4-1 にステートメント番号と Configure AdvancedPLD ダイアログボックスでそれらに対応するラベルを示します。

番号	対応ラベル
0	None
1	Quick
2	Quine-McCluskey
3	Presto

表 4-1 ステートメント番号とダイアログボックスでのラベル

MIN async_out = 0;	/* 最小化しない */
MIN [outa, outb] = 2;	/* 最小化レベル 2 */
MIN count.d = 4;	/* 最小化レベル 4 */

図 4-9 個別の最小化の例

## ヒューズの断線技術

デバイスの中には、デバイスの特性を変えることができるヒューズを持ったものがあります。現在、これらのヒューズは MISER ビットや TURBO ビットとして参照されます。デバイスの特性に応じて、これらのヒューズを断線したり断線しなかったりして必要なデバイス特性を実現します。このような動作は FUSE ステートメントにより行ないます。ヒューズ番号と断線値を記述するとコンパイラにより指定された値にフューズが設定されます。このステートメントは、正しく使用しないと予測できない結果を招きますので充分注意して使用する必要があります。

FUSE(101,1);	/* Turbo ビットを断線します */
FUSE(102,0);	/* Miser ビットを断線しません */

図 4-10 FUSE ステートメントの使用例

## CUPL 言語による設計

回路を設計する場合、トップ-ダウンアプローチを使用して設計されます。トップ-ダウン設計は最初設計の全体定義から始めてメインの各要素の定義プロセスなどを繰り返し、最終的にプロジェクト全体を定義する方法です。CUPL はこの種の設計を実行するのに都合の良い多くの機能を持っています。このセクションでは CUPL が持つ設計実行するための各種の命令について説明します。

### 言語要素

このセクションでは、CUPL 論理記述言語に含まれる要素について説明します。

#### ピン/ノードの定義

ピンの定義はソースファイルの始めで宣言する必要があるため、通常それらの定義から設計作業が開始されます。埋められたレジスタを定義するために使用するノードやピンノードはソースファイルの最初で定義する必要があります。使用するデバイスがすでに決まっている場合、ピン割り付けが必要です。しかし、VIRTUAL 設計を行なう場合、設定する必要があるのは後でピンに割り付けられる変数名だけです。通常ピン番号が入るエリアは空白のままでもかまいません。

#### 中間変数の定義

中間変数は式に割り付けられる変数です。ただし、ピンやノードには割り付けられません。これらの変数は多くの変数により使用される式を定義するために使用されたり、設計をわかりやすくするために使用されます。インデックス付き変数の使用

0 から 31 までの十進数で終わる変数名はインデックス付き変数として参照されます。この変数はアドレスラインやデータライン、その他一連の番号が付けられたアイテムのグループを表わすために使用されます。インデックス付き変数がビットフィールドで使用されると、インデックス番号 0 が付けられた変数は常に最下位ビットを表わします。

## 基数の使用

コンパイラで数値を含むすべての操作は32ビットの精度で実行されます。したがって、数値は 0 から 232-1 の値を持ちます。数値は 4 種類の基数で表わすことができます。すなわち、2 進数、8 進数、10 進数、16 進数です。デフォルトの基数は 16 です。ただし、デバイスのピン番号やインデックス付き変数には 10 進数が使用されます。2 進数や 8 進数、16 進数の数値は、数値と混合される dont care(X)バリュウを持ちます。

数値	基数	10 進数値
'b'0	2 進数	0
'B'1101	2 進数	13
'O'663	8 進数	435
'D'92	10 進数	92
'h'BA	16 進数	186
'O'[300..477]	8 進数(範囲)	192..314
'H'7FXX	16 進数(範囲)	32512..32767

表 4-2 基数の使用法

## リスト表記の使用

リストは変数のグループを定義する簡単な方法です。ピンやピンノード宣言、ビットフィールドの宣言、論理式、演算の組みで使用する場合に通常使用されます。かぎ括弧を使用してリストを区切ります。

## ビットフィールドの使用

ビットフィールド宣言はビットのグループに変数名を割り付けるために使用されます。FIELD キーワードを使用してビットフィールドの割り付けを行なうと、その名前は、表現の中で使用されます。すなわち、演算はグループの各ビットに適用されます。FIELD ステートメントが使用されると、コンパイラは内部に一つの 32 ビットフィールドを作成します。これを使用して、ビットフィールドに変数を表現します。各ビットはビットフィールドの一つを表わします。ビットフィールドを表わすビット番号は、インデックス付き変数が使用されている場合、インデックス番号と同じです。すなわち、A0 はビットフィールドのビット 0 を常に使用します。これは主に、アドレスやデータバスを定義したり扱うために使われます。

```

FIELD ADDRESS = [A7, A6, A5, A4, A3, A2, A1, A0];
or
FIELD ADDRESS = [A7..0];

FIELD Mode = [Up, Down, Hold];

```

図 4-11 FIELD ステートメントの使用例

## 言語シンタックス

このセクションでは、CUPL 言語を使用して設計を行なうために必要な論理演算子や算術演算子、算術関数について説明します。

### 論理演算子

NOT、AND、OR、XOR の 4 種類の基本的な論理演算子を使用することができます。以下の表は、演算子と優先順位の一覧です。優先順位が高いほうから低いほうに表示されています。

演算子	例	説明	優先順位
!	!A	NOT	1
&	A & B	AND	2
#	A # B	OR	3
\$	A \$ B	XOR	4

表 4-3 論理演算子とそれらの優先順位

### 算術演算子と算術関数

\$repeat や \$macro コマンドで使用できる基本的な演算子が 6 種類使用できます。以下の表はこれらの演算子と優先順位の一覧です。優先順位が高いほうから低いほうに表示されています。

演算子	例	説明	優先順位
**	2**3	べき乗	1
*	2*i	掛け算	2
/	4/2	割り算	2
%	9%8	余り	2
+	2+4	足し算	3
-	4-i	引き算	3

表 4-4 算術演算子とそれらの優先順位

\$repeat\$macro コマンドで使用される表現には一つの算術関数を使用することができます。以下の表は、算術関数とその基数を示します。

関数	基数
LOG2	2 進数
LOG8	8 進数
LOG16	16 進数

表 4-5 算術関数

LOG 関数は整数を返します。

$\text{LOG2}(32) = 5 \iff 2^{*5} = 32$

$\text{LOG2}(33) = \text{ceil}(5.0444) = 6 \iff 2^{*6} = 64$

$\text{ceil}(x)$  は  $x$  より小さくない最小の整数を返します。

### 変数の拡張子

変数名に拡張子を追加しプログラマブルデバイスの中でメジャーノードに関連付けられた特定の関数を表わすことができます。これらの関数にはフリップフロップやプログラマブルトリステートなどが含まれます。コンパイラは拡張子の使用をチェックし、指定されたデバイスで有効かどうかや他の拡張子と競合していないかを判断します。コンパイラはこれらの拡張子を使用してデバイスのマクロセスを配置します。したがって、設計者は、マイクロセルの中のどのヒューズがなにをコントロールするかを考える必要がありません。

図 4-12 は拡張子の使用例を示します。この図の回路は実際の回路ではないことに注意して下さい。拡張子を使用して回路内の異なる関数の式を記述する方法を説明しています。

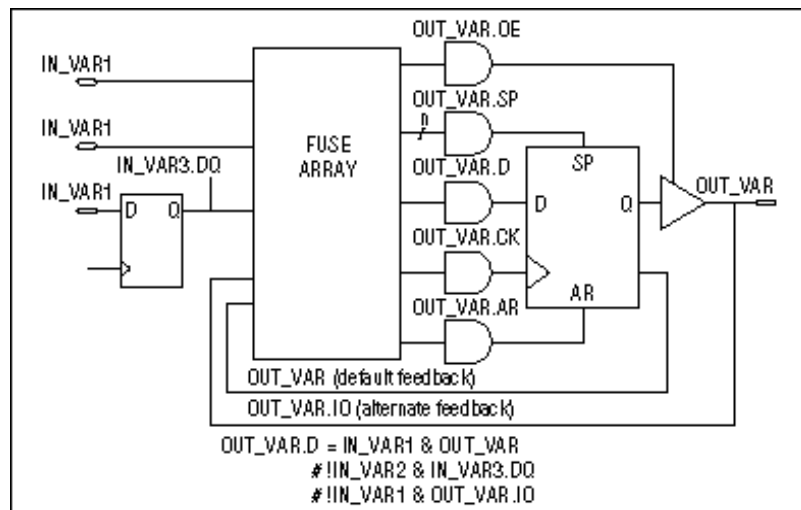


図 4-12 拡張子を図示した回路

### 論理式

論理式とは、CUPL 言語のビルディングブロックです。論理式の形式を以下に示します。

```
[!] var [.ext] = exp ;
```

ここで、var は変数またはリスト表記のルールに従って定義されたイ

ンデックス付きあるいはインデックスなしの変数のリストです。変数リストが使用される場合、式はリスト内の変数それぞれに適用されます。

また、.ext は省略可能な変数拡張子でこの拡張子を利用してプログラマブルデバイス内のメジャーノードに関数が割り付けられます。

exp は、式です。すなわち、変数と演算子の組み合わせです。

=は、割り付け演算子です。式の値を変数または変数の組みに割り付けます。

!は、補足演算子です。

標準の論理式では、通常ひとつの式が変数に割り付けられます。APPEND 命令により複数の式を一つの変数に割り付けることができます。APPEND された論理式はその変数の元の式に論理的に OR されます。APPEND 命令の書式は、論理式の定義とは独立です。ただし、論理式が始まる前にキーワード APPEND が現れたときは別です。

テンプレートファイルで与えられるソースファイルの論理式セクションに論理式を記述して下さい。

### セット演算の使用

入力ピンや一つのレジスタ、出力ピンなどの 1 ビットの情報を扱う演算子はすべて、グループになった複数のビットに適用することができます。セット演算は、ある組みと変数または式、あるいは二つの組みの間で実行することができます。

ある組みと一つの変数（または式）の間の演算の結果は、演算が組みの各要素と変数（または式）の間で行われ新しい組みになります。

演算が二つの組みの間で実行される場合、その組みは同じ大きさである必要があります（すなわち、要素の数が同じ）。二つの組みの間の演算結果は、各組みの要素の間で実行され新しい組みになります。

数値がセット演算で使用される場合、それらは 2 進数値の組みとして扱われます。8 進数の数値は 3 つの 2 進数値の組みとして扱われます。また、10 進数や 16 進数の数値は 4 つの 2 進数値の組みとして扱われます。

### 等価演算

他のセット演算と違い、等価演算はひとつのブーリアン式を評価します。変数の組みと定数とをビット単位でチェックします。定数のビット位置は、その組みの対応する位置の値と比較されます。ビット位置が 2 進数の 1 の場合、組みの要素は変更されません。ビット位置が 2 進数の 0 の場合、組みの要素は、否定されます。ビット位置が 2 進数の X の場合、組みの要素は削除されます。演算後の各要素は互いに AND され一つの式を作ります。

The equality operator can also be used with a set of variables that are to be operated upon identically. For example, the following three expressions

```
[A3,A2,A1,A0]:&
[B3,B2,B1,B0]:#
[C3,C2,C1,C0]:$
```

are equivalent respectively to:

```
A3 & A2 & A1 & A0
B3 # B2 # B1 # B0
C3 $ C2 $ C1 $ C0
```

## レンジ演算

レンジ演算は、等価演算と似ていますが、定数フィールドが一つの値ではなくて値の範囲になっています。ビットの比較は範囲内の各定数値により行われます。

まず始めに、アドレスバスを以下のように定義して下さい。

```
FIELD address = [A3..A0]
```

次に RANGE 式を記述して下さい。

```
select = address:[C..F] ;
```

これは以下の式と等価です。

```
select = address:C # address:D # address:E # address:F;
```

## 真理値表

論理表現を明確に表現するために情報のテーブルを使用することがあります。CUPL は TABLE キーワードにより情報テーブルを作成することができます。まず、関連した入力と出力の変数リストを定義します。次に、入力と出力の変数リストの値の間を一对一で割り付けて下さい。入力値として dont-care 値を使用できます。ただし、出力には使用できません。

入力値のリストには、一つの命令で複数の割り付けを行なうことができます。以下のブロックは簡単な hex-to-BCD コードコンバータを表わします。

```
FIELD input = [in3..0] ;
FIELD output = [out4..0] ;
TABLE input => output {
0=>00; 1=>01; 2=>02; 3=>03;
4=>04; 5=>05; 6=>06; 7=>07;
8=>08; 9=>09; A=>10; B=>11;
C=>12; D=>13; E=>14; F=>15;
}
```

## ステートマシン

AMD/MMI によるとステートマシンは順番にあらかじめ決められた状態を繰り返すデジタルデバイスです。同期ステートマシンはフリップフロップを利用した論理回路です。出力を自分自信または他のフリップフロップの入力へフィードバックすることができるので、フリップフロップの入力

値は、自分自信の出力または他のフリップフロップの出力に影響されます。従って、最終的な出力は自信の前の値と他のフリップフロップの前の値に影響されます。

図 4-13 に示される CUPL ステートマシンモデルは 6 個の部品すなわち入力と組み合わせ論理、ストレージレジスタ、ステートビット、レジスタード出力、ノンレジスタード出力を使用しています。

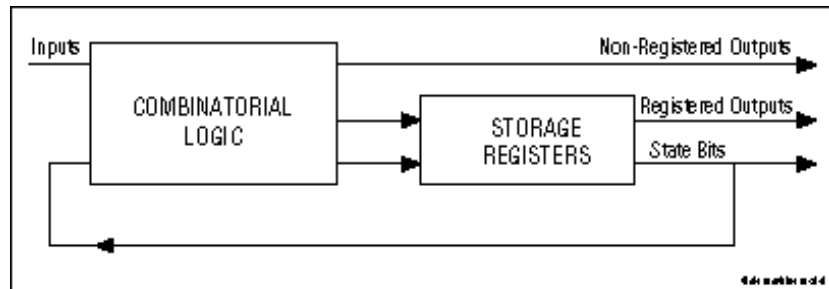


図 4-13 ステートマシンモデル

## 入力

他のデバイスからの信号をデバイスへ入力します。

## 組み合わせ論理

出力信号を作る任意の組み合わせ論理（通常 AND-OR）です。出力信号は、これらのゲートを変更して  $T_{pd}$ （伝播遅れ時間）nsec 後に現れます。 $T_{pd}$  は、入力またはフィードバックイベントの始めとノンレジスタード出力の変化の間の時間です。

## ステートビット

組み合わせ論理を駆動するためにフィードバックされるストレージレジスタの出力です。これらには、現在の状態に関する情報が含まれています。

## ストレージレジスタ

フリップフロップは、ステートマシン組み合わせ論理から入力を受け取ります。レジスタの中には、ステートビットに使用されるものもあります。その他はレジスタード出力に使用されます。レジスタード出力は、クロックパルスから  $T_{co}$ （クロックから出力までの時間）nsec 後に現れます。 $T_{co}$  はクロック信号の始めとフリップフロップ出力が現れるまでの間の遅れ時間です。

ステートマシンを実行するために、CUPL は、ステートマシンにどんな関数でも記述できるようなシンタクスを持っています。SEQUENCE キーワードによりステートマシンの出力が識別され、このキーワードに続いてステートマシンの関数の定義の命令が記述されます。SEQUENCE キーワードによりストレージレジスタやターゲットデバイスのデフォルトの出力タイプが設定されます。SEQUENCE キーワードの他に、SEQUENCED キーワードや SEQUENCEJK キーワード、SEQUENCERS キーワード、SEQUENCET キーワード



ードがあります。それぞれのキーワードは、DタイプやJ - Kタイプ、S - Rタイプ、Tタイプのフリップフロップを設定します。SEQUENCE シンタックスのフォーマットを以下に示します。

```
SEQUENCE state_var_list {  
    PRESENT state_n0  
    IF (condition1)    NEXT state_n1;  
    IF (condition2)    NEXT state_n2    OUT out_n0;  
    DEFAULT            NEXT state_n0;  
PRESENT state_n1  
    NEXT state_n2;  
.  
.  
.  
    PRESENT state_nn statements ;  
}
```

ここで

state\_var\_list は、ステートマシンブロックで使用するステートビット変数のリストです。変数リストは、フィールド変数により表現することができます。

state\_n は、ステート数で、state\_variable\_list をデコードしたあたいです。また、state\_n は、各 PRESENT 命令に固有の値である必要があります。

条件命令や next 命令、出力命令はこのセクションの以下のサブセクションで説明します。

## マルチステートマシン

CUPL シンタックスでは、同じ PLD 設計の中に複数のステートマシンを定義することができます。複数のステートマシンが定義されると、設計者は互いのステートマシン間で通信を行いたい場合があります。例えば、あるステートマシンがある状態になったら、他のステートマシンを起動するような回路です。ステートマシンの通信を実現するには2つの方法があります。すなわち、ステートビットのセット演算を使用する方法か、ステートマシン間でアクセス可能なグローバルレジスタを定義する方法です。

1つのステートマシンに、条件命令にステート番号やステート番号レンジに続いて他のステートマシン名を記述できます。他のステートマシンが特定の状態になると、条件命令が TRUE になります。複数のステートマシンでアクセスするレジスタを使用する場合、同じことが起こると TRUE です。しかし、この方法はレジスタやデバイス出力を一つ消費してしまいます。状態に応じて、他のステートマシンから情報を受け取り異なる状態になるグローバルレジスタを組み合わせて使用することができます。

## 条件命令

CONDITION シンタックスにより、組み合わせ論理の標準ブーリアン論理式を記述して設計するよりわかりやすく論理関数を設計することができます。

フォーマットを以下に示します。

```
CONDITION {  
  IF expr0 OUT var ;  
  .  
  .  
  IF exprn OUT var ;  
  DEFAULT OUT var ;  
}
```

CONDITON シンタクスは、ステートマシンシンタクスの同期条件出力命令と等価です。ただし、特定の状態を参照することは行われません。式や DEFAULT 状態に適合すると、変数が論理的に宣言されます。

### 関数の定義

FUNCTION キーワードにより、あるロジックに名前を付けてカプセルに入れて関数化し、個人的なキーワードを作成することができます。この名前は、論理式内で使用することができる関数を表わします。ユーザ定義関数のフォーマットを以下に示します。

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

body の中の命令が、関数に割り付けられます。

省略可能なパラメータを使用する場合、関数定義または参照内のパラメータの名前は、一致する必要があります。関数の本体で定義されるパラメータは、論理式内で参照されるパラメータで置き換えられます。関数を呼び出す変数は関数本体により式に割り当てられます。命令本体で割り付けが行われない場合、関数を呼び出す変数は ho の値に割り付けられます。

## 簡単なステートマシンの設計

地下鉄の回転木戸のコントローラは最も簡単なステートマシン設計です。このコントローラは、コインが入れられた信号を待って、状態が変化すると、ロックからオープンに代わります。オープン状態では、だれかが回転木戸を通貨するのを待ちます。それから、オープンからロックに変わります。この二つの状態設計は、入力としてコインが入れられたことによる信号と人が通過したことによる信号とのあいだで繰り返されます。以下の図は、二つの状態とデバイスがある状態から次の状態へ変更するパルスを示します。

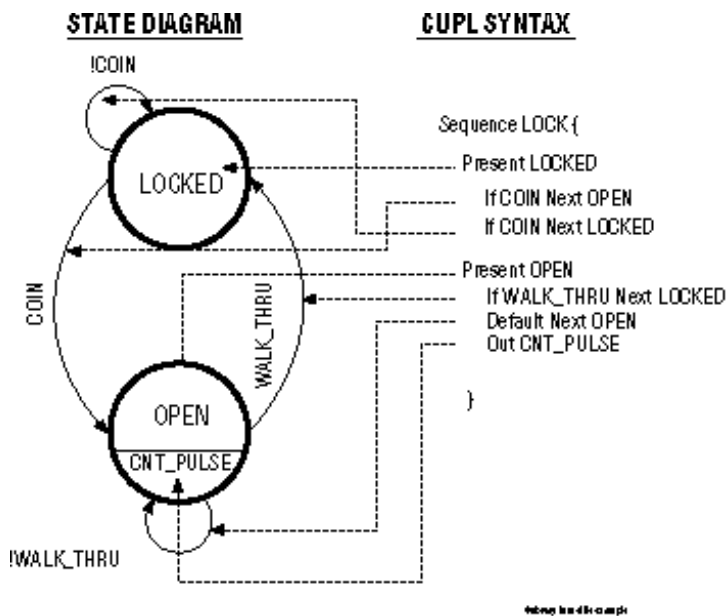


図 4-14 地下鉄の回転木戸の例

コードと設計概念の関係がわかりやすく理解できるように、CUPL ステートマシンコードに相当するものが右に表示されます。

```

TURNSTIL.PLD
Name          Turnstil;
Partno        FL00001;
Date          03/06/89;
Designer      R. Teixeira;
Company       LDI;
Location      D21;
Assembly      Example

/*****
/*
/*  CUPL が地下鉄の回転木戸コントローラを          */
/*  どのようにコンパイルするかの例です。          */

```

```

/*
/*****
/* Target Devices: P16R4
/*****

/* Inputs:  define inputs to Turnstile controller */
Pin 1 = CLK;
Pin 2 = WALK_THRU;
Pin 3 = COIN;

/* Outputs:  define outputs as active HI levels
..... */
Pin 14 = CNT_PULSE;
Pin 15 = LOCK;

/* Logic:  Subway Turnstile example expressed in CUPL */
$DEFINE LOCKED 'b'0
$DEFINE OPEN 'b'1
Sequence LOCK{

Present LOCKED
    if COIN      Next OPEN;
    if !COIN     Next LOCKED;

Present OPEN

    if WALK_THRU Next LOCKED;
    Default      Next OPEN;
    Out CNT_PULSE;
}

```

❏ 4-15 TURNSTIL.PLD

## 設計のサンプル

このセクションでは、コンパイラとシミュレーションを使用する設計例を通して段階的に説明をします。

- Step 1. 設計作業の確認
- Step 2. コンパイラソースファイルの作成
- Step 3. 式の公式化
- Step 4. ターゲットデバイスの選択
- Step 5. ピン割り付け
- Step 6. PLD ソースファイルのコンパイル
- Step 7. シミュレーションテストベクタファイルの作成
- Step 8. デバイスのシミュレーション
- Step 9. シミュレーション波形の表示

### Step 1 - 設計作業の確認

このプログラマブルロジックデバイス(PLD)設計例のシステムは、マイクロプロセッサをベースとした ROM と RAM を用いた CPU インターフェースを使用しています。図 5-1 にシステムダイアグラムを示します。

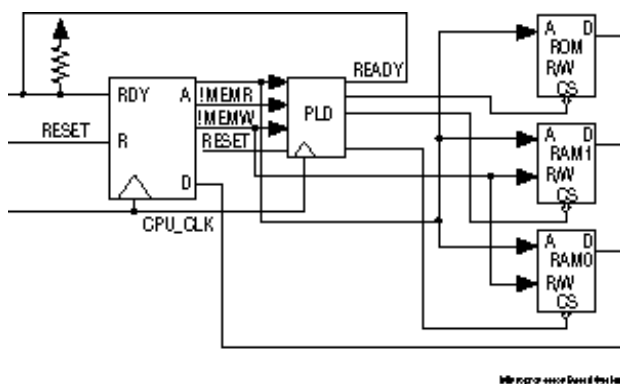


図 5-1 マイクロプロセッサをベースとしたシステム

PLD により、アドレスのデコードやタイミング制御関数による CPU と周辺装置とのフレキシブルなインターフェースを設計できます。ダイアグラムで示されるように、ROM (または PROM) を使用してシステムを制御し、2 個の静的 RAM をスクラッチパッドと補助メモリ関数に使用します。

このサンプルセクションでは、PLD の回路の目的は、メモリのマッピングに使用する CPU のアドレスをデコードしたり、CPU アドレスと CPU データストローブに基づいて ROM と RAM のチップセレクト信号を生成することです。

図 5-2 のメモリマップは、CPU のアドレススペースのどこに ROM と RAM

の領域が確保されるかを示します。

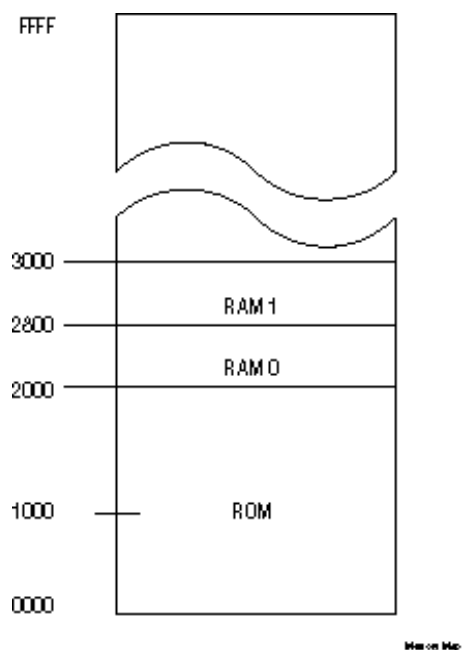


図 5-2 メモリマップ

アドレスは、メモリマップでは 16 進数で表わされます。PLD の論理回路を設計する場合、このメモリマップを使用して下さい。

ROM チップは遅いので、PLD は ROM アクセスに 1CPU クロック以上の待ち状態を生成するように設計する必要があります。

図 5-3 のタイミングダイアグラムの矢印は他の信号により影響されたり生成される信号を示します。

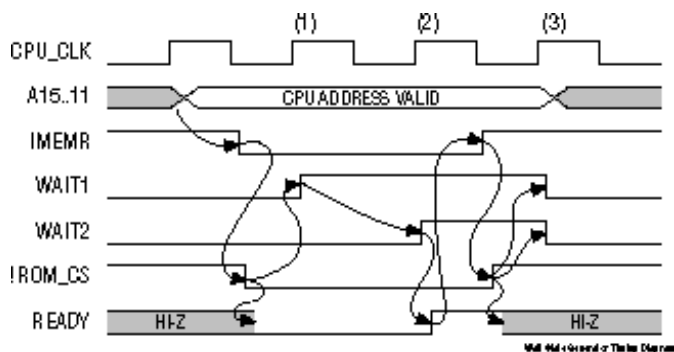


図 5-3 待ち状態ジェネレータタイミングダイアグラム

タイミングダイアグラムの演算の記述が以下に続きます。丸括弧内の番号は CLOCK 信号の立ち上がりエッジを示します。

待ち状態シーケンスは、メモリの読み取りストローブの前の CPU アドレスが有効になった時に始まります。待ち状態は、ROM のためだけに生成されるので、!MEMR 信号だけを考慮に入れる必要があります。

!MEMR ストローブが ROM に対応したアドレスでアクティブの場合、!ROM\_CS 信号が呼び出され、CPU の READY 信号を LO に駆動する(レディー状態か待ち状態かを示す)スリーステートバッファがオンされます。!ROM\_CS がアクティブになった後、CPU クロックの次の立ち上がりエッジ(1)で WAIT1 信号がセットされます。1CPU クロック後に WAIT2 信号が呼び出され(2)ます。すなわち、待ち状態期間(1CPU クロック)が完了し、CPU の READY 信号が HI に駆動されます。この信号により、CPU は読みだしサイクルを続け、!MEMR ストローブが適当な時刻で削除されます。!ROM\_CS 信号は反転されレディー信号を駆動するスリーステートバッファはディスエーブルになります。そして、CPU クロックの次の立ち上がりエッジ(3)により、WAIT1 と WAIT2 がリセットされます。待ち状態ジェネレータは、次の CPU アクセス時間のために初期化されます。

## Step 2 - コンパイラソースファイルの作成

このステップでは、PLD の設計を記述するために論理記述ファイルを作成します。この論理記述ファイルは、CUPL 論理記述言語で記述されます。論理記述ファイルは、デバイスプログラマーヘダダウンロードする設計をコンパイルするコンパイラへの入力です。

論理記述ファイルに必要なフォーマットを簡単に設定するために、アドバンスト PLD にはテンプレートファイル、TMPL.PLD があります。Text Expert で TMPL.PLD を開き SAMPLE.PLD という名前で保存して下さい。

図 5-4 は SAMPLE.PLD にあるテンプレート情報を示します。

```

      TEMPLATE FILE
Name          XXXXXX;
Partno        XXXXXX;
Date          XX/XX/XX;
Revision      XX;
Designer      XXXXXX;
Company       XXXXXX;
Assembly      XXXXXX;
Location      XXXXXX;

/*****
/* Title Block                                     */
/*                                                     */
/*                                                     */
/*****
/* Allowable Target Device                           */
/*****

/** Inputs **/

Pin          =          ;          /*
Pin          =          ;          /*
Pin          =          ;          /*

```

```

Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */

/** Outputs **/

Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */
Pin      =      ;      /*      */

/**      宣言と中間変数の定義      **/

```

図 5-4 SAMPLE.PLD のテンプレート情報

特定のヘッダーやタイトル情報、入出力ピンの指定、中間変数や論理式の記述のためにファイルは編集することができます。

ヘッダーセクションでは、XXX は会社や設計される PLD を参照する特定の情報と置き換えられます。

ヘッダーセクションの下は、コメント記号 (/\*と\*/) 使用するタイトルブロックです。タイトルブロックに設計を説明する情報を記述して下さい。

図 5-5 にこの例で入力する情報を示します。

```

SAMPLE.PLD
Name      Sample;
Partno    P9000183;
Date      07/16/87;
Revision  02;
Designer  Osann;
Company   ATI;
Assembly  PC Memory;
Location  U106;

/*****
/* このデバイスは、8Kx8 の ROM と 2Kx8 のスタティック */
/* RAM に使用するチップセレクト信号を生成します。 */

```



```

/* また、このデバイスは、システムのREADY信号を駆動し */
/* ROMのアクセス時に1CPUクロック以上の待ち状態を挿入します。 */
/*****

```

図 5-5 SAMPLED.PLD のヘッダーとタイトルブロック

### Step 3 - 式の公式化

アドレスのデコードや待ち状態ジェネレータの特定の式の入力を簡単に行なうために、まず、中間変数の式を入力して下さい。中間変数は特定のピンを表わしません。従って、中間変数はあまり意味の無い名前です。宣言や中間変数の定義のために SAMPLED.PLD ファイルに確保されたスペースに中間式を入力して下さい。

入力する最初の間中式は、アドレスバスを定義するビットフィールド宣言です。アドレスを表わすために MEMADR (メモリアドレス) という名前を使用し、以下のように式を入力して下さい。

```
FIELD MEMADR = [A15..11] ;
```

図 5-1 のシステムダイアグラムでは、スタティック RAM のチップセレクト信号はアドレスに依存しておらず MEMW または MEMR データストロープのために宣言する必要があることに注意して下さい。

スタティック RAM のチップセレクト信号のための式を簡単にするため、MEMREQ (メモリリクエスト) と呼ばれる信号を作成して下さい。以下のように入力して下さい。

```
MEMREQ = MEMW # MEMR ;
```

MEMREQ が他の式で使用される場合はいつでも、コンパイラは MEMW#MEMR を置き換えます。

図 5-3 のタイミングダイアグラムでは、ROM に対応するアドレスのデコードが !MEMR ストロープと組み合わせられて ROM のチップセレクト (ROM\_CS) を生成し、待ち状態シーケンスを初期化します。

以下のように入力して、!MEMR ストロープを組み合わせる ROM のアドレス空間の特定のアドレスをデコードする SELECT\_ROM と呼ばれる中間変数を作成して下さい。

```
SELECT_ROM = MEMR & MEMADR : [0000..1FFF] ;
```

上記の間中式を入力した後、アドレスデコードや待ち状態ジェネレータの式を入力して下さい。

アレイヘフィードバックされる ROM\_CS 信号が、待ち状態のタイミングを初期化するために使用される場合、PLD にさらに追加遅れが生じます。クロックレートが比較的遅い (4-8MHz) ので、この例では追加遅れは問題になりません。しかし、クロックレートが早い場合、同じロジックを (中間変数 SELECT\_ROM を使い) レジスタを使用した論理式に書き換えた方がベターです。

以下のようにタイプし、中間変数 SELECT\_ROM を使用して ROM のチップセレクト (ROM\_CS) を作成して下さい。

```
ROM_CS = SELECT_ROM ;
```

2 個の RAM、RAM\_CS0 と RAM\_CS1 のチップセレクトは MEMREQ とアドレスマップから取得される 16 進数の範囲内にあるアドレスバスに依存します。デコードする範囲の上限と加減のアドレスを用いて CUPL のレンジ演算を使用して下さい。以下のように入力して下さい。

```
RAM_CS0 = MEMREQ & MEMADR : [2000..27FF];  
RAM_CS1 = MEMREQ & MEMADR : [2800..2FFF];
```

次に、待ち状態を生成するための式を作成します。最初にタイミングダイアグラム (図 5-3) に示されるように、CPU クロックの次の立ち上がりエッジで設定される ROM チップの選択に必要な WAIT1 と呼ばれる信号が必要です。D タイプのフリップフロップの特性から、D 入力での論理レベルはクロック後に Q 出力に変換されます。この信号の式を以下のように入力して下さい。ここで WAIT1.D は PLD のフリップフロップの D 入力の信号を表わします。

```
WAIT1.D = SELECT_ROM & !RESET ;
```

WAIT1.D の式で、!RESET 信号が AND され式の残りの部分で AND され RESET 信号により同期リセットが実行されることに注意して下さい。

次に、WAIT1 が設定される次のクロックエッジで WAIT2 信号を WAIT1 に依存する WAIT2.D の式により作成して下さい。WAIT2.D は、ROM の CPU アクセスが終了すると次のクロックエッジでリセットされる必要があります。以下のように式を入力して下さい。

```
WAIT2.D = SELECT_ROM & WAIT1 ;
```

これにより、タイミングダイアグラム (図 5-3) の信号 SELECT\_ROM 信号が作成され、ROM がデコードされている間と MEMR データストローブがアクティブな間、スリーステートバッファがオンされることが示されます。従って、以下のように入力してスリーステート出力の式を入力して下さい。

```
READY.OE = SELECT_ROM ;
```

この式が、スリーステートバッファがその出力を実際に駆動しハイインピーダンス状態に保持するかを決める間、どちらの論理レベルに信号が駆動されるかは決定されません。READY の式により、論理レベルをどちらに駆動するかが決められます。すなわち、1CPU クロックサイクルに相当する待ち状態期間が完了するまで READY の信号はインアクティブのままです。この状態は WAIT2 が設定されるまで起こらないように、READY の式を以下のように入力して下さい。

```
READY = WAIT2;
```

## Step 4 - ターゲットデバイスの選択

式が完成すると、次のステップは、使用する PLD を決定します。ターゲットデバイスを選択するときに考慮すべき点を以下に示します。

必要な入力ピンの数

- レジスタード出力ピンとノンレジスタード出力ピンの数の比率。
- スリーステート出力制御（必要な場合）
- 各式の論理関数を実行するために必要なプロダクトタームの数

図 5-6 に PLD パッケージダイアグラムに、PAL16R4 または 82S155IFL と互換性のあるデバイスのピン配置を示します。

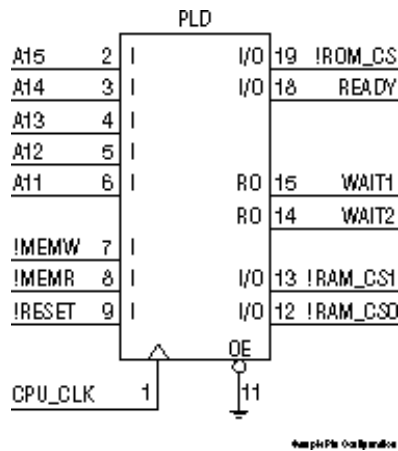


図 5-6 ピン配置のサンプル

図 5-6 のピン配置では、チップセレクト信号が 3 本、出力駆動モードの i/o タイプのピンに割り付けられています。CPU バスに接続される READY ピンは、切り替え可能なスリーステートモードで使用されます。待ち状態ジェネレータを実行するために必要な 2 個のフリップフロップは、内部でレジスタに接続される出力ピンに割り付けられます。

READY の論理関数は WAIT2 信号の論理関数と同じなので、レジスタード出力の一つを使用して READY 信号を直接駆動することができます。しかし、ターゲットデバイスのピン 11 に接続されるスリーステート出力イネーブル信号を使用する必要があります。ピン 11 は内部レジスタに接続される 4 本のピンすべてのスリーステート出力をコントロールするので、これにより他の 2 つのレジスタード出力ピンを待ち状態ジェネレータに使用することができます。

設計の発展段階での変更をすべて予測するのは困難なことなので、スリーステートコントロールを使用しないでオプションを開いたままにした方が良いでしょう。したがって、ピン 11 は、レジスタからのスリーステート出力常にイネーブルになるようにグラウンドに接続されます。

PAL16R4 は、出力に少なくとも 7 個のプロダクトタームがあります。この数はこのアプリケーションに適當です。このソケット位置のセカンドソースの IFL82S155 は、全部で 32 個のプロダクトタームを使用できます。この

数も今回のアプリケーションには適当です。

PAL16R4 デバイスはDタイプのフリップフロップしか持っていません。一方、82S155 デバイスはDまたは JK タイプのフリップフロップを使用することができます。Step3 で WAIT1 と WAIT2 に記述された式は、拡張子.D が付いているので、コンパイラはDタイプフリップフロップの配置を自動的に決めます。

## Step 5 - ピン割り付け

ピン割り付けを図 5-6 の PAL16R4 または 82S155 デバイスのピンに合わせて下さい。まず、SAMPLE.PLD の Allowable Target Device Types とラベルが付けられたコメントスペースに次のように入力して下さい。

```
pal16r4, 82s155
```

ピン割り付けを行なう時は、割り付けられる信号の極性（信号レベル）が論理スキマチックの信号と同じであることを確認して下さい。

図 5-7 で示すようにピン割り付けを行なって下さい。

SAMPLE PIN ASSIGNMENTS			
/** Inputs **/			
Pin 1	= cpu_clk ;	/* CPU clock	*/
Pin [2..6]	= [a15..11] ;	/* CPU Address Bus	*/
Pin [7,8]	= ![memw,memr] ;	/* Memory Data Strobes	*/
Pin 9	= reset ;	/* System Reset	*/
Pin 11	= !oe ;	/* Output Enable	*/
/** Outputs **/			
Pin 19	= !rom_cs ;	/* ROM Chip select	*/
Pin 18	= ready ;	/* CPU Ready signal	*/
Pin 15	= wait1 ;	/* Start wait state	*/
Pin 14	= wait2 ;	/* End wait state	*/
Pin [13,12]	= ![ram_cs1..0] ;	/* RAM Chip selects	*/

図 5-7 SAMPLE.PLD

ピン割り付けが完了したら、extra pin=;の行を削除して下さい。

図 5-8 に、完成した論理記述ファイル、SAMPLE.PLD を示します。

SAMPLE.PLD	
Name	Sample;
Partno	P9000183;
Date	07/16/87;
Revision	02;
Designer	Osann;
Company	ATI;
Assembly	PC Memory;
Location	U106;

```

/*****
/* このデバイスは、8Kx8 の ROM と 2Kx8 のスタティック */
/* RAM に使用するチップセレクト信号を生成します。 */
/* また、このデバイスは、システムの READY 信号を駆動し */
/* ROM のアクセス時に 1CPU クロック以上の待ち状態を挿入 */
/* します。 */
/*****
/*****
/** Allowable Target Device Types : PAL16R4, 82S155 **/
/*****
/** Inputs **/
Pin 1      = cpu_clk;          /* CPU clock */
Pin [2..6] = [a15..11];       /* CPU Address Bus */
Pin [7,8]   = ![memw,memr] ;
                /* Memory Data Strobes (active low)*/
Pin 9       = reset;          /* System Reset */
Pin 11      = !oe; /* Output Enable (active low) */

/** Outputs **/
Pin 19      = !rom_cs; /* ROM chip select (active low)*/
Pin 18      = ready ;      /* CPU ready */
Pin 15      = wait1 ;      /* Wait state 1 */
Pin 14      = wait2 ;      /* Wait state 2 */
Pin [13,12] = ![ram_cs1..0] ;
                /* RAM chip select (active low) */

/* Declarations and Intermediate Variable Definitions */
Field memadr = [a15..11] ; /* Give the address bus */
                /* the Name "memadr" */

memreq = memw # memr ; /* Create the intermediate */
                /* variable "memreq" */
select_rom = memr & memadr:[0000..1FFF] ; /* = rom_cs */

/** Logic Equations **/
rom_cs = select_rom;
ram_cs0 = memreq & memadr:[2000..27FF] ;
ram_cs1 = memreq & memadr:[2800..2FFF] ;

/* read as: when select_rom is true and reset is false */
wait1.d = select_rom & !reset ;

/* read as: when when select_rom is true and wait1 is true
*/
/* Synchronous Reset */
wait2.d = select_rom & wait1 ; /* wait1 delayed */

ready.oe = select_rom ; /* Turn Buffer off */
ready = wait2 ; /* end wait */

```

## Step 6 - PLD ソースファイルのコンパイル

このステップでは、論理記述ファイル SAMPLE.PLD を、ターゲットデバイス PAL16R4 で使用できるようにコンパイルします。

ターゲット PLD とコンパイルオプションを Configure Advanced PLD ダイアログボックスで指定して下さい。以下のオプションをイネーブルにして下さい。

- Absolute ABS - SAMPLE.ABS を作成します。後でシミュレータにより使用されるアブソリュートファイルです。（このファイルはステップ 7 で必要です。）このファイルには、デバイスの内部ヘブプログラムされる論理関数が記述されています。シミュレータはこの記述とユーザが作成した入力ファイルのテストベクタとを比較し入力ファイルの応答ベクタが正しい刺激ベクタの応答であるかどうかを確認します。
- Fuse Plot in Doc File and Equations in Doc File - SAMPLE.DOC を作成します。このファイルはドキュメンテーションファイルです。このファイルには、中間変数と出力ピン変数の完全に展開されたプロダクトタームやヒューズプロットとチップダイアグラムが記述されます。
- Error List LST - SAMPLE.LST を作成します。このファイルは、リストファイルです。記述ファイルをサイド作成したものです。ただし、行番号が追加され、コンパイル中に生成されたエラーメッセージがファイルの最後に付け加えられます。
- JEDEC - SAMPLE.JED を作成します。このファイルは、デバイスプログラマーへダウンロードする JEDEC ファイルです。このファイルには、ヒューズパターンが記述されます。テストベクタは、シミュレーション中に JEDEC ファイルに加えられます。

SAMPLE.JED というファイル名は、論理記述ファイルのヘッダー情報セクションの NAME フィールドで決められます。ひとつのデバイスだけがファイルに記述されている場合、同じ名前をファイル名に使用して下さい。（この場合、SAMPLE）

現在のドキュメントの最新のファイルがコンパイラのソースファイルとなるように、PLD ソースファイルを保存してから、コンパイラを実行して下さい。

コンパイラを設定が完了したら、Configure Advanced PLD ダイアログボックスの OK をクリックして下さい。

コンパイルするには、SAMPLE.PLDWO現在のドキュメントにし PldTools ツールバーの Compile ボタンを押して下さい。

The Advanced PLD - Compiling ダイアログボックスがポップアップ表示されます。Info ボタンを押して下さい。以下のメッセージがダイアログボック

スに表示されます。各コンパイラモジュールが完了するまでに要した時間が表示されます。実際の時間は使用されるシステムに応じて変わります。

SAMPLE.LST と SAMPLE.DOC ファイルを Text Expert にオープンして下さい。

リストファイル、SAMPLE.LST は、ソースファイルと同じです。ただし、行番号とエラーメッセージが追加されています。行番号により、エラーが発生した場合、エラーの位置を速く見つけることができます。

```

                                SAMPLE.LST
CUPL Version 4.XX  Serial # XX-XXX-XXXX
Copyright (C) 1996 Protel International
CREATED Thur Jan 14 08:42:12 1990

LISTING FOR LOGIC DESCRIPTION FILE: sample.pld;

1:Name                        Sample;
2:Partno                      P9000183;
3>Date                        07/16/87;
4:Revision                    02;
5:Designer                    Osann;

6:Company                     ATI;
7:Assembly                    PC Memory;
8:Location                    U106;
9:
10:/*****
11:/* This device generates chip select signals for one */
12:/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
13:/* the system READY line to insert a wait-state of at
14:/* least one cpu clock for ROM accesses */
15:/*****
16:/*****
17:/** Allowable Target Device Types : PAL16R4, 82S155 */
18:/*****
19:/**   Inputs   */
20:
21:Pin 1      = cpu_clk    ;      /* CPU clock */
22:Pin [2..6] = [a15..11] ;      /* CPU Address Bus */
23:Pin [7,8]  = ![memw,memr] ; /* Memory Data Strobes */
24:Pin 9      = reset     ;      /* System Reset */
25:Pin 11     = !oe       ;      /* Output Enable */
26:
27:/**   Outputs  */
28:
29:Pin 19     = !rom_cs    ;      /*
30:Pin 18     = ready     ;      /*
31:Pin 15     = wait1     ;      /*
```

```

32:Pin 14      = wait2      ;      /*                      */
33:Pin [13,12] = ![ram_cs1..0] ; /*                      */
34:
35:/* Declarations and Intermediate Variable Definitions
*/
36:
37:Field memadr = [a15..11] ; /* Give the address bus */
38:                      /* the Name "memadr" */
39:
40:memreq = memw # memr ; /* Create the intermediate */
41:                      /* variable "memreq" */
42:
43:select_rom = memr & memadr:[0000..1FFF] ; /* = rom_cs
*/
44:
45:/** Logic Equations **/
46:
47:rom_cs = select_rom;
48:ram_cs0 = memreq & memadr:[2000..27FF] ;
49:ram_cs1 = memreq & memadr:[2800..2FFF] ;
50:wait1.d = select_rom & !reset ;
51:                      /* Synchronous Reset */
52:wait2.d = select_rom & wait1 ; /* wait1 delayed */
53:ready.oe = select_rom ; /* Turn Buffer off */
54:ready = wait2 ; /* end wait */
Jedec Fuse Checksum      (4D50)
Jedec Transmit Checksum  (E88F)

```

#### 図 5-9 SAMPLE.LST

図 5-10 に、コンパイラにより作成されたドキュメンテーションファイルを示します。

```

                                SAMPLE.DOC
*****
                                Sample
*****
CUPL                4.XX Serial# XX-XXX-XXXX
Device              pl6r4 Library DLIB-d-26-11
Created              Mon Aug 20 10:48:32 1990
Name                 Sample;
Partno               P9000183;
Date                 04/1/90;
Revision             02;
Designer             Osann;
Company              ATI;
Assembly             PC Memory;
Location             U106;
=====
                                Expanded Product Terms

```



```

=====
wait1.d =>
    !memr
    # a15
    # a14
    # a13
    # reset

select_rom =>
    !a13 & !a14 & !a15 & memr

wait2.d =>
    !memr
    # a15
    # a14
    # a13
    !wait

memadr =>
    a15,a14,a13,a12,a11

ready =>
    !wait2

ready.oe =>
    !a13 & !a14 & !a15 & memr
rom_cs =>
    !a13 & !a14 & !a15 & memr
memreq =>
    memw
    # memr

ram_cs0 =>
    !a11 & !a12 & !a13 & !a14 & !a15 & memw
    # !a11 & !a12 & !a13 & !a14 & !a15 & memr

ram_cs1 =>
    a11 & !a12 & a13 & !a14 & !a15 & memw
    # a11 & !a12 & a13 & !a14 & !a15 % memr

rom_cs.oe =>
    1

ram_cs0.oe =>
    1

ram_cs1.oe =>
    1

```

=====							
Symbol Table							
=====							
Pin	Variable				Pterms	Max	Min
<u>Pol</u>	<u>Name</u>	<u>Ext</u>	<u>Pin</u>	<u>Type</u>	<u>Used</u>	<u>Pterms</u>	
	Level						
	wait1		15	V	-	-	-
	wait1	d	15	X	5	8	1
	all		6	V	-	-	-
	select_rom		0	I	1	-	-
	wait2		14	V	-	-	-
	wait2	d	14	X	5	8	1
	a12		5	V	-	-	-
	a13		4	V	-	-	-
	a14		3	V	-	-	-
	a15		2	V	-	-	-
!	oe		11	V	-	-	-
!	memr		8	V	-	-	-
	memadr		0	F	-	-	-
	ready		18	V	1	7	1
	ready	oe	18	X	1	1	1
!	memw		7	V	-	-	-
	cpu_clk		1	V	-	-	-
!	rom_cs		19	V	1	7	1
	reset		9	V	-	-	-
	memreq		0	I	2	-	-
!	ram_cs0		12	V	2	7	1
!	ram_cs1		13	V	2	7	1
	rom_cs	oe	19	D	1	1	0
	ram_cs0	oe	12	D	1	1	0
	ram_cs1	oe	13	D	1	1	0
LEGEND F : field D : default M : extended node							
N : node I : Intermediate variable T : function							
V : variable X : extended variable U : undefined							
=====							
Fuse Plot							
=====							
Pin #19							
0000	-----						
0032	-x---x---x-----x-----						
0064	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
0096	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
0128	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
0160	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
0192	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						
0224	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx						

```

Pin #18
0256 -x---x---x-----x-----
0288 -----x-----
0320 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0352 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0384 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0416 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0448 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0480 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #17
0512 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0544 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0578 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0608 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0640 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0672 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0704 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0738 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #16
0768 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0800 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0832 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0864 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0896 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0928 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0960 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0992 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #15
1024 -----x-----
1056 x-----
1088 ---x-----
1120 -----x-----
1152 -----x---
1184 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1216 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1248 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #14
1280 -----x-----
1312 x-----
1344 ---x-----
1378 -----x-----
1408 -----x-----
1440 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1472 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1504 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

Pin #13

```

1536 -----
1568 -x---x--x---x--x---x-----
1600 -x---x--x---x--x-----x-----
1632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

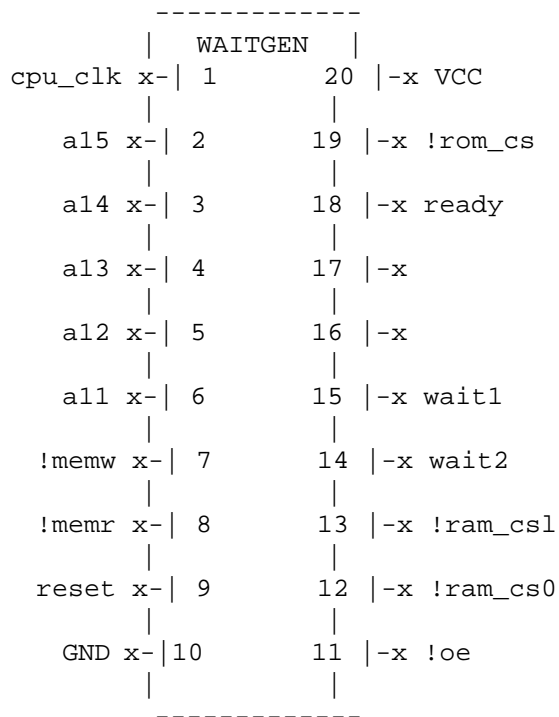
Pin #12

```

1792 -----
1824 -x---x--x---x--x---x-----
1856 -x---x--x---x--x-----x-----
1888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

=====  
Chip Diagram  
=====



特定のデバイスに対してピンリストでアクティブハイと宣言された出力変数がある場合、PAL16R4 の固定反転出力バッファは、コンパイラによりドモルガンの定理が実行されるので、WAIT1.D と WAIT2.D の展開されたプロダクトタームは、5 つのプロダクトタームで示されます。

## Step 7 - シミュレーションテストベクタファイルの作成

このステップでは、PAL16R4 デバイス用にコンパイルされた設計をシミュレーションにより検証します。この段階を踏んでから、ロジックプログラマーへダウンロードするとロジックの間違いが発生する率が減ります。

このセクションでは、ソース仕様ファイル、SAMPLE.SI が作成されます。このファイルには、シミュレータの入力に使用されるテストベクタが記述されます。シミュレータはテストベクタ入力と予想される出力と、コンパイラの演算中に作成される SAMPLE.ABS に記述された実際の値とフラグとの違いを比較します。

図 5-11 にソース仕様ファイルのサンプルを示します。

```

Name                Sample;
Partno              P9000183;
Date                07/16/87;
Revision            02;
Designer            Osann;
Company             ATI;
Assembly            PC Memory;
Location            U106;

/*****
/* This device generates chip select signals for one */
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
/* the system READY line to insert a wait-state of at */
/* least one cpu clock for ROM accesses                */
*****/

ORDER:
    cpu_clk, %2, a15, %2, a14, %2,
    a13, %2, a12, %2, a11, %2,
    !memw, %2, !memr, %2, reset, %2, !oe,
    %4, !ram_cs1, %2, !ram_cs0, %2, !rom_cs, %2,
    wait1, %2, wait2, %2, ready;

VECTORS:
/* 123456-leave six blanks to allow for numbers in .SO
file */
$msg "      Power On Reset                                ";
      O X X X X X 1 1 1 0      H H H * * Z
$msg "      Reset Flip Flops                                ";
      C X X X X X 1 1 0 0      H H H L L Z
$msg "      Write RAM0                                       ";

```

```

$msg "      0 0 0 1 0 0 0 1 0 0      H L H L L Z      ";
      Read RAM0

$msg "      0 0 0 1 0 0 1 0 0 0      H L H L L Z      ";
      Write RAM1

$msg "      0 0 0 1 0 1 0 1 0 0      L H H L L Z      ";
      Read RAM1

$msg "      0 0 0 1 0 1 1 0 0 0      L H H L L Z      ";
      Begin ROM read

$msg "      0 0 0 0 0 0 1 0 0 0      H H L L L L      ";
      Two clocks for wait state, Then drive READY High
";
$repeat 2;
      C 0 0 0 0 0 1 0 0 0      H H L * * *
$msg "      End ROM Read      ";
      0 0 0 0 0 1 1 0 0      H H H H H Z
$msg "      End ROM Read      ";
      C 0 0 0 0 0 1 1 0 0      H H H L L Z

```

図 5-11 SAMPLE.SI

ソース仕様ファイルには、ヘッダー情報とタイトルブロックと、ORDER 命令、VECTORS 命令の 3 つの主な部分があります。

SAMPLE.SI は、同じ SAMPLE.PLD と同じヘッダー情報を持ち、現在のリビジョンレベルを含む適切なファイルがお互いに比較されることを確認する必要があります。従って、Text Expert では、SAMPLE.PLD を SAMPLE.SI として保存し SAMPLE.SI のすべてを削除して下さい。ただし、ヘッダーとタイトルブロックは残して下さい。図 5-12 に結果を示します。

```

Name      Sample;
Partno    P9000183;
Date      07/16/87;
Revision  02;
Designer  Osann;
Company   ATI;
Assembly  PC Memory;
Location  U106;

/*****
/* This device generates chip select signals for one */
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
/* the system READY line to insert a wait-state of a */
/* least one cpu clock for ROM accesses              */
*****/

```

図 5-12 SAMPLE.SI のヘッダー情報

ORDER 命令では、テストベクタにインクルードされる SAMPLE.PLD からの入出力変数が一覧で表示されます。テスト変数で使用される順番に変数が並べられます。すなわち、クロック変数、CPU\_CLK が最初に置かれ、続いて他の入力変数が置かれます。出力変数は右に置いて下さい。変数はコ

ンマで区切って下さい。％記号を使用して変数間にスペースを挿入して下さい。すなわち、各変数の間にはスペースを 2 つ置き、一覧の最後の入力変数!OE の間と最初の出力変数!RAM\_CS1 の間には、スペースを 4 つ置いて下さい。ORDER 命令を以下のように入力して下さい。

```
ORDER:
    CPU_CLK, %2, A15, %2, A14, %2,
    A13, %2, A12, %2, ALL, %2,
    !MEMW, %2, !MEMR, %2, RESET, %2, !OE,
    %4, !RAM_CS1, %2, !RAM_CSO, %2, !ROM_CS, %2
    WAIT1, %2, READY;
```

ORDER 命令に続いて、11 個のテストベクタ (図 5-11 参照) がある関数テーブルを作成する VECTORS 命令を入力して下さい。ベクタを簡単に理解するために、テストベクタのヘッダーが作成され、.SO ファイルの ORDER 命令のすぐ後に置かれます。テストベクタは ORDER 命令に現れる信号名で構成され、ベクタセクションが読みやすいように配置されます。

さて、テストベクタを入力して下さい。値を各入力変数に入力し、出力変数に予想される値を入力して下さい。

このガイドのテスト仕様ソースファイルの作成のセクションを参照して下さい。

\$MSG ディレクティブを仕様して、関数によりテストされるデバイス関数を記述して下さい。上記の ORDER 命令により、テストベクタを作成する場合、スペーシングが指定されます。例えば、最初のベクタ、Power On Reset は、以下のように入力して作成して下さい。

```
$msg " Power On Reset      ";
      0 X X X X X 1 1 1 0 H H H * * Z
```

デバイスの中には電源の投入 X になったりまた H や L になるデバイスもあるので、出力値(\*)が WAIT1 や WAIT2 で使用されシミュレータにレジスタの電源の投入を指示することに注意して下さい。アスタリスクを使用すると統一的なシミュレーションファイルを作成することができます。

テストベクタの残りの部分は、図 5-13 で示されるように入力して下さい。

```
/* 123456-leave six blanks to allow for numbers in .SO
file */
$msg "      Power On Reset      ";
      O X X X X X 1 1 1 0   H H H * * Z
$msg "      Reset Flip Flops      ";
      C X X X X X 1 1 0 0   H H H L L Z
$msg "      Write RAM0      ";
      0 0 0 1 0 0 0 1 0 0   H L H L L Z
$msg "      Read RAM0      ";
      0 0 0 1 0 0 1 0 0 0   H L H L L Z
$msg "      Write RAM1      ";
      0 0 0 1 0 1 0 1 0 0   L H H L L Z
$msg "      Read RAM1      ";
```

```

          0 0 0 1 0 1 1 0 0 0   L H H L L Z
$msg "      Begin ROM read                                     ";
          0 0 0 0 0 0 1 0 0 0   H H L L L L
$msg " Two clocks for wait state, Then drive READY High
";
$repeat2;
          C 0 0 0 0 0 1 0 0 0   H H L * * *
$msg "      End ROM Read                                       ";
          0 0 0 0 0 0 1 1 0 0   H H H H H Z
$msg "      End ROM Read                                       ";
          C 0 0 0 0 0 1 1 0 0   H H H L L Z

```

図 5-13 テストベクタ

テストベクタファイルの\$REPEAT ディレクティブにより、8 番目のベクタが2 回繰返されます。WAIT1 と WAIT2、READY の8 番目のベクタのアスタリスクにより、シミュレータは入力を基に出力を計算し出力ファイルに結果を記述します。

クロック変数、CPU\_CLK の値は、あるベクタでは0 でその他ではC になります。0 の値になるクロックは作動しません。C の値によりシミュレータはベクタの入力値を検証しクロック前に内部的にフィードバックされるレジスタード出力の以前のベクタに注目します。それから、クロックが適用されるとシミュレータは、レジスタード出力やノンレジスタード出力の予想される出力を計算します。

VECTORS 命令の入力が完了したら、ファイルを保存して下さい。下記のステップはシミュレータを使用してシミュレーションを実行します。

## Step 8 - デバイスのシミュレーション

シミュレータが実行されると、SAMPLE.SO ファイルが作成されます。このファイルには、シミュレーションの結果が保存されます。エラーメッセージは、エラーの信号名と一緒に各ベクタに続いて一覧で記述されます。

シミュレータを実行するには、SAMPLE.PLD をカレントドキュメントにし、PldTools ツールバーの Simulate ボタンを押して下さい。

Advanced PLD-Simulate ダイアログボックスで Info ボタンを押すと、シミュレーション情報が表示されます。View Results チェックボックスをイネーブルにするとシミュレーション結果を Waveform Editor へ自動的にロードします。

SAMPLE.SO は ASCII ファイルです。従って、Text Expert で開くことができます。

図 5-14 に SAMPLE.SO の内容を示します。

```

          SAMPLE.SO
CSIM:      CUPL Simulation Program
Version 4.XX Serial # XX-XXX-XXXX
copyright (c) 1996 Protel International

```



CREATED Thur Aug 20 09:34:16 1990

1: Name Sample;  
2: Partno P9000183;  
3: Date 07/16/87;  
4: Revision 02;  
5: Designer Osann;  
6: Company ATI;  
7: Assembly PC Memory;  
8: Location U106;

9:

10:

/\*  
11: /\*This device generates chip select signals for one\*/  
12: /\*8Kx8 ROM and two 2Kx8 static RAMs. It also drives\*/  
13: /\*the system READY line to insert a wait-state of  
at\*/

14: /\*least one cpu clock for ROM accesses \*/

15:

/\*  
/  
/

16:

17: ORDER:

18: cpu\_clk, %2, a15, %2, a14, %2,

19: a13, %2, a12, %2, a11, %2,

20: !memw, %2, !memr, %2, reset, %2, !oe,

21: %4, !ram\_csl, %2, !ram\_cs0, %2, !rom\_cs, %2,

22: wait1, %2, wait2, %2, ready;

23:

#### Simulation Results

```

                                     ! !
c                                     r r !
p                                     a a r
u                                     m m o w w r
-                                     - - m a a e
c a a a a a e e s ! c c _ i i a
l 1 1 1 1 1 m m e o s s c t t d
k 5 4 3 2 1 w r t e 1 0 s 1 2 y
```

#### Power On Reset

0001: O X X X X X 1 1 1 0 H H H X X Z

#### Reset Flip Flops

0002: C X X X X X 1 1 0 0 H H H L L Z

#### Write RAM0

0003: 0 0 0 1 0 0 0 1 0 0 H L H L L Z

#### Read RAM0

0004: 0 0 0 1 0 0 1 0 0 0 H L H L L Z

#### Write RAM1

0005:	0	0	0	1	0	1	0	1	0	0	L	H	H	L	L	Z
	Read RAM1															
0006:	0	0	0	1	0	1	1	0	0	0	L	H	H	L	L	Z
	Begin ROM read															
0007:	0	0	0	0	0	0	1	0	0	0	H	H	L	L	L	L
	Two clocks for wait state, Then drive READY High															
0008:	C	0	0	0	0	0	1	0	0	0	H	H	L	H	L	L
0009:	C	0	0	0	0	0	1	0	0	0	H	H	L	H	H	H
	End ROM Read															
0010:	0	0	0	0	0	0	1	1	0	0	H	H	H	H	H	Z
	End ROM Read															
0011:	C	0	0	0	0	0	1	1	0	0	H	H	H	L	L	Z

図 5-14 SAMPLE.SO

SAMPLE.SO を、図 5-11 の SAMPLE.SI と比較して下さい。\$REPEAT ディレクティブの結果としてベクタ 8 と 9 が作成され、シミュレータにより SAMPLE.SI のアスタリスクが WAIT1 や WAIT2、READY 信号の適切な論理レベル（HまたはL）に置き換えられていることに注意して下さい。

シミュレーションが完了すると、コンパイラの実行中（STEP6）に作成される JEDEC ファイルにテストベクタが追加されます。Configure Advanced PLD ダイアログボックスでイネーブルにされた JEDEC オプションでシミュレーションを再び実行して下さい。

図 5-15 に SAMPLE.JED の内容を示します。この時点で、このファイルにはプログラム情報とテスト情報が記述されています。

SAMPLE.JED	
CUPL	4.XX Serial# XX-XXX-XXXX
Device	p16r4 Library DLIB-d-26-11
Created	Thur Aug 20 09:52:02 1990
Name	Sample
Partno	P9000183
Revision	02
Date	12/16/89
Designer	Osann
Company	ATI
Assembly	PC Memory;
Location	U106;
*QP20	
*QF2048	
*G0	
*F0	
*L00000	11111111111111111111111111111111
*L00032	1011101110111111111111111110111111
*L00256	1011101110111111111111111110111111
*L00288	111111111111111111111111111011111111
*L01024	111111111111111111111111111011111111
*L01056	0111111111111111111111111111111111
*L01088	1111011111111111111111111111111111

```

*L01120 11111111011111111111111111111111
*L01152 111111111111111111111111111110111
*L01280 111111111111111111111111101111111
*L01312 01111111111111111111111111111111
*L01344 11110111111111111111111111111111
*L01376 11111111011111111111111111111111
*L01408 11111111111111111111011111111111
*L01536 11111111111111111111111111111111
*L01568 10111011011110110111101111111111
*L01600 10111011011110110111111110111111
*L01792 11111111111111111111111111111111
*L01824 10111011011110111011101111111111
*L01856 10111011011110111011111110111111
*C4D50
*V0001 0XXXXX111N0HHXXXXZHN
*V0002 CXXXXX110N0HLLXXZHN
*V0003 000100010N0LHLLXXZHN
*V0004 000100100N0LHLLXXZHN
*V0005 000101010N0HLLXXZHN
*V0006 000101100N0HLLXXZHN
*V0007 000000100N0HLLXXLLN
*V0008 C00000100N0HLLHXXLLN
*V0009 C00000100N0HHHHXXHLN
*V0010 000000110N0HHHHXXZHN
*V0011 C00000110N1HLLXXZHN
*3152

```

図 5-15 テストベクタのある SAMPLE.JED

## Step 9 - シミュレーション波形の表示

波形の組みとしてシミュレーション結果を観察するには、SAMPLE.SO ファイルを Waveform Editor を用いて開いて下さい。これをいつでも行なうには；

- File-Open を選択して下さい。
- Open Document ダイアログボックスで Editor を Wave に設定して下さい。Wave が一覧の中にない場合、Wave Server がインストールされていないこと意味します。インストレーションセクションのこのサーバーのインストールに関する説明を参照して下さい。
- Type を PLD Simulation ファイル (\*.so) に設定して下さい。
- SAMPLE.SO を選択し OK をクリックして開いて下さい。
- シミュレーション結果が波形の組みとして、スプレッドシート形式のウィンドウに表示されます。Waveform Editor の使用法についての簡単な説明は、A Quick Tour of AdvancedPLD のセクションを参照して下さい。

## 概要

このセクションでは、PLD ソースファイルとシミュレータテストベクタを作成しコンパイルする機会を与えます。

- テンプレートファイルを使用して下さい。
- 中間式や論理式を書いて設計を記述して下さい。
- コンパイラを起動して下さい。
- テスト仕様ファイルを作成しコンパイルして設計を検証して下さい。
- シミュレータを実行し論理設計のシミュレーションを実行して下さい。
- シミュレーション波形を観察して下さい。

## 設計例

このセクションでは設計例に沿って、CUPL 言語を使用してどのように設計を記述するかを説明します。設計例を以下に示します。

- 例 1. 簡単なゲート (GATES.PLD)
- 例 2. TTL 変換 (WGTTL.PLD)
- 例 3. 2 ビットカウンタ (FLOPS.PLD)
- 例 4. ステートマシンシンタクスを使用したデカードアップ/ダウンカウンタ (COUNT10.PLD)
- 例 5. セグメントのディスプレイデコーダ (HEXDISP.PLD)
- 例 6. ロード機能とリセット機能のある 4 ビットカウンタ

各設計の論理記述ファイルを括弧内に示します。これらのファイルはコンパイラに入力してドキュメントやダウンロードファイルを作成することができます。

それぞれのテスト仕様ファイルは(ファイル名.SI)は、シミュレータで設計を確認できるようにそれぞれの論理記述ファイルについてあります。

## 例 1 - 簡単なゲート

このセクションでは、PLD の簡単なゲートプログラムの作成について詳しく説明します。以下のインプリメントされる設計を示します。

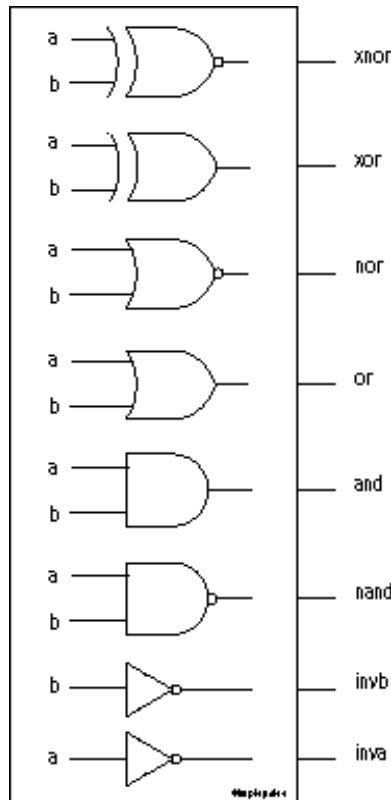


図 3-16 簡単なゲート

この設計は、簡単なゲートの出力を作成するロジックです。出力は、ゲートの関数を表わすラベルが付けられています。例えば、AND ゲートは出力に AND とラベルが付けられます。

図 4-17 に、この設計を記述する CUPL ソースファイル(アドバンスド PLD パッケージの中の GATES.PLD)を示します。

```
GATES.PLD
Name      Gates;
Partno    CA0001;
Date      07/16/87;
Designer  G Woolheiser;
Company   ATI;
Location  San Jose, CA.;
Assembly  Example

/*****
```

```

/
/*
/* This is an example to demonstrate how Advanced PLD */
/* compiles simple gates */
/*
/*****
/* Target Devices: P16L8, P16P8, EP300, and 82S153 */
/*****

/* Inputs:  define inputs to build simple gates */
Pin 1 = a;
Pin 2 = b;

/* Outputs:  define outputs as active HI levels

For PAL16L8 and PAL16LD8, De Morgan'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 */

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 */

```

図 4-17 簡単なゲートのソースファイル(GATES.PLD)

ファイルの最初の部分は、互換性のある PLD や設計されている関数の説明などの記述です。まず、コンパイラにより出力ファイルの名前として使用される名前の行があります。Partno は会社の固有の部品番号を表わします。部品番号はターゲット PLD のタイプではありません。Data はコンパイル時のデータを示します。データは、現在のものに常に更新されます。Designer は、設計者の名前または設計チームの名前です。Assembly はアッセンブリ名または PLD が使用される PC 基盤の名前を表わすために使用されます。必要に応じて、省略形の ASSY を使用して下さい。Location は、PLD が配置される PC 基盤上の座標を表わすために使用されます。省略形の LOC も

使用することができます。

ピン宣言は設計ダイアグラムの入力と出力に対応して行われます。この例でのゲートには、ゲートに接続される 2 つの入力が必要です。a と b は入力ピンの名前です。次に出力ピンに名前が割り付けられます。付けられた名前は関数の動作を表わしています。このように関数の動作を表わす名前を付けることを推奨します。これにより、ファイルのデバッグ時や更新時に内容が解りやすくなります。

ファイルの Logic セクションでは、式によりゲートが記述されます。ブーリアンシンタックスを使用して各出力ピンが入力ピン a と b の関数の出力として指定されます。

固定の反転バッファを持つ PAL16L8 と PAL16LD8 デバイスでは、ピンリストの中でアクティブハイで宣言されているので、コンパイラはドモルガンの定理を適用して出力をすべて反転します。例えば、アクティブハイとして宣言されている出力ピンの OR ゲートの以下のような式はコンパイラにより反転され、

$$\text{or} = a \# b ;$$

以下のような一つの展開されたプロダクトタームになります。(ドキュメンテーションファイルで示されるように)

$$\text{or} \Rightarrow !a \& !b$$

前に示された今回使用されるデバイスは、デバイスを決める基準に合致するために使用されます。すなわち、入力と出力のピン数が適当であり、スリーステートコントロールを持ち、レジスタードとノンレジスタードピンやデバイスが扱うことのできるプロダクトタームの数が適当であるということです。

### ソースファイルのコンパイル

このセクションでは、論理記述ファイル(GATES.DOC)がコンパイルされ、シミュレータにより使用されるドキュメンテーションファイルやデバイスプログラマヘダウンロードされるダウンロードファイルが作成されます。

GATES.PLD をコンパイルするには:

- Configure Advanced PLD ダイアログボックスで、以下のオプションをイネーブルして下さい。

Equations in Doc File	このオプションにより、コンパイラにより作成されたリストを見ることができるよう、展開された論理式や変数シンボルの表、プロダクトタームの使用状況が記述されたドキュメンテーションファイル(GATES.PLD)が作成されます。
-----------------------	---

Fuse Plot in Doc File	このオプションにより、.DOC ファイルにフューズマップ情報が追加されます。
-----------------------	--

Absolute ABS.	このオプションにより、シミュレータにより使用される GATES.ABS ファイルが作成されます。
---------------	--



Jedec. このオプションにより、デバイスプログラマへの入力ファイルやシミュレータへの入力ファイルとして使用される GATES.JED が作成されます。

- Change ボタンを押してターゲットデバイスを指定して下さい。Target Device ダイアログボックスでは、Device Type 19 と選択されたデバイス P16L8 がハイライト表示されます。
- OK ボタンを押して、ダイアログボックスを閉じて下さい。
- .PLD ファイル(GATES.PLD)をカレントドキュメントにして下さい。
- PldTools ツールバーの Compile ボタンを押してコンパイルプロセスを開始して下さい。

ターゲットデバイスは PAL16L8 でソースファイルは GATES.PLD です。出力ファイルは、GATES.DOC、GATES.ABS、GATES.JEC です。

コンパイルが終了すると、GATES.DOC を開いて下さい。GATES.DOC には作成されたリストが記述されています。これにより、コンパイラが論理式をどのように展開するかが解ります。

コンパイラレポートエラーを見る方法

- GATES.PLD を編集し、割付命令のどれかの終わりに記述されているセミコロンを削除して下さい。
- ファイルを保存して下さい。
- エラーリスティングファイルを作成するには、Configure Advanced PLD ダイアログボックスの Error list LST format をイネーブルして、コンパイラを実行して下さい。

コンパイラが終了したら、GATES.LST ファイルを調べて下さい。エラーに続いてエラー行が挿入されていることと、各行の先頭に行番号が付けられていることに注意して下さい。

アドバンスド PLD により、オープンされた PLD ファイルのエラーがハイライト表示されます。

設計のシミュレーション

このセクションでは、GATES.PLD 論理設計がアドバンスド PLD シミュレータにより視ミュレートされます。コンパイル中に作成された GATES.JED ファイルにテストベクタが付けられます。

テスト仕様ソースファイル(ファイル名.SI)が、シミュレータへの入力です。この例では、GATES.SI がアドバンスド PLD パッケージの中にあります。このファイルには、回路内のデバイスの関数についての記述があります。テスト仕様ファイルの作成についての詳細は、このマニュアルのシミュレーションテストベクタファイルの作成を参照して下さい。

入力の関数として出力を定義することにより、テストベクタにより予測された PLD の機能が指定されます。プログラムが完了すると、テストベクタはデバイスの機能テストにも使用されます。

図 4-18 のように、シミュレータにより、入力ピンの信号や GATES.SI ファイルに入力される出力ピンのテスト値が、CUPL ソースファイルの論理式から計算された実際の値とが比較されます。これらの計算された値は、コンパイル中に作成されるアブソリュートファイル GATES.ABS に記録されます。

シミュレーションを実行するには以下のようにして下さい。

- Gates.PLD を EDA/クライアントの現在のドキュメントにして下さい。Gates.SI が、カレントディレクトリにあることを確認して下さい。カレントドキュメントを .SI ファイルにしてシミュレーションを実行しないで下さい。
- PldTools ツールバーの Simulate ボタンを押して下さい。

シミュレーションにより作成されるテストベクタが、コンパイル中に作成される JEDEC ダウンロードファイルに自動的に追加されます。

図 4-18 に、シミュレータ入力ファイル(GATES.SI)を示します。

```
Name      Gates;
Partno     000000;
Revision   03;
Date       9/12/83;
Designer   CUPL Engineering;
Company     Protel International.;
Location    None;
Assembly    None;
/*****/
/*
/*      This is a example to demonstrate how the Compiler*/
/*      compiles simple gates.                          */
/*
/*      */
/*****/
/*Target Devices: P16L8, P16LD8, P16P8, EP300, and2S153 */
/*****/

/* Order:  define order, polarity, and output */
/* spacing of stimulus and response values    */

Order:  a, %2, b, %4, inva, %3, invb, %5, and, %8,
        nand, %7, or, %8, nor, %7, xor, %8, xnor;

/* Vectors:define stimulus and response values, with
header*/
/* and intermediate messages for the simulator listing*/
/* Note: Don't Care state (X) on inputs is reflected in  */
/* outputs where appropriate.    */
Vectors:
$msg " ";
$msg "      Simple Gates Simulation";
```

```

$msg " ";
$msg "  inverters  and    nand    or    nor    xor
xnor
$msg "a b !a !b   a & b !(a & b)  a # b !(a # b)  a $ b !(a
$b)";
$msg  - - - - -  -----  -----  -----  -----  -
----- ";

00 HHLHLHLH
01 HLLHHLHL
10 LHLHHLHL
11 LLHLHLLH
1X LXXXHLXX
X1 XLXXHLXX
0X HXLHXXXXX
X0 XHLHXXXXX
XX XXXXXXXXX

```

図 4-18 Gates のシミュレータ入力ファイル(GATES.SI)

図 4-19 にシミュレータ出力ファイル(GATES.SO)を示します。入力に対応する出力と一緒に一覧表示されます。

```

1:Name      Gates;
2:Partno    000000;
3:Revision  03;
4>Date      9/12/83;
5:Designer  CUPL Engineering;
6:Company   Protel International;
7:Location  None;
8:Assembly  None;
9:
10:/*****
11:/*
12:/* This is a example to demonstrate how the Compiler */
13:/*  compiles simple gates.                          */
14:/*
15:/*****
16:/* Target Devices: P16L8, P16LD8, P16P8, EP300, and 82S153*/
17:/*****
18:
19:
20:/*
21: * Order: define order, polarity, and output
22: * spacing of stimulus and response values
23: */
24:
25:Order: a, %2, b, %4, inva, %3, invb, %5, and, %8,
26:      nand, %7, or, %8, nor, %7, xor, %8, xnor;
27:

```

```

28:/*
29: * Vectors: define stimulus and response values, with header
30: *   and intermediate messages for the simulator listing.
31: *
32: * Note: Don't Care state (X) on inputs is reflected in
33: *   outputs where appropriate.
34: */
35:
=====
                Simulation Results
=====

                Simple Gates Simple Simulation
                inverters and nand or nor xor xnor
                a a !a !b a&b !(a&b) a#b !(a#b) a$b !(a$b)
                - - - - -
0001: 0 0 H H L H L H L H
0002: 0 1 H L L H H L H L
0003: 1 0 L H L H H L H L
0004: 1 1 L L H L H L L H
0005: 1 X L X X X H L X X
0006: X 1 X L X X H L X X
0007: 0 X H X L H X X X X
0008: X 0 X H L H X X X X
0009: X X X X X X X X X X

```

図 4-19 Gates のシミュレータ出力ファイル(GATES.SO)

## 例 2 - TTL 設計の PLD への変換

この例では、PLD を使用して既存の TTL 回路を変換する方法を示します。変換には、TTL 論理設計のゲートを PLD でコンパイルできる等価なブール論理式に変換する必要があります。

図 6-3 に、設計する論理システムで使用する TTL ゲート表現とそれぞれのゲートに対応するブール式を示します。

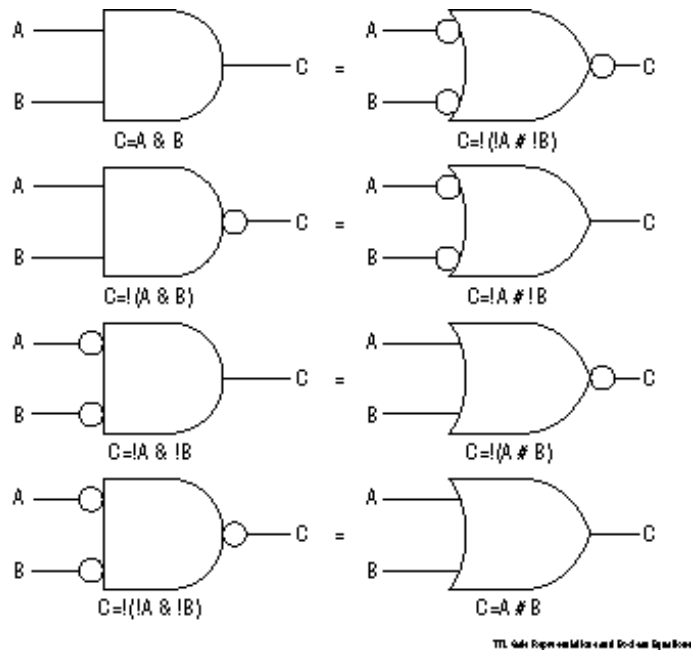


図 6-3 TTL ゲート表現とブール式

図 6-3 で示される基本的な変換ルールを使用すれば、TTL ゲートのシステムの各ゲートの式を記述できます。CUPL は表現の置き換えプロセスを使用して TTL の各ゲートを表現する式からブール式を構築します。式の置き換えは一つのゲート毎に行われます。

図 6-4 に例 1 で変換された TTL ロジックの図を示します。

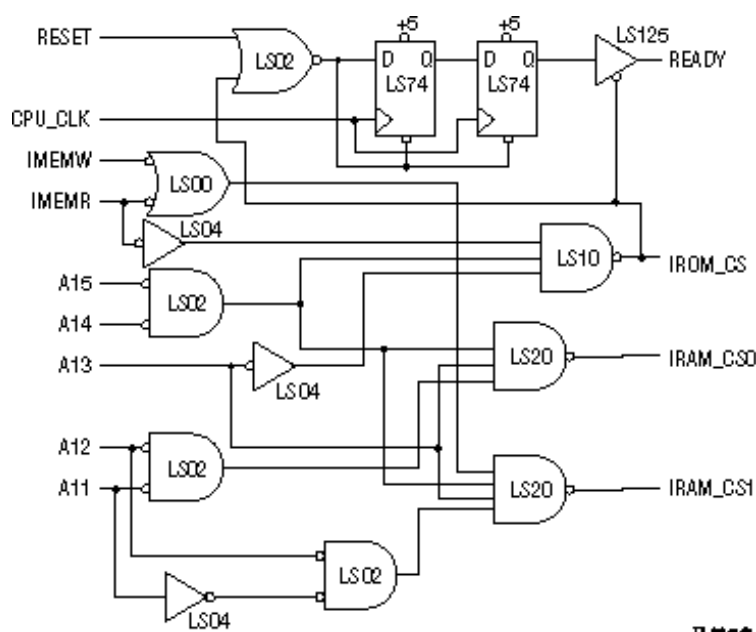


図 6-4 TTL の図

図 6-4 で示される TTL ロジックにより、Sample Design Session で作成された SAMPLED.PLD と同じアドレスデコードや待ち状態を実行することができます。この TTL 回路と等価な PLD により、デバイスを一つ使用し 5~6 パッケージが置き変わります。変換プロセスの最初のステップは TTL の図から PLD に変換するロジックを決めることです。

図 6-5 に、TTL の図の四角で囲まれたロジックと等価な PLD ダイアグラムと PLD のピン割り付けを示します。

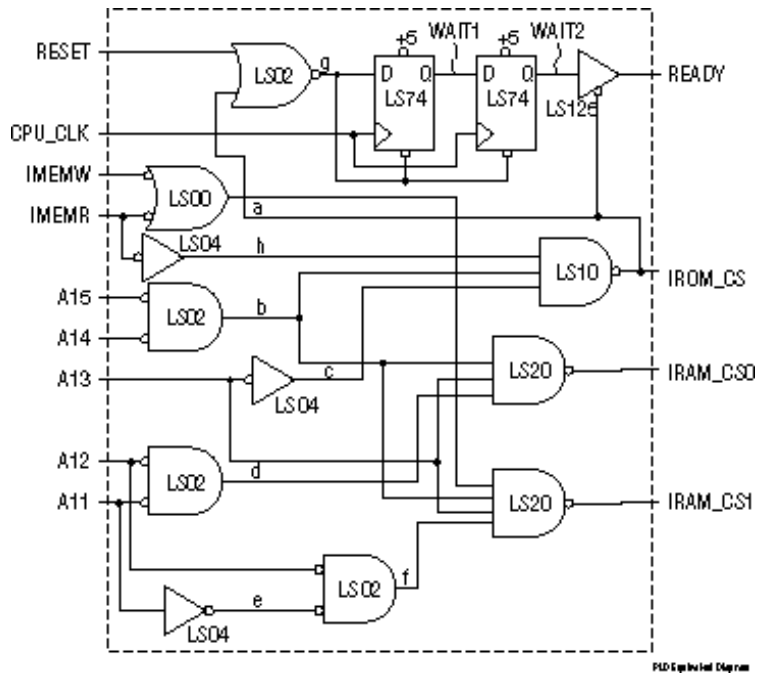


図 6-5 PLD の等価な図

(PLD 出力ピンに接続されていない)内部ゲートの出力に A-H の固有の名前が付けられていることに注意して下さい。これにより論理記述ファイルへの式の入力がやりやすくなります。

TTL 設計を基に待ち状態を生成する回路のため、今回の設計の変換使用される論理記述ファイルは、WGTTT.PLD と名前を付けます。

図 6-6 に WGTTT.PLD の内容示します。

```

Name                Sample;
Partno              P9000183;
Date                07/16/87;
Revision            02;
Designer            Osann;
Company             ATI;
Assembly            PC Memory;
Location            U106;

/*****
/* このデバイスは一つの 8Kx8ROM と 2 つの 2Kx8 スタティック RAM */
/* ヘッチップセレクト信号を発生します。また、その信号はシステム */
/* の READY 線を駆動し ROM アクセス時に少なくとも 1CPU クロック */
/* 分の待ち状態を生成します。                                     */
*****/

/** Inputs **/

PIN 1                = cpu_clk          ; /* CPU clock          */

```

```

PIN [2..6]      = [a15..11]      ; /* CPU Address Bus      */
PIN [7 8]       = ![memw,memr]   ; /* Memory Data Strobes*/
PIN 9           = reset          ; /* System Reset       */
PIN 11          = !oe            ; /* Output enable      */

/** Outputs **/

PIN 19          = !rom_cs        ; /* ROM Chip Select    */
PIN 18          = ready          ; /* CPU ready signal   */
PIN 15          = wait1          ; /* Start wait state   */
PIN 14          = wait2          ; /* End wait state     */
PIN [13,12]     = ![ram_cs1..0] ; /* RAM chip selects   */

/** Declarations and Intermediate Variable Definitions **/

a = !(!memw) # !(!memr) ;
b = !a15 & !a14 ;
c = !a13 ;
d = !a12 & !a11 ;
e = !a11 ;
f = !a12 & !e ;
g = !(!rom_cs # reset) ;
h = !(!memr) ;

/** Logic Equations **/

!rom_cs = !(h & b & c);
!ram_cs0 = !(a & b & a13 & d) ;
!ram_cs1 = !(a & b & a13 & f) ;
wait1.d = g ;
wait2.d = wait1 & g ;
ready.oe = !(!(h & b & c)) ;
ready = wait2 ;

```

図 6-6 WGTTL.PLD

機能は同じのため、ヘッダー情報は、SAMPLE.PLD の物と同じです。

内部ゲートの論理式は、Declarations と Intermediate Variable Definitions のセクションに置かれます。このセクションの式は、図 6-5 で図に割り付けられる A-H の出力変数名を使用します。例えば、AND ゲート LS02 は以下の式で表わされます。

$$d = !a12 \ \& \ !a11 \ ;$$

このセクションの式は、簡単化できます。例えば、以下の式のような二重否定をまとめることができます。

$$a = !(!memw) \ \# \ !(!memr) \ ;$$

となります。

$$a = memw \ \# \ memr \ ;$$



ファイルの Logic Equations のセクションには、PLD の出力信号を記述する式があります。これらの式は、内部ゲートの出力を表わす中間式で記述されます。例えば、AND ゲート LS10 は、!ROM\_CS という出力信号があり、信号 H には、B と C という入力があります。したがって、LS10 は以下の式で表わすことができます。

$$\text{!rom\_cs} = \text{!(h \& b \& c) ;}$$

内部ゲートの定義が違うので WGTTL.PLD と SAMPLE.PLD は厳密には同じではありません。しかし、コンパイルした場合、同じ機能になります。これは、それぞれの論理記述ファイルのシミュレーションを実行することにより確認できます。

TTL 設計を PLD に変換する場合、機能が多少変更される場合があります。LS74 のような TTL フリップフロップにある非同期リセットの機能は通常使用される PLD にはほとんどありません。しかし、同じリセット機能は RESET 変数をすべてのプロダクトタームに取り込みクロックを利用して同期リセットを行なうことができます。

従って、WGTTL.PLD は WAIT1(wait1.d=g;)と WAIT2(wait2.d=wait1.d&g)の式で使用される G の式に !RESET を取り込みます。非同期リセットのタイミングと同期リセットのタイミングに違いがあるものの、同期リセットにしてもデバイスは正しく動作します。

例 2 で示される簡単な方法を使用すると多くの TTL 設計を変換することができます。特に簡単なゲートで構成された TTL 設計を等価な PLD に変換できます。ただし、元の設計の TTL(ロジック)とのタイミングのずれが多少起こる場合があります。ほとんどの場合、TTL 設計と PLD との間の違いは回路内の伝播遅れだけです。

### 例 3 - 2 ビットカウンタ

この例では、Dタイプのフリップフロップを使用する2ビットカウンタのインプリメントを示します。

図 6-7 にカウンタのタイミング図を示します。

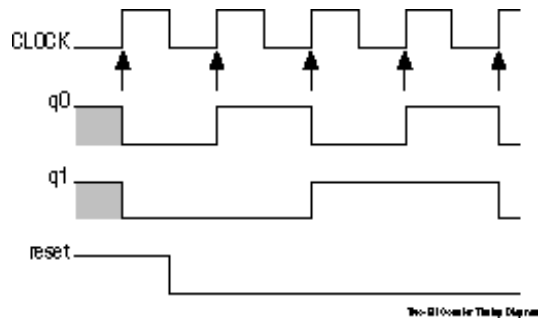


図6-72 ビットカウンタのタイミング図

矢印で示されるように、レジスタはクロック信号の立ち上がりエッジでクロックされます。

図 6-8 に、2 ビットカウンタの(アドバンスト PLD パッケージにある FLOPS.PLD)CUPL ソースファイルを示します。

```
FLOPS.PLD
Name      Flops;
Partno    CA0002;
Revision  02;
Date      07/16/87;
Designer  G. Woolheiser;
Company   ATI;
Location  None;
Assembly  None;

/*****
/
/*                               */
/* この例により、Dタイプのフリップフロップを使用して */
/* インプリメントする方法を示します。 */
/* timing diagram.                */
/*                               */
/* clock |__| |__| |__| |__| |__|          */
/* q0    /// |_____| |_____| |__         */
/* q1    /// |_____||             *//
/*                                     */
/*                                     */
*****/
```

```

/* reset      |_____ */
/*
/*
/*****
/
/*   Target Devices: PAL16R8, PAL16RP8, GAL16V8   */
/*****
/

Pin 1 = clock;
Pin 2 = reset;

/* Outputs: define outputs and output active levels */

Pin 17 = q0;
Pin 16 = q1;

/* Logic: two bit counter using expanded exclusive ors */
/*   with d-type flip-flop           */
q0.d = !reset & (!q0 & !q1
      # !q0 & q1);
q1.d = !reset & (!q0 & q1
      # q0 & !q1);
/* ANDed !reset defines a synchronous register reset */

```

図 6-8 2 ビットカウンタソースファイル(FLOPS.PLD)

ファイルの最初の部分に、ファイルの管理情報や設計されている機能に関する情報、互換性のある PLD の情報が記述されます。

ピン宣言は、設計ダイアグラムの入力と出力に対応して行われます。

ファイルの Logic セクションでは、カウンタの記述が行われます。q0 の式は、q0 がアサートされる時を定義するために記述されます。すなわち、クロックの立ち上がりエッジの前の状態を定義します。

!reset タームは q0 と q1 の式で使用され回路を初期化し、同期リセットを実行します。電源の投入時、タイミング図(図 6-7 参照)の DONT CARE スラッシュで示されるように、レジスタはハイカローの状態です。リセット信号が始めアサートされます。!reset を各変数の式へ AND することでも、電源の投入の状態ではありません。したがって、レジスタはセットされません。電源投入プロセスが完了してリセット信号が LO(偽)に戻ると、!reset は真になり回路内のレジスタに影響を与えなくなります。

式の中の.d 拡張子は D タイプのフリップフロップを示します。しかし、出力がフィードバックに使用される場合、.d 拡張子は削除されます。例えば、q0 は q1 へフィードバックされる場合、式は以下の様に記述することができます。

```
q1.d = q0 & !reset ;
```

以下のようには記述できません。

```
q1.d = q0.d & !reset ;
```

または

```
q1.d = q0.dq & !reset ;
```

## 4 - デカードアップダウンカウンタ

この例では、同期クリア機能を持つ 4 ビットのアップ / ダウンデカードカウンタを示します。また、このカウンタには、多重カスケードデバイスの非同期リップルキャリー出力があります。カウンタを実行するソースファイルは CUPL ステートマシンシンタックスを使用します。

図 6-9 にカウンタ設計とその図を示します。

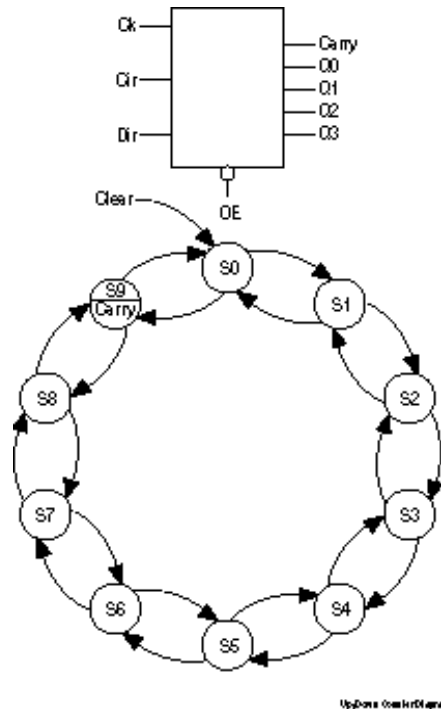


図 6-9 アップ / ダウンカウンタダイアグラム

入力信号 dir により、カウントの方向が決まります。dir がハイの場合、クロック毎に 1 ずつカウントダウンされ、dir がローの場合、クロック毎に 1 ずつカウントアップされます。clr 信号により、同期リセットが行われます。

図 6-10 に設計を実行する(アドバンスド PLD パッケージの COUNT10.PLD) ファイルを示します。

COUNT10.PLD	
Name	Count10;
Partno	CA0018;
Revision	02;
Date	07/16/87;
Designer	Kahl;
Company	ATI;
Location	None;
Assembly	None;

```

Device      pl6rp4;

/*****
/*
/*      デカードカウンタ      */
/* これは、同様のリア機を持つ4ビットのアップ/ダウンカウンタです。 */
/* 非同期的リップルキャリー出が、多量カスケードデバイスに      */
/* 与えられます。CUPLステートマシンシンタックスが使されます。 */
/* is used      */
*****/
/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
*****/
/
/** Inputs */
Pin 1 = clk;      /* counter clock      */
Pin 2 = clr;      /* counter clear input  */
Pin 3 = dir;      /* counter direction input */
Pin 11 = !oe;     /* Register output enable */

/* Outputs      */

Pin [14..17] = [Q3..0]; /* counter outputs      */
Pin 18 = carry;      /* ripple carry out      */

/* Declarations and Intermediate Variable Definitions */
field count = [Q3..0]; /* declare counter bit field */
$define S0 'b'0000
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001
field node = [clr,dir]; /* declare filed node control */
up = mode:0;      /* define count up mode */
down = mode:1;    /* define count down mode */
clear = mode:[2..3]; /* define count clear mode */

/* Logic Equations */
sequence count { /* free running counter */

present S0    if up    next S1;
               if down  next S9;
               if clear next S0;
present S1    if up    next S2;
               if down  next S0;

```

```

        if clear    next S0;
present S2    if up    next S3;
        if down    next S1;
        if clear    next S0;
present S3    if up    next S4;
        if down    next S2;
        if clear    next S0;
present S4    if up    next S5;
        if down    next S3;
        if clear    next S0;
present S5    if up    next S6;
        if down    next S4;
        if clear    next S0;
present S6    if up    next S7;
        if down    next S5;
        if clear    next S0;
present S7    if up    next S8;
        if down    next S6;
        if clear    next S0;
present S8    if up    next S9;
        if down    next S7;
        if clear    next S0;
present S9    if up    next S0;
        if down    next S8;
        if clear    next S0;
out      carry;    /* assert carry output */

```

図 6-10 アップ/ダウンカウンタソースファイル(COUNT10.PLD)

ファイルの最初の部分にはファイルの管理情報や設計の機能情報、互換性のある PLD の情報が記述されます。

設計ダイアグラムの入力と出力に対応するピン宣言が行われます。

Decralations と Intermediate Variable Definitions セクションには、表記を簡単にするための宣言が記述されます。

名前 count は出力変数 Q3 や Q2、Q1、Q0 に割り付けられます。

\$DEFINE 命令を使用して、ステートマシン出力を表わす 10 個の 2 進数ステートに名前が割り付けられます。対応する 2 進数値を表わすためにそのステート名は論理式の中で使用されます。

FIELD キーワードを使用して、clr と dir 入力を mode と呼ばれるセットに結合します。mode は以下の式により定義されます。

```

up = mode:0;
down = mode:1;
clear = mode [2..3];

```

mode は入力 clr と dir を表わします。従って、上記の 3 つの式は以下の式と等価です。

```

up = !clr & !dir ;
down = !clr & dir ;
clear = (clr & !dir) # (clr & dir) ;

```

3つのモードは以下のように定義されます。

```

up      dir と clr 入力 は両方ともアサートされません。
down    dir 入力 はアサートされ clr 入力 はアサートされません。
clear   clr 入力 はアサートされ dir はアサートされるかされないかのどちらかです。

```

Logic Equations セクションには、カウンタの状態を表わすステートマシンシンタクスが記述されます。最初の行で、SEQUENCE キーワードにより、count(すなわち、Q3 や Q2、Q1、Q0)がステート値を適用する出力とされます。

三つのモードのそれぞれの現在の状態から次の状態への遷移を指示するために条件ステートメントが記述されます。例えば、現在の状態が S4 の時、モードが up の場合、counter は S5 になり、モードが down の場合、counter は S3 になります。そして、モードが clear の場合、counter は S0 になります。例 4 で示すように、ステートマシンシンタクスの良い点は、設計の動作を分かりやすくに記述できる点です。

例 4 では、clr 信号の結果のステート 0(2 進数値 0000)が定義されます。ステート 0 で別の値にならないように有効な 0000 をすべての設計に設定することを推奨します。例えば、この設計では、電源投入時に 16 進数の A-F のような定義されていない状態になる場合、式に記述された条件がすべて合わないで、ステートはステート 0(hex 値 0000)になります。0000 が有効なステートとして定義されていない場合、カウンタはステート 0 のままです。

図 6-11 にこの設計を仮想設計として記述する方法を示します。このファイルは同じファイルですが、仮想設計とデバイスで使用する設計との違いを表わすための変更が必要です。

```

COUNT10.PLD
Name      Count10;
Partno     CA0018;
Revision   02;
Date       07/16/87;
Designer   Kahl;
Company    ATI;
Location   None;
Assembly   None;
Device     VIRTUAL;

/*****
/*
/*          デカードカウンタ          */
/* これは、同様のリア機を持つ 4 ビットのアップ / ダウンカウンタです。 */
/* 同様のリップルキャリア出が、多段カスケードデバイスに */

```



```

/* 与られます。CUPLステートマシンシンタックスが使われます。 */
/*****
/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
*****/
/** Inputs **/
Pin = clk;          /* counter clock          */
Pin = clr;          /* counter clear input   */
Pin = dir;          /* counter direction input */
Pin = !oe;          /* Register output enable */

/** Outputs **/

Pin = [Q3..0];      /* counter outputs      */
Pin = carry;        /* ripple carry out      */
/** Declarations and Intermediate Variable Definitions **/
field count = [Q3..0]; /* declare counter bit field */
$define S0 'b'0000
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001
field node = [clr,dir]; /* declare filed node control */
up = mode:0;          /* define count up mode   */
down = mode:1;        /* define count down mode */
clear = mode:2..3]; /* define count clear mode */
/* Logic Equations */
sequence count { /* free running counter */

present S0    if up    next S1;
               if down  next S9;
               if clear next S0;
present S1    if up    next S2;
               if down  next S0;
               if clear next S0;
present S2    if up    next S3;
               if down  next S1;
               if clear next S0;
present S3    if up    next S4;
               if down  next S2;
               if clear next S0;
present S4    if up    next S5;
               if down  next S3;
               if clear next S0;
present S5    if up    next S6;

```

```

        if down    next S4;
        if clear   next S0;
present S6    if up    next S7;
        if down    next S5;
        if clear   next S0;
present S7    if up    next S8;
        if down    next S6;
        if clear   next S0;
present S8    if up    next S9;
        if down    next S7;
        if clear   next S0;
present S9    if up    next S0;
        if down    next S8;
        if clear   next S0;
out    carry;    /* assert carry output */

```

図 6-11 アップ/ダウンカウンタのソースファイル(仮想)

CUPL プリプロセッサの機能を使用するとこの PLD をさらに短くすることもできます。図 6-12 にファイルの大きさを小さくする\$REPEAT 構造を使用して同じファイルを記述する方法を示します。

```

Name      Count10;
Partno    CA0018;
Revision  02;
Date      07/16/87;
Designer  Kahl;
Company   ATI;
Location  None;
Assembly  None;
Device    VIRTUAL;

/*****
/*
/*          デカードカウンタ          */
/* これは、同期リア機を持つ 4 ビットのアップ/ダウンカウンタです。 */
/* 非同期のリップルキャリー出が、多量スケードデバイスに */
/* 与られます。CUPL ステートマシンシンタックスが使われます。 */
/*          */
*****/

/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
/*****

/** Inputs **/

Pin = clk;      /* counter clock      */
Pin = clr;      /* counter clear input */
Pin = dir;      /* counter direction input */
Pin = !oe;      /* Register output enable */

```

```

/** Outputs */

Pin = [Q3..0];      /* counter outputs      */
Pin = carry;        /* ripple carry out      */

/* Declarations and Intermediate Variable Definitions */

field count = [Q3..0]; /* declare counter bit field */
field node = [clr,dir]; /* declare filed node control */
up = mode:0;          /* define count up mode    */
down = mode:1;        /* define count down mode  */
clear = mode:2..3];   /* define count clear mode */

/* state machine description */
sequence count {
present 0
    if up & !clear next 1;
    if down & !clear next 9;
    if clear    next 0;

$REPEAT i=[1..9]
present i
    if up & !clear next {(i+1)%10};
    if down & !clear next {(i-1)%10}
    if clear    next 0;
$REPEND

```

図 6-12 アップ/ダウンカウンタのソースファイル(仮想)

このバリエーションでは、代わりに番号をそのままに使用するので \$DEFINE 命令を削除しました。最も大きな変更は、状態を一つ一つ定義する代わりに \$REPEAT ループを使用してほとんどの状態を定義していることです。このようなことができる理由は、これらの状態が次に進む状態以外は同じだからです。状態 0 を単独で定義し、それからその他の状態を \$REPEAT ループで定義していることに注意して下さい。\$REPEAT ループは、コンパイル時に展開され各状態が定義されます。次の状態で示される命令は繰り返し変数  $i$  から計算されます。ループ内では、 $i$  は現在の状態の番号を示します。従って、次の状態は  $i+1$  です。これは、最後の状態以外はすべての状態で成り立ちます。最後の状態では、状態マシンは状態 0 へ進む必要があります。これを実行するために、次の状態を計算する式は  $(i+1)\%10$  となります。この式は、 $i+1$  を 10 で割ったあまりを表わします。番号 10 は状態の番号を表わします。従って、状態 9 の場合、次の状態は  $(9+1)\%10=0$  となり次の状態は 0 になります。同じような状態は、以前の状態を計算する場合に起こります。状態 0 が独立して定義されていることに注意して下さい。これは、\$REPEAT 変数は正の値しか扱うことができないためです。\$REPEAT ループの中で状態 0 を定義すると、次の状態を  $-1$  とする可能性があり、この場合、コンパイラがは想できな

い結果を生成します。

## 例 5 - 7 セグメントのディスプレイデコーダ

この例では、アノードがコモン LED を駆動する 16 進数から 7 セグメントへのデコーダを示します。この設計には、数値の先頭のゼロを表示しないようにするリップルブランキング入力と桁の表示を簡単にするリップルブランキング出力が組み込まれます。

図 6-13 にセグメントディスプレイデコーダを示します。

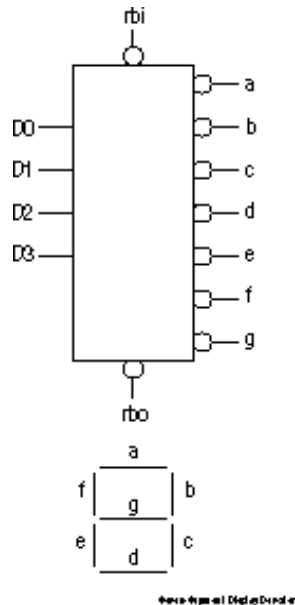


図 6-13 7 セグメントのディスプレイデコーダ

a-g のラベルが付けられた表示のセグメントは図の出力に対応します。

図 6-14 にソースファイル(アドバンスド PLD パッケージの HEXDISP.PLD)を示します。

これは、アノードがコモン LED を駆動する 16 進数から 7 セグメントへのデコーダです。このデコーダには、数値の先頭のゼロを表示しないようにするリップルブランキング入力と桁の表示を簡単にするリップルブランキング出力が組み込まれます。

```

HEXDISP.PLD
Name      Hexdisp;
Partno    CA0007;
Revision  02;
Date      07/16/87;
Designer  T. Kahl;
Company   ATI;
Location  None;
Assembly  None;
/*****

```



```

        & !rbi
/* 1 */ # [OFF, ON, ON, OFF, OFF, OFF, OFF] & data:1
/* 2 */ # [ ON, ON, OFF, ON, ON, OFF, ON] & data:2
/* 3 */ # [ ON, ON, ON, ON, OFF, OFF, ON] & data:3
/* 4 */ # [OFF, ON, ON, OFF, OFF, ON, ON] & data:4
/* 5 */ # [ ON, OFF, ON, ON, OFF, ON, ON] & data:5
/* 6 */ # [ ON, OFF, ON, ON, ON, ON, ON] & data:6
/* 7 */ # [ ON, ON, ON, OFF, OFF, OFF, ON] & data:7
/* 8 */ # [ ON, ON, ON, ON, ON, ON, OFF] & data:8
/* 9 */ # [ ON, ON, ON, ON, OFF, ON, ON] & data:9
/* A */ # [ ON, ON, ON, OFF, ON, ON, ON] & data:A
/* B */ # [OFF, OFF, ON, ON, ON, ON, ON] & data:B
/* C */ # [ ON, OFF, OFF, ON, ON, ON, OFF] & data:C
/* D */ # [OFF, ON, ON, ON, ON, OFF, ON] & data:D
/* E */ # [ ON, OFF, OFF, ON, ON, ON, ON] & data:E
/* F */ # [ ON, OFF, OFF, OFF, ON, ON, ON] & data:F;

rbo = rbi & data:0;

```

#### 6-~~24~~ Sheet 2 of 2

論理式は、関数テーブルとして設定され、各入力パターンに応じたセグメントが記述されます。コメントにより関数テーブルのヘッダーが作成され、出力セグメントは表の一番上に表示され、入力番号は表の横に縦方向に表示されます。

表の各行には、デコードされた hex 値と hex 値でオンオフする表示のセグメントが記述されます。例えば、入力値が 4 の行は以下のように記述されます。

```
[OFF, ON, ON, OFF, OFF, ON, ON] & data:4
```

関数テーブルのフォーマットで記述すると、設計の内容が式で記述するよりも解りやすくなります。すなわち、上記の例では、入力値 4 により、セグメント a がオフ、b がオン、c がオンとなることを表わします。

## 例 6 - ロードリセット機能付きの4 ビットカウンタ

この例では、ロードとリセットのできるカウンタを示します。設計は仮想デバイス VirtualDevice を使用して行いますが、4 つ以上のレジスタのある PLD ならば簡単にインプリメントすることができます。

```
Name          Counter;
Partno         FL1201;
Revision       01;
Date           08/26/91;
Designer       RGT;
Company        LDI;
Location       None;
Assembly       None;
Device         VIRTUAL;
/*****
/* 4-bit counter
*****/

/** inputs **/
PIN = clk; /* clock signal for registers */
PIN = load; /* load signal */
PIN = !ClrFlag;
PIN = [LoadPin0..3]; /* pins from which to load data */

/** outputs **/
PIN = [CountPin0..3];

/* intermediate variables and fields */
field STATE_BITS = [Count0..3];
field LOAD_BUS = [LoadPin0..3];

/** state machine definition **/
Sequenced STATE_BITS {
/* build a repeated loop for the states */
$REPEAT i = [0..15]
    Present 'h'{i}
    /* clear 信号が真の場合ステート 0 へ進み、ロード信号が偽の場合
*/
    /* 次のステートへ進みます。次のステートが(現在のステート+1) */
    /* を 16 で割った余りであることに注意して下さい。これにより、 */
    /* カウンタは最後のステートからステート 0 へ戻ることができます。
*/
    If !load Next 'h' {(i+1)%16};
    If !load & ClrFlag Next 'b'0;
$REPEND
    /* APPEND 命令を使用してロード機能を追加します。
*/
    /* これは、出力の OR ゲートに inputs を追加することを意味します。 */
```



```

/* からロードされることをこの式は表わしています。 */
/* これが状態定義の IF 命令の式で load が使用される理由です。 */
APPEND STATE_BITS.d = load & LOAD_BUS;

```

図 6-15 リセットとロード機能付きのカウンタ

これは仮想設計のため、ピンの番号付けは無視されます。使用される信号はピン番号なしで宣言されます。

load 信号が宣言されると、ロードピンの値がステートビットレジスタへ送られます。Clear 信号が宣言されると、ステートマシンは強制的にステート 0 になります。

\$ REPEAT ループは[0..15]で定義されます。ループの内部は、現在のステートが h[i] で定義されます。これにより番号は hex 番号として評価されます。従って、ステートが 16 個あるステートマシンの、16 進数で 0-F となります。h が取り除かれると、ステートは 10 進数で 0-15 になります。いずれにしても、コンパイラはこれらのステートを hex として解釈し、ステート A-F は定義されません。さらに、ステート 10(HEX)は存在しないため、今回扱うステートビットは 4 ビットです。

IF 命令がすべて !load で AND されていることに注意して下さい。これは、ロード機能の優先順位を一番高くしたいためです。これにより、ロード信号とその他の信号が同時に朝一トされた時に起こる競合を取り除くことができます。

次のステートは、(present\_state+1)を 16 で割った余りで計算されます。これにより、最後のステートからステート 0 へ戻ることができます。ステートマシンのステートの数が 16 なので 16 の余りを使用します。

この設計の最後の部分で設計にロード機能が追加されます。ここでは、APPEND 命令を使用します。APPEND 命令により指定された変数へ OR された式が与えられます。結局このステートマシン全体は、各ステートビットの組みの式になります。out は入力ピンからの値がロードされるレジスタに他の条件を追加するためのものです。IF 命令のすべてに !load が使用されていることを忘れないで下さい。IF 命令により生成された式が APPEND 命令と矛盾しないことを確認して下さい。

```

CountPin0.d = !load & ???.....
              # !load & ???.....;

```

図 6-16 APPEND の前の式

```

CountPin0.d = !load & ???.....
              # !load & ???.....
              # load & LoadPin0;

```

図 6-17 APPEND の後の式

練習のように、この例にアップダウン機能を追加してみてください。また、実際のデバイスにこの設計をインプリメントしてみてください。

# プログラマブルロジックデバイスの歴史

## イントロダクション

1982 年以來、プログラマブルロジックと呼ばれる IC テクノロジーの分野は空前の発展を遂げてきました。プログラマブルロジックという言葉は、理論的には集積回路の分野の大部分を指す言葉ですが、実際には、ほとんどの場合プログラマブルアレイロジック(PAL)や FPLA(フィールドプログラマブルロジックアレイ)、GAL(ジェネリックアレイロジック)として知られるデバイスのより狭い分野を指す用語として使用されています。

よく知られているプログラマブルリードオンリーメモリ(PROM)はプログラマブルロジックデバイス(PLD)の一種に分類されることがよくありますが、このマニュアルでは PROM は扱っていません。このような分野にも興味のある読者は、論理設計に関する専門書を参照して下さい。従って、このマニュアルでは明らかに PROM を示す場合の除いて、PLD という言葉は狭い意味で使用します。すなわち、PAL、FPLA、GAL、アプリケーション指定の PLD(ASPLD)だけを表わします。

70 年代に開発されたこれらの集積回路は、論理設計を効率よくそして早くインプリメントできるために最近になって注目され始めました。PLD の設計に使用できる機能が豊富で簡単に使える CAE ツールが無かったことが、これらの性能がよく効率的なセミカスタムデバイスの普及が遅れた理由であると考えます。最初は、限られた 2、3 の実験室で評判になってはいませんでした。それから時間が経ち、論理設計の分野で、よく知られた off-the-shelf TTL や CMOS ブロックで設計をインプリメントしたいという要求がうまれてきました。

幸いにも、最近のマクロコンピュータ技術の進歩やワークステーション上の CAD/CAE 設計ツールやハードウェア開発システム関連製品の氾濫は、論理設計者の開発方法をまったく変えてしまいました。プログラムロジックデバイスは、まさにブラックボックス的設計の方法論を具体化する回路だったので、CAE ソフトウェアにより拍車のかかったブラックボックス的設計の方法論への回帰は、設計者の意識をプログラマブルロジックデバイスへ向けさせました。PLD は、現在のソフトウェアが効率的に行なっているブラックボックス設計の元になるデバイスであると思われます。ゲートレベル関数の定義、ブレッドボードの作成、トラブルシューティングという設計プロセスは、現在では、2、3 時間に短縮され、数分のプログラムの後ブラックボックスの PLD をカスタムのサブシステムに変更できます。

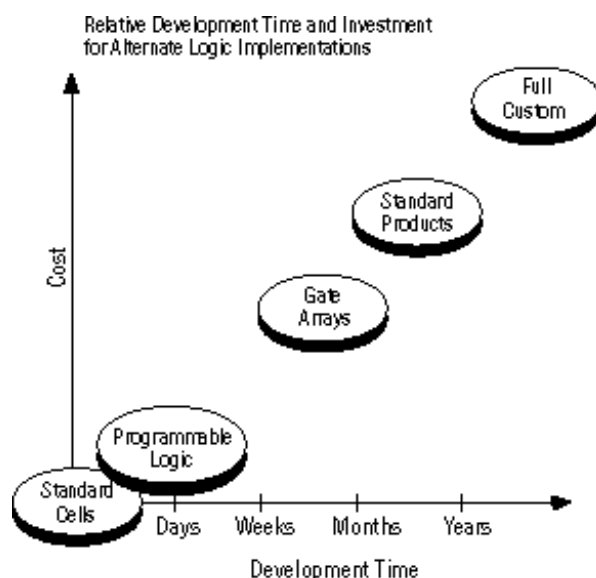


図 7-1 開発時間の比較とロジックのインプリメントの比較

Semi-Custom Solutions			
	Programmable Logic	Gate Arrays	Standard Cells
Description	Off-the-shelf programmed by user	Standard wafers with custom metal mask	Standard logic cells simplify custom layout
Development	Hours	8-12 weeks	12-20 weeks
Tooling Charge	None	\$15K-50K	\$30K-100K
Logic Density	100-1000 gates	100-5000 gates	100-5000 gates

Estimation of ASICs

表 7-1 セミカスタムソリューション

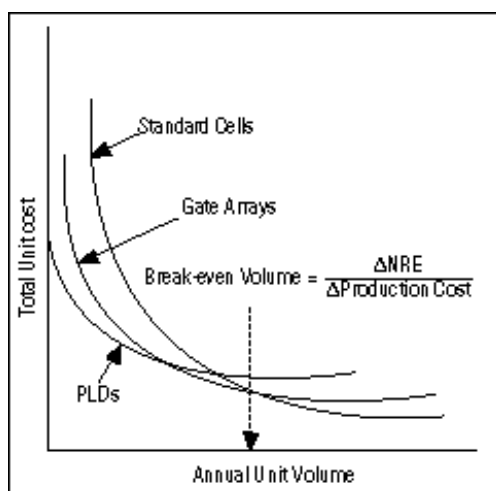


図 7-2 セミカスタム技術を使用して設計された IC の総量についてのトータルのユニットコスト

プログラマブルロジックの進歩は続いており、アーキテクチャの効率化や PLD の高密度化は日進月歩ですすでいます。(図 7-1 参照)同様に、このような進歩はソフトウェアの分野でも進んでおり、これまで一週間かかっていた作業が一時間でできるようになってきている(表 7-1、図 7-2 参照)

このマニュアルは論理設計について何も知らないプログラマブルロジックアレイの潜在的ユーザに PLD の世界を紹介するために書かれています。同時に何等かの工業的な経験を持つ論理設計者の多くは、すでに基本的なブール代数の原理やスイッチング技術、繰り合わせ論理設計やシーケンシャル論理設計についての知識があることを前提しています。

一方、論理設計の初心者や電子工学やコンピュータサイエンスの学生には、このマニュアルの上記のセクションは役に立つものであると思います。特に、内容が PLD 指向であるのでこれから PLD についての知識を増やしたい人には最適です。さらに、このマニュアルで使用される用語や記号は、PLD 開発に関連する PLD のデータシートやアプリケーションノート、ソフトウェアマニュアルで最も広く使用されているものです。

セミカスタムデバイスのアプリケーションを支配する最も基本的な原理の一つは、シリコンの中に与えられたシステム設計をエミュレートすることができることです。この観点から、システム関数の初期定義が入力信号とそれに対応する出力信号の項で記述されるようなよく知られたブラックボックス設計を、直接シリコンのブランクボックスへ変換できます。

## プログラマブルロジックデバイスの分類

プログラマブルロジックデバイスは見た目はただの箱にしか見えませんが、設計者がそれらにエミュレートしてほしい回路の最終的なアーキテクチャをシリコンの中に定義したりインプリメントしたりできます。ユーザは、それらを実現するために必要な大きさの土地のオーナーになり、コードや基本的な外形をそこに構築します。それらで何をつくりたいのかと、どのようにそれを行なうのかを決めてやる必要があります。

PLD を使用して実現したい機能があるにもかかわらず、一般的に PLD は他のデバイスよりも特定のアプリケーションに合う特別なものとして考えられています。今日使用できる PLD の種類は非常にたくさんあります。PLD を分類する方法には、それらを分類する時に考慮するアプリケーションの状況により多くの方法があります。ここでは、分類法の最も基本的な方法であるジェネリックブーリアンセットインプリメンテーションによる分類を示します。

## ジェネリックブーリアンセットインプリメンテーションによる PLD の分類

PLD の最も一般的で理論的な分類法は、スイッチング代数の 3 つの基本的

な演算、すなわち、OR、AND、COMPLEMENT(NOT)演算が行われる方法を分析することで分類する方法です。論理関数の任意のサブセットをエミュレートする能力を唄い文句にしているアーキテクチャにはそれ自身の中にこれらが含まれています。プログラムの中でこれらの要素のうちのどれかを使用していなくても問題はありません。上記の要素のプログラマビリティに基づいて PLD の 3 つのクラスが定義できます。

## PROMs

PROM とはプログラマブルリードオンリーメモリです。PROM はその名前からプログラマブルロジックとは違う印象がありますが、実際には、アドレスで指定された固定データを保存する要素としてだけでなく複雑な組み合わせアーキテクチャをエミュレートできる強力なプログラマブルロジックデバイスです。多くの PROM のアーキテクチャは一般的にプログラム可能な OR アレイの元になる固定された AND アレイで構成されます。それらは主に特定の入力の組み合わせをマイクロプロセッサ環境のメモリマッピングなどの出力関数にデコードするために使用されます。

設計者の多くは、PROM をアドレスへデータを置いたり、データを取得する関数と結び付けますが、そのような演算を別の観点からみる方法があります。マルチビットデータの PROM の場合、そのデータ出力はそれぞれ(通常いわゆる ADDRESS 線に連結される)任意の入力に対して TRUE を生成することができます。(PROM で使用できるアドレス入力の総数に応じて)入力の組み合わせを選択できるオプションがあるので、入力の 2 進数信号の  $A(0) \dots A(N)$  の任意の組み合わせ関数を出力に割り付けることができます。ハードウェアの観点からアドレス入力状態のすべての 2 進数の組み合わせはデバイスアーキテクチャ(図 7-3)であらかじめ配線され、設計者は単に出力を TRUE にする選択された入力の組み合わせを割り付けるだけです。これらの状態を出力に割り付ける実際の物理的作業がプログラミングで、この作業はデバイス内部のスイッチを ON または OFF にする過程に応じていろいろな形式が考えられます。デバイスアーキテクチャにより実行される論理関数の割り付けについての詳細は後で説明します。

アプリケーションのある部分では非常に多くの機能があり効率的ではありませんが、PROM には、論理関数のブランクボックスとして機能に関しては制限があります。入力が非常にたくさんあり、それらにより少ない出力が生成されるような場合、非常に効率がわるくなります。このようなアプリケーションの場合、N 個の入力の可能な組み合わせの数は、爆発的に多くなります。しかし、ほとんどの論理アプリケーションでは、これらの組み合わせのほんの少数が実際にアクティブ TRUE 出力の生成に使用されます。チップ上のスイッチアレイの急速な増加にも関わらず、そのうちの一部しか TRUE 出力の生成に使用されません。

PROM のプログラマブルロジック要素としてのその他の問題点は入力の拡張性の問題です。デバイスを 7 入力のかわりに 8 入力で(アドレス信号線)使用する状況になった場合、臨時の入力信号はたった一度しか使用しなくても、デバイスのヒューズの数を 2 倍にする必要があります。

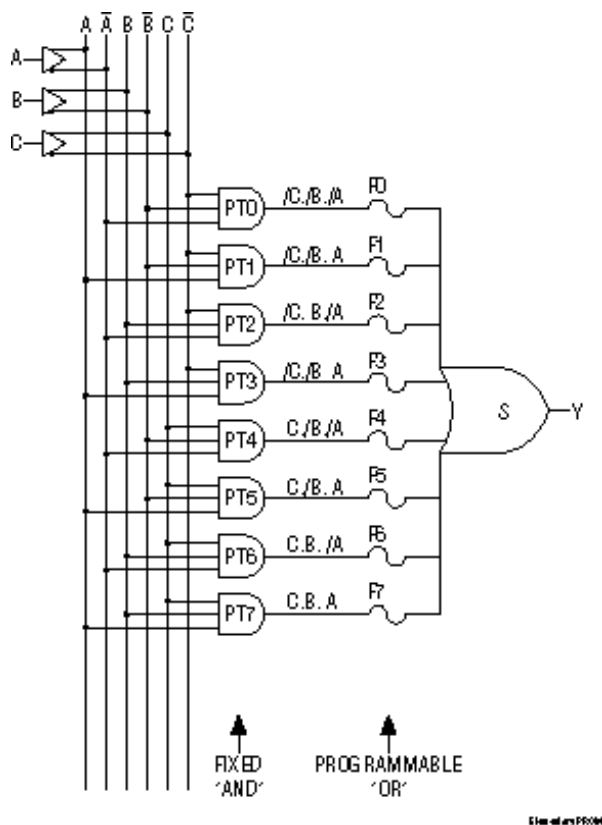


図 7-3 PROM アーキテクチャの基本

## PALs

PAL と GAL のイントロダクションでは、デバイスの入力の増加にともなうヒューズアレイの増加の問題について触れませんでした。PAL や GAL では、アドレス線やデータ線の役目は反対です。すなわち、PAL/GAL の出力がアクティブ TRUE 信号になるアドレスをユーザが選択できます。このようなことは他のアドレスでも同様で、入力に適用された場合、FALSE かハイインピーダンスが生成されます。このようなプロセスはすべての出力に対して行われます。最近の多くのデバイスでは、入力ピンや出力ピン、両方向 I/O ピンを外部から設定できる機能を持っています。このように PLD のアーキテクチャはその機能を拡張しています。

プログラマブルアレイを単純な形で表現できることは、入力変数により生成されるアドレスの組み合わせの数が、N 本の入力線で生成できる組み合わせの数よりも非常に少なく制限されているために価値があります。しかし、このような制限はよくあることで、TRUE 出力を生成するために必要な入力変数の組み合わせの数はロジックアプリケーションで通常制限されます。内部のアーキテクチャは、固定された OR タームへ行く AND タームで構成されます。PAL には大変良く知られたアーキテクチャがあり、おそらくユーザプログラマブルデバイスの中で最も広く使用されているものです。デバイスにマイクロセルがある場合、それは普通 PAL アーキテクチャを使用しています。典型的なマイクロセルは入力や出力、入出力(I/O)をブ

プログラムできます。それらのマイクロセルには I/O ピンと接続できる出力レジスタがあります。その他のマイクロセルには複数のレジスタを持つものや、アレイへの各種のフィードバックタイプを持つもの、マイクロセル同志でフィードバックできるものがあります。これらのデバイスは主に、マルチ TTL ロジック関数を置き換えるために使用され、一般にグルーロジックと呼ばれます。

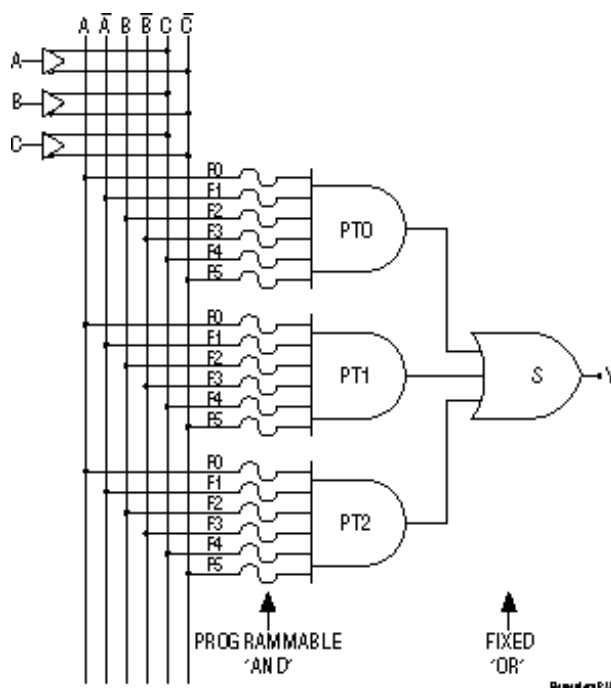


図 7-4 PAL アーキテクチャの基本

## GALs

GAL とは、ジェネリックアレイロジックデバイスのことです。GAL はマイクロセルを使用して PAL デバイスをエミュレートするためのデバイスです。数種類の PAL デバイスを使用してインプリメントされた設計がある場合、同じ GAL デバイスを使用してエミュレートできます。これにより、在庫しておくデバイスの種類を減らすことができ、したがって購入するデバイスの数を増やすことができます。同じデバイスを大量に購入できれば、それだけデバイスの単価を下げるすることができます。また、これらのデバイスは電氣的に消去できます。このために、設計者にとって都合の良いデバイスになっています。

## PLAs

PLA とは、プログラマブルロジックアレイのことです。これらのデバイスには、プログラムできる AND と OR のタームがありどの AND タームでも OR タームと接続することができます。論理の機能面で他のデバイスよりもフレキシビリティが高いといえます。それらには、通常非同期ステートマシンをインプリメントするために使用する OR アレイから AND アレイへの

フィードバックがあります。しかし、ほとんどのステートマシンは同期マシンとしてインプリメントされます。このようなことから、シーケンサと呼ばれる PLA のタイプが作成されました。シーケンサは OR アレイの出力から AND アレイへレジスタを介するフィードバックループを持つデバイスです。

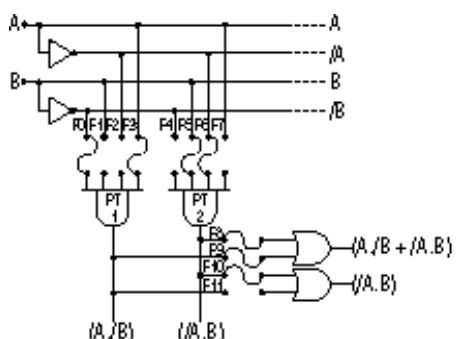


図 7-5 PLA アーキテクチャの基本

## コンプレックスPLD

コンプレックス PLD という名前から、複雑なプログラマブルロジックデバイスという意味を連想できます。実際にそれらは、PLA の特性の一部を持つ大規模な PLA とみなされています。基本的なアーキテクチャは PAL とよく似ており OR タームへ行く AND タームの総数を増やすことができる機能を持っています。このように AND タームを増やすことは、近接する AND タームや拡張アレイの AND タームを使用することで行われます。このようなことは、これらのデバイス内でインプリメントされるほとんどの設計で行なうことができます。

## FPGAs

FPGA とは、フィールドプログラマブルゲートアレイのことです。これらのデバイスは多重論理レベルを持つプログラマブルゲートアレイのことです。FPGA の特徴は、ゲート密度が高く、高性能でユーザが定義できる入出力を多数持っており、多様な内部接続方法があり、さらにゲートアレイライクな設計環境であることです。それらは、AND-OR ゲートアレイではありません。FPGA には配置可能なロジックブロック(CLB)の内部マトリックスと I/O インタフェースブロックの周囲を囲む輪があります。内部接続のリソースは CLB と I/O ブロック(IOB)の間にあります。各 CLB には、プログラム可能な組み合わせ論理とストレージレジスタがあります。ブロックの組み合わせ論理のセクションは、入力変数の任意のブーリアン関数をインプリメントできます。各 IOB は、入力ピンと出力ピン、両方向ピンを個別にプログラムできます。また、IOB には、バッファ入力やバッファ出力に使用されるフリップフロップがあります。内部接続のリソースは、CLB の間を縦横に走る配線のネットワークです。プログラム可能なスイッチにより IOB の入出力ピンと CLB が近くの配線に接続されます。デバイスの幅いっぱいには渡り走っている配線の信号は、内部の変換が行われないので、遅れ



や歪みの少ない信号を分配できます。FPGA を使用する設計者は回路の論理関数を定義したり必要に応じてこれらの関数を修正したりできます。このように FPGA を使用すると、カスタムゲートアレイで数週間かかるような設計検証の作業が数日でできるようになります。

## 論理アーキテクチャによるPLD の分類

プログラマブルロジックデバイスでエミュレートされる論理関数に関係なく、上記で示されるような 3 つの基本的なアーキテクチャ特性のタイプがあります。これらの特性は、論理設計の特定のアプリケーションの分野に合うものです。この 3 つのアーキテクチャの外観を理解することは大切なことです。一般に、小規模の論理システムには 2 つのクラスがあります。すなわち、シーケンシャル回路と組み合わせ回路です。純粋に組み合わせ回路だけの大規模ディジタルシステムは想像することが難しいかもしれませんが、小さなサブシステムではこのような回路はよくあります。バスアービタやチップセレクトデコーダ、カスタムマルチプレクサは典型例です。実際、組み合わせシステムは、回路出力の値に関係なく、回路入力の状態だけで論理関数を実行するような内部接続された論理ゲートの組み合わせです。言い換えると、組み合わせ論理の部分には、処理を薦めるために必要な回路ノードの中間状態を保存するために必要なストレージ要素(ラッチやエッジトリガのフリップフロップなど)がありません。ほとんどの場合、組み合わせ回路は、動作のために外部タイミング信号(CLOCK)を必要としません。というのは、このような回路は、AND ゲートや OR ゲート、INVERTER ゲートなどの基本的なディジタルブロックの組み合わせで構成されており、入力信号はそれらのゲートをリアルタイムで流れていくためです。(図 7-6)

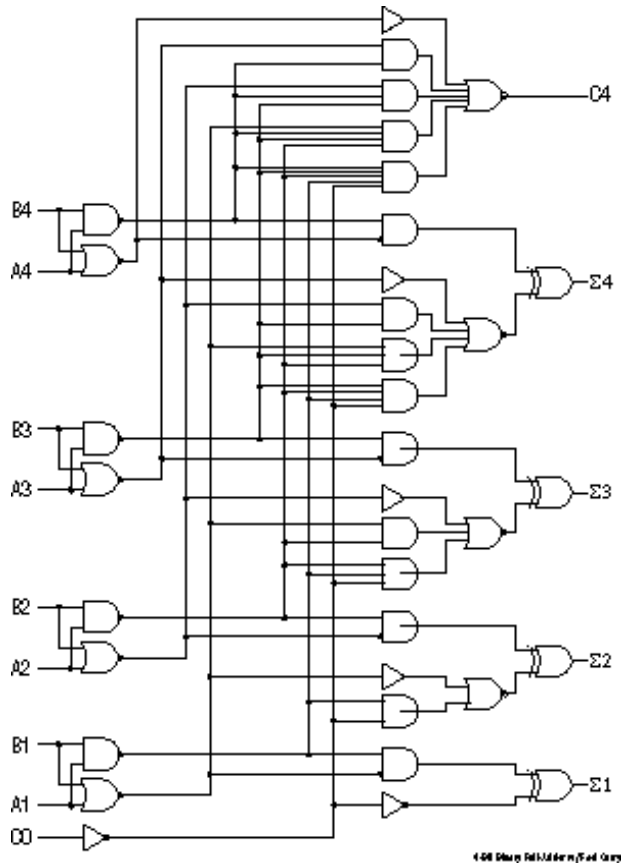


図 7-6 4 ビットの 2 進数のフルアッダー

一般の組み合わせ回路を実行するために開発された PLD のクラスは組み合わせ PLD と呼ばれます。

ネットワークの以前の状態を保存したり回収するための回路で双安定要素のインプリメントが必要なより複雑な回路はシーケンシャル回路と呼ばれます。それらの設計に応じて、2 進数信号の流れは外部のタイミング信号 (CLOCK) やシーケンシャルネットワークの信号の特定のトランジションにより制御されます。前者のような場合、非同期シーケンシャル回路と呼ばれ、後者の場合、同期シーケンシャル回路と呼ばれます。図 7-7 で示されたようなシーケンシャル論理設計のエミュレーションに適した PLD のクラスはレジスタード PLD と呼ばれます。このような名前が付いた理由は、ストレージ要素が PLD にインプリメントされているためです。レジスタと呼ばれるこれらの要素は実際に PLD の出力でインプリメントされ、以前の状態をスイッチアレイへフィードバックするために使用されます。

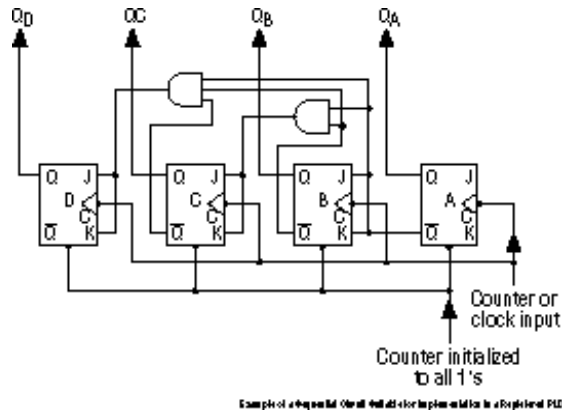


図 7-7 レジスタード PLD にインプリメントするのに適したシーケンシャル回路

アプリケーションスペシフィック PLD と呼ばれる PLD のもう一つのクラスについてここで触れておきます。初期の PLD 以来、多くの PAL や GAL、FPLA、PROM が開発されてきました。より複雑な論理関数をエミュレートできるように、それらのアーキテクチャはより高密度で多くのピンを扱うように改良されてきました。しかし、最近のアプリケーションスペシフィック PLD の開発の流れは、一般的な使用目的からマイクロプロセッサの発展に似た開発に変わってきています。ヒューズマトリックスによるユーザが設定できるオプションはあるものの、これらデバイスのアーキテクチャは論理設計のある特定の分野のために作られています。このようにすると、特定の関数のエミュレーションが一般的な PLD を使用するよりも非常に効率的に行なうことができます。特に、アプリケーションスペシフィック PLD はアドレスデコードやメモリマッピング、マイクロシーケンシング、コード変換のために開発されてきています。ASPLD には、通常アプリケーションの特定の分野で常に使用される論理関数が装備されており、カスタムメイドのアプリケーションスペシフィックの関数は設計者の裁量により使用しないことも可能です。これらのデバイスのスイッチアレイの大きさは通常のデバイスと比較して少ないことがあります。これにより、使用しない部分の大きさを少なくしたり、コストを下げることができます。通常使用される高密度 IC の PROM に同様の考え方を適用すると、レジスタードデバイスやダイアグノスティックデバイス、シリアルデバイス、マルチポートデバイスなどのよりアプリケーションよりの PROM が浮かびあがってきます。

## 技術による PLD の分類

通常、使用できる PLD で最速の PLD は、Schottky や Advanced Schottky などの標準のバイポーラプロセスの拡張版を使用して製造されています。これらの PLD により、速度や価格の面で設計者の選択の幅が広がっています。それらのデバイスには、ヒューズスイッチアレイが装備され、20 から 180mA の動作電流が必要です。

最近増加しつつある 3 番目の PLD 製造技術は CMOS です。バイポーラ技術と違い、CMOS では、熔断方式のスイッチアレイとフローティング方式の

スイッチアレイのどちらでもインプリメントでき、どちらのデバイスも広く出まわっています。さらに、純粋な CMOS 設計では、電力消費の少ないデバイスを製造できます。すなわち、スタンバイモードでは無視できるくらいに消費電力が少なくなります。CMOS デバイスでは、PLD の消費電力は入力と出力が切り替わる速度に比例します。言い換えると、デバイスがアクティブでない間は消費電力はほとんど 0 になります。通常、これらのデバイスはスタンバイモードでもかなりの電力を消費します。しかし、速度の要求がそれ程厳しくない場合、これらのデバイスを PAL や GAL の代わりに使用すると、かなりの消費電力の節約になる場合が少なくありません。さらに、それらの再プログラム可能な機能は大変便利です。図 7-8 に、これまでに説明された PLD の 4 つの技術面からの分類の速度と消費電力の特性を図示します。

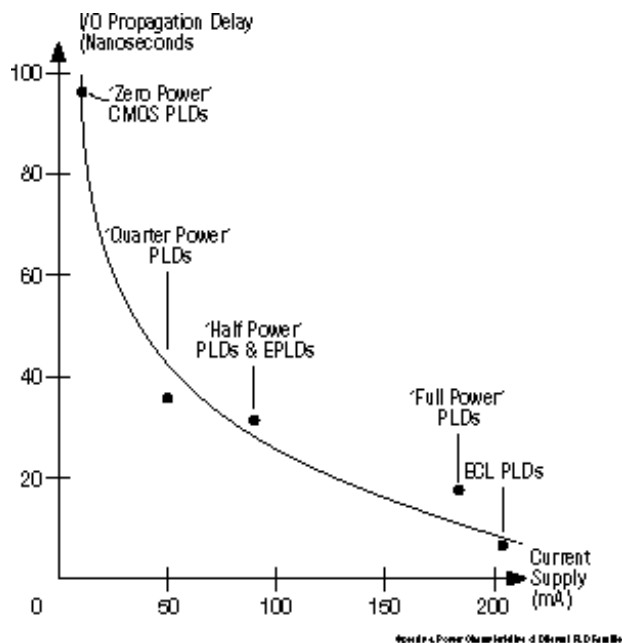


図 7-8 PLD の違いによる速度対消費電力の特性

## PLD のパッケージング

デバイスのパッケージングは、イレーサビリティと物理的コンフィギュレーションの 2 つのカテゴリに分けられます。デバイスの中には、ロジックを消去して再プログラムできるものがあります。これらのデバイスは紫外線をあてるか高電圧によりクロスコネクションリンクのヒューズを元に戻します。紫外線で消去するデバイスはデバイスの中央に窓があり内部に紫外線が当てられるようになっています。電気的に消去するデバイスには、デバイスを消去するための高電圧を引加するためのピンがあります。消去できないその他のデバイスは、ワンタイムプログラマブル(OTP)と呼ばれます。名前からわかるように、これらのデバイスは一回しかプログラムできません。

プログラマブルデバイスはいろいろな形や大きさになってきています。ほ

とんどのデバイスは、以下のような形態になってきています。すなわち、DIP(デュアルインラインパッケージ)、SKINNYDIP、LCC(リーディドチップキャリア)、PLCC(プラスチックリーディドチップキャリア)、PGA(ピングリッドアレイ)です。これらのデバイスの形状は、両側にピンが配置された長方形や、すべての辺にピンが配置された正方形、下面にピンが配置された正方形です。ハードウェアやソフトウェア開発ツールにとって重要なことは市場に出まわっている無数のデバイスの中から利用できる機能をすべて使用できるようなデバイスをすべてサポートできることです。

## 論理デバイスのプログラミング

プログラマブルロジックデバイスは、デバイスアレイ内部の接続をつないだり切断したりしてプログラムします。ほとんどのハードウェアプログラマは、ASCII フォーマットのソフトウェア開発パッケージからヒューズ情報を受け取ります。アドバンスド PLD では、これはコンパイラが行います。ASCII ファイルは PLD 用の JEDEC フォーマットと PROM 用の HEX フォーマットがあります。このファイルには、デバイスのプログラムに必要な情報が記述されています。JEDEC ファイルには、断線状態が 1 で接続状態が 0 で表わされるヒューズの結線情報が記述されています。また、JEDEC ファイルには、ハードウェア設計者がデバイスの機能テストを行なうために必要な情報も記述されています。

## 論理デバイスの機能テスト

ハードウェアやソフトウェア開発環境がテストベクタの生成や使用をサポートできるデバイスの場合、プログラム後に機能テストを実行することができます。テストベクタは設計のピンリストや機能テストの各段階での入力値、回路の予測される出力値で構成されています。プログラマは入力値を順番にならべ、出力値を検索し、結果をユーザに知らせます。これにより、設計者や製造のスタッフは設計されたようにプログラムされたデバイスの動作を確認することができます。アドバンスド PLD では、このようなことはシミュレータで行います。

## PLD 設計理論

このセクションでは、PLD 設計に関する一般的な理論を説明します。議論される話題は PLD 設計に関するものです。設計者は比較的楽に論理設計の理論概要を理解できるでしょう。

- ブール代数
- ステートマシン設計
- カウンタの設計
- 交通の制御
- アクティブハイとアクティブロー設計

### ブール代数

PLD 設計の最も特徴的な概要の一つは、ブール代数の原理の理解が必要なことです。論理機械である PLD は、論理の変換や表現、最小化に精通した設計者の非常に強力な設計ツールです。

ゲートの選択やそれらの入力の種類が限られるシリコン TTL によるブロックの構築と異なり、PLD では与えられた設計を多くの方法で実行できます。通常、それらの内の一つがゲートの利用において最も効率的な設計になります。

#### ブール代数の仮定と概念

スイッチング代数の理論はイギリスの数学者ジョージブールにより確立されました。彼は、2 値の数値システムに適用する計算法の仮定と理論に関する研究成果を 1847 年に発表しました。しかし、1938 年にクロードシャノンがブールの理論に再び注目し、2 つの状態の内の一つだけの状態しか持たない要素で構築される物理システムの解析とシンセシスの価値を見出すまで、その理論はあまり注目されていなかった。このようなシステムや 2 進数算術について詳しく触れることはこのマニュアルの範囲を越えています。これら初期の業績として、2 進数のシステムは結局 2 値の物理減少、ON/OFF や明るい / 暗いなど、にマップされるということを理解できれば充分です。10 進数値のシステムが、私達が生きる上で 10 本の指にマップされるのと同じことです。

G.ブールによって作られた代数規則は一組の仮定(例えば、規則の正当性に影響しない任意の仮定)に基づいています。システムに適用する場合、それは 2 つの使用できる値の一つだけを要素が持つ事ができることを意味します。この 2 進数システムでは、要素が持つ事ができる 2 つ値は 1 か 0 です。ここで 1 と 0 をその 2 つの値にしたのは広く使用されているからです。従って、他の記号等を使用してもかまいません。また、それらの 10 進数の意味とは全く関係ありません。同様な理由で 1 を TRUE として扱い、0 を FALSE として扱います。これらの状態、すなわち 1 / 0 と TRUE / FALSE

は、このセクションを通じて同じ用語として使用されます。また通常、論理値 1 は論理回路の高電圧レベルに結びつけられ、論理値 0 は低電圧レベル(通常グランド)に結び付けられるので、これらの状態をアクティブ HIGH とアクティブ LOW として表現します。TRUE や FALSE 状態を 1 や 0 と関連付けるのはとてもふさわしいやり方です。従って、アクティブ HIGH やアクティブ LOW を TRUE または 1 と定義するのもとてもふさわしいやり方です。ブール代数のこのような二重性はそれが持つ力の源でもあり、論理設計の初心者のフラストレーションの元でもあります。同じようなことがシンボル中にも言えるものがあります。すなわち、10 進数演算にそれらのシンボルを使用する場合とブール代数で使用する場合、全く違った演算になることがあります。

G.ブールにより確立された仮説を以下に示します。

*1 に等しい 2 進数システムの要素の状態 A は、この状態が 0 と等しくないことを意味します。反対に 0 に等しい 2 進数システムの要素の状態 A は、この状態が 1 と等しくないことを意味します。*

さらに、2 進数システムの要素の状態を 2 進数変数 A として表わします。すると、上記の仮定は以下のように簡潔に表わすことができます。

A) If  $A = 0$  then  $!A = 1$  and If  $A = 1$  then  $!A = 0$

ここで!は否定演算子です。

この仮定は以下のようにも表現できます。

If A is FALSE then A is not TRUE and if A is TRUE then A is not FALSE

上記の 2 つの仮定により、スイッチング代数の 2 進数特性が定義されます。関連して、次の 6 個の仮定が定義されます。

B)  $0 \& 0 = 0 \& 1 = 1 \& 0 = 0$

C)  $1 \& 1 = 1$

D)  $0 \# 0 = 0$

E)  $1 \# 1 = 1 \# 0 = 0 \# 1 = 1$

F)  $!0 = 1$

G)  $!1 = 0$

これら 7 つの仮定(A-G)により、論理変数で実行される代数演算の規則が定義されます。特に、仮定 B と C は 2 つの論理変数の論理 AND で呼び出される演算を扱っています。これは、このマニュアルの中で & で示されています。仮定 D と E は、論理 OR 演算の結果を定義し、# で示されます。仮定 F と G は、論理変数の COMPLEMENT または INVERSION を定義します。通常、NOT として参照されます。COMPLEMENT 演算子 ! は変数の前に置かれます。このマニュアルでは、慣例的に使用されるバー( )は使用されません。

論理設計を行なう場合、上記の演算と物理的に等価な 3 つの基本てきな論理要素が上記の仮定により定義されます。すなわち、AND ゲートと OR ゲ

ート、通常 INVERTER として参照される論理 COMPLEMENT 要素です。図 8-1 に、これら 3 つの要素のアクティブ HIGH とアクティブ LOW の TRUE 状態の表現を示します。これらの要素の入力と出力の部分の丸により、入出力変数は TRUE 状態から FALSE 状態へ反転されます。図 8-2 に、NAND や NOR などの要素とそれらが論理設計で使用される場合の代わりの表現を示します。これらの要素は、A-D の仮定から導かれます。また、図 8-2 には、2 次的な論理要素の例もあります。これらは基本要素から導かれるものですが、よく使われるので独立のゲートとして定義されています。これらの演算子は、EXCLUSIVE-OR(XOR)と EXCLUSIVE-NOR(XNOR)です。

CUPL には XNOR の演算子はありません。XNOR 関数は XOR 関数を否定して導くことができます。

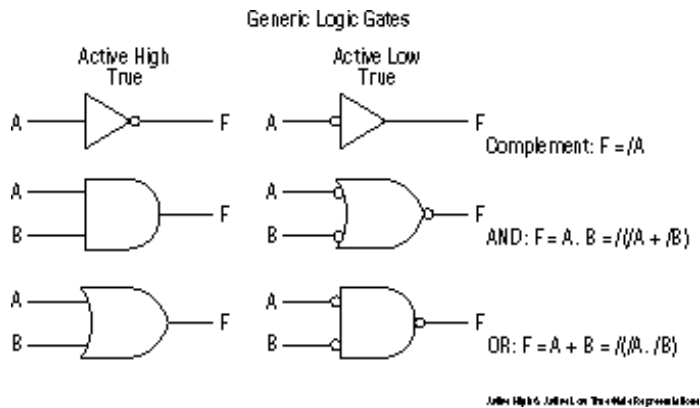


図 8-1 アクティブハイ&アクティブローの TRUE 状態表現



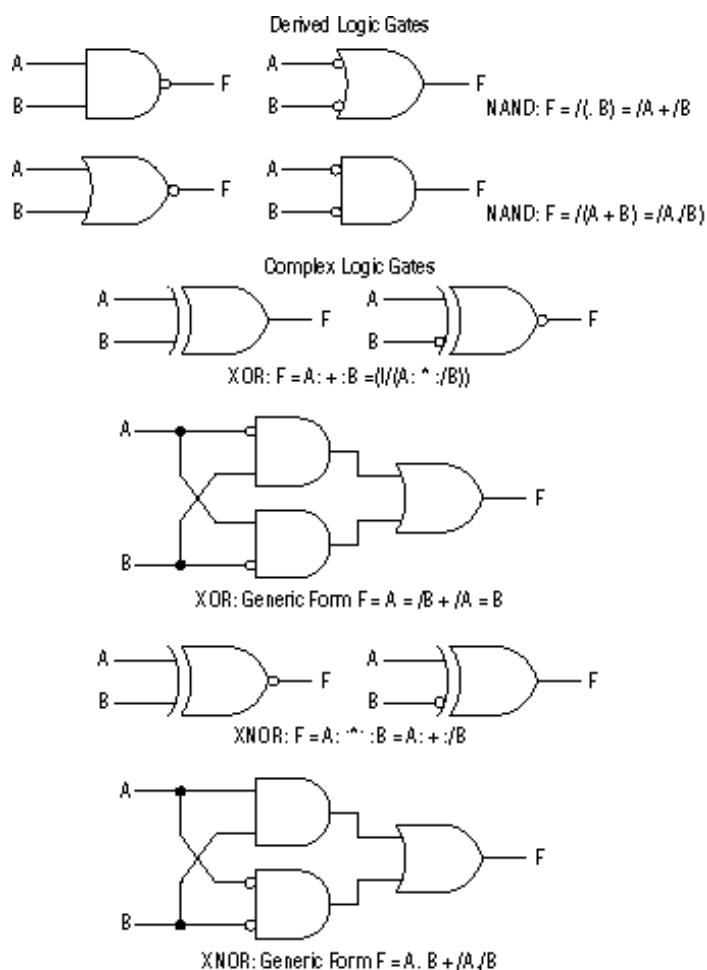


図8-2 基本要素から導出される要素と2次論理要素

ここまで、2進数システムの振る舞いを支配する仮定について説明してきました。つぎは、これらの仮定(図8-3)から導出される理論が法則について説明します。ボールド体の大文字で示される名前は、論理ゲートの入力や出力などの物理的な2進数要素の状態に応じた2進数変数を示します。

$A \# 0 = A$	
$A \# 1 = 1$	
$A \& 0 = 0$	
$A \& 1 = A$	
$A \# A = A$	
$A \& A = A$	
$A \# !A = 1$	
$A \& !A = 0$	
$A \# B = B \# A$	(第1の交換法則)
$A \& B = B \& A$	(第2の交換法則)
$A \# (B \# C) = (A \# B) \# C = (A \# C) \# B$	(第1の結合法則)

$A \& (B \& C) = (A \& B) \& C = (A \& C) \& B$	(第2の結合法則)
$A \# (B \& C) = (A \# B) \& (A \# C)$	(第1の分配法則)
$A \& (B \# C) = (A \& B) \# (A \& C)$	(第2の分配法則)
$!(A \# B) = !A \& !B$	(第1のドモルガンの定理)
$!(A \& B) = !A \# !B$	(第2のドモルガンの定理)

図8-3前提の理論と法則

上記の法則と理論は、ブール代数の基本的な規則を網羅しています。従って、スイッチング理論のその他の特定の理論はこれらの規則から導出できます。

他の代数と同様に、ブール代数の概念は論理変数の機能を利用して論理システムの振る舞いを表現することです。ブール式や真理値関数、変換関数として表わされるこれらの機能は、TRUE と FALSE の論理値の組み合わせで定義されます。それらは、基本演算子や 2 次演算子により論理変数の組み合わせで表現されます。

以下のように定義された 3 つの論理変数 A,B,C の関数 F を考えてみます。

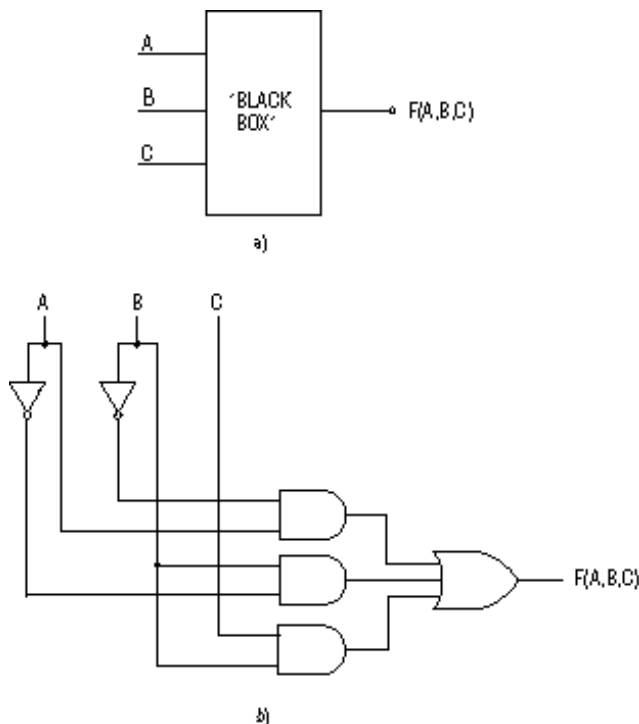
$$(A, B, C) = A \& !B \# !A \& B \# B \& C$$

表 8-1 に、変数 A,B,C の可能な 8 つの組み合わせの関数 F(A,B,C)の値を示します。

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 8-F(A,B,C) Truth Table 3-1 - Functions

表形式の論理関数の定義は通常真理値表定義と呼ばれます。多くの設計例で、特定の設計の真理値表の公式はその変換関数の定義の最初の段階になっています。設計の真理値表定義は入力変数のすべての組み合わせを示すので、そこから作成される変換関数は正準形と呼ばれます。しかし、正準形の変換関数は論理的に縮小できるタームや削除できる変数が含まれているため、ハードウェアインプリメンテーションの観点から非常に効率の悪いものになります。



LOGIC SOLUTION FOR THE FUNCTION  $F(A,B,C)$

図 8-4 関数  $F(A,B,C)$  の論理ブラックボックス

表 8-1 から関数  $F(A,B,C)$  を作成するために、関数(ブラックボックスの出力)が TRUE 状態(1)を出力する変数の組み合わせを選択して下さい。論理値 1 はアクティブ HIGH、すなわち物理的な論理要素の TRUE 状態に対応します。表 8-1 で示されるように変数 A,B,C には、TRUE または FALSE 状態の特定の組み合わせがあります。このようなタームの組み合わせを言葉で表わすと

もし、A が FALSE で B が TRUE で C が FALSE ならば、 $F(A,B,C)$  は TRUE です。

この表現を TRUE だけで表わすと

もし、A が NOT TRUE で B が TRUE で C が NOT TRUE ならば、 $F(A,B,C)$  は TRUE です。

2 番目の表現はブール表現で簡単に表わすことができます。このように複数の変数を一つの AND で結合するような表現は論理プロダクトタームと呼ばれます。 $F(A,B,C)$  が TRUE のプロダクトタームを以下に示します。

PT0 =  $\neg A \& B \& \neg C$   
 PT1 =  $\neg A \& B \& C$   
 PT2 =  $A \& \neg B \& C$   
 PT3 =  $A \& \neg B \& \neg C$   
 PT4 =  $A \& B \& C$

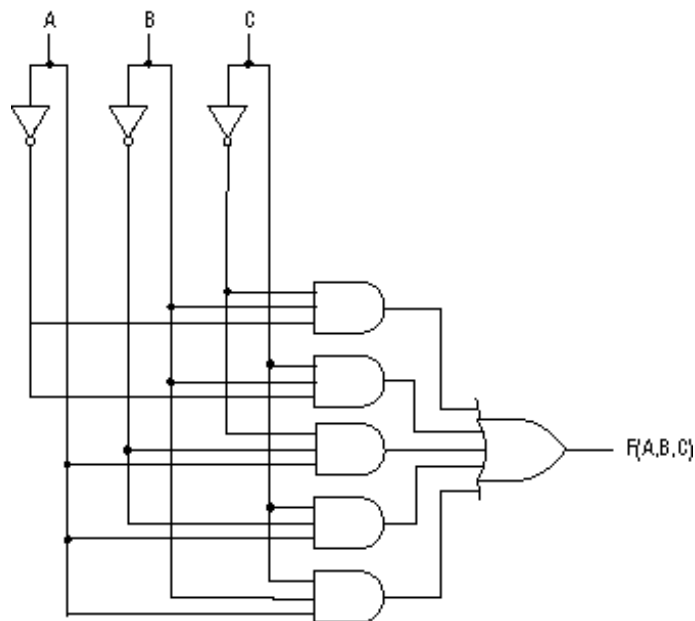
$F(A,B,C)=1$  のプロダクトタームの定義を記述すると

PT0 OR PT1 OR PT2 OR PT3 OR PT4 に等しい場合のみ F(A,B,C)は TRUE です。

従って、ブール演算子を使用すると上記の表現は以下のように厳密に表現できます。

$$F(A,B,C) = !A\&B\&!C \# !A\&B\&C \# A\&!B\&!C \# A\&!B\&C \# A\&B\&C$$

この関数の各プロダクトタームには関数定義の 3 つの論理関数がすべて含まれるので、この表現は、完全に変換関数 F(A,B,C)を正準形で定義しています。後で、これらのプロダクトタームの一部は削除でき式は、 $F(A,B,C)=A\&!B\&!A\&B\&C\&B$  と簡潔に表現できることを証明します。上記の式を縮小しないで一般的なインプリメンテーションを実行すると図 8-5 のようになります。



Universal Gate Implementation of the Three-Variable Function

図 8-5 F(A,B,C)のインプリメンテーション

F(A,B,C)に使用されたステートメントは 5 つのプロダクトタームの論理 OR を明確に表わしています。この表現は、プロダクトの算術加法と相似であるので、通常 Sum Of Products(SOP)表現と呼ばれます。複数のお AND ゲートの出力を OR ゲートで結合するグラフィック SOP 表現は、PLD アーキテクチャの基本要素です。

SOP 表現は、同じ論理関数の表現の多くの表現のうちのひとつにすぎません。その他の表現は、図 8-3 で示される法則を適用すると作成することができます。プロダクトターム PT0 に、第 2 モルガンの定理を適用してみます。

$$!!(!A\&B\&!C) = !(A\#!B\#C) = !ST0$$

ここでは、元のプロダクトタームに 2 つの COMPLEMENT(反転)が付き、

仮定 A により、プロダクトタームの論理値は変化しません。これは、数値の算術二重否定と等価です。丸括弧内の式は、算術加算の相似であるために通常、SUM TERMS または MAXTERMS と呼ばれ、導出される要素(図 8-2)としてハードウェアに等価なものがあります。ハードウェアや PLD のマニュアルでよく使われる OR TERM という名前の方をよく目にするかも知れません。

同様に残りのプロダクトタームは以下のように表現できます。

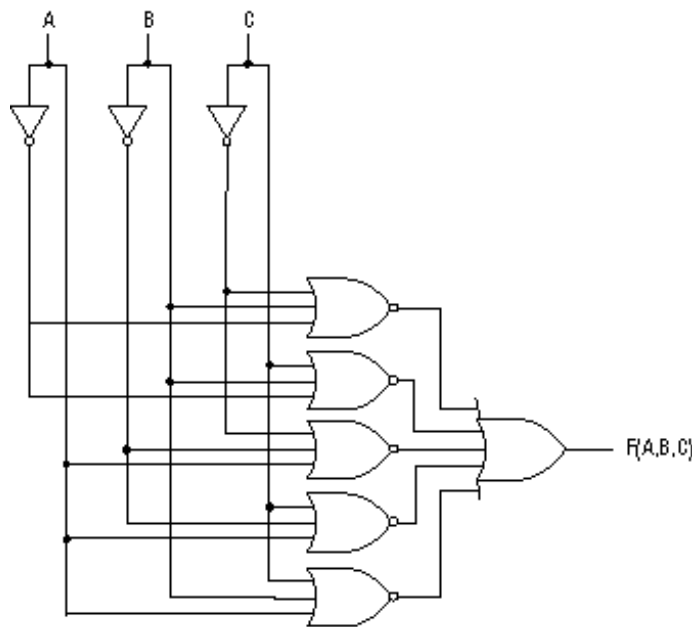
$$\begin{aligned} \neg(\neg A \& B \& C) &= \neg(A \# \neg B \# \neg C) = \neg ST1 \\ \neg(A \& \neg B \& \neg C) &= \neg(\neg A \# B \# C) = \neg ST2 \\ \neg(A \& \neg B \& C) &= \neg(\neg A \# B \# \neg C) = \neg ST3 \\ \neg(A \& B \& C) &= \neg(\neg A \# \neg B \# \neg C) = \neg ST4 \end{aligned}$$

上記表現の右辺のそれぞれは変数 A,B,C の反転値の反転された OR です。従って、関数 F(A,B,C)は以下のように表現できます。

$$F(A,B,C) = \neg ST0 \# \neg ST1 \# \neg ST2 \# \neg ST3 \# \neg ST4$$

第一のドモルガンの定理を上記の式に適用すると以下のように表わせます。

$$F(A,B,C) = \neg(ST0 \& ST1 \& ST2 \& ST3 \& ST4)$$



Logic diagram for F(A,B,C) implementation 1

図 8-6a 論理要素 F(A,B,C)のインプリメント 1

この表現は、論理変換関数の inverted Product Of Sums(IPOS)と呼ばれ、別の基本論理要素を使用して同じ変換関数をインプリメントする方法の一つです。論理関数を使用した F(A,B,C)のインプリメントを図 8-6a と図 8-6b に示します。

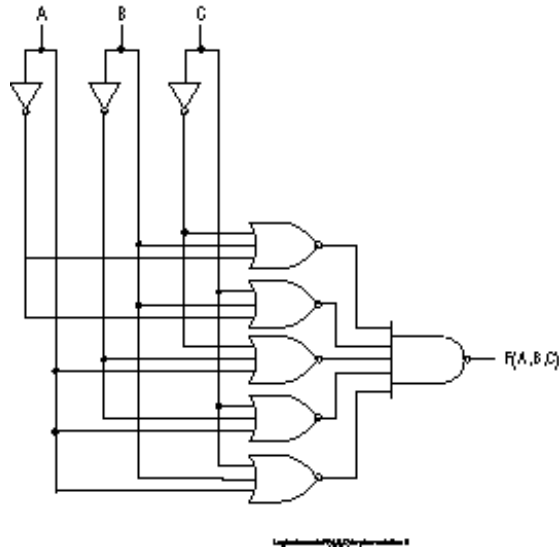


図 8-6b 論理要素  $F(A,B,C)$  のインプリメント 2

Product Of Sums(POS)、Inverted Sum Of Products(ISOP)と呼ばれる 3 番目と 4 番目の表現は同様にブール代数の理論を用いて導出されます。結果を示すと、以下ようになります。

$$F(A, B, C) = !A \& B \& C \# !A \& B \& C \# A \& B \& C \# A \& B \& C$$

変換関数  $F(A,B,C)$  の異なる論理実現の方法を見返すと、形式にかかわらず、変換関数を絵ミュレートする論理要素の 3 つの連続した段階が必要になることがわかります。理論的には、一つのゲートへの入力の数に制限が無い場合、任意の組み合わせ変換関数は、反転、AND ゲート、OR ゲートの論理要素の 3 段階だけで実行することができます。このような基本 3 段階構造は論理レイヤと呼ばれます。

実際には、一つのゲートに使用できる入力数(TTL や CMOS 回路では最大 8 本が普通)には制限があります。その結果、8 個以上の変数を持つプロダクトタームやサムタームのある変換関数では、タームの中で多重の AND や OR を行なう必要があります。これにより論理レベルの追加が必要になりこれに付随するシステムの縮小が必要になります。

PLD の重要な機能に内部の AND/OR レイヤの fan-in、fan-out の制限を緩和できる機能があります。これにより、一つの論理レイヤに最大で 32 個の変数のプロダクトタームを持つ変換関数をインプリメントできます。従って、シングルレイヤアーキテクチャにより、変換関数の実行がより速くなります。

## シーケンシャル回路とステートマシンの設計

前章で説明された論理設計の理論面の原理や変換関数のインプリメントを表わす簡潔な回路は、論理回路の分野では組み合わせ回路と呼ばれる部分です。組み合わせ回路とは、論理回路の出力すなわち論理変換関数の値が入力値の組み合わせにより完全に決められる回路のことです。実際の論理

設計でも純粋な組み合わせ回路で表現できる大きなシステムの一部に出くわすことがあります。しかし、ほとんどの大規模論理システムでは、入力変数の論理値や前に入力された入力変数の値により生成された中間結果や最終結果に応じてその出力が影響されます。簡単に言うと、on/off の決定は現在の状態と以前の状態を比較することで決まります。また、このような比較を実行するために以前に状態をそれが必要になるまで保存しておく必要があります。

論理を保存する要素を持つデジタル回路や入力変数と前の出力変数との組みに依存する変換関数を実行するデジタル回路はシーケンシャル回路と呼ばれます。この名前は、これらの回路が入力変数の値と入力を与えられるシーケンスに依存するところからきています。

シーケンシャルシステムには、新しく変数の組みが入力されるタイミングと現在の値が以前の値になるタイミングがわかるようにする必要があります。タイミングの制限を付けなくてもシーケンシャル回路を設計することは可能ですが、その演算は非常に複雑で難解なものになります。タイミングパルスはCLOCKSと呼ばれ、回路は同期シーケンシャル回路として定義されます。(この反対に外部のタイミングソースを使用しない非同期シーケンシャル回路があります。)

これまでの説明から解るように、ほとんどの組み合わせ回路は非同期回路です。というのは、ほとんどの組み合わせ回路は、入力が有効である限り有効な出力が保証されるからです。多くの場合、組み合わせ回路では外部の時間を参照する必要はなく、入力それ自身が出力のタイミングを決定します。ただし、外部のタイミング信号が不可欠な組み合わせ回路もあります。

1 つの例として組み合わせ回路でパイプラインレジスタを使用する場合があります。この場合、ストレージ要素(レジスタ)も外部クロックも使用しませんが、回路のノードの以前の状態が現在の状態に何も影響しないため、システムがシーケンシャルであるということを意味します。このようなシステムは同期組み合わせ回路に分類されます。

2 種類の論理回路の主な違いは、組み合わせ回路の変換関数が、厳密な変数として時間をするかどうかではなく、シーケンシャル回路の動作が、同期タイミングまたは非同期タイミングで信号を処理するかどうかにあります。

今まで、我々は時間に関連して基本論理ゲートの動作について検討しませんでした。一般に、これらの回路の出力は入力変数が入力されるとすぐに生成されるということが仮定されています。しかし、実際には入力が新たに入ってから出力が生成されるまでに、伝播遅れ  $T_{pd}$ 、すなわち要素の遅れと呼ばれる有限時間が存在します。製造技術により、この遅れは、数十ピコ秒から数十ナノ秒になっています。通常、好ましくない影響がありますが、この組み込みの伝播遅れにより、ストレージ要素やほとんどの基本シーケンシャル回路のフリップフロップを構築できます。

簡単な非同期シーケンシャル回路の例を考えてみます。まず、回路の真理値表の定義から始めます。回路には、その入出力の伝播遅れ  $t_{PD}$  があります。ブラックボックスが必要とする条件を以下に示します。

1. 回路には 2 つの入力 R と S と互いに反転の 2 つの出力があります。  
(Q,!Q)
2. 入力 R が TRUE で入力 S が FALSE の場合、出力 Q は FALSE で出力!Q は TRUE です。
3. 入力 R が FALSE で入力 S が TRUE の場合、出力 Q は TRUE で出力!Q は FALSE です。
4. R と S が FALSE の場合、出力 Q、!Q は以前の状態を保持します。
5. R と S が同時に TRUE の場合、(可能ですが)許されていません。このような場合出力は、意味がありません。このようにシステムで許可されていない意味のない変数の状態は DONT CARE と呼ばれ、X として表わされます。

上記の仕様に基づき、合成する回路の振る舞いを表現する表(表 8-2)を作成できます。表には出力 Q の 2 つの列があります。すなわち、Q(t)は出力の以前の値を示し、列 Q(t+1)は出力 Q に必要とされる現在の値が示されます。

S	R	Q(t)	Q(t+1)
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
0	0	0	0
0	0	1	1
1	1	0	X
1	1	1	X

表 8-2 基本ストレージ要素の真理値表

シーケンシャルシステムの特性から、出力の以前の値は、システムの真理値表や変換関数を定義する場合、入力変数として扱われます。このように、回路の変換関数を合成する場合、出力 Q(t)の以前の状態が入力変数として扱われます。

回路の現在の出力状態が TRUE である 3 つの例があります(表 8-2)。それから、以下のように変換関数 F(R,S,Q(t))を公式化します。

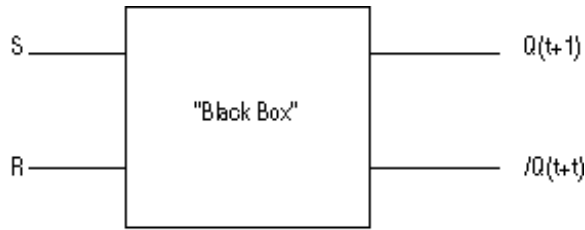
$$F(R, S, Q(t)) := S \& !R \& !Q(t) \# S \& !R \& Q(t) \# !S \& !R \& Q(t)$$

再び、変換関数の正準形が出てきました。関数の右辺と左辺の間の論理等価サイン(=)は、ここでは:=に置き換えられ置き換えの意味で使用されていることに注意して下さい。この新しいシンボルは、シーケンシャル回路の出力の変化が入力が与えられた時ではなく内部または外部のタイミング信号がある状態からある状態へ変化した時に起こることを意味します。

まず、第 2 の分配法則(A&1=A)を図 8-7 の、3 つのプロダクトタームに共通の!R に関して適用します。すると以下の式を得ます。

$$F(R, S, Q[t]) := !R \& [S \& !Q(t) \# S \& Q(t) \# !S \& Q(t)]$$





Black box definition of the basic storage element

図 8-7 基本ストレージ要素のブラックボックス定義

この法則を S についても適用すると(括弧の中の左の 2 つのプロダクトタームに)IK の式を得ます。

$$F(R, S, Q[t]) := !R \& [S \# !S \& Q(t)]$$

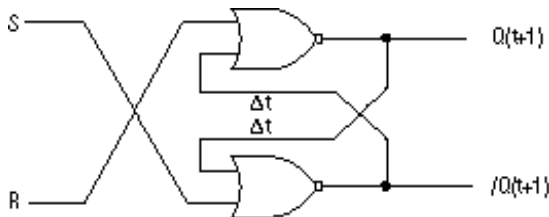
A#!A#1 従って、 $Q(t) + !Q(t) = 1$  です。従って、 $S \# !S \& !Q(t) = S \# !Q(t)$  となります。上式を書き直すと以下ようになります。

$$F(R, S, Q(t)) := !R \& [S \# Q(t)]$$

第 2 のドモルガンの定理  $!(A \& B) = !A \# !B$  を上式に適用すると以下ようになります。

$$F(R, S, Q(t)) := R + ![S \# Q(t)]$$

この形式で表現すると、変換関数は図 8-8 に示されるように、2 つの互いに交わる NOR ゲートの組み合わせになります。



Example of a real delay at the basic level in logic synthesis

図 8-8 第 2 のドモルガンの定理変換関数 2 つの互いに交わる NOR ゲート

最終的に得られた回路はよく知られた SET-RESET フリップフロップ回路です。この回路は、シーケンシャル設計を通じて使用される 4 つの基本ストレージ要素の一つです。その最も特徴的な機能は、その基本形では R-S フリップフロップは外部のタイミングソース(CLOCK)を必要としないことです。その代わりにある論理レベルからある論理レベルへの入力の遷移に依存しています。このような論理要素は通常パルスコントロール要素と呼ばれ、外部のタイミングソース信号によりコントロールされる他のストレージデバイスと区別されます。クロックコントロールドストレージデバイスでは、クロック制御回路の変化は、CLOCK の HIGH から LOW または LOW から HIGH への遷移の間にだけ起こります。従って、これらの回路では CLOCK が変化しない限り、入力での変化にかかわらず回路を制御することはできません。図 8-9 に、R-S フリップフロップの入出力端子での入力と出力の遷移のタイミングを示します。図の中で示される  $\Delta t$  の項は時間の最小単位を示し、非同期シーケンシャル回路の場合、2 つの入力の NOR ゲートの

伝播遅れと同じになります。また回路の時間を参照する目安にもなります。

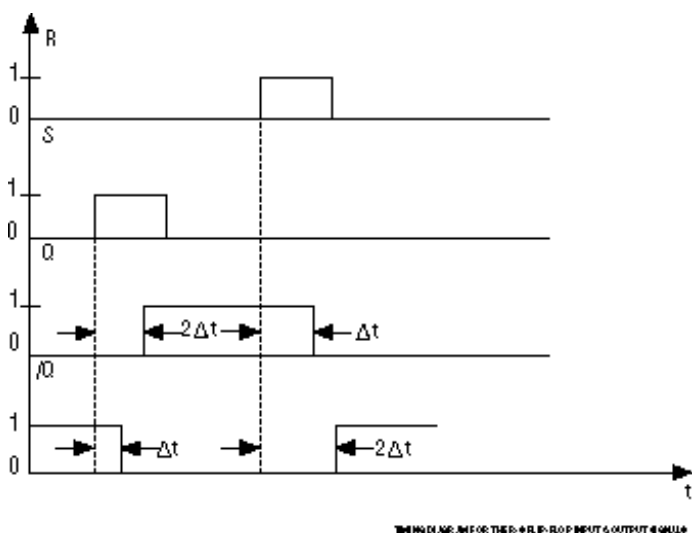


図 8-9 非同期シーケンシャル回路の一般化モデル

上記の例に基づいて、図 8-10 に示されるような非同期シーケンシャル回路の一般化モデルを導出できます。このような回路は以下のように構成されます。まず、組み合わせネットワークに入力される外部入力  $A(0) \dots A(n)$ 、出力の組み  $Q(0) \dots Q(m)$ 、出力一部が組み合わせ回路にフィードバックされるフィードバック回路  $L(0) \dots L(i)$  により生成されるフィードバック信号  $F(0) \dots F(i)$  です。非同期シーケンシャル回路の場合、これらのフィードバック状態は内部状態と呼ばれます。以前の出力の状態と現在の入力の状態を区別するには論理要素の内部遅れ  $t_{PD}$  を使用します。

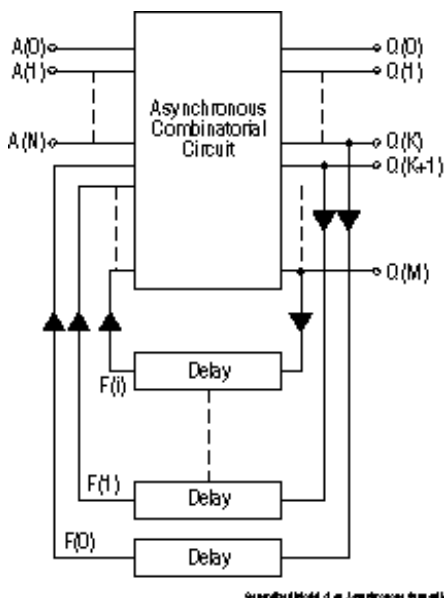
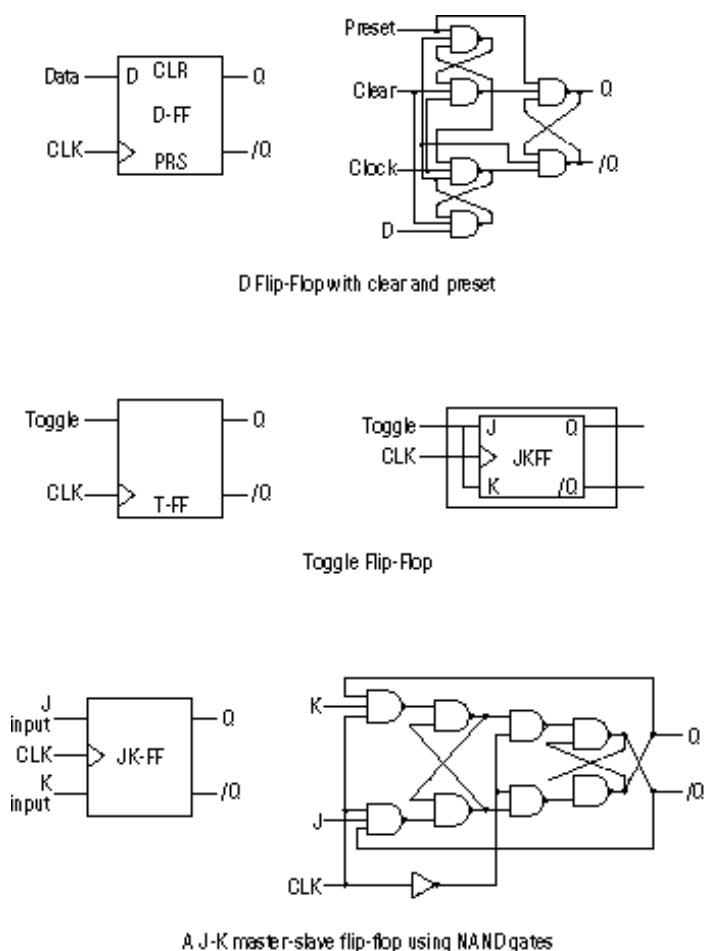


図 8-10 非同期シーケンシャル回路の一般化モデル

同期シーケンシャル設計のために開発された数々の先進のフリップフロ

ップ設計法がありますが、最もよく知られている 3 種類のクロック付きのフリップフロップ:D タイプフリップフロップ、T タイプフリップフロップ、J-K タイプフリップフロップをそれぞれの真理値表や可能なゲートレベルと一緒に図 8-11 で示します。シーケンシャル回路用に開発された出力レジスタを持つプログラマブル論理デバイスの場合、D タイプフリップフロップが最も一般的です。J-K フリップフロップと違い、それらは通常アレイの中のひとつのサムタームにより生成される、ひとつのデータ入力しか持ちません。このようなアーキテクチャについては次のセクションで説明します。



The Three Most Popular Clocked Flip-Flops

図 8-11 3 種類の最もよく知られたクロック付きフリップフロップ

これまでで、シーケンシャル回路と組み合わせ回路の違いが理解できたと思います。見てきたように、真理値表での組み合わせ回路の簡単な表現は、シーケンシャル回路の合成を簡単にするものではありません。シーケンシャル回路に必要な機能に基づき真理値表を作成ことは、その変換関数や回路図の合成には不十分です。

## ステートマシンの概念 2つの設計例

より複雑なシーケンシャル設計の場合、手作業で真理値表を式にすることは、複雑でエラーを起こしやすい作業になります。なぜなら、入力信号の可能な組み合わせは、ネットワークに外部から入力される入力の数だけでなくレジスタネットワークを介して組み合わせアレイにフィードバックされる以前の状態の出力の数に依存するためです。従って、比較的少ない外部入力を持つ回路の真理値表でも可能な入力状態は非常に大きくなります。

図 8-12 に、現在の入力と以前の出力を処理する組み合わせネットワークと以前の出力を保存するレジスタネットワークを持つ同期シーケンシャル回路のブロック図を示します。特定の設計要求に応じて、回路の出力の一部あるいは全部が以前の出力として扱われる場合があります。反対に、多くのアプリケーションでは、以前の状態は回路への入力として使用される状態だけです。ただし、論理変数として扱われない CLOCK 入力は別です。このような回路は、自律ステートマシンまたは Moore マシンと呼ばれます。これらの回路の典型例には、しばしばプリロード機能の無い dividers/counters が使用されます。一方、大規模なシーケンシャル回路では、それらの以前の状態と現在の状態の入力が現在の状態の出力を決めるために利用されます。このような回路は解析や合成が困難で Mealy マシンと呼ばれています。

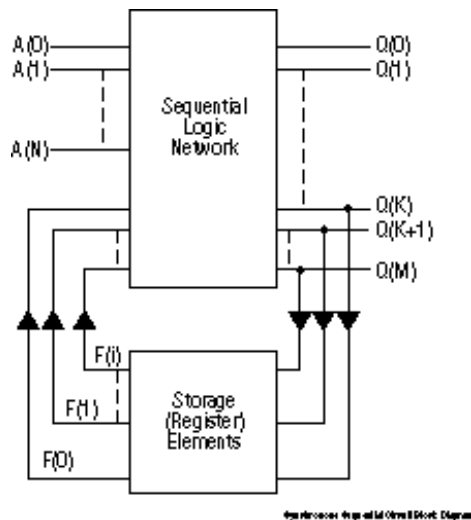


図 8-12 同期シーケンシャル回路のブロック図

以下にこれら 2 つの回路を扱う設計例を示します。それにより、変換テーブルを導入するとどのように設計が簡単になるかがわかります。また、ステートダイアグラムのグラフィック形式で回路の動作が表現されます。

### 例 1

入力のクロックパルスをカウントし現在のカウント値をバイナリ形式で出力するシーケンシャル回路を作成するとします。また、8 パルスをカウント後、回路は 0 にリセットされ再びカウントを開始するとします。最大のカウント値は 7 で 0 から始まるので、回路は 0(000)から 7(111)までの 2 進数

値をすべて表わすために3本の出力が必要です。また、CLOCK用に1本の入力が必要です。この回路の真理値表を公式化してみましょう。(表8-3)

CLK	以前の状態			現在の状態		
	Q2(t)	Q1(t)	Q0(t)	Q2(t+1)	Q1(t+1)	Q0(t+1)
—/	0	0	0	0	0	1
—/	0	0	1	0	1	0
—/	0	1	0	0	1	1
—/	0	1	1	1	0	0
—/	1	0	0	1	0	1
—/	1	0	1	1	1	0
—/	1	1	0	1	1	1
—/	1	1	1	0	0	0

表8-3 例1の真理値表

回路の動作が2組みの2進数ベクタ、すなわち、現在の入力(以前の出力)状態ベクタと現在の出力状態ベクタ、により定義されたとすると、表8-3の表現は簡略化されます。ベクタ(Q2,Q1,Q0)からそれらの出力の論理値の系列が形成されます。このように、表8-3は、表8-4で示されるより簡単な形式で表現することができます。

現在の入力	以前の状態/現在の状態
(111)	(111)/(000)
(000)	(000)/(001)
(001)	(001)/(010)
(010)	(010)/(011)
(011)	(011)/(100)
(100)	(100)/(101)
(101)	(101)/(110)
(110)	(110)/(111)

表8-4 例1の遷移表

表8-4の現在の入力と以前の状態で示されるような状態ベクタは我々の設計でも同じことが起こることがあります。ひとつの出力状態が組み合わせネットワークにより入力変数に応じて処理される複雑な設計では、必ずしもそのようになるとは限りません。表8-4で示される状態マシンシーケンスの表形式の表現では遷移表と呼ばれることがあります。また、状態ダイアグラムと呼ばれるグラフィック形式で表現されることもあります。状態ダイアグラムは、連続するクロックパルスと外部信号に応じた回路出力により規定される状態のシーケンスを示します。我々の設計の状態ダイアグラムを図8-13に示します。3ビットのカウンタの設計の場合、状態ダイアグラムは比較的簡単ですがMealyタイプの回路設計では非常に複雑になります。図の中の丸はマシンの出力のそれぞれの現在値を表わします。矢印は、マシンが遷移できる状態への遷移を示します。(この場合、次の状態への遷移だけが可能です。)

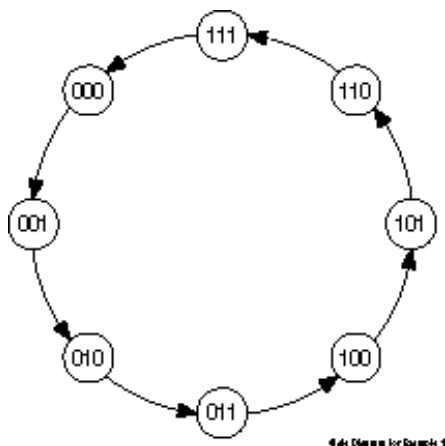


図 8-13 例 1 の状態ダイアグラム

回路合成の間、通常遷移表と状態ダイアグラムが使用されます。設計段階に応じて、回路の動作を確認するために遷移表や状態ダイアグラムが必要になります。

回路の動作を記述している間には、回路のカウントがステート(000)から実際にどのように始まるかについて記述する必要はありません。これは状態ダイアグラムでも同様で、初期状態をどのようにするかは状態ダイアグラムでは表現されていません。従って、回路の合成を行なう前には、必ず取り込まれる外部信号/RST を明記する必要があります。これが行われると、外部信号は同期的(最初の CLOCK パルスが入力されると)にすべての出力を 0 にします。

マシンの 3 種類の表現を使用して、出力 Q(0)、Q(1)、Q(2)の以前の状態のストレージ要素のような 3 個の D タイプのフリップフロップを使用する回路の式を作成することができます。以下の式により、上で示された真理値表と遷移表が実行されます。

$$\begin{aligned}
 Q0(t+1) &:= !Q0(t) \& RST \\
 Q1(t+1) &:= Q0(t) \& !Q1(t) \& RST \# !Q0(t) \& Q1(t) \& RST \\
 Q2(t+1) &:= Q0(t) \& Q1(t) \& RST \# !Q0(t) \& Q2(t) \& RST \# \\
 &\quad !Q1(t) \& Q2(t) \& RST
 \end{aligned}$$

上式は、表 8-3 から導かれる正準形から導出されます。この回路の多くの可能な論理図のひとつを以下の図に示します。このマニュアルの以降の部分で、レジスタード PLD によりどのようにそれが実行されるかを示します。

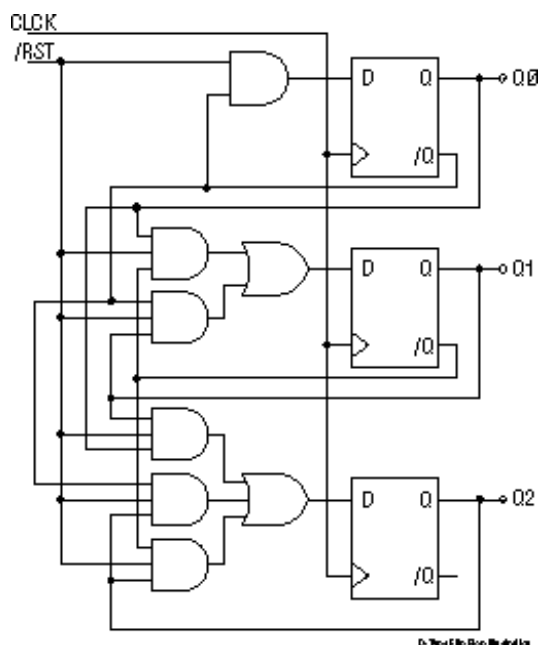


図 8-14 D タイプフリップフロップ

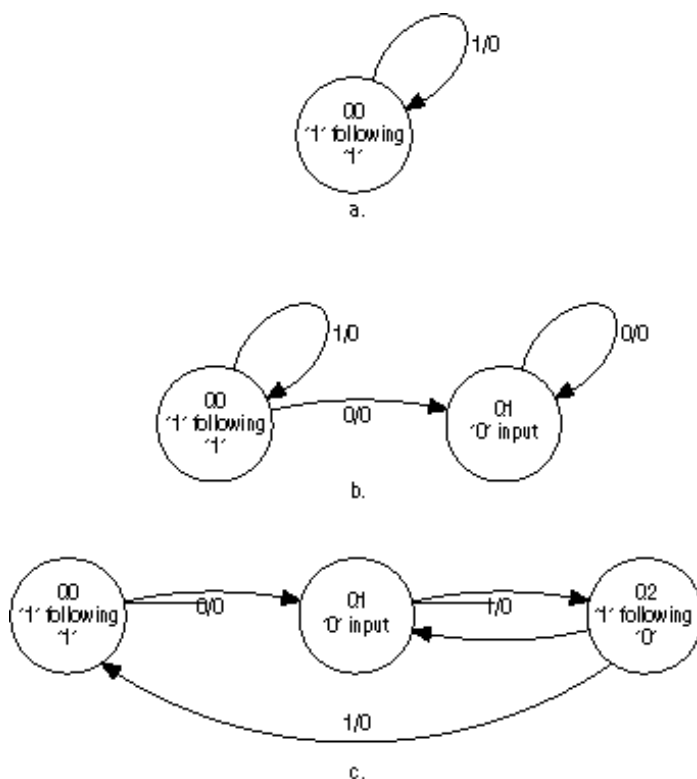
この簡単な例により、シーケンシャル回路を使用して、クロック入力がある間、出力に必要な信号を生成するアプリケーションが説明できます。次の例では Mealy タイプのステートマシンをどのように使用すれば入力信号のシーケンスをモニタし、設計者のアルゴリズムに応じて入力に反応する回路を設計できるかを示します。

## 例 2

シーケンスディテクタと呼ばれる回路について考えてみます。これらの回路はプロセス制御のアプリケーションでは非常に一般的な回路です。入力 X でテストされる値のシーケンスを 010 とします。すなわち、8 ビットシーケンス 01101010 には、印が付けられているようなシーケンスが 2 つ含まれます。回路の出力は TRUE ステート(すなわち、論理 1)を出力するとします。開始ステートと呼ばれる最初のステート、Q0 からステートダイアグラムの合成を始めて下さい。回路はこのステートでは、図 8-15a のこのステートに対応するサークルにより示されるすべてのクロックサイクルで入力が 1 になる状態でない限りこのままの状態を保ちます。小さなループを描いた矢印は入力の状態が常に 1 でその以前の出力状態は 0、従って、Q0 の状態は 0(1/0)のままであることを示します。しかし、入力状態が 0 に変化すると、この状態は選択されたシーケンスを完了するための状態のひとつになります。0/0(現在の入力 / 以前の出力)状態によりマシンはステート Q1 の現在の出力状態を 0 にします。(図 8-15b)2 つの状態 1/0 と印が付けられた矢印は遷移の条件を示します。

ステート Q1 に付けられたループした矢印は、次の Q0 の状態が 0 になる場合、Q1 の状態が変化しない事を示します。(0 のシーケンスは 010 シーケンスの元になることができるので)前の 0 に続いて 1 になる場合はいつでも、

求めるシーケンスはより似たものになります。従って、最終状態 Q2 が初期化され入力状態に対する 2 つの以前の状態がモニタされます。(図 8-15c) 次の入力が 0 になるか 1 になるかにより、回路は違う出力になります。次の入力が 0 の場合、シーケンスは検知され回路は 1 を出力します。(0 は次の 010 シーケンスの始まりになることもできるので)その出力はステート Q1 と見なされます。一方、入力が 1 の場合、シーケンスは検知されず、マシンはステート Q0 と見なされません。(図 8-15c)



©2014 by D. Diagrams, Japan Inc.

図 8-15a,b,c ステートダイアグラムシンセシス

この回路の状態遷移図を作成し、図 8-16 で示されるロジックダイアグラムやタイミングダイアグラムがここで示されるステートダイアグラムで示される関数を実際に実行するかを確認して下さい。



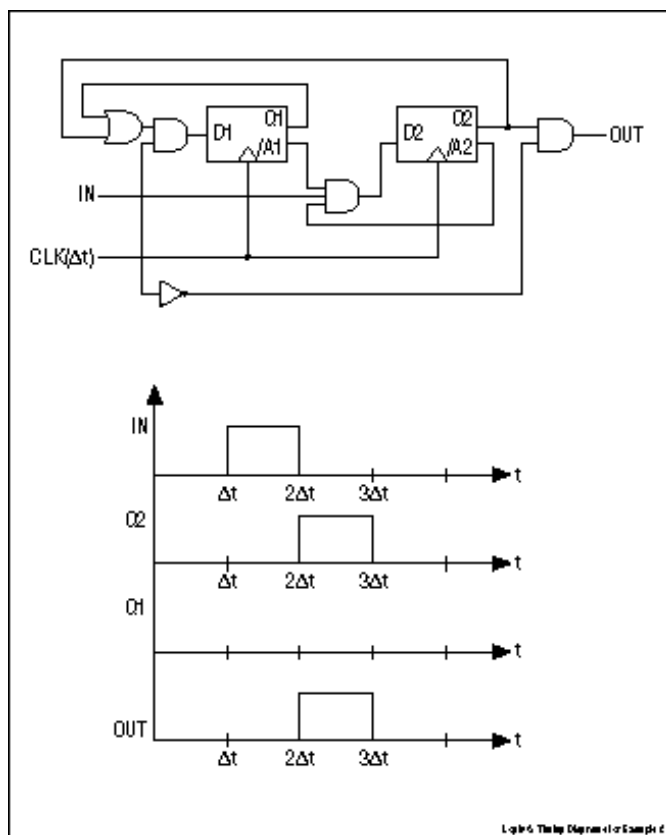


図 8-16 例 2 のロジックダイアグラムとタイミングダイアグラム

### 論理リダクション: より小さく、より良く

論理変換関数から余分な論理を削除するために使用される原理やアルゴリズムについて説明します。前のセクションでは、S-R フリップフロップの動作を解析する場合、このような縮小を手作業で行なっていました。論理の縮小は、ブール代数の中でも異なる理論と法則を使用して行われます。このように論理を縮小する上で難しい点は、その方法の高度に発見的な手順にあります。すなわち、試行錯誤です。そして、この方法を変換関数に適用して同じ変換関数の最小の表現に縮小します。最小の表現は真理値表で示される変換関数を表わしており、しかも最も小さい表現になっています。

現在のところ、論理を縮小する最善の方法は M.Karnaugh により開発された論理縮小法です。この方法は、ブール代数の基本公理をグラフィック形式で実行し、小さい論理変換関数を最小化する最もよく知られた方法です。Karnaugh マップ法は、4-6 変数の変換関数の解析に使用されます。ただし、6 変数以上の変換関数では複雑になり扱いにくいものになります。

理論的観点から、Karnaugh マップは与えられた設計や変換関数の真理値表の別表現であるといえます。しかし、この関数の値を表現する表は、設計者が簡単に  $A \neq 1$  の公式を使用して式を縮小できる状態を認識でき

るような形式で表現されます。

再び、 $A \& !B \# !A \& B \# B \& C = 1$  の正準形が

$F(A,B,C) = !A \& B \& !C \# !A \& B \& C \# A \& !B \& !C \# A \& B \& C$  で表現されることを考えてみます。それからその真理値表も同時に考えてみます。これを参考にして以下の例の Karnaugh マップを作ってください。

変換関数の Karnaugh マップを作る第一段階は、それに含まれるバリュースボックスの数を決めることです。このボックスの数は、変換関数が正準形で表わされた時にそこに含まれるプロダクトタームの数を表わします。従って、3 変数の関数の場合、PTs の数は 8 である必要があります。また、マップは 2 つの列にそれぞれ 4 つボックスがあるマップになります。次に各ボックスを変数、すなわち、ひとつの minterm、の値の組み合わせに対応する位置に割り付けて下さい。

ボックスの位置は、隣合う 2 つのボックスの違いが 1 ビット以下になるように配置して下さい。プロダクトタームでの位置は無関係です。2 進数表記の特性により、変数の数やマップ内のボックスの数に関係なく、このような配置は常に実現可能です。図 8-17 に関数  $F(A,B,C)$  のマップを示します。

$!A \& !B \& !C$ (000)	$!A \& B \& !C$ (010)	$A \& B \& !C$ (110)	$A \& !B \& !C$ (100)
$!A \& !B \& C$ (001)	$!A \& B \& C$ (011)	$A \& B \& C$ (111)	$A \& !B \& C$ (101)

図 8-17 3 変数論理関数の Karnaugh マップ

Karnaugh マップのバリュースボックスにラベルを付ける方法はたくさんあります。ここで使用されるラベル(右上の隅で括弧に囲まれた 2 進数値)は、マップの理解には役立ちますが、あまり一般的ではありません。

値のひとつの変数だけが違う隣接する 2 つのボックスは互いに反転の関係にあります。さらに、Karnaugh マップを 3 次元的に見ると、右側のボックスの底は、最上部の左側のボックスの隣であると考えられます。これは、ひとつの違いの条件に合っています。

空の Karnaugh マップの 3 変数表現を、設計の解析に使用することができません。設計の解析に使用するには、表 8-1 の真理値表を参照して変換関数が TRUE になるように対応するプロダクトタームのボックスすべてに 1 を挿入し、その他のボックスに 0 を挿入して下さい。マップは例えば図 8-18 のようになります。

(000)	(010)	(110)	(100)
0	1	0	1
(001)	(011)	(111)	(101)
0	1	1	1

図 8-18 関数  $F(A,B,C)$  の Karnaugh マップ

解りやすくするために、変数名はボックスの表示から削除しています。変換関数の縮小はこれで比較的簡単になります。マップの 1 つのボックス

は、変換関数  $F(A,B,C)$  の正準形のプロダクトタームを表わしているので、縦と横で隣合う 1 は、異なる変数の部分に関しては縮小できるタームであることを表わしています。1 を含む隣接するボックスのすべてを丸で囲んで下さい。丸で囲まれたそれぞれのペアは 2 変数だけを含むひとつのプロダクトタームを表わします。このプロダクトタームは  $X+/X$  変数が取り除かれてできました。隣接するボックスと組み合わせることができなかったプロダクトタームは何もしないまま残っています。

このように、図 8-18 で示される Karnaugh マップでは、縮小された Sum Of Products 表現は以下ようになります。

$$F(A,B,C) = !A\&B \# A\&C \# B\&C \# A\&!B$$

プロダクトターム  $AC$  があるので、この式は明らかに変換関数  $F(A,B,C)$  として考えられるもとの形とは違います。真理値表をみると、ターム  $A\&C$  は削除できる項で 3 つの項に共通に存在していることが解ります。何か違うのでしょうか。ボックス(111)は縮小できるタームの 2 組で使用されているので、1 つのボックス(111)を 2 回使用していることに注意して下さい。多くの場合、このようなことは必要ありません。そして、合成される変換関数の縮小できるタームが挿入されます。このような場合のための固定された規則はありませんが、Karnaugh マップを使用する場合の一般的なガイドラインのようなものがあります。それを以下に示します。縮小されるターム  $A\&C$  はブール代数の基本理論により、式  $F(A,B,C)=!A\&B\#A\&C\#B\&C\#A\&!B$  から削除することが出来ます。そして、読者はそうするようにアドバイスされます。もちろん、縮小された形式は、ターム(111)を(101)と組み合わせて決められるだけでなく、図 8-18 に概略が示される 3 つのペアを使用して形成されます。

4 個以上の変数で構成される大きな Karnaugh マップの場合、隣のタームやペアが 4 重や 8 重になることがよくあります。このような場合、4 重になったところは、2 つの互いに背反な変数で縮小できます。また、8 重になった部分は 3 つの変数に縮小できます。図 8-19 から 8-23 に、このやり方を用いた論理縮小の異なる場合を示し、式の最小化が完了するために要する段階を図示します。

Starting Function:  $F(A,B,C,D) = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D +$   
 $\bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D$

1	(0000)	0	(0001)		(0011)	1	(0010)
1	(0100)	0	(0101)		(0111)	1	(0110)
1	(1100)	0	(1101)	1	(1111)	1	(1110)
1	(1000)	0	(1001)	1	(1011)	1	(1010)

Resultant Function:  $F(A,B,C,D) = \bar{D} + AC$

Logic Reduction Using Karnaugh Map-1

図 8-19 Karnaugh マップ 1 を使用する論理の縮小

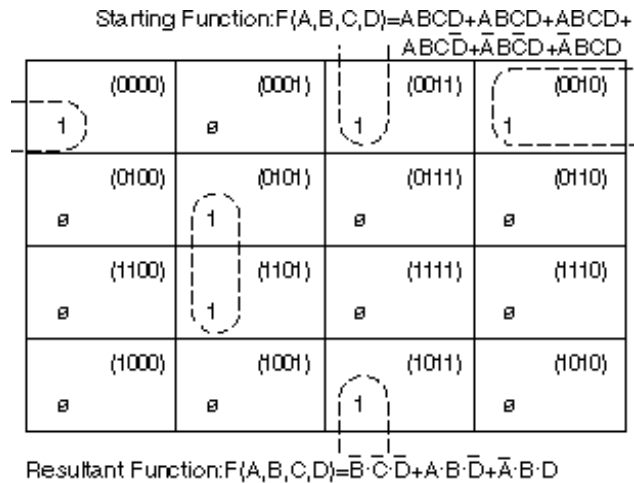
Starting Function:  $F(A,B,C,D) = \bar{B}\bar{D} + ACD + ACD$

1	(0000)	0	(0001)	0	(0011)	1	(0010)
0	(0100)	0	(0101)	0	(0111)	0	(0110)
1	(1100)	1	(1101)	0	(1111)	0	(1110)
1	(1000)	1	(1001)	0	(1011)	1	(1010)

Resultant Function:  $F(A,B,C,D) = \bar{B}\bar{D} + AC$

Logic Reduction Using Karnaugh Map-2

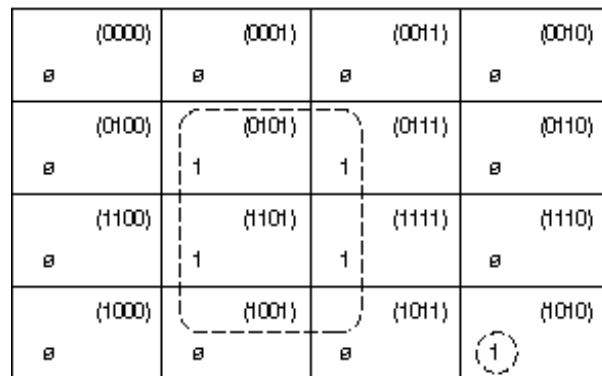
図 8-20 Karnaugh マップ 2 を使用する論理の縮小



Logic Solution Using Karnaugh Map-2

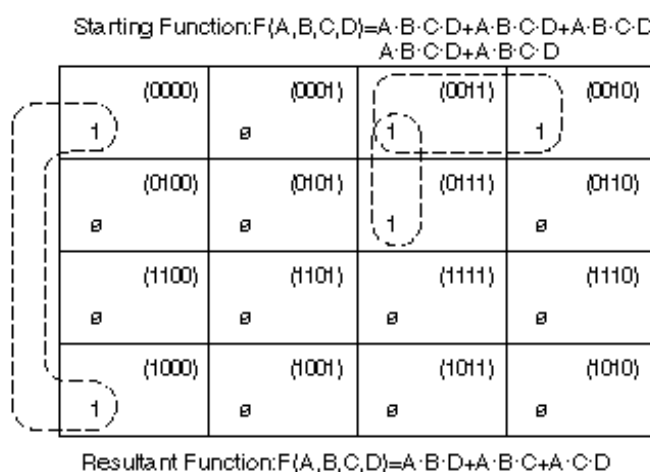
図 8-21 Karnaugh マップ 3 を使用する論理の縮小

Starting Function:  $F(A,B,C,D) = \bar{A} \cdot B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot C \cdot D + A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot C \cdot D$



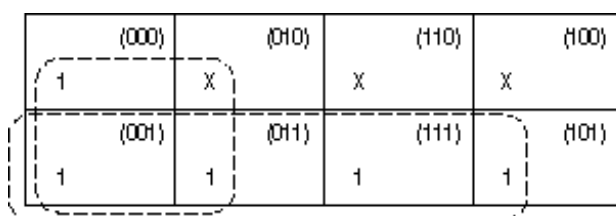
Logic Solution Using Karnaugh Map-4

図 8-22 Karnaugh マップ 4 を使用する論理の縮小



Logic Reduction Using Karnaugh Map 5

図 8-23 Karnaugh マップ 5 を使用する論理の縮小



Logic Reduction Using Karnaugh Map 6

図 8-24 Karnaugh マップ 6 を使用する論理の縮小

真理値表の変数が X(DONT CARE)の場合、X がボックスの中に入力されます。これらのボックスは隣接する 1 のペア(または 4 重のペア)を作る必要がある場合は 1 として扱われます。その他の場合は無視されます。図 8-23 に 3 変数関数の縮小について説明します。

Karnaugh マップを使用する論理の縮小に熟練するには、多くの経験が必要です。しかし、変換関数の最小化に必要な一般的なガイドラインがあります。

1. 他のボックスと結合することができない 1 のボックスをすべて見つけて丸で囲んで下さい。
2. 他の 1 つのボックスとだけ結合できる 1 のボックスを見つけて下さい。このペアのボックスをグループ化し、後で他の方法で結合できるようになるまでそのままにしておいて下さい。
3. 4 重に結合できるボックスを見つけて下さい。これはステップ 2 でまだグループ化されていないボックスで行なって下さい。これらのボックスを使用して 4 つのボックスのグループ化を行なって下さい。
4. グループ化されていないボックスについて 8 個のボックスのグループ化を行なって下さい。

5. グループに入らないボックスについて、カバーされていない1のボックスをなるべく多く含むようにそしてなるべく大きなグループをつくるようにグループ化を行なって下さい。ただし、それらの番号はなるべく小さくなるようにして下さい。

上記のアルゴリズムのステップ5は、我々が扱ってきた変換関数では失敗します。上記のようなプロセスを繰り返すと以下のような式が選られます。

$$F(A, B, C) = A \& B \# !A \& B \# B \& C.$$

多くのその他の表形式やグラフィック形式の方法が、ブール式を縮小するために開発されてきました。これらの方法のすべては上で説明したKarnaugh マッププロセスの単純なバリエーションです。論理変数が5-6になると処理が複雑になるのはどれも同じです。

ソフトウェアで実行するのに適した多くの変数を扱う設計に適用する新しいアルゴリズムが開発されてきました。これらのアルゴリズムのうちQuine-McCluskey アルゴリズムと PRESTO アルゴリズム、Harris proprietary Cracow アルゴリズムの3種類は論理設計用のCAEツールの基本になっています。これらプログラムの原理や内容について言及するのはこのマニュアルの範囲を越えています。興味のある読者は適当な文献を参照して下さい。(例えば、{B1}や{H1}、{R1}、{DIO1}などがあります。)PC やメインフレームのコンピュータで使用されるこれらのプログラムは、最小化する前にバイナリ形式(すなわち、 $A \& !B \& !C \& D \& E$  ではなく 10011)で表現されるプロダクトタームの展開された式で与えられる変換関数が必要です。しかし、コンパイラに含まれるアドバンスドPLD最小化アルゴリズムなどのこれらの最小化アルゴリズムの中には、このような変換を自動的に行なうプリプロセッサやプロセッサを持っているものもあります。

PLD の場合、論理の最小化は設計のプロセスの中で不可欠なものであり多くの場合、与えられた設計を実際のデバイスにフィットできるかどうかを示します。ほとんどのPLD設計では非常に多くの入力変数(10-16)や1つの出力に対して最大16個のプロダクトタームを扱うので、手作業で最小化の作業を行なうのはほとんど不可能です。このような作業は、FPLA の設計でも困難になりつつあります。FPLA では多くのプロダクトタームを複数の出力に分配するためにプロセスの最小化が別の次元で行われています。

## 論理の落とし穴

名前から解るように、論理の落とし穴とは論理設計者がよく陥る設計の落とし穴のことです。これらの解りにくく見つけにくい落とし穴は、その論理回路がどんなによく理解され検証されていても論理回路の性能を著しく悪化します。

静的な論理回路の落とし穴、すなわち突然の故障は、以下のような設計上の欠陥が論理回路内で影響しあって起こります。

1. ある要素の入力到達する前に信号が通過する経路の論理レベルの数が同じでないために起こる異なる伝播おくれ。
2. 回路の同じ論理レベル内の回路要素の異なる伝播遅れ。

3. 温度や湿度、電源電圧など周囲の条件により発生する異なる要素の伝播遅れのばらつき。
4. 信号の遷移の方向の違いによる同じ要素での伝播遅れのばらつき。(すなわち、NAND ゲートの LOW から HIGH の信号と HIGH から LOW への信号の伝播遅れの違い。)

上記の状態により、ひとつの論理要素の伝播遅れよりも短い信号が生成される可能性があります。この様に短い信号は、シーケンシャル回路のご動作の原因になります。設計が確実に動作するように検証されている場合、回路出力が入力信号に依存する組み合わせ非同期論理回路と違い、同期回路は入力信号の最初のトランジション(エッジ)に依存します。従って、回路内で生成される意味不明の故障でも、それに続くクロック付きのフリップフロップ(すなわち、シーケンシャル回路)は暴走し、全体としてでたらめな状態になり完全に間違った演算を行なってしまいます。

上記の欠陥 1 のために論理の落とし穴が生成され回路内に残ってしまった典型例を考えてみます。この回路は一見非常に簡単そうですが、PLD で設計する場合最もよく使われる配置 N ひとつを表わしています。適切に使用されない場合、この回路により、大規模なシーケンシャル設計が不調になる可能性があります。図 8-25 でわかるように、回路は 2 つのプロダクトターム(!A&C と B&!C)で構成され、出力 F によりひとつの OR タームがこれらのプロダクトタームにフィードバックされます。変換関数は以下のように定義されます。

$$F(A, B, C) = !A \& C \# B \& !C$$

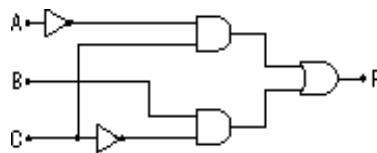
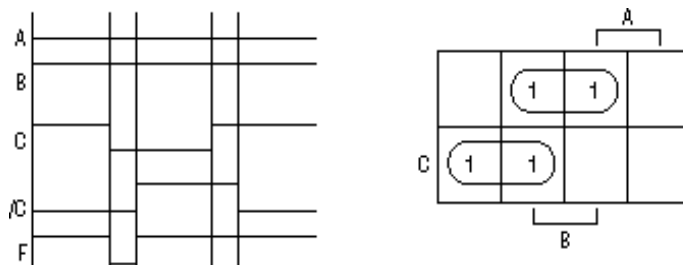


Figure 8-25: Combinational Logic Circuit

図 8-25 間違いやすい論理回路

この回路のタイミングダイアグラムから各論理要素(図 8-25)の最小の伝播遅れを  $t$  とすると、F を TRUE にするために A,B,C の信号を同時に入力したとしても出力に極短い FALSE が現れることになるのは明らかです。たとえば FALSE レベルが  $t$  だけ続いたとしても、D フリップフロップをトリガーするには十分です。これにより、フリップフロップは間違った状態にセットされることになります。





Timing Diagram and Karnaugh Map for the Hazard-Free Circuit in Figure 8-24

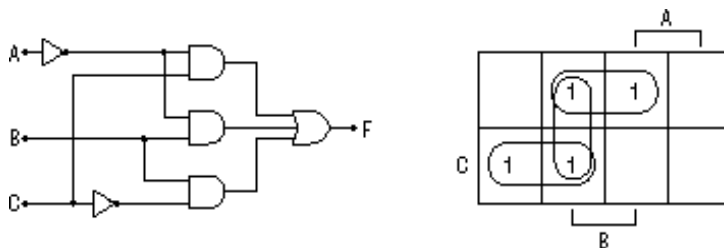
図 8-26 図 3-24 の間違いやすい回路のタイミングダイアグラムと Karnaugh マップ

短い FALSE レベルの原因は、信号 C が AND ゲートの入力 a に到達する時間とその反転信号が AND ゲートの入力 b に到達する時間の間にずれがあるためです。このように非常に短い変換関数がエラーの原因になることがあります。

このようなことを回避することはできるのでしょうか。変換関数(図 8-26)の Karnaugh マップにより、論理的な観点から 2 つの縦の 1 のボックス (!A&B)の組み合わせから導き出される削除可能なタームがあるものの、2 つのターム !A&C と B&!C は、変換関数の最小の形を実際には表わしていることがわかります。しかし、変数 C の反転値に対応する立て方向に隣接する 2 つの 1 のボックスが落とし穴の原因です。従って、ターム (!A&B)の縮小された組み合わせに対応する 3 番目のプロダクトタームを導入することにより落とし穴を回避できます。最終的な変換関数を以下に示します。

$$F'(A,B,C) = !A\&C \# B\&!C \# !A\&B$$

ただし、この表現は最小ではありません。論理の落とし穴は回避しています。動作中にエラーを起こさないことを証明する図 8-27 で示される回路のタイミングダイアグラムを描いて下さい。



Logic Diagram and Karnaugh Map of a Hazard-Free Version of the Circuit in Figure 8-24

図 8-27 図 8-25 のエラーを起こしやすい回路をエラーを起こしにくく改良した回路の論理ダイアグラムと Karnaugh マップ

一般的な結論は、論理の最小化を行なう時に Karnaugh マップで結合されなかった正準形のプロダクトタームは論理的なエラーを起こしやすいということです。従って、それらは回路に合成する前によく調べる必要があります。もっと多くの変数を扱う Karnaugh マップの場合、このように各プロダクトタームを調べることは不可能です。最小化された式や論理ダイアグラムを直接調べるのが唯一の答えです。

論理設計者はCAEツールで実行した論理の最小化の結果は十分に調べる必要があります。実際に最小化が完了するまではプロダクトチームの結合を調査する方法はありません。

論理の回路の設計でエラーを起こすような論理的な落とし穴は多くの設計者により検討されてきました。今回の場合はそうではなかったのですが、実際には、論理の落とし穴の多くは、設計上の見落としや論理ネットワークの物理的な振る舞いを十分に行なっていなかったことが原因になっています。すなわち、回路が動作する技術や設計、環境により規定される物理的な性能限界を十分に理解していないということです。

## ステートマシンの設計

ステートマシンを使用して短い設計サイクルを実行することでPLDの潜在能力をフルに発揮できます。式レベルでの表現は必要ありません。システムの振る舞いの記述を式にしPLDに直接それをインプリメントできます。

今日では、プログラマブルロジックデバイス(PLD)を使用する統合化された論理デバイスの設計を簡単に行なうことができます。ステートマシンを使用する設計は、CUPL言語のようなコンパイラを基本とするPLD設計言語と組み合わせることができます。この組み合わせにより、論理設計の基本ゲートや式の割合を減らすことができます。設計をシステムレベルの記述から実際のPLDインプリメンテーションへと進めることができます。

ステートマシンを基本とする設計の有利な点は、ステートマシンによる設計では、将来のユーザがシステムを理解する上で参考になるドキュメントが作られることです。アセンブラを基本とする設計は、まず始めに言語を習得するのが難しいので、あまり用いられていません。CUPLは一般の簡単なステートマシンモデルを作成する場合、簡単に設計できます。図8-28に簡単なステートマシンモデルの例を示します。CUPLを使用すると、設計モデルを必要な仕様のアプリケーションに調整することができます。

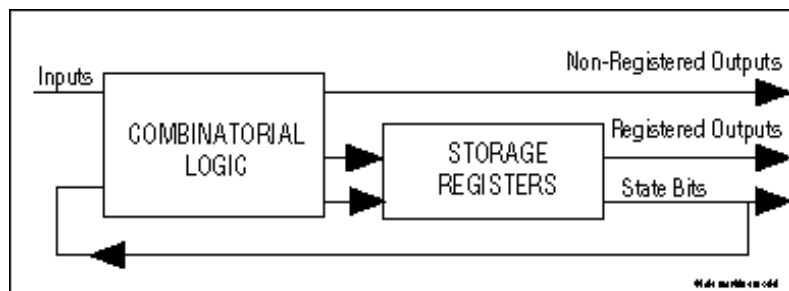


図8-28 ステートマシンモデル

図8-28で示されるようなステートマシン理論は複雑に見えます。CUPLを使用すると、ステートマシン理論の複雑な詳細を簡単にすることができます。例のような簡単なモデルとCUPLの学びやすいシンタックスを使用すれば、ステートマシンモデルの設計が早くなります。また設計の検証やインプリメントも早く行なうことができます。

## ステートモデル設計

通常、ステートマシンは論理回路をフリップフロップを用いて結合します (図 8-29 参照)。古典的な論理設計では Karnaugh マップなどのブール代数を縮小するテクニックを必要とします。

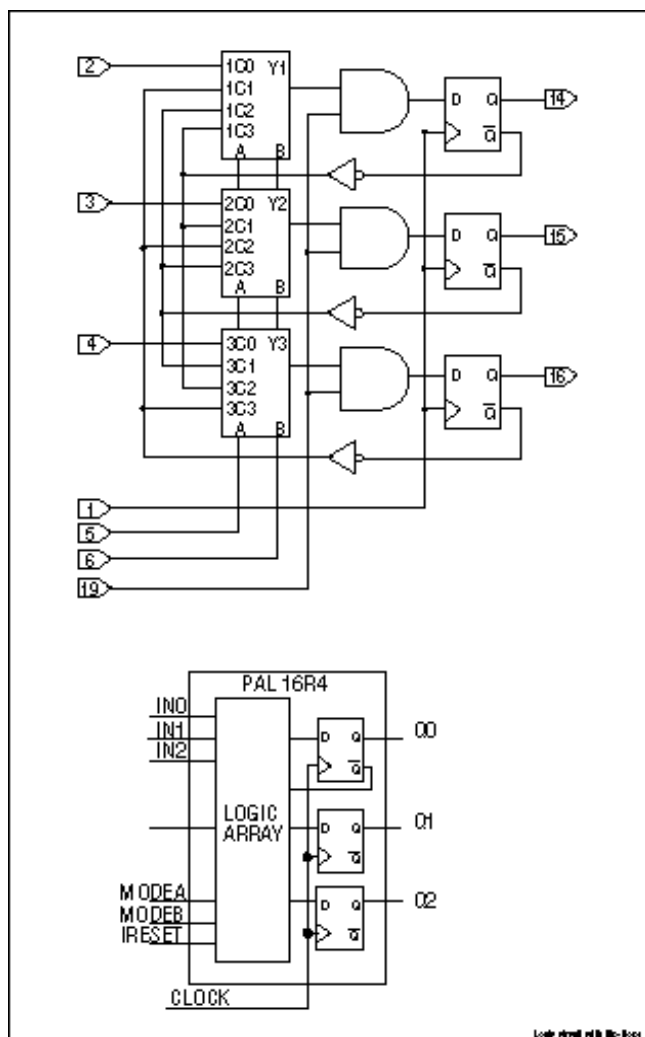


図 8-29 フリップフロップを使用した論理回路

フリップフロップの出力は、自分自身の入力やその他のフリップフロップの入力として使用することができるので、フリップフロップの入力値は、自分自身の出力とその他のフリップフロップの出力に依存します。従って、フリップフロップの最終的な出力値は自分自身やその他のフリップフロップの前の状態により決定されます。

ステートマシンを使用して設計を行なうと、論理式レベルの記述は必要なくなり、新しく設計した論理設計を直接 PLD にインプリメントできます。

CUPL により定義されたステートマシンモデルは 6 個のコンポーネントを

使用します。

- 入力
- 組み合わせ論理
- フリップフロップ(ストレージレジスタ)
- ステートビット
- レジスタード出力
- ノンレジスタード出力

図 8-30 にこれら 6 個のコンポーネント間のタイミングの関連図を示します。入力は他のデバイスからの出力され、デバイスに入力される信号です。

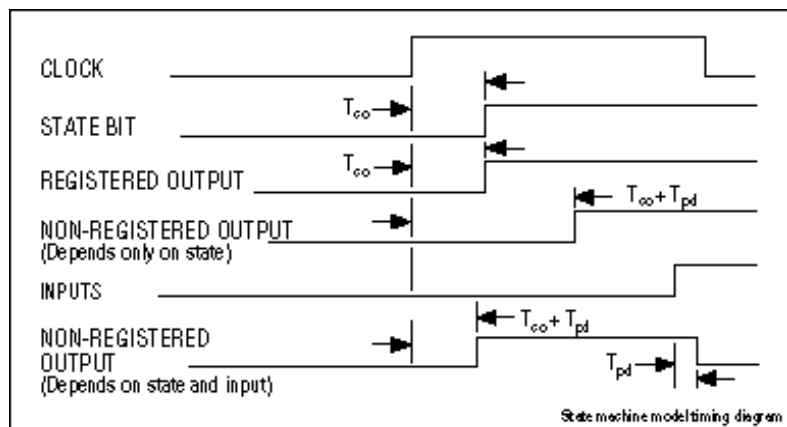


図 8-30 ステートモデルタイミングダイアグラム

CUPL による簡単なステートマシンモデルの特徴は、図 8-30 のタイミングダイアグラムで示されます。レジスタード出力のセットとリセットは、ステートビットの状態により行われます。同様に、ノンレジスタード出力は、現在のステートビットの状態か、ステートビットの状態と入力の状態の組み合わせにより行われます。

組み合わせ論理は、出力信号を生成する論理ゲート(通常 AND-OR)の任意の組み合わせです。出力信号の変化は、入力信号の変化が起きてから  $T_{pd}$  nsec(伝播遅れ時間)に起こります。 $T_{pd}$  は入力またはフィードバックが入ってからノンレジスタード出力が出力されるまでの時間です。

ステートビットは、現在の状態をフィードバックし組み合わせ論理を駆動するフリップフロップです。

ストレージレジスタは、ステートマシン組み合わせ論理から入力を受け取る任意のフリップフロップです。レジスタの中には、ステートビットとして使用されるものもあります。また、その他のレジスタはレジスタード出力として使用されます。レジスタード出力は、クロックパルス後  $T_{co}$  nsec で出力されます。 $T_{co}$  は、クロック信号の始めからフリップフロップ出力が出力されるまでの間の時間です。

PLD に必要なセットアップとホールド時間は、正しく動作させるためにシステムに合っている必要があります。ほとんどの PLD では、組み合わせ論理の伝播遅れとフリップフロップの実際のセットアップ時間を含むセットアップ時間( $T_{su}$ )が必要です。 $T_{su}$  はフィードバックにかかる時間か入力イベントがフリップフロップの入力に現れるまでにかかる時間のどちらかです。この結果がフリップフロップの入力で有効になるまで、その後のクロック入力は受け付けられません。D タイプや RS タイプ、JK タイプのフリップフロップが使用できます。RS と JK のフリップフロップでは D タイプよりもプロダクト(P)タームの数を少なくできるので、RS と JK はステートマシンのインプリメンテーションでよく使われます。

組み合わせ論理回路はノンレジスタード出力を生成します。ノンレジスタード出力はステートビットや入力信号の(非同期タイミングの)関数です。従って、それらはアクティブなクロックエッジが入力されてから  $T_{co}$  と  $T_{pd}$  nsec 後に起こる現在のステートビットの値に依存します。

ストレージレジスタはレジスタード出力を生成します。ただし、実際のステートビットには含まれません(すなわち、ビットフィールドはすべてステートビットで構成されます。)。ステートマシンの理論では、これらレジスタード出力のセットとリセットは、現在の状態から次の状態への遷移で起こることが示されています。これは、マシンがどのようにその状態になるかということです。従って、レジスタード出力は、DONT CARE モードの動作をサポートします。DONT CARE モードでのレジスタード出力は、現在の状態遷移がそのレジスタード出力を指定しない限り、最後の値を保持し続けます。

## ステートマシンシンタクス

CUPL の簡単なステートマシンシンタクスにより、ステートマシンモデルを簡単にインプリメントできます。CUPL シンタクスは簡単なフォーマットです。このフォーマットによりステートマシンの任意の関数を記述できます。ステートマシンシンタクスの通例のフォーマットを以下に示します。

```
SEQUENCE state_bit_field(  
  
    PRESENT present_state  
        IF input_cond NEXT next_state OUT outputs;  
        IF input_cond NEXT next_state OUT outputs;  
        IF ...  
    PRESENT present_state  
        IF input_cond NEXT next_state OUT outputs;  
        IF input_cond NEXT next_state OUT outputs;  
        IF ...  
    PRESENT ...  
}
```

図 8-31 ステートマシンシンタクスとフォーマット

それぞれの現在の状態のブロックには、非同期(現在の状態)と同期(トランジション)動作の両方を記述します。このフォーマットでステートマシンの

すべてのコンポーネントを記述できます。

同期出力のフォーマットを以下に示します。

IF input_cond	NEXT next_state	OUT outputs
¥_____ /	¥_____ /	¥_____ /
CONDITIONAL	TRANSITION	OUTPUT ASSOCIATED
	WITH TRANSITION	
	NEXT next_state	OUT outputs
¥_____ /	¥_____ /	¥_____ /
UNCONDITIONAL		OUTPUT ASSOCIATED
TRANSITION	WITH TRANSITION	

図 8-32 ステートマシン同期出力

同期出力は、状態遷移が条件付きで起こるかそうでないかによります。CUPL のキーワードをどのように使用してこれらの式を利用しシステムを記述するかを習得する必要があります。例えば、NEXT 命令によりコンパイルに、そのブロック内のすべての要素が状態遷移情報(すなわち、現在の状態から次の状態への遷移)によって遷移する出力のレジスタード出力であることが示されます。If 命令は条件イベントを意味します。If キーワードがノンレジスタード記述の中で使用されると、入力と出力は非同期的に行われることを示します。NEXT キーワードがない場合、ノンレジスタードイベントを示します。ノンレジスタード非同期出力では、以下のフォーマットを使用します。

IF input_cond		OUT outputs
¥_____ /	¥_____ /	¥_____ /
INPUT CONDITION	NO STATE	OUTPUTS ASSOCIATED
AFFECTS OUTPUT	TRANSITION	WITH INPUT CONDITION
AFTER Tpd	AND PRESENT STATE	
	OUT outputs	
¥_____ /	¥_____ /	¥_____ /
NO INPUT	NO STATE	OUTPUT ASSOCIATED
CONDITION	TRANSITION	SOLELY WITH PRESENT
	STATE. VALID Tco + Tpd	
	AFTER CLOCK	

図 8-33 ステートマシン非同期出力

レジスタードフォーマットとノンレジスタードフォーマットのどちらかを選択する理由は、システムのタイミングにあります。完全な同期システムでは、タイミングを確実に合わせる必要があります。レジスタード出力を使用すると反応を早くすることができます。その反応はクロックパルス後の Tco nsec 以内です。このように早い反応により、回路時間はレジスタード出力を次のクロックパルスの前に回路内のどこかの入力として使用できます。反対に、ノンレジスタード出力は非同期アプリケーションで使用して下さい。ノンレジスタード出力は、プレゼントステートデコーダなどの極簡単なアプリケーションで使用します。

2 ビットカウンタの例

ステートマシンモデルやそのシンタクスをよく理解するために以下の簡単な例について考えてみます。この例は、一つの入力と一つのレジスタード出力、一つのノンレジスタード出力を持つ 2 ビットのフリーランニングカウンタです。

図 8-35 にこの例のダイアグラムを示します。カウンタには一つの入力と一つのノンレジスタード出力、一つのレジスタード出力があります。

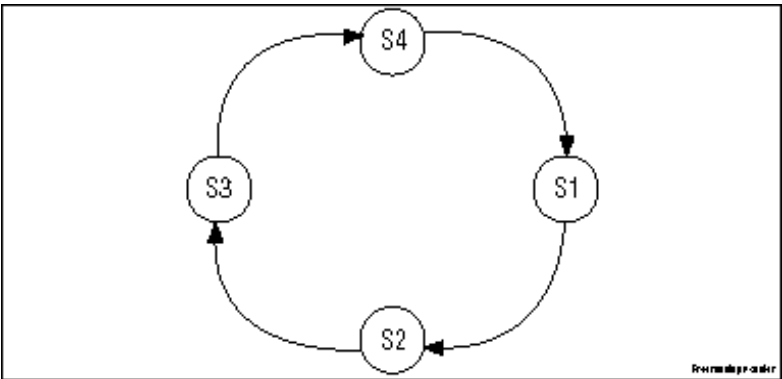


図 8-35 フリーランニング 2 ビットカウンタ

丸は状態(ステートビットの特定の組み合わせ)を表わします。矢印は状態と状態との間の遷移を表わします。この例の遷移には条件が無いので、カウンタはフリーランニングになります。従って、この論理記述では、次の状態を表わす命令に If キーワードが使用されていません。ノンレジスタード出力は入力アクティブな場合、2 つのカウント(S2)でアクティブになります。レジスタード出力は S3 から S0 への遷移でセットされ、S3 から S0 への遷移でリセットされます。リスト 2 は CUPL のソースファイルとカウンタの論理記述を示します。

ファイルの最初の部分は、履歴情報やデバイスの関数についての説明です。互換性のある PLD の一覧も記述されます。ピン宣言は設計ダイアグラムの入力と出力に対応して行われます。論理式はカウンタをインプリメントするために記述されます。q0 の式は q0 が呼び出される時に定義されます。!reset のタームは、同期リセットが必要な回路を初期化するために q0 と q1 の両方に使用されます。電源が投入された時、フリップフロップは、タイミングダイアグラムで DONT CARE スラッシュで示されるようなどちらかの状態になっています。リセット信号が最初に呼び出されることに注意して下さい。各変数の式と!reset 信号を AND することでは、パワーオンの条件にはなりません。すなわち、フリップフロップはセットされません。パワーオンサイクル後にリセットが LOW(FALSE)に戻るとと完了します。この状態は、回路のフリップフロップの値には影響を与えません。

リスト 2 に 2 ビットのカウンタを生成する CUPL プログラムを示します。

Name	FLOPS;
Partno	CA0002;

```

Revision 02;
Date 9/29/89;
Designer Osann;
Company PROTEL INTERNATIONAL;
Location NONE;
Assembly NONE;
/*****
**/
/* This example demonstrates the use of D-type */
/* flip-flops to implement a two bit counter */
/* using the following timing diagram */
/*
/*
/*
/* clock |___| |___| |___| |___| |___| */
/*
/*
/* q0  /////|_____| |_____| |___ */
/*
/*
/*
/* q1  /////|_____| |_____| |___ */
/*
/*
/*
/* reset |_____ */
/*
/*
/*****
**/
/* TARGET DEVICES: PAL16R8, PAL16RP8, GAL16V8 */
/*****
**/

Pin 1 = clock; Pin 2 = reset;

/* Outputs: define outputs and output active levels */

Pin 17 = q0; Pin 16 = q1;
FIELD COUNT = [Q1, Q0]; /* THE STATE FIELD BIT IS "COUNT" */

/*****
*/
$DEFINE S0 0/*DEFINE SYMBOLIC NAMES FOR THE ACTUAL STATE BIT */
$DEFINE S1 1/*CONSTANT VALUES USING PREPROCESSOR COMMANDS */
$DEFINE S2 2/*CONSTANTS DEFAULT TO HEX AND REPRESENT VALUES */
$DEFINE S3 3/*OF "COUNT" WITHIN THE SEQUENCE "BLOCK" BELOW */
/*****
**/

SEQUENCE COUNT { /* NOTE USE OF BRACES FOR ENCLOSING */
/* STATE SEQUENCE DESCRIPTION BLOCK */

```



```

PRESENT S0
  NEXT S1 PRESENT S1
  NEXT S2 PRESENT S2
IF INPUT OUT NON_REG_OUT; /* ASYNCHRONOUS WITHIN */
  NEXT S3 OUT REG_OUT /* S2 SETS ON TRANSITION */
PRESENT S3
  NEXT S0 OUT !REG_OUT ; } /* RESETS ON TRANSITION */
}

```

リスト22 ビットカウンタの論理記述

## 回路に必要なヒステリシスの例

ヒステリシスの必要なアプリケーションでは、現在の状態の情報を用いてトランジション情報を使用することが重要です。例えば、アナログ信号のスレッシュホールドを検知する回路には、アナログコンパレータのヒステリシスバンドよりも精度がよくまたそれよりも幅の広いヒステリシスが必要です。トランジション情報は、このようなアプリケーションにヒステリシスを持たせるために必要です。一つの解決法は、スレッシュホールドディテクタ出力(デジタルシュミットトリガ出力)がステートマシンのレジスタード出力であるようなトラッキング A/D コンバータを構築することです。(図 8-36)

図 8-36 に、上記の説明に基づくスレッシュホールドディテクタを示します。レジスタード出力は現在のステート情報の代わりにトランジション情報によりセットあるいはリセットされます。これによりヒステリシスを持つ関数が与えられます。

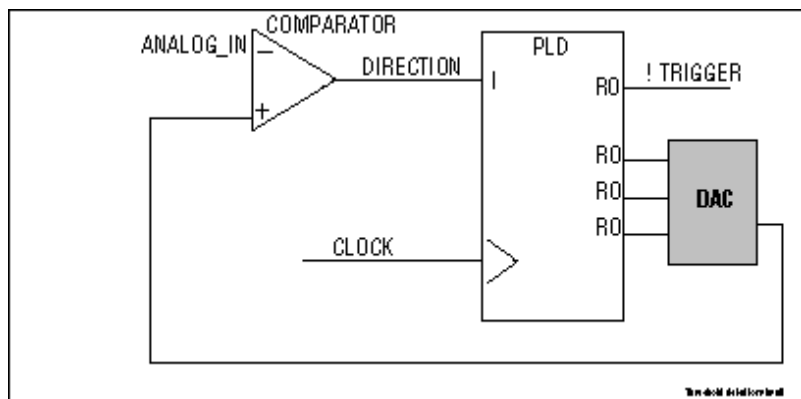


図 8-36 スレッシュホールドディテクタ回路

D/A コンバータに入力される 3 つのカウントビット(Q1,Q2,Q3)はステートビットです。トリガ出力を S5 から S6 へのトランジションでセットする場合と、S2 から S1 へのトランジションでトリガをリセットする場合、ヒステリシスが持たされます。その他の場合、トリガ出力は DONT CARE 状態にセットされます。また、トリガ出力は、マシンの状態に応じて S2 から S5 のステートに違う値を持ちます。

このアプリケーションのステートビットにより外の世界へ情報が出力されることに注意して下さい。その情報は、この場合は D/A コンバータへの入

力になります。外の世界へ PLD のアクセスがあると、標準 Mealy や Moore ステートメントマシンモデルから離れることができます。これにより PLD により多くのロジックを詰め込むことができます。

図 8-37 にヒステリシス付きのアナログコンパレータのステートダイアグラムを示します。図示したように、ステート値はヒステリディペンデントです。トリガ出力はマシンの状態に応じてステート S2 から S5 で異なる値を持ちます。トリガをリセットするステートを一番下に示します。S2 と S3,S4,S5 のステートは、2 つの値を持つことができるので 2 回現れていることに注意して下さい。各ステート値は、システムの以前のステートにより決まります。

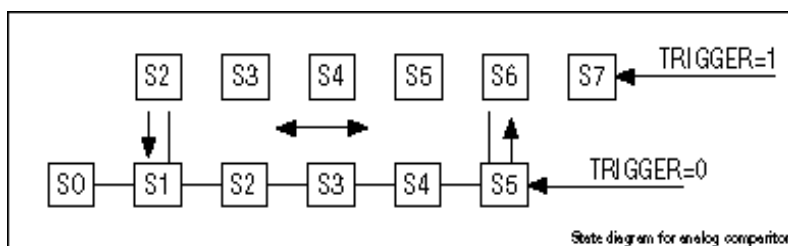


図 8-37 ヒステリシス付きのアナログコンパレータのステートダイアグラム

リスト 3 は、ステートマシンの論理記述ファイル(LDF)です。LDF では、ステートビットはビットフィールドとして宣言され Count という名前を与えられます。次に入力の方向は、Up と Down カウンタモードの名前により定義されます。それから、数値ステートの割り付けは CUPPL のプリプロセッサから \$DEFINE コマンドを使用して S0 から S7 で行われます。

```
Partno      CT0001;
Name        SCHMITT;
Date        9/29/89;
Revision    02;
Designer    MOT;
Company     PROTEL INTERNATIONAL;
Assembly    ANALOG_INTERFACE;
Location    U27;

/*****
/* THIS DEVICE RECEIVES A 'COUNT DIRECTION COMMAND' FROM AN */
/* ANALOG COMPARATOR AND RESPONDS BY INCREMENTING OR          */
/* DECREMENTING AN INTEGRAL UP/DOWN COUNTER. A REGISTERED     */
/* OUTPUT IS CREATED AND ACTS AS A DIGITAL SCHMITT TRIGGER    */
/* WITH HYSTERESIS.                                           */
*****/

/** INPUTS **/

PIN 1= CLOCK;          /* CLOCK PIN FOR THE COUNTER */
PIN 2= DIRECTION;      /* COUNT MODE DIRECTION PIN */
```

```

/** OUTPUTS **/
PIN [14..16]= ![Q0..2];/* COUNTER STATE BITS */
PIN 17= !TRIGGER; /* SCHMITT TRIGGER OUTPUT BIT */

/** DECLARATIONS AND INTERMEDIATE VARIABLE DEFINITIONS **/
UP = DIRECTION; /* COUNTER MODES */
DOWN = !DIRECTION;

FIELD COUNT = [Q2..0]; /* FIELD FOR COUNTER STATES */

$DEFINE S0 0 /* COUNTER STATES DEFINED AS */
$DEFINE S1 1 /* STATES 0 THROUGH 7 */
$DEFINE S2 2
$DEFINE S3 3
$DEFINE S4 4
$DEFINE S5 5
$DEFINE S6 6
$DEFINE S7 7

SEQUENCE COUNT {
  PRESENT S0
    IF UP      NEXT S1;
    IF DOWN    NEXT S0;
  PRESENT S1
    IF UP      NEXT S2;
    IF DOWN    NEXT S0;
  PRESENT S2
    IF UP      NEXT S3;
    IF DOWN    NEXT S1      OUT !TRIGGER;
  PRESENT S3
    IF UP      NEXT S4;
    IF DOWN    NEXT S2;
  PRESENT S4
    IF UP      NEXT S5;
    IF DOWN    NEXT S3;
  PRESENT S5
    IF UP      NEXT S6;      OUT TRIGGER
    IF DOWN    NEXT S4;
  PRESENT S6
    IF UP      NEXT S7;
    IF DOWN    NEXT S5;
  PRESENT S7
    IF UP      NEXT S7;
    IF DOWN    NEXT S6;
}

```

### リスト3 シュミットトリガ論理記述ファイル

ステートマシンの定義には、プレゼントステートのブロックに IF キーワードや NEXT キーワードを使用します。NEXT キーワードの使用は、ステータ

トマシンが同期回路であることを表わします。IF キーワードの使用は、遷移が条件付きで行われることを示します。カウンタがカウントする方向により遷移の方向が変わります。言い換えると、Direction 入力の値により方向が決まります。

ヒステリシスのあるカウンタやコンパレータのようなアプリケーションがあまり複雑になりそうにない場合でも、PLD を使用する設計業務が最も効率のよい設計であることが解ります。この場合、既存の設計をデバイスに置き換える場合でもステートマシン設計を行なう場合でも同じです。

## デュアルクアドラチャエンコーダ/デコーダ

この例では、同期ステートマシンをどのように非同期設計に使用するかを示します。設計の詳細は PLD リストファイルを参照して下さい。この設計のシミュレーションを行なうためのテストベクタも用意されています。

```

Name      Encoder;
Partno    EP1210;
Revision  0;
Date      Sept. 1989;
Designer  Jim Millener;
Company   Protel International;
Assembly  Encoder Counter;
Location  U1;
Device    P16R4;
Format    j;
/*****
***/
/*これは、CUPL PLD コンパイラを使用するデュアルクアドラチャエンコーダデコーダの同期ステートマシンの定義インプリメントの方法の例の第段階です。デュアルクアドラチャエンコーダは2つのデジタル出力を持つデバイスで、両方の動作の向きと方を計測するために使われます。動作がある場合 デジタル出力は同じ周波数が出されます。ただし、互いに位相は90度ずれています。渡の向きはクアドラチャのエッジをカウントすることで得られます。方、どの出力位相にあるかで解ります。エンコーダの出力を Phase A と Phase B と名を付けます。以下のダイアグラムによりエンコーダの機能を説明します。*/
/*
/*
/*
/* Phase A  _____| |_____| |_____| |_____ */
/*
/*
/* Phase B  |_____| |_____| |_____| |_____| */
/*
/*
/* Count/Phase | 0| 1| 2| 3| 0| 1| 2| 1| 0| 3| 2| 1| 0| */
/*
/*
/*
/* Forward      Backward      */
/*
/*
/* すべてのカウント値には、Phase A と Phase B の安定したステートが存在することに注意して下さい。これらのステートは次のように定義されます。*/
/*

```

```

/* State 0 -> Phase A = 0 and Phase B = 0.          */
/* State 1 -> Phase A = 1 and Phase B = 0.          */
/* State 2 -> Phase A = 1 and Phase B = 1.          */
/* State 3 -> Phase A = 0 and Phase B = 1.          */
/*
*/
/*ステートマシンを定義するために、少なくとも2個のフリップフロップからなるシ
ーケンスに4つのステートがあります。以下のCUPLソースコードにより、この例で使
される PAL16R4 のピンとエンコードの Phase のステートマシンが定義されます。
*/
/*
*/
/*****
*/
/* Input Pin Assignments */
pin 1 = Clock;
pin 2 = PhaseAin;
pin 3 = PhaseBin;
pin 11 = !OutPutEnable;

/* Output Pin Assignments */
pin 12 = ForwardCount;
pin 13 = ReverseCount;
pin 14 = StateVar0;
pin 15 = StateVar1;
pin 16 = Count;
pin 17 = Direction;

/* Input State Definition */
$Define InputState0 !PhaseBin&!PhaseAin
$Define InputState1 !PhaseBin&PhaseAin
$Define InputState2 PhaseBin&PhaseAin
$Define InputState3 PhaseBin&!PhaseAin

/* Stable State Definitions */

$Define StableState0 'b'00
$Define StableState1 'b'01
$Define StableState2 'b'11
$Define StableState3 'b'10

$Define State0 !StateVar1&!StateVar0
$Define State1 !StateVar1&StateVar0
$Define State2 StateVar1&StateVar0
$Define State3 StateVar1&!StateVar0

/* State Variable Definitions */

Field EncoderState = [StateVar1,StateVar0];

```

```

/* State Equations */
Sequence EncoderState {
  Present StableState0
  if InputState1 next StableState1; /* Forward motion detected */
  if InputState3 next StableState3; /* Reverse motion detected */
  default next StableState0;
  present StableState1
  if InputState2 next StableState2; /* Forward motion detected */
  if InputState0 next StableState0; /* Reverse motion detected */
  default next StableState1;
  present StableState2
  if InputState3 next StableState3; /* Forward motion detected */
  if InputState1 next StableState1; /* Reverse motion detected */
  default next StableState2;
  present StableState3
  if InputState0 next StableState0; /* Forward motion detected */
  if InputState2 next StableState2; /* Reverse motion detected */
  default next StableState3;
}
/*****
***/
/*ステートが定義されると出が生じます。まず、CountUpとCountDownという2
つの中間変数が定義されます。これらの変数が設定されると、ステートマシンサイクルが次の
安定状態になる前に次の状態がデコードされます。そして、動作の方向に応じてステートマ
シンの以下のシーケンスでステートを変えます。 */
/*
*/
/* State0 -> State1 -> State2 -> State3 -> State0 -> ... */
/* Forward
*/
/*
*/
/* State0 -> State3 -> State2 -> State1 -> State0 -> ... */
/* Reverse
/
/*
*/
/* The input transitions that will cause a change in state of */
/* the state-machine for the two directions are listed below: */
/*
*/
/* Input Transition State-Machine Present State */
/*
*/
/* Forward
*/
/* InputState0 -> InputState1 State0 -> State1 */
/* InputState1 -> InputState2 State1 -> State2 */
/* InputState2 -> InputState3 State2 -> State3 */
/* InputState3 -> InputState0 State3 -> State0 */
/*
*/
/* Reverse
*/
/* InputState0 -> InputState3 State0 -> State3 */
/* InputState1 -> InputState0 State1 -> State0 */
/* InputState2 -> InputState1 State2 -> State1 */
/* InputState3 -> InputState2 State3 -> State2 */
/*
*/

```

```

/* The intermediate variables "CountUp" and "CountDown" will */
/* thus be defined as the time from the change in input state */
/* until the time that the state-machine changes states. */
/*****
***/
Append ForwardCount = InputState1 & State0;
Append ForwardCount = InputState2 & State1;
Append ForwardCount = InputState3 & State2;
Append ForwardCount = InputState0 & State3;
Append ReverseCount = InputState3 & State0;
Append ReverseCount = InputState0 & State1;
Append ReverseCount = InputState1 & State2;
Append ReverseCount = InputState2 & State3;
/*****
***/
/* 次の段は出力 Count と Direction を定義することです。Count.D により、
ForwardCount または ReverseCount のどちらかに定義されます。Count は 1 クロ
ックサイクルだけ存在できるので、!Count のタームを Count.D に連し、それがセ
ットされた後 クロックサイクルだけ出力をリセットします。Count と Direction 出
は、カウントアップとカウントダウンを操作するために使用されます。従って、
Direction 出力は Count サイクルの前でセットされるので、Count 出力の極は、
Count 出力の後でカウンタがクロックされるようになってることを確認して下さい。こ
れにより、アップカウンタ / ダウンカウンタの競合を避けることができます。
Direction 出力は Forward=high に定義されるので、Direction フリップフロップ
への D 入力には ForwardCount タームが必要です。フリップフロップは連続にクロック
されるので、前のステートはセットされると次のクロックサイクルでリセットさ
れることに注意して下さい。ラッチステートはラッチすることが必要です。これにより、
Direction&!ReverseCount タームを Direction フリップフロップの D 入力に加
ることによる影響を最小にできます。このステートメントは、一度 ForwardCount が
見つかり、フリップフロップへの入力は出力と同じになります。すなわち、状態がラッ
チされます。そしてこの状態をリセットする ReverseCount パルスが来るまでこの状態
が続きます。 */
/*
*/
/*****
***/

Count.D = !Count & (ForwardCount # ReverseCount);
Direction.D = ForwardCount # (Direction & !ReverseCount);
Listing 4 Dual Quadrature Encoder/Decoder
リスト 5 に、エンコーダ / デコーダのシミュレーション ファイルを示します。
Name Encoder;
Partno EP1210;
Revision 0;
Date Sept. 1989;
Designer Jim Millener;
Company Protel International;
Assembly Encoder Counter;
Location U1;
Device P16R4;

```

```

Format      j;

Order:Clock, PhaseAin, PhaseBin, StateVar0, StateVar1,
ForwardCount, ReverseCount, Count, Direction, OutPutEnable;
Vectors:
PXX00XXXX1
$Repeat 4;
C00*****1
$Repeat 4;
C10*****1
$Repeat 4;
C11*****1
$Repeat 4;
C01*****1
$Repeat 4;
C11*****1
$Repeat 4;
C10*****1
$Repeat 4;
C00*****1

```

リスト5 エンコーダ/デコーダのシミュレーション入力ファイル

## カウンタの設計

このセクションでは、ブール式を用いた 4 ビットカウンタの設計について説明します。一般に、カウンタの設計は CUPL ステートマシンシンタクスを使用するほうが簡単ですが、ここではそちらの方法には触れません。すべての例に D - レジスタを使用します。使用するデバイスは P20X8 です。このデバイスには、このタイプのカウンタを構築するために都合の良い XOR ゲートがあります。

ここでの基本的な前提条件は、カウンタがカウントアップする場合、最下位ビット以下のビットはすべて HIGH にトグルされます。逆にカウンタがカウントダウンする場合、最下位ビット以下のビットはすべて LOW にトグルされます。XOR ゲートの入力の一つに最下位ビット以下のすべてのビットの AND を入力しもう一つの入力にレジスタ自身の以前の値を入力して上記の動作を実現します。最下位ビットの場合、これ以下のビットは無いので代わりに 2 進数の 1 が使用されます。これは、カウンタでは最下位ビットが常にトグルされているということです。カウンタの一つのステップがどのように動作するかを示します。

以下の表は、XOR ゲートの真理値表です。ゲートの出力は 2 つの入力が違う時だけハイになることに注意して下さい。入力が同じ場合、出力はローです。このような動作が、私達がこれから作成しようとするカウンタでは重要になります。

IN1	IN2	OUT
0	0	0



0	1	1
1	0	1
1	1	0

図 8-38 XOR ゲートの真理値表

## 1 ビットアップカウンタ

1 ビットカウンタを設計するのは簡単です。クロック周期毎にレジスタを単にトグルするだけです。従って、式は以下のようになります。

```
Q0.d = !Q0;
```

各クロック周期で Q0 は、前の状態の反転になるのでこれにより常にトグルされます。P20X8 の XOR ゲートを使用して上記と同じ機能を実現することもできます。式を以下に示します。

```
Q0.d = 'b'1 $ Q0;
```

XOR ゲートは入力 A で常に 2 進数の 1 を持つので、XOR ゲートの出力は、XOR ゲートへの入力 B の反対になります。これにより Q0 は以前の Q0 の反対になります。

## 2 ビットカウンタ

ビットを追加して、2 ビットのカウンタを作ります。2 ビットカウンタでは、最上位ビットは、クロックパルスが 2 つ毎にトグルします。最下位ビットが 1 の時にクロックパルスが入力されると、最上位ビットがトグルします。従って、Q1 は Q1 の前の値と Q0 の前の値に基づいて決められます。

```
Q0.d = 'b'1 $ Q0;
Q1.d = Q0 $ Q1;
```

## 3 ビットカウンタ

他の式を追加すれば、簡単にビットを追加できます。特定のビットより下位のビットがすべてハイになると、そのビットはトグルします。Q0 と Q1 がハイでクロックパルスが入力されると Q2 がトグルします。この機能を追加するには、Q2 の式を追加して下さい。

```
Q0.d = 'b'1 $ Q0;
Q1.d = Q0 $ Q1;
Q2.d = (Q1 & Q0) $ Q2;
```

## 4 ビットカウンタ

以下のような Q3 の式を追加すると 4 ビットカウンタができます。

```
Q0.d = 'b'1 $ Q0;
Q1.d = Q0 $ Q1;
Q2.d = (Q1 & Q0) $ Q2;
Q3.d = (Q2 & Q1 & Q0) $ Q3;
```

特定のビットより下位のビットをすべて AND して、そのビットの以前のビ

ットの値と XOR をすると任意の大きさのカウンタの式を生成できることが分かります。

### アップ/ダウンを変更できる4ビットカウンタ

次に、カウンタのアップ/ダウンを変更できる機能を追加します。アップ/ダウンをコントロールする機能を追加するには、AND ゲートを追加する必要があります。このゲートにより、アップコントロール信号をチェックします。信号が TRUE の場合、カウントアップされ、その他の場合、カウントダウンされます。

アップモードの場合、特定のビットより下位のビットがすべて 1 になるとそのビットがトグルし、ダウンモードの場合特定のビットより下位のビットがすべて 0 になるとそのビットがトグルします。最下位ビットはアップカウントやダウンカウントに関らず常にトグルします。

特定のビットより下位のビットをすべて AND しそのビットの以前の値と XOR する操作はここでも行ないます。ただし、これはカウントアップの時点で、カウントダウンの場合、下位ビットと反対の値を AND します。これは、カウントアップの場合、下位ビットがすべてハイの時にそのビットをトグルするためです。カウントダウンの場合、下位ビットがすべてローの時にそのビットをトグルします。

```
Q0.d = 'b'1 $ Q0;
Q1.d = (Q0 & UP # !Q0 & !UP) $ Q1;
Q2.d = (Q1 & Q0 & UP # !Q1 & !Q0 & !UP) $ Q2;
Q3.d = (Q2 & Q1 & Q0 & UP # !Q2 & !Q1 & !Q0 & !UP) $ Q3;
```

### アップダウンとロードのできる4ビットカウンタ

カウンタにロード機能を追加するには、ロード信号をカウンタに取り込む事と、式をわずかに変更する事が必要です。ロードの表現式とカウンタの表現式を分離する必要があります。カウンタの表現式とロード信号の反転とを AND して、カウントがロードが行われていない間だけ実行されるようにします。それから入力データ信号の組みを追加します。これらは、D0、D1、D2、D3 と表わされます。

```
Q0.d =      !LD & 'b'1
          $ !LD & Q0
          # LD & D0;
Q1.d = (    !LD & Q0 & UP
          # !LD & !Q0 & !UP)
          $ !LD & Q1
          # LD & D1;
Q2.d = (    !LD & Q1 & Q0 & UP
          # !LD & !Q1 & !Q0 & !UP)
          $ !LD & Q2
          # LD & D2;
Q3.d = (    !LD & Q2 & Q1 & Q0 & UP
          # !LD & !Q2 & !Q1 & !Q0 & !UP)
```

```
$ !LD & Q3
# LD & D3;
```

LD が偽の場合、ロード式を除くすべての式は偽と評価されます。

## ホールド機能の追加

ホールド機能をカウンタに追加するには、レジスタがそれ自身にフィードバックするように式を設定する必要があります。現在の設定は、XOR ゲートを介してレジスタがそれ自身にフィードバックするようになっています。今のところ、この XOR ゲートは、XOR ゲートの片方の入力に真が入力されるとレジスタがトグルされるために使用されています。カウンタが値を保持できるようにしたい場合、XOR ゲートへの一つの入力、レジスタがその値を保持するようにローに評価される必要があります。

```
Q0.d = !LD & !hold
      $ !LD & Q0
      # LD & D0;
Q1.d = ( !LD & UP & !hold & Q0
      # !LD & !UP & !hold & !Q0)
      $ !LD & Q1
      # LD & D1;
Q2.d = ( !LD & UP & !hold & Q1 & Q0
      # !LD & !UP & !hold & !Q1 & !Q0)
      $ !LD & Q2
      # LD & D2;
Q3.d = ( !LD & UP & !hold & Q2 & Q1 & Q0
      # !LD & !UP & !hold & !Q2 & !Q1 & !Q0)
      $ !LD & Q3
      # LD & D3;
```

ホールドが真の場合、レジスタはそれ自身に直接フィードバックし、その値が保持されます。

```
Name      cnt4;
Partno    XXXXX ;
Date      08/28/91 ;
Revision  1 ;
Designer  Teixeira ;
Company   PROTEL INTERNATIONAL ;
Assembly  XXXXX ;
Location  XXXXX ;
Device    P20X8;

/*****
/* 4-bit synchronous counter with parallel load and hold */
/*****
/* Allowable Target Device Types: P20X8 */
/*****
```

```

/** Inputs **/
PIN 1 = CLK ;.d = 4; /* minimize equations using Espresso */
Hold = !CBI; /* counter hold when CBI is FALSE */

Q0.d = !LD & !hold
      $ !LD & Q0
      # LD & D0;

Q1.d = ( !LD & UP & !hold & Q0
        # !LD & !UP & !hold & !Q0) $ !LD & Q1 # LD & D1;

Q2.d = ( !LD & UP & !hold & Q1 & Q0
        # !LD & !UP & !hold & !Q1 & !Q0)
        $ !LD & Q2 # LD & D2;

Q3.d = ( !LD & UP & !hold & Q2 & Q1 & Q0}
        # !LD & !UP & !hold & !Q2 & !Q1 & !Q0}
        $ !LD & Q3 # LD & D3;

CBO = UP & [Q3..0]:& /* Carry-out */
      # !UP & ![Q3..0]:&; /* Borrow-out */

```

ホールド機能とロード機能のある4ビットアップ/ダウンカウンタ

このカウンタを8ビットカウンタに拡張するのは簡単です。

```

Name      cnt8;
Partno    XXXXX ;
Date      08/28/91 ;
Revision  1 ;
Designer  Teixeira ;
Company   PROTEL INTERNATIONAL ;
Assembly  XXXXX ;
Location  XXXXX ;
Device    P20X8;

/*****
***/
/* 8-bit synchronous counter with parallel load and hold */
/* This is a 74ls469 emulator */
/*****
***/
/* Allowable Target Device Types: P20X8 */
/*****
***/

/** Inputs **/
PIN 1 = CLK ; /* Clock */
PIN 2 = !LD ; /* Load pin */
PIN [3..10] = [D0..7] ; /* Data in */

```

```

PIN 11 = !UP ;      /* Up/Down signal */
PIN 13 = !OE ;      /* Output Enable */
PIN 23 = !CBI ;     /* Carry/Borrow in */

/** Outputs **/
PIN [15..22] = [Q7..0] ; /* Outputs */
PIN 14 = !CBO ;     /* Carry/Borrow out */
Min [Q7..0].d = 4;   /* minimize equations using Espresso */
Hold = !CBI;        /* counter hold when CBI is FALSE */

Q0.d = !LD & !hold
$ !LD & Q0
# LD & D0;

Q1.d = ( !LD & UP & !hold & Q0
# !LD & !UP & !hold & !Q0) $ !LD & Q1
# LD & D1;

Q2.d = ( !LD & UP & !hold & Q1 & Q0
# !LD & !UP & !hold & !Q1 & !Q0)
$ !LD & Q2
# LD & D2;

Q3.d = ( !LD & UP & !hold & Q2 & Q1 & Q0
# !LD & !UP & !hold & !Q2 & !Q1 & !Q0)
$ !LD & Q3
# LD & D3;

Q4.d = ( !LD & UP & !hold & Q3 & Q2 & Q1 & Q0
# !LD & !UP & !hold & !Q3 & !Q2 & !Q1 & !Q0)
$ !LD & Q4
# LD & D4;

Q5.d = ( !LD & UP & !hold & Q4 & Q3 & Q2 & Q1 & Q0
# !LD & !UP & !hold & !Q4 & !Q3 & !Q2 & !Q1 & !Q0)
$ !LD & Q5
# LD & D5;

Q6.d = ( !LD & UP & !hold & Q5 & Q4 & Q3 & Q2 & Q1 & Q0
# !LD & !UP & !hold & !Q5 & !Q4 & !Q3 & !Q2 & !Q1 & !Q0)
$ !LD & Q6
# LD & D6;

Q7.d = ( !LD & UP & !hold & Q6 & Q5 & Q4 & Q3 & Q2 & Q1 & Q0
# !LD & !UP & !hold & !Q6 & !Q5 & !Q4 & !Q3 & !Q2 & !Q1
& !Q0)
$ !LD & Q7
# LD & D7;

```

```
CBO = UP & [Q7..0]:& /* Carry-out */  
# !UP & ![Q7..0]:&; /* Borrow-out */
```

ホールド機能とロード機能のある8ビットアップ/ダウンカウンタ

## カウントダウンの実行

この設計は、15 から始まり 0 までカウントダウンするカウンタで、自分自身を中断するために自分自身により信号が送られます。このステートマシンには 0 から 15 までの番号の付けられた 16 個のステートがあります。各ステートは 2 進数値を表わしています。

システムの電源が投入されると、最初のクロックでステートマシンはステート 15 に行きます。この状態は START ボタンが押されるまでそのままです。ステートマシンが、START 信号を検出するとカウントダウンを始めます。カウント中は、START ボタンを保持する必要があります。システムが 0 に到達する前にボタンが放されるとステートマシンはステート 15 に戻り START 信号を待ちます。

ステートマシンがステート 0 に到達すると、OFF と呼ばれる組み合わせ信号がハイに駆動されます。この信号により外部からシステムをシャットダウンできます。START 信号が真でステートマシンが 0 ステートの間、OFF 信号が出されます。この場合、システムをシャットダウンする前に、START をローにすると、ステートマシンはステート 15 にリセットされ、START の待ち状態になります。

\$REPEAT を使用してステートの \$DEFINE を作成していることに注意して下さい。このような方法をしなくてはならないわけではありませんが、PLD ファイルが読みやすくなります。

PLD のデバイス指定は G16V8MS です。これは G16V8 のレジスタモードです。これは以前に中間の同期モードとして知られていました。デバイスを G16V8 に指定すると、コンパイラはピンの使用状況を解析してどのモードが選択されたか判断します。この設計では、モードは特別に選択され、自動のモード選択はキャンセルされています。特別な設定を行なう場合このようなことが必要になります。

MIN ステートを使用する場合注意が必要です。これにより、コンパイラが呼び出され、これらの出力ピンに Espresso 最小化法を施します。MIN ステートを使用しない場合、この設計をインプリメントするために必要なプロダクトタームの数は、G16V8 の容量を越えてしまいます。ピン毎に最小化を適用できる機能はこのコンパイラの特徴の一つです。必要なピンにだけ最小化を行なうことができるので処理速度が速くなります。

```
Name      execnt;  
Partno    execnt;  
Date      09/19/91;  
Revision  00;  
Designer  Teixeira;  
Company   PROTEL INTERNATIONAL;
```

```

Location none;
Assembly none;
Device GL6V8MS;
/*****
/* Implement a counter the counts from 15 to 0 */
/* then shuts itself off */
*****/
/** inputs **/
Pin 1 = clk;
Pin 2 = START;
Pin 11 = !Enable; /* output enable is active low */
/** outputs **/
Pin [16..19] = [Q3..0];
Pin 15 = OFF;
$REPEAT i = [0..15]
$DEFINE S{i} 'h'{i}
$REPEND
field count = [Q3..0];
MIN count.d = 4;

sequence count {
    Present S0 if !START Next S15; if START Next S0; if START out
    OFF;
    Present S1 if !START Next S15; Default Next S0;
    Present S2 if !START Next S15; Default Next S1;
    Present S3 if !START Next S15; Default Next S2;
    Present S4 if !START Next S15; Default Next S3;
    Present S5 if !START Next S15; Default Next S4;
    Present S6 if !START Next S15; Default Next S5;
    Present S7 if !START Next S15; Default Next S6;
    Present S8 if !START Next S15; Default Next S7;
    Present S9 if !START Next S15; Default Next S8;
    Present S10 if !START Next S15; Default Next S9;
    Present S11 if !START Next S15; Default Next S10;
    Present S12 if !START Next S15; Default Next S11;
    Present S13 if !START Next S15; Default Next S12;
    Present S14 if !START Next S15; Default Next S13;
    Present S15 if !START Next S15; Default Next S14;
}

```

### カウンタの PLD ファイルの実行

```

Name    execnt;
Partno  execnt;
Date    09/19/91;
Revision 00;
Designer Teixeira;
Company  PROTEL INTERNATIONAL;
Location none;
Assembly none;

```

```

Device    G16V8MS;

order: clk, %1, !Enable, %1, START, %1, Q3..0, %1, OFF;
vectors:
$MSG" E";
$MSG" n S ";
$MSG" a T ";
$MSG" c b A O";
$MSG" l l R QQQQ F";
$MSG" k e T 3210 F";
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
$MSG" Start counting down";
C 0 1 HHHL L
C 0 1 HHLH L
C 0 1 HHLL L
C 0 1 HLHH L
C 0 1 HLHL L
C 0 1 HLLH L
C 0 1 HLLL L
C 0 1 LHHH L
C 0 1 LHHL L
C 0 1 LHLH L
C 0 1 LHLL L
C 0 1 LLHH L
C 0 1 LLHL L
C 0 1 LLLH L
$MSG" reached 0 so 'OFF' should go high";
C 0 1 LLLL H
C 0 1 LLLL H
C 0 1 LLLL H
C 0 1 LLLL H
C 0 1 LLLL H
C 0 1 LLLL H

```



```

C 0 1 LLLL H
$MSG" turn off start and go back to FF state";
C 0 0 HHHH L
$MSG" start counting down again";
C 0 1 HHHL L
C 0 1 HHLH L
C 0 1 HLLH L
C 0 1 HLHH L
C 0 1 HLHL L
C 0 1 HLLH L
$MSG" turn off start and go back to FF state";
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L
C 0 0 HHHH L

```

カウンタの SI ファイル(シミュレーション入力)の実行

## 交通官制装置

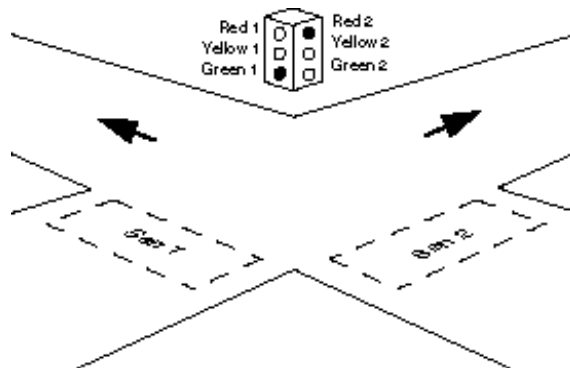


図 8-39 交差点の信号機

この例はコンパイラによるステートマシンの簡単なチュートリアルです。

## ステートマシンのインプリメント

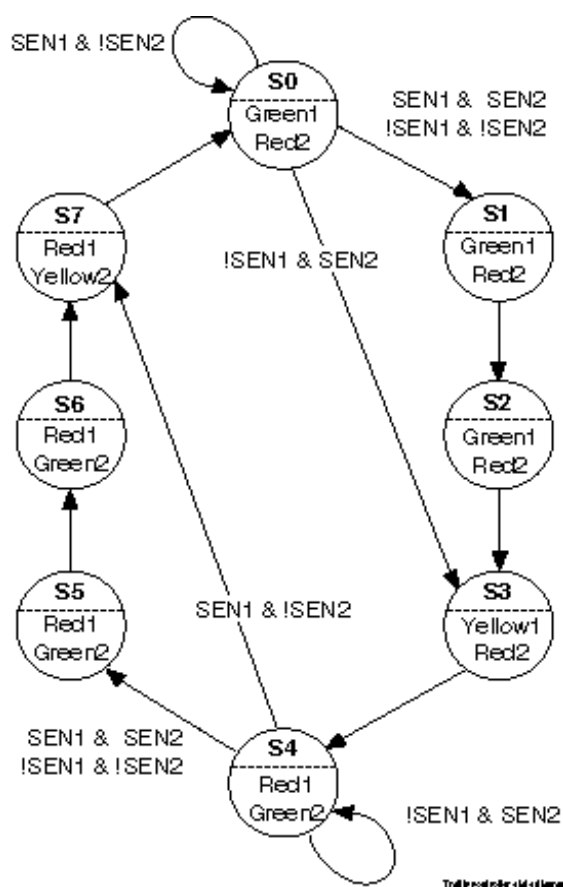


図 8-40 交通官制装置のステートダイアグラム

LDI 交通官制装置で、交差点の交通をコントロールします。それぞれの方向には、赤、黄、緑のライトがあります。設計は、3 ビットのステートマシンで、8 ステートの状態を使用してインプリメントされます。各方向には、センサが一つずつあります。センサにより交通の状況に応じて信号機を変化することができます。例えば、方向 1 の赤信号で車が待っていて方向 2 には車がない場合、信号は赤から緑に変わり車をいかせます。方向 2 の信号は赤にかわります。現在の状況または外部の入力によりある状態からある状態への遷移が起こるために、これは Moor ステートマシンの例です。

ステートは 2 つの組みに分けることができます。ステート 0 から 3 は、Green1/Red2 と Green1/Red1 への遷移を表わします。ステート 4 から 7 は Green2/Red1 と Green1/Red2 への遷移を表わします。ステート 0 とステート 4 は単に方向が反対のステートです。

ステート 0 は Green1 が on で Red2 が on の状態を表わします。このステートの間、センサは常に交通状況を監視しています。この場合、センサ 1 が真でセンサ 2 がフォールスの場合、このままの状態が続けられます。この状態は、ステート 0 に付けられたループバック線でしめされます。

両方のセンサが真で、両方向に車がいる場合、信号は方向 1 と方向 2 の間で切り替える必要があります。Sen1&Sen2 というラベルが付けられたステ

ート 0 からステート 1 への遷移がこれを表わしています。両方のセンサが偽の場合も同じ動作になります。ステート 1、2、3 は、過渡状態です。官制装置が、これらの状態を続けることはありません。これらにより、遅れを持たせたり、信号が赤に変わる前に黄色を表示したりします。

ステート 0 ではセンサ 1 が偽になりセンサ 2 が真になると、官制装置は直接ステート 3 からステート 4 に変わります。待っている車がないので遅れが必要ないからです。

## PLD ソースファイル

交通官制装置のステートマシンのインプリメントは P22V10 または P20G10 に行われます。この PLD ファイルには数種類のセクションがあります。

### ヘッダー

これにより、設計の名前やドキュメントフィールド、DEVAICE フィールドをインクルードできます。このセクションではこれ以上のことは省略します。

### ピン宣言

これらの宣言により名前や信号の極性をデバイスのピンに割り付けます。信号を駆動するピンはエクスクラメーションマーク(!)付きで宣言されます。従って、信号はアクティブローです。これは、通常のコンパイラとは異なる点で多くのコンパイラではピン宣言で極性を指定しても無視されます。これらの宣言をアクティブローにする理由は、この設計は LED を使用する回路基盤で使用されるためです。従って、LED から信号を引っ張るためにアクティブローになっています。簡単にピン宣言を変更できる CUPL の機能により、式を変更しないで設計の内容を変更できます。

### \$DEFINE 命令

この命令により、PLD ファイルをより読みやすくなります。これらの命令により、ステートマシンのすべてのステートを 2 進数番号ではなく名前で参照することができます。\$DEFINE 命令は、文字列を置き換えるプリプロセッサ命令です。ステートを定義する以外にも多くの利用方法があります。

### 中間変数

これには、FIELD のステートビットがあります。FIELD は複数の信号を一つお名前て扱うのに使用されます。

### ステートマシン記述

これは、状態遷移図のテキスト表現です。図とステートマシン記述は一対一に対応します。

ステートマシン記述の最初は、SEQUENCE 命令です。この命令はこれから記述されようとするステートマシンを示します。この場合、D レジスタを使用したいので SEQUENCED を使用しました。P22V10 は D レジスタしか

持たないので単に SEQUENCE キーワードを使用することもできます。この場合、コンパイラはデフォルトで D レジスタを使用します。ステートビットに使用される変数は SEQUENCE キーワードと一緒に宣言されます。ここでは、3 つのレジスタ出力ピンを Q2,Q1,Q0 を持つ Traffic と呼ばれる FIELD を前に宣言しました。

ステートマシンの本体は、中括弧({ })で囲まれます。

```
SEQUENCE state_bits {  
    ...  
    body of state machine  
    ...  
}
```

PRESENT 命令によりステートを宣言します。ステートマシンには各ステートに PRESENT 命令があります。以下の PRESENT 命令では、遷移がどのように行われるかを定める IF 命令があります。各 IF 命令はステート図の遷移に対応します。IF ステートメントに続く PRESENT S0 に注意して下さい。IF 命令によりセンサがインプリメントされ対応する他のステートへのトランジションが実行されます。両方のセンサが真あるいは偽の場合方向 1 と方向 2 を連続的に繰り返すことに注意して下さい。

```
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;
```

ステート S0 の最初の行は以下を読む行で指定されます。

Present S0 これは現在のステートが S0 の時と読みます。

これは、IF 命令以下のステートメントです。最初の IF 命令は、両方のセンサが真の場合どうするかを表わします。この場合、S1 と S2 を繰り返したいので、方向 1 が緑から方向 2 が緑へ状態を遷移させます。

```
if Sen1 & Sen2 Next S1;
```

2 番目の IF 命令は、両方のセンサが偽の場合どうするかを表わします。この場合、S1 と S2 を繰り返したいので、ステート S1 へ戻されます。

```
if !Sen1 & !Sen2 Next S1;
```

3 番目の IF 命令は、方向 1 が緑でこの方向に車がなく車がセンサ 2 をトリガした場合どうするかを表わします。この場合、方向 1 を赤、方向 2 を緑にする必要があります。これは、方向 1 を黄色から赤にし方向 2 を緑にする動作が必要です。すなわち、方向 1 を赤にし方向 2 を緑にするステート S4 へ自動的に行くステート S3 へステートを遷移させます。

```
if !Sen1 & Sen2 Next S3;
```

4 番目の IF 命令は、方向 1 が緑で方向 1 の車がセンサ 1 をトリガした場合どうするかを表わします。同時に、センサ 2 はトリガされません。この場

合、方向 1 は緑のままで方向 2 は赤のままです。このような場合、ステートマシンが同じ状態を続けるように現在の状態を記述します。

```
if Sen1 & !Sen2 Next S0;
```

次のステートメントは OUT 命令です。このステートで駆動する信号を表わします。OUT 命令は単独で使用されるのでこれらは非同期出力です。NEXT 命令と一緒に使用すると同期出力にすることができます。詳細は、CUPL 言語リファレンスの OUT 命令の使用に関するガイドを参照して下さい。この場合、Green1 で Red2 が真になります。

```
Out Green1 Out Red2;
```

### その他の式と宣言

ここには、出力イネーブルの式と命令の最小化が記述されます。

一旦、状態遷移図が完成すると、CUPL ステートマシン記述へ変換することは簡単です。

## アクティブハイとアクティブロー設計

### イントロダクション

設計がアクティブローの場合、予想と違う動作をする場合があります。アクティブローで設計するという事は、動作が厳密の同じで出力信号だけが反転されているということではありません。Tレジスタの例を示します。

この例では、出力の一つが自分自身にフィードバックされる R7 と名前が付けられた出力 T レジスタがあります。アクティブハイ設計の場合とアクティブロー設計の場合を調べてみます。

ここでは、以下の 2 項目を仮定します。

1. R7 は PLD ファイルでアクティブハイ(R7)で定義される。
2. R7 は PLD ファイルでアクティブロー(!R7)で定義される。

### アクティブハイ出力

図 8-41 に R7 がアクティブハイで定義された場合の構成を示します。

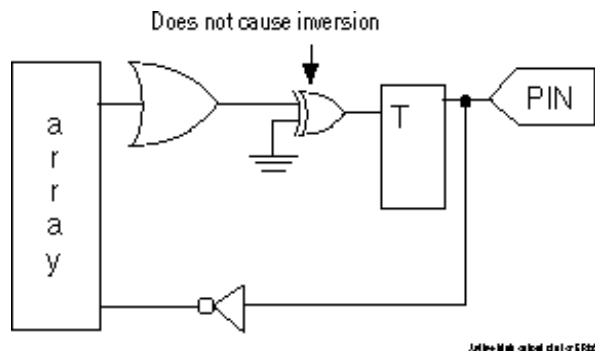


図 8-41 EP324 のアクティブハイ出力ピン

全タームが真でクロックが入力されるとレジスタの状態が変化します。図 8-42 にシミュレータによりベクタが加えられた場合のこのような状況を示します。

```
PIN 1 = CLK;

PIN 38 = D7;
PIN 23 = D6;
PIN 5 = ROW;
PIN 6 = LD;

PIN 12 = R7;

PINNODE 47 = R6;

R7.T = LD & ROW & !R7 & D7    /* pterm 1 */
      # LD & ROW & R7 & !D7;  /* pterm 2 */

R6.T = LD & ROW & !R6 & D6
      # LD & ROW & R6 & !D6;
図 8-42 アクティブハイ出力を使用する PLD ファイル
ORDER: CLK, ROW, LD, D7..6, R7, R6;

VECTORS:
0001:C 1 1 01 LH
0002:C 0 0 00 LH
0003:C 1 1 10 HL
0004:C 0 0 00 HL
```

図 8-43 アクティブハイ出力を使用する SO ファイル

図 8-44 に R7 がアクティブハイで宣言された場合、コンパイラがフィードバック経路をどのように配置するかを示します。

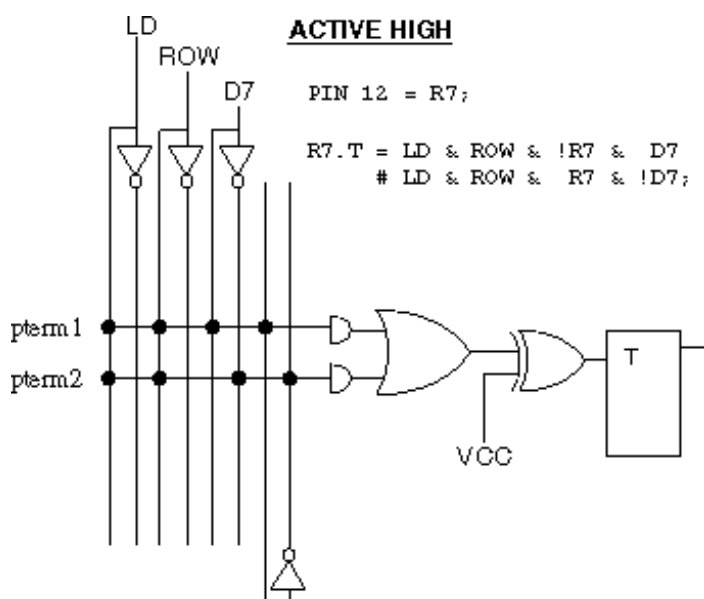


図 8-44 アクティブハイダイアグラム

最初のベクタ: R7 への T 入力の最初のベクタを確認して下さい。それからその他の事項に注目して下さい。

第一に、デバイスの電源が OFF の場合、レジスタはローです。この場合、レジスタへ直接フィードバックされるので電源が投入された時にレジスタがどんな状態であるかを確認することは重要です。

第二に、R7 と D7 の以前の値はローのため、ベクタはレジスタへの T 入力をローにします。D7 がローなので、R7 の最初のプロダクトタームはローです。R7 の以前の値はローなので、次のプロダクトタームはローです。このように、クロックが入力された時に、T 入力がローの場合、レジスタは同じ状態(ロー)のままです。

2 番目のベクタ: 2 番目のベクタを確認して下さい。R7 のプロダクトタームがローでレジスタがトグルされないことも確認して下さい。この理由の一つは LD と ROW がローだからです。レジスタはローのままです。

3 番目のベクタ: 3 番目のベクタにより、T レジスタの R7 がトグルされます。LD と ROW、D7 がハイで R7 の以前の状態がローのためにこのようなことが起こります。これにより、R7.T のプロダクトターム 1 はハイでクロックが入力されるとレジスタはトグルされハイになります。

4 番目のベクタ: 4 番目のベクタにより、レジスタへの T 入力はローになります。従って、クロックが入力されてもレジスタの状態は変化しません。このために LD と ROW は両方ともローです。

## アクティブロー出力

図 4-45 に R7 がアクティブローで定義される場合の構成を示します。

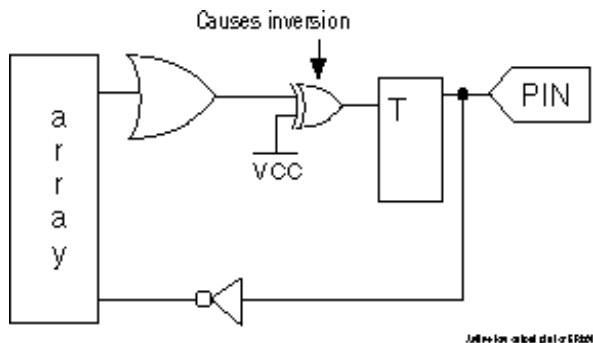


図 8-45 EP324 のアクティブロー出力ピン

ピンの極性がアクティブローとして宣言されているので、コンパイラによりフィードバックは反転されることに注意して下さい。

全タームが真でクロックが入力されると、レジスタの状態は変化しません。これは、レジスタの前で反転が行われるためです。両方のタームの評価が偽の場合にだけレジスタはトグルされます。このようなことは、シミュレータによりベクタが与えられると図 6.7 のように示されます。

図 8-46 にアクティブローで宣言された R7 の PLD ファイルを示します。

```
PIN 1 = CLK;

PIN 38 = D7;
PIN 23 = D6;
PIN 5 = ROW;
PIN 6 = LD;

PIN 12 = !R7;

PINNODE 47 = R6;

R7.T = LD & ROW & !R7 & D7    /* pterm 1 */
      # LD & ROW & R7 & !D7;  /* pterm 2 */

R6.T = LD & ROW & !R6 & D6
      # LD & ROW & R6 & !D6;
```

図 8-46 アクティブロー出力を使用する PLD ファイル

```
ORDER: CLK, ROW, LD, D7..6, !R7, R6;

VECTORS:
0001:C 1 1 01 LH
0002:C 0 0 00 HH
0003:C 1 1 10 HL
0004:C 0 0 00 LL
```

図 8-47 アクティブロー出力を使用する PLD ファイル

図 8-48 に R7 がアクティブローで宣言される場合、コンパイラがフィード



バック経路をどのように配置するかを示します。

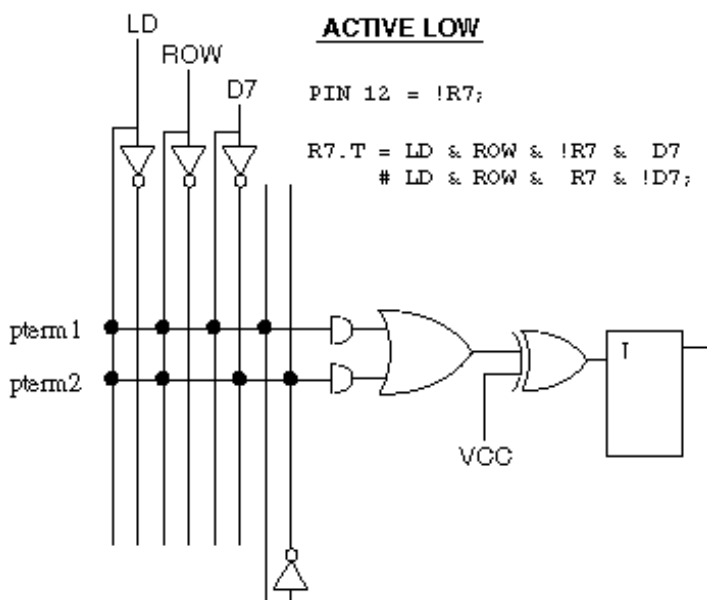


図8-48 アクティブローダイアグラム

最初のベクタ: R7 への T 入力の最初のベクタを確認して下さい。それからその他の事項に注目して下さい。

第一に、デバイスの電源が OFF の場合、レジスタはローです。この場合、レジスタへ直接フィードバックされるので電源が投入された時にレジスタがどんな状態であるかを確認することは重要です。

第二に、ベクタはクロックが入力された時にレジスタがトグルされないレジスタへの T 入力をローにします。これは、プロダクトタームの少なくとも一つが T レジスタへ到達する前に XOR ゲートにより極性が反転されるハイの信号を評価するためです。

R7 や LD、ROW の最初のプロダクトタームへの入力ハイで D7 はローです。これによりこのタームはローになります。

R7 の 2 番目のプロダクトタームはハイになります。LD と ROW はハイで D7 はローです。D7 は 2 番目のプロダクトタームの式で !D7 となるので、AND ゲートへのこの入力真です。T レジスタの以前の値は電源投入時の状態のためローです。フィードバックは 2 番目のプロダクトタームで R7 として定義されます。すなわち、ピンはアクティブローで宣言されたので、R7 が真すなわちローの時、この式では式へのフィードバックが真になります。

R7 のピンがピンステートメントでアクティブローで定義されているので、R7 が式の右辺で使用される場合、フィードバック信号はレジスタの信号と反対になります。式の右辺の R7 は R7 が真であることを意味します。R7 がハイという意味がありません。レジスタがローの時に R7 は真なので、コンパイラによりフィードバックされる信号はレジスタとは反対になります。

式の右辺の!R7 は R7 が偽であることを意味します。R7 がローという意味ではありません。レジスタがハイの場合、R7 は偽のため、コンパイラによりフィードバックされた信号はレジスタの反対になります。

アクティブロー出力ピンを使用する場合、フィードバックは反転されます。アレイへフィードバックされる信号はレジスタの反対になります。

従って、2 番目のプロダクトタームへのすべての入力是真です。このため OR ゲートからの出力も真になります。

OR ゲートからの出力は信号を反転する XOR ゲートに入ります。これにより、レジスタへの T 入力の信号はローになります。レジスタはトグルされません。

2 番目のベクタ: 2 番目のベクタを確認して下さい。R7 のプロダクトタームが両方ともローの場合、レジスタがトグルされます。この理由の一つは LD と ROW がローのためです。XOR ゲートによりレジスタはハイへとグルされます。

3 番目のベクタ: 3 番目のベクタは T レジスタ R7 をトグルする原因にはなりません。これは、タームの一つがハイになるためです。LD や ROW、D7 はハイで R7 の以前の状態はハイです。従って、アクティブローに定義される出力 R7 は偽です。これにより出力 R7 が偽になるので、プロダクトターム 1 を満足します。これは、ターム 1 をハイにします。従って、OR ゲートからの出力はハイです。信号は、T レジスタに到達する前に XOR ゲートにより反転されます。レジスタはトグルされません。

4 番目のベクタ: 4 番目のベクタによりレジスタへの T 入力はハイになり、従って、クロックが入力されるとレジスタはトグルします。このようなことは、プロダクトタームが偽の場合に起こります。LD や ROW がローなので、両方のプロダクトタームは偽になります。OR ゲートからの出力は XOR ゲートにより反転され、ハイ信号が T レジスタへ入力されます。クロックが入力されるとレジスタがトグルされます。

### アクティブロー設計を用いてアクティブハイのシミュレーションを行なう方法

アクティブローで設計してもシミュレーションでは、出力が真の時は HIGH で出力が偽の時は LOW で表示したい場合があります。CUPL では、設計をアクティブローで行い、(シミュレータ入力).SI ファイルでピンをアクティブハイで宣言すれば、このようなことができます。

```
PIN 1 = CLK;  
  
PIN 38 = D7;  
PIN 23 = D6;  
PIN 5 = ROW;  
PIN 6 = LD;  
  
PIN 12 = !R7;  
  
PINNODE 47 = R6;
```

```

R7.T = LD & ROW & !R7 & D7
      # LD & ROW & R7 & !D7;

R6.T = LD & ROW & !R6 & D6
      # LD & ROW & R6 & !D6;

```

図 8-49 PLD ファイルの例 (R7 active-low)

図 8-50 に R7 をアクティブハイでシミュレーションした出力ファイルを示します。ただし、この場合 R7 は PLD ファイルではアクティブローで宣言されています。

```

ORDER: CLK, ROW, LD, D7..6, R7, R6;

VECTORS:
0001:C 1 1 01 HH
0002:C 0 0 00 LH
0003:C 1 1 10 LL
0004:C 0 0 00 HL

```

図 8-50 SO ファイル(R7 は SI の中でアクティブハイです。)

この例では、R7 は PLD ファイルの中でアクティブローで定義されました。これは、PLD ファイルのピン命令に !R7 があるのでわかります。設計では、アクティブローです。

シミュレータ入力ファイルでは、R7 はアクティブハイで定義されます。これは、出力を真の時に HIGH で表示し偽の時に LOW で表示させるためです。これらの値は予想とは反対になるので、機能的なテストではエラーになります。シミュレータは、機能テストのために正しい値を JEDEC ファイルに書き込みます。

シミュレータは設計をアクティブローでシミュレーションします。ただし、信号が真の場合 H、信号が偽の場合 L と表わされます。

# CUPL 言語リファレンス

この章では CUPL 言語の要素とシンタックスを説明します。

## 言語要素

このセクションでは、

### 変数

変数とは、英数文字の文字列でデバイスピンや内部ノード、定数、入力信号、出力信号、中間信号、信号の組みを指定します。このセクションでは変数を作成する規則について説明します。

- 変数は、英数字またはアンダースコアで始めることができます。但し、少なくともひとつ以上のアルファベット文字が含まれる必要があります。
- 変数は大文字と小文字を区別します。
- 変数名にスペースを使用することはできません。アンダースコアを使用して語を区切って下さい。
- 変数には 31 文字まで使用できます。変数名が長すぎる場合、31 文字に切り詰められます。
- CUPL の予約シンボル(表 9-2 参照)を変数に使用することはできません。
- CUPL の予約キーワード(表 9-1 参照)を変数に使用することはできません。

有効な変数名の例を以下に示します。

```
a0
A0
8250_ENABLE
Realtime_clock_interrupt_address
```

上記の例でアンダースコアを使用することにより、変数名が読みやすくなることに注意して下さい。また、大文字と小文字の変数の違いに注意して下さい。変数 A0 と a0 は区別されます。

無効な変数名の例を以下に示します。

- |            |                                   |
|------------|-----------------------------------|
| 99         | アルファベット文字がありません。                  |
| I/O enable | 予約シンボル (/) があります。                 |
| out 6a     | スペースがあります。システムは 2 個の別の変数として認識します。 |
| tbl-2      | ダッシュがあります。システムは 2 個の変数として認識します。   |

## インデックス付き変数

数名を使用してアドレス線のブルーパやデータ線、その他の順番に番号が付けられるアイテムを表わすことができます。例えば、以下の変数名により、マイクロプロセッサの下位のアドレス線 8 本を割り付けることができます。

A0 A1 A2 A3 A4 A5 A6 A7

上記のように、番号で終わる変数名はインデックス付き変数として参照されます。

インデックス付き変数は 0 から始めることを推奨します。すなわち、X1...5 とする代わりに X0...4 として下さい。

インデックス番号は常に 0 から 31 の 10 進数値を指定して下さい。ビットフィールド演算（このセクションのサブトピックの Bit field Declaration Statements を参照）で使用される場合、インデックス番号 0 の変数が最下位ビットになります。

31 より大きい番号で終わる変数はインデックス付きの変数ではありません。

有効なインデックス付き変数名の例を以下に示します。

a23  
D07  
D7  
counter\_bit\_3

0 に続くインデックス付き変数の違いに注意して下さい。すなわち、変数 D07 は D7 とは違います。

無効なインデックス付き変数名の例を以下に示します。

D0F          インデックス番号が 10 進数値ではない。

a36          インデックス番号が範囲から出ている。

これらは、変数としては有効ですが、インデックスとしては無効です。

## 予約語と予約シンボル

CUPL は、特定の文字列をキーワードと呼ばれる予め定義された意味持たせて使用しています。これらのキーワードは CUPL で名前として使用することはできません。

APPEND	FORMAT	OUT
ASSEMBLY	FUNCTION	PARTNO
ASSY	FUSE	PIN
COMPANY	GROUP	PINNNO
CONDITION	IF	PRESENT
DATE	JUMP	REV
DEFAULT	LOC	REVISION
DESIGNER	LOCATION	SEQUENCE

DEVICE	MACRO	SEQUENCED
ELSE	MIN	SEQUENCEJK
FIELD	NAME	SEQUENCERS
FLD	NODE	SEQUENCET
	TABLE	

表 9-1 CUPL 予約語

また、CUPL はある特定のシンボルを予約しています。したがって、これらの記号を変数名に使用することはできません。

&	#	( )	-
*	+	[ ]	/
:	.	..	/* */
;	,	!	'
@	\$	^	=

表 9-2 CUPL 予約シンボル

## 数値

CUPL コンパイラでの数値が含まれる演算はすべて 32 ビット精度で行われます。したがって、数値は 0 から  $2^{32}-1$  の値をもつことができます。数値は 4 種類の基数で表わすことができます。すなわち、2 進数、8 進数、10 進数、16 進数です。デフォルトの基数は 16 です。ただし、デバイスのピン番号やインデックス付き変数には 10 進数が使用されます。基数の違う数値同志は、表 9-3 で示されるプリフィックスを数値の頭に付けることで使用することができます。基数の変更があると、デフォルトの基数も変更されます。

基数名	基数	プリフィックス
2 進数	2	'b'
8 進数	8	'o'
10 進数	10	'd'
16 進数	16	'h'

表 9-3 数値 基数 プリフィックス

基数文字はシングルクォートで囲まれ大文字でも小文字でもかまいません。有効な数値の指定の例を表 9-4 に示します。

数値	基数	10 進数値
'b'0	2 進数	0
'B'1101	2 進数	13
'O'663	8 進数	435

'D'92	10 進数	92
'h'BA	16 進数	186
'O'[300..477]	8 進数(レンジ)	192..314

表 9-4 基数変換の例

2 進数や 8 進数、16 進数値は dont-care 値(X)と数値を持つ事ができます。dont-care 値を用いた有効な数値指定の例を表 9-5 に示します。

数値	基数
'b'1X11	2 進数
'O'0X6	8 進数
'H'[3FXX..7FFF]	16 進数(レンジ)

表 9-5 Dont-Care 数値の例

## コメント

コメントは、論理記述ファイルの重要な部分です。コメントによりコードを読みやすくしたりコードの目的をドキュメント化したりできます。ただし、コメントはシンタクスの検査が行われる前にプリプロセッサにより削除されるので、コンパイル時間にはあまり影響はありません。/\*と\*/を使用してコメントを囲んで下さい。プログラムはこれらの記号の間の記述をすべて無視します。

コメント行を、複数行に設定する事ができます。すなわち、コメントは行の終わりで終了されません。コメントを入れ子にすることはできません。有効なコメントの例を図 9-1 に示します。

```

/*****
/* This is one way to create a title or */
/* an information block */
*****/

/*
This is another way to create an information block
*/

out1=in1 # in2; /* A Simple OR Function */
out2=in1 & in2; /* A Simple AND Function */
out3=in1 $ in2; /* A Simple XOR Function */

```

図 9-1 コメントの例

## リスト表記

省略表記は CUPL 言語の重要な特徴です。

最もよく使われる省略表記はリストです。この表記はピンやノードの定義やビットフィールド宣言、論理式、セット演算に一般的に使用されます。リストフォーマットを以下に示します。

```
[variable, variable, ... variable]
```

ここで

[ ]は、変数の組みとしてリストのアイテムを囲むために使用される括弧です。

リスト表記の例を以下に示します。

```
[UP, DOWN, LEFT, RIGHT]  
[A0, A1, A2, A3, A4, A5, A6, A7]
```

変数名に順番に番号が付けられている場合、昇順でも降順でもかまいません。フォーマットを以下に示します。

```
[variablem..n]
```

ここで

m は変数のリストの最初のインデックス番号です。

n は変数のリストの最後の番号です。変数名を書かないで n だけを書く事ができます。

例えば上記の例の 2 行目は以下のように記述することができます。

```
[A0..7]
```

インデックス番号は 10 進数値で連続であると仮定されます。変数インデックスの最初の 0 は作成される変数名から削除されます。例えば

```
[A00..07]
```

は、以下の省略形です。

```
[A0, A1, A2, A3, A4, A5, A6, A7]
```

以下の省略形ではありません。

```
[A00, A01, A02, A03, A04, A05, A06, A07]
```

リスト表記の 2 つの形は組み合わせて使用することができます。例えば、以下の 2 つの表記法は等価です。

```
[A0..2, A3, A4, A5..7]  
[A0, A1, A2, A3, A4, A5, A6, A7]
```

## テンプレートファイル

CUPL 言語を用いて論理記述ソースファイルが作成されると、ヘッダー情報やピン宣言、論理式などの特定の情報が入力されます。File-New を選択すると、アドバンスド PLD は Wizard を自動的に起動し、論理記述ファイルを構築する手助けをします。そして、ソースファイルの適切な構造を含むテンプレートファイルがインクルードされます。テンプレートファイルは以下のセクションをインクルードします。

図 9-2 テンプレートファイル



```

Name          XXXXX;
Partno        XXXXX;
Date          XX/XX/XX;
Revision      XX;
Designer      XXXXX;
Company       XXXXX;
Assembly      XXXXX;
Location      XXXXX;
/*****/
/*              */
/*              */
/*****/
/* Allowable Target Device Types:          */
/*****/

/** Inputs **/
Pin    =    ; /*          */
Pin    =    ; /*          */
Pin    =    ; /*          */
Pin    =    ; /*          */

/** Outputs **/
Pin    =    ; /*          */
Pin    =    ; /*          */
Pin    =    ; /*          */
Pin    =    ; /*          */

/** Declarations and Intermediate Variable Definitions **/

/** Logic Equations **/

```

図 9-2 テンプレートファイル

テンプレートファイルには以下のセクションがあります。

ヘッダー情報 - ソースファイルのヘッダー情報セクションはリビジョンや文書管理の目的でファイルを識別します。

Title Block - 設計の関数の説明や使用できるターゲットデバイスを記述するために囲まれた空間です。

Pin Declaration - 入出力ピン宣言やピン割り付けを説明するためのコメントスペースの適切なフォーマットのキーワードや演算です。ピンが定義されると、extra pin=;行はすべて削除されます。この行が残っていると、シンタクスエラーが発生します。

/\*Inputs\*/と/\*Outputs\*/は、読みやすさのためのグループ分けを示すコメントです。ピンタイプがどんな順番で割り付けられても、論理記述ファイルでそれらがどんなに使用されても問題はありません。

Declaration and Intermediate Variable - ビットフィールド宣言(このセクションのサブトピック Bit Field Declaration Statements and Node Declaration

Statement を参照して下さい。)やノード宣言(このセクションのサブトピック Logic Equations を参照して下さい。)などの定義をおこなうためのスペースです。

Logic Equation - デバイス (このセクションのサブトピック Logic Equations を参照して下さい。)の関数を表現する論理式を記述するための空間です。

通常、このセクションはファイルの最初に置かれます。CUPL により 10 個のキーワードをヘッダー情報ステートメントに使用することができます。各ステートメントは有効な ASCII 文字が続くキーワードで始まり、スペースと特別なシンボルが含まれるキーワードで始まります。そして、各ステートメントはセミコロンで終わります。表 9-6 は CUPL ヘッダーキーワードとヘッダー情報を示します。

### ヘッダー情報

ソースファイルのヘッダー情報セクションはリビジョンや文書管理の目的でファイルを識別します。テンプレートファイルにより、DEVICE と FORMAT 以外のヘッダーキーワードが与えられます。適切な CUPL ヘッダー情報の例を以下に示します。

キーワード	情報
NAME	通常は、ソース論理記述ファイル名を使用します。オペレーティングシステムで有効な文字列だけを使用して下さい。ここで指定される名前により、JEDEC、ASCII-HEX、HL ダウンロードファイルの名前が決まります。NAME フィールドは 32 文字までのファイル名を受け入れます。8 文字までのファイル名を使用する DOS などのシステムを使用する婆し、ファイル名は、切り詰められます。
PARTNO	特定の PLD 設計の会社特有の部品番号 (通常、製造元から発行されます。)を指定して下さい。部品番号はターゲット PLD の型ではありません。GAL デバイスでは、最初の 8 文字がエンコードされデバイスヒューズマップの User Signature Fuses で 7 ビットの ASCII を使用します。
REVISION	ファイルが最初に作成されると 01 から始まりファイルが変更されるとインクリメントされます。省略形で REV を使用することができます。
DATE	ソースファイルが変更される毎に現在の日付に変更されます。
DESIGNER	設計者の名前を入力して下さい。
COMPANY	ドキュメンテーションの慣習のためと、大量の PLD を製造するために半導体製造メーカーに仕様書送るためとの理由で会社名を入力して下さい。
ASSEMBLY	PLD が使用される PC 基板のアッセンブリ名またはアッ

センブリ番号を与えて下さい。省略形の ASSY を使用することができます。

LOCATION PC 基板のリファレンスまたは PLD が置かれる座標を指定して下さい。省略形 LOC を使用することができます。

DEVICE コンパイル時のデフォルトのデバイスタイプを設定して下さい。複数のデバイス用のソースファイルでは、デバイスのタイプが違う場合、DEVICE を各セクションで使用する必要があります。

FORMAT 現在の論理記述セクションにオーバーライドするダウンロード出力フォーマットを設定して下さい。出力フォーマットに有効な値を以下に示します。

- h ASCII-hex 出力を生成します。
- i HL 出力を生成します。
- j JEDEC 出力を生成します。

FORMAT は、Configure AdvancedPLD ダイアログボックスで設定されるオプションをオーバーライドします。異なる部品が互換性のない出力フォーマットを持つ複数デバイスのソースファイルで使用するのに便利です。複数のフォーマット値を同時に指定して複数のタイプの出力を生成できます。フォーマット値は小文字で設定する必要があります。

```
Name Name                SAMPLE ;
Partno Partno             P9000183 ;
Revision Revision         02 ;
Date Date                 1/11/89 ;
Designer Designer         Osann ;
Company Company           Protel International ;
Assembly Assembly         PC Memory Board ;
Location Location         U106 ;
Device Device             F155;
Format Format              ij ;
```

ヘッダー情報を省略すると、CUPL はワーニングメッセージを出します。ただし、コンパイルはそのまま続けられます。

## ピン宣言命令

ピン宣言命令を使用して、ピン番号宣言したり、ピンを記号変数名にピンを割り付けたりします。ピン宣言のフォーマットを以下に示します。

```
PIN pin_n=[!]var ;
```

ここで

PIN はピン番号を宣言したりピンに変数名を割り付けたりするキーワードです。

pin\_n は、10 進数のピン番号またはリスト表記を使用してグループ化

されたピン番号のリストです。すなわち、

```
[pin_n 1, pin_n 2 ... pin_nn]
```

!は、省略可能なシンボルです。入出力信号の極性を定義します。

=は割り付け演算子です。

var はひとつの変数名またはリスト表記を使用してグループ化された変数のリストです。すなわち、

```
[var, var ... var]
```

;は、ピン宣言命令の終わりを示すセミコロンです。

テンプレートファイルには、ピン変数を入力するためのセクションが用意されています。

極性の考え方は、あいまいになることがしばしばあります。どんな PLD 設計でも設計者は信号が真か偽かを考慮しています。設計者にとっては、これが信号のハイになるかローになるかは重要な問題ではありません。基板設計のいろいろな理由から、真の場合論理レベル 0 (ロー) で、偽の場合、論理 1 (ハイ) になります。この信号は、信号がローの時にアクティブにされるので、アクティブローと見なされます。また、ロートゥルーと呼ばれることもあります。アクティブハイからアクティブローへ信号が変わると、極性が変わります。

このように、CUPL を使用するとピン定義で信号の極性を宣言できるので、上記のことを考慮する必要はありません。CUPL シンタックスで式を記述する場合、設計者は信号の極性を考慮する必要がありません。ピン宣言により信号の極性の変更を宣言します。

以下の関数を記述する場合

```
Y = A & B;
```

このステートメントは、A が真で B が真であれば Y は真を意味します。このステートメントを P22V10 デバイスで簡単に実行できます。

```
Pin 2 = A;  
Pin 3 = B;  
Pin 16 = Y;  
Y = A & B;
```

デバイスが回路に組み込まれる場合、論理 1 がピン 2 とピン 3 に加えられるとピン 16 の信号がハイになります。

入力を論理 0 を真としたい場合、設計を以下のように変更して下さい。

```
Pin 2 = !A;  
Pin 3 = !B;  
Pin 16 = Y;  
Y = A & B;
```

!記号がピン宣言に置かれると、反転した極性を示します。従って、この場合も A が真で B が真の場合、Y が真になります。変更されたのは、TRUE=0

と FALSE=1 の変換です。設計レベルでは何も変更されていません。変更したのは、ピン宣言での真が 0 で偽が 1 の変換です。

これにより、設計者は極性の曖昧な記述を最小限にするように設計をレイヤに区別することができます。CUPL は真 / 偽レイヤを保つようにフィードバック信号を変更します。

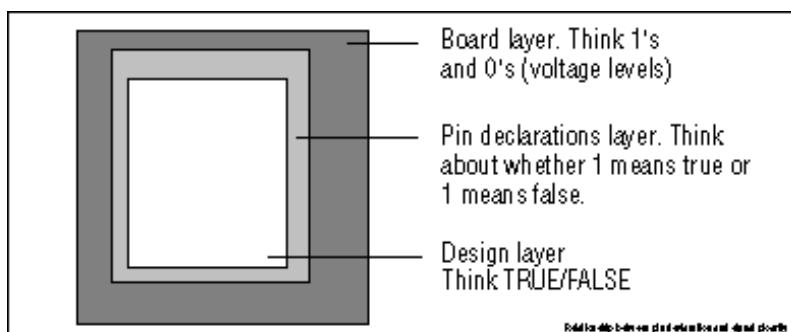


図 9-3 ピン宣言と信号極性の関係

エクスクラメーションピンと(!)を使用して入出力信号の極性を定義して下さい。入力信号がアクティブラー（すなわち、TTL 信号電圧レベルの 0 ボルト）の場合、ピン宣言の変数名の前に!を記述して下さい。!により、論理式でアクティブとされる場合、コンパイラは信号を反転して検知します。仮想デバイス VirtualDevice ではこのルールは例外的に適用できません。仮想デバイスを使用する場合、コンパイラは、ピン宣言の極性を無視します。この場合、式自身を否定する必要があります。

同様に、出力信号がアクティブラーの場合、ピン宣言に!を付けて変数を定義し、論理的に真の形で論理式を記述して下さい。!を使用すると、ターゲットデバイスの制限に関係なくピンを宣言することができます。仮想デバイスを使用する場合、ピン宣言の極性が無視されるため、式自体を否定する必要があります。

アクティブレベルに HI 出力を指定するピン宣言が反転出力した持たないデバイス（PAL16L8 など）用にコンパイルされる場合、コンパイラは論理式にドモルガンの定理を自動的に適用し関数をデバイスに合わせます。

以下の例について考えてみます。論理記述ファイルは PAL16L8 用に記述されています。出力ピンはすべてアクティブハイとして宣言されています。以下の式は OR 関数を指定するために記述されました。

$$c = a \# b ;$$

しかし、PAL16L8 には出力ピンに固定反転バッファがあるために、コンパイラはドモルガンの定理を利用してロジックをデバイスに合わせます。コンパイラは以下のようなプロダクトタームをドキュメンテーションファイルに作成します。

$$c \Rightarrow ! a \& ! b$$

図 9-4 に上記のように記述された過程を示します。

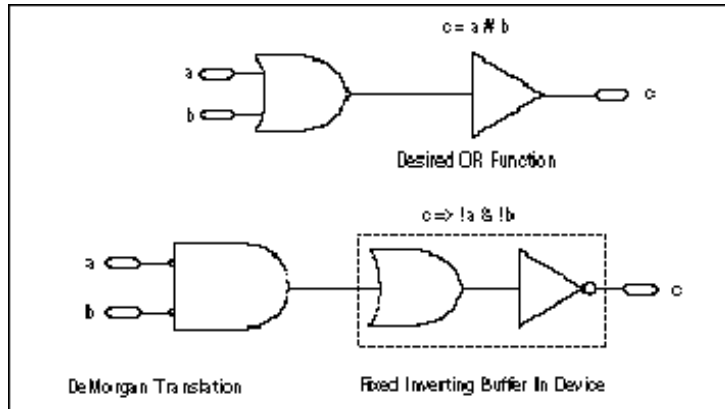


図 9-4 反転バッファのアクティブハイのピン宣言

設計のプロダクトタームが多すぎる場合、コンパイラはエラーメッセージを表示しコンパイル作業を停止します。ドキュメンテーションファイル(ファイル名.DOC)に論理関数を実行するために必要なプロダクトタームの数とデバイスが特定の出力ピンに物理的に持つプロダクトタームの数が記述されます。

有効なピン宣言の例を以下に示します。

```
pin 1      =      clock;          /* レジスタクロック */
pin 2      =      !enable;        /* イネーブル i/o ポート*/
pin [3,4]  =      ![stop,go];     /* 制御信号 */
pin [5..7]=      [a0..2];         /* アドレスビット 0-2 */
```

上記の例の最後から 2 番目は以下の表現の省略形です。

```
pin 3      =      !stop;          /* 制御信号 */
pin 4      =      !go;           /* 制御信号 */
pin 5      =      a0;            /* アドレスビット 0 */
pin 6      =      a1;            /* アドレスビット 1 */
pin 7      =      a2;            /* アドレスビット 2 */
```

仮想デバイス VirtualDevice,ではピン番号は省略されます。これにより、デバイスの制限に関係なく設計を行なうことができます。設計者は結果を検証し、実行するために必要な事柄を決定します。ターゲットデバイスをその後選択して下さい。仮想デバイスを使用する時の有効なピン宣言を以下に示します。

```
pin      =      !stop;          /* 制御信号 */
pin      =      !go;           /* 制御信号 */
pin      =      a0;            /* アドレスビット 0 */
pin      =      a1;            /* アドレスビット 1 */
pin      =      a2;            /* アドレスビット 2 */
```

デバイスピンの入力や出力、入出力は、ピン宣言では指定されません。コンパイラは、ピン変数名が論理仕様で使用される方法からピンの性質を推測します。ターゲットデバイスの論理仕様や物理特性に互換性がない場合、

コンパイラは、間違った使われかたをしているピンを示すエラーメッセージを表示します。

## ノード宣言命令

デバイスの中には、外部のピンを使用できない関数を含むものがあります。しかし、論理式では外部ピンを使う必要がある場合があります。例えば、82S105 にはバリードステートレジスタ（フリップフロップ）とコンプリメントアレイによりトランジションタームを反転する機構があります。それらには、変数名を割り付ける必要があります。これらの関数に関連付けられるピンは無いので、PIN キーワードを使用することはできません。NODE キーワードを使用して埋めこみ関数の変数名を宣言して下さい。

ノード宣言の書式を以下に示します。

```
NODE [!] var ;
```

ここで

NODE は埋めこみ関数の変数名を宣言するためのキーワードです。

!は、内部信号の極性を定義するための省略可能なエクスクラメーションです。

var はひとつの変数名またはリスト表記を使用してグループ化された変数のリストです。

;はステートメントの終わりを表わすセミコロンです。

テンプレートファイルにより与えられるソースファイルの Declarations and Intermediate Variables Definitions section でノードを宣言して下さい。

内部ノードのほとんどはアクティブレベル HI です。従って、エクスクラメーションを内部信号をアクティブレベル LO として定義するために使用しないで下さい。エクスクラメーションを使用すると、コンパイラが多くのプロダクトタームを作成する原因になります。コンプリメントアレイノードの場合は例外です。この場合、アクティブレベル LO の信号です。

宣言ステートメントにピン番号がない場合でも、コンパイラにより、内部的に仮のピン番号が割り付けられます。これらの番号は、付けることができる最も小さな番号から始まり、PINNODE ステートメントにより割り付けられたノードの場合でも順番に定義されます。割り付けは自動的に行われ、使われかた（フリップフロップ、コンプリメントアレイなど）により決められます。従って、変数の順番は関係ありません。しかし、一度ノード変数が宣言されると、変数のために論理式を作成する必要があります。そうしないと、コンパイルエラーになります。

CUPL はノード宣言を使用して、埋めこみ関数の論理式を中間表現とを区別します。

NODE キーワードの使用例を以下に示します。

```
NODE [State0..5]; /* 内部ステートビット */
```

```

        NODE !Invert;          /* コンプリメントアレイ用 */

```

CUPL により埋めこみ関数の割り付けを自動的に行なう NODE キーワードの代わりに、埋めこみ関数の割り付けを行なうキーワードに PINNODE キーワードがあります。PINNODE キーワードは、ノード番号を変数名に割り付けることにより埋めこみノードを厳密に定義するために使用します。これは、ピン宣言ステートメントの働きに似ています。ピンノード宣言のフォーマットを以下に示します。

```

        PINNODE node_n = [!]var;

```

ここで、

PINNODE は、ノード番号を宣言し、それらに変数名を割り付けるためのキーワードです。

node\_n は、10 進数値のノード番号またはリスト表記を使用してグループ化されたノード番号のリストです。例を以下に示します。

```

        [node_n1,node_n2 ... node_nn]

```

!は内部信号の極性を定義する省略可能なエクスクラメーションです。

=は、割り付け演算子です。

var は、一つの変数名または、リスト表記を使用してグループ化された変数のリストです。例を以下に示します。

```

        [var,var ... var]

```

;は、ステートメントの終わりを示すために使用されるセミコロンです。

テンプレートファイルにより与えられるソースファイルの Declarations and Intermediate Variables Definitions section にピンノード宣言を記述して下さい。

ノード宣言と同様に、内部ノードのほとんどはアクティブレベル HI です。従って、エクスクラメーションを使用して内部信号の極性をアクティブレベル LO に定義しないで下さい。エクスクラメーションを使用すると、コンパイラが多くのプロダクトタームを作成する原因になります。コンプリメントアレイノードの場合は例外です。この場合、アクティブレベル LO の信号です。

内部ノードを持つデバイスのノード番号のリストは付録 D にあります。これらのノード番号をピンノード宣言に使用して下さい。

PINNODE キーワードの使用例を以下に示します。

```

PINNODE [29..34] = [State0..5]; /* Internal State Bits */
PINNODE 35 = !Invert;          /* For Complement Array */
PINNODE 25 = Buried;           /* For Buried register part */
/* I/O マクロセルの埋めこみレジスタ部 */

```

## ビットフィールド宣言命令

ビットフィールド宣言により、一つの変数名にビットのグループを割り付



けることができます。フォーマットを以下に示します。

```
FIELD var = [var, var, ... var] ;
```

ここで

FIELD はキーワードです。

var は任意の変数名です。

[var, var, ... var]はリスト表記の変数名のリストです。

=割り付け演算子です。

;)はステートメントの終わりを示すセミコロンです。

括弧 ( ) は、省略することはできません。これらは、リストアイテムを囲むために使用されます。

テンプレートファイルにより与えられるソースファイルの Declarations and Intermediate Variable Definitions セクションにビットフィールド宣言を記述して下さい。

変数名をビットのグループに割り付けると、名前を式のなかで使うことができます。式で表現される演算がグループの各ビットに適用されます。FIELD ステートメントで可能な演算の説明についてはこのセクションの Set Operations を参照して下さい。以下の例は I/O デコーダの 8 ビットの入力アドレス (A0 から A7) を一つの変数名 ADDRESS として参照する 2 つ方法を示します。

```
FIELD ADDRESS = [A7,A6,A5,A4,A3,A2,A1,A0] ;
```

または

```
FIELD ADDRESS = [A7..0] ;
```

FIELD ステートメントが使用されると、コンパイラは内部に 32 ビットのフィールドを作成します。これによりビットフィールドで変数を表わします。各ビットは、ビットフィールドの一つのメンバを表わします。ビットフィールドのメンバを表わすビット番号はインデックス付きの変数が使用される場合インデックス番号と同じです。つまり、A0 は常にビットフィールドのビット 0 に割り付けられます。また、ビットフィールドのインデックス番号の順番は意味がありません。[A0..7]で宣言されたビットフィールドは、[A7..0]で宣言されたビットフィールドと厳密に同じです。このために、異なるインデックス付き変数が同じビットフィールドに含まれることがあるかもしれません。A2 と B2 を含むビットフィールドは両方とも同じビット位置に割り付けられます。これにより、間違った式を作成する可能性があります。

また、ビットフィールドはインデックス付き変数とインデックス付きでない変数を同時に含むことはありません。

インデックス付きの変数とインデックス付きでない変数をフィールドステートメントに混ぜて使用しないで下さい。コンパイラは予期しない結果を生成する可能性があります。

## MIN 宣言命令

MIN 宣言ステートメントは、特定の変数に対して、Configure Advanced PLD ダイアログボックスで指定される最小化レベルをオーバーライドします。書式を以下に示します。

```
MIN var [.ext] = level ;
```

ここで

MIN は Configure Advanced PLD ダイアログボックスで指定される最小化レベルをオーバーライドするためのキーワードです。

var はファイルで宣言されたひとつの変数またはリスト表記を使用してグループ化された変数のリストです。すなわち

```
[var, var, ... var]
```

.ext は変数の機能を表わす省略可能な拡張子です。

level は 0 から 4 までの整数で、最小化レベルに対応します。

;はステートメントの終わりを示すセミコロンです。

MIN 宣言により、同じ設計の異なる出力に異なる最小化レベルを指定できます。例えば、冗長さが必要な出力やプロダクトタームを含む出力には最小化を行なわないようにしたり（非同期ハザード状態を避けるため）、ステートマシンアプリケーションには最高レベルの最小化を行なったりできます。

有効な MIN 宣言の例を以下に示します。

```
MIN async_out      = 0;    /* 最小化は行われません */
MIN [outa, outb]    = 2;    /* 最小化レベル 2 */
MIN count.d = 4;        /* 最小化レベル 4 */
```

上記例の最後の宣言は、d 拡張子を使用し最小化されるレジスタード変数の一つであることを表わしています。

## FUSE 命令

FUSE 命令により、TURBO または MISER ビットの断線を行なうことができます。この命令は慎重に使用することが必要です。正しく使用されなかった場合、予想できない結果になる可能性があります。

```
FUSE (fusenumber, x)
```

この例では、ヒューズ 101 が MISER ビットまたは TURBO ビットです。これにより、ヒューズ番号 101 に断線されます。

例:

```
FUSE(101,1)
```

**FUSE 命令を使用してでたらめなヒューズを断線しないでください。**

ヒューズ命令は MISER ビットまたは、TURBO ビットだけを断線するよう

に設計されています。TURBO ビットまたは MISER ビットの正しいヒューズ番号を指定して下さい。この命令が使用されるたびに、コンパイラはワーニングを生成します。これは、ヒューズ番号が正しいかを 2 重に確認するための注意です。ヒューズ番号を間違えて指定すると、悲惨な結果になります。この命令を使用する場合、くれぐれも注意が必要です。FUSE 命令を設計で使用しおかしな結果になった場合、指定したヒューズ番号とそれが MISER ビットまたは TURBO ビットであるかを確認して下さい。

## プリプロセッサコマンド

ソースファイルが分析される前に、コンパイラのプリプロセッサ部により、ソースファイルが処理されます。プリプロセッサにより、ファイルのインクルードや条件コンパイル、コンパイラのソースファイル処理機能の文字列の置き換えが行われます。

\$DEFINE	\$IFDEF	\$UNDEF
\$ELSE	\$IFNDEF	\$REPEAT
\$ENDIF	\$INCLUDE	\$REPEND
\$MACRO	\$MEND	

表 9-7 プリプロセッサコマンド

プリプロセッサコマンドはすべてダラーサイン (\$) で始まり、一行で使用して下さい。これらのコマンドは、大文字小文字の任意の組み合わせで使うことができます。

### \$DEFINE

このコマンドにより、文字列が他の指定された演算にや数値、記号に置き換えられます

フォーマットを以下に示します。

```
$DEFINE argument1 argument2
```

ここで

argument1 は変数名すなわち特別な ASCII 文字です。

argument2 は有効な演算や数値、変数名です。

\$DEFINE コマンドが記述されると (\$UNDEF コマンドまで) ソースファイルのどんな位置でも argument1 は argument2 で置き換えられます。文字列の置き換えはコンパイラによりソースファイルが分析される前に実行されます。セミコロンやイコール記号をこのコマンドに使用することはできません。

\$DEFINE コマンドにより、数値または定数は他の名前に置き換えることができます。以下に例を示します。

```
$DEFINE ON 'b'1
$DEFINE OFF 'b'0
$DEFINE PORTC 'h'3F0
```

また、\$DEFINE コマンドにより、論理演算子を変更することができます。  
論理演算子の変更例を以下に示します。

\$DEFINE	{	/*	コメントの開始記号の置き換え
\$DEFINE	}	*/	コメントの終了記号の置き換え
\$DEFINE	/	!	NOT 記号の置き換え
\$DEFINE	*	&	AND 記号の置き換え
\$DEFINE	+	#	OR 記号の置き換え
\$DEFINE	:+:	\$	XOR 記号の置き換え

上記の定義は、アドバンスド PLD パッケージに含まれる PALASM.OPR に記述されています。このファイルをソースファイルにインクルードすると（\$INCLUDE コマンドを参照して下さい。）、上記のように置き換えられた論理演算子を使用することができます。また、標準の CUPL 演算子記号も使用できます。

## \$UNDEF

このコマンドは\$DEFINE コマンドの反対の働きをします。フォーマットを以下に示します。

\$UNDEF argument

ここで

argument はこれより前に\$DEFINE コマンドで使用された引き数です。

\$DEFINE コマンドを使用して定義された文字列や記号を再定義する前に、\$UNDEF コマンドを使用して、以前の定義を取り消して下さい。

## \$INCLUDE

このコマンドを使用して、指定されたファイルをソースファイルに取り込まれます。取り込まれたファイルは、ソースファイルに組み込まれコンパイラで処理されます。書式を以下に示します。

\$INCLUDE filename

ファイルをインクルードすることにより、共通に使用されるプログラムの部分を標準化することができます。また、多くのソースファイルで共通に使用される定数を定義するパラメータファイルを分離する場合に便利です。インクルードされるファイルに\$INCLUDE コマンドを記述することもできます。これにより、インクルードのネストが可能です。\$INCLUDE コマンドの場所でファイルがインクルードされます。

例えば、以下のコマンドはソースファイルに PARASM.OPR をインクルードします。

\$INCLUDE PARASM.OPR

PALASM.OPR は、アドバンスド PLD によりインクルードされ、このファイルには、以下の論理記号の置き換えが指定される\$DEFINE コマンドが記述されています。

\$DEFINE	/	!	NOT 記号の置き換え
\$DEFINE	*	&	AND 記号の置き換え
\$DEFINE	+	#	OR 記号の置き換え
\$DEFINE	:+:	\$	XOR 記号の置き換え
\$DEFINE	{	/*	コメントの開始記号の置き換え
\$DEFINE	}	*/	コメントの終了記号の置き換え

## \$IFDEF

このコマンドを使用して、ソースファイルの一部を条件に応じてコンパイルすることができます。フォーマットを以下に示します。

```
$IFDEF argument
```

ここで

argument は、このコマンドより以前に\$DEFINE コマンドにより定義されているかもしれない文字列です。

引き数がすでに定義されている場合、\$IFDEF コマンドに続くソースファイルの内容は\$ELSE または\$ENDIF コマンドまでコンパイルされます。

引き数が定義されていない場合、\$IFDEF コマンドに続くソースファイルの内容は無視されます。\$IFDEF コマンドに続くソースファイルの内容は\$ELSE または\$ENDIF コマンドまでコンパイルされません。

\$IFDEF コマンドを使用すると、コメントを含むソースファイルの一部を一時的にコンパイラに無視させることができます。コメントをネストすることができないので、コメントを含む部分をコメントにすることはできません。以下にこの方法の使用例を示します。NEVER は定義されていない引き数です。

```
$IFDEF NEVER
out1=in1 & in2;      /* 簡単な AND 関数 */
out2=in3 # in4;      /* 簡単な OR 関数 */
$ENDIF
```

NEVER は定義されていないので、コンパイル時に式は無視されます。すなわち、これらの関数はコメントとして扱われる場合と同じです。

## \$IFNDEF

このコマンドにより、ソースファイルの中でコンパイルをする部分の条件を設定できます。

```
$IFNDEF argument
```

ここで

argument は、このコマンドより前に\$DEFINE コマンドにより定義されているかもしない引き数です。

\$IFNDEF コマンドは、\$IFDEF コマンドと反対の動作をします。引き数がかれより以前に定義されていない場合、\$IFNDEF コマンドから\$ELSE または

\$ENDIF コマンドまでのソースステートメントはコンパイルされます。

引き数が定義されている場合、\$IFDEF コマンドから\$ELSE または\$ENDIF コマンドまでのソースステートメントは無視されます。

\$IFDEF コマンドを使用して、互いに排他的な 2 種類の式を持つひとつのソースファイルを作成することができます。\$IFDEF と\$ENDIF コマンドを使用して式のひとつを囲み、引き数を定義するかしないかによりこの 2 つの式を切替えることができます。

例えば、トリステートバッファをすべて直接制御する共通の出力イネーブルピンを持つデバイスとトリステートバッファをイネーブルにするプロダクトタームをひとつ持つデバイスがあるとします。以下の例では、引き数、COMMON\_OE が定義されていないので、それに続く式はコンパイルされます。

```
$IFDEF    COMMON_OE
pin 11          = !enable;      /* OE の入力ピン */
[q3,q2,q1,q0].oe      = enable; /* トリステートの割り
付け */

                                /* 4 の式 */
                                /* 出力 7 */

$ENDIF
```

デバイスに共通の出力イネーブルがある場合、式でそれを表わす必要はありません。従って、上記の例では、共通の出力イネーブルを持つデバイスに対して COMMON\_OE を定義し、コンパイラが\$IFDEF と\$ENDIF の間の式をとばしてコンパイルするようにして下さい。

## **\$ENDIF**

このコマンドを使用して、\$IFDEF コマンドや\$IFDEF コマンドで始まる条件コンパイルを終了して下さい。フォーマットを以下に示します。

```
$ENDIF
```

\$ENDIF コマンドに続くステートメントは、先の\$IFDEF コマンドや\$IFDEF コマンドで説明したように、コンパイルされます。条件コンパイルがネストされている場合、\$IFDEF コマンドや\$IFDEF コマンドのそれぞれのネストのレベルに、\$ENDIF コマンドが必要です。

以下にネストのある\$ENDIF の使用例を示します。

```
$IFDEF prototype_1
pin 1 = set;      /* ピン 1 に set を割り付けます */
pin 2 = reset;    /* ピン 2 に reset を割り付けます */
$IFDEF prototype_2
pin 3 = enable;   /* ピン 3 に enable を割り付けます */
pin 4 = disable;  /* ピン 4 に disable を割り付けます */
$ENDIF
pin 5 = run;      /* ピン 5 に run を割り付けます */
pin 6 = halt;     /* ピン 6 に halt を割り付けます */
```

`$ENDIF`

## **\$ELSE**

このコマンドは、`$IFDEF` コマンドや`$ENDIF` コマンドで定義される条件コンパイルの反対の条件でコンパイルされるソースステートメントを表わします。以下にフォーマットを示します。

`$ELSE`

`$IFDEF` コマンドや`$IFNDEF` コマンドの条件が真(すなわち、それらに続くステートメントがコンパイルされる)の場合、`$ELSE` コマンドと`$ENDIF` コマンドとの間のソースステートメントは、コンパイルされません。

条件が偽の場合、`$IFDEF` コマンドと`$IFNDEF` コマンドと、`$ELSE` との間のソースステートメントはコンパイルされず、`$ELSE` コマンドと`$ENDIF` コマンドとの間のソースステートメントがコンパイルされます。

以下の例では、Prototype が定義されているので、`$IFDEF` に続くソースステートメントがコンパイルされ、`$ELSE` に続くステートメントはコンパイルされません。

```
$DEFINE Prototype X /* Prototype の定義 */
$IFDEF Prototype
pin 1 = memreq; /* プロとタイプのピン 1 に */
          /* メモリリクエストを割り付ける */
pin 2 = ioreq; /* プロトタイプのピン 2 に */
          /* メモリリクエストを割り付ける */
$ELSE
pin 1 = ioreq; /* PCB のピン 1 に */
          /* I/O リクエストを割り付ける */
pin 2 = memreq; /* PCB のピン 2 に */
          /* I/O リクエストを割り付ける */
$ENDIF
```

`$ELSE` 以下のステートメントをコンパイルするには、Prototype の`$DEFINE` 文を削除して下さい。

## **\$REPEAT**

このコマンドはC言語のFOR文やFORTRAN言語のDO文に似ています。このコマンドにより、インデックスの回数だけステートメントが繰り返されます。フォーマットを以下に示します。

```
$REPEAT index=[number1,number2,_numbern]
repeat body
$REPEND
```

ここで、n は 0 から 1023 までの任意の数値です。

プリプロセッサにより、繰り返しの本体は、number1 から numbern まで複写されます。インデックス番号は、番号が連続している場合、[number1..numbern]のように省略して記述することができます。繰り返しの

本体は、任意の CUPL ステートメントです。算術演算を繰り返し本体の中で使用できます。算術表現は中括弧{}で囲む必要があります。

例えば、3 8デコーダを設計すると、

```
FIELD sel = [in2..0]
$REPEAT i = [0..7]
    !out{i} = sel:'h'{i} & enable;
$REPEND
```

ここで、インデックス変数 i は 0 から 7 までの値になり、  
out{i}=sel:h{i}&enable;がプリプロセッサにより以下のように展開されます。

```
FIELD sel = [in2..0];
!out0 = sel:'h'0 & enable;
!out1 = sel:'h'1 & enable;
!out2 = sel:'h'2 & enable;
!out3 = sel:'h'3 & enable;
!out4 = sel:'h'4 & enable;
!out5 = sel:'h'5 & enable;
!out6 = sel:'h'6 & enable;
!out7 = sel:'h'7 & enable;
```

以下の例では、加算(+)演算子や商余(%)演算子が繰り返し本体でどのように使われるかを示します。

```
/* 制御信号 advance を用いた 5 ビットのカウンタです。 */
/* advance がハイの場合、カウンタは1 ずつ増えます。 */
FIELD count[out4..0]
SEQUENCE count {
    $REPEAT i = [0..31]
        PRESENT S{i}
    IF advance & !reset NEXT
    S{(i+1)%(32)};
    IF reset NEXT S{0};
    DEFAULT NEXT S{i};
$REPEND
}
```

## **\$REPEND**

このコマンドは、\$REPEAT で始まる繰り返しの反対を終わりを表わします。フォーマットを以下に示します。

```
$REPEND
```

\$REPEND コマンドに続くステートメントは前の\$REPEAT コマンドのステートメントと同様にコンパイルされます。\$REPEAT コマンドのそれぞれに対応した\$REPEND コマンドが必要です。

## **\$MACRO**

このコマンドにより、ユーザ定義のマクロを作成します。フォーマットを



以下に示します。

```
$MACRO name argument1
argument2...argumentn
macro function body
$MEND
```

マクロ関数本体はマクロ名が呼び出されるまでコンパイルされません。ファンクション名が記述され関数が呼び出されると、関数へパラメータが渡されます。

\$REPEAT コマンドのように、マクロ関数の本体で算術演算を行なうことができます。算術演算は中括弧({})で囲む必要があります。

以下の例で、\$MACRO コマンドの使い方を説明します。

```
$MACRO decoder bits MY_X MY_Y MY_enable;
FIELD select = [MY_Y{bits-1}..0];
$REPEAT i = [0..{2** (bits-1)}]
!MY_X{i} = select:'h'{i} & MY_enable;
$REPEND
$MEND
_/* Other statements */
decoder(3, out, in, enable); /*macro function call*/
```

関数 decoder をコールすると、マクロ展開により以下のステートメントが作成されます。

```
FIELD sel = [in2..0];
!out0 = sel:'h'0 & enable;
!out1 = sel:'h'1 & enable;
!out2 = sel:'h'2 & enable;
!out3 = sel:'h'3 & enable;
!out4 = sel:'h'4 & enable;
!out5 = sel:'h'5 & enable;
!out6 = sel:'h'6 & enable;
!out7 = sel:'h'7 & enable;
```

マクロが呼び出されると、キーワード NC を使用して no connection が表わされます。NC はキーワードのため、文字 NC は CUPL では使用できません。

マクロ展開ファイルは、PLD ファイルをコンパイルする時に Expand Macro MX オプションをイネーブルにすると作成されます。ファイルの中のマクロ関数をすべて見ることができます。以下の例では、別々のファイルに記述されたマクロ定義と呼び出し命令を示します。

デコーダのマクロ定義は、macrolib.m に保存されています。

```
$INCLUDE macrolib.m /* マクロライブラリの指定 */
... /* その他の命令 */
decoder(4, out, in enable);
... /* その他の命令 */
```

アドバンスド PLD パッケージの例題のファイルの中には、他の例もありま

す。

## \$MEND

このコマンドは、\$MACRO で始まるマクロ関数本体の終わりを表わすコマンドです。フォーマットを以下に示します。

\$MEND

\$MEND コマンドに続く命令は、前述の\$MACRO コマンドと同じようにコンパイルされます。各\$MACRO コマンドに対して、対応する\$MEND コマンドが必要です。

## 言語シンタックス

### 論理演算

CUPL はブール表現に使用される 4 つの標準的な演算子をサポートしています。表 9-8 にこれらの演算子と優先順位を高い方から低い方へ示します。

演算子	使用例	説明	優先順位
!	!A	NOT	1
&	A & B	AND	2
#	A # B	OR	3
\$	A \$ B	XO	4

表 9-8 論理演算子の優先順位

図 9-5 の真理値表に各演算子のブーリアン論理規則を示します。

NOT : ones complement !		AND &		
A	!A	A	B	A & B
0	1	0	0	0
1	0	0	1	0
		1	0	0
		1	1	1
OR #		XOR : exclusive OR \$		
A	B	A	B	A \$ B
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	1	1	0

図 9-5 真理値表

## 算術演算

CUPL は、算術表現で使用される 6 つの標準的な算術演算子をサポートしています。算術表現は\$REPEAT コマンドまたは\$MACRO コマンドで使用できます。その他では使用できません。算術表現は中括弧({})で囲む必要があります。表 9-9 に、これらの演算子と優先順位を高い方から順に示します。

演算子	使用例	説明	優先順位
**	2**3	べき乗	1
*	2*i	乗算	2
/	4/2	除算	2
%	9%8	商余	2
+	2+4	加算	3
-	4-i	減算	3

表 9-9 算術演算の優先順位

## 算術関数

CUPL は、算術表現で使用される算術関数を一つサポートしています。算術表現は、\$REPEAT コマンドと\$MACRO コマンドで使用できます。その他では、使用できません。表 9-10 に関数を示します。

関数	基数
LOG2	2 進数
LOG8	8 進数

LOG16	16 進数
LOG	10 進数

表 9-10 算術関数

LOG 関数は整数値を返します。例えば、

```
LOG2(32) = 5 <==> 2**5 = 32
LOG2(33) = ceil(5.0444) = 6 <==> 2**6 = 64
```

ceil(x)は、x を越えない最大の整数を返します。

## 拡張子

プログラマブルデバイス内部のメジャーノードに関連付けられる特定の関数（例えば、フリップフロップやプログラマブルスリーステートイネーブルなど）を表わすために、拡張子を変数名に付けることができます。表 9-11 に CUPL でサポートされる拡張子とイコールサイン(=)のどちらで使われるかを示します。コンパイラは拡張子の使われかたや、指定されたデバイスで有効かどうか、またその使われかたが他の拡張子と矛盾しないかを確認します。

拡張子	Side	説明
.AP	L	フリップフロップの非同期プリセット
.AR	L	フリップフロップの同期リセット
.APMUX	L	マルチプレクサの選択の非同期プリセット
.ARMUX	L	マルチプレクサの選択の非同期リセット
.BYP	L	プログラマブルレジスタバイパス
.CA	L	コンプリメントアレイ
.CE	L	イネーブルの D-CE タイプのフリップフロップの CE 出力
.CK	L	フリップフロップのプログラマブルクロック
.CKMUX	L	マルチプレクサのクロックセレクション
.D	L	D-タイプのフリップフロップの D 出力
.DFB	R	D レジスタードフィードバックパスの選択
.DQ	R	D-タイプのフリップフロップの Q 出力
.IMUX	L	2 つのピンの入力マルチプレクサの選択
.INT	R	レジスタードマクロセルの内部フィードバック経路
.IO	R	P ピンフィードバック経路の選択
.IOAR	L	ピンフィードバックレジスタの非同期リセット
.IOAP	L	ピンフィードバックレジスタの非同期プリセット
.IOCK	L	ピンフィードバックレジスタのクロック
.IOD	R	D レジスタによるピンフィードバック経路
.IOL	R	ラッチによるピンフィードバック経路
.IOSP	L	ピンフィードバックレジスタの同期プリセット

.IOSR	L	ピンフィードバックレジスタの同期リセット
.J	L	JK-タイプ出力フリップフロップの J 入力
.K	L	JK-タイプ出力フリップフロップの K 入力
.L	L	ラッチの D 入力
.LE	L	P プログラマブルラッチイネーブル
.LEMUX	L	ラッチイネーブルのマルチプレクサの選択
.LFB	R	ラッチされたフィードバック経路の選択
.LQ	R	入力ラッチの Q 出力
.OBS	L	埋めこみノードのプログラムできる可観測性
.OE	L	プログラマブル出力イネーブル
.OEMUX	L	トリ - ステートマルチプレクサ選択
.PR	L	プログラマブルプリロード
.R	L	SR-タイプ出力フリップフロップの R 入力
.S	L	SR-タイプ出力フリップフロップの S 入力
.SP	L	フリップフロップの同期プリセット
.SR	L	フリップフロップの同期リセット
.T	L	トグル出力フリップフロップの T 入力
.TEC	L	ヒューズの選択
.TFB	R	T レジスタードフィードバック経路の選択
.T1	L	2-T フリップフロップの T1 入力
.T2	L	2-T フリップフロップの T2 入力

Table 9-11. 拡張子

各拡張子を使用して、特定の関数にアクセスできます。例えば、出力イネーブルの式を指定する場合（デバイスがこの機能を持っている場合）、拡張子.OE を使用して下さい。式は以下のようになります。

```
PIN 2 = A;
PIN 3 = B;
PIN 4 = C;
PIN 15 = VARNAME;
VARNAME.OE = A&B;
```

デバイス内で、物理的に実行されるフリップフロップだけをコンパイラはサポートします。例えば、コンパイラはDタイプのレジスタしか持たないデバイスでJKタイプのフリップフロップをエミュレートしません。デバイスが持たない機能を使用するとコンパイラはエラーを発生します。

プログラマブル出力イネーブルにより双方向 I/O ピンが設定されているデバイスでは、コンパイラはピンの使われかたに応じた出力イネーブル表現を生成します。変数名が式の左辺で使用される場合、ピンは出力と見なされ 2 進値 1 が割り付けられます。すなわち、出力イネーブル表現はデフォルトで以下のようになります。

```
PIN_NAME.OE = 'b'1; /* トリステートバッファ */
/* 常時 ON */
```

入力としてだけ使用される（すなわち、式の右辺で使用される変数名）これらのピンには、2 進値 0 が割り付けられます。出力イネーブル表現のデフ

ォルトを以下に示します。

```
PIN_NAME.OE = 'b'0;      /* トリステートバッファ */
                          /* 常時 OFF */
```

I/O ピンが入出力として使用される場合、ユーザが新しく指定する出力イネーブル表現がデフォルトのピンにオーバーライドされ、必要な時にトリステートバッファをイネーブルにします。

JK-タイプまたはSR-タイプのフリップフロップを使用する場合、J 入力と K 入力（または S 入力と R 入力）の両方に式を記述する必要があります。入力のどちらかに式を書く必要がない場合、以下のステートメントを使用して入力ピンを OFF にして下さい。

```
COUNT0.J='b'0 ; /* J input not used*/
```

非同期リセットや非同期プリセット等のコントロール関数は通常デバイスのレジスタのグループ（または全部）に接続されます。コンパイラにより、これらのグループの各レジスタにコントロール関数が接続されているかが検査されます。そして、固有の式を持たないグループの構成要素に対してワーニングメッセージを作成します。あるグループのコントロール関数がすべて違う式により定義されている場合、どの式が正しいものか判断できないので、コンパイラはエラーを生成します。このようなことはデバイス特有の問題です。従って、デバイスの機能を理解することは大切なことです。

図 9-6 に拡張子の使用例を示します。この図は実際の回路を表わしていません。拡張子を使用して回路の別々の関数に式を記述する方法を示します。

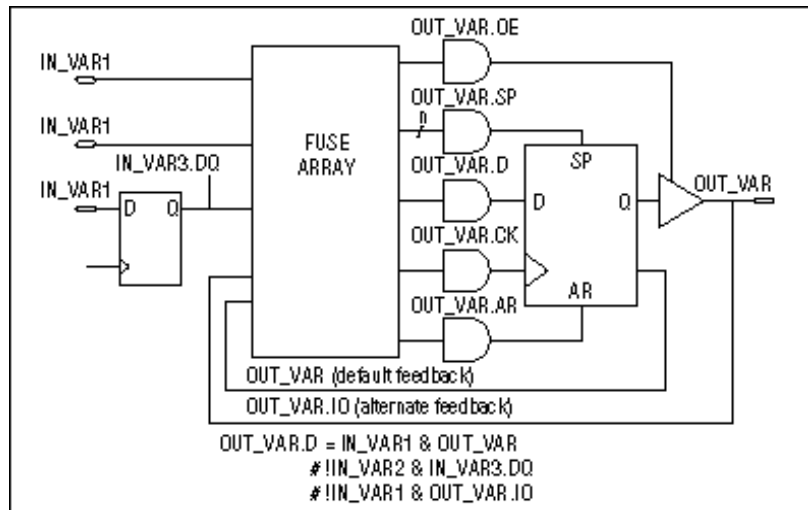


図 9-6 拡張子を説明する回路

レジスタード出力として指定された.D 拡張子の式を図に示します。フィードバック（OUT\_VAR）が式の中で使用される場合、.D 出力を使用しません。

.DQ 拡張子は入力ピンにだけ使用されます。

他のタイプのコントロールやコントロールポイントを指定するためにさらに式が記述されます。例えば、出力イネーブル用の式は以下の様に記述されます。

$$\text{OUT\_VAR.OE} = \text{IN\_VAR1} \# \text{IN\_VAR2}$$

## フィードバック拡張子の使用法

デバイスの中には、フィードバック経路をプログラムすることができるものもあります。例えば、EP300 にはフィードバック経路を内部フィードバックとレジスタードフィードバック、ピンフィードバックの内から選択できるように各出力にマルチプレクサがあります。

図 9-7 に EP300 のプログラマブルフィードバック機能を示します。

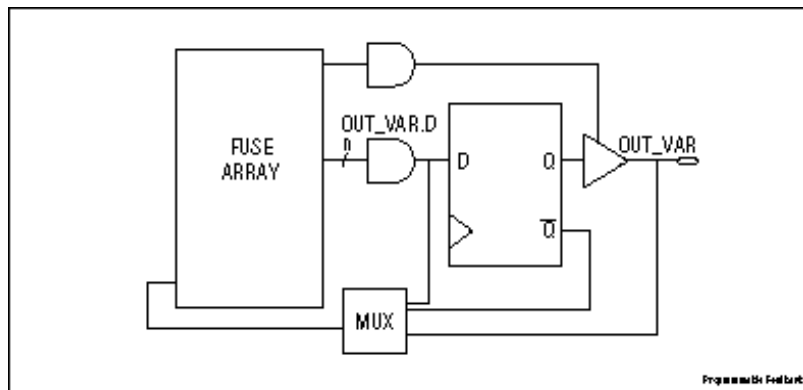


図 9-7 プログラマブルフィードバック

コンパイラにより、出力ピンの使われかたに応じて自動的にデフォルトのフィードバックが選択されます。例えば、出力がレジスタード出力として使用される場合、デフォルトのフィードバック経路は、図 9-6 のようにレジスタを介されます。このデフォルトはフィードバック変数に拡張子を追加するとオーバーライドできます。例えば、レジスタード出力のフィードバック変数に .IO 拡張子を追加するとコンパイラはピンフィードバック経路を選択します。

図 9-8 にピンフィードバックを使用したレジスタード出力を示します。

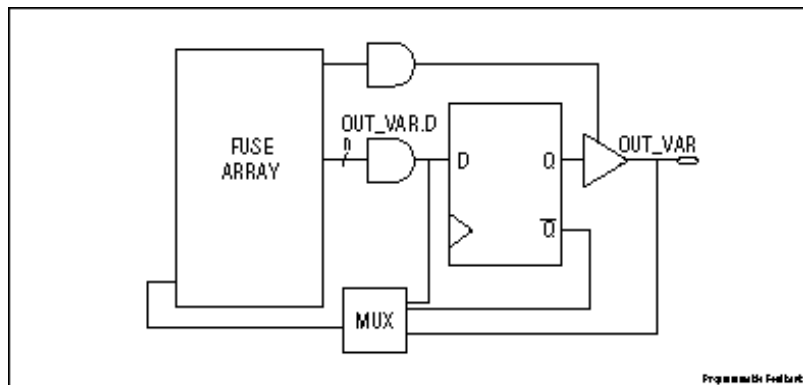


図 9-8 プログラマブルピン(I/O) フィードバック

図 9-9 にレジスタードフィードバックを使用した組み合わせ出力を示します。

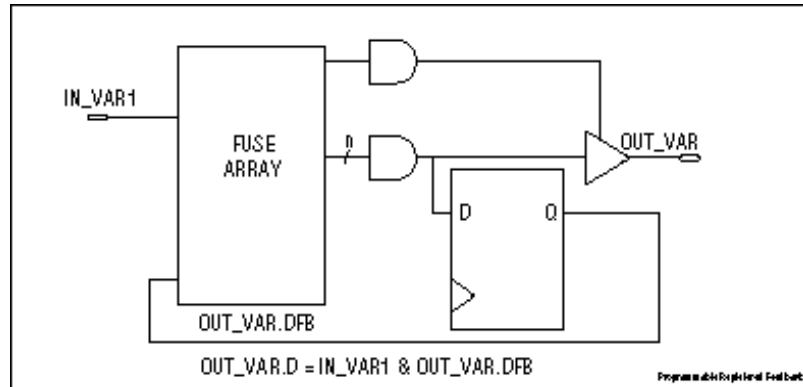


図 9-9 プログラマブルレジスタードフィードバック

図 9-10 に内部フィードバックを使用した組み合わせ出力を示します。

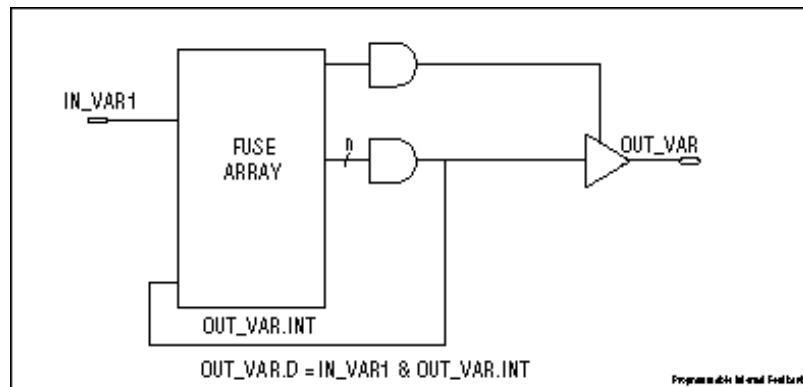


図 9-10 プログラマブル内部フィードバック

## マルチプレクサ拡張子の使用法

デバイスの中には、プログラマブルとコモンのコントロール関数を選択できるものがあります。例えば、P29MA16 では各出力に、コモンとプロダクトタームクロック、出力イネーブルを選択するマルチプレクサが付いています。

図 9-11 に、P29MA16 のプログラマブルクロックと出力イネーブル機能を示します。



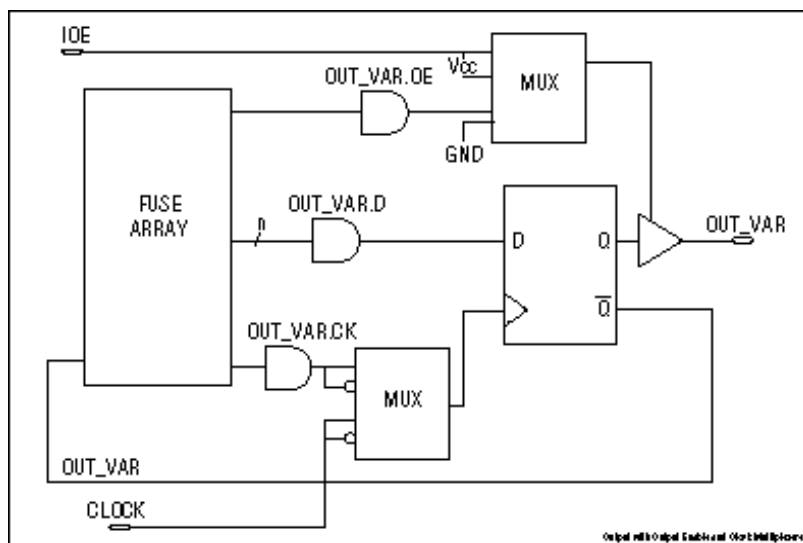


図 9-11 出力イネーブルとクロックのマルチプレクサのある出力

式が.OE や.CK 拡張子で記述されている場合、マルチプレクサ出力はプロダクトタームと出力イネーブル、クロックの中から選択されます。式が.OEMUX や.CKMUX 拡張子で記述されている場合、マルチプレクサ出力は、コモンと出力イネーブル、クロックの中から選択されます。

.OEMUX と.CKMUX 拡張子で記述された式は一つの変数しか持つ事ができません。そして、否定演算子!で動作します。これは、それらの入力ヒューズアレイからではなく、クロックピンなどのコモンソースから入力されるためです。このようなことは、入力をヒューズアレイから取得する.OE や.CK 拡張子で書かれた式とは反対です。

図 9-12 に、VCC と出力イネーブル、反転されたコモンクロックピンから選択されたクロックマルチプレクサ出力から選択された出力イネーブルマルチプレクサ出力のレジスタード出力を示します。

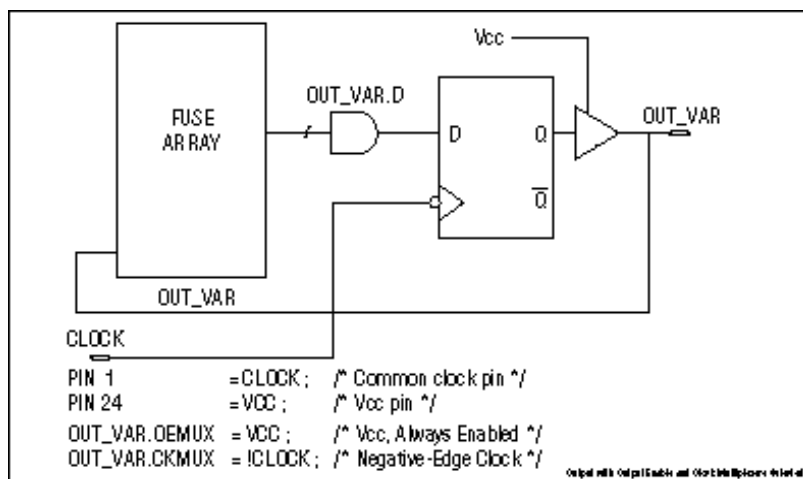


図 9-12 出力イネーブルとクロックマルチプレクサの中から選択された出力

.OE や.OEMUX 拡張子の式はお互いに排他的な関係にあります。すなわち、各出力にはどちらか一つを記述できます。同様に、.CK や.CKMUX 拡張子の式は互いに排他的です。

## 拡張子の使用

このセクションでは

### .AP 拡張子

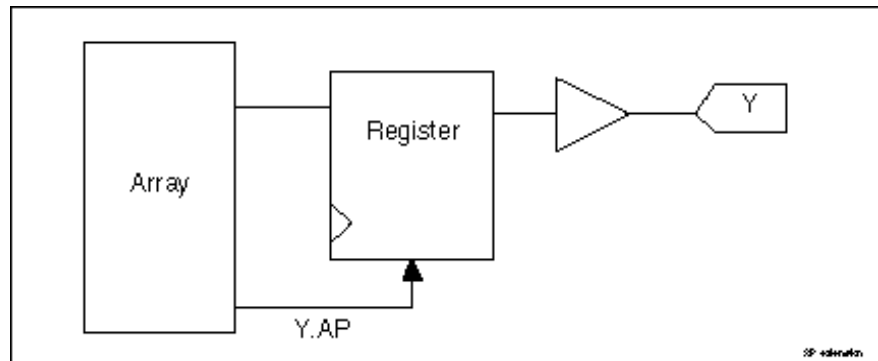


図 9-13 .AP 拡張子

.AP 拡張子を使用して、レジスタの非同期プリセットを設定します。例えば、等式  $Y.AP=A\&B$ ;によりレジスタは、A と B が論理的に真の場合非同期でプリセットされます。

### .APMUX 拡張子

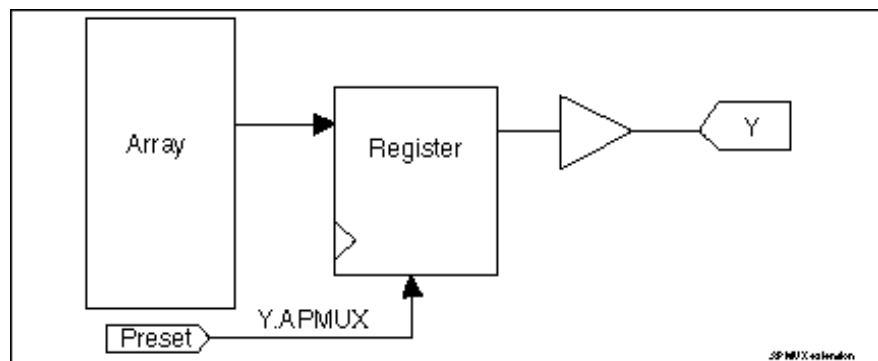


図 9-14 .APMUX 拡張子

デバイスの中には、ピンの中の一つに接続された非同期プリセットをイネーブルするマルチプレクサを持つものがあります。.APMUX 拡張子を使用して非同期プリセットとピンを直接接続することができます。

### .AR 拡張子

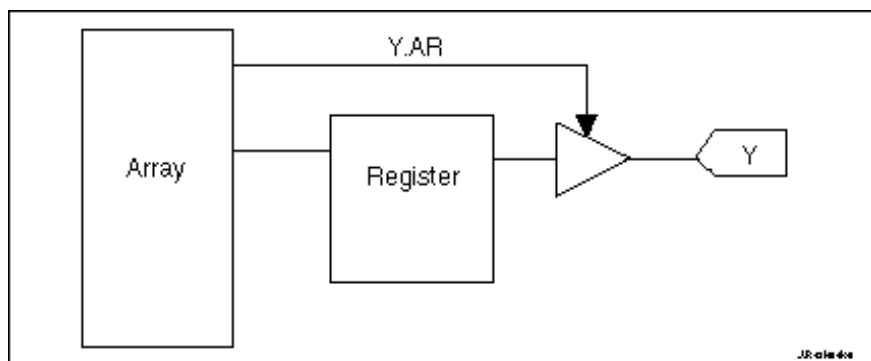


図 9-15 .AR 拡張子

.AR 拡張子を使用して、レジスタの非同期リセットの式を定義します。レジスタの非同期リセットに接続された複数のプロダクトタームを持つデバイスで使用して下さい。

#### .ARMUX 拡張子

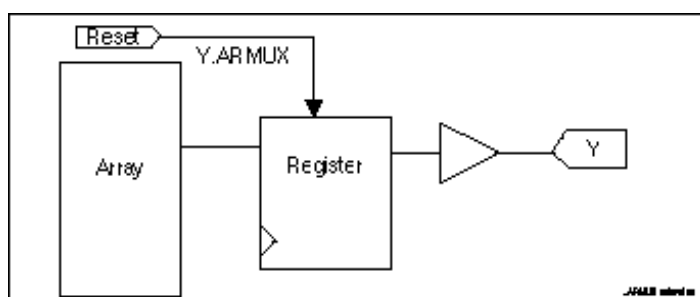


図 9-16 .ARMUX 拡張子

レジスタの非同期リセットと複数のピンを直接接続するマルチプレクサを持つデバイスでは、.ARMUX 拡張子を使用して接続を行なって下さい。デバイスは、ピンまたはプロダクトタームのどちらかに接続される非同期リセットを持つことができます。この場合、.AR 拡張子はプロダクトタームの接続を選択するために使用されます。ところが、.ARMUX 拡張子はピンに接続するために使用します。

#### .CA 拡張子

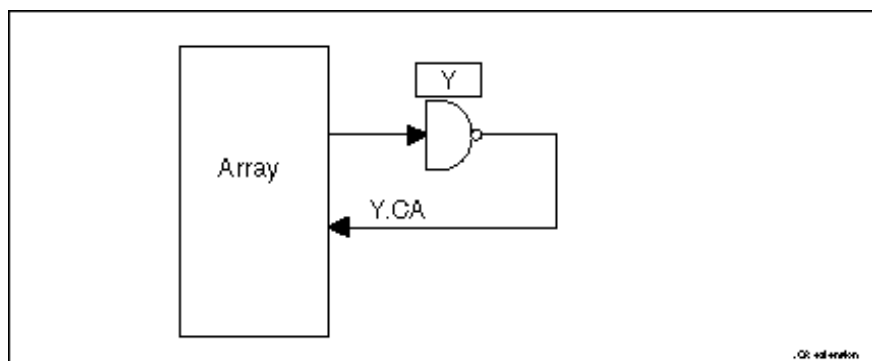


図 9-17 .CA 拡張子

.CA 拡張子は D-CE レジスタで使用されます。これにより、レジスタの CE へ入力を指定することができます。

#### .CE 拡張子

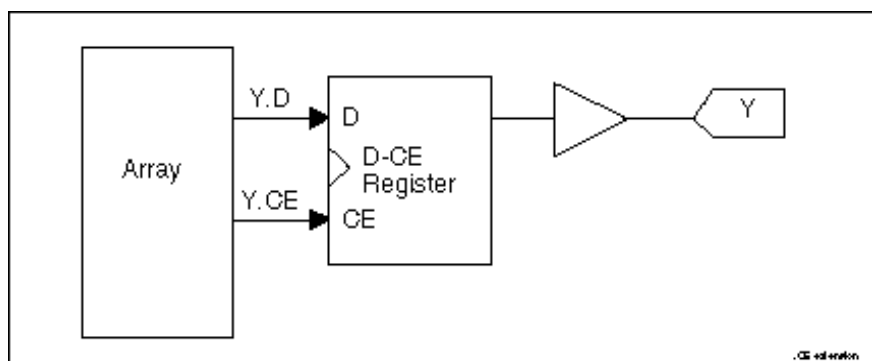


図 9-18 .CE 拡張子

CE 拡張子は D-CE レジスタを持ち、CE タームが使用されないデバイスの場合、レジスタが D レジスタと同じ動作をするように、それらのレジスタには 2 進数の 1 が設定されます。CE タームのイネーブルに失敗すると値が全く変わらない D レジスタになります。

#### .CK 拡張子

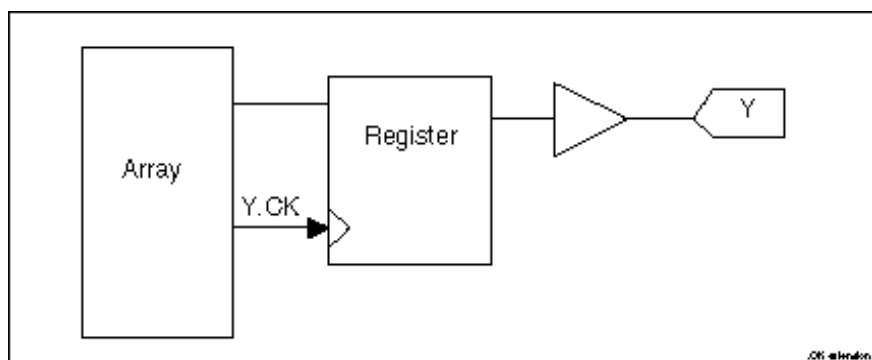


図 9-19 .CK 拡張子

.CK 拡張子は、クロックで駆動されるプロダクトタームを選択するために使用します。デバイスの中には、レジスタのクロックを複数のピンまたはプロダクトタームに接続できるものがあります。.CK 拡張子によりプロダクトタームが選択されます。クロックを直接ピンに接続するには、.CKMUX 拡張子を使用して下さい。

#### .CKMUX 拡張子

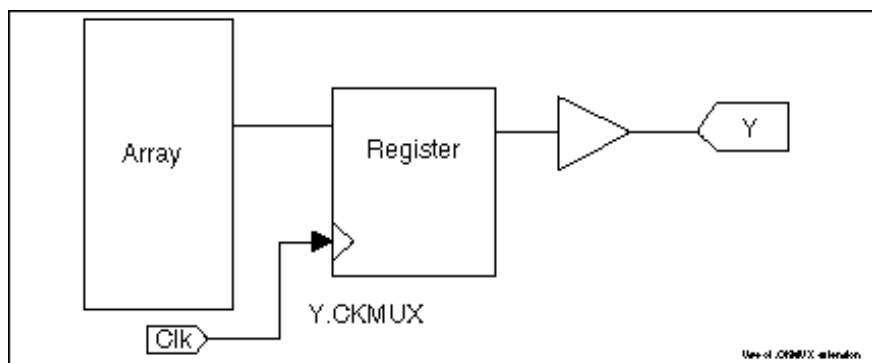


図 9-20 .CKMUX 拡張子

.CKMUX 拡張子は、レジスタのクロック入力をピンに接続するために使用されます。このような機能は、クロックをピンに接続するマルチプレクサを持つデバイスに必要です。これは、クロックが任意のピンと接続できるという意味ではありません。一般に、マルチプレクサにより、クロックは2つの内の一つに接続することができます。デバイスの中には、4つの内の一つに接続するマルチプレクサを持つデバイスもあります。

#### .D 拡張子

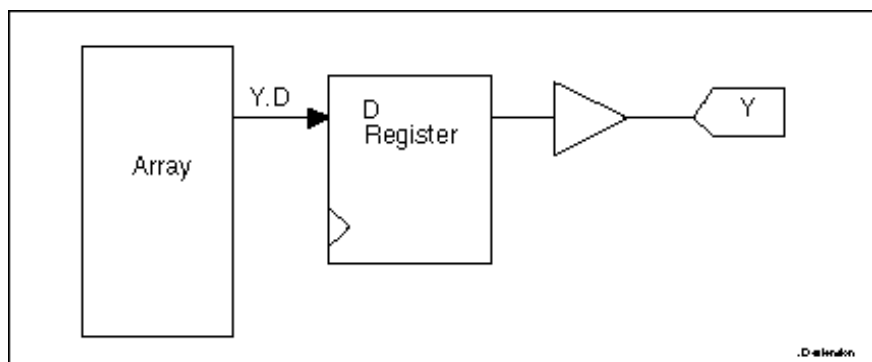


図 9-21 .D 拡張子

.D 拡張子を使用して D レジスタへの D 入力を指定して下さい。.D レジスタを使用するとコンパイラは、プログラマブルマクロセルを D レジスタとして配置します。D レジスタ出力しか持たない出力では、.D 拡張子を使用しないで下さい。.D 拡張子が D レジスタを持たない出力に使用される場合、

コンパイラはエラーを発生します。

#### .DFB 拡張子

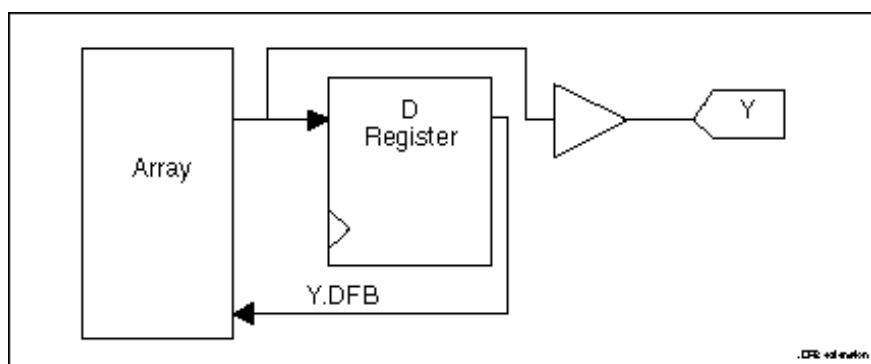


図 1-22 .DFB 拡張子

.DFB 拡張子はプログラマブル出力マクロセルが組み合わせて使用され、しかも D レジスタが出力と接続されたままであるような特別な場合に使用されます。.DFB 拡張子によりレジスタからフィードバックをすることができます。普通の状態では、出力がレジスタを介して配置されると、拡張子を指定しないとレジスタからのフィードバックが選択されます。

#### .DQ 拡張子

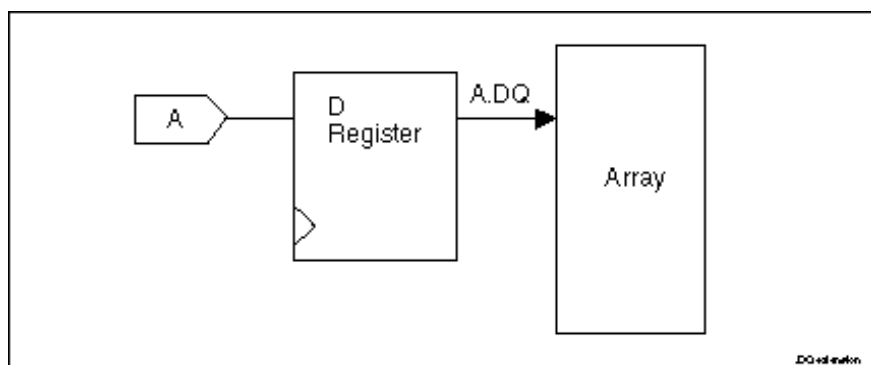


図 1-23 .DQ 拡張子

.DQ 拡張子を使用して入力 D レジスタを指定します。.DQ 拡張子の使用により、入力はレジスタを介されます。.DQ 拡張子を使用しても出力 D レジスタから Q 出力を指定することはできません。

#### .IMUX 拡張子

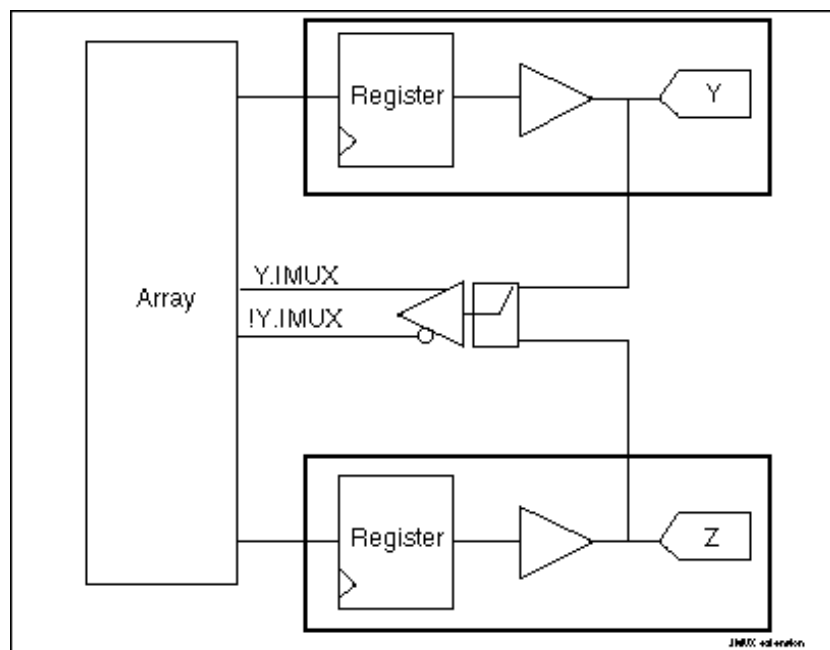


図 9-24 .IMUX 拡張子

.IMUX 拡張子は先進的な拡張子で、これにより、フィードバック経路を選択することができます。マルチプレクサに接続された 2 つの I/O ピンからのピンフィードバックがあるデバイスで使用する下さい。ピンの一つをフィードバック経路に使用します。

#### .INT 拡張子

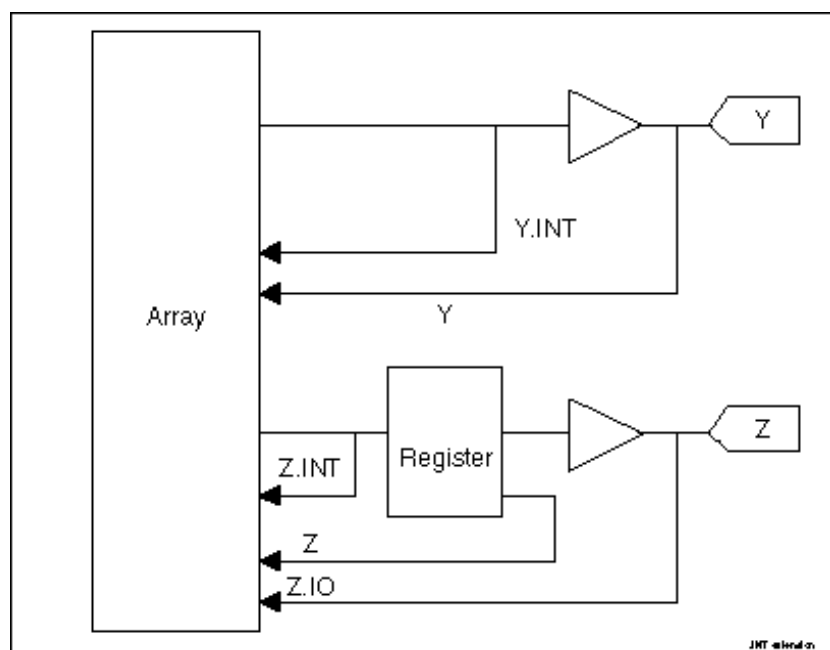


図 9-25 .INT 拡張子

.INT 拡張子は、内部フィードバック経路を選択するために使用されます。これは、組み合わせ出力またはレジスタード出力に使用して下さい。.INT 拡張子により組み合わせフィードバックを行なうことができます。

.IO 拡張子

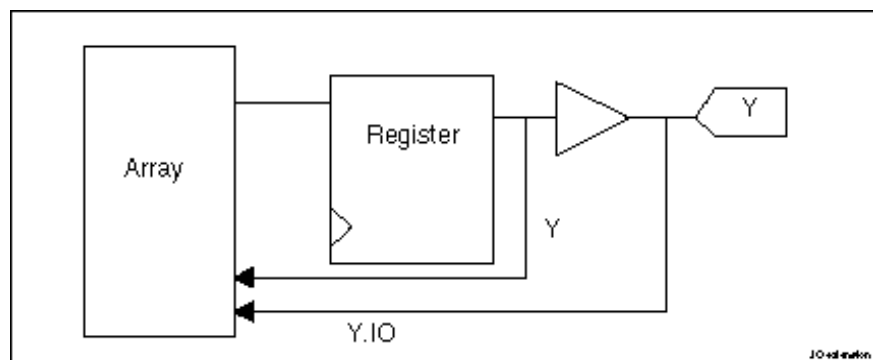


図 9-26 .IO 拡張子

マクロセルがレジスターを介して配置される場合、.IO 拡張子を使用して、ピンフィードバックを選択して下さい。

.IOAP 拡張子

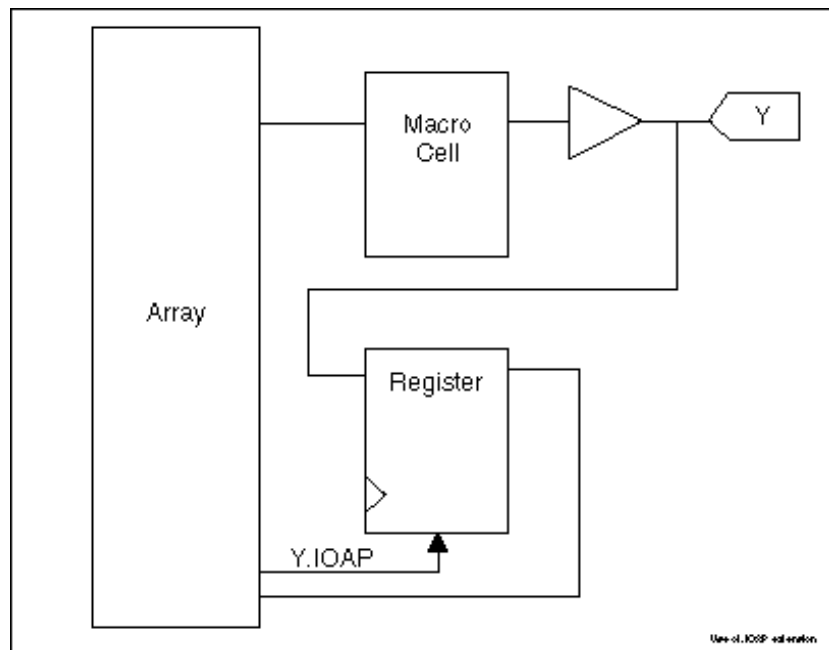


図 9-27 .IOAP 拡張子

出力マクロセルからのレジスタードピンフィードバックがある場合、.IOAP 拡張子を使用して、非同期プリセットの式を記述して下さい。



## .IOAR 拡張子

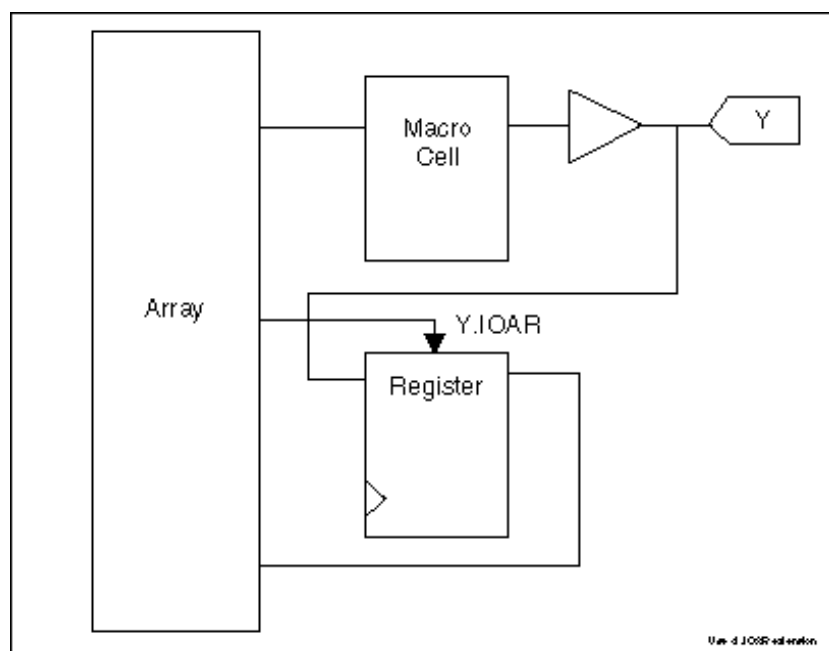


図 9-28 .IOAR 拡張子

出力マクロセルからのレジスタードピンフィードバックがある場合、.IOAR 拡張子を使用して、非同期リセットの式を記述して下さい。

## .IOCK 拡張子

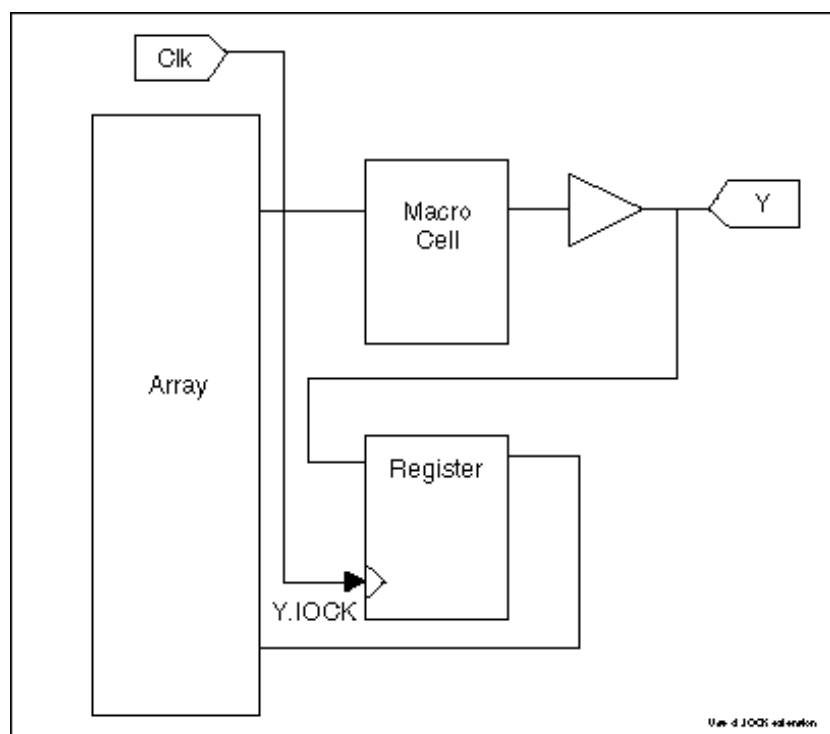


図 9-29 .ILOCK 拡張子

.ILOCK 拡張子を使用して、出力マクロセルに接続されるレジスタードピンフィードバックのクロックを記述して下さい。

.IOD 拡張子

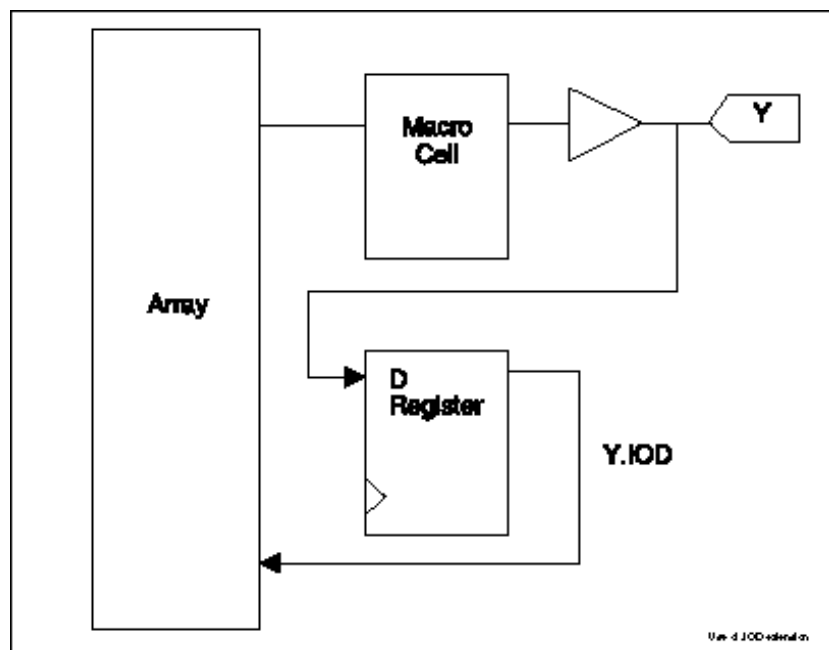


図 9-30 .IOD 拡張子

.IOD 拡張子を使用して、ピンフィードバック経路による出力マクロセルに接続されるレジスタからのフィードバックを指定して下さい。

.IOL 拡張子

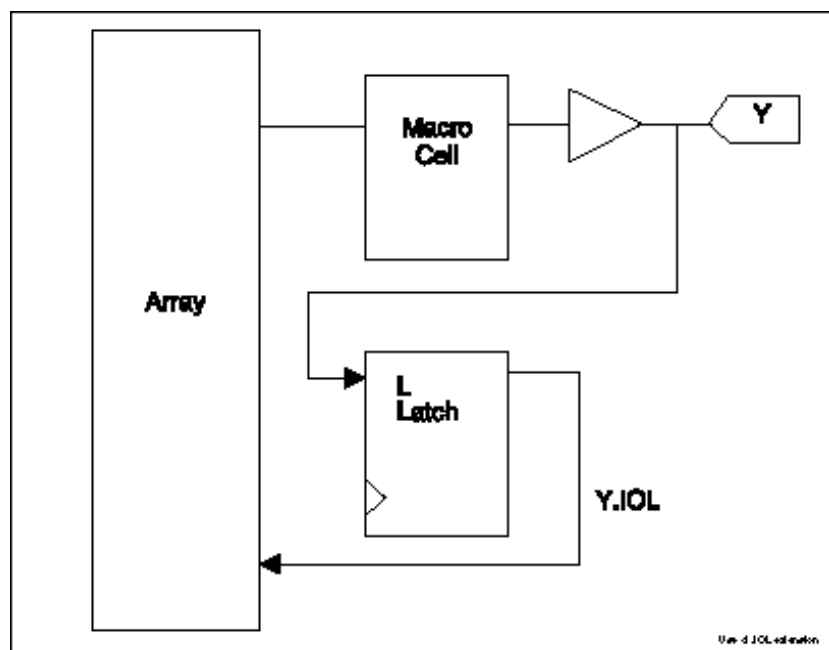


図 9-31 .IOL 拡張子

.IOL 拡張子を使用して、ピンフィードバック経路による出力マクロセルに接続される埋めこみラッチからのフィードバックを指定して下さい。

.IOSP 拡張子

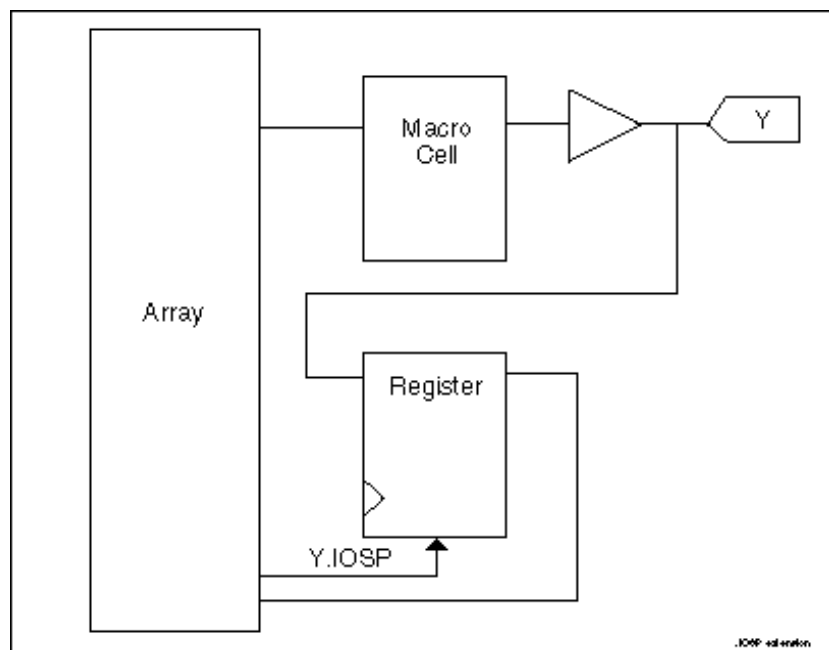


図 9-32 .IOSP 拡張子

.IOSP 拡張子を使用して、出力マクロ競るからのレジスタードピンフィードバックがある場合の、同期プリセットの式を記述して下さい。

.IOSR 拡張子

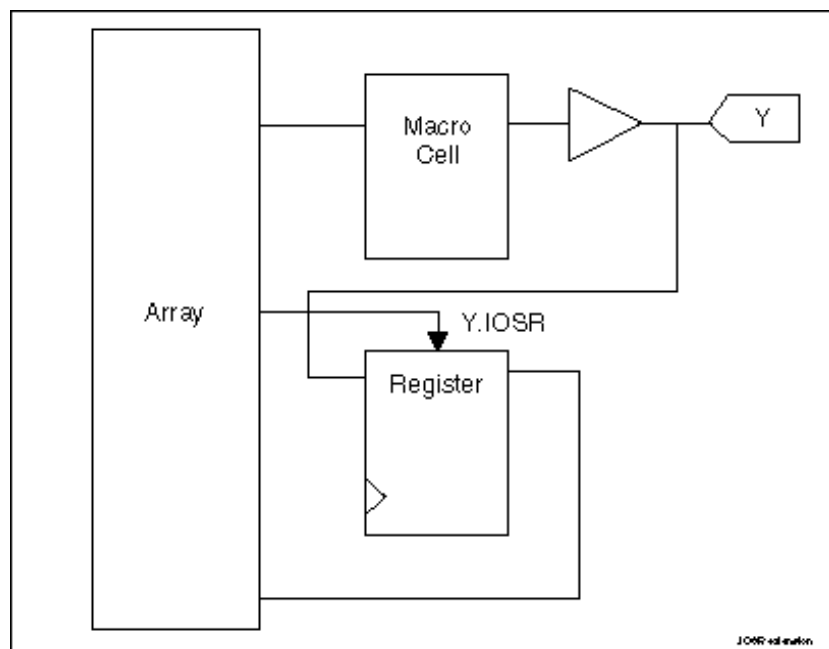


図 9-33 IOSR 拡張子

.IOSR 拡張子を使用して、出力マクロ競るからのレジスタドピンフィードバックがある場合の、同期リセットの式を記述して下さい。

#### .J,.K 拡張子

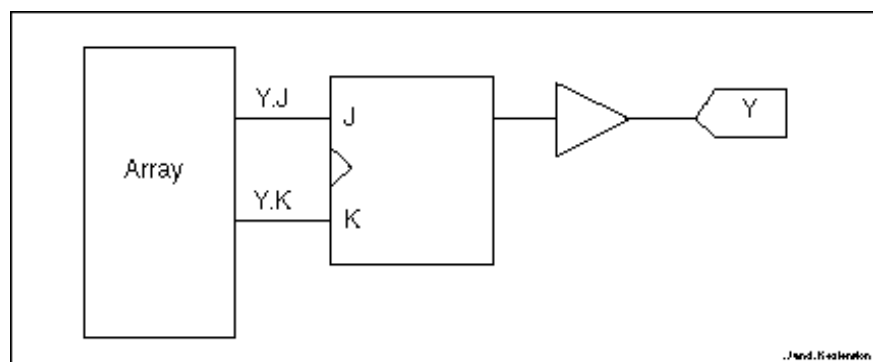


図.J,.K 拡張子

.J や.K 拡張子を使用して、JK レジスタへの J 入力や K 入力を記述して下さい。マクロセルがプログラム可能な場合、.J 拡張子や.K 拡張子を使用すると、コンパイラは JK 出力として配置します。J と K の式を両方とも記述する必要があります。入力の片方が記述されない場合、使用しない入力をディスエーブルにするために 2 進数 0 を設定して下さい。

#### .L 拡張子

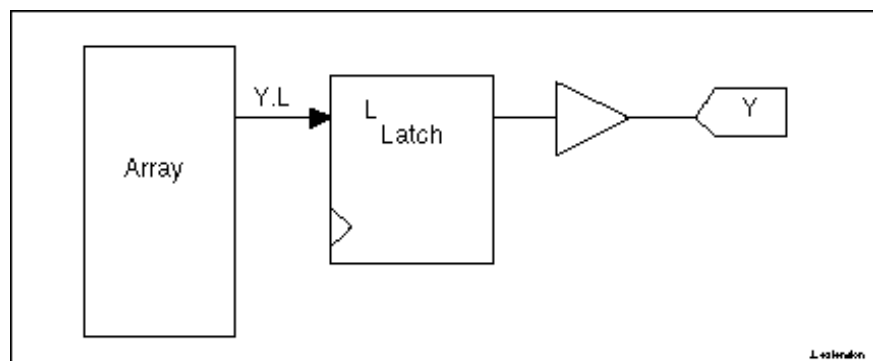


図 9-35 .L 拡張子

.L 拡張子を使用して、ラッチへの入力を指定して下さい。プログラム可能なマクロセルを持つデバイスでは、.L 拡張子を使用するとコンパイラは、マクロセルをラッチ出力として配置します。

#### .LE 拡張子

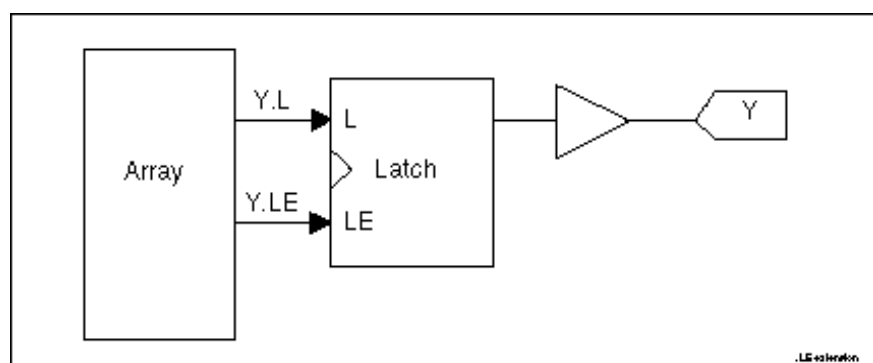


図 9-36 .LE 拡張子

.LE 拡張子を使用して、ラッチのラッチイネーブルの式を記述して下さい。.LE 拡張子により、プロダクトタームはラッチイネーブルに接続されます。

#### .LEMUX 拡張子

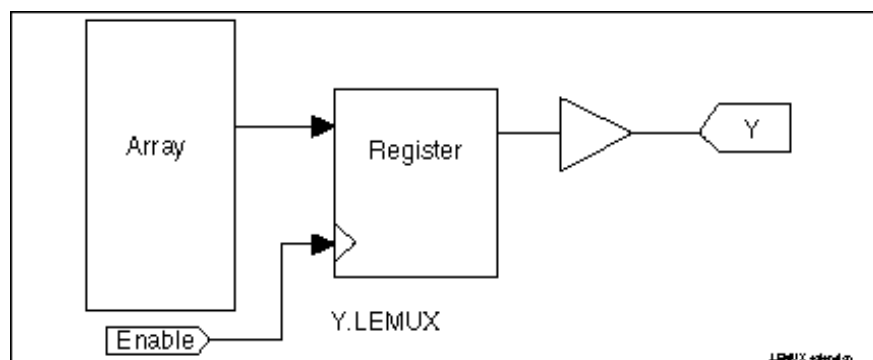


図 9-37 .LEMUX 拡張子

.LIMEX 拡張子を使用して、ラッチイネーブルのピン接続を指定して下さい。

#### .LFB 拡張子

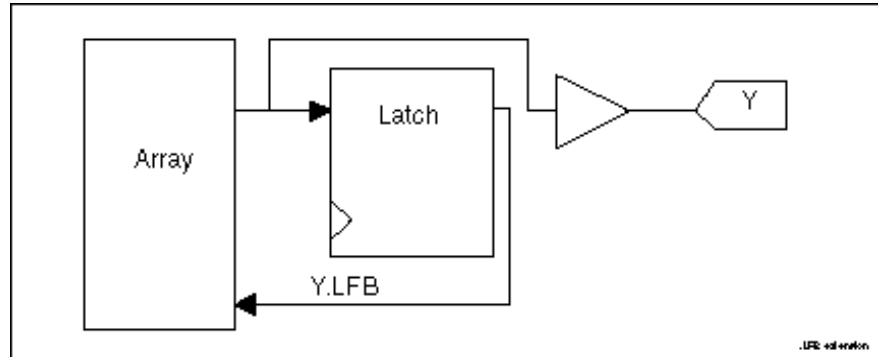


図 9-38 .LFB 拡張子

.LFB 拡張子は、プログラマブル出力マクロセルが組み合わせて配置され、しかもラッチは出力に接続されたままの特殊な状況で使用されます。.LFB 拡張子を使用すると、ラッチからのフィードバックを使用することができます。通常では、出力がラッチに配置されると、ラッチからのフィードバックは、拡張子なしで記述されます。

#### .LQ 拡張子

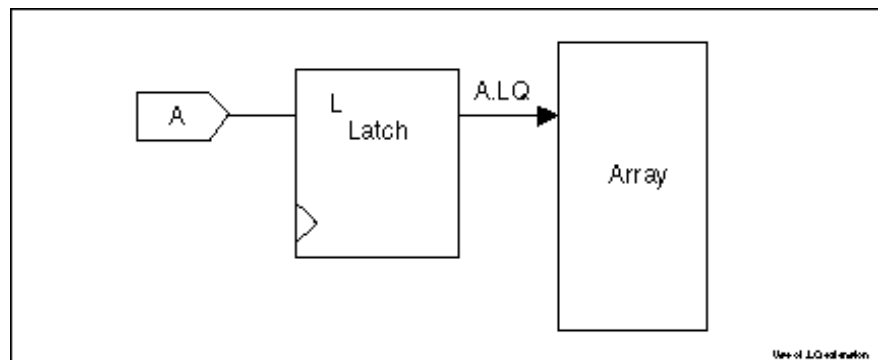


図 9-39 .LQ 拡張子

.LQ 拡張子を使用すると、入力ラッチを指定することができます。.LQ 拡張子を使用すると、入力ラッチとして配置されます。.LQ 拡張子を使用して、出力ラッチからの Q 出力を記述しないで下さい。

#### .OE 拡張子

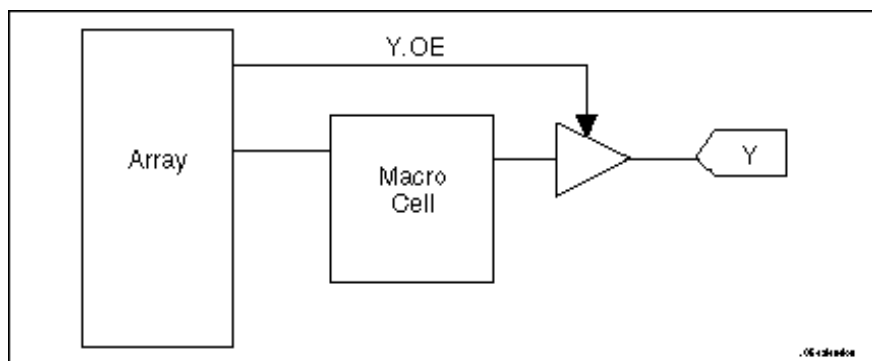


図 9-40 .OE 拡張子

.OE 拡張子を使用して、出力イネーブル信号を駆動するプロダクトタームを記述します。

#### .OEMUX 拡張子

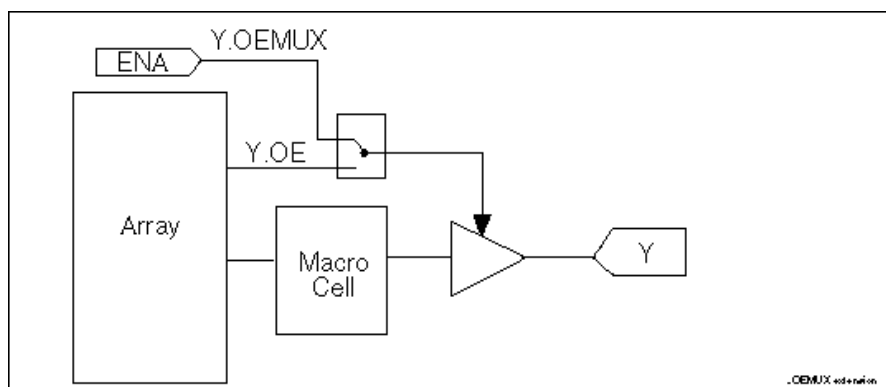


図 9-41 .OEMUX 拡張子

.OEMUX 拡張子を使用して、出力イネーブルをピンのどこかに接続します。デバイスの中には、ピンに出力イネーブルを接続するためのマルチプレクサをもつデバイスがあります。そのような場合にこの拡張子が必要になります。これは、出力イネーブルをピンのどれかに接続するという意味ではありません。一般的には、マルチプレクサにより出力イネーブルは 2 つのうちの一つのピンに接続されます。デバイスの中には、4 つのうちの一つに接続するものもあります。

#### .S、.R 拡張子



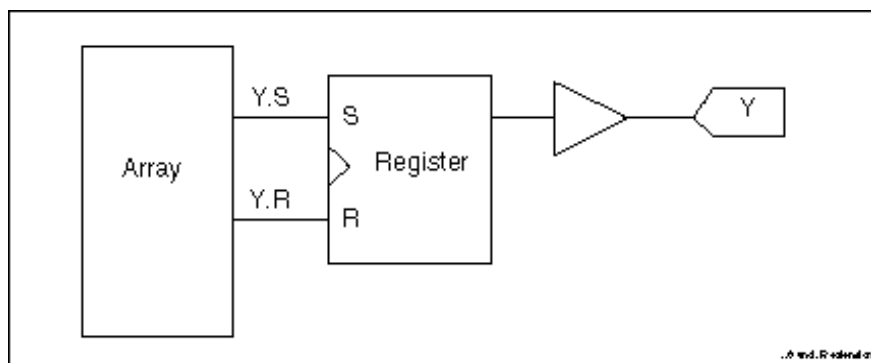


図 9-42 .S、.R 拡張子

.S 拡張子や.R 拡張子を使用して SR レジスタの S 入力や R 入力を指定して下さい。マクロセルがプログラマブルの場合、.S 拡張子や.R 拡張子を使用すると、コンパイラは出力を SR として配置します。S と R の式は両方とも記述する必要があります。入力の片方が使用されない場合、使用されない出力をディスエーブルにするために 2 進数 0 を設定して下さい。

#### .SP 拡張子

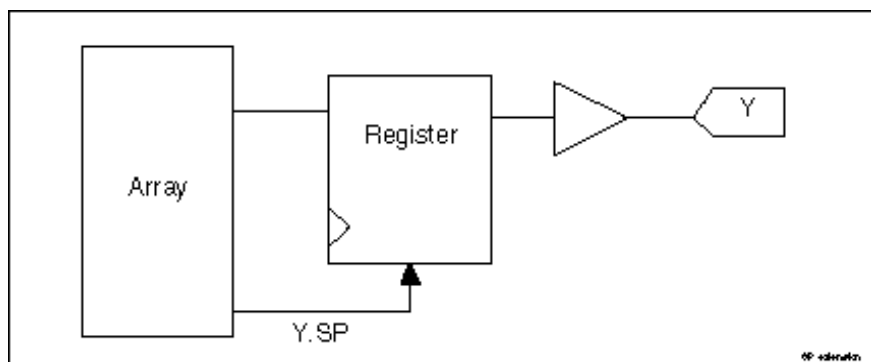


図 9-43 .SP 拡張子

.SP 拡張子を使用して、レジスタの同期プリセットを記述して下さい。例えば、式  $Y.SP = A \& B$ ;により、A と B が論理的に真の場合、出力は同期的にプリセットされます。

#### .SR 拡張子

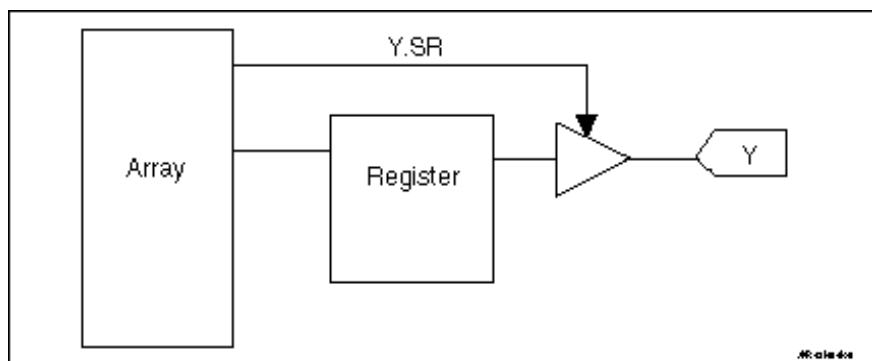


図 9-44 .SR 拡張子

.SR 拡張子を使用して、レジスタの同期リセットを記述して下さい。これは、レジスタの同期リセットに接続される複数のプロダクトタームを持つデバイスで使用されます。

#### .T 拡張子

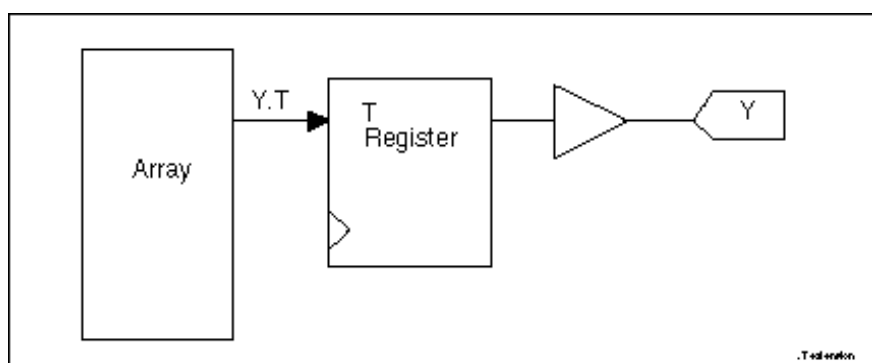


図 9-45 .T 拡張子

.T 拡張子により、T レジスタの T 入力を記述して下さい。 .T 拡張子を使用すると、コンパイラは、マクロセルを T レジスタとして配置します。 T レジスタを持ち、しかもレジスタの前で極性をプログラムできるデバイスは特に注意が必要です。入ってくる信号が真の場合 T レジスタはトグルするので、極性を変えられて信号がレジスタに到達する前に信号が反転されるとレジスタの状態が変化します。 T レジスタを使用するピンは常にアクティブハイで宣言するようにして下さい。

#### .TFB 拡張子

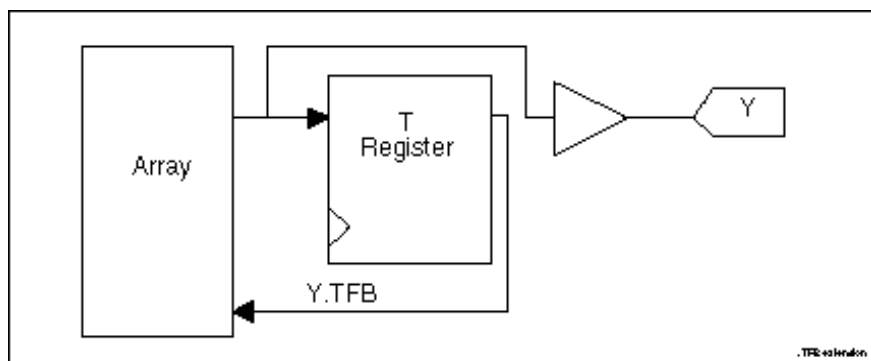


図 9-46 .TFB 拡張子

.TFB 拡張子は、プログラマブル出力マクロセルが組み合わせて配置され、しかも T レジスタが出力に接続されたままの特殊な状況で使用されます。.TFB 拡張子を使用すると、レジスタからのフィードバックを使用することができます。通常では、出力がレジスタに配置されると、レジスタからのフィードバックは、拡張子なしで記述されます。

## 論理式の再検討

表 9-12 にコンパイラが論理式を評価するために使用する規則を示します。これら基本的な規則はリファレンスの目的で表示されているだけです。

交換法則:

$$A \& B = B \& A$$

$$A \# B = B \# A$$

結合法則:

$$A \& (B \& C) = (A \& B) \& C$$

$$A \# (B \# C) = (A \# B) \# C$$

分配法則:

$$A \& (B \# C) = (A \& B) \# (A \& C)$$

$$A \# (B \& C) = (A \# B) \& (A \# C)$$

吸収法則:

$$A \& (A \# B) = A$$

$$A \# (A \& B) = A$$

ドモルガンの定理:

$$\!(A \& B \& C) = \!A \# \!B \# \!C$$

$$\!(A \# B \# C) = \!A \& \!B \& \!C$$

XOR 恒等式:

$$A \$ B = (\!A \& B) \# (A \& \!B)$$

$$\!(A \$ B) = A \$ \!B = \!A \$ B$$

$$= (\!A \& \!B) \# (A \& B)$$

定理:

$$A \& 0 = 0 \quad A \& 1 = A$$

$$A \# 0 = A \quad A \# 1 = 1$$

$$\begin{aligned} A \ \& \ A &= A & A \ \& \ !A &= 0 \\ \# \ A &= A & A \ \# \ !A &= 1 \end{aligned}$$

表 9-12 論理式の規則

## 式

式は、変数と演算子の組み合わせで式が評価されると結果を一つ生成します。式を幾つかの式で構成することができます。

式は、演算子の優先順位に従って処理されます。同じ優先順位のオペレータが式にある場合、左から右に処理されます。括弧(())を使用して処理の順序を変更することができます。すなわち括弧の内側から処理されます。

表 9-13 では、処理の順番や括弧の使用が式の値にどのように影響するかに注意して下さい。

式	結果	コメント
A # B & C	A # B & C	Parentheses change order
(A # B) & C	A & C # B & C	
!A & B	!A & B	DeMorgan's Theorem
!(A & B)	!A # !B	
A # B & C # D	A # D # B & C	Parentheses change order
A # B & (C # D)	A # B & C # B & D	

Table 9-13. Sample Expressions

## 論理式

論理式は、CUPL 言語の構成単位です。論理式の形式を以下に示します。

[!] var [.ext] = exp ;

ここで

var は、一つの変数またはリスト表記の規則に従って定義されたインデックス付きの変数またはインデックス付きでない変数のリストです。（このセクションのリスト表記の項を参照して下さい）変数リストが使用される場合、式はリストの各変数に割り付けられます。

.ext は省略可能な拡張子で、関数をプログラマブルデバイスの内部ノードに割り付けます。（表 9-11 を参照して下さい）

exp は式、すなわち変数と演算子の組み合わせです（このセクションの式の項を参照して下さい。）。

=は割り付け演算子です。式の値を変数または変数のリストに割り付けます。

!は補助演算子です。

補助演算子を使用して、論理式を否定することができます。この演算子は変数の前に付けて右辺の式が変数に割り付けられる前に右辺が補足されることを示します。左辺に補助演算子を使用するのは単に便利だからです。右辺の式全体を補足すると、式の記述が簡単になります。

自動ドモルガン機能（ピン変数宣言により割り付けられる出力極性）を持たない古い論理設計ソフトウェアでは、反転バッファのあるデバイスを使用する場合、補助演算子を使用する必要がありました。

テンプレートファイルにより与えられるソースファイルの Logic Equation セクションに論理式を記述して下さい。

論理式は、単にピン（またはノード）変数に規定され、任意の変数名で記述されます。このように定義される変数を中間変数と言います。中間変数名は論理式や追加の中間変数を生成するための他の式で使うことができます。このようにトップダウン形式で論理式を記述することにより、論理記述ファイルが読みやすくそして理解しやすくなります。

ソースファイルの Declarations and Intermediate Variable Definitions セクションに中間変数を記述して下さい。

論理式の例を以下に示します。

```
SEL_0=A15 & !A14;      /* A simple, decoded output pin */
Q0.D=Q1 & Q2 & Q3;      /* Output pin w/ D flip-flop */
Q1.J = Q2 # Q3;         /* Output pin w/ JK flip-flop */
Q1.K = Q2 & !Q3;
MREQ=READ # WRITE;     /* Intermediate Variable */
SEL_1=MREQ & A15;       /* Output intermediate var */
[D0..3] = 'h'FF;        /*Data bits assigned to constant*/
[D0..3].oe=read;        /*Data bits assigned to variable*/
```

## APPEND 命令

標準の論理式では、通常一つの式が変数に割り付けられます。APPEND 命令を使用すると、複数の式を一つの変数に割り付けることができます。フォーマットを以下に示します。

```
APPEND [!]var[.ext] = expr ;
```

ここで

!は変数の極性を定義する省略可能な補助演算子です。

var は 1 つの変数または標準のリストフォーマット記述されたインデックス付きまたはインデックスのない変数のリストです。

.ext は変数の機能を定義する省略可能な拡張子です。

=は割り付け演算子です。

expr は式です。

;  
:は命令の終わりを示すセミコロンです。

複数の APPEND 命令の結果の式は、すべての APPEND 命令の OR になります。式が変数に割り付けられていない場合、最初の APPEND 命令が最初の割り付けになります。

APPEND 命令の使用例の幾つかを以下に示します。

```
APPEND Y = A0 & A1 ;  
APPEND Y = B0 & B1 ;  
APPEND Y = C0 & C1 ;
```

上記の 3 つの命令は以下の式と等価です。

```
Y = (A0 & A1) # (B0 & B1) # (C0 & C1) ;
```

APPEND 命令は、ステートマシン変数にターム（リセットなど）を追加したり、ユーザ定義関数を作成したりする場合に便利です。（このセクションのステートマシンシンタクスとユーザ定義関数の項を参照して下さい。）

## セット演算

1 ビットの情報に関する演算（例えば、入力ピンやレジスタ、出力ピンなど）はすべてグループ化された複数のビットに割り付けられます。セット演算により、変数または式と情報の組みとの演算を行なうことができます。

情報の組みと一つの変数との間の演算では、情報の組みのそれぞれの要素と変数（または式）との間で実行された演算により情報の組みが新しく作成されます。例えば、

```
[D0, D1, D2, D3] & read
```

は以下の式と等価です。

```
[D0 & read, D1 & read, D2 & read, D3 & read]
```

演算が 2 つの組みで行われる場合、それらの組みは同じ大きさである必要があります。（すなわち、同じ数の要素を持っている必要があります。）2 つの組みの間の演算の結果、両方の組みの各要素間での演算結果の組みが新しく作成されます。

例えば

```
[A0, A1, A2, A3] & [B0, B1, B2, B3]
```

は以下の式と等価です。

```
[A0 & B0, A1 & B1, A2 & B2, A3 & B3]
```

ビットフィールド命令（このセクションのビットフィールド宣言命令を参照して下さい。）を使用すると、一つの変数名により参照される情報の組みに変数をグループ化できます。例えば、上記演算の変数の組みは以下のようにグループ化できます。

```
FIELD a_inputs = [A0, A1, A2 A3] ;
FIELD b_inputs = [B0, B1, B2, B3] ;
```

その後、上記の変数の組みにセット演算、例えば、AND を実行する場合、以下ようになります。

```
a_inputs & b_inputs
```

セット演算で数値が使用される場合、2 進数の組みとして扱われます。一つの 8 進数値は 3 つの 2 進数値の組みとして表わされます。10 進数値や 16 進数値は 4 つの 2 進値として表わされます。

Equivalent Number	Binary Set	Equivalent Number	Binary Set
'O'X	[X, X, X]	'H'X	[X,X,X,X]
'O'0	[0, 0, 0]	'H'0	[0,0,0,0]
'O'1	[0, 0, 1]	'H'1	[0,0,0,1]
'O'2	[0, 1, 0]	'H'2	[0,0,1,0]
'O'3	[0, 1, 1]	'H'3	[0,0,1,1]
'O'4	[1, 0, 0]	'H'4	[0,1,0,0]
'O'5	[1, 0, 1]	'H'5	[0,1,0,1]
'O'6	[1, 1, 0]	'H'6	[0,1,1,0]
'O'7	[1, 1, 1]	'H'7	[0,1,1,1]
'D'0	[0,0,0,0]	'H'8	[1,0,0,0]
'D'1	[0,0,0,1]	'H'9	[1,0,0,1]
'D'2	[0,0,1,0]	'H'A	[1,0,1,0]
'D'3	[0,0,1,1]	'H'B	[1,0,1,1]
'D'4	[0,1,0,0]	'H'C	[1,1,0,0]
'D'5	[0,1,0,1]	'H'D	[1,1,0,1]
'D'6	[0,1,1,0]	'H'E	[1,1,1,0]
'D'7	[0,1,1,1]	'H'F	[1,1,1,1]

数値を論理式のビットマスクとして使用すると効果的です。以下の 4 ビットカウンタでこのような例を示します。

```
field count = [Q3, Q2, Q1, Q0];
count.d = 'b' 0001 & (!Q0)
# 'b' 0010 & (Q1 $ Q0)
# 'b' 0100 & (Q2 $ Q1 & Q0)
# 'b' 1000 & (Q3 $ Q2 & Q1 & Q0);
```

セット表記をしないで記述した等価な論理式を以下に示します。

```
Q0.d = !Q0;
Q1.d = Q1 $ Q0;
Q2.d = Q2 $ Q1 & Q0;
Q3.d = Q3 $ Q2 & Q1 & Q0;
```

## 等価演算

他のセット演算と違い、等価演算は一つのブーリアン式を評価します。変数の組みと定数との比較をビット単位で行います。等価演算のフォーマッ

トを以下に示します。

1. [var, var, ... var]: constant ;
2. bit\_field\_var:constant ;

ここで

[var, var, ... var]はリスト表記の省略形です。

constant は数値です。(デフォルトでは16進数です。)

bit\_field\_var は、ビットフィールド命令を使用して定義された定数です。

:は、等価演算子です。

;は、ステートメントの終わりを示すセミコロンです。

かぎ括弧 ([]) は省略できません。変数リストを表わしています。

フォーマット 1 は、変数のリストと定数の比較で使用されるフォーマットです。

フォーマット 2 は、ビットフィールド変数と定数の比較で使用されるフォーマットです。

定数のビット位置とそれに対応する変数の組みの位置が比較されます。ビット位置が 1 の場所は、セット要素は変更されません。ビット位置が 0 の場所は、セット要素が否定されます。ビット位置が X の場所は、セット要素が削除されます。比較された要素は互いに AND され一つの式を作成します。以下の例では、16 進数の D (2 進数 1011) が A3、A2、A1、A0 と比較されます。

```
select = [A3..0]:'h'D ;
```

対応するビット位置が 1 すなわち真であるので、セット要素 A3、A2、A0 は変更されません。対応するビット位置が 0 すなわち偽であるので、セット要素 A1 は否定されます。従って、上記の式は以下の式と等価です。

```
select = A3 & A2 & !A1 & A0 ;
```

以下の例では、2 進数 1X0X が A3、A2、A1、A0 と比較されます。

```
select = [A3..0]:'b'1X0X ;
```

対応するビット位置が 1 すなわち真であるので、セット要素 A3 は変更されません。対応するビット位置が 0 すなわち偽であるので、セット要素 A1 は否定されます。対応するビット位置が dont-care のため、セット要素 A2 と A0 は式から削除されます。従って、上記の式は以下の式と等価です。

```
select = A3 & !A1 ;
```

アドレスのデコード以外にも、等価演算は、カウンタやステートマシンの指定にも使用できます。例えば、以下のように記述すると 4 ビットのカウンタを指定できます。

```
FIELD count = [Q0..3];
```



```

Q0.J = count:0 # count:2 # count:4 # count:6
      # count:8 # count:A # count:C # count:E ;
Q0.K = count:1 # count:3 # count:5 # count:7
      # count:9 # count:B # count:D # count:F ;
Q1.J = count:1 # count:5 # count:9 # count:D ;
Q1.K = count:3 # count:7 # count:B # count:F ;
Q2.J = count:3 # count:B ;
Q2.K = count:7 # count:F ;
Q3.J = count:7 ;
Q3.K = count:F ;

```

等価演算子は、セット変数のそれぞれに演算されるセット演算と一緒に使用することができます。シンタクスを以下に示します。

```
[var, var, ... , var]:op
```

上式は以下の式と等価です。

```
var op var op ... var
```

ここで

op は、&や#、\$演算子です。（または、これに相当する演算です。）

var は、任意の変数名です。

例えば以下の式

```

[A3,A2,A1,A0]:&
[B3,B2,B1,B0]:#
[C3,C2,C1,C0]:$

```

はそれぞれ以下の式と等価です。

```

A3 & A2 & A1 & A0
B3 # B2 # B1 # B0
C3 $ C2 $ C1 $ C0

```

等価演算は等価な 2 進数の組みで使用され出力値の関数テーブルを作成します。例えば、以下の Binary-to-BCD コードコンバータでは、出力値は、入力と、等価な 2 進数のセットを定義する等価演算を使用して割り付けられます。

```

FIELD input = [in3..0] ;
FIELD output = [out4..0] ;
/* in3..0 ->out4..0*/

```

```

$DEFINE L 'b'0
$DEFINE H 'b'1
output = input:0 & [L, L, L, L, L]
      # input:1 & [L, L, L, L, H]
      # input:2 & [L, L, L, H, L]
      # input:3 & [L, L, L, H, H]
      # input:4 & [L, L, H, L, L]

```

```

# input:5    & [L,  L,  H,  L,  H]
# input:6    & [L,  L,  H,  H,  L]
# input:7    & [L,  L,  H,  H,  H]
# input:8    & [L,  H,  L,  L,  L]
# input:9    & [L,  H,  L,  L,  H]
# input:A    & [H,  L,  L,  L,  L]
# input:B    & [H,  L,  L,  L,  H]
# input:C    & [H,  L,  L,  H,  L]
# input:D    & [H,  L,  L,  H,  H]
# input:E    & [H,  L,  H,  L,  L]
# input:F    & [H,  L,  H,  L,  H];
$UNDEF L
$UNDEF H

```

## インデックス付き変数のビットフィールドと等式

インデックス付き変数やフィールド命令、レンジ関数の演算では、それらは互いに関連があります。このセクションでは、これらの関係を説明します。

このセクションの始めの方で説明したように、インデックス付き変数を使用すると複数の変数を短いステートメントで宣言することができます。

例えば

```
Pin [2..4] = [AD0..2];
```

は以下のように展開されます。

```

Pin 2 = AD0;
Pin 3 = AD1;
Pin 4 = AD2;

```

FIELD 命令を使用すると、関連する信号の組みを一つの要素にグループ化できます。32 ビットのビットフィールドを使用する場合、フィールドの各ビットはフィールドのメンバを示します。メンバの数が 32 以下の場合、余分なビットは無視されます。例えば、

```

Pin 2 = VAR_A;
Pin 3 = VAR_B;
Pin 4 = VAR_C;
Pin 15 = ROM_SEL;
FIELD ADDR = [VAR_A,VAR_B,VAR_C];

```

以下の図に、変数 VAR\_A や VAR\_B、VAR\_C がビットフィールドにどのようにマップされるかを示します。

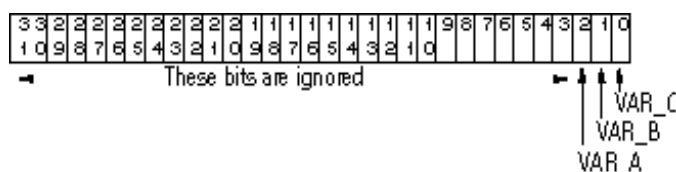


図9-47 メンバ変数のビットフィールドマッピング

出力を以下のように定義すると

```
ROM_SEL = ADDR:3;
```

この式のビットフィールドの内容は以下になります。

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```

これは、以下の式と等価です。

```
ROM_SEL = !VAR_A & VAR_B & VAR_C;
```

インデックス付き変数を使用する場合、内部表現はわずかに変わります。変数のインデックス番号によりビットフィールドの位置が決まります。従って、フィールドの宣言にかかわらず、VAR0 のビット位置はビット 0 の位置になります。以下の 2 つの宣言の内部表現は違います。

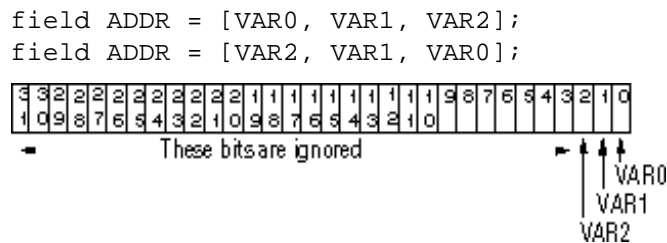


図 9-48 インデックス付き変数のビットフィールド表現

出力を以下のように定義すると

```
ROM_SEL = ADDR:3;
```

この式のビットフィールドの内容は以下になります。

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```

これは以下の式と等価です。

```
ROM_SEL = !VAR2 & VAR1 & VAR0;
```

インデックス番号の大きい値を使用する変数の組みを取る場合、インデックス変数が扱われる方法は、予想よりも違ったものになることがわかります。使用される変数が VAR17、VAR18、VAR19 の場合、ビットマップはそれに応じて変わります。ただし、3 はビット 0、1、2 にマップされて 3 であるので、3 との比較は正しく行われません。この場合、必要な位置になるまで 0 を加える必要があります。

出力を以下のように定義すると

```
FIELD ADDR = [VAR18, VAR17, VAR16];
ROM_SEL = ADDR:3;
```

変数は、ビットフィールド ADDR に以下のようにマップされます。

```
{bmc 1-49P205.bmp}
```

図 9-49 0 以外のインデックス番号から始まるインデックス付き変数のビットフィールド表現

このビットフィールドと 3 を比較する場合、ビットは正しく合いません。

以下の行はビットフィールドに定数 3 がどのようにマップされるかを示します。

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```

上記の比較で意味のあるビットは、変数を示すビット上にマップされていません。これを修正するには、定数の後ろにゼロを付けて位置を合わせます。

```
ROM_SEL = ADDR:30000;
```

これにより、この定数は以下のようにビットフィールドにマップされ、正しい結果を出力できます。

```
"XXXXXXXXXXXX011000000000000000"
```

```
ROM_SEL = !VAR18 & VAR17 & VAR16;
```

## レンジ演算

レンジ演算は等式演算に似ています。ただし、定数フィールドが一つの定数ではなくて値のレンジになっています。レンジの各定数に対して、ビットが等しいかが比較されます。レンジ演算のフォーマットを以下に示します。

```
1. [var, var, ... var]:[constant_lo..constant_hi] ;  
2. bit_field_var:[constant_lo..constant_hi] ;
```

ここで

[var, var, ... var]は変数のリストの省略形です。

bit\_field\_varはビットフィールド命令を使用して定義された変数です。

:は等価演算子です。

;はステートメントの終わりを示すセミコロンです。

[constant\_lo constant\_hi]はレンジ演算を定義する数値（デフォルトでは 16 進数）です。

かぎ括弧 ( ) は省略できません。これで囲まれたアイテムはリストのアイテムを意味します。

Format 1 は変数のリストと定数のレンジとのレンジ演算を示します。

Format 2 はビットフィールド変数と定数のレンジのレンジ演算を示します。

constant\_lo 以上で constant\_hi 以下の数値はすべて等価演算のように AND されます。それから、互いに OR されて最終的な結果が得られます。例えば、RANGE 表記を使用してアドレスバス A3、A2、A1、A0 の 1100 と 1111 との間のデコードされた 16 進値を検索することができます。まず、アドレスバスを以下の様に定義します。

```
FIELD address = [A3..A0]
```

それから、RANGE 式を記述して下さい。

```
select = address:[C..F] ;
```

これは以下の式と等価です。

```
select = address:C # address:D  
        # address:E # address:F ;
```

この式を展開すると以下のようになります。

```
select = A3 & A2 & !A1 & !A0  
        # A3 & A2 & !A1 & A0  
        # A3 & A2 & A1 & !A0  
        # A3 & A2 & A1 & A0 ;
```

コンパイラの論理最小化機能を使用すると、上記の式を一つのプロダクトタームにすることができます。レンジの最小化機能を以下に説明します。まず、1 行目と 2 行目が結合され、3 行目と 4 行目が結合されて以下の式を作成します。

```
select = A3 & A2 & !A1 & (!A0 # A0)  
        # A3 & A2 & A1 & (!A0 # A0) ;
```

式(!A0#A0)は常に真のため、式から削除されます。そして、式は以下のように縮小されます。

```
select = A3 & A2 & !A1  
        # A3 & A2 & A1 ;
```

同様のプロセスで式は以下のように縮小されます。

```
select = A3 & A2 & (!A1 # A1) ;
```

式(!A1#A1)は常に真のため、式から削除され、一つのプロダクトタームに縮小されます。

```
select = A3 & A2 ;
```

等価演算またはレンジ演算のどちらかにインデックス付き変数が使用される場合、CONSTANT フィールドには、インデックス付き変数と同じ数のビットが必要です。ピンリストやフィールド宣言にないインデックス位置は演算で DONT CARE になります。

以下の例では、ピンが割り付けられ、アドレスバスが宣言され、そして 16 進のメモリアドレス 8000 から BFFF の範囲でデコードされた出力が処理されます。

```
PIN      [1..4] = [A15..12] ;  
FIELD    address = [A15..12] ;  
chip_select = address:[8000..BFFF] ;
```

変数 A15 と A14、A13、A12 だけがデバイスへの入力です。16 ビットのアドレスはレンジ表現で使用されます。最上位ビット A15 によりフィールドが 16 ビットであることが決まります。変数のインデックス番号がビット位置を決定するために使用されるので、下位のアドレスビット(A0 から A11)

は実際には式の中で DONT CARE されます。下位ビットが存在しないデバイスでも、定数はそれらが存在するように記述され、より詳しいドキュメンテーションを生成します。

例えば、I/O ポートのマイクロプロセッサアドレスをデコードする以下のようなアプリケーションを考えます。

```
PIN [3..6] = [A7..10] ;
FIELD ioaddr = [A7..10]; /*order of field
/* インデックス付き変数を使用する場合、フィールド宣言の順番は
重要ではありません。*/
io_port = ioaddr:[400..6FF] ;
```

最上位ビットは A10 なので、11 ビットの定数フィールドが必要です。(3桁の 16 進数は 12 ビットですが、A11 のビット位置は無視されます。)

A0 から A6 のアドレスビットは、式の中で DONT CARE されます。ビット位置の調整をしない場合、レンジ式は以下のように書くことができます。

```
io_port = ioaddr:[8..D] ;
```

この式は、必要な I/O アドレスレンジを明確に表現していません。

レンジ演算の無い元の式は、以下のように記述されます。

```
io_port = A10 & !A9 & !A8 & !A7
# A10 & !A9 & !A8 & A7
# A10 & !A9 & A8 & !A7
# A10 & !A9 & A8 & A7
# A10 & A9 & !A8 & !A7
# A10 & A9 & !A8 & A7 ;
```

コンパイラによりこの式は、以下のように縮小されます。

```
io_port = A10 & !A9 # A10 & A9 & !A8 ;
```

特に、インデックス番号の値を大きい変数で構成されるフィールドの場合、レンジ機能を不注意に使用すると、プロダクトタームの数を増大させることになります。レンジ演算のアルゴリズムは、演算の中の 2 つの定数のビット毎の比較をインデックス番号  $\alpha$  (フィールドにこれが存在しなくても) から行なうことです。constant\_lo のビット位置の値が constant\_hi よりも小さい場合、そのビット位置の変数は AND の式が作成されません。constant\_lo のビット位置の値が constant\_hi の値と等しいか大きい場合、新しい値と元の constant\_hi の値の間の定数すべてに対して AND 式が作成されます。

例えば 16 ビットアドレスフィールドのレンジ関数を使用する以下の論理式を考えます。

```
field address = [A15..12] ;
board_select = address:[A000..DFFF] ;
```

図 9-50 はコンパイラアルゴリズムにより、この式がどのように処理されるかをしめします。

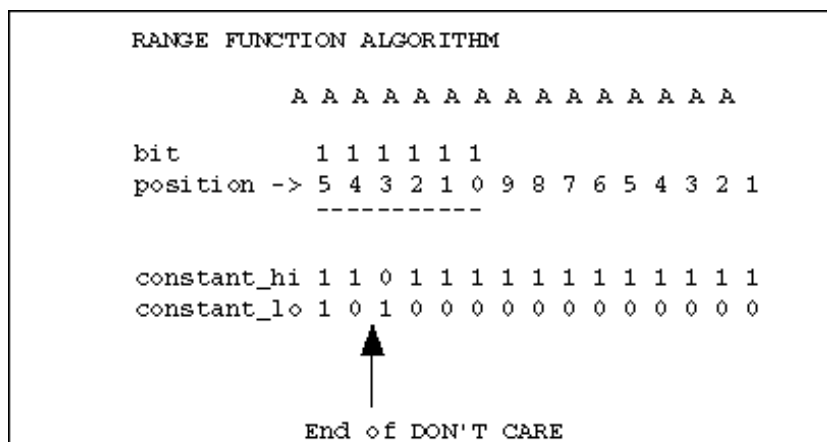


図 9-50 レンジ関数アルゴリズム

ビット位置 13 より下位のビットは constant\_lo が constant\_hi よりも小さいので、アルゴリズムにより、ビット位置 13 より下位のビットは無視されます。図 9-51 に結果を示します。

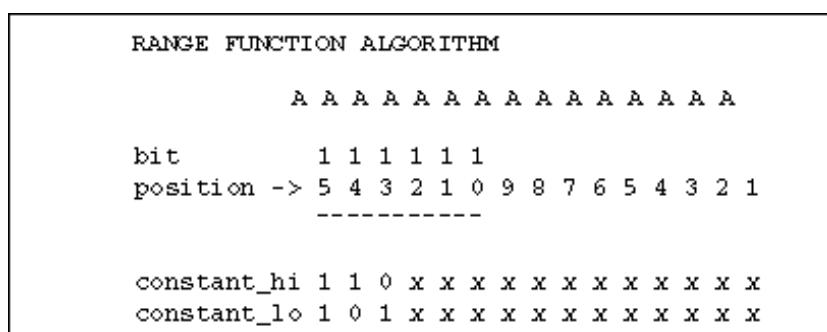


図 9-51 レンジ関数の結果

以下の 2 つのプロダクトタームは、図 9-51 のレンジ関数で生成されました。

```

A15 & A14 & !A13
A15 & !A14 & A13

```

以下の式はレンジ関数の別の使用例です。

```

board_select = address:[A000..D000] ;

```

constant\_lo の値と constant\_hi の値が最下位ビットにとあっているので、アルゴリズムはプロダクトタームを以下のように生成します。

```

1010 0000 0000 0000
1010 0000 0000 0001
1010 0000 0000 0010
1010 0000 0000 0011
.
1100 1111 1111 1111
1101 0000 0000 0000

```

生成されたプロダクトタームの数は、12000 ( 4096x3+1 ) を越えます。コンパイラは、一度にこれだけの数をメモリに記憶することができないので、これだけの数のプロダクトタームの場合、アウトオブメモリのエラーが発生します。

## 真理値表

論理記述をわかりやすく表現するには情報のテーブルを利用するとよい場合があります。CUPL では、TABLE キーワードを使用して情報テーブルを作成することができます。TABLE キーワードを使用するためのフォーマットを以下に示します。

```
TABLE var_list_1 => var_list_2 {  
    input_n => output_n ;  
    .  
    .  
    input_n => output_n ;  
}
```

ここで

var\_list\_1 は、入力変数の定義です。

var\_list\_2 は、出力変数の定義です。

input\_n は、var\_list\_1 のデコードされた値 ( でフォールトでは 16 進数値 ) または、var\_list\_1 のデコードされた値のリストです。

output\_n は、var\_list\_2 のデコードされた値 ( デフォルトでは 16 進数値 ) です。

{ } は割り付けブロックの始めと終わりを示す中括弧です。

=> は、変数リスト間や入力と出力の値との間の 1 対 1 の割り付けを示します。

まず、関連した入力変数と出力変数のリストを定義して下さい。次に、入力のデコードされた値と出力変数のリストとの間の 1 対 1 の割り付けを指定して下さい。入力のデコード値には DONT-CARE 値を使用できます。ただし、出力には使用できません。

入力値のリストを指定して一つの命令で複数の割り付けを行うことができます。以下のブロックは簡単な hex-to-BCD 変換器の記述です。

```
FIELD input = [in3..0] ;  
FIELD output = [out4..0] ;  
TABLE input => output {  
    0=>00; 1=>01; 2=>02; 3=>03;  
    4=>04; 5=>05; 6=>06; 7=>07;  
    8=>08; 9=>09; A=>10; B=>11;  
    C=>12; D=>13; E=>14; F=>15;  
}
```

以下の例は、いろいろな大きさの RAM や ROM、I/O デバイスのアドレス



のデコードを行うための入力番号のリストの使用例です。アドレスレンジはレンジ演算（このセクションのレンジ演算の項を参照して下さい。）の規則（インデックス付き変数の使用法に関する）にしたがってデコードされます。

```
PIN [1..4] = [a12..15] ;           /* Upper 4 address*/
PIN 12 = !RAM_sel ;                 /* 8K x 8 RAM */
PIN 13 = !ROM_sel ;                 /* 32K x 8 ROM */
PIN 14 = !timer_sel ;               /* 8253 Timer */
FIELD address = [a15..12] ;
FIELD decodes = [RAM_sel,ROM_sel,timer_sel] ;
TABLE address => decodes {
[1000..2FFF] => 'b'100;           /* select RAM */
[5000..CFFF] => 'b'010;           /* select ROM */
F000 => 'b'001;                   /* select timer */
}
```

## フィールド比較演算 "=="

フィールド比較演算"=="により2つのフィールドを比較します。すなわち、2つのフィールドが一致する場合だけ TRUE を生成します。2つのフィールド変数は同じ数の要素（ビット）のリストである必要があります。

例えば

```
Field f1 = [a3..0];
Field f2 = [b3..0];
x = f1 == f2;
```

a3..0 と b3..0 が一致する場合のみ出力 x が真になります。コンパイラは以下のようなロジックを使用してフィールドの比較演算を実行します。

```
x = !(a0 $ b0) & !(a1 $ b1) & !(a2 $ b2) & !(a3 $ b3);
```

## DECLARE

DECLARE 命令を使用してピンやピンノードの属性を宣言します。属性は、グローバル入力バッファや RAM ブロックなどのハードウェア特性です。

```
DECLARE <manuf ID> <attrib> <variable list>
```

コンパイラは、DECLARE ステートメントの設計規則のチェックを行います。

例えば

```
DECLARE XILINX 3VOLT [x, y];
```

3VOLT は XILINX デバイスでは無効な属性のためコンパイラはエラーを発生します。

有効な属性を表 9-15 に示します。

製造元	属性	説明
-----	----	----

XILINX	ROM4	ROM (1 つの CLB)
XILINX	ROM5	ROM (半分の CLB)
XILINX	RAM	(1 つの CLB)
XILINX	RAM5	RAM (半分の CLB)
XILINX	ACLK	オルタネートクロックバッファ
XILINX	GCLK	グローバルクロックバッファ
XILINX	BUFGS	ハイファンアウトバッファ
XILINX	BUFGP	ハイファンアウトバッファ
XILINX	INIT_TO_R	パラメータ、フリップフロップを R 状態に初期化します。
XILINX	INIT_TO_S	パラメータ、フリップフロップを S 状態に初期化します。

表 9-15 DECLARE 命令の属性

DECLARE 命令の使用例を以下に示します。

```
Pin = [A3..0];
Pin = WE;
Pin = D;
Pin = O;

DECLARE XILINX RAM4 [A0,A1,A2,A3,WE,D,O];
...
```

図 9-52 RAM4 の DECLARE

```
Pin = CLK;
DECLARE XILINX GCLK [CLK];
[x, y, z].ck = CLK;
...
```

図 9-53 グローバルクロックの DECLARE

```
Pin = x;
Pin = y;
DECLARE XILINX INIT_TO_R [x, y];
...
```

図 9-54 リセット状態を開始するレジスタの DECLARE

## PROPERTY

PROPERTY 命令は DECLARE 命令と同じ動作をします。DECLARE 命令と違うのは、コンパイラが PROPERTY 命令に関する設計ルールのチェックを行わないことです。

PROPERTY <manuf ID> { property statement };

PROPERTY 命令の使用例を以下に示します。

```
Pin = CLK;
Pin = X;
```

```

Pin    =    Y;
...
PROPERTY INTEL { @PIN_ATTRIB X DELAYCLK };
PROPERTY INTEL { @PIN_ATTRIB Y DELAYCLK };
...
[X, Y].ck = CLK;
...

```

図 9-55 PROPERTY 命令を使用したディレイクロックの指定

```

Pin    =    X;
Pin    =    Y;
PROPERTY INTEL { @PIN_ATTRIB X 3VOLT };
PROPERTY INTEL { @PIN_ATTRIB Y 5VOLT };
...

```

図 9-56 PROPERTY 命令を使用した 3V と 5V ピンの指定

```

Pin    =    [Data9..0];

PROPERTY INTEL { @PIN_ATTRIB Data[0:9] OPEN_DRAIN };

```

図 9-57 オープンドレンタイプの出力の指定

DECLARE 命令と PROPERTY 命令の違いは、PROPERTY 命令ではフィールド変数を使用できないことです。PROPERTY 命令は、{ } で囲まれた情報を OPENPLA ファイルへ流します。

## DEMORGAN

DEMOROGAN 命令を使用してドモルガンの定理を式に適用できます。これにより、使用されているプロダクトタームの数を減らすことができます。

DEMORGAN [var\_list] = Demorgan\_Option;

Demorgan\_Option は 0 から 2 までの数値です。

- 0 デフォルト
- 1 ドモルガンの定理を強制的に式に適用します。
- 2 プロダクトタームの数が減る場合だけドモルガンの定理が式に適用されます。

デバイスが極性をプログラムできる機能がある場合にドモルガン命令を使用できます。すなわち、極性のヒューズまたはマルチプレクサを持つデバイスです。この命令は VIRTUAL でも使用できます。

P16L8 などの固定極性のデバイスを選択した場合、コンパイラは DEMORGAN 命令を無視しデバイスに合うように式を評価します。極性をプログラムできるデバイスや VIRTUAL を選択した場合、コンパイラは DEMORGAN 命令で指定された値に応じて式にドモルガンの定理を適用します。DEMORGAN 命令の動作について以下に示します。

```

Device p16l8;

```

```
Pin 16 = !x;

DEMORGAN [x] = 2; /* best usage of product terms */
x = a # b;
```

図 9-58 固定極性デバイス、DEMORGAN 命令は無視

P16L8 のピン 16 は固定反転バッファのため、式  $x=a\#b$  にはドモルガンの定理が適用されません。

```
Device p22v10;

Pin 16 = !x;

DEMORGAN [x] = 2; /* best usage of product terms */
x = a # b;
```

図 9-59 極性をプログラムできるデバイスの出力に対する最適な解答

式がプロダクトタームの最小化を使用しているため、コンパイラは、出力 ( $!x=!a\&!b$ ) にドモルガンの定理を適用した結果を生成します。

## REGISTER\_SELECT

REGISTER\_SELECT 命令により、レジスタタイプを変更することができます。コンパイラは、指定されたレジスタタイプの論理式を生成します。

```
REGISTER_SELECT [var_list] = register_type;
```

register\_type は、目的のレジスタを現す数値です。

- |   |                                      |
|---|--------------------------------------|
| 0 | 指定されたレジスタを使用します。                     |
| 1 | D                                    |
| 2 | T                                    |
| 3 | JK                                   |
| 4 | SR                                   |
| 5 | D と T のうちでプロダクトタームの使用に適したレジスタを使用します。 |

```
REGISTER_SELECT [x] = 1;
x.j = a;
x.k = b;
```

図 9-60 JK レジスタを使用した式の D レジスタの式への変換

コンパイラにより、JK タイプの式が変換され、以下のような D タイプの式が生成されます。

```
x.d =>    a & !x
        # a & !b
        # !b & x;
```

図 9-61 変換された D レジスタを使用した式

## ステートマシンシンタックス

CUPL コンパイラベース PLD 言語で使用するステートマシンにより論理設計のゲートや式レベルの段階をとばして直接システムレベルの記述から PLD 手法へ移行することができます。加えて、アセンブラベースのアプローチと違い、ステートマシンのアプローチにより設計の明確なドキュメンテーションの作成が可能になります。

### ステートマシンモデル

非同期ステートマシンの論理回路には、フリップフロップがあります。その出力は、それ自体またはその他のフリップフロップの入力にフィードバックされるので、フリップフロップの入力値は、それ自身の出力やその他のフリップフロップの出力に依存します。従って、最終的な出力値はそれ自身の以前の値とその他のフリップフロップの以前の値に依存します。

図 9-52 に示すように、CUPL ステートマシンモデルは、入力と組み合わせ論理、ストレージレジスタ、ステートビット、レジスタード出力、ノンレジスタード出力の 6 個の部品を使用します。

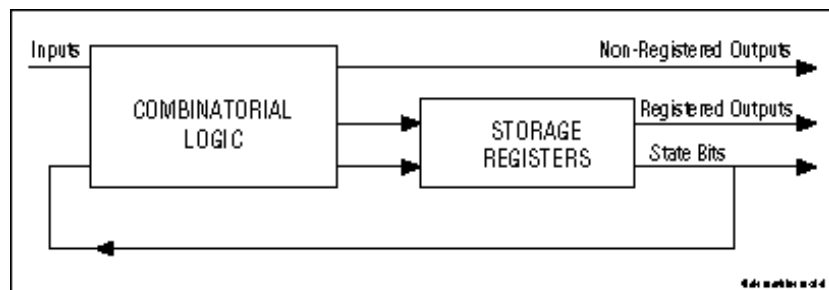


図 9-62 ステートマシンモデル

以下の定義は図 9-52 のタイミングダイアグラムを参照して記述されています。

Inputs - 入力他は他のデバイスからデバイスへ入る信号です。

Combinatorial Logic - 組み合わせ論理は、任意の組み合わせ論理ゲート（通常 AND-OR）で、このゲートより、ゲートの状態の変化を駆動する信号の  $T_{pd}$ （伝播遅れ時間）nsec 後に出力信号が作成されます。 $T_{pd}$  は、入力またはフィードバックの入力とノンレジスタ出力の発生との間の遅れです。

State Bits - ステートビットはステートマシン組み合わせ論理から入力を受け取る任意のフリップフロップ要素です。レジスタの中には、ステートビットに使用されるものもあります。その他はレジスタード出力に使用されます。

Storage Registers - レジスタード出力はクロックパルスの発生後  $T_{co}$ （クロックから出力までの時間）nsec に出力されます。 $T_{co}$  はクロック信号の入力とフリップフロップ出力の発生との間の時間遅れです。

図 9-53 にステートマシンの部品間のタイミング関係を示します。

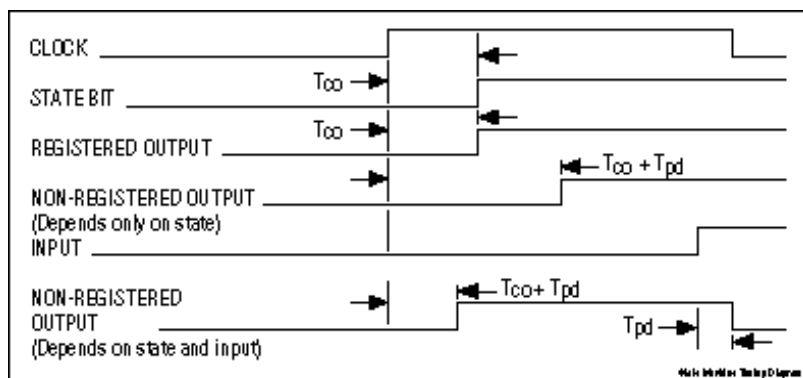


図 9-63 ステートマシンタイミングダイアグラム

システムを正しく動作させるには、PLD が必要とするセットアップやホールド時間が合っている必要があります。ほとんどの PLD では、セットアップ時間 ( $T_{su}$ ) には、通常、組み合わせ論理の伝播遅れとフリップフロップのセットアップ時間が含まれます。 $T_{su}$  は、フィードバックや入力のイベントがフリップフロップの入力に現れるのに要する時間です。続くクロック入力は、この結果がフリップフロップの入力で有効になるまで適用されません。フリップフロップは D、D-CE、J-K、S-R、T タイプを使用できます。

Non-registered Outputs - ノンレジスタード出力は、組み合わせ論理ゲートから直接入力される出力です。それらは、ステートビットまたは入力信号の関数（で非同期のタイミング）です。または、現在のステートビットの値に依存します。この場合、それらは、アクティブなクロックエッジが発生した後  $T_{co} + T_{pd}$  後に発生します。

Registered Outputs - レジスタード出力はストレージレジスタからの出力です。ただし、実際のステートビットフィールド（すなわち、ビットフィールドはすべてのステートビットで構成されます。）には含まれません。ステートマシン理論には、プリセット状態から次の状態への移行に応じたレジスタード出力のセットとリセットが必要です。これにより、レジスタード出力は、マシンが移行する状態に応じてセットまたはリセットになります。ホールドモードでは、現在のステートランジションがそのレジスタード出力の動作を指定しない間は、レジスタード出力は最後の値のままです。

演算のホールドモードが使用できるデバイスは D-CE または J-K、S-R タイプのフリップフロップを使用するデバイスです。

## ステートマシンシンタクス

ステートマシンモデルを実行するために、ステートマシンに任意の関数を記述できるシンタクスが CUPL により供給されます。

SEQUENCE キーワードにより、ステートマシンの出力が識別されます。このキーワードはステートマシンの機能を定義する命令の後に記述されます。SEQUENCE シンタクスのフォーマットを以下に示します。

```
SEQUENCE state_var_list {
```

```

PRESENT state_n0 statements ;
.
.
.
PRESENT state_nn statements ;
}

```

ここで

state\_var\_list はステートマシンブロックで使用されるステートビット変数のリストです。変数リストはフィールド変数により表現されます。

state\_n はステート番号で、state\_variable\_list のデコードされた値です。それぞれの PRESENT 命令に固有のものである必要があります。

statements はこのセクションのサブセクションで説明される条件命令や next 命令、出力命令です。

;  
: はステートメントの終わりを表わすセミコロンです。

{ } は、ステートマシンの記述の最初と最後を示す中括弧です。

\$DEFINE コマンドを使用して定義される名前を使用して、state\_numbers を表わすことができます。

SEQUENCE キーワードにより、ターゲットデバイスのデフォルトタイプで生成されるストレージレジスタ出力やレジスタード出力を設定することができます。例えば、P16R8 をターゲットデバイスとする設計で SEQUENCE キーワードを使用して、ステートストレージレジスタやレジスタード出力を D タイプのフリップフロップとして生成することができます。

デバイスの中には、ストレージレジスタを複数のタイプでプログラムできるものがあります。F159 ( 符号付き PLS159 ) の場合、ストレージレジスタは D または J-K タイプのフリップフロップになることができます。デフォルトでは、F159 の設計に SEQUENCE ステートメントを使用することにより、ステートストレージレジスタやレジスタード出力を J-K タイプのフリップフロップとして生成することができます。このデフォルトをオーバーライドするには、SEQUENCED キーワードを、SEQUENCE キーワードの代わりに使用して下さい。これにより、ステートレジスタやレジスタード出力を D タイプのフリップフロップとして生成することができます。

SEQUENCE キーワードや SEQUENCED キーワードに加えて SEQUENCEJK キーワードや SEQUENCERS キーワード、SEQUENCET キーワードがあります。それぞれ、ステートレジスタやレジスタード出力を J-K タイプや S-R タイプ、T タイプのフリップフロップとして生成することができます。

### 無条件の NEXT 命令

この命令は、現在の状態から指定された次の状態へのトランジションを表わします。フォーマットを以下に示します。

```
PRESENT state_n
```

```
    NEXT state_n ;
```

ここで

state\_n はステートマシンの出力であるステートビットのデコードされた値です。

\$DEFINE コマンドを使用して state\_n を表わす別の名前を割り付けることができます。

命令は無条件に実行されるので、各 PRESENT 命令に一つの NEXT 命令を記述できます。

以下の例では、2 進数の状態 01 から 2 進数の状態 10 へのトランジションを指定します。

```
PRESENT 'b'01
NEXT 'b'10 ;
```

図 9-64 に上記の例で示されるトランジションを示します。

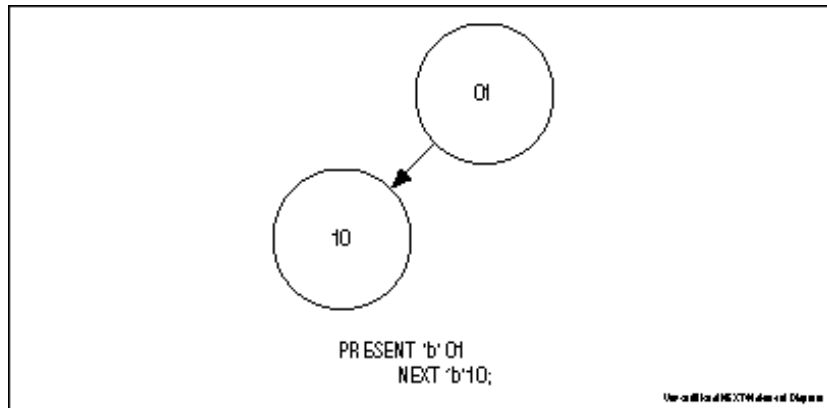


図 9-64 無条件の NEXT 命令ダイアグラム

上記の例と図で示されたトランジションでは、コンパイラにより、指定されたフリップフロップのタイプに応じて、以下の式が生成されます。

#### D タイプフリップフロップ

```
APPEND Q1.D = !Q1 & Q0;
APPEND Q0.D = 'b'0; /* implicitly resets */
```

#### J-K タイプフリップフロップ

```
APPEND Q1.J = !Q1 & Q0;
APPEND Q1.K = 'b'0;
APPEND Q0.J = 'b'0;
APPEND Q0.K = !Q1 & Q0;
```

#### S-R タイプフリップフロップ

```
APPEND Q1.S = !Q1 & Q0;
APPEND Q1.R = 'b'0;
APPEND Q0.S = 'b'0;
```



```
APPEND Q0.R = !Q1 & Q0;
```

#### D-CE タイプフリップフロップ

```
APPEND Q1.D = !Q1 & Q0;  
APPEND Q1.CE = !Q1 & Q0;  
APPEND Q0.D = 'b'0;  
APPEND Q0.CE = !Q1 & Q0;
```

#### T タイプフリップフロップ

```
APPEND Q1.T = !Q1 & Q0;  
APPEND Q0.T = !Q1 & Q0;
```

APPEND コマンドについての説明は、このセクションの APPEND 命令の項を参照して下さい。

#### 条件付き NEXT 命令

この命令は、指定された入力の条件が合えば、現在の状態から次の状態へのトランジションを示します。

```
PRESENT state_n  
IF expr NEXT state_n;  
.  
.  
.  
IF expr NEXT state_n;  
[DEFAULT NEXT state_n;]
```

ここで

state\_n はステートマシンの出力であるステートビットのデコードされた値です。

expr は任意の式です。(このセクションの式の項を参照して下さい。)

は命令の終わりを示すセミコロンです。

かぎ括弧 [] は省略可能なアイテムを示します。

各ステート番号の値は、固有の値である必要があります。

各 PRESENT 命令に対して複数の条件命令を指定することができます。

DEFAULT 命令は省略可能です。この命令は、指定された条件命令に合うものがない場合、現在の状態から次の状態へのトランジションを示します。言い換えると、条件命令の補助的な役割をします。

DEFAULT 命令を使用するときは、気を付けて下さい。DEFAULT 命令は、他の条件の補助的な命令なので、DEFAULT 命令により式が複雑になりコンパイラの動作が遅くなる場合があります。多くのアプリケーションでは、DEFAULT 命令の代わりに一つか二つの条件命令が指定されます。

以下に、DEFAULT 命令を使用しない 2 つの条件付き NEXT 命令の例を示

します。

```
PRESENT 'b'01
IF INA NEXT 'b'10;
IF !INA NEXT 'b'11;
```

図 9-65 に上記の例で示されるトランジションを示します。

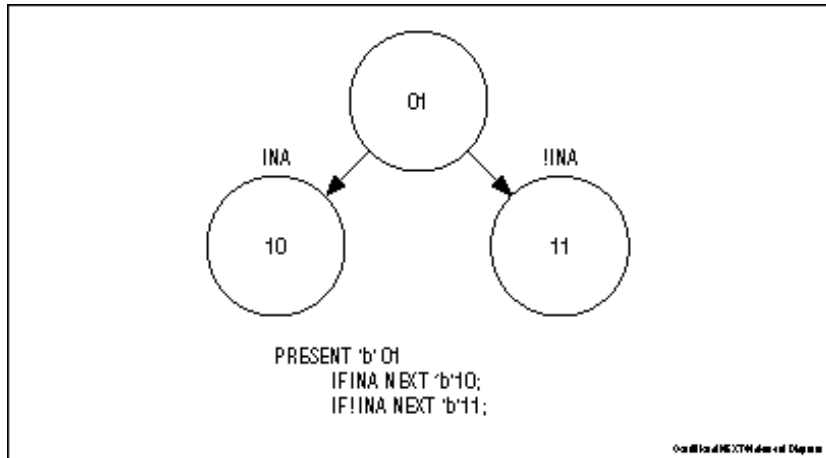


図 9-65 条件付き NEXT 命令ダイアグラム

上記の例と図で示されたトランジションでは、コンパイラは、指定されたフリップフロップのタイプに応じて以下の式を生成します。

#### D タイプフリップフロップ

```
APPEND Q1.D = !Q1 & Q0;  
APPEND Q0.D = !Q1 & Q0 & !INA;
```

#### D-CE タイプフリップフロップ

```
APPEND Q1.D = !Q1 & Q0;  
APPEND Q1.CE = !Q1 & Q0;  
APPEND Q0.D = !Q1 & Q0 & !INA;  
APPEND Q0.CE = !Q1 & Q0 & INA;
```

#### J-K タイプフリップフロップ

```
APPEND Q1.J = !Q1 & Q0;  
APPEND Q1.K = 'b'0;  
APPEND Q0.J = 'b'0;  
APPEND Q0.K = !Q1 & Q0 & INA;
```

#### S-R タイプフリップフロップ

```
APPEND Q1.S = !Q1 & Q0;  
APPEND Q1.R = 'b'0;  
APPEND Q0.S = 'b'0;  
APPEND Q0.R = !Q1 & Q0 & INA;
```

#### T タイプフリップフロップ

```

APPEND Q1.T = !Q1 & Q0;
APPEND Q0.T = !Q1 & Q0 & INA;

```

以下に DEFAULT 命令を使用した二つの条件命令の例を示します。

```

PRESENT 'b'01
IF INA & INB NEXT 'b'10';
IF INA & !INB NEXT 'b'11';
DEFAULT NEXT 'b'00';

```

図 9-66 に上記例で示されるトランジションを示します。DEFAULT 命令で生成される式に注意して下さい。

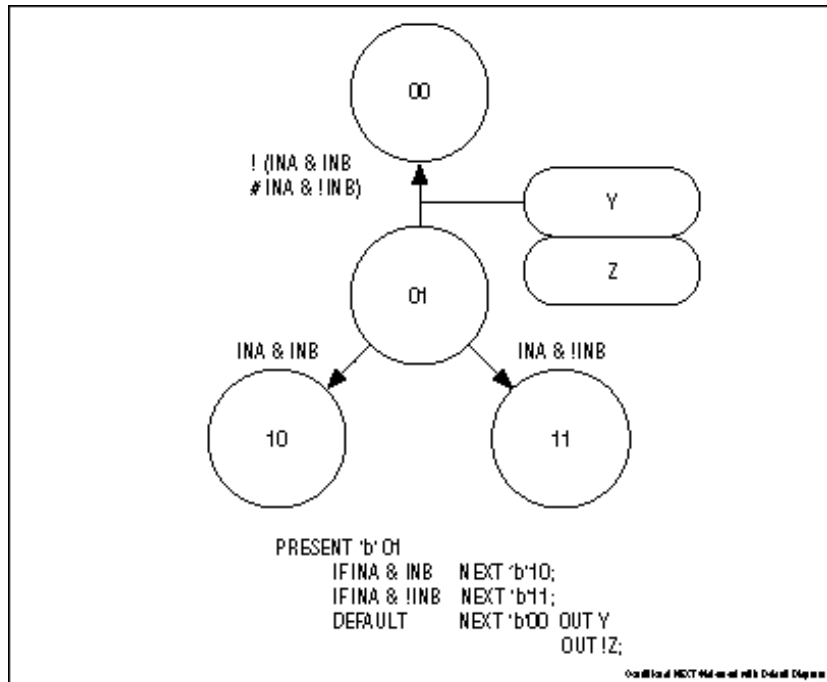


図 9-66 デフォルトのある条件付き NEXT 命令のダイアグラム

上記の例と図で示されるトランジションでは、コンパイラは、指定されたフリップフロップのタイプに応じて以下の式を生成します。

#### D タイプフリップフロップ

```

APPEND Q1.D = !Q1 & Q0 & INA;
APPEND Q0.D = !Q1 & Q0 & INA & !INB;

```

#### D-CE タイプフリップフロップ

```

APPEND Q1.D = !Q1 & Q0 & INA;
APPEND Q1.CE = !Q1 & Q0 & INA;
APPEND Q0.D = 'b'0;
APPEND Q0.CE = !Q1 & Q0 & !INA
# !Q1 & Q0 & INA & INB;

```

#### J-K タイプフリップフロップ

```

APPEND Q1.J = !Q1 & Q0 & INA;
APPEND Q1.K = 'b'0;
APPEND Q0.J = 'b'0;
APPEND Q0.K = !Q1 & Q0 & INA & INB
           # !Q1 & Q0 & !INA;

```

### S-R タイプフリップフロップ

```

APPEND Q1.S = !Q1 & Q0 & INA;
APPEND Q1.R = 'b'0;
APPEND Q0.S = 'b'0;
APPEND Q0.R = !Q1 & Q0 & INA & INB
           # !Q1 & Q0 & !INA;

```

### T タイプフリップフロップ

```

APPEND Q1.T = !Q1 & Q0 & INA;
APPEND Q0.T = !Q1 & Q0 & !INA
           # !Q1 & Q0 & INA & INB;

```

### 無条件の同期出力命令

この命令は、現在の状態から次の状態へのトランジションを表わします。また、トランジションに関連するレジスタード（同期）出力を指定します。さらに、変数が論理的に宣言されるかどうかを定義します。フォーマットを以下に示します。

```

PRESENT state_n
NEXT state_n OUT [!]var... OUT [!]var;

```

#### ここで

state\_n はステートマシンの出力であるステートビットのデコードされた値（デフォルトでは16進数）です。

var はピン宣言で宣言された変数名です。SEQUENCE state\_var\_list からの変数ではありません。

!は補助演算子です。これを使用して変数を論理的に否定できます。また、省略すると変数をそのまま宣言します。

;は命令の終わりを表わすセミコロンです。

かぎ括弧 [] は省略可能なアイテムを表します。

ピン宣言命令 PinDeclarationStatements により、宣言された時に変数がアクティブハイかアクティブローかが決まります。例えば、変数に否定のマーク（!var）がピン宣言で付けられている場合、OUT 命令でこの変数が使用される場合、その値は、アクティブローになります。

否定モードは D-CE タイプ、J-K タイプ、T タイプ、S-R タイプのフリップフロップに使用して下さい。D タイプのフリップフロップは、使われかたが指定されない場合、リセットされます。

以下に無条件の同期出力命令の例を示します。

```
PRESENT 'b'01
NEXT 'b'10 OUT Y OUT !Z ;
```

図 9-67 に上記の例で記述されたランジションと出力変数の定義を示します。

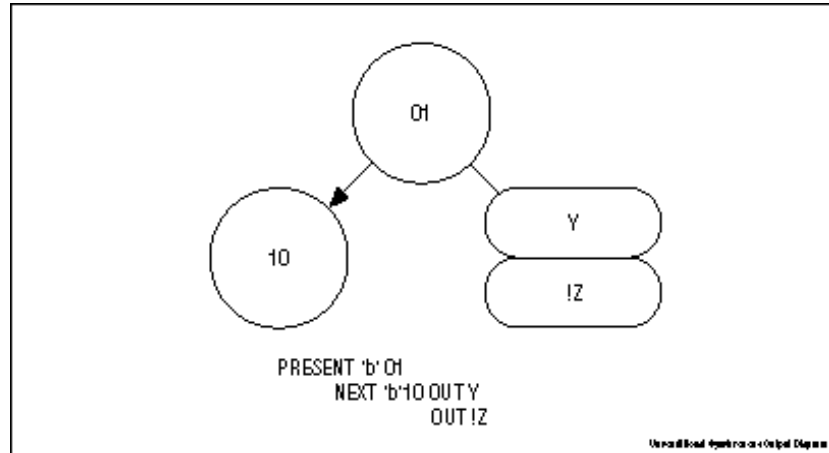


図 9-67 無条件の同期出力ダイアグラム

上記の例と図の同期出力定義では、コンパイラは、指定されたフリップフロップのタイプに応じて以下の式を生成します。

#### D タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0;
```

(not defined for Z output)

#### D-CE タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0;
APPEND Y.CE = !Q1 & Q0;
APPEND Z.D = 'b'0;
APPEND Z.CE = !Q1 & Q0;
```

#### J-K タイプフリップフロップ

```
APPEND Y.J = !Q1 & Q0;
APPEND Y.K = 'b'0;
APPEND Z.J = 'b'0;
APPEND Z.K = !Q1 & Q0;
```

#### S-R タイプフリップフロップ

```
APPEND Y.S = !Q1 & Q0;
APPEND Y.R = 'b'0;
APPEND Z.S = 'b'0;
APPEND Z.R = !Q1 & Q0;
```

#### T タイプフリップフロップ

```
APPEND Y.T = !Q1 & Q0;
```

```
APPEND Z.T = !Q1 & Q0;
```

### 条件付き同期出力命令

この命令は、現在の状態から次の状態へのトランジションを表わします。また、トランジションに関連するレジスタード（同期）出力を指定します。さらに、入力の式で指定された条件が合うと、変数が論理的に宣言されるかどうかを定義します。フォーマットを以下に示します。

```
PRESENT state_n
IF expr NEXT state_n OUT [!]var...OUT [!]var;
.
.
IF expr NEXT state_n OUT [!]var...OUT [!]var;
[ [DEFAULT] NEXT state_n OUT [!]var;
```

ここで

*state\_n* はステートマシンの出力であるステートビットのデコードされた値（デフォルトでは16進数）です。

*var* はピン宣言で宣言された変数名です。SEQUENCE *state\_var\_list* からの変数ではありません。

!*var*は補助演算子です。これを使用して変数を論理的に否定できます。また、省略すると変数をそのまま宣言します。

;*var*は命令の終わりを表わすセミコロンです。

*expr* は任意の表現です。

かぎ括弧 [ ] は省略可能なアイテムを表わします。

ピン宣言命令 *PinDeclarationStatements* により、宣言された時に変数がアクティブハイかアクティブローかが決まります。例えば、変数に否定のマーク (!*var*) がピン宣言で付けられている場合、OUT 命令でこの変数が使用される場合、その値は、アクティブローになります。

否定モードは D-CE タイプ、J-K タイプ、T タイプ、S-R タイプのフリップフロップに使用して下さい。D タイプのフリップフロップは、使われかたが指定されない場合、リセットされます。

DEFAULT 命令は省略可能です。この命令は、指定された条件命令に合うものがない場合、現在の状態から次の状態へのトランジションを示します。言い換えると、条件命令の補助的な役割をします。

DEFAULT 命令を使用するときは、気を付けて下さい。DEFAULT 命令は、他の条件の補助的な命令なので、DEFAULT 命令により式が複雑になりコンパイラの動作が遅くなる場合があります。多くのアプリケーションでは、DEFAULT 命令の代わりに一つか二つの条件命令が指定されます。

以下に、DEFAULT 命令を使用しない条件付き同期出力命令の例を示します。

```
PRESENT 'b'01
```

```
IF INA NEXT 'b'10 OUT Y;
IF !INA NEXT 'b'11 OUT Z;
```

図 9-68 に上記例の命令で定義されるトランジションと出力を示します。

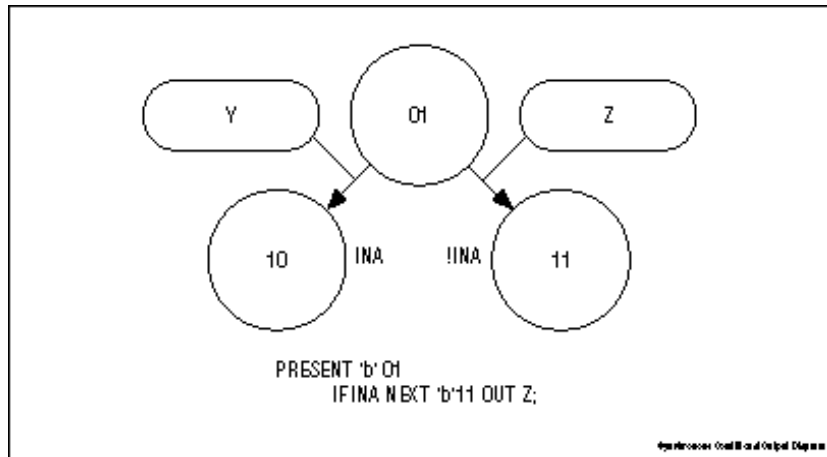


図 9-68 同期条件付き出力ダイアグラム

上記の例と図の同期出力定義では、コンパイラは、指定されたフリップフロップに応じて以下の式を生成します。

#### D タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0 & INA;
APPEND Z.D = !Q1 & Q0 & !INA;
```

#### D-CE タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0 & INA;
APPEND Y.CE = !Q1 & Q0 & INA;
APPEND Z.D = !Q1 & Q0 & !INA;
APPEND Z.CE = !Q1 & Q0 & !INA;
```

#### J-K タイプフリップフロップ

```
APPEND Y.J = !Q1 & Q0 & INA;
APPEND Y.K = 'b'0;
APPEND Z.J = !Q1 & Q0 & !INA;
APPEND Z.K = 'b'0;
```

#### S-R タイプフリップフロップ

```
APPEND Y.S = !Q1 & Q0 & INA;
APPEND Y.R = 'b'0;
APPEND Z.S = !Q1 & Q0 & !INA;
APPEND Z.R = 'b'0;
```

#### T タイプフリップフロップ

```
APPEND Y.T = !Q1 & Q0 & INA;
APPEND Z.T = !Q1 & Q0 & !INA;
```

以下に DEFAULT 命令を使用した条件付き出力命令の例を示します。

```
PRESENT 'b'01
  IF INA & INB NEXT 'b'10;
  IF INA & !INB NEXT 'b'11;
  DEFAULT NEXT 'b'00 OUT Y
  OUT !Z;
```

図 9-69 に上記の例により記述されたトランジションを示します。DEFAULT 命令により生成された式に注意して下さい。

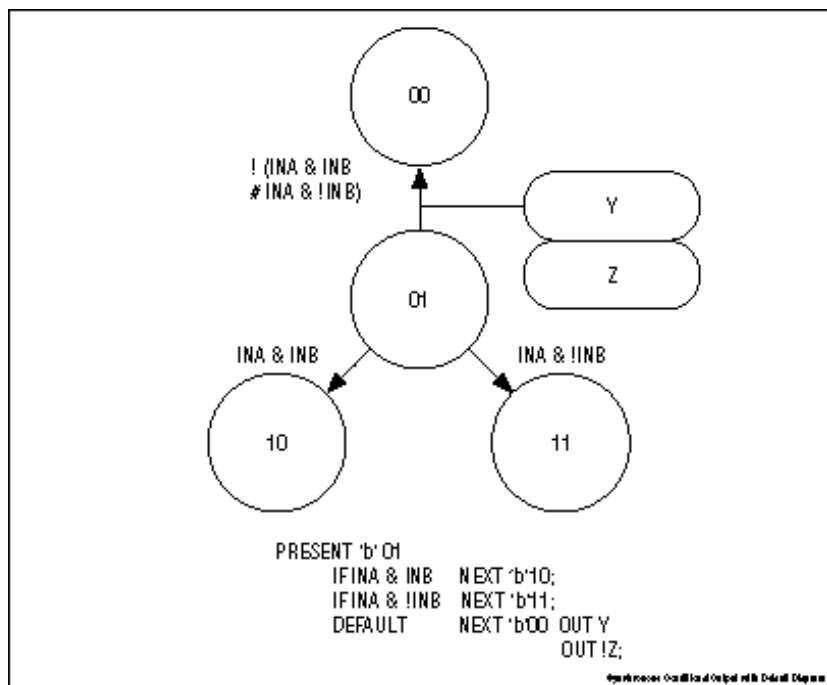


図 9-69 デフォルトを使用した同期条件付き出力ダイアグラム

上記の例と図で示されたトランジションでは、コンパイラは、指定されたフリップフロップのタイプに応じて以下の式を生成します。

#### D タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0 & !INA;
```

(not defined for Z output)

#### D-CE タイプフリップフロップ

```
APPEND Y.D = !Q1 & Q0 & !INA;
APPEND Y.CE = !Q1 & Q0 & !INA;
APPEND Z.D = 'b'0;
APPEND Z.CE = !Q1 & Q0 & INA;
```

#### J-K タイプフリップフロップ

```
APPEND Y.J = !Q1 & Q0 & !INA;
```



```

APPEND Y.K = 'b'0;
APPEND Z.J = 'b'0;
APPEND Z.K = !Q1 & Q0 & !INA;

```

### S-R タイプフリップフロップ

```

APPEND Y.S = !Q1 & Q0 & !INA;
APPEND Y.R = 'b'0;
APPEND Z.S = 'b'0;
APPEND Z.R = !Q1 & Q0 & !INA;

```

### T タイプフリップフロップ

```

APPEND Y.T = !Q1 & Q0 & !INA
APPEND Z.T = !Q1 & Q0 & INA;

```

## 無条件の非同期出力命令

与えられた現在の条件に関連するノンレジスタード（非同期）出力の変数をこの命令により指定します。また、変数が論理的に使用される場合を定義します。フォーマットを以下に示します。

```

PRESENT state_n
OUT var ... OUT var ;

```

ここで

state\_n はステートマシンの出力であるステートビットのデコードされた値（デフォルトでは16進数）です。

var はピン宣言で宣言された変数名です。SEQUENCE state\_var\_list からの変数ではありません。

;は命令の終わりを表わすセミコロンです。

ピン宣言命令 PinDeclarationStatements により、宣言された時に変数がアクティブハイかアクティブローかが決まります。例えば、変数に否定のマーク（!var）がピン宣言で付けられている場合、OUT 命令でこの変数を使用される場合、その値は、アクティブローになります。

この命令では（補助演算子を用いて）変数を否定することはできません。

現在の状態それぞれに対して、出力命令を一つだけ記述することができます。複数の OUT キーワードを使用すると複数の変数を定義することができます。

以下に無条件の非同期出力命令の例を示します。

```

PRESENT 'b'01
OUT Y OUT Z;

```

図 9-70 に上記の例の命令で定義された出力を示します。

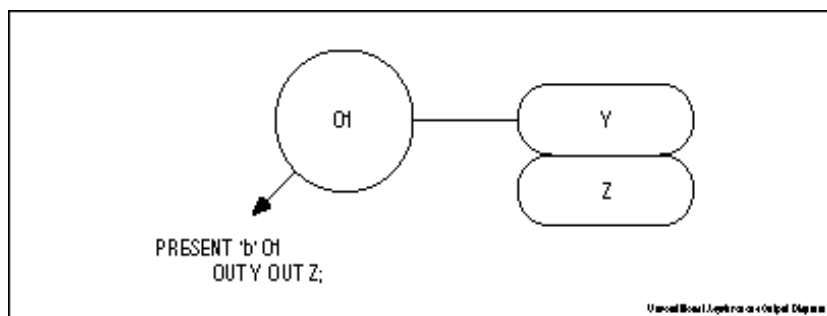


図 9-70 無条件の非同期出力ダイアグラム

上記の例と図の非同期出力定義では、コンパイラは以下の式を生成します。

```
APPEND Y = !Q1 & Q0;
APPEND Z = !Q1 & Q0;
```

### 条件付き非同期出力命令

この命令により、与えられた状態に関連するノンレジスタード（非同期）出力の変数を指定します。さらに、入力の式で指定された条件が合うと、いつ変数が論理的に宣言されるかを定義します。フォーマットを以下に示します。

```
PRESENT state_n
IF expr OUT var ... OUT var;
.
.
IF expr OUT var ... OUT var;
[DEFAULT OUT var ... OUT var;]
```

ここで

state\_n はステートマシンの出力であるステートビットのデコードされた値（デフォルトでは16進数）です。

var はピン宣言で宣言された変数名です。SEQUENCE 命令からの変数ではありません。

expr は任意の表現です。

;は命令の終わりを表わすセミコロンです。

かぎ括弧 [ ] は省略可能なアイテムを表わします。

ピン宣言命令 PinDeclarationStatements により、宣言された時に変数がアクティブハイかアクティブローかが決まります。例えば、変数に否定のマーク (!var) がピン宣言で付けられている場合、OUT 命令でこの変数が使用される場合、その値は、アクティブローになります。

この命令では、（補助演算子を用いて）変数を否定することはできません。

現在の状態それぞれに対して、出力命令を一つだけ記述することができます。複数の OUT キーワードを使用すると複数の変数を定義することができます。

ます。

DEFAULT 命令は省略可能です。この命令は、指定された条件命令に合うものがない場合、現在の状態から次の状態へのトランジションを示します。言い換えると、条件命令の補助的な役割をします。

DEFAULT 命令を使用するときは、気を付けて下さい。DEFAULT 命令は、他の条件の補助的な命令なので、DEFAULT 命令により式が複雑になりコンパイラの動作が遅くなる場合があります。多くのアプリケーションでは、DEFAULT 命令の代わりに一つか二つの条件命令が指定されます。

以下に DEFAULT 命令を使用しない条件付き非同期出力命令の例を示します。

```
PRESENT 'b' 01
IF INA OUT Y;
IF !INA OUT Z;
```

図 9-71 に上記の例の命令により定義される出力を示します。

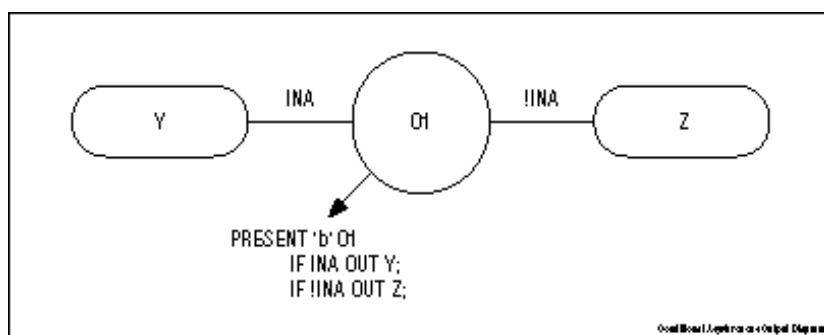


図 9-71 条件付き非同期出力ダイアグラム

上記の例と図の非同期出力定義では、コンパイラは以下の式を生成します。

```
APPEND Y = !Q1 & Q0 & INA;
APPEND Z = !Q1 & Q0 & !INA;
```

以下に DEFAULT 命令を使用した条件付き非同期出力命令の例を示します。

```
PRESENT 'b' 01
IF INA & INB OUT X;
IF INA & !INB OUT Y;
DEFAULT OUT Z;
```

図 9-72 に上記の例で示されるトランジションを示します。DEFAULT 命令により生成される式に注意して下さい。

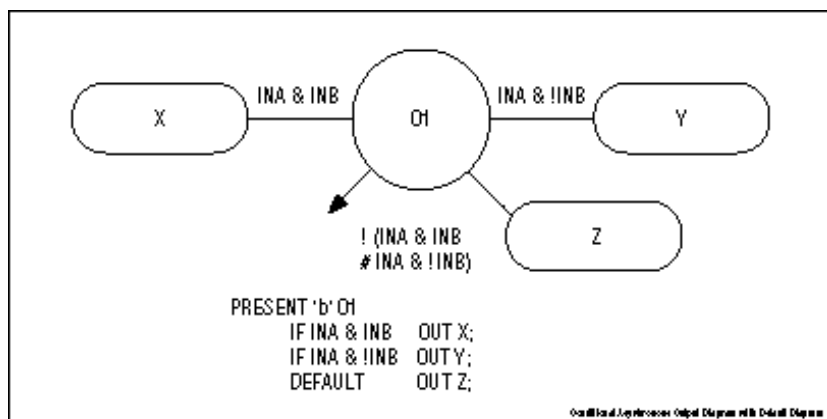


図 9-72 デフォルトを使用する条件付き非同期出力ダイアグラム

上記の例と図で示されるトランジションでは、コンパイラは、指定されるフリップフロップのタイプに応じて以下の式を生成します。

```

APPEND X = !Q1 & Q0 & INA & !INB;
APPEND Y = !Q1 & Q0 & INA & INB;
APPEND Z = !Q1 & Q0 & !INA;

```

## ステートマシンシンタクスファイルのサンプル

このセクションでは、ステートマシンシンタクスを用いて実行される簡単な 2 ビットカウンタの例を示します。

図 9-73 にカウンタ動作のダイアグラムを示します。

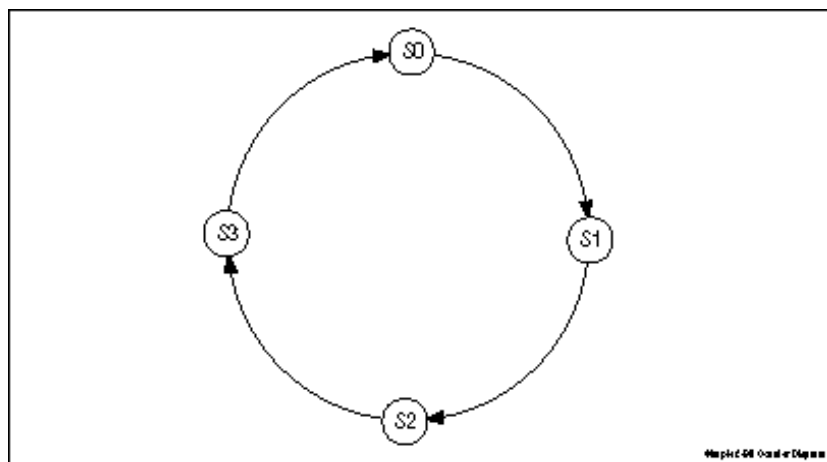


図 9-73 2 ビットカウンタのダイアグラム

\$DEFINE コマンドにより、カウンタの状態に名前を割り付けることができます。また、SEQUENCE 命令により、トランジションを定義できます。

```

$DEFINE S0 0 /* assign symbolic names */
$DEFINE S1 1 /* to states */
$DEFINE S2 2

```

```

$DEFINE S3 3
FIELD count = [Q1, Q0];
/* assign field variable to statebits */
SEQUENCE count {
PRESENT S0 NEXT S1 ;
PRESENT S1 NEXT S2 ;
PRESENT S2 NEXT S3 ;
PRESENT S3 NEXT S0 ;
}

```

ステートマシンの実行に関する詳細はセクション U5 の、Decade Up/Down Counter の例を参照して下さい。

### 条件シンタクス

CONDITION シンタクスにより、組み合わせ論理の標準のブーリアン論理式を記述するよりもより高レベルのアプローチで論理関数を指定することができます。フォーマットを以下に示します。

```

CONDITION {
IF expr0 OUT var ;
.
.
IF exprn OUT var ;
DEFAULT OUT var ;
}

```

ここで

expr は任意の式です。

var はピン宣言で宣言される変数名です。リストの表記のインデックス付きあるいはインデックスの無い変数のリストも指定できます。

;}は命令の終わりを示すセミコロンです。

CONDITON シンタクスは、ステートマシンシンタクスの非同期条件付き出力命令と等価です。ただし、CONDITION シンタクスは特定の状態を参照することはありません。式または、DEFAULT の状態になると論理的に式が宣言されます。

このフォーマットでは論理否定を変数に使用することができません。

DEFAULT 命令を使用する場合は注意して下さい。DEFAULT 命令は、他の条件の補助的な命令なので、DEFAULT 命令により式が複雑になりコンパイラの動作が遅くなる場合があります。多くのアプリケーションでは、DEFAULT 命令の代わりに一つか二つの条件命令が指定されます。

CONDITION シンタクスを使用する2線から4線へのデコーダの例を示します。ENABLE 信号が入力されると、2本のデータ入力 A と B により4本のデコード出力 Y0 から Y3 の一つを指定します。4本の出力に真が無い場合、NO\_MATCH 出力が出力されます。

```

PIN [1,2] = [A,B] ;          /* Data Inputs */
PIN 3 = !enable ;           /* Enable Input */
PIN [12..15] = [Y0..3] ;    /* Decoded Outputs */
PIN 14 = no_match ;         /* Match Output */
CONDITION {
  IF enable & !B & !A out Y0 ;
  IF enable & !B & A out Y1 ;
  IF enable & B & !A out Y2 ;
  IF enable & B & A out Y3 ;
}

```

上記例の DEFAULT 式は以下の論理式と等価です。

```

no_match = !( enable & !B & !A)
# enable  &  !B  &  A
# enable  &   B  & !A
# enable  &   B  &  A ;

```

以下のように縮小できます。

```

no_match = !enable ;

```

## ユーザ定義関数

FUNCTION キーワードにより、あるロジックを関数としてまとめて個人的なキーワード作成しそれに名前を付けることができます。論理式の中でその名前を関数として使用することができます。ユーザ定義関数のフォーマットを以下に示します。

```

FUNCTION name ([parameter0,...,parameterN])
{
  body
}

```

ここで

name は関数を参照するために使用する名前です。CUPL の予約キーワードは使用しないで下さい。

parameter は関数が論理式の中で使用される場合、引き数として使用される省略可能な変数です。

body は任意の組み合わせ論理で、真理値表やステートマシンシンタックス、条件シンタックス、ユーザ関数を記述できます。

()は、パラメータリストを囲む括弧です。

{ }は関数の本体を囲む中括弧です。

かぎ括弧 [] は省略可能なアイテムを示す括弧です。

本体の中の命令は関数の式として割り付けられます。

引き数を使用する場合、関数定義と関数が呼び出しでのパラメータの数は、同じになる必要があります。関数の本体で定義されるパラメータは論理式で参照されるパラメータに置き換えられます。

例えば以下に排他的 OR 関数を定義します。

```

FUNCTION xor(in1, in2) {
/* in1 と in2 はパラメータです */
xor = in1 & in2 # !in1 & in2 ;
}

```

xor は以下のように引き数として入力 A と B を用いて使用されます。

```
Y = xor(A,B) ;
```

以下の論理式が結果として出力変数 Y に割り付けられます。

```
Y = A & !B # !A & B ;
```

関数が式の中で使用される場合、コンパイラは以下のように動作します。

1. 関数名や引き数に特別な関数呼び出し変数が割り付けられます。この変数名はユーザが使用することができません。
2. 残りの式が評価されます。
3. 呼び出しパラメータを使用する関数が評価されます。
4. 関数呼び出し変数は関数本体により式に割り付けられます。本体の命令で割り付けが行われない場合、関数呼び出し変数は ho の変数に割り付けられます。

関数は、それらが参照される前に定義される必要があります。関数は再帰的に使用することはできません。すなわち、関数本体にその関数を呼び出す式を記述することはできません。

以下の例は、内部フィードバックを使用しないノンレジスタードデバイスのステートマシンタイプトランジションを構築するユーザ定義関数を示します。

```

FUNCTION TRANSITION(present_state,
next_state,
input_conditions ) {
APPEND state_out = state_in:present_state &
input_condition &
next_state;
}

```

上記の例で定義される関数は以下の例で使用され、簡単なアップ / ダウンカウンタを実行します。

```

PIN [10,11] = [Qin0..1]; /* Registered PROM */
/*output feed */
/*back externally */
/*on input pins */
PIN [12,13] = [count0..1] ; /*Count Control */
PIN [1,2] = [Q0..1] ; /* PROM Outputs */
FIELD state_in = [Qin0..1] ;
FIELD state_out = [Q0..1] ;
count_up = !count1 & !count0 ; / * count up */
count_dn = !count1 & count0 ; /* count down */

```

```

hold_cnt = count1;          /* hold count */
$DEFINE STATE0 'b'00
$DEFINE STATE1 'b'01
$DEFINE STATE2 'b'10
$DEFINE STATE3 'b'11
/* (transition function definition made here)*/
TRANSITION (STATE0, STATE1, count_up) ;
TRANSITION (STATE1, STATE2, count_up) ;
TRANSITION (STATE2, STATE3, count_up) ;
TRANSITION (STATE3, STATE0, count_up) ;
TRANSITION (STATE0, STATE3, count_dn) ;
TRANSITION (STATE1, STATE0, count_dn) ;
TRANSITION (STATE2, STATE1, count_dn) ;
TRANSITION (STATE3, STATE2, count_dn) ;
TRANSITION (STATE0, STATE0, hold_cnt) ;
TRANSITION (STATE1, STATE1, hold_cnt) ;
TRANSITION (STATE2, STATE2, hold_cnt) ;
TRANSITION (STATE3, STATE3, hold_cnt) ;

```



## 設計のシミュレーション

このセクションでは、どのようにテスト仕様ソースファイルを作成しプログラマブルロジックデバイスのシミュレーションを実行するかを説明します。テストベクタにより、出力を入力の変数として定義し PLD の変数の動作を指定します。テストベクタは、プログラム前のデバイスロジックのシミュレーションと論理がプログラムされた後のデバイスのテスト変数とに使用されます。アドバンスド PLD シミュレータは、JEDEC 互換のテストベクタを生成します。このテストベクタは、コンパイル中に作成される JED ファイルに添付されます。

### シミュレータへの入力

テスト仕様ソースファイル(ファイル名.SI)は、シミュレータへの入力です。このファイルには、回路でのデバイスに機能的に必要な情報が記述されています。

ソースファイルに入力される入力ピンの刺激や出力ピンのテストは、PLD ソースファイルで計算される実際の値と比較されます。計算された値は、アブソリュート ABS フォーマットが指定される場合、コンパイル中に作成されるアブソリュートファイル(ファイル名.ABS)に保存されます。アブソリュートファイルは、シミュレーションを実行する前にコンパイル中に作成する必要があります。

### シミュレータからの出力

シミュレータの出力は、

- シミュレーションリスティングファイル
- JEDEC フォーマットのダウンロード可能なヒューズリンクファイルに添付されるベクタです。

シミュレーションリスティングファイル(ファイル名.SO)にはシミュレーションの結果が保存されます。ファイル名は、入力の変数仕様ファイルと同じ名前です。

ヘッダー情報は、適当に印が付けられたヘッダーエラーと一緒にリスティングファイルに記述されます。失敗した出力テストには、実際の出力(シミュレータが決めた)値と一緒にフラグが付けられます。エラーの各変数は予測された(ユーザが与えた)値と一緒に表示されます。無効または予測されないテスト値は適当なエラーメッセージと一緒に表示されます。

シミュレーションリスティングファイルは ASCII です。したがって、Text Expert でオープンすることができます。EDA/クライアントの Waveform Editor で SO ファイルを開くと、シミュレーション結果を、一連の波形として表示することもできます。波形の表示については、A Quick Tour of

Advanced PLD のセクションを参照して下さい。

シミュレータは、コンパイル中に作成された既存の JEDEC ヒューズリンク ファイル(ファイル名.SI)にテストベクタを追加します。Configure Advanced PLD ダイアログボックスの Format Tab の JEDEC オプションをイネーブル にして、このファイルを作成して下さい。

シミュレータは、複数のデバイスのシミュレーションをサポートしていません。この場合、デバイスファイルの最初のファイルがシミュレーションされます。

## シミュレーションテストベクタファイルの作成

テスト仕様ファイル(ファイル名.SI)は ASCII ファイルです。Text Expert を使用して作成して下さい。ファイル名は、対応する CUPL 論理記述ソース ファイルと同じ名前です。ただし、ファイルの拡張子は違います。以下の情報をテスト仕様ファイルに記述して下さい。

- ヘッダー情報
- コメント
- 変数オーダー
- 基本セット
- テストベクタ
- シミュレータディレクティブ

シミュレーションテストベクタファイルの作成については A Quick Tour of Advanced PLD セクションを参照して下さい。

## ヘッダー情報

入力されるヘッダー情報は、対応する CUPL 論理記述ファイル(.PLD)の情報と一致している必要があります。ヘッダー情報が一致していない場合、ワーニングメッセージが表示され、論理式の状態がテスト仕様ファイルの現在のテストベクタと一致していないことが知らされます。

表 10-1 にヘッダー情報に使用されるキーワードの一覧を示します。

---

PARTNO
NAME
REVISION
DATE
DESIGNER
COMPANY
ASSEMBLY
LOCATION
DEVICE
FORMAT

---

表 10-1 シミュレータソースファイルヘッダー情報

テスト使用ファイルを作成する場合、対応する CUPL ソースファイルの内容をコピーして作成すると間違いの無いヘッダー情報を記述できます。

## デバイスの指定

シミュレータは、デバイスライブラリファイル(CUPL.DL)でデバイス情報にアクセスします。ライブラリには、各デバイスの内部アーキテクチャやピンの数、使用できるレジスタのタイプ、(レジスタードとノンレジスタードピンなどの)論理特性、フィードバック機能、レジスタのパワーオン状態、レジスタ制御機能などの物理特性が記述されています。

デバイスニーモニックを使用してターゲットデバイスを参照して下さい。各ニーモニックは、デバイスファミリプリフィックスと業界標準部品番号プリフィックスで構成されます。表 10-2 にデバイスニーモニックプリフィックスを示します。

プリフィックス	デバイスファミリ
EP	イレーサブルプログラマブルロジックデバイス(EPLD)
G	ジェネリックアレイロジック(GAL)
F	フィールドプログラマブルロジックアレイ(FPLA)
F	フィールドプログラマブルゲートアレイ(FPGA)
F	フィールドプログラマブルロジックシーケンサ(FPLS)
F	フィールドプログラマブルシーケンスジェネレータ(FPSG)
P	プログラマブルロジックアレイ(PAL)
P	プログラマブルロジックデバイス(PLD)
P	プログラマブルエレクトロニクスイレーサブルロジック(PEEL)
PLD	スードロジカルデバイス
RA	バイポーラプログラマブルリードオンリメモリ(PROM)

表 10-2 デバイスニーモニックプリフィックス

例 : PAL10L8 のデバイスニーモニックは、P10L8 です。また、82S100 のデバイスニーモニックは、F100 です。バイポーラ PROM では、サフィックスがアレイサイズを表わします。

例 : 1024X8 のバイポーラ PROM のデバイスニーモニックは、10 本のアドレス入力ピンと 8 本のデータ出力ピンがあるので RA10P8 になります。

## コメント

テスト仕様ファイルの任意の位置にコメントを置くことができます。コメ

ントを仕様して仕様ファイルの内容やテストベクタの関数の説明を記述することができます。コメントは、スラッシュ - アスタリスク(/\*)で始まり、アスタリスク - スラッシュ(\*)で終わります。複数行に渡りコメントを設定することができます。コメントをネストすることはできません。

## ステートメント

シミュレータには、キーワード ORDER, BASE, VECTORS があり、これらを使用してシミュレータ出力や出力の表示の仕方をソースファイルに記述します。以下のセクションで、CUPL キーワードによりステートメントをどのように記述するかを説明します。

### ORDER ステートメント

ORDER キーワードを使用して、シミュレーションテーブルで使用される変数の一覧を表示して下さい。また、それらがどのように表示されるかを定義して下さい。通常、変数名は、CUPL 論理記述ファイルと同じ名前が使用されます。ORDER の後ろにコロンを記述し、リストの変数をコンマでくぎって下さい。さらに、リストの最後にセミコロンを記述して下さい。以下に ORDER 命令の使用例を示します。

```
ORDER: inputA, inputB, output ;
```

セミコロンの左で実際に使用された変数だけが表示されます。

変数名の極性は、CUPL 論理記述ファイルで宣言されたものと違います。これにより、アクティブハイのシミュレーションベクタを使用してアクティブロー出力のシミュレーションができます。変数名は、任意の順番で入力できます。シミュレータは、デバイスの JEDEC ダウンロードフォーマットに応じてシミュレーション結果の極性や変数の適当な順番を自動的に作成します。インデックス付き変数を ORDER 命令で使用する場合、リスト表記で記述して下さい。しかし、ORDER 命令はすでにリスト形式になっているので、ORDER の組みを囲むかぎ括弧([])は必要ありません。以下に二つの等価な ORDER 命令の例を示します。最初の命令は、変数をすべて記述し、2 番目はリスト形式で記述しています。

```
ORDER: A0, A1, A2, A3, SELECT, !OUT0, !OUT1;  
ORDER: A0..3, SELECT, !OUT0..1 ;
```

リスト表記フォーマットでは、最初のインデックス付き変数(上記例の!OUT0)の極性により、リスト全体の極性が決まります。CUPL 論理記述ファイルで宣言されるビットフィールドは、一つの変数名で参照することができます。また、ビットフィールドは、FIELD 宣言命令を使用して、シミュレータのテスト仕様ファイルで宣言することもできます。FIELD 命令は、ORDER 命令より先に記述する必要があります。

ORDER 命令を使用して、シミュレータリスティングファイルのシミュレーション結果ベクタのフォーマットを指定できます。デフォルトでは、変数値はコロンとコロンの間にスペースがありません。ORDER 命令の例を以下に示します。

```
ORDER: clock, input, output ;
```

により出力ファイルに以下のように記述されます。

```
0001: C0H
```

```
0002: C1L
```

%記号と 1 から 80 までの 10 進数を使用してコロンの間にスペースを挿入できます。例えば、以下の ORDER 命令により

```
ORDER: clock, %2, input, %2, output ;
```

出力ファイルに以下のように記述されます。

```
0001: C 0 H
```

```
0002: C 1 L
```

ORDER 命令は、セミコロンで終了する必要があります。

ORDER 命令の中にダブルクォート(" ")で囲まれた文字列を記述すると、出力ファイルにテキストを記述できます。例えば、以下の ORDER 命令により

```
ORDER: "Clock is ", clock,  
      " and input is ", input,  
      " output goes ", output ;
```

以下の結果が出力ファイルに記述されます。

```
0001: Clock is C and input is 0 output goes H
```

```
0002: Clock is C and input is 1 output goes L
```

## 多重 ORDER 命令

いろいろな ORDER 命令を SI ファイルに定義することができます。例えば、TEST.SI ファイルに以下の記述がある場合、

```
Name      test;  
Partno     XXXXX;  
Date       XX/XX/XX;  
Revision   XX;  
Designer   XXXXX;  
Company    XXXXX;  
Assembly   XXXXX;  
Location   XXXXX;  
Device     gl6v8;  
  
Order: A, %1, B, %1, X, %1, Y;  
Vectors:  
  0 0 H L  
  0 1 H H  
  1 0 H H  
  1 1 L L  
  0 X H X  
  X 0 H X
```

```

1 X X X
X 1 X X
Order: A, B, X;
Vectors:
0 0 H
0 1 H
1 0 H
1 1 L
0 X H
X 0 H
1 X X
X 1 X

```

図 10-1 TEST.SI

TEST.SO ファイルは以下のようになります。

```

CSIM: CUPL Simulation Program
Version 4.2a Serial# ...
Copyright (c) 1996 Protel International
CREATED Wed Dec 04 02:14:12 1991
LISTING FOR SIMULATION FILE: test.si
1: Name      test;
2: Partno    XXXXX;
3: Date      XX/XX/XX;
4: Revision  XX;
5: Designer  XXXXX;
6: Company   XXXXX;
7: Assembly  XXXXX;
8: Location  XXXXX;
9: Device    gl6v8;
10:
11: Order: A, %1, B, %1, X, %1, Y;
12:
=====
      A B X Y
=====
0001: 0 0 H L
0002: 0 1 H H
0003: 1 0 H H
0004: 1 1 L L
0005: 0 X H X
0006: X 0 H X
0007: 1 X X X
0008: X 1 X X
25: Order: A, B, X; 26:
=====
      ABX
=====
0010: 00H

```

```
0011: 01H
0012: 10H
0013: 11L
0014: 0XH
0015: X0H
0016: 1XX
0017: X1X
```

図 10-2 TEST.SO

## BASE ステートメント

多くの場合、(FIELD 変数を除く)ORDER 命令の各変数には、出力ファイルのテストベクタテーブルに記述される一つのキャラクタテスト値があります。多重テストベクタ値を引用符で囲まれた番号で表わすことができます。入力値には、シングルクォートを使用し、出力値にはダブルクォートを使用して下さい。BASE ステートメントを入力し、引用符で囲まれた数値をどのように展開するかを指定して下さい。BASE ステートメントのフォーマットを以下に示します。

```
BASE: name;
```

ここで

name は octal か decimal、hex のどれかです。

BASE のうしろにはコロンを記述します。

ベースステートメントは、セミコロンで終了して下さい。

引用符で囲まれたテスト値のデフォルトの基数は 16 進数です。BASE ステートメントは ORDER 命令より前に記述する必要があります。

基数が 10 進数または 16 進数の場合、引用符で囲まれた値は 4 桁に展開されます。基数が 8 進数の場合、3 桁に展開されます。例えば、'7'と入力されたテストベクタは以下ようになります。

```
1 1 1      基数が 8 進数
```

または

```
0 1 1 1    基数が 10 進数
```

または

```
0 1 1 1    基数が HEX
```

16 進数や 8 進数の複数の桁が引用符の間に入力されます。例えば、'563'は以下のように展開されます。

```
1 0 1 1 1 0 0 1 1      基数は 8 進数
```

または

```
0 1 0 1 0 1 1 0 0 0 1 1  基数は 10 進数
```

または

0 1 0 1 0 1 1 0 0 0 1 1 基数は hex

引用符で囲まれた値は、他のテスト値で使用することもできます。例えば、基数が 8 進数に設定されている場合

"XX" X X X X X X に展開されます。

"LL" L L L L L L に展開されます。

"45" H L L H L H に展開されます。

引用符で囲まれる値に\*は使用できません。

FIELD 変数のテスト値は、個別(例えば、001、HHLL)に記述することもできるし、引用符で囲まれた値(例えば、'1'、"C")で記述することもできます。引用符で囲まれた値が使用される場合、値は自動的にフィールドの値に展開されます。例えば、以下のアドレスフィールド

FIELD address = [A0..5] ;

に以下のテスト値が記述されている場合、

```
/*
A      A      A      A      A      A
5      4      3      2      1      0
-----*/
1      1      1      0      0      1
```

上記の値は、引用符で囲まれた一つのテスト値、'39'を使用して記述することができます。

## VECTORS 命令

VECTORS キーワードを使用して、ベクタテーブルを前もって設定して下さい。以下のキーワードは、一つのテストベクタまたは引用符で囲まれたテスト値に付けて使用されます。表 10-3 に使用できるテストベクタの値を一覧表示します。

テスト値	説明
0	入力を LO(0 ボルト)に駆動します。(アクティブハイの入力を否定します。)
1	入力を HI(+5 ボルト)に駆動します。(アクティブハイ入力を宣言します。)
C	(CLOCK)入力を LO、HI、LO に駆動します。
K	(CLOCK)入力を HI、LO、HI に駆動します。
L	テスト出力 LO(0 ボルト)(アクティブハイ出力を否定します。)
H	テスト出力 HI(+5 ボルト)(アクティブハイ出力を宣言します。)
Z	ハイインピーダンスのテスト出力
X	入力が HI または LO、出力が HI または LO



N	テストされない出力
P	プリロード内部レジスタ(値は!Q に適用されます)
R	ランダム入力の生成
*	Outputs only - 出力のみの場合はシミュレータによりテスト値が決定され、ベクタ(シングルクオート)に入力されます。 引用符で囲まれた入力値は、指定された BASE(8 進数、10 進数、ヘキサ)で展開されます。有効な値は、0-F と X です。
""	(double quotes) (ダブルクオート)この引用符で囲まれた出力値は、指定された BASE(8 進数、10 進数、ヘキサ)で展開されます。有効な値は、0-F、H、L、Z、X です。

表 10-3 テストベクタ値

すべてのデバイスプログラマが X を入力に指定できる分けではありません。あるものは、0 としたり、あるものは、1 としたり、またあるものは、以前の値のままにしたりします。

以下にテストベクタ表の例を示します。

```
VECTORS:
0 0 1 1 1 'F' Z "H" /* テスト出力 HI */
0 1 1 0 0 '0' Z "L" /* テスト出力 LO */
```

## ランダム入力の生成

R ベクタにより、0 や 1 をどこでも出現させることができます。ランダム値があると、ランダム値(0 または 1)は、テストベクタの対応する信号で生成されます。

R を使用してランダム入力値を生成することができます。

An example of R in the SI file:

```
$repeat 10;
C 0 RRR 1RRRRRRR *****
```

上記の記述により、SO ファイルに以下の結果が出力されます。

```
0035: C 0      000 10001011      HLLLHLHH
0036: C 0      000 11100111      HHHLLHHH
0037: C 0      110 10111101      HHHHLHHL
0038: C 0      111 11000100      HLLLHLHL
0039: C 0      101 10001011      LHLHHHLL
0040: C 0      101 10000110      LLHHLHLL
0041: C 0      010 10000001      LHLLLLLL
0042: C 0      000 10010000      HLLHLLLL
0043: C 0      001 11110100      LHHHHLHL
0044: C 0      001 10011110      LHLLHHHH
```

## Don't Care

他のシミュレータと違い、アドバンスド PLD シミュレータは DONT-

CARE(X の状態)を扱うことができます。状態 X により、図 10-3 の真理値表に示されるルールに応じて、どの入力が出力に影響を与えているかを特定することができます。

NOT: ones complement !		AND &		
A	!A	A	B	A & B
0	1	0	0	L
1	0	0	1	L
X	X	0	X	L
		1	0	L
		1	1	H
		1	X	X
		X	X	X

OR #			XOR: exclusive OR \$		
A	B	A # B	A	B	A \$ B
0	0	L	0	0	L
0	1	H	0	1	H
0	X	X	0	X	X
1	0	H	1	0	H
1	1	H	1	1	L
1	X	H	1	X	X
X	X	X	X	X	X

図 10-3 ベクタ真理値表

## プリロード

デバイスが TTL レベルのプリロードピンを持っていない場合、レジスタードデバイスのクロックピンテスト値 P を使用して、ステートマシンの内部レジスタやカウンタをの設計をプリロードしたりできます。デバイスプログラマは、管理電圧を使用して実際にレジスタをロードします。デバイスの入力ピンは無視されます。したがって、X で定義されます。レジスタード変数に現れる値は、レジスタの!Q 出力にロードされます。以下に、レジスタ Q 出力とデバイスピンとの間に反転バッファを持つデバイスのアクティブロー出力変数のプリロードシーケンスの例を示します。

```
ORDER: clock, input1, input2 , !output ;
VECTORS:
P X X 1      /* reset flip-flop */
              /* !Q goes to 1 */
              /* Q goes to 0 */
0 X X H      /* output is HI due to */
              /* inverting buffer */
```

シミュレータでは、プリロード機能を持っていないデバイスでもプ

リロードテストベクタを作成したりシミュレーションを実行したりできます。しかし、PLD がすべて管理電圧を使用してプリロードを利用できるわけではありません。デバイスの中には、プリロードピンをこの目的のために提供しているものがあります。製造元が違くと特性も違うので、シミュレータでは、シミュレーション中のデバイスが実際にプリロードできるかどうかは確認できません。プリロード機能を使用する前に、テストされるデバイスが物理的にプリロードできるかどうかを確認する必要があります。

## クロック

ほとんどの同期デバイス(出力ピンに接続された共通のクロックを持つレジスタのあるデバイス)では、アクティブハイ(立ち上がりエッジでトリガーされる)クロックが使用されます。これらのデバイスに対してシミュレータの動作を適切に行うために、クロックピンにはCテスト値(1または0ではなく)を常に使用して下さい。アクティブロー(立ち下がりエッジでトリガーされる)クロックを使用する同期デバイスでは、Kテスト値をクロックピンに使用して下さい。

## 非同期ベクタ

非同期フィードバックのある回路のテストベクタを記述する場合、一度に2つのテストベクタを変更することにより、例外的な結果を生成するスパイク状態を作成することができます。(図 10-3 を参照。図 10-3 では、3つの入力[A,B,C]とフィードバックになる一つの出力Yを持つ回路のダイアグラムを示します。)

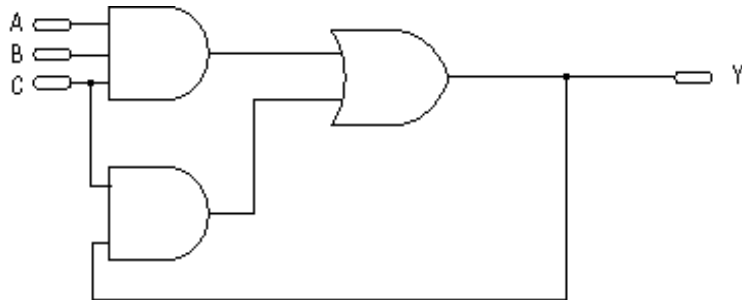


図 10-3 フィードバックを持つ回路

Y での出力の式を以下に示します。

$$Y = A \& B \& C \# C \& Y$$

図 10-5 のベクタテーブルに、指定された入力値に基づく、予測される出力を示します。

	A	B	C	Y
0001	0	0	0	L
0002	0	1	1	L
0003	1	0	1	L

Vector Table for Q1001 with Feedback

図 10-5 フィードバックのある回路のベクタテーブル

各ベクタで入力の一つが 0 であるので、A,B,C で定義される AND ゲートによりロー出力が生成されます。Y 出力からフィードバックされるロー出力は、他の AND ゲートをローに保ちます。したがって、(2 つの AND ゲートの出力により駆動される)OR ゲートと Y の出力は指定されたテストベクタに対してローのままです。

しかし、プログラマーがテストベクタを操作すると、最初のピンから順に値が入力されます。ベクタ間で 2 つのテスト値が変わるので、プログラマーにより(図 10-6 で"a"のラベルが付けられた)中間結果が作成されます。

	A	B	C	Y
0001	0	0	0	L
0001a	0	1	0	L
0002	0	1	1	L
0002a	1	1	1	H
0003	1	0	1	H

Vector Table with Intermediate Results

図 10-6 中間結果のあるベクタテーブル

中間結果[0002a]は、出力 Y にハイを生成します。このハイ信号がフィードバックされ、ベクタ[0003]の入力 C に指定される"1"の値に結び付けられ AND ゲートのハイ出力を生成します。そして、OR ゲートや Y 出力もハイになります。このハイ出力は 3 番目のテストベクタで指定される予想されたロー出力と矛盾します。したがって、結果はスパイク状態になります。

テストベクタ間では一つの値だけを変更することに注意すれば、上記のようなスパイク状態を避けることができます。また、ソース仕様ファイルで、シミュレータに中間結果を出力ファイルに出力するように指示する TARCE 値 1,2,3(デフォルトでは 0)を指定することができます。(以下のシミュレータディレクティブの"TRACE"を参照して下さい。)

## I/O ピンシミュレーション

入出力機能やコントロールできる出力イネーブル(OE)をもつ設計のテストベクタを記述する場合、I/O ピンに配置されるテストベクタは、出力イネーブルの値に依存します。出力イネーブルがアクティブの場合、I/O ピンには出力テスト値(L,H,\*,...)が必要になります。出力イネーブルがアクティブでなくなると、I/O ピンはハイインピーダンス状態になります。この時、入力テスト値(0,1,...)を I/O ピンに配置してピンを入力ピンとして使用することができます。出力イネーブルが再びアクティブになると、ピンのテスト値により、マクロセルの出力が影響を受けます。

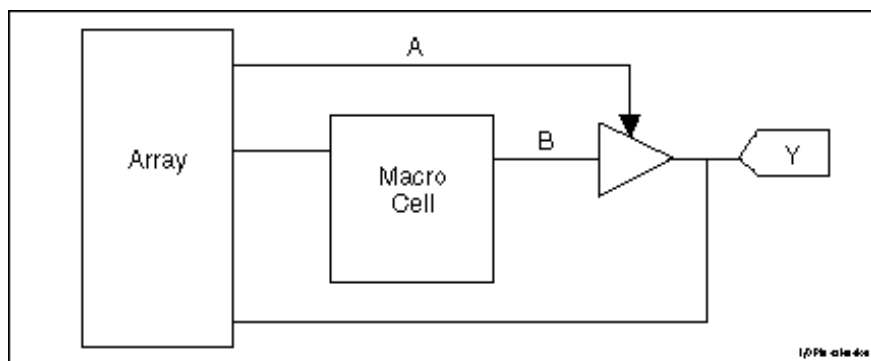


図 10-7 I/O ピンシミュレーション

以下の式は、図 10-7 のブーリアン式を記述したものです。

```
Y = B;
Y.OE = A;
```

A が真の場合、マクロセル(B)の出力がピン(Y)に現れます。A が偽の場合、出力イネーブルはアクティブでなくなりピン(Y)はハイインピーダンス状態になります。出力イネーブルがアクティブでなくなると、入力値をピンに配置することができます。シミュレーションファイルがどのようなかを示します。

```
Order: A, %1, B, %3, Y;

Vectors:
1 0 L /* OE is ON */
1 1 H
0 0 Z /* OE is OFF */
0 0 1 /* a valid input value can be placed on pin Y */
1 0 L /* OE is ON again */
```

## 変数宣言 (VAR)

Syntax: VAR <var\_name> = <var\_list>;

<var\_name> - 文字までの文字列で文字、数値、アンダースコアを使用できます。数字で終わってははいけません。

<var\_list> - (一つまたは、グループやフィールドの)order 命令からのシンボリックリストです。前に宣言された変数でコンマで区切られます。

<var\_list> = [!]<field> | [!]<group> | [!]<var> [..[!]<var> | ,<var\_list>]

動作:

<var\_list>の変数をグループ化します。FIELD 変数と似ています。ただし、この命令は ORDER 命令より前に記述することができません。ORDER 命令と VECTORS 命令の間に使用して下さい。

例:

```
VAR Z = Q7..4;
```

## シミュレータディレクティブ

シミュレータには、VECTOR 命令の後のファイルの任意の行に記述できる 6 個のディレクティブがあります。ディレクティブ名はすべてダラーマークで始まります。各ディレクティブステートメントはセミコロンで終了して下さい。

---

\$MSG	\$SIMOFF
\$REPEAT	\$SIMON
\$TRACE	\$EXIT

---

表 10-4 シミュレーターディレクティブ

### \$MSG

\$MSG ディレクティブを使用して、ドキュメントやフォーマット情報をシミュレータファイルに記述して下さい。例えば、変数名が一覧表示されるシミュレータ関数テーブルのヘッダーが作成できます。フォーマットを以下に示します。

```
$MSG "任意の文字列" ;
```

出力テーブルでは、文字列には引用符(ダブルクオート)はありません。

ブランク行を出力ファイルに挿入できます。例えば、以下のフォーマットによりベクタの間に挿入できます。

```
$MSG " " ;
```

### \$REPEAT

\$REPEAT ディレクティブにより、ベクタを指定された回数だけ繰返すことができます。フォーマットを以下に示します。

```
$REPEAT n ;
```

ここで

n は 1 から 9999 までの 10 進数値です。

\$REPEAT ディレクティブに続くベクタが、指定された回数だけ繰返されます。

\$REPEAT ディレクティブは、カウンタやステートランジションに便利です。アスタリスク(\*)を使用して、シミュレーションにより与えられる出力テスト値を示します。以下の例は、CUPL ソースファイルから 2 ビットのカウントを示し、そのカウントをテストするために \$REPEAT ディレクティブを使用する VECTORS 命令を命令を示します。

コンパイラから:

```
Q0.d = !Q0 ;
```

```
Q1.d = !Q1 & Q0 # Q1 & !Q0 ;
```

シミュレータで:

```
ORDER: clock, input, Q1, Q0 ;
VECTORS:
0 0 X X /* power-on condition */
P X 1 1 /* reset the flip-flops */
0 0 H H
$REPEAT 4 ; /* clock 4 times */
C 0 * *
```

上記のファイルにより以下のテストベクタが作成されます。

```
00XX
PX11
00HH
C0LL
C0LH
C0HL
C0HH
```

シミュレータによりベクタ値の4つのセットが与えられます。

## \$TRACE

\$TRACE ディレクティブを使用して、シミュレータが記述するベクタの情報を設定できます。以下にフォーマットを示します。

```
$TRACE n ;
```

ここで

n は 0 から 4 までの 10 進数値です。

**Trace level 0** (デフォルト) 追加情報はありません。テストベクタの結果だけが記述されます。ノンレジスタのフィードバックが使用されている場合、フィードバックの出力値はベクタの最初の評価が終わるまで不定です。新しいフィードバック値が出力値を変更する場合、ベクタは再び評価されます。ベクタが安定するまでは、出力はすべて同じである必要があります。

**Trace level 1** 安定するのに複数の評価パスを必要とする任意のベクタ中間結果を出力します。20 以上の評価パスが必要なベクタは、不安定と見なされます。

**Trace level 2** レジスタを使用する設計のシミュレーションの 3 フェーズ分を記述します。最初のフェーズは "Before the Clock" で、ノンレジスタードフィードバックを使用する中間ベクタが評価されます。2 番目のフェーズは、"At the Clock" で、レジスタの値がクロックの後すぐに与えられます。3 番目のフェーズは、"After the Clock" で、フィードバックに使用される出力がトレースレベル 1 で評価されます。

**Trace level 3** これにより、シミュレータでできる最高レベルの情報が出力されます。"Before the Clock"、"At Clock"、"After Clock" の各シミュレーショ

ンフェーズが出力され、各変数のそれぞれのプロダクトタームが一覧表示されます。AND ゲートの出力値は AND アレイへの入力値と一緒に一覧表示されます。

**Trace level 4** これにより、出力バッファの前の論理値を観察することができます。\$TRACE 4 を使用して、シミュレータに出力ピンが真であることをレポートさせたり、入力や埋めこみノードに"?"を割り付けたりします。組み合わせ出力では、トレースレベル 4 により、OR タームの結果を出力できます。レジスタード出力では、トレースレベル 4 により、レジスタの Q 出力を出力できます。以下の例では、p22v10 を使用しています。

```
pin 1 = CLK;
pin 2 = IN2;
pin 3 = IN3;
....
pin 14 = OUT14;
pin 15 = OUT15;
....
OUT14.D = IN2;
OUT14.AR = IN3;
OUT14.OE = IN4;
```

図 10-8 P22V10 の使用

図 10-9 はシミュレーション結果のファイルを示します。

```
order CLK, IN2, IN3, IN4, OUT14, OUT15;
***** before output buffer *****
???? ..LL...0001:
0011 ..HH.....
*****before output buffer*****
????  HH...0004
C100  ...ZZ.....
```

図 10-9 シミュレーションファイル

## \$EXIT

\$EXIT ディレクティブを使用して、任意の場所でシミュレーションを中断することができます。\$EXIT ディレクティブ以降のテストベクタは無視されます。このディレクティブは、あるベクタの間違ったトランジションにより起こる、それより後ろのベクタの誤動作をデバッグする時に有効です。

エラーのあるベクタの後に\$EXIT コマンドを置くと問題がはっきりします。

## \$SIMOFF

\$SIMOFF シミュレータディレクティブを使用すると、テストベクタの評価が停止されます。\$SIMOFF ディレクティブ以降のテストベクタは、テスト値が正しいかあるいは無効であるかだけが評価されます。このディレクティブは、シミュレータがレジスタード出力を正しく評価できない非同期クロックの設計をテストする時に有効です。



## \$SIMON

\$SIMON ディレクティブを使用して、\$SIMOFF ディレクティブの影響をキャンセルします。\$SIMON ディレクティブ以降のテストベクタは、通常どおり評価されます。

## アドバンストシンタクス

以下のコマンドはすべて、SI ファイルの VECTORS キーワード以降のテストベクタセクションで記述して下さい。

### 割り付け命令(\$SET)

Syntax: \$SET <variable> = <constant>;

<variable> = <single\_sym> | <field> | <defined\_variable>

<constant> = <quoted\_val> | <tv\_string>

<quoted\_val> = シングルクォートで囲まれた数値は、入力を表わし、ダブルクォートで囲まれた数値は出力示します。それらは基数に応じて展開されます。DONT-CARE 値を指定することはできません。

<tv\_string> = テストベクタ値の文字列です。値の数値は、それらが割り付けられる変数のビット番号に対応します。

動作:

定数をシンボルやフィールド、変数に割り付けます。テストベクタセクションの任意の位置に記述できます。

例:

```
$set input = '3F'; /* single quotes for inputs */
$set output = "80"; /* double quotes for outputs */
$set Z = HHHH; /* test vector values for a 4-bit output
variable */
```

### 算術及び論理演算子(\$COMP)

Syntax: \$COMP <variable> = <expression>;

<variable> = <single\_sym> | <field> | <defined\_variable>

<expression> = 任意の論理式または算術式で、オペランドには変数や定数を使用できます。

以下の定数は、10 進数値(引用符無し)で、丸括弧を使用できます。

演算子	機能	優先順位
!	NOT	1
&	AND	2
#	OR	3
\$	XOR	4

図 10-5 論理演算子

演算子	機能	優先順位
*	乗算	1
/	割り算	1
+	加算	2
-	減算	2

図 10-6 算術演算

論理演算子や算術演算子は、式の中で自由に組み合わせて使用できます。通常、論理演算子のほうが、優先順位がたかくなっています。しかし、この規則は、丸括弧を使用すればオーバーライドできます。

動作:

式や計算結果の変数への割り付けを評価します。オペランド(ユーザ値)の現在の値が式の評価に使用されます。変数のユーザ値の使用の時に影響します。ベクタセクションの任意の位置に記述できます。

例:

```
$COMP A = (!B + C) * A + 1;
$COMP X = (Z / 2) # MASK;
```

## テストベクタの生成\$OUT)

Syntax: \$OUT;

動作:

シンボルの現在の値のシミュレーションを開始しテストベクタを作成します。以前に割り付けられた値がベクタの評価で評価されるようにするため、\$SET コマンドや\$COMP コマンドの後に使用すると有効です。

例:

以下に SI ファイルのコマンドの組みを以下に示します。

```
ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
VECTORS:
0 0 'X' XXXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$set shift = '0';
$set input = '80';
$set output = "80";
$out;
```

図 10-10 .SI ファイル

これにより、SO ファイルに以下の結果が出力されます。

```
0001: 0 0 XXX XXXXXXXXX LLLLLLLL
0002: C 0 000 10000000 HLLLLLLL
```

## 条件シミュレーション(\$IF)

```
Syntax: $IF <condition> :  
    <block_1>  
    [ $ELSE :  
      <block_2> ]  
    $ENDIF;
```

<condition> = <var\_list> <logic\_operators> <constant>

```
logic operators :  
    = equal  
    # not equal  
    > greater than  
    < less than  
    >= greater than or equal to  
    <= less than or equal to
```

<constant> = <quoted\_val> | <tv\_string>

<block\_1>,<block\_2> = テストベクタを含む任意の命令シーケンス

\$ELSE は省略可能です。

動作:

変数の現在のシミュレーション値を使用して条件が評価されます。結果が真の場合、<block\_1>が実行されます。その他の場合、\$ELSE があれば、<block\_2>が実行されます。\$ENDIF は、IF ステートメントの終わりを示します。

## 繰り返し命令

### FOR 命令

```
Syntax: $FOR <count> = <n1>..<n2> :  
    <block>  
    $ENDF;
```

<count> = FOR ループのカウンタです。<n1> と <n2> との間で変化します。

<n1>,<n2> = <count>値の限界を示します。正の 10 進数値を設定して下さい。

<block> = テストベクタを含む任意の命令シーケンスです。

動作:

Step 1. <count>が最初の値<n1>に初期化されます。

Step 2. <block>が実行されます。

Step 3. <count>=<n2>の場合、停止します。

その他の場合、<count>が 1 ずつインクリメント(<n1>が<n2>より小さい場

合)されます。また、(<n1>が<n2>より大きい場合、1 ずつデクリメントされます。それから、ステップ 2.と 3.が繰返されます。

## WHILE 命令

```
Syntax: $WHILE <condition> :  
        <block>  
        $ENDW;
```

<condition> = IF 条件と同じです。

<block> = 命令の任意のシーケンスです。

動作:

Step 1: 条件が比較され、フォールスの場合停止します。

その他の場合、ステップ 2 へ進みます。

Step 2: <block>を実行します。

Step 3: ステップ 1 を繰返します。

## DO..UNTIL 命令

```
Syntax: $DO:  
        <block>  
        $UNTIL <condition> ;
```

<condition> = IF 条件と同じです。

<block> = 命令の任意のシーケンスです。

動作:

Step 1: <block>を実行します。

Step 2: 条件を比較し、真の場合停止します。

その他の場合、ステップ 1 を続けます。

IF 命令や繰り返し命令はネストすることができます。ただし、最多のネスト数は 10 までです。

## MACRO ステートメントとCALL ステートメント

### マクロ定義

```
Syntax: $MACRO name(<arg_list>);  
        <macro_body>  
        $MEND;
```

name = マクロ名です。

<arg\_list> = コンマで区切られた引き数名です。

<macro\_body> = 任意のステートメントのシーケンスです。ただし、(マクロセルの除く)\$MACRO コマンドは使用できません。

マクロ本体で変数名や定数を使用する所に、引き数を使用することができます。演算子や特別な文字、予約語を置き換えることはできません。

## マクロコール

Syntax: \$CALL name(<act\_arg\_list>);

name = 定義されているマクロの名前です。

<act\_arg\_list> = 引き数のリストです。

CALL ステートメントがマクロ本体に記述される場合、引き数には、変数名や定数、マクロ定義を指定できます。

動作:

引き数に指定された変数を置き換えてマクロの本体を実行します。

マクロコールを確実に実行するために、マクロ本体のシンタクスに引き数が合っている事を確認して下さい。すなわち、置き換えられる引き数がシンタクスエラーの原因にならないように注意して下さい。

例:

```
$MACRO m1(a,b,c);                /* Macro definition */
$set shift = a;
$set shift = b;
$set output = c;
$MEND;

$CALL m1('0','80',*****);      /* Macro call */
```

以下のステートメントが実行されます。

```
$set shift = '0';
$set shift = '80';
$set output = *****;
```

以下にこれらのステートメントがどのように動作するかをしめし、多くのテストベクタを入力しなくても設計のシミュレーションを実行できることを示します。

これら二つの SI ファイルは同じ出力を生成します。

### 1. 古い方法

Name	Barrel22;
Partno	CA0006;
Date	05/11/89;
Revision	02;
Designer	Kahl;
Company	Protel International;
Assembly	None;
Location	None;

```

Device      g20v8a;

ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
VECTORS:
0  0 'X' XXXXXXXX HHHHHHHH /* power-on reset state */
C  0 '0' 10000000 HLLLLLLL /* shift 0 */
C  0 '1' 10000000 LHLLLLLL /* shift 1 */
C  0 '2' 10000000 LLHLLLLL /* shift 2 */
C  0 '3' 10000000 LLLHLLLL /* shift 3 */
C  0 '4' 10000000 LLLLHLLL /* shift 4 */
C  0 '5' 10000000 LLLLLHLL /* shift 5 */
C  0 '6' 10000000 LLLLLLHL /* shift 6 */
C  0 '7' 10000000 LLLLLLLH /* shift 7 */
C  0 '0' 01111111 LHHHHHHH /* shift 0 */
C  0 '1' 01111111 HLHHHHHH /* shift 1 */
C  0 '2' 01111111 HHLHHHHH /* shift 2 */
C  0 '3' 01111111 HHHLHHHH /* shift 3 */
C  0 '4' 01111111 HHHHLHHH /* shift 4 */
C  0 '5' 01111111 HHHHHLHH /* shift 5 */
C  0 '6' 01111111 HHHHHHLH /* shift 6 */
C  0 '7' 01111111 HHHHHHHL /* shift 7 */

```

図 10-12.SI ファイル

## 2. 新しい方法

```

ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
VECTORS:
0 0 'X' XXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$set shift = '0';
$set input = '80';
$set output = "80";
$for i = 1..16 :
$out;
$if shift = '7':
$set shift = '0';
$set input = '7f';
$set output = "7f";
$else:
$comp shift = shift + 1;
$comp output = output / 2;
$if input = '7f':
$comp output = output # 128;
$endif;
$endif;
$endif;

```

図 10-13.SI ファイル

マクロを使用すると

```

ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
VECTORS:
$macro m1(x,y,z);
$set shift = x;
$set input = y;
$set output = z;
$mend;

$macro m2(a,b,c,d);
$call m1(a,b,c);
$for i = 1..8 :
$out; $comp shift = shift + 1;
$comp output = output / 2 + d;
$endf;
$mend;
0 0 'X' XXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$call m2('0','80',"80", 0);
$call m2('0','7f',"7f", 128);

```

図 10-14.SI ファイル

### 3. 出力:

```

CSIM: CUPL Simulation Program Version
4.2a Serial# ...
Copyright (c) 1996 Protel International
CREATED Wed Dec 04 03:00:11 1991
LISTING FOR SIMULATION FILE: barrel22.si
1: Name      Barrel22;
2: Partno    CA0006;
3: Date      05/11/89;
4: Revision  02;
5: Designer  Kahl;
6: Company   Protel International;
7: Assembly  None;
8: Location  None;
9: Device    g20v8a;
10:
11: FIELD input = [D7,D6,D5,D4,D3,D2,D1,D0];
12: FIELD output = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
13: FIELD shift = [S2,S1,S0];
14:
15: ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
16:
17: var X = Q7;
18: var Y = Q7..4;
19:
=====

```

	C				
	L				
	O	—			
	C	O	shi		
	K	E	ft	input	output
=====					
0001:	0	0	XXX	XXXXXXXXXX	LLLLLLLLL
0002:	C	0	000	10000000	HLLLLLLLL
0003:	C	0	001	10000000	LHLLLLLLL
0004:	C	0	010	10000000	LLHLLLLLL
0005:	C	0	011	10000000	LLLHLLLLL
0006:	C	0	100	10000000	LLLLHLLLL
0007:	C	0	101	10000000	LLLLLHLLL
0008:	C	0	110	10000000	LLLLLLHLL
0009:	C	0	111	10000000	LLLLLLLLH
0010:	C	0	000	01111111	LHHHHHHHH
0011:	C	0	001	01111111	HLHHHHHHH
0012:	C	0	010	01111111	HHLHHHHHH
0013:	C	0	011	01111111	HHHLHHHHH
0014:	C	0	100	01111111	HHHHLHHHH
0015:	C	0	101	01111111	HHHHHLHHH
0016:	C	0	110	01111111	HHHHHHLHH
0017:	C	0	111	01111111	HHHHHHHLH

図 10-15. SO ファイル

新しいシンタクスを使用してシミュレーション入力ファイルを作成する場合、ユーザが注意しなければならないことが一つあります。

中間に\$OUT ステートメントのない条件ステートメント(IF、WHILE、UNTIL)のすぐ前にひとつ以上の\$SET コマンドや\$COMP コマンドが記述される場合、これらのコマンドで設定される値(ユーザ値)は条件の値に影響しません。変数の最後のシミュレーション結果を使用して条件は評価されます。

例えば、以下のシミュレーション結果を生成したい場合、

ORDER: _CLOCK,clr,dir,!_OE,%2,count,%1,carry;									
var mode = clr,dir;									
VECTORS:									
C	100	LLLL	L	/*	synchronous	clear	to	state	0 */
C	000	LLLH	L	/*	count	up	to	state	1 */
C	000	LLHL	L	/*	count	up	to	state	2 */
C	000	LLHH	L	/*	count	up	to	state	3 */
C	000	LHLL	L	/*	count	up	to	state	4 */
C	000	LHLH	L	/*	count	up	to	state	5 */
C	000	LHHL	L	/*	count	up	to	state	6 */
C	000	LHHH	L	/*	count	up	to	state	7 */
C	000	HLLL	L	/*	count	up	to	state	8 */
C	000	HLLH	H	/*	count	up	to	state	9 - carry */



図 10-16 予想した出力

以下のシーケンスでは、違う出力が生成されます。

```
ORDER: _CLOCK,clr,dir,!_OE,%2,count,%1,carry;
var mode = clr,dir;
VECTORS:
C 100 LLLL L $set mode = '0';
$for i=1..9 :
$comp count = count + 1;
$if count="9":
$set carry = H;
$endif;
$out;
$endf;
```

図 10-17 シミュレーション入力(誤)

that is:

```
0001: C 100 LLLL L
0002: C 000 LLLH L
0003: C 000 LLHL L
0004: C 000 LLHH L
0005: C 000 LHLL L
0006: C 000 LHLH L
0007: C 000 LHHL L
0008: C 000 LHHH L
0009: C 000 HLLL L
0010: C 000 HLLH H
      ^
[0019sa] user expected (L) for carry
```

図 10-18 シミュレーション出力

これは、ベクタ 10 の IF 条件の評価に使用された count の値は現在のシミュレーション値(ベクタ 9 に表示される値)であり、\$COMP コマンドに設定された値ではないためです。

正しいシーケンスを以下に示します。

```
C 100 LLLL L
$set mode = '0';
$for i=1..9 :
$if count="8":
$set carry = H;
$endif;
$comp count = count + 1;
$out;
$endf;
```

図 10-19 シミュレーション入力(正)

## 仮想シミュレーション

仮想シミュレーションによりターゲットをデバイスを使用しないで作成した設計のシミュレーションができます。従って、ターゲットにするアーキテクチャが決まる前に設計のシミュレーションを実行できます。これは、部分的な設計のシミュレーションに便利です。

仮想シミュレーションの使用法はわかりやすくなっています。コマンドやシンタクスを新しく習得する必要はありません。コンパイルやシミュレーションで `Virtual DeviceVirtualDevice` オプションを使用すると仮想シミュレーションを実行できます。

仮想シミュレーションは FPGA の設計のシミュレーションにも使用できます。デバイス内部の特性や内部論理リソースの複雑さのためにアーキテクチャのシミュレーションを完全に実行できない場合、仮想シミュレーションの使用は非常に有効です。

## 欠陥のシミュレーション

テストベクタの欠陥の保険の意味で、任意のプロダクトタームの内部欠陥のシミュレーションを実行できます。このオプションのフォーマットを以下に示します。

```
STUCKL n ;
```

または

```
STUCKH n ;
```

ここで

`n` はプロダクトタームの最初のヒューズのヒューズ番号です。

ドキュメンテーションファイル(ファイル名.DOC)のヒューズマップにデバイスの各プロダクトタームの最初のヒューズのヒューズ番号が示されます。

フォーマット 1 によりプロダクトタームは 0 番にスタックされます。

フォーマット 2 によりプロダクトタームは 1 番にスタックされます。STUCK コマンドは、ORDER ステートメントと VECTORS ステートメントの間に置かれます。

# デバイスリスト (Jan 96)

## ACTEL PLD 2

ACT1010#      ACT1020#

#-オプションのソフトウェアが必要です。

## ALTERA PLD 42

EP1200	EP1210	EP1800G	EP1800J/L
EP1810G	EP1810J/L	EP1810T	EP1830L
EP300	EP310	EP320	EP330L
EP330P	EP330S	EP512	EP600D/P
EP610/T	EP610AL	EP610AP	EP610D/P
EP610S	EP630P	EP630S	EP900D/P
EP900J/L	EP910A	EP910D/P	EP910J/L
EP910T	EPM5016*	EPM5032D/P&	EPM5032J/L&
EPM5032S&	EPM5064J/L*	EPM5128G*	EPM5128J/L*
EPM5130G*	EPM5130J/L*	EPM5130Q/W*	EPM5192G*
EPM5192J/L*	MAX7032		

#-オプションのソフトウェアが必要です。

## AMD/MMI PLD 293

AMPAL16H8/A	AMPAL16HD8/A	AMPAL16L8A/B/L/AL/Q
AMPAL16LD8/A		
AMPAL16R4A/B/L/AL/Q	AMPAL16R6A/B/L/AL/Q	
AMPAL16R8A/B/L/AL/Q	AMPAL18P8A/B/L/AL/Q	
AMPAL20L10/B	AMPAL20L8A/B	AMPAL20R4A/B
AMPAL20R6A/B		
AMPAL20R8A/B	AMPAL20RP10A/AL/B	AMPAL20RP4A/AL/B
AMPAL20RP6A/AL/B		
AMPAL20RP8A/AL/B	AMPAL20XRP10	AMPAL20XRP4
AMPAL20XRP6		
AMPAL20XRP8	AMPAL22P10A/AL/B	AMPAL22V10/A
AMPAL22XP10		
AMPAL23S8	MACH111	MACH120
MACH130	MACH131	MACH210
MACH215	MACH220	MACH210-10
MACH235	MACH230	MACH231
MACH355	MACH435	MACH231
	MACH445	MACH465
PAL10H/10020EV8	PAL10H20G8	PAL10H20P8
PAL10H8/CE16V8	PAL10H8/CE16V8HD	PAL10H8/CE16V8Z

PAL10L8/-2  
 PAL10L8/CE16V8 PAL10L8/CE16V8HD PAL10L8/CE16V8Z  
 PAL10P8/CE16V8  
 PAL10P8/CE16V8HD PAL10P8/CE16V8Z PAL12H6/-2  
 PAL12H6/CE16V8  
 PAL12H6/CE16V8HD PAL12H6/CE16V8Z PAL12L10 PAL12L6/-2  
 PAL12L6/CE16V8 PAL12L6/CE16V8HD PAL12L6/CE16V8Z  
 PAL12P6/CE16V8  
 PAL12P6/CE16V8HD PAL12P6/CE16V8Z PAL14H4/-2  
 PAL14H4/CE16V8  
 PAL14H4/CE16V8HD PAL14H4/CE16V8Z PAL14H8/CE20V8 PAL14L4/-2  
 PAL14L4/CE16V8 PAL14L4/CE16V8HD PAL14L4/CE16V8Z  
 PAL14L8  
 PAL14L8/CE20V8 PAL14P4/CE16V8 PAL14P4/CE16V8HD  
 PAL14P4/CE16V8Z  
 PAL14P8/CE20V8 PAL16C1/-2 PAL16H2/-2  
 PAL16H2/CE16V8  
 PAL16H2/CE16V8HD PAL16H2/CE16V8Z PAL16H6/CE20V8  
 PAL16H8/CE16V8  
 PAL16H8/CE16V8HD PAL16H8/CE16V8Z PAL16L2/-2  
 PAL16L2/CE16V8  
 PAL16L2/CE16V8HD PAL16L2/CE16V8Z PAL16L6 PAL16L6/CE20V8  
 PAL16L8-4 PAL16L8-5 PAL16L8-7 PAL16L8/A/A-2/A-4  
 PAL16L8/CE16V8 PAL16L8/CE16V8HD PAL16L8/CE16V8Z  
 PAL16L8B  
 PAL16L8B-2/B-4 PAL16L8BP PAL16L8D PAL16L8D/2  
 PAL16L8H-10 PAL16L8H-15 PAL16P2/CE16V8  
 PAL16P2/CE16V8HD  
 PAL16P2/CE16V8Z PAL16P6/CE20V8 PAL16P8/CE16V8  
 PAL16P8/CE16V8HD  
 PAL16P8/CE16V8Z PAL16P8A PAL16P8B PAL16R4-4  
 PAL16R4-5 PAL16R4-7 PAL16R4/A/A-2/A-4  
 PAL16R4/CE16V8  
 PAL16R4/CE16V8HD PAL16R4/CE16V8Z PAL16R4B PAL16R4B-2/B-4  
 PAL16R4BP PAL16R4D PAL16R4D/2 PAL16R4H-10  
 PAL16R4H-15 PAL16R6-4 PAL16R6-5 PAL16R6-7  
 PAL16R6/A/A-2/A-4 PAL16R6/CE16V8 PAL16R6/CE16V8HD  
 PAL16R6/CE16V8Z  
 PAL16R6B PAL16R6B-2/B-4 PAL16R6BP PAL16R6D  
 PAL16R6D/2 PAL16R6H-10 PAL16R6H-15 PAL16R8-4  
 PAL16R8-5 PAL16R8-7 PAL16R8/A/A-2/A-4  
 PAL16R8/CE16V8

PAL16R8/CE16V8HD PAL16R8/CE16V8ZPAL16R8B PAL16R8B-2/B-4  
 PAL16R8BP PAL16R8D PAL16R8D/2 PAL16R8H-10  
 PAL16R8H-15 PAL16RA8 PAL16RP4/CE16V8  
 PAL16RP4/CE16V8HD  
 PAL16RP4/CE16V8Z PAL16RP4A PAL16RP6/CE16V8  
 PAL16RP6/CE16V8HD  
 PAL16RP6/CE16V8Z PAL16RP6A PAL16RP8/CE16V8  
 PAL16RP8/CE16V8HD  
 PAL16RP8/CE16V8Z PAL16RP8A PAL18H4/CE20V8 PAL18L4  
 PAL18L4/CE20V8 PAL18P4/CE20V8 PAL20C1 PAL20H2/CE20V8  
 PAL20H8/CE20V8 PAL20L10 PAL20L10A PAL20L2  
 PAL20L2/CE20V8 PAL20L8 PAL20L8-10 PAL20L8-10/2  
 PAL20L8-5 PAL20L8-7 PAL20L8/CE20V8 PAL20L8A/A-2  
 PAL20L8B PAL20L8B-2 PAL20P2/CE20V8 PAL20P8/CE20V8  
 PAL20R4 PAL20R4-10 PAL20R4-10/2 PAL20R4-5  
 PAL20R4-7 PAL20R4/CE20V8 PAL20R4A/A-2 PAL20R4B  
 PAL20R4B-2 PAL20R6 PAL20R6-10 PAL20R6-10/2  
 PAL20R6-5 PAL20R6-7 PAL20R6/CE20V8 PAL20R6A/A-2  
 PAL20R6B PAL20R6B-2 PAL20R8 PAL20R8-10  
 PAL20R8-10/2 PAL20R8-5 PAL20R8-7  
 PAL20R8/CE20V8  
 PAL20R8A/A-2 PAL20R8B PAL20R8B-2 PAL20RA10  
 PAL20RP4/CE20V8 PAL20RP6/CE20V8PAL20RP8/CE20V8PAL20RS10  
 PAL20RS4 PAL20RS8 PAL20S10 PAL20X10  
 PAL20X10A PAL20X4PAL20X4A PAL20X8  
 PAL20X8A PAL22IP6-35 PAL22RX8 PAL22RX8A  
 PAL22V10 PAL24L10 PAL24R10 PAL24R4  
 PAL24R8 PAL32R16 PAL32VX10 PAL32VX10A  
 PAL64R32 PAL6L16A PAL8L14A PALC16L8Q  
 PALC16L8Z PALC16R4Q PALC16R4Z PALC16R6Q  
 PALC16R6Z PALC16R8Q PALC16R8Z PALC18U8  
 PALC20L8Z PALC20R4Z PALC20R6Z PALC20R8Z  
 PALC22V10H PALC22V10Q PALCE16V8/4/5  
 PALCE16V8H/Q  
 PALCE16V8HD PALCE16V8Z PALCE20RA10  
 PALCE20V8/4/5  
 PALCE20V8/5\* PALCE20V8H/Q PALCE22V10/4  
 PALCE22V10/5  
 PALCE22V10H/Q PALCE22V10Z PALCE24V10H  
 PALCE26V12H

PALCE26V12H/4	PALCE29M16	PALCE29M16/4	
PALCE29MA16			
PALCE29MA16/4	PALCE610	PALLV22V10	PLS105
PLS167A/B	PLS168A/B	PLS30K12	PLS30S16
PMS14R21			

#-オプションのソフトウェアが必要です。

#### AMD/MMI PROM 49

53/6300	53/6301	53/6305	53/6306
53/6308	53/6309	53/6330	53/6331
53/6348	53/6349	53/6352	53/6353
53/6380	53/6380JS	53/6380S	53/6381
53/6381JS	53/6381S	53/6388	53/6389
53/63S1641	53/63S1641A	53/63S1681	53/63S1681A
53/63S3281/A/B	AM27LS19	AM27PS191/A	AM27PS41
AM27S12	AM27S13/A	AM27S18	AM27S180
AM27S181/A	AM27S184	AM27S185/A	AM27S19/A/SA
AM27S190	AM27S191/A/SA	AM27S20	AM27S21/A
AM27S28	AM27S29/A/SA	AM27S32	AM27S33/A
AM27S37/A	AM27S40	AM27S41/A	AM27S43/A
AM27S49/A/SA			

#### AMI/GOULD PLD 11

PA7024	PA7040	PEEL153	PEEL173
PEEL18CV8	PEEL18CV8-10/15	PEEL22CV10	PEEL22CV10A
PEEL22CV10Z	PEEL253	PEEL273	

#### ATMEL PLD 14

AT18V8Z	AT22LV10/L	AT22V10/L	AT22V10B
ATF16V8B/L	ATF20V8B/L	ATF22V10B/L	ATS415
ATS42VA12	ATV2500/H	ATV2500B/L	ATV5000/L
ATV750/L	ATV750B/L		

#### CYPRESS EPROM 5

CY7C258	CY7C259	CY7C264	CY7C282
CY7C292			

#### CYPRESS PLD 71

CY100E301	CY100E302	CY10E301	CY10E302
CY7B336	CY7B337	CY7B338	CY7B339
CY7C330	CY7C331	CY7C332	CY7C335

CY7C341	CY7C342	CY7C343	CY7C344
CY7C361	CY7C371	CY7C372	CY7C373
CY7C374	CY7C375	pASIC381#	pASIC382#
pASIC383#	pASIC384#	pASIC385#	pASIC386#
pASIC387#	pASIC388#		
PAL16L8-4	PAL16L8-5/7/10	PAL16L8A	
PAL16L8A-2	PAL16R4-4	PAL16R4-5/7/10	PAL16R4A
PAL16R4A-2	PAL16R6-4	PAL16R6-5/7/10	PAL16R6A
PAL16R6A-2	PAL16R8-4	PAL16R8-5/7/10	PAL16R8A
PAL16R8A-2	PAL22V10C	PAL22VP10C	PALC12L10
PALC14L8	PALC16L6	PALC16L8	PALC16R4
PALC16R6	PALC16R8	PALC18L4	PALC20L10
PALC20L2	PALC20L8	PALC20R4	PALC20R6
PALC20R8	PALC22V10	PALC22V10B	PALC22V10D
PLD20G10C	PLD610	PLDC18G8	PLDC20G10
PLDC20G10B	PLDC20RA10		

#-オプションのソフトウェアが必要です。

#### **EXEL PLD 1**

XL78C800

#### **FAIRCHILD PLD 14**

93Z458	93Z459	F16L8	F16P8
F16R4	F16R6	F16R8	F16RP4
F16RP6	F16RP8	F20P8	F20RP4
F20RP6	F20RP8		

#### **HARRIS PLD 14**

HPL16H8	HPL16L8	HPL16LC8	HPL16P8
HPL16R4	HPL16R6	HPL16R8	HPL16RC4
HPL16RC6	HPL16RC8	HPL77153	HPL77209
HPL77216	HPL82S153		

#### **HARRIS PROM 46**

HM7602	HM7603	HM7610	HM7610A
HM7610B	HM7611	HM7611A	HM7611B
HM76160	HM76161	HM76161A	HM76161B
HM76164	HM76165	HM7620	HM7620A
HM7620B	HM7621	HM7621A	HM7621B
HM76321	HM7642	HM7642A	HM7642B

HM7643	HM7643A	HM7643B	HM7648
HM7649	HM7649A	HM76641	HM76641A
HM7680	HM7680P	HM7680R	HM7680RP
HM7681	HM7681A	HM7681P	HM7681R
HM7681RP	HM7684	HM7684P	HM7685
HM7685A	HM7685P		

#### ICT PLD 15

PA7024	PA7128	PA7140	PEEL153
PEEL173	PEEL18CV8	PEEL18CV8-10/15	PEEL20CG10
PEEL20CG10A	PEEL22CV10	PEEL22CV10A	PEEL22CV10A+
PEEL22CV10Z	PEEL253	PEEL273	

#### INTEL PLD 20

5AC312	5AC324	5C031	5C032
5C060	5C090	5C121	5C180
85C060	85C090	85C220	85C224
85C22V10	85C508	85C960	IFX740&
IFX780	IPLD22V10	IPLD610	IPLD910

#### LATTICE PLD 83

GAL16LV8	GAL16V8	GAL16V8A	GAL16V8B
GAL16V8C/D/Z	GAL16VP8B	GAL16Z8	GAL18V10
GAL18V10B	GAL20LV8	GAL20RA10/B	GAL20RA10/B(UES)
GAL20V8	GAL20V8A	GAL20V8B	GAL20V8C/Z
GAL20VP8B	GAL20XV10B	GAL22V10(UES)	GAL22V10B
GAL22V10B(UES)	GAL22V10BQ/C	GAL22V10BQ/C(UES)	
	GAL26CV12/B/C		
GAL6001	GAL6001B	GAL6002B	ISPGAL22V10
ISPLSI1016#	ISPLSI1024#	ISPLSI1032#	ISPLSI1048#
PLSI1016#	PLSI1024#	PLSI1032#	PLSI1048#
ISPLSI2032#	PLSI2032#	ISPLSI3256#	PLSI3256#
RAL10H8	RAL10L8	RAL10P8	RAL12H6
RAL12L6	RAL12P6	RAL14H4	RAL14H8
RAL14L4	RAL14L8	RAL14P4	RAL14P8
RAL16C1	RAL16H2	RAL16H6	RAL16H8
RAL16L2	RAL16L6	RAL16L8	RAL16P2
RAL16P6	RAL16P8	RAL16R4	RAL16R6
RAL16R8	RAL16RP4	RAL16RP6	RAL16RP8
RAL18H4	RAL18L4	RAL18P4	RAL20H2



RAL20H8	RAL20L2	RAL20L8	RAL20P2
RAL20P8	RAL20R4	RAL20R6	RAL20R8
RAL20RP4	RAL20RP6	RAL20RP8	

#-オプションのソフトウェアが必要です。

## MOTOROLA PLD 1

MC22V10S

## NATIONAL PLD 121

87X839	87X840	GAL16V8	
GAL16V8-10/7			
GAL16VA/QS	GAL20RA10	GAL20V8	GAL20V8-10/7
GAL20V8A/QS	GAL22V10	GAL6001	
MAPL128&			
MAPL144 (UES)	MAPL244 (UES)	PAL10012C4	PAL10016C4
PAL10016LD4	PAL10016LD8	PAL10016LM4	PAL10016P4
PAL10016P8	PAL10016PE8	PAL10016RD4	PAL10016RD8
PAL10016RM4	PAL10020RP4	PAL1012C4	PAL1016C4
PAL1016LD4	PAL1016LD8	PAL1016LM4	PAL1016P4
PAL1016P8	PAL1016PE8	PAL1016RD4	PAL1016RD8
PAL1016RM4	PAL1020RP4	PAL10H8/16V8	PAL10H8/A/A2
PAL10L8/16V8	PAL10L8/A/A2	PAL10P8/16V8	PAL12H6/16V8
PAL12H6/A/A2	PAL12L10/A	PAL12L6/16V8	PAL12L6/A/A2
PAL12P6/16V8	PAL14H4/16V8	PAL14H4/A/A2	PAL14H8/20V8
PAL14L4/16V8	PAL14L4/A/A2	PAL14L8/20V8	PAL14L8/A
PAL14P4/16V8	PAL14P8/20V8	PAL16C1/A/A2	PAL16H2/16V8
PAL16H2/A/A2	PAL16H6/20V8	PAL16H8/16V8	PAL16L2/16V8
PAL16L2/A/A2	PAL16L6/20V8	PAL16L6/A	PAL16L8/16V8
PAL16L8/A/A2/B/B2	PAL16L8D/-7	PAL16P2/16V8	PAL16P6/20V8
PAL16P8	PAL16P8/16V8	PAL16R4/16V8	
	PAL16R4/A/A2/B/B2		
PAL16R4D/-7	PAL16R6/16V8	PAL16R6/A/A2/B/B2	
	PAL16R6D/-7		
PAL16R8/16V8	PAL16R8/A/A2/B/B2	PAL16R8D/-7	PAL16RA8
PAL16RP4	PAL16RP4/16V8	PAL16RP6	
	PAL16RP6/16V8		
PAL16RP8	PAL16RP8/16V8	PAL18H4/20V8	PAL18L4/20V8
PAL18L4/A	PAL18P4/20V8	PAL20C1/A	PAL20H2/20V8
PAL20H8/20V8	PAL20L10/A	PAL20L2/20V8	PAL20L2/A
PAL20L8/20V8	PAL20L8/A/B	PAL20L8D	PAL20P2/20V8

PAL20P8/20V8	PAL20R4/20V8	PAL20R4/A/B	PAL20R4D
PAL20R6/20V8	PAL20R6/A/B	PAL20R6D	PAL20R8/20V8
PAL20R8/A/B	PAL20R8D	PAL20RA10	
	PAL20RP4/20V8		
PAL20RP6/20V8	PAL20RP8/20V8	PAL20X10/A	PAL20X4/A
PAL20X8/A			

#### **NATIONAL PROM 39**

DM74LS471	DM74S188	DM74S287	DM74S288
DM74S387	DM74S472	DM74S472A	DM74S472B
DM74S473	DM74S473A	DM74S570	DM74S570A
DM74S571	DM74S571A	DM74S571B	DM74S572
DM74S572A	DM74S573	DM74S573A	DM74S573B
DM87S180	DM87S181	DM87S181A	DM87S184
DM87S185	DM87S185A	DM87S185B	DM87S190
DM87S190A	DM87S190B	DM87S191	DM87S191A
DM87S191B	DM87S195	DM87S195A	DM87S195B
DM87S321	DM87S321A		

#### **PHILIPS PLD 60**

10020EV8	10H20EV8	PHD16N8	PHD48N22
PL22V10	PLC153	PLC16V8	PLC18V8Z
PLC20V8	PLC415	PLC42VA12	PLC42VA12&
PLC473	PLHS153	PLHS16L8A	PLHS16L8B
PLHS18P8A	PLHS18P8B	PLHS473	PLHS501
PLHS502	PLS100/82S100	PLS101/82S101	PLS103/82S103
PLS105/82S105	PLS105A/82S105/A	PLS151/82S151	PLS152/82S152
PLS153/82S153	PLS153A/82S153A	PLS155/82S155	PLS157/82S157
PLS159/82S159	PLS159A/82S159A	PLS161/82S161	PLS162/82S162
PLS163/82S163	PLS167/82S167	PLS167A/82S167A	PLS168/82S168
PLS168A/82S168A	PLS173/82S173	PLS179/82S179	PLUS105
PLUS153B/D	PLUS16L8D/-7	PLUS16R4D/-7	PLUS16R6D/-7
PLUS16R8D/-7	PLUS173B/D	PLUS20L8D/-7	PLUS20R4D/-7
PLUS20R6D/-7	PLUS20R8D/-7	PLUS405	PLV2500H/L*
PLV5000/L&	PLV750/L	PML2552&	PML2852&

#-オプションのソフトウェアが必要です。

#### **PHILIPS PROM 17**

82S123/A	82S126/A	82S129/A	82S130/A
82S131/A	82S135	82S137/A/B	82S147/A

82S180	82S181/A/B	82S184	82S185/A/B
82S191/A/B/C	82S195	82S23/A	82S321
82S641			

#### **PLESSY PLD 1**

ERA60100\*

#-オプションのソフトウェアが必要です。

#### **PLUS LOGIC PLD 2**

FPGA2020#      FPGA2020A#

#-オプションのソフトウェアが必要です。

#### **PLX TECH. PLD 2**

PLX448      PLX464

#### **QUICKLOGIC PLD 4**

P8X12B#      P12X16B#      P16X24B#      P24X32B#

P8X12A#

#-オプションのソフトウェアが必要です。

#### **RICOH PLD 14**

EPL10P8A(I)	EPL10P8B(I)	EPL12P6A(I)	EPL12P6B(I)
EPL14P4A	EPL14P4B	EPL16P2A(I)	EPL16P2B(I)
EPL16P8B(I)	EPL16RP4B(I)	EPL16RP6B(I)	EPL16RP8B(I)
EPL204E*	EPL241E*		

#-オプションのソフトウェアが必要です。

#### **SAMSUNG PLD 18**

CPL16L8	CPL16L8L	CPL16R4	CPL16R4L
CPL16R6	CPL16R6L	CPL16R8	CPL16R8L
CPL20L10	CPL20L10L	CPL20L8	CPL20L8L
CPL20R4	CPL20R4L	CPL20R6	CPL20R6L
CPL20R8	CPL20R8L		

#### **SEEQ PLD 2**

20RA10Z      26V12H

#### **SGS-THOM. PLD 48**

GAL16V8	GAL16V8A	GAL16Z8	GAL20V8
GAL20V8A	GAL6001	RAL10H8	RAL10L8

RAL10P8	RAL12H6	RAL12L6	RAL12P6
RAL14H4	RAL14H8	RAL14L4	RAL14L8
RAL14P4	RAL14P8	RAL16H2	RAL16H6
RAL16H8	RAL16L2	RAL16L6	RAL16L8
RAL16P2	RAL16P6	RAL16P8	RAL16R4
RAL16R6	RAL16R8	RAL16RP4	RAL16RP6
RAL16RP8	RAL18H4	RAL18L4	RAL18P4
RAL20H2	RAL20H8	RAL20L2	RAL20L8
RAL20P2	RAL20P8	RAL20R4	RAL20R6
RAL20R8	RAL20RP4	RAL20RP6	RAL20RP8

## SIGNETICS

SEE PHILIPS

## SPRAGUE PLD 6

SPL16LC8	SPL16RC4	SPL16RC6	SPL16RC8
SPL20LC8	SPL20XC8		

## TI PLD 88

EP1810	EP330	EP610	EP630
EP910	N82S105AN82S167A	PAL16L8A/-2	
PAL16R4A/-2	PAL16R6A/-2	PAL16R8A/-2	PAL20L8A
PAL20R4A	PAL20R6A	PAL20R8A	
	TIB82S105A/B		
TIB82S167B	TIBFPGA529	TIBPAD16N8-7.5	TIBPAD18N8-6
TIBPAL16H8	TIBPAL16HD8	TIBPAL16L8-10	TIBPAL16L8-12/15/25
TIBPAL16L8-5	TIBPAL16L8-7	TIBPAL16LD8	TIBPAL16O2
TIBPAL16R4-10	TIBPAL16R4-12/15/25	TIBPAL16R4-5	TIBPAL16R4-7
TIBPAL16R6-10	TIBPAL16R6-12/15/25	TIBPAL16R6-5	TIBPAL16R6-7
TIBPAL16R8-10	TIBPAL16R8-12/15/25	TIBPAL16R8-5	TIBPAL16R8-7
TIBPAL20L10	TIBPAL20L8-15/25	TIBPAL20R4-15/25	
	TIBPAL20R6-15/25		
TIBPAL20R8-15/25		TIBPAL20RSP4	
	TIBPAL20RSP6	TIBPAL20RSP8	
TIBPAL20SP8	TIBPAL20X10	TIBPAL20X4	TIBPAL20X8
TIBPAL22V10-5	TIBPAL22V10/A	TIBPAL22VP10	
	TIBPALR19L8		

TIBPALR19R4	TIBPALR19R6 TIBPALT19L8	TIBPALR19R8	
TIBPALT19R4	TIBPALT19R6	TIBPALT19R8	TIBPLS506A
TIBPSG507A	TICPAL16L8-55	TICPAL16R4-55	TICPAL16R6-55
TICPAL16R8-55	TICPAL16RSP4 TICPAL16RSP8	TICPAL16RSP6	
TICPAL16SP8	TICPAL18V8 TICPAL22V10Z(T)	TICPAL22V10	
TICPAL22V10Z(ZP)	TIEPAL10016P8-6	TIEPAL10016ET6 TIEPAL10016TE6	
TIEPAL10H16ET6	TIEPAL10H16P8-6	TIEPAL10H16TE6	TIFPLA839
TIFPLA840	TPC1010#	TPC1020#	

#-オプションのソフトウェアが必要です。

#### TI PROM 18

TBP18S030	TBP18S22	TBP18SA030	TBP18SA22
TBP24S10	TBP24S166	TBP24S41	TBP24S81
TBP24SA10	TBP24SA166	TBP24SA41	TBP24SA81
TBP28S166	TBP28S42	TBP28S86A	TBP28SA166
TBP28SA42	TBP28SA86A		

#### TOSHIBA PLD 2

TC9800P	TC9801P
---------	---------

#### TRIQUINT PLD 4

GA22V10	GA22VP10	GA23S8	GA23SV8
---------	----------	--------	---------

#### VLSI PLD 10

VP10P8	VP12P6	VP14P4	VP16P2
VP16P8	VP16RP4	VP16RP6	VP16RP8
VP16V8E	VP20V8E		

#### XILINX PLD 30

XC2018#	XC2064#	XC3020/B#	XC3030#
XC3042/B#	XC3064#	XC3090/B#	XC4002#
XC4003#	XC4004#	XC4005#	XC4006#
XC4008#	XC4010#	XC4013#	XC4016#
XC4020#	XC7236APC	XC7236PC	XC7272APC
XC7272PC	XC7272PG	XC73108	XC73108BG
XC73108PG	XC73108PQ	XC7336PC	XC7336PQ

XC7354PC

XC7372PC

#-オプションのソフトウェアが必要です。

# ファイルフォーマット

このセクションでは、以下のフォーマットについて説明します。

## ダウンロードフォーマット

### JEDEC フォーマット

JEDEC JC-42.1 標準は、ASCII Start-of-Text(STX)キャラクタで始まり、トランスミッションに続いて各種の情報のフィールドにより構成されます。情報フィールドには、ASCII End-of-Text(ETX)キャラクタや送信チェックサムがあります。使用できるキャラクタは hex 20 から hex 7E までの ASCII の表示できるキャラクタと表 11-1 で示される 4 つの制御文字です。

STX	Start-of-Text	hex 02
ETX	End-of-Text	hex 03
LF	Line Feed	hex 0A
CR	Carriage Return	hex 0D

表 11-1 制御文字

図 11-1 に、コンパイラとシミュレータを使用して作成した JEDEC ファイルのサンプルを示します。

```
<STX>
Cupl          3.0  Serial # 0-00000-000
Device        p16r4  Library DLIB-h-24-11
Created       Tue Jul 07 15:22:33 1987
Name          SAMPLE
Partno        P9000183
Revision      02
Date          03/14/85
Designer      Osann
Company       P-CAD
Assembly      PC Memory
Location      U106
*QP20
*QF2048
*G0
*F0
*L00000 10110101110111111100111000110111
*C0307
*QV
*P 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
*V0001 CXXXXXX110N0HHLLZXXHN
*<ETX>6AA1
```

図 11-1 JEDEC ファイルのサンプル

このセクションの残りの部分では、図 11-1 のサンプルファイルの各フィールドについて説明します。

設計の仕様はフォーマットの最初のフィールドです。このフィールドには、STX と最初のアスタリスク(\*)の間の情報がすべて含まれます。この情報は、ドキュメントに使用され、それ以外には使用されません。さらに、コンパイラやデバイスライブラリのバージョンに沿った CUPL ソースファイルからのヘッダー情報で構成されます。

設計仕様フィールドの後の各フィールドは表 11-2 で示される 1 文字の識別子で始まります。

A - *	N - *
B - *	O - *
C - ヒューズチェックサム	P - ピン順序
D - デバイスタイプ	Q - 値
E - *	R - *
F - デフォルトヒューズステイト	S - *
G - セキュリティヒューズ	T - *
H - *	U - *
I - *	V - テストベクタ
J - *	W - *
K - *	X - *
L - ヒューズリンクデータ	Y - *
M - *	Z - *

\* - 将来の使用が考えられるために予約されています。

表 11-2. フィールド識別子

フィールドは、複数の文字で識別することができます。例えば、QF はデフォルトヒューズステイトの値を示します。

デバイスフィールド(D)はサポートされていません。

バリューフィールド QP は、デバイスのピン番号を表わします。その他のバリューフィールド QF はデバイスの中でプログラムできるヒューズの総数を表わします。値は両方とも 10 進数です。

セキュリティーヒューズフィールド(G)は、このオプションのあるデバイスのセキュリティヒューズのプログラムがディスエーブル(G0)はイネーブル(G1)かをプログラマに指示します。スペースを一つあけて互換性のあるデバイスの番号が示されます。

デフォルトヒューズステイトフィールド(F)により、L フィールドよりさらに厳密にヒューズのステイトが定義されます。コンパイラは、一部のヒューズ状態しか変換しない(大規模な設計のではデータ変換の速度をあげる



ため)ので、このフィールドはデバイスプログラマにより認識されます。

ヒューズリンクフィールド(L)には、実際のデータがあります。各デバイスのヒューズリンクが 10 進数で割り付けられます。この値は 0000 から始まります。番号が付けられたそれぞれのヒューズには、2 つの状態があります。2 進数の 0 は、無傷のヒューズを表わし、2 進数の 1 は、ヒューズの断線を表わします。

製造元の中にはプログラミング前のデバイスの AC パラメータテストを実行するためのテストヒューズを指定するところもあります。これらのヒューズはヒューズリンクデータの部品ではありません。

L 識別子によりフィールドが始まり、フィールドで定義される最初のヒューズの番号がこれに続きます。複数の 2 進数値が指定される場合、最初のヒューズから連続で番号が付けられたヒューズに追加の値が割り付けられます。

次のフィールドは、ヒューズチェックサム(C)フィールドです。チェックサムは、デバイスの各ヒューズの指定された状態で形成される 8 ビットワードを加えることにより計算される 16 ビットの 16 進値です。リンク番号 0 は最下位ビットを表わし、リンク番号 7 はワード 0 の最上位ビットを表わします。最後の 8 ビットの指定されていないビットは、チェックサムが計算される前に 0 がセットされます。図 11-1 で、最初の 22 のヒューズにより 8 ビットワードは以下のように作成されます。

	msb							lsb	
word 00	1	0	1	0	1	1	0	1	-> AD
word 01	1	1	1	1	1	0	1	1	-> FB
word 02	0	1	1	1	0	0	1	1	-> 73
word 03	1	1	1	0	1	1	0	0	-> EC

-----  
0307

テストベクタフィールドは、シミュレータで作成します。このフィールドには、各デバイスピンの機能テスト情報が記述されます。QV バリュースフィールドにより、ファイルに記述されているテストベクタの数が定義されます。テストベクタには、0001 から番号が付けられ、テストされるデバイスに番号順に適用されます。表 11-3 に有効なピンの条件を示します。

0	-	入力を LO(0 volts)へ駆動
1	-	入力を HI(+5 volts)へ駆動
C	-	入力を LO,HI,LO に駆動
K	-	入力を HI,LO,HI に駆動
L	-	出力 LO(0 volts)をテスト
H	-	出力 HI(+5 volts)をテスト
Z	-	ハイインピーダンスの出力をテスト
X	-	定義されていない入力、テストされない出力
N	-	電源ピンとテストされない出力
P	-	ブリロードレジスタ

表 11-3 テスト条件

ベクタの中で表わされるテスト条件はピンオーダーフィールド(P)で定義される順番でデバイスピンに適用されます。この例(図 C-1)では、最初の条件はピン 1 に適用され最後の条件は 20 ピンデバイスのピン 20 に適用されます。C や K の駆動信号は他の入力安定してから与えられます。L や H、Z 条件は入力がすべて安定してからテストされます。

クロックピンの駆動信号 P は、スーパー電圧のプリロードレジスタの機能をもつデバイスでしか使用できません。TTL レベルのプリロードピンを使用するデバイスでは、レジスタにプリロードするためにこれらのピンで C、K 駆動信号を使用する必要があります。

送信は、4 つの ASCII hex 文字の送信チェックサム(サムチェック)に続く、表示されない ASCII ETX 文字で終わります。チェックサムは STX 文字で始まり ETX 文字で終わるトランスミット文字の ASCII 値の 16 ビットの総数です。サンプルファイル(図 C-1)では、送信チェックサムは、各行の最後のキャリッジリターンとラインフィードを考慮して 46C9 になります。

### ASCII-Hex フォーマット

ASCII-hex フォーマットは PROM で使用されます。このフォーマットのデータは、実行文字(スペース)で区切られた連続するバイトで構成されます。実行文字のすぐ前の文字がデータバイトと解釈されます。フォーマットは、一桁の 16 進数(x4PROM)か二桁の 8 進数でデータバイトを表わします。

送信は、ASCII STX [Ctrl]-[B]文字で始まります。16 個のデータバイトは \$ や A、カンマ(\$A,)に続く 4 桁の 16 進数アドレスから始まります。ASCII ETX [Ctrl]-[C]で送信のデータ部が終わります。この次に 40 個のスペースが入ります。

図 11-2 に hex ファイルのサンプルを示します。

```

^B
$A0000,00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
$A0010,10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
$A0020,20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
$A0030,30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
^C
$S07E0,

```

図 11-2 Hex ファイルのサンプル

### HL フォーマット

HL ダウンロードフォーマットは Signetics IFL デバイスで使用されます。各デバイスにはそれぞれのフォーマットがあります。フォーマットはすべて STX [Ctrl]-[B]で始まり、ETX [Ctrl]-[C]で終わります。以下のセクションで IFL デバイスの各タイプのフォーマットについて説明します。

## 82S100/101 FPLA

アクティブレベル識別子\*A に続き、F7 から F0 のアクティブレベルの状態が記述されます。ここでは、H がアクティブハイを表わし、L がアクティブローを表わします。プロダクトタームは\*P 識別子に続くスペースと P ターム番号で表わされます。入力変数識別子\*I に続き、入力変数 I15 から I0 が記述され出力関数識別子\*F に続き F7 から F0 が記述されます。

図 11-3 に、このフォーマットのサンプルを示します。

```
^B
*A LHLHLHLH
*P 00 *I HHHHLLLLHHHHLLLL *F A.A.A.A.
.
.
*P 47 *I LLLLLLLLLLLLLLLLLL *F ....AAAA
^C
```

図 11-3 82S100/101 FPLA ファイルのサンプル

## 82S103 FPGA

プロダクトタームは、\*G 識別子で始まりこれに続いて、スペースとターム番号が記述されます。さらに、アクティブレベル識別子\*A とアクティブレベルのデータが続きます。入力変数識別子\*I に続き I15 から I0 の入力変数が記述されます。

図 11-4 にこのフォーマットのサンプルを示します。

```
^B
*G 00 *A L *I HHHHLLLLHHHHLLLL
.
.
*G 47 *A H *I LLLLLLLLLLLLLLLLLL
^C
```

図 11-4 82S103 FPGA ファイルのサンプル

## 82S105 FPLS

プリセット/出力イネーブルオプションが、\*A 識別子に続く H(プリセット) か L(出力イネーブル)で入力されます。トランジションタームは、ターム識別子\*T に続くターム番号で記述されます。コンプリメントアレイ識別子\*C の後にこのタームの値が記述されます。タームの入力変数は入力変数識別子\*I をつけて表わされ、\*I に続いて I15 から I0 の入力変数のデータが記述されます。現在のフリップフロップのステートは\*P に続く P5 から P0 で与えられます。次のステートの値は、\*N に続く N5 から N0 で与えられます。出力関数は\*F に続く F7 から F0 で表わされます。

図 11-5 にこのフォーマットのサンプルを示します。

```
^B
*AL
```

```

*T 00 *C.*ILLLLHHHLLLLLHHH*PHHLLLL
      *NHHLLLL *F HHHHHHHH
.
.
*T 47 *C A *I LL----- *P LLLLLL
      *N HHHHHH *F HHHLLL-
^C

```

図 11-5 82S105 FPLS ファイルのサンプル

## 82S151 FPGA

I/O ピンの方向は\*DIR に続いて記述され、出力極性は\*POL 識別子に続いて記述されます。プロダクトタームは、\*P 識別子に続くスペースと P - ターム番号で表わされます。

コントロールタームの番号は\*D で始まり、その後ターム番号が記述されます。入力変数識別子\*I に続き入力変数 I5 から I0 が記述され、I/O フィードバック識別子\*B に続いて B11 から B0 が記述されます。

図 11-6 にこのフォーマットのサンプルを示します。

```

^B
*DIR HHLHLHHLHLHL *POL HHHHLLLLLHHL
*P 00 *I HHLLLL *B HL--HL--LLHH
.
.
*P 11 *I LLLLLL *B HHHHLLLLLHHL
*D 02 *I ----HH *B ----LLLLLL-
.
.
*D 00 *I LLLL-- *B HHHHHHLLLL-
^C

```

図 11-6 82S151 FPGA ファイルのサンプル

## 82S153 FPLA

出力極性識別子\*POL に続き、出力 B9 から B0 のアクティブレベルの状態が記述されます。ここで、Hはアクティブハイを表わし、Lはアクティブローを表わします。プロダクトタームは、\*P 識別子に続くスペースと P - ターム番号で表わされます。コントロールターム番号はDで始まり、その後ターム番号が記述されます。入力変数識別子\*I に続き、入力変数 I7 から I0 が記述されます。フィードバック変数 B9 から B0 は、\*BI 識別子に続いて記述され、出力関数 B9 から B0 は、\*BO 識別子に続いて記述されます。

図 11-7 にこのフォーマットのサンプルを示します。

```

^B
*POL HHLHLHHLHH
*P 00 *I --HH--LL *BI --HL----- *BO A..A..A..A
.

```

```

.
*P 31 *I -----HH *BI HLHLHLHLHL *BO ....AA....
*P D9 *I --HHHHHH *BI ----HHHHL
.
.
*P D0 *I LLLLLLLLLL *BI -----
^C

```

図 11-7 82S153 FPLA ファイルのサンプル

## 82S155 FPLS

グループ A と B の出力イネーブルモードは、\*E 識別子に続いて記述されます。各レジスタのフリップフロップモードは、\*F/F 識別子に続いて記述されます。出力ピンの極性は、\*POL 識別子に続いて記述されます。トランジションタームは、ターム識別子\*T に続くターム番号により表わされます。コンプリメントアレイ識別子\*C に続いてこのタームの値が記述されます。タームの入力変数は、入力変数識別子\*I に続く I3 から I0 で与えられます。I/O フィードバックデータは、\*B 識別子に続いて記述されます。フリップフロップの現在の状態は、\*QP 識別子に続く Q3 から Q0 で表わされます。次の状態の値は、\*ON 識別子に続く O3 から O0 で表わされます。グループ PB と PA のプリセットタームは、プリセット識別子\*P に続いて記述されます。グループ RB と RA のリセットタームは、リセット識別子\*R に続いて記述されます。出力関数は、\*BO に続く B7 から B0 で表わされます。フリップフロップのコントロールやリセット、プリセット、ロード、出力の各イネーブルが続きます。

図 11-8 にこのフォーマットのサンプルを示します。

```

^B
*E AA *F/F A.A. *POL HLHLHLHL
*T 00 *C . *I HLLL *BI HL--HLHL *QP LH-
      *QN LLHH *P .. *R .. *BO .A.A.A.A
.
.
*T 31 *C A *I LLHH *BI --HLHL *QP HLLL
      *QNHHHH *P .A *R .A *BO ..A.AA.A
*T FC *C . *I LLLL *BI LLLLHHHH *QP LLHH
*T LB *C . *I HLLL *BI --LL--LL *QP HHHH
*T LA *C . *I LL-- *BI LLLLHHHH *QP --LL
*T D3 *C . *I LLLL *BI LLLLLLLL- *QP LLHH
.
.
*T D0 *C . *I LLLL *BI LLHHHHLL *QP HLLH
^C

```

図 11-8 82S155 FPLS ファイルのサンプル

## 82S157 FPLS

グループ A と B の出力イネーブルモードは、\*E 識別子に続いて記述されま

す。各レジスタのフリップフロップモードは、\*F/F 識別子に続いて記述されます。出力ピンの極性は、\*POL 識別子に続いて記述されます。トランジションタームは、ターム識別子\*Tに続くターム番号により表わされます。コンプリメントアレイ識別子\*Cがこのタームの値に続いて記述されます。タームの入力変数は、入力変数識別子\*Iに続く I3 から I0 で与えられます。I/O フィードバックデータは、\*BI 識別子に続いて記述されます。フリップフロップの現在の値は、\*QP に続く Q5 から Q0 で与えられます。次のステートの値は、\*QN に続く Q5 から Q0 で与えられます。グループ PA のプリセットタームは、プリセット識別子\*P に続いて記述されます。グループ PA のリセットタームは、リセット識別子\*R に続いて記述されます。出力関数は、\*BO に続く B5 から B0 で表わされます。フリップフロップのコントロールやリセット、プリセット、ロード、出力の各イネーブルがそれに続きます。

図 11-9 にこのフォーマットのサンプルを示します。

```

^B
*E AA *F/F A.A. *POL HLHLLHLH
*T 00 *C . *I HHLL *BI HL-HL *QP LH-HL
  *QN LLHHHL *P . *R . *BO .A.A.A
.
.
*T 31 *C A *I LLHH *BI -HLHL *QP HHLLHH
  *QNHHHLL *P A *R A *BO ..AA.A
*T FC *C . *I LLLL *BI LLLLHH *QP LLHHHH
*T PB *C . *I ---- *BI ----HH *QP ----LL
*T RB *C . *I HHHL *BI HHLLLL *QP HLLLHH
*T LB *C . *I HLLL *BI --L-LL *QP HHHHLL
*T LA *C . *I LL-- *BI LLLHHH *QP --LLHH
*T D3 *C . *I LLLL *BI LLLL- *QP LLHH-
.
.
*T D0 *C . *I LLLL *BI LLHHLL *QP HLLHLL
^C

```

図 11-9 82S157 FPLS ファイルのサンプル

## 82S159 FPLS

グループ A と B の出力イネーブルモードは、\*E 識別子に続いて記述されます。各レジスタのフリップフロップモードは、\*F/F 識別子に続いて記述されます。出力ピンの極性は、\*POL 識別子に続いて記述されます。トランジションタームは、ターム識別子\*Tに続くターム番号により表わされます。コンプリメントアレイ識別子\*Cがこのタームの値に続いて記述されます。タームの入力変数は、入力変数識別子\*Iに続く I3 から I0 で与えられます。I/O フィードバックデータは、\*BI 識別子に続いて記述されます。フリップフロップの現在の値は、\*QP に続く Q7 から Q0 で与えられます。次のステートの値は、\*QN に続く Q7 から Q0 で与えられます。出力関数は、\*BO に続く B7 から B0 で表わされます。フリップフロップのコントロールやリセ

ット、プリセット、ロード、出力の各イネーブルがそれに続きます。

図 11-10 にこのフォーマットのサンプルを示します。

```
^B
*E AA *F/F A.A.A.A. *POL LHLH
*T 00 *C . *I HHLL *BI HL-- *QP HHLLHH-
  *QN LLHHLLHH *BO .A.A
.
.
*T 31 *C A *I LLHH *BI ---- *QP --HHHLLL
  *QN LLLLHHHH *BO ...A
*T FC *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T PB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T RB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T LB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T PA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T RA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T LA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T D3 *C . *I LLLL *BI LLLL *QP LLLLHHHH
.
.
*T D0 *C . *I LLLL *BI LLLL *QP LLLLHHHH
^C
```

図 11-10 82S159 FPLS ファイルのサンプル

## 82S161 FPLA

アクティブレベル識別子\*A に続いて、F7 から F0 のアクティブレベルの状態が記述されます。ここで、Hはアクティブハイを表わし、Lはアクティブローを表わします。プロダクトタームは、\*P 識別子に続くスペースと P-ターム番号で表わされます。入力変数識別子\*I に続いて入力変数 I11 から I0 が記述され、出力関数識別子\*F に続いて F7 から F0 が記述されます。

図 11-11 にこのフォーマットのサンプルを示します。

```
^B
A LHLHLHLH
*P 00 *I LLLLHHHHLLLL *F A.A.A.A.
.
.
*T 47 *I LLLLLLLLLLLLLL *F ....AAAA
^C
```

図 11-11 82S161 FPLA ファイルのサンプル

## 82S162 FPGA

出力極性識別子\*POL に続いて出力 F4 から F0 のアクティブレベルの状態が記述されます。プロダクトタームは、\*G 識別子で始まり、スペースとターム番号が記述されます。入力変数識別子\*I に続いて入力値 I5 から I0 が記述

されます。

図 11-12 にこのフォーマットのサンプルを示します。

```
^B
*POL HHLL
*G 00 *I HHHHLLLLHHHHLLLL
.
.
*G 04 *I LLLLLLLLLLLLLLLLLL
^C
```

図 11-12 82S162 FPGA ファイルのサンプル

### 82S163 FPGA

出力極性識別子\*POL に続き出力 F8 から F0 のアクティブレベルの状態が記述されます。プロダクトタームは、\*G 識別子で始まりスペースとターム番号が続いて記述されます。入力変数識別子\*I に続いて、入力変数 I11 から I0 が記述されます。

図 11-13 にこのフォーマットのサンプルを示します。

```
^B
*G 00 *I HLLLHHHHLLLL
.
.
*G 08 *I LLLLLLLLLLLLLL
^C
```

図 11-3 82S163 FPGA ファイルのサンプル

### 82S167 FPLS

プリセット/出力イネーブルオプションが、\*A 識別子に続く H(プリセット) か L(出力イネーブル)で入力されます。トランジションタームは、ターム識別子\*T に続くターム番号で記述されます。コンプリメントアレイ識別子\*C の後にこのタームの値が記述されます。タームの入力変数は入力変数識別子\*I をつけて表わされ、\*I に続いて I13 から I0 の入力変数のデータが記述されます。現在のフリップフロップのステートは\*P に続く P7 から P0 で与えられます。次のステートの値は、\*N に続く N7 から N0 で与えられます。出力関数は\*F に続く F3 から F0 で表わされます。

図 11-14 にこのフォーマットのサンプルを示します。

```
^B
*AL
*T 00 *C . *ILLHHHHLLLLHHHH *PHHHLLLLHH
          *NHHHLLLLLL *F HHHH
.
.
*T 47 * A *I LL----- *P --LLLLLL
*N HHHLLHHHH *F HL-
```



^C

図 11-14 82S167 FPLS ファイルのサンプル

### 82S168 FPLS

プリセット/出力イネーブルオプションが、\*A 識別子に続く H(プリセット) か L(出力イネーブル)で入力されます。トランジションタームは、ターム識別子\*T に続くターム番号で記述されます。コンプリメントアレイ識別子\*C の後にこのタームの値が記述されます。タームの入力変数は入力変数識別子\*I をつけて表わされ、\*I に続いて I11 から I0 の入力変数のデータが記述されます。現在のフリップフロップのステートは\*P に続く P9 から P0 で与えられます。次のステートの値は、\*N に続く N9 から N0 で与えられます。出力関数は\*F に続く F3 から F0 で表わされます。

図 11-15 このフォーマットのサンプルを示します。

```
^B
*AL
*T 00 *C.*ILLHHHLLLHHHH*PHHHLLLHLH
      *NHHLHHLLLLL *F HHHH
.
.
*T 47 *C A *I LL----- *P --LLLLLL-
*N HHHLLHHHH-L *F HL-
^C
```

図 11-15 82S168 FPLS ファイルのサンプル

### 82S173 FPLA

出力極性識別子\*POL に続き、出力 B9 から B0 のアクティブレベルの状態が記述されます。ここで、Hはアクティブハイを表わし、Lはアクティブローを表わします。プロダクトタームは、\*P 識別子に続くスペースとP -ターム番号で表わされます。コントロールターム番号はDで始まり、その後ターム番号が記述されます。入力変数識別子\*I に続き、入力変数 I11 から I0 が記述されます。フィードバック変数 B9 から B0 は、\*BI 識別子に続いて記述され、出力関数 B9 から B0 は、\*BO 識別子に続いて記述されます。

図 11-16 にこのフォーマットのサンプルを示します。

```
^B
*POL HHHLLHHLLH
*P 00 *I --H--LLHH *BI --HL----- *BO A..A..A..A
.
.
*P 31 *I LL-----HH *BI HLHLHLHLHL *BO ....AA....
*P D9 *I LL--HHHHHH *BI ---HHHHLL
.
.
*P D0 *I --LLLLLLLLL *BI -----
```

図 11-16 82S173 FPLA ファイルのサンプル

## 82S179 FPLS

グループ A と B の出力イネーブルモードは、\*E 識別子に続いて記述されます。各レジスタのフリップフロップモードは、\*F/F 識別子に続いて記述されます。出力ピンの極性は、\*POL 識別子に続いて記述されます。トランジションタームは、ターム識別子\*T に続くターム番号により表わされます。コンプリメントアレイ識別子\*C がこのタームの値に続いて記述されます。タームの入力変数は、入力変数識別子\*I に続く I7 から I0 で与えられます。I/O フィードバックデータは、\*BI 識別子に続いて記述されます。フリップフロップの現在の値は、\*QP に続く Q7 から Q0 で与えられます。次のステートの値は、\*QN に続く Q7 から Q0 で与えられます。出力関数は、\*BO に続く B7 から B0 で表わされます。フリップフロップのコントロールやりセット、プリセット、ロード、出力の各イネーブルがそれに続きます。

図 11-17 にこのフォーマットのサンプルを示します。

```

^B
*E AA *F/F A.A.A.A. *POL LHLH
*T 00 *C . *I HHLLHHLH *BI HL-- *QP HHLLHH-
  *QN LLHHLLHH *BO .A.A
.
.
*T 31 *C *1 LLH-LHH *BI LLLL *QP -HHHLL
  *QN LLLLHHHH *BO ...A
*T FC *C . *1 LLLHHHHL *BI LLLL *QP LLLLHHHH
*T PB *C . *I LLLHLHLH *BI LLLL *QP LLLLHHHH
*T RB *C . *I LLLLLLHL *BI LLLL *QP LLLLHHHH
*T LB *C . *I LLLHHHHL *BI LLLL *QP LLLLHHHH
*T PA *C . *I LLLLHLHL *BI LLLL *QP LLLLHHHH
*T RA *C . *I LLLL---- *BI LLLL *QP
*T LA *C . *I LLLL---H *BI LLLL *QP LLLLHHHH
*T D3 *C . *I LLLLLLLL *BI LLLL *QP LLLLHHHH
.
.
*T DO *C . *I HHHHHHHH *BI LLLL *QP LLLLHHHH
^C

```

図 11-17 82S179 FPLS ファイルのサンプル

## ドキュメンテーションファイルフォーマット

このセクションでは、ドキュメンテーションファイル(ファイル名.DOC)のフォーマットについて説明します。このファイルには、ヒューズプロット情報も記述されます。ドキュメンテーションファイルを生成するには、コンパイラを実行する前に Doc File オプションの Equations をイネーブルして下さい。Doc File オプションの Fuse Plot をイネーブルするとドキュメンテ

ーションファイルにヒューズプロット情報を生成することができます。図 11-18 にドキュメンテーションファイルのサンプルを示します。

```

WAITGEN.DOC
*****
                Waitgen
*****
CUPL  3.0 Serial# 9-99999-999
Device
Created
Name
Partno
Revision
Date
Designer
Company
Assembly
Location
=====
                Expanded Product Terms
=====
wait1.d =>
    !memr
    # a15
    # a14
    # a13
    # reset
select_rom =>
    !a13 & !a14 & !a15 & memr
wait2.d =>
    !memr
    # a15
    # a14
    # a13
    # !wait1
memadr =>
    a15 , a14 , a13 , a12 , a11
ready =>
    !wait2
ready.oe =>
    !a13 & !a14 & !a15 & memr
rom_os =>
    !a13 & !a14 & !a15 & memr
memreq =>
    memw
    # memr
ram_cs0 =>
    !a11 & !a12 & a13 & !a14 & !a15 & memw
    # !a11 & !a12 & a13 & !a14 & !a15 & memr

```

```

ram_cs1 =>
    a11 & !a12 & a13 & !a14 & !a15 & memw
    a11 & !a12 & a13 & !a14 & !a15 & memr

rom_cs.oe =>
    1

ram_cs0.oe =>
    1

ram_cs1.oe =>
    1

```

# Symbol Table

Pin	Variable				Pterms	Max	Min
Pol	Name	Ext	Pin	Type	Used	Pterms	Level
	wait1	15	V	-	-	-	
	wait1	d 15	X	5	8	1	
	a11	6	V	-	-	-	
	select_rom	0	I	1	-	-	
	wait2	14	V	-	-	-	
	wait2	d 14	X	5	8	1	
	a12	5	V	-	-	-	
	a13	4	V	-	-	-	
	a14	3	V	-	-	-	
	a15	2	V	-	-	-	
!	oe	11	V	-	-	-	
!	memr	8	V	-	-	-	
	memadr	0	F	-	-	-	
	ready	18	V	1	7	1	
	ready	oe 18	X	1	1	1	
!	memw	7	V	-	-	-	
	cpu_clk	1	V	-	-	-	
!	rom_cs	19	V	1	7	1	
	reset	9	V	-	-	-	
	memreq	0	I	2	-	-	
!	ram_cs0	12	V	2	7	1	
!	ram_cs1	13	V	2	7	1	
	rom_cs	oe	19	D	1	1	0
	rom_cs0	oe	12	D	1	1	0
	rom_cs1	oe	13	D	1	1	0

LEGEND F:field D:default variable T:function  
 N:node I:intermediate variable V:variable  
 X:extended variable U:undefined  
 W:extended node

```

=====
                        Fuse Plot
=====
Pin #19
  0000 -----
  0032 -x---x-----x-----
  0064 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0096 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0128 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0160 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0192 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0224 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #18
  0256 -x---x---x-----x-----
  0258 -----x-----
  0320 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0352 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0384 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0416 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0448 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0480 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #17
  0512 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0544 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0576 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0608 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0640 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0672 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0738 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin#16
  0788 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0800 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0832 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0864 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0898 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0928 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0980 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  0992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15
  1024 -----x-----
  1056 x-----
  1088 ----x-----
  1120 -----x-----
  1152 -----x-----
  1184 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  1216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  1248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

```

Pin #14
1280 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1312 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1344 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1378 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1408 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1440 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1472 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1504 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

```

Pin#13
1536 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1568 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1600 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

```

Pin #12
1792 -----
1824 -x---x--x---x---x---x-----
1856 -x---x--x---x---x-----x-----
1888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

```

LEGEND      X : fuse not blown
            - : fuse blown

```

# Chip Diagram

```

=====
*****
*      * *      *
****          ****
cpu_clk * 1          20 * VCC
****          ****
*      Waitgen      *
****          ****
a15 * 2          19 * !rom_cs
****          ****
*              *
****          ****
a14 * 3          18 * ready
****          ****
*              *
****          ****

```

```

a13 * 4          17 *
****          ****
*              *
****          ****
a12 * 5          16 *
****          ****
*              *
****          ****
a11 * 6          15 * wait1
****          ****
*              *
****          ****
!memw * 7          14 * wait2
****          ****
*              *
****          ****
!memr * 8          13 * ram_cs1
****          ****
*              *
****          ****
reset * 9          12 * !ram_cs0
****          ****
*              *
****          ****
GND * 10          11 * !oe
****          ****
*              *
*****

```

図 11-18 ドキュメンテーションファイルのサンプル

ファイルの最初の部分には、ファイル管理情報やファイル履歴情報の対応する CUPL ソースファイルと同じヘッダー情報が記述されます。また、デバイスライブラリやコンパイラプログラムのバージョン情報やファイルが作成された日付や時間の情報が記述されます。

ファイルの次のセクション、Expanded Product Terms には、論理記述ファイルの式からコンパイラにより生成されたプロダクトタームが記述されます。アドバンスト PLD パッケージの中の WAITGEN.PLD は、図 11-18 のドキュメンテーションファイルのサンプルの元になった論理記述ファイルです。内容を見れば、元の論理式とコンパイラにより生成されたプロダクトタームを比較することができます。

指定されたデバイスのプロダクトタームはコンパイラにより生成されます。例えば、PAL16L8 などのデバイスの中には、固定された反転バッファを持つものがあり、デバイスに論理を適合させるためにコンパイラはドモルガンの定理を使用する場合があります。例えば、論理記述ファイルが PAL16L8 向けに作成される場合、ピンリストの中の出力はすべてアクティブ HI で宣言される必要があります。以下の式により OR 関数が指定されます。

$$c = a \# b ;$$

しかし、PAL16L8 には固定反転バッファがあります。反転バッファを変更することはできないので、コンパイラは OR 式にドモルガンの定理を適用して以下の式を生成し論理をデバイスに適合させます。

$$c \Rightarrow !a \ \& \ !b$$

ファイルの次のセクション、Symbol Table には、論理記述ファイルの各変数に関する情報が記述されます。この中の情報には、ピン番号や拡張子、変数のタイプ、使用できるプロダクトタームの番号、使用されているプロダクトタームの番号、コンパイラにより使用された最小化レベルがあります。

デバイスで使用できる最大のプロダクトタームを越える場合、コンパイラは、コンパイル時のピンに名前を付ける処理を行なっている時にエラーメッセージを表示します。しかし、メッセージには限界がどのくらいかは表示されません。シンボルテーブルのプロダクトタームに関するこれらの情報(図 11-18 参照)により、使用できるプロダクトタームの数が大きく限度を越えているのか、それともわずかに越えているのかが分かります。

## PDF ファイルフォーマット

このセクションでは、コンパイラにより生成される PDF(P-CAD Database Interchange Format)ファイル(ファイル名:PDF)の使用法について説明します。PDF-フォーマットファイルは、コンパイル時に PDF PDF オプションをイネーブルすることにより生成されます。

PDF フォーマットは、P-CAD スケマチックキャプチャプログラム PC-CAPS のインタフェースとして使用されます。これは、コンパイラに生成される PDF-フォーマットファイルを PDF-IN プログラムを使用して PC-CAPS シンボルに変換することにより行ないます。このシンボルは PLD 設計の論理表現を表わします。この中には、ピンパッケージ情報や PCB リファレンスデシグネータ、PLD タイプ、設計名が記述されます。

図 11-19 に、PDF-IN により生成された PC-CAP シンボルの例を示します。

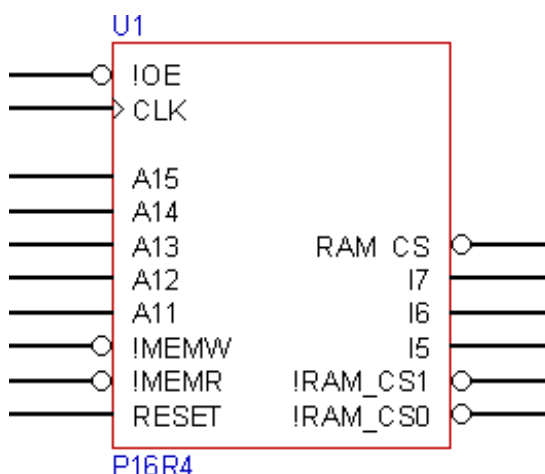




図 11-19 PDIF-IN により生成された PC-CAPS シンボル

PDIF-IN プログラムや PC-CAPS プログラムの実行に関する詳細は PDIF ユーザーズマニュアルと PC-CAPS ユーザーズマニュアルを参照して下さい。

## バークレイ PLA ファイルフォーマット

このセクションでは、バークレイ PLA ファイル(ファイル名.PLA)のフォーマットについて説明します。バークレイ PLA フォーマットは、バークレイ PLA ツールなどの PLA 論理合成ツールのインターフェースフォーマットとして使用されます。バークレイ PLA-フォーマットファイルは、コンパイル時に Berkely PLA オプションをイネーブルすると生成できます。図 11-20 に、バークレイ PLA-フォーマットファイルのサンプルを示します。

ファイルの最初の部分には、ファイルの管理情報や履歴情報が記述されます。#文字によりコメントが示されます。この情報は、対応する CUPL ソースファイルのヘッダー情報と同じです。また、デバイスライブラリやコンパイラプログラムのバージョン情報やファイルが作成された日付や時間の情報が記述されます。

次のセクションは、(アドバンスド PLD パッケージにある)論理記述ファイル COUNT10.PLD の式からコンパイラにより生成される PLA 記述で構成されます。その内容を見て、元の論理記述式をコンパイラにより生成された PLA 記述と比較することができます。

PLA 記述は、入力.i や出力.o、プロダクトターム.p、プロダクトターム毎に一行に記述される AND や OR プレーンの数を定義するフィールドで構成されます。AND プレーンの接続は、1 が非反転入力で 0 が反転入力として表わされます。接続のない入力は-で表わされます。OR プレーンの接続は 1 で表わされ、接続されていない入力は 0 で表わされます。PLA 記述の最後は.end で示されます。

図 11-20 にバークレイ PLA-フォーマットファイルのサンプルを示します。

```
#を使用して生成されたバークレイ PLA フォーマット
# CUPL                      3.0 Serial# 9-99999-999
# Device                    pl6rp4 Library DLIB-g-24-15
# Created                   Thu Feb 26 13:45:23 1987
# Name                      Count10
# Partno                    CA0018
# Revision                  01
# Date                      07/16/87
# Designer                  Kahl
# Company                   Assisted Technology
# Assembly                  None
# Location                  None
# Inputs 1 Q0 Q1 Q2
#           Q3 clr dir
# Outputs Q0.d Q1.d Q2.d Q3.d
#           carry carry.oe
```

```
.i 7
.o 6
.p 18
-00010- 100000
-0--00- 100000
-000101 010000
-10-000 010000
-01-000 010000
-11-001 010000
-001001 010000
-000101 001000
-110000 001000
-01000- 001000
-1-1001 001000
-01100- 001000
-100101 000100
-000001 000100
-111000 000100
-000100 000100
-1001-- 000010
1----- 000001
.end
```

図 11-20 バークレイ PLA フォーマットファイルのサンプル