The CUPL Environment

Note that this is not a complete instructional manual as it only contains the items necessary to create most general types of programs. For more in depth information, please refer to the manual or the online help file.

Areas to be covered:

File extensions that will be useful to know

PLD	Created by the user Contains all of the logic instructions necessary for your device, i.e. the cupl program itself.
DOC	Generated by CUPL Contains all of the logic equations that CUPL generated from your program Tells you errors encountered in compiling the program (including location) Provides information about how it fit the compiled logic into the selected device.
ABS	Generated by CUPL File used by CUPL to perform the simulation
LST	Generated by CUPL Error Listing file that contains all the original lines of code numbered. All errors are listed at the end with offending line number.
JED	Generated by CUPL File used by the programmer to actually burn the chip Filename comes from the Name field in the pld file header
SI	Simulation Input file created by user Contains your list of test vectors
SO	Simulation Output file generated by CUPL Contains the results of the simulation run including any errors Used for the graphical display of your simulation results

CUPL Programming Language

General Conventions

Variables

- are case sensative
- cannot have spaces
- cannot use reserved words or symbols

Reserved Words (not case sensitive)

APPEND	ASSEMBLY	ASSY	COMPANY	CONDITION
DATE	DEFAULT	DESIGNER	DEVICE	ELSE
FIELD	FLD	FORMAT	FUNCTION	FUSE
GROUP	IF	JUMP	LOC	LOCATION
MACRO	MIN	NAME	NODE	OUT
PARTNO	PIN	PINNNODE	PRESENT	REV
REVISION	SEQUENCE	SEQUENCED	SEQUENCEJK	SEQUENCERS
SEQUENCET	TABLE			

Reserved Symbols & Symbol sets

&	#	(	)	-
*	+	[	]	/
:	.	..	/*	*/
;	,	!	'	=
@	$	^		

Indexed Variables

- Example: [A0, A1, A2, A3] might be a 4 bit address
- Indexing should always start from 0, not 1 or such.
- Another notation for these is [A0..3] when using the entire group
- The index cannot be greater than 31

Numbers can be represented in binary, octal, decimal, or hexadecimal. Pin numbers and indexed variable will always be represented in decimal. The default base for all other number is hexadecimal. To indicate a particular base, precede the number will the particular prefix (which is not case sensitive).

Number	**Base**	**Decimal Value**
'b'0	binary	0
'b'1101	binary	13
'o'663	octal	435
'D' 92	decimal	92
'h' BA	hexadecimal	186
'b'[001..100]	binary	range from 1 to 4

Binary, octal and hex numbers can also have don't care values.

Number	**Base**
'o'0X6	octal
'b'11X1	binary
'h'[3FXX..7FFF]	hexadecimal (range)

Commenting is done the same as in ANSI C where your comment is enclosed between the /* and */ symbol pairs.
Note: Double slash (//) is not valid for comments.

Most instruction statements end with a semicolon.

CUPL program format

There are three parts to a CUPL file (a .pld file)

- Header
- Declarations
- Main body

If you use the template file (or what will automatically come up for a new file), the basic format will be already set up.

Header

The following are the basic header fields:

```
Name       XXXXX;
Partno     XXXXX;
Date       XX/XX/XX;
Revision   XX;
Designer   XXXXX;
Company    XXXXX;
Assembly   XXXXX;
Location   XXXXX;
Device     XXXXX;
```

All of the header fields are required to be present in the file, but only the Name field is required to have a real value in it. It will be used as the name for the JEDEC output file (the file you will use with the programmer) so it needs to be a valid DOS name (i.e. max of 8 characters). The compiler will perform a check to verify all of the fields are present.

The Device field allows you to set the default device type used in compilation. I would suggest using this so you don't have to worry about setting the device manually before compiling (see the device selection section). The name to put in here will be the name you would have selected manually. For example, a GAL22V10 would be "g22v10" and a GAL16V8 would be "g16v8". Note that this field is not required.

Declarations

The standard place to set up all your variables for the program.

Pin Assignment

This section allows you specifically assign variable names to individual pins on the part. You need to make sure that your assignment of pins does not conflict with device specification of pins (i.e. clock, input only, input or output, etc.). You will have to refer to the part documentation for information.

```
e.g.
  Pin 1 = !a;  /*Assign pin 1 to be "not a".*/
               /*The optional '!' defines the polarity.*/
  Pin [2..5] = [b, c, d, e];
```

```
                /* assign pin 2=b, pin 3=c, pin 4=d, pin5=e*/
  Pin [6,10] = [f,g];    /*Assign pin 6 to f and pin 7 to g.*/
```

The polarity selection allows you to choose whether the signal is active high or active low. Default, without a bang, is active high. This allows you to simply write equations without having to worry about if the signal is active high or low. You can simply refer to it as being asserted or not.

```
e.g.
  Pin 2 = !A;
  Pin 3 = !B;
  Pin 16 = Y;

  Y = A & B;  /*Y will be true (high) when both A and B are */
              /* true (both are low). */
```

You can also define your pins for a virtual device simply by not putting the pin numbers in. This will allow the compiler to pick pins for the signal. Even though this is possible, I would not suggest using it as it seems to have difficulty with the dynamic assignment and the polarity assignment doesn't apply.

Node Assignment

Nodes are used primarily for devices that contain functions not directly available to external pins, and subsequently unavailable for logic equations. These functions are called buried nodes of which state registers (flip flops) in the 82S105 are an example.
Note: For the case of the Gal22V10 and the Gal16V8, there are no buried nodes, so these would function exactly the same as the pin assignment above.

The two key commands that are used for this is Node and PinNode. For exact syntax refer to the manual and online help.

Bit Field Declaration

Bit Fields allow you to refer to a group of bits by a single variable.

```
e.g.
  FIELD Data = [D0..D7];  /*Assigns Data name to group of bits.*/
  FIELD Control = [strobe,ack,rdy];
                          /*Also works on individual pins.*/
```

Rules for Bit fields
1. Never mix indexed and non-indexed variables in a single field statement.
2. Don't use the same index of different indexed variable in a field declaration (i.e. A2 and B2 cannot be used in the same field variable).

MIN declaration Statements

These allow you to select the minimization level for specific variables, overriding the general minimization specified at compile time. This can be useful if you need to minimize certain variable more to make them fit in the part, but don't want to minimize everything else as much (See the compilation section for more information).

```
e.g.
  MIN async_out = 0;
  MIN outa = 2;
  MIN Data.D = 4;
```

Note that Data uses the D extension to specify that that the registered output is the one to be reduced.

Available levels to use:

Level	Amount of minimization (or method)
0	No minimization
1	Quick minimization
2	Quine McCluskey
3	Presto
4	Expresso

Preprocessor Commands

Just as in C programming you can use preprocessor commands. The most common command used is the $DEFINE. This works exactly the same as declaring constants in C.

```
e.g.
  $DEFINE ON 'b'1
  $DEFINE OFF �b�0
```

$MACRO and $REPEAT are two other that you might use. Other commands can be found in the documentation. Keep in mind that since these only for the preprocessor, they can be used in any part of the program.

Note: There is no semicolon at the end of the Preprocessor commands

```
e.g.
  $repeat i = [0..7]
  $define point{i}  'b'{i}
  $repend
```

This is the same as if the following had been entered.

```
  $define point0 'b'000
  $define point1 'b'001
  $define point2 'b'010
  $define point3 'b'011
  $define point4 'b'100
  $define point5 'b'101
  $define point6 'b'110
  $define point7 'b'111
```

Arithmetic Operations

CUPL does support some standard mathematic functions (including the logarithmic ones below).
NOTE: These can only be used as part of the $REPEAT and $MACRO preprocessor commands. They must also appear in braces {}.

Operator	Example	Description
**	2**3	Exponential

*	2*1	Multiplication
/	4/2	Division
%	9%8	Modulus
+	2+4	Addition
-	4-1	Subtraction

Function	**Base**
LOG2	Binary
LOG8	Octal
LOG16	Hexadecimal
LOG	Decimal

The following is an example of arithmetic operations being used with the $REPEAT preprocessor command. In this case, it is all that is necessary for the state machine declaration and it coincides with the above define statements.

```
e.g.
  $repeat i = [0..7]
        present point{i}
            if score next point{(i+1)%8};
            default next point{i};
  $repend
```

This would be equivalent to entering:

```
  present point0
      if score next point1;
      default next point0;
  present point1
      if score next point2;
      default next point1;
  ...
  present point7
      if score next point0;
      default next point7;
```

Main body

The main body contains all of the logic necessary for your programmable part.

Variable Extensions

Extensions indicate specific functions associated with nodes of the part. The compiler checks the usage of each extension to verify that it is valid for the specific device (check the data sheets carefully). From the data sheets you can also determine exactly what you will need to use.

Useful extensions for the g22v10 and g16v8 are:

```
 .AR     /* Asynchronous Reset of flip-flop */
 .SP     /* Synchronous Preset of flip-flop */
 .OE     /* Output Enable */
 .D      /* D input of D-type flip-flop */
```

Other extensions can be found in the manual or online help.

Logic equations

Logical operators in their order of precedence.

Operator	Example	Description
!	!A	NOT
&	A&B	AND
#	A#B	OR
$	A$B	XOR

Equality Operation

The equality operator (:) allows you to check for bit equality between a set of variables and a constant.

```
e.g.
  field mode = [clr,dir];

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

This is equivalent to:

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

The equality operator can also be used with a set of variables that are to be operated on identically. This is valid for only the &, #, and $ operators.

```
e.g.
  [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
```

Append Statement

The Append statement allows you to assign multiple expressions to a single variable. Without this, only a single expression can be assigned to a variable. Append statements can also be appended together to generate more complex expressions.
Note: You are only allowed to use the above logic operators for generating your expressions.

```
e.g.
  APPEND Y = A0 & A1;
  APPEND Y = B0 & B1;
  APPEND Y = C0 & C1;
```

The three statements are equivalent to the following equation:

```
  Y = (A0 & A1) # (B0 & B1) # (C0 & C1);
```

The append statement is useful in adding additional terms (such as a reset) to state machine variable.

Truth Table

```
TABLE var_1 => var_2 {
       input_n => output_n;
            ...
            ...
       input_y => output_y;
       }
```

```
e.g.
   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;
   }
```

```
e.g.
   PIN [1..4] = [a12..15];                  /*upper 4 addresses*/
   PIN 12 = !RAM_sel;                       /*8Kx8 RAM*/
   PIN 13 = !ROM_sel;                       /*32Kx8 ROM*/
   PIN 14 = !timer_sel;                     /*8253 Timer*/
   FIELD address = [a12..15];
   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 */
   }
```

Notice that the outputs are all active low. Therefore in the truth table, when the output is 'b'100, only RAM_sel will be asserted (set to logic 0) while ROM_sel and timer_sel are deasserted (set to logic 1).

State Machine

```
 SEQUENCE state_var_list {              /* state_var_list is a list of the state bit */
     PRESENT  state_no statements;      /*   variables used in the state machine */
     ...                                /*   block.  Can be represented as a field */
     ...
     PRESENT state_nn statements;
 }
```

```
e.g.
   PIN [10..11] = [Count1..0];
   PIN 4 = up;
   PIN 5 = down;
   PIN 6 = enabled;
   PIN 13 = stuck;
   PIN 14 = two;
   FIELD counter = [Count1..0];
   $DEFINE s0  'b'00;
   $DEFINE s1  'b'01;
   $DEFINE s2  'b'10;
```

```
$DEFINE s3  'b'11;

sequence counter {
     present s0
          if up & enabled next s1;
          if down & enabled next s2;
          if enabled & !up & !down next s0 OUT stuck
          if !enabled next s0 OUT stuck;
     present s1
          if up & enabled next s2;
          if down & enabled next s0;
          default next s1 OUT stuck;
     present s2
          if up & enabled next s0;
          if down & enabled next s1;
          default next s2 OUT stuck OUT two;
     present s3
          next s0;
}
```

To define multiple state machines, merely set up different variables.
The default condition is the complement of the other conditional statements so it can generate a complex expression. In the above example, the default statement of state s1 is equivalent to the last two if statements in state s0. See the following section to understand the OUT statement.

Output Statement

This allows you to specifically assert an output in the state machine (or condition statement below). As you can see in the previous example, an OUT statement was added to the end of the default conditions. As long as the state machine remains in the s1 state, the 'stuck' output will be asserted. Once the machine is enabled (asserted) and either up or down is asserted, then stuck will become deasserted. If an output is not explicitly stated as being asserted for the transition, then it will be deasserted. The default statement in state s2 shows how you would have a second output signal asserted while stuck in s2.

Condition Statement

The condition statement is equivalent to the asynchronous conditional output statement of the state machine, except there is no reference to any particular state.

```
 CONDITION {
    if expr0 OUT var1;
    ...
    ...
    if exprn OUT var2;
    default OUT var3;
}
```

```
e.g.
  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;
    default no_match;
  }
```

The above default statment is equivalent to:

```
IF !enable out no_match;
```

CUPL Simulator Language

The header of the simulation file should be exactly the same as the program file. Now you simply add two more statements to the end of this file: Order and Vectors. These should be placed at the end of the header as follows:

```
ORDER: Var, Var, Var;
VECTORS: X, X, X
```

Order lists the order of all of the inputs and outputs for the device.
Vectors then contains the list of the test vectors to be applied to previously stated inputs and outputs.

```
e.g.
  ORDER: Clock, Input1, Input2, Output;
  VECTORS: 1 0 X 0
           1 0 0 *
           1 0 1 1
```

Note: Formatting doesn't matter, it uses white spaces to determine which vector applies to which input or output. No semicolons are used anywhere in the vector list.

Values that can be used in the Vector table

Test Value	**Used with**	**Description**
0	Input	Drive low
1	Input	Drive high
C	Input	Drive (clock) low, high, low (in 1 cycle)
K	Input	Drive (clock) high, low, high (in 1 cycle)
L	Output	Test if low
H	Output	Test if high
Z	Output	Test for high impedance
X	Input or Output	High or Low (don�t care)
N	Output	not tested
*	Output	simulator determines what the value is
' '	Input	Encloses values to be expanded from a specified BASE
" "	Output	Encloses values to be expanded from a specified BASE

Use the BASE statement to specify what base you are using with the ' ' and " " vectors. Valid bases are: octal, decimal or hex. See the following example for implementation.

```
e.g.
  ORDER: Clock, Input1, Output1;
  BASE: hex
  VECTORS: C 'F' *
           C '0' *
```

Normally you will want to use the * on outputs to let the simulator tell you what it has calculated. This way you have an idea of what is actually going on inside. Otherwise, with the �L� and �H�, you just know whether or not it is correct. If it's not correct then you will just get CSIM errors.

This is all that is necessary to simulate your file. The graphical display will then use the SO file to generate the picture of your waveforms. If you would like to make the SO file itself easier to read and follow, then you can add more information to the SI file as follows:

```
e.g.
  ORDER: "clock is ", clock, "and input is", A2..0, ...
```

Produces the following results in the output file:

```
     0001: Clock is C and input is 000 ...
     0002: Clock is C and input is 001 ...
```

Usage of the CUPL Environment

There are several areas to look at in this section. Note that these instructions are only valid for version 4.7-4.9 of CUPL.

Selecting your device

- This is only necessary if you haven't put in the Device header item and are trying to do device specific compiling and simulation.
- If you do select device here, it will override any device listed in the header.
- Under the Options menu, choose the "Select Device" item.
- In the left hand windows, you can select the family of devices and the right hand windows allows you to select the exact device. Note that you don't have to exactly match the part name, like you do when programming.
- At the bottom you can also select if you are using a DIP or LCC package.
- If necessary, you can also unselect any devices (such as if you want to go back to using the default devices defined in the program header.

Selecting Compiling options

- This item allows you to select items specifically for compilations
- Under the Options menu, choose the "Compiler Options" item.

- The Logic Minimization drop down box allows you to select what minimization level you want.
  - This selection can be overridden on specific outputs by using the MIN statement.
  - Quine McCluskey is the best optimization method, but it is slow and more prone to run out of memory than the other methods. Use this only if necessary.
  - Another thing to be aware of is CUPL will compile away any hazard or glitch protection you may have designed in when set for a high level of minimization.

Level of Minimization	**Comments**
None	Disables logic minimization during a CUPL compilation. It is useful when working with PROMs, to keep contained product terms from being eliminated.
Quick	Balances the reduction efficiency, memory usage, and execution time.
Quine-McCluskey	Provides the highest level of reduction efficiency, but requires more memory and time to compile.
Presto	Provides a high level of reduction efficiency, but requires less memory and time to compile. This option will perform multiple output minimization in IFL devices. This maximizes product sharing in these types of devices.
Expresso	Provides a higher level of reduction efficiency than Presto, requires more memory to compile than Presto, but requires less time for compilation. As with Presto, this option will perform multiple output minimization in IFL devices. This maximizes product sharing in these types of devices.

- Miscellaneous Options
  - **Secure device**: Adds necessary code to allow the programmer to blow the security fuse of the device. Generally you won't want to do this since you aren't actually producing a product. I'm also not sure if this functionality is available with Allpro.
  - **Deactivate Unused OR Terms**: Normally on an OR gate array output, unused OR gate inputs are left connected to the product term array so that new terms may be added. By selecting this option, all unused inputs will be disconnected, thereby reducing propagation delay through the device.
  - **Simulate & Display Waveform**: These two options allow you to perform compilation and simulation all in one step. Make sure that the Display Waveform option is not checked unless you are also simulating. I would suggest that you don't use this as it can become a pain as you are trying to debug syntax of your program.
  - **One Hot bit State Machine**: To use this, you need to define each state with a one-hot bit code. Checking this will then allow CUPL to use slightly different optimization techniques. The results will vary depending on the part being used. Generally leave this unchecked.
  - **JEDEC Name = Filename**: This forces cupl to make the name of your output file (JED) the same as your program name (PLD) rather than using the name specified in the NAME field of the header.

- Optimization Button
  - **Best For Polarity**: Optimize product term usage for pin or pinnode variables.
  - **Demorgan**: DeMorganize all pin and pinnode variables.
  - **Keep XOR Equations**: Do not expand XOR to AND-OR equations. This is used for device independent designs or designs targeted for fitter-supported devices where the fitter supports XOR

  gates.
  - **P-Term Sharing**: Force product term sharing during minimization. This is also referred to as group reduction.

- Output File Button

  - Download
    - **JEDEC/POF/PRD**: Generates a JEDEC-compatible ASCII download file with the .JED extension. This is the standard type of file that we will be using.
    - **HL**: Generates an HL download file with the .HL extension. This format is available only for the Signetics IFL devices.
    - **ASCII/Hex**: Generates an ASCII-hex download file with the .HEX extension. This format is available only for PROMS.
  - Doc File Options
    - **Fuse Plot**: Generates a fuse plot in the documentation file. This is generally not very useful, unless you know how to read fuse plots. :)
    - **Equations**: Lists all of the equations generated in the documentation file. This is very important to have because you can see how CUPL interpreted your program in generating equations.
  - General
    - **Absolute**: Generates the file necesary for CUPL to be able to simulate your device. (ABS file)
    - **List**: Generates an error listing file with each line in the original source file numbered. Error messages are then listed at the end of the file and use the line numbers for reference. (LST file)
    - **Expanded Macro**: Generates an expanded macro definition file with the .MX extension containing an expanded listing of all macros used in the source file. It also contains the expanded expressions that use the REPEAT statement.
    - **PDIF**: Generates a PDF file to be used with P-CAD Schematic Capture to generate a symbol for your device.
    - **PLA**: Generates a PLS file to be used by PLA layout tools.
    - **PALASM**: Generates input files for other logic design tools and gate array fitters such as PDS2XNF from XILINX.
    - **XNF**: Generates a XILINX XNF file.

- Select Files Button: Allows you to select a pld file that can't be opened in the CUPL editor (it's limited to a 32k file)
- Select Device Button: This functions the same as the Select Device menu option.
- Select Library Button: Allows you to choose a different device library. No change should be necessary

Selecting Simulation options

- This allows you to specify options directly for the simulations.
- Under the options menu, choose the "Simulator Options" item.
- **Listing File**: Generates a simulator listing file with the input and output values for each variable are listed. Error messages are listed following each vector, with the signal name in error displayed. (SO file)
- **Append Vectors**: Appends the structured test vectors generated by the simulation onto the existing JEDEC download file. This should allow you to physically test the device with the programmer.
- **Display Results**: Displays the contents of the listing file in a window. An SO file is necessary for this to work.

Standard Compilation and Simulation Procedure

1. Open a CUPL file (PLD file) or create a new one.
2. Make sure that your device is correctly selected (menu, or in the program header)
3. In the Compiler options menu, check that the correct logic minimization is selected. Have everything unchecked in the Miscellaneous section. Have the JEDEC, equations, absolute, and list output file options selected.
4. Select Device Specific Compile under the Run menu.
5. Now debug all of your errors as listed in LST file and DOC file.
6. To Simulate, open or create the simulation file (SI file).
7. In the Simulator Options menu, make sure that the Listing File and Display Results Options are selected.
8. Select Device Specific Simulate under the Run menu.
9. If you get any simulation errors, check in the SO file. Otherwise, look at the waveforms in the new windows and see if they are correct. Press F4 to close this window or �?� to get more help.

Troubleshooting

Q. When I compile or simulate, it tells me that there were CSIM errors but then doesn't tell me what they are.
A. Look in the SO file. It will tell you what the error is.

Q. When I am compiling, I get CSIM errors, but I'm not trying to simulate it yet.
A. Turn off simulation in the Compiler Options menu.

Q. It tells me I have too many product terms when I compile it. Now what do I do
A. Look in the DOC file where it tells how many are available and how many are used. Now that you know this, you can:

1. Adjust which pin you are using to get more available product terms (this is part dependent).
2. Change the minimization options.
3. Change your logic so it uses less product terms (definitely the most difficult).

Q. When I compile I get an error similar to: "Unable to open RUNFIT.$$$".
A. This generally indicates that CUPL has run out of memory (used it up) so you need to exit and restart CUPL.

Created by Brian Warneke. Last edited 6/16/98