Using Source Constraints in Lattice Devices with ispLEVER™ Software

May 2002 **Application Note AN8068**

Introduction

Constraining a design is becoming more important throughout the whole design process because new Lattice features such as PLL and sysIO™ are constrained in the source design. Although traditional constraints and sysIO can be constrained in the Constraint Editor, many designers still prefer to constrain their design in the source level. Specifically the PLL functions can be implemented using source constraints only.

Lattice implemented a unified constraining scheme in ispLEVER where the user can constrain a design using one constraint syntax over the device families. Once a design is constrained using the correct syntax, the constraints are transferred to the Lattice constraint file called the LCT file. This file contains the constraints from the source file and the Constraint Editor. Once the constraints are imported to the LCT file, the Project Navigator allows the user to process the design with the assigned constraints. Even though ispLEVER provides context-sensitive processing and a unified design flow, the user has to use appropriate attributes for the target device since some constraints are hardware architecture-dependent. Therefore, using constraints in source designs is under the user's responsibility.

The following traditional constraints can be found in ispDesignEXPERT™, Lattice's software for MACH® and ispLSI® devices. The syntax for these constraints in ispLEVER may vary from ispDesignEXPERT:

- Pin/Node Assignment
- Group Assignment
- Node Preservation
- Resource Reservation
- Slew Rate Assignment
- Pull Assignment
- Open Drain Assignment

ispLEVER includes these additional constraints:

- I/O Type Configuration
- PLL Configuration

This application note presents the syntax and several examples of each constraint. The complete set of example designs is located in the Constraints folder under the Examples directory.

Importing Constraints from Source Files

ispLEVER lets the user control the manner of importing source constraints from the design file. The three methods include Auto Import, Always Import, and Do Not Import. These options are under the Tools > Import Source Constraint Option path in the Project Navigator.

Auto Import Source Constraints

By default, this option is checked. When this option is enabled, the software displays a confirmation dialog prior to importing the constraints from the source file. This dialog box appears every time the user changes the source file and then processes it to compile or to fit. If the user selects "Yes" when prompted, constraints from the source files are written into the project constraint file. It is important to notice that the existing constraints in the constraint file are overwritten by the new constraints. If no constraints in the source file are present or the constraint syntax is not legal, then the confirmation dialog box does not appear and no constraints will be written into the file.

Always Import Source Constraints

When the user selects this option, source constraints are imported automatically and current constraints are overwritten during compiling or fitting. No warning message appears when this option is enabled. Therefore, it is suggested to use this option when the user knows the proper source constraint syntax and does not want to see the confirmation dialog box.

Do Not Import Source Constraints

When this option is enabled, source constraints are not imported from the source file and current constraints are not overwritten. The user can use the Constraint Editor to add constraints also.

Considerations for Source Constraints

Using Lattice Library for ABEL Source Constraints

To support various constraint requirements, Lattice prepared a dedicated macro library called *lattice*. This library must be declared first before the constraints are assigned as in the following example.

```
module      ablconst
"library declaration
library     lattice';

"pin declaration
in0, in1    pin;
out1        pin istype 'com';

"source constraints
LAT_PIN(in0, 3);
LAT_PIN(out1, 4);

equations
...
...
end
```

Using Lattice Library for VHDL PLL Constraints

ispLEVER requires the PLL circuits to be declared and configured in the source file. VHDL reads the PLL modules specified in the source code as black boxes. The *lattice.vhd* library supports those black boxes and Lattice's primitives in the source code. The following library declaration should be included in the VHDL source file to use the PLL functions or other Lattice primitive functions. Then Project Navigator can recognize them as black-box functions or Lattice primitive cells.

```
library ieee;
use ieee.std_logic_1164.all;

-- Library declaration
library lattice;
use lattice.components.all;
 ...
 ...
architecture behave of smppll is

component spll
            port  ( clk_in   : in  std_logic;
            clk_out  : out std_logic);
```

```
end component;
 ...
 ...
inst1: SPLL
port map (  clk_in   =>    clk,
            clk_out  => pllclk);
 ...
 ...
```

Using Proper Libraries for Schematic Constraints

Constraints in schematic designs can be assign in schematic symbols. Since multiple symbol libraries are available, the user has to choose the proper library and component symbols to make them constrained as desired. For example, pin assignments can be made on I/O pads in the *IOPADS.LIB* library. To use PLL functions, the component in *PLL.LIB* should be selected.

Important information When Using Synplicity and Exemplar in Verilog

Since Verilog does not support attribute features like VHDL, which has constraints assigned in the comments. Synplicity allows source constraints to be made in the pin declaration section while Exemplar, in the pins declaration section and entities. Therefore, the constraint syntax in Synplicity has to be within the same statement as the pin declaration. In contrast, Exemplar allows the constraints to be located as separate comment lines.

It is important to notice that the Synplicity requires a semicolon at the end of each statement while Exemplar does not. The following are the syntax and examples:

Syntax

Exemplar

```
// exemplar attribute Pin/EntityName ConstraintName value
```

Synplicity

```
PinType PinName   /* synthesis ConstraintName= "value" */;
```

Examples

Exemplar

```
Input inA3;
//exemplar  attribute  inA3 LOC PA3
```

Synplicity

```
input inA3        /* synthesis LOC= "PA3" */;
```

Pin/Node Assignment

Pre-assignment is necessary when the pin-out of a design has been established before the creation of the design or during iterations while fitting. With this constraint, designs fit with resources assigned to the same physical locations. For the best results, the user should let the fitter make initial assignments to pins and nodes. The decision of pin/node assignment should be made very carefully because the pre-assigned pins and nodes may restrict the fitter from efficiently placing pins and nodes prohibiting the fitter from using algorithms to optimize the placement of the design signals.

ABEL Pin Assignment using ABEL Syntax

Syntax

```
PinNameList   pin PinNumberList;
```

Note: The number of pin names declared in the PinNameList must be the same as the number of pin numbers specified in the PinNumberList.

Example

```
inA3, inB4, inC5  pin A3, B4, C5;
outB3, outC4      pin B3, C4 istype 'com';
outF8, outA2      pin istype 'com';
clk               pin;
nodeB12           node istype 'reg';
```

ABEL Pin/Node Assignment using Lattice Macro

Syntax

General form of pin assignment:

```
LAT_LOC(PinName, Pin#, Seg#, GLB#, MC#);
```

To a pin:

```
LAT_PIN(PinName, Pin#);
```

To a pin in a GLB:

```
LAT_PIN_GLB(PinName, GLB#);
```

To a pin in a Segment/GLB:

```
LAT_PIN_SEGGLB(PinName, Seg#, GLB#);
```

Node assignment to a macrocell:

```
LAT_NODE_GLB(SigName, GLB#, MC#);
```

Node assignment to a macrocell:

```
LAT_NODE_GLB(SigName, GLB#, MC#);
```

Note: LAT_LOC can be used to backannotate a successful placement result to the source. LAT_PIN_GLB is for devices without segments only. LAT_PIN_SEGGLB is for devices with segments only.

Example

```
LAT_LOC(out0, pin, F8, -, -, -);
LAT_PIN(out1, A2);
LAT_NODE(node1, 1, B, 12);
```

VHDL Pin Assignment

Syntax

```
attribute LOC : string;
attribute LOC of SigName: signal is "P[Pin#]";
```

Example

```
attribute LOC : string;
attribute LOC of out0: signal is "PA3";
attribute LOC of out1: signal is "PF8 PA2 PB3";
```

Verilog Pin Assignment

Syntax

Exemplar

```
//exemplar attribute PinName LOC P[Pin#]
```

Synplicity

```
PinType PinName /* synthesis LOC= "P[Pin#]" */;
```

Note: For Synplicity, PinType can be set to input/output.

Examples

Exemplar

```
input        inA3;//exemplar attribute inA3 LOC PA3
output [0:2] sout;//exemplar attribute sout LOC PF8PA2PB3
```

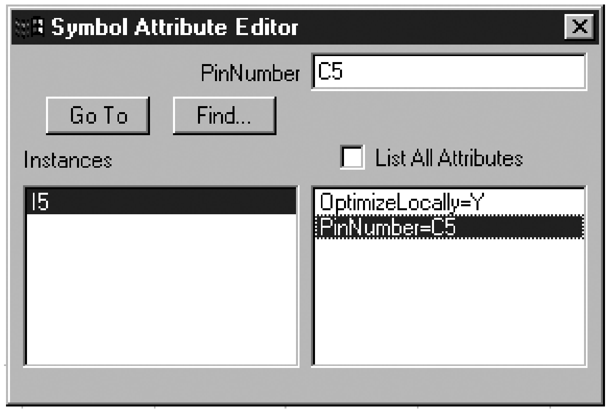
Synplicity

```
input        inA3/* synthesis LOC= "PA3" */;
output [0:2] sout/* synthesis LOC= "PF8PA2PB3" */;
```

Schematic Pin Assignment

To assign pins in a schematic, the user must utilize a generic symbol from the *IOPADS* library and follow these steps:

Figure 1. Schematic Pin Assignment Illustration of Step 3



PinNumber C5

I5

OptimizeLocally=Y
PinNumber=C5

1. Select Edit > Attribute > Symbol Attribute.
2. Click on the I/O pad to be assigned.
3. Select the PinNumber attribute and enter the desired pin number.
4. Close this dialog or click on the next I/O pad. Then the pin number appears in the selected symbol.

Group Assignment

Using the grouping constraint allows the fitter to locate signals inside a logic block specified in the source. If grouping is made without any physical block assignment, the fitter can control the location of the signals efficiently inside the target device. This approach of assigning groups is recommended over assigning signals to specific pins/macrocells.

ABEL Group Assignment

Syntax

General Form:

```
LAT_GROUP(GrpName, Seg#, GLB#, SigList);
```

To a GLB:

```
LAT_GROUP_GLB(GrpName, GLB#, SigList);
```

Without Physical Block Assignment:

```
LAT_GROUP_LOGICAL(GrpName, SigList);
```

Note 1: LAT_GROUP is a superset of other grouping macros. LAT_GROUP_GLB is for devices without segments only. LAT_GROUP_GLB(GrpName, GLB#, SigList) has the same effect as LAT_GROUP(GrpName, -, GLB#, SigList) and SHOULD be used for the device architecture without Segments. LAT_GROUP_LOGICAL is for grouping without any physical block assignment.

Note 2: You can put '*' and '-' that mean 'any block' and 'not applicable' respectively. *,* and -, - are invalid combinations for Seg# and GLB# respectively.

Example

```
//GrpA will be placed in Seg 2, GLB C
LAT_GROUP(GrpA, 2, C, inA3: inA2: inA1: inA0: outA3: outA2: outA1: outA0);

//GrpB will be placed in Seg 1, any GLB
LAT_GROUP(GrpB, 1, *, inB3: inB2: inB1: inB0: outB3: outB2: outB1: outB0);

//GrpC will be placed in any Seg, Any GLB
LAT_GROUP_LOGICAL(GrpC, inC3: inC2: inC1: inC0: outC3: outC2: outC1: outC0);
```

VHDL Group Assignment

Syntax

Exemplar

```
attribute GROUPING : string;
attribute of GROUPING of EntityName: entity is "GrpName = Seg#, GLB#, SigList;"
```

Note: For Exemplar, one GROUPING attribute assignment per entity is allowed. A std_logic_vector signal must be converted to the final form for the fitter handling (ex. inA(3) -> inA_3_).

Example

Exemplar

```
attribute  GROUPING : string;
--  GrpA will be placed in Seg 0, GLB A
attribute GROUPING of grouping : entity is "GrpA = 0, A, Den: inA_3_:
inA_2_:inA_1_:inA_0_:outA_3_:outA_2_:outA_1_:outA_0_";

-- GrpA will be placed in Seg 1, any GLB
attribute GROUPING of grouping : entity is "GrpA = 1, *, Den: inA_3_: inA_2_:
inA_1_: inA_0_: outA_3_: outA_2_: outA_1_: outA_0_";
```

```
      -- GrpA will be placed in any Seg, Any GLB
      attribute GROUPING of vhdgroup : entity is "GrpA = *, *, Den: inA_3_: inA_2_:
      inA_1_: inA_0_: outA_3_: outA_2_: outA_1_: outA_0_";
```

Node Preservation

As a design goes through the process flow, the pre-fitter or optimizer normally optimizes each of the sources and the linked design to minimize the logic needed. The software usually conducts node collapsing to improve the design's speed, i.e. fMAX. However, logic can be manually partitioned to achieve the speed and/or area goal. The technique is called node preserving. This is similar to the opposite of node collapsing, which keeps a specific combinatorial node that takes a physical location inside the device. Implementing the preservation constraint ensures that no collapsing will be performed to remove the nodes. These nodes retain the same name and functionality also. When preserving nodes, the output of a commonly used combinatorial node can be shared with many other logic equations. This saves the amount of logic resource in the device. Sometimes this technique is used to intentionally generate the timing delay on a signal path. On the contrary, node preservation can cause unneeded nodes to be kept and may cause more logic to be implemented.

ABEL Node Preservation

Syntax

```
      NodeName(s)  node istype 'com, keep';  OR
      NodeName(s)  node istype 'keep';
```

Example

```
      nodeA, nodeB      node istype 'keep';
```

VHDL Node Preservation

Syntax

Exemplar

```
      attribute PRESERVE_SIGNAL : boolean;
      attribute PRESERVE_SIGNAL of NodeName(s): signal is TRUE;
      attribute OPT : string;
      attribute OPT of NodeName(s): signal is "KEEP";
```

Synplicity

```
      attribute syn_keep : integer;
      attribute syn_keep of NodeName(s): signal is 1;
      attribute OPT : string;
      attribute OPT of NodeName(s): signal is "KEEP";
```

Examples

Exemplar

```
      attribute PRESERVE_SIGNAL : boolean;
      attribute PRESERVE_SIGNAL of nodeA, nodeB: signal is TRUE;

      attribute OPT : string;
      attribute OPT of nodeA, nodeB: signal is "KEEP";
```

Synplicity

```
attribute syn_keep : integer;
attribute syn_keep of nodeA, nodeB: signal is 1;

attribute OPT : string;
attribute OPT of nodeA, nodeB: signal is "KEEP";
```

Verilog Node Preservation

Syntax

Exemplar

```
//exemplar attribute NodeName PRESERVE_SIGNAL TRUE
//exemplar attribute NodeName OPT KEEP
```

Synplicity

```
wire NodeName/* synthesis syn_keep= 1 OPT= "KEEP" */;
```

Examples

Exemplar

```
//exemplar attribute nodeA PRESERVE_SIGNAL TRUE
//exemplar attribute nodeA OPT KEEP
```

or

```
/**** The following comment form also works ****/
/*exemplar attribute nodeB PRESERVE_SIGNAL TRUE
exemplar attribute nodeB OPT KEEP*/
```

Synplicity

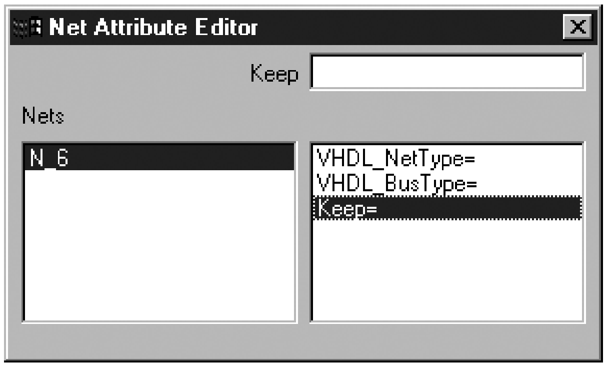
```
wire    nodeA /* synthesis syn_keep= 1 OPT= "KEEP" */;
wire    nodeB /* synthesis syn_keep= 1 OPT= "KEEP" */;
```

Schematic Node Preservation

To assign pins in a schematic, the user must use the Net Attribute dialog and follow these steps:

1. Select Edit > Attribute > Net Attribute.
2. Click on the desired net to be preserved.
3. Select the Keep attribute and enter the Yes or Y.
4. Close this dialog or click on the next net.

Figure 2. Schematic Node Preservation Illustration or Step 3



Resource Reservation

ispLEVER allows the reservation of logic resources in the devices for future use. This feature ensures that the optimizers and fitter do not use those reserved resources. Checking the Fitter Report and Post_Fit Pinouts report ensures if the resources were saved. ispLEVER allows resource reservation only in ABEL.

ABEL Resource Reservation

Syntax

General Form

```
LAT_RESERVE(Type, Number, PinState);
```

Segment Reservation

```
LAT_RESERVE_SEGMENT(Number, PinState);
```

GLB Reservation

```
LAT_RESERVE_GLB(Number, PinState);
```

Note 1: LAT_RESERVE is a superset of other preserving macros. LAT_RESERVE_SEGMENT is for devices with segments only. LAT_RESERVE_GLB is for devices without segments only.

Note 2: Resource type can be set to Pin/GLB/Segment. PinState can be set to input/out_z/out_low/out_high.

Example

```
LAT_RESERVE_GLB(C, out_z);
LAT_RESERVE(Pin, A3, out_low);
```

I/O Type Configuration

The I/O Type feature is included in some Lattice devices. Setting "IO_TYPES" configures an I/O pin to a specific I/O standard. For more information on the valid "IO_TYPES" settings for a device, please refer to the device data sheet and technical note number TN1000, *sysIO Design and Usage Guidelines*.

Below is a list of valid I/O types:

PCI, PCI-X, AGP_1X, SSTL3_I, SSTL3_II, SSTL2_I, SSTL2_II, HSTL_I, HSTL_III, CTT3, CTT2, GTLPLUS, LVTTL, LVCMOS33, LVCMOS33_OD, LVCMOS25, LVCMOS25_OD, LVCMOS18, LVCMOS18_OD, LVDS, LVPECL_S, LVPECL_D, LVCMOS33_OD, LVCMOS25_OD, LVCMOS18_OD

ABEL I/O Type Configuration

Syntax

```
LAT_IOTYPES(PinName, Type, DriveCurrent);
```

Note: DriveCurrent can be set to 20, 16, 12, 8, 5, 4 or NONE(or -). The NONE or dash (-) means "not applicable".

Example

```
LAT_IOTYPES(out1, SSTL3_I, NONE);
LAT_IOTYPES(out2, LVCMOS33, 20);

out1,out2      pin istype 'com';
load           pin istype 'com';
```

VHDL I/O Type Configuration

Syntax

```
attribute IO_TYPES : string;

attribute IO_TYPES of PinName: signal is "Type, DriveCurrent";
```

Example

```
--*** Attribute declaration
attribute IO_TYPES : string;

--*** IO types for I/O pins ***
attribute IO_TYPES of md: signal is "PCI, NONE";
attribute IO_TYPES of portA: signal is "PCI, NONE";
attribute IO_TYPES of portB: signal is "LVCMOS33, 20";      -- 20mA
attribute IO_TYPES of portC: signal is "CTT33, NONE";
attribute IO_TYPES of portD: signal is "LVCMOS25_OD, 8";    -- 8mA
```

Verilog I/O Type Configuration

Syntax

Exemplar

```
//exemplar attribute PinName  IO_TYPES Type, DriveCurrent
```

Synplicity

```
PinType PinName/* synthesis  IO_TYPES= "Type, DriveCurrent" */;
```

Note: In Verilog, the PinName for IO Type must be the same as the name declared. The individual set expression such as portA[0] is not allowed as a valid name. For PinType, pin should be either input or output.

Examples

Exemplar

```
/*** IO types for I/O pins ***/
//exemplar attribute md IO_TYPES  PCI, NONE
//exemplar attribute portA IO_TYPES PCI, -
//exemplar attribute portB IO_TYPES LVCMOS33, 20 - 20mA
//exemplar attribute portC IO_TYPES CTT33, NONE
//exemplar attribute portD IO_TYPES LVCMOS25_OD, 8 - 8mA
```

Synplicity

```
/*** IO types for I/O pins ***/
input     [1:0] md   /* synthesis  IO_TYPES=  "PCI, NONE" */;
input     [4:0] Din;
output[4:0] portA   /* synthesis IO_TYPES= "PCI, -" */;
output[4:0] portB   /* synthesis IO_TYPES= "LVCMOS33, 20" */;     //20mA
output[4:0] portC   /* synthesis IO_TYPES= "CTT33, NONE" */;
output[4:0] portD   /* synthesis IO_TYPES= "LVCMOS25_OD, 8" * /;  //8mA
```

Slew Rate Assignment

Slew rates can be assigned to each I/O pin; however, only output pins accept the slew rate control. The slew type can be set to SLOW/FAST. Each output can be configured either to the low noise transition or to the high-speed transition. For high-speed designs with long, unterminated traces, the slow slew rate will introduce fewer reflections, less noise, and keep ground bounce to a minimum. For designs with short traces or well-terminated lines, the fast slew rate can be used to achieve the higher speed. The slew rate is independent of power.

ABEL Slew Rate Assignment

Syntax

```
LAT_SLEW(Type, PinList);
```

Slew Rate Set to Default

```
LAT_SLEW_DEFAULT(Type);
```

Example

```
LAT_SLEW (SLOW, slewS3: slewS2: slewS1: slewS0);
LAT_SLEW (FAST, slewF);
```

VHDL Slew Rate Assignment

Syntax

```
attribute SLEW : string;
attribute SLEW of SigName: signal is "Type";
```

Example

```
attribute SLEW : string;
attribute SLEW of slewS: signal is "SLOW";
attribute SLEW of slewF: signal is "FAST";
```

Verilog Slew Rate Assignment

Syntax

Exemplar

```
//exemplar attribute  NodeName SLEW Type
```

Synplicity

```
output PinName /* synthesis  SLEW= "Type" */;
```

Examples

Exemplar

```
output slewS; //exemplar attribute slewS SLEW SLOW
output slewF; //exemplar attribute slewF SLEW FAST
```

Synplicity

```
output slewS /* synthesis SLEW= "SLOW" */;
output slewF /* synthesis SLEW= "FAST" */;
```

Pull Assignment

The PULL attribute affects the I/O pins only. The I/O pins within our devices include internal circuitry to allow pin functions, which include pull-up resistors, pull-down resistors, and Bus-Friendly(tm) configurations.

The default value of PULL is "UP". If the signal is set to PULL UP, the pin utilizes a weak pull-up resistor to pull signals high when not driven. If the signal is set to PULL HOLD, the pin is held to the last state (Bus-FriendlyTM) after the source has been removed. If the signal is set PULL OFF, the pin remains floating when not driven. In this case, the pin remains floating when not driven. If the signal is set to PULL DOWN, the pin utilizes a weak pull-down resister to pull signals low when not driven. PULL DOWN is not available on all devices though. When the output pin is set to HOLD, then it is configured as Bus-Friendly. When using devices that support the I/O Types feature, not all pin functions are permitted with certain I/O standards. For more information, please refer to TN1000, *sysIO Design and Usage Guidelines*.

ABEL Pull Assignment

Syntax

```
LAT_PULL(Type, PinList);
Pull Set to Default
LAT_PULL_DEFAULT(Type);
```

Example

```
LAT_PULL(UP, pullup: inA);
LAT_PULL(DOWN, pulldn: inB);
LAT_PULL(HOLD, pullhd: inC);
LAT_PULL(OFF, pullof);
```

VHDL Pull Assignment

Syntax

```
attribute PULL : string;
attribute PULL of SigName: signal is "Type";
```

Example

```
attribute PULL : string;
attribute PULL of ina: signal is "UP";
attribute PULL of pullup: signal is "UP";
attribute PULL of pulldn: signal is "DOWN";
attribute PULL of pullhd: signal is "HOLD";
attribute PULL of pullof: signal is "OFF";
```

Verilog Pull Assignment

Syntax

Exemplar

```
//exemplar attribute  PinName PULL Type
```

Synplicity

```
PinType PinName /* synthesis PULL= "Type" */;
```

Note: For Synplicity, PinType can be set to input/output.

Examples

Exemplar

```
input  A,B,C;     //exemplar attribute A PULL UP
                  //exemplar attribute B PULL DOWN
                  //exemplar attribute C PULL HOLD
output YUP;       //exemplar attribute YUP PULL UP
output YDN;       //exemplar attribute YDN PULL DOWN
output YHD;       //exemplar attribute YHD PULL HOLD
output YOF;       //exemplar attribute YOF PULL OFF
```

Synplicity

```
Input  A        /* synthesis PULL= "UP" */;
input  B        /* synthesis PULL= "DOWN" */;
input  C        /* synthesis PULL= "HOLD" */;
output YUP      /* synthesis PULL= "UP" */;
output YDN      /* synthesis PULL= "DOWN" */;
output YHD      /* synthesis PULL= "HOLD" */;
output YOF      /* synthesis PULL= "OFF" */;
```

Open Drain Assignment

Open drain allows flexible bus interface capability and implementation of wired-or or bus arbitration logic. If the user implements the open drain output on an output pin, the pin drives only the specified V_{OL}. The V_{OH} level on the open drain output depends on the extended loading and pull-up. The user can assign open drain to each I/O pin using the IO_TYPES syntax in ispLEVER. Only I/O types of LVCMOS accept the open drain control. Each type of LVCMOS outputs has a different current driving capability. In ABEL, Lattice defined a dedicated open drain attribute LAT_OPENDRAIN. Since this has the same effect as the IO_TYPES attribute, it should not be used with LAT_IOTYPES on the same pin. Devices without multi-driving capability must be set to NONE(-) for DriveCurrent.

ABEL Open Drain Assignment

The user can follow either syntax to implement Open Drain. The alternative syntax is LAT_OPENDRAINxx.

Syntax

Type LVCMOS33

```
LAT_IOTYPES(PinName, LVCMOS33_OD, DriveCurrent);
```

Type LVCMOS25

```
LAT_IOTYPES(PinName, LVCMOS25_OD, DriveCurrent);
```

Type LVCMOS18

```
LAT_IOTYPES(PinName, LVCMOS18_OD, DriveCurrent);
```

Alternative ABEL Syntax

```
LAT_OPENDRAIN33(PinName, DriveCurrent);
```

or

```
LAT_OPENDRAIN25(PinName, DriveCurrent);
```

or

```
LAT_OPENDRAIN18(PinName, DriveCurrent);
```

Note: The PinName is the Output pin name. DriveCurrent can be set to 20, 16, 12, 8, 5, 4 or NONE(or -).

Example

```
LAT_IOTYPES(outOD3, LVCMOS33_OD, 16);"3.3V Opendrain, 16 mA
LAT_IOTYPES(outOD2, LVCMOS25_OD, 8); "2.5V Opendrain, 8 mA
LAT_IOTYPES(outOD1, LVCMOS18_OD, 4); "1.8V Opendrain, 4 mA
LAT_IOTYPES(outOD0, LVCMOS18_OD, 8); "1.8V Opendrain, 8 mA
```

VHDL Open Drain Assignment

Syntax

```
attribute IO_TYPES : string;
attribute IO_TYPES of PinName: signal is "LVCMOS33_OD, DriveCurrent";
```

or

```
attribute IO_TYPES of PinName: signal is "LVCMOS25_OD, DriveCurrent";
```

or

```
attribute IO_TYPES of PinName: signal is "LVCMOS18_OD, DriveCurrent";
```

Example

```
--*** Attribute declaration
attribute IO_TYPES : string;

--*** Open Drain setting for output pins ***
attribute IO_TYPES of outA_OD: signal is "LVCMOS33_OD, 20";
attribute IO_TYPES of outB_OD: signal is "LVCMOS25_OD, 16";
attribute IO_TYPES of outC_OD: signal is "LVCMOS18_OD, 8";
```

Verilog Open Drain Assignment

Syntax

Exemplar

```
//exemplar attribute PinName IO_TYPES LVCMOS33_OD, DriveCurrent;
```

or

```
//exemplar attribute PinName IO_TYPES LVCMOS25_OD, DriveCurrent;
```

or

```
//exemplar attribute PinName IO_TYPES LVCMOS18_OD, DriveCurrent;
```

Synplicity

```
output PinName  /* synthesis IO_TYPES= "LVCMOS33_OD, DriveCurrent" */;
output PinName   /* synthesis IO_TYPES= "LVCMOS25_OD, DriveCurrent" */;
output PinName   /* synthesis IO_TYPES= "LVCMOS18_OD, DriveCurrent" */;
```

Examples

Exemplar

```
//exemplar attribute outA_OD IO_TYPES  LVCMOS33_OD, 20;
```

or

```
//exemplar attribute outB_OD IO_TYPES  LVCMOS25_OD, 16;
```

or

```
//exemplar attribute outC_OD IO_TYPES  LVCMOS18_OD, 8;
```

Synplicity

```
output [3:0] outA_OD /* synthesis IO_TYPES= "LVCMOS33_OD, 20" */;
output [3:0] outB_OD /* synthesis IO_TYPES= "LVCMOS25_OD, 16" */;
output [3:0] outC_OD /* synthesis IO_TYPES= "LVCMOS18_OD, 8" */;
```

PLL Configuration

For the ispLSI devices that have sysIO PLL circuits, the user needs to assign values for these settings: "IN_FREQ", "MULT", "DIV", "POST", "DELAY" and "SECDIV."

The general syntax is

Simple PLL

```
SPLL(CLK_IN, CLK_OUT);
```

Standard PLL

```
STDPLL(CLK_IN, PLL_LOCK, CLK_OUT);
```

Extended PLL

```
STDPLLX(CLK_IN, PLL_FBK, PLL_RST, PLL_LOCK, CLK_OUT, SEC_OUT);
```

Where

CLK_IN:	PLL clock input
CLK_OUT:	PLL clock output
SEC_OUT:	PLL secondary clock output
PLL_LOCK:	PLL lock output
PLL_FBK:	PLL feedback input
PLL_RST:	PLL reset input

Other signals associated with the PLL are

clk_in:	Input clock signal name
in_freq:	Input frequency of the PLL input signal
clk_out:	Primary PLL clock out signal name
sec_out:	Secondary PLL clock out signal name
clk_out_to_pin:	PLL output clock routed out to CLK_OUT pin option, ON or OFF
secdiv:	Secondary clock divider, 2,4,8,16, or 32
mult:	Clock multiplier, any integer from 1 to32
div:	Clock divider, any integer from 1 to 32
post:	Post-scalar divider, 1,2,4,8,16,or 32
pll_rst:	PLL reset signal name
pll_fbk:	PLL feedback signal name
pll_lock:	PLL lock signal name
pll_dly:	PLL delay parameter, -3.5 to 3.5 in 0.5 increments

The "IN_FREQ" attribute assigns a frequency to the input clock of the PLL. This attribute can be set to any numeric value up to 4-digit decimal in MHz within the specifications (e.g. 125.2538).

The "MULT" attribute assigns a multiply factor to the PLL. By setting the "MULT" attribute, the output clock frequency is the input clock frequency multiplied by this factor. The "MULT" attribute can be set to any integer between 1 and 32.

The "DIV" attribute assigns a divide factor to the PLL. By setting the "DIV" attribute, the output clock frequency is the input clock frequency divided by this factor. The "DIV" attribute can be set to any integer between 1 and 32.

The "POST" attribute assigns a post-scalar divide value to the PLL. Only advanced users should use this attribute.

The "PLL_DLY" attribute assigns a delay value to the output clock. Effectively, this phase shifts the clocks. The "PLL_DLY" attribute can be set to any value between -3.5(ns) and 3.5(ns) in 0.5ns increments.

The "SECDIV" attribute assigns a divide value to the Secondary Clock. The Secondary Clock is a second clock output with the frequency equal to the output clock divided by the "SECDIV" value. The "SECDIV" attribute can be set to 2, 4, 8, 16, and 32.

For more information about the PLL, please refer to technical note number TN1003, *sysCLOCK PLL Design and Usage Guidelines* for the particular device.

ABEL PLL Configuation

Configuration Syntax

The PLL configuration in ABEL is required in the Declarations section.

Simple PLL

```
XLAT_STDPLLX(clk_in, in_freq, clk_out);
```

Standard PLL

```
XLAT_STDPLL(clk_in, in_freq, clk_out, clk_out_to_pin, mult, div, post,
pll_lock, pll_dly);
```

Extended PLL

```
XLAT_STDPLLX(clk_in, in_freq, clk_out, sec_out, clk_out_to_pin, secdiv, mult,
div, post, pll_rst, pll_fbk, pll_lock, pll_dly);
```

Note: LAT_PLL is a superset of LAT_SPLL, LAT_STDPLL, and LAT_STDPLLX.

Instantiation Syntax

The PLL instantiation in ABEL is required in the Equations section.

Simple PLL

```
pll_name SPLL(clk_in, clk_out);
```

Standard PLL

```
pll_name STDPLL(clk_in, pll_lock, clk_out);
```

Extended PLL

```
pll_name STDPLLX(clk_in, pll_fbk, pll_rst, pll_lock, clk_out, sec_out);
```

Examples:

Simple PLL

```
"PLL Configuration in declaration
XLAT_SPLL(clk, 133.0, pllclk, 0.0);

"PLL Instantiation in equations
pll_1 SPLL(pllclk, clk);
```

Standard PLL

```
"PLL Configuration
XLAT_STDPLL(clk, 80.0000, ppclk, spclk, off, 2, 6, 7, 1, lock, 2.5);

"PLL Instantiation
pll_std STDPLL(lock, spclk, ppclk, clk);
```

Extended PLL

```
"PLL Configuration
XLAT_STDPLLX(clk, 100.0000, ppclk, spclk, OFF, 2, 7, 10, 1, pprst, pfbk, sprst,
lock, 2.5);

"PLL Instantiation
pll_std STDPLLX(lock, spclk, ppclk, clk, pfbk, pprst, sprst);
```

VHDL PLL Configuration

To use PLL functions in VHDL, the user needs to declare the Lattice library, PLL component, and PLL parameters. Then the PLL module must be instantiated in the architecture section. The output signals from the PLL module can be used in the design. For more examples, please refer to the Examples\Constraints\PLL\VHDL directory in the ispLEVER software.

PLL Usage Steps

The following steps are required to use PLL functions in VHDL:

Step 1. Lattice Library Declaration

```
library lattice;
use lattice.components.all;
```

Step 2. PLL Component Declaration with Generics
This step is required for simulation and Synplify synthesis.

```
component stdpllx
  generic( in_freq  : string;
         mult     : string;
         iv       : string;
         post     : string;
         pll_dly  : string;
         secdiv   : string);
  port(  clk_in   : in  std_logic;
         pll_fbk  : in  std_logic;
         PLL_RST  : in  std_logic;
         pll_lock : out std_logic;
         sec_out  : out std_logic;
         clk_out  : out std_logic);
```

Step 3. Parameter Passing Through Attributes for the Fitter.
This step is required for Exemplar synthesis.

```
attribute in_freq              : string;
attribute mult                 : string;
attribute div                  : string;
attribute post                 : string;
attribute pll_dly              : string;
attribute secdiv               : string;
attribute clk_out_to_pin       : string;
attribute in_freq of i1        : label is "100.0000";
attribute mult    of i1        : label is "8";
attribute div     of i1        : label is "5";
attribute post    of i1        : label is "1";
attribute pll_dly of i1        : label is "2.0";
attribute secdiv  of i1        : label is "2";
attribute clk_out_to_pin of i1: label is "OFF";
```

Step 4.PLL Hardcore Instantiation with PLL Parameter Mapping

```
I1:  STDPLLX
     generic map( in_freq  => "100.0000",
            mult           => "8",
            div            => "5",
            post           => "1",
            pll_dly        => "2.0",
            secdiv         => "2")
     port map   ( clk_in   => clk,
            pll_fbk        => pllfbk,
            pll_rst        => pllrst,
            pll_lock       => lock,
            clk_out        => ppclk,
            sec_out        => spclk);
```

Step 5. Use PLL outputs in the VHDL architecture

Verilog PLL Configuration

The PLL module must be instantiated in the architecture section in Verilog. Output signals from the PLL module can be used in the design. For more examples, please refer to the Examples\Constraints\PLL directory in the ispLEVER software.

PLL Usage Steps

The following steps are required for simple PLL functions when using Exemplar in Verilog.

Step 1. PLL hardcore block declaration

```
module stdpllx(clk_in, pll_fbk, pll_rst, pll_lock, sec_out, clk_out);
parameter in_freq = "1";
parameter mult = "1";
parameter div = "1";
parameter post = "1";
parameter pll_dly = "1";
parameter secdiv = "1";
input     clk_in;
input     pll_fbk;
input     pll_rst;
output    clk_out;
output    pll_lock;
output    sec_out;
endmodule
```

Step 2. PLL parameter definition

```
defparam    I1.in_freq = "100.0000",
            I1.mult    = "8",
            I1.div     = "5",
            I1.post    = "1",
            I1.pll_dly = "2.0",
            I1.secdiv  = "2";
```

Step 3.PLL block instantiation

```
stdpllx I1 (.clk_in(clk),
            .pll_fbk(pllfbk),
            .pll_rst(pllrst),
            .clk_out(ppclk),
            .pll_lock(lock),
            .sec_out(spclk));
```

Step 4. PLL parameters configuration using Exemplar attribute syntax

```
// exemplar attribute I1 in_freq 100.0000
// exemplar attribute I1 mult    8
// exemplar attribute I1 div     5
// exemplar attribute I1 post    1
// exemplar attribute I1 pll_dly 2.0
// exemplar attribute I1 secdiv  2
// exemplar attribute I1 clk_out_to_pin OFF
```

Step 5. Use PLL output in the design

Conclusion

ispLEVER software supports unified source constraint syntax. Constraints work over device families as long as the assigned constraints are device architecture independent such as PLL. Due to the nature of HDL language coverage, most constraints in Verilog and some constraints in the VHDL are different depending on the selected HDL synthesizer. The examples in this application note show the constraint only. The complete set of design examples is available under the Examples\Constraints directory in the ispLEVER software.

Appendix: Source Constraint Syntax Summary

ABEL

Constraint	Syntax
Pin/Node Assignment	
Pin assignment using ABEL syntax	*PinNameList* pin *PinNumberList;*
Pin/node assignment using Lattice macros	LAT_LOC*(PinName, Pin#, Seg#, GLB#, MC#);*
To a pin	LAT_PIN*(PinName, Pin#);*
To a pin in a GLB	LAT_PIN_GLB*(PinName, GLB#);*
To a pin in a segment/GLB	LAT_PIN_SEGGLB*(PinName, Seg#, GLB#);*
Node assignment to a macrocell	LAT_NODE*(SigName, Seg#, GLB#, MC#);*
Node assignment to a macrocell	LAT_NODE_GLB*(SigName, GLB#, MC#);*
Group Assignment	LAT_GROUP*(GrpName, Seg#, GLB#, SigList);*
To a GLB	LAT_GROUP_GLB*(GrpName, GLB#, SigList);*
Without physical block assignment	LAT_GROUP_LOGICAL*(GrpName, SigList);*
Node Preservation	*NodeName(s)* node istype 'keep';
Resource Reservation[1]	LAT_RESERVE*(Type, Number, PinState);*
Segment reservation	LAT_RESERVE_SEGMENT*(Number, PinState);*
GLB reservation	LAT_RESERVE_GLB*(Number, PinState);*
I/O Type Configuration[2]	LAT_IOTYPES*(PinName, Type, DriveCurrent);*
Slew Rate Assignment	LAT_SLEW*(Type, PinList);*
Slew rate set to default	LAT_SLEW_DEFAULT*(Type);*
Pull Assignment	LAT_PULL*(Type, PinList);*
Pull set to default	LAT_PULL_DEFAULT*(Type);*
Open Drain Assignment[2, 3]	LAT_IOTYPES*(PinName, OpenDrainType, DriveCurrent);*
Alternative syntax	LAT_OpenDrain_Type*(PinName, DriveCurrent);*
PLL Configuration	
Simple PLL	XLAT_STDPLLX*(clk_in, in_freq, clk_out);*
Standard PLL	XLAT_STDPLL*(clk_in, in_freq, clk_out, clk_out_to_pin, mult, div, post, pll_lock, pll_dly);*
Extended PLL	XLAT_STDPLLX*(clk_in, in_freq, clk_out, sec_out, clk_out_to_pin, secdiv, mult, div, post, pll_rst, pll_fbk, pll_lock, pll_dly);*
Instantiation Syntax	
Simple PLL	pll_name SPLL*(clk_in, clk_out);*
Standard PLL	pll_name STDPLL*(clk_in, pll_lock, clk_out);*
Extended PLL	pll_name STDPLLX*(clk_in, pll_fbk, pll_rst, pll_lock, clk_out, sec_out);*

1. *Type* can be set to either Pin, GLB or Segment. *PinState* can be set to Input, Out_z, Out_low or Out_high.
2. *DriveCurrent* can be set to 20, 16, 12, 8, 5, 4 or NONE(or -). The NONE or dash (-) means "not applicable".
3. *OpenDrainType* can be set to LVCMOS33_OD/LVCMOS25_OD/LVCMOS18_OD.

VHDL

Constraint	Syntax

Pin Assignment	attribute LOC : string;
	attribute LOC of *SigName*: signal is "P[*Pin#*]";

Group Assignment	attribute GROUPING : string; attribute of GROUPING of *EntityName*: entity is "*GrpName= Seg#, GLB#, SigList;*"

Node Preservation (Exemplar)
 	
Node Preservation (Synplicity)
	attribute PRESERVE_SIGNAL : boolean;
	attribute PRESERVE_SIGNAL of *NodeName(s)*: signal is TRUE;
	attribute syn_keep : integer;
	attribute syn_keep of *NodeName(s)*: signal is 1;
	attribute OPT : string;
	attribute OPT of *NodeName(s)*: signal is "KEEP";

I/O Type Configuration[1]	attribute IO_TYPES : string;
	attribute IO_TYPES of PinName: signal is "*Type, DriveCurrent*";

Slew Rate Assignment	attribute SLEW : string;
	attribute SLEW of *SigName*: signal is "*Type*";

Pull Assignment	attribute PULL : string;
	attribute PULL of *SigName*: signal is "*Type*";

Open Drain Assignment[1, 2]	attribute IO_TYPES : string;
	attribute IO_TYPES of *PinName*: signal is "*OpenDrain_Type, DriveCurrent*";

PLL Configuration	See example in application note

1. *DriveCurrent* can be set to 20, 16, 12, 8, 5, 4 or NONE(or -). The NONE or dash (-) means "not applicable".
2. *OpenDrain_Type* can be set to LVCMOS33_OD/LVCMOS25_OD/LVCMOS18_OD.

Verilog

Constraint		Syntax
Pin Assignment	Exemplar[2]	//exemplar attribute *PinName* LOC P[*Pin#*]
	Synplicity[1, 2]	*PinType PinName* /* synthesis LOC= "P[*Pin#*]" */;
Node Preservation	Exemplar	//exemplar attribute *NodeName* PRESERVE_SIGNAL TRUE
	Synplicity	//exemplar attribute *NodeName* OPT KEEP
I/O Type Configuration	Exemplar[2, 3]	//exemplar attribute *PinName* IO_TYPES *Type, DriveCurrent*
	Synplicity[1, 2, 3]	PinType *PinName* /* synthesis IO_TYPES= "*Type, DriveCurrent*" */;
Slew Rate Configuration	Exemplar	//exemplar attribute *NodeName* SLEW *Type*
	Synplicity[2]	output *PinName* /* synthesis SLEW= "*Type*" */;
Pull Assignment	Exemplar[2]	//exemplar attribute *PinName* PULL *Type*
	Synplicity[1, 2]	*PinType PinName* /* synthesis PULL= "*Type*" */;
Open Drain Assignment	Exemplar[2, 3, 4]	//exemplar attribute *PinName* IO_TYPES *OpenDrain _Type*, DriveCurrent;
	Synplicity[2, 3, 4]	output *PinName* /* synthesis IO_TYPES= *OpenDrain _Type*, *DriveCurrent*" */;
PLL Configuration		See example in application note

1. For Synplicity, *PinType* can be set to input/output.
2. In Verilog, the *PinName* for I/O type must be the same as the name declared. The individual set expression such as portA[0] is not allowed as a valid name.
3. *DriveCurrent* can be set to 20, 16, 12, 8, 5, 4 or NONE(or -). The NONE or dash (-) means "not applicable".
4. *OpenDrain_Type* can be set to LVCMOS33_OD/LVCMOS25_OD/LVCMOS18_OD.