Hints, Tips and Tricks for using XABEL with Xilinx M1.5 Design and Implementation Tools

XAPP109 October 21, 1998 (Version 2.0) Application Note

Summary

This application note summarizes the issues and design techniques specific to the Xilinx ABEL Interface, version M1.5.

Xilinx Family

All

Introduction

Products

XABEL included only in Foundation product

XABEL is included in all variations of the Foundation F1.5 product (Base or Standard, with or without HDL). When used in Foundation, ABEL design entry is supported by the HDL Editor, and design development is tightly integrated into the Foundation Project Manager.

XABEL is not included in any Alliance products. A separate XABEL Interface package is available for download which can be used to develop ABEL modules to be included as macros in schematic-based design (see ABEL modules for Alliance designs below). To download XABEL go to the Service & Support website and select **xabel150.zip**.

No workstation version of XABEL software

The ABEL6 compiler from Synario is not available for any UNIX workstations. ABEL modules to be included in designs prepared on workstation-based schematic capture systems should be compiled on a PC. The EDIF netlist for each ABEL module can be transferred to the workstation and included in the top-level design.

The older ABEL5 compiler that was available with XACT version 5 was available in a UNIX compatible form. XNF netlists (for FPGAs) or Plusasm equation files (for CPLDs) produced by that interface are still readable by Xilinx M1.5 software. However, the ABEL5 software cannot be integrated into the Xilinx-M1 system and cannot be used to produce EDIF netlists.

Documented in Foundation on-line help

Because XABEL is sold only in the Foundation product, all documentation supporting the XABEL interface is included in the Foundation on-line help system. This documentation includes ABEL design techniques for FPGAs and CPLDs, and an ABEL-HDL Language Reference. The on-line help document can be useful for preparing ABEL modules for use in an Alliance design. The set of on-line help files normally shipped with Foundation 1.5 can be downloaded separately from the Service & Support website; the filename is **`f15_help.zip`**.

Capabilities

Supports all families

All XABEL designs are compiled into EDIF netlists consisting of architecture-independent Xilinx Unified Library components, which can be read and incorporated into designs targeting any Xilinx device family.

Creates macro modules and stand-alone designs

The XABEL Interface can optionally add pad components to all pins declared in the ABEL design, thus generating a top-level EDIF netlist that can be read directly by the Xilinx implementation software. Otherwise, pad components can be omitted to generate a netlist used to define the logic of a macro symbol embedded in a schematic design. Only simple input and output buffers (including tristate outputs) are supported by XABEL. So any FPGA designs requiring clock buffers or registered input/output pads should be based on top-level schematics containing the desired I/O symbols.

Based on Synario ABEL 6 with hierarchy

The ABEL compiler used in the XABEL Interface (Xilinx version 1.5) is the ABEL version 6.3 engine from Synario Design Automation, formerly a subsidiary of Data I/O, and now a subsidiary of Minc.

The ABEL6 compiler has several improvements over ABEL5, including the ability to combine multiple ABEL modules into a hierarchical design.

EDIF netlists are encrypted

The XABEL Interface uses a special version of the Synario ABEL compiler which does not require a parallel port security key. Because of this, the EDIF netlists produced by XABEL are encrypted so they cannot be used to target any non-Xilinx technologies. Xilinx cannot provide the non-encrypted version of any EDIF netlist produced by XABEL to any of our customers.

Even though the netlist is encrypted, functional simulation of the XABEL design is supported at 3 levels:

1. Test vectors embedded in the ABEL source design are automatically simulated during compilation by the built-in ABEL simulator (BLIFSIM), producing a tabular report file.
2. The Foundation gate-level simulator can perform direct functional simulation on the encrypted EDIF netlists.
3. A simulation netlist can be generated by the Xilinx implementation software after the top-level design is read and translated. The simulation netlist consists of Xilinx simulation primitives (simprims) that can only be used for simulation modeling and not for design entry.

Installation requirements

Local hard-drive only

The ABEL compiler software does not run reliably when installed on some networks. Therefore, XABEL should only be installed onto the local hard disk of the PC.

Same directory as Xilinx-M1

The XABEL interface is dependent on Xilinx-M1 implementation (core) software. XABEL should therefore reside in the same directory as the Xilinx-M1 implementation software. This is the directory referenced by the XILINX environment variable. This requirement applies whether XABEL is used with Foundation software or with Alliance software.

Workaround for installing implementation tools on network

Because XABEL must be installed to a local hard drive, the entire Xilinx-M1 implementation software would normally also need to be installed on the same local hard drive if XABEL is to be used. If you need to install Xilinx implementation software to a network drive and you want to run XABEL on the same PC, it is possible to install each to a different installation directory by using the MYXILINX environment variable:

1. Install the implementation software (without XABEL) to a network drive and let the installer set the XILINX environment variable and PATH in the registry, as usual.

2. Install the XABEL interface to the PC's local hard-disk drive.

If you are using the installer on the Foundation 1.5 CD, install XABEL as follows:

a) Select the **`Typical Install`** type.
b) Select a **`Destination Folder`** on your local hard-drive.
c) In the **`Select Software Components`** window, remove the check-marks from all software components, highlight **`Implementation Tools`** and click **`Change`**.
d) Set the check-mark next to **`XABEL Interface`** and click **`Continue`**.
e) Proceed to the **`Select Registry Setting`** window and turn on only the following two options: **`Set/Update PATH in Registry`** and **`Initialize XABEL Registry Settings`**.

If you downloaded the XABEL interface from the Xilinx website, install as follows:

a) Select a destination directory on your local hard-drive.
b) After installation, manually add the XABEL's bin\nt directory to the front of your PATH variable. (On Windows-NT systems, use System Properties to update the PATH in your environment. On Windows 95 systems, add the XABEL executable directory to the PATH setting in your autoexec.bat file.)

3. After installation, manually create a MYXILINX environment variable and set it to the XABEL root installation directory (the destination directory you specified during installation). On Windows-NT systems, use System Properties to set your environment. On Windows 95 systems, set the MYXILINX variable in your autoexec.bat file.

What's different in XABEL 1.5

In general, features of the XABEL interface, language and flow remain the same between version 1.4 and 1.5. However, a number of significant problems have been resolved.

XABEL registry not corrupted by other ABEL products

In version 1.4, the Windows registry entries for XABEL, which controlled the software's licensing mechanism, were adversely altered if a different version of the ABEL compiler was installed by a different (non-Xilinx) product, including the Synario and Workview Office design systems. Similarly, the installation of XABEL may have caused a non-Xilinx version of ABEL to stop running.

The ABEL compiler used in version M1.5 no longer has this problem. Different versions of the ABEL compiler installed by different products maintain separate registry entries. The compiler gets the proper licensing information from the registry for the ABEL product being invoked.

OLE no longer used by XABEL interface

On some Windows-95 systems, the XABEL version 1.4 translator (**`abl2edif`** or **`abl2pld`**) ran correctly the first time it was invoked, but would not run again subsequently. This was because of a problem with the OLE communications server (**`ntolesrv.exe`**) used by the ABEL compiler prior to M1.5. This typically required you to re-boot your PC to clear the problem.

The ABEL compiler used in version M1.5 software no longer depends on the OLE server.

INIT values no longer reversed for registers with asynchronous preset

In the M1.3 and M1.4 versions of the XABEL interface, there was a bug which causes the initial state to be reversed for some flip-flops in CPLD designs. If your design contained a register with an asynchronous preset (**.AP**) equation, but without an asynchronous reset (**.AR**) equation, the register was implemented in negative-logic form that effectively reversed its initial state, whether specified by an **INIT** property or by default. That is, by default, any register with asynchronous preset (and no reset) would have initialized to logic one.

This problem has been fixed in M1.5. Initial states of registers are set correctly, both by default (logic zero) and by means of the **INIT** property, regardless of the **.AP**/**.AR** equations that may be specified.

One workaround to the previously existing problem was to specify an **INIT** property with the opposite initial state value than the one you wanted. For example, if you had a register with an **.AP** condition and no **.AR** condition in your design and you wanted that register to initialize to logic zero (normally the default), you would have specified the following declaration:

```
xilinx property 'INIT=S my_register';
```

If so, these reversed property declarations should be removed (if you want to preload to logic zero) or else corrected before processing your design using XABEL version 1.5.

An alternative workaround to the previously existing problem was to specify a dummy **.AR** equation in addition to your existing **.AP** equation. For example:

```
my_register.AR = 0;
```

In this case, you do need to modify your design before running XABEL 1.5. The dummy **.AR** equation will not cause any problems.

ABEL design techniques

This section discusses special precautions to take while developing your ABEL source design.

Name of module must match filename.

The module name specified in each ABEL file must match the name of the file. For example, if your file is named myfile.abl, it must contain “module myfile” at the top.

If you are creating a hierarchical ABEL design, each module of the design must be contained in a separate file and each filename must match the contained module name.

Pin vs. internal feedback

In a top-level ABEL design, if you define an output pin and also use the name of that output pin as the name of a signal source in another equation, the interpretation of that signal name reference may be ambiguous.

For example, your ABEL design may contain:

```
my_output pin;
equations
my_output = a & b;
y = my_output & x;
```

The ABEL compiler may interpret **my_output** in the last equation as the internal feedback from the logic expression **a & b**. Alternatively, it may implement **my_output** as a bidirectional pin and interpret the reference to **my_output** in the last equation as the pin input. Generally, the latter interpretation will occur if you also define a tristate output enable condition for **my_output**, as in:

```
my_output.oe = my_oe;
```

To prevent ambiguity, you should specify the **.FB** extension for internal feedback (before tristate control), or the **.PIN** extension for bidirectional pin input, wherever an output pin name is used as an input name. The equation for **y** should therefore be written as either:

```
y = my_output.FB & x;
```

or

```
y = my_output.PIN & x;
```

Feedback interpretation in XABEL-M1 different than in pre-M1 versions.

The interpretation of feedback is different in Xilinx-M1 than in earlier versions of the XABEL Interface, such as in XACT version 6 (or earlier) or in the XABEL-CPLD product (this applies only to top-level ABEL designs for CPLD). In pre-M1 versions, references to a feedback signal were always interpreted as internal feedback unless the **.PIN** extension was specified. In XABEL-M1 (using the EDIF interface), references to feedbacks from outputs with tristate enable conditions are interpreted as pin feedback unless the **.FB** extension is specified. For example, in the following case, the pre-M1 version of XABEL would interpret the feedback from `my_output` in line 4 as internal feedback (before the tristate control); XABEL-M1 interprets **my_output** in line 4 as input from a bidirectional pin (after tristate control).

```
my_output pin;
equations
my_output = a & b;
y = my_output & x;  // line 4
my_output.oe = my_oe;
```

Note: The obsolete Plusasm flow provided with XABEL-M1 interprets feedback the same way as pre-M1 XABEL; i.e., it assumes internal feedback unless “.PIN” is specified.

Register support

XC3000-series FPGAs do not support asynchronous preset registers

The XC3000-series FPGA families (including XC3100 derivatives) do not support the use of asynchronous preset in their logic cell registers. The implementation software will reject any flip-flop primitives that have asynchronous preset pins that occur in the EDIF netlist produced by XABEL. The XABEL translator itself does not perform any family-specific design rule checking, such as the use of asynchronous preset. If you specify an **.AP** equation in an ABEL design and attempt to target an XC3000-series FPGA, you will get a fatal error during the translate step of implementation.

Asynchronous preset can be emulated, however, by inverting the contents of the register. To do this, you must invert the D-input to the register and the Q-output from the register. Then specify an **.AR** equation instead of **.AP** to define the asynchronous condition. When the flip-flop is reset by the **.AR** equation, the resulting zero state in the flop will be interpreted as a logic one in the context of your design, because of the Q-output inversion.

To invert the Q-output of a flip-flop, you must not use the inversion operator (**!**) directly on the register's output variable in the registered assignment equation (for example, **!my_reg := !my_input**). That would be interpreted by ABEL as an inversion that occurs on the flop's D-input, not its Q-output. Instead, invert all occurrences of the register's feedback when used as a signal source in all other equations. If the register drives an output pin, you must insert an inverter between the register and the output pin using a separate equation. Change your output pin declaration from registered to combinatorial and declare a new registered node to represent the inverted register. Use a combinatorial equation to set the output pin equal to the inverse of the registered node.

For example, the following would be how a registered pin and registered node with asynchronous preset would normally appear for Xilinx families other than XC3000:

```
ext_reg pin istype 'reg';
int_reg node istype 'reg';
equations
ext_reg := input1;
ext_reg.AP = my_preset;
ext_reg.clk = my_clock;
int_reg := input2;
int_reg.AP = my_preset;
int_reg.clk = my_clock;
other1 = ext_reg.FB & input3;
other2 = int_reg.FB & input4;
```

The following shows how the above example would need to be modified to target an XC3000-series FPGA:

```
ext_reg pin istype 'com';
ext_reg_not node istype 'reg';
int_reg node istype 'reg';
equations
ext_reg_not := !input1;
ext_reg_not.AR = my_preset;
ext_reg_not.clk = my_clock;
ext_reg = !ext_reg_not;
int_reg := !input2;
int_reg.AR = my_preset;
int_reg.clk = my_clock;
other1 = !ext_reg_not.FB & input3;
other2 = !int_reg.FB & input4;
```

For XC9500XL, the .CE equation forces the use of the clock-enable product-term

When targeting an XC9500XL device, logic specified in a **.CE** equation for a register will be implemented using the clock enable p-term of the XC9500XL macrocell, provided the same register does not also have both **.AR** and **.AP** equations defined. If you use **.CE** equations and target an XC9500XL device, you may find that the CE logic for some registers may not get optimized into the same macrocell as the flip-flop output.

The XC9500XL macrocell contains only a single product-term to implement clock enable input logic. The CPLD fitter does not attempt to transform your CE logic into multiplexer logic on the D-input of the flip-flop if it cannot be completely implemented using the clock enable p-term. In general, only non-inverted, single p-term expressions in a **.CE** equation can be completely implemented using the CE p-term of the XC9500XL macrocell. If you specify a more complex logic function in the **.CE** equation and it does not get completely implemented on the clock enable p-term, your design may use extra macrocell resources and incur combinational macrocell feedback delays.

If you do not want the logic specified in a **.CE** equation to be implemented using the clock enable p-term in the XC9500XL macrocell, you can simply add dummy **.AR** and **.AP** equations for the register. When any register has **.CE**, **.AR** and **.AP** equations all specified, the logic in the **.CE** equation automatically gets expanded into a multiplexer that routes the flip-flop's Q-feedback into its D-input; the clock enable p-term is not used. Dummy **.AR** and **.AP** equations can be added to a register as follows:

```
my_register.AR = 0;
my_register.AP = 0;
```

After the CE logic is decomposed onto the flop's D-input, the dummy **.AR** and **.AP** inputs to the register will be automatically trimmed away by the CPLD fitter.

Note: When targeting an XC9500 device, any CE equations in your design will always be decomposed onto the flip-flop's D-input.

Register initial states

For FPGA use .AP and .AR equations

In designs targeting FPGA devices, the initial (power-on) state of each flip-flop is determined by whether there is an asynchronous set or reset condition specified. The initial state always coincides with the specified asynchronous condition. Registers with an asynchronous reset (**.AR** equation) will initialize to logic zero. Registers with an asynchronous preset (**.AP** equation) will initialize to logic one. Registers with neither asynchronous reset or preset will, by default, initialize to logic zero. If you just want to specify an initial state, but do not want to actively reset or preset the flip-flop, just define an **.AP** or **.AR** equation with the value of logic zero, as in:

```
my_register.AP = 0;
```

If you are targeting an XC3000-series FPGA, you must not use the **.AP** equation to specify an initial state of logic one. Instead, invert the register as described in *XC3000-series FPGAs do not support asynchronous preset registers* above.

For CPLD use INIT property

In designs targeting CPLD devices, the initial (power-on) state of each flip-flop is determined by the **INIT** attribute specified using a Xilinx property statement. Unlike FPGA devices, initial states in CPLDs can be set independently of any asynchronous set or reset conditions applied to the registers. Also, CPLD devices support both asynchronous reset and preset on each register, so you can specify both an **.AR** and **.AP** equation for the same registered signal.

To specify the initial state of a register in a CPLD design, use the following declaration in the ABEL design:

```
xilinx property 'INIT=state
signal_list...';
```

where state is either **R** for reset (logic zero) or **S** for set (logic one). For example:

```
xilinx property 'INIT=S my_reg1 my_reg2';
```

By default, all flip-flops in a CPLD design initialize to logic zero, regardless of any **.AR** or **.AP** equations that may be specified.

FSM initial state

For FPGA designs

For FPGA designs, 1-hot is the preferred state machine encoding style due the abundance of registers. 1-hot FSMs can be expressed either symbolically or explicitly. Either way, one of the state flip-flops must initialize (power-up) to the logic one state so that the FSM begins in a legal state.

Symbolic 1-hot FSMs

If you are defining a symbolic 1-hot FSM for an FPGA device, always use the **async_reset** statement to identify the initial state of the FSM. If you want to actively and asynchronously reset the FSM to its initial state, declare both the initial state and the asynchronous reset condition as follows:

```
async_reset state_name : reset_condition;
```

If you only want to specify the initial state, but do not wish to actively reset the FSM, declare just the initial state as follows:

```
async_reset state_name : 0;
```

Explicit 1-hot FSMs

If you are defining a 1-hot encoded FSM explicitly using ordinary registered signals, always define an asynchronous preset condition (**.AP** equation) for the register that corresponds to the initial state of your FSM. If you want to actively and asynchronously reset the FSM to its initial state, declare both the initial state register and the asynchronous reset condition as follows:

```
initial_reg_name.AP = reset_condition;
```

Then also declare the same condition for resetting the remaining registers of the FSM, as follows:

```
other_reg_name.AR = reset_condition;
```

If you only want to specify the initial state, but do not wish to actively reset the FSM, declare just the initial state as follows:

```
initial_reg_name.AP = 0;
```

You do not need to specify **.AR** equations for the remaining flops because they will, by default, initialize to logic zero.

If you are targeting an XC3000-series FPGA, you must not use the **.AP** equation to specify an initial state of logic one. Instead, invert the register as described in *XC3000-series FPGAs do not support asynchronous preset registers* above.

INITIALSTATE property ignored (bug)

The Xilinx property **INITIALSTATE** was supported in earlier (pre-M1) versions of XABEL as a way to specify the initial state of 1-hot symbolic FSMs. This property is still supported in the XABEL-M1 EDIF interface, but it currently does not work properly for FPGA designs. In XABEL-M1, the **INITIALSTATE** property translates to an **INIT=S** property in the EDIF netlist, which is not supported for FPGA designs. Please use the **async_reset** statement or **.AP** equation extension instead, as described above.

For CPLD designs

Binary encoded FSM

For CPLD designs, binary encoding is the preferred coding style because of the high logic-to-register ratio in the architecture. In a binary encoded FSM, the initial state is commonly assigned the all-zero value. Since logic zero is the default initial state of all flip-flops in CPLDs, there is typically no need to do anything special to define the initial (power-up) state of the FSM.

If you want to actively and asynchronously reset the binary-encoded FSM to its initial state, simply define **.AR** equations for all the state registers, assuming an all-zero initial state value.

Symbolic 1-hot FSM

Alternatively, 1-hot encoding is also applicable to CPLD designs and may prove to yield better performance in cases where there are fewer states and state transition logic is particularly complex. 1-hot FSMs can be expressed either symbolically or explicitly. Either way, if you are defining a 1-hot FSM, one of the state flip-flops must initialize (power-up) to the logic one state so that the FSM begins in a legal state.

If you are defining a symbolic 1-hot FSM for a CPLD device, use the Xilinx property **INITIALSTATE** to declare the initial (power-on) state of the FSM, as follows:

```
xilinx property 'INITIALSTATE
state_name';
```

This will automatically apply the property **INIT=S** to the flip-flop corresponding to the initial state in the EDIF netlist. The remaining state flops will, by default, initialize to logic zero on power-up.

If you want to actively and asynchronously reset the symbolic 1-hot FSM to its initial state, then also declare the asynchronous reset condition using the **asynch_reset** statement as follows:

```
async_reset state_name : reset_condition;
```

This will automatically generate an asynchronous preset condition for the initial state flop. You should continue to also specify the Xilinx property **INITIALSTATE** to define the power-on state, as described above. Simply specifying an asynchronous preset condition does not imply a high power-on state for registers in CPLD devices.

Explicit 1-hot FSM

If you are defining a 1-hot encoded FSM explicitly using ordinary registered signals, use the Xilinx property **INIT** to set the initial state flip-flop of the FSM to logic one at power-on, as follows:

```
xilinx property 'INIT=S
initial_reg_name';
```

The remaining registers will, by default, initialize to logic zero.

If you want to actively and asynchronously reset the explicitly-defined 1-hot FSM to its initial state, then also declare an asynchronous preset equation for the initial state register, and declare asynchronous reset equations for the remaining state bits, as follows:

```
initial_reg_name.AP = reset_condition;
other_reg_name.AR = reset_condition;
```

You should also continue to specify the Xilinx property **INIT=S** for the initial state flop to define its power-on state, as described above. Simply specifying an asynchronous preset condition does not imply a high power-on state for registers in CPLD devices.

Transparent latches

.LH equations use flip-flops in CPLDs

When you use the **.LH** equation to define a transparent latch, a Xilinx latch primitive (LD) is used in the resulting netlist. For example:

```
mylat node istype 'reg_l';
Equations
mylat := d_input;
mylat.LH = latch_enable;
```

For CPLD designs, the latch primitive is implemented as a flip-flop with the grounded clock and the data input gated by the latch-enable into the flop’s asynchronous reset and preset inputs. Essentially, the above latch equations are implemented in a manner equivalent to the following flip-flop equations:

```
mylat node istype 'reg_d';
Equations
mylat := 0;
mylat.clk = 0;
mylat.AP = d_input & latch_enable;
mylat.AR = !d_input & latch_enable;
```

In CPLD devices, each macrocell flip-flop has a single p-term available for each of its reset and preset inputs. A simple latch equation in which the data and latch-enable inputs are primary inputs (from an input pin or a register output) is implemented efficiently in a single macrocell. However, if the data input is preceded by any combinatorial logic, or if the latch-enable input is any logic function other than a simple AND-gate, then that combinatorial logic cannot be optimized into the same macrocell as the flip-flop and a macrocell feedback delay will be incurred.

Combinatorial feedback latches not implemented well in CPLDs (bug)

Normally, there is an alternative way to express a transparent latch is by using a combinatorial feedback equation, as follows:

```
mylat node istype 'com, retain';
Equations
mylat = d_input & latch_enable
   # mylat & !latch_enable
   # mylat & d_input;
```

The **retain** attribute turns off Boolean minimization in the ABEL compiler and CPLD fitter to retain the redundant product term "**mylat & d_input**" required to implement a stable combinatorial feedback latch. Without this redundant term, a logic one being stored into the latch may get lost when the **latch_enable** input transitions from high to low (if **latch_enable** goes low before **!latch_enable** becomes high).

Normally, all logic in a combinatorial feedback latch equation should be implemented in the same macrocell. However, the current version of the CPLD fitter has a problem processing the feedback loop, and often implements the latch incorrectly using two macrocells per latch. If possible, use **.LH** equations shown above to represent transparent latches using macrocell flip-flops. Otherwise, the obsolete Plusasm flow provided in F1.5 can successfully implement combinatorial feedback latches without encountering the CPLD fitter problem.

If you want to construct a combinatorial feedback latch in your ABEL design that will implement efficiently using the EDIF interface, you must apply the **COLLAPSE** property to the intermediate nodes of the latch. First, break down each latch equation into four separate intermediate equations, one for each of the three product terms and one for the sum term. Then use the special **BLOCK** property in ABEL to apply **COLLAPSE** to each of the three p-term nodes, as follows:

```
mylat node istype 'com, retain';
mylat_dg, mylat_dq, mylat_qg node istype
'com';
xilinx property 'BLOCK mylat_dg COLAPSE';
xilinx property 'BLOCK mylat_dq COLAPSE';
xilinx property 'BLOCK mylat_qg COLAPSE';
Equations
mylat = mylat_dg # mylat_dq # mylat_qg;
mylat_dg = d_input & latch_enable;
mylat_qg = mylat & !latch_enable;
mylat_dq = mylat & d_input;
```

If you need to use several such latches in your design, you can save work by creating ABEL macros for the declarations and equations used in the latch. Then the macros can be used in several instances for all the latches you need to implement. An example of how to create and use such a latch macro is provided on the Xilinx website under Service & Support; the file is **latch_9k.zip**.

Cannot preserve delay buffer nodes in ABEL

When using XABEL, it is impossible to preserve a combinatorial delay buffer node (or inverter node). A buffer node is any node equation of the form **a=b** or **a=!b**. Even if you apply the **istype'KEEP'** attribute to the node, the **abl2edif** translator still removes the node. This is because the EDIF netlist writer used in the Synario ABEL compiler is unable to maintain buffer nodes.

For CPLD designs, you can work around this problem by replacing the buffer node with a pass-through register. First, declare the node using **istype'reg'** (instead of **'com'**). Then, replace the buffer equation **a=b** with the following:

```
a := b;
a.ap = b;
a.ar = !b;
a.clk = 0;
```

This will create a flip-flop that passes through the value of **b** asynchronously, similar to a transparent latch permanently enabled. However, do not try to use a latch equation (**a.lh=1**) because ABEL will automatically convert it to a buffer node and remove it. If **b** is an input pin or register feedback, this will work as is. If **b** is a combinatorial node, you will also need to apply **istype'keep'** to the declaration for **b** so that its logic is not collapsed into the **.ar** or **.ap** equations.

Large comparator/decoder logic may cause ABEL compiler to fail (bug)

Large comparator or decoder functions may prevent compilation of an ABEL design. After invoking the ABEL compiler (**abl2edif** translator), you may observe that the **AHDL2BLF** step completes, then the **BLIFOPT** step begins but never completes. In other cases, the process runs all the way into the **BLIF2NET** step, which either never completes or produces a system error after exhausting all available virtual memory. One possible cause is that the design contains some large comparator logic (of the form Y = A == B) or decode logic (of the form Y = A == constant), especially if the comparator or decode logic is combined with any other combinatorial logic in the same equation. The suspected cause of this problem is that the ABEL compiler is trying to process both the positive logic and negated logic form of the expression which contains an exceedingly large number of min-terms.

To remedy this problem, insert the following ABEL directive at the top of your declaration section or before the problematic equation:

```
@carry 2;
```

This will cause the ABEL compiler to break up the large equation into a number of smaller intermediate nodes, which can then be translated into EDIF more efficiently. The

Xilinx Implementation software will optimize the logic across the intermediate nodes.

Attributes for controlling design implementation

The discussion in this section assumes you are familiar with the capabilities of the target PLD architecture and the general means for controlling the implementation software. This section focuses on the techniques and issues specific to controlling XABEL designs.

Pin assignment

Numeric pin names in ABEL "pin" declarations

You can indicate pin assignments for most packages directly in your ABEL pin declarations. For example:

```
a pin 34;
b, c pin 35, 36 istype 'reg';
```

Pin numbers are applicable only to top-level ABEL designs for either FPGA or CPLD devices.

BGA pin names in UCF file

ABEL only accepts numeric pin numbers. If you are using a BGA package (which uses alphanumeric pin designations), specify your pinout in a User Constraint file (UCF) using the **LOC** property. For example:

```
net a LOC=A7;
```

Output slew (FAST, SLOW)

You can control slew rate for specific output or I/O pins with the **FAST** and **SLOW** attributes. Use these attributes selectively to control whether specific pins operate in fast slew rate (**FAST**) or slew rate limited (**SLOW**) mode as follows:

```
xilinx property 'FAST | SLOW signal_list';
```

For Example:

```
xilinx property 'SLOW q1 q2 q3';
```

The **FAST** and **SLOW** properties are applicable only to top-level ABEL design for either FPGA or CPLD devices.

Preserving combinatorial nodes (KEEP)

If you want to preserve a combinatorial node in your ABEL design so that it remains intact throughout design implementation, apply the attribute **KEEP** to the node declaration in your ABEL design as follows:

```
signal_list node istype 'KEEP';
```

The **KEEP** attribute is recognized by the ABEL compiler so that it will not collapse the node during ABEL compilation and netlisting. The **KEEP** attribute is also propagated to the EDIF netlist so that the core implementation software (CPLD fitter or FPGA mapper) does not collapse the node during design optimization.

The **KEEP** attribute is applicable to top-level ABEL designs and modules for schematic designs.

Global buffers for CPLD (BUFG)

You can manually assign selected input pin signals in your top-level ABEL design to global nets on a CPLD using the **BUFG** attribute as follows:

```
xilinx property 'BUFG={CLK | OE | SR}
signal_name';
```

For example:

```
xilinx property 'BUFG=CLK my_clock';
xilinx property 'BUFG=OE my_enable';
xilinx property 'BUFG=SR my_reset';
```

Note: To use a global clock buffer in an FPGA design, you must instantiate the buffer in a top-level schematic and instantiate the ABEL module as a macro in the schematic.

Macrocell power mode for CPLD (PWR_MODE)

Use the PWR_MODE attribute to selectively control whether specified logic operates in high speed or low power mode. The default power mode for the design is controlled in the Implementation Options menu of the GUI, and is initially set to STD for new projects. Use the following syntax:

```
xilinx property 'PWR_MODE={LOW|STD}
signal_list';
```

For example, to set the functions out0 and out1 to low power mode (the remaining functions will use the default power mode), use the following:

```
xilinx property 'PWR_MODE=LOW out0 out1';
```

Timespecs

For top-level ABEL designs, timespecs must be specified in a UCF file. For ABEL modules in a schematic design, timespecs are typically added to the schematic, but could also be specified in a UCF file.

Either way, timespecs can reference timing groups by names attached to elements in the design. You can attach a timing group name (**TNM**) to a signal in an ABEL design or module using the **TNM** property, as follows:

```
xilinx property 'TNM=group_name
signal_list';
```

XC9500 local feedback

The local feedback path will be used when it is required to meet a timing constraint, provided the fitter can group the required logic functions together in the same function block. If necessary, you can control function block mapping using a special form of the LOC attribute. The LOC property required to control function block placement of internal logic is as follows:

```
xilinx property 'BLOCK signal_name
LOC=FBnn';
```

For example, assume you want to group flip-flops REG_X and REG_Y into the same function block so that you can speed up the cycle time by using the local feedback path. Also assume there is some combinatorial logic between these flip-flops that was not fully optimized, resulting in an additional macrocell feedback between the flip-flops and whose output is GATE_A. You would also need to group the combinatorial logic (GATE_A) into the same function block. To place these functions in function block 1, the required Xilinx **LOC** properties would be expressed as follows:

```
xilinx property 'BLOCK REG_X LOC=FB1';
xilinx property 'BLOCK REG_Y LOC=FB1';
xilinx property 'BLOCK GATE_A LOC=FB1';
```

Note: You may specify only one node name per BLOCK property statement.

Mapping ABEL equations directly to CPLD macrocells

Occasionally, the CPLD fitter may implement a particular equation or set of equations less efficiently than originally expressed in ABEL. When an ABEL design is compiled and translated into an EDIF netlist, each equation is broken down into a network of primitive gates, and intermediate nodes are generated by the compiler to interconnect the primitive gates. By default, the fitter does not give preference to the partitioning of logic as expressed in the ABEL equations. It is free to optimize combinatorial logic across equation boundaries and use the ABEL-generated intermediate nodes as macrocell outputs that feed back to other macrocells.

If, for certain equations in your design, you prefer that the fitter implement the equations as expressed in ABEL, you could use the **KEEP** attribute on combinatorial nodes to prevent the fitter from optimizing logic across equation boundaries. Then, if you notice that the logic for a given equations does not get fully optimized into the same CPLD macrocell, you could increase the **Collapsing Pterm Limit** in the **Implementation Options** menu of the GUI to further flatten that logic.

Processing XABEL designs

This section discusses special procedures and flows you may need to use when processing your design using XABEL.

Improving Performance in CPLD designs

For some CPLD designs you may notice that the logic of some of your equations does not get fully flattened, resulting in poor performance and, sometimes, excessive logic utilization. Performance problems can be observed in the **Timing Report** summaries. Incomplete flattening is also indicated in the **Fitting Report** under the “Resources Used by Successfully Mapped Logic” section, by the presence of several “internal” signal names that are generated by the XABEL translator. These internal names each represent a CPLD macrocell feedback used to implement intermediate logic nodes within your equations.

To get the fitter to further flatten your design, increase the **Collapsing Pterm Limit** in the **Advanced Optimization** tab of the **XC9500 Implementation Options** menu of the GUI. First try increasing the **Pterm Limit** to 90 (the maximum) and rerun the fitter. For many designs, all your equation logic will become flattened and the internal signal names will disappear from the **Fitting Report**.

For some designs, especially those containing combinatorial node equations or complex logic functions, increasing the **Pterm Limit** may cause the fitter to flatten too much and the design will not fit. You may be able to find a **Pterm Limit** less than 90 that produces satisfactory performance while successfully fitting the device. Otherwise, you may need to apply the **KEEP** attribute to some of the combinatorial node equations in your design to prevent these nodes from being flattened by the fitter when you raise the **Pterm Limit**. If you still have some equations in your design that consume too many logic resources when flattened, you may need to decompose large equations into smaller intermediate equations and apply the **KEEP** property to your intermediate nodes as needed.

ABEL modules for Alliance designs

The XABEL interface is only shipped in the Xilinx Foundation product and is not included in the Alliance product. If you are using the Alliance M1.5 software with a third-party schematic capture tool and you want to include ABEL macros in your schematic design, you can add the XABEL Interface to your PC to process your ABEL modules.

Your alternatives are:

1. Install all design entry tools from the Foundation F1.5 product and use the Foundation HDL Editor GUI for ABEL design entry and translation. Your Alliance M1.5 software can either be on the PC or workstation.
2. Install only the XABEL Interface from the Foundation F1.5 product and use a text editor for ABEL design entry and use commands in a DOS window for XABEL translation. Your Alliance M1.5 software can either be on the PC or workstation.
3. Download the XABEL Interface from the Xilinx website into your existing Alliance M1.5 installation and use a text editor for ABEL design entry and use commands in a DOS window for XABEL translation. You must have Alliance M1.5 software on your PC.

Using Foundation design entry tools for ABEL module development

This section describes how to install and use the interactive Foundation design environment to develop ABEL modules for use in a third-party (non-Foundation) schematic design.

Installation

If you are installing both Alliance and Foundation software onto the same PC, they can be installed in either order.

1. Mount the Foundation F1.5 Design Environment CD and run the installer (**setup.exe**).
2. Follow the instructions on the screen. When asked for the type of Install, choose **Typical Install**.
3. If you are running Xilinx Alliance implementation software on the same PC, make sure the installer's destination folder is set to the directory where Alliance software is (or will be) installed.
4. On the **Select Software Components** screen, deselect all components except **Design Entry Tools**.
5. Highlight **Implementation Tools** and click on the **Change** button.
6. In the **Select Sub-components** menu, select only **XABEL Interface** (you do not need to install Program Executables if you plan to implement your design using Xilinx Alliance implementation software installed separately).
7. In the next menu, select the design entry libraries for the device families you plan to target.
8. In the **Select Registry Settings** menu, make sure the **Initialize XABEL Registry Settings** box is checked. Other settings do not matter.

Design flow

The following procedure outlines the basic flow for creating a project in Foundation and compiling one or more ABEL macros to incorporate into a third-party schematic. For complete documentation of the Foundation Design Entry tools and the XABEL Interface, refer to the on-line help in the Foundation Project Manager (**Help** → **Foundation Help Contents**).

1. Invoke the Foundation Project Manager.
2. Create a new project by selecting **File** → **New Project**.
3. Choose the appropriate design directory and family.
4. Invoke the HDL Editor by clicking on the **HDL Editor** button in the Project Manager.
5. Select **Use HDL Design Wizard** or **Create Empty** to enter a new ABEL design. Otherwise, select **Existing File** and browse to find an existing **.ABL** file. The Design Wizard creates a template ABEL design based on the macro pin names you enter. The HDL Editor color codes ABEL keywords and provides syntax checking capability.
6. Since the ABEL file will be a module in a top-level schematic, be sure that the **Macro** compile switch is selected in the **Synthesis** → **Options** dialog.
7. To synthesize the ABEL code, select **Synthesis** → **Synthesize**. An EDIF (**.EDN**) netlist file will be created for the module and placed in the project directory.

Using the XABEL Interface in command-line mode

This section describes how to install and use the XABEL interface in DOS command-line mode. You can install the XABEL interface either from the Foundation F1.5 CDs or by downloading it from the Xilinx website.

Installation from Foundation CDs

If you are installing both Alliance and XABEL Interface software onto the same PC, they can be installed in either order.

1. Mount the Foundation F1.5 Design Environment CD and run the installer (**setup.exe**).
2. Follow the instructions on the screen. When asked for the type of install, choose **Typical Install**.
3. If you are running Xilinx Alliance implementation software on the same PC, make sure the installer's destination folder is set to the directory where Alliance software is (or will be) installed.
4. On the **Select Software Components** screen, deselect all components.
5. Highlight **Implementation Tools** and click on the **Change** button.
6. In the **Select Sub-components** menu, select only **XABEL Interface**.
7. In the **Select Registry Settings** menu, make sure the **Initialize XABEL Registry Settings** box is checked. Other settings do not matter.

Downloading from the web

You must install Alliance M1.5 on your PC to use the XABEL Interface provided on the web. This is because the XABEL Interface relies on software libraries included in Xilinx-M1.

You should install both Alliance M1.5 implementation (core) software and the XABEL Interface onto a local hard disk drive on your PC. This is because the XABEL Interface does not run reliably if installed on some networks.

1. The XABEL Interface is located at **Service & Support** . The filename is **xabel150.zip** (or the highest

numbered revision of **xabel15*.zip**). Download the ZIP file to any location.

2. Unzip the file using PKZip or equivalent. The ZIP file contains an installer package that you can extract to any location on your PC.
3. Read the installation and usage instructions contained in the text file **read_abl.txt**.
4. Run the installer, **setup.exe**. Follow the directions on the screen. You must set the Destination Directory to the same directory where Alliance M1.5 implementation software is (or will be) installed.

For detailed documentation on using XABEL, including CPLD and FPGA design techniques and an ABEL language reference guide, you can also download the Foundation on-line help files from the Service & Support website; the filename is **f15_help.zip**.

Design flow

When using the XABEL Interface alone (without the Foundation Design Entry Tools), ABEL design entry is performed using a conventional text editor. ABEL designs are then compiled using a single line command entered in a DOS window as follows:

1. Open a DOS window.
2. Change directory (**CD**) to the directory containing your ABEL source file.
3. Execute the **abl2edif** command as follows:

```
abl2edif -s level module_name
```

where **level** is **top** for top-level ABEL design, or **mod** (default) for module to use in a schematic. The **abl2edif** program generates the following output files:

abl2edif.log: log file of program execution

module_name.edn: output EDIF netlist

module_name.err: error log from program execution

module_name.smx: simulation output file (if the design contains test vectors)

module_name.tmv: test vector file for XC9500/XL functional test (if the design contains test vectors)

Instantiating ABEL macros in a schematic

Viewlogic Workview Office

Once you have created the EDIF file in the Foundation environment, you must instantiate that EDIF in your Viewlogic schematic.

Create a new symbol for the ABEL module complete with input and output pins. Make sure that all the input and output ports match the symbol by name. Use square brackets for bus notation: **BUS[3:0]**. If a symbol created by **SYMGEN** already exists, simply remove the **DEF=XABEL** and **FILE=module_name.abl** properties before continuing.

Two more properties must be added to complete the symbol. Right-click in the symbol window (but outside the symbol box itself) and select **Properties**. Under the **Block** tab, change the **Symbol Type** to **Module**. This will prevent the EDIF netlister from looking for an underlying schematic. Then, under the **Attributes** tab, add an attribute with a **Name of FILE** and a **Value** of **module_name.edn**. If the EDIF file is not located in the project directory, then the full path to the **.EDN** file must be specified.

If the ABEL code is modified in such a way that the input or output ports are modified, then the symbol will have to be manually updated to match the new ABEL module.

Because the ABEL module does not have a gate-level representation within the Viewlogic realm, the design will have to be compiled through **NGDBUILD** in order to process the ABEL portions before performing a functional simulation.

There is a push-button solution available for these steps. Solution #1985 in the Xilinx Answers Database contains the files and setup instructions for this flow. No changes to the timing simulation flow are required. This procedure applies to Powerview users as well, although some of the commands listed above will differ slightly for the workstation version of ViewDraw.

Mentor Graphics Design Architect

Once you have created the EDIF file in the Foundation environment, you must instantiate that EDIF in your Mentor Graphics schematic. Note that, since this EDIF file comes from a Windows 95/NT environment, you should run a DOS-to-UNIX file-conversion utility (such as **dos2unix**) on this EDIF file to avoid possible file-format problems.

Create a new symbol for the ABEL module complete with input and output pins. Make sure that all the input and output ports match the symbol by name. Use parentheses for bus notation: **BUS(3:0)**. If a symbol created by **SYMGEN** already exists, simply remove the **DEF=XABEL** and **FILE=module_name.abl** properties before continuing.

One more property must be added to complete the symbol. With the symbol for the ABEL module loaded into the symbol editor, select **Right Mouse Button** → **Properties (logical)** → **Add Single Property**. For **Property Name**, enter **FILE**. For **Property Value**, enter **module_name.edn**, where **module_name** is the name of the ABEL module represented by this symbol. If the EDIF file is not located in the project directory, then the full path to the **.EDN** file must be specified.

If the ABEL code is modified in such a way that the input or output ports are modified, then the symbol will have to be manually updated to match the new ABEL module.

Because the ABEL module does not have a gate-level representation within the Mentor realm, the design will have to be compiled through **NGDBUILD** and **PLD_EDIF2SIM** in order to process the ABEL modules and generate a suitable simulation model for **PLD_QuickSim**. **PLD_QuickSim** must then be run on the output netlist from **PLD_EDIF2SIM**. To annotate simulation values to the original schematic, you may enable cross-probing in **PLD_QuickSim**. Chapter 6, *Mixed Designs with Schematic on Top* in the Mentor Graphics Interface/Tutorial Guide, describes this process in the *Functional Simulation After Synthesis* section.

No changes to the timing simulation flow are required.

Optimization of XABEL logic

FPGA designs

Prior to Xilinx-M1, all logic from ABEL modules had to be optimized for FPGA architectures by the XABEL Interface itself before writing the XNF netlist. In M1, the XABEL Interface does not perform any of the Xilinx-specific optimization. The EDIF netlist produced by XABEL is architecturally generic. FPGA-specific optimization is performed by the **OPTX** program, which is part of the mapping step of the implementation software. **OPTX** optimizes logic on a module-by-module basis. ABEL modules are identified by the **OPTIMIZE** property which XABEL automatically writes to the root level of each ABEL module netlist. When the top-level schematic design is submitted to the implementation software for mapping, each of the ABEL netlists are, in turn, read in. The **OPTIMIZE** property remains associate with each ABEL module and **OPTX** performs logic optimization on each of the modules tagged with the **OPTIMIZE** property.

OPTX optimization is normally desirable for all ABEL modules. If you prefer to disable **OPTX** optimization on an ABEL module, you can turn off the **OPTIMIZE** property by specifying the following in your ABEL source declarations:

```
xilinx property 'OPTIMIZE=OFF';
```

CPLD designs

All logic in CPLD designs, including XABEL-generated logic, is always optimized by the CPLD fitter. The **OPTIMIZE** property which XABEL automatically writes into the EDIF netlist is ignored by the CPLD fitter. The only way to control logic optimization in a CPLD design is by using the **KEEP**, **RETAIN** and **COLLAPSE** properties (these are described elsewhere). You can also adjust the settings of the CPLD implementation options in the GUI, as described in the Foundation on-line help document.

XC9500 JEDEC test vectors

In addition to programming information, the JEDEC download files produced by the CPLD fitter software for XC9500 and XC9500XL devices can also contain functional test vectors. These test vectors can be used by the Xilinx JTAG download system as well as third-party programming equipment to functionally test CPLD devices after programming.

The CPLD fitter can automatically translate simulation test vectors embedded in ABEL designs and write them into the JEDEC file. If your ABEL design contains a Test_vector section, then in addition to performing simulation, the XABEL translator produces a test vector interface (**.TMV**) file.

How to use .TMV file

When you are ready to implement your ABEL design, open the **XC9500 Implementation Options** template in the GUI. Select the **Programming** tab and browse for the **.TMV** file produced by the XABEL Interface. The test vectors from the **.TMV** file will automatically be written into the programming JEDEC file.

Note: Only test vectors from a top-level (stand-alone) ABEL design can be used for device functional testing.

Use upper case signal names (bug)

Due to a bug in the F1.5 CPLD fitter, only signal names that are all upper-case will get their test vector information translated into the JEDEC file. Any signal name containing lower-case letters will result in **X** being written in the corresponding column of the test vector section of the JEDEC file. Therefore, if you plan to perform a device function test, use all upper-case names for all pin signals in your ABEL design.

When to use Plusasm (CPLD only)

In pre-M1 versions of the XABEL Interface, all CPLD designs were translated into the Plusasm equation language which was the file format read by the CPLD fitter. In Xilinx-M1, the primary design file format is the EDIF netlist. Plusasm is still supported by M1.5 software for back-compatibility of interfaces that still depend on Plusasm equation files. However, EDIF should be used for all new designs if at all possible.

PLUSASM LANGUAGE WILL NO LONGER BE ACCEPTED FOR DESIGN ENTRY IN LATER RELEASES OF XILINX DESIGN IMPLEMENTATION SOFTWARE.

One of the characteristics of Plusasm is that each equation in the Plusasm file (**.PLD**) is automatically forced to be fully optimized and mapped into a single CPLD macrocell. While this could be beneficial if the equations are optimally designed to suit the CPLD architecture, it could otherwise prevent the fitter from finding a successful or more efficient implementation. In contrast, when an ABEL design is translated to EDIF, the combinatorial logic in each ABEL equation is decomposed into a network of individual AND/OR gates. The fitter no longer recognizes equation boundaries and is free to optimize the logic as well as it can.

Existing design with unconverted Plusasm properties

To constrain a design when using the Plusasm flow, it was necessary to embed Plusasm language declarations in the ABEL source design. These Plusasm declarations were not in the form of conventional properties and cannot be supported, as is, via the EDIF interface. In Xilinx-M1, a new set of properties are provided that are compatible with EDIF as well as other forms of Xilinx design entry including schematics and HDLs.

If you have an existing XABEL CPLD design that required XEPLD PROPERTY or PLUSASM PROPERTY statements, you must either remove or replace these properties with the supported EDIF-compatible properties, or you must use the Plusasm flow (see *Converting Plusasm properties into M1 attributes,* next column). If your existing design does not contain Plusasm based properties, you should be able to use the new EDIF-based flow without design modification. If you are developing a new XABEL CPLD design, DO NOT USE Plusasm based properties. If you use the Plusasm flow, you cannot use any of the XILINX PROPERTY statements supported by Xilinx-M1 software.

Pinlocking designs in XABEL-M1 using pre-M1 pinouts

If you have a guide file (**`.gyd`**) containing a pinout that you want to use to pinlock a design iteration using the XABEL-M1 Interface, and the guide file was created using a pre-M1 version of XABEL, you may have difficulty with case-sensitivity of pin names. In pre-M1 XABEL, which used the Plusasm flow, all signal names were automatically converted to upper case. The resulting guide files therefore contained only upper case pin names. In XABEL-M1, which uses the EDIF flow, the case of the signal names in your ABEL file is preserved throughout design implementation. When trying to pinlock using an old guide file, if the name of a pin in the guide file does not exactly match the name read in from the EDIF netlist, the pinout information will be ignored.

To workaround this problem, either edit the old guide file to restore the case of each of your pin names to match the names in your ABEL design, or modify your ABEL design to use all upper-case names for external pins.

EDIF flow bugs with no workaround

You may find that you may not be able to work around some of the known problems in the current version of XABEL when using the EDIF flow. If none of the workarounds suggested in this application note solve the problem, the Plusasm flow can sometimes provide an alternative solution.

Persistent fitter problems or poor results

In some cases, the current version of the CPLD fitter may not be able to achieve the same quality of results as obtained when using an earlier version of XABEL based on the Plusasm flow. This typically occurs when the distribution of logic among the equations in the ABEL design is particularly well matched to the CPLD architecture. This might also occur if the earlier design was constrained using Plusasm properties in a way that cannot be reproduced using the EDIF flow.

If none of the optimization suggestions described in this application note allow you to obtain satisfactory performance, the Plusasm flow can sometimes provide an alternative solution.

Using the Plusasm design flow

The XABEL Interface, whether installed from the Foundation F1.5 CD or by downloading from the Xilinx website, provides an alternative design flow for CPLD designs using Plusasm equation files instead of EDIF netlists.

Using the Foundation Design Entry tools

If you are using Foundation F1.5 to develop a top-level ABEL design for CPLD, the Plusasm flow is enabled through the Foundation Project Manager. The Foundation tools can only use the Plusasm flow if you are developing a top-level ABEL design. If you are developing ABEL modules for use in schematic-based designs and you need to use the Plusasm flow, you must compile your ABEL modules using the command-line interface described later.

To enable the Plusasm flow in the Foundation GUI:

1. In the Foundation Project Manager, select **`File`** → **`Preferences`** → **`Configuration`**.
2. In the **`Configuration`** window, click on **`View INI File`**.
3. In the **`Report Browser`** window that appears, find the lines containing

```
[EXTENSIONS]
;XABELNETLIST=PLUSASM
```

4. Delete the semicolon (**`;`**) in front of **`XABELNETLIST`** to enable the feature.
5. Save the file (**`File`** → **`Save`**) and close the **`Report Browser`**.
6. Click **`OK`** in the **`Configuration`** window to close it.
7. Exit the Foundation Project Manager (**`File`** → **`Exit`**) and restart it to read the new configuration.
8. Create a project in which to develop your ABEL design.
9. Invoke the HDL Editor and use the **`Synthesis`** → **`Synthesize`** command as you would normally. Be sure the **`Chip`** compile switch is selected in the **`Synthesize`** → **`Options`** menu to produce a top-level Plusasm design.

After the Plusasm flow is enabled, the Foundation software creates Plusasm equation files (**`.PLD`**) for all top-level ABEL CPLD designs. When you invoke the **`Implementa-`**

tion step from the Foundation Project Manager, it will automatically read the Plusasm (**.PLD**) file for design implementation instead of looking for an EDIF netlist. FPGA designs and all ABEL macros to be used in schematic designs will continue to use EDIF netlists for design implementation.

Note: After it creates the Plusasm equations file (runs the **ABL2PLD** translator), the Foundation system continues to run the **ABL2EDIF** translator to produce an EDIF netlist. The EDIF file is used only for functional simulation, which is optional; it is not used for design implementation.

Note: If you have already compiled the ABEL design using the EDIF flow, you should create a new project before re-compiling using the Plusasm flow. Otherwise, the Xilinx Design Manager may continue to read the existing **.EDN** file instead of the new **.PLD** file.

Using DOS command-line

If you are using the XABEL Interface installed from the Foundation CD or downloaded from the Xilinx website, ABEL designs are translated to Plusasm using a single line command as follows:

1. Open a DOS window.
2. Change directory (**CD**) to the directory containing your ABEL source file.
3. Execute the **abl2pld** command as follows:

   ```
   abl2pld -s level module_name
   ```

   where **level** is **top** for top-level ABEL design, or **mod** (default) for module to use in a schematic. The **abl2pld** program generates the following output files:

   abl2pld.log: log file of program execution

   module_name.pld: output Plusasm equation file

   module_name.err: error log from program execution

   module_name.smx: simulation output file (if the design contains test vectors)

   module_name.tmv: test vector file for XC9500/XL functional test (if the design contains test vectors)
4. If you are using the Xilinx Design Manager GUI, create a project and specify (**Browse**) the **.PLD** file as the Input Design.

Converting Plusasm properties into M1 attributes

This is a summary of the Plusasm properties that were supported in the pre-M1 XABEL Interface and the equivalent Xilinx properties compatible with the XABEL-M1 EDIF-based interface:

Old Plusasm Properties	New XABEL-M1 Properties
`plusasm property 'FASTCLOCK signal_list';`	`xilinx property 'BUFG=CLK signal_list';`
`plusasm property 'FOEPIN signal_list';`	`xilinx property 'BUFG=OE signal_list';`
`plusasm property 'PARTITION FBnn signal_name...';`	`xilinx property 'BLOCK signal_name LOC=FBnn';`
`plusasm property 'LOGIC_OPT OFF signal_list';`	`signal_list {NODE | PIN} istype 'KEEP';`
`plusasm property 'MINIMIZE OFF signal_list';`	`signal_list {NODE | PIN} istype 'RETAIN';`
`plusasm property 'PWR {LOW | STD} signal_list';`	`xilinx property 'PWR_MODE={LOW | STD} signal_list';`
`plusasm property 'FAST ON signal_list';`	`xilinx property 'FAST signal_list';`
`plusasm property 'FAST OFF signal_list';`	`xilinx property 'SLOW signal_list';`