XILINX®

XAPP444 (v1.1) July 15, 2005

CPLD Fitting, Tips and Tricks

Summary

Most designers wish to utilize as much of a device as possible in order to enhance the overall product performance, or extend a feature set. As a design grows, inevitably it will exceed the architectural limitations of the device. Exactly why a design does not fit can sometimes be difficult to determine. Programmable logic devices can be configured in almost an infinite number of ways. The same design may fit when you use certain implementation switches, and fail to fit when using other switches. This application note attempts to clarify the CPLD software implementation (CPLDFit) options, as well as discuss implementation tips in CoolRunner™-II designs in order to maximize CPLD utilization.

Introduction

Trying to understand why a design does not fit is rarely black and white. There are often a number of reasons why a design will not fit. If a design claims that it cannot place an output in a particular location, is it because there are insufficient function block inputs, product terms, or control terms? Is it because of the requirements for the other equations within that function block? Is it due to a timing constraint that forces the equations to be wider instead of narrower and shareable? This application note reviews the architecture limits as a background, then defines reasons for not fitting and provides steps to follow to determine the reason and find a solution.

Note: A product term is any number of inputs that are logically AND-ed together.

Note: A control term is a product term with a special function such as asynchronous reset/preset, clock enable, clock or output enable.

Understanding the Limits

From a high level, a CPLD is comprised of interconnected function blocks. A common misconception is that if a design has product term or macrocell resources available, then there is surely room to add logic. There are a number of reasons why this may not be true. The distribution of logic into function blocks can 'strand' logic resources. Also careless pin assignments can prevent the software from taking advantage of the highly shareable architecture.

An analogy of a CPLD's resource distribution would be that of a train with a number of boxcars. Each boxcar represents a function block. Equations would be a trunk that is stored in the boxcar. Product terms could be thought of as weight for each trunk. The function block inputs could be thought of as the size for each trunk. The boxcar has both a weight and a size limit for the trunks stored in it. If you fill a boxcar with large light trunks, you can't utilize the leftover weight allowance in the boxcar because you've run out of space. Similarly if you fill a boxcar with one very heavy trunk, you can't add any more trunks because of the total weight limit. This demonstrates how a function block may be fully utilized in one capacity but not in another, and be unable to implement more logic.

A CPLD is built up from Function Blocks which are then built up by macrocells and product terms and function block inputs. When a Function Block has exhausted any one of these three resources, then the *remainder of these resources are unavailable to the rest of the device*. For example, if a Function Block uses all 16 of its macrocells but only 20 of its 48 product terms, the remaining 28 product terms are not usable. It is important to understand that the CPLD as a whole may contain X number of macrocells or Y number of product terms, but these resources are partitioned into distinct function blocks which cannot export unused resources. Therefore it is important to maximize the utilization of each function block in order to have maximum usage of the CPLD.

The CoolRunner-II Function Block contains a logic element called a Programmable Logic Array (PLA) within each Function Block (see Figure 1). This PLA allows multiple equations to share a single product term. Other CPLD architectures use a PAL which does not have this flexibility and must generate duplicate product terms for each equation. However to properly utilize this architecture, similar logic must be grouped into the same Function Block. The Xilinx CPLD software is intelligent and can recognize when product terms can be shared and automatically groups similar logic together. User pin constraints have higher placement priority than sharable logic, therefore poor pin placement can create a logic network larger than necessary.

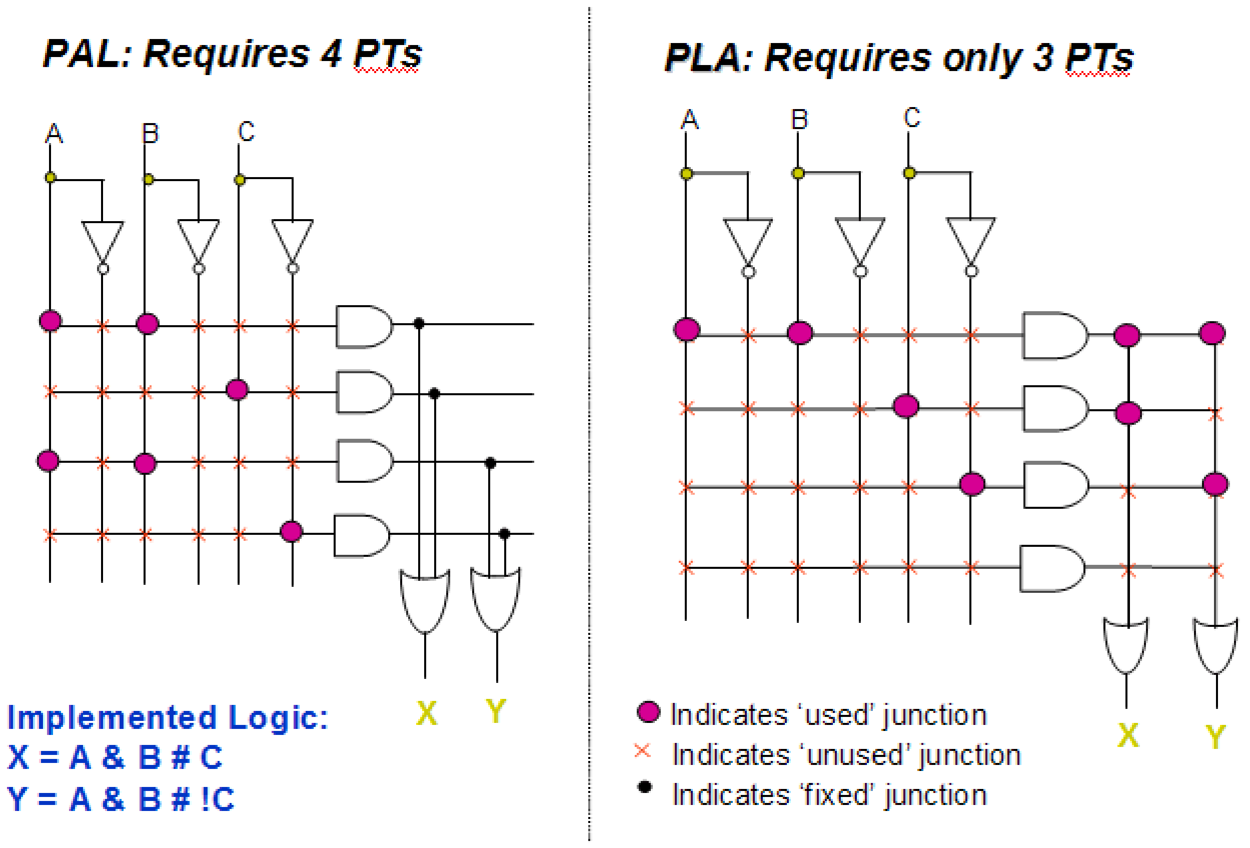

Figure 1: **PLA Implementation of Equations**

So what is the best set of software options and coding techniques that result in the highest utilization of a CPLD? There is no one set of options that will ***always*** provide the best results, as every design is different and what applies to one certainly does not apply to all. In general, Density Optimization with no pin constraints will have the greatest chance of fitting. If this does not work, or if removing pin constraints is not an option, the rest of this document will help to identify the most common reasons why a design does not fit, and provide guidance on how to modify either software switches or design elements to allow you to move forward.

Understanding Software Options

The solutions described below refer to software switches whose effects on placement and fitting may not be immediately clear. This section summarizes the options and how they impact design optimization.

Optimize Density/Speed: This is a basic fitter option for the setting of Implementation Template. The concept of optimizing for density allows for additional levels of logic whenever it results in higher sharing. The end result is that more logic can be placed into a device at the possible cost to operating frequency. This optimization generally produces smaller equations than Optimize Speed. The Optimize Speed algorithm flattens logic levels as much as possible to increase operating frequency. This results in wide logic (high input and product terms per equation) which may be difficult to place into function blocks.

Collapsing Product Term Limit: This option creates a maximum product term limit per equation. This is best observed by an example.

Given an initial design:

```
Y <= (A(0) and A(1)) or Node_a;
Node_a <= (not B(0) and B(1)) or Node_b;
Node_b <= (not C(0) and C(1)) or (D(0) and D(1));
```

A product term limit of five allows all product terms to collapse into one level of logic:

```
Y <= (A(0) and A(1)) or (B(0) and B(1)) or (C(0) and C(1)) or (D(0) and
D(1));
```

The following would occur if the product term limit were set to three:

```
Y = Y8 or (A(1) and A(0)) or (D(1) and D(0));
Y8 = (C(1) and C(0)) or (B(1) and B(0));
```

The collapsing input limit instructs the software to collapse logic until the limit of three is reached, then collapse no further. This results in two levels of logic with Y8 as an intermediate node.

Collapsing Input Limit: This option creates a maximum input limit per equation. This behavior is the same as for Collapsing Product Term Limit. Logic is collapsed until the input limit is reached.

Note: These options will not split up logic that is larger than limit produced by synthesis (XST, Synplify, etc.). For example, if an equation consists of ten product terms coming out of synthesis, and the Collapsing Product Term Limit is set to eight, CPLDFit will not split that equation into two equations.

Function Block Input Limit: Each function block has an architecture limit of 40 inputs per function block. By setting this limit below 40, this essentially preserves some inputs from being used. The benefit of this is that a design may be pinlocked, and having inputs reserved permits for more flexibility to fit the same pinout for the software should design modifications be required in the future.

Advanced Fitter Options are not visible in the ISE process properties unless the user enables them from ISE by toggling the Property Display Level.

Exhaustive Fitting: This option is a simple iterative algorithm where all combinations of Collapsing Input limit and Collapsing Product Term limit are tried. The algorithm uses the Collapsing Input limit and Collapsing Product Term limit as starting points and iterates through every combination until a fit is found. The moment a design fits, the iterations halt. At this point, the user should write down the values where the iterations stopped, so this process does not need to be repeated. If the Collapsing Input limit is set to 24 and the Collapsing Product Term limit is set to 16, this results in 24*16=384 iterations. Even if each iteration takes 5 seconds (common for smaller density devices, large devices may take a minute or more), one can see that it may take a fair amount of time to complete the iterations, so consider this before using the exhaustive fit option.

Fitting Roadblocks

This section describes the most common reasons why a design does not fit. Each section will also detail different software switches or possible design modifications to get a design to fit.

The following list gives the primary reasons why a design might not fit.

1. Design requires too many pins
2. Design requires too many macrocells
3. Design requires too many product terms

4. Design cannot fit, even though there are sufficient resources

Design Requires Too Many Pins

The fitter report will display an error message similar to the following:

```
Insufficient number of pins: needs at least X but only Y left after
allocating other resources.
```

If using SSTL2-1, SSTL3-1, or HSTL2-1 I/O Standards, remember that VREF pins are dynamically assigned if not assigned by the user (or if insufficient VREF pins are assigned by the user). Improper VREF pin assignment may result in more pins being utilized than actually necessary. For more information on VREF pin placement, see Xilinx **Application Note 399**.

Assuming none of the above I/O Standards apply and there are no removable PROHIBIT constraints, the only solution to this would be to either remove some unnecessary I/O pins from the design or to migrate to a package that has more I/O in the same device density. If no such package exists, then move up to a larger density device.

Design Requires Too Many Macrocells

The fitter report will display an error message similar to the following:

```
Design contains XX macrocells, exceeds device limit YY.
```

A macrocell implements an equation, whether registered or combinatorial. This error indicates that after optimization, there are still too many equations for the macrocells within the device.

Quick Fixes:

- CPLDFit Option: Implement the design using speed optimization
- Design Change: Use binary encoding for finite state machines
- XST Option: Set Optimize Goal in XST to Speed, and set the Effort Level to High
- CPLDFit Advanced Option: Increase the Product Term collapsing limit and the Input collapsing limit

For all designs in this category, it is helpful to re-fit for the next largest device, and examine the Logic section. This is because CPLDFit recognizes that the design will not fit immediately, and so it does not attempt to fit the design. Targeting a larger device will provide fitting and usage results that you can then analyze.

The first step is to determine if the number of macrocells used is what you expect, and also if it is reasonable for you to spend time trying to get it to fit into a smaller device. In Xilinx CPLDs, the following will help you determine the minimum number of macrocells required by your design.

- 1 macrocell per latch/register
- 1 macrocell per combinatorial output
- 1 macrocell per multi-Pterm control logic (for example, asynchronous preset/reset, clock enable, output enable)
- 1 macrocell per multiplexer

Example One:

A design has a 16-bit counter (four registers, binary encoding) that will have all its registers as output pins, logic that will output a ‘1’ whenever the count is "1100", and outputs that will go to tri-state whenever the count value is greater is "1100". This will require six macrocells to implement. Four macrocells are necessary for the counter, one macrocell is required for the combinatorial output, and one macrocell is necessary to generate the output enable. Only one macrocell is needed for the output enable because it is shared by all four register outputs.

Determining this minimum number is important because this is logic that cannot be optimized by the fitter. There is no optimization strategy to remove a register, and all outputs must use a macrocell, therefore no amount of *software* optimization will optimize out these macrocells. So this is the *best case* macrocell usage. If this minimum value exceeds the device density, the design will never fit unless registers or outputs can be removed from the design (that is, use binary encoding instead of one-hot for state machines, remove test outputs, and so on). No amount of adjusting fitter options will ever get this design to fit. It is important to note that in most cases this best case value is unattainable due to the need for buried nodes (logic equations that do not go out to pins).

In all CPLDs, equations that do not drive I/O pins are considered ‘nodes’. These can be combinatorial equations or registers. In the case of registers, they can not be optimized. However, for combinatorial equations, they may be optimized by collapsing.

Example Two:

```
dout <= equationA or equation B; (1 MC, 2 PTs, 2 inputs)
equationA <= (d1 and d2) or (d3 and d4); (1 MC, 2 PTs, 4 inputs)
equationB <= (d5 and d6) or (d7 and d8); (1 MC, 2 PTs, 4 inputs)
```

The above equations may be flattened into one macrocell.

```
dout <= (d1 and d2 ) or (d3 and d4) or (d5 and d6) or (d7 and d8); (1 MC,
4 PTs, 8 inputs)
```

Note: All examples in this document are not necessarily reproducible in software and are for the sake of clarification only.

Example Two is trivial, and would be flattened in any architecture. However, if equationA and equationB were each 10 Product Terms, one can see how flattening this logic would result in a very large equation for Dout, requiring 20 Product Terms. This clearly consumes a large amount of function block (FB) inputs and product terms and could cause possible fitting problems.

It is these nodes that we are trying to flatten in order to minimize macrocell usage (at the sacrifice of increasing Product Term usage).

First of all, re-fit using Speed Optimization. Speed Optimization will flatten logic as much as possible to get the fastest timing. This typically results in less macrocell usage, but higher product term usage as seen in Example Two.

Speed optimization may result in a fitting failure due to excessive Product Term usage. At this point one would manually decrease the Product Term limit until the proper balance is found between macrocell usage and total product term usage, and the design fits.

Sample fitting flow

This example demonstrates a design that required too many macrocells in the beginning, then by iterating through different options, arrives at an optimized design that does not require too many macrocells or product terms.

Table 1: **Fitting Options**

Implementation Attempt	Implementation Template	Product Term Limit	Function Block Input Limit	Resulting Macrocell Usage	Resulting Product Term Usage
1	Density ▼	28	32	132/128(103%)	410/448(92%)
2	Speed	32 ▼	36	122/128(95%)	470/448(105%)
3	Speed	28 ▼	36	124/128(97%)	456/448(102%)
4	Speed	24	36	126/128(98%)	439/448(98%)

Design Requires Too Many Product Terms

The fitter report will display an error message similar to the following:

Design contains 121 unique product terms, exceeds device limit 112.

Quick Fixes:

- CPLDFit Option: Implement the design using density optimization
- CPLDFit Advanced Option: Reduce the Collapsing Product Term Limit
- XST Option: Set the Optimize Goal to Density, and set the Effort Level to High

The concern in this situation is that the equations in the design are very large and exceed the total number of Product Terms available in the device. This typically occurs when the design is optimized for speed because this form of optimization flattens logic as much as possible to eliminate levels of logic which insert delay. This in turn reduces the sharing and can result in higher than necessary product term usage.

Example Three:

```
doutA <= equationA or equation B; (1 MC, 2 PTs, two levels of logic)
doutB <= equationB or d3; (1 MC, 2 PTs, 2 levels of logic)

equationA <= d1 or d2 or d3; (1 MC, 3 PTs)
equationB <= d4 or d5 or d6 or d7 or d8; (1 MC, 5 PTs)
```

The above implementation requires a total of four Macrocells and 12 Product Terms.

When using speed optimization, the fitter attempts to remove the extra level of logic for doutA and doutB by flattening equationA and equationB.

```
doutA <= d1 or d2 or d3 or d4 or d5 or d6 or d7 or d8; (1 MC, 8 PTs, 1 level
of logic)
doutB <= d3 or d4 or d5 or d6 or d7 or d8; (1 Mc, 6 PTs, 1 level of logic)
```

This flattening results in two Macrocells and 14 Product Terms. The extra level of logic for equationB was useful because it contained shared logic for both doutA and doutB. When flattened, this logic can not be shared. In addition, if doutA and doutB are in different function blocks, then d3..d8 will need to be routed to both function blocks, resulting in increased function block input requirements.

The first step would be to use timing optimization and lower the Product Term limit by half. This will generate large equations but place a cap for each equation with regards to Product Terms. If an equation exceeds this limit, the equation will be broken up to meet the limit specified. This helps create the logic sharing as shown in the original implementation of Example Three.

Lowering the Product Term limit will usually increase macrocell usage, and may cause the design to fail to fit due to excessive macrocell usage. At this point you should increase the Product Term limit until a balance is found and the design fits. The downside to this is that the Product Term limit impacts all equations, so any changes to this could have a dramatic impact on overall timing in the design.

A solution to preserving sharing is to inform the fitter not to collapse a piece of sharable logic. To do this, the user must be familiar with their design to recognize what logic is sharable. The KEEP attribute is what is used to tell synthesis and CPLDFit not to collapse a level of logic. For examples on the KEEP attribute, see Appendix A.

Note: In the above example, a KEEP attribute on Equation B would prevent it from being collapsed. However, other parts of the design would be collapsed. This allows the user to be able to selectively protect equations from overall optimization.

Design Cannot Fit, Even Though There are Sufficient Resources

The fitter report will display an error message containing the signal(s) that it cannot fit, similar to the following:

```
Cannot place 12 output(s) including 'gpio<26>' to any function block.
```

Quick Fixes:

- XST Option: Set Optimize Goal to Density, and set the Effort Level to High
- CPLDFit Advanced Option: Increase the Function Block Input Limit
- Design Change: Use binary encoding for finite state machines
- Design Change: Release pin constraints
- CPLDFit Advanced Option: Adjust the Product Term collapsing limit and the Input collapsing limit
- CPLDFit Advanced Option: Exhaustive fitting is always an option, but let it run overnight

This is the most difficult situation to resolve. First try the simple settings such as XST optimization style and effort level as well as increasing the Function Block Input Limit. Designs in this category sometimes may be very close to fitting, and a small change in the logic network could result in a placed and routed design. Increasing the Function Block Input Limit from the default of 38 is perfectly acceptable, remember the reason it is not set to 40 is to provide a notch for future belt loosening. Feel free to use it.

Determine how close a design is to fitting by looking at the number of outputs that cannot be fit. Obviously the fewer this number is, the closer you are to a fit. Also consider the size of these outputs. An equation with three product terms and five inputs is easier to place than an equation with 15 product terms and 12 inputs. Check the size of the unplaced equations by looking at the UnMapped Logic section of the fitter report.

If the simple option changes do not result in a fit, then it's time to do the real legwork. Adjust the Input and Product Term limits until you find a balance between macrocells and product terms. You do not want to settle on a combination that uses all the macrocells, as it leaves little space to work with. Conversely, you do not want a combination that uses the fewest macrocells, as that tends to create very wide equations that may be difficult to fit into heavily occupied function blocks. The goal is to make medium sized equations that can use as much of a function block's resources as possible to gain the maximum device utilization. Too many small equations will result in all macrocells being used, but with leftover product terms and function block inputs. Too many large equations will result in all product terms and/or function block inputs used, but not many macrocells used.

Review the fitter report. The fitter will automatically attempt to place related logic in the same function block so inputs will not have to be duplicated across multiple function blocks. If you find that this is not the case, manually lock equations or pins to the appropriate function blocks to achieve this. This is done with the following constraint:

```
NET mynet LOC=FBxx;
```

Where *xx* is the function block number.

So how does one determine which equations to move?

Check for function blocks with excessively low input/Product Term/macrocell usage and find one that has opposite characteristics. For example, if Function Block 1 has 12 macrocells used with 54/56 product terms used and Function Block 5 has 16 macrocells used with 19/56 product terms used, these would be ideal to exchange equations. This is because Function Block 1 could implement more equations (4 unused macrocells) but probably cannot due to lack of available product terms. Function Block 5 cannot implement more equations, but has plenty of space to implement wider ones if one was switched out. So by exchanging a wide product term equation from Function Block 1 with a small product term equation from Function Block 5 could free up product term resources and allow more equations to be placed into Function Block 1. Again, the goal is to have evenly balanced usage across all function blocks.

This data is found in the Function Block link in the Fitter Report.

Table 2: **Function Block Usage**

Function Block	Macrocells Used/Total	Function Block Inputs Used/Total	Product Terms Used/Total	Pins Used/Total	Local Control Terms Used/Total
FB1	16 / 16	34 / 40	43 / 56	5 / 11	1 / 4
FB2	16 / 16	38 / 40	43 / 56	3 / 11	1 / 4
FB3	16 / 16	38 / 40	46 / 56	8/ 11	1 / 4
FB4	16 / 16	38 / 40	42 / 56	6 / 11	1 / 4
FB5	15 / 16	38 / 40	42 / 56	9 / 11	1 / 4
FB6	12 / 16	38 / 40	37 / 56	4 / 11	1 / 4
FB7	13 / 16	38 / 40	39 / 56	4 / 11	1 / 4
FB8	2/ 16	38 / 40	28 / 56	2 / 11	1 / 4
FB9	16 / 16	25 / 40	30 / 56	5 / 11	1 / 4
FB10	16 / 16	22 / 40	26 / 56	4 / 11	1 / 4
FB11	16 / 16	24 / 40	29 / 56	2 / 11	0 / 4
FB12	15 / 16	38 / 40	41 / 56	8 / 11	2 / 4
FB13	16 / 16	37 / 40	32 / 56	11 / 11	1 / 4
FB14	13 / 16	20 / 40	26 / 56	11 / 11	1 / 4
FB15	10 / 16	37 / 40	27/ 56	10 / 11	1 / 4
FB16	12 / 16	35 / 40	45 / 56	12 / 11	1 / 4

Table 2 shows the Function Block summary of a design that does not fit. Function Block 8 clearly stands out. It shows a Function Block that has only used 2 macrocells out of 16 because it's near the limit for Function Block Inputs. Possible solutions include: increasing the Function Block Input Limit to 40 (default is 38), moving the outputs from Function Block 8 to a different function block, reducing the collapsing input limit, exchanging the high input limit equations with an equation in a different Function Block that has many inputs to spare (FB14 for example). In many cases, there is more than one solution.

Additional Information

CoolRunner-II Data Sheets, Application Notes, and White Papers

Access to all Xilinx Data Sheets, Application Notes, and White Papers

Software Documentation

Device Packages

Conclusion

Understanding the fitter report and the implementation options are the skills required to determine why a design does not fit. Using the information from the fitter report, one can identify which resources are near the maximum and manipulate the manner in which the fitter optimizes either the entire design or specific equations and therefore guide the software to finding a proper fit.

For your reference, a sample project is available with this application note to use as a learning tool. Appendix B is a walkthrough that help you step through the process of identifying the resource bottleneck and finding a way to get the design to fit.

Design files for the walkthrough can be found at :

http://www.xilinx.com/products/silicon_solutions/cplds/resources/coolvhdlq.htm

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/22/05	1.0	Initial Xilinx release.
07/15/05	1.1	Added link to design files.

Appendix A

XST-VHDL Example:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity top is
Port ( din : in std_logic;
clk : in std_logic;
dout: out std_logic);
end top;
architecture behavioral of top is
signal dout_node : std_logic ;--define the output-signal of the flip-flop
attribute KEEP : string;
attribute KEEP of dout_node : signal is "TRUE"; --keep buffer from being -
--optimized out
component buf
port( i : in std_logic; o : out std_logic);
end component;
begin
my_buf : buf
port map (
i => dout_node,
o => dout);
process (din, clk)
begin
if (clk'event) and (clk='1') then
dout_node <= din;
end if;
end process;
end behavioral;
```

XST-Verilog Example:

```
module testkeep(input_a,clk,comb);
input input_a;
input clk;
output comb;
reg input_a_reg;
//synthesis attribute keep of input_a_reg
BUF mybuf (.I(input_a_reg),.O(comb));
always @(posedge clk)
begin
input_a_reg <= input_a;
end
endmodule
```

Abel Example:

```
MODULE top
clk PIN;
din PIN;
dout PIN;
delay node istype 'keep';
BUF external (I -> O);
U1 functional_block BUF;
EQUATIONS
U1.I = din;
delay = U1.O;
dout := delay;
dout.clk = clk;
END
```

Appendix B

The following walks through the debugging process of getting a design to fit by analyzing the fitter results and modifying user options.

- Implement the design using the default options, and you should see that the design fails to fit.

  - The error message should say: **`Design requires 34 Macrocells, exceeds device limit 32.`**
- Examine the design and calculate the minimum number of macrocells needed for the design.

  - There is a 30-bit counter (q), so each bit is a register, and cannot be minimized.

  - There are two combinatorial outputs (o1 and o2), which cannot be minimized.

  - 30 + 2 = 32, so there are two macrocells extra being used that we need to identify.
- The 'Quick Fixes' suggest the Optimize Speed mode to see if design fits.

  - This will fail with the same number of macrocells required (34).
- Examine the design to see if there are state machine registers that could be minimized via binary encoding.

  - The design only contains a counter which already is binary, so it cannot be minimized any further.
- Re-fit the design, targeting a CoolRunner-II 64A (XC2C64A) device and look at the Summary of Mapped Logic.

  - Look for any equations that you do not recognize, or do not expect.

  - Note any equations that have an input usage or product term usage very close to the maximum specified in the software options (see Compiler Options for these values).
- Signals o1 and o2 both have equations that have very high input usage; perhaps, they are constrained by the maximum input limit?

  - Increase the maximum input limit from 32 to 36 and re-fit.

  - The design should now only use 32 Macrocells.

  - Retarget to the original 32 macrocell device, and the design should now fit.