LatticeMico32 Migration Concerns Post ispLEVER 8.1 and Diamond 1.0

November 2010 Technical Note TN1221

Introduction

The LatticeMico32™ System Builder software provides a convenient user interface for building a microprocessor-based System on Chip (SoC) solution inside of Lattice FPGAs. Introduced in September 2006 it has provided a lightweight, and very capable open-source microprocessor.

Over the years that the LatticeMico32 processor has been available there have been many requests for improvements in the development system. Implementing some of these features required making some changes to the original design concepts for the LatticeMico32 development environment.

Lattice Semiconductor has decided to adopt some of the requests, and is also laying groundwork for future LatticeMico32 System Builder enhancements. Adopting these features impacts some LatticeMico32 platforms and firmware developed on LatticeMico32 System Builder software prior to ispLEVER® 8.1 SP1 and Lattice Diamond™ 1.1. We have strived to minimize the impact to existing LatticeMico32 platforms.

This document describes the changes made in the LatticeMico32 System Builder software and how these changes need to be managed by you, as the developer of a LatticeMico32 platform.

Definitions of Terms

These terms are used throughout this document:

- WORD/HWORD/BYTE: 32-bit, 16-bit, and 8-bit data or address alignment
- MSB: Most Significant Byte (LatticeMico32 data bus bits 31:24)
- LSB: Least Significant Byte (LatticeMico32 data bus bits 7:0)
- ELF: Executable and Linking Format output from GCC/LD
- Program: ELF output file that is the result of compiling and linking C/C++ source code.
- GCC: GNU C Compiler
- LD: GNU binutils linker
- GPIO: General Purpose Input/Output
- SEL: Wishbone byte lane select control strobes

Platform Migration Decisions

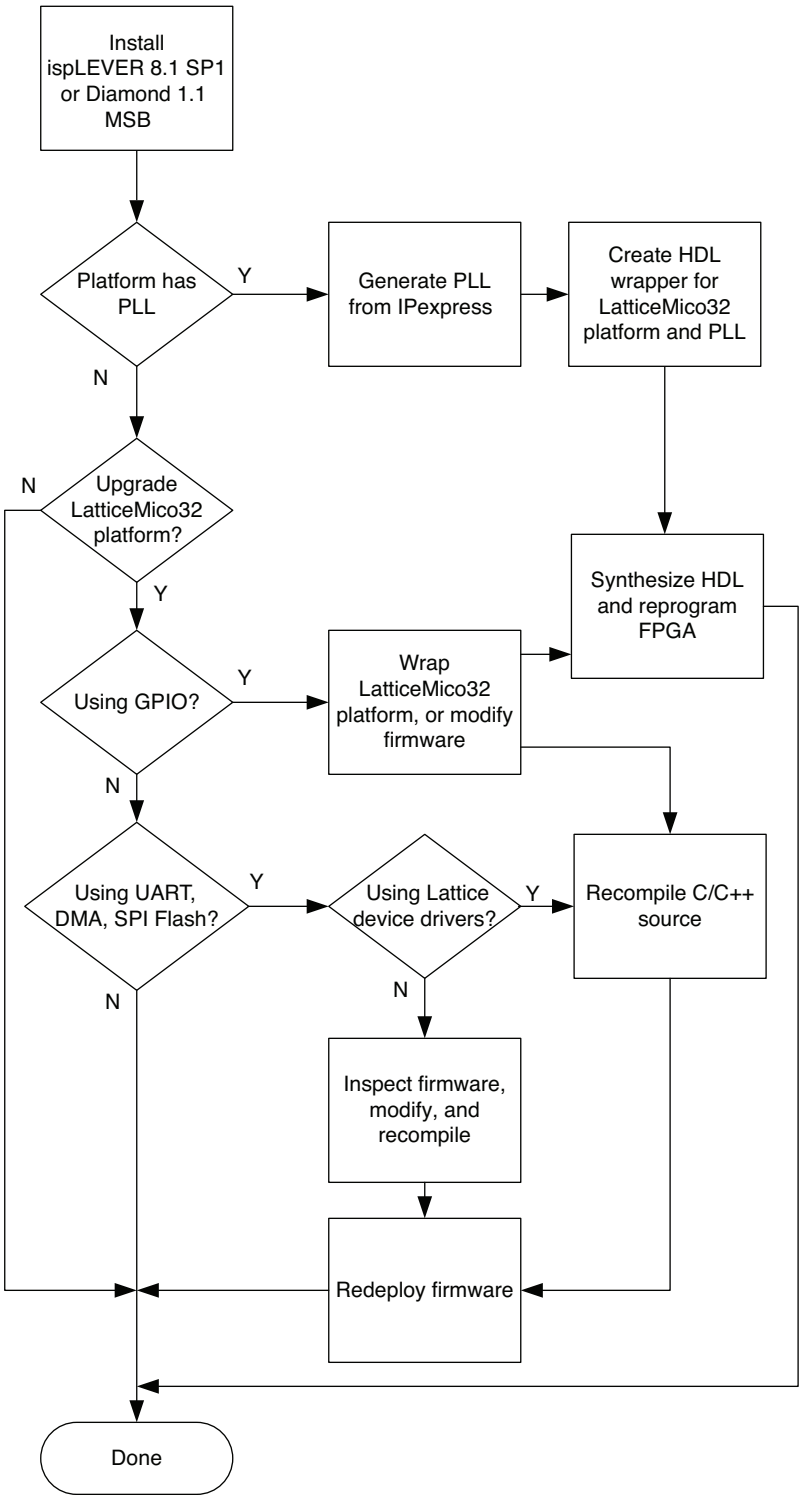
When you install the latest LatticeMico32 System Builder tools you must decide how and when to upgrade your LatticeMico32 design. Each component in a LatticeMico32 platform has a version number associated with it. LatticeMico32 System Builder uses this information to determine if updates have been made to any of the components generated by an earlier version of the software.

In order to make upgrading your microprocessor based system occur on your schedule, Lattice does not force components to be upgraded automatically. It is up to you to decide how and when to upgrade components.

Each time a platform is opened with a newer version of the LatticeMico32 System Builder software you are given the opportunity to update any out-of-date components. If you choose not to upgrade your LatticeMico32 platform then the information provided in the remainder of this document does not apply to you. Your LatticeMico32 powered design, and your accompanying firmware, do not require any modification.

However, if you do choose to upgrade your platform, carefully review the following information.

Figure 1. ispLEVER 8.1 SP1 and Diamond 1.1 Upgrade Flow



Platforms with the Master Passthru Component Included

The LatticeMico32 System Builder has, for several revisions, provided a Master Passthru component. The Master Passthru component permits 32-bit, 16-bit, or 8-bit Wishbone bus masters to gain access to the components attached to the LatticeMico32 Wishbone bus.

The possibility of 8-bit or 16-bit masters has an impact on the design decisions originally applied to the LatticeMico32 System Builder implementation. In order to save FPGA resources needed for address, and data lane decoding some LatticeMico32 System components were either DWORD aligned, or responded only to 32-bit wide data transactions. The first artifact makes using bus masters that have 16-bit or 8-bit data bit buses a little more challenging to work with. The second prevents 8-bit or 16-bit bus masters from interacting with certain components at all.

To permit 32-bit and 8-bit bus masters to operate correctly Lattice Semiconductor decided to update several components in the ispLEVER 8.1 SP1 and Diamond 1.1 software releases. To keep the consumption of FPGA resources down a decision was made not to support 16-bit bus masters. If you want to use a 16-bit bus master it must be connected to the Passthru as if it were an 8-bit master. The components that have been changed are:

- UART
- GPIO
- SPI Flash
- DMA

Any platforms you have created with these components will need to be modified in order to operate correctly. It is highly recommended that the data sheets for each of these components be read carefully to determine what changes you need to make to your system to make sure it operates after these components are upgraded.

Overview of UART Changes

The UART has received the following changes:

- Aligned on 8-bit boundaries. Earlier versions were 32-bit aligned.
- The data bus is only 8 bits wide. The byte lane select SEL input pins are ignored.
- All data accesses must be performed as BYTE transactions. Using HWORD/WORD transfers are ignored.
- C data structures representing the register layout and the component context have been changed to use *unsigned char* variables where appropriate.
- C device driver functions are updated to account for the change from 32-bit aligned registers to 8-bit aligned registers

After a design using the new UART is generated it is necessary to rebuild your FPGA bitstream.

You are also required to take action to update your firmware. The LatticeMico32 design team upgraded all of the UART device driver code. Designs implemented using the Lattice supplied UART driver code require the least effort to update. Simply recompile your code and deploy the program to your system.

Designs using custom device drivers and data structures require more effort. As mentioned in the list above the UART registers are now aligned on byte boundaries instead of WORD (i.e. 32-bit) boundaries. Some examples of changes that you may need to perform:

```
Example 1:
// UART controller is at 0x80000000
MicoUart_t *uart = (MicoUart_t *)0x80000000;
unsigned int rxtxData;

//Code Pre-8.1SP1 or Diamond 1.1
rxtxData = 0x55;  // BYTE constant promoted to WORD
uart->rxtx = rxtxData;

//Code Post-8.1SP1 or Diamond 1.1
unsigned char rxtxData;
rxtxData = 0x55;  // BYTE data
uart->rxtx = rxtxData;  // BYTE transaction
```

In Example 1 the base address of the MicoUart_t data structure remains the same. What you need to change is any *unsigned int* variables performing assignments to the *unsigned char* structure entries. The C compiler will emit warning messages for each of these assignments because the behavior of code generated is not defined.

```
Example 2:
// UART controller is at 0x80000000
volatile unsigned int * uartIER = (unsigned int *)0x80000004;
unsigned int ierData;

//Code Pre-8.1SP1 or Diamond 1.1
ierData = 0x55;  // BYTE constant promoted to WORD
*uartIER = ierData; // WORD transaction

//Code Post-8.1SP1 or Diamond 1.1
// BYTE aligned pointer
unsigned char * uartIER = (unsigned char *)0x80000001;
unsigned char ierData;
ierData = 0x55;  // BYTE data
*uartIER = ierData;  // BYTE transaction
```

In Example 2 two changes must be applied. The first is to change the pointer and data variables from *unsigned int* typed variables to *unsigned char* variables. The second change is to modify the address constant assigned to the specific UART register. Code, written prior to ispLEVER 8.1 SP1 that is left unmodified, will compile without errors or warnings, but the UART will not function.

Overview of GPIO Changes

The GPIO has received the following changes:

- Accesses are no longer forced to be 32-bit transactions.
- Each control register is byte-accessible.
- The definition of each byte in the register map is fully defined, not implied.

The GPIO component has had a few minor modifications made to it. The original GPIO component only responded to 32-bit memory transfers. If the SEL lines coming from the LatticeMico32 were not all active then the transaction to the GPIO registers were ignored. The new GPIO component enables access to each individual byte in the 32-bit register using 8-bit memory accesses.

Another change that has been made is to explicitly describe which port I/O pin is controlled by each byte within the register. The new definition provides an addressing order that is friendly to an 8-bit Wishbone master. This has a side effect for the LatticeMico32. The side effect is to swap the port I/O bytes relative to the LatticeMico32's preferred big endian format. This is illustrated in Figures 2 and 3.

Figure 2. Pre ispLEVER 8.1 SP1/Diamond 1.1

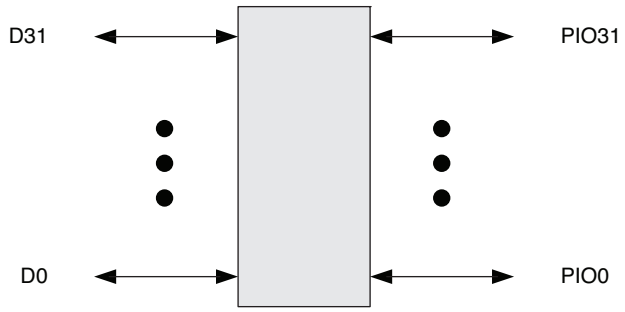
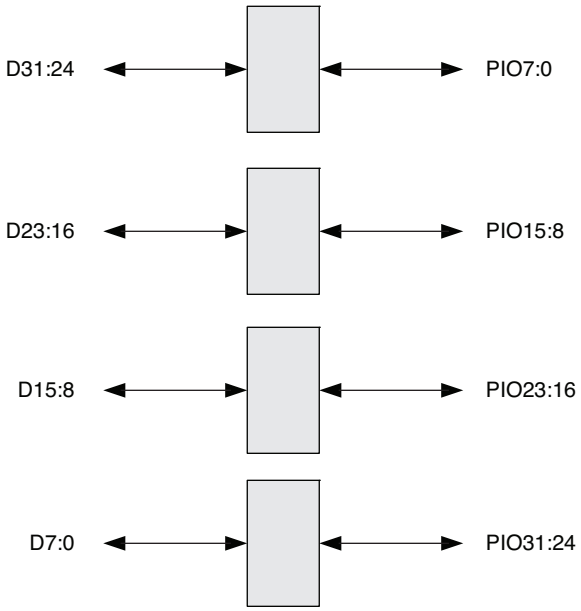


Figure 3. ispLEVER 8.1 SP1/Diamond 1.1 and Later

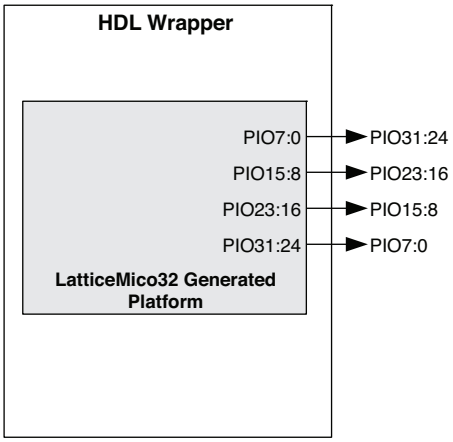


There are a couple of options open to system designers to handle the change in the GPIO's new behavior.

- The LatticeMico32 platform developer can place a top-level HDL wrapper around the Verilog code generated by LatticeMico32 System Builder.
- The C/C++ firmware developer can update the code that accesses the GPIO registers.

In general, it is expected the best solution is to wrap the LatticeMico32 Verilog. This solution is shown in Figure 4.

Figure 4. HDL Fix-up



Placing the Verilog source code generated by LatticeMico32 System Builder into a Verilog or VHDL wrapper that swaps the byte lanes permits existing C/C++ firmware to continue to run without modification. It is also the less error prone technique for dealing with the change to the GPIO component.

Overview of SPI Flash Changes

The SPI Flash Memory Controller has received the following changes:

- Accesses are no longer forced to be 32-bit transactions.
- Each control register is WORD/HWORD/BYTE accessible.

- The definition of each byte in the register map is fully defined, not implied.
  - Offset 0 within each register aligns to D7:0
  - Offset 3 within each register aligns to D31:24
- C macros are updated and the interfaces/functions documented

After a design using the new SPI Flash memory controller is generated it is necessary to rebuild your FPGA bitstream.

You are also required to take action to update your firmware. The LatticeMico32 design team upgraded the SPI Flash device driver code. Designs implemented using the Lattice supplied SPI Flash driver code require the least effort to update. Simply recompile your code and deploy the program to your system.

Code that either uses the Lattice-defined C macros or directly accesses the SPI Flash controller's register space needs to be inspected. The LatticeMico32 SPI Flash controller added C macros, but the macros were not published until the ispLEVER 8.1 SP1 and Diamond 1.1 product release. However, you may have discovered these macros by inspecting the controller's device driver code. If you implemented code using these macros you will be required to update your code as the names of all of the macros have changed. You must perform a global search and replace, as shown:

```
// Replace the SPI_CMD_ prefix
SPI_CMD_
// with
MICO_SPI_
```

Care must be taken during the replacement process due to the presence of numerous SPI_CMD_xxxxxx_OFFSET definitions. Some of the macros have also been updated to accept additional parameters. These macros will cause compile time errors making them easy to find and fix.

Direct memory accesses to the following registers do not need to be changed:

- Page Program Address Register
- Page Read Address Register
- Block Erase Type 1 Register
- Block Erase Type 2 Register
- Block Erase Type 3 Register
- User Command 0 Register
- User Command 1 Register
- User Return Data Register

Any code used to access the other registers in the memory map must be updated. The registers in the list above all behave correctly without modification because the big endian LatticeMico32 bus is mapped to place the MSB of the Wishbone bus into D7:0 of the SPI Flash controller's register set. The impact of this is that the remaining registers

are, in most instances, now located on the LatticeMico32's MSB, not the LSB. Giving specific examples will help clarify what must be changed:

```
Example 1:
// SPI Flash controller is at 0x80000000
volatile unsigned int *slow_read_cfg = (unsigned int *)0x80000180;

//Code Pre-8.1SP1 or Diamond 1.1
*slow_read_cfg = 0xAB;  // unsigned char promoted to int
                        // (i.e. 0xAB == 0x000000AB

//Code Post-8.1SP1 or Diamond 1.1
*slow_read_cfg = 0xAB000000;  // unsigned int places MSB
                              // of the LatticeMico32 into
                              // bits D7:0 of the DMA CR
```

```
Example 2:  Alternate solution
// SPI Flash controller is at 0x80000000
volatile unsigned int *slow_read_cfg = (unsigned int *)0x80000180;

//Code Pre-8.1SP1 or Diamond 1.1
*slow_read_cfg = 0xAB;  // unsigned char promoted to int
                        // (i.e. 0xAB == 0x000000AB

//Code Post-8.1SP1 or Diamond 1.1
volatile unsigned char *slow_read_cfg = (unsigned char *)0x80000180;
*slow_read_cfg = 0xAB;  // Notice the change of data type
```

Overview of DMA Changes

The DMA has received the following changes:

- Accesses are no longer forced to be 32-bit transactions.
- Each control register is byte-accessible.
- The definition of each byte in the register map is fully defined, not implied.
  - Offset 0 within each register aligns to D7:0
  - Offset 3 within each register aligns to D31:24

After a design using the new DMA controller is generated it is necessary to rebuild your FPGA bitstream.

You will need to analyze and may need to update your firmware. The LatticeMico32 design team upgraded the DMA controller's device driver code. Designs implemented using the Lattice supplied DMA controller driver code require the least effort to update. Simply recompile your code and deploy the program to your system.

Code that either uses the Lattice-defined DMA Controller Register Map Structure or directly accesses the DMA register space needs to be inspected.

Direct memory accesses to the following registers will not need to be changed:

- Source Address Register
- Destination Address Register
- Length Register

Specifically, any code used to access the Control Register or the Status Register must be updated. The Source Address, Destination Address, and the Length Register all behave correctly without modification because the big

endian LatticeMico32 bus is mapped to place the MSB of the Wishbone bus into D7:0 of the DMA controllers register set. The impact of this is the Control Register and the Status Register are now located on the LatticeMico32's MSB, not the LSB. Giving a specific examples will help clarify what must be changed:

```
Example 1:
// DMA controller is at 0x80000000
volatile unsigned int *control_register = (unsigned int *)0x8000000C;
volatile unsigned int *status_register = (unsigned int *)0x80000010;

//Code Pre-8.1SP1 or Diamond 1.1
*control_register = 0x54;  // unsigned char promoted to int
                           // (i.e. 0x54 == 0x00000054
*status_register = 0x2;    // 0x02 promoted to 0x00000002

//Code Post-8.1SP1 or Diamond 1.1
volatile unsigned char *status_register = (unsigned char *)0x80000010;
*control_register = 0x54000000;  // unsigned int places MSB
                                 // of the LatticeMico32 into
                                 // bits D7:0 of the DMA CR
*status_register = 0xAB;         // Notice the change of data type
```

```
Example 2:
// DMA Controller is at 0x80000000
MicoDMA_t *dma_controller = (MicoDMA_t *)0x80000000;
unsigned int status_reg;

// Working code Pre-8.1SP1 or Diamond 1.1
status_reg = 0x02;  // modify IE bit
dma_controller->status = status_reg;

// Code Post-8.1SP1 or Diamond 1.1
unsigned char status_reg;
status_reg = 0x02;  // modify IE bit
dma_controller->status = status_reg;
```

The code in Example 2 will encounter two problems after the DMA controller is updated in ispLEVER 8.1 SP1 or Diamond 1.1. The first is a compile time warning because the *status* variable in the MicoDMA_t data structure has changed from an *unsigned int* typed variable to an *unsigned char* typed variable. The assignment of an *unsigned int* to an *unsigned char* variable is undefined and the wrong data value may be written into the *status* register. In this instance the most practical solution is to change the *status_reg* variable in the bottom half of Example 2 to an *unsigned char* type, which resolves the compiler warning and guarantees correct behavior.

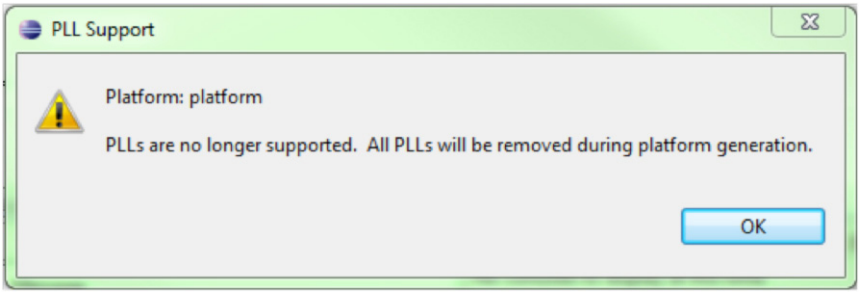
Platforms Using an Integral PLL

The LatticeMico32 System Builder software, from inception, provided the option of inserting a PLL into the microprocessor platform as a convenience. While this worked in many cases there are also a number of side-effects to this approach. Experience has shown that the side-effects outweigh the convenience of the automatic inclusion. Some of the observed side-effects include:

- The PLL output frequency generated from the base frequency may not be exactly what is desired.
- The PLL lock time may differ significantly from the deassertion time of the *reset_n* input signal.
- The PLL output frequency can only be used for LatticeMico32 platform components.
- PLL secondary clocks can't be implemented, nor accessed outside of the LatticeMico32 platform.

The issues enumerated could cause inconsistent or aberrant platform behavior. In order to improve the robustness of LatticeMico32 platforms the optional insertion of a PLL feature has been removed. Platforms that use the embedded PLL option prior to ispLEVER 8.1 SP1/Diamond 1.1 **cannot** be migrated without the removal of the PLL.

Platforms created with a PLL integral to the LatticeMico32 platform display the following dialog when they are opened:



In order to correct the removal of the PLL you need to take the following steps:

- Switch to the MSB perspective in the LatticeMico32 System Builder software.
- Navigate to the Platform Tools -> Properties menu, and change the Board Frequency to match the frequency that used to be generated by the integral PLL.
- Generate the new LatticeMico32 platform.
- Launch IPexpress and create a PLL module targeted to your FPGA family that replicates the desired output frequency based on the PLL's input frequency.
- Create a top level wrapper for the LatticeMico32 platform. Within the top level wrapper instantiate the PLL and the LatticeMico32 platform.
- Include a circuit to manage reset_n to the LatticeMico32 platform, see below.
- Use ispLEVER or Diamond to build the FPGA bitstream.

Figure 5 shows how the PLL was integrated into the LatticeMico32 platform source code. The PLL appeared in the top level just like any other LatticeMico32 System Builder component. The Wishbone *clk_i* input was automatically connected to the PLL and the PLL output clocked all of the platform components.

Figure 5. LatticeMico32 Platform Pre ispLEVER 8.1 SP1/Diamond 1.1

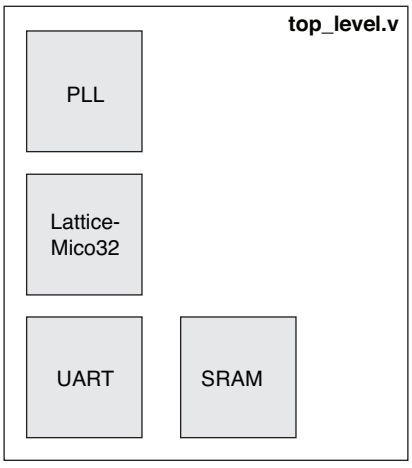
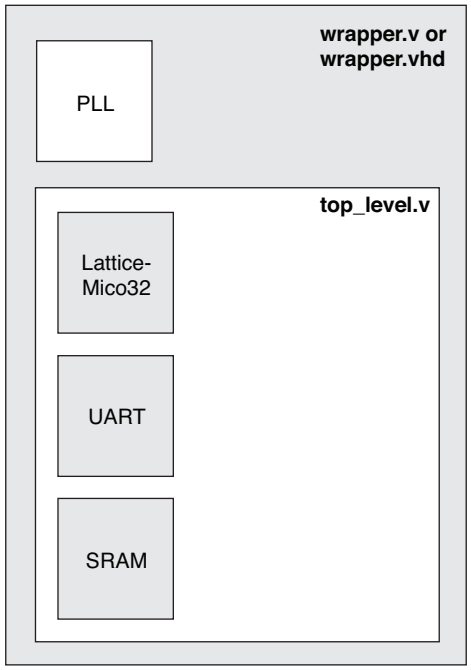


Figure 6 shows the structure of a LatticeMico32 platform after the PLL has been inserted according to the instructions provided above. The PLL is completely under your control. The clock source, feedback paths, and the various

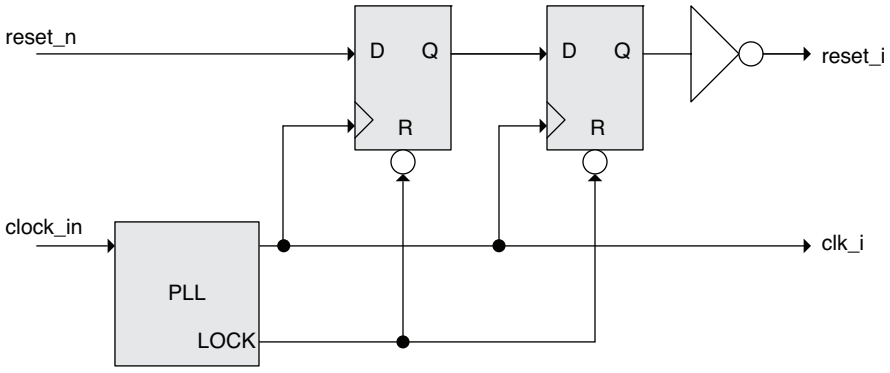
output frequencies the PLL can provide are all available for your use. The PLL clock outputs can not only be used by the LatticeMico32 platform, but also for other logic within the FPGA.

Figure 6. LatticeMico32 Platform Pre ispLEVER 8.1 SP1/Diamond 1.1



After restoring the PLL in the wrapper it is recommended that a circuit to manage *reset_n* be implemented. LatticeMico32 platforms clocked by a PLL operate best when *reset_n* is released after the PLL reports that it has locked to the input frequency. One possible solution is shown in Figure 7.

Figure 7. Example reset_i Controller



Conclusion

The LatticeMico32 development environment is undergoing some changes to make it more compatible with a wider range of platform designs. The changes are a direct result of feedback from LatticeMico32 designers like you. The information in this technical note describes the actions you must take to reduce the effort to transition from earlier versions of the LatticeMico32 System Builder development tools. The LatticeMico32 team recognizes the inconvenience caused by the change and will strive to limit the impact of such changes, but will continue to be responsive to the needs of system designers like yourself.

Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-503-268-8001 (Outside North America)
e-mail: techsupport@latticesemi.com
Internet: www.latticesemi.com

Revision History

Date	Version	Change Summary
October 2010	01.0	Initial release.
November 2010	01.1	Updated “Platforms with the Master Passthru Component Included” section.