ispGDX2 Pin Locking Recommendations

January 2004 Technical Note TN1022

Introduction

The ispGDX2™ is the next generation In-System Programmable Generic Digital Crosspoint (ispGDX®) family from Lattice Semiconductor. Based on the success of the first generation ispGDX family, the ispGDX2 continues to address a variety of system-level digital signal routing and interface requirements with its fast and flexible architecture. While the ispGDX family provides "bit oriented" architecture, the ispGDX2 family organizes the architecture into nibble-sized blocks. This block architecture greatly improves the efficiency of bus-oriented applications such as data/address bus multiplexing and programmable bus routing. Technical note TN1035, *ispGDX2 vs. ispGDX Architecture Comparison,* provides a detailed architecture comparison between the two device families.

A brief comparison of the ispGDX2 and ispGDX architectures is given here as a background for the subsequent pin locking discussion. It is expected that readers have a basic knowledge of both architectures, as provided in the data sheets. The ispGDX2 architecture is based on 4:1 multiplexers similar to the ispGDX architecture. The ispGDX2 offers enhanced flexibility compared with the ispGDX, both in terms of pin placement and total allowable control signals. The ispGDX2 device family has added sysIO™ banks in the architecture to incorporate many popular I/O standards. This technical note provides pin-locking recommendations in two areas: one associated with sysIO banks, the other with the routing architecture.

sysIO Banks

The ispGDX2 family supports eight sysIO banks per device. Each sysIO bank supports various I/O standards. In general, the I/O standards supported can be grouped into three categories:

1. Non-terminated, single-ended interfaces such as LVTTL, LVCMOS, PCI 3.3, PCI-X, and AGP-1X.
2. Terminated, single-ended interface standards including various versions of SSTL and HSTL, CTT, and GTL+. These require a V_{REF} signal, and a termination voltage (V_{TT}) at the system level.
3. Differential standards such as LVDS, bus-LVDS (BLVDS), and LVPECL.

Users must take V_{CCO} and V_{REF} into consideration when locking pins with different I/O standards. Since each sysIO bank shares common V_{CCO} and $V_{REF,}$ every output pin within a bank should conform to the selection of $V_{CCO,}$ and every input pin should conform to the selection of V_{REF} if it belongs to the second I/O standard category. For example, if V_{CCO} is set to 2.5V, only LVCMOS2.5, SSTL2, CTT2, GTL+ and differential standards outputs can be locked in the bank. Similarly, if V_{REF} is set to 1.5V, only SSTL3, CTT3 inputs, and non-terminated standard inputs can be locked in that bank. For more information on sysIO banks, refer to technical note TN1000, *sysIO Guidelines for Lattice Devices.* The ability to group appropriate V_{CCO} and V_{REF} and to check the violation of I/O standards in a sysIO bank is supported in the current version of Lattice's ispLEVER® development tool.

All dedicated inputs (e.g., global clock/clock enable, global MUX select, global output enable, and global reset pins) are conformed to LVTTL and LVCMOS standards by default. Table 1 lists other I/O standards supported by the various dedicated inputs.

Table 1. Standards Supported by Dedicated Pins

	LVCMOS	LVDS	All other ASIC I/Os
Global OE Pins	Yes	No	Yes[1]
Global MUX Select Pins	Yes	No	Yes[1]
ResetB	Yes	No	Yes[1]
Global Clock/Clock Enables	Yes	Yes	Yes[1]
IspJTAG™ Port	Yes[2]	No	No
TOE	Yes	No	No

1. No PCI clamp 2. LVCMOS as defined by the V_{CCJ} pin voltage

Routing Architecture and Pin Locking Recommendations

The ispGDX2 routing architecture strikes a delicate balance by significantly increasing routing flexibility and reducing routing fuse requirements and device sizes without compromising the performance of the device. The ispGDX2 architecture divides the routing resources into two parts, one for the data paths in the GDX blocks, and the other for the control arrays shared within the GDX block.

Data Paths in the GDX Block

The routing architecture divides the input pins into groups of 16. The 16 input pins are arranged such that if the pin location M = pin location of N modulo 16, then pins M and N cannot be routed into the data path of the same nibble of the GDX Block. For example, if signal A is locked at I/O 0 and signal B is locked at I/O 16, then they cannot be used in the same nibble within the GDX block. Each nibble contains four MUX and Register Blocks (MRBs). Each input of the 4:1 MUX of the MRB is sourced from the same group of input pins. In other words, each MRB within the nibble is able to use any one of the 16 input signals coming into the nibble.

Data Path Pin Locking Recommendations

Users who understand the data path routing schemes of the ispGDX2 architecture can effectively lock pins. The following are several recommendations for data path pin locking.

1. Use Automatic Pin Locking with Back Annotation

This is one of the easiest ways is to allow the design tool to lock the pins based on the design. The design tool takes various features of the architecture into consideration when assigning the pins and buses. When it recognizes a bus, the design tool groups the signals within a bus in the same GDX Block or sysIO bank. The automatic pin locking result provides the user with a good sense of how the pins should be locked, without requiring additional resources in the device. The automatic pin locking result can be back annotated after a successful design fit and utilized for subsequent fittings of the design. It is recommended to free all the pins for fitting if significant changes, such as bus routing, have been made in the design.

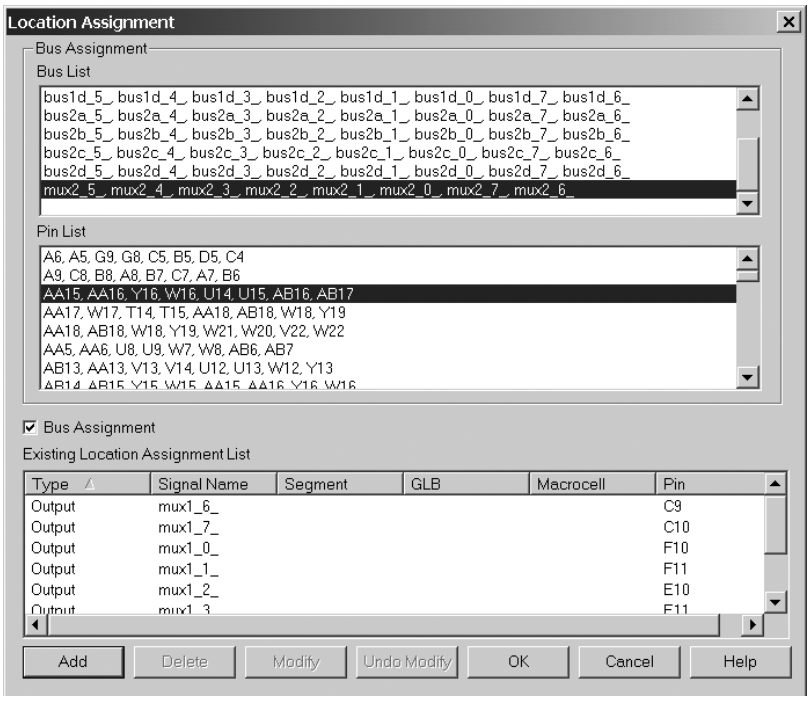
2. Use Bus Assignment Dialog Box

This dialog box can be found within the Location Assignment dialog box of the Constraint Editor. The Bus Assignment dialog box will appear when the “Bus Assignment” option is checked. The buses in the design will be shown as grouped signals and the fitter will provide a list of candidate pins for each bus. The candidate pins provided for a bus are grouped either within the same sysIO bank, or across multiple sysIO banks in a sequential manner, depending on the bus sizes.

The current version of ispLEVER is able to recognize input buses, output buses, and bi-directional buses in the Bus Assignment Dialog Box. Output signals can be extracted as a bus when all the signals have the same control signals, there are no V_{CCO} conflicts within the group, and the signals are not yet assigned. The control signals are the MUX selects, output enables, clock/clock enables, and set/reset signals. For input signals, ispLEVER will check that there are no V_{REF} conflicts in the group, that the pins are not yet assigned, and that all the signals only fanout to a single bus. The extraction of bi-directional buses will require the combination of the rules mentioned above. Therefore it is more challenging for the design tool to recognize or extract bi-directional buses in the Bus Assignment Dialog Box.

When using the Bus Assignment dialog box, it is recommended to lock the output buses first. This is a good starting point for locking the pins in the desired locations without worrying about the modulo-16 constraint. The advantage of using the dialog box for bus locking is that candidate pins are provided based on logic equation analysis results which, if used correctly, greatly reduces the possibility of routing conflicts in the design. The Logic Signal Connections table in the data sheet lists the locations of the candidate pins.

Figure 1. Constraint Editor Bus Assignment Dialog Box



3. Use Location Assignment Dialog Box

While the Bus Assignment dialog box is a powerful tool for bus locking, the candidate pins do not cover all the possible pin combinations in the device. When locking individual pins, or locking specific locations not covered by the candidate pin list, users can access the Location Assignment dialog box or the tabular form in the Constraint Editor. Either method requires a thorough understanding of the ispGDX2 architecture and the pin locations in the device.

As with bus locking in the Bus Assignment dialog box, it is recommended to lock the output signals first. When locking input or bi-directional signals, users must keep the “modulo-16” rule in mind. If two or more buses pass through the same nibble data path, the buses should not be locked in multiples of 16. For example, when busA and busB each have 16 bits and each corresponding pair is MUXed together for the same output, users should try to offset the arrangement of the buses by a certain number of pins. There are many ways to do this. The following list illustrates several methods.

- Lock the first 16-bit busA in I/O X to I/O Y in sequence, then lock the second 16-bit busB at I/O (Y+2) to I/O (Y+17). This will offset the bus location by one to avoid the modulo-16 pin locations.
- Lock the first 16-bit busA in I/O X to I/O Y in sequence, then lock the control signals at I/O (Y+1) to I/O Z. After that, lock the second 16-bit bus at I/O (Z+1) to I/O (Z+16). It is assumed that the total number of control signals between the two buses is not of modulo-16.
- Lock the first 16-bit busA in I/O X to I/O Y with MSB at I/O X, then lock the second 16-bit busB sequentially with the LSB at I/O (Y+1).
- In the source code, define the MSB and LSB with the opposite index. For example, in the VHDL source code, define busA to be (15 down to 0) and busB to be (0 to 15). Then lock busA and busB from bit 0 to bit 15 sequentially.

Users may devise other ideas to offset the pins. The goal is to make sure input or bi-directional pins going into the same nibble are not locked in multiples of 16. Consulting the signal name column of the Logic Signal Connections table in the data sheet is a good way to avoid this problem during input pin locking.

Control Array in the GDX Block

The Control Array for each GDX Block is separate and completely independent from the data paths in the GDX Block. Only the GRP inputs are common between the Control Array and the data path. This means that both the data paths and the Control Array derive their signal sets from the same GRP lines, thus removing boundary limitations. The Control Array in each GDX Block is designed to provide a balance between functionality and speed.

Control Array Pin Locking Recommendations

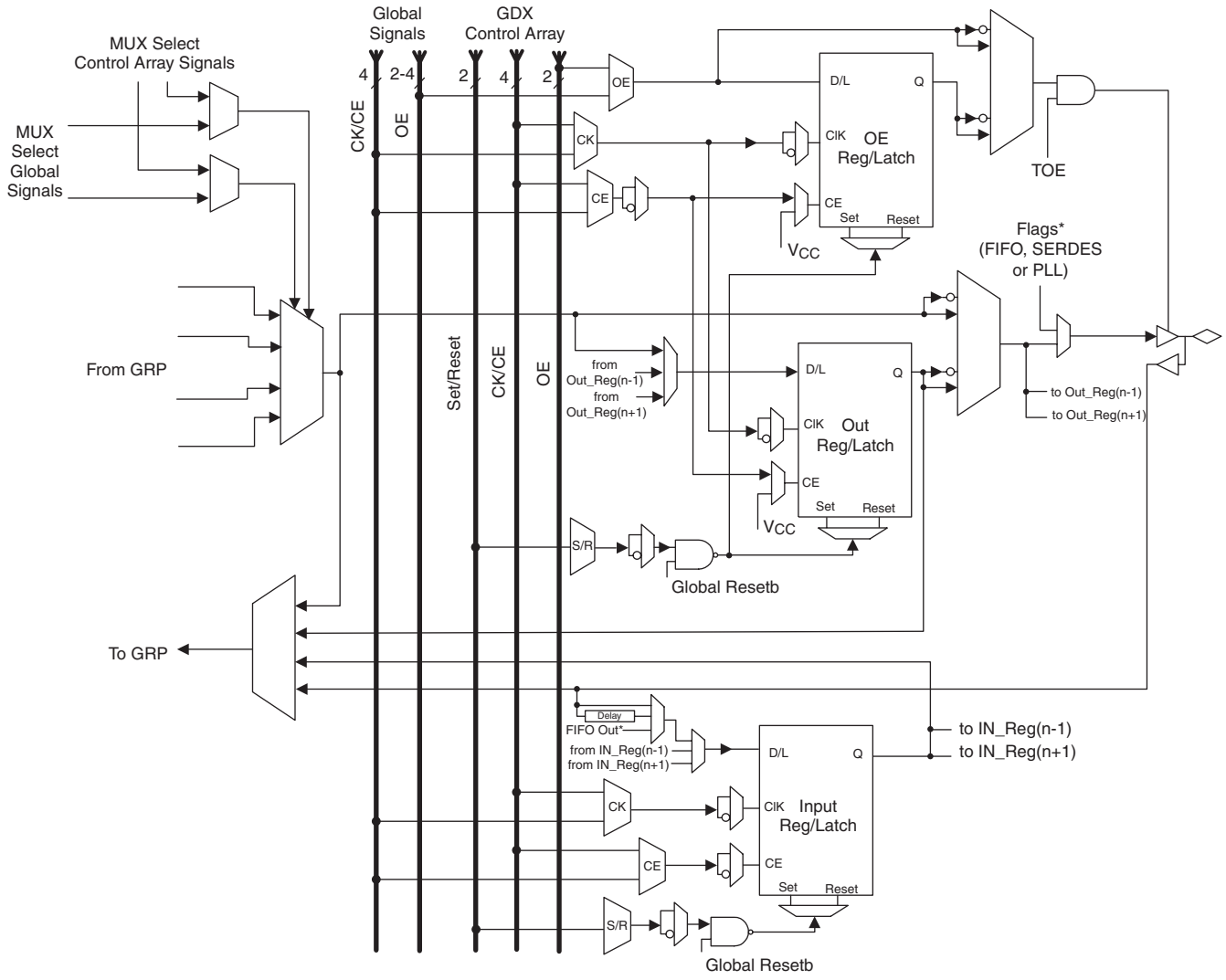
The following are several recommendations for using control signals in the same GDX Block.

1. Lock Pins Automatically with ispLEVER
Similar to the rule used for data paths, automatic pin locking provides the ispLEVER design tool great flexibility in pin placement to achieve the most effective resource utilization. For example, the design tool will utilize global control signals, if possible, to save routing resources in the device for other purposes.

2. Utilize Global Control Pins
Global control signals will free up routing resources in the device. The global signals available to each MRB increase the options for the control signals in each GDX Block. For example, the four global clock pins plus the four clock signals from the Control Array provide up to eight clock options in a GDX Block. Another example relates to the Global MUX Select signals. In the ispGDX2 architecture, the four MRBs in a nibble share two common MUX select signals coming from Control Array. Utilization of Global MUX Select signals allows each MRB in a nibble to have two options of MUX control, one set from the global pins and the other set from the local Control Array. This enables logic with different MUX control signals to be grouped into the same nibble of a GDX Block. As a result, appropriately assigned global control signals are able to make the device more flexible.

To optimize the performance of the ispGDX2-256 device, the architecture divides the Global MUX Select signals to control half of the sysIO Banks. Users should take this into consideration when assigning the Global MUX Select signals for the logic. Global MUX Select signals SEL0 and SEL1 can be used in sysIO banks 4 to 7, while SEL2 and SEL3 can be used for sysIO banks 0 to 3. Smaller devices with 64 I/Os have two Global MUX Select signals which are available to the entire device.

Figure 2. MRB of the ispGDX2 Family

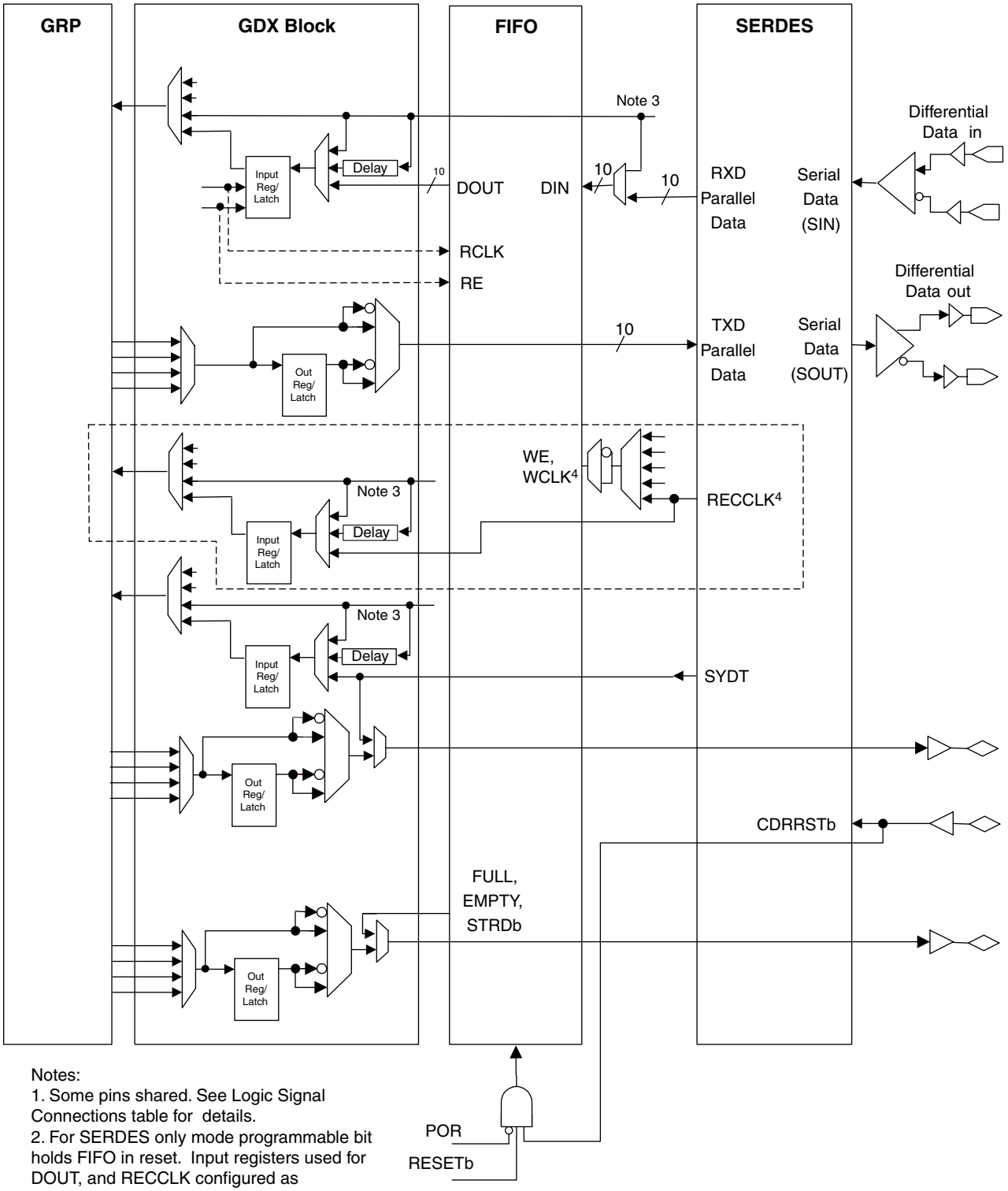


*Selected MRBs see Logic Signal Connection Table for details

Pin Locking Considerations for sysHSI™ and FIFO blocks

Every ispGDX2 device has multiple sysHSI blocks for high-speed serial interface. A sysHSI block supports up to two SERDES blocks. The data to and from SERDES blocks, and/or FIFOs, share the same data paths in the GDX blocks with the generic logic. The input path or the output path of an MRB cannot be used by generic logic if they are occupied by the SERDES parallel data or FIFO data. Figure 3 shows the concept of signal routing between a GDX block and the corresponding sysHSI block and FIFO. The Logic Signal Connection tables in the data sheet detail the multiple functions of each MRB and its corresponding I/O. The connection information of SERDES block 0A of a ispGDX2-64 device is extracted from the data sheet and is listed in Table 1 for reference.

Figure 3. Operation in SERDES Only and SERDES with FIFO Modes



Notes:
1. Some pins shared. See Logic Signal Connections table for details.
2. For SERDES only mode programmable bit holds FIFO in reset. Input registers used for DOUT, and RECCLK configured as latches and held in pass through.
3. From general I/O pins.
4. WCLK is generated from RECCLK or the Global Clocks. WE is generated from the Global Clocks.

The “SERDES Mode I/O Functions” column in the table shows the I/O cells, or pins, occupied by the SERDES function. The “SERDES to FIFO Core” column shows the internal connections between the SERDES channel and the FIFO in SERDES with FIFO mode, or between the SERDES block and the GDX block in SERDES only mode. In both modes, a part of MRB or the entire MRB will be occupied. A Receive parallel data bit (HSIxx_RXDy) will take up the input path of the MRB while the Transmit parallel data bit (HSIxx_TXDy) will take up the output path of the same MRB. For example, if the Receive channel of the SERDES block 0A 10B12B mode is used, it will not be possible to assign input signals to the pins associated with MRB 4 to MRB13 in GDX block 0A. By the same token, an utilized transmit channel of SERDES block 0A will prevent other signals going out to the pins through MRB 4 to MRB13. However, if only the Transmit channel is used in the SERDES block, the corresponding input paths of the MRBs are still open for input signals, except MRBs 12 and 13 because the pins are occupied by the serial outputs (SOUTP and SOUTN). This is because the input path and the output paths for a given MRB are independent of each other. Same rules can be applied to FIFO function. The “FIFO Mode I/O Function” column shows the I/O cells, or pins, occupied by FIFO inputs or outputs in FIFO mode. When FIFO_FULL signal is output to the pin, its associated output path of the MRB cannot be used for output but its associated input path is still available for internal logic. Just like the generic logic, the modulo-16 rule should be taken into consideration when SERDES and FIFO blocks are used in the design. Appendix A gives several pin-conflict examples when various functional blocks are used in the ispGDX2 device.

Table 2. Logic Signal Connections for SERDES Block 0A of ispGDX2-64 Device

Signal Name	sysIO Bank	LVDS Buffer		GDX Block	MRB	SERDES Mode I/O Functions	SERDES to FIFO Core	FIFO Mode I/O Function	100 fpBGA
		Polarity	Pair						
GOE0	-	-	-	-	-	-	-	-	H6
BK0_IO0/PLL_LOCK0	0	N	0	0A	0	-	-	FIFO0_FULL	J6
BK0_IO1	0	P	0	0A	1	HSI0A_CDRRSTb	-	FIFO0_FIFORSTb	K6
GND	0	-	-	-	-	-	-	-	GND
BK0_IO2	0	N	1	0A	2	HSI0A_SINN	HSI0A_RECCLK	-	G7
BK0_IO3	0	P	1	0A	3	HSI0A_SINP	-	-	H7
GND	0	-	-	-	-	-	-	-	GND
BK0_IO4/PLL_RST0	0	N	2	0A	4	-	HSI0A_RXD0	FIFO0_DIN0	K7
BK0_IO5	0	P	2	0A	5	-	HSI0A_RXD1	FIFO0_DIN1	K8
BK0_IO6/CLK_OUT0	0	N	3	0A	6	-	HSI0A_RXD2	FIFO0_DIN2	J8
BK0_IO7	0	P	3	0A	7	-	HSI0A_RXD3	FIFO0_DIN3	K9
GND	0	-	-	-	-	-	-	-	GND
TCK	-	-	-	-	-	-	-	-	J10
RESETb	-	-	-	-	-	-	-	-	J9
BK1_IO0/PLL_FBK0	0	P	4	0A	8	HSI0A_SYDT	HSI0A_RXD4	FIFO0_DIN4	H10
BK1_IO1	0	N	4	0A	9	-	HSI0A_RXD5	FIFO0_DIN5	H9
BK1_IO2	0	P	5	0A	10	-	HSI0A_RXD6	FIFO0_DIN6	H8
BK1_IO3/VREF(0,1)	0	N	5	0A	11	FIFO0_STRDb	HSI0A_RXD7	FIFO0_DIN7	G10
GND	0	-	-	-	-	-	-	-	GND
BK1_IO4	0	P	6	0A	12	HSI0A_SOUTP	HSI0A_RXD8	FIFO0_DIN8	G9
BK1_IO5	0	N	6	0A	13	HSI0A_SOUTN	HSI0A_RXD9	FIFO0_DIN9	G8
GND	0					-	-	-	GND
BK1_IO6	0	P	7	0A	14	SS_CLKIN1P	HSI0A_SYDT	-	F9
BK1_IO7	0	N	7	0A	15	SS_CLKIN1N	-	FIFO0_ EMPTY	F8

Special Considerations for Differential Signals

The modulo-16 rule still applies when using LVDS/BLVDS I/O standards. However, the constraint is more relaxed because of the LVDS/BLVDS positive (P) and negative (N) pair assignment across two consecutive banks. For an ispGDX2 device, any two consecutive sysIO banks have their LVDS/BLVDS P and N locations swapped. For example, sysIO bank 1 has P locations assigned to even-numbered I/O pins while sysIO bank 2 have P locations assigned to odd-numbered I/O pins. In the case of muxing two 16-bit LVDS/BLVDS input buses, users can assign each LVDS/BLVDS pair consecutively for the entire two buses without running into modulo-16 constraints.

Lattice macros may be used in the source code to identify whether the signals are LVDS/BLVDS input, output or bi-directional pins. The differential I/O types may also be assigned directly in the Constraint Editor of the ispLEVER tool. The macro or the Constraint Editor usages are the same for LVDS/BLVDS signals with and without sysHSI blocks. Please refer to technical note TN1020, *sysHSI Block Usage Guidelines,* for the definitions of LVDS/BLVDS macros. It is important to understand that each differential signal pair occupies two I/O pins. Therefore when the positive side of a LVDS pair is locked, the design tool will automatically reserve the corresponding negative pin of the pair.

Summary

The ispGDX2 family supports various I/O standards, including single-ended and differential, to offer great interfacing capability for aggregate bandwidth up to 38Gbps. Its multiplexer-based architecture provides efficient implementation of high-speed switching and routing functions. The integrated sysHSI blocks support standard serial link technologies and FIFO resources to reduce the components in the system and improve the overall performance. At the device level, the ispGDX2 architecture completely removes pin location requirements existing in the current ispGDX architecture. It also supports standard LVDS/BLVDS with and without the sysHSI blocks. Utilization of new design tool features and an understanding of the device architecture allow users to achieve high resource utilization and enjoy the flexibility of pin assignments. Design tool enhancements, such as GDX block assignment and reservation, will be included in future versions of the development tools to ensure an easy and straightforward pin locking experience.

Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-408-826-6002 (Outside North America)
e-mail: techsupport@latticesemi.com
Internet: www.latticesemi.com

Appendix A. Possible Routing Conflicts Due to Pin Locking

Routing conflicts happen when pin locations are assigned manually by the user. When pin-locking conflicts arise during fitting, the design tool will give out explicit error messages to explain the situation and list the pins involved in the conflicts. Most of the time it is related to modulo-16 restriction, occasionally it is related to the MUXSELECT signal, or SERDES/FIFO blocks. Following discussion addresses the majority pin-locking conflicts during design fitting. The pin location information is based on ispGDX2-64 device in the 100-ball fpBGA package.

Case 1: Modulo-16 Conflicts with Generic Logic

```
entity generic_logic is
port(          a, b           : in std_logic;
               sel            : in std_logic;
               q              : out std_logic
               );
               attribute loc: string;
               attribute loc of a: signal is “PG7”;
               attribute loc of b: signal is “PD8”;
end;

architecture behave of generic_logic is
begin
               q <= a when sel = ‘1’ else b;
end behave;
```

The two input pins are modulo-16 apart. Therefore they cannot be routed into the same output function. The workaround is to assign one of the input pins to a different location to avoid modulo-16 conflict.

Case 2: Modulo-16 Conflicts with SERDES Parallel Data

```
entity serdes is
               port ( sinp    : in std_logic;
                      sinn    : in std_logic;
                      refclk  : in std_logic;
                      rst     : in std_logic;
                      bufin   : in std_logic;
                      bufout  : out std_logic_vector(1 downto 0);
                      recclk  : out std_logic;
                      rxd_out: out std_logic_vector(9 downto 0)
                      );
               attribute loc  : string;
               attribute loc of sinp: signal is "PH7";
               attribute loc of refclk : signal is "PE4";
               attribute loc of rxd_out: signal is "PE8 PE9 PD8 PD9 PD10 PC9 PC10 PB10 PA9 PB8";
               attribute loc of bufin: signal is "PA6";
               attribute loc of bufout: signal is "PA8 PA7";
end;

architecture behave of serdes is
               signal rxd_int: std_logic_vector(9 downto 0);
               signal sin: std_logic;

               component LVDSIN
                      port(
                             P_IN : in    STD_LOGIC;
                             N_IN :in    STD_LOGIC;
                             O :out   STD_LOGIC );
                      end component;
```

```
              component CDRX_10B12B
                     generic( IN_FREQ  : string  := "50");
                     port(
                            SIN  : in    STD_LOGIC; REFCLK  : in    STD_LOGIC;
                            CDRRST : in    STD_LOGIC; RXD0  : out   STD_LOGIC;
                            RXD1: out   STD_LOGIC; RXD2 : out   STD_LOGIC;
                            RXD3: out   STD_LOGIC; RXD4 : out   STD_LOGIC;
                            RXD5: out   STD_LOGIC; RXD6 : out   STD_LOGIC;
                            RXD7: out   STD_LOGIC; RXD8 : out   STD_LOGIC;
                            RXD9: out   STD_LOGIC; RECCLK : out   STD_LOGIC;
                            CSLOCK : out   STD_LOGIC; CDRLOCK  : out   STD_LOGIC;
                            LOSS : out   STD_LOGIC; SYDT  : out   STD_LOGIC );
              end component;
              attribute in_freq: string;
              attribute in_freq of u1: label is "50";

begin

x1: lvdsin port map (sinp, sinn, sin);

u1 : CDRX_10B12B
              port map(
                     SIN => sin,          REFCLK => refclk,
                     CDRRST => not rst,  RXD0 => rxd_int(0),  RXD1 => rxd_int(1),
                     RXD2 => rxd_int(2),  RXD3 => rxd_int(3),  RXD4 => rxd_int(4),
                     RXD5 => rxd_int(5),  RXD6 => rxd_int(6),  RXD7 => rxd_int(7),
                     RXD8 => rxd_int(8),  RXD9 => rxd_int(9),  RECCLK => recclk,
                     CSLOCK => open,    CDRLOCK => open, LOSS => open,
                     SYDT => open );

rxd_out <= rxd_int;
bufout <= bufin & bufin;

end behave;
```

The pin-locking conflict comes from the RXD bit0 of SERDES block 0A and the input signal BUFIN signal. The RXD bit 0 occupies the input path of MRB 4 of SERDES block 0A. BUFIN comes in from pin A6, which is the MRB 4 of GDX block 1A. There is no routing conflict at this stage even though they are modulo-16 apart. The routing conflict happens when RXD bit 0 is output to RXD_OUT bit 0, and BUFIN is output to BUFOUT. The RXD_OUT bit 0 and BUFOUT are located in the same nibble (nibble 3 of GDX block 0B). The modulo-16 restriction prevents the two signals to be routed to the same nibble. The conflict can be resolved by either moving one of the inputs to a different location, or one of the outputs to a different nibble.

Case 3: SERDES Data and Control Signal Conflicts

```
entity serdes_data_ctrl is
              port ( sinp   : in std_logic;
                     sinn   : in std_logic;
                     refclk : in std_logic;
                     rst    : in std_logic;
                     enb    : in std_logic;
                     data_in: in std_logic_vector(9 downto 0);
                     recclk : out std_logic;
                     sydt_out: out std_logic;
                     rxd_out: out std_logic_vector(9 downto 0);
                     soutp  : out std_logic;
                     soutn  : out std_logic
                     );
              attribute loc  : string;
              attribute loc of sinp: signal is “PH7”;
```

```
              attribute loc of refclk : signal is “PE4”;
              attribute loc of enb: signal is “PK9”;
              attribute loc of sydt_out: signal is “PH10”;
end;

architecture behave of serdes_data_ctrl is
              signal rxd_int: std_logic_vector(9 downto 0);
              signal sin: std_logic;
              signal sout: std_logic;

              component LVDSIN
              port(
                     P_IN   :in    STD_ULOGIC;
                     N_IN    :in    STD_ULOGIC;
                     O      :out   STD_ULOGIC );
              end component;

              component LVDSOUT
              port(
                     I      :in    STD_ULOGIC;
                     P_OUT  :out   STD_ULOGIC;
                     N_OUT  :out   STD_ULOGIC );
              end component;

              component CDRX_10B12B
                     generic( IN_FREQ  : string  := “50”);
                     port(
                            SIN             : in    STD_LOGIC; REFCLK          : in    STD_LOGIC;
                            CDRRST      : in    STD_LOGIC; RXD0            : out   STD_LOGIC;
                            RXD1            : out   STD_LOGIC; RXD2            : out   STD_LOGIC;
                            RXD3            : out   STD_LOGIC; RXD4            : out   STD_LOGIC;
                            RXD5            : out   STD_LOGIC; RXD6            : out   STD_LOGIC;
                            RXD7            : out   STD_LOGIC; RXD8            : out   STD_LOGIC;
                            RXD9            : out   STD_LOGIC; RECCLK          : out   STD_LOGIC;
                            CSLOCK          : out   STD_LOGIC; CDRLOCK         : out   STD_LOGIC;
                            LOSS            : out   STD_LOGIC; SYDT            : out   STD_LOGIC );
              end component;
              attribute in_freq: string;
              attribute in_freq of u1: label is “50”;

              component TX_10B12B
                     generic( IN_FREQ  : string  := “50”);
                     port(
                            REFCLK   : in    STD_LOGIC; TXD0    : in    STD_LOGIC;
                            TXD1     : in    STD_LOGIC; TXD2    : in    STD_LOGIC;
                            TXD3     : in    STD_LOGIC; TXD4    : in    STD_LOGIC;
                            TXD5     : in    STD_LOGIC; TXD6    : in    STD_LOGIC;
                            TXD7     : in    STD_LOGIC; TXD8    : in    STD_LOGIC;
                            TXD9     : in    STD_LOGIC; SOUT    : out   STD_LOGIC;
                            CSLOCK   : out   STD_LOGIC );
              end component;
              attribute in_freq of u2: label is “50”;

begin

x1: lvdsin port map (sinp, sinn, sin);
x2: lvdsout port map (sout, soutp, soutn);

u1 : CDRX_10B12B
```

```
                port map(
                        SIN => sin, REFCLK => refclk, CDRRST => not rst,
                        RXD0 => rxd_int(0), RXD1 => rxd_int(1), RXD2 => rxd_int(2),
                        RXD3 => rxd_int(3), RXD4 => rxd_int(4), RXD5 => rxd_int(5),
                        RXD6 => rxd_int(6), RXD7 => rxd_int(7), RXD8 => rxd_int(8),
                        RXD9 => rxd_int(9), RECCLK => recclk, CSLOCK => open,
                        CDRLOCK => open, LOSS => open, SYDT => sydt_out );
rxd_out <= rxd_int when enb = '1' else (others => 'Z');

u2: TX_10B12B
                port map(
                        refclk => refclk, txd0 => data_in(0),  txd1 => data_in(1),
                        txd2 => data_in(2),  txd3 => data_in(3), txd4 => data_in(4),
                        txd5 => data_in(5), txd6 => data_in(6), txd7 => data_in(7),
                        txd8 => data_in(8),  txd9 => data_in(9), sout => sout,
                        cslock => open );

end behave;
```

There are two pin-locking conflicts in this design. The input signal, ENB, is assigned to MRB 7 (pin K9) of GDX block 0A. The input path of the same MRB is being used by the RXD3 of the Receive SERDES channel because the SINP is locked to HSI0A_SINP location. The two signals are trying to be routed into GRP through the input path of the same MRB therefore the design tool issues an error. The SYDT output is locked to MRB 8 of SERDES block 0A. This MRB's output path is used for Transmit data TXD4. Again, the design tool issues an error because of the routing path conflict. The first conflict can be workaround by assigning ENB to a different GDX block. The second conflict can be resolved by assigning SYDT to a different pin to allow the SYDT signal to be routed through GRP.

Case 4 : Two Paths of the SYDT Signal

```
entity sydt_paths is
                port ( sinp   : in std_logic;
                        sinn   : in std_logic;
                        refclk : in std_logic;
                        rst    : in std_logic;
                        cdrrst : in std_logic;
                        flags  : in std_logic_vector(1 downto 0);
                        ready  : out std_logic;
                        rxd_out: out std_logic_vector(9 downto 0)
                        );
                attribute loc  : string;
                attribute loc of flags : signal is "PF9 PF8";
                attribute loc of sinp: signal is "PH7";
                attribute loc of refclk : signal is "PF7";
end;

architecture behave of sydt_paths is
                signal rxd_int: std_logic_vector(9 downto 0);
                signal sin: std_logic;
                signal sydt: std_logic;
                signal recclk: std_logic;

                component LVDSIN
                        port(
                                P_IN : in    STD_LOGIC;
                                N_IN     : in    STD_LOGIC;
                                O        :out   STD_LOGIC );
                end component;
```

```
                component CDRX_10B12B
                generic( IN_FREQ  : string  := "50");
                        port(
                        SIN   : in    STD_LOGIC; REFCLK  : in    STD_LOGIC;
                        CDRRST   : in    STD_LOGIC; RXD0    : out   STD_LOGIC;
                        RXD1     : out   STD_LOGIC;   RXD2    : out   STD_LOGIC;
                        RXD3     : out   STD_LOGIC;   RXD4    : out   STD_LOGIC;
                        RXD5     : out   STD_LOGIC;   RXD6     : out   STD_LOGIC;
                        RXD7     : out   STD_LOGIC;   RXD8     : out   STD_LOGIC;
                        RXD9     : out   STD_LOGIC;   RECCLK   : out   STD_LOGIC;
                        CSLOCK  : out   STD_LOGIC; CDRLOCK : out   STD_LOGIC;
                        LOSS   : out   STD_LOGIC;    SYDT      : out   STD_LOGIC );
                end component;
                attribute in_freq: string;
                attribute in_freq of u1: label is "50";

begin

x1: lvdsin port map (sinp, sinn, sin);

u1 : CDRX_10B12B
                port map(
                SIN => sin, REFCLK => refclk, CDRRST => not cdrrst,
                RXD0 => rxd_int(0), RXD1 => rxd_int(1), RXD2 => rxd_int(2),
                RXD3 => rxd_int(3), RXD4 => rxd_int(4), RXD5 => rxd_int(5),
                RXD6 => rxd_int(6), RXD7 => rxd_int(7), RXD8 => rxd_int(8),
                RXD9 => rxd_int(9), RECCLK => recclk, CSLOCK => open,
                CDRLOCK => open, LOSS => open, SYDT => sydt );

rxd_out <= rxd_int;

u2: process(recclk, rst)
                begin
                if (rst = '0') then
                                ready <= '0';
                elsif falling_edge(recclk) then
                                if sydt = '1' then
                                ready <= flags(0);
                                else
                                ready <= flags(1);
                        end if;
                end if;
                end process u2;

end behave;
```

The SYDT signal generated by the SERDES block has two routing paths. It will take up the output path of MRB 8 of SERDES block 0A when it is assigned to pin H10. If this signal is used internally or assigned to a pin other than the H10, it will be routed to GRP through the input path of MRB 14. In this case, SYDT is used internally and therefore will use the MRB 14 (pin F9) of the SERDES block 0A. Since signal FLAGS bit1 is assigned to F9, there is a routing conflict because both SYDT and FLAGS bit 1 are trying to use the input path of the same MRB. The workaround is to assign FLAGS bit 1 to another pin location.

Case 5: The Reserved CDRRST Signal

```
entity serdes_data_ctrl is
              port ( refclk : in std_logic;
                     sigA   : in std_logic;
                     data_in: in std_logic_vector(9 downto 0);
                     lvdsAp : out std_logic;
                     lvdsAn : out std_logic;
                     soutp  : out std_logic;
                     soutn  : out std_logic
                     );
              attribute loc  : string;
              attribute loc of soutp: signal is “PG9”;
              attribute loc of refclk : signal is “PE4”;
              attribute loc of sigA: signal is “PK6”;
end;

architecture behave of serdes_data_ctrl is
              signal sout: std_logic;

              component LVDSOUT
                     port(
                            I       :in    STD_LOGIC;
                            P_OUT   :out   STD_LOGIC;
                            N_OUT   :out   STD_LOGIC );
                     end component;

              component TX_8B10B
                     generic( IN_FREQ  : string  := “50”);
                      port(
                            REFCLK   : in    STD_LOGIC;
                            TXD0     : in    STD_LOGIC; TXD1     : in    STD_LOGIC;
                            TXD2     : in    STD_LOGIC; TXD3     : in    STD_LOGIC;
                            TXD4     : in    STD_LOGIC; TXD5     : in    STD_LOGIC;
                            TXD6     : in    STD_LOGIC; TXD7     : in    STD_LOGIC;
                            TXD8     : in    STD_LOGIC; TXD9     : in   STD_LOGIC;
                            SOUT    : out   STD_LOGIC; CSLOCK   : out   STD_LOGIC );
              end component;
              attribute in_freq: string;
              attribute in_freq of u1: label is “50”;

begin

x1: lvdsout port map (sout, soutp, soutn);
x2: lvdsout port map (sigA, lvdsAp, lvdsAn);

u1: TX_8B10B
              port map(
                     refclk => refclk,
                     txd0 => data_in(0), txd1 => data_in(1), txd2 => data_in(2),
                     txd3 => data_in(3), txd4 => data_in(4), txd5 => data_in(5),
                     txd6 => data_in(6), txd7 => data_in(7), txd8 => data_in(8),
                     txd9 => data_in(9), sout => sout, cslock => open );

end behave;
```

The CDRRST pin has effect on the entire SERDES block. Consequently the pin is occupied whenever the SERDES block is utilized, whether it is used for Transmit only, or Receive only, or for the full-duplex function. This design uses the Transmit channel of the SERDES block with an input signal (SigA) assigned to the CDRRST pin (K6). This will cause an error during fitting. The simple workaround is to assign the SigA signal to a different pin.

Case 6: Different MUXSEL Signal in the Same Nibble

```
entity muxsel_ctrl is
              port ( rst    : in std_logic;
                     selA   : in std_logic_vector(1 downto 0);
                     sigA   : in std_logic_vector(3 downto 0);
                     selB   : in std_logic;
                     outa   : out std_logic;
                     outb   : out std_logic
                     );
              attribute loc  : string;
              attribute loc of outa: signal is “PG9”;
              attribute loc of outb: signal is “PF8”;
end;

architecture behave of muxsel_ctrl is
              signal cntl: std_logic_vector(2 downto 0);
begin
              cntl <= rst & selA;

              outa <= sigA(0) when cntl = “100” else
                     sigA(1) when cntl = “101” else
                     sigA(2) when cntl = “110” else
                     sigA(3) when cntl = “111” else
                     ‘0’;

              outb <= selA(0) when selB = ‘0’ else selA(1);

end behave;
```

The conflict comes from the MUXSEL signals for the two output pins. Signal OUTA is an 8:1 MUX, and the final stage MUXSEL is signal RST. Signal OUTB is a 2:1 MUX, and the MUXSEL is signal SELB. The two signals cannot be assigned to the same nibble because they do not share the same MUXSEL signals. There are several workarounds for this situation. The easiest one is to assign the two output signals to two different nibbles. The conflict can also be resolved by moving one pair of select signals to global MUXSEL pins. The second solution does not work for this particular design because the MUXSEL signals are not a pair (RST for OUTA, and SELB for OUTB).