Using ispGDX™ Generic Digital Crosspoint Devices

Introduction

The ispGDX Family from Lattice Semiconductor is a powerful new series of devices that can implement many digital crosspoint functions with minimum effort due to their in-system programmability (ISP™). An ideal solution for applications such as multiplexing, bit inversion, multiprocessor interfaces and signal routing, the ispGDX can also emulate many other common logic functions.

Each pin of an ispGDX device is associated with an I/O cell containing a 4-to-1 MUX, a D flip-flop that can be bypassed, a tri-state buffer and a boundary scan cell. The output can be fed back to any other cell of the device. Each data input of the 4-to-1 MUX can come from a quarter of the device I/O pins (one from the "A" pin bank, one from the "B" pin bank, etc.). In addition, each MUX select (MUXsel) control line can come from a quarter of the device I/O pins. The I/O pins can be used for data or control signal inputs: which functionality is used is specified by the application. Figure 2 shows the architecture an I/O cell.

This architecture provides the connectivity needed for crosspoint applications, as well as logic functions with a slightly different interface. If the logical inputs are input on the two MUX select lines and the data inputs are fixed high and low accordingly, any two input combinatorial logic functions can be emulated. Figure 3 shows an example of this, an ispGDX MUX emulating an XOR function.

Figure 1. ispGDX 160 Device Pinout

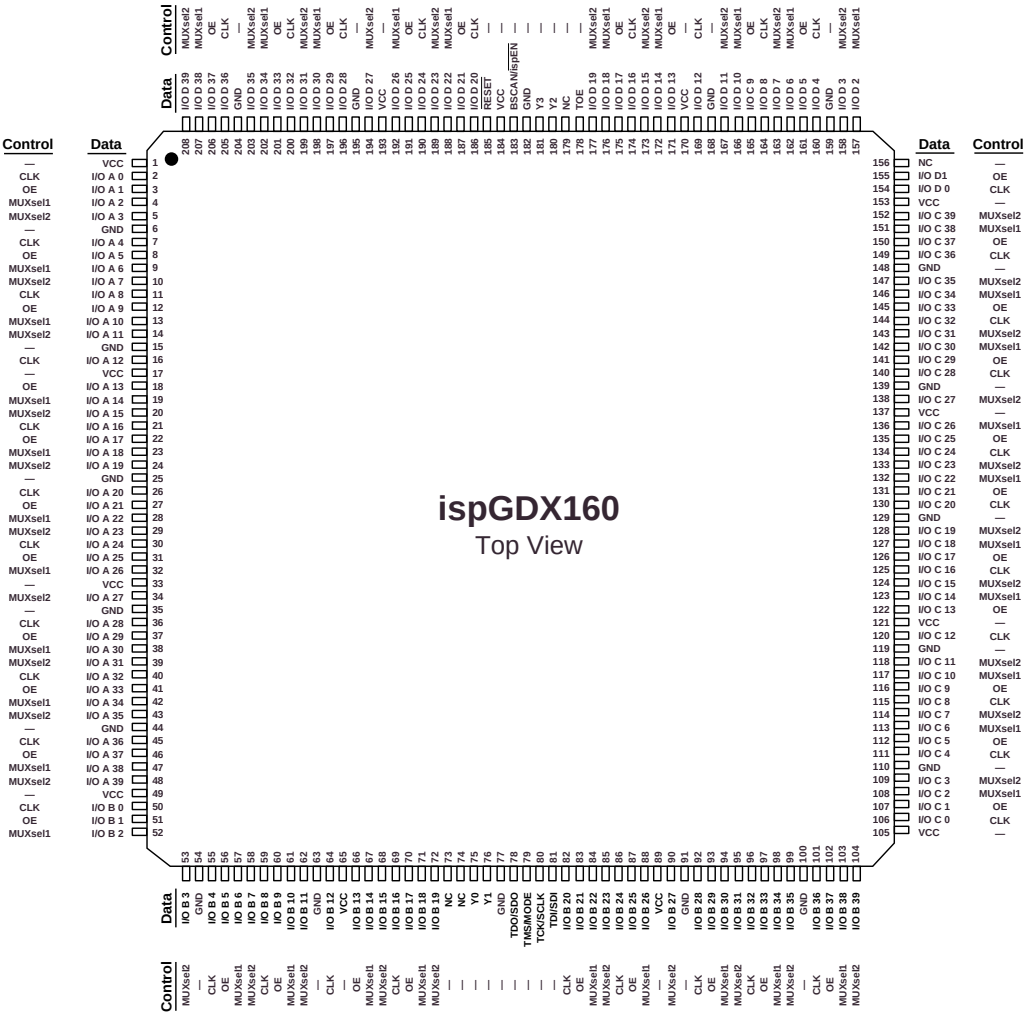
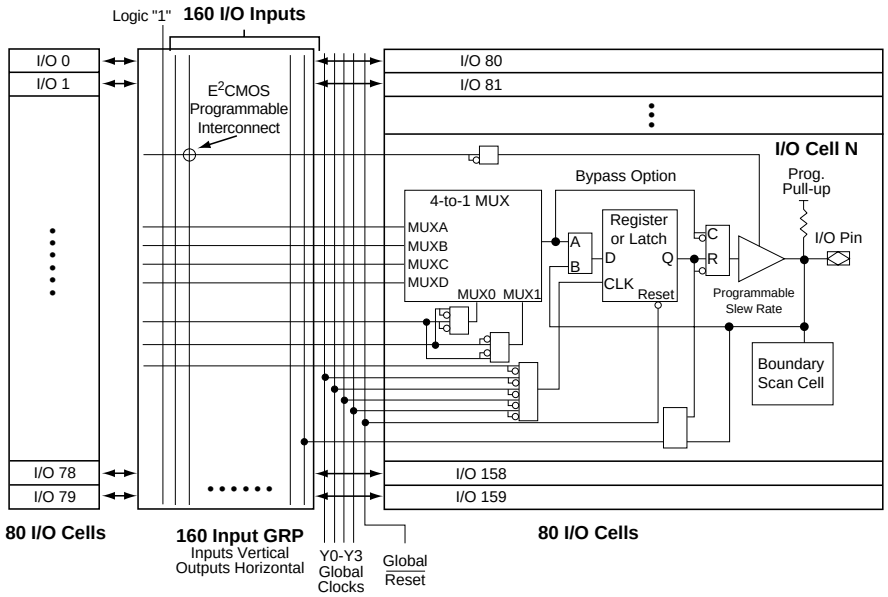


Figure 2. I/O Cell and GRP, ispGDX 160



Two or more I/O cells can be cascaded to implement logic functions with more than one input. This combination of crosspoint functions and logical emulation allows the ispGDX device to emulate a wide number of standard TTL functions, usually with much higher density. Table 1 shows some of the TTL functions the ispGDX Family can emulate efficiently. The ispGDX device can emulate more complicated TTL functions, such as parity generators and comparators, but it works best for functions like the ones described below that can easily be translated to its architecture.

Three types of applications are best served by the ispGDX Family: Programmable Random Signal Interconnect (PRSI), Programmable Data Path (PDP) and Programmable Switch Replacement (PSR). This application note describe each of these applications, emphasizing data path functions used to be implemented with TTL. In addition, a summary of the powerful ispGDX Development System is provided.

The ispGDX Development System

Before the application types are outlined, it is necessary to explain the structure of the ispGDX Development System and its language. The ispGDX Development System has a unique software language syntax developed specifically for the ispGDX Family of devices, thus requiring no external libraries or compilers.

Figure 4 shows a typical screen of the ispGDX Development System. The top part of the screen contains the text

Figure 3. MUX XOR Emulation

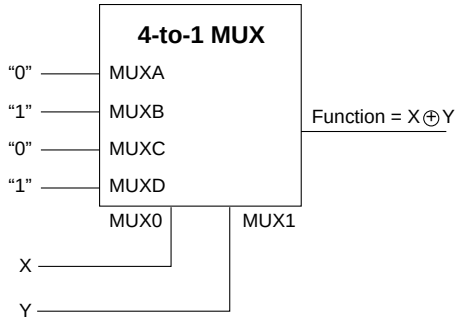
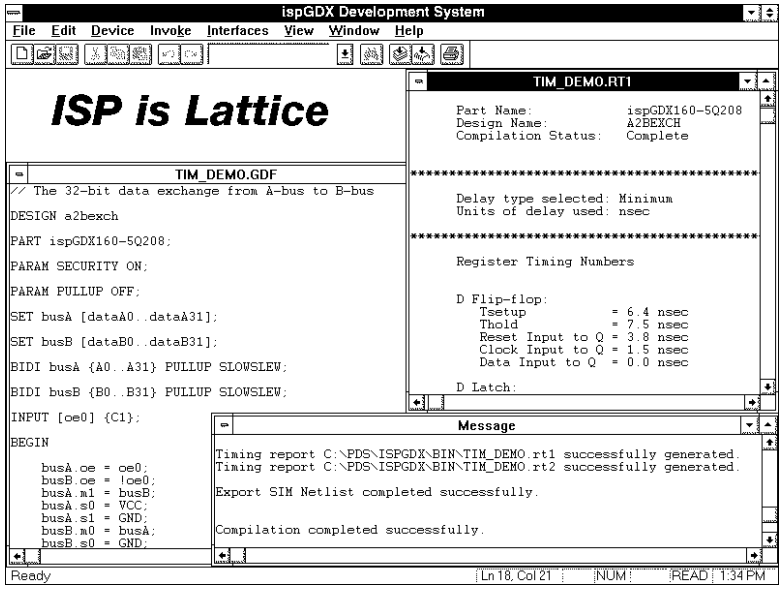


Table 1. ispGDX TTL Functions

FUNCTION	TTL DEVICE NUMBERS
Receivers, Drivers and Buffers	37, 125, 240, 241, 244, 540, 541, 1004, 1034
Decoders and Encoders (DEMUX, MUX)	138, 139, 151, 157, 158, 251, 257, 258, 9321
Flip-Flops and Latches	113, 174, 175, 373, 374, 533, 873
Transceivers (Registered/ Non-Registered)	623, 640, 645, 2645, 5620, 29863
Shift Registers	194, 195, 299, 93L00

GDX TTL Functions

Figure 4. ispGDX Development System User Environment



editor used for entering design syntax, and the bottom part shows messages during the compilation process. For details on compilation and other features, refer to the ispGDX Development System User Manual.

All files have the same layout containing four sections in this order: device, set/constant assignment, pin assignment and connections. Before the first section, the first line of the file simply names the file following the `DESIGN` keyword. The device section specifies which member of the ispGDX Family the file is for and sets global device parameters such as slew rate and security features. The set/constant section declares and assigns single signals to bus signals to reduce programming complexity, and allows the declaration of constant values and strings.

The pin assignment section describes each signal in the file by classifying it as input, output or bi-directional, assigning it to a pin and designating parameters on a pin-by-pin basis. If signals are not assigned to the appropriate pin type, the design will not compile. For example, a signal that is to be the data input to the first MUX input of an I/O cell (MUXA in Figure 2) must be assigned to a pin on bank "A."

Following the `BEGIN` keyword is the connection section. In this section, the connections to the I/O cell of each signal are defined and assigned. Dot extensions are used to describe various parts of the I/O cell and assign signals to it. The presence or absence of particular dot extensions as well as the polarity of the signals define the path within the I/O cell that the signals take. Table 2 defines these dot extensions.

Table 2. Dot Extensions

DOT EXTENSION	DEFINITION
.oe	Output Enable for the Tri-State Output Buffer
.clk	Common Clock for all Flip Flops
.en	Common Enable for all Flip Flops
.m0	Data Input from Pin Bank A
.m1	Data Input from Pin Bank B
.m2	Data Input from Pin Bank C
.m3	Data Input from Pin Bank D
.s0	MUX select from the MUXsel Pin Bank
.s1	MUX select from the MUXsel Pin Bank

GDX Dot Extensions

Table 3. Data Selection Codes

MUXsel1	MUXsel2	DATA BANK SELECTED
0	0	A (m0)
0	1	B (m1)
1	1	C (m2)
1	0	D (m3)

GDX Data Selection Codes

Parameters such as slew rate and pull-up resistors can be assigned globally or on a pin-by-pin basis. To assign these parameters globally, the code in the device section would be `PARAM PULL-UP ON` or `PARAM SLOWSLEW OFF`. These parameters can be specified individually in the pin assignment section at the end of any pin assign-

ment line. Specifying the parameters is done by adding the word at the end of the line to turn the parameter on for that signal; for example, ending the line with the keyword SLOWSLEW turns the slowslew option on for that pin (it should be a bi-directional or output pin).

Applications

PRSI

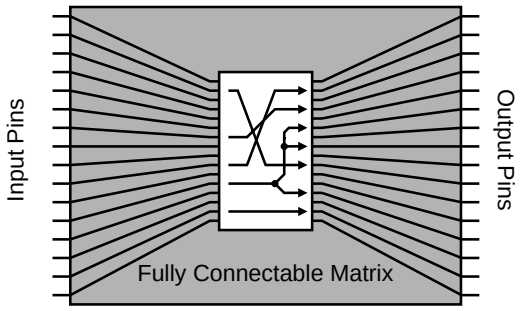
The first application type that the ispGDX Family can implement is Programmable Random Signal Interconnect or PRSI. PRSI refers to applications that arbitrarily swap signals between chips. PRSI provides static pin-to-pin connections on a PCB board level. The programmable logic capability of the ispGDX device allows switching between several hardware configurations simply by reprogramming.

Internal routing of signals from I/O cell to I/O cell is done by the E^2 CMOS® Global Routing Pool (GRP). Using the GRP, any input pin can be connected to any output pin in a static manner once the device is programmed. For example, an ispGDX 160 device can be used to construct a 64-by-64 crosspoint which can be reprogrammed in-system in seconds. Figure 5 shows the general architecture of this crosspoint.

In addition, a 16-by-16 bit high-speed crosspoint switch can be constructed. This switch allows connection between any input and output pin on one device with fast, dynamic signal switching; it utilizes every pin on an ispGDX 160. A 32-by-32 bit crosspoint can be similarly implemented using four ispGDX 160 devices. In the 16-by-16 switch, bits from the input are selected in two stages since the multiplexers select from four inputs. Each stage has a delay of 5ns. The inputs consist of the input data and the complete selection vectors needed to produce the appropriate crosspoint solution.

The syntax for this switch, shown in Listing 1, is an excellent example of using buses to assign many single bit signals using only a single connection statement. The X0 through X15 inputs are assigned to four MUXes, MUX0 through MUX4, which can then be selected by the SEL1 and SEL2 inputs. This then goes to ZOUT, the second stage MUX (and switch output), in which the selection of the input is finished by the SEL3 and SEL4 inputs. To select a particular input for a particular output, the appropriate selection vector must be entered on the appropriate input pins (for example, the first four selection bits assigned to the A4, B4, C4, and D4 pins choose which input goes to the first output Z0).

Figure 5. ispGDX Programmable Interconnect



Using an ispGDX solution provides many benefits over more conventional approaches such as in-system programmability, substantial size reductions and predictable delays among many others.

PDP

The next application type, Programmable Data Path (PDP), involves processing a stream of data to accomplish one of many possible functions. PDP includes the integration of the TTL interface functions mentioned in the Introduction. Some TTL functions are trivial to translate to a ispGDX device. Buffers and inverters are specified in syntax by simple assignment (e.g. X=Y or X=!Y). Using set notation allows buffers of longer lengths to be designated using the same notation as single signal assignment. Also, registered signals can be specified simply by assigning a clock signal (e.g. X.clk = clkin) that has been pin assigned as an input to the clock pin. The ispGDX Family of devices support much higher PCB densities than simple TTL parts.

Several examples of PDP applications, one for each type of TTL function mentioned in the introduction, will help illustrate the principles discussed so far. These examples implement an octal buffer/line driver, octal MUXes, an 8-bit register, an octal 3-state bus transceiver, and a 4-bit parallel shift register. The discussion of the parallel shift register also includes a detailed description of the software code needed in its implementation.

Octal Buffer / Line Driver

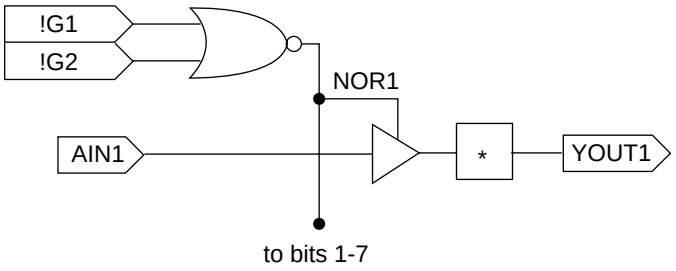
An octal buffer is a set of eight inverting tri-state buffers with a common output enable signal to drive the device into high impedance mode. Since each I/O cell of the ispGDX contains a tri-state buffer, this function can easily fit onto an ispGDX device.

Listing 1. 16 x 16 Switch Syntax (Partial)

```
SET MUX0 [M0Z0..M0Z15];
SET MUX1 [M1Z0..M1Z15];
SET MUX2 [M2Z0..M2Z15];
SET MUX3 [M3Z0..M3Z15];
SET SEL1 [S1Z0..S1Z15];
SET SEL2 [S2Z0..S2Z15];
SET SEL3 [S3Z0..S3Z15];
SET SEL4 [S4Z0..S4Z15];
SET ZOUT [Z0..Z15];

INPUT [X0,X4,X8 ,X12] {A0..A3};
INPUT [X1,X5,X9 ,X13] {B0..B3};
INPUT [X2,X6,X10,X14] {C0..C3};
INPUT [X3,X7,X11,X15] {D0..D3};
OUTPUT [Z0..Z3]   {A36..A39};
OUTPUT [Z4..Z7]   {B36..B39};
OUTPUT [Z8..Z11]  {C36..C39};
OUTPUT [Z12..Z15] {D36..D39};
BIDI MUX0 {A4,A5,A8,A9,A12,A13,A16,A17,A20,A21,A24,A25,A28,A29,A32,A33};
BIDI MUX1 {B4,B5,B8,B9,B12,B13,B16,B17,B20,B21,B24,B25,B28,B29,B32,B33};
BIDI MUX2 {C4,C5,C8,C9,C12,C13,C16,C17,C20,C21,C24,C25,C28,C29,C32,C33};
BIDI MUX3 {D4,D5,D8,D9,D12,D13,D16,D17,D20,D21,D24,D25,D28,D29,D32,D33};
INPUT SEL1 {A6,A10,A14,A18,A22,A26,A30,A34,B6,B10,B14,B18,B22,B26,B30,B34};
INPUT SEL2 {A7,A11,A15,A19,A23,A27,A31,A35,B7,B11,B15,B19,B23,B27,B31,B35};
INPUT SEL3 {C6,C10,C14,C18,C22,C26,C30,C34,D6,D10,D14,D18,D22,D26,D30,D34};
INPUT SEL4 {C7,C11,C15,C19,C23,C27,C31,C35,D7,D11,D15,D19,D23,D27,D31,D35};

BEGIN
       MUX0.m0 = X0;
       MUX0.m1 = X1;
       MUX0.m2 = X2;
       MUX0.m3 = X3;
       MUX0.s0 = SEL1;
       MUX0.s1 = SEL2;
       MUX0.oe = VCC;

       ZOUT.m0 = MUX0;
       ZOUT.m1 = MUX1;
       ZOUT.m2 = MUX2;
       ZOUT.m3 = MUX3;
       ZOUT.s0 = SEL3;
       ZOUT.s1 = SEL4;
       ZOUT.oe = VCC;
END
```

For this example, a TTL ‘540 was emulated using the ispGDX Development System (see schematic and code below). The output enable is driven by two active low inputs, !G1 and !G2. These two signals are then NORed together so that either one going high will drive the buffers into tri-state mode. This requires logical emulation, which is done in the code by the NOR1 signal. This buffer is an inverting buffer, so the input bus is negated. The inversion is easily accomplished by the ispGDX Development System without using any extra I/O cells. NOR1 is then assigned to be the input to the output enable of the output buffer (YOUT.oe), which allows the AIN bus to pass through. The individual AIN signals are assigned to a bus name by the SET statement, which allows the connections to be made on a bus level.

Figure 7. Octal Buffer Configuration



Listing 2. Octal Buffer Syntax

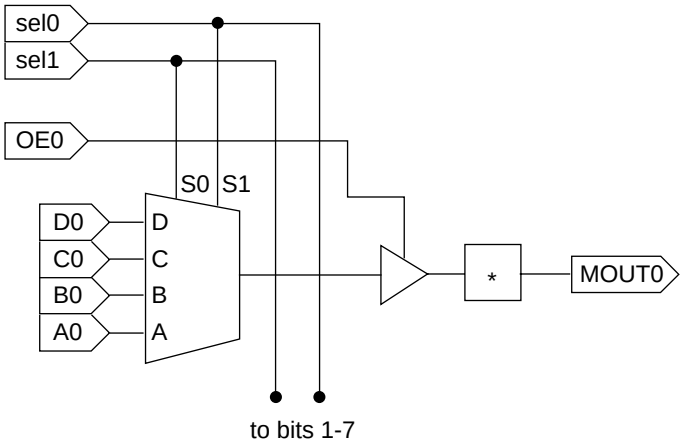
```
SET AIN  [AIN0..AIN7];
SET YOUT [YOUT0..YOUT7];

INPUT  [GNDB,GNDC,GNDD] {B39,C39,D39};
INPUT  [!G1,!G2]        {A2,A3};
INPUT  AIN              {A4,A5,A6,A7,A8,A9,A10,A11};
BIDI   NOR1             {A13};
OUTPUT YOUT             {B0,B1,B2,B3,B4,B5,B6,B7};

BEGIN
      NOR1.oe  = VCC;
      NOR1.m0  = VCC;
      NOR1.m1  = GNDB;
      NOR1.m2  = GNDC;
      NOR1.m3  = GNDD;
      NOR1.s0  = G1;
      NOR1.s1  = G2;

      YOUT.oe = NOR1;
      YOUT.m0 = !AIN;
      YOUT.s0 = GND;
      YOUT.s1 = GND;
END
```

Octal MUXes

An octal 4-to-1 MUX consists of eight 4-to-1 MUXes with individual output enables and common selection signals. Since each I/O cell of an ispGDX device contains a 4-to-1 MUX, this application is tailor-made for the ispGDX Family. Because of its functionality, this application is I/O-intensive, but it uses few extra resources on the chip. In the connection section of the syntax (see Listing 3), the eight `MOUT` cells are assigned eight output enables and MUX data inputs, but common selection signals to realize the function mentioned above. A0 in the schematic corresponds to the first signal on the `AIN` bus in the code.

An octal 2-to-1 MUX can be quickly constructed by modifying the 4-to-1 MUX file. There are several ways to do the selection. In the example shown in Listing 4, sel0 is tied low, so that selection can be done by a single bit, `sel`, connected to sel1. The MUX inputs must be connected to the "A" or "B" banks to be correctly configured so the selection can be made by `sel`.

Figure 8. Octal 4-to-1 MUX Configuration



Listing 3. Octal 4-to-1 MUX Syntax (Partial)

```
INPUT AIN   {A0,A1,A2,A3,A4,A5,A6,A7};
INPUT BIN   {B0,B1,B2,B3,B4,B5,B6,B7};
INPUT CIN   {C0,C1,C2,C3,C4,C5,C6,C7};
INPUT DIN   {D0,D1,D2,D3,D4,D5,D6,D7};
INPUT OE    {A9,A13,A17,A21,A25,A29,A33,A37};
INPUT SEL0  {A10};
INPUT SEL1  {A11};
OUTPUT MOUT {B8,B9,B10,B11,B12,B13,B14,B15};

BEGIN
      MOUT.oe = OE;
      MOUT.m0 = AIN;
      MOUT.m1 = BIN;
      MOUT.m2 = CIN;
      MOUT.m3 = DIN;
      MOUT.s0 = SEL0;
      MOUT.s1 = SEL1;
END
```

Listing 4. Octal 2-to-1 MUX Syntax (Partial)

```
INPUT AIN   {A0,A1,A2,A3,A4,A5,A6,A7};
INPUT BIN   {B0,B1,B2,B3,B4,B5,B6,B7};
INPUT OE    {A9,A13,A17,A21,A25,A29,A33,A37};
INPUT SEL   {A11};
OUTPUT MOUT {B8,B9,B10,B11,B12,B13,B14,B15};

BEGIN

      MOUT.oe = OE;
      MOUT.m0 = AIN;
      MOUT.m1 = BIN;
      MOUT.s0 = GND;
      MOUT.s1 = SEL;
END
```

Listing 5. 8-Bit Register Syntax (Partial)

```
SET DI [DI0..DI7];
SET Q  [Q0..Q7];

INPUT  CLK {Y0};
INPUT  !OC {A1};
INPUT  DI  {A2,A3,A4,A5,A6,A7,A8,A9};
OUTPUT Q   {B0,B1,B2,B3,B4,B5,B6,B7};

BEGIN
      Q.oe  = !OC;
      Q.clk = CLK;
      Q.m0  = DI;
      Q.s0  = GND;
      Q.s1  = GND;
END
```

8-Bit Register

An 8-bit register is contains eight D flip-flops with common clock and output enable signals. Implementing it is trivial in the ispGDX Development System. The signals that need to be specified are the inputs to the flip-flops, their outputs, an output enable signal, and a clock. Using set notation allows for assignment of signals as buses, simplifying the code. The connection section connects all eight signals in one set of connection statements. The compiler knows to use the D flip-flops in the output cells since `.oe` and `.clk` are specified. The `OC` and `CLK` signals are assigned to all of the registers. The input signals come in from the "A" bank, and are assigned to the `.m0` line in the ispGDX syntax. Grounding the select lines selects the "A" bank.

Octal Transceiver

The basic functional unit of the transceiver consists of two tri-state buffers connected in opposite directions along a common line. These buffers are enabled by two signals (`GAB` and `!GBA`). Asserting either one of the enabling inputs allows data propagation in the appropriate direction. When one enabling input is asserted, the signal travels through the buffer in one direction, while the buffer in the other direction is tri-stated.

This is easily implemented in ispGDX syntax using set notation (see Listing 6). The two buses are simply assigned to each other using `GAB` and `!GBA` as the `.oe` signals. The `SET` statement defines all the signals that are to be included in the bus. These signals can then be assigned to any bus of the same size in the connection section of the file. The `BUSA.m1=BUSB` assignment is actually assigning eight inputs to eight outputs, and the `BUSA.oe=!GBA` assigns the `!GBA` signal (which is not a bus signal) to all of the output enables on `BUSA`. The 8-

Figure 9. 8-Bit Register Configuration

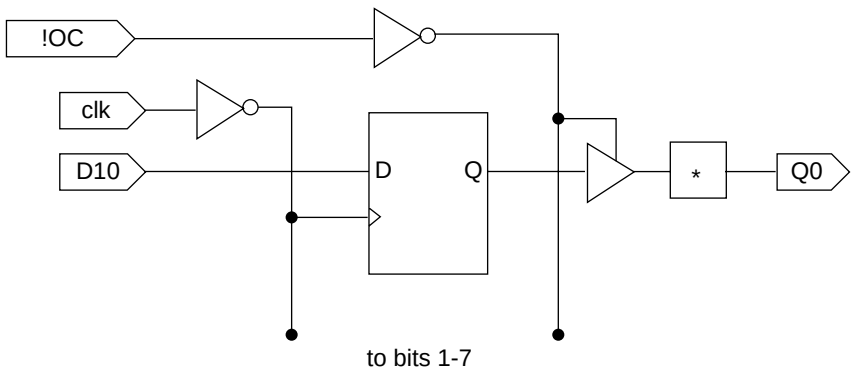
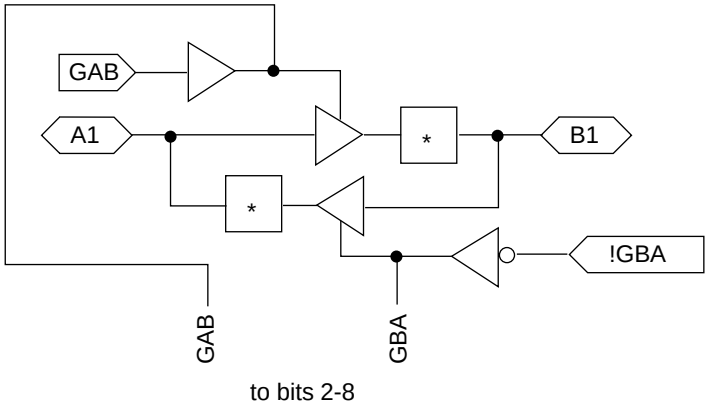


Figure 10. Octal Transceiver Configuration



Listing 6. Octal Transceiver Syntax (Partial)

```
SET BUSA [BUSA0..BUSA7];
SET BUSB [BUSB0..BUSB7];

INPUT [GAB,!GBA] {A13,B13};
BIDI BUSA  {A0,A1,A2,A3,A4,A5,A6,A7};
BIDI BUSB  {B0,B1,B2,B3,B4,B5,B6,B7};

BEGIN
       BUSA.oe=!GBA;
       BUSA.m1=BUSB;
       BUSA.s0=GND;
       BUSA.s1=VCC;

       BUSB.oe=GAB;
       BUSB.m0=BUSA;
       BUSB.s0=GND;
       BUSB.s1=GND;
END
```

bit wide design uses 18 I/O cells. With this implementation, an ispGDX 160 could easily be expanded to accommodate multiple 16 or 32 bit buses. Figure 10 shows the layout of one bit of the transceiver.

4-Bit Shift Register

A shift register is usually implemented using gates and flip-flops, but can be accommodated into an ispGDX 160 device with minimal design effort. Using the 4-to-1 MUXes, a shift register can be built, with each MUX having inputs corresponding to shift left, shift right, hold and load. Designing a shift register for an ispGDX can be used as an alternative to designing with a FPGA. The ispGDX architecture is geared toward this kind of application, while a FPGA uses generic logic blocks, leading to an implementation that would use up more resources.

If the pin assignments are done carefully, a four-bit shift register can be built using just a few I/O cells. For comparison purposes, Figure 11 shows the schematic of the first stage of a shift register built using traditional methods, and Figure 12 shows its implementation using ispGDX I/O cells (each cell is represented by a 4-to-1 MUX).

In the ispGDX device implementation (a selection of the code is listed in Listing 7), four output MUXes are used as the four outputs of the shift register. Four additional decoding MUXes are used to decode the mode selector and route the selection bits to the appropriate output MUX. These decoding MUXes translate the `MODE0` and `MODE1` inputs to the correct selection code to perform the shift left, shift right, hold and load functions in the output MUX. If the register is expanded to more than four bits, there is no need to add more decoding MUXes, since the same codes can be used. For example, in the code below, for the first bit (`SH_OUTA`), the decoding MUXes provide the correct codes to the s0 and s1 inputs to select the "B" line for a shift left, "D" for a shift right, "A" for a hold, and "C" for a load. The A1D0 decoding MUX provides the correct code for the s1 line of the first output MUX (shown in Figure 12) and the s0 line of the fourth output MUX (not shown). If the register is expanded to eight bits, the A1D0 will provide codes for the s1 line of the fifth output MUX and the s0 line of the eighth output MUX. Similarly, the B1A0 provides codes for the s0 line of the first MUX and the s1 line of the second MUX.

There is no set/constant section after the device section in this file because the routing to the output MUXes doesn't occur the same order for any two of the MUXes,

Figure 11. TTL Shift Register Configuration (First Bit)

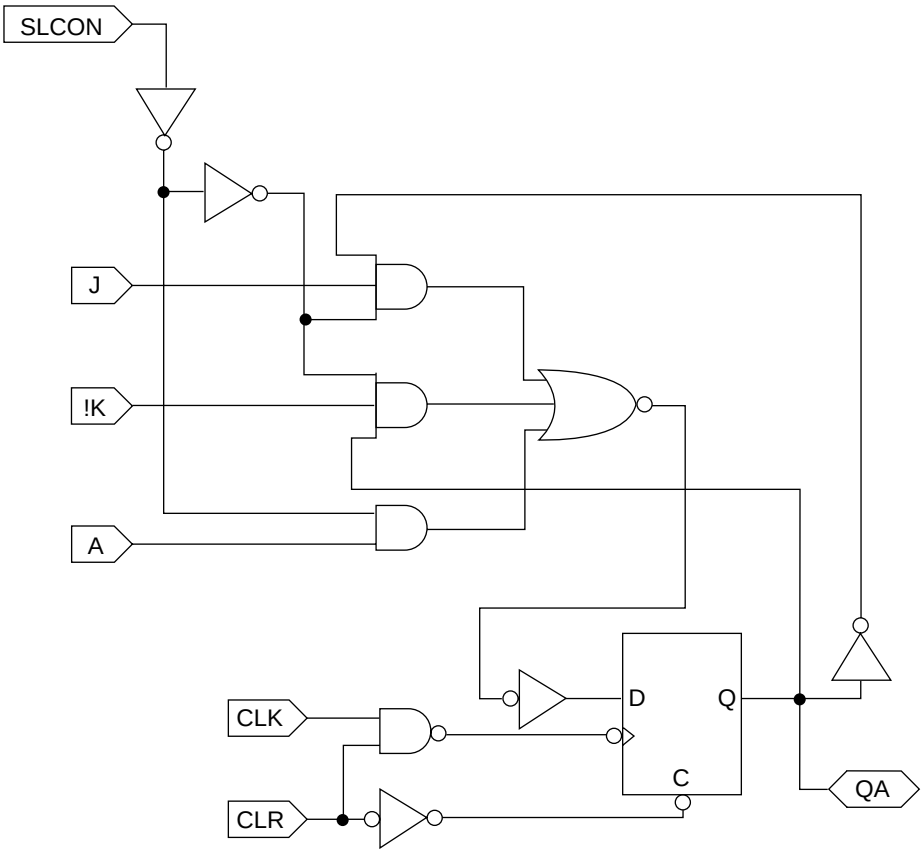
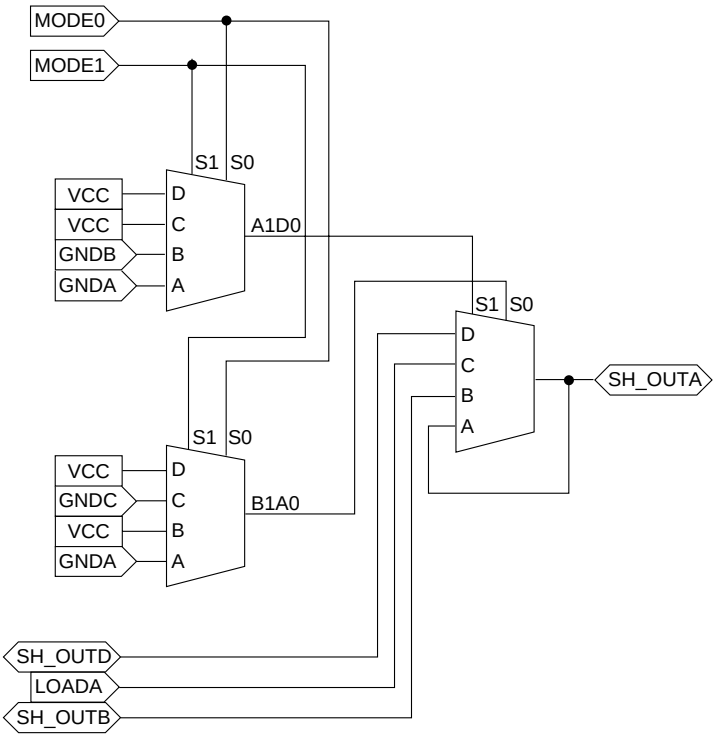


Figure 12. ispGDX Shift Register Implementation (First Bit)



Listing 7. Syntax for First Bit (Partial)

```
INPUT CLK                                {Y0};
INPUT [LOADA,LOADB,LOADC,LOADD]          {C0,D0,A0,B0};
INPUT [GNDA,GNDB,GNDC,GNDD]              {A39,B39,C39,D39};
INPUT [MODE0,MODE1]                      {A2,A3};
BIDI  [A1D0,B1A0,C1B0,D1C0]              {A6,A7,A10,A11};
BIDI  [SH_OUTA,SH_OUTB,SH_OUTC,SH_OUTD] {A4,B4,C4,D4};

BEGIN
A1D0.oe  = VCC;
A1D0.m0  = GNDA;
A1D0.m1  = GNDB;
A1D0.m2  = VCC;
A1D0.m3  = VCC;
A1D0.s0  = MODE0;
A1D0.s1  = MODE1;

B1A0.oe  = VCC;
B1A0.m0  = GNDA;
B1A0.m1  = VCC;
B1A0.m2  = GNDC;
B1A0.m3  = VCC;
B1A0.s0  = MODE0;
B1A0.s1  = MODE1;

SH_OUTA.clk = CLK;
SH_OUTA.oe  = VCC;
SH_OUTA.m0  = SH_OUTA;
SH_OUTA.m1  = SH_OUTB;
SH_OUTA.m2  = LOADA;
SH_OUTA.m3  = SH_OUTD;
SH_OUTA.s0  = A1D0;
SH_OUTA.s1  = B1A0;
END
```

eliminating the possibility of set assignment. As inputs, there is a clock assigned to a universal clock pin, four inputs, two mode select inputs and four ground inputs (one for MUX input bank). These ground inputs are needed because the I/O cells require data inputs that are mixed combinations of VCC and ground. Internally, the MUX data inputs can be tied either to VCC or ground, but not both. By providing the external ground inputs GNDA through GNDD, the MUX inputs can be tied to any combination of VCC and ground. Thus, one signal for each I/O bank must be specified as a virtual ground and externally tied low to implement logic functions. The codes for the four modes of the register are listed in Table 4.

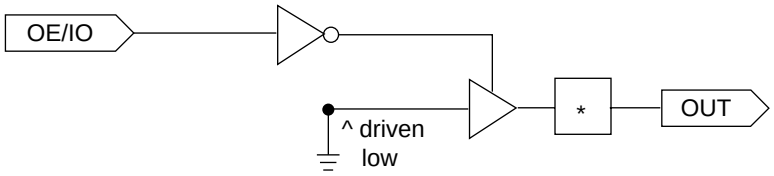
As mentioned earlier, the mode needs to be translated to an appropriate select code for each stage in the register since each stage has different locations for the different shift register functions. The same functions are assigned to different banks from bit to bit to satisfy the routing requirements of the ispGDX device. For example, shifting right from the "A" register to "B" register must be located in the MUXA input on the "B", while shifting right from the "D" register to the "A" register must be located on the MUXD input of "A." The output MUXes themselves are all connected in the same way: each one has inputs

Table 4. Bit Codes for Mode

MODE	CODE
Hold	00
Shift Left	01
Load	10
Shift Right	11

Register Modes

Figure 13. Open Drain Configuration



from itself (for the hold function), the register to its left and right, and a load input on the remaining MUX input.

PSR

Finally, the ispGDX Family can act as Programmable Switch Replacement (PSR). Designing a ispGDX device for switch replacement is a relatively trivial process, compared to some of the others described in this document. The outputs of the ispGDX device can be driven high or low to match the desired switch or jumper settings. The configurations of the device at a given time can easily be changed to emulate different switch/jumper settings through the use of in-system programmability.

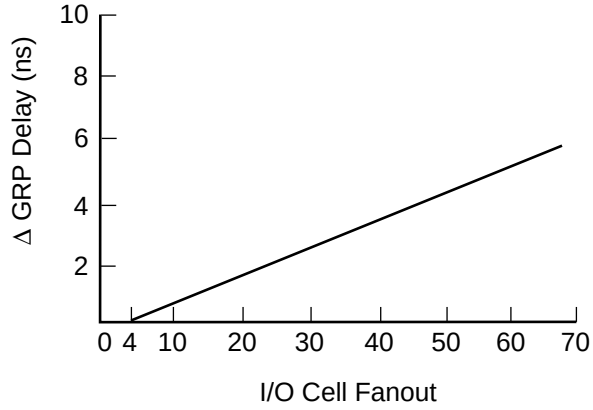
Other ispGDX Applications

There are a few other applications that the ispGDX Family can emulate that don't fall under the three general application categories described here. The 24 mA bus driving capability allows the ispGDX device to be used in bus driver and transceiver applications. In addition, the ispGDX device output buffers are designed to have PCI-compatible output drive. Many PCI applications require a very fast tco (5 ns) and tsu (4 ns) to get on and off the bus. The ispGDX Family is ideal for these applications. An ispGDX device along with a small ispLSI® device can accommodate custom PCI controller designs.

The combination of ISP and Boundary Scan test registers can provide a very powerful system test capability with the ispGDX devices. Similar to how the ispGDX is used in a central signal routing application, the boundary scan registers can provide the signal driving capability to the various nodes with known logic levels for test purposes. This can be done through the standard JTAG test port interface.

The ispGDX can also emulate CMOS open drain outputs. In an open drain output configuration, bused outputs are not actively driven high by devices on the bus. This is done by connecting their outputs together to a common pull up resistor. If any output turns on, the common output is driven low. In an ispGDX device, this can be done by connecting the inverted data input to the output enable and driving the data input to a fixed low. Thus, the output will only be driven low when the data (OE) input is low. Figure 13 shows this configuration, where OE/IO is the input signal that drives the OE node as well as the MUX input node, which is connected to a "zero."

Figure 14. Maximum Δ GRP Delay vs. I/O Cell Fanout



Hardware Considerations

There are a few hardware features of the ispGDX Family that are helpful to understand for design purposes. The first deals with output drive and slew, and the second with fanout.

The entire ispGDX Family's outputs have been specified for 24 mA sink and source current and can be tied in parallel to increase drive capability. In addition, each pin's slew rate can be defined independently. Defining a pin as a slow slew pin adds an additional 5 ns of delay to the output buffer (propagation delay is about 5 ns from pin to pin).

Increasing the fanout from an input causes a small increase in propagation delay through the Global Routing Pool (GRP). The GRP is the interconnection matrix through which all inputs and feedback are routed to outputs according to the JEDEC file produced by the ispGDX Development System. GRP delay added related to fanout is 0.1 ns for every extra line driven beyond four (the propagation delay given in the data sheet is for four GRP load). Figure 14 shows the extra propagation delay for a wide range of fanout values.

Conclusion

The ispGDX Family of devices are extremely versatile and powerful tools whose abilities range well beyond crosspoint. They can provide solutions in programmable system interconnect, data path (such as integrated TTL emulation) and switch replacement that are easy to code and download onto devices.

Technical Support Assistance

Hotline: 1-800-LATTICE (Domestic)
1-408-826-6002 (International)
e-mail: techsupport@latticesemi.com