Migrating iCEcube2 iCE40 UltraPlus Designs to Lattice Radiant Software

April 20, 2018

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

Migrating iCEcube2 Designs to Lattice Radiant Software User Guide

This document provides guidance for iCEcube2 users to accelerate learning of the Lattice Radiant™ software.

When migrating iCEcube2 iCE40 UltraPlus designs to Radiant software, users of the Radiant software should expect to go through the normal design process, such as design entry, design analysis, debug, simulation, and testing, because iCEcube2 and Radiant software are different tools.

The following general guidelines are recommended:

- For primitives that are inferable by the synthesis tool (LUT, I/O, RAM, etc.), let the synthesis tool infer them rather than directly instantiating them.
- If synthesis inference is not sufficient or is not available for a certain primitive, use the Radiant IP Catalog tool rather than directly instantiating them.
- If direct instantiation is required, use the Radiant Template Editor tool to help instantiate. Note that some primitives do not have direct equivalents between iCECube2 and the Radiant software. For more details, refer to Table 1.

More information on Radiant Primitives, refer to the Radiant software online Help under **References > FPGA Libraries Reference Guide**.

Table 1 provides guidelines to migrate iCEcube2 design functions to Radiant software.

Table 1: Design Function Migration from iCEcube2 to Radiant Software for iCE40UP

Category	Migration Functions	iCEcube2 Software	Radiant Software	Suggested Usage in Radiant Software
I/O	▶ Input buffer ▶ Output buffer ▶ Bidirectional buffer	Inference or SB_IO primitive instantiation.	Inference or I/O buffer primitive (IB, OB, OBZ, BB_B) instantiation.	Let the synthesis infer I/O buffers from the source RTL. If manual instantiation is needed, use IB for input, OB for output, OBZ for output with tristate, and BB_B for bidirectional buffer.
	▶ Input register ▶ Output register	Inference or SB_IO primitive instantiation.	Inference, I/O register primitive instantiation (IFD1P3AZ or OFD1P3AZ) or IOL_B primitive instantiation.	Let the synthesis infer I/O registers from the source RTL. To enable I/O register inference, the “Use IO Register” option must be enabled in Lattice Synthesis Engine (LSE). This option is enabled by default. The “syn_useioff=1/0” attribute can also be used to control I/O register inference globally or locally (for both LSE and Synplify Pro). To force an I/O register, instantiate an I/O register primitive (IFD1P3AZ for input register, OFD1P3AZ for output register). IOL_B can also be instantiated and then configured to be an I/O register, but direct instantiation of the I/O register primitives is suggested. While an I/O register primitive is instantiated, the synthesis tool can still infer an I/O buffer for the instantiated I/O register. When a bidirectional buffer is instantiated, BB_B must be used instead of IB and OBZ. To implement a bidirectional I/O register, IOL_B must be instantiated.
	▶ DDR output ▶ DDR output	SB_IO primitive instantiation with parameter settings to DDR.	IOL_B primitive instantiation with parameter settings to configure it to a DDR mode.	Instantiate an IOL_B primitive and use the LATCHIN or DDROUT parameters to configure to the desired DDR mode. For an input DDR, set “LATCHIN=NONE_DDR”. For an output DDR, set “DDROUT=YES”. By default, DDR is set to input mode. Let the synthesis infer an appropriate I/O buffer for the instantiated IOL_B. Or, IB, OB, or BB_B can also be instantiated for input, output and bidirectional.

Table 1: Design Function Migration from iCEcube2 to Radiant Software for iCE40UP (Continued)

Category	Migration Functions	iCEcube2 Software	Radiant Software	Suggested Usage in Radiant Software
I/O (Continued)	Primary clock buffer	SB_GB, SB_GB_IO primitive instantiation or inference.	Set "USE_PRIMARY=TRUE" attribute to "ldc_set_attribute" constraint	Use the clock input ports that have direct access to the primary clock spine. Non-direct access ports can be forced to use a primary clock spine through general routing. Use the "Use Primary Net" option in Radiant Device Constraint Editor (DCE) or add the following constraint: "ldc_set_attribute {USE_PRIMARY=TRUE} [get_nets <clock_obj>]".
	Open Drain buffer	SB_IO_OD primitive instantiation.	BB_OD primitive instantiation.	Instantiate BB_OD primitive. When a BB_OD primitve is instantiated to use an open drain I/O as a GPIO, use the T_N input to control the output enable. Note that T_N is an active low tri-state control. To enable the output, T_N should be driven high.
Memory	Block RAM (4Kb) with rising edge clocking.	Primitive instantiation of SB_RAM_256x16, SB_RAM_512x8, SB_RAM_1024x4, SB_RAM_2048x2.	Inference, RAM_DP or RAM_DQ module IP generation,PDP4K primitive instantiation.	Let the synthesis infer an appropriate RAM module from the RTL source. If a RAM instantiation is needed, generate a RAM_DP or RAM_DQ module from Radiant IP Catalog. Direct instantiation of PDP4K primitive is neither necessary nor recommended unless a special function like bit masking is needed.
	Block RAM (4Kb) with falling edge clocking.	Primitive instantiation of SB_RAM_256x16NR, SB_RAM_512x8NR, SB_RAM_1024x4NR, SB_RAM_2048x2NR, SB_RAM_256x16NW, SB_RAM_512x8NW, SB_RAM_1024x4NW, SB_RAM_2048x2NW, SB_RAM_256x16NRNW, SB_RAM_512x8NRNW, SB_RAM_1024x4NRNW, SB_RAM_2048x2NRNW.	Inference, RAM_DP or RAM_DQ module IP generation, PDP4K primitive instantiation.	Let the synthesis infer an appropriate RAM module from the RTL source with negated clock input(s). If a RAM instantiation is needed, generate a RAM_DP or RAM_DQ module from Radiant IP Catalog, and then drive the clock port(s) with inverted clock(s).
	Block RAM with different R/W port sizes	SB_RAM40_4K primitive instantiation with WRITE_MODE, READ_MODE parameters.	PDP4K primitive instantiation with DATA_WIDTH_W & DATA_WIDTH_R parameters.	Instantiate PDP4K primitive with DATA_WIDTH_W & DATA_WIDTH_R parameters to configure port sizes.

Table 1: Design Function Migration from iCEcube2 to Radiant Software for iCE40UP (Continued)

Category	Migration Functions	iCEcube2 Software	Radiant Software	Suggested Usage in Radiant Software
Memory (Continued)	Block RAM with port/bit mask (rising edge clocking)	SB_RAM_256x16 primitive instantiation.	PDP4K primitive instantiation.	Instantiate PDP4K primitive and use the MASK_N input to control bit mask.
	Single Port Block RAM with nibble masking/power control	SB_SPRAM256KA primitive instantiation.	SP256K primitive instantiation.	Instantiate SP256K primitive. If nibble masking or power control is not needed, RAM_DQ in Radiant IP Catalog can also be used.
PLL	PLL Generation	PLL generation from "Configure PLL Module" tool.	PLL module generation from Radiant IP Catalog.	Generate a PLL module using Radiant IP Catalog PLL module IP.
DSP	DSP Functions	Configuration through SB_MAC16 primitive instantiation.	Arithmetic module generation in Radiant IP Catalog: - Adder, Suntractor, Multiplier, Multi_Add_Sub - Complex_Mult, Multi_Accumulate PMI instantiation: - pmi_add, pmi_sub, pmi_mult, pmi_multaddsub - pmi_complex_mult, pmi_mac, pmi_dsp Use pmi_dsp to configure MAC16 primitive.	Use Radiant IP Catalog to generate a DSP module or instantiate a PMI module for a desired DSP function. When MAC16 primitive instantiation is needed for manual configuration, instantiate a pmi_dsp wrapper for an easier instantiation with bused ports.
OSC	Internal Oscillators	SB_HFOSC, SB_LFOSC primitive instantiation.	HSOSC, LSOSC primitive instantiation.	Instantiate HSOSC primitive with CLKHF_DIV parameter for high speed OSC. Instantiate LSOSC primitive for low speed OSC. When VPP_2V5 is connected to <2.3V, use HSOSC1P8V, LSOSC1P8V instead.
LED	RGB LED Drivers	SB_RGBA_DRV primitive instantiation.	RGB primitive instantiation.	Instantiate RGB primitive. The ports and parameters are identical. When VPP_2V5 is connected to <2.3V, use RGB1P8V instead Other RGB primitives are for SW use and not recommended for user use (except RGBPWM).
	PWM Generator	SB_LEDDA_IP primitive instantiation.	RGBPWM primitive instantiation.	Instantiate RGBPWM primitive. Note that unnecessary LEDDRST port in SB_LEDDA_IP is no longer available in RGBPWM.
I2C	I2C Function	I2C generation from “Configure SPI/I2C Module” tool.	I2C generation from Radiant IP Catalog SPI_I2C IP module.	Generate an I2C module from Radiant IP Catalog SPI_I2C module IP.
SPI	SPI Function	SPI generation from “Configure SPI/I2C Module” tool	SPI generation from Radiant IP Catalog SPI_I2C IP module.	Generate a SPI module from Radiant IP Catalog SPI_I2C module IP.

Table 1: Design Function Migration from iCEcube2 to Radiant Software for iCE40UP (Continued)

Category	Migration Functions	iCEcube2 Software	Radiant Software	Suggested Usage in Radiant Software
MISC	Warm Boot Function	SB_WARMBOOT primitive instantiation.	WARMBOOT primitive instantiation.	Instantiate WARMBOOT primitive.
	Registers	Inference or SB_DFF primitive instantiation.	Inference or FD1P3XZ primitive instantiation configured with REGSET and SRMODE parameters.	Let the synthesis infer register functions from the source RTL. If a manual instantiation is needed, instantiate FD1P3BZ, FD1P3DZ, FD1P3IZ or FD1P3JZ primitive: FD1P3BZ: active high CE, asynchronous SET. FD1P3DZ: active high CE, asynchronous RESET. FD1P3IZ: active high CE, synchronous SET (SET overrides CE). FD1P3JZ: active high CE, synchronous RESET (PRESET overrides CE).

Migration Examples

This section provides HDL examples of migrating iCEcube2 designs to Radiant software. Examples include:

- I/O Buffers
- Output Buffer
- DDR I/O Mode
- Primary Clock Net Access
- General Purpose Use of High-drive Open Drain Pads
- Block RAM (EBR)
- Block RAM (EBR) with Clock Polarity Inversion
- Block RAM (EBR) with Different Read/Write Port Sizes
- Block RAM (EBR) with Bit Masking Function
- Single Port Block RAM with Nibble Masking and Power Control
- PLL
- DSP Functions
- Oscillator Functions
- RGB LED Drivers
- RGB PWM Generator
- I2C and SPI Function
- Warm Boot Function

- Registers

I/O Buffers

I/O buffers are generally inferred by the synthesis tools. When a specific IO buffer is to be instantiated, use the instantiation examples shown below:

Input Buffer

```
IB u_IB (
  .I (reset),    // I
  .O (reset_c)   // O
);
```

Output Buffer

```
OB u_OB [2:0] (
  .I (c_in),  // I
  .O (c_out)  // O
);
```

Output Buffer with Tri-state Control

The following RTL expression example will infer an output buffer with a tri-state control:

```
assign sda = oe_sda ? sda_out : 1'bz ;
```

To instantiate an output buffer with tri-state control:

```
OBZ_B u_OBZ_B (
  .I   (sda_out), // I
  .T_N (oe_sda),  // I, active-low tristate
  .O   (sda)      // O, port pin
);
```

Bidirectional Buffer (RTL expression)

The following RTL expression example will infer a bidirectional buffer:

```
assign sda = oe_sda ? sda_out : 1'bz;
assign sda_in = sda;
```

To instantiate a bidirectional buffer:

```
BB_B u_BB_B (
  .T_N (oe_sda),  // I, active-low tristate
  .I   (sda_out), // I, from fabric
  .O   (sda_in),  // O, to fabric
  .B   (sda)      // IO, port pin
);
```

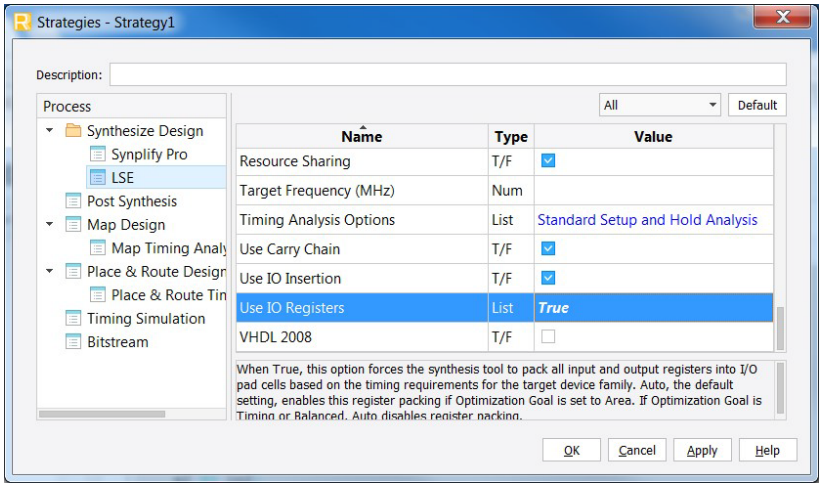
I/O Registers

To allow I/O register inference globally, apply the syn_useioff attribute to the module as shown below:

```
module test (in1, in2, clk, out1) /* synthesis syn_useioff = 1 */;
```

Note

LSE has a GUI-level global option control in Strategy_option > LSE > Use IO Registers. Default is set to Auto. It can be forced (True) or disabled (False).



To locally control I/O register inference, apply the syn_useioff attribute to the I/O port definition as shown below:

```
// in1 to use Input register, out1 to use Output register
module test (in1, in2, clk, out1) ;
input in1    /* synthesis syn_useioff = 1 */;
input in2    /* synthesis syn_useioff = 0 */;
input clk;
output out1; /* synthesis syn_useioff = 1 */;
```

To instantiate an input register:

```
IFD1P3AZ u_IFD1P3AZ (
  .D  (in1),   // I
  .SP (1'b1),  // I, active high clock enable
  .CK (clk),   // I
  .Q  (in1_r)  // O
);
```

To instantiate an output register:

```
OFD1P3AZ u_OFD1P3AZ (
  .D  (o1),    // I
  .SP (1'b1),  // I, active high clock enable
  .CK (clk),   // I
```

```
    .Q  (out1)   // O
  );
```

To instantiate a bidirectional input/output register, both IOL_B and BB_B primitives need to be instantiated:

```
inout bd_ior;
IOL_B u_IOL_B (
  .PADDI  (pad2ir),  // I, from pad to input register input
  .DO1    (1'b0),    // I
  .DO0    (fab2or),  // I, from fabric to output register input
  .CE     (1'b1),    // I
  .IOLTO  (fab2oe),  // I, from fabric to oe/tristate control
  .HOLD   (1'b0),    // I
  .INCLK  (clk),     // I
  .OUTCLK (clk),     // I
  .PADDO  (or2pad),  // O, from output register to pad
  .PADDT  (oe2pad),  // O, from oe/tristate output to pad
  .DI1    (),        // O
  .DI0    (ir2fab)   // O, from input register output to fabric
);
BB_B u_BB_B (
  .T_N (oe2pad),  // I,  from oe/tristate output to pad
  .I   (or2pad),  // I,  from output register to pad
  .O   (pad2ir),  // O,  from pad to input register input
  .B   (bd_ior)   // IO, bidirectional pad
);
```

BB_B instantiation can be replaced with an RTL expression as shown below:

```
  assign bd_ior = oe2pad ? or2pad : 1'bz ;
  assign pad2ir = bd_ior;
```

DDR I/O Mode

To instantiate an input DDR function, use the example shown below:

```
// 8-bit bus input DDR
input [7:0] ddrin;
IOL_B
#(
  .LATCHIN ("NONE_DDR"),
) u_ddr_IOL_B [7:0] (
  .PADDI  (ddrin),    // I, ddr data input from pad
  .DO1    (1'b0),     // I
  .DO0    (1'b0),     // I
  .CE     (1'b1),     // I, clock enabled
  .IOLTO  (1'b1),     // I
  .HOLD   (1'b0),     // I
  .INCLK  (clk),      // I, clock for input ddr
  .OUTCLK (clk),      // I
  .PADDO  (),         // O
  .PADDT  (),         // O
  .DI1    (ddrin_n),  // O, falling edge data to fabric
  .DI0    (ddrin_p)   // O, rising edge data to fabric
);
```

To instantiate an output DDR function, use the example shown below:

```
//single bit output DDR
output ddrout;
IOL_B
#(
  .DDROUT ("YES")
) u_ddr_IOL_B (
  .PADDI  (1'b0),     // I
  .DO1    (ddrout_n), // I, falling edge data from fabric
  .DO0    (ddrout_p), // I, rising edge data from fabric
  .CE     (1'b1),     // I, clock enabled
  .IOLTO  (1'b1),     // I
  .HOLD   (1'b0),     // I
  .INCLK  (clk),      // I
  .OUTCLK (clk),      // I, clock for output ddr
  .PADDO  (ddrout),   // O, ddr data output to pad
  .PADDT  (),         // O
  .DI1    (),         // O
  .DI0    ()          // O
);
```

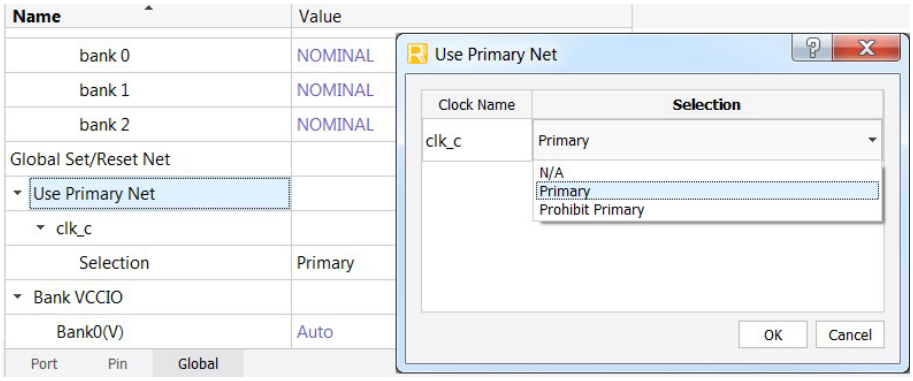
When the RGB pins are used for DDR, use the example shown below:

```
//input DDR
input [2:0] ddrin;  // 3 RGB pins

BB_OD u_ddr_BB_OD [2:0] (
  .T_N (1'b0),      // I,  output disabled, input only
  .I   (1'b0),      // I
  .O   (ddrin_buf), // O,  buffered data to IOL_B
  .B   (ddrin)      // IO, ddr input data
);

IOL_B
#(
  .LATCHIN ("NONE_DDR")
) u_ddr_IOL_B [2:0] (
  .PADDI  (ddrin_buf), // I, from pad buffer
  .DO1    (1'b0),      // I
  .DO0    (1'b0),      // I
  .CE     (1'b1),      // I, clock enabled
  .IOLTO  (1'b1),      // I
  .HOLD   (1'b0),      // I
  .INCLK  (clk),       // I, clock for input ddr
  .OUTCLK (clk),       // I
  .PADDO  (),          // O
  .PADDT  (),          // O
  .DI1    (ddrin_n),   // O, falling edge data to fabric
  .DI0    (ddrin_p)    // O, rising edge data to fabric
);
```

Primary Clock Net Access

To allow a net to access a primary clock spine, use “Use Primary Net” option to set the net to Primary in Device Constraint Editor. To prevent a net from using a primary net, on the other hand, select “Prohibit Primary”.

1. Open Device Constraint Editor and select Global tab.
2. Double click on Use Primary Net to open the Use Primary Net dialog box.
3. Find a clock net to be set primary and double click on the Selection to access the available options.
4. Select Primary or Prohibit Primary from the pull-down options.
5. Save the change.



Once saved, the setting is recorded in the PDC constraint file as shown the examples below:

```
# Set to use primary clock spine
ldc_set_attribute {USE_PRIMARY=TRUE} [get_nets clk_c]

# Prohibit primary clock spine
ldc_set_attribute {USE_PRIMARY=FALSE} [get_nets clk_c]
```

Note

If a net does not have a direct access to the primary net, Radiant software will use general routing with a warning message to indicate a use of general routing resource.

General Purpose Use of High-drive Open Drain Pads

Three high-drive pads (R,G,B) can also be used as GPIO. To use them as GPIO, BB_OD primitive must be instantiated. BB_OD can be used alone or in conjunction with IOL_B when IO register or a DDR function is needed. See the RGB pins for DDR example in the DDR IO Mode section for usage and the connections.

Block RAM (EBR)

This section provides Block RAM (EBR) examples.

Inference

EBR blocks can be inferred from an RTL memory expression as the example shown below:

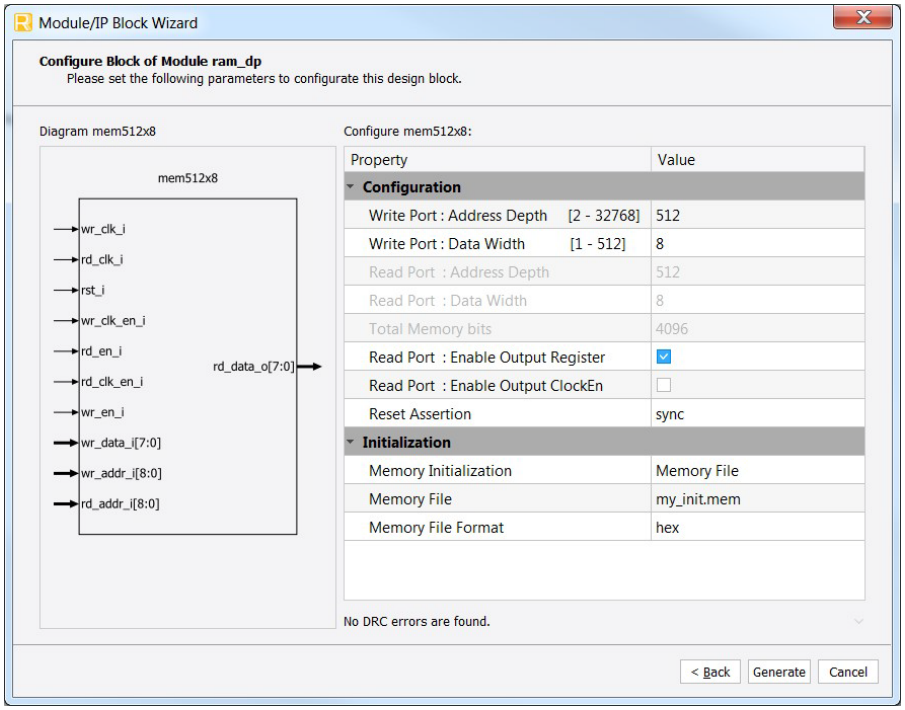
```
// 512x8 (4Kb) dual port RAM memory
wire [8:0] wr_addr, rd_addr;
wire [7:0] wr_data;
reg  [7:0] rd_data;
reg [511:0] mem [7:0];
always @ (posedge wr_clk)
  if (wr_en) mem[wr_addr] <= wr_data;
always @ (posedge rd_clk)
  if (rd_en) rd_data <= mem[rd_addr];
```

Module IP Generation

If a specific size or type of a memory function is to be generated, use of IP Catalog is recommended to generate a desired memory module. To generate a memory module from IP Catalog:

1. Select IP Catalog from the main GUI.
2. Select either dual port (RAM_DP) or single port (RAM_DQ) from EBR_Components under Memory_Modules category.
3. Follow the Module/IP Block Wizard to complete configuration.
4. If memory initialization is needed, prepare a memory initialization file and select “Memory File” from the Memory Initialization option. The file name for the memory initialization file is *.mem (<file_name>.mem). Each row includes the value to be stored in a particular memory location.
5. Generate the module IP and insert it to the project (done by default).
6. Instantiate the generated module using the instantiation template. (Right click on .ipx and select Copy Verilog (or VHDL) instantiation.)

For more detail information about memory module generation, refer to the *Memory Modules User Guide*.



An example of instantiation for the generated module (.ipx) is shown below:

```
mem512x8 u_mem512x8 (
    .wr_clk_i(wr_clk),
    .rd_clk_i(rd_clk),
    .wr_clk_en_i(1'b1  ),
    .rd_clk_en_i(1'b1  ),
    .rd_en_i(rd_en),
    .wr_en_i(wr_en ),
    .wr_addr_i(wr_addr[8:0]),
    .rd_addr_i(rd_addr[8:0]),
    .wr_data_i(wr_data[7:0]),
    .rd_data_o(rd_data[7:0)
);
```

Note

mem512x8 is the name of the generated module IP. The file mem512x8.ipx is inserted to the project when IP generation is completed.

Instantiation

Although generally not recommended, an EBR block can also be instantiated for a special function like bit masking or when different read/write port sizes are required. When a memory block instantiation is needed, instantiate PDP4K which can support various 4Kb memory configurations. For example, PDP4K can be used to instantiate a 2048x2, 1024x4, 512x8 or 256x16

memory configuration using DATA_WIDTH_W and DATA_WIDTH_R parameters. Based on the width parameter configuration, the write data or read data bits must be mapped to the corresponding ports as shown the table below:

PDP4K Parameters		**Data and Address Mapping**		
DATA_WIDTH_W	DATA_WIDTH_R	Input Data	Output Data	Address (MSb to LSb)
2	N/A	{DI[11], DI[3]}	N/A	ADW[10:0]
4	N/A	{DI[13], DI[9], DI[5], DI[1]}	N/A	ADW[9:0]
8	N/A	{DI[14], DI[12], DI[10], DI[8], DI[6], DI[4], DI[2], DI[0]}	N/A	ADW[8:0]
16	N/A	{DI[15:0]}	N/A	ADW[7:0]
N/A	2	N/A	{DI[11], DI[3]}	ADW[10:0]
N/A	4	N/A	{DI[13], DI[9], DI[5], DI[1]}	ADW[9:0]
N/A	8	N/A	{DI[14], DI[12], DI[10], DI[8],DI[6], DI[4], DI[2], DI[0]}	ADW[8:0]
N/A	16	N/A	{DI[15:0]}	ADW[7:0]

Below is an example of PDP4K instantiation for 512x8 configuration:

```
wire [7:0]  wr_data;
wire [7:0]  rd_data;
wire [15:0] wr_data_map;
wire [15:0] rd_data_map;
assign wr_data_map =
{1'b0, wr_data[7], 1'b0, wr_data[6], 1'b0, wr_data[5],
 1'b0, wr_data[4], 1'b0, wr_data[3], 1'b0, wr_data[2],
 1'b0, wr_data[1], 1'b0, wr_data[0]};

assign rd_data =
{rd_data_map[14], rd_data_map[12], rd_data_map[10],
 rd_data_map[8],  rd_data_map[6],  rd_data_map[4],
 rd_data_map[2],  rd_data_map[0]};

PDP4K #(
.DATA_WIDTH_W (8),
.DATA_WIDTH_R (8)
) u_PDP4K (
.DI      (wr_data_map),          // I, 16-bit data
.ADW     ({2'b0, wr_addr[8:0]}), // I, 11-bit address
.ADR     ({2'b0, rd_addr[8:0]}), // I, 11-bit address
.CKW     (wr_clk),               // I
.CKR     (rd_clk),               // I
.CEW     (1'b1),                 // I
.CER     (1'b1),                 // I
.RE      (rd_en),                // I
.WE      (wr_en),                // I
.MASK_N  (16'b0),                // I, tie to low
```

```
.DO      (rd_data_map)           // O, 16-bit data
);
```

Note

1. PDP4K primitive has the fixed sizes of data (16-bit) and address (11-bit) buses. Only the necessary part of the buses are connected depending on the parameter configuration. The unused bus lines should be terminated as shown in the example above.

2. MASK_N input is functional only for the 16-bit configuration. Terminate MASK_N to Low for all other data widths as shown in the example.

Block RAM (EBR) with Clock Polarity Inversion

In iCEcube2, driving a block RAM using one or more falling edge clocks requires to instantiate a corresponding block RAM module. In Radiant software, there is no need to instantiate a specific module, and just negating the clock is enough to implement falling edge clocking.

Inference

```
// 512x8 (4Kb) dual port RAM memory with falling edge
clocking
wire [8:0] wr_addr, rd_addr;
wire [7:0] wr_data;
reg  [7:0] rd_data;
reg [511:0] mem [7:0];
always @ (negedge wr_clk)
  if (wr_en) mem[wr_addr] <= wr_data;
always @ (negedge rd_clk)
  if (rd_en) rd_data <= mem[rd_addr];
```

Module IP Instantiation

```
mem512x8 u_mem512x8 (
 .wr_clk_i(~wr_clk),
 .rd_clk_i(~rd_clk),
 .wr_clk_en_i(1'b1),
 .rd_clk_en_i(1'b1),
 .rd_en_i(rd_en),
 .wr_en_i(wr_en ),
 .wr_addr_i(wr_addr[8:0]),
 .rd_addr_i(rd_addr[8:0]),
 .wr_data_i(wr_data[7:0]),
 .rd_data_o(rd_data[7:0)
);
```

Primitive Instantiation

From the PDP4K instantiation example above, the write clock and read clock inputs can be inverted as shown below:

```
.CKW     (~wr_clk),               // I
.CKR     (~rd_clk),               // I
```

Block RAM (EBR) with Different Read/ Write Port Sizes

To implement an EBR with different read and write data bus sizes, a PDP4K primitive must be instantiated. The example below shows the case of 8-bit of write data and 4-bit of read data buses:

```
wire [7:0]  wr_data;
wire [3:0]  rd_data;
wire [15:0] wr_data_map;
wire [15:0] rd_data_map;
assign wr_data_map =
{1'b0, wr_data[7], 1'b0, wr_data[6], 1'b0, wr_data[5],
 1'b0, wr_data[4], 1'b0, wr_data[3], 1'b0, wr_data[2],
 1'b0, wr_data[1], 1'b0, wr_data[0]};

assign rd_data =
{rd_data_map[13],rd_data_map[9],rd_data_map[5],rd_data_map[1]};

PDP4K #(
.DATA_WIDTH_W (8),
.DATA_WIDTH_R (4)
) u_PDP4K (
.DI      (wr_data_map),          // I, 16-bit data
.ADW     ({2'b0, wr_addr[8:0]}), // I, 11-bit address
.ADR     ({1'b0, rd_addr[9:0]}), // I, 11-bit address
.CKW     (wr_clk),               // I
.CKR     (rd_clk),               // I
.CEW     (1'b1),                 // I
.CER     (1'b1),                 // I
.RE      (rd_en),                // I
.WE      (wr_en),                // I
.MASK_N  (16'b0),                // I, tie to low
.DO      (rd_data_map)           // O, 16-bit data
);
```

Block RAM (EBR) with Bit Masking Function

To use the bit masking function, a PDP4K primitive must be instantiated and configured to have 16-bit data width (256x16 mode). The IP Catalog RAM_DP and RAM_DQ module IPs do not support bit masking.

```
wire [15:0]  wr_data;
wire [15:0]  rd_data;
wire [15:0]  mask;
```

```
PDP4K #(
.DATA_WIDTH_W (16),
.DATA_WIDTH_R (16)
) u_PDP4K (
.DI      (wr_data_map),          // I, 16-bit data
.ADW     ({3'b0, wr_addr[7:0]}), // I, 11-bit address
.ADR     ({3'b0, rd_addr[7:0]}), // I, 11-bit address
.CKW     (wr_clk),               // I
.CKR     (rd_clk),               // I
.CEW     (1'b1),                 // I
.CER     (1'b1),                 // I
.RE      (rd_en),                // I
.WE      (wr_en),                // I
.MASK_N  (mask),                 // I, 16-bit bit masking
.DO      (rd_data_map)           // O, 16-bit data
);
```

Single Port Block RAM with Nibble Masking and Power Control

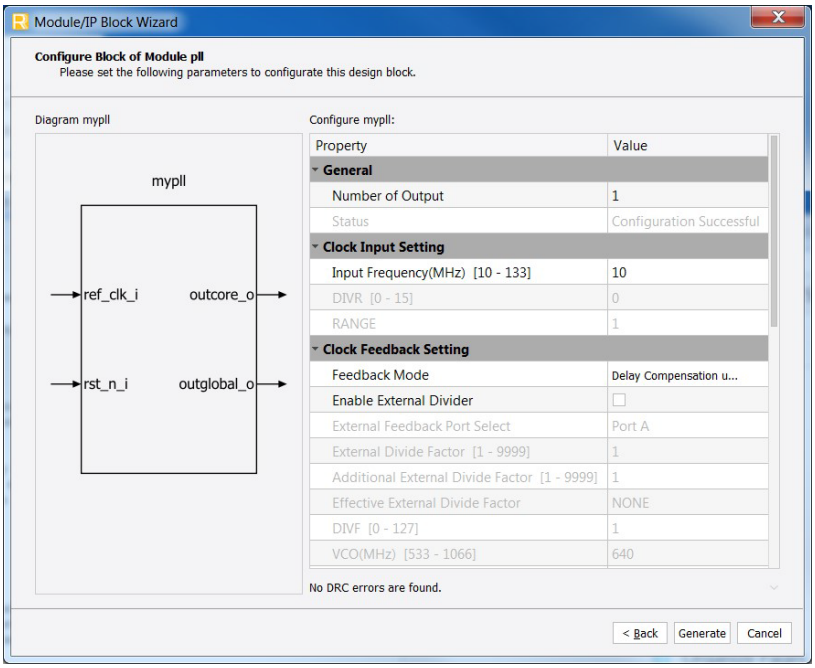
Radiant software has a dedicated primitive, SP256K, for a single port block RAM.

```
SP256K u_SP256K (
  .AD       (addr[13:0]),  // I, 14-bit address
  .DI       (wdata),       // I, 16-bit write data
  .MASKWE   (4'b1111),     // I, 4-bit nibble mask control
  .WE       (we),          // I, write(H)/read(L) mode select
  .CS       (cs),          // I, memory enable
  .CK       (clk),         // I, clock
  .STDBY    (1'b0),        // I, low leakage standby mode
  .SLEEP    (1'b0),        // I, periphery shutdown sleep mode
  .PWROFF_N (1'b1),        // I, no memory retention turn off
  .DO       (rdata)        // O, 16-bit read data
);
```

PLL

A PLL module can be configured and generated from Radiant IP Catalog. PLL is located under Architecture_Modules category in IP Catalog. Once selected, a user instance name can be provided, and the tool allows the user to

configure the PLL settings and generate a PLL module. The figure below shows the PLL configuration GUI with the user selected instance name “mypll”:



All options in iCEcube2 PLL generator tool are also available in the PLL GUI of Radiant IP Catalog. For detail usage, see TN1251 - *iCE40 sysCLOCK PLL Design and Usage Guide*.

Once configuration is completed, the PLL module can be generated by clicking the Generate button, and the generate module is inserted to the project in the form of .ipx entry. From there, you can find all necessary source, constraint and configuration files generated from the tool. A right-click on the .ipx file allows you to create and copy a Verilog/VHDL instantiation template or VHDL component declaration.

The following is an example of Verilog instantiation template pasted to the user source editor.

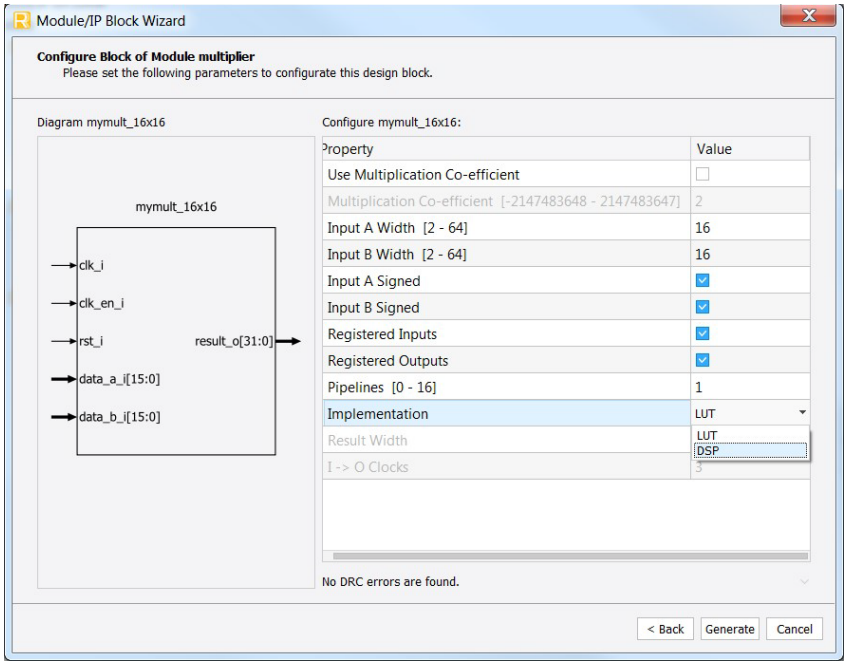
```
mypll __(.ref_clk_i( ), .rst_n_i( ), .lock_o( ), .outcore_o( ),
.outglobal_o( ));
```

Providing an instance name and make connections to the ports will complete an instantiation of the generated PLL module.

DSP Functions

DSP functions can also be generated from IP Catalog. All available DSP functions are located under the Arithmetic_Modules category. The module IP generation and usage for the DSP functions is similar to that of PLL.

For DSP functions instantiated using IP catalog, make sure to select the appropriate implementation for best resource utilization. Implementation can be “DSP” or “LUT”.



When a DSP primitive needs to be instantiated, it is suggested to instantiate a pmi_dsp instead of instantiating a MAC16 primitive directly. The pmi_dsp is a wrapper for MAC16, which allows buses on the signals. It is much more convenient to use than MAC16 which has all ports individually listed.

To access the pmi_dsp instantiation template:

1. Open Source Editor with a source file that will instantiate the pmi_dsp function.
2. Open Template Editor from View > Template Editor.
3. Select “dsp” from Verilog|VHDL > PMI Templates.
4. Right click on dsp and select Insert to Text. If an external editor is being used, copy the preview content at the bottom of Template Editor and paste to the external editor.

Below is the instantiation template for pmi_dsp provided by Template Editor:

```
pmi_dsp
#(
 .NEG_TRIGGER              ( ), //"0b0"|"0b1"
 .A_REG                    ( ), //"0b0"|"0b1"
 .B_REG                    ( ), //"0b0"|"0b1"
 .C_REG                    ( ), //"0b0"|"0b1"
 .D_REG                    ( ), //"0b0"|"0b1"
 .TOP_8x8_MULT_REG         ( ), //"0b0"|"0b1"
 .BOT_8x8_MULT_REG         ( ), //"0b0"|"0b1"
 .PIPELINE_16x16_MULT_REG1 ( ), //"0b0"|"0b1"
 .PIPELINE_16x16_MULT_REG2 ( ), //"0b0"|"0b1"
 .TOPOUTPUT_SELECT         ( ), //"0b00"|"0b01"|"0b10"|"0b11"
```

```
 .TOPADDSUB_LOWERINPUT     ( ), //"0b00"|"0b01"|"0b10"|"0b11"
 .TOPADDSUB_UPPERINPUT     ( ), //"0b0"|"0b1"
 .TOPADDSUB_CARRYSELECT    ( ), //"0b00"|"0b01"|"0b10"|"0b11"
 .BOTOUTPUT_SELECT         ( ), //"0b00"|"0b01"|"0b10"|"0b11"
 .BOTADDSUB_LOWERINPUT     ( ), //"0b00"|"0b01"|"0b10"|"0b11"
 .BOTADDSUB_UPPERINPUT     ( ), //"0b0"|"0b1"
 .BOTADDSUB_CARRYSELECT    ( ), //"0b00"|"0b01"|"0b10"|"0b11"
 .MODE_8x8                 ( ), //"0b0"|"0b1"
 .A_SIGNED                 ( ), //"0b0"|"0b1"
 .B_SIGNED                 ( )  //"0b0"|"0b1"
 <your_inst_label> (
 .CLK        ( ), // I:
 .CE         ( ), // I:
 .C          ( ), // I:
 .A          ( ), // I:
 .B          ( ), // I:
 .D          ( ), // I:
 .AHOLD      ( ), // I:
 .BHOLD      ( ), // I:
 .CHOLD      ( ), // I:
 .DHOLD      ( ), // I:
 .IRSTTOP    ( ), // I:
 .IRSTBOT    ( ), // I:
 .ORSTTOP    ( ), // I:
 .ORSTBOT    ( ), // I:
 .OLOADTOP   ( ), // I:
 .OLOADBOT   ( ), // I:
 .ADDSUBTOP  ( ), // I:
 .ADDSUBBOT  ( ), // I:
 .OHOLDTOP   ( ), // I:
 .OHOLDBOT   ( ), // I:
 .CI         ( ), // I:
 .ACCUMCI    ( ), // I:
 .SIGNEXTIN  ( ), // I:
 .O          ( ), // O:
 .CO         ( ), // O:
 .ACCUMCO    ( ), // O:
 .SIGNEXTOUT ( )  // O:
);
```

Oscillator Functions

To use the internal oscillators in iCE40UP, instantiate HSOSC for high speed or LSOSC for low speed. HSOSC provides a 48MHz, 24MHz, 12MHz or 6MHz oscillation frequency output while LSOSC generates a 10KHz frequency out. Use Template Editor to get an instantiation template. Instantiation examples are shown below:

```
HSOSC
#(
 .CLKHF_DIV ("0b00")
 // 0b00=48MHz(default), 0b01=24MHz, 0b10=12MHz, 0b11=6MHz
) u_HSOSC (
 .CLKHFPU (hsck_pu), // I: power up, active high
 .CLKHFEN (hsck_en), // I: output enable, active high
 .CLKHF   (hsclk)    // O: high speed clock output
```

```
);
LSOSC u_LSOSC (
 .CLKLFPU (lsck_pu), // I: power up, active high
 .CLKLFEN (lsck_en), // I: output enable, active high
 .CLKLF   (lsclk)    // O: 10KHz clock output
);
```

HSOSC1P8V and LSOSC1P8V are instantiated the same way as shown below:

```
HSOSC1P8V
#(
 .CLKHF_DIV ("0b00")
 // 0b00=48MHz(default), 0b01=24MHz, 0b10=12MHz, 0b11=6MHz
) u_HSOSC1P8V (
 .CLKHFPU (hsck_pu), // I: power up, active high
 .CLKHFEN (hsck_en), // I: output enable, active high
 .CLKHF   (hsclk)    // O: high speed clock output
);

LSOSC1P8V u_LSOSC1P8V (
 .CLKLFPU (lsck_pu), // I: power up, active high
 .CLKLFEN (lsck_en), // I: output enable, active high
 .CLKLF   (lsclk)    // O: 10KHz clock output
);
```

RGB LED Drivers

To use the RGB LED drivers, instantiate RGB and provide the current mode and current level parameters. Unlike iCEcube2, Radiant software has “RGB” primitive that includes the required IO buffers, and no additional IO buffer instantiation is necessary. There are other primitives having RGB as part of the name. They are generally for software use and not for user instantiation.

```
RGB
#(
 .CURRENT_MODE ("0"),        //"0":full,"1":half
 .RGB0_CURRENT ("0b111111"), //"0b000000":0mA,"0b000001":4mA,
 .RGB1_CURRENT ("0b111111"), //"0b000011":8mA,"0b000111":12mA,
 .RGB2_CURRENT ("0b111111")  //"0b001111":16mA,"0b011111":20mA,
                             //"0b111111":24mA
) u_RGB (
 .CURREN   (curr_en),   // I: reference current enable
 .RGBLEDEN (leddrv_en), // I: RGB driver enable
 .RGB0PWM  (pwm0),      // I: RGB0 input
 .RGB1PWM  (pwm1),      // I: RGB1 input
 .RGB2PWM  (pwm2),      // I: RGB2 input
 .RGB2     (led2),      // O: RGB2 LED output
 .RGB1     (led1),      // O: RGB1 LED output
 .RGB0     (led0)       // O: RGB0 LED output
);
```

RGB1P8V is instantiated the same way as shown below:

```
RGB1P8V
#(
 .CURRENT_MODE ("0"),        //"0": full,"1": half
```

```
 .RGB0_CURRENT ("0b111111"), //"0b000000":0mA,"0b000001":4mA,
 .RGB1_CURRENT ("0b111111"), //"0b000011":8mA,"0b000111":12mA,
 .RGB2_CURRENT ("0b111111")  //"0b001111":16mA,"0b011111":20mA,
                             //"0b111111":24mA
) u_RGB1P8V (
 .CURREN   (curr_en),   // I: reference current enable
 .RGBLEDEN (leddrv_en), // I: RGB driver enable
 .RGB0PWM  (pwm0),      // I: RGB0 input
 .RGB1PWM  (pwm1),      // I: RGB1 input
 .RGB2PWM  (pwm2),      // I: RGB2 input
 .RGB2     (led2),      // O: RGB2 LED output
 .RGB1     (led1),      // O: RGB1 LED output
 .RGB0     (led0)       // O: RGB0 LED output
);
```

RGB PWM Generator

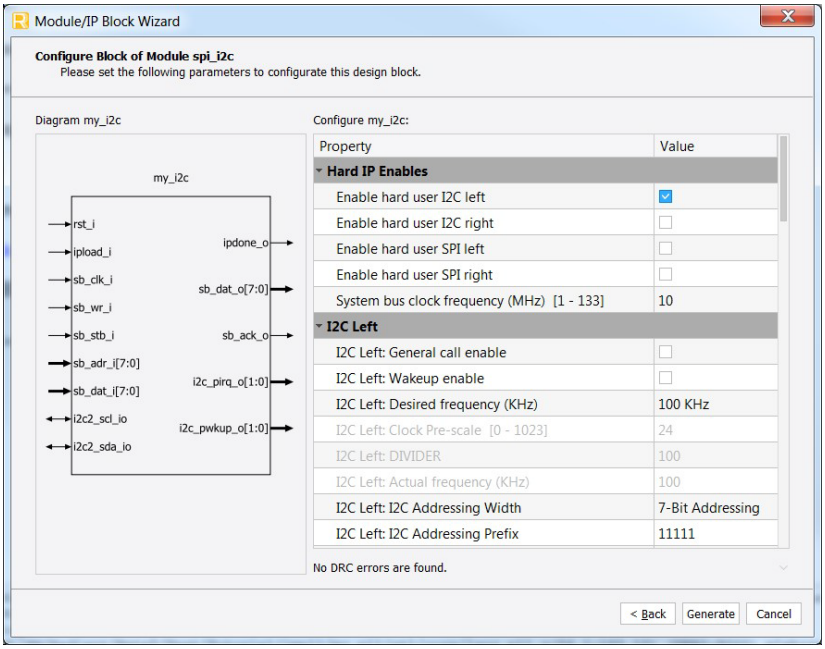
To add a PWM generator function for the RGB LED drivers to your design, instantiate RGBPWM as shown below:

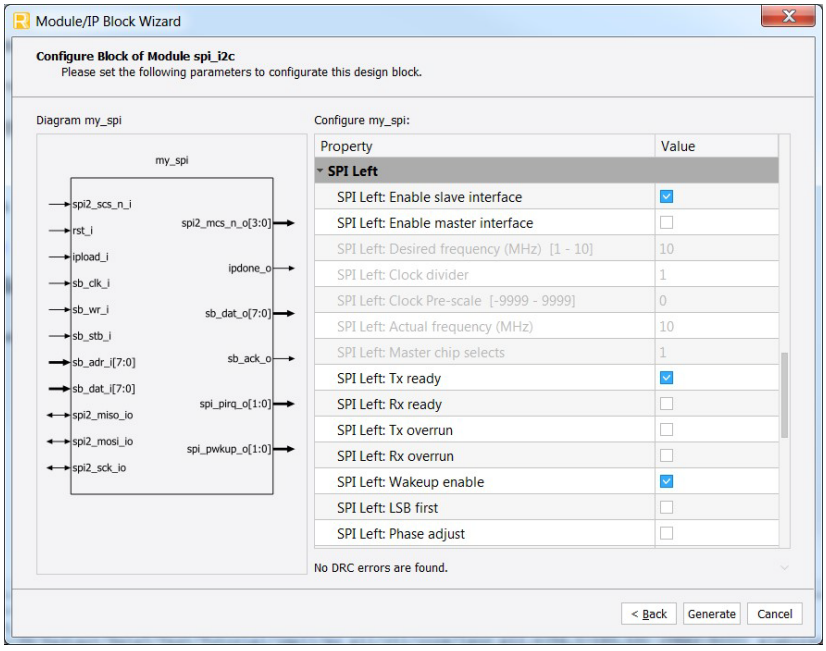
```
RGBPWM u_RGBPWM (
  .LEDDCS    (ledcs),     // I: chip select
  .LEDDCLK   (ledclk),    // I: clock input
  .LEDDDAT7  (ledin[7]),  // I: data input 7
  .LEDDDAT6  (ledin[6]),  // I: data input 6
  .LEDDDAT5  (ledin[5]),  // I: data input 5
  .LEDDDAT4  (ledin[4]),  // I: data input 4
  .LEDDDAT3  (ledin[3]),  // I: data input 3
  .LEDDDAT2  (ledin[2]),  // I: data input 2
  .LEDDDAT1  (ledin[1]),  // I: data input 1
  .LEDDDAT0  (ledin[0]),  // I: data input 0
  .LEDDADDR3 (ledadr[3]), // I: addr input 3
  .LEDDADDR2 (ledadr[2]), // I: addr input 2
  .LEDDADDR1 (ledadr[1]), // I: addr input 1
  .LEDDADDR0 (ledadr[0]), // I: addr input 0
  .LEDDDEN   (leden),     // I: data enable
  .LEDDEXE   (ledrun),    // I: enable blinking sequence
  .PWMOUT2   (pwmo[2]),   // O: pwm output 2
  .PWMOUT1   (pwmo[2]),   // O: pwm output 1
  .PWMOUT0   (pwmo[2]),   // O: pwm output 0
  .LEDDON    (ledon)      // O: led on indicator
);
```

I2C and SPI Function

An I2C or SPI module can be configured and generated from Radiant IP Catalog. The I2C/SPI module IP is used to generate an I2C/SPI module and located under Architecture_Modules category in IP Catalog. It has an equivalent set of I2C/SPI options to iCEcube2 I2C/SPI Module Generator.

Once selected, a user instance name can be provided, and the tool allows the user to configure the I2C/SPI settings by selecting “Enable hard I2C left”, “Enable hard I2C right”, “Enable hard SPI left” and/or “Enable hard I2C right” options. The figure below shows the I2C/SPI configuration GUI with the user selected instance name “my_i2c” and “my_spi” respectively:





Once configuration is completed, the selected I2C and/or SPI functions can be generated by clicking the Generate button, and the generate module is inserted to the project in the form of .ipx entry. From there, you can find all necessary source, constraint and configuration files generated from the tool.

A right-click on the .ipx file allows you to create and copy a Verilog/VHDL instantiation template or VHDL component declaration.

The following is an example of Verilog instantiation template pasted to the user source editor when both an I2C and a SPI functions are added:

```
my_i2c_spi __(.i2c2_scl_io( ), .i2c2_sda_io( ),
 .spi2_miso_io( ), .spi2_mosi_io( ), .spi2_sck_io( ),
 .spi2_scs_n_i( ), .spi2_mcs_n_o( ), .rst_i( ),
 .ipload_i( ), .ipdone_o( ), .sb_clk_i( ),
 .sb_wr_i( ), .sb_stb_i( ), .sb_adr_i( ), .sb_dat_i( ),
 .sb_dat_o( ), .sb_ack_o( ), .i2c_pirq_o( ),
 .i2c_pwkup_o( ), .spi_pirq_o( ), .spi_pwkup_o( ));
```

Providing an instance name and make connections to the ports will complete an instantiation of the generated I2C/SPI module.

Warm Boot Function

To load a different configuration image during regular operation, instantiate WARMBOOT in Radiant software as shown below:

```
WARMBOOT u_WARMBOOT (
  .S1   (sel1),  // I: configuration image selection input 1
  .S0   (sel0),  // I: configuration image selection input 1
  .BOOT (boot)   // I: Reconfigure trigger input
);
```

Registers

Difference types of register functions are supported in Radiant software, and they are generally inferred by the synthesis tools from the user RTL source expressions. In case a specific type of register function is to be used via a primitive instantiation, the following user register primitives can be instantiated:

FD1P3BZ – Asynchronous Set, Active-High Clock Enable

```
FD1P3BZ u_FD1P3BZ(
  .D  (din);  //I: data in
  .CK (clk);  //I: clock input
  .SP (ce);   //I: clock enable
  .PD (prst); //I: async preset input
  .Q  (dout)  //O: data out
  );
```

FD1P3DZ – Asynchronous Reset, Active-High Clock Enable

```
FD1P3DZ u_FD1P3DZ(
```

```
  .D  (din);  //I: data in
  .CK (clk);  //I: clock input
  .SP (ce);   //I: clock enable
  .CD (rst);  //I: async reset input
  .Q  (dout)  //O: data out
  );
```

FD1P3IZ – Synchronous Reset, Active-High Clock Enable

To reset the register, SP (clock enable) input must be enabled first.

```
FD1P3DZ u_FD1P3DZ(
  .D  (din);  //I: data in
  .CK (clk);  //I: clock input
  .SP (ce);   //I: clock enable
  .CD (rst);  //I: sync reset input
  .Q  (dout)  //O: data out
  );
```

FD1P3JZ – Synchronous Set, Active-High Clock Enable

To set the register, SP (clock enable) input must be enabled first.

```
FD1P3JZ u_FD1P3JZ(
  .D  (din);  //I: data in
  .CK (clk);  //I: clock input
  .SP (ce);   //I: clock enable
  .PD (prst); //I: sync preset input
  .Q  (dout)  //O: data out
  );
```

iCEcube2 to Radiant Software I/O Primitive Attribute Translation

Some Radiant software I/O primitives have equivalent functions to iCECube2 primitives. The following tables provide the attribute translation for iCECube2 primitives SB_IO and SB_IO_OD to the Radiant software equivalent primitives.

Table 2: Attributes from SB_IO and SB_IO_OD to Radiant Software Equivalent

iceCube2	Radiant Software
PIN_TYPE	See Input Pin Function & Output Pin Function table.
NEG_TRIGGER	Negate INCLK or OUTCLK of IOL_B.
IO_STANDARD	Use Device Constraint Editor -> Pin -> IO_TYPE
PULLUP	Use Device Constraint Editor -> Pin -> PULLMODE. Only applicable to SB_IO.

The following table show a comparison of iCECube2 and Radiant software input pin types and their functions

Table 3: Input Pin Function

iCECube2	PIN_TYPE[1:0]	Radiant Software
PIN_INPUT	01	Use IB for SB_IO; BB_OD for SB_IO_OD.
PIN_INPUT_LATCH	11	Use IB for SB_IO; BB_OD for SB_IO_OD. Feed to IOL_B's PADDI, and use HOLD signal for latch. Then use DI0.
PIN_INPUT_REGISTERED	00	Use IB for SB_IO; BB_OD for SB_IO_OD. Feed to IOL_B's PADDI, & provide INCLK. Registered input is DI0.

Table 3: Input Pin Function (Continued)

iCECube2	PIN_TYPE[1:0]	Radiant Software
PIN_INPUT_REGISTERED_LATCH	11	See PIN_INPUT_REGISTERED. Include HOLD signal for latching.
PIN_INPUT_DDR	00	See PIN_INPUT_REGISTERED. Also use DI1 that is clocked on falling edge of INCLK.

The following table show a comparison of iCECube2 and Radiant software output pin types and their functions.

Table 4: Output Pin Function

iCECube2	PIN_TYPE[5:2]	Radiant Software
PIN_NO_OUTPUT	0000	N/A
PIN_OUTPUT	0110	Use OB; use BB_OD for SB_IO_OD.
PIN_OUTPUT_TRISTATE	1010	Use OBZ_B; use BB_OD for SB_IO_OD.
PIN_OUTPUT_ENABLE_REGISTERED	1110	Feed tristate signal to IOL_B's IOLTO & provide OUTCLK. Then feed PADDT to OBZ_B/BB_OD port T.
PIN_OUTPUT_REGISTERED	0101	Feed signal to IOL_B's DO0 & provide OUTCLK. Then feed PADDO to OB/BB_OD.
PIN_OUTPUT_REGISTERED_ENABLE	1001	See PIN_OUTPUT_REGISTERED, but use OBZ_B instead of OB. (BB_OD for SB_IO_OD).
PIN_OUTPUT_REGISTERED_ENABLE_REGISTERED	1101	See PIN_OUTPUT_ENABLE_REGISTERED & PIN_OUTPUT_REGISTERED. Do both.
PIN_OUTPUT_DDR	0100	See PIN_OUTPUT_REGISTERED, but provide DO1 for falling edge of OUTCLK.
PIN_OUTPUT_DDR_ENABLE	1000	See PIN_OUTPUT_DDR. Use OBZ_B instead of OB.
PIN_OUTPUT_DDR_ENABLE_REGISTERED	1100	See PIN_OUTPUT_DDR_ENABLE & PIN_OUTPUT_ENABLE_REGISTERED. Do both.
PIN_OUTPUT_REGISTERED_INVERTED	0111	See PIN_OUTPUT_REGISTERED. Negate OUTCLK.
PIN_OUTPUT_REGISTERED_ENABLE_INVERTED	1011	See PIN_OUTPUT_REGISTERED_ENABLE. Negate OUTCLK.
PIN_OUTPUT_REGISTERED_ENABLE_REGISTERED_INVERTED	1111	See PIN_OUTPUT_REGISTERED_ENABLE_REGISTERED. Negate OUTCLK.

Revision History

The following table gives the revision history for this document.

Date	Version	Description
04/20/2018	1.0.1	Added Migration Examples section to Chapter 1.
02/20/2018	1.0	Initial Release.