

Technology Abstraction Eases Silicon Intellectual Property Portability

Jason Pecor
Matt Weber

Silicon Logic Engineering, Inc.

jason@siliconlogic.com
Matthew.D.Weber@ieee.org

One unique attribute of Silicon Intellectual Property (SIP) offerings is the ability to implement the IP in multiple technologies without requiring extensive effort in each step of the implementation flow. However, for some complex, high-speed SIP solutions, carefully analyzed custom digital circuits are required that necessitate hand instantiation of specific technology cells. This scenario can lead to a number of challenges when porting from one technology to another.

This paper will discuss a Technology Abstraction methodology that reduces the impact to design, synthesis, and verification that can occur when retargeting an IP solution to a new vendor technology. It is based on a layer of cell library abstraction used within the design that propagates to synthesis and design verification allowing for smoother transitions to different silicon solutions.

TABLE OF CONTENTS

1 INTRODUCTION.....	3
2 TECHNOLOGY ABSTRACTION DEFINED	4
2.1 OVERVIEW	4
2.2 MULTIPLE MAP FILES	5
<i>Generic Map File</i>	5
<i>Technology Specific Map File</i>	5
<i>Map File Examples</i>	5
3 BENEFITS FOR DESIGN AND SYNTHESIS	7
3.1 ELIMINATION OF MULTIPLE DESIGN FILE COPIES.....	7
3.2 FUNCTIONAL BEHAVIOR AVAILABLE BEFORE COMPLETION OF ANALYSIS.....	7
3.3 EASILY DEFINABLE FEATURES AND REQUIREMENTS	8
3.4 GREATER FLEXIBILITY	9
3.5 LIBRARY MAP FILE AND SYNTHESIS.....	10
4 BENEFITS FOR DESIGN VERIFICATION AND SIMULATION	12
4.1 BUILD LIST REFERENCES A SINGLE DESIGN FILE	12
4.2 FLEXIBLE BEHAVIORAL MODELING	12
<i>Embedding #<delays> in specific circuits</i>	12
<i>Modeling incomplete gate solutions</i>	13
4.3 STATE INITIALIZATION	13
5 RECOMMENDATIONS.....	14
5.1 TECHNOLOGY DEPENDENT SOURCE ISOLATION	14
5.2 COMMON SETUP FILE.....	15
5.3 ROBUST REVISION CONTROL	15
6 CONCLUSION.....	15

1 Introduction

Silicon Intellectual Property (SIP) is IP targeted for implementation in FPGAs or ASICs. The most desirable SIP solutions are those that can be easily implemented in any ASIC vendor technology. As a result, most SIP designs are delivered in the form of register transfer level (RTL) source code or generic gate-level cells. This provides the most flexibility in deciding where and when to use the IP.

However, there are cases where the complexity of the SIP solution necessitates specific custom circuits be designed using carefully chosen gates from the target technology library. For example, a high speed interconnect interface such as SPI-4.2 demands careful design consideration when implementing the dynamic skew recovery logic and requires control of the final library cells that will make up the circuitry. Hand-placed, vendor specific circuit implementations like this can add unique challenges to the development process and maintenance of a SIP offering.

This paper addresses a library abstraction methodology that helps to reduce the overhead associated with developing and maintaining SIP that requires technology specific gate selection and instantiation. The methodology is based on a layer of library abstraction used within the design source that propagates to synthesis, design verification, and other process steps, allowing for smoother transitions to additional silicon solutions.

2 Technology Abstraction Defined

2.1 Overview

The technology abstraction methodology combines a way to specify generic cell instances with a mechanism for mapping those generic instances to specific technology library cells. By using this layer of abstraction for hand-instantiated gates, it becomes very easy to modify cell selections for a given technology and eases the process of porting to new technologies.

The foundation of the methodology is the Library Map File. This file contains the definitions for all gate types, cells, and functional blocks that require hand instantiation in the design along with their specific technology mappings. The Library Map contents can include definitions for simplistic cells like buffers and inverters as well as design elements that contain multiple gates or logic functions. The map file is a simple verilog file, and may look something like the following example:

```
/* *****  
 * File name      : SLE_LIBRARY_MAP.v  
 * *****  
 *  
 * Description   : Simple wrappers around commonly instantiated  
 *                 cells to allow easier porting to new  
 *                 technologies.  
 *                 Just replace the behavioral code here with  
 *                 instantiations of elements from the library  
 *                 to be used.  
 * *****/  
module SLE_OR (Z,A, B);  
 .  
 . <module body>  
 .  
endmodule // SLE_OR  
  
module SLE_INVERT (Z,A);  
 .  
 . <module body>  
 .  
endmodule // SLE_INVERT  
  
module SLE_DELAY_MUX (Z,D0,D1,S);  
 .  
 . <module body>  
 .  
endmodule // SLE_DELAY_MUX
```

Once it has been created, the map file is used by all source code that requires hand instantiated gates, and rather than vendor-specific cells, the generic cells defined in the map file become the instantiated elements.

2.2 Multiple Map Files

One of the requirements for this methodology is the existence of multiple map files. There will be at least two files: one generic map file and one technology specific file. However, there could be as many tech-specific files as there are targeted vendor technologies.

Generic Map File

The generic map file contains behavioral or RTL modeling of the desired functionality. This version of the map file is useful early on in the development cycle, before the technology specific gates are selected, providing behavioral functionality for simulation and testing. In addition, the generic map file can be used as the source for RTL regressions with logical equivalency methods bridging the gap between RTL modeling and final gate-level function.

Technology Specific Map File

The technology specific map file contains the same cells as the generic map file with cell definitions implemented as vendor library elements instead of RTL or behavioral code. The tech-specific cell choices are based on functional and electrical requirements and determined by library specification data and SPICE simulation results. This file is then used for synthesis and technology specific simulation.

Map File Examples

The following examples show a simple abstracted gate as it would appear in a generic map file and two other vendor technology files. The gate functionality used for this example is a simple balanced MUX.

```
module SLE_BAL_MUX (Z,D0,D1,SD); // Generic Implementation

    output Z;
    input D0;
    input D1;
    input SD;

    assign Z = SD ? D1 : D0;

endmodule // SLE_BAL_MUX
```

```
module SLE_BAL_MUX (Z,D0,D1,SD); // Technology A Implementation

    output Z;
    input D0;
    input D1;
    input SD;
    wire ddr_int_a, ddr_int_b;

    TECHA_NAND_INVA ddr_inv_nand (.AN(SD), .B(D0), .Y(ddr_int_a));
    TECHA_NAND ddr_noninv_nand (.A(SD), .B(D1), .Y(ddr_int_b));
    TECHA_NAND ddr_cmb (.A(ddr_int_a), .B(ddr_int_b),.Y(Z));

endmodule // SLE_BAL_MUX
```

```
module SLE_BAL_MUX (Z,D0,D1,SD); // Technology B Implementation

    output Z;
    input  D0;
    input  D1;
    input  SD;

    TECHB_BAL_MUX U1 (.Z(Z),.D0(D0),.D1(D1),.SD(SD));

endmodule // SLE_BAL_MUX
```

As you can see, the MUX is represented with a simple assign statement in the generic map file. The tech-specific versions, however, are built using vendor library elements. Notice, also, that for technology A, the MUX was built using 3 components from the library, but technology B already had a balanced MUX that could simply be dropped in place.

If implemented from the onset of a project, this type of library abstraction can benefit a number of steps throughout the development effort. However, the most noticeable efficiency gains come in the areas of design/synthesis and verification.

3 Benefits for Design and Synthesis

3.1 Elimination of multiple design file copies

The key benefit realized from designing using verilog RTL is technology independence. A design can simply be described functionally, and a synthesis tool is used to choose the technology specific gates that are used to implement that functionality. However, as mentioned before, there are times when more control over the technology specific gates is desired. For example, specific gates may be necessary for:

1. A metastability hardened flip-flop for asynchronous clock domain crossings.
2. A clock divider flip-flop that ensures good duty cycle or rise time characteristics.
3. Clock generation logic or logic to ensure glitch-less clock gating.
4. Ensuring minimum output skew by using the same cell, including drive strength, to drive the ASIC I/Os of an output bus
5. Coercing instantiation of negative edge triggered flip-flop (which are generally not inferred by synthesis tools)

When the technology specific gates required for these functions are allowed to be instantiated within the ordinary verilog RTL files, design maintenance and management can become increasingly difficult. If the design is mapped to several technologies, each RTL file which instantiates a technology specific gate must have a unique copy made for each technology. Some of the technology specific gates, such as a metastability hardened flip flop, are likely used in many areas of the design, and the instantiation needs to be updated in each of the areas where it is used. Also, if any design changes are made to the non-technology specific areas of an RTL file, this change must be copied to all versions of that file. Changes like these are inherently error prone and add significant complexity to the revision control system.

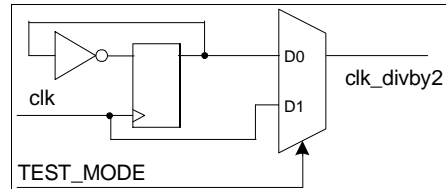
By using the library map file described above, all technology specific gates are pulled together into a single file. Mapping the design to a new technology no longer requires the modification of many design files. Instead, only the library map file needs to be updated. Also, because the basic functionality of the design has not been copied to technology specific versions of a design file, changes to that functionality only need to be made in one place.

3.2 Functional behavior available before completion of analysis

Mapping a design to a new technology typically requires some analysis to choose the gates that are best for a particular function such as asynchronous clock domain crossings. If the gates for this function are coded into the RTL source files, then all simulation, synthesis, placement, and static timing work cannot be started until the new gates are chosen and the RTL files modified. On the other hand, starting with a library map file which uses synthesizable, behavioral code for the required functions, allows the simulation and synthesis work to begin much earlier. Although the resulting netlist does not have the correct gates in some areas, the rest of the netlist will be accurate and can be used for floorplanning, clock planning, and timing analysis.

3.3 Easily definable features and requirements

Sometimes the cell or logic block implementations require different ports in different technologies. In these cases, 'defines can be used to provide the proper connectivity. For example, many test methodologies require that clock dividers be followed by a bypass MUX as shown in the following figure.



This bypass mux allows the clock to the divby2 domain to be controllable during test. Since not all technologies and test methodologies require this structure, and 'ifdef is used to connect the TEST_MODE signal only when it is required. The normal clock divider code is:

```
module SLE_CLKDIV_FLOP (Q,CLK);
    output Q;
    input  CLK;
    wire  Q_n;

    INVERT sle_dt_clkdivinv (.Z(Q_n),.A(Q));
    DFF    sle_dt_clkdivflop (.Q(Q),.D(Q_n),.CK(CLK));
endmodule //SLE_CLKDIV_FLOP
```

While a clock divider that includes the testability MUX would look like this:

```
module SLE_CLKDIV_FLOP (Q,CLK,TEST_MODE);
    output Q;
    input  CLK;
    input  TEST_MODE;

    wire  Qint;
    wire  Qint_n;

    INVERT  sle_dt_clkdivinv (.Y(Qint_n), .A(Qint));
    DFF_TEST sle_dt_clkdivflop(.Q(Qint), .QZ(), .D(Qint_n),
                               .CLK(CLK), .SCAN(1'b0), .SD(1'b0));
    MUX21   sle_dt_clkdivmux (.Y(Q), .A(Qint),
                               .B(CLK), .S(TEST_MODE));
endmodule // SLE_CLKDIV_FLOP
```

The additional input to the SLE_CLKDIV_FLOP module is handled with an 'ifdef at the next higher level of hierarchy.

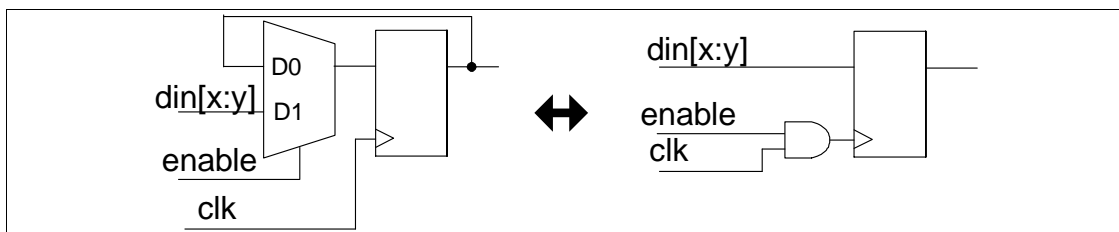
```
SLE_CLKDIV_FLOP clkdiv_reg (.Q(rdclk_divby2),
                             `ifdef CLKDIV_TEST
                               .TEST_MODE(TEST_MODE),
                             `endif
                             .CLK(xi_rdclk) );
```

Although adding a number of 'ifdefs like this begins to complicate the code of the upper level module, it is still much easier to maintain than having separate versions of that upper level module for each technology.

3.4 Greater Flexibility

The concept of a Library Map file can also be extended to provide even greater flexibility in the design. Instead of coding all the registers in the design with an @(posedge clk) block, a generic flip-flop model can be instantiated. If this model includes a reset input, the entire design can be changed from a synchronous reset strategy to an asynchronous strategy by simply modifying the library map file. Without the map file, every RTL file in the design would need to be modified to make this kind of a change.

Similarly, if the flip-flop model includes an enable input, the design can easily be converted between a muxed feedback style and a gated clock style of design as shown below. (More can be learned about clock gating from Darren Jones' SNUG Boston 2002 Paper about the subject).



When using the library map file for multi-bit registers, the coding effort can be significantly reduced through the smart use of parameters and arrays of instances. The width of the data input and output can be parameterized to make it easier to use the flip-flop model on multi-bit registers. In a gated clock design, there will typically be only one clock gate for the entire bus, and a flip-flop needs to be instantiated for each bit of the bus. This instantiation can be greatly simplified by using the array of instances construct.

```

module SLE_FLOP (Q,CLK, D, EN);

    parameter WIDTH =1;

    output [WIDTH-1:0] Q;
    input      CLK;
    input [WIDTH-1:0] D;
    input      EN;
    wire      CLK_EN;

    AND sle_dt_clkgate      (.Z(CLK_EN),.A(CLK),.B(EN));
    DFF sle_dt_flop [WIDTH-1:0](.Q(Q),.D(D),.CLK(CLK_EN));

endmodule      // SLE_FLOP

```

A multi-bit register can then be created by simply specifying the width of the register as a parameter setting in the instantiation.

```
SLE_FLOP #(8) count_reg (.Q    (count[7:0]),
                        .CLK  (clk),
                        .D    (next_count[7:0]),
                        .EN   (count_valid));
```

The array of instances construct has been part of the Verilog language since it was standardized in 1995. Design Compiler (with the Presto HDL reader) and VCS both support the construct. Unfortunately, Formality support is inconsistent.

For example, Formality can not link the above code for SLE_FLOP , but it can link the following code in which an extra layer of hierarchy has been added.

```
module SLE_FLOP (Q,CLK, D, EN);

    parameter WIDTH = 1;
    output [WIDTH-1:0] Q;

    input          CLK;
    input [WIDTH-1:0] D;
    input          EN;
    wire           CLK_EN;

    AND            sle_dt_clkgate          (.Z(CLK_EN),.A(CLK),.B(EN));
    SLE_1FLOP sle_dt_flop [WIDTH-1:0] (.Q(Q),.D(D),.CLK(CLK_EN));

endmodule        // SLE_FLOP

module SLE_1FLOP (Q,CLK, D);

    input  CLK;
    input  D;
    output Q;

    DFF    sle_dt_flop (.Q(Q),.D(D),.CK(CLK));

endmodule        // SLE_1FLOP
```

We hope these Formality problems will be resolved soon.

3.5 Library map file and synthesis

Synthesis using the library map file is very straightforward. The map file is read into Design Compiler along with the RTL for the design being synthesized:

```
analyze -f verilog [format "%s%s" $RTL_DIR {Library_Map.v}]
read_verilog [format "%s%s%s" $RTL_DIR $DESIGN_NAME {.v}]
```

Reading files into Design Compiler is basically a two-step process. The “analyze” step reads the RTL and does syntax checking and Synopsys rule checking. The “elaborate” step sets any parameter values and translates the design into GTECH. The “read_verilog” command does both “analyze” and “elaborate” steps with a single

command. When a module is parameterized like SLE_FLOP above, the RTL for the Library Map file should be read with the analyze command rather than the “read_verilog” command to ensure that the parameters passed from the upper level module are used instead of the default value given in the parameter declaration.

It is very important to “don’t touch” all mapped cells for the synthesis runs to avoid the risk of swapping out desired gates. One recommended approach is to use a common naming structure such that all mapped cells could be easily “don’t touched” by using wildcard naming in the synthesis commands.

4 Benefits for Design Verification and Simulation

In addition to the advantages in the design and synthesis domains, the use of an abstracted library also brings a number of benefits to the simulation environment when verifying designs that target multiple technologies.

4.1 Build list references a single design file

Since the library map file retains the same name regardless of which technology it is referencing, the simulation build scripts need not support multiple file names for the different implementations. A single persistent file name can be assumed to exist in each unique source code repository. For example, the following VCS[®] command line option could be used for every simulation build, regardless of the targeted vendor:

```
vcs ... -v <source_path>/LIB_MAP.v ...
```

Of course, it is still necessary to provide path information to the appropriate DUT source location. However, unique path information is typically required as a simulator setup parameter and can be easily encapsulated into the simulation control scripts.

4.2 Flexible behavioral modeling

Since the custom cells and logic are encapsulated in a single file, it is easy to modify and update the behavioral implementation of the cells. This flexibility is very handy when periodic modification and tweaks to a specific implementation are required. Two scenarios when this is useful are gate delay modeling and behavioral modeling for incomplete gate solutions.

Embedding #<delays> in specific circuits

Sometimes, the very reason a particular library element is selected for hand instantiation in a design is that the cell has been carefully analyzed and characterized for its timing performance. In addition, some cells may be chosen based not on the fact that they are the fastest solution but rather that they have a very specific propagation delay that meets the criteria of a given target frequency. The library map file provides the ability to model these delays when running in RTL.

Take the following delay block as an example. In this case, the library cells that will be selected will be chosen such that the delay through the block will meet specific timing requirement. Here is the generic/RTL from the map file that can be used to represent what will become the final gates:

```

module SLE_DELAY_STEP (Z,QR,QF,D,CLK);

    output Z,QR,QF;
    input  D;
    input  CLK;
    reg    QR,QF;

    assign #(0.045:0.09:0.135) Z=~D;
    always @(posedge CLK) QR <= D;
    always @(negedge CLK) QF <= D;

endmodule // SLE_DELAY_STEP

```

The timing information can also be added once the specific vendor cells have been chosen to facilitate RTL simulations that include the real gate level circuit solutions:

```

module SLE_DELAY_STEP (Z,QR,QF,D,CLK);

    output Z,QR,QF;
    input  D;
    input  CLK;
    wire   Zint;

    DFF  U1 (.Q(QR),.D (D), .CK(CLK)); //Rising Edge Triggered
    DFFN U2 (.Q(QF),.D (D),.CKN(CLK)); //Falling Edge Triggered

    INVERT U3 (.Z(Zint),.A(D));

    assign #(0.035:0.08:0.120) Z=Zint;

endmodule // SLE_DELAY_STEP

```

Modeling incomplete gate solutions

There may be times when the analysis and specific gate selection for a given technology are not complete, but the start of simulation efforts is desired. When this occurs, a behavioral or RTL representation of the cell/logic can be substituted into the map file until the final selections have been completed. When the selections are done, the map file is easily updated with the real gates, and the change is transparent to the simulation environment.

4.3 State Initialization

Even when the final cell selections have been made, there could still be the desire to add features or characteristics to the gate level representations that assist simulation efforts. For instance, there are times when it is desirable to provide a quick and easy way to coerce initial states in gate-level circuits. This can be accomplished with a simple initialization statement in the library map file.

Consider the case of a clock divider circuit that does not receive a reset input. In regular gate-level simulations the initial state of that clock divider register may be determined via some kind of state initialization mechanism within the library itself. However, it is convenient to provide the option of setting the initial value within the map file:

```

module SLE_CLKDIV_FLOP (Q,CLK);

    output Q;
    input  CLK;
    reg    Q;

    always @(posedge CLK) Q = ~Q;

    `ifdef EN_START_VALS
        initial Q = 1'b0;
    `endif

endmodule // SLE_CLKDIV_FLOP

```

In this case, a special define enables initialization of the clock divider register. An initial block can only be used for assignments to “reg” data types. Once real gates have been selected, there is often no “reg” data type. Therefore, an initial block cannot be used. However, a similar mechanism can be utilized to optionally remove the initial “X” from the register.

```

module SLE_CLKDIV_FLOP (Q,CLK);

    output Q;
    input  CLK;
    wire   Q_n,Q_n_noX;

    `ifdef EN_START_VALS
        assign Q_n_noX = (Q_n === 1'bx) ? 1'b0 : Q_n;
    `else
        assign Q_n_noX = Q_n;
    `endif

    INVERT U3 (.Y(Q_n), .A(Q));
    DFF_SCAN U1 (.Q(Q), .D(Q_n_noX), .CK(CLK), .SI(Q), .SE(1'b0));

endmodule //SLE_CLKDIV_FLOP

```

Using these mechanisms creates exposure to the risk that a simulation-induced force could be occurring inadvertently and masking a true reset problem. However, the risk can be significantly reduced by requiring special setup configurations to enable start value initialization features.

5 Recommendations

Though using this methodology assisted our efforts a great deal in the development and maintenance of our SIP, we recognize that there are always lessons to be learned and ideas for improvement for next time. The following recommendations highlight areas that we believe could improve this abstraction methodology in subsequent endeavors.

5.1 Technology Dependent Source Isolation

In general, it is a good idea to separate technology dependent design source from the rest of the source code. This allows code changes specific to a given technology to be implemented without impacting the balance of the design database. Pulling the correct tech specific files can then be handled by the use of defines or project setup scripts.

5.2 Common Setup File

As mentioned, abstracting portions of the design can improve efficiency throughout the entire development process. However, it does require some special handling to ensure that all tools and process steps are correctly pointing to valid source directories and libraries. While this can typically be handled within the context of each individual development stage (design, verification, synthesis, STA, etc...), the overall methodology would be improved by using a common setup file.

This file would become the central location for all generic and technology specific setup information. A single copy of the file would exist and would be referenced by all other tools and scripts. The most obvious benefit to this is creating a single place where file, path, and library references are maintained. There could be many ways to do this, but here are two ideas that we are evaluating for our next pass:

- ***Choose one scripting language, such as Tcl, for the common script, and add methods to pass data to other scripting languages.*** Although Tcl would not be this author's first choice for scripting (Go Perl!), it is a reasonable option given that most EDA tools have now migrated to Tcl as their main control language. When setup information is needed by a tool or process that cannot directly source Tcl, the Tcl script could provide needed setup data via a return process, standard out, or by utilizing environment variables.
- ***Generate the common setup file as a text file.*** With this method, the common file could be maintained in a format that is easily parsed by control scripts or programs. Again, not all languages execute text parsing as well as others (Go Perl!), but most should be capable of performing the rudimentary processes needed to retrieve the setup information.

5.3 Robust Revision Control

Revision control is another area that can significantly enhance or inhibit the effectiveness of the library abstraction methodology. Whether the revision control environment is commercially packaged software or a home-grown tool, how it is wrapped into the abstraction mechanism will be important. The flexibility for script access of source control will allow for better management of the design development process across multiple technologies.

6 Conclusion

Silicon Intellectual Property is a great mechanism for preserving the reusability and portability of ASIC/FPGA designs. In an ideal world, SIP source would remain generic enough to allow easy implementation into any silicon technology. However, at times, design complexity requires tight control over the specific library elements chosen for final circuit design. When hand-instantiation requirements occur in designs with multiple technology targets, product development and maintenance can become very challenging. Using library abstraction within the source code for such a design can add the flexibility and adaptability needed for a more maintainable SIP solution.