Synthesis Design Flows Using the Quartus University Interface Program (QUIP)

Version 1.1

February 6, 2008

Table of Contents:

1. Overview

This document is intended for developers working with the Quartus University Interface Program (QUIP). It describes design flows for using QUIP for synthesis and in combination with academic design tools. Quartus Integrated Synthesis (QIS) can be used either as a comparison tool for academic synthesis research or simply as a front end to convert VHDL and Verilog designs to formats (e.g. BLIF [2]) that can be used by academic tools (ABC[10], SIS[3], RASP[11], VPR[12]) without the need for an expensive commercial tool.

The original version of this document was written targeting version 6.1 of the Quartus II software. It has been updated for version 7.2 of the Quartus II software, and will most likely be updated for future versions as additional functionality is added.

Any questions on the content in this document should be sent to quip@altera.com for advice. For fastest response, please reference the name of the document so that the question can be routed to the most appropriate person.

Warning: Users of QIS who perform synthesis using these flows are reminded that the Quartus license allows you to use Quartus for academic purposes or for synthesizing logic to Altera devices. Though we encourage the use of Quartus for research, it is a violation of your license agreement to use Quartus to synthesize designs for commercial purposes (e.g. as the front end to a competitive product to Altera's FPGAs or as the front-end to a commercial EDA tool under development).

2. Quartus Integrated Synthesis (QIS)

The Quartus II software environment allows design entry as schematic, instantiated LPM modules or IP cores, or behavioral logic written in VHDL, Verilog, SystemVerilog or AHDL (Altera HDL or TDF). For documentation on QIS see the extensive online documentation [1].

QIS will parse and elaborate the entire synthesizable subset of these languages, including advanced constructs such as FOR..GENERATE, GENERIC, etc, and will infer finite-state-machines into logic, RAM into embedded memory blocks, and multipliers into dedicated DSP blocks whenever the underlying architecture supports such blocks.

Synthesis with QIS consists of

- VHDL/Verilog/SystemVerilog Analysis (parsing)
- Elaboration (binding and instantiation)
- RTL Inferencing and Synthesis (e.g. arithmetic, RAM, DSP blocks, shift-registers and MUXes are extracted from the netlist).
- LPM Instantiation (common functions such as barrel shifters are converted into optimized library functions that have been tuned for each architecture)
- Technology-Independent Multi-Level Optimization (MLS) (SIS-like algebraic and functional techniques to optimize the netlist).
- Technology Mapping (standard mapping to k-LUT functions). Tech Mapping can be for speed, area or "balanced" and has different variants for special architectures, most notably the Stratix II ALM
- Atom Building (create "placable objects" containing Altera logic cells and DFF registers).

When the input netlist is already mapped to “atoms”, e.g. using Synplicity’s or Mentor Graphics’ synthesis tools, there are still portions of the netlist that are synthesized by QIS using LPMs. There are also options that allow the netlist to be re-synthesized starting with the un-mapped gate-level netlist.

2.1 Netlist Definitions

An ***RTL design*** is described behaviorally in VHDL, Verilog, or SystemVerilog. The design file is characterized by expressions of combinational and sequential logic inside clock-controlled blocks. The following is a RTL design expressed in Verilog

```
module xbar(in,out,s,clk);
    input [15:0] in;
    input [63:0] s;
    input clk;
    output [15:0] out;
    reg [15:0] out;
    integer k;

    reg [15:0] inreg;

    always@ (posedge clk)
    begin
        inreg <= in;
        for (k = 0; k < 16; k = k+1)
        begin
            out[k] <= inreg[{s[4*k+3],s[4*k+2],s[4*k+1],s[4*k]}];
        end
    end
endmodule
```

A ***gate-level*** or ***structural*** netlist is a logic function in which the functionality of the design is already mapped into structural gates. Here is an example in Verilog.

```
module and_or(clk,a,b,c,d,out);
      input a,b,c,d,clk;
      output out;
      wire a,b,c,d,f;
      wire x,y;
      reg out;
      or(f,x,y);
      and(x,a,b);
      and(y,c,d);
      dff my_dff(.d(f),.clk(clk),.q(out));
endmodule
```

Here is an example of a structural netlist in ***BLIF***.

```
.model andgate
.inputs a b
.outputs f
.names a b f
11 1
```

```
.end
```

A ***mapped*** netlist is expressed in k-input LUTs and DFFs. In the case of BLIF, this can mean that the netlist is written so that all .names statements have k or fewer inputs. QIS encounters mapped netlists through an internal format consisting of k-input LUTs and DFFs or through netlists written in atom-level VQM format. A netlist in VQM format can be generated by Synplicity's synthesis tools or by Quartus II.

The VQM format and additional design examples are presented in [5].

2.2 Synthesizing a Design with QIS

There are several ways to synthesize a design using QIS.

From the command line, you can synthesize the design "test.v" as follows:

```
quartus_map test
```

This will synthesize the design and also create two files (in addition to the output/report files):

1. test.qpf – the Quartus Project File
2. test.qsf – the Quartus Settings File

If the design contains multiple source files, it is often beneficial to use the Quartus GUI to input the design file list. This is because Quartus needs to see libraries before they are used, Verilog `defines before they are used, etc. QIS will attempt to do this using rules (e.g. assuming module foo() is in the file foo.v), but these heuristics cannot guarantee success, and with large designs some manual effort may be required. If you use the GUI, the QSF file will contain the "add-file" commands in the correct order.

The GUI is generally easier if you are actually creating a design (rather than synthesizing an existing design) because it will allow you to file-locate to syntax errors, locate online help, etc. For benchmarking experiments, the command-line and scripting facilities will be more commonly used.

For more information on QIS, refer to the Quartus II Development Handbook[1].

3. BLIF Output from QIS

Beginning with Quartus II V4.2 there are methods to output BLIF (Berkeley Logic Interchange Format [2]) from Quartus. In Quartus II V6.1 we introduced methods to support hierarchical BLIF by handling all unknown netlist constructs such as DSP and memory blocks as black boxes. This increases not only the number of designs that can be written to BLIF, but also the quality of those designs since all modern designs of significant size contain either RAM or DSP blocks. First, we document the basic BLIF support. This is followed by a description of hierarchical BLIF.

3.1 Berkeley Logic Interchange Format (BLIF)

BLIF is a common netlist format for academic tools, particularly SIS [3] from Berkeley. It is a gate-level netlist format with basic primitives for input, output, gates and DFF with a named clock.

BLIF is rather restrictive as a synthesis netlist format, as it is unable to express back-boxes that would be required to implement primitives such as DSP (multiplier) blocks, RAM that are not defined at the gate-level. The most glaring issue with BLIF is that there is no way to express arithmetic carry chains without converting them to gates.

Nonetheless, BLIF is the standard format used by the MCNC benchmarks [4] and the majority of research in synthesis.

Note that the Verilog and EDIF versions of these netlists are translations from BLIF to Verilog, not the other way around.

An example of a BLIF netlist was shown in the preceding section.

3.2 Obtaining BLIF from QIS

The instructions herein refer to Quartus II Version 5.0 or newer. If you previously had instructions for obtaining BLIF output from Quartus 4.2 using hidden variables you need to use these newer instructions.

BLIF can be output from QIS in three different formats:

a) After RTL inference and LPM instantiation, but before technology independent multi-level logic optimization (MLS).

b) After technology independent optimization, but before technology mapping to LUTs.

c) After complete optimization and technology mapping.

It is important to note the differences between these three formats. In the first case, the assumption is that your research goal is to do gate-level synthesis on the design. In the second case, you have a good starting point for evaluating a new technology mapping algorithm. In the third case, your goal is either to extract a comparison point from QIS, or to use the mapped netlist for some other purpose (e.g. to convert to .net format for VPR).

Writing BLIF before Technology Independent Optimization

To dump BLIF before technology independent (MLS) optimization, you need to turn on the following control variables:

```
no_add_ops = on
opt_dont_use_mac = on
dump_blif_before_optimize = on
```

These variables indicate that adder-synthesis will not be performed (because BLIF cannot support adder carry-chains), DSP/multiply-accumulate blocks will not be used, and the BLIF output will take place before gate-level optimizations.

There are several methods to set these variables. The preferred method is to add the appropriate TCL-like syntax to your quartus settings file (QSF), which is named <projectname>.qsf once you have created the project. This example sets all three variables.

```
set_global_assignment -name INI_VARS “no_add_ops=on;opt_dont_use_mac=on;
dump_blif_before_optimize=on"
```

Writing BLIF after Technology Independent Optimization

To write BLIF after technology independent (MLS) optimization, you need to turn on the following control variables:

```
no_add_ops = on
opt_dont_use_mac = on
dump_blif_after_optimize = on
```

Use the following in your QSF:

```
set_global_assignment -name INI_VARS “no_add_ops=on;opt_dont_use_mac=on;
dump_blif_after_optimize=on"
```

Writing BLIF after LUT Mapping

To write BLIF after LUT mapping, we use a different 3rd setting:

```
no_add_ops = on
opt_dont_use_mac = on
dump_blif_after_lut_map = on
```

Use the following in your QSF.

```
set_global_assignment -name INI_VARS
"no_add_ops=on;opt_dont_use_mac=on;dump_blif_after_lut_map =on"
```

In all three the cases, a file called <projectname>.blif will be created in your project’s directory.

Example 1: Design and_or

This simple design consists of an OR gate fed by 2 2-input AND gates. Because of the simplicity of the design, the BLIF output before and after technology independent optimization is identical.

Verilog source for and_or.v

```
module and_or(a,b,c,d,f);
      input a,b,c,d;
      output f;
      wire a,b,c,d,f;
      wire x,y;
      or(f,x,y);
      and(x,a,b);
      and(y,c,d);
endmodule
```

BLIF output before technology independent optimization

```
.model and_or
.inputs a b c d
.outputs f

#g1 = x
.names a b g1
11 1

#g2 = y
.names c d g2
11 1

#g1 = x
#g2 = y
#f = comb~0
.names g1 g2 f
00 0

.end
```

BLIF output after technology independent optimization

```
.model and_or
.inputs a b c d
.outputs f

#g1 = x
.names a b g1
11 1

#g2 = y
.names c d g2
11 1

#g1 = x
#g2 = y
#f = comb~0
.names g1 g2 f
00 0

.end
```

BLIF output after technology mapping

```
.model and_or
.inputs a c d b
.outputs f

#f = comb~0
.names a c d b f
0110 1
1110 1
1001 1
1101 1
1011 1
0111 1
1111 1

.end
```

Example 2: Design adder

The second example features a simple 2-bit adder.

Verilog source for adder.v

```
module adder(a,b,f);
input [1:0] a;
input [1:0] b;
output [2:0] f;
assign f = a + b;
endmodule
```

BLIF output before technology independent optimization

```
.model adder
.inputs a[0] b[0] a[1] b[1]
.outputs f[0] f[1] f[2]
```

```
#f[0] = Add0~0
.names a[0] b[0] f[0]
10 1
01 1

#g2 = Add0~4
.names a[0] b[0] g2
11 1

#g2 = Add0~4
#f[1] = Add0~5
.names g2 a[1] b[1] f[1]
100 1
010 1
001 1
111 1

#g2 = Add0~4
#g4 = Add0~7
.names g2 a[1] g4
11 1

#g2 = Add0~4
#g5 = Add0~8
.names g2 b[1] g5
11 1

#g6 = Add0~9
.names a[1] b[1] g6
11 1

#g4 = Add0~7
#g5 = Add0~8
#g6 = Add0~9
#f[2] = Add0~6
.names g4 g5 g6 f[2]
000 0

.end
```

BLIF output after technology optimization

```
.model adder
.inputs a[0] b[0] b[1] a[1]
.outputs f[0] f[1] f[2]

#f[0] = Add0~0
.names a[0] b[0] f[0]
10 1
01 1

#g2 = Add0~4
.names a[0] b[0] g2
11 1

#g3 = Add0~9
.names b[1] a[1] g3
11 1
```

```
#g4 = Add0~19
.names b[1] a[1] g4
00 0

#g2 = Add0~4
#g4 = Add0~19
#g5 = Add0~25
.names g2 g4 g5
11 1

#g3 = Add0~9
#g5 = Add0~25
#f[2] = Add0~24
.names g3 g5 f[2]
00 0

#g2 = Add0~4
#g8 = Add0~131
#f[1] = Add0~127
.names g2 g8 f[1]
10 1
01 1

#g8 = Add0~131
.names b[1] a[1] g8
10 1
01 1

.end
```

BLIF output after technology mapping

```
.model adder
.inputs a[0] b[0] b[1] a[1]
.outputs f[0] f[1] f[2]

#f[0] = Add0~0
.names a[0] b[0] f[0]
10 1
01 1

#f[1] = Add0~127
.names a[0] b[0] b[1] a[1] f[1]
1100 1
0010 1
1010 1
0110 1
0001 1
1001 1
0101 1
1111 1

#f[2] = Add0~24
.names b[1] a[1] a[0] b[0] f[2]
1100 1
1110 1
1101 1
```

```
1011 1
0111 1
1111 1

.end
```

3.3 Hierarchical BLIF: Black Box Primitive Support

If the BLIF writer encounters any FPGA hard block such as a RAM or MAC block or any other unsupported gate, it gives an internal error. The supported gate types are combinational gates (and,or,xor,not), DFF, input pin, output pin, bidir, buffers (carry-sum,expander,cut and other wire buffers), LUT, tri-bus and IO buffer (tri-state, open-drain).

The setting "no_add_ops" described above will convert the adders in the netlist to equivalent combinational logic and avoid the issue for adder arithmetic. The setting “opt_dont_use_mac” prevents DSP blocks from being inferred, but doesn’t eliminate existing macro blocks.

Starting with Quartus II Version 6.1, we introduced black box primitives to BLIF. Any block in the netlist that is not supported by traditional BLIF will be turned into a black box leading to a hierarchical BLIF netlist. Black boxes are instantiated in the current model as sub-circuits using the .subckt construct. Every black box needs to be defined outside of the current model. It requires a name, an interface consisting of inputs and outputs and the keyword “.blackbox”.

The support of black box primitives is activated by adding the following variable to the QSF file:

```
set_global_assignment -name INI_VARS "dump_blif_with_blackboxes=on”
```

Note, that you can set the variable

```
      opt_dont_use_mac = off
```

if you want a multiplier to be transformed into a DSP block.

Example 3: 6x6bits multiplier in Verilog and the corresponding hierarchical BLIF netlist:

```
module mult(a,b,f);
input [5:0] a;
input [5:0] b;
output [11:0] f;
assign f = a * b;
endmodule


.model mult
.inputs a[0] a[1] a[2] a[3] a[4] a[5] b[0] b[1] b[2] b[3] b[4] b[5]
.outputs f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9] f[10] f[11]

.subckt blackbox_g1 g14=g14 g15=g15 g16=g16 g17=g17 g18=g18 g19=g19
g20=g20 g21=g21 g22=g22 g23=g23 g24=g24 g25=g25 f[0]=f[0] f[1]=f[1]
f[2]=f[2] f[3]=f[3] f[4]=f[4] f[5]=f[5] f[6]=f[6] f[7]=f[7] f[8]=f[8]
f[9]=f[9] f[10]=f[10] f[11]=f[11]
.subckt blackbox_g14 a[0]=a[0] a[1]=a[1] a[2]=a[2] a[3]=a[3] a[4]=a[4]
a[5]=a[5] b[0]=b[0] b[1]=b[1] b[2]=b[2] b[3]=b[3] b[4]=b[4] b[5]=b[5]
g14=g14 g15=g15 g16=g16 g17=g17 g18=g18 g19=g19 g20=g20 g21=g21 g22=g22
g23=g23 g24=g24 g25=g25
```

```
.end

# blackbox_g1 = lpm_mult:Mult0|mult_fl01:auto_generated|result[0]
#g14 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2
#g15 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT1
#g16 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT2
#g17 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT3
#g18 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT4
#g19 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT5
#g20 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT6
#g21 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT7
#g22 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT8
#g23 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT9
#g24 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT10
#g25 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT11
#f[0] = lpm_mult:Mult0|mult_fl01:auto_generated|result[0]
#f[1] = lpm_mult:Mult0|mult_fl01:auto_generated|result[1]
#f[2] = lpm_mult:Mult0|mult_fl01:auto_generated|result[2]
#f[3] = lpm_mult:Mult0|mult_fl01:auto_generated|result[3]
#f[4] = lpm_mult:Mult0|mult_fl01:auto_generated|result[4]
#f[5] = lpm_mult:Mult0|mult_fl01:auto_generated|result[5]
#f[6] = lpm_mult:Mult0|mult_fl01:auto_generated|result[6]
#f[7] = lpm_mult:Mult0|mult_fl01:auto_generated|result[7]
#f[8] = lpm_mult:Mult0|mult_fl01:auto_generated|result[8]
#f[9] = lpm_mult:Mult0|mult_fl01:auto_generated|result[9]
#f[10] = lpm_mult:Mult0|mult_fl01:auto_generated|result[10]
#f[11] = lpm_mult:Mult0|mult_fl01:auto_generated|result[11]
.model blackbox_g1
.inputs g14 g15 g16 g17 g18 g19 g20 g21 g22 g23 g24 g25
.outputs f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9] f[10] f[11]
.blackbox
.end

# blackbox_g14 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2
#g14 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2
#g15 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT1
#g16 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT2
#g17 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT3
#g18 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT4
#g19 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT5
#g20 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT6
#g21 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT7
#g22 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT8
#g23 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT9
#g24 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT10
#g25 = lpm_mult:Mult0|mult_fl01:auto_generated|mac_mult2~DATAOUT11
.model blackbox_g14
.inputs a[0] a[1] a[2] a[3] a[4] a[5] b[0] b[1] b[2] b[3] b[4] b[5]
.outputs g14 g15 g16 g17 g18 g19 g20 g21 g22 g23 g24 g25
.blackbox
.end
```

All recent industrial designs of significant size contain some kind of hard block, most of the time in form of RAM or DSP blocks. Those designs can now be handled by the BLIF writer increasing the number of supported designs and more importantly their quality significantly.

ABC [10], the new synthesis and verification tool developed by the Berkeley Logic Synthesis and Verification Group at UC Berkeley, supports hierarchical BLIF. More information on ABC is available on Alan Mishchenko's web page http://www.eecs.berkeley.edu/~alanmi/abc . You can also download the latest version of ABC from that web page.

We are not aware of any other tool that supports hierarchical BLIF as of January 2008. In particular, we believe that nobody will extend the capabilities of legacy tools such as SIS to support hierarchical BLIF.

3.4 Restrictions on BLIF output

The output of BLIF is limited in several ways. This is due both to the BLIF language itself, due to restrictions in SIS, and due to limitations in how we chose to output the netlist from QIS.

Asynchronous Signals

The BLIF writer ignores (i.e. just drops) any asynchronous signals (for example, aclear, aload etc.) in the design. These asynchronous signals are an intrinsic part of the logic cell in many Altera architectures, but would result in latches in BLIF that would basically just confuse SIS and most other synthesis tools.

If the source design has one or more asynchronous signal, the resultant BLIF design will not be functionally equivalent to the source design. A user warning is issued if this is the case.

To obtain the most accurate benchmarking methodology, you should re-write the HDL so that asynchronous signals are not used and this condition does not occur.

WYSIWYG Hard-Blocks

WYSIWIG (what-you-see-is-what-you-get) blocks are created using LPMs and other methods in synthesis. For pure HDL designs using the above flow all WYSIWIGs should be converted to plain gates, so any error to the contrary should be reported to quip@altera.com. However, if designs are generated using Altera system-level tools such as DSP-Builder they will be created using WYSIWIGs which will fail to be processed by the BLIF writer except if the user chooses to turn black box support on.

"Pre-Optimized" Netlists

It is difficult to guarantee zero optimization since some aspects of optimization such as removing un-connected gates from the netlist are seen even by the analysis portion of the flow. Nonetheless the "un-optimized" netlist means that no non-trivial algorithm was applied to the netlist, whereas the "post LUT-mapping" netlist indicates that all standard optimizations were performed.

VCC/GND Nets

The BLIF writer replaces the references to constant 1 with references to vcc, and the references to const 0 with references to gnd. It appends the definitions of vcc and gnd at the end of the output BLIF whenever they are used, unless the source file already defines them. However, the BLIF writer does not attempt to figure out whether vcc and gnd are already defined as constants in the source file. So if the source uses vcc and/or gnd as variable names or in any other non-conventional way, the output BLIF may not be functionally equivalent.

4. Returning SIS output to Quartus

To facilitate benchmarking, particularly correctness testing, we wrote a simple add-on to Berkeley SIS to output the BLIF netlist in gate-level Verilog. By returning the synthesized netlist to Quartus a researcher can use the Quartus simulator or other 3rd party tools such as ModelSim to verify the correctness of their synthesis transformations.

The instructions are as follows:

1. Obtain and compile a source-code version of Berkeley SIS. (We have tested with Version 1.3.6 – an unofficial distribution is available at http://www-cad.eecs.berkeley.edu/~pchong/sis.html).

2. Copy the file write_verilog.c provided in this Altera package **Error! Reference source not found.** to $SIS_HOME/sis/io directory, where SIS_HOME is the home directory of SIS where the SIS distribution is installed.

```
cp write_verilog.c $SIS_HOME/sis/io
```

3. Change the makefile under $SIS_HOME/sis/io as follows:

- Include `write_verilog.c` under `libio_a_SOURCES`
- Include `write_verilog.$(OBJEXT)` under `am_libio_a_OBJECTS`

Note that you should be working with the file "makefile" that is created after you compiled SIS for the first time. There are other makefiles such as "makefile.in" and "makefile.am" that are part of the SIS distribution for other purposes.

4. Make the following change in `$SIS_HOME/sis/io/com_io.c` to introduce the command `write_verilog`:

- In the function: `com_write`, add the following else if segment just below the if segment for `write_blif`:

```
else if(strcmp(argv[0], "write_verilog") == 0)
{
   write_verilog(fp, *network, short_name, net_list);
   status = 1;
}
```

- In the function: `init_io`, add the following line just below the similar line for `write_blif`:

```
com_add_command("write_verilog", com_write,0);
```

5. Add the following line in the `$SIS_HOME/sis/io/io.h` file just below the similar line for `write_blif`:

```
EXTERN void write_verilog ARGS((FILE *, network_t *, int, int));
```

6. Compile SIS, and start it.

After reading a file into SIS, type the command: `write_verilog <output_filename>` (or add it to an appropriate SIS script).

If SIS outputs a verilog version of the read file, your changes were correctly incorporated. If SIS does not understand the command or does not compile, re-check the changes you made. If SIS crashes while writing verilog, submit an error report to Altera at quip@altera.com.

5. Mapped VQM netlists

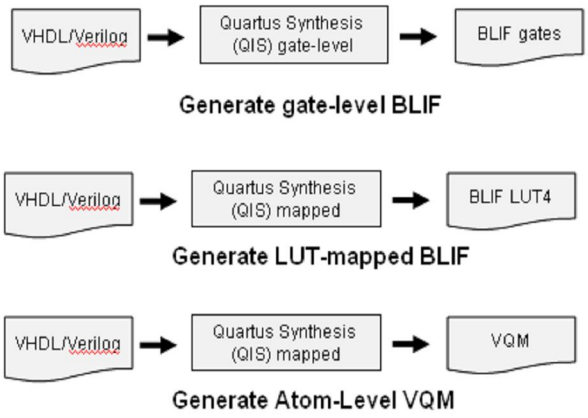
When working with “complete” Altera synthesized Logic Elements, a VQM output will show the adder logic. More information on the VQM format can be found in [5]. A VQM netlist is the physical atom-level netlist used by Quartus II placement and routing tools.

VQM netlists are generally preferred when the goal of the research is to directly emulate a member of the Stratix family of devices or other logic cell, e.g. to evaluate a placement algorithm on Stratix LEs which is then fed back into Quartus for routing and timing analysis.

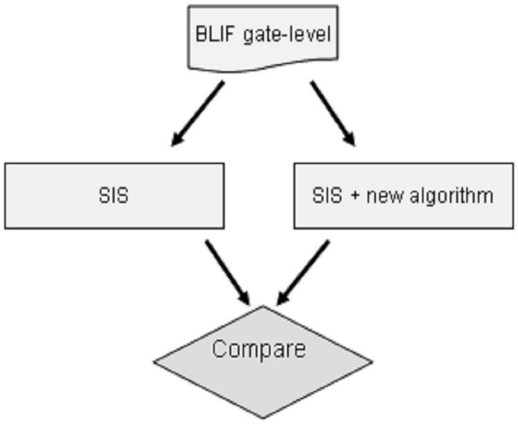
6. Design Flows and Benchmarking

This section proposes some basic design flows for synthesis using QUIP. These are just the most obvious uses. There are many other combinations that would be useful for research.

One of the primary benefits of QIS is simply converting designs in VHDL, Verilog or SystemsVerilog format to BLIF or VQM for use in other tools. These flows are described earlier in the document.

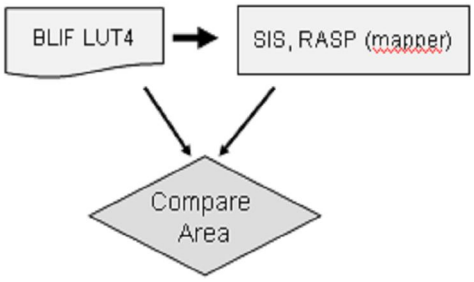


Once a BLIF netlist is obtained, however, that netlist can be used to evaluate new synthesis algorithms (e.g. SIS vs. SIS + new_algoroithm), different scripts, (e.g. SIS script 1 vs. SIS script 2), etc.



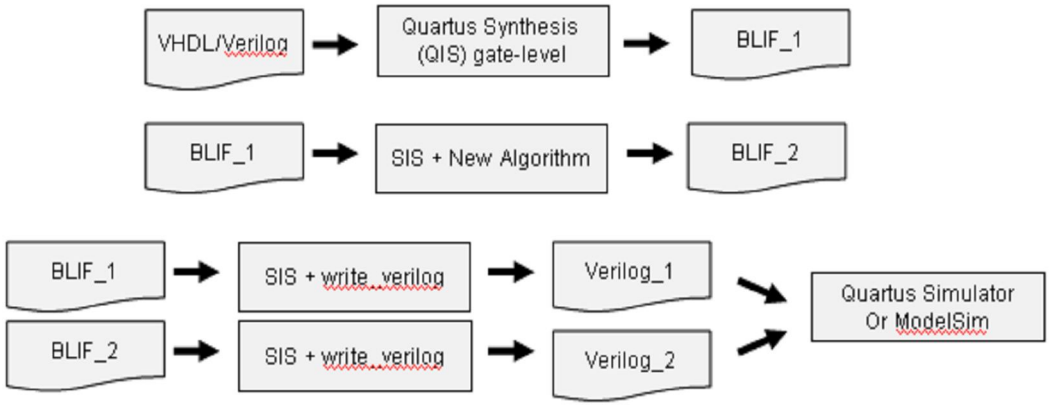
Compare New Synthesis Algorithm

Using SIS or SIS and RASP will further allow comparison of new technology mapping algorithms.



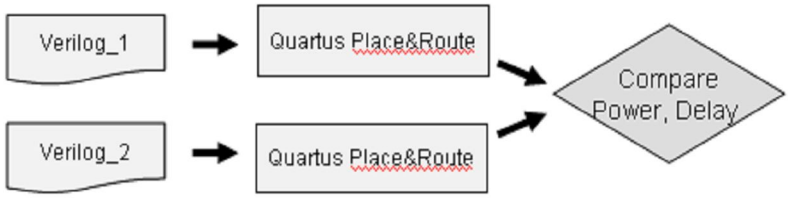
Compare Tech-Mapper

Using SIS and the write_verilog add-in, the netlist can be converted back to Verilog. This allows researchers to test the correctness of their synthesis transformation using either the Quartus simulator or Modelsim (an OEM version of Modelsim ships with Quartus).



Synthesis Algorithm Correctness Testing

The write_verilog interface also allows transformed netlists to re-enter Quartus for evaluation of post-place&route delay and power analysis.



Evaluate Synthesis Algorithm for Power, Delay

For research into FPGA architectures you can evaluate routability, delay, power, with VPR (and various add-ons to VPR) on bigger designs by converting LUT-level BLIF netlists for input to VPR.

FPGA Architecture Experiments

Clearly there are numerous other flows, but these are the ones we would expect to see most commonly.

7. Test Designs in VHDL/Verilog

To facilitate the use of QUIP and these flows, we have provided a number of benchmark designs with QUIP. These are described in [7]. More benchmark designs can be found in [8]. Note that

due to the restrictions on the BLIF format, designs with memory or other issues mentioned earlier will not compile into BLIF. The designs which are appropriate for this flow are indicated in [7]. All other designs are only supported by the hierarchical BLIF flow described in section 3.3.

8. Roadmap

Ideally these are some of the items we would like to accomplish for future versions of QUIP. However, some may become obsolete due to the creation of other tools, etc. in universities.

VHDL/Verilog gate-level output

- Identify some reasonable gate-level VHDL/Verilog format, and potentially a parser so that we can directly output easily readable VHDL and Verilog to remove the dependence on BLIF and SIS.

VHDL/Verilog RTL-level output

- Identify some reasonable way to output RTL-level constructs (prior to gate-level flattening) to allow research in RTL synthesis. Possibly with some kind of interface to Icarus Verilog

Future Enhancements for BLIF

- Converge on the "best" level of gate-level output (currently we fill the entire truth table, and output some wire-LUTs (buffers) that probably aren't necessary).
- Adding carry-chain support to BLIF and to the BLIF writer would further improve the quality of experimental results.

9. References

[1] "Quartus II Development Software "Online Handbook" at http://www.altera.com/literature/lit-qts.jsp. Volume 1 "Design and Synthesis".

[2] Berkeley Logic Interchange Format (BLIF). Available at multiple online locations such as, http://www.bdd-portal.org/docu/blif/, http://www1.cs.columbia.edu/~cs4861/sis/blif/ .

[3] SIS: A System for Sequential Circuit Synthesis, 1992 by Sentovich et al http://www.csc.uvic.ca/~csc485c/SIS/SIS_paper.pdf

[4] S. Yang, "Logic Synthesis and Optimization Benchmarks", Version 3.0", Tech Report MCNC P.O. Box 12889, Research Triangle Park, NC 27709.

[5] "VQM Extractor and Language Functional Description", Vqmx_doc.pdf included with QUIP.

[6] "Benchmarking Using the Quartus University Interface Program (QUIP)". Document quip_benchmarking,pdf in the QUIP package.

[7] "Benchmark Designs for the Quartus University Interface Program (QUIP", Document quip_benchmarks.pdf in the QUIP package.

[8] http://www.eecs.berkeley.edu/~alanmi/benchmarks/altera

[9] File /utils/blif/write_verilog.c in this QUIP package.

[10] ABC: A System for Sequential Synthesis and Verification, 2006 by the Berkeley Logic Synthesis & Verification Group, UC Berkeley, http://www.eecs.berkeley.edu/~alanmi/abc

[11] RASP: A General Logic Synthesis System for SRAM-based FPGAs, 1996 by J. Cong et al., UCLA, http://ballade.cs.ucla.edu/software_release/rasp

[12] VPR: Versatile Packing, Placement, and Routing for FPGAs, 1997, V. Betz et al., Univ. of Toronto, http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html

10. Revision History

Version 1.1: General updates and enhancements:

- Introducing hierarchical BLIF support.
- Adding SystemVerilog as a supported front-end language.
- Ability to generate a BLIF netlist after technology independent optimization.

Version 1.0: Initial version of this document.