



Synopsys/Xilinx High Density Design Methodology Using FPGA Compiler™

XAPP107 August 6, 1998 (Version 1.0)

Written by: Bernardo Elayda & Ramine Roane

Summary

This paper describes design practices to synthesize high density designs (i.e. over 100k gates), composed of large functional blocks, for today's larger Xilinx FPGA devices using the Synopsys FPGA Compiler. The Synopsys FPGA Compiler version 1998.02, Alliance Series 1.5, and the XC4000X family were used in preparing the material for this application note.

Introduction

For smaller designs, optimal quality of results can often be achieved by ungrouping hierarchical boundaries (`compile -ungroup_all`). For high density designs, designers traditionally partition their circuits into small 5-10k gate modules. With today's version of FPGA Compiler (1998.02) and currently available workstation hardware it is no longer necessary to partition a design into small 5K to 10K blocks for efficient synthesis. In fact, artificial partitioning the design can worsen optimization results by breaking paths into separate partitions and preventing FPGA Compiler from being able to optimize the entire path. Judicious partitioning into larger blocks simplifies overall design budgeting and chip-level synthesis methodology.

Compile Strategies

ASIC designers traditionally use the `compile-characterize-write_script-recompile` method which compiles lower level modules, and works up the hierarchy. This methodology, though effective for ASIC designs which have many complex timing requirements, is not as effective for FPGAs. FPGA timing constraints are applied mainly for RTL synthesis, post-synthesis static timing analysis and for driving the place and route tool via TIMESPEC constraints written by FPGA Compiler in the final netlist. This is because FPGA timing constraints, while consulted during synthesis, are used mainly by the place and route tool, which receives the constraints via TIMESPEC parameters written into the final netlist by FPGA compiler.

Selecting a Strategy

In this section several compile strategies and practical examples are presented.

Design goals and timing requirements require selecting a different compile strategy for each hierarchical block.

Selective flattening of hierarchical blocks is also discussed. There are trade-offs to be made for preserving or removing hierarchical boundaries. In most cases removing all hierarchical boundaries will lead to better post-synthesis quality of results but require longer compile times. With larger

designs preserving some levels of hierarchy will lead to better placement routing results and shorter compile times.

The designer must determine which trade-offs to make to meet his design goals and timing requirements.

High density designs require greater selectivity on which hierarchical boundaries to eliminate for synthesis. Flattening a 150k gate design can lead to long compile times, and compromise results. The following guidelines will help in choosing which blocks to flatten in a large design:

1. Eliminating boundaries between combinatorial blocks allows the synthesis tool to optimize glue logic across the hierarchy. Figure 1 illustrates this configuration. The hierarchical blocks A, B and C are eliminated so that the three combinatorial blocks can be merged and optimized at once.
2. There is no need to flatten hierarchical boundaries bounded by registers since combinatorial blocks are already separated by register boundaries. Figure 2 shows two examples matching this description. The boundaries of A and C can be preserved, especially if they are large blocks.
3. Flatten deeply nested small hierarchical blocks and preserve higher level module boundaries. In Figure 3 block A in `design_40k` contains deeply nested small hierarchical blocks. It is best to flatten block A into `design_40k` (`current_design A; ungroup -all -flatten`). Block B should be flattened some paths between A and B are not bounded by registers. The four top level blocks should be preserved if bounded by registers.

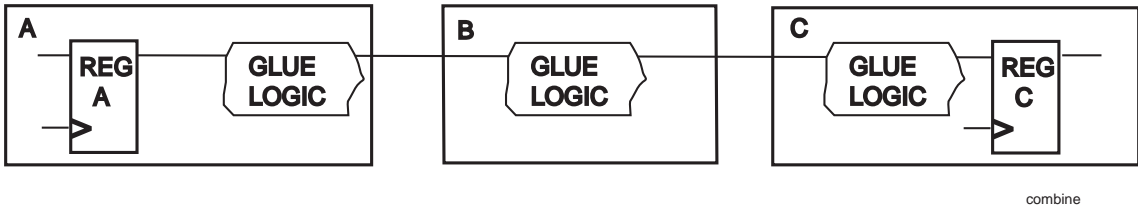


Figure 1: Hierarchical Boundaries Between Combinatorial Logic

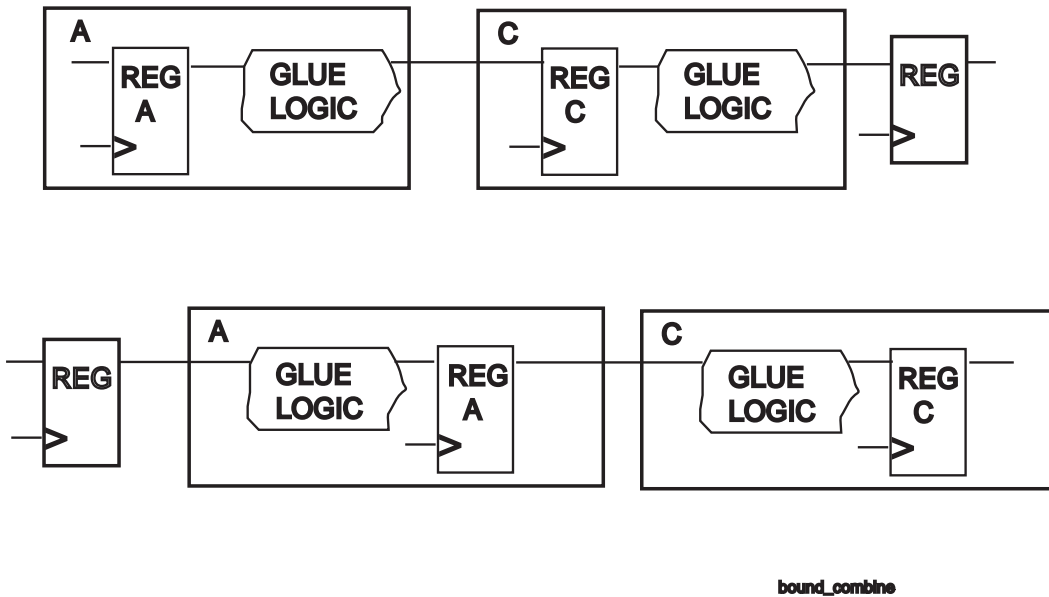


Figure 2: Hierarchical Boundaries Bounded by Registers

Writing DC Scripts

The full-chip design in Figure 3 will be used to illustrate different hierarchical compile methodologies. Different optimization strategies are used for each methodology to demonstrate simple and efficient compile scripts which have the ability to handle large designs.

Compile strategies described in this section are:

- Top-down compile
- Iterative compile: `compile -incremental`
- Bottom-up compile: `compile-characterize-write_script-recompile`

These are general guidelines to better manage `dc_shell` scripts:

- Create a "setup.scr" script containing user variables.
- Create a "constraint" file for each block. (These files typically have a ".con" file extension.)

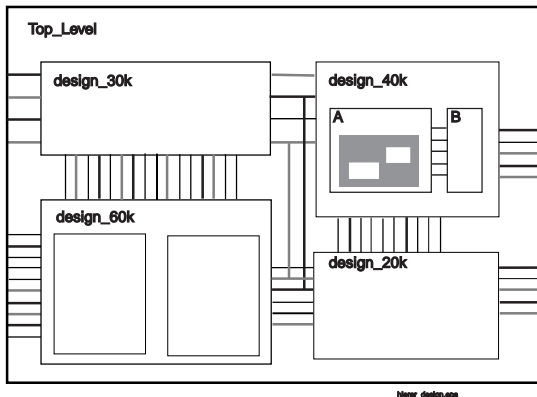


Figure 3: Hierarchical Design

- Name constraint files generated by `characterize` with a “.wscr” extension. This easily identifies that the files are generated by `characterize-write_script`.
- Keep action commands in the main script, i.e.: `compile`, `analyze`, `elaborate`, `link`, `include`, `write`, etc.
- For more efficient pad mapping (e.g. automatic use of IOB registers), always run the `insert_pads` command on top level primary ports before the `compile` command.

Top-down Compile

This is the simplest, fastest and often most efficient way of compiling a design. It can be combined with selective flat-

tening of hierarchical blocks. It is the preferred method for Xilinx FPGAs. Figure 4 contains a script for compiling the design illustrated in Figure 3, where the four top level hierarchical blocks are preserved and the blocks within `design_40k` are flattened.

Iterative Compile

Use this method when multiple designers work on the same circuit, successfully compile different blocks of the design, and need to link them together in a top-level design. Figure 5 provides the script used by the designer working on `design_40k`. Identical scripts are used on all other blocks.

```

Include setup.scr /* environment setup: link libraries, path */
SRC_FILES = "file1 file2 [...]"
analyze -f <format> SRC_FILES

/* Flatten design_40k */
elaborate design_40k
uniquify /* if multiple instances of same component exist */
ungroup -all -flatten

/* Compile entire design top-down */
elaborate TOP_LEVEL
include TOP_LEVEL.con /* timing constraints */
uniquify
check_design /* summary of design issues and warnings */
set_port_is_pad *
insert_pads
compile -map_effort medium

report_fpga
report_timing
write -f db -o TOP_LEVEL_topdown.db
quit

```

Figure 4: Top-down Compile Script With Selective Flattening Iterative Compile

```

Include setup.scr /* environment setup: link libraries, path ... */
SRC_FILES = "file1 file2 [...]"
analyze -f <format> SRC_FILES

/* Flatten and compile design_40k */
include design_40k.con
uniquify
check_design
set_port_is_pad out* /* primary ports in this example */
insert_pads
compile -map_effort medium -ungroup_all

report_fpga
report_timing
write -f db -o design_40k_flat.db
quit

```

Figure 5: Iterative Compile: Script Used For Design_40k

```

Include setup.scr /* environment setup: link libraries, path... */
SRC_FILES = "top level file"
analyze -f <format> SRC_FILES

read -f db design_60k.db
read -f db design_40k_flat.db
read -f db design_30k.db
read -f db design_20k.db
elaborate TOP_LEVEL
include design_40k.con
uniquify
compile -map_effort high -incremental

report_fpga
report_timing
write -f db -o TOP_LEVEL_incremental.db
quit

```

Figure 6: Iterative Compile: Incremental Update of TOP_LEVEL

Figure 6 shows the script used to link four blocks together in TOP_LEVEL. Note: that `compile -incremental` is used to map all unmapped logic that TOP_LEVEL itself may contain, and eventually, to fix new timing violations introduced while linking the four blocks.

Bottom-up: compile-characterize-write_script-recompile

Bottom-up compile with timing characterization and time budgeting is mainly used for ASIC designs. This method is not recommended when using Xilinx FPGAs. Using this methodology can result in long compile times, with little or no improvement in timing. Note: the scripts for this methodology are provided for designers who use FPGAs for prototyping, but whose final target is an ASIC design.

```
Include setup.scr /* environment setup: link libraries, path... */
SRC_FILES = "file1 file2 [...]"
analyze -f <format> SRC_FILES

/* First pass compile */
elaborate design_40k

current_design A
include A.con
compile -map_effort medium

current_design B
include B.con
compile -map_effort medium

current_design design_40k
include design_40k.con
compile -map_effort medium -incremental

report_fpga
report_timing
write -f db -hierarchy -o design_40k_pass1.db

/* Characterization */
characterize {A B}
current_design A
write_script > A.wscr
current_design B
write_script > B.wscr

remove_design -designs

/* Second pass compile */
elaborate design_40k

current_design A
include A.wscr
compile -map_effort medium

current_design B
include B.wscr
compile -map_effort medium

current_design design_40k
include design_40k.con
compile -map_effort high -incremental

report_fpga
report_timing
write -f db -hierarchy -o design_40k_pass2.db

quit
```

Figure 7: Bottom-up Compile With Characterization

Timing constraints passed to the place and route tool with Bottom-up: methodology are more realistic and will likely improve the place and route process.

Bottom-up methodology performs a first pass bottom-up compile using the designer's timing constraints. Once the entire circuit is mapped, the FPGA Compiler performs a static timing analysis on the hierarchical design. A `characterize` and `write_script` exports realistic timing constraints. Finally, restarting from the RTL source, a second pass of bottom-up `compile` is performed with the `write_script` data. Figure 7 above shows a script using this methodology to synthesize `design_40k` without flattening the hierarchy.

However we, still recommend a top-down or iterative compile methodology along with using realistic top-level timing constraints; methods can lead to similar results but with faster compile times.

Describing Timing Constraints

Designers set performance objectives for a design to guide FPGA Compiler's optimization engine and Xilinx's place and route tools. It is necessary to set timing constraints to perform static timing analysis. Note: It is extremely important to set realistic constraints to avoid excessive runtimes for both synthesis and place and route.

Design Constraint Commands

Timing constraints include:

- Input delay: `set_input_delay`
- Output delay: `set_output_delay`
- Clock specifications: `create_clock`,
`set_clock_skew`
- Path group specification: `group_path`

When synthesizing Xilinx to FPGAs, avoid using design-rule commands such as: `set_max_transition`, `set_max_capacitance`, `set_max_fanout`... since place and route tools do not require any specific design rules.

Timing Exception Commands

By default, FPGA Compiler calculates single cycle timing for the clocked paths in a design. Designers can override this with timing exceptions. Timing exceptions are used when logic functions require more than one cycle to execute, or when some signals are considered non-critical. FPGA Compiler allows designers to define multi-cycle paths, false paths, and point-to-point delays. These commands are only needed if designers have point-to-point timing exceptions. Timing exception commands must be specified on valid start and end points. A valid start point is an input port, or the clock pin of a register. A valid end point is an output port, or the data pin of a register. Timing exception commands include:

- `set_multi_cycle_path`
- `set_false_path`
- `set_max_delay`
- `set_min_delay`

Timing Versus Area

FPGA Compiler's numerous commands provide flexibility to target different optimization strategies based on design goals. The most useful commands and variables affecting compile are:

- **uniquify**: Creates a unique instance of the same design. Typically used in situations with multiple instantiations of the same design.
- **ungroup**: Ungroups a level of hierarchy.
- **set_dont_touch**: Sets a "dont_touch" attribute on a design or instance.
- **set_structure**: Selects between Boolean and timing-driven structuring.
- **compile_new_boolean_structure**: Variable which turns on the new Boolean structuring algorithm. This is typically used on designs that are area sensitive

Area-sensitive blocks should be separated from timing critical modules. Figure 8 shows an example of a script compiling `design_40k`, where only block A is critical for area. Block B was successfully compiled for timing.

HDL Design Guidelines

When designing with HDL languages, always think of how to implement complex functions with basic operators. For instance:

1. Dividing a bit vector **X** by a multiple of **2**, can be realized by simply shifting bits to the right (**X>>n**). This is an optimal implementation because it does not require any gates, and can be realized by wire connections.
2. Multiplying a bit vector **X** by **1.8** can be approximated by the following expression: $X + 0.5*X + 0.25*X = 1.75*X = X + X>>1 + X>>2$, which only requires two adders to implement.
3. For more accuracy, multiplying bit vector **X** by **1.8** can use this expression: $X*115/64 = 1.797*X = (X*115) >> 6$. This requires multiplication by a constant, which will be reduced by FPGA Compiler by performing constant pushing and gate optimization during compile.
For a better implementation, $X*115$ can also be realized with 3 subtractors: $X*115 = X*128 - X*8 - X*4 - X = X<<7 - X<<3 - X<<2 - X$.
4. More complex functions (FFT, sin, ...) can be realized by sampling data in a table lookup or using Xilinx CoreGEN™ or LogiCORE™.

```

elaborate design_40k
include design_40k.con

current_design design_40k
compile_new_boolean_structure = true
set_structure true -boolean true -boolean_effort high /*minimize area*/
set_dont_touch {B} /*but still compile block B for timing*/
check_design
compile -map_effort medium -ungroup_all

report_fpga
report_timing
write -f db -o -hierarchy design_40k_boolean.db
quit

```

Figure 8: Timing Versus Area

Use STD_LOGIC Data Types in VHDL

Whenever possible use the STD_LOGIC data types for both synthesis and simulation. Using INTEGER results in increased compile times and poor results thus making the simulation process very difficult. (An INTEGER type signal is 32 bits wide. If the designer must use INTEGER, use it with the correct RANGE statement.)

If-Then-Else and Case Statements

Though logically equivalent, if-then-else and case statements are not implemented in the same way. Case statements are more flexible in Verilog than VHDL, requiring usage of Synopsys Verilog directives (e.g. //synopsys_full_case_parallel_case) to drive the final implementation.

It is very important to define the value of all outputs for each condition when using sequential blocks, (*process* in VHDL and *always@* in Verilog). Failing to do so will result in synthesis of unwanted latches (in order to hold the value of the

missing outputs). Check for the detailed list of latches and registers inferred, in the “elaborate” command log.

VHDL

The conditions of a case statement (Figure 9) in VHDL are mutually exclusive constants; therefore, a case statement is always implemented as a wide multiplexer. Figure 9 shows an example of VHDL case statement and its implementation. If all possible cases are enumerated the “when others” statement is not required. When using an enumerated type, it is recommended not to use the “when others” statement.

If-then-else statements infer priority encoded logic, yielding slower nested two-input multiplexer implementations. Figure 10 shows an example of a VHDL if-then-else statement and its implementation. This description can only be synthesized to a priority encoded implementation since the non-constant conditions of the if-then-else statement cannot be assumed to be mutually exclusive. Whenever possible case statements should be used rather than if-then-else statements in VHDL.

```

process (SEL, A, B, C, D) begin
  case SEL is
    when "00" => OUTC <= A;
    when "01" => OUTC <= B;
    when "10" => OUTC <= C;
    when others => OUTC <= D;
  end case;
end process;

```

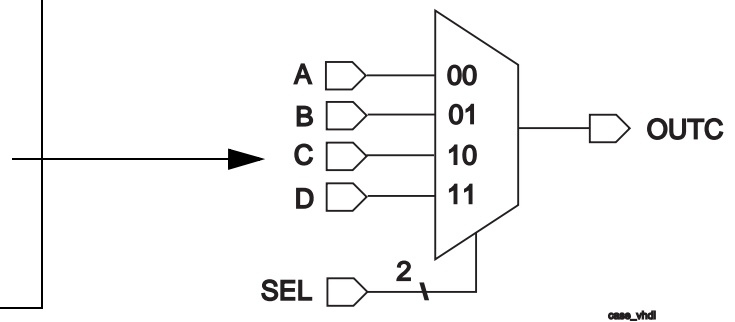


Figure 9: Case Statement Implementation in VHDL

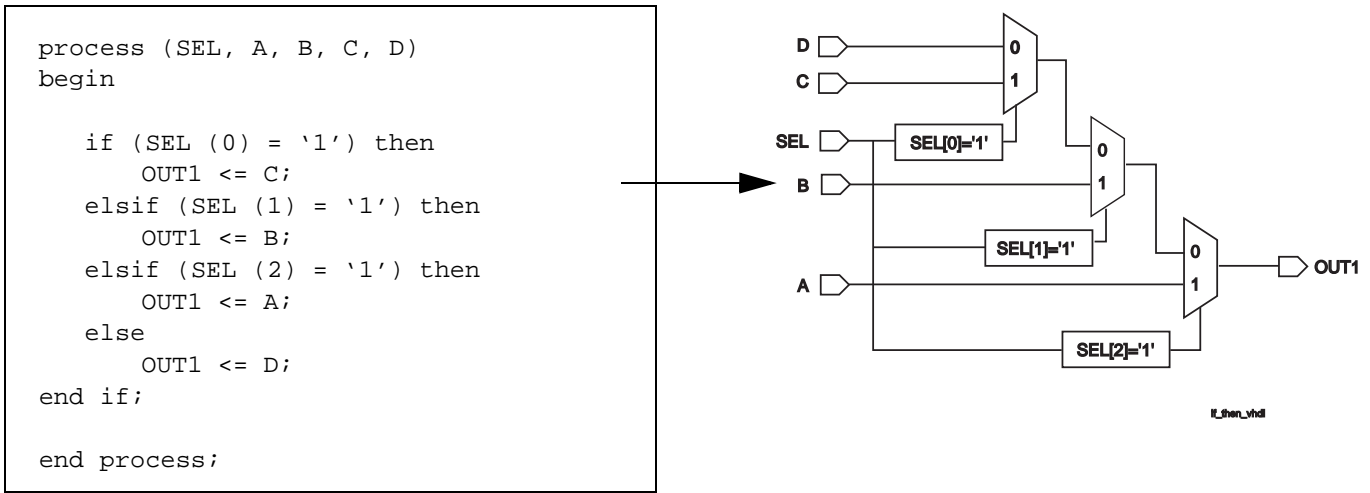


Figure 10: If-then-else Statement Implementation in VHDL

Verilog

Case statement conditions in Verilog can be variables, and not mutually exclusive constants. Variables are not assumed to be mutually exclusive. Implementation corresponding to a case statement including variables will result in priority encoded multiplexers. If a designer knows that, by design, values of these variables are actually mutually exclusive, they can use the //synopsys parallel_case statement in Verilog code to force FPGA Compiler to generate a wide multiplexer. Figure 11 illustrates where conditions of the case statement are variables (C1, C2, C3, C4), and the parallel_case directive is used.

The designer does not have to specify a “default” statement in Verilog (“when others” in VHDL). If the designer has described all possible cases, the use of the full_case statement (Figure 11) will enable FPGA Compiler to use the other states as the “don’t care” set of the multiplexer output. Using this directive normally leads to better optimization results. The equivalent construct in VHDL would be:

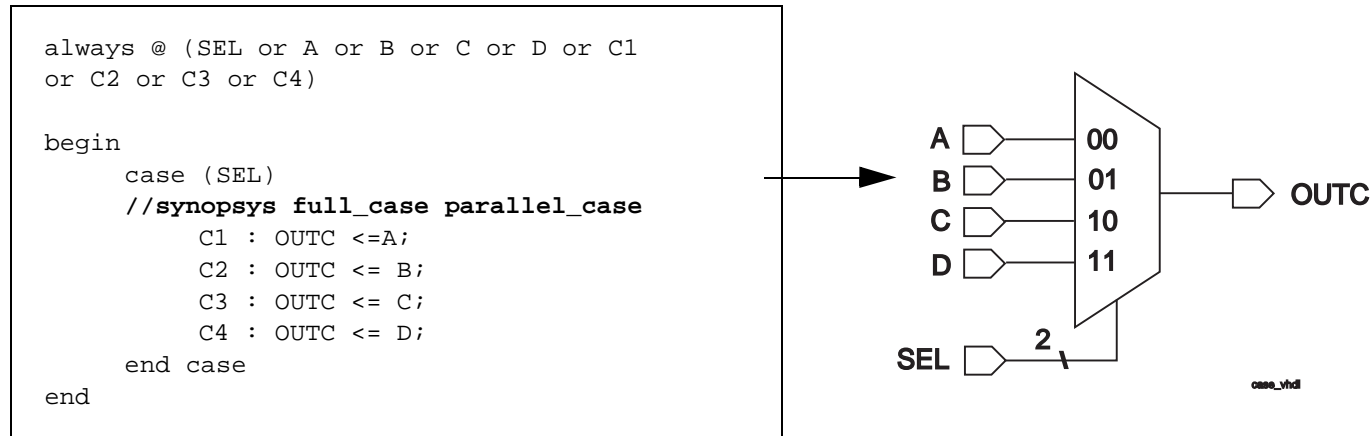


Figure 11: Case Statement Implementation With Verilog Directives

```

when others => outputs <= "----"
/* outputs assigned to don't care */
    
```

As in VHDL, if-then-else statements are implemented with priority encoded multiplexers.

Pipelining/Retiming

FPGA Compiler can apply retiming on a mapped design by moving the registers across combinatorial logic in order to minimize the critical combinatorial path. The command used to perform this operation is balance_registers.

It is possible to automatically pipeline a purely combinatorial design (e.g. multiplier) by adding a chain of registers at the outputs of the design at the RTL level, then applying balance_registers. Figure 12 illustrates a case, where the designer added a chain of three registers at the output of a combinatorial circuit in order to pipeline the design with balance_registers. This will nearly triple the throughput performance of the design with almost no increase in area but an increase in latency.

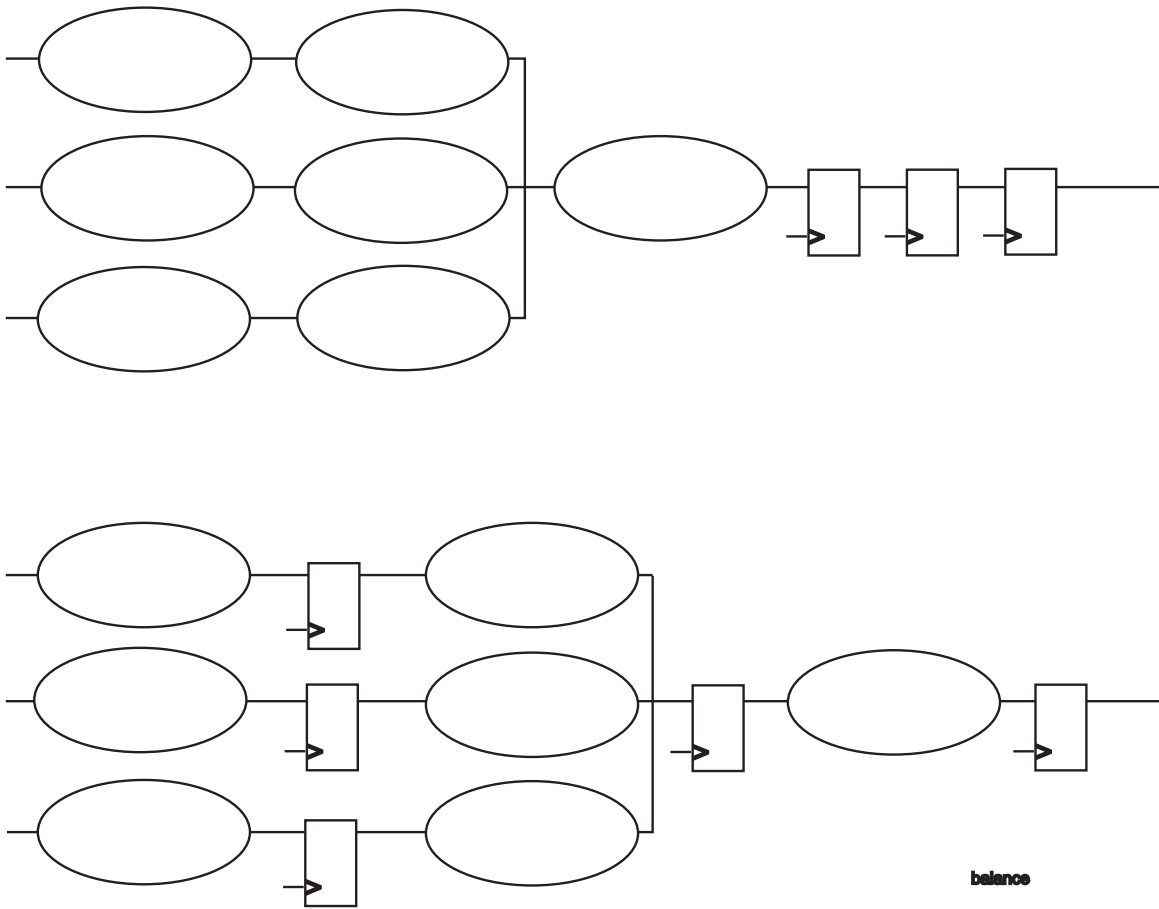


Figure 12: Effect of Balance_Registers

Because Xilinx FPGAs are extremely rich in registers, pipelining and retiming can dramatically improve the performance of a design with little or no increase in area.

Note: `balance_registers` does not apply to the XC4000 family, since the corresponding FPGA libraries are composed of CLBs and IOBs. The absence of individual registers in the library makes retiming impossible. Pipelining for XC4000 designs must be programmed in HDL code by the user.

Coding Style

Hierarchical Partitioning

It is important to hierarchically partition a design for best results and faster compile times, so that:

- Related functional blocks are grouped under a level of hierarchy. Recall Figure 3's block diagram of a hierarchically partitioned design. Each one of the four top-level blocks represents a different function in the design (e.g. FSM, decoder, PCI interface, ALU...). Partitioning the chip this way may greatly improve the place and route process.
- The portions of the design that have different design

goals (performance, area) are separated in different hierarchical blocks. This enables the designer to apply different compile strategies to these blocks.

- Boundaries of hierarchical blocks are registered as shown in Figure 2. This can be realized by partitioning the design into blocks that have registered outputs. In this case, hierarchical boundaries can be preserved and FPGA Compiler will be given smaller blocks to optimize at once, which will result in a gain in compile time and eventually in improved quality of results.

FSM Encoding

The encoding of Finite State Machines (FSM) can have dramatic repercussions on the performance and area of the circuit. FPGA Compiler has the ability to extract an FSM from a mapped design using a list of user-provided registers. The FSM can be automatically re-encoded and relinked in the design.

This methodology is not recommended for Xilinx designs. Though very flexible, it is not easy to use and has limitations for large FSMs. Note: this is not applicable to the XC4000 series. For the XC4000 series, it is best to explicitly encode state machines.

```

architecture FSM of EXAMPLE6 is
    type STATE_TYPE is (IDLE, GO, YIELD, STOP);
    signal CURRENT_STATE, STATE_NEXT: STATE_TYPE;
    attribute ENUM_ENCODING: STRING;
    attribute ENUM_ENCODING of STATE_TYPE: type is
        "0001 0010 0100 1000";
begin
    COMBO: process (CURRENT_STATE) begin
        case CURRENT_STATE is
            when IDLE => STATE_NEXT <= GO; OUT1 <= "01";
            when GO => STATE_NEXT <= YIELD; OUT1 <= "11";
            when YIELD => STATE_NEXT <= STOP; OUT1 <= "10";
            when STOP => STATE_NEXT <= IDLE; OUT1 <= "00";
        end case;
    end process COMBO;
    SEQ: process (CLOCK, RESET) begin
        if (RESET = '0') then
            CURRENT_STATE <= IDLE;
        elsif (CLOCK'EVENT and CLOCK = '1') then
            CURRENT_STATE <= STATE_NEXT;
        end if;
    end process SEQ;
end FSM;

```

Figure 13: VHDL Example of One-hot Encoded FSM

The recommended method is to encode the FSMs in the HDL code. The advantage of this method, is that the designer has full control over state encoding and can choose different encodings for each FSM.

VHDL

Here are some guidelines to follow for FSMs described in VHDL:

- Use enumerated types for the states.
- Use the `ENUM_ENCODING` directive to assign the states to the desired values.
- Use a `CASE` statement to describe the logic of the next state (and eventually the logic of the outputs).
- To create smaller and faster circuits **do not** use the `WHEN OTHERS` statement. To be able to use this statement all states need to be described in the `CASE` statement. Using the `WHEN OTHERS` statement will synthesize a state machine that can recover from unreachable states, at the expense of area and circuit speed. For an example see Figure 13, an One-hot Encoded FSM.

Verilog

The use of the Verilog `parameter` command enables manual state encoding. Figure 14 shows an example of an

One-hot Encoded FSM written in Verilog. Even though the `full_case` and `parallel_case` directives are used in this example, `parallel_case` is not required since the states are mutually exclusive constants. Using `full_case` prevents synthesis of unwanted latches leaving room for optimization since it marks all other possible state values as “don’t care” for the `OUT1` function.

Miscellaneous Coding Guidelines

These factors can have crucial effects on the area and performance of a design:

Comparators

Always include one of these arithmetic packages `ieee.std_logic_unsigned.all` or `ieee.std_logic_signed.all` before each entity declaration. If no arithmetic package is used, comparators inferred in the corresponding architecture will be synthesized with glue logic, as magnitude comparators, and mapped to lookup tables. Including an arithmetic package infers an implementation that uses the fast carry chain. The implementation may be slightly larger, but will be much faster than the magnitude comparator.

```

module EXAMPLE6 (RESET, CLK, OUT1);
input RESET, CLK;
output [1:0] OUT1;
parameter [3:0] IDLE=4'b0001, GO=4'b0010, YIELD=4'b0100, STOP=4'b1000;
reg [1:0] CURRENT_STATE, STATE_NEXT;
always @ (CURRENT_STATE)
begin: COMBO
    case (CURRENT_STATE) // synopsys full_case parallel_case
        IDLE: begin STATE_NEXT = GO; OUT1 = 2'b01; end
        GO: begin STATE_NEXT = YIELD; OUT1 = 2'b11; end
        YIELD: begin STATE_NEXT = STOP; OUT1 = 2'b10; end
        STOP: begin STATE_NEXT = IDLE; OUT1 = 2'b00; end
    endcase
end
always @ (posedge CLK or negedge RESET)
begin: SEQUENTIAL
    if (RESET == 1'b0)
        CURRENT_STATE <= IDLE;
    else
        CURRENT_STATE <= STATE_NEXT;
    end
end module

```

Figure 14: Verilog Example of One-hot Encoded FSM

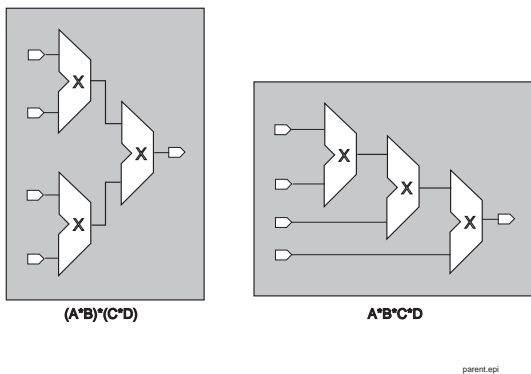


Figure 15: Effect of Parenthesis In Arithmetic Expressions

Parenthesis in Arithmetic Expressions

Parenthesis in expressions can dramatically modify the critical paths in the design. For example, Figure 15 shows that $(A*B)*(C*D)$ will infer a balanced tree of three multipliers. This is ideal if all inputs have the same arrival time. $(A*B*C*D)$ infers an unbalanced tree, which is useful if the inputs have different arrival times.

Designing With XC4000X I/O

FPGA Compiler is capable of inferring input buffers, output buffers, and non-registered bi-directional I/O. With the XC4000X IOB structure, there are input and output registers that can be used in input, output, and bi-directional I/O. Certain types of I/Os or coding styles require special handling in FPGA Compiler, e.g, if an I/O port uses IOB registers or if the HDL code tri-state behavior is not in the top-level HDL file.

Generally FPGA Compiler can infer simple I/O defined as a top-level port which requires an input buffer, output buffer, output tri-state, input register, or output register. If the designer wants to use an input or output register for a top-level port, the HDL that infers the register must not describe an asynchronous set or reset. The XC4000X I/O flip-flops do not have an asynchronous set or reset which a designer can toggle. A bi-directional pin which uses only an input buffer and tri-stateable output buffer is also a simple I/O.

To infer simple bi-directional I/O using FPGA Compiler, the HDL describing this type of I/O must reside at the top-level. An abnormal termination for `insert_pads` will be reported if the I/O is not described at the top-level HDL, or FPGA Compiler may also write out an IOB_4000 cell into the Xilinx .SXNF file. In the Xilinx place and route environment, the presence of an IOB_4000 cell will create an unexpanded block error from `ngdbuild` and M1 will not process the design.

```

module top (a,b,c.....);

input a;
input b;
.
.
.
assign internal_sig = a;
assign a = (control) ? (output_sig) : 1bz;
.
.
endmodule

```

Figure 16: Verilog Bi-directional Pin

For both Verilog and VHDL, a conditional assignment statement which shares a common port with a continuous assignment statement will infer a simple bi-directional I/O, a bi-directional pin composed of an IBUF and an OBUFT.

For Verilog, a simple bi-directional pin would be described in Figure 16.

A simple bi-directional pin for VHDL would be described in Figure 17.

For both Verilog and VHDL, the HDL code that describes the input path, output path, and control signal are at the top-level. If one or more of the three mentioned HDL structures is in a lower-level of hierarchy, FPGA Compiler may not be able to infer the simple bi-directional I/O.

The following compile strategies can be tried when it is not possible to move the entire behavior of the simple bi-direc-

tional I/O to the top-level HDL (Using one or more may produce a workable solution.):

1. Execute the `set_port_is_pad/insert_pads` commands **after** compile.
2. Execute the `set_port_is_pad/insert_pads` commands **before** compile.
3. Run the `insert_pads` command with the `thru_hierarchy` option.
4. Flatten the design using `compile -ungroup_all`. Follow up `compile` with `set_port_is_pad` and `insert_pads` commands.
5. On bi-directional ports where 10B_4000 cells were not replaced with the proper IBUF/OBUFT combination, instantiate the IBUF/OBUFT cells. Do not place a `port_is_pad` attribute on that port.

```

entity top is
port(A: inout STD_LOGIC; B: in STD_LOGIC,...);
end top;
architecture inside of top is
.
.
begin

internal_sig <= A;
process
begin
if(control == 1) then
A<=output_sig;
else
A<=z;
end if;
end process;
.
.
end inside;

```

Figure 17: VHDL Bi-directional Pin

Any type of bi-directional I/O that uses an IOB register, an IOB register with an IOB tri-state, and/or a registered and non-registered version of an input/output signal, cannot be inferred via FPGA Compiler. FPGA Express™ does not have this limitation. If using FPGA Express is not an option the best way to handle complex I/Os is to instantiate the cells. For names of I/O cells and their pins, refer to Appendix A of the XSI User Guide.

Designing Complex Muxes

Large XC4000X designs with high resource utilization can have problems with fanout and H-function utilization. Fanout problems can be detected using the `trce` command in M1. They can be solved by careful use of global buffers and duplication of logic. Higher use of H-function generators in a XC4000X design can reduce the number of combinational logic levels and increase design speed.

Minimizing Fanout Problems

Reduction of fanout can improve the overall speed of a design. There are two basic methods to reduce fanout. Choosing either method for a particular design is highly subjective. Using global buffers may not solve a fanout problem, especially if global buffers are already in use. Duplication of logic may not be possible, since it may increase resource utilization beyond the capacity of the target XC4000X device.

The M1 timing tool, `trce`, can be used to detect high fanout nets in a design. When using `trce`, there are two ways to determine high fanout nets in a design. The first method lists all nets by their delays, worst-case net delay. The second method lists all worst-case net delays per timespec. A third method can be created using both methods.

In cases where a designer wants to list the delays of all nets in a design, starting with the worst-case delays, the designer only needs to use the routed `.ncd` file and `trce`. The syntax to use for this methodology is:

```
trce -a design_routed.ncd
```

The `trce -a` option lists all paths with combinational logic, ordered by delay. The `.twr` file generated by `trce -a` can be used to evaluate worst-case net delays. Names of the net will come from the design. Additional information on the list of worst-case delays can be gained if the designer views the nets with EPIC, the graphical editor in M1. Based on the list of worst-case delays and the sources and loads of these worst-case delays which can be listed by EPIC, a designer can select which high fanout net to optimize.

The second method of evaluating high fanout nets uses timing constraints specified by the designer. One likely cause of high fanout nets is when specified timing constraints are not met. The designer tool lists the worst-case

net delays per constraint using the following `trce` command-line syntax:

```
trce -v 1 -a design_routed.ncd design.pcf
```

With the above command line syntax, the worst-case delay per timespec will be listed. If the timing constraints are combinational, then this method yields the same information as the previous method. If timing constraints involve timespec endpoints, like RAM, FF, and/or LATCHes, high fanout nets will be indirectly implied. Where timespecs have timespec endpoints, the high fanout nets are either the clock nets for these paths or the source/loads of these paths. As with the previous method, use the `.twr` report or Epic to select which high fanout nets to optimize.

There are two basic strategies to minimize high fanout: Global buffers or logic can be duplicated. Each has its own advantages and disadvantages. In some cases, using both methods may improve overall design performance.

The use of global buffers is an easy strategy to implement in HDL. Once identified, high fanout nets which degrade performance can be improved by instantiating a global buffer that will source that net. By default a high fanout sourced by an external source will cause FPGA Compiler to infer a global buffer when the `create_clock` synthesis constraint is used. Any clock pin which is directly connected to a top-level port is a candidate for global buffer insertion. In cases where a high fanout net is internally sourced, the global buffer must be instantiated.

When instantiating global buffers in the XC4000X architecture, keep the following limits in mind: there are only 8 global buffers in the XC4000X, and there are two types of global clock buffers in the XC4000X, the BUFGLS and BUFGE. The designer should let the software decide which global buffer to use by instantiating the BUFG in the HDL code. For a net that has its source from an external pin, a dedicated clock buffer IOB should be used. It is possible to connect a global buffer to a non-dedicated IOB, but two IOBs will be used for a global buffer instead of one. Similarly, whenever a global buffer is used for sourcing internal logic, the dedicated global clock buffer IOB is no longer available as a designer I/O.

Using the global net resources may not help all fanout cases. For example, if a design has a clock divider that is the source for all logic in a design, adding global buffers to the clock divider outputs may only have minimal effect. In this case, the best solution is to duplicate logic. FPGA Compiler was designed to minimize logic, and generally does not duplicate logic to reduce fanout. From experience, the best way to implement duplicate logic is to duplicate the logic in the HDL. Using the clock divider as an example, instead of making one clock divider in the design, make four. In general, duplication of logic means duplicating the HDL code that created the logic, and placing a `dont_touch` on the code to prevent FPGA Compiler from removing the duplicate logic.

Like global buffer usage, duplication of logic can also cause problems. In cases where the original source logic was loaded from external I/O, each time the logic is duplicated the amount of I/O used increases. Similarly, duplication of logic may limit the amount of resources available for additional design enhancements later.

Improving H-function Use

A design that has high CLB utilization, a large number of logic levels per path, or poor design performance may suffer from poor H-function use. The H-function generator is a 3-input function generator in the XC4000X CLB. The H-function generator can assist in the implementation of functions up to nine variables in size in one XC4000X CLB.

A typical cause of poor H-function utilization in the HDL FPGA Compiler flow is any design with muxes larger than 8:1. An 8:1 mux can be implemented in two-levels of logic, but FPGA Compiler is not designed to utilize the H-function. 8:1 muxes inferred by FPGA Compiler use three levels of logic instead of two. Another situation which can have poor H-function use involves counters. FPGA Compiler creates counters by creating a bank of registers in front of an incrementer/decrementer. This implementation of a counter is functionally correct, but occupies two columns of CLBs instead of one.

H-function utilization in a design can be improved by using LogiBLOX or instantiation of H-MAPs. LogiBLOX is a M1 tool which builds bit-slices optimized for Xilinx architecture based on designer requirements. The LogiBLOX tool can build muxes and counters more efficiently than FPGA Compiler. (Note: for more information on LogiBLOX, please refer to the LogiBLOX User Guide in the M1 on-line documentation.) H-function generators can be increased by instantiating the HMAP_PUC primitive in the HDL code. The HMAP_PUC used in the HDL code connects the nets of combinational logic the designer wants implemented in the H-function. For example, if there was an AND gate that was connected to two input nets a1 and a2, and the output of the AND gate was connected to net a3, a designer could map this function to the H-function by instantiating the HMAP_PUC in the HDL code and connecting nets a1, a2 to pins i3 and i2 of HMAP_PUC. Net a3 would be connected to the O pin of the HMAP_PUC.

Use of LogiBLOX and/or instantiation of HMAP_PUC will increase the utilization of the H-function generator. Keep the following drawback in mind: when using LogiBLOX, the LogiBLOX -created component must be instantiated. When using the HMAP_PUC through instantiation, there are certain uses of the H-function generator that are not supported. See pages 10 through 12 of the *Xilinx Software Conversion Guide from XACTstep v5.x to XACTstep vM1.x* for more information on unsupported H-function generator use. Typically, a designer who instantiates an HMAP_PUC may try to insure H-function use by additional constraints

via a UCF file. The designer must keep in mind some of the unsupported H-function uses when doing this.

As a final option, H-function generator use can be performed by specifying the `-c` option to the `map` tool. The `-c` option tells the M1 `map` tool how much of the target FPGA resources to use. By default, `map` tries to use 97% of the resources of a target device. By telling `map` to use fewer resources, `Map` will attempt to use the function generators, including the H-function, more aggressively. The default level of 97% can be changed by specifying a new percentage, e.g.:

```
map -c 50 design.ngd
```

which means use 50% of the designs resources.

Decreasing the default `-c` option may prevent a design from routing. Several different `-c` options should be tried for a given design. The `-c` option may impact design speed as well.

Specifying M1 Timing Constraints in the M1 XSI Flow

In the M1 XSI flow a designer will apply two sets of constraints. One set of constraints is applied in the Synopsys FPGA Compiler synthesis environment. The other is applied via a UCF file in the Xilinx M1 place and route software. These two sets of constraints are not compatible, since synthesis constraints and M1 place and route constraints use different name spaces. Despite their differences, there is a way to develop M1 UCF file timing constraints which will be compatible with the designer's synthesis constraints.

Make UCF Methodology - An Overview

The overall M1 XSI design flow is as follows:

- RTL simulation
- Synthesis
- Place and route
- Static timing analysis and/or gate-level timing simulation
- Bit file or prom file generator for download to device

The designer specifies all necessary constraints to get the synthesis tool to meet the area, speed, and quality of results requirements of the design. Timing constraints used in the synthesis process should be used as a guide for creating the UCF timing constraints. Generating more constraints than needed is, in general, a good design practice with Synopsys' FPGA Compiler because it is a constraint driven tool which tends to make assumptions when it can. To achieve optimal quality of results, the developer must apply constraints to synthesis.

In place and route, the synthesized netlist is converted into the M1 data format. In this step of the design process, the designer may need to specify timing constraints which

reflect the area, timing, and quality of results desired. Unlike the synthesis constraints, where over constraining a design is often an effective way to meet performance, in the place and route step, it is more effective to use fewer and more general constraints, like PERIOD or OFFSET. If a designer chooses a place and route timing constraint strategy similar to his synthesis constraint strategy, specifying a large number of constraints and overly aggressive (impossible to meet) timing constraints, the place and route step will fail to meet performance goals. Keep in mind that in M1, the tools work hard to meet performance specifications. If an impossible goal is specified, the M1 tools will attempt to meet this impossible goal. In the M1 place and route flow fewer and more general timing constraints means better performance. Sometimes general and few timing constraints are not enough. An example of this type of situation is when the design has multiple clocks, and/or multi-cycle timing paths. In a case like this, it is necessary to be specific and increase the number of timing constraints by using the original synthesis constraints as a guide.

When place and route is finished, the routed .ncd is converted into either a .bit file or prom file which can be downloaded into the FPGA.

There are two UCF methodologies available, the first "Using OFFSET and PERIOD in a UCF File Only," generates a netlist using OFFSET and PERIOD only, the second "Using makeucf.pl" uses top level ports, inferred latches, and instantiated RAM primitives.

Methodology 1: Using OFFSET and PERIOD in a UCF File Only

In this methodology, the UCF file for the synthesized netlist is limited to two types of constraints: OFFSET and PERIOD. OFFSET is equivalent to `set_input_delay` and `set_output_delay` constraints. PERIOD is equivalent to `create_clock` constraint. Regardless of how many synthesis constraints are specified for synthesis, the designer restricts his UCF file to only contain OFFSET and PERIOD constraints. To use this timing constraint strategy, the designer has only to know the top-level port names.

Details of Methodology 1

Methodology 1 takes advantage of the ability for Synopsys FPGA Compiler to preserve the top-level port names in the resulting Verilog/VHDL netlist. Thus if a designer has a top-level port called CLK, the port CLK can be referenced in a

UCF file. An example of how to write the Verilog part list is shown in Figure 18.

```
module top (A,B,C,D,CLK);
input A;
input B;
input C;
output D;
input CLK;
.
.
.
endmodule
```

Figure 18: Verilog Example

The VHDL counterpart is shown in Figure 19.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity top is
port(A,B,C: in STD_LOGIC; D: out
STD_LOGIC; CLK: in STD_LOGIC);
end top;

architecture inside of top is
.
.
.
begin
.
.
.
end inside;
```

Figure 19: VHDL Example

Based on these HDL examples, the synthesized netlist will contain five pins which can be referenced as A, B, C, D, and CLK in a UCF file. Remember, M1 is case-sensitive. When referencing a top-level port, make sure that the correct spelling and case is always used.

After writing the HDL code for a design based on the designer's specific top-level port names, the designer can create a UCF file before synthesis is finished since the resulting top-level pin names in the FPGA design are known. In this example, the design must run at 40MHz. Input port A has an input delay of 5 ns. Input port B has an input delay of 5 ns. Input port C has an input delay of 10 ns. Output port D has an output delay of 15 ns. Based on these timing requirements, the UCF file for these requirements is as shown in Figure 20.

```

NET "CLK" TNM="CLK";
TIMESPEC TS01 = PERIOD:"CLK" 25;
NET "A" OFFSET = IN : 5 : BEFORE : "CLK";
NET "B" OFFSET = IN : 5 : BEFORE : "CLK";
NET "C" OFFSET = IN : 10 : BEFORE : "CLK";
NET "D" OFFSET = OUT : 15 : AFTER : "CLK";

```

Figure 20: UCF File Requirements

Methodology 1 is a good method to try as a first attempt in meeting performance requirements. Based on the results from Methodology 1, the designer will be able to assess how easy or difficult it will be to achieve the desired performance.

Methodology 1 allows creation of an UCF file at the same time as the HDL code. This method works well for designs that are highly synchronous and do not have multi-cycle timing paths. Even if a design is highly synchronous and doesn't have multi-cycle timing requirements, the use of only PERIOD and/or OFFSET may not be enough. It may be necessary to use FROM:TO, and FROM:TO:TIG, especially if a design has multi-cycle timing paths or there are paths in the design the M1 tools should not optimize. When method 1 fails, there is a method 2, which uses additional commands in the Synopsys synthesis script and a Perl script to create a UCF file.

Methodology 2: Using makeucf.pl

In Methodology 2, the UCF file that is created has a list of all top-level ports, inferred registers, inferred latches, and

instantiated RAM primitives. The Perl script, `makeucf.pl`, lists these.

Details of Methodology 2

Begin using methodology 2 by modifying the basic M1 compile script as shown in Figure 21, to include the commands `all_registers`, `all_inputs`, and `all_ports`.

The `dc_shell` commands `ungroup -all -flatten`, `all_inputs`, `all_outputs` and `all_registers` must always be executed after `replace_fpga` and before writing out the synthesized netlist.

Next, use the above compile script template for synthesizing the design. If Synopsys Design Analyzer is used to run the script, do not delete the `view_command.log` file that is created. Keep this file. It is part of methodology 2. If `dc_shell` is used to run the script, be sure to create a log file called `view_command.log`. This can be done by executing `dc_shell` with the following options (In this example, the compile script is called `run.script`, Figure 22).

```

read-f VHDL "design.vhd."

set_port_is_pad "*"
insert_pads

/* Place synthesis and timing constraints here */
compile
replace_fpga

ungroup -all -flatten
all_inputs
all_outputs
all_registers

write -f xnf -h -o "design.sxnf"

```

Figure 21: Script Modifications for Example of Methodology 2

When synthesis is finished and the `view_command.log` has been created, place the `view_command.log` file in the same directory as the Perl script `makeucf.pl`.

Invoke the Perl script by typing at the UNIX prompt:

```
makeucf.pl
```

A file called `inst.tnm` will be created. `inst.tnm` will contain a list of `INST-TNM` statements that a designer can use to attach TNMs. After creating the desired TNMs, the designer can cut and paste the `INST-TNM` lines into a UCF file for creating `FROM:TO` and/or `FROM:TO:TIG` timespecs. For more information on TNMs, `FROM:TO`, and `TIG`, please refer to the constraints chapter of the Xilinx Libraries Guide.

Note, if the designer has instantiated FDP, FD, FDC flip-flops, the name for these flip-flops name listed by `makeucf.pl` in the `inst.tnm` file will not be correct. For any instantiated FDP, FD, or FDC the string `/$1I13` must be appended to the end of the instance name if a TNM is to be attached. The `makeucf.pl` script is available via the Xilinx web site at www.xilinx.com. An example script is shown in Figure 23.

Conclusion

This paper has detailed several methods which can be used to fully utilize the capabilities of the FPGA Compiler. It

has shown how to use the compiler in the most efficient manner possible over a broad range of issues. In the event that you desire further information contact your local sales representative or on the World Wide Web at: www.xilinx.com or www.synopsys.com.

The Perl script `makeucf.pl` is available at the Xilinx web-site at www.xilinx.com.

The techniques in this application note are suggestions for achieving high performance with high density FPGA's using Synopsys' FPGA Compiler. All of these techniques were gathered from many Xilinx/Synopsys users. None, some, or all of the techniques in this section can be used. To be successful at high density FPGA design with high density synthesis software, keep things simple. The designer should write his HDL code for the hardware in the FPGA technology. If it isn't clear to a designer what types of logic should be synthesized, chances are the synthesis tool won't make a good choice. Use careful and well considered judgement when creating Synopsys synthesis constraints, and/or M1 constraints. Over constraining or using constraints that are not relevant to design performance can unnecessarily lengthen synthesis/place and route time. High density FPGA design requires careful pre-planning of the synthesis and place and route process.

```
dc_shell -f run.script | tee view_command.log
```

Figure 22: Example Compile Script

```

emacs@camaro.xsj.xilinx
Buffers  Files  Tools  Edit  Search  Mule  Help

Input Ports
INST "CLK" TNM=;
INST "DMINS" TNM=;
INST "DPLUS" TNM=;
INST "RST" TNM=;

Output Ports
INST "ADDRBUS<0>" TNM=;
INST "ADDRBUS<1>" TNM=;
INST "ADDRBUS<2>" TNM=;
INST "ADDRBUS<3>" TNM=;
INST "ADDRBUS<4>" TNM=;
INST "ADDRBUS<5>" TNM=;
INST "ADDRBUS<6>" TNM=;
INST "CRC16BUS<0>" TNM=;
INST "CRC16BUS<10>" TNM=;
INST "CRC16BUS<11>" TNM=;
INST "CRC16BUS<12>" TNM=;
INST "CRC16BUS<13>" TNM=;
INST "CRC16BUS<14>" TNM=;
INST "CRC16BUS<15>" TNM=;
INST "CRC16BUS<1>" TNM=;
INST "CRC16BUS<2>" TNM=;
INST "CRC16BUS<3>" TNM=;
INST "CRC16BUS<4>" TNM=;
INST "CRC16BUS<5>" TNM=;
INST "CRC16BUS<6>" TNM=;
INST "CRC16BUS<7>" TNM=;
INST "CRC16BUS<8>" TNM=;
----- inst.tnm (Fundamental)--L18--Top-----
INST "PIDBUS<6>" TNM=;
INST "PIDBUS<7>" TNM=;

#Hierarchical-Alphabetical Sort of
#Clock-sensitive Instances

#Hierarchy Level 0 Sort
INST "rickjames" TNM=;

#Hierarchy Level 1 Sort
INST "U1/SNGDTA_reg" TNM=;
INST "U2/datareg0_reg" TNM=;
INST "U2/datareg1_reg" TNM=;
----- inst.tnm (Fundamental)--L68--25%-----
INST "U9/counter_reg<3>" TNM=;
INST "U9/counter_reg<4>" TNM=;

#Hierarchy Level 2 Sort
INST "U6/ganop/cntrout_reg<0>" TNM=;
INST "U6/ganop/cntrout_reg<1>" TNM=;
INST "U6/ganop/cntrout_reg<2>" TNM=;
INST "U8/bashful_1/Q_reg" TNM=;
INST "U8/bashful_2/Q_reg" TNM=;
INST "U8/bashful_3/Q_reg" TNM=;
INST "U8/bashful_4/Q_reg" TNM=;
INST "U8/bashful_5/Q_reg" TNM=;
INST "U8/doc_1/Q_reg" TNM=;
INST "U8/doc_2/Q_reg" TNM=;
INST "U8/doc_3/Q_reg" TNM=;
----- inst.tnm (Fundamental)--L116--60%-----

```

Figure 23: Example Makeucf.pl Script