DynaChip®

VHDL / Verilog Coding for FPGAs

Produced by: Technically Speaking, Inc for DynaChip Corporation

Introduction: FPGA designs have traditionally been entered using schematic capture and vendor specific libraries. This use of proprietary tools and macros gives designers a high degree of control and optimization at the device level, but it inherently limits the design to that particular product or technology.

On the other hand, VHDL and Verilog HDL offer a means of describing hardware and functionality at a very high level --a technology independent vantage. This affords designers an unprecedented degree of latitude and productivity. Ideally, a target technology can be chosen at a later point in the design cycle. In the meantime, the chip or system level functionality can be modeled and completely verified in a behavioral environment, as shown in figure 1.

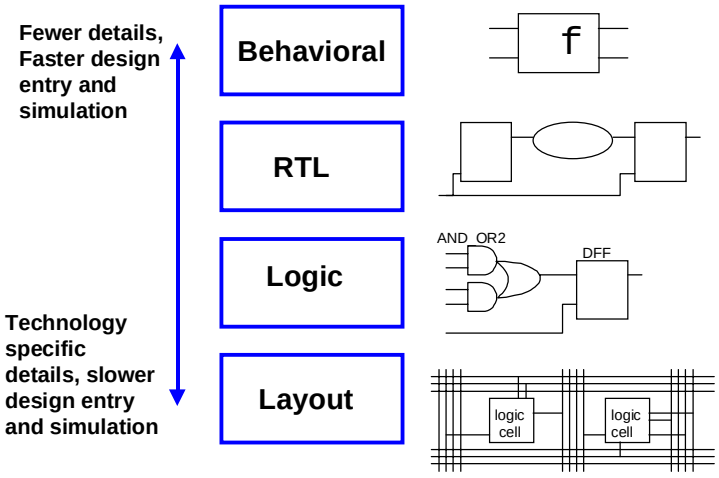


Figure 1. Levels of Abstraction for VHDL

Both VHDL and Verilog have their origins in "Hardware Modeling and Verification". That means simulation! – and not necessarily synthesis. The IEEE standards 1076 (VHDL) and 1364 (Verilog) are exhaustive with respect to simulation, but define only broad parameters for synthesis. Considering that potential target technologies—ASICs, FPGAs, CPLD, Etc. are quite diverse and entirely new ones are being created, it would be impossible to completely pre-define **optimal** synthesis requirements for each.

Therefore, completely generic HDL code is usually not optimal for FPGAs.

What makes FPGAs unique?

FPGAs are user programmable ASICs. As such, they must accommodate mixed combinatorial and sequential logic. FPGAs are generally characterized by coarse grain internal and I/O logic blocks. They may contain LUTs (Look-Up Tables), bi-directional I/O, dedicated registers and or latches, control muxes, distributed or block RAM, global Clock buffers, and programmable routing resources.

The DynaChip FPGA family logic cells contain dedicated “And-Or” in addition to Multiplexer, Arithmetic and RAM logic.

It’s worth noting that the initial primary target technology of VHDL and Verilog were traditional ASICs, which are characterized by fine grain architectures. That means that transistors at the substrate level are formed into gates, through the process of metalization and fabrication. SSI, MSI, and complex logic is built from this starting point.

As compared to traditional ASICs, FPGAs increase flexibility, reduce the total design cycle and enhance the “time to market”.

However, the flexible programmable FPGA architecture presents a formidable challenge to HDL synthesis compilers.

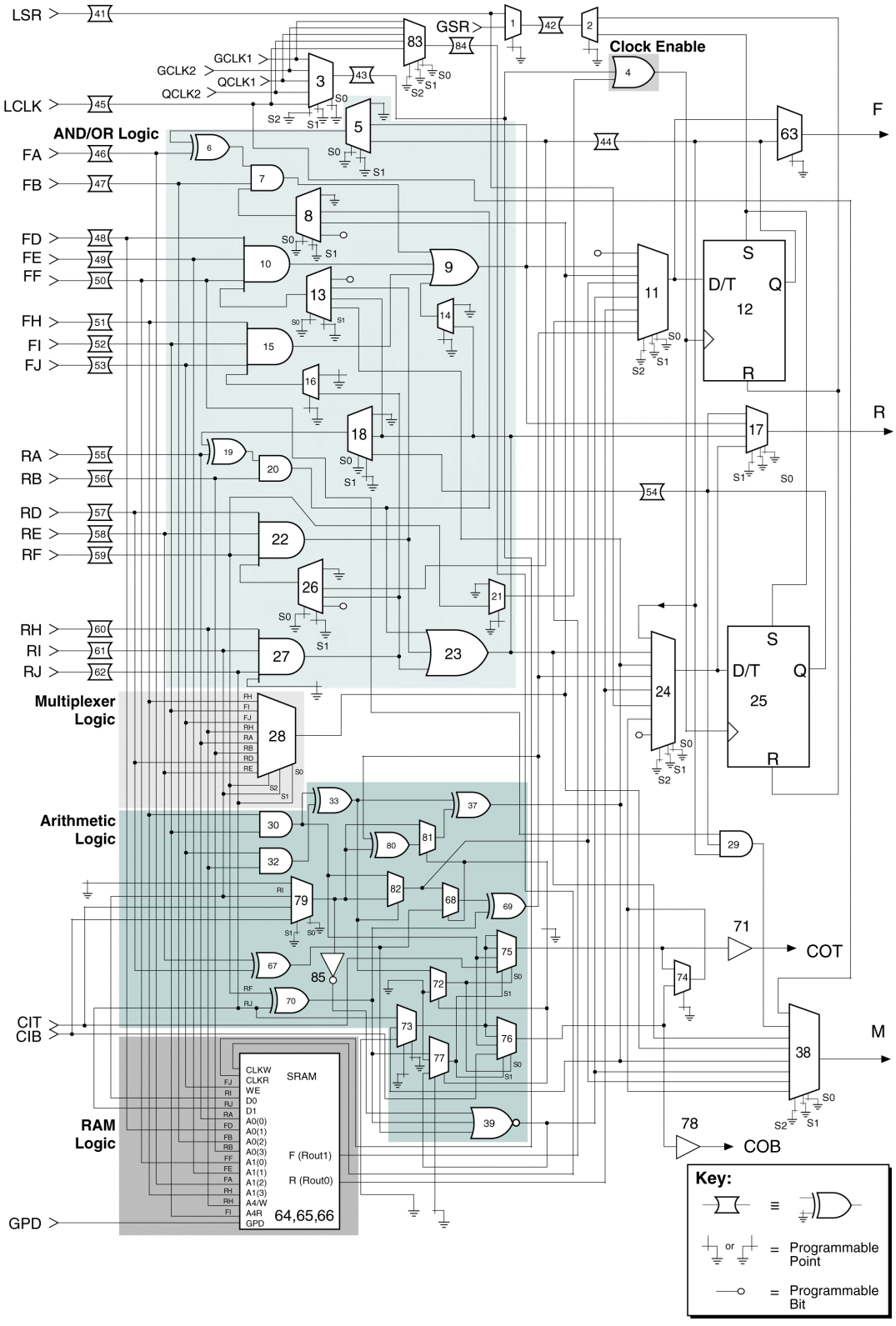


Figure 2. DynaChip Logic Cell DY6000 Family

Within a given FPGA logic block, a finite amount of combinatorial logic may be implemented and driven through buffered or registered outputs. The process of selecting what functionality goes into which logic cells is called **mapping**.

This is usually the first part of the implementation process as shown in figure 3.

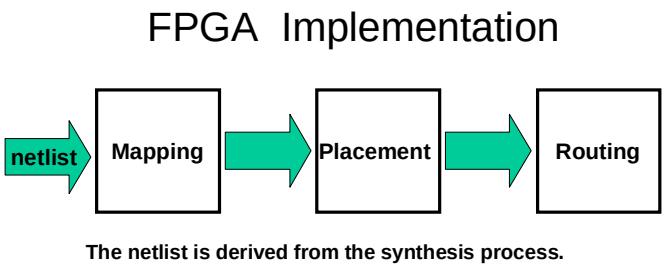


Figure 3. FPGA Implementation Process

The most challenging aspect of synthesis for FPGAs is that all three stages of implementation are inter-related, they are in-fact cumulative and dependent. The mapping affects the placement, which in turn impacts the routing.

Clocks, high fan-out signals, and logic levels are the most difficult items to optimize from a synthesis standpoint. Each additional logic level represents an irreducible block delay plus the necessary routing.

The DynaChip FPGA family uses Active Repeaters, a patented technology for buffering routing resources. This greatly decreases routing delays, increases performance and predictability by driving fixed loads. Even so, if the logic is poorly mapped, additional logic levels will undermine the routing advantage.

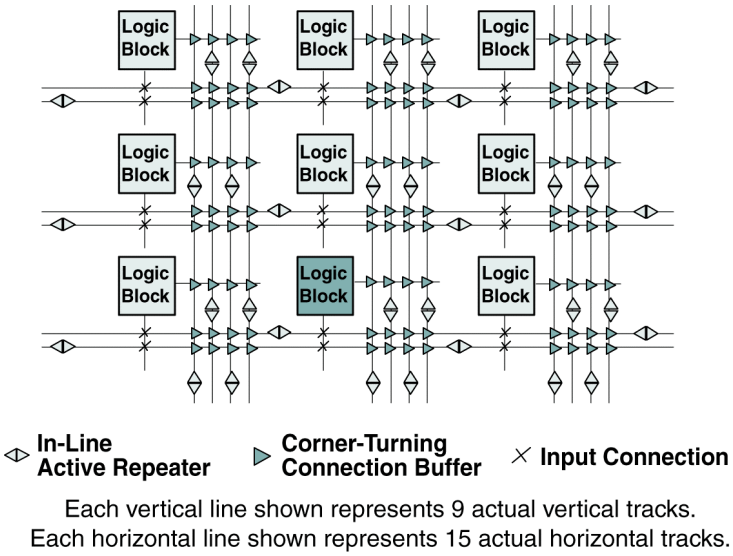


Figure 4. DynaChip Active Repeater routing resources.

The bottom-line is that design optimization, as measured in terms of maximum frequency and or area utilization must start with the synthesis process.

The ***QOR*** (Quality of Results) of the synthesis process is driven by two primary factors: the **user coding style** and the compiler's ability to **infer** optimal logic and or mapping for the particular FPGA architecture.

The ability to **infer** also includes any device specific resources or features that enhance chip level implementation, but are unique to the target FPGA!

The Synthesis Process

In an attempt to understand the broader concepts, lets examine the synthesis process. There are four distinct steps in VHDL or Verilog compilation.

- **Analyzation**
  The design unit is checked for syntactic errors, once finalized, it is stored in the "work" library.
- **Elaboration**
  The design hierarchy is fleshed out, starting from the top. A unique copy of each sub-module instance is created. Loops are unrolled, etc.
- **Execution**
  The model is simulated in discrete time steps in a behavioral environment. This is driven primarily by events on signals, which then trigger processes.

- **Synthesis**
  A netlist description of the logic is generated, in either an industry standard or vendor specific format.

From the standpoint of menu selection, most synthesis compilers do not distinguish between the “Synthesis” and “Elaboration” stages of processing. This is due largely to the fact that synthesis must include elaboration. They are nonetheless different and distinct steps in the overall compilation.

Elaboration is necessary to resolve hierarchy, create unique instances and verify that data-type restrictions are adhered to. For instance, during elaboration, the use of the **‘+’** operator infers that an *adder* be built. At this point however, only the behavioral or functional *adder* has been defined.

Meanwhile, Synthesis is the process of actually mapping the elaborated design to the target technology library. At this stage, a decision will be made concerning which available *adder* from the vendor’s library to use. **That choice is driven by the size requirements, along with user defined constraints for speed or area**. The output of the synthesis process is the netlist, in either a vendor specific or standardized format.

Again, note that most tools do not use the VHDL terms “analyze” and “elaborate”, rather they use menu options such as “check syntax” and “compile”.

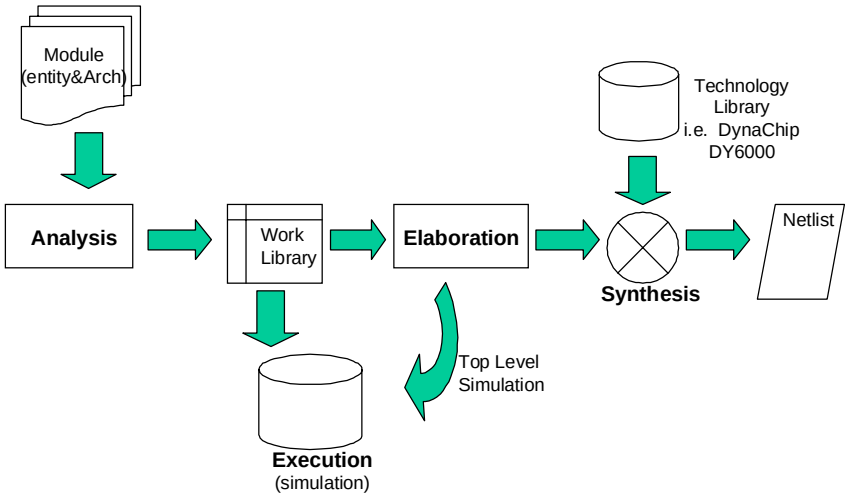


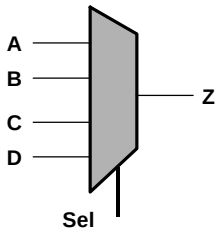
Figure 5. The Synthesis Compilation Process

11 FPGA Coding Styles Pointers

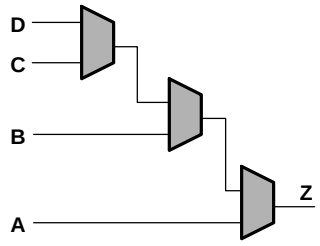
We now turn our attention to the actual coding process. We will examine 11 distinct HDL coding points that enhance design implementation within the DynaChip DY6000 product family and FPGAs in general. They relate to combinatorial logic, registers with combinatorial inputs, accessing dedicated high-speed carry logic and general guidelines.

Point 1: Prefer **case** over **if/ else if** !
Objective: Minimize FPGA logic levels
Benefit: Reduced path delay, increase design frequency.

One of the great benefits of an HDL design approach is the ability to describe relatively complex and conditional operations using simple "*if/else if*" or "*case*" statements. **Using a *case* statement will generally produce a "flatter" implementation as opposed to an *if/else*, which tends to result in "priority encoded" logic.**




```
process  ( A, B, C, D, Sel )
begin
  case ( Sel ) is
     when "00" => Z <= A;
     when "01" => Z <= B;
     when "10" => Z <= C;
     when "11" => Z <= D;
  end case;
end process ;
```




```
process  ( A, B, C, D, Sel )
begin
   If     ( Sel = "00" )  then Z <= A;
   elsif  ( Sel = "01" )  then Z <= B;
   elsif  ( Sel = "10" )  then Z <= C;
   elsif  ( Sel = "11" )  then Z <= D;
   end if;
end process ;
```

Figure 6. VHDL conditional branching example.

There are other important considerations here as well. The first is that if your conditions are indeed overlapping, (i.e. *if* **x < 3** *then*…*else if* **x < 5** *then*…) an *if/else if* statement will be required. In that circumstance, priority encoded logic will be necessary to satisfy the intended functionality. But this invariably leads to cascaded logic levels, which can dramatically reduce FPGA design performance. Avoiding overlapping conditions may necessitate re-considering the design style!

Another issue is covering all possibilities within a case statement. In VHDL, this is required, but not so in Verilog. As such, the concept of ***"full case"*** (all possible conditions covered) and ***"parallel case"*** *(* no conditions overlapping*)* are inherent within VHDL. This is usually accomplished using an "others" clause. Explicitly declaring each possible value is usually not practical when using STD_LOGIC or STD_LOGIC_VECTOR, where each element has nine possible values.

In **Verilog**, if all conditions are not specified and no "*default*" clause exist, a latch will be inferred, which may also affect the total number of logic levels required.

```
module  IF_MUX  ( Sel, A,B,C, D, Z_out ) ;
input [1:0] Sel ;
input     A, B, C, D;
output   Z_out ;
reg       Z_out ;

    always @ (A or B or C or D or Sel )
        begin
            if       ( Sel = = 2'b00)  Z_out = A ;
            else if ( Sel  == 2'b01)  Z_out = B ;
           else if ( Sel  == 2'b10)   Z_out = C ;
           else                              Z_out = D ;
        end
endmodule
```

```
module  CASE_MUX  ( Sel, A,B,C, D, Z_out ) ;
input    [1:0] Sel ;
input     A, B, C, D;
output   Z_out ;
reg       Z_out ;

    always @ (A or B or C or D or Sel )
        begin
            case  ( Sel)
              2'b00:   Z_out = A ;
              2'b01:   Z_out = B ;
              2'b10:   Z_out = C ;
             default:  Z_out  = D ;
            endcase
        end
endmodule
```

Figure 7. Verilog conditional branching execution.

For **Verilog**, if the code doesn't appear to cover all possible conditions, but actually does in the context of the design, it may be helpful to add the *"full case"* and or "*parallel case"* compiler directive to prevent latch inference or priority encoding of logic.

For **VHDL** or **Verilog**, do not describe the default assignment as being to '0'. This will cause an additional gate on the output of the mux, an assignment to *don't care* is preferable.

When using **VHDL** and the **DynaChip** architecture, another recommendation would be to use "*bit*" and "*bit_vector*" data-types (Mux only), and explicitly declare all possible values, thus avoiding the "others" or "default" clause that may produce a *nand* of all

other inputs. If this happens, the *nand* will force an additional logic level due to the inverted output.

Point 2: Group Arithmetic Operators Using Parentheses.
Objective: Minimize FPGA logic levels
Benefit: Reduce path delay, increase design frequency, enhance code readability.

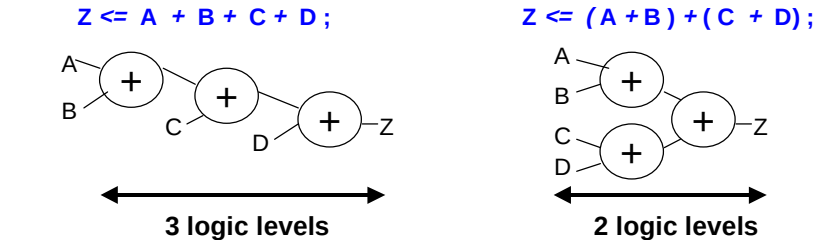
For both **VDHL** and **Verilog,** all operators (including logical) are modeled as 2 input functions. The expression:

Z <= ***A*** + ***B*** ; (VHDL) or ***assign Z*** = ***A*** + ***B*** ; (Verilog)

will elaborate to a 2 input adder with the operands **A** and **B** as inputs. With only two operands, this is appropriate. However when multiple operands exist, the process may yield unexpected results.

Z <= ***A*** + ***B*** + **C** + **D**; (VHDL) or ***assign Z*** = ***A*** + ***B***+ ***C*** + ***D***; (Verilog)

Yields a series of 2 input cascaded adders that may create multiple logic levels depending on the bit-width of the operands. Group operators with parentheses to control synthesis!



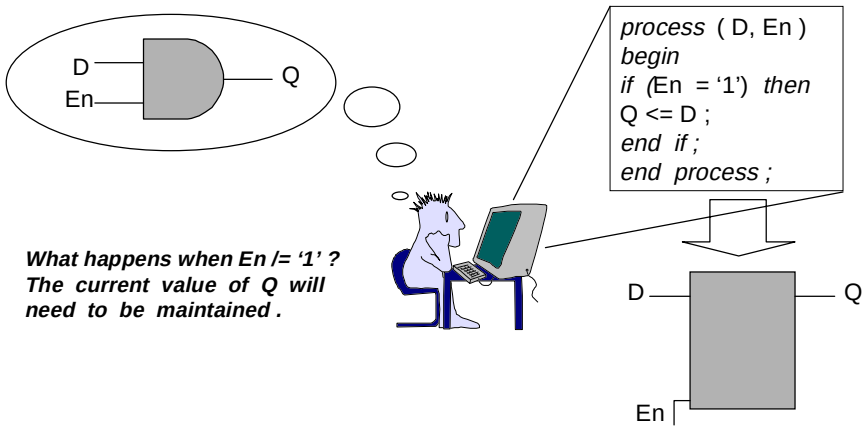
Using parenthesis to group operators has three important benefits:

- Forces minimal logic level implementation.
- Eliminates operator precedence ambiguity.
- Enhances code readability.

Figure 8. Grouping operators with parentheses.

Point 3: Avoid Inadvertent Latch Inference
Objective: Minimize FPGA logic levels
Benefit: Reduce path delay and area requirements, enhance design stability.

For both ***VHDL*** and ***Verilog*** the use of a "if" statement without an explicit "else" clause is usually considered incomplete, and will infer a transparent latch to preserve data in the event that the "if" condition is not true.



If you intend to simply gate a input signal with a control or enable, you must specify the appropriate "else" statement assigning '0' to the output.

If a latch is intended, then code it as shown, be sure however that the target FPGA technology supports internal latches without creating timing hazards and mutli logic levels.

Figure 9. Inadvertent Latch Inference

Point 4: Avoid Inadvertent Latch Inference via " *incomplete combinatorial process*"
Objective: Minimize FPGA logic levels
Benefit: Reduced path delay and area requirements, enhance design stability.

Within a combinatorial *VHDL* ***process*** or *Verilog* ***always*** block, each output must be assigned to each time the process resumes execution. If this is not the case, a latch will inferred on that output.

In the following example, there are two outputs referenced in the process, but depending on the selector expression (the signal "Sel"), only one output is actually assigned to.

```
BAD_MUX : process ( A, B, C, D, Sel, )
        begin
           case (Sel) is
             when "00" => Out_1 <= A;
             when "01" => Out_1 <= B;
             when "10" => Out_2 <= C;
             when "11" => Out_2 <= D;
         end case;
end process;
```

This leads to a transparent latch on the **Out_1** and **Out_2** signals because an explicit assignment is not made each time the process executes. For example, if **Sel = "10"**, the current value of **Out_1** will need to be maintained, hence the latch.

The are few approaches to avoid this:

1. Explicitly assign to each output under all conditions.
2. Include all outputs in an "***others"*** (VHDL) or "***default***" (Verilog) clause.
3. The best recommendation however is to separate the outputs into different functional blocks, and to use concurrent statements instead of processes or always blocks. In VHDL, this is a *conditional signal assignment*, in Verilog, it is a *data-flow* construct.

VHDL	Verilog

Out_1<= A when Sel = "00" else
B when Sel = "01" ;
 	
Out_2<= C when Sel = "10" else
D when Sel = "11" ;
	assign Out_1 = (A &(Sel==2'b00))| (B &(Sel==2'b01));
	
	assign Out _2 = (C &(Sel==2'b10)) | (D &(Sel==2'b11));

Figure 10. Concurrent signal assignment is VHDL / Verilog

Point 5: Avoid Assigning Intermediate Logic Terms.
Objective: Minimize FPGA logic levels
Benefit: Reduce path delay and area requirements.

Using variables or signals, it is quite common and intuitive to assign intermediate combinatorial logic terms to other signals. Functionality speaking, this is quite appropriate.

The problem however, is that FPGAs used either LUT or dedicated *And-Or* logic to implement such functionality. These structures are generally one dimensional in nature, such that a certain of inputs resolve to a single output. See figure 11.

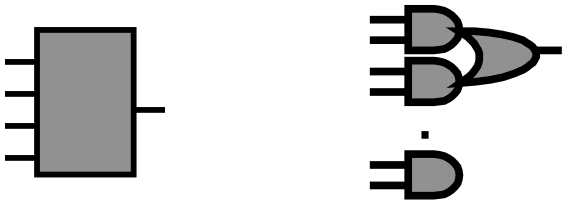


Figure 11. LUT or Dedicated *And-Or* Logic

That means that if an intermediate value is assigned to more than one target, it will necessitate an additional logic level to meet the fan-out requirement.

```
process ( A,B,C,D )
  begin
    F <= A and B or C ;
    G <= A and B or D ;
end process ;
```

```
signal H : bit ;
process ( A,B,C,D )
  begin
    H <= A and B ;
    F <= H or C ;
    G <= H or D ;
end process ;
```

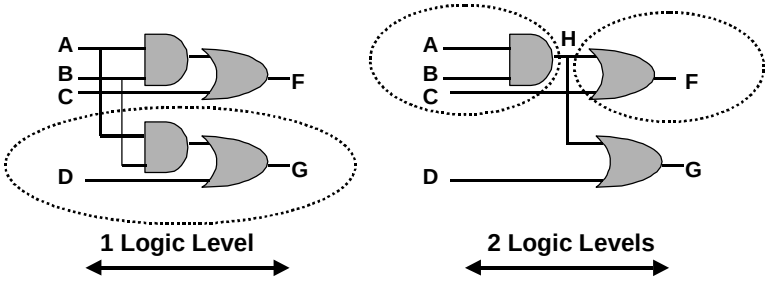


Figure 12. Replicating Gate Inputs

As shown in figure 12, replicating the input gate eliminates the fan-out requirement and allows the logic to be combined to the minimal logic-level. It may be a single logic level if the total number of inputs does not exceed the fan-in capability of the FPGA logic cell.

Point 6: If using *if/else if*, make critical signal first in the conditional branches
Objective: Reduce path delay for critical signals.
Benefit: Faster design performance.

When using "*if/else if*" for multi-conditional execution, priority encoded logic will likely be generate. This is always true when the conditions overlap, and quite likely when they do not. It should be noted that *if/else if* is normally used to indicate priority in a group of sequential statements.

With that in mind, we should anticipate that the synthesis compiler will build logic in the same order that the sequential statements are parsed. See figure 13.

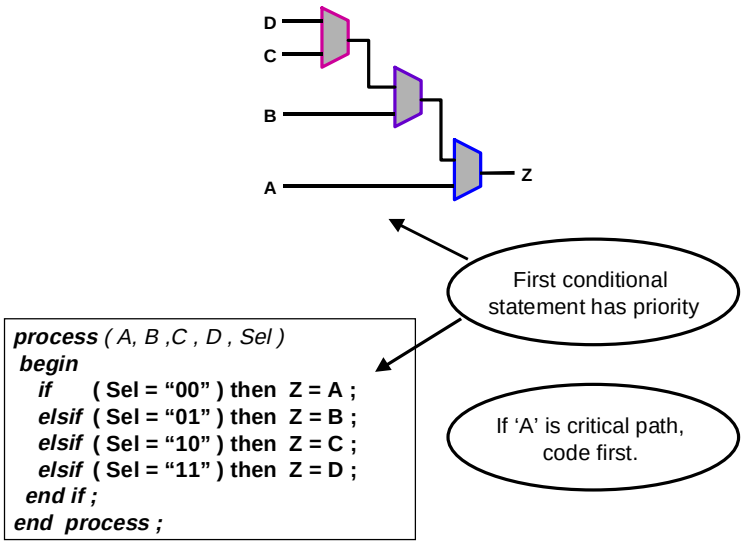


Figure 13. Priority Encoding

Ensure that your critical signal is coded first in an "*if / else if*" statement. This might also be appropriate if the particular input is a "late arriving" signal.

Point 7: Use OHE (One Hot Encoding) for State Machines.
Objective: Reduce wide gating requirements.
Benefit: Minimal logic levels, faster clock rates.

There are various approaches to state-machine encoding, the most intuitive would be sequential (binary). However, the larger the state-machine, the greater the number of terms and control inputs that must be decoded. That means potentially wide gating requirements and here in lies the problem for FPGAs.

Each logic cell can accommodate combinatorial logic up to its fan-in capability. In the case of the Xilinx XC4000 and the DynaChip DY6000, that capacity is nine. When that threshold is exceeded, the logic is cascaded across multiple logic-cell blocks.

As mentioned earlier, this increases the path delay substantially. Each block has a fixed and irreducible propagation delay, in addition to the routing between blocks. The DynaChip family has the advantage of fixed routing delays, but the data path would still suffer from a multi logic-level implementation.

At the same time, FPGAs are “register rich”, with dedicated storage elements within both the internal core and I/O. *One Hot Encoding* leverages the unique FPGA architecture by using flip-flops to actually represent each state, hence the name OHE—one flip-flop is “hot” or active per state.

The flip-flops are strung together in a shift-register like structure. Contrary to some interpretations, OHE does not necessarily mean self-decoding (although that may be possible depending of the exact nature of the logic). There is some requirement for decoding next-state logic, but the number of input terms is reduced substantially. That makes it easier to implement in one logic level, using the registered output and maintaining the maximum clock frequency possible for the device. See figure 14.

At the present time, synthesis compilers supporting the *DynaChip* product family generally do not utilize an optimal OHE implementation. The user can however, one-hot encode the state-machine directly. See figure 15 for VHDL and Verilog examples.

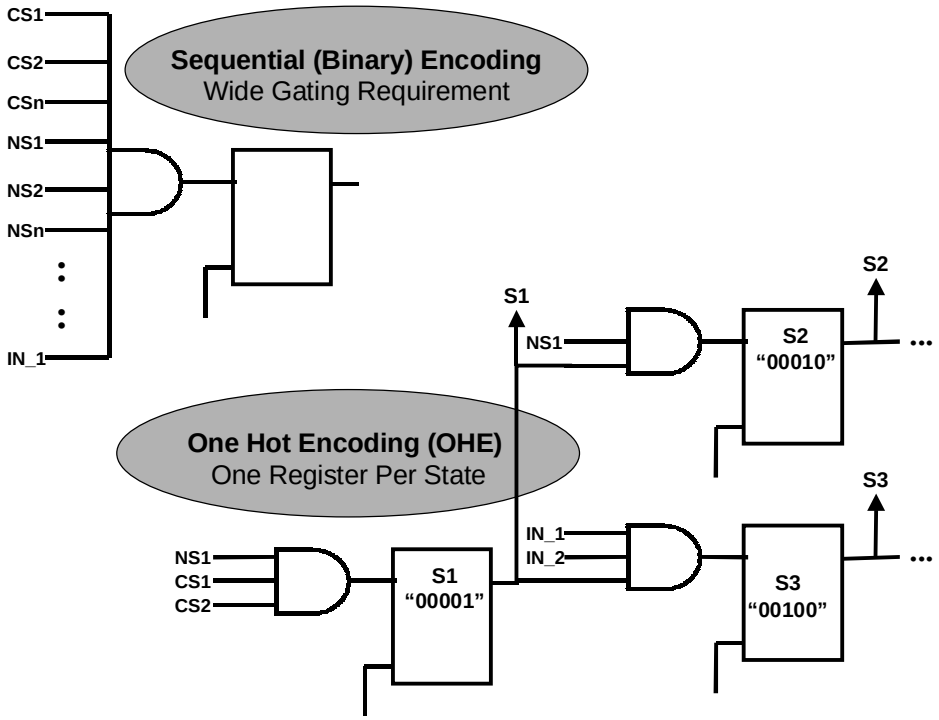


Figure 14. Sequential Encoding Vs. One Hot

<VHDL>

```
entity State_Mach is
port ( Cond_1, Cond_2, Cond_3 : in boolean ;
       Clk, Rst : std_logic;
       D_out : out std_logic ) ;
end State_Mach ;

architecture DynaChip of State_Mach is
subtype My_OHE is bit_vector ( 3 downto 0 );    --declare subtype to be used
 constant S1 : My_OHE := "0001" ;                -- declare states as constants, of the defined subtype
 constant S2 : My_OHE := "0010" ;
 constant S3 : My_OHE := "0100" ;
 constant S4 : My_OHE := "1000" ;
signal State, Next_State : My_OHE ;              -- declare signal of the same subtype
begin
Seq :process (Clk, Rst)
  begin
   if Rst = '1' then State <= S1;
    elsif (Clk'event and Clk ='1')  then
     State <= Next_State;
  end if ;
end process;
    . . .
Comb: process ( Cond_1, Cond_2, Cond_3, State )
    begin
case State is
when S1 =>
if Cond_1 and Cond_2 then
Next_State <= S2 ; else
Next_State <= S3 ;
    . . .
```

<Verilog>

```
module  OHE_Statmach  ( In1,In2, Clk,Rst,Out1) ;
input   In1,In2, Clk,Rst ;
output  Out1 ;
reg Out1 ;
reg [2:0] State ; // to hold current value
parameter  [2:0]  S1 = 3'b001,
                  S2 = 3'b010,
                  S3 = 3'b100 ;
always @  posede ( Clk  or  Rst )
begin
  if (Rst) begin
    state = S1 ;
    Out1 =  1'b0 ;
  end
else
 case (State)
     // synopsys  full_case  parallel_case
   S1:   . . .
   S2:   . . .
   S3:   . . .
   default : Out1 = 1'bx ;
  endcase
end
endmodule
```

Figure 15. VHDL /Verilog Examples for OHE

Another advantage of OHE (encoding) is that it allows designers to minimize the logic between the current-state register and the logic being controlled. A sequentially encoded state-machine usually requires gating the register output along with other state inputs or control signals.

An optimal FPGA solution would be to use One Hot Encoding, **and** design the state-machine so that each state controls only one output, or only valid state per register. This allows the single register output to drive the logic being controlled at that state. This will minimize or possibly eliminate gating requirements.

This should be considered a design issue, not merely HDL coding style!

Point 8: Use LFSR for Terminal Count
Objective: Reduce gating requirements.
Benefit: Minimize logic levels, enhance performance, faster clock rate.

Counters are an integral aspect of digital circuitry. However, a strict binary sequence is not always required. Given the inherent fan-in limitation for combinatorial logic within an FPGA, large binary counters may necessitate multiple logic levels to fully decode the outputs. Once again, this increases the path delay, and reduces the maximum frequency.

Depending on the application, it may be more appropriate to use an **LSFR**, (Linear Feedback Shift Register), especially if generating a terminal count is the primary objective, as in the case of a FIFO.

An LSFR counter is distinguished by its use of flip-flops in a shift-register sequence along with taps from various stages of the registers. The taps are either XORed or XNORed. The placement of the taps determines the count sequence, which although is it pseudo-random (non-binary), it does repeat and is therefore deterministic.

The value of the LSFR in an FPGA is that it leverages the use of the dedicated registers while minimizes gating requirements. From a coding standpoint, using an LFSR requires that you place the taps correctly to get a particular count sequence, and that you safeguard against the counter initializes to an illegal condition and "locking-up". For example, a lock-up would occur if all the registers were reset, and a '0' was fed into the first stage. This possibility exist anytime the maximum count sequence is not used.

Another consideration is whether to use a "*one to many*" or "*many to one*" approach. As shown in figure 16, a "*one to many*" uses only a single xor (xnor) gate prior to the registers as opposed to a tree of gates as required by the "*many to one*" approach.

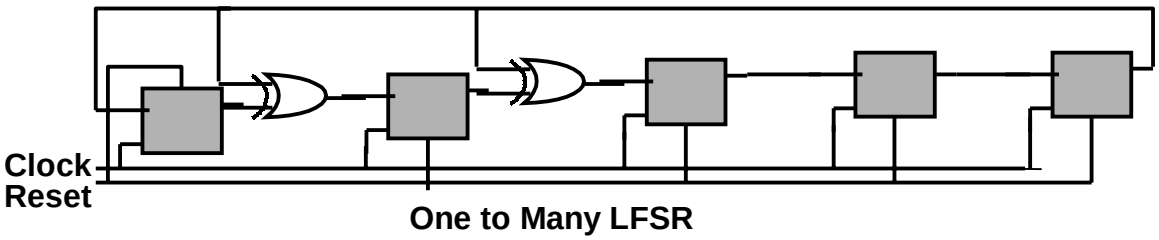


Figure 16. "One to Many" LFSR

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.Numeric_STD.all;

entity My_LFSR is
    port ( Clk, Rst: in std_logic ;
            Out_1 : out unsigned (3 downto 0)) ; --unsigned integer, defined in package Numeric.std
    end entity;

architecture RTL of My_LFSR is
  constant  TAPs : unsigned ( 3 downto 0) := “1100”; --Taps taken according to desired count sequence
begin
  process (Clk, Rst)
    variable LSFR_Int : unsigned (3 downto 0);  -- used to initialize LFSR
    variable Init_Zero, Feedback : std_logic;      --
  begin
    if (Rst = ‘1’) then
      LSFR_int := “0000”    -- reset counter
      elsif rising_edge (Clk) then
        Init_Zero := ‘0’;
          for I in 0 to 2 loop
            Init_Zero := Init_Zero nor LSFR_Int(I); – generate “nor” logic to allow all possible states
            end loop;
            Feedback := LSFR_Int(3) xor Init_Zero;
            for I in 3 downto 1 loop
              if (TAPs(I-1) = ‘1’) then
                LSFR_Int (I) := LFSR(I-1) xor Feedback;
                else
                LFSR_Int (I) := LFSR(I-1);
              end if;
          end loop;
        LFSR_Int(0) := Feedback;
     end if;
   Out_1 <= LFSR_Int;
 end process;
end architecture RTL;
```

Figure 17. VHDL Code for 4 bit “One to Many” LFSR (reaches all 16 states)

Point 9: Avoid Integer Data-type On Outputs & Use Little-Endian Notation.
Objective: Reduce synthesis conflicts.
Benefit: Consistent with back-end tools, enhance code portability.

In both *VHDL* and *Verilog*, output ports may be declared as bussed structures (vectors). The order of the bus is indicated when the signal or port declaration is made.

For VHDL, the following is a valid declaration for the output port '**Q**':

```
entity My_Cnt  is
  port  ( Clk, Rst : in std_logic;
           Q : out integer range 0 to 15  );
end entity My_Cnt;
```

Any of the following would also be valid:

```
Q : out  integer range 15 downto 0
Q : out  std_logic_vector ( 0 to 3 )
Q : out  bit_vector (0 to 3)
Q : out  std_logic_vector ( 3 downto 0 )
Q : out  bit_vector ( 3 downto 0 )
```

However, the use of integers on outputs is not recommended for RTL coding. If the range is not specified, the resulting bus will be a minimum word-wide length for the particular compiler, but not less than 32 bits as required by the IEEE 1076 standard.

Furthermore, *Std_logic_vector* and *bit_vector* should be assigned in descending order from left to right, this makes your code consistent with most back-end P&R tools. You may avoid some possible errors or conflicts by consistently adhering to this simple rule.

The last thing to note about VHDL is that any vector is an ***array*** of individual elements that have been grouped into a *composite* data type. The language **does not** contain built-in binary weighting, and thus no concept of LSB and MSB exist. Individual elements are referenced only by their left to right placement within the array. Package ***standard*** and ***std_logic_1164*** define *bit_vector* and *std_logic_vector* as unconstrained arrays of type *bit* and type *std_logic* respectively.

Verilog is more concise, a bus may be declared in the module's port declaration section as:

```
output [3:0] Q ;
or
output [0:3] Q ;
```

In either case, the MSB is always defined as the left boundary literal. Once again, for consistency, use the little-endian notation, i.e. ***output*** **[3:0]** Q;.

Point 10: Controlling Hierarchy
Objective: Enhance Synthesis, Place & Route.
Benefit: Better FPGA chip-level implementation.

Hierarchy is created in HDL by instantiating lower level modules into higher level ones.

In *VHDL*, the procedure is somewhat formal in that the lower-level *entity* must first be declared as a *component*, and then instantiated, along with a *port map* statement to indicate how the ports and signals will connect at the higher level.

In *Verilog*, the component declaration and instantiation are combined.

In either case, the lower level component must exist before it can be referenced at the higher level. As with schematic capture, using hierarchy lends structure and enhances the functional understanding of the design. When creating a hierarchical block for an FPGA target device, there are 3 very important guidelines to remember:

1. Use Registers As Natural Boundaries.

This is consistent with the FPGA architecture and the general concept of RTL. *Register Transfer Level* coding is generally defined as what logic (combinatorial) transformations are necessary between clock edges (sequential).

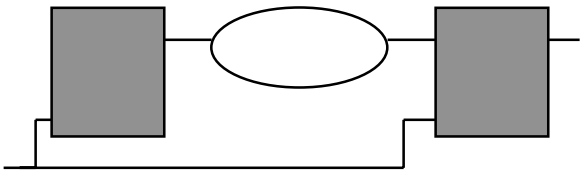


Figure 18. RTL Coding Model

2. Minimize Clocks Per Hierarchical Block.

Ideally, use only one clock per module. Few things are more difficult for back-end place & route software than optimizing logic with multiple clocks. Remember that most path based timing constraints are referenced to a given clock. The mapping, placement and routing will be guided by those constraints.

When multiple clocks are present in the same logic block, the constraints are not nearly as effective, and the implementation may not be optimal for any of the clocks or their associated logic.

3. Keep Critical Paths to a Single Block.

Keeping critical paths to a single block helps when optimizing all the logic associated with that signal. On the other hand, when signals cross module boundaries, it is much more difficult to effectively constrain logic across the total path.

When coding, try to reference the critical signal(s) within processes of a common architecture.

Point 11: Instantiating Key Components
Objective: Control Synthesis, Enhance Place & Route.
Benefit: Maximal Chip-Level Optimization.

To **instantiate** means to create an "instance of" a given component. As mentioned earlier, design hierarchy is created in VHDL and Verilog by instantiating lower level modules into higher level ones.

To describe any given logic block, you may elect to "*infer*" the functionality, or "*instantiate*" one or more lower level components. Occasionally the need for device-level optimization will drive the choice between the two. This degree of optimization may by necessary to satisfy the overall circuit performance objectives.

When this is the case, the instantiated component may in fact be part of the macro library for the target technology. The need for this sort of "direct instantiation" arises when the particular synthesis compiler can not infer the same chip-level optimal implementation from the generic code!

This situation is common to FPGAs since they often contain unique, dedicated resources that enhance on-chip performance. In general, synthesis tools create the intended functionality, but may not properly utilize dedicated and or technology specific resources.

As shown in figure 19, *inference* is accomplished through the use of standard language operators and expressions. On the other hand, *instantiating* a macro from the target technology requires the same syntax as building any other hierarchical logic, specifically, the component declaration, the instance and the port mapping designation.

The inherent drawback to library macro instantiation is that it limits the portability of the code. This however may be necessary to achieve the greater performance objective.

Furthermore, the negative effect on code portability may be minimized somewhat by using separate *architectures* or *modules* for all technology specific references.

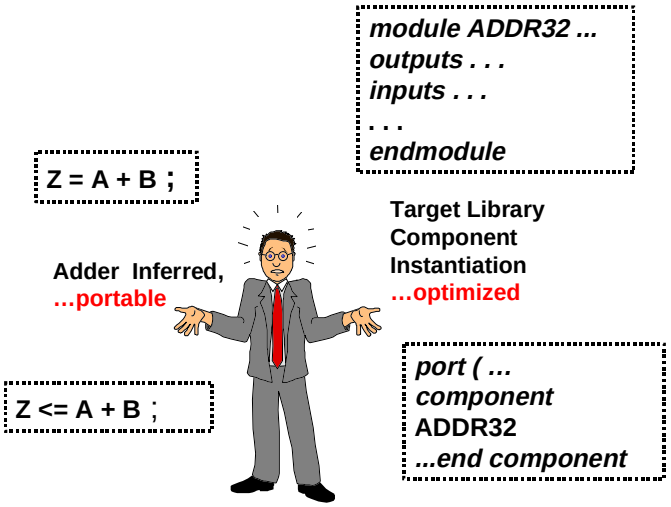


Figure 19. Inference Vs. Instantiation

As an example of using library macro instantiation to gain the utmost device-level optimization, consider the DynaChip DY6000 product family and its high speed ***carry logic.*** This is dedicated routing from each logic block to the next in a given column. The carry-logic enhances the performance of Adders and Comparators.

Because synthesis tools are constantly evolving, you should consult your specific compiler's documentation to determine if they currently support mapping to this valuable resource, and if so, under what circumstance? Some synthesis tools would make such mapping decisions due in part to user supplied performance constraints.

The next section contains examples of declaring and instantiating key components from the DynaChip DY6000 macro library. The ADDRx & ICOMPx macros use the dedicated ***carry logic***, thus enhancing the overall design performance.

In both VHDL and Verilog, any instantiated component must be visible to the compiler. Depending on the tool interface, project environment and particularly how vendor libraries are attached, it may be necessary to compile (analyze) the library so that its contents are part of the defined work library.

In most cases, however, a reference to the library logical name and particular package is all that is required. For example in VHDL:

library DynaChip;
use DynaChip.DY6000_Components.all;

This is meant only as an example of the “use” clause, consult your synthesis tool documentation to determine the exact logical name of the particular DynaChip library.

In addition, it will be necessary to properly and accurately state the ports in the component declaration. The order is not important, and VHDL is not case sensitive, but Verilog is! In either case, if the port name and mode (direction) does not match what the compiler finds, it will complain that the particular port is “not bound”, or “no binding” exist. Either message indicates that it did not find the port with the exact name and mode that you specified in the declaration.

Because order is not relevant, always use *named association* for port mapping when instantiating vendor library components, most tools require it, and it’s good coding practice under any circumstance! Figure 20 shows the ports and names for the ADDRx and ICOMPx macros.

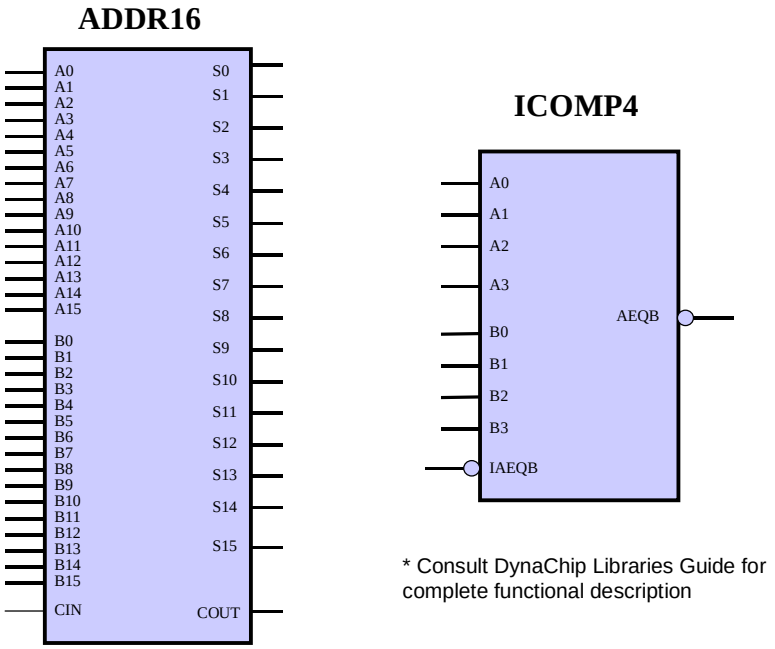


Figure 20. Port names for ADDRx & ICOMPx library macros.

(Note that future releases of the DynaChip library will employ bussed notation, this will ease the coding requirements when referencing inputs and outputs)

Figure 21 shows VHDL and Verilog examples of instantiating the ADDR8 macro from the DY6000 library.

-- VHDL instantiation using ADDR8 macro

```
library IEEE;
use IEEE.std_logic_1164.all;
library DynaChip;
use DynaChip.DY6000_Components.all; --example only, consult synthesis tool documentation

entity ADD8 is
  ports (A, B : in std_logic_vector (7 downto 0 );
          C_in : in std_logic;
          C_out: out std_logic;
          Sum : out std_logic_vector (7 downto 0 );
end entity ADD8;

architecture DYNA_ADDR of ADD8 is

   component ADDR8
       port (A7,A6,A5,A4,A3,A2,A1,A0 : in  std_logic;
             B7,B6,B5,B4,B3,B2,B1,B0  : in  std_logic;
             S7,S6,S5,S4,S3,S2,S1,S0      : out  std_logic;
             CIN                          : in  std_logic;
             COUT                         : out  std_logic
             );
    end component;

   signal  Ain, Bin, Sout : std_logic_vector (7 downto 0) ;
   signal  Carry_In, Carry_Out : std_logic ;

begin
U1: ADDR8 port map (
A7=>Ain(7), A6=>Ain(6), A5=>Ain(5), A4=>Ain(4), A3=>Ain(3), A2=>Ain(2), A1=>Ain(1), A0=>Ain(0),
B7=>Bin(7), B6=>Bin(6), B5=>Bin(5), B4=>Bin(4), B3=>Bin(3), B2=>Bin(2), B1=>Bin(1), B0=>Bin(0),
S7=>Sout(7), S6=>Sout(6), S5=>Sout(5), S4=>Sout(4), S3=>Sout(3), S2=>Sout(2), S1=>Sout(1), S0=>Sout(0),
CIN=>Carry_In, COUT=>Carry_Out
);
end architecture DYNA_ADDR ;
```

// Verilog instantiation using ADDR8 macro

```
uselib  DynaChip.DY6000_Components ;  //example only, consult synthesis tool documentation

module ADD8  ( A, B, C_in, C_out, Sum ) ;
input   [7:0] A, B ;
output  [7:0] Sum ;
input   C_in ;
output  C_out ;

 wire   [7:0] Ain, Bin, Sout ;
 wire   Carry_In, Carry_Out ;

ADDR8 : U1 (
A7.(Ain[7]), A6.(Ain[6]), A5.(Ain[5]), A4.(Ain[4]), A3.(Ain[3]), A2.(Ain[2]), A1.(Ain[1]), A0.(Ain[0]),
B7.(Bin[7]), B6.(Bin[6]), B5.(Bin[5]), B4.(Bin[4]), B3.(Bin[3]), B2.(Bin[2]), B1.(Bin[1]), B0.(Bin[0]),
S7.(Sout[7]), S6.(Sout[6]), S5.(Sout[5]), S4.(Sout[4]), S3.(Sout[3]), S2.(Sout[2]), S1.(Sout[1]), S0.(Sout[0]),
CIN.(Carry_In), COUT.(Carry_Out)
);

endmodule
```

Summary

Optimal HDL coding for FPGAs requires more than generic operators and expressions. The choice of design and user coding style, synthesis compiler and target technology all affect the end result.

The best approach is to understand and carefully consider each stage, and its contribution toward the end objective –maximizing performance within the target technology.

The goal of a high level, pure and technology independent HDL design may be attainable at some point in the future. Today, however, when targeting FPGAs, you can avoid unexpected problems and gain considerably better results by keeping the device-level implementation in mind.