Creating the Most Efficient Comparators

by Roberta Fulton, Technical Marketing Engineer, Xilinx, roberta.fulton@xilinx.com

Comparators are best modeled with word-wise compares within a **PROCESS** or an **ALWAYS** block that contains the **IF** statement and an **ELSE** clause, and no **ELSE-IF** clauses. Conditional signal assignments in VHDL or conditional continuous assignments in Verilog could be used, but at a high cost in simulation time. Without the sensitivity list in VHDL or the event list in Verilog the simulators would constantly be checking the statements even when the inputs are unchanging, thus slowing simulation time considerably. If compared bit-wise some synthesizers may not see optimizations available to them such as the use of H-MAPs.

Three alternative representations to infer an 8-bit equality comparator are shown below. The first, COMPARATOR_A does a bit by bit compare (**Figure 1**), the second COMPARATOR_B sets the default first, then compares (**Figure 2**), so does not have an **ELSE** clause, the third has a complete **IF-THEN-ELSE** statement (**Figure 3**).

Comparator A - Bit by Bit Compare

```
library IEEE;
use IEEE.STD_LOGIC_1164.all,
 IEEE.NUMERIC_STD.all;
entity COMPARATOR_A is
 port (AIN1, AIN2: in unsigned(7 downto 0);
     AEQ: out std_logic);
end entity COMPARATOR_A;
architecture RTL of COMPARATOR_A is
begin
EQUALITY:process (AIN1, AIN2)
begin
— Compare each bit in turn in a “for” loopp
AEQ <= ‘1’;
for I in 0 to 7 loop
 if(AIN1(I) /= AIN2(I)) then
  AEQ <= ‘0’;
  exit;
 else
  null;
 end if;
end loop;
end process EQUALITY;
end architecture RTL;

module COMPARATOR_A
(AIN1, AIN2, AEQ);
input [7:0] AIN1, AIN2;
output AEQ;

integer I;
reg AEQ;

//Compare each bit in turn in a “for” loop
always@(AIN1 or AIN2)
begin: EQUALITY
AEQ = 1;
for (I=0; I<7; I=I+1)
if (AIN1[I] != AIN2[I])
AEQ=0;
else
;
end
endmodule
```

Figure 1

Comparator B – No Else Clause

```
library IEEE;
use IEEE.STD_LOGIC_1164.all,
 IEEE.NUMERIC_STD.all;
entity COMPARATOR_B is
 port (BIN1, BIN2: in unsigned(7 downto 0);
     BEQ: out std_logic);
end entity COMPARATOR_B;
architecture RTL of COMPARATOR_B is
begin
EQUALITY:process (BIN1, BIN2)
begin
— No “else” is required since default is
 defined before the “if””
BEQ <=’0’;
if (BIN1 = BIN2) then
BEQ <= ‘1’;
end if;
end process EQUALITY;
end architecture RTL;

module COMPARATOR_B
(BIN1, BIN2, BEQ);
input [7:0] BIN1, BIN2;
output BEQ;

integer I;
reg BEQ;

//No “else” is required since default is
 defined before the “if”
always@(BIN1 or BIN2)
begin: EQUALITY
BEQ = 0;
if (BIN1 == BIN2)
BEQ=1;
end
endmodule
```

Figure 2

Complete IF-Then-Else Clause

```
library IEEE;
use IEEE.STD_LOGIC_1164.all,
 IEEE.NUMERIC_STD.all;
entity COMPARATOR_C is
 port (CIN1, CIN2: in unsigned(7 downto 0);
    CEQ: out std_logic);
end entity COMPARATOR_C;
architecture RTL of COMPARATOR_C is
begin
EQUALITY:process (CIN1, CIN2)
begin
— Easiest to read versionn
if (CIN1 = CIN2) then
CEQ <= '1';
else
CEQ <= '0';
end if;
end process EQUALITY;
end architecture RTL;
```

```
module COMPARATOR_C
(CIN1, CIN2, CEQ);
input [7:0] CIN1, CIN2;
output CEQ;

integer I;
reg CEQ;

// Easiest to read version
always@(CIN1 or CIN2)
begin: EQUALITY
if (CIN1 == CIN2)
CEQ=1;
else
CEQ=0;
end
endmodule
```

Figure 3

Reviewing RTL-level schematics or design browsers in the synthesizers we can see how the synthesizer sees the code immediately upon parsing it before any optimization or technology mapping.

Comparing the RTL level schematics of COMPARATOR_A (**Figure 4**) and COMPARATOR_B (**Figure 5**) we see that in both cases the EQUALITY operator is assigned to a generated module, a SELECT type in COMPARATOR_A and EQUALITY type in COMPARATOR_B. But we see that extra logic is inferred for comparator A to do the bit-by-bit comparison looping. COMPARATOR_C's RTL schematic is similar to COMPARATOR_B's so is not shown.

As the RTL schematics show, the initial logic for COMPARATOR_A and COMPARATOR_B is very different. Since the overall functions are equivalent we would normally assume that the optimization step would reduce them to the same logic.

But as seen in the gate-level schematics of **Figures 6** and **7**, COMPARATOR_B's word-wise compare and its more condensed optimization seed netlist allowed the synthesizer to "see" the opportunity to use two HMAPs instead of a fifth FMAP. This meant that 2 CLBs are required instead of 3 and that one level of logic is required instead of two. (An FMAP-HMAP combination is considered 1-level since no external CLB routing is required). Performance will be slightly enhanced because CLB internal routing is usually lower than between CLBs. Once again COMPARATOR_C's gate-level schematic is similar to COMPARATOR_B and is not shown. It's best to use COMPARATOR_C's code because of its readability.

When using inequality operators like ">" or "<" look for the better synthesizers to use carry logic (CYx cells) when mapping the generated modules to Xilinx technolgies. This is synthesizer dependent, if you do not see carry logic in your

COMPARATOR_A.RTL schematic

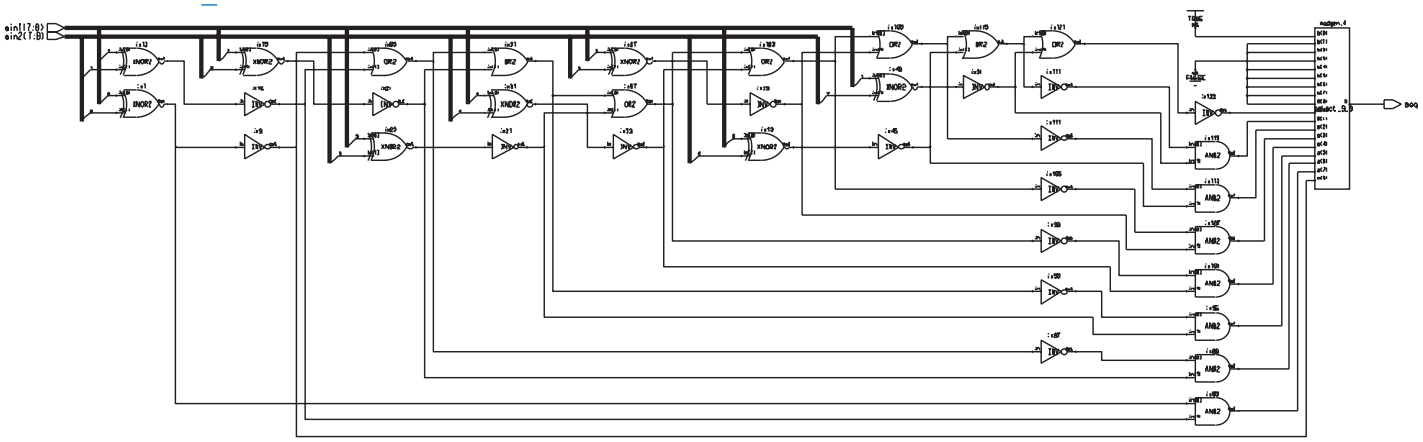

Figure 4

COMPARATOR_B
RTL Schematic

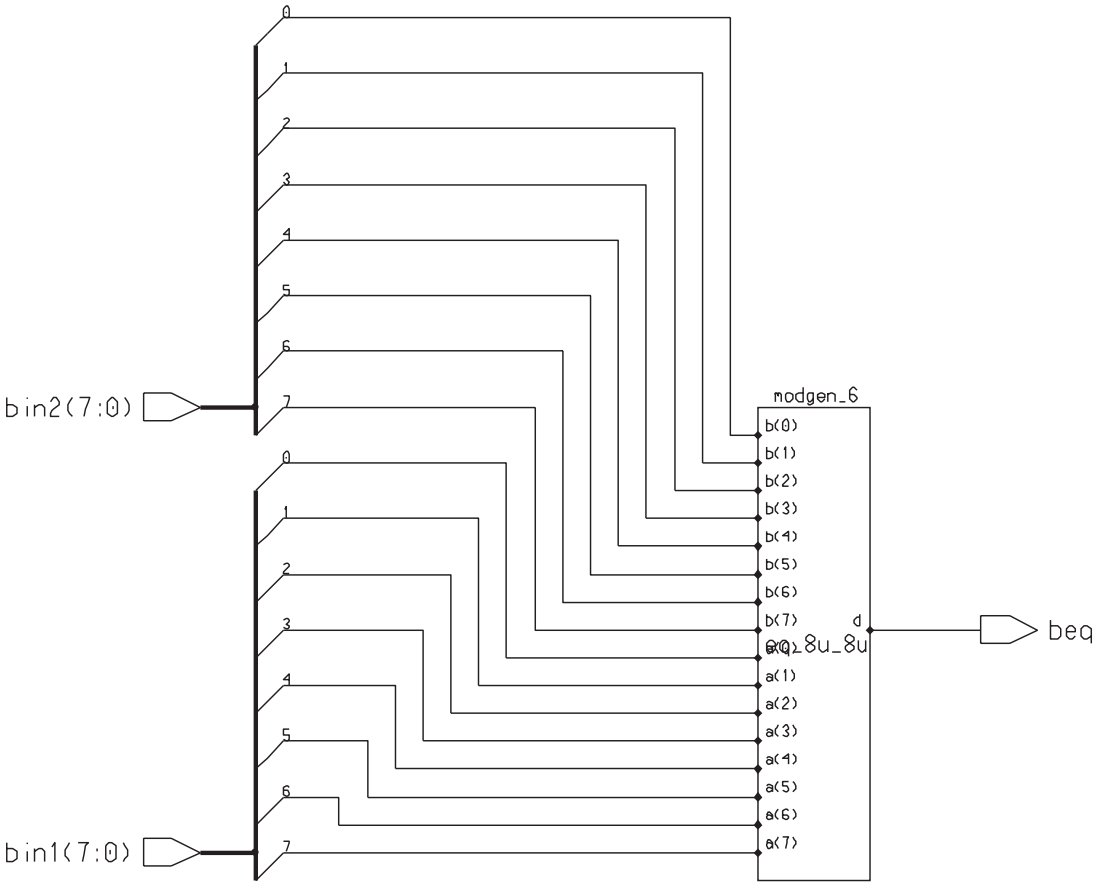

Figure 5

COMRATOR_A
Gate-level
Schematic

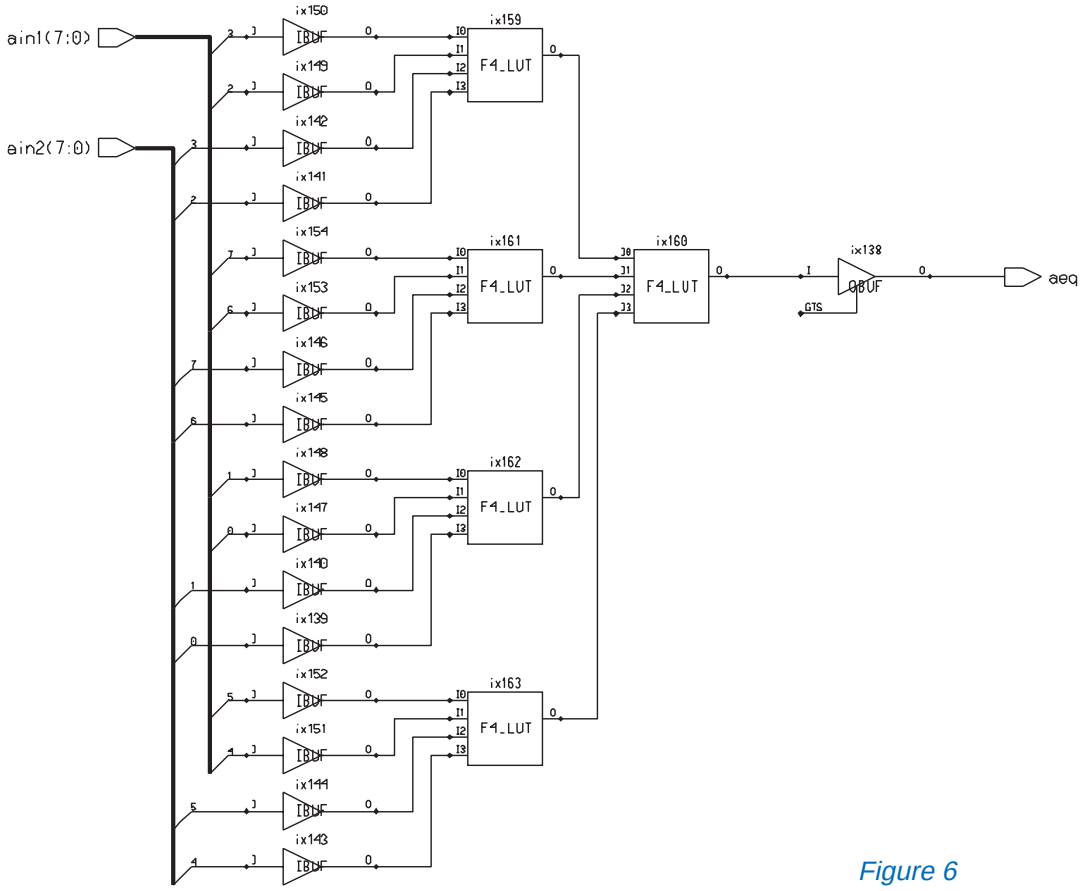

Figure 6

COMPARATOR_B Gate-level Schematic

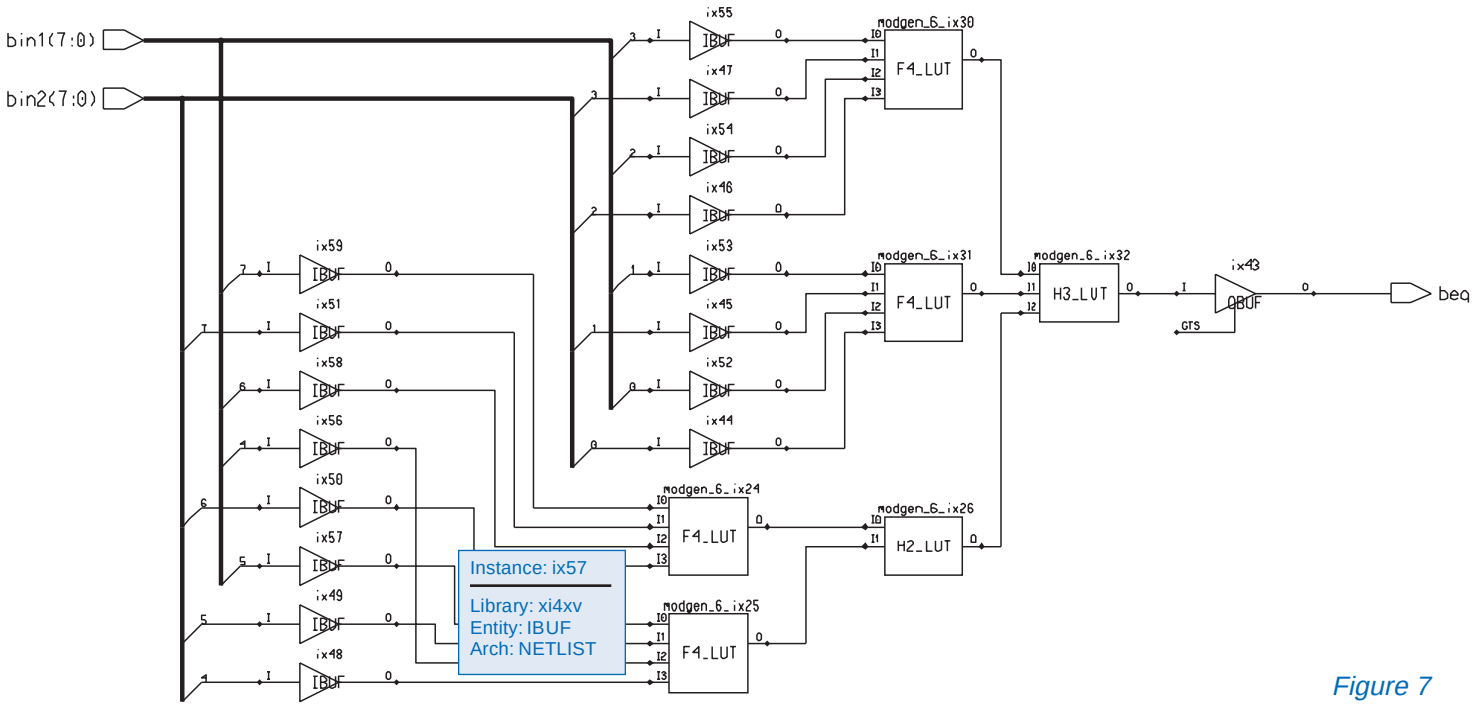

Figure 7

```
module COMPARATOR_D
(DIN1, DIN2, DGT, DLT, DEQ);
input [7:0] DIN1, DIN2;
output DGT, DLT, DEQ;

reg DGT, DLT, DEQ;

always@(DIN1 or DIN2)
begin: EQUALITY
DGT=0;
DLT=0;
DEQ=0;
if (DIN1 > DIN2)
  DGT=1;
 else
  if (DIN1 < DIN2)
  DLT=1;
  Else
// The “if” and equality compare is unnecessary
// only DEQ=1; is required after the “else”
  if (DIN1 == DIN2)
    DEQ=1;
end
endmodule
```

Figure 8

mapped operators ask your synthesis company about possible coding style dependencies or a future enhancement.

To prevent extra uneccessary logic watch for redundant compares when coding complex code sections. Some synthesizers will optimize out the redundancy, others will not.

Compares to a constant are usually smaller and faster than comparing two signals. **Figure 8** shows an example where the equality compare is redundant.

Conclusion

For best results, code your compares word-wise rather than bit-wise, check to see that your synthesizer uses carry logic for inequality operators (asking for help if it doesn't), remove all redundant compare operations, and compare to a constant rather than a signal when possible.

Code your compares word-wise rather than bit-wise, check to see that your synthesizer uses carry logic for inequality operators, remove all redundant compare operations, and compare to a constant rather than a signal when possible.