

Homework 1

Parameterized Pipelined Sorter

Eric Liskay

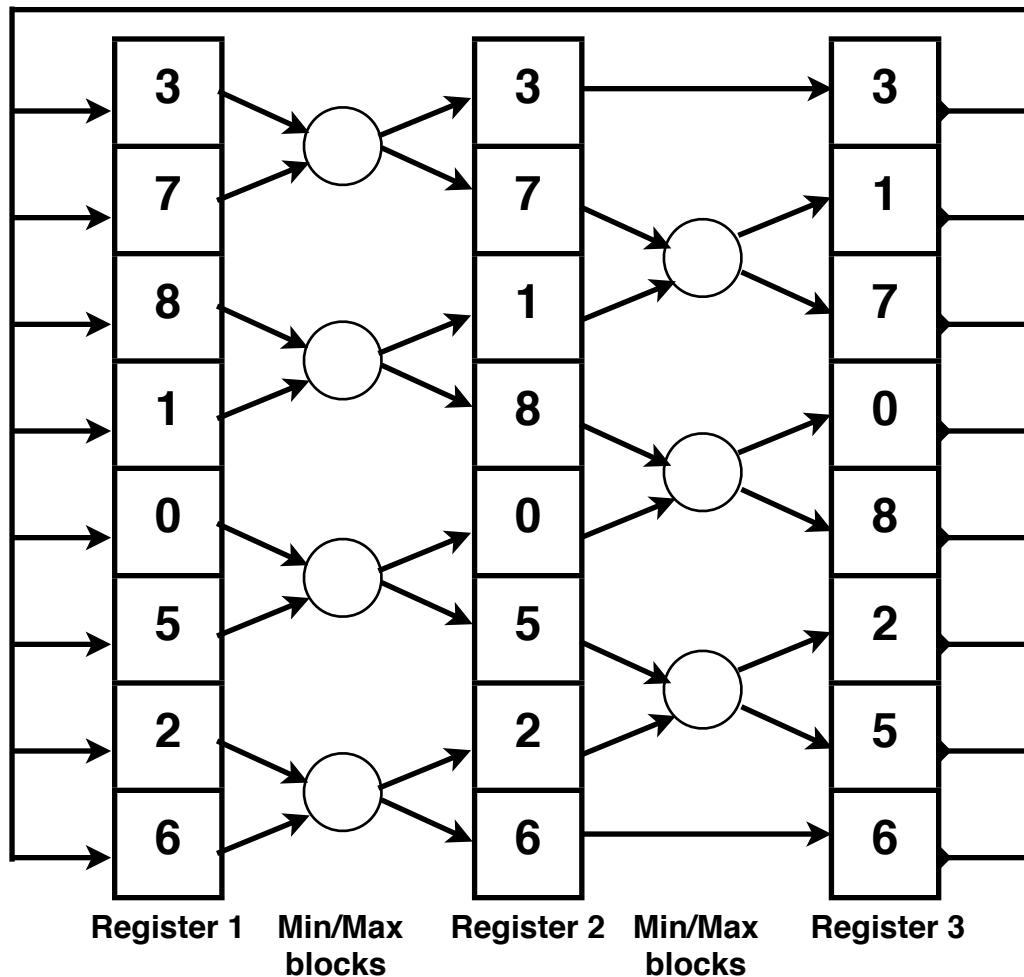
ECE 590
Spring 2012

Objective

The objective of this homework was to design and implement in VHDL a pipelined sorter using the butterfly network method. This sorter will be able to accept any number of N-bit numbers.

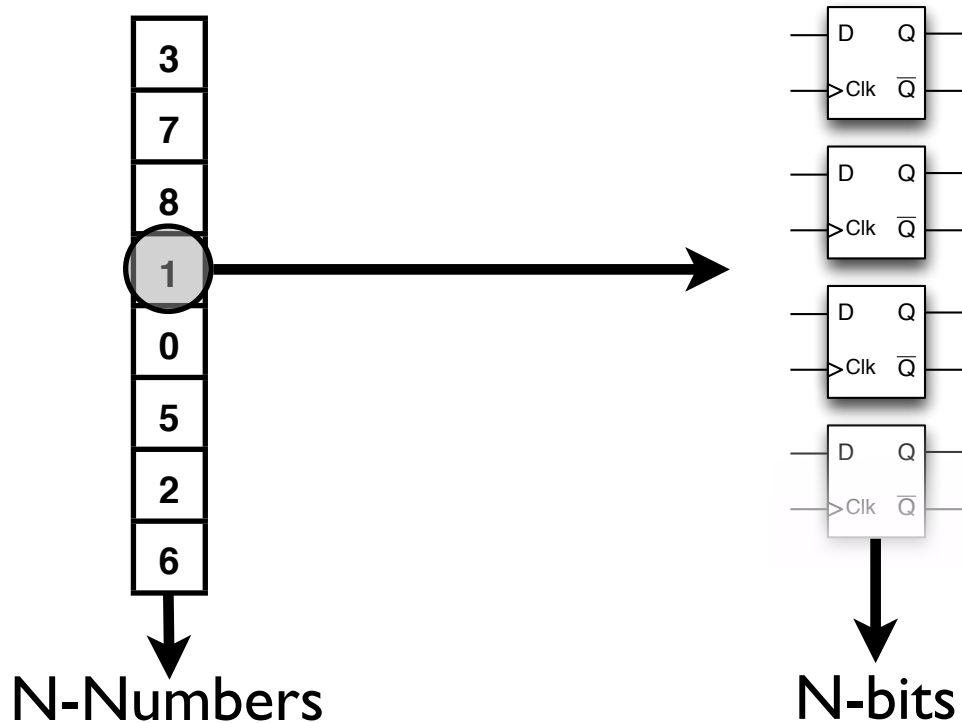
Design

Top-Level Design



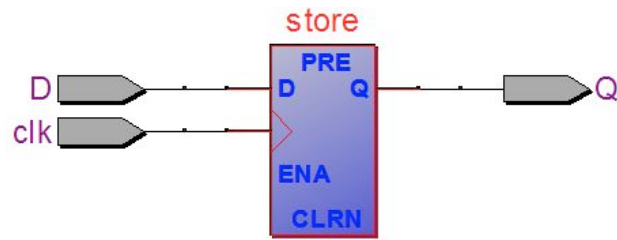
The above figure shows the top level of the design. It consists of three registers connected by Min/Max Blocks. The output of Register 3 is connected to the input of Register 1. Input into the sorter is inputted into Register 1. Output is taken from Register 3. Since there are three registers, three different sets of numbers to be sorted can be in flight at the same time. This is in essence a form of pipelining. Individual blocks of the design will be detailed in the following pages. The butterfly network architecture that this design is based on allows sorting to occur in two stages. The first stage features an even number of Min/Max blocks while the second features an odd number. The output from Register 3 loops back around to Register 1, allowing for alternating odd and even stages in the pipeline.

Registers

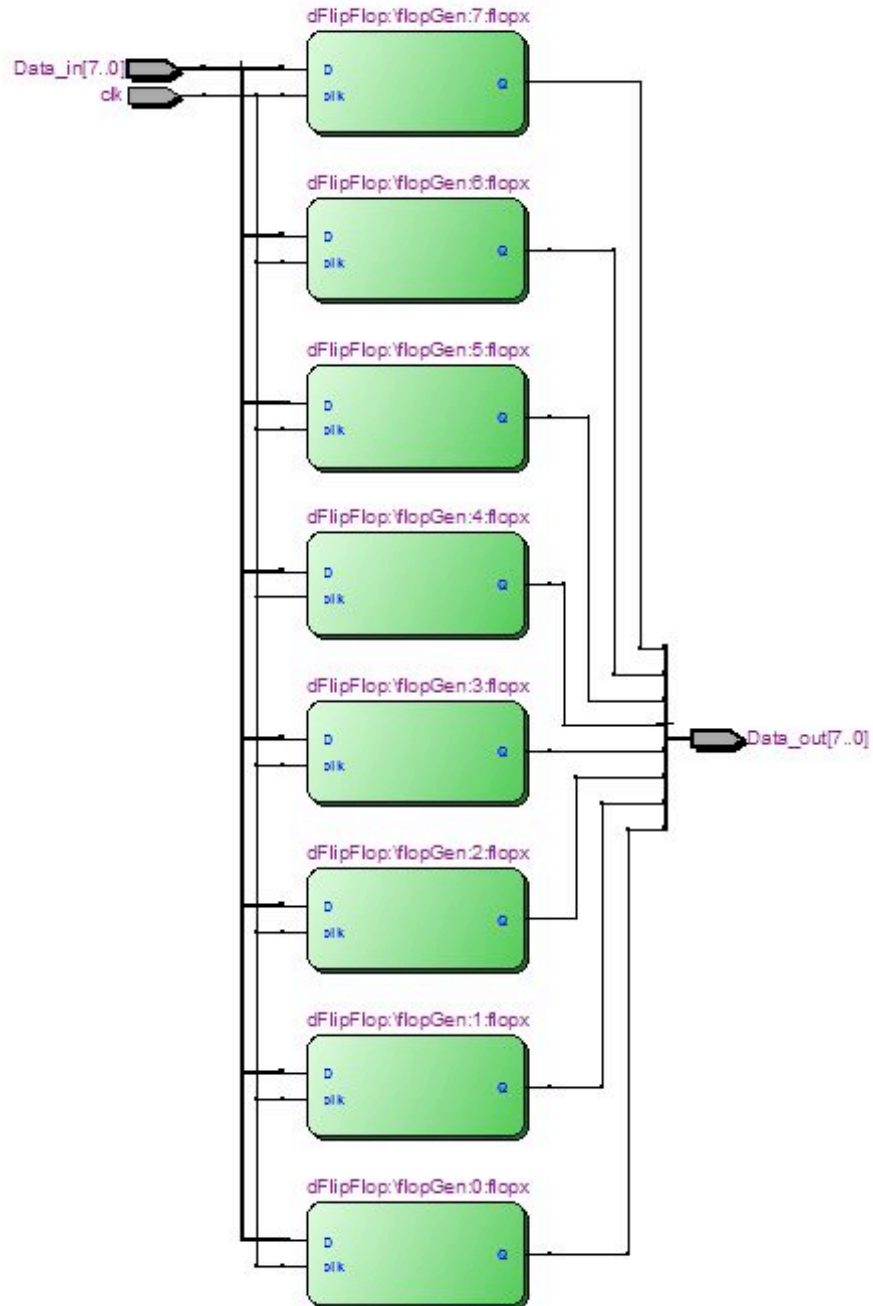


The figure above shows the implementation of the registers in the design. Each register consists of a certain number of numbers. Each of these numbers is modeled using a certain number of D flip-flops. The number of numbers to be sorted and the number of D flip-flops per number (corresponding to the number of bits) can be changed by the user via parameters. Generate blocks are used to generate the desired number of N-bit registers in each register and the number of flip flops in each register.

On the following page, figures with both the N-bit register and the individual flip-flops can be seen. These images were taken from Quartus II's RTL Netlist Viewer after synthesis.

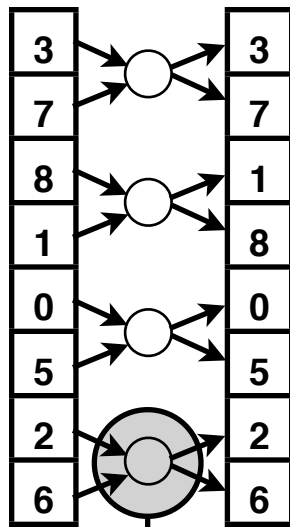


A D-Flip Flop from Quartus II's RTL Netlist Viewer



A Register from Quartus II's RTL Netlist Viewer

Min/Max Blocks

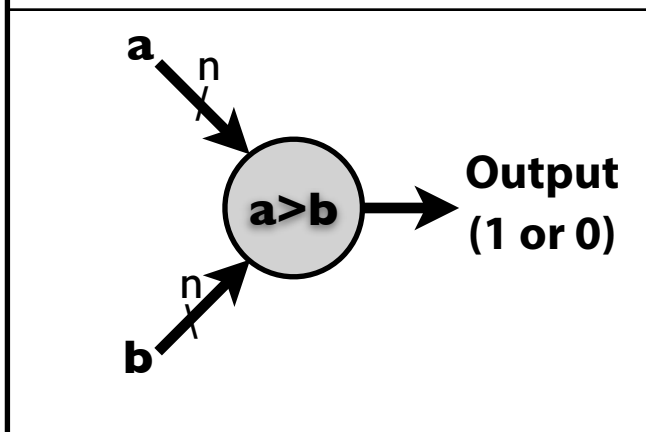
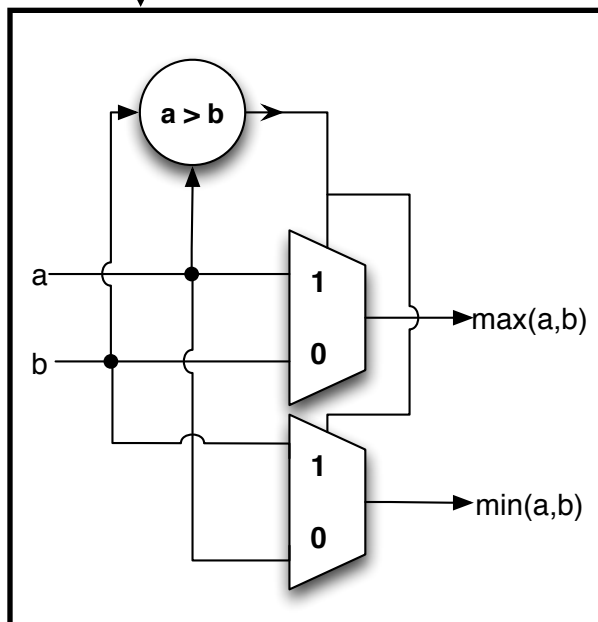
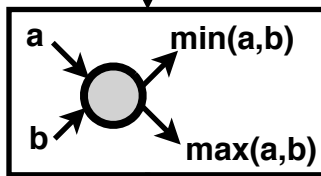


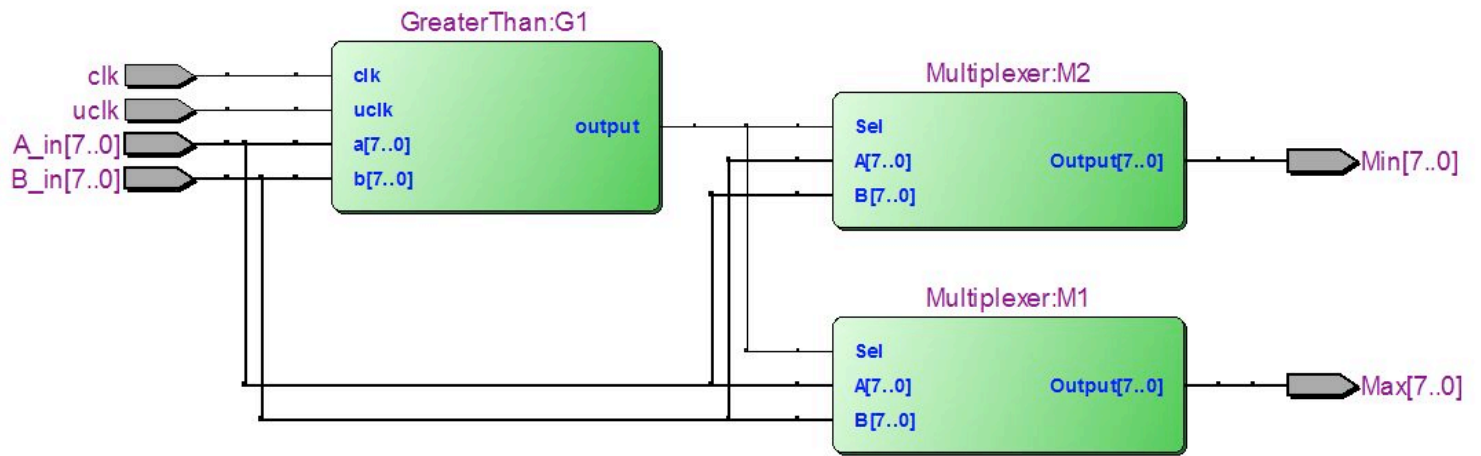
The Min/Max blocks are shown to the left in increasing detail. They take two inputs from a register, two N-bit numbers, compare them, and output them to the following registers. The value in the upper of the two registers to the output will be assigned the minimum of the two numbers and the lower register will contain the maximum.

The figure in the lower left shows the internal components of a Min/Max block. It consists of a greater-than block and two two-input multiplexers. The greater than block takes two numbers (two N-bit inputs) and outputs whether input A is greater than input B. The output consists of one bit, '1' if A is greater than B, and '0' if A is less than or equal to B.

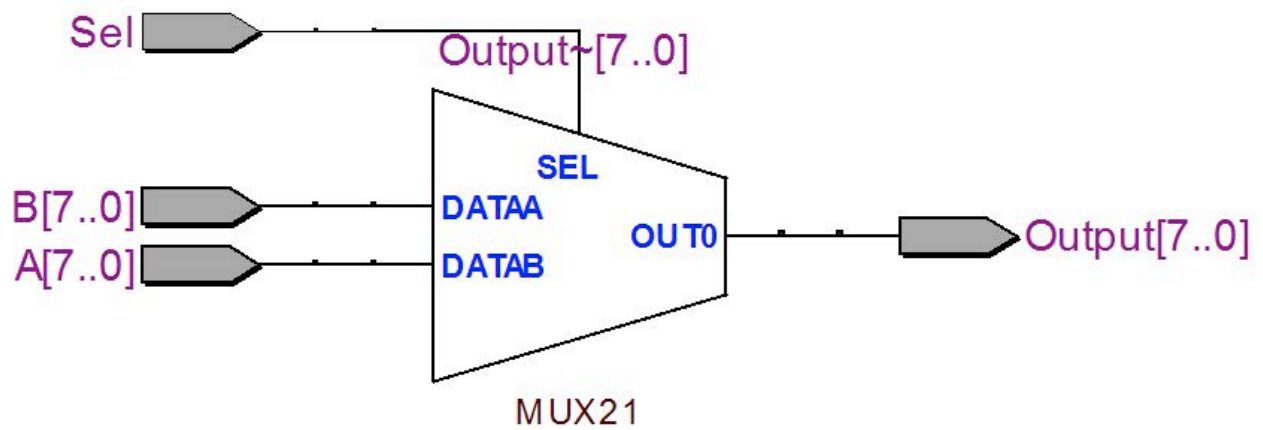
The output is connected to the select port of each multiplexer. The multiplexers are connected in such a way that they output the maximum and minimum number respectively based on the value of the select bit.

The following page shows a Min/Max block and a multiplexer taken from Quartus II's RTL Netlist Viewer post-synthesis. The page after that shows a gate level diagram of the 2-input multiplexer taken from Quartus II's Technology Map Netlist Viewer (Post-Fitted).

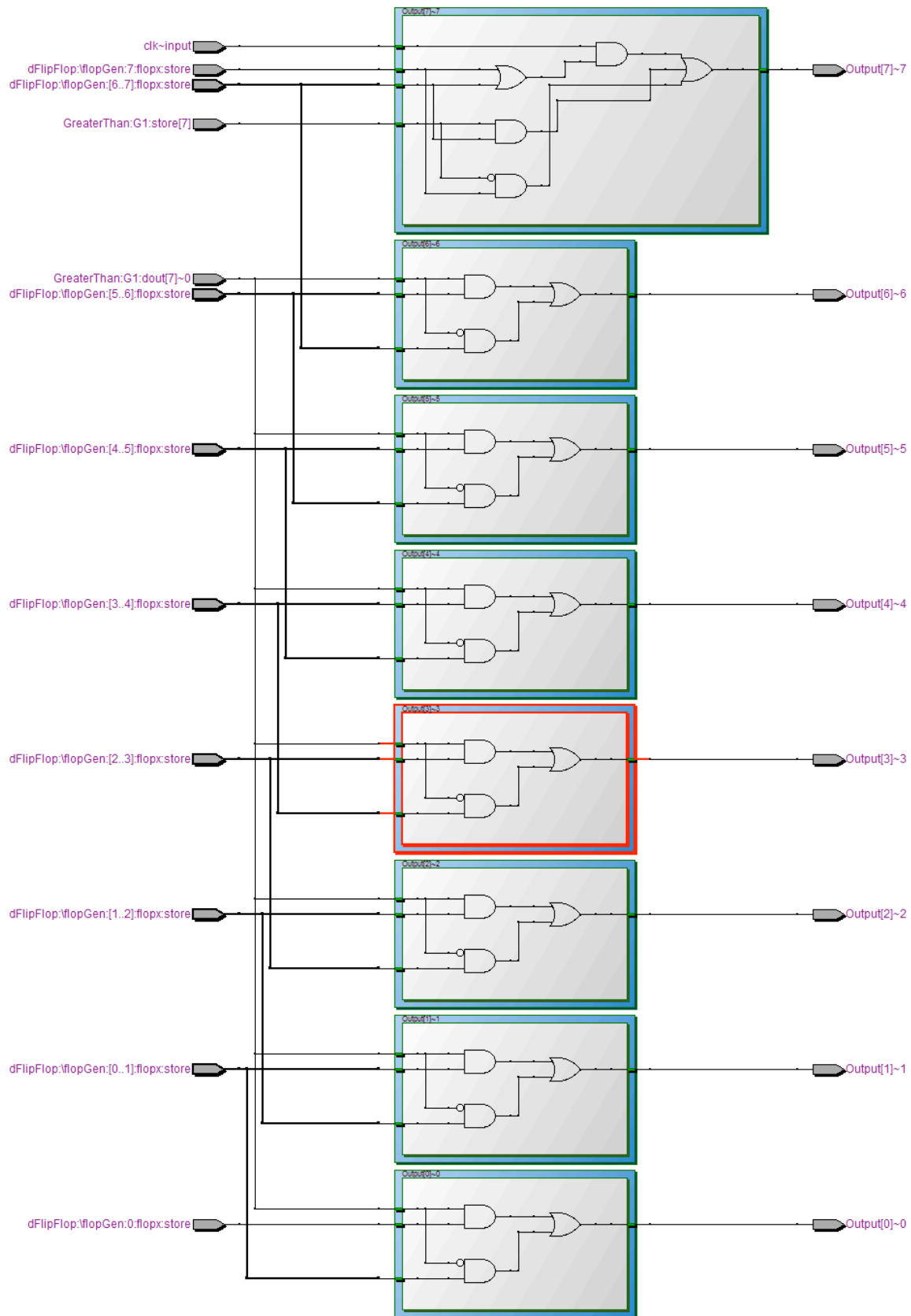




A Min/Max block from Quartus II's RTL Netlist Viewer

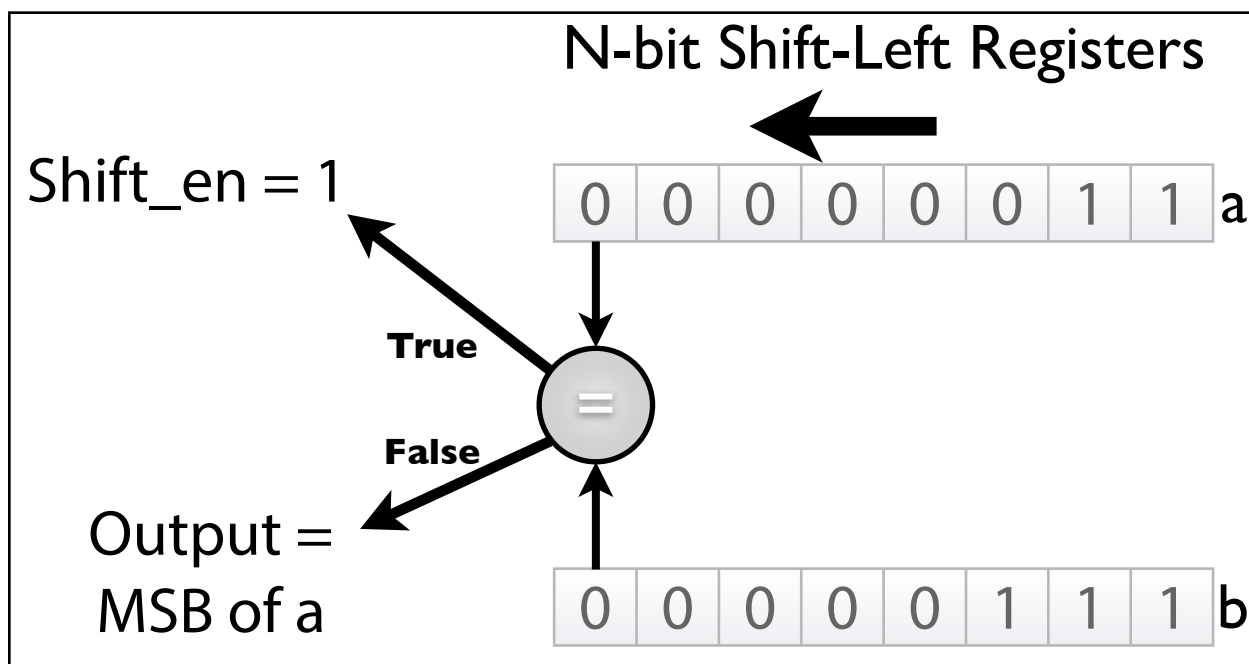


A 2-input Multiplexer from Quartus II's RTL Netlist Viewer



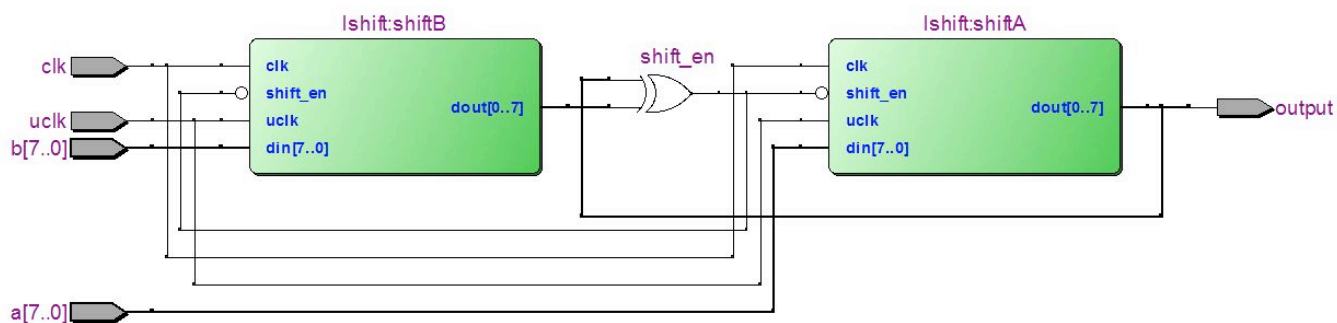
A 2-Input Multiplexer from Quartus II's Technology Mapped Netlist Viewer

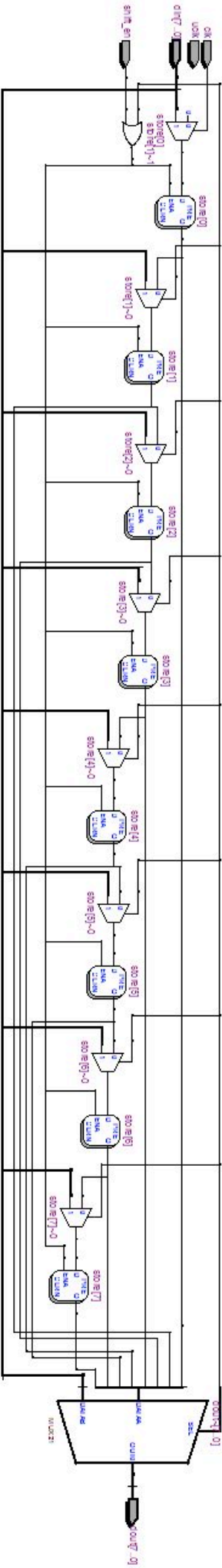
Greater-Than Block



The internal workings of the greater than block are shown in the above figure. The block consists of two N-bit shift-left registers and a comparator. One value is loaded into each of the shift registers. The comparator compares the most significant bits of each shift register. While these values are equal, the shift enable control signal is set to true, causing the shift registers to shift left on each clock. When the comparator finds that the two MSBs are not equal, it outputs the current most significant bit from shift register A. A '1' here means that the number in register A was greater than the number in register B, hence following the function of a greater-than operator.

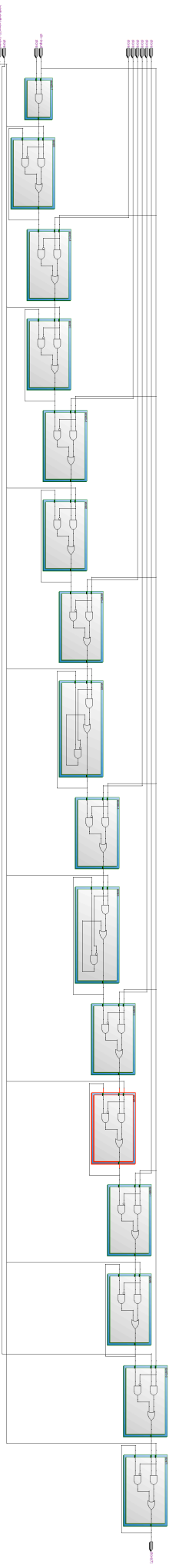
Below you can see a greater-than block from the model that was taken from Quartus II's RTL Netlist Viewer after synthesis. The XOR gate functions as the comparator. The following page shows an RTL level and gate-level schematic of the left-shift registers. In this schematic I show an 8-bit register, however this is based on the number bit-width which is modified using parameters.





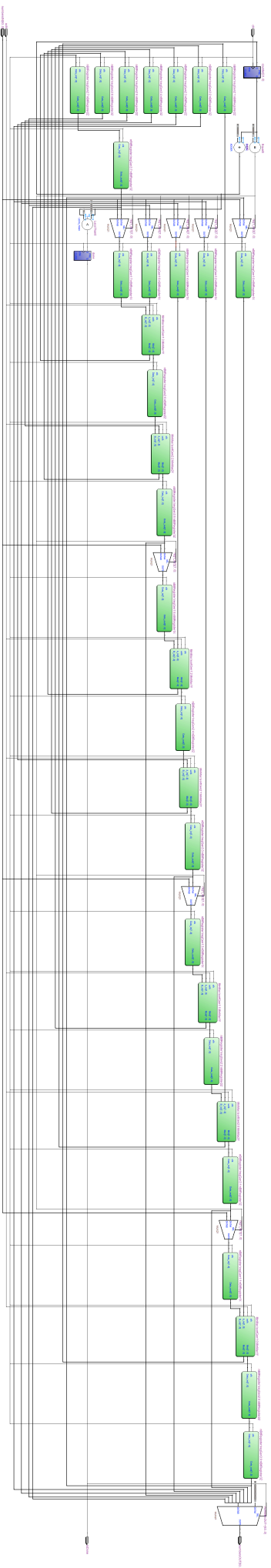
An 8-bit Shift Register from Quartus II's RTL Netlist Viewer

Than block from
RTL Netlist Viewer

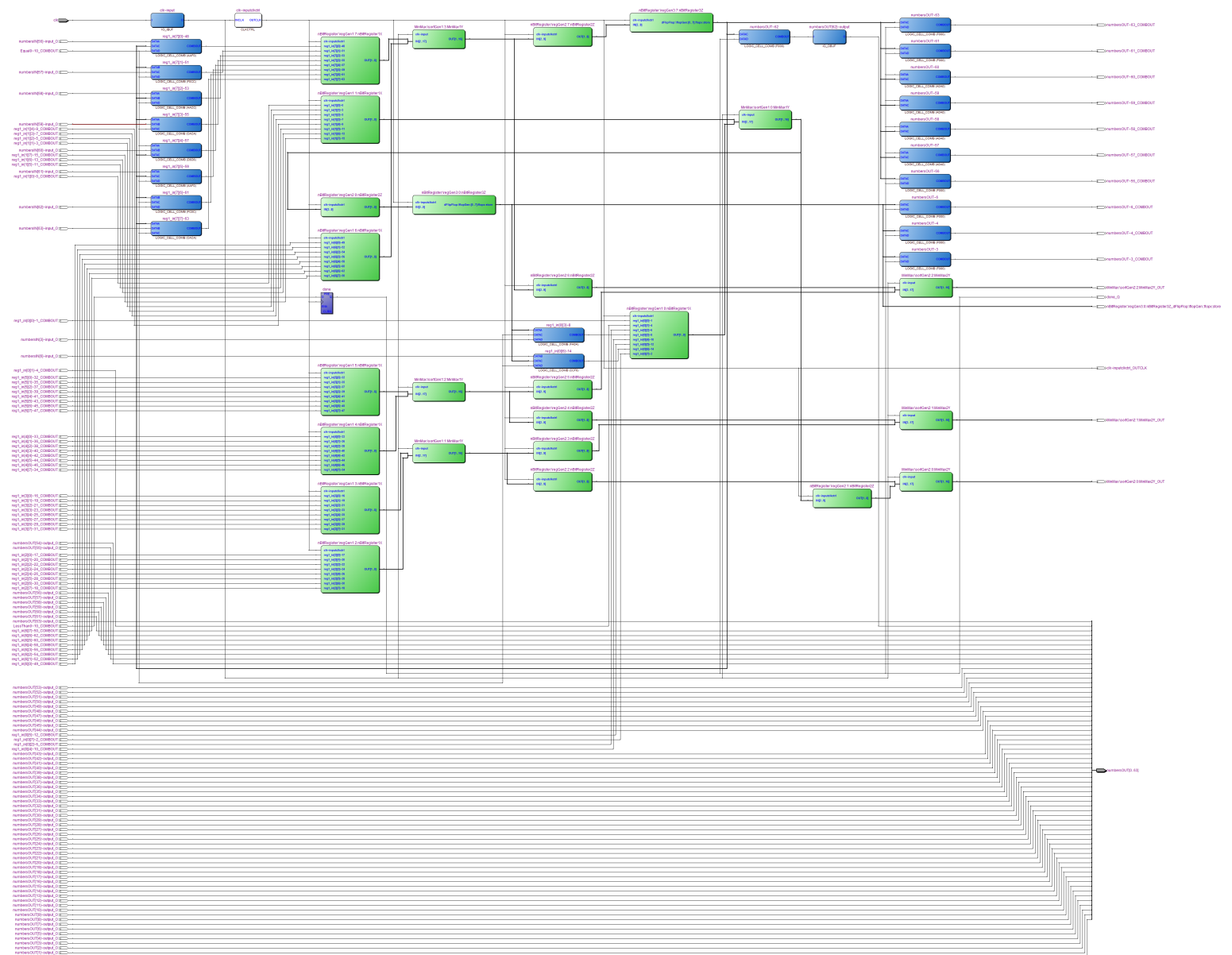


An 8-bit Shift Register from Quartus II's Technology Mapped Netlist Viewer

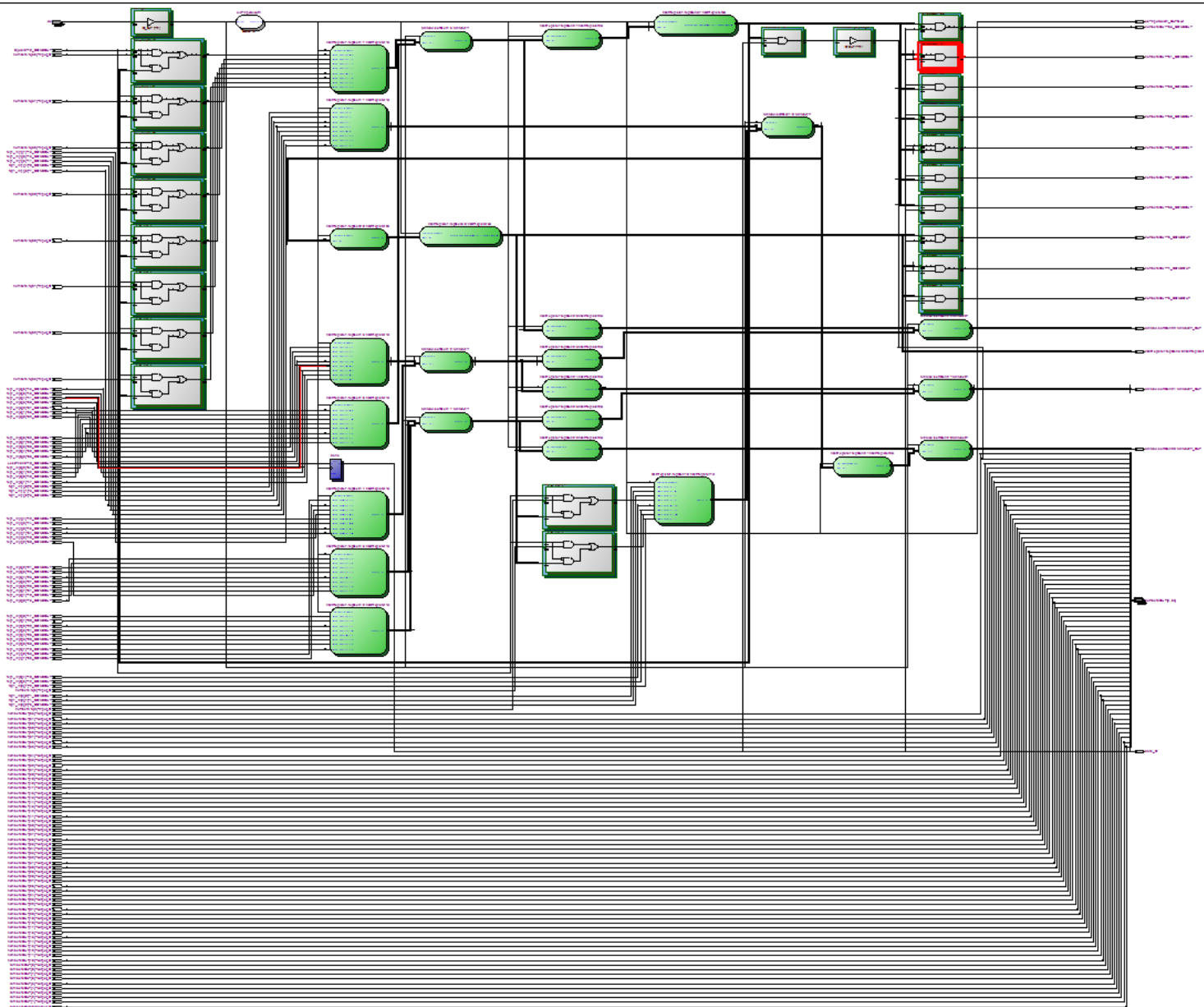
A Greater-
Quartus II's



Complete Sorter Module from Quartus II's RTL Netlist Viewer



Complete Sorter Module from Quartus II's Technology Mapped Netlist Viewer



**Complete Sorter Module from Quartus II's Technology Mapped Netlist Viewer
With Gates Expanded**

Implementation

The design of this sorter was written in VHDL. The code can be found on the following pages.

The top level module is called Sorter and contains generate blocks to instantiate the nBitRegister and MinMax components. Three separate columns of nBitRegisters are separate by and connected to MinMax blocks. On the first clock, the numbers are loaded through a port into Register 1. On every other clock, the input to Register 1 is the output of Register 3. A counter is used to determine when counting is done based on the worst case scenario. By simulating the sorter with the worst case scenario, I found that the number of clocks to complete sorting is equal to 1.5 times the number of numbers to be sorted plus one. When the counter reaches this value, done is asserted and the sorted numbers in Register 3 are assigned to the output port.

The nBitRegister module contains only a generate block to generate a certain number of D flip-flops based on the specified bit-width of the numbers. The dFlipFlop module contains the simple architectural description of a D flip-flop.

The MinMax module instantiates a GreaterThan component and two Multiplexer components and connects their signals. The Multiplexer module uses a with...select block to model the architecture of a multiplexer. The GreaterThan module instantiates two left-shift registers and sets shift_en to '1' when the MSB of each of the shift registers are equal. The architectural of the shift register uses a signal which enables it to load the input values into the local storage at the rising edge of the main clock. This is shifted left on ever positive edge of uclk while shift_en is high.

Sorter.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
use std.textio.all;

entity Sorter is
  generic (
    NUM_WIDTH : integer := 8;
    NUMBERS : integer := 8
  );
  port (
    clk,uclk      : in std_logic;
    numbersIN      : in std_logic_vector((NUM_WIDTH*NUMBERS-1) downto 0);
    numbersOUT     : out std_logic_vector((NUM_WIDTH*NUMBERS-1) downto 0);
    isDone         : out std_logic
  );
end Sorter;

architecture sortArch of Sorter is

  component MinMax is
    generic (
      NUM_WIDTH : integer := NUM_WIDTH
    );
    port (clk,uclk      : in std_logic;
          A_in, B_in    : in std_logic_vector((NUM_WIDTH-1) downto 0);
          Min, Max      : out std_logic_vector((NUM_WIDTH-1) downto 0));
  end component;

  component nBitRegister is
    generic (
      NUM_WIDTH : integer := NUM_WIDTH
    );
    port (clk          : in std_logic;
          Data_in      : in std_logic_vector((NUM_WIDTH-1) downto 0);
          Data_out      : out std_logic_vector((NUM_WIDTH-1) downto 0));
  end component;

  signal done : std_logic := '0';
  signal beginbit : std_logic := '1';
  type reglist is array ((NUMBERS-1) downto 0) of std_logic_vector((NUM_WIDTH-1) downto 0);
  signal reg1_in : reglist := (others => (others => '0'));
  signal reg1_out, reg2_in, reg2_out, reg3_in, reg3_out : reglist ;
```

```

begin

    regGen1:
    for x in (NUMBERS-1) downto 0 generate
        nBitRegister1X : nBitRegister port map
            (clk => clk, Data_in => reg1_in(x), Data_out => reg1_out(x));
    end generate regGen1;

    sortGen1:
    for y in (NUMBERS/2-1) downto 0 generate
        MinMax1Y : MinMax port map
            (clk,uclk,A_in => reg1_out(2*y), B_in => reg1_out((2*y)+1), Min =>
                reg2_in(2*y), Max => reg2_in((2*y)+1));
    end generate sortGen1;

    regGen2:
    for z in (NUMBERS-1) downto 0 generate
        nBitRegister2Z : nBitRegister port map
            (clk => clk, Data_in => reg2_in(z), Data_out => reg2_out(z));
    end generate regGen2;

    sortGen2:
    for y in (NUMBERS/2-2) downto 0 generate
        MinMax2Y : MinMax port map
            (clk,uclk,A_in => reg2_out(2*y+1), B_in => reg2_out((2*y)+2), Min =>
                reg3_in(2*y+1), Max => reg3_in((2*y)+2));
    end generate sortGen2;

    regGen3:
    for z in (NUMBERS-1) downto 0 generate
        nBitRegister3Z : nBitRegister port map
            (clk => clk, Data_in => reg3_in(z), Data_out => reg3_out(z));
    end generate regGen3;

    reg3_in(0) <= reg2_out(0);
    reg3_in(NUMBERS-1) <= reg2_out(NUMBERS-1);

    process(clk)
        variable counter : integer := 0;
        variable my_line : line;
    begin
        if (counter = 1) then
            for i in (NUMBERS-1) downto 0 loop
                reg1_in(i) <= numbersIN((NUM_WIDTH*i+(NUM_WIDTH-1)) downto (NUM_WIDTH*i));
            end loop;
        end if;
        counter := counter + 1;
    end process;
end

```

```

else
    reg1_in <= reg3_out;
end if;

if (rising_edge(clk)) then
    counter := counter + 1;

    if (counter >= ((NUMBERS/2)*3+1)) then
        done <= '1';
    else
        done <= '0';
    end if;

    --write(my_line, integer'image(counter) & ": ");
--    for i in 0 to (NUMBERS-1) loop
--        write(my_line, integer'image(conv_integer(UNSIGNED(reg3_out(i)))) & ' ');
--    end loop;
--    writeline(output, my_line);
end if;

if(done = '1') then
    for i in (NUMBERS-1) downto 0 loop
        numbersOUT((NUM_WIDTH*i+(NUM_WIDTH-1)) downto (NUM_WIDTH*i)) <= reg3_out(i);
    end loop;
    isDone <= '1';
else
    isDone <= '0';
    numbersOUT <= (others=>'0');
end if;
end process;
end sortArch;

```


nBitRegister.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nBitRegister is
    generic (
        NUM_WIDTH : integer := 8
    );
    port(
        clk      : in  std_logic;
        Data_in   : in  std_logic_vector((NUM_WIDTH-1) downto 0);
        Data_out  : out std_logic_vector((NUM_WIDTH-1) downto 0)
    );
end nBitRegister;

architecture regArch of nBitRegister is

    component dFlipFlop is
        port (clk, D : in  std_logic;
              Q      : out std_logic);
    end component;

begin

    flopGen:
    for x in (NUM_WIDTH-1) downto 0 generate
        flopx : dFlipFlop port map
            (clk => clk,
             D => Data_in(x),
             Q => Data_out(x));
    end generate flopGen;

end regArch;
```

dFlipFlop.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dFlipFlop is port(
    D,clk : in std_logic;
    Q      : out std_logic := '0');
end dFlipFlop;

architecture ff of dFlipFlop is

begin
    process(clk)
        variable store : std_logic := '0';
    begin
        if (rising_edge(clk)) then
            store := D;
        else
            store := store;
        end if;
        Q <= store;
    end process;

end ff;
```

MinMax.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MinMax is
    generic (
        NUM_WIDTH : integer := 8
    );
    port(
        clk,uclk    : in std_logic;
        A_in, B_in  : in std_logic_vector((NUM_WIDTH-1) downto 0);
        Min, Max    : out std_logic_vector((NUM_WIDTH-1) downto 0)
    );
end MinMax;

architecture MinMaxArch of MinMax is

    component GreaterThan is
        generic (
            NUM_WIDTH : integer := NUM_WIDTH
        );
        port (clk,uclk : in std_logic;
            a, b      : in std_logic_vector((NUM_WIDTH-1) downto 0);
            output    : out std_logic);
    end component;

    component Multiplexer is
        generic (
            NUM_WIDTH : integer := NUM_WIDTH
        );
        port (Sel      : in std_logic;
            A, B       : in std_logic_vector((NUM_WIDTH-1) downto 0);
            Output     : out std_logic_vector((NUM_WIDTH-1) downto 0));
    end component;

    signal greaterOut : std_logic := '0';

begin

    G1 : GreaterThan port map (clk,uclk, A_in, B_in, greaterOut);
    M1 : Multiplexer port map (greaterOut,A_in,B_in,Max);
    M2 : Multiplexer port map (greaterOut,B_in,A_in,Min);

end MinMaxArch;
```

Multiplexer.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multiplexer is
    generic (
        NUM_WIDTH : integer := 8
    );
    port(
        Sel      : in  std_logic;
        A, B     : in  std_logic_vector((NUM_WIDTH-1) downto 0);
        Output   : out std_logic_vector((NUM_WIDTH-1) downto 0)
    );
end Multiplexer;

architecture MultiArch of Multiplexer is
begin
    with Sel select
        Output <= A when '1',
                B  when others;
end MultiArch;
```

GreaterThan.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

entity GreaterThan is
    generic (
        NUM_WIDTH : integer := 8
    );
    port(
        clk,uclk : in std_logic;
        a, b      : in std_logic_vector((NUM_WIDTH-1) downto 0);
        output    : out std_logic
    );
end GreaterThan;

architecture shiftReg of GreaterThan is

    signal shift_en, shift : std_logic;
    signal a_out, b_out : std_logic_vector((NUM_WIDTH-1) downto 0);

    component lshift is
        generic (
            NUM_WIDTH : integer := NUM_WIDTH
        );
        port (
            clk,uclk : in std_logic;
            din      : in std_logic_vector((NUM_WIDTH-1) downto 0);
            shift_en : in std_logic;
            dout     : out std_logic_vector((NUM_WIDTH-1) downto 0)
        );
    end component;

begin
    shiftA : lshift port map (clk, uclk, a, shift_en, a_out);
    shiftB : lshift port map (clk, uclk, b, shift_en, b_out);

    shift_en <= '1' when ((a_out(NUM_WIDTH-1) = b_out(NUM_WIDTH-1))) else '0';
    output <= a_out(NUM_WIDTH-1);
end shiftReg;
```

lshift.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_MISC.all;
use ieee.std_logic_arith.all;

entity lshift is
    generic (
        NUM_WIDTH : integer := 8
    );
    port (
        clk, uclk : in std_logic;
        din       : in std_logic_vector((NUM_WIDTH-1) downto 0);
        shift_en  : in std_logic;
        dout      : out std_logic_vector((NUM_WIDTH-1) downto 0)
    );
end lshift;

architecture shifter of lshift is
    signal store : std_logic_vector((NUM_WIDTH-1) downto 0);
    signal count : std_logic := '0';
begin
    dout <= din when (count = '0') else store;
    process (uclk, clk) is
    begin
        if(rising_edge(clk)) then
            count <= '0';
        else
            count <= '1';
        end if;
        if(count = '0') then
            store <= din;
        elsif (shift_en = '1') then
            store(0) <= '0';    -- shift a zero into LSB
            store ((NUM_WIDTH-1) downto 1) <= store((NUM_WIDTH-2) downto 0);
        else
            store <= store;
        end if;
    end process;
end shifter;
```

Simulation and Testing

Simulation was performed using Questa 6.3g by Mentor Graphics. I created a testbench in order to test the sorter which can be seen on the following pages. The testbench contains three parameters, NUM_WIDTH, NUMBERS, and CLOCK_PERIOD to allow for the user to modify the bit width of the numbers, the number of numbers, and the period of the clock respectively. These parameters/generics are passed on to the Sorter and any sub-modules that require them.

The testbench instantiates an instance of the Sorter and drives the two clocks, clk and uclk. The uclk is used to drive the shift registers and is faster than the clk. The speed of uclk is based on the number of bits in a number. The first process in the module runs only once at the beginning of the simulation and loads the values of the numbers into the Sorter. The second process triggers on a done signal that is received from the Sorter. It copies the output into a local array to be printed. When this is finished, the final process is triggered, first printing the input values and then printing the sorted output values.

Initially, I tested the Sorter using directed testing. I assigned the numbers to the output of the testbench which inputs numbers into the sorter. This section is now commented out. Once I was satisfied with the results, I found a random number generator from a message board that I adapted to use in my testbench. It can be seen in the first process block where a random real number in the range of 0 to 1.0 is generated. This number is then scaled based on how many bits the numbers being tested are wide and converts it to an integer.

Below, you can see the output of my Sorter using 32 random 8-bit numbers. On the following page, I changed the parameters to sort 200 8-bit numbers and 64 32-bit numbers. With larger numbers, the range of values that the random number generator creates is narrower. A better random number generator would produce a better range of numbers.

Results

32 8-bit numbers:

```
# input: 56 1 224 37 92 231 22 235 18 112 233 244 230 23 242 178 67 76 28 232 32 167
42 52 21 169 41 9 84 165 248 254
```

```
# Output: 1 9 18 21 22 23 28 32 37 41 42 52 56 67 76 84 92 112 165 167 169 178 224 230
231 232 233 235 242 244 248 254
```

```
# ** Note: done!
```

```
# Time: 4900 ns Iteration: 2 Instance: /testbench
```

200 8-bit numbers:

```
# input: 201 13 101 66 242 52 97 43 215 170 24 69 116 155 23 158 194 78 34 11 203 22
198 163 98 190 244 57 117 71 15 116 49 249 237 186 201 122 213 228 94 203 241 47 18 4
116 46 158 87 102 215 48 93 208 109 57 5 164 19 67 101 10 203 83 206 9 33 215 158 179
189 74 231 206 94 99 138 33 39 211 189 115 82 198 15 182 194 176 90 5 55 28 6 86 64
133 92 101 205 130 160 110 117 118 18 211 101 174 145 127 7 16 87 83 188 64 75 27 170
48 121 243 96 196 41 137 187 250 82 113 174 250 104 8 205 118 223 134 128 94 92 114 63
187 176 201 70 85 68 201 143 52 74 141 59 105 2 248 40 132 117 86 199 62 1 211 249 56
1 224 37 92 231 22 235 18 112 233 244 230 23 242 178 67 76 28 232 32 167 42 52 21 169
41 9 84 165 248 254
# Output: 1 1 2 4 5 5 6 7 8 9 9 10 11 13 15 15 16 18 18 18 19 21 22 22 23 23 24 27 28
28 32 33 33 34 37 39 40 41 41 42 43 46 47 48 48 49 52 52 52 55 56 57 57 59 62 63 64 64
66 67 67 68 69 70 71 74 74 75 76 78 82 82 83 83 84 85 86 86 87 87 90 92 92 92 93 94 94
94 96 97 98 99 101 101 101 101 102 104 105 109 110 112 113 114 115 116 116 116 117 117
117 118 118 121 122 127 128 130 132 133 134 137 138 141 143 145 155 158 158 158 160
163 164 165 167 169 170 170 174 174 176 176 178 179 182 186 187 187 188 189 189 190
194 194 196 198 198 199 201 201 201 201 203 203 203 205 205 206 206 208 211 211 211
213 215 215 215 223 224 228 230 231 231 232 233 235 237 241 242 242 243 244 244 248
248 249 249 250 250 254
# ** Note: done!
# Time: 30100 ns Iteration: 2 Instance: /testbench
```

64 32-bit numbers:

```
# input: 1860413 3509940 2117220 2020863 1475958 1453104 1800822 990816 2948044
2766994 3154116 1111703 1337076 1075551 3164987 2245194 821895 1161979 2220732 931728
1656917 42468 3898319 643094 2079746 1843926 1362829 3128105 985230 21027 3320545
3921340 886998 21702 3515689 590831 1448934 3632711 352841 3697437 294193 1764716
3659901 3838746 3609365 367242 3801129 2799296 1061789 1196084 446864 3640326 515264
2619829 669227 816545 338691 2658580 646847 147778 1332342 2589935 3898078 3999998

# Output: 21027 21702 42468 147778 294193 338691 352841 367242 446864 515264 590831
643094 646847 669227 816545 821895 886998 931728 985230 990816 1061789 1075551 1111703
1161979 1196084 1332342 1337076 1362829 1448934 1453104 1475958 1656917 1764716
1800822 1843926 1860413 2020863 2079746 2117220 2220732 2245194 2589935 2619829
2658580 2766994 2799296 2948044 3128105 3154116 3164987 3320545 3509940 3515689
3609365 3632711 3640326 3659901 3697437 3801129 3838746 3898078 3898319 3921340
3999998

# ** Note: done!
# Time: 38800 ns Iteration: 2 Instance: /testbench
```


testbench.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.unsigned;
use ieee.std_logic_arith.conv_integer;
use std.textio.all;
USE ieee.math_real.ALL;
USE ieee.numeric_std.to_unsigned;

entity testbench is
  generic (
    NUM_WIDTH : integer := 8;
    NUMBERS : integer := 64;
    CLOCK_PERIOD : time := 10 ns
  );
end testbench;

architecture test of testbench is

  component Sorter is
    generic (
      NUM_WIDTH : integer := NUM_WIDTH;
      NUMBERS : integer := NUMBERS
    );
    port (
      clk, uclk : in std_logic;
      numbersIN : in std_logic_vector((NUM_WIDTH*NUMBERS-1) downto 0);
      numbersOUT : out std_logic_vector((NUM_WIDTH*NUMBERS-1) downto 0);
      isDone : out std_logic
    );
  end component;

  type reglist is array ((NUMBERS-1) downto 0) of std_logic_vector((NUM_WIDTH-1) downto 0);
  signal printOut : reglist;

  signal sortInput, sortOutput : std_logic_vector((NUM_WIDTH*NUMBERS-1) downto 0);
  signal clk, uclk : std_logic := '0';
  signal sortDone : std_logic := '0';

begin
  -- sortInput(79 downto 72) <= b"00000000";
  -- sortInput(71 downto 64) <= b"00000001";
  -- sortInput(63 downto 56) <= b"00000010";
  -- sortInput(55 downto 48) <= b"00000011";
```

```

--  sortInput(47 downto 40) <= b"00000100";
--  sortInput(39 downto 32) <= b"00000101";
--  sortInput(31 downto 24) <= b"00000110";
--  sortInput(23 downto 16) <= b"00000111";
--  sortInput(15 downto 8)  <= b"00001000";
--  sortInput(7  downto 0)  <= b"00001001";

S1 : Sorter port map (clk, uclk, sortInput, sortOutput, sortDone);

clk <= not clk after NUM_WIDTH*CLOCK_PERIOD;
uclk <= not uclk after CLOCK_PERIOD;

process is
    VARIABLE seed1, seed2: positive;    -- Seed values for random generator
    VARIABLE rand: real;                -- Random real-number value in range 0 to 1.0
    VARIABLE int_rand: integer;          -- Random integer value in range 0..4095
    VARIABLE stim: std_logic_vector(NUM_WIDTH-1 DOWNT0 0);--Random stimulus
begin
    for i in (NUMBERS-1) downto 0 loop
        UNIFORM(seed1, seed2, rand);      -- generate random number
        int_rand := INTEGER(TRUNC(rand*255.0));-- rescale to 0..2^N
        stim := std_logic_vector(to_unsigned(int_rand, stim'LENGTH));
        sortInput((NUM_WIDTH*i+(NUM_WIDTH-1)) downto (NUM_WIDTH*i)) <= stim;
    end loop;
    wait;
end process;

process(sortDone) is
begin
    if(sortDone = '1') then
        for i in (NUMBERS-1) downto 0 loop
            printOut(i) <= sortOutput((NUM_WIDTH*i+(NUM_WIDTH-1)) downto (NUM_WIDTH*i));
        end loop;
    end if;
end process;

process(printOut) is
    variable my_line : line;
begin
    if(sortDone = '1') then

        write(my_line, string'("input: "));
        for i in 0 to (NUMBERS-1) loop
            write(my_line, integer'image(conv_integer(UNSIGNED(sortInput(
                (NUM_WIDTH*i+(NUM_WIDTH-1)) downto (NUM_WIDTH*i)))) & ' ');
        end loop;

```

```

        writeline(output, my_line);

        write(my_line, string'("Output: ")');
        for i in 0 to (NUMBERS-1) loop
            write(my_line, integer'image(conv_integer(UNSIGNED(
                printOut(i)))) & ' ');
        end loop;
        writeline(output, my_line);
        report "done!";
    end if;
end process;

process(clk)
begin
    if (sortDone='1' and falling_edge(clk)) then
        ASSERT 1 = 2 REPORT "ending" SEVERITY FAILURE;
    end if;
end process;

end test;

```

Synthesis Results

I used Quartus II 11.1 Web Edition to synthesize my sorter. Although the design can be synthesized with parameters set to 8 8-bit numbers, larger values present a problem due to the parallel nature in which values are loaded into the sorter. The board that I targeted synthesis for was a Terasic DE0-Nano Development and Education Board featuring an Altera Cyclone IV EP4CE22F17C6N FPGA. Although I could configure and program the board, I had no way loading values onto the board or reading values from it because of the many number of pins that would have to be used.

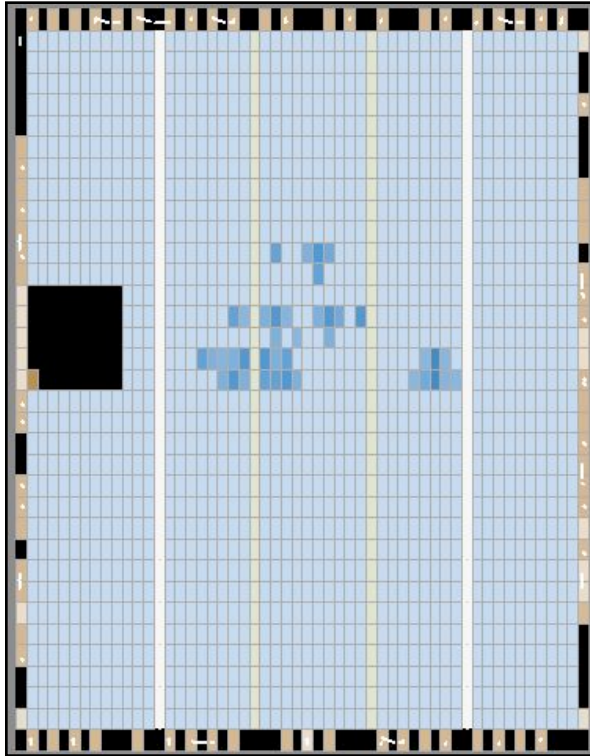
The following figures show the output summaries of synthesis. The figure on the left shows the output for 8 8-bit numbers. The figure on the right shows the summary for 150 32-bit numbers which uses 88% of the total combination functions on the board. In order to synthesize this second configuration, I had to mask out the ports. The figures on the following page show the block utilization and routing utilization for each configurations.

Flow Summary	
Flow Status	Successful - Wed Jun 06 17:31:17 2012
Quartus II 32-bit Version	11.1 Build 259 01/25/2012 SP 2 SJ Web Edition
Revision Name	testbench
Top-level Entity Name	Sorter
Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
▲ Total logic elements	532 / 22,320 (2 %)
Total combinational functions	532 / 22,320 (2 %)
Dedicated logic registers	225 / 22,320 (1 %)
Total registers	225
Total pins	131 / 154 (85 %)
Total virtual pins	0
Total memory bits	0 / 608,256 (0 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)

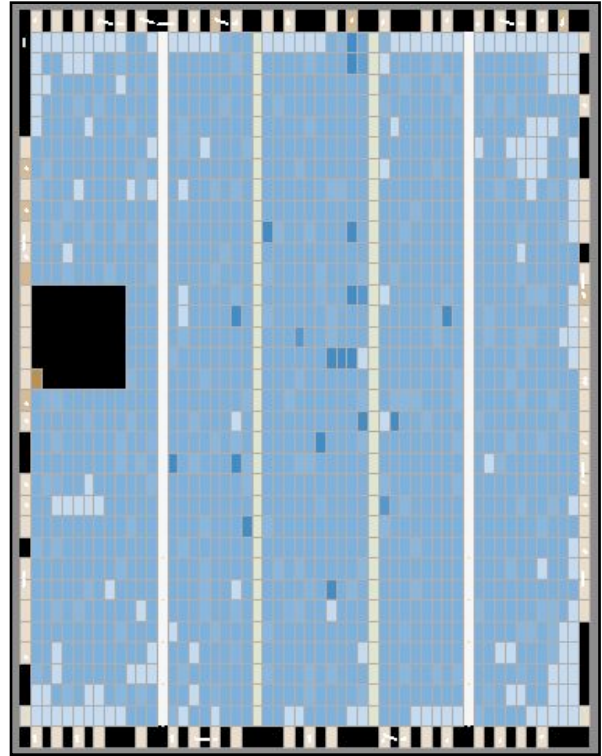
Summary for 8 8-bit numbers

Flow Summary	
Flow Status	Successful - Wed Jun 06 17:19:06 2012
Quartus II 32-bit Version	11.1 Build 259 01/25/2012 SP 2 SJ Web Edition
Revision Name	testbench
Top-level Entity Name	Sorter
Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
▲ Total logic elements	19,874 / 22,320 (89 %)
Total combinational functions	19,724 / 22,320 (88 %)
Dedicated logic registers	483 / 22,320 (2 %)
Total registers	483
Total pins	5 / 154 (3 %)
Total virtual pins	0
Total memory bits	0 / 608,256 (0 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)

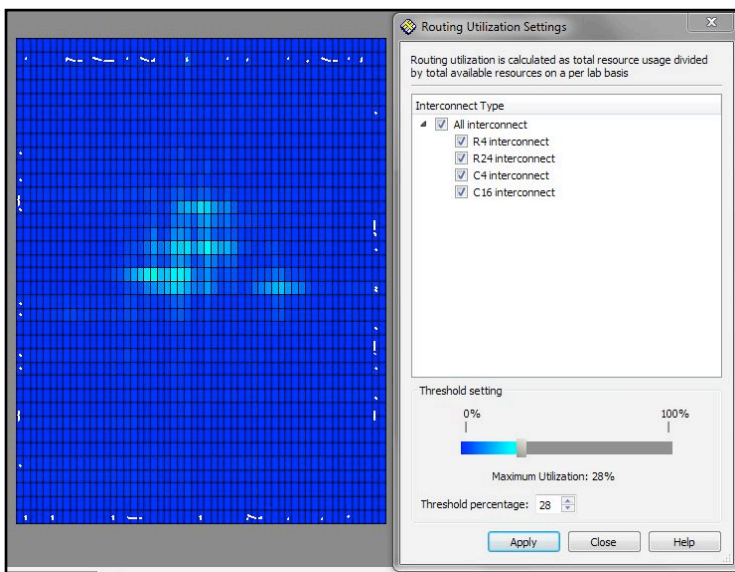
Summary for 150 32-bit numbers



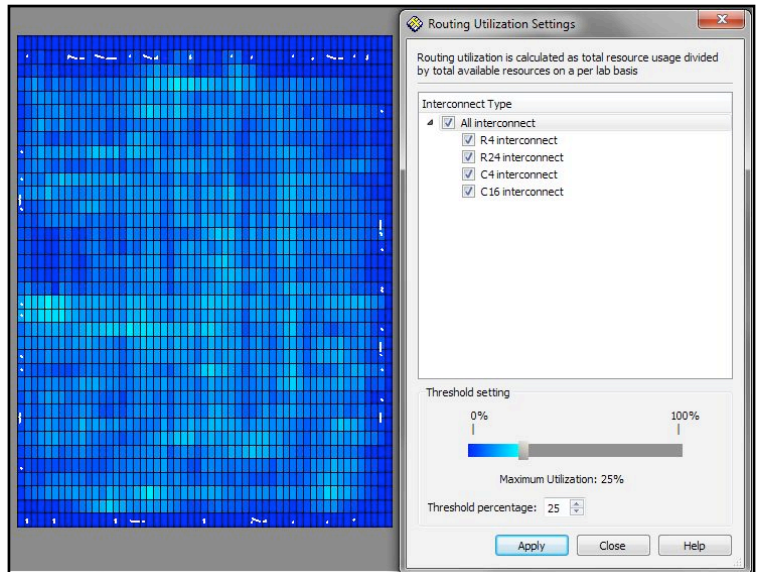
Block Utilization for 8 8-bit numbers



Block Utilization for 150 32-bit numbers



Routing Utilization for 8 8-bit numbers



Routing Utilization for 150 32-bit numbers

Possible Improvements

If only one set of numbers needed to be sorted at a time, the speed of this sorter could be improved by removing the pipelining. This would mean removing the second two registers and directly connecting the Min/Max Blocks.

Another possible improvement to my design might be in how it determines that sorting has been completed. Currently, I use a counter that finishes sorting based on the worst case scenario. An alternative would be to compare the values in each register to the values that they were previously. When the values are equal, sorting would be complete. This method would require quite a bit more hardware. The registers would have to be duplicated so that previous values would be retained since the values are currently being overwritten on each clock. There would also have to be comparisons between each of these registers.