

# CSE 834 Low Pass Filter

Derek Weitzel & Yutaka Tsutano

December 9, 2010

## 1 Introduction

Low pass filters are used in many applications from noise filtering to RF modulation. Low pass filters are well understood, and include logically simple components: registers, adders, and multipliers. In this project, we will design a low power low pass filter using VHDL and the Cadence hardware designer. We will include optimizations for power such as reducing components and reducing transitions.

In section 2, we will discuss our design philosophy and design decisions. In section 3 we will describe how we constructed the components and connected them. Finally, in section 4, we will discuss problems we experienced in our design and the tools we used.

## 2 Design

### 2.1 Q15 Format

For our filter design, fixed format number format called Q is used. Unlike floating point numbers, Q-format numbers require only a standard integer Arithmetic Logic Unit (ALU) to perform rational number calculations. This means that we do not need to add an Floating Point Unit (FPU) to our design, which requires additional power.

Q format numbers are represented using the Q notation which is written as  $Q_{m,n}$  where  $m$  is the number of bits set aside to designate the two's complement integer portion of the number, and  $n$  is the number of bits used to designate the fractional portion of the number.

In our case, we use simplified notation  $Q_n$  since we assume that the numbers are normalized into the range of  $[-1, 1)$ . Notice that this assumption does not affect the generality of the filter, but it simplifies the multiplication operation because the multiplication result never exceeds the range of  $[-1, 1)$ .

Figure 6 shows the comparison between two's complement integer and Q7 numbers. The only difference between the two is the weight assigned for each bit.

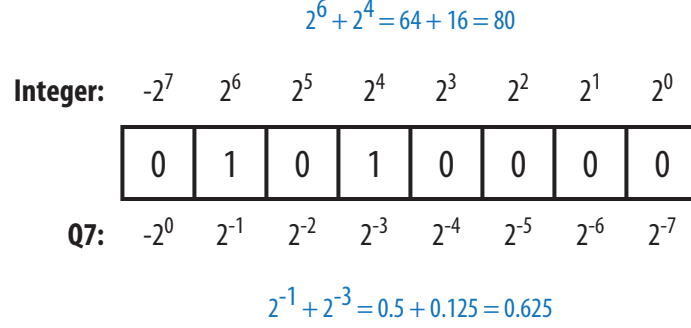


Figure 1: Comparison between integer and Q7 format.

## 2.2 Filter Characteristics

Coefficient	Value	Q15
$C_0$	-0.022663985459552	1111110100011010
$C_1$	1.04697822237622e-17	0000000000000000
$C_2$	0.273977082565524	0010001100010001
$C_3$	0.497373805788057	0011111110101001
$C_4$	0.273977082565524	0010001100010001
$C_5$	1.04697822237622e-17	0000000000000000
$C_6$	-0.022663985459552	1111110100011010

Table 1: Coefficient values.

The coefficients for the filter is shown in Table 1. These values were computed using Filter Design & Analysis Tool (Figure 2) in Matlab Signal Processing Toolbox which is one of the most widely used filter design tools in the industry. We have chosen a 8<sup>th</sup> order FIR filter with a Hamming window. This filter is also one of the most widely used in the industry and has straight-forward characteristic.

## 2.3 Input & Output

The input and output of our design are shown in Table 2. The external clock will allow us to connect to arbitrary clocked external input, as long as it meets the timing requirements of our design. The 16 bit input and output are in the Q15 format described in Section 2.1. The Q15 format is an industry standard, hence we expect that these required input formats are reasonable.

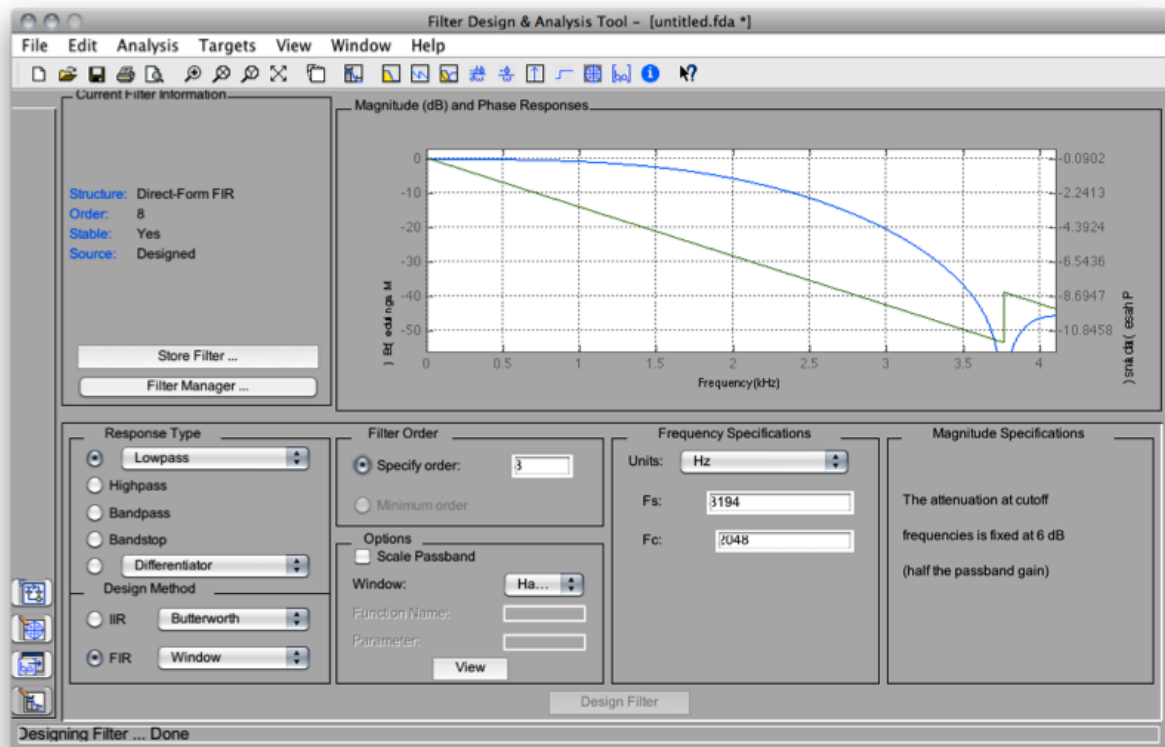


Figure 2: Matlab Filter Design & Analysis Tool.

Input/Output	Description	Bits
Input	Clock	1
Input	Data In	16
Output	Convolution	16

Table 2: Description of input and output to the filter.

## 2.4 Power Optimization

We will optimize power by removing duplicate multipliers. These duplicate multipliers are created because the filter is symmetric, therefore an adder can be used remove an adder. In the naive filter, shown in Figure 3, the constant is multiplied against each register value. Since the constants are symmetric, ie  $C0$  is the same as  $C6$ , this can be optimized using the associative rule. In Figure 4, you can see that we have removed 3 of the multipliers. Instead, we add the values in the registers before multiplying the constants. This eliminates 3 large and power draining multipliers. It also simplifies the final adder from 7 input to 4.

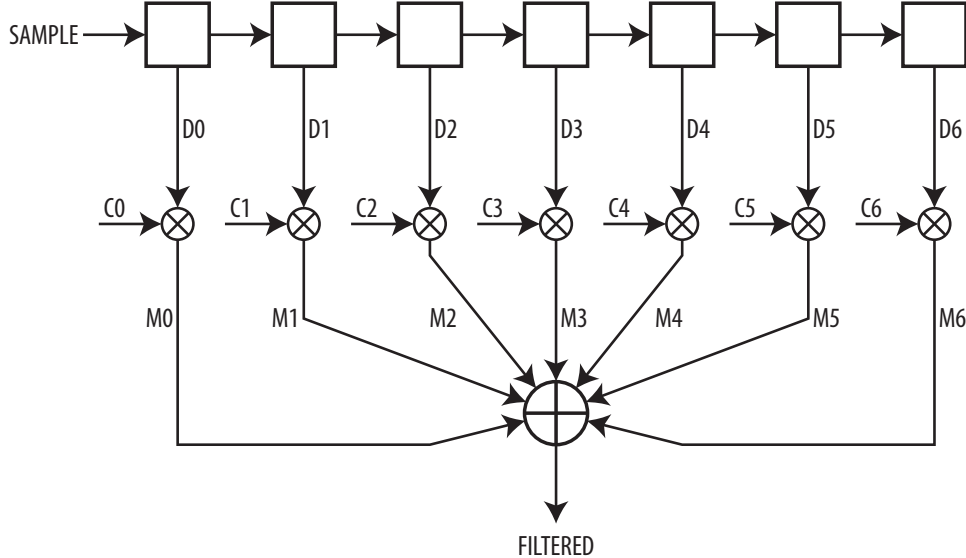


Figure 3: Naive filter implementation.

The adders shown at the bottom of Figure 4 are frequently changing as the final value is stabilized through the convolution hardware above. Therefore, it is important that this hardware is optimized for power. With reducing the number of transistors as our goal, we designed ripple-carry adders to do the final addition. The ripple-carry adders have less hardware than the optimized version that VHDL synthesized when doing a '+'. Also, the ripple carry adders will localize any transition to only the nodes that require it.

Compare the ripple carry to propagate/carry (PG) adders that are designed to optimize latency. The PG adders have significantly more hardware since it has to pre-calculate propagates and carries.

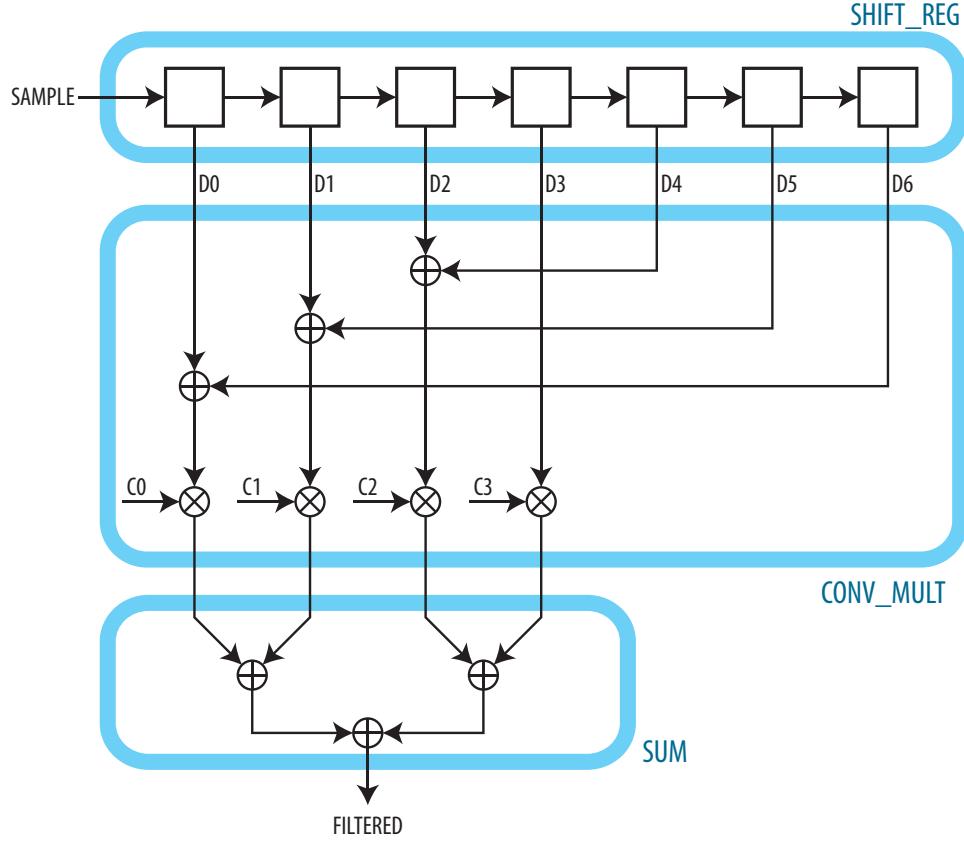


Figure 4: Optimized filter implementation.

We also attempted to use a clock-gated D flip flop seen in Figure 5. While doing simulations of the VHDL, we discovered that this design was flawed. When the clock was high, Q stabilized to the value of D. This is because the XOR gate with the clock and the D input would cause the ‘clock’ input to the flip flop to change, therefore causing it to save the input. The clock-gated D flip-flop acted as a latch. We did not use this optimization in the final design.

## 3 Implementation

### 3.1 VHDL Design

We decided to design our filter using as much HDL as possible. Therefore, every component is written in VHDL, and are synthesized for the final design. The VHDL is separated into components listed below. These components are then synthesized using the `rc` and Encounter cadence synthesizer.

Separate components:

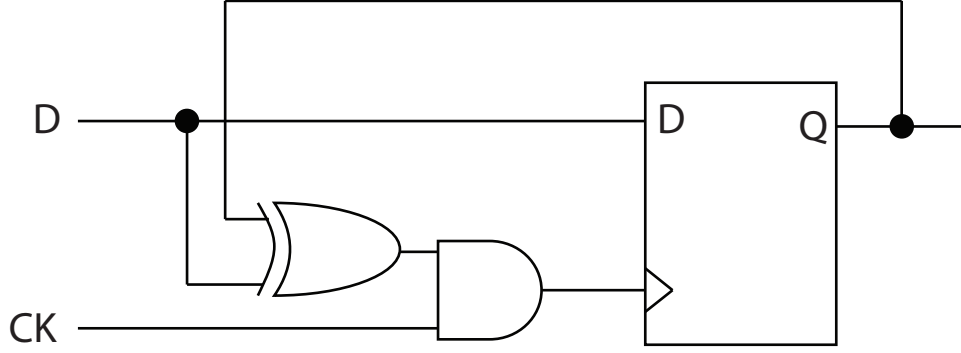


Figure 5: Clock gating low power flip-flop (Credit: Sharad Seth).

- **SHIFT\_REG**: Shift Register (Appendix A.2.1)
  - **D\_FF**: D-Flip Flop (Appendix A.2.2)
- **CONV\_MULT**: Convolution (Appendix A.3.1)
- **SUM**: Summation (Appendix A.4.1)
  - **ADD16**: 16-Bit Adder (Appendix A.4.2)
    - \* **FULL\_ADDER**: Full Adder (Appendix A.4.3)

The source of the VHDL components are included in the appendix.

During early synthesis, we found that the standard libraries did not have a flip-flop available. We needed a flip-flop in order to create our registers. Therefore, we created a D Flip Flop using the structural specification.

## 3.2 Functional Verification

To verify the correctness of the VHDL design files, we have written a C++ program that generates test cases. This program first generates test input with random noise. Then it computes the output using the equivalent filter algorithm in Q15 format. The result is stored in a text file (Appendix B.2) which can be read by a VHDL testbench `filter_tb.vhd` (Appendix A.1.1).

For VHDL simulation, we used ModelSim from Mentor Graphics. This tool loads and compiles all VHDL files including the test bench, then shows the waveform and output to the window. Our testbench file follows the VHDL standard and prints the results into a text file, so it should work on other environments other than ModelSim.

We verified that the every bit of the C++ program output and VHDL testbench output matched, therefore the VHDL should be correct with high probability.

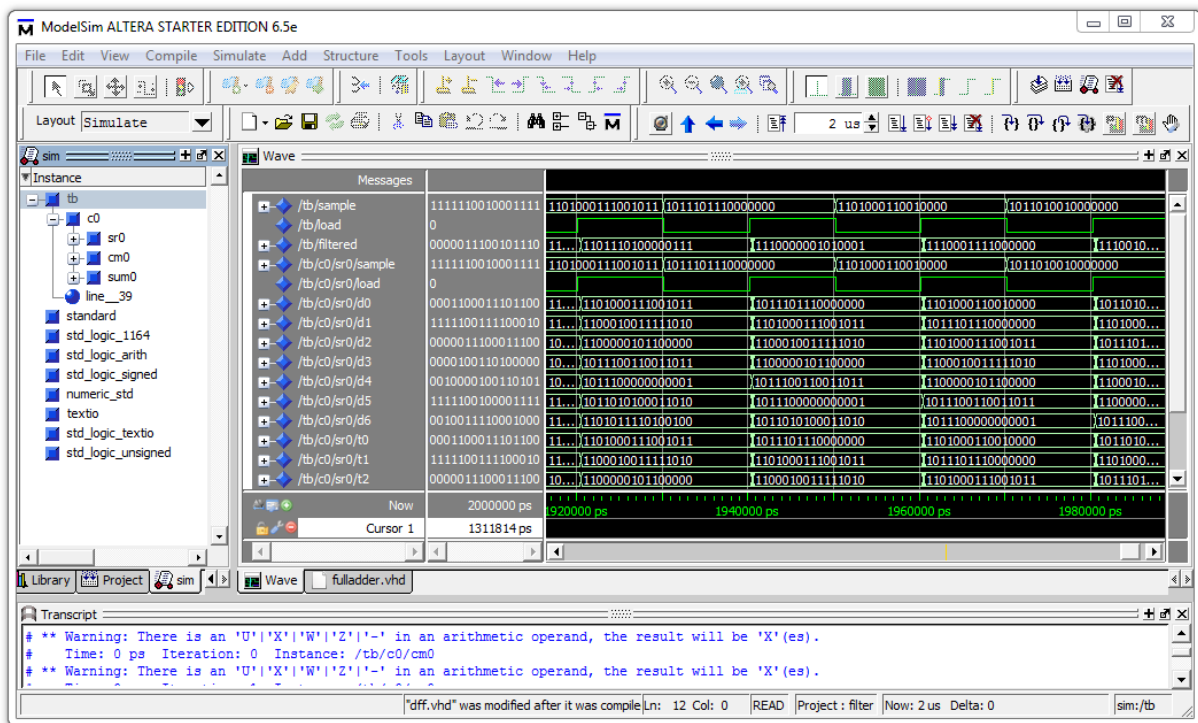


Figure 6: ModelSim from Mentor Graphics was used for VHDL simulation.

### 3.3 Synthesis

Synthesis was done using rc cadence synthesizer and the Encounter tool. For the encounter tool, we followed the instructions listed in lab 3. The Encounter routing and placement on the CONV\_MULT component took 45 minutes to complete while the FullAdder was near instant. See Figure 7 for the synthesizedConvolution hardware.

### 3.4 Final Layout & Routing

Since each component was designed in a separate VHDL file, they had to be connected at a high level. See Figure 8 for the schematic of the connected high level components. This logical layout closely resembles the layout shown in Figure 4.

After creating the schematic, we used the Cadence router to route between the top level components. The chip size needed to be increased to 2500x2500 microns in order to accommodate the components. The convolution chip alone is 1500x1500 microns. The routing completes in about an hour with no un-routable endpoints. See Figure 9 for the routed final layout.

## 4 Conclusion

In this project, we created a power optimized low pass filter. We avoid using a floating point unit by utilizing the Q15 format. The filter is symmetric, allowing for optimizations. The input and output is kept simple. The power optimizations were done at design time using the algorithmic level. The components were developed in VHDL. The design was verified both at the functional level with a C++ program, and using the VHDL through a simulator. The components were synthesized using the Cadence tools. Finally, the layout was done using the autorouter in Cadence.

Both the DRC and LVS for the final layout failed. During autorouting, the router introduced DRC errors. LVS fails due to a cadence configuration error that I was unable to correct. Although both DRC and LVS fail, the VHDL simulates correctly and is functionally correct.



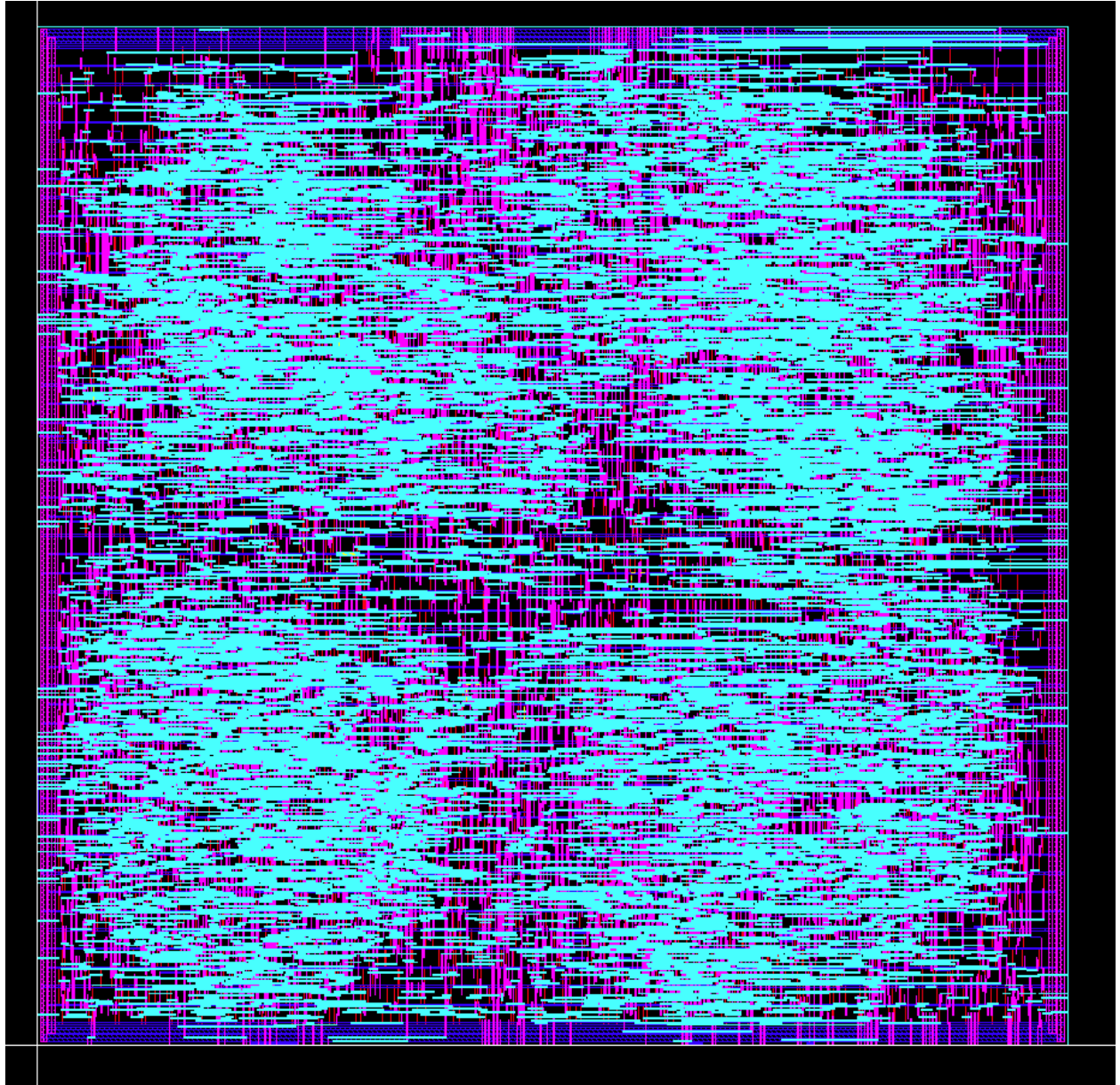


Figure 7: Convolution hardware after synthesis.



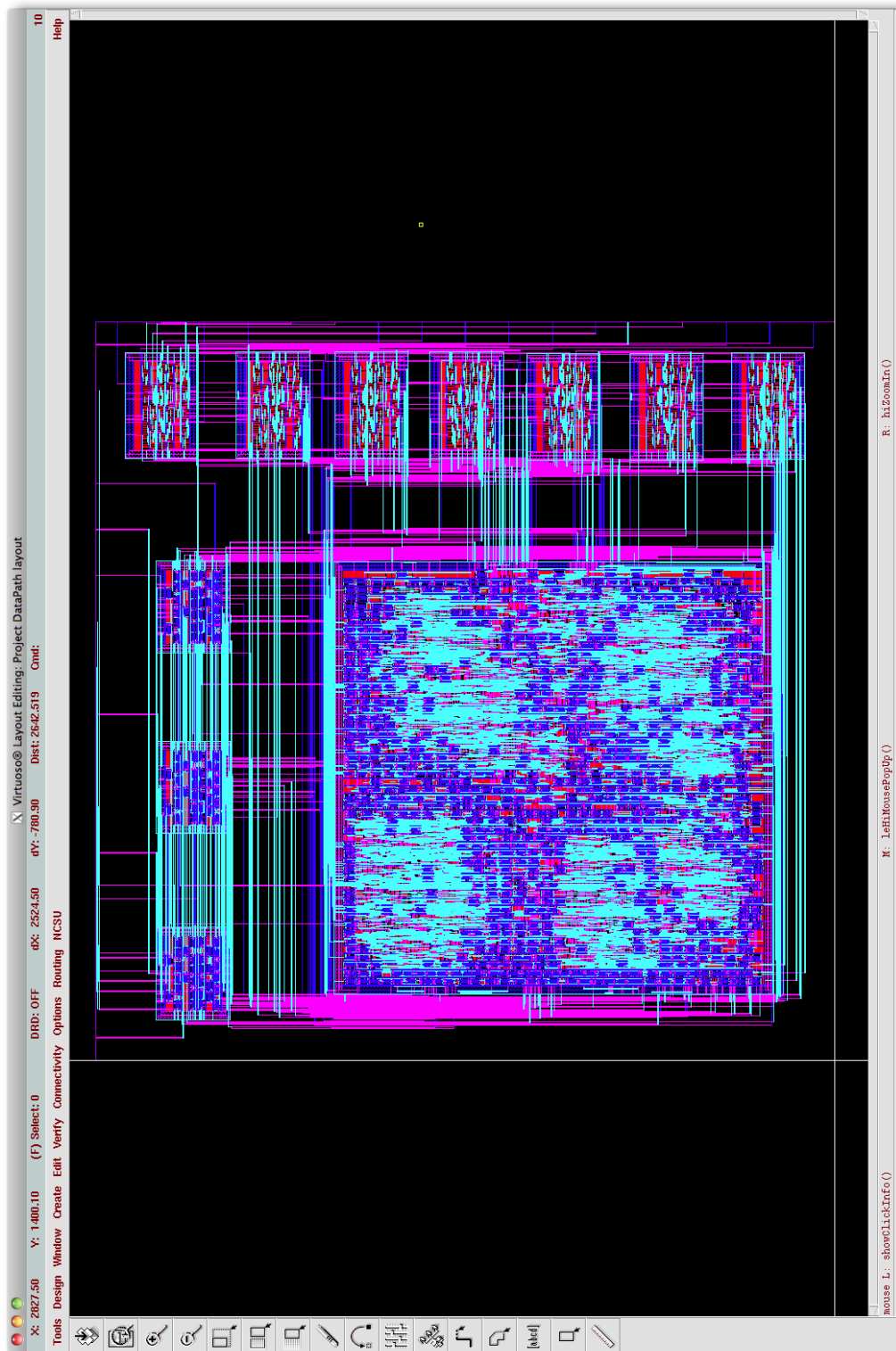


Figure 9: Filter Routed.

# A VHDL Design Files

## A.1 Top Level

### A.1.1 Test Bench (filter\_tb.vhd)

---

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_signed.all;
4 use IEEE.numeric_std.all;
5 use IEEE.std_logic_textio.all;
6 library STD;
7 use STD.textio.all;
8
9 entity TB is
10 end TB;
11
12 architecture FILTER_TEST of TB is
13
14     — Inputs.
15     signal sample: std_logic_vector(15 downto 0);
16     signal load: std_logic;
17
18     — Output.
19     signal filtered: std_logic_vector(15 downto 0);
20
21     component FILTER is
22         port (
23             sample: in std_logic_vector(15 downto 0);
24             load: in std_logic;
25             filtered: out std_logic_vector(15 downto 0)
26         );
27     end component;
28
29     —type TEXTFILE is file of String;
30     file INFILE: TEXT open READMODE is "input.txt";
31     file OUTFILE: TEXT open WRITEMODE is "output.txt";
32
33     for C0: FILTER use entity work.FILTER(BEHAVIORAL);
34 begin
35
36     C0: FILTER port map (sample, load, filtered);
37
38     process
39         variable v_ln: line;
```

```

40         variable v_sample: std_logic_vector(15 downto 0);
41     begin
42         sample <= "0000000000000000";
43         for I in 0 to 50 loop
44             load <= '0';
45             wait for 100 ps;
46             load <= '1';
47             wait for 100 ps;
48         end loop;
49
50         load <= '0';
51         for I in 0 to 100 loop
52             readline(INFILE, v_ln);
53             read(v_ln, v_sample);
54             sample <= v_sample;
55             load <= '0';
56             write(v_ln, filtered);
57             writeline(OUTFILE, v_ln);
58             wait for 10 ns;
59             load <= '1';
60             wait for 10 ns;
61         end loop;
62         wait;
63     end process;
64
65 end FILTER_TEST;

```

---

### A.1.2 Top Level Component (filter.vhd)

---

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4  use IEEE.numeric_std.all;
5
6  entity FILTER is
7      port (
8          sample: in std_logic_vector(15 downto 0);
9          load: in std_logic;
10         filtered: out std_logic_vector(15 downto 0)
11     );
12 end FILTER;
13
14 architecture BEHAVIORAL of FILTER is

```

```

15     signal D0, D1, D2, D3, D4, D5, D6: std_logic_vector(15 downto
16         0);
17     signal M0, M1, M2, M3: std_logic_vector(15 downto 0);
18     signal c_out: std_logic;
19
20     component SHIFT_REG is
21     port (
22         sample: in std_logic_vector(15 downto 0);
23         load: in std_logic;
24         D0, D1, D2, D3, D4, D5, D6: out std_logic_vector(15
25             downto 0)
26     );
27     end component;
28
29     component SUM16 is
30     port (
31         M0, M1, M2, M3: in std_logic_vector(15 downto 0);
32         S: out std_logic_vector(15 downto 0);
33         c_out: out std_logic
34     );
35     end component;
36
37     component CONV_MULT is
38     port (
39         D0, D1, D2, D3, D4, D5, D6: in std_logic_vector(15
40             downto 0);
41         C0, C1, C2, C3: in std_logic_vector(15 downto 0);
42         M0, M1, M2, M3: out std_logic_vector(15 downto 0)
43     );
44     end component;
45
46     for SR0: SHIFT_REG use entity work.SHIFT_REG(BEHAVIORAL);
47     for CM0: CONV_MULT use entity work.CONV_MULT(BEHAVIORAL);
48     for SUM0: SUM16 use entity work.SUM16(BEHAVIORAL);
49
50 begin
51
52     SR0: SHIFT_REG port map (sample, load, D0, D1, D2, D3, D4, D5
53         , D6);
54
55     CM0: CONV_MULT port map (D0, D1, D2, D3, D4, D5, D6,
56         "1111110100011010", "0000000000000000",
57         "0010001100010001", "0011111110101001",
58         M0, M1, M2, M3);
59
60     SUM0: SUM16 port map (M0, M1, M2, M3, filtered, c_out);

```

```
56
57 end BEHAVIORAL;
```

---

## A.2 Shift Register

### A.2.1 Shift Register (shiftreg.vhd)

---

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_signed.all;
4  use IEEE.numeric_std.all;
5
6  entity SHIFT_REG is
7      port (
8          sample: in std_logic_vector(15 downto 0);
9          load: in std_logic;
10         D0, D1, D2, D3, D4, D5, D6: out std_logic_vector(15
11             downto 0)
12     );
13 end SHIFT_REG;
14
15 architecture BEHAVIORAL of SHIFT_REG is
16     signal T0, T1, T2, T3, T4, T5, T6: std_logic_vector(15 downto
17         0);
18
19     component D_FF is
20         port(
21             D: in std_logic_vector(15 downto 0);
22             CLK: in std_logic;
23             Q: out std_logic_vector(15 downto 0)
24         );
25     end component;
26
27     for C0: D_FF use entity work.D_FF(BEHAVIORAL);
28     for C1: D_FF use entity work.D_FF(BEHAVIORAL);
29     for C2: D_FF use entity work.D_FF(BEHAVIORAL);
30     for C3: D_FF use entity work.D_FF(BEHAVIORAL);
31     for C4: D_FF use entity work.D_FF(BEHAVIORAL);
32     for C5: D_FF use entity work.D_FF(BEHAVIORAL);
33     for C6: D_FF use entity work.D_FF(BEHAVIORAL);
34 begin
```

```

35
36     C0: D_FF port map (sample , load , T0);
37     C1: D_FF port map (T0, load , T1);
38     C2: D_FF port map (T1, load , T2);
39     C3: D_FF port map (T2, load , T3);
40     C4: D_FF port map (T3, load , T4);
41     C5: D_FF port map (T4, load , T5);
42     C6: D_FF port map (T5, load , T6);
43
44     D0 <= T0;
45     D1 <= T1;
46     D2 <= T2;
47     D3 <= T3;
48     D4 <= T4;
49     D5 <= T5;
50     D6 <= T6;
51
52 end BEHAVIORAL;

```

---

### A.2.2 D-Flip Flop (dff.vhd)

---

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_FF is
5      port(
6          D: in std_logic_vector(15 downto 0);
7          CLK: in std_logic;
8          Q: out std_logic_vector(15 downto 0)
9      );
10 end D_FF;
11
12 architecture BEHAVIORAL of D_FF is
13
14     signal a, b, c, e, f, g, h, lclk: std_logic_vector(15 downto
15         0) := "0000000000000000";
16
17 begin
18     a <= not (D and g) after 100 ps;
19     b <= not (a and c) after 100 ps;
20     c <= not (b and lclk) after 100 ps;
21     e <= not (c and f) after 100 ps;
22     f <= not (g and e) after 100 ps;

```



```

23     g <= not (c and lclk and a) after 100 ps;
24     h <= e xor D after 100 ps;
25     — lclk <= h and (15 downto 0 => CLK) after 100 ps;
26     lclk <= (15 downto 0 => CLK) after 100 ps;
27
28     Q <= e;
29
30 end BEHAVIORAL;

```

---

## A.3 Convolution

### A.3.1 Convolution (convmult.vhd)

---

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_signed.all;
4  use IEEE.numeric_std.all;
5
6  entity CONVMULT is
7      port (
8          D0, D1, D2, D3, D4, D5, D6: in std_logic_vector(15 downto
9              0);
10         C0, C1, C2, C3: in std_logic_vector(15 downto 0);
11         M0, M1, M2, M3: out std_logic_vector(15 downto 0)
12     );
13 end CONVMULT;
14
15 architecture BEHAVIORAL of CONVMULT is
16     signal A0, A1, A2, A3: std_logic_vector(16 downto 0);
17     signal B0, B1, B2, B3: std_logic_vector(32 downto 0);
18 begin
19     — Addition for taking advantage of the filter symmetry.
20     — Result is in Q1.15.
21     A0 <= (D0(15) & D0) + (D6(15) & D6);
22     A1 <= (D1(15) & D1) + (D5(15) & D5);
23     A2 <= (D2(15) & D2) + (D4(15) & D4);
24     A3 <= (D3(15) & D3);
25
26     — Multiply by coeffs.
27     B0 <= A0 * C0;
28     B1 <= A1 * C1;
29     B2 <= A2 * C2;

```

```

30     B3 <= A3 * C3;
31
32     — Convert from Q1.31 to Q15.
33     M0 <= B0(31 downto 16);
34     M1 <= B1(31 downto 16);
35     M2 <= B2(31 downto 16);
36     M3 <= B3(31 downto 16);
37
38 end BEHAVIORAL;

```

---

## A.4 Summation

### A.4.1 Sum (sum.vhd)

---

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_signed.all;
4  use IEEE.numeric_std.all;
5
6  entity SUM16 is
7      port (
8          M0, M1, M2, M3: in std_logic_vector(15 downto 0);
9          S: out std_logic_vector(15 downto 0);
10         c_out: out std_logic
11     );
12 end SUM16;
13
14 architecture BEHAVIORAL of SUM16 is
15     signal S01, S23, S0123: std_logic_vector(15 downto 0);
16     signal C01, C23, C0123: std_logic;
17
18     component ADD16 is
19         port(
20             A, B: in std_logic_vector(15 downto 0);
21             c_in: in std_logic;
22             S: out std_logic_vector(15 downto 0);
23             c_out: out std_logic
24         );
25     end component;
26
27     for c0: ADD16 use entity work.ADD16(BEHAVIORAL);
28     for c1: ADD16 use entity work.ADD16(BEHAVIORAL);
29     for c: ADD16 use entity work.ADD16(BEHAVIORAL);

```

```

30 begin
31
32     c0: ADD16 port map (M0, M1, '0', S01, C01);
33     c1: ADD16 port map (M2, M3, '0', S23, C23);
34     c: ADD16 port map (S01, S23, '0', S0123, C0123);
35
36     c_out <= C01 or C23 or C0123;
37     S <= S0123;
38
39     --S <= M0 + M1 + M2 + M3;
40     --c_out <= '0';
41
42 end BEHAVIORAL;

```

---

#### A.4.2 16-Bit Adder (add16.vhd)

---

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_signed.all;
4 use IEEE.numeric_std.all;
5
6 entity ADD16 is
7     port (
8         A, B: in std_logic_vector(15 downto 0);
9         c_in: in std_logic;
10        S: out std_logic_vector(15 downto 0);
11        c_out: out std_logic
12    );
13 end ADD16;
14
15 architecture BEHAVIORAL of ADD16 is
16     signal im: std_logic_vector(14 downto 0);
17
18     component FULLADDER is
19         port(
20             a      : in std_logic;
21             b      : in std_logic;
22             c_in   : in std_logic;
23             sum    : out std_logic;
24             c_out  : out std_logic
25         );
26     end component;
27 begin
28

```

```

29     c0: FULLADDER port map(A(0), B(0), c_in, S(0), im(0));
30
31     c: for i in 1 to 14 generate
32         c1to14: FULLADDER port map (A(i), B(i), im(i-1), S(i),
33             im(i));
34     end generate;
35     c15: FULLADDER port map(A(15), B(15), im(14), S(15), c_out);
36
37 end BEHAVIORAL;

```

---

#### A.4.3 Full Adder (fulladder.vhd)

---

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_signed.all;
4  use IEEE.numeric_std.all;
5
6  entity FULLADDER is
7      port(
8          a      : in std_logic;
9          b      : in std_logic;
10         c_in   : in std_logic;
11         sum    : out std_logic;
12         c_out  : out std_logic
13     );
14 end FULLADDER;
15
16 architecture BEHAVIORAL of FULLADDER is
17 begin
18
19     sum <= a xor b xor c_in;
20     c_out <= (a and b) or (c_in and (a or b));
21
22 end BEHAVIORAL;

```

---

## B C++ Files for Verification

### B.1 C++ Source Code (filter.cpp)

---

```

1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4
5  typedef short Q15;
6
7  inline Q15 double_to_q15(double x)
8  {
9      return x * 32768.0;
10 }
11
12 inline double q15_to_double(Q15 x)
13 {
14     return x / 32768.0;
15 }
16
17 void print_q15(Q15 x)
18 {
19     for (int i = 15; i >= 0; i--) {
20         std::cout << ((x >> i) & 0x01);
21     }
22 }
23
24 Q15 lpf(Q15 *history, const Q15 *coeffs, Q15 data)
25 {
26     Q15 result = 0;
27
28     for (int i = 5; i >= 0; i--) {
29         history[i + 1] = history[i];
30     }
31     history[0] = data;
32
33     Q15 m0 = (coeffs[0] * (history[0] + history[6])) >> 16;
34     Q15 m1 = (coeffs[1] * (history[1] + history[5])) >> 16;
35     Q15 m2 = (coeffs[2] * (history[2] + history[4])) >> 16;
36     Q15 m3 = (coeffs[3] * (history[3])) >> 16;
37
38     return m0 + m1 + m2 + m3;
39 }
40
41 int main(void)
42 {
43     static const Q15 coeffs[] = {
44         double_to_q15(-0.022663985459552),
45         double_to_q15(0.0),

```

```

46         double_to_q15(0.273977082565524),
47         double_to_q15(0.497373805788057)
48     };
49
50     // Generate input data.
51     static const int DATALEN = 128;
52     double data_double[DATALEN];
53     Q15 data[DATALEN];
54     for (int i = 0; i < DATALEN; i++) {
55         double theta = 360 * i / 180.0 * 3.1415 / DATALEN;
56         data_double[i] = std::sin(theta) * 0.5;
57         data_double[i] += 0.2 * ((2.0 * rand() / RANDMAX) - 1.0)
58         ;
59         data[i] = double_to_q15(data_double[i]);
60     }
61
62     // Print coeffs.
63     std::cout << "Coeffs:\n";
64     for (int i = 0; i < 4; i++) {
65         std::cout << "c" << i << " = ";
66         print_q15(coeffs[i]);
67         std::cout << "\n";
68     }
69
70     // Print input data.
71     std::cout << "Input:\n";
72     for (int i = 0; i < DATALEN; i++) {
73         #if 0
74             std::printf("%16.8f", data_double[i]);
75             std::cout << "\t";
76         #else
77             print_q15(data[i]);
78         #endif
79         std::cout << "\n";
80     }
81
82     // Print output data.
83     std::cout << "Output:\n";
84     Q15 history[] = {0, 0, 0, 0, 0, 0, 0};
85     for (int i = 0; i < DATALEN; i++) {
86         Q15 result = lpf(history, coeffs, data[i]);
87         #if 0
88             std::printf("%16.8f", q15_to_double(result));
89             std::cout << "\t";
90         #else

```

```

90         print_q15(result);
91 #endif
92         std::cout << '\n';
93     }
94
95     return 0;
96 }

```

---

## B.2 Test Case (testcase.txt)

---

```

1 Coeffs:
2 c0 = 1111110100011010
3 c1 = 0000000000000000
4 c2 = 0010001100010001
5 c3 = 0011111110101001
6 Input:
7 1110011001100111
8 1111000001000111
9 0001001101011100
10 0000011101000101
11 0000111000101001
12 0000000100101001
13 1111101101100011
14 0001111010110111
15 0010000110101011
16 0011000110011110
17 0001100000110100
18 0010000111100101
19 0011010010000000
20 0000111001001011
21 0001000110111100
22 0010110001111111
23 0011011000000100
24 0001011000110110
25 0010101110000000
26 0001110100111001
27 0011000011111101
28 0100000001110100
29 0011110011111111
30 0100111111100100
31 0100110011011001
32 0011110110100011
33 0010100001011010

```

34 0100010111110110  
35 0011101001111000  
36 0100100110011011  
37 0101010010110011  
38 0100110101011000  
39 0011001111010110  
40 0010100011000000  
41 0100101111000111  
42 0011011010000011  
43 0100010110001111  
44 0100101100110110  
45 0101011001100010  
46 0011010101011101  
47 0010111000101101  
48 0101001010010000  
49 0100001111011000  
50 0100001111011110  
51 0011110011111001  
52 0001110110000111  
53 0011100000110111  
54 0100001100011110  
55 0010000110011110  
56 0010011110111010  
57 0011011000111111  
58 0010010011111101  
59 0001011000100010  
60 0001010101100001  
61 0001011011110111  
62 0000101001001011  
63 0001011111001110  
64 0010100111101101  
65 0010011110001000  
66 1111100100001111  
67 0010000100110101  
68 0000100110100000  
69 0000011100011100  
70 1111100111100010  
71 0001100011101100  
72 1111110010001111  
73 1110110111000011  
74 1110000110101010  
75 0000101001110010  
76 1101101010100010  
77 1110110101110111  
78 1110010010000100



79 1101110000011010  
80 1111100111010101  
81 1110001101011101  
82 1101110101001001  
83 1111001100001000  
84 1100001011011001  
85 1110010011001100  
86 1110001011011101  
87 1110001110001001  
88 1011110101101000  
89 1011010110111111  
90 1101011001000000  
91 1101110110100101  
92 1100111110111110  
93 1101001110101000  
94 1101000110110001  
95 1101111001110011  
96 1101011110100100  
97 1011010100011010  
98 1011100000000001  
99 1011100110011011  
100 1100000101100000  
101 1100010011111010  
102 1101000111001011  
103 1011101110000000  
104 1101000110010000  
105 1011010010000000  
106 1011110001011101  
107 1100001100100100  
108 1100000001000110  
109 1011011111011100  
110 1011001101000101  
111 1011001100100100  
112 1100100111010000  
113 1101011100001010  
114 1011000100110010  
115 1100110010001100  
116 1100110010000011  
117 1110010111010110  
118 1101110101001001  
119 1101010110001001  
120 1110100100000101  
121 1101110111001001  
122 1110101101100010  
123 1100101100000100

```

124 1101000001100010
125 1110110011010001
126 1101000110110101
127 1101001010001111
128 1101111011100100
129 1101001111110111
130 1110110000001101
131 1101101101001000
132 0000000101010111
133 0001000000100010
134 1110111110001001
135 Output :
136 0000000001001010
137 0000000000101101
138 1111110001000101
139 1111011101100100
140 1111101100010010
141 0000001110100011
142 0000011010111100
143 0000010010000000
144 0000000011111101
145 0000001010010010
146 0000101100101101
147 0001001011111001
148 0001001110111000
149 0001000011110001
150 0001001001011001
151 0001001010010110
152 0000110001001000
153 0000101111010010
154 0001001111001101
155 0001011000010010
156 0001001000011110
157 0001000010100000
158 0001001010100001
159 0001011111011010
160 0001110110111010
161 0010000111101000
162 0010010110111011
163 0010010011111000
164 0001111000000101
165 0001101001010001
166 0001110100011001
167 0010000010100001
168 0010010011011101

```

169 0010100001111011  
170 0010010001101000  
171 0001101110011100  
172 0001100111011100  
173 0001111000101000  
174 0001111111100101  
175 0010000111111111  
176 0010011010101101  
177 0010010110001001  
178 0001110111011110  
179 0001110001111010  
180 0010001001111001  
181 0010010010001000  
182 0010000101011100  
183 0001101011001110  
184 0001011000111101  
185 0001100111111110  
186 0001101110110000  
187 0001011000111101  
188 0001010100000101  
189 0001011011111110  
190 0001001100000100  
191 0000110011101101  
192 0000101010011010  
193 0000100100100110  
194 0000100001000100  
195 0000110011101000  
196 0001001001110110  
197 0000111001100100  
198 0000011111100010  
199 0000100000110111  
200 0000011100101110  
201 0000001001011101  
202 0000001010110001  
203 0000010100011110  
204 1111111111011011  
205 1111011101010011  
206 1111011101001111  
207 1111100110101010  
208 1111011000110101  
209 1111001011101010  
210 1111000111101001  
211 1111001101000101  
212 1111010111111010  
213 1111010001000110

214 1111001001100001  
 215 1111000000001010  
 216 1110101111101111  
 217 1110111000000010  
 218 1111001000011100  
 219 1110110011111000  
 220 1110001000010000  
 221 1101111110010000  
 222 1110011110010000  
 223 1110110001100110  
 224 1110101001101111  
 225 1110100011110001  
 226 1110101100001011  
 227 1110110100100100  
 228 1110100001100111  
 229 1101111100110110  
 230 1101101100111101  
 231 1101110100000111  
 232 1110000001010001  
 233 1110001111000000  
 234 1110010010110000  
 235 1110001110111111  
 236 1110001000010100  
 237 1101111011010110  
 238 1101111000010111  
 239 1110000001000010  
 240 1101111110101000  
 241 1101110000110010  
 242 1101100110101001  
 243 1101110010010001  
 244 1110001111001000  
 245 1110010100001011  
 246 1110000011101000  
 247 1110001001011100  
 248 1110100110000001  
 249 1110111011010110  
 250 1110111011101111  
 251 1110111001011000  
 252 1111000010101101  
 253 1111001001110010  
 254 1110111110011111  
 255 1110101001000010  
 256 1110101100101011  
 257 1110111011110111  
 258 1110110010111010

259	1110101010010101
260	1110110000100100
261	1110111001001001
262	1111000001001110
263	1111010011100000

---