Lab 5 Report:
Pipelined 8 Point FFT Algorithm using Butterfly Structures

Athokshay Ashok

Partner: Siegfred Madeghe

Date: 5/4/2020

EE 26 - Digital Logic Systems

Lab Section: Monday 4:30 pm - 6:30 pm

Tufts University School of Engineering (EECS)

Introduction

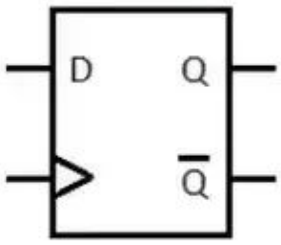
In Lab 4, we designed a non-pipelined 8-point FFT that uses 4 butterfly structures at each stage in parallel. Since there are not enough DSP slices available on the board to carry out all the floating point arithmetic, we modified the slice usage and included IPs for floating point adder/subtractor and multipliers.

In this lab, we designed a pipelined version of the 8-point FFT by using D-latches to connect the inputs and outputs of the different stages. We then implemented a ROM functionality that stores various sets of input data that can be used on the FFT. Finally, we displayed the results on the Nexys4 board with switches to control which input set to use from ROM, which output number to display, and which component (real or imaginary) to show.

Experimental Setup

To begin with, we created a new project and copied over the fft_butterfly file, the complex record package file, and the floating point IPs with medium DSP slice usage as was done in Lab 4. We also copied over the ROM file provided on Canvas. Since the ROM component returns a mem type, which is an array of 8 complex numbers, we defined this type in the complex package file. In Lab 4, we used individual signals for the inputs and outputs of the FFT (i.e. x0, x1, …, x7; y0, y1, … y7). After defining the mem type, we grouped the input and output signals into mem type arrays, which made the implementation cleaner.

The next step was to make the D-latch, which is a data latch that holds a value like a register. It takes in as input a data bit D to hold and a clock and returns bits Q (state) and the complement of Q. The symbol and the entity declaration for the D-latch are shown below:




```
entity D_Latch is
    Port (
        clk: in std_logic;
        D: in complex;
        Q: out complex
    );
end D_Latch;
```

Figure 1: Symbol and Entity Declaration for the D-Latch

Latches with clock inputs can be programmed to trigger either on level-edge, rising-edge, or falling-edge. We made a rising-edge latch where Q takes on D as the clock goes from 0 to 1.

clk	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

```
process(clk) begin
    if(rising_edge(clk)) then
        Q <= D;
    end if;
end process;
```

Figure 2: D-Latch Characteristic Table and VHDL Implementation

A pipeline is a process where each stage stores its outputs and passes it onto the next stage as soon as they are available rather than waiting for all the calculations of the first stage to be finished before moving onto the second. In order to do this, we need separate signals (of type mem) for the inputs and outputs of each stage:
Stage 1 → Input: input signal, Output: stage1_out
Stage 2 → Input: stage2_in, Output: stage2_out
Stage 3 → Input: stage3_in, Output: output signal

It is evident that stage1_out = stage2_in, and stage2_out = stage3_in. However, we store the outputs of the butterflies in the stageX_out signals and pass the values onto the D-latches. On the rising edge of the clock, these values are then passed into the stageX_in signals. The code for this is shown below:

```
stage1_X00: fft_butterfly port map (A=> input(0), B=> input(4), W=> W0, clk=> clk, A_comp=> stage1_out(0), B_comp=> stage1_out(4));-- 0, 4
stage1_X01: fft_butterfly port map (A=> input(1), B=> input(5), W=> W0, clk=> clk, A_comp=> stage1_out(2), B_comp=> stage1_out(6));-- 1, 5
stage1_X10: fft_butterfly port map (A=> input(2), B=> input(6), W=> W0, clk=> clk, A_comp=> stage1_out(1), B_comp=> stage1_out(5));-- 2, 6
stage1_X11: fft_butterfly port map (A=> input(3), B=> input(7), W=> W0, clk=> clk, A_comp=> stage1_out(3), B_comp=> stage1_out(7));-- 3, 7

stage1: for i in 0 to 7 generate
          d1:D_Latch port map (clk, stage1_out(i), stage2_in(i));
       end generate stage1;

stage2_0X0: fft_butterfly port map (A=> stage2_in(0), B=> stage2_in(1), W=> W0, clk=> clk, A_comp=> stage2_out(0), B_comp=> stage2_out(4));--
stage2_0X1: fft_butterfly port map (A=> stage2_in(2), B=> stage2_in(3), W=> W0, clk=> clk, A_comp=> stage2_out(1), B_comp=> stage2_out(5));--
stage2_1X0: fft_butterfly port map (A=> stage2_in(4), B=> stage2_in(5), W=> W2, clk=> clk, A_comp=> stage2_out(2), B_comp=> stage2_out(6));--
stage2_1X1: fft_butterfly port map (A=> stage2_in(6), B=> stage2_in(7), W=> W2, clk=> clk, A_comp=> stage2_out(3), B_comp=> stage2_out(7));--

stage2: for j in 0 to 7 generate
          d2:D_Latch port map (clk, stage2_out(j), stage3_in(j));
       end generate stage2;

stage3_00X: fft_butterfly port map (A=> stage3_in(0), B=> stage3_in(1), W=> W0, clk=> clk, A_comp=> output(0), B_comp=> output(4));-- 0, 1
stage3_01X: fft_butterfly port map (A=> stage3_in(2), B=> stage3_in(3), W=> W1, clk=> clk, A_comp=> output(7), B_comp=> output(3));-- 2, 3
stage3_10X: fft_butterfly port map (A=> stage3_in(4), B=> stage3_in(5), W=> W2, clk=> clk, A_comp=> output(6), B_comp=> output(2));-- 4, 5
stage3_11X: fft_butterfly port map (A=> stage3_in(6), B=> stage3_in(7), W=> W3, clk=> clk, A_comp=> output(5), B_comp=> output(1));-- 6, 7
```

Figure 3: Pipeline Process Between the Three FFT Stages

Here, we see the computations made at each stage, with the inputs being taken from the stageX_in signals and outputs being stored to the stageX_out signals. Between the stages, we use 8 latches for each of the values with a generate statement.

Using this approach involves assigning values to each complex number within the 8 point FFT component file itself. However, with a ROM, we can store multiple sets of input points and call them as needed. The provided ROM implementation is an 8x8 memory block that takes as input a 3 bit address and returns a mem type array with 8 complex numbers. Within this file, we defined the input points at each of the 64 locations. As a reference, we stored the sample input provided in Lab 4 at memory location 0 (this data set is shown in Table 1 below). We also created another data set by flipping the real and imaginary parts of each data complex number from the previous set and stored this at memory location 2 (shown in Table 2 below). To display the results to the 7-segment display on the Nexys 4 board, we used the file from Lab 4 that integrated seven_seg, decoder, and the eight_point_fft components and displayed the chosen number based on a 3-bit position input (that selects which of the 8 output numbers to display) and a 1-bit real/imaginary input using switches. However, now we added another 3-bit location input that selected which memory location to fetch the data from the ROM. The process of getting the data, performing the calculations, and displaying the results is shown in the code snippet below.

```
rom1: ROM port map(address, input);
eight_point: eight_point_fft_pipeline port map (input, clock, output);
seven_seg1: seven_seg port map (inp, clock, dig, position);
```

Figure 4: Port Maps for the ROM, 8-Point FFT, and Seven Segment Display

Results and Discussion

Once the D-Latch integration was completed, we wrote a testbench to verify that the pipeline process worked. The testbench file and input points were the same used in Lab 4, with slight modifications made to change the inputs from individual signals to an array as was done with the 8-point FFT. The results of the behavioral simulation are shown below.

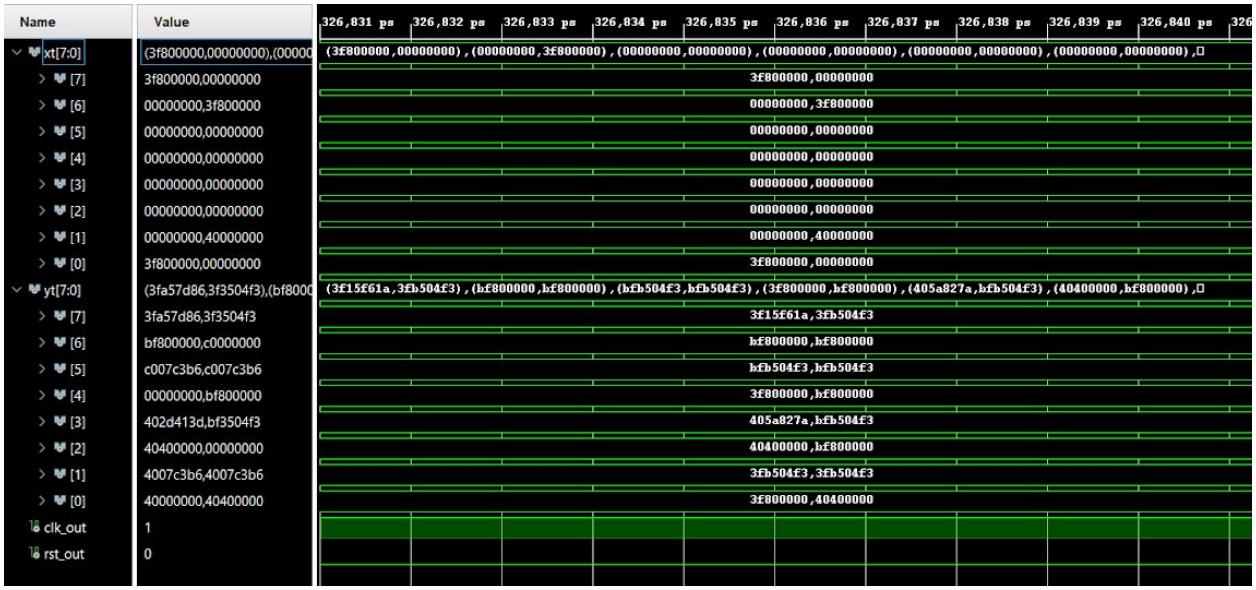


Figure 5: Pipelined 8-Point FFT Behavioral Simulation

Although a timing simulation gave us undefined outputs due to the floating point IPs, we could see from the behavioral simulation how and when each output value was calculated in the pipeline process. All 8 outputs were computed by 326,000 ps, which suggests that one output was being calculated approximately every 40,000 ps. The behavioral simulation from Lab 4 indicated that the outputs took around 1,000,000 ps, making the pipeline process around 3 times faster.

When displaying the results to the board, we used 3 sets of switches on the board (from left to right:

1) Switches 0-2: select the output point
2) Switches 4-6: select the memory location for the input data set from ROM
3) Switch 15: select real/imaginary

The data sets contained in the ROM are shown below.

Input in decimal	1,0	0,2	0,0	0,0	0,0	0,0	0,1	1,0
Output in decimal	2.000000, 3.000000	2.121320, 2.121320	3.000000, 0.000000	2.707107, -0.707107	0.000000, -1.000000	-2.121320, -2.121320	-1.000000, -2.000000	1.292893, 0.707107
Output in hex	40000000, 40400000	4007c3b6, 4007c3b6	40400000, 00000000	402d413d ,bf3504f3	00000000 ,bf800000	C007c3b6, c007c3b6	Bf800000, c0000000	3fa57d86, 3f3504f3

Table 1: Test Case Inputs Stored in mem_all(0)

Input in decimal	0,1	2,0	0,0	0,0	0,0	0,0	1,0	0,1
Output in decimal	3.000000, 2.000000	0.707107, 1.292893	-2.000000, -1.000000	-2.121320, -2.121320	-1.000000 ,0.000000	-0.707107 ,2.707107	0.000000, 3.000000	2.121320, 2.121320
Output in hex	40400000, 40000000	3f3504f7, 3fa57d85	c0000000, bf800000	c007c3b5, c007c3b5	bf800000, 00000000	bf3504f7, 402d413e	00000000, 40400000	4007c3b5, 4007c3b5

Table 2: Test Case Inputs Stored in mem_all(2)

The first data set was verified to be working accurately compared to the expected results. To find out the expected results of the second data set, which we created, we used an online FFT calculator and converted the decimals to hex values using an online single point precision calculator. We noticed that for a few numbers, there were slight discrepancies with the last hex position, which occurs due to rounding errors.

Conclusion

In this lab, we were successfully able to pipeline the FFT process by connecting the stages with D-latches and decreasing the time complexity by a factor of 3. We also used a ROM to store multiple data inputs and call them as needed using switches on the Nexys4 board. In the future, we would like to explore the idea of implementing a 16-point FFT by computing 2 sets of 8-point FFTs in parallel and displaying the results to the board without running out of DSP slices. We would also like to be able to do a detailed timing summary (if possible) to see exactly how much faster the pipeline process is.

Appendix

D-Latch

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

library IEEE_Proposed;
use IEEE_Proposed.float_pkg.all;
use work.complex_record.all;

entity D_Latch is
    Port (
        clk: in std_logic;
        D: in complex;
        Q: out complex
    );
end D_Latch;

architecture Behavioral of D_Latch is

begin

process(clk) begin
    if(rising_edge(clk)) then
        Q <= D;
    end if;
end process;


end Behavioral;
```

ROM

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;
library ieee_proposed;
use ieee_proposed.float_pkg.all;
use work.complex_record.all;

-- mem is a type defined as "array(7 downto 0) of complex" in the package file.

entity ROM is
   port (
      address : in std_logic_vector(2 downto 0);
      data : out mem
      );
end entity ROM;

architecture behavioral of ROM is

type mem_all is array (0 to 63) of complex;

constant my_rom : mem_all := (    -- floating-point numbers can be defined in decimal or IEEE
754 hex representation.
0  => (r => std_logic_vector(to_float(1, 8, 23)), i => std_logic_vector(to_float(0, 8, 23))),
1  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(2,8, 23))),
2  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
3  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
4  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
5  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
6  => (r => std_logic_vector(to_float(0, 8, 23)), i => std_logic_vector(to_float(1, 8, 23))),
7  => (r => std_logic_vector(to_float(1, 8, 23)), i => std_logic_vector(to_float(0, 8, 23))),

16  => (r => std_logic_vector(to_float(0, 8, 23)), i => std_logic_vector(to_float(1, 8, 23))),
17  => (r => std_logic_vector(to_float(2,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
18  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
19  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
20  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
21  => (r => std_logic_vector(to_float(0,8, 23)), i => std_logic_vector(to_float(0,8, 23))),
22  => (r => std_logic_vector(to_float(1, 8, 23)), i => std_logic_vector(to_float(0, 8, 23))),
23  => (r => std_logic_vector(to_float(0, 8, 23)), i => std_logic_vector(to_float(1, 8, 23))),

others => (r => x"00000000", i => x"00000000"));
```

```
begin

g1: for i in 0 to 7 generate
   data(i) <= my_rom(to_integer(unsigned(address & std_logic_vector(to_unsigned(i,3)))));
end generate g1;

end architecture behavioral;
```

Eight Point FFT Pipeline

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

library IEEE_Proposed;
use IEEE_Proposed.float_pkg.all;
use work.complex_record.all;

entity eight_point_fft_pipeline is
  Port (
     input : in mem;
     clk : in std_logic;
     output: out mem
  );
end eight_point_fft_pipeline;

architecture Behavioral of eight_point_fft_pipeline is

component fft_butterfly is
  Port (
    A : in complex;
    B : in complex;
    W : in complex;
    clk : std_logic;
    A_comp : out complex;
    B_comp : out complex
```

```
  );
end component fft_butterfly;

component D_Latch is
    Port (
        clk: in std_logic;
        D: in complex;
        Q: out complex
    );
end component D_latch;


signal W0, W1, W2, W3 : complex;

signal stage1_out, stage2_in, stage2_out, stage3_in : mem;


begin

W0.r <= std_logic_vector(to_float(1, 8, 23));
W0.i <= std_logic_vector(to_float(0, 8, 23));

W1.r <= std_logic_vector(to_float(0.70710678118, 8, 23));
W1.i <= std_logic_vector(to_float(0.70710678118, 8, 23));

W2.r <= std_logic_vector(to_float(0, 8, 23));
W2.i <= std_logic_vector(to_float(1, 8, 23));

W3.r <= std_logic_vector(to_float(-0.70710678118, 8, 23));
W3.i <= std_logic_vector(to_float(0.70710678118, 8, 23));


stage1_X00: fft_butterfly port map (A=> input(0), B=> input(4), W=> W0, clk=> clk,
A_comp=> stage1_out(0), B_comp=> stage1_out(4));-- 0, 4
stage1_X01: fft_butterfly port map (A=> input(1), B=> input(5), W=> W0, clk=> clk,
A_comp=> stage1_out(2), B_comp=> stage1_out(6));-- 1, 5
stage1_X10: fft_butterfly port map (A=> input(2), B=> input(6), W=> W0, clk=> clk,
A_comp=> stage1_out(1), B_comp=> stage1_out(5));-- 2, 6
```

```
stage1_X11: fft_butterfly port map (A=> input(3), B=> input(7), W=> W0, clk=> clk,
A_comp=> stage1_out(3), B_comp=> stage1_out(7));-- 3, 7

stage1: for i in 0 to 7 generate
        d1:D_Latch port map (clk, stage1_out(i), stage2_in(i));
    end generate stage1;

stage2_0X0: fft_butterfly port map (A=> stage2_in(0), B=> stage2_in(1), W=> W0, clk=> clk,
A_comp=> stage2_out(0), B_comp=> stage2_out(4));-- 0, 2
stage2_0X1: fft_butterfly port map (A=> stage2_in(2), B=> stage2_in(3), W=> W0, clk=> clk,
A_comp=> stage2_out(1), B_comp=> stage2_out(5));-- 1, 3
stage2_1X0: fft_butterfly port map (A=> stage2_in(4), B=> stage2_in(5), W=> W2, clk=> clk,
A_comp=> stage2_out(2), B_comp=> stage2_out(6));-- 4, 6
stage2_1X1: fft_butterfly port map (A=> stage2_in(6), B=> stage2_in(7), W=> W2, clk=> clk,
A_comp=> stage2_out(3), B_comp=> stage2_out(7));-- 5, 7

stage2: for j in 0 to 7 generate
        d2:D_Latch port map (clk, stage2_out(j), stage3_in(j));
    end generate stage2;

stage3_00X: fft_butterfly port map (A=> stage3_in(0), B=> stage3_in(1), W=> W0, clk=> clk,
A_comp=> output(0), B_comp=> output(4));-- 0, 1
stage3_01X: fft_butterfly port map (A=> stage3_in(2), B=> stage3_in(3), W=> W1, clk=> clk,
A_comp=> output(7), B_comp=> output(3));-- 2, 3
stage3_10X: fft_butterfly port map (A=> stage3_in(4), B=> stage3_in(5), W=> W2, clk=> clk,
A_comp=> output(6), B_comp=> output(2));-- 4, 5
stage3_11X: fft_butterfly port map (A=> stage3_in(6), B=> stage3_in(7), W=> W3, clk=> clk,
A_comp=> output(5), B_comp=> output(1));-- 6, 7


end Behavioral;
```

Eight Point FFT Rom Display

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

```
library IEEE_Proposed;
use IEEE_Proposed.float_pkg.all;
use work.complex_record.all;

entity eight_point_fft_pipeline_rom_display is
  Port (
        index: in std_logic_vector(2 downto 0);
        i_r: in std_logic;
        address: in std_logic_vector(2 downto 0);
        clock: in std_logic;
        dig : out std_logic_vector(6 downto 0);
        position: out std_logic_vector(7 downto 0)
  );
end eight_point_fft_pipeline_rom_display;

architecture Behavioral of eight_point_fft_pipeline_rom_display is

component ROM is
    port (
        address : in std_logic_vector(2 downto 0);
        data : out mem
        );
end component ROM;

component eight_point_fft_pipeline is
  Port (
      input : in mem;
      clk : in std_logic;
      output: out mem
  );
end component eight_point_fft_pipeline;



component seven_seg is
          port (
                    input : in std_logic_vector (31 downto 0);
         clk : in std_logic;
         digit : out std_logic_vector(6 downto 0);
```

```
        pos : out std_logic_vector(7 downto 0)
          );
end component seven_seg;

signal input, output: mem;
signal inp: std_logic_vector(31 downto 0);


begin

rom1: ROM port map(address, input);
eight_point: eight_point_fft_pipeline port map (input, clock, output);
seven_seg1: seven_seg port map (inp, clock, dig, position);

process(index, i_r, inp, clock, address, input) begin

if(index = "000" and i_r = '0') then
   inp <= output(0).r;
elsif(index = "000" and i_r = '1') then
   inp <= output(0).i;

elsif(index = "001" and i_r = '0') then
   inp <= output(1).r;
elsif(index = "001" and i_r = '1') then
   inp <= output(1).i;

elsif(index = "010" and i_r = '0') then
   inp <= output(2).r;
elsif(index = "010" and i_r = '1') then
   inp <= output(2).i;

elsif(index = "011" and i_r = '0') then
   inp <= output(3).r;
elsif(index = "011" and i_r = '1') then
   inp <= output(3).i;

elsif(index = "100" and i_r = '0') then
   inp <= output(4).r;
elsif(index = "100" and i_r = '1') then
```

```
    inp <= output(4).i;

elsif(index = "101" and i_r = '0') then
    inp <= output(5).r;
elsif(index = "101" and i_r = '1') then
    inp <= output(5).i;

elsif(index = "110" and i_r = '0') then
    inp <= output(6).r;
elsif(index = "110" and i_r = '1') then
    inp <= output(6).i;

elsif(index = "111" and i_r = '0') then
    inp <= output(7).r;
elsif(index = "111" and i_r = '1') then
    inp <= output(7).i;

else
    inp <= "11110000111100001111000011110000";

end if;

end process;


end Behavioral;
```