Final report MAPD part 1

Nicola Dainese, Matteo Guida, Stefano Mancone, Francesco Vidaich

29 October 2019

1 Code Development

This is the final report for the course in management and analysis of physical datasets. The aim of this project is to create a particular FPGA architecture that do various operations. The elements of the architecture are reported in Figure 1 and some modular components can be seen. These components to work properly depend one another in the sense that the input data of some module depends on the output result of the others. The core components are:

- The SPI master, that has to read the first 2 addresses in the flash memory, concatenate them and write them in the first address of the DPRAM;
- Square wave generator (implemented in the *square_wave.vhd*) has to produce a square wave given the values of its period and duty cycle. The values of the two states are fixed and equal to -1024 and $+1024$. These signed values are represented in *std_logic_vector* variables to be compatible with the other components. From this generator 1024 samples are picked and written in the DPRAM.
- The Finite Input Response filter has to read the addresses in the DPRAM written by the square generator, apply to them the fir filter and write the result back into the DPRAM.
- The final element of the architecture is a multiplexer that has the task of regulating access to the DPRAM from the square wave generator and the fir filter, since they both have to access to the same addresses of memory.

1.1 VIO policy

We choose to implement a single VIO core to manage the reset button and the start signals for the flash spi master, the square wave generator and the master fir. This is implemented in the top_level.vhd behavioural architecture as follows:

```
vio_start : vio_0
  PORT MAP (
    clk => clk_base_xc7a_i,
    probe_out0(0) => vio_rst,
    probe_out1(0) => start_spi,
    probe_out2(0) => start_sq,
    probe_out3(0) => start_fir
  );
```

1.2 Mux 21

Since the multiplexer of two signals is very short to implement, we used directly a process to do that, instead of writing a component. We choose to give priority to the square generator over the fir in accessing the square wave registers because it makes sense from a chronological point of view. Hence if the start of the square wave generator is high, its address is selected, otherwise it is selected the one of the FIR.

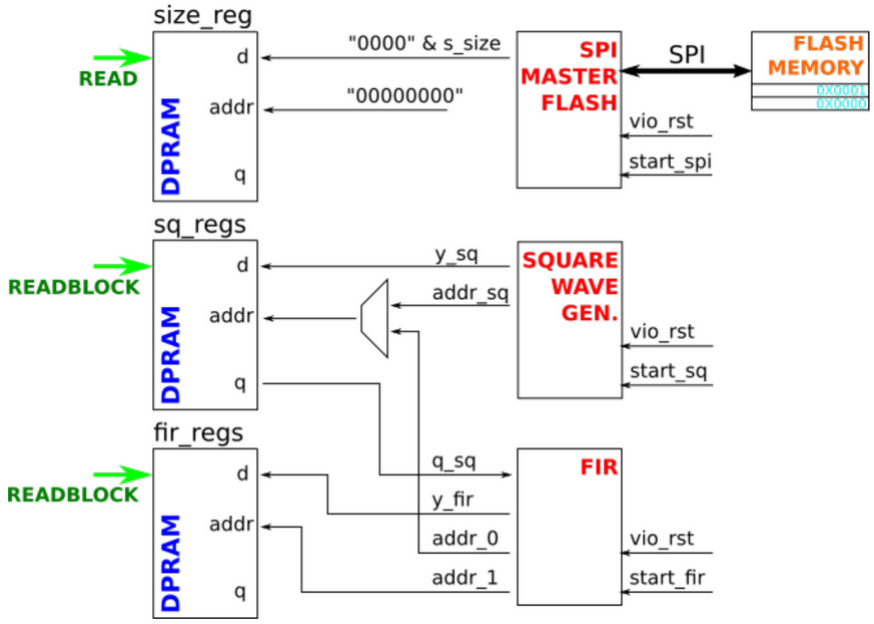


Figure 1: Architecture of the FPGA program

```
mux_select : process(clk_base_xc7a_i)
begin
if start_sq = '1' then
    mux_sel <= '0';
else
    mux_sel <= '1';
end if;
end process;

mux_addr <= addr_sq when mux_sel = '0' else addr_fir_0;
```

1.3 FSM of flash spi master

To implement the concatenation of the first two registers of the flash memory, we modified the case of the s_buildword state in the main process. Basically we introduced two variables, word1 and word2, that store the value of the signal s_rxd coming from the spi_master and contain the values read in the registers of the flash memory. Once the two signals are retrieved, they are concatenated in the right order and written in the size register DPRAM. Note that $N\_BYTES = 2$ in this project.

```
when s_buildword =>
    if cnt_o < 3 then
       cnt_o := cnt_o + 1;
       -- custom part
       if bcnt = N_BYTES - 1 then
       -- variable word1 : std_logic_vector(7 downto 0);
          word1 := s_rxd;
       elsif bcnt =  N_BYTES - 2 then
       -- variable word2 : std_logic_vector(7 downto 0);
          word2 := s_rxd;
       -- signal s_word  : std_logic_vector(N_BYTES*RXBITS-1 downto 0);
          s_word <= word1&word2;
```

```
        we_out <= '1';
        we_s_debug <= '1';
    end if;
    -- end custom part
else
        if bcnt = 0 then
        bcnt  := N_BYTES - 1;
        state <= s_stop;
        ready_s <= '1';
    else
        bcnt  := bcnt - 1;
        state <= s_getbyte;
        ready_s <= '0';
    end if;
    we_out <= '0';
    we_s_debug <= '0';
    cnt_o := 0;
end if;
```

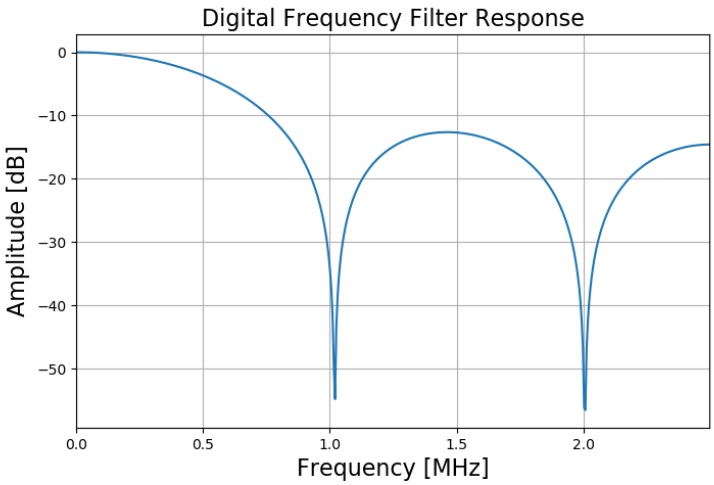
1.4 FIR architecture

We split the FIR architecture in two modules, the FIR itself and a master FIR that implements the communication with the DPRAM registers.

1.4.1 FIR

The FIR is a 5-tap filter, which response in frequency is reported in the graph below. The coefficients are computed using the coeff.py script, provided during the course.

In the 5-tap case those are: 0.19335315, 0.20330353, 0.20668665, 0.20330353, 0.19335315. To represent them in VHDL we multiplied them by 1024 (this term will be accounted for at the end of the FIR) and then truncated the decimal part. In this way we obtain: 198, 208, 212, 208, 198.



1.4.2 FIR master FSM

We implemented a FSM with 3 states that works as follows:

1. If reset signal is high, enters in idle and sets all signals and variables to default values;
2. When start signal is high and reset is low goes into read state;
3. In read and write state we implemented a clock called fir_clock with twice the period of the input clock, in order to slow down the fir; this is used to read the output of the fir when fir_clock is high and to write it to the fir registers when the fir_clock is low. It is in fact the equivalent of reading on the rising edge and writing on the falling one but it is a safer procedure.

```
type state is (s_idle, s_read, s_write);

p_fir: process(clk, rst) is
    --variable tcnt: integer;
    begin
        if rst = '1' then
            --tcnt := 0;
            samples <= 0;
            we <= '0';
            state_fsm <= s_idle;
            fir_clk <= '0';
            first_run <= '1';
        elsif rising_edge(clk) then
            case state_fsm is
            when s_idle =>
                we <= '0';
                if (start = '1') and (first_run = '1') then
                    first_run <= '0';
                    addr_0 <= std_logic_vector(to_unsigned(samples, addr_0'length));
                    state_fsm <= s_read;
                else
                    state_fsm <= s_idle;
                end if;
            when s_read =>
                we <= '0';
                fir_clk <= '1';
                state_fsm <= s_write;
            when s_write =>
                we <= '1';
                addr_1 <= std_logic_vector(to_unsigned(samples, addr_0'length));
                fir_clk <= '0';
                samples <= samples + 1;
                if samples = 1024 then
                    samples <= 0;
                    state_fsm <= s_idle;
                else
                    addr_0 <= std_logic_vector(to_unsigned(samples, addr_0'length));
                    state_fsm <= s_read;
                end if;
            when others =>
                state_fsm <= s_idle;
            end case;
        end if;
    end process;
```

1.5 State s_low of square wave generator's FSM

The job of the square wave generator is to produce 1024 samples to save into the DPRAM. After the system is reset, the generator is in the idle state until the user enables it through the VIO. This action will start the square wave generator (changing its state to *s_up*) and it will set to 1 the variable *gen_is_on*. At each clock cycle, a sample of the wave is stored at a new DPRAM address. More specifically, this address starts at 0x0000 and increases at each clock cycle until all 1024 samples are stored. To do so, the following piece of code is run at each rising edge of the clock:

```
if gen_is_on = '1' then
    address <= std_logic_vector(to_unsigned(samples, address'length));
    if samples = SAMPLE_N then
        samples <= 0;
        gen_is_on <= '0';
    else
        samples <= samples + 1;
    end if;
end if;
```

It is possible to see that when all samples are stored, the *gen_is_on* variable is reset to 0. When this happens, the generator has to stop cycling between *s_up* and *s_down* states, its output will be reset to 0 and its state to *s_idle*. This check is done during the *s_down* state:

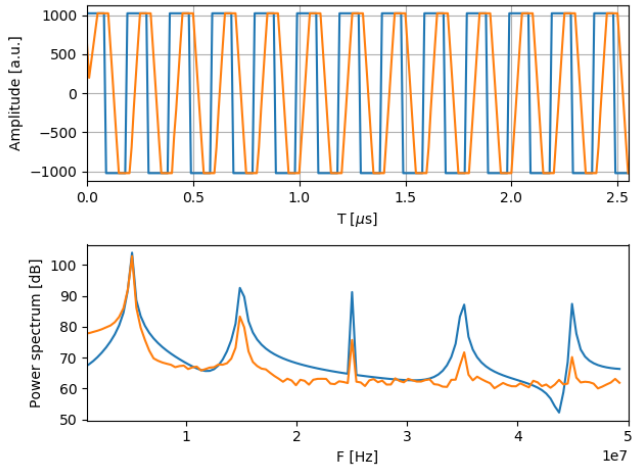
```
when s_down =>
    if tcnt = down_cycles-1 then
        tcnt := 0;
        if gen_is_on = '1' then
            y <= std_logic_vector(to_signed(1024 , y'length));
            state_fsm <= s_up;
        else
            y <= std_logic_vector(to_signed(0, y'length));
            state_fsm <= s_idle;
        end if;
    else
        tcnt := tcnt + 1;
    end if;
```

2 Result

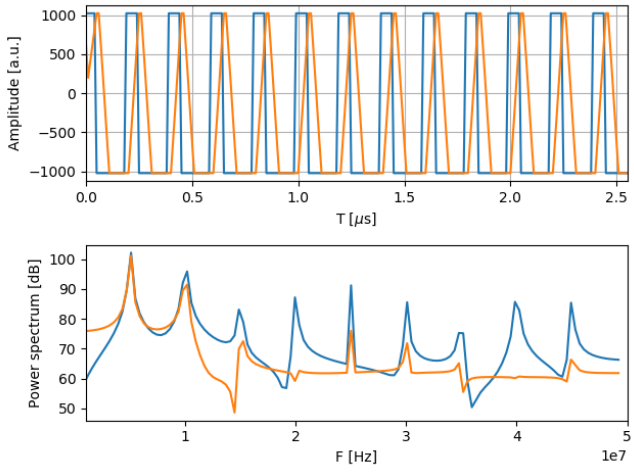
Here we present the final results obtained by our program. We run the program with 3 different combinations of parameters for the square wave generator:

- PERIOD = 20, DUTY_CYC = 50
- PERIOD = 10, DUTY_CYC = 50
- PERIOD = 20, DUTY_CYC = 30

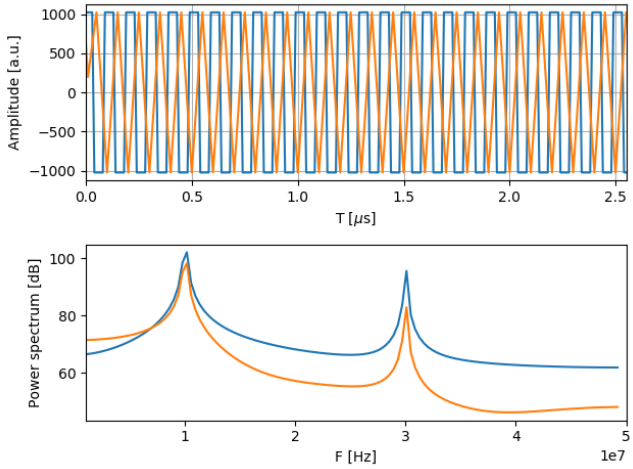
We can see that in the time domain the square wave transitions are smoothed to linear fronts and in the frequency domain the power spectrum at higher frequencies is suppressed with respect to that of lower frequencies.



(a) $PERIOD = 20$ clock cycles, $DUTY\_CYCLE = 50$



(b) $PERIOD = 20$ clock cycles, $DUTY\_CYCLE = 30$



(c) $PERIOD = 10$ clock cycles, $DUTY\_CYCLE = 50$