1. DSP Processor:

Table 1: Filter gain vs. frequency

Frequency	C5x Gain (output / input)	C6x Gain (output / input)
1kHz	2.40	1.32
1.5kHz	2.12	1.12
2kHz	1.46	0.59
2.5kHz	0.70	0.12

1) Comment on how well the filters work

A: It seems the filter implemented on floating-point DSP C6x chip is much better than that on the fixed-point chip. Evidence is that the floating-point version of filter has a sharper drop of gain from 1.12 at 1.5kHz to 0.59 at 2kHz and finally 0.12 at 2.5kHz. On the other side, gains of the filter on C5x chip are relatively high at frequency 2kHz and 2.5kHz which result in a larger cutoff frequency.

Table 2: DSP chip filtering cycle counts

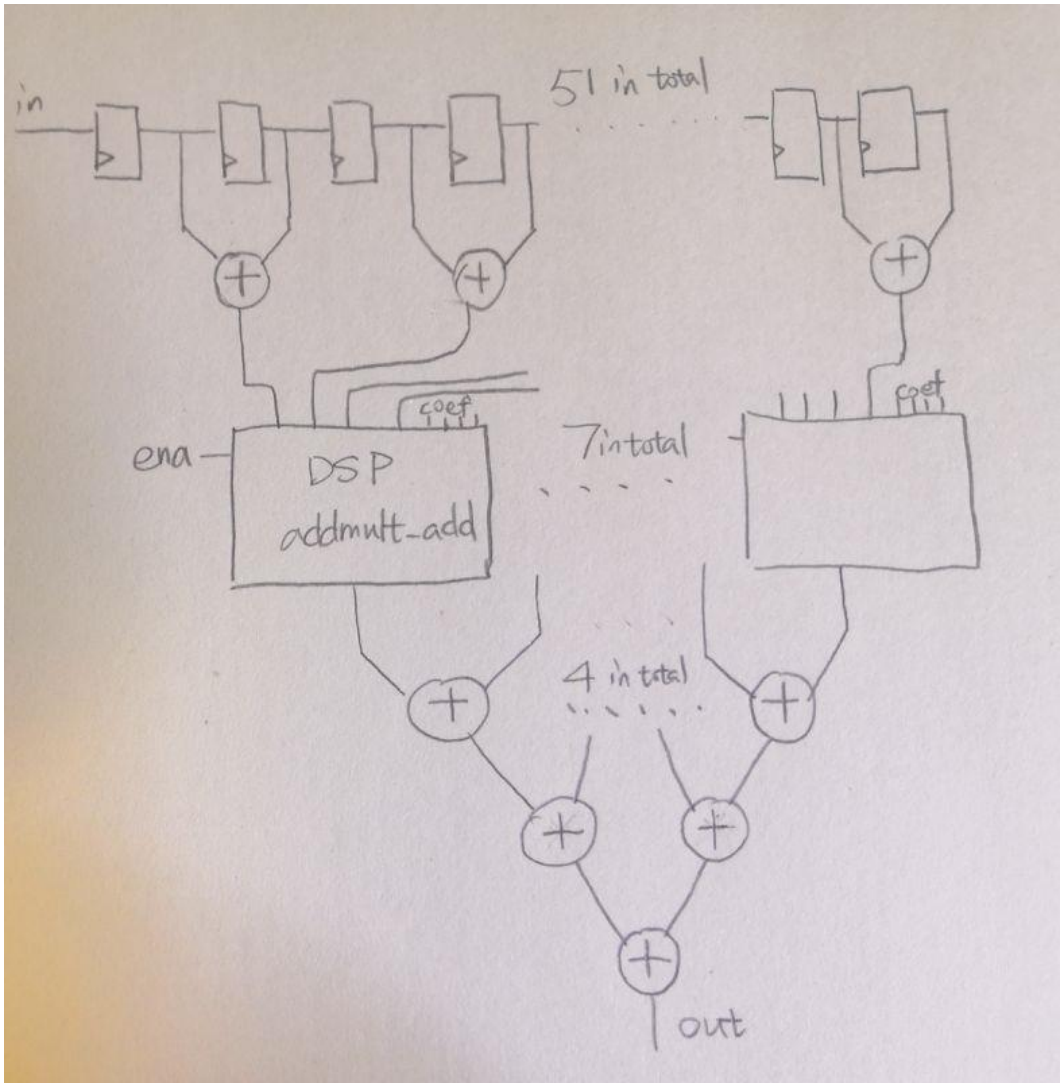
Filter implementation	C5x Cycle count	C6x Cycle count
C (w/o optimization)	2503	2041
C (with compiler optimization)	639	909
Assembly	702	138

2) Describe how the FIR filter C code given in the lab works. Comment in particular on exactly how the loops work, and why the various shift instructions are needed.

A: The FIR-filter function is called every time a new input is received. Static variable j points to the value in data buffer that should be replaced with new input data and be multiplied with the first filter coefficient in the next call. The for loop goes through the coefficient and value array and increments i and j accordingly. Multiplying and accumulation is done through the loop. Variable j basically increments by one with every invocation of the function unless reaches the end and keeps pointing to the newest or previously stored input data. Then a new input is stored in the position j points to. The multiplication result of one value and one coefficient is shifted right 5 bits to maintain precision as much as possible while avoiding possible overflow. The other 10 bits shifted before returning the result is to move the valid data bits to the lowest 16 bits and return in datatype short.

2. FPGA Implementation:

1) Include a description of and block diagram for your FPGA design in your report.



A: Above is the block diagram of the design. Input samples that share the same coefficients are added together and outputs of addition are connected to the inputs of 7 instantiations of 4-input DSP function *altmult_add*. Then the 7 outputs of the functions are put into a 7-input adder tree as shown in the bottom. Pipeline registers are put in every possible position. Valid and ready signals will stall the whole pipeline if any of the two become level 0. The 7-input adder tree basically has 3 levels with each level summing up input pairs and reducing numbers roughly by half. One thing to notice is that coefficients are read in through $readmemb as binary data generated from the given data.

2) Include data on resource utilization, operating frequency and power consumption of your FPGA design. You can use the same power estimation and flow as in assignment 1.
3) Include a listing of the testbench you use for power estimation (should be different than the verification testbench you are given).

	Optimized Circuit
Resource usage (1 Computation Circuit)	1712 LUT/FF pairs & 28 DSP 18-bit elements
Operating Freq.	510.2MHz (limited)
Critical path	LE-based adder (first level of adder tree)
Cycles per Valid Output	1
Maximum # of Copies / Device (limited by)	46 (DSP)
Max. Throughput per Device (Computation/s)	$2.35*10^{10}$
Dynamic Power of One Computation Circuit @ 50MHz	25.49mW
Maximum Computations / Joule (Full Device)	$1.83*10^{9}$

A: The testbench is modified to generate random inputs rather than 20000 as in the original testbench. To avoid the warmup time and delta function, the period of time used to estimate power starts from where the step function started, which is i>=120 as shown in the listing below.

Listing 1 Modified part of testbench

```
// Create known good input: a delta function, then a step function
for (int i = 0; i < INPUTS_TO_SIM; i++) begin
   if (i == 60 || i >= 120)
      /* impulse at i=60, step at i=120+ */
      inwave[i] = 16'd20000;      // functional simulation inputs
      //inwave[i] = $urandom(i); // feed random inputs to do realistic power estimation
   else
      /* 0 everywhere else */
      inwave[i] = 16'd0;
end
```

4) Include a source code listing of your FPGA design in your report. It should be well commented and use good (self-documenting) signal names. Using generate statements and/or a hierarchy of modules where appropriate to avoid highly repetitive code is also a good practice.

Listing 2 Source code of FPGA design

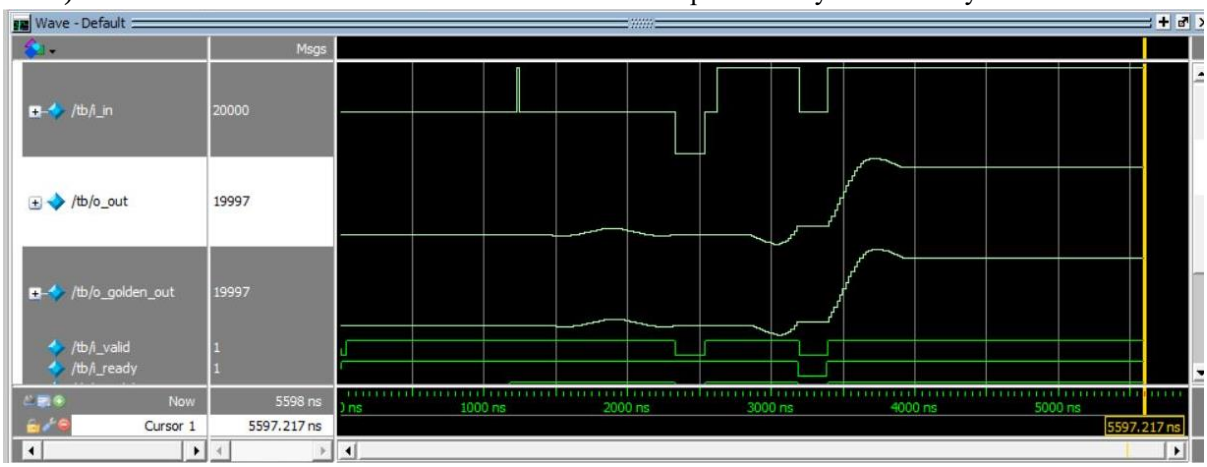
```
// top module of ECE1756 lab2  by Zhiyuan Gu (1004920400)
module lab2 ( i_in, o_out, i_valid, o_valid, i_ready, o_ready, clk, reset) /* synthesis multstyle = "dsp" */;
// All * operations in my_module will now use a DSP block if possible */
   // set local parameters
   localparam DSP_num = 7;       // number of DSP-based mac modules
   localparam input_width = 16;  // input signal width
   localparam coef_width = 16;   // coefficient width
   localparam output_width = 35; // output width of mac modules
   localparam filter_order = 50; // order of FIR filter
   // input and output signals
   input signed [input_width-1:0] i_in;
   output signed [input_width-1:0] o_out;
   input i_valid;
   output o_valid;
   input i_ready;
   output o_ready;
   input clk;
   input reset;

   integer i;

   wire enable;                                          // enable signal for sequential logic and passing into submodules
   wire signed [output_width-1:0] mac_res [DSP_num-1:0];       // 4-input DSP accumulation of multipliers result
   wire signed [output_width+2:0] adder_tree_res;              // adder tree result

   reg signed [coef_width-1:0] coef [filter_order>>1:0];       // to pass 26 coefficients to DSP inputs
   reg signed [coef_width-1:0] b [filter_order>>1:0];          // 26 coefficients read from memory initialization file
   reg signed [input_width-1:0] in_sample [filter_order:0];    // samples of input
   wire signed [input_width:0] sample_out [filter_order>>1:0];    // sum of input sample pair which share the same coefficient
   reg signed [input_width:0] sample_out_reg [filter_order>>1:0];    // register the above signal
   reg [filter_order+8:0] valid_reg ;                          // register to propagate valid signal

   // read coefficients
   initial begin
         $readmemb("coef_out.mif", b);    // read binary coefficients from generated file
   end
```

```
    // pass coefficients to DSP inputs
    always@(*) begin
       for(i = 0; i <= filter_order>>1; i = i + 1) begin
          coef[i] = b[i];
       end
    end

    assign enable = i_ready & i_valid;                    // if data not valid or downstream module not ready, stall pipeline
    assign o_ready = i_ready;                             // if downstream not ready, tell upstream not ready
    assign o_valid = i_valid & (&valid_reg) & i_ready;    // output is valid only when all the previous 51 inputs are valid and downstream module
    assign o_out = adder_tree_res[2*(input_width-1):input_width-1];  // assign output signal to be adder tree output and adjust precision

    // registers
    always@(posedge clk) begin
       if (reset) begin
          for(i = 0; i < filter_order + 1; i = i + 1) begin
             in_sample[i] <= 0;
          end
          valid_reg <= 0;
       end
       else begin
          if(enable) begin
             valid_reg[filter_order+8:0] <= {valid_reg[filter_order+7:0], i_valid};
             in_sample[0] <= i_in;
             for (i = 0; i < filter_order; i = i + 1) begin
                in_sample[i+1] <= in_sample[i];
             end
             for (i = 0; i <= filter_order>>1; i = i + 1) begin
                sample_out_reg[i] <= sample_out[i];
             end
          end
          else begin  // remain the same state until the downstream module is ready
             for(i = 0; i < filter_order + 1; i = i + 1) begin
                in_sample[i] <= in_sample[i];
             end
             valid_reg <= valid_reg;
             for (i = 0; i <= filter_order>>1; i = i + 1) begin
                sample_out_reg[i] <= sample_out_reg[i];
             end
          end
       end

    end

    // generate 25 adder for adding input samples that share the same coefficient
    genvar gi;
    generate
       for(gi = 0; gi < filter_order>>1; gi = gi + 1) begin: adder_l1
          assign sample_out[gi] = in_sample[gi] + in_sample[filter_order - gi];
       end
    endgenerate
    assign sample_out[filter_order>>1] = in_sample[filter_order>>1];

    // generate 7 4-input dsp-based mac ip module, 7*4 = 28 > 26, the last two input pairs are set to (0,0)
    genvar gj;
    generate
       for(gj = 0; gj < DSP_num; gj = gj + 1) begin : mac1
          mult_add_dsp mult_add_dsp1(
          clk,
          sample_out_reg[gj*4+0],
          sample_out_reg[gj*4+1],
          (gj==DSP_num-1) ? 0 : sample_out_reg[gj*4+2],
          (gj==DSP_num-1) ? 0 : sample_out_reg[gj*4+3],
          coef[gj*4+0],
          coef[gj*4+1],
          (gj==DSP_num-1) ? 0 : coef[gj*4+2],
          (gj==DSP_num-1) ? 0 : coef[gj*4+3],
          enable,
          mac_res[gj]    );
       end
    endgenerate

    // instantiate adder tree
    in7_adder_tree adder_tree1 (mac_res[0],
                                mac_res[1],
                                mac_res[2],
                                mac_res[3],
                                mac_res[4],
                                mac_res[5],
                                mac_res[6],
                                clk, enable,
                                adder_tree_res);


endmodule
```

adder tree submodule

```
// 7-input adder tree module of ECE1756 lab2  by Zhiyuan Gu (1004920400)
module in7_adder_tree(A, B, C, D, E, F, G,
                      clk, enable, out);
   // set local parameters
   localparam output_width = 35;
   // input and output signals
   input [output_width-1:0] A, B, C, D, E, F, G;
   input clk, enable;
   output [output_width+2:0] out;

   reg   [output_width-1:0]    inreg [6:0];

   wire  [output_width:0]    sum1 [3:0];
   reg   [output_width:0]    sumreg1 [3:0];

   wire  [output_width+1:0]    sum2 [1:0];
   reg   [output_width+1:0]    sumreg2 [1:0];

   wire  [output_width+2:0]    sum3;
   reg   [output_width+2:0]    sumreg3;

   integer i;

   // Registers
   always @ (posedge clk)
      begin
         if(enable) begin
            inreg[0] = A;
            inreg[1] = B;
            inreg[2] = C;
            inreg[3] = D;
            inreg[4] = E;
            inreg[5] = F;
            inreg[6] = G;

            for(i = 0; i < 4; i = i + 1) begin
               sumreg1[i] <= sum1[i];
            end
            for(i = 0; i < 2; i = i + 1) begin
               sumreg2[i] <= sum2[i];
            end
            sumreg3 <= sum3;
         end
         else begin
            for(i = 0; i < 4; i = i + 1) begin
               sumreg1[i] <= sumreg1[i];
            end
            for(i = 0; i < 2; i = i + 1) begin
               sumreg2[i] <= sumreg2[i];
            end
            sumreg3 <= sumreg3;
            for(i = 0; i < 7; i = i + 1) begin
               inreg[i] <= inreg[i];
            end
         end
      end

   // 3 level of addition and one output assignments
   assign           sum1[0] = inreg[0] + inreg[1];
   assign           sum1[1] = inreg[2] + inreg[3];
   assign           sum1[2] = inreg[4] + inreg[5];
   assign           sum1[3] = inreg[6];

   assign           sum2[0] = sumreg1[0] + sumreg1[1];
   assign           sum2[1] = sumreg1[2] + sumreg1[3];

   assign           sum3 = sumreg2[0] + sumreg2[1];

   assign           out = sumreg3;

endmodule
```

5) Include screen shots of waveforms and testbench output to verify functionality.



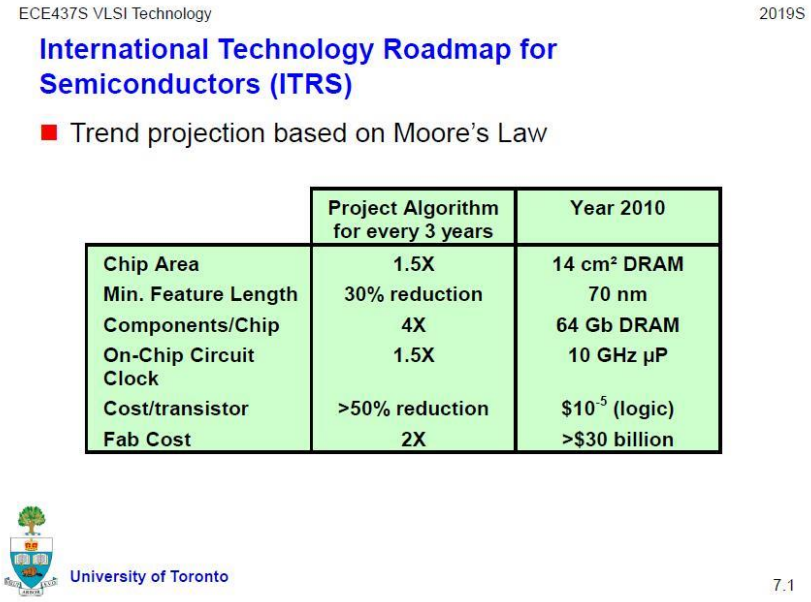
```
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5438000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5458000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5478000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5498000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5518000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5538000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5558000
# diff: 0.000000, rms: 0.000000, o_out: 19997.000000, golden: 19997.000000, at time:          5578000
# RMS Error: 0.000000
# Error is within 10 units (on a scale to 32,000) - great success!!
```

3. Efficiency Comparison:

Table 3: Throughput and efficiency of DSP and FPGA platforms

Device	Samples/s	Power	Samples/J	Samples/$
C5505 DSP	$1.88*10^5$	44mW	$4.27*10^6$	$2.09*10^4$
C674x DSP (Omap L138)	$3.30*10^6$	520mW	$6.35*10^6$	$1.65*10^5$
C5505 DSP scaled to 40 nm	$6.77*10^6$	759mW	$8.92*10^6$	$1.88*10^5$
C674x DSP scaled to 40nm	$1.98*10^7$	1495mW	$1.32*10^7$	$4.95*10^5$
Stratix IV 230 GX	$2.35*10^{10}$	12812mW	$1.83*10^9$	$7.83*10^7$

1) These three devices are not using the same process technology – the FPGA is on a more advanced node. Hence the comparison between them mixes process and architectural advantages. Estimate how the performance, performance / W and performance / $ values for the C5505 DSP and C674x DSPs would change if each were implemented in a 40 nm (instead of its 90 nm) process, and record the values in Table 3. It is not possible to do a perfect prediction – give your best estimate, and explain how you obtained it.



$$P_{dynamic} \propto {V_{DD}}^2 \cdot f_{clk}$$

A: The slide from an undergraduate course and the relationship of dynamic power depending on V_{DD} and clock frequency are the two basic guidelines to make estimations for DSP chips at

40 nm node. Scaling functions are listed below:

i) New throughput = Old throughput * area scaling factor * clock freq. scaling factor

ii) New power = Old power * V_{DD} scaling factor * area scaling factor * clock freq. scaling factor

iii) New cost = Old cost * Fab cost scaling factor

Power consumption of DSP chips are estimated to be mainly dynamic power so the V_{DD} scaling factor and clock freq. scaling factor is directly applied on total power.

2) Comment on the performance (sample/s), performance / W, and performance / $ of the FPGA and the DSP devices. Briefly explain what leads to the differences on these metrics between the devices.

A: In terms of performance, FPGA is around 1000x better than the DSP chip C6x and this advantage drops to about 150x better in terms of performance / W and performance / $.

The huge gap between FPGA and DSP chips seems to be the result of FPGA's efficient hardware implementation of FIR filters. DSP chips have many other modules that are not utilized by the filter and instruction set architecture introduces a lot of overhead. On the other hand, FPGA implements pipelined streaming hardware which has much less architectural overhead.

On the other hand, DSP chips target low-power and low-price market, meaning the chips are better at saving power and area. As a result, the performance / W and performance / $ metrics are doing better than simply the performance.

4. Reference:

1) Slide from ECE437, University of Toronto