Evaluation of the iCE40 FPGA for Use in the SlowDAQ*

Gen2 Hardware Meeting - June 21st, 2018

Bunheng Ty

ty@wisc.edu

WIPAC
Wisconsin IceCube Particle Astrophysics Center

WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

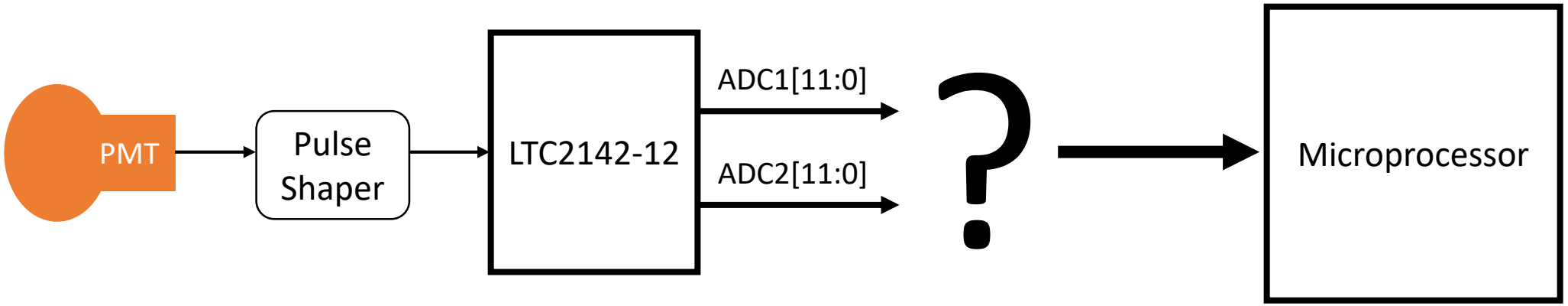
*official name still to be determined

Outline

- The need for an FPGA
- Lattice’s iCE40
- Basic firmware
- Power Measurements

Disclaimer: I am not, by any mean, an FPGA expert

Main Role of the FPGA



- LTC2142-12
  - 12 bits, 65Msps
  - 2 channels
  - 46 mW per channel
  - 125Msps variants available

- Need something to manage and buffer PMT waveforms
- We concluded that it has to be an FPGA
- Cheap and low power?

Max10 vs iCE40

10M02DCV36I7

- Newest in the Max series, now marketed as an FPGA
- 3 mm × 3 mm
- 2K Logic Elements
- 108Kbit RAM
- 27 IOs
- 2 PLL
- $4
- 35mW static power

ICE40LP1K-QN84

- “World’s smallest FPGAs”
- 7 x 7 mm
- 1.28K Logic Elements
- 64Kbit RAM
- 67 IOs
- 1 PLL
- $3
- 0.1mW static power

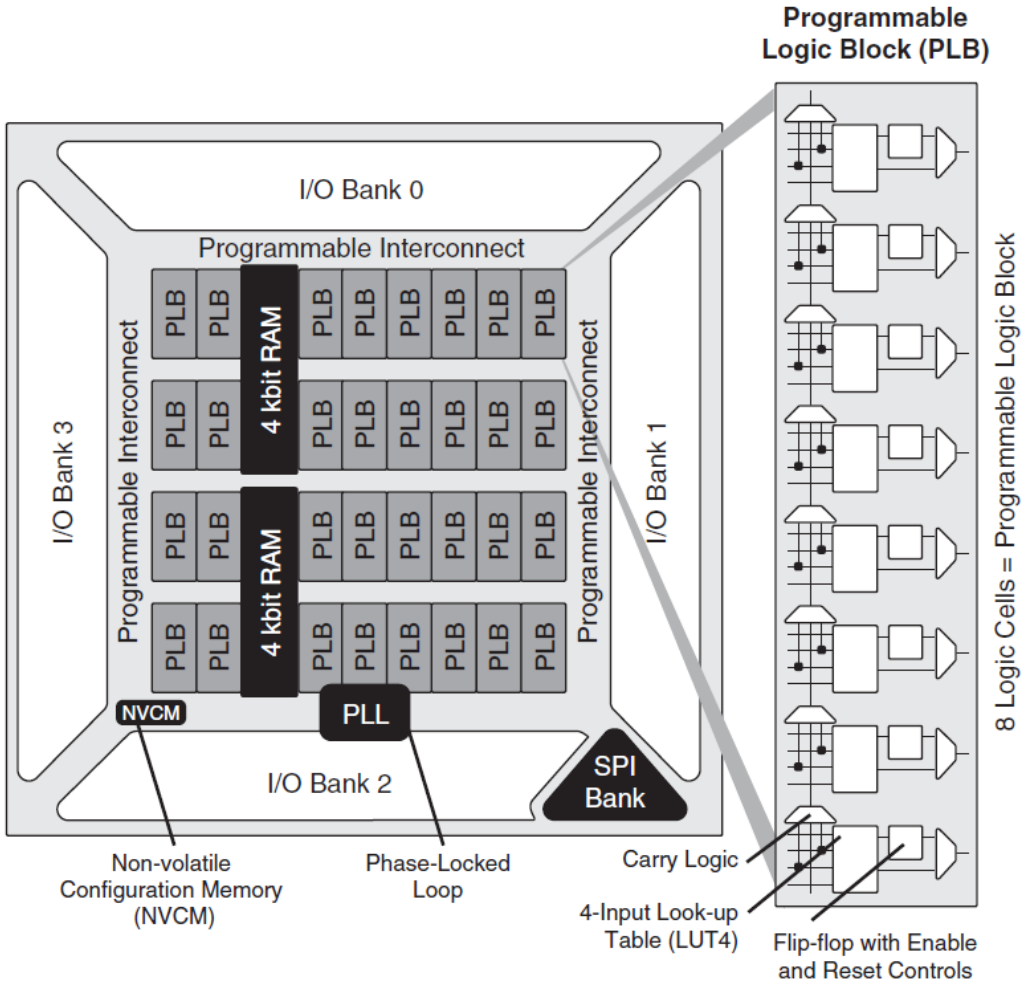
Lattice iCE40 LP/HX/LM

The iCE40 seems to be much more suitable for our design

More on the iCE40 FPGA

- Originally created by SiliconBlue Technologies, a start-up founded in 2005 by former employees of Actel, AMD, Lattice, Monolithic Memories, and Xilinx. Most notable was John Birkner, one of the inventors of programmable array logic. (Wikipedia)
- SiliconBlue was acquired by Lattice Semiconductor in 2011.
- The iCE40 family (40nm process) was launched in 2011. The latest release was the iCE40 UltraPlus series in 2016.
- Found in iPhone 7 (iCE5LP4K), Samsung Galaxy S5 (iCE40LP1K)

iCE40 FPGA - Reprogrammability



Three options:

1. Internal NVCM (one-time programmable)
2. External SPI Flash (Master SPI mode)
3. System microprocessor (Slave SPI mode)

Vey convenient. Programming and normal data transfer can be done via the same SPI connection to a microprocessor.

iCE40 FPGA Variants

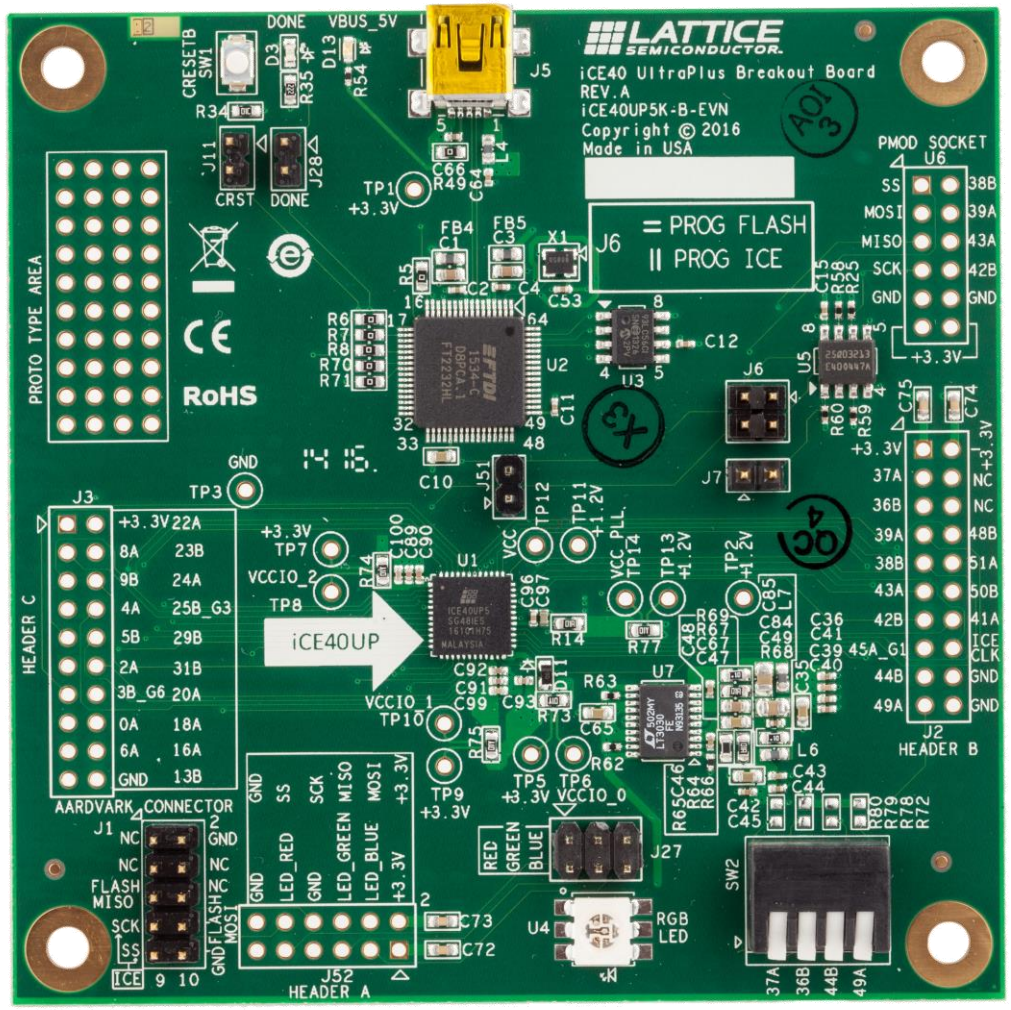
- Currently in the design: ICE40LP1K-QN84
  - This can easily handle 2 ADC channels (maybe even 4)
- Can also move up to the ICE40HX4K-TQ144
  - More logics, more IOs : more ADC channels
  - A higher performance variant

- But right now, testing ICE40UP5K-SG48.

  Because that’s what is on the evaluation kit we ordered

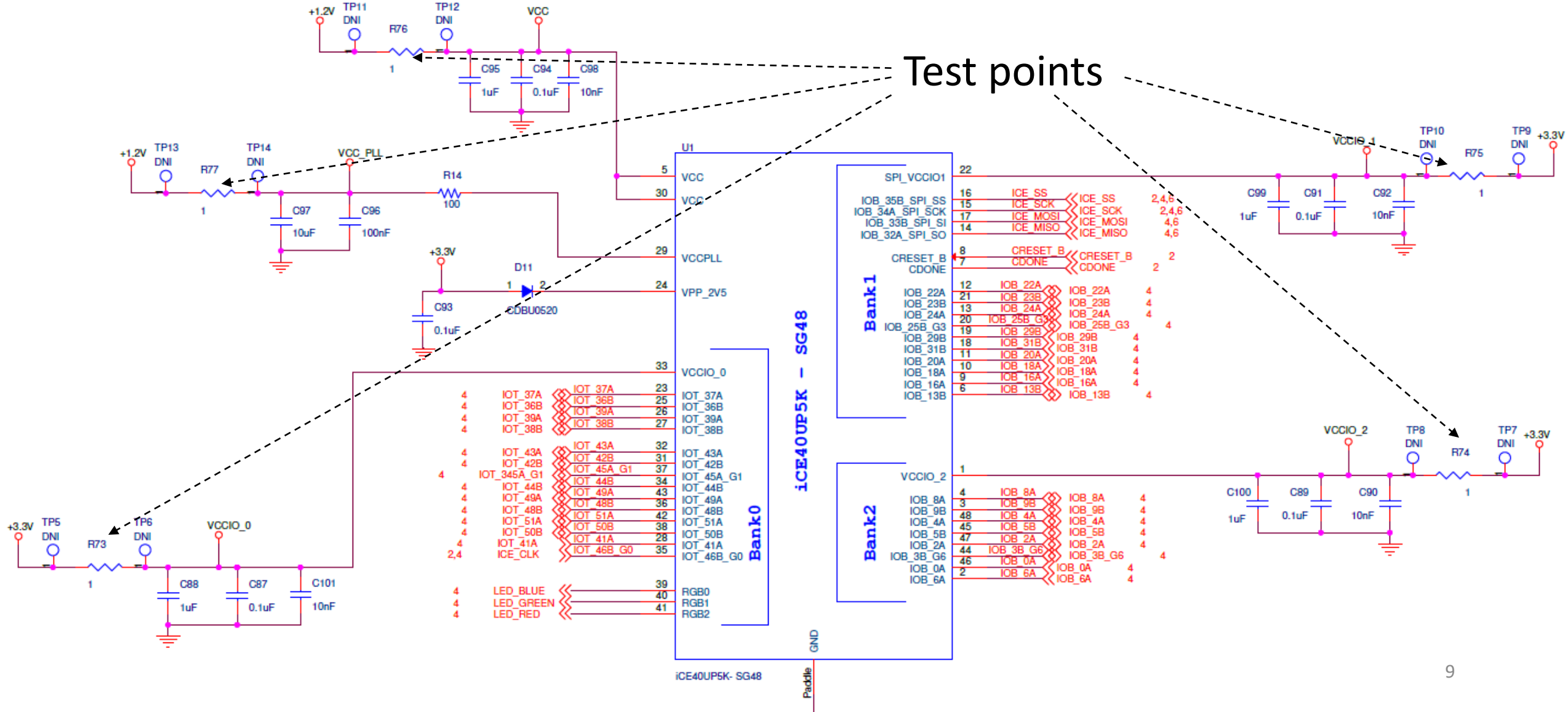
iCE40 ULTRAPLUS BREAKOUT BOARD

ICE40UP5K-B-EVN



- The actual device on board is ICE40UP5K-SG48.
- The UltraPlus family is the newest addition to the iCE40 series (2016). The main differences seem to be the presence of hard IPs (I2C, SPI, Oscillator, PWM, SPRAM…) and even lower power consumption.
- Program via a USB cable. There is a and FTDI's USB-to-SPI chip on board.

iCE40 ULTRAPLUS BREAKOUT BOARD



Firmware

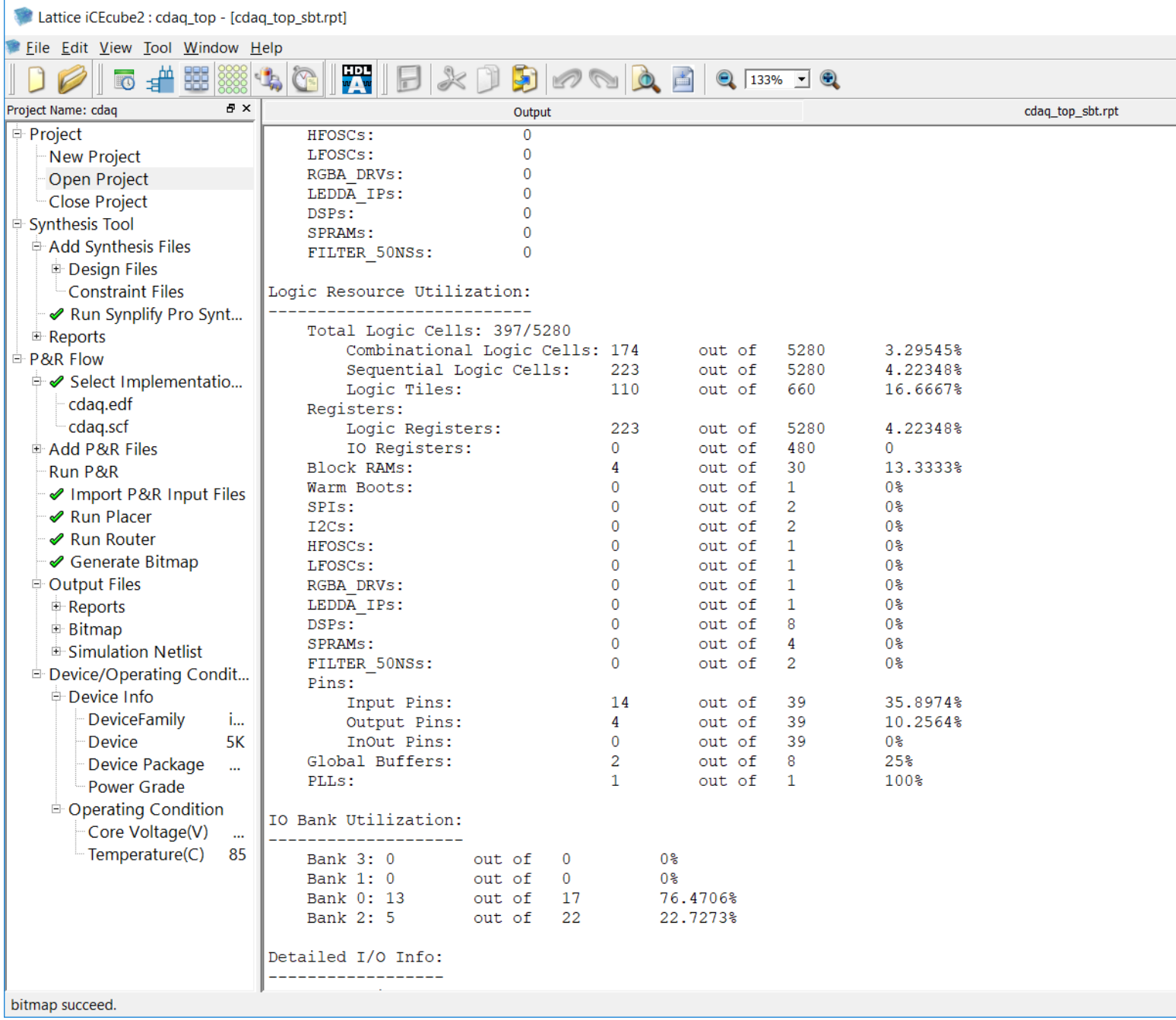
- A simple firmware has been written in Verilog for the dev board. Currently it does:
  - 1 Channel, 12 bits, data buffering at 60 MHz
  - 60 MHz clock routed to an output pin to drive an ADC
  - 12 Kbits Data FIFO
  - 2 Mbaud UART (tx only)
  - It pushes data into its FIFO whenever the external trigger input goes up (pre and post trig data available and adjustable). Then whenever the FIFO is not empty, it reads it out and send it via a 2 Mbaud UART.

Firmware

Compilation Report:

397 Logic Cells Used

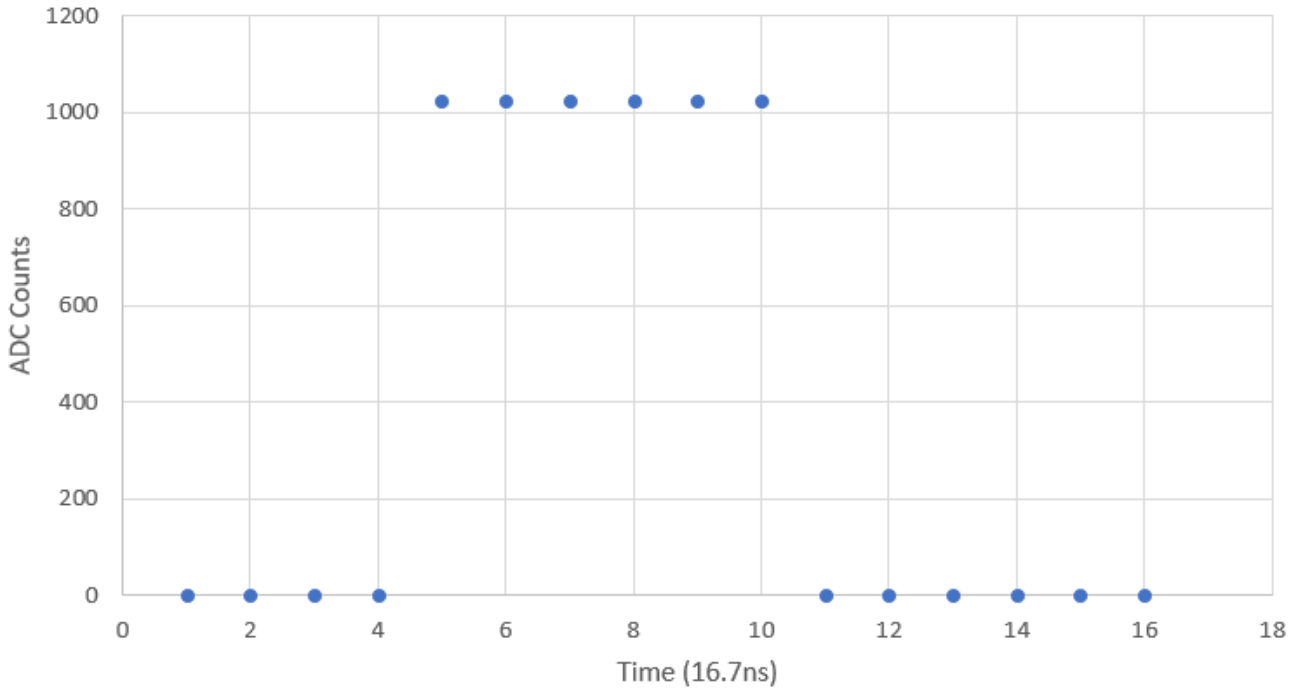
So no need to worry about having enough logic cells. The more immediate constraint is the number of IOs.



Firmware

Data readout demonstration:

A waveform



Command Prompt

```
C:\Users\Bunheng\Desktop\ice40UP>python pyserial_read_new.py
00000000
11111111

00010000
10000100
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000100
01000000
00000000
00000100
01000000
00000000
00000100
01000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000

00010000
00001100
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000100
01000000
00000000
00000100
01000000
00000000
00000100
```

Header

Power measurements

Board running the firmware described in previous slides. But no trigger, and ADC inputs floating

Test Point	FPGA Part	Current	Power
TP11-12	Core	2.2 mA	2.64 mW
TP13-14	PLL	0.1 mA	0.12 mW
TP5-6	IO Bank 0	2.4 mA	7.92 mW
TP9-10	IO Bank 1	0.2 mA	0.66 mW
TP7-8	IO Bank 2	0.0 mA	0 mW

TOTAL: 11 mW

This is larger than expected. We were hoping for <5mW per channel. But much of the power goes to IO bank 0, which has a 60 MHz clock output pin. Removing that pin, the total power drops to 3.63 mW.

Factory firmware
(Controls an RGB led on the board)

Test Point	FPGA Part	Current	Power
TP11-12	Core	0.3 mA	0.36 mW
TP13-14	PLL	0.0 mA	0 mW
TP5-6	IO Bank 0	0.6 mA	1.98 mW
TP9-10	IO Bank 1	0.2 mA	0.66 mW
TP7-8	IO Bank 2	0.0 mA	0 mW

TOTAL: 3 mW

Power measurements

Board running the firmware described in previous slides. But no trigger, and ADC inputs floating

Test Point	FPGA Part	Current	Power
TP11-12	Core	2.2 mA	2.64 mW
TP13-14	PLL	0.1 mA	0.12 mW
TP5-6	IO Bank 0	2.4 mA	7.92 mW
TP9-10	IO Bank 1	0.2 mA	0.66 mW
TP7-8	IO Bank 2	0.0 mA	0 mW

With 1KHz trigger → 1.8 mW

It draws less power when not idle?

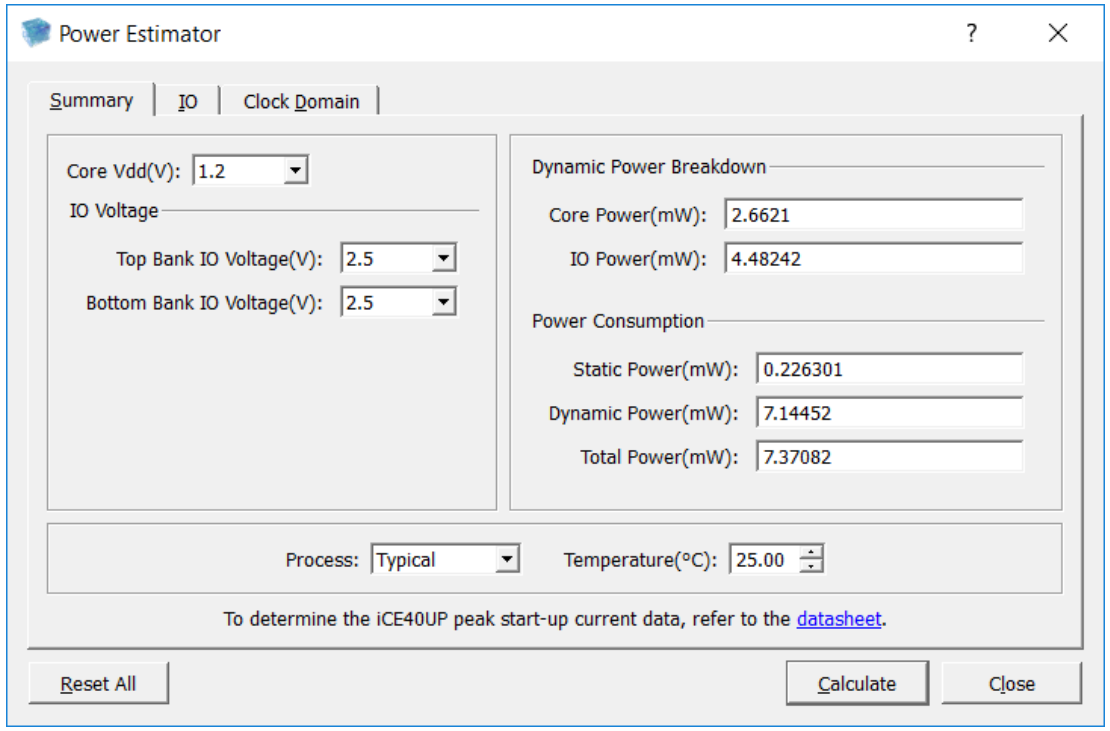
TOTAL: 11 mW

This is larger than expected. We were hoping for <5mW per channel. But much of the power goes to IO bank 0, which has a 60 MHz clock output pin. Removing that pin, the total power drops to 3.6 mW.

Power measurements

Two conclusions from the test:

1. Should be able to do basic PMT waveform buffering in iCE40 FPGA at negligible power cost (<5mW).
2. But power consumption is dominated by high-toggle-rate output pins.



This is also reflected in Lattice's own power estimator for the iCE40UP.

Conclusions

- The iCE40 FPGA seems to be ideal for our DAQ design.
  - Negligible power draw in the logic core
  - Available in easy-to-route QFN packages
  - Cheap : ~$5
- So far, no issue. Data buffering and readout demonstrated in actual test. Still to be tested: long term reliability and low temp performance.
- One thing to look out for is the IO power consumption, not just at the FPGA, but also throughout the board.
- Maybe this iCE40 FPGA is also the key to achieving modularity in our electronics.

```
`timescale 1ns / 1ps
//Verilog modules for cdaq firmware on the iCE40UP5K-B-EVN evaluation board. FPGA model number: iCE40UP5K - SG48
//Drive and buffer adc[11:0] at 60 MHz.
//A waveform starts several clock cycles (pre_trig) before external discriminator goes up.
//A waveform ends after disc goes and stay down for several clock cycles (post_trig).
//Save waveforms into data and header fifos, including pre and post triggger.
//Send those waveforms via uart when possible.
//ty@wisc.edu
//Last Update: June 13, 2018

module cdaq_top(
    input ICE_CLK,          //Pin 35 , 12MHz oscillator (to be multiplied by 5)
    // output LED_BLUE,     //Pin 39
    // output LED_GREEN,        //Pin 40
    // output LED_RED,          //Pin 41

    //input  ss,                //Pin 2  (IOB_6A)
    //input  mosi,          //Pin 46 (IOB_0A)
    output miso,            //Pin 47 (IOB_2A)
    //input  sclk,          //Pin 44 (IOB_3B_G6)    2MHz?

    input disc_trig,        //Pin 20 (IOB_25B_G3)
    input[11:0] adc1,       //Pin 28, 38(50B), 42, 36, 43, 34, 31, 32, 27, 26, 25, 23
    //input[11:0] adc2,     //Pin
    input resetn,           //active low, Pin 45 (IOB_5B)

    output trig_out,        //Pin 48 (IOB_4A)
    output trig_out2,       //Pin 3 (IOB_9B)

    output adc_clk          //Pin 37 (IOT_45A_G1)
    );

    //Parameters
    parameter post_trig_len = 5;
    parameter pre_trig_len  = 5;

    //PLL, input: 12MHz external oscillator, outputs: x2.5 and x5
    wire clk_A;             //12MHz x5 = 60MHz
    wire clk_B;             //clk_A/2  = 30MHz
    SB_PLL40_2F_PAD main_clks(.PACKAGEPIN(ICE_CLK),
                              .PLLOUTGLOBALA(clk_A),
                              .PLLOUTGLOBALB(clk_B),
                              .BYPASS(1'b0),
                              .RESETB(resetn)
                              );
    //\\ Fin=12, Fout=60;   (DIVF+1)/(2^DIVQ *(DIVR+1))
    defparam main_clks.DIVR = 4'b0000;
    defparam main_clks.DIVF = 7'b1001111;
    defparam main_clks.DIVQ = 3'b100;
    defparam main_clks.FILTER_RANGE = 3'b001;
    defparam main_clks.FEEDBACK_PATH = "SIMPLE";
    defparam main_clks.DELAY_ADJUSTMENT_MODE_FEEDBACK = "FIXED";
    defparam main_clks.FDA_FEEDBACK = 4'b0000;
    defparam main_clks.DELAY_ADJUSTMENT_MODE_RELATIVE = "FIXED";
    defparam main_clks.FDA_RELATIVE = 4'b0000;
    defparam main_clks.SHIFTREG_DIV_MODE = 2'b00;
    defparam main_clks.PLLOUT_SELECT_PORTA = "GENCLK";
    defparam main_clks.PLLOUT_SELECT_PORTB = "GENCLK_HALF";
    defparam main_clks.ENABLE_ICEGATE_PORTA = 1'b0;
    defparam main_clks.ENABLE_ICEGATE_PORTB = 1'b0;


    //---Temporary-------------------------------------------------------------
    reg[24:0] trig_counter=25'b0;
    reg       trig_state=0;
    assign trig_out = trig_state;
    assign trig_out2 = trig_state;
```

```
68      always@(posedge clk_B)
69      begin
70          trig_counter <= trig_counter + 1;
71          if(trig_counter == 25'b1110010011100001101111101) trig_state <= 1;
72          if(trig_counter == 25'b1110010011100001110000000)
73          begin
74              trig_counter <= 25'b0;                      //Counter is 1Hz
75              trig_state <= 0;                            //Trigger for 9 ticks -> 300ns
76          end
77      end
78      //-------------------------------------------------------------------------------
79
80
81
82
83      //adc's clock driven by pll in FPGA
84      assign adc_clk =  clk_A;
85
86      //Sequential always@ block runs on clk_A = 60MHz
87      reg[pre_trig_len-1:0] adc1_buff0, adc1_buff1, adc1_buff2, adc1_buff3,
88                            adc1_buff4, adc1_buff5, adc1_buff6, adc1_buff7,
89                            adc1_buff8, adc1_buff9, adc1_buff10, adc1_buff11;
90      reg[3:0] post_trig=0;               //Keeps track of how many post triggered data
        left to save
91      reg[9:0] data_len=0;
92      reg      write_data1;
93      reg      end_waveform;
94      reg      write_header1;
95      wire     keeping = disc_trig || (post_trig > 0);
96      always@(posedge clk_A)
97      begin
98
99          //digital buffer, <pre_trig_len> bits of 60MHz clk -> ~83ns
100         adc1_buff0 <= {adc1[0],adc1_buff0[pre_trig_len-1:1]};
101         adc1_buff1 <= {adc1[1],adc1_buff1[pre_trig_len-1:1]};
102         adc1_buff2 <= {adc1[2],adc1_buff2[pre_trig_len-1:1]};
103         adc1_buff3 <= {adc1[3],adc1_buff3[pre_trig_len-1:1]};
104         adc1_buff4 <= {adc1[4],adc1_buff4[pre_trig_len-1:1]};
105         adc1_buff5 <= {adc1[5],adc1_buff5[pre_trig_len-1:1]};
106         adc1_buff6 <= {adc1[6],adc1_buff6[pre_trig_len-1:1]};
107         adc1_buff7 <= {adc1[7],adc1_buff7[pre_trig_len-1:1]};
108         adc1_buff8 <= {adc1[8],adc1_buff8[pre_trig_len-1:1]};
109         adc1_buff9 <= {adc1[9],adc1_buff9[pre_trig_len-1:1]};
110         adc1_buff10 <= {adc1[10],adc1_buff10[pre_trig_len-1:1]};
111         adc1_buff11 <= {adc1[11],adc1_buff11[pre_trig_len-1:1]};
112
113         //Start storing waveform when disc_trig goes up
114         if(disc_trig) post_trig <= post_trig_len + pre_trig_len;    //When
            discriminator is triggered, reload post_trig, which also keeps keeping on
115
116         //Keeps track of how many samples have been loaded into fifo_data
117         if(keeping)
118         begin
119             write_data1 <= 1;
120             data_len <= data_len + 1;
121             if(!disc_trig) post_trig <= post_trig - 4'b0001;    //Count down to closure
                once discriminator falls back down
122             if(post_trig == 4'b001) end_waveform <= 1;          //If the countdown is
                allowed to reach 1, then the waveform is ended
123         end
124         else write_data1 <= 0;  //Need to make write_data1 stay on for one extra cycle
            because the fifo writes on negedge of wclk
125
126         //Takes care of finalizing waveform storage: saving time_counter value and
            length of waveform to fifo_header
127         if(end_waveform)
128         begin
129             end_waveform  <= 0;         //This block only runs one clock cycle per
                waveform saved
```

```
130             write_header1 <= 1;
131         end
132         if(write_header1) begin
133             data_len      <= 0;         //Reset data_len after header of waveform written
134             write_header1 <= 0;
135         end
136 
137     end
138 
139     //Data and Header fifo -- time_counter: [15:10], data_len: [9:0]
140     wire[11:0] data_in1 = {adc1_buff11[0], adc1_buff10[0], adc1_buff9[0], adc1_buff8[0],
         adc1_buff7[0], adc1_buff6[0],
141                            adc1_buff5[0], adc1_buff4[0], adc1_buff3[0], adc1_buff2[0],
                               adc1_buff1[0], adc1_buff0[0]};
142     wire[11:0] data1_out;
143     reg        read_data1;
144     wire[9:0]  adc1_fifo_lvl;
145     reg [14:0] time_counter=0;      //~1ms total at 30MHz
146     wire[15:0] header1_in = {time_counter[14:9], data_len};
147     wire[15:0] header1_out;
148     reg        read_header1;
149     wire[ 7:0] header1_fifo_lvl;
150     fifo_data adc1_fifo(.wclk(clk_A),
151                         .resetn(resetn),
152                         .data_in(data_in1),
153                         .write_en(write_data1),
154                         .rclk(clk_B),
155                         .data_out(data1_out),
156                         .read_en(read_data1),
157                         .fill_lvl(adc1_fifo_lvl));
158     fifo_header header1_fifo(.wclk(clk_A),
159                              .resetn(resetn),
160                              .data_in(header1_in),
161                              .write_en(write_header1),
162                              .rclk(clk_B),
163                              .data_out(header1_out),
164                              .read_en(read_header1),
165                              .fill_lvl(header1_fifo_lvl));
166 
167 
168 
169     //Communication
        logics------------------------------------------------------------------------------
        ----------------
170     reg[23:0] data_buffer;              //24 = 12*3 = 8*4
171     reg[ 9:0] num_data_read;
172     reg[ 2:0] readout_machine;
173     reg       byte_en;
174     reg       another_data;
175     reg       quit_tx;
176     wire[7:0] byte_to_send;
177     wire      uart_ready;
178     my_uart my_uart0(
179                      .clk(clk_B),                     //30MHz
180                      .byte_to_send(byte_to_send),
181                      .byte_en(byte_en),               //pulse one clk of this to load in
                         byte_to_send
182                      .ready(uart_ready),              //uart_ready means can load in
                         another byte to byte_to_send
183                      .tx(miso));
184 
185     assign byte_to_send = (readout_machine == 3'b001) ? header1_out[ 7: 0]:
186                           (readout_machine == 3'b010) ? header1_out[15: 8]:
187                           (readout_machine == 3'b011) ? data_buffer[ 7: 0]:
188                           (readout_machine == 3'b100) ? data_buffer[15: 8]:data_buffer[
                              23:16];
189 
190     always@(posedge clk_B)
191     begin
```

```
192
193         //Really coarse time counter ~1ms
194         time_counter <= time_counter + 1;
195
196         //State machine for sending out data to microcontroller
197         if(read_header1) read_header1 <= 0;
198         if(read_data1)
199         begin
200             if(!another_data) data_buffer[11: 0] <= data1_out;      //Store popped data
                in buffer
201             if(another_data)  data_buffer[23:12] <= data1_out;      //Store possibly
                popped another-data in buffer
202             another_data  <= !another_data;
203             num_data_read <= num_data_read + 1;
204             read_data1    <= 0;
205         end
206         if(uart_ready)
207         case(readout_machine)
208         default:if(header1_fifo_lvl > 0)                //Get to work when header fifo
            not empty
209                 begin
210                     byte_en         <= 1;               //This switch allows the tx
                        machine to work
211                     read_header1    <= 1;               //Pop header
212                     num_data_read   <= 0;               //Reset num_data_read at start
                        of new waveform readout
213                     another_data    <= 0;
214                     quit_tx         <= 0;
215                     readout_machine <= 3'b001;          //This connects first part of
                        header to byte_to_send (LSB first)
216                 end
217         3'b001: readout_machine <= 3'b010;              //This connects second part of
            header to byte_to_send
218         3'b010: begin
219                     read_data1      <= 1;               //Pop a data (12 bits)
220                     readout_machine <= 3'b011;          //This connects first 2/3 of
                        data to byte_to_send
221                 end
222         3'b011: begin
223                     if(num_data_read < header1_out[9:0]) read_data1 <= 1;   //Pop
                        another data if have it
224                     else quit_tx <= 1;                                      //If not
                        have it, quit at 3'b100
225                     readout_machine <= 3'b100;          //This connects the last 1/3 of
                        data and possibly first 1/3 of another-data
226                 end
227         3'b100: begin
228                     if(quit_tx)
229                     begin
230                         byte_en         <= 0;
231                         readout_machine <= 3'b000;
232                     end
233                     else readout_machine <= 3'b101;
234                 end
235         3'b101: begin
236                     if(num_data_read < header1_out[9:0])
237                     begin
238                         read_data1 <= 1;                //Pop data for next round if
                            have it
239                         readout_machine <= 3'b011;
240                     end
241                     else                                //If not have it, quit
242                     begin
243                         byte_en         <= 0;
244                         readout_machine <= 3'b000;
245                     end
246                 end
247         endcase
248
```

```
249         end
250
251     endmodule
252
253
254     //----------------------------------------***END OF MAIN
        MODULE***----------------------------------------//
255
256
257
258
259     //uart -- tx only -- 2Mbaud
260     module my_uart(
261             input clk,                  //30MHz
262             input[7:0] byte_to_send,
263             input byte_en,
264             output reg ready = 1,       //For more byte_en
265             output reg tx = 1);
266         reg[3:0] tx_counter;
267         reg[3:0] two_MHz_counter;   //Run only when transmitting
268         reg[8:0] uart_reg;
269
270         localparam[3:0] baud_setting = 14;      //baud rate = clk/(baud_setting + 1)
271
272         always@(posedge clk)
273         begin
274
275             //Ready state - ready for byte_en
276             if(ready && byte_en)
277             begin
278                 uart_reg        <= {byte_to_send, 1'b0};
279                 two_MHz_counter <= 0;
280                 tx_counter      <= 0;
281                 ready           <= 0;
282             end
283
284             //Not ready state
285             if(!ready)                  //Do things to become ready again
286             begin
287                 two_MHz_counter <= two_MHz_counter + 1'b1;
288                 if(two_MHz_counter == baud_setting)
289                 begin
290                     two_MHz_counter <= 0;
291                     if(tx_counter < 9)                          //tx machine. 10 total bits
                        sent per trigger
292                     begin
293                         tx_counter <= tx_counter + 1'b1;
294                         tx         <= uart_reg[tx_counter];     //Send bit, LSB first,
                            unless when tx_counter is zero. In that case, uart start-bit is
                            sent (zero)
295                     end
296                     if(tx_counter == 9)
297                     begin
298                         tx         <= 1;
299                         ready      <= 1;
300                     end
301                 end
302             end
303         end
304     endmodule
305
306
307     //Data FIFO module
308     module fifo_data(
309         input        wclk,
310         input        rclk,
311         input        resetn,
312         input[11:0]  data_in,
313         input        write_en,
```

```
    output[11:0] data_out,
    input        read_en,
    output[9:0]  fill_lvl
    );

    reg[9:0] write_addr = 0;
    reg[9:0] read_addr  = 0;
    assign fill_lvl = write_addr - read_addr;

    //For managing writing to EBR
    always@(negedge wclk or negedge resetn)
    begin
        if(!resetn) write_addr <= 0;
        else if(write_en) write_addr <= write_addr + 1;
    end

    //For managing reading from EBR
    always@(negedge rclk or negedge resetn)
    begin
        if(!resetn) read_addr <= 0;
        else if(read_en && (read_addr < write_addr)) read_addr <= read_addr + 1;
    end

    //12-bit width not an option, thus use three 4-bit width EBR blocks
    SB_RAM1024x4NRNW ram1024x4_inst2(                    //Note: Updates at negedge clk
                                .RDATA(data_out[11:8]),
                                .RADDR(read_addr),
                                .RCLKN(rclk),
                                .RCLKE(read_en),
                                .RE(read_en),
                                .WADDR(write_addr),
                                .WCLKN(wclk),
                                .WCLKE(write_en),
                                .WDATA(data_in[11:8]),
                                .WE(write_en)
                                );
    SB_RAM1024x4NRNW ram1024x4_inst1(                    //Note: Updates at negedge clk
                                .RDATA(data_out[7:4]),
                                .RADDR(read_addr),
                                .RCLKN(rclk),
                                .RCLKE(read_en),
                                .RE(read_en),
                                .WADDR(write_addr),
                                .WCLKN(wclk),
                                .WCLKE(write_en),
                                .WDATA(data_in[7:4]),
                                .WE(write_en)
                                );
    SB_RAM1024x4NRNW ram1024x4_inst0(                    //Note: Updates at negedge clk
                                .RDATA(data_out[3:0]),
                                .RADDR(read_addr),
                                .RCLKN(rclk),
                                .RCLKE(read_en),
                                .RE(read_en),
                                .WADDR(write_addr),
                                .WCLKN(wclk),
                                .WCLKE(write_en),
                                .WDATA(data_in[3:0]),
                                .WE(write_en)
                                );
endmodule

//Header FIFO module, dual clocks
module fifo_header(
    input        wclk,
    input        rclk,
    input        resetn,
    input[15:0]  data_in,
    input        write_en,
```

```
383       output[15:0] data_out,
384       input        read_en,
385       output[7:0]  fill_lvl
386       );
387
388       reg[7:0] write_addr = 0;
389       reg[7:0] read_addr  = 0;
390       assign fill_lvl = write_addr - read_addr;
391
392       //For managing writing to EBR
393       always@(negedge wclk or negedge resetn)
394       begin
395           if(!resetn) write_addr <= 0;
396           else if(write_en) write_addr <= write_addr + 1;
397       end
398
399       //For managing reading from EBR
400       always@(negedge rclk or negedge resetn)
401       begin
402           if(!resetn) read_addr <= 0;
403           else if(read_en && (read_addr < write_addr)) read_addr <= read_addr + 1;
404       end
405
406       //
407       SB_RAM256x16NRNW ram256x16_inst0(                    //Note: Updates at negedge clk
408                                   .RDATA(data_out),
409                                   .RADDR(read_addr),
410                                   .RCLKN(rclk),
411                                   .RCLKE(read_en),
412                                   .RE(read_en),
413                                   .WADDR(write_addr),
414                                   .WCLKN(wclk),
415                                   .WCLKE(write_en),
416                                   .WDATA(data_in),
417                                   .WE(write_en),
418                                   .MASK(16'b0)
419                                   );
420   endmodule
```