This describes an fpga project focused at the automated collection of some data as part of a test circuit. The fpga will generate a voltage ramp from gnd to vcc, while at the same time collect voltage data at two points in circuit.

I bought the following components from ebay for a little over $25 and received delivery in about three weeks. I bought two of the cyclone II fpga boards with the same old Cyclone II part, figured at about $10/board at least one will be usable – they both were. Also bought a couple ft232rl ($0.79!) and a few pcf8591 arduino adc modules (we are only targeting a 5-10% accuracy so 8 bits is just about ok). So, the fpga will also require rs232 and i2c capability.

FPGA Requirements Summary

- cyclone II

- rs232 interface

- voltage ramp generator

- i2c bus for ADC module

- collect two voltages

The latest version of the (free) Altera webpack Quartus tools to support cyclone II was 13.0sp1, this release is supported for a number of OSs including Windows and Ubuntu (up to 14.04) . I did try to install the Quartus tools on ubuntu 17.04 and ran into some library issues – which I decided not to try and fix but moved to a machine running ubuntu 14.04lts. There are still some usb blaster permission issues which can be fixed by going to the quartus install path bin directory and killing the jtag daemon (sudo killall -9 jtagd) and than starting it again with root privileges (sudo ./jtagconfig). There are several websites that can help with details one of which is listed below.

Hello World! First step is that we need to be able to send and receive data ‘packets’ to the fpga. We have a serial port communicating on the desktop side with a program like minicom. Taking little steps, short TX and RX on ft232rl dongle and verify that you get loopback in minicom terminal, next hook TX and RX up to fpga and connect the pins internally configure and test. Next we need to extract bytes from the rs232 serial, found a nice uart on fpga4fun.com that was pretty easy to include and loopback after hooking up the clock to my fpga (50MHz on both boards), changing downloaded uart clocks to 50MHz (from 200MHz) and setting my baud rate to 115.2K on both minicom and the uart verilog. I was pleased at this point to realize that you can use the +5v output of the ft232rl dongel to power the fpga board – even off an unpowered hub.

We need an ascii string that can be decoded by the fpga to be able to read and write data to user register space, these strings will be written and read from /dev/ttyUSBx either by program calls or by fat-fingering with minicom. What is implemented here uses an 8 bit address to access 16 bit registers in the fpga. The serial strings sent out have the form “w 0x12 0xabcd\n” and “r 0x12\n”, the read command will respond with 16 bit register read “0xabcd\n”, for example. We now have registers that we can write and read so now we can start hooking up some modules that will do something.

Ramp Generator. The ramp generator uses a 26 bit counter to count up clock ticks. The top nine bits are used to control the width of the output pulse. This code also generates a pwm sinewave on another output, costable is a 512x9 rom with a cosine .mif loaded. The rate of the output ramp or sinewave is determined by the value stored in the pwmtone user register, a value if 1 is about 1.5Hz, 2 for 3Hz, 0x3415 for 20KHz, etc… The outputs are filtered with a 68 ohm resistor and a 1uF capacitor. In another module data collection is started with a write to the i2c\_adata\_cmdreg user register, on sensing this signal the counters are reset so that data collection starts on ramp = 0volts.

//pwm sinewave synthesiser (and ramp)

reg [8:0]pulsewidth;

reg [25:0]pwcos;

always @(posedge clk) begin

pulsewidth <= pulsewidth + 1;

pwcos <= pwcos + pwmtone; //pwmtone adjusts frequency

if (pulsewidth > lut0out) cosout <= 0; else cosout <= 1; //sinewave 1 bit dac output

if (i2c\_adata\_cmdreg\_write == 1)pwcos <= 26'h0000000; //ramp sync - will glitch outputs

if (pulsewidth > pwcos[25:17] ) rampout <= 0; else rampout <= 1; //ramp 1 bit dac output

end

costable (

.address\_a(pwcos[25:17]),

.address\_b(),

.clock(clk),

.q\_a(lut0out),

.q\_b() );

This generates about a 100KHz (50MHz/512) variable width pulse train, at low frequencies the pulse width is repeated on many subsequent pulses. By the time we are incrementing the counter by 2^8 (about 400Hz) the pulse width is changing every pulse with. At 20KHz each cycle is only represented by about 5 different width pulses per second.

I2C Interface (Master). The i2c specifications seem to get a little vague after the basic byte level protocol. What is implemented here is a single byte write which returns a byte with both the master and slave contributions. The user register i2c\_data\_write at address 0x20 starts the i2c\_interface module state machine which toggle SDA and SCK pins to get each byte into/out of the attached i2c device. For example the ascii string “w 0x20 0xc190\n”; i2c\_data\_write[15]=1 sets an i2c start, i2c\_data\_write[14]=1 sets a stop after the byte, i2c\_data\_write[8]=1 is looking for slave ack. The i2c\_data\_write[7:0] bits is the byte scanned in the i2c cycle. The first byte in a i2c sequence is usually of the form {i2c\_addr[6:0],0,ack}. The command we sent in the example “w 0x20 0xc190\n” is setting a start and attempting a write on any i2c device attached at address 0x48 (in this case our pcf8591), the responding device pulls down ack and the fpga responds with “0x0090\n”, if the device were not plugged in we would get “0x0190\n”. Now we can write a program that polls i2c address 0x04-0x7f and detect any devices with unique addresses on our i2c bus.

for (my $addr = 0x04; $addr <= 0x7f; $addr++){

$modaddr = (1<<15) + (1<<14) + (1<<8) + ($addr<<1) + 0;

$temp = sprintf ("w 0x20 0x%04x\n", $modaddr);

$port->write( $temp ); }

and wait for packets to come pack. If the ack bit is low then a device is present at that location. My perl program i2c\_detect runs in about 100msec.

To read from the ADC we send the following commands :

$temp = sprintf ("w 0x20 0x8190\n"); $port->write( $temp );

$temp = sprintf ("w 0x20 0x0100\n" ); $port->write( $temp ); #channel 0 single

$temp = sprintf ("w 0x20 0x4100\n" ); $port->write( $temp );

$temp = sprintf ("w 0x20 0x8191\n"); $port->write( $temp );

$temp = sprintf ("w 0x20 0x00ff\n"); $port->write( $temp );

$temp = sprintf ("w 0x20 0x40ff\n"); $port->write( $temp ); #first conversion read out here

We should get back over the next 3-5msec;

0x0090

0x0000

0x0000

0x0091

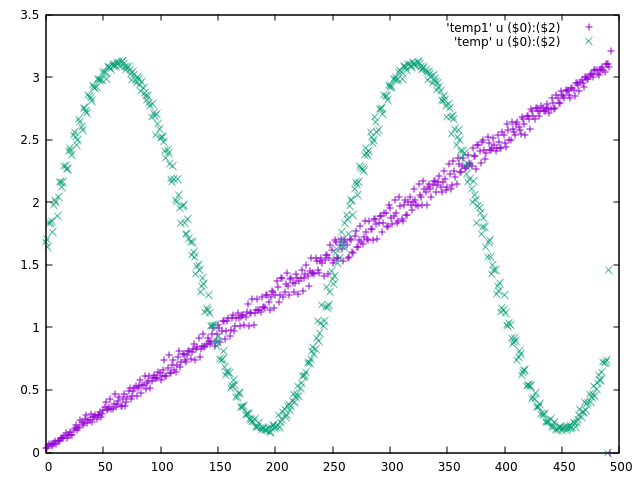
0x00?? ← result of last conversion (we don’t keep track)

0x00ab

The 0x00ab is telling us that there is 2.21 volts on the input of channel 0 (3.3\*171/255).

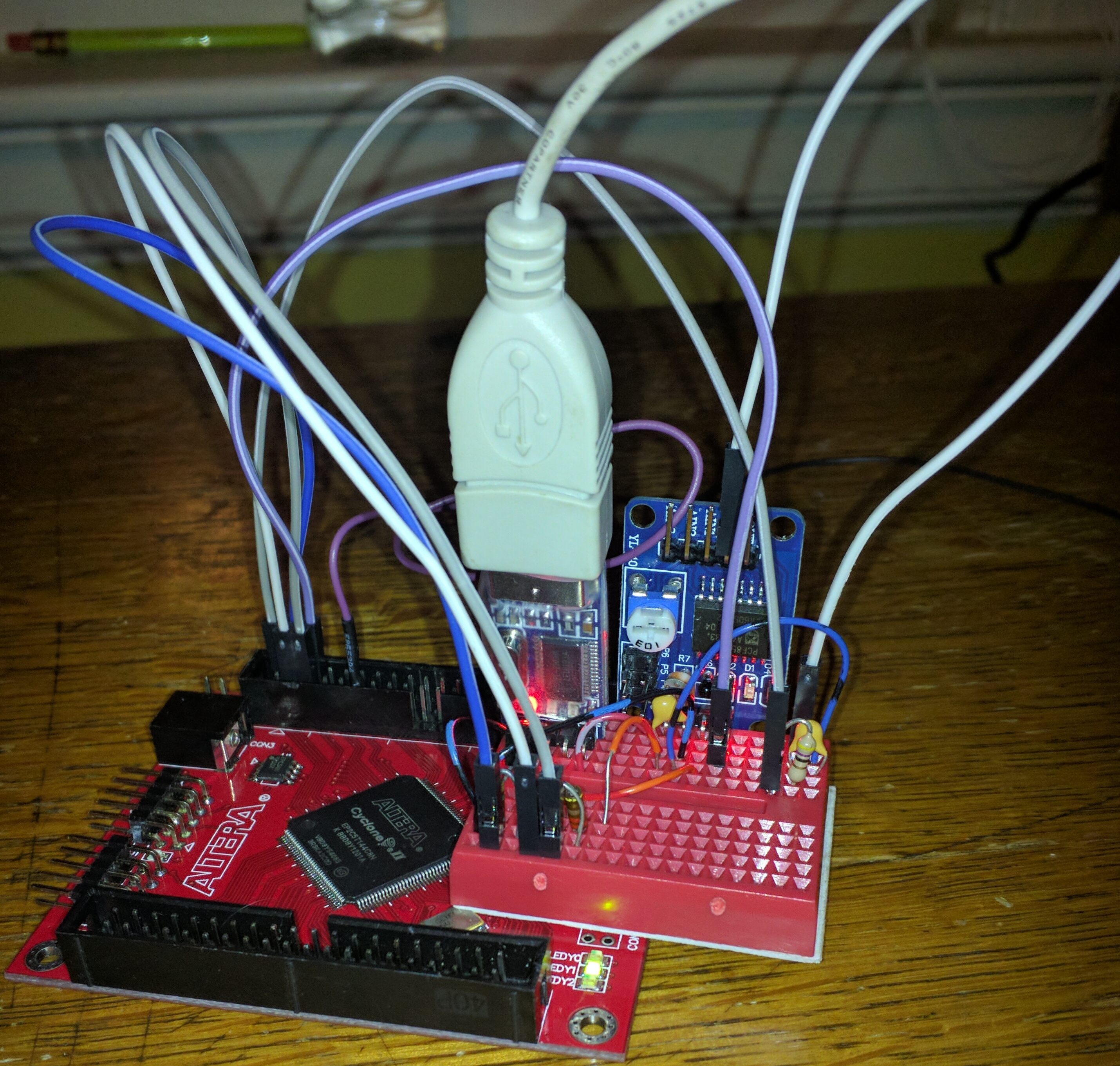
Turns out the maximum rate at which we can do this is about 1000 times per second before the serial port runs out of bandwidth. The write string is longer so that limits the rate at witch i2c bytes can be launched (14 bytes @ 115200buad = ~1msec/write command). However, once the pcf8591 is programmed to a channel and has made a measurement, each time that last command is repeated a new conversion occurs and the previous conversions data is read out. The maximum i2c SCK for this part is 100KHz before conversion accuracy suffers, so you can get a pretty good reading at 10K conversions per second. There is no way we can transmit this data over the serial port at 115.2Kbuad so we need to build a fifo buffer on the fpga. The module i2c\_autoread has a user register interface that specifies the number of samples (times to repeat the last i2c byte) and the rate at which to take each measurement. Timers were implemented for 0.1ms, .3msec, 1msec, 3msec and 10msec. Number of samples was specified powers of 2 from 64 to 1K. The data is read out of the fifo by reading its user register address (in this case “r 0x31\n” as many times as you need to flush the fifo) and waiting for the data to return, reading “r 0x32\n” returns the fifo status full, empty and usedwords.

Actually doing something. This is the output from a program that starts a 20Hz ramp (“w 0x60 0x0028\n”) and collects 512 points of data at 0.1msec/sample (“w 0x20 0x0400\n”) (with some help from a gnuplot system call). Looks like my .mif table stores cos(0) to cos(4pi) – oops. The x-axis is the sample number read back from the i2c fifo.



The solderless prototype boards that I have this thing stitched together with are a little noisy, not using the usb dongel for power did not improve the noise noticeably. I have dropped the first 10 packets because they were usually corrupted. When I started debugging the corruption problem it started looking like noise related (moving wires made it go away sometimes) – if this gets wired up better (or if we use fpga with working open drain outputs) we can see if there are any firmware/software components to the corruption.

Incorporation into test circuit. This fpga ramp output will be used to drive the gate of an fet with a source follower. The voltage across the source follower resistor and the total input voltage will be measured repeatedly as the gate voltage ramps up. We can either use two i2c adc modules installed at the same address with different sda signals or use the auto increment mode of the pcf8591.The final deliverable of the project will be used to measure IV curves with 256 measurement points in about 25msec.



Work in progress. The i2c\_interface module works but needs some code neatening. The protoboard is noisy – anything hardwired would probably be a lot better. Both adcs were working for a short period but the noise was worse, the dual channels have to be cleaned up. At some point I would like to experiment with the newer ft232h devices which have a mpsse spi mode that can transfer 480Mbps (with the ftdi C libraries could probably move data in/out of fpga several 100x faster).

Notes on Serial Port Latency. Not all usb to serial dongels are equal. One that I have with the pl2303 chip seemed to work fine until I started looking at the loopback latency. One packet round trip was up to ~60msec and there was data loss in sending over about 64 bytes. The ft232rl dongel gave me worst case round trips of less than 4msec, you can use the whole bandwidth and will write buffer at least 4Kbytes until data starts getting lost. The ft232rl dongels that I got on ebay for $.79 with free shipping also have output pins for +5V – with enough current to supply the fpga board, and +3.3V to drive the circuitry attached to the fpga board.

Other I2C Devices. Other arduino module I2C devices that are working include the little ssd1306 oled display, ads1115 and a mpc4725.

Notes on Programming. The non-fpga programming on this project was done on an ubuntu 17.04 desktop with the distro’s perl and the cpan modules Time::HiRes and Device::SerialPort. Device::SerialPort makes the port easy to setup, the lookfor() routine returns whole strings.

Notes on Cyclone II – I could not get the tristate/open drain outputs on the Cyclone II working, tried a couple workarounds that other people had on the web and instantiated a couple Altera output modules with no joy. I ended up inverting the SDA output and driving a 2n2222 through a 2K resistor. On the collector a 150ohm resistor to Vcc. A separate fpga input was attached to the transistor’s collector. More wiring than I wanted but doable.

getting quartus on ubuntu 14.04

<http://www.fpga-dev.com/altera-usb-blaster-with-ubuntu/>

fpga4fun serial uart

<http://www.fpga4fun.com/SerialInterface.html>

rpi forums posting

<https://www.raspberrypi.org/forums/viewtopic.php?f=37&t=194264>