ENEL 453 – Lab Five

9-bit SAR

Abdullah Cheema 10127183
Moustafa Youssef 30015452

December 8th 2017

Contents

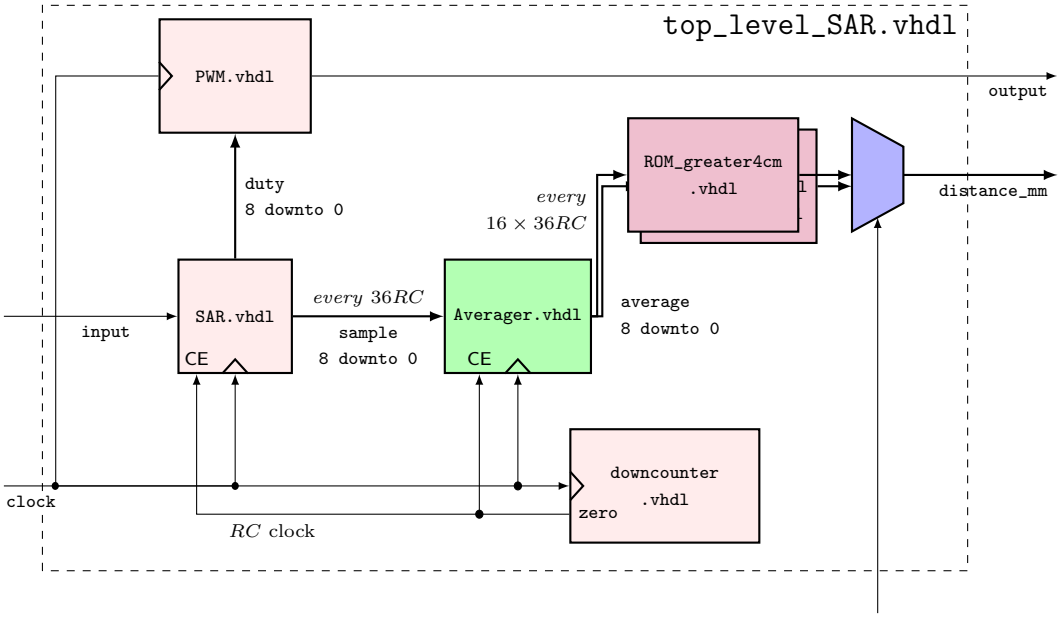


Figure 1: Block diagram of `top_level_SAR.vhdl`. The multiplexer shown was expected to allow the user decide whether to measure distances greater or less than 4 cm but wasn't implemented because of lack of time.

1 Technical Specifications

1.0.1 Design Summary

- **9-Bit Resolution** We have a measurement resolution of 1 mm, which there are 400 of in 40 cm. To achieve this resolution our PWM signal has a total period of 400 clock cycles, which divided by a board clock frequency of 100 MHz yields a frequency of 250kHz.
- We are using the **Successive Approximation Register** approach to determine the magnitude of the analog signal of the distance sensor. A resolution of $(1/400)\,0.25\%$ requires at least nine bits to be defined. Therefore it takes nine cycles of sending and receiving signals to create a complete sample.
- **Fully Tested** by the FPGA-self-testing approach reported here. Sensitivity verified at the LSB.
- **MicroBlaze Integration** The system is equipped with a MicroBlaze Soft Processor Core which allows VHDL programmers to tap some high level functionality. The MicroBlaze was used to integrate the VGA display and the ADC Converter and has proven to be a useful component of a physical testbench.

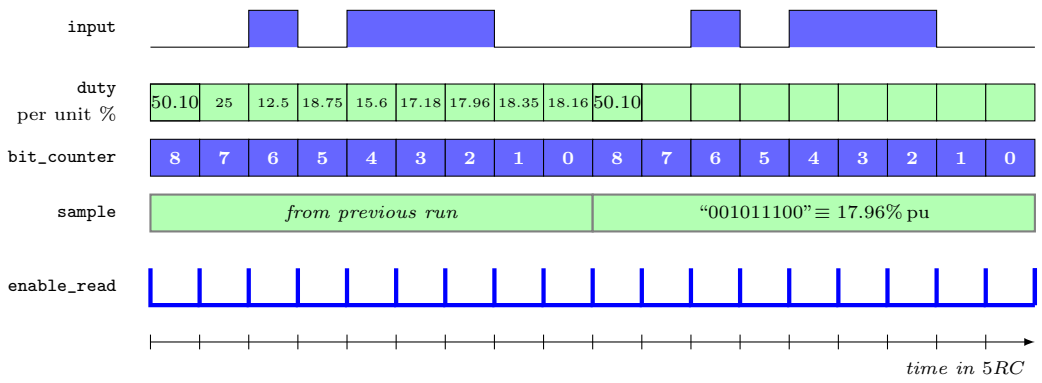


Figure 2: A simulation of a nine-bit SAR with a DC signal input with a voltage magnitude of 18% pu. Please see Subsection 1.3 to read about how we improved on this design.

1.1 Successive Approximation Register

The SAR method is an iterative binary search that tests voltage magnitudes down the bit range of a sample. Figure 2 shows a simulation of a nine-bit SAR with a constant wait time for every bit read-and-write cycle. Below is a quick explanation of how SAR works:

1. Set output to "$1\,0\cdots0$", which for a nine bit vector is equivalent to a per unit voltage magnitude of 50.10%.

2. If after waiting for the PWM signal to settle the comparator output is high, it means that the sensor voltage is higher than 50.10% pu. The first bit is determined to be one.

3. Now the second MSB is set high. Again, if the input happens to be low, it now means the sensor voltage is lower than the PWM voltage (of 75.15% pu), and the sensor voltage has the form "$10XXXXXXX$" where X are bits yet to be determined.

Synchronous and Enabled Design

We created a clock with a frequency of RC to enable signal writing and reading, where RC is the time constant of the low pass filter that supplies by `SAR_output`, and which will be discussed in greater detail in the next section. SAR's sequential actions are illustrated in Figure 3.

1.2 Filter Design

A low pass filter provides a natural transformation of a discrete signal into a continuous one. Figure 4 illustrates that the duty cycle of a PWM signal becomes the per unit voltage of the filtered output, where base voltage is equal to the magnitude of the high signal (or the analog voltage derived from a duty cycle of "$1\cdots1$"). We need to filter the signal that is generated by `SAR.vhdl` into an analog signal with a suitable ripple and settling time. However these two

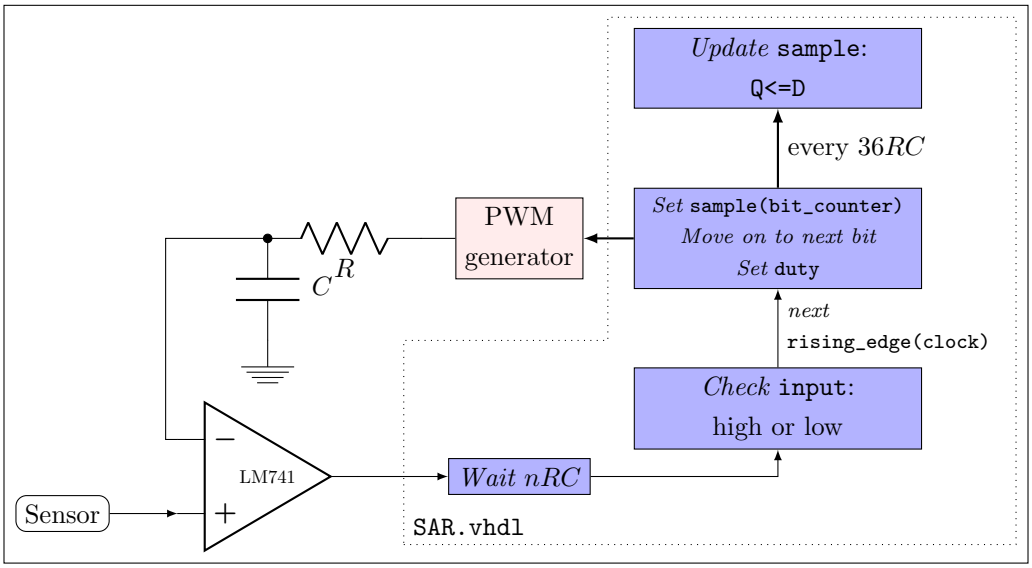


Figure 3: Top level overview of our Successive Approximation Register design showing external hardware. n depends on which bit is being tested and is explained in Subsection 1.3.

properties are antagonistic as far as a low pass filter is concerned: the higher the RC constant, the less voltage ripple and sample variance, but the longer it takes for the signal to settle from the last change in duty cycle. But since we are waiting longer to create higher quality samples, the (statistical) sample size can be justifiably reduced. The equation below was also obtained from [1]:

$$V_{ripple} = 4/\pi \times A$$

where $A = V_{out}/V_{in} = X/\sqrt{X^2 + R^2}$ at the operating frequency of 250 kHz. To utilize the entire resolution of our 9-bit vector we will need a ripple that is at least one order of magnitude less than our target resolution. The table below shows the properties of the low pass filter transforming our SAR output signal.

Settling time of a PWM signal

How long does it take for a PWM signal to reach steady state wrt to its ripple? Figure 5 adapted from [1] shows that a first order response to a change in duty cycle of a switching signal is identical to the first order response of a step change in input of an analog signal, provided that the PWM frequency is much larger than the filter's cut-off.

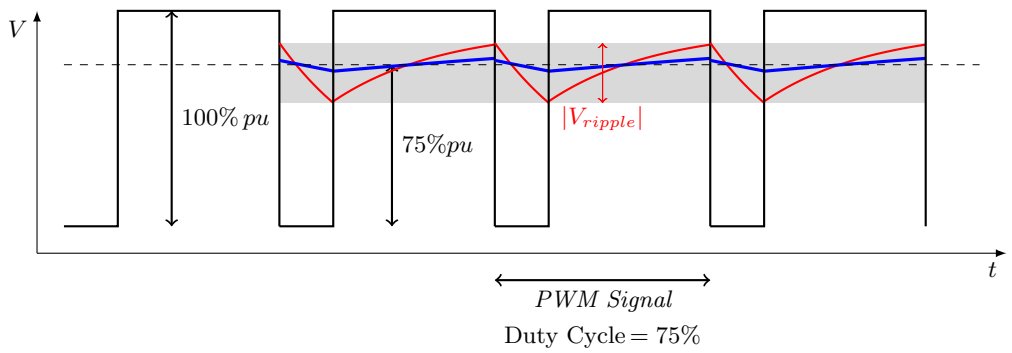


Figure 4: Low pass filter outputs of a PWM signal with a 75% duty cycle. The blue response has a time constant eight times larger than the red's and therefore has less of a ripple. However the blue curve would have a slower response to a change in duty cycle, which is illustrated in Figure 5.

Resistance		8.3	kΩ
Capacitance		470	nF
Cut-off frequency	$1/2\pi RC$	1691	Hz
Time constant	RC	940	μs
Sample time	$16RC$	15	ms
Maximum display update	$16 \times 36RC$	541	ms
$f = 250\,\text{kHz}$			
	$\lvert A \rvert$	0.07	% pu
	$\lvert V_{ripple} \rvert$	0.09	% pu

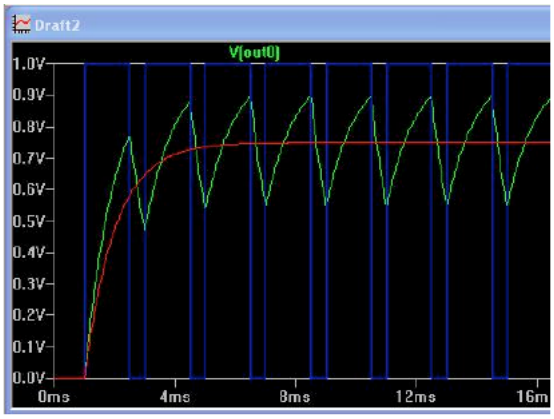


Figure 9 - One pole Low-pass with Continuous D=0.75 Pulse Train and a second with a 0.75V step.

Figure 5: Continuous and switching response to step change in input voltage and duty cycle, respectively. We can see that the time it takes for the PWM signal to have a steady state ripple is the same as the time it takes the continuous signal to settle.

1.3 Timing Optimization

The basic SAR design illustrated in Figure 2 and highlighted in [2] assume a constant settling time for all bits. In this section we will improve on this design by optimizing the waiting time for each bit. First of all many of the MSB's don't need a $5RC$ waiting time. For example, waiting just one RC causes a signal to decay by 64% which is accomodating of the first bit that has a value of 50%. The second signal has a value of 25% so waiting just one RC wouldn't provide enough time. This logic of offset vs bit value is illustrated in Figure 7. On the other side of the vector waiting only $5RC$ isn't enough time because it leaves an offset of 0.67% that is higher than the value assumed by the two LSB's[1], and which therefore require more settling time. Figure 6 shows the minimal time required for each bit in terms of RC, making the behavioural VHDL implementation simple[2]. By investigating and implementing a bit-specific wait time we managed to reduce sample time from $45RC$ to $36RC$ and we can now achieve our intended resolution.

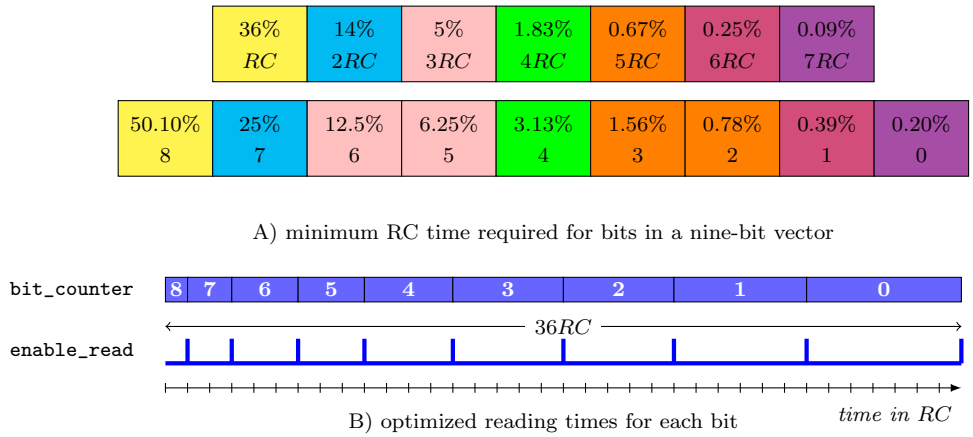


Figure 6: Figure A shows the minimum amount of time needed to wait before sampling bits of a nine-bit vector wrt of RC seconds where RC is the time constant of our first order low pass filter. The top boxes show the offset that is left behind nRC seconds, while the bottom boxes shows the relative value of each bit. For example, we only need to wait a minimum $3RC$ to allow changes in the sixth and fifth bits to settle. Instead of waiting $5RC$ for all bits, we provided each bit with the sampling time that's suited for its resolution, and reduced the sample time from $45RC$ to $36RC$.

8	···	2	1	0
50.10%		0.781%	0.391%	0.196%

[1]

[2]The basic SAR design with a constant $5RC$ wait time still achieves a centimeter resolution without this upgrade in design. This is because a centimeter requires four LSB's or more to be defined for which as shown in Figure 6A require no more than $4RC$ seconds to reach an acceptable offset.

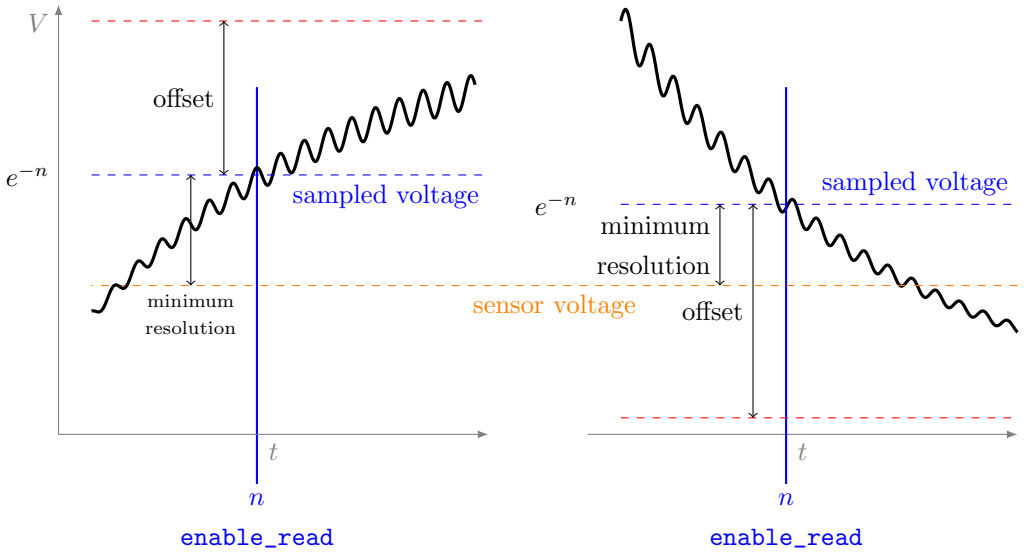


Figure 7: On the graph on the left we can see that the signal's magnitude is increasing suggesting that the new duty cycle is greater than previous. The blue analog signal derived from `SAR_output` was sampled to be higher than the sensor voltage, and waiting more time will not have changed that fact. On the graph on the right - where the new duty cycle happens to be lower than previous - we can see that if `enable_read` was sampled a little later the sampled voltage would have become less than the sensor voltage. We need to wait enough time to make sure the signal has decreased to the point where it's less than the bit resolution.

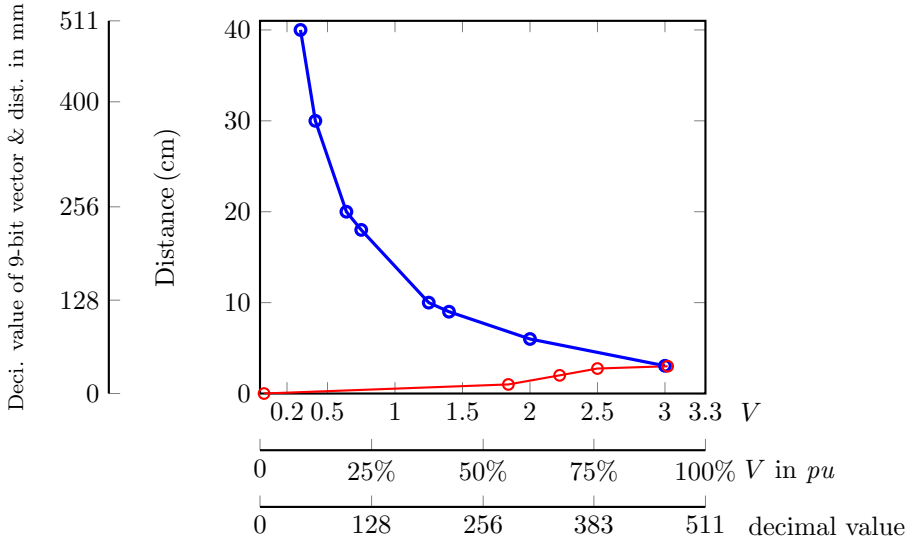
1.4 Data Processing

1.4.1 Averaging

Our comparator is going to be susceptible to white noise from the analog sensor input and a ripple from the PWM. Both these signals have no bias and we can therefore use an averaging technique to cancel their effect. Our intent is to perform the calculation within the limits of the FPGA. We will create a `sum` vector large enough to add up the value of all the samples. To simplify division we count in roots of two - that way dividing is only a matter of shifting `sum` to the right. We picked a sample size of 16, which means we need our `sum` vector to be at least 13 bits long[3]. To divide by 16 we take the nine bits from 12 down to 4[4].

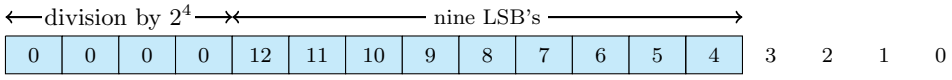
1.4.2 From Duty to Millimeters

The SAR module produces a nine bit vector that represents the voltage output of the distance sensor. We now have to translate this 9-bit voltage magnitude into a distance. The relationship is obtained from the sensor's datasheet and the values are linearly interpolated with the points shown in the graph below, and printed into a ROM file called `ROM_mm.vhdl`. Our voltage and distance variables are both operating as nine bit variables. As mentioned previously, we picked nine bits to have a resolution of 1mm over a 40cm range. As we can see below the decimal value of our `sample` vector is already in millimeters. No further arithmetic is necessary for conversion to cm's; all that's need is a decimal point.



[3] $\ln(16 \times 511)/\ln(2)$

[4]



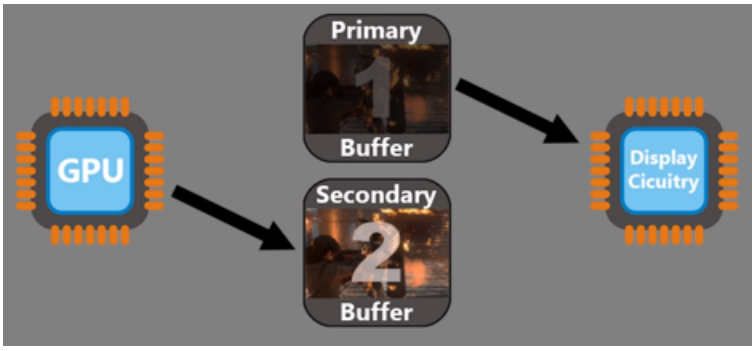


Figure 8

1.5 Display Hardware

When we started the project we realised that trying to generate output via computational method would be difficult. By computational we mean such as the bouncing box. The bouncing box activity calculated the value of what the pixels should be. So we decided to emulate some of the architecture we had used and decided to utilize a framebuffer. A framebuffer would allow us to map each pixel on the display to a memory location, so essentially creating a memory mapped display where one could change the pixel on the screen by writing a the corresponding rgb values into the memory that was mapped to the display. This behaviour very much is analogous to the linux /dev/fb0 file which is used by the OS to store what is displayed onto the screen.

We ended up facing a dilemma: since there were 720 by 640 pixels each needing 12 bits our display would be hogging up a lot of the resources on the FPGA. So we decided to scale up the pixels into pseudo pixels that we would rasterize onto the screen. Each pseudo pixel would include 8×8 real pixels. So our dimensions for the screen were reduced down to 80 by 60 pseudo pixels. This greatly improved our memory usage for creating the framebuffer.

The implementation of the framebuffer required a basic memory module. The framebuffer you would normally see implemented as parts of a video card/graphics card tend to map to all the physical pixels on the screen and tend to have two frambuffers. While one framebuffer is being used to rasterize an image onto the screen the other framebuffer is used to generate the next display screen that the user going see (see fig 1). This improves performance because as as one framebuffer is written to the other is being read and they continuously switch these roles depending on the refresh rate of the screen (mostly 60Hz). But since we were being memory conscious we decided on a single framebuffer which has a read and write port which can be read synchronously. So if you made a change to the display while the image is being rasterized it may be rasterized on the current refresh cycle or it may not, but it will definitely appear by the second refresh cycle.

The framebuffer was created using a RAM implementation in vhdl provided by Xilinx. We modified this file to accommodate us so that we could do synchronous reads and writes from the framebuffer by adding additional ports and logic onto the existing design provided by Xilinx (reference in the correspond RAM.vhd file). We then used some of the code from our previous lab (bouncing box) and gutted out all the non-essential components and then instantiated our framebuffer (RAM). We modified it heavily so that the screen would rasterize our framebuffer data. To test this vga module (with internal framebuffer) we preconfigured the framebuffer with an image I got from the internet. Then we wrote a python script to convert the image to corresponding pixel values which we could store in the framebuffer so that on power up the framebuffer would contain this image and thus this image should rasterize onto the screen if our code was correct. Which we knew when we saw the image in figure 2. We then tested this physically by connection the address and and pixel to the I/O of the board such as switches to see if could write pixels on the screen by playing around with the switch which we could do as shown in Figure 9.

Figure 9

1.5.1 Microblaze

After finally having built a level of abstraction where we could control the whole entire display by controlling only two busses for which we didn't have to worry about minute details such as where the scan line or the horizontal or vertical sync was. But now the tricky part of control and designing the logic that controls the display came into play. We initially decided to build

a basic implementation of the MIP32 architecture in vhdl (using chapter nine in the textbook) that could implement 6 opcodes (add, sub, jump, branch, lw, sw). Some of us were turned off by having to work down at that low level and sought an even greater abstraction. We discovered the plethora of soft-core CPUs that are offered by FPGA vendors such as Xilinx, Altera, etc. Xilinx offers a soft-core processor called the Microblaze which can be implemented at the fabric of the FPGA. This means instead of having dedicated hardware such as an atmel ARM processor chip on the processor you can implement a processor in vhdl which can be synthesized onto the FPGA. By providing users of their FPGAs this softcore CPU support they can satisfy their heavy are computing intensive needs. For example a user may implement multiple microblaze to process data in parallel allowing for great data processing throughput.

The journey we took in implementing the Microblaze was long and hard. Building and using the Microblaze is relatively easy but the learning curve was a bit steep. We had to become accustomed to AXI interfacing, Vivado IPs, Xilinx SDK environment, etc. A lot of the time we would get stuck on a problem and have no idea of how to proceed and would have to find ways through that problem or around it. For example interfacing with the Microblaze we needed to use GPIO IPs but then connecting components to the GPIO IPs we would need to have the component be AXI enabled. We ended up finding that GPIO also supported non AXI interfaces and used those. Then we ran into the problem of creating IP blocks from our VHDL code. Finally at the end we came up with the architecture in figure 3. You have the Microblaze at the center, the GPIO, UART, SAR and the VGA IP blocks the right. The other peripherals connected to the Microblaze include its instruction/data memory, blocks to enable AXI interfacing, etc as shown in Figure 10).

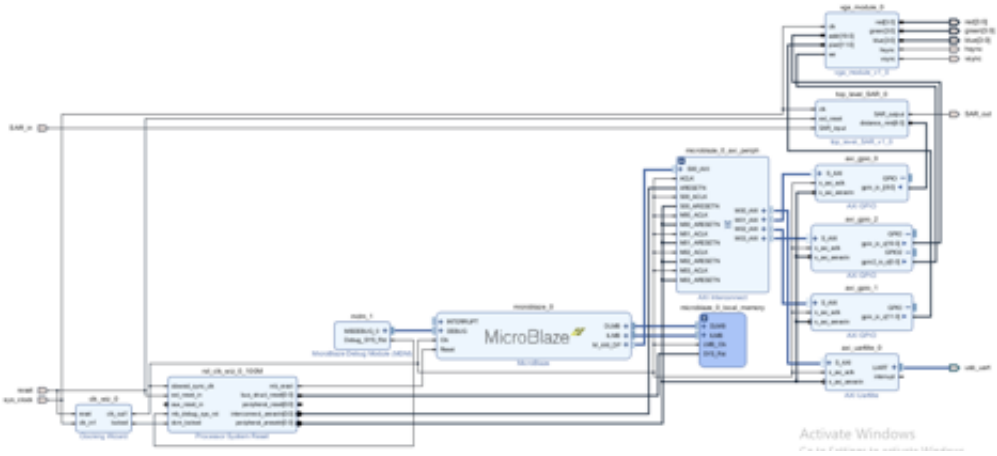


Figure 10

1.5.2 Xilinx SDK (C coding)

We had finally gotten to the level where we could command any pixel on the screen by making a single call to a function which would take the x coordinate and y coordinate and the pixel value. We were able to do this because Xilinx provides memory mapped I/O to peripherals and also has supporting C files to allow users to initialize and do I/O with their GPIO and UART. We could now get data and send data to other components such as the SAR module and the VGA module. After building all these levels of abstraction our hope was that we would implement a simple game such as the T-rex game for Google chrome where the distance sensor could control the jump. But being short on time and the need to catch up on other courses we had to decrease our scope. This was disheartening because after having built up all this abstraction we never fully got to utilize it. We converted numbers which we were images into hex values, which were placed into a data structure in C. We then put together some functions to set GPIO, read/write to GPIO, and control the framebuffer as to write the correct numbers onto the screen. At this our journey ended a bit too soon.

```
        }


        updateNumbers(currentValue);
    }

    return 0;
}

void setGPIO() {
    // intialize GPIO pins
    XGpio_Initialize(&gpio0, 0);
    XGpio_Initialize(&gpio1, 1);
    XGpio_Initialize(&gpio2, 2);

    // set GPIO directions
    XGpio_SetDataDirection(&gpio0, 1, 0xFFFFFFFF);
    XGpio_SetDataDirection(&gpio1, 1, 0x00000000);
    XGpio_SetDataDirection(&gpio2, 1, 0x00000000);
    XGpio_SetDataDirection(&gpio2, 2, 0x00000000);
}

void drawPixel(int x, int y, int color){
    if (x < 80 && y < 60) {
        XGpio_DiscreteWrite(&gpio2, 1, x + 80*y);
        XGpio_DiscreteWrite(&gpio1, 1, color);
        XGpio_DiscreteWrite(&gpio2, 2, 1);
    }
    XGpio_DiscreteWrite(&gpio2, 2, 0);
}
void drawImage(struct image anImage){
    for(int8_t y = 0; y < anImage.height; y++){
        // on the y th line
        for (int8_t x = 0; x < anImage.width; x++) {
            // on the x the line
            // pixel vertical address = (y + image.Y)
            // pixel horizontal address = (x + image.X)
            drawPixel(x + anImage.posX, y + anImage.posY, anImage.data[(x + (anImage.width)*y)]);
        }
    }
```

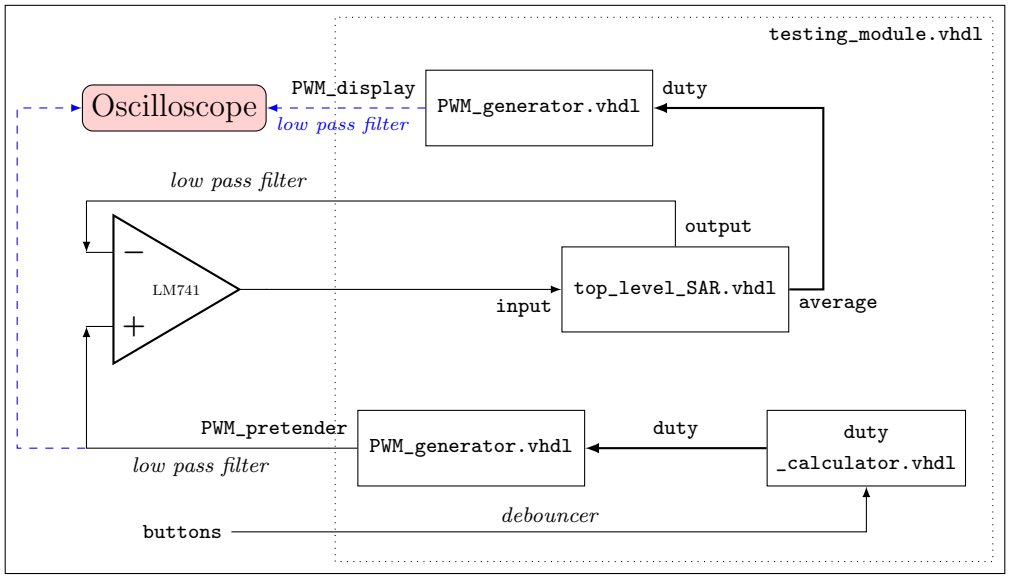


Figure 11: Top level overview of testing without sensors and displays

2 Lessons Learnt and Future Work

2.1 Testing an ADC without Sensors and Displays

This course emphasized the importance of testing our design at every possible stage of development. A testing strategy can also come in useful as designs become more sophisticated. We devised the testing setup shown in Figure 11 that allows full testing of an ADC using only an FPGA and an oscilloscope - no sensor, external voltage supply (except that's needed to supply the comparator) or a VGA display is required.

We already know how to generate a high quality analog signal using the FPGA. We can create a new PWM signal that will have a user-controlled duty cycle. We pass this signal through a suitable low pass filter and to the non-inverter input of the comparator instead of the sensor. As for the output, instead of displaying it on a screen or a seven segment display, we can feed it as the duty of another PWM signal generator, which will output a signal that can be read from a voltmeter. By having inputs and outputs (and intermediate) analog signals we are able to use the diagnostic powers of the oscilloscope for signal analysis. For example we can view the response to manual changes in real time or determine if our sample size is optimal. This testing procedure also separates the designer's task of optimizing their ADC design from the challenges and limitations that come with a sensor and display. Figure 11 describes the testing's top-level design. The top-level module is called `testing_module.vhdl` and can be found at the end of the paper.

3 Design Reflection

Working in Groups

- *Use your team members' skills and preferences to make life easier for yourself.* While some engineers prefer to foster and develop ideas on their own time and in their own space, working in teams allows them to share and demonstrate the soundness of their ideas.
- *Take the lead. Don't consistently find yourself watching over your partner's shoulder.* Although some people are naturally better programmers, that doesn't mean others can't be taking the lead on the project. Try taking the lead for a lab. Come prepared with a plan of action for yourself and the rest of your team. Use the Friday sessions that you have off to set the expectation for the week after.
- *Respect and trust your partners* Give your partners the respect and space they need to do work.

On VHDL

- *You're not coding - you're programming.* When you are writing VHDL code you are wiring logic gates and it's very easy to slip into the software mindset. If you want to implement something in VHDL you must first imagine what it would look like at the gate level.
- *Things are concurrent* Following on from the previous point, writing in VHDL means wiring actual hardware which means our code is actually describing pieces of hardware that are operating concurrently, which is of course far from the combinational logic of a high level language such as C.
- *It's not the speed-bumps, it's the pot holes.* Sometimes one can be so focused on their design that they overlook trivial errors that anyone with VHDL experience will tell you can take hours or no less than a fresh set of eyes to identify.
- Don't always assume that because a simulation works that a demonstration will be successful. Simulations are good for validating counts and arithmetic and checking synchronization but it has plenty of blind spots. Utilize Vivado's synthesis and implementation options to better understand your FPGA.

4 Appendix

4.1 BOM

Unit	Cost
2.2k Ohm Resistor	N/A
470 nF capacitor	N/A
Total	N/A

References

[1] J. Wagner, "Filtering PWM signals," 2009. Available at `http://ltwiki.org/images/8/82/PWM_Filters.pdf` Accessed: Nov. 12, 2017.

[2] Queen's University PHYS 352, "Analog-to-digital converters," Unknown Year. Available at `https://www.physics.queensu.ca/~phys352/lect08.pdf` Accessed: Dec. 1, 2017.