Vivado AXI Timer and Interrupts

Benjamin A. Blouin

*benjamin.blouin@temple.edu*

**Summary**

This lab uses the AXI Timer and interrupts on the Digilent Zybo embedded development board. It is built from the concepts learned through completing and executing the fourth section of chapter 2 of the Next Steps in Zynq SoC Design. The on-board buttons can increment or decrement the timer through an ISR; the current timer maximum or minimum count is shown (countChangeTimer) via on-board LEDs for two seconds. The on-board switches allow the buttons to change the countChangeTimer through control logic; switch and button zero increase the countChangeTimer up to a maximum of seven. Switch and button one decrease to a minimum of zero. Switch two allows for both behaviors but changes the buttons that carry out the behavior from zero and one to two and three, respectively. Switch three, if enable without any other input, enters a reset state, where the countChangeTimer is set to three. If, while in the reset state, another input happens either through switch, button, or any combination, an error state occurs. All LEDs blink at on and off for one second intervals, until all the other slide switches are off and no push buttons are being depressed. When slide switch three is turned off the program can return to normal operation.

**Introduction**

In embedded processing, an interrupt is a signal the processor receives that temporarily halts current activities. The current state the processor is in is saved, and then an interrupt service routine addresses the reason for the interrupt. There are many timers on the Zynq-7000 SOCs, and in the case of this lab, a PL fabric timer that is a built-in IP block implemented in the Vivado IDE and activated for behavior in the SDK.

Embedded design is generally used in real-time environments that need to react asynchronously, which cannot be temporally predicted. Interrupt requests for everyday computing include keyboards and mice, which generate an IRQ which must be serviced so the key click can become input, or the cursor can move smoothly. Using interrupts allows the processor to continue processing other instructions. The other way to service asynchronous events is polling, but this logic ties up the processor because it must continuously check input data. This leads to two outcomes: either the CPU can’t do other processors in an efficient timeframe, or the input is not real-time since the CPU is processing other instructions.

**Discussion**

The first project example is given, which includes a standard Zynq block, with the addition of the PmodAD and any blocks auto-generated. This design is then exported into the SDK, where an example main program is given in the PmodAD driver examples folder. The only change needed in the block design is to change to Pmod port from JA to JE; nothing needs to be changed in the main source code. Running the code outputs to the terminal SDK the current values on both channels in volts. All grounds must be shared to eliminate ground loop faults. With the Waveforms software outputting a DC voltage, readings are taken from zero to one volt in 0.1 V steps, at both the Waveform software oscilloscope and the SDK terminal. These values are used to determine what Vref and DC offset used for calibration. The oscilloscope is also used to read the chip select pin output to verify that the 50Mhz clock used for the ADC is correct, and then the source code is modified to remove some overhead. The values are read again, and there is no change. This shows that we can only change the sampling rate by use of moving window averaging, and not by clock manipulation. This is also how duty cycle and data rate are maximally limited. Windowed averaging is what controls the actual sampling rate.

The Vref global variable from the example is set to 3.3V, but after determining the calibration function, the vref global variable will be divided by two. Macros for window size and overlap for ease of modification testing. Global objects are needed for the gpio, pmod, raw data, and physical data. Global variables are made for vref, window array with a size of the previously defined macro, average, sum, sum of the overlap, a counter t, and dummy variable num. Function prototypes are defined for initializing the AD, initializing the gpio, and running the AD. These three functions are the only functions in main. Only one gpio object variable needs to be initialized, as there’s only one gpio block. The object has two channels, so each channel’s data direction register is set accordingly. The AD is initialized with its own function, and a one microsecond sleep command is required by the driver.

Switches are read, with the first behavior case being when the switches are all zeros. This prints out “ECE3622 Lab” to the terminal once every five seconds. The second case is started when the second switch from the right is enabled. The lab manual calls for this to happen when the first switch on the right is high, but my board’s switch is faulty, so the next switch is used. Reading the value from the ADC uses two built-in functions, and a flow control if-then cascade determines when the LEDs should be turned on or off. Then the behavior sleeps for one second.

The last case happens when the third switch from the right is high. A block diagram can be found in the appendix, figure 19. The first window is a special case which dictates the sequential specificity. The newest window is filled, and a sum is calculated. The average is then calculated. The first window is a special case because there is no overlap, and this attribute is taken care of by initializing the variable instantiation to zero. With the average having been calculated, we’ll now call this window the older window. The overlap sum is calculated with the old window. Now, when the while loop is restarted a new window is filled, a new window sum is calculated, and the average is calculated with the new window sum, and the old window overlap sum. These values are then reset, with a sum overlap calculated with the now old window. The new average, therefore, will always be the difference in new window sum and old window overlap.

Window choice is how sample rate is controlled. Picking a window size outside the Nyquist frequency results in a signal that doesn’t have enough information to be put back together. The overlap helps control hysteresis, where the value of a physical property (averaged signal) lags behind changes in the effect causing it (analog signal). The overlap helps smooth the transitional noise associated with everyday signals. In a perfect case scenario signals being sampled would be without noise.

The other reason for windowed average is, due to the central limit theorem, we have a better chance of discovering the original signal from the noise if we average out the noise. If the signal to be measured is either long enough in time to be sampled over many periods, or the sampling rate of the measurement device is fast enough, the original signal can be pieced back together regardless of the noise. The overlap represents a weighting of previous averages to compensate for a particularly noisy window. Another way this can be said is that it helps bring outlying averages closer to the best fit.

**Conclusions**

The lab was successful, as all conditions were met, and all logic was proved to be correct. The value of this lab is in showing that software overhead is so small that a FIR filter can be implemented in software because of the SOC being so data-bus efficient. This lab took longer than expected due to use of the xil\_printf versus the printf. The former can’t show float values, which led to code changes that probably worked but was thought to be faulty. Because the LIFO algorithm is just a modification of a standard FIFO stack the original code treated the filter as a special stack. The code was changed to be sequentially correct, but less intuitive to anyone else that might read the code. Extra care was added to the code by including explanatory comments beyond what would be needed if a stack naming and algorithm convention were used. My boards LSB switch is fault, which is why the switch choice was changed. The lab manual did not consistently keep its naming conventions, which led to confusion.

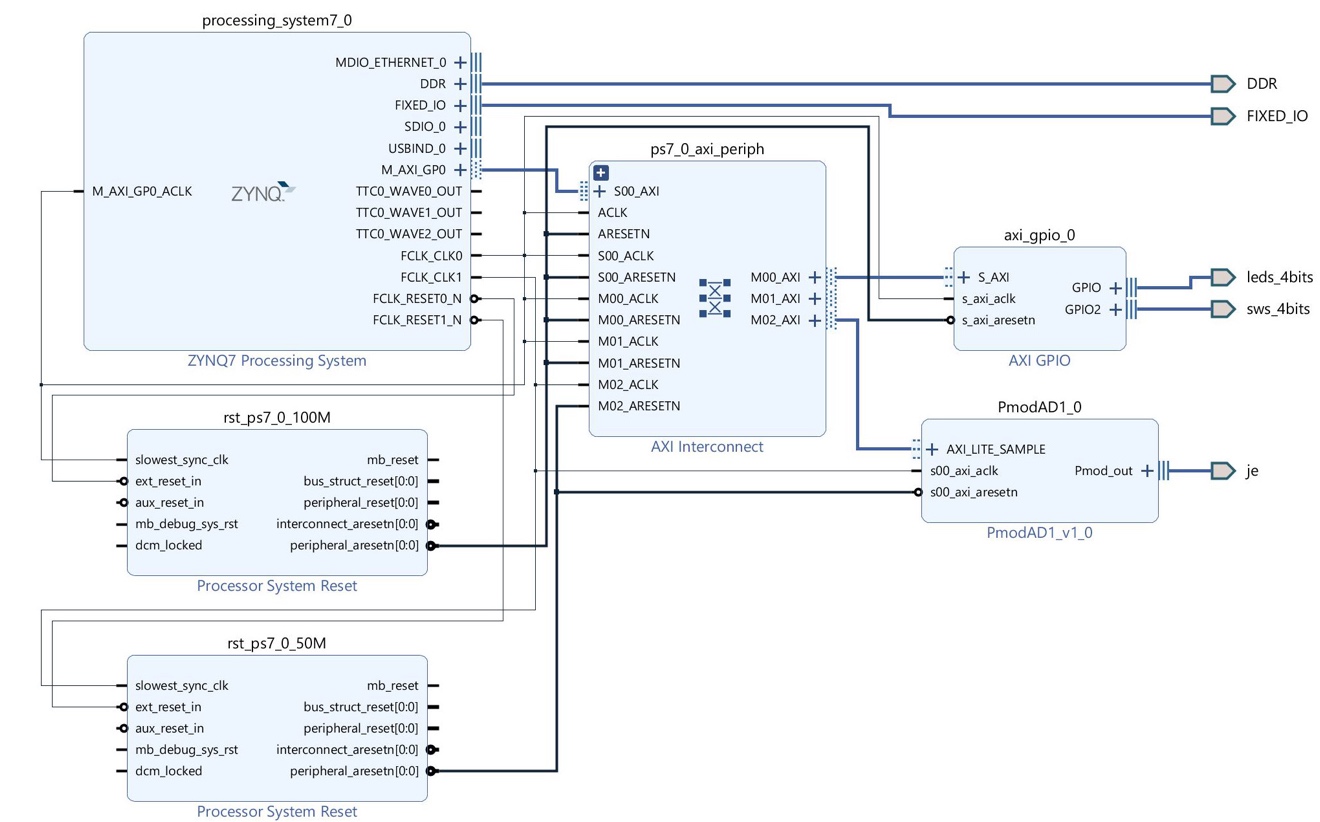
**Appendices**

Figure 1 Block Design

**Code**

A link was used to keep the aesthetic qualities coherent.

<https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/blob/master/bb_PmodAD1/bb_PmodAD1.sdk/bb_PmodAD1/src/main.c>

/\*

\* Embedded Design Lab

\* PmodAD1 Vivado SDK Project

\* ECE3623-003

\* Dennis Silage, PhD

\* Benjamin Blouin

\* 2/4/2019

\*

\* The PmodAD1 analog-to-digital converter (ADC) SPI peripheral from the Digilent Vivado IP Library

\* is configured and used to perform measurements on analog test signals.

\*

\* The lab uses the Digilent Electronic Explorer Board (EE Board)

\* to generate an analog test signal interfaced to channel 1 of the PmodAD1 ADC.

\*

\* SW0 OFF, SW1 OFF. The project prints ECE3622 Lab 2 on the SDK terminal every 5 seconds.

\*

\* SW0 is ON, SW1 is OFF. The input voltage signal from the AWG is a unipolar ramp

\* at 0.1 Hz (Waveforms indicates this as 100 mHz) with a range from 0 to 1 V.

\* The LEDs are to be ON if the voltage input exceeds the thresholds listed for the four LEDs:

\* LED0 0.2 V LED1 0.4 V LED2 0.6 V LED3 0.8V

\* SDK terminal output is a single string that identifies the event.

\*

\* SW0 is OFF, SW1 is ON. The input voltage signal from the AWG is again a unipolar ramp at 0.1 Hz with a range from 0 to 1 V.

\* The calculation is output to SDK Terminal every 1 second is the running average value of the voltage signal input.

\* The ramp and pulse unipolar signals are also to have symmetry differences

\* to validate the running average calculation.

\*

\*/

#include <stdio.h>

#include "PmodAD1.h"

#include "sleep.h"

#include "xgpio.h"

#include "xil\_io.h"

#include <xil\_types.h>

#include "xparameters.h"

// Macros

#define LEDS 1 //channel

#define SWITCHES 2 //channel

#define WINDOWSIZE 150 //moving average window size

#define OVERLAP 50 //Overlap of successive window

// Global Objects

XGpio LEDSWSInst; // GPIO Object for LEDS and Switches

PmodAD1 myDevice; // PmodAD1 Object for address

AD1\_RawData RawData; //AD1 Object to hold RawData

AD1\_PhysicalData PhysicalData; //AD1 Object to hold ActualData converted from RawData

// Global Variables

const float ReferenceVoltage = 3.3/2.0; //2.0 is vref offset calculated for calibration

float window[WINDOWSIZE]; //array to do moving averages

float avg = 0;

float sum = 0;

float sumOverlap = 0;

int t = 0;

int num = 0;

// Function prototypes

void ADInitialize();

void XGPIOInit();

void ADRun();

// Start main program

int main() {

ADInitialize();

XGPIOInit();

ADRun();

}

// Functions

void XGPIOInit() {

// One GPIO block with two channels

XGpio\_Initialize(&LEDSWSInst, XPAR\_AXI\_GPIO\_0\_DEVICE\_ID); // Pointer Init using devID

XGpio\_SetDataDirection(&LEDSWSInst,LEDS,0x00); //channel 1 output

XGpio\_SetDataDirection(&LEDSWSInst,SWITCHES,0xFF); //channel 2 input

}

void ADRun() {

int t = XGpio\_DiscreteRead(&LEDSWSInst,SWITCHES); //initial switch read

sumOverlap = 0; // used to reset old window overlap so it doesn't influence

// the new average when behavioral states are changed

while(1) {

// First case

if(t == 0) {

while(t == 0) {

xil\_printf("ECE3622 Lab 2\n");

sleep(5);

t = XGpio\_DiscreteRead(&LEDSWSInst,SWITCHES);

}

}

// Second Case

else if (t == 2) {

while(t == 2) {

AD1\_GetSample(&myDevice, &RawData); // Capture raw samples

// Convert raw samples into floats scaled to 0 - VDD

AD1\_RawToPhysical(ReferenceVoltage, RawData, &PhysicalData);

printf("Reading Output: %f\n",PhysicalData[0]);

if(PhysicalData[0] <= 0.2) {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,0);

}

else if(0.2 < PhysicalData[0] && PhysicalData[0] <= 0.4) {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,1);

}

else if(0.4 < PhysicalData[0] && PhysicalData[0] <= 0.6) {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,3);

}

else if(0.6 < PhysicalData[0] && PhysicalData[0] <= 0.8) {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,7);

}

else if(0.8 < PhysicalData[0] ) {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,15);

}

else {

XGpio\_DiscreteWrite(&LEDSWSInst,LEDS,0);

}

sleep(1);

t = XGpio\_DiscreteRead(&LEDSWSInst,SWITCHES);

}

}

// Third Case

else if (t == 4) {

while(t == 4) {

// Fill window

for (int t = 0; t<WINDOWSIZE; t++) {

AD1\_GetSample(&myDevice, &RawData); // Capture raw samples

// Convert raw samples into floats scaled to 0 - VDD

AD1\_RawToPhysical(ReferenceVoltage, RawData, &PhysicalData);

window[t] = PhysicalData[0];

}

// Sum of all values in window

for(int t = 0; t<WINDOWSIZE; t++) {

sum = window[t] + sum;

}

// Take the sum, remove the values to be removed (overlap),

// and divide by total number of samples in window

// Type-cast macro WINDOWSIZE because there's no explicit data type for macro

avg = (sum-sumOverlap) / (float) WINDOWSIZE;

sum = 0;

sumOverlap = 0;

// Summed Overlap, not the total sum, just the sum of values to be removed

// on next successive window array

for(int t = 0; t<OVERLAP; t++) {

sumOverlap = window[t] + sumOverlap;

}

printf("%f\n",avg); //output to terminal

sleep(1); //per lab

t = XGpio\_DiscreteRead(&LEDSWSInst,SWITCHES); //read switches in case of change

}

}

}

}

void ADInitialize() {

//Initialize the PmodAD1 device - note that the AD1 IP is free-running

AD1\_begin(&myDevice, XPAR\_PMODAD1\_0\_AXI\_LITE\_SAMPLE\_BASEADDR);

// Wait for AD1 to finish powering on, given in AD Technical Specifications

usleep(1); // 1 us (minimum)

}

**Analog Devices 12-/10-/8-Bit ADCs**

<https://www.analog.com/media/cn/technical-documentation/evaluation-documentation/AD7476A_7477A_7478A.pdf>

**Project**

Link to repository of entire project for author attribution integrity.

<https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/tree/master/bb_PmodAD1>

**Images**



Figure 2 EE Board Calibration



Figure 3 ADC Clock

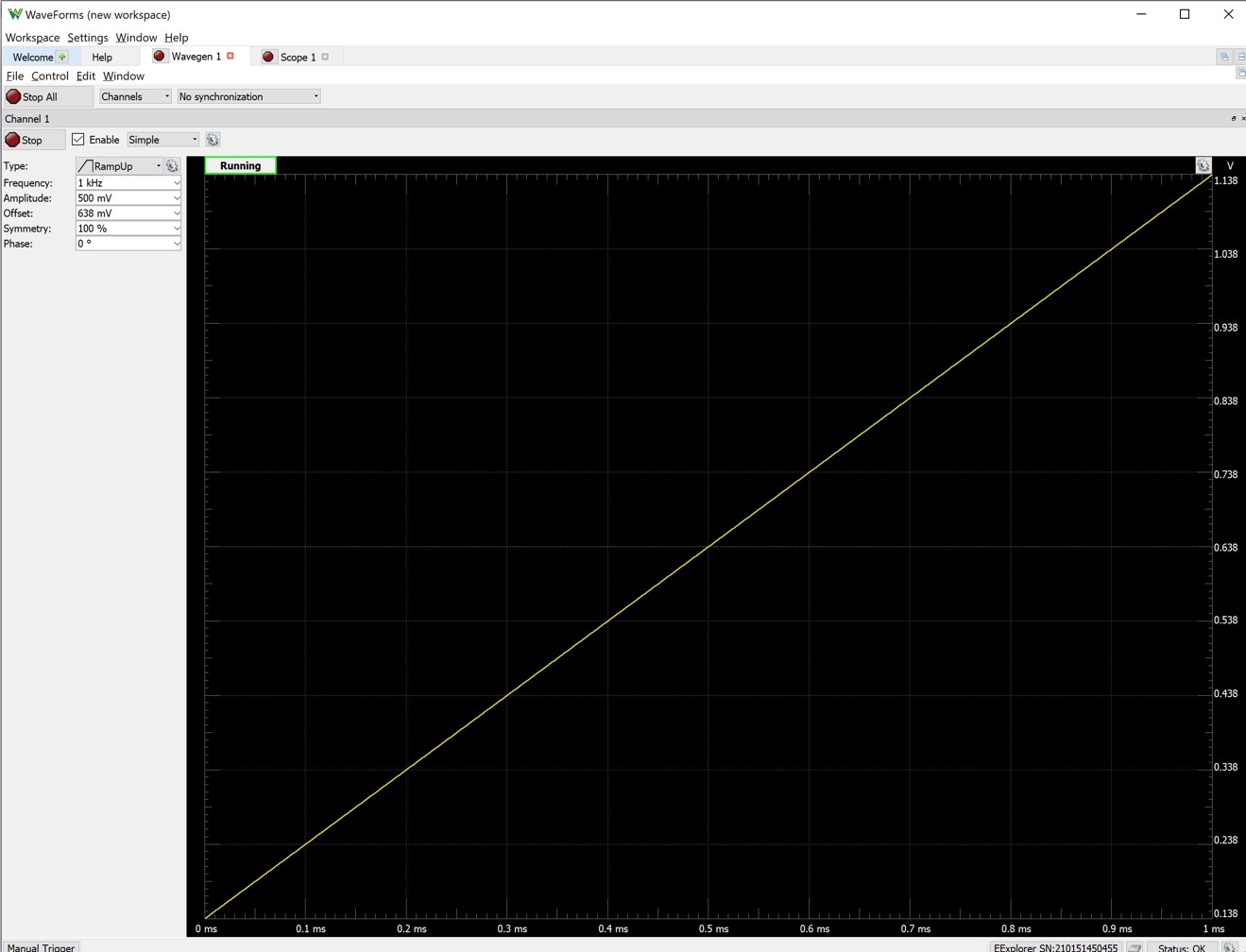
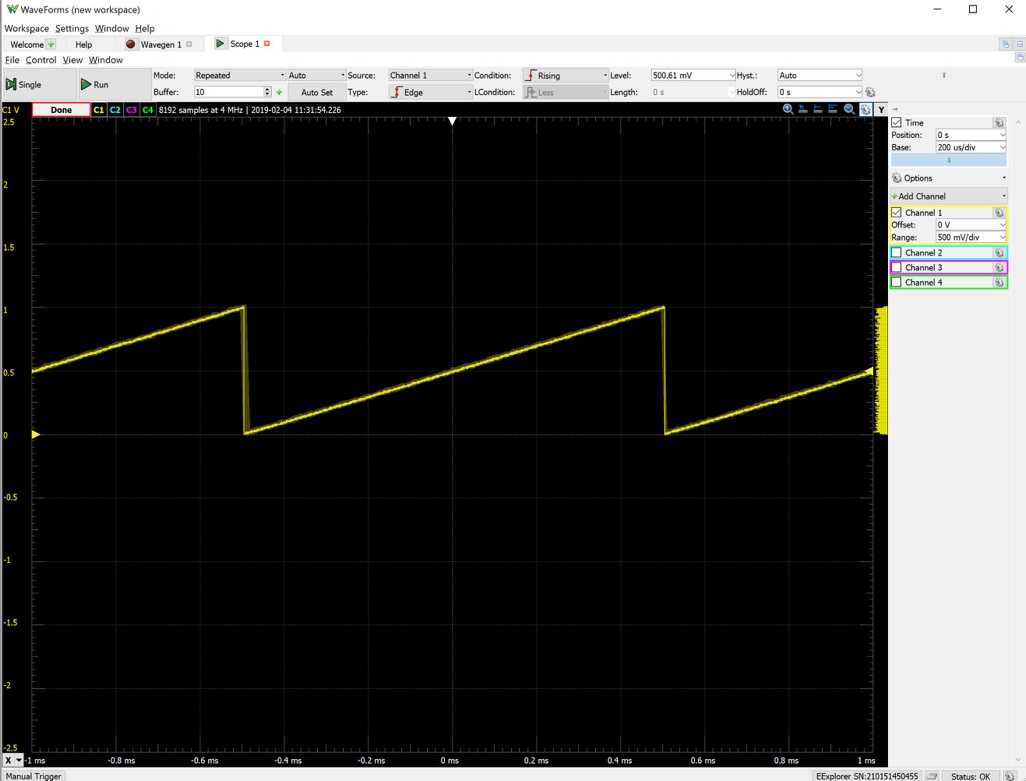


Figure 4 Calibrated Unipolar Ramp Reading from Waveforms

Figure 5 Calibrated Unipolar Ramp



Figure 6 Symmetric Ramp Window Size = 205 Overlap = 50

Figure 7 Symmetric Ramp Window Size = 105 Overlap = 50





Figure 8 Symmetric Ramp Window Size = 505 Overlap = 5

Figure 9 Symmetric Ramp Window = 255 Overlap = 5



Figure 10 Symmetric Ramp Size = 505 Overlap = 50

Figure 11 Symmetric Ramp Window Size = 705 Overlap = 150



Figure 12 Symmetric Ramp Window Size = 1005 Overlap = 5

Figure 13 Nonsymmetric Ramp Symmetry = 55% Window Size = 250 Overlap = 50



Figure 15 Nonsymmetric Impulse Symmetry = 55% Window Size = 250 Overlap = 50

Figure 14 Nonsymmetric Impulse Symmetry = 55% Window = 50 Overlap = 5



Figure 16 Nonsymmetric Sin Symmetry = 55% Window Size = 150 Overlap = 50

Figure 17 Nonsymmetric Sin Symmetry = 55% Window Size = 50 Overlap = 5

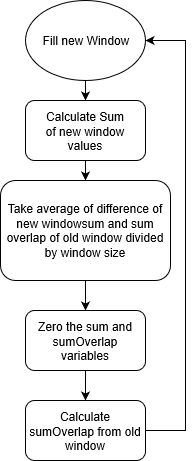
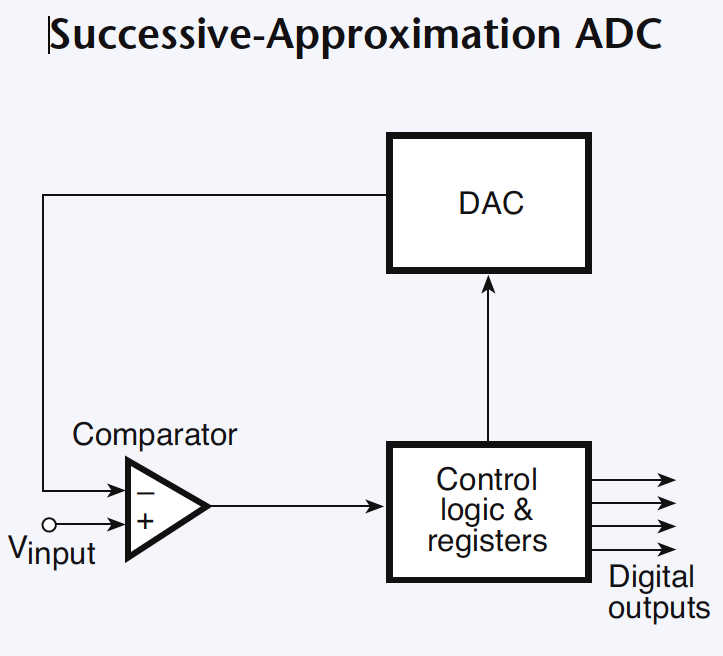
**Analog-to-Digital Background Information**

Figure 18 Averaged Moving Window Algorithm

Figure 19 Successive Approximation ADC

<https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/blob/master/bb_PmodAD1/bb_PmodAD1_data/Analog-to-Digital.pdf>