VHDL IMPLEMENTATION OF KALMAN FILTER

Abhishek Ratan Bhardwaj[1], Vashu Goel[2],
Deepanshu Garg[3], Ashwani kumar[4]
[1,2,3,4] Electronics and Communication Department,
Meerut Institute of Technology, Partapur Bypass, NH-58, Meerut (India)

ABSTRACT

The kalman filter estimates the state of a dynamic machine, even if the precise form of the system is unknown. The filter is very powerful in the sense that it supports estimations of past, present and even future states. A novel tracking algorithm presented in this paper is more advanced than other algorithms which have been presented before. The target tracking algorithm is implemented very clearly with the VHDL program.

Keywords: ***Kalman filter, VHDL, MATLAB, Precision Personal Locator, FPGA***

I INTRODUCTION

All the tracking algorithms are completely based on Kalman filtering which is recursive in nature. The Kalman filter has many applications, e.g. in dynamic positioning of ships where the Kalman filter estimates the position and the speed of the vessel and also environmental forces. These estimates are used in soft-sensor system used for supervision, in fault-detection systems and in model-based predictive controllers (MPCs) which is an important type of model-based controllers.
The Kalman filter is a recursive predictive filter that is based on the use of state space techniques and recursive algorithms. It estimates the state of a dynamic system. This dynamic system can be distributed by some noise, mostly assume as white noise. To improve the estimated state the Kalman filter uses measurements related to the state but distributed as well.

II LINEAR AND NONLINEAR MODELS

Kalman filter (KF), Extended Kalman Filter (EKF), unscented kalman Filter (UKF) and Particle Filter are mostly used for the state estimation.
The Kalman Filter plays a optimal role when the model is linear. The practical application of the Kalman Filter has its boundaries because normally most of the state estimation problems for eg. Tracking of the object(target) are non linear. On the other hand if the system is linear, the state estimation parameters like mean and covariance can be exactly updated with Kalman Filter.
The EKF has the principle that a linearized transformation is approximately equal to the true non linear transformation.
In this paper a simplified kalman filter is implemented to perform intertial system signal processing using a Hardware Description Language (HDL).

2.1 State Space Model

A state space model is a mathematical model of a process, where state x of a process is represented by a numerical vector. State space model (SSM) mainly consists of two sub models: The Process Model, which describes how the state propagates in time based on external influences, such as input and noise: and The Measurement Model, which describe how measurements z are taken from the process, typically simulating noisy and/or inaccurate measurements.

2.1.1 Linear State Space Model

A Linear State Space Model takes functions F and H as linear,in both state and input. The functions can then be expressed by using the matrices, B and H reducing state propagation calculations to linear algebra. Overall this results in the following state space model:

$$X_k = F_k \mathbf{x}_{k-1} + \mathbf{B_k u_{k-1}} + \mathbf{w}_{k-1}$$

$$Z_k = H_k \mathbf{x}_k + \mathbf{v}_k$$

Where

u is process input

w is state vector

v is measurement noise vector

k is discrete time

2.1.2 Nonlinear State Space Model

Non Linear Model is the most general form of state-space models. This model does typically consist of two functions, f and h:

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \mathbf{w}_{k-1})$$

$$\mathbf{z}_k = h(\mathbf{x}_k, \mathbf{v}_k)$$

III KALMAN FILTER

The problem of state estimation can be made tractable if we include certain constrains on the process model, by requiring both **f** and **h** to be linear functions, and the Gaussian and white noise terms **w** and **v** to be uncorrelated, with zero mean. As the model is linear and input is Gaussian, we know that the state and output will also be Gaussian .

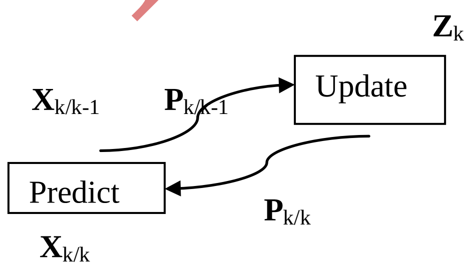


Fig1: Kalman Filter Loop

IV. IMPLEMENTATION

Implementing a Kalman Filter design on an FPGA is difficult. Although the Matlab code contains the necessary calculations to simulate the performance of this Kalman Filter, some functions which are simple in Matlab become quite complex when implementing them in a hardware description language. Functions like matrix multiplication, sine functions, and division, which are easily executed in software programs, present a challenge when creating them in hardware. Since the design was quite complicated, it was determined that the best approach was to break the design down into small pieces.

4.1 Matrix Multiplications

The first obstacles presented were the three matrix multiplications. There were certain instances where 3x3 matrix multiplications is required. Some of the multipliers are equipped with by single multiplications. So there is not enough multipliers are available in the FPGA. We came to the conclusion that we had to multiplex what values are being multiplied at certain times, so that each multiplier could be used more than once.
The entire 3x3 matrix remains the same for these calculations but the row of 3 is what is getting multiplexed for another 3 x 3 matrix. A module was created to perform the 9 multiplications between a 1x3 matrix and a 3x3 matrix. In order to perform a matrix multiplication with two 3x3 matrices, this module needs to be used 3 times.
For the multiply function we have used the logic of complementing the negative value then we get the 32 bit product of two 16bit operand. This multiplier function is used for the matrix multiplication.

*prod1 <= multiply(**row1,col11)** + multiply(**row2,col21)** + multiply(**row3,col31)**;*
*prod2 <= multiply(**row1,col12)** + multiply(**row2,col22)** + multiply(**row3,col32)**;*
*prod3<=multiply(**row1,col13)**+multiply(**row2,col23)**+ multiply(**row3,col33)**;*

4.2 Division

Another task that we needed to perform was dividing two values within the Kalman Filter. Performing division is a difficult task because it takes a lot VHDL code and uses a lot of resources. It was decided that the best approach was to use the built in core generator in the Xilinx software that the VHDL design was being written in. The core generator can create a number of different functions and it uses an efficient amount of resources. Once we created this module, we worked to include it within our Kalman Filter design.

4.3 Using Registers to Store Previous Values

The use of registers was very important to this design. Since the design uses previous values in the current calculations, the previous values needed to be stored in registers. Also, the values in these registers have to be loaded into the design along with the input. Without using registers and loading values in on each clock cycle, the design would cause a continuous loop. This happens because as the output changes, the current calculations would change causing the output to change again, and this would keep happening.
On the rising edge of the clock, or for testing purposes, when a button is pressed, flip flops load the previous output values as well as the current input. After going through the next state logic, the output values are stored in the registers and remain there until the next load.

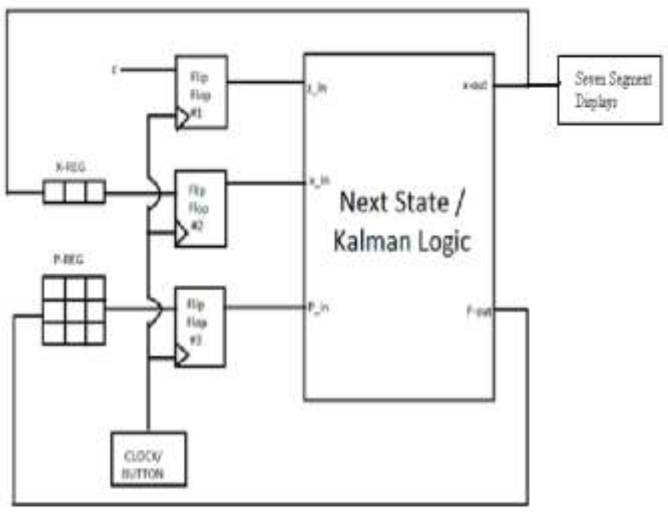


Fig2: Main Block Diagram

4.4 Kalman Filter Using Real Time Signals

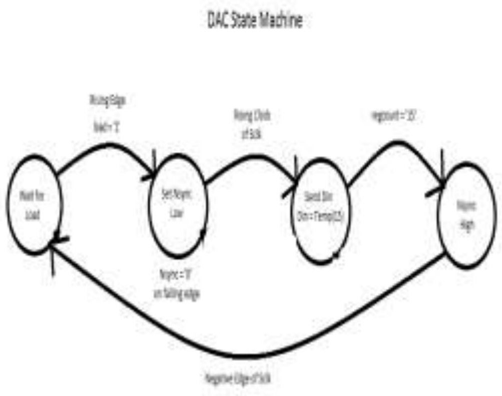
The final goal is to be able to send an analog signal into the project, and output the resulting analog signal. To successfully do this, it is necessary to use an analog to digital converter (ADC) and digital to analog converter (DAC) to stream data into and out of the system using analog signals.

The next step in our project was to integrate the DAC. If we used the same inputs that the MATLAB program uses, then we could examine how well our Kalman Filter and DAC were working together by comparing the oscilloscope output with the Matlab graph.

```
-- Control for temp
process(clk)
begin
        if rising_edge(clk) then
            if count = 0 then
                if load = '1' then
                      temp <= data;
            elsif currentstate = Send then
                      temp <= temp(14 downto 0) & '0';
                 end if;
             end if;
        end if;
end process;

-- Next State Logic
process(currentstate, load, regcount)
begin
         case currentstate is
               when Idle =>
                      if load = '1' then
                             nextstate <= Low;
                      else
                             nextstate <= Idle;
                      end if;
                when Low =>
                       nextstate <= Send;
                when Send =>
```

```
                    if regcount = 15 then
                          nextstate <= High;
                    else
                          nextstate <= Send;
                    end if;
               when High =>
                          nextstate <= Idle;
          end case;
end process;
```



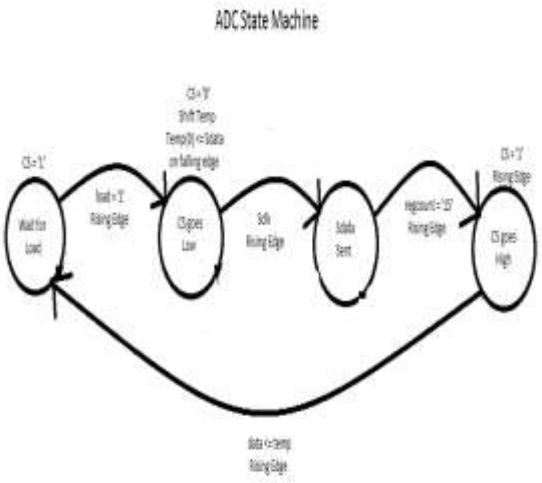


Fig3: State Machine Of DAC **Fig4: State Machine Of ADC**

```
--Next state logic
process(currentstate, load, regcount)
begin
        case currentstate is
              when Idle =>
                  if load = '1' then
                      nextstate <= CSLow;
                  else
                      nextstate <= Idle;
                  end if;
              when CSLow =>
                      nextstate <= Receive;
              when Receive =>
                  if regcount = 15 then
                      nextstate <= CSHigh;
                  else
                      nextstate <= Receive;
                  end if;
              when CSHigh =>
                      nextstate <= Idle;
           end case;
   end process;

-- Control for temp
process(clk)
begin
      if rising_edge(clk) then
            if nextstate = Receive then
                  if count = 0 then
                      temp <= temp(14 downto 0) & Sdata;
```

```
                    elsif count = 3 then
                         temp(0) <= Sdata;
                    end if;
               end if;
          end if;
end process;
```

-- Load 12 data bits to data

```
process(clk)
begin
          if rising_edge(clk) then
               if currentstate = CSHigh then
                         data <= temp(11 downto 0);
               end if;
          end if;
end process;
```

CS is what controls when a sample is taken. CS is active low and tells the ADC to create a 16 bit value out of the analog sample. Like the DAC, we created a load signal to tell the ADC controller that an input is desired. When load is high, CS goes low on the rising edge of sclk. This allows for the setup time to be achieved before the first value is input on the falling edge of the clock. temp shifts in one bit at a time on the rising edge of sclk and after 15 cycles, temp is ready to output a 16 bit value, and CS is sent high.

The values are available on the falling edge of sclk, but taking them on the rising edge assures that they are valid. The only bit that is taken on the falling edge is the first bit, and this is because it is sent along with the second bit on the first falling edge. The values of Sdata were shifted into temp.

4.5 Implementation of ADC to DAC

In this there will be sampling at 25 kHz. So before we connect our Kalman Filter, we wanted to make sure we can sample at this rate and still produce the correct output. In this module, a state machine was created to sample the input signal, and send the output to the DAC.

On each rising edge of the 25 kHz clock, the loads for both the ADC and DAC controller are sent high. The next couple of states determine when the loads of each controller should be sent back low. These states tell the ADC controller to sample a signal, and the DAC controller to output the digital value that is given. This repeats every rising edge of the 25 kHz clock.

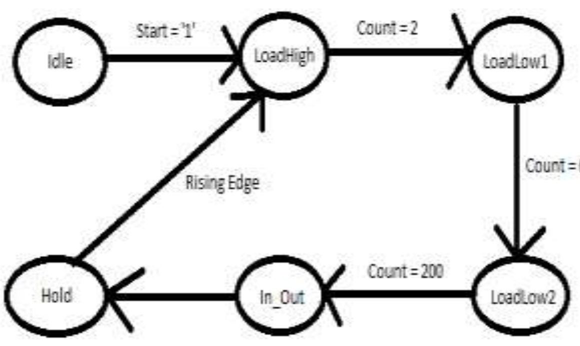


Fig5: State Machine of ADC to DAC

```
-- Next State Logic

process(currentstate, start, count25K)
begin
    case currentstate is
        when Idle =>
            if start = '1' then
                nextstate <= In_Out;
            else
                nextstate <= Idle;
            end if;
        when In_Out =>
            if count25K = 20 then
                nextstate <= LoadHigh;
            else
                nextstate <= In_Out;
            end if;
        when LoadHigh =>
            if count25K = 22 then
                nextstate <= LoadLow;
            else
                nextstate <= LoadHigh;
            end if;
        when LoadLow =>
            if count25K = 26 then
                nextstate <= Hold;
            else
                nextstate <= LoadLow;
            end if;
        when others =>
            if count25K = 2000 then
                nextstate <= In_Out;
            else
                nextstate <= Hold;
            end if;
    end case;
end process;
```

V Conclusion

In conclusion, we have created a successful Kalman Filter, which interfaces with an ADC and DAC to form a complete system that streams analog data in and out. We also created an ADC controller and DAC controller so that the Kalman Filter, ADC and DAC could be integrated together and used for testing purposes.

A complete system like the one we have built can be altered, and added onto, to perform the tasks of the Kalman Filter in the PPL(Precision Personal Locator) system, and can be included within the implementation of the actual system to process the data in real time. What all of this means is that another group can learn from everything that has been documented here to enhance our design to support a more complicated version of a Kalman Filter.

REFRENCES

[1]. Mohinder singh Grewal, Angus P. Andrews, “Kalman filtering: Theory and Practice using VHDL” , John Wiley & Sons, 2007,pp 8-32

[2]. Peter Maybeck, Stochastic Models, Estimationd and Control, Academic Press. Inc, 1979.

[3]. Grewal, Mohinder S., and Angus P. Andrews (1993). Kalman Filtering Theory and Practice using VHDL, Upper Saddle River, NJ USA, Prentice Hall.

[4]. B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. New York: Dover, 2005.

[5]. R. E. Kalman, “A new approach to linear filtering and prediction problems,” *J. Basic Eng.*, vol. 82, no. 1, pp. 35–45, Mar. 1960.

[6]. Alejandro Ribeiro , Ioannis D. Schizas, Stergios I. “Kalman filtering in VHDL ” IEEE Control System Magazine , pp 66-87, April 2010

[7]. Google

[8]. Wikipedia

[9]. Mahdy Nabaee, Ali Pooyafard, Ali Olfat “ Enhanced Object Tracking with Received Signal Strength using Kalman Filter in VHDL field” 2008 IEEE International Symposium on Telecommunications, pp 318- 324.

[10]. Y. Bar-Shalom and X. R. Lin, "Estimation and tracking: principles, techniques, andsoftware", Artech House, 1993, pp. 417-483.

[11]. F. A. Faruqi and R. C. Davis, “Kalman filter design for target tracking”, IEEE Trans. Aerosp. Electron. Syst., AES-16, pp. 500-508, 1980

[12]. L. P. Maguire and G. W. Irwin, “Transputer implementation of Kalman filters”, IEEE Proc.-D, Vol. 138, pp. 355-362, 1991