A Novel Pipelined Architecture for 4X4 Inversion Matrix of Kalman Filters Using Verilog in ASIC

V.K. Dinesh Prabu[1]
Research Scholar, Anna University, Chennai, Tamil Nadu, India,
Assistant Professor, Department of Electrical & Electronics Engineering,
S.K.P. Engineering College, Tiruvannamalai, Tamil Nadu, India.
E-mail: dineshprabuvk@yahoo.co.in. Contact Number: 9952656528

Dr.C.Kumar[2]
Director Academic, Senior Member IEEE, Department of Electrical & Electronics Engineering,
Sri Rangaboopathi College of Engineering, Gingee, Tamil Nadu, India.
E-mail:drchkumararima@gmail.com. Contact Number: 9443266567

Dr.P.Suresh[3]
Associate Professor, Department of ECE, Veltech Rangarajan Dr.Sakunthala R&D Institute of Science and Tech.Chennai-62.
Tamil Nadu. Contact No: 9940186845

Abstract: *A novel pipelined systolic array-based architecture for 4X4 matrix inversion is proposed. It is suitable for ASIC implementations as it is used for in Kalman filters. The 4X4 matrix inversion is implemented in verilog language for enabling the user for different size of Kalman filters suitable for different applications. It is scalable for different matrix size and as such allows employing parameterization that makes it suitable for customization for application-specific needs. The proposed architecture consists of pipeline registers, an innovative logic control unit, and a segmented Look up Table division scheme. This new proposed architecture has an advantage of reduced processing element complexity. The ASIC implementation architecture is useful to enable the novel pipelined systolic array for the quickest operation of Kalman filter. The precision error resulted is in the allowable range and it does not affect the performance of the overall system.*

Key words: *Systolic Array, Kalman Filter, Look up Table, Verilog, FPGA, ASIC, Floating Point Multiplier,*

1. Introduction

Many DSP algorithms such as Kalman filter involve several iterative matrix operations, the most complicated begin matrix inversion, which requires O (n^2) computations (n is matrix size). This becomes the critical bottleneck of the processing time in such algorithms. Kalman filters have been widely used in many applications such as target tracking, navigation systems, adaptive control and many other dynamic systems. Kalman filter algorithm is based on minimizing the mean-square error recursively. The algorithm of an adaptive Kalman filter involves several iterative matrix manipulations such as matrix inversion, multiplication, addition and subtraction. Real-time implementation of Kalman filters is hence limited by the computationally extensive nature of the algorithm. Many attempts have been made to employ various systolic architectures for VLSI implementation of Kalman filters [1],Systolic-based architectures should be modified to meet hardware requirements of the Field Programmable Gate Array (FPGA) technology [2]. This paper investigates the design and hardware implementation of a generic Kalman filter in verilog language where user is able to set the parameters to change the state number of the filter. Here for the 4X4 matrix has been implemented in verilog and speed and area are optimized.

2. Kalman Filter

The Kalman filter uses a system's dynamics model (e.g., physical laws of motion), known control inputs to that system, and multiple sequential measurements (such as from sensors) to form an estimate of the system's varying quantities (its state) that is better than the estimate obtained by using only one measurement alone. As such, it is a common sensor fusion and data fusion algorithm.
Kalman filter is used to remove noise from a signal, Many physical processes, such as a vehicle driving along a road, a satellite orbiting the earth, a motor shaft driven by winding currents, or a sinusoidal radio-frequency carrier signal, are the linear systems which uses the application of Kalman filter. The main feature is that only the previous state estimate and the new input data's are

required to generate the new state estimate in each computation cycle, which results in a low memory requirement [7]. Kalman filter algorithm has two basic operations; prediction and filtering, both executing in a single cycle recursively

State equation:

$$x_k+1^{=}Ax_k+ Bu_k+ w_k \longrightarrow \quad (2.1)$$

Output Equation:

$$y_k= Cx_k+ z_k \longrightarrow \quad (2.2)$$

In the above equations *A*, *B*, and *C* are matrices; *k* is the time index; *x* is called the state of the system; *u* is a known input to the system; *y* is the measured output; and *w* and *z* are the noise. The variable *w* is called the process noise, and *z* is called the measurement noise. Thus, this paper concentrates on FPGA implementation of matrix inversion, matrix division which is in fact the "heart" of Kalman filter. In the above equations *A*, *B*, and *C* are matrices; *k* is the time index; *x* is called the state of the system; *u* is a known input to the system; *y* is the measured output; and *w* and *z* are the noise. The variable *w* is called the process noise, and *z* is called the measurement noise. Each of these quantities is (in general) vectors and therefore contains more than one element. The vector *x* contains all of the information about the present state of the system, but we cannot measure *x* directly. Instead, we measure *y*, which is a function of *x* that is corrupted by the noise *z*. knowing that the measured output is equal to the position, we can write our linear system equations as follows:

$$X_{k+1} = \begin{pmatrix} 0 & 1 \\ 1 & T \end{pmatrix} X_k + \begin{pmatrix} T \\ T^2/2 \end{pmatrix} \longrightarrow \quad (2.3)$$

$$Y_k = [1\ 0]\ X_k + Z_k \longrightarrow \quad (2.4)$$

Z_k is the measurement noise due to such things as instrumentation errors. If we want to control the vehicle with some sort of feedback system, we need an accurate estimate of the position *p* and the velocity *v*. need a way to estimate the state *x*. This is where the Kalman filter is used.

First, the average value of the state estimate to be equal to the average value of the true state. That is why the estimate is to be biased one way or another. Mathematically, we would say that the expected value of the estimate should be equal to the expected value of the state. Second, we want a state estimate that varies from the true state as little as possible. That is, not only do we want the average of the state estimate to be equal to the average of the true state, but we also want an estimator that results in the smallest possible variation of the state estimate mathematically, we would say that we want to find the estimator with the smallest possible error variance. It so happens that the Kalman filter is the estimator that satisfies these two criteria. But the Kalman filter solution does not apply unless we can satisfy certain assumptions about the noise that affects our system. Remember from our system model that *w* is the process noise and *z* is the measurement noise. We have to assume that the average value of *w* is zero.

2.1 Kalman Gain Equations:

$$K_k=AP_kC^T\ (CP_kC^T+S_z)^{-1} \longrightarrow \quad (2.5)$$

$$X_{k+1}=(AX_k+ BU_k)+K_k(Y_{k+1}- CX_k) \longrightarrow \quad (2.6)$$

$$P_{k+1}=AP_kA^T+S_w- AP_kC^TS_z^{-1}CP_kA^T \longrightarrow \quad (2.7)$$

That's the Kalman filter. It consists of three equations, each involving matrix manipulation. In the above equations, a *–1* superscript indicates matrix inversion and a *T* superscript indicates matrix transposition. The *K* matrix is called the Kalman gain, and the *P* matrix is called the estimation error covariance. Hence the implementation of this filter in hardware is a great bottle neck due to inversion operation; hence systolic array is used to overcome this problem.

3. **Floating Point Multiplier:**

Floating-point algorithms are used frequently in modern applications such as speech recognition, image processing and financial engineering because of its ability to represent a good approximation to the real numbers. The IEEE 754 floating point standard [ANS85] has been widely accepted for representing floating point numbers. With this standard, the result and the error of each floating-point operation can be retained the same even if the platform of the computation is changed. The floating-point arithmetic, including addition, subtraction and multiplication is covered in this chapter. The rounding error imposed by using floating-point arithmetic will be discussed. The concepts of quantization error between IEEE standard and the variant used in this thesis will be introduced.

With the increasing size of FPGA devices, implementing floating point arithmetic on FPGAs are now possible. However, as the size of the FPGA is still limited, a carefully designed floating-point implementation is essential. In custom hardware designs, there is always trade-off between connecting requirements of performance, area and quantization error to be addressed. For example, area can usually be reduced if a larger quantization error is allowed for a hand-held application. It would be desirable to allow a program to automatically determine the minimum exponent and fraction sizes required for each signal to reach some user-specified quantization error. A floating point library called float is presented to enable users to optimize the design. In addition, a library, which can generate arbitrary sized floating-point adders and multipliers, was developed to facilitate the FPGA-based floating-point applications. The first section will discuss the software aspect of this system. An example using floating-point tools to develop and optimize a digital sine-cosine compiler is presented. To

generate an arbitrary sized of floating point operator, a Perl program has been developed as a Verilog generation module.

4. FPGA:

Field Programmable Gate Arrays (FPGAs) can be used to implement just about any hardware design. One common use is to prototype a lump of hardware that will eventually find its way into an ASIC. However, there is nothing to say that the FPGA can't remain in the final product. Whether or not it does will depend on the relative weights of development cost and production cost for a particular project. (It costs significantly more to develop an ASIC, but the cost per chip may be lower in the long run. The cost tradeoff involves expected number of chips to be produced and the expected likelihood of hardware bugs and/or changes. This makes for a rather complicated cost analysis, to say the least.)

The development of the FPGA was distinct from the PLD/CPLD evolution just described. This is apparent when you look at the structures inside. Figure 2 illustrates a typical FPGA architecture. There are three key parts of its structure: logic blocks, interconnect, and I/O blocks. The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable input, output, or bi-directional access to one of the general-purpose I/O pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of logic blocks. And connecting logic blocks to logic blocks and I/O blocks to logic blocks is the programmable interconnect wiring.

The logic blocks within an FPGA can be as small and simple as the macrocells in a PLD (a so-called fine-grained architecture) or larger and more complex (coarse-grained). However, they are never as large as an entire PLD, as the logic blocks of a CPLD are. Remember that the logic blocks of a CPLD contain multiple macrocells. But the logic blocks in an FPGA are generally nothing more than a couple of logic gates or a look-up table and a flip-flop.

Because of all the extra flip-flops, the architecture of an FPGA is much more flexible than that of a CPLD. This makes FPGAs better in register-heavy and pipelined applications. They are also often used in place of a processor-plus-software solution, particularly where the processing of input data streams must be performed at a very fast pace. In addition, FPGAs are usually denser (more gates in a given area) and cost less than their CPLD cousins, so they are the de facto choice for larger logic designs.

5. Hardware Design and Development:

The process of creating digital logic is not unlike the embedded software development process you're already familiar with. A description of the hardware's structure and behavior is written in a high-level hardware description language (usually VHDL or Verilog) and that code is then compiled and downloaded prior to execution. Of course, schematic capture is also an option for design entry, but it has become less popular as designs have become more complex and the language-based tools have improved. The overall process of hardware development for programmable logic is shown in Figure 3 and described in the paragraphs that follow.

Perhaps the most striking difference between hardware and software design is the way a developer must think about the problem. Software developers tend to think sequentially, even when they are developing a multithreaded application. The lines of source code that they write are always executed in that order, at least within a given thread. If there is an operating system it is used to create the appearance of parallelism, but there is still just one execution engine. During design entry, hardware designers must think-and program-in parallel. All of the input signals are processed in parallel, as they travel through a set of execution engines-each one a series of macrocells and interconnections-toward their destination output signals. Therefore, the statements of a hardware description language create structures, all of which are "executed" at the very same time. (Note, however, that the transference from macrocell to macrocell is usually synchronized to some other signal, like a clock.)

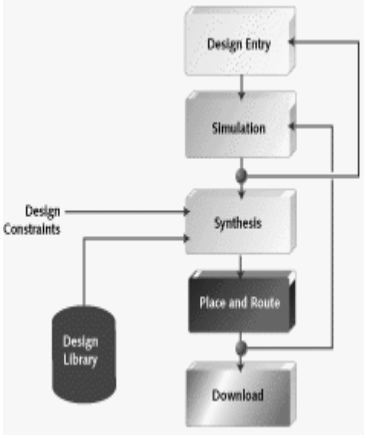


Fig 5.1 Programmable logic design process

Typically, the design entry step is followed or interspersed with periods of functional simulation. That's where a simulator is used to execute the design and confirm that the correct outputs are produced for a given set of test inputs. Although problems with the size or timing of the hardware may still crop up later, the designer can at least be sure that his logic is functionally correct before going on to the next stage of development. Compilation only begins after a functionally correct representation of the hardware exists. This hardware compilation consists of two distinct steps. First, an

intermediate representation of the hardware design is produced. This step is called synthesis and the result is a representation called a netlist. The netlist is device independent, so its contents do not depend on the particulars of the FPGA or CPLD; it is usually stored in a standard format called the Electronic Design Interchange Format (EDIF).

The second step in the translation process is called place & route. This step involves mapping the logical structures described in the netlist onto actual macrocells, interconnections, and input and output pins. This process is similar to the equivalent step in the development of a printed circuit board, and it may likewise allow for either automatic or manual layout optimizations. The result of the place & route process is a bit stream. This name is used generically, despite the fact that each CPLD or FPGA (or family) has its own, usually proprietary, bit stream format. Suffice it to say that the bit stream is the binary data that must be loaded into the FPGA or CPLD to cause that chip to execute a particular hardware design.

Increasingly there are also debuggers available that at least allow for single-stepping the hardware design as it executes in the programmable logic device. But those only complement a simulation environment that is able to use some of the information generated during the place & route step to provide gate-level simulation. Obviously, this type of integration of device-specific information into a generic simulator requires a good working relationship between the chip and simulation tool vendors.

6. Look up Division Scheme:*A. Division with multiplication:*Scalar division represents the most critical arithmetic operation within a processing element in terms of both resource utilization and propagation delay. This is particularly typical for FPGAs, where a large number of logic elements are typical used to implement division. For the efficient implementation of division, which still satisfies accuracy requirements, an approach with the use of LUT and an additional multiplier has been proposed and implemented. The numerical result of a divided by b is the same as a multiplied by 1/b, the FPGA built-in multiplier can be used to calculate the division if an LUT of all possible values of 1/b is available in advance. FPGA device provide a limited amount of memory, which can be used for LUTs. Due to the fact that 1 and b can be considered integers, the value of 1/b falls into decreasing hyperbolic curve, while b tends to one, and so the value difference between two consecutive numbers of 1/b decreases dramatically. To reduce the size of LUT, the inverse value curve can be segmented into several sections with different mapping ration. This can be achieved by storing in inverse value, the median of the group of consecutive values of b. on an altera APEX device, when combining the LUT and multiplier into a single division module, a 16 bit by 26 bit multiplier consumes 838 logic elements, operating at 25MHz clock frequency and total memory consumption of 53248 memory bits for the specific target FPGA device. The overall speed improvement achieved through using the DLM method is 3.5 times when compared to using a traditional divider. Because of the extra hardware required for efficiently addressing the LUT, the improvement in terms of LEs is rather modest. The hardware-based divider supplied by altera, configured as 16 bit by 26 bit, consumes 1123 Les when it is synthesized for the same APEX device.

*A. Optimum Segmentation Approach:*Since **b** is a 16-bit number in 1.15 format, there are totally $(2^{15} - 1)$ = 32767 different values of **1 / b**. Table 2 presents the mapping ratios for five different segmentation methods, namely *Seg-1* to *Seg-5*. Since the value of **1 / b** retrieved from the LUT is then multiplied by a, any precision error will be magnified. Therefore, it is important to consider the worst-case error. Table 5.1 presents a comparison of the various mapping schemes in Table 5.2. Table 5.1.Segmentation of 1/b

Name	Segmentation	Mapping ratio
Seg 1	No segmentation	1:8
Seg 2	1 - 1023 1024 – 2047 2048 – 4095 4096 – 8191 8192 – 32767	1:1 1:2 1:4 1:8 1:16
Seg 3	1-511 512 – 1023 1024 – 2047 2048 – 4095 4096 – 8191 8192 – 16383 16384 – 32767	1:1 1:2 1:4 1:8 1:16 1:32 1:64
Seg 4	1-511 512 – 1023 1024 – 2047 2048 – 4095 4096 – 8191 8192 –16383 16384 – 24575 24575-32767	1:1 1:2 1:4 1:8 1:16 1:32 1:64 1:128

Seg 5	1-511 512 – 1023 1024 – 2047 2048 – 4095 4096 – 8191 8192 – 9362 9363 – 13107 13108 – 21845 21846 – 32767	1:1 1:2 1:4 1:8 1:16 map to constant value 1/4 map to constant value 1/3 map to constant value 1/2 map to constant value 1

Table 5.2 Comparison of the Segmented 1/b

	Seg1	**Seg2**	**Seg3**
Average error	0.13%	0.78%	**0.07%**
Worst case error	300%	3.03%	**0.21%**
Resource(bit)	53248	81920	**53248**

Figures in Table 5.2 suggest *Seg-3* would be the natural choice for composition of the LUT. The LUT holds 4096 inverse values with a 26-bit word length in 16.10 data format.

6. Systolic Array:

Systolic architecture represents a network of processing elements (PEs) that rhythmically compute and pass data through the system. This PEs is regularly pump data in and out such that a regular flow of data. As a result, systolic systems feature modularity and regularity, which are important properties of VLSI design.

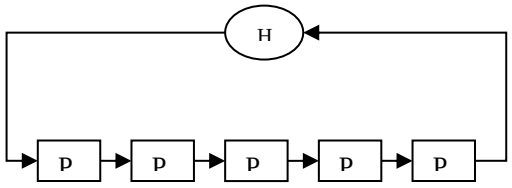


Figure 6.1: Basic systolic array

Systolic architectures are designed by using linear mapping techniques on regular dependence. A dependence graph (DG) is said to be regular if the presence of an edge in a certain at any node in the DG represents presence of an edge in the same direction at all nodes in the DG.

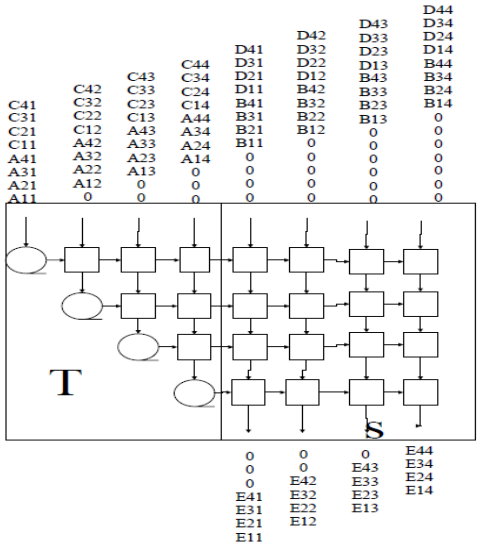


Fig.6.2: Systolic Array

7. Discussion of Results:

A. RTL Schematic of Systolic Array:

A systolic array which reduces the complexity of O(2n) from $O(n^2)$, where n is size of the matrix. Designing of the triangular systolic array with boundary cell and internal cell for 2x2 matrixes. The RTL schematic of array has five internal cells and two boundary cells. RTL schematic of systolic array

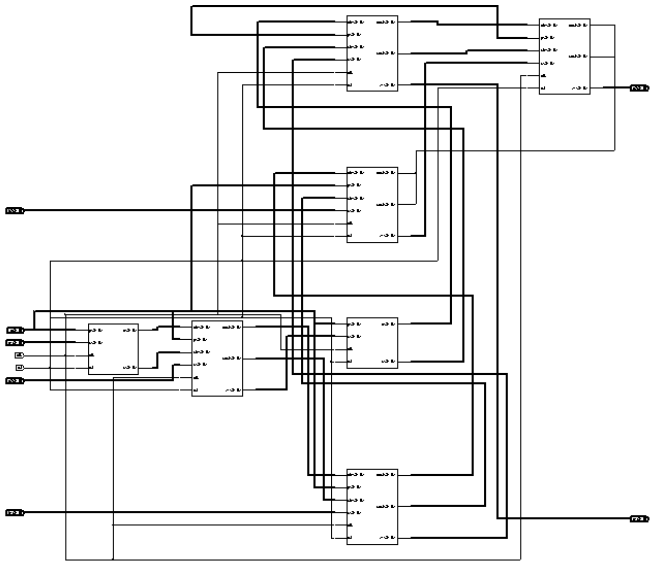


Fig 7.1 RTL of systolic array

B. RTL of Internal Cell:

The RTL schematic of internal cell is shown below with four inputs and clock. One input which filter coefficient P is fixed in internal cells. The two inputs are from the preceding boundary cell or internal cell. And another input is array input which is from the matrix for the two cases this will act according to the condition. The internal cells has two outputs which is regarded has sum, carry and next stage input is fed into the internal cells.

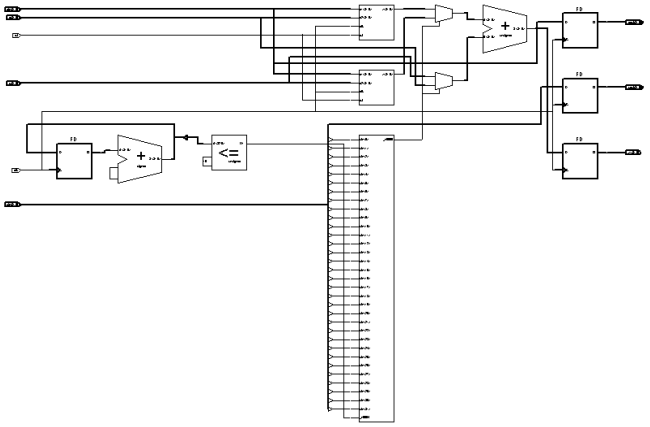
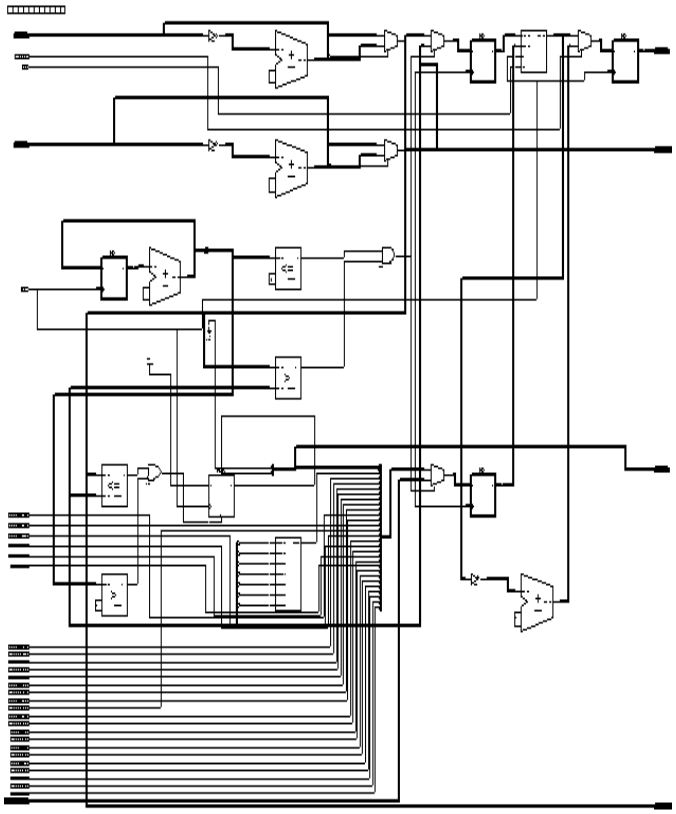


Fig 7.2: RTL schematic of internal cell

B.RTL Schematic of Boundary Cell:

The RTL schematic of boundary cell is shown bellow. It has two inputs. One of the inputs is filter coefficient and the one is feed from the previous stage or matrix input. There two outputs, one of the input is regarded as sum and another one is carry. These outputs are fed into the input of the internal cell.



7.3 RTL schematic of boundary cell

8. Analysis:

A. Timing Analysis Report:

The systolic array structure is working with in time slack of 500ps.hence the operation speed is increasing by using the new hardware structure for matrix inversion

Timing analysis:

```
  Generated by:           Encounter(r) RTL
Compiler v06.10-p003_1
  Generated on:      OCT 17 2017  11:08:23
PM
  Module:                 newarray
  Technology library:     tsmc18 1.0
  Operating    conditions:            fast
(balanced_tree)
  Wireload mode:          enclosed
=========================================
==================


     Pin            Type      Fanout   Load
Slew Delay Arrival
                                    (fF)
(ps)  (ps)   (ps)
-----------------------------------------
--------------------
(clock    clk)                       launch
0 R
(in_del_1)                        ext   delay
+2000    2000 F
a2[0]              in port        30 142.7
0    +0    2000 F
int1_1/x[0]
  a1/b[0]
    g4421/A                             +0
2000
    g4421/Y       NAND2X4          4   33.9
94   +45    2045 R
    g4388/AN                            +0
2045
    g4388/Y       NOR2BX4          2   22.4
82   +85    2130 R

    g329/B0                             +0
8848
    g329/Y        OAI21X2          1    8.1
108   +42    8890 R
  add_24_6/Z[15]
  g852/A                                +0
8890
  g852/Y          NAND2X2          1    3.9
86   +16    8906 F
  g834/B                                +0
8906
  g834/Y          NAND2X1          1    3.3
61   +56    8962 R
  z_reg[15]/D                      DFFRXL
+0    8962
  z_reg[15]/CK   setup                   0
+38    9000 R
- - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - -
(clock    clk)                     capture
9500 R
-----------------------------------------
--------------------
Timing slack :     500ps
Start-point  : a2[0]
End-point    : int1_1/z_reg[15]/D
```

B. Power analysis report:

The total number of cells which is used has the low power requirement compared to other hardware structures.

Power analysis

```
========================================
==================
  Generated by:           Encounter(r) RTL
Compiler v06.10-p003_1
  Generated on:               OCT 17 2017
05:07:46 PM
  Module:                 newarray
  Technology library:     tsmc18 1.0
  Operating    conditions:            fast
(balanced_tree)
  Wireload mode:          enclosed
========================================
==================
                    Leakage        Internal
Net        Switching    Instance     Cells
Power(nW)      Power(nW)          Power(nW)
Power(nW)
----------------------------------------
-------------------------
newa         56749 15994.652 200740657.572
146195430.012 346936087.585
int           8350  2689.544  31124197.183
20596695.649  51720892.832  a          2771
1362.033    15718497.163       9050443.340
24768940.504   add76           98     52.196
572022.309    236104.130    808126.439
adddd74         86     52.087     477953.595
214453.721    692407.316 add72          96
50.333        590474.967        221107.620
811582.587
add60           17     49.457     441485.089
56707.606    498192.695
add56           24     49.335     469194.415
75579.431    544773.846
addinc_add6    17     47.914     271723.274
44110.381     315833.654 addinc_add5    17
47.798        221014.771          38038.661
259053.432
add52           17     47.301     468169.456
59279.388    527448.844
addinc_add5    17     46.441     244519.323
46886.024    291405.347
addinc_add82 107     46.051     573222.147
280196.869    853419.016
add48           17     45.848     472034.305
56897.745    528932.050
add80          109     45.403     565195.762
280639.874    845835.635
add68           86     44.709     478448.790
198314.414    676763.204
addinc_add50   17     44.326     188885.740
47652.462    236538.202
addinc_add86 117     44.226     588185.784
280273.316    868459.100
addinc_add70   86     43.786     476716.225
198353.618    675069.843
addinc_add78 110     43.628     476937.324
261221.152    738158.476
add44           17     42.229     486347.928
54209.331    540557.259
addinc_add4    21     41.886     171761.175
51446.429    223207.604
add40           17     41.350     477275.334
57121.208    534396.542
addinc_add66   31     41.112     291790.461
83506.480    375296.941
add84          121     39.252     549697.013
290288.958    839985.971
add64           40     39.249     431338.643
114279.660    545618.303
add36           28     38.291     453463.634
52989.107    506452.740
add88           98     33.584     441432.094
235834.602    677266.696
```

C. Area analysis report:

The total area analysis report is shown below. The five internal cell and two boundary cell occupies the area of 56749 cells.

```
Area
========================================
==================
  Generated by:           Encounter(r) RTL
Compiler v06.10-p003_1
  Generated on:               OCT 17 2017
05:07:06 PM
  Module:                 newarray
  Technology library:     tsmc18 1.0
  Operating    conditions:            fast
(balanced_tree)
  Wireload mode:          enclosed
========================================
==================
Instance                  Cells   Cell Area
Net Area     Wireload
----------------------------------------
-------------------------------
newarray                  56749      860244
0       <none>
  int1_2                  11085      152492
0       <none>
    a1                     5343       72213
0       <none>
  int1_1                  11094      152289
0       <none>
    a1                     5422       73088
0       <none>
  int1_3                  10992      151877
0       <none>
     a1                    5245       72253
0       <none>
```

```
  int1_4                    8350      132078
0       <none>
     a1                     2771       52231
0       <none>
  int1_5                    8308      130847
0       <none>
     a1                     2657       51509
0       <none>
  bound2                    3465       70486
0       <none>
     a3                     2321       50318
0       <none>
  bound1                    3455       70174
0       <none>
     a3                     2307       49959
0       <none>
```

D.Gate Summary:

```
Gate summary
=========================================
==================
  Generated by:           Encounter(r) RTL
Compiler v06.10-p003_1
  Generated on:               Oct 17 2017
05:07:31 PM
  Module:                 newarray
  Technology library:     tsmc18 1.0
  Operating    conditions:            fast
(balanced_tree)
  Wireload mode:          enclosed
=========================================
==================
   Gate   Instances     Area     Library
-----------------------------------------
-
ADDFHX1          295   22569.624    tsmc18
ADDFHX2          127   14363.395    tsmc18
ADDFHX4            8     931.392    tsmc18
ADDFHXL            7     512.266    tsmc18
ADDFX1           362   25287.293    tsmc18
ADDFX2           484   33809.530    tsmc18
ADDFXL             1      69.854    tsmc18
ADDHXL            35    1280.664    tsmc18
AND2X1             8     106.445    tsmc18
AND2X2           601    7996.666    tsmc18


Type       Instances    Area      Area %
----------------------------------------
sequential       453  26977.104    3.1
inverter       17236 124024.824   14.4
buffer           299   3991.680    0.5
logic          38761 705250.022   82.0
----------------------------------------
total          56749 860243.630  100.0
```

E. Array Summary

The systolic array which is implemented in the hardware structure which does not violated the max transition design rule and max capacitance rule. The summary of the analysis report is shown below:

```
Array summary
=========================================
==================
  Generated by:           Encounter(r) RTL
Compiler v06.10-p003_1
  Generated on:               Apr 18 2007
05:08:49 PM
  Module:                 new array
  Technology library:     tsmc18 1.0
  Operating    conditions:            fast
(balanced_tree)
  Wire load mode:          enclosed
=========================================
==================
   Timing
   ------

 Slack       Endpoint      Cost Group
-----------------------------------------
--
 +500ps int1_5/z_reg[15]/D default

   Area
   ----
Instance  Cells     Cell  Area     Net  Area
Wire load
-----------------------------------------
---------------
New array  56749       860244           0
<none>
   Design Rule Check
   -----------------
Max_transition design rule: no violations.

Max_capacitance    design    rule:    no
violations.
```

Hence the result which is obtained by using the systolic array is compared with the direct implementation of the matrix inversion. Hence the complexity of $O(n^2)$ O(2n).The hardware implementation of the systolic array is done by using the cadence tool. The timing analysis, power analysis, area analysis results are shown which are much reduced then applying the direct inversion.

9. Simulation Results:

A. Systolic Array Output:

In the following fig shows the simulation output of systolic array. The array a1, a2, b1, b2 are stored in a file or given as a input while using for the filter. The filter coefficient P is given as a input to all the cells. And hence the output e1 is obtained in the 6 clock cycle and e2 is

obtained in the 7 clock cycle. And hence the output PA^{-1} is obtained by the systolic array method

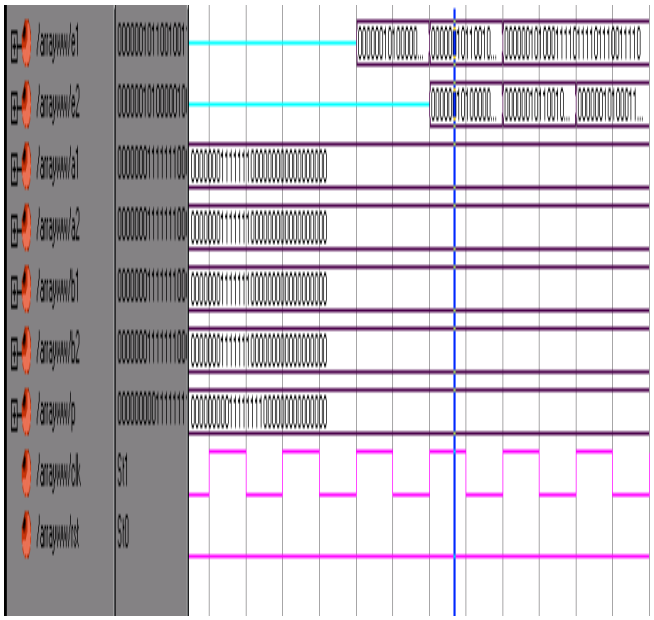


Fig.9.1 output of systolic array

B. Boundary Cell Output:

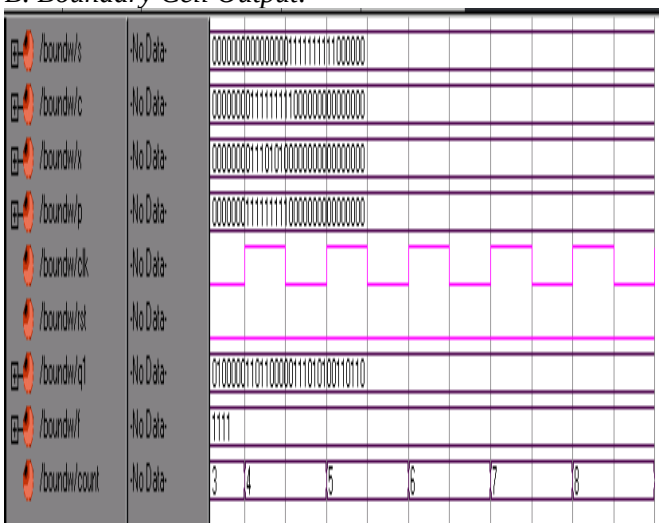


Fig. 9.2 Boundary Cell Output

C. Internal cell output:

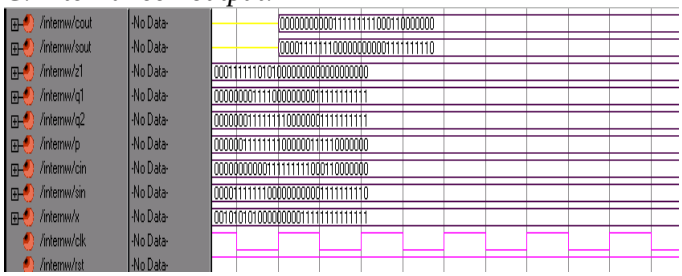


Fig. 9.3 Internal Cell Output

D. Synthesis report:

The systolic array is downloaded to the Xilinx Spartan iii, in which it occupies most of the area. But the flip flop and the clock is used very low and hence the other clocks are wasted. And hence we are going for the ASIC implementation of the structure which uses very low area.

Synthesis report :

Device utilization summary:

Selected Device: 4vfx12sf363-11

Number of Slices:	5984	out of	6144	97%
Number of Slice Flip Flops:	1051	out of	12288	8%
Number of 4 input LUTs:	10896	out of	12288	88%
Number of bonded IOBs:	225	out of	240	93%
Number of GCLKs:	1	out of	32	3%

10. Conclusion:

A customizable structure for an n-state Kalman filter has been designed in verilog and partly implemented in an Altera APEX device with smaller logic resource space. By applying Fadeev's algorithm, matrix manipulations can be performed via the Schur complement. This paper presented a generic Pipeline Systolic Array structure with an advantage of reduced resource consumption, compared to the conventional in fixed-size systolic array structures reported in the literature. The precision error is allowable and within a range. The estimated area of proposed architecture for 4x4 matrixes with an ASIC in 0.35µm CMOS technology is about $125mm^2$. By applying state-of-the-art CMOS technology at the 100nm node, the ASIC area can be reduced to about $20mm^2$.

References:

[1] S. Haykin, *Adaptive Filter Theory*, 4th Edition, Prentice Hall, USA, 2002.

[2] S-G. Chen, J-C. Lee and C-C. Li, "Systolic Implementation of Kalman Filter", *Circuits and Systems, APCCAS '94, IEEE AsiaPacific Conference*, pp 97-102, 1994.

[3] C.J.B. Fayomi, M. Sawan and S. Bennis, "Parallel VLSI Implementation of A New Simplified Architecture of Kalman Filter", *Electrical and Computer Engineering, 1995. Canadian Conference*, Vol 1, pp 117 – 119, 1995.

[4] Z. Salsic and C.R. Lee, "Scalar-based direct algorithm mapping FPLD implementation of a Kalman filter", *Aerospace and Electronic Systems, IEEE Transactions on*, Volume: 36 Issue: 3, pp 879-888, 2000.

[5] D.C. Swanson, *Signal Processing for Intelligent Sensor Systems*, Marcel Dekker Inc., New York, 2000.

[6] S.V. Vaseghi, *Advanced Digital Signal Processing and NoiseReduction*, 2nd edition, John Wiley & Sons Ltd., 2000.

[7] E.W. Kamen, and J.K. Su, *Introduction to Optimal Estimation*,Springer, UK, 1999.

[8] El-Amawy, "A Systolic Architecture for Fast Dense MatricInversion", *IEEE Transactions on Computers*, Vol. 38, NO. 3, March 1989.

[9] G.W., Irwin, “Parallel algorithms for control”, *Control Eng.Practice*, Vol 1, no 4, 1993.
[10] C.R. Lee, “FPLD Implementation and Customisation in Multiple Target Tracking Applications”, *Engineering PhD Thesis*, The University of Auckland, 1998.
[11] D. Lawrie and P., Fleming, “Fine-grain parallel processing implementations of Kalman filter algorithms”, *Control '91., International Conference* , Vol 2, pp 867 – 870, 1991.
[12] Walter T. HigginsArizona State University Tempe, Ariz. 85281”A Comparison of Complementary and Kalman Filtering” IEEE Transactions on Aerospace and Electronic Systems (Volume: AES-11, Issue: 3, May 1975)
321 - 325 May 1975 Print ISSN: 0018-9251