# Design of FPGA ICA for Hyperspectral Imaging Processing

*Anis Nordin*[*a], *Charles Hsu*[**b], *Harold Szu*[**c]
[a]*ECE Department, The George Washington University, Washington DC 20052*
[b]*Trident Systems Inc., Fairfax, Virginia 22030*
[c]*Digital Media RF Lab, The George Washington University, Washington DC 20052*

## ABSTRACT

The problem of remote sensing hyperspectral imaging can be solved by using unsupervised neural network technology named blind source separation (BSS). Using this model, hyperspectral imagery can be de-mixed into sub-pixel spectra that indicate different material presentation in the pixel. This can be further used to deduce areas, which contain forest, water, soil or biomass, without even knowing the sources of the image. This form of remote sensing allows previously blurred images to show the specific terrain involved in that region. The BSS problem can be implemented using an Independent Component Analysis algorithm. The ICA Algorithm has previously been successfully implemented using high-level system packages such as MATLAB to provide a free downloadable version of FastICA, which could easily solve the above problem. The challenge now lies in implementing it in a form of hardware, or firmware in order to improve its computational speed. Hardware implementation also solves insufficient memory problem encountered by software packages like MATLAB when employing ICA for high-resolution images and a large number of channels. In this paper, a pipelined ICA architecture is developed to provide a firmware solution and Field Programmable Gate Arrays (FPGAs) implementation realization using C. Since C code can be translated into High Description Language (HDLs, which is widely used in industry for system design) or be used directly on the FPGAs, it can be used to simulate its actual implementation in hardware. The partial simulation results are presented in this paper, in which seven channels are used to model the 200 different channels involved in hyperspectral imaging.

## 1. INTRODUCTION

Hyperspectral imagery contains a great number of spectral bands or spectra. Different from multispectral imagery that consists of dozens of spectra, hyperspectral imagery often times consists of hundreds of spectra. These spectra indicate the measurements of reflected energy, using the basis that all objects reflect a certain amount of electromagnetic energy. With hyperspectral imagery, instead of multispectral imagery, reflected energy is measured in very precise wavelengths and at very small increments. In this way, subtle spectral features associated with various objects and substances are recorded. Since each object or substance has a unique spectral characteristic, hyperspectral imagery can be used to detect specific substances or terrains, that might go unnoticed in broadband multispectral imagery. A typical imaging system records reflected energy from 405 - 912 nanometers. This covers most of the visible spectrum and samples well into the near infrared, which is beyond the limits of human vision. This can be further utilized to increase the resolution of a picture, allowing us to view previously hidden land topography from long-range images. A hyperspectral remote sensing system can typically generate over 200 channels of imagery simultaneously. These sensors provide spectral images having several hundred channels per pixel. The difficulty in the task now lies in the fact of de-mixing spectrally different materials at sub-pixel level in order to extract information about different materials within the pixel as shown in Figure (1). Each image can be defined as a mixture of different substances within each pixel. Thus the different substances indicate the different sources, denoted by $s$, while each image indicates the different $x$. In the figure, $s$ could be used to indicate rain forest, water, re-growth or deforestation. The main task is to find the sources or the different substances or minerals present within the pixel. In hyperspectral image processing, the use of a neural networks paradigm, more specifically, Independent Component Analysis (ICA) is extremely handy, not to mention computationally powerful. The problem can now be modeled as a source separation problem since we need to de-mix spectrally different materials at sub-pixel level. However, in order to obtain the best solution with incomplete knowledge of the spectral scene, specific pixels are selected for the analysis. Since these pixels may be mixtures themselves and the original spectral scene is unknown, it fits the blind source separation problem. This ICA paradigm was introduced by Szu [5,6,7] where the pixels under consideration are assumed to be linearly mixed spectra of different materials and each spectral scene is considered to be independent of each other.


Wavelet Applications VIII, Harold H. Szu, David L. Donoho, Adolf W. Lohmann, William J. Campbell, James R. Buss, Editors, Proceedings of SPIE Vol. 4391 (2001) © 2001 SPIE. · 0277-786X/01/$15.00

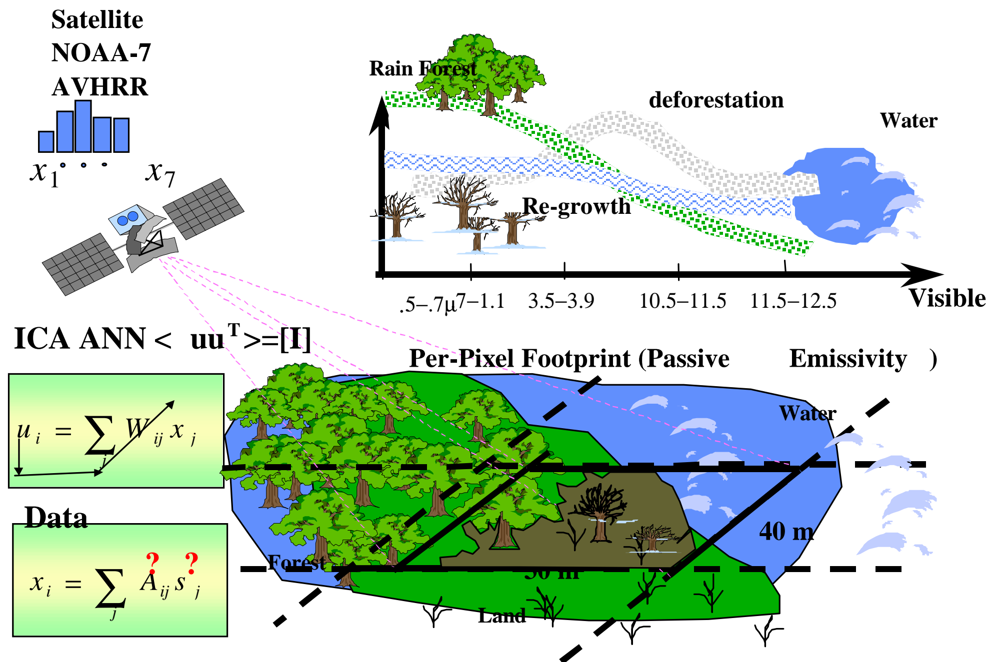


Figure 1 : Landsat remote sensing application. Reflectance matrix [A] changes daily, seasonally, and thus unkown in principle. Percentage $s_j$ varies from pixel to pixel.

The massively parallel nature of neural networks makes it highly suitable for hardware implementation of independent blocks. This implementation also makes it computationally powerful both due to the use of parallel blocks as well as dedicated circuitry, which allows its computational speed to rival any software implementation. The hardware implementation of neural networks also make it highly fault tolerant to system damage, since its distributed nature requires the damage to be extremely extensive before the overall performance of the network is degraded seriously. A digital neuroprocessor is a dedicated digital VLSI system of which its characteristics are optimized for computing the artificial neural network algorithms. It has important advantages such as good programmability, high flexibility, high precision and full pipelinability. Having said all that, a true hardware implementation of neural networks is extremely difficult and expensive since it requires application specific design of integrated circuits. Under this method, each neuron would have to be translated into its equivalent circuitry and this process can be quite tedious to implement networks with over 200 channels. A software implementation, however, is more popular and easily implemented, since specific neural networks algorithms can be easily translated into a high-level programming code. This fact is illustrated by the profusion of neural network programs, which are normally implemented using MATLAB. MATLAB, while user-friendly and extremely powerful in matrix computation, has a memory problem when handling high-resolution images. This problem is highlighted further in section 3.

A tradeoff is to implement the neural networks algorithm using firmware, which is a mixture of both hardware and software design. Field Programmable Gate Arrays (FPGA) is a general-purpose, multi-level programmable logic device that is customized by end users. FPGAs are composed of blocks of logic connected with programmable interconnect. The programmable interconnect between blocks allows users to implement multi-level logic, removing many size limitations of the Programmable Logic Devices (PLD). Its architecture supports thousands of gates of logic at a system speeds in tens of megahertz. The FPGA implementation of neural networks make it highly cost-effective since it has the low-cost of a software while at the same time incorporating the high-speed of dedicated hardware.

This paper is organized as follows. First of all, we will identify the problem and significance of hyperspectral imaging processing versus BSS and ICA. Secondly, we propose the potential solution using ICA. The detailed description of ICA algorithm is presented in the next section. For the hardware realization, we develop a pipelined architecture ICA to provide a high throughput and high performance of hyperspectral imaging processing in section 3. Section 4 shows the simulation results. And the conclusion, discussion, and future work are included in section 5.

## 2. INDEPENDENT COMPONENT ANALYSIS

Independent Component Analysis or ICA uses the idea that in a mixture of signals, the sources are statistically independent. This can be further illustrated using the following equation :

$$x_1(t) = a_{11}s_1 + a_{12}s_2 \quad (1)$$

$$x_2(t) = a_{21}s_1 + a_{22}s_2 \quad (2)$$

In such case, $x_1(t)$ and $x_2(t)$ are 2 separate channels. Suppose that in our case, we have two separate images, $x_1(t)$ and $x_2(t)$. $s_1(t)$ and $s_2(t)$ are the spectrum generated by different material substances resident in the pixel. $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$ are parameters that reflect the linear combination or the amount proportionality of the substances are combined together.

Equations (1) and (2) can be expressed more elegantly in a matrix form :

$$\mathbf{x} = \mathbf{A}\ \mathbf{s} \quad (3)$$

where **A** can also be written in the form of

$$x = \sum_{i=1}^{n} a_i s_i \quad (4)$$

In both cases, **x** is a linear combination of **s** components. There are several key assumptions made using ICA, namely :

- The components $s_i$ are statistically independent.
- The independent components have nongaussian distributions.
- The unknown mixing matrix is assumed to be a square matrix.

Upon estimating the matrix **A**, it's inverse, **W** can be computed and the independent component can be found using:

$$\mathbf{s} = \mathbf{W}\mathbf{x} \quad (5)$$

Using this model, there are two ambiguities that is apparent. Firstly, the variances (energies) of the independent components cannot be determined. Thus, it is assumed that the magnitudes of the independent components each has a unit variance $E\{s_i^2\} = 1$. Secondly, the order of the independent components are also unknown, as such it can be considered as unimportant and the ICA Algorithm will process the independent components randomly.

The basic principle of ICA estimation can be described as follows:

Putting Equation (5) in another form :

$$\begin{aligned} y &= \mathbf{w}^T\mathbf{x} \\ &= \mathbf{w}^T\mathbf{A}\mathbf{s} \\ &= \mathbf{z}^T\mathbf{s} \end{aligned} \quad (6)$$

$y$ then is considered to be a linear combination of $s_i$ with the weights given by $z^T$. It becomes least gaussian when it equals to one of $s_i$ or one of the independent components. Thus, the basic idea is to measure nongaussianity and **w** is taken as a vector that maximizes the nongaussianity of $\mathbf{w}^T\mathbf{x}$.

Nongaussianity can be measured by the fourth or order cumulant or kurtosis :

$$K(y) = E\{y^4\} - 3(E\{y^2\})^2 \quad (7)$$

Since $E(y^2)$ is approximated as 1 the above is reduced to $K(y) = E(y^4)$. The fourth order cumulant can be reduced to a second order cumulant if $y$ is gaussian. This in turn shows that the kurtosis is zero for gaussian random variables and thus, nongaussian random variables have a nonzero kurtosis.

The weight vectors are initialized to **I** and y is computed iteratively with the weight vectors are updated using the equation :

$$d\mathbf{w}/dt = dK/d\mathbf{w} \quad (8)$$

Since each image has its own unique value of Kurtosis, a stationary Kurtosis yields an the de-mixed images or in our case, de-mixed pixels, without even knowing supposed output.

Since the order of the independent components are unknown, the matrix **A** could also be represented using an arbitrary permutation matrix **P** and a diagonalized matrix **D**.

Multiplying Equation (3) with **W,**

$$\mathbf{W\,x} = \mathbf{W\,A\,s} = \mathbf{P\,D\,s} \quad (9)$$

### 2.1 ICA Preprocessing

Before applying an ICA algorithm on any data, it is useful and more efficient to do some preprocessing steps first. The basic pre-processing steps are :

- Centering : x is usually centered, which indicates that the mean vector m = E{x} has to be subtracted from it, to make x a zero-mean variable.
- Whitening : Transforming the observed vector x such that its components are uncorrelated and its variances equal unity.

## 3. PIPELINED STRUCTURE FPGA IMPLEMENTATION

### 3.1 Programming for FPGA

FPGAs are reconfigurable integrated circuits (ICs), which can be programmed to fit a specific function. The multi-purpose FPGAs have been successfully utilized for prototyping ASICs since they have both a very short development time and the cost of modifying the design is minimal. Programmable gate arrays have been used as reconfigurable processors in customized operating systems. They can be used to replace small quantities of simple ASICs. The FPGA code shown was modeled using the SRAM FPGA. SRAM programmable FPGAs allows programs to be reprogrammed and reloaded into the external memory for permanent storage of the program. It allows incremental build up of the programs where a small program is tested on board and more functions are added once each small part is working perfectly. This type of programming can also be termed as customized hardware where the software is configured into an FPGA board and is referenced as a hardware subroutine. The parallel nature of FPGAs allow a number of subroutines to be placed on one array of FPGAs and accessed in parallel via a very long instruction word. Xilinx SRAM based FPGAs have elements of configurable logic blocks (CLB), input/output blocks (IOBs) and interconnect.

When writing software, the designer implements many subroutines in order to perform certain functions. Similarly, in hardware programming, a hardware subroutine is a standard procedure call which requires several designated inputs and returns the necessary values upon completion. The major difference lies in the fact that in hardware subroutines the job will be executed by FPGAs instead of a series of micro-instructions. The figure bellow illustrates the concept of a hardware routine among software.

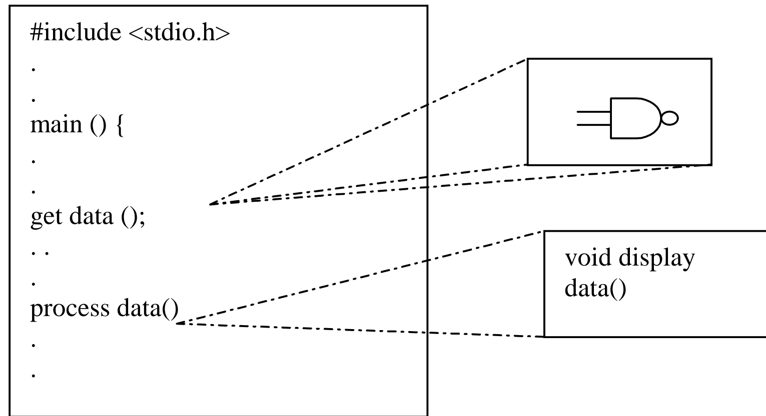


Figure 2 Process data is implemented in hardware [13]

Most of the high level functionality is designed in C code. The hardware subroutine is then later designed to replace computationally intensive software subroutines [13]. The hardware subroutines can either be designed using verilog or VHDL or even C since Xilinx now allows the FPGAs to be coded in C. Although FPGAs can be coded directly using C, the semantics of a high-level language programming and a low-level hardware language is naturally very different. Static variables and their associated state map in an obvious way to registers. Arrays, pointers, and their associated memory model can sometimes be localized and mapped to a RAM structure in hardware (especially for small fixed-size arrays), but often pointer arithmetic requires a full-fledged memory interface to implement properly. The behavior of these constructs is implementation dependent, but discouraged. The general rule is that non-obvious mappings are disallowed: if the programmer cannot easily visualize how a construct will map into hardware, then it is very difficult for the compiler and programmer to agree on the correctness of the result.

The design of the FPGA ICA is based on the Basic Source Separation Code in MATLAB provided by Tony Bell from Salk Institute. The original MATLAB code was fine-tuned by Charles Hsu in order to process images instead of audio files. The above modules were coded using C to improve its efficiency and for easier translation into High Descriptive Language or HDL which is essential for FPGA implementation. As mentioned earlier, MATLAB has a memory problem when dealing with high resolution images. This is due to the fact that each variable, in this case, a pixel in MATLAB requires 8 bytes of storage space. A typical bitmap image, as indicated in figure 7 and figure 8 requires 200 x 200 pixels. This is approximately 320Kb per image. This makes MATLAB unsuitable for hyperspectral image processing since the images are not only high resolution, there is also a large number (greater than 200) of channels in our problem. A program that has 200 channels or images, would require at least 64MB just to load the images. Considering that the ICA program involves numerous matrix computations such as covariance, multiplication and inverse functions, MATLAB would run out of memory very quickly. Conversely, C only requires 1 byte per pixel in bitmap images of 256 colors. C also has functions which can allocate and deallocate memory space, namely *malloc* and *free.* This allows C programs to initialize memory space for loading images and freeing the memory space later to store the newly generated images instead. C also runs computations at a higher speed than MATLAB, which takes more 60 seconds to perform the ICA algorithm for only two images. This makes C programs not only computationally more efficient, but economical in memory storage as well.

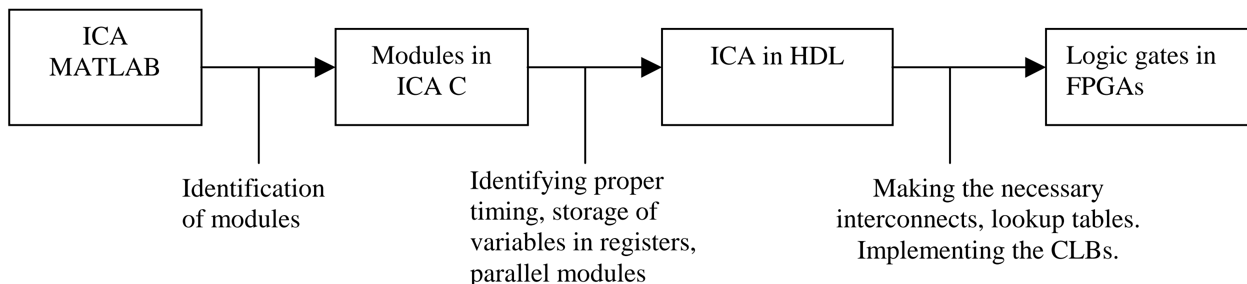


Figure 3 Steps in coding FPGAs from MATLAB

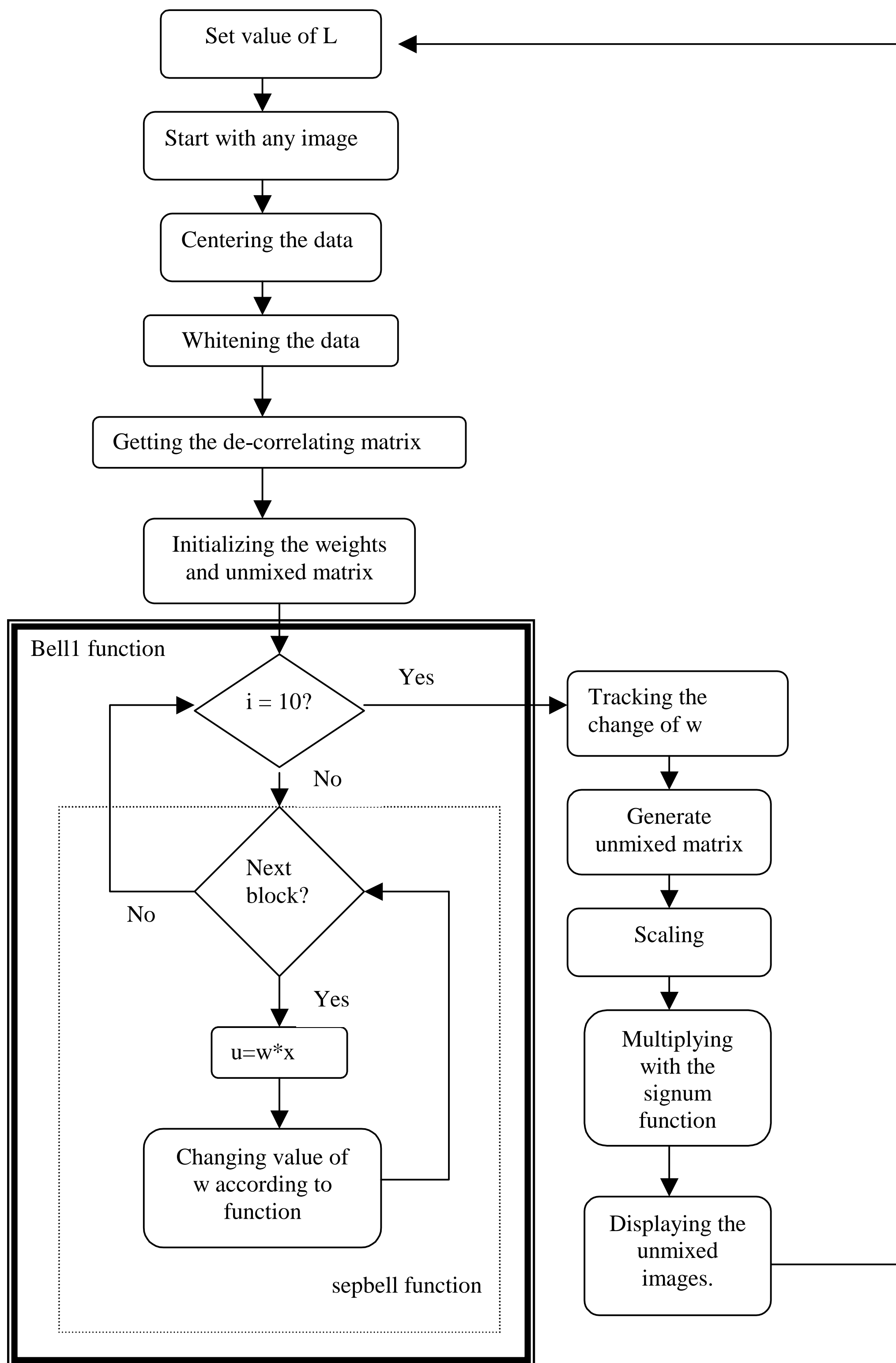


Figure 4: MATLAB and C program flowchart

In order to translate the code from MATLAB to C, the code was first broken down into smaller modules, where each module was hand coded into C. Although MATLAB has a compiler, which provides code translation from MATLAB into C, it generates C codes, which reference to MATLAB libraries. This however, makes it unsuitable for FPGA implementation. The flowchart in Figure 4 illustrates how each MATLAB function was broken down into C modules. It was also necessary to generate several more C subroutines to implement in-built MATLAB matrix functions such as inverse, covariance and matrix multiplications. As shown in both the flowchart and the samples of coding, the C program does a lot of referencing to functions. The use of subroutines allows the program to be implemented in blocks in the FPGAs.

Below are samples of translation of ICA MATLAB code into C.

```
function ICA
for L=5e-5:5e-6:2e-4
L
[Dmix,w,wz]=bell1(M',L);
% generate demixed output image files
..........................
end
%% FUNCTIONS
function [uu,w,wz]=bell1(mixes,L)
mx=mean(mixes');
x=x-mx'*ones(1,P);                    % subtract means from mixes
.........
x=wz*x;
M=size(w,2);
Id=eye(M);
......
% ica processing
...... . .
for i=1:10, [w,oldw,olddelta]=sep_bell(sweep,P,B,Id,x,w,L,oldw,olddelta,N,M);end
uu=w*wz*mixes;             % make unmixed sources
```

C program

```
# include <stdio.h>
main () {
for (L=5e-5;L< 5e-6; L=2e-4)
            bell1(Dmix,w,wz)  //Calling the bell1 function
..          imgdisp(pix,x,y)  //Calling the function that will display the unmixed image
end
double bell1(double mixes[row][cols], double L) //Declaring the bell1 function
{
mixestr=transpose(mixes)//Calling the transpose function
mx=mean(mixes)//Calling the mean or average function
.......
//get decorrelating matrix
x=matrixmult(wz*x) // Calling the matrix multiplication function
M=sizeof(w);
Id=idmatrix(r,c)//Calling the identity matrix function
......
for (i=1;i<10;i++)
      sepbell(w,oldw,olddelta);
uu=matrixmult(w,wz,mixes);
wwz=matrixmult(w,wz);
newsum=sum(wwz);
nnewsum=sum(newsum);
sizewz=sizeof(wz);
```

### 3.2 Pipelined Architecture

Similar to product assembly lines, pipelined architectures process more than one instruction at a specific time. Each job or program is divided into small, partitioned blocks of instruction, which are processed a bit at a time at each station. Just like a product is built a little at a time at each station, pipelines allow the job to run each partitioned block at each station. The use of pipelines allows the designer to build faster, more dedicated logic at each station. This increases the program execution speed. The speed is further improved using parallel processors, which are ideal for implementation of neural network models. This parallelism allows instructions to be processed concurrently. In order to maximize the concurrent processing, the blocks must be partitioned in such a way that all modules can run or are ready to be processed at a specific time period, and at the end of each period, all modules are completed. This method requires all inputs of a module to be ready at the start of time period t, and all outputs exit the module at the end of time period t.

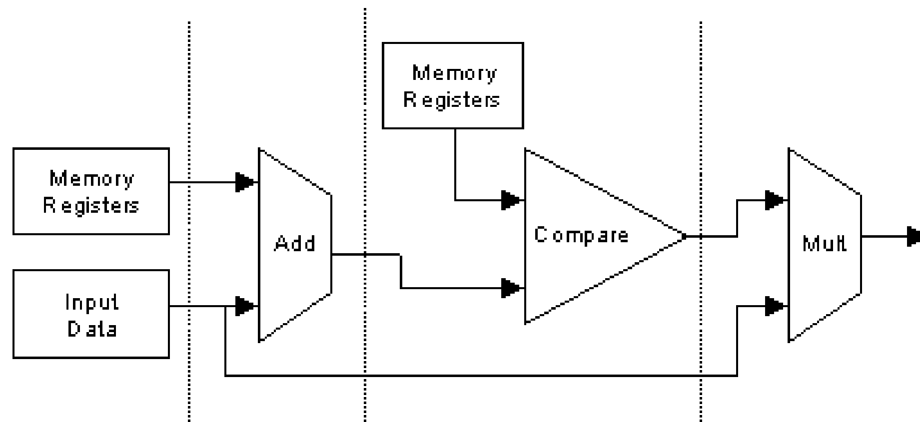


Figure 5. An example of a 4-stage pipelined processor

The efficiency of a pipelined structure can be illustrated using Figure 2 above. This processor has four stages in one job. The first stage involves user's input and memory addressing. The second stage performs the processing or the addition of these numbers, while the third stage compares this value with one stored in the registers. The final stage performs multiplication of the input data with the compared data. If the task is executed using the normal method, it requires four clock cycles to complete one job, eight clock cycles to complete two jobs and so forth. However, using the pipelined architecture, four clock cycles are need to complete 1 job and only five clock cycles are needed to complete 2 jobs. The number of clock cycles required for a pipelined architecture increments by an addition of 1, while a non-pipelined architecture increments by a multiplication of two.

### 3.3 Pipelined FPGA Implementation

Similarly, the FPGA ICA program was implemented using the pipelined architecture, where each block in the flowchart, which did not have data dependency with each other, was implemented in parallel. The program is first written in C completely at first, before the next stage is implemented. The next stage involves identifying the critical sections, timing considerations, parallel subroutines, and these are converted into logic functions that are realized in hardware. This can be done either in verilog or VHDL or even C. Tools such as Synopsys allows these logic gates to be simulated and verified before downloading it to the FPGAs through a cable. The figure below shows how the multiplication of a constant which is changed constantly according to a function is translated into logic design. The FPGA ICA algorithm executes this where the weights are incremented using the inverse of an exponential function. Here the values of the function are stored in a lookup table which is stored in the external memory of the FPGAs.

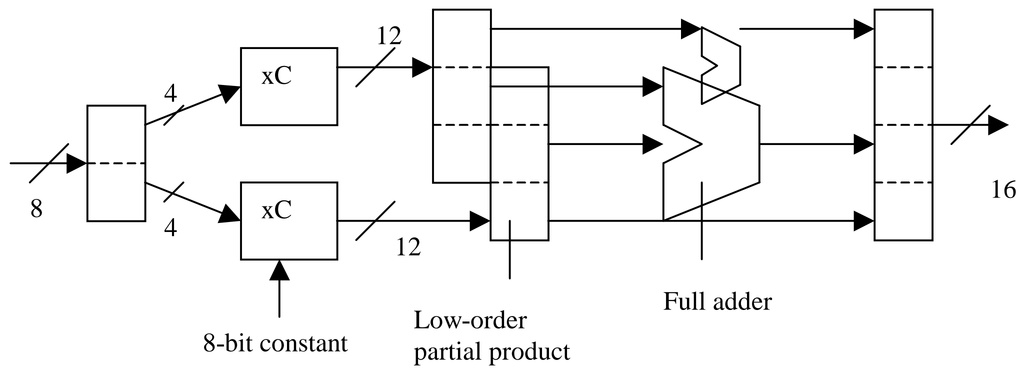


Figure 6. Multiplication by a constant. Figure taken from [12]

Figure 7 Landsat 7 channels of image data

## 4. SIMULATED RESULTS

The above modules were implemented using C and simulated using hyperspectral imagery. For testing purposes, only seven input images were used, compared to the actual 200 images generated by a hyperspectral image sensing system. The figures below show the input and output images of the ICA Algorithm in C. The images are Landsat images of Tel Aviv, where the black area is the sea and the lighter parts are the shore. The resolution of the images are greatly improved when processed through the ICA program, where the shorelines of Tel Aviv and the structure of the terrain on the shore can be visualized clearly.

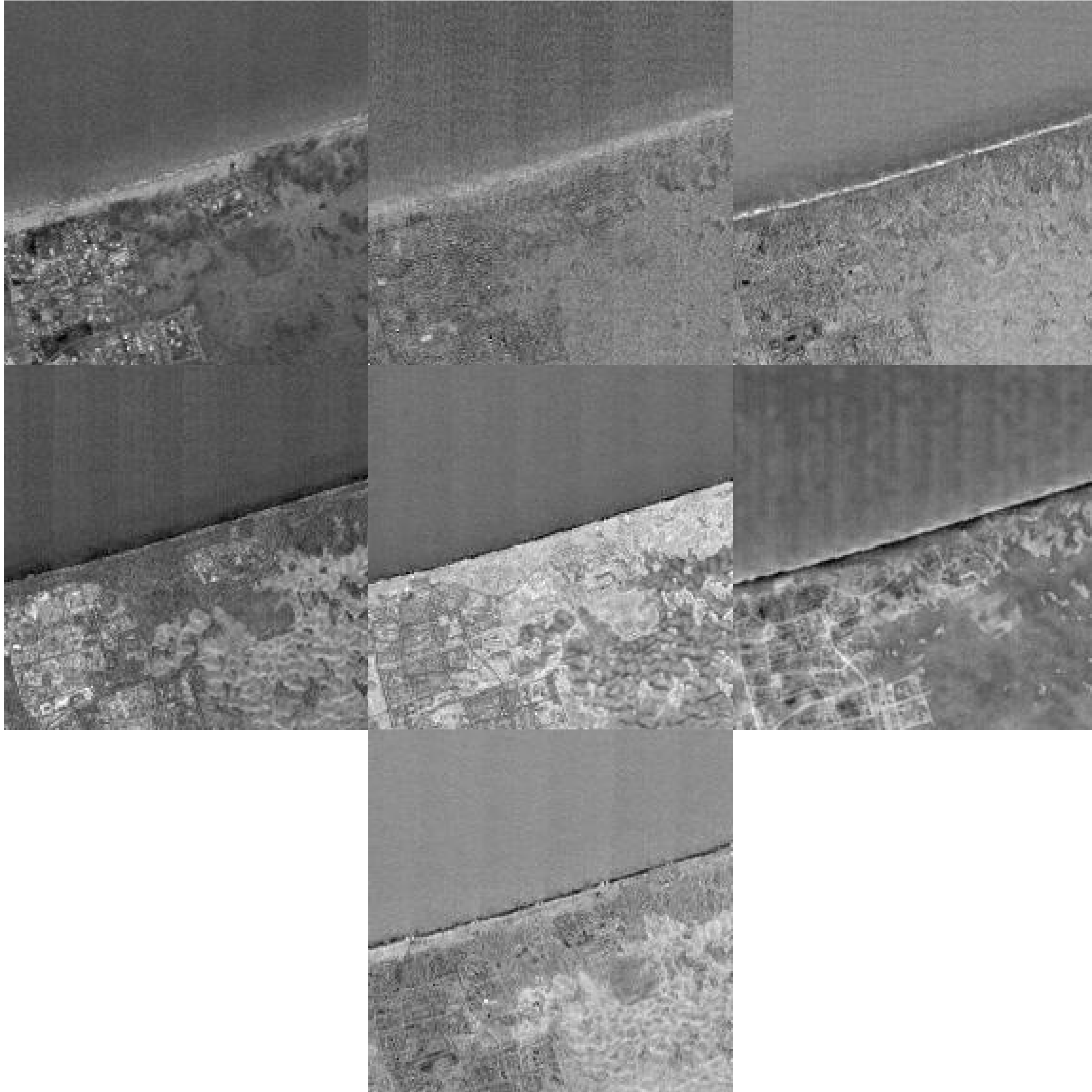

Figure 8. De-mixed image data

## 5. CONCLUSION

The implementation of the ICA algorithm using pipelined blocks of instructions, greatly improves the computational efficiency of the program. Since the algorithm has been partitioned into manageable and sizable blocks of coding, the C code FPGA implementation can be easily done. The program is then tested using various hyperspectral images, namely the 7

images from the Landsat which constitute as the mixed data. The de-mixed data is generated at the output, providing a high-resolution image of the terrain previously unable to be viewed using the raw data.

Currently, the FPGA ICA design has only completed its first stage, where the MATLAB code has been translated into modules in C and simulated. This design can either be totally hand coded into verilog or VHDL or realized incrementally using a mixture of C and hardware language as shown in figure 2. Hardware implementation can provide improvements of speed up to 146% especially in computations involving vector multiplications [13]. The FPGA implementation of the program can be expected to improve the speed of computation of about 100%, considering that most of the computation involves matrix operations like matrix multiplications and inverses. This can be implemented in parallel to speed up the execution time. Further work involve translating the C code into HDL or modifying the C code to fit the hardware. Comparisons between computational speed of HDLs and the C code could also be made to substantiate the advantages of implementing the software into firmware. While there are software which performs translations between HDLs and C, the technology is not yet perfect, and thus, it is perhaps better to hand code the translation or perform extensive verifications for automated translations of HDLs to C. Hand coding ensures that the functionality of each module is correctly translated into logic gates and provides better efficiency since the parallel subroutines can be identified and employed using the parallel gate arrays.

## REFERENCES


[1] H. Szu, G. Meurer, B. Cantrell, R. Stapleton, "Digital array technology for Radar", Radar 2000, Wash DC, 2000.
[2] H. Szu, IEEE Circuit & System Newsletter Dec. 99
[3] H. Szu, IEEE Industrial Electronic Newsletter June 99
[4] Telfer B. A., Szu H. H., Chiang R., "Classifying Multispectral data by Neural Networks", Telemetics & Informatics, vol. 10, no. 3, pp. 209-222, 1993
[5] H. Szu and C. Hsu, "Landsat Spectral Demixing à la Superresolution of Blind Matrix Inversion by Constraint MaxEnt Neural Nets," in Wavelet Applications IV, *Proc. SPIE*, **3078**, 1997, 147-160.
[6] H. Szu, C. Hsu, "Blind De -mixing with Unknown Sources", International Conference on Neural Networks, vol. 4, pp 2518-2523, Houston, June, 1997.
P. Comon, "Independent Component Analysis: A new concept." Signal Processing 36:287-314, 1994
[7] Bell A. J., Sejnowski T. J., "Learning the high -order structure of a natural sound", vol. 7, *Network: Computation in Neural Systems*, Feb. 1996.
[8] Amari I., Cichocki A., Yang H., "A new learning algorithm for blind signal separation", NIPS 1995, vol. 8, 1996, MIT press, Cambridge, Mass.
[9] Oja E., Karhunen J., "Signal s eparation by nonlinear Hebbian learning", Proceedings of ICNN -95, pp. 417-421, Perth, Australia, 1995.
[10] S. Arnold, C. Hsu, M. Zaghloul, H. Szu, N. Karangelen, J Buss, "A Fully Digital FOPEN SAR Processor", accepted SPIE (Wavelet Applications VIII), Orlando, April 2001.
[11]B.J. Sheu, J. Choi, "Neural Information Processing and VLSI", Kluwer Academic Publishers, Norwell, Mass. 1995.
[12]S.M.Trimberger, "Field-Programmable Gate Array Technology", Kluwer Academic Publishers, Norwell, Mass. 1994.
[13]T.J.Bauer, "The Design of an Efficient Hardware Subroutine Protocol for FPGAs", Masters Thesis, Massachusetts Institute of Technology, 1994.



* *anisnn@seas.gwu.edu; phone 1-703-288-1175; ECE Department, The George Washington University, 725 23rd St. NW, Washington DC 20052.*
***hsu@tridsys.com; phone 1-703-267-2313; fax 1-703-2736608; Trident Systems Inc., Fairfax Virginia 22030*
*** *szuh@seas.gwu.edu; phone 1-202-994-0880; fax 1-202-994-0223; Digital Media RF Lab., ECE Department, The George Washington University, 725 23rd St., NW, Washington DC 20052*