Hardware Decompresssion for Compressed Sensing Applications

Keith Dronson Frank Zovko Samuel Subbarao
Federico Garcia

May 16, 2009

Contents

1 Introduction **2**

2 Mathematical Background **2**

3 Setup and Compression **4**

4 Software Architecture **4**

5 Hardware Design **5**
5.1 Matrix-Vector Multiplier 5
5.2 LFSR-Based Random Number Generator 6
5.3 VGA Controller 6

6 Daubechie Wavelet Transform **6**

7 Results **9**

8 Conclusion **10**
8.1 Who Did What 10
8.2 Lessons Learned 11

A MatLab Code **11**
A.1 Compression and Decompression 11
A.1.1 SolveOMP 11
A.1.2 Compression Decompression 15
A.2 Daubechie Wavelet Transform and Inverse Transform 17
A.2.1 1d Daubechie Wavelet Transform 17
A.2.2 2d Daubechie Wavelet Transform 17
A.2.3 Full Image Daubechie Wavelet Transform 17
A.2.4 1d Inverse Daubechie Wavelet Transform 17
A.2.5 CPV code 18

A.2.6 2d Inverse Daubechie Wavelet Transform 18
A.2.7 Full Image Inverse Daubechie Wavelet Transform 18

B C Code 19
B.1 Daubechie Wavelet Inverse Transform 19
B.1.1 Driver 19
B.1.2 Header 21
B.1.3 Decompression functions 21
B.1.4 CPV Code 28
B.1.5 Transpose Code 28
B.1.6 1d Inverse Daubechie Wavelet Transform 29
B.1.7 2d Inverse Daubechie Wavelet Transform 30
B.1.8 Full Image Inverse Daubechie Wavelet Transform 30

C VHDL Code 31
C.1 Matrix Vector Multiplier 31
C.2 Column Module 41
C.3 Accumulator 43
C.4 LFSR 44
C.5 Single-Port RAM 46
C.6 VGA Raster 47

1 Introduction

Most conventional approaches to sampling a signal are based on Shannon's sampling theorem: the sampling rate should be twice the maximum frequency in the signal (aka Nyquist rate). When it comes to pictures, which are not band-limited the sampling rate is determined by the desired resolution of the picture. Compressive sensing (CS) provides a way to recover an image from far fewer samples than would normally be necessary. CS relies on two basic principles: Sparsity and Incoherence[1]. Sparsity is the idea that the bandwidth of a signal may be larger the actual number of "information" samples. This leads to the fact that if these samples were represented in the right basis Ψ they would be less sparse (more compressed). Incoherence extends the duality between time and frequency: something that is compressed in Ψ will be spread out in the domain that it was acquired in.

2 Mathematical Background

The typical approach to sensing is the following:

$$y_k = \langle f, \phi_k \rangle \tag{1}$$

where f is the image to be sampled, ϕ_k is the sensing waveform, and y_k is the sampled data. If the ϕ_k's are indicator functions of pixels, then the y_k's

are the typical image data collected from a camera. The complexity arises from the number of dimensions of y, which we'll call n. One could try to take n measurements (more pixels in a CCD) or one could be clever and find a solution that allows them to undersample; say collect m samples instead of n. In that case one could create an $m \times n$ sensing matrix, A, composed of n rows of the ϕ_k's: $\phi_1^*, \phi_2^*, \ldots, \phi_m^*$ (where a * denotes the complex transpose). Since f is n dimensional, but y is of dimension m and $y = Af$, there are an infinite number of possibilities for f. However in some cases there is a way out of this.

SPARSITY: If $f \in \mathbf{R}^n$ and sampled in an n dimensional basis $(\phi_1, \phi_2, \ldots, \phi_n)$, then we have the following relationship:

$$f = \sum_{1}^{n} x_i \phi_i \tag{2}$$

However, if some of those x_i's are small there may be a subset of the ϕ_i's that almost add up to f. In that case:

$$f = \sum_{1}^{s} x_i \phi_i \tag{3}$$

or

$$f = \Phi x_s \tag{4}$$

where Φ is an $n \times n$ matrix of $\phi_1 - \phi_n$ as columns, and x_s are the s largest coefficients of the x_i's. The figure 1 below shows how this works, and can be quite good at reconstructing the image.

INCOHERENCE: Since $f \in \mathbf{R}^n$, we can find two basis sets Φ and Ψ for the space. Φ will represent f as shown before, and Ψ will be used as the sensing basis. The coherence measures the largest correlation between any two elements of Φ and Ψ. Compressive sensing looks for low coherence pairs (maximum incoherence). Since Φ will be some fixed basis, it has been shown that the best basis for Ψ is a random basis (white Gaussian noise).

As mentioned before, y can be sensed in the Ψ basis: $y = \Psi f$ or $y_k = \langle f, \psi_k \rangle$ (dot product of f with each basis vector in Ψ). In order to recover the image we look at the following:

$$y_k = \langle \phi_k, \Psi f \rangle \tag{5}$$

where f is the signal to be recovered

Solving this equation for f is impossible. However, we believe (or know) that f is sparse. In that case we can look for the following signal that will solve the minimization problem:

$$\min \left(\|x\|_0, \Psi x = y \right) \tag{6}$$

Essentially here we are looking for an x with the least number of non-zero coefficients that will satisfy $\Psi x = y$. This too is intractable. So we will use:

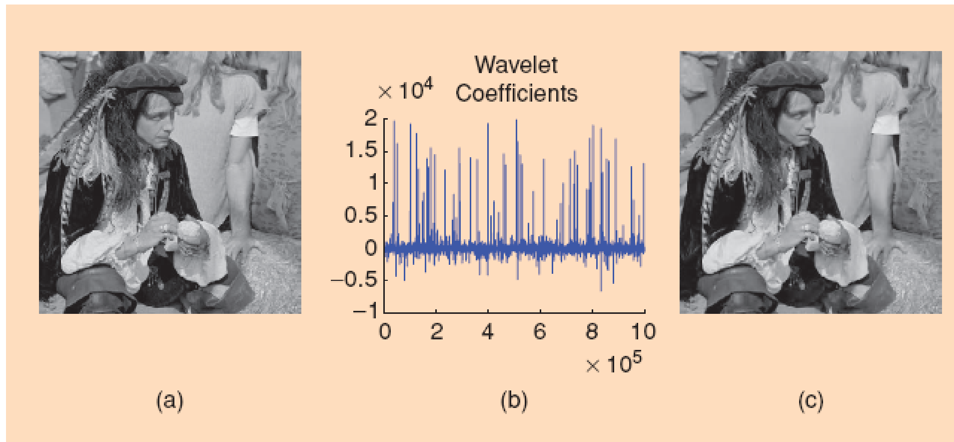

Figure 1: Part a of Figure 1 shows the initial image. Part b is the image in the ϕ basis. Note that there are only a few discrete ϕ_i's that have x_i's with large coefficients. Part c is the reconstruction of the image using the phi_i's linked to the largest 25,000 coefficients. This means that 97.5% of the sampled data was thrown away and the picture still looks pretty good.

$$\min\left(\|x\|_1, \Psi x = y\right) \tag{7}$$

This can be done in a reasonable amount of time. Finally $f_{\text{rec}} = \Phi x$.

L1 minimization algorithms are not the only way to recover compressively sensed data. There are greedy algorithms available that allow one to do this as well.

3 Setup and Compression

The compression of the image is done on a PC using Matlab. The image, which in our case is just black and white (although it can be extended to color) is first made sparse using the Daubechie Wavelet Transform (described later). The N largest elements are preserved while the rest are set to zero. The sparse image is then multiplied by the random matrix A, resulting in a smaller data set. This smaller data set is sent to the FPGA to be decompressed.

4 Software Architecture

The primary function of the design is to take compressively-sensed data and decompress it to yield an accurate representation of the original image. We chose the Orthogonal Matching Pursuit algorithm[2] to perform the decompression and obtained a Matlab implementation of the algorithm from Stanford

University's SparseLab[3]. The algorithm uses recursion to perform L1-norm minimization in approximating a solution to an underspecified linear equation.

We implemented this algorithm in C by directly mapping all of its functions from Matlab to C. The compressed image was hard-coded as a C array generated from our Matlab implementation of the compression algorithm. After converting all floating point calculations to fixed point, we first verified that our C implementation (running on a desktop PC) yielded the same results as our Matlab implementation. After this was confirmed, we designed logic to increase the speed of the three most computationally intensive parts of our algorithm, all of which were matrix-vector multiplications.

All three of these matrix-vector multiplications involve our random matrix A, which is composed of 4096 rows and 16384 columns and contains only the values 1 and -1 as its elements. Due to the size of this matrix and our limited storage space on the DE2 board, we opted to generate parts of this matrix as they were needed, using parallel 32-bit linear feedback shift registers with seeds stored in a ROM on the FPGA. When the C algorithm reaches a point in the program at which a matrix multiplication must be performed, it loads the necessary data into the FPGA's block RAM, transfers control to the multiplier unit until a result is available, and then retrieves the result. SDRAM is used solely for the Nios processor's data and instruction memory.

5 Hardware Design

5.1 Matrix-Vector Multiplier

The matrix-vector multiplier unit is responsible for taking an input vector and left-multiplying it by either a part of the random A matrix, or the entire matrix, depending on the particular point in the decompression algorithm at which it is used. As the matrix involved in a multiplication is always composed entirely of the numbers 1 and -1 (the latter being represented as a 0 in hardware), multiplying a row of the random matrix by a vector amounts to performing one-by-one addition or subtraction of the elements of the vector, to create an accumulation function.

The Nios processor loads an input vector into the FPGA's block RAM and specifies what rows of the transposed random A matrix should be multiplied by that vector. Once the processor signals that the multiplier may begin its operation, parallel 'column modules' generate the A matrix columns needed to multiply part of the matrix by part of the input vector. The matrix is transposed before being multiplied, so generating its columns in parallel will generate the rows of the multiplied matrix in parallel, thereby allowing parallel computation of the elements of the output vector. Limited by the available number of gates on the FPGA, we chose 128 as the number of these parallel elements to generate in hardware, providing the ability to compute 128 elements of the output vector in parallel at one time. Groups of 128 values from the output vector are computed until the entire output vector is filled; for an output vector containing 16384

elements, this would take 128 iterations.

A state machine controls the interaction between the block RAM, accumulators, LFSRs and Nios processor. A diagram of the system is shown in Figure 2 and a state diagram is shown in Figure 4.

5.2 LFSR-Based Random Number Generator

The columns of the random matrix are generated by parallel linear feedback shift registers. To generate a given column (which contains 4096 randomly distributed 1 and 0 elements), a 32-bit seed, stored in a ROM and specific to that column, is retrieved and used to seed a LFSR that is used to generate all the bits of the column. These bits, as they are generated, are used to accumulate the corresponding elements of the input vector in generating one of the numbers in the output vector. For example, if element 12 of the output vector were being computed by multiplying A's transpose by the input vector, each of the 4096 bits in column 12 of A would be generated and used to add or subtract each element of the input vector. Using 32-bit LFSRs to generate the entire A matrix yields a non-repeating sequence of bits, which is important for obtaining sufficient randomness.

The accumulator is schematically shown in Figure 3.

5.3 VGA Controller

Images to be displayed on the VGA screen are stored in the DE2's SRAM chip, which is used as a framebuffer; the SRAM is solely used for this purpose in our design. Four images are displayed on the screen, each of which are 128x128 pixels: the original, uncompressed image; the Daubechies wavelet-transformed image, which represents the edges found in the image; the output of the decompression algorithm (which is formatted similarly to the wavelet-transformed image, highlighting edges); and the Daubechies inverse wavelet-transformed final output image.

6 Daubechie Wavelet Transform

An example of an uncompressed image is shown in Figure 5. The parts of the image (the color rendering now is just a scale factor and does not correspond to color) are shown in Figure 6.

Each color is done seperately. A single row of the image is transformed using the Daubechie Wavelet Transform[4] as shown in Appendix A.2.1. The 2d Transform (shown in Appendix A.2.2) is the 1d Transform (from Appendix A.2.1) applied to each row. The entire transform is a 2d transform applied first to the rows and then to the columns. Then repeated on each subset of the image until the image left to be transformed is of size 2×2, as shown in Appenix A.2.3. The resulting transformed image is shown in Figure 7.

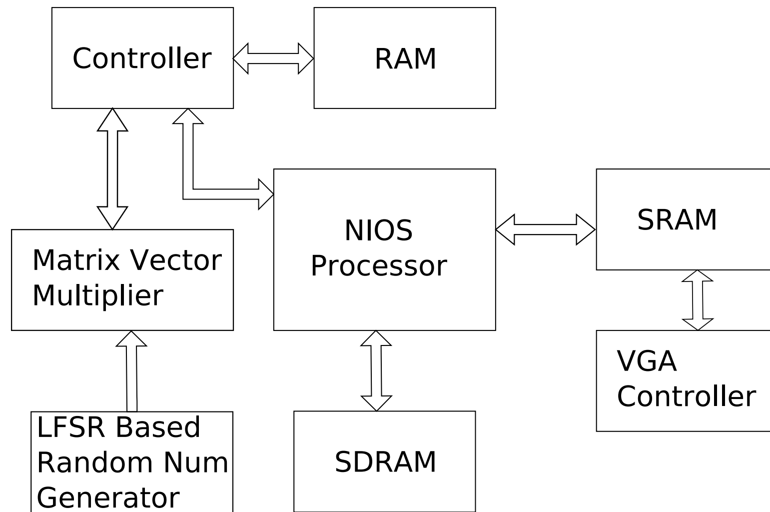

Figure 2: Overall Design Architecture

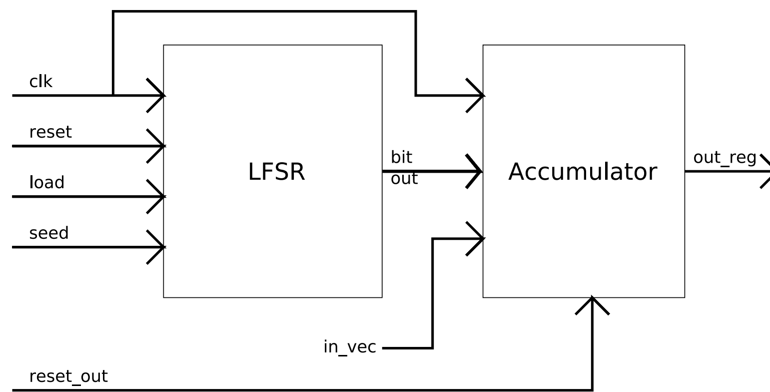

Figure 3: Accumulator Design

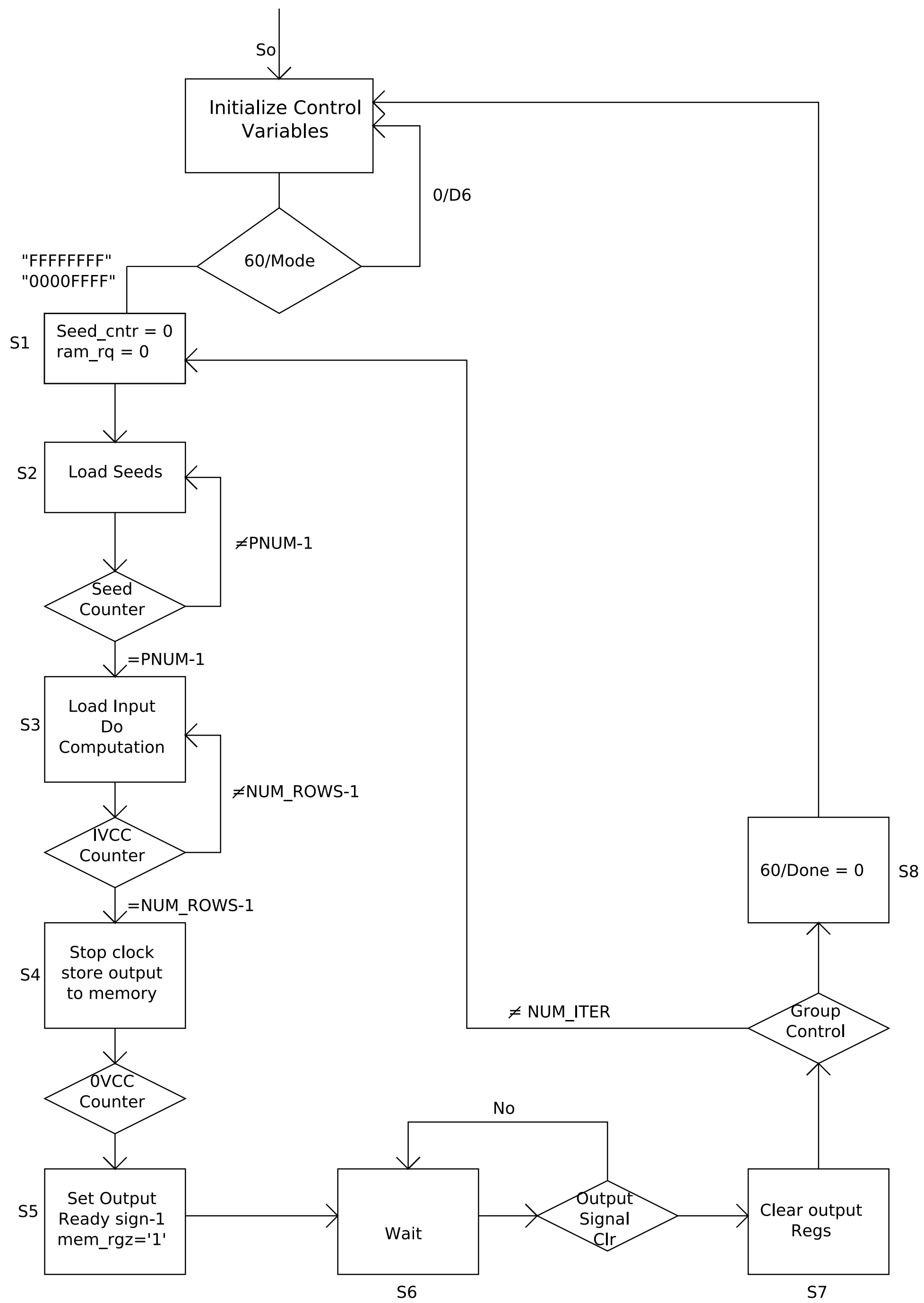

Figure 4: State Diagram

Figure 5: Uncompressed Image

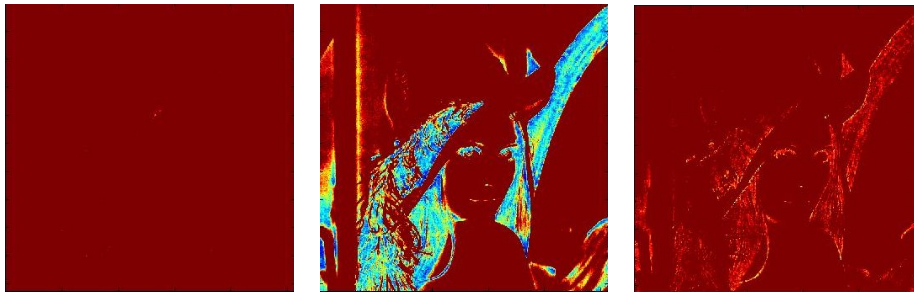

Figure 6: Uncompressed Image Parts: From left to right - Red, Green and Blue

The Red, Green and Blue subsets are all saved as one image into the SRAM. The reconstruction scheme is very similar to the transform: each row is inverse transformed in the matrix, and then each column; the inverse transform starting at the 2 left-most elements and then doubling every time until the entire matrix is inverse transformed. The Matlab code used to test this inverse transform is shown in Appendix A.2.4 A.2.6 A.2.7. The inverse transformed image is shown in Figure 8.

7 Results

Compression, decompression, and the forward and inverse Daubechies wavelet transforms were successfully implemented in Matlab. The decompression algorithm and inverse transform were then implemented correctly in C on the desktop, using fixed point calculations in the decompression to replace all floating point operations and quantities from the Matlab implementation. We then ported the C code to the Nios processor, in order to perform the algorithms

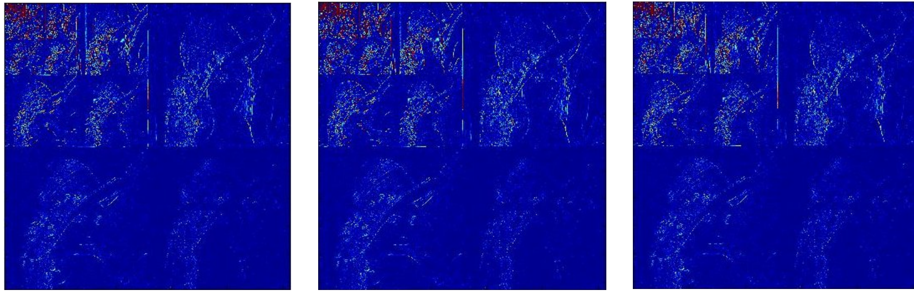

Figure 7: Transformed Image: Red, Green, Blue

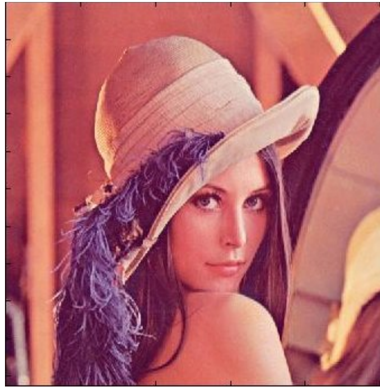

Figure 8: Inverse Transformed Image

purely in software without any hardware support; this worked properly for small images, but was not feasible for the full-sized 128x128 pixel image, primarily due to the memory constraints of the board on our ability to store the A matrix (which contains 4096 rows and 16384 columns when we use the full-sized image). In order to perform decompression on full-sized images, hardware support was required, and we were not able to successfully complete its design. We determined that our accumulation operation was not functioning properly, making the results of the decompression procedure invalid.

8 Conclusion

8.1 Who Did What

Keith Dronson: Helped with the mathematics and algorithm construction/understanding. Wrote the VGA controller code. Developed the matrix-vector multiplication block. Developed support code to make use of the Orthogonal Matching Pur-

suit (OMP) algorithm for performing decompression in Matlab.

Frank Zovko: Developed the C implementation of the SolveOMP algorithm. Wrote the code to interface it with the matrix-vector multiplication hardware block. Responsible for the random matrix generation in Matlab. Helped write the report.

Samuel Subbarao: Helped with the mathematics and algorithm construction/understanding. Implemented the Daubechie Wavelet Transform in Matlab. Implemented the Inverse Daubechie Wavelet Transform in Matlab and then C. Made the presentation. Helped write the report.

Federico Garcia: Assisted Keith with the VGA controller code. Assisted Samuel with the implementation of the Inverse Daubechie Transform in C.

8.2 Lessons Learned

Reflecting on our experience, one of our main observations is that we chose a project that required a bit too much effort to learn the fundamental mathematical concepts and algorithms. This cost us a significant amount of time that would have ideally been spent creating concrete plans for our implementation, particularly of our hardware. We left ourselves with insufficient time for debugging our hardware, and thus we were ultimately unable to make it work properly in accelerating our computation.

Additionally, we encountered problems in debugging our hardware, due to a relative lack of transparency in its operation. We would have been more successful debugging if we had relied more upon simulations and less upon trying to use the Nios IDE's console to relay information from hardware to the PC. While we were concerned that it would have been too complex for test benches to emulate the stimulus provided to the hardware by the Nios processor, we now feel that the overhead involved in setting this up would have been worthwhile.

While we did not succeed in meeting our ultimate goal of designing a hardware-assisted compressive sensing decompression engine, we were able to explore a relatively novel application of embedded systems design in a new and exciting field. We would finally like to thank Prof. Rui Castro of the Department of Electrical Engineering for providing tremendous assistance in the process of learning more about compressive sensing and the algorithms involved in it.

APPENDIX

A MatLab Code

A.1 Compression and Decompression

A.1.1 SolveOMP

```
function [sols, iters, activationHist] = SolveOMP(A, y, N, maxIters, lambdaStop,
solFreq, verbose, OptTol)
% SolveOMP: Orthogonal Matching Pursuit
```

```
% Usage
% [sols, iters, activationHist] = SolveOMP(A, y, N, maxIters, lambdaStop, solFreq,
verbose, OptTol)
% Input
% A           Either an explicit nxN matrix, with rank(A) = min(N,n)
%               by assumption, or a string containing the name of a
%               function implementing an implicit matrix (see below for
%               details on the format of the function).
% y           vector of length n.
%   N           length of solution vector.
% maxIters    maximum number of iterations to perform. If not
%               specified, runs to stopping condition (default)
%   lambdaStop  If specified, the algorithm stops when the last coefficient
%               entered has residual correlation <= lambdaStop.
%   solFreq     if =0 returns only the final solution, if >0, returns an
%               array of solutions, one every solFreq iterations (default 0).
%   verbose     1 to print out detailed progress at each iteration, 0 for
%               no output (default)
% OptTol      Error tolerance, default 1e-5
% Outputs
%  sols            solution(s) of OMP
%    iters           number of iterations performed
%    activationHist  Array of indices showing elements entering
%                    the solution set
% Description
%   SolveOMP is a greedy algorithm to estimate the solution
%   of the sparse approximation problem
%      min ||x||_0 s.t. A*x = b
%   The implementation implicitly factors the active set matrix A(:,I)
%   using Cholesky updates.
%   The matrix A can be either an explicit matrix, or an implicit operator
%   implemented as an m-file. If using the implicit form, the user should
%   provide the name of a function of the following format:
%     y = OperatorName(mode, m, n, x, I, dim)
%   This function gets as input a vector x and an index set I, and returns
%   y = A(:,I)*x if mode = 1, or y = A(:,I)'*x if mode = 2.
%   A is the m by dim implicit matrix implemented by the function. I is a
%   subset of the columns of A, i.e. a subset of 1:dim of length n. x is a
%   vector of length n is mode = 1, or a vector of length m is mode = 2.
% See Also
%   SolveLasso, SolveBP, SolveStOMP
%

if nargin < 8,
OptTol = 1e-5;
end
```

```
if nargin < 7,
    verbose = 1;
end
if nargin < 6,
    solFreq = 0;
end
if nargin < 5,
    lambdaStop = 0;
end
if nargin < 4,
    maxIters = length(y);
end

n = length(y);

% Parameters for linsolve function
% Global variables for linsolve function
global opts opts_tr machPrec
opts.UT = true;
opts_tr.UT = true; opts_tr.TRANSA = true;
machPrec = 1e-5;

% Initialize
x = zeros(N,1);
k = 1;
R_I = [];
activeSet = [];
sols = [];
res = y;
normy = norm(y);
resnorm = normy;
done = 0;

maxcorr = 0;
mincorr = 0;
maxres = 0;
minres = 0;

while ~done
    corr = A'*res;

    [maxcorr i] = max(abs(corr));
    newIndex = i(1);

    % Update Cholesky factorization of A_I
    [R_I, flag] = updateChol(R_I, n, N, A, activeSet, newIndex);
```

```
    activeSet = [activeSet newIndex];

    % Solve for the least squares update: (A_I'*A_I)dx_I = corr_I
    dx = zeros(N,1);

    z = linsolve(R_I,corr(activeSet),opts_tr);

    dx(activeSet) = linsolve(R_I,z,opts);
    x(activeSet) = x(activeSet) + dx(activeSet);

    % Compute new residual
    res = y - A(:,activeSet) * x(activeSet);

    resnorm = norm(res);

    if ((resnorm <= OptTol*normy) | ((lambdaStop > 0) & (maxcorr <= lambdaStop)))
        done = 1;
    end

    if verbose
        fprintf('Iteration %d: Adding variable %d\n', k, newIndex);
    end

    k = k+1;
    if k >= maxIters
        done = 1;
    end

    if done | ((solFreq > 0) & (~mod(k,solFreq)))
        sols = [sols x];
    end
end

iters = k;
activationHist = activeSet;
clear opts opts_tr machPrec


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [R, flag] = updateChol(R, n, N, A, activeSet, newIndex)
% updateChol: Updates the Cholesky factor R of the matrix
% A(:,activeSet)'*A(:,activeSet) by adding A(:,newIndex)
% If the candidate column is in the span of the existing
% active set, R is not updated, and flag is set to 1.
```

```
global opts_tr machPrec
flag = 0;

newVec = A(:,newIndex);

%newVec

if length(activeSet) == 0,
    R = sqrt(sum(newVec.^2));
else
    p = linsolve(R,A(:,activeSet)'*A(:,newIndex),opts_tr);

    q = sum(newVec.^2) - sum(p.^2);
    if (q <= machPrec) % Collinear vector
        flag = 1;
    else
        R = [R p; zeros(1, size(R,2)) sqrt(q)];
    end
end


%
% Copyright (c) 2006. Yaakov Tsaig
%

%
% Part of SparseLab Version:100
% Created Tuesday March 28, 2006
% This is Copyrighted Material
% For Copying permissions see COPYING.m
% Comments? e-mail sparselab@stanford.edu
%
```

A.1.2 Compression Decompression

```
% This code takes in an image from a file, compresses it, decompresses it,
% and displays the output.

clear all; close all;

% load the random A matrix (4096 rows x 16384 columns) from a file
load('A_final.mat'); A = double(A);

colormap(gray(256));axis('image');

% the number of data points in our compressed representation of the image
```

```
k = 4096;

% we zero out all but the N largest coefficients of the 16384 in the B matrix below
N = 1000;

% read in the image from a file and store it as a 128x128 matrix
x = double(imresize(imread('../images/Camera.tif'),.5));

% plot this original image in grayscale
subplot(1,2,1); image(x); colormap(gray(256)); axis('image');

% perform the Daubechies wavelet transform on the image and store in B (128x128)
% this produces B = Wx, where W represents the wavelet transform
h = daubcqf(4,'min'); B = mdwt(x,h,7);

% plot the wavelet-transformed image
subplot(1,2,2); image(B); colormap(gray(256)); axis('image');

% take the N largest coefficients from B and store them in the corresponding
positions in tmp
tmp = zeros(16384,1);
for i=1:N
    [Y I] = max(abs(B(:)));
    tmp(I) = B(I);
    B(I) = 0;
end

% compressed vector y = AB = AWt; k=4096 rows, 1 column
y = A*tmp;

% print this vector to a file so it can be fed to the decompression C code
print_matrix_to_file( y, 'y', 'y_4096.txt', 1);

% decompress the image and store as "theta" (16384 x 1)
[theta numIters] = SolveOMP(A,y,16384);

close all;

% plot the output of the decompression function (pre-inverse transform)
subplot(1,3,1); image(reshape(theta,128,128)); colormap(gray(256)); axis('image');

% take the output of the decompression function and display it as an image
imrecov = midwt(reshape(theta,128,128), h);
subplot(1,3,2); image(imrecov); colormap(gray(256)); axis('image');
```

A.2 Daubechie Wavelet Transform and Inverse Transform

A.2.1 1d Daubechie Wavelet Transform

```
function y=DWT_1d(S)

N=length(S);

s1=S(1:2:N-1)+sqrt(3)*S(2:2:N);
d1=S(2:2:N)-sqrt(3)/4*s1-(sqrt(3)-2)/4*[s1(N/2) s1(1:N/2-1)];
s2=s1-[d1(2:N/2) d1(1)];
s=(sqrt(3)-1)/sqrt(2)*s2;
d=(sqrt(3)+1)/sqrt(2)*d1;

y=[s d];
```

A.2.2 2d Daubechie Wavelet Transform

```
function theta = transform2(x)

x_size = size(x);
for i=1:1:x_size(1)
    theta(i,:)=DWT_1d(x(i,:));
end
```

A.2.3 Full Image Daubechie Wavelet Transform

```
function theta = transform(x)

x_size = size(x);
n = x_size(2);
while n > 1
    x1 = transform2(x(1:n, 1:n));
    x1t = x1';
    x2 = transform2(x1t);
    x(1:n, 1:n) = x2';
    n = n/2;
    clear x1;
    clear x2;
    clear x1t;
end

theta = x;
```

A.2.4 1d Inverse Daubechie Wavelet Transform

```
function S=iDWT_1d(y)
```

```
N=length(y);

s=y(1:N/2);
d=y(N/2+1:N);

d1=d/((sqrt(3)+1)/sqrt(2));
s2=s/((sqrt(3)-1)/sqrt(2));
s1=s2+cpv(d1,1);
S(2:2:N)=d1+sqrt(3)/4*s1+(sqrt(3)-2)/4*cpv(s1,-1);
S(1:2:N-1)=s1-sqrt(3)*S(2:2:N);
```

A.2.5 CPV code

```
function P=cpv(S,k)

if k>0
    P=[S(k+1:end) S(1:k)];
elseif k<0
    P=[S(end+k+1:end) S(1:end+k)];
end
```

A.2.6 2d Inverse Daubechie Wavelet Transform

```
function x = invtransform2(theta)

theta_size = size(theta);
for i=1:1:theta_size(1)
    x(i,:)=iDWT_1d(theta(i,:));
end
```

A.2.7 Full Image Inverse Daubechie Wavelet Transform

```
function x = invtransform(theta)

theta_size = size(theta);
n = theta_size(2);
index = 2;

while index <= n
    theta1 = invtransform2(theta(1:index, 1:index)');
    theta1t = theta1';
    theta (1:index, 1:index)= invtransform2(theta1t);
    index = index * 2;
    clear theta1;
    clear theta1t;
```

```
end

x = theta;
```

B C Code

B.1 Daubechie Wavelet Inverse Transform

B.1.1 Driver

```
#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include <io.h>
#include <math.h>
#include <stdlib.h>
#include "../invtransform.c"
#include "../decomp.c"

#define WRITE_TO_FB(base, address, data) \
        IOWR_16DIRECT(base, address, data)

int main()
{
  // original camera image
  unsigned short original[] = {20083, ... , 15855}; //all the values not
  included for printing purposes

  // image after performing Daubechies wavelet transform (in Matlab)
  // not used for computation- only for display
  unsigned short before[] = {2114, ... , 0}; //all the values not included
  for printing purposes

  int row, col, jj, ii;

  // set all the pixels in the VGA display to black
  for ( ii = 0; ii < 512*256; ii++ )
        IOWR_8DIRECT(VGA_BASE,ii,0);

  // write the original image and its wavelet transform to the first two boxes
  on the VGA
  // the images are interleaved in memory... we store row1 of image1, then row1
  of image2, ...
  for (row = 0; row < 128; row++)
  {
     for (col = 0; col < 128; col++)
```

```
   {
      IOWR_16DIRECT(VGA_BASE-4, 2*(128*(2*row)+col), original[row*128+col]);
      IOWR_16DIRECT(VGA_BASE-4, 2*(128*(2*row+1)+col), before[row*128+col]);
   }
}

  for (jj = K-1; jj >=0; jj--)
      IOWR_16DIRECT(IVEC_BASE, jj*2, 0);

  // compressed image from Matlab, to be decompressed by our code
  int y[] = {15326.209597, ... , -15955.310135}; //all the values not included
  for printing purposes

  int solveomp_solution[N];

  int i;
  for (i = 0; i < N; i++)
      solveomp_solution[i] = 0;

  // decompress the input
  decomp(y, solveomp_solution);

// decompression output (pre-Daubechies inverse wavelet transform)
// this is hard-coded for the purpose of our demo
int decomp_out_img[] = {1057, ... , 0}; //all the values not included for
printing purposes

// write decomp_out_img to the SRAM so it can be displayed in the lower left
VGA box
for (row = 0; row < 128; row++)
   for (col = 0; col < 128; col++)
      IOWR_16DIRECT(VGA_BASE-4, 2*(128*128*2 + 128*(2*row)+col),
      decomp_out_img[row*128+col]);

// theta is the output of the decomp function
// this is hard-coded for the purpose of our demo, rather than using solveomp_solution
from above
double theta[] = {15196.78906, ... , 0.00000}; //all the values not included for
printing purposes

// turn the 1D array into a 2D array
double sq_theta[128][128];

int j;

for (i=0; i<128; i++)
```

```
    for(j=0; j<128; j++)
        sq_theta[i][j] = theta[(128*i)+j];

   // inverse Daubechies transform on sq_theta gives us the final image
   invtransform(sq_theta, 128, 128 );

   // save the final image to SRAM for the VGA display
   for (row = 0; row < 128; row++)
  {
     for (col = 0; col < 128; col++)
     {
        if (sq_theta[col][row] < 0)
            IOWR_16DIRECT(VGA_BASE-4, 2*(128*128*2 + 128*(2*row+1)+col), (((int)((sq_theta[c
        else
            IOWR_16DIRECT(VGA_BASE-4, 2*(128*128*2 + 128*(2*row+1)+col), (((int)((sq_theta[c
     }
  }

    return 0;
}
```

B.1.2 Header

```
#define IVEC_BASE (MATRIX_VECTOR_MULT_INST_BASE + 4*1028)
#define OVEC_BASE (MATRIX_VECTOR_MULT_INST_BASE + 4*3076)
#define NEWVEC_BASE (MATRIX_VECTOR_MULT_INST_BASE + 4*3204)
#define P_NUM 128
#define SPARSITY 1000
#define K 4096
#define N 16384
```

B.1.3 Decompression functions

```
#include "./decmp/header.h"

// square root of the sum of the squares (L2 norm)
int vec_norm(int *vector, int vec_size)
{
    long int cur_element;
    long long int sum = 0;
    long int product = 0;

    int i;

    for(i = 0; i < vec_size; i++)
    {
```

```
        cur_element = vector[i];
        product = cur_element * cur_element;
        sum += product;
    }

    return (int)sqrt(sum);
}

//solves Ax = b for a UT cholesky decomposed matrix using backsubstitution
void linsolve(int R_I[SPARSITY][SPARSITY], int *vec, int activeSetSize, int
*result, char trans_mat)
{
  int i,k;
  int sum;

  if ( !trans_mat )
  {
    for(i = activeSetSize-1; i >= 0; i--)
    {
      for(sum = vec[i], k = i+1; k < activeSetSize; k++)
        sum -= R_I[i][k]*result[k];

      result[i] = sum/R_I[i][i];
    }
  }
  else
  {
     int i;

     for(i = 0; i < activeSetSize; i++)
     {
        for(sum = vec[i], k = i-1; k >= 0; k--)
           sum -= R_I[k][i]*result[k];

        result[i] = sum/R_I[i][i];
     }
  }

  return;
}

// updateChol: Updates the Cholesky factor R of the matrix
A(:,activeSet)'*A(:,activeSet) by adding A(:,newIndex)
void updateChol(int R[SPARSITY][SPARSITY], char A[K][SPARSITY], int *activeSet,
int activeSetSize, int newIndex)
{
```

```
int i,j,ix;
int sum = 0, q = 0;

int newVec[K];
int result[activeSetSize];
int p[activeSetSize];

char tmp = 0;

int newVecSqrSum = 0;

// INTERFACE WITH HARDWARE

// add a column to the active set of A

// write newIndex to its spot in the block RAM
IOWR_32DIRECT(NEWVEC_BASE, j*4, newIndex);

// write something other than EMPTY to ctrl_reg(2)
IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 8, 0x11110000);

// write MODE1 to ctrl_reg(1)
IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 4, 0xFFFF0000);

// write GO to ctrl_reg(0)
IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 0, 0xFFFFFFFF);

unsigned int notdone = 0;
while(notdone != 0xffffffff)
     notdone = IORD_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 8);

// fill A[*][newIndex] with the LFSR data from block RAM
for (i = 0; i < K; i++)
      A[i][newIndex] = 2*IORD_32DIRECT(IVEC_BASE, i*4) - 1;

for( i = 0; i < K; i++ )
{
  newVec[i] = A[i][newIndex];
  newVecSqrSum += newVec[i]*newVec[i];
}

if( !activeSetSize )
{
  for( i = 0; i < K; i++ )
  {
    sum += (int)pow(newVec[i],2);
```

```
    }

    R[0][0] = (int)sqrt(sum);
  }

  else
  {
    // hard-coding mat_vec_mul( (void **)A, newVec, n, activeSetSize, 1, result,
1 );
    for ( i = 0; i < activeSetSize; i++ )
    {
      ix = activeSet[i];
      result[i] = 0;
      for ( j = 0; j < K; j++ )
      {
        tmp = A[j][ix];

        if (tmp == 1)
            result[i] += newVec[j];
        else
            result[i] -= newVec[j];
      }
    }

    // solve R*p = result
    linsolve( R, result, activeSetSize, p, 1 );

    for( i = 0; i < activeSetSize; i++ )
    {
      q += p[i]*p[i];
      R[i][activeSetSize] = p[i];
    }

    R[activeSetSize][activeSetSize] = (int)sqrt(newVecSqrSum - q);
  }

  return;
}

void add_dx_activeSet_to_x_activeSet(int *dx, int *x, int *activeSet)
{
    int j;
    for(j = 0; activeSet[j] != -1; j++)
        x[activeSet[j]] += dx[activeSet[j]];

    return;
```

```
}

void res_update(int *y, int n, char A[K][SPARSITY], int *x, int *activeSet, int
*res)
{
    // Matlab code: res = y - A(:,activeSet) * x(activeSet)
    // res, y are nx1

    // this uses the subset of the random A matrix generated by the
matrix_vector_mult block

    int sum;
    char tmp = 0;

    int j, k;
    for(j = 0; j < n; j++)
    {
        sum = 0;

        for(k = 0; activeSet[k] != -1; k++)
        {
            tmp = A[j][activeSet[k]];

            if (tmp == 1)
                sum += x[activeSet[k]];
            else
                sum -= x[activeSet[k]];
        }

        res[j] = y[j] - sum;
    }

    return;
}

// takes in a compressed data set y [k rows, 1 col] and produces solution x [N
rows, 1 col]
// see Matlab code; this is mostly a direct translation
// except for where we use the hardware
void decomp(int *y, int *x)
{
    // K = length of y
    // N = length of solution (number of pixels in image)

    int i, j;
    int ind = 1;
```

```
// reduced-size A matrix stores only the active set
// small enough that we can store it in memory
char A[K][SPARSITY];
for (i = 0; i < K; i++)
    for (j = 0; j < SPARSITY; j++)
A[i][j] = 0;

int R_I[SPARSITY][SPARSITY];

for (i = 0; i < SPARSITY; i++)
{
    for (j = 0; j < SPARSITY; j++)
        R_I[i][j] = 0;
}

int activeSetSize = 0;

int activeSet[N];

for (i = 0; i < N; i++)
{
    activeSet[i] = -1;
}

int res[K];

for (i = 0; i < K; i++)
    res[i] = y[i];

int normy = vec_norm(y, K);
int resnorm = normy;

char done = 0;

// corr = A^T * res; A is nxN, so A^T is Nxn, and res is nx1
// so corr is Nx1
int corr[N];

// subset of corr vector with indices from activeSet
int corr_activeSet[N];

for (j = 0; j < N; j++)
    corr_activeSet[j] = 0;

i = 0;
```

```
    int maxcorr = 0;

    int newIndex;

    int dx[N];
    for (j = 0; j < N; j++)
        dx[j] = 0;

    int dx_activeSet[N];
    for (j = 0; j < N; j++)
        dx_activeSet[j] = 0;

    // stores result of first linsolve in updateChol function
    int z[N];
    for (j = 0; j < N; j++)
        z[j] = 0;

    unsigned int notdone, iter;

    while (done == 0)
    {

////////////////////////////////////////////////////////////////////
// this code multiplies A^T by res and assigns the result to corr

        // write res to the input vector space of matrix_vector_mult block
        for (j = 0; j < K/2; j++)
             IOWR_32DIRECT(IVEC_BASE, j*4, res[j]);

// set control signals for mode 0 of the matrix multiplier

// write something other than EMPTY to ctrl_reg(2)
        IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 8, 0x11110000);

        // write MODE0 to ctrl_reg(1)
        IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 4, 0x0000FFFF);

        // write GO to ctrl_reg(0)
        IOWR_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 0, 0xFFFFFFFF);

        notdone = 0;
        while(notdone != 0xffffffff)
            notdone = IORD_32DIRECT(MATRIX_VECTOR_MULT_INST_BASE, 8);

        // fill the whole corr array by getting P_NUM elements at a time
        while (iter * P_NUM < N)
```

```
        {
            for (j = 0; j < P_NUM; j++)
                corr[iter*P_NUM + j] = IORD_32DIRECT(OVEC_BASE, j*4);

            iter++;
        }

        maxcorr = 0;

        // [maxcorr i] = max(abs(corr))
        for (j = 0; j < N; j++)
        {
            if ((int)abs(corr[j]) > maxcorr)
            {
                i = j;
                maxcorr = (int)abs(corr[j]);
            }
        }

/////////////////////////////////////////////////////////////////////

        newIndex = i;

        // Update Cholesky factorization of A_I
        updateChol(R_I, A, activeSet, activeSetSize, newIndex);

        activeSet[activeSetSize] = newIndex;
        activeSetSize++;

        // Solve for the least squares update: (A_I'*A_I)dx_I = corr_I
        for (j = 0; j < N; j++)
            dx[j] = 0;

        for (j = 0; j < activeSetSize; j++)
            corr_activeSet[j] = corr[activeSet[j]];

        linsolve(R_I, corr_activeSet, activeSetSize, z, 1);
        linsolve(R_I, z, activeSetSize, dx_activeSet, 0);

        for(j = 0; j < activeSetSize; j++)
            dx[activeSet[j]] = dx_activeSet[j];

        add_dx_activeSet_to_x_activeSet(dx, x, activeSet);

        // compute new residual
        res_update(y, K, A, x, activeSet, res);
```

```
        resnorm = vec_norm(res, K);
        printf("resnorm = %d\n",resnorm);

        done = (resnorm <= 1000) ? 1 : 0;

        printf("Iteration %d: Adding variable %d\n", ind, newIndex);

        ind += 1;
    }

    return;
}
```

B.1.4 CPV Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void cpv(double S[], int S_length, int k){

    double P[S_length];
    int i;
    if (k > 0){
        // Mimicing P=[S(k+1:end) S(1:k)]
        for (i=k;i<S_length;i++){
            P[i-k] = S[i];
        }
        for (i=0; i<k; i++){
            P[S_length-k+i]=S[i];
        }
    }
    else if (k < 0){
        //mimicing P=[S(end+k+1:end) S(1:end+k)]
        for (i=S_length + k ;i<S_length;i++){
            P[i-S_length-k] = S[i];
        }
        for (i=0; i<(S_length+k); i++){
            P[(-1*k)+i]=S[i];
        }
    }
    if (k != 0){
     for (i=0; i<S_length; i++){
    S[i] = P[i];
     }
```

```
    }
}
```

B.1.5 Transpose Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void transpose(double theta[][128], int num_rows, int num_cols){
int i, j;
double temp;
for (i=0; i<num_rows; i++){
for (j=i; j<num_cols; j++){
temp = theta[i][j];
theta[i][j] = theta[j][i];
theta[j][i] = temp;
}
}
}
```

B.1.6 1d Inverse Daubechie Wavelet Transform

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "cpv.c";

void iDWT_1d(double y[], int y_length){
int i;
//mimic s=y(1:N/2);
double s[(y_length/2)];
for (i=0; i<(y_length/2); i++){
s[i] = y[i];
}
//mimic d=y(N/2+1:N);
double d[(y_length/2)];
for (i=(y_length/2); i<y_length; i++){
d[i - (y_length/2)] = y[i];
}
//mimic d1=d/((sqrt(3)+1)/sqrt(2));
double d1_for_cpv[(y_length/2)];
for (i=0; i< (y_length/2); i++){
d[i] = d[i]/((sqrt(3)+1)/sqrt(2));
d1_for_cpv[i] = d[i];
}
```

```
//mimic s2=s/((sqrt(3)-1)/sqrt(2));
for (i=0; i< (y_length/2); i++){
s[i] = s[i]/((sqrt(3)-1)/sqrt(2));
}
//mimic s1=s2+cpv(d1,1);
cpv(d1_for_cpv, (y_length/2), 1);

double s1[(y_length/2)];
double s1_for_cpv[(y_length/2)];
for (i=0; i<(y_length/2); i++){
s1[i] = s[i] + d1_for_cpv[i];
s1_for_cpv[i] = s1[i];
}
//mimic S(2:2:N)=d1+sqrt(3)/4*s1+(sqrt(3)-2)/4*cpv(s1,-1);
cpv(s1_for_cpv, (y_length/2), -1);
for (i=0; i<(y_length/2); i++){
y[1+(2*i)] = d[i] + sqrt(3) / 4 * s1[i] + (sqrt(3)-2) / 4 * s1_for_cpv[i];
}
//mimic S(1:2:N-1)=s1-sqrt(3)*S(2:2:N);
for (i=0; i<(y_length/2); i++){
y[2*i] = s1[i] - sqrt(3)*y[1+(2*i)];
}
}
```

B.1.7 2d Inverse Daubechie Wavelet Transform

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "iDWT_1d.c";

void invtransform2(double theta[][128], int num_rows, int num_cols){
int i, j;
double theta_row[num_cols];
/*mimic:
for i=1:1:theta_size(1)
     x(i,:)=iDWT_1d(theta(i,:));
end
*/
for (i=0; i<num_rows; i++){
for (j=0; j<num_cols; j++){
theta_row[j] = theta[i][j];
}
iDWT_1d(theta_row, num_cols);
for (j=0; j<num_cols; j++){
theta[i][j] = theta_row[j];
```

```
}
}
}
```

B.1.8 Full Image Inverse Daubechie Wavelet Transform

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "invtransform2.c";
#include "transpose.c";

void invtransform(double theta[][128], int num_rows, int num_cols){
/* mimic:
theta_size = size(theta);
n = theta_size(2);
index = 2;

while index <= n
    theta1 = invtransform2(theta(1:index, 1:index)');
    theta1t = theta1';
    theta (1:index, 1:index)= invtransform2(theta1t);
    index = index * 2;
    clear theta1;
    clear theta1t;
end
*/

int index = 2;
while (index <= num_cols) {
transpose(theta, index, index);
invtransform2(theta, index, index);
transpose(theta, index, index);
invtransform2(theta, index, index);
index = 2*index;
}
}
```

C VHDL Code

C.1 Matrix Vector Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
--------------Matrix Vector Multiplier------------------

--Block containing output accumulators, LFSRs needed for
--matrix generation, block RAM and ROM.  Operation of block
--is controlled using a state machine.

---------------------------------------------------------

entity matrix_vector_mult is
  port
    (
      state_out : out std_logic_vector(4 downto 0);

      --avalon bus signals
      clk        : in  std_logic;
      reset      : in  std_logic;
      read       : in  std_logic;
      write      : in  std_logic;
      chipselect : in  std_logic;
      address    : in  std_logic_vector(13 downto 0);
      writedata  : in  std_logic_vector(31 downto 0);
      readdata   : out std_logic_vector(31 downto 0)
    );

end matrix_vector_mult;

architecture rtl of matrix_vector_mult is
  -- ROM generated by wizard stores seeds for all the parallel LFSRs
  component rom_test
    port
      (
        address : in  std_logic_vector (13 downto 0);
        clock   : in  std_logic;
        q       : out std_logic_vector (31 downto 0)
        );
  end component;

  component column_module
    port
      (
        clk        : in  std_logic;
        acc_active : in  std_logic;
        reset      : in  std_logic;
        reset_acc  : in  std_logic;
        load       : in  std_logic;
```

```
      seed       : in  std_logic_vector(31 downto 0);
      i_vec      : in  std_logic_vector(15 downto 0);
      o_reg      : out std_logic_vector(31 downto 0)
      );
end component;

component single_port_ram
  generic
    (
      DATA_WIDTH :     natural   := 32;
      ADDR_WIDTH :     natural   := 12
      );
  port
    (
      clk        : in  std_logic;
      addr       : in  natural range 0 to 2**ADDR_WIDTH - 1;
      data       : in  std_logic_vector((DATA_WIDTH-1) downto 0);
      we         : in  std_logic := '1';
      q          : out std_logic_vector((DATA_WIDTH -1) downto 0)
      );
end component;

component lfsr
  port
    (
      clk, reset, load, enable : in  std_logic;
      seed                     : in  std_logic_vector(31 downto 0);
      bit_out                  : out std_logic
      );
end component;

constant DATA_WIDTH : natural := 32;
constant ADDR_WIDTH : natural := 12;
constant P_NUM      : natural := 128;

type o_register is array(0 to P_NUM-1) of std_logic_vector(31 downto 0);
type seed_register is array(0 to P_NUM-1) of std_logic_vector(31 downto 0);

--signals for column_module
signal acc_active : std_logic                     := '0';
signal reset_lfsr : std_logic                     := '0';
signal reset_acc  : std_logic                     := '0';
signal load       : std_logic                     := '0';
signal i_vec      : std_logic_vector(15 downto 0) := (others => '0');
signal o_reg      : o_register;
signal seed       : seed_register;
```

```
--signals for seed ROM
signal seed_mem : std_logic_vector(31 downto 0);

--signals for single_port_ram
signal addr : natural range 0 to (2**ADDR_WIDTH-1);
signal data : std_logic_vector((DATA_WIDTH-1) downto 0);
signal we   : std_logic := '1';
signal q    : std_logic_vector((DATA_WIDTH-1) downto 0);

--signals for writing reading from single_port_ram to controller
signal addr_cntrl : natural range 0 to (2**ADDR_WIDTH-1);
signal data_cntrl : std_logic_vector((DATA_WIDTH-1) downto 0);
signal we_cntrl   : std_logic := '1';
signal q_cntrl    : std_logic_vector((DATA_WIDTH-1) downto 0);

--signal for memory arbitration
signal ram_rq : std_logic := '1';

-- bit coming out of LFSR used to generate single column of A matrix
signal extra_lfsr_bit_out : std_logic;

--signals for state machine
constant NUM_ITER : integer := 16384/P_NUM;  --number of times we have to
assign new seeds to cover entire matrix
constant NUM_ROWS : integer := 4096;  --number of rows in matrix
constant NUM_COLS : integer := 16384;  --number of columns in matrix

-- RAM contents:
-- status registers:     0x000 to 0x003 (0-3 [4 words])
-- active set:           0x004 to 0x403 (4-1027 [1028 words])
-- input vector:         0x404 to 0xC03 (1028-3075 [2048 words = 4096 half-words])
-- output vector:        0xC04 to 0xC83 (3076-3203 [128==P_NUM words])
-- new active set index: 0xC84 (3204)

-- offsets of the above memory regions
constant aset_offset   : integer := 4;
constant ivec_offset   : integer := 1028;
constant ovec_offset   : integer := 3076;
constant newind_offset : integer := 3204;

type states is (S0, S1, S2, S3a, S3b, S4, S5, S6, S7, S8, S9a, S9b, S9c);

signal state, next_state         : states;
signal grp_cntr, next_grp_cntr   : integer range 0 to NUM_ROWS;  --number of
groups to cover all columns
```

```
  signal ivec_cntr, next_ivec_cntr : integer range 0 to NUM_ROWS-1;  --number
  of input vector
  signal seed_cntr, next_seed_cntr : integer range 0 to P_NUM-1;  --number of
  seeds that must be loaded
  signal ovec_cntr, next_ovec_cntr : integer range 0 to P_NUM-1;  --number of
  output registers
  signal seed_transfer             : std_logic := '0';

  --special control registers

  -- old values- 32 bits
  constant GO    : std_logic_vector((DATA_WIDTH-1) downto 0) := X"FFFFFFFF";
  constant DONE  : std_logic_vector((DATA_WIDTH-1) downto 0) := X"00030000";
  constant FULL  : std_logic_vector((DATA_WIDTH-1) downto 0) := X"FF00FFFF";
  constant EMPTY : std_logic_vector((DATA_WIDTH-1) downto 0) := X"00000000";
  constant MODE0 : std_logic_vector((DATA_WIDTH-1) downto 0) := X"0000FFFF";
  constant MODE1 : std_logic_vector((DATA_WIDTH-1) downto 0) := X"FFFF0000";

  type cntrl_register is array(0 to 2) of std_logic_vector((DATA_WIDTH-1) downto 0);

  signal cntrl_reg : cntrl_register;

  signal waitreq_reg1, waitrequest_int : std_logic;

begin

  ram_inst0 : single_port_ram
    generic map
    (
      DATA_WIDTH => 32,
      ADDR_WIDTH => 12
      )
    port map
    (
      clk        => clk,
      addr       => addr,
      data       => data,
      we         => we,
      q          => q
      );

  --Seed ROM holds the intitial seed values for each of the 16384 columns.
  --Values are loaded from a file.
  seed_rom_inst0 : rom_test
    port map
    (
```

```
      address => std_logic_vector(to_unsigned(addr_cntrl, address'length)),
      clock   => clk,
      q       => seed_mem
      );

  --Generate N elements for doing parallel computations
  col_gen   : for n in 0 to (P_NUM-1) generate
    col_map : column_module port map
      (
        clk        => clk,
        acc_active => acc_active,
        reset      => reset_lfsr,
        reset_acc  => reset_acc,
        load       => load,
        seed       => seed(n),
        i_vec      => i_vec,
        o_reg      => o_reg(n)
        );
  end generate;

--LFSR needed to generate column needed by processor.
extra_lfsr0 : lfsr port map
    (
      enable  => '1',
      clk     => clk,
      reset   => reset_lfsr,
      load    => load,
      seed    => seed_mem,
      bit_out => extra_lfsr_bit_out
    );

  --Helpful for debugging
  process(state)
  begin
    case state is
      when S0     =>
        state_out <= "00000";
      when S1     =>
        state_out <= "00001";
      when S2     =>
        state_out <= "00010";
      when S3a    =>
        state_out <= "00011";
      when S3b    =>
        state_out <= "00100";
      when S4     =>
```

```
      state_out <= "00101";
    when S5     =>
      state_out <= "00110";
    when S6     =>
      state_out <= "00111";
    when S7     =>
      state_out <= "01000";
    when S8     =>
      state_out <= "01001";
    when S9a     =>
      state_out <= "01010";
    when S9b     =>
      state_out <= "01011";
    when S9c     =>
      state_out <= "01100";
    when others =>
      state_out <= "11111";
  end case;
end process;

--Memory control processes
--process for avalon bus
process(clk)
begin
  if(rising_edge(clk)) then
    if ram_rq = '1' then
      if chipselect = '1' then
        if write = '1' then
          we                                        <= '1';
          data                                      <= writedata;
          if (address = std_logic_vector(to_unsigned(0, address'length))
              or address = std_logic_vector(to_unsigned(1, address'length))
              or address = std_logic_vector(to_unsigned(2, address'length)))
              then
            cntrl_reg(to_integer(unsigned(address))) <= writedata;
          end if;
        elsif read = '1' then
          we                                        <= '0';
          readdata                                  <= q;
        end if;
        addr                                        <= to_integer
        (unsigned(address));
      end if;
    else
      if we_cntrl = '1' then
        we                                          <= '1';
```

```
        data                                                  <= data_cntrl;
      else
        we                                                    <= '0';
        q_cntrl                                               <= q;
      end if;
      addr                                                    <= addr_cntrl;
    end if;
  end if;

end process;

-- state machine controller
process(clk)
begin

  if (rising_edge(clk)) then
    seed_cntr        <= next_seed_cntr;
    ivec_cntr        <= next_ivec_cntr;
    ovec_cntr        <= next_ovec_cntr;
    grp_cntr         <= next_grp_cntr;
    state            <= next_state;
    if seed_transfer  <= '1' then
      seed(seed_cntr) <= seed_mem;
    end if;
  end if;

end process;

process(reset, state, seed_cntr, ivec_cntr, ovec_cntr, grp_cntr, cntrl_reg,
o_reg, q_cntrl, addr_cntrl)
begin
  if reset = '1' then
    next_state     <= S0;
    next_seed_cntr <= 0;
    next_ivec_cntr <= 0;
    next_ovec_cntr <= 0;
    next_grp_cntr  <= 0;
    acc_active     <= '0';
    ram_rq         <= '1';
    we_cntrl       <= '0';
    load           <= '0';
    reset_lfsr     <= '1';
    reset_acc      <= '1';
    addr_cntrl     <= 0;
    seed_transfer  <= '0';
    i_vec          <= (others => '0');
```

```
    data_cntrl     <= (others => '0');
  else
--Default values
    next_seed_cntr <= seed_cntr;
    next_ivec_cntr <= ivec_cntr;
    next_ovec_cntr <= ovec_cntr;
    next_grp_cntr  <= grp_cntr;
    acc_active     <= '0';
    ram_rq         <= '0';
    we_cntrl       <= '0';
    load           <= '0';
    reset_lfsr     <= '0';
    reset_acc      <= '0';
    next_state     <= state;
    addr_cntrl     <= 0;
    seed_transfer  <= '0';
    i_vec          <= (others => '0');
    data_cntrl     <= (others => '0');

    case state is
      when S0  =>
        ram_rq         <= '1';
        reset_acc      <= '1';
        reset_lfsr     <= '1';
        if cntrl_reg(0) = GO then
          --add other modes here
          if cntrl_reg(1) = MODE0 then
            next_state <= S1;
          elsif cntrl_reg(1) = MODE1 then
            next_state <= S9a;
          else
            next_state <= S0;
          end if;
        end if;
      when S1  =>
        next_seed_cntr <= 0;
        next_ivec_cntr <= 0;
        next_ovec_cntr <= 0;
        next_grp_cntr  <= 0;
        ram_rq         <= '0';
        next_state     <= S2;
      when S2  =>
        seed_transfer  <= '1';
        addr_cntrl     <= grp_cntr + seed_cntr;
        next_seed_cntr <= seed_cntr + 1;
        load           <= '1';
```

```
        if seed_cntr = (P_NUM-1) then
          next_state   <= S3a;
        else
          next_state   <= S2;
        end if;
      -- don't increment ivec_cntr in this state
      when S3a =>
        acc_active     <= '1';
        we_cntrl       <= '0';
        addr_cntrl     <= ivec_offset + ivec_cntr;
        i_vec          <= q_cntrl(15 downto 0);
        next_ivec_cntr <= ivec_cntr;
        next_state     <= S3b;
      -- do increment ivec_cntr in this state
      when S3b =>
        acc_active     <= '1';
        we_cntrl       <= '0';
        addr_cntrl     <= ivec_offset + ivec_cntr;
        i_vec          <= q_cntrl(31 downto 16);
        next_ivec_cntr <= ivec_cntr + 1;
        if ivec_cntr = (NUM_ROWS/2)-1 then
          next_state   <= S4;
        else
          next_state   <= S3a;
        end if;
      when S4 =>
--Write the output back to Block RAM
        we_cntrl <= '1';
        addr_cntrl <= ovec_offset + ovec_cntr;
        data_cntrl <= o_reg(ovec_cntr);
        next_ovec_cntr <= ovec_cntr + 1;
        if ovec_cntr = (P_NUM-1) then
          next_state <= S5;
        else
          next_state <= S4;
        end if;
      when S5 =>
--Signal processor that the ouput is ready.
        we_cntrl <= '1';
        addr_cntrl <= 2;
        data_cntrl <= FULL;
        next_state <= S6;
      when S6 =>
        ram_rq <= '1';
        if cntrl_reg(2) = EMPTY then
  --processor has successfully read back the output set.
```

```
            next_state <= S7;
          else
            next_state <= S6;
          end if;
        when S7 =>
  --Do next set of columns.
          reset_acc <= '1';
          next_grp_cntr <= grp_cntr + P_NUM;
          next_state <= S8;
        when S8 =>
          if grp_cntr = NUM_COLS then
            --Finished
next_state <= S0;
          else
            next_state <= S1;
          end if;
        when S9a =>
          -- address the RAM to get the index of the new element of the A matrix
          addr_cntrl <= newind_offset;

          next_state <= S9b;
        when S9b =>
          -- address the seed ROM with the index retrieved from RAM
          addr_cntrl <= to_integer(unsigned(data((ADDR_WIDTH-1) downto 0)));
          next_ivec_cntr <= 0;
          load <= '1';
        when S9c =>
          addr_cntrl <= ivec_offset + ivec_cntr;
          next_ivec_cntr <= ivec_cntr + 1;
          if (extra_lfsr_bit_out = '0') then
             data_cntrl <= "00000000000000000000000000000001";
          else
             data_cntrl <= "11111111111111111111111111111111";
          end if;
          if (ivec_cntr = 4095) then
            next_state <= S0;
          else
            next_state <= S9c;
          end if;
      end case;
    end if;

  end process;

end rtl;
```

C.2 Column Module

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


----------------Column Module--------------------

--The parallel element of our matrix-vector
--multiplier block.  Each module is a combination
--of a LFSR and accumulator block capable of
--generating and producing an output for a row
--in the transposed random matrix.

------------------------------------------------

entity column_module is

  port
    (
      clk        : in  std_logic;
      acc_active : in  std_logic;
      reset      : in  std_logic;
      reset_acc  : in  std_logic;
      load       : in  std_logic;
      seed       : in  std_logic_vector(31 downto 0);
      i_vec      : in  std_logic_vector(15 downto 0);
      o_reg      : out std_logic_vector(31 downto 0)
      );

end column_module;


architecture df of column_module is

  --component declaration for LFSR
  component lfsr
    port
      (
        clk, reset, load, enable : in  std_logic;
        seed                     : in  std_logic_vector(31 downto 0);
        bit_out                  : out std_logic
        );
  end component;
```

```
  --component declaration for Accumulator
  component accumulator
    port
      (
        clk, acc_active, sel, reset : in  std_logic;
        i_vec                       : in  std_logic_vector(15 downto 0);
        o_reg                       : out std_logic_vector(31 downto 0)
        );
  end component;

  signal sel : std_logic;
begin

  -- Port Mappings
  lfsr0 : lfsr port map
    (
      enable  => acc_active,
      clk     => clk,
      reset   => reset,
      load    => load,
      seed    => seed,
      bit_out => sel
      );

  acc0 : accumulator port map
    (
      clk        => clk,
      acc_active => acc_active,
      sel => sel,
      reset => reset_acc,
      i_vec => i_vec,
      o_reg => o_reg
      );

end df;
```

C.3 Accumulator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;



--------------------------Accumulator-----------------------------
```

```
--Used for generating the result of matrix
--vector multiplications.  Matrix is composed of -1,1's
--so we can simplify the process to additions or subtractions.
--Inputs are the input vector coming from RAM and the next element
--of the matrix coming from the LFSR block.

------------------------------------------------------------------
entity accumulator is
  port (
    clk        : in  std_logic;
    acc_active : in  std_logic;
    sel        : in  std_logic;
    reset      : in  std_logic;
    i_vec      : in  std_logic_vector(15 downto 0);
    o_reg      : out std_logic_vector(31 downto 0)
    );
end accumulator;

architecture behavior of accumulator is
  signal i_reg   : signed(15 downto 0);
  signal acc_out : signed(31 downto 0);

begin
  process(clk, reset)
  begin
    if reset = '1' then
      acc_out       <= (others => '0');
    else
      if rising_edge(clk) then
        if (acc_active = '1') then
  --sel is the output bit from the LFSR
          if sel = '1' then
            acc_out <= acc_out + (i_reg);
          else
            acc_out <= acc_out - (i_reg);
          end if;
        else
          acc_out   <= acc_out;
        end if;
      end if;
    end if;
  end process;

  i_reg <= signed(i_vec);
  o_reg <= std_logic_vector(acc_out);
```

```
end behavior;
```

C.4 LFSR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;



-----------------------LFSR---------------------------

-- Linear feedback shift register for generating the
-- random A matrix. Output bits are used to select
-- the appropriate operation in the accumulator block.

------------------------------------------------------
entity lfsr is
  port(
    clk     : in  std_logic;
    reset   : in  std_logic;
    enable  : in  std_logic;
    load    : in  std_logic;
    seed    : in  std_logic_vector(31 downto 0);
    bit_out : out std_logic
    );
end entity lfsr;

architecture behavior of lfsr is

  signal reg : std_logic_vector(31 downto 0);
  signal bit : std_logic;

begin

  bit_out <= reg(0);

  process(clk, reset, enable)
  begin
    if (reset = '1') then
      reg                  <= (others => '0');
    elsif (rising_edge(clk)) then
      if (load = '1') then
        reg                <= seed;
```

```
      else
        if enable = '1' then
          -- taps: 32, 22, 2, 1 (using 1-indexing)
          -- shift and generate new bit
  reg(30 downto 0) <= reg(31 downto 1);
          reg(31)          <= reg(31) xor reg(30) xor reg(10) xor reg(0);
        else
          reg(31 downto 0) <= reg(31 downto 0);
        end if;
      end if;
    end if;
  end process;

end behavior;
```

C.5 Single-Port RAM

```
library ieee;
use ieee.std_logic_1164.all;


--------------Single-Port RAM----------------

--RAM is used for storing input and ouput
--vectors for the matrix-vector multiplication.
--As well as command and status registers for
--signaling the state of the controller.

--VHDL follows the syntax suggested by Altera
--for instantiating the altsyncram megafuntion.
-----------------------------------------------



entity single_port_ram is

generic
(
DATA_WIDTH : natural := 16;
ADDR_WIDTH : natural := 13
);

port
(
clk : in std_logic;
addr : in natural range 0 to 2**ADDR_WIDTH - 1;
```

```
data : in std_logic_vector((DATA_WIDTH-1) downto 0);
we : in std_logic := '1';
q : out std_logic_vector((DATA_WIDTH -1) downto 0)
);

end entity;

architecture rtl of single_port_ram is

-- Build a 2-D array type for the RAM
subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
type memory_t is array(addr'high downto 0) of word_t;

-- Declare the RAM signal.
signal ram : memory_t;

-- Register to hold the address
signal addr_reg : natural range 0 to addr'high;

begin

process(clk)
begin
if(rising_edge(clk)) then
if(we = '1') then
ram(addr) <= data;
end if;

-- Register the address for reading
addr_reg <= addr;
end if;
end process;

-- assign the value to output
q <= ram(addr_reg);

end rtl;
```

C.6 VGA Raster

```
-------------------------------------------------------------------------------
-- Simple VGA raster display
--
-- Modified version of lab 3 code
-------------------------------------------------------------------------------
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_vga_raster is

  port (

-- Avalon Bus Signals

    reset       : in  std_logic;
    clk         : in  std_logic;        -- Should be 25.125 MHz
    chipselect  : in  std_logic;
    write, read : in  std_logic;
    address     : in  std_logic_vector(17 downto 0);
    readdata    : out std_logic_vector(15 downto 0);
    writedata   : in  std_logic_vector(15 downto 0);
    byteenable  : in  std_logic_vector(1 downto 0);
    waitrequest : out std_logic;
-- SRAM Control Signals

    SRAM_DQ              : inout std_logic_vector(15 downto 0);
    SRAM_ADDR            : out   std_logic_vector(17 downto 0);
    SRAM_UB_N, SRAM_LB_N : out   std_logic;
    SRAM_WE_N, SRAM_CE_N : out   std_logic;
    SRAM_OE_N            : out   std_logic;

-- VGA control signals

    VGA_CLK,                           -- Clock
    VGA_HS,                            -- H_SYNC
    VGA_VS,                            -- V_SYNC
    VGA_BLANK,                         -- BLANK
    VGA_SYNC : out std_logic;          -- SYNC
    VGA_R,                             -- Red[9:0]
    VGA_G,                             -- Green[9:0]
    VGA_B    : out unsigned(9 downto 0)  -- Blue[9:0]
    );

end de2_vga_raster;

architecture rtl of de2_vga_raster is

  component sram_mux
    port (
      sel                 : in  std_logic;
      SRAM_DQ_IN_from_av1  : in  std_logic_vector(15 downto 0);
```

```
    SRAM_DQ_OUT_from_av1 : out std_logic_vector(15 downto 0);
    SRAM_ADDR_from_av1   : in  std_logic_vector(17 downto 0);
    SRAM_UB_N_from_av1   : in  std_logic;
    SRAM_LB_N_from_av1   : in  std_logic;
    SRAM_WE_N_from_av1   : in  std_logic;
    SRAM_CE_N_from_av1   : in  std_logic;
    SRAM_OE_N_from_av1   : in  std_logic;

    SRAM_DQ_IN_from_vga  : in  std_logic_vector(15 downto 0);
    SRAM_DQ_OUT_from_vga : out std_logic_vector(15 downto 0);
    SRAM_ADDR_from_vga   : in  std_logic_vector(17 downto 0);
    SRAM_UB_N_from_vga   : in  std_logic;
    SRAM_LB_N_from_vga   : in  std_logic;
    SRAM_WE_N_from_vga   : in  std_logic;
    SRAM_CE_N_from_vga   : in  std_logic;
    SRAM_OE_N_from_vga   : in  std_logic;

    SRAM_DQ   : inout std_logic_vector(15 downto 0);
    SRAM_ADDR : out   std_logic_vector(17 downto 0);
    SRAM_UB_N : out   std_logic;
    SRAM_LB_N : out   std_logic;
    SRAM_WE_N : out   std_logic;
    SRAM_CE_N : out   std_logic;
    SRAM_OE_N : out   std_logic
    );
end component;





-- Video parameters

constant HTOTAL       : integer := 800;
constant HSYNC        : integer := 96;
constant HBACK_PORCH  : integer := 48;
constant HACTIVE      : integer := 640;
constant HFRONT_PORCH : integer := 16;

constant VTOTAL       : integer := 525;
constant VSYNC        : integer := 2;
constant VBACK_PORCH  : integer := 33;
constant VACTIVE      : integer := 480;
constant VFRONT_PORCH : integer := 10;
```

```
  constant RECTANGLE_HSTART : integer := 100;
  constant RECTANGLE_HEND   : integer := 540;
  constant RECTANGLE_VSTART : integer := 100;
  constant RECTANGLE_VEND   : integer := 380;
  constant HCENTER          : integer := 320;
  constant Hleft            : integer := 160;
  constant Hright           : integer := 480;
  constant VCENTER          : integer := 240;
  constant Vtop             : integer := 120;
  constant Vbot             : integer := 360;

  -- Signals for the video controller
  signal Hcount               : unsigned(9 downto 0);  -- Horizontal position
(0-800)
  signal Vcount               : unsigned(9 downto 0);  -- Vertical position
(0-524)
  signal EndOfLine, EndOfField : std_logic;

  signal vga_hblank, vga_hsync,
    vga_vblank, vga_vsync : std_logic;  -- Sync. signals

  signal image_h, image_v, image_v_total, image_h_total, image_p : std_logic;
-- rectangle area
  signal clk25                                                    : std_logic :=
'0';

  -- Signals
  signal IMAGE_BASE : unsigned(17 downto 0) := "000000000000000000";  --stores
base address of image in SRAM
  signal SIZE       : unsigned(15 downto 0) := X"007F";  --size of image i.e.
200x200
  signal RED        : unsigned(9 downto 0)  := "0000000000";
  signal GREEN      : unsigned(9 downto 0)  := "0000000000";
  signal BLUE       : unsigned(9 downto 0)  := "0000000000";

  ------------------- Create Control Registers  -----------------
  -- cntl_regs( 0 to 2 ) = address of image in SRAM
  -- cntl_regs( 3 to 4 ) = image size i.e. 200x200 = 200
  -- cntl_regs( 5 ) = status register
  -- status register = invalid_W/valid_W|invalid_R/valid_R|0|0|0|0|0|GO/DONE|
  type regs_type is array(0 to 7) of std_logic_vector(7 downto 0);
  signal cntl_regs : regs_type;
  signal addr_cnt  : unsigned(17 downto 0) := "000000000000000000";
  signal CE_STATE  : std_logic             := '1';

  signal SRAM_DQ_IN_from_vga  : std_logic_vector(15 downto 0);
```

```
  signal SRAM_DQ_OUT_from_vga : std_logic_vector(15 downto 0);
  signal SRAM_ADDR_from_vga   : std_logic_vector(17 downto 0);
  signal SRAM_UB_N_from_vga   : std_logic;
  signal SRAM_LB_N_from_vga   : std_logic;
  signal SRAM_WE_N_from_vga   : std_logic;
  signal SRAM_CE_N_from_vga   : std_logic;
  signal SRAM_OE_N_from_vga   : std_logic;
  signal sel                  : std_logic;
begin

  -- static signals
  SRAM_OE_N_from_vga <= '0';
  SRAM_WE_N_from_vga <= '1';
  SRAM_UB_N_from_vga <= '0';
  SRAM_LB_N_from_vga <= '0';
  SRAM_CE_N_from_vga <= '0';

  -- VGA uses SRAM as FRAME BUFFER
  sram_mux0 : sram_mux port map (
    sel                  => sel,
    SRAM_DQ_IN_from_av1  => writedata,
    SRAM_DQ_OUT_from_av1 => readdata,
    SRAM_ADDR_from_av1   => address,
    SRAM_UB_N_from_av1   => not byteenable(1),
    SRAM_LB_N_from_av1   => not byteenable(0),
    SRAM_WE_N_from_av1   => not write,
    SRAM_CE_N_from_av1   => not chipselect,
    SRAM_OE_N_from_av1   => not read,

    SRAM_DQ_IN_from_vga  => SRAM_DQ_IN_from_vga,
    SRAM_DQ_OUT_from_vga => SRAM_DQ_OUT_from_vga,
    SRAM_ADDR_from_vga   => SRAM_ADDR_from_vga,
    SRAM_UB_N_from_vga   => SRAM_UB_N_from_vga,
    SRAM_LB_N_from_vga   => SRAM_LB_N_from_vga,
    SRAM_WE_N_from_vga   => SRAM_WE_N_from_vga,
    SRAM_CE_N_from_vga   => SRAM_CE_N_from_vga,
    SRAM_OE_N_from_vga   => SRAM_OE_N_from_vga,

    SRAM_DQ   => SRAM_DQ,
    SRAM_ADDR => SRAM_ADDR,
    SRAM_UB_N => SRAM_UB_N,
    SRAM_LB_N => SRAM_LB_N,
    SRAM_WE_N => SRAM_WE_N,
    SRAM_CE_N => SRAM_CE_N,
    SRAM_OE_N => SRAM_OE_N
    );
```

```
process (clk)
begin
  if rising_edge(clk) then
    clk25 <= not clk25;
  end if;
end process;

-- Horizontal and vertical counters

HCounter : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      Hcount <= (others => '0');
    elsif EndOfLine = '1' then
      Hcount <= (others => '0');
    else
      Hcount <= Hcount + 1;
    end if;
  end if;
end process HCounter;

EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

VCounter : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      Vcount   <= (others => '0');
    elsif EndOfLine = '1' then
      if EndOfField = '1' then
        Vcount <= (others => '0');
      else
        Vcount <= Vcount + 1;
      end if;
    end if;
  end if;
end process VCounter;

EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK
```

```
HSyncGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' or EndOfLine = '1' then
      vga_hsync <= '1';
    elsif Hcount = HSYNC - 1 then
      vga_hsync <= '0';
    end if;
  end if;
end process HSyncGen;

HBlankGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      vga_hblank <= '1';
    elsif Hcount = HSYNC + HBACK_PORCH then
      vga_hblank <= '0';
    elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
      vga_hblank <= '1';
    end if;
  end if;
end process HBlankGen;

VSyncGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      vga_vsync   <= '1';
    elsif EndOfLine = '1' then
      if EndOfField = '1' then
        vga_vsync <= '1';
      elsif Vcount = VSYNC - 1 then
        vga_vsync <= '0';
      end if;
    end if;
  end if;
end process VSyncGen;

VBlankGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      vga_vblank   <= '1';
    elsif EndOfLine = '1' then
      if Vcount = VSYNC + VBACK_PORCH - 1 then
```

```
        vga_vblank <= '0';
      elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
        vga_vblank <= '1';
      end if;
    end if;
  end if;
end process VBlankGen;

-- Rectangle generator

-- Vertical edges of the four squares that hold images
PosHGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' or Hcount = HSYNC + HBACK_PORCH + Hleft - (SIZE srl 1) - 1
then
      image_h_total <= '1';
      image_h       <= '1';
    elsif Hcount = HSYNC + HBACK_PORCH + Hleft + (SIZE srl 1) + 1 then
      image_h       <= '0';
    elsif Hcount = HSYNC + HBACK_PORCH + Hright - (SIZE srl 1) - 1 then
      image_h       <= '1';
    elsif Hcount = HSYNC + HBACK_PORCH + Hright + (SIZE srl 1) + 1 then
      image_h       <= '0';
      image_h_total <= '0';
    end if;
  end if;
end process PosHGen;

-- Horizontal edges
PosVGen : process (clk25)
begin
  if rising_edge(clk25) then
    if reset = '1' then
      image_v         <= '0';
      image_v_total   <= '0';
    elsif EndOfLine = '1' then
      if Vcount = (VSYNC + VBACK_PORCH - 1) + Vtop - (SIZE srl 1) - 1 then
        image_v_total <= '1';
        image_v       <= '1';
      elsif Vcount = (VSYNC + VBACK_PORCH - 1) + Vtop + (SIZE srl 1) + 1 then
        image_v       <= '0';
      elsif Vcount = (VSYNC + VBACK_PORCH - 1) + Vbot - (SIZE srl 1) - 1 then
        image_v       <= '1';
      elsif Vcount = (VSYNC + VBACK_PORCH - 1) + Vbot + (SIZE srl 1) + 1 then
        image_v_total <= '0';
```

```
        image_v       <= '0';
      end if;

    end if;
  end if;
end process PosVGen;

image_p     <= image_h and image_v;
sel         <= image_p;
waitrequest <= image_p;
-- Get data from SRAM

PixelData : process ( clk25, reset )
begin
  if reset = '1' then
    RED                   <= "0000000000";
    GREEN                 <= "0000000000";
    BLUE                  <= "0000000000";
  else
    if rising_edge(clk25) then
      if (image_v_total or image_h_total) = '0' then
        addr_cnt          <= "000000000000000000";
      elsif image_p = '1' then
-- retrieve the pixels from SRAM
        SRAM_ADDR_from_vga <= std_logic_vector(IMAGE_BASE + addr_cnt);
        RED               <= unsigned("1111111111" and
SRAM_DQ_OUT_from_vga(14 downto 10)) sll 5;
        GREEN             <= unsigned("1111111111" and SRAM_DQ_OUT_from_vga(9
downto 5)) sll 5;
        BLUE              <= unsigned("1111111111" and SRAM_DQ_OUT_from_vga(4
downto 0)) sll 5;
        addr_cnt          <= addr_cnt + 1;
      else
        addr_cnt          <= addr_cnt;
      end if;
    end if;
  end if;
end process PixelData;

-- Registered video signals going to the video DAC

VideoOut : process (clk25, reset)
begin
  if reset = '1' then
    VGA_R   <= "0000000000";
    VGA_G   <= "0000000000";
```

```
      VGA_B   <= "0000000000";
    elsif clk25'event and clk25 = '1' then
      if image_p = '1' then
        VGA_R <= RED;
        VGA_G <= GREEN;
        VGA_B <= BLUE;
      elsif vga_hblank = '0' and vga_vblank = '0' then
        VGA_R <= "0000000000";
        VGA_G <= "1111111111";
        VGA_B <= "0000000000";
      else
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      end if;
    end if;
  end process VideoOut;

  VGA_CLK <= clk25;
  VGA_HS  <= not vga_hsync;
  VGA_VS <= not vga_vsync;
  VGA_SYNC <= '0';
  VGA_BLANK <= not (vga_hsync or vga_vsync);

end rtl;
```

References

[1] RG Baraniuk. Compressive sensing [lecture notes]. *IEEE Signal Processing Magazine*, 24(4):118–121, 2007.

[2] YC Pati, R. Rezaiifar, and PS Krishnaprasad. Orthogonal matching pursuit: recursive function approximation withapplications to wavelet decomposition. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on*, pages 40–44, 1993.

[3] Stanford SparseLab: Seeking Sparse Solutions to Linear Systems of Equations. `http://sparselab.stanford.edu`.

[4] Ian Kaplan. The Daubechies D4 Wavelet Transform. `http://www.bearcave.com/misl/misl_tech/wavelets/daubechies/index.html`, January 2002.