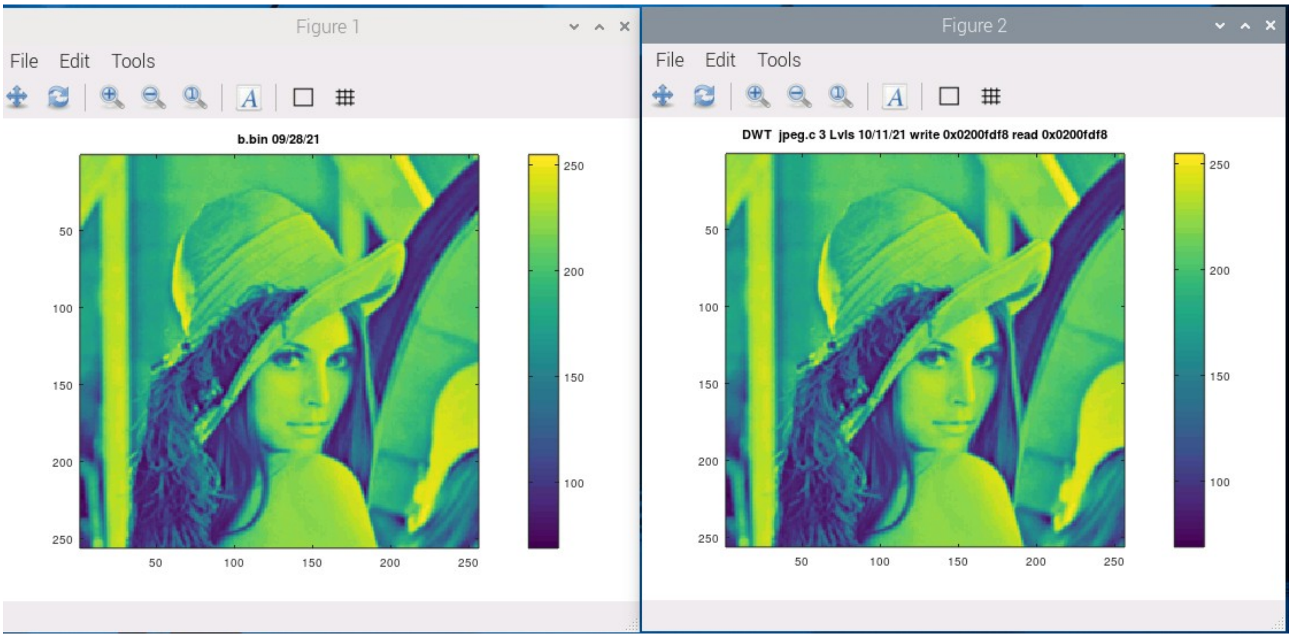
**********************Draft**********************

Creating a New PMOD Ethernet for pico-ice & catboard Starting with lifting step of Images Using 8X8 Blocks

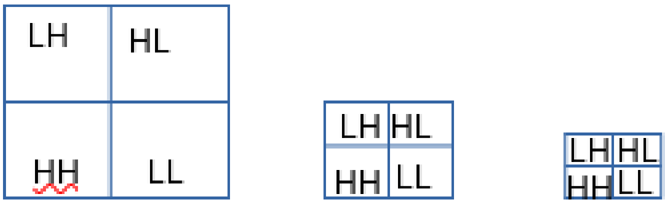
03/14/25

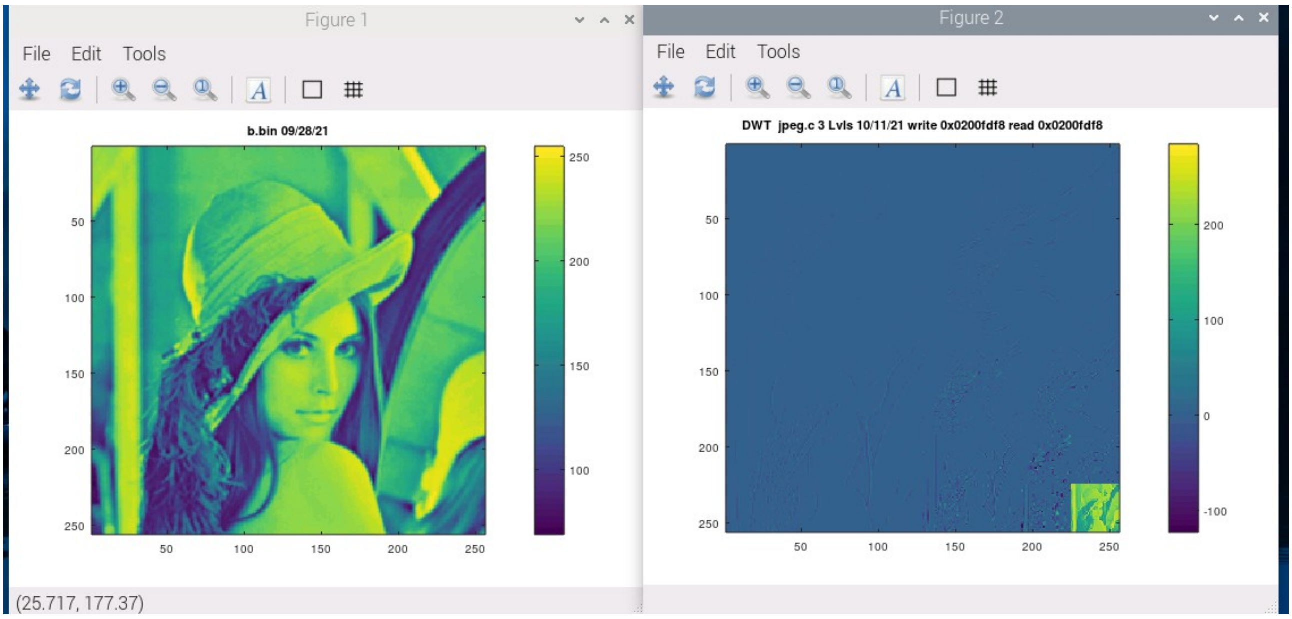
**********************Draft**********************

One question that needs to be answered is what is the optimal block size?
The first step is divide the image in

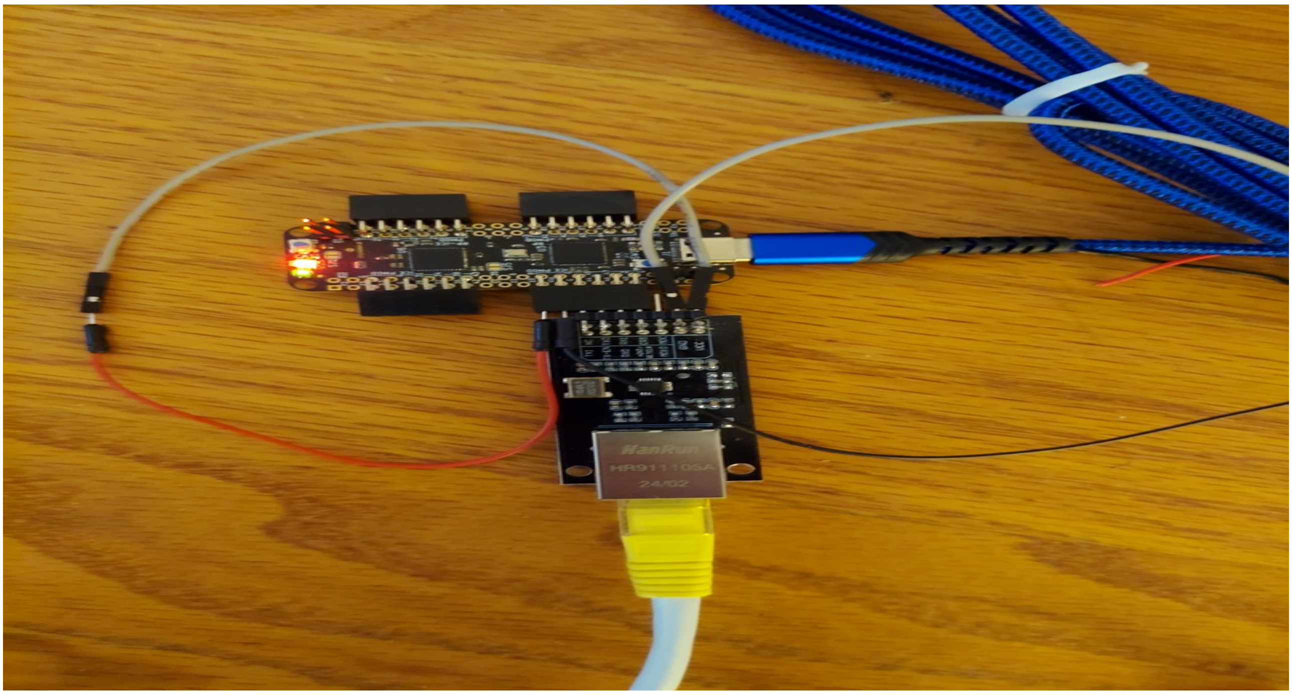


The default should be 3 lvls of decomposition





The pico-ice could have a 2nd pmod-ethernet this would require 2 different MAC address.



Current pi_jpeg.c Works on an entire image 256x256 RGB

```
devel@pi5-90:~/102121icozip/sw/board $ ./buildpi_lift.sh
#/bin/bash
rm -f ../host/pi_jpeg lifting.o pi_jpeg.o
gcc -g -c lifting.c -o lifting.o
gcc -g -c pi_jpeg.c -o pi_jpeg.o
gcc -g pi_jpeg.o lifting.o -o ../host/pi_jpeg
```

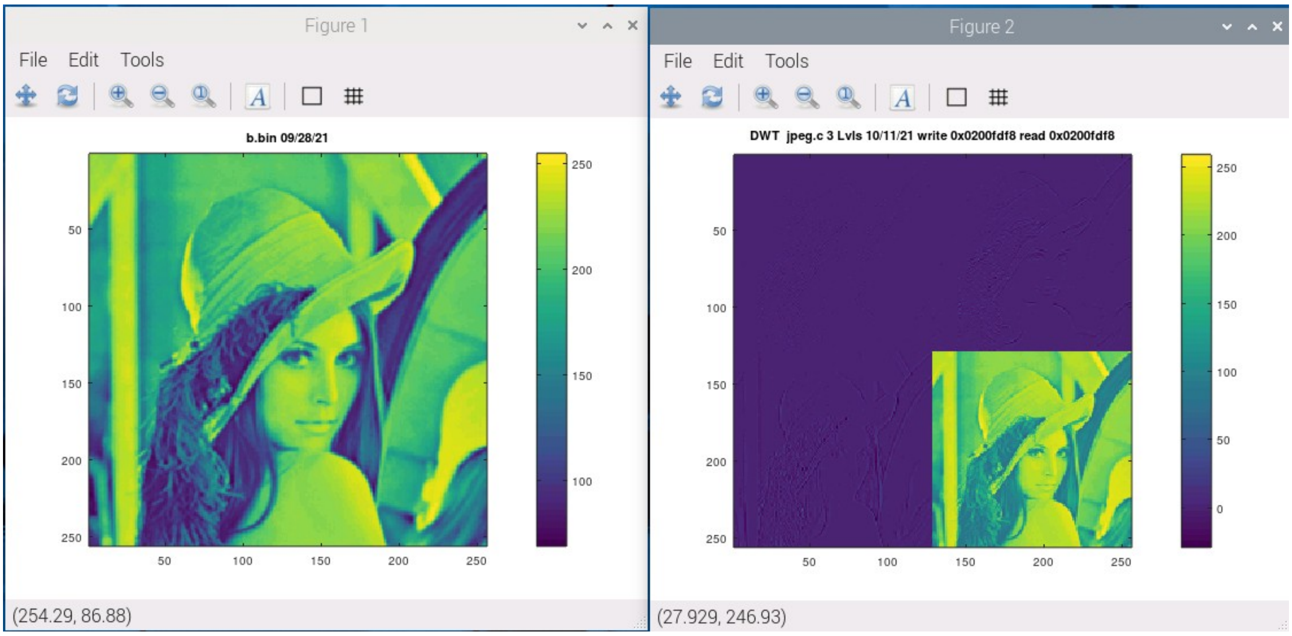
line 183 const int   LVLS = 1; controls the level of decomposition.

```
devel@pi5-90:~/102121icozip/sw/host $ ./pi_jpeg 0 1 Red
devel@pi5-90:~/102121icozip/sw/host $ ./pi_jpeg 1 1 Green
devel@pi5-90:~/102121icozip/sw/host $ ./pi_jpeg 2 1 Blue
```

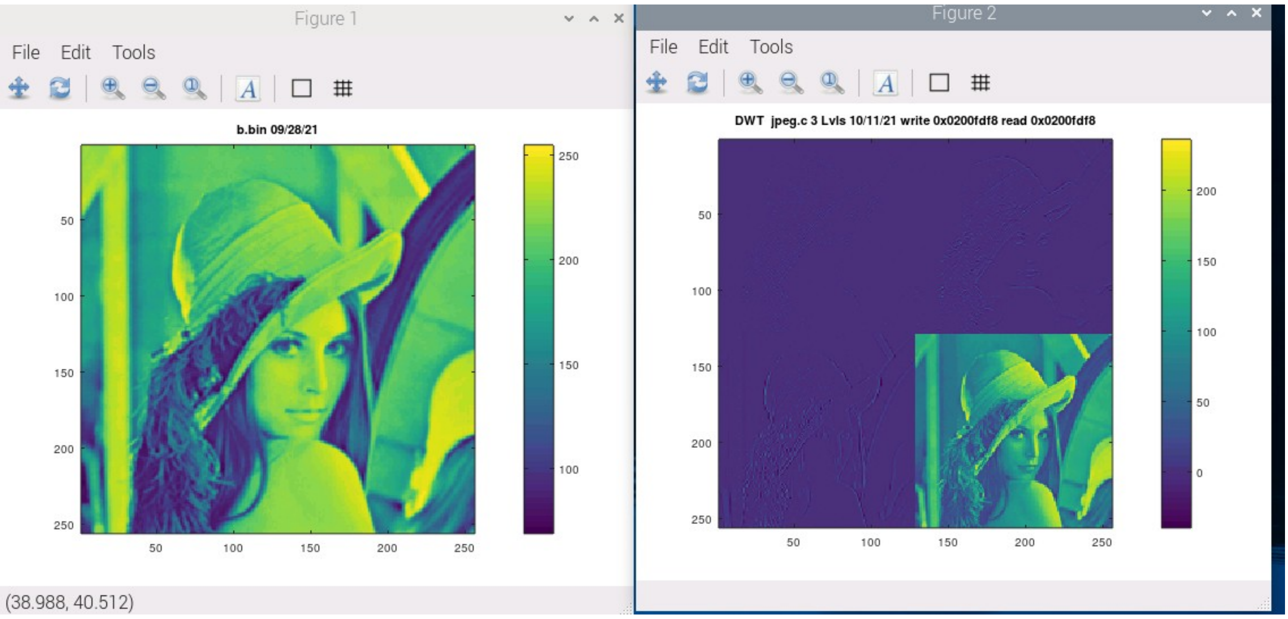
https://github.com/develone/102121icozip/tree/catzip/sw/board
rgb.m creates the figures below.

The C pi_jpeg.c found Appendix A. **Appendix B. lifting.c**

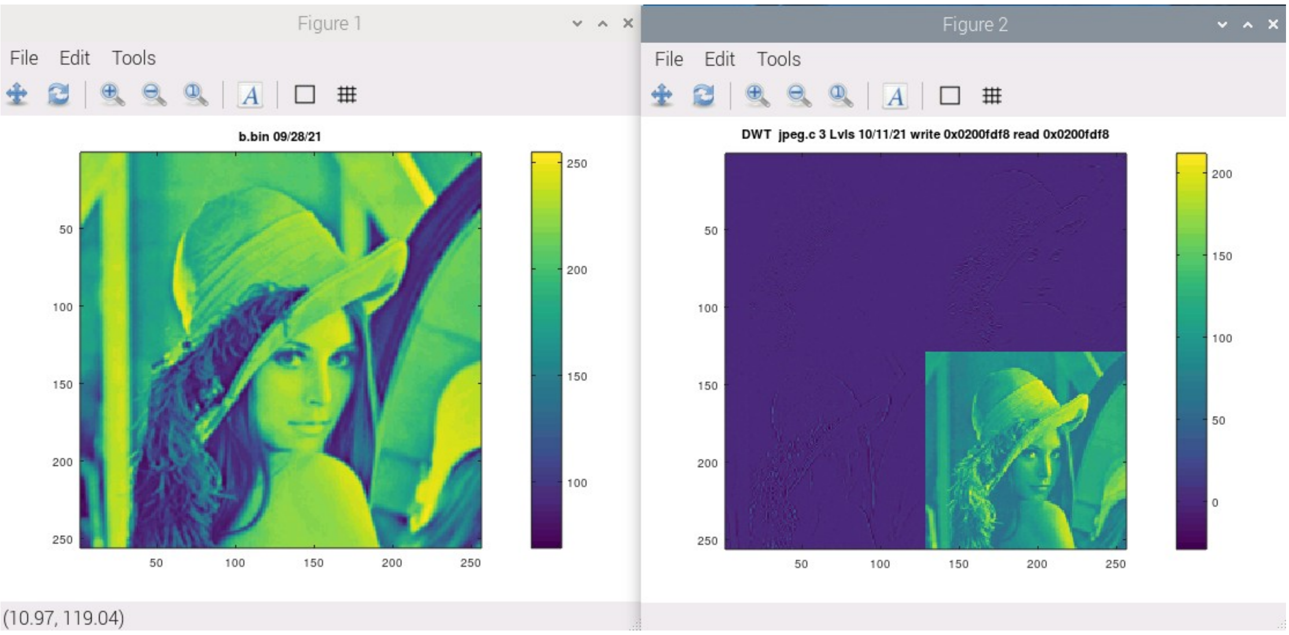
Red



Green



Blu



At a glance it didn't seem like a lot of VHDL inside jpeg.vhd?
Is that all of the code from myHDL or is there more?

Very likely can rewrite that in pipelinec in a few minutes

```vhdl
if bool(lo_hi_s) then
    if bool(fwd_inv_s) then
        res_s <= (sam_s - (shift_right(left_s, 1) + shift_right(right_s, 1)));
    else
        res_s <= (sam_s + (shift_right(left_s, 1) + shift_right(right_s, 1)));
    end if;
else
    if bool(fwd_inv_s) then
        res_s <= (sam_s + shift_right(((left_s + right_s) + 2), 2));
    else
        res_s <= (sam_s - shift_right(((left_s + right_s) + 2), 2));
    end if;
end if;
```

What you would want to do is define an input struct that rounds things to byte sized types for sharing data with software (notice lo_hi flag is u8, etc)

```
work_inputs_t
    int16_t left,
    int16_t right,
    int16_t sam,
    uint8_t lo_hi,
    uint8_t fwd_inv

work_outputs_t
   int16_t res;
```

work_outputs_t work(work_inputs_t) and fill in work() like JPEG_HDL from above

Appendix A pi_jpeg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lifting.h"
/* First parameter is used to tell the program which sub band to use
 * 0 Red
 * 1 Green
 * 2 Blue
 * 2nd parameter is used to tell the program to compute the fwd lifting step only or fwd lifting then inv lifting step
 * 0 fwd lifting then inv lifting step
 * 1 fwd lifting step only
 * ./pi_jpeg 0 1 or ./pi_jpeg 0 0
 * ./pi_jpeg 1 1 or ./pi_jpeg 1 0
 * ./pi_jpeg 2 1 or ./pi_jpeg 2 0
 */
struct PTRs {
    int inpbuf[65536];

    int flag;
    int w;
    int h;

     int *red;

     int *grn;
     int *blu;
     int *alt;
    //int *fwd_inv;
} ptrs;


int main(int argc, char **argv) {

    FILE *inptr,*outptr;
      char *ch;
      int tmp,loop;

     int *red_s_ptr, *gr_s_ptr, *bl_s_ptr;
     int *wptr,*wptr1,*wptr2;
     int *alt,*alt1,*alt2;


     int *buf_red, *buf_gr, *buf_bl;
     int ur,ug,ub,x,y,z;
    int *fwd_inv;

    int i,j;
```

```
        ptrs.w = 256;
        ptrs.h = 256;

        buf_red = ( int *)malloc(sizeof( int)* ptrs.w*ptrs.h*2);
        red_s_ptr = buf_red;

        fwd_inv = (int *)malloc(1);

        if(buf_red == NULL) return 2;

        if(fwd_inv == NULL) return 5;
        red_s_ptr = buf_red;
printf("buf_red = 0x%x\n",buf_red);

printf("fwd_inv = 0x%x\n",fwd_inv);
/*The file rgb_pack.bin contains the rgb images
 * packed in bits red 29-20
 * packed in bits grn 19-10
 * packed in bits blu 9-0
*/

        loop = 65536;
        for(i=0;i<loop;i++) buf_red[i]=ptrs.inpbuf[i];
                ch = argv[1];
                tmp = atoi(ch);
                if (tmp == 0) {
                        printf("spliting red sub band\n");
                        ptrs.flag = tmp;
                        inptr = fopen("r.bin","rb");
                        if (!inptr)
                        {
                                printf("Unle to open file!");
                                return 1;
                        }
                        else fread(ptrs.inpbuf,sizeof(int),65536,inptr);

                        fclose(inptr);

                }
                else if (tmp == 1) {
                        printf("spliting green sub band\n");
                        ptrs.flag = tmp;
                                                inptr = fopen("g.bin","rb");
                        if (!inptr)
                        {
                                printf("Unle to open file!");
                                return 1;
                        }
                        else fread(ptrs.inpbuf,sizeof(int),65536,inptr);

                        fclose(inptr);
```

```
          }
          else if (tmp == 2) {
                    printf("spliting blue sub band\n");
                    ptrs.flag = tmp;
                                        inptr = fopen("b.bin","rb");
                    if (!inptr)
                    {
                              printf("Unle to open file!");
                              return 1;
                    }
                    else fread(ptrs.inpbuf,sizeof(int),65536,inptr);

                    fclose(inptr);

          }
          else {
                    printf("First parameter can only be 0 1 2 \n");
                    free(buf_red);
        free(fwd_inv);
                    exit (1);
          }
          for(i=0;i<loop;i++) buf_red[i]=ptrs.inpbuf[i];
          ch = argv[2];
          tmp = atoi(ch);

          if (tmp == 0) {
                    printf("fwd lifting then inv lifting step\n");
                    *fwd_inv = tmp;
          }
          else if (tmp == 1) {
                    printf("fwd lifting step only\n");
                    *fwd_inv = tmp;
          } else
          {
                    printf("2nd parameter can only be 0 1  \n");
                    free(buf_red);
        free(fwd_inv);
                    exit (2);
          }

          buf_red = red_s_ptr;
          wptr = buf_red;
          alt = &buf_red[ptrs.w*ptrs.h];
          printf("w = 0x%x buf_red wptr = 0x%x alt =  0x%x fwd_inverse =  0x%x
fwd_inverse =  0x%x \n",ptrs.w, wptr,alt,fwd_inv,*fwd_inv);
          printf("starting red dwt\n");

          lifting(ptrs.w,wptr,alt,fwd_inv);
          printf("finished ted dwt\n");
          //pack(ptrs.flag, i,buf_red, ptrs.inpbuf);
```

```
    outptr = fopen("dwt.bin","wb");
    if (!outptr)
            {
            printf("Unle to open file!");
            return 1;
            }
            fwrite(buf_red,sizeof( int),65536,outptr);
            //fwrite(alt,sizeof( int),65536,outptr);
            fclose(outptr);

            free(buf_red);
            free(fwd_inv);

            return 0;

}
```

Appendix B. lifting.c

```
////////////////////////////////////////////////////////////////////////////////
//
// Filename:    lifting.c
//
// Project:     XuLA2-LX25 SoC based upon the ZipCPU
//
// Purpose:     This goal of this file is to perform, on either the ZipCPU or
//              a more traditional architecture, the lifting/WVT step of the
//      JPEG-2000 compression (and decompression) scheme.
//
//      Currently, the lifting scheme performs both forward and inverse
//      transforms, and so (if done properly) it constitutes an identity
//      transformation.
//
// Creator:     Dan Gisselquist, Ph.D.
//              Gisselquist Technology, LLC
//
////////////////////////////////////////////////////////////////////////////////
//
// Copyright (C) 2015-2016, Gisselquist Technology, LLC
//
// This program is free software (firmware): you can redistribute it and/or
// modify it under the terms of  the GNU General Public License as published
// by the Free Software Foundation, either version 3 of the License, or (at
// your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT
// ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or
// FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along
// with this program.  (It's in the $(ROOT)/doc directory, run make with no
// target there if the PDF file isn't present.)  If not, see
// <http://www.gnu.org/licenses/> for a copy.
//
// License:     GPL, v3, as defined and found on www.gnu.org,
//              http://www.gnu.org/licenses/gpl.html
//
//
////////////////////////////////////////////////////////////////////////////////
//
//
#include "lifting.h"
#include <stdio.h>
void    singlelift(int rb, int w, int * const ibuf, int * const obuf) {
        int     col, row;
        //printf("in singlelift \n");
        for(row=0; row<w; row++) {
                register int    *ip, *op, *opb;
                register int    ap,b,cp,d;

                //
                // Ibuf walks down rows (here), but reads across columns (below)
                // We might manage to get rid of the multiply by doing something
                // like:
                //      ip = ip + (rb-w);
                // but we'll keep it this way for now.
                //
                //setting to beginning of each row
                ip = ibuf+row*rb;

                //
                // Obuf walks across columns (here), writing down rows (below)
                //
                // Here again, we might be able to get rid of the multiply,
                // but let's get some confidence as written first.
                //
                op = obuf+row;
                opb = op + w*rb/2;
                printf("ip = 0x%x op = 0x%x opb = 0x%x\n",ip,op,opb);
                //
                // Pre-charge our pipeline
                //

                // a,b,c,d,e ...
                // Evens get updated first, via the highpass filter
                ap = ip[0];
                b  = ip[1];
                cp = ip[2];
                d  = ip[3]; ip += 4;
                printf("ap = %d b = %d cp = %d d = %d\n",ap,b,cp,d);
                //
                ap = ap-b; // img[0]-(img[1]+img[-1])>>1)
                cp = cp- ((b+d)>>1);
                op[0] = ap;
                opb[0]  = b+((ap+cp+2)>>2);

                for(col=1; col<w/2-1; col++) {
                        op +=rb; // = obuf+row+rb*col = obuf[col][row]
                        opb+=rb;// = obuf+row+rb*(col+w/2) = obuf[col+w/2][row]
                        ap = cp;
                        b  = d;
                        cp = ip[0];     // = ip[row][2*col+1]
                        d  = ip[1];     // = ip[row][2*col+2]

                        //HP filter in fwd dwt
                        cp  = (cp-((b+d)>>1)); //op[0] is obuf[col][row]
                        *op = ap; //op[0] is obuf[col][row]

                        //LP filter in fwd dwt
                        *opb = b+((ap+cp+2)>>2);
                        ip+=2; // = ibuf + (row*rb)+2*col
                }

                op += rb; opb += rb;
                *op  = cp;
                *opb = d+((cp+1)>>3);
        }
}

void    ilift(int rb, int w,  int * const ibuf,  int * const obuf) {
        int     col, row;

        for(row=0; row<w; row++) {
                register int    *ip, *ipb, *op;
                register int    b,c,d,e;

                //
                // Ibuf walks down rows (here), but reads across columns (below)
                // We might manage to get rid of the multiply by doing something
                // like:
                //      ip = ip + (rb-w);
                // but we'll keep it this way for now.
                //
                //setting to beginning of each row
                op = obuf+row*rb;

                //
                // Obuf walks across columns (here), writing down rows (below)
                //
                // Here again, we might be able to get rid of the multiply,
                // but let's get some confidence as written first.
                //
                ip  = ibuf+row;
                ipb = ip + w*rb/2;
                printf("ip = 0x%x op = 0x%x ipb = 0x%x\n",ip,op,ipb);
                //
                // Pre-charge our pipeline
                //
*****************************************
                // a,b,c,d,e ...
                // Evens get updated first, via the highpass filter
                c = ip[0]; // would've been called 'a'
                ip += rb;
                e = ip[0];              // Would've been called 'c'
                d  = ipb[0] -((c+e+2)>>2);

                op[0] = c+d;    // Here's the mirror, left-side
                op[1] = d;
                printf("c = %d e = %d d = %d c+d = %d\n",c,e,c,c+d);
                for(col=1; col<w/2-1; col++) {
                        op += 2;
                        ip += rb; ipb += rb;

                        c = e; b = d;
                        e = ip[0];
                        d = ipb[0] - ((c+e+2)>>2);
                        c = c + ((b+d)>>1);

                        op[0] = c;
                        op[1] = d;
                }

                ipb += rb;
                d = ipb[0] - ((e+1)>>3);

                c = e + ((b+d)>>1);
                op[2] = c;
                op[3] = d;              // Mirror
        }
}

void    lifting(int w, int *ibuf, int *tmpbuf, int *fwd) {
        const   int     rb = w;
        int     lvl;

        int     *ip = ibuf, *tp = tmpbuf, *test_fwd = fwd;
        printf("ip = 0x%x tp = 0x%x \n",ip,tp);
        int     ov[3];

        const int       LVLS = 1;

/*
        for(lvl=0; lvl<w*w; lvl++)
                ibuf[lvl] = 0;
        for(lvl=0; lvl<w*w; lvl++)
                tmpbuf[lvl] = 5000;
```

```
	for(lvl=0; lvl<w; lvl++)
		ibuf[lvl*(rb+1)] = 20;

	singlelift(rb,w,ip,tp);
	for(lvl=0; lvl<w*w; lvl++)
		ibuf[lvl] = tmpbuf[lvl];

	return;
*/

	for(lvl=0; lvl<LVLS; lvl++) {
		// Process columns -- leave result in tmpbuf
		//printf("in lifting \n");
		singlelift(rb, w, ip, tp);
		// Process columns, what used to be the rows from the last
		// round, pulling the data from tmpbuf and moving it back
		// to ibuf.
		printf("w = 0x%x ip = 0x%x tp = 0x%x \n",w,ip,tp);
		singlelift(rb, w, tp, ip);
		//printf("back from singlelift\n");
		// lower_upper
		//
		// For this, we just adjust our pointer(s) so that the "image"
		// we are processing, as referenced by our two pointers, now
		// references the bottom right part of the image.
		//
		// Of course, we haven't really changed the dimensions of the
		// image.  It started out rb * rb in size, or the initial w*w,
		// we're just changing where our pointer into the image is.
		// Rows remain rb long.  We pretend (above) that this new image
		// is w*w, or should I say (w/2)*(w/2), but really we're just
		// picking a new starting coordinate and it remains rb*rb.
		//
		// Still, this makes a subimage, within our image, containing
		// the low order results of our processing.
		int	offset = w*rb/2+w/2;
		ip = &ip[offset];
		tp = &tp[offset];
		ov[lvl] = offset + ((lvl)?(ov[lvl-1]):0);

		// Move to the corner, and repeat
		w>>=1;
	}
	//printf("testing test_fwd \n");
	if (test_fwd[0]==0) {
	for(lvl=(LVLS-1); lvl>=0; lvl--) {
		int	offset;

		w <<= 1;

		if (lvl)
```

```
				offset = ov[lvl-1];
			else
				offset = 0;
			ip = &ibuf[offset];
			tp = &tmpbuf[offset];
			printf("ip = 0x%x tp = 0x%x \n",ip,tp);

			ilift(rb, w, ip, tp);
			//printf("back from ilift\n");
			ilift(rb, w, tp, ip);
			//printf("back from ilift\n");
		}
	}
}
```