Compressing and Decompressing Sequences of Images

12

Besides enabling you to perform image-processing operations on XIL images, the XIL library also gives you the ability to compress one or more XIL images and to store the compressed images in a data structure called a compressed image sequence (CIS). The XIL images you compress may represent frames of video from a movie, pages from a document, or any number of things. You can also decompress the data stored in a CIS and write your output to one or more XIL images. This compression-decompression process is shown in Figure 12-1.

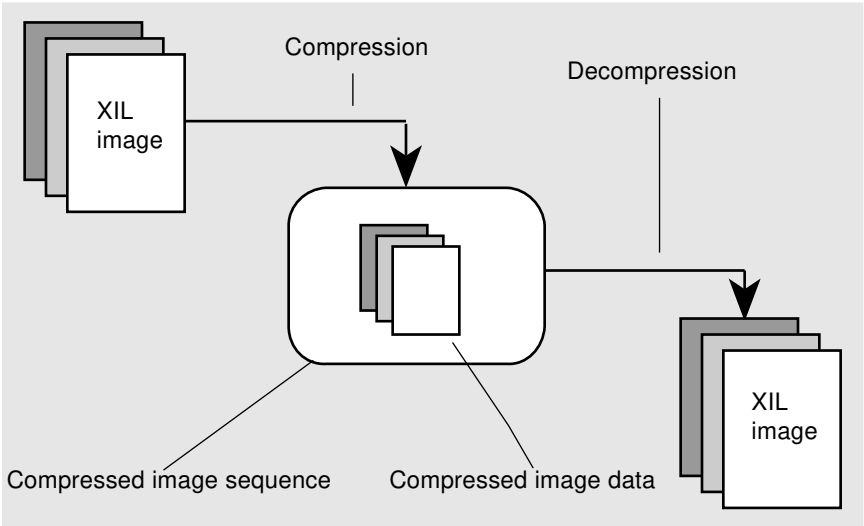

Figure 12-1 Compressing and Decompressing XIL Images

12

As you probably know, uncompressed images can take up a lot of disk space and take a long time to move over the network. For example, a 512-by-512 3-band image containing `XIL_BYTE` data takes up .75 Mbytes. A compressed version of this image might be anywhere from one-half to one-fiftieth this size, depending on the type of compressor used.

The XIL library contains several compression and decompression modules:

- Cell and CellB codecs
- JPEG baseline sequential and lossless codecs
- An H.261 decompressor
- An MPEG-1 decompressor
- CCITT Group 3 and Group 4 codecs

The Cell image compression technology, which was developed by Sun, has been optimized for the rapid decompression and display of images on simple hardware. Therefore, the Cell codec is able to achieve reasonable display quality on indexed-color frame buffers. The initial focus of the Cell technology is on Sun-to-Sun communications, where the benefits of fast decoding outweigh the benefits of standards. Possible areas of application include media distributions on CD-ROM and multimedia mail.

The CellB codec, which derives from its Cell counterpart, is intended for use primarily in videoconferencing applications. It features greater balance between the time spent compressing and decompressing images than the Cell codec. The CellB codec's strengths include

- Software compression at interactive rates
- Very fast decoding and display, especially on indexed-color frame buffers
- Low rates of CPU use
- Good quality output

The JPEG compression standards were developed by the Joint Photographic Experts Group to support the compression of still images, both grayscale and color. Although not specifically designed for the compression of sequences of images, or movies, JPEG compressors are also used frequently for that purpose. A *lossy* JPEG compressor compresses an image in such a way that when the compressed data is decompressed, the decompressed image and the original image may not match exactly. On the other hand, a *lossless* JPEG compressor processes an image so that the decompressed image does match the original image pixel for pixel.

The H.261 compression-decompression scheme was developed by the International Telegraph and Telephone Consultative Committee (CCITT). The H.261 video encoder is intended to be used to compress video data that will be sent over Integrated Services Digital Network (ISDN) lines. The H.261 codec is intended primarily for use in video telephony and videoconferencing applications.

The MPEG-1 video compression standard was developed by the Moving Picture Experts Group. The group's goal was to compress full-motion video and the associated audio at the rate of about 1.5 Mbits/s. This is approximately the rate at which data can be read from a CD-ROM, so MPEG-1 compressed video is a good choice for use in interactive multimedia applications.

The CCITT Group 3 and Group 4 compression standards were developed by the International Telegraph and Telephone Consultative Committee to enable facsimile machines to compress and decompress digitized documents. Now, Group 3 and Group 4 compressors and decompressors are used for general document storage and retrieval.

The remainder of this chapter introduces you to the subject of XIL compression and decompression by examining two example programs: `encode` and `xilcis_example`. The first program builds a Cell, CellB, or JPEG movie. It reads a series of video frames from disk files, compresses each frame, and writes the compressed image data to a CIS. As frames are compressed, the program writes the compressed data in the CIS to an output file. This output file can serve as input to the second example program, a movie player.

The second example program plays back a Cell, CellB, JPEG, H.261, or MPEG-1 datastream on an 8- or 24-bit frame buffer. It reads a datastream from a disk file and stores the compressed data in a CIS. Then, for each compressed image in the CIS, it takes the following actions. First, it decompresses a frame of video from the CIS and stores the resulting image in a 3-band `XIL_BYTE` image. Then, it prepares each image for display on a particular frame buffer and writes the image to a display image. These steps result in a frame of video being displayed.

12

Creating a JPEG Datastream

The source files for the movie-making example—`encode.c`, `load_file.c`, and `image_transform.c`—can be found in the directory `$XILHOME/examples/movie_maker_example`. This directory also contains the image files that the program should use as input and a makefile you can use to build the program.

After briefly discussing how to build and run the example, this section walks you through the program's code to clarify how the program produces a JPEG datastream. (Cell and CellB bytestreams are produced in a very similar way.) The basic algorithm is shown below:

1. Create a CIS.
2. While all the video frames have not been compressed, perform the following steps:
   a. Load data from an image file into an XIL image.
   b. Convert the format of the XIL image if that's necessary or desirable. It will be necessary if the image does not contain `XIL_BYTE` data because the JPEG compressor expects input of that type. If your image is not a YC_bC_r image, it is desirable to convert the image to that color space because YC_bC_r images are optimized for compression and decompression. The JPEG compressor will work with images of any color space, but it typically produces superior compression with YC_bC_r images.
   c. Compress the image and write the compressed data to the CIS.
   d. Destroy the image in order to free the memory allocated to hold the image structure and the image data.
   e. Write any compressed data in the CIS to a disk file.
3. Perform any outstanding compression operations, and write any resulting compressed data to the output file.

Building and Running the Example

To build and run the example `encode`, perform the following steps:

1. **Change your working directory to `$XILHOME/examples/movie_maker_example` and type `make`. The makefile in that directory will build the program.**

2. **While in the same directory, execute the program using the following syntax:**

   `% encode [-c | -cb] file-list [output-file]`

   By default, the program creates a JPEG bitstream. To produce a Cell bytestream instead, use the `-c` option, and to produce a CellB bytestream, use the `-cb` option.

   The argument *file-list* must be the name of a file that contains a list of files (images) to be processed. The list of files supplied with this example is called `mifkin.list`.

   The argument *output-file* is optional. If you supply a file name here, the example will write its output to a file of that name; otherwise, it will write its output to `out.jpg`, `out.cell`, or `out.cellb`, depending on which compressor you've used.

Creating a CIS

Before a program can compress an image or sequence of images, it must create a CIS in which to store compressed data. The example program creates this data structure using code similar to this.

```
XilSystemState state;
XilCis cis;

cis = xil_cis_create(state, "Jpeg");
xil_cis_set_attribute(cis, "ENCODE_411_INTERLEAVED",
    (void *)TRUE);
xil_cis_set_max_frames(cis, 100);
```

You use the function `xil_cis_create()` to create the CIS, which is a data structure of type `XilCis`. The two arguments to this function are a system-state structure (this was returned by an earlier call to `xil_open()`) and the name of a compressor/decompressor. The example requests the JPEG baseline sequential codec (`Jpeg`), but could have requested another compressor/decompressor. (For information about the strings you use to create CISs associated with other compressors/decompressors, see the section "Creating and Destroying a CIS" on page 250.)

The next line sets a JPEG-specific attribute called `ENCODE_411_INTERLEAVED` to `TRUE`. If the images being compressed are 3-band YC_bC_r images, setting this attribute causes the codec to subsample the data in the color bands so that only one color value will be encoded for each four color values in the original (4:1:1). This subsampling enables the compressor to achieve a much higher level of compression than it could otherwise.

The last line sets a CIS attribute that limits the number of frames the CIS will buffer. In this case, the CIS will maintain 100 frames in its buffer. If more than 100 frames are compressed, the oldest frames will become inaccessible. This attribute is set to keep the program from using too much memory in instances when there are many frames to be compressed.

CIS attributes are discussed further in the section "General CIS Attributes" on page 257.

Compressing Video Frames and Writing Compressed Data to a File

After creating a CIS, the example begins compressing frames of video. For each frame, the program performs the steps described in the sections below.

Loading Data from a File into an XIL Image

The example calls a routine called `load_file()` to read an image from a disk file into an XIL image. For a complete explanation of what is happening in this subroutine, see the section "Acquiring an Input Image" on page 13. The only thing new in this example is that `load_file()` reads a color space from the

image file header and, after creating the XIL image into which it will read pixel values, sets that XIL image's color-space attribute appropriately. See the code fragment below.

```
FILE *in_file;
char colorspace[80];
XilSystemState state;
unsigned int width, height, nbands;
XilDataType datatype;
XilImage image;

fscanf(in_file, "%s", colorspace);
image = xil_create(state, width, height, nbands, datatype);
xil_set_colorspace(image, xil_colorspace_get_by_name(state,
    colorspace));
```

The reason for checking the color space of the input image is that the JPEG compressor is optimized for compressing YC_bC_r images. The images supplied with the example are RGB images, so the example converts each image to the YC_bC_r color space before compressing it. This task is described in the next section.

Converting an XIL Image to the Proper Format

Compressors have certain requirements for the images they compress. (See the section "Putting Compressed Data into a CIS" on page 251 for a list of these requirements.) For example, the JPEG compressor used in this example requires `XIL_BYTE` images. In addition, the compressor will produce a bitstream that can be decompressed most rapidly if the input is YC_bC_r images.

The images provided for you to use as input to the example are RGB XIL_BYTE images. Thus, the example does not have to convert the images in any way, but can produce a bitstream that can be played back the fastest if it converts the color space of the images. This conversion is handled in the routine image_transform().

```
XilImage
image_transform(XilSystemState state, XilImage image,
    int desired_nbands, char *desired_colorspace)
{
    XilImage new_image;
    unsigned int width, height, nbands;
    XilDataType datatype;

    xil_get_info(image, &width, &height, &nbands, &datatype);
    new_image = xil_create(state, width, height, desired_nbands,
        datatype);
    if (new_image == NULL) {
      /* XIL sends an error msg to stderr if image create fails */
        return(NULL);
    }
    xil_set_colorspace(new_image,
        xil_colorspace_get_by_name(state, desired_colorspace));
    xil_color_convert(image, new_image);
    xil_destroy(image);
    return(new_image);
}
```

This routine is passed a source image and a desired color space (ycc601). Given this data, the routine:

- Creates a second XIL image, new_image, that has the same width, height, and data type as image and a number of bands that fits the desired color space. Because a YC_bC_r image must have three bands, new_image is a 3-band image.
- Uses the function xil_set_colorspace() to set new_image's color space attribute to ycc601.
- Calls xil_color_convert() to convert the data in image from the RGB color space to the YC_bC_r color space. The converted data is stored in new_image.
- Destroys image since it is no longer needed.

A JPEG playback program will also run fastest if the images it decompresses are multiples of 16 in width and height. For this reason, the example uses the following code to clip each source image to a multiple of 16 in width and height before compressing it.

```
XilImage src, child_clip;

child_clip = xil_create_child(src, 0, 0,
                              xil_get_width(src) & ~0xf,
                              xil_get_height(src) & ~0xf, 0, 3);
```

Compressing an XIL Image

Compressing the clipped YC_bC_r image requires a single function call.

```
XilImage src;
XilCis cis;

xil_compress(child_clip, cis);
```

The first argument to this function is the child YC_bC_r image `child_clip`, and the second is a handle to the program's CIS. The call both compresses the image and writes the compressed data to the CIS.

Destroying an XIL Image

Once an image has been compressed, the example destroys that image to free the resources associated with it.

```
xil_destroy(child_clip);
xil_destroy(src);
```

Writing the Compressed Data to a File

After compressing each frame of video, the example calls a routine `write_file()` to write any compressed data in the CIS to the program's output file. The basic algorithm for this routine is shown below.

- While the CIS contains data that has not been read,
  a. Get a pointer to the first byte of compressed data that has not been read yet. After the first call to the routine that gets this pointer, the pointer will point to the first byte of data in the CIS.
  b. Using the pointer retrieved in the previous step, read data from the CIS and write it to a file.

Note – The loop is necessary because all of the compressed data in the CIS may not be in one continuous buffer.

Here is code that implements this algorithm:

```
int nbytes, nframes;
XilCis cis;
Xil_unsigned8 *data;
FILE *out;

while (xil_cis_has_frame(cis) == TRUE) {
    data = (Xil_unsigned8 *)xil_cis_get_bits_ptr(cis, &nbytes,
        &nframes);
    fwrite((char *)data, sizeof(Xil_unsigned8), nbytes, out);
}
```

The function `xil_cis_has_frame()` determines whether there is at least one complete frame's worth of compressed data in the CIS that has not yet been read. If there is remaining data, the function returns `TRUE`, and the statements inside the loop are executed.

Note – The routine must check for more than one frame's worth of data in the CIS because compressors are not required to write data to the CIS each time `xil_compress()` is called. Instead, they may read a number of frames of video and store them internally before doing any compression. Then, they may write a number of frames' worth of compressed data to the CIS at once. This type of strategy is necessary, for instance, when a compressor wants to do interframe compression.

The first call to `xil_cis_get_bits_ptr()` returns a generic pointer to the beginning of the compressed data in the CIS. It also returns the number of bytes the pointer points to (`nbytes`) and the number of video frames these bytes represent (`nframes`). In addition to returning these values, the function changes an attribute of the CIS called its *read frame* from 0 to `nframes`. (Frames are numbered beginning with 0.) Thus, a second call to `xil_cis_get_bits_ptr()` will return a pointer to the first byte of compressed data that is part of the frame numbered `nframes`. The arguments to this function are a handle to your CIS and the addresses of the variables `nbytes` and `nframes`.

The only other action taking place in the loop is that the example is using the function `fwrite()` to write the compressed data pointed to by `data` to the output file `out`.

Performing Any Outstanding Compression Operations

When the loop described in the preceding section ends, the compressor may have read frames of video that it has not yet compressed. To deal with this possibility, the example makes the following two calls.

```
XilCis cis;
FILE *out;

xil_cis_flush(cis);
write_file(cis, &total_nbytes, &total_nframes, out);
```

The function `xil_cis_flush()` instructs the compressor to compress any images it has read but not compressed and to write the last of its output to the CIS. The call to `write_file()` is the same as the one used in the loop discussed in the last section. It writes to the output file any compressed data written to the CIS as a result of the call to `xil_cis_flush()`.

Playing a JPEG Movie

This section discusses a movie-player example supplied with the library. This program can display Cell, CellB, JPEG, H.261, and MPEG-1 movies on 8- and 24-bit displays.

The source files for the movie-player example—`xilcis_example.c`, `memmap.c`, `memmap.h`, and `xilcis_color.c`—can be found in the directory `$XILHOME/examples/movie_player_example`. This directory also contains a one-frame JPEG bitstream that the example can used as input and a makefile you can use to build the program. In addition, you can use as input to this program the output of the movie-maker example discussed earlier in this chapter.

Note – The JPEG code in the program is optimized to handle movies containing YC_bC_r images. The JPEG movie supplied with the example contains this type of image.

After briefly discussing how to build and run the example, the section takes a look at the code the example uses to play back a JPEG movie on an 8-bit display. (If you look at the program, you'll notice that the CellB, H.261, and MPEG-1 cases are handled very similarly to the JPEG case.) The code specific to the Cell case for 8-bit displays is covered in the section "Playing Cell Movies" on page 241.

The basic algorithm the example uses to play a JPEG movie on an indexed-color display is shown below.

1. Memory map the contents of the movie file (the compressed data) into your process's address space.
2. Create a CIS.
3. Give the CIS a pointer to the compressed data you memory mapped earlier. At this point, the CIS is all set up.

4. Create a display image in which to show the movie by following these steps:
   a. Determine the dimensions of the frames in the movie.
   b. Create an X window that is equal in width and height to the frames in the movie.
   c. Create an XIL display image from the X window.
5. Create an XIL image that will hold frames as they are decompressed. This image must have the same width, height, number of bands, and data type as the frames in the movie.
6. Initialize the parameters the example will use when dithering each 24-bit XIL image (frame of video) so that it can be displayed on an 8-bit frame buffer.
7. Install an X colormap.
8. Play back the movie. While the CIS contains unread frames of compressed data:
   a. Decompress a frame of video and store the resulting image in the XIL image created above.
   b. Dither the XIL image, and write the dithered image to the display.

Running the Movie Player

To run the movie player, change your working directory to `$XILHOME/examples/movie_player_example`, and then build the program in that directory by typing `make`. You should execute the example from this same directory, using a command line of the following form:

```
% xilcis_example [-i filename] [-c | -cb | -h | -m] [-s width height]
```

Table 12-1 describes the example's command-line options.

Table 12-1 Command-Line Options for `xilcis_example`

Option	What It Does
`-i` *filename*	Specifies the name of the movie (a file containing compressed movie frames) that you want to play back. If you omit this option, the program will play back a one-frame movie called `earth.jpg`.
`-c`	Instructs the movie player to play back a Cell movie. By default, the player is set up to play a JPEG movie.
`-cb`	Instructs the movie player to play back a CellB bytestream.
`-h`	Instructs the movie player to play back an H.261 bitstream.
`-m`	Instructs the movie player to play back an MPEG-1 bitstream.
`-s` *width height*	If you're decoding a CellB bytestream, you must use this option. The arguments *width* and *height* are the width and height in pixels of the images being decoded.

When the program runs, it shows the movie you have selected in an X window. To terminate the program, move your cursor into the program's window and press any mouse button.

Memory Mapping the Movie

When you run the example program, you use the `-i` *filename* option to pass it the name of a file containing a movie. Before the example can attach the compressed image data in the file to a CIS, it must map the contents of the file to system memory. You can find the code that handles this task in the source file `memmap.c`.

As you'll see in that source file, after opening the movie file, the example uses the system call `fstat(2V)` to determine the length of the movie in bytes and the system call `mmap(2)` to memory map the file and get a pointer to the beginning of the datastream. Following these calls, the structure member `memfile->mstart` is a pointer to the beginning of the datastream, and `memfile->mlen` is the number of bytes in the datastream.

Creating a CIS

After mapping the compressed movie data to memory, the example creates a CIS to hold that data. The code used to create this `XilCis` data structure is shown below.

```
XilCis cis;
XilSystemState state;
char *cis_type = "Jpeg";

cis = xil_cis_create(state, cis_type);
if (!cis) {
   /* XIL sends error message to stderr if xil_cis_create fails */
    exit(1);
}
```

The arguments to `xil_cis_create()` are a handle to a system-state data structure returned by an earlier call to `xil_open()` and the name of the codec or decompressor that will be used to decompress frames from the datastream, in this case "`Jpeg`". The function returns a handle to the newly created CIS.

Putting Compressed Data in a CIS

Now that the CIS exists and the example has a pointer to the JPEG bitstream, the example can set the data-pointer member of the CIS to point to the beginning of the JPEG bitstream. It does this using a call to `xil_cis_put_bits_ptr()`:

```
XilCis cis;
int frame_count = -1;

xil_cis_put_bits_ptr(cis, memfile->mlen, frame_count,
    memfile->mstart, NULL);
```

Table 12-2 explains what the several arguments to the function represent.

Table 12-2 Parameters to `xil_cis_put_bits_ptr()`

Argument	What It Represents
`cis`	A handle to the CIS whose data pointer is being set.
`memfile->mlen`	The length in bytes of the JPEG bitstream.
`frame_count`	Indicates that the number of video frames the JPEG bitstream represents is unknown (because `frame_count` is set to -1). If the number of frames in the movie were known, `frame_count` would be set to that number.
`memfile->mstart`	A pointer to the beginning of the JPEG bitstream.
`NULL`	Indicates that the example has assumed responsibility for freeing the memory in which the JPEG bitstream is stored. Alternatively, the example could have passed a pointer to a function to `xil_cis_put_bits_ptr()`. In the latter case, the function passed to `xil_cis_put_bits_ptr()` would be called if the CIS were destroyed or reset.

After the call to `xil_cis_put_bits_ptr()` has been made, the CIS is ready for use. Before the example can begin decompressing frames of video, however, it must create two XIL images: an 8-bit display image in which to display the video and a 24-bit memory image to serve as a destination image for the decompression function.

Creating a Display Image

There are three steps involved in creating the 8-bit display image in which the movie will be shown.

1. **Determine the dimensions of the images stored in the CIS.**
   The code used to get these dimensions is shown below.

```
XilImageType outputtype;
XilCis cis;
unsigned int cis_xsize, cis_ysize, cis_nbands;
XilDataType cis_datatype;

outputtype = xil_cis_get_output_type(cis);
xil_imagetype_get_info(outputtype, &cis_xsize, &cis_ysize,
    &cis_nbands, &cis_datatype);
```

   The first function called, `xil_cis_get_output_type()`, takes a handle to the CIS as its only argument and returns an image type. This image type, a data structure of type `XilImageType`, contains information about the images stored in the CIS, such as their width, height, number of bands, and data type.

   The following call to `xil_imagetype_get_info()` takes the image type as its first argument and returns the width, height, number of bands, and data type of the image type in its remaining arguments. The data type, stored in `cis_datatype`, will be one of the following enumeration constants: `XIL_BIT`, `XIL_BYTE`, or `XIL_SHORT`. (In this case, the value will be `XIL_BYTE` because the movie supplied with the XIL release was made from 3-band `XIL_BYTE` images.)

2. **Create an X window that matches the width and height of the images in the CIS.**
   In this example, the X window is created with a call to `XCreateSimpleWindow()`. When the example is run on a system with an 8-bit display, this window will be 8 bits deep by default.

```
int screen_num;
Display *display;
Window window;

screen_num = DefaultScreen(display);
window = XCreateSimpleWindow(display,
    RootWindow(display, screen_num), 0, 0, cis_xsize,
    cis_ysize, 0, BlackPixel(display, screen_num),
    WhitePixel(display, screen_num));
```

3. **Create a display image based on the X window created above.**
   This display image will be the destination image to which frames are written in order to display them. To create the display image, the example calls `xil_create_from_window()`.

```
XilImage displayimage = NULL;
XilSystemState state;
Display *display;
Window window;

displayimage = xil_create_from_window(state, display, window);
if (!displayimage) {
    /* XIL err msg to stderr if xil_create_from_window fails */
    exit(1);
}
```

   For a complete discussion of display images, see the section "Display Images" on page 44.

Creating an Image to Hold Decompressed Frames

If you have loaded the JPEG movie supplied with the XIL release (`earth.jpg`) or a JPEG movie you created with the example movie maker into your CIS, the images in the CIS are YC_bC_r images (24 bits deep). Because they cannot be decompressed directly into the program's display image (8 bits deep), the example must create an XIL memory image whose purpose is to hold those decompressed frames. That is, for each frame in the movie, the example must perform the two-step sequence shown in Figure 12-2.

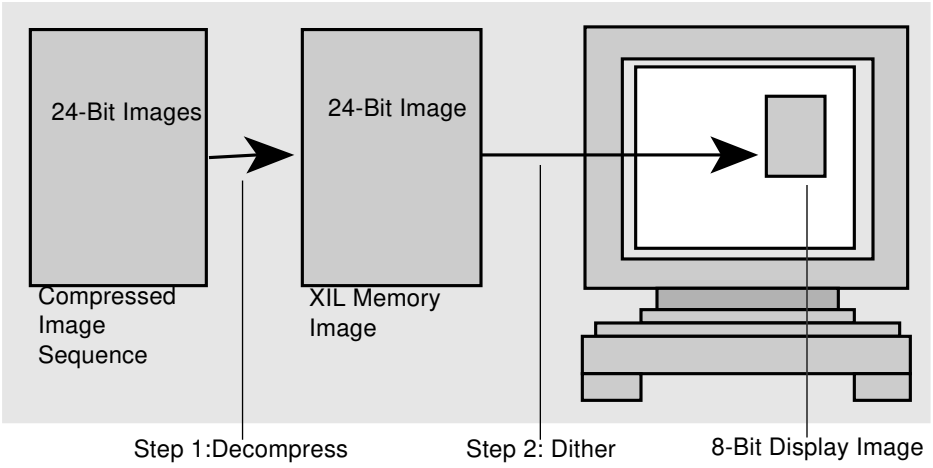

Figure 12-2 Decompressing and Dithering a Frame of Video

The code used to create the XIL memory image is shown below:.

```
XilSystemState state;
XilImage imageYCC = NULL;
unsigned int cis_xsize, cis_ysize, cis_nbands;
XilDataType cis_datatype;

imageYCC = xil_create(state, cis_xsize, cis_ysize, cis_nbands,
    cis_datatype);
```

The variables `cis_xsize`, `cis_ysize`, `cis_nbands`, and `cis_datatype` were set earlier by a call to `xil_imagetype_get_info()`, which returned the width, height, number of bands, and data type of the compressed images in the CIS. Thus, the image returned here, `imageYCC`, will be identical to the images stored in the CIS in both dimensions and data type.

Initializing Parameters to Be Used with the Dither Function

Next, the example program produces the colorcube and dither mask that it will need later to dither images to the display. To get the colorcube, the example calls `xil_lookup_get_by_name()`.

```
XilLookup colorcube;

colorcube = xil_lookup_get_by_name(state, "cc855");
```

This call returns a handle to a special colorcube that the library creates when it is initialized. This colorcube has dimensions of 8, 5, and 5 and is designed for dithering the values in a YC_bC_r image to 200 colors.

The program then creates a dither mask:

```
XilDitherMask dmask;

dmask = xil_dithermask_get_by_name(state, "dm443");
```

This call returns a handle to a special dither mask the library creates when it is initialized. The mask is 4 pixels high and wide and 3 bands deep.

Installing an X Colormap

At this point, the program calls the function `create_cmap()`, from the source file `xilcis_color.c`, to set up the X colormap that will be used in displaying the dithered frames. This function contains primarily Xlib code, but does get some information from the XIL library to get started.

First, the function is passed an XIL lookup table called `yuv_to_rgb`. This is a standard lookup table that is useful for converting the values in a dithered YC_bC_r image to RGB values.

Note – The YC_bC_r images must have been dithered using the standard colorcube `cc855`.

This lookup table specifies which color values should be written to which colorcells in the X colormap.

The function create_cmap() then calls xil_lookup_get_num_entries() to determine the number of entries in the colormap.

```
int cmapsize;

cmapsize = xil_lookup_get_num_entries(yuvtorgb);
```

Once the function knows how many colorcells it needs to allocate, it creates an X colormap and allocates the colorcells it needs to hold the information stored in yuvtorgb. Note that to avoid colormap flashing, the function avoids allocating the last two colorcells in the X colormap if possible and also leaves free as many colorcells as possible at the beginning of the X colormap.

At this point, create_cmap() allocates a buffer called data and then uses the function xil_lookup_get_values() to copy colormap entries from the lookup table yuvtorgb to the buffer data.

```
Xil_unsigned8 *data;

data = (Xil_unsigned8 *)malloc(sizeof(Xil_unsigned8) * cmapsize
    * 3);
xil_lookup_get_values(yuvtorgb,
    xil_lookup_get_offset(yuvtorgb), cmapsize, data);
```

The second argument to xil_lookup_get_values() enables you to start copying entries from the lookup table at an entry other than 0, and here is set to the offset of yuvtorgb.

Once create_cmap() has copied the colormap information from yuvtorgb to the buffer data, the function has direct access to the color values it needs to store in the X colormap, and it goes on to store those values. The only remaining XIL call in the function create_cmap() changes the offset of the colorcube that main uses to dither decompressed images to match the offset at which create_cmap() began writing color values to the X colormap.

```
xil_lookup_set_offset(colorcube, (unsigned int)pixels[0]);
```

The first argument to xil_lookup_set_offset() is a handle to the colorcube—main passed this handle to cmap_create()—and the second represents the pixel value associated with the first colorcell in which

`cmap_create()` stored color values. This call ensures that all the pixel values in the dithered images created in `main` will map to the correct spot in the X colormap.

Playing the Movie

After returning from `create_cmap()`, the program is ready to play the movie. The calls it uses to do this are shown below.

```
while (xil_cis_has_frame(cis)) {
    xil_decompress(cis, imageYCC);
    xil_rescale(imageYCC, imageYCC, scale, offset);
    xil_ordered_dither(imageYCC, displayimage, colorcube,
        dmask);
}
```

The loop is controlled by the return value of `xil_cis_has_frame()`. As long as the CIS contains at least one complete frame of video, the function returns 1, and the statements in the loop are performed.

The call to `xil_decompress()` decompresses an image from the CIS and stores it in the intermediate image `imageYCC`. Since this is a CCIR Rec. 601 YC_bC_r image, the values in the Y band of the image can range from 16 to 235, and those in the color bands can range from 16 to 240. Before such an image is dithered, the values in each band should be scaled to fall in the range 0 to 255.

This rescaling is handled by the call to `xil_rescale()`. Since `imageYCC` is both the source and destination image for the operation, the rescale is performed in place. The scale and offset arguments were defined earlier in this way:

```
float scale[3], offset[3];

scale[0] = 255.0 / (235.0 - 16.0);
scale[1] = 255.0 / (240.0 - 16.0);
scale[2] = 255.0 / (240.0 - 16.0);
offset[0] = -16.0 * scale[0];
offset[1] = -16.0 * scale[1];
offset[2] = -16.0 * scale[2];
```

Finally, `xil_ordered_dither()` dithers the 3-band YC_bC_r image in `imageYCC` using the colorcube and dither mask created earlier and writes the result to the display image `displayimage`. This results in a frame of the movie being shown in the X window.

Note – When a JPEG movie made from YC_bC_r images is displayed on an 8-bit frame buffer, as in this example, the exact sequence of calls you use to play the movie has a dramatic effect on the speed with which the movie is shown. The reason for this is that by using the deferred-execution scheme explained in Chapter 21, "Acceleration in XIL Programs," the XIL library can look for a certain sequence of functions at runtime and, if it finds that sequence, replace all the functions in the sequence with a single, highly optimized routine. In this example the sequence `xil_decompress()`, `xil_rescale()`, `xil_ordered_dither()` is such a sequence. Thus, when you play a JPEG movie using this program, these functions are not executed, but are replaced by an optimized function (*molecule*) that performs the jobs of all three functions. See the section "Video Decompression Molecules" on page 399 for a complete list of decompression molecules.

Playing Cell Movies

The example movie player performs the same steps regardless of whether its input is a JPEG movie or a Cell movie up through the point where it creates a display image. From that point on, however, it treats the JPEG and Cell cases differently to get the maximum playback speed for each case. The algorithm for the Cell case is summarized below:

1. Install an X colormap.
2. Create an XIL image to hold images as they are decompressed.
3. Decompress frames from the movie and display them.

Each of these steps is considered in more detail in the following sections.

Installing an X Colormap

As you look at this part of the example program, keep in mind that the Cell playback code may update the X colormap many times as it plays back a movie. In fact, the decompressor may use a different colormap for each frame of a movie. This need to modify the X colormap makes colormap handling a bit more complex than it is in the JPEG case.

The first thing the example does in this section is to call the function `create_cmap()`, which is defined in the file `xilcis_color.c`:

```
XilLookup xil_cmap;
Colormap x_cmap;
XilIndexList *ilist;

xil_cmap = create_cmap(state, cis, display, window,
  DefaultScreen(display), &x_cmap, CELL, ilist, NULL, NULL);
```

As you'll see below, this routine returns an XIL lookup table that serves as the colormap for the Cell decompressor. The argument `&x_cmap` is the address of an X `Colormap` and `ilist` is a structure that is defined (in an XIL header file) as follows:

```
typedef struct
{
    Xil_unsigned32 *pixels;
    Xil_unsigned16 ncolors;
} XilIndexList;
```

The array `pixels` can hold up to `ncolors` colormap index values, and these values will later determine which X colormap entries the Cell decompressor can modify.

The first thing the `create_cmap()` function does is to look at the CIS's `DECOMPRESSOR_MAX_CMAP_SIZE` attribute to determine how many colorcells to allocate in the X colormap.

```
int cmapsize;

xil_cis_get_attribute(cis, "DECOMPRESSOR_MAX_CMAP_SIZE",
    (void **)&cmapsize);
```

The function then creates an X colormap and allocates `cmapsize` colorcells in that colormap.

The next section of code sets up the `ilist` structure that will be used to determine which X colormap entries the Cell decompressor can modify. In this example, `ilist` is being set up so that the decompressor can modify any of the colorcells allocated by `create_cmap()`.

```
ilist->pixels = (Xil_unsigned32 *)malloc(sizeof(Xil_unsigned32)
    * cmapsize);
ilist->ncolors = cmapsize;
for (i = 0; i < cmapsize; i++)
    ilist->pixels[i] = (Xil_unsigned32)pixels[i];
```

Next, the `cmap_create()` function creates the XIL colormap that it will pass back to `main`. Because the XIL routine that creates this colormap requires as one of its arguments an array of bytes containing the color values to be stored

in the colormap, `cmap_create()` first reads the color values in the default X colormap into a buffer called `data`. It then uses `xil_lookup_create()` to create the XIL colormap.

```
data = (Xil_unsigned8 *)malloc(sizeof(Xil_unsigned8) * cmapsize
    * 3);
for (i = 0; i < cmapsize; i++)
    cdefs[i].pixel = i + pixels[0];
XQueryColors(display, DefaultColormap(display, screen), cdefs,
    cmapsize);
for (i = 0, j = 0; i < cmapsize; i++, j += 3) {
    data[j] = cdefs[i].blue >> 8;
    data[j + 1] = cdefs[i].green >> 8;
    data[j + 2] = cdefs[i].red >> 8;
}
lut = xil_lookup_create(state, XIL_BYTE, XIL_BYTE, 3, cmapsize,
    (int)pixels[0], data);
```

The lookup table created contains three bands on the output side and `cmapsize` entries. Also, note that color values are entered in the table in BGR order.

After `create_cmap()` returns the XIL lookup table to `main`, the example gets the version number of this colormap (`xil_cmap`):

```
XilVersionNumber lu_version;

lu_version = xil_lookup_get_version(xil_cmap);
```

This version number, `lu_version`, changes whenever the lookup table's contents are changed by the Cell decompressor. Therefore, after decompressing each movie frame, the program can test the version number to determine whether it needs to update the X colormap.

The example then sets two important attributes of the Cell decompressor: `DECOMPRESSOR_COLORMAP` and `RDWR_INDICES`.

```
xil_cis_set_attribute(cis, "DECOMPRESSOR_COLORMAP", xil_cmap);
xil_cis_set_attribute(cis, "RDWR_INDICES", ilist);
```

The DECOMPRESSOR_COLORMAP attribute specifies the XIL lookup table that will serve as the Cell decompressor's XIL colormap. Because colormap information is stored in the Cell bytestream along with pixel values, any time a frame is decompressed, the contents of the decompressor's colormap can change. As you'll see later, when the contents of the colormap change, a subset of its entries are written to the X colormap before the next video frame is displayed.

By default, however, the decompressor's colormap is read only. To make an entry writable, you register its index in RDWR_INDICES (in this case, ilist->pixels). Only the entries whose indexes are stored in this array can change. For example, in this program, only the entries whose indexes correspond to those of the X colorcells allocated earlier can be written to.

Note – If you want to play Cell movies using a fixed colormap (for example, to avoid colormap flashing between windows), don't set the RDWR_INDICES attribute. In this case, the Cell compressor will do its best using the colors stored in the lookup table xil_cmap.

Creating an Image to Hold Decompressed Frames

The program now creates a 3-band XIL_BYTE image to hold frames as they are decompressed. This task is handled with a call to xil_create():

```
XilImage image24 = NULL;

image24  = xil_create(state, cis_xsize, cis_ysize, 3, XIL_BYTE);
```

Later in the program, these images will be converted to 8-bit images for display.

12

Playing the Movie

Like the JPEG player, the Cell player plays a movie by executing a few statements in a loop that terminates when the function call `xil_cis_has_frame(cis)` returns 0.

First, the example decompresses a frame of data using the function `xil_decompress()` and checks the version number of the decompressor's XIL colormap:

```
xil_decompress(cis, image24);
if (lu_version != xil_lookup_get_version(xil_cmap))
{
    cell_install_cmap(x_cmap, displayimage, xil_cmap, ilist);
    lu_version = xil_lookup_get_version(xil_cmap);
}
```

The version number of the original XIL colormap was stored in `lu_version`. If this colormap has changed, then `xil_cmap`'s version number will have changed, and the program calls `cell_install_cmap()` to update the X colormap. The program then saves `xil_cmap`'s new version number in `lu_version` so that it can check again to see if the colormap has changed after the next call to `xil_decompress()`.

Finally, the example converts the RGB image stored in `image24` to an 8-bit image and copies the 8-bit image to `displayimage`. This is done using the function `xil_nearest_color()`.

```
xil_nearest_color(image24, displayimage, xil_cmap);
```

For each set of RGB values in `image24`, this routine performs a pixel-by-pixel search for the nearest matching color in the supplied colormap (`xil_cmap`) and sets the destination image pixel value to the appropriate colormap index.

Note – When a Cell movie is displayed on an 8-bit frame buffer, as in this example, the exact sequence of calls you use to play the movie has a dramatic effect on the speed with which the movie is shown. The reason for this is that by using the deferred-execution scheme explained in Chapter 21, "Acceleration in XIL Programs," the XIL library can look for a certain sequence of functions and, if it finds that sequence, replace all the functions in the sequence with a single highly optimized routine. In this example the sequence `xil_decompress(),xil_nearest_color()` is such a sequence. Thus, when you play a Cell movie using this program, these functions are not executed, but are replaced by an optimized function (*molecule*) that performs the jobs of both functions. See the section "Cell Molecules" on page 288 for a complete list of Cell-related molecules.

12