Video Codec Design
Iain E. G. Richardson

ISBNs: 0-471-48553-5 (Hardback); 0-470-84783-2 (Electronic)

13
Video CODEC Design

13.1 INTRODUCTION

In this chapter we bring together some of the concepts discussed earlier and examine the issues faced by designers of video CODECs and systems that interface to video CODECs. Key issues include interfacing (the format of the input and output data, controlling the operation of the CODEC), performance (frame rate, compression, quality), resource usage (computational resources, chip area) and design time. This last issue is important because of the fast pace of change in the market for multimedia communication systems. A short time-to-market is critical for video coding applications and we discuss methods of streamlining the design flow. We present design strategies for two types of video CODEC, a software implementation (suitable for a general-purpose processor) and a hardware implementation (for FPGA or ASIC).

13.2 VIDEO CODEC INTERFACE

Figure 13.1 shows the main interfaces to a video encoder and video decoder:

(a) Encoder input: frames of uncompressed video (from a frame grabber or other source); control parameters.

(b) Encoder output: compressed bit stream (adapted for the transmission network, see Chapter 11); status parameters.

(c) Decoder input: compressed bit stream; control parameters.

(d) Decoder output: frames of uncompressed video (send to a display unit); status parameters.

A video CODEC is typically controlled by a 'host' application or processor that deals with higher-level application and protocol issues.

13.2.1 Video In/Out

There are many options available for the format of uncompressed video into the encoder or out of the decoder and we list some examples here. (The four-character codes listed for options (a) and (b) are 'FOURCC' descriptors originally defined as part of the AVI video file format.)

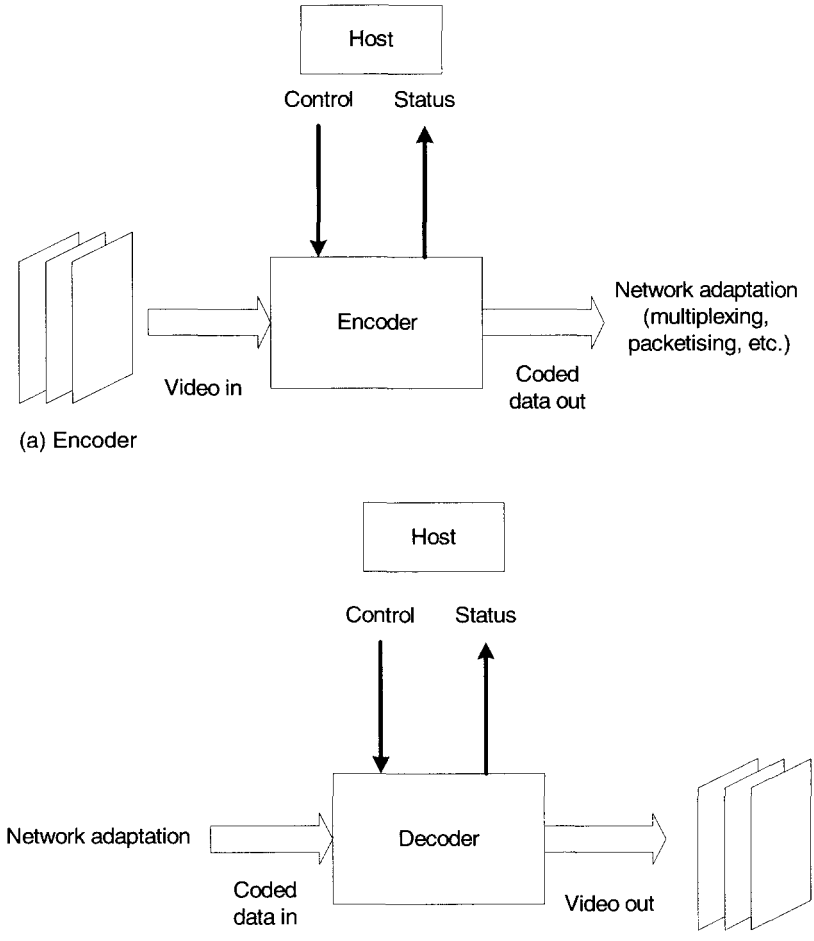

Figure 13.1 Video encoder (a) and decoder (b) interfaces

(a) YUY2 (4 : 2 : 2). The structure of this format is shown in Figure 13.2. A sample of Y (luminance) data is followed by a sample of Cb (blue colour difference), a second sample of Y, a sample of Cr (red colour difference), and so on. The result is that the chrominance components have the same vertical resolution as the luminance component but half the horizontal resolution (i.e. 4 : 2 : 2 sampling as described in Chapter 2). In the example in the figure, the luminance resolution is 176 × 144 and the chrominance resolution is 88 × 144.

(b) YV12 (4 : 2 : 0) (Figure 13.3). The luminance samples for the current frame are stored in sequence, followed by the Cr samples and then the Cb samples. The Cr and Cb samples have half the horizontal and vertical resolution of the Y samples. Each colour

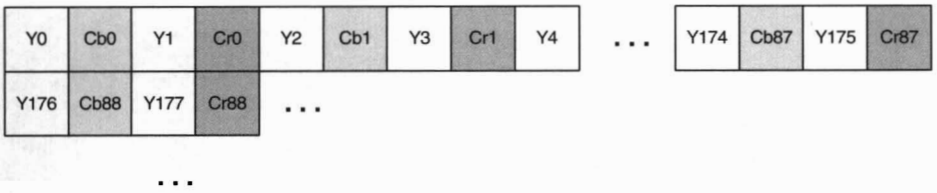

Figure 13.2 YUY2 (4:2:2)

pixel in the original image maps to an average of 12 bits (effectively 1 Y sample, $\frac{1}{4}$ Cr sample and $\frac{1}{4}$ Cb sample), hence the name 'YV12'. Figure 13.4 shows an example of a frame stored in this format, with the luminance array first followed by the half-width and half-height Cr and Cb arrays.

(c) Separate buffers for each component (Y, Cr, Cb). The CODEC is passed a pointer to the start of each buffer prior to encoding or decoding a frame.

As well as reading the source frames (encoder) and writing the decoded frames (decoder), both encoder and decoder require to store one or more reconstructed reference frames for motion-compensated prediction. These frame stores may be part of the CODEC (e.g. internally allocated arrays in a software CODEC) or separate from the CODEC (e.g. external RAM in a hardware CODEC).

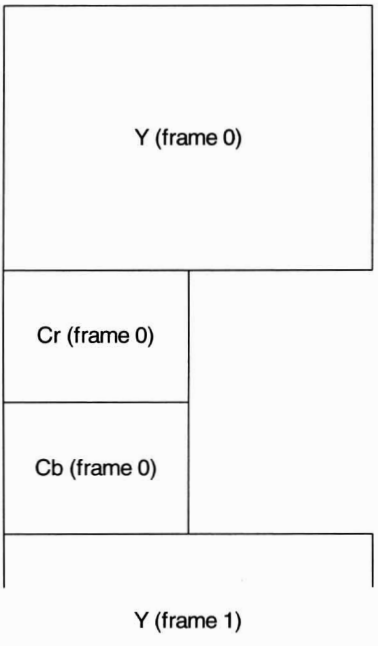

Figure 13.3 YV12 (4:2:0)

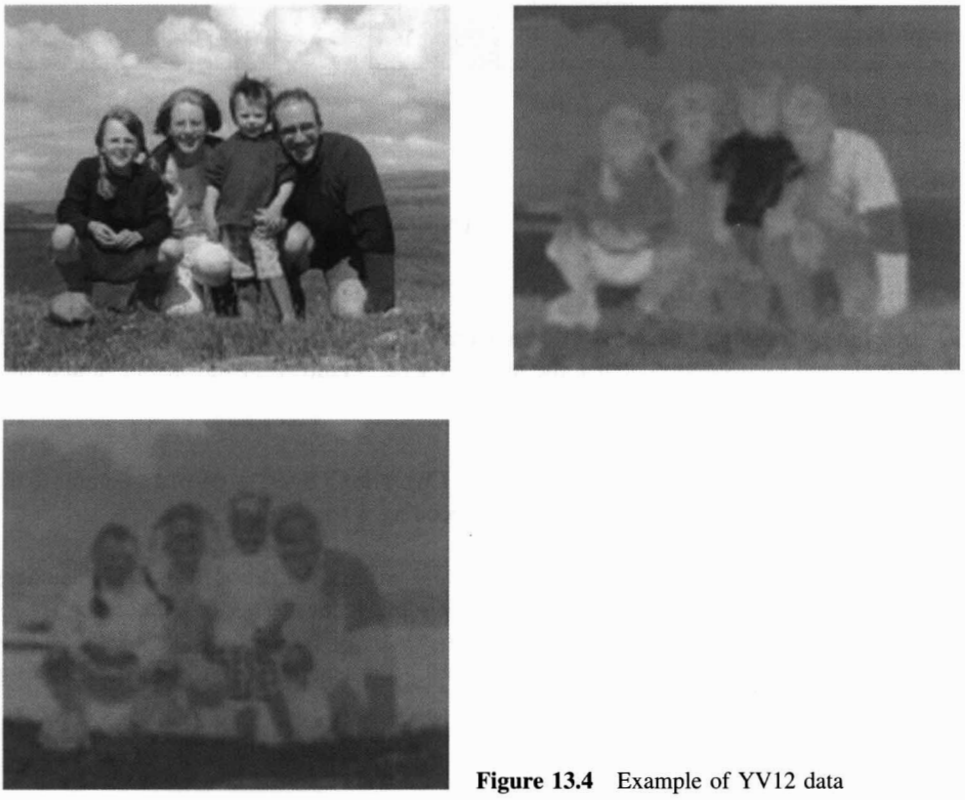

Figure 13.4 Example of YV12 data

Memory bandwidth may be a particular issue for large frame sizes and high frame rates. For example, in order to encode or decode video at 'television' resolution (ITU-R 601, approximately 576×704 pixels per frame, 25 or 30 frames per second), the encoder or decoder video interface must be capable of transferring 216 Mbps. The data transfer rate may be higher if the encoder or decoder stores reconstructed frames in memory external to the CODEC. If forward prediction is used, the encoder must transfer data corresponding to three complete frames for each encoded frame, as shown in Figure 13.5: reading a new input frame, reading a stored frame for motion estimation and compensation and writing a reconstructed frame. This means that the memory bandwidth at the encoder input is at least $3 \times 216 = 648$ Mbps for ITU-R 601 video. If two or more prediction references are used for motion estimation/compensation (for example, during MPEG-2 B-picture encoding), the memory bandwidth is higher still.

13.2.2 Coded Data In/Out

Coded video data is a continuous sequence of bits describing the syntax elements of coded video, such as headers, transform coefficients and motion vectors. If modified Huffman coding is used, the bit sequence consists of a series of variable-length codes (VLCs) packed together; if arithmetic coding is used, the bits describe a series of fractional numbers each

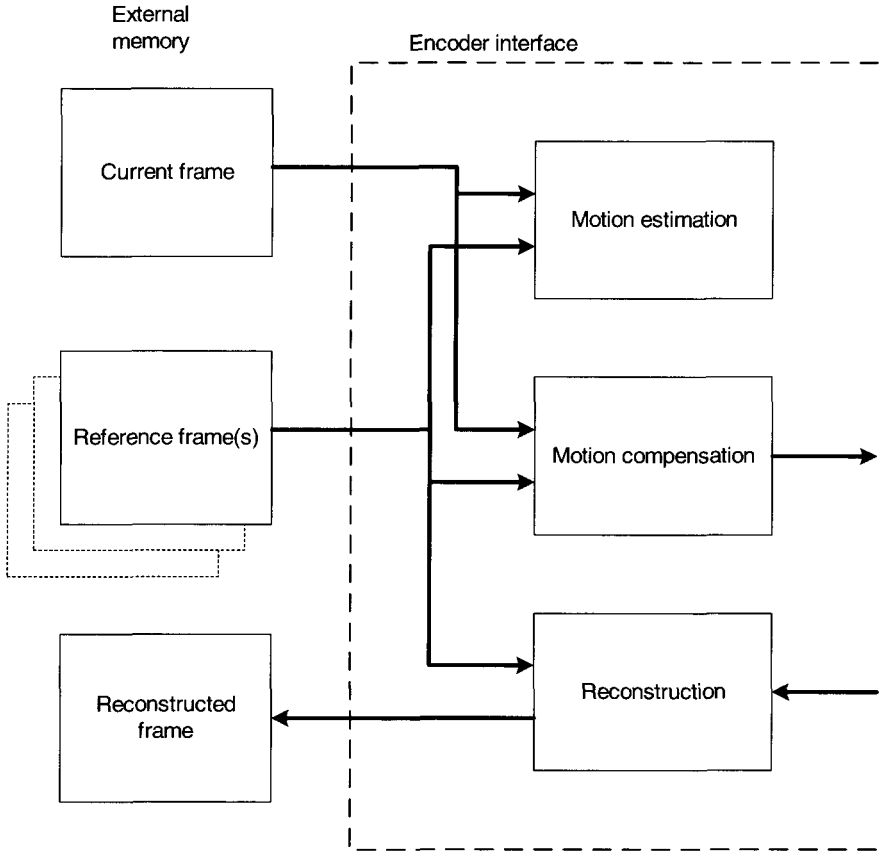

Figure 13.5 Memory access at encoder input

representing a series of data elements (see Chapter 8). The sequence of bits must be mapped to a suitable data unit for transmission/transport, for example:

1. *Bits*: If the transmission channel is capable of dealing with an arbitrary number of bits, no special mapping is required. This may be the case for a dedicated serial channel but is unlikely to be appropriate for most network transmission scenarios.
2. *Bytes or words*: The bit sequence is mapped to an integral number of bytes (8 bits) or words (16 bits, 32 bits, 64 bits, etc.). This is appropriate for many storage or transmission scenarios where data is stored in multiples of a byte. The end of the sequence may require to be padded in order to make up an integral number of bytes.
3. *Complete coded unit*: Partition the coded stream along boundaries that make up coded units within the video syntax. Examples of these coded units include slices (sections of a coded picture in MPEG-1, MPEG-2, MPEG-4 or H.263+), GOBs (groups of blocks, sections of a coded picture in H.261 or H.263) and complete coded pictures. The integrity of the coded unit is preserved during transmission, for example by placing each coded unit in a network packet.

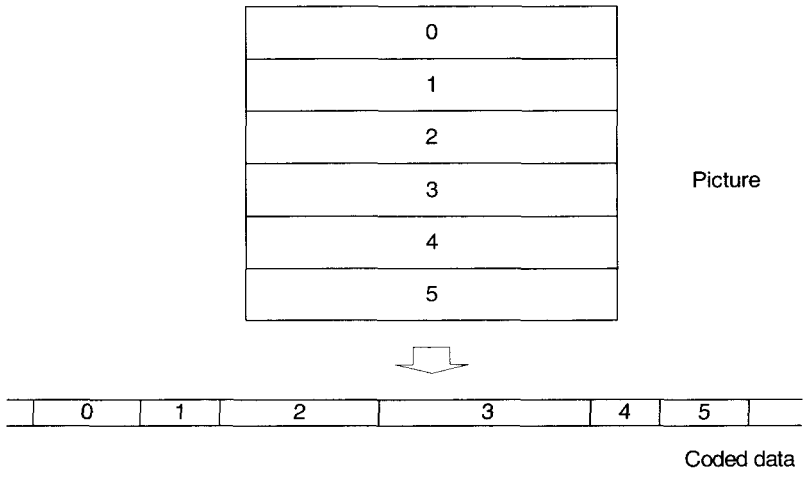

Figure 13.6 GOB locations in a frame and variable-size coded units

Figure 13.6 shows the locations of GOBs in a frame coded using H.263/MPEG-4. The coded units (GOBs in this case) correspond to regular areas of the original frame: however, when encoded, each GOB generates a different number of coded bits (due to variations in content within the frame). The result is that the GOBs generate the variable-size coded units shown in Figure 13.6.

An alternative is to use irregular-sized slices (e.g. using the slice structured mode in H.263+, video packet mode in MPEG-4). Figure 13.7 shows slice boundaries that cover irregular numbers of macroblocks in the original frame and are chosen such that, when coded, each slice contains a similar number of coded bits.

13.2.3 Control Parameters

Some of the more important control parameters are listed here (CODEC application programming interfaces [APIs] might not provide access to all of these parameters).

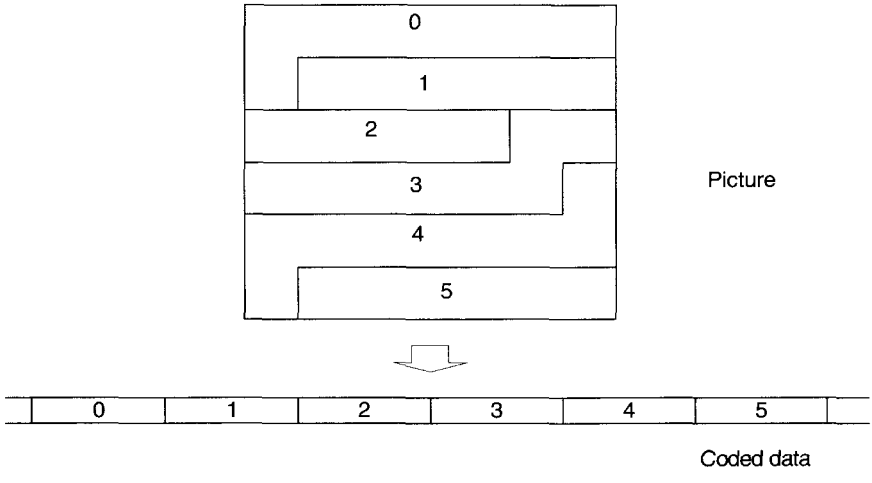

Figure 13.7 Slice boundaries in a picture and constant-size coded units

Encoder

Frame rate May be specified as a number of frames per second or as a proportion of frames to skip during encoding (e.g. skip every second frame). If the encoder is operating in a rate- or computation-constrained environment (see Chapter 10), then this will be a target frame rate (rather than an absolute rate) that may or may not be achievable.

Frame size For example, a 'standard' frame size (QCIF, CIF, ITU-R 601, etc) or a non-standard size.

Target bit rate Required for encoders operating in a rate-controlled environment.

Quantiser step size If rate control is not used, a fixed quantiser step size may be specified: this will give near-constant video quality.

Mode control For example 'inter' or 'intra' coding mode.

Optional mode selection MPEG-2, MPEG-4 and H.263 include a number of optional coding modes (for improved coding efficiency, improved error resilience, etc.). Most CODECs will only support a subset of these modes, and the choice of optional modes to use (if any) must be signalled or negotiated between the encoder and the decoder.

Start/stop encoding A series of video frames.

Decoder

Most of the parameters listed above are signalled to the decoder within the coded bit stream itself. For example, quantiser step size is signalled in frame/picture headers and (optionally) macroblock headers; frame rate is signalled by means of a timing reference in each picture header; mode selection is signalled in the picture header; and so on. Decoder control may be limited to 'start/stop'.

13.2.4 Status Parameters

There are many aspects of CODEC operation that may be useful as status parameters returned to the host application. These may include:

- actual frame rate (may differ from the target frame rate in rate- or computation-constrained environments);
- number of coded bits in each frame;
- macroblock mode statistics (e.g. number of intra/inter-macroblocks);
- quantiser step size for each macroblock (this may be useful for post-decoder filtering, see Chapter 9);

- distribution of coded bits (e.g. proportion of bits allocated to coefficients, motion vector data, header data);
- error indication (returned by the decoder when a transmission error has been detected, possibly with the estimated location of the error in the decoded frame).

13.3 DESIGN OF A SOFTWARE CODEC

In this section we describe the design goals and the main steps required to develop a video CODEC for a software platform.

13.3.1 Design Goals

A real-time software video CODEC has to operate under a number of constraints, perhaps the most important of which are computational (determined by the available processing resources) and bit rate (determined by the transmission or storage medium). Design goals for a software video CODEC may include:

1. Maximise encoded frame rate. A suitable target frame rate depends on the application, for example, 12–15 frames per second for desktop video conferencing and 25–30 frames per second for television-quality applications.
2. Maximise frame size (spatial dimensions).
3. Maximise 'peak' coded bit rate. This may seem an unusual goal since the aim of a CODEC is to compress video: however, it can be useful to take advantage of a high network transmission rate or storage transfer rate (if it is available) so that video can be coded at a high quality. Higher coded bit rates place higher demands on the processor.
4. Maximise video quality (for a given bit rate). Within the constraints of a video coding standard, there are usually many opportunities to 'trade off' video quality against computational complexity, such as the variable complexity algorithms described in Chapter 10.
5. Minimise delay (latency) through the CODEC. This is particularly important for two-way applications (such as video conferencing) where low delay is essential.
6. Minimise compiled code and/or data size. This is important for platforms with limited available memory (such as embedded platforms). Some features of the popular video coding standards (such as the use of B-pictures) provide high compression efficiency at the expense of increased storage requirement.
7. Provide a flexible API, perhaps within a standard framework such as DirectX (see Chapter 12).
8. Ensure that code is robust (i.e. it functions correctly for any video sequence, all allowable coding parameters and under transmission error conditions), maintainable and easily upgradeable (for example to add support for future coding modes and standards).

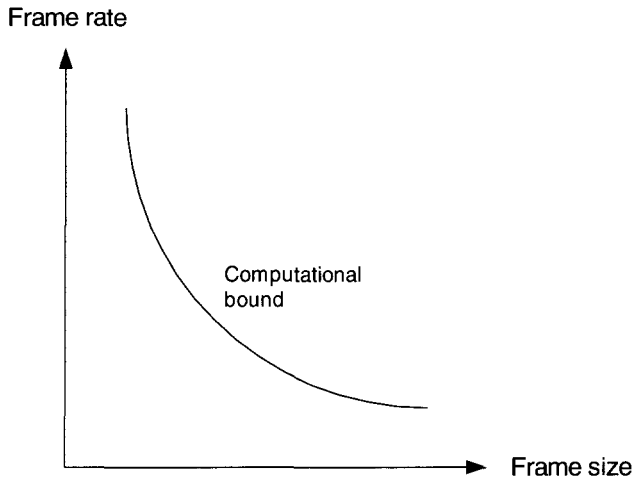

Figure 13.8 Trade-off of frame size and frame rate in a software CODEC

9. Provide platform independence where possible. 'Portable' software that may be executed on a number of platforms can have advantages for development, future migration to other platforms and marketability. However, achieving maximum performance may require some degree of platform-specific optimisation (such as the use of SIMD/VLIW instructions).

The first four design goals listed above may be mutually exclusive. Each of the goals (maximising frame rate, frame size, peak bit rate and video quality) requires an increased allocation of processing resources. A software video CODEC is usually constrained by the available processing resources and/or the available transmission bit rate. In a typical scenario, the number of macroblocks of video that a CODEC can process is roughly constant (determined by either the available bit rate or the available processing resources). This means that increased frame rate can only be achieved at the expense of a smaller frame size and vice versa. The graph in Figure 13.8 illustrates this trade-off between frame size and frame rate in a computation-constrained scenario. It may, however, be possible to 'shift' the line to the right (i.e. increase frame rate without reducing frame size or vice versa) by making better use of the available computational resources.

13.3.2 Specification and Partitioning

Based on the requirements of the syntax (for example, MPEG-2, MPEG-4 or H.263), an initial partition of the functions required to encode and decode a frame of video can be made. Figure 13.9 shows a simplified flow diagram for a block/macroblock-based inter-frame encoder (e.g. MPEG-1, MPEG-2, H.263 or MPEG-4) and Figure 13.10 shows the equivalent decoder flow diagram.

The order of some of the operations is fixed by the syntax of the coding standards. It is necessary to carry out DCT and quantisation of each block within a macroblock before generating the VLCs for the macroblock header: this is because the header typically contains a 'coded block pattern' field that indicates which of the six blocks actually contain coded transform coefficients. There is greater flexibility in deciding the order of some of the other

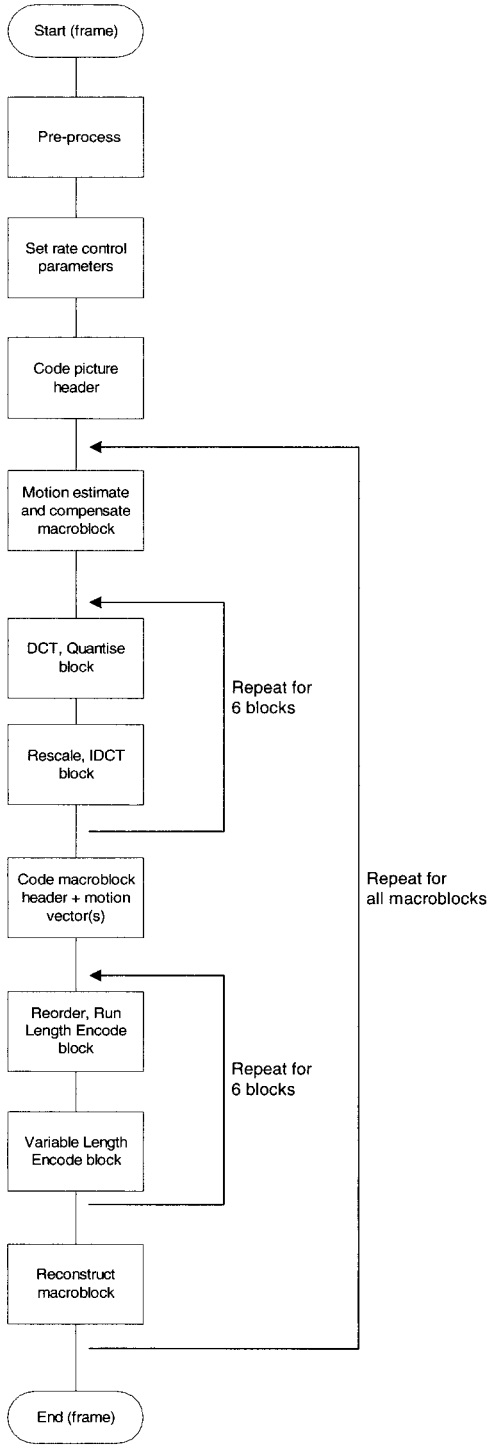

Figure 13.9 Flow diagram: software encoder

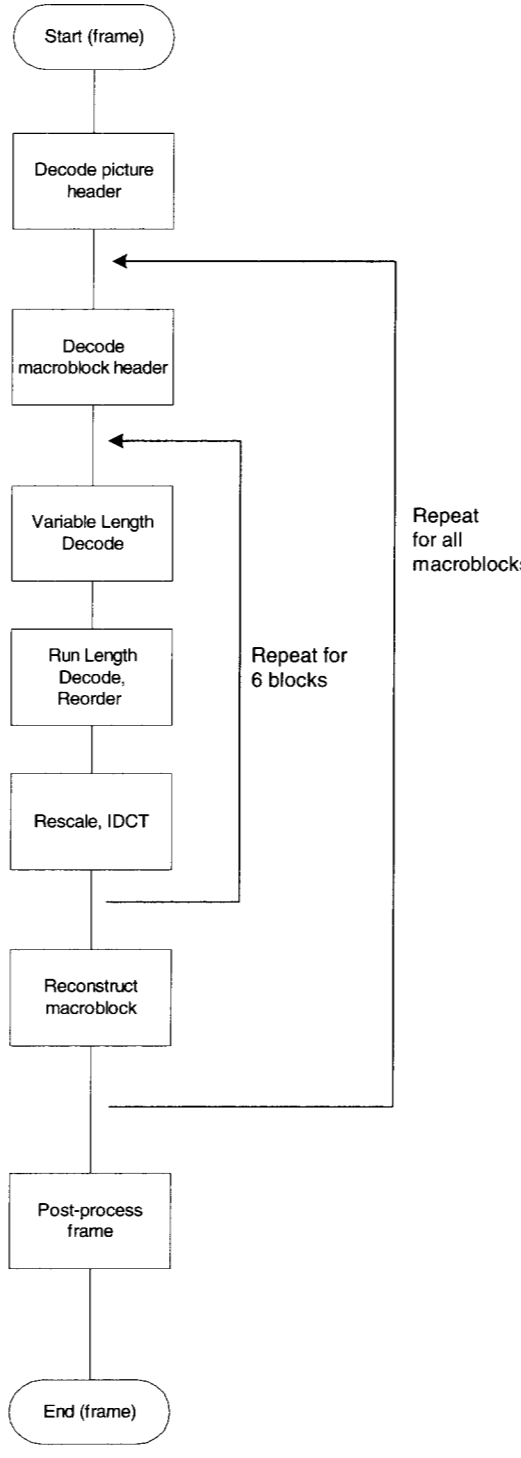

Figure 13.10 Flow diagram: software decoder

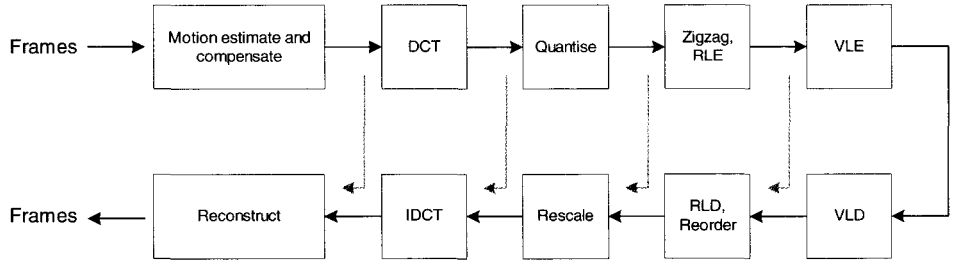

Figure 13.11 Encoder and decoder interoperating points

operations. An encoder may choose to carry out motion estimation and compensation for the entire frame before carrying out the block-level operations (DCT, quantise, etc.), instead of coding the blocks immediately after motion compensating the macroblock. Similarly, an encoder or decoder may choose to reconstruct each motion-compensated macroblock either immediately after decoding the residual blocks or after the entire residual frame has been decoded.

The following principles can help to decide the structure of the software program:

1. Minimise interdependencies between coding functions in order to keep the software modular.
2. Minimise data copying between functions (since each copy adds computation).
3. Minimise function-calling overheads. This may involve combining functions, leading to less modular code.
4. Minimise latency. Coding and transmitting each macroblock immediately after motion estimation and compensation can reduce latency. The coded data may be transmitted immediately, rather than waiting until the entire frame has been motion-compensated before coding and transmitting the residual data.

13.3.3 Designing the Functional Blocks

A good approach is to start with the simplest possible implementation of each algorithm (for example, the basic form of the DCT shown in Equation 7.1) in order to develop a functional CODEC as quickly as possible. The first 'pass' of the design will result in a working, but very inefficient, CODEC and the performance can then be improved by replacing the basic algorithms with 'fast' algorithms. The first version of the design may be used as a 'benchmark' to ensure that later, faster versions still meet the requirements of the coding standard.

Designing the encoder and decoder in tandem and taking advantage of 'natural' points at which the two designs can interwork may further ease the design process. Figure 13.11 shows some examples of interworking points. For example, the residual frame produced after encoder motion compensation may be 'fed' to the decoder motion reconstruction function and the decoder output frame should match the encoder input frame.

13.3.4 Improving Performance

Once a basic working CODEC has been developed, the aim is to improve the performance in order to meet the design goals discussed above. This may involve some or all of the following steps:

1. Carry out software profiling to measure the performance of individual functions. This is normally carried out automatically by the compiler inserting timing code into the software and measuring the amount of time spent within each function. This process identifies 'critical' functions, i.e. those that take the most execution time.
2. Replace critical functions with 'fast' algorithms. Typically, functions such as motion estimation, DCT and variable-length coding are computationally critical. The choice of 'fast' algorithm depends on the platform and to some extent the design structure of the CODEC. It is often good practice to compare several alternative algorithms and to choose the best.
3. Unroll loops. See Section 6.8.1 for an example of how a motion estimation function may be redesigned to reduce the overhead due to incrementing a loop counter.
4. Reduce data interdependencies. Many processors have the ability to execute multiple operations in parallel (e.g. using SIMD/VLIW instructions); however, this is only possible if the operations are working on independent data.
5. Consider combining functions to reduce function calling overheads and data copies. For example, a decoder carries out inverse zigzag ordering of a block followed by inverse quantisation. Each operation involves a movement of data from one array into another, together with the overhead of calling and returning from a function. By combining the two functions, data movement and function calling overhead is reduced.
6. For computationally critical operations (such as motion estimation), consider using platform-specific optimisations such as inline assembler code, compiler directives or platform-specific library functions (such as Intel's image processing library).

Applying some or all of these techniques can dramatically improve performance. However, these approaches can lead to increased design time, increased compiled code size (for example, due to unrolled loops) and complex software code that is difficult to maintain or modify.

Example

An H.263 CODEC was developed for the TriMedia TM1000 platform.[1] After the 'first pass' of the software design process (i.e. without detailed optimisation), the CODEC ran at the unacceptably low rate of 2 CIF frames per second. After reorganising the software (combining functions and removing interdependencies between data), execution speed was increased to 6 CIF frames per second. Applying platform-specific optimisation of critical functions (using the TriMedia VLIW instructions) gave a further increase to 15 CIF frames per second (an acceptable rate for video-conferencing applications).

13.3.5 Testing

In addition to the normal requirements for software testing, the following areas should be checked for a video CODEC design:

- Interworking between encoder and decoder (if both are being developed).
- Performance with a range of video material (including 'live' video if possible), since some 'bugs' may only show up under certain conditions (for example, an incorrectly decoded VLC may only occur occasionally).
- Interworking with third-party encoder(s) and decoder(s). Recent video coding standards have software 'test models' available that are developed alongside the standard and provide a useful reference for interoperability tests.
- Decoder performance under error conditions, such as random bit errors and packet losses.

To aid in debugging, it can be useful to provide a 'trace' mode in which each of the main coding functions records its data to a log file. Without this type of mode, it can be very difficult to identify the cause of a software error (say) by examining the stream of coded bits. A real-time test framework which enables 'live' video from a camera to be coded and decoded in real time using the CODEC under development can be very useful for testing purposes, as can be bit-stream analysis tools (such as 'MPEGTool') that provide statistics about a coded video sequence.

Some examples of efficient software video CODEC implementations have been discussed.[2–7] Opportunities have been examined for parallelising video coding algorithms for multiple-processor platforms,[5–7] and a method has been described for splitting a CODEC implementation between dedicated hardware and software.[8] In the next section we will discuss approaches to designing dedicated VLSI video CODECs.

13.4 DESIGN OF A HARDWARE CODEC

The design process for a dedicated hardware implementation is somewhat different, though many of the design goals are similar to those for a software CODEC.

13.4.1 Design Goals

Design goals for a hardware CODEC may include:

1. Maximise frame rate.
2. Maximise frame size.
3. Maximise peak coded bit rate.
4. Maximise video quality for a given coded bit rate.
5. Minimise latency.
6. Minimise gate count/design 'area', on-chip memory and/or power consumption.

7. Minimise off-chip data transfers ('memory bandwidth') as these can often act as a performance 'bottleneck' for a hardware design.
8. Provide a flexible interface to the host system (very often a processor running higher-level application software).

In a hardware design, trade-offs occur between the first four goals (maximise frame rate/frame size/peak bit rate/quality) and numbers (6) and (7) above (minimise gate count/power consumption and memory bandwidth). As discussed in Chapters 6–8, there are many alternative architectures for the key coding functions such as motion estimation, DCT and variable-length coding, but higher performance often requires an increased gate count. An important constraint is the *cycle budget* for each coded macroblock. This can be calculated based on the target frame rate and frame size and the clock speed of the chosen platform.

Example

Target frame size: QCIF (99 macroblocks per frame, H.263/MPEG-4 coding)
Target frame rate: 30 frames per second
Clock speed: 20 MHz

Macroblocks per second: $99 \times 30 = 2970$
Clock cycles per macroblock: $20 \times 10^6/2970 = 6374$

This means that all macroblock operations must be completed within 6374 clock cycles. If the various operations (motion estimation, compensation, DCT, etc.) are carried out serially then the sum total for all operations must not exceed this figure; if the operations are pipelined (see below) then any one operation must not take more than 6374 cycles.

13.4.2 Specification and Partitioning

The same sequence of operations listed in Figures 13.9 and 13.10 need to be carried out by a hardware CODEC. Figure 13.12 shows an example of a decoder that uses a 'common bus'

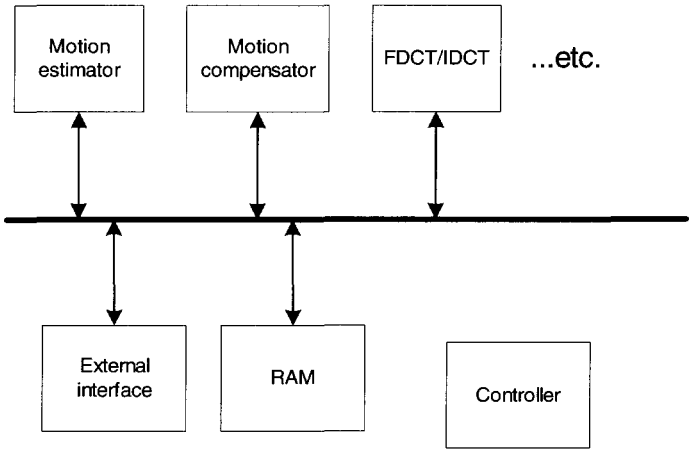

Figure 13.12 Common bus architecture

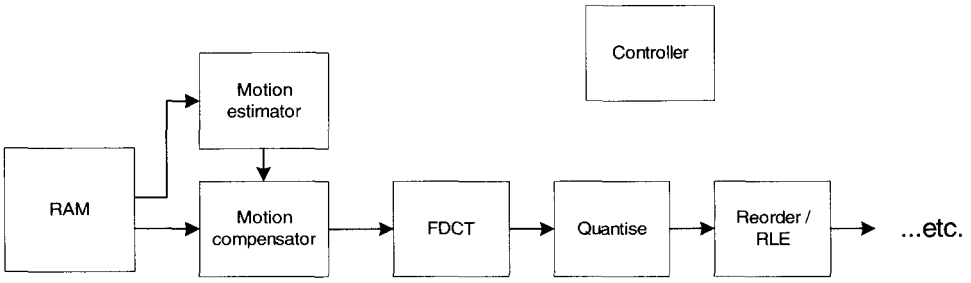

Figure 13.13 Pipelined architecture

architecture. This type of architecture may be flexible and adaptable but the performance may be constrained by data transfer over the bus and scheduling of the individual processing units. A fully pipelined architecture such as the example in Figure 13.13 has the potential to give high performance due to pipelined execution by the separate functional units. However, this type of architecture may require significant redesign in order to support a different coding standard or a new optional coding mode.

A further consideration for a hardware design is the partitioning between the dedicated hardware and the 'host' processor. A 'co-processor' architecture such as that described in the DirectX VA framework (see Chapter 13) implies close interworking between the host and the hardware on a macroblock-by-macroblock basis. An alternative approach is to move more operations into hardware, for example by allowing the hardware to process a complete frame of video independently of the host.

13.4.3 Designing the Functional Blocks

The choice of design for each functional block depends on the design goals (e.g. low area and/or power consumption vs. high performance) and to a certain extent on the choice of architecture. A 'common bus'-type architecture may lend itself to the reuse of certain 'expensive' processing elements. Basic operations such as multiplication may be reused by several functional blocks (e.g. DCT and quantise). With the 'pipelined' type of architecture, individual modules do not usually share processing elements and the aim is to implement each function as efficiently as possible, for example using slower, more compact distributed designs such as the distributed arithmetic architecture described in Chapter 7.

In general, regular, modular designs are preferable both for ease of design and efficient implementation on the target platform. For example, a motion estimation algorithm that maps to a regular hardware design (e.g. hierarchical search) may be preferable to less regular algorithms such as nearest-neighbours search (see Chapter 6).

13.4.4 Testing

Testing and verification of a hardware CODEC can be a complicated process, particularly since it may be difficult to test with 'real' video inputs until a hardware prototype is available. It may be useful to develop a software model that matches the hardware design to

assist in generating test vectors and checking the results. A real-time test bench, where a hardware design is implemented on a reprogrammable FPGA in conjunction with a host system and video capture/display capabilities, can support testing with a range of real video sequences.

VLSI video CODEC design approaches and examples have been reviewed[9–12] and two specific design case studies presented.[11,12]

13.5 SUMMARY

The design of a video CODEC depends on the target platform, the transmission environment and the user requirements. However, there are some common goals and good design practices that may be useful for a range of designs. Interfacing to a video CODEC is an important issue, because of the need to efficiently handle a high bandwidth of video data in real time and because flexible control of the CODEC can make a significant difference to performance. There are many options for partitioning the design into functional blocks and the choice of partition will affect the performance and modularity of the system. A large number of alternative algorithms and designs exist for each of the main functions in a video CODEC. A good design approach is to use simple algorithms where possible and to replace these with more complex, optimised algorithms in performance-critical areas of the design. Comprehensive testing with a range of video material and operating parameters is essential to ensure that all modes of CODEC operation are working correctly.

REFERENCES

1. I. Richardson, K. Kipperman and G. Smith, 'Video coding using digital signal processors', DSP World Fall Conference, Orlando, 1999.
2. J. McVeigh et al., 'A software-based real-time MPEG-2 video encoder', *IEEE Trans. CSVT*, **10**(7), October 2000.
3. S. Akramullah, I. Ahmad and M. Liou, 'Optimization of H.263 video encoding using a single processor computer', *IEEE Trans. CSVT*, **11**(8), August 2001.
4. B. Erol, F. Kossentini and H. Alnuweiri, 'Efficient coding and mapping algorithms for software-only real-time video coding at low bit rates', *IEEE Trans. CSVT*, **10**(6), September 2000.
5. N. Yung and K. Leung, 'Spatial and temporal data parallelization of the H.261 video coding algorithm', *IEEE Trans. CSVT*, **11**(1), January 2001.
6. K. Leung, N. Yung and P. Cheung, 'Parallelization methodology for video coding–an implementation on the TMS320C80', *IEEE Trans. CSVT*, **10**(8), December 2000.
7. A. Hamosfakidis, Y. Paker and J. Cosmas, 'A study of concurrency in MPEG-4 video encoder', *Proceedings of IEEE Multimedia Systems'98*, Austin, Texas, July 1998.
8. S. D. Kim, S. K. Jang, J. Lee, J. B. Ra, J. S. Kim, U. Joung, G. Y. Choi and J. D. Kim, 'Efficient hardware–software co-implementation of H.263 video CODEC', *Proc. IEEE Workshop on Multimedia Signal Processing*, pp. 305–310, Redondo Beach, Calif., 7–9 December 1998.
9. P. Pirsch, N. Demassieux and W. Gehrke, 'VLSI architectures for video compression–a survey', *Proceedings of the IEEE*, **83**(2), February 1995.
10. P. Pirsch and H. -J. Stolberg, 'VLSI implementations of image and video multimedia processing systems', *IEEE Transactions on Circuits and Systems for Video Technology*, **8**(7), November 1998, pp. 878–891.

11. P. Pirsch and H. -J. Stolberg, 'VLSI implementations of image and video multimedia processing systems', *IEEE Transactions on Circuits and Systems for Video Technology*, **8**(7), November 1998, pp. 878–891.
12. A. Y. Wu, K. J. R. Liu, A. Raghupathy and S. C. Liu, *System Architecture of a Massively Parallel Programmable Video Co-Processor*, Technical Report ISR TR 95-34, University of Maryland, 1995.