Preface

This master thesis was written during spring 2005, at The Department of Computer and Information Science, NTNU. The purpose of the project was to make an in-depth study in the field of computer design and architecture, with focus on open hardware, and to evaluate different methods for realizing parts of software as hardware acceleration modules. The project included literature studies as well as practical work in the field of hardware design.

Acknowledgments

I would like to thank my supervisor at Nordic Semiconductor, Torstein Heggebø, for the guidance, inpiration and help he has offered me.

Morten Haugseggen

Trondheim, June 15, 2005

Contents

1 Introduction **1**
- 1.1 Introduction to MPEG-2 . . . 2
- 1.2 MPEG-2 today . . . 3
- 1.3 Overview of the MPEG-2 decompression algorithm . . . 8
- 1.4 Hardware software partitioning . . . 11

2 The MPEG-2 decoder **12**
- 2.1 The decoder itself . . . 12
- 2.2 Software and hardware requirements . . . 12
- 2.3 Compiling and installing the library . . . 13
  - 2.3.1 Running the test program . . . 13
- 2.4 More details on the decoding process . . . 16
  - 2.4.1 VLC/VLD . . . 16
  - 2.4.2 Quantisation . . . 17
  - 2.4.3 DCT/IDCT . . . 18
  - 2.4.4 MC . . . 20

3 Three approaces to specialized hardware **25**
- 3.1 Expanding the instruction set . . . 25
- 3.2 IO unit . . . 26
- 3.3 DMA unit . . . 28

4 Selecting parts for optimization **31**
- 4.1 Requirements for optimizable parts . . . 31
- 4.2 Profiling of the MPEG-2 decoder . . . 32
- 4.3 Finding the bottlenecks . . . 34
- 4.4 Final choice . . . 36
  - 4.4.1 Adding new instructions . . . 37
  - 4.4.2 Adding IO/DMA module . . . 40

5 Adding new instructions to the architecture **44**
- 5.1 Hardware and software . . . 44
- 5.2 Modifying the processor core . . . 44

5.2.1 The internals of the module 45
5.2.2 Interfacing with the processor core 47
5.3 Modifying the compiler and software 52
5.4 Simulation and testing 59

6 Results **64**
6.1 Added instructions 64
6.2 Simulation of IO unit 67
6.3 Simulation of DMA unit 72
6.4 Comparison 75

7 Discussion **77**

8 Further work **79**

9 Conclusion **80**

A gprof log files **82**
A.1 Initial run - no modifications 82

B Source code **92**
B.1 idct.c 92
B.2 Assembly output of the butterfly 96
B.3 hello-uart 98

C HDL files **100**
C.1 Butterfly implementation 100

Bibliography **102**

List of Figures

1.1 Pixelation due to low resolution. Picture from the motion picture ”The Incredibles”. 6
1.2 Artifacts due to high levels of compression. Picture from the motion picture ”The Incredibles”. 7
1.3 In ”normal” frames it is nearly impossible to find either coding artifacts or pixelation due to low resolution. Picture from the motion picture ”Thirteen days”. 8
1.4 MPEG-2 datastream structure. 9
1.5 An example of the slice structure. 10
1.6 MPEG-2 data flow during decoding. 11

2.1 A screenshot from a standard DVD decoder. Picture from the motion picture Toy story 2. 14
2.2 A screenshot directly from the libmpeg2 MPEG-2 decoder. Picture from the motion picture Toy Story 2. 15
2.3 The variance distribution of DCT coefficients 19
2.4 The process of doing motion compensation. 22
2.5 Two frames from a movie, showing a moving ball. 23
2.6 The actual difference between the two frames. 24
2.7 A closeup of the moving ball. 24

4.1 Part of the logfile from the Linux utility ”gprof”. 33
4.2 Distribution of execution time. 36
4.3 The C code for doing column computations. 38
4.4 The butterfly part of the IDCT. 38
4.5 The logfile from gprof after a run without the butterfly. 39
4.6 The C kode implementing the butterfly only. 40
4.7 The main part of the assembly code from the butterfly. 41
4.8 Main part of the assembly code from the butterfly. But this time in an optimized version. 42

5.1 The groups of bits that make up each instruction. 46
5.2 Schematic of the internals of the butterfly module. 48
5.3 The Verilog description of the butterfly module. 49

5.4 How to use the asm() directive. 54
5.5 Example of inline assembly. 54
5.6 The resulting assembly code from the inline assembly example. . . . 55
5.7 The available custom instructions for the assembler. 55
5.8 The definitions for the new butterfly instructions. 55
5.9 The ”main” function of the hello-uart program used for testing. 57
5.10 The new ”idct.c” with assembly instructions. 58
5.11 Installation of the tools needed for testing. 61
5.12 Output from the JTAG utility. 62
5.13 Simulation of an early version of the butterfly module on the Modelsim simulator. 63
5.14 Output from the new test utility. 63

6.1 Assembly code for the NOP loop used for testing. 66
6.2 A diagram showing the execution time in percent of total execution time after implemeting the new instruction. 67
6.3 Assembly code for the NOP loop used for testing. 69
6.4 Logfile from gprof showing a simulation of the IO unit where IDCT has been replaced by NOPs. 70
6.5 A diagram showing the execution time in percent of total execution time for the simulation of the IO unit. 71
6.6 Logfile from gprof showing a simulation of the DMA unit where IDCT has been replaced by NOPs. 74
6.7 A diagram showing the execution time in percent of total execution time for the simulation of the DMA unit. 75

List of Tables

1.1 Video File Comparison (from www.videohelp.com) 3

2.1 Properties of the MPEG-2 stream used to test the decoder. 16
2.2 The connection between frequency, fixed-length codewords and variable-length codewords. 17

6.1 The hardware resources used by the instructions and the processor. . . 65
6.2 Properties of the three implementation methods I have described. . . . 76

Abstract

The MPEG-2 codec (coder/decoder) has been one of the most important factors for bringing high-quality digital video to the masses. The most successful medias have been DVDs and satellite broadcasts which, much thanks to MPEG-2, offers great quality when there is limited bandwidth and/or storage. Even so, the advances in display technology is about to render the MPEG-2 codec obsolete due to the fact that displays are getting higher resolution and better image quality, which makes the flaws of the codec more visible. However, the portable revolution has only begun, and MPEG-2 is a very well suited candidate for delivering high-quality video on portable devices.

”libmpeg2” is an open source MPEG-2 decoder. It is currently beeing used in a number of different open source media players. It is written in C, with optimization for selected processors. The fact that it is well tested, optimized and written in C makes it ideal for porting to an embedded system. In this case the system will consist of an open hardware processor called OpenRISC 1200 (OR1200). This is a processor targetting embedded devices and is well suited for this purpose since the Verilog code for the processor is available for download.

To run MPEG-2 decoding real-time without hardware acceleration requires the processor to run at about 350 MHz. For portable devices this is to high. Therefor three different methods for hardware acceleration has been explored in this report:

- Adding new instructions.
- Adding an IO unit.
- Adding a DMA unit.

By profiling libmpeg2, the critical parts of the decoder has been found (the parts that uses the most time). And by exchanging parts of the decoder, simulations have been run to see how much can be gained from implementing different parts in hardware. This report has mainly focused on the IDCT since this is the most time consuming part of the decoder. The new instructions have only implemented a small subset of the IDCT, while the IO and DMA unit have implemented the whole IDCT.

There has been gain in the decode speed for all of the three methods. The DMA unit is the one that offers the greates speedup but it also costs most when it comes to increase area and implementation effort. However, it and does not touch the internals of the processor, which is a great advantage. But basically the IO and DMA units are very similar. The new instructions are simpler to implement but makes modifications to the processor core. They offer lesser speedup, compared to the IO and DMA unit, but the increased area is significantly lower. Only the new instructions have actually been implemented.

Chapter 1

Introduction

My earliest memories related to MPEG-2 must have been from the early '90s. I remember reading about a new way of storing digital information on a disc containg two different layers. I am of course talking about the DVD, the medium where MPEG-2 has had its perhaps greatest success. A few years later, the DVD players were introduced and new DVD titles popped up at an ever increasing rate. After a while DVD players came to the PC platform, first in the form of hardware accelerated players, then in the form of pure software players. These software players often required a x86 processor to run at 350-450 MHz or above for decoding the film without skipping.

In this report I am going to use the MPEG-2 codec (coder/decoder) and the open hardware processor OpenRISC 1200, and see what advantages this combination can give. Since it is unrealistic to run a MPEG-2 decoder in software on a processor running much slower than 300 MHz it is necessary to use some form of hardware acceleration. To bring the processor speed down to 100 or perhaps 50 MHz would have a tremendous impact on battery usage in portable devices. This can be done by implementing different types of hardware that can do specific parts of the decoding more efficiently. These methods can then be evaluated against each other when it comes to area and power usage, and speedup of the decoding process. So this is not an attemt of re-inventing or improving the MPEG-2 decoder, it is a way of making the decoding process run as effectively as possible on the OR1200. Up until now, media players would have been plugged into the wall outlet in order to get enough power. This is now changing and people are beginning to want portable, often wireless products that uses a minimal amount of power, usually running on batteries. Therefor, this must not be seen as a MPEG-2 report at the wrong time, rather an OpenRISC report at the right time, where the possibilities of this processor are explored for use in new areas.

I will start the report by giving an introduction to MPEG-2 decoding in general, then link this information to the specific MPEG-2 decoder I am going to use. Then I will introduce different ways of enhancing the capabilities of the processor. These methods will then be used to make the MPEG-2 decoder I will be using, run faster.

1.1 Introduction to MPEG-2

MPEG-2 is a standard for compression and decompression of video streams. [ISO95] defines the standard for both encoding and decoding of MPEG-2 video streams. The idea behind MPEG-2 is that it should be used where limited bandwidth or storage is available and high quality video is required. Examples known to most people today would be digital sattelite broadcast of television and the Digital Versatile Disc or Digital Video Disc (DVD,) as it was called in the beginning. The DVD has a limitation in storage of 4.7 GB for single-layer discs and 9.4 GB for dual-layer discs. Digital sattelite broadcasts have a limitation in bandwidth of 18 Mb/s. A very simple calculation can easily show the need for such video compression and what is required from it: The resolution of the frames on a DVD is typically 720x576 pixels. The framerate, the number of frames per second, is 25 (according to [tvf] the european PAL system uses a framrate of 25, the american NTSC system uses a framerate of 30). When using an RGB color coding scheme with 24 bits per pixel, a film lasting one hour would require more than 100 GB of storage or 252 Mbit/s of bandwidth. This shows that broadcasting and storing video in high quality digital form would require a large amount of both storage and bandwidth without some form of compression. Without compression standards such as MPEG-2, distribution of digital video would be not be an easy task when using today's technology.

[ISO95] only defines the video-part of the MPEG-2 decoder. Dedicated standards have been written to define how the audio is beeing *synchronized* with the video. This means that a number of different audio codecs can be used in combination with the video as long as they are capable of synchronizing with the film.

As seen in table 1.1 the MPEG-2 codec is beeing used in a number of different products, including SVCD, DVD and HDTV. It offers very high quality video, as shown in figure 1.3, compared to many other codecs. It also satisfies both the bandwidth and storage requirements set by today's technology.

Format	VCD	SVCD	DVD	HDDVD HDTV (WMVHD)	AVI DivX XviD WMV	MOV Quick-Time	RM Real-Media	AVI DV
Resolution NTSC/PAL	352x240 352x288	480x480 480x576	720x480[1] 720x576[1]	1440x1080[1] 1280x720[1]	640x480[1] 640x480[1]	640x480[1] 640x480[1]	320x240[1] 320x240[1]	720x480 720x576
Video Compression	MPEG1	MPEG2	MPEG2, MPEG1	MPEG2 (WMV-MPEG4)	MPEG4	Sorenson, Cinepak, MPEG4...	RM	DV
Video bitrate	1150 kbps	∼2000 kbps	∼5000 kbps	∼20Mbps (∼8Mbps)	∼1000 kbps	∼1000 kbps	∼350 kbps	25 Mbps
Audio Compression	MP1	MP1	MP1, MP2, AC3, DTS, PCM	MP1, MP2 AC3, DTS, PCM	MP3, WMA, OGG, AAC, AC3	QDesign Music, MP3...	RM	DV
Audio bitrate	224 kbps	∼224 kbps	∼448 kbps	∼448 kbps	∼128 kbps	∼128 kbps	∼64 kbps	∼1500 kbps
Size/min	10 MB/min	10-20 MB/min	30-70 MB/min	∼150 MB/min	4-10 MB/min	4-20 MB/min	2-5 MB/min	216 MB/min
Min/74 min CD	74min	35-60min	10-20min	∼4min	60-180min	30-180 min	120-300 min	3min
Hours/DVD	N/A	N/A	1-2hrs (2-5hrs[2])	∼30min (∼1hrs)	7-18hrs	3-18hrs	14-35hrs	20min
Hours/ DLDVD[4]	N/A	N/A	2-4hrs (5-9hrs[2])	∼55min (∼2hrs)	13-30hrs	6-30hrs	25-65hrs	37min
DVD Player Compability	Great	Good	Excellent	None	Few	None	None	None
Computer CPU Usage[5]	Low	High	Very high	Super high	Very high	High	Low	High
Quality	Good	Great[3]	Excellent[3]	Superb[3]	Great[3]	Great[3]	Decent[3]	Excellent

Table 1.1: Video File Comparison (from www.videohelp.com)

1.2 MPEG-2 today

Today the MPEG-2 codec is widely used in a number of different applications. In [ISO95] a number of different usages are outlined. These are areas where the authors of the standard thought the codec would have influence. I will recite a few of those that have become the most important here:

- Broadcasting Satellite Service (to the home)
- Cable TV Distribution on optical networks, copper, etc.

[1]Approximately resolution, it can be higher or lower.

[2]DVD with lower video quality, similar to VCD/SVCD video quality.

[3]The video quality depends on the bitrate and the video resolution, higher bitrate and higher resolution generally means better video quality but bigger file size.

[4]Dual Layer DVD

[5]With continously increasing processing powers, the strain on the processor from running these decoders is decreasing rapidly

- Electronic Cinema
- Home Television Theatre
- Interpersonal Communications (videoconferencing, videophone, etc.)

The most successful and known application of the MPEG-2 codec is in DVD players and through satellite and cable broadcast of television. In both these cases the MPEG-2 codec delivers both high quality and resolution even when the bandwidth or storage is limited. An area where the MPEG-2 codec has not been so successful is video-conferencing/videophone. The main reason is perhaps that the MPEG-2 coder is very processor intensive. Other encoders may offer only slightly worse quality while beeing much less processor intensive. Since any delay is intolerable in real-time applications such as videophone, the MPEG-2 encoder has often been too processor intensive to run in software and therefor been replaced by other codecs. Although there exists DVD recorders that encodes MPEG-2 video in real-time, these encoders are often implemented in specialized hardware. A pure software based MPEG-2 encoder remains too processor intensive compared to other encoders to be a good choice in the case of video conference and other real-time applications.

But besides its failures in some areas, the codec has been a huge success. In many ways the MPEG-2 codec is one of the main reasons why it is possible to enjoy high quality digital video through DVD and sattelite/cable broadcast. It has perhaps been the most successful codec, along with MP3 audio codec, for bringing high quality digital multimedia to the public.

Although the MPEG-2 codec delivers video with very high quality (compared to most other codecs), the introduction of HDTV and other high-resolution equiment has shown that the quality is not good enough. The problem is, however, two-sided:

- The resolution used on DVDs are too low for modern display technology.
- Encoding artifacts are becoming visible during high-motion scenes.

As seen in table 1.1, standard DVDs are using a resolution of 720x576 pixels (PAL format). According to [tvf] a TV supporting the PAL format is using the same resolution. This basically means that DVDs are encoded using a resolution that generates a very good result when viewed on a standard TV. With the introduction of HDTV the resolution is no longer 720x576 but up to 1920x1080. This means that pixelation as shown in figure 1.1 are becoming visible (espesially note the area inside the circle in the enlarged image). This problem can easily be solved by increasing the resolution but this requires higher bandwidth or more storage if the same level of compression is used. In the case of a DVD, the storage is limited to 9.4 GB which means that the only solution is to increase the compression level to make an entire film fit on one DVD. This in turn would mean that compression artifacts, as seen in figure 1.2, are becoming more visible and not limited to high-motion scenes only. In other words the MPEG-2

codec is rapidly becoming outdated due to advances in display technology if the same storage technology, the DVD, are to be used in the future.

This does not mean the end of the MPEG-2 codec. It only means that it probably will be replaces by new and more advanced codecs in situations where high-resolution displays are available. But I belive MPEG-2 will still be the preferred codec for some years to come. On the other hand we are only beginning to see a portable revolution that begun with MP3 players. These players are usually physically small and has limitations when it comes to battery life, storage (if it has a harddrive), display size and resoltion, and processing powers. This means that audio or video stored on such a device has to use some kind of compression and the decoders have to place a minimal load on the hardware so that power consumption is minimized. In these cases the MPEG-2 decoder would be a good choice since it offeres high quality video while beeing less processor intensive than DivX, for instance (see table 1.1). When showing ”normal” frames, like the one in figure 1.3 on a small display with low resolution, it is nearly impossible to find neither coding artifacts nor pixelation.

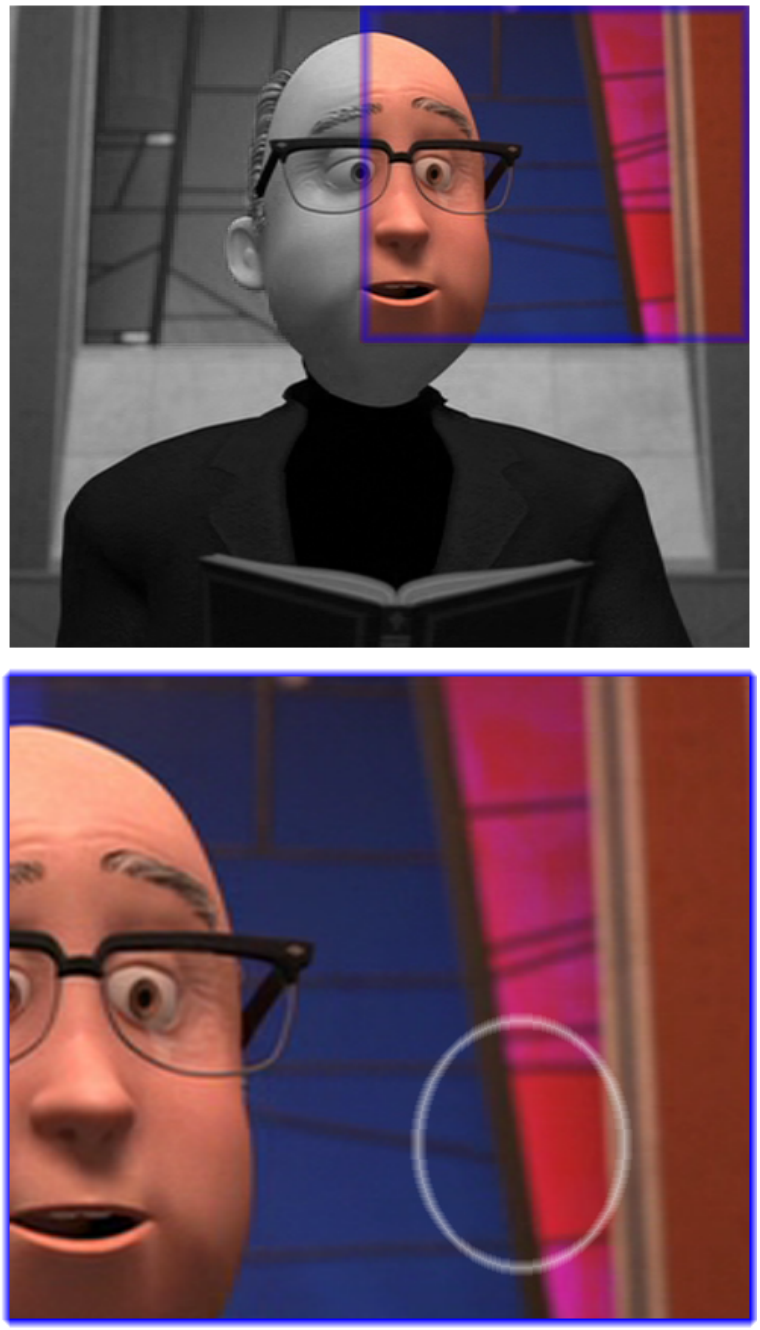

Figure 1.1: Pixelation due to low resolution. Picture from the motion picture "The Incredibles".

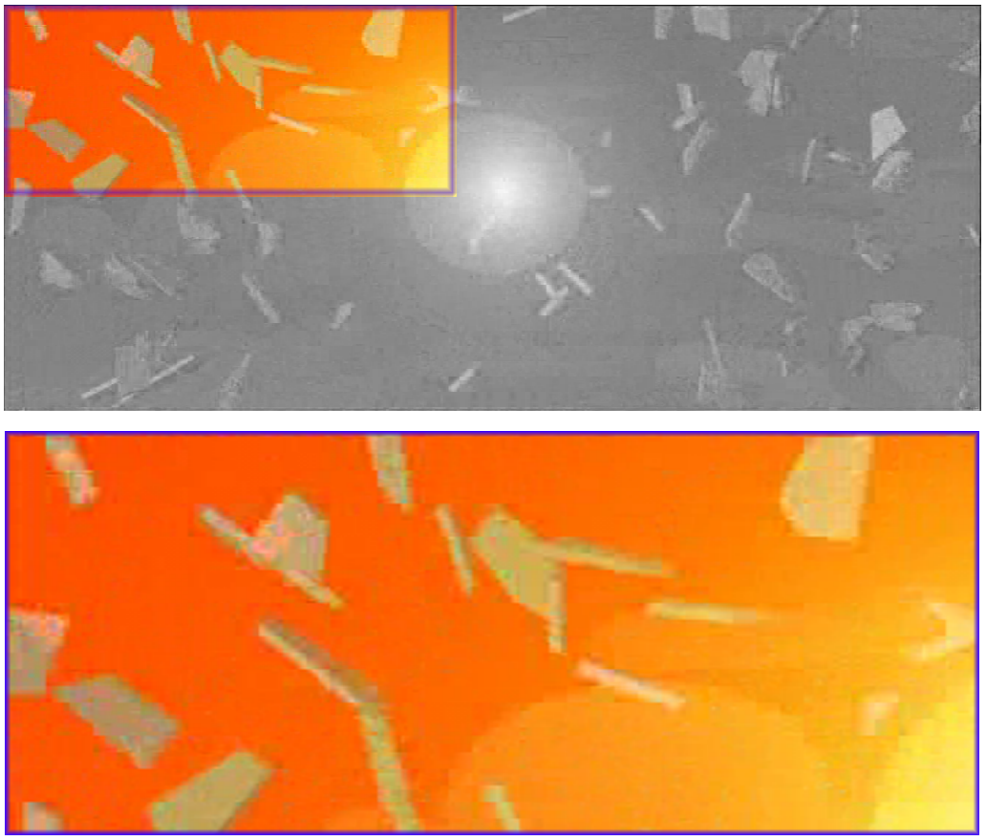

Figure 1.2: Artifacts due to high levels of compression. Picture from the motion picture ”The Incredibles”.

Figure 1.3: In ”normal” frames it is nearly impossible to find either coding artifacts or pixelation due to low resolution. Picture from the motion picture ”Thirteen days”.

1.3 Overview of the MPEG-2 decompression algorithm

In the rest of this paper I will mainly focus on the decompression algorithm, the decoder. Compared to the encoder, the decoder is less complex and less processor intensive. It also have some inherent properties that makes it more suitable to implement in hardware compared to the encoder. These properties will be further discussed in section 4.1.

The MPEG-2 stream produces by the encoder is structured as shown in figure 1.4:

- At the top level the MPEG-2 stream is organized into groups of pictures (GOPs).
- Each GOP consists of a series of pictures. Each picture can be one of three different types:

  Intra-coded (I) picture - This picture is coded using information only from itself. These pictures are intended to assist random access into the sequence.

  Predictive-coded (P) picture - This is a picture which is coded using motion compensated prediction from a past reference frame or past reference field.

  Bidirectionally predictive-coded (B) picture - This is a picture which is coded using motion compensated prediction from a past and/or future reference frame(s).

- Each picture is divided into slices. The slices do not have to cover the entire picture as shown in figure 1.5. This means that no information is encoded for the areas outside the slices.

- Each slice can contain an arbitrary number (greater than zero) of macroblocks.
- Each macroblock contains a various number of 8x8 blocks of data. In figure 1.4 the following items are shown:

  Luminance (Y) - four 8x8 blocks of data.

  Chrominance (Cb) - one 8x8 block of data.

  Chrominance (Cr) - one 8x8 block of data.
- The number of chrominance (Cr and Cb) blocks per macroblock may vary depending on the YUV coding scheme.

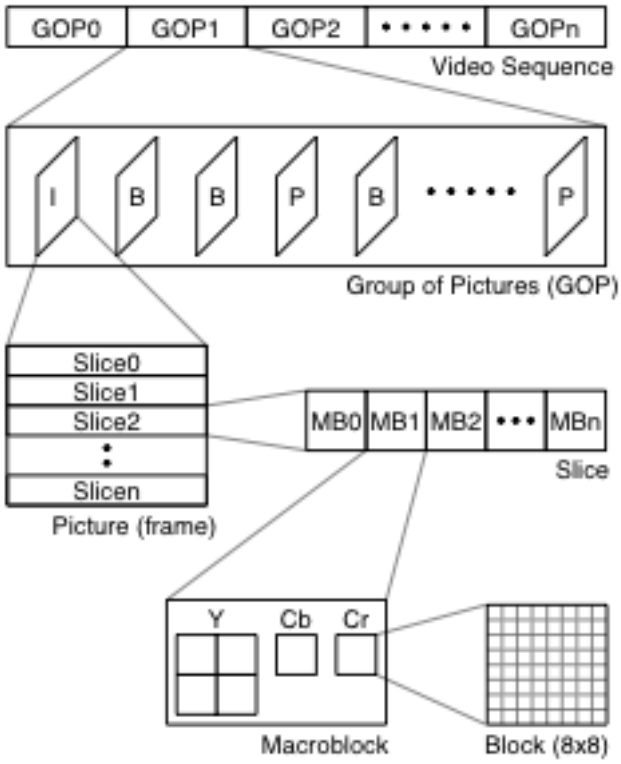

Figure 1.4: MPEG-2 datastream structure.

Figure 1.6 shows the different stages of the MPEG-2 decoding:

- **Variable Lenght Decoding (VLD):** Decoding of a Huffman style coding. Values used more often are coded using fewer bytes than lesser used values.
- **Inverse Scan:** This is the method used to convert the one-dimentional array output from the VLD into a two-dimentional matrix used in the rest of the process.

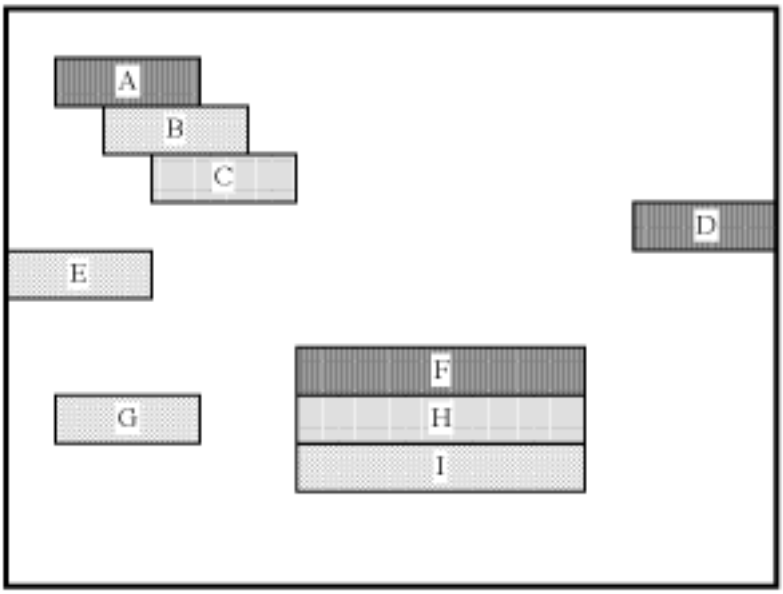

Figure 1.5: An example of the slice structure.

- **Inverse Quantisation:** Quantisation is a method used to reduce the number of possible values of a quantity. The inverse quantisation brings the values back to approximately its original values.
- **Inverse DCT (IDCT):** Transformes the DCT coefficients back to a pixel representation of the frame.
- **Motion Compensation:** Work on 16x16 blocks of pixels, in contrast to the steps mentioned before that work on 8x8 blocks. Use motion vectors to calculate block movement based on the frames stores in frame-store memory.
- **Frame-store Memory:** Memory where entire frames used as base for predicting other frames are stored.

More details on some of the stages mentioned here and shown in figure 1.6 will be given in section 2.4. For more information on topics not mentioned in section 2.4 see [ISO95] or [Wat94] for a more extensive explanation.

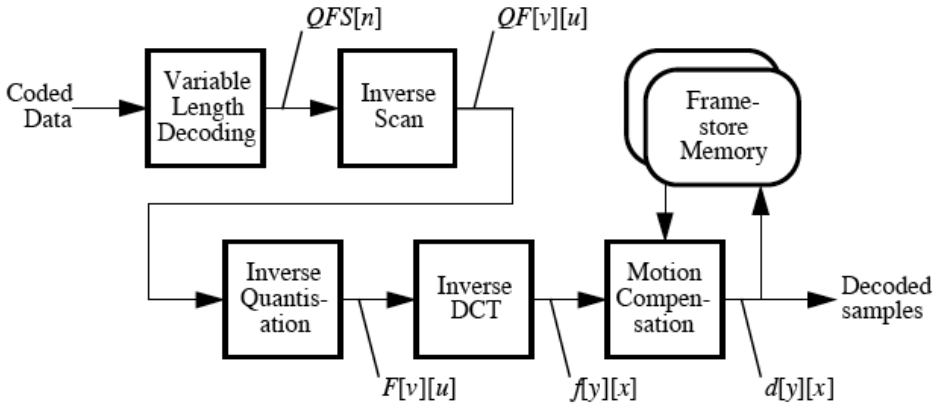

Figure 1.6: MPEG-2 data flow during decoding.

1.4 Hardware software partitioning

Some algorithms and programs are parallel in nature while others are not, which basically means they are serial. However there are no black and white and most programs lie somewhere in between been fully parallel or fully serial, meaning that most programs contain some parts that can be run in parallel and others that can only be run in serial. The basic idea behind hardware/software codesign is to develop the software along with the hardware to see which parts can be implemented in hardware and which parts in software. In this report I will use a similar approach. In this case I have a decoder that has already been completed in software. My task is to find parts that can be implemented efficiently in hardware. Since I also have a complete generic processor I have the choice between running the entire decoder on this processor or making new hardware to support specific parts of the decoder. More specificall, this means that I do not have to make any changes to parts of the program that are serial in nature and mainly focus on the ones that are parallel. There are other and perhaps more important factors than parallelism when deciding on which part of the decoder to build in hardware. These factors will be discussed in section 4.1.

Chapter 2

The MPEG-2 decoder

2.1 The decoder itself

The MPEG-2 decoder I have been using is an open source decoding library called libmpeg2. It has been released under GPL, which means it can be used by anyone in any *non-profit* situation. The source code can be downloaded from http://libmpeg2.sourceforge.net. I have been using version 0.4.0b. It has been optimized for a number of different processors, including the Intel Pentium (x86) family, PowerPC (PPC) and Alpha. This basically means that the most frequently used functions in the decoding process has been implemented using the extended instruction sets these processors offer. The MMX (multimedia extensions) instruction set of the Intel and AMD processors is an example of such an instruction set.

It is also worth mentioning that this is the library that is beeing used in media players such as MPlayer, VLC and xine. This should mean that it is well tested, is relatively bug free, has a tolerable performance and follows the MPEG-2 standard.

2.2 Software and hardware requirements

To test the decoder I compiled and installed it on two completely different systems. One system running the following configuration:

- Apple Powerbook G4 1.0 GHz
- 512 MB RAM
- Mac OS X 10.3.8 operating system
- GCC version 3.3
- Autoconf version 2.57
- Automake version 1.6.3

The other system used the following configuration:

- Intel Pentium 4 2.40 GHz
- 512 MB RAM
- Mandrake Linux 10.0
- GCC version 3.2.3
- Autoconf version 2.59
- Automake version 1.6

2.3 Compiling and installing the library

Compilation and installation of the library was very simple. The commands below should be enough to install it on any UNIX-style computer (MacOS X, Linux, BSD and Solaris, for instance):

```
./configure
make all install
```

The first command creates instructions for how to compile the program for the specific platform. The second command do the actual compilation and installation based on the information gathered in the first step. It is also possible to compile and install the library under Microsoft Windows. There exists a Microsoft Visual C++ project in one of the subfolders of the library. To use these files Microsoft Visual C++ has to be installed on the computer and this is not a free software package. It may also be possible to compile the library under Cygwin for Windows but I have not tried these options. I have only compiled and run the program under Linux.

2.3.1 Running the test program

Along with the library comes a couple of test programs to show some of the functionality of the decoder. If the library is compiled as shown in section 2.3 there should be a small program called ”mpeg2dec” in the ”src/” subfolder. By running this program it is possible to decode an MPEG-2 file and get a simple output just to show that the decoder actually works. Figure 2.1 shows an image from a standard DVD decoder, while figure 2.2 shows the image from the libmpeg2 decoder. The images are actually similar but figure 2.1 is shown in RGB colors while figure 2.2 is showing the grayscale representation of the YUV image as it is stored in the MPEG-2 stream. The larges part of the image is the luminance field at the top and the two small ones at the bottom is the chrominance. It is an example of YUV 4:2:0 encoding where the Cr and Cb fields

describe in section 1.3 are $\frac{1}{4}$ of the size of the luminance field. There are a number of switches that can be used when running the test program. These are the two I used when testing the decoder:

- "./mpeg2dec -o pgm [mpeg-2 stream]" for outputting one image in PGM format for each picture in the stream.
- "./mpeg2dec -c -o null [mpeg-2 stream]" for measuring the speed of the system. The "-c" means that the unoptimized C implementation of the code is used and "-o null" means that no output is created.

The properties of the movie I used for testing is shown in table 2.1. During decoding, the framerate (number of frames decoded per second) was $\sim$110 on the Machintosh and $\sim$148 on the Pentium. The results were measured as an average when decoding the whole movie with the "-c -o null" switch used on the test program. These switches tells the program to use the standard C implementation, without acceleration, and to not output any result.

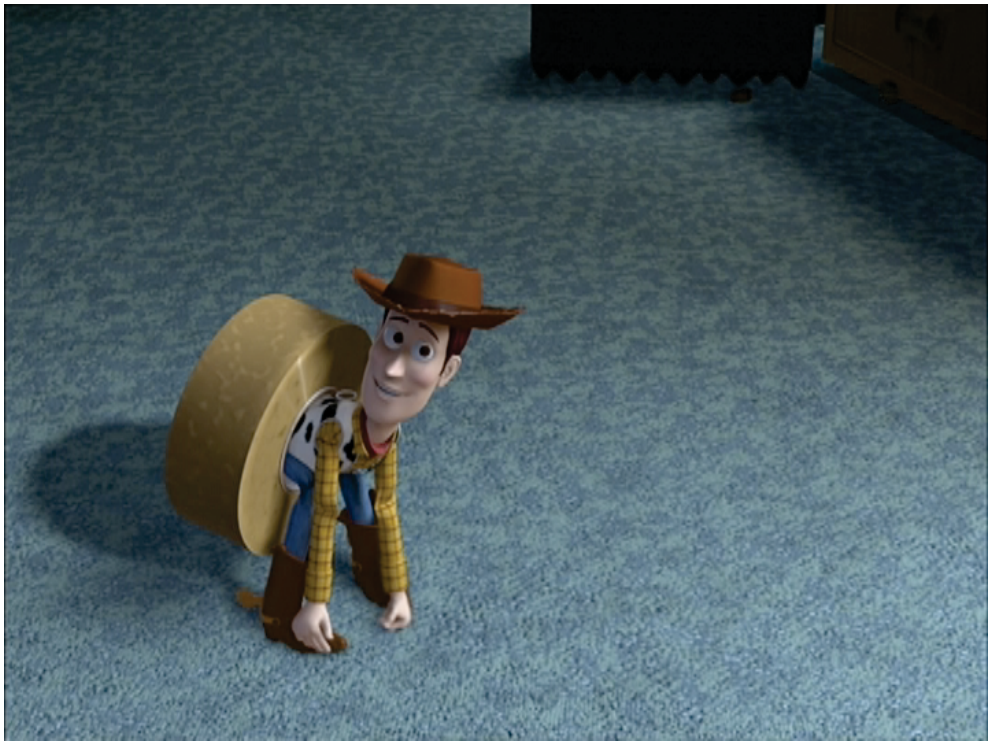

Figure 2.1: A screenshot from a standard DVD decoder. Picture from the motion picture Toy story 2.

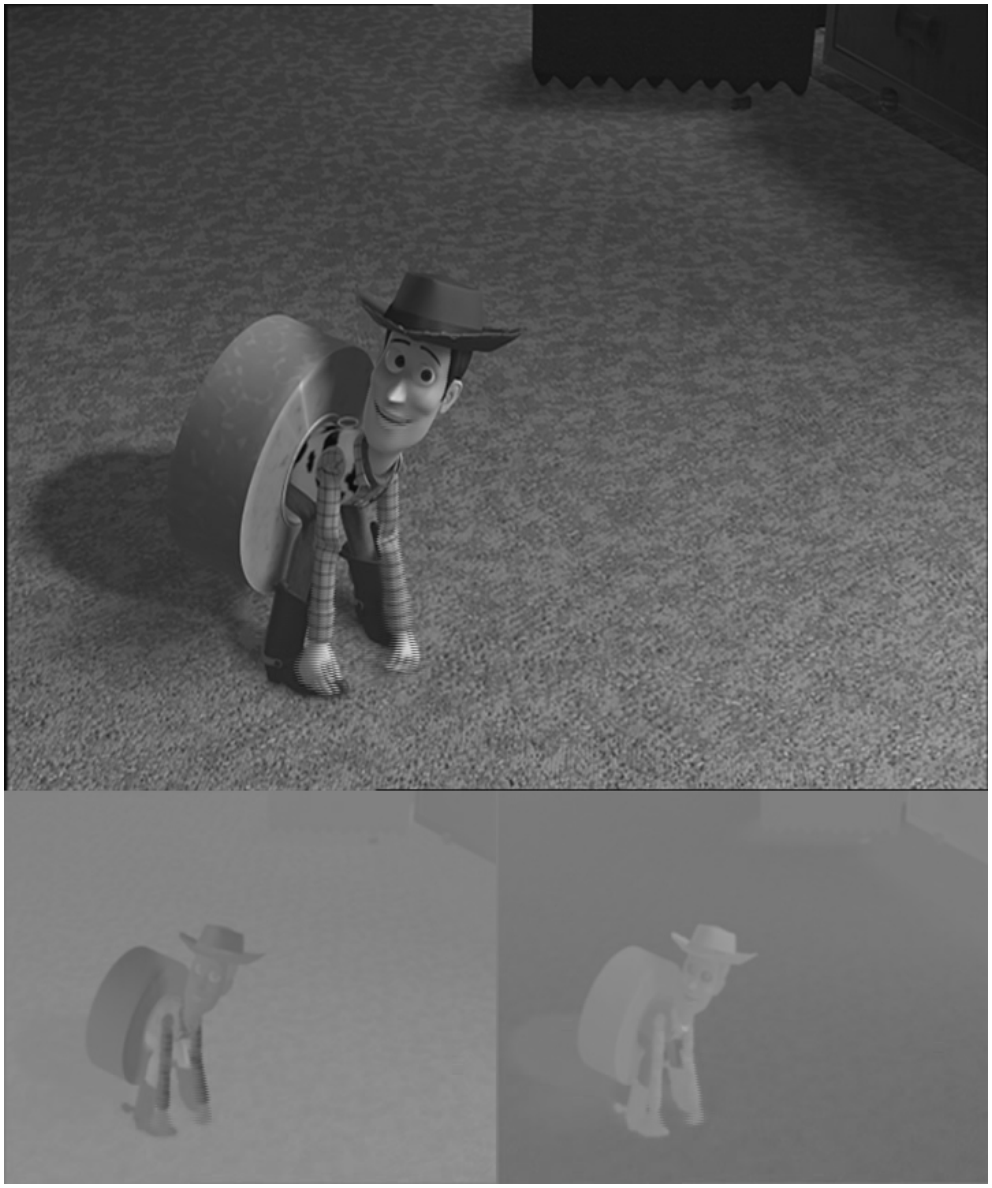

Figure 2.2: A screenshot directly from the libmpeg2 MPEG-2 decoder. Picture from the motion picture Toy Story 2.

Width	720 pixels
Height	576 pixels
Number of frames	8200
Length	328 seconds
Size	254.175.690 bytes (242,4 MB)
Framerate	25 fps
Bitrate	6.199.407 bps (6,2 Mbps)
Compression ratio	~27:1

Table 2.1: Properties of the MPEG-2 stream used to test the decoder.

2.4 More details on the decoding process

In the following section I will give a more thorough description of the decoder. Both of the algorithms behind the implementation, and how these algorithms are used in libmpeg2. Based on the results from [VZL] it seems that there are three parts of the decoder and encoder that are more important, when it comes to execution time, than the others;

- VLC/VLD (Variable Lengh Coding/Decoding)
- DCT/IDCT (Direct Cosine Tranformation/Inverse Discrete Cosine Transformation)
- MC (Motion Compensation)

In addition I will include the quantization process since understanding of this process is essential in understanding the MPEG-2 codec.

2.4.1 VLC/VLD

VLC is essentially a version of Huffman coding where different codewords are assigned different length (in bits) depending on the frequency of occurence in the bitstream. I will not give a detailed description of the coding and decoding of variable length codewords or Huffman codes. [CLR00] gives an introduction to Huffman codes and [Soh02] gives a more thorough description of the VLC/VLD process in a MPEG-2 setting. There is one thing that should be noted about the implementation in the MPEG-2 standard; there is no boundary information for detecting the beginning or the end of the codeword. This complicates matters when it comes to decoding the bit stream.

The VLC is used to compress data. According to [CLR00] a compression level of 20-90% is possible. The compression factor greatly depends on the data to be compressed. As said earlier, the VLC measures the frequency of each occuring value

and then assigns a codeword to each value based on the frequency. An example is shown in figure 2.2 where five different values (a-f) are encoded with fixed-length codewords and variable-length codewords generated from the frequency of the values. Without compression the size of the file will be $100.000 \cdot 3$ bits $= 300.000$ bits. With compression the equivalent file size will be $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4)x1000$ bits $= 224.000$ bits. This is a 25% decrease in file size. The example is from [CLR00]. The VLC itself is a lossless compression in difference to the MPEG-2 compression which is lossless as a whole.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Table 2.2: The connection between frequency, fixed-length codewords and variable-length codewords.

In libmpeg2 the VLD is done in the file called ”slice.c”. This part of the decoder is basically undocumented, but from what I can see, the following functions are responsible for the VLD:

- get_intra_block_B14
- get_intra_block_B15
- get_non_intra_block

In section 4.2 I will take a closer look at time consumption of the different parts of the decoder, and as seen in figure 4.2 the two functions representing the VLD uses about 18% of the total execution time of the decoder.

The VLD is not the most suited for implementation in hardware. As mentioned in [Soh02], exploiting parallelism in the VLD is difficult due to its inherently serial dataflow. However [NVT+02] proposes a method for implementation in hardware. This part experiences a delay of 110 ns and decodes 3.5 codewords per cycle in average.

2.4.2 Quantisation

Besides from the YCrCb encoding used instead of RGB, the quantisation is the only place in the encoding process where information is lost. This makes quantisation a lossy process, compared to both the VLC and DCT which are lossless. Quantisation is really just a way of representing the DCT coefficients as ranges of values instead of exact values. The reason for doing this can be clearly seen in figure 2.3. Most of the coefficients are zero or very close to zero and have only minor variation between

frames. By not including information about the values that are zero or very close to zero, the amount of data involved in representing one block can be significantly reduced. The non-zero values are the divided into ranges of values. For instace $2.5 \approx 5$, $7 \approx 5$ and $7.5 \approx 10$. Due to the approximation of values, this step in the encoding is losing information that cannot be recovered by the decoder.

2.4.3 DCT/IDCT

The DCT is a technique for converting a signal into elementary frequency components. The one-dimentional version of the DCT is typically used for converting audio signals, while the two-dimentional version is used for converting images, and in this context the individual frames of a video stream.

The one-dimentional DCT is given by equation 2.1. $s(x)$ is a list of n real numbers and $S(u)$ is the list containing the transformed elements. In all of the four equations $C(u)$ is defined as

$$C(u) = \begin{cases} 2^{-\frac{1}{2}} \text{ for } u = 0 \\ 0 \text{ otherwise} \end{cases}$$

To restore the original values equation the inverse DCT can be used. The IDCT is shown in equation 2.2. $S(u)$ contains the transformed values computed by the DCT. $s(x)$ will contain the original values after the calculation has been completed.

$$S(u) = \sqrt{\frac{2}{N}} C(u) \sum_{x=0}^{N-1} s(x) cos \frac{(2x+1)u\pi}{2N} \quad u = 0, \ldots, N \tag{2.1}$$

$$s(x) = \sqrt{\frac{2}{N}} \sum_{u=0}^{N-1} C(u) S(u) cos \frac{(2x+1)u\pi}{2N} \quad x = 0, \ldots, N \tag{2.2}$$

When applying the DCT to images the two-dimentional version is used. It is given in equation 2.3. In this case $f(x, y)$ is the value of one pixel in the frame. The position of the pixel is given by its x and y coordinates. $F(u, v)$ stores the transformed values. The two dimentional DCT can be computed by applying the one-dimentional version to the rows in the image first and then the columns. This separation of computation of rows and columns have relevance when trying to optimize the calculation for implementation in hardware. In the case of the two dimentional DCT we have that

$$C(u) = \begin{cases} 2^{-\frac{1}{2}} \text{ for } u = 0 \\ 0 \text{ otherwise} \end{cases}$$

and

$$C(v) = \begin{cases} 2^{-\frac{1}{2}} \text{ for } v = 0 \\ 0 \text{ otherwise} \end{cases}$$

When used in the MPEG-2 codec the constant N is set to 8 since the blocks inside each macroblock are of size 8x8 pixels.

The result from applying the DCT on a 8x8 matrix of pixels is a 8x8 matrix of DCT coefficients. Figure 2.3 is from [Sik] and shows the variance distribution of DCT coefficients. The figure clearly shows that most of the coefficients have values close to zero most of the time. By including no information for values equal to or close to zero, high levels of compression can be achieved.

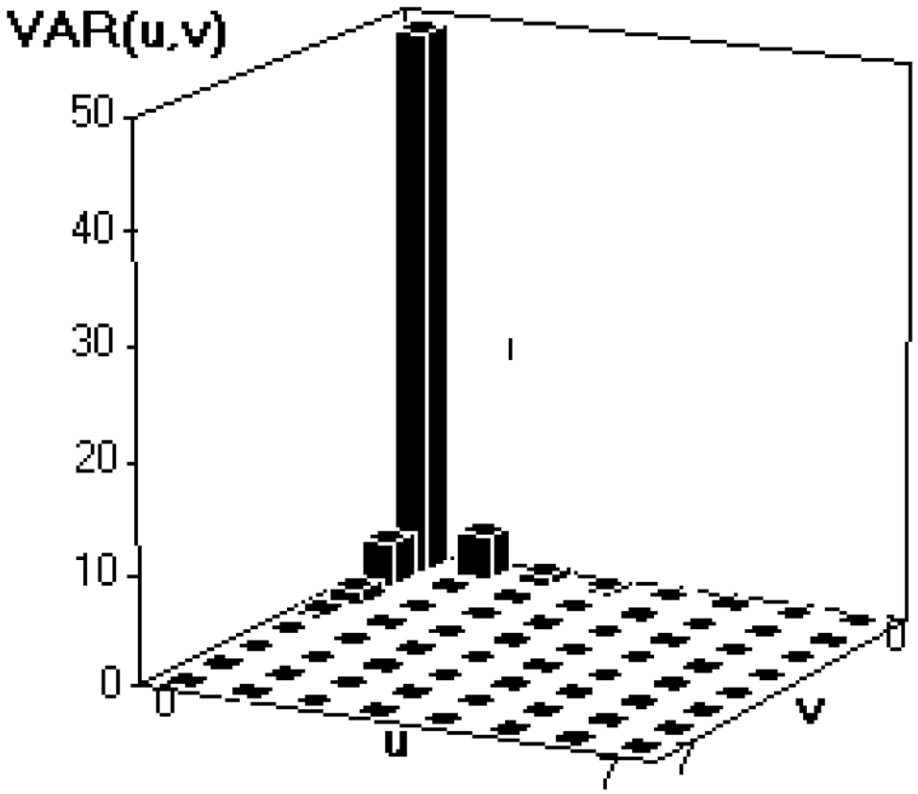

Figure 2.3: The variance distribution of DCT coefficients

$$F(u,v) = \frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)cos\frac{(2x+1)u\pi}{2N}cos\frac{(2x+1)v\pi}{2N} \qquad (2.3)$$
$$u = 0,\ldots,N \quad v = 0,\ldots,N$$

In this case I am most interested in equation 2.4, the two-dimentional inverse DCT. The definition of $C(u)$ and $C(v)$ are the same as for the DCT and $F(u,v)$ contains the transformed values from the DCT. The transformation itself, from pixel representation

to coefficients and back to pixel representation, is lossless, but due to the quantisation the resulting values from the IDCT, $f(x, y)$, are not the same as those in the original frame.

$$f(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)cos\frac{(2x+1)u\pi}{2N}cos\frac{(2y+1)v\pi}{2N} \qquad (2.4)$$
$$x = 0, \ldots, N \quad y = 0, \ldots, N$$

In libmpeg2 the IDCT is done in the C-files starting with "idct":

- **idct.c** - This is the standard C implementation without optimization for specific processors.
- **idct_alpha.c** - The implementation with optimization for the Alpha processors.
- **idct_altivec.c** - The implementation with optimization for the PowerPC processors used in Apple's computers for instance.
- **idct_mmx.c** - The implementation with optimization for the MMX instruction set used in most Intel and AMD processors.

In the standard C implementation without optimization, the computations are done by first calculating the rows and then the columns.

The whole software implementation of the IDCT can be replaced by a hardware module. [CVN99] presents a hardware module for the Xilinx FPGA that has a latency of 26 clock cycles, but since the rows and colums are computed separately a new block can be uploaded every 13 clock cycles. This module works on 8x8 matrixes where each element is 12 bits wide and in two's complement form.

2.4.4 MC

Motion compensation is perhaps the most complex part of both the encoder and the decoder. Still, in the decoder, doing MC is much simpler than in the encoder. MPEG-2 uses macroblocks of size 16x16 despite the other parts of MPEG-2 uses macroblocks of size 8x8, but as said in [ISO95]

> *...the choice of 16 by 16 macroblocks for the motion-compensation unit is a result of the trade-off between the coding gain provided by using motion information and the overhead needed to represent it.*

The simple explanation of MC is that the encoder encodes the motion of each macroblock as motion vectors. This means that macroblocks where motion vectors are available, contain information only from I-frames. The information is encoded only in the I-frames and then the motion vectors are used to move the macroblocks around in

following P-frames and previous/following B-frames. Together with the output from the IDCT, the output from MC form the basis for each frame. Figure 2.4 from [ISO95] shows the process of doing motion compensation in the decoder. What the individual modules do is explained below.

- **Vector predictors** - A storage for four vector predictors used for motion vector decoding.
- **Vector decoding** - The stage where the coded motion vectors (from the bit-stream) are decoded by using the vector predictors.
- **Additional Dual-Prime Arithmetic** - A stage for supporting one of the special prediction modes (dual prime mode).
- **Scaling for Colour Components** - Since the scale of the colour components are different than the luminance (4:2:0, 4:2:2 and 4:4:4) scaling has to be made with regards to the Cr and Cb components.
- **Half-pel Prediction Filtering** - Filtering done to improve final quality of the predicted frame.
- **Combine Prediction** - A stage to combine the various predictions together in order to form the final prediction blocks.
- **Summation** - Combining the coefficients from the IDCT with the data from the motion compensation.
- **Saturation** - Making sure that there are no negative values.
- **Prediction Field/Frame Selection** - The selection of which fields and frames shall be used as a base for predictions.
- **Framestore Addressing** - Finding the frame(s) to base the prediction on.
- **Framestores** - Storage for frames used for prediction.

By comparing the two frames in figure 2.5 it is easy to see that using MC, along with the fact that no information has to be included for pixels that do not change between frames, gives high levels of compression. Figure 2.6 shows the actual change between the two frames. The changes are shown with colors, the pixels that are the same between the frames are shown in gray. Actually it is only the pixels representing the ball that is changing. Figure 2.7 shows a picture zooming in on the ball. The yellow square of 16x16 pixels shows an example of a macroblock moving from picture to picture. Figure 2.5, 2.6 and 2.7 are all from the short ”Luxor Jr.” by Pixar Animation Studios.

In libmpeg2 MC is done in the files starting with ”motion_comp”. Besides the standard C implementation, DSP instructions for Alpha, Altivec (Apple Macintosh) and MMX (Intel Pentium and some AMD processors) are also used for optimization.

To sum it up, MC is simpler in the decoder than in the encoder, since all that is done is to move each macroblock in accordance to the motion vectors and then combine it with the results from the IDCT. In the encoder end, where the motion vectors has to be found, things are much more complicated.

MC is the main part of focus in the development of new codecs, along with the quantisation, which is the lossy parts of the codec. Naturally, since digital video is moving towards higher resolution it is natural that larger areas of pixels will have the same, or almost the same, value. This means that much can be gained from moving blocks of pixels around, instead of moving a single pixel around or constantly encoding new values for each frame. Due to this, and the fact that it is only possible to compress video so much with VLD, DCT and quantisation, and we are quite near the possible limit with these techniques, it is quite natural that the focus is on motion compensation.

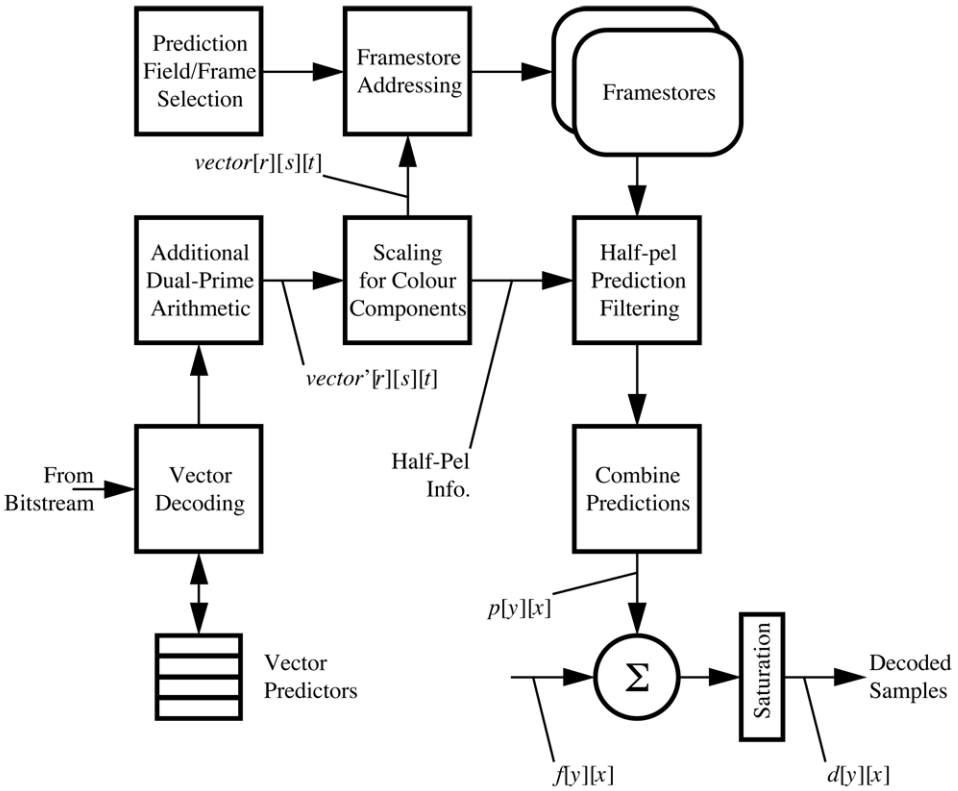

Figure 2.4: The process of doing motion compensation.

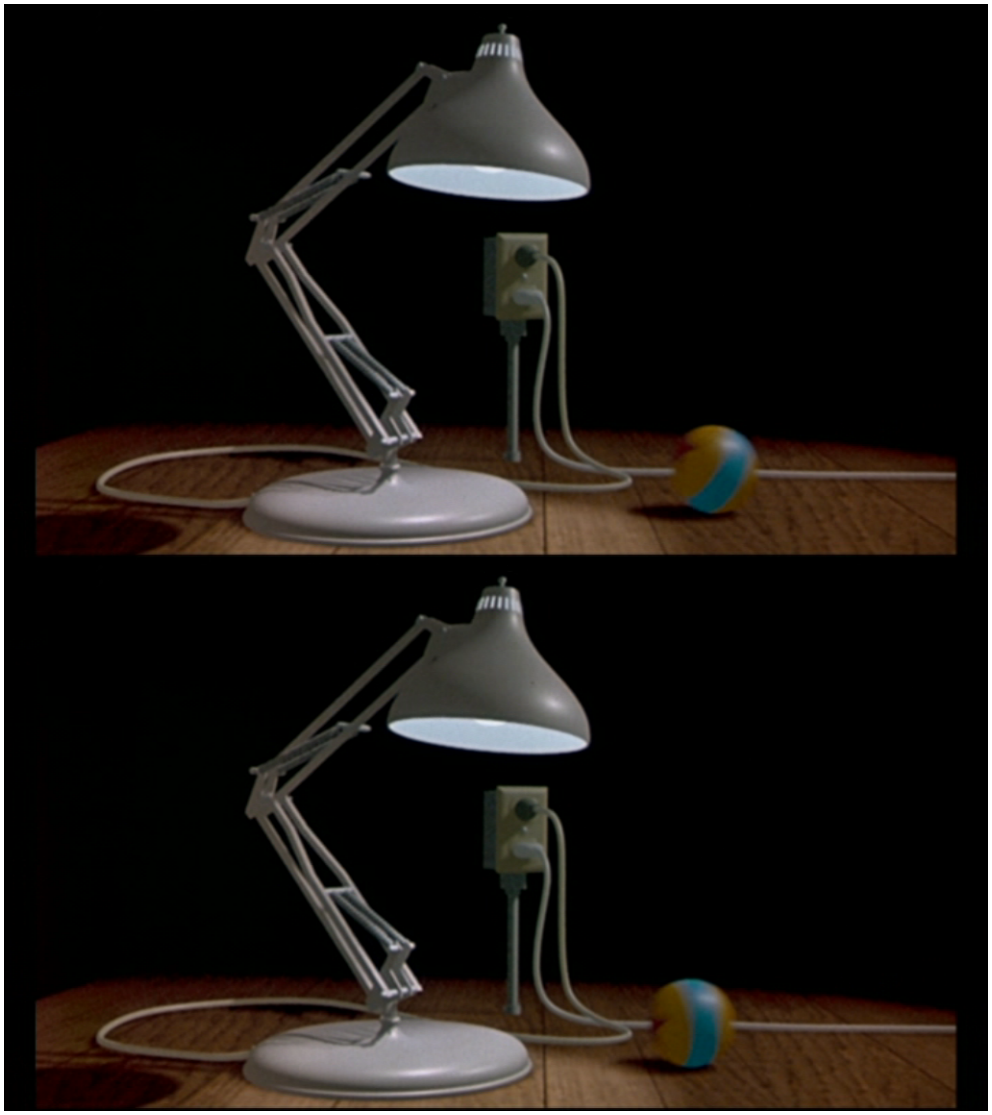

Figure 2.5: Two frames from a movie, showing a moving ball.

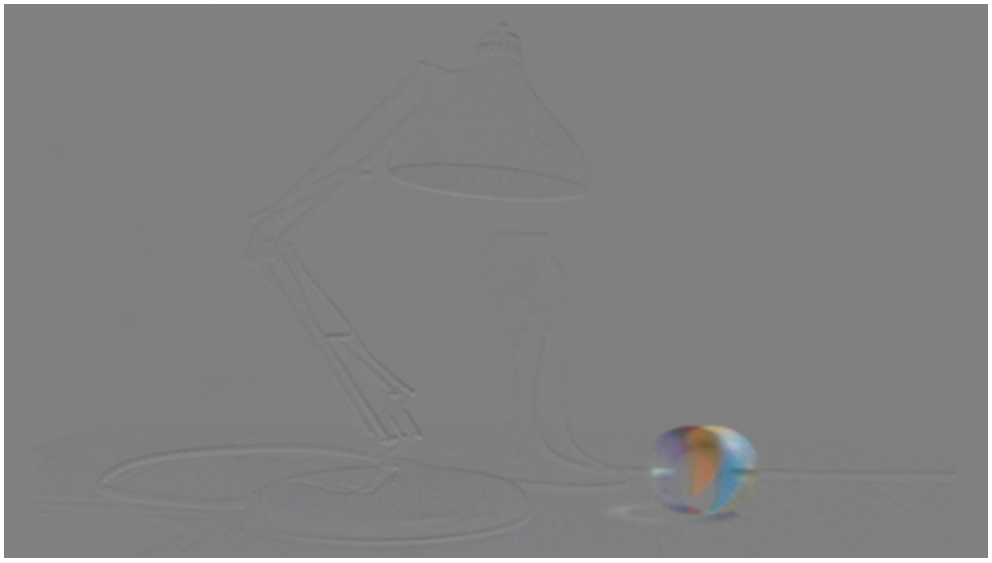

Figure 2.6: The actual difference between the two frames.

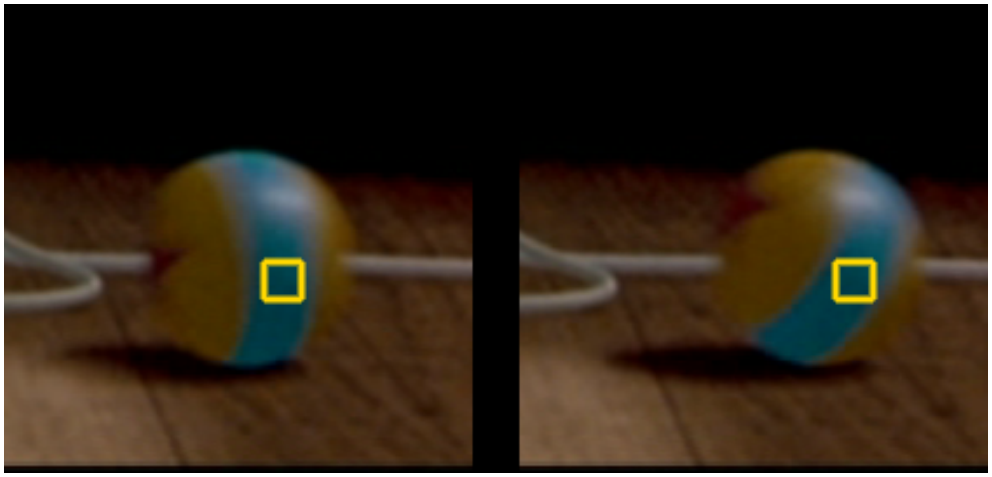

Figure 2.7: A closeup of the moving ball.

Chapter 3

Three approaces to specialized hardware

As you will see chapter 4 the gain from realizing parts of an algorithm in hardware is largely based on its properties, whether they are parallel or serial in nature. Another factor is also the implementation approach and the granularity level selected. Should the implementation be on instruction level, block level or some higher lever spanning several blocks, and how complex can the new implementation be to offer maximum performance and minimal power usage? In this chapter I am going to present some of the different approaches and granularity levels when implementing new parts in hardware.

3.1 Expanding the instruction set

Expanding the instruction set is a common way of enhancing the capabilities of general purpose processors, compared to the other two methods. Examples include the MMX instruction set of the Intel Pentium processor and some of AMD's processors. These extra instructions are often characterized as digital signal processing (DSP) instructions. According to [LCM+04] the OpenRISC architecture also supports a DSP instruction set. But according to [Lam01] the OR1200 does not implement this instruction set, only ORBIS32, the 32-bit OpenRISC Basic Instruction Set.

Expanding the instruction set is perhaps the simplest method to increase the capabilities of the processor. Essentially all that has to be added are the resources needed to perform the new instructions. Then all the new modules has to be connected to the existing modules in the processor.

The advantage of this method is clearly the simplicity of the implementation and the added amount of hardware involved, which is usually low compared to the other two options. The disadvantage is the fact that the modifications involves the processor core. This means that the whole core has to be tested for errors once more. An other important point is that the compiler and assembler has to be rewritten if they are to sup-

port the new instruction. To alter the assembler is usually a straight forward procedure, since it only maps the instruction into binary code. To make the compiler recognize where to insert the new instruction is a bit more complex and requires a lot more work since a lexical statement has to be written. The simplest way to use the new instruction along with a high-level compiler would be to use some form of inline assembly, which involves writing the assembly code directly into the C code, for instance. This, on the other hand, means that the software developers have to know that the compiler does not support the instruction and have to write assembly code directly to take advantage of it. This will of course be a problem if the processor is to be used in a general-purpose computer, for instance. With SoCs this is not so much an issue. In these cases there are a much narrower programmer base and in most cases the programmers will know the processor well anyway, since a difference in code size by only a few bytes can mean added cost of thousands of dollars[1], and optimization of the running code is often necessary.

The parts ideally suited for implementation as a new instruction would be ones that are beeing executed 10^6 times each second and upward. Usually the calculations are beeing done on only a few bytes of data. In general-purpose processors the DSP instruction set the new instructions are supposed to support calculations that can be found in most programs involving signal processing and mathematic calculations. In processors like the OR1200, which is clearly targetting embedded applications, more liberties can be taken. The new instructions can be custom made to better suite the programs to be run on the processor. In the case of running a MPEG-2 decoder on the OR1200, I would check which part of the MPEG-2 decoder was taking a lot of time, along with beeing among the parts of the decoder that executed the most times per second. The part small enough to operate at a granularity level of only a couple a few bytes (a couple of registers) and that was executed a high number of times each second would be the ideal candidate for implementation.

3.2 IO unit

For a simple introduction to IO and DMA units, [PH98] is a very good place to start.

Compared to adding new instructions to a processor, adding a IO unit is different in a number of ways:

- The IO unit is a module placed outside the processor core.
- No changes in the processor core should be necessary.
- The IO unit is connected to some form of interconnect, usually a bus.

[1]For instance if a poorly programmed program cannot fit onto the memory of some chip, and a larger and more expensive chip has to be bought. When dealing with millions of chips the added cost of perhaps a dollar becoms an issue.

- The IO unit works on larger datasets than a DSP instruction.
- No extra instructions are added since existing load and store instructions will be used for moving data back and forth between memories.

The IO unit is usually a module that is placed outside the processor core. It still has to be connected to the processor somehow, and the common way to do it is to connect the module to some form of bus. In the case of the OR1200 the new IO unit could implement the Wishbone interface, have a new adress assigned to it and be connected to the existing Wishbone bus.

Depending on choice of implementation this is the way an IO unit would work together with the OR1200 processor:

- The IO unit is placed outside the processor core.
- It implements a Wisbone interface and is connected to the Wishbone bus.
- By reading and writing from and to specific adresses, the processor can communicate with the IO unit and transfer data to and from it. All of these reads and writes requires access to the bus.
- Once the data required for the computation is in the IO unit's memory the processor signals that the data are ready by writing to a specific adress inside the IO unit.
- When the data are ready there are two different ways for the IO unit to signal the processor:
- The IO unit does not have any way of communicating directly with the processor. Therefor the processor has to poll the IO unit with regulare intervals to check whether it is finished or not. This is done by reading from a special adress inside the IO unit. These pollings do also require access to the bus.
- The IO unit takes advantage of the fact that the OR1200 has available interrupt lines and signals the processor via one of them. This means that a connection, other than the bus, has to be made between the IO unit and the processor; an interrupt signal. This should, however, be a relatively simple task.

If the IO unit has lot of memory available, it is possible for the processor to upload one block of data, start the IO unit and while the new data are beeing processed, download the ready data from the IO unit and upload new data to it. And after all that has been done, if the processor is using polling, it can check whether the new data is ready or not.

The advantages is that no or very few and simple modifications has to be made to the processor core. This leads to less testing than in the case of adding new instructions, where all the changes were made in the processor core. Besides, no modifications has

to be made to the compiler or the assembler since the programmer takes advantage of existing instructions when moving data back and forth between main memory the the memory inside the IO unit, which also exist in the global address space. Processing larger blocks of data usually also means that parallelism in the data can be better exploited; a module doing calculations on large chunks of data are usually more effective than modules doing calculations on small chunks of data. The disadvantage is that the processor is using valueable execution time to move data from its memory to the memory on the IO unit, and back when the computations are done. Polling the IO unit to check if it is done also takes time which the processor could have been using doing other and more useful work. The question is whether the resulting decrease in execution time from doing calculations on larger chunks of data will outweigh the increase in execution time from bus transfers and polling. And also if the increased efficiency weighs up to the increased hardware complexity compared to the simpler implementation of new instructions.

3.3 DMA unit

In comparison to the IO unit, the DMA unit is very similar:

- The DMA unit is a module placed outside the processor core.
- No changes in the processor core should be necessary.
- The DMA unit is connected to some form of interconnect, usually a bus.
- The DMA unit works on larger datasets than a DSP instruction, about the same size as the IO unit.
- No extra instructions are needed since the DMA will fetch data from main memory on its own. For communicating directly with the DMA unit, the processor uses existing load/store instructions.

In fact much of the internals of the DMA module is the same as the in the IO unit, and the bus interface is also the same; the DMA unit will be connected to the Wisbone bus and all communications, besides interrupt signals, will go through the bus.

There are two very different ways of doing DMA:

- The first is to use a DMA controller. The DMA controller is a unit that controls other units' access to a shared memory. This means that the DMA unit has to incorporate some means of communication with the DMA controller.
- The second is to design some scheme for assigning different parts of main memory different functions and making sure that the DMA unit does not accidentally read from or write to a memory location that the CPU is currently reading or writing. This method require less hardware than the DMA controller approach,

but at the same time it works best when there are few units accessing the same memory.

The fact that the DMA unit is capable of accessing the memory without help from the CPU is, along with a possible implementation of a DMA controller, the main difference between an IO unit and a DMA unit.

The method of using a DMA unit offer some advantages:

- The processor does not have to move data from the memory to the DMA unit, it only tells the DMA unit where the data can be found in the shared memory and then signals when they are ready.
- The DMA unit then fetches the data from memory, does the processing and writes it back to the shared memory.
- Then the DMA unit signals to the processor that new data are ready with an interrupt, and tells it where the data can be found.
- Instead of moving data around and polling the DMA unit, the processor can use the time to do useful computations.
- Correct reading and writing to shared memory is insured by the DMA controller or the scheme invented for preventing accidental memory reads and writes.
- With a DMA controller, there can easily be more than one DMA unit.

Actually the processor can use polling in the case of a DMA unit also, but it is less effective so an interrupt is the natural thing when working with DMA units. It must be said that this is the most elegant and effective method of the three. The processor only has to tell the DMA unit where new data can be found and where the resulting data should be put, and then all the work is done by the DMA unit. But it is also the most complex and the one that requires the most hardware resources.

In most modern computers there is a DMA controller. The reason for having an own controller is that a number of features are needed in these systems:

- A number of devices need direct access to memory; network interface cards, sound cards, disk controllers and IO controllers are examples.
- Due to a high number of devices, some form of arbitration is needed.
- Some form of prioritization is often needed.
- It is often desired to protect areas of the memory so that only one or a few devices have access to the area.

When it comes to SoCs, it is often not necessary to use a dedicated memory controller. The main issue is to make the memory accessible to the devices and then make sure that no device access the wrong memory area (at the wrong time). One of the main problems is that the DMA units will bypass both memory management unit and caches, so special attention has to made so that caches are updated correctly. In this case I will not be using a DMA controller since there are only two units that will be accessing main memory, and both the memory and CPU core is connected to the Wishbone interface. Besides the version of OR1200 that I will be using does not implement DMMU or caches, which leads to less problems when incorporating a DMA unit into the design. The fact that the memory is connected to the Wishbone interface makes it very easy to make it accessible to one more device. Actually the memory is connected to a simple memory controller that is connected to the Wishbone bus. The memory controller can hardly be called a controller, it has more resemblance with an interface between the memory module and the Wishbone bus. However this is an exampel of a simple approach that could have been used for connecting a DMA unit to the Wishbone bus and make it work together with the processor:

- At a predefined location in memory a table containing pointers to the memory location for the next read/write for the DMA unit is stored.
- Initially these values are 0.
- The processor stores a data block (on which the DMA unit is about to do some computations) in memory, writes the location in the table, reserves a memory block where the resulting data should be stored, writes that location in memory and signals to the DMA unit that a new block has been written.
- The DMA units looks up in the table and starts reading from the memory location that the value in the table is pointing to.
- When the DMA unit is done, it writes back the values to memory at the location indicated in the table and signals to the processor that new data have arrived.

In the event of further expansions in the form of additional DMA units, it would be wise to consider using a DMA controller. More DMA units means coming up with cleverer schemes for avoiding accidental reads and writes and after a certain number of units this will undoubtably become a difficult task. Besides most DMA controllers offer great performance and prioritization for the most time-critcal modules. For the OR1200 the DMA controller/Wishbone bridge presented in [Uss02] could be considered.

Chapter 4

Selecting parts for optimization

The profiling results presented in this sections and the following chapters are all run on a x86 processor due to an error in the C libraries for uClinux that made it difficult to compile the decoder. The compilation completed but the resulting binary file would not run on OR1200. Hopefully this matter has been resolved by the recent release of Linux for OR1200. However, all the assemblycode presented are run through GCC for OR1200. The reason for that is that the source code only goes through the assembly stage and it is not dependent on the C libraries. The system used was described in section 2.2.

4.1 Requirements for optimizable parts

Before selecting which parts of the program to optimize there are a few key points that charaterize parts suited for implementation in hardware. The modules of choice would incorporate some (or all) of the following properties:

- Parallel at low granularity.
- Parallel at high granularity.
- Low number of serial elements (if and case statements).
- Use a large percentage of the run-time.

Naturally, inherently parallel programs are better suited for implementation in hardware than inherently serial programs due to the fact that hardware is inherently parallel. The difference between granularity levels can be seen as the difference between working on single bytes in parallel and working on blocks of bytes in parallel. The advantage would be reuse of hardware modules but also the ability to do calculations on different data elements simultaneously. Since one module would compute N bytes, to double the performance it would be enough to add one more module so that two N-byte modules could be computed at the same time.

Serial parts of the program, like if and case statements, usually have a great impact on performace. Since computations to follow such statements strongly depends on the outcome of the statement, it basically creates a halt in the program. This means that all consecutive computations are stalled until the statement is completed. A program considered for implementation in hardware should have as few of these serial elements as possible.

The last point in the list is just common sense: Start optimizing where it is most to gain. Since hardware sometimes offer tremendous speedups compared to software it would be wise to explore the most time-consuming parts of the program. To decrease the execution time of these parts would definently decrease the total execution time noticeably.

4.2 Profiling of the MPEG-2 decoder

An easy and powerful way to find the bottlenecks in the decoder is by using a profiling tool. Such a tool will usually give information such as how many times a specfic function of the program is executed and how much time is spent in each function. The most advanced can even count instructions and give details on which instructions are executed most frequently. On the Linux platform there exists such a program called ”gprof”, which is a simple but powerful profiling tool. It does not offer more advanced features such as instruction level statistics but in this setting it offers just the right features. To use the profiler requires going through three steps:

- Compilation of the program with the ”-pg” switch enabled in GCC/CC.
- Running the program to completion to generate call-graph data.
- Running gprof on the generated call-graph data.

These three steps will result in a log file with data similar to the example shown in figure 4.1. The entire log can be seen in section A.1. The meaning of the seven fields shown in figure 4.1 are:

- % time: This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.
- Cumulative seconds: This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- Self seconds: This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.
- Self calls: This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 36.06    227.09   227.09 28911026     0.01     0.01  mpeg2_idct_add_c
 16.69    332.21   105.12 11020620     0.01     0.01  mpeg2_idct_copy_c
 10.72    399.70    67.49 28911026     0.00     0.00  get_non_intra_block
  7.23    445.21    45.51 11020620     0.00     0.00  get_intra_block_B15
  6.63    486.98    41.77  7738491     0.01     0.01  MC_put_o_16_c
  4.83    517.42    30.44 21908410     0.00     0.00  MC_put_o_8_c
  4.00    542.60    25.18   295200     0.09     2.11  mpeg2_slice
  2.85    560.58    17.98  2415495     0.01     0.01  MC_put_xy_16_c
  1.95    572.87    12.29  1807921     0.01     0.01  MC_put_x_16_c
  1.41    581.78     8.91  1498165     0.01     0.01  MC_put_y_16_c
  1.26    589.72     7.94  6912398     0.00     0.01  motion_fr_frame_420
  1.20    597.27     7.55  2424764     0.00     0.02  motion_fr_field_420
  1.13    604.39     7.12    79606     0.09     7.91  mpeg2_parse
  0.59    608.13     3.74  2360896     0.00     0.00  MC_put_x_8_c
  0.52    611.42     3.29   346120     0.01     0.01  MC_avg_xy_16_c
  0.50    614.54     3.12   487038     0.01     0.01  MC_avg_o_16_c
  0.40    617.07     2.53  1157494     0.00     0.00  MC_put_xy_8_c
  0.37    619.43     2.36  1351582     0.00     0.00  MC_avg_o_8_c
  0.37    621.77     2.34  1493344     0.00     0.00  MC_put_y_8_c
  0.29    623.57     1.80   263845     0.01     0.01  MC_avg_x_16_c
  0.24    625.10     1.53   183296     0.01     0.01  MC_avg_y_16_c
  0.17    626.17     1.07   568662     0.00     0.00  MC_avg_x_8_c
  0.13    627.01     0.84   292740     0.00     0.00  MC_avg_xy_8_c
  0.13    627.80     0.79   347614     0.00     0.00  MC_avg_y_8_c
  0.12    628.53     0.73  1502578     0.00     0.01  motion_reuse_420
  0.10    629.18     0.65  1475867     0.00     0.01  motion_zero_420
```

Figure 4.1: Part of the logfile from the Linux utility ”gprof”.

(probably because the function was not compiled with profiling enabled), the calls field is blank.

- Total ms/call: This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.
- Ms/call: This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function. This is the only field in the flat profile that uses call graph analysis.
- Name: This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

This list is from the gprof web page[1]

4.3 Finding the bottlenecks

In [VZL] the IDCT, VLD and motion compensation have been identified as the most time consuming parts of the decoder. It does not say which decoder they used, but since the parts identified are essential parts of any MPEG-2 decoder, it is very likely that they will be the most time consuming parts of libmpeg2 also. The reason for re-checking these results with this decoder and not simply take the results are threefold:

- I want to identify which functions used in *this* decoder that are the most time consuming.
- I want to see how large percentage each function use compared to the others. Most likely the results will only differ by a few percent compared to those in [VZL] but then again, the differences could be significant.
- I am propably using a different decoder than the one they have used and I want to implement specific parts of the decoder. That is, I want to take one or more parts out of the decoder and replace it with a (partly) custom made hardware module.

The first step in profiling the decoder is to compile each file with the ”-pg” flag enabled in GCC. In the GCC manual entry it says that the ”-pg ” switch is used to ”Generate extra code to write profile information suitable for the analysis program gprof”. To incorporate the ”-pg” flag in the compilation of every file use these two commands:

[1] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html

```
./configure CFLAGS="-pg"
make all install
```

The first command will signal to the compiler that it should compile each file with the ”-pg” flag switched on and creates a configuration file for the next step. The second command compiles and install the decoder library, and also the small test application ”mpeg2dec”.

Based on the numbers generated by gprof when running the ”mpeg2dec” test application, it is possible to find which parts of the decoder that take the longest time. Figure 4.1 shows a section of the logfile generated from a run of the decoder without any modification. The MPEG-2 stream used for testing is described in section 2.3.1. From the numbers in figure 4.1 we can see that the calculation of the IDCT through the execution of the functions ”mpeg2_idct_add_c” and ”mpeg2_idct_copy_c” takes almost 53% of the total execution time. The functions ”get_non_intra_block” and ”get_intra_block_B15”, which are used for VLD takes about 18% of the total execution time. The different parts of the motion compensation, the functions whose names start with ”MC”, uses about 24% alltogether. Together these three parts of the MPEG-2 decoder uses about 95% of the total execution time. The rest includes task such as file input and parsing of stream data. The distribution of execution time is shown in figure 4.2. The differents between these results and the initial results presented in **??** can in some areas be described as significantly different.

- In my simulation, the IDCT uses more of the execution time.
- There is also a difference in the MC part. In this case my simulation spends less time executing this part.
- The general household takes longer time in my simulation also.
- Some of the differences (especially the previous point) may be explained by the fact that I have not simulated the processin power needed to display the resulting images.

When running the libmpeg2 test program with profiling on the test film described in table 2.1, the numbers of function calls to the IDCT, VLD and MC functions were:

- IDCT:

  ∼60 million calls.

  ∼7300 calls/frame (average).

  ∼650000 calls/second (average).

- VLD:

  ∼40 million calls.

  ∼4900 calls/frame (average).

  ∼440000 calls/second (average).

- MC:

  ∼40 million calls.

  ∼4900 calls/frame (average).

  ∼440000 calls/second (average).

Since each of these parts of the decoder takes at least several hundred thousand clocks cycles each call, the improvement in reducing the number of clock cycles to hundreds or event tens would be dramatical. Such a reduction in number of clock cycles could clearly lead to a significant reduction in clock speed of the processor doing the decoding, which definently would lead to savings when it comes to power consumptions.

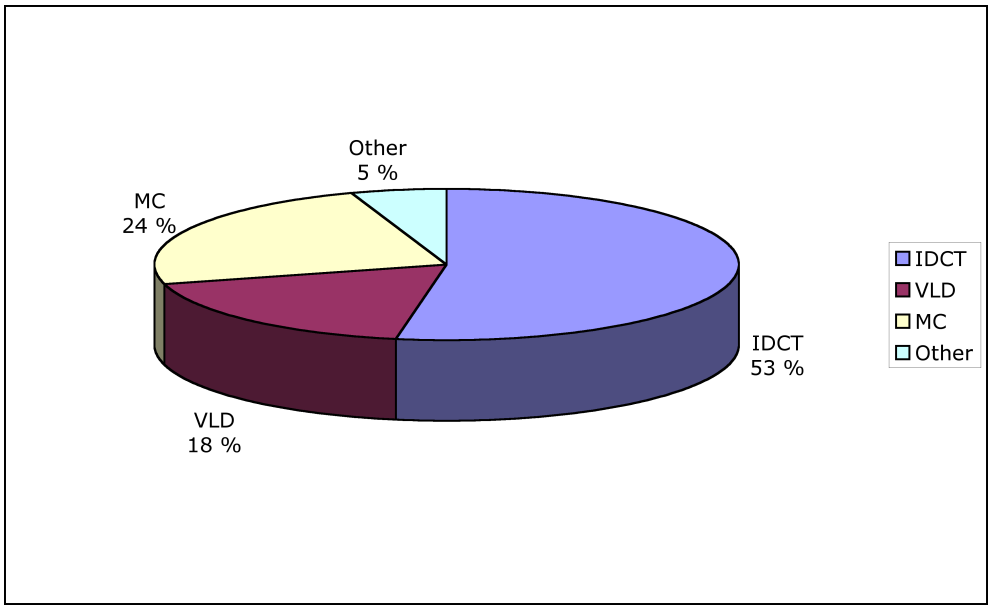

Figure 4.2: Distribution of execution time.

4.4 Final choice

Based on the properties of the parts described in section 2.4, and the profiling of the decoder done in section 4.3 the choice of part to start focusing on is pretty simple. Since the IDCT uses about 50% of the execution time, this alone should make it an obvoius choice. In addition the IDCT is highly parallel both on block and macroblock level, besides it is possible to pipeline the computation of rows and columns in each macroblock. Based on this information I will start focusing on hardware implementation of the IDCT from this point on. I will begin by describing two of the possible ways to implement parts of or the whole IDCT in hardware.

It should be noted that the approach described in the following subsections can be generalized and applied to the other parts of the decoder. This basically means that the methods for finding the right granularity level of the part to implement in hardware also applies when implementing other parts, the VLD for instance, in hardware.

4.4.1 Adding new instructions

Now that the decision on which part to focus on has been made, there are a few points that one should have in mind before starting implementing parts of the program in hardware as an instruction. The new instruction should ideally...

- ...cover a small and specific part of the program.
- ...contain some basic form of operations, additions and multiplications for instance.
- ...work on small amounts of data at a time.
- ...represent a significant part of the execution time.
- ...be executed a high number of times each second.

libmpeg2 includes optimized code for Alpha, AMD, Intel and PowerPC processors, besides there is also a standard C implementation of the IDCT. This is the version I will focus on. Since the OR1200 does not offer any DSP instruction set such as MMX or Altivec (PowerPC), the pure C version is the version that is used when compiling for uClinux on the OR1200. The entire file ”idct.c” can be found in section B.1.

From section 4.3 we already know that the IDCT is executed several thousand times each second and consumes large portions of the execution time. However in this case we are looking for a part of the IDCT that executes several times for each run, that operates on a small data set and that performes some basic operations on the data. By inspecting the file ”idct.c” I found that a part of the IDCT named ”BUTTERFLY” is well suited for implementation in hardware. Figure 4.4 shows the computations done in the butterfly; three types of computations; multiplication, subtraction and addition forming a pattern somehow similar to a multiply and accumulate (MAC) instruction. Figure 4.3 shows that the butterfly executes three times for each column computation. The butterfly is executed three times for each row computation also. Since each block is of size 8x8, the butterfly executes 48 times for each block. Using the data shown in figure 4.1, the butterfly executes over 3.8 billion times during the clip described in section B.2, which is 11.7 million times each second and 470.000 times for each frame.

By commenting out each ”BUTTERFLY” line, like the ones in figure 4.3, a simple simulation can be run. The goal is to give an indication on how much time can be saved by improving the butterfly. Figure 4.5 shows the log file from gprof after a run

```
static void inline idct_col (int16_t * const block)
{
    int d0, d1, d2, d3;
    int a0, a1, a2, a3, b0, b1, b2, b3;
    int t0, t1, t2, t3;

    d0 = (block[8*0] << 11) + 65536;
    d1 = block[8*1];
    d2 = block[8*2] << 11;
    d3 = block[8*3];
    t0 = d0 + d2;
    t1 = d0 - d2;
    BUTTERFLY (t2, t3, W6, W2, d3, d1);
    a0 = t0 + t2;
    a1 = t1 + t3;
    a2 = t1 - t3;
    a3 = t0 - t2;

    d0 = block[8*4];
    d1 = block[8*5];
    d2 = block[8*6];
    d3 = block[8*7];
    BUTTERFLY (t0, t1, W7, W1, d3, d0);
    BUTTERFLY (t2, t3, W3, W5, d1, d2);
    b0 = t0 + t2;
    b3 = t1 + t3;
    t0 -= t2;
    t1 -= t3;
    b1 = ((t0 + t1) >> 8) * 181;
    b2 = ((t0 - t1) >> 8) * 181;

    block[8*0] = (a0 + b0) >> 17;
    block[8*1] = (a1 + b1) >> 17;
    block[8*2] = (a2 + b2) >> 17;
    block[8*3] = (a3 + b3) >> 17;
    block[8*4] = (a3 - b3) >> 17;
    block[8*5] = (a2 - b2) >> 17;
    block[8*6] = (a1 - b1) >> 17;
    block[8*7] = (a0 - b0) >> 17;
}
```

Figure 4.3: The C code for doing column computations.

```
#define BUTTERFLY(t0,t1,W0,W1,d0,d1)
do {
    int tmp = W0 * (d0 + d1);
    t0 = tmp + (W1 - W0) * d1;
    t1 = tmp - (W1 + W0) * d0;
} while (0)
```

Figure 4.4: The butterfly part of the IDCT.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 23.36    105.90   105.90 28911026     0.00     0.00  mpeg2_idct_add_c
 14.88    173.37    67.47 28911026     0.00     0.00  get_non_intra_block
 10.41    220.56    47.19 11020620     0.00     0.00  mpeg2_idct_copy_c
 10.24    266.99    46.43 11020620     0.00     0.00  get_intra_block_B15
  9.17    308.58    41.59  7738491     0.01     0.01  MC_put_o_16_c
  6.93    340.00    31.42 21908410     0.00     0.00  MC_put_o_8_c
  5.57    365.26    25.26   295200     0.09     1.51  mpeg2_slice
  4.01    383.46    18.20  2415495     0.01     0.01  MC_put_xy_16_c
  2.86    396.41    12.95  1807921     0.01     0.01  MC_put_x_16_c
  1.98    405.39     8.98  1498165     0.01     0.01  MC_put_y_16_c
  1.82    413.64     8.25  6912398     0.00     0.01  motion_fr_frame_420
  1.70    421.36     7.72    79606     0.10     5.69  mpeg2_parse
  1.65    428.86     7.50  2424764     0.00     0.02  motion_fr_field_420
  0.86    432.75     3.89  2360896     0.00     0.00  MC_put_x_8_c
  0.67    435.81     3.06   346120     0.01     0.01  MC_avg_xy_16_c
  0.64    438.71     2.90   487038     0.01     0.01  MC_avg_o_16_c
  0.55    441.22     2.51  1351582     0.00     0.00  MC_avg_o_8_c
  0.53    443.62     2.40  1493344     0.00     0.00  MC_put_y_8_c
  0.46    445.69     2.07  1157494     0.00     0.00  MC_put_xy_8_c
  0.38    447.43     1.74   263845     0.01     0.01  MC_avg_x_16_c
  0.31    448.85     1.42   183296     0.01     0.01  MC_avg_y_16_c
  0.25    449.99     1.14   568662     0.00     0.00  MC_avg_x_8_c
  0.19    450.87     0.88   347614     0.00     0.00  MC_avg_y_8_c
  0.15    451.56     0.69   292740     0.00     0.00  MC_avg_xy_8_c
  0.14    452.21     0.65  1475867     0.00     0.01  motion_zero_420
  0.14    452.85     0.64  1502578     0.00     0.01  motion_reuse_420
```

Figure 4.5: The logfile from gprof after a run without the butterfly.

where all ”BUTTERFLY” lines were commented out. Compared to figure 4.1 the relative time consumptions of the functions ”mpeg2_idct_add_c” and ”mpeg2_idct_copy_c” have gone from 36.27% to 23.36% and from 16.64% to 10.41% respectively.

Based on the numbers from the gprof logs it is evident that reducing the number of clock cycles used by the butterfly have an positive impact on the performance. Now let us have a look at how many clock cycles it is possible to save by implementing the butterfly as a new instruction.

To find the effect of a new instruction I had to count the number of cycles the butterfly took without the new instruction and compare it to the number of cycles it took with the new instruction. It is possible to get the output from GCC in the form of the C statements followed by their assembly equivalent. Section **??** shows the entire output from the C file show in figure 4.6. The relevant part can be seen in figure 4.7. Adding the number of cycles, this totals to 54 cycles. It is worth mentioning that this is unoptimized code and that GCC probably generates faster code when optimization is enabled. Therefor I have included a hand-optimized code in figure 4.8. This assembly code totals to 22 cycles and this is the number of cycles that I will compare the new instruction with. The results from the actual implementation will be presented in section 6.1.

```
int main(int argc,char* args[]) {
  int W0,W1,d0,d1,t0,t1;

  int tmp = W0 * (d0 + d1);
  t0 = tmp + (W1 - W0) * d1;
  t1 = tmp - (W1 + W0) * d0;
  return 0;
}
```

Figure 4.6: The C kode implementing the butterfly only.

4.4.2 Adding IO/DMA module

Implementing an IO or DMA module usually means implementing a larger part of the functionality in hardware. Since there is a cost connected with moving data back and forth between memory and the IO/DMA module, the granularity has to be increased compared to a new instruction. This means that the IO/DMA modules has to work on larger data sets. And since each block is of size 8x8, this would be a reasonable data set size. [CVN99] describes a hardware implementation of an entire IDCT. This module can be used to replace the entire functionality of the file ”idct.c”. This module is capable of doing IDCT on an 8x8 block in 26 clock cycles. Since column and row computations are pipeline, a new block can be transferred to the module each 13th clock cycle.

```
 4:butterfly.c   ****   int tmp = W0 * (d0 + d1);
47               .LM2:
48 0014 8462FFEC  l.lwz     r3,-20(r2) # Load d0
49 0018 8482FFE8  l.lwz     r4,-24(r2) # Load d1
50 001c E0632000  l.add    r3,r3,r4 # Do addition (d0 + d1)
51 0020 D7E21FD8  l.sw     -40(r2),r3 # Store (d0 + d1)
52 0024 8462FFF4  l.lwz     r3,-12(r2) # Load W0
53 0028 8482FFD8  l.lwz     r4,-40(r2) # Load (d0 + d1)
54 002c E0632306  l.mul    r3,r3,r4 # Do multiplication W0 * (d0 + d1)
55 0030 D7E21FDC  l.sw     -36(r2),r3 # Store tmp (W0 * (d0 + d1))
 5:butterfly.c   ****   t0 = tmp + (W1 - W0) * d1;
57               .LM3:
58 0034 8462FFF0  l.lwz     r3,-16(r2) # Load W1
59 0038 8482FFF4  l.lwz     r4,-12(r2) # Load W0
60 003c E0632002  l.sub    r3,r3,r4 # Do subtraction (W1 - W0)
61 0040 D7E21FD4  l.sw     -44(r2),r3 # Store (W1 - W0)
62 0044 8462FFD4  l.lwz     r3,-44(r2) # Load (W1 - W0)
63 0048 8482FFE8  l.lwz     r4,-24(r2) # Load d1
64 004c E0632306  l.mul    r3,r3,r4 # Do multiplication (W1 - W0) * d1
65 0050 D7E21FD0  l.sw     -48(r2),r3 # Store (W1 - W0) * d1
66 0054 8462FFDC  l.lwz     r3,-36(r2) # Load tmp (W0 * (d0 + d1))
67 0058 8482FFD0  l.lwz     r4,-48(r2) # Load (W1 - W0) * d1
68 005c E0632000  l.add    r3,r3,r4 # Do addition tmp + (W1 - W0) * d1
69 0060 D7E21FE4  l.sw     -28(r2),r3 # Store t0 (tmp + (W1 - W0) * d1)
 6:butterfly.c   ****   t1 = tmp - (W1 + W0) * d0;
71               .LM4:
72 0064 8462FFF0  l.lwz     r3,-16(r2) # Load W1
73 0068 8482FFF4  l.lwz     r4,-12(r2) # Load W0
74 006c E0632000  l.add    r3,r3,r4 # Do addition (W1 + W0)
75 0070 D7E21FCC  l.sw     -52(r2),r3 # Store (W1 + W0)
76 0074 8462FFCC  l.lwz     r3,-52(r2) # Load (W1 + W0)
77 0078 8482FFEC  l.lwz     r4,-20(r2) # Load d0
78 007c E0632306  l.mul    r3,r3,r4 # Do multiplication (W1 + W0) * d0
79 0080 D7E21FC8  l.sw     -56(r2),r3 # Store (W1 + W0) * d0
80 0084 8462FFDC  l.lwz     r3,-36(r2) # Load tmp (W0 * (d0 + d1))
81 0088 8482FFC8  l.lwz     r4,-56(r2) # Load (W1 + W0) * d0
82 008c E0632002  l.sub    r3,r3,r4 # Do subtraction tmp - (W1 + W0) * d0
83 0090 D7E21FE0  l.sw     -32(r2),r3 # Store t0 (tmp - (W1 + W0) * d0)
```

Figure 4.7: The main part of the assembly code from the butterfly.

```
# tmp = W0 * (d0 + d1)
l.lwz r3,d0 # Load d0
l.lwz r4,d1 # Load d1
l.lwz r5,W0 # Load W0
l.lwz r6,W1 # Load W1

l.add r7,r3,r4 # Do addition (d0 + d1)
l.mul r7,r5,r7 # Do multiplication (W0 * (d0 + d1))
# tmp now stored in r7
# t0 = tmp + (W1 - W0) * d1
l.sub r8,r6,r5 # Do subtraction (W1 - W0)
l.mul r8,r8,r4 # Do multiplication ((W1 - W0) * d1)
l.add r8,r7,r8 # Do addition (tmp + (W1 - W0) * d1)
# t0 now stored in r8
# t1 = tmp - (W1 + W0) * d0
l.add r9,r6,r5 # Do addition (W1 + W0)
l.mul r9,r9,r3 # Do multiplication ((W1 + W0) * d0)
l.sub r9,r7,r9 # Do subtraction (tmp - (W1 + W0) * d0)
#t1 now stored in r9
```

Figure 4.8: Main part of the assembly code from the butterfly. But this time in an optimized version.

Normally the IDCT is done in the file ”idct.c”. When using the Xilinx IDCT module there are two different ways to replace ”idct.c” depending on whether it is an IO or DMA unit that is beeing implemented:

- **IO unit:** The calculations in ”idct.c” are replaced by memory operations moving data from main memory to the memory in the IO unit.
- **DMA unit:** The calculations in ”idct.c” are replaced by memory operations writing addressed to memory, telling the DMA unit where to find the data in main memory, and where to put the new data when they are ready.

According to [Her02] there can be one data transfer each clock cycle on the Wishbone bus. The specification limits each data transfer to a maximum of 64 bits. In the case of OR1200, 32 bit data transfers are beeing used. This means that it takes 32 cycles to transfer one block over the Wishbone bus, since each element in the block is 16 bits long. It takes twice as long to move to move each block back and forth, since it has to be moved from main memory to the memory on the IO/DMA unit and then back to main memory.

The advantage of implementing a DMA unit is that the unit itself can handle all memory read and writes. The main disadvantage is that added hardware is required in the form of a DMA controller that controls access to main memory. Or the extra effort in creating a scheme for controlling access to memory in cases when a DMA controller is not implemented. The IO controller does not need a DMA controller but the processor has to handle all memory read and writes. The memory in both the DMA and IO unit can be viewed as an extension of main memory address space.

The results from the simulated run with the IO unit are presented in section 6.2 and the results from the simulated run with the DMA unit are presented in section 6.3.

Chapter 5

Adding new instructions to the architecture

5.1 Hardware and software

The following hardware was used during implementation, simulation and testing of the new hardware:

- Intel Pentium 4 2.40 GHz.
- 512 MB RAM.
- Mandrake Linux 10.0 used for software simulation.
- Software from the Opencores.org website was used during software testing.
- Microsoft Windows XP with SP2 used for synthesis.
- Xilinx ISE 6 as synthesis tool.
- Modelsim SE 5.7g for simulating the hardware.
- Virtex-II™V2MB1000 Development Board was used for simulating the processor.
- Xilinx Parallel Cable IV JTAG cable.
- Serial cable (for the COM port).

5.2 Modifying the processor core

When creating a new module and adding this to the processor core there are two different aspects of the module that has to be considered:

- The module itself, the internals of the module has to be created and working according to specifications
- The external part of the module has to interface correctly with the rest of the processor core.

When I started making the hardware for the new instructions I wanted to do the following things:

- Make as few changes to the processor core as possible.
- Use the existing resources to fetch and store information, and do calculations.
- Make it as fast as possible. One clock cycle if possible.

In the following sections I will present how to modify the processor with respect to the points presented above.

5.2.1 The internals of the module

The internals of the module should of course use as short time as possible to do the required calculations. Ideally it would only take one clock cycle, but in this case there are two values that have to be written back to the register file (RF). As seen in figure 4.4, the values t0 and t1 have to be written back to the RF. This takes no less than two clock cycles, so it should be a goal for the module to do the calculations in no more than two clock cycles.

There are several ways to realize this module in hardware, but the one I have chosen takes advantages of these points:

- Due to the fact that write-back to RF takes two cycles, computations can also be divided over two clock cycles.
- In this case the two clock cycles are divided between two different instructions. Each responsible for one of the ”t” values.
- The ”W” values used in the IDCT[1] can be stored as constants inside the module.
- The calculations involving only the ”W” values can be pre-calculated and also stored in the module.
- Since the ”W” values are stored inside the module, only two operands has to be provided. This means that the existing mechanism for fetching operands can be used.
- A closer look at the IDCT source code shows that only three different combinations of ”W” values and operands exists.

[1] See section B.1 to see their definitions, and figure 5.2 to see the usage of them.

- As seen in figure 5.1 the bit pattern of the instructions in the OR1200 has an opcode field of four bits at the end. This opcode field can be used for selecting which ”W” values to be used for the three different combinations and also for selecting which calculations to do.

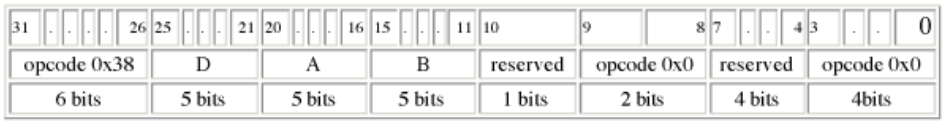

Figure 5.1: The groups of bits that make up each instruction.

Before starting to explain the implementations, I have to repeat that the IDCT only use three different combinations of ”W” values, and this is an important point since this means some of the values and calculations can be stored in arrays. Based on the facts presented above I designed the calculations of the ”t” values in the following manner (see figure 5.2 for reference to the variable names):

- The operands, the ”d” values are provided to the module from external resources.
- The ”W”-values and calculations involving only ”W”-values are stored in three arrays ”SW”, ”AW” and ”W”

  SW - the results from W1 - W0.

  AW - the results from W1 + W0.

  W - the W values.
- The three different combinations of ”W”-values are given by:

  BUTTERFLY (t2, t3, **W6, W2**, d3, d1);

  BUTTERFLY (t0, t1, **W7, W1**, d3, d0);

  BUTTERFLY (t2, t3, **W3, W5**, d1, d2);
- Which gives the following table of ”W”-values to store (the values are given in the ”idct.c” file):

  W[0] = W6 = 1108

  W[1] = W7 = 565

  W[2] = W3 = 2408
- The ”AW”-values to be stored are:

  AW[0] = W2 + W6 = 2676 + 1108 = 3784

AW[1] = W1 + W7 = 2841 + 565 = 3406

AW[2] = W5 + W3 = 1609 + 2408 = 4017

- The ”SW”-values to be stored are:

  SW[0] = W2 - W6 = 2676 - 1108 = 1568

  SW[1] = W1 - W7 = 2841 - 565 = 2276

  SW[2] = W5 - W3 = 1609 - 2408 = -799

- The three first of the four opcode bits in the new instructions are used to look up the values in the arrays.

- For each instruction I will be calculating tmp = W0 + (d0 + d1). This means double work, but it will not affect the time limit, which is 1 clock cycle.

- Bit [0] in each instruction will decide which calculation will follow:

  Bit [0] = ’0’ -> t0 = tmp + SW * d1;

  Bit [0] = ’1’ -> t1 = tmp - AW * d0;

The final implementation of the butterfly hardware can be seen in figure 5.3 and a schematic of the implementation can be seen in figure 5.2. The first ”always” process initialize the ”W”, ”SW” and ”AW” constants. This initialization cannot be done in an ”initial” process, since it is not supported by the synthesis tools. In the second ”always” process the first bit of ”bf_select” decides which values to use and which calculations to do. The reason for putting the operands and ”SW”/”AW” values in registers is so that the same multiplier can be used regardless of which calculation is being performed. This saves hardware by having one multiplier instead of two.

Since this is a module that is not dependent on the clock, values sort of just ”pass through”, meaning that the only thing we have to worry about is the delay from the operands are placed on the input ports until a new value is presented on the output port. In the case of this module, Xilinx ISE stated that the delay was a maximum of 6.322 ns, meaning that the chip can run at maximum 158 MHz before the delay of this module becomes an issue. Considering that a chip based on this processor should run at about 150-200 MHz max in order to save power, a delay of 6.322 ns might in fact be an issue. However, from personal experience, the number for maximum delay often changes dramatically between different synthesis tools. I think that this delay would not be a problem if realized in an ASIC, for instance. Just to compare, the Xilinx ISE synthesis tools reports that the maximum frequency for the processor as a whole is about 57 MHz which equals a minimum period of about 18 ns.

5.2.2 Interfacing with the processor core

Besides writing the module itself I have also added some statements to other files to connect my module to the rest of the processor. The steps taken for each file are listed

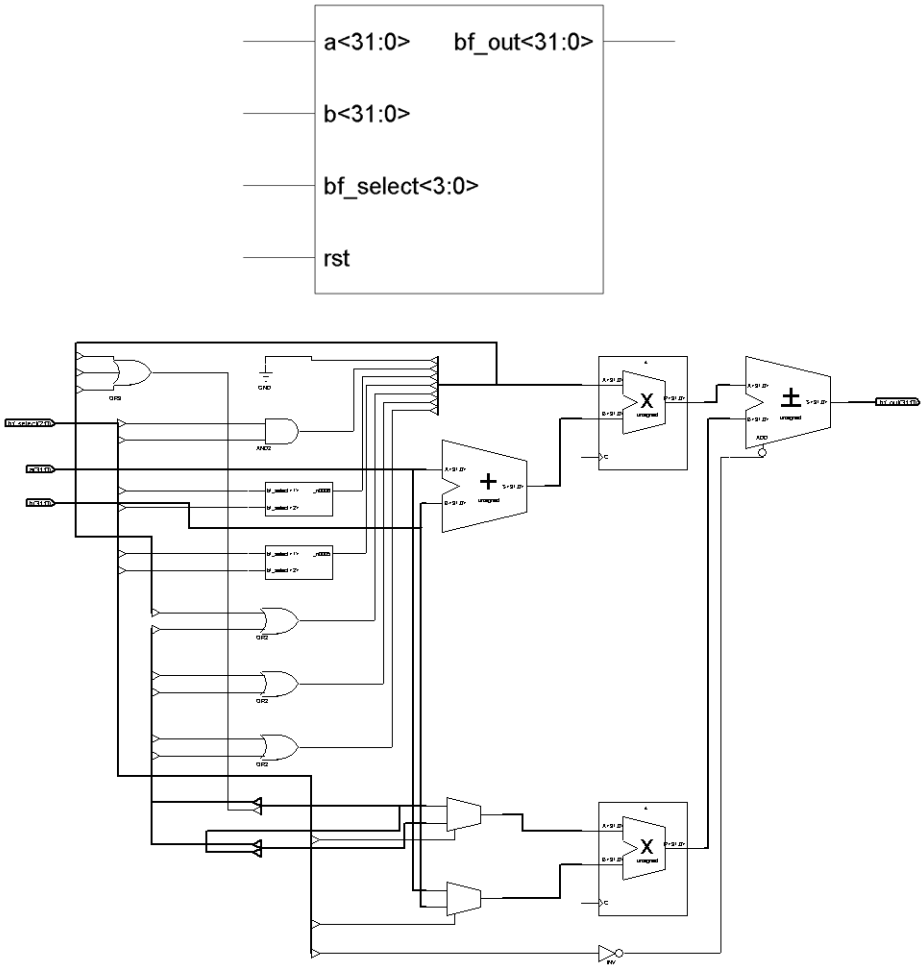

Figure 5.2: Schematic of the internals of the butterfly module.

```
module or1200_butterfly(rst,bf_select,a,b,bf_out);

input rst;
input [3:0] bf_select;
input [31:0] a;
input [31:0] b;
output [31:0] bf_out;

reg [31:0] tmp, mulvalue, wvalue, tmpvalue, tbf_out;
reg [31:0] W[0:3], SW[0:3], AW[0:3], T[0:3];

always @(posedge rst)
begin
W[0] = 32'd0;
W[1] = 32'd1108;
W[2] = 32'd565;
W[3] = 32'd2408;
SW[0] = 32'd0;
SW[1] = 32'd1568;
SW[2] = 32'd2276;
SW[3] = 32'b11111111111111111111110011100001;
AW[0] = 32'd0;
AW[1] = 32'd3784;
AW[2] = 32'd3406;
AW[3] = 32'd4017;
end

always @(a or b or bf_select)
begin
tmp = W[bf_select[3:1]] * (a + b);
if (bf_select[0])
begin
mulvalue = b;
wvalue = SW[bf_select[3:1]];
end
else
begin
mulvalue = a;
wvalue = AW[bf_select[3:1]];
end
tmpvalue = wvalue * mulvalue;
if (bf_select[0])
tbf_out = tmp + tmpvalue;
else
tbf_out = tmp - tmpvalue;
end

assign bf_out = tbf_out;

endmodule
```

Figure 5.3: The Verilog description of the butterfly module.

below:
or1200_defines.v:

- Add *'define OR1200_RFWBOP_BUFL 4'b1001* to line 662. This defines the register file write-back operation code.
- Add *'define OR1200_OR32_BUFL 6'b111101* to line 745. This defines the identification for this instruction, along with signalling that two operands should be read from the register file.
- Modify line 656. Change the value of *'OR1200_RFWBOP_WIDTH* from 3 to 4.
- Change the values on line 657 to line 661 from *3'b* to *4'b0*.

or1200_cpu.v:

- Add *wire [3:0] bf_select;* to line 358. These are the wires that signal which operations the butterfly module should perform.
- Add *wire [31:0] bf_out;* to line 359. These are the wires that carry the output from the butterfly module.
- Add *.bf_select(bf_select)* to line 491. This maps the "bf_select" output from the control unit to the "bf_select" wires on the top level.
- Add *.muxin_e(bf_out)* to line 681. This maps the values from the "bf_out" wires to the input port "muxin_e" on the write-back mux.
- Add
  or1200_butterfly or1200_butterfly(
  .rst(rst),
  .bf_select(bf_select),
  .a(operand_a),
  .b(operand_b),
  .bf_out(bf_out)
  );
  to line 775. This maps the reset signal, "bf_select", the two operands and the output "bf_out" respectively to wires on the top level.

or1200_ctrl.v:

- Add *bf_select* to line 121. This adds the signal "bf_select" to the port list of the module.
- Add *output [3:0] bf_select;* to line 166. This defines "bf_select" as a 4-bit output signal.

- Add
  reg [3:0] bf_select;
  wire [3:0] tbf_select;

  //Butterfly select
  assign tbf_select = bf_select;

  always (posedge clk or posedge rst)
  if (rst)
  bf_select <= # 4'b0;
  else
  case(id_insn[31:26])
  'OR1200_OR32_BUFL:
  bf_select <= #1 id_insn[3:0];
  default:
  bf_select <= #1 tbf_select;
  endcase
  to line 202. This creates the register for "bf_select" and the wire "tbf_select" which is used for outputting the correct "bf_select" value. The rest is a process for selecting the "bf_select" value from the instruction - "id_insn".

- Add
  'OR1200_OR32_BUFL:
  sel_imm <= #1 1'b0;
  to line 569. This sets the "sel_imm" signal to 0 - this is not an immediate instruction.

- Add *'OR1200_OR32_BUFL,* to line 630. This makes sure that the signal except_illegal is set to 0.

- Add
  'OR1200_OR32_BUFL:
  rfwb_op <= #1 'OR1200_RFWBOP_BUFL;
  to line 661. This sends the correct signal to the register file write-back mux.

or1200_wbmux.v:

- Add *muxin_e* to line 83. This creates a new item in the port list of the module.

- Add *input [width-1:0] muxin_e;* to line 107. This defines "muxin_e" as a 32-bit input signal (width=31).

- Add
  3'b100 : begin
  muxout = muxin_e;

end
to line 167. This defines what will happend when the input signal is of type "'OR1200_RFWBOP_BUFL". The output from the mux will be set to "muxin_e".

- From line 142 to line 160, change every *2'b* to *3'b0*.

What is basically done is:

- Create a new bit-string to identify the new instructions.
- Draw a signal from the control unit to the butterfly module, signalling which operation to perform.
- Route the operands from the register file to the butterfly module.
- Create a new input in the write-back mux.
- Define a new bit-string that enables the write-back to route values from the new input port to the output port.
- Create a path from the output port on the butterfly module to the new input port on the write-back mux.

The rest of the functionality already exists in the OR1200, I have only created and routed signals to take advantage of that functionality. The only thing that remains is to put the Verilog code from section 5.2.1 into the project, and the new instructions are ready. The hardware bit, at least. The software part is presented in the next section.

5.3 Modifying the compiler and software

Actually, no modifications has to made to the compiler. These are three special-purpose instructions written only for the purpose of doing MPEG-2 decoding faster. The instructions can probably be used in any IDCT and therefor in most compression algorithms, but still the usage of the instructions is very limited. Due to this limitation, modifying the compiler means very little gain in most cases. If the OR1200 was meant as a general-purpose processor such as Intel or AMD processors, new instrucions would have had to been added to the compiler since most programmers code in a high-level programming language such as C/C++ and has only limited knowledge of the underlying architecture.

Instead of modifying the compiler, I am going to use inline assembly. This means adding assembly instructions directly into the C source code. In this case no modifications has to made to the compiler, the compiler just inserts the assembly instructions during the assembly stage. What is actually done is that the compiler takes the text inside the quotes of the asm statement and passes that text to the assembler. This can be clearly seen in figure 5.6 where the code from figure 5.5 has been run through the

assembly stage of the compiler. The line ”new instruction” has just been inserted into the assembly code. For inline assembly, the directive asm() can be used. This directive has four fields separated by colons:

- The name of the instruction along with which registers to use.
- The output operand.
- The input operand.
- The register(s) that have their values overwritten.

The syntax, along with an example can be seen in figure 5.4. In the example ”l.buf11” is the instruction to be executed, ”%0,%1,%2” are the output register and two input registers, respectively. ”t0” is the output variable and ”d0” and d1” are the input variables. ”=r” means that this is an write-only output value. ”r” means that the value can be stored in any general-purpose register. This information is from [San03].

From the assembly code it is the assemblers responsibility to produce binary code. If there are external references to functions, a linker will also be needed, but in this case I will not be making any modifications to the linker. But since the assembler is now required to recognize a new instruction, some modifications has to be made to it. The modifications have to be made to a file called ”or32-opc.c”. Fortunately there has been made room for new instructions in that file already, as seen in figure 5.7. Now all that has to be done is to insert the right bits where we want them.

- I want bits [31:26] starting with ”11” since the module needs two inputs from the register file (the 0xD, 0xE, 0xF just gives the instruction the right order).
- Bits [25:21] have the value ”DDDDD” to signal that they hold the address to the destination register.
- Bits [20:16] have the value ”AAAAA” to signal that they hold the address to one of the source registers.
- Bits [15:11] have the value ”BBBBB” to signal that they hold the address to the other source register.
- Bits [10:4] have no function so they are set to ”— —-”.
- Bits [3:1] signal which calculations to be performed in the new module so they are set to ”001”, ”010” or ”011” depending on which instruction is beeing executed.
- Bit [0] signals which of the values t0 or t1 (see figure 4.6) to output.

Since there are three different calls to the ”BUTTERFLY” function in the original IDCT and the output ”t0” and ”t1” are computed with two different instructions, this totals to six different instructions. The resulting definition of the six instructions can be seen in figure 5.8. So when I insert a ”l.buf11 r2, r3, r4” the assembler will automatically output the right bitcode for the instruction type, registers and value for selecting output. These are the instructions I will be using in section 5.4 where I describe the simulation and testing.

Six new instructions sound like a lot and this is actually the price for making the implementation as small and simple as possible while at the same time make only minor modifications to the processor core. I could have made half as many instructions, but that would have meant more complex hardware inside the module, and more modifications to the processor cores. The perhaps worst thing is that it would not have increased the speed, since it would still have taken two clock cycles to write to memory. Therefor I made the choice of adding a few more instructions and grealy simply the hardware, while maintaining the same performance.

```
asm ( assembler template
: output operands                   /* optional */
: input operands                    /* optional */
: list of clobbered registers       /* optional */
);

Example:
asm ("l.buf11 %0,%1,%2" : "=r" (t0) : "r" (d0), "r" (d1));
```

Figure 5.4: How to use the asm() directive.

```
int main() {
  int a;
  a = 0;
  asm("new instruction");
  return a;
}
```

Figure 5.5: Example of inline assembly.

Now that the instructions are in place it is time to build a test utility that can run some values through the new hardware. As said before I will be using inline assembly to invoke the instructions. The test program is a modified version of the ”hello-uart” program that can be found in the Opencores.org CVS tree. This is basically a simple utility for sending characters through the serial port on the development board. First

```
l.addi    r1,r1,-12
l.sw      0(r1),r2
l.addi    r2,r1,12
l.addi   r3,r0,0  # move immediate
l.sw     -4(r2),r3
new instruction
l.lwz     r3,-4(r2)  # SI load
l.sw     -8(r2),r3
l.lwz     r11,-8(r2)  # SI load
l.lwz     r2,0(r1)
l.jr   r9
l.addi    r1,r1,12
```

Figure 5.6: The resulting assembly code from the inline assembly example.

```
{ "l.cust6",   "",             "11 0xD  ----- ----- ---- ---- ---- ----", EFI,
  0, it_unknown },
{ "l.cust7",   "",             "11 0xE  ----- ----- ---- ---- ---- ----", EFI,
  0, it_unknown },
{ "l.cust8",   "",             "11 0xF  ----- ----- ---- ---- ---- ----", EFI,
  0, it_unknown },
```

Figure 5.7: The available custom instructions for the assembler.

```
{ "l.buf11",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0010", EFI,
  0, it_unknown },
{ "l.buf12",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0011", EFI,
  0, it_unknown },
{ "l.buf21",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0100", EFI,
  0, it_unknown },
{ "l.buf22",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0101", EFI,
  0, it_unknown },
{ "l.buf31",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0110-", EFI,
  0, it_unknown },
{ "l.buf32",   "rA,rB",        "11 0xD  DDDDD AAAAA BBBB B--- ---- 0111", EFI,
  0, it_unknown },
```

Figure 5.8: The definitions for the new butterfly instructions.

i will describe the modifications that has to be made to make the program run on the OR1200 that I have synthesized. These steps are also presented in [PDT03].

- Some modifications has to be made to two files.
- To ”board.h”: the only lines that have to remain are:

  # define IN_CLK 10000000

  # define STACK_SIZE 0x1000

  # define UART_BAUD_RATE 19200 (higher rates might not work)

  # define UART_BASE 0x90000000

  the lines with *# define REG8*, *REG16* and *REG32*
- The ”ram.ld” file has to be adjusted:

  ram : ORIGIN = 0x00002000, LENGTH = 0x00002000

Now the code that enables the new instructions has to be written. In the file ”hello.c” the ”main” function should look like figure 5.9. The ”asm” statements are used for passing assembly instructions directly to the assembler, the ”l.buf..” instructions are the ones that take advantage of the newly implemented hardware.

The entire modified version of the utility can be found in secton B.3.

When using the assembly instructions in the ”idct.c” file, the two assembly instructions that belong together replace one call to ”BUTTERFLY”, as shown in figure 5.10.

```
int main (void)
{
  char *s;
  char t;
  int d0=3,d1=3, shift=8,u;

  uart_init ();
  asm("l.buf11 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf11 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf12 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf12 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf21 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf21 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf22 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf22 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf31 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf31 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf32 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf32 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  return 0;
}
```

Figure 5.9: The ”main” function of the hello-uart program used for testing.

```
static void inline idct_row (int16_t * const block)
{
    int d0, d1, d2, d3;
    int a0, a1, a2, a3, b0, b1, b2, b3;
    int t0, t1, t2, t3;

    /* shortcut */
    if (likely (!(block[1] | ((int32_t *)block)[1] | ((int32_t *)block)[2] |
  ((int32_t *)block)[3]))) {
uint32_t tmp = (uint16_t) (block[0] >> 1);
tmp |= tmp << 16;
((int32_t *)block)[0] = tmp;
((int32_t *)block)[1] = tmp;
((int32_t *)block)[2] = tmp;
((int32_t *)block)[3] = tmp;
return;
    }

    d0 = (block[0] << 11) + 2048;
    d1 = block[1];
    d2 = block[2] << 11;
    d3 = block[3];
    t0 = d0 + d2;
    t1 = d0 - d2;
    // BUTTERFLY (t2, t3, W6, W2, d3, d1);
    asm("l.buf11 %0,%1,%2" : "=r" (t2) : "r" (d3), "r" (d1));
    asm("l.buf12 %0,%1,%2" : "=r" (t3) : "r" (d3), "r" (d1));
    a0 = t0 + t2;
    a1 = t1 + t3;
    a2 = t1 - t3;
    a3 = t0 - t2;

    d0 = block[4];
    d1 = block[5];
    d2 = block[6];
    d3 = block[7];
    // BUTTERFLY (t0, t1, W7, W1, d3, d0);
    asm("l.buf21 %0,%1,%2" : "=r" (t0) : "r" (d3), "r" (d0));
    asm("l.buf22 %0,%1,%2" : "=r" (t1) : "r" (d3), "r" (d0));
    // BUTTERFLY (t2, t3, W3, W5, d1, d2);
    asm("l.buf31 %0,%1,%2" : "=r" (t2) : "r" (d1), "r" (d2));
    asm("l.buf32 %0,%1,%2" : "=r" (t3) : "r" (d1), "r" (d2));
    b0 = t0 + t2;
    b3 = t1 + t3;
    t0 -= t2;
    t1 -= t3;
    b1 = ((t0 + t1) >> 8) * 181;
    b2 = ((t0 - t1) >> 8) * 181;

    block[0] = (a0 + b0) >> 12;
    block[1] = (a1 + b1) >> 12;
    block[2] = (a2 + b2) >> 12;
    block[3] = (a3 + b3) >> 12;
    block[4] = (a3 - b3) >> 12;
    block[5] = (a2 - b2) >> 12;
    block[6] = (a1 - b1) >> 12;
    block[7] = (a0 - b0) >> 12;
}
```

Figure 5.10: The new ”idct.c” with assembly instructions.

5.4 Simulation and testing

The process of testing was heavily based on the work done in [GH04]. Besides a lot of the work done in [PDT03] was used for the software part of the testing and [PDT04] was used for the hardware part of the testing. The simulation part was mainly done alongside the implementation to make sure that the different modules behaved the way they should. Modelsim SE 5.7g was used for simulation. A screenshot from the utility is shown in figure 5.13

[PDT04] describe a method for synthesizing a ”stripped” version of the OR1200 without DMMU, IMMU (Data and Instruction Memory Management Unit) and caches. The same procedure was used in [GH04], and I have used the same setup of the OR1200 in this case also. I will not cover the topic of setting up the OR1200 any further since it is very well described in those two reports.

[PDT03] describes a method for running a simple program on top of the synthesized OR1200 when running on a FPGA. I will use a modified version of that program in my testing.

The procedure of my testing was based on the work done in [PDT03] and [GH04] and is as follows:

- Synthesize the version of OR1200 described in [PDT04] with the modifications done in section 5.2.
- Upload the OR1200 to the a Microblaze development board. This process is well described in [GH04].
- Both of these processes are done in Windows. The rest of the steps are done in Linux.
- To run the test program the following ”packages” have to be installed:

  Binutils - includes the GNU assembler, linker and objdump to mention a few.

  GCC - the GNU compiler collection. Used to compile the testprogram.

  JTAG tool - to upload the test program to the onchip memory on the processor.

  Hello-uart - a simple program to output text via the COM port on the development board.
- The installation procedure is shown in figure 5.11.
- If the compilation and installation completes without errors it is time to make some modifications to the assembler. The changes are described in section 5.3 and will give the assembler capabilities to output code for the new instructions.

- Go back into the ”b-b” directory and just type ”make clean all install” to compile and install the binutils with the new modifications.
- To do the rest of the test you need three command-line windows.
- In the first you have to bring up the JTAG tool.
- Go into the ”jtag” directory and type ”./jp1 xpc3 9999”[2].
- The JTAG tool will then open port 9999 so that tool like GDB can connect.
- The output from the JTAG util should be like in figure 5.12. Carefully check that the last two digits of the values in the column ”ppc = ...” are the same as the once in the figure.
- In the second window you have to capture the output from the serial port.
- Type ”agetty -L 4800 ttyS0” to set the baud rate on the serial port to 4800. Then press CTRL-c to get out of the program.
- Type ”cat /dev/ttyS0” to capture the data on the serial port.
- In the third window you have to run GDB.
- Go into the ”hello-uart” directory and start GDB by typing ”or32-uclinux-gdb hello.or32”.
- Connect to the JTAG utility by typing ”target jtag jtag://localhost:9999”.
- Load the program by typing ”load”.
- Set the program counter by typing ”set $pc=0x100”.
- Start the program by typing ”c”
- The program should now start running on the OR1200 inside the FPGA, and the output on the second window should be ”Hello world”.
- Now that we have verified that the program works in the default setup, it is time to make some modifications to the ”hello.c” file to enable the new instructions.
- Section 5.3 gives a descriptions of the steps that should be taken.
- Recompile the hello-uart utility by going into the ”hello-uart” directory and type ”make clean all”.

[2]If you are using a different cable than the Xilinx Parallel Cable IV JTAG cable you should replace ”xpc3” with something else.

- Select the window that is capturing output from the serial port and cancel the capture by pressing CTRL-c.
- Start the capture again, but this time redirect the output to a file. Type ”cat /dev/ttyS0 > output.txt” to redirect the output to a file called ”output.txt”.
- Quit GDB and restart it by typing ”or32-uclinux-gdb hello.or32”.
- Then type ”target jtag jtag://localhost:9999”, ”load”, ”set $pc=0x100” and ”c” to run the program.
- Run ”hexdump output.txt” to get the contents of the file in the form of a hex dump.
- The output from the serial port should look like the one in figure 5.14.

The value pairs in figure 5.14 are switches, meaning that the value ”582c” is really ”2c58”. This is due to the fact that the program first outputs the first eight bits (bits [7:0] of the value, then right-shift it by eight bits and outputs the first eight bits again. By comparing the output from the instructions with the values calculated by the ”BUTTERFLY” function it is evident that the new instructions in fact work and output the correct values.

```
export CVSROOT=:pserver:anonymous@cvs.opencores.org:/cvsroot/anonymous
cvs login
cvs -z9 co or1k/binutil or1k/gcc-3.2.3 or1k/jtag or1k/hello-uart
cd or1k
mkdir b-b
cd b-b
./configure --target=or32-uclinux --prefix=/opt/or32-uclinux
make all install
cd ..
mkdir b-gcc
cd b-gcc
./configure --target=or32-uclinux --with-gnu-ld --with-gnu-as --verbose \
--enable-threads --prefix=/opt/or32-uclinux --local-prefix=/opt/or32-uclinux/or32-uclinux \
--enable-languages=c
make all install
cd ../jtag
make
cd ../hello-uart
make
```

Figure 5.11: Installation of the tools needed for testing.

```
Connected to parallel port at 378
Read      npc = 0000000c ppc = 00000024 r1 = 00000005
Expected  npc = 4000000c ppc = 40000024 r1 = 00000005
Read      npc = 0000000c ppc = 00000024 r1 = 00000008
Expected  npc = 4000000c ppc = 40000024 r1 = 00000008
Read      npc = 00000024 ppc = 00000020 r1 = 0000000b
Expected  npc = 40000024 ppc = 40000020 r1 = 0000000b
Read      npc = 00000020 ppc = 0000001c r1 = 00000018
Expected  npc = 40000020 ppc = 4000001c r1 = 00000018
Read      npc = 0000001c ppc = 00000018 r1 = 00000031
Expected  npc = 4000001c ppc = 40000018 r1 = 00000031
Read      npc = 00000020 ppc = 0000001c r1 = 00000032
Expected  npc = 40000020 ppc = 4000001c r1 = 00000032
Read      npc = 00000010 ppc = 0000000c r1 = 00000063
Expected  npc = 40000010 ppc = 4000000c r1 = 00000063
Read      npc = 00000024 ppc = 00000020 r1 = 00000065
Expected  npc = 40000024 ppc = 40000020 r1 = 00000065
Read      npc = 0000000c ppc = 00000024 r1 = 000000c9
Expected  npc = 4000000c ppc = 40000024 r1 = 000000c9
result = 5eaddead
Dropping root privileges.
JTAG Proxy server started on port 9999
Press CTRL+c to exit.
```

Figure 5.12: Output from the JTAG utility.

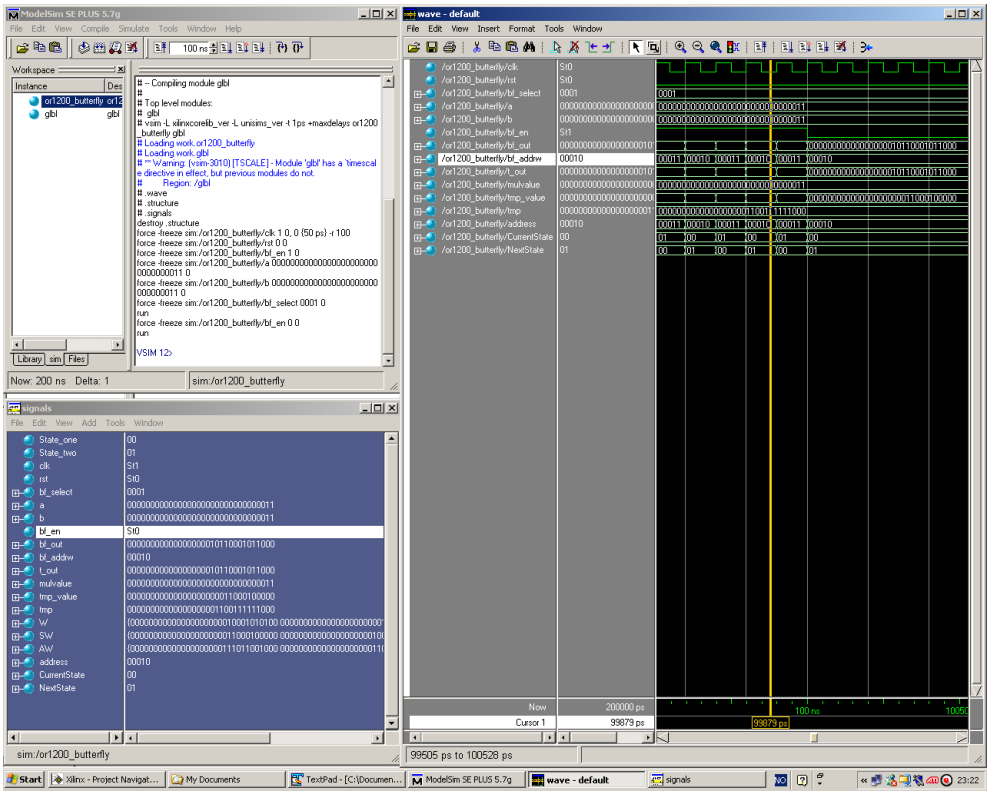

Figure 5.13: Simulation of an early version of the butterfly module on the Modelsim simulator.

```
0000000 582c a0ed ea27 54e5 132f 5d09
000000c
```

Figure 5.14: Output from the new test utility.

Chapter 6

Results

Since I have not implemented all three different methods, I will present the results from my implementation of the added instructions and results from simulation of the IO and DMA unit. All the simulations were run on the Linux system described in section 2.2, with the switches ”-c -o null” on the test utility for libmpeg2.

6.1 Added instructions

Compared to the other two methods for speedup, I have exact results for timing for the new instructions. Unfortunately I have not been able to compile a working executable of the libmpeg2 MPEG-2 decoder for OR1200 due to errors in the C libraries. Instead I have made a small program that runs on the processor that verifies that the instructions actually works for the FPGA implementation. The test process was presented in section 5.2.

To profile the new butterfly instructions I am going to use almost the same approach as in the rest of this chapter. I will replace the butterfly with NOP instructions. These instructions do nothing but guarantee that 1 clock cycle is spent doing nothing. So to simulate the instructions I insert 2 NOP instructions instead of the butterfly calls in the ”idct.c” file. To get the resulting ”t0” and ”t1” values, the program has to call two of the new butterfly instructions. That is why it takes two clock cycles. But compared to the 22 cycles the optimized version described in section 4.4.1 took, 2 cycles doing the same job is not so bad in comparison. The resulting output from the gprof profiling tool can be seen in figure 6.1. Compared to the results from the run with no modifications (see figure 4.1), the number of seconds spent in the IDCT has been reduced by 47%, the time spent on decoding the test movie has decreased by 20% and the number of frames decoded per second has increased 148 to 185 which is an increase of about 25%. If 400 MHz is assumed to be the limit for running MPEG-2 decoding real-time in software, the new instructions lower the speed to 320 MHz. The processing time (in percent) of each individual part of the decoder can be seen in figure 6.2. The increase in area is about 100 slices, which is really small as will soon be shown. Actually it is

an increase of 3,1% - the OR1200 version I have been using uses 3225 slices. Since I have only run the implementation on an FPGA I do not know the increase in mm^2 on an actual chip. Table 6.1 shows the actual harware resources used by the new instructions, the new processor and the original processor.

	New Instructions	New Processor	Original Processor
Slices	92	3362	3225
Slice Flip Flops	0	1946	1910
4 input LUTs	172	5933	5745
Bonded IOBs	99	10	10
BRAMs	0	36	36
MULT18X18s	6	10	4
GCLKs	0	3	3

Table 6.1: The hardware resources used by the instructions and the processor.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.61    120.76   120.76 28911026     0.00     0.00  mpeg2_idct_add_c
 14.04    186.93    66.17 28911026     0.00     0.00  get_non_intra_block
 11.68    242.01    55.08 11020620     0.00     0.00  mpeg2_idct_copy_c
  9.59    287.24    45.23 11020620     0.00     0.00  get_intra_block_B15
  8.81    328.78    41.54  7738491     0.01     0.01  MC_put_o_16_c
  6.35    358.74    29.96 21908410     0.00     0.00  MC_put_o_8_c
  5.18    383.15    24.41   295200     0.08     1.57  mpeg2_slice
  3.72    400.68    17.53  2415495     0.01     0.01  MC_put_xy_16_c
  2.65    413.16    12.48  1807921     0.01     0.01  MC_put_x_16_c
  1.99    422.55     9.39  1498165     0.01     0.01  MC_put_y_16_c
  1.91    431.54     8.99  6912398     0.00     0.01  motion_fr_frame_420
  1.64    439.27     7.73    79606     0.10     5.92  mpeg2_parse
  1.51    446.38     7.11  2424764     0.00     0.02  motion_fr_field_420
  0.78    450.08     3.70  2360896     0.00     0.00  MC_put_x_8_c
  0.72    453.48     3.40   346120     0.01     0.01  MC_avg_xy_16_c
  0.68    456.68     3.20   487038     0.01     0.01  MC_avg_o_16_c
  0.52    459.13     2.45  1157494     0.00     0.00  MC_put_xy_8_c
  0.46    461.30     2.17  1351582     0.00     0.00  MC_avg_o_8_c
  0.46    463.45     2.15  1493344     0.00     0.00  MC_put_y_8_c
  0.36    465.14     1.69   263845     0.01     0.01  MC_avg_x_16_c
  0.30    466.56     1.42   183296     0.01     0.01  MC_avg_y_16_c
  0.28    467.88     1.32   568662     0.00     0.00  MC_avg_x_8_c
  0.18    468.74     0.86   347614     0.00     0.00  MC_avg_y_8_c
  0.18    469.59     0.85   292740     0.00     0.00  MC_avg_xy_8_c
  0.16    470.34     0.75  1502578     0.00     0.01  motion_reuse_420
  0.15    471.04     0.70  1475867     0.00     0.01  motion_zero_420
```

Figure 6.1: Assembly code for the NOP loop used for testing.

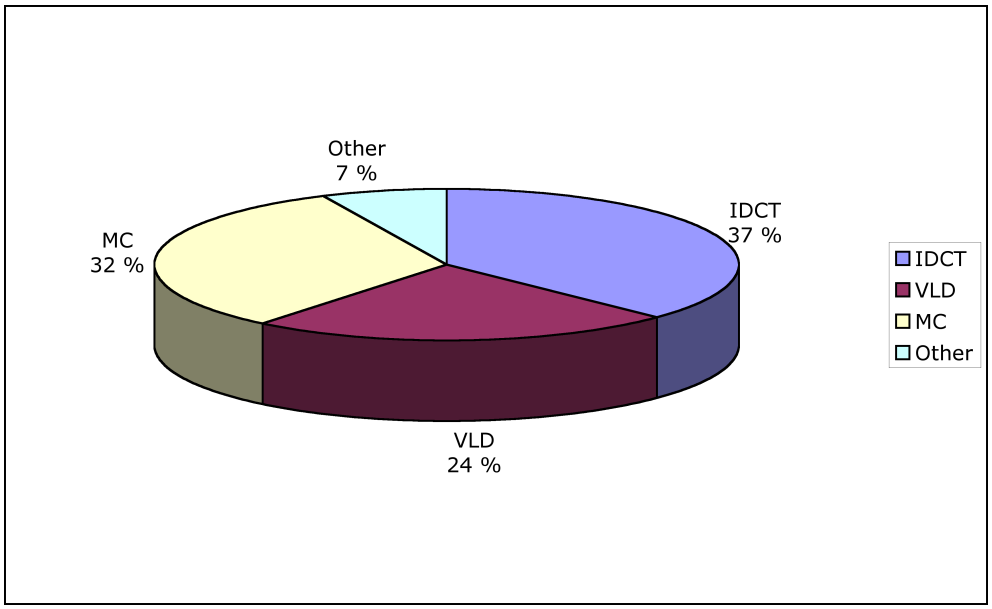

Figure 6.2: A diagram showing the execution time in percent of total execution time after implemeting the new instruction.

6.2 Simulation of IO unit

In the case of the IO unit, the processor has to move data back and forth between main memory and the memory on the IO unit. Since there is no DMA controller to insure correct access to memory, the processor has to do all the moving back and forth between memories.

[CVN99] states that each block of size 8x8 can be computed in 26 clock cycles. But since the row and column computations are pipelined, a new block can be fed to the module each 13th clock cycle. This ideally means that each 13th clock cycle the processor has to move 256 bytes[1] of data. According to [Lam01] each read from memory takes 2 clock cycles and each write takes 1 clock cycle. Let us also assume that this goes for the IO module, since the same load and store instructions will be used for reading and writing to the memory in the IO modules as to main memory. Assuming that each read cycle has to be completed before a write cycle can occur, the total number of cycles for moving the 256 bytes totals to 192 cycles[2]. On the other hand, if a store instruction can execute while a load instruction is execution, the average number of cycles per ”move” is 2:

- First load instruction.

[1]Two 8x8 matrix where each element is 16 bit long. $2 \cdot 8 \cdot 8 \cdot 16\ bit = 2048\ bit = 256\ bytes$.

[2]Each load/store can move 4 bytes of data. This means that 64 ”moves” has to occur. Since each move takes 3 clock cycles the total amounts to 192 cycles

- Second load instruction.
- First store instruction while second load instruction processing.
- Third load instruction.
- Second store instruction while third load instruction processing.
- ...

This optimistic calculation means that moving 256 bytes of data takes 128 clock cycles. But since I am interested in the worst-case scenario, I will use the estimate of 192 clock cycles. This means that the IDCT IO module will be inactive over 95% of the time since each stage of the pipeline takes only 13 clock cycles.

Based on these figures I have replaced the IDCT calculations with NOP instructions in a for loop. The assembly code can be seen in figure 6.3. The loop itself takes about 6 cycles, and exit and entry takes 2 cycles. Running the loop 33 times with the added 2 cycles for exit and entry, this equals to 200 cycles. The output from this simulated run, the frames generated, will of course not be correct when the IDCT is replaced by NOP instructions, but the elapsed time in each function will be correct. The number of NOP instructions are equal to the number of cycles it takes to move one block of data (256 bytes) back and forth between memories, plus added time for household operations such as signaling to IO module that new data are ready, processing interrups and so on. Estimating 1 cycle for signaling new data and 8 cycles for interrupt handling, the total number of NOPs becomes 201. Figure 4.1 shows part of the gprof log file from a run without any modifications to the IDCT. Figure 6.4 shows the gprof log file with the IDCT replaced by NOPs. In this simulation the IDCT functions (mpeg2_idct_add_test and mpeg2_idct_copy_test) have gone from using a total of 52.91% percent of the execution time, as shown in figure 4.1, to using only 2.62%, a 95%decrease in execution time. In section 2.3.1 the decode speed on the test movie was 148 frames per second, in this case this number increased to 282 frames per second, a 93% increase. The processing time (in percent) of each individual part of the decoder can be seen in figure 6.5. If 400 MHz is assumed to be the limit for running MPEG-2 decoding real-time, the new IO unit decreased the speed to 210 MHz.

The numbers presented are only from simulation and will most likely be a little different when the implementation in hardware is complete, but the simulation gives a good indication to how great the speedup will be.

When it comes to increase in area, it is quite significant. Only the Xilinx module for doing IDCT is using about 6500 slices. This is twice as many as the processor itself! In addition comes the memory for storing the blocks and the Wishbone interface. Totally this amounts to a 3-4 times increase in size. However, the IDCT module used in this case is very large and very effective. Even by slowing it down by a factor of 10 would still do the calculations faster than the bus could transfer the data. This would probably cut the amount of hardware required to implement the IDCT module dramatically and this is a factor that should be considered.

```
 3:nops.c        ****   for (i=0;i<100;i++) {
25                             .loc 1 3 0
26                     .LBB2:
27 0010 C745FC00               movl    $0, -4(%ebp)
27      000000
28                     .L2:
29 0017 837DFC63               cmpl    $99, -4(%ebp)
30 001b 7E02                   jle     .L5
31 001d EB08                   jmp     .L3
32                     .L5:
 4:nops.c        ****     asm("nop");
33                             .loc 1 4 0
34                     #APP
35 001f 90                     nop
36                             .loc 1 3 0
37                     #NO_APP
38 0020 8D45FC                 leal    -4(%ebp), %eax
39 0023 FF00                   incl    (%eax)
40 0025 EBF0                   jmp     .L2
41                     .L3:
 5:nops.c        ****   }
```

Figure 6.3: Assembly code for the NOP loop used for testing.

```
Flat profile:

Each sample counts as 0.01 seconds.
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 22.29     67.04    67.04 28911026     0.00     0.00  get_non_intra_block
 14.93    111.92    44.88 11020620     0.00     0.00  get_intra_block_B15
 13.49    152.49    40.57  7738491     0.01     0.01  MC_put_o_16_c
 10.27    183.37    30.88 21908410     0.00     0.00  MC_put_o_8_c
  7.55    206.07    22.70   295200     0.08     0.99  mpeg2_slice
  5.72    223.27    17.20  2415495     0.01     0.01  MC_put_xy_16_c
  4.29    236.17    12.90  1807921     0.01     0.01  MC_put_x_16_c
  3.03    245.29     9.12  1498165     0.01     0.01  MC_put_y_16_c
  2.78    253.64     8.35  6912398     0.00     0.01  motion_fr_frame_420
  2.44    260.98     7.34    79606     0.09     3.77  mpeg2_parse
  2.35    268.04     7.06  2424764     0.00     0.02  motion_fr_field_420
  1.95    273.91     5.87 28911026     0.00     0.00  mpeg2_idct_add_test
  1.23    277.61     3.70  2360896     0.00     0.00  MC_put_x_8_c
  1.07    280.83     3.22   346120     0.01     0.01  MC_avg_xy_16_c
  0.94    283.66     2.83   487038     0.01     0.01  MC_avg_o_16_c
  0.80    286.08     2.42  1351582     0.00     0.00  MC_avg_o_8_c
  0.77    288.39     2.31  1157494     0.00     0.00  MC_put_xy_8_c
  0.75    290.65     2.26  1493344     0.00     0.00  MC_put_y_8_c
  0.67    292.66     2.01 11020620     0.00     0.00  mpeg2_idct_copy_test
  0.55    294.31     1.65   263845     0.01     0.01  MC_avg_x_16_c
  0.54    295.93     1.62   183296     0.01     0.01  MC_avg_y_16_c
  0.40    297.14     1.21   568662     0.00     0.00  MC_avg_x_8_c
  0.28    297.98     0.84  1502578     0.00     0.01  motion_reuse_420
  0.28    298.81     0.83   292740     0.00     0.00  MC_avg_xy_8_c
  0.24    299.54     0.73   347614     0.00     0.00  MC_avg_y_8_c
  0.18    300.07     0.53  1475867     0.00     0.01  motion_zero_420
```

Figure 6.4: Logfile from gprof showing a simulation of the IO unit where IDCT has been replaced by NOPs.

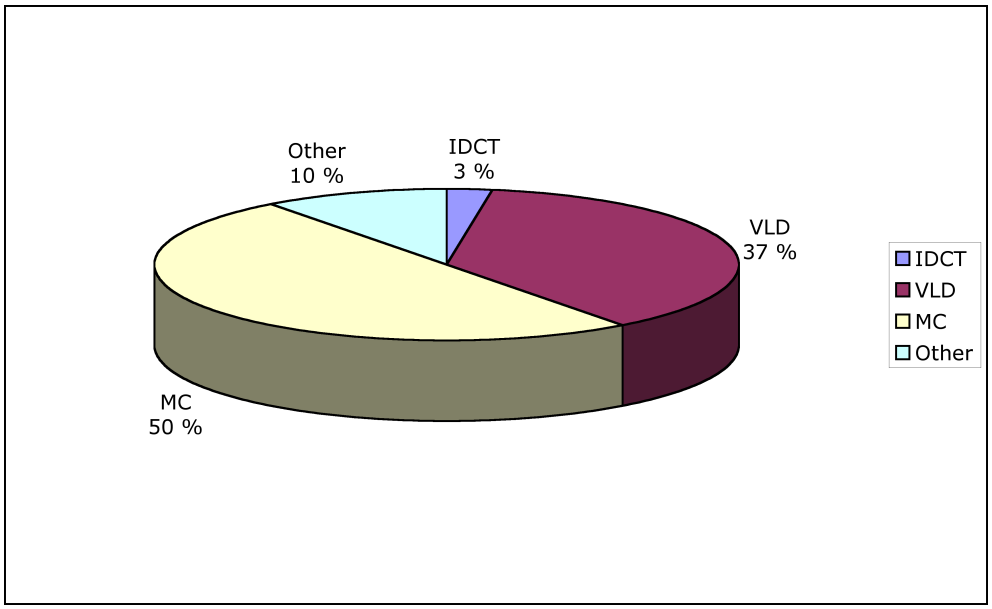

Figure 6.5: A diagram showing the execution time in percent of total execution time for the simulation of the IO unit.

6.3 Simulation of DMA unit

The simulation of the DMA unit will be done much the same way as the simulation of the IO unit. However calculating how many cycles the DMA unit will be blocking the CPU is more difficult. Since the DMA can access main memory directly, it does not need a memory of its own. It can simply fetch the needed data from main memory when it is needed. This means that the DMA unit is competing with the CPU for memory access. Since the CPU is the slowest of the two, it is a good idea to give the CPU priority over the DMA unit when it comes to conflicting memory accesses. This means that the CPU in most cases gets its data when it is needed, and the DMA unit has to wait until the CPU does operations that does not involve main memory.

In 4.4.2 I described how a DMA unit could be implemented. In this section I will do simulation using the simplest version; just connecting the DMA controller to the Wishbone bus, make it communicate with the memory controller and invent some scheme for making sure that memory violations[3] do not occur. A simple example of such a scheme can be:

- Two address variables are kept at predefined addresses in memory. The two variables point to the next block to be read and the next block to be written by the DMA unit. The block size is also predefined.
- At reset the CPU sets the variables to zero.
- At received interrupt from the DMA unit the CPU checks that the variables are zero.
- If the values of the variables are zero (and the reset signal is not set) they are set to point at two different blocks in memory.
- The CPU sets a register in the DMA unit to start reading from memory.
- The DMA unit reads the variables from memory, starts reading/writing and clears the read register.
- When all reads/writes are done, the DMA unit sets the adress variables to zero and issues an interrupt to the CPU.

Based on this scheme the number of cycles the DMA unit is blocking the CPU can be calculated in the following way: [Lam01] says that it takes 2 cycles to complete one load from memory. I will be using the same number since it is logical to assume that one read directly from memory takes 1 clock cycle, and according to [Her02] a total of 1 cycle is used for setup and transmission of the data. This is the number of cycles it takes to transfer one element (32 bit) of data. Since it is assumed that memory requests from CPU has higher priority than the requests from the DMA unit, each

[3]For instance memory that is read by one unit should not be written by another unit at the same time

memory request from the DMA unit will block the processor in one cycle. The reason for this is that most other instructions take only 1 cycle to complete[4]. A total of 128 cycles is then required to transfer each block back and forth between memories. This means that the processor will be blocked for 64 cycles for each block.

The reason for calculating only the cycles the DMA unit is blocking the CPU is simply that the DMA unit completes the IDCT calculations much faster than the CPU can do VLD, MC and general household tasks. It takes the IDCT module 26 clock cycles to complete one block if all data are available. In the same time the CPU can do only 13 loads, 8 multiplications or 26 arithmetic operations, for instance. For calculating speedup, it is the path that takes the longes time that is the important one, and since each cycle spent on communication with the DMA unit is blocking the CPU adds to this path, this is the number that is important.

Now that the number of blocking cycles is established I use the same method as in section 6.2 by replacing the calculations in the IDCT by NOPs. In this case I am running the for-loop 10 times and adding two NOP instructions after the loop has completed. The total number of cycles spent in the IDCT will then be 64 cycles. The resulting output from gprof can be seen in figure 6.6 and the processing time (in percent) of each individual part of the decoder can be seen in figure 6.7. In this case the number of frames decoded per second has increased from 282 in section 6.2 to 284. If 400 MHz is assumed to be the limit for running MPEG-2 decoding real-time, the new DMA unit has reduced the speed to 208 MHz.

The area of the unit is about the same as the IO unit without the DMA controller. With the DMA controller the area will probably increase significantly, but this depends heavily on the implementation of the DMA controller. [Uss02] does not give any number for the area of the implemented controller so it is hard to estimate an increase in ares. But given the marginal speedup, it is probably not worth the added design cost an complexity to create a DMA unit instead of an IO unit. But as with the IO unit, the IDCT module used in this case is very large. By reducing the speed by a factor of 10 the amount of hardware required to relize the module would shrink dramatically, while maintaining the same performace. This is due to the fact that the bus spends more time on moving data than the module spends on the calculations.

[4]The multiplication instruction is an exception, but there are very few multiplications in the code, so they can be disregarded

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 22.53     66.43    66.43 28911026     0.00     0.00  get_non_intra_block
 15.37    111.74    45.31 11020620     0.00     0.00  get_intra_block_B15
 13.76    152.32    40.58  7738491     0.01     0.01  MC_put_o_16_c
 10.16    182.28    29.96 21908410     0.00     0.00  MC_put_o_8_c
  7.54    204.50    22.22   295200     0.08     0.97  mpeg2_slice
  5.91    221.92    17.42  2415495     0.01     0.01  MC_put_xy_16_c
  4.20    234.29    12.37  1807921     0.01     0.01  MC_put_x_16_c
  2.98    243.07     8.78  1498165     0.01     0.01  MC_put_y_16_c
  2.95    251.76     8.69  6912398     0.00     0.01  motion_fr_frame_420
  2.65    259.58     7.82    79606     0.10     3.70  mpeg2_parse
  2.47    266.87     7.29  2424764     0.00     0.02  motion_fr_field_420
  1.17    270.33     3.46  2360896     0.00     0.00  MC_put_x_8_c
  0.98    273.23     2.90   346120     0.01     0.01  MC_avg_xy_16_c
  0.95    276.02     2.79 28911026     0.00     0.00  mpeg2_idct_add_test
  0.91    278.69     2.67   487038     0.01     0.01  MC_avg_o_16_c
  0.85    281.20     2.51  1493344     0.00     0.00  MC_put_y_8_c
  0.75    283.41     2.21  1351582     0.00     0.00  MC_avg_o_8_c
  0.75    285.62     2.21  1157494     0.00     0.00  MC_put_xy_8_c
  0.67    287.60     1.98   263845     0.01     0.01  MC_avg_x_16_c
  0.53    289.16     1.56   183296     0.01     0.01  MC_avg_y_16_c
  0.41    290.36     1.20   568662     0.00     0.00  MC_avg_x_8_c
  0.33    291.32     0.96 11020620     0.00     0.00  mpeg2_idct_copy_test
  0.30    292.20     0.88  1502578     0.00     0.01  motion_reuse_420
  0.29    293.05     0.85   347614     0.00     0.00  MC_avg_y_8_c
  0.26    293.82     0.77   292740     0.00     0.00  MC_avg_xy_8_c
  0.23    294.51     0.69  1475867     0.00     0.01  motion_zero_420
```

Figure 6.6: Logfile from gprof showing a simulation of the DMA unit where IDCT has been replaced by NOPs.

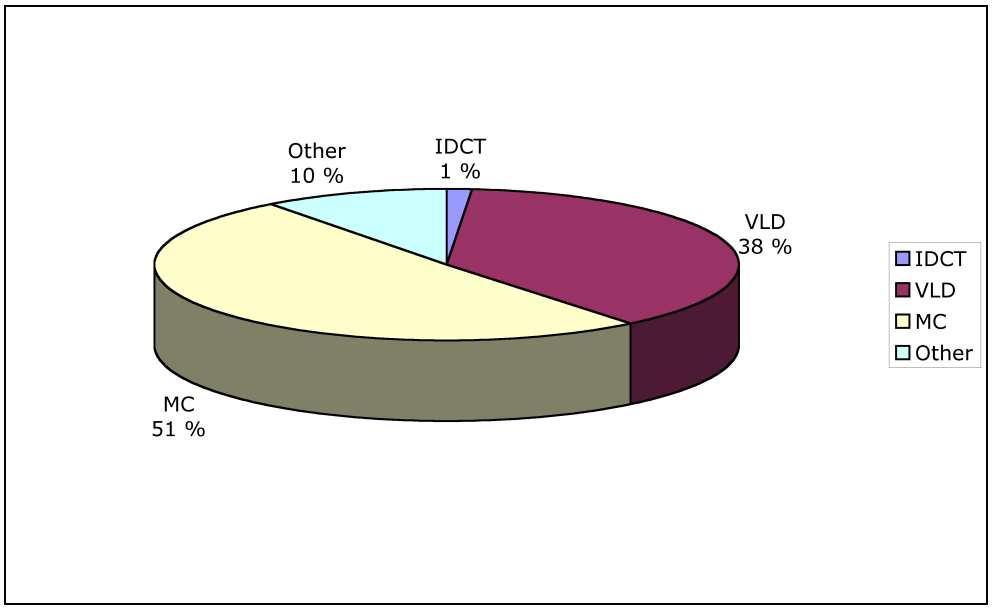

Figure 6.7: A diagram showing the execution time in percent of total execution time for the simulation of the DMA unit.

6.4 Comparison

The three methods I have described for enhancing the capabilites of the processor are very different. The IO and DMA unit will have nothing to do with the internals of the CPU, the unit will only see the Wisbone bus and the memory. The only connection to the CPU will be through an interrupt line. The new instruction, on the other hand, only deals with the internals of the processor and does not see anything on the outside. Actually the butterfly module is not ”aware” of anything but the control unit, the operand mux and the write-back mux. It does not see any memories, for instance, but takes advantage of some of the existing functionality inside the processor core, the register-file load and store, for instance. Due these differences the instructions and IO/DMA unit operate on totaly different granularity levels, and therefor has different capabilities when it comes to exploiting parallelism, for instaces. As seen in table 6.2 this results in totaly different speedups. It also leads to totally different hardware requirements and efforts when it comes to implementing the hardware. While adding a new instruction is very simple when the connections between the different modules in the processor has been figured out[5], it might be a much bigger task making an IO unit or DMA unit talk to the memory and processor through the Wishbone bus. And if a DMA controller has to be added also, this can be quite an undertaking. But from the table it is quite

[5]In fact, to figure out what each individual signal does in the core of the OR1200 is not such a simple task. Since the internals are undocumented at this level, of course, the function of the signals has to be figured out from the Verilog code.

clear that most can be gained when it comes to performance by implementing an IO or DMA unit. It has to be said, though, that the area required by the IO and DMA probably can be significantly reduced by decreasing the speed and complexity of the IDCT module, but this is an issue that will not be discussed further in this paper.

	Speedup	Area - slices	Decode time	Clock speed	Implementation effort
Original	-	3200	55.35s	400 MHz	-
Instruction	25%	∼3300-3400	44.18s	320 MHz	Medium
IO unit	91%	∼10000	29.00s	210 MHz	Medium - hard
DMA unit	92%	∼10000-13000	28.84s	208 MHz	Hard (with DMA controller)

Table 6.2: Properties of the three implementation methods I have described.

Regarding table 6.2, the speedup indicates the increased number of frames decoded per second, represented in percent. The decoded movie is the one described in section 2.3.1, specifically table 2.1 and the decode time in table 6.2 refers to the time it took to decode this movie. The area of the IO and DMA unit is a very rough estimate, and will probably change between FPGAs and synthesis tools. The figure for the DMA unit area is including the DMA controller. Without it the area of the DMA and IO unit would be roughly the same.

Chapter 7

Discussion

Why did I not implement the IO unit or DMA unit instead of six simple instructions? Both of them offer great advantages when it comes to speedup. The experiences gained from normal computers suggest so. That is, in fact, true, but on the other hand the OR1200 is not a Intel or AMD processor that fits into a normal computer. This is a processor that fits into a custom-built SoC and that is something entirely different. Before I started working on this report I had only [VZL] to support a gain from implementing an IO/DMA unit. And since the results presented in that report is partly[1] valid here also, I wanted to do something that had not been done before, and see what could be gained from implementing new instructions. Besides, in this report the steps leading up to the enhancements of the processor should be seen almost as important as the results, meaning that the experiences gained from implementing the instructions are of use when possibly implementing a IO or DMA unit in the future also.

However, the main reason for choosing to implement an instruction instead of an IO or DMA unit is that it fits better with the time span of this report. Meaning that when choosing to implement an instruction I had the following things in mind:

- I wanted somthing that actually worked when the report was completed.
- Implementing an IO or DMA unit takes longer time, most likely.
- I needed time to make thorough documentation of what I had done, and the background theory for those steps.
- I wanted to do something that gave me an even better understanding of the internals working of a processor.

Now that the new hardware is in place it is perhaps time to evaluate the usefulness of the OR1200 processor as a multimedia SoC. By the looks of chapter 8 I would say that it is not quite there yet. There are other alternatives that are tested and verified, and that are ready for implementing on an ASIC. This is a process that had to be

[1] Due to the fact that different decoders were used (most likely), the results are only partly valid.

completed with the OR1200 before it is ready for use. And this is one of the main disadvantages. The upside is that it is excellent for experimentation, and since the Verilog code is freely available, it is possible to make any kind of modifications. This would be hard with an IP from an IP vendor. But I still believe that OR1200 is not quite mature enough to be used by the industry. With recent developments (Linux release, for instance), it is getting closer.

Chapter 8

Further work

In order to make the OR1200 a good platform for media playback, there are a few steps that has to be completed:

- Run more thorough test to remove potential bugs.
- Make the C libraries work for OR1200. This might have been fixed by now, but the versions I have been using have not been working properly.
- Compile libmpeg2 for uClinux or Linux for OR1200.
- Do profiling of libmpeg2 while running on OR1200.
- Implement the IDCT in hardware.
- Implement the VLD in hardware.
- Possibly implement MC in hardware. This part is not as general as the two points above and would probably not work for other codecs, but would offer great speedups for MPEG-2 decoding.
- Optimize the source code for running on the OR1200 platform.

Completing these steps will most likely make the OR1200 platform a power efficient platform for portable and other devices where power usage is a critical factor. The IDCT and VLD will probably be used in audio and video codecs for many years to come, so if they are implemented at a general level, both present and future codecs will be able to use those resources.

Chapter 9

Conclusion

In this report I have presented three methods for hardware acceleration on the OR1200:

- Adding new instructions.
- Adding an IO unit.
- Adding a DMA unit.

The first one works in a totally different way than the other two, which are rather similar both in functionality and implementation. While the new module for realizing the new instructions interface with other modules inside the core of the processor, an IO or DMA unit interfaces at the Wishbone bus level. This makes their potential granularity level very different. The new instructions have to work on word level since it works only with registers. The IO and DMA units can work on arbitrary block sizes, although 8x8 matrixes would be preferable since this is the chosen block size for MPEG-2. This gives the IO and DMA unit a great advantage when it comes to taking advantage of block-level and macroblock-level parallelism.

By in fact taking advantage of block-level parallelism, the IO and DMA unit have an advantage over the instruction. This is evident when comparing execution times. While the new instructions had a decrease of 20%, both the IO and DMA unit had a decrease of about 48% when decoding a test movie. The number of frames decoded by the new instruction, the IO unit and the DMA unit per second increased by 25%, 91% and 92% respectively. This means that if the OR1200 has to run at 400 MHz to decode a MPEG-2 move real-time in software, without hardware acceleration, the OR1200 can run at

- 320 MHz with the new instructions.
- 210 MHz with an IO unit.
- 208 MHz with the DMA unit.

Today a 200MHz processor can be found in some advanced mobile phones. But because those processors only run at full speed a fraction of the time, the power consumption can be kept at low level. This is not the case with a portable media player, for instance. When running at a constant speed of 200 MHz, a battery found in a mobile phone or an MP3 player would not last very long. So to make OR1200 efficient enough it will probably be necessary to add other modules that perform parts of the decoding in hardware. The VLC is a good candidate, since it, along with the IDCT, is a component that is beeing used in a wide range of media compression technologies, and probably will be used in the nearest future also. The reason for not implementing MC is that this is the part of the decoding that is most likely to be changed when new codecs are invented.

When it comes to hardware cost, the new instruction is clearly the cheapest. While only adding about 3% to the area of the chip, this is by far the cheapest solution. Implementing the IDCT as an IO or DMA unit will increase the area by at least three times when using the IDCT module from Xilinx. This is a very drastic increase that most likely will have a great impact on power consumption also. The good news is that the IDCT most likely can be implemented much cheaper since the Xilinx IDCT module is way to fast compared to the time it takes to transfer the data over the bus. Less parallelism means that more resources can be reused, which means less hardware and lower power consumption.

However, there are some other issues besides the hardware that needs to be resolved before the OR1200 is a good platform for multimedia playback. The most important is perhaps to make it easier to compile program for the platform by making the C libraries work. This may have been corrected with new versions of Linux that were released while writing this report. If this point is resolved it would definitely make OR1200 a more attractive processor for both general-purpose use and for SoCs.

Appendix A

gprof log files

A.1 Initial run - no modifications

This is the gprof log file from a run where no modification has been made to the libmpeg2 decoder.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 36.06    227.09   227.09 28911026     0.01     0.01  mpeg2_idct_add_c
 16.69    332.21   105.12 11020620     0.01     0.01  mpeg2_idct_copy_c
 10.72    399.70    67.49 28911026     0.00     0.00  get_non_intra_block
  7.23    445.21    45.51 11020620     0.00     0.00  get_intra_block_B15
  6.63    486.98    41.77  7738491     0.01     0.01  MC_put_o_16_c
  4.83    517.42    30.44 21908410     0.00     0.00  MC_put_o_8_c
  4.00    542.60    25.18   295200     0.09     2.11  mpeg2_slice
  2.85    560.58    17.98  2415495     0.01     0.01  MC_put_xy_16_c
  1.95    572.87    12.29  1807921     0.01     0.01  MC_put_x_16_c
  1.41    581.78     8.91  1498165     0.01     0.01  MC_put_y_16_c
  1.26    589.72     7.94  6912398     0.00     0.01  motion_fr_frame_420
  1.20    597.27     7.55  2424764     0.00     0.02  motion_fr_field_420
  1.13    604.39     7.12    79606     0.09     7.91  mpeg2_parse
  0.59    608.13     3.74  2360896     0.00     0.00  MC_put_x_8_c
  0.52    611.42     3.29   346120     0.01     0.01  MC_avg_xy_16_c
  0.50    614.54     3.12   487038     0.01     0.01  MC_avg_o_16_c
  0.40    617.07     2.53  1157494     0.00     0.00  MC_put_xy_8_c
  0.37    619.43     2.36  1351582     0.00     0.00  MC_avg_o_8_c
  0.37    621.77     2.34  1493344     0.00     0.00  MC_put_y_8_c
  0.29    623.57     1.80   263845     0.01     0.01  MC_avg_x_16_c
  0.24    625.10     1.53   183296     0.01     0.01  MC_avg_y_16_c
  0.17    626.17     1.07   568662     0.00     0.00  MC_avg_x_8_c
  0.13    627.01     0.84   292740     0.00     0.00  MC_avg_xy_8_c
  0.13    627.80     0.79   347614     0.00     0.00  MC_avg_y_8_c
  0.12    628.53     0.73  1502578     0.00     0.01  motion_reuse_420
  0.10    629.18     0.65  1475867     0.00     0.01  motion_zero_420
  0.03    629.35     0.17    62055     0.00    10.15  decode_mpeg2
  0.02    629.48     0.13     8200     0.02     0.02  mpeg2_header_picture_finalize
  0.02    629.58     0.10     9413     0.01     0.04  mpeg2_parse_header
  0.01    629.62     0.04     8200     0.00     0.00  picture_coding_ext
```

```
  0.00    629.65     0.03                             main
  0.00    629.67     0.02    62055     0.00     0.00  mpeg2_info
  0.00    629.69     0.02     8200     0.00     0.00  mpeg2_header_picture
  0.00    629.71     0.02     8200     0.00     0.00  mpeg2_header_slice_start
  0.00    629.73     0.02     8200     0.00     0.00  mpeg2_init_fbuf
  0.00    629.75     0.02      575     0.03     0.03  finalize_matrix
  0.00    629.76     0.01     8775     0.00     0.01  mpeg2_header_extension
  0.00    629.77     0.01      575     0.02     0.02  mpeg2_header_sequence
  0.00    629.78     0.01      575     0.02     0.05  mpeg2_header_sequence_finalize
  0.00    629.78     0.00    62055     0.00     0.00  mpeg2_buffer
  0.00    629.78     0.00     8201     0.00     0.00  print_fps
  0.00    629.78     0.00     8200     0.00     0.00  mpeg2_header_picture_start
  0.00    629.78     0.00     8200     0.00     0.00  null_draw_frame
  0.00    629.78     0.00      575     0.00     0.00  mpeg2_header_gop
  0.00    629.78     0.00      575     0.00     0.00  mpeg2_header_gop_finalize
  0.00    629.78     0.00      575     0.00     0.00  sequence_ext
  0.00    629.78     0.00       20     0.00     0.00  mpeg2_malloc
  0.00    629.78     0.00       18     0.00     0.00  malloc_hook
  0.00    629.78     0.00       11     0.00     0.00  mpeg2_free
  0.00    629.78     0.00        2     0.00     0.00  mpeg2_header_state_init
  0.00    629.78     0.00        2     0.00     0.00  mpeg2_seek_header
  0.00    629.78     0.00        1     0.00     0.00  handle_args
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_accel
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_close
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_cpu_state_init
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_header_end
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_idct_init
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_init
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_malloc_hooks
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_mc_init
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_reset_info
  0.00    629.78     0.00        1     0.00     0.00  mpeg2_skip
  0.00    629.78     0.00        1     0.00     0.00  null_setup
  0.00    629.78     0.00        1     0.00     0.00  seek_sequence
  0.00    629.78     0.00        1     0.00     0.00  vo_drivers
  0.00    629.78     0.00        1     0.00     0.00  vo_null_open

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.

     Call graph (explanation follows)
```

```
granularity: each sample hit covers 4 byte(s) for 0.00% of 629.78 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.03  629.75                 main [1]
                0.17  629.58   62055/62055       decode_mpeg2 [2]
                0.00    0.00       1/1           handle_args [53]
                0.00    0.00       1/1           vo_null_open [67]
                0.00    0.00       1/1           mpeg2_init [59]
                0.00    0.00       1/1           mpeg2_malloc_hooks [60]
                0.00    0.00       1/11          mpeg2_free [50]
                0.00    0.00       1/1           mpeg2_close [55]
                0.00    0.00       1/8201        print_fps [42]
-----------------------------------------------
                0.17  629.58   62055/62055       main [1]
[2]    100.0    0.17  629.58   62055         decode_mpeg2 [2]
                7.12  622.44   79606/79606       mpeg2_parse [3]
                0.02    0.00   62055/62055       mpeg2_info [35]
                0.00    0.00   62055/62055       mpeg2_buffer [41]
                0.00    0.00    8200/8201        print_fps [42]
                0.00    0.00    8200/8200        null_draw_frame [44]
                0.00    0.00       1/1           null_setup [64]
                0.00    0.00       1/1           mpeg2_skip [63]
-----------------------------------------------
                7.12  622.44   79606/79606       decode_mpeg2 [2]
[3]    100.0    7.12  622.44   79606         mpeg2_parse [3]
               25.18  596.88  295200/295200      mpeg2_slice [4]
                0.10    0.24    9413/9413        mpeg2_parse_header [29]
                0.02    0.02    8200/8200        mpeg2_header_slice_start [32]
                0.00    0.00    8200/8200        mpeg2_header_picture_start [43]
                0.00    0.00       2/2           mpeg2_seek_header [52]
                0.00    0.00       1/1           seek_sequence [65]
                0.00    0.00       1/1           mpeg2_header_end [57]
-----------------------------------------------
               25.18  596.88  295200/295200      mpeg2_parse [3]
[4]     98.8   25.18  596.88  295200         mpeg2_slice [4]
              227.09    0.00 28911026/28911026     mpeg2_idct_add_c [5]
              105.12    0.00 11020620/11020620     mpeg2_idct_copy_c [6]
                7.94   62.39 6912398/6912398     motion_fr_frame_420 [7]
               67.49    0.00 28911026/28911026     get_non_intra_block [8]
                7.55   47.98 2424764/2424764     motion_fr_field_420 [9]
               45.51    0.00 11020620/11020620     get_intra_block_B15 [10]
                0.73   12.36 1502578/1502578     motion_reuse_420 [14]
                0.65   12.07 1475867/1475867     motion_zero_420 [15]
-----------------------------------------------
              227.09    0.00 28911026/28911026     mpeg2_slice [4]
[5]     36.1  227.09    0.00 28911026         mpeg2_idct_add_c [5]
-----------------------------------------------
              105.12    0.00 11020620/11020620     mpeg2_slice [4]
[6]     16.7  105.12    0.00 11020620         mpeg2_idct_copy_c [6]
-----------------------------------------------
                7.94   62.39 6912398/6912398     mpeg2_slice [4]
[7]     11.2    7.94   62.39 6912398         motion_fr_frame_420 [7]
               18.73    0.00 3470777/7738491     MC_put_o_16_c [11]
               16.13    0.00 11607630/21908410     MC_put_o_8_c [12]
                8.37    0.00 1124120/2415495     MC_put_xy_16_c [13]
                6.68    0.00  983239/1807921     MC_put_x_16_c [16]
                5.25    0.00  883032/1498165     MC_put_y_16_c [17]
                1.30    0.00  822288/2360896     MC_put_x_8_c [18]
                1.27    0.00  197785/487038      MC_avg_o_16_c [20]
                1.15    0.00  658672/1351582     MC_avg_o_8_c [22]
                0.88    0.00   92799/346120      MC_avg_xy_16_c [19]
                0.67    0.00   80101/183296      MC_avg_y_16_c [25]
```

```
                0.55    0.00   80545/263845      MC_avg_x_16_c [24]
                0.47    0.00  299514/1493344     MC_put_y_8_c [23]
                0.42    0.00  192904/1157494     MC_put_xy_8_c [21]
                0.29    0.00  155602/568662      MC_avg_x_8_c [26]
                0.12    0.00   52098/347614      MC_avg_y_8_c [28]
                0.10    0.00   36088/292740      MC_avg_xy_8_c [27]
-----------------------------------------------
               67.49    0.00 28911026/28911026     mpeg2_slice [4]
[8]     10.7   67.49    0.00 28911026         get_non_intra_block [8]
-----------------------------------------------
                7.55   47.98 2424764/2424764     mpeg2_slice [4]
[9]      8.8    7.55   47.98 2424764         motion_fr_field_420 [9]
                9.50    0.00 1276804/2415495     MC_put_xy_16_c [13]
                7.21    0.00 1335865/7738491     MC_put_o_16_c [11]
                6.06    0.00 4361952/21908410     MC_put_o_8_c [12]
                5.48    0.00  805942/1807921     MC_put_x_16_c [16]
                3.61    0.00  607073/1498165     MC_put_y_16_c [17]
                2.43    0.00 1532656/2360896     MC_put_x_8_c [18]
                2.41    0.00  253144/346120      MC_avg_xy_16_c [19]
                2.11    0.00  963990/1157494     MC_put_xy_8_c [21]
                1.87    0.00 1192770/1493344     MC_put_y_8_c [23]
                1.82    0.00  284734/487038      MC_avg_o_16_c [20]
                1.25    0.00  182960/263845      MC_avg_x_16_c [24]
                1.19    0.00  682692/1351582     MC_avg_o_8_c [22]
                0.86    0.00  103006/183296      MC_avg_y_16_c [25]
                0.78    0.00  412848/568662      MC_avg_x_8_c [26]
                0.74    0.00  256650/292740      MC_avg_xy_8_c [27]
                0.67    0.00  295498/347614      MC_avg_y_8_c [28]
-----------------------------------------------
               45.51    0.00 11020620/11020620     mpeg2_slice [4]
[10]     7.2   45.51    0.00 11020620         get_intra_block_B15 [10]
-----------------------------------------------
                7.21    0.00 1335865/7738491     motion_fr_field_420 [9]
                7.86    0.00 1455982/7738491     motion_reuse_420 [14]
                7.97    0.00 1475867/7738491     motion_zero_420 [15]
               18.73    0.00 3470777/7738491     motion_fr_frame_420 [7]
[11]     6.6   41.77    0.00 7738491         MC_put_o_16_c [11]
-----------------------------------------------
                4.10    0.00 2951734/21908410     motion_zero_420 [15]
                4.15    0.00 2987094/21908410     motion_reuse_420 [14]
                6.06    0.00 4361952/21908410     motion_fr_field_420 [9]
               16.13    0.00 11607630/21908410     motion_fr_frame_420 [7]
[12]     4.8   30.44    0.00 21908410         MC_put_o_8_c [12]
-----------------------------------------------
                0.11    0.00   14571/2415495     motion_reuse_420 [14]
                8.37    0.00 1124120/2415495     motion_fr_frame_420 [7]
                9.50    0.00 1276804/2415495     motion_fr_field_420 [9]
[13]     2.9   17.98    0.00 2415495         MC_put_xy_16_c [13]
-----------------------------------------------
                0.73   12.36 1502578/1502578     mpeg2_slice [4]
[14]     2.1    0.73   12.36 1502578         motion_reuse_420 [14]
                7.86    0.00 1455982/7738491     MC_put_o_16_c [11]
                4.15    0.00 2987094/21908410     MC_put_o_8_c [12]
                0.13    0.00   18740/1807921     MC_put_x_16_c [16]
                0.11    0.00   14571/2415495     MC_put_xy_16_c [13]
                0.05    0.00    8060/1498165     MC_put_y_16_c [17]
                0.03    0.00    4519/487038      MC_avg_o_16_c [20]
                0.02    0.00   10218/1351582     MC_avg_o_8_c [22]
                0.01    0.00    5952/2360896     MC_put_x_8_c [18]
                0.00    0.00     340/263845      MC_avg_x_16_c [24]
                0.00    0.00     177/346120      MC_avg_xy_16_c [19]
                0.00    0.00    1060/1493344     MC_put_y_8_c [23]
                0.00    0.00     189/183296      MC_avg_y_16_c [25]
                0.00    0.00     600/1157494     MC_put_xy_8_c [21]
                0.00    0.00     212/568662      MC_avg_x_8_c [26]
```

```
                0.00    0.00      18/347614      MC_avg_y_8_c [28]
                0.00    0.00       2/292740      MC_avg_xy_8_c [27]
-----------------------------------------------
                0.65   12.07 1475867/1475867     mpeg2_slice [4]
[15]     2.0    0.65   12.07 1475867         motion_zero_420 [15]
                7.97    0.00 1475867/7738491     MC_put_o_16_c [11]
                4.10    0.00 2951734/21908410     MC_put_o_8_c [12]
-----------------------------------------------
                0.13    0.00   18740/1807921     motion_reuse_420 [14]
                5.48    0.00  805942/1807921     motion_fr_field_420 [9]
                6.68    0.00  983239/1807921     motion_fr_frame_420 [7]
[16]     2.0   12.29    0.00 1807921         MC_put_x_16_c [16]
-----------------------------------------------
                0.05    0.00    8060/1498165     motion_reuse_420 [14]
                3.61    0.00  607073/1498165     motion_fr_field_420 [9]
                5.25    0.00  883032/1498165     motion_fr_frame_420 [7]
[17]     1.4    8.91    0.00 1498165         MC_put_y_16_c [17]
-----------------------------------------------
                0.01    0.00    5952/2360896     motion_reuse_420 [14]
                1.30    0.00  822288/2360896     motion_fr_frame_420 [7]
                2.43    0.00 1532656/2360896     motion_fr_field_420 [9]
[18]     0.6    3.74    0.00 2360896         MC_put_x_8_c [18]
-----------------------------------------------
                0.00    0.00     177/346120      motion_reuse_420 [14]
                0.88    0.00   92799/346120      motion_fr_frame_420 [7]
                2.41    0.00  253144/346120      motion_fr_field_420 [9]
[19]     0.5    3.29    0.00  346120         MC_avg_xy_16_c [19]
-----------------------------------------------
                0.03    0.00    4519/487038      motion_reuse_420 [14]
                1.27    0.00  197785/487038      motion_fr_frame_420 [7]
                1.82    0.00  284734/487038      motion_fr_field_420 [9]
[20]     0.5    3.12    0.00  487038         MC_avg_o_16_c [20]
-----------------------------------------------
                0.00    0.00     600/1157494     motion_reuse_420 [14]
                0.42    0.00  192904/1157494     motion_fr_frame_420 [7]
                2.11    0.00  963990/1157494     motion_fr_field_420 [9]
[21]     0.4    2.53    0.00 1157494         MC_put_xy_8_c [21]
-----------------------------------------------
                0.02    0.00   10218/1351582     motion_reuse_420 [14]
                1.15    0.00  658672/1351582     motion_fr_frame_420 [7]
                1.19    0.00  682692/1351582     motion_fr_field_420 [9]
[22]     0.4    2.36    0.00 1351582         MC_avg_o_8_c [22]
-----------------------------------------------
                0.00    0.00    1060/1493344     motion_reuse_420 [14]
                0.47    0.00  299514/1493344     motion_fr_frame_420 [7]
                1.87    0.00 1192770/1493344     motion_fr_field_420 [9]
[23]     0.4    2.34    0.00 1493344         MC_put_y_8_c [23]
-----------------------------------------------
                0.00    0.00     340/263845      motion_reuse_420 [14]
                0.55    0.00   80545/263845      motion_fr_frame_420 [7]
                1.25    0.00  182960/263845      motion_fr_field_420 [9]
[24]     0.3    1.80    0.00  263845         MC_avg_x_16_c [24]
-----------------------------------------------
                0.00    0.00     189/183296      motion_reuse_420 [14]
                0.67    0.00   80101/183296      motion_fr_frame_420 [7]
                0.86    0.00  103006/183296      motion_fr_field_420 [9]
[25]     0.2    1.53    0.00  183296         MC_avg_y_16_c [25]
-----------------------------------------------
                0.00    0.00     212/568662      motion_reuse_420 [14]
                0.29    0.00  155602/568662      motion_fr_frame_420 [7]
                0.78    0.00  412848/568662      motion_fr_field_420 [9]
[26]     0.2    1.07    0.00  568662         MC_avg_x_8_c [26]
-----------------------------------------------
                0.00    0.00       2/292740      motion_reuse_420 [14]
                0.10    0.00   36088/292740      motion_fr_frame_420 [7]
```

```
                0.74    0.00  256650/292740      motion_fr_field_420 [9]
[27]     0.1    0.84    0.00  292740          MC_avg_xy_8_c [27]
-----------------------------------------------
                0.00    0.00      18/347614      motion_reuse_420 [14]
                0.12    0.00   52098/347614      motion_fr_frame_420 [7]
                0.67    0.00  295498/347614      motion_fr_field_420 [9]
[28]     0.1    0.79    0.00  347614          MC_avg_y_8_c [28]
-----------------------------------------------
                0.10    0.24    9413/9413        mpeg2_parse [3]
[29]     0.1    0.10    0.24    9413          mpeg2_parse_header [29]
                0.13    0.00    8200/8200        mpeg2_header_picture_finalize [30]
                0.01    0.04    8775/8775        mpeg2_header_extension [31]
                0.01    0.02     575/575         mpeg2_header_sequence_finalize [34]
                0.02    0.00    8200/8200        mpeg2_header_picture [36]
                0.01    0.00     575/575         mpeg2_header_sequence [39]
                0.00    0.00     575/575         mpeg2_header_gop [45]
                0.00    0.00     575/575         mpeg2_header_gop_finalize [46]
-----------------------------------------------
                0.13    0.00    8200/8200        mpeg2_parse_header [29]
[30]     0.0    0.13    0.00    8200          mpeg2_header_picture_finalize [30]
                0.00    0.00       9/11          mpeg2_malloc <cycle 1> [48]
-----------------------------------------------
                0.01    0.04    8775/8775        mpeg2_parse_header [29]
[31]     0.0    0.01    0.04    8775          mpeg2_header_extension [31]
                0.04    0.00    8200/8200        picture_coding_ext [33]
                0.00    0.00     575/575         sequence_ext [47]
-----------------------------------------------
                0.02    0.02    8200/8200        mpeg2_parse [3]
[32]     0.0    0.02    0.02    8200          mpeg2_header_slice_start [32]
                0.02    0.00    8200/8200        mpeg2_init_fbuf [37]
-----------------------------------------------
                0.04    0.00    8200/8200        mpeg2_header_extension [31]
[33]     0.0    0.04    0.00    8200          picture_coding_ext [33]
-----------------------------------------------
                0.01    0.02     575/575         mpeg2_parse_header [29]
[34]     0.0    0.01    0.02     575          mpeg2_header_sequence_finalize [34]
                0.02    0.00     575/575         finalize_matrix [38]
-----------------------------------------------
                0.02    0.00   62055/62055       decode_mpeg2 [2]
[35]     0.0    0.02    0.00   62055          mpeg2_info [35]
-----------------------------------------------
                0.02    0.00    8200/8200        mpeg2_parse_header [29]
[36]     0.0    0.02    0.00    8200          mpeg2_header_picture [36]
-----------------------------------------------
                0.02    0.00    8200/8200        mpeg2_header_slice_start [32]
[37]     0.0    0.02    0.00    8200          mpeg2_init_fbuf [37]
-----------------------------------------------
                0.02    0.00     575/575         mpeg2_header_sequence_finalize [34]
[38]     0.0    0.02    0.00     575          finalize_matrix [38]
-----------------------------------------------
                0.01    0.00     575/575         mpeg2_parse_header [29]
[39]     0.0    0.01    0.00     575          mpeg2_header_sequence [39]
-----------------------------------------------
[40]     0.0    0.00    0.00      11+27       <cycle 1 as a whole> [40]
                0.00    0.00      20            mpeg2_malloc <cycle 1> [48]
-----------------------------------------------
                0.00    0.00   62055/62055       decode_mpeg2 [2]
[41]     0.0    0.00    0.00   62055          mpeg2_buffer [41]
-----------------------------------------------
                0.00    0.00       1/8201        main [1]
                0.00    0.00    8200/8201        decode_mpeg2 [2]
[42]     0.0    0.00    0.00    8201          print_fps [42]
-----------------------------------------------
                0.00    0.00    8200/8200        mpeg2_parse [3]
[43]     0.0    0.00    0.00    8200          mpeg2_header_picture_start [43]
```

```
-----------------------------------------------
                0.00    0.00    8200/8200        decode_mpeg2 [2]
[44]     0.0    0.00    0.00    8200         null_draw_frame [44]
-----------------------------------------------
                0.00    0.00     575/575         mpeg2_parse_header [29]
[45]     0.0    0.00    0.00     575         mpeg2_header_gop [45]
-----------------------------------------------
                0.00    0.00     575/575         mpeg2_parse_header [29]
[46]     0.0    0.00    0.00     575         mpeg2_header_gop_finalize [46]
-----------------------------------------------
                0.00    0.00     575/575         mpeg2_header_extension [31]
[47]     0.0    0.00    0.00     575         sequence_ext [47]
-----------------------------------------------
                                   9             malloc_hook <cycle 1> [49]
                0.00    0.00       2/11          mpeg2_init [59]
                0.00    0.00       9/11          mpeg2_header_picture_finalize [30]
[48]     0.0    0.00    0.00      20         mpeg2_malloc <cycle 1> [48]
                                  18             malloc_hook <cycle 1> [49]
-----------------------------------------------
                                  18             mpeg2_malloc <cycle 1> [48]
[49]     0.0    0.00    0.00      18         malloc_hook <cycle 1> [49]
                                   9             mpeg2_malloc <cycle 1> [48]
-----------------------------------------------
                0.00    0.00       1/11          main [1]
                0.00    0.00       1/11          mpeg2_close [55]
                0.00    0.00       9/11          seek_sequence [65]
[50]     0.0    0.00    0.00      11         mpeg2_free [50]
-----------------------------------------------
                0.00    0.00       1/2           mpeg2_init [59]
                0.00    0.00       1/2           mpeg2_close [55]
[51]     0.0    0.00    0.00       2         mpeg2_header_state_init [51]
-----------------------------------------------
                0.00    0.00       2/2           mpeg2_parse [3]
[52]     0.0    0.00    0.00       2         mpeg2_seek_header [52]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[53]     0.0    0.00    0.00       1         handle_args [53]
                0.00    0.00       1/1           vo_drivers [66]
                0.00    0.00       1/1           mpeg2_accel [54]
-----------------------------------------------
                0.00    0.00       1/1           handle_args [53]
[54]     0.0    0.00    0.00       1         mpeg2_accel [54]
                0.00    0.00       1/1           mpeg2_cpu_state_init [56]
                0.00    0.00       1/1           mpeg2_idct_init [58]
                0.00    0.00       1/1           mpeg2_mc_init [61]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[55]     0.0    0.00    0.00       1         mpeg2_close [55]
                0.00    0.00       1/2           mpeg2_header_state_init [51]
                0.00    0.00       1/11          mpeg2_free [50]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_accel [54]
[56]     0.0    0.00    0.00       1         mpeg2_cpu_state_init [56]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_parse [3]
[57]     0.0    0.00    0.00       1         mpeg2_header_end [57]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_accel [54]
[58]     0.0    0.00    0.00       1         mpeg2_idct_init [58]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[59]     0.0    0.00    0.00       1         mpeg2_init [59]
                0.00    0.00       2/11          mpeg2_malloc <cycle 1> [48]
                0.00    0.00       1/1           mpeg2_reset_info [62]
                0.00    0.00       1/2           mpeg2_header_state_init [51]
```

```
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[60]     0.0    0.00    0.00       1          mpeg2_malloc_hooks [60]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_accel [54]
[61]     0.0    0.00    0.00       1          mpeg2_mc_init [61]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_init [59]
[62]     0.0    0.00    0.00       1          mpeg2_reset_info [62]
-----------------------------------------------
                0.00    0.00       1/1           decode_mpeg2 [2]
[63]     0.0    0.00    0.00       1          mpeg2_skip [63]
-----------------------------------------------
                0.00    0.00       1/1           decode_mpeg2 [2]
[64]     0.0    0.00    0.00       1          null_setup [64]
-----------------------------------------------
                0.00    0.00       1/1           mpeg2_parse [3]
[65]     0.0    0.00    0.00       1          seek_sequence [65]
                0.00    0.00       9/11          mpeg2_free [50]
-----------------------------------------------
                0.00    0.00       1/1           handle_args [53]
[66]     0.0    0.00    0.00       1          vo_drivers [66]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[67]     0.0    0.00    0.00       1          vo_null_open [67]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function in the table.

% time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[20] MC_avg_o_16_c
[22] MC_avg_o_8_c
[24] MC_avg_x_16_c
[26] MC_avg_x_8_c
[19] MC_avg_xy_16_c
[27] MC_avg_xy_8_c
[25] MC_avg_y_16_c
[28] MC_avg_y_8_c
[11] MC_put_o_16_c
[12] MC_put_o_8_c
[16] MC_put_x_16_c
[18] MC_put_x_8_c
[13] MC_put_xy_16_c
[21] MC_put_xy_8_c
[17] MC_put_y_16_c
[23] MC_put_y_8_c
[2] decode_mpeg2
[9] motion_fr_field_420
[7] motion_fr_frame_420
[14] motion_reuse_420
[15] motion_zero_420
[54] mpeg2_accel
[41] mpeg2_buffer
[55] mpeg2_close
[56] mpeg2_cpu_state_init
[50] mpeg2_free
[57] mpeg2_header_end
[31] mpeg2_header_extension
[45] mpeg2_header_gop
[46] mpeg2_header_gop_finalize
[36] mpeg2_header_picture
[30] mpeg2_header_picture_finalize
[43] mpeg2_header_picture_start
[39] mpeg2_header_sequence
[35] mpeg2_info
[59] mpeg2_init
[37] mpeg2_init_fbuf
[48] mpeg2_malloc
[60] mpeg2_malloc_hooks
[61] mpeg2_mc_init
[3] mpeg2_parse
[29] mpeg2_parse_header
[62] mpeg2_reset_info
[52] mpeg2_seek_header
[63] mpeg2_skip
[4] mpeg2_slice
[44] null_draw_frame
[64] null_setup
[33] picture_coding_ext
[42] print_fps
[65] seek_sequence

```
[38] finalize_matrix        [34] mpeg2_header_sequence_finalize [47] sequence_ext
[10] get_intra_block_B15    [32] mpeg2_header_slice_start [66] vo_drivers
 [8] get_non_intra_block    [51] mpeg2_header_state_init [67] vo_null_open
[53] handle_args             [5] mpeg2_idct_add_c       [40] <cycle 1>
 [1] main                    [6] mpeg2_idct_copy_c
[49] malloc_hook            [58] mpeg2_idct_init
```

Appendix B

Source code

B.1 idct.c

```
/*
 * idct.c
 * Copyright (C) 2000-2003 Michel Lespinasse <walken@zoy.org>
 * Copyright (C) 1999-2000 Aaron Holtzman <aholtzma@ess.engr.uvic.ca>
 *
 * This file is part of mpeg2dec, a free MPEG-2 video stream decoder.
 * See http://libmpeg2.sourceforge.net/ for updates.
 *
 * mpeg2dec is free software; 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 2 of the License, or
 * (at your option) any later version.
 *
 * mpeg2dec is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY 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; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "config.h"

#include <stdlib.h>
#include <inttypes.h>

#include "mpeg2.h"
#include "attributes.h"
#include "mpeg2_internal.h"

#define W1 2841 /* 2048 * sqrt (2) * cos (1 * pi / 16) */
#define W2 2676 /* 2048 * sqrt (2) * cos (2 * pi / 16) */
#define W3 2408 /* 2048 * sqrt (2) * cos (3 * pi / 16) */
#define W5 1609 /* 2048 * sqrt (2) * cos (5 * pi / 16) */
#define W6 1108 /* 2048 * sqrt (2) * cos (6 * pi / 16) */
#define W7 565  /* 2048 * sqrt (2) * cos (7 * pi / 16) */

/* idct main entry point  */
void (* mpeg2_idct_copy) (int16_t * block, uint8_t * dest, int stride);
void (* mpeg2_idct_add) (int last, int16_t * block,
```

```
 uint8_t * dest, int stride);

/*
 * In legal streams, the IDCT output should be between -384 and +384.
 * In corrupted streams, it is possible to force the IDCT output to go
 * to +-3826 - this is the worst case for a column IDCT where the
 * column inputs are 16-bit values.
 */
uint8_t mpeg2_clip[3840 * 2 + 256];
#define CLIP(i) ((mpeg2_clip + 3840)[i])

#if 0
#define BUTTERFLY(t0,t1,W0,W1,d0,d1) \
do { \
    t0 = W0 * d0 + W1 * d1; \
    t1 = W0 * d1 - W1 * d0; \
} while (0)
#else
#define BUTTERFLY(t0,t1,W0,W1,d0,d1) \
do { \
    int tmp = W0 * (d0 + d1); \
    t0 = tmp + (W1 - W0) * d1; \
    t1 = tmp - (W1 + W0) * d0; \
} while (0)
#endif

static void inline idct_row (int16_t * const block)
{
    int d0, d1, d2, d3;
    int a0, a1, a2, a3, b0, b1, b2, b3;
    int t0, t1, t2, t3;

    /* shortcut */
    if (likely (!(block[1] | ((int32_t *)block)[1] | ((int32_t *)block)[2] |
  ((int32_t *)block)[3]))) {
uint32_t tmp = (uint16_t) (block[0] >> 1);
tmp |= tmp << 16;
((int32_t *)block)[0] = tmp;
((int32_t *)block)[1] = tmp;
((int32_t *)block)[2] = tmp;
((int32_t *)block)[3] = tmp;
return;
    }

    d0 = (block[0] << 11) + 2048;
    d1 = block[1];
    d2 = block[2] << 11;
    d3 = block[3];
    t0 = d0 + d2;
    t1 = d0 - d2;
    BUTTERFLY (t2, t3, W6, W2, d3, d1);
    a0 = t0 + t2;
    a1 = t1 + t3;
    a2 = t1 - t3;
    a3 = t0 - t2;

    d0 = block[4];
    d1 = block[5];
    d2 = block[6];
    d3 = block[7];
    BUTTERFLY (t0, t1, W7, W1, d3, d0);
    BUTTERFLY (t2, t3, W3, W5, d1, d2);
    b0 = t0 + t2;
    b3 = t1 + t3;
    t0 -= t2;
```

```
    t1 -= t3;
    b1 = ((t0 + t1) >> 8) * 181;
    b2 = ((t0 - t1) >> 8) * 181;

    block[0] = (a0 + b0) >> 12;
    block[1] = (a1 + b1) >> 12;
    block[2] = (a2 + b2) >> 12;
    block[3] = (a3 + b3) >> 12;
    block[4] = (a3 - b3) >> 12;
    block[5] = (a2 - b2) >> 12;
    block[6] = (a1 - b1) >> 12;
    block[7] = (a0 - b0) >> 12;
}

static void inline idct_col (int16_t * const block)
{
    int d0, d1, d2, d3;
    int a0, a1, a2, a3, b0, b1, b2, b3;
    int t0, t1, t2, t3;

    d0 = (block[8*0] << 11) + 65536;
    d1 = block[8*1];
    d2 = block[8*2] << 11;
    d3 = block[8*3];
    t0 = d0 + d2;
    t1 = d0 - d2;
    BUTTERFLY (t2, t3, W6, W2, d3, d1);
    a0 = t0 + t2;
    a1 = t1 + t3;
    a2 = t1 - t3;
    a3 = t0 - t2;

    d0 = block[8*4];
    d1 = block[8*5];
    d2 = block[8*6];
    d3 = block[8*7];
    BUTTERFLY (t0, t1, W7, W1, d3, d0);
    BUTTERFLY (t2, t3, W3, W5, d1, d2);
    b0 = t0 + t2;
    b3 = t1 + t3;
    t0 -= t2;
    t1 -= t3;
    b1 = ((t0 + t1) >> 8) * 181;
    b2 = ((t0 - t1) >> 8) * 181;

    block[8*0] = (a0 + b0) >> 17;
    block[8*1] = (a1 + b1) >> 17;
    block[8*2] = (a2 + b2) >> 17;
    block[8*3] = (a3 + b3) >> 17;
    block[8*4] = (a3 - b3) >> 17;
    block[8*5] = (a2 - b2) >> 17;
    block[8*6] = (a1 - b1) >> 17;
    block[8*7] = (a0 - b0) >> 17;
}

static void mpeg2_idct_copy_c (int16_t * block, uint8_t * dest,
       const int stride)
{
    int i;

    for (i = 0; i < 8; i++)
idct_row (block + 8 * i);
    for (i = 0; i < 8; i++)
idct_col (block + i);
    do {
```

```
dest[0] = CLIP (block[0]);
dest[1] = CLIP (block[1]);
dest[2] = CLIP (block[2]);
dest[3] = CLIP (block[3]);
dest[4] = CLIP (block[4]);
dest[5] = CLIP (block[5]);
dest[6] = CLIP (block[6]);
dest[7] = CLIP (block[7]);

((int32_t *)block)[0] = 0; ((int32_t *)block)[1] = 0;
((int32_t *)block)[2] = 0; ((int32_t *)block)[3] = 0;

dest += stride;
block += 8;
    } while (--i);
}

static void mpeg2_idct_add_c (const int last, int16_t * block,
      uint8_t * dest, const int stride)
{
    int i;

    if (last != 129 || (block[0] & (7 << 4)) == (4 << 4)) {
for (i = 0; i < 8; i++)
    idct_row (block + 8 * i);
for (i = 0; i < 8; i++)
    idct_col (block + i);
do {
    dest[0] = CLIP (block[0] + dest[0]);
    dest[1] = CLIP (block[1] + dest[1]);
    dest[2] = CLIP (block[2] + dest[2]);
    dest[3] = CLIP (block[3] + dest[3]);
    dest[4] = CLIP (block[4] + dest[4]);
    dest[5] = CLIP (block[5] + dest[5]);
    dest[6] = CLIP (block[6] + dest[6]);
    dest[7] = CLIP (block[7] + dest[7]);

    ((int32_t *)block)[0] = 0; ((int32_t *)block)[1] = 0;
    ((int32_t *)block)[2] = 0; ((int32_t *)block)[3] = 0;

    dest += stride;
    block += 8;
} while (--i);
    } else {
int DC;

DC = (block[0] + 64) >> 7;
block[0] = block[63] = 0;
i = 8;
do {
    dest[0] = CLIP (DC + dest[0]);
    dest[1] = CLIP (DC + dest[1]);
    dest[2] = CLIP (DC + dest[2]);
    dest[3] = CLIP (DC + dest[3]);
    dest[4] = CLIP (DC + dest[4]);
    dest[5] = CLIP (DC + dest[5]);
    dest[6] = CLIP (DC + dest[6]);
    dest[7] = CLIP (DC + dest[7]);
    dest += stride;
} while (--i);
    }
}

void mpeg2_idct_init (uint32_t accel)
{
```

```
#ifdef ARCH_X86
    if (accel & MPEG2_ACCEL_X86_MMXEXT) {
mpeg2_idct_copy = mpeg2_idct_copy_mmxext;
mpeg2_idct_add = mpeg2_idct_add_mmxext;
mpeg2_idct_mmx_init ();
    } else if (accel & MPEG2_ACCEL_X86_MMX) {
mpeg2_idct_copy = mpeg2_idct_copy_mmx;
mpeg2_idct_add = mpeg2_idct_add_mmx;
mpeg2_idct_mmx_init ();
    } else
#endif
#ifdef ARCH_PPC
    if (accel & MPEG2_ACCEL_PPC_ALTIVEC) {
mpeg2_idct_copy = mpeg2_idct_copy_altivec;
mpeg2_idct_add = mpeg2_idct_add_altivec;
mpeg2_idct_altivec_init ();
    } else
#endif
#ifdef ARCH_ALPHA
    if (accel & MPEG2_ACCEL_ALPHA_MVI) {
mpeg2_idct_copy = mpeg2_idct_copy_mvi;
mpeg2_idct_add = mpeg2_idct_add_mvi;
mpeg2_idct_alpha_init ();
    } else if (accel & MPEG2_ACCEL_ALPHA) {
int i;

mpeg2_idct_copy = mpeg2_idct_copy_alpha;
mpeg2_idct_add = mpeg2_idct_add_alpha;
mpeg2_idct_alpha_init ();
for (i = -3840; i < 3840 + 256; i++)
    CLIP(i) = (i < 0) ? 0 : ((i > 255) ? 255 : i);
    } else
#endif
    {
extern uint8_t mpeg2_scan_norm[64];
extern uint8_t mpeg2_scan_alt[64];
int i, j;

mpeg2_idct_copy = mpeg2_idct_copy_c;
mpeg2_idct_add = mpeg2_idct_add_c;
for (i = -3840; i < 3840 + 256; i++)
    CLIP(i) = (i < 0) ? 0 : ((i > 255) ? 255 : i);
for (i = 0; i < 64; i++) {
    j = mpeg2_scan_norm[i];
    mpeg2_scan_norm[i] = ((j & 0x36) >> 1) | ((j & 0x09) << 2);
    j = mpeg2_scan_alt[i];
    mpeg2_scan_alt[i] = ((j & 0x36) >> 1) | ((j & 0x09) << 2);
}
    }
}
```

B.2 Assembly output of the butterfly

```
GAS LISTING /home/openrisc/tmp/ccdfmGVO.s  page 1


   1              .file "butterfly.c"
   4              .text
   5              .Ltext0:
  28              .align 4
  29              .proc _main
```

```
  33                .global _main
  35                _main:
   1:butterfly.c    **** int main(int argc,char* args[]) {
  37                .LM1:
  38
  39                # gpr_save_area 0 vars 56 current_function_outgoing_args_size 0
  40 0000 9C21FFC4  l.addi    r1,r1,-60
  41 0004 D4011000  l.sw      0(r1),r2
  42 0008 9C41003C  l.addi    r2,r1,60
  43 000c D7E21FFC  l.sw     -4(r2),r3
  44 0010 D7E227F8  l.sw     -8(r2),r4
  45                .LBB2:
   2:butterfly.c    ****   int W0,W1,d0,d1,t0,t1;
   3:butterfly.c    ****
   4:butterfly.c    ****   int tmp = W0 * (d0 + d1);
  47                .LM2:
  48 0014 8462FFEC  l.lwz     r3,-20(r2) # Load d0
  49 0018 8482FFE8  l.lwz     r4,-24(r2) # Load d1
  50 001c E0632000  l.add    r3,r3,r4 # Do addition (d0 + d1)
  51 0020 D7E21FD8  l.sw     -40(r2),r3 # Store (d0 + d1)
  52 0024 8462FFF4  l.lwz     r3,-12(r2) # Load W0
  53 0028 8482FFD8  l.lwz     r4,-40(r2) # Load (d0 + d1)
  54 002c E0632306  l.mul    r3,r3,r4 # Do multiplication W0 * (d0 + d1)
  55 0030 D7E21FDC  l.sw     -36(r2),r3 # Store tmp (W0 * (d0 + d1))
   5:butterfly.c    ****   t0 = tmp + (W1 - W0) * d1;
  57                .LM3:
  58 0034 8462FFF0  l.lwz     r3,-16(r2) # Load W1
  59 0038 8482FFF4  l.lwz     r4,-12(r2) # Load W0
  60 003c E0632002  l.sub    r3,r3,r4 # Do subtraction (W1 - W0)
  61 0040 D7E21FD4  l.sw     -44(r2),r3 # Store (W1 - W0)
  62 0044 8462FFD4  l.lwz     r3,-44(r2) # Load (W1 - W0)
  63 0048 8482FFE8  l.lwz     r4,-24(r2) # Load d1
  64 004c E0632306  l.mul    r3,r3,r4 # Do multiplication (W1 - W0) * d1
  65 0050 D7E21FD0  l.sw     -48(r2),r3 # Store (W1 - W0) * d1
  66 0054 8462FFDC  l.lwz     r3,-36(r2) # Load tmp (W0 * (d0 + d1))
  67 0058 8482FFD0  l.lwz     r4,-48(r2) # Load (W1 - W0) * d1
  68 005c E0632000  l.add    r3,r3,r4 # Do addition tmp + (W1 - W0) * d1
  69 0060 D7E21FE4  l.sw     -28(r2),r3 # Store t0 (tmp + (W1 - W0) * d1)
   6:butterfly.c    ****   t1 = tmp - (W1 + W0) * d0;
  71                .LM4:
  72 0064 8462FFF0  l.lwz     r3,-16(r2) # Load W1
  73 0068 8482FFF4  l.lwz     r4,-12(r2) # Load W0
  74 006c E0632000  l.add    r3,r3,r4 # Do addition (W1 + W0)
  75 0070 D7E21FCC  l.sw     -52(r2),r3 # Store (W1 + W0)
  76 0074 8462FFCC  l.lwz     r3,-52(r2) # Load (W1 + W0)
  77 0078 8482FFEC  l.lwz     r4,-20(r2) # Load d0
  78 007c E0632306  l.mul    r3,r3,r4 # Do multiplication (W1 + W0) * d0
  79 0080 D7E21FC8  l.sw     -56(r2),r3 # Store (W1 + W0) * d0
  80 0084 8462FFDC  l.lwz     r3,-36(r2) # Load tmp (W0 * (d0 + d1))
  81 0088 8482FFC8  l.lwz     r4,-56(r2) # Load (W1 + W0) * d0
  82 008c E0632002  l.sub    r3,r3,r4 # Do subtraction tmp - (W1 + W0) * d0
  83 0090 D7E21FE0  l.sw     -32(r2),r3 # Store t0 (tmp - (W1 + W0) * d0)

GAS LISTING /home/openrisc/tmp/ccdfmGVO.s  page 2


   7:butterfly.c    ****   return 0;
  85                .LM5:
  86                .LBE2:
   8:butterfly.c    **** }
  88                .LM6:
  89 0094 9D600000  l.addi   r11,r0,0  # move immediate
  90 0098 84410000  l.lwz     r2,0(r1)
  91 009c 44004800  l.jr   r9
  92 00a0 9C21003C  l.addi    r1,r1,60
```

```
  93               .endproc _main
  94               .Lfe1:
 105               .Lscope0:
 107               .text
 109               Letext:
 110               .ident "GCC: (GNU) 3.2.3"

GAS LISTING /home/openrisc/tmp/ccdfmGVO.s  page 3


DEFINED SYMBOLS
                            *ABS*:00000000 butterfly.c
/home/openrisc/tmp/ccdfmGVO.s:35     .text:00000000 _main
/home/openrisc/tmp/ccdfmGVO.s:109    .text:000000a4 Letext

NO UNDEFINED SYMBOLS
```

B.3 hello-uart

```
#include "board.h"
#include "uart.h"

#define BOTH_EMPTY (UART_LSR_TEMT | UART_LSR_THRE)

#define WAIT_FOR_XMITR \
        do { \
                lsr = REG8(UART_BASE + UART_LSR); \
        } while ((lsr & BOTH_EMPTY) != BOTH_EMPTY)

#define WAIT_FOR_THRE \
        do { \
                lsr = REG8(UART_BASE + UART_LSR); \
        } while ((lsr & UART_LSR_THRE) != UART_LSR_THRE)

#define CHECK_FOR_CHAR (REG8(UART_BASE + UART_LSR) & UART_LSR_DR)

#define WAIT_FOR_CHAR \
         do { \
                lsr = REG8(UART_BASE + UART_LSR); \
         } while ((lsr & UART_LSR_DR) != UART_LSR_DR)

void uart_init(void)
{
  int divisor;

  /* Reset receiver and transmiter */
  REG8(UART_BASE + UART_FCR) = UART_FCR_ENABLE_FIFO | UART_FCR_CLEAR_RCVR | UART_FCR_CLEAR_XMIT | UART_FCR_

  /* Disable all interrupts */
  REG8(UART_BASE + UART_IER) = 0x00;

  /* Set 8 bit char, 1 stop bit, no parity */
  REG8(UART_BASE + UART_LCR) = UART_LCR_WLEN8 & ~(UART_LCR_STOP | UART_LCR_PARITY);

  /* Set baud rate */
  divisor = IN_CLK/(16 * UART_BAUD_RATE);
  REG8(UART_BASE + UART_LCR) |= UART_LCR_DLAB;
  REG8(UART_BASE + UART_DLL) = divisor & 0x000000ff;
  REG8(UART_BASE + UART_DLM) = (divisor >> 8) & 0x000000ff;
  REG8(UART_BASE + UART_LCR) &= ~(UART_LCR_DLAB);
}
```

```
void uart_putc(char c)
{
  unsigned char lsr;

  WAIT_FOR_THRE;
  REG8(UART_BASE + UART_TX) = c;
  if(c == '\n') {
    WAIT_FOR_THRE;
    REG8(UART_BASE + UART_TX) = '\r';
  }
  WAIT_FOR_XMITR;
}

char uart_getc(void)
{
  unsigned char lsr;
  char c;

  WAIT_FOR_CHAR;
  c = REG8(UART_BASE + UART_RX);
  return c;
}

char *str = "Hello world!!!\n";
int main (void)
{
  char *s;
  char t;
  int d0=3,d1=3, shift=8,u;

  uart_init ();
  asm("l.buf11 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf11 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf12 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf12 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf21 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf21 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf22 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf22 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf31 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf31 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  asm("l.buf32 %0,%1,%2" : "=r" (t) : "r" (d0), "r" (d1));
  uart_putc (t);
  asm("l.buf32 %0,%1,%2" : "=r" (u) : "r" (d0), "r" (d1));
  asm("l.srl %0,%1,%2" : "=r" (t) : "r" (u), "r" (shift));
  uart_putc (t);
  return 0;
}
```

Appendix C

HDL files

C.1 Butterfly implementation

```
module or1200_butterfly(rst,bf_select,a,b,bf_out);

input rst;
input [3:0] bf_select;
input [31:0] a;
input [31:0] b;
output [31:0] bf_out;

reg [31:0] tmp, mulvalue, wvalue, tmpvalue, tbf_out;
reg [31:0] W[0:3], SW[0:3], AW[0:3], T[0:3];

always @(posedge rst)
begin
W[0] = 32'd0;
W[1] = 32'd1108;
W[2] = 32'd565;
W[3] = 32'd2408;
SW[0] = 32'd0;
SW[1] = 32'd1568;
SW[2] = 32'd2276;
SW[3] = 32'b11111111111111111111110011100001;
AW[0] = 32'd0;
AW[1] = 32'd3784;
AW[2] = 32'd3406;
AW[3] = 32'd4017;
end

always @(a or b or bf_select)
begin
tmp = W[bf_select[3:1]] * (a + b);
if (bf_select[0])
begin
mulvalue = b;
wvalue = SW[bf_select[3:1]];
end
else
begin
mulvalue = a;
wvalue = AW[bf_select[3:1]];
end
tmpvalue = wvalue * mulvalue;
if (bf_select[0])
```

```
tbf_out = tmp + tmpvalue;
else
tbf_out = tmp - tmpvalue;
end

assign bf_out = tbf_out;

endmodule
```

Bibliography

[CLR00] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. 2000.

[CVN99] K. Chaudhary, H. Verma, and S. Nag. *An Inverse Discrete Cosine Transform (IDCT) Implementation in Virtex for MPEG Video Applications*, December 1999.

[GH04] Olav Gulling and Morten Haugseggen. *Synthesis of 32 bit microcontroller*, September 2004.

[Her02] Richard Herveille. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores Organization, September 2002.

[ISO95] ISO/IEC 1995. *ISO/IEC 13818-2: 1995 (E)*, 1995.

[Lam01] Damjan Lampret. *OpenRisc 1200 IP Core Specification*, September 2001.

[LCM+04] Damjan Lampret, Chen-Min Chen, Marko Mlinar, Johan Rydberg, Matan Ziv-Av, Chris Ziomkowski, Greg McGary, Bob Gardner, Rohit Mathur, and Maria Bolado. *OpenRISC 1000 Architecture Manual*. OPENCORES.ORG, August 2004.

[NVT+02] Jari Nikara, Stamatis Vassiliadis, Jarmo Takala, Mihai Sima, and Petri Liuha. *Parallel Multiple-Symbol Variable-Length Decoding*, 2002.

[PDT03] Patrick Pelgrims, Dries Driessens, and Tom Tierens. *Basic Custom OpenRISC system Software Tutorial*, 2003.

[PDT04] Patrick Pelgrims, Dries Driessens, and Tom Tierens. *Basic Custom OpenRISC System Hardware Tutorial*, 2004.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the hardware/software interface*. Morgan Kaufmann Publishers Inc., 1998.

[San03] Sandeep.S. *GCC-Inline-Assembly-HOWTO*, March 2003.

[Sik] Thomas Sikora. *MPEG-1 and MPEG-2 Digital Video Coding Standards*.

[Soh02] Oliver Sohm. *Variable-Length Decoding on the TMS320C6000 DSP Platform*, June 2002.

[tvf] *NTSC and PAL Formats*.

[Uss02] Rudolf Usselmann. *WISHBONE DMA/Bridge IP Core*. OPENCORES.ORG, January 2002.

[VZL] Matjaz Verderber, Andrej Zemva, and Damjan Lampret. *HW/SW PARTITIONED OPTIMIZATION AND VLSI-FPGA IMPLEMENTATION OF THE MPEG-2 VIDEO DECODER*.

[Wat94] Andrew B. Watson. *Image Compression Using the Discrete Consine Transform*, 1994.