Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Wintersemester 2003/04

DIPLOMA THESIS

for

Silvan Wegmann (D-INFK)

Main Reader: Herbert Walder

Issue Date:	27. october 2003
Submission Date:	26. february 2004

Video tasks for RHWOS

Contents

List of Figures

Chapter 1

Introduction

In the last few years FPGAs have become increasingly more powerful. Todays FPGAs provide a few hundred I/O pins and several thousand configurable logic blocks that correspond to millions of gate equivalents. With such components it is possible to implement complex applications within only one device. Xilinx fabricates a divers set of such FPGAs and with it also delivers powerful development tools. Complex applications that consist of control flow intensive parts often require some sort of general purpose processor. For this reason Xilinx also provides a fully featured RISC processor called MicroBlaze.

Instead of having only one complex application it is also possible to let several smaller applications coexist. They could be loaded and unloaded during runtime on demand. Xilinx devices advantage this procedure as they are partially reconfigurable which means that you can also change only part the FPGA and keep the rest in the previous state.

1.1 The XFBoard plattform

The XFBoard was developed as a prototyping plattform to study the principles of reconfigurable computing and reconfigurable operating systems. The board embodies two Xilinx FPGAs that are called C-FPGA and R-FPGA. The C-FPGA is used as a CPU and will not be reconfigured once the system is running. It is used as a host for the software parts of the operating system and depending on the type of application also for application software tasks. The R-FPGA is managed by the C-FPGA and is treated as an allocatable resource. Certain parts of the chip space on the R-FPGA however can not be used for application tasks as they are occupied by OS elements like ethernet driver, video driver, audio driver or FIFOs. On the board you will also find several IO components, like for example ethernet links, different types of memory, VGA connectors, an audio codec and also RS232 connectors. The FPGAs have several connecting buses that allow the OS to communicate with the R-FPGA, provide different types of clock signals and to access the configuration interface of R-FPGA. For further details consult [7] for the structure of the board and [4] and [10] for the principles of the operating system. Figures 1.1 and 1.2 give an impression of the infrastructure that is available on the XFBoard.

1.2 Motivation

While the OS plattform is being developed it is necessary to have a mean to measure its performance. In a past project [10] only small tasks could be tested while the OS was developped. For this reason the objective of my project was to provide a complex application consisting of several cooperating tasks that would challenge the limits of the RHWOS.

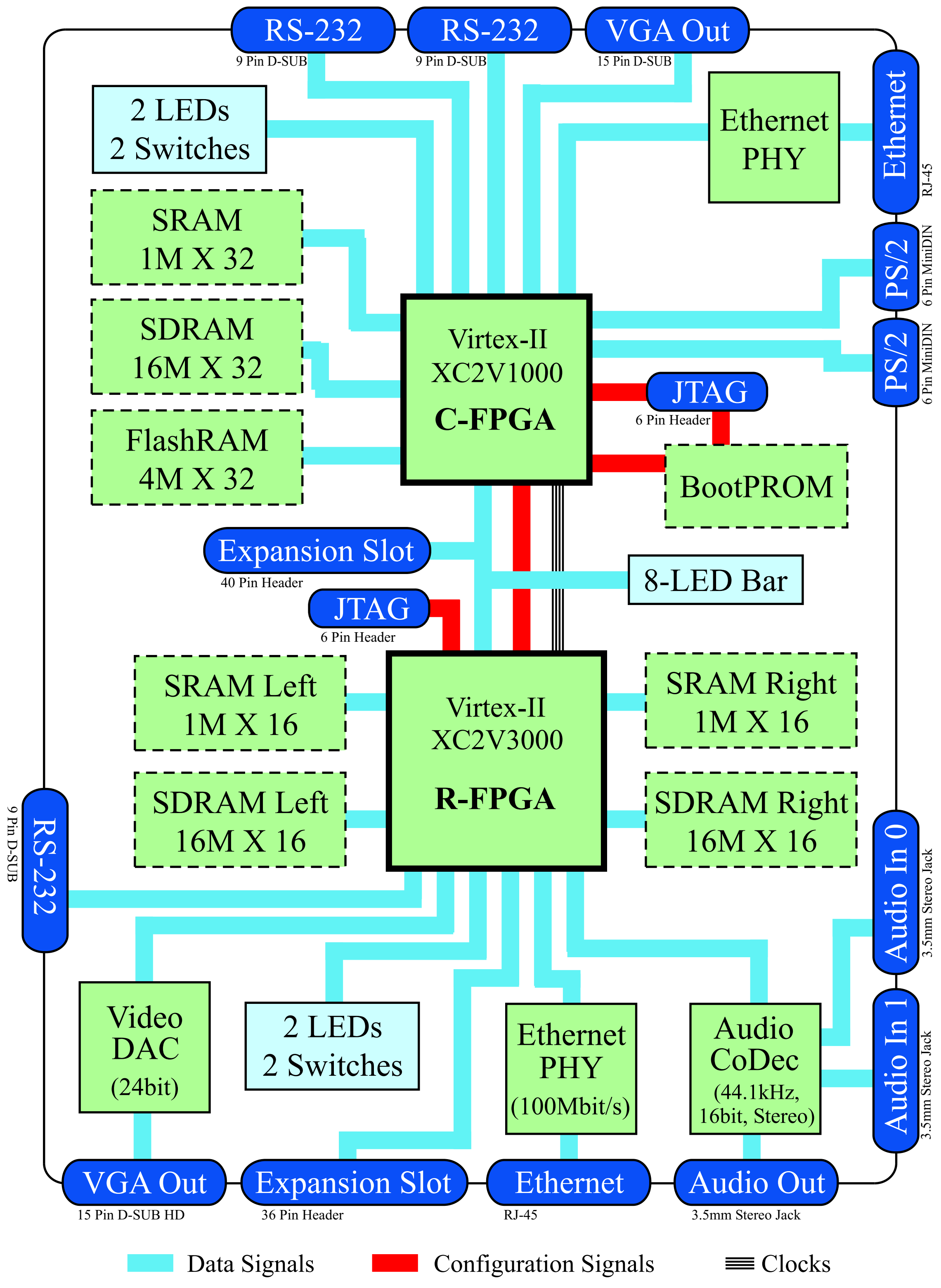


Figure 1.1: block diagram of the XF-Board

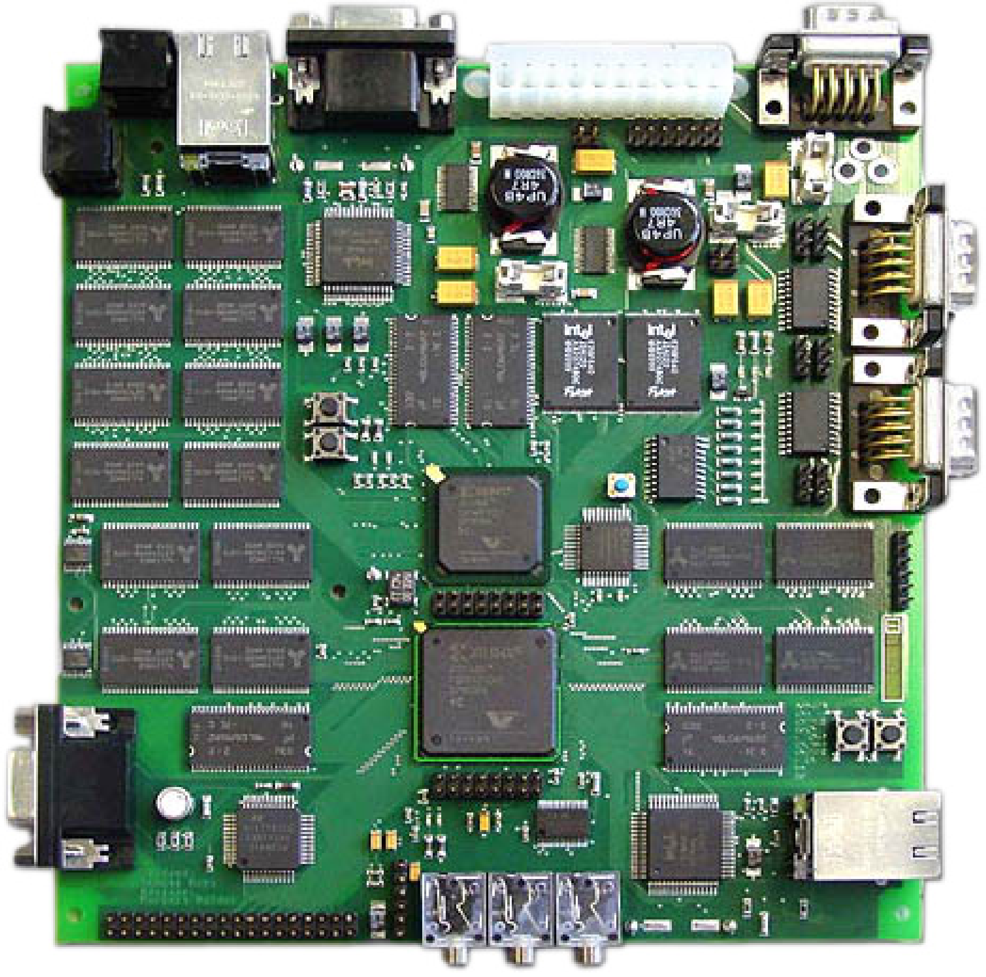


Figure 1.2: image of the XF-Board

Chapter 2

Project management

The video decoder application is only one project of a number of other projects that center around the XF board. While I worked on the evaluation and implementation of the video decoder other teams developed the underlying operating system infrastructure and hardware drivers for the external components. With this division I acted as a user of the future system and could give valuable feedback for several parts like the communication between tasks and the OS, resource management or task life cycles. There were also a number of thesis' that had a different main objective but provided basic OS components like a video driver, audio driver or ethernet transceiver. Figure 2.1 gives an overview of all works involved.

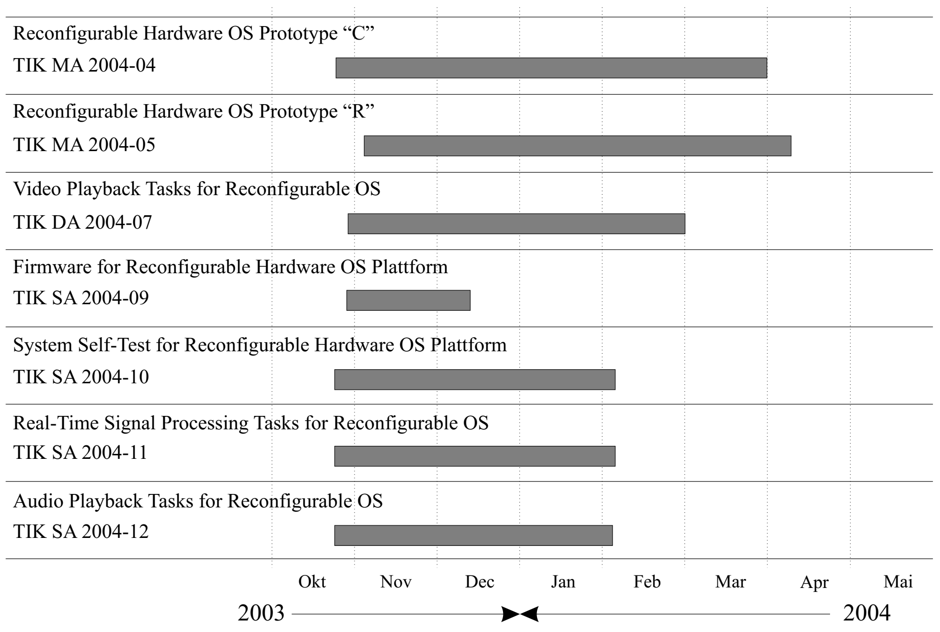


Figure 2.1: Gantt diagram of the XFBoard projects

Chapter 3

Short introduction to MPEG-2

The MPEG-2 standard as described in [5] defines the format of the data that represents a video stream. This includes the order of the different frame types, the structural elements of a frame and the sort of values that are encoding. This chapter will give a short introduction to the basic structural elements of an MPEG stream. For more details about Huffman coding tables, coefficient tables or even encoding issues the reader is refered to corresponding literature.

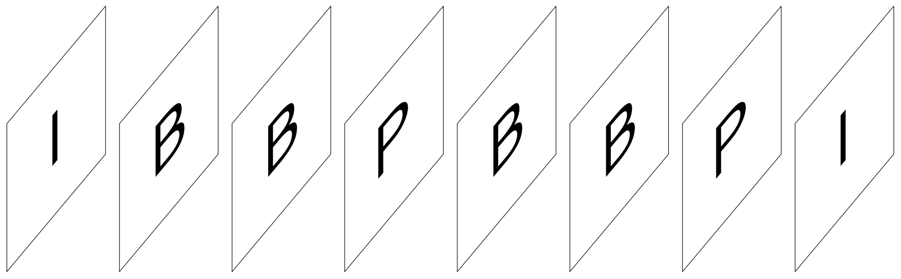


Figure 3.1: typical MPEG frame sequence

Generally an MPEG stream consists of a sequence of I-, B- and P-frames as shown in figure 3.1 I frames contain complete frames encoded in a JPEG like fashion. B- and P-frames contain only differential information that result in a complete frame when combined with other frames. The details about B- and P-frames will not be discussed in this document as they are not covered by the restricted implementation for XFBoard.

A frame consists of certain number of slices as shown in figure 3.2. The number slices can be chosen at encoding time and represents one of the design parameters for encoders. Each slice then consist of a certain number of macroblocks that again can be chosen at encoding time. All informations in MPEG files are variable length encoded and require several decoding tables to get the original information. A macroblock contains the encoded pixel information for a block of 16*16 pixels in YUV format. In the given figure the YUV data is

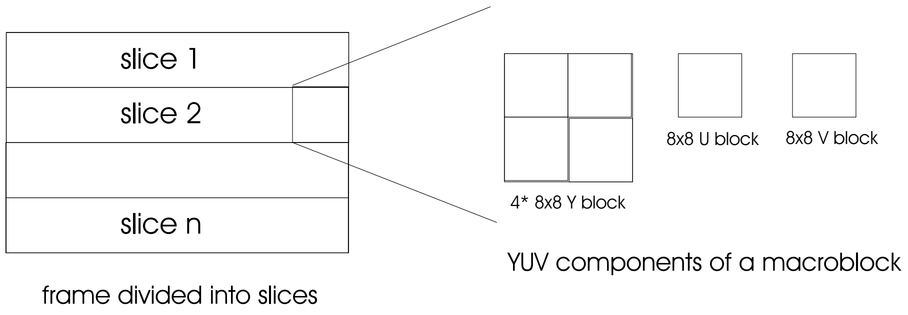


Figure 3.2: structure of a frame

provided in chroma 4:2:0 format which means that the full Y information is provided but only a forth of the U and V components is available. The remaining U and V components are copied four times to get the full information.

The Y, U and V blocks consisting of 8*8 pixel are the basis for the main decoding process. Each block is IDCT converted, combined to a chroma 4:4:4 block of 16*16 pixels and added to next frame that is being built up in the background buffer.

The following sequence of steps is therefore performed for each frame. For more details about the operations consult chapter 6 where you find a call graph for the decoding process or read respective literature [3], [11], [9] and [6].

- variable length decoding
- inverse discrete cosine transformation
- 4:2:0 to 4:4:4 chroma conversion
- YUV to RGB conversion

Chapter 4

Design decisions

The conceptual formulation asked for an implementation of a video decoder. It therefore kept the type of videos as one of the design parameters. Today we have a wide variety of formats that range from MPEG-1, MPEG-2 and Quicktime to DivX, MPEG-4 and Windows Media. Table gives an overview of the system requirements for some of the available players. Often these players support different formats which means, that the system requirements only give a quite rough idea of real performance requirements.

player	video format	requirements
Elecard MPEG2 Player	MPEG-2 High Level, MPEG-4 simple profile	MMX enhance CPU min. Pentium, K6 or Athlon, 32 MB RAM
DivX Video	DivX, MPEG-4 simple profile	Pentium II 450 MHz, 64 MB RAM with 640×480 and lowest settings for playback quality and post-processing
QuickTime 6.5	Quicktime Movie	400 MHz PowerPC G3, 128 MB RAM; No specific data available for PC

Table 4.1: Players and their respective performance requirements

For a complete evaluation, the performance requirements of PC applications do not suffice, other figures have to be considered as well. These parameters include bit rate, screen resolutions and compression ratios, but also availability of documentation and source code.

Among all the video formats MPEG-2 is the most widely used. MPEG-4 and DivX spread increasingly because of theire support for low bit-rate applications with still high image quality and they will eventually replace MPEG-2 some day. I still decided to implement an MPEG-2 decoder because this video format is well known and studied and one can get plenty of tools and source code for it.

Simple	4:2:0 sampling, only I- and P- pictures allowed
Main	all core MPEG-2 capabilites including B- pictures and support for interlaced video
4:2:2	same as main profile but 4:2:2 subsampling is used
SNR	as main profile but an enhancement layer is added to provide higher visual quality
Spatial	as SNR but spatial scalability may be used to provide higher quality enhancement layers
High	as Spatial but with support of 4:2:2 sampling

Table 4.2: Profiles defined in the MPEG standard

Low	Up to 352 x 288 frame resolution and up to 30 fps
Main	Up to 720 x 576 frame resolution and up to 30 fps
High-1440	Up to 1440 x 1152 frame resolution and up to 60 fps
High	Up to 1920 x 1152 frame resolution and up to 60 fps

Table 4.3: Levels defined in the MPEG standard

4.1 MPEG-2 variation

The MPEG-2 standard allows a wide range of values for typical video parameters like frame rate, bit rate, screen resolution and color encoding. Certain parameter combinations are defined in so called profiles and levels The MPEG-2 standard also defines a set of recommended combinations of profiles and levels. See tables 4.2, 4.3 and 4.4 for details.

The source code for a fully featured MPEG-2 decoder is freely available on the webpage of MPEG ([1]) and is also included on the CD. It served as basis for all further steps. It was clear that the XFBoard could not provide enough resources to support all of the features explained in the MPEG standard documents. Therefore I had to define a set of restrictions and priorities such that the resulting application would run at a reasonable speed and would be complete within predetermined time but would still follow the standards close enough to be called MPEG decoder. I decided constrain movies to the following specifications.

- I frames only (this also excludes features like spatial scalability or SNR)
- Limited to 25 frames per second
- No interleaced frames allowed (interleaced movies are mostly used for TV recorded streams)
- Frame size limited to 160×120 pixel (this simplifies the hardware complexity of the video core and limites the memory needs for frame buffers)
- Chroma format restricted to 4:2:0 in the hardware accelerated version

	Low	Main	High-1440	High
Simple		720*576 15 Mb/s no B frames		
Main	4:2:0 352*288 4 Mb/s	Broadcast digital TV 15 Mb/s	1440*1152 60 Mb/s	1920*1152 90 Mb/s HDTV
4:2:2		720*576 50 Mb/s		
SNR	352*288 4 Mb/s	720*576 15 Mb/s		
Spatial			1440*1142 60 Mb/s	
High		720*576 20 Mb/s up to 4:2:2	1440*1152 80 Mb/s up to 4:2:2	1920*1152 100 Mb/s up to 4:2:2

Table 4.4: Profile and level combinations defined in the MPEG standard together with typical values for the design parameters

- Decoding of audio streams was given low priority (this feature is not supported by the orignial source code and would have needed additional studies)

4.2 Memory considerations

The XFBoard can only provide very limited resources especially in terms of memory. The board provides SRAM, SDRAM and block RAM (BRAM) as given in table 4.5.

type	total amount	comments
SRAM	4 MB	2 independant banks of 1 M × 16 bit; Alliance AS7C34096 chips
SDRAM	64 MB	2 indepedant banks of 16 M × 16 bit; Infineon HYB39S256160CT chips
BRAM	210 kB	96 Blocks RAM of 18 kbit each, dual ported with configurable dimensions; integrated on the XC2V3000 FPGA

Table 4.5: Memory available on the R-FPGA of the XFBoard

The video decoder is a quite complex application and pushes the limites of the XFBoard. Especially the usage of memory represents a major issue. Code memory needs were not the biggest problem as the amount of around 45'000 bytes of code space could even be satisfied by the block RAMs. The bigger problem was the huge amount of memory used for the two video frames (one frame to display and one to construct the next image), and the buffers

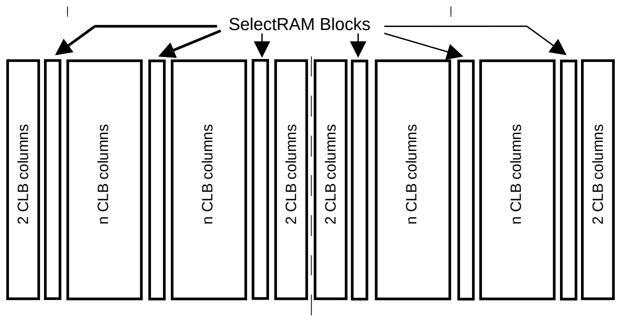


Figure 4.1: distribution of the block RAM resources over the FPGA

for the chroma conversions. These buffers alone used more than 100'000 bytes and would never fit into the block RAMs. Another major problem was the execution speed which led to the conclusion that only block RAMs were feasible for code storage. To satisfy the needs of image buffer memory the SRAM blocks were used. For this purpose a OPB core SRAM controller was developed and integrated into the decoder application. For the future integration of decoder into the OS environment it has to be considered that the amount of block RAM memory that can be used is not equal to the total amount of available block RAM. This is because the memory blocks are arranged in 6 columns of 16 blocks with 18kbit per block (see figure 4.1 for chip geometry).

Chapter 5

Steps to the final application

Because the underlying operating system was not available at the beginning of my work I could not start with the OS based implementation of a video decoder application straight away. Instead I concentrated my efforts on a standalone application that allowed me to familiarise myself with the FPGA environment and the MPEG-2 code. Based on this implementation I could later make performance measurements. From the results I could then derive the calculation intensive parts and implement them as hardware components. At the same time I was involved in the design meetings for the operating system development. There I could explain my needs and also give proposals for design decisions that would influence the performance of the decoder.

The project was split up into the following consecutive steps.

Drop-in solution	In this standalone version, the core consists of a Microblaze softprocessor that is surrounded by an ethernet lite OPB core, an UART lite OPB core and several OPB GPIO cores that connect to the video and audio drivers.
Profiling	Based on the drop-in solution, performance measurements are made. For this purpose another OPB core was inserted that allows the counting of execution cycles.
Hardware acceleration task decomposition	Speed critical parts are translated into hardware tasks and the Microblaze part is encapsulated into its own hardware task.
MPEG-2 application	All hardware tasks are available as partial bitstreams and can be downloaded one by one. They communicate with each other via OS elements.

5.1 Drop-in solution

As mentioned earlier in this chapter I could not start with the final OS based application right away. In the first version I had to provide most the infrastructure myself. The video and audio drivers were built by other teams and the ethernet receiver as well as the UART could be taken from the Xilinx EDK environment. Because the basis of the decoder was the source code of the motion picture expert group (MPEG) from theire website, the idea was to start with a pure software implementation. To start with we built a complete system as given in figure 5.1 that could run on the R-FPGA of its own.

The core of the software implementation was Microblaze system

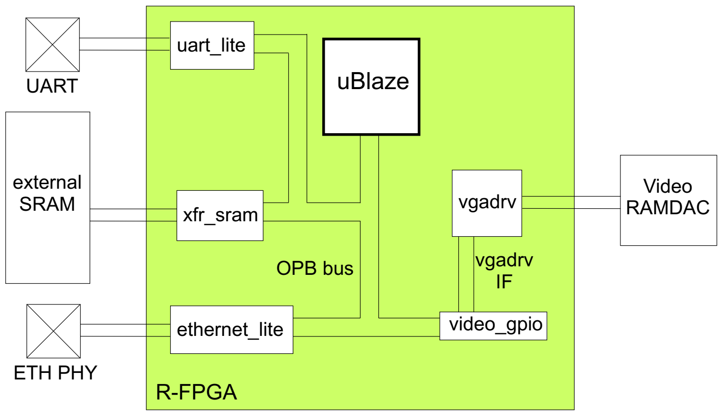


Figure 5.1: drop-in solution

5.2 Profiling

The drop-in solution showed the feasibility of a MPEG-2 decoder on the XF-Board but the performance could not reach the necessary speed of 25 fps to show videos in the original frame rate. 13'873'362 cycles for one frame at a system speed of 50MHz resulted in a frame rate of 4.4fps which is far frome the target speed. Such a result of course was expected as the low system frequency together with a pure software implementation could not reach the performance of even the smallest desktop PC. With a simple stop watch OPB core[1] the most time consuming operations were quickly identified. Table 5.1 shows all these operations and an average cycle count.

The operations in table 5.1 that are marked in green are combined under the term backend throughout the rest of this document.

[1]this core is also included on the CD. See appendix A for details about the CD

operation	reference	cycles	percent
network handling	frame	3'909'615	28
IDCT	block	5'212	17.5
Chroma 4:2:0 to 4:4:4	frame	682'888	5
YUV to RGB	pixel	70	9.5
store pixel	pixel	116	15
VLD and other	frame	3'207'899	25
a complete frame	frame	13'873'362	100

Table 5.1: Average number of cycles for a complete 160 × 120 pixel frame

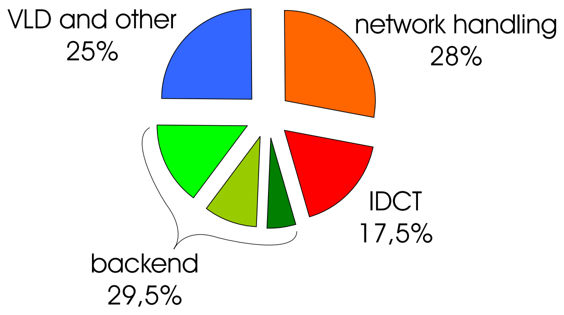


Figure 5.2: cycles in percent

5.3 Hardware acceleration and task decomposition

The next step was to accelerate the critical parts and also to prepare the system for the OS environment. This meant that the hardware components should become self-contained entities that could easily be transformed into tasks. The structure becomes more decentralized and all components can later be matched to OS elements or application tasks. For this project I decided to first concentrate on the backend of the decoder and still keep the rest in software. In the Embedded Development Kit the design flow was switched from Microblaze centered view to a view where the processor becomes only a submodule of a pure VHDL project. Two major tasks where identified and integrated as hardware components. One of them was a task that simply stores data from the FIFO into the YUV RAM. The other one reads the YUV RAM, performs chroma 4:2:0 to 4:4:4 and YUV to RGB conversion and stores the pixels in the VGA memory. Performance measurements of the hardware accelerated backend showed promising results. The number of cycles used to convert and store a complete frame dropped from around 4,8 million cycles to 192'000 cycles which is as a reduction of 95%. However the overall performance of the system remained poor and could only be increased by 0.5fps from 4.5fps to 5fps.

operation	reference	cycles	percent
network handling	frame	3'909'615	40
IDCT	block	5'212	20
backend	frame	192'000	2
VLD and other	frame	3'207'899	38
a complete frame	frame	9'853'282	100

Table 5.2: Average number of cycles for a complete 160 × 120 pixel frame with hardware acceleration of the backend

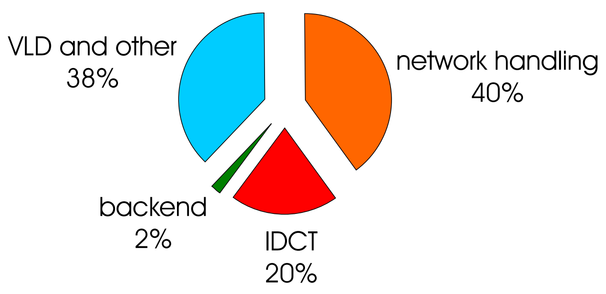


Figure 5.3: cycles in percent of the accelerated version

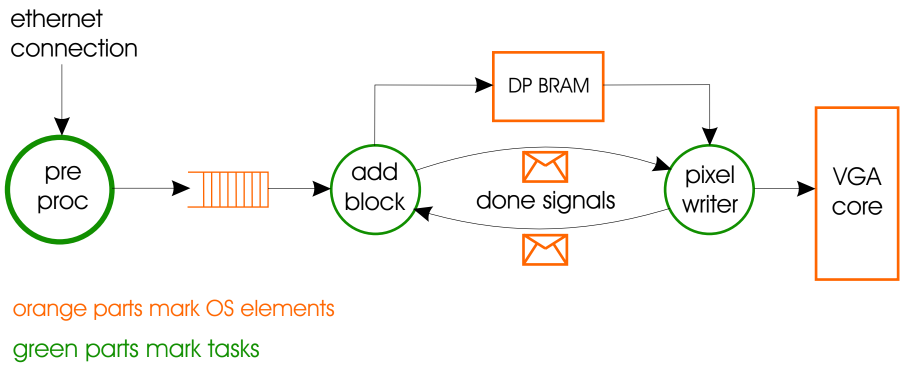


Figure 5.4: logic structure of the backend

5.4 MPEG-2 application

The final version of the decoder will consist of a few hardware and software tasks that are prepared to be load as soon as needed. The OS loads the software parts and reconfigures the R-FPGA with the hardware tasks. All services that I previously had to implement myself, like FIFOs, network traffic, share block RAM and the like, are now provided by the OS. When configured the partitioning of the FPGA might look like in figure 5.5. Until the end of my project however there was no possibility to evaluate the performance of the MPEG decoder under this conditions because the OS infrastructure for R-FPGA was still under development.

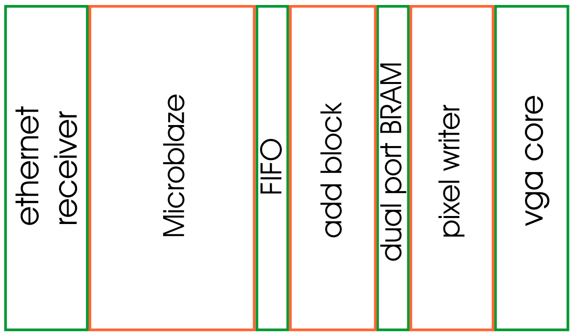


Figure 5.5: hardware tasks and OS elements arranged in the final application

Chapter 6

Detailed View

6.1 Overview of the MPEG-2 decoder source

In figure 6.1 you can see the callgraph of the `Decode_Picture` function from the original MPEG-2 source. This function is responsible for decoding a single frame and is therefore the starting point for performance measurements. The figure also shows which functions of the call graph have been removed in the restricted implementation that was used in this project. Compared to the original source (provided on the CD in directory ....) also other functions, mainly those involved in decoding B and P frames were removed.

module	omitted functions
getblk.c	Decode_MPEG1_Non_Intra_Block Decode_MPEG2_Non_Intra_Block
stascal.c	complete module omitted
motion.c	complete module omitted
getvlc.c	Get_dmvector Get_motion_code
gethdr.c	picture_spatical_scalable_extension picture_temporal_scalable_extension

Table 6.1: omitted functions in the MPEG-2 decoder

6.2 Streaming Protocol

The MPEG standard defines a format for packets sent via a network. This format requires the sending of redundant information to synchronize a streaming client with the server. It also means additional computation time will be necessary on the client side to unpack the video stream data. Furthermore this feature was not supported by the decoder source code and would have meant additional implementation effort. Because of this I decided to create a much simpler, UDP based streaming protocol. For a better control of the progress of a streamed video I let the client decide when he needs new data. He simply

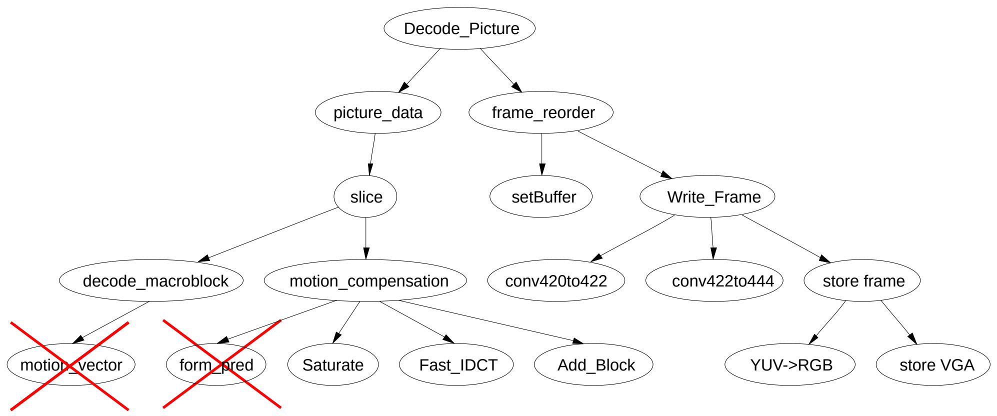


Figure 6.1: call graph for decoding one frame

sends a UDP packet containing the text 'NEXT' to a streaming server which in return sends a UDP packet containing payload data to the client. The size of the payload packets is defined on the server side and can vary among different application.

6.3 Implementation of the backend

The Microblaze still runs those parts in software that are not yet hardware accelerated. In figure 5.4 the Microblaze is contained in the task labeled 'pre proc' for pre processor. The connection to the output FIFO is implemented as a simple OPB core. The backend consists of two tasks called 'add block' and 'pixel writer'. The 'add block' task simply fills the YUV RAM with data that he reads from the FIFO. As soon as a complete macroblock is written to the RAM it stops reading the FIFO and signals the pixel writer to start its operation. The 'pixel writer' task reads the 4:2:0 YUV data from the block RAM, performs to 4:4:4 conversion on the fly, converts the values to RGB and finally writes the pixels to the VGA memory. As soon as the pixel writer has completed his work he signals the add block task, that the memory is now free again to be filled with new data. For an integration of the Microblaze task into the OS environment it has to noted that at least two columns of block RAMs are necessary. This makes the pre processor task very limited in the relocation possibilities. No detailed analysis has been made on this so far, but it seems like only one or two fixed locations on the FPGA can suffice this needs.

6.3.1 add block task

The add block task is a simple state machine that stores data from the FIFO into the dual ported block RAM that is used as memory for a complete YUV 4:2:0 encoded macroblock. The interface of the task has the following structure. This task has no special requirements to a fixed location within the FPGA. The only important things to notice are that he needs a connection to the FIFO that comes from the Microblaze and another connection to the dual ported YUV RAM.

```
entity store is
  port (
    -- Clock and Reset
    ClkxCI        : in  std_logic;
    RstxRI        : in  std_logic;
    -- signal to reset counters and start with a new frame
    RestartxSI    : in  std_logic;
    -- interface to the FIFO that is connected to the Microblaze
    -- on the other side
    FIFOURExEO    : out std_logic;
    FIFOEmptyxSI  : in  std_logic;
    FIFODataxDI   : in  std_logic_vector(7 downto 0);
    -- interface to the dual ported block RAM
    YUVWExEO      : out std_logic;
    YUVDataxDO    : out std_logic_vector(7 downto 0);
    YUVAddrxDO    : out std_logic_vector(8 downto 0);
    -- synchornization signals for the pixel writer
    StoreRdyxDO   : out std_logic;
    PixwrtRdyxDI  : in  std_logic
    );
end store;
```

6.3.2 pixel writer task

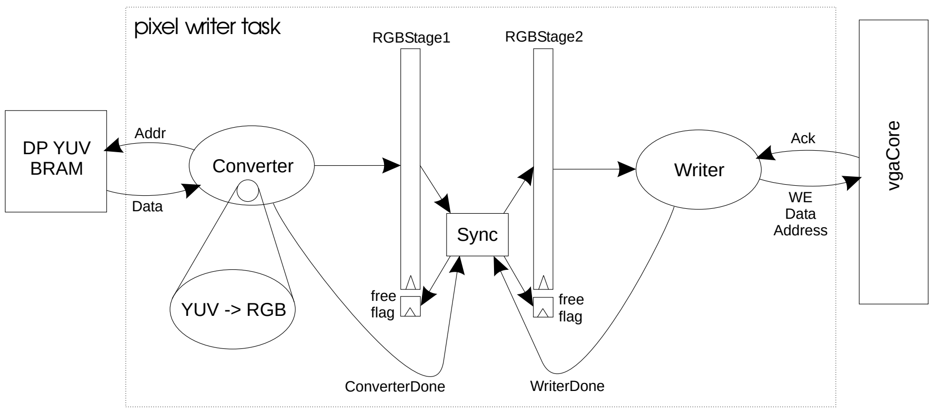


Figure 6.2: schematic of the pixel writer hardware task

The pixel writer task has no special requirements for the OS. It simply needs access to the dual ported YUV RAM on one side and a connection to one of the frames of VGA core. The interface of the pixel writer has the following structure.

```
entity pixwrt is
```

```
  generic (
    IMAGE_WIDTH  : integer                       := 160;
    IMAGE_HEIGHT : integer                       := 120;
    BASEADDR     : std_logic_vector(19 downto 0) := (others => '0')
    );
  port (
    -- Clock and Reset
    ClockxCI      : in  std_logic;
    ResetxRI      : in  std_logic;
    -- signal to reset counters and start with a new frame
    MBSwapxSI     : in  std_logic;
    -- messaging signals to synchronize the pixel writer
    -- and the add block task
    PixwrtRdyxDO  : out std_logic;
    StoreRdyxDI   : in  std_logic;
    -- interface to the block RAMs
    YUVDataxDI    : in  std_logic_vector(7 downto 0);
    YUVAddrxDO    : out std_logic_vector(8 downto 0);
    -- interface to the VGA core
    WAddrxDO      : out std_logic_vector(19 downto 0);
    WDataxDO      : out std_logic_vector(15 downto 0);
    WWExEO        : out std_logic;
    WAckxSI       : in  std_logic;
    WBIDxDO       : out std_logic
    );
end pixwrt;
```

6.3.3 YUV to RGB conversion

In figure 6.3 you can see the data flow graph of the YUV to RGB conversion as implemented in the software version of the MPEG decoder. To implement the backend, this code was taken as a basis and then pipelined as shown with the numbered registers. The numbers correlate with the numbers used in the VHDL source code.

6.3.4 IDCT

Altough no IDCT component could be included in the final version, the development was under way. The data flow graph in figure 6.4 is taken from the fast IDCT conversion function in the C sources. Based on this graph a first VHDL implementation was made and tested. The simulations proved the correctness of the core but the synthesis some timing problems. The numbers beside the registers correlate with the numbers used the VHDL source which allows easier identification. The current implementation is based on 32 bit operands for all as the original C implementation also used 4 byte wide integers. However this turned out to be a bad approach as not all operands use the complete 32 bit range. Fixing this problem by using appropriate bus widths can surely produce nicer timing because simpler routing.

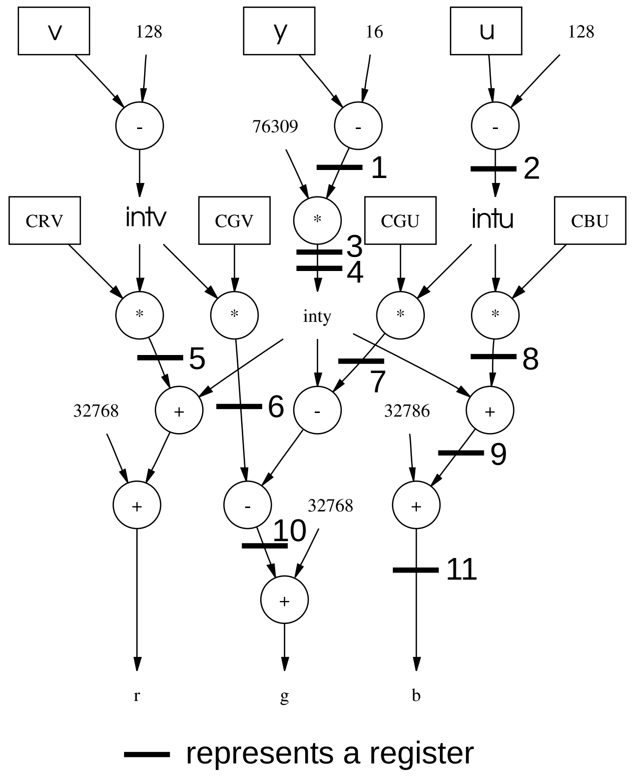
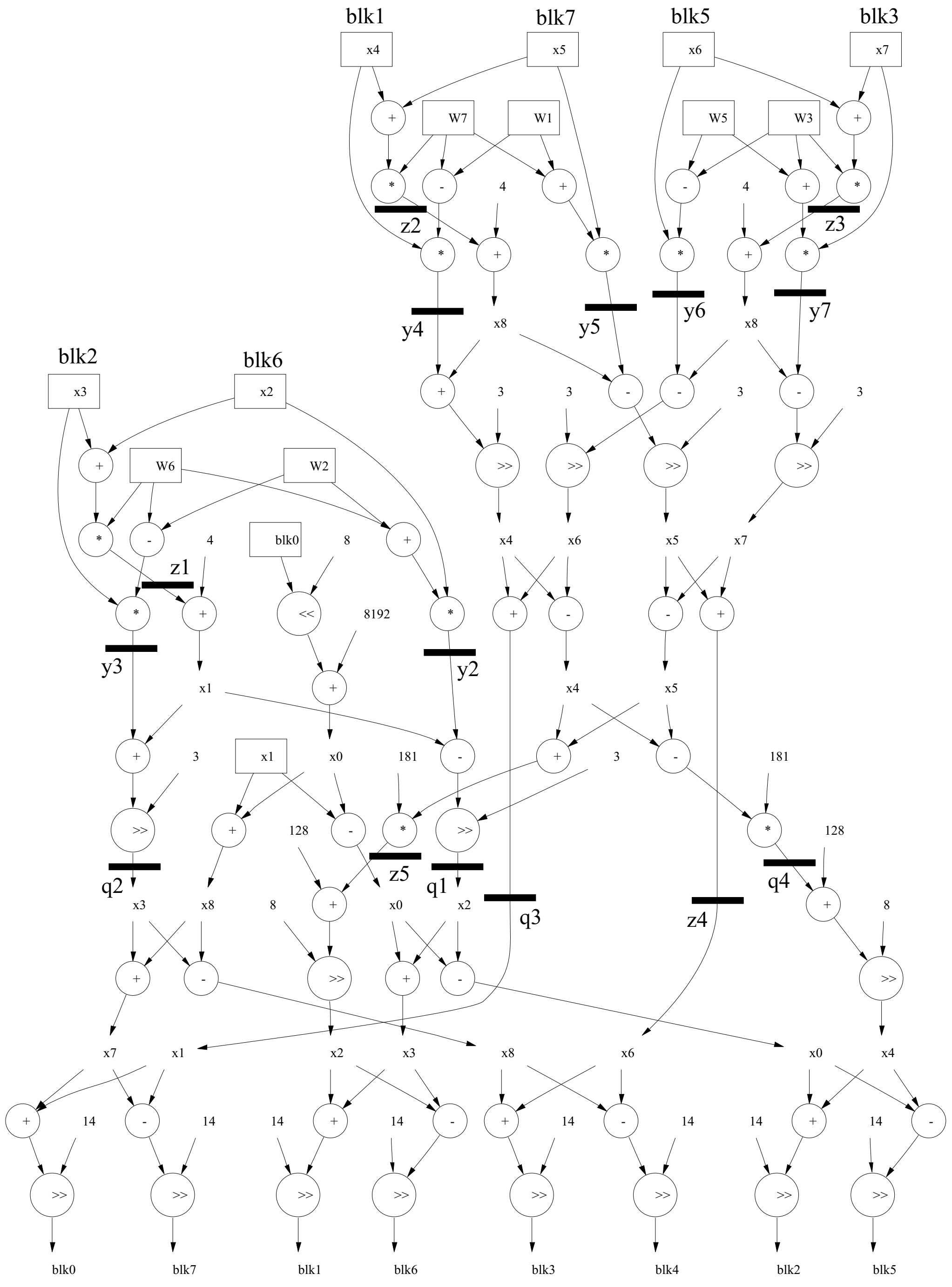


Figure 6.3: data dependency graph of the YUV to RGB conversion



▬ represents a register

Figure 6.4: data dependency graph of the IDCT conversion

Chapter 7

Conclusions

The acceleration of the backend part of the MPEG-2 decoder only resulted in a fairly small speed up for the whole processing time for one frame. But when comparing the execution time of the backend before and after the hardware acceleration it becomes clear, that hardware implementations of more software components can increase the performance. When additionally replacing IDCT and the networking component by hardware cores we can achieve a speed of around 3'750'000 cycles per frame which means 0.075s per frame or 13 fps. This circumstance is visualized in figure 7.1.

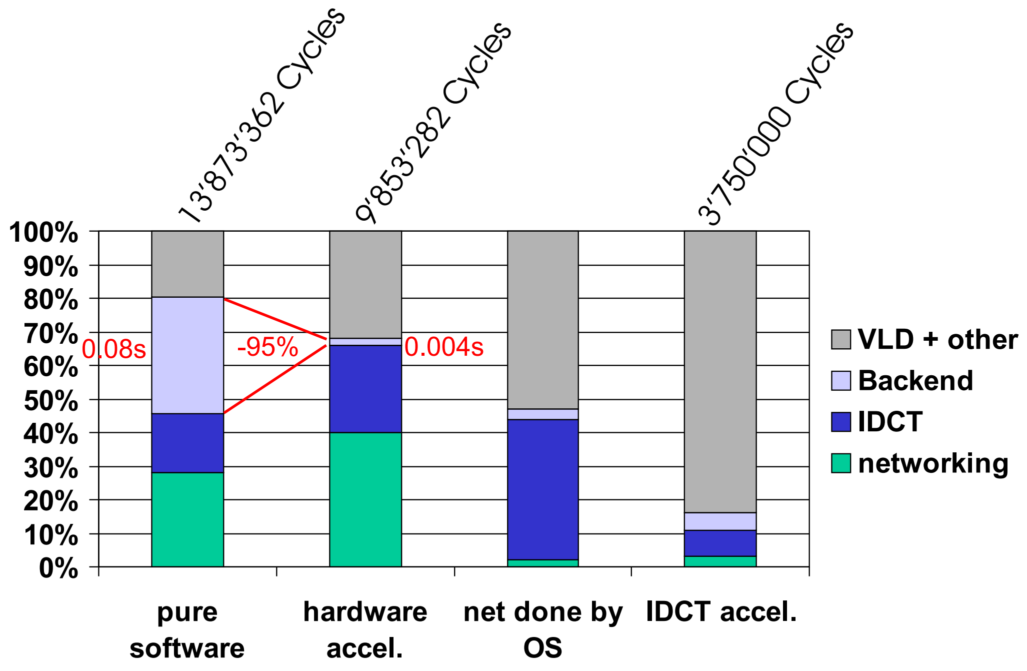


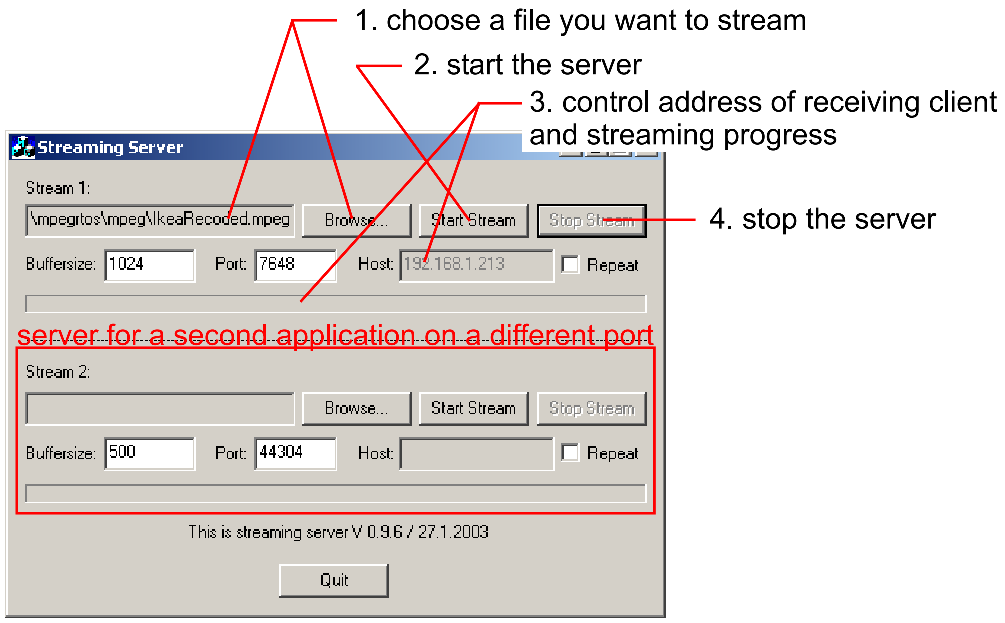
Figure 7.1: cycles ratios of different acceleration variations

Chapter 8

Documentation of the software tools

8.1 Streaming server

This Windows application generates the special data stream that is needed for the video and audio players in all of the XForces projects. Instead of a standard MPEG stream as defined in [5] I decided to use a much simpler UDP protocol. This saved me from implementing a complete TCP/IP stack and also provide the memory or gate space for it. The streaming server simply waits for requests from a vido and/or audio source and sends packets of a given size to the requesting client. 8.1 shows the steps for the standard scenario.


1. choose a file you want to stream
2. start the server
3. control address of receiving client and streaming progress
4. stop the server
server for a second application on a different port
Streaming Server
Browse...
Start Stream
Stop Stream
Repeat
Quit

Stream 1:
\mpegrtos\mpeg\IkeaRecoded.mpeg
Buffersize: 1024 Port: 7648 Host: 192.168.1.213

Stream 2:
Buffersize: 500 Port: 44304 Host:

This is streaming server V 0.9.6 / 27.1.2003

Figure 8.1: screen shot of the streaming application

8.2 XFBLoader

The XFBLoader can be used to configure the R-FPGA of the XFBoard. The OS on the C-FPGA provides a service to receive configuration bitstreams via the ethernet connection. On the PC side you can select a directory containing *.bit files, set a destination IP and start transmitting the bitstreams. As soon as the transfere is completed you can configure the R-FPGA by typing a few shell commands on the C-FPGA OS shell. For this purpose the `bitlist` and `config` commands are used. Listing ...... gives an example of a typical session.

8.3 MPEG encoder

The source of MPEG decoder that served as basis for the initial software implementation, was originally a standard C applications that could decode a MPEG-2 movie file and generate a sequence of images that represent the respective frames. Together with that a simple encoder was delivered that could encode a sequence of images into a MPEG-2 stream. For my purposes I had to reencode a given MPEG file in such a way that it would only contain I frames. The parameters used for the encoding with mpeg2enc are given in listing 8.1.

Listing 8.1: options file for the sample video

```
MPEG-2 Test Sequence, 25 frames/sec
IkeaDec%d    /* name of source files */
-          /* name of reconstructed images ("-": don't store) */
-          /* name of intra quant matrix file     ("-": default matrix) */
-          /* name of non intra quant matrix file ("-": default matrix) */
stat.out   /* name of statistics file ("-": stdout ) */
2          /* input picture file format: 2=*.ppm */
794        /* number of frames */
0          /* number of first frame */
00:00:00:00 /* timecode of first frame */
1          /* N (# of frames in GOP) */
1          /* M (I/P frame distance) */
0          /* ISO/IEC 11172-2 stream */
0          /* 0:frame pictures, 1:field pictures */
160        /* horizontal_size */
120        /* vertical_size */
2          /* aspect_ratio_information 2=4:3 */
3          /* frame_rate_code 3=25 */
50000000.0 /* bit_rate (bits/s) */
112        /* vbv_buffer_size (in multiples of 16 kbit) */
0          /* low_delay  */
0          /* constrained_parameters_flag */
4          /* Profile ID: Main = 4 */
6          /* Level ID:   High 1440 = 6 */
1          /* progressive_sequence */
1          /* chroma_format: 1=4:2:0, 2=4:2:2, 3=4:4:4 */
1          /* video_format: 1=PAL */
5          /* color_primaries */
```

```
5           /* transfer_characteristics */
5           /* matrix_coefficients */
160         /* display_horizontal_size */
120         /* display_vertical_size */
0           /* intra_dc_precision (0: 8 bit */
1           /* top_field_first */
0 0 0       /* frame_pred_frame_dct (I P B) */
0 0 0       /* concealment_motion_vectors (I P B) */
1 0 0       /* q_scale_type  (I P B) */
1 0 0       /* intra_vlc_format (I P B)*/
0 0 0       /* alternate_scan (I P B) */
0           /* repeat_first_field */
0           /* progressive_frame */
0           /* P distance between complete intra slice refresh */
0           /* rate control: r (reaction parameter) */
0           /* rate control: avg_act (initial average activity) */
0           /* rate control: Xi (initial I frame global complexity measure) */
0           /* rate control: Xp (initial P frame global complexity measure) */
0           /* rate control: Xb (initial B frame global complexity measure) */
0           /* rate control: d0i (initial I frame virtual buffer fullness) */
0           /* rate control: d0p (initial P frame virtual buffer fullness) */
0           /* rate control: d0b (initial B frame virtual buffer fullness) */
2 2 11 11 /* P:  forw_hor_f_code forw_vert_f_code search_width/height */
1 1 3  3  /* B1: forw_hor_f_code forw_vert_f_code search_width/height */
1 1 7  7  /* B1: back_hor_f_code back_vert_f_code search_width/height */
1 1 7  7  /* B2: forw_hor_f_code forw_vert_f_code search_width/height */
1 1 3  3  /* B2: back_hor_f_code back_vert_f_code search_width/height */
```

To create your own MPEG streams that can be viewed with the MPEG decoder you would use the following sequence. The imaginary filename ¡mymovie.mpeg¿ is used to denote the original MPEG stream and ¡mymovie_recoded.mpeg¿ to denoted the transformed MPEG stream. The mymovie.par file should contain exactly those parameters shown in listing 8.1. The number of frames that has to be adjusted in the .par file can be extracted from the highest frame number that is generated.

```
mpeg2dec.exe -b <mymovie.mpeg> -o3 <mymovie_recoded%d.mpeg>
mpeg2enc.exe mymovie.par <mymovie.mpeg>
```

Note: Do not forget the %d at the end of the output filename when deocding. This part is used to generate numbered file names, one for each frame.

8.4 Expression Parser

The expression parser was developed to visualize complex data flow graphs. It consist of a Pyhton script that can parse a very small subset of C expressions. The grammar is restricted to the operations +, -, *, /, ¡¡, ¿¿ and the grouping symbols ( ) (see source code for complete supported grammar). It takes a source file and generates a graph description in the dot format that is understood by the Graphviz package from AT&T. A typical session would look like in listing 8.2.

Listing 8.2: a typical session with expr_parse.py

```
python expr_parse.py inputfile.txt > ddg.dot
dot -Tps2 ddg.dot > ddg.ps
```

Appendix A

Contents of the CD

\root	
\src	Source code of all applications
\mpeg	Original source of the MPEG-2 decoder and encoder
\opb_xfstopwatch_v1_00_a	source of the OPB stopwatch core (documentation available)
\streamer	source code of the Visual C++ project for the streaming server
\apps	Binaries of all used applications
\streamer	Streaming server for audio and video applications
\XFBloader	
\exprparser	Python expression parser script
\mpegtools	MPEG decoder and encoder
doc	
\chapters	LaTeXsources of the chapters
\images	Corel Draw 10 files and thereof converted EPS files used in the documentation
\pres	Power Point slides of the introduction and the final presentation

Bibliography

[1] Mpeg website. http://www.mpeg.org.

[2] Tobias Gysi Andreas Ess. Signal processing tasks, 2004. TIK SA 2004.11.

[3] Xuemin Chen. *Transporting Compressed Digital Video.* Kluwer Academic Publishers, 2002.

[4] Herbert Walder et al. Reconfigurable hardware os prototype. TIK Report 168.

[5] Motion Pictures Expert Group. Iso/iec jtc1/sc29/wg11 coding of moving pictures, and associated audio, March 1994. Part 13818-2 - Video.

[6] Heiner Ksters. *Bilddatenkomprimierung mit JPEG und MPEG.* Franzis', 1995.

[7] Samuel Nobs. Prototype board for reconfigurable os, 2003. TIK SA 2003.22.

[8] Daniel Hobi Pascal Ldi. Audio playback tasks for rhwos. TIK SA 2004.12.

[9] Iain E. G. Richardson. *Video Codec Design.* John Wiley & Sons, Ltd, 2002.

[10] Michael Ruppen. Reconfigurable os prototype. Master's thesis, ETH Zurich, 2003. TIK DA 2003.11.

[11] John Watkinson. *MPEG-2.* Focal Press, 1999.

26th February 2004