ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Andreas Ess, Tobias Gysi

Signal Processing Tasks for RHWOS

Student Thesis SA-2004-11
Winter Term 2003/2004

Tutors:
Herbert Walder and
Matthias Frey

Supervisors:
Prof. Dr. H.-A. Loeliger and
Prof. Dr. L.Thiele

6.2.2004

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

In this semester thesis, we present both a VGA driver, as well as two signal processing tasks (turbo decoder, spectrum analyzer) for the XFBOARD [9]. As these two topics are mostly independent, the paper is split into two parts.

First, an overview of the VGA driver is given, including a sample application demonstrating how to use it for other tasks' purposes. This is mainly thought of as a reference for the user of the driver, but also contains enough information on the internals for further development.

Then, as sample tasks for the XFBOARD's Reconfigurable Hardware Operating System (RHWOS), two computationally intensive signal processing applications were chosen.
First, a spectrum analyzer task is discussed. We focus on the basic building blocks, and the few steps that yet have to be taken to actually put it to work.
As main part of this thesis, an implementation of a turbo decoder according to the UMTS specifications is presented. In the respective chapter, we deal with an overview over the algorithm, considerations when translating it to hardware, and qualitative comparisons between ASIC and FPGA implementations.

Acknowledgements

It was very instructional to be able to work on such an interdisciplinary semester thesis. This would not have been possible without the help of a lot of people, who we would like to acknowledge in this section.

We would like to thank our advisors Herbert Walder and Matthias Frey. Both have provided valuable information throughout the whole project. They invested much of their time, and many of their advices helped to make this project what it is.
We also very much appreciate the help of Prof. Loeliger, who came up with the suggestion of implementing a turbo decoder, and thus allowed us to actually tackle this interdisciplinary project. He took the time on several occasions to explain things to us and gave useful input to this project.
Thanks to Prof. Thiele for his confidence in this project and providing the infrastructure at TIK.
Thanks go to Christian Plessl as well, who has provided us with some useful information concerning differences between FPGAs and ASICs.
Last but not least, we would also like to thank the other students working in the same room. They have provided a pleasant and helpful atmosphere, and it was not only instructional, but also fun to work with them in a team.

Contents

Chapter 1

Introduction

1.1 Reconfigurable Hardware Operating System

Reconfigurable systems in general have a great potential to not only speed up applications, but also save chip area. On top of that, the system can be made more flexible. To give a quick example, there exist many versions of turbo decoders. A native ASIC implementation needs support for each standard built in, whereas a reconfigurable version can easily specialize on one decoder type. A different version can then be easily obtained by reconfiguring the FPGA.

Thus, a good system would run on a reasonably small FPGA, and dynamically load tasks on demand from some sort of external memory. This asks for some sort of hardware operating system that manages the scheduling of resources.

1.2 XFBoard platform

This semester thesis was done in the scope of the X-Forces project, which is supposed to show the possibilities and limits of such a reconfigurable hardware operating system.

In [9], a demonstration board was developed specifically for this project. Figure 1-1 shows an abstract overview over the board. The core is formed by two FPGAs. On top, there's the C-FPGA (CPU), which handles resources and tasks for the R-FPGA (reconfigurable). A possible application now can run large parts of the code on the C-FPGA, and swap out data-flow intensive parts directly to hardware in order to dramatically speed up calculations.

This thesis concentrates on the R-FPGA part of the system. It was necessary to develop a means for the tasks to output their information on the screen, thus the VGA driver.
The RHWOS is currently being developed in [10] and [12]. As it is not completely ready yet, the spectrum analyzer and turbo decoder tasks are currently running entirely on the R-FPGA.

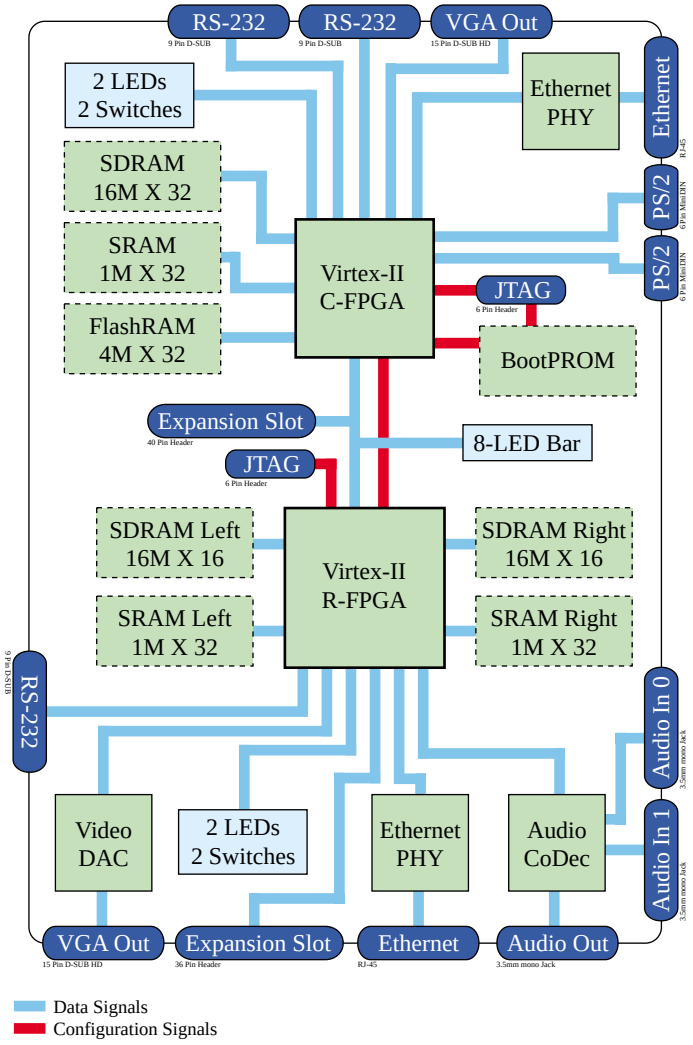

Figure 1-1: XFBoard - Scheme

Chapter 2

VGA Driver

2.1 Introduction

As part of the RHWOS project [14], the first task was to design an appropriate VGA driver for the R-FPGA part of the system.
A hardware based multi-window approach was chosen. The main reason for this was to take the burden of implementing a windowing system from the RHWOS. Furthermore, this facilitates the handling of windows for the tasks themselves. A task can access its part of the screen without worrying about the window position or other tasks.

The VGA driver was also used later in the course of our project to display the output of the turbo decoder sample application.

2.2 Features

- $640 \cdot 480$ px standard VGA resolution at a refresh rate of 60 Hz
  - adjustable by recompilation
- 16 bit color depth
- up to 4 independent windows
  - separate video RAM area for each task
  - freely moveable and sizeable
  - double-buffering capability
- memory independent test picture

2.3 Requirements

- Assumes 50 MHz clock for monitor timing
- 1 BlockRAM for temporary line buffer
- 405 out of $14'336$ slices (2%)

- 323 out of $28'672$ slice flip flops (1%)
- 710 out of $28'672$ input LUTs (2%)
- 1 DCM (optional, for different resolutions)

2.4 Overview

As seen in picture 2-1, the VGA core is partitioned into three parts:

- the VGA driver is the actual interface to the DAC
- the window manager translates screen coordinates into the respective window's memory addresses
- the memory manager handles concurrent accesses to the video memory

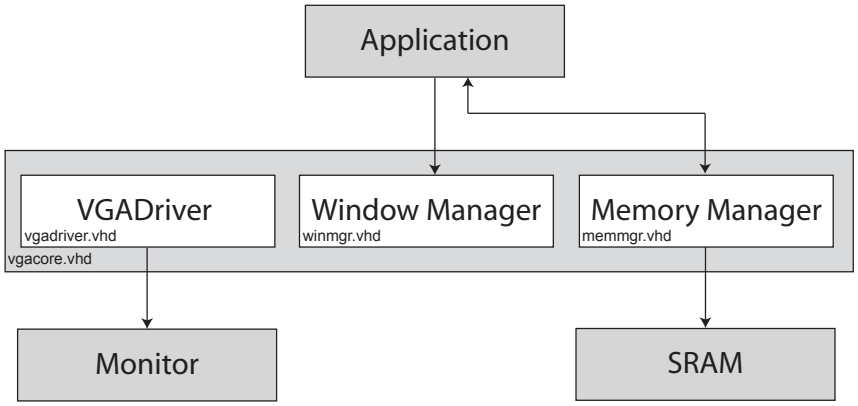

Figure 2-1: VGA core - scheme

These three parts are linked together in a top file (`vgacore.vhd`) that provides the most important signals to the user. None of the input data is registered, that means window positions, etc. have to be kept as long as they are valid. Changes in position and size are applied immediately, which can result in an unstable picture for one frame.

The next sections describe the parts and their respective interfaces in detail.

2.5 VGA core

The VGA core, found in `vgacore.vhd`, is the interface to the tasks and handles the internal connections between the three units stated above. Signals provided to the user are described below in the pertaining sections.

The VGA core needs one of Virtex2's BlockRAMs for local buffering. This is needed to burst lines from RAM into a local buffer, from where the actual drawing takes place.

Note: Using a DCM, these bursts could be made faster, such that less time is used by the VGA driver to access memory.
A DCM requires a buffered clock input using IBUFG. This, however, means that no other part of the design can directly access the original clock pad—instead, everything must work on the buffered clock that is also fed to the VGA core. It's thus needed to instantiate an IBUFG at the top level and provide the rest of the design with its output.

Signal	Meaning
`CLKxCI`	50 MHz clock signal
`RSTxRI`	Reset signal

Table 2-1: General signals

2.6 VGA driver

The VGA driver, `vgadriver.vhd`, has two concurrent processes: the first one buffers an entire line from memory into a FIFO at a clock period of 20 ns, the second one does the actual VGA drawing at 25 MHz from this buffer.

Monitor timing is done according to [13]. A pixel clock of 25 MHz yields a screen refresh rate of 60 Hz at $640 \cdot 480$ px. Higher resolutions or rates are possible. However, these would not only reduce the available memory bandwidth for applications but would also require an additional DCM to accommodate to the necessary pixel clock. Still, this is possible by adjusting the constants at the beginning of the file, in addition to clocking the process `VGACLKxCI_P` with an appropriate DCM.

To implement the FIFO buffer, one of the Virtex2's own BlockRAMs is used. A buffer size of 512 pixels (16 bit, thus $1'024$ bytes) is used. While this can't hold an entire line, drawing is started early enough to prevent buffer overflow.

Signal	Meaning
`RxD`	Red signal (8 bit)
`GxD`	Green signal (8 bit)
`BxD`	Blue signal (8 bit)
`PCLK`	Pixel clock for RAMDAC
$\overline{\texttt{HSYNCB}}$	Horizontal sync
$\overline{\texttt{VSYNCB}}$	Vertical sync

Table 2-2: Output signals to video DAC

Signal	Meaning
VGABlack	Set high for black screen
VGATest	Set high for test picture

Table 2-3: VGA driver, input signals

2.6.1 Line buffer

The FIFO used for buffering lines was generated from an asynchronous FIFO using Xilinx's CoreGen [19]. The file `linebuffer.xco` holds all necessary information. In order to compile it, one has to select the option "Regenerate core" in the Xilinx Project Navigator. Also, "View VHDL Function Model" might be necessary to get it working.

Please refer to section 2.10 on page 12 for detailed information on how to compile a test project with the VGA driver.

2.7 Window manager

The window manager, `winmgr.vhd`, receives screen coordinates from the VGA driver and then translates these to the appropriate memory addresses. A standard background color is used for pixels not covered by windows.

For each window (`W1x ...W4x`), the input signals stated in table 2-4 are provided. These are not registered and must thus be kept active throughout their life cycle.

Signal	Meaning
W1xX	X coordinate of window $(0 \ldots 639)$
W1xY	Y coordinate of window $(0 \ldots 479)$
W1xW	Width of window $(0 \ldots 639)$
W1xH	Height of window $(0 \ldots 479)$
W1xMemPage1	Base address of first buffer (20 bit)
W1xMemPage2	Base address of second buffer (20 bit)
W1xBID	Buffer to display (0 = first buffer)
W1xValid	Set high to show window

Table 2-4: Per-window signals

Two buffers can be defined by the user, with the signal `W1xBID` selecting which area to draw from. This can be used to implement double buffering: while one buffer is displayed, the other is updated with new information. As soon as this data is ready, the buffers are exchanged by flipping the `W1xBID` signal.

Besides the signals defined for all windows, there's two global signals, as seen in table 2-5.

First, the background color will fill the entire space not covered by windows. For testing purposes, a switch has been implemented which bypasses the memory manager and uses a single fill color for all windows instead.

Signal	Meaning
WxBG	Background color (16 bit RGB)
WxTest	Replaces contents of windows by a single color

Table 2-5: General signals

The background color is given as a 16 bit value, given in $5-6-5$ representation, as seen in table 2-6.

R	R	R	R	R	G	G	G	G	G	G	B	B	B	B	B
15															0

Table 2-6: RGB representation in a 16 bit word

2.8 Memory manager

The memory manager, `memmgr.vhd`, provides the actual interface to write image data into. The SRAM [1] on the left side of the R-FPGA is used as video memory.

Since the SRAM is not dual-port, a simple access management was implemented. General rule is that the window manager has highest priority and can suspend any other accesses to the memory. This ultimately means that at any given point in time, a task might have to wait for a period of a whole line to be fetched into the VGA driver's buffer.

This, and the possibility to have the VGA core itself run at a higher clock rate, led to the implementation of an asynchronous protocol for accessing the SRAM. As a side effect, the task doesn't have to worry about access timing for SRAM.
In section 2.8.1, read and write waveforms for accessing the memory manager are given.

Table 2-7 gives per-window signals for reading and writing data from/to the SRAM.

Note: read data is registered - i.e. as soon as `W1xACK` is set, the memory manager can continue serving other requests, while the data in `W1xDataOut` will stay valid.

Finally, the memory manager's output pins have to be connected to the left SRAM bank of the XFBOARD using the signals from table 2-8.

Signal	Meaning
W1xAddr	Address of memory (20 bit)
W1xDataIn	Data to be written (16 bit RGB)
W1xDataOut	Read data (registered) (16 bit RGB)
W1xREn	Read enable
W1xWEn	Write enable
W1xACK	Asynchronous ack

Table 2-7: Per-window signals for memory manager

Signal	Meaning
Addr	Address in SRAM
Data	Data
A19B	Inverted MSb of address
$\overline{\text{WEn}}$	Write enable

Table 2-8: Connections to SRAM

2.8.1 Timing diagram

Example write waveform

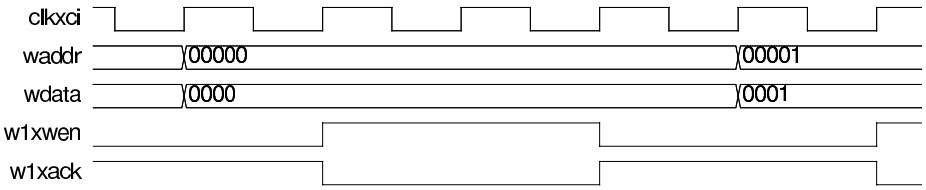

Figure 2-2: Write wave form

The write wave form seen in 2-2 was generated by the behavioural simulation of the demo program given in section 2.10.
To write, address and data have to be valid at the time W1xWEn is set to 1. Writing is finished as soon as W1xACK is set to 1. To actually finish the write process, W1xWEn has to be taken down again. This cancels W1xACK and makes the memory manager ready for new data.

The reading process is analogous to writing.

2.9 Internals

2.9.1 VGA driver internals

The VGA driver continously feeds the DAC with the necessary information to draw the screen. It can be switched off with `VGABlack`. The signal `VGATest` draws a test picture.
By default, the picture is read from memory, buffered locally and then drawn to the screen.

The interface of the VGA driver can be splitted into three parts: control interface, connections to the window manager and outputs to the video DAC. The window manager is controlled by the two signals `PixelX` and `PixelY`. Corresponding pixels are returned with a delay of 4 cycles. Then, the pixel value can be read from `PixelData`.

The screen resolution is determined by the constants in `vgadriver.vhd`. Timing data for several resolutions can be found in [13].

Table 2-9 describes the different clocks and their use.

Clock	Use
`RAMCLKxCI`	Used for communication with the memory manager
`VGACLKxCI`	50 MHz main clock of the driver
`OurClk`	25 MHz used for VGA output side, equivalent to the pixel clock `PCLK`

Table 2-9: VGA driver clocks

The following processes are found in the VGA driver:

- ReadReg and ReadMemless fill the line buffer if necessary. The delay of the memory manager has to be taken into account. This is done using the counter `FIFOWaiting`.
- The pair `VGACLKxCI` and `OurClk` are responsible for generating `OurClk` for $\overline{\texttt{HSYNC}}$, `HCount` and also for the read operation from the linebuffer.
- VCounter increments VCount after each horizontal synchronization signal
- VSyncB produces the vertical synchronization signal.
- Blitter is the main process. It either writes the `FIFOOut` to the output or generates the test picture.

2.9.2 Window manager internals

The window manager handles the translation of screen coordinates to memory addresses.

Instead of doing actual calculations for every pixel, a state machine is used to keep track of the current memory address for each window. Table 2-10 describes the states and their meanings.

State	Meaning
SSCAN	The current pixel is outside all windows
SWAIT	Waiting for the memory manager to fulfill a request
SDRAW	The current pixel is inside the window denoted by the signal user

Table 2-10: Window manager states

The four most important signals are the `inW1` ... `inW4` signals, which check for every pixel coordinate whether it falls inside a window. If this condition becomes true in SSCAN mode, reading from memory for the respective window is started.

For each of window, there is a 20 bit register `WxAddr` holding the current address. Upon starting screen redraw, these signals are reset to the respective base addresses `WxMemPage1` or `WxMemPage2` , depending on the `WxBID` flag.
Additionally, the register `MemAddr` holds the current memory address. As soon as a window is entered, this is set to the respective `WxAddr` . Inside a line, `MemAddr` can then just be incremented. Upon leaving the window's canvas, `MemAddr` is stored back again in the respective `WxAddr` register, so that drawing can be resumed in the next line at the correct address.

The data output to the VGA driver is determined by the flag `inWindow` . If it is set to 1, data from the memory manager is redirected, otherwise, the background color is fed to the VGA driver.
Note that the memory manager returns its data a few clocks later than requested, so the current pixel coordinates are not used to yield this decision. Instead, `inWindow` is set high after the first `SDRAW` state and taken down 2 cycles after returning to `SSCAN` state.

2.9.3 Memory manager internals

The memory manager abstracts from the actual SRAM, and handles concurrent accesses by both the tasks and the window manager. In any case, the window manager has highest priority.

There's four different states, seen in 2-11.

State	Meaning
SIDLE	Idle state, waiting for a request
SWAIT	Address set, waiting for SRAM
SACCESS	Actual access. If reading, store read data in register
SWRITE	Write data to actual address

Table 2-11: Memory manager states

Accesses to the SRAM by the tasks are handled in form of a simple Request-Acknowledge protocol. This is because in every state, when the window manager issues an request, other accesses have to be suspended in order to ensure a stable picture. This means that in the worst case, a task has to wait for an entire line to be bursted from SRAM before its own request can be fulfilled.

The signal `user` indicates the current user of the SRAM. Depending on this, the register `RAddr` holds the address currently fed to the SRAM.

Timing of reading and writing is done according to [1], assuming a 50 MHz clock.

2.9.4 Timing

Since both window manager and memory manager are states machines waiting for a request from outside, timing is slightly tricky. It takes a few clocks for a pixel request from the VGA driver to be served. But as a pixel has to be served every two cycles (assuming the XFBOARD's standard 50 MHz clock), this process has to be pipelined. The resulting timing diagram can be seen in figure 2-3.

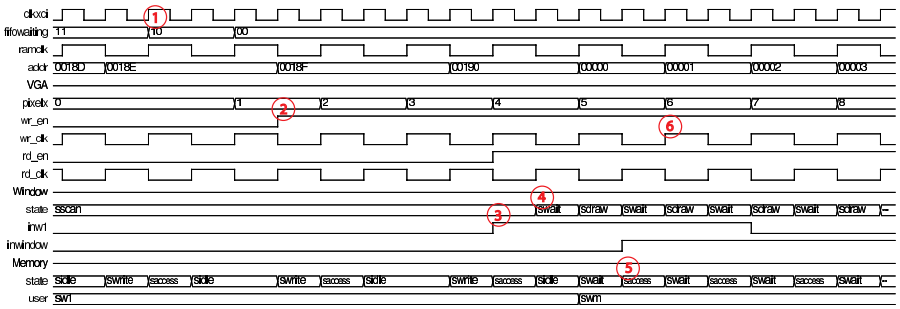

Figure 2-3: VGA internal timing - overview

Here is a rundown of what is happening in the above wave form:

1. When drawing of a line is started, `FIFOWaiting` is decremented.
2. One cycle after `FIFOWaiting` reaches 0, the writing to the buffer is started.
3. Meanwhile, `PixelX` is incremented, and upon reaching the 4th pixel (in this example), the `inW1` flag is set.
4. The window manager changes from scanning into drawing state, passing on requests to the memory manager.
5. The memory manager delivers the data read with `SACCESS` .
6. Data is written on rising clock edge into the FIFO.

2.10 Demo application

In order to demonstrate the usage of the VGA core, we compiled a small demo application that writes a pattern into video memory, and displays a test picture upon pressing the XFBOARD's switch.

The following code framgment shows the part that is used to write the pattern to video memory using the memory manager.

```
-- This example state machine writes stripes into the video memory
--  WAddr ...   address to write to
--  WData ...   data to be written
--  W1xWEn ...    write enable signal
--  State ...   current state
reg : process(CLKIN_IBUFG, RSTxRI)
begin
  if RSTxRI = '1' then
    State <= SADDR;
    WAddr <= (others => '1');
  elsif CLKIN_IBUFG'event and CLKIN_IBUFG = '1' then
    State <= nextState;
    WAddr <= nextWAddr;
  end if;
end process;

memless : process(State, WAddr, W1xACK)
begin
  nextWAddr <= WAddr;
  nextState <= State;
  W1xWEn <= '0';

  -- Combinatorial calculation of data based on address
  if WAddr(0) = '0' then
    WData <= (others => '0');
  else
    WData(3 downto 0) <= WAddr(3 downto 0);
    WData(14 downto 4) <= (others => '0');
  end if;
  WData(15) <= WAddr(19);

  case State is
    when SADDR =>
      -- 1st step: if ACK is down again, mem manager
      -- is ready to receive new data: set write enable
      if W1xACK = '0' then
        W1xWEn <= '1';
        nextState <= SWAIT;
      end if;

    when SWAIT =>
      -- 2nd step: wait for ACK
      W1xWEn <= '1';

      if W1xACK = '1' then
        -- ACK received, take down request
        -- increment address
        nextState <= SADDR;
        nextWAddr <= WAddr + 1;
        W1xWEn <= '0';
      end if;

    when others =>
      nextState <= SWAIT;
  end case;
end process;
```

2.10.1 Compiling the demo application

The demo application described above can be found on the accompanying CD, see chapter 6 on page 45.

Open the corresponding file `vgademo.vhd` in the Xilinx Project Navigator, and select “Generate programming file”.

2.11 Notes

For simplicity’s sake, this implementation of the VGA core uses a lot of signals, instead of providing an interface that would allow the user to pick a window and then use some control information to set the required datum.

Furthermore, the possibility is provided to clock the memory accesses at a higher speed, thus increasing the possible throughput. This was removed because of timing problems.

On the feature side, overlapping windows are not supported yet. This change should not be too difficult, but was left off due to the fact that it is not that important.

2.11.1 Microblaze

In [15], a few routines were developed for the Microblaze soft processor that allow access to the VGA driver’s most important signals. These routines were used in both the spectrum analyzer and turbo decoder task for displaying their respective output.

2.11.2 Font routines

The following routines can be used to write out text or numbers inside a window. A $6 \cdot 8$ font is provided, but other fonts can be defined. All of these routines take advantage of the pixel writing routines developed in [15].

The common parameters to all these functions are the base pointer of the window, the color of the font, as well as the x, y position to plot the character(s) at. In order to correctly calculate the address inside the window, the width has to be specified as well.

```
void f_putc(Xuint32 baseptr, Xuint16 color, Xuint16 x,
            Xuint16 y, Xuint16 wdt, char c);
```

Prints a single character.

```
Xuint16 f_puts(Xuint32 baseptr, Xuint16 color, Xuint16 x,
               Xuint16 y, Xuint16 wdt, char* s);
```

Prints out an entire string and returns the x position directly following the strings.

```
Xuint16 f_putnum(Xuint32 baseptr, Xuint16 color, Xuint16 x,
                 Xuint16 y, Xuint16 wdt, Xuint32 num, Xuint8 base);
```

Prints out the number `num` in the corresponding `base`. Returns x position directly following the printed number.

```
void f_printf(Xuint32 baseptr, Xuint16 color, Xuint16 x,
              Xuint16 y, Xuint16 wdt, const char* ctrl1, ...);
```

Writes out a formatted string, with the same basic arguments inside the format strings as the ANSI C function printf.

Chapter 3

Spectrum Analyzer

3.1 Introduction

As a first signal processing task for the RHWOS, a spectrum analyzer was implemented. It reads an audio signal from the audio driver and displays the corresponding spectrum on the screen. In the following sections, the general layout of this task is described.

3.2 Features

- 44.1 kHz, mono audio input
- 16 bit resolution
- $1'024$ point FFT
- Output using VGA driver (32 bars displayed on-screen)

3.3 Overview

The spectrum analyzer is partitioned into 3 hardware parts and control software running on an instance of the Microblaze soft processor.

- wrapper for the audio driver
- FFT core
- root core
- Microblaze with control software and graphics output

3.4 FFT core

To perform the Fourier transfomation, a Xilinx core is used. Therefore, it was only necessary to study the interface and develop a suitable wrapper. The wrapper, seen in figure 3-1, reads data from an input FIFO, calculates the FFT and writes the result to an

output FIFO. Both FIFOs have to provide enough space for $1'024$ values. As the FFT calculates a complex number with imaginary and real parts of 16 bits each, the output has to be 32 bits wide.

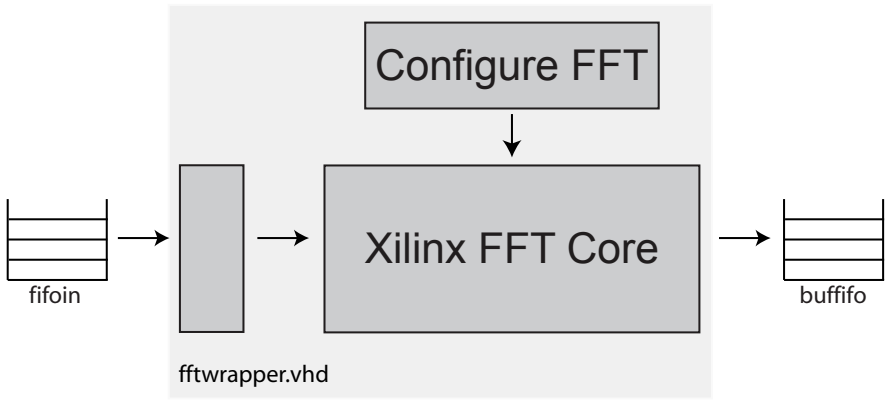

Figure 3-1: FFT wrapper

Table 3-1 describes the signals to control the wrapper.

Signal	Meaning
`go`	Start calculation. A full input FIFO expected.
`finished`	Set if data in output FIFO is ready.
`busy`	Set while core is calculating.
`idle`	Set when core is idle.
`rerror`	Core could not read enough data. Trigger `go` to return to idle state.
`werror`	Write error occured. Trigger `go` to return to idle state.

Table 3-1: Control interface for FFT wrapper

The wrapper is found in `fftwrapper.vhd`. There, it is possible to change core options, e.g. the size of the FFT and scaling options. For more details, refer to the Xilinx core documentation [20].

3.5 Root core

The output of the FFT core consists of $1'024$ complex values. As the spectrum of a real signal is symmetric, only the first 512 values need to be processed by the root core, as seen in figure 3-2. It calculates the absolute value by squaring both imaginary und real part, sum them up and subsequently extracting the root from this number. Last, tuples of 32 values are taken and averaged together to form a total of 32 output values. These

are the values that actually will be displayed.

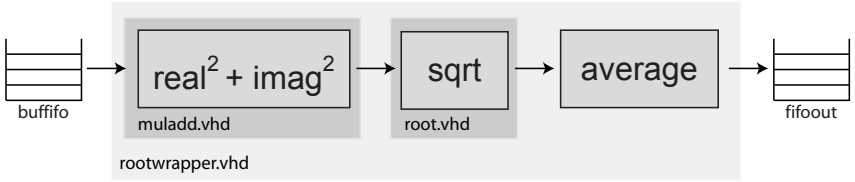

Figure 3-2: Root wrapper

The interface given in 3-2 is very similar to the one of the FFT core.

Signal	Meaning
`go`	Start calculation. A full input FIFO is expected.
`finished`	Set if data in output FIFO is ready.
`idle`	Set if core is idle.
`rerror`	Core could not read enough data. Trigger `go` to return to idle state.
`werror`	Write error occured. Trigger `go` to return to idle state.

Table 3-2: Control interface for root wrapper

The main challenge of this part was the extraction of roots. As there was no already running core available, this was implemented by hand. The squaring of imaginary and real part is done using two Virtex2 block multipliers. The number of values to be averaged is adaptable by a constant. The depth of the output FIFO then has to be changed accordingly.

3.5.1 Extracting root in Hardware

The root algorithm calculates one bit per cycle, starting from the MSb.
At the beginning, the result is set to zero and the current bit to the MSb. Then, the algorithm checks if the square of the current result with current bit set is smaller than the input. If this is the case, the new result value is the old one with set current bit. This process is now repeated, decrementing the current bit until it reaches the LSb. The result is the integer root of the input.

3.6 Putting it all together

The three components audio, FFT and root are put together in one top file. The single components are connected by FIFOs. Figure 3-3 gives an overview over the data flow.

The control interfaces are connected to the Microblaze core. Furthermore, `top.vhd` introduces logic for the Microblaze to read from the output FIFO. The following code fragment shows the code for one transformation.

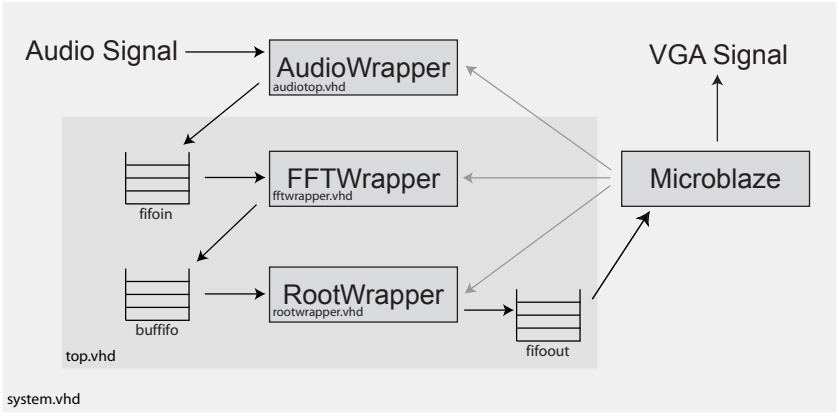

Figure 3-3: Spectrum analyzer - data flow

```
// wait till input fifo full
while(!(XGpio_mGetDataReg(XPAR_SPEC_CONT_BASEADDR) & FFT_IN_FULL));

// start fft
XGpio_mSetDataReg(XPAR_SPEC_CONT_BASEADDR, FFT_GO);
XGpio_mSetDataReg(XPAR_SPEC_CONT_BASEADDR, 0);

// wait till fft is finished
while(!(XGpio_mGetDataReg(XPAR_SPEC_CONT_BASEADDR) & FFT_FIN));

// start root
XGpio_mSetDataReg(XPAR_SPEC_CONT_BASEADDR, ROOT_GO);
XGpio_mSetDataReg(XPAR_SPEC_CONT_BASEADDR, 0);

// wait till root is finished
while(!(XGpio_mGetDataReg(XPAR_SPEC_CONT_BASEADDR) & ROOT_FIN));

// read 32 values from output FIFO to Microblaze
for(i=0; i<32; i++) {
  rd_clk = rd_clk ^ 1;

  // read in next value by toggling rd_clk
  XGpio_mSetDataReg(XPAR_SPEC_CONT_BASEADDR, rd_clk);
  // value is scaled by shift left 12, not yet optimal value!
  bars[curbar][i] = XGpio_mGetDataReg(XPAR_SPEC_IN_BASEADDR) >> 12;

  // avoid bar overflow
  if(bars[curbar][i] > 64) {
    bars[curbar][i] = 64;
  }
}

drawBars(0, bars[curbar^1], bars[curbar]);
curbar ^= 1;
```

3.7 Limitations

Both FFT and root wrapper have been thoroughly simulated and tested. When feeding them with a square wave, the output seems correct.
However, when trying to feed the system with the audio driver's output, nothing happens. There seems to be some very low signal (presumably noise), but we did not

manage to receive the actual signal fed to the audio input.

3.8 Further steps

To actually get this task working, the audio wrapper has to be revised. As soon as this is working, only the software part should need some adaptation concerning the scaling of the bars.
Other than that, we are confident that the system works.

Chapter 4

Turbo Decoder

The Virtex2 FPGA chip [16] used on the XFBoard offers a very good basis for signal processing applications, since the V3000 version used for the R-FPGA has 96 onboard 18x18 bit multipliers. Using these multipliers in parallel, a great deal of concurrency can be achieved, improving performance significantly in dataflow-oriented applications.
In our case, some parts of the system use up to 18 multipliers in parallel, which resulted in almost the same performance as the corresponding PC application, even at the modest clock rate of 50 MHz.

In the scope of this project, a turbo decoder according to the UMTS specifications was implemented.

4.1 Turbo codes - overview

Turbo codes were introduced in 1993 by Berrou, Glavieux and Thitimajshima [3].
The turbo codes considered in this project are based on 2 parallel concatenated convolutional codes. The upper encoder is fed directly with the input sequence, while the second encoder receives an interleaved version of the same information sequence.

4.1.1 Turbo encoder

Figure 4-1 shows the encoder suggested by the UMTS specifications. It adds redundancy to the input signal, such that the decoder can infer on the original signal despite noise introduced by the channel.
The basic idea is to split the input into blocks of length K, and transmit multiple bits per single input bit over the channel. While $x[k]$ is the same as the data input to the encoder, $z[k]$ is formed by a convolutional encoder working directly on the input sequence. Additionally, each block of the input is permuted by an interleaver, and then fed to another convolutional encoder to form $z'[k]$.

One of the main design issues for a linear code is to maximize the hamming weight. This is roughly the same as maximizing the weight of the minimum-weight codeword. On one hand, recursive codes are used for this, on the other, an interleaver.

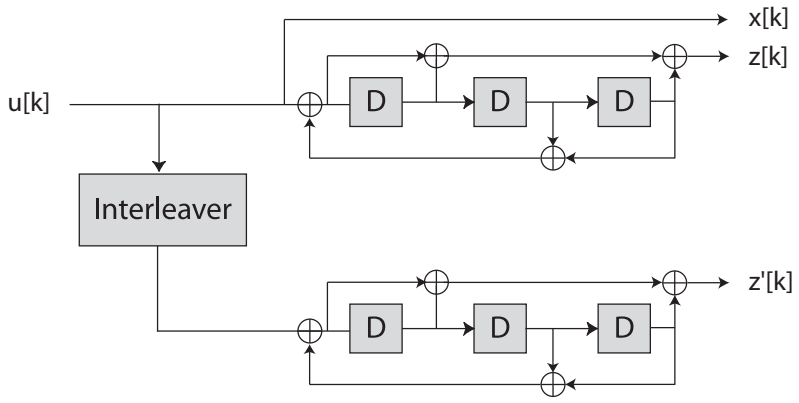

Figure 4-1: Turbo encoder according to UMTS specifications

The interleaver permutes the bits inside a block, and then feeds the second encoder with this interleaved block. This ensures that two successive input bits will never be next to each other in the interleaved version.

Moreover, by using an interleaver, extrinsic information and observations of the two decoders are only weakly correlated. Thus, they can both be used for decoding. For further details, please refer to [11].

4.1.2 Channel

Figure 4-2 shows an overview of the entire sender-receiver system. Input data $x[k]$ is fed in blocks to the encoder. As described in the previous section, this will yield two additional bits $z[k]$ and $z'[k]$.
Using a modulator, 0 bits are mapped on to $+1$, 1 bits on to -1. This data is then sent over an additive white Gaussian noise (AWGN) channel. As a result, the receiver will see bits $y_x[k]$, $y_z[k]$ and $y_{z'}[k]$. Channel estimation is applied in order to gain probabilities whether the received information is more likely a 0 bit or a 1 bit.
The decoder can use this channel information to form an estimate for the original sent bit, $P(x[k]|y_x[.], y_z[.], y_{z'}[.])$.

4.1.3 Turbo decoder

The turbo decoder can be represented as a factor graph, as seen in figure 4-3[1]. The algorithm implemented in hardware is very similar to the sum-product algorithm— introduced in [6]— running on this factor graph.
The two decoders basically run in parallel, seperated by the interleaver that exchanges

[1]The quantities $X_{1,k}$, $X_{2,k}$ and $X_{3,k}$ correspond to the channel estimates for $y_x[k]$, $y_z[k]$ and $y_{z'}[k]$, respectively.

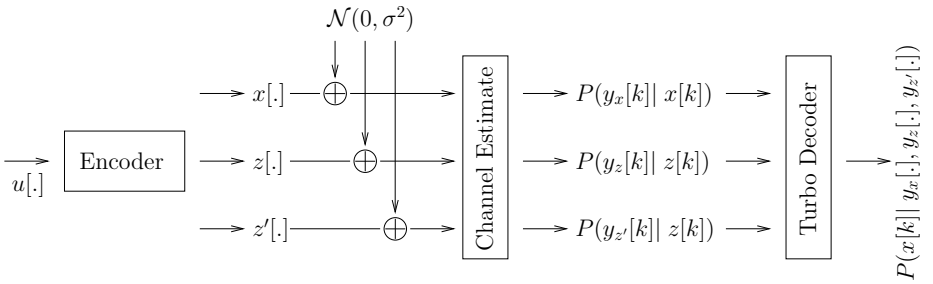

Figure 4-2: Sender-receiver system with AWGN channel

the extrinsic information between the two. Equal gates are used to combine input information (from the received data) with this extrinsic information.

Each box in the factor graph corresponds to a trellis section. Based upon the encoder, the trellis can be constructed.

In order to maximize the a-posteriori probability of each information bit, the forward-backward algorithm is used inside each single decoder. Then, the newly gained information is exchanged with the other decoder.

After several iterations, a stable state is reached. A hard-decision is then made by comparing $P(x[k] = +1|y_x[k], y_z[k], y_{z'}[k])$ against $P(x[k] = -1|y_x[k], y_z[k], y_{z'}[k])$.

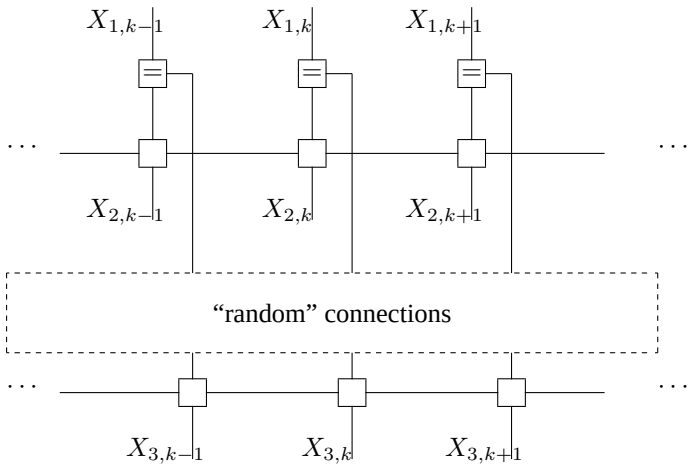

Figure 4-3: Turbo decoder - Factor graph

4.2 Implementation

The implementation of the turbo decoder presented here was a three-step process.
First, a software version, written in C, was written to evaluate the design later to be translated to hardware. This mainly covered the analysis of sections with large amounts of dataflow. These are prone to a hardware-based implementation. At this point, floating-point arithmetic was used.
Next, the C code was adapted to use fixed-point arithmetic, as would be the case in hardware as well. This step was useful in order to analyze the behaviour of the algorithm with less arithmetic precision.
In the final step, the most time-consuming parts—both interleaver and the forward-backward algorithm—were translated to hardware. Only the control flow between decoding iterations was left to a C program running on the Microblaze [18] soft processor.

4.2.1 Code in use

The turbo code used for this project was constructed according to the UMTS specifications [4]. A rate-$1/3$ code with a block length of $K = 4'900$ was chosen. Besides nicely fitting a 70x70 black and white picture for our testing environment, a higher block length also provides better performance.

4.2.2 Forward-backward algorithm

While the factor graph seen in 4-3 suggests a parallel execution of both decoders, a serial approach was chosen for several reasons:

- The software version inherently had to work in a serial fashion. The plan was to derive the hardware version directly from it.
- Even though enough arithmetic units were available in hardware, early analysis showed that memory would be the main bottleneck. This is further explained in section 4.5 on page 37.
- A single instance of the forward-backward algorithm almost fills an entire task slot of the RHWOS. Thus, chip area (as restricted by the RHWOS) was not sufficient to run both decoders at the same time.

Linearized form

Instead of running the two decoders in parallel, the information flow was changed to:

- Run decoder 1 on $y_x[.]$ and $y_z[.]$
- Interleave its extrinsic information, send it to decoder 2
- Run decoder 2 on $y_{x'}[.]$ and $y_{z'}[.]$
- Deinterleave its extrinsic information, send it to decoder 1

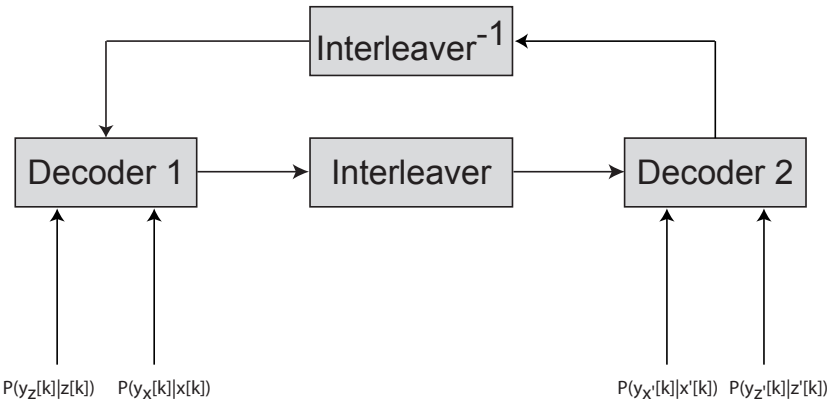

Figure 4-4: Internal flow of turbo decoder

Note that the received data only covers $y_x[.]$, $y_z[.]$ and $y_{z'}[.]$. The factor graph representation elegantly provides decoder 2 with an interleaved version of $y_x[.]$ on the fly. In this implementation, $y_{x'}[.]$ was explicitly derived from $y_x[.]$ before starting the actual iterations. This is depicted in 4-4.

With the control flow between the two decoders resolved, the next step was to actually implement the decoder internals, i.e. the forward-backward algorithm.

Numerical considerations

Generally, literature suggests two approaches: the preferred implementation uses a log-likelihood approach, implicitly allowing for numbers of very different magnitude to be represented in a smaller interval. This lends itself very well to a fixed-point, hardware based representation. This is discussed in [8].
The difficulty of this approach is the log-likelihood arithmetic, which is not that intuitive and easy to implement in hardware—approximations are suggested e.g. in [11].

Thus, an implementation using messages based on probabilities was chosen. This simplified the calculations inside the trellis a lot, as only standard multiplications and additions were needed.
However, there is an inherent loss of information when trying to represent real numbers as fixed-point numbers— preferably with few bits. Not only the integer representation of channel estimates, extrinsic information and internal signals proofs to be problematic.
The state metrics $\mu_f(s)$ and $\mu_b(s)$ converge quickly to 0 within a single iteration, and make the decoder implementation very prone to numerical underflows. To resolve this problem, two measures were taken:

- **Scaling.** After each section of the trellis, state metrics were rescaled such that for each section, the maximum state metric would equal the maximum number of the fixed-point representation. This scaling doesn't affect the actual output of

the forward-backward algorithm, [6].

- **Introduce a minimum number**. Still with scaling, numbers with a magnitude several times smaller than the current maximum $\mu_f(s)$ could wrongfully be assumed to be 0. It is thus suggested to have a lower bound for the numbers. While this introduces additional numerical errors to the calculation, it will prevent underflows. It is important to prevent these, as they would rendering the whole calculation completely unuseable.

These measures were slightly simplified for the hardware version:
After the actual calculation for a trellis section, scaling is done by bit-shifting the result to the left until the most significant bit of one message is set.
Instead of bounding all calculations by introducing extra comparators, logic was saved by simply fixing the lowest input bit of all multipliers to 1.

Additionally, by setting the SNR of the channel estimation smaller than the one suggested by the channel's actual SNR, one basically "trusts" the calculation less. This can compensate for numerical errors as well.

Data-flow analysis

The exact application of the forward-backward algorithm is described in [6]. Here, a small example for calculating a single state metric is given. This is then used to point out ways to introduce parallelism into the calculation.

Figure 4-5 shows a trellis section corresponding to one input-bit $x[k]$. State metrics for a state s are designated as $\mu_f(s)$, branch metrics for a branch b as $\mu(b)$.
For a branch b, lst(b) and rst(b) define the left and right state, respectively.

Consider the states on the right side of figure 4-5, i.e. states $s \in S[k]$. As an example, the new state metric for the state s_{111} is calculated.

Branches are designated with two numbers: the first one representing the value of the original information bit $x[k]$, the second for $z[k]$. Latter can be inferred from the encoder: depending on the state and information bit $x[k]$, the decoder's output $z[k]$ will differ. This is used to gain more confidence inside the trellis: not all combinations of 1's and 0's can be used.
There are two branches leading to s_{111}: the first one, b_0, associated with $y_x[k] = +1$ and $y_z[k] = +1$; the second one, b_1, with $y_x[k] = -1$ and $y_z[k] = -1$.
$\mu_f(s_{111})$ can now be calculated as follows:

$$\mu_f(s_{111}) = \mu_f(\text{lst}(b_0)) \cdot \mu(b_0) + \mu_f(\text{lst}(b_1)) \cdot \mu(b_1)$$

The branch metrics $\mu(b_0)$ and $\mu(b_1)$ combine the channel estimates for $y_x[k]$, $y_z[k]$ and the extrinsic information gained from the other decoder, here called $P(x[k] = +1)$ and $P(x[k] = -1)$, respectively.

The extrinsic information, as well as the channel estimates for $y_x[k]$, are derived from the same information bit. Thus, they are first combined by an equal gate, here denoted as equ$(+1)$ and equ(-1).

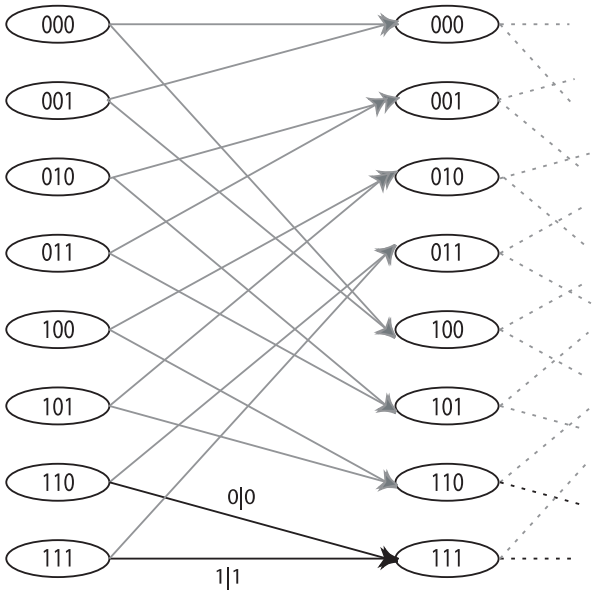

Figure 4-5: Section of a trellis

$$\text{equ}(+1) = \frac{P(y_x[k]|x[k]=+1)P(x[k]=+1)}{P(y_x[k]|x[k]=+1)P(x[k]=+1) + P(y_x[k]|x[k]=-1)P(x[k]=-1)}$$

$$\text{equ}(-1) = \frac{P(y_x[k]|x[k]=-1)P(x[k]=-1)}{P(y_x[k]|x[k]=+1)P(x[k]=+1) + P(y_x[k]|x[k]=-1)P(x[k]=-1)}$$

Note that the denominator is only for scaling purposes—it is omitted in the hardware version.
Furthermore, the result of the equal gate has to be computed only once for each section of the trellis.

Now, the branch metrics can be calculated as:

$$\mu(b_0) = P(y_z[k]|z[k]=+1) \cdot \text{equ}(+1)$$
$$\mu(b_1) = P(y_z[k]|z[k]=-1) \cdot \text{equ}(-1)$$

Seeing that there are 8 states for each $S[k]$, the above already suggests an obvious point to introduce parallelism. Basically, for a given $S[k]$, all $\mu_f(s)$ with $s \in S[k]$ can be calculated in parallel.
Since it is advisable to have only one multiplication per cycle, the calculation is done in three cycles, as seen in table 4-1. A peak parallelism of 18 multiplications at the same time is reached.

SFCALC1	Calculate result of equal gate (2 multiplications). For each branch b, multiply the channel estimate of $y_z[k]$ with corresponding state metric lst(b) (16 multiplications)
SFCALC2	Multiply previous temporary results with equal gate (16 multiplications)
SFCALC3	Infer $\mu_f(s)$ by adding together previous temporary results

Table 4-1: States of calculation

For the multiplications, the Virtex2's on-board 18x18 bit multipliers were used. As the actual data was only 8 bit wide, advantage was taken of the higher precision in intermediate calculation steps. This means that after the first calculation step, 16 bit numbers were used, the final result being 32 bit. Because of this, scaling only had to be applied at the very end of a trellis section.

4.2.3 Interleaver

Original implementation

The software implementation of the interleaver works exactly as the algorithm described in the UMTS specifications [4]. This yields a permutation array of size K, where each entry contains the new position of the corresponding interleaved bit.

Less memory

Because of its memory usage, it is not feasible to implement this algorithm in hardware. Therefore, an adapted version, which calculates the offsets on the fly, was written. This is only possible due to the fact that the forward-backward algorithm only accesses the data sequentially.

To reach sufficient speed and at the same time, minimize the use of resources, the calculations performed per step were limited to simple additions. The rest of the calculations were kept in pre-calculated tables.
The result is code that uses less than 300 constants and 20 variables. This is an improvement of more than a factor of 10, compared to the original memory usage. The main idea of the modified algorithm is that on each row, a specific row increment modulo $(P-1)$ has to be added.

C implementation

This section deals with the adapted version of the interleaver. For simplicity's sake, the backward iteration is not discussed - it can be obtained by mirroring the following program code.

The following defines a function which stores the interleaved version of `datasrc` in `datadest`.

```
void interleave(int *datasrc, int *datadest)
```

First, some constants are defined. s is an array with the standard permutation for one row.

```
int s[P] = {
  0, 5, 35, 215, ... , 91, 49, 48, 42, 6, 41
};
```

Another array holds the increment for each row. This guarantees that each row is permuted differently.

```
int row_inc[ROWS] = {
  1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
  53, 59, 61, 67, 71, 73, 79
};
```

Each row has its own offset which is added to the permutation value. Therefore, all offset values between 0 and $4'899$ are generated.

```
int row_offset[ROWS] = {
  4750, 2250, 3500, 1000, 0, 500, 1250, 1750, 3000, 4500,
  2500, 2000, 3250, 4250, 750, 250, 4000, 1500, 3750, 2750
};
```

Actual code:

```
// current value for each row
int row[ROWS];

// variables
int j, i;
int count;

// init
for (j=0; j<ROWS; j++)
  row[j] = 0;
count = 0;

// iterating through matrix

// for all columns calcualte next value in row
for(i=0; i<COLS; i++) {

  // for all rows calc val
  for(j=0; j<ROWS; j++) {

    if((count < K) && (s[row[j]] + row_offset[j] < K)) {
      // if we're not yet finished (count < K)
      // and the result is valid then permute

      // s[row[j]] + row_offset[j]
      // is now the interleaved value of count
      datadest[count] = datasrc[s[row[j]] + row_offset[j]];

      // further value written inc counter
      count++;
    }

    // calculate row permutation for next column
    row[j] = row[j] + row_inc[j];

    // perform modulo operation
    if(row[j] >= (P-1))
      row[j] -= (P-1);
  }
 }
}
```

Hardware version

The hardware version is the adaptation of this improved C code to VHDL.

The interleaver allows to read and write the data in both forward and backward direction, in either sequential or interleaved order.

An operation in interleaved mode normally takes 2 cycles, in special cases up to 4. For further information, please refer to the accompanying CD, described in chapter 6 on page 45.

Adaptation to different blocksizes

It is possible to adapt the interleaver to different block sizes K with just a few changes in the code. Respective constants for K, P, etc. can be found in [4]. To simplify the adaptation, a software to calculate the constant arrays introduced above was developed. This is described in section 6 on page 45.
When changing K, it is also advisable to change the size of the BlockRAM used for the extrinsic information. This has to be done using the Xilinx CoreGenerator.

4.2.4 Memory considerations

Memory wise, the XFBOARD offers three possibilities for a task to store its data in:

- **BlockRAM.** Found on the Virtex2 chip, this memory is fast, but very limited—a total of $1'728$ Kbits is available on the chip. The RHWOS will offer a maximum of 576 Kbits, i.e. 72 KB, to a task.
- **SRAM.** A total of 4 MB is available, 2 MB on either left and right side of the chip. Former is used for the VGA core, latter is still free.
- **SDRAM.** A total of 64 MB is available, but wasn't considered due to the more complicated access and refresh control.

Note that SRAM and SDRAM is abstracted from using a basic memory manager that is later to be replaced by the actual RHWOS' memory manager.

The data to be stored by the turbo decoder is

- **Extrinsic information.** $4900 \cdot 2 \cdot 8$ bits
  For each original information, both $P(x[k] = +1)$ and $P(x[k] = -1)$ has to be stored.
- **Channel estimates.** $4903 \cdot 4 \cdot 2 \cdot 8$ bits
  This is used for storing the values $P(y_x[k]|x[k] = +1)$, $P(y_x[k]|x[k] = -1)$, and the corresponding estimates for $y_z[.]$, $y_{z'}[.]$, $y_{x'}[.]$.
- **State metrics.** $4904 \cdot 8 \cdot 2 \cdot 8$ bits
  Used to store $\mu_f(s)$, $\mu_b(s)$. For every section of the trellis, there's 8 states, and two seperate memory areas are used for the forward and backward iteration.

This amounts to total a memory usage of $127'488$ bytes. In section 4.5 on page 37, a way to drastically reduce this figure is presented.

We opted for the extrinsic information to be stored in BlockRAM, which also simplified the interleaver. Other data is stored in SRAM, including space to store the extrinsic information for both decoders—this is of interest in the last iteration when the

extrinsic information of both decoders has to be combined for a hard-decision. This is currently done in software.

4.2.5 Putting it all together

The forward-backward algorithm, described in `fb.vhd`, as well as the interleaver, given in `interleaver.vhd`, form the main part of the decoder system. From a hardware point of view, these are complemented by the memory manager (`memmanager.vhd`), a basic memory interface (`basicmem.vhd`), as well as a top file (`top.vhd`) putting everything together.

Controlling the decoder

To control the turbo decoder, the top file provides the interface shown in 4-2.

Signal	Meaning
`fb_interleaved`	Switches the forward-backward algorithm to interleaved mode, i.e. data used is $y_{x'}[.]$ and $y_{z'}[.]$, and the interleaved extrinsic information. Thus, this is basically switching between the two decoders.
`fb_start`	Start a forward-backward iteration
`fb_done`	Indicates the end of an iteration
`int_init`	Initialize extrinsic information within interleaver to 0.5
`int_rst`	Reset interleaver to start at first information bit again
`int_tomem`	Write extrinsic information to memory, storage location depends on `io_interleaved`
`int_done`	Acknowledges interleaver operations
`io_write`	Write flag
`io_interleaved`	Interleaved flag. When not set, data is read/written in sequential, otherwise in interleaved order
`io_dataclk`	Data clock, data is transmitted on falling and rising clock edge
`io_priors`	Reads extrinsic information from memory, information from both decoders is transmitted in one 32 bit value
`io_xz`	Write or read channel estimates for $y_x[k]$ and $y_z[k]$ to memory, signal depends on `io_write` and `io_interleaved` flags
`io_done`	Set if input/output operation is finished
`input`	32 bit input bus, used for write operations
`output`	32 bit output bus, used for read operations

Table 4-2: Controlling signals

To transmit data between the Microblaze and the decoder, a 32 bit wide interface is used. As the Microblaze is the slower part, it generates the data clock. To maximize performance, data is sent at rising and falling clock edge.

Memory manager

The memory manager is used for accessing the channel estimates $P(y_x[k]|x[k] = +1)$, ...and the state metrics $\mu_f(s)$ and $\mu_b(s)$. It abstracts from the actual memory by requiring another entity that gives access to data according to an address and a request line.

Since the forward-backward algorithm accesses the data sequentially (either in forward or backward direction), the control signals were limited to a reset, step and acknowledge signal. The memory manager will then translate these to memory addresses (basis addresses are defined as constants at the beginning of the architecture), redirect the request to the basic memory interface, and pass on the acknowledge to the forward-backward algorithm as soon as the data is read or written.

Note that especially the basic memory interface is platform dependent, and when porting the program to another system using the Virtex2 FPGA, this has to be changed accordingly.

4.3 Testing environment

To test our implementation, we opted for an environment as seen in figure 4-6. The general idea was to simulate a noisy channel on the PC by adding random Gaussian noise to the encoded data and to send this noisy data over Ethernet to the XFBOARD to be decoded.

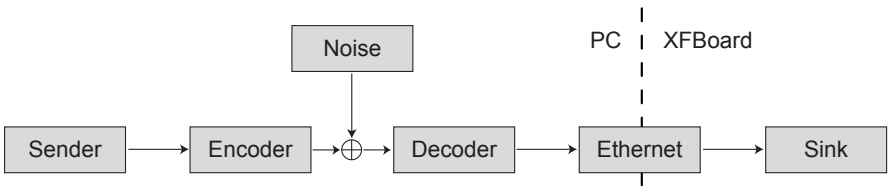

Figure 4-6: Testing environment

Note that the signal never leaves the digital domain. This would not be the case in an actual scenario. The testing application adds digital noise and sends the noisy signal to the decoder. In an actual application, a quantizer of some form would receive the analog information and then feed the decoder with digital input.

4.3.1 PC application (encoder)

On the PC side, a simple Windows application was programmed that allows the user to:

- load a $70 \cdot 70$ pixel black and white bitmap
- encode it
- add variable Gaussian noise to it
- send it over Ethernet to the XFBOARD

The modulator maps bits 0 and 1 onto $+1$ and -1, respectively. These modulated bits are then transmitted by an AWGN channel.
In order to simulate the noisy channel, similar to the one that would effect a real transmitted signal, the encoded data is represented as an 8 bit integer in the range of -2.5 to 2.5.
This is then affected by additive white Gaussian noise with selectable variance. We opted to use the Mersenne Twister [7] for this task, as this reportedly outperforms the random generators found in standard libraries.

The distorted data is then sent to the XFBOARD via Ethernet. The original and uncorrupted information sequence is also transmitted for measurement purposes.

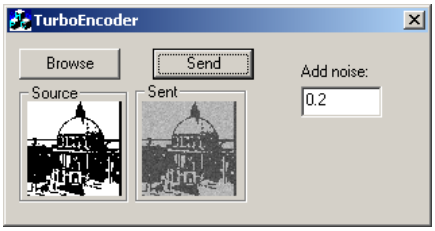

Figure 4-7: Encoder application

4.3.2 XFBoard (decoder)

The Microblaze part of the system receives the noisy, encoded bit-sequence from the PC application. Next, channel estimation is applied—this is done using a precalculated lookup table.
Then, the turbo decoder is set up. First, the estimated data is sent to the decoder to be stored in memory. Extrinsic information is initialized to a prior value of 0.5.

The actual decoding is controlled by the following code segment:

```
for (it=0; it<8; it++) {
  /* Decoder 1 */
  startFB(0);
  waitForFB();
  priorsToMem(0);

  /* Decoder 2 */
  startFB(1);
  waitForFB();
  priorsToMem(1);

  recvPriors();
  drawPic(win);
}
```

Note that after every iteration, the extrinsic information is written to memory by invoking the function `priorToMem`. This is only necessary as the progress of the decoder is shown after every iteration, i.e. a hard decision has to be formed. The function `recvPriors` reads the extrinsic information from the memory and forms this hard decision.

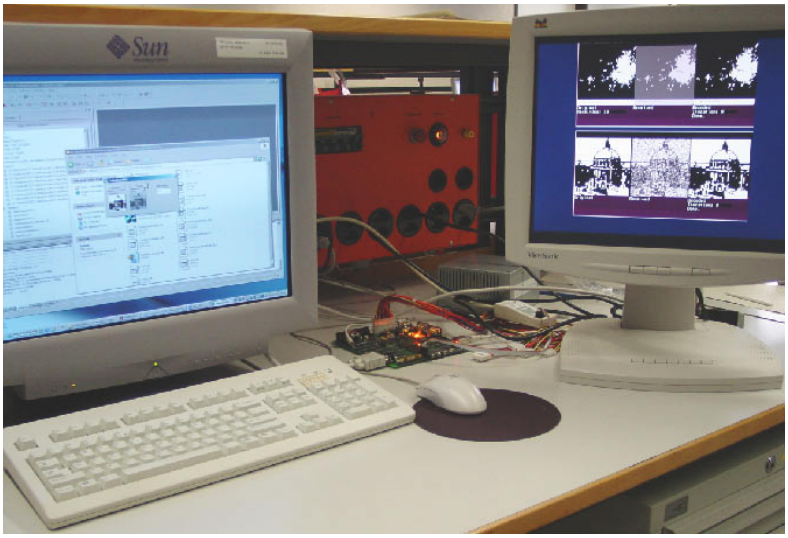

Figure 4-8: Testing setup

4.4 Results

4.4.1 Performance - bit errors

This section gives an overview over the performance of both the fixed point algorithm and the floating point one (figure 4-9).

SNR was calculated as

$$\text{SNR} = \frac{E_b}{N_0}$$

with $E_b = \frac{E_c}{R} = \frac{3K+12}{K}$ and $N_0 = \frac{P_N}{W} = 2\sigma^2$. Latter assumes an AWGN channel with double-sided power spectral density.

Both graphs were calculated for 8 iterations, using the following program fragment:

```
iterations = 8;
reach = 10000;
i = 0;
errors = blockerrors = 0;
for (;;) {
  if((p = test(sigma, iterations))) {
    errors += p;
    blockerrors++;
  }
  i++;
```

```
    if (i==reach) {
      if (blockerrors >= 100)
        break;
      else
        reach *= 2;
    }
  }
```

The routine `test` does the generation of the codeword, encoding and decoding. Then, a hard decision is done after the 8th iteration. The result is then compared to the original codeword, and the number of mismatches returned.

At least 100 blockerrors had to occur before the simulation was stopped. Also note that simulation is not stopped immediately, but rather when reaching a multiple of $10'000$.[2]

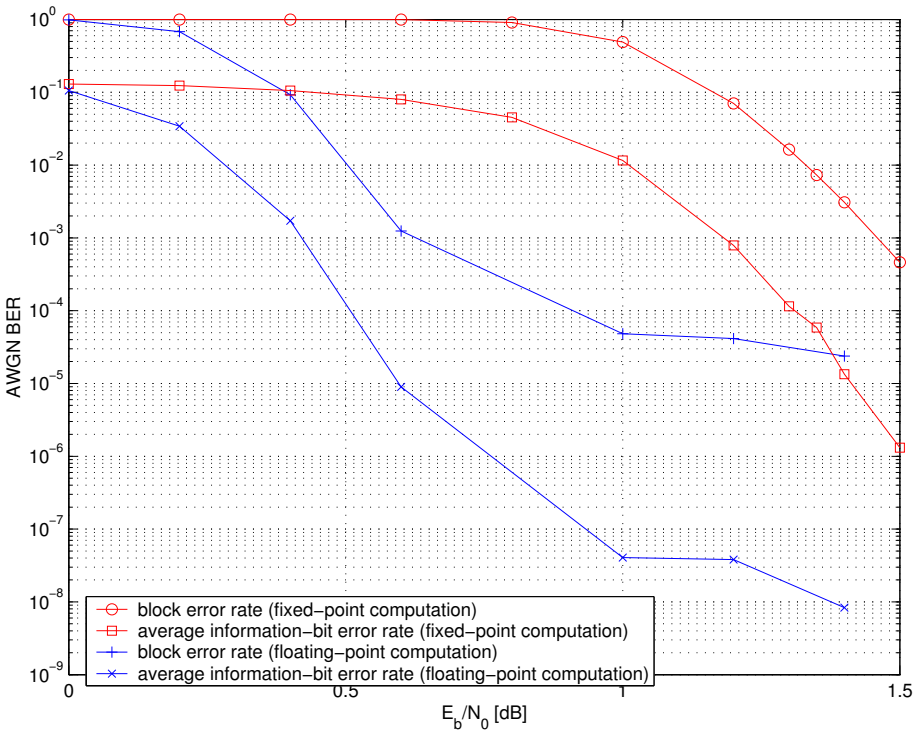

Figure 4-9: BER curve for turbo decoder implementations

The floating point version shows a significant advantage over the fixed point one. This is discussed in section 4.5.

4.4.2 Performance - execution time

Table 4-3 gives a short overview of the decoding-time used by the different implementations written in the course of this semester thesis.

[2] Floating point versions have to be stopped after about $300'000$ iterations, as time was running out.

For testing purposes, $10'000$ random codewords (= $6'125'000$ bytes) were generated and encoded. Then, the respective decoder implementation ran 8 iterations on the encoded codeword.
Please note that these numbers only give a rough idea of the performance—all implementations can still be optimized.

Implementation	Platform	Time
Floating point	AMD 1600XP	19 min
Fixed point	AMD 1600XP	64 min
Fixed point	XFBOARD	31 min

Table 4-3: Time used for decoding $10'000$ codewords

With the software version achieving a peak data rate of 5.18 KB/s, none of these implementations are suitable for actual applications.
Still, even at the modest clock rate of 50 MHz, the hardware implementation almost reaches the performance of the significantly higher clocked PC version. In section 4.5 on page 37, we suggest a few measures to gain enough speed for medium data rates.

4.4.3 ASIC

Without any doubt, a proprietary ASIC solution would be superior in speed, and even more in area and power consumption. But also FPGAs have their advantages:

- The manufacturing technology used for the Virtex2 FPGAs is $0.13\mu m$, current designs even use $0.09\mu m$. This state of the art technology is normally not applied to ASICSs which are produced in much lower quantities.
- An ASIC—in smaller quantities—is much more expensive than an FPGA. Additionally, all possible options for varying block lengths and the like already have to be hard-wired into the chip, which can increase the chip size significantly (assuming similar interleaver tables as we used).
  For an FPGA, a mere resynthesis can adapt the decoder to another standard, eliminating the need for parts that allow the selection of different codes.
- In addition to this, with a system as the proposed RHWOS, it would be possible to share the FPGA area with other processing tasks. This applies when the turbo decoder does not have to be available at all times, or can decode quickly enough into a buffer.
- Development of an FPGA system is easier, especially testing.

These points should be considered in the following sections when the weaknesses of FPGAs are discussed in more detail.

Execution time

Some parts of the FPGA work with optimal performance, e.g. multipliers and Block-RAM. The rest has quite a reasonable performance. Under the assumption that the VHDL code of the ASIC and the FPGA are more or less equivalent, the performance drawback is most likely below a factor of 4.

Other than that, the ASIC offers a lot more freedom concerning memory and multiplier elements. There are also fewer size restrictions. Therefore, it would be possible to implement several decoders in parallel, each with its own onboard memory. In terms of clock rate, the differences are not notable. This holds true especially because of the more recent manufacturing technology.

Area usage

Normally, it is assumed that only 10% of the FPGA die is used for actual logic, the rest is consumed by routing. Thus, an ASIC may be about 10 times smaller. But this is a very rough approxmation, depending on implementation and manufactoring technolgy. Another important point is the fixed size of an FPGA. As we use only 15% of the chip, we are certainly far away from an optimal space usage. As mentioned above, the rest of the chip could be used for other tasks.

Power dissipation

Power dissipation heavily depends on the FPGA technology used. The Virtex2 works with power consuming SRAM technology, whereas chips based on antifuse technology would be by far better. The disadvantage of the latter is that they can be configured only once, thus they were not suitable for this project.

Xilinx offers the tool XPower which calculates the actual power consumption of a design. Due to lack of time, we could not test it. But the approximated overall power consumption of the FPGA may be around $1'000$ mW, with 500 mW being static power consumption, independent of the actual configuration of the chip.
Again, an ASIC could be at least 10 times more efficient, probably even more.

4.5 Further steps

The aim of this project was to implement a turbo decoder on an FPGA. As only a limited amount of time was available, performance was not highest priority.
The following chapter presents some ideas for possible improvements. They can be organized into three parts: execution time, memory usage and decoding quality.

Memory usage

Currently, the memory usage of the decoder system is not optimal (approximately 120 kB are needed). Several steps can be taken to drastically whittle it down.

First, since the channel estimation is normalized to 1, only half the information has to be stored, as the other half can be calculated on the fly.

The same can be applied to the extrinsic information of the decoders when normalizing it before sending it to the next decoder.

Also, it would be possible to abandon the space needed for the backward iteration's state metrics by calculating the extrinsic information during the backward iteration.
In this implementation, SRAM had to be used anyway, so the simpler alternative of a

seperate calculation step for extrinsic information was chosen.

Finally, the current implementation overrides the least significant bit of the data read in most cases. Thus, if the precision is satisfying, 7 bit could be used instead of 8 to represent the data.

Overall, for $K = 4'900$ and 8 bit precision, a memory usage of $63'744$ bytes would thus suffice.

Execution time

The two main bottlenecks of the current architecture are memory bandwidth and the Microblaze soft processor. Also, the forward-backward algorithm has some room for improvements.

Concerning the Microblaze parts, not much can be done, except moving further parts to hardware or increasing the clockrate. Because of this, only improvements concerning the hardware part of the system are considered.

One iteration of the forward-backward algorithm takes approximately $668'500$ cycles [3]. $637'300$ of these are used for accessing memory.
The current memory interface is designed for simplicity rather than speed, taking between 4 and 5 cycles per memory access.[4] This figure could be reduced to 2 cycles, thus yielding $215'600$ cycles in total - a speedup of

$$S = \frac{31'200 + 637'300}{31'200 + 215'600} = 2.71$$

This problem hopefully will be resolved by the further development of the RHWOS and an accompanying memory manager.

Also, using some sort of cache architecture, one could ensure that the memory is busy all the time, saving some extra cycles. This turns out to be rather straight-forward, as the next value to be accessed is known in every step of the algorithm.
Another possible solution to achieve constant usage of memory would be pipelining of memory loading, calculation and storage.

The weak point of the forward-backward algorithm is the scaling of the state metrics. As of now, this is done in a loop that shifts all messages to the left by one bit until the most significant bit of one message is set.
A parallel version could bring considerable speedup - the current maximum bit can easily be found by ORing all the current state metrics. From this, the shift count can be inferred and executed in a single cycle.

Furthermore, calculation of the extrinsic information is done in a seperate step. By integrating this into the backward iteration, one would not only require less memory for storing the state metrics, but roughly, another $98'000$ cycles could be saved, resulting

[3] Based on simulating the behavioural model in ModelSim. This figure can vary depending on input.

[4] The reason for this is a very simple request-acknowledge protocol used for communicating with the memory manager.

in an overall speedup of 5.17.

Summing up, at a clock rate of 50 MHz, a run of a single decoder could be optimized to take 2.5 ms instead of 13.3 ms.

If the entire chip area of the Virtex2 FPGA could be used, and memory usage is optimized as described above, internal BlockRAM could be used for all data. Accessing data there only takes 1 clock cycle, yielding a speedup of 7.42 compared to the current version.

Of course, increasing to clock rate can also give a reduction in execution time. However, this is not advisable due to both increased power dissipation and the timing constraints given by the external memory.

Numerical improvement

As mentioned above, the implementation presented here is working with fixed point numbers. This is not optimal for very noisy signals. Floating point arithmetic—or log-likelihood arithmetic—should yield a significant improvment in such situations. [8] gives a detailed analysis of the latter.
The current implementation assumes 8 bit per probability. To accomodate for a larger range, one byte could be splitted into a 5 bit mantissa and 3 bit exponent. Then, each multiplication needs a parallel addition for the exponents and an additional step to recombine the result.

RHWOS

At the time of writing, the RHWOS infrastructure was not ready yet. Thus, the turbo decoder presented here is still a stand-alone version, that has to be adapted slightly to actually become suitable as a task.
Basically, the idea is to keep the actual decoder hardware on the R-FPGA, whereas the controlling Microblaze program has to be taken to the C-FPGA part of the system.

Most of these points also apply when porting the decoder to another platform utilizing Virtex2 FPGAs.

Communication Currently, the turbo decoder interfaces to the Microblaze via 3 GPIO registers (input, output and control) that have to be replaced by the standard task interface (STI).

The controlling program abstracts from these communications via the file `turbo.c`. On the VHDL side, `top.vhd` is the interface to the turbo decoder - its signals have to be routed to the STI.

Memory The turbo decoder accesses its data using the memory manager found in `memmanager.vhd`, which in turn abstracts from the actual memory (in our application, the right SRAM bank of the XFBOARD) using the file `basicmem.vhd`.
Addresses are generated by `memmanager.vhd`, and at the very beginning of the `architecture` part of the file, a base address can be given. As stated above in the memory section, a total of $127'488$ bytes are needed.

This memory manager has to be connected to the RHWOS' memory manager, which handles concurrent accesses by the tasks to the SRAM. The interim solution, `basicmem.vhd` assumes the signals seen in table 4-4 from the memory manager.

Signal	Meaning
`Addr`	20 bit address for SRAM
`DataIn`	16 bit data to be written
`DataOut`	16 bit data read (registered)
`TRQ`	transaction request
`ACK`	acknowledge signal
`WEn`	write enable (high active)

Table 4-4: Ports for memory manager

4.6 Running the demo

4.6.1 Software needed

- **Xilinx Project Navigator (ISE).** This is used to compile the hardware part of the system.
- **Xilinx Embedded Development Kit (EDK).** Used to program the Microblaze processor in C.
- **Microsoft Visual C.** Used for the encoder application on the PC side.

4.6.2 Compiling the demo

Please refer to chapter 6 on page 45 concerning the location of the project files on the CD.

PC application

The PC application used to send packets over Ethernet is Microsoft Visual C project. Only the actual sending, and the displaying of the picture are platform dependent.
The project file provided together with the source files contains all the dependencies, so executing the program from inside Visual C will do all the work needed.

XFBoard application

The XFBOARD application is split in both a hardware and a software part. Thus, compilation differs depending on where changes were applied.

Software Start EDK, and open the file `projnav\system.xmp`. Modifications to the software part of the application can be applied here, and the finished project downloaded to the FPGA using the option Download Bitstream.

Hardware This has to be done in two steps:

1. Start ISE, and open the file `projnav\system.npl`. This project connects the hardware portion of the decoder to the Microblaze architecture. The option "Generate programming file" will prepare a `.bit` stream for the Microblaze architecture
2. Start EDK, open the project file `system.xmp` and select the option "Import from ProjNav" to import the generated `.bit` stream, which is located in the directory `projnav\system.bit`.
   The .bmm file can be found in `implementation\system_bd.bmm`.

After importing, the software will now work with the updated hardware part of the system.

IP Addresses

IP addresses are hard coded in both applications. The default IP addresses used currently are $169.254.45.1$ for the PC, and $169.254.45.2$ for XFBOARD. As sending is only done in one direction, only the address of the XFBOARD can be changed.

- **PC.** Change the variable `m_IPAddressStr` in `TurboEncoderAppDlg.cpp`
- **XFBoard.** Change the variable `IPBoardAddress` in `network.c`

4.6.3 Running the demo

Start EDK, open the file `system.xmp`. Select the option "Download" from the "Tools" menu to download the bitstream to the XFBOARD.
The application will now poll the Ethernet and wait for the PC side to send an encoded block (in 13 packets). This block is then decoded and the progress of the turbo decoder shown on screen.

Chapter 5

Conclusion

This semester thesis concentrated on the R-FPGA part of the XFBOARD.
Both a device driver for the OS frame, as well as two signal processing tasks were implemented.

An overview over turbo codes is given, with references to papers concerning the subject. In the implementation section, issues in bringing the design to hardware are analyzed. Specifically, the translation to fixed-point arithmetics is described, along with a few solutions to prevent numerical underflows.

Then, an actual hardware version for the XFBOARD is presented. It's used in a testing environment that receives encoded, noisy data from a PC and decodes it. Finally, some results pertaining execution time and performance are shown.

While the turbo decoder implemented in the scope of this thesis is not useable in a real-world scenario, measures are suggested that could yield a significant performance boost. This should be sufficient for medium data rates.
Especially the aspect of using reconfigureable hardware for such a decoder seems tempting, and in comparison to proprietary ASIC solutions, a few promising aspects are hinted at.

Chapter 6

CD

This chapter gives a quick overview over the directory structure of the accompanying CD.

- **Presentations** Slides from the two presentations held
- **Report** Final version of this report, including all pictures
- **SpectrumAnalyzer** Files for spectrum analyzer task
- **TestProjects** Various test projects
  - **MemTest** Test for turbo decoder's memory manager (right SRAM bank)
- TurboDecoder: directory of turbo decoder project, including all PC software
  - Software
    - **InterleaverConstantGenerator** Generates constants for the hardware version of the interleaver
    - **TurboEncoderApp** Windows applications used in our testing environment to send an encoded, noisy picture to the XFBOARD.
    - **SNR_FixedPoint** SNR test for fixed point software implementation of turbo decoder. Use "make" to compile.
    - **SNR_FloatingPoint** SNR test for floating point software implementation of turbo decoder. Use "make" to compile.
  - **TurboDecoder** Turbo decoder hardware project.
- **VGADriver** Files for the VGA driver project
  - **VGADriver** Actual VGA driver - files need
  - **VGADemo** Demo as described in section 2.10 on page 12
  - **VGAMicroblaze** Demo for the Microblaze part of the VGA driver

Bibliography

[1] Alliance Semiconductor. *AS7C4096/ AS7C34096, 5V/3.3V 52K x 8 CMOS SRAM*, March 2002, v.1.8.
http://www.alsc.com/pdf/sram.pdf/fa/AS7C34096.pdf

[2] L. Bahl, J. Cocke, F. Jelinek and J. Raviv. *Optimal decoding of linear codes for minimizing symbol error rate*, IEEE trans. Inf. Theory, March 1974, pp. 284-287.

[3] C. Berrou, A. Glavieux., and P. Thitimajshima. *Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes*, IEEE Proc. ICC '93, Geneva, Switzerland, May 1993, pp. 1064-1070.

[4] ETSI. *Universal Mobile Telecommunications System (UMTS); Multiplexing and channel coding (FDD) (3GPP TS 25.212 version 3.4.0 Release 1999)*

[5] Intersil Corporation. *HI1178 Triple 8-bit, 40MSPS, RGB, 3-Channel D/A Converter*, 2000.
http://www.intersil.com/design/images/DataSheet.jpg

[6] H.-A. Loeliger. *Stochastische Modelle und Signalverarbeitung*, Lecture Notes, ETH Zurich, winter term 2003.

[7] M. Matsumoto and T. Nishimura. *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January 1998, pp. 3-30.

[8] G. Montorsi, and S. Benedetto. *Design of Fixed-Point Iterative Decoders for Concatenated Codes with Interleavers* IEEE Journal on selected areas in communications, vol. 19, no. 5, May 2001, pp. 871-882.

[9] S. Nobs. Student Thesis, Swiss Federal Institute of Technology Zurich (ETH), Computer Engineering and Networks Laboratory. *Prototype Board for Reconfigurable OS*, July 2003.
http://www.tik.ee.ethz.ch/∼walder/HomePage/SADA/
PrototypeBoardForReconfigurableOS/PBFROS.pdf

[10] S. Nobs. Diploma Thesis, Swiss Federal Institute of Technology Zurich (ETH), Computer Engineering and Networks Laboratory. *Reconfigurable Hardware OS - Prototype Part "C"*, 2004.

[11] B. Sklar. *A Primer on Turbo Code Concepts*, IEEE Communications Magazine, December 1997, pp. 94-102.

[12] S. Steinegger. Diploma Thesis, Swiss Federal Institute of Technology Zurich (ETH), Computer Engineering and Networks Laboratory. *Reconfigurable Hardware OS - Prototype Part "FPGA"*, 2004.

[13] *VGA timing information*, http://www.epanorama.net/documents/pc/vga_timing.html

[14] H. Walder, M. Platzner. Swiss Federal Institute of Technology Zurich (ETH), Computer Engineering and Networks Laboratory, TIK Report Nr. 168. *Reconfigurable Hardware OS Prototype*, April 2003.
http://www.tik.ee.ethz.ch/∼walder/XFORCES/TIKR168.pdf

[15] S. Wegmann. Diploma Thesis, Swiss Federal Institute of Technology Zurich (ETH), Computer Engineering and Networks Laboratory. *Video Playback Tasks for RHWOS*, 2004.

[16] Xilinx, Inc. *Virtex-II Platform FPGAs: Advance Product Specification*, 2002.
http://direct.xilinx.com/bvdocs/publications/ds031.pdf

[17] Xilinx, Inc. *Virtex-II Platform FPGA User Guide*, December 2002.
http://www.xilinx.com/publications/products/v2pro/ug_pdf/ug012.pdf

[18] Xilinx, Inc. *MicroBlaze RISC 32-Bit Soft Processor*, August 2001.
http://www.xilinx.com/ipcenter/catalog/logicore/docs/
microblaze_risc_32bit_proc_final.pdf

[19] Xilinx, Inc.
http://www.xilinx.com

[20] Xilinx, Inc.
http://www.xilinx.com/ipcenter/catalog/logicore/docs/xfft.pdf

Please note that the URLs noted below the reference entries have been valid at the time of writing. However, they may have become outdated in the meantime.

6th February 2004