A hardware implementation of an MP3 decoder

Irina Fältman, Marcus Hast, Andreas Lundgren, Suleyman Malki, Erik Montnemery, Anders Rångevall, Johannes Sandvall, Milan Stamenkovic

***Abstract*— In recent years, several commercial system-on-chip solutions for MP3 decoding have been developed. They are usually built around a specialised RISC processor with an instruction set suitable for MPEG audio decoding. The purpose of this project is to evaluate and implement the different parts of the MP3 decoding without using a processor core. Instead, small hardware accelerators are implemented for each stage in the decoding chain. Using specialised logic could lower the power consumption that is of most importance in handheld devices. The implementation is based on Thomas Lenarts C-code for MP3 decoding.**

I. Introduction

DURING the last years the usage of the MPEG-1 layer-III (mp3) audio codec has exploded, and a large part of the global bandwidth consumed is used for transferring layer-III compressed audio data, or in casual language ”mp3 files”. During the first years of widespread mp3 usage, software decoders were the most common, but during the last couple of years portable and other stand-alone players have gained in popularity. Particularly in hand held devices dedicated hardware for accelerating the mp3-decoding process, in terms of clock cycles and power usage, is important. This report describes an attempt to create an efficient dedicated mp3-decoder in hardware.

II. Functionality

The input to the MP3 decoder is a bitstream and the decoding process turns this into samples which are sent to a DAC[1] add-on board. The MP3 stream is divided into *frames* where each frame contains 27 ms of music data. The decoding process can be divided into six blocks and these are implemented independent from each other.

The first three, *huffman decoding*, *requantize* and *reordering* convert the original MP3 bitstream into 576 frequency lines divided into 32 subbands with 18 frequency lines in each. This is the format which the last three blocks *antialias*, *IMDCT* and *filterbank* use. These blocks transform the MP3 from the frequency domain into the time domain. A more in-depth description of the decoding process as well as mp3 encoding can be found in [1] and [2].

The design presented here has a couple of limitations. It does not do stereo decoding, so the output is a mono signal mixed into both channels. Furthermore it does not handle mixed blocks correctly. Both of these are limitations that can be rectified later.

[1]Digital to analog converter.

A. Huffman

The task of the huffman decoder is to transform incoming compressed data into scalefactors and symbols representing the 576 original frequency lines. The scalefactors are then used in the next block (the requantizer block) to rescale the symbols into non-scaled frequency lines. The information about how to create these symbols and scalefactors is found in the side information part of the MP3 frame.

The decoder compares the input sequence with information in the Huffman table and produces a symbol when a match is found. Information on what table to use for any given frame is found in the frames side information. Output from the Huffman decoder is 576 scaled frequency lines (symbols) which are divided into three partitions:

- *Big-values* contain the lowest frequency lines and are coded with the highest precision. Normally the scaled value is between -15 and 15, but higher precision can be obtained by using an escape sequence. When the decoder finds the value 15 it assumes that higher precision is needed and reads additional bits from the input stream. This value is then added to the original value of 15. The number of bits is specified in the Huffman table and are called *linbits*.
- *Count1* represents the higher frequency lines and does not need the high precision, they are simply coded with the values 1, 0 and -1.
- *Rzero* represents the highest frequency lines. They have simply been removed by the encoder. These values are filled with zeros by the decoder.

The boundaries of the partitions are specified in the side information.

B. Requantize

The Requantizer block rescales Huffman decoded scaled and quantized frequency lines. The result from the Requantizer is the original frequency lines. The complete descaling equations for both short (equation 1) and long (equation 2) blocks are presented below. What equations to use depends on the windowing function used in the encoding process. The Huffman decoded value at index *i* is *is(i)*, the output from the Requantizer block at index *i* is *xr(i)*.

$$\begin{aligned} xr_i &= sign(is_i)\cdot|is_i|^{\frac{4}{3}}\cdot 2^{\frac{1}{4}(global\_gain[gr]-210)}\cdot \\ &\quad \cdot 2^{-2\cdot subblock\_gain[window][gr]}\cdot \\ &\quad \cdot 2^{-scalefac\_multiplier\cdot scalefac\_s[gr][ch][sfb][window]} \end{aligned} \quad (1)$$

$$\begin{aligned} xr_i &= sign(is_i)\cdot|is_i|^{\frac{4}{3}}\cdot 2^{\frac{1}{4}(global\_gain[gr]-210)}\cdot \\ &\quad \cdot 2^{-scalefac\_multiplier\cdot scalefac\_l[sfb][ch][gr]}\cdot \\ &\quad \cdot 2^{-preflag[gr]\cdot pretab[sfb]} \end{aligned} \quad (2)$$

Scalefactors *scalefac_s* and *scalefac_l* used by Requantizer are provided by the Huffman decoder. The frequency lines for long and short blocks are divided into 23 and 13 *scalefactor bands* for each window respectively, each scalefactor band having it's own scalefactor.

The other parameters such as *global_gain*, *subblock_gain* and *preflag* can be found in the frame information provided by the Synchronizer block.

The requantization step must be performed once for each frequency line in the bitstream.

C. Reordering

The reorder block has one task; it reorders the frequency lines within a granule. The way that the frequency lines are reordered depends on flags in the side information header.

The block works in three different ways depending on the side information header. (Only support for MP3 streams with 44.1 kHz sample frequency is implemented.)

- All frequenct lines are reordered.
- Only frequency lines after after line number 36 are reordered.
- No frequency lines are reordered.

D. Antialias

The antialias block attempts to reduce the inevitable alias effects introduced by the use of a non-ideal bandpass filter. This reduction is done by merging frequencies using butterfly calculations as seen in figure 1.

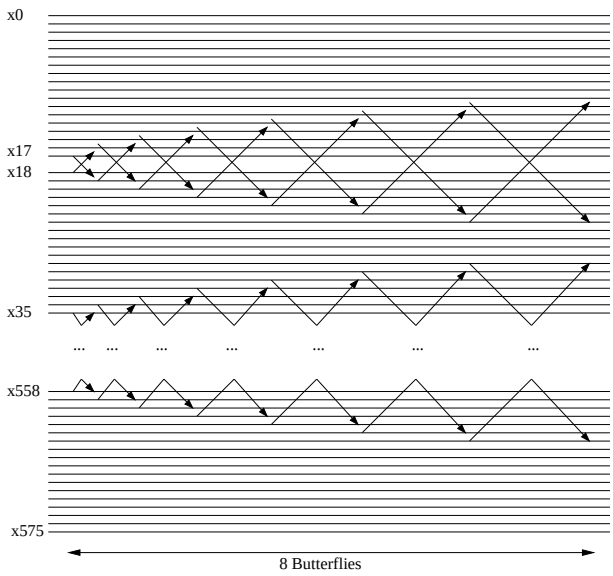

Fig. 1. Antialias butterflies

E. IMDCT

Inverse Modified Discrete Cosine Transform, IMDCT, reproduces, in cooperation with the synthesis polyphase filterbank, time samples from frequency lines. Given the frequency lines X_k, time samples x_i can be obtained by using the following equation:

$$x_i = \sum_{k=0}^{n/2-1} X_k \cos(\frac{\pi}{2n}(2i+1+n/2)(2k+1)), \quad 0 \le i < n \quad (3)$$

In our case $n = 36$, which means that the IMDCT takes as input 18 frequency lines and generates 36 polyphase filter sub-band samples. These samples are multiplied with a 36-point window before they can be passed on to the next step in the decoding process. Windowing contains four different types of windows, the types are normal, short, start and stop. Information on what type to use is found in the side information part of each frame. Depending on window type two different implementations are used.

Producing 36 samples from 18 frequency lines means that only 18 of the samples are unique. Therefore IMDCT is said to use a 50% overlap. The 36 values from the windowing operation are divided into two groups, a low group and a high group, containing 18 values each. Overlapping is carried out by interleaving (adding) values from the lower group with corresponding values from the higher group from the previous frame.

General view of the "operation flow" is shown in figure 2.

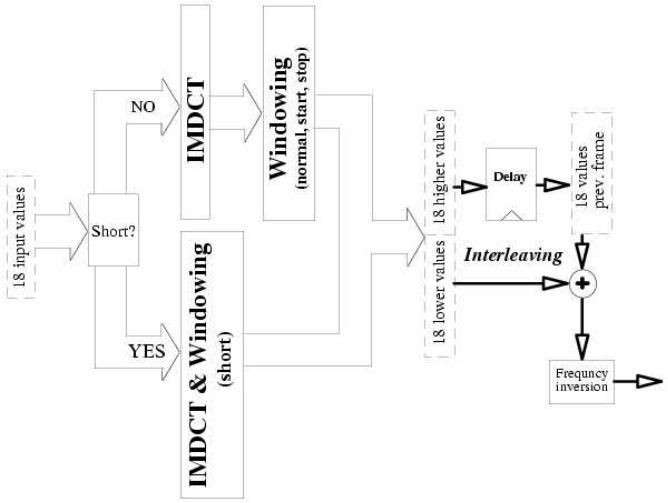

Fig. 2. IMDCT operation flow

The entire cosine term for each output can be treated as a known constant factor based on the combination of i and k, in equation 3. Some investigation of these terms shows a symmetry between the different x_i. Only half of the values are uniquely determined. The rest can be obtained as a function of the previously calculated terms. The following symmetry holds:

$$x_i = -x_{n/2-i-1}, \; i = 0 \ldots n/4-1 \quad (4)$$

and

$$x_i = x_{3n/2-i-1}, \; i = n/2 \ldots 3n/4-1 \quad (5)$$

Therefore calculating the first quarter and the third quarter of all values will be enough to determine the entire set.

Another optimization was achieved by merging calculations of IMDCT-values with the subsequent windowing into one single multiplication step, when short windowing was used. It was also noticed that a number of calculations could be omitted because several multiplications contained the factor zero.

F. Filterbank

The synthesis polyphase filterbank is the final step in the decoding process. It exploits aliasing and windowing to move the subbands back into their frequency domain origins. The process is naturally divided in two parts, an MDCT part for translating the aliased subband signals, and a windowing part to filter out the undesired aliasing in the translated signal.

1) Modified discrete cosine transform: The sub-samples from the transpose block are ordered is such a way that the 32 first values are the first sub-sample from each subband, the next 32 are the second sub-sample and so forth. The MDCT processes 32 values at a time using the following equations:

$$Y_i = \sum_{k=0}^{31} N_{ik} * S_k \tag{6}$$

$$N_{ik} = cos(\frac{\pi}{2 * 32}(16 + i)(2 * k + 1)) \tag{7}$$

The output values Y_i are stored in a barrel shifter.

2) Windowing: Multiplying the values from the barrel shifter with the window function, specified in the ISO standard. 32 PCM samples are computed each iteration. The MDCT and windowing are performed 18 times for each granule, resulting in 576 PCM samples, i.e. 27 ms at 44.1 KHz.

III. Architecture

The target hardware of the project is a Virtex-II 1000 FPGA manufactured by Xilinx. The FPGA features 5120 logic slices and 40 pairs of one 18x18-bit multiplier and one Block Select RAM macro[2]. Though hardware resources were plenty it was decided early during the project that hardware use had to be kept on a realistic level.

By dividing the decoding process into blocks which run sequentially it is easy to share common resources. In this design the main memory containing the frequency data and the 32-bit multipliers are shared.

The memory is put into two Block Select RAMs and it is the *huffman decoder* which fills it with data. The control to read and write to this block is then given to the currently active block, so the blocks which are inactive are not able to do anything with the main memory.

Furthermore many of the blocks use multiplications in the decoding process. These multiplications are 32x32-bit and are made by joining a total of 4 18x18-bit multipliers into one. Since there is a limited number of multiplier and BSR pairs it was decided to also share the multiplier between the blocks in the same way as the main memory.

[2]There are limitations on how one such pair is used. Some of the wires are shared. It is not possible to use both the multiplier and BSR at 18-bit length at the same time.

A. Huffman

The Huffman decoder uses a number of tables in the decoding process. There are a total of 4 tables with constants and 32 additional Huffman tables. The first decision to be made was which of these to leave hard coded into the chip as constant values and which to put in Block Select RAM.

As it turns out the four tables with constants are all quite small. The largest of them is 690 bits and can thus be placed on the chip as constants without taking a lot of space. This also cuts down on local accesses to memory which makes the block run smoother. The Huffman tables are huge in comparison (the largest is 8176 bits), and there are 32 tables to store. Thus it was decided that the Huffman tables should be placed in a Block Select RAM.

The decoder works by reading a variable number of bits from the input stream and decoding these as a block. The number of bits to be read in every step can vary anywhere from 0 to 13 bits, but access to memory is only possible in 8 bit words. This means that if 13 bits are to be read two or three reads from the input stream would be needed causing the decoder to stall. To reduce the number of reads one byte is always prefetched from the input stream so that there are between 9 and 16 bits available from the stream at any given time (depending on how many bits were read in the previous step). If 13 bits are to be read it can often be done in one read (two reads in the worst-case scenario).

The total number of Huffman tables according to the ISO-specification is 32, but the actual number of tables stored is 17. This is due to the fact that tables number 16 to 23 are the same and so are tables number 24 to 31. Also, one table has been omitted since its only task is to invert bits from the input stream. Instead this special case is detected and treated separately.

The representation of the Huffman tables is not specified in the ISO-specification, but it can be viewed as a tree-like lookup table that is processed from the root. The input sequence tells the decoder which branch to take and bits are continually read from the input stream until there are no more branches to take.

The interested reader might want to know what the states of the decoder look like. For this reason figure 3 has been included.

B. Requantize

The Requatizer has been partitioned into a number of blocks for parallelizing the process as can be seen in the figure 4.

The *requantize* block is the highest stage in the hierarchy interfacing with the Controller, the main memory and the Huffman decoder and retrieves the necessary data from the frame header and side information. The state machine for the *requantize* can be seen in the figure 5.

The first step of the requantization process is to calculate $|is|^{\frac{4}{3}}$. The calculation of numbers raised to the power of $\frac{4}{3}$ is computationally expensive to implement. One way to improve

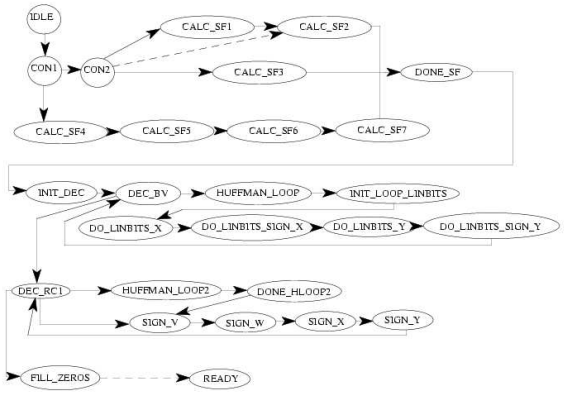

Fig. 3. Figure showing all states in the decoder. The state CALC_SF3 only differs from CALC_SF2 in that it is initialized differently, so there is actually no CALC_SF3 state in the decoder. It is still shown in the figure so that there would be no confusion as to why there is no CALC_SF3. In most states a number of bits have to be read. This is done in the GET_BITS state, but since the figure would become overly cluttered if this state was included it has been omitted.

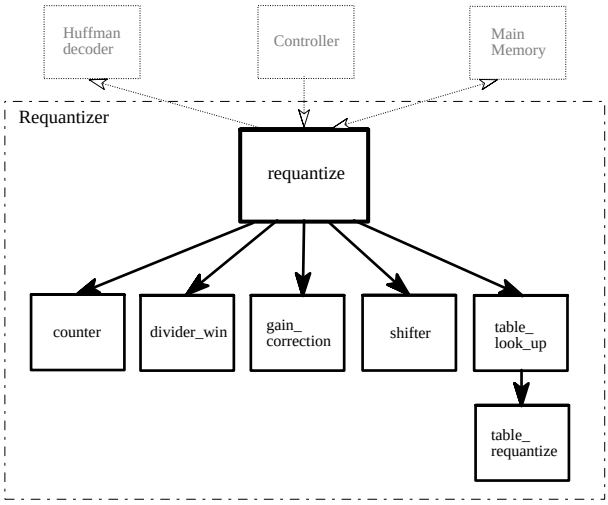

Fig. 4. The architecture of the Requantizer. Communication between the blocks within the Requantizer is shown with black lines. Communication with the external blocks is shown with dashed lines.

the performance is to use a look-up table containing all the 8192 possible input values. A look-up table is fast but requires approximately 256 kbits of memory space. The computation has been divided in two cases, so the size of the look-up table has been reduced to 32 kbits.

Case 1: If *is(i)* is less than 1024, the result can directly be found in the look-up table;

Case 2: If *is(i)* is greater than or equal to 1024, the value is first divided by 8. The result from the look-up table is then multiplied by 16. This is possible because of the relation in the equation 8.

$$|is_i|^{\frac{4}{3}} = 16 \cdot |\frac{is_i}{8}|^{\frac{4}{3}} \quad (8)$$

The block performing the look-up is called *table_look_up*. The table is stored in Block Select RAM and is included as a part of the initialized data section.

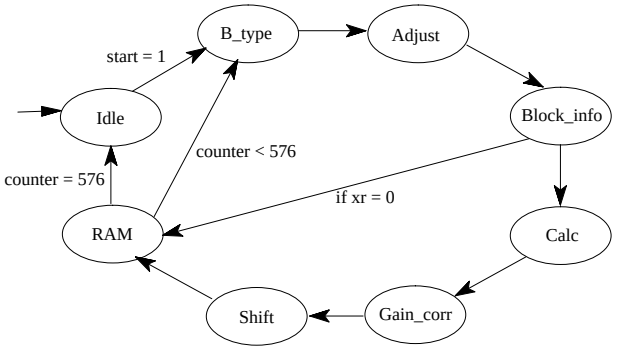

Fig. 5. State machine for the *requantize*

A separate frequency line *counter* block has been created to provide the *requantize* block with the information about what frequency line that is being requantized at the time. The total amount of frequency lines per frame is 576. When the *counter* has become 576 the *ready* signal for the Requantizer is generated.

The *divider_win* block has been created for a simple and fast *window* calculation. There can be totally three windows in a short block.

The *shifter* is used for multiplying $(sign(is_i)|is_i|^{\frac{4}{3}})$ with $2^{\frac{1}{4}\cdot C}$. The equations for factor *C* can be seen below. Equation 9 is used for calculation of short blocks and equation 10 for calculation of long blocks. The factor *C* is pre-calculated in the *requantize* block.

$$C = global\_gain[gr] - 210 - 8 \cdot subblock\_gain[window][gr] \quad (9)$$

$$C = global\_gain[gr] - 210 - scalefac\_multiplier \times \times scalefac\_l[sfb][ch][gr] - preflag[gr] \cdot pretab[sfb]) \quad (10)$$

The *gain_correction* is another table used for storing the correction factor. The table is stored in Distributed RAM and is included as a part of the initialized data section.

A shared multiplier is used in the requatization calculations for multiplying *xr(i)* with the correction factor.

C. Reordering

The reorder block is built around two memories. One memory contains the temporary storage for sample data and the second memory contains the addresses for the main memory and the temporary storage memory. The order of these addresses describes the functionality of the reorder block.

D. Antialias

The implementation of the antialias block is a statemachine containing one butterfly calculation and some counters. Data for one butterfly is read from memory, four multiplications using the shared multiplier are carried out and then finally one addition and one subtraction are made. The results from these are stored back into the main memory.

The eight pairs of alias constants are not placed in a Block Select RAM since that wasted the BSR needlessly. Instead the constants are placed in ROM in LUTs. In order to minimize

register use partial results are stored in two different registers. Doing this instead of putting each of the different partial calculations in separate registers lowered the size of the unit to less than half of the first ”naive” implementation.

1) Butterfly calculations: The first observation to make is that the main limit on the antialias block is the multiplier. Besides the multiplications the main delay is accessing memory. To minimize this latency as much as possible multiplications and memory access are made in parallel as much as possible (but no ”prefetch” of the next butterfly values is made while the first one is still carried out). A diagram of one butterfly can be seen in figure 6.

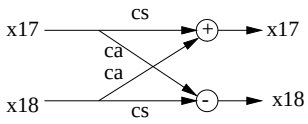

Fig. 6. One butterfly merge

E. IMDCT

For all multiplications the common multiplier is used. To obtain the summations of each x_i (see formula 3) an accumlator is used. All cosine-values, used for IMDCT-computations, and sine-values, used for windowing, are stored in Block Select RAMs. Figure 7 show how most components are connected.

The following table shows the number of clock cycles needed to compute all time samples in a frame. The time required, when using a clock frequency of 24 MHz, is shown too.

IMDCT type	Clock Cycles	time/μs
Short Windowing	15,615	651
“Normal” Windowing	15,196	633

F. Filterbank

During the implementation of the filterbank it has been a strict design goal to keep the design simple, yet reasonably efficient. With this philosophy in mind the implementation of the DCT part was done very straight forward, using matrix multiplications instead of a butterfly scheme such as Lee’s algorithm [3]. One might object to this method since a large number of multiplications are unnecessary and time is thus wasted. As will later be shown, time was not critical. Another benefit from this approach is that the amount of logic used in the DCT is small.

Two Block Select ROMs were instantiated to keep DCT and windowing constants, as well as two Block Select RAMs used as a dual port shift register for passing data between the DCT and windowing blocks.

G. I2S and communication with the real world

Sending data to the real world requires a properly formatted continuous stream of samples. It was clear from the beginning that the output data should be sent according to the I2S protocol, originally developed by Philips, but now a de-facto standard supported by virtually every DAC, SPDIF-converter or any other component that processes real time digital audio. The I2S standard dictates that data is sent over a synchronous serial bus, in two complement and MSB first. The bus is a three wire bus consisting of a serial data line, a word select line and a clock line.

Since the mp3 decoder works with frames and granules it is not able to produce a continuous flow of data, but rather produces chunks of 576 samples at a time. A continuous data flow is an absolute requirement however, and a FIFO-buffer of some sort is a natural choice. A block select RAM was instantiated, and is used as a 1024 sample buffer.

The mp3 standard supports a wide number of sample rates, but output is currently fixed at 44.1kHz. Furthermore, the 24MHz system clock is not well suited for audio applications as it can not be divided to form any of the standard sample rates. Dividing the clock by 544 produces a 44.118kHz word clock, which means that the audio is played somewhat too fast, $44118/44100 = 1.0004$, i.e. 0.04% too fast.

Adding functionality to support other sample rates would not be taxing to implement, and is actually prepared for, but since the majority of all mp3s are encoded in 44.1kHz it was decided that supporting this sample rate would be enough.

Furthermore, to be able to test our implementation of the mp3-decoding standard, a very simple, yet perfectly working, mp3 playing system was created. The structure is implemented in the entity ’mp3player’ which instantiates a decoder as well as instances of entitities to initialize the DAC and transfer data from an external flash ROM containing mp3 encoded music, to the decoder. The system lacks any ability to interact with a user, and will continuously loop the music stored in the external memory.

Since the development board used during the project does not feature a DAC, but does feature expansion possibilities, a PCB containing a DAC as well as a flash ROM was designed. The DAC used is a Texas Instruments TLV320AIC23. The TLV320AIC23 actually contains an ADC as well as the oversampling sigma-delta D/A converter used in this project.

IV. Synthesis

During the synthesis different tools were used, Synopsis FPGA compiler, Xilinx XST and Synplicities Synplify Pro. Due to problems with the software only Synplify Pro was able to successfully synthesize the project.

A. Huffman

Logic	Logic Blocks	Utilization
Slices	576	11%
Flip Flops	181	1%
4-input LUTs	1046	10%
Block RAMs	3	7%

B. Requantize

Logic	Logic Blocks	Utilization
Slices	634	12%
Flip Flops	166	1%
4-input LUTs	1106	10%
Block RAMs	2	5%

C. Reordering

Logic	Logic Blocks	Utilization
Slices	36	1%
Flip Flops	16	1%
4-input LUTs	51	1%
Block RAMs	5	12%

D. Antialias

Logic	Logic Blocks	Utilization
Slices	165	3%
Flip Flops	102	1%
4-input LUTs	287	2%

E. IMDCT

Logic	Logic Blocks	Utilization
Slices	400	7%
Flip Flops	155	1%
4-input LUTs	654	6%
Block RAMs	3	7%

F. Transpose

The order that the Filterbank and I2S blocks read data from main memory is different than that of preceding blocks. The purpose of the transpose block is to reorder main memory data into a format that suites the Filterbank and I2S.

The implementation of the block is really straight forward. It reads data from the main memory and writes it in the new order into Block Select RAM. The data is then written back to the main memory again.

The transpose block could be eliminated altogether if the filterbank and I2S blocks where to read data in the same order that the IMDCT block writes it.

Logic	Logic Blocks	Utilization
Slices	68	1%
Flip Flops	38	1%
4-input LUTs	111	1%
Block RAMs	2	6%

G. Filterbank

Logic	Logic Blocks	Utilization
Slices	319	6%
Flip Flops	149	1%
4-input LUTs	596	5%
Block RAMs	4	10%

H. I2S

Logic	Logic Blocks	Utilization
Slices	83	1%
Flip Flops	60	1%
4-input LUTs	123	1%
Block RAMs	1	2%

I. MP3 project

Logic	Logic Blocks	Utilization
Slices	2881	56%
Flip Flops	1587	15%
4-input LUTs	4614	45%
Block RAMs	23	57%
18x18 Multipliers	4	10%

As has been mentioned previously, when using adjacent multiplier and Block Select RAM, limits are put on the address width used, in order to keep it routable on a Virtex II. This reduces the available number of multiplier / Block Select RAM blocks to a total of 40, compared to 40 for each type. This means that 27 out of 40 primitve blocks are used.

The calculations of the timing for the blocks were done by measuring the time with an oscilloscope on the hardware, running at 24 MHz.

Block	Time
sync	140 μs
huffman	120 μs
requantize	140 μs
reorder	$<$10 μs
antialias	83 μs
imdct	630 μs
transpose	48 μs
filterbank	1.16 ms
All	2.3 ms

The Virtex-II development board supplies a 24 MHz clock. A comparison between the timing of the design and the requirements, 44.1 KHz $\rightarrow$ 27 ms, shows that it is capable of running at a significant lower clock speed than 24 MHz.

V. Conclusions

A. Huffman

The design is satisfactory, although some improvements are possible. The main drawback of the decoder are the scalefactors, and the way they are communicated from the decoder to the requantizer block. By using ”Gaisler’s” method of coding made it (too) easy to put the scalefactors in the feedback flip-flops, however this proved quite inefficient since this lead to excessive routing. The implementation actually leads to 248 parallel connections between the two blocks. This routing could be substantially decreased simply by storing the scalefactors in Block Select RAM. Also othertables besides the Huffman tables could be put in Block Select RAM to further decrease the size of the design. But as always there is a tradeoff between time and area. The design now is fast but large, whereas decreasing the size would make it slower (since there would be more memory reads).

B. Requantize

The design of the requantizer block can be further improved. In this version of the Requantizer rather heavy computations

are performed for the function $2^{\frac{1}{4}\cdot C}$. The factor C is precalculated in the *requantize* and then 2 is raised to the power of $\frac{1}{4} \cdot C$ by shifting. The function $2^{\frac{1}{4}\cdot C}$ does not take on more than 384 different values. Both calculations and shifter are area consuming, that's why a small look-up table would probably be the best choise and would improve both the timing and the block area. The table can be made even smaller (196 values) by rounding small values down to zero.

C. Reordering

Since this block is already very fast there is little point in spending resources on making it faster. Neither does it use a lot of logic on the chip. The only part which is big is the number of Block Select RAMs it utilizes, but this is a trade off to keep logic use down.

D. Antialias

The largest optimizations made in the antialias block was to put constants in a LUT based ROM and to store as few temporary values from multiplications as possible. One further improvement could be to fetch the next iterations values while doing the multiplications for the current iteration[3]. The benefits of doing this to the entire chip would be marginal however. Since the antialias block currently is both quite small and quite fast it would be better to concentrate any efforts on other blocks.

E. IMDCT

Already at the beginning of the project, critical decisions concerning what algorithms of IMDCT to use, had been made. Many fast algorithms were found, but choice was done to use the most straight forward one, due its low complexity compared to another more sophisticated algorithms. Loss of efficiency could be accepted to as quickly as possible achieve working architecture in the first hand, especially that loss of efficiency was not critical for the entire design. The result showed to be satisfactory.

It's worth mentioning that best (fastest) algorithm [4], and at the same time most complex to implement, that could be found, is carried out by Vladimir Britanak and K. R. Rao. This algorithm is based on the DCT/DST, of types II and III, and a butterfly pattern for calculating both IMDCT- and windowing-values.

[3]This is basically "software pipelining" of the antialias loop.

F. Filterbank

The implementation of the filterbank satisfies the design goals, and does not use excessive amounts of hardware. If need be, the DCT calculation could easily be modified to exploit the redundancy in the DCT algorithm with a butterfly scheme such as Lee's algorithm [3]. Such a modification would lower the number of clock cycles needed to perform the entire filterbank calculation by a factor ten.

ACKNOWLEDGMENT

The authors would like to thank our instructors, Hugo Hedberg, Fredrik Kristensen and Thomas Lenart for all help during the project. We would also like to thank Martin Nilsson for all his help and support with developing the DAC daugherboard.

REFERENCES

[1] *ISO/IEC 11172-3:1993 Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*, 1993.

[2] S. Gadd and T. Lenart, "A hardware accelerated mp3 decoder with bluetooth streaming capabilities," Master's thesis, Lund Institute of Technology, Sweden, 2001.

[3] B. G. Lee, "A new algoritm to compute the discrete cosine transform," IEEE transactions on acoustics, speech and signal processing, vol ASSP-32, No 6, December 1984.

[4] V. Britanak and K. R. Rao, "A new fast algorithm for the unifi ed forward and inverse mdct/mdst computaion," Signal Processing, 2002.

APPENDIX I
PICTURES

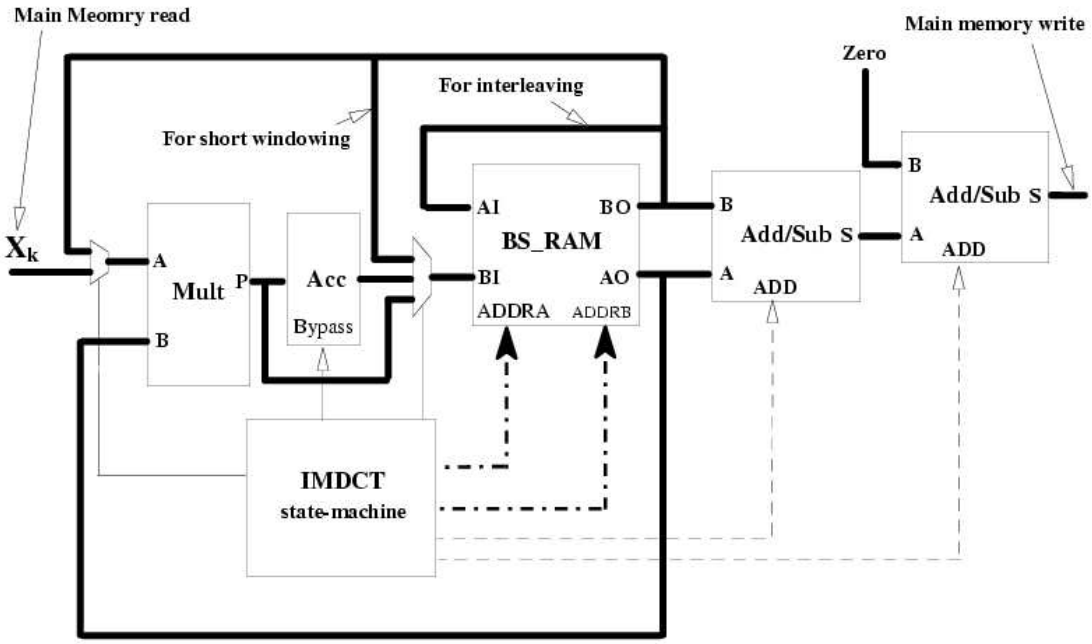

Fig. 7. IMDCT's block diagram

Fig. 8. Xilinx devel board, DAC board