FPGA-BASED ARCHITECTURE OF MP3 DECODING CORE FOR MULTIMEDIA SYSTEMS

Thuong Le-Tien, Vu Cao-Tuan, Chien Hoang-Dinh
Hochiminh City University of Technology
268 Ly Thuong Kiet, Dist 10, Hochiminh City, Vietnam
NT: +84-8-8654600; Fax: +84-8-8645922
Emails: ThuongLe@hcmut.edu.vn, CaoTuanVudh@yahoo.com

ABSTRACT

A multimedia system links images and audio to distribute and transmit information to users. These images and audio data have to be compressed for raising capacity of storing and processing tasks in real time, and allowing the content of signals to be suitable for the band-width of processing systems. For each terminal, softwares are the most common tools to read the compressed data, but in recent years portable devices such as mobile phones, mp3 players, etc… have gained in popularity, therefore designing of decompressing cores used in the handheld devices are the necessary demand for professional ASIC designers.

This paper introduces the proposed architecture of an mp3 decoding core. The core are partitioned into subcores named *Huffman*, *Requantizer*, *Reorder*, *Antialias*, *IMDCT*, *Filterbank*, *Interface* and *Controller*, in which each subcore can individually be designed, coded, and tested easily. The core is firstly described as a functional specification in VHDL, and is synthesized, compiled and simulated on Xilinx ISE 7.1i software. Finally, it is implemented on the hardware of FPGA Virtex-II Pro-board of Xilinx Company. The results are met the standard requirement from both simulations and hardware implementation.

Keywords: MP3, Huffman decoding, Filterbank, VHDL, Multimedia, Xilinx ISE 7.1i.

1. OVERVIEW OF MP3 DECODING

MP3 *(MPEG 1 layer 3)* is a standard [4] for compressing digital audio was being developed from 1988 to 1992 by MPEG (*Moving Picture Experts Group*)- this standard makes use of "Sub-band Synthesis" for transforming audio signal in Multimedia systems. The decoding process is shown in figure 1.

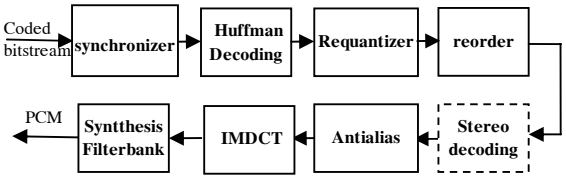


Fig 1. The mp3 decoding process [4]

Each block of the process can be funtionalized as following in details.

1.1 Synchronizer

Before decoding, the start of the frame must be found. If the frame is interrupted, we can not find the exact position of the next frame. The structure of a frame is shown in figure 2. The header contains enough information to calculate the size of the frame, which is also the start of the next frame, using the bit rate and frequency fields.

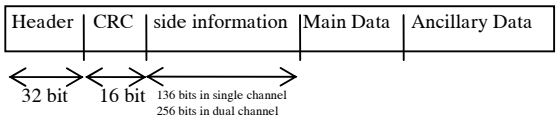


Fig 2. The mp3 bitstreams format [4]

1.2 Huffman decoding

The decoding procedure is based on several Huffman tables (32 Huffman tables for MP3) that are used for mapping Huffman codes to symbols. The mapping of symbols to Huffman codes is based on the statistic contents of the input sequence. Symbols that occur more frequently are coded with a short code, while symbols that occur less frequently are coded with longer codes.

The output of the Huffman decoder is 576 scaled frequency lines for each granule; they are partitioned into three parts as illustrated in figure 3: *Big-values, count1* and *rzero*.

Big-values represent the lowest frequency lines and are coded with the highest fidelity with the value ranging from -15 to 15. To decode the higher values, some code tables use the value 15 as an *escape code* and read additional bits from the input stream. This value is then added to the *escape code*. Numbers of bits is specified in the Huffman table and are called *linbits*.

Count1 represents the higher frequency lines and contains small values ranging from -1 to 1,

R*zero* represents the highest frequencies and is not a part of the bitstream. Instead, *rzero* represents the frequency lines that have been removed by the encoder, and it should be filled with zeros by the decoder.

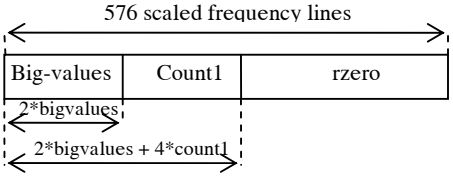


Fig 3. The 3 parts of frequency lines [3]

The boundaries of the parts are specified in the side information, and are selected by the psycho acoustic model during the encoding process.

When decoding *big-values* region, the Huffman tables create two frequency lines and four frequency lines for decoding *count1*.

1.3 Requantizer

The symbols from the Huffman decoder can be reconstructed into the original frequency lines by using the scalefactors in the *side information* of the frame. The frequency lines are divided into 21 groups, called *scalefactor bands*, each using its own scalefactor. A low frequency scalefactor band contains fewer values than a scalefactor band representing higher frequencies.

The complete descaling equations for both short and long blocks are shown in eq. (1) and (2). What equations to use depend on the windowing function used in the encoder. The Huffman decoded value at index *i* is *is(i)*, the output from the Requantizer block at index *i* is *xr(i)*.

- For short blocks:

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

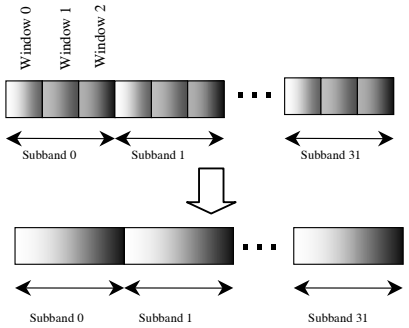
- For long blocks:

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

with *scalefac_s* and *scalefac_l* are the scalefactors that supplied by Huffman decoder. *Global_gain*, *subblock_gain* and *preflag* are parameters that found in the information of the frames.

1.4 Reorder

During encoding process, the MDCT can arrange the output values in two different ways depending on the *side information header* (Only support for MP3 streams with 44.1 kHz sample frequency is implemented.). Normally the output from the MDCT is sorted by subbands in increasing frequency. When a short block is decoded, a short window will be used. The output will in this case be sorted on subbands, then on windows and then on increasing frequency. In order to increase the efficiency of the Huffman coding the frequency lines for the short windows case were reordered into subbands first, then frequency and at last by window. An example for reordering process is shown in figure 4



(the darker color represents the higher frequency)

Fig 4. The reordering process [10]

1.5 Antialias

It is the attempts for reducing the inevitable alias effects because of using a non-ideal bandpass filtering in the subband synthesis block of *encoder*. The alias reconstruction calculation consists of eight butterfly calculations for each subband, as illustrated in Figure 5. The constants in the figure are in the specified standard [3].

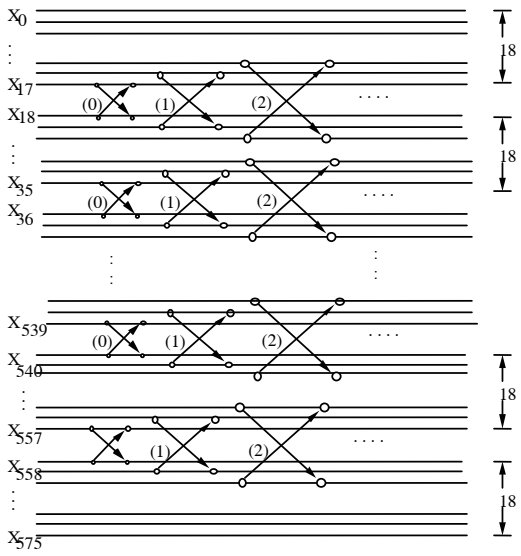


Fig 5. Alias reduction butterflies [3]

1.6 Inverse Modified Discrete Cosine Transform (IMDCT)

The IMDC will output 18 time domain samples for each of the 32 subbands:

$$xi = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left[\frac{\pi}{2n}\left[2i+1+\frac{n}{2}\right](2k+1)\right] \quad (3)$$

$$0 \le i \le n-1$$

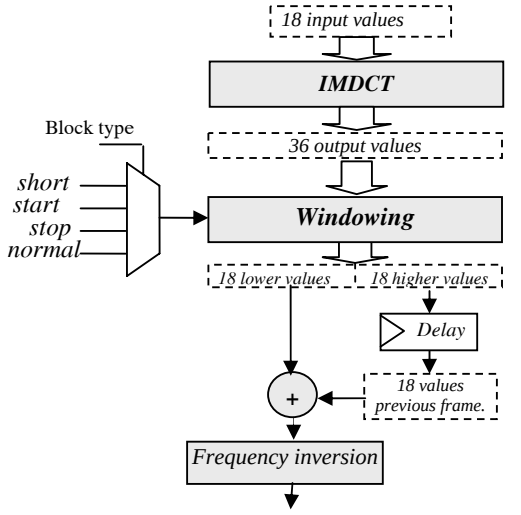


Fig 6. The IMDCT operation flow [10]

The IMDCT operation flow is shown in figure 6. The IMDCT block is an *18-point DCT* that produces 36 output values from 18 input values. These samples are multiplied with a *36-point window* before it can be used by the next step in the decoding process. The window to use is based on the *block type* (there are four block types that used, they are: *short, start, stop, normal*), that can be found in the *side information*. Producing 36 samples from 18 frequency lines means that only 18 of the samples are unique. So, the IMDCT use a 50% overlap. The 36 values from the *windowing* are divided into two groups, a low group and a high group, containing 18 values each. Overlapping is performed by adding values from the lower group with corresponding values from the higher group from the previous frame.

1.7 Synthesis polyphase filterbank

Synthesis polyphase filterbank (subband synthesis) is the last step of the decoding process. It converts 32 subbands to produce 32 PCM

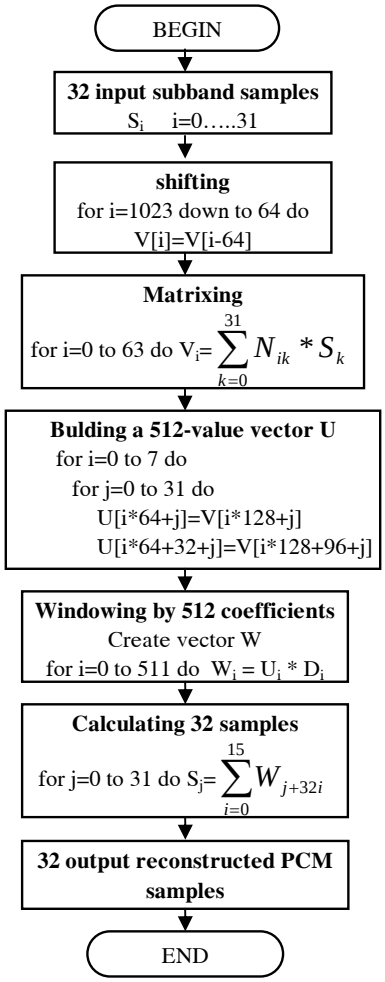


Fig 7. Synthesis subband filter flow chart

samples at a time using the input samples from the *filter bank* (figure 7).
In the synthesis process, the 32 subband values are transformed to the 64- value V vector using *matrixing*. The V vector is pushed into a FIFO which stores the last 16 V vectors. A U vector is created from the 32 component blocks in the FIFO and a window function D is applied to U to produce the W vector. The reconstructed PCM samples are obtained from the W vector by decomposing it into 16 vectors each 32 values in size and summing these vectors.

2. OVERVIEW OF CORE DESIGN PROCESS

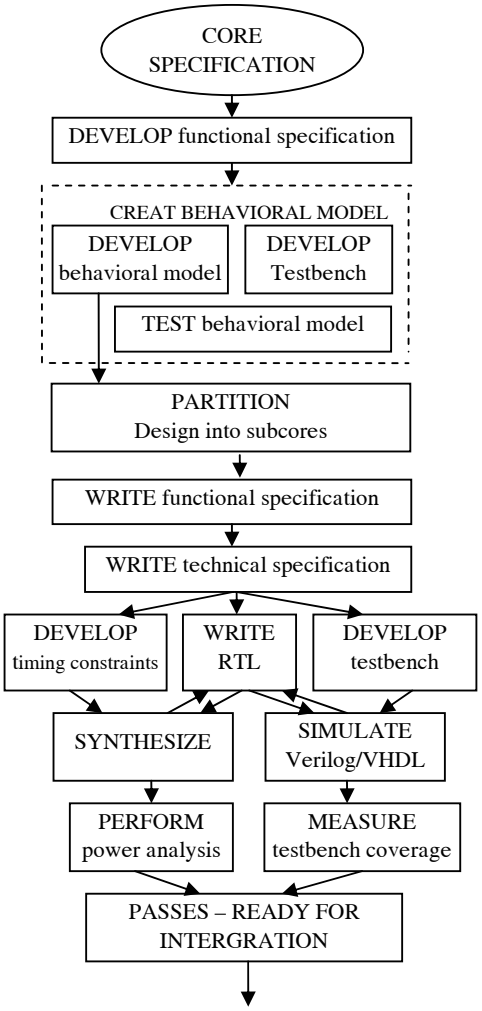


Fig 8. The Core design process [2]

Core is a predesigned, preverified silicon circuit block, usually containing at least 5.000 gates, which can be used in building a larger or more complex application on a semiconductor chip. In SoC *(system on a chip)* design, the use of cores (*design-reuse)* is very important because of the incessantly increasing of the system's complexity. Figure 8 shows a standard core design process [2], including the major steps as follows,

- **Specification and partitioning –** The first important thing is the initial core specification must be understood completely. Then the design is partitioned into subcores.
- **Sub core specification and design –** Once the partitioning is complete, the designer develops a functional specification for the subcore, emphasizing the timing and functionality of the interfaces to other subcores.
- **Testbench development** – refining the behavioral testbench into a testbench that can be used for RTL (register transfer level) testing of the entire core.
- **Timing checks –** checking the timing budgets of sub cores to ensure that they are consistence and achievable.
- **Integration –** Integrating the subcores into the core includes generating the top-level netlist and using it to perform functional test and synthesis.

3. ARCHITECTURE OF THE CORE

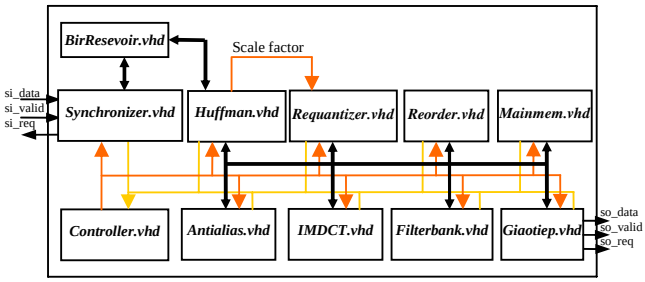


Fig 9. *Mp3 decoding Core*

The proposed CORE is described as a functional specification in VHDL - the most popular *hardware description language*. After that, this core is implemented on the FPGA hardware of *Virtex-II Pro XC2VP30-FF896* of *Xilinx Company*. The FPGA features 30816 logic slices and 136 pairs of one 18x18-bit multiplier and 136 BSRs (Block Select RAM) and up to 644 IO (input/output) signals.

The flow chart of the core design process was shown in figure 8 and the specification of the core is also discussed in part II. The diagram block of mp3 decoding core is illustrated in figure 9. The core is divided into the following subcores:

3.1 Synchronizer Subcore

The Synchronizer subcore ***(synchronizer.vhd)*** reads input data and implements the synchronization. The result of synchronizing is stored in a memory ***(bitreservoir.vhd)***.

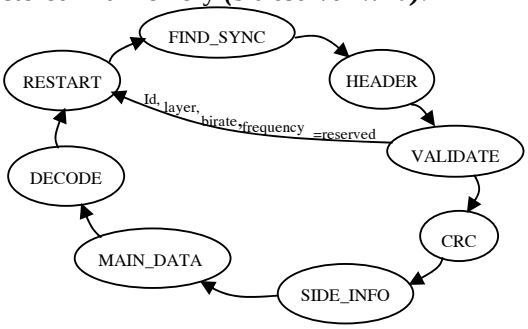


Fig. 10 The state machine of Synchronizer subcore

This subcore was designed on state machine that is shown in figure 10 with the following states:
FIND_SYNC: the state that finds the synchronous word including a bit pattern of 12 consecutive ones.
HEADER: the state that specifies the start of a frame.
VALIDATE: the state that determines the validation of an mp3 frame. If *id, layer, bitrate, frequency* is reserved then the found bitstream is not mp3 frame and the next state is RESTART.
SIDE_INFO: the state that confirms the side information of frame
MAIN_DATA: the state that confirms the main data of a frame.
DECODE: the state that confirms the completion of the frame definition.
RESTART: the state that restarts a new frame.

3.2 Huffman decoding Subcore

The main task of H*uffman decoding* subcore ***(huffman.vhd)*** is to transform compressed data into scalefactors and symbols representing the 576 original frequency lines. The scalefactors are then used in *Requantizer* subcore. The output values from *Huffman* subcore are put in a memory ***(mainmem.vhd)*** with the size of 576 words. All Huffman tables are stored in BSRs of Virtex-II Pro (the actual number of tables stored is 17, tables number 16 to 23 are the same and so are tables number 24 to 31).

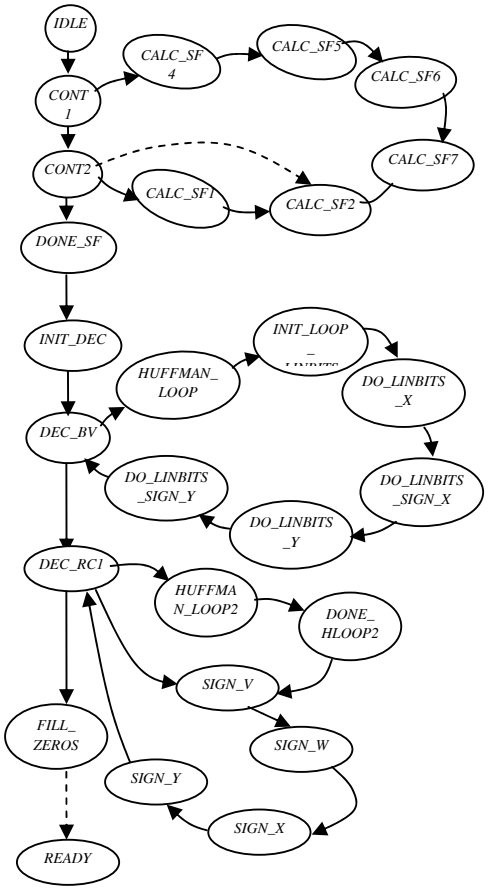


Fig. 11 The state machine diagram of *Huffman*

The Huffman subcore was designed on state machine that is shown in figure 11 with the following states:
IDLE: the state that does nothing, waiting for *start*.
CONT1: the state that checks the type of window and appoints to the next state.
CONT2: the state that assigns the next states: the sate using *mixed window* and the state using *short window*.
CALC_SF1: the state that calculates the scalefactors using *mixed window*.
CALC_SF2: the state that calculates the scalefactors using *short window.*

CALC_SF4, CALC_SF5, CALC_SF6, CALC_SF7, DONE_SF: the states that calculate the scalefactors using *long window*, these states use the *sfsi* (*Scale factor Selection Information)* to determine weather the same scalefactors are transferred for both *granules* or not.
INIT_DEC: After all of scalefactors are calculated, they are divided into three parts. This state is the state that starts the decoding and assigns the first decoding state.
DEC_BV: the sate that decodes *big-values* region, it decides what Huffman tables are used.
HUFFMAN_LOOP: this state performs the decoding for *big-values* region and determine weather the *linbits* are used or not.
INIT_LOOP_LINBITS: the state that implements the adding the *linbits* to the pairs of *(x, y)* value if they are *escape codes*.
DO_LINBITS_X, DO_LINBITS_SIGN_X: two state that implement the adding the *linbits* to value x (x is an *escape code*).
DO_LINBITS_Y, DO_LINBITS_SIGN_Y: two state that implement the adding the *linbits* to value y (y is an *escape code*).
DEC_RC1, HUFFMAN_LOOP2, DONE_HLOOP2: The states represent the decoding process for *coun1* region. As a result, produce the values in four *frequency lines* for each input stream.
SIGN_V, SIGN_W, SIGN_X, SIGN_Y: four states are corresponding to calculating four frequency lines for each input stream when decoding in the *count1* region.
FILL_ZEROS: the state that represents the last region of 576 output values, they are filled with zeros.
READY: the state is ready to read the current *granule*.

3.3 Requantizer Subcore

The architecture of *Requantizer* subcore ***(requantizer.vhd)*** is shown in figure 12. The data stored in the *Gain_correction* (made from *Core Generator*) is coefficients defined in *ISO/IEC 11172 standard* [3].

Requantize block is written in the way of state machine (figure 13). This block interfaces direct with *Controller* block *(controller.vhd)*, main memory block *(mainmem.vhd)* and Huffman decoder block *(huffman.vhd)*. The task of the block is to search *frame header* and *side information* for data.

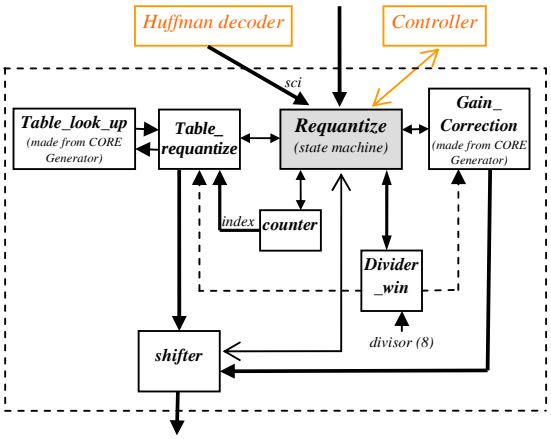


Fig. 12 The architecture of Requantizer subcore

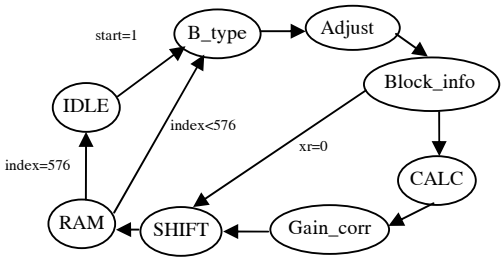


Fig. 13 The state machine diagram for requantize

The first step of the requantization process is to calculate $|is_i|^{\frac{4}{3}}$. The direct calculation of exponent 4/3 in VHDL code is difficult to implement and uses much logic resources. This paper offers a way to improve the performance is to use a *look-up table* containing all the 8192 possible input values. The above calculation can be divided into two cases:

- If *is(i)* < 1024, the result can directly be found in the look-up table
- If is(i) ≥ 1024, the value is first divided by 8. The result from the look-up table is then multiplied by 16.

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.
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 $2^{\frac{1}{4}C}$, see equations (1) and (2)).
The *gain correction* is another table used for storing the correction factor. The table is stored in BSRs of *Virtex-II Pro* and is included as a part of the initialized data section. A shared multiplier is used in the requantization calculations for multiplying *xr(i)* with the correction factor.

3.4 Reorder Subcore

Reorder subcore ***(reorder.vhd)*** 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 architecture of *Reorder* subcore is shown in figure 14.

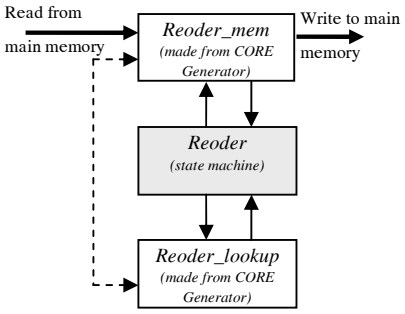


Fig. 14 The architecture of reorder subcore

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

3.5 Antialias Subcore

The encoder applies an alias reduction after the subband synthesis, because alias effects are introduced after a non-ideal bandpass filtering. So, the task of the *antialias* subcore ***(antialias.vhd)*** is to decrease these alias effects. This work can be performed by merging the frequency lines using eight butterfly calculations for each subband. The architecture of this subcore is shown in fig. 15.

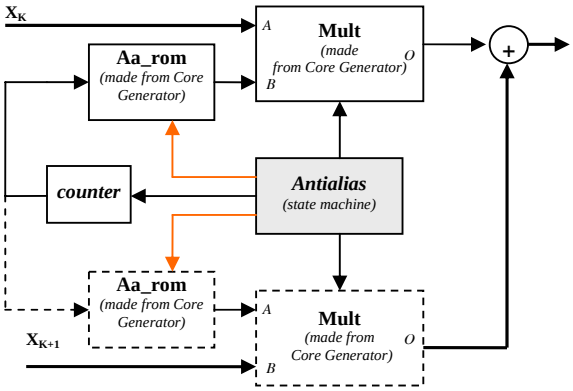


Fig. 15 The architecture of subcore antialias

The implementation of the antialias subcore is a state machine containing one butterfly calculation and some counters. Data (X_k) 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 constants (specified standard [3]) used for butterfly calculations are stored in BSRs of Virtex-II pro *(aa_rom)*.

3.6 IMDCT subcore

Give the frequency lines X_k, the time samples x_i can be obtained by using equation (3). The architecture for subcore IMDCT ***(imdct.vhd)*** is shown in figure 16.

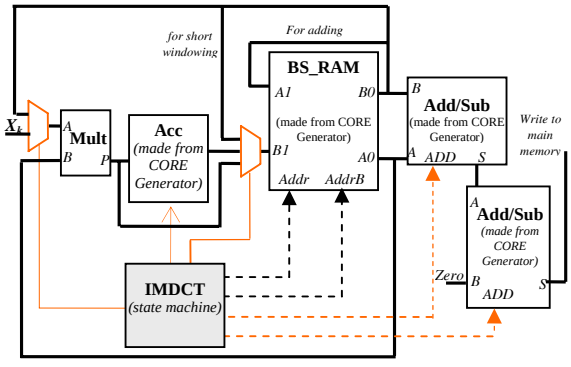


Fig. 16 The architecture of IMDCT subcore

All cosine-values for each output are considered as a the known constants based on the combination of i and k in (3). Some survey of

these values shows symmetry between the different x_i. Only half of the values are determined. The rest can be obtained as a function of the previously calculated values. Therefore calculating the first quarter and the third quarter of all values will be enough to determine the entire set.

For all multiplications the common multiplier is used. To obtain the summations of each x_i (see equation 3) an accumulator is used. All cosine-values (used for IMDCT-computations), and sine-values (used for windowing) are stored in BSRs of Virtex-II Pro (BS_RAM block).

3.7 Filterbank subcore

The architecture of the *Synthesis polyphase filterbank* subcore ***(filterbank.vhd)*** is shown figure 17. The *synthesis polyphase filterbank* process can be divided in two parts: a part for calculating 32 point - MDCT and the second part - *windowing* and summation of 512 values to produce output samples.

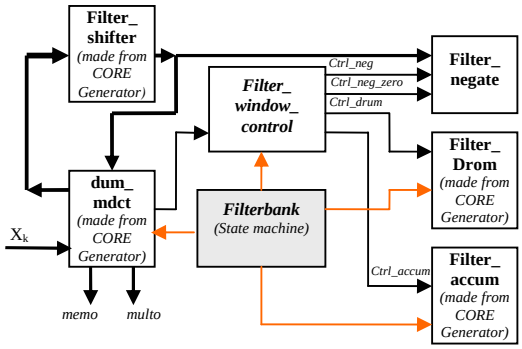


Fig 17. The architecture for filterbank subcore

A 32-point modified DCT requires 32x32multiplications using a non-optimized calculation method. Algorithms performing fast DCT computations are available, and based on the symmetry of the DCT matrix. This paper suggests an algorithm for DCT calculation – it is Lee's algorithm [11]. It has a simple recursive structure where the transform is decomposed into even and odd parts:

$$X(n) = \sum_{k=0}^{N-1} x(k)\cos(\pi(2k+1)\frac{n}{2N}),$$

$$for\ \ n=0\ to\ N-1 \qquad (4)$$

$$g(k) = x(k) + x(N-1-k)$$

$$h(k) = \frac{1}{2\cos(\pi(2k+1)\frac{n}{2N})}(x(k) - x(N-1-k))$$

$$G(n) = \sum_{k=0}^{\frac{N}{2}-1} g(k)\cos(\pi(2k+1)\frac{n}{N}),\ for\ n = 0\ to\ N/2-1$$

$$H(n) = \sum_{k=0}^{\frac{N}{2}-1} h(k)\cos(\pi(2k+1)\frac{n}{N}),\ for\ n = 0\ to\ \frac{N}{2}-1$$

$$For\ n=0\ to\ N/2-1:\ X(2n) = G(n)$$

$$X(2n+1) = H(n) + H(n+1),\ H(N/2)=0 \qquad (5)$$

In this design, two BSRs are used to store the DCT and *windowing* coefficients as well as two BSRs used as a dual port shift register for passing data between the DCT and *windowing* blocks.

3.8 Interface subcore

The task of Interface subcore ***(giaotiep.vhd)*** is *communication with the real world. T*he output data should be sent according to the I2S protocol – the standar developed by Philips. The I2S standard dictates that data is sent over a synchronous serial bus. The bus is a three wire bus consisting of a clock line, a word select line and a serial data line.

Since the mp3 decoder works with *frames* and *granules*, therefore it can not produce a continuous chain of data, but can produce 576 samples at a time. To make good this weak point, we design a FIFO-buffer using BSRs of Virtex-II Pro to store output data, and is used as a 1024 sample buffer.

In this design, the sample rate is fixed at 44.1 KHz. However, the system clock of Virtex-II Pro is 24MHz; this is not suitable for audio applications. So, we design a *division* block that it can divide the clock by 544 and produces a 44.118 kHz word clock.

4. LOGIC HARDWARE RESOURCES

The source code for the entire Core (referred from C code in sources [6], [7], [8], [9]) is compiled and synthesized on *Xilinx ISE 7.1i* by *Xilinx XST* tool, the result as follows:

```
Device utilization summary:
Selected Device : 2vp30ff896-7

Number of Slices:        3630  out of  13696    26%
Number of Slice FFs:     1583  out of  27392     5%
Number of 4 input LUTs:  6213  out of  27392    22%
Number of bonded IOBs:     27  out of    556     4%
Number of BRAMs:           23  out of    136    16%
```

```
Number of MULT18X18s:       4  out of    136     2%
Number of GCLKs:            2  out of     16    12%
Timing Summary:
   Minimum  period:  22.351ns  (Maximum  Frequency:
44.741MHz)
   Minimum input arrival time before clock: 7.907ns
   Maximum  output  required  time  after  clock:
13.127ns
   Maximum combinational path delay: 6.277ns
```

To verify the operation of the CORE, a testbench model for core and a model of MP3 player using the designed core are presented in figures 18 and 19.

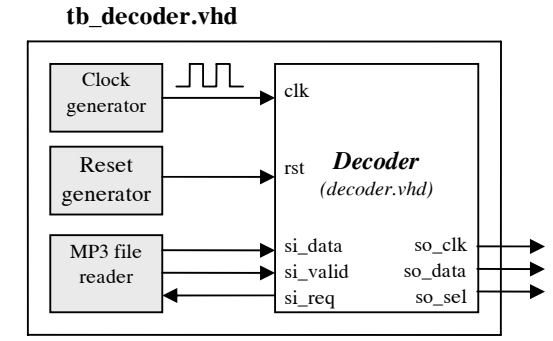


Fig. 18 The testbench model for core

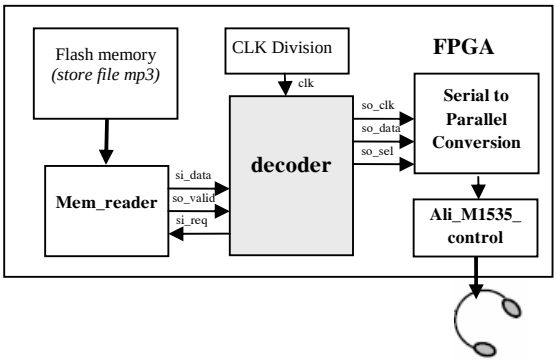


Fig. 19 The model of MP3 player

5. CONCLUSIONS

From the results obtained by the verification of two proposed models in Figures 18 and 19, the CORE has been met the aim of the design's demands. However, the decoding process was only implemented in *mono* mode, so the sound quality of decompressed audio data of a sample stereo Mp3 signal is not completely perfect.

To increase the quality of the core and decrease logic resources for the core implementation, the authors were interested in the choice of decoding algorithms. Some algorithms for optimizing can be used in the implementation of subcores, especially *IMDCT, filterbank, requantizer* subcores. In this paper, the authors have implemented some algorithms to the VHDL hardware programming for fast computations, such as the fast MDCT, Lee's algorithm [11]. The coefficients defined in [3] are proposed to be stored in BSRs of Virtex-II Pro instead of writing to VHDL code as constants for a fast signal processing and saved resources of FPGA board.

Acknowledgement:
The authors would like to express our gratitude to the Electronics Division for supporting the use of Virtex-II Pro.

REFERENCES

[1] Thuong Le-Tien, Signal Processing and Wavelets, Hochiminh city National University Publisher, 2002.

[2] Michael Keating and Pierre Bricaud, Reuse Methodology manual for SoC designs, Kluwer Academic Publishers, 2001.

[3] Michael Robin and Michel Poulin, ISO/IEC 11 172-3, Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3, Digital Television Fundamentals, Mc Graw-Hill, 1997.

[4] K. Brandenburg, H. Popp, An Introduction to MPEG Layer-3, Fraunhofer Institute, EBU Technical Review, June 2000.

[5] D. Pan et al., IIS MP3 Decoder Source Code, *http://www.mp3-tech.org*, April 1995.

[6] W.Jung, SPLAYMP3 Decoder Source Code, *http://splay.sourceforge.net*, April 2001.

[7] M.Hipp et al., MPG123 Decoder Source Code, *http://www.mpg123.de*, April 2001.

[8] K. Lagerstrum, MP3 Ref. Decoder Source Code, *http://www.dtek.chalmers.se*, 2001.

[9] Staffan Gadd: A hardware accelerated mp3 decoder with bluetooth streaming capabilities, *Master of science Thesis*, 2001.

[10] 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.