

國立交通大學

資訊科學與工程研究所

碩 士 論 文

以 Balsa 設計之非同步 MP3 解碼器

An Asynchronous MP3 Decoder Design with Balsa

研 究 生：羅浩暉

指 導 教 授：陳昌居 教授

中 華 民 國 九 十 六 年 六 月

以 Balsa 設計之非同步 MP3 解碼器

An Asynchronous MP3 Decoder Design with Balsa

研究生：羅浩暉

Student：Hau-Wei Lo

指導教授：陳昌居

Advisor：Chang-Jiu Chen

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

以 Balsa 設計之非同步 MP3 解碼器

學生：羅浩暉

指導教授：陳昌居

國立交通大學資訊科學與工程所碩士班

摘要

MP3 是最流行的音樂壓縮標準之一，被應用在很多手持式裝置上，特別是 PDA 和手機上。因為這些裝置都是攜帶式的，都被期望有低耗電，快速研發及上市時間，以及好的相容性，因此模組化變成越來越重要。

我設計並且實現一個適合於管線化架構下的非同步 MP3 解碼器。這個設計被分成八個主要的階段，Synchronizer&Huffman, Requantizer, Reordering, Anti-alias, IMDCT, BUFF, FilterBank and PCM_out。在行為模式下成功地通過驗證，並且以 Synplify 和 Synopsys Design Compiler 合成器轉成電路。最後，對於合成的結果做一些討論。

因此，在使用 Balsa 去設計一個較複雜的非同步電路上，我做了一個新的嘗試，並且使用 CAD 工具去設計非同步電路的流程也得以確認。

An Asynchronous MP3 Decoder Design with Balsa

student : Hao-Wei Lo

Advisors : Dr. Chang-Jiu Chen

Abstract

MP3 is one of the most popular digital audio compression standards, and it is applied to many handset electronic products, especially PDAs and mobile phones. Because all of these products are portable, they demand speed to market, low power consumption and better composability, therefore, modularity becomes more and more important.

I designed and implemented a processing flow of an asynchronous MP3 decoder, which is suitable for pipeline architecture, called PAMP3. The PAMP3 is divided into 8 stages that are Synchronizer&Huffman, Requantizer, Reordering, Anti-alias, IMDCT, BUFF, FilterBank and PCM_out. I successfully passed the behavior simulation and synthesized the design with the Synplify synthesis tool and the Synopsys Design Compiler tool. Finally, I made a discussion about the synthesis results.

Therefore, I made an attempt to implement a complex circuit with Balsa in ways of asynchronous circuit design, and the design flow of using CAD tools to design an asynchronous circuit is confirmed.

Acknowledgment

能夠完成這篇論文，首先我要感謝我的指導教授，陳昌居 老師，謝謝兩年的辛苦教導。還要感謝一直熱心幫助我的學長，鄭緯民，以及所有 LAB 616 的學長和學弟妹的支持鼓勵。

謝謝一直陪在我身邊的女友，還有我最親愛的家人，沒有你們，我無法順利完成學業，謝謝。



CONTENTS

摘要	i
Abstract.....	ii
Acknowledgment.....	ii
CONTENTS	iii
List of Figures.....	v
List of Tables.....	viii
Chapter 1: Introduction.....	1
1-1 Motivations.....	1
1-2 Asynchronous circuit design	2
1-3 Balsa Synthesis Tool	4
1-4 Organization of this thesis.....	6
Chapter 2: Related works	7
2-1 Introduction to MP3	7
2-1-1 Frame format.....	8
2-1-2 Side information and Main data.....	9
2-1-3 Huffman decoding	10
2-2 The MP3 processing flow	12
2-2-1 Bitstream decoding	12
2-2-2 Inverse quantization.....	13
2-2-3 The Frequency to time mapping block	15
2-3 Overview of Pipeline Architecture.....	19

2-4 Balsa back-End.....	21
2-4-1 Basic Elements.....	21
2-4-2 Handshake Components.....	24
2-5 Concluding Remarks	27
Chapter 3: The Design of the PAMP3	28
3-1 The architecture of the PAMP3	28
3-2 The synchronizer&Huffman stage	30
3-3 The re-quantizer stage	31
3-4 The reorder stage and anti-alias stage	32
3-5 The IMDCT stage.....	34
3-6 The BUFF stage	37
3-7 The poly-phase filterbank stage.....	38
3-8 The PCM_out stage.....	41
Chapter 4: Implementation and Verification	43
4-1 The Design Flow	43
4-2 Implementation Issues.....	45
4-3 Verification	45
Chapter 5: The Results	49
5-1 Simulation Result	49
5-2 Area cost.....	49
Chapter 6: Conclusions.....	53
Reference	54



List of Figures

Figure 1: The two-phase handshake protocol.....	4
Figure 2: The Four-phase handshake protocol	4
Figure 3. The Balsa Design Flow	5
Figure 4: The frame structure of a MP3 file.....	9
Figure 5: The bit reservoir technology	10
Figure 6: The five regions of Huffman data.....	11
Figure 7: The Huffman decoding flow	12
Figure 8: The MP3 decoding process.....	12
Figure 9: The bitstream decoding block.....	13
Figure 10: The reordering method.....	14
Figure 11: The frequency to time mapping block.....	15
Figure 12: The alias reduction.....	16
Figure 13: The overlapping of the IMDCT	18
Figure 14: The flow of poly-phase synthesis.....	19
Figure 15. Synchronous Pipeline V.S. Asynchronous Pipeline	20
Figure 16: The Muller C-element, (a) symbol (b) true table (c) gate-level implementation....	22
Figure 17: The NC2P-element (a) symbol (b) true table (c) gate-level implementation.....	22
Figure 18: The S-element (a) symbol (b) gate-level implementation (c) handshaking protocol	23
Figure 19: The Fetch Component (a) handshake component (b) gate-level implementation ..	24

Figure 20: The Sequence Component (a) handshake component (b) gate-level implementation	25
Figure 21: The Concurrent Component (a) handshake component (b) gate-level implementation	25
Figure 22: The variable Component (a) handshake component (b) gate-level implementation	26
Figure 23: The architecture view of the PAMP3	29
Figure 24: The Synchronizer&HUFFMAN stage	31
Figure 25: the Requantizer stage	32
Figure 26: The reorder stage.....	33
Figure 27: The anti-alias stage.....	34
Figure 28: The register banks of the anti-alias stage.....	34
Figure 29: The IMDCT processing flow of Szu-Wei Lee's algorithm.....	35
Figure 30: the sub-pipeline of the IMDCT stage.....	36
Figure 31: The BUFF stage	38
Figure 32: The DCT simplification of Konstantinides' algorithm	39
Figure 33: The 8-point DCT simplification of B.G. Lee's algorithm [11].....	39
Figure 34: the sub-pipeline of the synthesis filterbank stage	40
Figure 35: The Balsa and FPGA design flow	44
Figure 36: PAMP3 behavior simulation enviroment.....	46
Figure 37: Playing output in the GoldWave	47
Figure 38: Observer for MP3 decoder.....	47

Figure 39: The Balsa memory model48

Figure 40: The CallMux handshake component (three 10-bit ports)51



List of Tables

Table 1: The advantages and disadvantages of asynchronous design	3
Table 2: The comparisons between the three layers of MPEG-1	8
Table 3: The Cell Area Cost of two kinds of multipliers (μm^2).....	49
Table 4: The Cell Area Cost of Every Part of PAMP3 (μm^2)	50
Table 5: The FPGA Cost of Every Part of PAMP3.....	50



Chapter 1: Introduction

1-1 Motivations

A digital system is usually organized by many subsystems, and these subsystems exchange information with each other. To guarantee the validity of data exchanged, the whole system needs to be synchronized on each transaction. The synchronization method is use of a global clock. The global clock is limited by the longest execution time of subsystems and it causes the worst case performance in the pipeline architecture. In addition, the global clock also causes higher power consumption. Therefore, the clock distribution is increasingly becoming a costly problem.

A number of asynchronous circuit design methodologies and implementations have been proposed and developed [7], for example, asynchronous ARM RISC processor (AMULET1, AMULET2e, AMULET3) [8] [14] [15] in University of Manchester and Lutonium processor in CalTech [3].

The MP3 is one of the most popular digital audio compression standards [6] [13], and it is applied to many handset electronic products, especially PDAs and mobile phones. Because all of these products are portable, they demand speed to market, low power consumption and better composability; therefore, modularity becomes more and more important.

We used the advantages of the asynchronous circuit design to implement a MP3 decoder with the pipeline method and the asynchronous design to increase its modularity and reduce power consumption.

1-2 Asynchronous circuit design

Synchronous circuit design is the major design method recently, because of being used widely, and it has a complete design flow and tools. The system clock may cause problems in designing a large high clock frequency chip, and that's why the asynchronous design becomes more and more important. There are some advantages in the asynchronous circuits while comparing with synchronous circuits. The first major problem of synchronous circuits is the clock skew, while asynchronous circuits don't need the clock. Second, synchronous circuits are limited as worst case performance, while asynchronous are not. Third, asynchronous circuits have no clock signal; therefore, the power consumption can be reduced. It also almost attains zero power dissipation when there is no useful work to do. Forth, asynchronous circuit has better modularity. It is easier to connect every component with the same communication protocol. Finally, an asynchronous design has a low EMI(Electromagnetic Influence) problem because of no clock distribution.

But asynchronous circuits still have some challenges over synchronous circuits. First, asynchronous circuits without clock signals need more control signals and thus the area cost may be increased. Second, there are few CAD tools to support asynchronous designs and tests. Therefore, it makes asynchronous circuit design harder and it also causes longer designing time. These are the major challenges of asynchronous circuit designs. The advantages and challenges of asynchronous circuit design are shown in Table 1.

Advantages	challenges
Low power consumption	Overhead(Area, Speed, Power)
Average-case instead of worst-case performance	Hard to design
Elimination of clock skew problems	Few CAD tools
Component modularity and reuse	Lack of tools for testing
Low EMI	

Table 1: The advantages and challenges of asynchronous design

In asynchronous circuits, the major communication method between two components is by handshaking. There are two main types of control signaling protocol in the asynchronous circuit designs: two-phase and four-phase. The active signal of a two-phase handshake protocol can be a falling or rising edge. After an activity, the control signal doesn't need to be reset to zero. The two-phase handshake protocol is shown in Figure 1. When the data of the sender is ready, the sender changes the request signal state (0 → 1, 1 → 0). Then the receiver changes its acknowledged signal and gets the data at the same time, and the handshake is completed. The periods between a request and an acknowledgement are the handshake itself, and the periods between an acknowledgement and the next request are an idle phase. The data transition must obey the setup time and the holding time constrain.

The other type of handshake protocol is a four-phase. It is different from the two-phase protocol; the active signal must be a rising edge. It means the handshake signal must be reset after an activity. The four-phase handshake protocol is shown in Figure 2. When the data of the sender is ready, the sender pulls up the request signal, Req. Then, the receiver will pull up the acknowledge signal, Ack, and the data at the same time. At the end, the sender will push down the Req, and the receiver will push down the Ack. The handshake is completed.

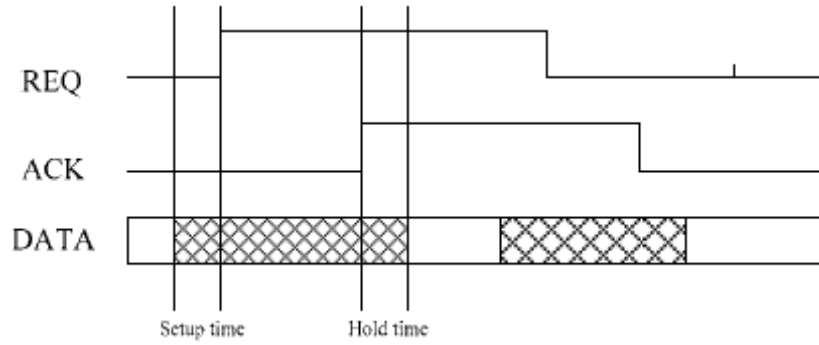


Figure 1: The two-phase handshake protocol

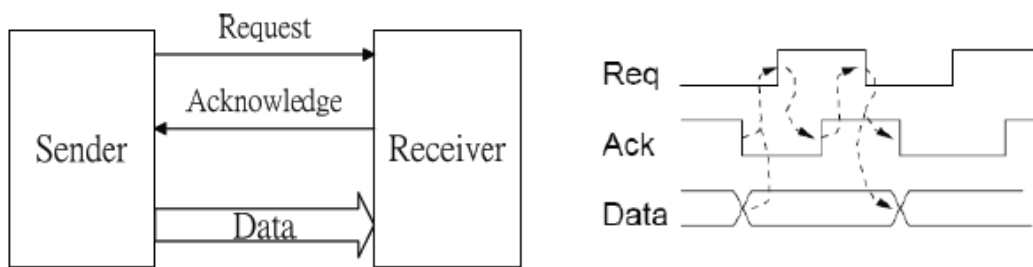


Figure 2: The Four-phase handshake protocol



1-3 Balsa Synthesis Tool

Balsa is an asynchronous hardware description language & synthesis tool developed by the Advanced Processor Technologies Group of the Manchester University [5] [12]. Balsa synthesis tool can compile the Balsa HDL into handshake components (breeze file) with one-to-one mapping. It is relatively easy for an experienced user to explore the architecture of the circuit that results from the original description. We can describe the behavior of our design with Balsa HDL, and then compile it into the most popular hardware description language such as Verilog HDL. An overview of the Balsa design flow is shown in Figure 3.

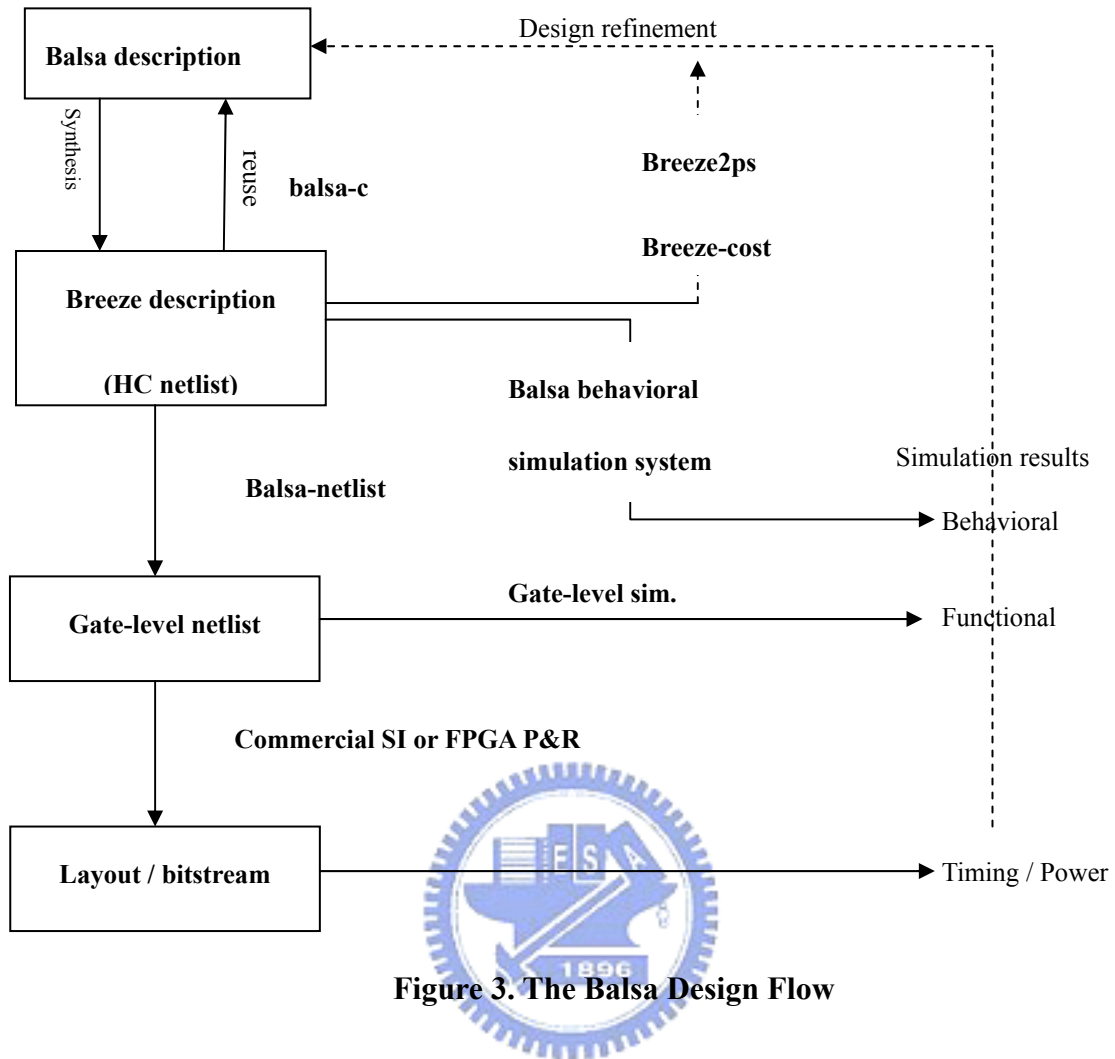


Figure 3. The Balsa Design Flow

Balsa uses CSP-based constructs to express RTL design descriptions in terms of channel communications and fine grain concurrent and sequential process decompositions. A Balsa description (*.balsa* file) is translated into an implementation in a syntax directed fashion. Balsa uses one-to-one mapping between the language circuits and the intermediate handshake circuits. After translating, it will be a file (*.breeze* file) in a language with networks of handshake components. *Balsa-netlist* can generate CAD native netlist files, and then these netlist files can be fed into commercial CAD tools. Finally, the CAD tools synthesize the netlist to the layout. Three commercial CAD systems are currently supported: Compass Design Automation tools from Avant, Xilinx Alliance FPGA design tools and Cadence Design Framework II.

Balsa supports three back-end technologies: signal rail (bundled data), dual rail, 1-of-4. The bundled-data back-end should be faster and smaller, but needs more careful post-layout timing validation. The dual rail and 1-of-4 schemes are larger and slower, but they could be more robust to layout variations.

1-4 Organization of this thesis

In the following chapters, the related work will be introduced, such as the processing flow of a MP3 decoder, the overview of pipeline architecture and the basic Balsa back-end. Then, the fully MP3 decoder design of the pipelined asynchronous MP3 will be illustrated in chapter 3. In chapter 4 and 5, the implementation, verification and results will be illustrated. Finally, a brief conclusion and future work are discussed in chapter 6.



Chapter 2: Related works

2-1 Introduction to MP3

The compression technology supported by MPEG (Moving Picture Expert Group) is widely used in various current multimedia applications, for example, network multimedia streamings, online music stores, digital televisions, and portable devices.

In the MPEG-1 standard, the compression of an audio signal can be categorized to three layers, MPG Layer 1, MPEG Layer 2, and MPEG Layer 3. These layers are different in codec complexity and compressed audio quality. Layer 1 forms the basic algorithms and is suitable for the bit rate above 128 Kbps per channel. Layer 2 targets the bit rates around 128 Kbps per channel and provides additional coding of bit allocations, scalefactors and samples. Layer 3 is the most complex, but it offers the best audio quality. A common CD music is about 44.1KHZ in frequency and 16 bits in sampling, so it consumes around 10 MB of storage space per minute. MP3 music only needs 1 MB storage space per minute. The compression rate of the MP3 music is 10 ~ 12 times the compression rate of a common CD music. The comparisons between the three layers of MPEG-1 are shown in Table 2.

	Layer I	Layer II	Layer III
Analysis/Transform	32 sub-bands	32 sub-bands	32 sub-bands
Psychoacoustics model	Model 1	Model 1	Model 2
Bit Rate	32~448 kbps	32~384 kbps	32~320 kbps
Sample Frequency	32, 44.1, 48 KHZ		
Quantize	Uniform	Uniform	Non-uniform
Samples per frame	384 samples	1152 samples	1152 samples

Table 2: The comparisons between the three layers of MPEG-1

2-1-1 Frame format



All MP3 files are divided into smaller fragments called frames. Each frame stores 1152 audio samples divided into two granules of 576 samples each and lasts for 26 ms. The frame structure of a MP3 can be divided into five parts as shown in Figure 4. Each header of the MP3 frames is 32 bits, includes some information about this frame, Sync word, ID, Layer, CRC, Sampling frequency, etc. The side information of each frame will be used in the following parts: the Huffman decoder and the scalefactor decoder. The main data part of the frame consists of scale factors, Huffman coded bits and ancillary data.

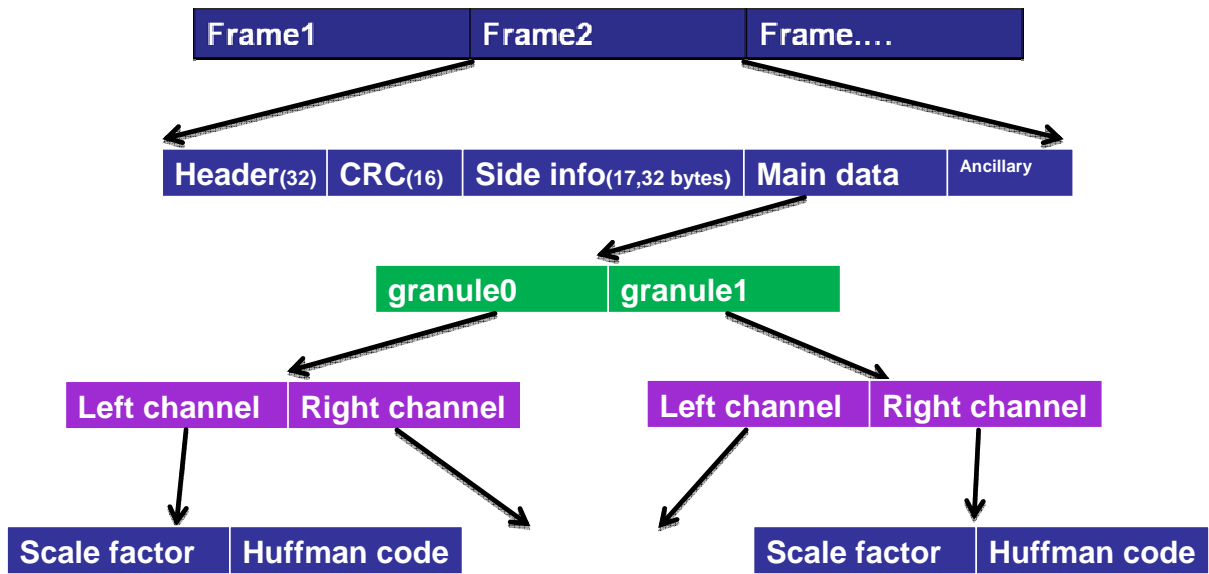


Figure 4: The frame structure of a MP3 file

2-1-2 Side information and Main data

The side information section contains the necessary information to decode the main data. This section is 17 bytes long in the single channel mode and 32 bytes long in the dual channel mode.

The main data section contains the coded scale factor values and the Huffman coded frequency lines. Its length depends on the bitrate and the length of the ancillary data. The length of the scalefactor part depends on whether scale factors are reused, and also on the window length (short or long).

The first 9 bits of side information is a point tag which points out the main data beginning address in the current frame. Because the MP3 is encoded in the Huffman encoding, the lengths of the audio data after Huffman encoding are not all the same. In order to increase the space utility rate, the bit reservoir technology is used as shown in Figure 5. Therefore, the main data beginning address of each frame is not always after the side information of itself.

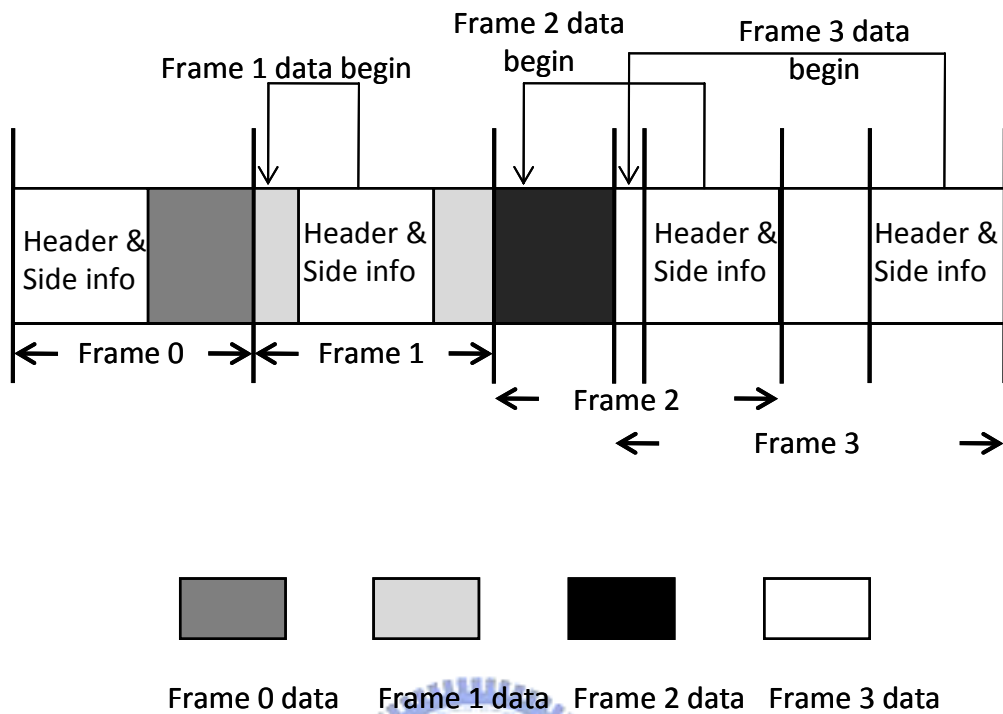


Figure 5: The bit reservoir technology

2-1-3 Huffman decoding

There are two parts in the main data section, the scalefactor part and the Huffman data part. The size of the Huffman data part can be known by the side information and the scale factors. The `big_values` of the side information are the spectral values coded with different Huffman code tables. These frequencies range from zero to the Nyquist frequency and are divided into five regions (See Figure 6). The `rzero` region contains pairs of quantized values that equal to zero and represents the highest frequencies. The `count1` region contains quadruples of quantized values that equal to -1, 0 or 1. Finally, the `big_values` region contains pairs of values, and the maximum of these values in the range are constrained to 8191 (13 bits). The `big_values` field indicates the size of the `big_values` region, and the maximum value

is 288.

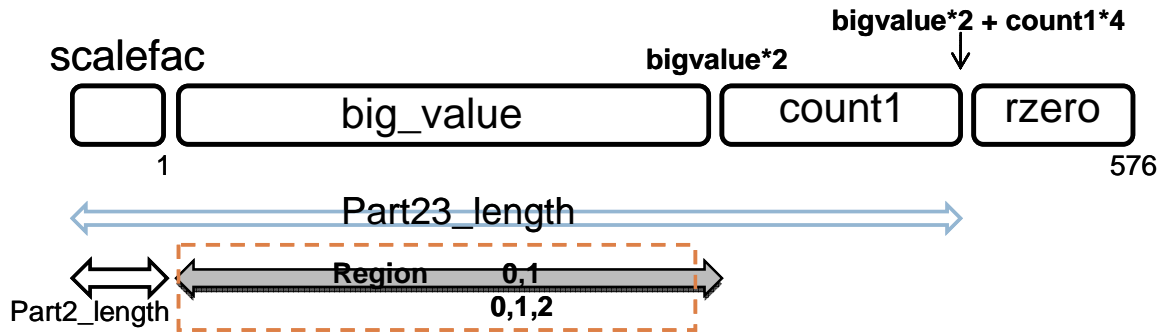


Figure 6: The five regions of Huffman data.

The Huffman decoding flow is shown in Figure 7.

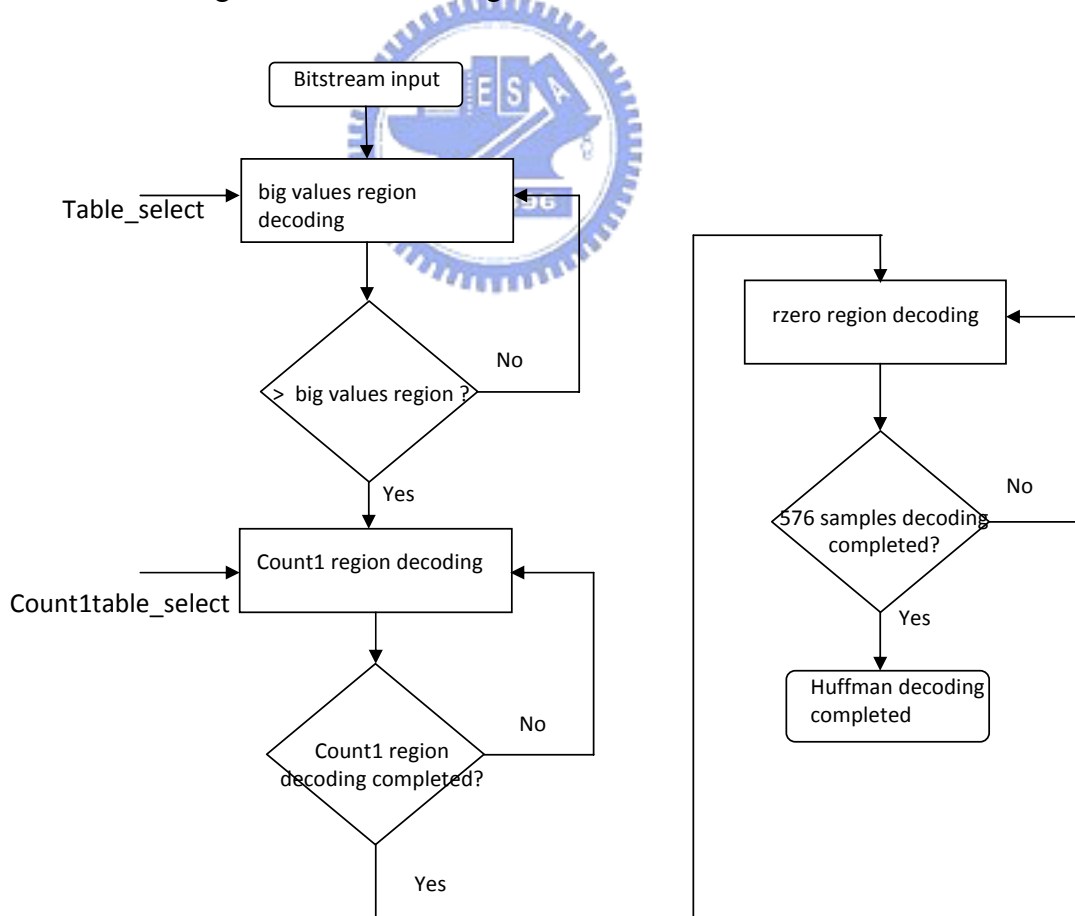


Figure 7: The Huffman decoding flow

2-2 The MP3 processing flow

The MPEG/Audio layer 3 decoding process has three main parts [10]: the bitstream decoding, the inverse quantization and the frequency-to-time mapping as shown in Figure 8

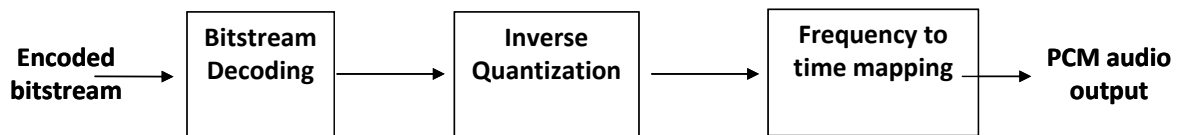


Figure 8: The MP3 decoding process

The bitstream data is fed into the decoder. The bitstream decoding block receives header and error detection if error-check (CRC error detection code) is applied in the encoder. The bitstream data are unpacked to recover the various pieces of information, and the inverse quantization block reconstructs the quantized version of the set of mapped samples. Finally, the frequency-to-time mapping block transforms these mapped samples back into a uniform PCM.

2-2-1 Bitstream decoding

There are four phases in the bitstream decoding part, which are the header decoding, the side information decoding, the scale factor decoding, and the Huffman data decoding. First, the bitstream decoder synchronizes one header address of a frame, and then it receives header

and side information data into buffer for usage in later phases. Third, the scale factor decoding phase decodes the scale factor data that is needed in re-quantization. Fourth, the Huffman data phase receives 576 factor values which are computed by MDCT and the quantization to do an ascending power sort. The bitstream decoding block is shown in Figure 9.

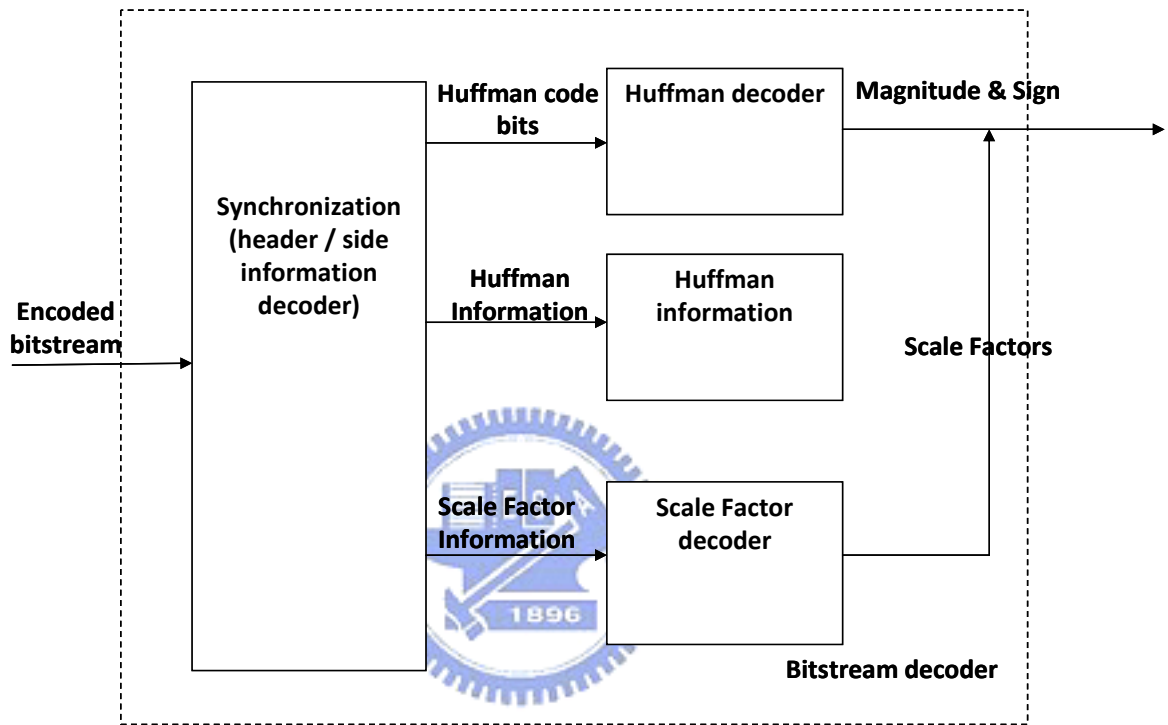


Figure 9: The bitstream decoding block

2-2-.2 Inverse quantization

There are three parts in the inverse quantization block: the re-quantization, the reordering and the joint-stereo processing. The re-quantization part covers the Huffman decoded values back to their spectral values using a power law. For each output value Y from Huffman decoder, $Y^{4/3}$ is calculated. So, it needs the scale factors and Huffman values that were decoded before. The following is the re-quantization formula:

$$Xr_i = is_i^{4/3} * 2^{(0.25 * C)} \quad (1)$$

The factor “C” in the equation consists of the global gain and the scalefactor band information from the side information and the scale factors. The value, is_i , means the Huffman decoded value at buffer index i , and the input to the next processing block at index i is called $Xr(i)$.

Reordering

In order to make the Huffman decoding more efficient, we must reorder the frequency value from MDCT and quantization. This part is only used in short block windowing. Because the three window samples in the same frequency of each subband are put together into one window during the Huffman encoding, and they must be converted back to the original order. The reordering method is shown in Figure 10.

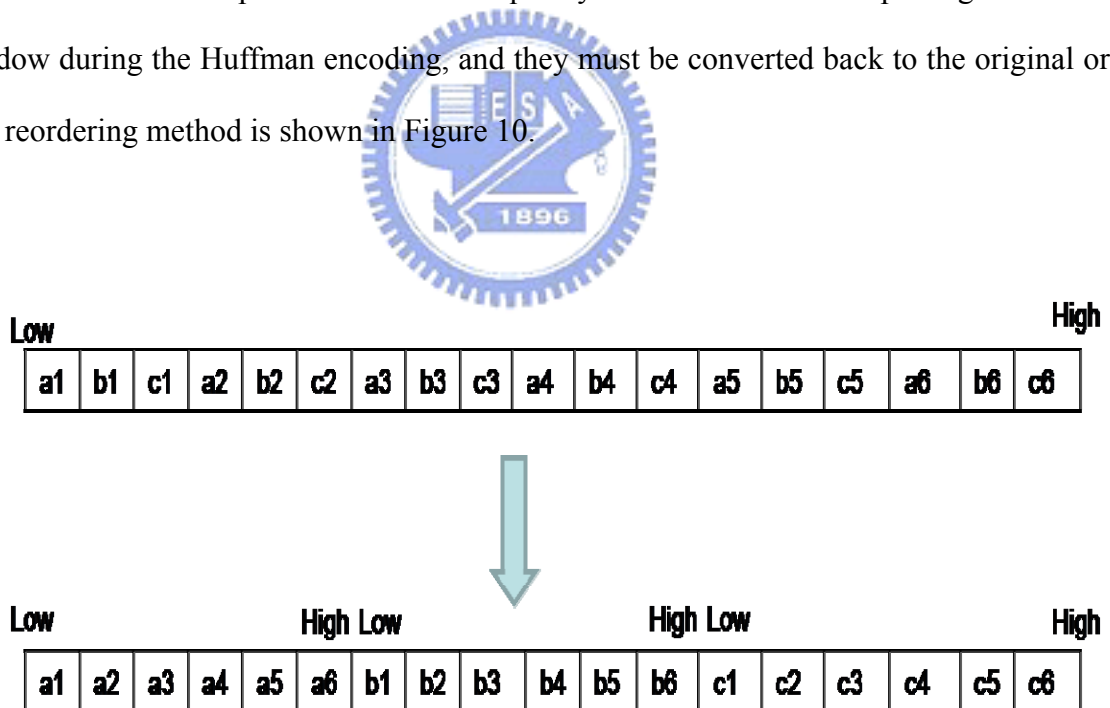


Figure 10: The reordering method

Join stereo processing

The MP3 decoding does not only support the mono or dual mode channel mode

decoding, but also support stereo mode channel mode decoding.

2-2-3 The Frequency to time mapping block

The frequency to time mapping block can be divided to three phases such as alias reduction, IMDCT, ploy-phase synthesis filter bank (Figure 11). The purpose of this block is converting the decoded re-quantization frequency domain values to the time domain values.

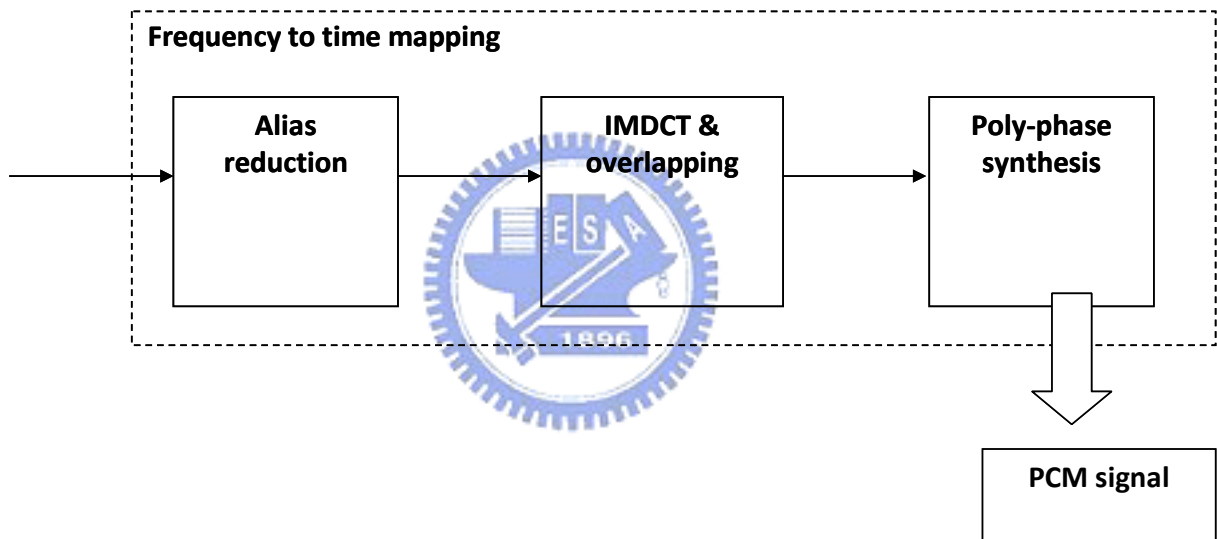


Figure 11: The frequency to time mapping block

Alias Reduction

The alias reduction is required to negate the aliasing effects of the poly-phase synthesis filter bank during encoding. It is used in the long block. There are eight butterfly calculations for each sub-band as shown in Figure 12. The $x(i)$ is the frequency value that is processed by a reorder module, and the cs and the ca are the constants that can be found in standard tables.

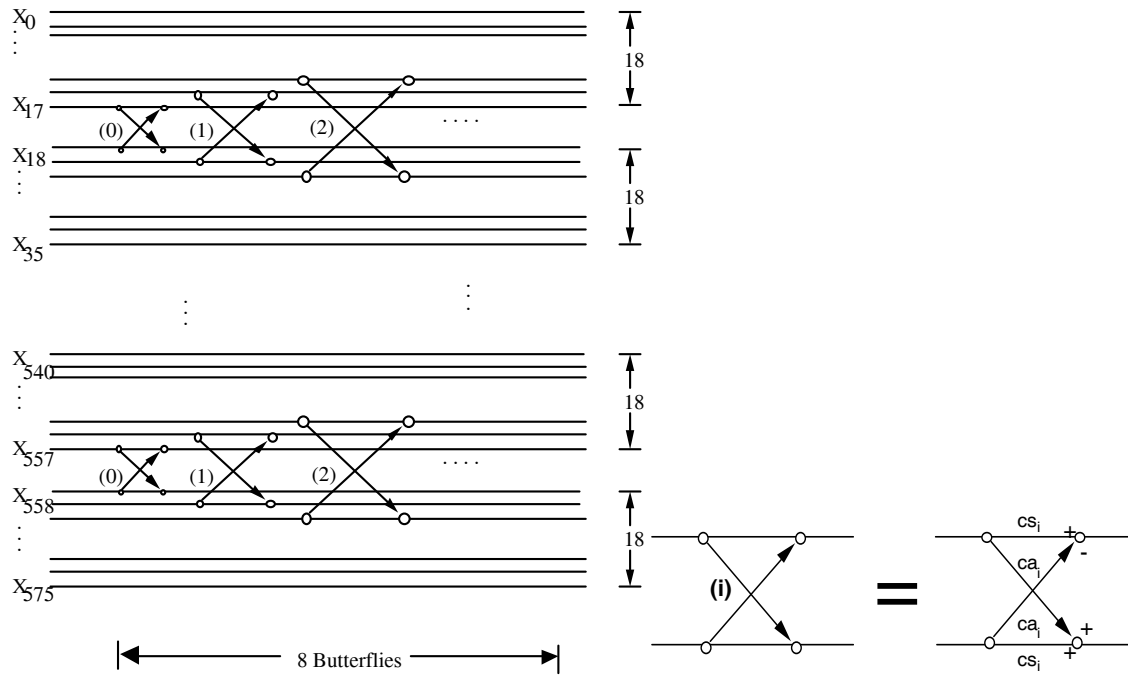


Figure 12: The alias reduction

IMDCT

The IMDCT (Inverse Modified Discrete Cosine Transform) transforms the frequency lines to poly-phase filter subband samples. The analytical expression of the IMDCT is as shown as below where n is 12 for short blocks and 36 for long blocks.

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left(\frac{\pi}{2n} \left(2i + 1 + \frac{n}{2}\right) (2k + 1)\right) \quad \text{for } i=0 \text{ to } n-1 \quad (2)$$

In long blocks, the input of IMDCT is formed with 18 frequency lines, and then the IMDCT produces 36 outputs. In a serial of three window blocks, the input of the IMDCT is formed with 6 frequency lines, every block produces 12 outputs.

After the IMDCT process, the result X_i must multiply with the function of windowing. And the function of windowing depends on the block_type different shapes of windows used.


1. Block_type = 0

$$z_i = x_i \sin\left(\frac{\pi}{36}\left(i + \frac{1}{2}\right)\right) \quad \text{for } i=0 \text{ to } 35 \quad (3)$$

2. Block_type = 1

$$z_i = \begin{cases} x_i \sin\left(\frac{\pi}{36}\left(i + \frac{1}{2}\right)\right) & \text{for } i=0 \text{ to } 17 \\ x_i & \text{for } i=18 \text{ to } 23 \\ x_i \sin\left(\frac{\pi}{12}\left(i - 18 + \frac{1}{2}\right)\right) & \text{for } i=24 \text{ to } 29 \\ 0 & \text{for } i=30 \text{ to } 35 \end{cases} \quad (4)$$

3. Block_type = 3



$$z_i = \begin{cases} 0 & \text{for } i=0 \text{ to } 5 \\ x_i \sin\left(\frac{\pi}{12}\left(i - 6 + \frac{1}{2}\right)\right) & \text{for } i=6 \text{ to } 11 \\ x_i & \text{for } i=12 \text{ to } 17 \\ x_i \sin\left(\frac{\pi}{36}\left(i + \frac{1}{2}\right)\right) & \text{for } i=18 \text{ to } 35 \end{cases} \quad (5)$$

4. Block_type = 2

$$z_i^{(j)} = y_i^{(j)} \sin\left(\frac{\pi}{12}\left(i + \frac{1}{2}\right)\right) \quad \text{for } i=0 \text{ to } 11, \text{ for } j=0 \text{ to } 2 \quad (6)$$

$$y_i = \begin{cases} 0 & \text{for } i=0 \text{ to } 5 \\ y_{i-6}^{(1)} & \text{for } i=6 \text{ to } 11 \\ y_{i-6}^{(1)} + y_{i-12}^{(2)} & \text{for } i=12 \text{ to } 17 \\ y_{i-12}^{(2)} + y_{i-18}^{(3)} & \text{for } i=18 \text{ to } 23 \\ y_{i-18}^{(3)} & \text{for } i=24 \text{ to } 29 \\ 0 & \text{for } i=30 \text{ to } 35 \end{cases} \quad (7)$$

After windowing, the results must be overlapped and added with the previous block. Half of the block of the 36 values is overlapped with the second half of the previous block. The second half of the actual block is stored to be used in the next block as shown in Figure 13.

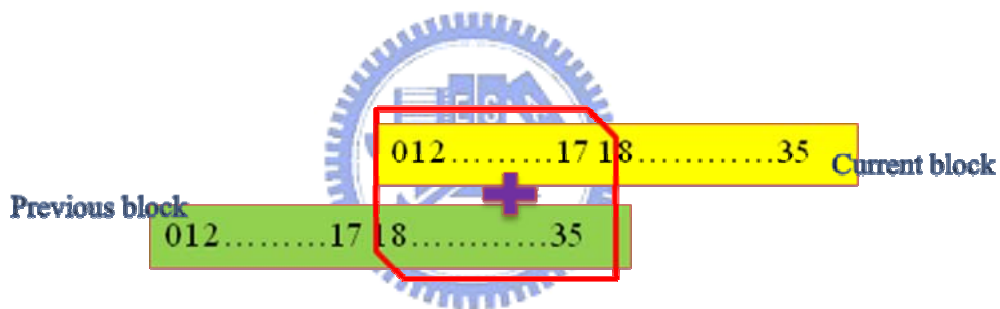


Figure 13: The overlapping of the IMDCT

Poly-phase Synthesis

The poly-phase synthesis (filterbank) block transforms the 32 subband blocks of 18 time-domain samples in each granule to 18 blocks of 32 PCM samples. This block can be divided to four parts: moving, DCT, matrix multiply and overall adding. The flow of poly-phase synthesis is shown in Figure 14.

In the synthesis operation, the 32 subband values are transformed to the 64 values V vector via the DCT computation. The V vector is pushed into the FIFO buffer, and a new vector, U vector, is created from the FIFO. Finally, the U vector is multiplied with the constant

D window to get the W vector, and these 16 W vectors are added with each other. The final 32 samples become a PCM vector.

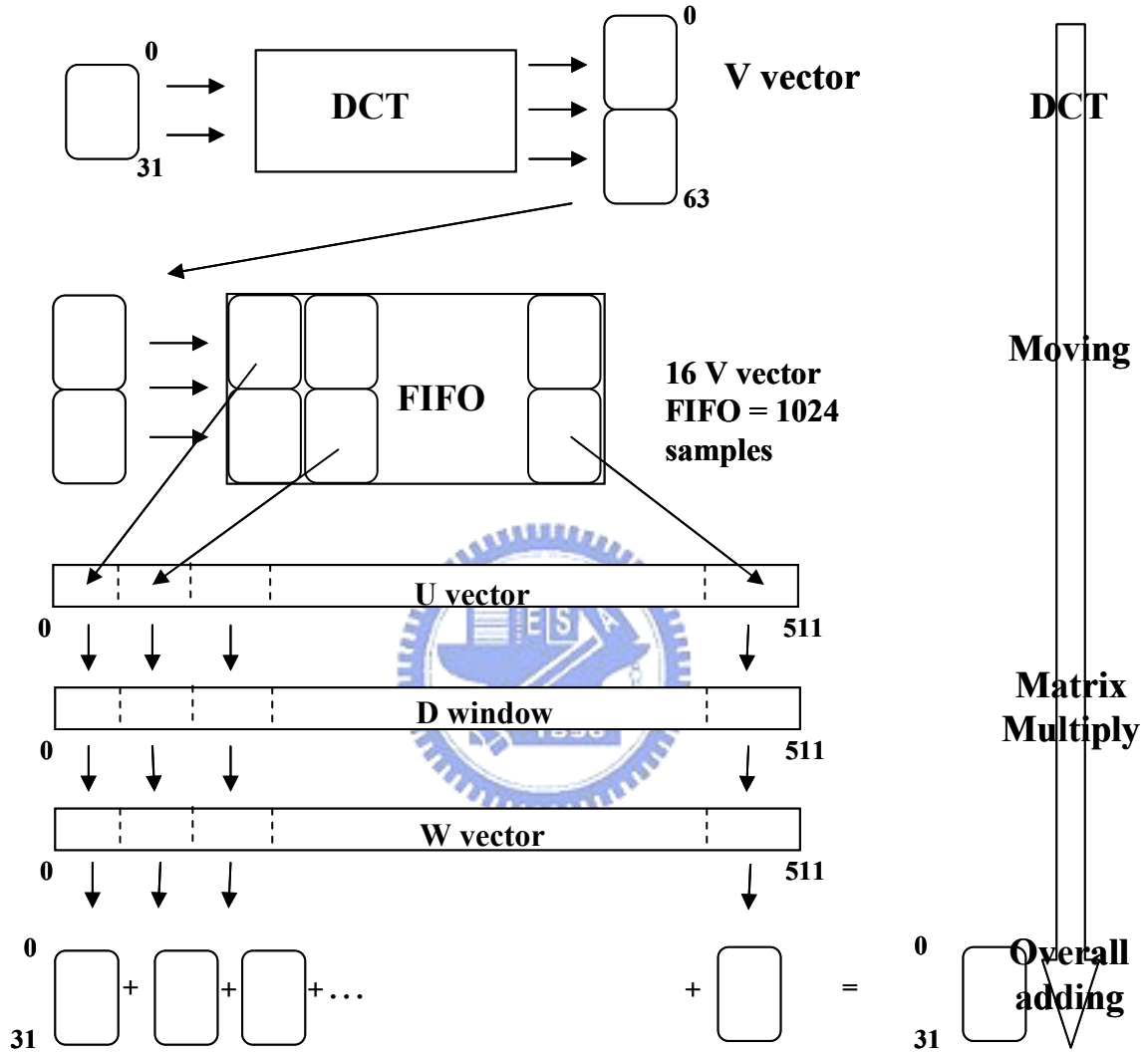


Figure 14: The flow of poly-phase synthesis

2-3 Overview of Pipeline Architecture

The pipeline architecture is used widely in microprocessor designs. It can increase the throughput due to the parallel processing of instructions. In Figure 16, every stage of the

synchronous pipeline must be controlled by a global clock. The period of the global clock is set to the slowest pipeline stage. However, each stage of the asynchronous pipeline can be processed at its own speed. Some instructions even can bypass the stages that aren't processed, such as instructions 3 and 4 in Figure 15. The instruction 3 doesn't need to be processed in the WB stage, and the instruction 4 doesn't need to be processed in the EXE stage. In the synchronous pipeline, these instructions still need to wait a complete clock cycle time before moving to the next stage. But in the asynchronous pipeline, these instructions can be processed quickly (bypassing) to the next stage.

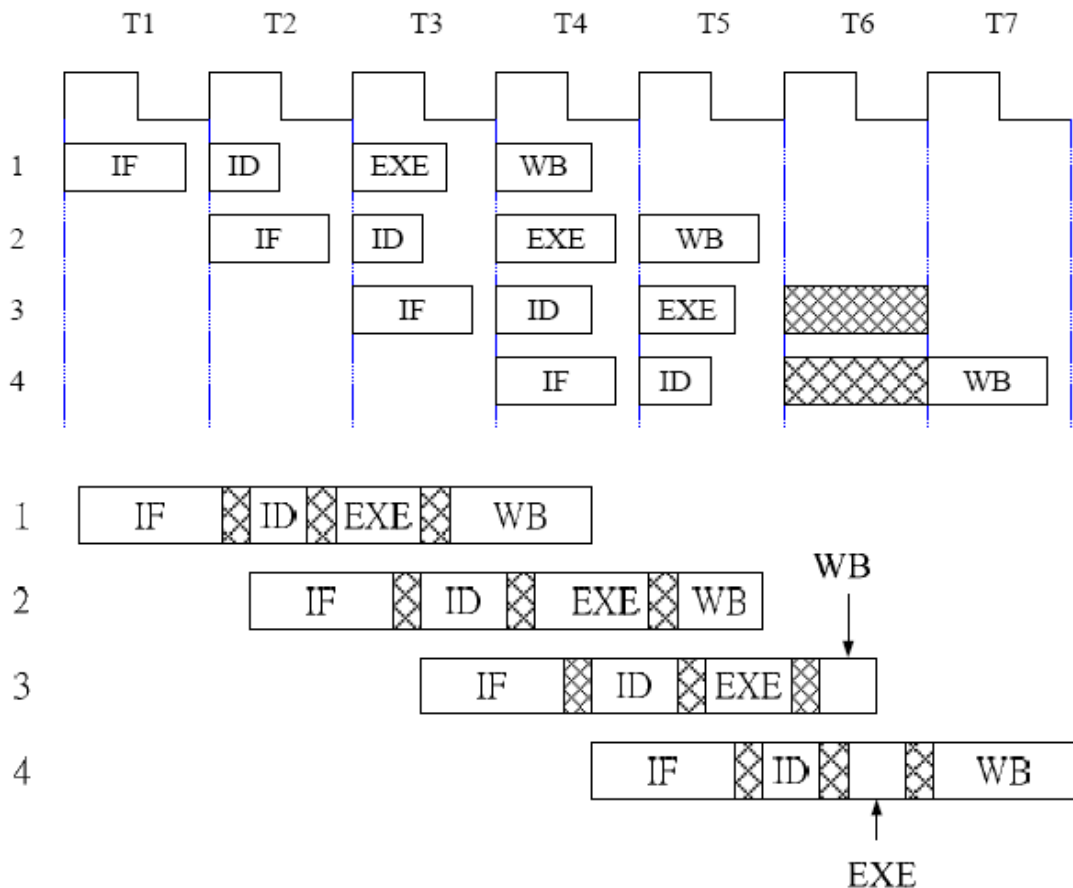


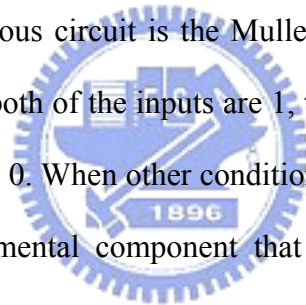
Figure 15. Synchronous Pipeline V.S. Asynchronous Pipeline

2-4 Balsa back-End

The Balsa back-end can be used to generate gate level netlists for supported CAD systems [1] [2]. In this section, we will describe some basic cells for the Xilinx FPGA generated by Balsa and some handshake components in the Balsa synthesis system.

2-4-1 Basic Elements

There are many basic cells generated by Balsa for the Xilinx FPGA, including AND, OR, NOR, XOR, NAND, BUF, XNOR, INV, FD (D-type flip-flop), FDC and FDCE. The most important cell for a asynchronous circuit is the Muller C-element as shown in Figure 16. It can hold the past state. When both of the inputs are 1, the output is set to 1. When both of the inputs are 0, the output is set to 0. When other conditions happen, the output is not changed. A Muller C-element is a fundamental component that is extensively used in asynchronous circuits.



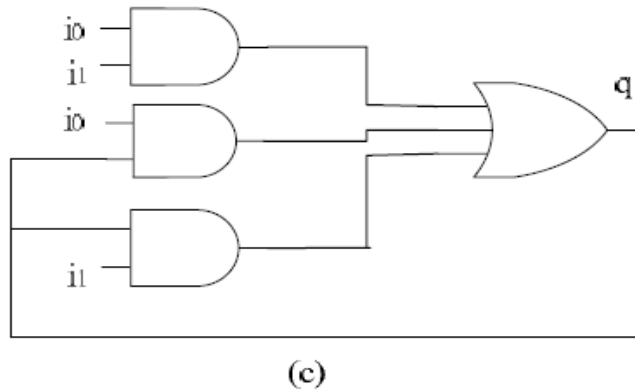
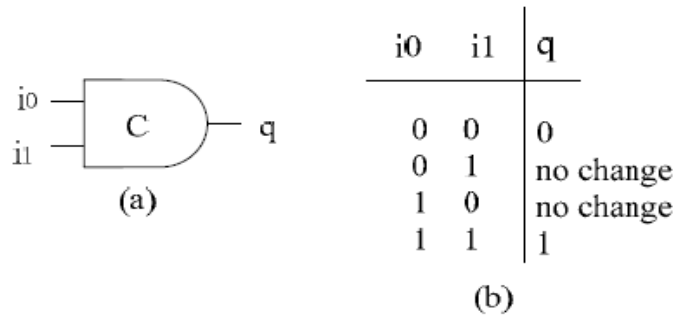


Figure 16: The Muller C-element, (a) symbol (b) true table (c) gate-level implementation

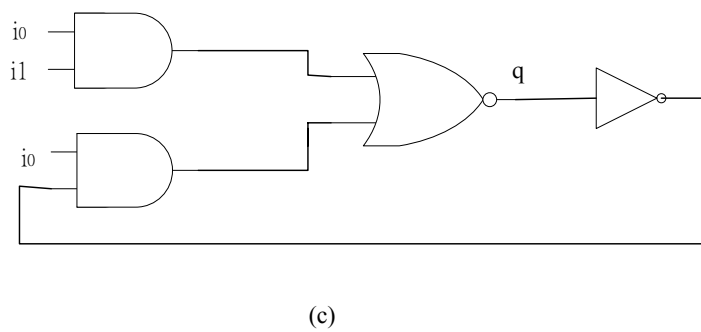
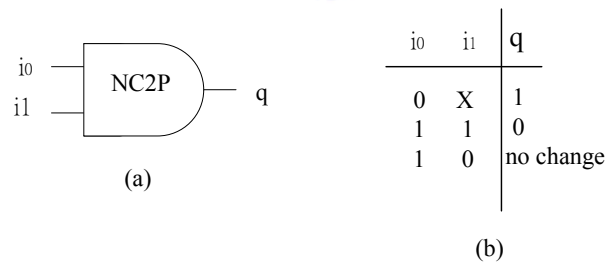
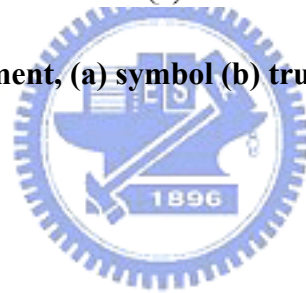


Figure 17: The NC2P-element (a) symbol (b) true table (c) gate-level implementation

Figure 17 shows the NC2P element. When the input, i_0 , is equal to 0, the output, q , is set to 1. When the i_0 and i_1 are both 1, the output is set to 0. For other input conditions, the output is not changed.

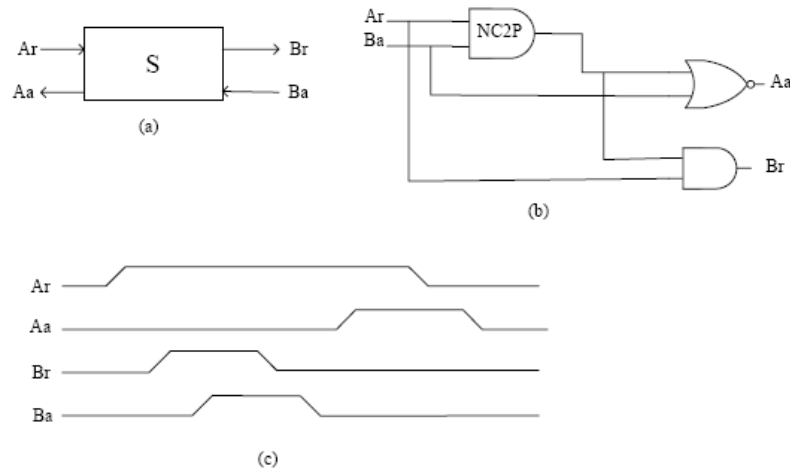


Figure 18: The S-element (a) symbol (b) gate-level implementation (c) handshaking



Figure 18 shows the S-element. An S-element performs a series of handshake. It has 4 inputs including two request/acknowledge handshake pairs, ‘Ar’/’Aa’ and ‘Br’/’Ba’. It is composed by NC2P, NOR and AND gates. In the Balsa system, it usually replaces the “inverter of C element” with a NC2P element, because the behavior of a NC2P element is much like a C element. Hence, it can reduce the number of gates because an “inverter of C element” uses 3 AND gates, 1 OR gate and 1 INV, but a NC2P element uses 2 AND gates, 1 NOR gate and 1 INV.

2-4-2 Handshake Components

There are 40 handshake components in the Balsa system. Each handshake component is constructed by a gate level implementation. In the following section, we will illustrate some of them.

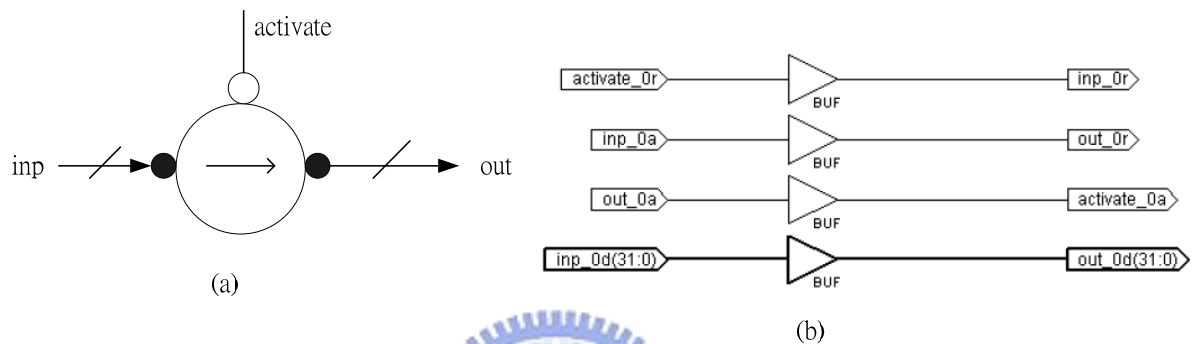
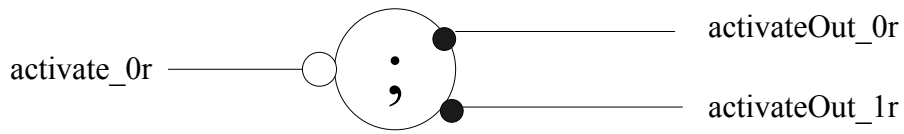


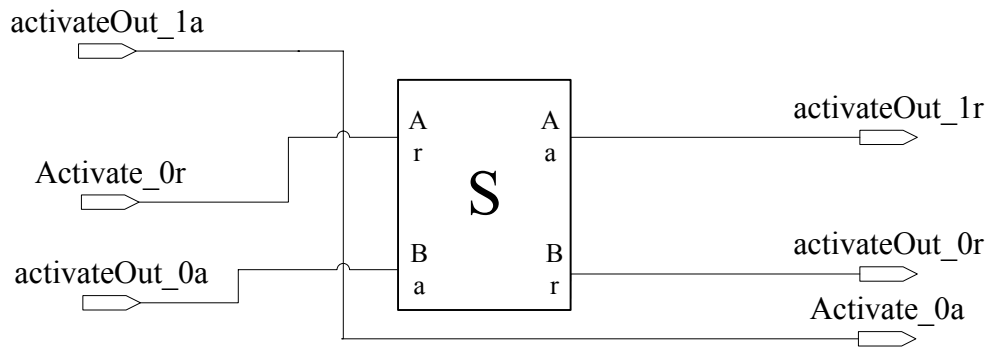
Figure 19: The Fetch Component (a) handshake component (b) gate-level implementation

Figure 19 shows the Fetch component. This component is used to transfer data from input channels to variables, from variables to output channels, and from variables to variables.

Figure 20 and Figure 21 are the sequence and concurrent components. The sequence components control the output signals in sequence, and the concurrent component controls the output signals in parallel.

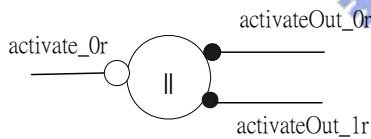


(a)

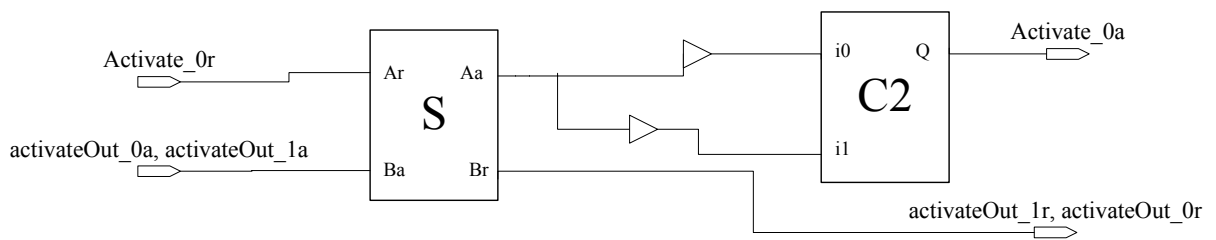


(b)

Figure 20: The Sequence Component (a) handshake component (b) gate-level implementation



(a)



(b)

Figure 21: The Concurrent Component (a) handshake component (b) gate-level implementation

Figure 22 is a variable component which is composed by the FD (D-type flip-flop) gate. The Balsa system will map the “variable “description to this component when translating handshake component files (*.breeze*). Data is stored if the signal `write_or` is set, and data is read when the signal `read_or` is set.

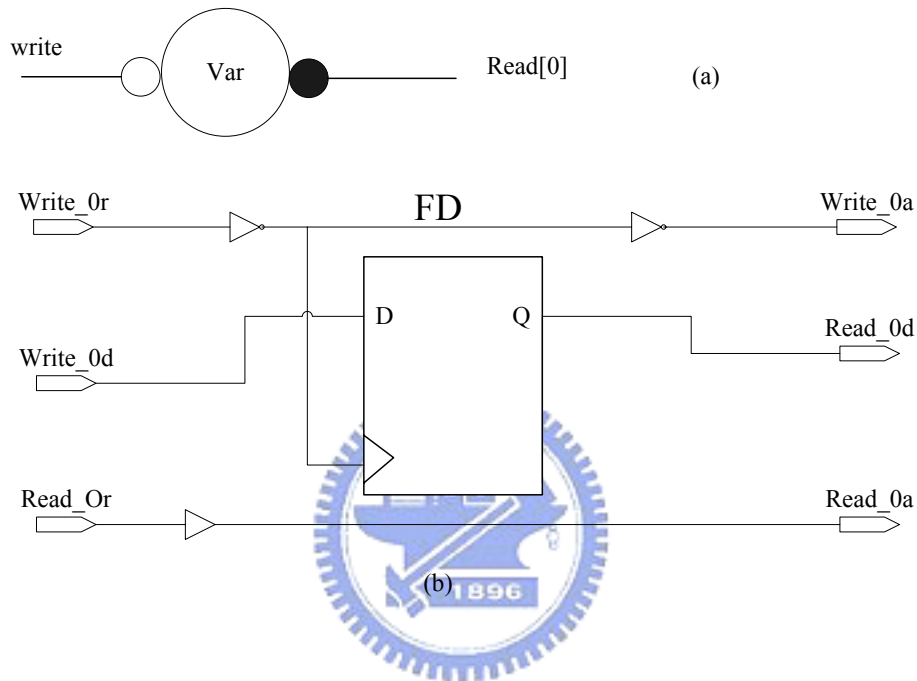


Figure 22: The variable Component (a) handshake component (b) gate-level implementation

2-5 Concluding Remarks

In this chapter, we introduce the MP3 (MPEG1 Layer 3) architecture. The frame structure of a MP3 file contains 5 parts, Header, CRC, Side information, Main data, Ancillary. This process flow of the MP3 operates in the sequence as the above structure. It can be divided into three main parts: the bitstream decoding, the inverse quantization and the frequency-to-time mapping. We then introduce the concepts of asynchronous pipeline. Finally we illustrate the Balsa back-end. Balsa synthesis system is composed of about 40 components, which can be translated into gate-level netlists.



Chapter 3: The Design of the PAMP3

This chapter describes the design of the PAMP3. The PAMP3 consists of eight main parts, the synchronizer&Huffman, the requantizer, the reorder, the anti-alias, the IMDCT, the BUFF, the filterbank and the PCM_out. These eight parts work in parallel with the communication channel connected in between them. In the following section, we will introduce the top view of the PAMP3 in the first, and then describe the eight main components.

3-1 The architecture of the PAMP3

The architecture view of the PAMP3 is shown in Figure 23. All of the operations can be completed in the eight stages, and then the PAMP3 outputs a serial of PCM data. The synchronizer&Huffman stage takes the MP3 music data from the Main Memory and puts the header, the side information and the main data into the buffers. The buffers are used as the source and the information of the decoding scale factors and the decoding Huffman data. The requantizer stage is responsible for decoding inverse quantization. It converts the Huffman decoded values back to their spectral values using a power law. The reorder stage and anti alias stage reorders the frequency value from the MDCT and the quantization, and reduces the aliasing effects of the poly-phase synthesis filter bank during its encoding process. The IMDCT stage and the filterbank stage decode the IMDCT and the poly-phase synthesis but here they are implemented in a different way. The BUFF stage holds the data until all of the samples are completed, then the BUFF stage outputs data to the next stage. The PCM_out stage checks the channel mode before outputting data.

PAMP3 decoder

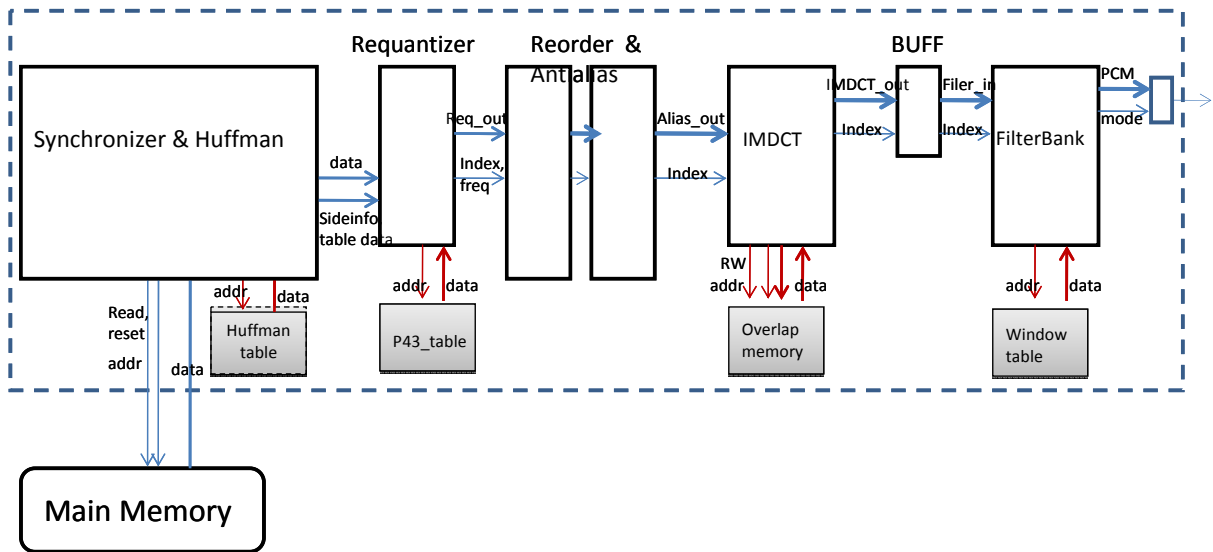


Figure 23: The architecture view of the PAMP3

The following code is the top level process of the PAMP3. Every component is connected by internal channels.

```
procedure pamp3_decoder(
```

```
    input mem_out : 64 bits;
```

```
    input mem_boundary : 20 bits;
```

```
    output mem_reset : bit;
```

```
    output mem_addr : 20 bits;
```

```
    output data_out : 16 bits
```

```
) is
```

```
.....
```

```
scale_huffman_top(mem_out, mem_boundary, mem_reset, mem_addr,
```

```
    index, freq1, nonzero, global_gain, subgain, scale,
```

```
    preflag, scale_l, scale_s, t_l, t_s, data) ||
```



```

requantization(index,freq1,global_gain,subgain,scale,preflag,nonzero,
          data,scale_l,scale_s,t_s,t_l,req_index_out,freq2,req_out) ||

reorder(req_index_out,freq2,req_out,reorder_out,order_index) ||

alias_reduction(order_index,reorder_out,alias_out,index_out) ||

imdct_top(index_out,alias_out,imdct_out,ch_in) ||

imdct_filterbank(ch_in,imdct_out,filterbank_in,ch_out) ||

filterbank_top(ch_out,filterbank_in,pcm_in,pcm_ch) ||

pcm_out(pcm_ch,pcm_in,data_out)

end

```

3-2 The synchronizer&Huffman stage



This stage contains three buffers and three modules: the synchronizer, the SCALE&HUFFMAN and the BUFF_RW_ARBITOR. The three buffers are the header buffer, the side information buffer and the main data buffer. The synchronizer module retrieves the data from the main memory and decodes the header data and side information data into two buffers. After decoding the header and the side information, the synchronizer sends the side information data and some header data to the SCALE&HUFFMAN module. Then, the synchronizer writes the main data fetched from the main memory into the main data buffer. While the synchronizer is writing the data, the SCALE&HUFFMAN module also reads the data from the main data buffer to decode the scale factors and the Huffman data. Therefore, the BUFF_RW_ARBITOR controller arbitrates the two control signals that are read and write, from the synchronizer and the SCALE&HUFFMAN modules at the same time, and checks whether the buffer data is valid. The SCALE&HUFFMAN module does the scale factor

decoding and the Huffman decoding. We use the direct table lookup method for Huffman decoding. All the data of the Huffman tables are stored in the ROM, which can be read by the SCALE&HUFFMAN module.

The Figure 24 shows the modules of the Synchronizer&HUFFMAN stage. After the SCALE&HUFFMAN module decodes one value (13 bits) from the data, it immediately transfers the value to the next stage. This is convenient because the next stage does not need the SCALE&HUFFMAN module to decode the entire 576 values before processing.

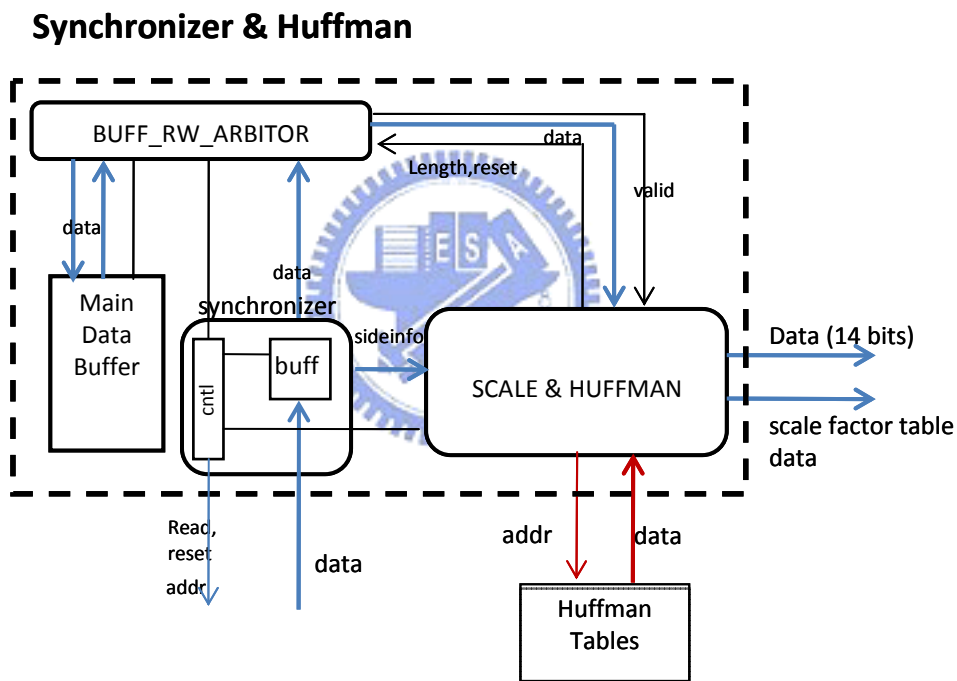


Figure 24: The Synchronizer&HUFFMAN stage

3-3 The re-quantizer stage

The re-quantizer stage (Figure 25) contains four modules: requant_ctrl, fras_1, fras_s and fras. The requant_ctrl module controls the other modules to compute the output data. The

fras_l module and fras_s module mainly calculate the right side of the multiply sign in eq(1) according to the conditions of long block and short block. The fras module mainly calculates the right side of the equation in eq(1) using the input data, “ISi”, and value, “a”, that were calculated before. Because the value of the 4/3 power calculation is very difficult, a ROM is designed to store all the value of the 4/3 power for future usage. Finally, the requant_ctrl module outputs a value whenever the fras module completes its calculation.

The input data of the requantizer stage, ISi, is 14-bits data is used in the process of looking up the table. After decoding the input, the output data of the requantizer stage, req_out, is a 32-bits data. The format of this output data is represented by a integer of 4 bits and a decimal of 28 bits.

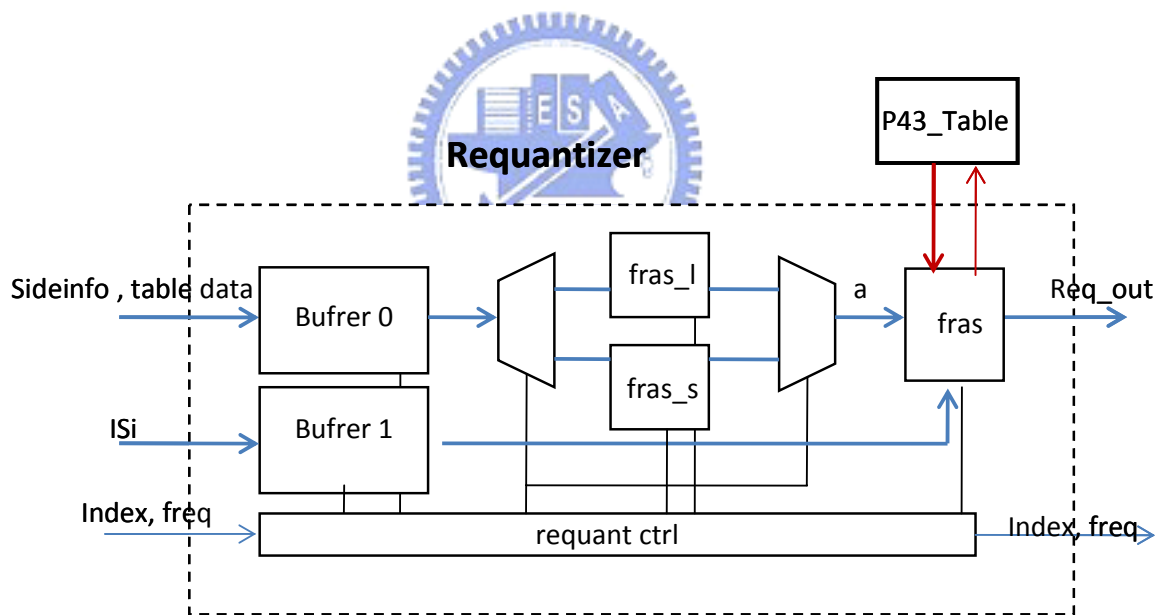


Figure 25: the Requantizer stage

3-4 The reorder stage and anti-alias stage

The reorder stage (Figure 26) immediately assigns the correct position of the input data

according to the frequency mode of the side information. The `order_ctrl` controls the receiving and the output of a buffer (576 x 32 bits). The output methods are according to the block type of the current frame. In the long block type, the output value can be sent out directly. Because the anti-alias stage performs eight butterfly multiplications for every two subbands (2 x 18 values). In the short and mixed block type, a counter is set to count the received data of current frame, and the reordered outputs are sent out when the counter equals to a specified number. The specified number means that 18 reordered values have been received. (Only support for MP3 streams with 44.1kHz sample frequency is implemented.)

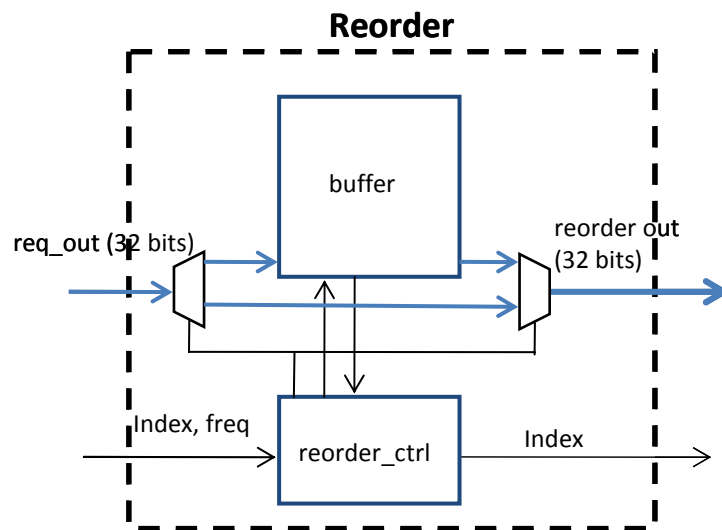


Figure 26: The reorder stage.

The anti-alias stage (Figure 27) uses two register banks (2*18*32 bits) to store two subbands from the reorder stage and performs an 8-butterfly multiplication as shown in Figure 28. After doing the 8 butterfly multiplication, the anti-alias stage outputs the first half of the results (18*32 bits) to the next stage. Then, the anti-alias stage moves the remaining half of the results forward to one of the previous register banks while waiting for the new 18 reordered values from the previous stage.

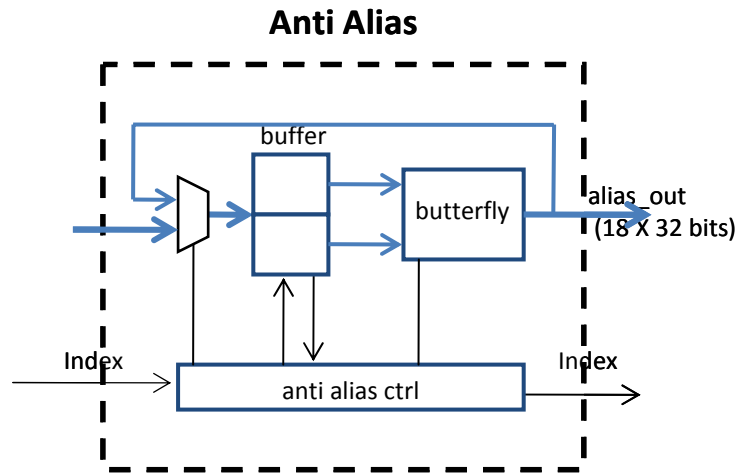


Figure 27: The anti-alias stage.

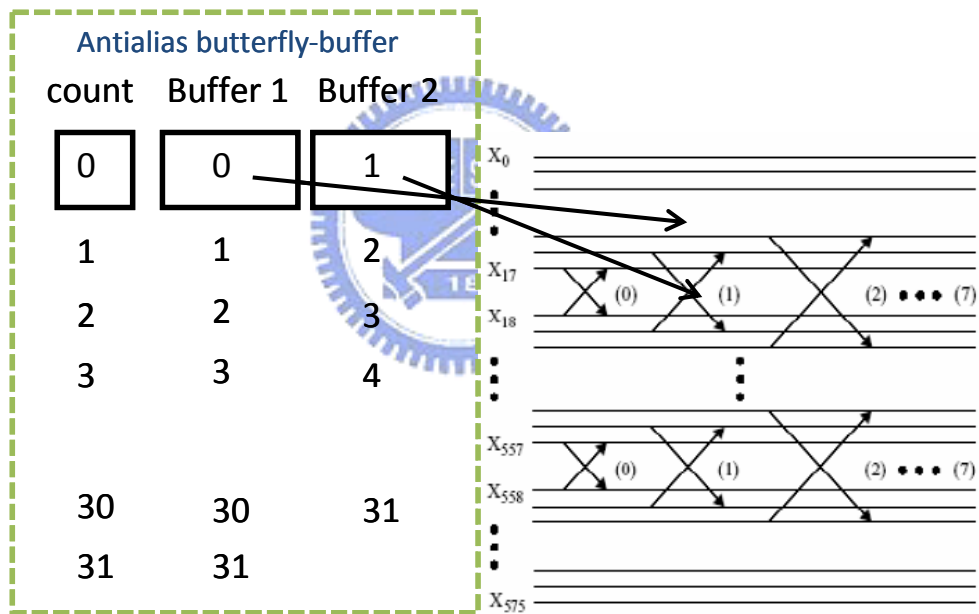


Figure 28: The register banks of the anti-alias stage.

3-5 The IMDCT stage

The IMDCT stage does the inverse modified discrete cosine transform, the windowing and the overlapping processes. In the IMDCT processing, we use the method created by

Szu-Wei Lee [16] to implement. Figure 29 shows the computational flow of Szu-Wei Lee's algorithm. The N -point inverse MDCT is converted to a $N/2$ -point DCT-IV first, then it is converted to a $N/2$ -point SDCT-II. Finally, a $N/2$ -point SDCT-II can be divided to two identical $N/4$ -point SDCT-IIs. Therefore, this algorithm can be simplified into 3-point and 9-point SDCT-II modules, which compute the inverse MDCT for a MPEG layer III. In this algorithm, the total of the multiplications and the additions are only 43 and 115 when the length $N = 36$.

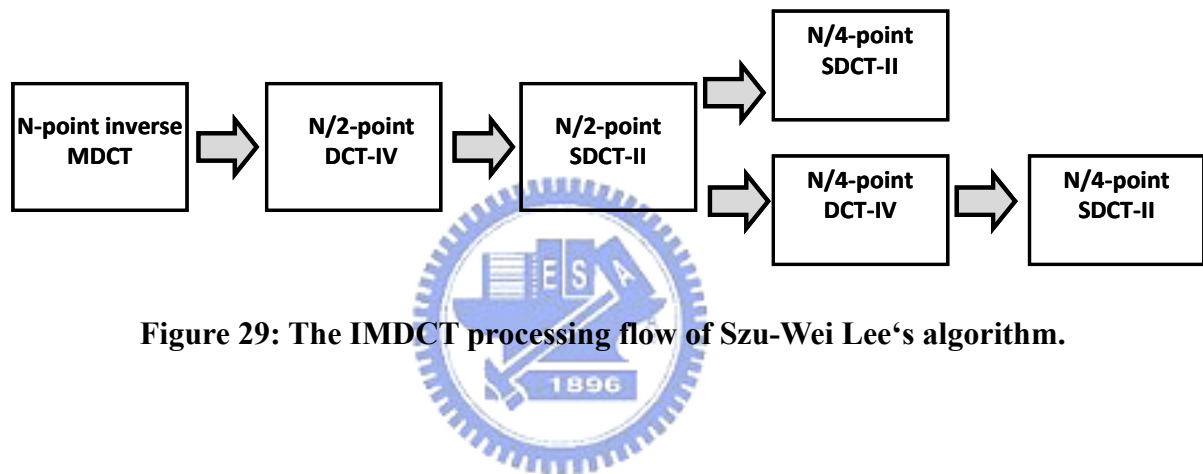


Figure 29: The IMDCT processing flow of Szu-Wei Lee's algorithm.

According to the previous algorithm, 5 sub-stages were constructed for the pipeline architecture in the IMDCT stage as shown in Figure 30. These sub-stages are scaling&butterfly, SDCT-II, post-process, windowing and overlapping. The first three sub-stages are the computing flow in the Szu-Wei Lee's algorithm. The others execute multiplication between the inputs and the long or short window table data and then overlap between the inputs and the previous frame. The input values from the anti-alias stage are multiplied by constants and then they pass through a butterfly addition with each other in the scaling&butterfly sub-stage. The SDCT-II sub-stage is decomposed into two blocks, the $N/4$ -point SDCT-II and the $N/4$ -point DCT-IV. The first half of the outputs from the previous sub-stage performs the $N/4$ -point SDCT-II immediately. The second half implement the reordering process first and then perform the $N/4$ -point SDCT-II. The 3-point and 9-point

SDCT-II can be used directly. The 3-point SDCT-II requires one multiplication and 5 additions, and the 9-point SDCT-II requires 8 multiplications and 36 additions. After the post-processing sub-stage, 36 outputs of the IMDCT processing will be created. In the windowing sub-stage, the windowing_ctrl controller controls the multiplying process (mult_long_short) between the data of the window table and the input from the last sub-stage. The data will differ according to long or short block types. Finally, the overlap sub-stage performs the overlapping between half of the data from the current block and the data from the previous block in the overlapping memory and then outputs 18 overlapped data to the next stage.

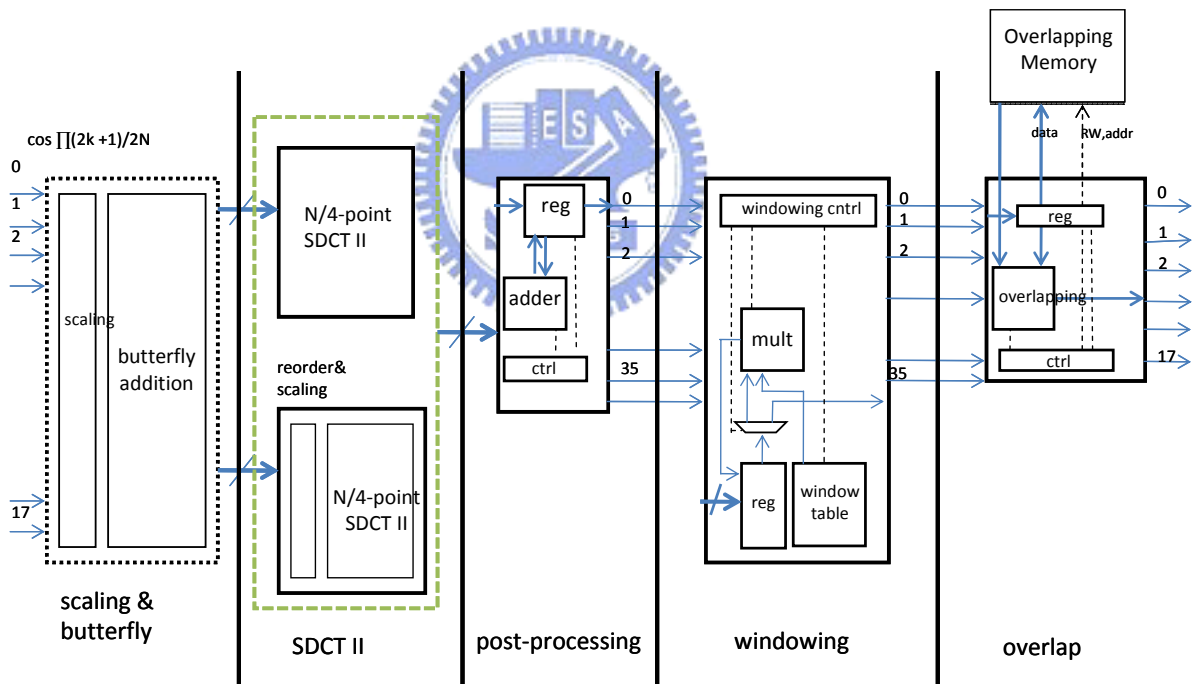


Figure 30: the sub-pipeline of the IMDCT stage

The following code is the top level process of the IMDCT stage. The whole IMDCT stage is decomposed into 5 sub-stages of the sub-pipeline.

```

procedure imdct_top(

    input index : 10 bits ;

    array 0 .. S-1 of input data_in : data_type ;

    array 0 .. S-1 of output data_out : data_type ;

    output ch_out : 3 bits

) is

... ..

imdct_stage1(index,data_in,reg1,index1) ||

imdct_stage2(index1,reg1,reg2,index2) ||

imdct_stage3(index2,reg2,reg3,index3) ||

IMDCT_windowing(index3,reg3,index_out,imdct_over) ||

IMDCT_overlap(index_out,imdct_over,data_out,ch_out)

end

```



3-6 The BUFF stage

The outputs of the IMDCT stage are 18 time-domain samples, but the inputs of the poly-phase filterbank stage are 32 subband samples. The BUFF stage (Figure 31) is needed to buffer the inputs from the IMDCT stage until receiving the 576 samples. Then, the output of the BUFF stage delivers 32 subband samples in the buffer to the poly-phase filterbank stage. The pipeline architecture must work abidingly during the data buffering. Therefore, the buffer is divided into two blocks and the two blocks are read and written in turn. During the buffer0 is being written, data is from the output of the IMDCT stage, and the buffer1 is being read

data to output to the filterbank stage; the process continues until receiving the 576 samples. In the next 576 samples, the buffer0 is being read data to output to the filterbank stage and the buffer1 is being written data from the output of the IMDCT stage, and so on.

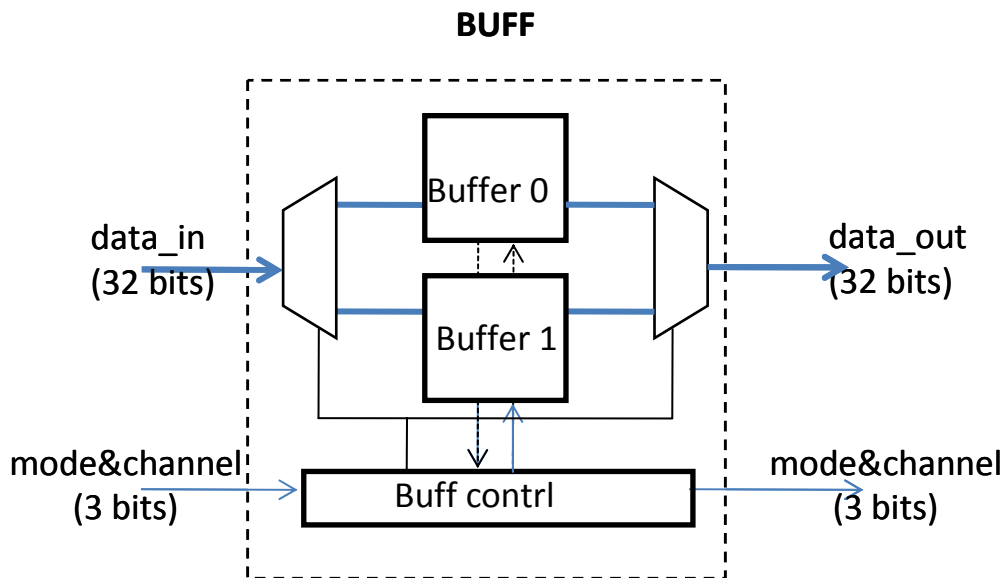


Figure 31: The BUFF stage

3-7 The poly-phase filterbank stage

The poly-phase filterbank converts the time domain samples from the IMDCT transform in each subband to PCM samples. As mentioned in the previous chapter, the poly-phase synthesis filterbank can be decomposed into four parts, moving, DCT, matrix multiply and overall adding. The Konstantinides' algorithm [9] and the B.G. Lee's algorithm [4] are both used to find a good implementation of the 32-point DCT. The 32 subband samples are the inputs of the DCT and then they are converted by the B.G. Lee's algorithm. Finally, by using a symmetric way as shown in the Konstantinides' algorithm (Figure 32), the previous 32-point results become 64-point final results. Figure 33 shows the 8-point DCT using B.G. Lee's

algorithm [11].

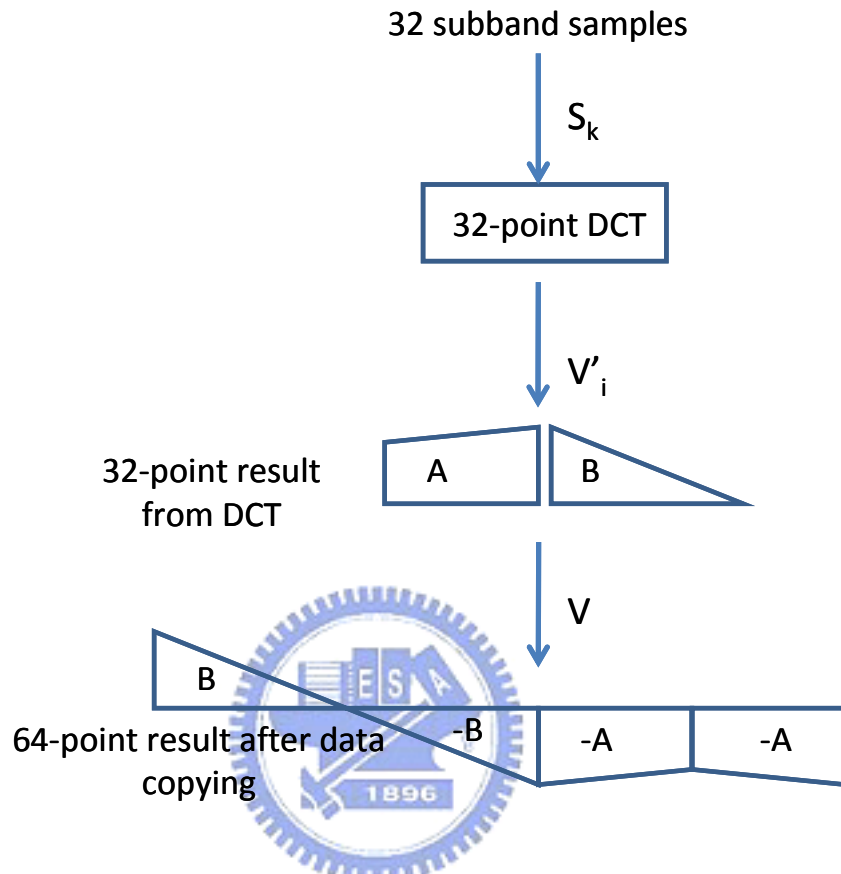


Figure 32: The DCT simplification of Konstantinides' algorithm

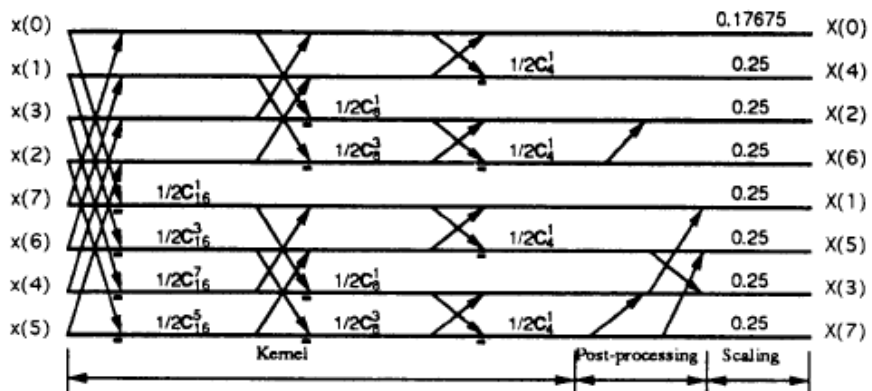


Figure 33: The 8-point DCT simplification of B.G. Lee's algorithm [11]

This pipeline stage is divided into 7 sub-stages as shown in Figure 34. The first 6 sub-stages perform the 32-point DCT by the previous method. The remaining 1 sub-stage performs the FIFO moving, window table multiplying and final scaling. The B.G. Lee's fast DCT algorithm is recursive, and for a 32-point DCT. It requires only 80 multiplications and 209 additions. Therefore, the first 5 sub-stages are recursively integrating the 32-point subband samples into smaller units. Then the next sub-stage performs the similar post-processing part as shown in the Figure 34. The windowing_ctrl controller in the windowing sub-stage controls the inputs from the previous sub-stage to do data copying as shown in Figure 33. Then the controller pushes the inputs into the FIFO and performs multiplication between the data from the FIFO and the constants from the window table ROM. After overall adding, the windowing_ctrl outputs the 32 PCM data to perform scaling. Finally, the windowing and scaling sub-stage outputs 32 scaled PCM data to the next stage.

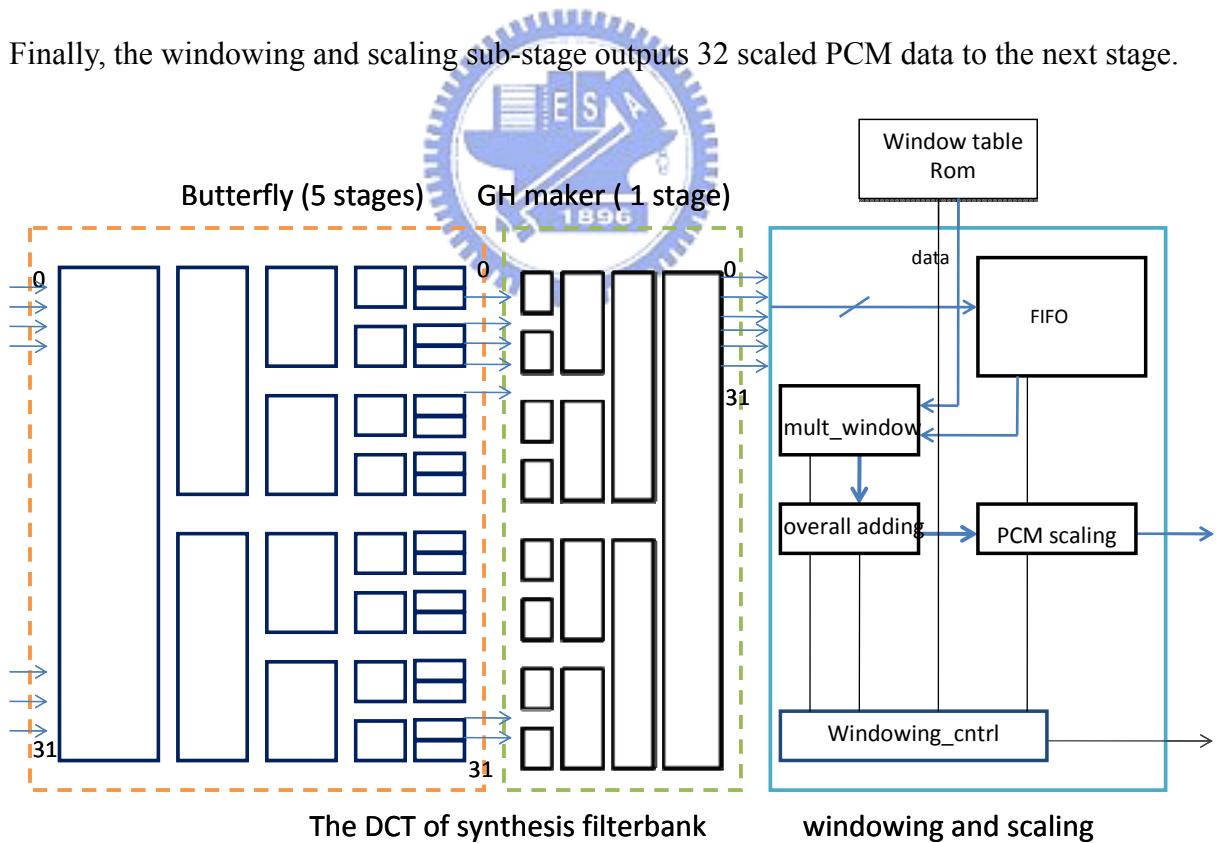


Figure 34: the sub-pipeline of the synthesis filterbank stage

The following code is the top level process of the poly-phase filterbank stage.

```
procedure filterbank_top(  
  
    input ch : 3 bits ;  
  
    input data_in : data_type ;  
  
    output data_out : 16 bits;  
  
    output ch_out : 3 bits  
  
    ) is  
  
    .....  
  
    filterbank_butterfly_5(ch,data_in,reg1,ch1) ||  
    filterbank_butterfly_4(ch1,reg1,reg2,ch2) ||  
    filterbank_butterfly_3(ch2,reg2,reg3,ch3) ||  
    filterbank_butterfly_2(ch3,reg3,reg4,ch4) ||  
  
    filterbank_butterfly_1(ch4,reg4,reg5,ch5) ||  
  
    filterbank_GHmake(ch5,reg5,reg6,ch6) ||  
  
    filterbank_windowing(reg6, ch6, data_out,ch_out)  
  
end
```



3-8 The PCM_out stage

Because the MP3 decoding can be divided into a mono channel mode or a dual channel mode, the output of the PCM_out stage is a 16-bits PCM data according to the mode bits transferred from the previous stage. When the mode bits are equal to 0 or 2, this stage must

store the entire 576 samples of one channel until the data of the other channel is received and then the stage outputs the data by turns of channel 0 and channel 1. When the mode bits are equal to 3, this stage inputs and then outputs directly. When the mode bits equal to 1, it means the mp3 music compressed in the joint-stereo mode. In this article, the joint-stereo decoding is not discussed because it won't be really implemented.



Chapter 4: Implementation and Verification

This chapter contains three parts. First, our design flow for the asynchronous implementation on FPGA is illustrated and some implementation issues are described. Finally, the behavior simulation is illustrated.

4-1 The Design Flow

The PAMP3 core is modeled with the Balsa language, and then it is compiled into a network of handshake components (*.breeze file*) by the *balsa-c* compiler. Each of these components has one mapping gate-level implementation. Using the “*balsa-netlist*” tool can generate the gate-level implementations in the Verilog for the Xilinx or other target synthesis tools. And the *balsa* provides a Verilog simulation tool, *balsa-verilog-sim*. It supports some open sources or commercial Verilog simulators: Icarus Verilog, Synopsys VCS, Cadence NC-Verilog, Cadence Verilog-XL, Model Technologies Modelsim and Cver.

The *balsa-verilog-make-builtin-lib* tool can generate the *balsa* built-in functions into a library and register the library in the specified Verilog simulator. Then the gate-level simulation can be performed to verify the PAMP3 using the Balsa tool.

Figure 35 describes the FPGA design flow of Balsa. The Verilog netlist generated by the *balsa-netlist* is converted into a netlist of basic gates in the synthesis of the design flow. However, the synthesis tool may optimize the hazard-free circuits and buffers generated by the *balsa-netlist*. The constraint “keep hierarchy” or “syn_hier” is added to avoid the logic minimization. The “keep hierarchy” is used for the Xilinx synthesis tool, and the “syn_hier” is used for the Synplify synthesis tool. Then the synthesized netlists are mapped to the target device using a technology-mapping algorithm. The placement and the routing algorithm, map the logic blocks from the netlists to the physical locations on an FPGA, and determine how to

interconnect the logic blocks using the available routing. The final output of the design flow is the FPGA programming file.

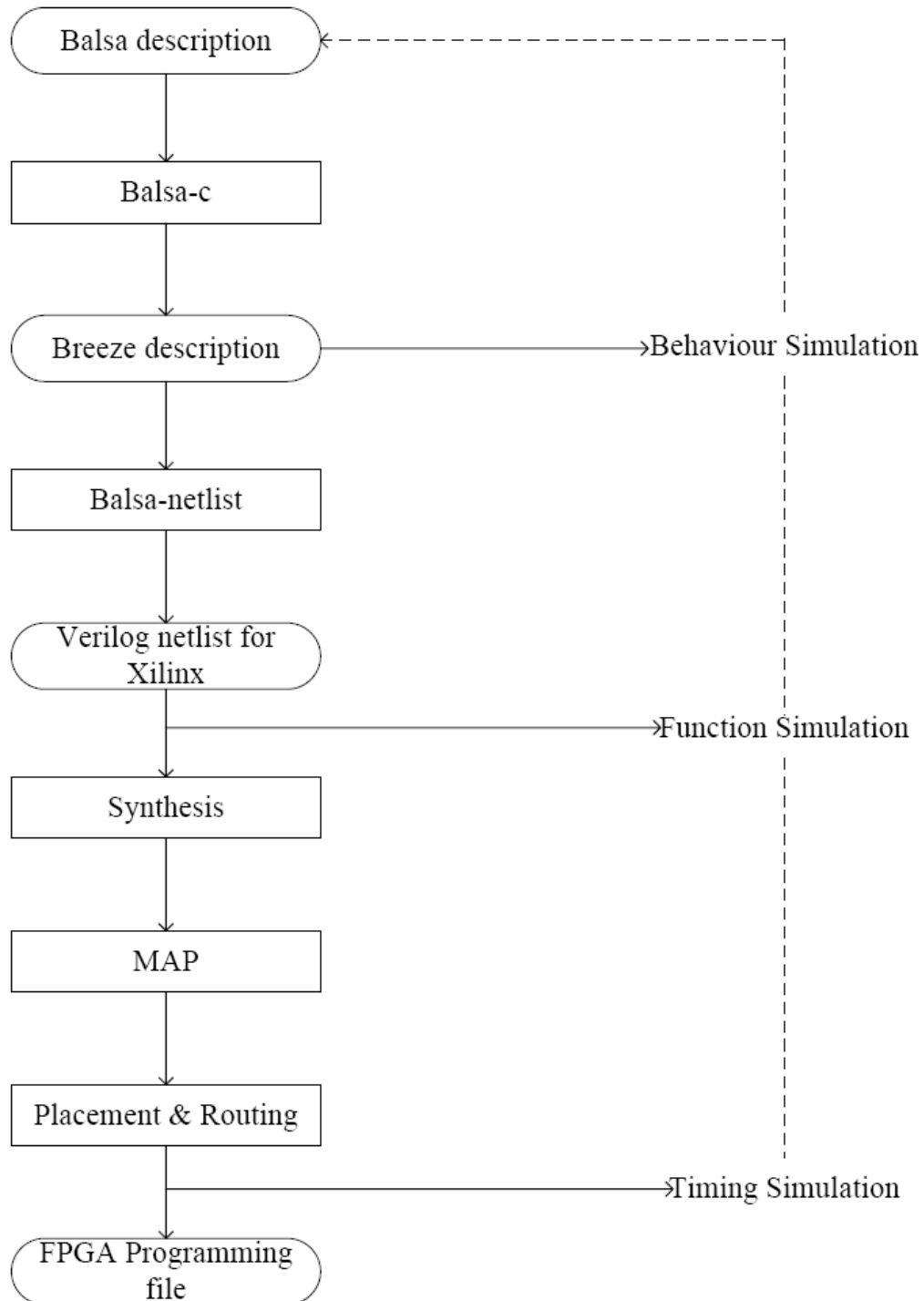


Figure 35: The Balsa and FPGA design flow

4-2 Implementation Issues

Four-phase bundle-data protocol is chosen to implement the handshake circuit instead of the dual-rail encoding in order to reduce the area cost. The Balsa provides some technologies for implementations. The circuits are implemented using the Xilinx standard cells: AND, OR, INV and FD when choosing the Xilinx ISE technology. Other target synthesis tools must choose the “example” technology, and the basic cell with the standard cell in the target synthesis tool needs to be modified.

The synthesis tools could perform the logic minimization during synthesizing. It would break some buffers or redundant circuits that are used to ensure hazard-free. These buffers and circuits can't be minimized directly. Therefore, the constraint “keep hierarchy” is added to avoid the logic minimization when using the Xilinx synthesis tool.

The Balsa RAMs and the ROMs are only modeled to perform the Balsa behavior simulation. They can't be implemented in the Verilog, because they are built-in functions written in C. Therefore, the block RAM on the FPGA should be used in the FPGA implementation.

4-3 Verification

The behavior simulation for the PAMP3 (pipelined asynchronous MP3) can be performed in the Balsa and the Modelsim. The simulation environment is shown in Figure 36. The outputs from PAMP3 are written into a file by using a Balsa builtin function, and the output file (PCM format) can be played by audio softwares such as GoldWave as shown in Figure 37. An open source software MP3 decoder, AMP mp3 decoder, and a simple decoder that are written by us, OfMP3 (Observer for MP3 decoder), are used to compare the output of

each stage with the PAMP3 as shown in Figure 38. If the results are not equal, the design needs to be remodified. If the results are equal, the design is correct. The memory model is the predefined procedure in the Balsa as shown in Figure 39. The total size of the memory is 8M bytes. The MP3 data are loaded from a binary file during the initialization process. Whenever an addressing arrives at the memory model from the memory address channel, the memory outputs the data. When the PAMP3 core is writing data, it sets the signal rNw and sends out the address and the data.

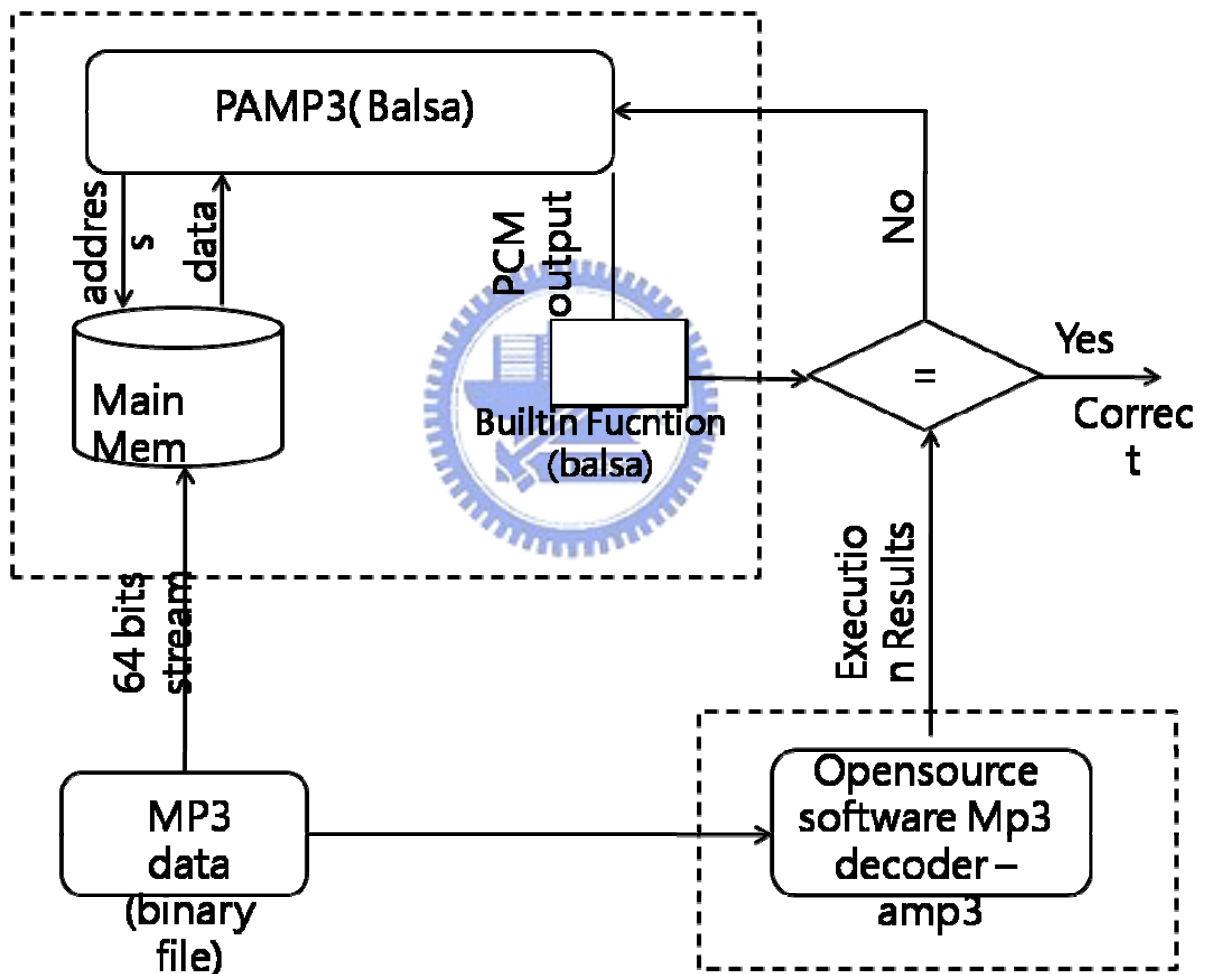


Figure 36: PAMP3 behavior simulation enviroment

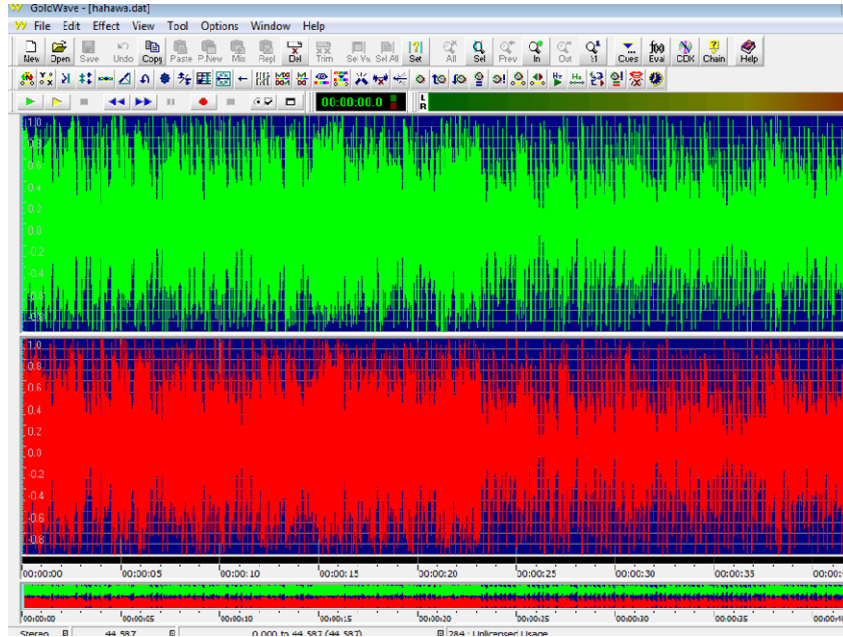


Figure 37: Playing output in the GoldWave

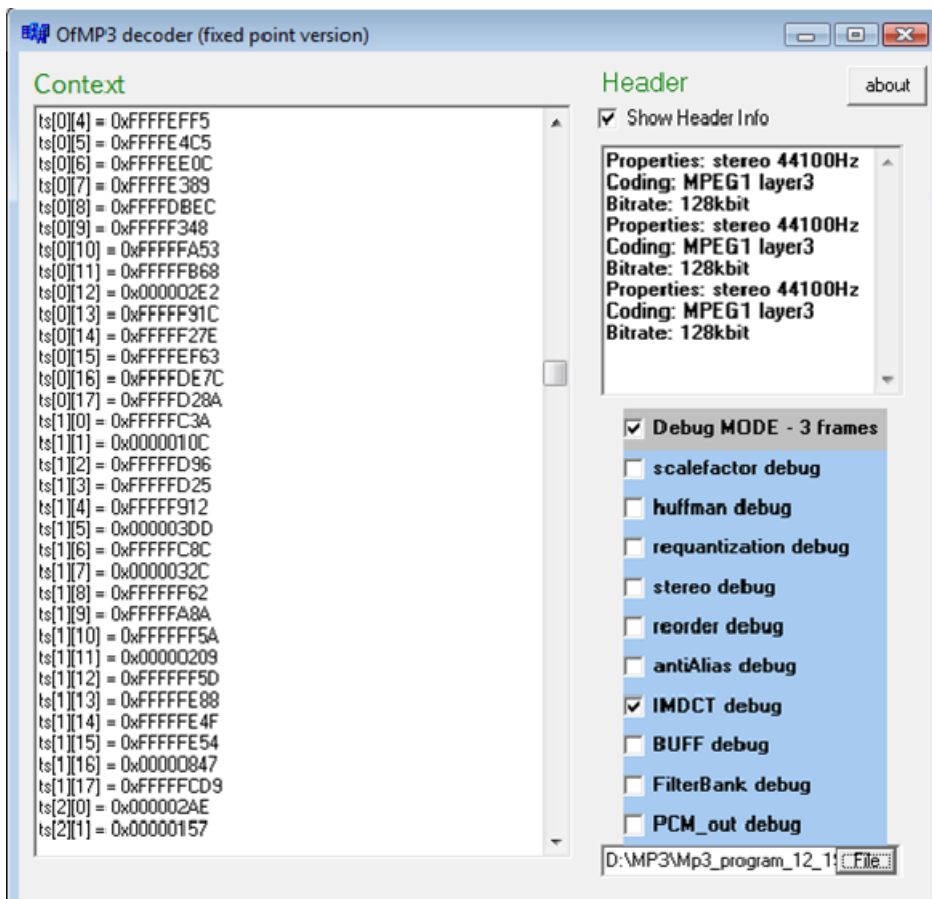


Figure 38: Observer for MP3 decoder

```
BalsaMemoryRAM( { 20,    --addr width  
                  64 },  --data width  
                ← BalsaMemoryNew(), --direct expression to port connection  
                RAM_addr, RAM_rNw, RAM_write_data, RAM_read_data)
```

Figure 39: The Balsa memory model



Chapter 5: The Results

5-1 Simulation Result

In the Balsa simulation, all kinds of the channel modes and the block types have been simulated using the verification method as Figure 37. Because the simulation in the Balsa system costs much time, the specified characteristic test files are simulated first such as short block, long block, mixed block, single channel and dual channel. Then, we used some MP3 music with fewer frames to confirm the entire PAMP3 correctness. Finally, a MP3 music with more than 1000 frames is used to simulate the performance of the PAMP3.

5-2 Area cost



In the VLSI design flow, we used the Synopsys Design Compiler to synthesis our design with TSMC 0.13 μm process. Table 3 shows the cell area of the two kinds of multipliers, array adder algorithm and booth algorithm. Multipliers are the main computation components in the MP3 processing. Therefore, we compared how the two different kinds of multipliers affected the area cost of MP3 decoder. The total area cost with booth multipliers is as shown in Table 4. The filterbank stage and the IMDCT stage are the most dominant stages of the whole design.

Multiplier	cell area
Array adder	33577.964
Booth	21996.607

Table 3: The Cell Area Cost of two kinds of multipliers (μm^2)

Different synthesis tools were used, Xilinx XST and Synplificities Synplify Pro during the FPGA implementation. Only the Synplify Pro was able to synthesize the project successfully because of some problems of the Xilinx XST. Table 5 shows the slice and the gate count of each stage.

stages	Total cell area	
Synchronizer&Huffman	2139331.75	17.4%
Requantizer	409842.31	3.3%
Reorder	476468.65	3.9%
Anti-alias	1080126.87	8.8%
IMDCT(with Buff)	3630446.63	29.5%
FilterBank (with PCM_out)	4563784.11	37.1%
Total	12300000.32	100.0%

Table 4: The Cell Area Cost of Every Part of PAMP3 (μm^2)

stages	Slice	Gate count	
Synchronizer&Huffman	45,903	607,630	19.4%
Requantizer	6,244	98,947	3.2%
Reorder	11,685	157,309	5.0%
Anti-alias	16,758	255,366	8.1%
IMDCT(with Buff)	46,743	877,597	28.0%
FilterBank (with PCM_out)	72,045	1,138,104	36.3%
Total	199,378	3,134,953	100.0%

Table 5: The FPGA Cost of Every Part of PAMP3

The area overhead mainly comes from the handshake circuit in each handshake component. During handshake components translation, the networks of the handshake components are created by one-to-one mapping without any simplifications. In the complex

circuits, the connections between these networks cost very much. The circuits of the completion detection on the control path need large C elements. Numbers of registers are used during computation and save the frame information during the process of PAMP3. In Balsa system, the registers need more handshake components to be implemented. For example, every bit of all the ports need complete detection in the CallMux implementation that multiplexes a writing port into registers as shown in Figure 40. The three 10-bit ports module was very cost-consuming and in a MP3 processing, the module would use more ports and be more complex. There are two main reasons that the filterbank stage is the most dominant stage of the whole design. First, the filterbank stage is divided into more sub-stages than the IMDCT stage. Second, it controls the bigger FIFO buffer to perform multiplication and overall addition.

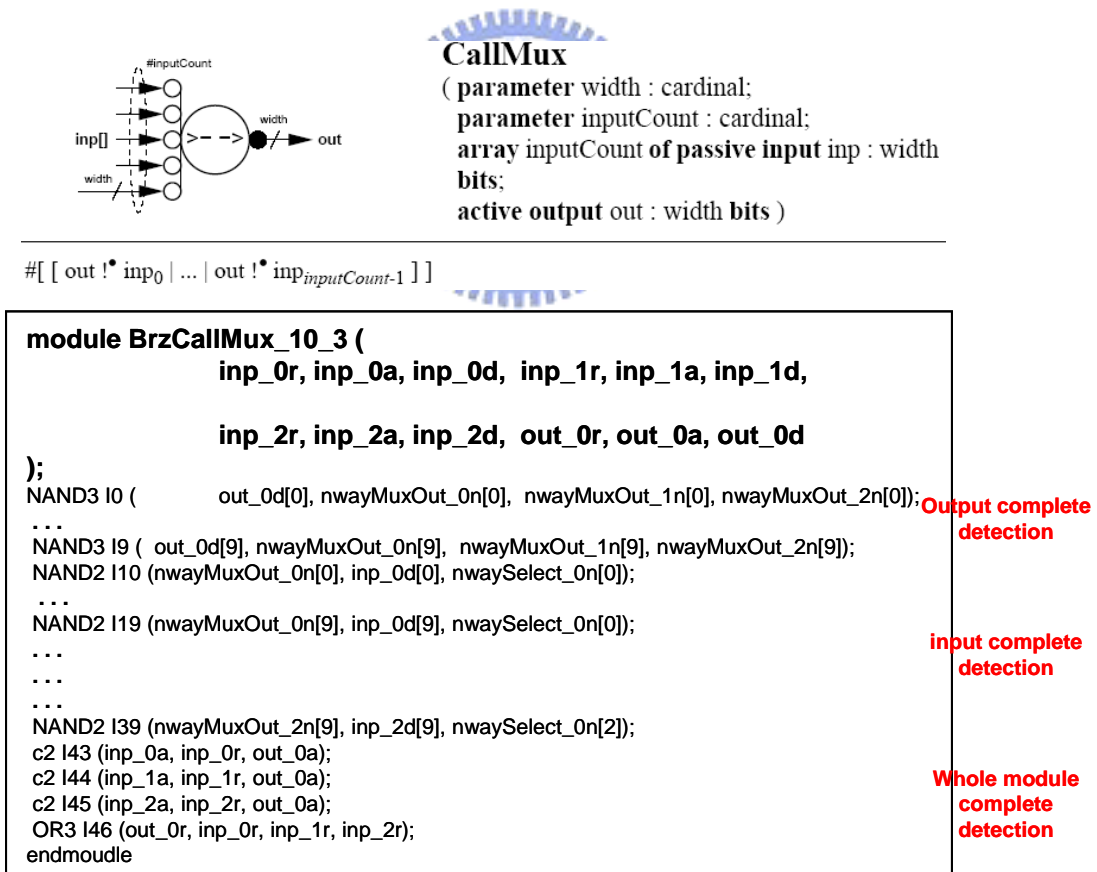


Figure 40: The CallMux handshake component (three 10-bit ports)

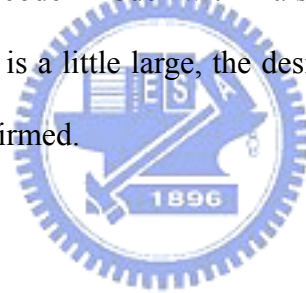
There are timing issues in the Xilinx back-end with certain combinations of handshake components. The Advanced Processor Technologies Group will address these problems and fix them in the future release. Therefore, synthesis is accomplished now, and the Balsa model of PAMP3 can be used directly when the next version of Balsa is released.



Chapter 6: Conclusions

In this thesis, we designed a pipelined architecture in the asynchronous MP3 decoder in Balsa, which is an asynchronous design language and synthesis tool. We used an OfMP3 decoder to verify the output of each stage of the PAMP3. And the final outputs of the PAMP3 can be played by the audio software. All functions can be executed correctly except the Joint-Stereo channel mode. We used the Synopsys Design Compiler with TSMC 0.13 μ m process and the Xilinx ISE to synthesis our design. There are timing issues in the Xilinx back-end with certain combinations of handshake components. The Advanced Processor Technologies Group will address these problems and fix them in the future release.

We implement a MP3 decoder model with Balsa in ways of an asynchronous circuit design. Although the area cost is a little large, the design flow of using CAD tools to design an asynchronous circuit is confirmed.



Reference

- [1] A. Bardsley, D. A. Edwards, "*The Balsa Asynchronous Circuit Synthesis System*," University of Manchester, 2000
- [2] A . Bardsley, "*Implementing Balsa Handshake Circuits*," University of Manchester, 2000
- [3] A.J. Martin et al., "The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller," Proc. 9th Int'l Symp. Asynchronous Circuits and Systems (ASYNC 03), IEEE CS Press, 2003, pp. 14-23
- [4] B.G. Lee, "A new algorithm to compute the discrete Cosine transform," IEEE Trans.ASSP-32, (1984), 1240-1245.
- [5] Doug Edwards, Andrew Bardsley, Lilian Janin & Will Toms, "*Balsa: A Tutorial Guide*," 2004
- [6] ISO/IEC 11172-3 "*Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3*", 1993
- [7] Jens Spars, S. Furber, "*Principle of Asynchronous Circuit Design*," 2001
- [8] J. V. Wood et.al, "*AMULET1: An Asynchronous ARM Microprocessor*," IEEE Transactions on Computers, Volume 46, Issu 4, April 1997 Page(s): 385 – 398
- [9] Konstantinos Konstantinides, "*Fast Subband Filtering in MPEG Audio Coding*," IEEE Signal Processing Letters , Vol . 1 , No . 2 , February 1994 .
- [10] Krister Lagerström, "*Design and Implementation of an MPEG-1 Layer III Audio Decoder*", Master's Thesis, Chalmers university of technology, Department of Computer Engineering. 2001
- [11] M. Kocher and K. Rose, "*Silicon Assembler. Design of a DCT/IDCT ASIC for real-time. JPEGNPEG compression*", Proc. 5th Annual. IEEE ASIC Conference and Exhibit, 1992, pp.

185-. 188.

[12] Q. Zhang & G. Theodoropoulos, “*Modelling SAMIPS: A Synthesizable Asynchronous MIPS Processor*,” Proceeding of the 37th Annual Simulation Symposium

[13] R. Raissi, “*The theory behind MP3*,” Dec. 2002, <http://rassol.com/cv/mp3.pdf>

[14] S. B. Furber et.al, “*AMULET2e: An Asynchronous Embedded Controller*,” Proceedings of the IEEE Volume 87, Issue 2, Feb. 1999 Page(s): 243 – 256

[15] S. B. Furber et.al, “*AMULET3: A High-Performance Self-Timed ARM Microprocessor*,” ICCE '98. Proceedings, Page(s): 247- 252

[16] Szu-Wei Lee, “*Improved Algorithm for Efficient Computation of the Forward and Backward MDCT in MPEG Coder*,” IEEE Transactions on Circuits and Systems II: analog and. digital proceeding, vol48, No.10, pp.990-994, 2001.

