White Paper
SPRAAJ6–July 2008

Bitstream Buffer in Audio Coprocessor

Fitzgerald J. Archibald, Stephen H. Li, Michael O. Polley, Ramesh Naidu G.

This paper explains compressed audio bitstream buffer implementation in an audio coprocessor to handle MPEG-1 Layer 3 (MP3) bitstreams. This paper provides hardware and firmware partitioning of bitstream buffer to address low power, and low gate count requirement in audio coprocessor and at the same time being capable of decoding different types of compressed audio streams like MP3 and MPEG-2/4 Advanced Audio Coding (AAC).

Index Terms: Audio coprocessor input/output buffering, audio streaming, bit reservoir handling in MP3 decoder, buffer management in audio codecs, circular buffer, compressed audio stream buffering, low power audio coprocessor.

Contents

List of Figures

1 Introduction

Compressed audio is packed into a bitstream by an encoder. the bitstream is a serialized compressed audio data stream for broadcast or storage media. The decoder will unpack the bitstream, and perform any necessary transformation of the unpacked data using information available in the unpacked data.

Compressed audio bitstreams are generally segmented as frames for error recovery, reducing decoding latency and buffering size. In the case of MPEG 1 Layer 3 audio compression, the current frame can be decompressed using data from the current and a limited number of past frames. The past frame dependency is due to bit reservoir support in MPEG 1 Layer 3 [3].

In order to decode the compressed audio bitstream, the bitstream needs to be buffered in a memory accessible by decoder hardware. The decoder hardware must perform bit extraction from the bitstream buffer to retrieve information needed for decompression. This paper addresses optimal bitstream buffering by eliminating redundant data copy with a focus on the buffering requirements for MPEG1 Layer 3 audio, especially the bit reservoir.

2 Bitstream Buffer Design

2.1 Audio Core Overview

The audio core, shown in Figure 1, is initialized by the host (possibly an ARM core) through the Control Input Port (CIP). The Bitstream Processing Unit (BPU) and the Arithmetic Unit (AU) are the processing modules of the audio core [1], [2]. the Data Input Port (DIP) is for fetching in the compressed audio into the BPU data memory. The Pulse Code Modulation (PCM) (Inter IC system Bus (I2S)) interface provides uncompressed linear PCM audio samples to the Digital to Audio Converter (DAC). The BPU and AU have separate program memory.

Program memory could be

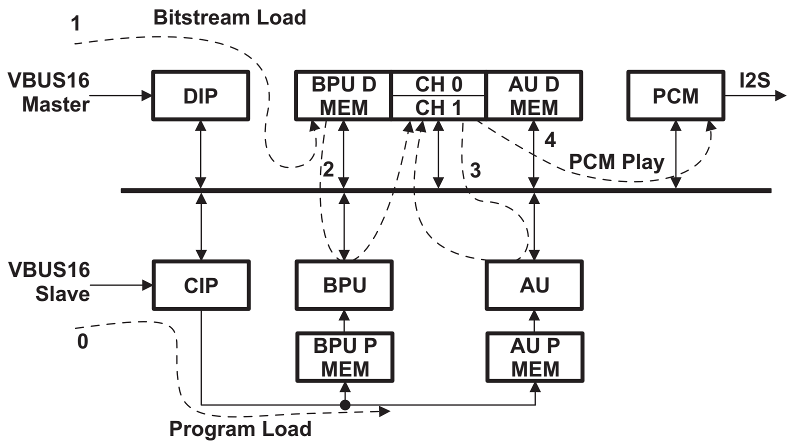
- Entirely ROM
- Partly ROM and partly RAM
- Entirely RAM

The BPU and AU data memory can be

- A mix of RMA and ROM
- RAM only

The BPU and AU communicate by way of a shared memory bank. The BPU serves as master and the AU as slave. The BPU can initiate the AU to start executing instructions from an AU program memory address, and the AU can interrupt the BPU to indicate AU processing completion. The BPU and AU can be idled (sleep) independently when not in use.

Figure 1. Audio Core Block Diagram



2.2 Hardware Design

The audio coprocessor contains the basic building blocks listed below for the firmware implementation of the compressed audio bitstream buffering.

- **Data Input Port / DMA**
  The DIP is the VBUS16 master to fetch the compressed audio bitstream into the BPU data memory. The DMA is used for the data transfer so the BPU can continue bitstream processing.
- **Interrupt**
  The DIP DMA data transfer completion interrupt is used to inform the BPU of DMA completion.
- **Burst Transfer**
  The burst data transfer capability allows for faster data transfer (two bytes per vbus cycle).
- **Idle**
  When the BPU is waiting for the DIP DMA data transfer to finish, the BPU can be idled. The BPU is woken from IDLE on the DIP DMA interrupt.
- **Control Input Port**

the CIP is useful in passing control information needed for processing the bitstream from the host to the audio core (especially the BPU). Also, the audio core can inform the host of bitstream properties, playback completion of stream, etc.

- **Bit Extractor**

  The bitstream buffer is 16-bit memory. During unpacking of the bitstream, a varying number of bits needs to be extracted from the bitstream. The number of bits extracted does not need to fall on a 16-bit (word) boundary. Thus, a bit register is used to point to the current position on the bitstream from where the next bit unpack needs to be performed. The MS 12 bits of the bit hold word address and the LS 4 bits hold the bit position in the word. Further, to extract bits, a 16-bit funnel shifter is used [2].

- **Bit Stream Break Interrupt**

  The bitstream contains more bits than available bit memory in the bitstream buffer. Thus, part of the bitstream is fetched into the bitstream buffer for decoding. A bitstream break interrupt can be set on any word boundary in the bitstream buffer. When the bit pointer reaches the word breakpoint, the breakpoint interrupt is generated. The bitstream break interrupt is needed for handling:

  – The end of bitstream buffer wrap-around

  – The main_data discontinuity caused by bit reservoir

    Without a bitstream break interrupt, the data will have to be stitched by means of extensive memory copy/move.

2.3 *Firmware Design*

2.3.1 Buffer Wrap-Around

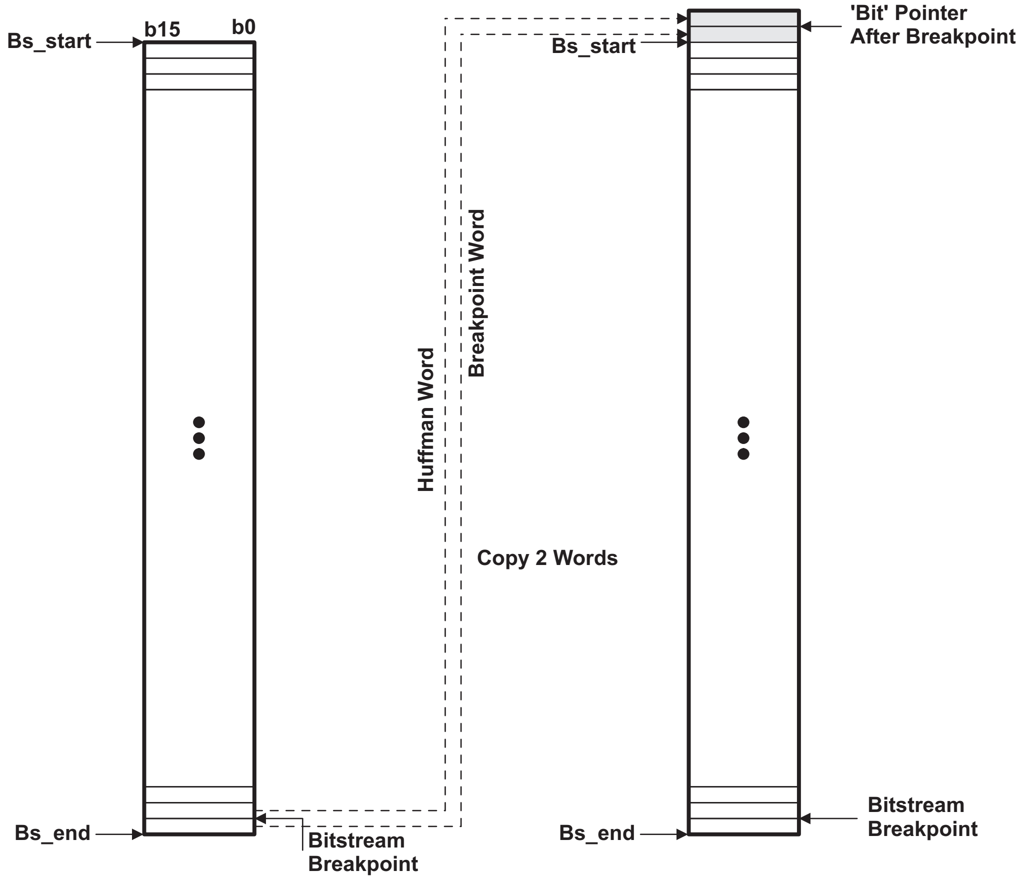
In order to have continuity of bitstream data fetched into the bitstream buffer, the bitstream buffer has to be a circular buffer. Otherwise, the bitstream data remaining in the bitstream buffer will have to be moved to the start of the bitstream buffer before the bitstream (the next part of) can be fetched from the DIP. This concatenation is needed because:

1. The bitstream is contiguous data.
2. The frame length is not fixed.
3. The bitstream buffer is smaller than the bitstream.

Memory copy uses excess power and is unwanted in managing bitstreams if the hardware architecture permits some sort of circular buffering. Circular buffering can be implemented using the bitstream breakpoint. Figure 2 illustrates the use of a breakpoint to realize circular buffering. The breakpoint is set on the last word of the bitstream buffer. When the bit pointer reaches within last 1-16 bits in the buffer following the previous bit extraction, the breakpoint interrupt is generated. Thus, the last word is either partially consumed or fully available for the next bit extraction when the breakpoint interrupt is serviced. So the last word in the bitstream buffer is stitched to the start of the bitstream buffer using memory copy. The bit pointer is also updated to point to this word for the next bit extraction instruction. When the wrap-around happens, it is definite that the data before the end are consumed, and hence, can be refilled with the next bitstream packet. Thus, data continuity is established on wrap-around.

Huffman decode look-back: The Huffman decoder in MPEG1 Layer3 could move the bit pointer backwards by up to 16 bits (one word). This means on circular buffer wrap-around, one more word needs to be copied in order to satisfy the Huffman decoder. The word before the breakpoint word (the last word in the bitstream buffer) is copied to the top of the *bitstream buffer+breakpoint word stitch* as shown in Figure 2.

Figure 2. Circular Buffer Using Bitstream Breakpoint



2.3.2 Bit Reservoir Handling

In order to have a fixed frame length and a varying amount of bits to encode a frame, a bit reservoir is used in MPEG 1 Layer 3. The amount of bits used for encoding each frame depends on the audio content in the frame. Thus, if frame(s) use lesser bits than the maximum possible and successive frames need more bits, the successive frame data can be embedded in previous frames as shown in Figure 3. By use of the bit reservoir, the bit rate is maintained constant and frames that need more bits can have more bits if the previous frames had unused bits. This essentially means that main_data of a frame can span over multiple frames (extending backwards/previous frames) limited by main_data_begin information in the bitstream. Also, the MPEG 1 Layer 3 standard states the maximum bitstream buffer required for the decoder is one frame size of 320 kbps, 48 KHz stream [3].

Key points to be remembered from MP3 bit reservoir are:

- The main_data of frame N will always follow the main_data of frame (N-1).
- Ancillary data of the present frame will follow the present frame data up to the following frame's main_data or next frame syncword, whichever comes first.
- 960 bytes of input bitstream buffer is enough to decode any MPEG 1 Layer 3 stream [3].

Figure 3. MPEG 1 Layer 3 Bit Reservoir

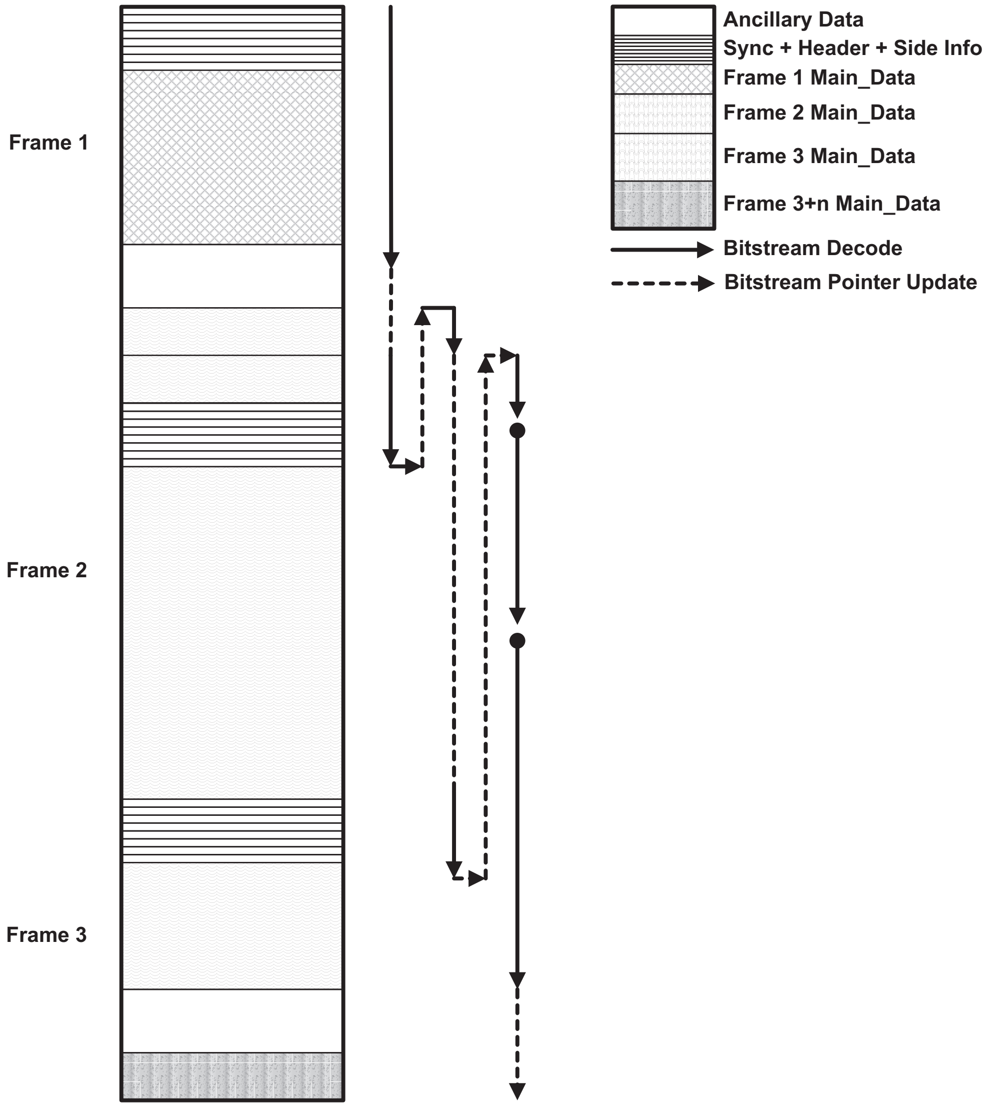
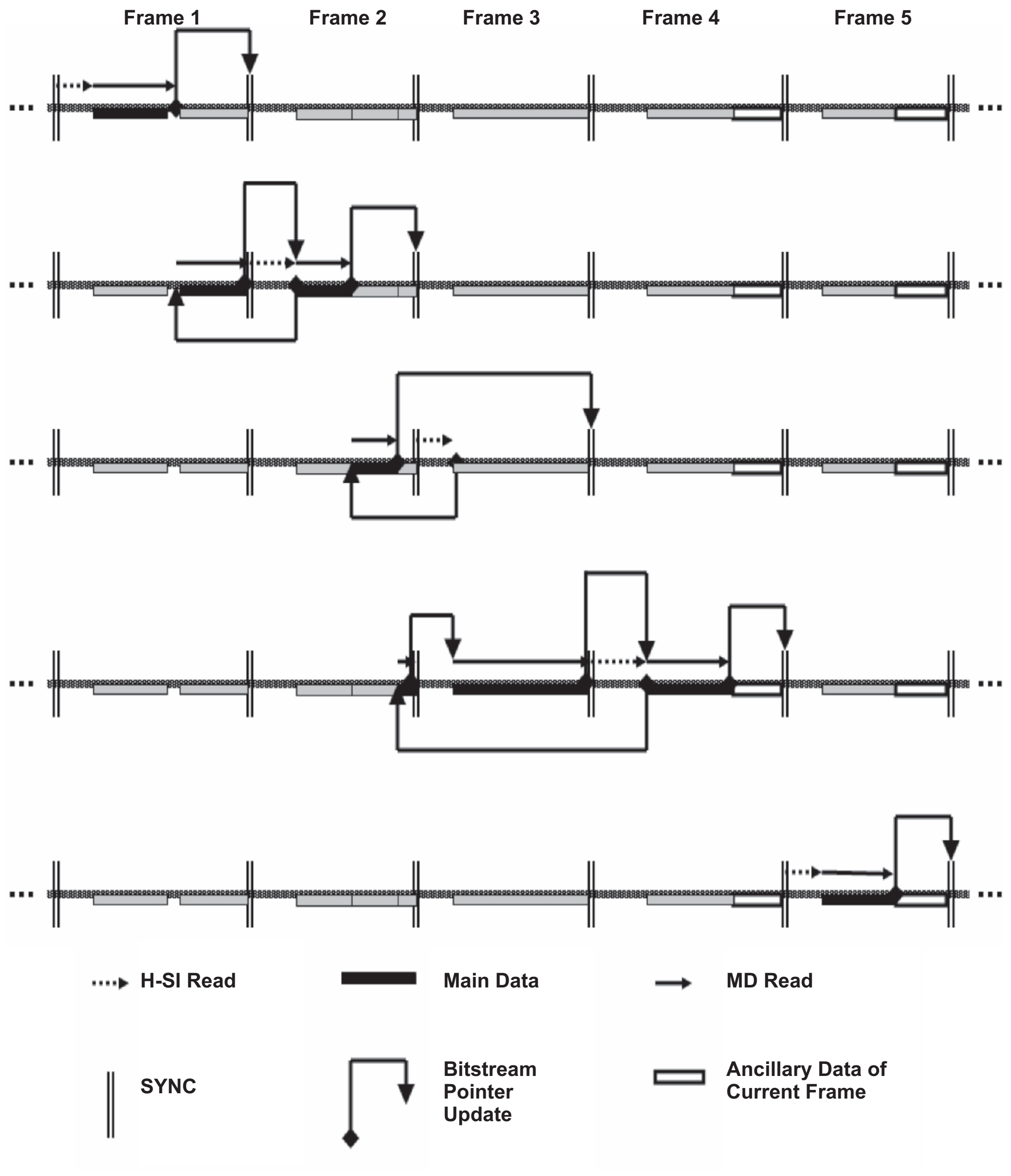


Figure 4 illustrates the data access for each frame decode as seen from the MP3 decoder. The points to note are:

- On detection of syncword and header, the frame length is computed. Frame length is used for finding the next frame syncword position.
- On completion of the present frame main_data parsing, the bit pointer is advanced to the next frame syncword position.
- In the case of main_data spanning across frame boundaries, the bit pointer needs to skip the *syncword+header+cyclic redundancy code (CRC)+side information* of the frames that cause discontinuity in the main_data [3].
- With 32 kbps, a 48 KHz stream has a frame length of 96 bytes [3]. The minimum buffer size required for the decoder is 960 bytes [3]. The maximum main_data_begin value can be 512 bytes [3]. This means the main_data can be spread in a maximum of 10 frames.

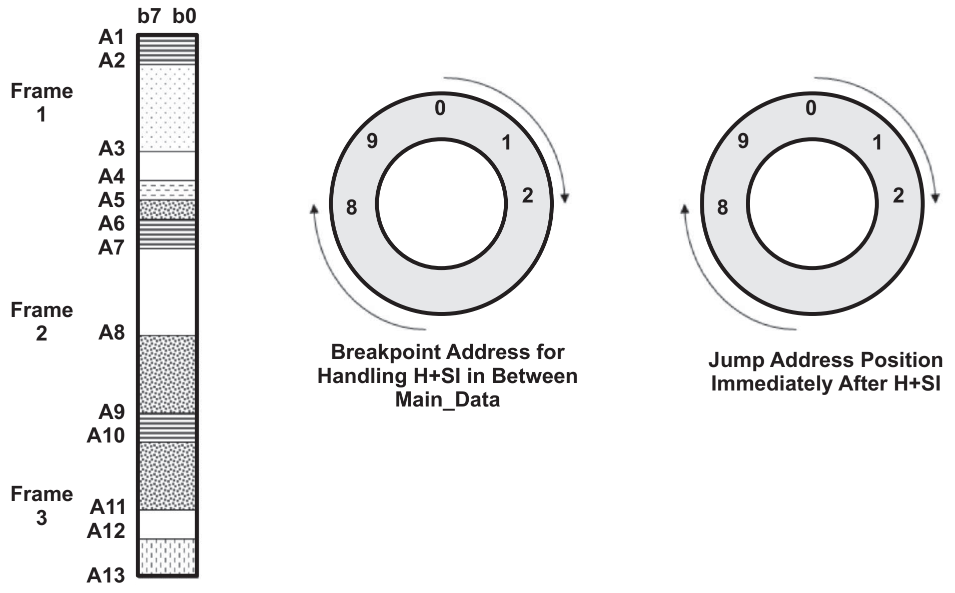
Figure 4. Decoder Bitstream Access Sequence



The syncword start address and the side information end addresses of the last 10 frames are stored in a buffer, as shown in Figure 5. The buffer could be circular since only one frame address will be inserted to the latest, and the oldest frame address will be deleted as the decoder processes one frame at a time. These addresses are used for:

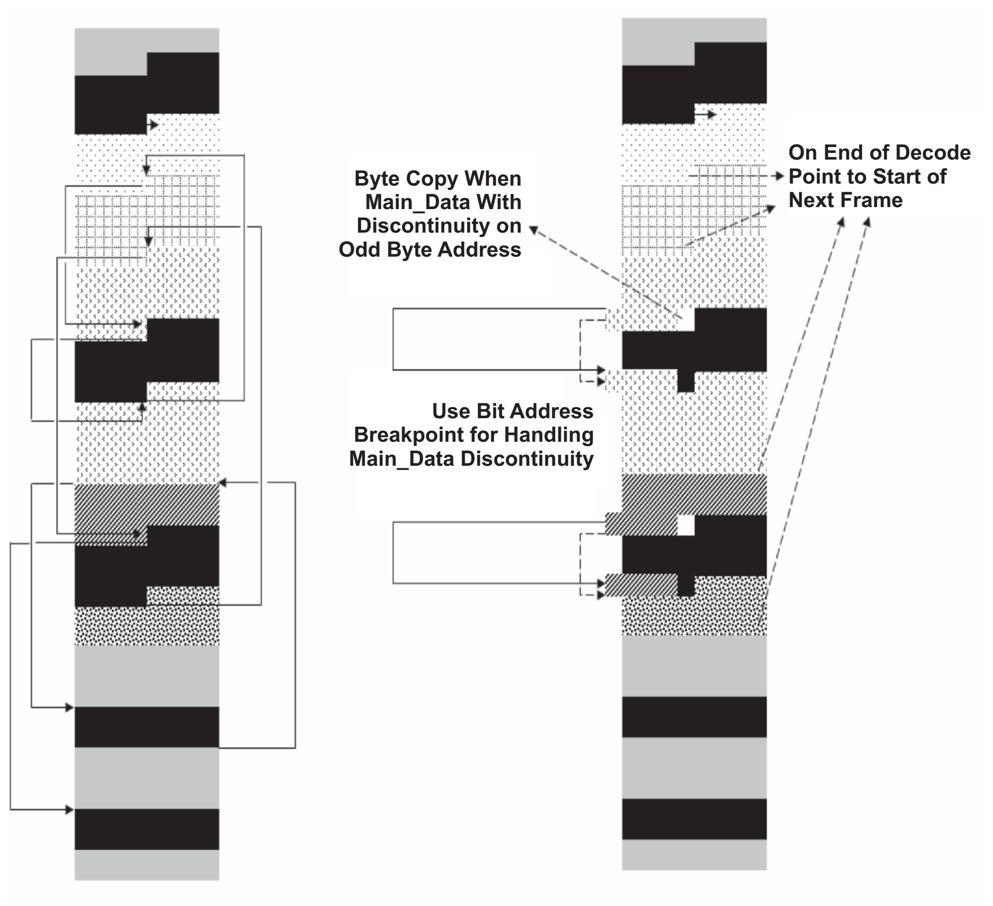
1. Setting breakpoints to skip the non main_data part of data in the buffer during main_data parsing, in the case of main_data being spread across multiple frames.
2. For detecting the actual start of the main_data as main_data_begin does not contain the *syncword+header+CRC+side information* size of the frames across which the main_data spreads. The syncword position is used for setting up the bitstream breakpoint. The end of the *header+side information* position is used for repositioning the bit pointer.

Figure 5. The main_data Discontinuity Break Address Buffers


b7 b0
A1
A2
Frame 1
A3
A4
A5
A6
A7
Frame 2
A8
A9
A10
Frame 3
A11
A12
A13
0
1
2
8
9
Breakpoint Address for Handling H+SI in Between Main_Data
Jump Address Position Immediately After H+SI

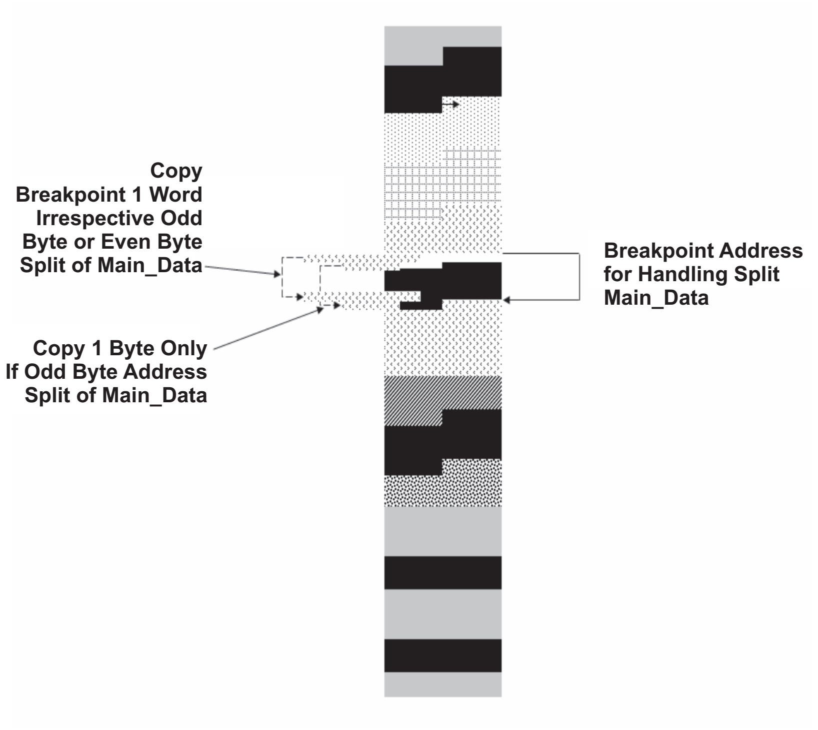
Bitstream Stitching: MP3 data is byte aligned (syncword, main_data start on byte boundary, side information ends on byte boundary). The bitstream breakpoint is on a word address (16 bit). So, any non-word aligned breakpoint has to be turned into a word-aligned breakpoint by moving/stitching the odd byte of data with the next set of data and setting the breakpoint before this odd byte. In Figure 6, the main_data of the second frame is in the previous frame and current frame. Both the syncword start and side information end are on odd-byte addresses. The odd byte before the syncword word is copied just before the main_data continuation after the side information end. The byte copy onto side information is acceptable since the side information of the current frame is already parsed and the main_data of the current frame will never extend beyond the current frame end.

Figure 6. Odd Byte Address Breakpoint Handling


Byte Copy When Main_Data With Discontinuity on Odd Byte Address
On End of Decode Point to Start of Next Frame
Use Bit Address Breakpoint for Handling Main_Data Discontinuity

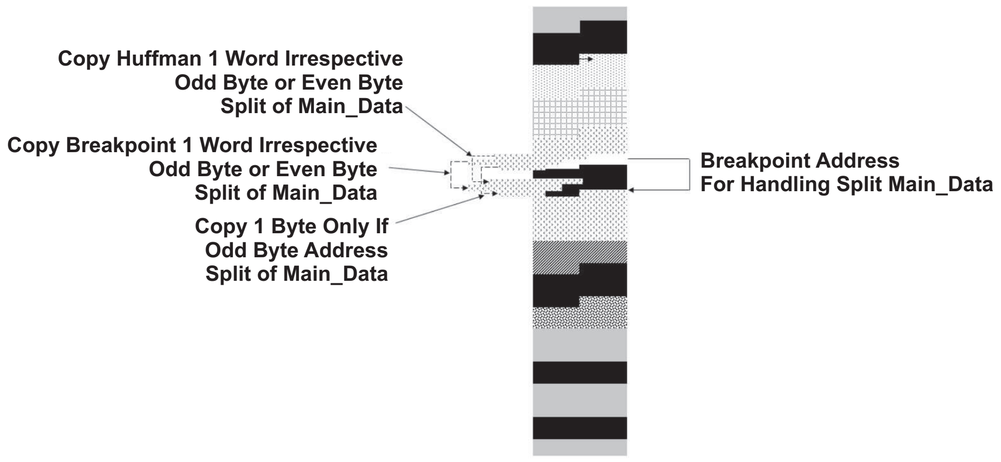
The setting of a breakpoint necessitates the need for copying an additional single word (breakpoint word) as illustrated in Figure 7 and explained in Section 2.3.1.

Figure 7. Odd Byte and Breakpoint Word Stitching



The Huffman decoder look back, as explained in Section 2.3.1, needs an additional single word before the breakpoint word to be stitched as well. Thus either two words, or two words and a byte need to be stitched depending on whether an odd byte is absent or present, respectively, as shown in Figure 8.

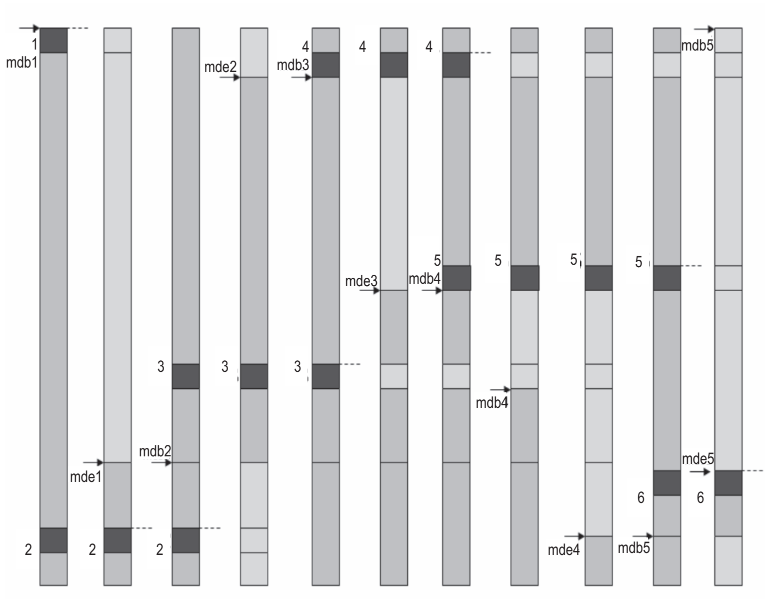
Figure 8. Odd Byte, Breakpoint Word and Huffman Look Back Word Stitching



2.3.3 Bitstream Buffer Refill

Figure 9 illustrates the buffer refill mechanism. As soon as a frame is decoded/parsed, the bits consumed can be refilled. In order to be able to decode MP3 streams using the least possible buffer size, the bitstream buffer needs to be refilled precisely with respect to the refill start position, the refill size, and the start time of the refill. If the frame is self-contained, the refill is as simple as fetching new data into the last decoded frame area in the circular bitstream buffer. The refill end position in the bitstream buffer can be determined from the main_data_begin information contained in the frame. The buffer position starting from the refill start position (past frame syncword position) up to the refill end position (main_data begin position of the current frame) can be refilled. The header and side information of the current frame had already been parsed. By doing the refill this way, it can be ensured with a buffer size of 480 words that the entire current frame including the main_data of the current frame contained in previous frames is available in the bitstream buffer for decoding.

Figure 9. Buffer Refill Diagram



2.3.3.1 Buffer Wrap on Refill

If the refill area wraps around on the bitstream buffer end, the refill has to be within two DMA transfers. The first transfer is to fill from the refill start position until the end of the bitstream buffer. The second transfer is to fill from the bitstream buffer start until the refill end position.

2.3.3.2 Successive Syncword Search

For bit rates other than free format, the frame length can be determined from the bit rate and sample rate information contained in the header. So the bit pointer can be advanced by the frame length from the current frame syncword position to obtain the next syncword position. This ensures the detected syncword is not a pseudo-syncword as the syncword pattern is not unique. Also, it is a good means to check the header to detect stream consistency. In other words, the header can be assumed as an extension to the syncword. Note that size of the header including the syncword (4 bytes), CRC word (16 bits) if present, and side information (17 bytes for mono; 32 bytes for stereo) are a fixed number of bits [3]. Extending the theory further, the main_data_begin is contained in the side information. Thus, with 480 words + 19 words of buffer, the entire current frame (including main_data contained in previous frames) and the next frame header and side information is contained in the bitstream buffer.

End of stream handling: On the last frame decode, the next frame syncword will not exist. Hence, successive syncword logic should be skipped. The last frame can be detected from the DIP DMA source address if the file size of the stream being decoded is known. Typically, the host would know the file size.

2.3.3.3 Bitstream Error Handling

If the next frame syncword is not detected in the estimated buffer position, the bit position is advanced by 8-bits from the current syncword position and the syncword search is performed again. This is a means to recover from a pseudo-syncword.

In the case that a stream starts with non-MP3 data followed by an MP3 bitstream, on detection of the first syncword, the buffer start until the syncword position is immediately refilled, so that one full frame is in the buffer. If no syncword is found until the end of the buffer, then the entire buffer is refilled. Then, the syncword search is performed. In case no successive syncword is found, other than for the last frame, the frame is assumed erroneous and not decoded.

In the case that non-MP3 data occurs between MP3 data, the handling is similar to non-MP3 data in the beginning. The MP3 data before the non-MP3 data is decoded. The MP3 frame preceding the non-MP3 data is skipped from the decoder. A syncword search is performed on the available data. On syncword detection, the buffer is refilled up to the syncword position. The refill is slightly different as the refill may not start on the bitstream buffer start. Then, decoding continues normally.

In the case that the main_data_begin field is pointing to invalid data, the frame is not decoded. Invalid data could be main_data_begin pointing to future data, since it is a circular buffer. If the stream starts with non-zero main_data_begin (i.e. the frame refers to a previous frame for the current frame decode), the bit pointer will look at future data for main_data decode/parse because of the circular buffer wrap around on the bitstream buffer start. In this case, the current frame is neither decoded nor refilled. Then, the next stream header and side information is parsed. Again, the validity of the main_data_begin pointer position is established before proceeding to decoding. Note that the refill is not performed on non-decodable frames since the data is established as MP3 frames, but is not decodable since the previous frames are not present in memory. This necessitates the case that only partial frame or full frame but no next frame start is present in the bitstream buffer. If it happens in a way that the next syncword is not detectable in the bitstream buffer, refill up to the side information of the next frame. This would ensure the present frame is definitely present in the bitstream buffer including the main_data in past frames.

2.3.4 DMA Cycle Hiding

The DMA refill operation can be run in parallel with the decode of the last granule/channel in the current frame in the AU. The DMA refill operation needs the side information (main_data_begin) of the next frame be parsed while the current frame PCM samples are being generated. This is possible because of dual core architecture. The side information needed for the AU for processing the PCM samples is already copied on the AU memory. Hence, the BPU can update the BPU memory with the next frame side information.

3 Results

Buffer logic program memory is 540 locations, and Data RAM memory for state variables is 64 words. Bitstream buffer RAM size of 480+19 words is sufficient for decoding MPEG 1 Layer 3 streams. Error handling/recovery is performed without increasing the bitstream buffer size.

Worst case data copy, due to data stitching, is 5*10+4 = 54 bytes per frame. Bit reservoir handling by the circular buffer implemented using the bitstream breakpoint avoids a huge memory copy. Huffman look back, breakpoint word, and odd byte alignment is handled with data copy/stitching. Worst case data stitching cycles per frame is 120*(10+1) = 1320 BPU clock cycles. Buffer refill cycles is hidden with PCM decode.

4 Summary

By use of data breakpoints, a pseudo circular buffer is implemented for handling the compressed audio bitstream. Thus eliminating the need for complex hardware logic and/or unwanted data movement by the system required to handle the seamless stream flow to the decoder. As seen from the design and implementation of bitstream buffering for the MP3 decoder, the buffering logic can be easily extended for other audio decoders for compression standards like AAC, and Windows Media Audio (WMA9).

5 Acknowledgment

The authors thank Mohamed Mansour of Texas Instruments for his contributions in MP3 decoder implementation in audio coprocessor.

6 References

1. *Programming of Audio Coprocessor for MP3 Decoder* by G. Naidu., F. J. Archibald, S. H. Li, submitted for publication.
2. *An AC-3/MPEG Multi-standard Audio Decoder IC* by S. H. Li, J. Rowlands, P. Ng, M. Gill, D.S. Youm, D. Kam, S.W. Song, P. Look , CICC 1997, Proceedings of the IEEE 1997 Volume , 5-8 May 1997, pp. 245 - 248.
3. *Coding of moving pictures and associated audio and digital storage media at up to about 1.5Mbit/s, Part3: Audio* ISO/IEC 11172-3:1993, 1993.

Fitzgerald Archibald earned a B.E (Electronics and Communication Engineering) from PSG College of Technology, Coimbatore, Tamil Nadu, India in 1996. He worked on control systems software development for geo-synchronous satellites from 1996 to 1999 in ISRO Satellite Centre, Bangalore, India. In 2001-2002, he worked on speech decoder, real-time kernel, and audio algorithms for DVD audio team in Sony Electronics, San Jose, USA. While in Philips Semiconductors (Bangalore, India, and Sunnyvale, USA) in 1999-2001 and 2002-2004, he worked on audio algorithms and systems for STB, DTV, and internet audio. He is part of the Personal Audio Video and Digital Imaging groups in Texas Instruments Inc, Bangalore, India from 2004-till date working on audio, video, and imaging systems and algorithm development. Interests include multimedia and control algorithms and systems.

Mr. Archibald is member of AES.

Stephen Li earned his BS in Electrical Engineering at University of Texas at Arlington, MS in Biomedical Engineering at UT Arlington/UT Southwestern Medical Center, and MS in Applied Mathematics at Southern Methodist University.

He joined Texas Instruments in 1984, has worked in different capacities on the design, modeling, processing, application, and architecture definition of VLSI IC. He is the author of several technical papers and journal articles covering VLSI architecture and design. He owns over 25 U.S. patents in the same area, as well as A/V algorithm development.

Mike Polley received his B.S., M.S., and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology.

In 1996 he joined Texas Instruments to help TI influence the DSL standards bodies and enter the ADSL market. He led the ADSL research group developing DSP based solutions for evolving DSL standards. He then came unwired - he helped TI establish a fixed-wireless access chipset business and then led the way developing multiple-antenna wireless LAN technology and implementations in advance of the IEEE 802.11n standard. He currently manages the architecture team developing camera and video chips for cell phones and he is a Distinguished Member of Technical Staff.

Ramesh Naidu was born in Anantapur and received BTech in electronics and communications engineering, from Intell Engineering College, Anantapur, India in 2004. This author completed MTech in telematics and signal processing from NIT Rourkela, India from July 2005-June 2007.

He has completed MTech project work in Texas Instruments, Bangalore, India in the role of project trainee.