Lab 12: Implementation of MP3 Player in LabVIEW

In this lab, the entire MP3 decoder system for mono MP3 files is implemented in LabVIEW. The last section of this lab covers the modifications made in order to obtain a real-time throughput from the decoder.

L12.1 System-Level VI

Figure 12-12 illustrates the system-level BD of the implemented MP3 decoder. The subVIs located in the `For Loop` are repeated as many as number of frames.

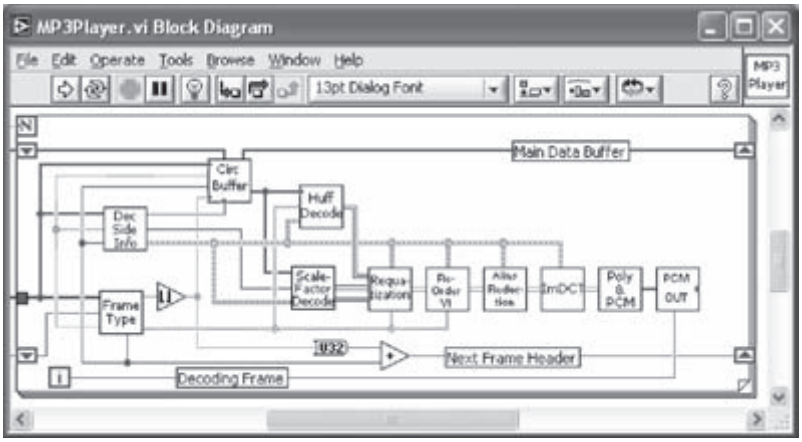

Figure 12-12: System-level Block Diagram of MP3 decoder.

The `Frame Type` VI finds the frame header from the bit stream and extracts the decoding bitrate and sampling frequency. The location of the frame header is passed to the `Dec Side Info` VI to identify its beginning. The `Dec Side Info` VI decodes the Side Information and bundles the decoded parameters into a cluster for easy access by the other VIs.

A `Shift Register` is placed to serve as a buffer for Main Data as part of the bit reservoir technique. The `Circ Buffer` VI is used to fill the buffer with Main Data, to calculate a pointer marking the start of Main Data of a current frame, and to provide Main Data and its starting index as outputs to the other VIs.

The Scale Factor Decode VI decodes the scale factors that are used to suppress quantization and any other noise. The output of this VI consists of two arrays: scalefac_l and scalefac_s. The Huffman Decode VI incorporates a number of subVIs to determine the required information about Huffman decoding, to calculate the length of Huffman coded bits, and to decode different regions of Huffman coded data. The Requantization VI combines the output of the Scale Factor Decode and Huffman Decode VIs. This VI implements the requantization equations.

The Reorder VI arranges the frequency lines of short blocks in the same order as in the MDCT block of the encoder. The Alias Reduction VI computes the antialias coefficients and weighs the frequency lines accordingly. The IMDCT VI computes IMDCT, does windowing on the IMDCT output, and performs overlap/add on the windowed output to generate the polyphase filter input.

The Poly & PCM VI carries out three operations. It multiplies every odd sample of each odd subband by '–1' (referred to as Frequency Inversion), implements the polyphase filter, and generates PCM samples.

L12.2 LabVIEW Implementation

L12.2.1 MP3 Read

As the first step of the MP3 decoding process, an MP3 file is opened and read by the MP3 Read VI, see Figure 12-13. This VI reads an MP3 file as specified by the File Path control via the Read Characters From File VI (**Functions** → **All Functions** → **File I/O** → **Read Characters From File**). The data stream is converted to unsigned integer byte using the String to Byte Array function (**Functions** → **All Functions** → **String** → **String/Array/Path Conversion** → **String to Byte Array**).

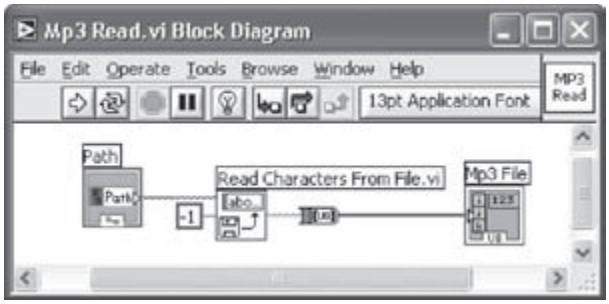

Figure 12-13: BD of MP3 Read VI.

L12.2.2 MP3 File Info

The `MP3 File Info` VI extracts and displays the information about an MP3 file that is present in the file header, see Figure 12-14. It consists of two VIs: `Frame Type` and `MP3 Info Display`. The former finds the header and extracts the decoding information, while the latter displays this information in an ordered fashion.

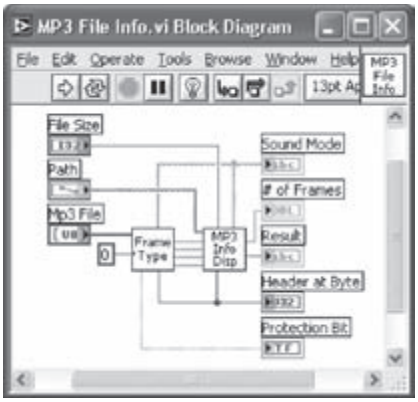

Figure 12-14: BD of MP3 File Info VI.

The `Frame Type` VI, see Figure 12-15, uses the `Find Header` subVI to find the new location of the header by searching for twelve consecutive ones. Once the header is found, three bytes of the header which contain the frame information are passed to the `Find Type` VI. Next, the `Find Type` VI performs the actual decoding of the header information using table look-ups.

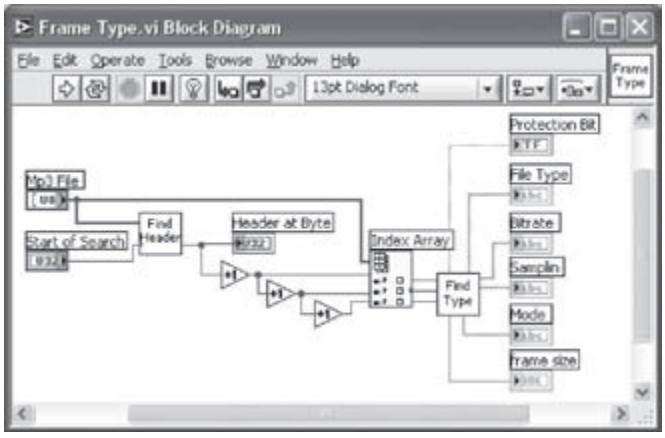

Figure 12-15: BD of Frame Type VI.

The MP3 Info Display VI generates a string array based on the output of the Frame Type VI. The output string array of this VI is displayed on the FP.

L12.2.3 Dec Side Info

The Dec Side Info VI extracts the 17 bytes of the Side Information from the bit stream and decodes it in a sequential order. Figure 12-16 illustrates the extraction of the 17 bytes of the Side Information using the Array Subset function. In case an optional CRC is present, indicated by the Protection bit, the starting location of the Side Information is moved by two bytes after the header. Next, each byte of the 17-byte long Side Information is wired to the Get Bits subVIs in order to extract the individual parameters.

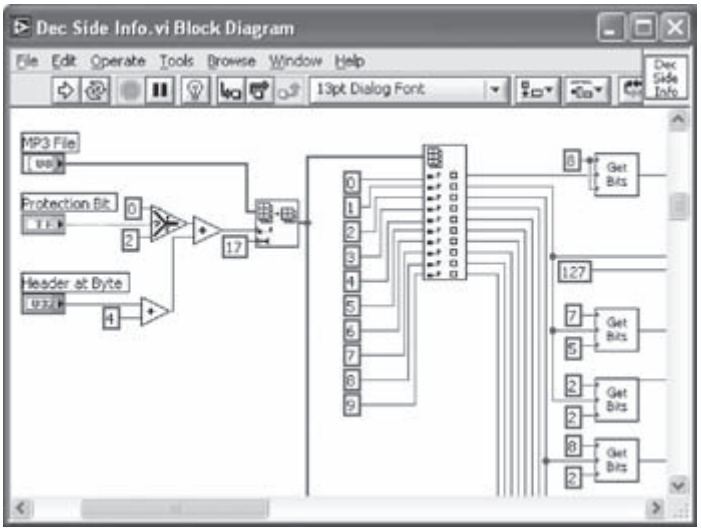

Figure 12-16: A BD section of Dec Side Info VI.

The Get Bits subVI performs 'logical AND' to extract bits from the input byte. This VI provides a mask that contains ones at the positions to be extracted from the input byte. The length and location of the ones in the mask are specified by the input parameters to the VI. Notice that the MSB and LSB of the byte are indicated by the indices 8 and 1, respectively. If a field is split into two bytes, the bits extracted from the higher byte is logically shifted and added to the bits from the lower byte to form one value. All the parameters of the Side Information extracted from the bit stream are bundled to form a cluster, indicated by Side Info.

L12.2.4 Circ Buffer

Main Data of a frame can be obtained from previous frames as well as a current frame depending on the value of the Main Data Begin Pointer. The `Circ Buffer` VI, see Figure 12-17, extracts the Main Data section of the current frame and fills a 1024 point buffer. Thus, a total of 1024 Main Data samples from the previous frames and current frame are stored in the buffer. This buffer is rotated in such a way that a new Main Data section is inserted to the end of the buffer. In order to fetch Main Data for decoding of a current frame, the index of the start of Main Data is provided by subtracting the sum consisting of the size of the new Main Data section and `Main Data Ptr` from the length of the buffer.

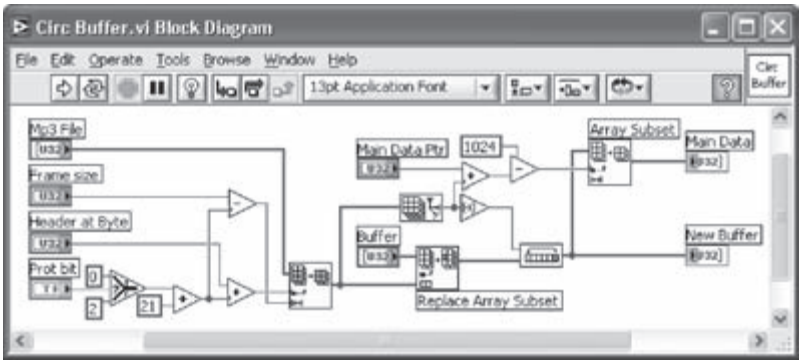

Figure 12-17: BD of Circular Buffer VI.

L12.2.5 Scale Factor Decode

The `Scale Factor Decode` VI consists of two VIs: `Scale Factor Decode0` and `Scale Factor Decode1`, which are used for decoding Granule0 and Granule1, respectively. Considering that the basic components of these VIs are the same, here only a general description for the `Scale Factor Decode0` VI is mentioned. A BD section of the VI is illustrated in Figure 12-18.

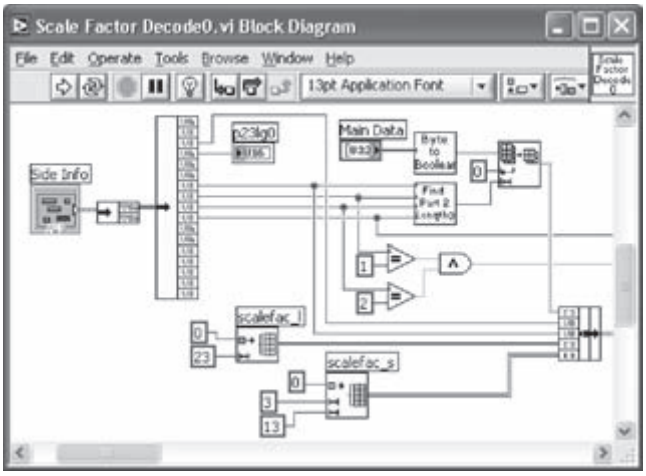

Figure 12-18: A BD section of Scale Factor Decode0 VI.

The Side Information input of the VI is unbundled to obtain the individual fields. `Find Part2 Length0` VI calculates `part2_length`, which forms the length of the coded scale factors in Main Data. Once `part2_length` is calculated, the scale factors are extracted. Note that the equivalent C code for extracting the scale factors is provided as part of the BD on the accompanying CD. The `Byte to Boolean Array` VI converts each element of the integer array to Boolean digits and concatenates them to form a Boolean array.

Once the scale factor decoding for Granule0 is done, the scale factor decoding for Granule1 is done creating two scale factor outputs, `scalefac_l` and `scalefac_s`.

L12.2.6 Huffman Decode

The `Huffman Decode` VI decodes coded data by performing a table look-up using thirty-four standard Huffman tables. This VI consists of the following subVIs: `Part2 Length`, `Byte to Boolean Array`, `Huffman Info`, `Big Value Search`, `Big Value Sign`, `Count1 Search`, and `Count1 Sign`. In what follows, this VI and its subVIs are described. Considering that the `Part2 Length` and `Byte to Boolean Array` subVIs are already explained, we begin by the `Huffman Info` VI.

The `Huffman Info` VI, see Figure 12-19, extracts the individual table_select for each of the three sub-regions of the big_value region from the table_select field of the Side Information.

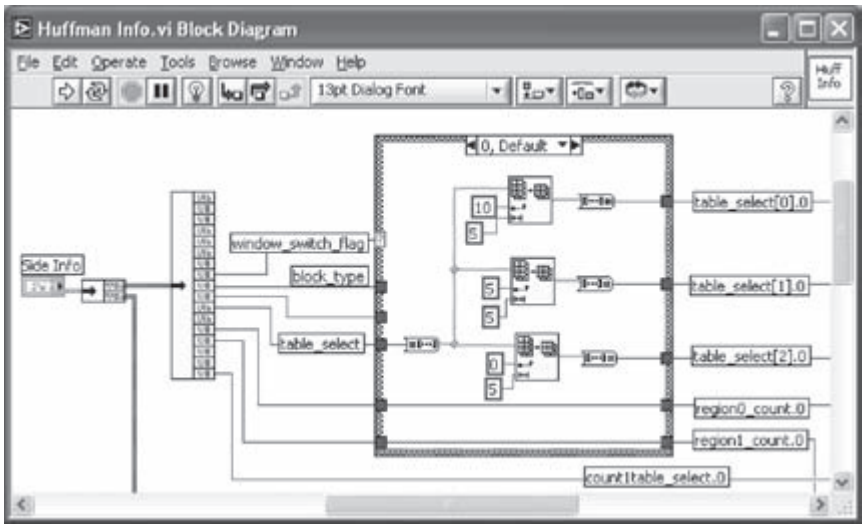

Figure 12-19: A BD section of Huffman Info VI.

Note that the two parameters region0_count and region1_count are used to determine the boundaries of the big_value region. Once table_select and the boundaries are determined in the `Huffman Info` VI, the decoding of the Big Value and Count1 regions begins.

The big_value stage of the `Huffman Decode` VI consists of the `search loop` section which is located in the `While Loop` shown in Figure 12-20. This section searches for a codeword match in the big_value region. Before this search takes place in the `search loop`, the current region to be decoded must be determined. This is achieved by using the `Formula Node` exhibited in Figure 12-20. This is important, as the table_select value is region dependent. The `search loop` extracts one bit at a time, appends it to the previously extracted bit, and passes it to the `Big Value Search` VI. The `search loop` terminates when a match is found.

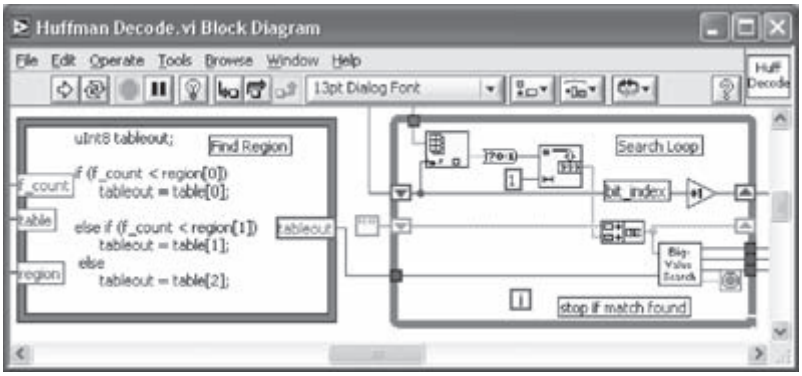

Figure 12-20: Search loop section of big_value stage.

The `Big Value Search` VI performs the search for the bit string generated in the `search loop` using the tables and generates the decoded values as its output. Figure 12-21 illustrates the BD of the `Big Value Search` VI. It uses the standard Huffman tables with `table_select` being used as an index.

The `Search 1D Array` function (**Functions → All Functions → Array → Search 1D Array**) is used to search for the codewords in the selected tables. Once a match is found, the VI extracts the decoded values, `X` and `Y`, from the selected table at the index produced by the `Search 1D Array` function. The Boolean indicator, denoted by `Match`, is used to terminate the `search loop` of the `Huffman Decode` VI.

Next, the `Big Value Sign` VI, see Figure 12-22, is used to determine the Escape and the sign value for `X` and `Y`. As mentioned earlier, the Escape value is only evaluated when the decoded value is equal to fifteen. This value is determined by fetching

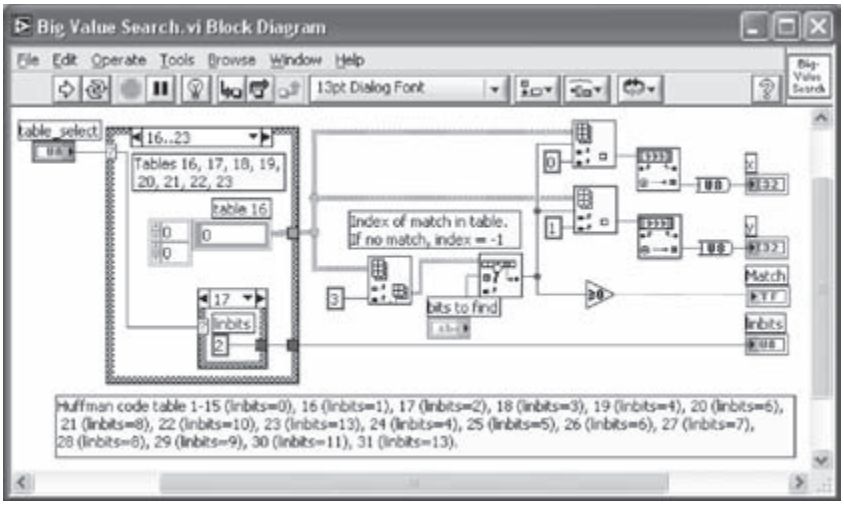

Figure 12-21: BD of Big Value Search VI.

Linbits number of bits. Then, the value represented by the fetched bits is added to the decoded outputs as shown in the `Escape Value` case structure in the BD. The decoded outputs are updated with the Escape and sign value to form the final output.

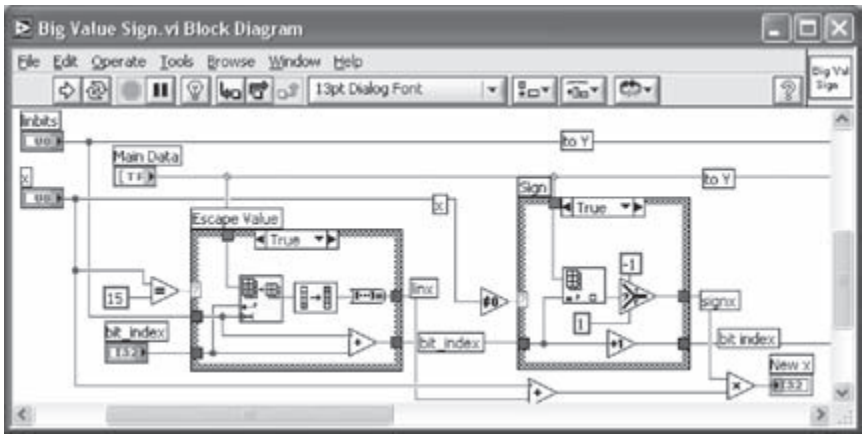

Figure 12-22: Escape and Sign bit decoding.

After the big_value stage, the decoding of the count1 stage follows. This stage is similar to the big_value decoding, except that this time four decoded outputs, `V`, `W`, `X`, and `Y` are obtained. The process of determining the sign bit is similar to the big_value sign process. Figure 12-23 illustrates the count1 stage of the Huffman decoding.

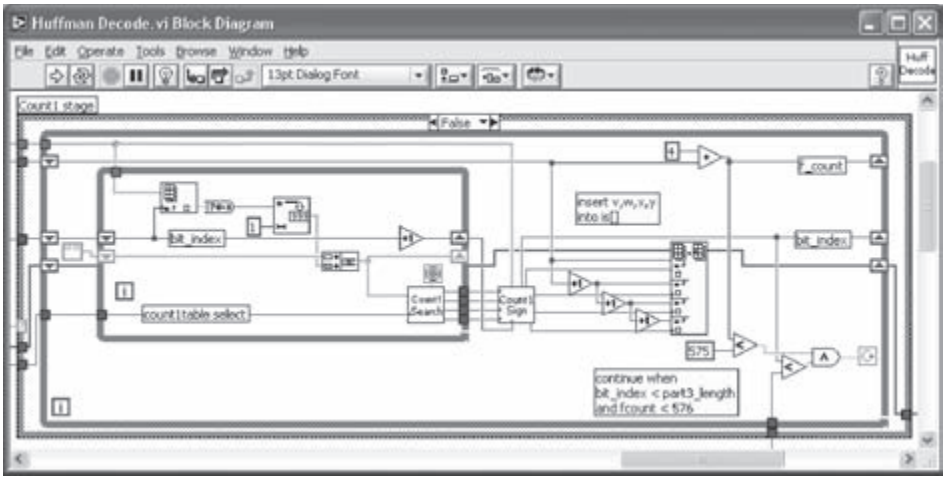

Figure 12-23: Count1 sage of Huffman Decoding.

L12.2.7 Requantization

The `Requantization` VI combines the outputs of the `Scale Factor Decode` and `Huffman Decode` VIs to implement the requantization equations given by Equations (12.2) and (12.3). Figure 12-24 shows one section of the `Requantization` VI, which illustrates the implementation of the requantization equation for long blocks.

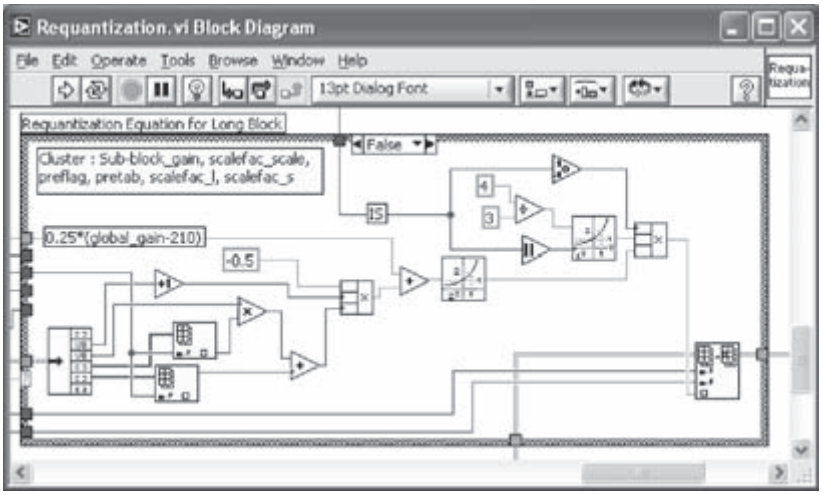

Figure 12-24: A BD section of Requantization VI.

This VI calculates the overall scaling factor and raises the Huffman decoded outputs to the power 4/3. Then, it multiplies them with the overall scaling factor and the sign of the Huffman decoded outputs. An array consisting of all the outputs is built to serve as the output of the requantization block.

In the other section of the `Requantization` VI shown in Figure 12-24, the `MainSI Params` VI is used. This VI is created to extract the frequently used fields of the Side Information including `global_gain`, `window_switch_flag`, `block_type`, `mix_block_flag`, `scalefac_scale`, and `preflag`. The `MainSI Params` subVI is used in the other decoding stages, e.g. `IMDCT` VI, as well as in the requantization block.

L12.2.8 Reordering

The `Reorder` VI changes the order of frequency lines or the output of the requantization block. Since reordering is required for short blocks and mix blocks only, the `Reorder` VI first identifies the block type before performing reordering. The BD of the `Reorder` VI for short blocks is illustrated in Figure 12-25. Note that this VI simply copies the frequency lines to a new array having a reordered index.

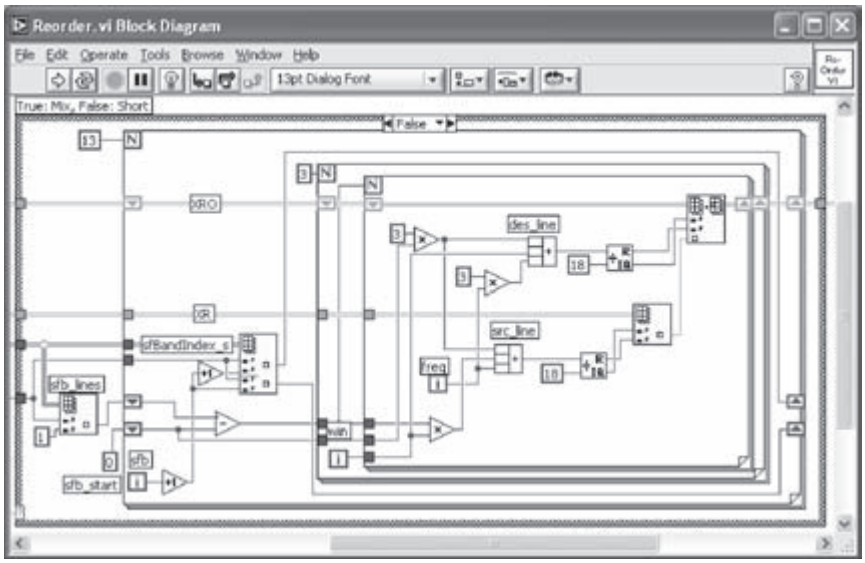

Figure 12-25: BD of Reorder VI.

For mix blocks, the reordering is not carried out for the first two subbands, as they consist of long blocks.

L12.2.9 Alias Reduction

The `Alias Reduction` VI scales the reordered frequency lines with the alias coefficients. Since alias reduction is performed only for long blocks, this VI identifies such blocks and performs alias reduction on them. The BD shown in Figure 12-26 carries out the butterfly calculation. Note that the alias coefficients are defined in the arrays `ca` and `cs`, see Figure 12-26.

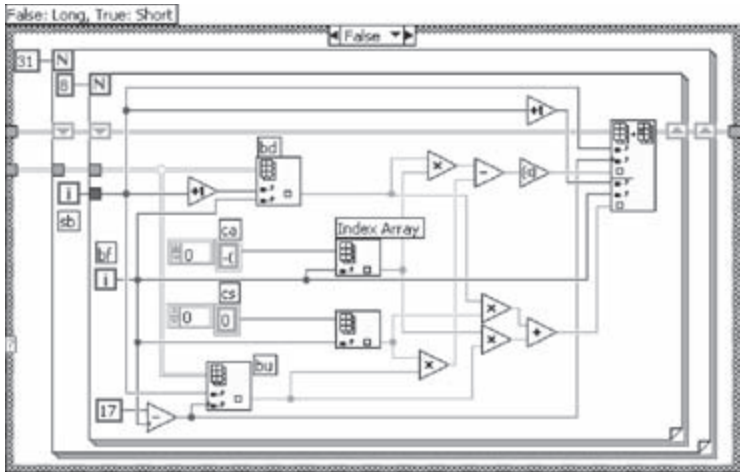

Figure 12-26: A BD Section of Alias Reduction VI.

L12.2.10 IMDCT

The `IMDCT` VI converts the frequency domain samples to time domain samples, thereby providing the input samples to the Polyphase filter. This VI also performs windowing and an overlap/add operation on the output samples as shown in Figure 12-27. The global variable `PrevBlck` is used to pass the output values used for the overlap/add operation from one frame to another. Note that this value is initialized only once

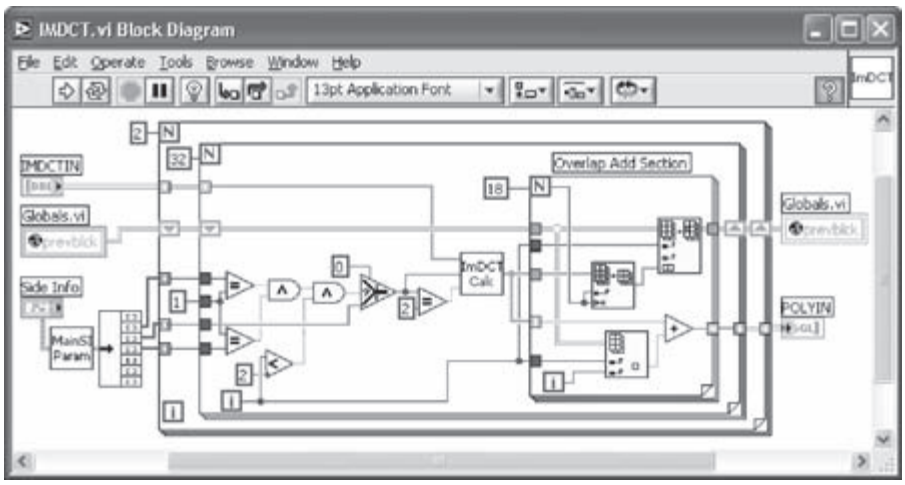

Figure 12-27: BD of IMDCT VI.

at the beginning of decoding. The `MainSI Params` VI, mentioned earlier, is located to obtain the main Side Information fields used for the IMDCT operation.

The `IMDCT Calc` subVI performs the actual IMDCT computations. Since the formula used for the calculation of IMDCT is different for long and short blocks, the implementation of each case is done separately.

The BD for calculating IMDCT for long blocks is illustrated in Figure 12-28. Input values are convolved with the IMDCT coefficients denoted by `Cos_l`. The windowing is performed on the convolution output. This is done by multiplying the IMDCT output with the predefined window coefficients, denoted by `win`.

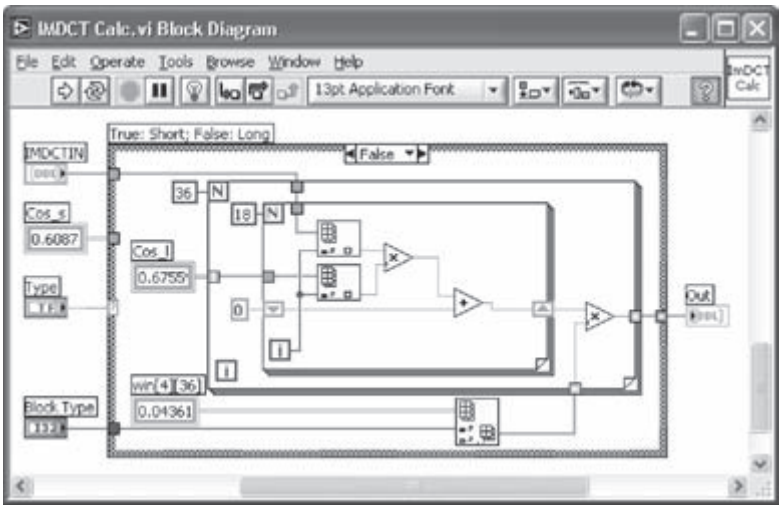

Figure 12-28: BD of IMDCT Calc VI (long block case).

For short blocks, the three outputs of IMDCT are overlapped and added for the final IMDCT array.

L12.2.11 Poly & PCM

The `Poly & PCM` VI generates the final output of the MP3 decoding process, i.e., `PCM` samples. This VI transforms the thirty-two subbands of eighteen samples each to eighteen subbands of thirty-two PCM samples. It consists of three subVIs: `FreqInv`, `MDCT & Wvec`, and `Add & PCM`. Figure 12-29 displays the BD of the `Poly & PCM` VI and its associated subVIs. Two global variables, a 1024 point `Vvector` and `bufferOffset`, are created for transferring the data of the current frame to the next frame.

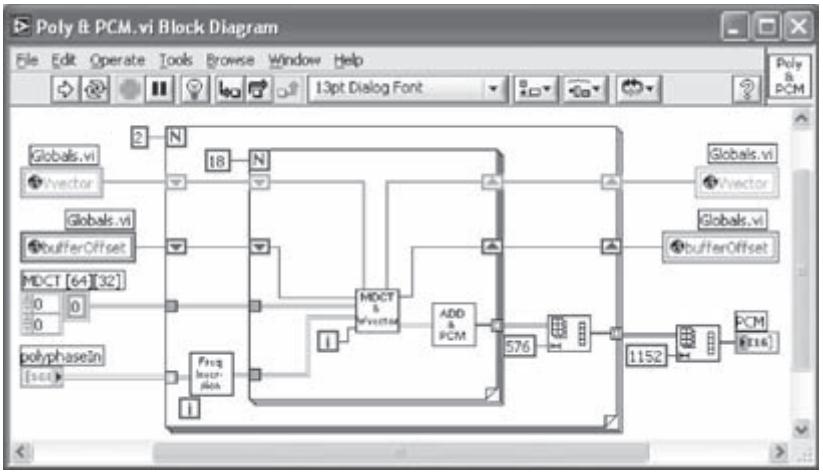

Figure 12-29: BD of Poly & PCM VI.

The `FreqInv` VI performs frequency inversion on the input samples by negating every odd sample of every odd subband. This inversion is done to compensate for the negation of values during the MDCT stage.

The `MDCT & Wvec` VI, see Figure 12-30, computes the MDCT array and generates the windowed vector, indicated by `Wvector`. The global variable `Vvector` plays a circular buffer role where the elements are shifted circularly and a new element is inserted at the index specified by `bOf`. Sixteen 32-point arrays of the replaced `Vvector` are extracted to form the pre-windowed vector. This vector is then multiplied with the window function to form the `Wvector` as illustrated in Figure 12-30.

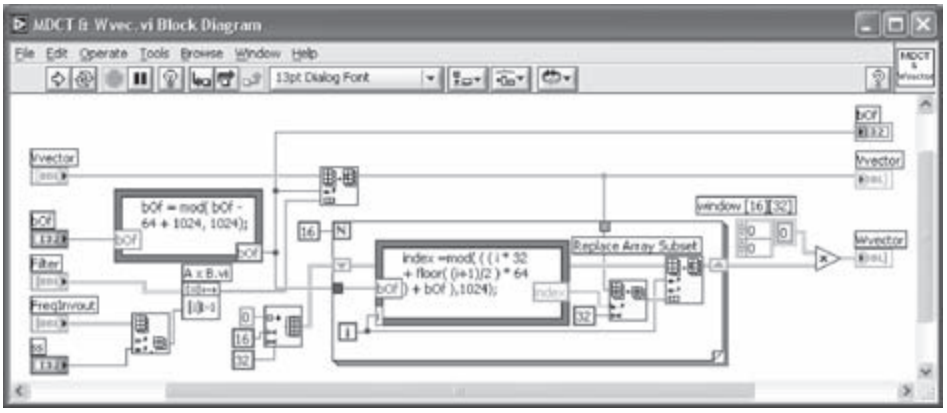

Figure 12-30: BD of MDCT & Wvec VI.

The Add and PCM VI adds the elements of the Wvector. The addition of the 16×32 Wvector takes place column wise so that the sixteen values in each column are added to generate one value for the final output array as previously illustrated in Figure 12-11. This operation is repeated eighteen times to form the final 18×32 output array. This array is then converted to the I16 format for the final PCM samples.

L12.2.12 PCM Out

This VI writes the output of the Poly & PCM VI to a file. It uses the Build Path function (**Functions → All Functions → File I/O → Build Path**) and the Write to I16 File VI (**Functions → All Functions → File I/O → Binary File VIs → Write to I16 File**). The controls are connected to the appropriate functions as illustrated in Figure 12-31.

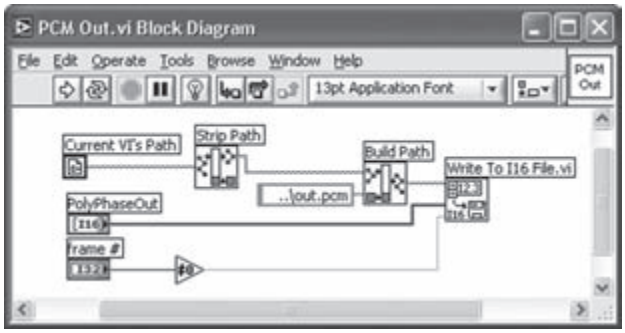

Figure 12-31: BD of PCM Out VI.

L12.2.13 MP3 Player

The MP3 Player VI is the top-level VI of the MP3 decoder system. It integrates all the discussed VIs. A new VI is opened and the MP3 Read VI is placed in its BD. The file path of the MP3 file is obtained from the path terminal of the File Dialog function (**Functions → All Functions → File I/O → Advanced File Functions → File Dialog**). The output of the MP3 Read VI is wired to an Array Size function and the MP3 Frame Info VI. The output of the Array Size function is wired to the MP3 frame Info VI. Indicators for all the outputs of the MP3 Frame Info VI are created to display the MP3 file information on the FP. A 1024 point array, Buffer, and the global variable used in the IMDCT and Poly & PCM VIs are initialized with zeros. The bufferOffset global variable in the Poly & PCM VI is initialized to 64.

Finally, a For loop is created and the Frame # output of the MP3 Frame Info VI is wired to the loop count N. All the VIs are wired together as shown in

Figure 12-12. Two `Shift Registers` are located in the `For Loop`. The initialized `Buffer` array is wired to one of the shift registers and a constant '0' to the other.

Running the VI brings up a file dialog to choose an MP3 file to play. The decoded PCM file can be played by any audio application software, such as Adobe® Audition™. Figure 12-32 displays the FP of the MP3 Player VI.

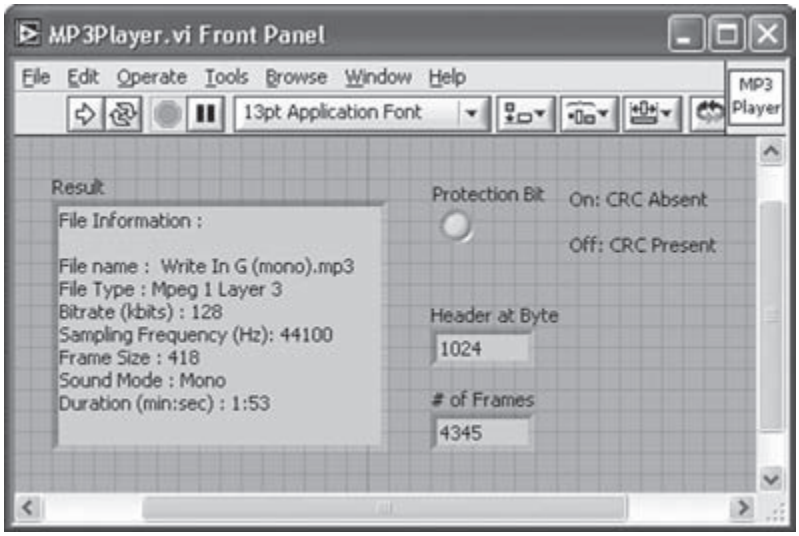

Figure 12-32: FP of MP3 Player VI.

L12.3 Modifications to Achieve Real-Time Decoding

By profiling the VI built in the previous section, one can find that four of the subVIs account for over 93% of the execution time. The time consuming subVIs along with their time usage (in percentage) are listed in Table 12-2. The overall decoding time for an examined file of 21 seconds takes 58 seconds, which means not having a real-time throughput. The following VI modifications are thus carried out in order to decode MP3 files within their play times.

Table 12-2: Processing time percentages for the time consuming VIs.

SubVI	Relative time consumption (%)
Huffman Decode VI	47.34
Requantization VI	4.50
IMDCT VI	8.90
Polyphase VI	32.28
Other VIs	6.8

L12.3.1 Huffman Decode

In the above version of this VI, the section for the codeword table look-up in the Huffman tables is quite slow. Also, the search algorithm employed is not efficient, which results in a very slow execution. In this section, a faster way of implementing the Huffman Decoding block is presented. The outer structure of the VI remains the same. The changes made are in the `search loop`, the `Big Value Search` VI, and the `Count1 Search` VI.

In the `search loop`, the search is done at bit level. However, a faster version can be obtained by performing the search on multiple bits of Huffman coded data. The length of the bits to be extracted is determined by the predefined codeword lengths in the selected table. As a result, all the redundant searches for codewords whose lengths are not specified in the selected table are avoided. A `Case Structure` is created where its contents are arrays corresponding to the codeword lengths of the tables. The array for a table is selected using the `tableout` parameter. Figure 12-33 illustrates the implementation of this new `search loop`.

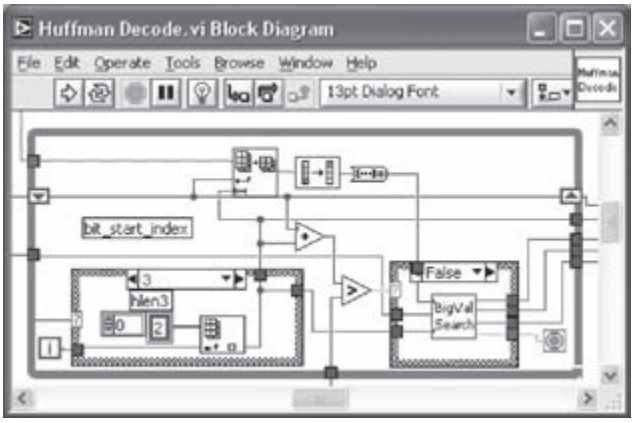

Figure 12-33: Faster search loop for Huffman Decode VI.

The `Big Value Search` VI creates the 2D Huffman tables consisting of integers instead of strings, thereby improving the speed of this VI by a factor of ten. This VI uses a new style of look-up tables (LUTs), whose description follows next.

Divide the Huffman tables such that codewords with the same length occur together. Construct a 2D array with two columns and rows given by the number of codewords of the selected length. The elements are specified by the decoded values, `X` and `Y`. Construct another 1D array with the integer values of the codewords. For example,

the codeword '111001' is represented in the array as 57. Combine the two arrays into a cluster and form a 2D array of all such clusters. The clusters in the 2D array are indexed by `table_select` and `length`. As a result, the search is only done on the table specified by `table_select` and `length`. This makes the VI run quite fast. Also, create a 1D array whose elements are `Linbits` for each table.

After indexing the correct table, codeword searching is carried out using the `Search 1D Array` VI. This VI returns '–1' for no match. Figure 12-34 shows the BD of the `Big Value Search` VI.

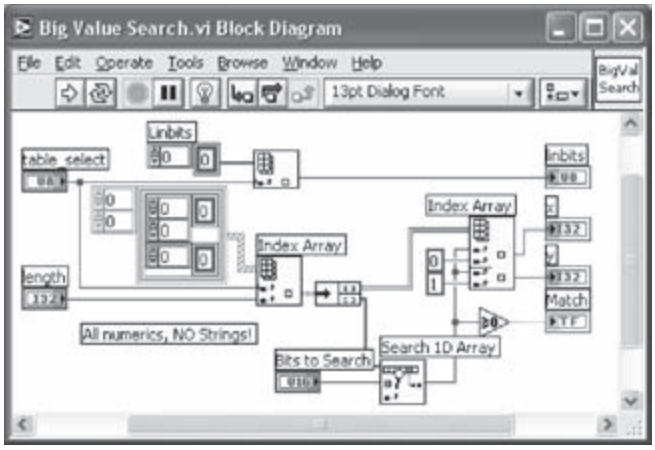

Figure 12-34: BD of faster Big Value Search VI.

Now, create three controls: `table_select`, `length`, and `Bits to search`. Next, place three functions: `Index Array`, `Unbundle`, and `Search 1-D Array`. Implement the `Big Value Search` VI using these functions as illustrated in Figure 12-34. Link the `Big Value Search` VI to the `Big Value Sign` VI. Make the same changes in the `search loop` of the count1 region as done in the big_value region earlier. Similarly, place the `Count1 Search` VI and repeat the above steps to construct a cluster of LUTs. The difference is that the 2D array in the cluster has four columns. The four columns correspond to the decoded values `V`, `W`, `X`, and `Y`. Search for the bits as before and index out elements from the 2D array at the row index given by the output of the `Search 1-D Array` function and columns 0, 1, 2, and 3. Place all the indicators and complete the updated version of the `Huffman Decode` VI by linking this VI to the `Count1 Sign` VI.