Part II

Appendices

A

Complexity Analysis for DSP-Algorithms

In this appendix an estimate for the complexity of the DSP related parts of the Layer III algorithms will be conducted. The estimate searched for is a worst-case estimate, and therefore the complexity analysis will be based on a stereo audio signal sampled at 48 kHz. The complexity analysis is required in order find a MIPS estimate to be used for selecting an adequate DSP to function together with the BSP in a Layer III decoder.

The appendix is subdivided into smaller sections, each applying for one sub algorithm. All estimates in the small sections are based on decoding one frame only. The appendix concludes with a MIPS index in order to decide which requirements to set for a DSP. In Figure A.1 a block diagram illustrates the algorithms to be executed in a DSP.

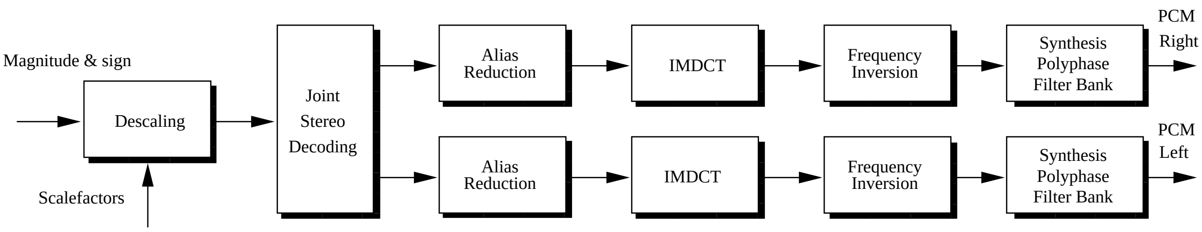

Figure A.1: *Block diagram of the algorithmes to be executed in a DSP.*

Reading Serial Port

When decoding a Layer III coded audio signal sampled at 48 kHz, $41\frac{2}{3}$ frames arrives per second. The BSP will therefore transmit 96,000 quantized frequency lines and 96,000 scalefactor plus additional system information, that is approximately 200,000 words per second. The DSP will therefore receive 4,800 words per frame, each requiring to be stored in memory. Most DSP's can offer zero overhead reading of the serial receive register, if a DAG register is dedicated to address memory for storing the incoming words. Therefore only one instruction can be assigned for storing each input word.

Operation type	Always applied
STORE	4,800

Subdivide Quantized Frequency Lines and Scalefactors

Both the quantized frequency lines and the scalefactors received from the BSP requires preprocessing. The frequency lines are in signed magnitude number format. It is required to divide each quantized frequency line into a sign part and a magnitude part in order to calculate the following descaling. The scalefactors received are composed by the scalefactor itself and *pretab* or `subblock_gain`, depending on the block type. Assigning 4 shift operations to process each of the 4,608 received scalefactors and quantized frequency lines seems a reasonable estimate.

Operation type	Always applied
SHIFT	9,000

Descaling

Descaling of the quantized frequency lines is conducted using Equation (A.1). This equation adapted to the reordered quantized frequency lines and goes for both long and short blocks. The equation must be calculated 1152 times for each channel, in order to descale data within a frame. The results of the operation count are shown below.

$$xr[i] \quad = \quad \mathrm{sign}(is_{ro}[i]) \cdot \mathrm{abs}(is_{ro}[i])^{\frac{4}{3}} \cdot \frac{2^{\frac{1}{4}(\texttt{global\_gain}-210-8\cdot sbg_{ro}[i])}}{2^{(scalefac\_\ multiplier\cdot sf_{ro}[i]+\texttt{preflag}\cdot pt_{ro}[i])}} \tag{A.1}$$

Operation type	Always applied
MUL	13824
DIV	2304
ADD	2304
SUB	4608
2^x	2304
$y^{4/3}$	2304

Joint stereo decoding

Two stereo redundancy coding methods are used in Layer III, MS stereo and intensity stereo. The formulas used for decoding the frequency lines for each channel are reproduced in Equation (A.2) and Equation (A.3). In the table below the results of the operation count for each method are shown. The complexity of the intensity stereo mode depends on the number of scalefactor bands, and therefore the complexity is calculated for short blocks only. Also, the tangents calculation will be based on table look up. Using look up tables for this calculation will be reasonable because only 36 values are required. Both of the equations most be executed 1152 times in order to process a frame.

$$L_i = \frac{M_i + S_i}{\sqrt{2}} \quad \text{and} \quad R_i = \frac{M_i - S_i}{\sqrt{2}} \tag{A.2}$$

$$\begin{aligned} &is\_ratio = \tan(\texttt{is\_pos}_{sfb} \cdot \frac{\pi}{12}) \\ &R_i = L'_i \cdot \frac{1}{1 + \texttt{is\_ratio}_{sfb}} \quad \text{and} \quad L_i = L'_i \cdot \frac{\texttt{is\_ratio}_{sfb}}{1 + \texttt{is\_ratio}_{sfb}} \end{aligned} \tag{A.3}$$

Operation type	MS stereo	Intensity stereo
MUL	0	1152
DIV	2304	1152
ADD	1152	1152
SUB	1152	0

Alias Reconstruction

The alias reconstruction was done by conducting a series of butterfly computations for frequency lines within long blocks. The butterfly computations implies the use of 16 coefficients $cs[i]$ and $ca[i]$, all defined in [ISO/IEC 11172-3, 1993]. In an implementation these constants will be used in a look up table. In Algorithm A.1 the butterfly computations are defined and a operation count is summarized in the table below. The algorithm must be executed four times for a frame.

Algorithm A.1 *Algorithm for reconstruction of alias in one channel in one granule*

for $j = 1$ **to** 32 **do**
  for $i = 0$ **to** 7 **do**
    $xar[18j - 1 + i] = xr[18j - 1]cs[i] - xr[18j + i]ca[i]$
    $xar[18j + i] = xr[18j + i]cs[i] + xr[18j - 1 - i]ca[i]$
  end for
end for

Operation type	Long blocks
MUL	2048
ADD	512
SUB	512

IMDCT

The complexity estimates for the IMDCT calculations will be performed for both short blocks and normal blocks. Start and stop blocks will not be included because they represent an intermediate complexity. Starting with the short blocks, the IMDCT calculations imply the IMDCT itself in order to calculate 3 sets of 12 subband samples each. Next the subband is reconstructed by first windowing each of the 3 sets and then overlapping and adding. The long blocks requires the IMDCT and a post windowing in order to reconstruct a subband. Common to all 32 reconstructed subbands is that they must be overlapped and added with the previous subbands. In the following the sequence of operations to perform for both block types are listed, and a operation count is tabulated below.
IMDCT short blocks, to be calculated for all 384 short blocks:

$$x_i = \sum_{k=0}^{5} X_k \cdot \cos\left(\frac{\pi}{24}(2i+7)(2k+1)\right), \quad \text{for } i = 0 \text{ to } 11 \tag{A.4}$$

Windowing short blocks, to be calculated for 384 short blocks:

$$\begin{aligned} y_i &= x_i \cdot w_i, \quad \text{for } i = 0 \text{ to } 11 \\ w_i &= \sin\left(\frac{\pi}{12} \cdot \left(i + \frac{1}{2}\right)\right) \end{aligned} \tag{A.5}$$

Short blocks overlap and add, to be calculated for 128 trios of short blocks:

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

IMDCT long blocks, to calculated for 128 long blocks:

$$x_i = \sum_{k=0}^{17} X_k \cdot \cos\left(\frac{\pi}{24}(2i+19)(2k+1)\right), \quad \text{for } i = 0 \text{ to } 35 \tag{A.7}$$

Windowing long blocks, to be calculated for 128 long blocks:

$$\begin{aligned} z_i &= x_i \cdot w_i, \quad \text{for } i = 0 \text{ to } 35 \\ w_i &= \sin\left(\frac{\pi}{36} \cdot \left(i + \frac{1}{2}\right)\right) \end{aligned} \tag{A.8}$$

Previous block overlap and add, to be calculated twice for 128 long blocks or trios of short windows:

$$\begin{aligned} S_i &= z_i + s_i\ , \quad \text{for } i = 0 \text{ to } 17 \\ s_i &= z_{i+18}\ , \quad \text{for } i = 0 \text{ to } 17 \end{aligned} \tag{A.9}$$

Operation type	Short blocks	Long blocks
MAC	27,648	82,944
MUL	4608	4608
ADD	5376	2304

Frequency Inversion

Prior to the synthesis polyphase filtering a frequency inversion is conducted. The algorithm to select the proper frequency lines to apply the sign changes to is shown in Algorithm A.2 and must be applied for two channels. For the operation count it will be assumed that the frequency inversions are conducted by multiplying with -1. The operation count is shown in the table below.

Operation type	Always applied
MUL	576

Synthesis polyphase filtering

The last processing required for reconstructing the 2×1152 PCM samples described by a frame is conducted in this block. The algorithm to execute is listed in Algorithm A.3 and must be executed 36 times per frame. An operation count for lines 5, 9, 13 and 16 is shown in the table within this subsection. The remaining lines are ignored because they only requires pointer updating.

Algorithm A.2 *Algorithm for frequency inversion*

for $sb = 0$ **to** 31 **do**
  for $i = 0$ **to** 35 **do**
    if ODD(i) AND ODD(sb) **then**
      $S_{i,sb} = -S_{i,sb}$
    end if
  end for
end for

Algorithm A.3 *Algorithm for calculation of the synthesis polyphase filter bank*

for $i = 1023$ **downto** 64 **do** //Shifting
  $V[i] = V[i-64]$
end for
for $i = 0$ **to** 63 **do** //Matrixing
  $V[i] = \sum_{k=0}^{31} N_{ik} S_k$
end for
for $i = 0$ **to** 7 **do** //Build a 512 values vector U
  for $j = 0$ **to** 31 **do**
    $U[64i + j] = V[128i + j]$
    $U[64i + 32 + j] = V[128i + 96 + j]$
  end for
end for
for $i = 0$ **to** 511 **do** //Window by 512 coefficients
  $W[i] = U[i]D[i]$
end for
for $j = 0$ **to** 31 **do** //Calculate 32 PCM samples
  $p[i] = \sum_{i=0}^{15} W[j + 32i]$
end for

Operation type	Always applied
MAC	73,728
MOVE	18,432
MUL	18,432
ADD	17,280

Digital to analog conversion

The audio reproduction requires that the DSP writes two PCM samples to a dual-channel D/A-converter every 21 ms. For a frame 2304 PCM samples must be written. Zero overhead is assumed for writing to a serial transmit register, if a DAG register is dedicated to address memory for loading the PCM samples to be transmitted. Therefore only one instruction can be assigned for storing each input word. The operation count for a frame is shown in the table below.

Operation type	Always applied
STORE	2304

A.0.1 Summary on DSP Complexity Analysis

Based on the instruction count conducted in the preceding subsections a total operation count can be found. The operation count wanted must apply for the worst case situation. A survey on the operation counts reveiles that the worst case situation will appear for a frame containing long windows only and coded in MS stereo. In the table below all the operations to be executed for one frame are summed up.

Operation type	Worst Case
MAC	156,672
MUL	39,488
ADD	23,552
SUB	6,272
SHIFT	9,000
2^x	2,304
$y^{4/3}$	2,304
STORE	7104

In the table above two operations requiring extra attention are present, 2^x and $y^{4/3}$, x is a rational number and y is an integer. Both operations are not supported directly by any general purpose DSP. A software implementation of both operations is therefore required. [Schiøning and Simonsen, 1996] suggests methods for both operations each requiring only few typical DSP instructions. Calculating 2^x is shown in Equation (A.10). In terms of DSP instructions 2^x can be calculated using a shift, a load and a multiplication.

$$2^x = 2^{\text{integer}(x)} \cdot 2^{\text{fraction}(x)} = \text{SHIFT}(2, \text{integer}(x)) \cdot \text{LUT}(\text{fractional}(x)) \tag{A.10}$$

Calculating $y^{4/3}$ is done by first performing a Taylor series expansion as shown in Equation (A.11). According to [Schiøning and Simonsen, 1996] $f(y_0)$ and $f'(y_0)$ can be found using two and a single table look up, respectively. y_0 requires maximally 5 shift operations and tests. Having decided the contents of Equation (A.11) the equation only requires an addition, a subtraction and a multiplication.

$$f(y) \approx f(y_0) + f'(y_0)(y - y_0), \quad \text{where} \quad f(y) = y^{4/3} \tag{A.11}$$

Having established an instruction count for one frame, the MIPS number can be decided by multiplying with total number of frames arriving per second. When assuming a sample rate of

Instruction type	Worst Case
MAC	6,528,000
MUL	2,029,333
ADD	1,077,333
SUB	357,333
SHIFT	471,000
LOAD	384,000
STORE	296,000
Total:	11,143,000

48 kHz, $41\frac{2}{3}$ frames arrives per second. Multiplying this number with the contents of the table above results in the MIPS number shown at the bottom line in the table below.

The approximately 11 MIPS shown in the table above does not count for overhead. The overhead will, however, not contribute significantly to the complexity because the largest parts of the computations, e.g. polyphase filtering and IMDCT, are vector/matrix multiplications.

B

Bitstream Format for the MPEG/Audio Layer III.

In this appendix the format for the bitstream used in MPEG/Audio Layer III will be described. An Layer III frame consists of information to construct 1152 samples. The 1152 samples are subdivided into two granules each of 576 samples. A Layer III frame is divided into five parts, i.e. header, Cyclic Redundancy Code (CRC), Side information, Main data and Ancillary Data as shown in Figure B.1. Each of the five parts of the frame will be described in turn.

Header	**CRC**	**Side Information**	**Main Data**	**Ancillary Data**
(32)	(0,16)	(136,256)		

Figure B.1: Frame format of a MPEG/Audio Layer III frame. Note that the main data is not necessarily linked to this frame, which will be explained later.

B.1 Header

An MPEG/Audio bitstream is identified by frame headers. The header of the MPEG/Audio Layer III is composed of 32 bit, 12 bits for synchronization and 20 bits containing system information. The header format is shown Figure B.2.

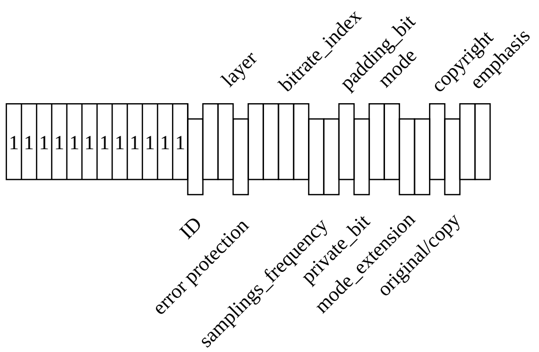

Figure B.2: *MPEG/Audio Layer III header format.*

`synchronization`: The 12 synchronization bits equals the bit string '1111 1111 1111'. The syncronization bits are used for detecting the beginning of a frame.

`ID`: A single bit which represents the ID of the used algorithm. '1' indicates the ISO/IEC 11172-3 Audio is used.

`layer`: Two bits which indicates whether Layer I, II or III is used.

`protection_bit`: This bit indicates whether or not a check word has been added to the bitstream in order to enable CRC check.

`bitrate_index`: Four bits that indicate the bitrate used. The four bits enable variable bitrates during the transmission of the bitstream. This is an exclusive feature for Layer III. The maximum bitrate possible in Layer III is 320 kbit/s for stereo signals.

`sampling_frequency`: Indicates the sampling frequency.

`padding_bit`: Indicates whether an additional slot (one byte) has been added to the frame in order to adjust the mean bitrate to the sampling frequency.

`private_bit`: Bit for private use.

`mode`: Two bits determining the mode, i.e. stereo, joint_stereo (intensity_stereo and/or ms_stereo), dual channel and single channel mode.

`mode_extension`: The two mode extension bits inform which type of joint stereo coding method is used, i.e. intensity_stereo and/or ms_stereo.

`copyright`: Indicates if a copyright on the ISO/IEC 11172-3 bitstream exists.

`original/copy`: This bit is '1' if the bitstream is an original and '0' if it is a copy.

`emphasis`: Indicates which type of pre-emphasis that must be used.

B.2 CRC

If the protection bit is set, a 16 bit CRC word is appended to the header.

B.3 Side Information

The side information consists of 17 bytes for a single channel bitstream and 32 bytes otherwise. The information contained in the side information consists of four parts, i.e. `main_data_begin` pointer, side information for both granules, side information for granule 0 and side information for granule 1.

`main_data_begin`: The main_data is not necessarily located next to the side information. main_data_begin is a 9 bit pointer indicating the location of the beginning byte of the main_data belonging to the current frame, see Figure B.3. The location is specified as a negative offset from the first byte of the synchronization word.

B.3.1 Side Information Common to both Granules

`private_bits`: The number of `private_bits` depends on the number of channels (5 for single and 3 for stereo). The number of bits allocated for `private_bits` is user defined.

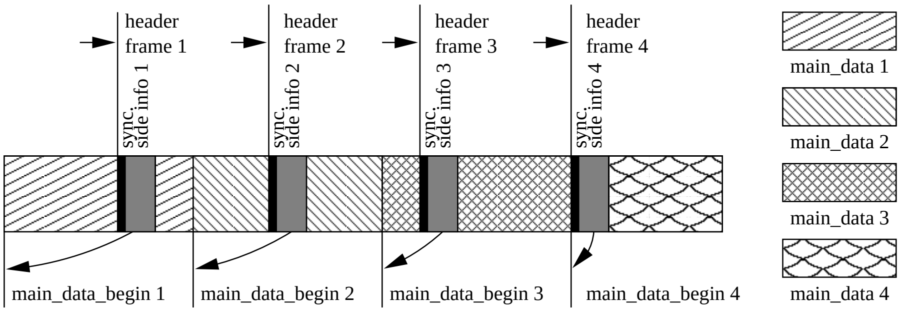

Figure B.3: *An example Layer III bitstream organization. It is shown how the main_data belonging to one frame can be linked to another frame with the use of the pointer* `main_data_begin`*. For instance the pointer for the main_data belonging to frame 4, i.e.* `main_data_begin` *4 equals zero. That means that main_data 4 starts immediately after the side information of frame 4. The number of bytes contained in synchronization and side information is not included in the* `main_data_begin` *pointer.*

main_data_begin	private_bits	scfsi	Side_info gr. 0	Side_info gr. 1
(9)	(5,3)	(4,8)	(59,118)	(59,118)

Figure B.4: Bit allocation in the side information. The numbers in the parenthesis correspond to the number of bit allocated in single channel and stereo respectively.

`scfsi`: The contents of the scale factor selection information (`scfsi`) variable determines whether one scale factors are sent for each granule or for both granules, per channel. Four bits per channel are transmitted, a single bit belonging to each group of scalefactor bands. A zero for a specific group of scalefactor bands means that scalefactors are transmitted for each granule for that particular group of scalefactor bands. A one indicates the same scalefactors are used and hence only the scalefactors for the first granule are transmitted.

B.3.2 Side Information for Each Granule

`par2_3_length`: Denotes the number of main_data bits allocated for scalefactors and Huffmancode data. 12 bits per granule are used for the `par2_3_length` variable in single channel mode and 24 bits in stereo mode. Because the number of bits used for the side information is constant, `par2_3_length` can be used to calculate the location of the beginning of the next granule.

`big_values`: After quantization the encoder orders the 576 quantized MDCT samples in a predetermined order. The encoder then orders the values in three following partitions: rzero, count1 and big_values. The first partition rzero, located in the high frequencies, consists of pairs of zeros. The second partition count1 consists of quadruples of either 1, 0 or -1. The last partition big_value extends towards zero frequency and is composed of pairs of bits. The `big_values` variable indicates the size of the big_value partition each granule. 9 bits are allocated for `big_values` in single channel mode and 18 bits in stereo mode.

`global_gain`: Contains information about the step size used in the quantizer where the quantization is done logarithmically. The variable `global_gain` uses 8 bits in single channel mode and 16 bits in stereo mode.

`scalefac_compress`: `scalefac_compress` is a 4 bit variable (in single channel mode) transmitted for each granule, which determines the number of bits used for the transmission of scalefactors. Each granule is partitioned into either 12 or 21 scalefactor bands depending on which type of window is used. These scalefactor bands are again divided into two groups i.e., 0-10 and 11-20 in case of long windows and 0-5 and 6-11 in case of short windows. The `scalefac_compress` variable is used to index a table given in [ISO/IEC 11172-3, 1993], which returns two numbers *slen1* and *slen2*. *slen1* and *slen2* are the number of bits assigned to the first and second group of scalefactor bands respectively. *slen1* and *slen2* can vary between 0 and 4 bits.

`windows_switching_flag`: The `windows_switching_flag` variable signals whether other than a normal window (type 0) is used. The value of `windows_switching_flag` influences the succeeding 22 bits (single channel mode). If the `windows_switching_flag` is set i.e., equal to one the bits in Figure B.5 (b) is used, otherwise the bits in Figure B.5 (c) is appended.

par2_3_length	**big_values**	**global_gain**	**scale_fac_compress**	**windows_sw_flag**
(12,24)	(9,18)	(8,16)	(4,8)	(1,2)

(a)

block_type	**mixed_block_flag**	**table_select**	**subblock_gain**
(2,4)	(1,2)	(10,20)	(9,18)

(b)

table_select	**region0_count**	**region1_count**
(15,30)	(4,8)	(3,6)

(c)

preflag	**scalefac_scale**	**count1table_select**
(1,2)	(1,2)	(1,2)

(d)

Figure B.5: Bit allocation of side information. The numbers in parentheses correspond to the number of bits allocated single channel and stereo respectively. The bits in (a) are always qtqaqsmitted. The following 22 bits(in single channel mode) depends on the value of `windows_switching_flag`. If the flag is set the the bits in (b) are appended else the the bits in (c) are appended. The last three flags (d) are always appended after (c) or (b).

`block_type`: In single channel mode block_type is two bit variable, indicating the window type used in a particular granule.

`mixed_block_flag`: The `mixed_block_flag` indicates that different types of windows are used in the lower and higher frequencies. If `mixed_block_flag` is set , i.e. equal to one, the two lowest subbands are transformed using a normal window i.e. `block_type`=0 and the remaining 30 subbands are transformed using the window specified by the `block_type` variable.

`table_select`: `table_select` enables the use of different Huffman code tables. 5 bits per region per channel are used to encode which of the 32 possible tables to select.

`subblock_gain`: Enables a gain by a factor 4 for one subblock. The 3 bit variable is only used together with short windows indicated by `block_type`=2.

`region0_count`: To improve performance of the Huffman coder the partion described by `big_values` is further subdivided into three regions named region0, region1 and region2. Each region can be coded with a separate Huffman code table. `region0_count` specifies the boundary between region0 and region1.

`region1_count`: Indicates the boundary between the region 1 and the region 2 of the `big_values` partition. The three bit variable counts one less than the number of scalefactor bands in region 1.

`preflag`: The `preflag` bit is used to indicate additional amplification in the high frequencies of the quantized values. `preflag` is set to a value, specified in [ISO/IEC 11172-3, 1993], which is added to the scalefactors indicating that an preemphasis has occurred. For `block_type`=2 (short blocks) `preflag` is not used.

`scalefac_scale`: The scalefactors are logarithmically quantized with a step size of 2 or $\sqrt{2}$ depending on the value of `scalefac_scale`.

`count1_table_select`: This flag indicates which of two possible Huffman code tables was used to code the partition count1.

B.4 Main Data

The main data part of a Layer III bitstream is composed of scalefactors, Huffman coded bits and ancillary data.

B.4.1 Scalefactors

The scalefactors are used to color the quantization noise. Scalefactors are transmitted for every group of frequency lines (scalefactor bands). Whether the scalefactors are sent for each scalefactor band for each granule depends on the value of `scfsi` for the particular scalefactor band. The number of scalefactors actually sent per scalefactor band also depends on whether short or long windows or a combination of these was used in the encoder. In summary the number depends on the values of both `block_type`, `windows_swithing_flag` and `mixed_block_type`.

B.4.2 Huffmancodebits

The frequency lines of each granule was divided into 3 partitions as previous described (big_value, count1 and rzero). The rzero partition is not encoded, since this partition only contain quantized values equal to zero.

The count1 partition contains quadruples of quantized values with magnitude less than or equal to zero. The quadruples are coded using one of two possible tables specified by `count1table_select`. For each nonzero value a sign bit is appended.

The big_value partition is further divided in to three regions, which are coded separately. Within the big_value partition pairs of quantized frequency lines with absolute value less than 15 are encoded directly. If the absolute value is equal to or greater than 15 an extra field is appended, i.e. an ESC-value. Again for each nonzero value a sign bit is appended.

B.4.3 Ancillary Data

The ancillary data is optional and if used they number of ancillary bits indicate the bits available in the bitstream when the bits allocated for header, CRC and audio data are subtracted. For Layer III the number of ancillary bits correspond to the distance between the end of the Huffmancodebits and the location in the bitstream where the next frame's `main_data_begin` pointer points to.

C

Profiling Graphs

This appendix contains the profiling graphs used in Chapter 12. Figure C.1 and Figure C.2 shows the distribution in instruction demands between the major blocks in the Layer III decoder, when decoding stream1 and stream3. Figure C.3, Figure C.4 and Figure C.5 is the distribution in instructions during the Huffman decoding. The profiling results from decoding stream2 can be seen in Chapter 12.

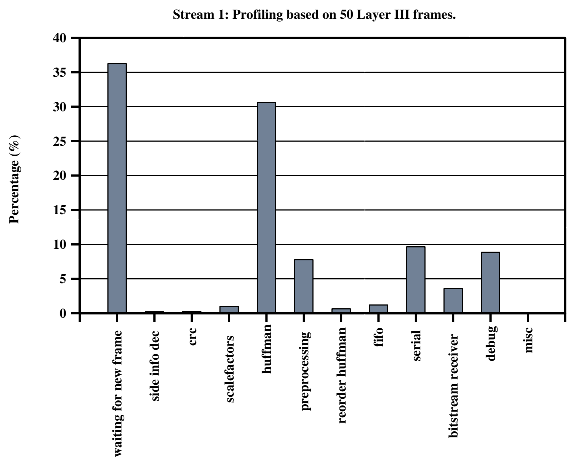

Figure C.1: *Profiling result from decoding Stream1.*

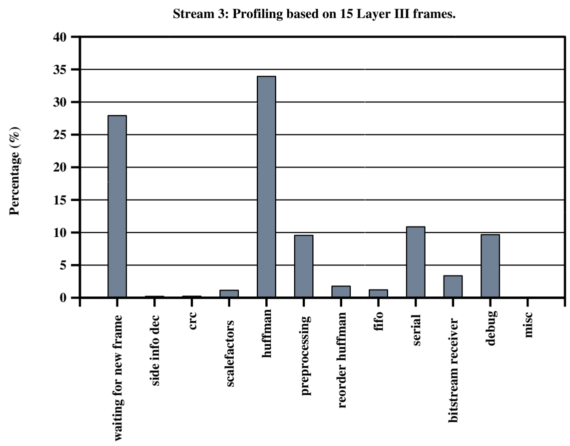

Figure C.2: *Profiling result from decoding Stream3.*

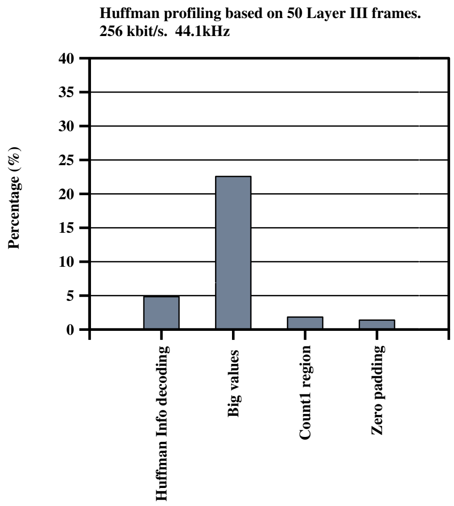

Figure C.3: *Profiling of the Huffman decoding decoding in Stream1.*

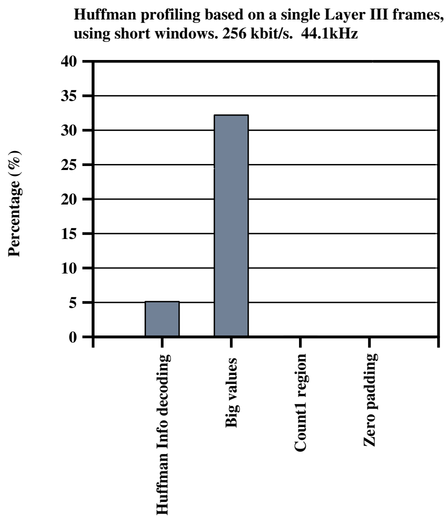

Figure C.4: *Profiling of the Huffman decoding decoding in Stream2.*

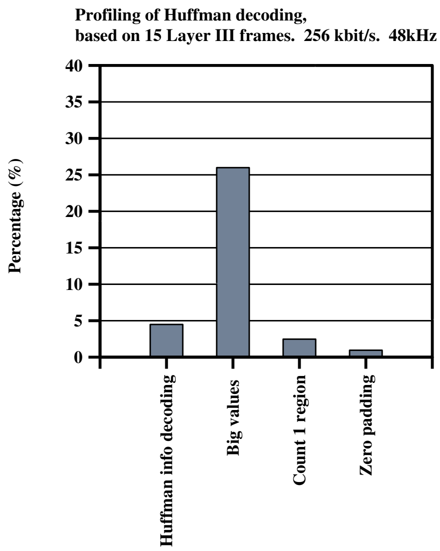

Figure C.5: *Profiling of the Huffman decoding decoding in Stream3.*

D

Debugging Samples

This appendix presents a two examples of the use of the debug facility provided by the BSP.

Example 1 In the first example a frame from one of the compliance bitstreams are decoded. Figure D.1 shows the output from the existing decoder and Figure D.2 shows the result calculated by the BSP. When using the existing decoder the variables displayed can be determined using command line options. Some choices to display is the side information, Huffman data etc. The first five lines in Figure D.1 are always displayed and contain the all the header variables. The fifth line contains some decoded information about the frame such as the layer number, sampling frequency, bitrate and whether it is a mono or a stereo frame. The example frame is single channel Layer III frame with a sampling frequency of $44, 1kHz$ and a bitrate on 64 kbit/s. The decoder was provided with options to display the side information and the decoded Huffman data. Figure D.1 shows the decode side information and the decoded Huffman values for the first granule. The side information provides important information about the frame, which can be used in the debugging of the programmed algorithms. For instance it can be seen from the `window_switching_flag` and the `block_type` that short window types were used in the frame. The informations provided by the existing decoder can now be used for comparing the results calculated by the BSP. A debug port was used to display informations about the decoded data from the BSP. This information is shown in Figure D.2. The functional validation can now be performed by a simple comparison of the output from the two decoders. Besides displaying the header, side information, scalefactors and Huffman data some additional information is displayed. The output from the BSP is displayed different as the output from the existing decoder. The displayed output from the BSP always contain all possible variables from a frame even though they are not provided in the current frame. This means that only the values actually displayed by the existing decoder could be compared.

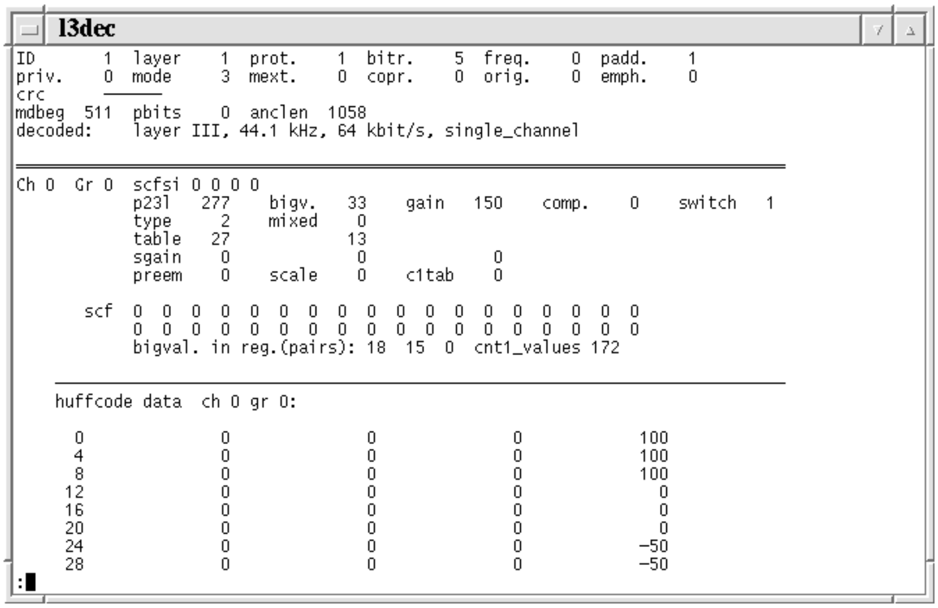

Figure D.1: *The output from the existing decoder.*

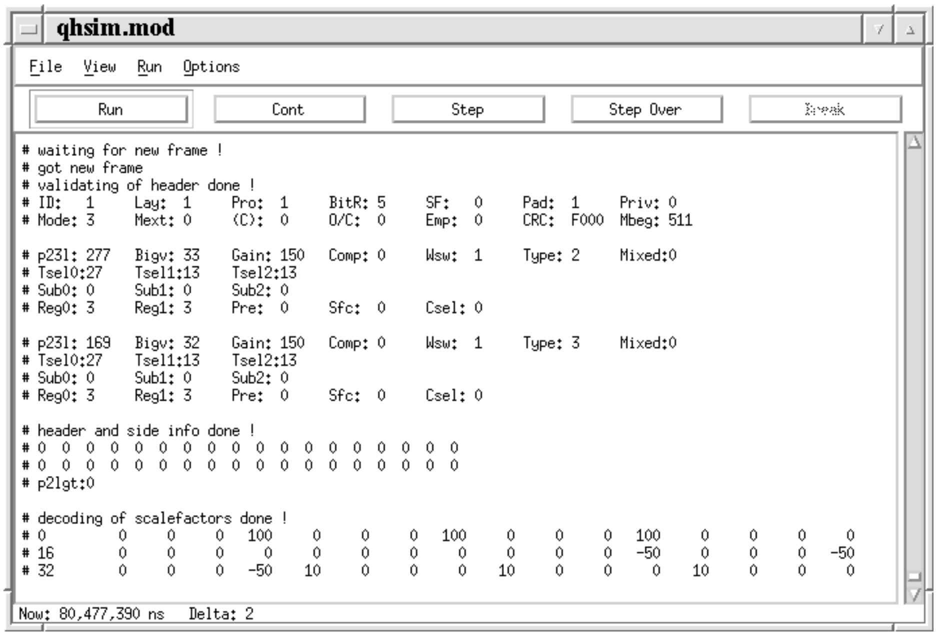

Figure D.2: *The output from the designed decoder.*

Example2 The second example shows how the trace interrupt can be used for debugging. For illustration purposes an small assembly program has been written, the assembly code is shown in page 168. The assembly code will in the following be shortly described. The first 14 lines constitutes the vector table, where the addresses of the interrupt routines. In the example the addresses of the trace interrupt routine and the fifo interrupt routine are specified. Then the stack pointer is set up and the two interrupts fifo and trace are enabled by asserting the corresponding bits in the status register. The two nop's inserted are necessary because the trace routine requires two instructions to starts. The example program simply decrements one counter g0, increments another g1 and performs left shift on the initial values placed in r0. When the counter g0 reaches zero a resetfifo is executed. The resetfifo operation makes the fifo controller initiate four reads, 3 for filling the fifo register and 1 for filling the fifo buffer. The reason for having the resetfifo operation is that it illustrates how the interrupt work. The trace of this example assembly program is shown in Figure D.3. The traceall routine displays all the registers in the register file. Each row in Figure D.3 contains from left: PC - the program counter, the two status flags zero - Z and less than zero - LZ , CI - current interupt, EI - enabled interrupt, SP - the stack pointer, G0-G1 and R0-R7 the registers and BSR the register to which the bitstream receiver is mapped.

```
.word 0                 ; External 1
.word 0                 ; External 2
.word 0                 ; External 3
.word 0                 ; External 4
.word 0                 ; External 5
.word 0                 ; External 6
.word 0                 ; External 7
.word 0                 ; External 8
.word traceall          ; Trace
.word 0                 ; Overflow in incoming BS
.word 0                 ; Incoming BS
.word 0                 ; Serial
.word 0                 ; Overflow
.word fifo              ; Fifo

stackpointer_adr:
        .word stackpointer
fifo_adr:
        .word fifo

#set address = $20
start:
        g0 = mem(stackpointer_adr)
        sp = g0
        r7 = fifo
        sr = (2^0)+(2^5)
        nop
        nop

        g0 = 4
        g1 = 1
        r0 = 1

loop_start:
        r0 = r0 << 1
        g1++
        g0--
        jumpnz loop_start
        resetfifo
        nop

lable_deadlock:
        jump lable_deadlock

fifo:
        fifo = mem(r7)
        mask(r7++,128)
        rti

#include ../intr/trace_debug

stackpointer:
```

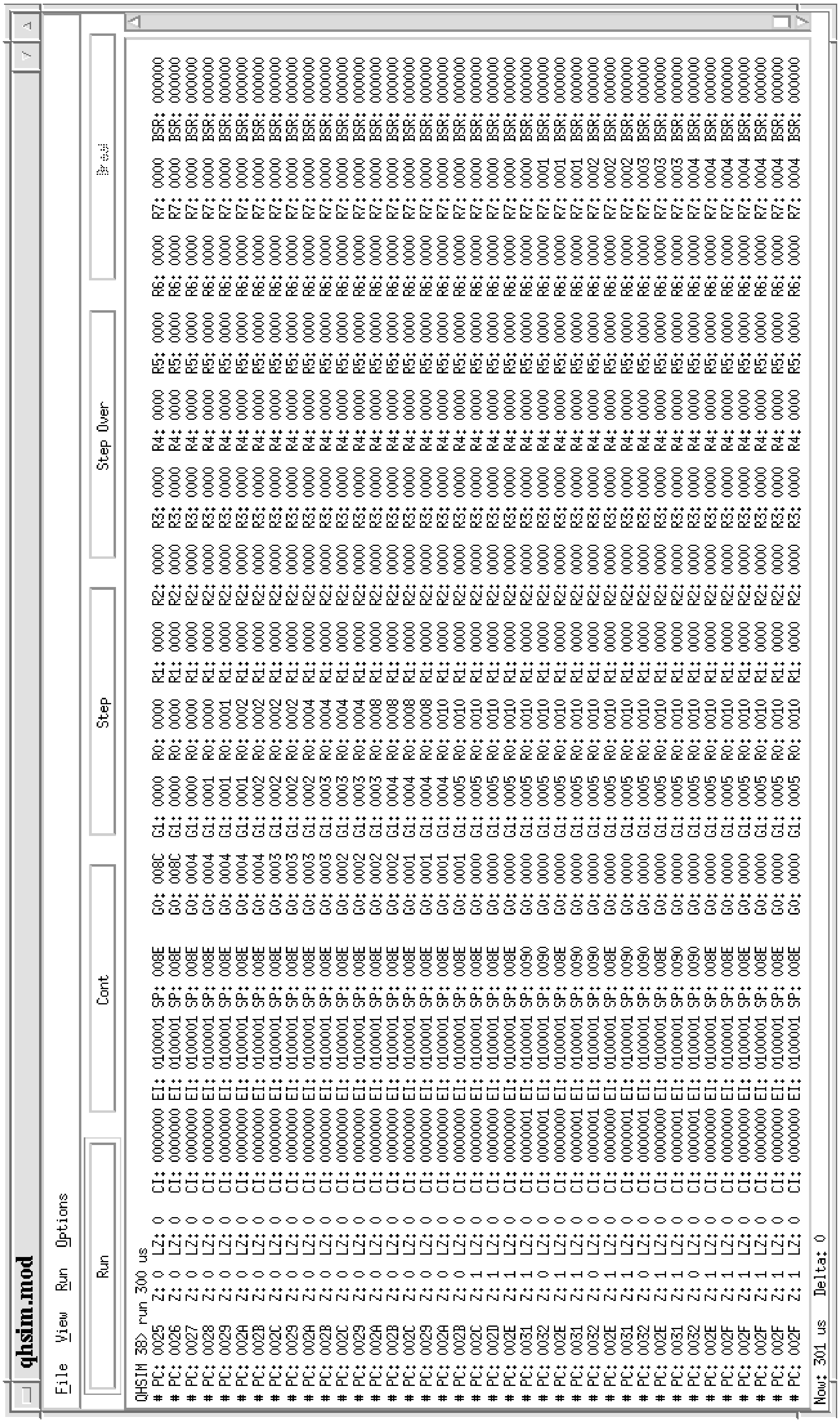

Figure D.3: *Sample trace output.*

E

Area Report from AutoLogic II

This appendix presents the area report for the hardware units in the BSP. First the entire BSP is presented, then all the units in the BSP is presented in turn.

Entire BSP

```
Global Cell Usage Statistics for / (netlist:work:bsp.bsp_v1)
Area Units: Optimization COST
================================================
 Cell                  Instance Cost/
 Name                     Count  Cell Subtotal
================================================
 GFL_LIB:FALSE                4
 GFL_LIB:TRUE                 1
 es2_ecpd07:LIBAN2           61  0.50    30.50
 es2_ecpd07:LIBAN3            6  0.65     3.90
 es2_ecpd07:LIBAN4            6  0.76     4.56
 es2_ecpd07:LIBAOI21        265  0.52   137.80
 es2_ecpd07:LIBAOI22        317  0.60   190.20
 es2_ecpd07:LIBAOI221        40  0.86    34.40
 es2_ecpd07:LIBAOI222         7  0.98     6.86
 es2_ecpd07:LIBBUF39         12  0.97    11.64
 es2_ecpd07:LIBDFF          317  1.75   554.75
 es2_ecpd07:LIBDFFR         334  2.09   698.06
 es2_ecpd07:LIBDFFS           8  1.90    15.20
 es2_ecpd07:LIBFADD1         15  1.93    28.95
 es2_ecpd07:LIBINV          386  0.27   104.22
 es2_ecpd07:LIBLAT           22  1.50    33.00
 es2_ecpd07:LIBMUX21        136  0.88   119.68
 es2_ecpd07:LIBMUX41        205  1.89   387.45
 es2_ecpd07:LIBNA2          448  0.38   170.24
 es2_ecpd07:LIBNA3           86  0.52    44.72
 es2_ecpd07:LIBNA4           59  0.60    35.40
 es2_ecpd07:LIBNO2          233  0.36    83.88
 es2_ecpd07:LIBNO3S          41  0.55    22.55
 es2_ecpd07:LIBNO4           30  0.60    18.00
 es2_ecpd07:LIBOAI21        447  0.52   232.44
 es2_ecpd07:LIBOAI22         58  0.62    35.96
 es2_ecpd07:LIBOAI221        22  0.83    18.26
 es2_ecpd07:LIBOAI222         1  0.94     0.94
 es2_ecpd07:LIBOR2           36  0.52    18.72
 es2_ecpd07:LIBOR3            7  0.65     4.55
 es2_ecpd07:LIBOR4            4  0.87     3.48
 es2_ecpd07:LIBSFF            9  2.46    22.14
 es2_ecpd07:LIBSFFR         148  2.52   372.96
 es2_ecpd07:LIBSFFS           2  2.72     5.44
 es2_ecpd07:LIBTRI3          48  1.14    54.72
 es2_ecpd07:LIBXNOR          26  0.83    21.58
 es2_ecpd07:LIBXOR           37  0.83    30.71
------------------------------------------------
              Total:     3884         3557.86
```

Serial Port

```
Global Cell Usage Statistics for /ser_port/
Area Units: Optimization COST
==============================================
 Cell                 Instance Cost/
 Name                    Count  Cell Subtotal
==============================================
 es2_ecpd07:LIBAN2          15  0.50     7.50
 es2_ecpd07:LIBAOI21        13  0.52     6.76
 es2_ecpd07:LIBAOI22        16  0.60     9.60
 es2_ecpd07:LIBAOI221        1  0.86     0.86
 es2_ecpd07:LIBDFF         301  1.75   526.75
 es2_ecpd07:LIBINV          25  0.27     6.75
 es2_ecpd07:LIBMUX21         3  0.88     2.64
 es2_ecpd07:LIBMUX41         5  1.89     9.45
 es2_ecpd07:LIBNA2          27  0.38    10.26
 es2_ecpd07:LIBNA3          21  0.52    10.92
 es2_ecpd07:LIBNA4           7  0.60     4.20
 es2_ecpd07:LIBNO2          23  0.36     8.28
 es2_ecpd07:LIBNO3S          5  0.55     2.75
 es2_ecpd07:LIBNO4           4  0.60     2.40
 es2_ecpd07:LIBOAI21        13  0.52     6.76
 es2_ecpd07:LIBOAI22         6  0.62     3.72
 es2_ecpd07:LIBOR2           1  0.52     0.52
 es2_ecpd07:LIBOR4           1  0.87     0.87
 es2_ecpd07:LIBSFFR          4  2.52    10.08
 es2_ecpd07:LIBXNOR          2  0.83     1.66
 es2_ecpd07:LIBXOR           7  0.83     5.81
----------------------------------------------
               Total:      500         638.54
```

Program Sequencer

```
Global Cell Usage Statistics for /sequencer/
Area Units: Optimization COST
==============================================
 Cell                 Instance Cost/
 Name                    Count  Cell Subtotal
==============================================
 es2_ecpd07:LIBAN2          16  0.50     8.00
 es2_ecpd07:LIBAN3           6  0.65     3.90
 es2_ecpd07:LIBAN4           4  0.76     3.04
 es2_ecpd07:LIBAOI21        37  0.52    19.24
 es2_ecpd07:LIBAOI22        25  0.60    15.00
 es2_ecpd07:LIBAOI221        6  0.86     5.16
 es2_ecpd07:LIBAOI222        1  0.98     0.98
 es2_ecpd07:LIBDFFR          9  2.09    18.81
 es2_ecpd07:LIBDFFS          1  1.90     1.90
 es2_ecpd07:LIBINV          99  0.27    26.73
 es2_ecpd07:LIBLAT          22  1.50    33.00
 es2_ecpd07:LIBMUX21         5  0.88     4.40
 es2_ecpd07:LIBNA2         110  0.38    41.80
 es2_ecpd07:LIBNA3          41  0.52    21.32
 es2_ecpd07:LIBNA4          26  0.60    15.60
 es2_ecpd07:LIBNO2         126  0.36    45.36
 es2_ecpd07:LIBNO3S         26  0.55    14.30
 es2_ecpd07:LIBNO4          16  0.60     9.60
 es2_ecpd07:LIBOAI21        68  0.52    35.36
 es2_ecpd07:LIBOAI22        21  0.62    13.02
 es2_ecpd07:LIBOAI221       19  0.83    15.77
 es2_ecpd07:LIBOAI222        1  0.94     0.94
 es2_ecpd07:LIBOR2          13  0.52     6.76
 es2_ecpd07:LIBOR3           5  0.65     3.25
 es2_ecpd07:LIBOR4           1  0.87     0.87
 es2_ecpd07:LIBSFFR          6  2.52    15.12
 es2_ecpd07:LIBSFFS          2  2.72     5.44
 es2_ecpd07:LIBXOR           3  0.83     2.49
----------------------------------------------
               Total:      715         387.16
```

Fifo Controller

```
Global Cell Usage Statistics for /fifo_ctrl/
Area Units: Optimization COST
==============================================
 Cell                 Instance Cost/
 Name                    Count  Cell Subtotal
==============================================
 es2_ecpd07:LIBAN2           1  0.50     0.50
 es2_ecpd07:LIBAN4           2  0.76     1.52
 es2_ecpd07:LIBAOI21         2  0.52     1.04
 es2_ecpd07:LIBAOI22         3  0.60     1.80
 es2_ecpd07:LIBAOI222        1  0.98     0.98
 es2_ecpd07:LIBDFF          13  1.75    22.75
 es2_ecpd07:LIBINV          14  0.27     3.78
 es2_ecpd07:LIBNA2          25  0.38     9.50
 es2_ecpd07:LIBNA3          11  0.52     5.72
 es2_ecpd07:LIBNA4           8  0.60     4.80
 es2_ecpd07:LIBNO2          26  0.36     9.36
 es2_ecpd07:LIBNO3S          5  0.55     2.75
 es2_ecpd07:LIBOAI21         8  0.52     4.16
 es2_ecpd07:LIBOAI22         5  0.62     3.10
 es2_ecpd07:LIBOAI221        2  0.83     1.66
 es2_ecpd07:LIBOR4           1  0.87     0.87
 es2_ecpd07:LIBXNOR          2  0.83     1.66
 es2_ecpd07:LIBXOR           3  0.83     2.49
----------------------------------------------
              Total:      132          78.44
```

Datapath

```
Global Cell Usage Statistics for /dp/
Area Units: Optimization COST
==============================================
 Cell                 Instance Cost/
 Name                    Count  Cell Subtotal
==============================================
 GFL_LIB:FALSE               1
 es2_ecpd07:LIBAN2          24  0.50    12.00
 es2_ecpd07:LIBAOI21        31  0.52    16.12
 es2_ecpd07:LIBAOI22       180  0.60   108.00
 es2_ecpd07:LIBAOI221       32  0.86    27.52
 es2_ecpd07:LIBBUF39        12  0.97    11.64
 es2_ecpd07:LIBDFFR        213  2.09   445.17
 es2_ecpd07:LIBDFFS          1  1.90     1.90
 es2_ecpd07:LIBFADD1        15  1.93    28.95
 es2_ecpd07:LIBINV          38  0.27    10.26
 es2_ecpd07:LIBMUX21       103  0.88    90.64
 es2_ecpd07:LIBMUX41       200  1.89   378.00
 es2_ecpd07:LIBNA2         264  0.38   100.32
 es2_ecpd07:LIBNA3           7  0.52     3.64
 es2_ecpd07:LIBNA4           6  0.60     3.60
 es2_ecpd07:LIBNO2          48  0.36    17.28
 es2_ecpd07:LIBNO3S          1  0.55     0.55
 es2_ecpd07:LIBNO4           1  0.60     0.60
 es2_ecpd07:LIBOAI21       265  0.52   137.80
 es2_ecpd07:LIBOAI22        16  0.62     9.92
 es2_ecpd07:LIBOR2           3  0.52     1.56
 es2_ecpd07:LIBOR4           1  0.87     0.87
 es2_ecpd07:LIBSFFR         26  2.52    65.52
 es2_ecpd07:LIBXOR          18  0.83    14.94
----------------------------------------------
              Total:     1506        1486.80
```

Bitstream Receiver

Global Cell Usage Statistics for /bs_rec/
Area Units: Optimization COST

Cell Name	Instance Count	Cost/ Cell	Subtotal
GFL_LIB:FALSE	1		
GFL_LIB:TRUE	1		
es2_ecpd07:LIBAN2	5	0.50	2.50
es2_ecpd07:LIBAOI21	182	0.52	94.64
es2_ecpd07:LIBAOI22	93	0.60	55.80
es2_ecpd07:LIBAOI221	1	0.86	0.86
es2_ecpd07:LIBAOI222	5	0.98	4.90
es2_ecpd07:LIBDFFR	112	2.09	234.08
es2_ecpd07:LIBDFFS	6	1.90	11.40
es2_ecpd07:LIBINV	204	0.27	55.08
es2_ecpd07:LIBMUX21	25	0.88	22.00
es2_ecpd07:LIBNA2	20	0.38	7.60
es2_ecpd07:LIBNA3	5	0.52	2.60
es2_ecpd07:LIBNA4	7	0.60	4.20
es2_ecpd07:LIBNO2	10	0.36	3.60
es2_ecpd07:LIBNO3S	3	0.55	1.65
es2_ecpd07:LIBNO4	9	0.60	5.40
es2_ecpd07:LIBOAI21	93	0.52	48.36
es2_ecpd07:LIBOAI22	10	0.62	6.20
es2_ecpd07:LIBOAI221	1	0.83	0.83
es2_ecpd07:LIBOR2	19	0.52	9.88
es2_ecpd07:LIBSFF	9	2.46	22.14
es2_ecpd07:LIBSFFR	112	2.52	282.24
es2_ecpd07:LIBXNOR	22	0.83	18.26
es2_ecpd07:LIBXOR	5	0.83	4.15
Total:	960		898.37

F

Assembler Syntax

This appendix describes the syntax of the assembler in Bachus Naur Form. This description serves as a basic for implementation of the assembler in Lex, Yacc and C. Further information conserning the implementation of the assembler can be found in the enclosed folder entitled “User’s Manual for the BSP Assembler”.

$::=$ Equals

{ } Repeat zero or more times

$< >$ syntactic construction

| or

‘ ’ token

$\mathcal{C}$ $\in$ {all single character}

$\mathcal{N}$ $\in \{0,1,2,3,4,5,6,7,8,9\}$

$\mathcal{H}$ $\in \{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f\}$

⟨ *code* ⟩	::=	{ ⟨ *statement* ⟩ } ‘`EOF`’
⟨ *statement* ⟩	::=	⟨ *indent* ⟩ ⟨ *command line* ⟩
	|	⟨ *beginning of line* ⟩ ⟨ *control line* ⟩
	|	⟨ *comment* ⟩
	|	⟨ *beginning of line* ⟩ ⟨ *include file* ⟩ { ⟨ *statement* ⟩ } ‘`EOF`’

```
⟨ binary instruction ⟩ ::= ⟨ register ⟩ '=' ⟨ register ⟩ ⟨ operator ⟩ ⟨ register ⟩
                         | ⟨ register ⟩ '=' ⟨ register ⟩ ⟨ operator ⟩ ⟨ value ⟩
                         | ⟨ register ⟩ '=' ⟨ register ⟩
                         | ⟨ register ⟩ '=' 'readfifo' '(' ⟨ value ⟩ ')'
                         | ⟨ register ⟩ '=' 'readfifo' '(' ⟨ register ⟩ ')'

⟨ unary instruction ⟩ ::= 'nop' | 'resetfifo'

⟨ constant ⟩           ::= '.word' ⟨ value ⟩

⟨ memory operation ⟩ ::= ⟨ register ⟩ '=' 'mem' '(' ⟨ register ⟩ '+' ⟨ register ⟩ ')'
                       | ⟨ g-register ⟩ '=' 'mem' '(' ⟨ register ⟩ '+' ⟨ value ⟩ ')'
                       | ⟨ register ⟩ '=' 'mem' '(' ⟨ register ⟩ ')'
                       | ⟨ g-register ⟩ '=' 'mem' '(' ⟨ label reference ⟩ ')'
                       | 'mem' '(' ⟨ register ⟩ '+' ⟨ register ⟩ ')' '=' ⟨ register ⟩
                       | 'mem' '(' ⟨ register ⟩ '+' ⟨ value ⟩ ')' '=' ⟨ g-register ⟩
                       | 'mem' '(' ⟨ label reference ⟩ ')' '=' ⟨ g-register ⟩

⟨ jump ⟩               ::= 'jump' ⟨ label reference ⟩
                       | 'jumpnz' ⟨ label reference ⟩
                       | 'jumplz' ⟨ label reference ⟩

⟨ masking ⟩            ::= 'mask' '(' ⟨ register ⟩ '++' ',' ⟨ buffer size ⟩ ')'

⟨ call ⟩               ::= 'call' ⟨ label reference ⟩
                       | 'lcall' ⟨label reference ⟩

⟨ fifo operation ⟩     ::= ⟨ register ⟩ '=' 'popfifo' '(' ⟨ register ⟩ ')'
                       | ⟨ register ⟩ '=' 'popfifo' '(' ⟨ value ⟩ ')'
                       | ⟨ register ⟩ '=' 'readfifo' '(' ⟨ register ⟩ ')'
                       | ⟨ register ⟩ '=' 'readfifo' '(' ⟨ value ⟩ ')'
                       | 'shiftfifo' '(' ⟨ register ⟩ ')'
                       | 'shiftfifo' '(' ⟨ value ⟩ ')'

⟨ return ⟩             ::= 'rtn' | 'rti'

⟨ huffman table operation ⟩ ::= 'jumpnode' '(' ⟨ value ⟩ '.' ⟨ label reference ⟩ ')'
                       | 'endnode' '(' ⟨ value ⟩ ',' ⟨ value ⟩ ',' ⟨ value ⟩ ')'

⟨ counter operation ⟩ ::= ⟨ register ⟩ '++'
                       | ⟨ register ⟩ '---'

⟨ label definition ⟩   ::= { C/{newline, space, tab} } ':'

⟨ label reference ⟩    ::= { C/{newline, space, tab} }

⟨ define ⟩             ::= '#define' { C/{newline, space, tab} } '=' ⟨ value ⟩
```

⟨ *command line* ⟩ ::= ⟨ *binary instruction* ⟩
| ⟨ *unary instruction* ⟩
| ⟨ *constant* ⟩
| ⟨ *memory operation* ⟩
| ⟨ *jump* ⟩
| ⟨ *masking* ⟩
| ⟨ *call* ⟩
| ⟨ *fifo operation* ⟩
| ⟨ *stack operation* ⟩
| ⟨ *counter operation* ⟩
| ⟨ *return* ⟩
| ⟨ *huffman table operation* ⟩

⟨ *control line* ⟩ ::= ⟨ *define* ⟩
| ⟨ *label definition* ⟩
| ⟨ *set address* ⟩
| ⟨ *huffman table base address* ⟩
| ⟨ *comment* ⟩

⟨ *define reference* ⟩ ::= { $\mathcal{C}$/{newline, space, tab} }

⟨ *set address* ⟩ ::= '`#set address`' '`=`' ⟨ *value* ⟩

⟨ *huffman table base address* ⟩ ::= '`#hufftable`'

⟨ *comment* ⟩ ::= '`;`' { $\mathcal{A}$/{ newline } '`newline`'

⟨ *buffer size* ⟩ ::= '`64`' | '`128`' | '`256`' | '`512`' | '`1024`' | '`2048`' | '`4096`' |
'`8192`'

⟨ *value* ⟩ ::= ⟨ *value* ⟩ '`+`' ⟨ *value* ⟩
| ⟨ *value* ⟩ '`-`' ⟨ *value* ⟩
| ⟨ *value* ⟩ '`^`' ⟨ *value* ⟩
| ⟨ *value* ⟩ '`*`' ⟨ *value* ⟩
| ⟨ *value* ⟩ '`/`' ⟨ *value* ⟩
| '(' ⟨ *value* ⟩ ')'
| ⟨ *decimal* ⟩
| ⟨ *hexadecimal* ⟩
| ⟨ *define reference* ⟩

⟨ *operator* ⟩ ::= '`and`' | '`or`' | 'xor' | '`+`' | '`-`' | '`<<`' | '`>>`'

⟨ *register* ⟩ ::= ⟨ *g-register* ⟩
| ⟨ *r-register* ⟩
| ⟨ *control registers* ⟩

⟨ *g-register* ⟩ ::= '`g0`' | '`g1`'

⟨ *r-register* ⟩ ::= '`r0`' | '`r1`' | '`r2`' | '`r3`' | '`r4`' | '`r5`' | '`r6`' | '`r7`'

⟨ *control registers* ⟩ ::= '`sp`' | '`pc`' | '`fifo`' | '`sr`'

⟨ *decimal* ⟩ ::= { $\mathcal{N}$ }

⟨ *hexadecimal* ⟩ ::= '`$`' { $\mathcal{H}$ }