FPGA DESIGN OF DAUBECHIES WAVELET LIFTING SCHEME
FOR AUDIO PROCESSING

Jay Maheshkumar Bhalodia
B.S, C.U.Shah College of Engineering and Technology, India, 2005

PROJECT

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND ELECTRONIC ENGINEERING

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

FALL
2008
FPGA DESIGN OF DAUBECHIES WAVELET LIFTING SCHEME
FOR AUDIO PROCESSING

A Project

by

Jay Maheshkumar Bhalodia

Approved by:

Dr. Jing Pang, Committee Chair

Dr. Preetham Kumar, Second Reader

12/03/2008

Date
Name of Student: Jay Maheshkumar Bhalodia

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

Dr Preetham Kumar, Graduate Coordinator

Date 12/04/18

Department of Electrical and Electronic Engineering
Abstract

of

FPGA DESIGN OF DAUBECHIES WAVELET LIFTING SCHEME
FOR AUDIO PROCESSING

by

Jay Maheshkumar Bhalodia

The goal of this project is to design and implement FPGA design of Daubechies lifting scheme for audio compression and reconstruction. The design approach is to use top down hierarchical design. The algorithm implemented performs compression and reconstruction for three levels. Daubechies scheme is recursive process in which output of previous level is input to another level and so forth. It was challenging to implement this project, as it needed lots of research effort for both software algorithm and hardware interface implementation. The completely audio compression system was implemented initially with Matlab and then with Verilog Hardware description language and emulated on Cyclone II FPGA of Altera DE2 kit. The algorithm generates satisfactory audio decompressed outputs.

Committee Chair

Dr. Jing Pang

12/03/2008

Date
ACKNOWLEDGMENTS

Special thanks to Dr Jing Pang for being my Project guide and guiding me when needed and always encouraging me to implement new things. Also special thanks to my colleague Shitalben Chauhan for helping me in resolving hardware issues when implementing the code. Special thanks to Dr. Preetham Kumar graduate coordinator of Electrical and Electronic Engineering Department at California State University Sacramento for being my second project guide and for providing me this wonderful opportunity of gaining the best knowledge and providing me this well equipped laboratory.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
</tbody>
</table>

Chapter

1. INTRODUCTION ............................................................... 1
   Purpose of the Study ....................................................... 1
   Design Work .............................................................. 1

2. DAUBECHIES WAVELETS .................................................... 3
   Daubechies Wavelet Scaling Function and Mother Wavelet ........... 3
   The Discrete Wavelet Transform and Multiresolution .............. 4
   Daubechies Coefficients .................................................. 5

3. BASIC LIFTING SCHEME ................................................... 10
   Lifting Scheme Introduction ............................................. 10
   Working of Lifting Scheme .............................................. 10

4. DAUBECHIES D4 TRANSFORM LIFTING SCHEME .................................. 15
   Working of Daubechies Lifting Scheme ................................ 15
   Inverse Transform .......................................................... 18
   FPGA Architecture Implementation of one dimension lifting Scheme ... 21

5. VERILOG IMPLIMENTATION .................................................. 23
   Audio Compression Using Daubechies Lifting Scheme ................ 23
   State Machines For Compression ....................................... 27
   Audio Reconstruction Using Daubechies Lifting Scheme ............ 32
   State Machine for Reconstruction ................................... 35

6. BLOCK DIAGRAM ............................................................ 39
   FPGA Block Diagram .................................................... 39
   Altera DE2 Kit ........................................................... 40
TABLE OF CONTENTS

7. RESULT ANALYSIS ........................................................................................................41
   Matlab Results ............................................................................................................41
   Hardware Results .......................................................................................................44

8. CONCLUSION .............................................................................................................45

Bibliography ..................................................................................................................46
Appendix ..........................................................................................................................48
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Daubechies Lifting Scheme Coefficients</td>
<td>7</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Three level wavelet decomposition tree</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Three level reconstruction tree</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Sample of Sound Wave</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Compressed Sound Sample</td>
<td>9</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Lifting Scheme Forward Wavelet Transform</td>
<td>11</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Two level lifting Scheme</td>
<td>12</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Multilevel implementation of basic lifting scheme</td>
<td>12</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Basic Lifting Scheme Inverse Transform</td>
<td>13</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Two Level basic lifting Scheme Inverse Transform</td>
<td>13</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Three Level basic lifting Scheme Inverse Transform</td>
<td>14</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Forward wavelet transform</td>
<td>15</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Two level implementation of Daubechies lifting scheme</td>
<td>16</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Three level implementation of Daubechies lifting scheme</td>
<td>16</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Daubechies inverse wavelet transform</td>
<td>18</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Two level implementation of Daubechies lifting scheme Reconstruction</td>
<td>19</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Three level implementation of Daubechies lifting scheme Reconstruction</td>
<td>19</td>
</tr>
<tr>
<td>Figure 17</td>
<td>FPGA Architecture of one Dimension lifting Scheme</td>
<td>21</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Basic components for compression algorithm</td>
<td>22</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 19 State Machine for Real time compression ........................................ 27
Figure 20 State Machine for Post Compression .............................................. 30
Figure 21 Basic Components of Wavelet Reconstruction algorithm ..................... 32
Figure 22 State machine for reconstruction ..................................................... 35
Figure 23 FPGA Internal diagram ................................................................. 39
Figure 24 Altera DE2 Kit ............................................................... 40
Figure 25 Matlab plot for level 1 compression .................................................. 41
Figure 26 Matlab plot for compression level 2 .................................................. 42
Figure 27 Matlab plot for compression level 3 .................................................. 42
Figure 28 Matlab plot for reconstruction level 3 ............................................... 43
Figure 29 Matlab plot for reconstruction level 2 ............................................... 43
Figure 30 Matlab plot for reconstruction level 1 ............................................... 44
1.1 Purpose of the Study

This project focused on the FPGA design of emerging wavelet data compression technique. In this project, the data is compressed in three different levels and then reconstructed. Level one compression is implemented in real time, which means data is input and compressed immediately before being stored for another level compression. This design used Daubechies lifting scheme to compress and decompress the data. The input data is multiplied by Daubechies coefficients to get compressed. Similarly, the original data can be retrieved by applying reconstructing Daubechies coefficients. This way of compression and decompression is fast and easily achievable in real time as implemented in level first compression. Thus after studying this project it is proved that very few components of original sound is required to reconstruct the original signal, because human ear can not distinguish very well between low frequency sounds [1].

1.2 Design Work

In this project, forward transformation of sound using Daubechies lifting scheme for three levels achieves 87.5% compression, meaning that 12.5% of original sound components are capable of regenerating original signal. This is highly efficient technique
for processing audio signal for applications such as telecommunication or memory storage.

The compression algorithm using Daubechies lifting scheme consists of four operational steps. The architectural detail of the hardware design is described in section 5.1. The four major steps of operations are: update 1, predict 1, update 2 and normalization. Normalized data is truncated and downscaled in order to co-op with reduced sized storage bits and up scaled coefficients.

The initial study of compression algorithm is performed using MATLAB. The verilog design implements the hardware architecture for wavelet compression and decompression. The verilog finite state machine (FSM) design was verified by running Modelsim simulation.
Chapter 2

DAUBECHIES WAVELETS

2.1 Daubechies Wavelet Scaling Function and Mother Wavelet

Nico Temme has derived equation to obtain wavelet scaling function by polynomial analysis. The equation to get Daubechies wavelet coefficients for db4 is shown in here.

\[( (1 + 1/z)/2 )^2 * (f(0) + (f(1)/z)) = 1/\sqrt{2} [h(0) + h(1)/z + h(2)/z^2 + h(3)/z^3]\]

Here, h(n) represents Daubechies coefficients. Accepting numerical analysis of Nico,

\[ f(0) + f(1) = 1 \text{ and } [f(0) - f(1)]^2 = 3. \]

Using these values in above equation gives scaling function value for db4 as shown below [14].

\[ h(0) = (1 + \sqrt{3}) / (4 * \sqrt{2}) \]
\[ h(1) = (3 + \sqrt{3}) / (4 * \sqrt{2}) \]
\[ h(2) = (3 - \sqrt{3}) / (4 * \sqrt{2}) \]
\[ h(3) = (1 - \sqrt{3}) / (4 * \sqrt{2}) \]

Wavelet scaling function and mother wavelet can be described using following equation [16].

Equation for scaling function:

\[ \phi(t) = \sum_{k=0}^{n} \sqrt{2} h_k \phi(2t-k) = \sum_{k=0}^{n} c_k \phi(2t-k) \]
Equation for mother wavelet:

\[ \psi(t) = \sqrt{2} \sum_k g_k \phi(2t - k). \]

Equation for scaling function performs summation of signal components on interval \([0, n]\). \(k\) represents component value and \(t\) represents time. With the known value of scaling function mother wavelet can be obtained using equation of \(\psi(t)\). [16]

2.2 The Discrete Wavelet Transform and Multiresolution

2.2.1 Wavelet Vs Fourier transforms

Fourier transform provides frequency domain representation of signal, while "wavelet transform provides time-frequency representation of signal" [13]. Fourier transform is good for analysis of stationary signal, wavelet works well for both stationary and non-stationary signals [13]. Fourier is a transform which provides all frequency components without giving time-domain information, while wavelet is a multiresolution analysis, which provides different time and frequency resolution to analyze different frequency components [13].
2.2.2 The Discrete Wavelet Transform

The advantage of using discrete wavelet transform over continuous wavelet transform is to enable fast computation by computer and hardware resources. In discrete wavelet transform, the signal components are processed with different resolution depending upon their frequencies. The signal is passed through multiple digital filters to separate different frequency components. Multilevel digital filter banks are used to perform the filtering [13].

2.2.3 Multiresolution Analysis

![Wavelet Decomposition Tree](image)

Fig. 1 Three level wavelet decomposition tree [13]

Fig. 1 represents three level decomposition tree, where each level is composed of low and high pass filter and down sampler. H0 and G0 represent high and low pass filters respectively. The output of first level low pass filter has signal component with half the sampling frequency. The consecutive higher-level building blocks will again decompose
signal components in two-frequency domain. The high frequency filter output is called
detailed components and the low frequency filter output is called approximation
coefficients. Each time output at low pass filter has signal components having half the
frequency spectrum of input signal at that level. Down sampling by half will discard half
the components and will keep every alternate components. The number of signal
components after any n-level decomposition will be (original number of signal
components) / \(2^n\).

According to Nyquist’s formula, sampling frequency should be twice the signal
frequency. If original signal has frequency \(f_s\) then sampling rate should be \(2 \times f_s\). Down
sampling discards half the components, so required sampling frequency also reduces by
half still all signal information is conserved in decimated coefficients. Discrete wavelet
transform provides good time resolution at higher frequencies and good frequency
resolution at lower frequencies is used [13].

Reconstruction of signal is exact reverse procedure as shown in Fig.2.

\[\text{Fig. 2 Three level reconstruction tree [13]}\]
In Fig.2, \( H_i \) and \( G_i \) represents high and low pass filters respectively. Both filters use same filter coefficients as used for decomposition. The up sampler prior to filter block pads zero between filter components that doubles signal length [13].

2.3 Daubechies Coefficients

Multiplying the signal by a coefficient is known as scaling. In this project, I am using the Daub4 coefficients with the values shown in Table 1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update 1</td>
<td>( \sqrt{3} )</td>
</tr>
<tr>
<td>Predict 1</td>
<td>( \sqrt{3}/4 )</td>
</tr>
<tr>
<td>Normalize</td>
<td>( (\sqrt{3} - 1)/\sqrt{2} )</td>
</tr>
</tbody>
</table>

Table 1 Daubechies Lifting Scheme Coefficients [5]

The values shown in Table 1 are used as multiplying coefficients that is explained in preceding chapters [5].
Wavelet transform is composed of two main steps:

1) Scaling a signal by some number which has typical characteristics which preserves the original harmonics of the signal that is multiplying a signal by the Daubechies coefficients.

2) Splitting the original signal and compressed output into odd and even components in order to preserve or utilize them for next level.

It works on the concept of conservation of energy. Daub4 wavelet transform conserves the energy of signals and redistributes its energy into a more compact form. This principle applies to all Daub4 wavelet transform.

For example, consider the signal in Fig.3.

![Fig.3 Sample of Sound Wave [1]](image)

Using FAWAV (FAWAVE is software for performing digital signal processing on digital audio and images.), it is possible to calculate the energy of signal. Thereafter we apply Daub4 wavelet transform on the signal and again measure the energy of the signal and it
remains the same. Similarly apply it after second and third level, it still remains the same.

The graph just shows the redistribution of energy for second level [1].

![Compressed Sound Sample](image)

**Fig. 4 Compressed Sound Sample [1]**

The energy is redistributed in second order harmonics from first order harmonics and so on. Therefore, even though we are not using the full original signal the energy of the signal remains constant. The conservation of energy phenomenon is shown here.

The Daubechies wavelet coefficients are chosen in such a way that the original signal harmonics are retained and the energy level of original signal is redistributed. Only special set of scaling coefficients, which have some mathematical values, can be used. It varies for different wavelets but all follow the same basic rule of conservation and redistribution of original signal energy [1].

One of the fundamental application of all wavelet transform is compression of signals. Daub4 wavelet transform is proven good for audio signal compression. In my project I used Daub4 lifting scheme.
Chapter 3

BASIC LIFTING SCHEME

3.1 Lifting Scheme Introduction

In mathematical analysis, wavelets were defined as translates and dilates of one mother wavelet function and were used to analyze and represent general function. Several techniques to construct wavelet bases exist and one of these is lifting scheme. Lifting scheme wavelet algorithms are recursive, which means output of first step becomes the input to next step. The initial data set consists of $2^n$ elements, then each successive step operates on $2^{n-i}$ elements. Where $i = 1, 2, \ldots, n-1$. For example if initial data set is of 128 length long than the wavelet transform of level 1 consists of 64, second level transform consists of 32, third level consists of 16 steps and so on. Basically the number of components for each level decreases by half. [2.]

3.2 Working of Lifting Scheme

The basic lifting scheme works as follow first it divides the data into two parts, the odd data and even data components. The lifting scheme is three step process in which first step is either split or merge followed by Update and thereafter predict in the end. After Predict step the data is normalized. Normalization is just scaling the data by some predetermined value. The study of each step in detail is shown below [2].
3.2.1 Predict

The predict step predicts the odd samples of original signal by linear interpolation using Daubechies coefficients. The predict step calculates the wavelet function in wavelet transform. The update step calculates the scaling function which results in smoother version of data. Predict step uses the even components and then generates the odd components so that it is only necessary to store the even components [2].

3.2.2 Update step.

Update step is low pass filter and it is opposite to predict step. Fig. 5 shows the basic concept of lifting scheme [2].

In forward transform, the input signal are split into two, the odd component and the even components. Thereafter it first does predict and then update. So first it calculates even components by subtracting odd components from even components and in update step it calculates the components by adding predict components with even components.
Fig. 6 shows the two level-lifting scheme for forward transform. In this two level of compression of data takes place.

Fig. 6 Two level lifting Scheme [2]

Fig. 7 shows the three level-lifting scheme for forward transform. Data is compressed for three level. Output of each level is feed into next level.

Fig. 7 Multilevel implementation of basic lifting scheme [2]

The inverse reconstruction of lifting scheme is the opposite which means we do update first and then do predict. In addition, addition is replaced by subtraction and subtraction is replaced by addition as shown in the Fig. 8.
For level 2 and level 3, the operation is same to that in level 1[2].

Figure 9 and figure 10 below shows level-2 and 3-level inverse lifting reconstruction schemes. Data from each level is merged and which becomes input for next level.
Figure 10 shows three level of basic lifting scheme for inverse transform. The data of level 1 is merged and feed back into level 2 and from level 2 to level 3 for reconstruction.

The advantage of lifting wavelet transform is that it is fast compared to other wavelet transform. As a result, it finds its application in real time signal processing. The architecture of lifting scheme is reusable which means the same block can be used a number of times. As a result, from architectural point of view the lifting scheme utilizes very less number of gates and simplified design at gate level. In addition, it is less complicated design compared to other wavelet transforms [17].
Chapter 4

DAUBECHIES D4 TRANSFORM LIFTING SCHEME

4.1 Working of Daubechies Lifting Scheme

Wim Sweldens and his co-workers developed this wavelet-lifting scheme. The major advantage of this wavelet lifting scheme algorithm is that they are memory efficient and do not require a temporary storage as the version of Daubechies D4 transform does [5].

Fig. 11 shows the forward transform of Daubechies lifting scheme

![Diagram of Daubechies Forward Wavelet Transform](image)

Daubechies Forward Wavelet Transform

As shown in Fig 11 the first step is split. The original signal is divided into two, odd components and even components. Thereafter the next step is update 1 step. In this all the even components are scaled. The next step is predict in which all the odd components are scaled by Daubechies coefficients. Thereafter the third step is again
update 2 which is kind of similar to update 1 step but slightly different scaling. The last step is normalization in which both final odd and final even components are again scaled by Daubechies coefficients. The whole summary can be summarized in the following steps. [5].

Figure 12 shows two level implementation of Daubechies lifting scheme which has update 1 step followed by predict 1 step and in the end followed by update 2 step.

Fig. 12 Two level implementation of Daubechies lifting scheme

Figure 13 shows three level implementation of Daubechies lifting scheme in which output of level 1 is feed back to level 2 and from level 2 to level 3.

Fig. 13 Three level implementation of Daubechies lifting scheme [5]
For simplicity and ease of understanding, I will follow the same naming convention used so it is easier to understand. The split step divides the input signal into even elements which are stored in the first half of an N element array section (So to S(half-1)) and the odd elements which are stored in the second half of an N element array section (S half to S(n-1)). The expression uses S(Half +n) references as odd elements and S(n) as even element.

Step 1: Split digital input samples into odd and even components

Step 2: Update 1 [5]

Update 1:
For n = 0 to half - 1
S[n] = S[n] + \sqrt{3} S[\text{half} + n]

Step 3: Predict [5]

Predict:
S[\text{half}] = s[\text{half}] - \frac{\sqrt{3}}{4} S[0] - \frac{\sqrt{3} - 2}{4} S[\text{half} - 1]
for n = 1 to half - 1
S[\text{half} + n] = s[\text{half} + n] - \frac{\sqrt{3}}{4} S[n] - \frac{\sqrt{3} - 2}{4} S[n - 1]

Step 4: Update 2 [5]
Update 2:

for \( n = 0 \) to \( \text{half} - 2 \)

\[
S[n] = S[n] - S[\text{half} + n + 1]
\]

\[
S[\text{half} - 1] = S[\text{half} - 1] - S[\text{half}]
\]

Step 5: Normalization [5]

Normalize:

For \( n = 0 \) to \( \text{half} - 1 \)

\[
S[n] = \frac{\sqrt{3} - 1}{\sqrt{2}}
\]

\[
S[n + \text{half}] = \frac{\sqrt{3} + 1}{\sqrt{2}} S[n + \text{half}]
\]

For level 2 the same steps are repeated. The input for level 2 will be the output of normalized level 1 even components. Similarly for level 3 the inputs will be normalized level 2 even components [5].

4.2 Inverse Transform

In Lifting scheme, the inverse transform is the opposite to the forward transform as shown in the inverse transform Fig. 14[5].
Fig. 14: Daubechies inverse wavelet transform [5]

The only difference between Fig. 11 and Fig. 14 is that the + sign in Fig. 11 is replaced by – sign and the – sign in Fig. 11 is replaced by + signs. Everything else remains the same. All the naming conventions such as $S(n)$ for even components and $S(n+1)$ for odd component applies here. Below are the formulas used for signal reconstruction.

Figure 15 shows two level implementation of Daubechies inverse wavelet transform in which output of level 1 is feed back to level 2 input.

Fig. 15 Two level implementation of Daubechies lifting scheme Reconstruction [5]
Figure 16 below shows three level implementation of Daubechies inverse lifting scheme used to reconstruct signal after three level of compression.

Step 1: De-normalization: The formulae used for de-normalizing coefficient are given below. [5]

\[
\text{Denormalize:} \\
\text{For } n = 0 \text{ to } \frac{\text{half}}{2} - 1 \\
S[n] = S[n] \frac{\sqrt{3} + 1}{\sqrt{2}} \\
S[n + \text{half}] = \frac{\sqrt{3} - 1}{\sqrt{2}} S[n + \text{half}] 
\]

Step 2: Inverse Update 2 [5]

\[
\text{Inverse update 2:} \\
\text{for } n = 0 \text{ to } \frac{\text{half}}{2} - 2 \\
S[n] = S[n] + S[\text{half} + n + 1] \\
S[\text{half} - 1] = S[\text{half} - 1] + S[\text{half}] 
\]

Step 3: Inverse Predict [5]
**Inverse Predict:**

\[
S[\text{half}] = s[\text{half}] + \frac{\sqrt{3}}{4} S[0] + \frac{\sqrt{3} - 2}{4} S[\text{half} - 1]
\]

for \( n = 1 \) to \( \text{half} - 1 \)

\[
S[\text{half} + n] = s[\text{half} + n] + \frac{\sqrt{3}}{4} S[n] + \frac{\sqrt{3} - 2}{4} S[n - 1]
\]

**Step 4: Inverse Update 1 [5]**

**Inverse Update 1:**

For \( n = 0 \) to \( \text{half} - 1 \)

\[
S[n] = S[n] - \sqrt{3} S[\text{half} + n]
\]

4.3 FPGA architecture implementation of one dimension lifting scheme

---

Fig. 17 FPGA Architecture of one Dimension lifting Scheme

Fig.17 above shows the one dimension architecture of lifting scheme. Output of each step is latched in flip flop (FF) and thus can be used in later step. This Fig.17 shows the output
values of predict step are used to calculate the result for the update step. Then the update step output values are used to facilitate calculating the predict values [17].
5.1 Audio Compression using Daubechies lifting scheme:

Fig. 18 indicates basic functional components for compression algorithm.

![Fig. 18 Basic components for compression algorithm](image)

Description:

[a] Input Sample:

Scenario I: Real time implementation

The analog audio input coming from microphone is converted to digital sample and directly sent to compression block as an input. This implementation helps avoid
large memory storage to keep original audio vector coming from audio codec. Instead one can directly store first level compressed data.

Scenario II: Post implementation

This implementation is done particularly for enhancing higher level of compression. The higher level compression algorithm uses repetition of steps followed in first level of compression, except that input data comes from synchronous dynamic random access memory.

[b] Split:

Daubechies lifting scheme operates odd and even components of incoming sound vector differently. Therefore, split step is required as initial operational step where two sixteen bit registers are dedicated to hold odd components and two for even components. Each incoming sample is extracted in a way that odd component remains in odd register and even component remains in even register. One pair of even and odd components are processed at a time, and additional pair is provided for predict step which needs next odd component in its calculation. Similarly previous even component is also needed which is stored in temporary register and kept for one iteration of compression calculation.

[c] Adder:

After split operation, even components are sent directly to one input of adder and odd components are sent to multiplier. The first operation of Daubechies lifting scheme is update_1. It requires following operation to be performed on even and odd components and the generated results will be used as updated even component.
updated_even = even + (up_1 * odd)

Here, up_1 is an up scaled coefficient. Up scaling of coefficients is necessary to convert floating point number into integer format.

For update_1 operation the multiplication up_1*odd is done first. Output of multiplier goes to second input port of adder. Finally, addition of both inputs will give output of first update.

[d] Multiplier1, Multiplier2 and Subtractor:

The three blocks including two multipliers and subtraction unit cumulatively implement predict_1 arithmetic. The equation of operation for predict_1 step is given bellow.

updated_odd = odd - (pr_coef1*updated_even) - (pr_coef2*updated_even_previous)

In addition to current even-odd pair, predict step calculation requires previous updated even component. For the fist sound vector pair, previous even component is assumed to be zero and for all other vector pairs, the previous operated component is stored in temporary register. Again pr_coef1 and pr_coef2 are up scaled coefficients. Multiplier1 performs (pr_coef1* updated_even) and its result goes to input port of Subtractor. Similarly, Multiplier2 performs (pr_coef2*updated_even_previous) and goes to another input port of Subtractor. Subtractor block is composed of two subtraction units, where one unit subtracts Multiplier1 output from odd component and another unit subtracts Multiplier2 output from result of first subtraction unit. Output of second subtraction is the final output of predict_1 step, which is treated as updated odd component for further processing.
[e] Subtractor:

Second update step needs to perform subtraction of advanced [next] updated odd component from current even component coming out from predict block. Each prior step operates on two pair of even and odd components in order to make next updated odd component available for update_2 operation. Equation for update_2 is as below.

\[
\text{updated}_{\text{even}} = \text{updated}_{\text{even}} - \text{updated}_{\text{next odd}}
\]

update_2 is a simple subtraction operation that doesn’t require any compression coefficient.

[f] Multiplier:

The last step for Daubechies lifting scheme is normalization, which requires both even and odd elements to be multiplied with normalization coefficients. Normalization coefficients are also scaled up to obtain integer value. Normalization step can be described using following equations.

\[
\text{Normalized}_{\text{even}} = \text{updated}_{\text{even}} \times \text{norm}_{\text{coef}}_1
\]

\[
\text{Normalized}_{\text{odd}} = \text{updated}_{\text{odd}} \times \text{norm}_{\text{coef}}_2
\]

Although Normalization is the last operation for Daubechies lifting scheme, one more step of truncation is needed to reduce size of normalized components to sixteen bit.

[g] Shift Right register:

The Shift Right register serves two purposes – down scaling and truncation of result of normalized block. Down scaling is required to revert change in results occurred
due to up scaled coefficients and the compressed results will be written to SDRAM (Synchronous Dynamic Random Access Memory).
5.2 State Machines for Compression

Figure 19 shows the state machine for real time compression with each step as designed in verilog code. States are named based on the type of operation it does.

Scenario 1: Real time compression
State Description:

Idle state:

Reset state. All registers are initialized. Reset is active low and when it goes high state transition will happen on next rising edge of algorithm clock.

Shift_reg state:

It contains set of four sixteen bit registers. For real time application input data is sampled at thirty second rising edge of bit clock once rising edge of sampling clock is detected. This enables real time operation. It makes transition to Shift_ctrl & wait state after each shift is performed.

Shift_ctrl & wait_state states:

This state controls shifting operation of four shift registers. It also contains 32 bit counter to enable sampling of input digitized audio at appropriate instant to support real time operation.

update_1, predict_1, update_2, normalize, down_scale states:

These are the lifting operation states where each is responsible to perform corresponding operation as described in Section X.1.1.

mem_wr & wait_state states:

This state generates write command to SDRAM and provides write address and compressed data. It also contains shift control counter so state transition to next shif_reg
state is only done at the instant when new sampled data is available from ADC data collection unit.

Shift_reg and Shift_ctrl & wait_state states:

The function of these two states are similar to first two shifting states. In addition, they also compared current address value with predefined maximum address value to define memory region where compressed data is stored. When the upper limit of the address is reached, the state transits to the halt state.

Halt state:

It generates an output signal to indicate completion of real time compression. The state machine remains in this state until reset goes low.
Scenario 2: Post compression [Compression level >1]

Figure 20 shows state machine of post compression as designed in verilog code. Each state name describes the type of operation it does.
State Description:

The post compression is performed after real time compression is completed to achieve higher level of compression. The post compression state machine reads lower level compressed data from memory and goes through same operational states in level 1 compression and write back the compressed elements again in memory in different region.

Additional states:

Memory read state: Idle is a reset state. In post compression algorithm, when reset is high, state transition goes to Memory read state to read lower level compressed data. Another memory read operation state is introduced after memory write state to read next compressed data form memory to continue higher level compression.

Removed operation states:

There is no wait state with shift control state as data is coming from SDRAM in parallel format.
5.3 Audio Reconstruction using Daubechies lifting scheme:

Figure 21 shows basic components of wavelet reconstruction algorithm

Description:

[a] Divider:

This block performs denormalization of compressed data. It is an inverse operation of normalization. In normalization elements are multiplied with coefficient, in denormalization incoming compressed sound components are divided with the same coefficients.

Equations for denormalization operation are:
denorm\_even = even / norm\_1

denorm\_odd = odd / norm\_2

[b] Adder:

Adder performs inverse update\_2 operation. As update\_2 performs subtraction of next odd components from present even components, in reconstruction denormalized odd\_next and even components are added.

Equation for invert update 2 operation is:

\[ \text{inv\_update\_even} = \text{denorm\_even} + \text{denorm\_odd}\_next \]

[c] Multiplier1, Multiplier2 and adder:

These three components perform invert predict 1 operation. The operation of Multiplier1 and Multiplier2 are same as in compression. In this block, the subtraction operation is replaced by addition to obtain inversion.

The equation for invert predict operation is:

\[ \text{inv\_predict\_odd} = \text{denorm\_odd} + \text{inv\_update\_even}\_present + \text{inv\_update\_even}\_previous \]

Here, the adder block is composed of two adders, one of them adds odd denormalized component and inverted even component, while the other adds this result to inverted previous even component.

[d] Multiplier and Subtractor:

These blocks implement invert update 1 logic. The inputs to the block are inverted odd components from inv\_predict\_1 block and inverted even component from inv\_update\_2 block. Odd components are multiplied with coefficient up\_1 and then subtracted from even component.
Inverse update 1 operation can be described by following equation.

\[
\text{inv\_update\_1\_even} = \text{inv\_update\_2\_even} - \text{up\_l} * \text{inv\_predict\_1\_odd}
\]

[e] Right Shifter:

As at each state coefficients of compression are up scaled the final result is down scale in order to compensate for up scaling and also truncated to fit the result in sixteen bit size. The output of this block is reconstructed component that goes to defined region of SDRAM.
5.4 State machine for reconstruction

Figure 22 shows state machine for reconstruction. Each state name is describes the operation it does.
State description:

Idle state:

Idle is a reset state, where all registers are initialized. Reset is active low signal. When it goes high, transition to memory read state occurs on next positive edge of clock.

Memory read: The compressed audio components are stored in predefined memory regions of SDRAM and later they are used as inputs to the reconstruction block to acquire original data. In Memory read state read signal for SDRAM is generated. The data on addressed location becomes available on data bus and is sampled in shift register.

Shift_reg and counter states:

The function of these two state is to collect two pair of odd and even components to be processed in first iteration of reconstruction cycle. The working of both states is same as described in section X.1.2.1.

denormalized, inv_update_2, inv_predict_1, inv_update_1 states:

All of these states contain mathematical equations to perform corresponding lifting scheme reconstruction step as described in section X.2.1.

Downscale states:

Daubechies compression coefficients used at each states are up scaled values. In downscale state right shifting is done to compensate for up scaling of coefficients. The down scaled components are stored in sixteen bit register and send to memory write state.

Memory write state:

The reconstructed components are written back to SDRAM. This state generates write signal to SDRAM memory and reconstructed component goes to defined memory
The reconstructed components are written back to SDRAM. This state generates write signal to SDRAM memory and reconstructed component goes to defined memory region. The state is transited to Memory read state to obtain next components to be reconstructed.

Memory read, Shift_reg, counter states:

These three states are repetition of the same states described initial except that they deal with one pair of odd-even compressed components. When read address reached the higher limit of compressed data region, state transition is done to halt state.

Halt state:

State machine remains in halt state until Reset again goes low to restart reconstruction algorithm.
6.1 FPGA BLOCK DIAGRAM

Fig. 23 above shows block diagram of FPGA. Audio Data Collection Logic was responsible for converting audio data into digital format. It takes in signals bit clock, sampling clock and audio input signal. The output is digital signal which is feed into real time level 1 compression block. This block compresses the audio signal and passes it to SDRAM controller which is responsible for writing the data in SDRAM. After level 1 compression SDRAM controller fetches the level 1 compressed data and through FPGA it does level 2 and level 3 compression and stores the level 3 compressed data back into SDRAM. After three level 3 compression FPGA calls back the level 3 compressed data for reconstruction of original signal and reconstructs the full original signal which is
SDRAM. After three level 3 compression FPGA calls back the level 3 compressed data for reconstruction of original signal and reconstructs the full original signal which is stored back into SDRAM through SDRAM controller. Finally this reconstructed data is feed into digital to analog Convertor which performs the task of converting parallel data to serial and also playing the audio data.

6.2 Altera DE2 Kit

Fig. 24 Altera DE2 Kit

Fig.24 was the Altera DE2 Kit used in the project. It has FPGA named as Altera Cyclone II, SDRAM of 4 MB size located to left of FPGA. On bottom left there sixteen on/off switches and on bottom right there are four push to on switches. Also it has led on top of each switches and eight seven segment display to display output. It also has LCD and programmable interface through USB blast. On top there are three pins of which first one is microphone input, second is line input and the third one is output. On top left there is power on switch and run/program switch to program the FPGA.
Chapter 7

RESULT ANALYSIS

7.1 Matlab Results

Three levels of compression and reconstruction were done in Matlab. The compressed low frequency results were played in audio successfully. Thereafter three level of reconstruction were implemented and the reconstructed audio data were played successfully. The following figures show the simulation results. Figure 25 shows the output of level 1 compression.

![Figure 25 Matlab plot for level 1 compression](image)

Figure 25 Matlab plot for level 1 compression

Figure 26 shows Matlab simulation output for level 2 compression of signal. The signal is played with half the sampling frequency.
Figure 27 shows the Matlab simulation output for level 3 compression of the signal. The signal is played with three times slower the original signal.

Figure 28 shows Matlab simulation result for reconstruction of level 3. The reconstructed signal is played at frequency three times lower than original signal.
Figure 28 Matlab plot for reconstruction level 3
Figure 29 shows the reconstructed signal for level 2. The output signal has all odd and even components and its played at half the frequency of original signal.

Figure 29 Matlab plot for reconstruction level 2
Figure 30 shows the original signal after reconstruction. It is exactly similar to the original signal. It has all odd and even components.
7.2 Hardware Results

The verilog hardware design was implemented on FPGA kit. The audio sound was recorded through a microphone. After three levels of lifting scheme wavelet compression and reconstruction, the reconstructed audio data were played successfully.
Chapter 8

CONCLUSION

This project shows that Daubechies lifting scheme is really an efficient scheme to compress audio sound and using this technique we can compress audio sound to 87.5% of its original value. Also this technique can be used in real time compression as a result the original memory required to store the data will be much less, Daubechies coefficients redistributes the energy of the original signal in lower harmonics and thus the overall energy of the signal remains constant. Moreover the reconstruction has not much noise and we were able to recover the original signal from the compressed signal with sound as good as original sound.
BIBLIOGRAPHY


http://documents.wolfram.com/applications/wavelet/FundamentalsofWavelets/1.4.3.html


APPENDIX

SOURCE CODE

Main Module

module sdrm_ad(reset, clk_50Mhz, iAUD_BCK, key0, iAUD_DATA, iAUD_LRCK, AUD_XCK, I2C_SCLK, I2C_SDAT, dac_out, rdwr, zs_addr,
               zs_ba, zs_cas_n, zs_cke, zs_cs_n, zs_dq, zs_dqm, zs_ras_n,
               zs_we_n, sdrm_clock, sdrm_reset, led1, counter1, level1);
input iAUD_BCK, key0, iAUD_DATA, iAUD_LRCK, reset, clk_50Mhz, sdrm_reset;
output AUD_XCK, I2C_SCLK, zs_cke, zs_cs_n, zs_ras_n, zs_we_n, sdrm_clock, led1;
inout [1:0] rdwr;
inout I2C_SDAT, zs_cas_n;
output dac_out;
output [11:0] zs_addr;
output [1:0] zs_ba;
inout [15:0] zs_dq;
output [1:0] zs_dqm;
output [3:0] counter1;
wire out_12, bclk, tclk;
wire AUD_DACDAT;
wire [4:0] cnt_1;
wire [21:0] az_addr;
wire [15:0] adc_in;
wire [21:0] wr_addr;
wire [21:0] rd_addr;
wire [15:0] iSDRAM_DATA;
wire [1:0] az_be_n;
wire az_cs, az_rd_n, az_wr_n;
wire [15:0] az_data;
wire [15:0] in_1;
wire read, write;
wire [21:0] address;
wire [15:0] out_1;
output [1:0] level1;
wire reset_out;
reg [9:0] cnt;
reg clk_500khz;
wire trigger;
assign in_1 = zs_dq;
assign az_cs = 1'b0;

reg clk_1hz;
reg [25:0]count_1hz;
wire [1:0]level1,level2,level3;
wire [15:0]out_11,out_111;
wire readl,write1,read11,write11;
wire [21:0]address1,address11;
wire [3:0]counter1_1,counter1_2,cownter1_3;

always@(posedge clk_50Mhz or negedge reset)
begin
  if(!reset)
    begin
      count_1hz <= 26'd0;
      clk_1hz <= 1'b0;
    end
  else if(count_1hz < 26'd 24999999)
    begin
      count_1hz <= count_1hz + 1'b1;
      clk_1hz <= 1'b0;
    end
  else if(count_1hz >= 26'd 24999999 && count_1hz < 26'd 49999999)
    begin
      count_1hz <= count_1hz + 1'b1;
      clk_1hz <= 1'b1;
    end
  else
    begin
      count_1hz <= 26'd0;
      clk_1hz <= 1'b0;
    end
end

assign iSDRAM_DATA = (rdwr == 2'b11)?zs_dq: adc_in;
assign az_rd_n = (rdwr == 2'b00) ? read11 : (rdwr == 2'b11) ? 0 : (rdwr == 2'b01) ? read : read1;
assign az_wr_n = (rdwr == 2'b00) ? write11 : (rdwr == 2'b11) ? 1 : (rdwr == 2'b01) ? write : write1;
assign az_addr = (rdwr == 2'b00) ? address11 : (rdwr == 2'b11) ? rd_addr : (rdwr == 2'b01) ? address : address1;
assign az_data = (rdwr == 2'b00) ? out_111 : (rdwr == 2'b01) ? out_1 : (rdwr == 2'b10)? out_1:16'd0;

//assign counter1 = (rdwr == 2'b01)? counter1_1:(rdwr == 2'b10)? counter1_2:(rdwr == 2'b00)? counter1_3:4'd0;

//assign levell = (trigger == 1'b0)? levell1: levell2;
assign levell = levell3;
assign counter1 = counter1_3;

wire [3:0]counter1_3;

top_code a81(iAUD_BCK, reset_out,adc_in, r0(clk_500khz,reset,in_1,
out_1,read,write,address,led1,counter1_1,levell1,trigger);
recons_1 out_111,read11,write11,address11,,counter1_3,levell3);
top_code_2_3 clk_500khz, trigger,in_1,
out_11,read1,write1,address1,,counter1_2,levell2);
top_adc s0(wr_addr, adc_in,iAUD_BCK,iAUD_DATA,iAUD_LRCK,reset,reset_out);
top_dac s1(rd_addr,ISDRAM_DATA,iAUD_BCK,dac_out,iAUD_LRCK,reset);

DE2_i2sound a2 //INPUTS:

clk_50Mhz,key0,iAUD_DATA,iAUD_BCK,iAUD_LRCK,iAUD_LRCK,
//OUTPUTS:
AUD_DACDAT,AUD_XCK,I2C_SCLK,
//INOUT
I2C_SDAT,out_12,bclk,lrclk);

dsram_0 a4( // inputs:
az_addr,
az_be_n,
az_cs,
az_data,
az_rd_n,
az_wr_n,
clk_50Mhz,
sdram_reset,

// outputs:

"","
zs_addr,
zs_ba,
zs_cas_n,
zs_cke,
zs_cs_n,
always@(posedge clk_50Mhz or negedge reset)
begin
  if(!reset)
  begin
    cnt <= WM'd0;
    clk_500khz <= 1'b0;
  end
  else if(cnt < 10'd50)
  begin
    cnt <= cnt + 1'b1;
    clk_500khz <= 1'b0;
  end
  else if(cnt>= 1O'd50 && cnt<10'd100)
  begin
    cnt <= cnt + 1'b1;
    clk_500khz <= 1'b1;
  end
  else
  begin
    cnt <= WM'd0;
    clk_500khz <= 1'b0;
  end
end
endmodule

ADC CODE

module top_adc(wr_addr, data, iAUD_BCK, iAUD_DATA, iAUD_LRCK, reset, reset_out);
input iAUD_BCK, iAUD_DATA, iAUD_LRCK, reset;
output reg[21:0]wr_addr;
output reg [15:0]data;
reg [4:0]cnt;
reg [31:0]temp;
reg [1:0] cs;
reg [4:0] index;
output reg reset_out;
parameter s0 = 2'd0,
                     s1 = 2'd1,
                     s2 = 2'd2,
                     s3 = 2'd3;

always@(posedge iAUD_BCK or negedge reset)
begin
  if(!reset)
  begin
    wr_addr <= 22'd0;
    data <= 16'd0;
    cnt <= 5'd0;
    cs <= s0;
    index <= 5'd31;
    reset_out <= 1'b0;
  end
  else
  begin
    case(cs)
      s0: begin
        if(iAUD_LRCK == 1'b1)
          cs <= s1;
        else
          cs <= s0;
      end
      s1: begin
        if(iAUD_LRCK == 1'b0)
          cs <= s2;
        else
          cs <= s1;
      end
      s2: begin
        if(cnt <= 5'd16)
          begin
            cnt <= cnt + 1;
            temp[index] <= iAUD_DATA;
            index <= index - 1'b1;
            cs <= s2;
          end
        else if(cnt <= 5'd31)
  end
begin
  cnt <= cnt + 1;
  reset_out <= 1'b1;
  index <= index - 1'b1;
  data <= temp[31:15];
  cs <= s2;
  if(cnt == 5'd17)
    begin
      wr_addr <= wr_addr + 1;
    end
  end
endcase
end
endmodule

COMPRESSION CODE LEVEL 1 REAL TIME

module top_code (clk, reset,in_1, out_1,read,write,address,led1,counter1,levell,trigger);
  input clk,reset;
  input signed [15:0] in_1;
  output reg signed [15:0] out_1;
  output reg read,write,led1;
  output reg [21:0] address;
  reg signed [15:0] o1,o2,e1,e2;
  reg signed [22:0] temp_e1,temp_e2,e0;
  reg signed [28:0] temp_o1,temp_o2, temp_c1;
  reg signed [34:0] norm_e1, norm_o1;
  reg signed [25:0] shift_e1,shift_o1;
  reg signed [15:0] final_e1,final_o1;
  reg [2:0]count1;
  reg [1:0] count2;
  reg [3:0]cs;
  reg [21:0]w_address, r_address;
  output reg [3:0]counter1;
  output reg [1:0]levell;
  reg [4:0]counter_in;
  output reg trigger;

  parameter s0 = 4'd0,
              s1 = 4'd1,
s2 = 4'd2,
s3 = 4'd3,
s4 = 4'd4,
s5 = 4'd5,
s6 = 4'd6,
s7 = 4'd7,
s8 = 4'd8,
s9 = 4'd9,
s10 = 4'd10,
s11 = 4'd11,
s12 = 4'd12,
s13 = 4'd13,
s14 = 4'd14;

//assign out_1 = shift_e1;  // this is the output of 1st level- To be written in to SDRAM
//assign out_1 = final_e1;

always@ (posedge clk or negedge reset)
begin
    if (!reset)
        cs <= s0;
    else
        begin
            case(cs)
                s0: begin  // Ideal state all reset
                    o2 <= 0;
                    e2 <= 0;
                    o1 <= 0;
                    e1 <= 0;
                    e0 <= 23'd0;
                    count1 <= 3'd4;
                    count2 <= 2'd2;
                    address <= 22'd0;
                    //w_address <= 22'h20_0001;
                    w_address <= 22'h00_0001;  // write address start at 00_0001h.
                    r_address <= 22'd0;        // read address - ignored
                    read <= 1'b1;
                    write <= 1'b1;
                    led1 <= 1'b1;
                    counter1 <= 4'd0;
                    levell <= 2'd0;
                    // state transition
                    cs <= s1;
                    count_in <= 5'd0;
            endcase
            end
end
trigger <= 1'b0;

s11: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    counter1 <= 4'd11;
    cs <= s1;
end

s1: begin
    // loading the registers and shifting 4 times
    o2 <= in_1; // 3rd sound component
    e2 <= o2; // 2nd sound component
    o1 <= e2; // 1st sound component
    e1 <= o1; // 0th sound component
    // e0 <= 0;
    count1 <= count1 - 1'b1;
    r_address <= r_address + 1'b1;
    counter1 <= 4'd11;
    // state transition
    cs <= s9;
end

s9: begin
    // shifting control state
    counter1 <= 4'd9;
    if(count1 == 4'd0)
        cs <= s2;
    else
        begin
            if(count_in < 5'd31)
                begin
                    cs <= s9;
                    count_in <= count_in + 1'b1;
                end
            else
                begin
                    cs <= s1;
                    count_in <= 5'd0;
                end
        end
end

s2: // update 1
    begin
counter1 <= 4'd2;
read <= 1'b1;
write <= 1'b1;
temp_e1 <= e1 + (14 * o1);
temp_e2 <= e2 + (14 * o2);
// state transition
cs <= s3;
end
s3: // predict 1
begin
  counter1 <= 4'd3;
temp_o1 <= o1 - 3 * temp_e1 + (e0>>1); // shift operator has highest priority
temp_o2 <= o2 - 3 * temp_e2 + (temp_e1>>1); // as above
  // state transition
cs <= s4;
end
  // special condition
  // first odd component = first odd component - (7* first even component) + last even component
s4: // update 2
begin
  temp_c1 <= temp_e1 - temp_o2;
  counter1 <= 4'd4;
  // state transition
cs <= s5;
end
  // special condition
  // last even component = last even component - first odd component
s5: // normalize
begin
  norm_e1 <= 4 * temp_c1;
  norm_o1 <= 16 * temp_o1;
  counter1 <= 4'd5;
  // state transition
cs <= s6;
end
s6: // shifting by 8 which is divide by 256 = 8*8*8 as we are multiplying by 8 in each stage
begin
  shift_e1 <= norm_e1>>4;
  shift_o1 <= norm_o1>>4;
  counter1 <= 4'd6;
  // state transition
cs <= s7;
end
s7: // Truncation
begin
final_e1 <= {shift_e1[25],shift_e1[17:3]};
out_1 <= {shift_o1[25],shift_o1[17:3]}; // writing odd component start address
00_0001h
   count2 <= 2'd2;
      read <= 1'b1;
      write <= 1'b0;
      address <= w_address;
      counter1 <= 4'd7;
   //cs <= s14; // state transition
      if(count_in < 5'd25)
        begin
          count_in <= count_in + 1'b1;
          cs <= s7;
          //w_address <= w_address;
        end
      else
        begin
          count_in <= 5'd0;
        end
   cs <= s14;
end
s14:begin
   read <= 1'b1;
      write <= 1'b0;
      address <= w_address + 1'b1;
      out_1 <= final_e1; // writing even component start address 00_0002h
//w_address <= w_address + 1'b1;
cs <= s8;
end
/*s12: begin
   read <= 1'b0;
   write <= 1'b1;
   address <= r_address;
   w_address <= w_address + 1'b1;
   counter1 <= 4'd12;
cs <= s8;
s8:begin // getting next two components shifting the register by 2
if(w_address < 22'h10_0001) // max_write

begin
w_address <= w_address + 2'd2;
cs <= s10;
end
else
begin
w_address <= w_address;
cs <= s13;
end

o2 <= in_1;
e2 <= o2;
o1 <= e2;
e1 <= o1;
e0 <= temp_e1;
counter1 <= 4'd8;
write <= 1'b1;
read <= 1'b1;
count2 <= count2 - 1'b1;
if(r_address < 22'h20_0000)
levell <= 2'd1;
else if((r_address > 22'h20_0000) && (r_address < 22'h30_0001))
levell <= 2'd2;
else if((r_address > 22'h30_0001) && (r_address < 22'h38_0002))
levell <= 2'd3;
else
levell <= 2'd0;
end

s10: // 2 shift control getting next two components
begin
counter1 <= 4'd10;
if(count2 == 2'd0)
cs <= s2; // getting next two components
else
begin
if(count_in < 5'd31)
begin
count_in <= count_in + 1'b1;
cs <= s10;
end
end

else

begin

count_in <= 5'd0;

cs <= s2;
end

end

s13: begin

led1 <= 1'b0;

counter1 <= 4'd13;

cs <= s13;

trigger <= 1'b1;

end

default: begin

cs <= s0;
end
case

default: begin

end

endmodule

COMPRESSION LEVEL 2 AND LEVEL 3

module top_code_2_3(clk, reset, in_1, out_1, read, write, address, led1, counter1, levell);

input clk, reset;
input signed [15:0] in_1;
output reg signed [15:0] out_1;
output reg read, write, led1;
output reg [21:0] address;

reg signed [15:0] o1, o2, e1, e2;
reg signed [22:0] temp_e1, temp_e2, e0;
reg signed [28:0] temp_o1, temp_o2, temp_c1;
reg signed [34:0] norm_e1, norm_o1;
reg signed [25:0] shift_e1, shift_o1;
reg signed [15:0] final_e1, final_o1;
reg [2:0] count1;
reg [1:0] count2;
reg [3:0] cs;
reg [21:0] w_address, r_address;
output reg [3:0] counter1;
output reg [1:0] level1;

parameter s0 = 4'd0,
    s1 = 4'd1,
    s2 = 4'd2,
    s3 = 4'd3,
    s4 = 4'd4,
    s5 = 4'd5,
    s6 = 4'd6,
    s7 = 4'd7,
    s8 = 4'd8,
    s9 = 4'd9,
    s10 = 4'd10,
    s11 = 4'd11,
    s12 = 4'd12,
    s13 = 4'd13,
    s14 = 4'd14;

//assign out_1 = shift_e1; // this is the output of 1st level- To be written in to SDRAM
//assign out_1 = final_e1;

always@(posedge clk or negedge reset)
begin
    if (!reset)
        cs <= s0;
    else
        begin
            case(cs)
                s0: begin // Ideal state all reset
                    o2 <= 0;
                    e2 <= 0;
                    o1 <= 0;
                    e1 <= 0;
                    e0 <= 23'd0;
                    count1 <= 3'd4;
                    count2 <= 2'd2;
                    address <= 22'd0;
                    w_address <= 22'h10_0002; //first
                end
            endcase
            even data for level 2 on 10_0002h
            r_address <= 22'h00_0002; //first read address
            00_0002 -> first level compressed even component.
            read <= 1'b1;
            write <= 1'b1;
            led1 <= 1'b1;
            counter1 <= 4'd0;
        end
end
levell <= 2'd0;
// state transition

cs <= s11;
end

s11: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    counter1 <= 4'd11;
    cs <= s1;
end

s1: begin  // loading the registers and shifting 4 times
    o2 <= in_1; // 3rd sound component
    e2 <= o2;    // 2nd sound component
    o1 <= e2;    // 1st sound component
    e1 <= o1;    // 0th sound component
    // e0<=0;
    count1 <= count1 - 1'b1;
    r_address <= r_address + 2'd2;  // read address incremented by 2 to access only even components
    counter1 <= 4'd1;
    // state transition
    cs <= s9;
end

s9: begin  // shifting control state
    counter1 <= 4'd9;
    if(count1 == 4'd0)
        cs <= s2;
    else
        cs <= s1;
end

s2: // update 1
    begin
        counter1 <= 4'd2;
        read <= 1'b1;
        write <= 1'b1;
        temp_e1 <= e1 + (14 * o1);
        temp_e2 <= e2 + (14 * o2);
        // state transition
        cs <= s3;
    end
s3: // predict 1
    begin
        counter1 <= 4'd3;
        temp_o1 <= o1 - 3* temp_e1 + (e0)>>1; // shift operator has highest priority
        temp_o2 <= o2 - 3* temp_e2 + (temp_e1)>>1; // as above
        // state transition
        cs <= s4;
    end
    // special condition
    // first odd component = first odd component - (7* first even component) + last even component
s4: // update 2
    begin
        temp_c1 <= temp_e1 - temp_o2;
        counter1 <= 4'd4;
        // state transition
        cs <= s5;
    end
    // special condition
    // last even component = last even component - first odd component
s5: // normalize
    begin
        norm_e1 <= 4 * temp_c1;
        norm_o1 <= 16 * temp_o1;
        counter1 <= 4'd5;
        // state transition
        cs <= s6;
    end
s6: // shifting by 8 which is divide by 256 = 8*8*8 as we are multiplying by 8 in each stage
    begin
        shift_e1 <= norm_e1>>4;
        shift_o1 <= norm_o1>>4;
        counter1 <= 4'd6;
        // state transition
        cs <= s7;
    end
s7: // Truncation
    begin
        // final_e1 <= shift_e1[25:10]; // have to work on kit to make sure which bits are working
        // out_1 <= shift_e1[25:10];
        // out_1 <= {shift_e1[25],shift_e1[14:0]};
out_1 <= {shift_e1[25], shift_e1[17:3]};  // writing even components starting
address 10_0002h
final_o1 <= {shift_o1[25], shift_o1[17:3]};
count2 <= 2'd2;
read <= 1'b1;
write <= 1'b0;
address <= w_address;
counter1 <= 4'd7;
cs <= s14; // state transition
end
s14: begin
read <= 1'b1;
write <= 1'b0;
address <= w_address + 1'b1;
// writing odd
components starting address 10_003h.
out_1 <= final_o1;
// w_address <= w_address + 1'b1;
cs <= s12;
end

s12: begin
read <= 1'b0;
write <= 1'b1;
address <= r_address;
w_address <= w_address + 2'd2;
counter1 <= 4'd12;
cs <= s8;
end
s8: begin  // getting next two components shifting the register by 2
o2 <= in_l;
e2 <= o2;
o1 <= e2;
e1 <= o1;
e0 <= temp_e1;
counter1 <= 4'd8;
counter2 <= count2 - 1'b1;
// if(r_address < 22'h 20_0001)
if(r_address < 22'h 18_0003)  // level 2 last address:
18_0003h and level 3 last address: 1c_0003h
begin
r_address <= r_address + 2'd2;
// state transition
cs <= s10;
end
else
begin
  r_address <= r_address;
  cs <= s13;
end
if(r_address < 22'h20_0000)
  levell <= 2'd1;
else if((r_address > 22'h20_0000) && (r_address <
  22'h30_0001))
  levell <= 2'd2;
else if((r_address > 22'h30_0001) && (r_address <
  22'h38_0002))
  levell <= 2'd3;
else
  levell <= 2'd0;

s10: // 2 shift control getting next two components
begin
  counter1 <= 4'd10;
  if(count2 == 2'd0)
    cs <= s2; // getting next two components
  else
    cs <= s8;
end
s13: begin
  led1 <= 1'b0;
  counter1 <= 4'd13;
  cs <= s13;
end
default: cs <= s0;
endcase
end
endmodule

DECOMPRESSION Code

module top_dac(rd_addr,o_data,iAUD_BCK,dac_out,iAUD_LRCK,reset);
input reset;
input iAUD_BCK;
input iAUD_LRCK;
input [15:0] o_data;
output reg [21:0] rd_addr;
output reg dac_out;
reg [4:0] cnt;
reg [31:0] temp;
reg [1:0] cs;
reg [4:0] index;
 progressDialog[1:0] flag;
reg flag;

parameter s0 = 2’d0,
        s1 = 2’d1,
        s2 = 2’d2,
        s3 = 2’d3;

always @(posedge iAUD_BCK or negedge reset) begin
    if(!reset) begin
        rd_addr <= 22’h20_0001; // level3 reconstructed data
        rd_addr <= 22’h20_0000;
//        rd_addr <= 22’h38_0003;
//        rd_addr <= 22’d0;
//        rd_addr <= 22’d00_0001;
        cnt <= 5’d0;
        cs <= s0;
        index <= 5’d31;
        flag <= 1’d0;
    end
    else begin
        case(cs)
            s0: begin
                temp[31:16] <= o_data;
                if(iAUD_LRCK == 1'b1)
                    cs <= s1;
                else
                    cs <= s0;
            end
            s1: begin
                if(iAUD_LRCK == 1'b0)
                    cs <= s2;
                else
.
module recons_1(clk, reset, in_l, out_l, read, write, address, led_l, counter_l, levell);
    input clk, reset;
    input signed [15:0] in_l;
    output reg signed [15:0] out_l;
    output reg read, write, led_l;
    output reg [21:0] address;
    reg signed [15:0] o1, o2, e1, e2;
    reg signed [19:0] temp_e1;
    reg signed [15:0] temp_o1;

    // RECONSTRUCTION

    endmodule

    end

    end

    end

    end
reg signed [13:0] temp_c1,e0;
reg signed [13:0] denorm_e1,denorm_e2;
reg signed [11:0] denorm_o1,denorm_o2;
reg signed [15:0] shift_e1,shift_o1;
reg signed [15:0] final_e1,final_o1;
reg [2:0] count1;
reg [1:0] count2;
reg [5:0] cs;
reg [21:0] w_address, r_address;
output reg [3:0] counter1;
output reg [1:0] level1;

parameter s0 = 6'd0,
s1 = 6'd1,
s2 = 6'd2,
s3 = 6'd3,
s4 = 6'd4,
s5 = 6'd5,
s6 = 6'd6,
s7 = 6'd7,
s8 = 6'd8,
s9 = 6'd9,
s10 = 6'd10,
s11 = 6'd11,
s12 = 6'd12,
s13 = 6'd13,
s14 = 6'd14,
s15 = 6'd15,
s16 = 6'd16,
s17 = 6'd17,
s18 = 6'd18,
s19 = 6'd19,
s20 = 6'd20,
s21 = 6'd21,
s22 = 6'd22,
s23 = 6'd23,
s24 = 6'd24,
s25 = 6'd25,
s26 = 6'd26,
s27 = 6'd27,
s28 = 6'd28,
s29 = 6'd29,
s30 = 6'd30,
s31 = 6'd31,
s32 = 6'd32,
s33 = 6'd33,
s34 = 6'd34,
s35 = 6'd35,
s36 = 6'd36,
s37 = 6'd37,
s38 = 6'd38,
s39 = 6'd39,
s40 = 6'd40,
s41 = 6'd41,
s42 = 6'd42,
s43 = 6'd43,
s44 = 6'd44,
s45 = 6'd45,
s46 = 6'd46;

always@ (posedge clk or negedge reset)
begin
  if (!reset)
    cs <= s0;
  else
    begin
      case(cs)
        s0: begin // Idle state all reset
          o2 <= 0;
          e2 <= 0;
          o1 <= 0;
          e1 <= 0;
          e0 <= 15'd0;
          count1 <= 3'd4;
          count2 <= 2'd2;
          address <= 22'd0;
          w_address <= 22'h10_0002;
          r_address <= 22'h18_0003; // reading third level compressed data starting at 18_0003h.
          read <= 1'b1;
          write <= 1'b1;
          led1 <= 1'b1;
          counter1 <= 4'd0;
          levell <= 2'd3;
          // state transition
        end
        // state transitions...
      endcase
    end
  end
end
s11: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    counter1 <= 4'd11;
    cs <= s1;
end

s1: begin // loading the registers and shifting 4 times
    e2 <= in_1; // 3rd sound component
    o2 <= e2; // 2nd sound component
    e1 <= o2; // 1st sound component
    o1 <= e1; // 0th sound component
    count1 <= count1 - 1'b1;
    r_address <= r_address + 1'b1;
    counter1 <= 4'd1;
    // state transition
    cs <= s9;
end

s9: begin // shifting control state
    counter1 <= 4'd9;
    if(count1 == 4'd0)
        cs <= s2;
    else
        cs <= s1;
end

s2: // denormalize
    begin
        read <= 1'b1;
        write <= 1'b1;
        denorm_e1 <= e1 >> 2;
        denorm_o1 <= o1 >> 4;
        denorm_e2 <= e2 >> 2;
        denorm_o2 <= o2 >> 4;
        cs <= s3;
        counter1 <= 4'd2;
    end

s3: // update 2
    begin
        temp_c1 <= denorm_e1 + denorm_o2;
        counter1 <= 4'd3;
    // state transition
cs <= s4;
end
s4: // predict 1
begin
    temp_o1 <= denorm_o1 + 3* temp_c1 - (e0>>1); // shift operator has highest priority
    temp_o1-18 bit
    counter1 <= 4'd4;
    // state transition
    cs <= s5;
end

s5: // update 1
begin
    counter1 <= 4'd5;
    temp_e1 <= temp_c1 - (14*temp_o1); // temp_e1 -- 22 bit
    // state transition
    cs <= s6;
end
s6: // dividing each by 16
begin
    shift_e1 <= {temp_e1[19],temp_e1[16:2]}; // shift_e1 will be sixteen bit
    shift_o1 <= {temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[11:0]}; // shift_o1 will be 16 bit
    counter1 <= 4'd6;
    // state transition
    cs <= s7;
end
s7: // Truncation
begin
    final_e1 <= shift_e1[25:10]; // have to work on kit to make sure which bits are working
    out_1 <= shift_e1[25:10];
    out_1 <= {shift_e1[25],shift_e1[14:0]};
    final_e1 <= shift_e1;
    out_1 <= shift_o1; // writing odd components
    count2 <= 2'd2;
    read <= 1'b1;
    write <= 1'b0;
    address <= w_address;
    counter1 <= 4'd7;
    cs <= s14; // state transition
end
s14: begin
    out_1 <= final_e1;
    address <= w_address + 2'd2;
    read <= 1'b1;
    write <= 1'b0;
    counter1 <= 4'd14;
    cs <= s12;
end
s12: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    w_address <= w_address + 3'd4;
    counter1 <= 4'd12;
    cs <= s8;
end
s8: begin // getting next two components shifting the register by 2
    e2 <= in_1;
    o2 <= e2;
    e1 <= o2;
    o1 <= e1;
    e0 <= temp_c1;
    counter1 <= 4'd8;
    count2 <= count2 - 1'b1;
    //if(r_address < 22'h 20_0001)
    if(r_address < 22'h 1c_0003) // level3 address of last compressed component : 1c_0003h
        begin
            r_address <= r_address + 1'b1;
        // state transition
        cs <= s10;
        end
    else
        begin
            r_address <= r_address;
            cs <= s13;
        end
        /*if(r_address < 22'h20_0000)
        level1 <= 2'd1;
        else if((r_address > 22'h20_0000) && (r_address <
22'h30_0001))
            level2 <= 2'd2;*/
else if((r_address > 22'h30_0001) && (r_address < 22'h38_0002))
    levell <= 2'd3;
else
    levell <= 2'd0; /*
end

s10: // 2 shift control getting next two components
    begin
    counterl <= 4'd10;
    if(count2 == 2'd0)
        cs <= s2; // getting next two components
    else
        cs <= s8;
    end

s13: begin
    ledl <= 1'b0;
    counterl <= 4'd15;
    levell <= 2'd0;
   /cs <= s15;
end

///////////////////////////////////////////////////////////////////////////

s15: begin // Idle state all reset
    o2 <= 0;
    e2 <= 0;
    o1 <= 0;
    e1 <= 0;
    e0 <= 15'd0;
    count1 <= 3'd4;
    count2 <= 2'd2;
    address <= 22'd0;
    w_address <= 22'h00_0002;
    r_address <= 22'h10_0002; // reading third level compressed
    data starting at 18_0003h.
    read <= 1'b1;
    write <= 1'b1;
    led1 <= 1'b1;
    counterl <= 4'd0;
    levell <= 2'd1;
    // state transition
    cs <= s16;
end
s16: begin
  read <= 1'b0;
  write <= 1'b1;
  address <= r_address;
  counter1 <= 4'd11;
  cs<= s17;
end

s17: begin
  // loading the registers and shifting 4 times
  o2 <= in_1;  // 3rd sound component
  e2 <= o2;   // 2nd sound component
  o1 <= e2;   // 1st sound component
  e1 <= o1;   // 0th sound component
  count1 <=count1 - 1'b1;
  r_address <= r_address + 1'b1;
  counter1 <= 4'd1;
  // state transition
  cs<= s18;
end

s18: begin
  // shifting control state
  counter1 <= 4'd9;
  if(count1 == 4'd0)
    cs <= s19;
  else
    cs<= s17;
end

s19: // denormalize
  begin
    read <= 1'b1;
    write <= 1'b1;
    denorm_e1 <= e1 >> 2;
    denorm_o1 <= o1 >> 4;
    denorm_e2 <= e2 >> 2;
    denorm_o2 <= o2 >> 4;
    cs <=s20;
    counter1 <= 4'd2;
  end

s20: // update 2
  begin
    temp_c1 <= denorm_e1 + denorm_o2;
    counter1 <= 4'd3;
  // state transition
cs<= s21;  
end  
s21: // predict 1  
begin  
temp_o1 <= denorm_o1 + 3* temp_c1 - (e0>>1); // shift operator has highest priority  
temp o1-18 bit  
counter1 <= 4'd4;  
// state transition  
cs<= s22;  
end  
s22: // update 1  
begin  
counter1 <= 4'd5;  
temp_e1 <= temp_c1 - (14 * temp_o1); // temp_e1 -- 22 bit  
// state transition  
cs <= s23;  
end  
s23: // dividing each by 16  
begin  
shift_e1 <= {temp_e1[19],temp_e1[16:2]} ; //shift_e1 will be sixteen bit  
shift_o1 <= {temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[11:0]} ; //shift_o1 will be 16 bit  
counter1 <= 4'd6;  
// state transition  
cs<= s24;  
end  
s24: // Truncation  
begin  
//final_e1 <= shift_e1[25:10]; // have to work on kit to make sure which bits are working  
//out_1 <= shift_e1[25:10];  
//out_1 <= {shift_e1[25],shift_e1[14:0]};  
final_e1 <= shift_e1;  
out_1 <= shift_o1; // writing odd components  
count2 <= 2'd2;  
read <= 1'b1;  
write <= 1'b0;  
address <= w_address;  
counter1 <= 4'd7;  
cs <= s25; // state transition  
end
s25: begin
    out_1 <= final_e1;
    address <= w_address + 2'd2;
    read <= 1'b1;
    write <= 1'b0;
    counter1 <= 4'd14;
    cs <= s26;
    end
s26: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    w_address <= w_address + 3'd4;
    counter1 <= 4'd12;
    cs <= s29;
    end
s29: begin // getting next two components  shifting the register by 2
    e2 <= in_1;
    o2 <= e2;
    e1 <= o2;
    o1 <= e1;
    e0 <= temp_c1;
    counter1 <= 4'd8;
    count2 <= count2 - 1'b1;
    //if(r_address < 22'h 20_0001)
    if(r_address < 22'h 18_0002) // level3 address of last compressed component: lc_0003h
        begin
            r_address <= r_address + 1'b1;
        end
    // state transition
    cs <= s30;
    end
else begin
    r_address <= r_address;
    cs <= s31;
    end
/*if(r_address < 22'h20_0000)
levell <= 2'd1;
else if((r_address > 22'h20_0000) && (r_address < 22'h30_0001))
levell <= 2'd2;*/
else if((r_address > 22'h30_0001) \&\& (r_address < 22'h38_0002))
levell <= 2'd3;
else
levell <= 2'd0;*/

end
s30: // 2 shift control getting next two components
begin
counter1 <= 4'd10;
if(count2 == 2'd0)
cs <= s19; // getting next two components
else
cs<= s29;
end
s31: begin
led1 <= 1'b1;
counter1 <= 4'd00;
cs <=s32;
levell <= 2'd10;
end

/////////////////////////////////////////// level 3 Reconstruction///////////////////////////////////////////
s32: begin // Idle state all reset
o2 <= 0;
e2 <= 0;
o1 <= 0;
e1 <= 0;
e0 <= 15'd0;
count1 <=3'd4;
count2 <=2'd2;
address <= 22'd0;
w_address <= 22'h20_0000;
r_address <= 22'h00_0001; //reading third level compressed

data starting at 18_0003h.
read <= 1'b1;
write <= 1'b1;
led1 <= 1'b1;
counter1<= 4'd0;
levell <= 2'd2;
// state transition

cs <=s33;
end
s33: begin
    read <= 1'b0;
    write <= 1'b1;
    address <= r_address;
    counter1 <= 4'd11;
    cs <= s34;
end

s34: begin  // loading the registers and shifting 4 times
    e2 <= in_1;  // 3rd sound component
    o2 <= e2;    // 2nd sound component
    e1 <= o2;    // 1st sound component
    o1 <= e1;    // 0th sound component
    count1 <= count1 - 1'b1;
    r_address <= r_address + 1'b1;
    counter1 <= 4'd1;
    // state transition
    cs <= s35;
end

s35: begin  // shifting control state
    counter1 <= 4'd9;
    if(count1 == 4'd0)
        cs <= s36;
    else
        cs <= s34;
end

s36: // denormalize
    begin
        read <= 1'b1;
        write <= 1'b1;
        denorm_e1 <= e1 >> 2;
        denorm_o1 <= o1 >> 4;
        denorm_e2 <= e2 >> 2;
        denorm_o2 <= o2 >> 4;
        cs <= s37;
        counter1 <= 4'd2;
    end

s37: // update 2
    begin
        temp_c1 <= denorm_e1 + denorm_o2;
        counter1 <= 4'd3;
    // state transition
cs <= s38;
end
  s38: // predict 1
  begin
    temp_o1 <= denorm_o1 + 3* temp_c1 - (e0>>1); // shift operator has highest priority temp o1-18 bit
    counter1 <= 4'd4;
    // state transition
    cs <= s39;
  end
  s39: // update 1
  begin
    counter1 <= 4'd5;
    temp_e1 <= temp_e1 - (14* temp_o1); // temp_e1 -- 22 bit
    // state transition
    cs <= s40;
  end
  s40: // dividing each by 16
  begin
    shift_e1 <= {temp_e1[19],temp_e1[16:2]} ; //shift_e1 will be sixteen bit
    shift_o1 <= {temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[15],temp_o1[11:0]} ; //shift_o1 will be 16 bit
    counter1 <= 4'd6;
    // state transition
    cs <= s41;
  end
  s41: // Truncation
  begin
    //final_e1 <= shift_e1[25:10]; // have to work on kit to make sure which bits are working
    //out_1 <= shift_e1[25:10];
    //out_1 <= {shift_e1[25],shift_e1[14:0]};
    final_e1 <= shift_e1;
    out_1 <= shift_o1; // writing odd components
    count2 <= 2'd2;
    read <= 1'b1;
    write <= 1'b0;
    address <= w_address;
    counter1 <= 4'd7;
    cs <= s42; // state transition
  end
s42: begin
  out_1 <= final_e1;
  address <= w_address + 2'd1;
  read <= 1'b1;
  write <= 1'b0;
  counterl <= 4'd14;
  cs <= s43;
end
s43: begin
  read <= 1'b0;
  write <= 1'b1;
  address <= r_address;
  w_address <= w_address + 2'd2;
  counterl <= 4'd12;
  cs <= s44;
end
s44: begin // getting next two components shifting the register by 2
  e2 <= in_1;
  o2 <= e2;
  e1 <= o2;
  o1 <= e1;
  e0 <= temp_c1;
  counterl <= 4'd8;
  count2 <= count2 - 1'b1;
  if(r_address < 22'h 10_0001) // level3 address of last compressed component : 1c_0003h
    r_address <= r_address + 1'b1;
  end
  else begin
    r_address <= r_address;
    cs <= s46;
  end
  /*if(r_address < 22'h20_0000)
  level1 <= 2'd1;
  else if((r_address > 22'h20_0000) && (r_address < 22'h30_0001))
    level1 <= 2'd2;*/
  end

else if((r_address > 22'h30_0001) && (r_address < 22'h38_0002))
    levell <= 2'd3;
else
    levell <= 2'd0; /*
end

s45: // 2 shift control getting next two components
    begin
    counter1 <= 4'd10;
    if(count2 == 2'd0)
        cs <= s36; // getting next two components
    else
        cs <= s44;
    end

s46: begin
    led1 <= 1'b1;
    counter1 <= 4'd13;
    cs <= s46;
    levell <= 2'd0;
end

default: cs <= s46;
endcase
end
endmodule