DESIGN AND IMPLEMENTATION OF FPGA BASED DSP GRAPHIC EQUALIZER AND A DIRECT-MAPPED CACHE

EN3030
Circuits and Systems Design

GROUP MEMBERS

150001C : G.Abarajithan
150172A : T.T.Fonseka
150689N : W.M.R.R.Wickramasinghe
150707V : C.Wimalasuriya

Contents

1 Graphic Equalizer: An Introduction

> Equalization or equalisation is the process of adjusting the balance between frequency components within an electronic signal. The most well-known use of equalization is in sound recording and reproduction but there are many other applications in electronics and telecommunications. The circuit or equipment used to achieve equalization is called an equalizer. These devices strengthen (boost) or weaken (cut) the energy of specific frequency bands or 'frequency ranges'
>
> *–Wikipedia–*

The graphic equalizer is a side project alongside the main circuit and system design project of image down sampling processor. In this project we implemented an equalizer which can in real time process an audio stream, separating it into frequency bands and allowing the user to selectively attenuate or amplify these bands.

To implement the given task, we decided to use an Altera DE2 115 FPGA development board due to the existence of a sophisticated audio interface on the board itself, the Wolfson WM8731 audio CODEC. The audio signal was extracted through this codec using a custom audio signal parser.

The extracted signal was then processed on our DSP logic, containing a pair of FIR filters, for preservation of stereo. The filters are generated on the fly in a manner specified by the control interface and attenuates specific frequencies in the magnitude specified by the user. When the attenuation values change, the filter is instantly reconfigured to accommodate the new equalization. The result from the filter pair is then fed back to the audio parser, which in turn hands over the data to the Wolfson codec, which performs DAC conversion on the result and outputs the signal.
Testing was done using normal music with a clear frequancy division, such as songs with heavy bass instruments alongside with high pitched vocals. For demonstration, a frequancy generator was used to generate a single frequancy tone and it was observed on an oscilloscope while changing both filter charachteristichs and input frequency.

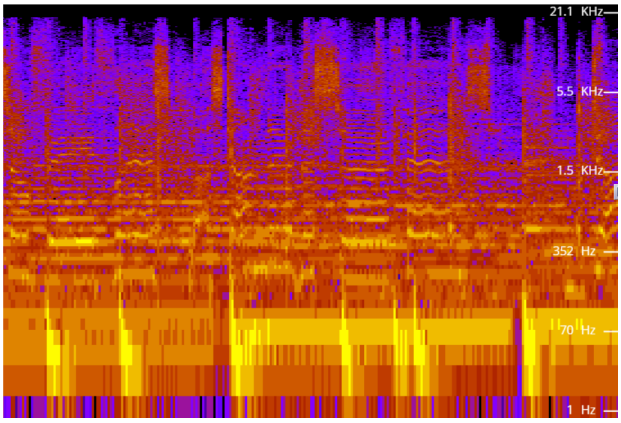


Figure 1.1: Time-Frequency analysis of "Numb" by Linkin park, one of the audio tracks used to test the equalizer.

2 Graphic Equalizer: Implementation

2.0.1 Hardware Overview

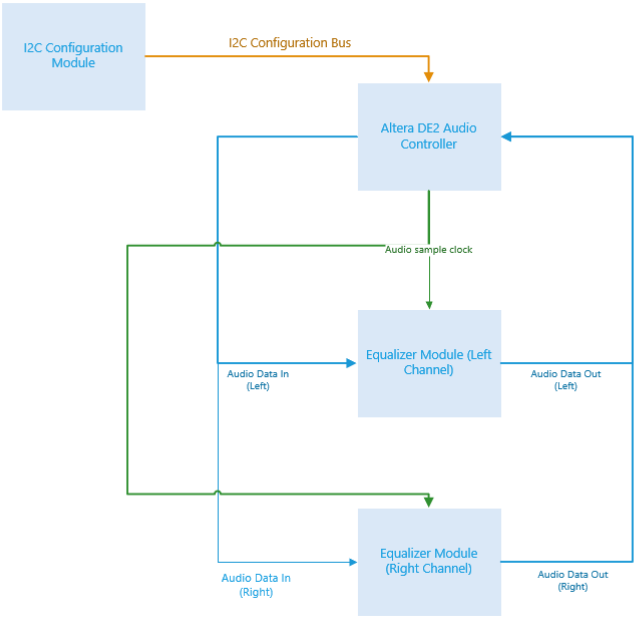


Figure 2.1: Top level block diagram

The overview of the equalizer is actually rather simple. The audio signal is extracted through the Altera audio controller and the signal parser encapsulating it. The behavior of the controller is governed by the configuration given by an I2C config module. The extracted audio streams is separately processed channelwise to preserve stereo and the results are fed back to the audio controller. Equalizer modules are governed by the 'audio clock' from the audio controller, which signifies that a new audio sample has fully stabilized on the input channels.

2.0.2 Audio Interface

Altera DE2-115 provides a sophisticated audio interface with Line in, Line out and Mic in ports, much similar to an advanced sound card. These ports are connected to the Wolfson audio CODEC which performs a quick and accurate DAC/ADC conversion and makes the data available to the FPGA.

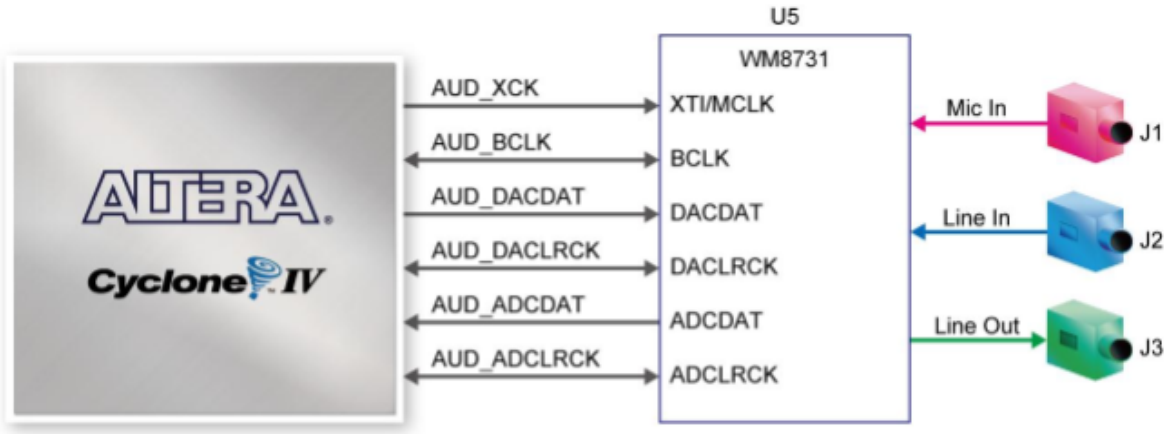


Figure 2.2: DE2-115 Audio Interface

The audio data travels along the ADCDAT and DACDAT for input and output streams respectively. The sample rate is 48 kHz and each sample has a pair of 32 bit signed integers, for preservation of stereo channeling. The pair of samples are separated by the LRCLK line, which alternates between logic high and low to indicate whether the sample is for the left channel or the right channel. The data flows in a serial format which is hard to process and the existence of two samples for left and right channels alternatively makes the process even harder. To avoid this complexity, we use the signal parser to convert this serial stream to a pair of parellel channels which are easy to manipulate.

2.0.3 I2C Configuration

The Wolfson audio codec can be configured to route its audio without ever actually making its way to the CODEC units. And unfortunately, the module is by default configured to directly output the input from the line in jack to the line out jack. This behavior can be changed by adding a I2C config module to the controller interface. The module instructs the CODEC to make the data available to the FPGA for processing rather than directly outputting. It performs the configuration process upon power on and goes idle after that.

2.0.4 Signal Parser

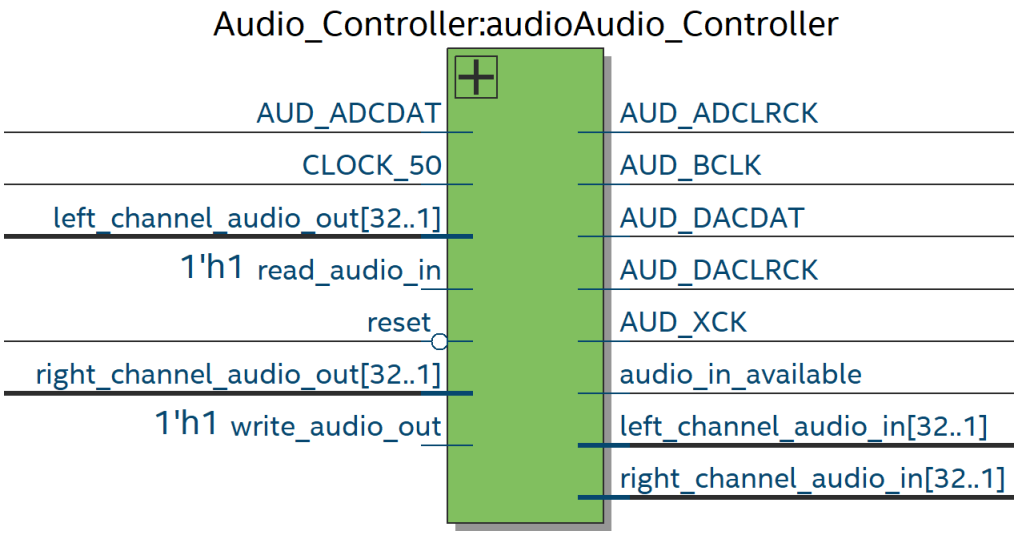


Figure 2.3: Signal Parser

As clearly evident in the above diagram, the parser directly connects to the interface of the Wolfson controller CODEC. It reads the serial data from the ADCDAT line and converts it to a more usable pair of parellel data streams in 32 bit signed integer format. These data are available in the left/right channel audio in buses. whenever the module fully stabilizes an audio sample on these buses, it provides a "Tick" on the audio in available line, which we currently use as a clock for validating samples to the DSP circuitry.

As for data output, it accepts a pair of 32 bit signed integers on the left/right channel audio out buses. The data on these lines are then serialized and sent to the Wolfson CODEC, which performs the DAC conversion and outputs them on the line out port. The whole module can be reset using the reset line, which is useful if the internal buffers overflow due to some reason.

This controller is a publicly available project of Department of Electrical and computer engineering of university of Toronto. Full link given in references.

2.0.5 Filters

The filtration of audio signals is done through a single filter which combines three FIR digital filters. A low pass filter of 500 Hz cutoff, a bandpass filter of 500-2000 Hz and a 2000 Hz high pass filter. These frequency bands are chosen to represent the audible Bass, Midtone and Treble frequencies. The exact computation of these filter coefficients are discussed in chapter 5. The coefficients of these filters are then multiplied by the necessary gain and added together to form the final filter, exploiting the linear property of the filter systems

For example, if the three filters are repectively $F_1(f)$, $F_2(f)$ and $F_3(f)$, we take an input $G(f)$. For user selected gains k_1,k_2 and k_3, the final output $Fout(f)$ should be,

$$F_{out}(f) = k_1F_1(f)G(f) + k_2F_2(f)G(f) + k_3F_3(f)G(f) \tag{2.1}$$

But this implementation would require a lot of hardware resources since three filters would need to operate in unison. When the linear property is applied, we can rewrite $k_1F_1(f) + k_2F_2(f) + k_3F_3(f)$ in to a single filter $F(f)$ and then we get,

$$F_{out}(f) = F(f)G(f) \tag{2.2}$$

which is much simpler.

When converted in to time domain, this equation becomes,

$$f_{out}(t) = f(t) * g(t) \tag{2.3}$$

which is linear convolution. Linear convolution can be implemented on hardware as a shifting bank of registers. The hardware implementation can be represented as follows.

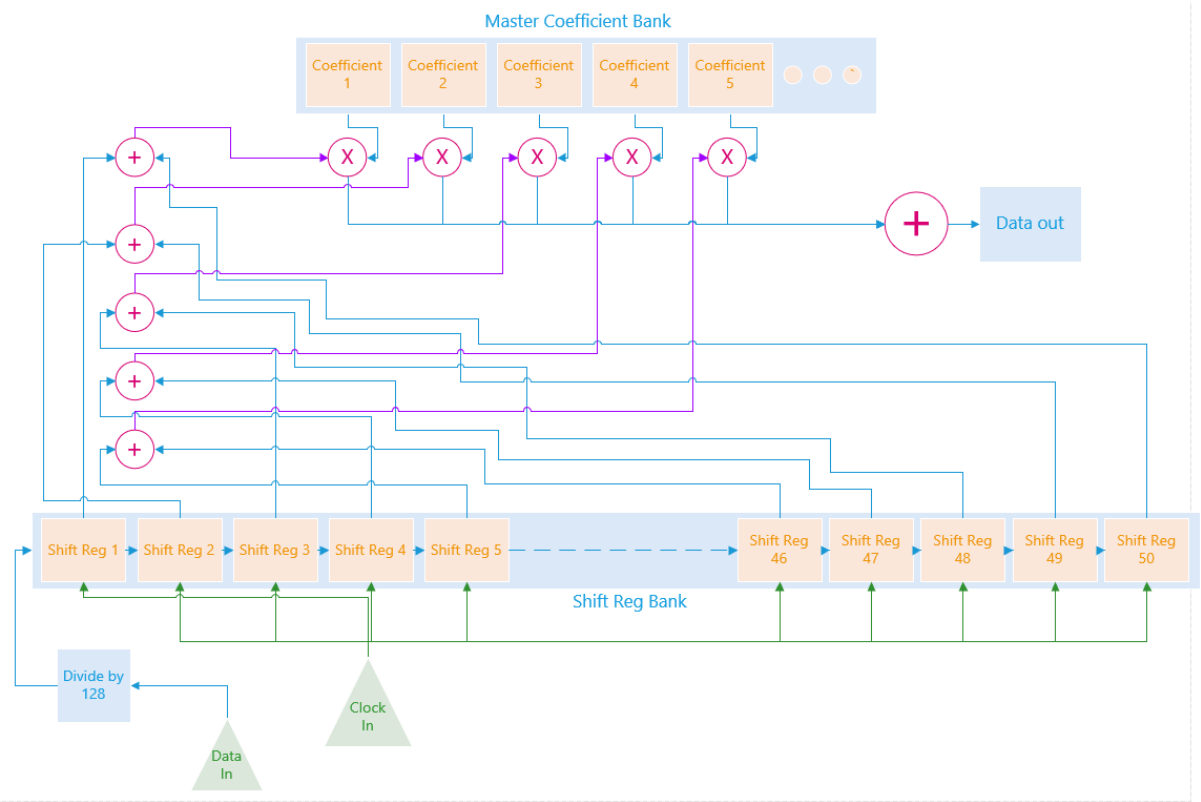


Figure 2.4: FIR filter hardware implementation

The implementation is simply a bank of data registers, which shift the data forward with the arrival of a new sample. There is another coefficient bank, which holds the coefficient values of the designed filter folded in half (first 26 coefficients if the filter width is 51). With the arrival of a new sample, the coefficients are multiplied with the data and added together to form the final output. This implementation causes a delay of 51 samples but with the practical 48 kHz sampling rate, this is as close as it gets to a real-time output.

2.0.6 Coefficient Generation

It was explained earlier that this equalizer implementation only uses one filter per channel as opposed to the classic three. This is achieved through regeneration of filter coefficients every time the equalizer settings are adjusted.

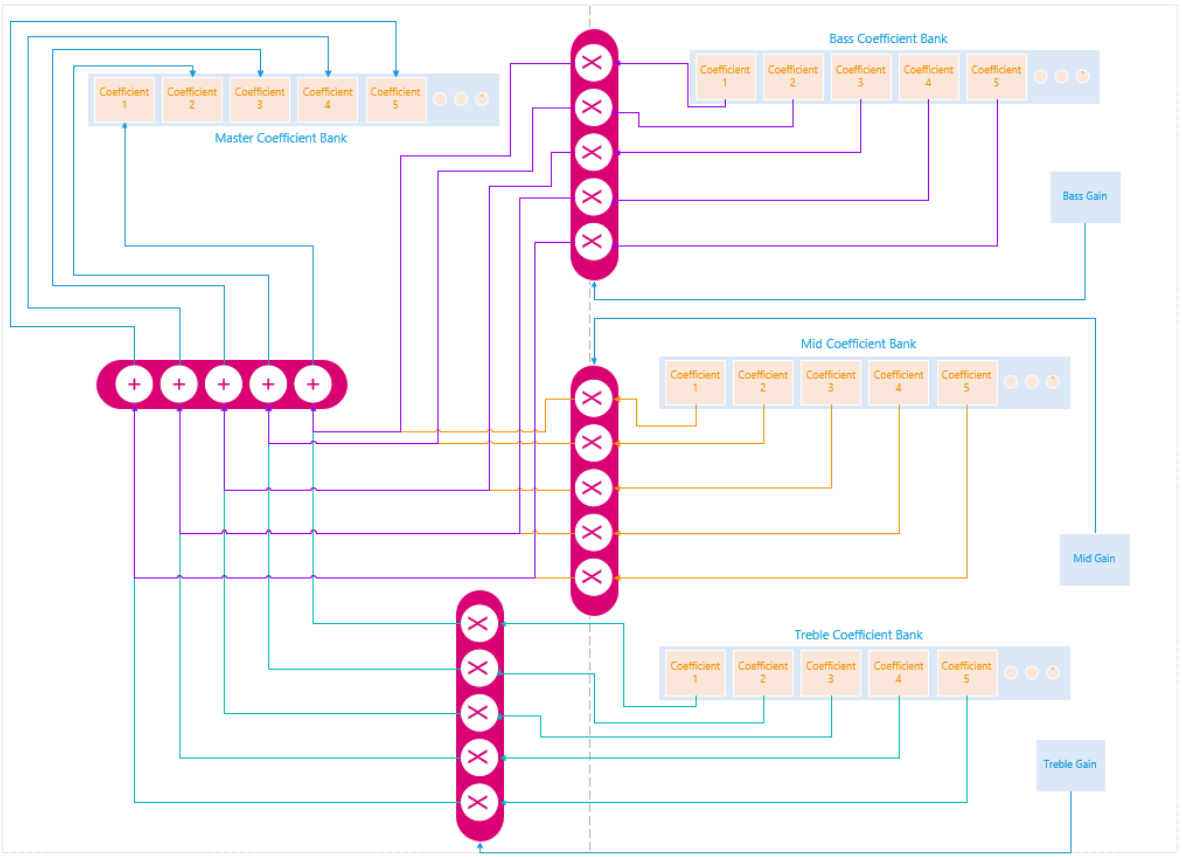


Figure 2.5: Coefficient Generator

The bass, mid and treble coefficient banks are hardcoded into the DSP logic. What the user can set are the gain values for each band. Once the user sets these gain values and confirms the input, equalizer multiplies the value of each coefficient bank by its respective gain and adds them together to generate the master coefficient bank which we use for filtering the audio stream. The linear property of the signal makes sure that the new filter exactly matches the gain values set by the user.

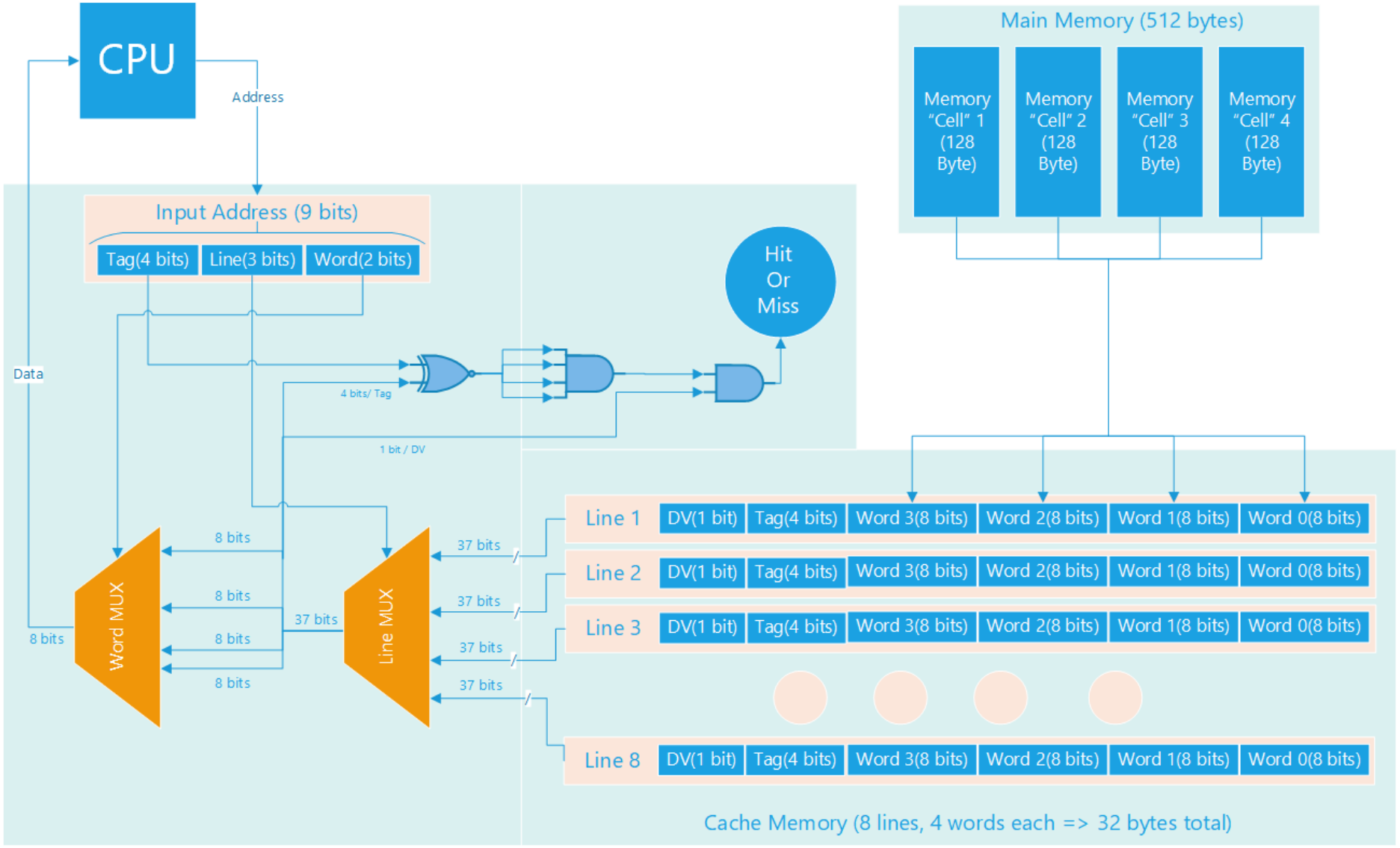
3 Cache Memory

Cache is used in between memory and the processor in-order to fetch instructions and data quickly to process.
A 32 byte cache that uses direct mapping was implemented. It was designed to work with a 512 location RAM which has 9 bits for the address. The cache contains 8 cache lines and 4 words for each line. Each cache line contains a Data valid bit (1 bit), Tag bits (4 bits), and data bits (4 x 8 = 32 bits) making a cache line 37 bit wide. Therefore the address is split into 3 segments containing the,

- Tag (4 bits)
- Cache Line (3 bits)
- Word Offset (2 bits)

This process is done by a combinational logic circuit as shown in the figure. All cache lines are given to the inputs of a multiplexer where the selection bits are the "cache line" portion of the address given, so the corresponding cache line is filtered and given out from the multiplexer according to the address given. Then the "Tag" portion of the selected cache line is compared with the "Tag" portion of the address using XNOR to check whether they match. If they match a bit combination of "1111" can be observed at the output of the XNOR gate which can be reduced to a single bit using a 4 input AND gate. Therefore if a logic 1 is observed at the output of the AND gate, it confirms that the correct word is in the cache. The correct word then can be extracted from the selected cache line using another multiplexer which takes the 4 date bytes of the cache line in, and one word out of them is selected using the "word offset" portion of the address.

A data-valid bit was included in the cache line to indicate whether a write operation is performed on the RAM. When valid bit is false, it will identify the cache request as a miss and update the corresponding cache line with the new data.


CPU
Address
Main Memory (512 bytes)
Memory "Cell" 1 (128 Byte)
Memory "Cell" 2 (128 Byte)
Memory "Cell" 3 (128 Byte)
Memory "Cell" 4 (128 Byte)
Input Address (9 bits)
Tag(4 bits)
Line(3 bits)
Word(2 bits)
Hit Or Miss
Data
4 bits/ Tag
1 bit / DV
8 bits
8 bits
8 bits
8 bits
Word MUX
37 bits
Line MUX
37 bits
37 bits
37 bits
37 bits
Line 1
Line 2
Line 3
Line 8
DV(1 bit)
Tag(4 bits)
Word 3(8 bits)
Word 2(8 bits)
Word 1(8 bits)
Word 0(8 bits)
Cache Memory (8 lines, 4 words each => 32 bytes total)

4 User Experience and Operation

The Equalizer is designed to make the operation as simple as possible. we took the maximum advantage of the built in interfaces of the DE2-115 board and made the controls very intuitive. To use the equalizer, user must give an audio input to the blue color line in port of the board and take an output from the green line out jack. Both ports accept standard 3.5mm AUX cables. There are three main controls

1. Reset button
2. Value switches
3. Set button

4.0.1 Reset Button

This directly connects to the reset function of the Wolfson audio CODEC itself. It is always a good idea to reset the audio controller before starting a session by pressing this button to clear out any garbage values in the buffers. If and when the equalizer gets stuck while using, the user can always press this button to resolve it. The reset sequence is near instantaneous and has negligible effect on the audio stream after resetting.

4.0.2 Value switches

These are the what actually defines the equalizer. There are 15 switches, 5 per band. These switches generate a 5 bit integer value to be given as the band gain, providing us with a precision of 2^5 steps. To equalize the audio stream, the user sets the gain of each band using these switches, while setting all switches of a band to zero, representing zero gain will mute the selected band.

4.0.3 Set Button

After setting the gain values from the switches, pressing this button will apply the values to the master filter pair. This button is what activates the coefficient generator we discussed earlier.

After performing the above steps, user can directly get the filtered audio through the line out jack. It can be either listened to using sound devices or visualized on a spectrum analyzer/oscilloscope.

5 Issues Faced

Implementing a graphic equalizer is no easy task without any prior practical knowledge on the technology. Therefore, we faced a number of issues during the implementation, such as,

1. Since our sample rate is very high, the filter needs a lot of coefficients to properly filter the signal. Using a large number of coefficients rapidly consume the resources of the FPGA, therefore number of coefficients must be limited.
2. The coefficient limit then in turn produces a new problem. Without the proper number of coefficients, both the passband and stopband ripples of the filters become very high, resulting in a 'leaky' equalizer.
3. Before proper optimization, compile times reached up to 40 minutes. This made debugging the equalizer a slow and difficult process.
4. First tests did not include the I2C config module. But the default route-input-directly-to-output nature led us to believe the equalizer implementation was actually working.
5. Clipping is very commonplace when testing, making it very hard to get proper readings at a volume higher than a certain threshold.
6. Overflowing of internal buffers cause errors in the output sometimes.
7. Faulty cables can give misleading results.

6 References

1. Altera DE2-115 User Guide
2. Wolfson WM873 datasheet
3. http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/audio/audio.html

7 Appendix: Codes

7.1 Graphic Equalizer

7.1.1 Main Code

```
module Graphic_EQ_V2 (
  CLOCK_50,
  reset,
  AUD_ADCDAT,

  AUD_BCLK,
  AUD_ADCLRCK,
  AUD_DACLRCK,

  AUD_XCK,
  AUD_DACDAT,
  I2C_SCLK,
  I2C_SDAT,

  set,
  bass_level,
  mid_level,
  treble_level
);

  input CLOCK_50, reset, AUD_ADCDAT,set;
  inout AUD_BCLK, AUD_ADCLRCK, AUD_DACLRCK,I2C_SCLK,I2C_SDAT;
  output AUD_XCK, AUD_DACDAT;

  input signed [4:0] bass_level,mid_level,treble_level;

  wire [31:0] l_channel_audio_in, r_channel_audio_in, l_channel_audio_out,
  ↪  r_channel_audio_out;
  wire read_ready, write_ready, read_enable, write_enable;
  wire read_clock;
  wire [9:0] bass_ctrl, low_mid_ctrl, mid_ctrl, treble_ctrl;

  assign read_enable = read_ready;
  assign write_enable = write_ready;

  Audio_Controller audioAudio_Controller(
    // Inputs
```

```
37      .CLOCK_50(CLOCK_50),
38      .reset(~reset),
39
40      .read_audio_in(1'b1),
41      .left_channel_audio_out(l_channel_audio_out),
42      .right_channel_audio_out(r_channel_audio_out),
43      .write_audio_out(1'b1),
44
45      .AUD_ADCDAT(AUD_ADCDAT),
46
47      // Bidirectionals
48      .AUD_BCLK(AUD_BCLK),
49      .AUD_ADCLRCK(AUD_ADCLRCK),
50      .AUD_DACLRCK(AUD_DACLRCK),
51
52      // Outputs
53      .left_channel_audio_in(l_channel_audio_in),
54      .right_channel_audio_in(r_channel_audio_in),
55      .audio_in_available(read_ready),
56
57      .audio_out_allowed(write_ready),
58
59      .AUD_XCK(AUD_XCK),
60      .AUD_DACDAT(AUD_DACDAT)
61    );
62
63
64    avconf conf(
65      //  Host Side
66              .CLOCK_50(CLOCK_50),
67              .reset(reset),
68              //  I2C Side
69              .I2C_SCLK(I2C_SCLK),
70              .I2C_SDAT(I2C_SDAT)
71    );
72
73  equalizer_ctrl eq_left(
74    .enter(set),
75
76    .bass_level(bass_level),
77    .mid_level(mid_level),
78    .treble_level(treble_level),
79
80    .d_in(l_channel_audio_in),
81    .d_out(l_channel_audio_out),
82    .clk(read_ready));
83
84  equalizer_ctrl eq_right(
85    .enter(set),
86
87    .bass_level(bass_level),
```

```
88      .mid_level(mid_level),
89      .treble_level(treble_level),
90
91      .d_in(r_channel_audio_in),
92      .d_out(r_channel_audio_out),
93      .clk(read_ready));
94
95  endmodule
```

7.1.2 Equalizer

```
module Graphic_EQ_V2 (
  CLOCK_50,
  reset,
  AUD_ADCDAT,

  AUD_BCLK,
  AUD_ADCLRCK,
  AUD_DACLRCK,

  AUD_XCK,
  AUD_DACDAT,
  I2C_SCLK,
  I2C_SDAT,

  set,
  bass_level,
  mid_level,
  treble_level
);

  input CLOCK_50, reset, AUD_ADCDAT,set;
  inout AUD_BCLK, AUD_ADCLRCK, AUD_DACLRCK,I2C_SCLK,I2C_SDAT;
  output AUD_XCK, AUD_DACDAT;

  input signed [4:0] bass_level,mid_level,treble_level;

  wire [31:0] l_channel_audio_in, r_channel_audio_in, l_channel_audio_out,
   ↪  r_channel_audio_out;
  wire read_ready, write_ready, read_enable, write_enable;
  wire read_clock;
  wire [9:0] bass_ctrl, low_mid_ctrl, mid_ctrl, treble_ctrl;

  assign read_enable = read_ready;
  assign write_enable = write_ready;

  Audio_Controller audioAudio_Controller(
    // Inputs
    .CLOCK_50(CLOCK_50),
    .reset(~reset),

    .read_audio_in(1'b1),
    .left_channel_audio_out(l_channel_audio_out),
    .right_channel_audio_out(r_channel_audio_out),
    .write_audio_out(1'b1),

    .AUD_ADCDAT(AUD_ADCDAT),

    // Bidirectionals
    .AUD_BCLK(AUD_BCLK),
    .AUD_ADCLRCK(AUD_ADCLRCK),
```

```
50      .AUD_DACLRCK(AUD_DACLRCK),
51
52      // Outputs
53      .left_channel_audio_in(l_channel_audio_in),
54      .right_channel_audio_in(r_channel_audio_in),
55      .audio_in_available(read_ready),
56
57      .audio_out_allowed(write_ready),
58
59      .AUD_XCK(AUD_XCK),
60      .AUD_DACDAT(AUD_DACDAT)
61    );
62
63
64    avconf conf(
65      //  Host Side
66              .CLOCK_50(CLOCK_50),
67              .reset(reset),
68              //  I2C Side
69              .I2C_SCLK(I2C_SCLK),
70              .I2C_SDAT(I2C_SDAT)
71    );
72
73  equalizer_ctrl eq_left(
74    .enter(set),
75
76    .bass_level(bass_level),
77    .mid_level(mid_level),
78    .treble_level(treble_level),
79
80    .d_in(l_channel_audio_in),
81    .d_out(l_channel_audio_out),
82    .clk(read_ready));
83
84  equalizer_ctrl eq_right(
85    .enter(set),
86
87    .bass_level(bass_level),
88    .mid_level(mid_level),
89    .treble_level(treble_level),
90
91    .d_in(r_channel_audio_in),
92    .d_out(r_channel_audio_out),
93    .clk(read_ready));
94
95  endmodule
```

7.1.3 Control Module

```
module equalizer_ctrl(
  enter,

  bass_level,
  mid_level,
  treble_level,

  d_in,
  d_out,

  clk
  );

input signed [32:1] d_in;
input enter, clk;

input signed [4:0] bass_level,mid_level,treble_level;

output reg signed[32:1] d_out;

reg signed [17:0] coef_bass [25:0] = '{
2,
2,
3,
3,
4,
5,
6,
8,
10,
11,
14,
16,
18,
21,
23,
26,
28,
31,
33,
35,
37,
39,
40,
41,
42,
42
};

reg signed [17:0] coef_mid [25:0] = '{
```

```
-1,
-1,
-2,
-3,
-4,
-6,
-9,
-11,
-14,
-16,
-18,
-19,
-19,
-17,
-13,
-7,
1,
11,
22,
34,
47,
58,
69,
78,
84,
87
};

reg signed [17:0] coef_treb [25:0] = '{
-1,
-1,
1,
2,
0,
0,
4,
8,
6,
0,
4,
15,
11,
-9,
-15,
4,
7,
-33,
-67,
-40,
0,
-49,
```

```
-155,
-134,
97,
344
};

reg signed [17:0] coef_final [25:0];
reg signed [31:0] shift_reg [50:0];
integer i;
integer j;

always @(negedge enter)begin

  for (i = 25; i > -1; i = i - 1)
    coef_final[i] <= (coef_bass[i]*bass_level + coef_mid[i]*mid_level +
    ↪  coef_treb[i]*treble_level)/3;

end

always @ (posedge clk)
  begin

    shift_reg[0] <= d_in/128;


    for (j = 50; j > 0; j = j - 1)
      shift_reg[j] <= shift_reg[j-1];

end

always @(negedge clk)
  begin
      d_out <= (coef_final[25] * shift_reg[25]
      + coef_final[24] * (shift_reg[26] + shift_reg[24])
      + coef_final[23] * (shift_reg[27] + shift_reg[23])
      + coef_final[22] * (shift_reg[28] + shift_reg[22])
      + coef_final[21] * (shift_reg[29] + shift_reg[21])
      + coef_final[20] * (shift_reg[30] + shift_reg[20])
      + coef_final[19] * (shift_reg[31] + shift_reg[19])
      + coef_final[18] * (shift_reg[32] + shift_reg[18])
      + coef_final[17] * (shift_reg[33] + shift_reg[17])
      + coef_final[16] * (shift_reg[34] + shift_reg[16])
      + coef_final[15] * (shift_reg[35] + shift_reg[15])
      + coef_final[14] * (shift_reg[36] + shift_reg[14])
      + coef_final[13] * (shift_reg[37] + shift_reg[13])
      + coef_final[12] * (shift_reg[38] + shift_reg[12])
      + coef_final[11] * (shift_reg[39] + shift_reg[11])
      + coef_final[10] * (shift_reg[40] + shift_reg[10])
      + coef_final[9] * (shift_reg[41] + shift_reg[9])
      + coef_final[8] * (shift_reg[42] + shift_reg[8])
      + coef_final[7] * (shift_reg[43] + shift_reg[7])
```

```
152         + coef_final[6] * (shift_reg[44] + shift_reg[6])
153         + coef_final[5] * (shift_reg[45] + shift_reg[5])
154         + coef_final[4] * (shift_reg[46] + shift_reg[4])
155         + coef_final[3] * (shift_reg[47] + shift_reg[3])
156         + coef_final[2] * (shift_reg[48] + shift_reg[2])
157         + coef_final[1] * (shift_reg[49] + shift_reg[1])
158         + coef_final[0] * (shift_reg[50] + shift_reg[0]));
159 
160     end
161 
162 endmodule
```

7.2 Cache Memory

7.2.1 Top Level

```
module top_level_cache(clk,p_address,wen,hex0,hex1,hex2,dv);

input clk,wen;
input [8:0] p_address;
output dv;
output [6:0] hex0;
output [6:0] hex1;
output [6:0] hex2;
//output [6:0] hex5;
//output [6:0] hex6;
//output [6:0] hex7;
reg [31:0] din=32'd117893636;
wire [31:0] dout;
wire [7:0] cache_to_processor;
wire [6:0] mem_address;
wire _10MHz;
wire wen_debounced;

debouncer   manual_clk_button(
      .button_in(wen),
      .button_out(wen_debounced),
      .clk(clk));           //give a fast clock

pll _50MHz_to_10MHz(
  .inclk0(clk),
  .c0(_10MHz));

RAM ram(
  .address(mem_address),
  .clock(~_10MHz),
  .data(din),
  .wren(~wen_debounced),
  .q(dout));

bi2bcd bi1(
  .din(cache_to_processor),
  .dout2(hex2),
  .dout1(hex1),
  .dout0(hex0));

//bi2bcd bi2(
//  .din(dout[15:8]),
//  .dout2(hex7),
//  .dout1(hex6),
//  .dout0(hex5));

cache_controller cache_Ctr(
```

```
48    .wren(~wen_debounced),
49    .clock(_10MHz),
50    .p_address(p_address),
51    .mem_address(mem_address),
52    .din(dout),
53    .dout(cache_to_processor),
54    .DV(dv));
55
56  //always @ (negedge _10MHz)
57  //  begin
58  //    p_address_reg<=p_address;
59  //  end
60
61  endmodule
```

7.2.2 Controller

```
module cache_controller(wren,clock,p_address,mem_address,din,dout,DV);

input [8:0] p_address;
input [31:0] din;
input clock,wren;
output [7:0] dout;
output [6:0] mem_address;

output DV;
wire [36:0]selected_line;
reg [1:0] STATE=2'd0;


reg [36:0] cache [7:0];

initial
  begin
    cache[0]=37'd0;
    cache[1]=37'd0;
    cache[2]=37'd0;
    cache[3]=37'd0;
    cache[4]=37'd0;
    cache[5]=37'd0;
    cache[6]=37'd0;
    cache[7]=37'd0;
  end

line_multiplexer line_multiplexer(
  .data0x(cache[0][36:0]),
  .data1x(cache[1][36:0]),
  .data2x(cache[2][36:0]),
  .data3x(cache[3][36:0]),
  .data4x(cache[4][36:0]),
  .data5x(cache[5][36:0]),
  .data6x(cache[6][36:0]),
  .data7x(cache[7][36:0]),
  .sel(p_address[4:2]),
  .result(selected_line));

word_multiplexer word_multiplexer(
  .data3x(selected_line[31:24]),
  .data2x(selected_line[23:16]),
  .data1x(selected_line[15:8]),
  .data0x(selected_line[7:0]),
  .sel(p_address[1:0]),
  .result(dout));

assign DV = (selected_line[36]&(&(p_address[8:5] ~^ selected_line[35:32])));
assign mem_address=p_address[8:2];

```

```
51  always @(posedge clock)
52    begin
53      if (wren==1)
54        begin
55          cache[p_address[4:2]][36]<=0;
56        end
57      else if (DV==0)
58        begin
59          case(STATE)
60            2'd0:STATE<=2'd1;
61            2'd1:STATE<=2'd2;
62            2'd2:
63              begin
64                cache[p_address[4:2]]<={1'b1,p_address[8:5],din};
65                STATE<=2'd0;
66              end
67            default:STATE<=2'd0;
68          endcase
69        end
70    end
71  endmodule
```

7.2.3 Line Multiplexer

```
module line_multiplexer(
  data0x,
  data1x,
  data2x,
  data3x,
  data4x,
  data5x,
  data6x,
  data7x,
  sel,
  result);

input [36:0] data0x;
input [36:0] data1x;
input [36:0] data2x;
input [36:0] data3x;
input [36:0] data4x;
input [36:0] data5x;
input [36:0] data6x;
input [36:0] data7x;
input [2:0]  sel;
output reg [36:0] result=37'd0;


parameter d0=3'd0;
parameter d1=3'd1;
parameter d2=3'd2;
parameter d3=3'd3;
parameter d4=3'd4;
parameter d5=3'd5;
parameter d6=3'd6;
parameter d7=3'd7;

initial
  begin
    case(sel)
      d0: result <= data0x;
      d1: result <= data1x;
      d2: result <= data2x;
      d3: result <= data3x;
      d4: result <= data4x;
      d5: result <= data5x;
      d6: result <= data6x;
      d7: result <= data7x;
      default: result<=37'd0;
    endcase
  end


always @(*)
```

```
51    begin
52      case(sel)
53        d0: result <= data0x;
54        d1: result <= data1x;
55        d2: result <= data2x;
56        d3: result <= data3x;
57        d4: result <= data4x;
58        d5: result <= data5x;
59        d6: result <= data6x;
60        d7: result <= data7x;
61        default: result<=37'd0;
62      endcase
63    end
64
65  endmodule
```

7.2.4 Word Multiplexer

```
module word_multiplexer(
  data0x,
  data1x,
  data2x,
  data3x,
  sel,
  result);

input [7:0] data0x;
input [7:0] data1x;
input [7:0] data2x;
input [7:0] data3x;
input [1:0]  sel;
output reg [7:0] result=8'd0;

parameter d0=3'd0;
parameter d1=3'd1;
parameter d2=3'd2;
parameter d3=3'd3;

initial
  begin
    case(sel)
      d0: result <= data0x;
      d1: result <= data1x;
      d2: result <= data2x;
      d3: result <= data3x;
      default: result<=8'd0;
    endcase
  end



always @(*)
  begin
    case(sel)
      d0: result <= data0x;
      d1: result <= data1x;
      d2: result <= data2x;
      d3: result <= data3x;
      default: result<=8'd0;
    endcase
  end

endmodule
```

7.2.5 RAM

```
// megafunction wizard: %RAM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: RAM.v
// Megafunction Name(s):
//       altsyncram
//
// Simulation Library Files(s):
//       altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 15.0.0 Build 145 04/22/2015 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2015 Altera Corporation. All rights reserved.
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, the Altera Quartus II License Agreement,
//the Altera MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Altera and sold by Altera or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module RAM (
  address,
  clock,
  data,
  wren,
  q);

  input  [6:0]  address;
  input    clock;
  input  [31:0]  data;
  input    wren;
```

```
51    output  [31:0]  q;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_off
54  `endif
55    tri1    clock;
56  `ifndef ALTERA_RESERVED_QIS
57  // synopsys translate_on
58  `endif
59
60    wire [31:0] sub_wire0;
61    wire [31:0] q = sub_wire0[31:0];
62
63    altsyncram  altsyncram_component (
64          .address_a (address),
65          .clock0 (clock),
66          .data_a (data),
67          .wren_a (wren),
68          .q_a (sub_wire0),
69          .aclr0 (1'b0),
70          .aclr1 (1'b0),
71          .address_b (1'b1),
72          .addressstall_a (1'b0),
73          .addressstall_b (1'b0),
74          .byteena_a (1'b1),
75          .byteena_b (1'b1),
76          .clock1 (1'b1),
77          .clocken0 (1'b1),
78          .clocken1 (1'b1),
79          .clocken2 (1'b1),
80          .clocken3 (1'b1),
81          .data_b (1'b1),
82          .eccstatus (),
83          .q_b (),
84          .rden_a (1'b1),
85          .rden_b (1'b1),
86          .wren_b (1'b0));
87    defparam
88      altsyncram_component.clock_enable_input_a = "BYPASS",
89      altsyncram_component.clock_enable_output_a = "BYPASS",
90      altsyncram_component.init_file = "memory.mif",
91      altsyncram_component.intended_device_family = "Cyclone IV E",
92      altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
93      altsyncram_component.lpm_type = "altsyncram",
94      altsyncram_component.numwords_a = 128,
95      altsyncram_component.operation_mode = "SINGLE_PORT",
96      altsyncram_component.outdata_aclr_a = "NONE",
97      altsyncram_component.outdata_reg_a = "CLOCK0",
98      altsyncram_component.power_up_uninitialized = "FALSE",
99      altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
100     altsyncram_component.widthad_a = 7,
101     altsyncram_component.width_a = 32,
```

```
102     altsyncram_component.width_byteena_a = 1;
103
104
105 endmodule
106
107 // ============================================================
108 // CNX file retrieval info
109 // ============================================================
110 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
111 // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
112 // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
113 // Retrieval info: PRIVATE: AclrData NUMERIC "0"
114 // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
115 // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
116 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
117 // Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
118 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
119 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
120 // Retrieval info: PRIVATE: Clken NUMERIC "0"
121 // Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
122 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
123 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
124 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
125 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
126 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
127 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
128 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
129 // Retrieval info: PRIVATE: MIFfilename STRING "memory.mif"
130 // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "128"
131 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
132 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
133 // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
134 // Retrieval info: PRIVATE: RegData NUMERIC "1"
135 // Retrieval info: PRIVATE: RegOutput NUMERIC "1"
136 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
137 // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
138 // Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
139 // Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
140 // Retrieval info: PRIVATE: WidthAddr NUMERIC "7"
141 // Retrieval info: PRIVATE: WidthData NUMERIC "32"
142 // Retrieval info: PRIVATE: rden NUMERIC "0"
143 // Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
144 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
145 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
146 // Retrieval info: CONSTANT: INIT_FILE STRING "memory.mif"
147 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
148 // Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
149 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
150 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "128"
151 // Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
152 // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
```

```
153 // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "CLOCK0"
154 // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
155 // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING
    ↪  "NEW_DATA_NO_NBE_READ"
156 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "7"
157 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
158 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
159 // Retrieval info: USED_PORT: address 0 0 7 0 INPUT NODEFVAL "address[6..0]"
160 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
161 // Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL "data[31..0]"
162 // Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q[31..0]"
163 // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
164 // Retrieval info: CONNECT: @address_a 0 0 7 0 address 0 0 7 0
165 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
166 // Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
167 // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
168 // Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
169 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM.v TRUE
170 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM.inc FALSE
171 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM.cmp FALSE
172 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM.bsf FALSE
173 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM_inst.v FALSE
174 // Retrieval info: GEN_FILE: TYPE_NORMAL RAM_bb.v TRUE
175 // Retrieval info: LIB_FILE: altera_mf
```

7.2.6 Decoder

```
module decoder(din,dout);

input [3:0] din;
output reg [6:0] dout;



always @(din)
case(din)
  4'd0:dout<=7'b1000000;
  4'd1:dout<=7'b1111001;
  4'd2:dout<=7'b0100100;
  4'd3:dout<=7'b0110000;
  4'd4:dout<=7'b0011001;
  4'd5:dout<=7'b0010010;
  4'd6:dout<=7'b0000010;
  4'd7:dout<=7'b1111000;
  4'd8:dout<=7'b0000000;
  4'd9:dout<=7'b0011000;
endcase

endmodule
```