Programmable Music Visualizer

Dany Qumsiyeh Mike Spindel

December 14, 2005

Abstract

This project is a programmable music visualizer capable of combining many simple effects to produce complex graphics based on audio input. It is written in Verilog and is composed of many modules grouped into four major system blocks: Audio Preprocessor, CPU, Video Memory, and Effects. The Audio Preprocessor block maintains a 1024 sample audio sample FIFO buffer and presents access ports for Effects to read data. The CPU configures the parameters of the Effects modules in order to composite different effects and produce a variety of visualizations based on time and user input. The Video Memory block interfaces with two frame buffers stored on ZBT SRAMs. The Effects module accesses Video Memory and steps through the video buffer generating new frames based on the parameters from the CPU. The CPU memory can be programmed on-the-fly through the JTAG interface with code assembled by BSim.

Contents

1 Overview **1**

2 Description **2**

2.1 Audio Preprocessing Stage (by Mike Spindel) 2

2.1.1 Audio Acquisition: `audio` 2

2.1.2 Sample Storage: `replram` and `buf_replram` 2

2.2 CPU (by Mike Spindel) 3

2.2.1 Overview 3

2.2.2 Changes from the Orignal Beta2 Design 3

2.3 Effects Generation (by Dany Qumsiyeh) 5

2.4 Video Memory (by Dany Qumsiyeh) 6

3 Testing and Debugging **8**

4 Conclusion **8**

5 Appendix **9**

5.1 Mapping between CPU registers and Effects Parameters 9

5.2 Verilog 11

5.2.1 Audio Preprocessor 17

5.2.2 CPU 23

5.2.3 Effects 28

5.2.4 Video Memory 35

List of Figures

1 High level block diagram. 1
2 The audio preprocessing stage. 2
3 The modified Beta2 used by the visualizer. 4
4 The effects generation hardware. 6
5 The display memory interface. 7

List of Tables

1 Summary of new opcodes. 5
2 Input Registers. 5

1 Overview

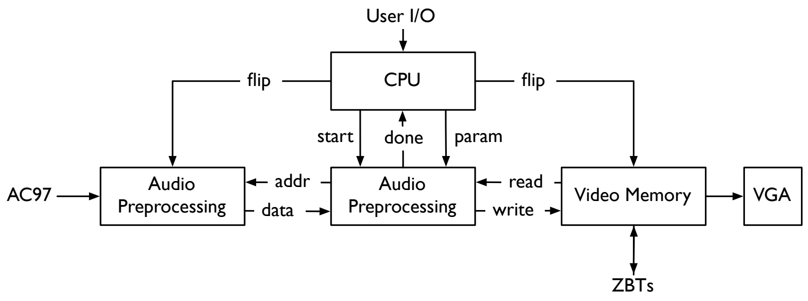

Figure 1: High level block diagram.

The music visualizer runs on an FPGA with an AC-97 A/D converter, ZBT SRAMs, and VGA output. It takes in audio data from the AC-97, and outputs VGA graphics at 640x480 pixels, at 60Hz. It is controlled by a CPU which can be programmed at runtime through a JTAG interface.

The system is composed of several major components. An audio preprocessor, a CPU, an effects generator, and a display module. Sound information flows from the AC97 codec into the audio preprocessor where it is stored in a replicated BRAM. The CPU triggers the effects generator according to a program stored in its own instruction memory by pulsing a start signal. The effects FSM then iterates through pixel coordinates, transforms them, and feeds them to a set of graphics generators. The graphics generators then access the buffered sound information and output a color. A series of blending operators are used to determine the a net color, which is then written to a frame buffer stored in SRAM and managed by the display block. When the program running on the CPU has finished rendering a single frame, it issues an instruction that causes the display block to flip buffers: the buffer that was being displayed is now being used to render new frames.

The effects that this framework is capable of can be defined as an arbitrary waveform and transformation. To generate each subsequent frame, the previous frame can be transformed by a motion (such as rotation), and a color operation (such as color cycling). Then, a number of waveforms based on audio data can be overlayed and combined with the previous frame. This is achieved through dedicated effects hardware, which generates a new frame based on a large number of configurable parameters. The output of many waveform generators can be combined with the previous frame and the current frame, and the coordinates supplied to these generators can be transformed to create motion effects. This allows for

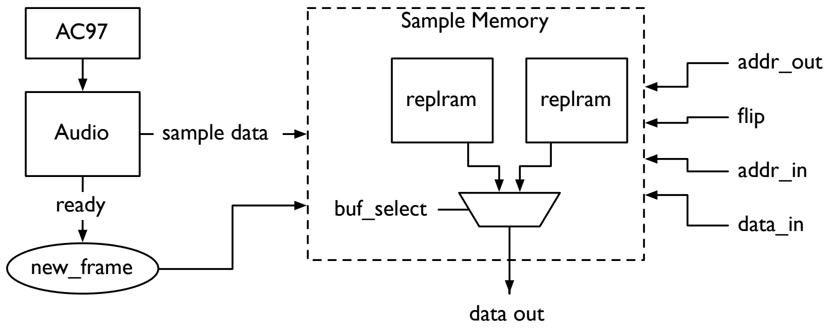

Figure 2: The audio preprocessing stage.

many complex effects to be created through the combination of simple modules.

2 Description

2.1 Audio Preprocessing Stage (by Mike Spindel)

2.1.1 Audio Acquisition: `audio`

The audio stage provides a simple stereo interface to the microphone input on the labkit. It takes the system clock, AC97 bit clock, and AC97 data bus as input and outputs a 20 bit signal for the left and right audio channels and a ready signal that is high when the audio signals contain valid data. It also outputs the AC97 sync signal and the *audio_reset_b* signal.

The *audio_reset_b* enables the AC97 codec. It is set high several clock cycles after the power-on reset by an FSM. The audio data and AC97 control signals, including the sync, are generated by two black box modules: `ac97` and `ac97commands`. The `ac97` module contains an FSM that deals with sending and receiving AC97 frames and the `ac97commands` module encapsulates the logic necessary to generate AC97 commands.

2.1.2 Sample Storage: `replram` and `buf_replram`

The effects generators access audio sample data through the sample memory. Because there can be a number of different generators, the sample memory needs to have a single read port and multiple write ports. It looks like a FIFO on `audio` side and a ROM on the generator side. However, if the data is updated while a frame is being drawn, then the drawn waveform will be distorted. In order to prevent this from happening, the sample memory is double buffered: one memory is written to while another is being read from. The two switch roles every time the visualizer starts rendering a new frame.

This system is implemented through two layers of abstraction. At the bottom is the `replram` module. It provides a clean interface to a replicated dual-port BRAM. It takes as input two clocks, *rst*, *addr_in*, *d_in*, *addr_out*, and *we*. Its only output is the read output *d_out*. The address in line is a single signal sent to all of the constituent BRAMs, however the address out and data out lines are a concatenation of as many input signals as there are read ports.

On top of this is the `buf_replram` module that handles all of the buffering and switching logic. It takes as input *clk*, *rst*, *d_in*, *addr_out*, *we*, *select*, and *start*. It outputs *d_out*. The first four of these signals play the same role that they do in `replram`. Buffer switches are synchronized against changes in the *select* signal. In practice, it is wired to the *flip* signal output by the CPU. Internally, the module works by switching the read/write buffers at the first change of *select* after having filled up the write buffer. Input addresses are generated by the same counter used to determine if the buffer is full. The *start* signal must be high in order to start filling the new write buffer after a buffer switch. Although it isn't necessary for audio sample data, it is necessary when the data needs to be properly aligned in the buffer (e.g. FFT data).

These modules are almost completely parameterized, however it's still necessary to have different modules for each size of BRAM that's used (e.g. replram10x64 and replram10x1024) because verilog is not particularly adept at code generation.

2.2 CPU (by Mike Spindel)

2.2.1 Overview

The CPU is used to configure the effects generators. In order to have a maximum flexibility in designing visualizations, it is necessary to use a reasonably full featured processor here. Towards that end, we use a modified Beta2 core. Since the Beta platform is very well documented elsewhere, this section will focus on the modifications made for use in the visualizer.

2.2.2 Changes from the Orignal Beta2 Design

Since the music visualizer does not need an OS and does not handle interrupted, all of the wiring related to this has been removed from our Beta. This means that there is no distinction between "supervisor" mode and other code. The IRQ lines

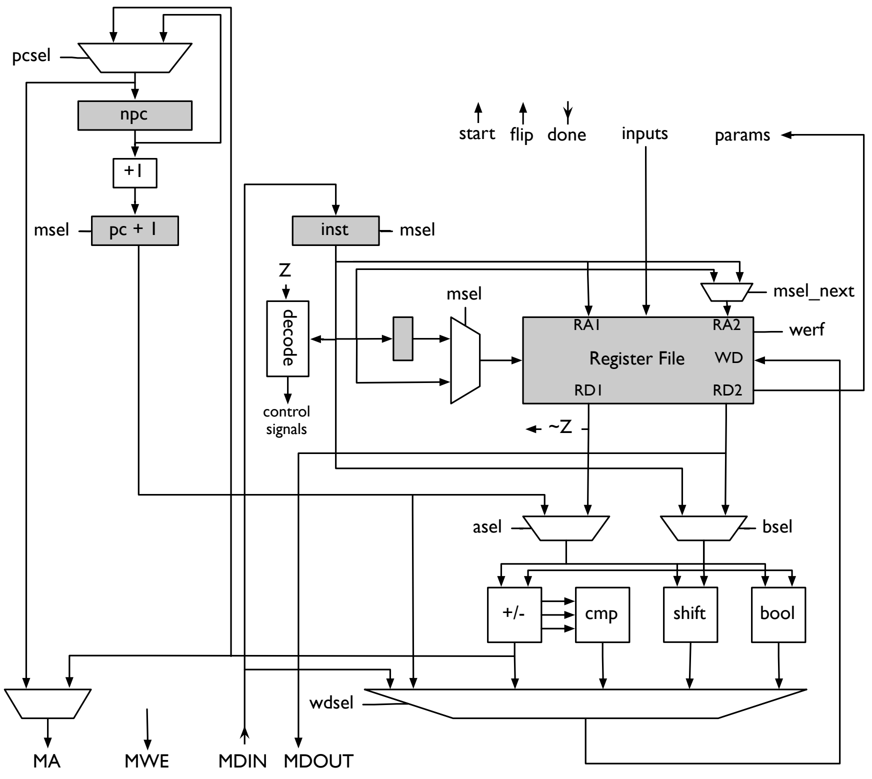

Figure 3: The modified Beta2 used by the visualizer.

have been removed. Illops and traps do not cause the beta to jump to a particular memory location, rather they are ignored and execution continues. This leads to the removal of several wires from the control logic and the pcsel mux and the complete removal of the wasel mux.

A single dual port BRAM module is used for the instruction memory. Since it is just as easy to synthesize a 32 bit wide memory as an 8 bit wide memory in this case, our Beta has been changed to use a word-based memory scheme rather than a byte-based scheme. This leads to the removal of the csel mux and some fairly minor changes to the control logic and PC increment scheme.

Since the CPU needs to be able to interface with the effects FSM directly, it needs to deal with an additional three signals: *start*, *done*, *flip*. The *start* signal tells the FSM to start generating a new frame into memory. When it's done, it sets theh *done* signal high. The *flip* signal determines which display buffer is being displayed and which is being written into. This requirement is met by adding two

Table 1: Summary of new opcodes.

Name	Opcode	Action
GEN	010100	Put *start* high and stall IF until *done*.
FLIP	010101	Toggle value of *flip* output.

new opcodes: GEN (6'b010100) and FLIP (6'b010101). These are summarized in table 1. The GEN command will put the start signal high for one cycle and will immediately stall the instruction fetch stage by putting *msel* high. As soon as *done* goes high, *msel* is taken low and the instruction fetch resumes its normal operation. These transitions are performed combinationally so as not to waste cycles needlessly. Since there will be a cycle in which *msel* and the opcode is still GEN right after *done* is high, *done* is registered and either the initial done or the registered done is enough to keep *msel* low. The FLIP opcode sets the control logic line *done_prg* high which simply causes output *flip* signal to toggle.

Table 2: Input Registers.

Register	Value
r28	[button0-3, button_right,left,down,up, switch7-0]
r29	user1
r30	user2

The CPU also needs to be able to set parameters for the different effects generators. This is implemented by reserving registers 16 through 27 for use as parameters. These registers are accessed by the CPU in the same way as registers 0 through 15, however they are also wired directly to the effects module for use as parameters. A summary of which registers map to which effects is in the appendix. Also, in order to facilitate some degree of input from the outside world, registers 28 through 30 are mapped to external inputs that can be sampled by the running program. This is summarized in table 2.

2.3 Effects Generation (by Dany Qumsiyeh)

The effects modules step through all pixels of the frame, and apply configurable transformations to them based on parameters from the CPU. The pixel_fsm module steps through the coordinates. These coordinates are connected to the write address for the newframe, with an appropriate delay to match the pipelining of the color computation. The coordinates pass through motion modules, which trans-

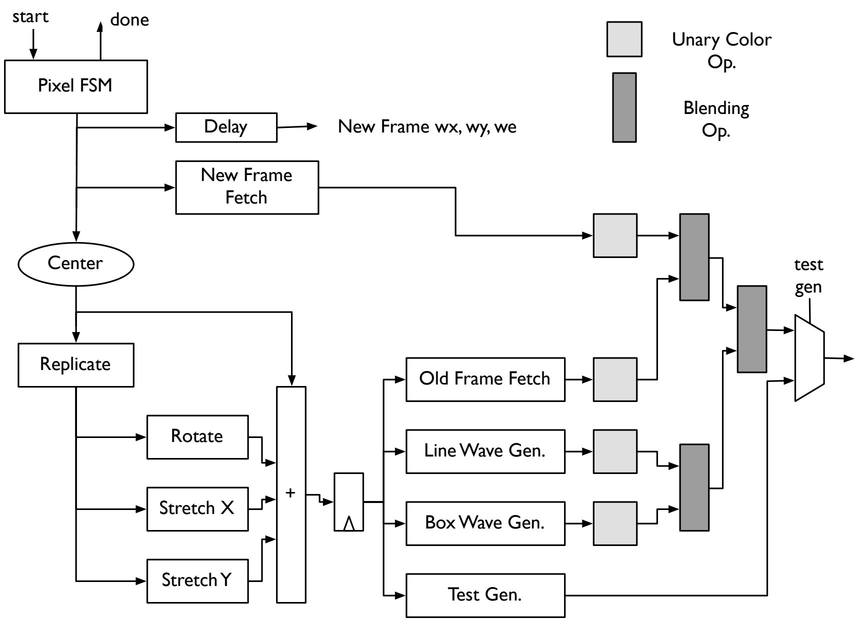

Figure 4: The effects generation hardware.

form them to produce the effects of rotation or scaling of X and Y. The vectors from the motion modules are added together to produce new coordinates, used for the oldframe_fetch and generator modules. Generator modules output a color based on audio data, and these colors are combined through configurable color and blend operator modules to produce the final color written to memory.

Other features include a replicate module, which can do a modulo operation on the coordinates supplied to the motion modules, effectively duplicating the motion across the screen. A translation offset can be added to the motion vectors allowing generators to be relocated on screen, and for effects to drift offscreen. Unary color operators allow generators to be given any color, and previous frames to be modified through operations such as color rotation. The parameters for all of these modules are outlined in the appendix.

2.4 Video Memory (by Dany Qumsiyeh)

The video memory modules retain a video buffer in each of two ZBT SRAMs. In this implimentation, the buffer size is 640x480 pixels at 18 bits per pixel. When generating each frame, one buffer holds the previous frame, and new information is written to the other buffer. The buffers switch places based on a flip signal, which

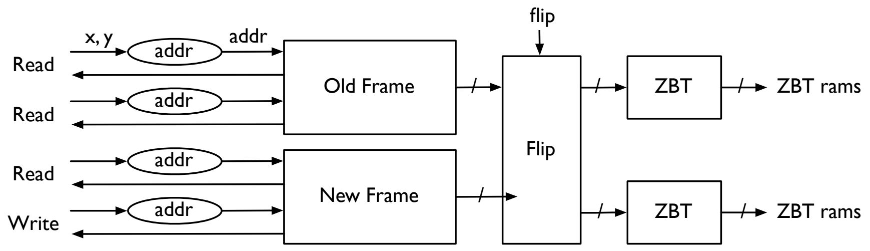

Figure 5: The display memory interface.

is changed when processing for a frame is done. Two read ports are presented for the old frame: one for the VGA module to actually display pixels, and one for the Effects modules. A read and write port are presented for the new frame, so that Effects modules can composite effects over the new frame and write the new data to memory.

The newframe_memory module interlaces sequential reads and writes to present two simultaneous ports. As pixels are 18 bits and the ZBTs are 36 bits wide, two adjacent pixels can be written and read in one clock cycle. The newframe_memory module alternates reads and writes to the ZBTs, buffering the bits to allow one pixel to be read and written each clock cycle.

The oldframe_memory module allows the two read ports to make requests on alternating cycles. This allows Effects modules to make non-sequential requests, and the VGA output to receive pixels at 25MHz.

3 Testing and Debugging

The audio preprocessing stage was written in several stages, which made testing fairly straightforward. The code necessary for the AC97 codec to operate was written first. Its operation was verified using the ChipScope logic analyzer. Chipscope's ease of use and ability to easily deal with wide signals made it an exceptionally valueable debugging tool throughout the entire design process. The replicated memory modules were written next and were also verified using Chipscope. A simple FSM was used to provide input for the purposes of testing.

The changes made to the beta were tested in a similar manner. Simple test programs were used to provide input while Chipscope was used to verify that the internal signals and outputs behaved according to spec. Although this method was largely effective, more thorough testing programs would have been extremely useful in verifying that changes to one part of the CPU didn't have unexpected consequences to the rest of its functionality. The lack of forma verification caused trouble a few times during development.

The modularity of the effects made incremental testing very easy. The Effects module began as a simple generator, requesting addresses from audio memory, and outputting colors from the data. Each of the modules was added and tested one at a time, with actual video output, leaving the Effects module functional at every stage.

ZBT interfacing provided difficulty, but was mostly solved with the logic analyzer and small test programs that wrote sequential numbers. The ZBT was finally made reliable by making some assignments on the falling edge of the clock, and extending the pipeline. Test code for that interface was reliable up to 60 or 70 MHz. The proposal described running the ZBTs faster than the global clock to present the simultaneous interfaces, but after this testing the plan was modified to allow everything to run at 50 MHz, with the access interlacing described above.

4 Conclusion

This project has been characterized by continuous change: changing goals, changing responsibilities, and changing designs. As we moved from the design stage to the implementation stage, it became clear that, in the initial design, we had unnecessarily complicated parts of the design. The effects generation scheme notably went through several major changes in architecture. For example, where we had initially envisioned the memory fetch as being part of a blending operation, it became clear that they would be running in parallel with the other effects generators. This removed the need for a monolithic "blend" module, which was replaced by a number of blending operators. In turn this inspired the addition of unary operations on generator outputs, which was never part of the original design.

A number of our ambitions also had to be scaled back due to time constraints. The convolution operator that was initially going to be part of the effects generator was lost. The FFT and frequency based statistic beat detection was dropped from the preprocessing stage and there simply wasn't enough time to write enough complicated programs that took full advantage of the CPU.

Some of the most important lessons learned from this project involved building a stronger intuition when it comes to debugging and designing large systems. This sort of intuition stems from a better understanding of common pitfalls and effective processes to avoid them. It also showed that spending more time considering the initial design will lead to more well thought out system in the end. But, it also showed that it is important to remain flexible throughout the development process in order to take advantage of the increased understanding of the problem gained by actually implementing a solution.

5 Appendix

5.1 Mapping between CPU registers and Effects Parameters

```
reg16: motion modules
   reg16[5:0] rotate
   reg16[11:6] stretch x
   reg16[17:12] stretch y

for the 6 bits of each motion module:
   0  enable
   1  invert
   [5:2] magnitude bitshift


reg17[9:0] translate x (number)
reg17[19:10] translate y (number)
reg17[25:20] replicate x
reg17[31:26] replicate y

for the 6 bits of each replicate dimension:
   0 enable
   [5:2] magnitude bitshift (size of each replication)
```

```
reg18: generators
   reg18[7:0] line waveform
     0     vertical?
     [4:1] magnitude bitshift
     [6:5] thickness (powers of 2)
   reg18[23:8] box waveform
     [3:0] magnitude bitshift
     [5:4] thickness (powers of 2)
     [15:6] size (width/2)


unary color operators:
reg19: newframe fetch color operator
reg20: oldframe fetch color operator
reg21: line waveform color op.
reg22: box waveform color op.

for the 32 bit unary color operators:
   [3:0] operation
   [21:4] color constant

   operations:
   0000 no op
   0001 bitwise NOT
   0010 bitwise AND with color
   0011 bitwise OR with color
   0100 shift right on each component
   0101 shift to blue
   0110 shift to red
   0111 cycle colors R-G-B
   1000 cycle colors B-G-R


reg23: blend operations
   reg23[3:0] newframe OP oldframe => color1
   reg23[7:4] line waveform OP box waveform => color2
   reg23[11:8] color1 OP color2 => output

for the 4 bit binary color operators:
```

```
0000 pick A
0001 pick B
0010 A over B
0011 B over A
0100 bitwise OR
```

5.2 Verilog

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

module labkit(beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
     ac97_bit_clock,

     vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
     vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
     vga_out_vsync,

     tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
     tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
     tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

     tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
     tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
     tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
     tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

     ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
     ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

     ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
     ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

     clock_feedback_out, clock_feedback_in,

     flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
     flash_reset_b, flash_sts, flash_byte_b,

     rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

     mouse_clock, mouse_data, keyboard_clock, keyboard_data,

     clock_27mhz, clock1, clock2,

     disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
     disp_reset_b, disp_data_in,

     button0, button1, button2, button3, button_enter, button_right,
     button_left, button_down, button_up,

     switch,

     led,

     user1, user2, user3, user4,

     daughtercard,

     systemace_data, systemace_address, systemace_ce_b,
     systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

     analyzer1_data, analyzer1_clock,
       analyzer2_data, analyzer2_clock,
       analyzer3_data, analyzer3_clock,
       analyzer4_data, analyzer4_clock);

 output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
 input  ac97_bit_clock, ac97_sdata_in;

 output [7:0] vga_out_red, vga_out_green, vga_out_blue;
 output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
vga_out_hsync, vga_out_vsync;

 output [9:0] tv_out_ycrcb;
 output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
tv_out_subcar_reset;

 input  [19:0] tv_in_ycrcb;
 input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
tv_in_hff, tv_in_aff;
 output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
tv_in_reset_b, tv_in_clock;
 inout  tv_in_i2c_data;

 inout  [35:0] ram0_data;
 output [18:0] ram0_address;
 output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
```

```
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
  button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
 analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

   ////////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////////////////

   // Audio Input and Output
   assign beep= 1'b0;
   //assign audio_reset_b = 1'b0;
   //assign ac97_synch = 1'b0;
   //assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
```

```
assign tv_in_i2c_data = 1’bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
/*
assign ram0_data = 36’hZ;
assign ram0_address = 19’h0;
assign ram0_adv_ld = 1’b0;
assign ram0_clk = 1’b0;
assign ram0_cen_b = 1’b1;
assign ram0_ce_b = 1’b1;
assign ram0_oe_b = 1’b1;
assign ram0_we_b = 1’b1;
assign ram0_bwe_b = 4’hF;
assign ram1_data = 36’hZ;
assign ram1_address = 19’h0;
assign ram1_adv_ld = 1’b0;
assign ram1_clk = 1’b0;
assign ram1_cen_b = 1’b1;
assign ram1_ce_b = 1’b1;
assign ram1_oe_b = 1’b1;
assign ram1_we_b = 1’b1;
assign ram1_bwe_b = 4’hF;
assign clock_feedback_out = 1’b0;
*/
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16’hZ;
assign flash_address = 24’h0;
assign flash_ce_b = 1’b1;
assign flash_oe_b = 1’b1;
assign flash_we_b = 1’b1;
assign flash_reset_b = 1’b0;
assign flash_byte_b = 1’b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1’b1;
assign rs232_rts = 1’b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
assign disp_blank = 1’b1;
assign disp_clock = 1’b0;
assign disp_rs = 1’b0;
assign disp_ce_b = 1’b1;
assign disp_reset_b = 1’b0;
assign disp_data_out = 1’b0;
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8’hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32’hZ;
assign user2 = 32’hZ;
assign user3 = 32’hZ;
assign user4 = 32’hZ;

// Daughtercard Connectors
assign daughtercard = 44’hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16’hZ;
assign systemace_address = 7’h0;
assign systemace_ce_b = 1’b1;
assign systemace_we_b = 1’b1;
assign systemace_oe_b = 1’b1;
// systemace_irq and systemace_mpbrdy are inputs
```

```
   /*
   // Logic Analyzer
   assign analyzer1_data = 16’h0;
   assign analyzer1_clock = 1’b1;
   assign analyzer2_data = 16’h0;
   assign analyzer2_clock = 1’b1;
   assign analyzer3_data = 16’h0;
   assign analyzer3_clock = 1’b1;
   assign analyzer4_data = 16’h0;
   assign analyzer4_clock = 1’b1;
   */





   ////////////////////////////////////////////////////////////////////////////
   // GENERATE BASIC SIGNALS
   ////////////////////////////////////////////////////////////////////////////

   wire clk,clk_unbuf;  //create 50 MHz clock (actually 50.14 MHz)
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clk_unbuf));
   defparam vclk1.CLKFX_DIVIDE = 7;
   defparam vclk1.CLKFX_MULTIPLY = 13;
   defparam vclk1.CLK_FEEDBACK = "NONE";
   defparam vclk1.CLKIN_PERIOD = 37;
   BUFG vclk2(.O(clk),.I(clk_unbuf));


   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1’b0), .CLK(clock_27mhz), .Q(power_on_reset),
   .A0(1’b1), .A1(1’b1), .A2(1’b1), .A3(1’b1));
   defparam reset_sr.INIT = 16’hFFFF;


   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ˜button_enter, user_reset);
   defparam db1.DELAY = 500000;
   assign reset = user_reset | power_on_reset;


   // output reset signal to leds
   assign led = ˜{8{reset}};


   ////////////////////////////////////////////////////////////////////////////
   // MODULE INSTANCES
   ////////////////////////////////////////////////////////////////////////////

   parameter BUFW = 640;
   parameter BUFH = 480;
   parameter XY_BITS = 10;


   // video_memory
   wire flip;
   wire [XY_BITS-1:0] oldframe_rx0; //read x-pos
   wire [XY_BITS-1:0] oldframe_ry0; //read y-pos
   wire [17:0] oldframe_rc0; //read color
   wire oldframe_ready0; //indicates when address will be read
   wire [XY_BITS-1:0] oldframe_rx1; //read x-pos
   wire [XY_BITS-1:0] oldframe_ry1; //read y-pos
   wire [17:0] oldframe_rc1; //read color
   wire oldframe_ready1;
   wire [XY_BITS-2:0] newframe_rx; //missing the LSB (word)
   wire [XY_BITS-1:0] newframe_ry;
   wire [17:0] newframe_rc;
   wire [XY_BITS-2:0] newframe_wx; //missing the LSB (word)
   wire [XY_BITS-1:0] newframe_wy;
   wire [17:0] newframe_wc;
   wire newframe_we = 1;
   wire newframe_xbit; //the lowest order bit of both newframe read and write x-pos
   video_memory vm0(clk,flip,oldframe_rx0,oldframe_ry0,oldframe_rc0,oldframe_ready0,
oldframe_rx1,oldframe_ry1,oldframe_rc1,oldframe_ready1,
newframe_rx,newframe_ry,newframe_rc,
newframe_wx,newframe_wy,newframe_wc,newframe_we,newframe_xbit,
ram0_data,ram0_address,ram0_adv_ld,ram0_clk,ram0_cen_b,
ram0_ce_b,ram0_oe_b,ram0_we_b,ram0_bwe_b,
ram1_data,ram1_address,ram1_adv_ld,ram1_clk,ram1_cen_b,
ram1_ce_b,ram1_oe_b,ram1_we_b,ram1_bwe_b);
   defparam vm0.BUFW = BUFW;
   defparam vm0.BUFH = BUFH;
   defparam vm0.XY_BITS = XY_BITS;


   //vga
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire [17:0] color;
   vga vga0(.clk(clk),.color(color),.mem_ready(oldframe_ready0),.hcount(hcount),.vcount(vcount),
    .pixel_clock(vga_out_pixel_clock),.hsync(vga_out_hsync),.vsync(vga_out_vsync),.sync_b(vga_out_sync_b),
.blank(vga_out_blank_b),.red_out(vga_out_red),.green_out(vga_out_green),.blue_out(vga_out_blue));

   //display_test
   wire [17:0] color_test;
   display_test dt0(clk,hcount,vcount,color_test);

   //switch between display_test and memory
   assign color = button0 ? oldframe_rc0 : color_test;


   //connect hcount,vcount to x,y coordinates
   assign oldframe_rx0 = hcount[9:0];
   assign oldframe_ry0 = vcount[9:0];


   //audio pre-processing
   wire [9:0] l_sample_addr1, r_sample_addr1;
   wire [9:0] l_sample_data1, r_sample_data1;
   preproc ab0(clk, reset, flip,
                 ac97_bit_clock, ac97_sdata_out, ac97_sdata_in, ac97_synch, audio_reset_b,
                 l_sample_addr1, l_sample_data1, r_sample_addr1, r_sample_data1);


   //effects
   wire [31:0] param0,param1,param2,param3,param4,param5,param6,param7,param8,param9;
   wire start,done,gen_test;
   effects eff0(clk,reset,start,done,oldframe_rx1,oldframe_ry1,oldframe_rc1,oldframe_ready1,
newframe_rx,newframe_ry,newframe_rc,newframe_wx,newframe_wy,newframe_wc,
newframe_we,newframe_xbit,l_sample_addr1,l_sample_data1,r_sample_addr1,r_sample_data1,
gen_test,param0,param1,param2,param3,param4,param5,param6,param7);
   defparam eff0.BUFW = BUFW;
   defparam eff0.BUFH = BUFH;
   defparam eff0.XY_BITS = XY_BITS;

   //cpu
   //assign start = done | reset;
   //always @ (posedge clk) if (done) flip <= ~flip;



   // JTAG Programming Interface
   wire        jtag_we;
   wire [3:0]  jtag_state;
   wire [7:0]  jtag_addr;
   wire [31:0] jtag_read, jtag_write;
   jtag2mem #(.DBITS(32), .ABITS(6)) jtagr (reset, clk,
              jtag_read, jtag_write, jtag_addr, jtag_we, jtag_state);

   // CPU
   wire        mwe, msel, sgen_next, started_gen;
   wire [31:0] mdin, mdout, ma, npc;
   wire [5:0] opcode;

   wire        dbt0,   dbt1,   dbt2,   dbt3,   ds1,    ds2,    ds3,    ds4;
   wire [31:0] input0, input1, input2;

   debounce db2[7:0] (reset, clk,
                 {button0, button1, button2, button3, button_right,
                  button_left, button_down, button_up},
                 {dbt0, dbt1, dbt2, dbt3, ds4, ds3, ds2, ds1});

   assign input0 = {16'b0, dbt0, dbt1, dbt2, dbt3, ds1, ds2, ds3, ds4, switch[7:0]};
   assign input1 = user1;
   assign input2 = user2;

   cpuram mem (jtag_addr, ma[5:0], clk, clk, jtag_write, mdin, jtag_read, mdout, jtag_we, mwe);
   beta2 cpu (clk, reset | jtag_we, ma, mdout, mdin, mwe, flip, start, done,
              param0, param1, param2, param3, param4, param5, param6, param7, param8, param9,
              input0, input1, input2,
              opcode, msel, sgen_next, started_gen, npc);


   assign analyzer1_clock = clk;
   assign analyzer1_data = {opcode, start, done};
   assign analyzer2_data = {npc[7:0]};


   assign gen_test = ~button1;

   /*
   assign reg1[5:0] = {4'b0010,1'b0,switch[6]};  //rotate_mot_param
   assign reg1[11:6] = {4'b0010,switch[3:2]};  //stretch_mot_x_param
   assign reg1[17:12] = {4'b0010,switch[5:4]};  //stretch_mot_y_param

   assign reg2[9:0] = 10'd0; //translate_x_param
   assign reg2[19:10] = 10'd0; //translate_y_param
   assign reg2[25:20] = {4'b0011,1'b0,switch[0]};  //replicate_x_param
   assign reg2[31:26] = {4'b0011,1'b0,switch[1]};  //replicate_y_param

   assign reg3[7:0] = {1'b0,2'b01,4'b0010,1'b0}; //basic_waveform_gen_param
   assign reg3[23:8] = {10'd100,2'b00,4'b0011}; //box_wav_gen_param

   assign reg4 = 32'b0; //convolve_cop_param
   assign reg5 = {28'b0,4'b0101}; //oldframe_cop_param
   assign reg6 = {{9{1'b0}},{6{1'b1}},{6{1'b0}},{6{1'b0}},4'b0010}; //basic_waveform_gen_cop_param
   assign reg7 = {{9{1'b0}},{6{1'b1}},{6{1'b0}},{6{1'b0}},4'b0010}; //box_wav_gen_cop_param

   assign reg8[3:0] = {3'b000,button3}; //op1
   assign reg8[7:4] = {3'b000,switch[7]}; //op2
   assign reg8[11:8] = ~button3 ? 4'b0000 : {1'b0,button2,1'b0,~button2}; //op3
   */

   //assign analyzer1_clock = clk;
   //assign analyzer1_data = {ram0_address[7:0],ram0_data[7:0]};
   //assign analyzer2_data = {newframe_rc[13:6],newframe_wc[13:6]};
   //assign analyzer3_data = {zbt0_rd[13:6],3'b0,ram0_we_b,3'b0,flip};

endmodule
```

5.2.1 Audio Preprocessor

```
/*
 * Audio Frontend Module
 */

module preproc (sysclock, rst, flip,
                 ac97_bit_clock, ac97_sdata_out, ac97_sdata_in, ac97_synch, audio_reset_b,
                 l_sample_addr1, l_sample_data1, r_sample_addr1, r_sample_data1);

   input        sysclock, rst, flip, ac97_bit_clock, ac97_sdata_in;
   input  [9:0] l_sample_addr1, r_sample_addr1;
   output       audio_reset_b, ac97_sdata_out, ac97_synch;
   output [9:0] l_sample_data1, r_sample_data1;

   wire        new_frame, ready, sample_we;
   wire [19:0] left, right;
   reg old_ready;


   always @ (posedge ac97_bit_clock) old_ready <= rst ? 0 : ready;
   assign new_frame = ready & ~old_ready;
   assign sample_we = audio_reset_b & new_frame;

   audio audio (sysclock, ac97_bit_clock, rst, ready, left, right,
                audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch);

   buf_replram #(.DEPTH(1024), .WIDTH(10), .ADDR_WIDTH(10), .READ_PORTS(1)) l_sample_mem
                 (sysclock, rst, left[19:10],
                 {l_sample_addr1},
                 {l_sample_data1}, sample_we, flip, 1'b1);

   buf_replram #(.DEPTH(1024), .WIDTH(10), .ADDR_WIDTH(10), .READ_PORTS(1)) r_sample_mem
                 (sysclock, rst, right[19:10],
                 {r_sample_addr1},
                 {r_sample_data1}, sample_we, flip, 1'b1);

endmodule

/*
 * Simple interface to AC97 microphone input
 */

module audio (sysclock, ac97_bit_clock, rst, ready, left, right,
              audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch);

   input  sysclock, rst, ac97_bit_clock, ac97_sdata_in;
   output ready,         audio_reset_b,  ac97_sdata_out, ac97_synch;
   output [19:0] left, right;

   wire       source;
   wire [4:0] volume;
   assign source = 1;    //mic
   assign volume = 4'd28;  //a reasonable volume value

   wire        command_valid;
   wire [7:0]  command_address;
   wire [15:0] command_data;
   wire [19:0] left_out_data, right_out_data;

   assign left_out_data  = 12'b0;
   assign right_out_data = 12'b0;

   reg       audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge sysclock) begin
      if (rst) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   ac97 ac97(ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left,
             right, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   ac97commands cmds(sysclock, ready, command_address, command_data,
                     command_valid, volume, source);
endmodule


// Testing Module for replfifo (logical predecessor to replram)
module repltest(clk, rst, debug);
   input clk, rst;
   output [31:0] debug;
   wire [1:0] madd1, madd2, madd3, out1, out2, out3, addr_in;
   wire sinit1, sinit2, sinit3;

   reg [1:0] data;
   always @(posedge clk) begin
      if (rst)
         data <= 0;
      else
         data <= data + 1;
   end
   replfifo #(.DEPTH(3), .WIDTH(2), .ADDR_WIDTH(2)) fifo
             (clk, clk, rst, data,
             {2’b10,  2’b01,  2’b00},
             {out3,   out2,   out1}, 1’b1);

   assign debug = {clk, rst, data, out3, out2, out1};
endmodule

// Buffered, replicated BRAM with FIFO like input.
module buf_replram (clk, rst, d_in, addr_out, d_out, we, select, start);
   parameter READ_PORTS=3,
             WIDTH=10,
             ADDR_WIDTH=10,
             DEPTH=64;

   input                              clk, we, rst, select, start;
   input  [READ_PORTS*ADDR_WIDTH-1:0] addr_out;
//   input  [ADDR_WIDTH-1:0]            addr_in;
   input  [WIDTH-1:0]                 d_in;
   output [READ_PORTS*WIDTH-1:0]      d_out;

   reg                        started, old_sel, buf_select;
   reg  [ADDR_WIDTH:0]        writes;
   wire [ADDR_WIDTH-1:0]      addr_in;
   wire [READ_PORTS*WIDTH-1:0] d_out1, d_out2;
   wire                       we1, we2;

   assign we1 = we & ˜buf_select;
   assign we2 = we &  buf_select;
   assign d_out = (buf_select) ? d_out1 : d_out2;
   assign addr_in = writes - 1;

   assign new_sel = select != old_sel;
   always @ (posedge clk) begin
      old_sel <= rst ? 0 : select;

      if (rst) begin
         old_sel <= 0;
         writes <= 0;
         buf_select <= 0;
         started <= 0;
      end else if (new_sel && started && writes >= DEPTH)  begin
         buf_select <= ˜buf_select;
         writes <= 0;
         started <= 0;
      end else if (we && (started | start) && writes < DEPTH)
         writes <= writes + 1;

      if (start) started <= 1;
   end

   replram #(.READ_PORTS(READ_PORTS), .WIDTH(WIDTH), .ADDR_WIDTH(ADDR_WIDTH), .DEPTH(DEPTH)) rf1
             (clk, clk, rst, addr_in, d_in, addr_out, d_out1, we1);
   replram #(.READ_PORTS(READ_PORTS), .WIDTH(WIDTH), .ADDR_WIDTH(ADDR_WIDTH), .DEPTH(DEPTH)) rf2
             (clk, clk, rst, addr_in, d_in, addr_out, d_out2, we2);
endmodule

/* Replicated memory module
 *
 * Parameters:
 *  - READ_PORTS: number of output read ports
 *  - WIDTH: memory width
 *  - ADDR_WIDTH: memory address width
 *  - DEPTH: memory depth
 */
module replram (clk_in, clk_out, rst, addr_in, d_in, addr_out, d_out, we);
   parameter READ_PORTS=3,
             WIDTH=10,
             ADDR_WIDTH=10,
             DEPTH=64;

   input                              clk_in, clk_out, we, rst;
   input  [READ_PORTS*ADDR_WIDTH-1:0] addr_out;
   input  [ADDR_WIDTH-1:0]            addr_in;
   input  [WIDTH-1:0]                 d_in;
   output [READ_PORTS*WIDTH-1:0]      d_out;
   // Use replicated ram for storage
   replbrm brm[READ_PORTS-1:0] ({READ_PORTS{addr_in}}, addr_out,
                                  {READ_PORTS{clk_in}},  {READ_PORTS{clk_out}},
                                  {READ_PORTS{d_in}},    d_out,
                                  {READ_PORTS{1'b0}},    {READ_PORTS{we}});
endmodule

/* Replicated Memory Module with FIFO like input.  Designed for continuous read and write.
 *
 * Parameters:
 *  - READ_PORTS: number of output read ports
 *  - WIDTH: memory width
 *  - ADDR_WIDTH: memory address width
 *  - DEPTH: memory depth
 */
module replfifo (clk_in, clk_out, rst, d_in, addr_out, d_out, we);

   parameter READ_PORTS=3,
             WIDTH=10,
             ADDR_WIDTH=10,
             DEPTH=1024;

   input                              clk_in, clk_out, we, rst;
   input  [READ_PORTS*ADDR_WIDTH-1:0] addr_out;
   input  [WIDTH-1:0]                 d_in;
   output [READ_PORTS*WIDTH-1:0]      d_out;

   wire [ADDR_WIDTH-1:0]              next_addr;
   reg                                first;
   reg  [ADDR_WIDTH-1:0]              addr_in;

   wire [ADDR_WIDTH-1:0]              start;
   wire [READ_PORTS*ADDR_WIDTH-1:0]   m_addr_out;
   wire [READ_PORTS-1:0]              sinit;

   assign looped = !first && (addr_in == 0  || looped);

   // If we're going to write data this cycle (we high), the earliest
   // address useful for read is the next address. If we're not going
   // to write anything, the earliest is the current address.
   assign next_addr = addr_in + 1;
   assign start = (looped) ? (we) ? next_addr : addr_in :
                          {ADDR_WIDTH{1'b0}};

   // Calculate physical location of requested addr.
   // This is equal to: (start + addr) or (start + addr - depth)
   // if it has wrapped around.
   genvar i;
   generate
      for(i=1; i<=READ_PORTS; i=i+1) begin:shift
         wire[ADDR_WIDTH:0] sum, dif;
         assign sum  = {0, addr_out[i*ADDR_WIDTH-1:i*ADDR_WIDTH-ADDR_WIDTH]} + {0, start};
         assign dif  = sum - DEPTH;
         assign m_addr_out[i*ADDR_WIDTH-1:i*ADDR_WIDTH-ADDR_WIDTH] =
               (dif > DEPTH) ? sum[ADDR_WIDTH-1:0] : dif[ADDR_WIDTH-1:0];
         assign sinit[i-1] = rst || (˜looped && addr_out[i*ADDR_WIDTH-1:i*ADDR_WIDTH-ADDR_WIDTH] > addr_in);
      end
   endgenerate

   // Use replicated ram for storage
   replbrm brm[READ_PORTS-1:0] ({READ_PORTS{addr_in}}, m_addr_out,
                             {READ_PORTS{clk_in}},  {READ_PORTS{clk_out}},
                             {READ_PORTS{d_in}},    d_out,
                             sinit,                 {READ_PORTS{we}});

   // Increment the incoming write address if we is high
   always @(posedge clk_in) begin
      if (rst) begin
         addr_in <= 0;
         first <= 1;
      end else if (we) begin
         first <= 0;
         addr_in <= (addr_in < DEPTH-1) ? addr_in + 1 : 0;
      end
   end
```

```
endmodule


// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;

   initial begin
      ready <= 1’b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1’b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1’b0;
      // synthesis attribute init of ac97_synch is "0";

      bit_count <= 8’h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1’b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1’b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1’b0;
      // synthesis attribute init of l_right_v is "0";

      left_in_data <= 20’h00000;
      // synthesis attribute init of left_in_data is "00000";
      right_in_data <= 20’h00000;
      // synthesis attribute init of right_in_data is "00000";
   end

   always @(posedge ac97_bit_clock) begin
      // Generate the sync signal
      if (bit_count == 255)
        ac97_synch <= 1’b1;
      if (bit_count == 15)
        ac97_synch <= 1’b0;

      // Generate the ready signal
      if (bit_count == 128)
        ready <= 1’b1;
      if (bit_count == 2)
        ready <= 1’b0;

      // Latch user data at the end of each frame. This ensures that the
      // first frame after reset will be empty.
      if (bit_count == 255)
        begin
```

```
           l_cmd_addr <= {command_address, 12'h000};
           l_cmd_data <= {command_data, 4'h0};
           l_cmd_v <= command_valid;
           l_left_data <= left_data;
           l_left_v <= left_valid;
           l_right_data <= right_data;
           l_right_v <= right_valid;
        end

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
  4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

      else if ((bit_count >= 16) && (bit_count <= 35))
        // Slot 1: Command address (8-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

      else if ((bit_count >= 36) && (bit_count <= 55))
        // Slot 2: Command data (16-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

      else if ((bit_count >= 56) && (bit_count <= 75))
        begin
           // Slot 3: Left channel
           ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
           l_left_data <= { l_left_data[18:0], l_left_data[19] };
        end
      else if ((bit_count >= 76) && (bit_count <= 95))
        // Slot 4: Right channel
           ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
      else
        ac97_sdata_out <= 1'b0;

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
      if ((bit_count >= 57) && (bit_count <= 76))
        // Slot 3: Left channel
        left_in_data <= { left_in_data[18:0], ac97_sdata_in };
      else if ((bit_count >= 77) && (bit_count <= 96))
        // Slot 4: Right channel
        right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input source;

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   reg [3:0] state;

   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      done <= 1'b0;
      // synthesis attribute init of done is "0";
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

   assign command_address = command[23:16];
   assign command_data = command[15:0];

   wire [4:0] vol;
   assign vol = 31-volume;

   always @(posedge clock) begin
      if (ready && (!old_ready))
        state <= state+1;

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= 24'h1A_0000; // microphone
        4'h7: // Record gain = max
  command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands
```

5.2.2 CPU

```
module beta2(clk,reset,ma,mdin,mdout,mwe, flip, start_gen, done_gen,
             param0, param1, param2, param3, param4, param5, param6, param7, param8, param9,
             input0, input1, input2,
             opcode, msel, done_prg, sgen_msel_next, npc);
  input clk,reset, done_gen;
  output [31:0] ma,mdout;
  input [31:0] mdin;
  output mwe, flip, start_gen, sgen_msel_next;
  output [31:0] param0, param1, param2, param3, param4, param5, param6, param7, param8, param9;
  input [31:0] input0, input1, input2;

  output[31:0] npc;
  output [5:0] opcode;
  output msel, done_prg;

  // beta2 registers
  reg [31:0] regfile[31:0];
  reg [31:0] npc,pc_inc;
  reg [31:0] inst;
  reg [4:0]  rc_save;  // needed for second cycle on LD,LDR
  reg flip;
  assign opcode = inst[31:26];


  // internal buses
  wire [31:0] rd1,rd2,wd;
  wire [31:0] a,b,xb,c,addsub,cmp,shift,boole,mult;

  // control signals
  wire werf,z,asel,bsel;
  wire addsub_op,cmp_lt,cmp_eq,shift_op,shift_sxt,boole_and,boole_or;
  wire wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  wire /*msel,*/ msel_next,branch/*, done_prg*/;

  // pc
  wire [31:0] npc_inc,npc_next;
  assign npc_inc = npc + 1;
  assign npc_next = (reset) ? 32'h0 :
                  msel ? npc :
//                  done_prg ? 32'h0 :
                  branch ? addsub :
                  npc_inc;
  always @ (posedge clk) begin
    npc <= npc_next;   // logic for msel handled above
    if (!msel) pc_inc <= npc_inc;
  end

  // instruction reg
  always @ (posedge clk) if (!msel) inst <= mdin;

  // flip signal
  always @ (posedge clk) if (done_prg) flip <= ~flip;

  // control logic
  decode ctl(.clk(clk),.reset(reset),.z(z),
             .opcode(inst[31:26]),
             .asel(asel),.bsel(bsel),
             .werf(werf),.msel(msel),.msel_next(msel_next),.mwe(mwe),
             .addsub_op(addsub_op),.cmp_lt(cmp_lt),.cmp_eq(cmp_eq),
             .shift_op(shift_op),.shift_sxt(shift_sxt),
             .boole_and(boole_and),.boole_or(boole_or),
             .wd_addsub(wd_addsub),.wd_cmp(wd_cmp),
             .wd_shift(wd_shift),.wd_boole(wd_boole),.wd_mult(wd_mult),
             .branch(branch), .done_gen(done_gen), .start_gen(start_gen),
             .done_prg(done_prg), .sgen_msel(sgen_msel_next), .pdgen(pdgen) );

  // register file
  wire [4:0] ra1,ra2,wa;
  always @ (posedge clk) if (!msel) rc_save <= inst[25:21];
  assign ra1 = inst[20:16];
  assign ra2 = msel_next ? inst[25:21] : inst[15:11];
  assign wa = msel ? rc_save : inst[25:21];
  assign rd1 = (ra1 == 31) ? 0 : (ra1 != 26 && ra1 != 30) ? regfile[ra1] : // read port 1
             (ra1 == 26) ? input0 : (ra1==30) ? input1 : 0;

  assign rd2 = (ra2 == 31) ? 0 : (ra2 != 26 && ra2 != 30) ? regfile[ra2] : // read port 2
             (ra2 == 26) ? input0 : (ra2==30) ? input1 : 0;

  always @ (posedge clk) if (werf && (wa != 26 && wa != 30)) regfile[wa] <= wd;  // write port

  // output parameters
  assign param0 = regfile[16]; assign param1 = regfile[17];
  assign param2 = regfile[18]; assign param3 = regfile[19];
  assign param4 = regfile[20]; assign param5 = regfile[21];
  assign param6 = regfile[22]; assign param7 = regfile[23];
  assign param8 = regfile[24]; assign param9 = regfile[25];

  // input registers

  assign z = ~| rd1;   // used in BEQ/BNE instructions

  // alu
  assign a = asel ? pc_inc : rd1;
  assign b = bsel ? c : rd2;
  assign c = {{16{inst[15]}},inst[15:0]};
  wire addsub_n,addsub_v,addsub_z;
  assign xb = {32{addsub_op}} ^ b;
  assign addsub = a + xb + addsub_op;
  assign addsub_n = addsub[31];
  assign addsub_v = (addsub[31] & ~a[31] & ~xb[31]) |
                    (~addsub[31] & a[31] & xb[31]);
  assign addsub_z = ~| addsub;

  assign cmp[31:1] = 0;
  assign cmp[0] = (cmp_lt & (addsub_n ^ addsub_v)) | (cmp_eq & addsub_z);

  //mul32 mpy(a,b,mult);

  wire [31:0] shift_right;
  // Verilog >>> operator not synthesized correctly, so do it by hand
  shift_right sr(shift_sxt,a,b[4:0],shift_right);
  assign shift = shift_op ? shift_right : a << b[4:0];

  assign boole = boole_and ? (a & b) : boole_or ? (a | b) : a ^ b;

  // result mux, listed in order of speed (slowest first)
  assign wd = msel ? mdin :
              wd_cmp ? cmp :
              wd_addsub ? addsub :
              //wd_mult ? mult :
              wd_shift ? shift :
              wd_boole ? boole :
              pc_inc;

  // assume synchronous external memory
  assign ma = msel_next ? {npc[31],addsub[30:0]} : npc_next;
  assign mdout = rd2;
endmodule


module decode(clk, reset, z, opcode,
              asel, bsel, werf, msel, msel_next, mwe,
              addsub_op, cmp_lt, cmp_eq,
              shift_op, shift_sxt, boole_and, boole_or,
              wd_addsub, wd_cmp, wd_shift, wd_boole, wd_mult,
           branch, done_gen, start_gen, done_prg, sgen_msel, pdgen);
  input clk,reset,z, done_gen;
  input [5:0] opcode;
  output asel,bsel,werf,msel,msel_next,mwe;
  output addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  output wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  output branch, start_gen, done_prg , sgen_msel, pdgen;

  reg asel,bsel,mem_next;
  reg addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  reg wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  reg branch, gen, done_prg;

  // a little bit of state...
  reg annul,cmsel,mwrite;

  always @ (opcode or z or annul or msel or reset)
  begin
    // initial assignments for all control signals
    asel = 1'hx;
    bsel = 1'hx;
    addsub_op = 1'hx;
    shift_op = 1'hx;
    shift_sxt = 1'hx;
    cmp_lt = 1'hx;
    cmp_eq = 1'hx;
    boole_and = 1'hx;
    boole_or = 1'hx;

    mem_next = 0;

    wd_addsub = 0;
    wd_cmp = 0;
    wd_shift = 0;
    wd_boole = 0;
    wd_mult = 0;
```

```
 done_prg = 0;
 branch = 0;

 casez (opcode)
   6’b010101: begin   // END PROGRAM -> BUFFER FLIP, RESET PC
                done_prg = !annul && !msel;
              end
   6’b010100: begin   // GENERATE
              end
   6’b011000: begin   // LD
                asel = 0; bsel = 1;
                addsub_op = 0;
mem_next = 1;
              end
   6’b011001: begin   // ST
                asel = 0; bsel = 1;
                addsub_op = 0;
mem_next = 1;
              end
   6’b011011: begin   // JMP
                asel = 0; bsel = 1;
                addsub_op = 0;
                branch = !annul && !msel;
              end
   6’b011101: begin   // BEQ
                asel = 1; bsel = 1;
                addsub_op = 0;
                branch = !annul && !msel && z;
              end
   6’b011110: begin   // BNE
                asel = 1; bsel = 1;
                addsub_op = 0;
                branch = !annul && !msel && ˜z;
              end
   6’b011111: begin   // LDR
                asel = 1; bsel = 1;
                addsub_op = 0;
mem_next = 1;
              end
   6’b1?0000: begin   // ADD, ADDC
                asel = 0; bsel = opcode[4];
                addsub_op = 0;
                wd_addsub = 1;
              end
   6’b1?0001: begin   // SUB, SUBC
                asel = 0; bsel = opcode[4];
                addsub_op = 1;
                wd_addsub = 1;
              end
   //6’b1?0010: begin   // MUL, MULC
   //             asel = 0; bsel = opcode[4];
   //             wd_mult = 1;
   //           end
   6’b1?0100: begin   // CMPEQ, CMPEQC
                asel = 0; bsel = opcode[4];
                addsub_op = 1;
cmp_eq = 1; cmp_lt = 0;
                wd_cmp = 1;
              end
   6’b1?0101: begin   // CMPLT, CMPLTC
                asel = 0; bsel = opcode[4];
                addsub_op = 1;
cmp_eq = 0; cmp_lt = 1;
                wd_cmp = 1;
              end
   6’b1?0110: begin   // CMPLE, CMPLEC
                asel = 0; bsel = opcode[4];
                addsub_op = 1;
cmp_eq = 1; cmp_lt = 1;
                wd_cmp = 1;
              end
   6’b1?1000: begin   // AND, ANDC
                asel = 0; bsel = opcode[4];
                boole_and = 1; boole_or = 0;
                wd_boole = 1;
```

```
                 end
      6’b1?1001: begin   // OR, ORC
                   asel = 0; bsel = opcode[4];
                   boole_and = 0; boole_or = 1;
                   wd_boole = 1;
                 end
      6’b1?1010: begin   // XOR, XORC
                   asel = 0; bsel = opcode[4];
                   boole_and = 0; boole_or = 0;
                   wd_boole = 1;
                 end
      6’b1?1100: begin   // SHL, SHLC
                   asel = 0; bsel = opcode[4];
                   shift_op = 0;
                   wd_shift = 1;
                 end
      6’b1?1101: begin   // SHR, SHRC
                   asel = 0; bsel = opcode[4];
                   shift_op = 1; shift_sxt = 0;
                   wd_shift = 1;
                 end
      6’b1?1110: begin   // SRA, SRAC
                   asel = 0; bsel = opcode[4];
                   shift_op = 1; shift_sxt = 1;
                   wd_shift = 1;
                 end
      default:   begin   // illegal opcode

                 end
    endcase
  end

  // state
  reg psgen, pdgen;

  wire sgen = !reset && opcode==6’b010100;

  wire msel_next = !reset && !annul && mem_next && !msel;
  assign sgen_msel = !reset && !annul && !done_gen && !pdgen && sgen;

  wire mwrite_next = msel_next && opcode==6’b011001;

  wire start_gen = sgen && ˜psgen;

  assign msel = cmsel | sgen_msel;
  always @ (posedge clk)
  begin
   psgen <= sgen;
   pdgen <= done_gen;

   annul <= !reset && (branch/* | done_prg*/);
   cmsel <= msel_next;
   mwrite <= mwrite_next;
  end

  assign mwe = mwrite_next;   // assume synchronous memory
  assign werf = msel ? !mwrite : (!annul & !mem_next);
endmodule


module shift_right(sxt,a,b,shift_right);
  input sxt;
  input [31:0] a;
  input [4:0] b;
  output [31:0] shift_right;

  wire [31:0] w,x,y,z;
  wire sin;

  assign sin = sxt & a[31];
  assign w = b[0] ? {sin,a[31:1]} : a;
  assign x = b[1] ? {{2{sin}},w[31:2]} : w;
  assign y = b[2] ? {{4{sin}},x[31:4]} : x;
  assign z = b[3] ? {{8{sin}},y[31:8]} : y;
  assign shift_right = b[4] ? {{16{sin}},z[31:16]} : z;
endmodule
```

5.2.3 Effects

```
module one_color_op(color_in,color_out,param);

     input [17:0] color_in;
     output [17:0] color_out;

     input [31:0] param;

     //parameter function assignments
     wire [3:0] op = param[3:0];
     wire [17:0] color = param[21:4];

     assign color_out =
          (op == 4’b0000) ? color_in :  // noop
          (op == 4’b0001) ? ˜color_in : // not
          (op == 4’b0010) ? color_in & color : // and C
          (op == 4’b0011) ? color_in | color : // or C
          (op == 4’b0100) ? {color_in[17:12]>>1,color_in[11:6]>>1,color_in[5:0]>>1} : //shr
          (op == 4’b0101) ? color_in >> 1 : //shift to blue
          (op == 4’b0110) ? color_in << 1 : //shift to red
          (op == 4’b0111) ? {color_in[0],color_in[17:1]} : //cycle down
          (op == 4’b1000) ? {color_in[16:0],color_in[17]} : // cycle up
          18’b0;

endmodule


module two_color_op(color_a,color_b,color_out,param);

     input [17:0] color_a,color_b;
     output [17:0] color_out;

     input [3:0] param;

     assign color_out =
          (param == 4’b0000) ? color_a :  // A
          (param == 4’b0001) ? color_b :  // B
          (param == 4’b0010) ? ((color_a == 18’b0) ? color_b : color_a) :  // A over B
          (param == 4’b0011) ? ((color_b == 18’b0) ? color_a : color_b) :  // B over A
          (param == 4’b0100) ? (color_a | color_b) :  // bitwise OR
          18’b0;

endmodule


//basic_waveform_gen: produces a waveform from audio data
//                  across the center of the frame

module basic_waveform_gen(clk,x,y,sample_addr,sample_data,color,param);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;

     input signed [XY_BITS:0] x, y;

     //audio data
     output [9:0] sample_addr;
     input signed [9:0] sample_data;

     output [17:0] color; //color value for coordinate x,y

     input [7:0] param;

     //parameter function assignments
     wire vertical = param[0];
     wire [3:0] mag_shift = param[4:1];
     wire [1:0] thickness = param[6:5];

     wire signed [9:0] sample_data_shift = sample_data >>> mag_shift;
     assign sample_addr = (vertical ? y : x) + 512; //request addresses from middle of audio buffer
     //when position-value and data match (within thickness), output on_color
     wire [17:0] color_nodelay =
          ((sample_data_shift >>> thickness) == ((vertical ? x : y) >>> thickness)) ?
          {18{1’b1}} : {18{1’b0}};

     //delay output: all generators must have a 5-cycle pipeline to match oldframe_fetch
     pipeline_delay pd0[17:0](clk,color_nodelay,color);

endmodule


module box_wav_gen(clk,x,y,sample_addr,sample_data,color,param);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;

     input signed [XY_BITS:0] x, y;

     //audio data
     output [9:0] sample_addr;
     input signed [9:0] sample_data;

     output [17:0] color; //color value for coordinate x,y

     input [15:0] param;

     //parameter function assignments
     wire [3:0] mag_shift = param[3:0];
     wire [1:0] thickness = param[5:4];
     wire [9:0] size = param[15:6];

     wire x_gt_y = (x > y);
     wire x_gt_ny = (x > -y);

     wire signed [XY_BITS:0] addr_dim =
          (˜x_gt_y && x_gt_ny) ? x : //top side
          (x_gt_y && ˜x_gt_ny) ? -x : //bottom side
          (x_gt_y && x_gt_ny) ? y : //right side
          -y; //left side
     wire signed [XY_BITS:0] data_dim =
          (˜x_gt_y && x_gt_ny) ? y : //top side
          (x_gt_y && ˜x_gt_ny) ? -y : //bottom side
          (x_gt_y && x_gt_ny) ? x : //right side
          -x; //left side


     wire signed [9:0] sample_data_shift = sample_data >>> mag_shift;
     assign sample_addr = addr_dim + 512; //request addresses from middle of audio buffer

     //when position-value and data match (within thickness), output on_color
     wire [17:0] color_nodelay =
          ((sample_data_shift >>> thickness) == ((data_dim - size) >>> thickness)) ?
          {18{1’b1}} : {18{1’b0}};

     //delay output: all generators must have a 5-cycle pipeline to match oldframe_fetch
     pipeline_delay pd0[17:0](clk,color_nodelay,color);

endmodule


//effects: collection of effects modules which generates a new frame
//         based on the parameters in the register inputs.

module effects(clk,reset,start,done,oldframe_rx,oldframe_ry,oldframe_rc,oldframe_ready,
          newframe_rx,newframe_ry,newframe_rc,newframe_wx,newframe_wy,newframe_wc,
          newframe_we,newframe_xbit,sample_addr0,sample_data0,sample_addr1,sample_data1,
          gen_test,param0,param1,param2,param3,param4,param5,param6,param7);

     parameter BUFW = 4;
     parameter BUFH = 3;
     parameter XY_BITS = 10;

     input clk;
     input reset;

     input start; //begins operation over the entire frame buffer in memory
     output done; //signals when the operation is finished

     //oldframe read port (half-rate, 5-cycle pipeline)
     output [XY_BITS-1:0] oldframe_rx; //read x-pos
     output [XY_BITS-1:0] oldframe_ry; //read y-pos
     input [17:0] oldframe_rc; //read color
     input oldframe_ready;

     //newframe pipelined read port (full-rate, 5-cycle pipeline)
     output [XY_BITS-2:0] newframe_rx; //missing the LSB (word)
     output [XY_BITS-1:0] newframe_ry;
     input [17:0] newframe_rc;

     //newframe pipelined write port (full-rate, 6-cycle pipeline)
     output [XY_BITS-2:0] newframe_wx; //missing the LSB (word)
     output [XY_BITS-1:0] newframe_wy;
     output [17:0] newframe_wc;
     output newframe_we;

     output newframe_xbit; //the lowest order bit of both newframe read and write x-pos

     //audio data
     output [9:0] sample_addr0;
     input [9:0] sample_data0;
     output [9:0] sample_addr1;
     input [9:0] sample_data1;

     //parameters
     input gen_test; //activates test generator, overriding other generators
     input [31:0] param0,param1,param2,param3,param4,param5,param6,param7;



     //parameter function assignments
     wire [5:0] rotate_mot_param = param0[5:0];
     wire [5:0] stretch_mot_x_param = param0[11:6];
     wire [5:0] stretch_mot_y_param = param0[17:12];

     wire [9:0] translate_x_param = param1[9:0];
     wire [9:0] translate_y_param = param1[19:10];
     wire [5:0] replicate_x_param = param1[25:20];
     wire [5:0] replicate_y_param = param1[31:26];

     wire [7:0] basic_waveform_gen_param = param2[7:0];
     wire [15:0] box_wav_gen_param = param2[23:8];

     wire [31:0] newframe_cop_param = param3;
     wire [31:0] oldframe_cop_param = param4;
     wire [31:0] basic_waveform_gen_cop_param = param5;
     wire [31:0] box_wav_gen_cop_param = param6;

     //wire [31:0] convolve_param = reg8;

     wire [3:0] op1_param = param7[3:0];
     wire [3:0] op2_param = param7[7:4];
     wire [3:0] op3_param = param7[11:8];



     //pixel_fsm
     wire [XY_BITS-1:0] x,y;
     wire oldframe_enable = ((op1_param != 4’b0000) && (op3_param != 4’b0000));
     pixel_fsm pf0(clk,reset,start,done,oldframe_ready,oldframe_enable,x,y);
          defparam pf0.BUFW = BUFW;
          defparam pf0.BUFH = BUFH;
          defparam pf0.XY_BITS = XY_BITS;


     //newframe fetch
     assign newframe_rx = x[XY_BITS-1:1];
     assign newframe_ry = y;
     wire [17:0] newframe_color = newframe_rc;
```

```
//generate "running" signal, to control newframe_we
reg running = 0;
always @ (posedge clk) begin
     if (start) running <= 1;
     else if (done) running <= 0;
end

//delay all control signals to memory
// 1 pipeline stage + 5 for oldframe (and generators)
pipeline_delay #(6) pd0(clk, running, newframe_we);
pipeline_delay #(6) pd1[XY_BITS-1:0](clk, x, {newframe_wx,newframe_xbit});
pipeline_delay #(6) pd2[XY_BITS-1:0](clk, y, newframe_wy);



//convert to signed, centered coordinates
wire signed [XY_BITS:0] x_signed = x;
wire signed [XY_BITS:0] y_signed = y;
wire signed [XY_BITS:0] x_offset = x_signed - BUFW/2;
wire signed [XY_BITS:0] y_offset = y_signed - BUFH/2;

//replicate
wire signed [XY_BITS:0] x_repl;
wire signed [XY_BITS:0] y_repl;
replicate rx0(x_offset,x_repl,replicate_x_param);
     defparam rx0.XY_BITS = XY_BITS;
replicate ry0(y_offset,y_repl,replicate_y_param);
     defparam ry0.XY_BITS = XY_BITS;

//rotate motion
wire signed [XY_BITS:0] rotate_mot_dx;
wire signed [XY_BITS:0] rotate_mot_dy;
rotate_mot mot0(x_repl,y_repl,rotate_mot_dx,rotate_mot_dy,rotate_mot_param);
     defparam mot0.BUFW = BUFW;
     defparam mot0.BUFH = BUFH;
     defparam mot0.XY_BITS = XY_BITS;

//stretch motion
wire signed [XY_BITS:0] stretch_mot_dx;
stretch_mot stx0(x_repl,stretch_mot_dx,stretch_mot_x_param);
     defparam stx0.XY_BITS = XY_BITS;
wire signed [XY_BITS:0] stretch_mot_dy;
stretch_mot sty0(y_repl,stretch_mot_dy,stretch_mot_y_param);
     defparam sty0.XY_BITS = XY_BITS;


//pipeline stage: sum motion vectors
reg signed [XY_BITS:0] mot_x;
reg signed [XY_BITS:0] mot_y;
always @ (posedge clk) begin
     mot_x <= x_offset + rotate_mot_dx + stretch_mot_dx - translate_x_param;
     mot_y <= y_offset + rotate_mot_dy + stretch_mot_dy - translate_y_param;
end

// generators
wire [17:0] flat_wav_color;
basic_waveform_gen gen0(clk,mot_x,mot_y,sample_addr0,
     sample_data0,flat_wav_color,basic_waveform_gen_param);
     defparam gen0.BUFW = BUFW;
     defparam gen0.BUFH = BUFH;
     defparam gen0.XY_BITS = XY_BITS;

wire [17:0] box_wav_color;
box_wav_gen gen1(clk,mot_x,mot_y,sample_addr1,
     sample_data1,box_wav_color,box_wav_gen_param);
     defparam gen1.BUFW = BUFW;
     defparam gen1.BUFH = BUFH;
     defparam gen1.XY_BITS = XY_BITS;

wire [17:0] test_color;
test_gen gen2(clk,mot_x,mot_y,test_color);
     defparam gen2.BUFW = BUFW;
     defparam gen2.BUFH = BUFH;
     defparam gen2.XY_BITS = XY_BITS;
     //oldframe_fetch
     wire [17:0] oldframe_color;
     oldframe_fetch of0(clk,mot_x,mot_y,oldframe_color,
          oldframe_rx,oldframe_ry,oldframe_rc,oldframe_ready);
          defparam of0.BUFW = BUFW;
          defparam of0.BUFH = BUFH;
          defparam of0.XY_BITS = XY_BITS;

     //blend ops
     wire [17:0] newframe_postop,oldframe_postop,flat_wav_postop,box_wav_postop,test_postop;
     one_color_op cop0(newframe_color,newframe_postop,newframe_cop_param);
     one_color_op cop1(oldframe_color,oldframe_postop,oldframe_cop_param);
     one_color_op cop2(flat_wav_color,flat_wav_postop,basic_waveform_gen_cop_param);
     one_color_op cop3(box_wav_color,box_wav_postop,box_wav_gen_cop_param);

     wire [17:0] op1_color,op2_color,op3_color;
     two_color_op cop4(newframe_postop,oldframe_postop,op1_color,op1_param);
     two_color_op cop5(flat_wav_postop,box_wav_postop,op2_color,op2_param);
     two_color_op cop6(op1_color,op2_color,op3_color,op3_param);
     assign newframe_wc = gen_test ? test_color : op3_color;

endmodule


module newframe_fetch(clk,x,y,color,newframe_rx,newframe_ry,newframe_rc);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;

     input [XY_BITS-1:0] x,y;
     output [17:0] color;

     //newframe pipelined read port (full-rate, 5-cycle pipeline)
     output [XY_BITS-2:0] newframe_rx; //missing the LSB (word)
     output [XY_BITS-1:0] newframe_ry;
     input [17:0] newframe_rc;

     assign newframe_ry = y;
     assign newframe_rx = x[XY_BITS-1:1];

     assign color = newframe_rc;

endmodule


//oldframe_fetch: gets pixels from oldframe_memory

module oldframe_fetch(clk,x_offset,y_offset,color,
     oldframe_rx,oldframe_ry,oldframe_rc,oldframe_ready);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;

     //these coordinates are based on the center of the screen
     input signed [XY_BITS:0] x_offset,y_offset;
     output [17:0] color;

     //oldframe read port (half-rate, 5-cycle pipeline)
     output [XY_BITS-1:0] oldframe_rx; //read x-pos
     output [XY_BITS-1:0] oldframe_ry; //read y-pos
     input [17:0] oldframe_rc; //read color
     input oldframe_ready;

     //connect oldframe read port
     wire signed [XY_BITS:0] x_abs = x_offset + BUFW/2; //create absolute coordinates
     wire signed [XY_BITS:0] y_abs = y_offset + BUFH/2;
     assign oldframe_rx = x_abs[XY_BITS-1:0]; //make unsigned
     assign oldframe_ry = y_abs[XY_BITS-1:0];
     //connect color, blanking if offscreen
     wire offscreen = (x_abs < 0 || x_abs >= BUFW || y_abs < 0 || y_abs >= BUFH);
     wire offscreen_delayed; //pipeline the bit indicating an offscreen coordinate
```

```
                         // to match when the color is returned from oldframe
     pipeline_delay pd0(clk,offscreen,offscreen_delayed);
     assign color = offscreen_delayed ? 18’b0 : oldframe_rc; //blank if offscreen

endmodule


//pipeline_delay: passes signals from "in" to "out"
//   with a pipeline delay of DELAY, to correct for
//   memory pipelining.

module pipeline_delay(clk,in,out);

     parameter DELAY = 5; // must be >= 4

     input clk;
     input in;
     output out;

     reg [DELAY-3:0] buffer = 0;
     reg out = 0;
     always @ (posedge clk) begin

          buffer[0] <= in;
          buffer[DELAY-3:1] <= buffer[DELAY-4:0];
          out <= buffer[DELAY-3];

     end

endmodule


//pixel_fsm: steps through all pixel coordinates
//       asserts output_ready for each new coordinate
//       and waits until input_ready to advance

module pixel_fsm(clk,reset,start,done,oldframe_ready,oldframe_enable,x,y);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;
     input reset;

     input start;
     output done;

     input oldframe_ready;
     input oldframe_enable;

     output [XY_BITS-1:0] x,y;


     reg [XY_BITS-1:0] x = 0;
     reg [XY_BITS-1:0] y = 0;
     reg running = 0;
     reg done = 0;
     always @ (posedge clk) begin //loop through all display coordinates
          if (start) begin
               x <= 0;
               y <= 0;
               running <= 1;
               done <= 0;
               end
          else if (running && ˜(oldframe_enable && ˜oldframe_ready)) begin
               if (x < BUFW) x <= x + 1;
               else begin
                    x <= 0;
                    if (y < BUFH) y <= y + 1;
                    else begin
                         running <= 0; //done with frame- stop
                         done <= 1;
                         end
                    end
               end
          else done <= 0;
     end

endmodule


//replicate: replicates coordinates with a modulo operation

module replicate(pos_in,pos_out,param);

     parameter XY_BITS = 10;

     input signed [XY_BITS:0] pos_in;
     output signed [XY_BITS:0] pos_out;

     input [5:0] param;

     //parameter function assignments
     wire enable = param[0];
     //wire mirror = param[1];
     wire [3:0] amount_shift = param[5:2]; //mod by 2ˆamount_shift

     //do modulo operation by removing upper bits with shift left and shift right
     wire signed [XY_BITS:0] pos_shl = pos_in <<< amount_shift;
     assign pos_out = enable ? (pos_shl >>> amount_shift) : pos_in;

endmodule


//rotate_mot: generate a vector from offset coordinates that simulates
//            a rotation of up to 45 degrees.
//            this simple algorithm scales the frame as a side effect,
//            and has significant artifacts at small rotations.

module rotate_mot(x_in,y_in,dx_out,dy_out,param);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input signed [XY_BITS:0] x_in, y_in; //offset coordinates: rotation is around 0,0

     output signed [XY_BITS:0] dx_out, dy_out; //change in coordinates simulating rotation

     input [5:0] param; //parameters controlling rotation

     //parameter function assignments
     wire enable = param[0];
     wire clockwise = param[1];
     wire [3:0] amount_shift = param[5:2]; //bit-shifts the output vector
                                        // to control amount of rotation

     //rotate algorithm: x=c*y, y=-c*x
     wire signed [XY_BITS:0] x_dir = clockwise ? y_in : -y_in;
     wire signed [XY_BITS:0] y_dir = clockwise ? -x_in : x_in;

     assign dx_out = enable ? (x_dir >>> amount_shift) : 0;
     assign dy_out = enable ? (y_dir >>> amount_shift) : 0;

endmodule


module test_gen(clk,x,y,color);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;

     input signed [XY_BITS:0] x, y;

     output [17:0] color;

     wire [17:0] color_nodelay = (x == (-BUFW/2) || x == (BUFW/2-1) || y == (-BUFH/2) || y == (BUFH/2-1)) ?
               {18{1'b1}} : {x[8:3],x[5:0],y[5:0]};

     //delay output: all generators must have a 5-cycle pipeline to match oldframe_fetch
     pipeline_delay pd0[17:0](clk,color_nodelay,color);
```

```
endmodule


//stretch_mot: stretches an axis

module stretch_mot(pos_in,dpos_out,param);

     parameter XY_BITS = 10;

     input signed [XY_BITS:0] pos_in; //position input

     output signed [XY_BITS:0] dpos_out; //change in position for stretch

     input [5:0] param;

     wire enable = param[0];
     wire invert = param[1];
     wire [3:0] amount_shift = param[5:2];

     //the change is proportional to position
     wire [XY_BITS:0] div_pos = (pos_in >>> amount_shift);

     assign dpos_out = enable ? (invert ? div_pos : -div_pos) : 0;

endmodule
```

5.2.4 Video Memory

```
//display_test: outputs a color pattern with white boundary
//  from hcount,vcount for testing the display

module display_test(clk,hcount,vcount,color);
     input clk;
     input [10:0] hcount;
     input [9:0] vcount;
     output [17:0] color;

     wire [17:0] color_unbuf;

     assign color_unbuf = (hcount==0 | hcount==639 | vcount==0 | vcount==479) ?
          {18{1'b1}} :
          {hcount[8:3],hcount[5:0],vcount[5:0]};

     pipeline_delay pd0[17:0](clk,color_unbuf,color);

endmodule


// memory_flip: muxes oldram and newram interfaces to two ram modules,
//     switching them based on the flip signal

module memory_flip(flip,oldram_we,oldram_addr,oldram_wd,oldram_rd,
          newram_we,newram_addr,newram_wd,newram_rd,
          ram0_we,ram0_addr,ram0_wd,ram0_rd,
          ram1_we,ram1_addr,ram1_wd,ram1_rd);

     input flip; //specifies which ram connects to oldram

     //oldram interface
     input oldram_we;
     input [18:0] oldram_addr;
     input [35:0] oldram_wd;
     output [35:0] oldram_rd;

     //newram interface
     input newram_we;
     input [18:0] newram_addr;
     input [35:0] newram_wd;
     output [35:0] newram_rd;

     //zbt ram0 connections
     output ram0_we;
     output [18:0] ram0_addr;
     output [35:0] ram0_wd;
     input [35:0] ram0_rd;
     //zbt ram1 connections
     output ram1_we;
     output [18:0] ram1_addr;
     output [35:0] ram1_wd;
     input [35:0] ram1_rd;


     //muxes

     assign oldram_rd = flip ? ram1_rd : ram0_rd;
     assign newram_rd = flip ? ram0_rd : ram1_rd;

     assign ram0_we = flip ? newram_we : oldram_we;
     assign ram0_addr = flip ? newram_addr : oldram_addr;
     assign ram0_wd = flip ? newram_wd : oldram_wd;

     assign ram1_we = flip ? oldram_we : newram_we;
     assign ram1_addr = flip ? oldram_addr : newram_addr;
     assign ram1_wd = flip ? oldram_wd : newram_wd;


endmodule


//newframe memory: interlaces ram access to provide
//  pipelined read and write ports.
//  the LSBs of read and write addresses are shared.

module newframe_memory(clk,read_addr,read_data,write_addr,write_data,write_enable,word,
     ram_we,ram_addr,ram_wd,ram_rd);

     //interlacing places restrictions on address access:
     //  LSB of read address and write address must be identical,
     //   so it is separated into the "word" signal.
     //Addresses should be accessed sequentially:
     //  The other 19 bits of the read address are only read when word=0
     //  and for the write address when word=1.

     input clk;

     //pipelined read port (full-rate, 5-cycle pipeline)
     input [18:0] read_addr; //missing the LSB (word)
     output [17:0] read_data;

     //pipelined write port (full-rate, 6-cycle pipeline)
     input [18:0] write_addr; //missing the LSB (word)
     input [17:0] write_data;
     input write_enable;

     input word; //the lowest order bit of both read and write addresses

     //zbt ram connections
     output ram_we;
     output [18:0] ram_addr;
     output [35:0] ram_wd;
     input [35:0] ram_rd;


     reg [17:0] wd_word0_buf = 0; //write happens when word1 is received
     reg [17:0] rd_word1_buf = 0; //word0 read data presented first
     reg word_buf0 = 0;
     reg word_buf1 = 0; //remember which word of the read data is needed
     reg word_buf2 = 0;
     reg ram_we = 0;
     reg [18:0] ram_addr     = 0;
     reg [35:0] ram_wd = 0;
     reg [17:0] read_data = 0;
     always @ (posedge clk) begin

          //input

          if (˜word) begin //first word, do a read
               ram_we <= 0;
               ram_addr <= read_addr;
               wd_word0_buf <= write_data; //buffer word0 for writing
               end
          else begin       //second word, do a write
               ram_we <= write_enable;
               ram_addr <= write_addr;
               ram_wd <= {write_data, wd_word0_buf};
               end

          //output

          word_buf0 <= word;
          word_buf1 <= word_buf0;
          word_buf2 <= word_buf1;

          if (~word_buf2) begin
               read_data <= ram_rd[17:0];     //output first word
               rd_word1_buf <= ram_rd[35:18];//buffer second word
               end
          else read_data <= rd_word1_buf; //output second word
     end

endmodule


//oldframe memory: manages oldram read access, providing
//   two half-rate read ports with a 5-cycle pipeline.
//   Ready signals indicate which port can provide an address that cycle.

module oldframe_memory(clk,addr0,data0,ready0,addr1,data1,ready1,
     ram_we,ram_addr,ram_wd,ram_rd);

     input clk;

     //read port 0 (half-rate, 5-cycle pipeline)
     input [19:0] addr0;
     output [17:0] data0;
     output ready0; //indicates when address will be read

     //read port 1 (half-rate, 5-cycle pipeline)
     input [19:0] addr1;
     output [17:0] data1;
     output ready1;

     //zbt ram connections
     output ram_we;
     output [18:0] ram_addr;
     output [35:0] ram_wd;
     input [35:0] ram_rd;

     assign ram_we = 0; //oldframe is read-only
     assign ram_wd = 0;

     reg active_port = 0;
     reg [17:0] data0 = 0;
     reg [17:0] data1 = 0;
     reg [18:0] ram_addr = 0;
     reg ram_word_buf0 = 0; //buffer the last bit so we know which data to pass..
     reg ram_word_buf1 = 0;
     reg ram_word_buf2 = 0;
     always @ (posedge clk) begin
          active_port <= ~active_port; //alternate active ports
          {ram_addr,ram_word_buf0} <= active_port ? addr0 : addr1;
          ram_word_buf1 <= ram_word_buf0;
          ram_word_buf2 <= ram_word_buf1;

          if (~active_port) data0 <= ram_word_buf2 ? ram_rd[35:18] : ram_rd[17:0];
          else data1 <= ram_word_buf2 ? ram_rd[35:18] : ram_rd[17:0];
     end

     assign ready0 = ~active_port;
     assign ready1 = active_port;

endmodule


// vga: generate svga display signals (800x600 @ 60Hz)
//     from a color and a (25MHz) mem_ready signal.
//     signals are delayed with pipeline_delay to match
//     the expected delay of the color.
```

```
module vga(clk,color,mem_ready,hcount,vcount,pixel_clock,
     hsync,vsync,sync_b,blank,red_out,green_out,blue_out);

   input clk; //50 MHz
   input [17:0] color; //18bit color in
   input mem_ready; //oldframe_memory ready signal, every other cycle

   //coordinates to request
   output [10:0] hcount;
   output [9:0] vcount;

   //vga outputs
   output pixel_clock; //25 MHz
   output vsync,hsync,sync_b,blank;
   output [7:0] red_out,green_out,blue_out;


   reg hsync_u,vsync_u,hblank_u,vblank_u,blank_u;     //un-delayed signals
   reg [10:0] hcount;    // pixel number on current line
   reg [9:0] vcount;      // line number


   //color outputs wired directly to color input
   assign red_out = {color[17:12],2’b00};
   assign green_out = {color[11:6],2’b00};
   assign blue_out = {color[5:0],2’b00};

   assign sync_b = 1; //not used
   assign pixel_clock = ˜mem_ready;

   //delay vga control signals to account for memory pipeline
   pipeline_delay pd0(clk, ˜blank_u, blank);
   pipeline_delay pd1(clk, hsync_u, hsync);
   pipeline_delay pd2(clk, vsync_u, vsync);



   // horizontal: 800 pixels total
   // display 640 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 639); //hsize-1
   assign    hsyncon = (hcount == 655);      //hsize+fp-1
   assign    hsyncoff = (hcount == 751); //hsize+fp+sp-1
   assign    hreset = (hcount == 799);      //hsize+fp+sp+bp-1

   // vertical: 524 lines total
   // display 480 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 479);
   assign    vsyncon = hreset & (vcount == 490);
   assign    vsyncoff = hreset & (vcount == 492);
   assign    vreset = hreset & (vcount == 523);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank_u;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank_u;
   always @(posedge clk)
        if (mem_ready) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank_u <= next_hblank;
      hsync_u <= hsyncon ? 0 : hsyncoff ? 1 : hsync_u;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank_u <= next_vblank;
      vsync_u <= vsyncon ? 0 : vsyncoff ? 1 : vsync_u;  // active low

      blank_u <= next_vblank | (next_hblank & ˜hreset);
        end
endmodule


//video_memory: wraps zbt ram management modules into one module which
//     presents four memory ports based on pixel coordinates.

module video_memory(clk,flip,oldframe_rx0,oldframe_ry0,oldframe_rc0,oldframe_ready0,
          oldframe_rx1,oldframe_ry1,oldframe_rc1,oldframe_ready1,
          newframe_rx,newframe_ry,newframe_rc,
          newframe_wx,newframe_wy,newframe_wc,newframe_we,newframe_xbit,
          ram0_data,ram0_address,ram0_adv_ld,ram0_clk,ram0_cen_b,
          ram0_ce_b,ram0_oe_b,ram0_we_b,ram0_bwe_b,
          ram1_data,ram1_address,ram1_adv_ld,ram1_clk,ram1_cen_b,
          ram1_ce_b,ram1_oe_b,ram1_we_b,ram1_bwe_b);

     parameter BUFW = 640;
     parameter BUFH = 480;
     parameter XY_BITS = 10;

     input clk;
     input flip; //switch rams

     //oldframe read port 0 (half-rate, 5-cycle pipeline)
     input [XY_BITS-1:0] oldframe_rx0; //read x-pos
     input [XY_BITS-1:0] oldframe_ry0; //read y-pos
     output [17:0] oldframe_rc0; //read color
     output oldframe_ready0; //indicates when address will be read

     //oldframe read port 1 (half-rate, 5-cycle pipeline)
     input [XY_BITS-1:0] oldframe_rx1; //read x-pos
     input [XY_BITS-1:0] oldframe_ry1; //read y-pos
     output [17:0] oldframe_rc1; //read color
     output oldframe_ready1;

     //newframe pipelined read port (full-rate, 5-cycle pipeline)
     input [XY_BITS-2:0] newframe_rx; //missing the LSB (word)
     input [XY_BITS-1:0] newframe_ry;
     output [17:0] newframe_rc;

     //newframe pipelined write port (full-rate, 6-cycle pipeline)
     input [XY_BITS-2:0] newframe_wx; //missing the LSB (word)
     input [XY_BITS-1:0] newframe_wy;
     input [17:0] newframe_wc;
     input newframe_we;

     input newframe_xbit; //the lowest order bit of both newframe read and write x-pos


     //ram connections
     inout [35:0] ram0_data;
     output [18:0] ram0_address;
     output ram0_adv_ld;
     output ram0_clk;
     output ram0_cen_b;
     output ram0_ce_b;
     output ram0_oe_b;
     output ram0_we_b;
     output [3:0] ram0_bwe_b;

     inout [35:0] ram1_data;
     output [18:0] ram1_address;
     output ram1_adv_ld;
     output ram1_clk;
     output ram1_cen_b;
     output ram1_ce_b;
     output ram1_oe_b;
     output ram1_we_b;
     output [3:0] ram1_bwe_b;



     //zbt ram interface modules

     wire zbt0_we;
     wire [18:0] zbt0_addr;
     wire [35:0] zbt0_wd;
     wire [35:0] zbt0_rd;
     zbt_6111 zbt0(.clk(clk), .cen(1'b1), .we(zbt0_we), .addr(zbt0_addr),
             .write_data(zbt0_wd), .read_data(zbt0_rd),
            .ram_clk(ram0_clk), .ram_we_b(ram0_we_b), .ram_address(ram0_address),
            .ram_data(ram0_data), .ram_cen_b(ram0_cen_b));
     assign ram0_ce_b = 1'b0;
     assign ram0_oe_b = 1'b0;
```

```
     assign ram0_adv_ld = 1’b0;
     assign ram0_bwe_b = 4’h0;

     wire zbt1_we;
     wire [18:0] zbt1_addr;
     wire [35:0] zbt1_wd;
     wire [35:0] zbt1_rd;
     zbt_6111 zbt1(.clk(clk), .cen(1’b1), .we(zbt1_we), .addr(zbt1_addr),
             .write_data(zbt1_wd), .read_data(zbt1_rd),
            .ram_clk(ram1_clk), .ram_we_b(ram1_we_b), .ram_address(ram1_address),
            .ram_data(ram1_data), .ram_cen_b(ram1_cen_b));
     assign ram1_ce_b = 1’b0;
     assign ram1_oe_b = 1’b0;
     assign ram1_adv_ld = 1’b0;
     assign ram1_bwe_b = 4’h0;

     //memory flip
     wire oldram_we, newram_we;
     wire [18:0] oldram_addr, newram_addr;
     wire [35:0] oldram_wd, newram_wd;
     wire [35:0] oldram_rd, newram_rd;
     memory_flip memflip0(flip,oldram_we,oldram_addr,oldram_wd,oldram_rd,
          newram_we,newram_addr,newram_wd,newram_rd,
          zbt0_we,zbt0_addr,zbt0_wd,zbt0_rd,
          zbt1_we,zbt1_addr,zbt1_wd,zbt1_rd);

     //newframe memory interface
     wire [18:0] newframe_ra, newframe_wa;
     newframe_memory newmem0(.clk(clk),.read_addr(newframe_ra),.read_data(newframe_rc),
               .write_addr(newframe_wa),.write_data(newframe_wc),.write_enable(newframe_we),
               .word(newframe_xbit),
          .ram_we(newram_we),.ram_addr(newram_addr),.ram_wd(newram_wd),.ram_rd(newram_rd));

     //oldframe memory interface
     wire [19:0] oldframe_ra0, oldframe_ra1;
     oldframe_memory oldmem0(.clk(clk),.addr0(oldframe_ra0),.data0(oldframe_rc0),.ready0(oldframe_ready0),
               .addr1(oldframe_ra1),.data1(oldframe_rc1),.ready1(oldframe_ready1),
          .ram_we(oldram_we),.ram_addr(oldram_addr),.ram_wd(oldram_wd),.ram_rd(oldram_rd));


     //coordinate address translation
     assign oldframe_ra0 = oldframe_ry0 * BUFW + oldframe_rx0;
     assign oldframe_ra1 = oldframe_ry1 * BUFW + oldframe_rx1;
     assign newframe_ra = ((newframe_ry * BUFW) >> 1) + newframe_rx; //exclude last bit of x
     assign newframe_wa = ((newframe_wy * BUFW) >> 1) + newframe_wx; //exclude last bit of x

endmodule


//
// change: effectively inverted external clock, to account for delay in data return path
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

/////////////////////////////////////////////////////////////////////////////
// Ike’s simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
  ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk; // system clock
   input cen; // clock enable for gating ZBT cycles
   input we; // write enable (active HIGH)
   input [18:0] addr; // memory address
   input [35:0] write_data; // data to write
   output [35:0] read_data; // data read from memory
   output   ram_clk; // physical line to ram clock
   output   ram_we_b; // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data; // physical line to ram data
   output   ram_cen_b; // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire   ram_cen_b = ˜cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(negedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(negedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ˜we;
   assign      ram_clk = clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA’s
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1’bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111
```