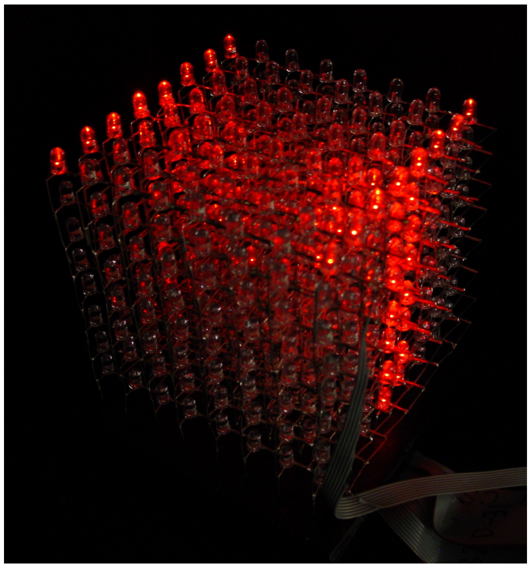
MIT 6.111: Introductory Digital Systems Laboratory, Fall 2005
Final project report

A Volumetric 3D LED display

David Wyatt
wyatt@mit.edu

Lawrence Wujanto
lwujanto@mit.edu



Abstract

A 3D volumetric LED display was designed and built, along with a number of accompanying applications. The display consisted of a cube of 512 individually controllable ultra-bright LEDs hand-soldered into a lattice, and was intended to demonstrate some of the capabilities that a true 3D display have over 2D representations of 3D objects. In addition, a renderer was built which concurrently displays an orthographic representation of the cube on an SVGA display.

Applications built include 3D Pong, 2D and 3D versions of Cellular Automata, a music visualizer, and a "trip-let" displaying the letters MIT. Different applications can be activated by means of a switch. They incorporate various peripherals, including as mouse control and audio input with CIC filtering.

The project was implemented in the hardware description language Verilog, compiled and downloaded to a Xilinx FPGA chip. Conclusions arising from this aspect of the design process, and from the rest of the project, are presented at the end of the report.

Table of Contents

List of Figures

1 Overview

1.1 Background

Many aspects of contemporary work and recreation require the effective visualisation of three-dimensional data:

- studying the structures and interactions of biochemicals
- designing a new space vehicle
- extracting a relationship from multi-variable plots in the social sciences
- diagnosing a patient's illness from non-invasive scans
- planning a new sculpture
- ...or playing the latest computer game!

Conventional methods for displaying 3D data exclusively involve flat (2D) displays that give the illusion of depth. These range from (albeit sophisticated) rendering techniques that still ultimately generate a flat image, to displays that direct a different 2D image to each eye (through head-mounted displays, coloured filter glasses or optical methods involving lenticular surfaces or parallax barriers). However, these all suffer a number of disadvantages, related to the fact that the 3D images do not occupy the same space as the observer:

- Images typically have a limited angle of view and/or are only visible with special goggles
- The observer cannot interact with the images in intuitive ways (e.g. pointing out a part of the image to a colleague with one's hand)
- The display can give erroneous impressions of relative size/scale

These shortcomings can be overcome by a true volumetric display – one where the image of a volume of space actually occupies a volume (composed of "voxels", volumetric pixels). A number of methods for constructing such displays have been attempted in the past, including the following:

- Swept volume methods ([1],[2]) – projecting light onto a moving surface such that the reflected projection appears to originate from the appropriate location within the volume
- Dot-matrix LED cubes ([3], [4], [5], [6]) – individual LEDs in a lattice form the voxels for displaying images
- Solid-state fluorescence ([2])– invisible laser beams excite specific regions within a crystal to emit visible light

1.2 Chosen system configuration

For this project a dot-matrix volumetric display was built using a lattice of 512 LEDs, arranged in an 8x8x8 cube. The number of voxels was chosen as a compromise between resolution and complexity – since the number of voxels required increases as the cube of the side length, another row of voxels would have increased the number of LEDs by over 40% for a marginal increase in display resolution.

In order to take advantage of the capabilities of the display, the project involved the production of several applications whose outputs would have been difficult, if not impossible, to display with a conventional 2D monitor.

A block diagram showing the overall decomposition of the project into subsystems and modules, is shown in Figure 1; more detailed block diagrams of each subsystem can be found in section 2.

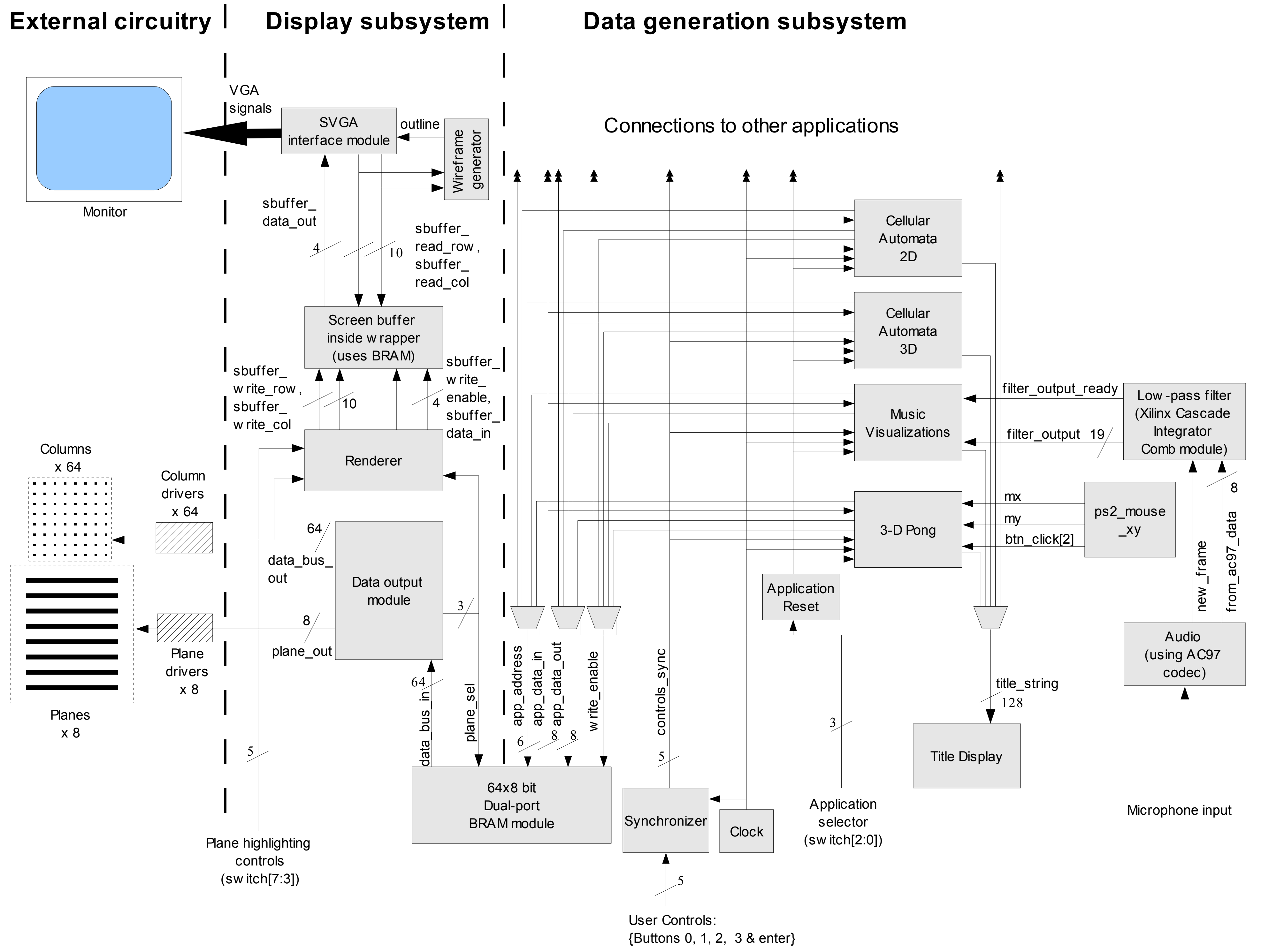


Figure 1 - Overall block diagram. The data generation subsystem on the right writes patterns to be shown on the cube into the space buffer, which the display subsystem then reads and displays on the cube and an SVGA screen.

An additional aspect of the project was the inclusion of an SVGA output to display an orthographic-projection image of the cube on a conventional PC monitor. This was primarily undertaken in case unforeseen problems made it impossible to implement the 3D display successfully, but this auxiliary output can also be used if other circumstances (such as debugging) require the data generation subsystem to be tested without the physical display hardware.

The system was implemented using the Xilinx VirtexII field-programmable gate array (FPGA) development board developed for the 6.111 laboratory class at MIT, shown in Figure 2. The board contains a small number of switches and LEDs that were used as signals for some of the modules, as well as connections to other input and output devices (in particular, the PS/2 mouse input and monitor output).

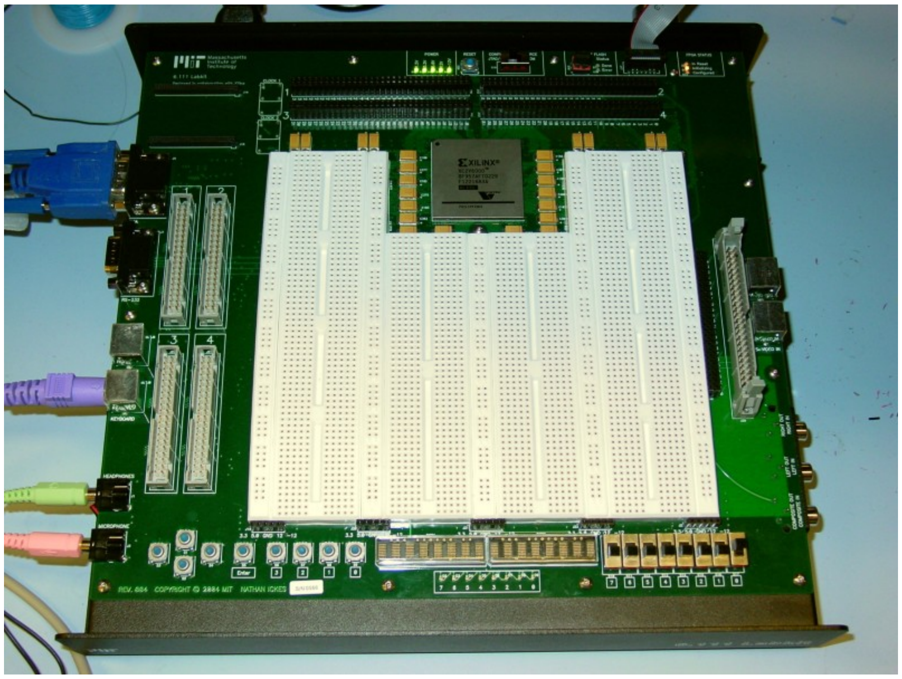


Figure 2 - The FPGA development board on which the project was implemented. The switches, buttons and LEDs are located at the front of the board, while the large FPGA chip is visible to the rear of the breadboard area.

The remainder of this report describes the design and implementation of the project; the computer-aided testing and debugging phase; and the conclusions drawn from the experience.

2 Technical description

Verilog code for all the modules implemented in this project is attached in the appendices to this report.

2.1 Data generation subsystem

A number of applications have been written for the volumetric display. Switches 0 to 2 are used to select between the different applications, allowing for up to 8 applications. These switches set the multiplexors in the data generation subsystem to select the appropriate app_address, app_data_out, write_enable and title_string lines for the currently active application.

The 512 bit "space buffer" serves as an interface between the data generation display subsystems. Applications have read and write access via port B of this dual port BRAM, and update this buffer constantly with the present state of the active application.

Figure 1 shows the interconnection between applications in the subsystem, while figure 3 shows more details on the implementation of each application.

2.1.i Application Reset

In order to make applications reset themselves whenever they are selected, an Application Reset module drives the reset line of all applications, sending a signal high pulse for one clock cycle whenever the application selector switches are toggled. This is used in applications such as 2D cellular automata to reset the playing field and await user input.

2.1.ii 3D Pong

A natural extension of the popular two-dimensional game, 3D pong is a one-player game which involves defending the bottom surface of the cube from being struck by a bouncing "ball" by manipulating a paddle which deflects the ball. The paddle is controlled using a PS/2 interface mouse.

The PS/2 mouse driver module was used with X & Y motion limited from 0 (7'b000_0000) to 95 (7'b101_1111). The 4 least significant bits of the mouse coordinates were ignored so as to decrease the sensitivity of the mouse, while the next 3 bits were used for the location of the paddle.

Since a bouncing ball that travels with equal magnitudes of velocity along each axis (as was the case for the 2D pong implemented for Lab 4 in 6.111) would be rather boring in that it would follow the same path round the cube every time, we specified that the velocity of the ball should be altered during every bounce off the paddle, as a function of where it strikes the paddle.

To allow "intermediate" angles of motion that are not along one of the principle axes/diagonals, the location of the ball is stored using a 6 bit register for each of the three axes, and then truncated to 3 bits wide for determining the ball's location in the cube.

A set of registers store data governing the velocity and location of the ball. This data is updated regularly to shift the ball; ensure that it bounces off walls; and change its velocity upon contact with the paddle. At the same time, the registers are read off and used to update the contents of the space buffer to display the paddle and ball on the playing field. An 'inv' register was included in the game that is high for a short period after a collision, causing the state of the LEDs to invert. This indicates when a player has failed to catch the ball.

2.1.iii 2D Cellular Automata

Cellular automata, a popular form of which is Game of Life, involves a grid of cells which may each be either "living" or "dead", and a set of rules which dictates the next state of each cell, as a function of the states of its neighbouring cells. This application has a number of predefined initial states that are loaded onto the top plane of the cube when one of buttons 0 to 3 are pressed. At a frequency of 1Hz, the contents of the top plane are evolved using the standard Game of Life rules set[1], while previous states of the plane are propagated downwards in the cube so as to show the time evolution of the game.

The state of the entire cube was stored using a 512 bit register. The propagation of previous states could be done quite easily by a single cube[511:64] <= cube[447:0] command. Evolving the state of the plane was a little more complicated, as this involved summing the number of neighbouring live cells for each cell, and then deciding on the next state of each cell as a function of the value of this sum. This was done by incrementing through all possible values of an 11-bit register (state_counter) to cycle through every operation that had to be done. This could be thought of as a 2 x 64 x 16 state FSM, allowing for 16 operations on each of the 64 LEDs on a plane, and doubling this to allow more operations which update the cube contents to the space buffer.

The first 64 x 16 states are used for computing the next state of the top plane, with bits [9:4] used to represent the active cell, and bits [3:0] to specify 16 operations for each cell, including resetting the cell counter, adding the contents of each of the 8 neighbouring cells, and applying the rule set to determine the next state of the active cell.

The next 64 x 16 states allow for updating the state buffer with the new state of the cube, with many leftover unused states. Note that when a new pattern is requested, the state jumps straight to this half of the state sequence so as to display the new state before evolving the cube state.

2.1.iv 3D Cellular Automata

The same idea from 2D Cellular Automata can be extrapolated into the third dimension by summing all 26 neighbouring cells for each of the 512 LEDs in the LED cube. The rule set we used was that 2 neighbouring live cells let life persist, while 3 or 4 neighbouring live cells cause life to be born, otherwise the cell “dies”. A 15-bit state counter was required to allow for 2 x 512 x 32 states.

It has been suggested that a ROM could be used to store all possible combinations of the neighbouring cells and the cell itself, and return the next state of the cell. This would certainly be a good idea for 2D cellular automata in reducing the number of operations required. However, this may be infeasible for 3D cellular automata due to the large number of possible states that 27 cells may have, and hence the large ROM size.

1 2 neighbouring live cells – life persists; 3 neighbouring cells – life is born; otherwise cell “dies”

Application block diagrams

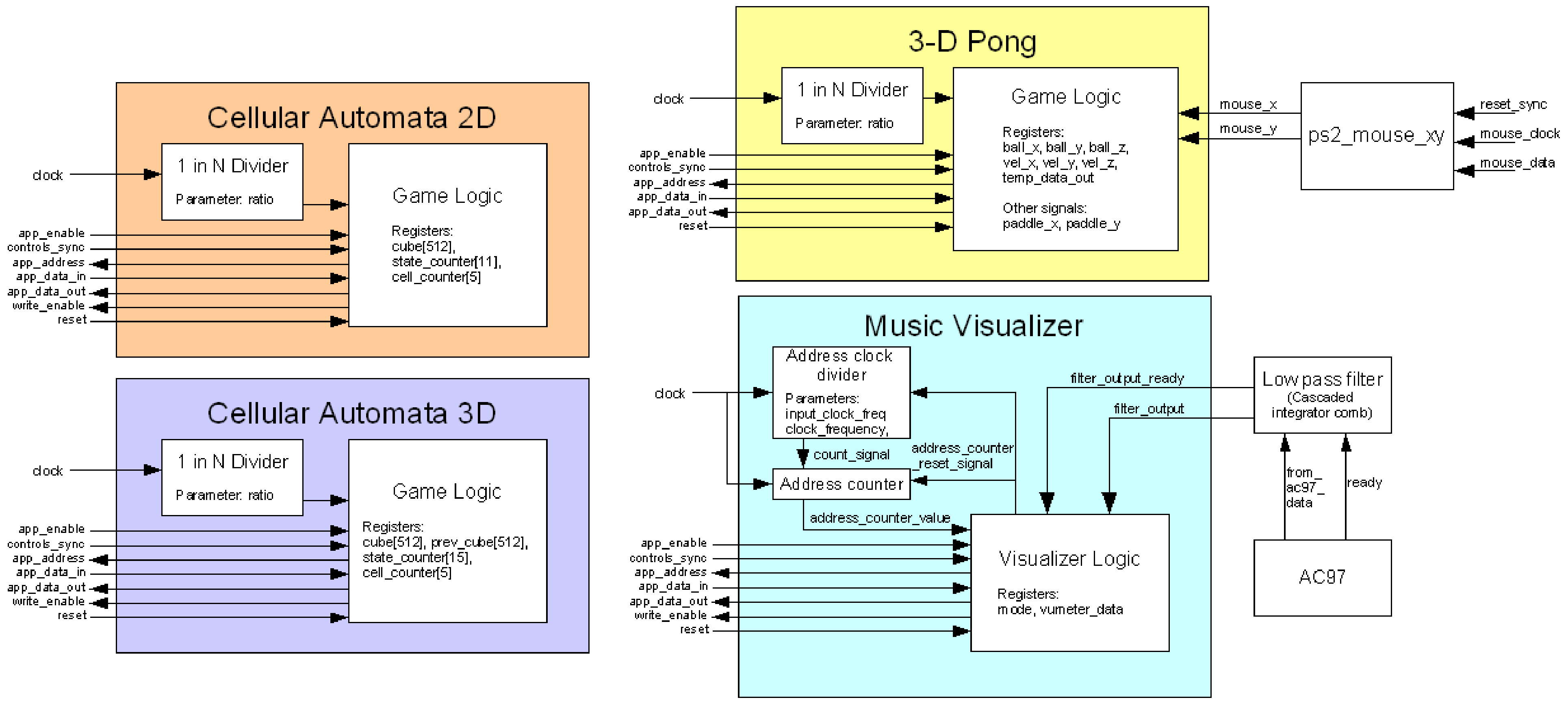


Figure 3 - Layout of various application modules.

2.1.v "MIT" trip-let

As a demonstration of the 3D display capability of the volumetric display, we created an application which displays the letters M, I & T from each of the principal axes, inspired by the cover of Douglas Hofstadter's *Gödel, Escher, Bach*. This was done by storing 64-bit parameters which describe 8 x 8 "font grids" for these letters, and using Boolean AND operators on elements of these grids, referenced using different pairs of the cube's three coordinates.

```
app_address <= next_address;
app_data_out[0] <= M[next_address] && I[{next_address[2:0], 3'b000}] && T[{next_address[5:3], 3'b000}];
app_data_out[1] <= M[next_address] && I[{next_address[2:0], 3'b001}] && T[{next_address[5:3], 3'b001}];
```
... (continued for all 8 bits of app_data_out)

next_address[5:3] and next_address[2:0] effectively represent the z and y axes of the cube respectively. The 8 bits of app_data_out contain the contents of a row of LEDs along the x axis. This idea could be extended to allow any combination of letters to be displayed, perhaps using the PS/2 keyboard to input letters.

In this module, we also experimented with varying the intensity of the LEDs by cycling the contents of each activated LED through the contents of different 8 bit strings. For example, this included an "all on" string, an alternating on/off string, and an "all off" string. The different number of "on" states in the string changed the brightness of the LED. A different string was selected as a function of time (using the top few bits of a large counter), and the z coordinate in the cube. The result of this was a wave-like pattern that propagated down the z axis of the cube.

2.1.vi Music visualiser

This module uses audio input to create 3D patterns within the cube.

The audio input is first converted to digital form by a National Semiconductor LM4550 audio codec (compatible with the AC97 standard), controlled by a wrapper module supplied by [9]. It is then low-pass filtered by a Cascaded Integrator Comb filter produced by the Xilinx Coregen application, configured to downsample by a ratio of 1920 to 1 (thus giving a 25Hz output since new samples are supplied by the AC97 at 48KHz) with 1 stage and a differential delay of 2. These parameters were arrived at through experimentation and simulation of the filter's frequency response using Microsoft Excel, to determine a suitable cut-off frequency.

When ready, the top 3 bits of the filter's 19-bit output (plus 4, to convert signed 2's complement into unsigned format) are fed into the first of a queue of 8 vumeter_data registers, the contents of the rest of which propagate down a register. The contents of these registers then generate the output to be shown on the cube according to the currently-selected mode:

- In mode 0, "Linear", the cube displays an oscilloscope that has been extruded in the third dimension; the contents of each register determines the height of the "oscilloscope trace" in the corresponding vertical plane.
- In mode 1, "Ripples", the contents of the registers determine the left-right positions of concentric squares of a vertical plane in the cube, with the newest register controlling the central square; the aim was to give the impression of ripples spreading out from the centre of the plane, though the pattern could also be regarded as an oscilloscope trace rotated about a vertical axis through its left-hand end.

Switching between modes is controlled by pressing the numbered pushbuttons on the labkit board. Writing data to the space buffer is accomplished using a 6-bit counter incremented at 65KHz and reset every time a new sample is received from the filter (to keep the space buffer writing in rough synchrony with the changes in the underlying data). The write address sent to the space buffer is the value of the counter, and the data is determined according to the mode and the values of the vu_meter_data registers.

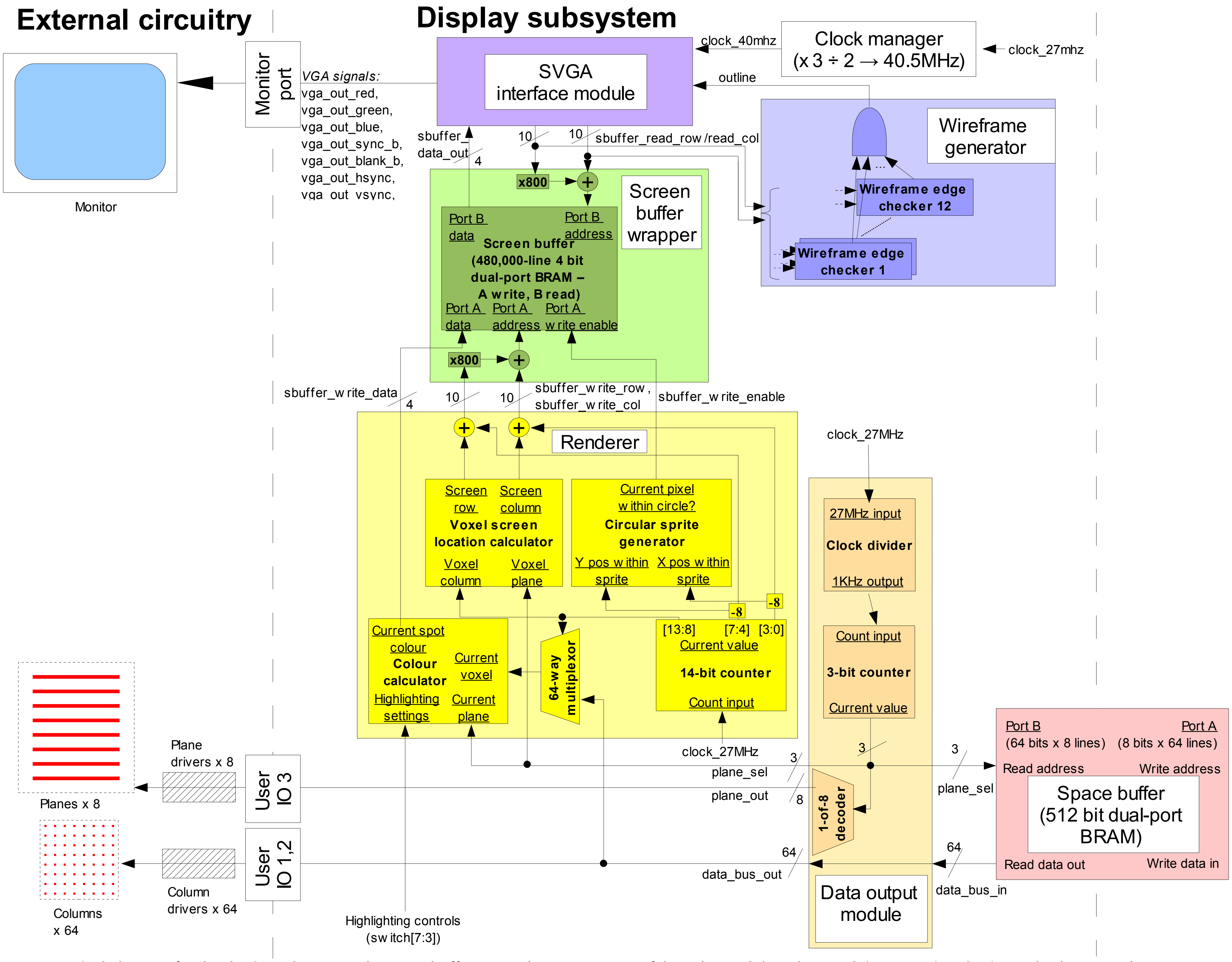


Figure 4 - Block diagram for the display subsystem. The space buffer stores the current state of the cube, and the other modules control its display on both output devices.

2.2 *Display subsystem*

This subsystem:

- Transfers arbitrary data representing spatial patterns from the “space buffer” RAM to an 8x8x8 matrix of LEDs
- Has a refresh rate of 125Hz to exploit persistence of vision
- Produces an auxiliary SVGA output which displays an orthographic image of display state at 800 x 600 resolution, with controls to highlight specific planes of the cube

See Figure 4 for a block diagram of this subsystem.

2.2.i Voxel addressing

Since there are only 192 output pins available on the labkit and 512 LEDs in the display, it is not possible to address each LED individually; thus a passive matrix multiplexing system is used. Each horizontal plane of 64 LEDs has a unique “enable” pin, corresponding to a row in a traditional (2D) passive matrix display, while the 8 LEDs in each vertical column correspond to the columns of a conventional display. Thus the LEDs that are lit are at the intersection of enabled planes and columns. This requires 64 output pins for the columns plus 8 for the planes.

With this pattern of interconnection, either 8 or 64 LEDs could be driven at once; the complete cube could be lit by cycling through each column or each plane in turn. In order to achieve the maximum the duty cycle for the LEDs and thus enhance their visibility, it was chosen to enable a single plane at a time while driving the columns with the correct data for that plane.

2.2.ii Data output module

The data output module implements the above method for driving the cube's voxels. It contains a 3-bit counter that is incremented at 1kHz and is used as the address input for the B (read) side of the space buffer, which then outputs 64 bits on the 64-bit-wide data_bus_in wire, corresponding to the voxels in the currently selected plane. In the present implementation the data_bus_in wire is connected directly to the data_bus_out wire, which drives the output pins on user ports 1 and 2 on the labkit, but any processing necessary in future development (such as low-level pulse width modulation for adjusting the output brightness levels) could be incorporated at this stage.

This gives an overall refresh rate for the cube of 125Hz, quite sufficient for persistence of vision [7]. It would be possible to refresh the cube more frequently, but it was thought unnecessary; also, a reduction in brightness of the LEDs was observed if they were driven at significantly higher frequencies.

2.2.iii Renderer

This module performs the orthographic projection from 3D to the 2D SVGA-resolution screen and writes the appropriate data to the screen buffer. It was designed to use the signals from the data output module without requiring any additional control signals so that the latter could be implemented in a straightforward way.

In order to generate the projection, it has four major components (shown as submodules on the block diagram but mostly implemented as code within the main module):

- A 14-bit counter, clocked by the main system clock at 27MHz – the value of this counter can be regarded as the state of a finite state machine controlling the module. The top 6 bits of the counter are treated as the number of the voxel whose projection is to be generated, voxel_column.

- The screen location calculator – this takes as input the current voxel column and the current plane number (input to the renderer module from the data output module), and generates the screen co-ordinates of the centre of its projection as follows (where the capitalised terms are constant parameters):
  (screen column) = OFFSET_X + D*voxel_column[2:0] + D_SIN_THETA*(7 - voxel_column[5:3])

(screen row) = OFFSET_Y + D*(7 - voxel_plane) + D_COS_THETA*(7 – voxel_column[5:3])

A diagram showing the mapping from voxel_column and voxel_plane (specifying the (x,y,z) co-ordinates of a voxel) to the 2D screen co-ordinates can be seen in Figure 5.

- The colour calculator – this uses voxel_column to select the data_bus_out signal corresponding to the current voxel, and outputs red if it is on and black if it is off (the shades are configurable via constant parameters). In addition, if highlighting is enabled (that is, if switch[7] is on) it dims the shade of red output when the current plane is not the plane to be highlighted (as set up on switch[6:4]), either in terms of horizontal planes (switch[3] = 0) or vertical planes (switch[3] = 1).

- The circular sprite generator – this takes the co-ordinates of the current point within the current voxel's sprite, extracted from the least significant 8 bits of the counter's value, and calculates whether they represent a point within a parameterised radius of the sprite's centre. If so, the write enable to the screen buffer is set high.

The net result of the operation of these four components is that the address inputs to the screen buffer wrapper cycle through the projections of each voxel in turn within the current plane, performing a 16x16 pixel raster scan around the centre of the projection. However, the write enable to the screen buffer is only high when the scan point is within a certain radius of the centre point, generating circles of the appropriate colour in the screen buffer memory.

The constant parameters used by the renderer to determine the scale, location and colours of the projected image can be taken from the instantiating module, allowing for easy reconfiguration of the output.

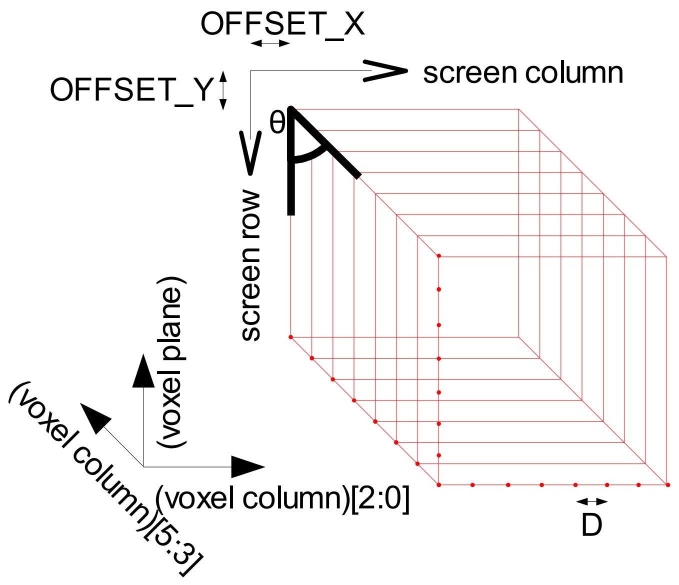


Figure 5 - Diagram to show mapping from 3D (x,y,z) co-ordinates to 2D screen rows and columns.

2.2.iv Screen buffer wrapper

The renderer and SVGA output modules output pixel co-ordinates on an SVGA screen as 10-bit numbers. These could be simple concatenated to produce the memory location to write; however, such an approach would be inefficient (since only 800x600 pixels out of the 1024x1024 would ever be used) and would mean that the screen buffer would be too large to be implemented using the BRAMs on the FPGA chip. Thus, the 480,000-line 4-bit dual-port BRAM comprising the screen buffer was surrounded by a wrapper module to translate the two 10-bit input co-ordinates into memory addresses in a by performing a constant multiplication and an addition, on both the write (port A) and read (port B) sides of the memory.

2.2.v Wireframe generator

This module takes as input the co-ordinates of the current screen pixel from the SVGA interface module and produces a single-bit signal, outline, that indicates whether the pixel falls on one of the lines making up a wireframe image of the cube. Inside the module this detection is implemented as a series of twelve if statements corresponding to the twelve edges of the cube, parameterised from the labkit module to allow easy compile-time adjustment of the size and location of the image.

At present the projection angle is fixed to be 45° due to the algorithm chosen for drawing the oblique lines at the corners of the cube; use of a more general algorithm, such as the Bresenham line drawing algorithm [8], would remove this restriction.

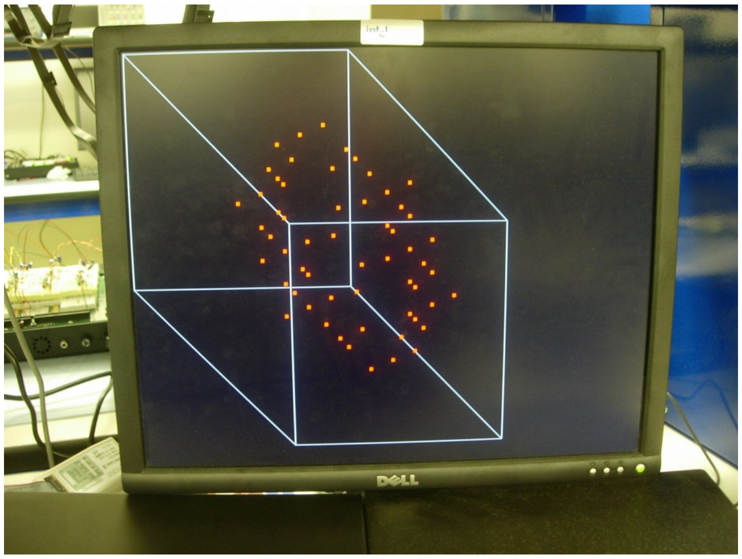


Figure 6 - SVGA display output, with 3D cellular automata as the active application.

2.2.vi SVGA interface module

This module, based heavily on the one provided by [9], generates the required sync and blanking signals for SVGA (800x600) display at a refresh rate of 60Hz. It is clocked at 40.5MHz, the nearest frequency to the ideal 40MHz that could be synthesised by a Digital Clock Manager (a delay-locked-loop, here set up to output the input frequency x 3 ÷ 2). As well as outputting the VGA control signals, it produces the signals hcount and vcount – 10-bit signals specifying the point on the screen whose data is being output at that time. These signals are sent to the read side of the screen buffer wrapper and to the wireframe generator, which return (respectively) signals representing the presence of a voxel sprite and/or a a wireframe outline pixel at that location; logic in the main labkit module ensures that voxel sprites receive priority if both are present at the same location.

2.3 Physical display hardware

2.3.i Mechanical design and construction

As described in Section 2.2.i, the LEDs were connected in a passive matrix in which:

- All LED cathodes within a horizontal plane are wired together
- All anodes within a vertical column are wired together

With this polarity, the selected cathode plane is driven low while all others are driven high, and simultaneously the data to be displayed on that plane is presented on the columns in uninverted logic (i.e. if the column is high the corresponding LED will be on).

For simplicity when using 5mm LEDs, and to reduce obscuring of the display by a support frame, the structure that holds the LEDs in the lattice was built by soldering together the LEDs' own wires and some extra strands of solid-core wire whose insulation had been removed. The resulting lattice has a 15mm pitch, due to the lengths of the LED leads. Diagrams of the arrangement are shown in Figure 7 and Figure 8.

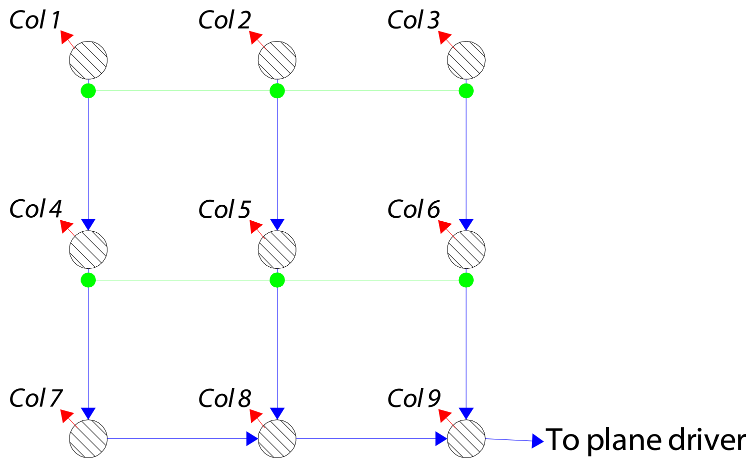


Figure 7 - Top view of wiring within a horizontal plane.

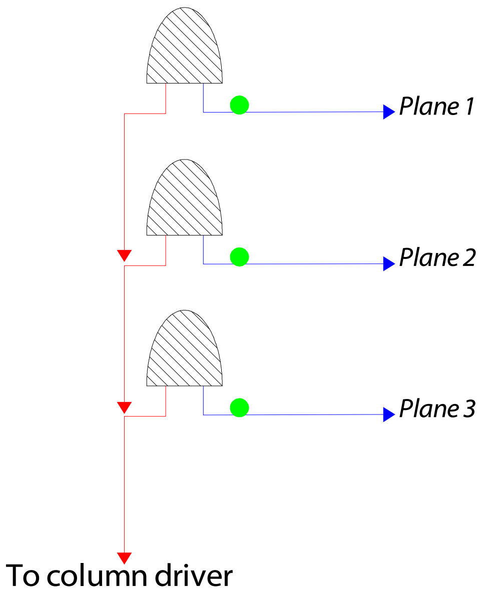


Figure 8 - Side view of wiring within a vertical column.

The LED cathodes (blue) are joined together in rows within each plane, with each row joined to the next at one end; supplementary wires (green) provide redundant cathode connections and mechanical stability. The LED anodes (red) are connected together in columns.

This design permitted easy, if repetitive, assembly as follows:

1. Each LED's leads were bent according to the diagrams in Figure 7 and Figure 8.
2. The LEDs were soldered together in planes of 64 (wiring up the cathodes), using a simple jig to hold them in place while soldering (Figure 9).
3. The planes were then stacked into a cube (soldering together the anodes), starting from the topmost layer and working downwards (Figure 10).
4. Finally, a wooden baseboard was attached to the cube for addition mechanical strength, and the plane and column connection wires were attached.

The completed cube, shown in Figure 11, contains 1472 hand-soldered joints, and took approximately 10 hours to construct. During the initial periods of LED lead-shaping before cube construction began, despite the fact that no count was kept, exactly 508 LEDs were processed of which only 1 was incorrectly bent.

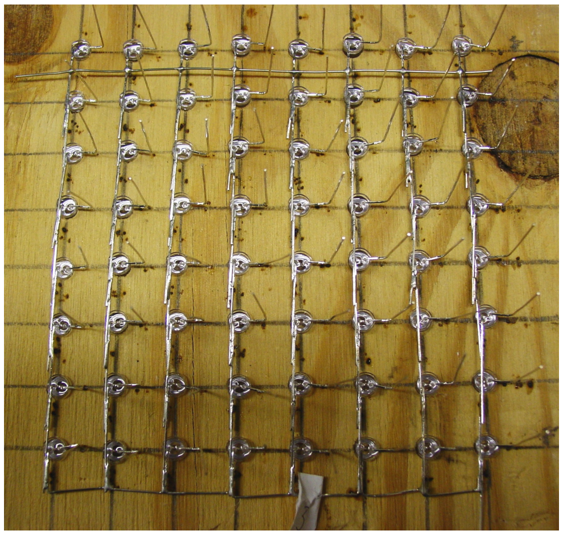


Figure 9 - A plane of LEDs laid out on the jig. The LEDs' cathodes have been soldered together, and the first stabilising cross-wire has been soldered in place.

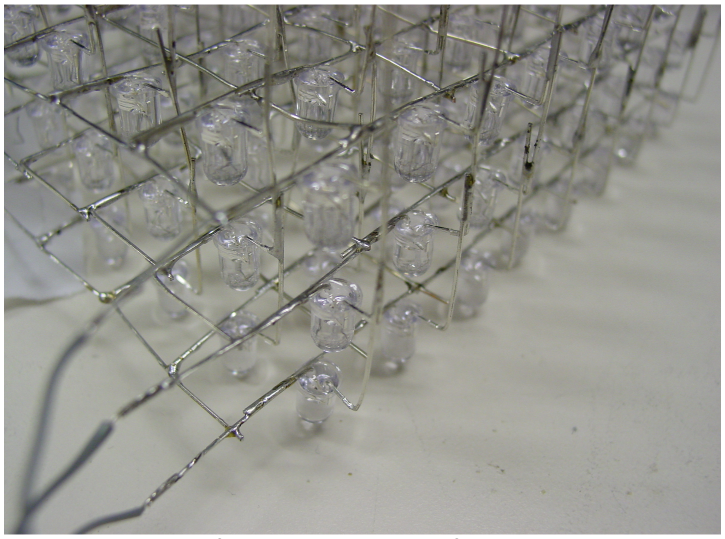


Figure 10 - The first three planes of LEDs assembled. The corner closest to the camera is the corner at which the cathode wires for each plane were connected, and the way in which the anodes in each plane are soldered to those in the plane below is also evident.

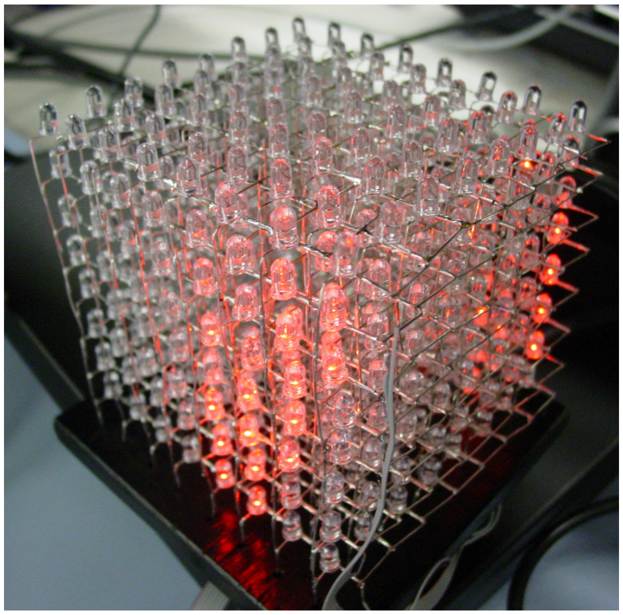


Figure 11 - The completed LED cube. The current pattern was generated by the 2D Cellular Automaton module; the cathode connection corner, voxel column 0, is in the foreground.

2.3.ii Driver electronics and power issues

In order to give an acceptable level of brightness at a duty cycle of 1 in 8, high-brightness LEDs with a specified light intensity of 8000mcd and a viewing angle of 20° total were used. They were specified to draw 30mA at 2.7V, and pass less than 30μA when reverse biased. The FPGA output pins can only handle up to 24mA, so driver circuitry was needed whose maximum ratings were calculated as follows.

- Per plane:
  - when active, at maximum 64 LEDs will be on – *plane driver must sink 1.92A*
  - when inactive, at maximum 64 LEDs will be reverse biased – *plane driver must source 1.92mA*
- Per column:
  - when high, 1 LED will be on and 7 off – *column driver must source 30mA*
  - when low, 1 LED will be off and 7 reverse biased – *column driver must sink 0.21mA*

Eight MIC4429 6A inverting MOSFET drivers were thus used for each plane, driven directly by FPGA output pins. The column driver circuitry used 7404 TTL inverters along with 68Ω current-limiting resistors. The schematic for the driver circuitry is shown in Figure 12.

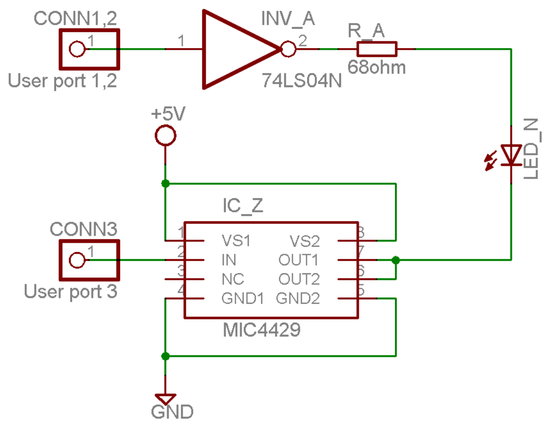


Figure 12 - Driver circuitry for a representative LED. LED N is at the intersection of column A (between 0 and 63) and plane Z (between 0 and 7).

Since only one plane is be active at once, the current supply requirement for the entire system were expected to be 2A at 5V (in addition to the power requirement of the FPGA and peripherals). This was more than could be supplied by the FPGA board, but well within the capability of a standard laboratory bench power supply. In fact current consumption was limited to around 0.3A due to unanticipated voltage drops within the driver circuitry, but this did not appear to hinder brightness.

3 Implementation, testing and debugging

3.1 Design flow

The submodules specified in Section 2 were implemented in Verilog, an industry-standard hardware description language, using the Xilinx Integrated Software Environment (ISE) v6 toolchain. The initial steps involved writing the modules by hand using ISE's built-in text editor and project-based file organisation system; a bitstream configuration file for a VirtexII XC2V6000 field-programmable gate array (FPGA) chip was then generated from within ISE. This involved the following sequence of steps (automated by the software):

1. Compiling Verilog to a netlist of primitive gates and circuit elements (using the synthesis tool XST).
2. Mapping the netlist onto the resources available in the FPGA.
3. Placing and routing the components of the circuit for maximum performance.

The bitstream file was downloaded to the FPGA through a JTAG interface. The resulting system behaviour was compared with expectation, leading to modifications of the source files and further compilation cycles.

3.2 Debugging

In general, Verilog modules were tested in the hardware rather than by simulating them on a PC. A number of factors contributed to the decision to use this method for fault-finding:

- It was generally simpler to test on the FPGA than in software, once a body of known functional code had been achieved. In the initial stages of the project, priority was given to establishing the shared interface between the subsystems (the space buffer in particular) and creating the display subsystem before embarking on the construction of the physical cube and the programming of the applications. This meant that the two members of the team could work independently, each with access to a functioning (if minimal) system; thus, the patterns produced by an application under construction could be viewed on the SVGA output, and by the same token the partially-complete cube could be tested with a pregenerated test pattern to check its functioning.

- There were usually sufficient inputs and outputs available on the labkit to use those for debugging, rather than having to use a test bench waveform.

- Signals usually changed states sufficiently slowly that it was possible to perform the functions of a logic analyser/oscilloscope by eye.

- The coding style of the members of the team was such that mistakes were usually fairly straightforward slips that could be found without extensive experimentation – indeed, the errors were usually obvious after a little consideration of the behaviour of the output.

By good fortune there were no faults or mistakes during the physical construction of the cube, so debugging as such was not carried out. Careful testing took place, however, to ensure that any potential faults were detected early (as it would be almost impossible to replace a faulty LED in the centre of the cube): each plane's 64 LEDs were tested individually after soldering, and the complete stack was tested after the new plane had been attached. The driver circuitry was similarly checked for function with a current-limited power supply before connecting it to the FPGA for the first time.

3.3 Division of Labour

The work was divided between the two partners as follows.

- Lawrence Wujanto:
    - 3D Pong
    - 2D Cellular Automaton
    - 3D Cellular Automaton
    - MIT Trip-let
- David Wyatt:
    - Display subsystem
    - Music visualiser
    - Display hardware construction

4 Conclusions

4.1 Results from the project

A 512-voxel volumetric display was successfully constructed and interfaced to a number of different applications running in an FPGA. The control logic of the applications uses memories, digital signal processing units and human interface devices, and contains multiple finite-state-machines executing different tasks in parallel. All the features specified in the checklist were implemented successfully, as well as completion of or significant progress being made towards some additional features (the MIT Trip-let and an application to rotate arbitrary shapes in 3 dimensions, which unfortunately was not completed by the end of the project).

Undertaking this project contributed greatly to the team's knowledge of and familiarity with contemporary methods of digital logic design, prototyping and debugging. It also gave useful practice in project planning, teamwork and dividing an engineering project into sections for parallel implementation. Communication skills were exercised in preparing reports at multiple stages, from a proposal abstract through to this document, and presenting the design concept for review at an early stage. Lastly, both team members greatly enjoyed the experience and are very grateful for the opportunity to take part.

4.2 Possibilities for future expansion

- Low-level (as opposed to application-level) pulse width modulation brightness control of the LEDs, with corresponding intensity variations on the SVGA output
- Implementation of the Bresenham line drawing algorithm to allow projection angles other than 45°.
- Display of 3D data stored on a CompactFlash card - may be used as initial conditions for cellular automata
- True 3D rendering (rather than orthographic projection) of the cube on SVGA output, rotatable in real time by user
- Modification of the cube to increase resolution/enhance visibility – use a larger lattice spacing or smaller LEDs (ideally SMT, but this would require a new construction technique)

5 Components used

- 6.111 class labkit (designed by Nathan Ickes (MIT) and Xilinx)
- Windows PC and Xilinx ISE software for development (monitor can also be used for displaying auxilary SVGA output)
- 512 ultra-bright LEDs: 5mm, 8000mcd, 20° viewing angle (source: www.ledshoppe.com)
- Wire: solid-core, multi-core ribbon cable
- Baseboard: plywood, 10mm thick, 135mm x 135mm
- Bench power supply: capable of supplying 1A at 5V
- 2 breadboards
- LED driver circuitry: 12x 7404 hex inverter chips, 64x 68Ω resistors, 8x MIC4429 inverting 6A MOSFET driver chips (source: www.digikey.com)

6 References

[1] Actuality Systems, "Perspecta 3d display", http://www.actuality-systems.com/ (accessed 3 November 2005)

[2] Felix 3D, "Felix 2"/"solidFELIX", http://www.felix3d.com/ (accessed 3 November 2005)

[3] James Clar, "3d display cube – white", http://www.jamesclar.com/product/2005/3dcubewhite/ (accessed 3 November 2005)

[4] Network Wizards, "Cubatron", http://nw.com/nw/projects/cubatron/ (accessed 3 November 2005)

[5] Todd Holoubeck, "LED cube", http://www.toddholoubek.com/projects/ledpage/ (accessed 3 November 2005)

[6] Chris Lomont, "LED Cube", http://www.lomont.org/Projects/LEDCube/LEDCube.php (accessed 3 November 2005)

[7] Wikipedia, "Persistence of Vision", http://en.wikipedia.org/wiki/Persistence_of_Vision (accessed 3 November 2005)

[8] Colin Flanagan, "The Bresenham Line-Drawing Algorithm", http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html (accessed 12 December 2005)

[9] 6.111 course staff, MIT, sample Verilog files, http://web.mit.edu/6.111/www/f2005/handouts.html#samplecode (accessed 15 November 2005)

Appendix A: Labkit.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 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;

   inout  mouse_clock, mouse_data;
   input  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

// VGA Output
/*assign vga_out_red = 10'h0;
assign vga_out_green = 10'h0;
assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;*/

// 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
   // 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;
   // ********************************************************************
   // Generic inputs and outputs

   // Reset switch = enter button
   wire reset_sync;
   debounce reset_debouncer(.reset(1'b0), .clock(clock_27mhz), .noisy(~button_enter),
.clean(reset_sync));

   // Debug bus - wired to LEDs (invertedly) and low byte of Analyser 1 data port
   wire [7:0] debug_bus;
   assign led[7:0] = ~debug_bus[7:0];
   assign analyzer1_data[15:0] = {8'b0, debug_bus};

   // ********************************************************************
   // Space buffer, with inputs and outputs

   // Data generation subsystem wires (port A)
   wire[5:0] app_address;
   wire[7:0] app_data_in, app_data_out;
   wire write_enable;

   // Display subsystem wires (port B)
   wire[2:0] plane_sel;
   wire[63:0] data_bus_in;

   space_buffer space_buffer(.addra(app_address), .dina(app_data_out), .douta(app_data_in),
.wea(write_enable), .clka(clock_27mhz),
      .addrb(plane_sel), .doutb(data_bus_in), .clkb(clock_27mhz));

   // *******************************************************************
   // Modules for data generation subsystem
   // Last modified by David Wyatt
   // 2005/12/11

   // Labkit button inputs
   wire [4:0] controls_sync;
   synchronize synchronize0(.clk(clock_27mhz), .in(~button0), .out(controls_sync[0]));
   synchronize synchronize1(.clk(clock_27mhz), .in(~button1), .out(controls_sync[1]));
   synchronize synchronize2(.clk(clock_27mhz), .in(~button2), .out(controls_sync[2]));
   synchronize synchronize3(.clk(clock_27mhz), .in(~button3), .out(controls_sync[3]));
   synchronize synchronize_enter(.clk(clock_27mhz), .in(~button_enter),
.out(controls_sync[4]));

   // PS/2 Mouse
   wire [2:0] mouse_btn;
   wire [11:0] mouse_x, mouse_y;
   ps2_mouse_xy my_mouse(.clk(clock_27mhz), .reset(reset_sync), .ps2_clk(mouse_clock),
.ps2_data(mouse_data), .mx(mouse_x), .my(mouse_y),
      .btn_click(mouse_btn));

   // AC97 Audio
   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;
   audio a(clock_27mhz, ac97_reset, from_ac97_data, to_ac97_data, ready,
        audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);
   // Loopback input AC97 audio to output
   assign to_ac97_data = from_ac97_data;
   // detect clock cycle when READY goes 0 -> 1
   // f(READY) = 48khz
   wire new_frame;
   reg old_ready;
   always @ (posedge clock_27mhz) old_ready <= reset_sync ? 0 : ready;
   assign new_frame = ready & ~old_ready;

   // Instantiate the cascade integrator comb filter, a Xilinx Coregen module
   wire[18:0] filter_output;     // Filter output - 19 bits wide, according to Coregen...
   wire filter_output_ready;     // Filter output ready signal
   low_pass_filter low_pass_filter1(.DIN(from_ac97_data), .ND(new_frame), .CLK(clock_27mhz),
.DOUT(filter_output), .RDY(filter_output_ready));

   // ASCII string display
   wire [127:0] title;
   display_string my_display(.reset(reset_sync), .clock_27mhz(clock_27mhz),
.string_data(title), .disp_blank(disp_blank),
      .disp_clock(disp_clock), .disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
.disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));

   // Application selection (including Muxes to select appropriate inputs)

   // MUX connectors
   wire [5:0] app_address0, app_address1, app_address2, app_address3, app_address4,
app_address5;
   wire [7:0] app_data_in0, app_data_in1, app_data_in2, app_data_in3, app_data_in4,
app_data_in5;
   wire [7:0] app_data_out0, app_data_out1, app_data_out2, app_data_out3, app_data_out4,
app_data_out5;
   wire write_enable0, write_enable1, write_enable2, write_enable3, write_enable4,
write_enable5;
   wire [127:0] title0, title1, title2, title3, title4, title5;

   // MUXes
   mux_8input_6bit app_address_mux(.in0(app_address0), .in1(app_address1),
.in2(app_address2), .in3(app_address3), .in4(app_address4),
      .in5(app_address5), .in6(6'b0), .in7(6'b0), .selector(switch[2:0]),
.out(app_address));
   mux_8input_8bit app_data_in_mux(.in0(app_data_in0), .in1(app_data_in1),
.in2(app_data_in2), .in3(app_data_in3), .in4(app_data_in4),
      .in5(app_data_in5), .in6(8'b0), .in7(8'b0), .selector(switch[2:0]),
.out(app_data_in));
   mux_8input_8bit app_data_out_mux(.in0(app_data_out0), .in1(app_data_out1),
.in2(app_data_out2), .in3(app_data_out3), .in4(app_data_out4),
      .in5(app_data_out5), .in6(8'b0), .in7(8'b0), .selector(switch[2:0]),
.out(app_data_out));
   mux_8input_1bit write_enable_mux(.in0(write_enable0), .in1(write_enable1),
.in2(write_enable2), .in3(write_enable3), .in4(write_enable4),
      .in5(write_enable5), .in6(1'b0), .in7(1'b0), .selector(switch[2:0]),
.out(write_enable));
   mux_8input_128bit title_mux(.in0(title0), .in1(title1), .in2(title2), .in3(title3),
.in4(title4),
      .in5(title5), .in6(128'b0), .in7(128'b0), .selector(switch[2:0]), .out(title));

   // Application reset controller
   wire reset;
   application_reset application_reset1(.selector(switch[2:0]), .clock(clock_27mhz),
.reset(reset));

   // Applications
   // ------------
   // 0: 3D Pong
   pong pong1(.app_address(app_address0), .app_data_in(app_data_in0),
.app_data_out(app_data_out0), .write_enable(write_enable0),
      .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title0),
.pos_x(mouse_x[6:4]), .pos_y(mouse_y[6:4]), .mclick(mouse_btn));

   // 1: 2D Game-of-Life cellular automaton
   cell_aut_2D cell_aut_2D1(.app_address(app_address1), .app_data_in(app_data_in1),
.app_data_out(app_data_out1), .write_enable(write_enable1),
      .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title1));

   // 2: 3D cellular automaton
   cell_aut_3D cell_aut_3D1(.app_address(app_address2), .app_data_in(app_data_in2),
.app_data_out(app_data_out2), .write_enable(write_enable2),
      .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title2));

   // 3: Music visualiser
   music_visualizer music_visualizer1(.app_address(app_address3),
.app_data_in(app_data_in3), .app_data_out(app_data_out3), .write_enable(write_enable3),
      .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title3),
.filter_output_ready(filter_output_ready), .filter_output(filter_output));

   // 4: MIT triple-letter cube
   mit mit1(.app_address(app_address4), .app_data_in(app_data_in4),
.app_data_out(app_data_out4), .write_enable(write_enable4),
     .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title4));

   // 5: 3D spinning shapes
   spinning_shapes spinning_shapes1(.app_address(app_address5), .app_data_in(app_data_in5),
.app_data_out(app_data_out5), .write_enable(write_enable5),
      .controls_sync(controls_sync), .clock(clock_27mhz), .reset(reset), .title(title5),
.debug(debug_bus), .mouse_x(mouse_x), .mouse_y(mouse_y));

   // ***************************************************************
   // Modules for display subsystem
   // Last modified by David Wyatt
   // 2005/12/6

   // Display on the cube:
   //---------------------

   wire[63:0] data_bus_out;
   wire[7:0] plane_out;
   wire toggled;

   // Data output module
   data_output_module data_output_module1(.clock(clock_27mhz), .reset(reset_sync),
.plane_sel(plane_sel), .plane_out(plane_out), .data_bus_in(data_bus_in),
.data_bus_out(data_bus_out), .toggled(toggled));
      // Configure it for a system clock frequency of 27MHz
      defparam data_output_module1.input_clock_freq = 27000000;

   // Actual inputs and outputs!
   assign user1 = ~data_bus_out[31:0];
   assign user2 = ~data_bus_out[63:32];
   assign user3 = {24'b0, plane_out[7:0]};

   // Debugging
   // assign led = ~plane_out[7:0];
   // assign analyzer1_clock = clock_27mhz;

   // SVGA display on a monitor:
   //---------------------------

   // Parameters to configure the colours of the output on the screen
   parameter OFF_LED = 4'h0;
   parameter DIMMED_LED = 4'h4;
   parameter LIT_LED = 4'hF;
   parameter WIREFRAME_SHADE = 8'h7F; // Grey shade of the wireframe cube outline
   // Parameters for screen display size (in pixels)
   // A good combination is D=50, Dsin(theta) = 45, Dcos(theta) = 27, offsets x,y = 20
   // But for the moment we will use a 45 degree projection angle for simplicity of grid
lines...
   parameter LED_RADIUS = 3;     // Radius of circles to represent LEDs, maximum 8
   parameter D = 47;             // Linear LED spacing (corresponds to real 3D spacing)
   parameter D_SIN_THETA = 34;   // Projection of a Z-step of size D into the X direction
   parameter D_COS_THETA = 34;   // Projection of a Z-step of size D into the Y direction
   parameter OFFSET_X = 10;      // Distance of graphic from left screen edge
   parameter OFFSET_Y = 10;      // Distance of graphic from top screen edge

   // Renderer - does orthographic 3d to draw projection of cube on a screen buffer
   wire sbuffer_write_enable;
   wire [9:0] sbuffer_write_col, sbuffer_write_row, sbuffer_read_col, sbuffer_read_row;
   wire [3:0] sbuffer_data_in, sbuffer_data_out;
   renderer renderer1(.clock(clock_27mhz), .reset(reset_sync), .voxel_plane(plane_sel),
.voxel_data_bus(data_bus_out),
      .highlight_enable(switch[7]), .highlight_mode(switch[3]),
.highlighted_plane(switch[6:4]),
      .screen_rdata_out(sbuffer_data_in), .screen_row(sbuffer_write_row),
.screen_column(sbuffer_write_col),
      .screen_write_enable(sbuffer_write_enable));
      // Configure the colours and scale of the screen output
      defparam renderer1.OFF_LED = OFF_LED;
      defparam renderer1.DIMMED_LED = DIMMED_LED;
      defparam renderer1.LIT_LED = LIT_LED;
      defparam renderer1.LED_RADIUS = LED_RADIUS;
      defparam renderer1.D = D;
      defparam renderer1.D_SIN_THETA = D_SIN_THETA;
      defparam renderer1.D_COS_THETA = D_COS_THETA;
      defparam renderer1.OFFSET_X = OFFSET_X;
      defparam renderer1.OFFSET_Y = OFFSET_Y;

   // Clock manager to generate the 40MHz-ish clock (actually 40.5MHz) needed for the video
signal...
   // Copied from Lab 4 - I hope this works!
   wire clock_40mhz_unbuf, clock_40mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 2
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

   // Screen buffer (wrapped in a module) - stores 800 x 600 record of the top 4 bits of the
screen's red channel (!)
   screen_buffer_wrapper screen_buffer_wrapper1(.write_clock(clock_27mhz),
.write_col(sbuffer_write_col),
      .write_row(sbuffer_write_row), .write_data(sbuffer_data_in),
.write_enable(sbuffer_write_enable), .read_clock(clock_40mhz),
      .read_col(sbuffer_read_col), .read_row(sbuffer_read_row),
.read_data(sbuffer_data_out));

   // SVGA video interface module - based on Lab 4
   wire [9:0] hcount;
   wire [9:0] vcount;
   wire hsync, vsync, blank;
   svga svga1(.vclock(clock_40mhz), .hcount(hcount), .vcount(vcount), .hsync(hsync),
.vsync(vsync), .blank(blank));
   assign sbuffer_read_col[9:0] = hcount[9:0];
   assign sbuffer_read_row[9:0] = vcount[9:0];

   // Small module to draw a wireframe outline of the cube on the screen
   // Has a single output, wireframe/outline, which when high causes the current pixel to be
```

```
grey
   wire outline;
   wireframe_generator wireframe_generator1(.hcount(hcount), .vcount(vcount),
.wireframe(outline));
      // Configure the scale of the screen output
      defparam wireframe_generator1.D = D;
      defparam wireframe_generator1.D_SIN_THETA = D_SIN_THETA;
      defparam wireframe_generator1.D_COS_THETA = D_COS_THETA;
      defparam wireframe_generator1.OFFSET_X = OFFSET_X;
      defparam wireframe_generator1.OFFSET_Y = OFFSET_Y;

   // SVGA Output - based on Lab 4.
   // In order to meet the setup and hold times of the AD7125, we send it clock_40mhz.
   assign vga_out_red = (sbuffer_data_out[3:0]) ? {sbuffer_data_out[3:0], 4'b0} : (outline ?
WIREFRAME_SHADE : 8'h0);
   assign vga_out_green = (sbuffer_data_out[3:0]) ? 8'b0 : (outline ? WIREFRAME_SHADE :
8'h0);
   assign vga_out_blue = (sbuffer_data_out[3:0]) ? 8'b0 : (outline ? WIREFRAME_SHADE :
8'h0);
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~blank;
   assign vga_out_pixel_clock = clock_40mhz;
   assign vga_out_hsync = hsync;
   assign vga_out_vsync = vsync;


 endmodule
```

Appendix B: Data generation subsystem code

Note that the following files were used as provided by the 6.111 website:

- display_string.v <http://web.mit.edu/6.111/www/f2005/code/display_string.v>
- ps2_mouse.v <http://web.mit.edu/6.111/www/f2005/code/ps2_mouse.v>
  (parameters MAX_X and MAX_Y were both changed to 95)
- audio.v <http://web.mit.edu/6.111/www/f2005/handouts/lab3.v>
- debounce.v <http://web.mit.edu/6.111/www/f2005/handouts/debounce.v>
- synchronize.v <http://web.mit.edu/6.111/www/f2005/handouts/synchronize.v>

1. application_reset.v

```
module application_reset(selector, clock, reset);
  input [2:0] selector;
  input clock;
  output reset;

  reg [2:0] prev_selector;
  reg reset;

  always @ (posedge clock)
  begin
    if (prev_selector == selector)
      reset <= 1'b0;
    else begin
      reset <= 1'b1;
       prev_selector <= selector;
    end
  end
```

```
endmodule
```

2. cell_aut_2d.v

```
module cell_aut_2D(app_address, app_data_in, app_data_out, write_enable, controls_sync,
clock, reset, title);

  // Standard application inputs
  input [7:0] app_data_in;
  input [4:0] controls_sync;
  input clock, reset;

  // Standard application outputs
  output [5:0] app_address;
  output [7:0] app_data_out;
  output write_enable;
  output [127:0] title;

  assign title = "Cell Automata 2D";

  reg [5:0] app_address;
  reg [7:0] app_data_out;


  // -------------------------Cell_aut_2D--------------------------

  // Register to store state of entire cube
  reg [511:0] cube;
  reg [10:0] state_counter;   // [3:0] - 16 substates; [9:4] - 64 LEDs, [10] - Allow time to
load to space buffer
  reg [3:0] cell_counter;     // Counts the number of live neighbouring cells
  parameter pattern_ring =
64'b00000000_00001000_00010100_00010100_00001000_00001000_00000000_00000000;
  parameter pattern_glider  =
64'b00000000_00001000_00010000_00011100_00000000_00000000_00000000_00000000;
  parameter pattern_ship =
64'b00000000_00000000_01111000_10001000_00001000_10010000_00000000_00000000;

  wire clock_2khz;
  clock_divider cell_aut_2D_clock1_divider (clock, clock_2khz);
  defparam cell_aut_2D_clock1_divider.ratio = 13500;

  // Cell_aut_2D operates always in write mode wrt space_buffer
  assign write_enable = 1'b1;

  always @ (posedge clock)

  // Handle new pattern requests
  if (reset || controls_sync[0])
  begin
    cube <= 512'b0;
    state_counter <= 1024;
  end
  else if (controls_sync[1])
  begin
    cube <= {448'b0, pattern_ring};
    state_counter <= 1024;
  end
  else if (controls_sync[2])
  begin
    cube <= {448'b0, pattern_glider};
    state_counter <= 1024;
  end
  else if    (controls_sync[3])
  begin
    cube <= {448'b0, pattern_ship};
    state_counter <= 1024;
  end

  else if (clock_2khz)
  begin
    state_counter <= state_counter + 1;

    // Calculate next cube state
    if (state_counter < 1024)
    case (state_counter[3:0])
      0: if (state_counter[9:4] == 0) cube[511:64] <= cube[447:0];
      1: cell_counter <= 0;
      2: cell_counter <= cell_counter + cube[(state_counter[9:4] - 8 - 1) % 64 + 64];
      3: cell_counter <= cell_counter + cube[(state_counter[9:4] - 8) % 64 + 64];
      4: cell_counter <= cell_counter + cube[(state_counter[9:4] - 8 + 1) % 64 + 64];
      5: cell_counter <= cell_counter + cube[(state_counter[9:4] - 1) % 64 + 64];
      6: cell_counter <= cell_counter + cube[(state_counter[9:4] + 1) % 64 + 64];
      7: cell_counter <= cell_counter + cube[(state_counter[9:4] + 8 - 1) % 64 + 64];
      8: cell_counter <= cell_counter + cube[(state_counter[9:4] + 8) % 64 + 64];
      9: cell_counter <= cell_counter + cube[(state_counter[9:4] + 8 + 1) % 64 + 64];
      10: case (cell_counter)
                3: cube[state_counter[9:4]] <= 1'b1;               // Generate living cell
                2: cube[state_counter[9:4]] <= cube[state_counter[9:4]];    // Maintain
living cell
                default: cube[state_counter[9:4]] <= 1'b0;         // Cell dies
(isolation/overcrowding)
            endcase
    endcase

    // Store cube state into space buffer
    else if (state_counter < 1024 + 64)
    begin
      app_address <= 64 - state_counter[5:0];        // Subtracted from 64 to ensure that
most recent
                                                      // Life state is on the top plane
      app_data_out[0] <= cube[8*state_counter[5:0]+0];
      app_data_out[1] <= cube[8*state_counter[5:0]+1];
      app_data_out[2] <= cube[8*state_counter[5:0]+2];
      app_data_out[3] <= cube[8*state_counter[5:0]+3];
      app_data_out[4] <= cube[8*state_counter[5:0]+4];
      app_data_out[5] <= cube[8*state_counter[5:0]+5];
      app_data_out[6] <= cube[8*state_counter[5:0]+6];
      app_data_out[7] <= cube[8*state_counter[5:0]+7];
    end
  end



endmodule
```

3. cell_aut_3d.v

```
module cell_aut_3D(app_address, app_data_in, app_data_out, write_enable, controls_sync,
clock, reset, title);
  // Standard application inputs
  input [7:0] app_data_in;
  input [4:0] controls_sync;
  input clock, reset;
  // Standard application outputs
  output [5:0] app_address;
  output [7:0] app_data_out;
  output write_enable;
  output [127:0] title;

  assign title = "Cell Automata 3D";

  reg [5:0] app_address;
  reg [7:0] app_data_out;


  // -------------------------Cell_aut_3D--------------------------

  // Register to store state of cube
  reg [511:0] cube;
  reg [511:0] prev_cube;

  // Predefined game states
  parameter pattern_empty = 512'b0;
  parameter pattern_row = {192'b0,32'b0,8'b00111000,24'b0,256'b0};
  parameter pattern_corners = {8'b10000001, 48'b0, 8'b10000001, 384'b0, 8'b10000001, 48'b0,
8'b10000001};
  parameter pattern_random = {256'b0,
64'b1000110100100100100001001000001001001000100010000101001000100101, 192'b0};

  // 1khz clock -- For updating space buffer contents
  wire clock_27khz;
  clock_divider cell_aut_3D_clock1_divider (clock, clock_27khz);
  defparam cell_aut_3D_clock1_divider.ratio = 1000;

  // Cell_aut_3D operates always in write mode wrt space_buffer
  assign write_enable = 1'b1;

  //   Cell_aut_3D works like a 1024 x 32 state FSM:
  //     * First 512 x 32 states for computing next state of the cube
  //         - 512 main states for each of the 512 LEDs
  //         - 32 sub states for adding up neighbouring cell contents and deciding the next
cell state
  //     * Next 64 states used for updating space buffer with contents of cube register.
  //     * Remaining states are "do nothing" states

  reg [14:0] state_counter;         // Counter for maintaining state of game machine
  reg [4:0] cell_counter = 0;

  always @ (posedge clock)
  if (reset)
  begin
    cube <= pattern_empty;
  end
  else if (clock_27khz) begin

    // Check if new pattern has been requested (or if application has been reset)
    // If a pattern has been requested, jump to state 16384 to skip computing the next state
and
    // start displaying the next state.
    if (controls_sync[0] || reset)
    begin
      cube <= pattern_empty;
        state_counter <= 16384;
        cell_counter <= 0;
    end
    else if (controls_sync[1])
    begin
      cube <= pattern_row;
        state_counter <= 16384;
        cell_counter <= 0;
    end
    else if (controls_sync[2])
    begin
      cube <= pattern_corners;
        state_counter <= 16384;
        cell_counter <= 0;
    end
    else if (controls_sync[3])
    begin
      cube <= pattern_random;
        state_counter <= 16384;
        cell_counter <= 0;
    end

    // The "compute next cube" states
    else if (state_counter < 16384)
    begin

      // Select the appropriate sub-state
      case(state_counter[4:0])

          // Make copy of previous state of cube (to be used as a source for calculating
next state),
        // but only right at the beginning (before any states have cells have been
manipulated)
          0: if (state_counter[13:5] == 0) prev_cube <= cube;

          // Add up neighbouring cells one by one.
          // Note that state_counter[13:5] identifies the cell that we are manipulating
          1: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 - 8 - 1 +
512) % 512];
          2: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 - 8 + 512) %
512];
          3: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 - 8 + 1 +
512) % 512];
          4: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 - 1 + 512) %
512];
          5: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 + 512) %
512];
          6: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 + 1 + 512) %
512];
          7: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 + 8 - 1 +
512) % 512];
          8: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 + 8 + 512) %
512];
          9: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 64 + 8 + 1 +
512) % 512];
          10: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 8 - 1 + 512) %
512];
          11: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 8 + 512) %
512];
          12: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 8 + 1 + 512) %
512];
          13: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] - 1 + 512) %
512];
          // cell_counter <= cell_counter + prev_cube[(state_counter[13:5]) % 512];  (Don't
add the cell itself)
          14: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 1) % 512];
          15: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 8 - 1) % 512];
          16: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 8) % 512];
          17: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 8 + 1) % 512];
          18: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 - 8 - 1) %
512];
          19: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 - 8) %
512];
          20: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 - 8 + 1) %
512];
          21: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 - 1) %
512];
          22: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64) % 512];
          23: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 + 1) %
512];
          24: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 + 8 - 1) %
512];
          25: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 + 8) %
512];
          26: cell_counter <= cell_counter + prev_cube[(state_counter[13:5] + 64 + 8 + 1) %
512];

          // After counting the number of neighbouring cells, decide the next state for the
active
          // cell, and reset the cell counter.
          27: case(cell_counter)
                2: cube[state_counter[13:5]] <= prev_cube[state_counter[13:5]];
                3: cube[state_counter[13:5]] <= 1'b1;
                  4: cube[state_counter[13:5]] <= 1'b1;
                  default: cube[state_counter[13:5]] <= 1'b0;
            endcase
        28: cell_counter <= 0;
      endcase

      state_counter <= state_counter + 1;
    end

    // Update contents of space buffer
    // state_counter[5:0] identifies the column of cells we are updating (the space buffer
takes in
    // 8 cells at a time (i.e. 1 column)
    else if ((state_counter >= 16384) && (state_counter < 16384 + 64))
    begin
      app_address <= state_counter[5:0];
        app_data_out[7] <= cube[8*state_counter[5:0]+7];
        app_data_out[6] <= cube[8*state_counter[5:0]+6];
        app_data_out[5] <= cube[8*state_counter[5:0]+5];
        app_data_out[4] <= cube[8*state_counter[5:0]+4];
        app_data_out[3] <= cube[8*state_counter[5:0]+3];
        app_data_out[2] <= cube[8*state_counter[5:0]+2];
        app_data_out[1] <= cube[8*state_counter[5:0]+1];
        app_data_out[0] <= cube[8*state_counter[5:0]+0];
        state_counter <= state_counter + 1;
    end

    // The "do nothing state"
    else state_counter <= state_counter + 1;
  end

endmodule
```

4. clock_divider.v

```
module clock_divider(clock_in, clock_out);
  input clock_in;
  output clock_out;
  reg clock_out;

  parameter ratio = 1000;
  reg [14:0] counter = 0;

  always @ (posedge clock_in)
  if (counter >= ratio)
  begin
    counter <= 0;
    clock_out <= 1'b1;
  end
  else
  begin
    counter <= counter + 1;
    clock_out <= 1'b0;
  end

endmodule


module big_clock_divider(clock_in, clock_out);
  input clock_in;
  output clock_out;
  reg clock_out;

  parameter ratio = 1000;
  reg [24:0] counter = 0;

  always @ (posedge clock_in)
  if (counter >= ratio)
  begin
    counter <= 0;
    clock_out <= 1'b1;
  end
  else
  begin
    counter <= counter + 1;
    clock_out <= 1'b0;
  end

endmodule
```

5. mit.v

```
module mit(app_address, app_data_in, app_data_out, write_enable, controls_sync, clock,
reset, title);
  // Standard application inputs
  input [7:0] app_data_in;
  input [4:0] controls_sync;
  input clock, reset;

  // Standard application outputs
  output [5:0] app_address;
  output [7:0] app_data_out;
  output write_enable;
  output [127:0] title;
  assign title = "  MIT - 6.111   ";

  reg [5:0] app_address;
  reg [7:0] app_data_out;


  // ------------------------MIT--------------------------

  parameter M = 64'b10000001_11000011_10100101_10100101_10011001_10000001_10000001_10000001;
  parameter I = 64'b11111111_00011000_00011000_00011000_00011000_00011000_00011000_11111111;
  parameter T = 64'b11111111_00011000_00011000_00011000_00011000_00011000_00011000_00011000;

  parameter fade0 = 8'b00000000, fade1 = 8'b10000000, fade2 = 8'b10101010, fade3 =
8'b11111110;
  parameter fade4 = 8'b11111111, fade5 = 8'b11111110, fade6 = 8'b10101010, fade7 =
8'b10000000;
  reg [23:0] fadecounter;
  reg fader;

  wire [5:0] next_address;
  assign next_address = app_address + 1;
  assign write_enable = 1'b1;

  always @ (posedge clock)
  begin
    app_address <= next_address;
    fadecounter <= fadecounter + 1;
    case ((fadecounter[23:21] + app_address[5:3]) % 8)
      0: fader <= fade0[fadecounter[14:12]];
      1: fader <= fade1[fadecounter[14:12]];
      2: fader <= fade2[fadecounter[14:12]];
      3: fader <= fade3[fadecounter[14:12]];
      4: fader <= fade4[fadecounter[14:12]];
      5: fader <= fade5[fadecounter[14:12]];
      6: fader <= fade6[fadecounter[14:12]];
      7: fader <= fade7[fadecounter[14:12]];
    endcase

    app_data_out[0] <= fader && M[next_address] && I[{next_address[2:0], 3'b000}] &&
T[{next_address[5:3], 3'b000}];
    app_data_out[1] <= fader && M[next_address] && I[{next_address[2:0], 3'b001}] &&
T[{next_address[5:3], 3'b001}];
    app_data_out[2] <= fader && M[next_address] && I[{next_address[2:0], 3'b010}] &&
T[{next_address[5:3], 3'b010}];
    app_data_out[3] <= fader && M[next_address] && I[{next_address[2:0], 3'b011}] &&
T[{next_address[5:3], 3'b011}];
    app_data_out[4] <= fader && M[next_address] && I[{next_address[2:0], 3'b100}] &&
T[{next_address[5:3], 3'b100}];
    app_data_out[5] <= fader && M[next_address] && I[{next_address[2:0], 3'b101}] &&
T[{next_address[5:3], 3'b101}];
    app_data_out[6] <= fader && M[next_address] && I[{next_address[2:0], 3'b110}] &&
T[{next_address[5:3], 3'b110}];
    app_data_out[7] <= fader && M[next_address] && I[{next_address[2:0], 3'b111}] &&
T[{next_address[5:3], 3'b111}];
  end

endmodule
```

6. music_visualizer.v

```
module music_visualizer(app_address, app_data_in, app_data_out, write_enable, controls_sync,
clock, reset, title, debug, filter_output_ready, filter_output);
```

```
   // Standard application inputs
   input [7:0] app_data_in;
   input [4:0] controls_sync;
   input clock, reset;

   // Standard application outputs
   output [5:0] app_address;
   output [7:0] app_data_out;
   output write_enable;
   assign write_enable = 1'b1; // Always write to the space buffer!
   output [127:0] title;
   output [7:0] debug;

   reg [5:0] app_address;
   reg [7:0] app_data_out;
   reg [7:0] debug;

   // Input data samples - from a low pass filter
   input filter_output_ready; // Signals when data is ready
   input [18:0] filter_output; // Actual data bus - 19 bits wide

   // Title
   assign title = "Music Visualizer";

   // Registers to store the current mode - linear or rippling
   reg[1:0] mode = 0;
   parameter LINEAR = 0;
   parameter RIPPLING = 1;

   // Counter parameters
   parameter input_clock_freq = 27000000;
   parameter address_write_rate = 65000; // A bit more than 64kHz, so that we will have
finished writing out data before the next AC97 frame


   // -------------------------music_visualizer---------------------------
   reg [2:0] vumeter_data[7:0]; // Stores the VU meter datastream
   // reg [1:0] vumeter_selector; // The VU meter register to be output at the present time

   // N.B. To make sure we keep the CIC filter and the address counter in step, reset the
latter when the former outputs data
   wire address_counter_reset_signal;
   assign address_counter_reset_signal = filter_output_ready;

   // Clock divider to produce a 64kHz count_signal (high for one clock period every time)
   wire count_signal;
   divider address_clock_divider(.clock(clock), .reset(address_counter_reset_signal),
.out(count_signal));
      defparam address_clock_divider.input_clock_freq = input_clock_freq;
      defparam address_clock_divider.desired_frequency = address_write_rate;

   // A 6-bit counter (counts at 64kHz thanks to the clock divider) to generate write
addresses and generally act as a "cursor"
   // Reset by address_counter_reset_signal
   wire [5:0] address_counter_value;
   counter_6bit address_counter(.clock(clock), .count_input(count_signal),
.reset(address_counter_reset_signal), .value(address_counter_value));

   // Main loop
   always @ (posedge clock) begin
      // Every time the filter says it's ready, sample the top 3 bits of the filter output,
      // put them into vumeter_data[0]...
      if (filter_output_ready) begin
            vumeter_data[0] <= filter_output[18:16] + 4;

            // ...and shunt all the other sampled_ac97_datas down a register
            vumeter_data[1] <= vumeter_data[0];
            vumeter_data[2] <= vumeter_data[1];
            vumeter_data[3] <= vumeter_data[2];
            vumeter_data[4] <= vumeter_data[3];
            vumeter_data[5] <= vumeter_data[4];
            vumeter_data[6] <= vumeter_data[5];
            vumeter_data[7] <= vumeter_data[6];
      end

      // Select current mode depending on which button is pressed
      if (controls_sync[0]) mode <= LINEAR;
      else if (controls_sync[1]) mode <= RIPPLING;

      // Debug: filter output
      debug <= filter_output[18:11];

      // Use the whole cube as a kind of oscilloscope for the filtered AC97 signal - waves
ripple out from the centre
      // I.e. if the current horizontal plane number (address_counter_value[5:3]) equals the
vumeter_data reg
      // for that square (address_counter_value[2:0]), light all the LEDs (which will be a
rowful)
      app_address <= address_counter_value;

      // Choose the output data depending on the current mode
      case (mode)
         // Linear - a kind of slow oscilloscope
         LINEAR: app_data_out <= (address_counter_value[5:3] ==
vumeter_data[address_counter_value[2:0]]) ? 8'hFF : 8'h0;

         // Rippling - patterns ripple out from the centre - so the outermost square of the
cube (top, bottom, front, back) gets the oldest value etc.
         RIPPLING: begin
            if ((address_counter_value[5:3] == 0) || (address_counter_value[5:3] == 7) ||
(address_counter_value[2:0] == 0) || (address_counter_value[2:0] == 7)) app_data_out <= 8'h1
<< vumeter_data[6];
            else if ((address_counter_value[5:3] == 1) || (address_counter_value[5:3] == 6)
|| (address_counter_value[2:0] == 1) || (address_counter_value[2:0] == 6)) app_data_out <=
8'h1 << vumeter_data[4];
            else if ((address_counter_value[5:3] == 2) || (address_counter_value[5:3] == 5)
|| (address_counter_value[2:0] == 2) || (address_counter_value[2:0] == 5)) app_data_out <=
8'h1 << vumeter_data[2];
            else if ((address_counter_value[5:3] == 3) || (address_counter_value[5:3] == 4)
|| (address_counter_value[2:0] == 3) || (address_counter_value[2:0] == 4)) app_data_out <=
8'h1 << vumeter_data[0];
         end

      endcase
   end
endmodule

// 6 bit counter
module counter_6bit(clock, count_input, reset, value);
   input clock, count_input, reset;
   output [5:0] value;
   reg [5:0] value;

   // Value at which the counter resets itself
   parameter max_value = 63;
   always @(posedge clock) begin
      if (reset) value <= 0;
      else if (value >= max_value) value <= 0;
      else if (count_input) value <= value + 1;
   end
endmodule
```

7. mux_8input.v

```
module mux_8input_1bit(in0, in1, in2, in3, in4, in5, in6, in7, selector, out);

  input in0, in1, in2, in3, in4, in5, in6, in7;
  input [2:0] selector;
  output out;

  reg out;

  always @ (in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7 or selector)
    case(selector)
      3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
      3'b100: out = in4;
        3'b101: out = in5;
        3'b110: out = in6;
        3'b111: out = in7;
    endcase
endmodule

module mux_8input_6bit(in0, in1, in2, in3, in4, in5, in6, in7, selector, out);

  input [5:0] in0, in1, in2, in3, in4, in5, in6, in7;
  input [2:0] selector;
  output [5:0] out;

  reg [5:0] out;

  always @ (in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7 or selector)
    case(selector)
      3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
      3'b100: out = in4;
        3'b101: out = in5;
        3'b110: out = in6;
        3'b111: out = in7;
    endcase
endmodule

module mux_8input_8bit(in0, in1, in2, in3, in4, in5, in6, in7, selector, out);

  input [7:0] in0, in1, in2, in3, in4, in5, in6, in7;
  input [2:0] selector;
  output [7:0] out;

  reg [7:0] out;

  always @ (in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7 or selector)
    case(selector)
```

```
      3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
      3'b100: out = in4;
        3'b101: out = in5;
        3'b110: out = in6;
        3'b111: out = in7;
    endcase
endmodule

module mux_8input_128bit(in0, in1, in2, in3, in4, in5, in6, in7, selector, out);

  input [127:0] in0, in1, in2, in3, in4, in5, in6, in7;
  input [2:0] selector;
  output [127:0] out;

  reg [127:0] out;

  always @ (in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7 or selector)
    case(selector)
      3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
      3'b100: out = in4;
        3'b101: out = in5;
        3'b110: out = in6;
        3'b111: out = in7;
    endcase
endmodule
```

8. pong.v

```
module pong(app_address, app_data_in, app_data_out, write_enable, controls_sync, clock,
reset, title, pos_x, pos_y, mclick);

  // Standard application inputs
  input [7:0] app_data_in;
  input [4:0] controls_sync;
  input clock, reset;

  // Pong inputs
  input [2:0] pos_x, pos_y;
  input [2:0] mclick;

  // Translate mouse coordinates into game coordinates
  wire [2:0] paddle_x, paddle_y;
  assign paddle_x = pos_x + 1;
  assign paddle_y = 6 - pos_y;

  // Standard application outputs
  output [5:0] app_address;
  output [7:0] app_data_out;
  output write_enable;
  output [127:0] title;

  assign title = "3D Pong         ";

  // Pong registers & settings
  reg [2:0] x, y, z;                // 3d "cursor"
  reg [7:0] app_data_out;           // Register to store column of data to be written
  reg [7:0] temp_data_out;           // Register to store next column of data to be written
  reg [5:0] app_address;
  reg [5:0] ball_x = 42, ball_y = 42, ball_z = 42;       // Ball location
  reg signed [4:0] vel_x = 0, vel_y = 0, vel_z = 4;      // Ball velocity
  reg [7:0] count;                   // Frame counter (count frames before moving the ball)
  reg inv = 0;                           // Invert voxels to indicate a crash
  reg compute_mode;                  // Separates velocity computation from shifting of ball
                                     // Prevents end effects.

  parameter N = 4;                  // No. of frames generated before shifting ball
  parameter delta_vel = 3;           // Amount to change velocity of ball when bouncing off
paddle.

  assign write_enable = 1'b1;

  // Slow down the clock to 27kHz
  wire slow_clock;
  clock_divider pong_clock_divider (clock, slow_clock);
  defparam pong_clock_divider.ratio = 1000;

  always @ (posedge clock)
  if (reset)
  begin
    ball_x <= 42;
    ball_y <= 42;
    ball_z <= 42;
    vel_x <= 0;
    vel_y <= 0;
    vel_z <= 4;
  end
  else if (slow_clock) begin
    // Adjust "cursor" & write out data at appropriate point
    x <= x + 1;
    if (x == 7) y <= y + 1;
    if ((x == 7) && (y == 7)) z <= z + 1;

    if (x == 0)
    begin
      app_data_out <= temp_data_out;
      app_address <= {z, y} - 1;
    end

    // Check state at present cursor position
    temp_data_out[x] <= (inv ^^           // Crash indicator
        (((x == ball_x[5:3]) && (y == ball_y[5:3]) && (z == ball_z[5:3])) ^^  // Check if
we're at the ball
        (((x == paddle_x - 1) || (x == paddle_x) || (x == paddle_x + 1)) &&   // Check if
we're at the paddle
        ((y == paddle_y - 1) || (y == paddle_y) || (y == paddle_y + 1)) &&
          (z == 0))));

    // Shift game elements at the end of every N cycles
    if ((x == 7) && (y == 7) && (z == 7))
    begin
      if (count >= N) begin
        compute_mode <= compute_mode + 1;
        if (compute_mode) begin
          count <= 0;

          // Check for collisions with sides + top
          if (ball_x[5:3] == 0) vel_x <= vel_x[4] ? -vel_x : vel_x;
          if (ball_x[5:3] == 7) vel_x <= vel_x[4] ? vel_x : -vel_x;
          if (ball_y[5:3] == 0) vel_y <= vel_y[4] ? -vel_y : vel_y;
          if (ball_y[5:3] == 7) vel_y <= vel_y[4] ? vel_y : -vel_y;
          if (ball_z[5:3] == 7) vel_z <= vel_z[4] ? vel_z : -vel_z;

          // Check if ball is at lowest plane
            if (ball_z[5:3] == 0)
            begin
              // Reverse z-direction of ball
              vel_z <= vel_z[4] ? -vel_z : vel_z;

               // Check if ball collided with paddle
               if ((ball_x[5:3] >= paddle_x - 1) && (ball_x[5:3] <= paddle_x + 1) &&
                   (ball_y[5:3] >= paddle_y - 1) && (ball_y[5:3] <= paddle_y + 1))
            begin
              // Alter velocity according to where ball struck paddle,
                 // ensuring magnitude of velocity does not exceed 2.
                 if (ball_x[5:3] == paddle_x - 1) vel_x <= (vel_x <= -2) ? -2 : vel_x - 1;
                 else if (ball_x[5:3] == paddle_x + 1) vel_x <= (vel_x >= 2) ? 2 : vel_x +
1;

                 if (ball_y[5:3] == paddle_y - 1) vel_y <= (vel_y <= -2) ? -2 : vel_y - 1;
                 else if (ball_y[5:3] == paddle_y + 1) vel_y <= (vel_y >= 2) ? 2 : vel_y +
1;
              end
            else inv <= 1'b1;
            end

            // Clear the "crash signal"
            if (ball_z[5:3] == 2) inv <= 1'b0;
          end
          else begin
          // Advance ball position
          ball_x <= ball_x + vel_x;
          ball_y <= ball_y + vel_y;
          ball_z <= ball_z + vel_z;
        end
       end
       else count <= count + 1;
    end
  end

endmodule
```

Appendix C: Display subsystem code

1. data_output_module.v

```
// Module to read data out of space buffer RAM
// and output it to whereever it will be displayed
module data_output_module(clock, reset, plane_sel, plane_out, data_bus_in, data_bus_out,
toggled);
   parameter input_clock_freq = 27000000; // Frequency of clock in Hz
   parameter plane_rate = 1000; // Rate at which the display cycles through the planes in Hz

   // Inputs and outputs
   input clock, reset;
   input [63:0] data_bus_in;
   output [2:0] plane_sel;
   output [63:0] data_bus_out;
   output [7:0] plane_out;
   output toggled;
   wire count_signal;

   // Clock divider
   divider clock_divider(.clock(clock), .reset(reset), .out(count_signal),
.toggled(toggled));
      defparam clock_divider.input_clock_freq = input_clock_freq;
      defparam clock_divider.desired_frequency = plane_rate;

   // 3 bit counter
   counter_3bit counter(.clock(clock), .count_input(count_signal), .reset(reset),
.value(plane_sel));

   // 1-of-8 decoder
   assign plane_out[7:0] = (8'b1 << plane_sel);

   assign data_bus_out = data_bus_in;
endmodule


//*********************************************************************************************
********
//*********************************************************************************************
********

//Divide an input clock to produce one active pulse at a lower frequency
module divider(clock, reset, out, toggled);

   // Frequency of the input clock (defaults to 27MHz)
   parameter input_clock_freq = 27000000;

   // Frequency of output pulses (defaults to 1Hz)
   parameter desired_frequency = 1;

   //Initial value of the counter that will be decremented every cycle
       parameter CYCLES = input_clock_freq/desired_frequency;

   //Input and output
       input clock, reset; //27MHz clock and reset line
       output out, toggled; // output that is high for one clock cycle every 1s, and one
that is toggled every 1s
       reg out = 0, toggled = 0;

       //Register array to count down - 25 bits wide (to store up to 27e6 in binary)
       reg [24:0] delay = 0;

       always @(posedge clock)
             if (reset) // Clear the timer if reset
         begin
            delay <= 0;
            out <= 0;
         end
      else if (delay == CYCLES) // If we reach the 1s mark, clear timer, toggle toggled and
set out
                   begin
                          delay <= 0;
                          out <= 1;
            toggled = ~toggled;
                   end
             else // If we're just counting normally, count normally and clear out (so it
only stays high for one cycle)
                   begin
                          delay <= delay + 1;
                          out <= 0;
```

```
                    end

endmodule

//***************************************************************************************
********
//***************************************************************************************
********

// 3 bit counter
module counter_3bit(clock, count_input, reset, value);
   input clock, count_input, reset;
   output [2:0] value;
   reg [2:0] value;

   always @(posedge clock) begin
      if (reset) value <= 0;
      else if (count_input) value <= value + 1;
   end

endmodule
```

2. renderer.v

```
// SVGA rendering and display module
// David Wyatt, 2005/12/4

// Renderer - projects an image of the cube onto a 2d screen buffer

module renderer(clock, reset, voxel_plane, voxel_data_bus, highlight_enable, highlight_mode,
highlighted_plane, screen_rdata_out, screen_row, screen_column, screen_write_enable);
   // Inputs and outputs
   input clock, reset, highlight_enable, highlight_mode;
   input [2:0] voxel_plane, highlighted_plane;
   input [63:0] voxel_data_bus;
   output screen_write_enable;
   output [3:0] screen_rdata_out;
   output [9:0] screen_row, screen_column;

   // Parameters to determine red shades of lit, off and highlighted LEDs - set in the main
labkit module
   parameter OFF_LED = 4'h0;
   parameter DIMMED_LED = 4'h4;
   parameter LIT_LED = 4'hF;
   // Parameters for screen display size (in pixels) - set in the main labkit module
   parameter LED_RADIUS = 8;     // Radius of circles to represent LEDs, maximum 8
   parameter D = 40;             // Linear LED spacing (corresponds to real 3D spacing)
   parameter D_SIN_THETA = 28;   // Projection of a Z-step of size D into the X direction
   parameter D_COS_THETA = 28;   // Projection of a Z-step of size D into the Y direction
   parameter OFFSET_X = 20;      // Distance of graphic from left screen edge
   parameter OFFSET_Y = 20;      // Distance of graphic from top screen edge

   // 14-bit counter and output
   wire [13:0] counter_value;
   counter_14bit counter(.clock(clock), .count_input(1'b1), .reset(reset),
.value(counter_value));

   // Voxel screen location calculator
   wire [9:0] screen_row_base, screen_column_base;
   screen_loc_calc screen_loc_calc1(.clock(clock), .reset(reset), .voxel_plane(voxel_plane),
.voxel_column(counter_value[13:8]),
       .screen_row(screen_row_base), .screen_column(screen_column_base));
```

```
      // Configure the scale of the screen output
      defparam screen_loc_calc1.D = D;
      defparam screen_loc_calc1.D_SIN_THETA = D_SIN_THETA;
      defparam screen_loc_calc1.D_COS_THETA = D_COS_THETA;
      defparam screen_loc_calc1.OFFSET_X = OFFSET_X;
      defparam screen_loc_calc1.OFFSET_Y = OFFSET_Y;

   assign screen_column = screen_column_base + counter_value[3:0] - 8;
   assign screen_row = screen_row_base + counter_value[7:4] - 8;

   // Definitions for possible highlighting modes - either horizontal or vertical planes
   parameter HORIZ = 0;
   parameter VERT = 1;

   // Code to determine colour of the current voxel's projection
   reg [3:0] current_spot_colour;
   always @ (posedge clock) begin
      if (highlight_enable == 1) begin
         if (highlight_mode == HORIZ) begin
            current_spot_colour <= voxel_data_bus[counter_value[13:8]] ? ((highlighted_plane
== voxel_plane) ? LIT_LED : DIMMED_LED) : OFF_LED;
         end
         if (highlight_mode == VERT) begin
            current_spot_colour <= voxel_data_bus[counter_value[13:8]] ? ((highlighted_plane
== counter_value[13:11]) ? LIT_LED : DIMMED_LED) : OFF_LED;
         end
      end
      else begin
         current_spot_colour <= voxel_data_bus[counter_value[13:8]] ? LIT_LED : OFF_LED;
      end

   end

   //assign current_spot_colour = voxel_data_bus[counter_value[13:8]] ? ((highlighted_plane
== voxel_plane) ? LIT_LED : DIMMED_LED) : OFF_LED;

   // Write the LED's colour to the screen buffer only if the current writing pixel position
   // within the 16x16 square corresponding to a particular voxel's projection
   // is within a certain radius of the LED's centre position, to draw circles
   // using the register pixel_on
   reg pixel_on = 0;
   reg signed [4:0] x_diff, y_diff;
   always @ (counter_value) begin
      x_diff <= counter_value[3:0] - 5'h8;
      y_diff <= counter_value[7:4] - 5'h8;
      if (x_diff * x_diff + y_diff * y_diff < LED_RADIUS * LED_RADIUS) pixel_on <= 1;
      else pixel_on <= 0;
   end
   assign screen_rdata_out = current_spot_colour;
   assign screen_write_enable = pixel_on;

endmodule

//*******************************************************************************************
********

// 14 bit counter
module counter_14bit(clock, count_input, reset, value);
   input clock, count_input, reset;
   output [13:0] value;
   reg [13:0] value;

   always @(posedge clock) begin
```

```
      if (reset) value <= 0;
      else if (count_input) value <= value + 1;
   end

endmodule

//*****************************************************************************************
********

// Voxel screen location calculator
module screen_loc_calc(clock, reset, voxel_plane, voxel_column, screen_row, screen_column);
   // Inputs and outputs
   input clock, reset;
   input [2:0] voxel_plane;
   input [5:0] voxel_column;
   output [9:0] screen_row, screen_column;

   // Parameters to configure the scale of the output on the screen - set in the main labkit
module
   // A good combination is D=50, Dsin(theta) = 45, Dcos(theta) = 27, offsets x,y = 20
   // But for the moment we will use a 45 degree projection angle for simplicity of grid
lines...
   parameter D = 40;             // Linear LED spacing (corresponds to real 3D spacing)
   parameter D_SIN_THETA = 28;   // Projection of a Z-step of size D into the X direction
   parameter D_COS_THETA = 28;   // Projection of a Z-step of size D into the Y direction
   parameter OFFSET_X = 20;      // Distance of graphic from left screen edge
   parameter OFFSET_Y = 20;      // Distance of graphic from top screen edge

   // Actual calculations
   assign screen_column = OFFSET_X + D * voxel_column[2:0] + D_SIN_THETA * (7 -
voxel_column[5:3]);
   assign screen_row = OFFSET_Y + D * (7 - voxel_plane) + D_COS_THETA * (7 -
voxel_column[5:3]);

endmodule

//*****************************************************************************************
**********

// Module to translate from the block diagram's idea of a screen buffer into reality!
module screen_buffer_wrapper(write_clock, write_col, write_row, write_data, write_enable,
read_clock, read_col, read_row, read_data);
   //Inputs and outputs
   input write_clock, write_enable, read_clock;
   input [9:0] write_col, write_row, read_col, read_row;
   input [3:0] write_data;
   output [3:0] read_data;

   // Parameters to specify screen resolution; default is 800 x 600
   parameter ROWSIZE = 800;
   parameter COLSIZE = 600;

   // Wires for the actual read and write addresses to the memory
   wire [18:0] write_address, read_address;

   // Calculate the actual addresses meant by the wires
   assign write_address = write_col + ROWSIZE * write_row;
   assign read_address = read_col + ROWSIZE * read_row;

   // Instantiate a screen_buffer memory (which is not 1024 x 1024!)
   screen_buffer screen_buffer1(.addra(write_address), .dina(write_data),
.wea(write_enable), .clka(write_clock),
      .addrb(read_address), .doutb(read_data), .clkb(read_clock));
```

```
endmodule

//*********************************************************************************************
**********

// 800 x 600 SVGA display module, to display the contents of the screen buffer
module svga(vclock, hcount, vcount, hsync, vsync, blank);
   input vclock;
   output [9:0] hcount;
   output [9:0] vcount;
   output    vsync;
   output    hsync;
   output    blank;

   reg         hsync,vsync,hblank,vblank,blank;
   reg [10:0] hcount_internal;    // pixel number on current line
   reg [9:0] vcount;       // line number

   // horizontal: 1055 pixels total
   // display 800 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount_internal == 799);
   assign    hsyncon = (hcount_internal == 839);
   assign    hsyncoff = (hcount_internal == 967);
   assign    hreset = (hcount_internal == 1055);

   // Output only values of hcount between 0 and 799, otherwise the screen buffer will get
confused!
   assign hcount = (hcount_internal <= 799) ? hcount_internal[9:0] : 10'b0;

   // vertical: 627 lines total
   // display 600 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 599);
   assign    vsyncon = hreset & (vcount == 600);
   assign    vsyncoff = hreset & (vcount == 604);
   assign    vreset = hreset & (vcount == 627);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount_internal <= hreset ? 0 : hcount_internal + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule


//*********************************************************************************************
**********

// Module to generate wireframe image of a 45 degree cube on SVGA output
// Generates pixels on the fly, and sets output "wireframe" high if the current
// pixel is part of the wireframe
```

```
module wireframe_generator(hcount, vcount, wireframe);
   // Inputs
   input [9:0] hcount, vcount;

   // Output (also a reg)
   output wireframe;
   reg wireframe = 0;

   // Parameters for screen display size (in pixels) - set in the main labkit module
   parameter D = 40;             // Linear LED spacing (corresponds to real 3D spacing)
   parameter D_SIN_THETA = 28;   // Projection of a Z-step of size D into the X direction
   parameter D_COS_THETA = 28;   // Projection of a Z-step of size D into the Y direction
   parameter OFFSET_X = 20;      // Distance of graphic from left screen edge
   parameter OFFSET_Y = 20;      // Distance of graphic from top screen edge

   always @ (hcount or vcount) begin
      // Horizontal lines at the back of the cube
      // Top
      if ((hcount >= OFFSET_X) && (hcount <= OFFSET_X + 7*D) && (vcount == OFFSET_Y))
wireframe <= 1;
      // Bottom
      else if ((hcount >= OFFSET_X) && (hcount <= OFFSET_X + 7*D) && (vcount == OFFSET_Y +
7*D)) wireframe <= 1;

      // Horizontal lines at the front of the cube
      // Top
      else if ((hcount >= OFFSET_X + 7*D_SIN_THETA) && (hcount <= OFFSET_X + 7*D +
7*D_SIN_THETA) && (vcount == OFFSET_Y + 7*D_COS_THETA)) wireframe <= 1;
      // Bottom
      else if ((hcount >= OFFSET_X + 7*D_SIN_THETA) && (hcount <= OFFSET_X + 7*D +
7*D_SIN_THETA) && (vcount == OFFSET_Y + 7*D + 7*D_COS_THETA)) wireframe <= 1;

      // Vertical lines at the back of the cube
      // Left
      else if ((vcount >= OFFSET_Y) && (vcount <= OFFSET_Y + 7*D) && (hcount == OFFSET_X))
wireframe <= 1;
      // Right
      else if ((vcount >= OFFSET_Y) && (vcount <= OFFSET_Y + 7*D) && (hcount == OFFSET_X +
7*D)) wireframe <= 1;

      // Vertical lines at the front of the cube
      // Left
      else if ((vcount >= OFFSET_Y + 7*D_COS_THETA) && (vcount <= OFFSET_Y + 7*D +
7*D_COS_THETA) && (hcount == OFFSET_X + 7*D_SIN_THETA)) wireframe <= 1;
      // Right
      else if ((vcount >= OFFSET_Y + 7*D_COS_THETA) && (vcount <= OFFSET_Y + 7*D +
7*D_COS_THETA) && (hcount == OFFSET_X + 7*D + 7*D_SIN_THETA)) wireframe <= 1;

      // Diagonal lines
      // Top left
      else if ((hcount >= OFFSET_X) && (hcount <= OFFSET_X + 7*D_SIN_THETA) &&
          (vcount >= OFFSET_Y) && (vcount <= OFFSET_Y + 7*D_COS_THETA) &&
          (hcount - OFFSET_X == vcount - OFFSET_Y)) wireframe <= 1;
      // Top right
      else if ((hcount >= OFFSET_X + 7*D) && (hcount <= OFFSET_X + 7*D + 7*D_SIN_THETA) &&
          (vcount >= OFFSET_Y) && (vcount <= OFFSET_Y + 7*D_COS_THETA) &&
          (hcount - OFFSET_X - 7*D == vcount - OFFSET_Y)) wireframe <= 1;
      // Bottom left
      else if ((hcount >= OFFSET_X) && (hcount <= OFFSET_X + 7*D_SIN_THETA) &&
          (vcount >= OFFSET_Y + 7*D) && (vcount <= OFFSET_Y + 7*D + 7*D_COS_THETA) &&
          (hcount - OFFSET_X == vcount - OFFSET_Y - 7*D )) wireframe <= 1;
      // Bottom right
      else if ((hcount >= OFFSET_X + 7*D) && (hcount <= OFFSET_X + 7*D + 7*D_SIN_THETA) &&
          (vcount >= OFFSET_Y + 7*D) && (vcount <= OFFSET_Y + 7*D + 7*D_COS_THETA) &&
          (hcount - OFFSET_X - 7*D == vcount - OFFSET_Y - 7*D )) wireframe <= 1;
      else wireframe <= 0;
   end


endmodule
```