Pong in Bluespec SystemVerilog
An extended example

June 15, 2004

Document generated on June 16, 2004

Abstract

This document gives a complete description of the Bluespec SystemVerilog code for the Pong video game. The implementation, which includes complete video signal generation, consists of 14 files (just over a 1000 lines), plus two small Verilog wrapper files for interfacing to standard FPGA tools (130 lines), plus an off-the-shelf Verilog keyboard-scanning module (338 lines). This description is intended for those would like to get a feel for a real, non-trivial Bluespec SystemVerilog application. It also illustrates how Bluespec SystemVerilog interoperates with standard Verilog modules.

1 The Pong video game

The Pong video game has been implemented and demonstrated on an XSA FPGA card[1]. The card is connected to a standard video monitor and to a standard PS2 keyboard. The image on the screen is depicted in Figure 1. It contains a border, a ball, left and right paddles and left and right scores. The ball

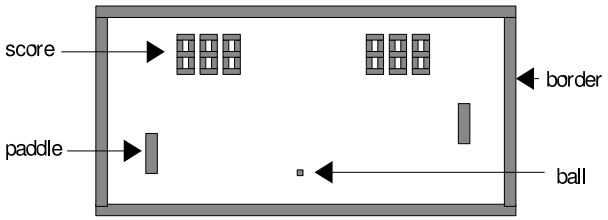

Figure 1: The Pong screen.

can move in arbitrary directions, and bounces off the walls and paddles. The paddles can move vertically, either automatically or manually in response to keystrokes. In autoplay mode, a paddle moves to meet the ball except with a little randomization so that it can occasionally miss. The left paddle can be toggled from automatic to manual mode, and vice versa, using the "`q`" key. In manual mode, it can be moved up and down with the "`a`" and "`z`" keys, respectively. The corresponding keys for the right paddle are "`]`", "`'`" and "`/`", respectively. The left and right paddle keys are arranged vertically on the left and right edge of a standard QWERTY keyboard. There are also a few switches and lamps on the FPGA card which can be used for control and status.

[1] By Niklas Röjemo.

1.1 Caveat about syntax details

At time of writing SystemVerilog is still an evolving standard. Bluespec, Inc. is participating in the standardization process, including donating language proposals. Bluespec, Inc. is also simultaneously developing its Bluespec SystemVerilog product. Consequently, the syntax of Bluespec SystemVerilog as shown in this document may change to keep track of these developments. The application was originally implemented in "Bluespec Classic", which predates SystemVerilog and has been used for several years.

2 Package/Module hierarchy and general principles

Bluespec SystemVerilog code is organized into packages. A common style is that often (but not always) a package P contains an interface type P and a module constructor *mkP* (or more than one, *mkP1*, *mkP2*, and so on). Thus, the package import hierarchy often follows the module hierarchy, and we are somewhat loose in referring to P as a package, interface or module.

The following table depicts the package import hierarchy from the root (`TopLevel`, in the first row) to the leaves (`Global` ... `VGACore` in the last row). A package name is shown on the left (e.g., `Paddle`) and the packages that it imports are shown on the right (e.g., `Shape`, `Global`, `Color` and `Counter`).

package	**imports**
TopLevel	Controller Ball Border Paddle Score Shape Global Color Decimal KbdV Switch VGACore
Controller	Ball Paddle KbdV
Ball	Paddle Border Shape Global Color
Border	Shape Global Color
Paddle	Shape Global Color Counter
Score	Shape Global Color Decimal
Shape	Global Color
Global Color Counter Decimal Kbd Switch VGACore	

In this design, the video signal is generated on the fly, i.e., there is no separate "video memory" holding the screen image. The video signal consists of 8 output wires: `hsync` (1 bit), `vsync` (1 bit), `red` (2 bits), `green` (2 bits) and `blue` (2 bits). The `VGACore` module continually updates a pair of registers holding the horizontal and vertical video beam position, in a standard raster pattern. Based on this, it generates `hsync` and `vsync` directly.

The hardware maintains state for each "object" on the screen (border, paddles, ball and score segments), including its position, shape and color. The red, green and blue signals are continuously computed as follows. At each position (h,v) of the raster beam, every object is queried about its color at that position. Each object reports its color if (h,v) is inside it, else it reports "no color". All these object colors are then combined into a single composite (red,green,blue) signal value for this raster position. The combining function is user-defined, so that overlapping objects can be treated properly.

The `TopLevel` module receives a "tick" from `VGACore` on every vertical sync and passes this on to the `Ball`, causing it to compute its new position, direction and velocity based on its current position, direction and velocity and whether it encounters a paddle or the border, along with some randomization.

The `Controller` module gets keystrokes from the `Kbd` module and manages the paddle state (autoplay or manual) and position accordingly.

The remaining sections of this document describe each source file in more detail. All the source files are shown in the appendix.

3 The `Global` package

The package `Global` collects a number of global constants, types and functions that are used throughout the application. An `import` M`::*` statement imports all identifiers defined in the package M. The definitions in the package are fairly self-explanatory. Note that constants can be defined in terms of other constants using built-in operators (e.g., `ballWidth`) as well as defined functions (e.g., `vgaTiming`; the function `sizeToTiming` is defined in package `VGACore`).

The types `XCoord` and `YCoord` are each defined to be structures with a single field. This ensures that they are new distinguished types that cannot be confused with any other types. Static type-checking helps ensure that we never confuse these types with any other types, even if they had the same representation.

4 The `TopLevel` module

In package `TopLevel` the module constructor `mkTopLevel` exports an interface of `TopLevel` type. The interface consists of a number of output wires for the video output: `hsync`, `vsync`, `red` (2 bits), `green` (2 bits) and `blue` (2 bits). It also contains sub-interfaces `rawkbd`, `rawsw1` and `rawsw2` for the keyboard and on-board switches, respectively. Finally, it contains two output wires `aL` and `aR` to drive lamps indicating the autoplay status of the left and right paddles, respectively.

The attributes just before the `mkTopLevel` constructor tell the compiler that this module will become a Verilog module and, further, that the normal handshaking in Bluespec SystemVerilog interfaces should be omitted (so, outputs are always ready, and inputs are always acceptable).

The `mkTopLevel` constructor exports an interface of type `TopLevel`. It instantiates sub-modules for the keyboard, on-board switches, a random-number generator for the automatic play mode, score values, score

displays, the video generator, border, paddles, ball and the controller. It creates a register `the_color` with interface `color` to hold the color to be sent to the video outputs at each instant in time.

The functions `flipCol` and `flipBCol` are meant to change the color of various objects and of the entire screen, depending on certain states and inputs. The function `flipCol` allows one to change the color seen through a `Shape` interface depending on the boolean argument `b`. The function `flipBCol` is similar, and is intended to invert a color into its "reverse video" color.

These functions are used in the next series of definitions. `padL` and `padR` are the left and right paddle shapes, respectively, except flipped to yellow if in autoplay mode. `border1` and `ball1` are the border and ball shapes, except with colors changed if the board's switch 1 is ON. `pict` combines all the shapes on the screen into one composite shape. `pict1` reverses the video (by flipping with White) if the board's switch 2 is ON. Finally, `pictBL` is the same as `pict1` except that the color is supressed during the "blank" period of the raster cycle.

The next several lines of code (methods) simply connect up all the interface wires. These are followed by two rules expressing the top-level behavior.

The two attributes preceding the first rule assert to the compiler that this rule can fire every time its explicit condition allows it to fire (which here means on every clock since the explicit condition is `True`). The first asserts that none of the method invocations in the rule have any implicit conditions (and hence they cannot disable the rule). The second asserts that the rule has no conflicts with any other rule, and so will fire whenever it is enabled. The compiler checks these assertions, and the compilation fails if they are untrue.

On each clock, the rule extracts the current horizontal and vertical raster position from `vgaCore`, and sends it to the composite Shape `pict` which will, recursively, send it to all its sub-Shapes. The rule also advances the random-number generator `lfsr`, and also computes the composite color for the composite shape `pictBl`, storing it in the register `color`.

The attribute preceding the second rule asserts that the rule has no conflicts with any other rule, and so will fire whenever it is enabled, i.e., whenever `vgaCore.frameTick` is True (in package `VGACore` we can see that this happens on every vertical refresh). Whenever it fires, it invokes the `tick` method on the ball.

5 The `VGACore` module

From the `TopLevel` module at the root let us now examine `VGACore`, a leaf module (at the bottom of the module hierarchy). The figure in comments at the top of the file shows the shape of each of the hsync and vsync signals of the raster scan. There is an active region during which pixels are displayed, followed by a blanking region. The sync signal is high except for a period within the blanking region, as shown in the comment figure.

The top of the file defines structure types to contain the timing parameters, defines various structure constants containing specific timing parameters, and contains the functions `hzToTiming` and `sizeToTiming` which may also be used to compute timing parameters (not all of them are used).

In the interface type `VGACore`, all the methods are outputs. The module constructor `mkVGACore` creates a module that exports a `VGACore` interface. Note that both the interface type and the module constructor are parameterized by the type of the horizontal and vertical coordinates, i.e., they are polymorphic in those types.

The first few statements in `mkVGACore` just instantiate registers for the horizontal and vertical positions and syncs, etc. The next few statements are some local definitions (`hSize`, `vSize`, ...) for convenience, based on the timing parameters and registers. These are followed by the rules that compute the signals.

Each rule has two attributes that together assert that the rule will fire whenever the rule's explicit condition is True– there are no implicit conditions nor conflicts that can stop it from firing. The first rule allows the clock to be stretched. If `preScale` is greater than 1, then the `tick` rule and the `preScale` constant (defined in package `Global`) cause `scale` to count down repeatedly from `preScale-1` to 0. The `hclk` rule either fires on every clock (if `preScale` is 1), or else every time `scale` counts down to 0. On each firing, it increments `hPosR`, wrapping around to 0 whenever it reaches `hTotal`, the total horizontal size (see local definition for `hTickLocal`). Similarly the `vclk` rule increments `vPosR` after each horizontal scan, wrapping around to 0 whenever it reaches `vTotal`, the total vertical size (see local definition for `vTickLocal`).

The `hSyncOn`, `hSyncOff`, `vSyncOn` and `vSyncOff` rules simply raise and lower the horizontal and vertical syncs at the appropriate horizontal and vertical counts. Finally, the interface method outputs are defined as simple combinations of previously defined values.

6 The `Color` and `Shape` packages

The `Color` package defines the `Color` type (a structure containing red, green and blue values), a function `mkRGB` to construct such values, and various fixed colors.

The functions `colorOr` and `colorXOr` simply apply the relevant operator pairwise to the red, green and blue components of its arguments, returning the resulting `Color`. Finally, the "get" functions retrieve the color components. They serve to hide the fact that a `Color` is represented as a structure, so that `Color` can be treated as an abstract type; we can freely change its representation without affecting any other packages that use this package.

In the `Shape` package, a `Shape` interface contains two methods: `newPos` accepts information about the current raster position, and `color` returns a color for the previously given raster position. Conceptually, `color` returns "no color" (`cNone`) when the raster position is outside the shape, and some particular color when it is inside.

The `emptyShape` can be defined as a pure interface, i.e., we don't need to define a module constructor which returns this interface because the module would be empty, with no internal state and no internal behavior (rules). The interface simply ignores the `newPos` information, and always returns the color `cNone`.

The function `visCheck` and the module constructor `mkRectangle` together define a shape corresponding to a rectangle. `mkRectangle` is parameterized by the rectangle's x- and y-coordinates and sizes, and by its color. Its internal state consists of just two boolean registers `xVis` and `yVis` which record whether or not the current raster position is within its x-extent and y-extent, respectively. These booleans are computed using the function `visCheck`, which takes advantage of our raster scan where the x-position (respectively, y-position) has a sawtooth waveform. It also exploits the fact that `VGACore` presents every position value in turn without omitting any. Thus, it can just use simple equality checks to toggle visibility, instead of the more expensive "$\leq$" and "$\geq$" comparisons.

`modShapeVis` is another higher-order function. Given a `Color`-transforming function `f` and a shape interface `s`, it returns a new shape interface which is the same as `s` except that its color is transformed by `f`.

The last two functions are shape-combiners. `joinShapes` combines two shapes: it simply passes on the `newPos` information to the sub-shapes, and combines their colors using the "or" function. `joinManyShapes`

extends this to an array of shapes, doing a "reduction" operation, applying `joinShapes` pairwise across the array to produce a net composite shape. It uses the library function `foldr` to perform the reduction. `foldr` is a very general-purpose reduction function. Given the binary operator "`+`", the initial value 0, and an array of numbers, it adds them together. Given the binary operator "`*`", the initial value 1, and an array of numbers, it multiplies them together. Here, we give it the binary function `joinShapes`, the initial value `emptyShape` and an array of shapes.

The `mkBorder` module in the `Border` package implements the border as follows. First we define the rectangle contained inside the border. Then we define its "inverse", using the `modShapeVis` function, giving it the color exclusive-or function. The net effect is to define the border band around the edges of the screen.

All the pieces are now in place to understand how the video color signal is generated. The screen image consists of a hierarchical composite of rectangular shapes. The border is a shape, described in the previous paragraph. The paddles and ball are rectangles. Each score consists of 3 digits, each of which consists of seven rectangular segments. In module `mkTopLevel`, we combine all these shapes using `joinManyShapes`, we supply the current raster position using `newPos`, and we get the color using `color`.

7 The `Counter` and `Paddle` packages

In the `Counter` package, `mkCounter` constructs a module that implements a simple up/down counter within the range `lower` to `upper` and initial value `start`. Each call to the `inc_dec` interface method bumps the counter up or down, depending on the boolean argument.

In the `Paddle` package, the `Paddle` interface contains the paddle's `Shape` interface and the `inc_dec` method to move its position up or down depending on its boolean argument (using an up/down `Counter`). The `inside` method takes the x- and y-coordinates of the boundaries of another rectangle (the ball) and returns 4 booleans corresponding to whether each corner of the ball rectangle is within the paddle area, and a y-dimension corresponding to the difference in the heights of their centers. These 5 results are returned in a `Tuple` type, which is simply a structure (record) with 5 components. The `center` method just returns the y-coordinate of the center of the paddle (used during autoplay to compare the y-coordinate of the center of the paddle with the y-coordinate of the center of the ball).

The `mkPaddle` module constructor is straightforward. It is parameterized by the x-position of the paddle (since it will be invoked once for each of the two paddles). It instantiates an up/down `Counter` for the y-position of one edge, a register for its center, a register for the y-position of the other edge, and its Shape. The function `inside1` tests if the given x and y are within the paddle area. The single rule just keeps the `centerR` and `yposPlusHeight` registers in sync with the up/down counter `yposr`. Finally, the interface methods are defined as straightforward computations.

8 The `Ball` package

In the `Ball` package, the interface type `Ball` presents the `Shape` interface of the ball, the y-coordinate of its center, and its current horizontal direction `dir`, and accepts a `tick` command that causes the ball to update its internal status (position, velocity, etc.

The `mkBall` constructor is parameterized by a random number (which is actually a collection of wires, connected by `TopLevel` when the ball is instantiated; the value is updated every time `TopLevel` advances the random-number generator). The assertion at the top of the module ensures that the random number is

at least 2 bits wide, since this is what the module needs. Other parameters include the two paddles and two actions `lAct` and `rAct` which are to be performed when the ball hits the left and right walls, respectively. Looking at the ball instantiation in `TopLevel`, we see that the actual parameters for `lAct` and `rAct` are the actions to increment the left and right scores, respectively. This is an example of the use of `Action`s as first-class objects, i.e., we can compute with them in the same way we compute with numbers, strings, and other common data types. Here, we are able to pass two Actions as parameters to the `Ball` module constructor where they are invoked. Thus, we can separate out the question of *what* an Action does from the the question of *when/where* it is invoked.

The `r2` definition truncates the supplied random number to 2 bits (if it is longer) and this is then used to compute a random y-velocity `randV`. The next several lines instantiate the state of the ball: various registers holding the ball's coordinates, direction, velocity, speed change factor, shape, etc.

Next, we define the steps in updating the ball's state. In principle these could be computed in one step, but we have split it into multiple steps in order to meet timing on the FPGA at the desired clock speed. We have defined a sequence of `Action`s and combined them using the `seq` and `endseq` keywords into a "statement" `updateBallStmt`, which will take 5 clocks to perform. A FSM module implementing this statement is constructed by `mkFSM` and invoked by the `updateBallFSM.start` call which forms the action `updateBalls`. This in turn is invoked near the end of the `mkBall` module constructor. All this is an example of programming with various types as first-class objects.

The single rule in the module simply keeps track of the vertical center coordinate of the ball based on the top-edge coordinate `ball_y`. The first three interface method implementations are straightforward.

The fourth interface method, `tick`, is more interesting. Again it is implemented using the FSM mechanism during each vertical refresh, as it needs to take more than one clock cycle. It computes updates to the ball (position, velocity, etc.) taking into account bounces against the walls and paddles. Wall-bounces are easy; the associated `lAct` and `rAct` actions are performed if there is a bounce.

Paddle-bounces are more subtle. A bounce is detected as a momentary intersection between the ball rectangle and a paddle rectangle. Figure 2 shows the various ways in which such intersections can occur, characterized by which corners of the ball are inside the paddle rectangle. Remember that the ball can

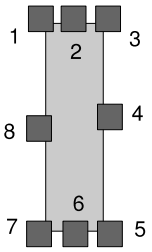

Figure 2: Ways in which the ball can "bounce" against a paddle

approach each paddle from any direction, since it can go around a paddle and bounce back at it from the wall. In the `tick` method code, the `paddleL.inside` and `paddleR.inside` method calls compute this characterization. In each case we get back 4 booleans representing the intersection of each of the four ball corners with the paddle, and a y-coordinate representing the vertical displacement between the paddle and the ball. The top-left, bottom-left, top-right and bottom-right corners of the ball are corners 0, 1, 2 and 3, respectively (see figure in comments in the code). For example, if `tpl_1(lpd)` and `tpl_3(lpd)` are True, then the top edge of the ball has intersected with the left paddle. If `R5.sel1` and `R5.sel2` are True, then the left edge of the ball has intersected with the right paddle.

To calculate changes in direction, we reason as follows. For example suppose the intersection is of the "nw" type. If the ball is moving downward, obviously we want it to "bounce" and start moving upwards. However, the ball could have already been moving upwards, along with the paddle, and they happen to intersect in the "nw" configuration. In this case, we don't want to change the direction of the ball; it should continue moving upwards. Similar reasoning applies for each of the other intersection configurations and each possible pre-bounce ball direction. This logic is encapsulated in the `bounce` function, defined later in the package, and in the two calls to it in the `dirx` and `diry` definitions. The `change_y` definition computes the amount by which the vertical velocity of the ball should change.

After computing the new values of the ball state, the last three lines of the `tick` method actually perform the updates. Note that `updateBalls` just kicks off the previously computed sequence of actions, which will take several subsequent clocks. Further, the implicit condition of this `start` method will go False until the sequence of actions has completed. This will automatically propagate to any rule that invokes this `tick` method, and that rule, in turn, will not be enabled until the sequence has completed. This is an example of how a complex timing condition which spans several modules is taken care of automatically by the semantics of rules– the designer simply does not have to think about it.

9 The `Controller` package

Whew! The heavy lifting is over, Heartbreak Hill has been crossed, and it's mostly downhill from here.

The `mkController` module in the `Controller` package is responsible for updating the paddle positions either because they are autoplaying or in response to keystrokes. The `Controller` interface simply presents two booleans which will ultimately drive two lamps on the FPGA board indicating the current autoplay status of the left and right paddles, respectively. The `mkController` module constructor is given, as parameters, the interfaces to the keyboard, paddles and the ball. The `repeat` register is just used as clock-stretching counter to slow down the rate of updates to the paddles. The remaining registers remember whether the paddles are in autoplay mode, whether to update a paddle, in which direction to update a paddle, and whether a key stroke is a press or a release.

The first rule gets a keycode from the keyboard and decides what to do with it, if anything. The second rule is the clock-stretcher using `repeat`. The last rule actually performs the paddle update.

10 The `Decimal` and `Score` packages

These two packages implement decimal counters and a way to display decimal numbers, respectively. The `Decimal` package defines a decimal digit type as a 4-bit number. The `DecCounter` interface has a method to increment such a counter, and a method to retrieve the array of digits in the counter. The `mkDecCounter` module constructor has a register containing the array of digits. The interface is straightforward, using the `incr` function to increment the digits.

The `incr` function uses a locally-defined function `addC` that adds a carry-in to a digit, returning a pair (a 2-tuple) of results consisting of the carry-out and the resulting digit. Then, it uses the library `mapAccumL` function to apply `addC` at every location of the `xs` array, feeding the carry-out at each index into the carry-in at the next index. The initial carry-in is 1 (thus implementing an increment operation). It returns two results (in a 2-tuple), the final carry-out and the updated array of values. The function returns the latter (using the selector `sel2`).

The `Score` package defines the `LedDisplay` type to be 7 bits, for a 7-segment display. The constants `scoreLong` and `scoreShort` are defined in package `Global`, and represent the length and width of each of the seven segments in a 7-segment display. The function `ledDecode` specifies which segments to light up for each decimal digit. The `mkDigit` module constructor presents a `Shape` interface. It is parameterized with the digit value itself and the desired location (coordinates) of the display for the digit. It uses the `ledDecode` function to produce a mask for the seven segments (an array of 7 booleans). It creates a rectangle (with a `Shape` interface) for each of the seven segments, passing in the x- and y-coordinates of the top-left of each segment's rectangle, and the rectangles width and height. For example, for segment 0 (the horizontal bottom segment), the x-coordinate is `x` and x-width is `scoreLong`. Its y-coordinate is `y + 2*(scoreLong + scoreShort)` since it has two horizontal and two vertical segments above it. Its y-height is `scoreShort`.

`mkDigit` then defines a local function `maskShape` to "switch off" a shape depending on a mask value. It uses the `zipWith` function from the Array library to associate the seven mask booleans with the seven segments, and finally joins the resulting seven Shapes into the composite shape for the digit.

The `mkDisplay` module constructor creates a display for an array of n digits. It uses `for` statement to apply the `mkDigit` module constructor once for each j in the range 0 to $n-1$, and then uses `joinManyShapes` to combine them into the required composite shape.

The `mkScore` module constructor takes a decimal counter and makes a display for it, using `mkDisplay`. Note that it is parameterized to handle an arbitrary number of digits (n).

11 Interfacing to external inputs: `Switch` and `Kbd`

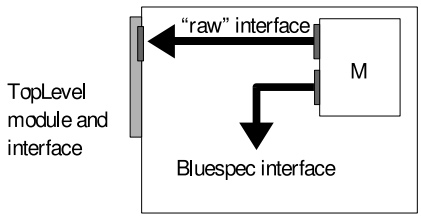

Figure 3: A module M with external connections.

These two packages interface to the outside world, in particular to the switches and PS2 keyboard attached to the FPGA. In both cases, the general strategy we follow is illustrated in Figure 3. A module M with external connections exports two interfaces, a "raw" interface that is propagated all the way to the top-level interface and is connected to the external wires, and a normal Bluespec SystemVerilog interface by which it serves the rest of the Bluespec SystemVerilog design. The `mkSwitch` module constructor is a trivial example of this.

The `Kbd` module is a bit more complex and illustrates how Bluespec SystemVerilog and Verilog components can be mixed in a design. It is actually a wrapper around an external piece of IP, an off-the-shelf Verilog module written by someone else to perform PS2 keyboard signal parsing. The `VKbd` interface describes the connections to the external Verilog module, and the `mkVKbd` module constructor uses the Bluespec-Verilog interface (BVI) to import a Verilog implementation for that module (`kbscan`), in the `kbscan.v` file (not shown in full). The module constructor in the import statement specifies how to connect the Bluespec SystemVerilog `VKbd` interface to the Verilog module wires, and some information for the Bluespec

SystemVerilog compiler indicating constraints, if any, on scheduling the interface methods from inside Bluespec SystemVerilog.

The `mkKbd` module constructor instantiates the `mkVKbd` Verilog wrapper module, and defines the interfaces.

12 Putting it all together

The `pong.v` file is the very top-level of the design. It is a simple piece of structural Verilog that simply instantiates the Bluespec SystemVerilog top level using the `mkTopLevel` module constructor. It also instantiates a Verilog reset-generation module from `reset.v`.

13 Conclusion

This document has described a complete, non-trivial application in Bluespec SystemVerilog in some detail. It illustrates many of the benefits of designing with Bluespec SystemVerilog:

- *Atomic Rules:* The whole presentation is remarkably free of any discussion of race conditions or questions of the form "On what clock was this signal supposed to be asserted/sampled?". Such questions are often at the forefront when coding in Verilog and are responsible for substantial verification effort. Instead, we just ask if each rule expresses the right behavior when viewed in isolation, and we can be confident that the generated code will preserve this reasoning. The logic of ball movement and bounces with respect to walls and paddles; the logic of video signal generation in response to the moving ball and paddles; all these are non-trivial, and the designer can focus on such these core issues rather than worrying about nitty-gritty timing and module interfacing details.

  There were a few places we were concerned about timing. In `VGACore` we were generating a video signal from some rules, and we placed assertions to ensure that these rules fired whenever we expected them to so that the video signal had the right synchronicity. These assertions are checked statically by the Bluespec SystemVerilog compiler.

  In `Ball` we needed to perform a number of dependent updates that would not fit into a single clock, and so we split them up into a sequence of steps, each of which could fit into a clock.

- *Advanced Module Composition:* This concerns the ability to look at modules as true black boxes (because they capture not only connectivity but also behavior), and the the ability to express systematic structure succinctly through iteration and recursion. It manifests itself in several ways:

  - *Interface methods fit organically into atomic rules:* Interface methods are invoked from rules. Interface methods have implicit conditions, and can perform actions. These aspects fit smoothly and seamlessly into rule conditions and rule actions. Indeed, an interface action method can be seen as a piece of a rule that can fit into several other rules, wherever that method is invoked.
  - *First-class functions and other constructs:*

    In Bluespec SystemVerilog, interfaces are first-class objects. This made it possible to write `Shape`-transforming functions, i.e., functions that took one or more `Shape` interfaces and returned other `Shape` interfaces. It allowed us easily to combine all the different shapes on the screen into one composite shape.

In `Ball`, where we split a number of dependent updates into a sequence of steps, the fact that `Action`s are first-class objects which can be combined into statements made it easy. The `StmtFSM` library completely automated the sequencing of these steps. This makes the retuning of pipelines quite easy.

Similarly, in Bluespec SystemVerilog, modules and rules (indeed any type) are also first-class objects, although the capability to compute with rules is not exploited in this application.

In `Decimal`, we defined a function to do a carry-add on a single decimal digit, and then simply used a library higher-order function, `mapAccumL`, to apply this to each digit in a whole array of digits. In `Score`, we defined a function to create a module to display a decimal digit, and then simply used a `for`-statement to initialise an array of such display modules for an array of digits.

These capabilities make it almost trivial to change the shapes of objects on the screen, to add extra digits to the scores, or to add more objects on the screen (e.g., extra balls, or "spoiler" islands, stationary or moving, which modify ball trajectories).

- *Polymorphism and parameterization:* In several cases, an interface or a module constructor or a function was parameterized by a *data type*, or by static parameters such as bit widths, etc. This made it easy to change the code, to reuse the code, and to use components from the Bluespec SystemVerilog library.
- *Automatic genereneration of control circuits and resource management:* Rules can interfere with each other, either because they compete in updating the same state, or because they compete in access to a method in another module. The Bluespec SystemVerilog compiler manages all this interaction, generating the appropriate hardware arbitration and multiplexing.

Most importantly, all this is *synthesizable* into RTL code that is competitive with hand-coded RTL.

A Source file `Global.bsv`

```
import VGACore::*;

Integer border;
border = 10;

Integer paddleDistFromWall;
paddleDistFromWall = 64;

Integer paddleWidth;
paddleWidth = 16;

Integer paddleHeight;
paddleHeight = 80;

Integer paddleEdgeDist;
paddleEdgeDist = 20 + border;

Integer ballWidth;
ballWidth = paddleWidth - 1;

Integer ballHeight;
ballHeight = div(ballWidth, 2) + 2;

Integer scoreLong;
scoreLong = 40;

Integer scoreShort;
scoreShort = 8;

typedef 3 NScoreDigits;

Integer scoreWallDist;
scoreWallDist = 100;

Integer scoreLx;
scoreLx = xMin + scoreWallDist;

Integer scoreRx;
scoreRx = xMax - (scoreWallDist + valueOf(NScoreDigits) * (scoreLong + scoreShort));

Integer scoreY;
scoreY = 30;

VGATiming vgaTiming;
vgaTiming = vga1280x480;

Integer hSize;
hSize = 1280;

Integer vSize;
vSize = 480;

Integer preScale;
preScale = 1;

Integer xMin;
```

```
xMin = 2 * border;

Integer xMax;
xMax = hSize - 2 * border;

Integer yMin;
yMin = border;

Integer yMax;
yMax = vSize - border;

typedef 12 XCoordSize;
typedef 11 YCoordSize;

typedef union tagged {
    Bit#(XCoordSize) XCoord;
} XCoord deriving (Literal, Eq, Ord, Arith, Bits, Bitwise);

typedef union tagged {
    Bit#(YCoordSize) YCoord;
} YCoord deriving (Literal, Eq, Ord, Arith, Bits, Bitwise);

typedef XCoord XSize;
typedef YCoord YSize;

function YCoord shiftY(YCoord y, Nat s);
  YCoord r;
  case (y) matches
    tagged YCoord .yc:
      r = YCoord (signedShiftRight(yc, s));
  endcase
  return r;
endfunction

function Bool posY(YCoord yc);
  Bool r;
  case (yc) matches
    tagged YCoord .x:
      r = x[fromInteger(valueOf(YCoordSize) - 1):fromInteger(valueOf(YCoordSize) - 1)] == 0;
  endcase
  return r;
endfunction

function Bool posX(XCoord xc);
  Bool r;
  case (xc) matches
    tagged XCoord .x:
      r = x[fromInteger(valueOf(XCoordSize) - 1):fromInteger(valueOf(XCoordSize) - 1)] == 0;
  endcase
  return r;
endfunction

function YCoord limitY(YCoord yc);
  Bit#(YCoordSize) limit = fromInteger(ballHeight - 4);
  YCoord r;
  case (yc) matches
    tagged YCoord .y:
      r = signedLT(y, negate(limit)) ? YCoord(negate(limit)) :
          (signedLT(limit, y) ? YCoord(limit) : yc);
```

```
  endcase
  return r;
endfunction
```

B Source file TopLevel.bsv

```
import List::*;
import LFSR::*;
import VGACore::*;
import Global::*;
import LedDecoder::*;
import Controller::*;
import KbdV::*;
import Switch::*;
import Border::*;
import Paddle::*;
import Ball::*;
import Shape::*;
import Score::*;
import Color::*;
import Decimal::*;

interface TopLevel;
    method Bit#(1) hsync();
    method Bit#(1) vsync();
    method Bit#(2) red();
    method Bit#(2) green();
    method Bit#(2) blue();
    method RawKbd rawkbd();
    method RawSwitch rawsw1();
    method RawSwitch rawsw2();
    method Bit#(1) aL();
    method Bit#(1) aR();
endinterface: TopLevel

Integer paddleLXMin;
paddleLXMin = xMin + paddleDistFromWall;

Integer paddleRXMin;
paddleRXMin = xMax - paddleDistFromWall - paddleWidth;

(* synthesize, always_ready, always_enabled *)
module mkTopLevel(TopLevel);
  Tuple2 #(RawKbd, Kbd) rawKbd_kbd();
  mkKbd the_kbd(rawKbd_kbd);
  Kbd kbd = tpl_2(rawKbd_kbd);

  Tuple2 #(RawSwitch, Switch) rawSwitch_switch1();
  mkSwitch the_switch1(rawSwitch_switch1);
  Switch sw1 = tpl_2(rawSwitch_switch1);

  Tuple2 #(RawSwitch, Switch) rawSwitch_switch2();
  mkSwitch the_switch2(rawSwitch_switch2);
  Switch sw2 = tpl_2(rawSwitch_switch2);

  LFSR#(Bit#(32)) lfsr();
  mkLFSR_32 the_lfsr(lfsr);
```

```
  DecCounter#(NScoreDigits) scoreL();
  mkDecCounter the_scoreL(scoreL);

  DecCounter#(NScoreDigits) scoreR();
  mkDecCounter the_scoreR(scoreR);

  Shape dispL();
  mkScore#(scoreL, fromInteger(scoreRx), fromInteger(scoreY)) the_dispL(dispL);

  Shape dispR();
  mkScore#(scoreR, fromInteger(scoreLx), fromInteger(scoreY)) the_dispR(dispR);

  VGACore#(XCoord, YCoord) vgaCore();
  mkVGACore#(preScale, vgaTiming) the_vgaCore(vgaCore);

  Shape border();
  mkBorder the_border(border);

  Paddle paddleL();
  mkPaddle#(paddleLXMin) the_paddleL(paddleL);

  Paddle paddleR();
  mkPaddle#(paddleRXMin) the_paddleR(paddleR);

  Ball ball();
  mkBall#(lfsr.value, paddleL, paddleR, scoreL.inc, scoreR.inc) the_ball(ball);

  Controller controller();
  mkController#(kbd, paddleL, paddleR, ball) the_controller(controller);

  Reg#(Color) color();
  mkRegU the_color(color);

//flipCol col b =  modShapeVis(\ c -> b && c != cNone ? col <^> c : c);
  function Shape flipCol(Color col, Bool b, Shape s);
    function Color f(Color c);
      return  (b && c != cNone ? colorXOr(col, c) : c);
    endfunction
    return modShapeVis(f,s);
  endfunction

//flipBCol col b =  modShapeVis(\ c -> b ? col <^> c : c);
  function Shape flipBCol(Color col, Bool b, Shape s);
    function Color f(Color c);
      return  (b ? colorXOr(col, c) : c);
    endfunction
    return modShapeVis(f,s);
  endfunction

  Shape shapes[6];

  shapes[0] = flipCol(makeRGB(2, 0, 1), sw1.value, border); // border'
  shapes[1] = flipCol(makeRGB(1, 1, 3), sw1.value, ball.shape); // ball'
  shapes[2] = flipCol(cYellow, controller.autoPlayL, paddleL.shape); // padL
  shapes[3] = flipCol(cYellow, controller.autoPlayR, paddleR.shape); // padR
  shapes[4] = dispL;
  shapes[5] = dispR;
```

```
  Shape pict =  joinManyShapes(shapes);
  Shape pictN =  flipBCol(cWhite, sw2.value, pict);
  function Color f (Color c);
    return (vgaCore.blank ? cNone : c);
  endfunction
  Shape pictBl =  modShapeVis(f, pictN);

  (* no_implicit_conditions, fire_when_enabled *)
  rule make_color
   (!(vgaCore.frameTick));
      (pict.newPos)(vgaCore.hPos, vgaCore.vPos);
      lfsr.next;
      color <= pictBl.color;
  endrule

  (* fire_when_enabled *)
  rule tick
   (vgaCore.frameTick);
      ball.tick;
  endrule

  method hsync(); return (pack(vgaCore.not_hsync));
  endmethod: hsync

  method vsync(); return (pack(vgaCore.not_vsync));
  endmethod: vsync

  method red(); return (getRed(color));
  endmethod: red

  method green(); return (getGreen(color));
  endmethod: green

  method blue(); return (getBlue(color));
  endmethod: blue

  method rawkbd(); return tpl_1(rawKbd_kbd);
  endmethod: rawkbd

  method rawsw1(); return tpl_1(rawSwitch_switch1);
  endmethod: rawsw1

  method rawsw2(); return tpl_1(rawSwitch_switch2);
  endmethod: rawsw2

  method aL(); return (pack(controller.autoPlayL));
  endmethod: aL

  method aR(); return (pack(controller.autoPlayR));
  endmethod: aR

endmodule: mkTopLevel
```

C Source file VGACore.bsv

```
//
// either one of the sync signals
//
//       |<-- Active Region ---->|<----------- Blanking Region ---------->|
//       |      (Pixels)         |                                        |
//       |                       |                                        |
//       |                       |                                        |
//   ----+--------- ... ---------+-------------             --------------+---
//   |   |                       |            |             |             |
//   |   |                       |<--Front    |<---Sync     |<---Back     |
//   |   |                       |    Porch-->|     Time--->|    Porch--->|
// ---   |                       |            ---------------             |
//       |                       |                                        |
//       |<------------------- Period ------------------------------------>|
//

// Sync info for horizontal or vertical
typedef struct {
                  Integer activeSize;
                  Integer syncStart;
                  Integer syncEnd;
                  Integer totalSize;
              } VGAHVTiming;

// Sync info for VGA
typedef struct {
                  VGAHVTiming h;
                  VGAHVTiming v;
              } VGATiming;

// Better?
// 640  650  710  762    480  482  488  500
// 800  812  888  952    600  602  610  626

VGATiming vga640x480;
vga640x480 = VGATiming { h : VGAHVTiming { activeSize : 640,
                                           syncStart : 664,
                                           syncEnd : 760,
                                           totalSize : 800},
                         v : VGAHVTiming { activeSize : 480,
                                           syncStart : 490,
                                           syncEnd : 494,
                                           totalSize : 526}};

VGATiming vga1280x480;
vga1280x480 = VGATiming { h : VGAHVTiming { activeSize : 1280,
                                            syncStart : 1328,
                                            syncEnd : 1520,
                                            totalSize : 1600},
                          v : VGAHVTiming { activeSize : 480,
                                            syncStart : 490,
                                            syncEnd : 494,
                                            totalSize : 526}};

function VGATiming hzToTiming(Integer hz);
  function Integer nsCycles(Integer x);
```

```
    return (div((x * hz), 1000000000));
  endfunction: nsCycles
  function Integer usCycles(Integer x);
    return (div((x * hz), 1000000));
  endfunction: usCycles

  Integer hTotalSize;
  hTotalSize =  nsCycles(31770);

  return (VGATiming { h : VGAHVTiming { activeSize : nsCycles(25170),
                                        syncStart : nsCycles(26110),
                                        syncEnd : nsCycles(29880),
                                        totalSize : hTotalSize},
                      v : VGAHVTiming { activeSize : div(usCycles(15250), hTotalSize),
                                        syncStart : div(usCycles(15700), hTotalSize),
                                        syncEnd : div(usCycles(15764), hTotalSize),
                                        totalSize : div(usCycles(16784), hTotalSize)}});
endfunction: hzToTiming

function VGATiming sizeToTiming(Integer hSize, Integer vSize);
  return (VGATiming { h : VGAHVTiming { activeSize : hSize,
                                        syncStart : div((hSize * 856), 800),
                                        syncEnd : div((hSize * 880), 800),
                                        totalSize : div((hSize * 982), 800)},
                      v : VGAHVTiming { activeSize : vSize,
                                        syncStart : div((vSize * 602), 600),
                                        syncEnd : div((vSize * 610), 600),
                                        totalSize : div((vSize * 626), 600)}});
endfunction: sizeToTiming

interface VGACore #(type hCoord, type vCoord);
    method Bool not_hsync();
    method Bool not_vsync();
    method Bool blank();
    method hCoord hPos();
    method vCoord vPos();
    method Bool lineTick();
    method Bool frameTick();
endinterface: VGACore

module mkVGACore#(Integer preScale, VGATiming vt)(VGACore#(hCoord, vCoord))
  provisos (Bits#(hCoord, hs),
            Literal#(hCoord),
            Eq#(hCoord),
            Arith#(hCoord),
            Bits#(vCoord, vs),
            Literal#(vCoord),
            Eq#(vCoord),
            Arith#(vCoord));

    Reg#(hCoord) hPosR();
    mkReg#(0) the_hPosR(hPosR);
    Reg#(vCoord) vPosR();
    mkReg#(0) the_vPosR(vPosR);
    Reg#(Bool) hVisible();
    mkReg#(True) the_hVisible(hVisible);
    Reg#(Bool) vVisible();
    mkReg#(True) the_vVisible(vVisible);
    Reg#(Bool) not_hsyncR();
```

```
mkReg#(True) the_not_hsyncR(not_hsyncR);
Reg#(Bool) not_vsyncR();
mkReg#(True) the_not_vsyncR(not_vsyncR);
Reg#(Bit#(4)) scale();
mkReg#(0) the_scale(scale);

hCoord hSize;
vCoord vSize;
hCoord hSyncStart;
vCoord vSyncStart;
hCoord hSyncEnd;
vCoord vSyncEnd;
hCoord hTotal;
vCoord vTotal;
Bool hTickLocal;
Bool vTickLocal;
Bool hTickExternal;
Bool vTickExternal;

hSize =  fromInteger(vt.h.activeSize);
vSize =  fromInteger(vt.v.activeSize);
hSyncStart =  fromInteger(vt.h.syncStart);
vSyncStart =  fromInteger(vt.v.syncStart);
hSyncEnd =  fromInteger(vt.h.syncEnd);
vSyncEnd =  fromInteger(vt.v.syncEnd);
hTotal =  fromInteger(vt.h.totalSize);
vTotal =  fromInteger(vt.v.totalSize);
hTickLocal =  hPosR == hTotal;
vTickLocal =  hTickLocal && (vPosR == vTotal);
hTickExternal =  hPosR == (hSize + 1);
vTickExternal =  hTickExternal && (vPosR == (vSize + 1));

(* no_implicit_conditions, fire_when_enabled *)
rule tick (preScale != 1);
    scale <= (scale == 0 ? fromInteger(preScale - 1) : scale - 1);
endrule: tick

(* no_implicit_conditions, fire_when_enabled *)
rule hclk ((preScale == 1) || (scale == 0));
    hPosR <= (hTickLocal ? 0 : hPosR + 1);
    hVisible <= ((hPosR != hSize) && (hTickLocal || hVisible));
endrule: hclk

(* no_implicit_conditions, fire_when_enabled *)
rule vclk (hTickLocal);
    vPosR <= (vTickLocal ? 0 : vPosR + 1);
    vVisible <= ((vPosR != vSize) && (vTickLocal || vVisible));
endrule: vclk

(* no_implicit_conditions, fire_when_enabled *)
rule hsyncOn (hPosR == hSyncStart);
    not_hsyncR <= False;
endrule: hsyncOn

(* no_implicit_conditions, fire_when_enabled *)
rule hsyncOff (hPosR == hSyncEnd);
    not_hsyncR <= True;
endrule: hsyncOff
```

```
    (* no_implicit_conditions, fire_when_enabled *)
    rule vsyncOn (vPosR == vSyncStart);
        not_vsyncR <= False;
    endrule: vsyncOn

    (* no_implicit_conditions, fire_when_enabled *)
    rule vsyncOff (vPosR == vSyncEnd);
        not_vsyncR <= True;
    endrule: vsyncOff

    method hPos();
      return (hPosR);
    endmethod: hPos
    method vPos();
      return (vPosR);
    endmethod: vPos
    // blank video outside of visible region
    method blank();
      return (!(hVisible && vVisible));
    endmethod: blank
    method not_hsync();
      return (not_hsyncR);
    endmethod: not_hsync
    method not_vsync();
      return (not_vsyncR);
    endmethod: not_vsync
    method lineTick();
      return (hTickExternal && vVisible);
    endmethod: lineTick
    method frameTick();
      return (vTickExternal);
    endmethod: frameTick

endmodule: mkVGACore
```

D Source file `Color.bsv`

```
// no explicit ‘export lines here so as to export everything

typedef  Bit#(2) Val;

typedef struct {
    Val r;
    Val g;
    Val b;
} Color deriving (Eq, Bits);

function Bool colorEQ(Color c1, Color c2);
  return ((c1.r == c2.r) && (c1.g == c2.g) && (c1.b == c2.b));
endfunction: colorEQ

function Bit#(6) colorPack(Color c);
  return ({ c.r, c.g, c.b });
endfunction: colorPack

function Color makeRGB(Val rr, Val gg, Val bb);
  return (Color { r : rr, g : gg, b : bb });
```

```
endfunction: makeRGB

Color cNone;
cNone = makeRGB(0, 0, 0);

Color cRed;
cRed = makeRGB(3, 0, 0);

Color cGreen;
cGreen = makeRGB(0, 3, 0);

Color cBlue;
cBlue = makeRGB(0, 0, 3);

Color cYellow;
cYellow = makeRGB(3, 3, 0);

Color cPurple;
cPurple = makeRGB(3, 0, 3);

Color cMagenta;
cMagenta = makeRGB(0, 3, 3);

Color cWhite;
cWhite = makeRGB(3, 3, 3);

function Color colorOr (Color c1, Color c2);
  return (Color { r : (c1.r | c2.r),
                  g : (c1.g | c2.g),
                  b : (c1.b | c2.b) });
endfunction: colorOr

function Color colorXOr (Color c1, Color c2);
  return (Color { r : (c1.r ^ c2.r),
                  g : (c1.g ^ c2.g),
                  b : (c1.b ^ c2.b) });
endfunction: colorXOr

function Val getRed(Color c);
return (c.r);
endfunction: getRed

function Val getGreen(Color c);
return (c.g);
endfunction: getGreen

function Val getBlue(Color c);
return (c.b);
endfunction: getBlue
```

E Source file `Shape.bsv`

```
import List::*;
import Global::*;
```

```
import Color::*;

interface Shape;
    method Action newPos(XCoord x1, YCoord x2);
    method Color color;
endinterface: Shape

Shape emptyShape;
emptyShape =
  (interface Shape
     method newPos(x,y);
       return noAction;
     endmethod
     method color();
       return cNone;
     endmethod
   endinterface);

function Shape joinShapes(Shape s1, Shape s2);
return (interface Shape
          method newPos (x,y);
            action
              s1.newPos(x, y);
              s2.newPos(x, y);
            endaction
          endmethod
          method color();
            return ( colorOr (s1.color,s2.color) );
          endmethod
        endinterface);
endfunction: joinShapes

function Action visCheck(Reg#(Bool) r, a lo, a v, a hi)
  provisos (Eq#(a));
  return (action
            r <= ((v != hi) && ((v == lo) || r));
          endaction);
endfunction: visCheck

module mkRectangle#(XCoord xl, XSize xs, YCoord yl, YSize ys, Color col)(Shape);
    Reg#(Bool) xVis();
    mkRegU the_xVis(xVis);
    Reg#(Bool) yVis();
            mkRegU the_yVis(yVis);

            XCoord xh;
            YCoord yh;
            xh =  xl + xs;
            yh =  yl + ys;

            method newPos(x, y);
              action
                visCheck(xVis, xl, x, xh);
                visCheck(yVis, yl, y, yh);
              endaction
            endmethod: newPos
            method color();
              return (xVis && yVis ? col : cNone);
            endmethod: color
```

```
endmodule: mkRectangle

function Shape modShapeVis(function Color f(Color x1), Shape s);
  return (interface Shape
    method newPos();
              return (s.newPos);
    endmethod: newPos
    method color();
              return (f(s.color));
    endmethod: color
  endinterface: Shape);
endfunction: modShapeVis

function Shape joinManyShapes(List#(Shape) shapes);
  return (foldr(joinShapes, emptyShape, shapes));
endfunction: joinManyShapes
```

F Source file `Border.bsv`

```
import Global ::*;
import Shape ::*;
import Color::*;

Color col;
col = cBlue;

module mkBorder(Shape);
     Shape rect();
     mkRectangle#(fromInteger(xMin),
                          fromInteger(xMax - xMin),
                          fromInteger(yMin),
                          fromInteger(yMax - yMin),
                          col) the_rect(rect);
     return(modShapeVis(colorXOr(col), rect));
endmodule
```

G Source file `Counter.bsv`

```
// Counter with start value, and upper and lower limits
// possible to increment and decrement

interface Counter#(parameter type a);
    method a get;
    method Action set(a x1);
    method Action inc_dec(Bool x1); // increment if True, otherwise decrement
endinterface: Counter

module mkCounter#(parameter a lower, parameter a start, parameter a upper)(Counter#(a))
  provisos (Bits #(a,b) , Eq #(a), Literal #(a), Arith #(a));
  Reg#(a) value_reg();
  mkReg#(start) the_value_reg(value_reg);
  method a get();
    return value_reg;
```

```
  endmethod: get
  method Action set(a x1);
    action
      value_reg <= x1;
    endaction
  endmethod: set
  method Action inc_dec(Bool up);
    action
      value_reg <= (up && (value_reg != upper)) ? (value_reg + 1) :
                   ((!up && (value_reg != lower)) ?
                   (value_reg - 1) :
                   (value_reg));
    endaction
  endmethod: inc_dec

endmodule: mkCounter
```

H Source file Paddle.bsv

```
import Global   ::*;
import Counter          ::*;
import Shape    ::*;
import Color::*;

XSize paddleWidthC;
paddleWidthC = fromInteger(paddleWidth);

YSize paddleHeightC;
paddleHeightC = fromInteger(paddleHeight);

YCoord paddleLowC;
paddleLowC = fromInteger(paddleEdgeDist);

YCoord paddleHighC;
paddleHighC = fromInteger(vSize - paddleHeight - paddleEdgeDist);

YCoord paddleStartC;
paddleStartC = fromInteger(div((vSize- paddleHeight), 2));

interface Paddle;
    method Shape shape;
    method YCoord center;
    method Action inc_dec(Bool x1);
    method Tuple5#(Bool, Bool, Bool, Bool, YSize) insideit (XCoord x1, XCoord x2, YCoord x3, YCoord x4, YSize x5);
endinterface: Paddle

module mkPaddle#(parameter Integer xpos)(Paddle);
  Counter#(YCoord) yposr();
  mkCounter#(paddleLowC, paddleStartC, paddleHighC) the_yposr(yposr);
  Reg#(YCoord) centerR();
  mkRegU the_centerR(centerR);
  Shape paddleRect();
  mkRectangle#(fromInteger(xpos),
               paddleWidthC,
```

```
              yposr.get,
              paddleHeightC,
              cRed) the_paddleRect(paddleRect);
  Reg#(YCoord) yposPlusHeight();
  mkRegU the_yposPlusHeight(yposPlusHeight);

  function Bool insideit1(XCoord x, YCoord y);
  return (x > fromInteger(xpos) &&
          x < fromInteger(xpos+ paddleWidth) &&
          y > yposr.get &&
          y < yposPlusHeight);
  endfunction: insideit1

  rule rule1Paddle (True);
    centerR <= yposr.get + (paddleHeightC>> 1);
    yposPlusHeight <= yposr.get + paddleHeightC;
  endrule

  method shape();
    return paddleRect;
  endmethod: shape
  method center();
    return centerR;
  endmethod: center
  method inc_dec(up);
   action
    yposr.inc_dec(up);
   endaction
  endmethod: inc_dec

  method insideit(x0, x1, y0, y1, dy);
    return tuple5( (insideit1(x0, y0)),
                   (insideit1(x0, y1)),
                   (insideit1(x1, y0)),
                   (insideit1(x1, y1)),
                   ((y0 - yposr.get) - ((paddleHeightC- dy)>> 1))
                 );
  endmethod: insideit

endmodule: mkPaddle
```

I Source file `Ball.bsv`

```
import StmtFSM  ::*;
import Global   ::*;
import Paddle   ::*;
import Border   ::*;
import Shape    ::*;
import Color::*;
//import List::*;
import ListN::*;

typedef Tuple5#(Bool, Bool, Bool, Bool, YSize) PaddleInsideTyp;
```

```
interface Ball;
    method Shape shape;
    method YCoord center;
    method Bool dir;
    method Action tick;
endinterface: Ball

Color col;
col = cYellow;

XSize ballWidthC;
ballWidthC = fromInteger(ballWidth);

YSize ballHeightC;
ballHeightC = fromInteger(ballHeight);

// bounce top   bottom up/down      => up/down
// bounce right left   right/left   => right/left
// The result is the new direction

function Bool bounce (Bool x , Bool y , Bool z);
  Tuple3#(Bool,Bool,Bool) xyz;
  xyz = tuple3(x,y,z);
  case (xyz) matches
   {False , False , False } : return  False; // no hit
   {False , False , True  } : return  True;  // no hit
   {False , True  , False } : return  False; // dir same as perpendicular
   {False , True  , True  } : return  False; // dir opp to perpendicular
   {True  , False , False } : return  True;  // dir opp to perpendicular
   {True  , False , True  } : return  True;  // dir same as perpendicular
   {True  , True  , False } : return  False; // dir orthog to perpendicular
   {True  , True  , True  } : return  True;  // dir orthog to perpendicular
   default                  : return  False; // all cases specified, not needed
  endcase
endfunction: bounce

module mkBall#(Bit#(n) random,
               Paddle paddleL,
               Paddle paddleR,
               Action lAct,
               Action rAct)(Ball)
  provisos (Add#(x,2,n));

  Reg#(YCoord) centerR();
  mkRegU the_centerR(centerR);

  Reg#(XCoord) ball_x();
  mkReg#(fromInteger(div(hSize, 2))) the_ball_x(ball_x);

  Reg#(YCoord) ball_y();
  mkReg#(fromInteger(div(vSize, 2))) the_ball_y(ball_y);

  Reg#(XCoord) ball_x_r();
  mkRegU the_ball_x_r(ball_x_r);

  Reg#(YCoord) ball_y_b();
  mkRegU the_ball_y_b(ball_y_b);

  Reg#(XCoord) ball_vx();
```

```
mkReg#(7) the_ball_vx(ball_vx);

Reg#(YCoord) ball_vy();
mkReg#(1) the_ball_vy(ball_vy);

Shape ballRect();
mkRectangle#(ball_x, ballWidthC, ball_y, ballHeightC, col) the_ballRect(ballRect);

Reg#(YCoord) change_y();
mkReg#(0) the_change_y(change_y);

Reg#(Bool) ball_dx();
mkRegU the_ball_dx(ball_dx);

Reg#(Bool) ball_dy();
mkRegU the_ball_dy(ball_dy);

// break updating into several steps to meet timing

// The following uses only one register, but more muxes and logic

// Declaring a new name and making it a synonym for an existing
// interface so no () needed for the following two lines
Reg#(YCoord) ball_vy2;
ball_vy2 =  asReg(ball_vy);
Reg#(YCoord) ball_vy3;
ball_vy3 =  asReg(ball_vy);

// steps in updating the ball
Stmt updateBallStmt =
 seq
    action
      ball_vx<= (posX(ball_vx)== ball_dx ? ball_vx : negate(ball_vx));
      ball_vy2<= (posY(ball_vy)== ball_dy ? ball_vy : negate(ball_vy));
    endaction
    action
      // random velocity change
      Bit#(2) r2;
      YCoord randV;
      // truncate is defined in both Prelude and UInt, must specify
      r2 = Prelude::truncate(random);
      case (r2)
        2'b00    :  randV = 1;
        2'b01    :  randV = -1;
        default  :  randV = 0;
      endcase
      ball_vy3 <= ball_vy2 + shiftY(change_y, 3) + ((change_y != 0) ? randV : 0);
    endaction
    action
      ball_vy<= limitY(ball_vy3);
    endaction
    action
      ball_x <= ball_x + ball_vx; ball_y <= ball_y + ball_vy;
    endaction
    action
      ball_x_r <= ball_x + ballWidthC; ball_y_b <= ball_y + ballHeightC;
    endaction
  endseq;
```

```
FSM updateBallFSM();
mkFSM#(updateBallStmt) the_FSM(updateBallFSM);

Action updateBalls = updateBallFSM.start;

Reg#(PaddleInsideTyp) lpd();
mkRegU the_lpd(lpd);

Reg#(PaddleInsideTyp) rpd();
mkRegU the_rpd(rpd);

Stmt tickActionStmt =
  seq
           action
             // Left Paddle
             lpd <= paddleL.insideit(ball_x, ball_x_r, ball_y, ball_y_b, ballHeightC);
             // Right Paddle
             rpd <= paddleR.insideit(ball_x, ball_x_r, ball_y, ball_y_b, ballHeightC);
           endaction
           action
             // New Direction for Y
                    //bounce (b00 || b10),
                    //       (b01 || b11),
                    //       bounce (a00 || a10),
                    //              (a01 || a11), ...
                    //              ()
             Bool diry;
             diry = bounce( (tpl_1(rpd) || tpl_3(rpd)),
                            (tpl_2(rpd) || tpl_4(rpd)),
                    bounce( (tpl_1(lpd) || tpl_3(lpd)),
                            (tpl_2(lpd) || tpl_4(lpd)),
                    ((ball_y <= (fromInteger(yMax- ballHeight))) &&       // down if too large
                               ((ball_y < (fromInteger(yMin))) || (posY(ball_vy))))
                                                                          //  up  if too small
                                 )
                          );

             // New direction for X
             // Need to know which wall for counting points
             // Need to know where on the paddle for changing vy
                    //bounce (a00 || a01),
                    //       (a10 || a11),
                    //       bounce (b00 || b01),
                    //              (b10 || b11), ...
                    //              ()

             Bool hitWallR;
             Bool hitWallL;
             hitWallR =  (ball_x > fromInteger(xMax- ballWidth));
             hitWallL =  (ball_x < fromInteger(xMin));

             Bool dirx;
             dirx = bounce( (tpl_1(rpd) || tpl_2(rpd)),
                            (tpl_3(rpd) || tpl_4(rpd)),
                    bounce( (tpl_1(lpd) || tpl_2(lpd)),
                            (tpl_3(lpd) || tpl_4(lpd)),
                                    ((!hitWallR) && (hitWallL || posX(ball_vx)))
                                  )
                          );
```

```
            change_y <= (((tpl_1(lpd) || tpl_2(lpd)) != (tpl_3(lpd) || tpl_4(lpd))) ?
                         tpl_5(lpd) :
                         (((tpl_1(rpd) || tpl_2(rpd)) != (tpl_3(rpd) || tpl_4(rpd))) ?
                          tpl_5(rpd) :
                          0));

            if (hitWallR)
              rAct;
            else
              noAction;

            if (hitWallL)
              lAct;
            else
              noAction;

            ball_dx <= dirx;
            ball_dy <= diry;

            updateBalls;

          endaction
    endseq;

  FSM tickActionFSM();
  mkFSM#(tickActionStmt) the_tickAction_FSM(tickActionFSM);

  Action tickAction  = tickActionFSM.start;

  rule rule1Ball (True);
    centerR <= ball_y + fromInteger(div(ballHeight, 2));
  endrule

  method shape();
    return ballRect;
  endmethod: shape
  method center();
    return centerR;
  endmethod: center
  method dir();
    return ball_dx;
  endmethod: dir
  method tick();
    action
      tickAction;
    endaction
  endmethod: tick

endmodule: mkBall
```

J Source file `Controller.bsv`

```
import KbdV ::*;
import GetPut ::*;
import Paddle ::*;
import Ball ::*;

interface Controller;
    method Bool autoPlayL;
    method Bool autoPlayR;
endinterface: Controller

module mkController#(parameter Kbd kbd,
     parameter Paddle paddleL,
     parameter Paddle paddleR,
     parameter Ball ball)(Controller);
  Reg#(UInt#(20)) repeatit();
  mkReg#(0) the_repeatit(repeatit);
  Reg#(Bool) autoL();
  mkReg#(True) the_autoL(autoL);
  Reg#(Bool) autoR();
  mkReg#(True) the_autoR(autoR);
  Reg#(Bool) doL();
  mkReg#(False) the_doL(doL);
  Reg#(Bool) upL();
  mkReg#(False) the_upL(upL);
  Reg#(Bool) doR();
  mkReg#(False) the_doR(doR);
  Reg#(Bool) upR();
  mkReg#(False) the_upR(upR);
  Reg#(Bool) releaseit();
  mkReg#(False) the_releaseit(releaseit);


  rule rule1Controller (True);
      ScanCode keycode;
      keycode <- kbd.get;
      case (keycode) matches
      tagged ScanCode ('hF0) :
begin
  releaseit<= True;
end
      tagged ScanCode ('h1C) :
begin
  doL<= !releaseit; upL<= False; releaseit<= False;
end
      tagged ScanCode ('h1A) :
begin
  doL<= !releaseit; upL<= True; releaseit<= False;
end
      tagged ScanCode ('h15) :
begin
  autoL<= (releaseit ? autoL : !autoL); releaseit<= False;
end
      tagged ScanCode ('h52) :
begin
  doR<= !releaseit; upR<= False; releaseit<= False;
end
```

```
      tagged ScanCode ('h4A) :
begin
  doR<= !releaseit; upR<= True; releaseit<= False;
end
      tagged ScanCode ('h5B) :
begin
  autoR<= (releaseit ? autoR : !autoR); releaseit<= False;
end
      default :
begin
  releaseit<= False;
end
      endcase
  endrule


  rule rule2Controller (True);
    repeatit <= (repeatit== 0 ? 110000 : repeatit- 1);
  endrule

  rule rule3Controller (repeatit== 0);
      if (doL)
        begin
  paddleL.inc_dec(upL);
        end
      else if (autoL&& !ball.dir)
        begin
          paddleL.inc_dec(ball.center> paddleL.center);
        end
      else noAction;
      if (doR)
        begin
  paddleR.inc_dec(upR);
        end
      else if (autoR&& ball.dir)
        begin
          paddleR.inc_dec(ball.center> paddleR.center);
        end
      else noAction;
  endrule

  method autoPlayL();
    return autoL;
  endmethod: autoPlayL
  method autoPlayR();
    return autoR;
  endmethod: autoPlayR

endmodule: mkController
```

K Source file `Decimal.bsv`

```
import ListN::*;
```

```
typedef  Bit#(4) DecDigit;

interface DecCounter#(parameter type n);
    method Action inc;
    method ListN#(n, DecDigit) getDigits;
endinterface: DecCounter

module mkDecCounter (DecCounter#(n));
    Reg#(ListN#(n, DecDigit)) digits();
    mkReg#(unpack(0)) the_digits(digits);

    method getDigits();
      return (digits);
    endmethod: getDigits

    method inc();
      return (action
                digits <= incr(digits);
              endaction);
    endmethod: inc
endmodule

function ListN#(n, DecDigit) incr (ListN#(n, DecDigit) xs);
  function addC (ci, x);
    return ((x + ci == 10) ? tuple2(1 , 0) : tuple2(0 , (x + ci)));
  endfunction
  let co_xsnew =  mapAccumL(addC, 1, xs);
  return tpl_2(co_xsnew);
endfunction: incr
```

L Source file `Score.bsv`

```
import ListN ::*;
import List ::*;
import Monad ::*;
import Global ::*;
import Shape ::*;
import Color ::*;
import LedDecoder ::*;
import Decimal::*;

Integer long;
long = 40;

Integer short;
short = 8;

Integer longs;
longs =  long- short;

XSize longsx; longsx = fromInteger(longs);
YSize longsy; longsy = fromInteger(longs);

module mkDigit#(parameter LedDigit digit, parameter XCoord x, parameter YCoord y)(Shape);
```

```
  function m#(Shape) mkRectangle2 (XCoord x, Integer xw, YCoord y, Integer yw, Color c)
            provisos(IsModule#(m));
            return mkRectangle(x, fromInteger(xw), y>> 1, (fromInteger(yw))>> 1, c);
          endfunction

  Shape seg6();
  mkRectangle2#(x, long, y, short, cWhite) the_seg6(seg6);
  Shape seg5();
  mkRectangle2#(x, short, y, long, cWhite) the_seg5(seg5);
  Shape seg4();
  mkRectangle2#(x+ longsx, short, y, long, cWhite) the_seg4(seg4);
  Shape seg3();
  mkRectangle2#(x, long, y+ longsy, short, cWhite) the_seg3(seg3);
  Shape seg2();
  mkRectangle2#(x, short, y+ longsy, long, cWhite) the_seg2(seg2);
  Shape seg1();
  mkRectangle2#(x+ longsx, short, y+ longsy, long, cWhite) the_seg1(seg1);
  Shape seg0();
  mkRectangle2#(x, long, y + 2 * longsy, short, cWhite) the_seg0(seg0);
  LedDecoder ldec();
  mkLedDecoder the_ldec(ldec);
  List#(Bool) mask;
  mask = toList(unpack(ldec.decode(digit)));

          List#(Shape) segs = newList(7);
          segs[0] = seg0;
          segs[1] = seg1;
          segs[2] = seg2;
          segs[3] = seg3;
          segs[4] = seg4;
          segs[5] = seg5;
          segs[6] = seg6;

          function Shape maskShape(Bool vis, Shape s);
            function Color f(Color c);
              return (vis ? c : cNone);
            endfunction
            return modShapeVis(f , s);
  endfunction
  Shape disp =  joinManyShapes(List::zipWith(maskShape, mask, segs));
  return(disp);
endmodule: mkDigit

function m#(Shape) mkScore(DecCounter#(n) cnt, XCoord x, YCoord y) provisos(IsModule#(m));
return (mkDisplay(cnt.getDigits, x, y));
endfunction: mkScore

module mkDisplay#(ListN#(n, Bit#(4)) digits, XCoord x, YCoord y)(Shape);
   Integer nv =  valueOf(n);
   Integer sep =  long+ short;
   XCoord right =  fromInteger((nv-1)* sep)+ x;

   function m#(Shape) f(Integer i) provisos (IsModule#(m));
      return mkDigit((toList(digits))[i], right - fromInteger(i * sep),
                     y);
   endfunction

   Shape glyphs[nv];
   for (Integer j = 0; j<nv; j=j+1)
```

```
    begin
      Shape s <- f(j);
      glyphs[j] = s;
    end

   return(joinManyShapes(glyphs));
endmodule: mkDisplay
```

M Source file Switch.bsv

```
import ConfigReg::*;

interface Switch;
    method Bool value();
endinterface: Switch

interface RawSwitch;
    method Action iput(Bool x1);
endinterface: RawSwitch

module mkSwitch (Tuple2 #(RawSwitch, Switch));
     Reg#(Bool) state();
     mkConfigRegU the_state(state);

     RawSwitch rawsw;
     rawsw = (interface RawSwitch
                method iput(x);
                  action
                    state <= x;
                  endaction
                endmethod: iput
              endinterface);
     Switch sw;
     sw = (interface Switch
             method value();
               return (!state);
             endmethod: value
           endinterface: Switch);
     return tuple2(rawsw,sw);
endmodule
```

N Source file KbdV.bsv

```
import GetPut::*;

typedef union tagged {
    Bit#(8) ScanCode;
} ScanCode deriving (Bits, Eq, Literal);

typedef Get#(ScanCode) Kbd;

interface RawKbd;
```

```
    method Action kbclk(Bool x1);
    method Action kbdata(Bool x1);
    method Bit#(8) key();
endinterface: RawKbd

module mkKbd(Tuple2#(RawKbd, Kbd));
     let mkVKbd1 = liftModule(mkVKbd);

     VKbd vkbd();
     mkVKbd1 thevkbd(vkbd);

     rule kbd_ack (vkbd.err == 1);
       fromPrimAction(vkbd.ack);
     endrule

     return(tuple2(
            interface RawKbd
      method kbclk(x);
               action
                  if (x) fromPrimAction(vkbd.clk);
       endaction
      endmethod: kbclk

              method kbdata(x);
               action
                  if (x) fromPrimAction(vkbd.dta);
       endaction
      endmethod: kbdata

              method key();
                  return (vkbd.key);
      endmethod: key
    endinterface: RawKbd
           ,
    interface Get
      method get() if (vkbd.rdy == 1 && vkbd.err == 0);
       actionvalue
fromPrimAction(vkbd.ack);
return(ScanCode(vkbd.key));
       endactionvalue
      endmethod: get
    endinterface: Get
           ));
endmodule: mkKbd

interface VKbd;
    method Bit#(8) key();
    method Bit#(1) rdy();
    method Bit#(1) err();
    method PrimAction ack();
    method PrimAction clk();
    method PrimAction dta();
endinterface: VKbd

import "BVI" kbscan = module mkVKbd(VKbd);
  clock clk;
  reset rstn;
  method key(o_byte);
  method rdy(o_byte_rdy);
```

```
  method err(o_byte_err);
  method ack(i_byte_ack);
  method clk(i_kbclk);
  method dta(i_kbdata);

  schedule (key,rdy,err,ack,clk,dta) <> (key,rdy,err,ack,clk,dta);


endmodule: mkVKbd
```

O Source file pong.v

```
module pong(CLK,
    hsync,
    vsync,
    red,
    green,
    blue,
    kbclk,
    kbdata,
    sw1,
    sw2,
    display);
  input CLK;
  output hsync;
  output vsync;
  output [1 : 0] red;
  output [1 : 0] green;
  output [1 : 0] blue;
  output [6 : 0] display;
  input kbclk;
  input kbdata;
  input sw1;
  input sw2;

  wire RST_N;

  wire al, ar;

  wire CLOCK;
  assign CLOCK = CLK;

  mkTopLevel toplevel(
.CLK(CLOCK),
.RST_N(RST_N),
.hsync(hsync),
.vsync(vsync),
.red(red),
.green(green),
.blue(blue),
.rawkbd_kbclk_x1(kbclk),
.rawkbd_kbdata_x1(kbdata),
.rawsw1_iput_x1(sw1),
.rawsw2_iput_x1(sw2),
.aL(al),
.aR(ar));
```

```
  wire expired;
  wire resetdone;
  wire rkbdata;

  reset rstgen(.clk(CLOCK), .kbdata(kbdata), .rstn(RST_N), .expired(expired), .resetdone(resetdone), .rkbdata(rkbdata))

  assign display[0] = RST_N;
  assign display[1] = expired;
  assign display[2] = resetdone;
  assign display[3] = rkbdata;
  assign display[4] = sw1;
  assign display[5] = al;
  assign display[6] = ar;


endmodule
```

P Source file `reset.v`

```
//
// Generate reset signal.
// If the keyboard data is low for about 0.1s we generate a reset
// (this handles the reset button as well).
// Also generate a reset on start up (assuming FPGA regs start at 0).
//
module reset(clk, kbdata, rstn, expired, resetdone, rkbdata);

input  clk;              // system clock
input  kbdata;           // keyboard data line
output rstn;             // global out (active low)
output expired;
output resetdone;
output rkbdata;

reg rkbdata;             // registered kbdata

reg rstn;

parameter KBCNT = 22;
parameter POCNT = 25;     // Way to large for simulations

// reg [KBCNT:0] count;        // counts low data on kbdata

reg [POCNT:0] porst;        // "power on" counter

initial begin    // Needed if simulating
//  count <= 0;
  porst <= 0;
end

wire expired;
wire resetdone;

assign expired = 0; // count[KBCNT];
assign resetdone = porst[POCNT];

always @ (posedge clk)
```

```
begin
  rkbdata <= kbdata;
end

//always @ (posedge clk)
//begin
//  if (rkbdata)
//    count <= 0;
//  else if (!expired)
//    count <= count + 1;
//end

always @ (posedge clk)
begin
  if (!rkbdata && expired)
    porst <= 0;
  else
    if (!resetdone)
      porst <= porst+1;
end

always @ (posedge clk)
begin
  rstn <= resetdone;
end

endmodule
```

Q Source file kbscan.v

```
// Excerpts from kbscan.v, software distributed on the Internet
// with the license below.
// This excerpt just shows the module port list.
// Full file length is 338 lines.

////////////////////////////////////////////////////////////////////////// -----
//
//   liaor@iname.com - http://members.tripod.com/~liaor  (05/01/2001)
//
//   This program is free software; you can redistribute it and/or
//  modify it under the terms of the GNU General Public License
//  as published by the Free Software Foundation; either version 2
//  of the License, or (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., http://www.gnu.org
//
//////////////////////////////////////////////////////////////////////////

module kbscan(
  // inputs
```

```
    clk, rstn,
    i_kbclk, i_kbdata, i_byte_ack,
  // outputs
    o_byte, o_byte_rdy, o_byte_err, o_kbclk_en
   );

//parameter SM_0 = 0;
parameter SM_1 = 0;
parameter SM_2 = 1;

parameter S_W = 10; // shift register width (DO NOT CHANGE)
parameter BYTE_ERROR = 8'd0;  // error-code to flag parity-error
parameter KBCNT = 11; // 11 clocks

input  clk;              // system clock
input  rstn;             // async global reset (active low)

input  i_kbclk;          // Keybaord clock line
input  i_kbdata;         // Keyboard data line
input  i_byte_ack;       // byte acknolwedge

output       o_byte_rdy; // Byte data ready (cleared on i_byte_ack)
output       o_byte_err; // Byte parity error (1=error, cleared on i_byte_ack)
output [7:0] o_byte;     // Byte-scancode (valid on o_byte_rdy)
output       o_kbclk_en; // enable-input on kbclk (1=allow in, 0=force low)

// ... details of Verilog implementation omitted ..

endmodule
```