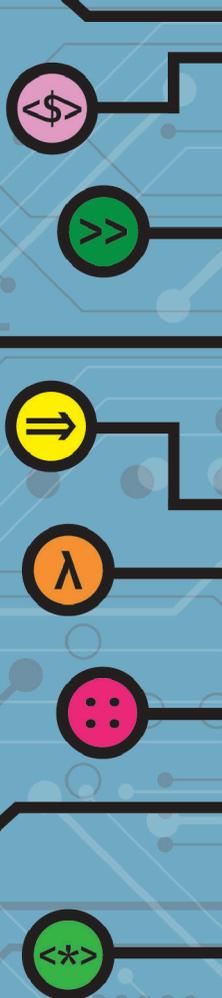
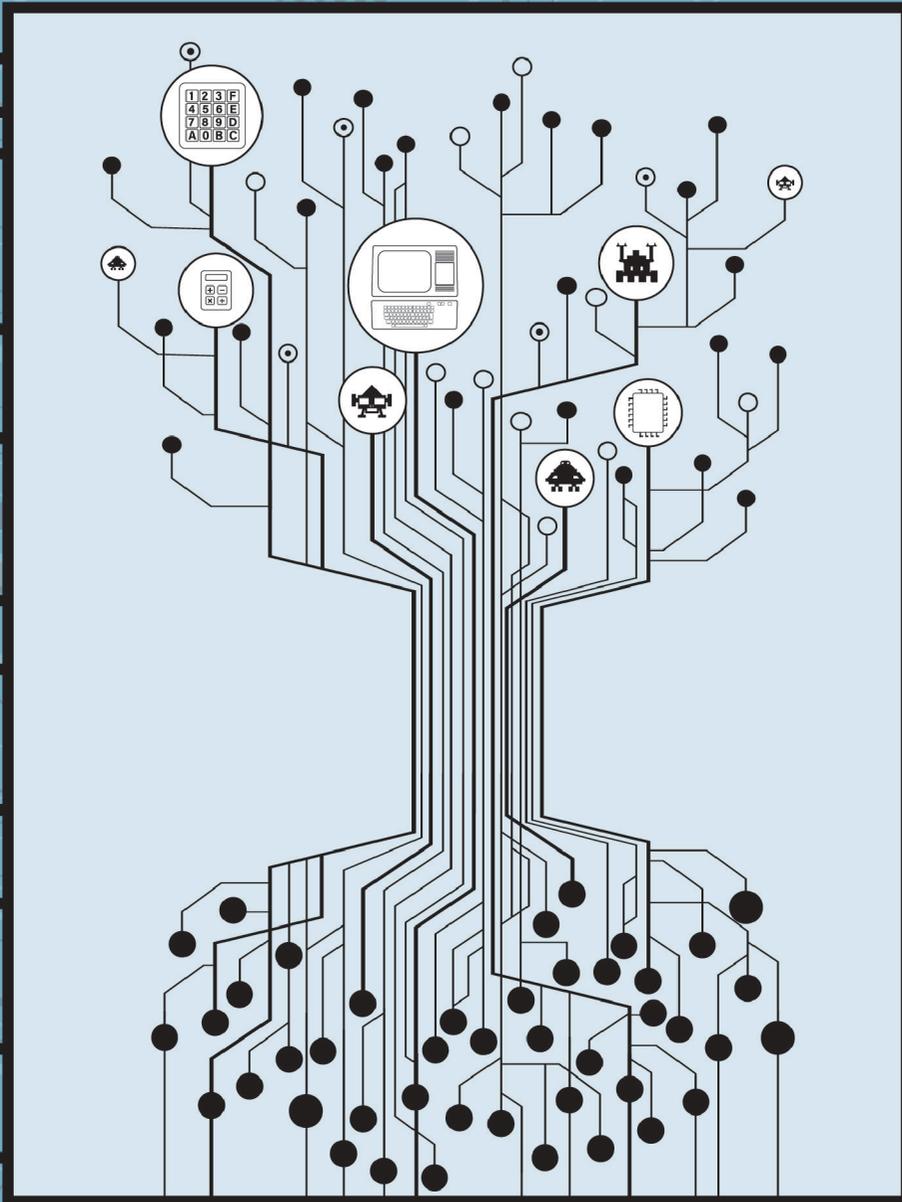


Retrocomputing with Clash

Haskell for FPGA Hardware Design



Gergő Érdi

Retrocomputing with Clash

Haskell for FPGA Hardware Design

GERGŐ ÉRDI

<https://unsafePerform.IO/retroclash/>

Retrocomputing with Clash: Haskell for FPGA Hardware Design

Copyright ©2021 Gergő Érdi

All rights reserved. No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the copyright owner, except for the use of quotations in a book review and certain other noncommercial uses permitted by copyright law.

First edition, 2021.

Cover by Jutka Sallai

Bouncy pushbutton scope screenshot from https://commons.wikimedia.org/wiki/File:Bouncy_Switch.png licensed under the terms of the Creative Commons CC0 1.0 Universal Public Domain Dedication 

<https://unsafePerform.IO/retroclash/>

Contents

Introduction	1
1 Into the world of FPGAs	5
1.1 Computers everywhere	5
1.2 Field-programmable Gate Arrays	8
1.3 Retrocomputing	15
1.4 Haskell meets Hardware	16
2 Hello, Clash!	19
2.1 Bit	19
2.2 Signal	20
2.3 Our first circuit	20
2.4 Summary	25
3 Combinational Circuits are Applicative Functors	27
3.1 Signal is an applicative functor	27
3.2 BitVectors and Vectors	29
3.3 Controlling many LEDs	30
3.4 Seven-segment display	32
3.5 Summary	38
4 State, Sequencing and Clocks: The Register Transfer-Level Model	39
4.1 Clocks and registers	40
4.2 The RTL model: register and delayed feedback	42
4.3 Finally blinkenlights!	46
4.4 Passing around Clock, Reset and Enable lines implicitly	51
4.5 Multiple clocks	54
4.6 Pushbutton-toggled LED	55
4.7 Summary	60

5	Time-domain Multiplexing	63
5.1	Does this have anything to do with mux?	63
5.2	Seven-segment displays, revisited	64
5.3	Keyboard matrix sweeping	69
5.4	Showing keypad input on a seven-segment output	74
5.5	Summary	78
6	Project: Pocket Calculator	79
6.1	A Minimal Viable Calculator	79
6.2	Binary Coded Decimal arithmetic	81
6.3	State and state transitions	85
6.4	An interactive software implementation	89
6.5	Hooking it up to hardware peripherals	91
6.6	Summary	95
7	Video Output Using VGA	97
7.1	Basic operation of a CRT display	97
7.2	Video Graphics Array	101
7.3	VGA from Clash	106
7.4	Summary	114
8	Generative Graphics	117
8.1	Combinational patterns	117
8.2	Stateful pattern generators	118
8.3	Animation	120
8.4	Coordinate transformations	124
8.5	Animation, differently	130
8.6	High-level simulation with SDL2	135
8.7	Summary	143
9	Project: Pong	145
9.1	What is Pong?	145
9.2	Top-level design	146
9.3	What is our state?	148
9.4	Drawing	153
9.5	Summary	154
10	Asynchronous Serial Communication	155
10.1	Synchronicity	155
10.2	Universal Asynchronous Serial Communication	156
10.3	Serial Transmitter	158

10.4	Serial Receiver	165
10.5	Applications	168
10.6	Summary	172
11	Programmable Machines	173
11.1	RAM machines	173
11.2	Memory	175
11.3	CPU	181
11.4	Summary	186
12	Brainfuck	187
12.1	Why Brainfuck	187
12.2	Brainfuck as a programming language	188
12.3	Brainfuck as byte code	191
12.4	Brainfuck with external memory	194
12.5	A complete Brainfuck computer	198
12.6	Brainfuck as machine code	200
12.7	High-level simulation of the CPU	218
12.8	The logic board	220
12.9	Low-level simulation of the logic board	226
12.10	Top-level circuit and peripherals	230
12.11	Summary	236
13	CHIP-8	237
13.1	History	237
13.2	The CHIP-8 computer	238
13.3	Instruction set	243
13.4	Video	250
13.5	CPU	267
13.6	Simulation, take 1	286
13.7	The complete machine	289
13.8	Simulation, take 2	292
13.9	Memory contention	293
13.10	Summary	302
14	Address decoding and memory maps	303
14.1	Room for improvement	303
14.2	A whirlwind intro to Template Haskell	305
14.3	A memory map DSL	311
14.4	Backpane connections	320
14.5	Access contention	327

14.6	Summary	329
15	Intel 8080	331
15.1	History	331
15.2	Veracity	331
15.3	Interface	333
15.4	Instruction set architecture	338
15.5	Instruction decoding	344
15.6	Microcoded implementation	349
15.7	Micro-architecture & micro-instructions	357
15.8	A direct software implementation	376
15.9	The complete CPU	384
15.10	Summary	398
16	Project: TinyBASIC	401
16.1	What is Tiny BASIC?	401
16.2	Asynchronous Communications Interface Adapter	402
16.3	The core logic board	404
16.4	Version 1: serial I/O	410
16.5	PS/2 keyboard interface	410
16.6	Textual video	417
16.7	Screen editing	424
16.8	Version 2: Keyboard and video	427
16.9	Summary	428
17	Space Invaders	431
17.1	The design of Space Invaders	432
17.2	How it fits together	433
17.3	Peripherals	434
17.4	Video	436
17.5	Logic board	443
17.6	Simulation	446
17.7	Summary	450
18	CompuColor II	451
18.1	Design	451
18.2	A Minimal Viable CompuColor II	454
18.3	Detailed rendering with SDL	461
18.4	Video hardware	467
18.5	TMS 5501	484
18.6	Keyboard	499

18.7	Floppy drive	503
18.8	Cycle-count accuracy	517
18.9	Slowing down the CPU	521
18.10	Our complete computer	524
18.11	Summary	526
Parting words		527
Index		529

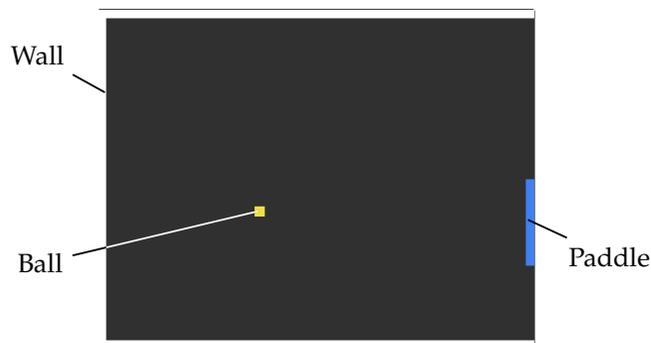
Project: Pong

In this chapter, we answer the call of the bouncing ball example circuit from the previous chapter, and build our own version of Pong, one of the earliest video games.

The original Pong didn't run on a computer: instead, its game logic and its video output was all implemented directly as a circuit of discrete components. Our version will also be computer-less; however, rather than building a rat's nest of connected registers, we will apply the same principled design as we did in the Calculator project.

9.1 What is Pong?

Pong is a very minimalistic video game simulation of tennis. Players control paddles on the sides of the screen, by moving them vertically. A ball is bouncing between the paddles and the top and bottom edges of the screen. The aim of the player is to not let the ball go out of bounds on their side. In the original two-player version, there are two paddles, one for each player. Here, we will build a solitaire version first, and leave the two-player version as an exercise. Basically, our game is going to be the squash equivalent to Pong's tennis.



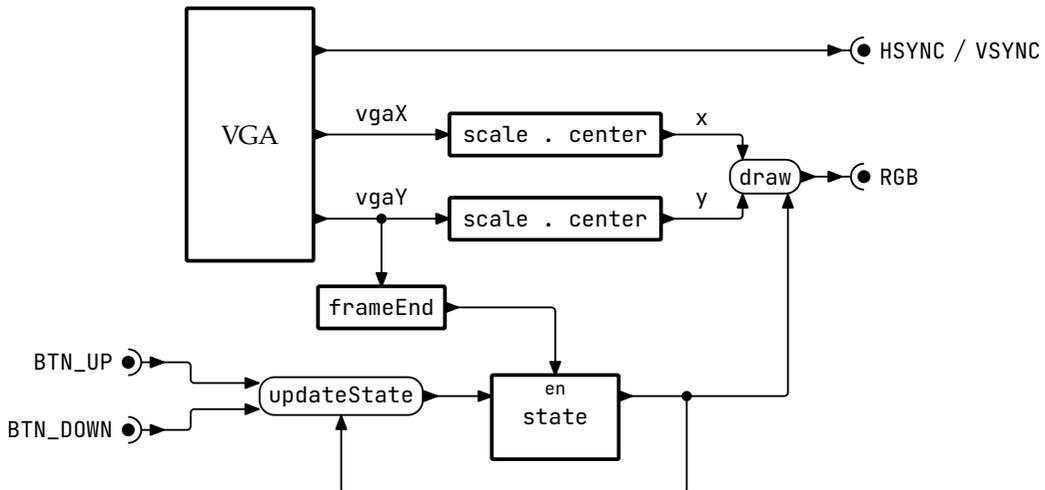
From the outside point of view, Pong is a circuit which outputs a video signal

and connects to inputs for paddle control. We will use two pushbuttons for moving the paddle up or down, and generate VGA in $640 \times 480@60$ mode for the video output. Just for the fun of it, and to give it that nice chunky retro look, the game itself will only use 256×200 resolution, which we will scale up by two and centered it on the screen.

9.2 Top-level design

Internally, we will follow the same design as the bouncing ball toy: a register holding the state, a state transition function consuming input, and a drawing function that takes the current state, and turns it into output.

Putting it all together, our design will be as follows:



This is the same design as the bouncing ball one, with an added coordinate transformation to end up with a drawing area of 256×200 , and input from the outside world in the form of the two pushbuttons controlling the paddles.

With all this groundwork laid, it is time to work out the missing details:

- data Inputs, a record that holds all user inputs
- data St, the game state
- updateState :: Inputs -> St -> St, the state transition function
- draw :: St -> Index 256 -> Index 200 -> Color, the drawing function

Once these are filled in, our `topEntity` will mostly match that of the bouncing ball circuit:

```

data Inputs = MkInputs
  { paddleUp :: Bool
  , paddleDown :: Bool
  }

type ScreenWidth = 256
type ScreenHeight = 200

topEntity
  :: "CLK_25MHZ" ::: Clock Dom25
  -> "RESET"      ::: Reset Dom25
  -> "BTN_UP"     ::: Signal Dom25 (Active High)
  -> "BTN_DOWN"   ::: Signal Dom25 (Active High)
  -> "VGA"        ::: VGAOut Dom25 8 8 8
topEntity = withEnableGen board
  where
    board (fmap fromActive -> up) (fmap fromActive -> down) = vgaOut
      vgaSync rgb
        where
          VGADriver{..} = vgaDriver vga640x480at60
          frameEnd = isFalling False (isJust <$> vgaY)

          params = defaultParams
          inputs = MkInputs <$> up <*> down

          st = regEn initState frameEnd $
              updateState params <$> inputs <*> st

          rgb = fmap (maybe (0, 0, 0) bitCoerce) $
                liftA2 <$> (draw params <$> st) <*> x <*> y
            where
              (x, _) = scale (SNat @2) . center $ vgaX
              (y, _) = scale (SNat @2) . center $ vgaY

```

Accordingly, the SDL-based simulator's main function also remains mostly the same. We increase the scaling factor to 4, since the virtual screen we are simulating now is only 256×200 pixels – remember, the transformation to 640×480 takes place only in `topEntity`, as a measure to convert to a standard video format that real-world screens can understand. This conversion is morally no different from converting the Active High pushbutton values to a semantic `Bool`, so we put it outside `draw`.

```

main :: IO ()
main =
  flip evalStateT initState $
  withMainWindow videoParams $ \events keyDown -> do
    guard $ not $ keyDown ScancodeEscape

    let params = defaultParams
        modify $ updateState params $ MkInputs
            { paddleUp = keyDown ScancodeUp
            , paddleDown = keyDown ScancodeDown
            }
        gets $ rasterizePattern . draw params
    where
      videoParams = MkVideoParams
        { windowTitle = "Pong"
        , screenScale = 4
        , screenRefreshRate = 60
        }

```

To recap, the signatures of the remaining parts to implement are:

```

data St

updateState :: Params -> Inputs -> St -> St
draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color

```

9.3 What is our state?

Compared to just a ball bouncing around in the emptiness of a video screen, a game of Pong is similar in some ways and different in others:

- There is a ball bouncing around in both cases. Our State will need to hold the ball's position and speed just like before.
- Pong has another moving part: the paddle. Since it can only be moved vertically, only its Y position needs to be stored.
- We want to draw the ball as before, but of course we also want to draw the paddle (at the right position). To emphasize to the player that it is their responsibility to hit the ball back from the right-hand edge, we will also draw walls around the other three edges of the screen.
- Pong is interactive: the player can move the paddle up or down. We take care of this by adding an extra Inputs parameter to updateState.

- There is also some complicated interaction between the ball and the paddle. At the minimum, the ball bounces off the paddle when it hits it; but that alone makes for a very boring variant of Pong. We will spice it up a notch by allowing the paddle to nudge the ball vertically, if the paddle itself is moving vertically at the moment of contact.
- Pong is a game with a goal: to avoid the ball leaving the playfield by flying off to the right. We should give some kind of indication of failure when that happens: we will flash the background color in red for one frame.

Based on this analysis, it is clear that `St` should extend the horizontal and vertical speed-and-position of the ball with a vertical paddle position, and a flag denoting if we should draw the background in the given frame in red. `updateFlag` will always clear that flag (unless, of course, the ball is just now leaving the game area); this ensures that it will flash for one frame only.

```
type Coord = Signed 10

data St = MkSt
  { _ballH, _ballV :: (Coord, Coord)
  , _paddleY :: Coord
  , _gameOver :: Bool
  }
  deriving (Show, Generic, NFDataX)
makeLenses ''St

initState :: St
initState = MkSt
  { _ballH = (10, 2)
  , _ballV = (100, 3)
  , _paddleY = 100
  , _gameOver = False
  }
```

The `Params` datatype likewise extends the ball size with new fields for the width of the walls and the size of the paddle. We also need to know how much to move the paddle on each frame if the user is holding one of the input buttons, and how much nudge should be applied to the ball when hitting it with a moving paddle.

```
data Params = MkParams
  { wallSize, ballSize :: Coord
  , paddleHeight, paddleWidth :: Coord
  , paddleSpeed, nudgeSpeed :: Coord
  }
```

```

defaultParams :: Params
defaultParams = MkParams
  { wallSize = 5
  , ballSize = 5
  , paddleHeight = 50
  , paddleWidth = 5
  , paddleSpeed = 3
  , nudgeSpeed = 3
  }

```

9.3.1 updateState

If we know what is in our state, we also know how to update it: we just update every component of it, using appropriate helper functions. The devil will be in the details of them, but we can keep this top-level `updateState` simple:

```

updateState :: Params -> Inputs -> St -> St
updateState params inp = execState $ do
  updateBall params inp
  updatePaddle params inp
  checkBounds params

```

Updating the ball is very similar to our previous code: we update its horizontal and vertical position and speed separately. Vertically, there is no extra complication; but horizontally, we need to include the paddle as a reflector only if the ball is (vertically) where the paddle is. Moreover, to implement nudging the ball on a hit with the paddle, we need to detect that collision; so we will extend `reflect` slightly to return an additional `Bool` denoting collisions.

```

reflect
  :: (Num a, Num a', Ord a, Ord a')
  => (a, a')
  -> (a, a')
  -> (Bool, (a, a'))
reflect (p, n) (x, dx)
  | sameDirection n dist = (True, (p + dist, negate dx))
  | otherwise = (False, (x, dx))
  where
    sameDirection u v = compare 0 u == compare 0 v
    dist = p - x

```

We will use `reflect` and `move` in our `State St` monad, so let's use a shorthand for their lifted versions:

```
moveM :: (Num a) => State (a, a) ()
moveM = modify move

reflectM
  :: (Num a, Num a', Ord a, Ord a')
  => (a, a') -> State (a, a') Bool
reflectM = state . reflect
```

This gives us everything to implement `updateBall`:

- `updateVert` simply moves the ball and checks for reflections from the top and bottom walls:

```
updateVert :: Params -> State St ()
updateVert MkParams{..} = void $ do
  zoom ballV $ do
    moveM
    reflectM (wallSize, 1)
    reflectM (screenHeight - wallSize - ballSize, -1)
```

- `updateHoriz` looks at the vertical position to see we are at the height of the paddle, and decides based on that whether to include a second reflector on the right-hand side.

```
updateHoriz :: Params -> State St Bool
updateHoriz MkParams{..} = do
  atPaddle <- do
    y0 <- use paddleY
    (y, _) <- use ballV
    return $ y `between` (y0 - ballSize, y0 + paddleHeight)
  zoom ballH $ do
    moveM
    reflectM (wallSize, 1)
    if not atPaddle then return False
    else reflectM (screenWidth - paddleWidth - ballSize, -1)
```

- `updateBall` itself runs `updateVert` and `updateHoriz`, and changes the vertical ball speed if `updateHoriz` returns `True`, i.e. if there was a collision with the paddle:

```

updateBall :: Params -> Inputs -> State St ()
updateBall params@MkParams{..} MkInputs{..} = do
  updateVert params
  hitPaddle <- updateHoriz params
  when hitPaddle $ ballV._2 += nudge
  where
    nudge | paddleDown = nudgeSpeed
          | paddleUp   = negate nudgeSpeed
          | otherwise  = 0

```

Compared to the complicated logic of checking for bounces and paddle hits, updating the paddle's state is much simpler: we increase or decrease `paddleY` by `paddleSpeed` depending on which input buttons are held in the given frame, and then ensure it stays in the playfield:

```

updatePaddle :: Params -> Inputs -> State St ()
updatePaddle MkParams{..} MkInputs{..} = do
  when paddleUp $ paddleY -= paddleSpeed
  when paddleDown $ paddleY += paddleSpeed
  paddleY %= clamp (wallSize, screenHeight - (wallSize + paddleHeight))

clamp :: (Ord a) => (a, a) -> a -> a
clamp (lo, hi) = max lo . min hi

```

To detect the ball going out of bounds, we simply check if its X coordinate is larger than the screen width. This way, even after the ball passes the point of no return at the edge of the paddle, we will keep drawing it until it fully leaves the screen – increasing the player's frustration just a bit more.

It is here that we set the `gameOver` field of the state. As mentioned earlier, this field is set to `True` only for the duration of the single frame when the ball actually flies out of bounds – in the next frame, the ball is already reset into the middle of the playfield (keeping its current speed and Y coordinate).

```

checkBounds :: Params -> State St ()
checkBounds MkParams{..} = do
  outOfBounds <- zoom ballH $ gets $ \(x, _) -> x > screenWidth
  gameOver    .= outOfBounds
  when outOfBounds resetBall
  where
    resetBall = ballH._1 .= half screenWidth

```

9.4 Drawing

The main structure of draw is very similar to the bouncing ball example: we just have more shapes to check against the current raster beam position.

```
draw :: Params -> St -> Index ScreenWidth -> Index ScreenHeight -> Color
draw MkParams{..} MkSt{..} ix iy
  | isWall    = white
  | isPaddle  = blue
  | isBall    = yellow
  | otherwise = if _gameOver then red else gray
where
  x = fromIntegral ix
  y = fromIntegral iy

  rect (x0, y0) (w, h) =
    x `between` (x0, x0 + w) &&
    y `between` (y0, y0 + h)

  -- Continued below
```

The definitions of the individual shapes are all straightforward: the paddle and the ball are rectangles, and each wall is an even simpler comparison.

```
isWall = or
  [ x < wallSize
  , y < wallSize
  , y >= screenHeight - wallSize
  ]

paddleStart = screenWidth - paddleWidth
isPaddle = rect (paddleStart, _paddleY) (paddleWidth, paddleHeight)

(ballX, _) = _ballH
(ballY, _) = _ballV
isBall = rect (ballX, ballY) (ballSize, ballSize)
```

For completeness's sake, here are some RGB values for these colors that hopefully won't hurt the player's eyes too much:

```
white, blue, yellow, red, gray :: Color
white = (0xff, 0xff, 0xff)
blue  = (0x40, 0x80, 0xf0)
yellow = (0xf0, 0xe0, 0x40)
red    = (0x80, 0x00, 0x00)
gray   = (0x30, 0x30, 0x30)
```

By plugging these definitions of `St`, `updateState`, and `draw` into our `topEntity` and `main`, we finish our implementation of (solitaire) Pong, including an interactive SDL simulation.

Exercises:

- On each successful hit, flash the part of the physical screen that is outside the playing area of the virtual screen.
- Increase difficulty as the game goes on by decreasing the paddle size on every k^{th} successful hit, up to a reasonable minimum paddle size.
- Draw the ball as a more round shape.
- Keep score. Drawing numerals would be quite hard with what we have so far, but we could e.g. draw a progress bar counting up to 10 misses.
- Two-player mode. This should be quite self-explanatory: hook up two more buttons to move a second, left-hand side paddle up and down. The score could be displayed as a tug-of-war progress bar.
- A fun variant of two-player Pong is to give a third “boost” button to both players, and scale up the ball’s speed by two if exactly one of the players is holding their boost button.

9.5 Summary

- Starting with the implementation of the bouncing ball circuit, the only structural change is adding an **input** signal.
- The rest of the changes are in the definition of the **state datatype** and its **transition function**. These are all “normal”, pure Haskell parts that we just happen to use in the context of a Clash circuit.

Haskell has become the functional programming language of choice for many developers due to its excellent tools for abstraction and principled program design. The open source Clash hardware description language now brings these features to FPGA development.

Retrocomputing with Clash takes the experienced Haskell programmer on a journey into the world of hardware design with Clash. Our approach is based on using Haskell to its fullest potential, using abstractions like monads and lenses in building a library of reusable components.

But that wouldn't put the *fun* in *functional programming*! And so we put these components to good use in implementing various retro-computing devices:

- Pocket calculator
- Pong
- A simple, but Turing-complete computer that uses Brainfuck as its machine code
- An implementation of the CHIP-8 virtual computer specification
- Intel 8080 CPU
- Space Invaders arcade machine
- Compucolor II, a home computer from 1977 complete with keyboard, color, video, and a floppy drive

I absolutely love the very Haskell approach to circuit design in this book, as opposed to my own write-Verilog-in-Haskell style. It leverages Haskell's type system in a very natural way to protect against many traps we as circuit designers often fall into.

The book clearly demonstrates the benefits of using a modern programming language for circuit design, where it builds reusable functionality and components at a far finer granularity than what I'm used to in traditional hardware description languages.

— Christiaan Baaij,
Clash lead developer, QBayLogic co-founder

This is the book for functional programmers looking to get into FPGAs and digital logic design. Learn Clash, the "I cant believe it's not Haskell!" hardware description language, while indulging in nostalgia for the 1980s. Take a joyride through a variety of hands-on projects, including Pong, Space Invaders, and the Compucolor II, a personal computer based on the Intel 8080. Recommended.

— Miętek Bak,
mathematician