# Clash: Haskell for FPGA Design
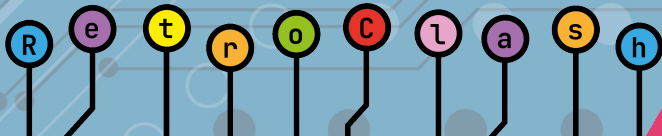
## It's easy as 1 - 2 - 3 ... 419,200

**Gergő Érdi**

https://unsafePerform.IO/

Lambda Days 2025
12th June 2025.

lambda
DΛVS
12-13 JUNE 2025
KRAKÓW | POLAND

# 1. Introduction: FPGAs and Clash

# Regularity of integrated circuits

- Zoom in far enough, and all ICs are just a bunch of transistors wired together (modulo analog/electric components)
- We can make any digital circuit by just changing how the transistors are connected
- Two transistors can form a NAND gate, which is *universal*
- Example: nine NAND gates make one full adder
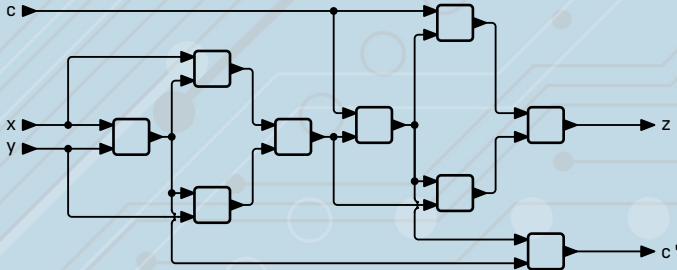
# Stateful circuits

Connecting NAND gates in a clever way, we can create a *flip-flop* that stores its d input when clk goes from low to high, and keeps that on its q output:
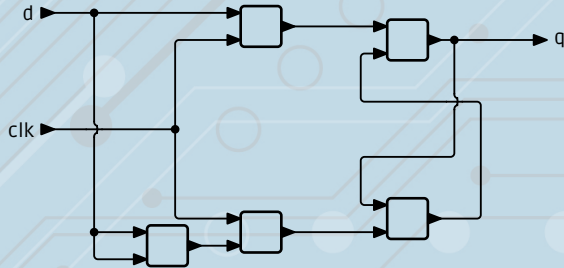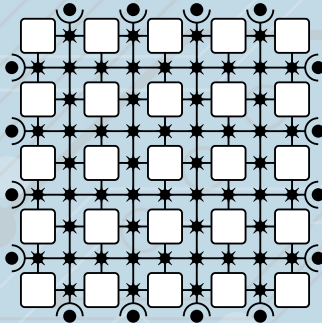
# Abstractions

- **Bus**: multiple wires in parallel

- **Multi-bit registers**: a fixed-size array of flip-flops

- **Lookup table-based Boolean functions**: instead of worrying about building circuits from individual transistors, we can build a circuit that can be **configured** for any $\mathbb{B}^n \rightarrow \mathbb{B}^m$ function (for fixed $n$ and $m$)

- **Synchronous ciruits**: shared clock for all registers, intra-clock-cycle behaviour ignored (as long as everything settles in time)

# Field-Programmable Gate Arrays

- A large amount of logic blocks, with configurable intra-block Boolean functions and registers, and configurable inter-block connections
- I/O ports also accessible through these interconnects



- "A chip fab on your desktop": designs can be uploaded electronically

# RTL: the register-transfer level model

- A convenient way to abstract the details of individual FPGAs: arbitrary-sized "super-functions" and "super-registers"
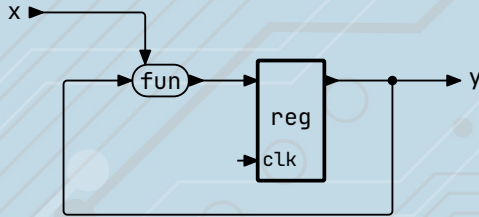


- These get mapped to as many logic blocks as needed, depending on the concrete target FPGA

# Clash: Haskell to Hardware (FPGAs, ASICs)

- `Signal :: Domain -> Type -> Type`
- `instance Applicative (Signal dom)`
- `register :: a -> Signal dom a -> Signal dom a`

# Clash: Haskell to Hardware (FPGAs, ASICs)

- `Signal :: Domain -> Type -> Type`
- `instance Applicative (Signal dom)`
- `register :: a -> Signal dom a -> Signal dom a`



```
y = register y0 (fun <$> x <*> y)
```
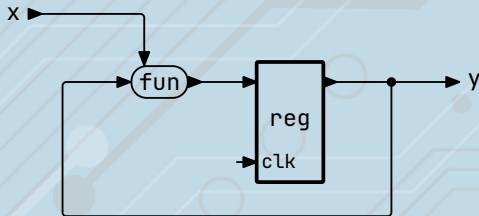
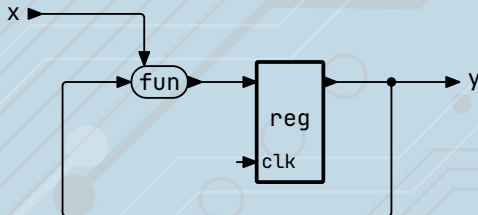# Clash: Haskell to Hardware (FPGAs, ASICs)

- `Signal :: Domain -> Type -> Type`
- `instance Applicative (Signal dom)`
- `register :: a -> Signal dom a -> Signal dom a`



```
y = register y0 (fun <$> x <*> y)
```

y is recursively defined, but the recursion is guarded by `register`.

# Example: Counters

```
countTo800
    :: (HiddenClockResetEnable dom)
    => Signal dom (Index 800)
countTo800 = cnt
  where
    cnt = register 0 (countSucc <$> cnt)
```

- dom: clock domain
- HiddenClockResetEnable dom: implicit routing of clock, reset, and enable lines to all registers
- Index 800: type of integers between 0 and 799 (Fin 800 if you know Agda, Idris, &c.)
- countSucc: uses the Counter typeclass

# Example: Counters (cont'd)
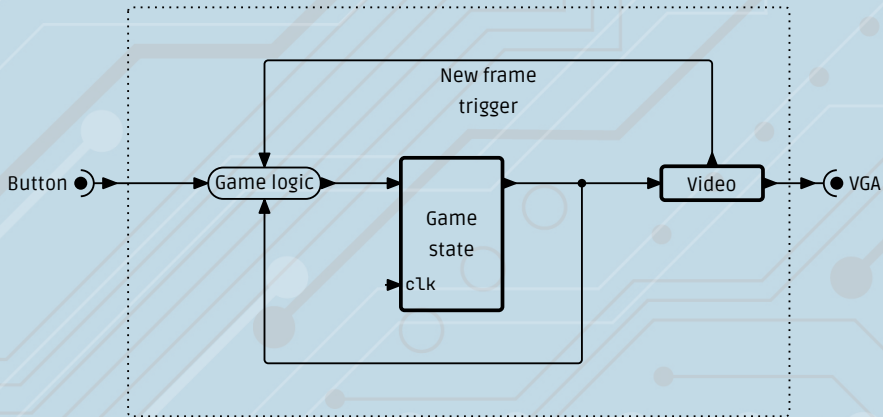
```
countTo524And800
    :: (HiddenClockResetEnable dom)
    => (Signal dom (Index 524), Signal dom (Index 800))
countTo524And800 = unbundle cnt
  where
    cnt = register (0, 0) (countSucc <$> cnt)
```
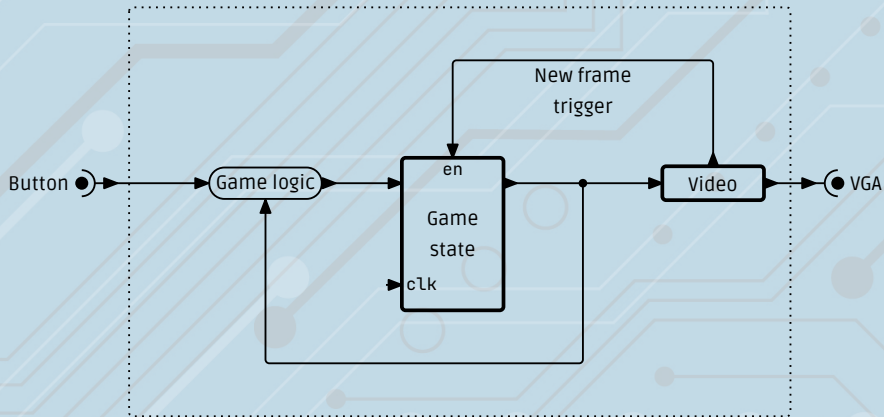
- Counter is closed over products
- unbundle :: Signal dom (a, b) -> (Signal dom a, Signal dom b)

# 2. Video game hardware

# Minimal viable game console

# Minimal viable game console

# Minimal viable game console



```
topEntity
    :: "CLK"   ::: Clock Dom25
    -> "RESET" ::: Reset Dom25
    -> "BTN"   ::: Signal Dom25 Bool
    -> "VGA"   ::: VGAOut Dom25
topEntity clk rst btn = withEnableGen board clk rst btn
  where
    board btn = vga
      where
        state = regEn initState newFrame (updateState <$> btn <*> state)
        (vga, newFrame) = video state

createDomain vSystem{vName="Dom25", vPeriod = hzToPeriod 25_175_000}
```
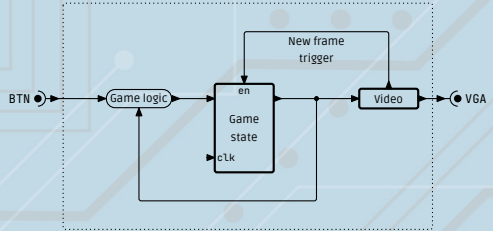
# Minimal viable game console
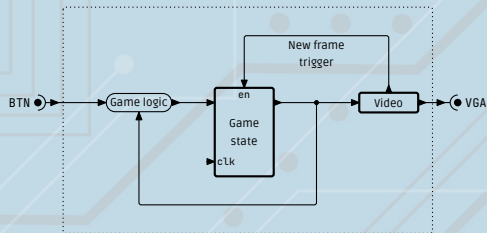


```
topEntity
    :: "BTN"    ::: Signal Dom25 Bool
    -> "VGA"    ::: VGAOut Dom25
topEntity btn = vga
  where
    state = regEn initState newFrame (updateState <$> btn <*> state)
    (vga, newFrame) = video state
```

# Video signal generation

- Cathode ray tube (CRT): electron beam scans the screen (our) left to right, top to bottom, like a typewriter writing out a paragraph, changing intensity to render the image
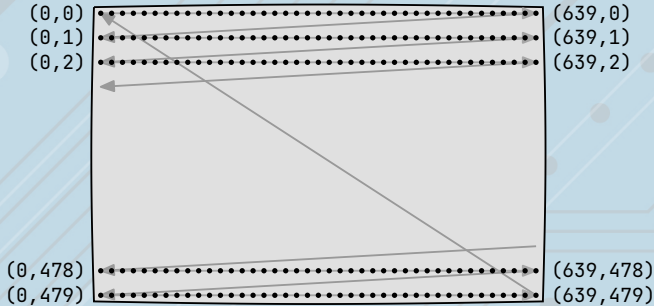


- At the end of each line / frame, a horizontal / vertical sync signal triggers the beam to go return to the next line / frame's start.

# VGA

- VGA: old analog video standard
- Sweet spot of theoretical simplicity and widespread support both by displays and FPGA development boards
- Three separate analog color channels (red/green/blue)
- Separate horizontal and vertical sync trigger lines
- Different frame rates and resolutions possible, depending on exact sync timings
- Timings quantized to a given **pixel clock**

# VGA sync signals



Visible area     Front porch     Back porch

Horizontal sync

Sync pulse

Vertical sync

Visible area

Front porch
Sync pulse
Back porch

# VGA sync signals: 640 × 480 at 60 Hz



Visible area
25.422 $\mu$s

Front porch
0.635 $\mu$s

Back porch
1.907 $\mu$s

Horizontal sync

Sync pulse
3.813 $\mu$s

Vertical sync

Visible area: 15.253 ms

Front porch: 349.551 $\mu$s

Sync pulse: 63.555 $\mu$s

Back porch: 981.104 $\mu$s

# VGA sync signals: 640 × 480 at 60 Hz



Visible area 25.422 $\mu$s — Front porch 0.635 $\mu$s — Back porch 1.907 $\mu$s

Horizontal sync

Sync pulse 3.813 $\mu$s

Vertical sync

Visible area: 480 lines

Front porch: 11 lines
Sync pulse: 2 lines
Back porch: 31 lines

# VGA sync signals: 640 × 480 at 60 Hz (25.175 MHz)
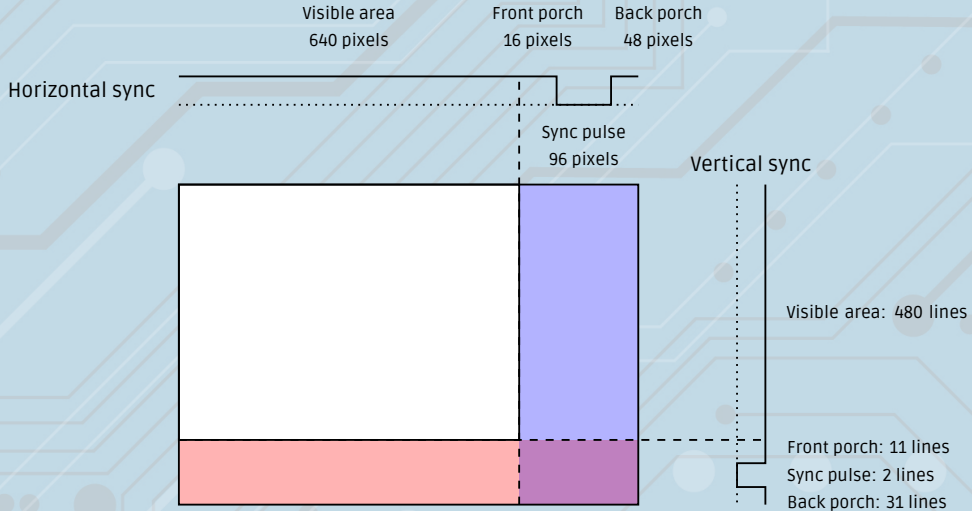


Visible area
640 pixels

Front porch
16 pixels

Back porch
48 pixels

Horizontal sync

Sync pulse
96 pixels

Vertical sync

Visible area: 480 lines

Front porch: 11 lines
Sync pulse: 2 lines
Back porch: 31 lines

# VGA sync signals: Generating with counters

```
generateSync
    :: (HiddenClockResetEnable dom)

    => (Signal dom Bit, Signal dom Bit)
```

- Complete horizontal raster line: 640+16+96+48 = 800 pixels
- Complete frame: 480+11+2+31 = 524 lines
- To generate a valid VGA signal, all we need to do is **count** to 800 $\times$ 524 = 419,200, and pull the sync lines low if the counter falls into the sync pulse territory

# VGA sync signals: Generating with counters

```
generateSync
    :: (HiddenClockResetEnable dom
       ,DomainPeriod dom ~ HzToPeriod 25_175_000)
    => (Signal dom Bit, Signal dom Bit)
```

- Complete horizontal raster line: 640+16+96+48 = 800 pixels
- Complete frame: 480+11+2+31 = 524 lines
- To generate a valid VGA signal, all we need to do is **count** to 800 $\times$ 524 = 419,200, and pull the sync lines low if the counter falls into the sync pulse territory
- This only works out if the circuit runs at the right clock speed

## VGA sync signals: Generating with counters

```
generateSync = (vgaHSync, vgaVSync)
  where
    cnt = register (0, 0) (countSucc @(Index 524, Index 800) <$> cnt)
    (vcount, hcount) = unbundle cnt

    vgaHSync = sync low . (`between` (656, 751)) <$> hcount
    vgaVSync = sync low . (`between` (491, 492)) <$> vcount
```

# VGA sync signals: Generating with counters

```
generateSync = (vgaHSync, vgaVSync)
  where
    cnt = register (0, 0) (countSucc @(Index 524, Index 800) <$> cnt)
    (vcount, hcount) = unbundle cnt

    vgaHSync = sync low . (`between` (656, 751)) <$> hcount
    vgaVSync = sync low . (`between` (491, 492)) <$> vcount

between :: (Ord a) => a -> (a, a) -> Bool
x `between` (lo, hi) = lo <= x && x <= hi

sync :: Bit -> Bool -> Bit
sync polarity b = if b then polarity else complement polarity
```

# Beyond a blank screen

- The sync signals alone describe a valid, but blank, picture
- To render something more interesting, we need to know which particular visible pixel (if any) is drawn by the electron beam right now

```
data VGASync dom = VGASync
    { vgaHSync :: Signal dom Bit
    , vgaVSync :: Signal dom Bit
    }

data VGADriver dom w h = VGADriver
    { vgaSync :: VGASync dom
    , vgaX    :: Signal dom (Maybe (Index w))
    , vgaY    :: Signal dom (Maybe (Index h))
    }
```

# Beyond a blank screen

```
vgaDriver640x480at60 :: ... => VGADriver dom 640 480
vgaDriver640x480at60 = VGADriver{ vgaSync = VGASync{..}, .. }
  where
    ...
    vgaX = strengthen <$> hcount
    vgaY = strengthen <$> vcount

strengthen
    :: forall n k. (KnownNat n, KnownNat k)
    => Index (n + k) -> Maybe (Index n)
strengthen x
    | x <= fromIntegral (maxBound @(Index n)) = Just (fromIntegral x)
    | otherwise = Nothing
```

# VGA connector format

```
data VGAOut dom = VGAOut
    { vgaSync  :: VGASync dom
    , vgaR     :: "RED"   ::: Signal dom Word8
    , vgaG     :: "GREEN" ::: Signal dom Word8
    , vgaB     :: "BLUE"  ::: Signal dom Word8
    }
```

# VGA connector format

```
type Color = (Word8, Word8, Word8)

vgaOut
    :: (HiddenClockResetEnable dom)
    => VGASync dom
    -> Signal dom Color
    -> VGAOut dom
vgaOut vgaSync@VGASync{..} rgb = VGAOut{..}
  where
    (vgaR, vgaG, vgaB) = unbundle (blank <$> vgaVisible <*> rgb)
    vgaVisible = isJust <$> vgaX .&&. isJust <$> vgaY
    blank visible color = if visible then color else (0, 0, 0)
```

# The complete video subsystem

```
video
    :: ( DomainPeriod dom ~ HzToPeriod 25_175_000
       , HiddenClockResetEnable dom )
    => Signal dom St
    -> (VGAOut dom, Signal dom Bool)
video state = (vgaOut vgaSync rgb, newFrame)
  where
    VGADriver{..} = vgaDriver640x480at60
    newFrame = isFalling False (isJust <$> vgaY)
    rgb = draw
      <$> state
      <*> (fromJust <$> vgaX)
      <*> (fromJust <$> vgaY)
```

# 3. Flappy Bird

# 3. Flappy ~~Bird~~ Square

# We're in familiar territory now!

What are the missing parts of our circuit?

- `data St`
- `initState :: St`
- `updateState :: Bool -> St -> St`
- `draw :: St -> Index 640 -> Index 480 -> Color`

Note that all of these are *pure (non-Signal) Haskell functions*, and these are all *game-specific.* We really only needed to know enough Clash to count to 419,200!

# *Flappy Bird? More like Crappy Bird amirite?!*

To fit into this talk, we make a ton of simplification:

- Fixed, looping level layout
- "Bird" drawn as a square, "pipes" drawn with just three colors
- "Game over" is a single frame with red background, then immediately restarts
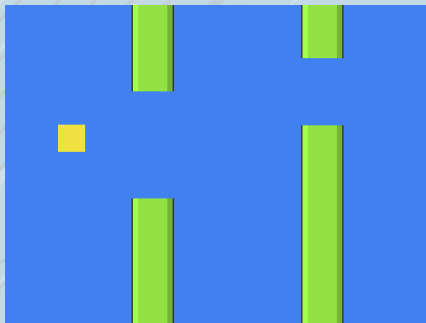
# Game state

```
data St = MkSt
    { birdY       :: Signed 10
    , birdSpeed   :: Signed 10
    , scrollOffset :: Index (NumPipes * PipeWidth * PipeGap)
    , gameOver    :: Bool
    }
    deriving (Show, Generic, NFDataX)

type PipeGap = 4
type PipeWidth = 64
type NumPipes = 4
```

# Game state

```
data St = MkSt
    { birdY       :: Signed 10
    , birdSpeed   :: Signed 10
    , scrollOffset :: Index (NumPipes * PipeWidth * PipeGap)
    , gameOver    :: Bool
    }
    deriving (Show, Generic, NFDataX)

type PipeGap = 4
type PipeWidth = 64
type NumPipes = 4

pipes :: Vec NumPipes (Index 480, Index 480)
```

# Game state: movement

```haskell
updateState :: Bool -> St -> St
updateState btn st@MkSt{..} = st
    { scrollOffset = countSucc scrollOffset
    , birdSpeed   = if btn then birdSpeed - 5 else birdSpeed + 1
    , birdY       = birdY + birdSpeed `shiftR` 3
    }
```

# Game state: game over

```haskell
updateState :: Bool -> St -> St
updateState btn st@MkSt{..}
  | gameOver  = initState
  | otherwise = st
    { ...
    , gameOver     = not birdClear
    }
  where
    (top, bottom) = pipeAt birdX st
    birdClear = birdY `between` (top + birdHeight, bottom - birdHeight)
```

# Game state: pipes

```
pipeAt :: Index 640 -> St -> (Index 480, Index 480)
pipeAt x MkSt{..} = (top, bottom)
  where
    idx    :: Index NumPipes
    gap    :: Index PipeGap
    offset :: Index PipeWidth
    (idx, gap, offset) =
        bitCoerce (satAdd SatWrap (fromIntegral x) scrollOffset)
    (top, bottom)
        | gap == maxBound = pipes !! idx
        | otherwise       = (minBound, maxBound)

               x = 598 = 1 0 0 1 0 1 0 1 1 0
                         idx  gap   offset
                         = 2  = 1    = 22
```

# Rendering the gamefield

```
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | otherwise = blue
```

# Rendering the gamefield



```haskell
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | otherwise = blue
  where
    isBird =
        x `around` (birdX, birdWidth) &&
        fromIntegral y `around` (birdY, birdHeight)

around :: (Ord a, Num a) => a -> (a, a) -> Bool
x `around` (p, r) = x `between` (p - r, p + r)
```

# Rendering the gamefield



```haskell
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | isPipe    = green
    | otherwise = blue
  where
    isPipe = not (fromIntegral y `between` (top, bottom))
    (top, bottom) = pipeAt x st
```
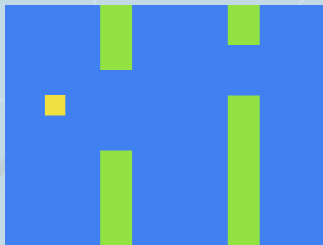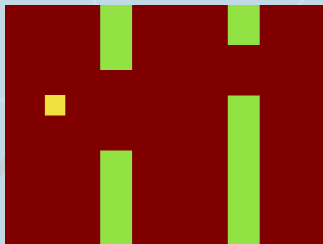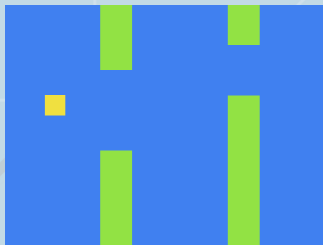
# Rendering the gamefield



```haskell
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | isPipe    = green
    | otherwise = if gameOver then red else blue
  where
    isPipe = not (fromIntegral y `between` (top, bottom))
    (top, bottom) = pipeAt x st
```

# Rendering the gamefield: snazzing it up



```haskell
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | isPipe    = green
    | otherwise = if gameOver then red else blue
  where
    (top, bottom) = pipeAt x st
```

# Rendering the gamefield: snazzing it up



```haskell
draw :: St -> Index 640 -> Index 480 -> Color
draw st@MkSt{..} x y
    | isBird    = yellow
    | isPipe    = pipeColor
    | otherwise = if gameOver then red else blue
  where
    (top, bottom, offset) = pipeAt x st

    pipeColor
        | offset < 2                = gray
        | offset < 10               = lightGreen
        | offset > (maxBound - 2)    = gray
        | offset > (maxBound - 10)   = darkGreen
        | otherwise                 = green
```
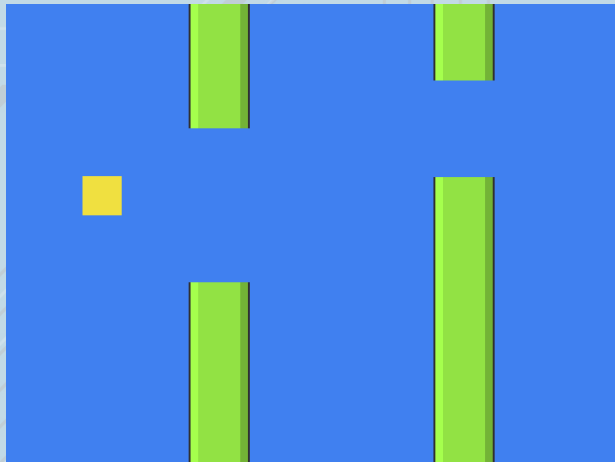
# Finished version



https://unsafePerform.IO/flappy/
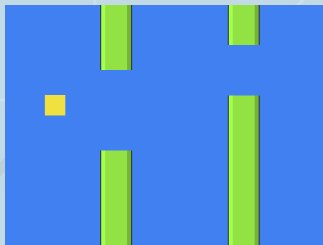
# Next steps

- Wait for button press at game startup and after death:

  ```
  data GameState = Welcome | Playing St
  ```

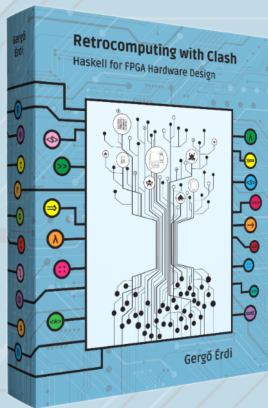- Half-second "flapping" for each single button press:

  ```
  data BirdState = Floating | Flapping (Index 30)
  ```

- Randomize the pipes: move `pipes` and add an LFSR to `St`

  ```
  lfsr :: Unsigned 9 -> Index 9
  ```

# 4. Smarmy salesmanship

# *Retrocomputing with Clash*

Using Haskell's tools of abstraction to their fullest potential in hardware design

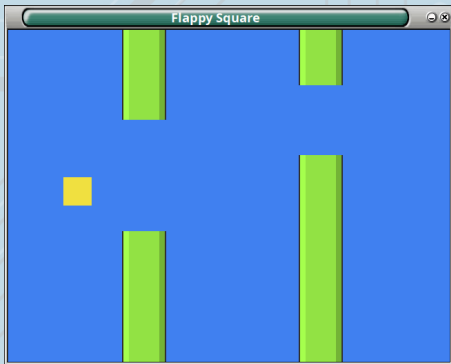Full implementation of various fun retrocomputing devices:

- Desktop calculator
- Pong
- Brainfuck as machine code
- CHIP-8
- Intel 8080 CPU
- Space Invaders arcade machine
- Compucolor II home computer

Available in print and PDF at:

https://unsafePerform.IO/retroclash/

# 5. Extra slides

# High-level simulation



- Compile the logic (`updateState` and `draw`) as normal Haskell functions, connect keyboard events to `updateState`, render the output using e.g. SDL
- End-to-end simulation: VGA signal interpreter

# Resource usage

Xilinx Vivado report:

```
Adders :
    2 Input    10 Bit     Adders := 14

Registers :
               31 Bit  Registers := 1
               20 Bit  Registers := 1
                1 Bit  Registers := 1

Muxes :
    2 Input    31 Bit     Muxes := 1
    8 Input    24 Bit     Muxes := 1
    4 Input    18 Bit     Muxes := 3
    2 Input    18 Bit     Muxes := 3
    2 Input     1 Bit     Muxes := 2
```

# Resource usage

Xilinx Vivado report:

```
Adders :
    2 Input    10 Bit    Adders := 14  -- birdSpeed, birdY, around

Registers :
               31 Bit  Registers := 1   -- state
               20 Bit  Registers := 1   -- (vcount, hcount)
                1 Bit  Registers := 1   -- newFrame

Muxes :
    2 Input    31 Bit    Muxes := 1  -- updateState
    8 Input    24 Bit    Muxes := 1  -- draw
    4 Input    18 Bit    Muxes := 3
    2 Input    18 Bit    Muxes := 3
    2 Input     1 Bit    Muxes := 2
```