# Functional Programming as a Tool of Thought

Haskell & Clash for higher level hardware design

Alex Mason - FP-Syd 2024 - 24/11/2024

## What's this about

What we'll (try to) cover

- What is Haskell - very brief
- What is Clash
- Features of Clash
- How it's might be useful for us
- Some examples

## What is Haskell?

# What is Haskell?

- Functional Programming Language

```
f x y z = x * y + z
```

# What is Haskell?

- Functional Programming Language
- Strong Types

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z

g = (7 :: Int) + "4"
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```

```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```

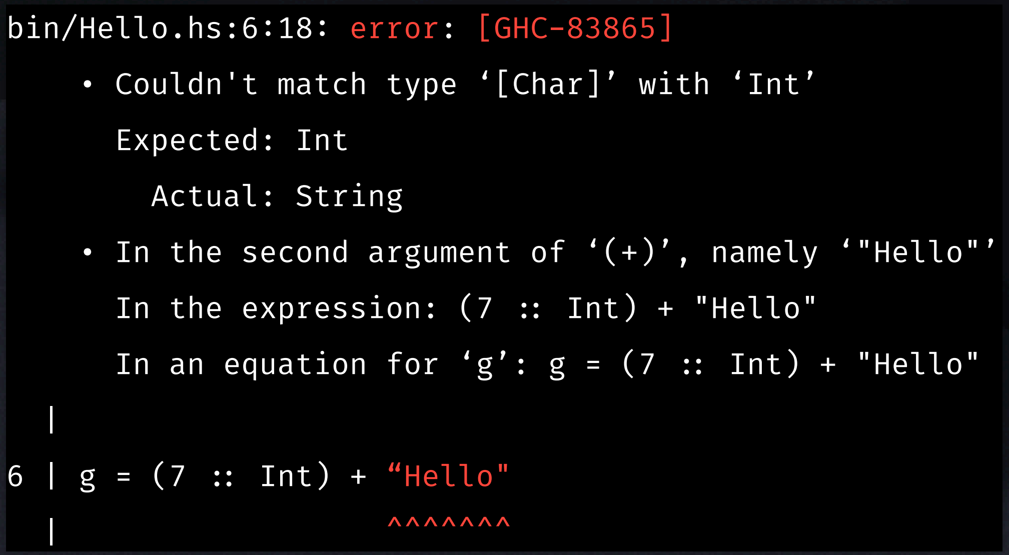
```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems

```haskell
f :: Integer -> (Integer -> (Integer -> Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```



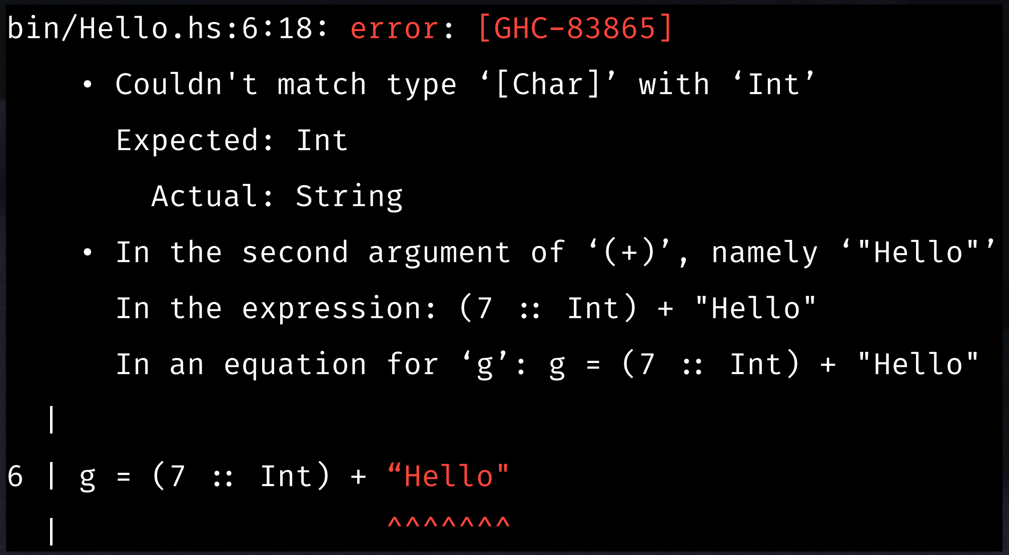
```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```



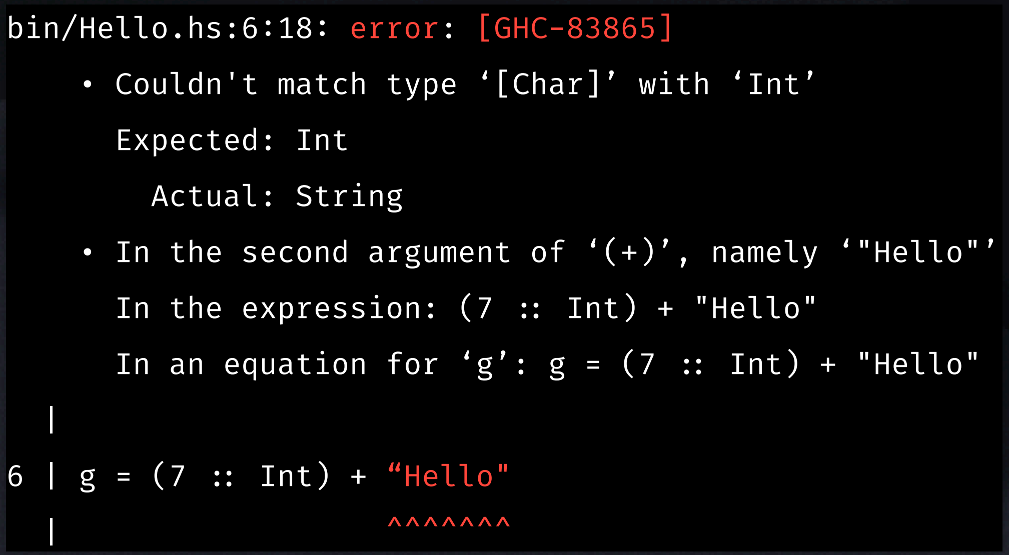
```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```



```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type '[Char]' with 'Int'
      Expected: Int
        Actual: String
    • In the second argument of '(+)', namely '"Hello"'
      In the expression: (7 :: Int) + "Hello"
      In an equation for 'g': g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + "Hello"
  |                  ^^^^^^^
```

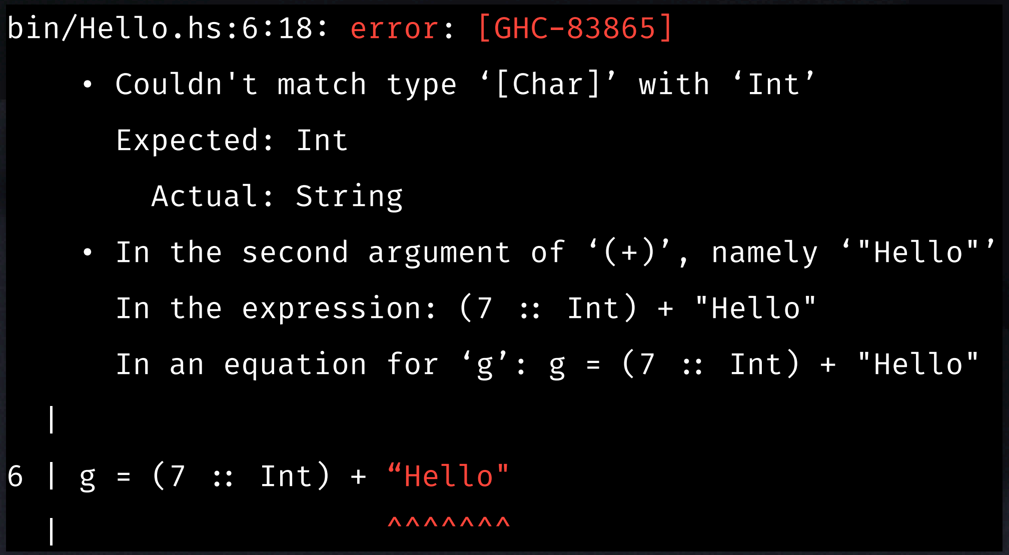
# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```

```haskell
g = (7 :: Int) + "4"
```



```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

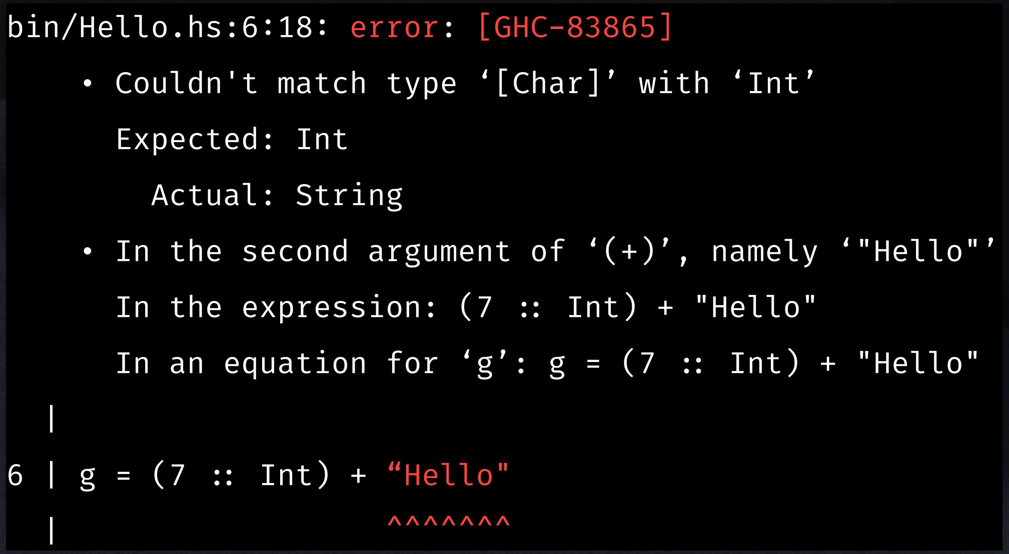
- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```

```
$ cabal repl bin/Hello.hs
```

```haskell
g = (7 :: Int) + "4"
```



```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

```haskell
f :: Integer -> (Integer -> (Integer -> Integer))
f x y z = x * y + z
```

```haskell
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```

```haskell
g = (7 :: Int) + "4"
```

```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

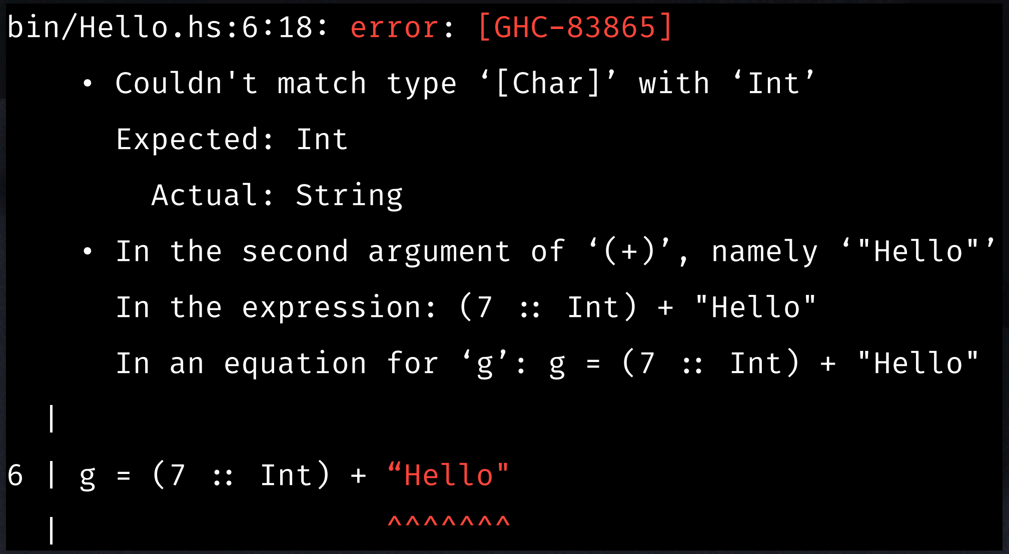
```
$ cabal repl bin/Hello.hs
Build profile: -w ghc-9.8.2 -O1
In order, the following will be built (use -v for more details):
 - LI-clash-talk-0.1 (exe:hello) (first run)
Preprocessing executable 'hello' for LI-clash-talk-0.1..
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( bin/Hello.hs, interpreted )
Ok, one module loaded.
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

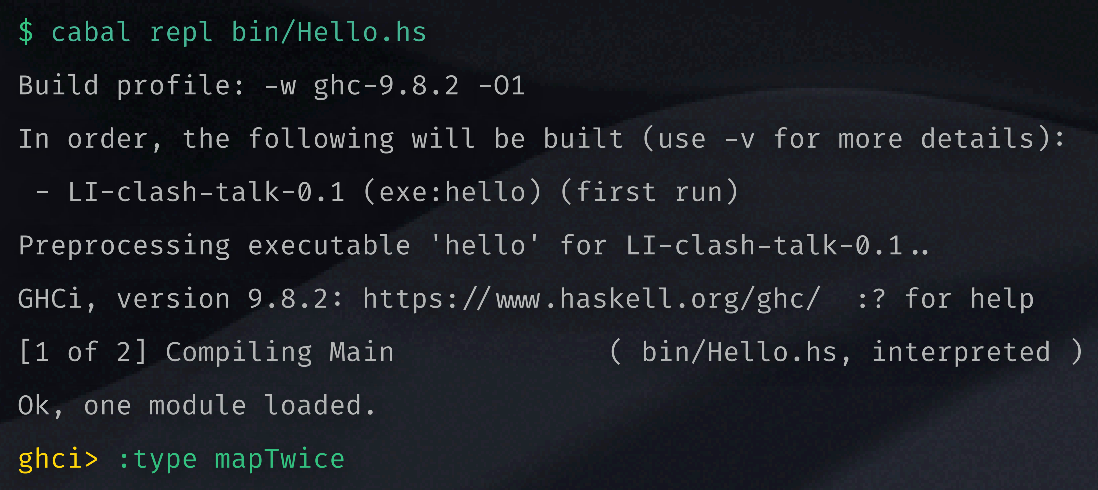
```haskell
f :: Integer → (Integer → (Integer → Integer))
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```



```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

```haskell
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```



```
$ cabal repl bin/Hello.hs
Build profile: -w ghc-9.8.2 -O1
In order, the following will be built (use -v for more details):
 - LI-clash-talk-0.1 (exe:hello) (first run)
Preprocessing executable 'hello' for LI-clash-talk-0.1..
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( bin/Hello.hs, interpreted )
Ok, one module loaded.
ghci> :type mapTwice
```

# What is Haskell?

- Functional Programming Language
- Strong Types
- Static Types
- Compiled to fast, native code
- One of the world's most powerful type systems
- Type Inference
- Loved by first year ANU students ❤️🙃

```haskell
f :: Integer → (Integer → (Integer → Integer))
```

```haskell
f x y z = x * y + z
```

```haskell
g = (7 :: Int) + "4"
```

```
bin/Hello.hs:6:18: error: [GHC-83865]
    • Couldn't match type ‘[Char]’ with ‘Int’
      Expected: Int
        Actual: String
    • In the second argument of ‘(+)’, namely ‘"Hello"’
      In the expression: (7 :: Int) + "Hello"
      In an equation for ‘g’: g = (7 :: Int) + "Hello"
  |
6 | g = (7 :: Int) + “Hello"
  |                  ^^^^^^^
```

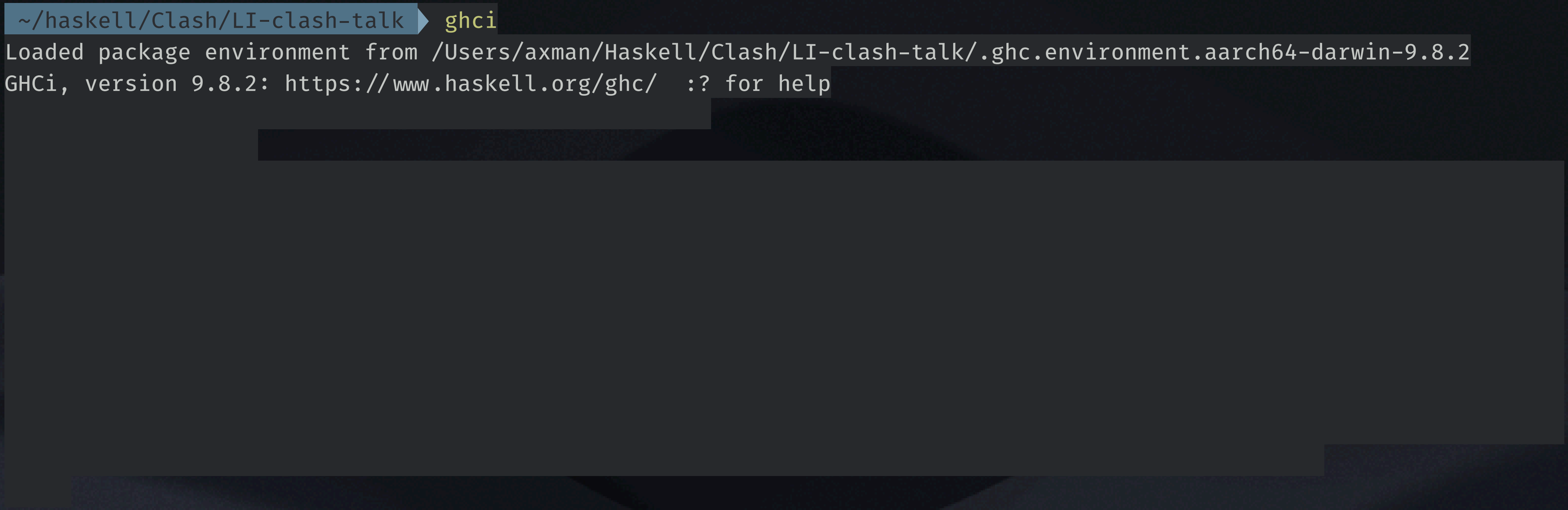
```haskell
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```

```
$ cabal repl bin/Hello.hs
Build profile: -w ghc-9.8.2 -O1
In order, the following will be built (use -v for more details):
 - LI-clash-talk-0.1 (exe:hello) (first run)
Preprocessing executable 'hello' for LI-clash-talk-0.1..
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( bin/Hello.hs, interpreted )
Ok, one module loaded.
ghci> :type mapTwice
mapTwice :: (a → a) → [a] → [a]
ghci>
```

# Tools

## Tools

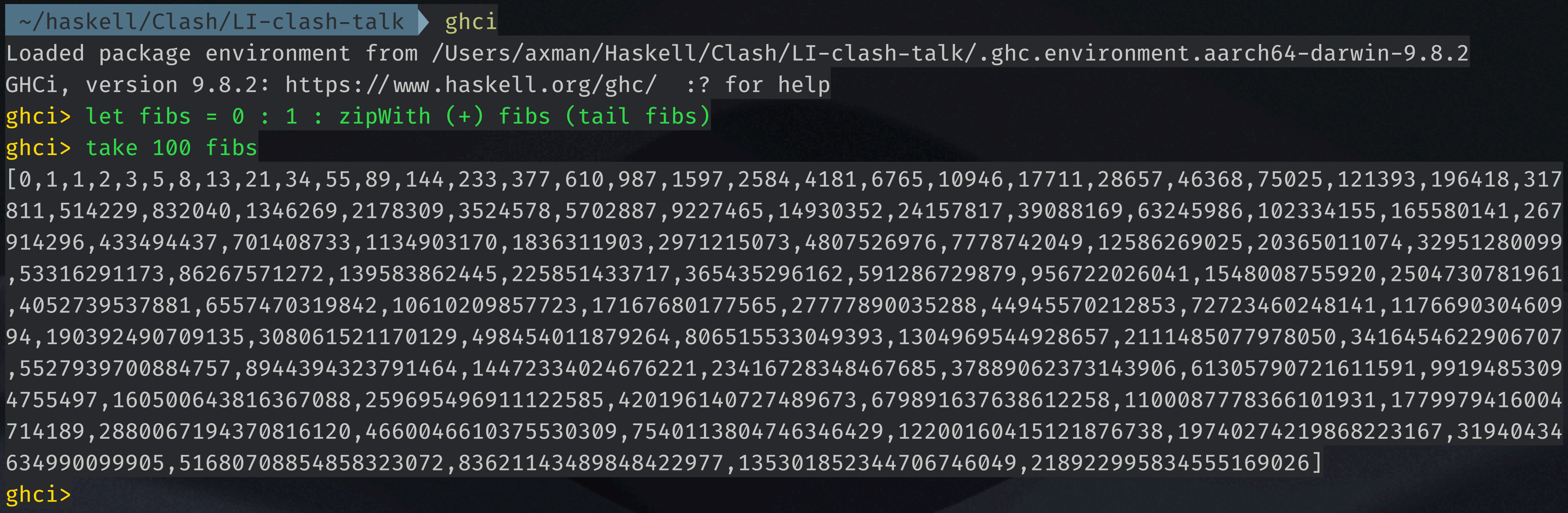
- Interactive REPL (like python)



```
~/haskell/Clash/LI-clash-talk  ghci
Loaded package environment from /Users/axman/Haskell/Clash/LI-clash-talk/.ghc.environment.aarch64-darwin-9.8.2
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
```

# Tools

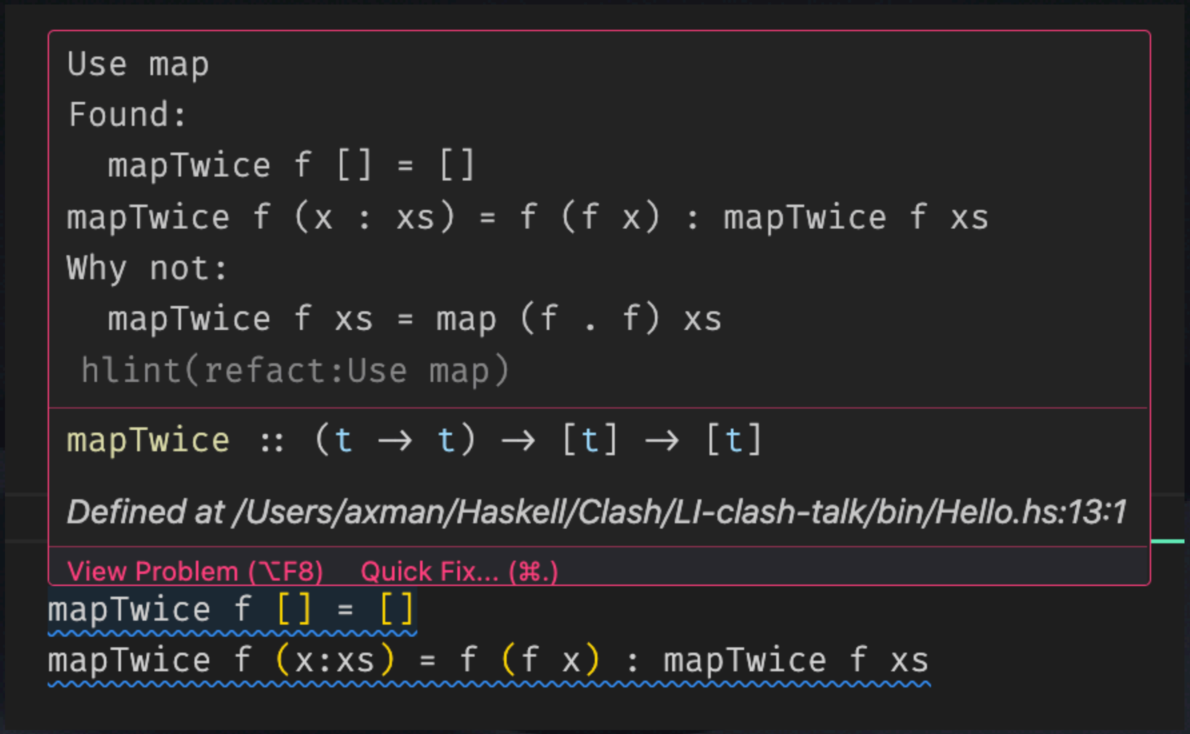
- Interactive REPL (like python)



```
~/haskell/Clash/LI-clash-talk  ghci
Loaded package environment from /Users/axman/Haskell/Clash/LI-clash-talk/.ghc.environment.aarch64-darwin-9.8.2
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
ghci> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
ghci> take 100 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317
811,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986,102334155,165580141,267
914296,433494437,701408733,1134903170,1836311903,2971215073,4807526976,7778742049,12586269025,20365011074,32951280099
,53316291173,86267571272,139583862445,225851433717,365435296162,591286729879,956722026041,1548008755920,2504730781961
,4052739537881,6557470319842,10610209857723,17167680177565,27777890035288,44945570212853,72723460248141,1176690304609
94,190392490709135,308061521170129,498454011879264,806515533049393,1304969544928657,2111485077978050,3416454622906707
,5527939700884757,8944394323791464,14472334024676221,23416728348467685,37889062373143906,61305790721611591,9919485309
4755497,160500643816367088,259695496911122585,420196140727489673,679891637638612258,1100087778366101931,1779979416004
714189,2880067194370816120,4660046610375530309,7540113804746346429,12200160415121876738,19740274219868223167,31940434
634990099905,51680708854858323072,83621143489848422977,135301852344706746049,218922995834555169026]
ghci>
```

## Tools

- Interactive REPL (like python)
- Language Server



```
Use map
Found:
  mapTwice f [] = []
mapTwice f (x : xs) = f (f x) : mapTwice f xs
Why not:
  mapTwice f xs = map (f . f) xs
 hlint(refact:Use map)
```

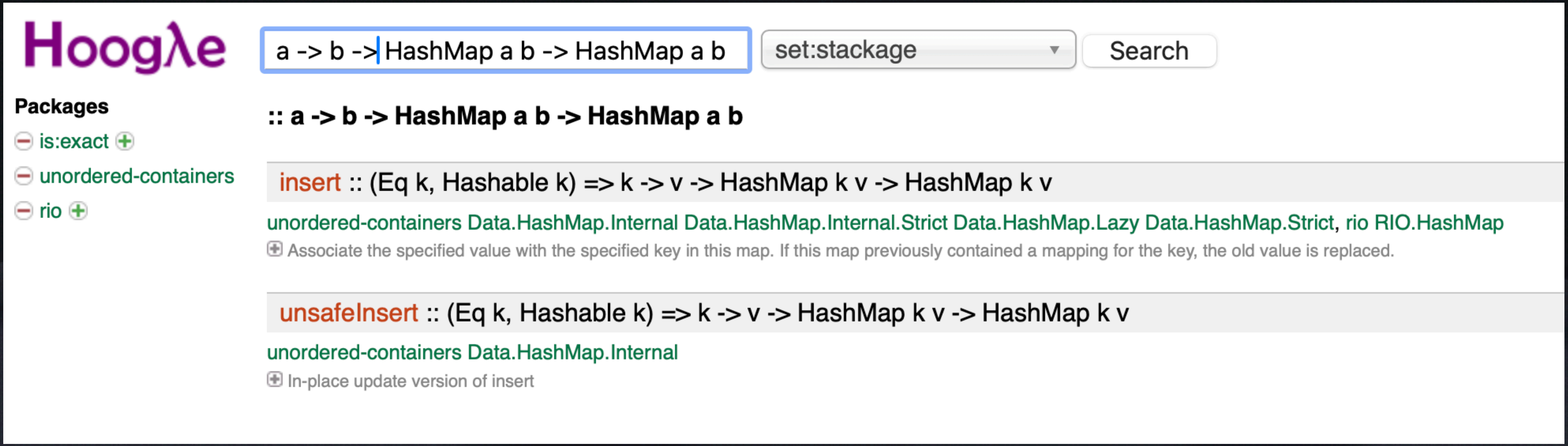
```
mapTwice :: (t → t) → [t] → [t]
```

*Defined at /Users/axman/Haskell/Clash/LI-clash-talk/bin/Hello.hs:13:1*

```
mapTwice f [] = []
mapTwice f (x:xs) = f (f x) : mapTwice f xs
```

## Tools

- Interactive REPL (like python)
- Language Server
- Hoogle - search for code by name, or type



a -> b -> HashMap a b -> HashMap a b

is:exact
unordered-containers
rio

**:: a -> b -> HashMap a b -> HashMap a b**

insert :: (Eq k, Hashable k) => k -> v -> HashMap k v -> HashMap k v

unordered-containers Data.HashMap.Internal Data.HashMap.Internal.Strict Data.HashMap.Lazy Data.HashMap.Strict, rio RIO.HashMap

Associate the specified value with the specified key in this map. If this map previously contained a mapping for the key, the old value is replaced.

unsafeInsert :: (Eq k, Hashable k) => k -> v -> HashMap k v -> HashMap k v

unordered-containers Data.HashMap.Internal

In-place update version of insert

## What is Clash

## What is Clash

- Haskell for describing circuits

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language
- Has specific support for Xilinx and Intel FPGAs.

## What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language
- Has specific support for Xilinx and Intel FPGAs.
- Used in real application

# What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language
- Has specific support for Xilinx and Intel FPGAs.
- Used in real application
    - TOmCAT - terabit space laser communication, controlling mirrors at 5000fps, < 3μs latency

# What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language
- Has specific support for Xilinx and Intel FPGAs.
- Used in real application
  - TOmCAT - terabit space laser communication, controlling mirrors at 5000fps, < 3μs latency
  - Google Bittide - distributed, synchronous hardware for data centres

# What is Clash

- Haskell for describing circuits
- Can produce VHDL, Verilog, SystemVerilog
- You can interact with your circuits as normal functions, in the REPL
- Has a strong type system for being precise about **space** (sizes) and **time** (cycles).
- Can wrap IP in your target language
- Has specific support for Xilinx and Intel FPGAs.
- Used in real application
    - TOmCAT - terabit space laser communication, controlling mirrors at 5000fps, < 3µs latency
    - Google Bittide - distributed, synchronous hardware for data centres
    - Lion - open source, formally verified, pipelined RISC-V CPU implementation

## Clash Types

## Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16
```

## Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16
```

## Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16

Signed n     -- signed n bit numbers
y = -3 :: Signed 7
```

## Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16

Signed n     -- signed n bit numbers
y = -3 :: Signed 7
```

# Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16

Signed n     -- signed n bit numbers
y = -3 :: Signed 7

BitVector n -- vector of n bits
b = 0b1111_1010_1010_0000 :: BitVector 16
```

# Clash Types

```haskell
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16

Signed n     -- signed n bit numbers
y = -3 :: Signed 7

BitVector n -- vector of n bits
b = 0b1111_1010_1010_0000 :: BitVector 16
```

# Clash Types

```
Unsigned n   -- unsigned n bit numbers
x = 1 :: Unsigned 16

Signed n     -- signed n bit numbers
y = -3 :: Signed 7

BitVector n -- vector of n bits
b = 0b1111_1010_1010_0000 :: BitVector 16

Fixed Unsigned n m -- Fixed point numbers with n
Fixed Signed   n m -- integral and m fractional bits
f = -1.125 :: Fixed Signed 4 4 -- a.k.a SFixed 4 4
```

## Clash Types

## Clash Types

```
a → b                 -- Functions taking in a, and returning b

inc :: Unsigned 8 → Unsigned 8
inc n = n + 1
```

# Clash Types

```
a → b               -- Functions taking in a, and returning b
inc :: Unsigned 8 → Unsigned 8
inc n = n + 1

a → b → c           -- Functions of “multiple” arguments
add :: Unsigned 8 → Unsigned 8 → Unsigned 8
add x y = x + y
```

# Clash Types

```
a → b                -- Functions taking in a, and returning b

inc :: Unsigned 8 → Unsigned 8
inc n = n + 1


 a → b → c           -- Functions of “multiple” arguments

 add :: Unsigned 8 → Unsigned 8 → Unsigned 8
 add x y = x + y


(a,b), (a,b,c), … -- tuples, of arbitrary length and type
x :: (Unsigned 7, Signed 8)
x = (127,-128)
```

# Clash Types

```
a → b                 -- Functions taking in a, and returning b

inc :: Unsigned 8 → Unsigned 8
inc n = n + 1


a → b → c             -- Functions of "multiple" arguments

add :: Unsigned 8 → Unsigned 8 → Unsigned 8
add x y = x + y


(a,b), (a,b,c), … -- tuples, of arbitrary length and type
x :: (Unsigned 7, Signed 8)
x = (127,-128)

Maybe a               -- a value with might not exist - a better NULL
data Maybe a = Nothing | Just a     -- Think valid-bit + value:
x, y :: Maybe (Unsigned 8)          -- Just (7 :: Unsigned 8) → 0b1_0000_0111
x = Just 7                          -- Nothing                → 0b0_XXXX_XXXX
y = Nothing                         -- This can save power when anded with register enable!
```

# Clash Types

```
a → b                -- Functions taking in a, and returning b
inc :: Unsigned 8 → Unsigned 8
inc n = n + 1

 a → b → c           -- Functions of “multiple” arguments
 add :: Unsigned 8 → Unsigned 8 → Unsigned 8
 add x y = x + y

(a,b), (a,b,c), … -- tuples, of arbitrary length and type
x :: (Unsigned 7, Signed 8)
x = (127,-128)

Maybe a               -- a value with might not exist - a better NULL
data Maybe a = Nothing | Just a     -- Think valid-bit + value:
x, y :: Maybe (Unsigned 8)          -- Just (7 :: Unsigned 8) → 0b1_0000_0111
x = Just 7                          -- Nothing                → 0b0_XXXX_XXXX
y = Nothing                         -- This can save power when anded with register enable!

Vec n a               -- a vector of n elements, all with type a
                      -- (inductive list in Clash, an array in hardware)
x :: Vec 3 (Unsigned 8)
x = 1 :> :> 2 :> 3 :> Nil -- there are nicer ways to build static vectors
```

## Clash Types

## Clash Types

```
Signal dom a -- Values of type a which change over time, like VHDL signal
             -- Each signal is associated with a clock domain - multiple clocks!

ones :: Signal dom (Signed 16)
ones = pure 1
-- pure :: a → Signal dom a -- produce value forever
-- produces 1,1,1,…
```

## Clash Types

```
Signal dom a -- Values of type a which change over time, like VHDL signal
             -- Each signal is associated with a clock domain - multiple clocks!

ones :: Signal dom (Signed 16)
ones = pure 1
-- pure :: a → Signal dom a -- produce value forever
-- produces 1,1,1,…

register :: a → Signal dom a → Signal dom a
         -- delay a signal by one clock cycle, with a value for resets
x :: Signal dom (Signed 16)
x = register 0 ones -- produces 0,1,1,1,…
```

# Clash Types

```
Signal dom a -- Values of type a which change over time, like VHDL signal
             -- Each signal is associated with a clock domain - multiple clocks!

ones :: Signal dom (Signed 16)
ones = pure 1
-- pure :: a → Signal dom a -- produce value forever
-- produces 1,1,1,…

register :: a → Signal dom a → Signal dom a
         -- delay a signal by one clock cycle, with a value for resets
x :: Signal dom (Signed 16)
x = register 0 ones -- produces 0,1,1,1,…

feedback -- primitive feedback can be built with register
-- produces 0,1,2,3,4,…
-- "x is equal to itself one cycle ago, plus 1, and resets to 0"
counter :: Signal dom (Signed 16)
counter =
  let x = register 0 (x + 1)
  in x
```

# Clash Types

```
Signal dom a -- Values of type a which change over time, like VHDL signal
             -- Each signal is associated with a clock domain - multiple clocks!

ones :: Signal dom (Signed 16)
ones = pure 1
-- pure :: a → Signal dom a -- produce value forever
-- produces 1,1,1,…

register :: a → Signal dom a → Signal dom a
         -- delay a signal by one clock cycle, with a value for resets
x :: Signal dom (Signed 16)
x = register 0 ones -- produces 0,1,1,1,…

feedback -- primitive feedback can be built with register
-- produces 0,1,2,3,4,…
-- “x is equal to itself one cycle ago, plus 1, and resets to 0”
counter :: Signal dom (Signed 16)
counter =
  let x = register 0 (x + 1)
  in x

fibonacci :: Signal dom (Unsigned 64)
fibonacci =
  let x = register 0 x + register 0 (register 1 x)
  in x
```

## Clash Types

# Clash Types

```
domains -- Describe clock (and reset, and enable) domains

-- Creates a type called Dom50
createDomain vSystem{vName="Dom50", vPeriod=hzToPeriod 50e6}

createDomain vSystem{vName="MokuProInst", vPeriod=hzToPeriod 300e6}

createDomain vSystem{vName="MokuProPlatform", vPeriod=hzToPeriod 325e6}


-- Can be used in Signals - but generally write code against a generic
-- domain and pick a specific domain when generating HDL code
counter :: Signal Dom50 (Unsigned 8)
counter = let x = register 0 (x+1) in x
```

# Clash Types

```
domains -- Describe clock (and reset, and enable) domains

-- Creates a type called Dom50
createDomain vSystem{vName="Dom50", vPeriod=hzToPeriod 50e6}

createDomain vSystem{vName="MokuProInst", vPeriod=hzToPeriod 300e6}

createDomain vSystem{vName="MokuProPlatform", vPeriod=hzToPeriod 325e6}


-- Can be used in Signals - but generally write code against a generic
-- domain and pick a specific domain when generating HDL code
counter :: Signal Dom50 (Unsigned 8)
counter = let x = register 0 (x+1) in x
```

Multiple domains -- Can express multiple clock domains so you don’t
                 -- accidentally cross ~~the streams~~ clock domains
                 -- **you’re not expected to understand all this!**

```
asyncFIFOSynchronizer
  :: SNat addrSize  -- ^ Size of the internally used addresses, the  FIFO contains 2^addrSize
                    -- elements.
  → Clock  wdom     -- ^ 'Clock' to which the write port is synchronised
  → Clock  rdom     -- ^ 'Clock' to which the read port is synchronised
  → Reset  wdom → Reset  rdom  -- ^ each domain's associated reset
  → Enable wdom → Enable rdom  -- ^ and enables
  → Signal rdom Bool      -- ^ Read request - get next value in next cycle
  → Signal wdom (Maybe a) -- ^ Element to insert
  → ( Signal rdom a      -- ^ ( Oldest element in the FIFO
    , Signal rdom Bool   --    , empty flag
    , Signal wdom Bool) --    , full flag)
asyncFIFOSynchronizer … = …

platformADCSynchroniser :: Signal MokuProPlatform (Maybe (Vec 4 (Signed 16)))
                        → Signal MokuProInst     (Maybe (Vec 4 (Signed 16)))
platformADCSynchroniser adc = asyncFIFOSynchronizer (SNat @4)  ...
```

# Making the type system work for you

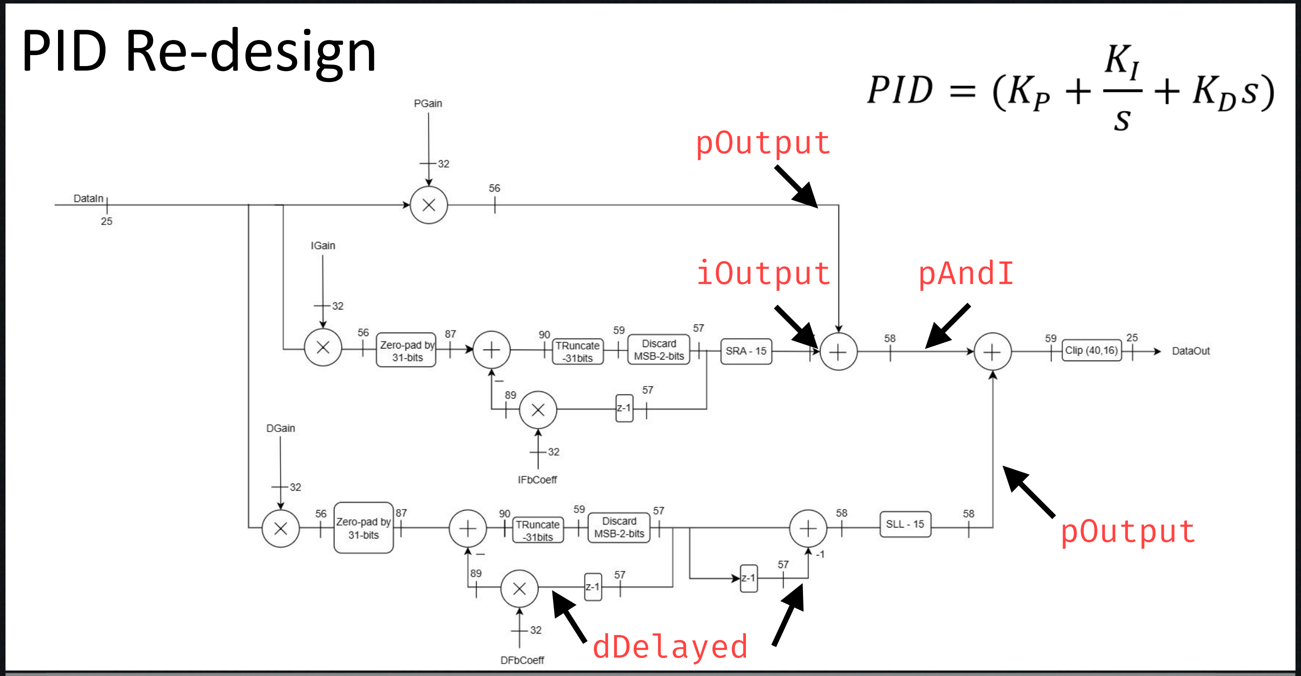
```
add, mul -- Tracking sizes in the type system

-- addition, avoiding overflow
add :: Unsigned 3 → Unsigned 5 → Unsigned 6
add :: Unsigned a → Unsigned b → Unsigned (1 + Max a b)

-- multiplication too
mul :: Unsigned 3 → Unsigned 5 → Unsigned 8
mul :: Unsigned a → Unsigned b → Unsigned (a+b)
mul :: SFixed i1 f1 → SFixed i2 f2 → SFixed (i1+i2) (f1+f2)
-- You can extend this to your own types too
```

# Making the type system work for you

```
DSignal dom n a -- tracking delays in the type system

-- Delay some signal by d cycles
delayI :: forall (d :: Nat) a n. -- usually implicit, but useful sometimes
       → a
       → DSignal dom n     a
       → DSignal dom (n+d) a
delayI init = …

-- Modelling the DSP48E1 (ish)
dsp48MulAdd :: (KnownDomain dom,HiddenClockResetEnable dom) -- ^ Ignore pls (not relevant, for now)
            ⇒ DSignal dom n     (Signed 25)
            → DSignal dom n     (Signed 18)
            → DSignal dom (n+1) (Signed 47)
            → DSignal dom (n+1) (Signed 48)
dsp48MulAdd x y a =
  let mult = delayI @1 0 (liftA2 mul x y) -- the 'forall' above lets us say what 'd' is - @1
      accumulated = liftA2 add mult a     -- mult and a now have the same delay, so we
  in accumulated                          -- we can add them

-- Apply pure/combinatorial functions to signals with the same delay
fmap, (<$>) :: (a → b)                 → DSignal dom n a → DSignal dom n b
liftA2      :: (a → b → c) → DSignal dom n a → DSignal dom n b → DSignal dom n c
```

# A PID Example

```haskell
data PID1In dom = PID1In
  { pGain     :: Signal dom (Signed 32)
  , iGain     :: Signal dom (Signed 32)
  , dGain     :: Signal dom (Signed 32)
  , iFbCoeff  :: Signal dom (Signed 32)
  , dFbCoeff  :: Signal dom (Signed 32)
  } deriving (Generic)

pid1 :: forall dom. (KnownDomain dom, HiddenClockResetEnable dom )
  ⇒ PID1In dom
  → Signal dom (Signed 25)
  → Signal dom (Signed 25)
pid1 (PID1In {..}) dataIn = output
  where
    -- Proportional
    pOutput      = dataIn *! pGain                       :: _ (Signed 57)

    -- Integral
    iAccum       = (dataIn *! iGain) +! iFeedback        :: _ (Signed 90)
    iMSBs        = msbs @59 iAccum                       :: _ (Signed 59)
    iOutput      = lsbs @57 iMSBs                        :: _ (Signed 57) -- sign ??? SRA???
    iDelayed     = register 0 iOutput                    :: _ (Signed 57)
    iFeedback    = iDelayed *! iFbCoeff                  :: _ (Signed 89)

    -- Derivative
    dgain       = dataIn *! dGain                        :: _ (Signed 57)
    dAccum      = zeroPad @31 dgain +! (-dFeedback)      :: _ (Signed 90)
    dTruncated = msbs @59 dAccum                         :: _ (Signed 59)
    dNarrowed  = lsbs @57 dTruncated                     :: _ (Signed 57)
    dDelayed    = register 0 dNarrowed
    dFeedback  = dDelayed  *! dFbCoeff                   :: _ (Signed 89)
    dPreshift  = dNarrowed +! (-dDelayed - 1)            :: _ (Signed 58) -- -1?
    dOutput = (`shiftL` 15) <$> dPreshift                :: _ (Signed 58)

    pAndI  = pOutput +! iOutput                          :: _ (Signed 58)
    pid    = pAndI   +! dOutput                          :: _ (Signed 59)

    output = clip @18 @16 <$> pid                        :: _ (Signed 25) -- This clip specifies how many MSB/LSBs get dropped
    a +! b = liftA2 add a b
    a *! b = liftA2 mul a b
```



## Case study in exploring ideas

### Squaring 64bit numbers

For a problem at work, we needed to be able to sum up the squares of 64 bit numbers on every cycle

```
var <= var + sample*sample;
```

But this doesn't work. A 64*64->128 bit multiplication can't be done in a single cycle. We can tell the synthesiser to try to spread it over several cycles

```
var0 <= var + sample*sample;

var1 <= var0;

var <= var + var1;
```

Does, but uses too many DSPs - Vivado generates 16 DSPs because it doesn't take advantage of the symmetry of squaring...

# Case study in exploring ideas

## Squaring 64bit numbers

The DSP48E1 can do a 24x17 bit unsigned multiply, so we need to split our 64bit interval into pieces that can fit into that.

If we treat our number as a polynomial:

$$N = ax^3 + bx^2 + cx + d \qquad (x = 2^{16})$$

Vivado doesn't see that the inputs are the same, so it computes:

$$\begin{aligned} N * M &= (ax^3 + bx^2 + cx + d)(Ax^3 + Bx^2 + Cx + D) \\ &= aAx^6 + aBx^5 + aCx^4 + aDx^3 + Abx^5 + Acx^4 + Adx^3 + bBx^4 \\ &\quad + bCx^3 + bDx^2 + Bcx^3 + Bdx^2 + cCx^2 + cDx + Cdx + dD \end{aligned}$$

Vivado can't see $a = A, b = B, etc$ . If we square N's polynomial, we get:

$$\begin{aligned} N^2 &= a^2x^6 + 2abx^5 + 2acx^4 + 2adx^3 + b^2x^4 \\ &\quad + 2bcx^3 + 2bdx^2 + c^2x^2 + 2cdx + d^2 \end{aligned}$$

Only 10 multiplications, because many terms appear twice (and get an extra coefficient of 2).

# Case study in exploring ideas

## Squaring 64bit numbers

The DSP48E1 can be modelled easily in clash in a few ways

A primitive multiplier which adds one cycle of delay,

```
dspMul' :: forall a b n dom c.
        ( KnownDomain dom, HiddenClockResetEnable dom
        , KnownNat a, a < 25                        -- Constrain how large the inputs can be to fit DSP48E1
        , KnownNat b, b < 18
        , KnownNat c, c ~ a+b)                     -- The output grows to accommodate the
        ⇒ DSignal dom n     (Unsigned a)  -- inputs come from some cycle 'n'
        → DSignal dom n     (Unsigned b)  -- Both inputs need to have the same delay
        → DSignal dom (n+1) (Unsigned c)  -- Outputs are produced at cycle 'n+1'
dspMul' a b = delayedI 0 (liftA2 mul a b)
```

```
dspPostadd :: forall a b n dom c. -- No registers ⇒ no constraints on dom.
    ( KnownNat a, a < 42          -- Max size of unsigned mul
    , KnownNat b, b < 48          -- Max size of unsigned post-mul adder input (Might have this wrong?)
    , c ~ 1 + Max a b)            -- Account for bit growth again
    ⇒ DSignal dom n (Unsigned a) -- If this is the result of a multiplier
    → DSignal dom n (Unsigned b) -- And this is the result of a mul-add combination
    → DSignal dom n (Unsigned c) -- these can be done without extra delay
dspPostadd = liftA2 add
```

## Case study in exploring ideas

Squaring 64bit numbers

# Case study in exploring ideas

## Squaring 64bit numbers

$$N^2 = a^2x^6 + 2abx^5 + 2acx^4 + 2adx^3 + b^2x^4$$
$$+2bcx^3 + 2bdx^2 + c^2x^2 + 2cdx + d^2$$

$x^n$ is a shift by 16n bits, and the ×2's add another shift - not very easy to keep track of - can we make the compiler do it for us?

```
-- An n-bit number, with z zeros following
-- like unsigned(N+Z-1 downto Z) (Maybe? Or Unsigned(N+Z-1 down to Z-1)?)
newtype UShifted n z
  = Shifted (Unsigned n)

-- Similar to Fixed, multiplication is simple
mul' :: UShifted a z1 → UShifted b z2 → UShifted (a+b) (z1+z2)

-- (optimal) addition is a little more complicated...
add' :: UShifted a z1 → UShifted b z2 → <exercise left to the reader>
```

# Case study in exploring ideas

## Squaring 64bit numbers

$$N^2 = a^2x^6 + 2abx^5 + 2acx^4 + 2adx^3 + b^2x^4$$
$$+2bcx^3 + 2bdx^2 + c^2x^2 + 2cdx + d^2$$

$x^n$ is a shift by 16n bits, and the ×2's add another shift - not very easy to keep track of - can we make the compiler do it for us?

```
-- An n-bit number, with z zeros following
-- like unsigned(N+Z-1 downto Z) (Maybe? Or Unsigned(N+Z-1 down to Z-1)?)
newtype UShifted n z
  = Shifted (Unsigned n)

-- Similar to Fixed, multiplication is simple
mul' :: UShifted a z1 → UShifted b z2 → UShifted (a+b) (z1+z2)

-- (optimal) addition is a little more complicated...
add' :: UShifted a z1 → UShifted b z2 → <exercise left to the reader>
```

We can handle just the cases we need for this circuit, where one argument overlaps with the LSBs of the other (and propagate bits below the LSBs)

```
add' :: UShifted a    (o + z)
     → UShifted (b + o)   z
     → UShifted (1 + Max a b + o) z
```

*A powerful type system lets you teach the compiler your expectations, and write more reusable, less error-prone code.*

# Case study in exploring ideas

## Squaring 64bit numbers

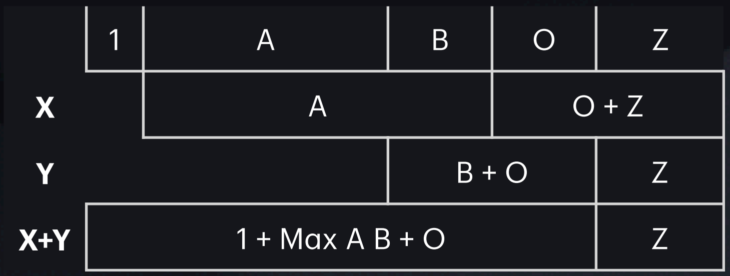
$$N^2 = a^2x^6 + 2abx^5 + 2acx^4 + 2adx^3 + b^2x^4$$
$$+2bcx^3 + 2bdx^2 + c^2x^2 + 2cdx + d^2$$

$x^n$ is a shift by 16n bits, and the ×2's add another shift - not very easy to keep track of - can we make the compiler do it for us?

```
-- An n-bit number, with z zeros following
-- like unsigned(N+Z-1 downto Z) (Maybe? Or Unsigned(N+Z-1 down to Z-1)?)
newtype UShifted n z
  = Shifted (Unsigned n)

-- Similar to Fixed, multiplication is simple
mul' :: UShifted a z1 → UShifted b z2 → UShifted (a+b) (z1+z2)

-- (optimal) addition is a little more complicated...
add' :: UShifted a z1 → UShifted b z2 → <exercise left to the reader>
```

We can handle just the cases we need for this circuit, where one argument overlaps with the LSBs of the other (and propagate bits below the LSBs)

```
add' :: UShifted a    (o + z)
     → UShifted (b + o)  z
     → UShifted (1 + Max a b + o) z
```



*A powerful type system lets you teach the compiler your expectations, and write more reusable, less error-prone code.*

## Case study in exploring ideas

Squaring ~~64bit~~ 32bit numbers

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
```

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32) -- input
        → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
```

[redacted]

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32) -- input
        → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        ⇒ DSignal dom m     (UShifted a z)
        → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32) -- input
        → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        ⇒ DSignal dom m     (UShifted a z)
        → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input              ::   DSignal dom n (Unsigned 16, Unsigned 16)
```

## Case study in exploring ideas

### Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
       => DSignal dom n     (Unsigned 32) -- input
       -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
         => DSignal dom m     (UShifted a z)
         -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input               ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                                 , DSignal dom n (Unsigned 16))
```

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32) -- input
        → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        ⇒ DSignal dom m     (UShifted a z)
        → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input               ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                                 , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ─────────────────────V
    a = Shifted <$> hi'                       ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                       ::   DSignal dom n (UShifted 16  0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
         ⇒ DSignal dom n     (Unsigned 32) -- input
         → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
         ⇒ DSignal dom m     (UShifted a z)
         → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input               ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                                 , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                       ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                       ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                            ::   DSignal dom (n+1) (UShifted 32 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        => DSignal dom m     (UShifted a z)
        -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input               ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                               , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                       ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                       ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                            ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)          ::   DSignal dom (n+2) (UShifted 32 16)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32) -- input
        → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        ⇒ DSignal dom m     (UShifted a z)
        → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input              ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts            :: ( DSignal dom n (Unsigned 16)
                                                , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                      ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                      ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                           ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)         ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                 ::   DSignal dom (n+2) (UShifted 32 17)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        => DSignal dom m     (UShifted a z)
        -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input              ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts            :: ( DSignal dom n (Unsigned 16)
                                                , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ─────────────────────V
    a = Shifted <$> hi'                      ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                      ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                           ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)         ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                 ::   DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2   -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq)  ::   DSignal dom (n+3) (UShifted 50 0)
```

[redacted]

[redacted]

[redacted]

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        => DSignal dom m     (UShifted a z)
        -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input                 ::    DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts               ::  ( DSignal dom n (Unsigned 16)
                                                    , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ---------------------V
    a = Shifted <$> hi'                         ::    DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                         ::    DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                              ::    DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)            ::    DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                    ::    DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2    -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq)     ::    DSignal dom (n+3) (UShifted 50 0)

    -- a^2x^2    -- Delay a by two cycles
    aSq = mul' (del @3 a) (del @3 a)            ::    DSignal dom (n+4) (UShifted 32 32)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
        => DSignal dom m     (UShifted a z)
        -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input             ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts           :: ( DSignal dom n (Unsigned 16)
                                               , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                     ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                     ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                          ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)        ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                ::   DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2    -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq) ::   DSignal dom (n+3) (UShifted 50 0)

    -- a^2x^2    -- Delay a by two cycles
    aSq = mul' (del @3 a) (del @3 a)        ::   DSignal dom (n+4) (UShifted 32 32)

    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' (del aSq) (del ab2bSq) ::   DSignal dom (n+5) (UShifted 65 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
         => DSignal dom n     (Unsigned 32) -- input
         -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
         => DSignal dom m     (UShifted a z)
         -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input               ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                                 , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  --------------------V
    a = Shifted <$> hi'                       ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                       ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                            ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)          ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                  ::   DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2   -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq)   ::   DSignal dom (n+3) (UShifted 50 0)

    -- a^2x^2   -- Delay a by two cycles
    aSq = mul' (del @3 a) (del @3 a)          ::   DSignal dom (n+4) (UShifted 32 32)

    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' (del aSq) (del ab2bSq)   ::   DSignal dom (n+5) (UShifted 65 0)

    -- Convert back to Unsigned 64 (no more delay because we're just giving shuffling bits)
    asUnsigned = toUnsigned <$> aSqab2bSq     ::   DSignal dom (n+5) (Unsigned 65)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32) -- input
        -> DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del :: forall d m a z. (KnownNats [a,z,d])
        => DSignal dom m     (UShifted a z)
        -> DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input                ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts              :: ( DSignal dom n (Unsigned 16)
                                                  , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  --------------------V
    a = Shifted <$> hi'                        ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                        ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                             ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)           ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                   ::   DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2   -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq)    ::   DSignal dom (n+3) (UShifted 50 0)

    -- a^2x^2   -- Delay a by two cycles
    aSq = mul' (del @3 a) (del @3 a)           ::   DSignal dom (n+4) (UShifted 32 32)

    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' (del aSq) (del ab2bSq)    ::   DSignal dom (n+5) (UShifted 65 0)

    -- Convert back to Unsigned 64 (no more delay because we're just giving shuffling bits)
    asUnsigned = toUnsigned <$> aSqab2bSq      ::   DSignal dom (n+5) (Unsigned 65)
    -- Truncate top bit - (maxBound :: Unsigned n)^2 never has its top bit set.
    output = truncateB <$> asUnsigned          ::   DSignal dom (n+5) (Unsigned 64)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute (ax + b)^2, one operation per cycle
-- = a^2x^2 + 2abx + b^2
square32Slow :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
         ⇒ DSignal dom n     (Unsigned 32) -- input
         → DSignal dom (n+5) (Unsigned 64) -- output
square32Slow input = output
  where
    del  :: forall d m a z. (KnownNats [a,z,d])
         ⇒ DSignal dom m     (UShifted a z)
         → DSignal dom (m+d) (UShifted a z)
    del = delayedI (Shifted 0)
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> input              ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts            :: ( DSignal dom n (Unsigned 16)
                                                , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                      ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                      ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                           ::   DSignal dom (n+1) (UShifted 32 0)

    -- abx
    ab  = mul' (del @1 a) (del @1 b)         ::   DSignal dom (n+2) (UShifted 32 16)
    -- 2abx  -- shifts are free, they happen in the type system at compile time
    ab2 = shiftedL @1 <$> ab                 ::   DSignal dom (n+2) (UShifted 32 17)
    -- 2abx + b^2    -- Delay just to be safe
    ab2bSq = add' (del @1 ab2) (del @2 bSq)  ::   DSignal dom (n+3) (UShifted 50 0)

    -- a^2x^2    -- Delay a by two cycles
    aSq = mul' (del @3 a) (del @3 a)         ::   DSignal dom (n+4) (UShifted 32 32)

    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' (del aSq) (del ab2bSq)  ::   DSignal dom (n+5) (UShifted 65 0)

    -- Convert back to Unsigned 64 (no more delay because we're just giving shuffling bits)
    asUnsigned = toUnsigned <$> aSqab2bSq    ::   DSignal dom (n+5) (Unsigned 65)
    -- Truncate top bit - (maxBound :: Unsigned n)^2 never has its top bit set.
    output = truncateB <$> asUnsigned        ::   DSignal dom (n+5) (Unsigned 64)
```

Almost all these type annotations can be inferred - included for the presentation

(though they can be useful to check your assumptions)

## Case study in exploring ideas

Squaring ~~64bit~~ 32bit numbers

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
[redacted]

[redacted]

[redacted]

[redacted]
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                :: DSignal dom n (Unsigned 16, Unsigned 16)
```

[redacted]

[redacted]

[redacted]

[redacted]

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                ::    DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                    ::    DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                    ::    DSignal dom n (UShifted 16  0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                  ::    DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts            :: ( DSignal dom n (Unsigned 16)
                                               , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                      ::    DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                      ::    DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                           ::    DSignal dom (n+1) (UShifted 32 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                 ::    DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts           :: (  DSignal dom n (Unsigned 16)
                                               ,  DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ---------------------V
    a = Shifted <$> hi'                     ::    DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                     ::    DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                          ::    DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                          ::    DSignal dom (n+1) (UShifted 32 16)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--    (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                :: DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                             , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                    ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                    ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                         ::   DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                         ::   DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab               ::   DSignal dom (n+1) (UShifted 32 17)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--   (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32)
        -> DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                 ::  DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts           :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ---------------------V
    a = Shifted <$> hi'                     ::  DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                     ::  DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                          ::  DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                          ::  DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab                ::  DSignal dom (n+1) (UShifted 32 17)
    -- a^2x^2
    aSq = mul' (del a) (del a)              ::  DSignal dom (n+2) (UShifted 32 32)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--   (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        ⇒ DSignal dom n     (Unsigned 32)
        → DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                ::    DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ────────────────────V
    a = Shifted <$> hi'                    ::    DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                    ::    DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                         ::    DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                         ::    DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab               ::    DSignal dom (n+1) (UShifted 32 17)
    -- a^2x^2
    aSq = mul' (del a) (del a)             ::    DSignal dom (n+2) (UShifted 32 32)

    -- 2abx + b^2
    ab2bSq = add' ab2 bSq                  ::    DSignal dom (n+1) (UShifted 50 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--   (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n      (Unsigned 32)
        -> DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                   ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts             :: ( DSignal dom n (Unsigned 16)
                                                 , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  --------------------V
    a = Shifted <$> hi'                       ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                       ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                            ::   DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                            ::   DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab                  ::   DSignal dom (n+1) (UShifted 32 17)
    -- a^2x^2
    aSq = mul' (del a) (del a)                ::   DSignal dom (n+2) (UShifted 32 32)

    -- 2abx + b^2
    ab2bSq = add' ab2 bSq                     ::   DSignal dom (n+1) (UShifted 50 0)
    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' aSq (del ab2bSq)         ::   DSignal dom (n+2) (UShifted 65 0)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--   (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n      (Unsigned 32)
        -> DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ---------------------V
    a = Shifted <$> hi'                    ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                    ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                         ::   DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                         ::   DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab               ::   DSignal dom (n+1) (UShifted 32 17)
    -- a^2x^2
    aSq = mul' (del a) (del a)             ::   DSignal dom (n+2) (UShifted 32 32)

    -- 2abx + b^2
    ab2bSq = add' ab2 bSq                  ::   DSignal dom (n+1) (UShifted 50 0)
    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' aSq (del ab2bSq)      ::   DSignal dom (n+2) (UShifted 65 0)
    -- Convert back to Unsigned 65
    asUnsigned = toUnsigned <$> aSqab2bSq  ::   DSignal dom (n+2) (Unsigned 65)
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```haskell
-- Compute
--   (ax + b)^2
-- = a^2x^2 + 2abx + b^2
square32 :: forall dom n. (KnownDomain dom, HiddenClockResetEnable dom)
        => DSignal dom n     (Unsigned 32)
        -> DSignal dom (n+2) (Unsigned 64)
square32 i = output
  where
    -- Break into 16 bit pieces - can coerce between types with the same number of bits
    parts = bitCoerce <$> i                ::   DSignal dom n (Unsigned 16, Unsigned 16)

    -- Split into two signals for the low and high halves
    (hi', lo') = D.unbundle parts          :: ( DSignal dom n (Unsigned 16)
                                              , DSignal dom n (Unsigned 16))

    -- Use the UShifted type to track implicit shifts  ---------------------V
    a = Shifted <$> hi'                    ::   DSignal dom n (UShifted 16 16)
    b = Shifted <$> lo'                    ::   DSignal dom n (UShifted 16  0)

    -- b^2
    bSq = mul' b b                         ::   DSignal dom (n+1) (UShifted 32 0)
    -- abx
    ab  = mul' a b                         ::   DSignal dom (n+1) (UShifted 32 16)
    -- 2abx
    ab2 = shiftedL @1 <$> ab               ::   DSignal dom (n+1) (UShifted 32 17)
    -- a^2x^2
    aSq = mul' (del a) (del a)             ::   DSignal dom (n+2) (UShifted 32 32)

    -- 2abx + b^2
    ab2bSq = add' ab2 bSq                  ::   DSignal dom (n+1) (UShifted 50 0)
    -- a^2x^2 + 2abx + b^2
    aSqab2bSq = add' aSq (del ab2bSq)      ::   DSignal dom (n+2) (UShifted 65 0)
    -- Convert back to Unsigned 65
    asUnsigned = toUnsigned <$> aSqab2bSq  ::   DSignal dom (n+2) (Unsigned 65)
    -- Truncate top bit - (maxBound :: Unsigned n)^2 never has its top bit set.
    output = truncateB <$> asUnsigned      ::   DSignal dom (n+2) (Unsigned 64)
```

## Case study in exploring ideas

Squaring ~~64bit~~ 32bit numbers

## Case study in exploring ideas

### Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]
```

## Case study in exploring ideas

### Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0,0,0,0] <> Prelude.map (\a -> mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
```

## Case study in exploring ideas

### Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0,0,0,0] <> Prelude.map (\a → mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0,0,0,0] <> Prelude.map (\a -> mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> simulateN (16+2) (toSignal . square32 @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0,0,0,0] <> Prelude.map (\a -> mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> simulateN (16+2) (toSignal . square32 @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]
```

# Case study in exploring ideas

## Squaring ~~64bit~~ 32bit numbers

```
clashi> :l src/LI/Moku/Square64.hs
[1 of 1] Compiling Li.Moku.Square64 ( src/LI/Moku/Square64.hs, interpreted )
Ok, one module loaded.

clashi> simulateN (16+5) (toSignal . square32Slow @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0,0,0,0] <> Prelude.map (\a -> mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
[0,0,0,0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> simulateN (16+2) (toSignal . square32 @XilinxSystem  . fromSignal) ([1..10] <> [maxBound - 5 .. maxBound])
[0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]

clashi> [0,0] <> Prelude.map (\a -> mul a a) ([1..10] <> [maxBound - 5 .. maxBound :: Unsigned 32])
[0,0,1,4,9,16,25,36,49,64,81,100,18446744022169944100,18446744030759878681,18446744039349813264,18446744047939747849,
18446744056529682436,18446744065119617025]
```

## More examples & some docs

- An implementation of Liquid Instruments' early neural network instrument, which tracks delay based on matrix size and number of hidden layers.
- Resources
  - Clash website
  - Tutorial
  - Book: Retrocomputing with Clash
- Lion, a RISC-V CPU
- Some interesting things in the Clash docs (depending on time)