Quantum Computing with Haskell

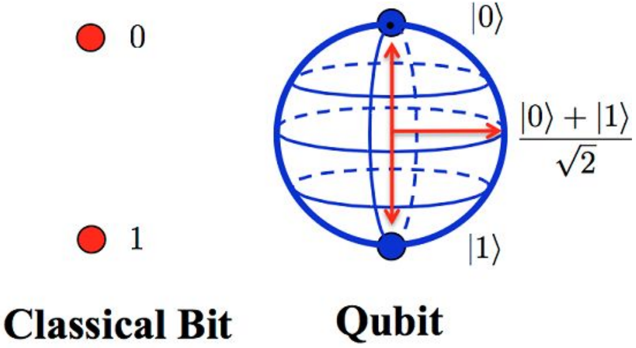
and FPGA simulation

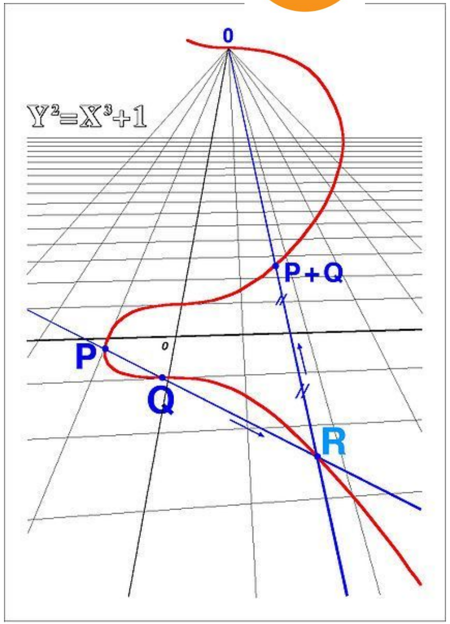
shuchang.zhou@gmail.com

Jan. 18, 2018

Why quantum computing?

- Can crack elliptic curve cryptography …
    - And threaten your Bitcoin





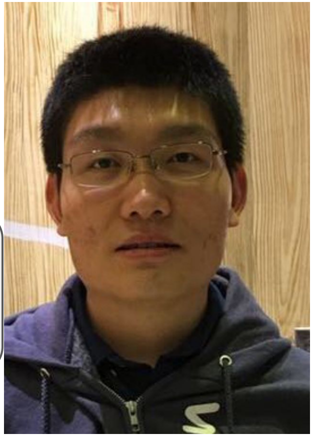
Why study quantum computing
… even when you don't have a quantum computer

- Many fast classic algorithms can be traced to simulations of quantum algorithms.

Discrete Fourier Transform	Simulated Annealing	Probabilistic checking	BPP
Quantum Fourier Transform	Quantum Annealing	Deutsch's algorithm	BQP

You ***steal*** the joy of quantum computing!

We ***preempt*** the benefits of quantum computing!



Why study quantum computing
… even when you don't have a quantum computer

"*Quantum computing may be the key, to understanding Deep Learning.*"

-- Andrew Yao, 2017

Quantum Mechanics

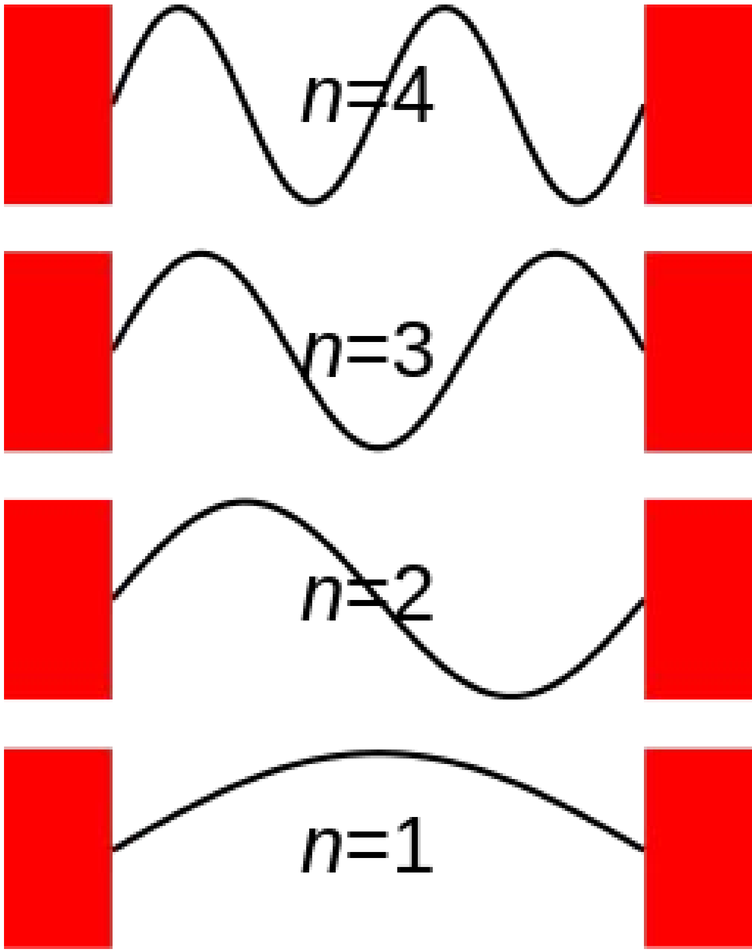
- Schrödinger's equation

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \left[\frac{-\hbar^2}{2\mu}\nabla^2 + V(\mathbf{r},t)\right]\Psi(\mathbf{r},t)$$

- von Neumann's equation
  - Lax-pair, isospectral

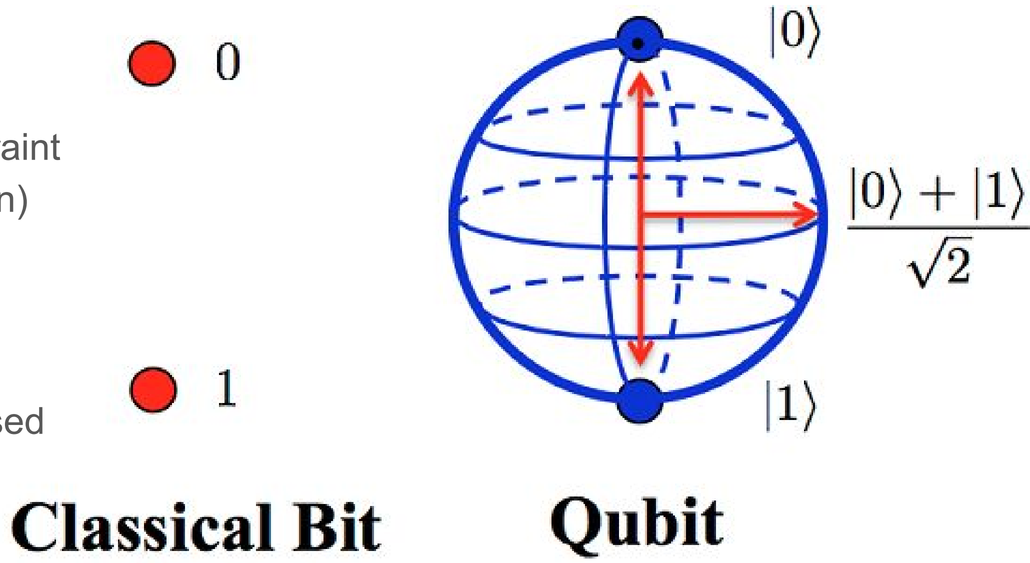
$$i\hbar\frac{\partial\rho}{\partial t} = [H,\rho]\ ,$$

Pure states (eigenvectors)



Bit to Qubit ("Q-bit")

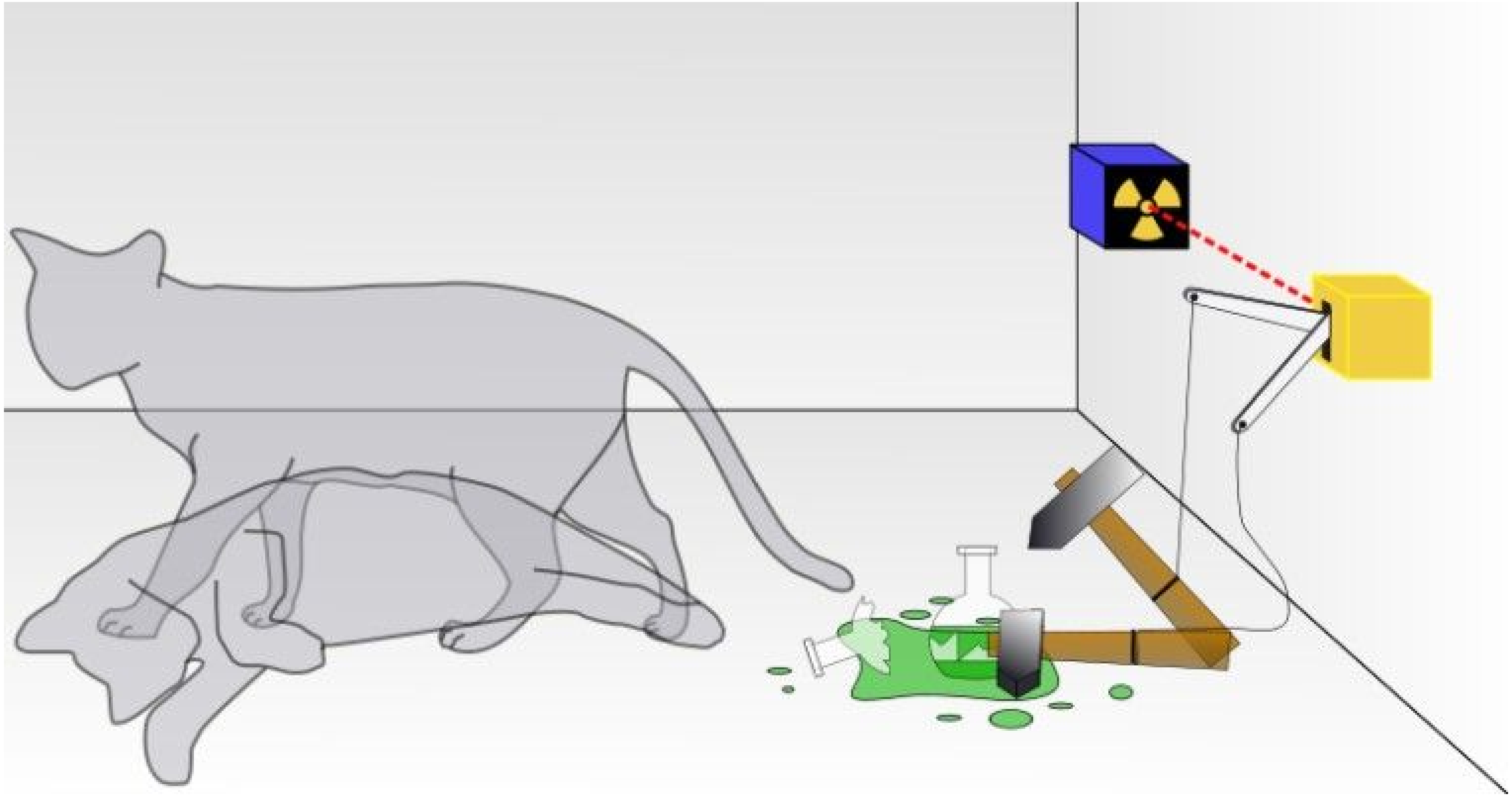
- Bit: +1 / -1
- Qubit: 2DOF
  - Angles: theta / phi
  - Two complex numbers + norm constraint
    - $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$ (superposition)
    - $|\alpha|^2 + |\beta|^2 = 1.$
    - $|\alpha|^2$ and $|\beta|^2$ are probabilities
- Bra-ket notation
  - <x| for Bra, or row-vector, or transposed vector.
  - |x> for vector



Exponential number of bits for simulating Qubits

- N qubits need 2^N classic bits
- (a |0> + b|1>)(c |0> + d|1>)
  - = ac |00> + ad |01> + bc |10> + bd |11>
- Entanglement: when (a, b; c, d) not rank 1
  - a |00> + b |01> + c |10> + d|11>

Superposition and Measurement



Quantum operations

- Reversible
    - Apply a unitary transform: all kinds of gates
        - Unconditional
        - Conditional
- Irreversible (Quantum decoherence)
    - “Create” a qubit
    - Measurement

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$ Hadamard

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$ Pauli-X

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$ Pauli-Y

$$R_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$ Phase Shift, pi / 4 for “pi-over-eight” gate

…

$$\mathrm{CNOT} = cX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

…

Modeling quantum operations with Haskell

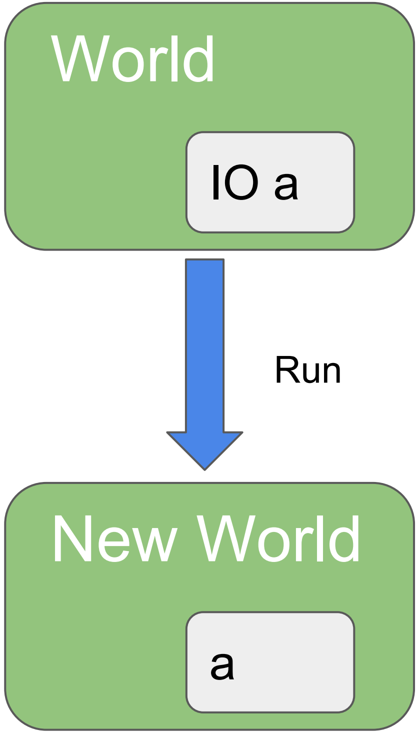
- Reversible
  - Apply a unitary transform
    - Unconditional
    - Conditional
- Irreversible
  - “Create” a qubit
  - Measurement

Has side-effects, how to model?

```haskell
-- | The underlying data type of a U unitary operation
data U = UReturn -- A List like construct
       | Rot Qbit Rotation U
       | Swap Qbit Qbit U
       | Cond Qbit (Bool -> U) U
       | Ulet Bool (Qbit -> U) U
```

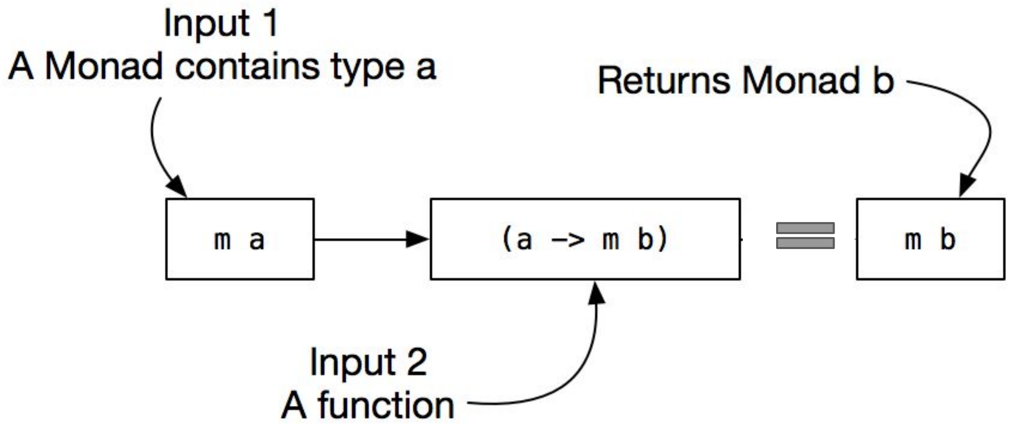
Monad and the “multi-world”

- Monad is a representation for computation graph
  - Construct first, run later
    - Haskell put everything with side-effects in IO monad
- putStrLn :: String -> IO ()
  - “write : World -> Filename -> String -> World”
- type IO a = World -> (a, World)



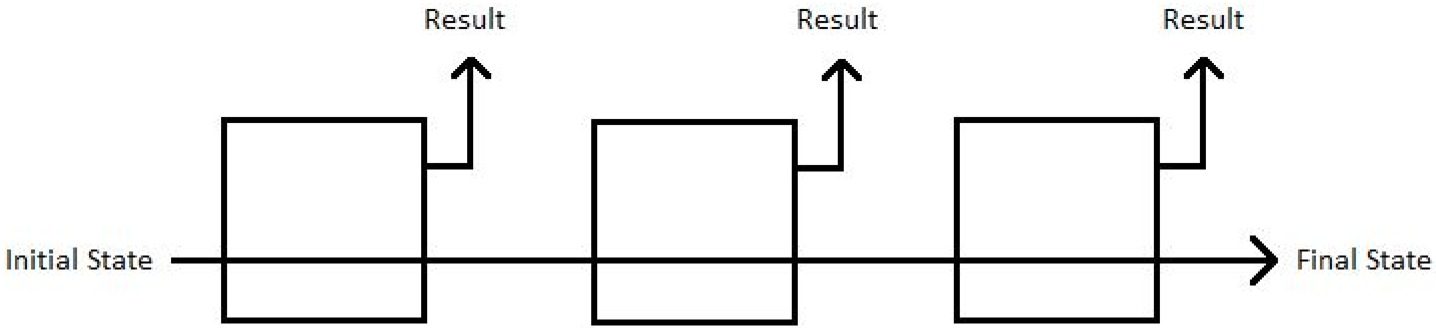
Example: IO Monad

- (>>=) :: IO a -> (a -> IO b) -> IO b
  (action1 >>= action2) world0 =
    let (a, world1) = action1 world0
        (b, world2) = action2 a world1
    in (b, world2)
  - "Bind" operation
- return :: a -> IO a
  return a world0 = (a, world0)



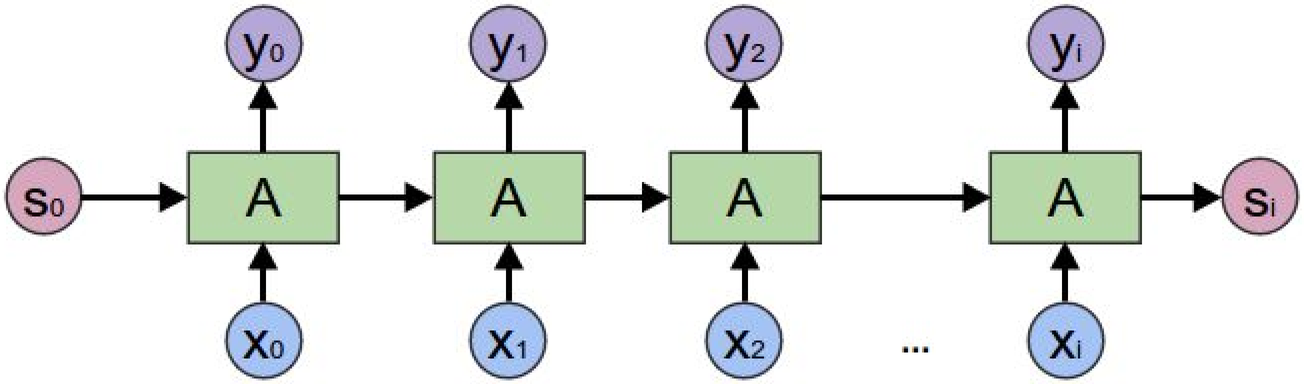
Example: State Monad

- newtype State s a = State { runState :: s -> (a, s) }
- return a = State $ \s -> (a, s)
- (>>=) :: State s a ->(a -> State s b) ->State s b
  - m >>= k = State $ \s -> let (a, s') = runState m s in runState (k a) s’



Example: “RNN monad”

- newtype Rnn s i a = Rnn { runRnn :: (i, s) -> (a, s) }
- return a = Rnn $ \(i, s) -> (a, s)
- (>>=) :: Rnn s i a -> (a -> Rnn s j b) -> Rnn s (i, j) b
  - m >>= k = Rnn $ \((i, j), s) -> let (a, s') = runRnn m (i, s) in runRnn (k a) (j, s’)



QIO Haskell package

- QIO models the “irreversible” part: decoherence of the qubits
  - Forming a monad

```
instance Monad QIO
  mkQbit :: Bool → QIO Qbit
  applyU :: U → QIO ()
  measQbit :: Qbit →QIO Bool
```

QIO Monad can be simulated or sampled

- “run” for sampling
- “sim” for distributional representation

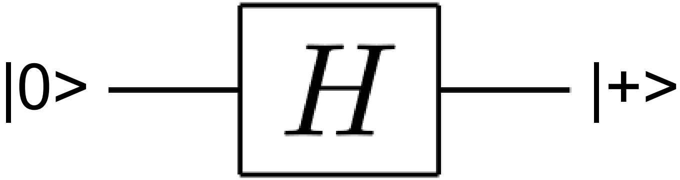
Prob :: $*\rightarrow*$
instance Monad Prob
  run :: QIO a → IO a
  sim :: QIO a → Prob a
  runC :: QIO a → a

Creating qubits

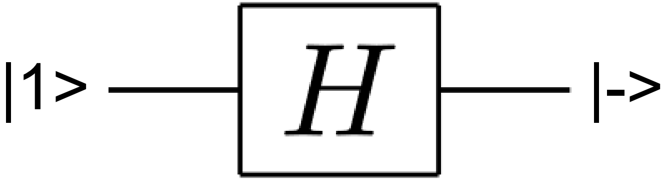
```haskell
-- | Initialise a qubit in the |0> state
q0 :: QIO Qbit
q0 = mkQ False

-- | Initialise a qubit in the |1> state
q1 :: QIO Qbit
q1  =  mkQ True

-- | Initialise a qubit in the |+> state. This is done by applying a Hadamard gate to the |0> state.
qPlus :: QIO Qbit
qPlus  =  do  qa <- q0
              applyU (uhad qa)
              return qa
```

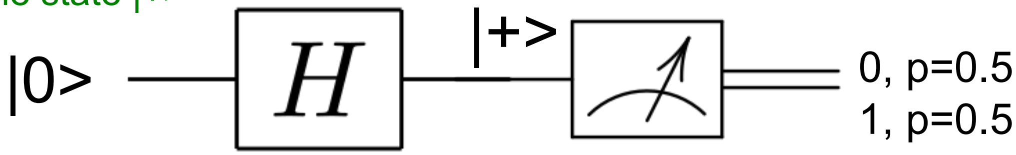



```haskell
-- | Initialise a qubit in the |-> state. This is done by applying a Hadamard gate to the |1> state.
qMinus :: QIO Qbit
qMinus  =  do  qa <- q1
               applyU (uhad qa)
               return qa
```

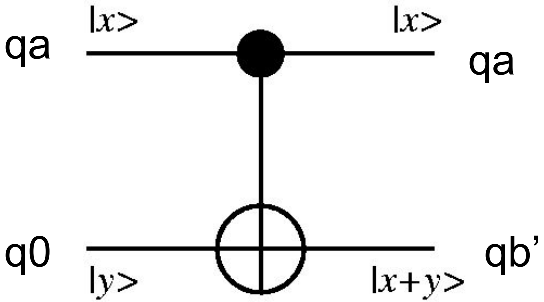


Measuring and “sharing”

```
-- | Create a random Boolean value, by measuring the state |+>
randBit :: QIO Bool
randBit  =  do  qa <- qPlus
                x <- measQbit qa
                return x
```




```
-- | This function can be used to "share" the state of one qubit, with another
-- newly initialised qubit. This is not the same as "cloning", as the two qubits
-- will be in an entangled state. "sharing" is achieved by simply initialising
-- a new qubit in state |0>, and then applying a controlled-not to that qubit,
-- depending on the state of the given qubit.
share :: Qbit -> QIO Qbit
share  qa  =  do  qb <- q0
                  applyU (cond qa (\a -> if a then (unot qb)
                                           else (mempty)  )  )
                  return qb
```



Deutsch–Jozsa's algorithm

- Given a balanced/constant boolean function (Bool^k -> Bool)
  - ***Do a 2-classification***

const True	const False	\ x -> x	\ x -> not x
1	1	0	0

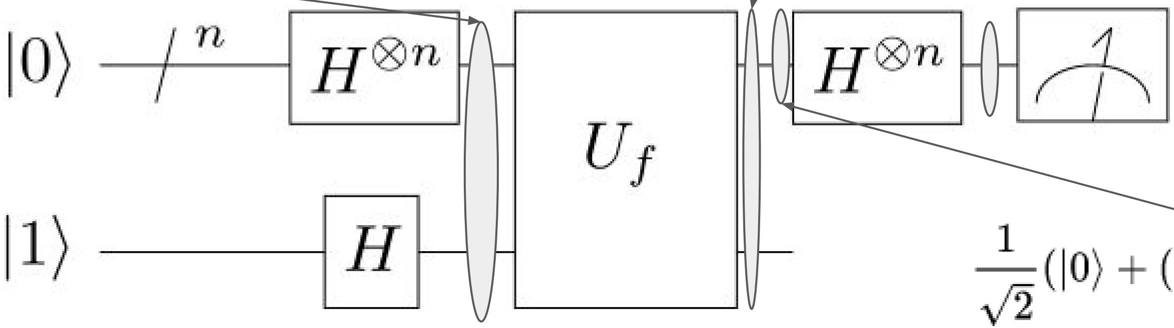
- Exact solution on a Quantum computer requires **1** evaluation
  - Exact solution on a classic computer requires exponential many evaluations
  - … But if allowing bounded errors, require k answers to obtain $\epsilon \leq 1/2^{k-1}$

Manual work out

U_f maps $|x\rangle|y\rangle$ to $|x\rangle|y \oplus f(x)\rangle$

$$\frac{1}{2}(|0\rangle(|f(0)\oplus 0\rangle - |f(0)\oplus 1\rangle) + |1\rangle(|f(1)\oplus 0\rangle - |f(1)\oplus 1\rangle))$$
$$= \frac{1}{2}((-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle))$$
$$= (-1)^{f(0)}\frac{1}{2}\left(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle\right)(|0\rangle - |1\rangle).$$

$$\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle).$$



$$\frac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle).$$

A custom quantum gate

Clash: Haskell for FPGA

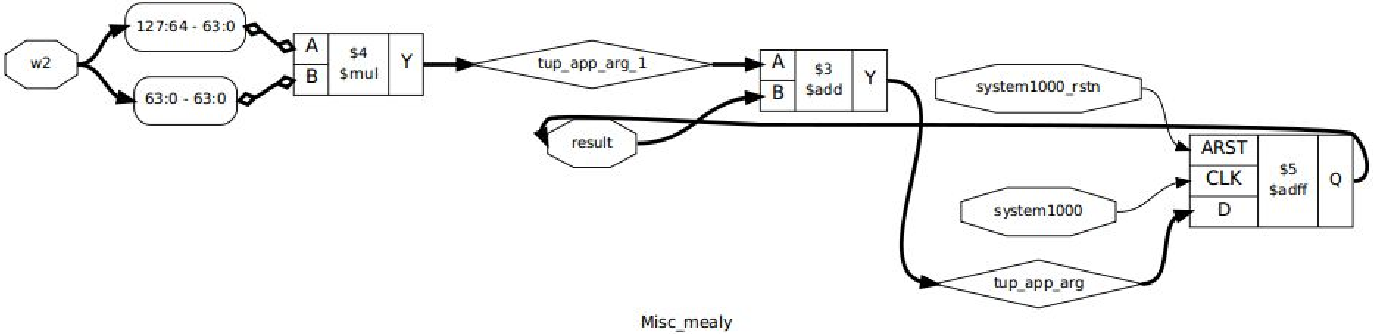
- CλaSH http://www.clash-lang.org/
  - A Haskell spin-off
- Models wires as infinite stream, and sequential logic as State machines
  - counter :: Signal (Unsigned 2)
  - counter = register 0 (liftA (+1) topEntity)
    - > sampleN 8 $ topEntity
    - [0,1,2,3,0,1,2,3]
- Dependent type for bit width (partial support)
  - Type checking for bit width checking
    - (++) :: Vec n a -> Vec m a -> Vec (n + m) a

Clash: Haskell for FPGA

mealy :: (s -> i -> (s, o)) -> s -> Signal i -> Signal o

```
mac :: Int        -- Current state
    -> (Int,Int)  -- Input
    -> (Int,Int)  -- (Updated state, output)
mac s (x,y) = (s',s)
  where s' = x * y + s
```

```
topEntity :: Signal (Int, Int) -> Signal Int
topEntity = mealy mac 0
```



Clash/FPGA: implement Complex Number

```
type CC = Vec 2 RR

c0 = 0 :> 0 :> Nil
c1 = 1 :> 0 :> Nil

sqr_norm :: CC -> RR
sqr_norm (a :> b :> Nil) = a * a + b * b

cadd :: CC -> CC -> CC
cadd = zipWith (+)

cmul :: CC -> CC -> CC
cmul (a :> b :> Nil) (c :> d :> Nil) = (a * c - b * d) :> (a * d + b * c) :>
Nil

dotProduct xs ys = foldr cadd c0 (zipWith cmul xs ys)
matrixVector m v = map (`dotProduct` v) m
```

Clash/FPGA: Qubit

```
type QBit = Vec 2 CC

q0 :: Signal QBit
q0 = register (c1 :> c0 :> Nil) q0

q1 :: Signal QBit
q1 = register (c0 :> c1 :> Nil) q1

qPlus = hadamardG q0
qMinus = hadamardG q1

hadamard :: QBit -> QBit
hadamard = matrixVector ((h :> h :> Nil) :> (h :> (cneg h) :> Nil) :> Nil)
  where h = ($$(fLit (1 / sqrt 2)) :: RR) :> 0 :> Nil

hadamardG :: Signal QBit -> Signal QBit
hadamardG = register (repeat c0) . liftA hadamard

measure :: Signal QBit -> Signal RR
measure = register 0 . liftA (\ x -> sqr_norm (x !! 1))
```

$$\begin{pmatrix} h & h \\ h & -h \end{pmatrix}$$

Multi-Qubit interaction

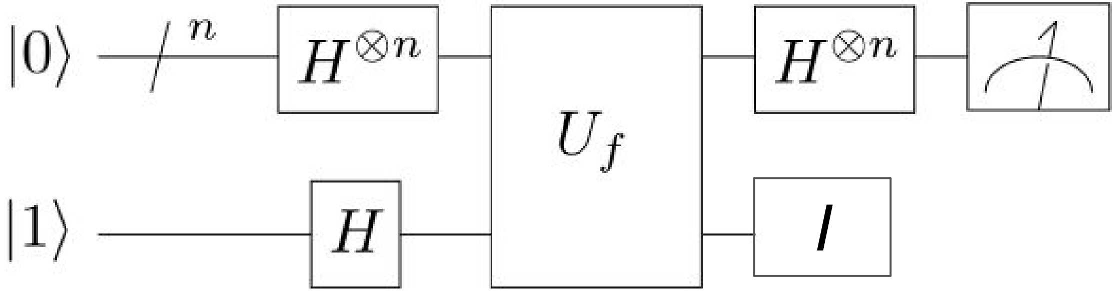
From (a|0> + b|1>) (c|0> d|1>) to
ac |00> + ad |01> + bc |10> + bd |11>

```
explode :: Signal QBit -> Signal QBit -> Signal (Vec 4 CC)
explode qx qy = register (repeat c0) $ liftA2 outer qx qy
  where
    outer :: QBit -> QBit -> Vec 4 CC
    outer (x0 :> x1 :> Nil) y = (map (cmul x0) y) ++ (map (cmul x1) y)

measure0 :: Signal (Vec 4 CC) -> Signal RR
measure0 = register 0 . liftA (\ x -> sqr_norm (x !! 0) + sqr_norm (x !! 1))
```

Measures ||00>|^2 + ||01>|^2

Deutsch-Jozsa's algorithm



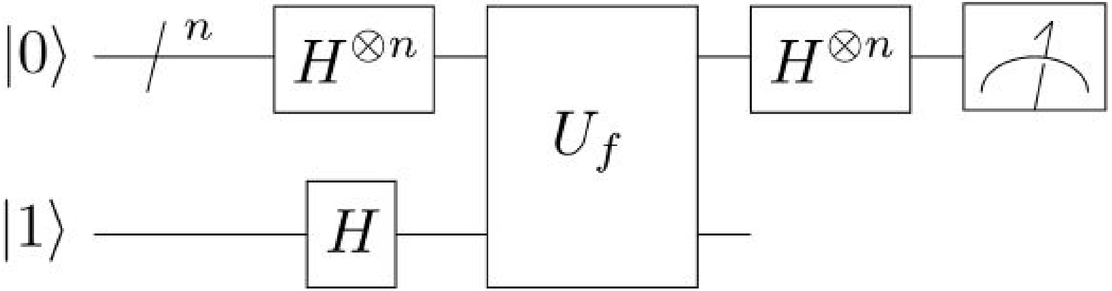

```
deutsch_u :: Vec 2 RR -> Vec 4 CC -> Vec 4 CC
deutsch_u (f0 :> f1 :> Nil) =
  matrixVector (make complex (
                  ((1 - f0) :> f1 :> 0 :> 0 :> Nil) :>
                  (f0 :> (1 - f1) :> 0 :> 0 :> Nil) :>
                  (0 :> 0 :> (1 - f0) :> f1 :> Nil) :>
                  (0 :> 0 :> f0 :> (1 - f1) :> Nil) :> Nil))
```

$$\begin{pmatrix} 1-f0 & f1 & 0 & 0 \\ f0 & 1-f1 & 0 & 0 \\ 0 & 0 & 1-f0 & f1 \\ 0 & 0 & f0 & 1-f1 \end{pmatrix}$$

```
hadamard_I :: Vec 4 CC -> Vec 4 CC
hadamard_I =
  matrixVector (make complex (
                  (h :> 0 :> h :> 0 :> Nil) :>
                  (0 :> h :> 0 :> h :> Nil) :>
                  (h :> 0 :> - h :> 0 :> Nil) :>
                  (0 :> h :> 0 :> - h :> Nil) :> Nil))
  where h = $$(fLit (1 / sqrt 2)) :: RR
```

$$H \otimes I = \begin{pmatrix} h & 0 & h & 0 \\ 0 & h & 0 & h \\ h & 0 & -h & 0 \\ 0 & h & 0 & -h \end{pmatrix}$$

Deutsch-Jozsa's algorithm

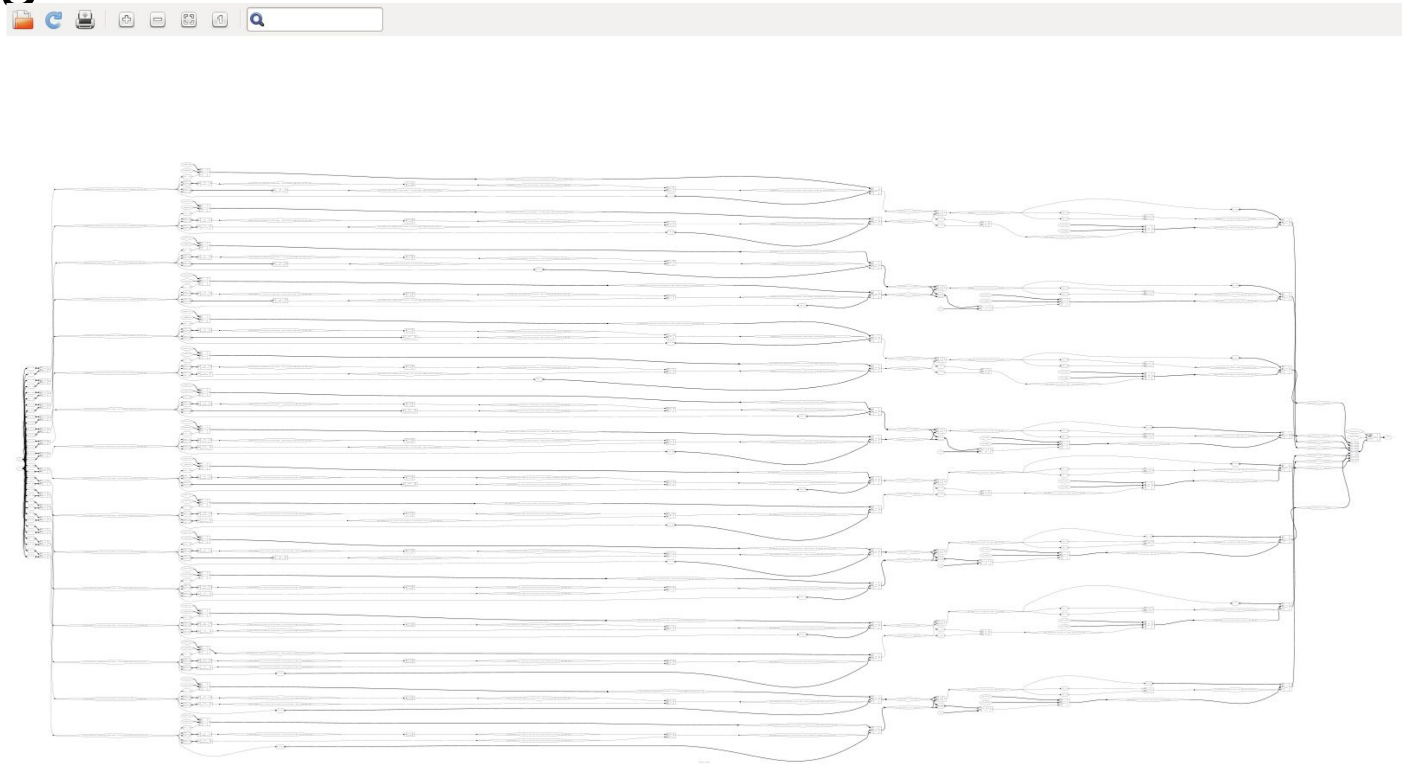



```
deutsch :: Vec 2 RR -> Signal RR
deutsch f0f1 =
  let xy = explode qPlus qMinus  in
  let xy2 = register (repeat c0) $ liftA (deutsch u f0f1) xy  in
  let xy3 = register (repeat c0) $ liftA hadamard_I xy2  in
  measure0 xy3

topEntity :: Signal (Vec 4 RR)
topEntity = bundle (map deutsch (f0 :> f1 :> f2 :> f3 :> Nil))
  where f0 = 0 :> 0 :> Nil
        f1 = 1 :> 1 :> Nil
        f2 = 0 :> 1 :> Nil
        f3 = 1 :> 0 :> Nil
sampleN 8 $ topEntity
[<0.0,0.0,0.0,0.0>,<0.0,0.0,0.0,0.0>,<0.0,0.0,0.0,0.0>,<0.0,0.0,0.0,0.0>,<0.0,0.0,0.0,0.0>
,<0.999847412109375,0.999847412109375,0.0,0.0>,<0.999847412109375,0.999847412109375,0.0,0.
0>,<0.999847412109375,0.999847412109375,0.0,0.0>]
```

Synthesizing on FPGA

Yosys Open SYnthesis Suite



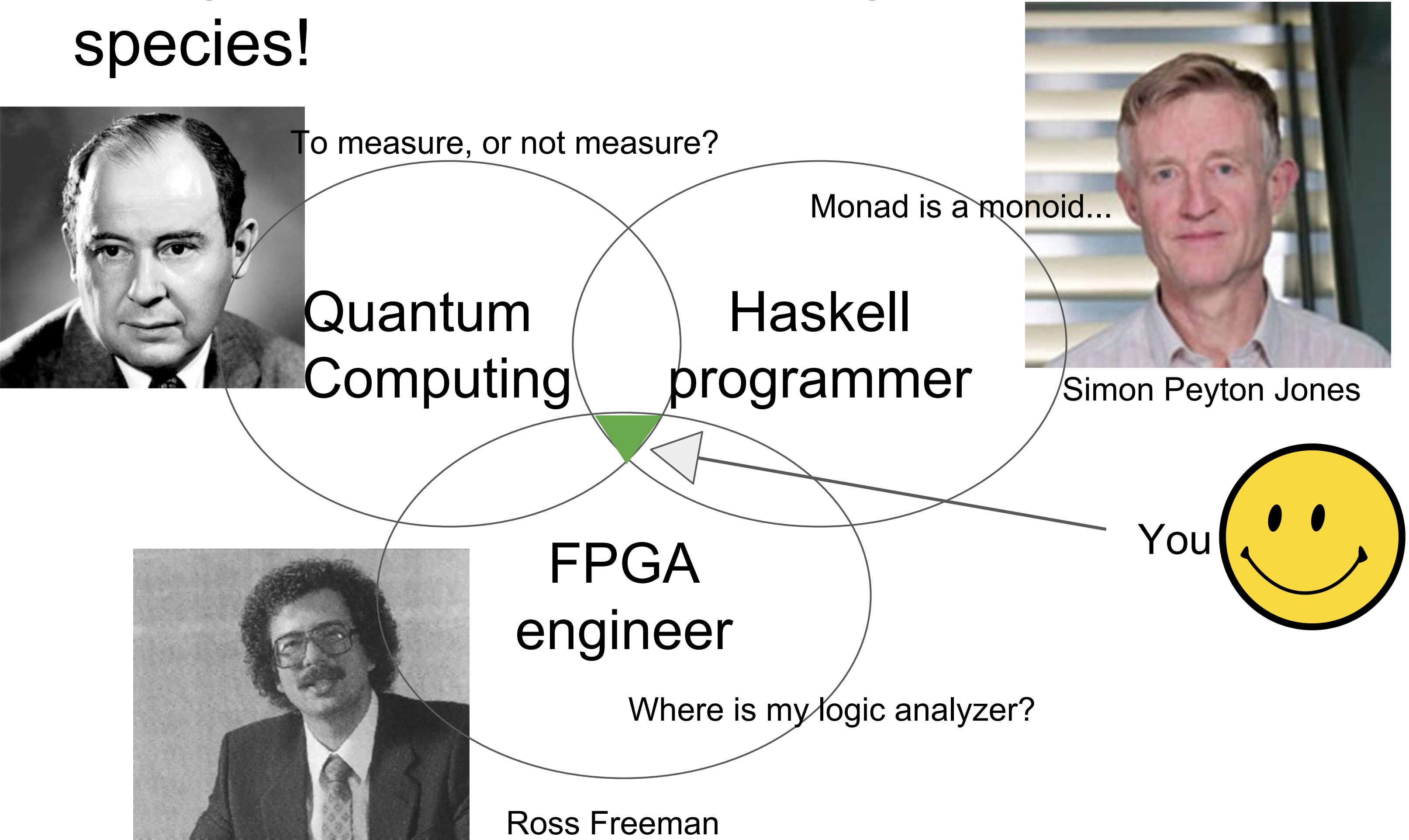
yosys> show Deutsch_explode

Problem: no usage of ALU, very resource intensive.

Future work

- Try more Quantum Computing algorithms
- Do the matrix multiplications in multiple cycles

Congratulation for becoming one of the rarest species!



Backup after this slide

Quantum Computing

- Qubit
- Inherently reversible
  - Quantum coherence exploits entanglement
- Quantum Decoherence
- Introduction to Quantum Information

Schmidt decomposition (yet another SVD)

- vector w in tensor product space H_1 \otimes H_2
- separable state
- entangled state
- Schmidt rank
- Schmidt decomposition

$$w = \sum_{i=1}^{m} \alpha_i u_i \otimes v_i.$$

- Partial trace
- von Neumann entropy

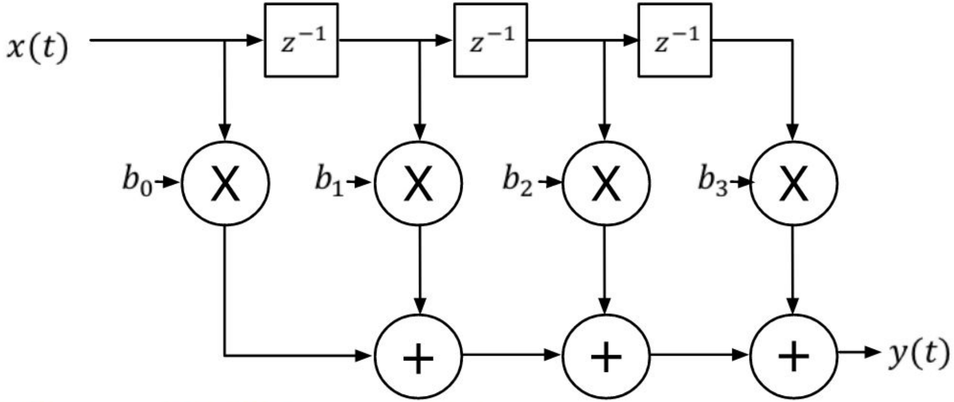
$$-\sum_i |\alpha_i|^2 \log |\alpha_i|^2$$

Quantum Physics

- matrix w with first dimension being H_1 and second being H_2
- rank 1 matrix
- rank(w) > 1
- rank
- SVD: w = U S V^T
- S^2
- entropy of square of singular values

Linear Aglebra

A Finite Input Response Filter in Clash




```
dotp :: SaturatingNum a
     => Vec (n + 1) a
     -> Vec (n + 1) a
     -> a
dotp as bs = fold boundedPlus (zipWith boundedMult as bs)


fir
  :: (Default a, KnownNat n, SaturatingNum a, HasClockReset domain gated synchronous)
  => Vec (n + 1) a -> Signal domain a -> Signal domain a
fir coeffs x_t = y_t
  where
    y_t = dotp coeffs <$> bundle xs
    xs  = window x_t
```