Hobbist FPGA development with Kansas Lava

Gergő Érdi
http://unsafePerform.IO/

Haskell.SG Meetup, May 2015.

Section 1

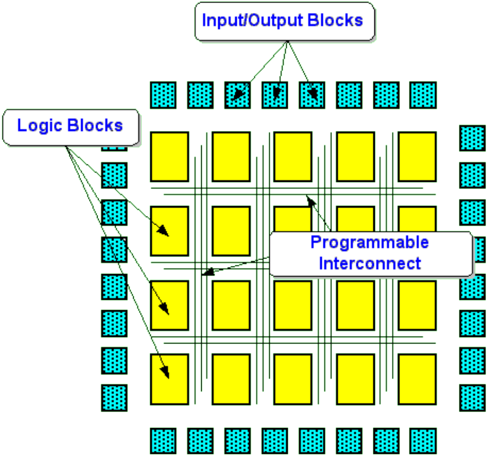
Introduction to FPGAs

What is an FPGA?

- FPGA stands for *Field-Programmable Gate Array*
- Conceptually, a bunch of logic gates that can be wired up in a software-defined configuration

What is an FPGA?

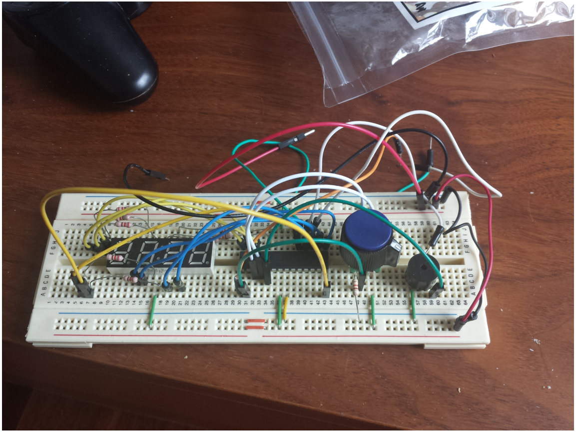
- FPGA stands for *Field-Programmable Gate Array*
- Conceptually, a bunch of logic gates that can be wired up in a software-defined configuration



From http://www.fpga-site.com/

Why would you want to use one?

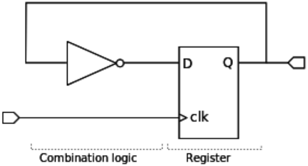
- In real life: cheap (in low volume) and fast-turnaround alternative to custom silicon chips
- For hobbists: building complicated digital circuits without all those messy wires. . .



Kitchen timer prototype by yours truly

Circuit design with FPGAs

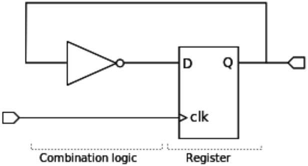
1. The FPGA configuration is usually specified on the *register-transfer level* of abstraction (RTL): combinational logic + registers



2. RTL description is turned into list of components and their connections (*netlist*)
3. Based on the actual hardware (the concrete FPGA chip), these are *mapped* to physical components on the board
4. For each component, a particular instance is chosen, and wiring routes are decided (*place-and-routing*)

Circuit design with FPGAs

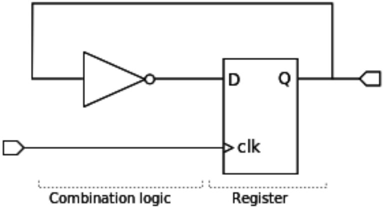
1. The FPGA configuration is usually specified on the *register-transfer level* of abstraction (RTL): combinational logic + registers



2. RTL description is turned into list of components and their connections (*netlist*)
3. Based on the actual hardware (the concrete FPGA chip), these are *mapped* to physical components on the board
4. For each component, a particular instance is chosen, and wiring routes are decided (*place-and-routing*)

In practice, you only have control over step 1, the rest is proprietary.

Hardware description languages

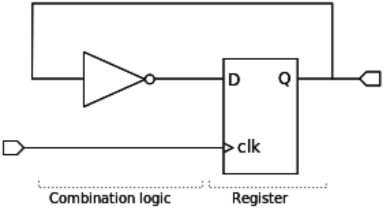


VHDL

```
D <= not Q;

process(clk)
begin
    if rising_edge(clk) then
        Q <= D;
    end if;
end process;
```

Hardware description languages

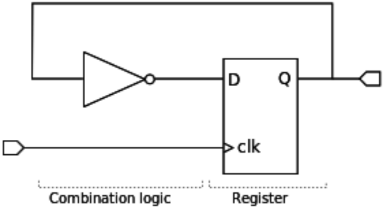


Verilog

```
d = ! q;

always @(posedge clk)
  q <= d;
```

Hardware description languages



Kansas Lava

```
q = register False d
  where
    d = bitNot q
```

Kansas Lava

```
q = register False d
  where
    d = bitNot q
```

- Both Verilog and VHDL are first-order languages with poor abstraction and type construction facilities
- Lava is a DSL embedded in Haskell
- There's a whole family of Lava forks by now; Kansas Lava seemed the most usable and recently updated when I first looked into it, circa 2012
- Haskell sharing is reified into shared wires (see *Type-Safe Observable Sharing in Haskell* and `data-reify` by Andy Gill)

Kansas Lava

```
q = register False d
  where
    d = bitNot q
```

- Both Verilog and VHDL are first-order languages with poor abstraction and type construction facilities
- Lava is a DSL embedded in Haskell
- There's a whole family of Lava forks by now; Kansas Lava seemed the most usable and recently updated when I first looked into it, circa 2012
- Haskell sharing is reified into shared wires (see *Type-Safe Observable Sharing in Haskell* and `data-reify` by Andy Gill)
- Kansas Lava and the libraries around it has had some bitrot which meant there was no way to compile the version on Hackage with GHC > 7.4
- I took over maintainance of the stable Kansas Lava branch

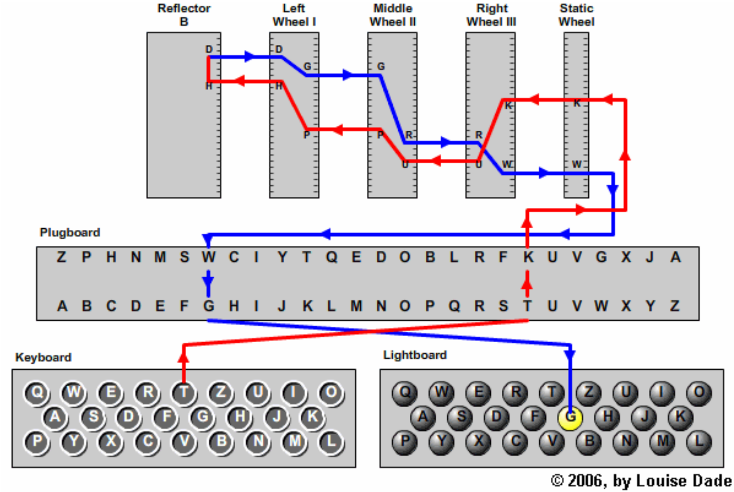
Section 2

Enigma

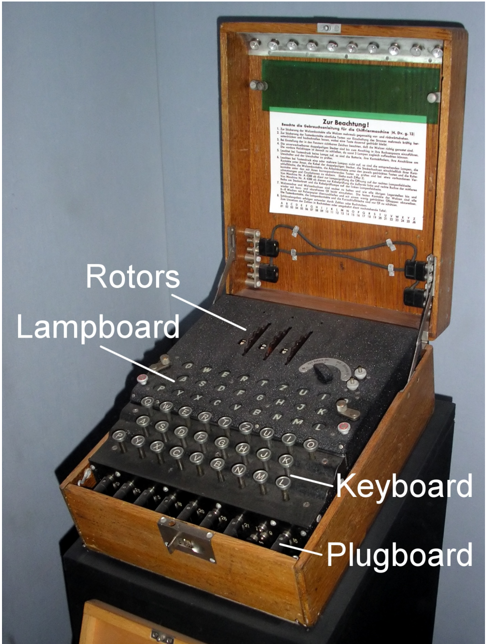
The Enigma Machine

- German symmetric-key cipher machine originally from the '20's.
- Composition of several permutations...
- ...some of which change (rotate) as the input stream is processed
- Initial configuration: plugboard, some rotors with notches, reflector
- Running configuration: rotation of rotors
- Electro-mechanical implementation

The Enigma Machine



The Enigma Machine



Enigma in Cryptol

- Cryptol is a DSL for cryptographic algorithms
- There's a Cryptol implementation of the Enigma encryption scheme in the *Programming Cryptol* book
- The idea for this talk came from Rishiyur S. Nikhil who showed how he turned the Cryptol spec into Bluespec, another, functional-ish HDL
- So let's do the same in Kansas Lava!

Enigma in Cryptol

- Cryptol is a DSL for cryptographic algorithms
- There's a Cryptol implementation of the Enigma encryption scheme in the *Programming Cryptol* book
- The idea for this talk came from Rishiyur S. Nikhil who showed how he turned the Cryptol spec into Bluespec, another, functional-ish HDL
- So let's do the same in Kansas Lava!
- I will not go into the specifics of the Cryptol implementation in this talk
- But all the function names here will match up with the Cryptol implementation

Section 3

Enigma in Kansas Lava

Code available at
https://github.com/gergoerdi/enigma-kansas-lava/

Types

Kansas Lava uses the `sized-types` package for container types that specify their size precisely in their type.

```
type Permutation a = Matrix a a
type Rotor a = Matrix a (a, Bool)

type Letter = X26
type Plugboard = Permutation Letter
type Reflector = Permutation Letter

type Decoded clk n = Matrix n (Signal clk Bool)

enigma :: (Clock clk, Size n)
       => Plugboard -> Matrix n (Rotor Letter) -> Reflector
       -> Matrix n Letter
       -> Signal clk (Enabled Letter)
       -> Signal clk (Enabled Letter)
```

Permutation

Given the decoded representation we have, a static permutation is just a reshuffling of the wires:

```
permuteBwd :: (Size n)
           => Permutation n
           -> Decoded clk n
           -> Decoded clk n
permuteBwd p = Matrix.ixmap (p !)

permuteFwd :: (Size n)
           => Permutation n
           -> Matrix n (Signal clk Bool)
           -> Matrix n (Signal clk Bool)
permuteFwd p = Matrix.ixmap $ \i ->
                 maybe (error "Not␣surjective") fst $
                 find ((== i) . snd) $ Matrix.assocs p
```

Rotating a rotor

Bit-rotation on the decoded input can be used to implement the rotating rotors.

```
rotateFwd :: (Size a, Rep a, Integral a)
          => Signal clk a -> Decoded clk a -> Decoded clk a
rotateFwd r = onBits (rol r)

rotateBwd :: (Size a, Rep a, Integral a)
          => Signal clk a -> Decoded clk a -> Decoded clk a
rotateBwd r = onBits (ror r)

onBits :: (Size n, Rep n)
     => (Signal clk (Unsigned n) -> Signal clk (Unsigned n))
     -> Decoded clk n -> Decoded clk n
onBits f = unpackMatrix . bitwise . f . bitwise . packMatrix
```

A full rotor

A full rotor consists of a rotation, a permutation, and, when the previous rotor has a notch at its current position, updating the rotation.

```haskell
type Rotor a = Matrix a (a, Bool)

rotorFwd :: (Size a, Rep a, Integral a)
         => Rotor a -> Signal clk Bool -> Signal clk a
         -> Decoded clk a
         -> (Signal clk Bool, Signal clk a, Decoded clk a)
rotorFwd rotor rotateThis r sig = (rotateNext, r', sig')
  where
    (p, notches) = (fmap fst &&& fmap snd) rotor
    rotateNext = packMatrix (pureS <$> notches) .!. r
    r' = mux rotateThis (r, loopingIncS r)
    sig' = rotateFwd r >>> permuteFwd p $ sig
```

A full rotor

A full rotor consists of a rotation, a permutation, and, when the previous rotor has a notch at its current position, updating the rotation.

```
type Rotor a = Matrix a (a, Bool)

rotorBwd :: (Size a, Rep a, Integral a)
         => Rotor a -> Signal clk a -> Decoded clk a
         -> Decoded clk a
rotorBwd rotor r = permuteBwd p >>> rotateBwd r
  where
    p = fmap fst rotor
```

Lining up the rotors

We need to thread through the signal carrying the rotation trigger:

```haskell
joinRotors :: (Size n, ...)
           => Matrix n (Rotor a)
           -> Matrix n (Signal clk a)
           -> Decoded clk a
           -> (Matrix n (Signal clk a), Decoded clk a)
joinRotors rotors rs sig = (rs', sig')
  where
    (rs', (_, sig')) = Matrix.scanR step
                        ((high, sig), zipMatrix rotors rs)
    step ((rotateThis, x), (rotor, r)) =
        let (rotateNext, r', x') = rotorFwd rotor rotateThis
        in (r', (rotateNext, x'))
```

Lining up the rotors

Again, it's much simpler backwards:

```
import qualified Data.Foldable as F

backSignal :: (Size n, ...)
           => Matrix n (Rotor a) -> Matrix n (Signal clk a)
           -> Decoded clk a
           -> Decoded clk a
backSignal rotors rs sig = F.foldr (uncurry rotorBwd) sig $
                           zipMatrix rotors rs
```

Putting it all together

```
enigmaLoop :: (Clock clk, Size n, Enum n)
      => Plugboard -> Matrix n (Rotor Letter) -> Reflector
      -> Matrix n (Signal clk Letter) -> Decoded clk Letter
      -> (Matrix n (Signal clk Letter), Decoded clk Letter)
enigmaLoop plugboard rotors reflector rs sig0 = (rs', sig5)
  where
    sig1 = permuteFwd plugboard $ sig0
    (rs', sig2) = joinRotors rotors rs sig1
    sig3 = permuteFwd reflector sig2
    sig4 = backSignal rotors rs sig3
    sig5 = permuteBwd plugboard sig4
```

Adding state

The key idea is to have a register for each rotor's rotation, which we only update when there is input available.

```haskell
enigma :: (Clock clk, Size n, Enum n)
       => Plugboard -> Matrix n (Rotor Letter) -> Reflector
       -> Matrix n Letter
       -> Signal clk (Enabled Letter)
       -> Signal clk (Enabled Letter)
enigma plugboard rotors reflector rs0 input =
  packEnabled ready letterOut
 where
  (ready, letterIn) = unpackEnabled input
  sig = decode letterIn
  (rs', sig') = enigmaLoop plugboard rotors reflector rs sig
  letterOut = encode sig'
  rs = Matrix.zipWith rReg rs0 rs'

  rReg r0 r' = fix $ \r -> register r0 $ mux ready (r, r')
```

Demo unit

For this demonstration, I'm using a Papilio One FPGA dev board based on the Xilinx Spartan 3E chip. The peripherals are built on a breadboard and are also driven by Kansas Lava code:

- Input: keyboard via a PS/2 connector
- Output: 1602 LCD with 4-bit semi-parallel interface
- Reset button