Hobbist FPGA development with Kansas Lava

Gergő Érdi
http://unsafePerform.IO/

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

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...
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

![Diagram of combinational logic and register]

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

\[ D \leftarrow \text{not} \ Q; \]

\textbf{process} (clk) \begin{align*}
\text{begin} \\
\quad \text{if rising\_edge} (\text{clk}) \text{ then} \\
\quad \quad Q \leftarrow D; \\
\quad \text{end if;} \\
\text{end process;}
\end{align*}
Hardware description languages

Verilog

\[ d = ! q; \]

\textbf{always} @@\textbf{posedge} clk \textbf{always} \textbf{posedge} clk \textbf{always} @\textbf{posedge} clk \textbf{always} @posedge clk

\[ q \leftarrow d; \]
Hardware description languages

Kansas Lava

\[ q = \text{register False d} \]
\[ \text{where} \]
\[ d = \text{bitNot q} \]
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)
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 maintenance 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

© 2006, by Louise Dade
The Enigma Machine

- Rotors
- Lampboard
- Keyboard
- Plugboard
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!
- 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/
Kansas Lava uses the sized-types package for container types that specify their size precisely in their type.

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

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

```haskell
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.assocss p
```
Rotating a rotor

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

\[
\text{rotateFwd} :: (\text{Size } a, \text{Rep } a, \text{Integral } a) \\
\quad \Rightarrow \text{Signal clk } a \rightarrow \text{Decoded clk } a \rightarrow \text{Decoded clk } a \\
\text{rotateFwd } r = \text{onBits} (\text{rol } r)
\]

\[
\text{rotateBwd} :: (\text{Size } a, \text{Rep } a, \text{Integral } a) \\
\quad \Rightarrow \text{Signal clk } a \rightarrow \text{Decoded clk } a \rightarrow \text{Decoded clk } a \\
\text{rotateBwd } r = \text{onBits} (\text{ror } r)
\]

\[
\text{onBits} :: (\text{Size } n, \text{Rep } n) \\
\quad \Rightarrow (\text{Signal clk } (\text{Unsigned } n) \rightarrow \text{Signal clk } (\text{Unsigned } n)) \\
\quad \rightarrow \text{Decoded clk } n \rightarrow \text{Decoded clk } n \\
\text{onBits } f = \text{unpackMatrix} \ . \ \text{bitwise} \ . \ f \ . \ \text{bitwise} \ . \ \text{packMatrix}
\]
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 >>> permutationFwd p $ sig
```
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)

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:

\[
\begin{align*}
\text{joinRotors} &:: (\text{Size } n, \ldots) \\
&\Rightarrow \text{Matrix } n (\text{Rotor } a) \\
&\rightarrow \text{Matrix } n (\text{Signal clk } a) \\
&\rightarrow \text{Decoded clk } a \\
&\rightarrow (\text{Matrix } n (\text{Signal clk } a), \text{Decoded clk } a) \\
\text{joinRotors} &\text{ rotors rs sig } = (rs', \text{ sig}') \\
\text{where} \\
(rs', (_, \text{ sig}')) &= \text{Matrix}\_\text{scanR} \text{ step} \\
&((\text{high}, \text{ sig}), \text{zipMatrix rotors } rs) \\
\text{step} &((\text{rotateThis, x}), (\text{rotor, r})) = \\
\text{let} & (\text{rotateNext, r', x'}) = \text{rotorFwd} \text{ rotor rotateThis} \\
\text{in} & (r', (\text{rotateNext, x'}))
\end{align*}
\]
Lining up the rotors

Again, it’s much simpler backwards:

```haskell
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

\[
\text{enigmaLoop} :: (\text{Clock} \ \text{clk}, \ \text{Size} \ n, \ \text{Enum} \ n) \\
\Rightarrow \text{Plugboard} \rightarrow \text{Matrix} \ n (\text{Rotor Letter}) \rightarrow \text{Reflector} \\
\rightarrow \text{Matrix} \ n (\text{Signal clk Letter}) \rightarrow \text{Decoded clk Letter} \\
\rightarrow (\text{Matrix} \ n (\text{Signal clk Letter}), \ \text{Decoded clk Letter})
\]

\[
\text{enigmaLoop plugboard rotors reflector rs sig0} = (rs', \ \text{sig5})
\]

where

\[
\text{sig1} = \text{permuteFwd plugboard} \ $ \ \text{sig0}
\]
\[
(rs', \ \text{sig2}) = \text{joinRotors rotors rotors rs sig1}
\]
\[
\text{sig3} = \text{permuteFwd reflector} \ \text{sig2}
\]
\[
\text{sig4} = \text{backSignal rotors rotors rs sig3}
\]
\[
\text{sig5} = \text{permuteBwd plugboard} \ \text{sig4}
\]
The key idea is to have a register for each rotor’s rotation, which we only update when there is input available.

```
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')
```
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