Handshaking in Kansas Lava using Patch Logic

Andy Gill and Bowe Neuenschwander

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
`andygill@ku.edu` and `bneuen@ku.edu`

Abstract. Designing hardware is like writing music for an orchestra - lots of pieces have to come together at the correct time for everything to work. In systems design, there is a confusing array of standards for allowing cooperating components, and little type-level support in traditional design methodologies for helping connect components with pre-arranged protocols. In this paper, we explore bringing protocol-level types to communicating processes. Inside our hardware description language Kansas Lava we introduce the notation of a patch, which is a communicating component with well-understood protocols. We build a theory round the notion of patches, which we call patch logic, and then use the patch abstraction to build a small driver for an FPGA board.

1 Introduction

When writing cooperative components for hardware fabrics like FPGAs, some form of handshaking or inter-component cooperation is required. One common solution is using central control logic. This allows for maximizing global throughput, but at the cost of composability of the sub-components. Another solution is to allow components to act independently, and throttle the communication between components using bus protocols. In this paper, we take the second approach to hardware design, and build a set of types and combinators to facilitate the construction of hardware using composition.

We program in the language Haskell [12], and design hardware using Kansas Lava [8, 7], our version of a Haskell library for describing hardware. Kansas Lava, like other the versions of Lava before it [4, 14], makes extensive use of types to describe signals. Kansas Lava has distinct types for combinatorially and sequentially generated values, a family of lifting functions to coerce between these two styles of hardware logic, and various structured types for complex signals. This paper takes this type-based approach one step further, where we express the *protocol* between components using types, and build abstractions around these types.

The contribution of this paper is explaining in detail how Lava may be successfully used in the large by utilizing both the traditional idioms and a new Lava idiom which we call patches. Of course, all circuits could be described using simple logic and structural composition. Types, higher-order functions and mathematical structures like monads give functional language-based hardware description languages the advantage of powerful and composable abstractions. Specifically, we make the following contributions.

– We introduce a new Haskell structure for constructing large circuits, a `Patch`, which facilitates the construction of cooperating components.
– We give the laws for our patches, and show how complex dataflow can be managed and connected to the interfaces of our circuits.
– To support our thesis that this new abstraction support the modular construction of circuits we give an extended example of building an LCD driver.

2 Kansas Lava

Kansas Lava is a Haskell library for simulating and generating hardware components, in the spirit of Chalmers Lava and Xilinx Lava. Like Chalmers Lava, Kansas Lava uses observable sharing [5, 6] to represent and capture cycles in hardware, but uniquely uses an explicit monad to represent external connectivity. The major novel feature of Kansas Lava is the aggressive use of types and type extensions to capture hardware restrictions and concerns. This paper represents an extension of this initiative, by giving a typed interface to protocols.

All variants of Lava are based around the idea that you can write a structural description of hardware, and then observe or extract the description of the described hardware. The classical example is the half-adder.

```
halfAdder :: Seq Bool -> Seq Bool -> (Seq Bool,Seq Bool)
halfAdder a b = (carry,sum) where
  carry = and2 a b
  sum   = xor2 a b
```

In this example, two arguments are combined to make two results, the carry and the result of the sum. The `Seq :: * -> *` type constructor lifts its argument to make it observable; a well-understood Domain Specific Language trick [11, 8, 2]. `Seq` here is a sequence of values of time, interpreted by an implicit clock, in much the same way as `signal` is typically used in VHDL.

There are several structures like `Seq` in Kansas Lava; here is a short taxonomy of the major structures.

– There are fixed width values, signed (`S`n), unsigned (`U`n), and fixed range values (`X`n).
– These values, and other built-in Haskell types, like tuples and `Bool`, can be lifted into an observable signal using `Comb`, `CSeq c`, a signal interpreted by a given clock type, or `Seq`, a signal interpreted by a global clock. For the remainder of this paper we will use `Seq` with its global clock for conciseness, the ideas presented here generalize to multi-clock designs.
– We have what we call the protocol gap, where types are used to represent protocol. Filling this gap is the subject of this paper. As an example of a protocol, `Enabled` represents an optional value implemented using an extra ENABLE status flag.

– Finally, we have the Fabrics. `Fabric` is the monad used to connect a Kansas Lava program to an external device, like a LCD screen or RJ-45 port. We have two implementations of `Fabric`: one for use on real hardware, and one for use in simulation.

These different type classifications are connected using combinators. There is a lift function that lifts pure Haskell functions into a function that operates on signals, and a fixed set of lift functions that lifts `Comb`-based functions into `Seq`-based functions. As well as these functions, many primitives are provided, and overloaded to work over `Signal`, a class that abstracts over `Comb` and `Seq`.

`Fabric` is the monad for connecting to the outside world, via specific pins on the FPGA. As an example, consider listening to physical switches and pushbuttons on a specific FPGA board, and lighting up a row of LEDs. In Kansas Lava, this would read:

```
test_leds :: Fabric ()
test_leds = do sw <- switches
               bu <- buttons
               leds (sw ‘M.append‘ bu)

-- switches, buttons and leds are provided by a board-specific prelude
switches :: Fabric (Matrix X4 (Seq Bool))
switches = do
        inp <- inStdLogicVector "SW" :: Fabric (Seq (Matrix X4 Bool))
        return (unpack inp)

buttons :: Fabric (Matrix X4 (Seq Bool))
buttons = do i0 <- inStdLogic "BTN_WEST"
             i1 <- inStdLogic "BTN_NORTH"
             i2 <- inStdLogic "BTN_EAST"
             i3 <- inStdLogic "BTN_SOUTH"
             return (matrix [i0,i1,i2,i3])

leds :: Matrix X8 (Seq Bool) -> Fabric ()
leds inp = outStdLogicVector "LED" (pack inp :: Seq (Matrix X8 Bool))
```

There is a Haskell class for each FPGA board supported, with peripherals being provided as overloaded `Fabric`-based functions. Furthermore, each board is given two instances, one for programming the physical device using an associated UCF specification, and one for programming a board simulator. In this way, programs can be developed for specific boards, and tested offline before being actually deployed.

3 Protocols

When building larger Lava circuits, we want some typed idioms to help us with our hardware development. The need for an idiom appears when allowing two components to communicate when a new datum does not appear every cycle. We

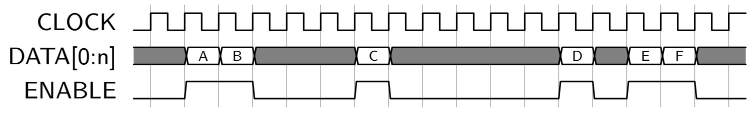

Fig. 1. The Enabled Protocol

now introduce two such well-understood idioms, built on top of `Seq`: sequences of optionally defined values, and sequences of handshaken values.

3.1 The `Enabled` protocol

The basic way of arranging one-direction optional communication is marking data as valid or invalid at each clock cycle. In Kansas Lava, we call this protocol the `Enable` protocol. Figure 1 gives the timing table for `Enabled`. On every clock cycle, DATA is either transmitted with the ENABLE bit high, or the DATA is unknown/ignored and ENABLE is low. In this example, the sequence A,B,C,D,E,F is transmitted, taking just over a dozen cycles to do so.

In order to help describe our use of protocols, we use a simple notation to give a type to this agreement between producer and consumer. We notate the `Enabled` protocol using a pair of `E` α, indicated an `Enabled` value of type α is sent; and $\bullet$, to indicate that nothing is sent from consumer to producer. We place the outgoing type above the back-edge type, and group with Oxford-style brackets, to notate the use of `Seq` to do the actual transmission.

$$\left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \bullet\end{matrix}\right]\!\!\right] \tag{1}$$

3.2 Handshaking

The basic shortcoming of `Enabled` is the inability of the consumer to slow down the dataflow. This is overcome by handshaking. There are two general types of handshaking: sending an acknowledgment to an `Enabled` datum ("Yes, I got that, please send the next one."), or indicating readiness to receive an `Enabled` datum ("If you send me something, I promise to be ready for it."). In Kansas Lava, we support both, but standardize around the acknowledge, simply because this use of acknowledge complies with the `opencores.org` Wishbone bus protocol, and one standard protocol with necessary coercions is more manageable than supporting both handshake styles everywhere. Figure 2 gives the timing diagram for standardized handshaking, which we called AckBox (acknowledge/mailbox). Again using our invented notation, we can describe this protocol mnemonically.

$$\left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right]\!\!\right] \tag{2}$$

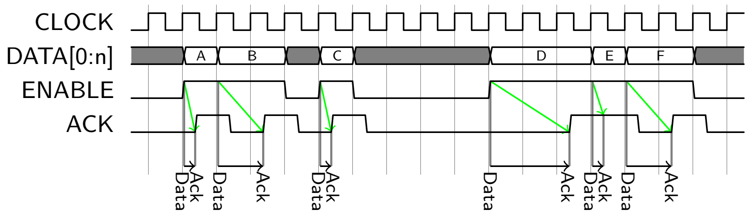

Fig. 2. The Handshake Protocol

Like before, we transmit datums using the `Enable` protocol. This time, however, we have an acknowledgment, `A`, which asynchronously responds to the ENABLE line, and provides actionable evidence of receipt of data. In Kansas Lava, the acknowledgment is typed `Ack`, and `Ack` is always used to indicate successful receipt of data.

The issue with sending acknowledgment to a producer of data is purely logistical. If Lava was based on relations, like Ruby [10], this would be easy. However, we need to instead take an extra output from the consumer, and feed it backwards, as an extra input into the transmitter. This wiring is error-prone and cumbersome to do manually.

4 Patches

A patch is what we call a circuit, or stream processor, between two protocols. There can be communication patches, that act as bridges between protocols, and there can be computational patches, that perform some computation on the input to generate the output. A patch becomes a mid-level unit of expressing computation on an FPGA fabric.

In Kansas Lava, we can represent a patch with a function that takes input from the left hand protocol and the acknowledge from the right hand protocol, and returns the acknowledge to the left hand protocol, and the output for the right hand protocol. A patch is a function from two-tuple to two-tuple. For example, a fifo (a bounded channel in hardware, a pipe in UNIX), could have the type:

```
--                    lhs input * rhs ack     lhs ack * rhs output
fifo :: ... -> (Seq (Enabled a), Seq Ack) -> (Seq Ack, Seq (Enabled a))
```

The connection and direction of each input and output in not as clear as could be here. We found using this tuple convention confusing, and wiring between patches tedious, because wiring is needed in both directions.

We want to find a good type abstraction to clarify the relationship between data-flow direction and protocol used. Further, with this abstraction we hope to build combinators that make the wiring of patches straightforward in practice. Towards this, we choose to represent our patches using the following notation, based on the idea that both input and output is performed on a specific protocol.

$$\texttt{fifo} :: \left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right]\!\!\right]$$

This notates, using a left-to-right dataflow assumption, that a component called `fifo` takes an enabled, handshaken value using the AckBox protocol from a (unrepresented) component, and passes on results, also using the AckBox protocol. The diagram presents the types of values being passed, and how the protocol is used, but says nothing about what `fifo` actually does.

We can capture the cleaner notational style of the protocol based patch description in Haskell, by using a type synonym. Specifically, a `Patch` is defined as

```
type Patch lhs_dat rhs_dat
           lhs_ack rhs_ack = (lhs_dat,rhs_ack) -> (lhs_ack,rhs_dat)
```

Now `Patch` can mirror the $[]\triangleright[]$ notation for patches, by making use of whitespace.

```
fifoP :: (...) => Patch (Seq (Enabled a)) (Seq (Enabled a))
                        (Seq Ack)         (Seq Ack)
```

The first and the third argument line up as a column, to specify the protocol used as input. Likewise for the output, with the second and forth arguments respectively. We use the suffix `P` to denote the use of a patch, as `M` is sometimes used to denote the use of a monad.

Patches can, of course, be used as coercions between different sequence-based protocols. For example, the `Enabled` protocol can be translated into AckBox protocol, assuming frequent enough handshakes, using `latch`:

$$\texttt{latch} :: \left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \bullet\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right]\!\!\right] \qquad (3)$$

Therefore, `latch` would have the following Haskell type:

```
latchP :: (...) => Patch (Seq (Enabled a)) (Seq (Enabled a))
                         ()                (Seq Ack)
```

4.1 Multi-protocol patches

Some patches have multiple input or output protocols. We notate a multi-protocol interface using multiple columns inside our protocol box. A multi-protocol interface consisting of two AckBox protocols that send an α and β respectively, can be notated using:

$$\left[\!\!\left[\begin{matrix}\mathtt{E}\ \alpha & \mathtt{E}\ \beta \\ \mathtt{A} & \mathtt{A}\end{matrix}\right]\!\!\right] \qquad (4)$$

Having multi-protocol patches allows various standard list processing idioms to be captured at the protocol level. For example, zip and unzip are possible:

$$\texttt{zip} :: \left\llbracket\begin{matrix}\mathtt{E}\ \alpha & \mathtt{E}\ \beta \\ \mathtt{A} & \mathtt{A}\end{matrix}\right\rrbracket \triangleright \left\llbracket\begin{matrix}\mathtt{E}\ (\alpha,\beta) \\ \mathtt{A}\end{matrix}\right\rrbracket \tag{5}$$

$$\texttt{unzip} :: \left\llbracket\begin{matrix}\mathtt{E}\ (\alpha,\beta) \\ \mathtt{A}\end{matrix}\right\rrbracket \triangleright \left\llbracket\begin{matrix}\mathtt{E}\ \alpha & \mathtt{E}\ \beta \\ \mathtt{A} & \mathtt{A}\end{matrix}\right\rrbracket \tag{6}$$

When writing multi-protocol interfaces, we use an infix tupling constructor `:>`, and line up the columns to mirror the patch description. We give the type of `zip` here; the `unzip` follows from the patch description in the same way.

```
data a :> b = a :> b

zipP :: (...)
    => Patch (Seq (Enabled a) :> Seq (Enabled b)) (Seq (Enabled (a,b)))
             (Seq Ack         :> Seq Ack)         (Seq Ack)
```

This ability to write multi-protocol patches allows the implementation of data-flow style hardware descriptions, furthering abstractions provided to the Kansas Lava user.

4.2 Patches for Kansas Lava sequences

So far, patches and protocols are a hand-shake of sequences, mapping to signals in VHDL. However, there is a powerful generalization that can be introduced. We want to generalize patches and protocols to other transport mechanisms other than just Kansas Lava sequences. The protocols rendered with Oxford brackets can be written explicitly, using single-line brackets. We therefore define our Oxford brackets in terms of the more primitive notation.

$$\left\llbracket\begin{matrix}\alpha_1 \ldots \alpha_n \\ \beta_1 \ldots \beta_n\end{matrix}\right\rrbracket \equiv \begin{bmatrix}(\mathtt{Seq}\ \alpha_1) & \ldots & (\mathtt{Seq}\ \alpha_n) \\ (\mathtt{Seq}\ \beta_1) & \ldots & (\mathtt{Seq}\ \beta_n)\end{bmatrix} \tag{7}$$

We take a small liberty with the rewriting using rule (7), where • inside Oxford brackets represents `()`, not `Seq ()`. We appeal to the isomorphism between `()` and `Seq ()`, where our use of `()` is unlifted, to justify this syntactical shortcut. We retain, however, the Oxford-style brackets for conciseness.

This generalization allows us to insert and extract Haskell values straight into our patches. We can have a patch that has conventional Haskell values on one side, and uses Kansas Lava `Seq` on the other. For example, we can have patches that coerce to and from Haskell lists, here called `toAckBox` and `fromAckBox`.

$$\texttt{toAckBox} :: \begin{bmatrix}[\mathtt{E}\ \alpha] \\ \bullet\end{bmatrix} \triangleright \left\llbracket\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right\rrbracket \tag{8}$$

$$\texttt{fromAckBox} :: \left\llbracket\begin{matrix}\mathtt{E}\ \alpha \\ \mathtt{A}\end{matrix}\right\rrbracket \triangleright \begin{bmatrix}[\mathtt{E}\ \alpha] \\ \bullet\end{bmatrix} \tag{9}$$

We call these patches “shallow” because they have a shallow embedding; that is, they can never be rendered into hardware because of the direct use of the Haskell lists; they exist for simulation and test-bench use only. The protocol

$$\begin{bmatrix}[\texttt{E}\ \alpha]\\ \bullet\end{bmatrix} \tag{10}$$

is implemented directly as a Haskell list of `Maybe` (and the returned `()`.)

There is no requirement for the left-hand side of a patch to share the same timings as the right-hand side. In the case of `toAckBox` the left-hand side is a Haskell list, the right hand side is a Lava sequence. The use of $[\texttt{E}\ \alpha]$ inside (10), rather than simply a $[\alpha]$ allows `toAckBox` to generate a punctured use of the AckBox protocol; `toAckBox` can literally be told to send `Nothing`.

4.3 Chaining together patches

The principal thing we can do with patches is combine them into bigger patches. We do this using a type-safe bus builder, $$.

$$\$\$ \ :: \ \begin{bmatrix}\alpha\\ \beta\end{bmatrix}\triangleright\begin{bmatrix}\gamma\\ \delta\end{bmatrix} \ \longrightarrow \ \begin{bmatrix}\gamma\\ \delta\end{bmatrix}\triangleright\begin{bmatrix}\pi\\ \phi\end{bmatrix} \ \longrightarrow \ \begin{bmatrix}\alpha\\ \beta\end{bmatrix}\triangleright\begin{bmatrix}\pi\\ \phi\end{bmatrix} \tag{11}$$

Unsurprisingly, $$ is also associative:

$$([]\triangleright_1[] \ \ \$\$ \ \ []\triangleright_2[]) \ \ \$\$ \ \ []\triangleright_3[] \ = \ []\triangleright_1[] \ \ \$\$ \ \ ([]\triangleright_2[] \ \ \$\$ \ \ []\triangleright_3[]) \tag{12}$$

When chaining components together, we sometimes use an infix variant of our signatures, where the $\triangleright$ is replaced by the name of the component, and $$ to notate a bus.

$$\cdots \ \left(\left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \bullet\end{matrix}\right\rrbracket \texttt{latch} \left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \texttt{A}\end{matrix}\right\rrbracket\right) \$\$ \left(\left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \texttt{A}\end{matrix}\right\rrbracket \texttt{fifo} \left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \texttt{A}\end{matrix}\right\rrbracket\right) \ \cdots \tag{13}$$

The bus can also be collapsed, by replacing the $$ and the duplicate protocol description with a single instance of the protocol, thus:

$$\cdots \ \left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \bullet\end{matrix}\right\rrbracket \texttt{latch} \left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \texttt{A}\end{matrix}\right\rrbracket \texttt{fifo} \left\llbracket\begin{matrix}\texttt{E}\ \alpha\\ \texttt{A}\end{matrix}\right\rrbracket \ \cdots \tag{14}$$

This gives us a concise notation to describe a pipeline of cooperating components.

5 Patch logic

At this point of the discourse of protocols and patches, traditional functional programming kicks in. Do we have a unit for our patch, for example? What are the laws for patches? This section introduces patch logic.

First, we have the combinators for lifting into the patch world and for executing a patch, called `output` and `run` respectively, and the law that they form an identity.

$$\texttt{output} :: \alpha \quad\longrightarrow\quad \begin{bmatrix}\bullet\\\bullet\end{bmatrix}\triangleright\begin{bmatrix}\alpha\\\bullet\end{bmatrix} \tag{15}$$

$$\texttt{run} :: \begin{bmatrix}\bullet\\\bullet\end{bmatrix}\triangleright\begin{bmatrix}\alpha\\\bullet\end{bmatrix} \quad\longrightarrow\quad \alpha \tag{16}$$

$$\texttt{run}\circ\texttt{output} = \texttt{id}_\alpha \tag{17}$$

The next primitive is `empty`, which is the identity for patches.

$$\texttt{empty} :: \begin{bmatrix}\alpha\\\beta\end{bmatrix}\triangleright\begin{bmatrix}\alpha\\\beta\end{bmatrix} \tag{18}$$

$$\texttt{empty}\ \$\$\ []\triangleright[] = []\triangleright[] = []\triangleright[]\ \$\$\ \texttt{empty} \tag{19}$$

We have a way of changing the incoming or outgoing component of a protocol. using `forward` and `backward`.

$$\texttt{forward} :: (\alpha\to\beta) \quad\longrightarrow\quad \begin{bmatrix}\alpha\\\gamma\end{bmatrix}\triangleright\begin{bmatrix}\beta\\\gamma\end{bmatrix} \tag{20}$$

$$\texttt{forward}\ f\ \$\$\ \texttt{forward}\ g = \texttt{forward}\ (g\circ f) \tag{21}$$

$$\texttt{forward id} = \texttt{empty} \tag{22}$$

$$\texttt{backward} :: (\beta\to\alpha) \quad\longrightarrow\quad \begin{bmatrix}\gamma\\\alpha\end{bmatrix}\triangleright\begin{bmatrix}\gamma\\\beta\end{bmatrix} \tag{23}$$

$$\texttt{backward}\ f\ \$\$\ \texttt{backward}\ g = \texttt{backward}\ (f\circ g) \tag{24}$$

$$\texttt{backward id} = \texttt{empty} \tag{25}$$

$$\texttt{forward}\ f\ \$\$\ \texttt{backward}\ g = \texttt{backward}\ g\ \$\$\ \texttt{forward}\ f \tag{26}$$

Finally, we have a way of stacking patches.

$$\texttt{stack} :: \begin{array}{c}\begin{bmatrix}\alpha_1\\\beta_1\end{bmatrix}\triangleright\begin{bmatrix}\gamma_1\\\delta_1\end{bmatrix}\\\times\\\begin{bmatrix}\alpha_2\\\beta_2\end{bmatrix}\triangleright\begin{bmatrix}\gamma_2\\\delta_2\end{bmatrix}\end{array} \quad\longrightarrow\quad \begin{bmatrix}\alpha_1 & \alpha_2\\\beta_1 & \beta_2\end{bmatrix}\triangleright\begin{bmatrix}\gamma_1 & \gamma_2\\\delta_1 & \delta_2\end{bmatrix} \tag{27}$$

As a note, `stack` has the property of a form of distributivity between stacks.

$$\begin{array}{c}\texttt{stack}([]\triangleright_1[]\times[]\triangleright_2[])\ \$\$\ \texttt{stack}([]\triangleright_3[]\times[]\triangleright_4[])\\=\\\texttt{stack}\big(([]\triangleright_1[]\ \$\$\ []\triangleright_3[])\times([]\triangleright_2[]\ \$\$\ []\triangleright_4[])\big)\end{array} \tag{28}$$

From these primitives, a number of useful combinators can be constructed. Often, `forward` and `backward` are used together to jointly built the edge of a stack. For example, `open`, which opens a new channel, has the protocols

$$\texttt{open} :: \begin{bmatrix}\alpha\\ \beta\end{bmatrix} \triangleright \begin{bmatrix}\bullet\ \alpha\\ \bullet\ \beta\end{bmatrix} \tag{29}$$

and can be defined thus:

$$\begin{bmatrix}\alpha\\ \beta\end{bmatrix} \texttt{forward}\ (\lambda x \to ()\ \texttt{:>}\ x) \begin{bmatrix}\bullet\ \alpha\\ \beta\end{bmatrix} \texttt{backward}\ (\lambda(()\ \texttt{:>}\ x) \to x) \begin{bmatrix}\bullet\ \alpha\\ \bullet\ \beta\end{bmatrix} \tag{30}$$

From experience, `forward` and `backward` are often used as a pair in this way.

6 Case Study: LCD Driver

As an extended example of a real hardware driver, consider the problem of controlling the LCD panel on the Xilinx Spartan3e FPGA, which is driven by the Sitronix ST7066U Dot Matrix LCD controller. The LCD panel has 16x2 character elements, each of which can display a single ASCII character. Laying aside the more advanced features like user-definable character sets and auto-scroll modes, we want to write a simple memory-mapped driver for this LCD in Kansas Lava.

6.1 Description of the Sitronix ST7066U

Control commands for the LCD are sent to the ST7066U in 9-bit datums. We can represent possible control commands using a Haskell data-structure, with representative constructors shown here, and a table (not given) which maps between these Haskell values, and the relevant 9-bit pattern.

```
data LCD
  = ClearDisplay
  | ReturnHome
  | EntryMode   {moveRight::Bool, displayShift::Bool}
  | SetDisplay  {displayOn::Bool, cursorOn::Bool, blinkingCursor::Bool}
  | FunctionSet {eightBit::Bool, twoLines::Bool, fiveByEleven::Bool}
  | SetDDAddr   {dd_addr::U7}
  | WriteChar   {char::U8}
  | ...
```

The ST7066U itself is programmed on the Spartan3e board via a slow 4-bit data-bus, with a number of control wires, as explained in Figure 3.

Each command is physically transmitted by using two 4-bit nibbles, with a small $1\mu s$ delay between them. The 9th bit (a status-bit) of the command is is sent with *both* nibbles, on the LD_RS line. The gap between 9-bit commands is longer, at least $40\mu s$. Figure 4 explains how complete commands are transmitted. There is also a boot sequence for setting up the ST7066U.

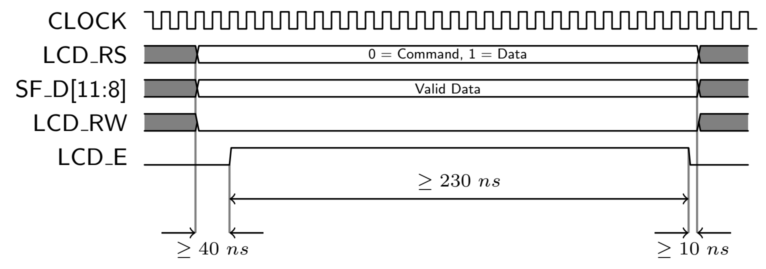

Fig. 3. Timing for the ST7066U 4-bit asynchronous bus (with 50MHz clock)

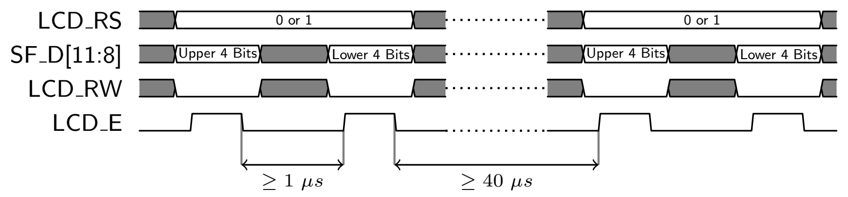

Fig. 4. Sending 9-bit commands to ST7066U over the 4-bit asynchronous bus

- There is a nibble-based boot sequence, with 4 separate nibbles sent, with non-standard delays between them.
- After the nibble boot sequence has been sent, a small sequence of 9-bit LCD commands are required to be sent to reset the device.
- After *most* 9-bit commands, a $40\mu s$ delay is required; with 2 exceptions requiring a $1.40ms$ delay – the joy of hardware interfaces.

6.2 LCD Driver Design

From these specifications, a design based on four separate patches emerges.

- A patch that takes memory writes, and outputs a sequence of LCD commands.
- A patch that prepends the fixed LCD command boot sequence to a stream.
- A patch that takes LCD commands, and outputs nibbles and intra-nibble pause durations. This patch can be responsible for issuing the correct nibble boot sequence.
- Finally, a patch that takes nibbles and intra-nibble pause durations, and drives the bus according to Figure 3, then waits for the given duration before accepting the next nibble, based on Figure 4.

The pipeline of patches we therefore need is:

$$\left[\!\!\left[\begin{matrix}\texttt{E ((X2,X16),U8)}\\ \texttt{A}\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\texttt{E LCD}\\ \texttt{A}\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\texttt{E LCD}\\ \texttt{A}\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\texttt{E (U5,U18)}\\ \texttt{A}\end{matrix}\right]\!\!\right] \triangleright \left[\!\!\left[\begin{matrix}\texttt{(U1,U4,Bool)}\\ \bullet\end{matrix}\right]\!\!\right]$$

The use of patches has informed our design decisions here, and modularized our implementation into reusable components. We now discuss the implementation of these components, in right-to-left order.

6.3 LCD bus driver

The lowest-level patch takes 5-bit nibble-and-status values, a delay before accepting the next nibble, and physically drives the bus, including pausing after each nibble for the prescribed delay.

```
phy_4bit_LCD :: Patch (Seq (Enabled (U5,U18))) (Seq (U1,U4,Bool))
                      (Seq Ack)                ()
```

phy_4bit_LCD is implemented in Kansas Lava using an internal 6-state state-machine and a 20-bit counter, taking approximately 40 lines of code. We use a flip-flop on the output signal to clean up any glitches; perhaps an extension could require this using types in the future.

6.4 LCD instruction compiler

The LCD instruction compiler is the heart of the LCD driver, so we give the complete code here (some whitespace has been removed for space reasons).

```
phy_Inst_4bit_LCD :: Patch (sig (Enabled LCD))  (sig (U1,U4,Bool))
                           (sig Ack)            ()
phy_Inst_4bit_LCD = mapP splitCmd
                 $$ matrixExpandP
                 $$ prependP bootCmds
                 $$ phy_4bit_LCD      -- invokes patch from section 6.3

bootCmds :: Matrix X4 (U5,U18)
bootCmds = matrix [ (3, 205000), (3, 5000), (3, 2000), (2, 2000) ]

splitCmd :: Comb LCD -> Comb (Matrix X2 (U5,U18))
splitCmd cmd = pack $ matrix
        [ pack ( high_op `KL.append` mode, smallGap )
        , pack ( low_op `KL.append` mode, otherGap ) ]
 where
   otherGap = mux ((bitwise) cmd .<=. (0x03 :: Comb U9)) (bigGap,hugeGap)

   (op :: Comb U8, mode :: Comb U1) = unappend ((bitwise) cmd :: Comb U9)
   (low_op :: Comb U4, high_op :: Comb U4) = unappend op

   smallGap = 50     -- between nibbles
   bigGap   = 2000   -- between commands
   hugeGap  = 100000 -- after clear display or return cursor home
```

The compiler is itself built from three patches, and because the only possible client of the nibble instructions is the LCD bus driver, phy_4bit_LCD it is directly invoked by phy_Inst_4bit_LCD. The three patches split the commands into nibbles, intersperse the nibbles in one channel, then prepend the nibble-based boot sequence, as given by bootCmds.

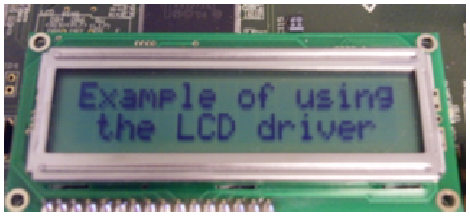

Fig. 5. Example of using the LCD driver

6.5 LCD instruction boot sequence

The `init_LCD` patch prepends the instruction-level boot sequence, which is not magic instructions, but rather the setting of the LCD into a sensible state (no cursor, correct LCD hardware setup, no display shift). Though this is the recommended sequence, it may certainly be possible for a user to use a different setup sequence, so we have `init_LCD` as its own patch. We include the code for `initCmds`, because it demonstrates that we are programming using Haskell structures not bit-level representations at this point.

```
init_LCD :: Patch (sig (Enabled LCD)) (sig (Enabled LCD))
                  (sig Ack)           (sig Ack)
init_LCD = prependP initCmds

initCmds :: Matrix X4 LCDInstruction
initCmds = matrix
    [ FunctionSet { eightBit=False, twoLines=True, fiveByEleven=False }
    , EntryMode { moveRight=True, displayShift=False }
    , SetDisplay { displayOn=True, cursorOn=False, blinkingCursor=False }
    , ClearDisplay ]
```

6.6 Memory-mapped LCD interface

Finally, we have a simple memory mapped interface, that is (2-dimensional) index-value pairs, and an enable. The user literally writes, using this interface, to the LCD. This implementation is almost trivial; for each write command sent to the memory mapped LCD interface, we can simply issue two LCD instructions, one to place the cursor, and the second to place the specified character. There is no reason to optimize the commands sent; the entire display can be rewritten in a fraction of a second. The memory mapped LCD interface has the type:

```
mm_LCD_Inst :: Patch (sig (Enabled ((X2,X16),U8))) (sig (Enabled LCD))
                     (sig Ack)                     (sig Ack)
```

Chaining our sub-components together gives us our working LCD driver, taking character write operations, and returning the bus interactions for our LCD controller. Figure 5 gives an example of using the LCD driver.

7 Discussion

Kansas Lava builds on a long history of hardware description languages, which can be traced back to μFP [13]. There have also been many code generators for VHDL, including JHDL [3], and the combinations of Xilinx Lava to JBits [15].

The use of `Patch` resembles an early feature of Lava: to suggest layout; for example Chapter 16 of [9]. In early versions of Lava, implementations could be configured to work like systolic arrays, with data flowing left to right on every clock cycle, and extra information like carry being passed in the vertical direction. Kansas Lava explicitly chooses to rely on the Xilinx tools for physical layout, based on the improvements made by Xilinx over the last decade. Use of `Patch` is about channels between components, not proximity, though there is no reason why patches could not be extended to give layout hints.

A `Patch` could be represented in Wired [1], a DSL that uses Chalmers Lava. Wired uses a monad called `Let`, and a writable `Var` constructor that allows the acknowledge back-edge to be wired. For example, a fifo could be given the following type:

```
fifo :: (...)
     =>     (Seq (Enabled a), Var (Seq Ack)) -- left hand side
     -> Let (Seq (Enabled a), Var (Seq Ack)) -- right hand side
```

The differences are that in Wired traditional monadic constructions can be used, though `Var` needs to be dynamically checked for single usage at code generation time. In Kansas Lava the back-edge connection can only be connected using juxtapositioning, avoiding the need for this check. In Wired however, the bidirections of a protocol can be expressed inside a single tuple, perhaps allowing for new protocol abstractions.

The `Patch` idiom emerged from frustration when developing a FPGA-centric protocol stack we call λ-bridge. We experimented with different variations of `Patch`, for example originally there were additional control and status signals for each `Patch`. This was later subsumed when needed by simply using extra columns of protocol interactions.

It turns out that `Patch` is a useful place to hide an environment, and we plan to make `Patch` abstract in the future, then experiment with passing in resource hints. It would be useful to know, for example, if a 1-element FIFO will suffice with the necessary bubbles, or if a larger fifo is required.

Overall, we are pleased with how the `Patch` idiom has helped us structure our Kansas Lava programs. Extensions to the simple `Patch` logic, however, would be useful to show specific properties, like compliance to protocol. Patches such as `forward` and `backward` can be used to break the protocol abstraction, in the same way that mapping over an even-only list can destroy the even-only property. Higher level combinators, on top of `forward` and `backward`, will help here.

Acknowledgments

We would like to thank the anonymous referees for their useful comments, Garrin Kimmell who wrote the Haskell datatype in section 6.1, as well as Andrew Farmer, Ed Komp, and the other members of the CSDL lab at KU for their feedback. This work was partially supported by NSF grant CCF-1117569.

References

1. Axelsson, E.: Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction. Ph.D. thesis, Department of Computer Science and Engineering Chalmers University of Technology and University of Gothenburg (2008)
2. Axelsson, E., Claessen, K., Dvai, G., Horvth, Z., Keijzer, K., Lyckegrd, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: MEMOCODE'10. pp. 169–178 (2010)
3. Bellows, P., Hutchings, B.: JHDL - an HDL for reconfigurable systems. Field-Programmable Custom Computing Machines, Annual IEEE Symposium on p. 175 (1998)
4. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. In: International Conference on Functional Programming. pp. 174–184 (1998)
5. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Proc. of Asian Computer Science Conference (ASIAN). Lecture Notes in Computer Science, Springer Verlag (1999)
6. Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium (Sep 2009)
7. Gill, A.: Declarative FPGA circuit synthesis using Kansas Lava. In: The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'11). Las Vegas, Nevada, USA (July 2011)
8. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E.: Types and type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. In: Proceedings of Trends in Functional Programming (May 2010)
9. Hauck, S., DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
10. Jones, G., Sheeran, M.: Circuit design in ruby. In: Staunstrup (ed.) Formal Methods for VLSI Design. Elsevier Science Publications (1990)
11. Matlage, K., Gill, A.: ChalkBoard: Mapping functions to polygons. In: Proceedings of the Symposium on Implementation and Application of Functional Languages (Sep 2009)
12. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
13. Sheeran, M.: mufp, a language for vlsi design. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming. pp. 104–112. ACM, New York, NY, USA (1984)
14. Singh, S.: Designing reconfigurable systems in lava. VLSI Design, International Conference on p. 299 (2004)
15. Singh, S., James-Roxby, P.: Lava and JBits: From hdl to bitstream in seconds. In: FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. pp. 91–100. IEEE Computer Society, Washington, DC, USA (2001)