## Contents

1 The Clash License ............................................. 3

2 Introduction to Clash ........................................ 5
   2.1 What is Clash? ........................................... 5
      2.1.1 Can you just show me? ............................ 6
      2.1.2 Why should I use it? Who should use it? .... 14
      2.1.3 How mature is it? .................................. 15
      2.1.4 What is Clash not good for? ..................... 16
   2.2 Obtaining Clash ......................................... 16
      2.2.1 Ubuntu ............................................. 16
      2.2.2 macOS .............................................. 17
      2.2.3 Windows ............................................ 17
      2.2.4 NixOS and Nix ..................................... 17
      2.2.5 Docker ............................................. 17
      2.2.6 Other ............................................... 17
   2.3 Meta-information: web sites, mailing lists, etc. ......... 18
   2.4 Reporting bugs in Clash .................................. 18
   2.5 Clash version numbering policy ......................... 18

3 Release notes for Clash 1.0.0 ............................... 21
   3.1 Highlights ............................................... 21
   3.2 Full details ............................................. 22
   3.3 Libraries ................................................ 22
      3.3.1 clash-prelude ...................................... 22
      3.3.2 clash-intel ......................................... 23
      3.3.3 clash-lattice ....................................... 23
      3.3.4 clash-xilinx ........................................ 23
   3.4 Known bugs and limitations ............................. 23

4 The Clash Tutorial ............................................ 25

5 The Clash Language ............................................ 27
   5.1 Differences from GHC Haskell ........................... 27
   5.2 The Standard Library (â˘AIPreludeâ˘A`) .................. 27

6 The Clash Toolchain ........................................... 29
   6.1 Using clash and clashi ................................. 29
Welcome to the Clash language manual! Here you will find all the main documentation for the Clash programming language.

Contents:
CHAPTER 1

The Clash License

The Clash toolchain, associated libraries, and documentation are all available under the BSD2 License.

Some of the jokes in this documentation were shamelessly ripped off from beautiful prose for Phabricator, released under the Apache 2.0 License, and maintained by Phacility.

Copyright (c) 2012-2017 University of Twente
2012-2017 The Clash Developers
2017 QBayLogic
2017 Google Inc

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Welcome to the manual for the Clash programming language! The following few sections of this guide will catch you up on what you’re reading and quickly explain the Clash programming language. If you landed here by choice, you might have an idea about some of this, but read on and find out!

2.1 What is Clash?

Clash is a functional hardware description language (or HDL) that closely mirrors the syntax and semantics of the Haskell programming language, and is used for creating hardware designs — most commonly, on modern Field-Programmable Gate Arrays (FPGAs), or Application-Specific Integrated Circuits (ASICs).

Clash is both a compiler, and a set of libraries for circuit design, that transform high level Haskell descriptions of synchronous, sequential logic into low-level VHDL, Verilog, or SystemVerilog. It provides a unique approach to design of sequential circuits, but with a high amount of abstraction power that blurs the line between strict behavioral and structural synthesis approaches.

Clash is:

- **Statically typed**: Clash uses Haskell type system in all its glory — including all its modern extensions and techniques — to bring a high level of type safety and polymorphic programming to hardware design. But it still retains a high degree of type inference: so users can quickly and easily prototype their designs with little effort.

- **Interactive**: unlike traditional older HDLs, Clash comes with a fully interactive Read-Eval-Print-Loop (REPL) that you can use to interactively design, write, and test your circuits.

- **Fast**: Clash uses the Glasgow Haskell Compiler to provide fast executable simulations of hardware circuits.

- **Safe**: Clash uses the type system to enforce global safety invariants in your design. For example, the Clash standard library separates each clock domain by type, and requires type safe clock domain crossing via synchronizers.

- **Efficient**: The compiler uses a whole program synthesis approach in order to view the entire circuit description at once, and optimizes this design aggressively before translating to the chosen HDL. While the overhead isn’t always zero, the compiler is being improved all the time, and careful design can often be used to minimize any overhead.
**Maintainable**: Clash uses the type system to separate sequential from combinational logic. Values that are of type `Signal` exist in the world of sequential circuits, while pure functions are naturally combinational. The type system helps maintain invariants and keep code clean.

**Natural**: Clash uses `Signal` values to represent sequential streams of values over time. Thanks to the lazy semantics that Clash shares with Haskell, lazy `Signal` values give rise to natural, concise feedback loops that youâ€™d see in traditional HDLs between different components. This allows high-level structural design components to be connected quickly and easily. (The usage of `Signal` values will be familiar to those who have experimented with Functional Reactive Programming.)

**Extensible**: If Clash doesnâ€™t offer what you need, thereâ€™s an extensible template language for defining your own HDL primitives in your language of choice. Remap how Clash translates your code into low-level RTL, or add new primitives for your vendor or IP library.

**Low level, but with powerful abstraction tools**: unlike most high level synthesis tools, Clash gives you precise control over register placement and pipelining in your design. Thereâ€™s no inference of resource usage: itâ€™s roughly what youâ€™d get in (behavioral) Verilog. But it offers nearly all of the modularity features of modern Haskell: namespaced modules, pattern matching with sum types, records, real tuples, type-level arithmetic, advanced type system features like DataKinds and TypeApplications, a package manager and set of community-driven tools, and more. This leads to a powerful spectrum of design approaches.

**Modern Haskell**: â€œHasochism in angerâ€. Dependent type features, type level arithmetic and constraint solving, full pattern matching, ghc-mod and indent, Spacemacs or Neovim, cabal or stack – the type classes and libraries youâ€™re familiar with. Nearly all modern GHC Haskell features and tools â€œJust Workâ€œ. (Including some surprising ones, like: the lens library, test frameworks like HSpec, QuickCheck or Hedgehog – or old-school abstractions like the ordinary State monad!)

### 2.1.1 Can you just show me?

Sure! While we wonâ€™t take the time here to explain all these in detail, people always want to know – and this should give you a feeling for what youâ€™re getting into if you have HDL experience already.

Weâ€™ll write 5 circuits: weâ€™ll show the definition up-front, play with it in various ways, and finally give some finer points about the implementation – highlighting Clashâ€™s features.

---

**Warning**: These examples assume some Haskell and FPGA/DSP experience that will not be elaborated upon. Check out the The Clash Tutorial (page 25) for more!

---

#### Toggled Counter

Todo: FIXME

#### Multiply-And-Accumulate

Our next example is a basic multiply-and-accumulate (MAC) circuit, that multiplies and adds its inputs over a series of clock cycles. We separate the MAC design into its combinational (pure) and sequential (stateful) components.

```haskell
-- | A "multiply-and-accumulate circuit" that can be compiled
-- to instantly usable HDL (VHDL, or (System)Verilog).
module MAC
```
There are some important things to note about this snippet:

- **Polymorphic definition, monomorphic circuit**: The type of the underlying transfer and logic functions are polymorphic. We could instantiate the resulting circuit to any synthesizable numeric type we desired – in this case, a 9-bit signed integer. The overall circuit we intend to emit must, however, be monomorphic, with all of its type variables and higher order parameters made concrete.

- **Separation of concerns**: Management of state is separated from circuit logic, in the transfer and circuit functions, respectively. The mealy function ties a state transformation and a pure function of that state into a sequential, synchronous circuit that is synchronized to some clock.

- **Higher-order, stateful circuits**: The mealy function is a completely parametric, higher order sequential circuit transformer – a classic AIJMealy Machine AIFSM – that creates stateful logic out of pure, functional parts. There are also moore and medvedev machines for various other circuit patterns, and all of these are fully reusable.

- **Type safe clocking**: Clash requires that Signal types be annotated with the appropriate clock domains they correspond to. Two signals cannot cross without an explicit synchronizer to ensure type safety. Circuits can also be fully polymorphic over the clock domain, as well.
• **Clocking and reset control**: The MAC example shows how synthesizable hardware must tie the circuit *clock line* and *reset line* — effectively, the *clock domain* of an incoming signal — into the resulting HDL. These lines are controlled by the Clock and Reset types, parameterized over the clock domain. In this case, `withClockReset` establishes that the incoming Clock and Reset lines are used for the mealy machine. (You could alternatively tie the clock lines onto an onboard PLL, or establish a mix of multiple clocks from oscillators and PLLs.

• **Top entities and code generation**: Clash allows you to control the input and output ports of circuits relatively faithfully, making integration simpler. This is controlled by using a *TopEntity annotation* on the desired top-level function, specified with the `ANN` pragma.

The above code can also be synthesized immediately: to compile to Verilog, just run `clash` with `--verilog`:

```
$ clash --verilog MAC.hs
```

The resulting circuit is located in the `./verilog/MAC/mac/mac.v` file, relative to where you ran the compiler. The `mac.v` file contains a `mac` Verilog module, which can be instantiated with a clock signal, a reset signal, two 9-bit signed input ports named `input1` and `input2`, and a resulting 9-bit signed output port named `output`. The clock and reset aren’t tied to anything, so they can come from anywhere else in your design (like a button pin, or a PLL). Here’s what the entity looks like:

```verilog
/* AUTOMATICALLY GENERATED VERILOG-2001 SOURCE CODE.
** GENERATED BY CLASH 1.0.0. DO NOT MODIFY.
*/
module mac
  ( /* Inputs */
    input clk // clock,
    input rst // asynchronous reset: active high
    , input signed [8:0] in1
    , input signed [8:0] in2
    // Outputs
    , output wire signed [8:0] out
  );
```

Feel free to hook this result up to any synthesis tool you desire. This is your first Clash circuit!

**Finite Impulse Response**

Next, we have a **Finite Impulse Response (FIR) filter**, a basic building block of signal processing. It computes the dot product of a vector of coefficient signals, against a *sliding window* over the input signal:

```
import Clash.Prelude

-- The dot product: a summation of products between two vectors
dotp xs ys = sum (zipWith (*) xs ys)

-- The FIR: a dot product of coefficient signals against a sliding window of
-- the input
fir coeffs x = dotp coeffs (window x)
```

This code is not an error: load it into `clashi` and ask for the type of `fir`!

```
$ clashi FIR.hs
*Main> :t fir
```

For simplicity, the cleaned up type is as follows:
There are also some important notes to take away from this example:

- **Parametric in the number of taps**: the FIR definition is invariant in the underlying number of coefficients or \(n\) chosen by the user – you chose the number of the taps, at the call site.

- **Parametric and extensible in the underlying data type**: The FIR can be defined over any data type that satisfies the `Num` constraint and `Default` constraint – including any synthesizable numeric type you wrote.

- **No type annotations, yet completely type safe**: Despite all of the above, the Clash compiler can infer the type of the `fir` function with no help at all! The size of the input coefficient vector determines the size of the sliding window automatically.

And remember (from the MAC example): you can immediately synthesize the above example, because it contains polymorphic type variables!

### Bitonic Sorting

Another classic FPGA example is a **Bitonic sorter** – a parallel sorting network that uses \(O(\log^2(n))\) comparators, with a latency of \(O(\log^2(n))\) as well:

```haskell
module Sort (bmerge, bsort, bsorter) where

import Clash.Prelude

cas :: Ord a => a -> a -> (a, a)
cas x y | x > y = (x, y)
        | otherwise = (y, x)
```

**Todo**: INCOMPLETE

This design is purely combinational, not sequential – but it still shows off some distinctive, advanced features:

- **Advanced type level arithmetic**: The Clash compiler augments the normal type checking capabilities of GHC with a more advanced solver for *type level integers*, including new operations, as well as *limited* automatic discharge of equality proofs and constraints.

  For example, in the above, Clash is able to see that the type of `bmerge` is a function that takes a HOF, from vectors of length \(n\) to \(n\), as well as an input vector of **twice that size**. Types like \(2 \times n\) naturally constrain the given type to double the size. In the event you passed a `Vec` with a type like `Vec (n \times n) a`, Clash would be able to prove they sizes are equivalent automatically. Similarly, a type like `Vec (2 ^ n) a` constrains a vector to a size which is a power of two.

- **Dependent, type-level generic programming**: The `bsorter` code above shows off the power of Clash and Haskell’s type system to **automatically** derive type-safe code that is generic in the \(\text{width}\) of the
sorting network, using a limited form of dependent types.

The `dfold` function performs a **dependent vector fold** using a given base case, induction step, and input vector. The Clash compiler type checks this code, and the type of `dfold` ensures that folding always terminates over the input, and is structurally inductive over the input list. With this knowledge/proof of termination, the Clash compiler is free to unroll this definition for every given concrete set of type variables.

All of this however is wrapped up in a completely generic, reusable type that is easy to understand:

```haskell
-- | Size-generic Bitonic Sorter. Input must be a power-of-two size.
bsorter :: (Ord a, KnownNat k) => Vec (2^k) a -> Vec (2^k) a
```

For example, if we chose the type `k=4`, this would give a bitonic sorter for arbitrary 16-entry vectors. The use of `bsorter` in this case would be equivalent to writing out the following definition of `bsort16`, manually:

```haskell
bsort16 :: Ord a => Vec 16 a -> Vec 16
bsort16 = sort16
where
  sort16 = bsort sort8 merge16
  merge16 = bmerge merge8
  sort8 = bsort sort4 merge8
  merge8 = bmerge merge4
  sort4 = bsort sort2 merge4
  merge4 = bmerge merge2
  sort2 = bsort id merge2
  merge2 = bmerge id
```

**Extreme Overengineering**

Finally, to show you the range of design approaches in Clash, let’s revisit the second example – a simple MAC circuit – and make it an extremely over-engineered, unreadable mess by using a lot of fancy features!

**Warning:** This is a **dark, spooky piece of overdone code** and if you stare at it for too long, your family line will be cursed!

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TemplateHaskell #-}
module Overdrive
  (circuit
   , testBench
   , main
  ) where
-- clash and base imports
```

10 Chapter 2. Introduction to Clash
import Prelude hiding (undefined)
import Clash.Prelude hiding (length, reverse, take, zip)
import Clash.Prelude.Moore (moore)

-- monads
import Data.Functor.Identity (Identity, runIdentity)
import Control.Monad.State
import Control.Monad.Reader

-- lens
import Control.Lens (use, view)
import Control.Lens.Operators ((.=))
import Control.Lens.TH (makeFieldsNoPrefix)

-- testing
import Hedgehog
import Hedgehog.Gen as HH.Gen
import Hedgehog.Range as HH.Range
import Test.Tasty (defaultMain, testGroup)
import Test.Tasty.Hedgehog (testProperty)
import Test.Tasty.HUnit (testCase, assertBool)

--------------------------------------------------------------------------------

-- reader environment: the incoming parameters to the MAC and the accumulator
-- value
data In a = In { _x :: a, _y :: a }
  deriving (Eq, Show, Ord, Bounded)

-- state environment: the current accumulator value. by using 'newtype' we can
-- derive a 'Num' and 'BitPack' instance automatically via
-- GeneralizedNewtypeDeriving. this lets us convert between equally sized types,
-- and have literals like '0 :: Acc (Unsigned 23)'
newtype Acc a = Acc { _acc :: a }
  deriving (Eq, Show, Ord, Bounded, Num, BitPack)

-- monadic representation of a combinational MAC circuit state function. use GND
-- to derive instances automatically
newtype M a r = M { _unM :: ReaderT (In a) (StateT (Acc a) Identity) r }
  deriving ( Functor, Applicative, Monad
            , MonadReader (In a)
            , MonadState (Acc a)
            )

-- generate fancy lenses
makeFieldsNoPrefix ''In
makeFieldsNoPrefix ''Acc

-- run a monadic representation of a combinational MAC state function
runM :: Num a => In a -> Acc a -> M a r -> Acc a
runM inp initial act
  = runIdentity -- unwrap identity
  $ flip execStateT initial -- unwrap state
  $ flip runReaderT inp -- unwrap reader
  $ _unM act -- unwrap action

2.1. What is Clash? 11
translate an 'M' monadic action into a type that is closer to a moore
-- transfer function. this is isomorphic to 'runM', as it simply moves some of
-- the parameters around.
mToTransfer :: Num a => M a r
-- input: monadic action
-- output: transfer function
mToTransfer act st inp = runM inp st act

-- A highly generalized MAC function, parameterized over the fields used by the
-- underlying state and reader monad instances, generated by the lens package
-- NOTE: GHC and Clash can *fully* infer the type of "mac" with no help!
mac' :: forall env st m a.
  ( Num a
    , MonadReader env m
    , HasX env a
    , HasY env a
    , MonadState st m
    , HasAcc st a
  ) => m ()
mac' = do
  in1 <- view x
  -- get the input parameters
  in2 <- view y
  result <- use acc
  -- get current accumulator
  acc .= (result + (in1 * in2))
  -- calculate MAC, update state all at once
  return ()
  -- return nothing

-- specialize the mac function to our chosen monad
mac :: Num a => M a ()
mac = mac'

circuit :: SystemClockReset
  => Signal System (Signed 9)
  -> Signal System (Signed 9)
  -> Signal System (Signed 9)
circuit in1 in2
  = fmap bitCoerce
  $ moore (mToTransfer mac) id 0
  $ fmap (uncurry In)
  $ bundle (in1, in2)

{-# ANN circuit
  (defTop
    { t_name   = "mac"
      , t_inputs = [ PortName "clk"
                    , PortName "rst"
                    , PortName "in1"
                    , PortName "in2"
                    ]
      , t_output = PortName "out"
    } #-)

-- simulated version of the input circuit. the definition is slightly
-- contorted because 'simulate' (normally) only takes a single input signal,
-- but we need two
}
circuit_simulated xs ys = go (zip xs ys)  
  where go = simulate_luxy \ins ->  
    let (in1, in2) = unbundle ins  
    in  circuit in1 in2

-- an ideal model of the circuit’s behavior, one that’s elegant and beautiful,
-- not a horrifying monstrosity

---
circuit_model :: [Signed 9] -> [Signed 9] -> [Signed 9]
circuit_model x y = Prelude.scanl ('a (x', y') -> a + (x' * y')) 0 (zip x y)

---

-- a test bench that can be synthesized
testBench :: Signal System Bool
testBench = done
  where
    -- two input vectors, fed every clock cycle
    in1 = 1 :> 2 :> 3 :> 4 :> Nil
    in2 = 1 :> 2 :> 3 :> 4 :> Nil

    -- the expected result of the circuit for every clock cycle
    out = 0 :> 1 :> 5 :> 14 :> Nil

    -- use clash-prelude to create "test generator signals" out of the inputs
    -- and outputs
    genInput1 = stimuliGenerator in1
    genInput2 = stimuliGenerator in2
    checkOutput = outputVerifier out

    -- generate the resulting signal to check, and tie it together
    -- with a clock line.
    result = checkOutput (circuit genInput1 genInput2)
    done = withClockReset (tbSystemClock (not <$> done)) systemReset result

-- a hedgehog property, stating that the simulated MAC circuit is the same
-- as the original version

prop_model_equiv :: Int ~> Property
prop_model_equiv cycles = property $ do
  -- generate random individual 'Signed 9' values in the appropriate range
  let sr = HH.Range.constantFrom 0 minBound maxBound

  -- generate synthetic input signals, that effectively model the number of
  -- simulated cycles to test for
  let lr = HH.Range.linear 0 cycles

  -- generate two lists representing input signals. the length of this list
  -- effectively determines how many input cycles to sample, meaning that
  -- property shrinking will cause hedgehog to find the "earliest invalid cycle"
  -- that the property is violated.
  xs <= forAll $ HH.Gen.list lr (HH.Gen.integral sr)
  ys <= forAll $ HH.Gen.list lr (HH.Gen.integral sr)

  -- establish the real circuit is equal to the spec, up-to the specified
  -- number of cycles
  let trim = take (min (length xs) (length ys))
  trim (circuit_model xs ys) === trim (circuit_simulated xs ys)
main :: IO ()
main = defaultMain $ testGroup "Multiply-And-Accumulate"
  [ testGroup "Unit Tests"
    [ testCase "Synthesised testBench" $
      assertBool "" $ Prelude.all (== False) (sampleN 4 testBench)
    ]
  , testGroup "Properties"
    [ testProperty "circuit == spec (100 cycles)" (prop_model_equiv 100)
    , testProperty "circuit == spec (1000 cycles)" (prop_model_equiv 1000)
    ]
]

Aside from turning 3 lines of easy-to-read-code into 200 lines of obscure nightmare code (improving our job security) – this example also shows off some important features, most of them unique to Clash:

- **(Almost) No feature loss**: Unlike DSLs or complex staging approaches, Clash is Haskell, and is a pure compiler from (GHC) Haskell to HDL. It supports almost every Haskell feature and tool – like GeneralizedNewtypeDeriving and TemplateHaskell.

- **Clash and Haskell interoperate**: the above example cleanly uses lens and mtl from the installed package set to define the circuit. You can just use Haskell packages!

- **Not a DSL**: Clash isn’t a DSL – it really takes Haskell input, and compiles it to HDL, using a static analysis approach to compilation. There is no object/meta language distinction – there is only one language, with the same set of abstractions you’re used to. In the above example, packages like lens or mtl aren’t just managed with the same tools – they are analyzed and optimized away during the compile stage, just like always!

- **Existing code works**: much existing Haskell code – like the core foundations of lens and mtl – transparently work with Clash with little effort. While the programming model is different, the high level of abstraction in most Haskell code means that most transformations are independent of the underlying semantic data representation or shape of code. So most existing, foundational abstractions – Monoid, Applicative, Bounded types and Enum types, or classic tools like Maybe and Either – work effortlessly in a completely new domain.

- **Synthesis and simulation, in one language**: Much like Verilog or VHDL, Clash can be simulated in software, and synthesized to hardware – this means you can write test benches and surrounding tools for your circuits trivially in Haskell. The Clash compiler won’t attempt to synthesize parts of the code your circuits don’t use – so a single file can contain both hardware and software.

### 2.1.2 Why should I use it? Who should use it?

Clash seems neat, and more importantly, it obviously looks really cool. It even sounds super cool when you tell people you designed a usable RISC-V processor in Haskell. There are already 20 of those, but in Haskell? But still – is it really for you? It might be, if:

- **You are a hardware engineer looking for something new and exciting**: If this is you, you are an hardware engineer looking for something new and exciting, and you hopefully aren’t frightened by very different ways of thinking about the world. For example, maybe you want to try a tool that only takes about 2GB of space to install as opposed to 40GB, or you would like a language that has things like data types, but everyone you’ve ever worked with still hated it anyway.

As far as traditional HDLs and HDL alternatives go, Clash is unique – it retains the mixed simulation/synthesis approach of Verilog/VHDL, and has a real compiler, with a powerful language – allowing powerful abstractions.
In fact, you can nearly transliterate Verilog into Clash, with a little work. But it primarily revolves around the concepts of structural design with a functional flavor.

For experienced hardware engineers, knowledge of design physicality, e.g. the physical consequences of FPGA and hardware design (timing, metastability, clocking, paths, etc) will already be at hand. As Clash is fairly low level and structural, most of the effort will go to learning effective Haskell programming.

The most important takeaway is that, for an experienced hardware engineer, learning to think like a Haskell programmer will be difficult. But by using Clash, you should hopefully end up writing circuits that are safer, efficient, more abstract, and more declarative than you otherwise could.

• You are a Haskell programmer, looking to get into hardware design:

If this is you, you woke up this morning and thought love to learn about hardware design. If you did, that’s fantastic! You can learn a lot about software programming and software design when you learn about hardware. Or maybe you want to learn about the dark, scary world of extremely expensive, proprietary EDA software – it all makes you cringe as a software programmer. Fantastic!

You are also in luck: Clash is effectively real Haskell with all the bells and whistles you used to. It acts like an ordinary Haskell compiler and Haskell toolchain for 99% of all features, in fact. It also works just fine with scary EDA tools. But more importantly, the semantics of Haskell/Clash are close to the semantics of hardware!

If you got some experience thinking about lazy functional programming – the idea of sequential circuit design should come relatively naturally. For familiar Haskell programmers, the Signal type in Clash feels reminiscent of a Functional Reactive Programming API. This lets you focus on the most important ideas you must master: physicality of the design, i.e. the physical characteristics and consequences of FPGA and hardware design.

Using Clash will help free you from some of the irrelevant details like traditional, annoying HDL semantics (which only occasionally coincide with hardware), and unfamiliar toolchains so you can focus on the truly important lessons of hardware design and implementation, not the annoying details.

• You are new to Haskell, new to hardware design, and looking to learn:

This might be tough, but we believe in you! If you join the community we’ll help you out. And you can help us: suggest stuff that we can improve or editorialize - here, elsewhere, everywhere.

• You a Haskell programmer and a hardware engineer looking for a new tool:

Hello? Who are you? Can you help us work on Clash?

2.1.3 How mature is it?

Clash has had raving reviews from its users, including works and better than Verilog – but it’s still evolving relatively frequently, despite being several years old. We should say it now: circuits you write now will likely be broken by future versions of the Clash compiler in big ways! You can always stick to older compilers, but the language and toolchain is still evolving (TODO: fixme: pvp ref?)

Furthermore, Clash still undeniably has rough edges (see the troubleshooting page (page 37)) and some bugs (see the known bugs and limitations (page 21)). We hope to polish these off in the future as time allows – we understand prickly little issues are big annoyances for people.

Hopefully that doesn’t dissuade you: several companies and individuals have been using Clash successfully for real work, ranging from tiny 1k-LUT designs on $20 FPGAs, to extremely modern FPGA fabrics and large-scale designs, to ASICs, and more. (And some of these projects wouldn’t have been nearly and easy as they were, without Clash!)

Note: Some companies that use Clash include the following:
• QBayLogic
• Myrtle Software

Some companies that do not use Clash include the following:
• Enron Corporation
• Red Bull GmbH

If you’re already a functional programmer who’s looking to learn hardware, we think Clash fits in a unique
space among the other functional HDLs available (low level, synthesis-driven, with unique abstractions), and should
be considered stable for most use cases. If you’re willing to give it a shot, you might like it!
(And for a Haskell programmer who wants to develop hardware, we think Clash is the most effective HDL choice
there is, at least!)

2.1.4 What is Clash not good for?

TODO FIXME: mention clockless logic (e.g. greenarrays chips)? HLS advantages?

2.2 Obtaining Clash

Note: You must use cabal-install to install stable versions of Clash at this time. Stack LTS/nightly snapshots
are not yet available.

The recommended way for users to currently install Clash is using cabal-install 2.0 or later with GHC 8.2.1
The basic installation only requires cabal-install and GHC, and installation of Clash from Hackage. After
you’re finished, you’ll have clash and clashi executables available in your cabal binary directory.

2.2.1 Ubuntu

Ubuntu is one of the easiest platforms to get started with. We’ll grab GHC 8.2.1 and Cabal from the popular hvr
ppa archives.

```
$ sudo apt-add-repository -y ppa:hvr/ghc
$ sudo apt update
$ sudo apt install -y ghc-8.2.1 cabal-install-2.0
```

GHC and Cabal have been installed under /opt/ghc and /opt/cabal, respectively. Next, add these to your
$PATH – here it will be temporary, but you could also add this to your .profile or .bashrc to make the change
permanent:

```
$ export PATH=/opt/ghc/8.2.1/bin:/opt/cabal/2.0/bin:$PATH
```

Finally, you’re ready to install Clash. This step might take a while, so be ready to grab some coffee or tea:

```
$ cabal update
$ cabal install clash-ghc
```
The resulting `clash` and `clashi` binaries are now installed (by default) under `$HOME/.cabal/bin`. Add these to your `$PATH` just like before, and you're ready to go:

```
$ export PATH=$HOME/.cabal/bin:$PATH
$ clashi
CLaSHi, version 1.0.0 (using clash-lib, version 1.0.0):
http://www.clash-lang.org/ :? for help
Clash.Prelude>
```

### 2.2.2 macOS

**Todo:** FIXME

### 2.2.3 Windows

**Todo:** FIXME

### 2.2.4 NixOS and Nix

NixOS/Nix users can obtain Clash from the latest version of the `nixpkgs` package set. Clash expressions for Hackage packages should already be available upstream. The three packages you need are:

- `haskellPackages.clash-prelude`
- `haskellPackages.clash-lib`
- `haskellPackages.clash-ghc`

Note that because Clash only works with GHC version 8.2.1, you'll probably need to use the `ghcWithPackages` attribute of the `ghc821` compiler set, in order to get the right packages. Otherwise, you may install older versions of Clash!

### 2.2.5 Docker

**Todo:** FIXME, the docker builds were wiped. We might be able to use Nix to publish images continuously.

### 2.2.6 Other

In general, the instructions for Ubuntu or any of the above platforms apply: the most important thing is **you must have GHC 8.2.1 exactly, and cabal-install 2.0 at minimum!**

Once you have `cabal` and `ghc` ready, just install `clash-ghc` from Hackage, and the binaries will be inside `$HOME/.cabal/bin`:

```
2.2. Obtaining Clash
```
2.3 Meta-information: web sites, mailing lists, etc.

On the World-Wide Web, there are several places of interest for Clash hackers:

- Clash home page
- Clash GitHub organization

There is also a mailing list for Clash:

**clash-language** This is a list for Clash users to chat among themselves, and for development announcements. If you have a specific question about Clash, please see the FAQ (page 31), the Troubleshooting guide (page 37), and the known bugs and limitations (page 21) first.

The list is hosted by Google Groups at the following location: http://groups.google.com/group/clash-language

There are also several IRC channels that may be of interest to you:

**#clash-lang on Freenode** This channel is for Clash users, and (mostly) for Clash development discussion.

**#haskell-embedded on Freenode** This is a general channel for using Haskell/GHC on embedded devices. While not Clash-specific, many adjacent discussions and similar developers may be here.

2.4 Reporting bugs in Clash

Clash is constantly evolving (often in big ways, still) so reporting bugs is vital, and we really appreciate you doing it! Clash uses GitHub to host its code and issue tracking, so if you find a bug, please report it to the clash-lang organization.

- If the bug/feature request affects the Prelude, please file the bug in the clash-prelude repository.
- If the bug is otherwise a deficiency in the compiler, please file it in the clash-compiler repository.

2.5 Clash version numbering policy

Clash follows the Haskell PVP Specification for its version numbers, for all packages. (The Haskell PVP is something of an equivalent to Semver in other communities.)

Compiler and prelude libraries maintain the same major and supermajor number, although they may have disjoint bugfix releases (with appropriate constraints applied, depending on the nature of the fix).

**Note:** As a slight exception to this rule, due to the nature of Clash’s tight integration with GHC, updates to the GHC version that Clash uses, even minor ones, often result in major upgrades to the Clash version. As GHC’s internals change frequently, even for minor bumps, it cannot be guaranteed that these changes will not result in Clash changes (for example, in clash-ghc).
For example, Clash 1.0.0 might be compatible with GHC 8.2.1. But if Clash was then updated to use GHC 8.2.2, the next release would likely be Clash 1.1.0 – even if there were no intervening changes otherwise.

IP/Vendor package, such as `clash-xilinx`, also follow the PVP specification, but do not use the same supermajor/major number as Clash itself: they are otherwise external packages, simply maintained by the Clash Developers.

It's recommended (but not required) that downstream Clash packages, and published Clash code, follow the PVP specification.
The significant changes to core parts of the compiler and libraries are listed in the following sections. There have also been numerous bug fixes and performance improvements over the 0.7 branch.

**Note:** Clash 1.0 requires GHC 8.2.1 exactly, and, consequently, Cabal 2.0 at minimum.

### 3.1 Highlights

Clash has seen several major updates for the 1.0 release, the most significant improvements being:

- **New, explicit clock lining and reset control**, along with a simpler interface based on *implicit parameters*. As a result of this, features like DDR and PLL primitives can now be defined almost entirely using Clash itself (and some basic primitives).

  **Warning:** This effectively breaks all known Clash programs, as the `Signal` type has changed and has a new parameter!

- **A new namespace for `clash-prelude`**. Previously, it was under the name `CLaSH`, but is now under the name `Clash`! Your fingers and auto-completion engine will thank you.

  **Warning:** This effectively breaks all known Clash programs! But you can fix it with `sed`.

- **Support for multiple `TopEntity` annotations in a single project**. Previously, the Clash compiler only allowed you to have a single `topEntity` function within your set of modules, and would only generate a single HDL module per invocation. Clash can now handle multiple `TopEntity` functions – in any number of modules – and generate appropriate HDL modules that interface with each other.
• The Xilinx and Intel PLL/clocking interfaces have moved to separate packages outside of \texttt{clash-\texttt{prelude}}, inside the new \texttt{clash-xilinx} and \texttt{clash-intel} packages.

• An improved optimizer has landed, that does better constant folding, is more aggressive in eliminating indirection, and generates more readable HDL output.

• The compile-time performance of the RTL synthesis has improved, in some cases exponentially (with respect to the input program), thanks to some reports, tuning, new algorithms, and a lot of analysis. At the same time, the compiler also more cleanly backs off and does not \texttt{generate} large programs from small ones as much. Overall, Clash should compile your programs faster and generate RTL quicker than before.

• Code cleanup and consolidation in the compiler, with more to come over time.

### 3.2 Full details

\textbf{Todo:} Lorem ipsumâ€”

• The \texttt{clash\{-vhdl,verilog,systemverilog\}} packages no longer exist, and have been folded into \texttt{clash-lib} directly under the same module names. They have proven stable enough and do not require separate external packages.

• Several fixes to the Verilog and SystemVerilog backends concerning \texttt{Integer} truncation and out-right compilation errors have been fixed. Please see Clash issues \#219, \#220, and \#222.

### 3.3 Libraries

Changes to core libraries maintained by the Clash developers are listed below.

#### 3.3.1 clash-\texttt{prelude}

\textbf{Todo:} Lorem ipsumâ€”

• The \texttt{clash-prelude} namespace now starts with \texttt{Clash}, not \texttt{CLaSH}! (issue \#125)

\textbf{Warning:} This effectively breaks all known Clash programs! But you can fix it with \texttt{sed}.

• Added the \texttt{plusToLe} and \texttt{plusToLeKN} functions to \texttt{Clash.Promoted.Nat}.

• The superclass constraint for the \texttt{Foldable} instance of \texttt{Clash.Sized.Vector.Vec} has been improved: previously it required an awkward type equality constraint \((m \sim (n+1))\), whereas now it requires a simple inequality constraint \((1 \leq n)\).

• Added fixities for all infix-versions of \texttt{SNat} related operators, under \texttt{Clash.Promoted.Nat} (issue \#113).

• Added \texttt{BitPack} instances for \texttt{GHC.Generics} product types (issue \#112).

• Added a \texttt{leToPlus} function to \texttt{Clash.Promoted.Nat} (issue \#111).

• Added a bounds check to \texttt{Clash.Sized.Index.maxBound} internally. (issue \#89).
3.3.2 clash-intel

This is a brand new library, providing PLL and DDR primitives for Intel (formerly Altera) devices. These primitives are available under the Clash.Intel namespace.

3.3.3 clash-lattice

This is a brand new library, providing PLL primitives for Lattice Semiconductor iCE40 devices. These primitives are available under the Clash.Lattice namespace.

3.3.4 clash-xilinx

This is a brand new library, providing PLL and DDR primitives for Xilinx Series-7 (and later) devices. These primitives are available under the Clash.Xilinx namespace.

3.4 Known bugs and limitations

- Clash occasionally has bad compilation and synthesis complexity (in both time and space) on certain inputs. See clash-compiler issue #240 and clash-compiler issue #251 for both big and small examples. This is sometimes possible to work around (via rewriting or compiler option magic), and other times is not. If you suspect your circuit has unreasonably high memory usage or synthesis time, please file a bug with reproducible instructions and a minimum viable sample so we can help.

  **Note:** This only affects RTL synthesis; compilation of simulations to native executable code is still quite fast and efficient.

- Asynchronous and synchronous resets are globally positive in the current design of clash-prelude. While reset polarity polymorphism for the Signal type (allowing async/sync positive and negative resets, and mixing them) is possible, it currently makes the API more complex.

  As a workaround, users can redefine primitive .json mappings for their designs, and remap clash-prelude functions (e.g. write a new mapping for Clash.Signal.register that uses negative resets and use this as necessary). (TODO FIXME: ref link) Currently clash-prelude offers no alternative primitive mappings for negative resets.

  It is unclear in the future whether or not, and how, this restriction may be lifted. (Features like Backpack may, in future GHC/Clash releases, make this possible.)

- Clash currently does not support inout parameters for compiled RTL code in any way, for any of its backends. In the future, this limitation may be lifted to some degree. See clash-compiler issue #239 for more.

- Clash does not allow pattern matching on structurally recursive GADT types, that would otherwise provide a type-driven proof of terminating recursion. As a result, you cannot pattern match on any GADTs.

  In the near future, we plan on lifting this restriction for the built-in Cons data type. In a farther future, we plan on lifting this restriction for all appropriately defined GADTs. This requires a new core language and synthesis analysis pass.

  See clash-compiler issue #170 for more.

- Clash needlessly recompiles any module that uses a compiler plugin, regardless if it needs to be recompiled or not. This affects all compiler plugins and their users, but especially Clash users, as Clash comes equipped
with several compiler plugins for type checking. For large Clash codebases, this often slows recompilation of large builds, as a build that would otherwise be a no op demands many needless recompilations.

This bug is due to an upstream GHC limitation – see GHC issue #7414 for more information. We hope to fix this in a future GHC release.

There is currently no workaround for this bug.

• Clash occasionally has unnecessary overhead in the resulting circuits it generates. While Clash is normally quite low-level and space efficient, has an aggressive whole program synthesizer, and generated circuits are often small – the compiler currently does not remove all forms of compile-time overhead, as of right now.

Working around this often requires deep knowledge of the Haskell toolchain and the input program. If you suspect the Clash compiler is generating needlessly inefficient circuits, please file a bug so we can reproduce it and help.

In the future, we plan to tackle this with more aggressive optimizations (e.g. better constant propagation) and, inevitably, techniques like Partial Evaluation.

• TODO FIXME: mention lack of register retiming passes, and FAQ entry, issue #165.
• TODO FIXME: mention that compiled clash code (e.g. a library from hackage) cannot use bang patterns
CHAPTER 4

The Clash Tutorial

Lorem ipsum dolor sit amet, consectetur adipiscing elit.
The next few sections document aspects of Clash the programming language, including differences from Haskell and the standard set of blessed libraries.

5.1 Differences from GHC Haskell

Todo: Lorem ipsum

5.2 The Standard Library (â€œPreludeâ€)

Todo: Lorem ipsum
The Clash Toolchain

Todo: Lorem ipsum

6.1 Using clash and clashi

6.1.1 Simulation mode

6.1.2 RTL synthesis mode

6.2 Cosimulation and compilation

6.3 External RTL

6.3.1 Blackboxes

6.3.2 Primitives

6.4 Compiler flags
Frequently Asked Questions & Answers

• Q: How do I install Clash?
  A: Check out the *Obtaining Clash* (page 5) section of the manual, in the introduction, for information on how to reliably install the latest version of the Clash compiler.

• Q: Is the name â˘AIJClaSHâ˘I, â˘AIJCLaSHâ˘I, or â˘AIJClaSHâ˘I?
  A: The original name â˘AIJClashâ˘I comes from the acronym â˘AIJCLaSHâ˘I (prounounced: â˘AIJclashâ˘I), the CAES Language for Synchronous Hardware – developed and maintained by the lead developer, Christiaan Baaij, at the University of Twente.

  The acronym â˘AIJCLaSHâ˘I was originally used pervasively in the libraries and source code originally, but since the 1.0 release has moved more towards the simple name â˘AIJClashâ˘I for user-facing code and documentation. (Itâ˘A ´Zs much easier to type, at least!)

  The stylization â˘AIJClaSHâ˘I is an homage to the Haskell programming language, whose official logo has long been the venerable Greek lambda character, and graphically the name â˘AIJCLaSHâ˘I has often been stylized this way.

• Q: Is Clash its own programming language? Or is it â˘AIJHaskellâ˘I? It uses GHC?
  A: The answers to these questions are, in order:

  1. Yes: Clash is a *hardware design language* (HDL) for describing synchronous hardware designs. The Clash programming language is composed of two components: *compiler* and a *standard library* (the â˘AIJpreludeâ˘I) for circuit design.

  2. Yes: Clash is also Haskell. It would be more accurate to call it â˘AIJa Haskellâ˘I, in a sense, as a member of the â˘AIJHaskell family of programming languagesâ˘I.

     Broadly speaking, Clash views its â˘AIJsource languageâ˘I, its high level input, in terms of Haskell: the Clash compiler takes Haskell input and converts it into a circuit. This uses the same syntax and type checking rules as any ordinary Haskell implementation.
However, Clash programs have different semantics, ones that match the domain of the hardware world. The key, binding insight is that HDL semantics, Haskell semantics, and the semantics of the functional language called Clash are approximately the same thing. (A result that has been known in its broad forms since the 1970s.)

So it makes sense to think of Clash as ÒAIJa HaskellÁÍ, although it has different semantics and a different compilation pipeline. But the end result is a language that looks like Haskell, talks like Haskell, and quacks like Haskell – but is also suitable for large-scale hardware design.

3. Yes: Clash uses the Glasgow Haskell Compiler, which offers an API, in order to transform the internal GHC view of Haskell programs, called GHC Core, into ClashÁŽs internal representation, Clash Core. By doing this, Clash uses GHC in order to do all of the hard work of compiling Haskell: analysis, type checking, feature detection, parsing, and more. Clash instead only deals with a simpler internal representation and compiles that, making it easier and more robust to develop.

Utilizing the Glasgow Haskell Compiler also comes with other notable advantages: it allows the Clash frontend to transparently interoperate with GHC; nearly 100% of ÒAIJmodernÁÍ Haskell syntax and type-system features are available; a large amount of existing Haskell code can be shared and leveraged; ordinary tools and packaging systems are already available. GHC is also the secret behind executable simulation of Clash programs, by using GHCÁŽs powerful native-code compiler to emit executables to simulate a circuit.

• Q: Clash has more powerful type level arithmetic than GHC. Can I get GHC and ghci to have the same functionality?

A: Yes! ClashÁŽs enhanced type checking functionality is due to extensible compiler plugins that can be used in any Haskell codebase, not just Clash code.

To enable these plugins in your code, just put the following pragmas in your modules at the very top. Clash and GHCi should then type check your code in the same way (modulo any language extensions that may need to be enabled):

```
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Normalise #-}
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Extra.Solver #-}
{-# OPTIONS_GHC -fplugin GHC.TypeLits.KnownNat.Solver #-}
```

These plugins come from the ghc-typelits-natnormalise, ghc-typelits-extra, and ghc-typelits-knownnat plugins respectively, which are all available from Hackage and Stackage.

Todo: Move this to troubleshooting (page 37)?

• Q: Is Clash a ÒAIJhigh level synthesisÁÍ tool?

A: This is a bit complicated to answer depending on your view of things, but for the purposes of this question, the answer is no. In fact, Clash is actually fairly low level in comparison to most HDLs, in some respects.

For the purposes of the argument, ÒAIJhigh level synthesisÁÍ tools sit above what we would call behavioral synthesis, which, in turn, sits on top of structural synthesis (or RTL synthesis). ÒAIJBehavioral synthesisÁÍ is a synthesis model in which EDA tools interpret behavioral specifications of a circuit, such as ÒAIJIncrement this number on every rising clockÁÍ and create digital hardware designs – structural designs – that implement that behavior exactly, using hardware resources available on a device. For example, ÒAIJUse a D Flip-Flop with a clock enable line to increment this counter, with a feedback loop directing the DFF output back to its inputÁÍ.
These days, most hardware designers operate at the behavioral level of design where possible, and let tools infer exact resource usage. This is easier to do for larger and more complex designs, and truly helps keep the designs understandable.

Behavioral synthesis, then, takes a high level description, where resource usage is not explicit, and turns it into a structural one: where physical device resources are explicit. High level synthesis is then a single step above this model.

The traditional flow of a high level synthesis tool is to normally take a feature-reduced, cut-down variant of a language like C, and infer the behavioral model of the input C code. This behavioral model is then turned into Verilog or VHDL code, and passed onto the ordinary synthesis tools.

Note: The wording is a bit confusing since high level synthesis is also sometimes referred to as behavioral synthesis, but in traditional parlance, when people say high level synthesis, they almost always mean things like C/C++ based HDL compilers. Some of this could have been avoided if we just called it Behavioral synthesis and C-based synthesis instead.

Clash is not high level, in this sense: it does not take a reduced feature set language where the semantics do not match the domain. It supports nearly all of the ordinary Haskell programming language, and retains its semantics, yet at the same time these semantics also closely match those of structural hardware languages.

Clash, then, in a sense is a structural HDL, not a behavioral one, and by this definition is lower level than most alternative HDLs. Clash is a language where clocking, routing, register usage, pipelining and any kind of IP interfacing is fairly explicit. (In fact, by abandoning all pretense of convenient abstraction – you can write nearly direct, structural HDL with manual clock routing between components, with little overhead!)

But this also doesn’t mean Clash is bad at abstraction: indeed, thanks to its Haskell heritage, and powerful abstraction capabilities, the structural approach to circuit design in Clash requires rethinking how you approach behavioral design in the first place. Using powerful tools like mealy machines, combined with regular Haskell abstractions (such as the State Monad), can approximate extremely high level behavioral descriptions, while retaining structural levels of control. Simple techniques – like using higher order functions – can abstract stateful and sequential circuit components from being tied to underlying structural representations, like BlockRAM vs distributed RAM, or particular vendor IPs.

As a result, Clash is both simultaneously lower level, while offering a higher abstraction ceiling, than most competing HDLs – that are either traditional or modern.

Note: This is made more interesting by the fact that while Clash itself is structural rather than behavioral in its semantics, it emits HDL that, in many ways, is behavioral! For example, while you instantiate a register or blockRam manually in Clash, which feels structural – the underlying HDL often relies on the synthesizer’s behavioral inference to e.g. infer usage of a BlockRAM or D-Flip-Flop.

This gives a nice spectrum of trade offs, where designs feel structural, with powerful levels of control and good abstractions – but actual RTL results are behavioral, and carefully generated to allow the synthesizer to correctly and efficiently utilize device resources, as needed. This is important e.g. for writing external IP in Clash, which will fit into some unknown design, and where the synthesizer can likely choose structural device logic better than you can. Clash users can even overload the generation of RTL primitives if they want to really emit structural code, if they are daring.

• Q: Does Clash work with my EDA tools?

A: We hope so! In general, Clash should work very well for the big FPGA vendors and all their EDA tools (Xilinx and Intel) – this is what most of the testing and real world deployments use. But Clash has also been successfully used on Microsemi (formerly Actel) SmartFusion 2 FPGAs, as well as Lattice...
Semiconductor iCE40 FPGAs, and the developers maintain basic IP libraries for most of these toolchains. (These 4 vendors make up approximately 95% of the market in its entirety.)

In general, Clash should support your toolchain just fine, and the default `clash-prelude` primitives should work effortlessly, provided it supports certain aspects of behavioral inference (e.g. inferring BlockRAMs). If it doesn’t, you can often call out manually to your vendor’s technology library, or use a tool like Yosys to do technology mapping for you.

**Q**: Does Clash support Project IceStorm?

**A**: Yes! Clash’s Verilog backend emits clean Verilog 2001, which is supported by Yosys – and can thus be placed and packed with `arachne-pnr` and `icestorm`. So you can immediately start using it with one of the only truly open source FPGA Flows.

Due to the low cost of iCE40 FPGAs and the freely available toolchain, we expect Clash to support Project IceStorm (via Verilog) for the foreseeable future.

**Q**: What’s the difference between Clash and â˘AJLavaâ˘AI?

**A**: TODO FIXME: history chalmers->york->xilinx->kansas, DSL vs synthesis differences

**Q**: What’s the difference between Clash and Bluespec Verilog?

**A**: TODO FIXME: guarded atomic actions, history, etc

**Q**: What’s the difference between Clash and Chisel/Spinal, or Hardcaml?

**A**: The most obvious difference between these two toolchains is that Clash exists as a Haskell derivative, with a full synthesizing compiler to RTL – while Chisel exists as an embedding of hardware semantics inside Scala. The Chisel compiler does not synthesize RTL from Scala – it synthesizes RTL from an embedded DSL, constructed by a Scala program at runtime.

Aside from the â˘AJhost languageâ˘AI differences, this means that Chisel is conceptually closer to something like Kansas Lava than Clash – and this difference manifests in most of the same ways (other design points, aside).

Another fair point worth mentioning is that while Clash and Chisel have both been around for numerous years, Chisel has quite a lot more infrastructure and has public, taped out production cores (in the form of e.g. Rocket and BOOM). Chisel also has accompanying add-on tools, such as Spatial, which allow the clean co-development of hardware and software, all within Scala.

Similarly, Hardcaml is an embedded DSL for RTL semantics, using OCaml as the host language.

**Q**: Can Clash be used for ASIC designs, as well as FPGA designs?

**A**: Yes, but maybe not out of the box! The RTL produced by the Clash compiler is largely vendor agnostic. However, ASIC tool flows will vary from foundry to foundry and different fabrication processes – certain tools may not support e.g. behavioral inference in the same way FPGA tools do. Furthermore, all of the flows for these processes are very proprietary – and they require you pass through a dangerous gauntlet, complete with a dramatic ritual, in order to obtain them. The nature and location of these gauntlets and rituals are a closely guarded secret. This makes testing particular ASIC flows relatively difficult for the developers.
As a concrete example of this, Clash supports BlockRAMs in FPGA designs – and while most modern FPGA EDA tools will infer BlockRAMs from e.g. initial blocks in Verilog, ASIC tools may not – this means that certain modules like Clash.Prelude.BlockRam may not work out of the box with your ASIC flow, as they expect behavioral inference to work. Or it might work – we can’t easily know!

For this particular case, either using an external RTL primitive, to call out to a technology library for your ASIC toolchain (TODO FIXME: link), or remapping the Clash.Prelude.BlockRam module to a new set of primitives, would fix this.

If you’re attempting to do an ASIC design with Clash and need help supporting your particular toolchain, or find bugs, please contact the Clash Developers (TODO FIXME: link).

• Q: Do I need to know Haskell in order to use Clash?

A: This is a complicated subject, but it is strongly advised that you know a bit of Haskell before approaching Clash. This isn’t necessarily mandatory. However, in practice, Clash is a complex tool that is deeply integrated into the Haskell ecosystem and toolchain, and uses advanced Haskell language features to perform some of its more unique tricks. Advanced designs tend to blend both hardware RTL and Haskell software libraries for powerful code sharing, build systems, etc.

If you’re scared, don’t be! You can always get help – but you should temper your expectations if you started writing Haskell yesterday. In general, if you’re reasonably comfortable with Haskell build tools, and can work around and work with the type checker, you should be good to go, if you’re persistent. While advanced designs and the Prelude tend to use some highly powerful features, users can often get away without worrying about them.

However, on the flip side of this, Clash’s unique semantics and design (such as lazily-modeled feedback loops, undefined values, etc) makes the semantic framework behind RTL and hardware design rather approachable to a seasoned Haskell programmer. Sequential vs combinational logic is easily identified by the type (stateful vs pure), behavioral logic is often easily obtainable with tools like State monads, and applicative Signal types closely resemble structural, FRP-inspired programming models. The semantics of lambda calculus is, in many ways, relatively close to RTL semantics.

But while Clash and Haskell give a good conceptual model for hardware design, they cannot (alone) teach you about critical, physical techniques: such as timing constraints, power analysis, critical paths, verification, floor-planning, understanding physical flip-flop/LUT design, and more. These are just as important as the semantic RTL model itself, and will be the difference between your design working – or not at all.

The most difficult part won’t be knowing Haskell – it will be learning about Hardware!

• Q: Do I need to know existing RTL/HDL languages in order to use Clash?

A: Lorem ipsum

• Q: Is Clash production ready? Can I use it today?

A: Yes! Several companies and individuals have been using Clash successfully on real world designs, ranging from 100s of LUTs to large 200k+ LUT designs, on modern FPGA fabric (UltraScale and Arria 10).

However, Clash is constantly evolving, so be prepared for some bumps! It’s possible you’ll run into compiler or language deficiencies, or you need help for your toolchain, just outright bugs, or a number of things. (While we think Clash is great, we also want to be honest!)

If you’re prepared to make the jump, just be sure to join the Clash community in case you need to ask for help – or help someone out! (TODO FIXME: ref link)
Troubleshooting

Some errors that occur when using Clash are fairly frequent when getting started, and are cataloged here.

**Type error:** Couldn’t match expected type `Signal (a, b)` with actual type `(Signal a, Signal b)`

Signals of product types and product types (to which tuples belong) of signals are **isomorphic** due to synchronisity principle, but are not (structurally) equal. Use the `bundle` function to convert from a product type to the signal type. So if your code which gives the error looks like:

```
z = f a b (c, d)
```

add the `bundle` function like so:

```
z = f a b (bundle (c, d))
```

Product types supported by `bundle` are:

- All tuple types, up-to and including 8-tuples.

---

**Clash.Normalize.Transformations(155): InlineNonRep:** `<FUNCTION>` already inlined 100 times in:`<FUNCTION>`, `<TYPE>`

You left a `TopEntity` in your design with a polymorphic or higher order type. Use `:t <name>` inside `clashi` in order to check if the type is truly polymorphic and/or higher order. If it is, add a type signature, and supply all the higher order arguments.
CHAPTER 9

References, tools, and links

The following lists include useful tools, papers, and 3rd party material for Clash developers and users.

9.1 Useful tools

- yosys: TODO FIXME
- iverilog: TODO FIXME
- GHDL: TODO FIXME
- verilator: TODO FIXME
- arachne-pnr, icestorm: TODO FIXME

9.2 Papers


CHAPTER 10

The Clash Lexicon

Todo: Write a novel containing all the EDA/FP acronyms and synonyms you could ever want because nothing is ever actually simple.
Occasionally, brave souls must venture into the ancestral homeland of Clash in order to understand its inner workings, and its true nature. The location of this homeland is unchanging but cannot be mentioned here, and is only known to be connected in a spooky, mystical way to homeland of GHC.

This page is kept by a dedicated scribe who catalogs these adventures, so that others may learn.
If you are reading this, you are a lucky hacker who is looking to work on Clash. Thanks for your effort! Clash is a large Haskell project and may not always be the easiest to grapple with – hopefully this guide will get you up to speed quickly. If it doesn’t, be sure to file a bug or ask, so we can help!

### 12.1 Requirements

By itself, Clash has fairly modest requirements, but for hacking on Clash, you’ll need a few things:

- You need **GHC 8.2.1** no matter what, as of this writing.
- You’ll need a build tool: either
  - `cabal-install` (version 2.0, or preferably a recent `git` copy)
  - `stack` (any modern version)
- You’ll need some simulation tools too, in order to run the tests and ensure you haven’t broken code generation:
  - Icarus Verilog 10.x, for Verilog Simulation.
  - GHDL, for VHDL simulation.
- Optionally, if you have ModelSim available, you can also use it for SystemVerilog simulation.

### 12.2 Cloning the code

Clash is primarily structured in two main repositories:

1. `clash-compiler.git`, containing the compiler code. This repository contains multiple packages, but the most important ones are:
   - `clash-lib`, which exposes Clash Core and the compiler backends as a library.
• *clash-ghc*, which uses *clash-lib* to provide a GHC-compatible *clash* driver. This package provides the *clash* and *clashi* executables.

• *clash-testsuite*, which contains the *clash* testsuite.

2. *clash-preflude.git*, containing the *Clash.Prelude* code that user-defined circuits use. *clash-prelude* is a submodule of *clash-compiler*.

Grab the code using *git*. **Make sure you pass --recursive so you grab the *clash-prelude* submodule!**

```
$ git clone --recursive https://github.com/clash-lang/clash-compiler.git
$ cd clash-compiler
```

You can use either *cabal* or *stack* in order to build *clash*. Either should work fine, but you must use GHC 8.2.1 for the build to work!

Furthermore, if you choose to use *cabal new-build*, a feature deficiency currently requires another step as a workaround. In order to install proper, local versions of the up to date *ghc-plugins-* packages used by Clash, you will need to clone them as a subdirectory of the *clash-compiler* repository! *Stack* will automatically check out proper revisions, and in the future, *cabal new-build* will be able to do so as well.

```
$ git clone https://github.com/clash-lang/ghc-typelits-extra
$ git clone https://github.com/clash-lang/ghc-typelits-knownnat
$ git clone https://github.com/clash-lang/ghc-typelits-natnormalise
$ git clone https://github.com/clash-lang/ghc-tcplugins-extra
```

### 12.3 Building the code

Once you’ve cloned the repository, and got a proper GHC installation (or an automated one, if you’re using *stack*), building the *clash* executables is easy using either *stack* or *cabal*.

**Warning:** While the official user recommended installation procedure uses the legacy *cabal install* scheme, if you are going to hack on Clash itself, we strongly recommend using *stack* or *cabal new-build* instead of the legacy *cabal install* commands. Currently, there is no *cabal new-install*, and there is also currently no Clash package available in any Stack snapshots or LTS releases, either. In the future, we will recommend *cabal new-install* for users to get bleeding-edge releases, as well as up-to-date Stackage LTS specs, so developers and users are always using the same build tools.

**Warning:** *stack* version 1.5.x is rather unstable when building Clash 1.x with GHC 8.2.1. If you see intermittent and non-deterministic *package registration* related failures while attempting to build, simply re-running the commands will (normally) result in a fix. Alternatively, you may use *cabal new-build*. **TODO FIXME:** report this bug upstream after diagnosing further.

**Note:** Cabal 2.0 is required for GHC 8.2.1, but it is strongly advised to build a recent copy of *cabal-install* from the *git repository* in order to get all the latest bugfixes and features, for now.

**Note:** *cabal new-build* uses a locked set of Haskell dependencies from Hackage inside *cabal.project*, *freeze*, as well as a locked index file version inside *cabal.project* (which control *cabal* *âAJrevisionsâÅI
on Hackage). This ensures that every developer has a reproducible experience, and not even dependency revisions can break this, similar to Stack snapshots.

- **Using stack to launch a copy of clashi:**

  ```
  $ stack build clash-ghc
  $ stack exec clash-ghc -- clashi
  ```

- **Using cabal new-build to launch a copy of clashi:**

  ```
  $ cabal new-build clash-ghc
  $ cabal new-run -- clashi --help
  ```

  OR, in a single command:

  ```
  $ cabal new-run clash-ghc:clashi -- --help
  ```

### 12.4 Running the test suite

**Todo:** Lorem ipsum

### 12.5 Convenient hacking tips

The following are some convenient tips for when you're hacking on the compiler and library source code for Clash.

#### 12.5.1 Debugging the compiler

The `clash` and `clashi` executables provide a `-fclash-debug=<level>` flag which allows dumping of the various intermediate representations used by the compiler. This can be used in conjunction with GHC's usual `-ddump-simpl` flag to see how the compiler is transforming your program.

- `-fclash-debug` accepts several debug levels:
  - `DebugNone` disables debug output
  - `DebugFinal` shows the final, completely normalized expression
  - `DebugName` shows the names of transformations as they are performed
  - `DebugApplied` shows the result of each sub-expression rewrite
  - `DebugAll` enables all of the above.

#### 12.5.2 Quick one-shot iteration

**Todo:** Lorem ipsum

12.4. Running the test suite 49
12.5.3 GHC 8.2 environment files

Note: This ONLY works if you are developing with cabal new-build as of right now.

Warning: You MUST also have a version of Cabal 2.1, or later, from the git repository.

When using cabal new-build 2.1 or later, cabal writes out package environment files into the root directory of the clash-compiler repository.

After you have successfully run new-build once, it will write out a file into the root directory named .ghc.environment-<platform>. This file is automatically read by the ghci command (or clashi command!) in order to load dependent packages of a project into scope.

At this point, simply executing ghci anywhere puts all dependent packages, and local packages, into scope at the REPL. This makes it easy and convenient to do things like run Clash directly from GHCi itself, using the clash-ghc library API:

```
$ cabal new-build clash-ghc
...
$ ghci
GHCi, version 8.2.1: http://www.haskell.org/ghc/ :? for help
Prelude> import CLaSH.Main
Prelude CLaSH.Main> defaultMain []
<interactive>: no input files
Usage: For basic information, try the `--help' option.
*** Exception: ExitFailure 1
```

This REPL works just like any ordinary ghci repl, or just like cabal new-repl, so youâ€™re free to develop incrementally at this point.

12.5.4 Updating Cabal freeze files

cabal new-build features a much more powerful version of the â€œfreezeâ€œ functionality available from the previous â€œCabal Sandboxâ€œ features. This feature allows the Clash developers to exactly control the given build dependencies during development, when using cabal.

Freezing is controled by two components:

- The frozen dependency specification, located in the cabal.project.freeze file at the root of the project. This exactly specifies which packages, their versions, and the build flags are needed for successful compilation.

- The Hackage index state, specified in the cabal.project file at the root of the directory. This specifies the exact Hackage index state of the Hackage package index at a given point in time. While a varying index state does not change the versions of needed, dependent packages, it may change their constraints and options, due to Hackage metadata revisions of the upstream .cabal files. (Revisions may occur in a package to try and help constrain invalid solver plans from being formed by cabal install, and also keep constraint bounds between packages consistent.)

A lock-step upgrade and pinning, of both the index state, and dependency specification, yield deterministic builds and upgrades.
**Updating the index state**

**Warning:** Updating the index state is currently a **destructive operation** that will update the package index in your $HOME directory! In the future, cabal will provide a way to query the current local and upstream index states.

The easiest way to update the index state is simply to run `cabal update twice`. The first time will bring you up to date, if necessary. The second one will be a no-op. When `cabal update` is run, it tells you what the **prior** index state was. Therefore, running it twice tells you the `current state` assuming the second operation was a no-op, like follows:

```
$ cabal update && echo && cabal update
```

Download the latest package list from hackage.haskell.org

To revert to previous state run:

```
cabal update --index-state='2017-09-20T19:05:42Z'
```

Downloading the latest package list from hackage.haskell.org

To revert to previous state run:

```
cabal update --index-state='2017-09-21T14:38:54Z'
```

In the above example, the new index state is **2017-09-21T14:38:54Z**. You can write this value into the `index-state` field of the `cabal.project` file, in the `clash-compiler.git` repository.

**Updating the freeze file**

**Todo:** Lorem ipsum...
Addendum: Hacking on this documentation

Check out the clash-docs README for information on hacking on the documentation that is in front of your eyes, right now!
CHAPTER 14

Indices and tables

- genindex
- search