# Hydra User Guide

John T. O'Donnell

September 16, 2023

## Contents

## 1 Introduction

Hydra is a computer hardware description language (CHDL or HDL) that enables you to design, analyse, and simulate digital circuits. It focuses on logic design, not on the physics of electronic components. It is concerned with how to connect logic gates into a system, but not with the electronic characteristics of an individual transistor. Hydra supports synchronous circuits, where a clock keeps all the flip flops synchronised with each other.

This is free and open source software; see README and LICENSE for details. To use the system you need the ghc compiler suite for Haskell, which is also free software. The README section explains how to download, install, and run the system.

The language provides facilities that help to design complex circuits as well as small ones. The notation is concise and readable. Large circuits can be described using circuit generators, which remove most of the repetition in a circuit description without losing any of the detail. New circuit generators can be defined: you aren't limited to the ones provided in the standard libraries.

Hydra has a semantic foundation for circuits that supports the use of equational reasoning. This is a formal method that can be used to transform a circuit to make it more efficient according to a cost model, as well as to prove that an implementation satisfies a specification. In some cases, it is possible to start with an abstract specification of the desired behavior and to derive mathematically a circuit that implements it.

Hydra is not suitable for all circuit design problems. It assumes that all the values on wires are digital, rather than continuously varying analogue voltages, and it requires sequential circuits to be synchronous, using one global clock.

## 1.1 Hardware description languages

To design a circuit, and to do anything useful with it, we need a way to describe it. The description can take two forms. A *specification* is a clear statement of what the system is intended to do, while an *implementation* shows how the primitive components can be connected together into a circuit that satisfies the specification. The specification of an adder says that the output is the sum of the inputs; the implementation of an adder says that a specific collection of logic gates connected by wires in a specific way will output the specified result. An implementation of hardware is a circuit design. Both the specification and implementation need to be clear, complete, and precise.

An implementation (i.e. a circuit design) may take the form of a *diagram* or a piece of *text*. For digital circuits, a pictorial specification is called a *schematic diagram*, while a textual specification uses a *hardware description language*. Both forms can also be used for software: a textual specification of a software algorithm uses a programming language, but there are also visual programming languages that use diagrams to describe software algorithms.

A schematic diagram is an abstract picture of a circuit. It shows the components and how they are connected with wires, but it doesn't directly describe the circuit's function. A schematic diagram implies some geometric information, such as the relative placement of components, and this geometry may be used in fabricating the circuit.

An alternative approach is to use a computer hardware description language (CHDL or HDL). A circuit specification using an HDL is a text document, so it looks superficially like a computer program, but it describes hardware rather than software. Most hardware description languages are based on existing programming languages intended for software, and they model circuits using the paradigms of programming.

An algorithm can be realised using either software, hardware, or a combination of both. Algorithms may be specified abstractly, without referring to either hardware or software. A common misconception is to think "if it's a picture, then it must be hardware; if it's text it must be software". That's doubly wrong: visual programming languages use pictures to describe software, and hardware description languages like Hydra use text to describe hardware.

Schematic diagrams and hardware description languages each have their merits. A diagram may be easier for a beginner to understand, and it's more obvious that a schematic diagram describes hardware and isn't just a computer program. However, schematics for large and complex circuits are cumbersome, while hardware description languages scale up well. Diagrams just sit there inertly on paper, and don't do anything, but CHDLs support useful software tools for simulation, fabrication, and testing.

## 1.2 Modeling circuits as functions

Some HDLs are based on imperative programming languages, using the assignment statement to model a state change in a circuit. In contrast, Hydra is based on pure functional programming, and it models a circuit as a function that takes inputs and produces outputs. This is natural, because both a circuit and a function act as a black box that takes some inputs and produces some outputs. The underlying model – circuits as functions rather than state changes as assignments – is the fundamental difference between Hydra and imperative HDLs.

Hydra is an example of an *embedded domain-specific language* (DSL or EDSL). A domain-specific language is a language intended for one specific application domain, not for general purpose programming. Hydra is suitable specifically for expressing algorithms as digital circuits, but it isn't a general purpose programming language. It is implemented by *embedding* in Haskell, a general purpose functional language. This means that Hydra doesn't have a separate compiler; instead, its implementation consists of a library of Haskell modules. A circuit specification written in Hydra is compiled by the Haskell compiler and linked with the Hydra library. However, it is best to think of Hydra as a distinct language: designing a circuit is not the same as writing a program in Haskell, and some Haskell programs don't correspond to circuits.

## 1.3 Examples

The following sections explain the language and show how to write circuit specifications. A collection of examples can be found in `hydra/examples`. If you would like to see a simple circuit before going on, look at `examples/simple/SimpleCirc.hs`. To simulate it, enter these commands:

```
cd examples/simple
ghc -e main SimpleCircRun
```

## 2 README

### 2.1 About Hydra

Hydra is a functional computer hardware description language for specifying the structure and behavior of synchronous digital circuits. It supports several levels of abstraction, including logic gates, register transfer level, datapath and control, and processors. There are tools for simulating circuits, generating netlists, and emulating instruction set architectures. It is an embedded domain specific language implemented using Haskell.

- The *User Guide* is available online at `https://jtod.github.io/home/Hydra/UserGuide/HydraUserGuide.html`. The Guide is in development, and this online link may give a newer version than the one in the installation directory.
- The User Guide is also available in the installation directory `HydraUserGuide.html`
- The *API reference* gives types of the exported definitions. If you build using cabal, it will be in the the `dist-newstyle` directory. The path depends on software versions, but may be something like this: `../../dist-newstyle/build/x86_64-windows/ghc-9.2.3/hydra-3.4.16/doc/html/hydra/index.html`. If you install using cabal, it should also be placed in the standard location for Haskell API references.

This is free and open source software released under the GPL-3 license.

- Author: John T. O'Donnell, School of Computing Science, University of Glasgow
- Copyright (c) 2023 John T. O'Donnell
- Author's home page: `https://jtod.github.io/index.html`
- License: This software is free and open source, released under the GPL-3 license. See LICENSE.txt.
- Source code repository: `https://github.com/jtod/Hydra`
- Version: see Hydra.cabal

## 2.2 Installation

Hydra runs in a shell using a command line interface. Any shell can be used; the examples use the bash shell.

### 2.2.1 Install Haskell

Hydra requires the ghc toolset for Haskell, including ghc and cabal. Haskell installers for Macintosh, Windows, and Linux are available at `https://www.haskell.org/ghcup/`.

For Windows, an alternative way to install Haskell is to use chocolatey; see `https://hub.zhox.com/posts/introducing-haskell-dev/`. If you have chocolatey installed, you can use it to install Haskell easily. Run these commands in Windows PowerShell with administrator privileges:

```
choco install ghc --force -y
choco install cabal --force -y
```

- `-y` tells choco to answer with y automatically when it needs permission to do something. Without the -y, it doesn't actually ask permission and the whole installation fails.
- `--force` shouldn't be necessary but seems to be needed if installing after a failed installation attempt.

*Verify that Haskell is working*

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.2.3
$ cabal --version
cabal-install version 3.6.2.0
compiled using version 3.6.2.0 of the Cabal library
```

*In case of difficulty*

If you have a previous installation of ghc, and a new installation fails, the following might work:

```
choco upgrade ghc cabal -y
```

### 2.2.2 Install Hydra

The Hydra source code is available at `https://github.com/jtod/Hydra`. See the Releases section on the right side of the page and click on the latest release. Download the source code file (in the Assets section), which is available in both `zip` and `.tar.gz` format. The installation file is Hydra-i.j.k.zip (or .tgz), where i.j.k is the version number (for example, Hydra-3.5.4.zip). Put the file somewhere in your user workspace and unpack it, using the correct version number:

- On Linux: tar -xzf Hydra-3.5.4.tar.gz
- On Windows use zip, 7zip or tar

This will create a directory named Hydra-3.5.4 that contains documentation (`docs` directory and `examples` directory), the source code (`src` directory), and build tools. Install using these commands (but use the right version number for Hydra):

```
cd Hydra-3.5.4
cabal install --lib
cabal haddock
```

*Verify that Hydra is working*

The following commands should simulate a 4-bit word adder for several clock cycles, with different inputs during each cycle.

```
$ cd examples/adder
$ ghc -e main Add4Run
```

*In case of difficulty*

See the user guides for ghc and cabal for more information. The `ghc-pkg list` command shows the installed packages.

Sometimes an installation may fail if there was a previous installation. If this happens, find the ghc environment default file, which is located at a path similar to the following (with your username and the right version numbers):

Linux:

```
~/.ghc/x86_64-linux-8.8.3/environments/default
```

Windows:

```
C:/Users/username/AppData/Roaming/ghc/x86_64-mingw32-9.2.3/environments/default
```

Open this file in a text editor and find the line containing an entry for hydra, which should look something like this:

```
package-id hydra-3.4.5-VeryLongHashKey
```

Delete the lines for hydra, save the `default` file, and try the `cabal install --lib` command again.

*Alternative: using ghci*

Instead of using the batch command `ghc -e main Add4Run` (where Add4Run is the name of a simulation driver), an alternative is to use the interactive Haskell interpreter:

```
$ ghci
ghci> :load Add4Run
ghci> :main
ghci> :quit
```

The ghci interpreter offers a number of debugging and tracing tools. See the GHC User Guide for details.

If you get a message saying that there are "hidden packages", copy the following into a file named `.ghci`

```
:set -package mtl
:set -package parsec
:set -package ansi-terminal
```

*Alternative: using cabal*

You can turn a directory with a circuit and simulation driver into a cabal package by defining some metadata files. Once this is done, you can then execute the driver using the command `cabal run driver`, where `driver` is replaced by the actual name of the simulation driver. One advantage of this method is that you can put command line arguments on the `cabal run` command, which you cannot do with the `ghc -e main`. See the M1 processor circuit directory for an example.

## 3 Connecting components with signals

A data value in a circuit is called a **signal**. A signal is carried by a wire, and it transmits information from one component to another. In logic design we don't usually care about the physical characteristics of a wire, although these can be important at the lower levels of chip design. Therefore we will usually refer to signals rather than wires.

The information carried by a signal may be represented as an individual bit or a cluster comprising several bits. We can also describe circuits at a higher level, where signals represent natural numbers, integers, or other data types.

A bit (binary digit) can have one of two distinct values. Several names are commonly used for these values, including 0/1, Low/High, False/True, and F/T. In real hardware a bit signal is represented by a voltage, but the precise voltage value is unimportant at the level of logic design. The particular names chosen for the two bit values are also unimportant, although they can affect the readability of a table showing the behavior of a circuit.

When Hydra prints out the values of bit signals, it will normally use 0 and 1, but you can tell it to use False and True, or any other names you prefer. One advantage of 0/1 is that they are consistent with treating a bit as a binary digit (False/True suggest treating a bit as a Boolean). Another advantage of 0 and 1 is that they take up only one character and they look different. Try reading a table showing thousands of F and T characters – they can be hard to tell apart!.

## 3.1 Components

To design a new circuit, you need to take a set of existing circuits and connect them with signals. The simplest circuits are primitive components. Every circuit is either a primitive component, or a number of circuits connected by wires.

There are two kinds of primitive: combinational components (called logic gates) and sequential components (called latches or flip flops). Here are a few of the most commonly used logic gates:

- The simplest logic gate is the inverter, which takes one input and produces one output. The inverter outputs 0 if its input is 1, and outputs 1 if its input is 0. The name of the inverter is `inv`.

- The `and2` gate takes two inputs and outputs 1 if both inputs are 1; otherwise it outputs 0.

- The `or2` gate takes two inputs and outputs 1 if either (or both) input is 1; it outputs 0 if all inputs are 0.

## 3.2 Connecting a circuit to inputs

Suppose we have two signals named `x` and `y`, and want to connect them to the inputs of an `or2` gate. This is done by writing the name of the component, followed by the names of the input signals:

```
or2 x y
```

The value of this expression is the signal which is the output of the `or2` gate. Such an expression is called an **application** because the component is applied to its input signals.

Each circuit takes a specific number of inputs, and an application using that circuit must supply the corresponding number of input signals. Here are several applications of logic gates, each with the right number of inputs:

```
inv x
and2 a one
xor3 p q r
nor4 a zero c d
```

## 3.3 Anonymous signals

A signal may be given a name, such as `x` or `y`, although this is optional. You can also refer to a signal using an application of a component to its inputs, such as `inv x`; the output of the inverter is an *anonymous* signal as it has no name.

An anonymous signal is described by an expression, and that expression is written with several tokens. When you use it as an input to a circuit, this expression must be enclosed by parentheses, to turn it into a single entity. For example, suppose we have an inverter whose input is `x`, and we want to connect the inverter's output to the first input of an `and2` gate. The second input to the `and2` gate should be `y`. Here is the correct way to write it:

```
and2 (inv x) y
```

There are two expressions following `and2`, denoting its two inputs. The first expression is `(inv x)` and it denotes an anonymous signal (the output of the inverter). The second expression is `y`, which of course is a named signal. The following notation would be wrong:

```
and2 inv x y   -- Wrong!
```

This wrong expression says that the `and2` gate is being given three inputs, and the first input (`inv`) isn't even a signal.

Parentheses are used for grouping, just as in mathematics. You don't need to use parentheses just to specify the arguments to a function (that is, the inputs to a circuit). Some programming languages requires lots of punctuation to indicate function application, but Hydra doesn't do that:

```
inv (x)                        -- Wrong!
and2 (a,b)                     -- Wrong!
nand3 (x, and2 (p,q), z);      -- Wrong!
```

In Hydra (as in Haskell) you don't need the extra parentheses and commas, and they will lead to error messages. Use parentheses only when they are necessary to get the right grouping. Here is the correct notation:

```
inv x
and2 a b
nand3 x (and2 p q) z
```

## 3.4 Standard logic gates and flip flops

The following table lists all of the standard logic gates. Each gate is shown applied to input signals named `a`, `b`, etc.

| Gate | Description |
|---|---|
| buf a | buffer |
| inv a | inverter |
| and2 a b | 2-input and gate |
| and3 a b c | 3-input and gate |
| and4 a b c d | 4-input and gate |
| or2 a b | 2-input or gate |
| or3 a b c | 3-input or gate |
| or4 a b c d | 4-input or gate |
| xor2 a b | 2-input xor gate |
| xor3 a b c | 3-input xor gate |
| xor4 a b c d | 4-input xor gate |
| nand2 a b | 2-input nand gate |
| nand3 a b c | 3-input nand gate |
| nand4 a b c d | 4-input nand gate |
| nor2 a b | 2-input nor gate |
| nor3 a b c | 3-input nor gate |
| nor4 a b c d | 4-input nor gate |
| xnor2 a b | 2-input xnor gate |
| xnor3 a b c | 3-input xnor gate |
| xnor4 a b c d | 4-input xnor gate |

The buffer produces an output that is the same as the input; it is the identify function.

Many of the logical operations can be performed on any number of inputs. For example, there is the logical conjunction (`and`) of two, three, or four inputs. These correspond to distinct logic gates: the `and2` gate has two input ports and there is no way to connect three inputs to it. Therefore Hydra doesn't have an `and` gate; it has distinct `and2`, `and3`, `and4` gates. This doesn't go on indefinitely; Hydra does not define the `and5` gate or `and73`.

If you want to and together a lot of signals, use `andw`. For example, `andw [a, b, c, d, e, f, g] = acts like =and7` would if it existed. The square bracket notation describes a word, covered in a later section.

Most of these logic gates are provided for convenience; only a few of them are necessary. For example, you can replace `and3 a b c` by `and2 a (and2 b c)`. However, logic gates with several inputs can be fabricated on chips, they are slightly more efficient, and most importantly, it's more readable to use `and3` rather than two `and2` gates.

Hydra normally uses just one kind of flip flop: the delay flip flop, named `dff`. It is sometimes called a latch. This takes one input and produces one output: for example, the value of `dff x` is the output signal produced by a delay flip flop with input `x`. All flip flops must be connected to the same clock; we do not consider the clock to be a data input to the flip flop.

## 3.5 Text and schematic

It can be helpful to give both a schematic diagram and a textual specification for a circuit. Each form of description provides insight, and having both together is often worthwhile.

It's important to check that the two descriptions of the circuit are consistent with each other. To do this,

- Check that every box in the diagram corresponds to a circuit (function) in the text, and check that every circuit in the text corresponds to a box in the diagram.
- Check that each wire in the diagram corresponds to a signal in the text.
- The wire should be connected to exactly one component output, which places a logic value on the wire. The wire may be connected to one or more component inputs.

## 3.6 Defining equation names a signal

Sometimes it's useful to give a name to a signal, rather than using it anonymously. A named signal can be used as an input to several different components, but an anonymous signal cannot. Names can also make it easier to explain the circuit, and well chosen names help document the purpose of a signal.

A signal can be named using an equation. The left hand side of the equation is the name, and the right hand side is an expression that defines the signal. The following equation says that the output of the `and3` gate has the name `x`.

```
x = and3 a (inv b) c
```

With that equation, `x` can be used as the input to any component:

```
x = and3 a (inv b) c
y = or2 a x
```

A *defining equation* defines the name on the left hand side to have the same value as the signal expression on the right hand side. The order of equations is immaterial, just as with equations in mathematics. It's fine to use a signal name before the equation that defines it. The example above could be written like this, and it would specify exactly the same circuit:

```
y = or2 a x
x = and3 a (inv b) c
```

However, in mathematics you can swap the left and right hand sides, but a Hydra defining equation *must* have the name on the left side and its value on the right side. It's called a *defining equation* because it defines the name on the left hand side to have the value of the right hand side. If you reverse the two sides, you'll get an error:

```
a = inv b       -- Correct, defines the name a
inv c = d       -- Error, need a name (not application) on left
```

Later we will see a technique called *equational reasoning*, which is useful for analysing circuits. This uses equations with a more general form. Both of the two lines in the example above are valid equations, but only the first is a valid *defining equation.* To give a name to a signal, you need to use a defining equation, where the left hand side is always a signal name.

It's a good idea to write the equations in an order that makes the circuit more readable. If most of the signals are defined before they are used, a person reading the text will get a bottom-up understanding. Using signals before they are defined gives a top-down presentation. Both styles are useful, and we will see examples of both.

Sometimes the choice between anonymous and named signals is just a matter of style. Here is a signal defined using three anonymous signals:

```
x = nand2 (xor2 a b) (inv (nor2 c d))
```

This can be rewritten so as to give every signal an explicit name, by introducing additional equations:

```
x = nand2 p q
p = xor2 a b
q = inv r
r = nor2 c d
```

Both styles are correct. Use whichever you find more readable.

## 3.7 Constant signals

A constant signal always carries the same value: either it is always 0, or always 1. The names of these two constant signals are written as `zero` and `one`. Names in Hydra always begin with a lower case letter, never with a digit. Don't use 0/1, or T/F, or True/False in a circuit specification; those notations have other meanings and will lead to bizarre error messages.

```
x = and2 a one    -- Correct, x is same as a
y = and2 a 1      -- Error, 1 isn't a signal
```

## 3.8 Standard logic gates and flip flops

A logic gate is a primitive combinational component (combinational means that the output depends on the current values of the inputs).

There are several libraries of existing circuits that you can use, and you can also define libraries of your own circuits for further use. The Hydra libraries provide as primitives the standard logic gates, summarised in the following table.

| Gate | Description |
|---|---|
| buf a | buffer |
| inv a | inverter |
| and2 a b | 2-input and gate |
| and3 a b c | 3-input and gate |
| and4 a b c d | 4-input and gate |
| or2 a b | 2-input or gate |
| or3 a b c | 3-input or gate |
| or4 a b c d | 4-input or gate |
| xor2 a b | 2-input xor gate |
| xor3 a b c | 3-input xor gate |
| xor4 a b c d | 4-input xor gate |
| nand2 a b | 2-input nand gate |
| nand3 a b c | 3-input nand gate |
| nand4 a b c d | 4-input nand gate |
| nor2 a b | 2-input nor gate |
| nor3 a b c | 3-input nor gate |
| nor4 a b c d | 4-input nor gate |
| xnor2 a b | 2-input xnor gate |
| xnor3 a b c | 3-input xnor gate |
| xnor4 a b c d | 4-input xnor gate |

The buffer produces an output that is the same as the input; it is the identify function.

Most of these logic gates are provided for convenience; only a few of them are necessary. For example, you can replace `and3 a b c` by `and2 a (and2 b c)`. However, logic gates with several inputs can be fabricated on chips, they are slightly more efficient, and most importantly, it's more readable to use `and3` rather than two `and2` gates.

## 4 Defining new circuits

To design larger scale systems, we need the ability to define a circuit as a new *black box* component that can be reused to define even larger circuits. A new circuit can be designed by connecting together a number of existing ones. This is similar to using abstraction in a programming language by defining a function or procedure for a commonly used computation. A circuit definition contains up to three parts:

- Circuit type (optional)
- Circuit defining equation (mandatory)
- Internal signals (optional)

Circuit types are discussed in a later section. The circuit type is a line that looks something like this:

```
halfAdd :: Signal a => a -> a -> (a,a)
```

### 4.1 Circuit defining equation

An equation that defines a circuit called `circuit_name` has this general form:

```
circuit_name input1 ... inputn = output
```

The left hand side of the equation is an application of the circuit's name to the names of the input signals, for example `input1`, etc. There can be any number of inputs. The right hand side is the value of the circuit's output.

As a simple example: `mycirc` takes two inputs. It outputs 1 iff the first input is 1 and the second is 0:

```
mycirc a b = and2 a (inv b)
```

As another example, suppose we want to define a new circuit called `and5` that takes five inputs and outputs 1 if and only iff all the inputs are 1. It is a 5-input `and` gate, but `and5` is not one of the standard primitives. This will do it:

```
and5 a b c d e = and2 (and3 a b c) (and2 d e)
```

Given this definition, you can use `and5` by applying it to some input signals, for example `and5 x y p q r`.

The input names in the circuit defining equation are local. They are in scope only within the definition of the circuit, including the right hand side of the circuit defining equation.

## 4.2 Black boxes with internal signals

The expression that defines the circuit's output can become fairly complicated, and it's often simpler to define it using several other named signals which are inside the black box. Each of these needs a defining equation. To do this, write the keyword **where** after the circuit defining equation, and after the **where** you can write any number of signal defining equations. The general form is:

```
circuit_name input1 ... inputn = output_expression
  where x = ...
        y = ...
        ...
```

The definition of `and5` above had a fairly large output expression. This can be simplified by defining some internal signals `and5`:

```
and5 a b c d e = and2 p q
  where p = and3 a b c
        q = and2 d e
```

This definition describes exactly the same circuit as the previous version of `and5`; the only difference is that local names `p` and `q` are given to the outputs of the internal `and3` and `and2` gates. These names are just part of the *description* of the circuit; both versions of `and5` have exactly the same components and wires.

Here is an example of a circuit named c22 that takes three inputs and produces one output.

```
c22 a b c = x
  where
    x = xor2 p q
    p = and2 a b
    q = or2 b c
```

The equations should be indented consistently, and there is no extra punctuation (no curly braces, no semicolons). The compiler determines the structure of a definition from the indentation, not from punctuation. Therefore

The indentation is mandatory: it determines which equations are inside the black box. Note that you don't need curly braces around the internal equations, and don't need semicolons at the end of each line. The essential rules to remember are that

- Each internal equation must be indented relative to the circuit defining equation
- All the internal equations must start in the same column

The indentation is essential, and if it's wrong then the specification will be parsed incorrectly. There are several indentation styles, which differ in the placement of the **where** keyword and what column the internal equations start in. Each of the following examples is ok; use whichever you prefer.

```
c22 a b c = x
  where
    x = xor2 p q
    p = and2 a b
    q = or2 b c

c22 a b c = x
  where x = xor2 p q
        p = and2 a b
        q = or2 b c

c22 a b c = x where
  x = xor2 p q
  p = and2 a b
  q = or2 b c
```

## 4.3 Feedback

A register is a circuit with an internal state, and with the ability to load an external value into the state and to read out the state.

```
module Reg1 where
import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational

reg1 :: CBit a => a -> a -> a
reg1 ld x = r
  where r = dff (mux1 ld r x)
```

The reg1 circuit has a feedback loop: the output of the flip flop is connected to one of the inputs to the mux1, whose output in turn is input to the flip flop. Hydra does not allow feedback loops in pure combinational logic, but feedback that goes through a flip flop is fine. When a circuit contains a feedback loop, there will be a circular path in the schematic diagram, and there will be circular equations in its specification. For the reg1 circuit. the feedback loop can be seen in the equation which has r on both the left and right hand side. Thus r is being defined in terms of itself. The way this works, and the reason that r is well-defined, is explained in the section on circuit semantics.

```
-- Simulation driver for reg1
module Main where
import HDL.Hydra.Core.Lib
import Reg1

main :: IO ()
main = do
  runReg1 testdata

testdata :: [[Int]]
testdata =
------------------------
--  ld  x       output
------------------------
  [ [1, 1],  -- 0  output in cycle 0 is initial state
    [0, 0],  -- 1  state changed to 1 at tick between cycles 0/1
    [0, 1],  -- 1  no change
    [0, 0],  -- 1  no change
    [1, 0],  -- 1  still see 1 but at end of cycle, set state to 0
    [0, 0],  -- 0 during this cycle can see result of state change
    [1, 1],  -- 0 but set state to 1 on tick at end of cycle
    [1, 0],  -- 1 the 1 becomes visible in this cycle
    [0, 0],  -- 0 the 0 now becomes visible
    [0, 0],  -- 0 no change
    [0, 0]   -- no comma after last element of list
  ]

runReg1 input = runAllInput input output
  where
-- Input signals
    ld = getbit input 0
    x  = getbit input 1
-- Circuit
    y = reg1 ld x
-- Format the input and output signals
    output =
      [string "Input ld = ", bit ld,
       string " x = ", bit x,
       string "   Output = ", bit y]
```

## 4.4 Producing several outputs: halfAdd

The circuits we have looked at so far have just one output. We need a way to allow circuits to produce any number of outputs.

Files: **examples/adder/HalfAdd.hs** and **examples/adder/HalfAddRun.hs**

A half adder circuit takes two inputs **x** and **y**, and produces a pair of outputs, the carry output and the sum output. The carry is the logical **and** of **x** and **y**, while the sum is their exclusive **or**. The circuit specification is in the file `HalfAdd.hs`.

The module statement gives a name to this module, and the import statement brings in the essential Hydra library definitions. The circuit definition is a one-line equation which says **halfAdd** is a circuit, gives names **x** and **y** to its inputs, and calculates the outputs using **and2** and **xor2** logic gates.

To see the circuit working, we can simulate it. This requires three things, all provided in **HalfAddRun.hs**:

- Suitable test data, expressed as a list of **[x,y]** inputs

- A [Simulation driver](#simulation-drivers), which converts between human readable input and output and the internal signal representations. The simulation driver is not part of the circuit; it's simply formatting inputs and outputs.

- A main program that runs the simulation driver on the test data.

See 'examples/adder/HalfAddRun.hs'

Run the simulation using any of the methods given above, e.g. enter **ghc -e main HalfAddRun**. Here is the result:

```
$ ghc -e main HalfAddRun
Input: x = 0 y = 0  Output: c = 0 s = 0
Input: x = 0 y = 1  Output: c = 0 s = 1
Input: x = 1 y = 0  Output: c = 0 s = 1
Input: x = 1 y = 1  Output: c = 1 s = 0
```

## 4.5 Example: multiplexer

The *multiplexer* is an example of a circuit that can be defined by conecting several logic gates together. The multiplexer is one of the most important building blocks for larger systems. There are many varieties of multiplexer; here we look at the 1-bit multiplexer, called `mux1`.

Files:

- `examples/mux/Mux1.hs`

- `examples/mux/Mux1Run.hs`

A multiplexer is a hardware version of the if-then-else expression, and is used to perform conditional actions in a circuit. It takes three inputs: a control input `c`, and two data inputs `x` and `y`.

The idea is that the multiplexer will choose one of the data inputs – either `x` or `y` – and output it. The data input that is not chosen is simply ignored. The choice is determined by the value of `c`.

The behavior of the multiplexer can be described informally using an if–then–else expression. This is just an English description, not a circuit, because **if** is not a circuit: it's programming language notation.

```
-- informal specification
mux1 c x y = if c = zero then x else y
```

The multiplexer can be implemented with logic gates. The circuit is defined in the file `Mux1.hs`, and a *simulation driver* is in `Mux1Run.hs`. A simulation driver is a software interface to the actual circuit; the software takes inputs in a readable notation, formats the outputs, and controls the simulation.

Files:

- `examples/mux/Mux1.hs`
- `examples/mux/Mux1Run.hs`

To run the simulation, enter

```
cd examples/mux
ghc -e main Mux1Run
```

Given inputs c, x, y, the circuit outputs a bit whose value is

```
or2 (and2 (inv c) x) (and2 c y)
```

There are several ways to understand how this Boolean logic implements the if-then-else construct. One way is simply to build a truth table showing the output for each possible input.

This expression can be read by working through each subexpression:

- `inv c` says there is an inverter, and its input is connected to `c`. The subexpression `inv c` denotes the output signal produced by the inverter.
- `and2 (inv c) a` denotes the output of an and2 gate; its first input is the output of the inverter and its second input is `a`. This signal is the first input to the `or2` gate.
- `and2 c y` is the output of an and2 gate with inputs c and y; the output of the gate is the second input to the `or2` gate.

The file `examples/mux/Mux1.hs` defines the circuit as follows.

```
mux1 c x y = or2 (and2 (inv c) x) (and2 c y)
```

Circuit definitions are discussed in more detail later, as well as the rest of the file:

```
module Mux1 where
import HDL.Hydra.Core.Lib

-- mux1 is defined in the Hydra circuit libraries, so here
-- the circuit is called mymux1 to ensure that we're testing
-- this definition

mymux1 :: Bit a => a -> a -> a -> a
mymux1 c x y = or2 (and2 (inv c) x) (and2 c y)

module Main where
-- Mux1Run: test the mux1 circuit

import HDL.Hydra.Core.Lib
import Mux1

main :: IO ()
main = mux1Run testdata

testdata :: [[Int]]
testdata =
-----------------------------------------
--   c  x  y       expected result
-----------------------------------------
  [ [0, 0, 0]  --  0  (c=0 so output=x)
  , [0, 0, 1]  --  0  (c=0 so output=x)
  , [0, 1, 0]  --  1  (c=0 so output=x)
  , [0, 1, 1]  --  1  (c=0 so output=x)
  , [1, 0, 0]  --  0  (c=1 so output=y)
  , [1, 0, 1]  --  1  (c=1 so output=y)
  , [1, 1, 0]  --  0  (c=1 so output=y)
  , [1, 1, 1]  --  1  (c=1 so output=y)
  ]

mux1Run input = runAllInput input output
  where
-- Extract input signals
    c = getbit input 0
    x = getbit input 1
    y = getbit input 2
```

```
-- The circuit to be simulated
    z = mymux1 c x y
-- Format the output
    output =
      [string "  c=", bit c,
       string "  x=", bit x,
       string "  y=", bit y,
       string "    output z=", bit z
      ]
```

We can run it with a simulation driver that runs the circuit on all possible inputs, so the outputs form a truth table. It's good practice to write the test data with clean indentation, so the inputs line up in columns, and to include the expected outputs in comments.

See examples/mux/Mux1Run.hs')

## 5 Modules and files

We start with an example in the directory `examples/simple`, consisting of three files. We will normally keep circuit definitions and their simulation drivers in separate files, although that isn't required. For a circuit named `MyCirc.hs`, the naming convention is to call the driver `MyCircRun.hs`.

- `SimpleCirc.hs` defines a circuit named `simpleCirc`, which takes two input bits and outputs their logical *and*. The circuit is just an `and2` logic gate.

### 5.1 The circuit specification

The circuit is defined in a module `SimpleCirc`, which is in the file named `SimpleCirc.hs`.

```
-- SimpleCirc: minimal example of a circuit specification
-- This file is part of Hydra. See README and https://github.com/jtod/Hydra
-- Copyright (c) 2022 John T. O'Donnell

module SimpleCirc where
import HDL.Hydra.Core.Lib

-- Define a circuit "simpleCirc" which takes two input bits and
-- outputs their logical conjunction, using an and2 logic gate.

-- To run the circuit simulation, see SimpleCircRun and
-- SimpleCircRunInteractive.

simpleCirc :: Bit a => a -> a -> a   -- interface: two inputs, one output
simpleCirc x y = and2 x y            -- circuit is just an and2 logic gate
```

A module begins with a `module` statement that gives its , followed by statements that import other modules. All Hydra modules must contain `import HDL.Hydra.Core.Lib`, which defines, among other things, the `and2` logic gate.

Following the imports, the module may contain any number of circuit definitions. The definition of the circuit `simpleCirc` contains two lines of code: a *type declaration* which contains the symbol `::` and a *defining equation* which contains the symbol =.

A type declaration specifies the name of the circuit and its interface. The declaration *simpleCirc :: Bit a => a -> a -> a* contains several parts:

- The beginning means "simpleCirc has type. . . ".
- Bit a means the circuit uses Bit signals, and we will use the name *a* for the type of a bit signal
- *a -> a -> a* says the circuit takes an input of type *a*, a second input also of type *a*, and it outputs a signal of type *a*
- There may be any number of inputs, and each is followed by ->. This means that the number of inputs is the number of -> in the type.
- There must be exactly one output. The last *a* in the type is the type of the output. Later, we will see how to allow a circuit to produce several outputs.

The defining equation specifies local names for the circuit's inputs, and it gives a circuit that produces its output. The defining equation for this circuit is `simpleCirc x y = and2 x y`. This says

- We will use the names `x` and `y` as local names for the inputs to the circuit
- There is an `and2` logic gate with inputs `x` and `y`
- The output of that logic gate is the output of the circuit

### 5.2 Modules and files

## 6 Circuit simulation

Running a circuit is necessary for testing, and observing its behavior also helps in understanding. One way to run a circuit is to fabricate it in hardware, connect it to I/O devices, and observe its behavior. That takes a lot of time. It's easier and faster to *simulate* the circuit, which allows you to run it in software.

This section shows how to simulate circuits using automated tools. These techniques are well suited for small circuits, such as registers and adders. Section ?? introduces more powerful tools which are valuable for larger and more complex circuits, such as processors.

We start with an example in the directory `examples/simple`, consisting of three files. We will normally keep circuit definitions and their simulation drivers in separate files, although that isn't required. For a circuit named `MyCirc.hs`, the naming convention is to call the driver `MyCircRun.hs`.

- `SimpleCirc.hs` defines a circuit named `simpleCirc`, which takes two input bits and outputs their logical *and*. The circuit is just an `and2` logic gate.
- `SimpleCircRun.hs` defines a simulation driver for the circuit. The file provides a main program named `main` which runs the simulation on test data which is also defined in the file.
- `SimpleCircRunInteractive.hs` is the same as `SimpleCircRun`, except it runs interactively. On each clock cycle it prompts the user to enter the input values, and then displays the outputs.

To run the simulation, go to `examples/simple` and enter this on the command line: `ghc -e main SimpleCircRun`. The driver runs the circuit on all possible inputs: 00, 01, 10, 11. Since the circuit is really just an `and2` gate, the output should be 0, 0, 0, 1.

### 6.1 The simulation driver

To test the circuit, we can simulate it with some inputs. This requires a *simulation driver* which is defined in a separate module in the file SimpleCircRun.hs. The simulation driver defines the interface to the circuit and specifies how to format the inputs and outputs, so you don't have to read and write thousands of 0s and 1s.

The file begins by defining the module and importing some required libraries. Every Hydra file needs to import HDL.Hydra.Core.Lib. In addition, a simulation driver like SimpleCircRun needs to import the circuit file, which should be in the same directory.

```
-- SimpleCircRun:  simulation driver for SimpleCirc
-- This file is part of Hydra. See README and https://github.com/jtod/Hydra
-- Copyright (c) 2022 John T. O'Donnell

-- Usage:  $ ghc -e main SimpleCircRun

module Main where
import HDL.Hydra.Core.Lib
import SimpleCirc
```

A simulation driver can define input data, although this is not required. The data is given as a list of strings. Each string corresponds to one clock cycle, and it gives the values of the inputs during that cycle.

```
testdata1 :: [String]
testdata1=
------------------------------
--   x   y     expected output
------------------------------
  [ "0  0"   --  0
  , "0  1"   --  0
  , "1  0"   --  0
  , "1  1"   --  1
  ]
```

The file defines a function `main` that will run the simulation. These two lines are boiler plate and every driver should contain them:

```
main :: IO ()
main = driver $ do
-- Input data
  useData testdata1

-- Inputs
```

```
  x <- inputBit "x"
  y <- inputBit "y"

-- Circuit
  let z = simpleCirc x y

-- Output ports
  outputBit "z" z

-- Run
  runSimulation
```

If you want the simulation to use any test data (e.g. testdata1), the useData statement specifies which test data to use. This command is optional; if you don't include it (or if you comment it out) the simulation will be interactive, and prompt you for the input values every clock cycle. But if the useData statement is present, the simulation will run to completion without any intervention by the user.

```
useData testdata1
```

Every driver is required to define an input port for every circuit input. The convention is that in input signal named x has an input prt named $in_x$. Each input port is created with a statement of the form portname <- inPortBit "signame". If the inport is a word, rather than a bit, you would definit like this: $in_w$ <- inPortWord "w" 8 where 8 is the word size.

There must be an input signal for each port. These are defined using inbsig for bits, and inwsig for words.

What is the difference between a port and a signal? A signal is a wire, and it can be connected directly to a circuit. A port is a data structure that contains the signal and also some addiitonal metadata required by the simulator.

Notice that ports are defined using <- and signals are defined with the let keyword and =.

The next piece of the driver is an equation that connects the input and output signals to the circuit. This equation is given in a let statement

The simulator will print out the values of all the signals for each clock cycle. You can either have the system do this automatically, or you can supply a detailed format that gives precise control over the output.

The easiest approach is to ask for automatic output of all the signals. To do this, you need to define an output port for each output signal. This uses the outPortBit function, which takes the name of the signal "z" and the corresponding signal z.

The final piece of the driver is the command runSimulation. If you omit this, the simulation won't run and there won't be any output.

```
runSimulation
```

Alternatively, you can use the format statement which gives precise control over the formatting of the simulation output. The format is a list of field specifiers. The field specifier string "..." says that the literal string "..." should appear in the output. The field specifier bit x says that x is a signal; its value should be printed in the output (it will be either 0 or 1).

```
-- Formatted output
  format [string "x=", bit x,
          string "  y=", bit y,
          string "   output = ", bit z,
          string "\n"
         ]
```

There are several format specifiers, that allow output of words using bit strings, decimal, hexadecimal, and both binary and two's complement. See examples/SimDriver for examples of these specifiers.

## 6.2 Running the circuit simulation

To run a circuit you should cd to the directory that contains the circuit (Circuit.hs) and its simulation driver (CircuitRun.hs) files.

One way to run the simulation is to use the ghc command, which will run the main program in CircuitRun as a single command:

```
ghc -e main CircuitRun
```

An alternative is to use the ghci command, which launches the interactive version of ghc. This offers many useful debugging tools, but you need to give it several commands to run the simulation:

```
ghci
:load CircuitRun
:main
```

### 6.3 Batch and interactive simulation

You can run a circuit simulation in batch mode, which will run to completion with no intervention by the user. Alternatively, you can run it interactively. The choice between the modes is determined by just one line of code: the presence or absence of a useData statement.

If you run a simulation in interactive mode, the system will prompt "hydra>" and wait for you to enter a command. When it does so, just press Enter, which is the command to simulate one clock cycle. It will then prompt you for the value of each signal; enter a decimal number. When all the inputs have been provided, the driver will now print out all the signal values and prompt for a command again. The help command will print a list of the available commands.

You can try out both approaches with the SimpleCircRun driver. It contains a useData, so it will run in batch mode. But if you comment out the useData, the driver will revert to interactive mode.

Files: **adder/Add4.hs** and **examples/Add4Run.hs**

```
*Main> :main
  x =  5  y =  8  cin = 0    ==>    cout = 0  s = 13
  x =  7  y =  3  cin = 0    ==>    cout = 0  s = 10
  x =  8  y = 12  cin = 0    ==>    cout = 1  s =  4
  x =  8  y =  1  cin = 0    ==>    cout = 0  s =  9
  x = 12  y =  1  cin = 1    ==>    cout = 0  s = 14
  x =  2  y =  3  cin = 1    ==>    cout = 0  s =  6
  x = 15  y = 15  cin = 1    ==>    cout = 1  s = 15
(0.00 secs, 252,808 bytes)
*Main>
```

This part of a definition is optional; if present it follows the **where** keyword.

## 7 Syntax

This section summarises the language syntax. Hydra is actually Haskell with some additional libraries, and it adopts all the syntax rules of Haskell.

### 7.1 Modules

### 7.2 Comments

There are two ways to indicate a comment

- Enclose the comment in brackets {- so this is a comment -}
- A double dash – indicates that everything else – on the line is a comment

## 7.3 Names

Names (also called identifiers) are used for circuits (e.g. logic gates) and signals. A name must begin with a lower case letter, and may contain letters, digits, underscores, and primes (single quote). The following are valid names:

```
x
select
adder
y'
bypass_ctl

Product    -- begins with upper case letter
x?3        -- contains invalid character ?
0          -- use zero to get the constant 0 signal
```

The following identifiers cannot be used to name a circuit or signal:

## 7.4 Scope

## 7.5 Signal expressions

A signal is specified in Hydra by an expression. The simplest form of expression is simply the name of a signal. For example, suppose we have signals named **x** and **y**. Then the following expressions denote the corresponding signals. Note that **zero** and **one** are simply the names of the constant signals.

```
zero
one
x
y
```

A signal is denoted by an expression, which can have any of the following forms:

- The constants **zero** and **one** are names that are pre-defined. These should not be redefined: don't write zero or one on the left hand side of an equation.
- The name of a signal which is in scope: **x**, **carry**, $\mathbf{ctl_{ld}}$. Later we will see how to define these, using equations or circuit inputs.
- An application of a circuit to inputs denotes a signal: **or2 x y** specifies a signal which is the output of an **or2** gate connected to inputs **x** and **y**.

Any of these notations can be used as input to a component.

The component and the inputs are separated by a space; don't use punctuation.

```
c = and2 a b          -- correct
d = and2 (a,b);       -- wrong! don't use ( , ) ;
```

## 7.6 Indentation

Haskell normally uses indentation, rather than punctuation, to determine the structure of a definition. There are good reasons behind this approach to syntax.

It is also possible to use braces and semicolons to determine the structure, instead of indentation. This is particularly useful for generators, where the specification is not written by hand and also not intended to be human readable. There are also some situations where a large number of very short equations can be more readable with many placed on each line, separated by punctuation. However, these situations are relatively uncommon. Normally it's best to use indentation and to make the layout of the code as readable as possible.

The equations in a definition need to be lined up vertically

```
circ x y = a   -- a good definition
  where
    p = bla bla...
    q = bla...
    r = bla...

circ x y = a  -- a bad definition
  where
    p = bla bla...
   q = bla...
    r = bla...
```

## 8 Interfaces and types

The **type** of a value determines what operations you can perform on it. This holds for hardware description just as for programming. The type of a signal determines what kind of information it carries, and the type of a circuit specifies the types and organisation of its input and output signals.

A circuit has an interface to the outside world, and an internal organization. To use the circuit, all we need to know about is the interface: what inputs need to be provided and what the outputs mean. The type expresses a useful portion of this information: it describes the number and organization of the inputs and outputs. The meanings of the circuit outputs are not specified by the type; they should be described in documentation for the circuit. Since Hydra models a circuit as a function, a circuit type looks just like a function type.

The type declaration for a circuit is optional, as the compiler can work out the type for itself. If you omit the type, your circuit will still run. However, there are several benefits in writing out the type explicitly:

- The type gives useful information about the interface to the circuit. Later on, if you want to use this circuit in a larger one, you will be more interested in the interface than the internal components inside the circuit.

- There is some redundancy between the type and the defining equation. If there is any inconsistency between the two, the compiler will give a type error message. That may be annoying, but at least you know that the error lies somewhere in the (small) specification of this one circuit. If you omit the type declaration, but there is an error in the defining equation, you may get an error message that says, in effect, "there is an error somewhere in the (large) file", but it's up to you to figure out **where** the error is.

- If you do get a type error message, the compiler will do its best to give a helpful and informative message. In practice, though, the error messages will be far more understandable if you include type declarations for your circuits.

If present, the type of a circuit should come immediately before the defining equation. Type declarations are easily recognizable: they always contain the symbol **::**, and usually contain some arrows => and **->**. A typical example is

```
reg1 :: CBit a => a -> a
```

A type declaration contains several parts:

- The circuit name (e.g. reg1)

The symbol, read as "has type"

- The signal class ending with => (e.g. CBit a =>)
- The input and output signal types (e.g. a -> a)

## 8.1 Interface

The interface gives the name of the circuit and names its inputs and outputs. A circuit is created with a **circuit defining equation**. The left hand side of the equation is the name of the circuit followed by the names of the input signals. There may be any number of inputs. The right hand side is an expression giving the value of the output signal:

```
circ_name input1 input2 = expression
```

This defines a circuit whose name is $\mathbf{circ_{name}}$, which takes two inputs named **input1** and **input2**, and produces an output with the specified signal value. Here is an example:

```
mycirc a b c = and3 a (inv b) c
```

The input names **a**, **b**, and **c**, are local to the definition of **mycirc**, and they can be used to calculate the value of the output. Another circuit can connect signals with arbitrary names, or no names at all, to the inputs of **mycirc**.

## 8.2 Interfaces and Types

The *type* of a circuit describes its interface; in particular

- how a signal is represented: this determines the choice of semantic model;
- how many inputs it takes, how they are structured;
- how many outputs it produces, and how they are structured;
- the size parameters, if any;
- the building block circuits, if any.

If you try to plug the wrong number of signals into a circuit, you will get a type error message.

## 8.3 Signal types

**Short version.** If you're writing a routine circuit and just want to simulate it, you can just write **CBit a** => for the signal class constraint and then use **a** as the type for every bit signal. In more complicated situations, or if you want to know what this means, read on.

When a circuit specification is executed, each signal has a specific type. Many types can be used, for example **Bool** or **Stream Lattice**. The choice of type determines what happens during execution. Some types lead to combinational simulation, others lead to synchronous simulation, others perform a path depth analysis, or generate a netlist.

It's possible to define a circuit with a specific type, and if you do this the class constraint (the part before =>) is omitted. For example, we could define a Bool version of the mux1 circuit (call it halfAddB) to operate in signals of type Bool:

```
halfAddB :: Bool -> Bool -> (Bool,Bool)
halfAddB x y = (and2 x y, xor2 x y)
```

This is a little simpler than the standard definition halfAdd, which (1) uses the type class constraint Bit a =>, and (2) uses **a** rather than **Bool** as the bit signal type.

## 8.4 Inputs and outputs

After the signal class (i.e. after the => symbol) come the types of the inputs and output of the circuit. In the simplest case, each input or output signal is just a bit of type **a**. There may be any number of input arguments, and there must be one output result. A single arrow **->** must follow each input; thus the number of single arrows in the type is the same as the number of inputs.

The inverter has one input of type **a**, which is followed by **->**, and the type **a** of the output appears last. The type declaration can be read as "inv uses signals in the Bit class; it takes one input and produces one output": Thus the entire type declaration “*inv :: Bit a => a -> a*” says “*inv* is a circuit that takes an input bit signal, and produces an output bit signal.”

```
inv :: Bit a => a -> a
```

The notation **a -> a** means "the circuit takes an input signal and produces an output signal". This is similar to conventional mathematical notation; for example in mathematics there is a function **im** that is given a complex number (type $C$) and returns its imaginary part (type $R$), and a mathematician might write its type as im : C -> R. (The reason :: is used in Haskell (and Hydra) is that : is used for something else.)

Circuits that take several inputs have a slightly more complicated type. For example, here are the types for the family of and-gates:

```
and2 :: Bit a => a -> a -> a
and3 :: Bit a => a -> a -> a -> a
and4 :: Bit a => a -> a -> a -> a -> a
```

There is always one output, but any number of inputs, and every input is followed by **->**. To find out how many inputs a circuit takes, just count the number of times **->** appears in its type.

If a circuit has several outputs, they must be enclosed in a container, and this is reflected in the type. See the section on Containers.

## 8.5 Circuit type

The circuit type is covered in a later section. It’s optional, although it is generally best to include it. If present, the type can be recognized by the :: symbol and a number of right arrow symbols; a typical example is

```
halfAdd :: Bit a => a -> a -> (a,a)
```

## 8.6 Signal classes

### 8.6.1 Combinational signals: Bit a

```
halfAdd :: Bit a => a -> a -> (a,a)
halfAdd x y = (and2 x y, xor2 x y)
```

The main disadvantage of using Bool as the signal type is that combinational simulation is the **only** thing you can do with the circuit. However, Hydra provides many other options. For example, you can perform synchronous simulations over many clock cycles, but to do that, the signals must have a different type. You can do these other things with **halfAdd**, but not with **halfAddB**.

There are several different types that can be used to represent a signal. These are organized into two main sets: **Bit** and **CBit**. **Bit** is used for combinational circuits, and **CBit** ("clocked bit") is used for sequential circuits.

signal. The notation **Bit a** => means that **a** can be any type in the set **Bit**, and therefore all of the Bit operations can be performed on a signal of type **a**.

The commonest signal class constraints are:

- **Bit a** => is used when **a** is a bit signal in a combinational circuit. The circuit may contain logic gates, but not flip flops.
- **CBit a** => is used when **a** is a bit signal in a sequential circuit, which may contain flip flops and feedback loops as well as logic gates.

### 8.6.2 Clocked signals: CBit a

The signal class constraint Classes

Base signal types

- Bool (defined in Haskell standard libraries)
- Word16 (defined in Haskell standard libraries)
- Word32 (defined in Haskell standard libraries)
- Lattice (defined in Hydra Core library)

## 8.7 Multiple outputs

## 8.8 Form of a type declaration

```
circname :: signalClass => inType -> inType -> outType
```

- circname is the name of the circuit; starts with lower case letter
- signalClass gives information about the signal. We will normally use either {Bit a} or {CBit a}, but there are many other options we won't be using
- input and output types: {a} for a bit signal, {[a]} or {(a,a)} for a container holding several bits

## 8.9 Type declaration is optional

- You can omit the type declaration; the compiler will work it out automatically
- There are some advantages for declaring the type:
  - It provides helpful documentation
  - It enables the compiler to give better error messages, if there is an error in a circuit
  - A useful design technique is to specify first the types of your circuits and to go back later to implement them

## 8.10 Signal types and circuit semantics

- There is a close relationship between *{*how a signal is represented} and *how the circuit is evaluated.*
  - Example: if you use Bool, then you can do logic simulation but not synchronous simulation.
  - If you use streams, you can handle timing.
  - If you want to allow special techniques (tristate drivers, wired or) you need a multiple-value logic.
- For this reason, it's not a good idea to give circuit types like `inv :: Bool -> Bool`.
- Instead, we use a *type variable*, typically `a`, to represent the signal type.

## 8.11 The signal class constraint

**When you write {inv** Bit a => a -> a}, the {Bit a =>} part means "for an arbitrary type `a`, provided that `a` is in the set of types that can represent a bit signal".

**Similarly, {reg1** CBit a => a -> a -> a}, the constraint `CBit a =>` means the signal type `a` has to be capable of representing clocked synchronous bit signals.

Hydra offers many signal representations.

Normally, the choice of specific signal type will be made by the simulation driver.

## 8.12 Naming conventions

- By convention, the type of a bit signal is called `a`.
- The scope of the type variable `a` is the type statement.
- This means you can have a signal named `a` in the circuit definition, without confusion.

```
circ1 :: Bit a => a -> a -> a
circ1 a b = and2 a (inv b)
```

In the first line, `a` is the signal type, but in the second line `a` is the name of the first input signal.

## 8.13 The usual cases

- For a combinational circuit, use {|Bit a =>|} and the type of a bit signal is {|a|}
- For a clocked synchronous circuit, use {|CBit a =>|} and the type of a bit signal is {|a|}

There are many other options for the signal classes, but we won't use them in CA4.

### 8.14 Input and output types

- A bit signal is indicated by the signal type `a`.
- Each input signal group is followed by an arrow `->`. This means that the number of arrows in the type is the number of inputs.
- The output signal group follows the last arrow.

**{and2** Bit a => a -> a -> a} has two arrows `->`, so there are two input groups, both with type `a`. The output has type `a`.

**{mux2** Bit a => (a,a) -> a -> a -> a -> a -> a} has five input groups: a bit pair (used for control), four bits (for data), and there is one output bit.

## 9 Containers

- Most circuits (except the simplest ones) have groups of signals that travel together.
- To keep circuit descriptions concise, we need to be able to put these signals into a container and treat it as a single entity
- There are two kinds of container:
  - A  is a group of several signals that may be unrelated
  - A  is a sequence indexed by a bit position, normally used for binary numbers
- A container of signals is indicated by a tuple type `(a,[a])` or by a word type `[(a,a)]`.
- Containers aren't part of the hardware! They are just *notation* used to shorten the description of the hardware.
- *Any* circuit can be described without groups—but the description could be millions of lines long

## 9.1 Tuples

You construct a tuple simply by writing its components in parentheses:

```
code = (x,y,z)
```

Given a tuple, you can supply names for its components with an equation. The tuple is a *pattern* that defines the elements

p , q , r

```
(p,q,r) = code
```

## 9.2 Circuits with several outputs

- Hydra requires every circuit to have *one* output.
- If there are several output signals, just group them in a tuple.

The half adder and demux1 circuits produce two outputs, which can be specified as expressions in a tuple.

```
halfAdd x y = (and2 x y, xor2 x y)
demux1 c x = (and2 (inv c) x, and2 c x)
```

In |demux2| it's necessary to define the output as a tuple of names which are defined by equations.

```
demux2 (c0,c1) x = (y0,y1,y2,y3)
  where  (p,q) = demux1 c0 x
         (y0,y1) = demux1 c1 p
         (y2,y3) = demux1 c1 q
```

## 9.3 Words

- A word $[x_0, x_1, \ldots, x_{k-1}]$ is used to represent binary numbers.
- There are notations to allow you to build words and extract parts of words.
- Some circuits are defined recursively: the definition for a $k+1$ bit word uses a subcircuit of size $k$.
- Sometimes you need to extract a bit or a field from a word, or construct new words.
- Therefore we need operators to add a new bit to a word, extract a bit from a word, etc.

### 9.3.1 Word size

If `x` is a word, then `length x` gives the number of elements.

```
n = length x
```

- With this definition, `n` can be used as a size parameter to define other circuits.
- Important: `length` is not a circuit! It is a meta-operator; it is a calculation on the notation.

### 9.3.2 Constructing words

1. List notation for words

   A word can be either

   - An *empty word*, written `[]`, which contains no bits at all.
   - A *nonempty word*, which has to be of the form `(x:xs)`, where `x` is the initial bit and `xs` is a word comprising the rest of the bits.
   - Example: Suppose `xs = [p,q,r,s]`. Then {x:xs = [x,p,q,r,s]}.

   Attaching a singleton `x` onto a word `xs`

   ```
   (x:xs)
   ```

2. Concatenation

   Concatenating two words `x` and `y`

   ```
   x++y
   ```

   - If |w1 = [a,b,c]| and |w2 = [d,e,f,g]|, then
   - |w1++w2| = |[a,b,c,d,e,f,g]|

   You could also write |[a,b,c] ++ [d,e,f]| although that usually isn't very useful.

3. Accessing parts of words with a pattern

   - The left hand side is defined to have the value of the right hand side.

- When the lhs is a pattern, it defines the names in the pattern to be the corresponding elements of the rhs.
- This only works if the number of elements in the pattern matches the number of elements in the word

```
[a,b,c,d] = w
```

4. Indexing an element of a word

   - The `(!!)` operator indexes an element in a word, where the left-most element has index 0.
   - Example: |[a,b,c,d] !! 2| = |c|
   - The most significant element of |w| is |w!!0|.
   - There is also a "most significant bit" function; thus |msb w| is equivalent to |w!!0|
   - The least significant element is |w!!(length w - 1)|. There is also a "least significant bit" function; thus |lsb w| is equivalent to |w!!(length w - 1)|.

   Examples: the following are equivalent:

   ```
   lt_tc = mux2 (xy!!0) lt zero one lt
   lt_tc = mux2 (msb xy lt zero one lt
   ```

5. Big Endian and Little Endian

   - Notations are always somewhat arbitrary!
   - You can number bits in a word increasing from the left, or from the right; starting from 0, or starting from 1.
   - Hydra numbers the bit positions from left to right, starting from 0
   - This convention makes the formula for binary value of a word slightly more complicated—a one-time cost of 3 or 4 characters
   - But it makes many other expressions in a large circuit slightly simpler
   - Another common convention (*not* used in Hydra) is to mix the two notations in the same specification, and to let the reader guess which one is in effect everywhere

### 9.3.3 Deconstructing words

1. Fields

   - Usually the best way to extract part of a word is to use a , which is a part of a word.
   - Do this with the |field| function; again this is "meta notation", not a circuit.
   - |field w i k| gives the word consisting of the portion of |w| starting at index |i| and continuing for |k| elements.
   - It is an error if a field is specified that doesn't lie within the word.

   Here is a typical example: |ir| is the instruction register, a 16-bit word, and |$\text{ir}_{\text{sa}}$| is a field within an instruction.

   ```
   ir_sa = field ir 8 4
   ```

2. Patterns

   - Words are treated as lists, and you can use Haskell's list operations on them.
   - Normally it is better to use |(++)|, |(!!)|, and |field|.
   - The |(:)| operator is useful for defining "design patterns" tree-structured circuits (later!)
   - Suppose you have a word `w`, and you want to give a separate name to the first bit, and to all the rest.
   - Example: `w = [a,b,c,d,e]` and you want to use the name `x` to refer to the first bit (which is `a`) and `xs` to refer to the rest (which is `[b,c,d,e]`).

   This is achieved by writing:

   ```
   (x:xs) = w
   ```

3. Using list pattern notation

   - The left hand side `(x:xs)` is a *pattern* giving names to components of a structure

- Similarly a circuit input can be written as `(x:xs)`. (This is sometimes done in the recursive case of the definition of a design pattern.)

The parentheses in `(x:xs)` are just for grouping, so the : operator gets the right precedence.

## 9.4 Nested containers

- You can have a tuple containing two words |:: ([a], [a])|
- Or a word of tuples |:: [(a,a,a)]|
- Or arbitrarily deeply nested containers |:: ([(a,a)], a, ([a],a), [a])|

|rippleAdd4 Bit a => a -> [(a,a)] -> (a,[a])| has two input groups: a singleton bit (the carry input) and a word of pairs (the numbers to add). The output contains a singleton (carry output) and a word (the sum).

  - The type {|[(a,a)]|} is an important special case which is called {bit slice format}. The |rippleAdd4| type is an example of bit slice format.

## 9.5 Containers

In a physical circuit, every wire carries one bit, and doesn't have any relationship to any other wire (unless it is actually connected to that other wire). When we design a circuit, however, it takes several wires to carry any data value that isn't just a Boolean. For example, it takes 16 wires to transmit a 16-bit word, and to the designer there is definitely a clear relationship among these wires.

Circuits may contain large numbers of signals, and it would be tiresome to name them all. You can simplify the description of a circuit by defining **containers** that hold a collection of signals. Then you can use the container as a single object, without referring explicitly to its components.

A design is clearer if related signals together are grouped together, with a name for the entire collection. For example, we could give the name **x** to a 16-bit word, and just use **x** to refer to all the wires collectively.

Hydra provides two kinds of container: **tuples** and **words**. Tuples are useful for circuits that have multiple inputs and outputs; an example of a tuple is **(x, (a,b))**. Words are appropriate when several signals are used to represent a number, for example **[x0,x1,x2,x3]**.

Both kinds of container are written with several elements separated by commas. A quick way to tell them apart is that tuples use round parentheses **( ... )** but words use square brackets **[ ... ]**.

Containers are just notations that help to simplify the description of large circuits. If you look at the layout of a chip under a microscope, you won't see any tuples or words—just thousands of individual wires and components. A circuit specification that names each one explicitly would be long and unreadable; containers enable us to write compact and readable descriptions of such large circuits.

### 9.5.1 Tuples

Tuples provide the simplest way to give a single name to a bundle of signals.

Suppose we have a couple of signals named **a** and **b**. They can be collected together into a tuple by writing **(a,b)**. The elements are written inside round parentheses ( ... ) and separated by commas.

The elements of the tuple are expressions that describe signals. Any expression can be used; it doesn't have to be a signal name. For example, the tuple **(and2 x y, or2 x y)** is a tuple consisting of two signals, the outputs of two logic gates. In this example, the actual signals in the tuple don't have names.

A tuple can have any number of elements. Thus **(inv x, y, z)** is a 3-tuple and **(a,b,c,d)** is a 4-tuple.

If the basic signal type is **a**, as usual, then a 2-tuple has type **(a,a)**, a 3-tuple has type **(a,a,a)**, and so on. The type shows explicitly the number of elements.

One of the commonest ways to use a tuple is to describe a circuit that has several outputs. Indeed, there is no way to do this without using a cluster (a tuple or a word). Recall that the type of a circuit contains a number of arrows (**->**) and the type of the output comes after the last arrow. If there are actually several outputs, we need to combine them into a cluster and give the cluster's type as the type of the output.

Here is an example. Suppose we want to define a circuit that has two input bit signals, called **x** and **y**. The circuit produces two outputs, **and2 x y** as well as **or2 x y**. Let's name the circuit **aor2**. Here is a full definition:

```
aor :: Bit a => a -> a -> (a,a)
aor x y = (and2 x y, or2 x y)
```

The definition of **aor** consists of two parts: a type declaration (the line containing **::**), and a defining equation (thie line containing the =). In general, every circuit specification should contain these two parts.

Notice that there are two arrows (**->**) in the type. This means that there are two inputs, and each has type **a** — that is, each input is a bit signal. The type of the output comes after the last arrow, and it is **(a,a)**, so the output of the circuit is a tuple containing two bit signals.

The signal defining equations we have considered up to now have had a signal name on the left hand side: **x = ...** . In general, however, the left hand side of an equation is a **pattern**.

It is also possible to have an input cluster. The **aor** circuit above has two inputs, and these were treated separately: there are two arrows in the type, one after each input type. An alternative notation is to say that the circuit has just one input, which is a cluster containing two elements:

```
aorTup :: Bit a => (a,a) -> (a,a)
aorTup (x,y) = (and2 x y, or2 x y)
```

Compare the definitions of **aorTup** and **aor**. Both of them have two input bits named **x** and **y**, but they are organized differently. In **aor**, the inputs are treated as separate arguments, each of type **a**, and each followed by an arrow **->**. In **aorTup**, the input bits are collected together into the tuple **(x,y)** which has type **(a,a)**, and this tuple is the sole argument.

These two circuits, **aor** and **aorTup**, are essentially the same. They would look identical on a VLSI chip under the microscope. The only difference between them is the notation used to describe them.

There is an asymmetry in the notation. If a circuit has several inputs, there is a choice of notation: they can be treated as separate arguments, or they can be collected together into a tuple. However, if a circuit has several outputs, there is no choice: they **must** be collected together into a tuple.

This notation for types, with the arrows and the (apparently) different treatment of circuit inputs and outputs, may look strange and counterintuitive. There is actually a very good reason the type notation is designed this way, but it involves some techniques we are not ready to discuss yet (see the chapter on design patterns).

There are other uses for tuples besides just handling circuits with multiple outputs. Sometimes tuples are useful just for cutting some of the boilerplate in a specification, making it shorter and easier to read. Suppose we have a circuit where two signals, say **x** and **y**, are needed as inputs to several other building block circuits **f1**, **f2**, and **f3**. We could write the specification with all the signals written out explicitly:

```
circ :: Bit a => a -> a -> a
circ x y = z
  where
     p = f1 x y
     q = f2 x y
     r = f3 x y
     z = xor3 p q r
```

But we might be able to simplify this by changing the types of **circ**, **f1**, **f2**, and **f3** to collect **x** and **y** into a tuple.

```
circ :: Bit a => (a,a) -> a
circ xy = z
  where
     p = f1 xy
     q = f2 xy
     r = f3 xy
     z = xor3 p q r
```

In a large and complicated system, this technique can make a big difference. For example, in a processor circuit there may be a number of signals needed to control the arithmetic-logic unit, and those signals travel together. It can cut down on the notation significantly just to combine them into a tuple, give the tuple a name, and pass around the whole cluster without mentioning the individual components.

Sometimes you may have a cluster, but you need to extract its elements and give them individual names. This can be done in a circuit black box definition using a signal defining equation. For example, the following equation defines **alpha** and **beta** to be the names of the elements of a tuple named **pair**:

```
(alpha,beta) = pair
```

Tuples can be nested. For example, **(p, (x,y,z))** is a 2-tuple (**not** a 4-tuple!). The first element is **p**, and the second element is a 3-tuple **(x,y,z)**. The type is

```
(p, (x,y,z)) :: (a, (a,a,a))
```

This example shows a crucial property of tuples: their elements may have different types; in this case the type of the first element is **a** and the type of the second element is **(a,a,a)** and those types are different, just as a physical wire is not the same thing as a bundle of three physical wires.

Why use a tuple type like **(a,(a,a,a))** when a simple 4-tuple would seem simpler? The reason is that sometimes, in larger systems, a sub-circuit produces many outputs, and groups of them will then be connected to different destinations. The notation to describe this is simpler if the tuple structure matches the logical organization of the circuit. We will see several examples of this, especially in the design of processors.

It is also possible to have two different signal representations in a specification. Each one needs its own distinct type variable name. For example, suppose we are designing a circuit that has a basic bit signal type **a**, but the circuit also has some values where we aren't concerned about the bit representation (floating point numbers, perhaps). To abstract away from the bit representation, we could give another type **b** to these abstract values. Then a black box circuit that outputs both a bit and a floating point number would have the output type **(a,b)**.

### 9.5.2 Words

There are two kinds of cluster that allow several signals to be collected together into one entity. The previous section discussed tuples, and now we introduce words. Tuples allow arbitrary groupings, while words have a regular structure and their elements can be accessed by indexing. Words are frequently used for collections of bits that represent binary numbers.

In a word, bit indices are 0, 1, . . . , n-1 where bit 0 is most significant. The expression **[x0,x1,x2,x3]** denotes a word containing the individual signals **x0**, . . . , **x3**. The syntax is similar to a tuple; the difference is that an expression for a word uses square brackets **[** \ **]** while a tuple uses round parentheses **(** \ **)**.

The basic usage of a word is similar to a tuple. For example, a circuit could collect several signals into a word and output that. Here is an alternative definition of the half adder:

```
halfAddw :: Bit a -> a -> a -> [a]
halfAddw x y = [c,s]
  where
    c = and2 x y
    s = xor2 x y
```

There two differences between this definition and the one given earlier. First the output expression here is **[c,s]**, so it's a word, while the output expression given for the original **halfAdd** is **(c,s)**, which is a tuple. The other difference is quite important: the output type is **[a]**, rather than **(a,a)** for the original **halfAdd**.

All the elements of a word must have the same type. If this type is **a**, then the word has type **[a]**. The type of a word doesn't specify how many elements the word contains. This is different from a tuple, where **(a,a)** contains exactly two elements, and **(a,a,a,a)** contains exactly four elements.

Each element of a word has an index, a natural number that gives its position within the word. You can think of a word as an array or vector. The index of the leftmost position is 0, and the index of the rightmost position is **k-1**, where **k** is the length of the word. If we have defined some bit signals **x0**, **x1**, **x2**, and **x3**, then we could define a word **x** of these bits with the equation

```
w = [x0,x1,x2,x3]
```

There are actually two conventions commonly used in computer systems. One convention starts with position 0 at the left end, and counts up going to the right. This is called **big Endian** notation. The other convention, naturally called **little Endian**, starts with 0 as the index of the rightmost element, and the indices count up going to the left.

```
[x0,x1,x2,x3]   -- Big Endian convention
[x3,x2,x1,x0]   -- Little Endian convention
```

As you might imagine, neither convention is fundamentally better than the other, but there are all sorts of minor issues that might cause one to be preferred over the other. Hydra allows both conventions, but in this book we will stick to Big Endian consistently.

There seems to be a phenomenon in computer systems, where the less significant an issue is, the more heated debate there is about it. This phenomenon was actually the inspiration for the odd names Big/Little Endian. The names come from Gullivers Travels, by Jonathan Swift, where the citizens of the kingdom of Blefuscu open their eggs at the big end, while the citizens of Lilliput open their eggs at the little end. The application of this story to computer systems comes from an article by Danny Cohen, "On Holy Wars and a Plea for Peach" (IEEE Computer, October 1981).

The point here (aside from an entertaining digression) is that having a standard is a good idea, and arguments for one particular choice are less compelling than having a consistent standard. Nevertheless, there is one situation in hardware description where Little Endian is slightly more convenient than Big Endian (see ref????) and some authors actually combine both conventions. The confusion isn't worth it!

The size or length of a word is the number of elements it contains. If a word contains $k$ elements, then their indices range from 0 to $k-1$. Hydra provides a meta-function **length** that takes a word and returns an integer giving its size.

```
length :: [a] -> Int
```

For example, **length [x0,x1,x2] = 3**. With just the parts of Hydra covered so far, there is no way to use the length of a word, but later we will encounter some more powerful features where an algorithm will generate a circuit of a given size, and then the **length** function will be useful. It's important to remember that **length** is not a circuit; it is part of the notation used to describe circuits.

There are several notations and operators that can be used to build words from signals, and for extracting the signals within a word. The following sections introduce these notations, and then a couple of example circuits will be presented.

### 9.5.3 A circuit with words and internal signals

Files: **Add4.hs** and **Add4Run.hs**

The **add4** circuit takes two 4-bit binary numbers **x** and **y**, and a carry input **c**. It adds them and outputs a carry output bit and a 4 bit sum. The circuit is defined in **Add4.hs**. A main program containing test data and a simulation driver is in **Add4Run.hs**. To run the simulation, enter **ghc -e main Add4Run**. Here is the output:

1. Building words

   If you have expressions that define some signals, a word comprising the signals can be constructed by writing the expressions in square brackets, separated by commas.

   ```
   [p,q,r,s]
   ```

   The length of a word can be any natural number. Thus **[]** is the empty word, **[x]** is a word containing just one element, and so on.

```
[]                            -- length = 0
[x]                           -- length = 1
[x,y]                         -- length = 2
[x0,x1,x2,x3,x4,x5,x6,x7]    -- length = 8
```

Suppose you have a word **w**, of any length, and a bit signal **x**. Thus **w :: [a]** and **x :: a**, where **a** is the basic signal type. Then we can construct a new word which is just like **w** except that the singleton **x** is attached to the front. The notation for this is **x:w**, which is pronounced "*x cons w*". For example, suppose **w** = **[p,q,r,s]**. Then **(x:w)** = **[x,p,q,r,s]**. The properties of the **(:)** operator are summarized as follows:

```
x :: a
w :: [a]
(x:w) :: [a]
length (x:w) = 1 + length w
```

It's often useful to take two words that have already been defined, and to define a bigger one that contains the elements of both. This is called **append** or **concatenation**, and is done using the **(++)** operator. The word **w1 ++ w2** is a word containing first the elements of **w1**, and then the elements of **w2**. Here are some examples and properties of append:

```
[x0,x1,x2,x3] ++ [y0,y1] = [x0,x1,x2,x3,y0,y1]
length (w1 ++ w2) = length w1 + length w2
```

2. Accessing parts of a word

   Often we can perform operations on entire words, using word-oriented digital circuits, without ever accessing individual elements of a word. Later we will see a family of building block circuits that operate on words. Normally this is the best way to organize a circuit that works with words.

   Sometimes, however, it's necessary to extract one or more elements of a word. One way to do this is by **indexing**. Each element of a word **w** has an index, ranging from 0 to $k-1$, where **k** = **length w**. The **(!!)** operator uses an index to extract the element; thus **w!!i** gives the $i$th element of the word **w**. This is well defined if the index **i** is in range: $i \leq \mathit{length}\ w$. If $i < 0$, or $i \geq \mathit{length}\ W$, then **w!!i** is an error.

```
w!!i                    i'th bit of word w
field w i j             bits i..i+j-1 of word w
```

There are two special cases for indexing that are supported by specific operators: you can get the least significant (or most significant) bit of a word **w** using **lsb w** (or **msb w**). The least significant bit **lsb w** is equivalent to **w !! (length w -1)**, and the most significant bit **msb w** is equivalent to **w !! 0**.

```
w !! i                  (!!) :: [a] -> Int -> a
lsb w                   lsb :: [a] -> a
msb w                   msb :: [a] -> a
```

There are three functions that give a field from a word; that is, the result is itself a (smaller) word, not just an individual bit. The **take** and **drop** functions give a sub-word that is at the beginning or end of a word. Thus **take i w** gives a word consisting of the leftmost *i* elements of **w**, while **drop i w** gives a word consisting of all the elements of **w** **except for** the leftmost *i* elements.

More generally, it is sometimes necessary to extract an arbitrary field from a word. A **field** is a word consisting of any consecutive set of elements. A field has type **Field**, and it consists of a pair of integers **(i,s)** where **i** is the index of the starting position of the field, and **s** is its size. Thus **field (i,s) w = [w!!i, w!!(i+1), . . . , w!!(i+s-1)]**.

```
type Field = (Int,Int)
take i w                 take :: Int -> [a] -> [a]
drop i w                 drop :: Int -> [a] -> [a]
field f w                field :: Field -> [a] -> [a]
```

An example of a circuit that operates on words is the 4-bit word inverter **inv4**. Its input and output are both 4-bit words, and each output bit is the inversion of the corresponding input bit. The type notation for the word is concise, since the types of the individual bits don't have to be repeated, but on the other hand the type doesn't express the fact that this circuit works only on 4-bit words.

```
inv4 :: Bit a => [a] -> [a]
inv4 [x0,x1,x2,x3] = [inv x0, inv x1, inv x2, inv x3]
```

The circuit specification for **inv4** is simple enough, but it would be painful to extend this to much larger sizes, say 64-bit words. The chapter on design patterns shows a more elegant approach, but for small words the style used here is adequate. The Hydra libraries provide a collection of straightforward circuit specifications written in the same style as **inv4**, and they also provide circuits that are defined using design patterns and that work for arbitrary word sizes, no matter how large.

### 9.5.4 Nested clusters

The cluster types can be nested. A tuple may contain words (or deeper tuples), and a word may contain tuples (or deeper words, although that is unusual).

There is a style of circuit design called **bit slice organization**. The idea is that a building block circuit is defined for an arbitrary position within a word, and these building blocks can then be combined. Bit slice style often results in complex groupings, with words of tuples, and notwithstanding the relatively complex types it can result in simple specifications of efficient circuits. The essence of bit slice organization is to keep the corresponding bits of several words together. Thus two words $x$ and $y$ could be represented as a word of pairs, rather than two separate words:

$[(x0, y0), (x1, y1), (x2, y2), (x3, y3)] :: [(a, a)]$

Collecting a group of signals into a cluster is just a notational convenience; it doesn't affect the actual circuit. However, grouping can simplify the way you **describe** the circuit, and this is essential for large and complex circuits.

When you are designing a circuit with several input signals, you can decide whether to treat them as separate arguments (each followed by an arrow **->**) or as a single argument which is a tuple or word. However, if you are using a circuit that has already been specified, you need to follow the type used in its specification.

When a circuit has several outputs, there is no choice—the output signals must be collected into a tuple or a word. The reason for this is that the underlying functional language requires that each function has one result. This does not limit our ability to express complex circuits; it simply means that we need to use tuples or words.

Grouping is often helpful just to simplify the notation and to make specifications more readable.

A tuple (x, (a,b)) is used to collect several values which may be unrelated to each other. Tuples are used for groups where the components are unrelated, and indexing doesn't make sense. The components may have different types: $(a,(a,a),a)$ A word is used to collect values that belong to specific bit positions, typically to form a binary number. Tuples and words can be combined to form complex clusters.

Example: a 4-Bit ripple carry adder

For the multiplexer (the hardware equivalent of an if-then-else) there is little to gain by grouping the inputs, so we use separate parameters without grouping: *mux1 c x y = … *

For the full-adder, which adds three bits, it's convenient to group the bits $x$ and $y$ from the $i$th position in a word together, and to keep them separate from the carry input bit $c$. *fullAdd (x,y) c = … *

Don't worry—the reasons for these decisions will become clear later, when we start making advanced uses of these circuits. It's common to make some changes to the grouping notation for a circuit after you start using it extensively!

```
rippleAdd4 c [(x0,y0), (x1,y1), (x2,y2), (x3,y3)] =
    (c0, [s0,s1,s2,s3])
  where
    (c0,s0) = fullAdd c1 (x0,y0)
    (c1,s1) = fullAdd c2 (x1,y1)
    (c2,s2) = fullAdd c3 (x2,y2)
    (c3,s3) = fullAdd c  (x3,y3)
```

**Exercise.** A circuit has the type declaration **circ :: Bit a => a -> (a,a) -> [a] -> (a,[a])**. How many groups of input bits are there? How are they structured? How is the output structured?

**Exercise.** Modify the definition of **rippleAdd4** to handle 6-bit words.

**Exercise.** Define an 8-bit adder, named **rippleAdd8**. Don't follow the pattern of **rippleAdd4**, with eight equations. Instead, use **rippleAdd4** as a building block circuit. In your definition of **rippleAdd8**, use two separate internal **rippleAdd4** circuits, and connect them up appropriately.

**Exercise.** Suppose **x = [x0,x1,x2]**, **y = [y0,y1,y2,y3]**, and **z = x++y**. What are the values of **z**, **length z**, and **z!!4**?

## 10 Simulation drivers with format and state

This section shows several examples of circuits of increasing complexity. You should be able to design and simulate some circuits on your own by following and modifying these examples. The various design techniques are described in more detail in later sections. For now, just run the examples, and refer back to them as the subsequent sections explain the language. See the examples directory for a collection of circuits.

To run a circuit, two definitions are needed: a circuit specification, and a simulation driver. The circuit specification states precisely the interface to the circuit, what components it contains, and how they are connected. The simulation driver describes how to provide inputs using a readable input format, and how to format the outputs to make them readable. For example, Reg1.hs defines a 1-bit register circuit, and Reg1Run.hs is the simulation driver which handles conversion between readable notation and internal signals.

It's good practice to place the circuit definition and its simulation driver in separate files. By convention, the filename of the driver ends in "Run".

## 10.1 Automatic output and formatted output

You can choose between automatic or manual formatting of the simulation results.

- If a format statement is present, the automatic output is disabled and the outport is produced entirely by the format. However, the system will indicate the clock cycles.
- If the format is omitted, then the system will generate output using the port definitions. In this case, outport definitions are required.

A simulation driver must either define the outports, or it must contain a format. If it contains both, the outports are ignored. So if you wish to provide a format there is no need to define the outports.

For small circuits, it may be easiest to define the outports and use the automatic output. For large circuits, you can make the results more compact and more readable by defining a format.

## 10.2 Feedback and changing state: BSR4

Files: **BSR4.hs** and **BSR4Run.hs**

A bidirectional shift register

Define a shift register that takes an operation code op and data inputs x, li, ri, and performs an a state change depending on op:

- op=0 – no state change

- op=1 – load input word x
- op=2 – shift right
- op=3 – shift left

The circuit uses a building block srb ("shift register block") which has an internal state to hold the bit in that position in the word. The inputs to an srb are an input from the left (for shifting to the right), an input from the right (for shifting to the left), and a bit input from the word x (for loading a word). The circuit outputs a triple: the left and right outputs, and the word giving the current state of the register. (Minor point: the left and right outputs aren't essential, as they also appear as the most and least significant bits of the word output, but this approach makes it easier to connect several sr4 circuits together, and it also fits well with the definition of the more general sr circuit below.)

The structure of the 4-bit version comes directly from the data dependencies.

The shift register block uses a dff to hold the state, and it uses a mux2 to determine the new value of the state. This is either the old value, the data bit x from a load, or the input from the left or right in case of a shift.

examples/shift/BSR4.hs examples/shift/BSR4Run.hs

The test data and simulation driver are defined in **BSR4Run.hs**. Running the circuit produces this:

```
$ ghc -e main BSR4Run
op=01 l=0 r=0 x=9   Output lo=0 ro=0 y=0
op=00 l=0 r=0 x=0   Output lo=1 ro=1 y=9
op=11 l=0 r=0 x=0   Output lo=1 ro=1 y=9
op=11 l=0 r=1 x=0   Output lo=0 ro=0 y=2
op=00 l=0 r=0 x=0   Output lo=0 ro=1 y=5
op=01 l=0 r=0 x=4   Output lo=0 ro=1 y=5
op=10 l=1 r=0 x=0   Output lo=0 ro=0 y=4
op=10 l=0 r=0 x=0   Output lo=1 ro=0 y=a
op=10 l=0 r=0 x=0   Output lo=0 ro=1 y=5
op=10 l=1 r=0 x=0   Output lo=0 ro=0 y=2
op=00 l=0 r=0 x=0   Output lo=1 ro=1 y=9
```

## 11 Records with named fields

When a circuit has a small number of inputs or outputs, it's straightforward to provide them in a fixed order. For example, here is the definition of a multiplexer:

```
mux1 :: Signal 1 => a -> a -> a -> a
mux1 c x y = or2 (and2 (inv c) x)
                 (and2 c y)
```

To use it, you need to know that the first input `c` is the control, that the input `x` is output if `c` is zero, and the input `y` is output if `c` is one.

However, complex circuits may have too many inputs and outputs to be able to remember all their positions in a list of ports. Even if you remember that the signal you want is the 19th one, it's awkward to get access to it. Some modern chips have thousands of input and output signals.

### 11.1 Defining a group of signals

The solution is to identify the signals by name and to collect a group of named signals in a *record*. Here is a definition from the Datapath module, which provides a number of output signals that are collected into a record with type `DPoutputs a`:

```
data DPoutputs a = DPoutputs
     { aluOutputs :: ([a], [a])
     , r :: [a]        -- alu output
     , ccnew :: [a]     -- alu output
     , ma :: [a]
...
     }
```

The Datapath module defines the signals: it contains an equation for each element of the record giving the value.

```
...
aluOutputs = ...
r = ...
ccnew = ...
...
```

## 11.2 Defining a group of signals

The record itself is named `dp` and is defined by this equation:

```
dp = DPoutputs {..}
```

The notation `{..}` means that each defined value whose name is listed in the record type (`DPoutputs`) should be included in the actual record (`dp`). For example, the record type `DPoutputs a` defines a field named `ccnew` and there is an equation defining the value `ccnew`, so `ccnew` is included in the record `dp`.

The definition of the datapath circuit is an equation where the right hand side says that the output is `dp`, which is defined to be the record of signals `dp`:

```
datapath (CtlSig {..}) (SysIO {..}) memdat = dp
  where

-- Interface
    dp = DPoutputs {..}
```

The first input to the datapath circuit is `(CtlSig {..})`, which is a record of signals output by the control circuit. When used as an input, this `{..}` notation means that every field in the `CtlSig` record defines the corresponding name. With this notation, you don't need to write a separate equation for every element of the `CtlSig` record.

The `{..}` notation makes it easy to add a new signal to a record. For example, suppose you modify the datapath to contain two new signals: `pqr` which is a bit, and `xyz` which is a word of bits. The following changes are required:

- Add equations defining the new signals to the `Datapath` module:

```
pqr = ...
xyz = ...
```

- Add the new signals to the record definition in the `Interface` module:

```
data DPoutputs a = DPoutputs
  { ...
  , pqr :: a
  , xyz :: [a]
  ...
}
```

- Now you can use `pqr` and `xyz` in any module that receives the `DPoutputs`.

## 12 Circuit generators

- Often we won't describe every component "by hand"
- We can use *circuit generators* — algorithms that create the circuit
- The simplest kind is a *generic circuit* which works for any word size
  - For now we'll look at some examples of generic circuits defined using a circuit generator
  - Later we'll look at how to use the generators
  - And even later, we'll see how to define new circuit generators.

### 12.1 Inverting all the bits in a word

Word inverter: |winv4| takes a word and inverts each of its bits

```
winv4 :: Bit a => [a] -> [a]
winv4 [a,b,c,d] = [inv a, inv b, inv c, inv d]
```

- You can use {|winv4 x|} if |x| is a 4-bit word, but not if it's an 8 or 16 bit word.
- The generic circuit {|winv|} works for any word size. It's defined using a {circuit generator} called {map}.

```
winv :: Bit a => [a] -> [a]
winv = map inv
```

### 12.2 How does a circuit generator work?

- winv4 is a 4-bit word inverter
- winv generates an $n$-bit word inverter: it works for *every word size*
- The circuit generator works by (1) measuring the size of the input word and (2) generating the required number of components
- {This is not software to do the operation — it is still a real circuit.}

## 12.3 Calculating the and/or of many bits

Take a word, and and/or all its bits together

```
andw4 [a,b,c,d] = and2 (and2 a b) (and2 c d)
```

For bigger words, it's better to use generic circuits andw, orw that are defined using the circuit generator {foldl}:

```
andw, orw :: Bit a => [a] -> a
andw = foldl and2 one
orw = foldl or2 zero
```

The input is a word of bits, and the output is the and/or of all those bits. This is like an $n$-bit or gate with an $n$-bit input word.

## 12.4 4-bit ripple carry adder

```
rippleAdd4 :: Bit a => a -> [(a,a)] -> (a,[a])
rippleAdd4 cin [(x0,y0),(x1,y1),(x2,y2),(x3,y3)]
  = (c0, [s0,s1,s2,s3])
  where
    (c0,s0) = fullAdd (x0,y0) c1
    (c1,s1) = fullAdd (x1,y1) c2
    (c2,s2) = fullAdd (x2,y2) c3
    (c3,s3) = fullAdd (x3,y3) cin
```

In Sigma16 we need a 16-bit adder. Many current machines contain 64-bit adders. This would be painful to write in the style of

$$\text{rippleAdd4} \quad .$$

## 12.5 Ripple carry adder schematic

## 12.6 Generic $n$-bit ripple carry adder

- The `rippleAdd` generic circuit is similar to `rippleAdd4`, except it works on all word sizes.
- It's defined using a very powerful generator called {mscanr}
- At circuit generation time, the generator measures the sizes of its input words, and then it constructs a circuit with the right number of full adders, and automatically does all the wiring.

- Notice that |rippleAdd| is equivalent to an infinite set of definitions |rippleAdd0|, |rippleAdd1|, |rippleAdd2|, |rippleAdd3|,

$$\text{rippleAdd4} \quad , \ldots, \quad \text{rippleAdd64} \quad , \ldots,$$

- Yet the definition of |rippleAdd| is far simpler than e.g. |rippleAdd4|.
- This is due to the power of the generator |mscanr|.

```
rippleAdd :: Bit a => a -> [(a,a)] -> (a,[a])
rippleAdd = mscanr fullAdd
```

## 12.7 "Times"

- Programmers talk about two "times" involved in running a program:
  - Compile time
  - Run time
- This makes a real difference: an error message at compile time is very different from a crash at run time
- Hydra has four "times"
  - Model transformation time
  - Circuit generation time
  - Circuit compilation time
  - Runtime (simulator execution time)
  - And to get real hardware, there is
    - Netlist and layout generation time
    - Fabrication time
    - Running the hardware circuit

## 12.8 Circuit generators

We will generally specify large circuits using a circuit generator, not by drawing every component individually. There are two kinds of circuit generator. Design patterns (higher order functions) are the focus of this chapter. Special languages for special kinds of circuit (e.g. control algorithms) are covered later.

Design patterns use circuits as building blocks

Design patterns are **higher order** functions: they take one or more **circuit specifications** as parameters. The pattern defines how to connect up these given circuits in a regular pattern. A pattern definition looks just like an ordinary circuit specification, except It uses recursion to decompose groups of signals. It uses abstract circuits, supplied as parameters, instead of specific circuits. Its type may include building block circuits (these parameters contain an **->** in their type) and/or size parameters (with a type like **Int**).

### 12.8.1 Map

Word inverter: winv takes a word and inverts each of its bits

```
winv :: Bit a => [a] -> [a]
winv x = map inv x
```

Operating on each element of a word of known size: mapn

```
wlatch :: CBit a => Int -> [a] -> [a]
wlatch k x = mapn dff k x
```

The word register

```
reg
  :: CBit a =>
  Int              -- ** k = the word size
  -> a           -- ** ld = the load control signal
  -> [a]         -- ** input word of size k
  -> [a]         -- ** output is the register state
reg k ld x = mapn (reg1 ld) k x
```

Mapping a circuit with multiple inputs

```
mux1w :: Bit a => a -> [a] -> [a] -> [a]
mux1w c x y = map2 (mux1 c) x y

mux2w cc = map4 (mux2 cc)
```

1. The map combinator

   Sometimes you have a circuit (it's arbitrary, so call it f) that takes an input (say it has type a) and produces an output (call its type b). You need to take a word of signals, and process each one with the circuit f. For example, **inv4** processes each signal with an **inv**. The **map** pattern describes this in general.

```
map :: (a->b) -> [a] -> [b]
```

- The first argument to the pattern is a circuit with type **a->b**
- The pattern then generates a circuits, which takes an input word of type **[a]** and produces an output word of type **[b]**.

Example of map

We can define a word inverter using the pattern that places an inverter on each input signal, to produce the corresponding output signals.

```
winv :: Bit a => [a] -> [a]
winv x = map inv x
```

Technical note: in a defining equation of the form **f a b c = g c**, you can "factor out" the rightmost parameter from both sides, giving a slightly shorter form.

```
winv :: Bit a => [a] -> [a]
winv = map inv
```

This is attractive because it describes just the pattern.

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

This definition is a recursion, based on the word structure of the input.

The base case is an empty input word **[]**. In this case, the output is also empty.

The recursion (or induction) case has an input word **x:xs** consisting of an initial bit **x** followed by the rest of the word, **xs**. The circuit introduces a copy of the **f** circuit to process **x**, and handles the rest recursively.

Extending map to multiple inputs

The **map2** pattern is similar to **map**, but it uses a circuit that takes two inputs (thus its type is **a->b->c**). Note that **map2** is **not** a bit-slice pattern; it uses separate words.

```
map2 :: (a->b->c) -> [a] -> [b] -> [c]
```

We can extend the basic multiplexor to handle words:

```
mux1w :: Bit a => a -> [a] -> [a] -> [a]
mux1w c x y = map2 (mux1 c) x y
```

2. Sized map

   The **mapn** pattern is similar to **map**, except it takes a size parameter, and guarantees to produce an output of that size.

   ```
   mapn :: (a->b) -> Int -> [a] -> [b]
   ```

   Registers are defined using **mapn**, to ensure that the number of flip flops is defined Combinational circuits may be defined using **map**, so they inherit the word size of their input

### 12.8.2 Fold

The folding patterns define a linear circuit structure.

There is an input word of type **[b]**.

The elements of the word are combined using a building block **f**.

There is a "horizontal" signal of some type (call it **a**), which is goes across the word from left to right.

An initial horizontal input, of type **a** is provided.

The output is the final horizontal output (produced by the rightmost **f** circuit).

Folding corresponds to a linear computation from one end of the word to the other, starting with an initial value a (sometimes called an accumulator, but this is not to be confused with accumulator registers!).

1. Fold from the left

   In general, a fold can proceed either direction across the word. The **foldl** pattern describes a **fold from the left**; i.e. the information flows from left to right across the word.

```
foldl :: (a->b->a) -> a -> [b] -> a
```

The pattern is defined recursively:

```
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

xamples: orw, andw

The **orw** circuit determines whether there is any 1 bit in a word.

```
orw :: Bit a => [a] -> a
orw = foldl or2 zero
```

The **andw** circuit determines whether all the bits in a word are 1.

And/Or over a word

```
orw, andw :: Bit a => [a] -> a
orw = foldl or2 zero
andw = foldl and2 one
```

The time required (the path depth) is linear in the word size. There are also tree-structured patterns that can do these computations in logarithmic time.

Efficiency

The definitions of **orw** and **andw** are not very efficient

If a large number of signals are being combined, a tree structure of logic gates reduces the path depth. If this circuit is on the critical path, that will help. If the technology supplies 3 or 4 input gates, it would likely be faster to use some of those, rather than just the 2 input gates. The **foldl** pattern uses one extra gate to include the "default" value of zero or one. This is overhead.

This inefficiency is not a concern, because

- There are alternative patterns that generate more efficient circuits
- A circuit optimiser can generate optimal results If the circuit isn't on the critical path, it makes no difference anyway.

2. Binary comparison using foldl

   The problem: input two binary numbers, in bit slice form: **[(x0,y0), (x1,y1), . . . , (xk,yk)]** Output the result of a comparision: **(lt,eq,gt)**, giving the values of ($x < y, x = y, x > y$). Exactly one of the three output bits must be 1. Idea: start from left, assuming the numbers are equal so far: **(0,1,0)**. Move over the columns from left to right, updating the results of the comparision using a building block circuit **cmp1**. Going from left to right, once we have established either $<$ or $>$, that result will never change. If the current result is $=$ and $x = y$, it's still $=$. If the current result is $=$ but $x$ and $y$ are different, the new result becomes $<$ or $>$.

   A bit comparison building block circuit:

```
cmp1 :: Bit a => (a,a,a) -> (a,a) -> (a,a,a)
cmp1 (lt,eq,gt) (x,y) =
  (or2 lt (and3 eq (inv x) y),
   and2 eq (inv (xor2 x y)),
   or2 gt (and3 eq x (inv y))
  )
```

   The ripple comparison circuit is defined simply using the pattern:

```
rippleCmp :: Bit a => [(a,a)] -> (a,a,a)
rippleCmp = foldl cmp1 (zero,one,zero)
```

3. Fold from the right: foldr

   You can also run a fold across a word from the right to the left.

```
foldr :: (b->a->a) -> a -> [b] -> a
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

   This is symmetric with **foldl**.

### 12.8.3 Circuit generators 2

1. lec18 Circuit Generators

   (a) Circuit Generators

i. Abstraction

   We will generally specify large circuits using a circuit generator, not by drawing every component individually.

   - Design patterns (higher order functions)
   - Special languages for special kinds of circuit (e.g. control algorithms)

ii. Design patterns: Circuits as building blocks

   - Design patterns are *higher order* functions: they take one or more *circuit specifications* as parameters.
   - The pattern defines how to connect up these given circuits in a regular pattern.
   - A pattern definition looks just like an ordinary circuit specification, except
     - It uses recursion to decompose groups of signals.
     - It uses abstract circuits, supplied as parameters, instead of specific circuits.
     - Its type may include building block circuits (these parameters contain an `->` in their type) and/or size parameters (with a type like `Int`).

iii. Map

   A. A circuit with a regular pattern: word inverter

   ```
   winv4 :: Bit a => [a] -> [a]
   winv4 [x0,x1,x2,x3] = [inv x0, inv x1, inv x2, inv x3]
   ```

   - Definition is simple
   - But it works only for one word size, and only for inverters
   - Extending it to larger word size is tedious and error-prone

   B. Map — expressing the regular pattern directly

   - Sometimes you have a circuit (it's arbitrary, so call it $f$) that takes an input (say it has type $a$) and produces an output (call its type $b$).
   - You need to take a word of signals, and process each one with the circuit $f$. For example, `inv4` processes each signal with an `inv`.
   - The `map` pattern describes this in general.

   ```
   map :: (a->b) -> [a] -> [b]
   ```

- The first argument to the pattern is a circuit with type `a->b`
- The pattern generates a circuit that takes an input word of type `[a]` and produces an output word of type `[b]`.

C. Example of map

We can define a word inverter using the pattern that places an inverter on each input signal, to produce the corresponding output signals.

```
winv :: Bit a => [a] -> [a]
winv x = map inv x
```

Technical note: in a defining equation of the form {f a b c = g c}, you can "factor out" the rightmost parameter from both sides, giving a slightly shorter form.

```
winv :: Bit a => [a] -> [a]
winv = map inv
```

This is attractive because it describes just the pattern.

D. Word inverter: ys = map inv xs

E. Definition of map

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- A recursion, based on the word structure of the input.
- The base case is an empty input word `[]`. In this case, the output is also empty.
- The recursion (or induction) case has an input word `x:xs` consisting of an initial bit `x` followed by the rest of the word, `xs`. The circuit introduces a copy of the `f` circuit to process `x`, and handles the rest recursively.

F. Structure of map recursion

G. After the recursion has completed

H. Extending map to multiple inputs

The `map2` pattern is similar to `map`, but it uses a circuit that takes two inputs (thus its type is `a->b->c`). Note that `map2` is *not* a bit-slice pattern; it uses separate words.

```
map2 :: (a->b->c) -> [a] -> [b] -> [c]
```

We can extend the basic multiplexor to handle words: