# FPGA design with CλaSH

Conrad Parker <conrad@metadecks.org>

fp-syd, February 2016

## FPGA

- Essentially a bunch of logic gates and memory blocks that can be told to connect up to be whatever digital circuit you want it to be.
- You load a binary file onto it that describes that configuration.
- You generate that binary file using an application like Vivado, from source code that you wrote in a language like **VHDL**.
- At this point we're back in software land – we have a programming language and some tool that can compile that down to a binary.
- Flash it, reboot, and you’re executing on hardware!

# VHDL

Unfortunately, VHDL is fairly tedious to develop:

- It is more low-level than assembly, in that you are describing circuits of logic gates like AND and OR and NOT and XOR, and worrying about clock timings and which values need to be latched between stages of some pipeline you're inventing.
- You don't necessarily have luxuries like floating point numbers, and the tools can be flaky.

On the flip-side it is possible to create very efficient circuits: you can optimize for space by using fewer bits, and you can optimize for time by doing more things in parallel.

## Clash

Clash is based on Haskell, with features of suitable for hardware design such as length-typed bit vectors (the length is part of the type declaration), primitive data flow strategies like Mealy and Moore machines, and parallel reducers.

For example, the expression:

```
fold ( + ) vec
```

will construct a depth log n tree of adders, to sum the contents of vec.

## Clash

- There is "poor support" for recursive functions, meaning that the recursion depth is explicitly limited.
- Other than that, most commonly used Haskell features seem to be supported.

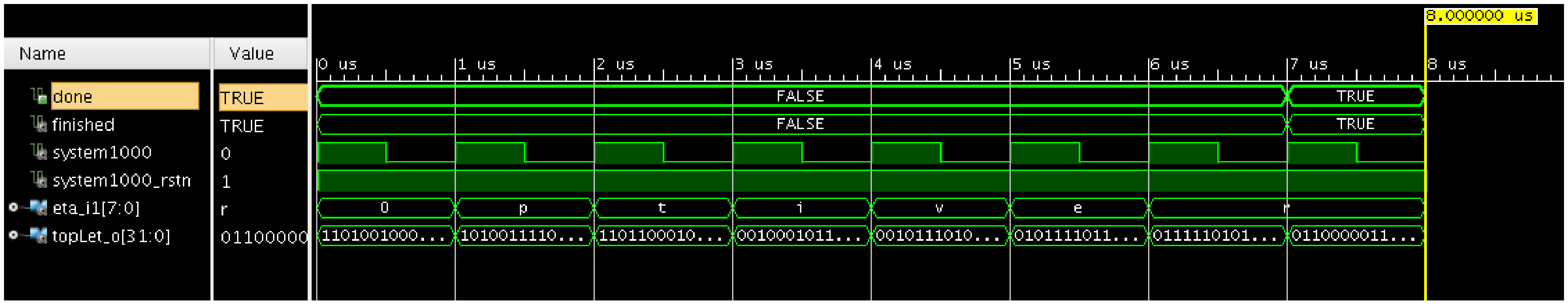
- Haskell seems well-suited to designing parallel systems, as the language simply describes dependencies between expressions without specifying an explicit order of evaluation.

# CRC32

In this evaluation, I implemented and tested CRC32.

- I wrote **32 lines of Haskell**
- ... which included 2 one-liners to describe some input and expected output for testing.
- From that, Clash **generated 742 lines** (11 files) **of VHDL**
- ... **including** 394 lines (5 files) for **the test bench** (which you would normally write by hand ...)
- Vivado uses the VHDL to make a fairly simple circuit (the **elaborated design**)
- ... and **synthesis** creates an optimal circuit layout.

- The test bench result, from Vivado's hardware simulator, includes a boolean "done" signal which switches to TRUE when the test passes.



| Name | Value |
|---|---|
| done | TRUE |
| finished | TRUE |
| system1000 | 0 |
| system1000_rstn | 1 |
| eta_i1[7:0] | r |
| topLet_o[31:0] | 01100000 |

# Test data

First we generate some test data, using a known-good implementation (the crc32() function in zlib).

We use C to print the CRC32 for each successive letter of the input string.

Running this, we get the expected output for our test data:

```
$ gcc -std=c11 CRC32.c -o CRC32 -lz
$ ./CRC32 Optiver
3523407757, 2814004990, 3634756580, 584440114, 781824552,
1593772748, 2102791115,
```

```c
#include <stdio.h>
#include <zlib.h>

unsigned long generate_crc32(const unsigned char * buf,
    unsigned int len)
{
    unsigned long crc_init = crc32(0L, Z_NULL, 0);
    unsigned long crc_0 = crc32(crc_init, "\0", 1);

    return crc32(crc_0, buf, len);
}

int main (int argc, char *argv[])
{
    unsigned int len = 0;

    if (argc < 2) return 1;
    for (len=0; argv[1][len] != '\0'; len++)
        printf ("%ld, ",
            generate_crc32((const unsigned char*)argv[1], len));
    printf ("\n");

    return 0;
}
```

# Mealy machine

An abstraction that hardware designers use to describe a component that has an input and output and keeps some state.

- Our crc32 function is a mealy machine with an initial state of 0.
- The transfer function `crc32T` updates the state (an `Unsigned 32`) using an input `Unsigned 8`, producing a tuple of the new state and the output (also an Unsigned 32, being the CRC up to the previous character).

```
crc32T :: Unsigned 32 -> Unsigned 8 -> (Unsigned 32, Unsigned 32)
crc32T prevCRC c = (prevCRC', o)
  where
    prevCRC' = crc32Step prevCRC c
    o        = prevCRC

crc32 :: Signal (Unsigned 8) -> Signal (Unsigned 32)
crc32 = mealy crc32T 0
```

The `crc32Step` function does the actual CRC32 calculation, being a bunch of `XOR`s and bit shifts.

This is directly adapted from the pure Haskell implementation in Data.Digest.Pure.CRC32, using Clash functions like `truncateB`.

The target bit width of `truncateB` does not need to be specified, it is inferred by the context: Here, the result of truncation has to be `xor'`d with the input character `c`, which is an `Unsigned 8`, so `crc` gets truncated to 8 bits.

```
crc32Step :: Unsigned 32 -> Unsigned 8 -> Unsigned 32
crc32Step prevCRC c =
  flipAll (tblEntry `xor` (crc `shiftR` 8))
  where
    tblEntry = crc32Table (truncateB crc `xor` c)
    crc = flipAll prevCRC
    flipAll = xor 0xffffffff
```

## Async ROM

That 8 bit value is used to index into crc32Table, which we implement as an async ROM. Async means that you can read it without waiting a clock cycle; ROM of course means that it's read-only, but that doesn't force the hardware tools to actually use a dedicated ROM (or RAM) component.

```
crc32Table :: Unsigned 8 -> Unsigned 32
crc32Table = unpack . asyncRomFile d256 "crc32_table.bin"
```

The values for the ROM are read from a file, which we generate by dumping an array from zlib as binary.

# Top entity

The topEntity is like a main function. The types of this are `Signal`s, which is a wrapper that Clash uses to describe synchronous (clocked) circuits. You might notice that the `crc32T` function that we use to construct a mealy machine does not use `Signal`s, it is just a pure function. The compiler can safely compose pure functions without worrying about clocking. Clash uses this to allow things like composing mealy machines.

```
crc32 :: Signal (Unsigned 8) -> Signal (Unsigned 32)
crc32 = mealy crc32T 0

topEntity :: Signal (Unsigned 8) -> Signal (Unsigned 32)
topEntity = crc32
```

# Testbench

Writing a test is as simple as providing some `testInput` (here, the word "Optiver" in ASCII), and the `expectedOutput`.

```
testInput :: Signal (Unsigned 8)
testInput = stimuliGenerator $(v [0 :: Unsigned 8, 79,
112,116,105,118,101,114])

expectedOutput :: Signal (Unsigned 32) -> Signal Bool
expectedOutput = outputVerifier $(v [0 :: Unsigned 32,
3523407757, 2814004990, 3634756580, 584440114, 781824552,
1593772748, 2102791115])
```

This will eventually be used to generate a circuit that we can use in hardware simulation, **but first …**

# Testing in the Clash REPL

We can do something that we can't normally do with VHDL: we can test it purely in software! Loading it into `clash --interactive`:

```
*CRC32> sampleN 9 $ expectedOutput (topEntity testInput)
[False,False,False,False,False,False,False,False,
cycle(system1000): 8, outputVerifier
expected value: 2102791115, not equal to actual value:
1623372462
True]
```

Regarding the "expected value not equal to actual value" error, the Clash tutorial says:

“We can see that for the first N samples, everything is working as expected, after which warnings are being reported. The reason is that `stimuliGenerator` will keep on producing the last sample, ... while the `outputVerifier` will keep on expecting the last sample, .... In the VHDL testbench these errors won't show, as the the global clock will be stopped after N ticks.”

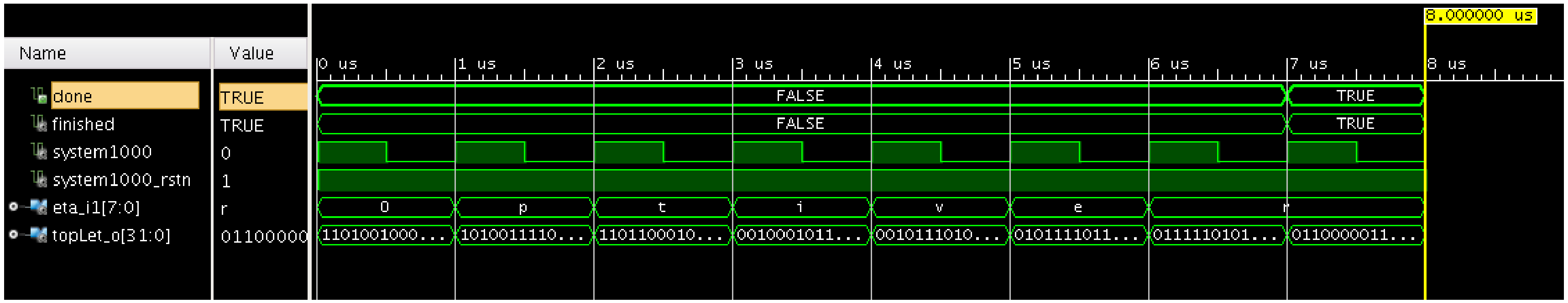
# Generating VHDL

Ok, finally it's time to generate some VHDL:

```
*CRC32> :vhdl
[1 of 1] Compiling CRC32            ( CRC32.hs, CRC32.o )
Loading dependencies took 0.053376s
Applied 143 transformations
Normalisation took 0.145971s
Netlist generation took 0.013298s
Applied 65 transformations
Applied 111 transformations
Testbench generation took 0.505447s
Total compilation took 0.752854s
```

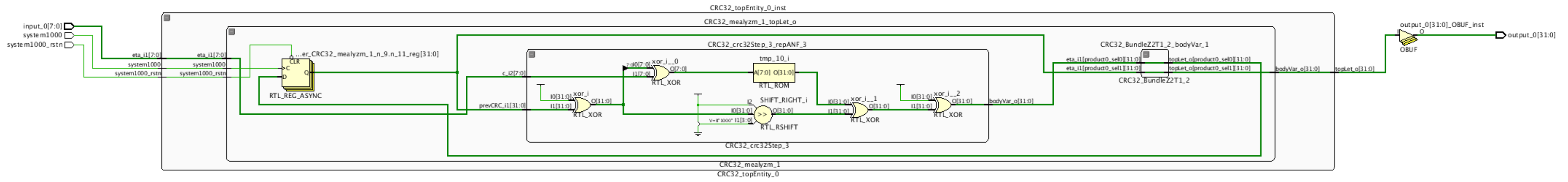
# Simulation

- The done signal indicates the test passed, finished that it finished.
- system1000 is a simulated 1000ns clock,
  and system1000_rstn is a reset line.
- eta_i1[7:0] is the 8 bit input, which we display as ASCII ("Optiver"), and topLet_0[31:0] is the CRC32 output.



# Elaborated design

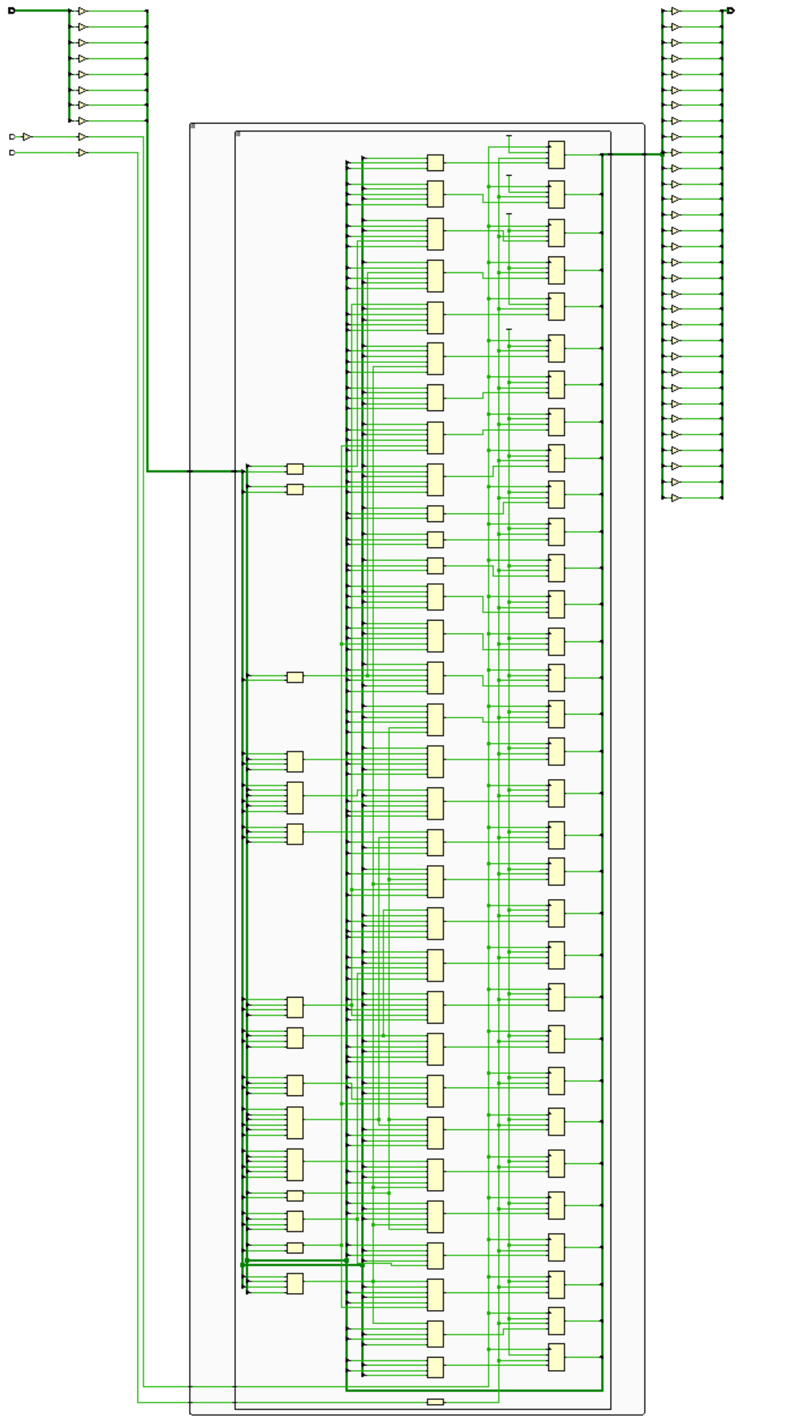
- The elaborated design is the logical design that Vivado generates after interpreting the VHDL. We can see that it includes a ROM in the middle, a few XORs and a right shift.



# Synthesis

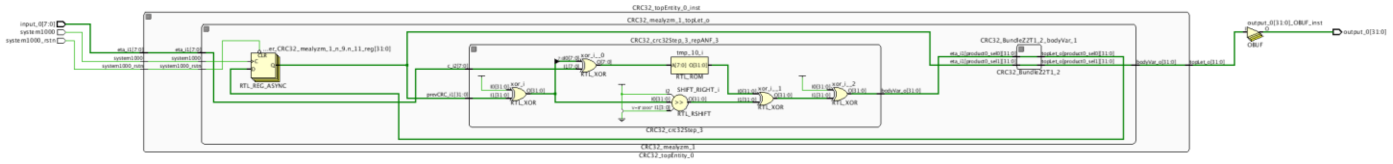
Finally, we can synthesize an actual design using gates.

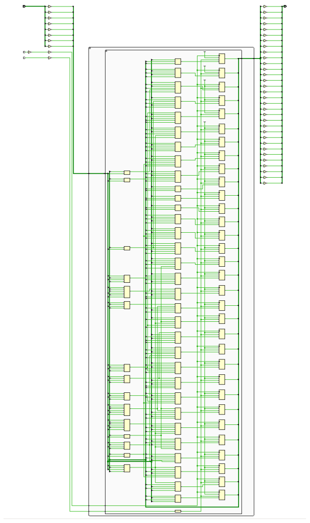
- 8 bits of input on the left, and 32 bits of output on the right
- In the middle is a whole heap of lookup tables and flip-flops.
- What happened to the ROM?



# What happened to the ROM?

Although the FPGA does contain some "block RAM" which could be used to store the crc table, for this little circuit Vivado actually found it more efficient/better to implement the lookup as a collection of one-bit lookup tables of different sizes (variously 3, 4, 5, or 6 inputs).





Good luck working backwards from the synthesized design to the logical circuit!

## Conclusion

The generated VHDL is quite readable, and the source Haskell is far easier to reason about than VHDL.

Agile hardware development:

- The automated testbench generation is useful, as that part of VHDL design is often quite tedious
- You can run tests in the software interpreter, even before generating VHDL

The world’s finest imperative programming language is also useful for implementing in hardware.