CORDIC Lab: Part 2

Design

In the previous lab, you implemented a fixed-point CORDIC generator. It this iteration of the lab, you will:

1) Make your generator type-generic
2) Run synthesis for different paramterizations of your design
3) Interface your CORDIC generator to rocket-chip

Type-generic CORDIC

In the previous iteration of the lab, you implemented a CORDIC generator that took an object of type `FixedCordicParams` and produced a fixed-point implementation of CORDIC. In this version of the lab, you will modify your generator to accept arbitrary `CordicParams[T]`.

To assist in making type-generic DSP generators, you will use the dsptools library. You may want to review 3.6 from the bootcamp.

For CORDIC, some typeclasses that may be useful are:

- `Ring`: allows you to do +, -, * and gives you 0 and 1
- `BinaryRepresentation`: allows you to do shifting, i.e. a >> 3
- `ConvertableTo`: allows you to construct a type T from Scala `Int`, `Double`, etc.

Read the dsptools documentation (link here) for more details about these typeclasses.

A class can say multiple typeclasses are available for a given type `T` like so:

```
class HasMultipleTypeclasses[T <: Data : Typeclass1 : Typeclass2 : Typeclass3] { ... }
```

Port your existing `FixedPoint` tests, but add a test for `DspReal` too. Investigate how many stages do you need for a floating point implementation of CORDIC.

Synthesis

Hammer is a UC Berkeley project with the goal of making scripted VLSI flows that are parameterizeable and portable. The ultimate goal is that your Chisel generator and FIRRTL compiler should be able to emit constaints and hints for the downstream synthesis and P&R tools to consume in an automated fashion. This should improve the ability for users to perform well-informed design-space exploration.

You will generate several instances of your `FixedPoint` CORDIC and use hammer to synthesize them for FPGA. Instructions for running hammer are in the `README`- reports will show up in the output directory.

Make plots showing utilization of LUTs and DSPs where you sweep:

- Different bitwidths for XY + Z
- Amount of unrolling (stages per cycle)

Choose what you think makes the most sense (e.g. line or bar chart), but each chart should only sweep one variable. Make these plots without gain correction enabled.

Add your plots and a discussion of the results to `doc/synthesis.md`. Don't forget that different parameterizations will also have different performance characteristics (e.g. accuracy, throughput).

Rocket-chip Integration

You will integrate your CORDIC with the rocketchip and write some C code that uses your CORDIC.

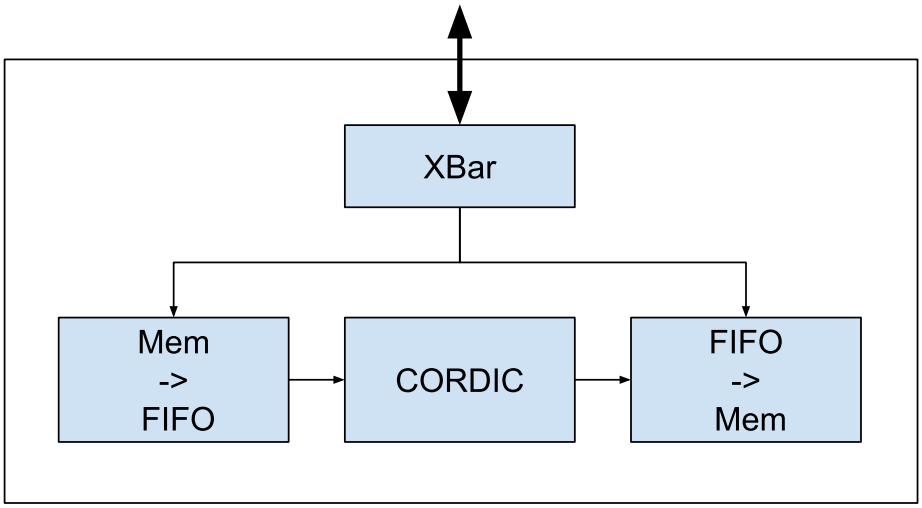

Figure 1: Block diagram

A `DspBlock` has a streaming interface and an optional memory interface. These are interfaces are implemented as diplomatic nodes. A diplomatic node lives inside a `LazyModule` (`DspBlock`s are `LazyModule`s with some extra fields). `LazyModule`s have two phases:

1) Diplomatic nodes exchange parameters and negotiate the actual values. Nodes can be connected like `a := b`, in which case nodes `a` and `b` will

exchange parameters and hardware will be automatically created in the next phase to connect `a` and `b`.

2) `lazy val module = new LazyModuleImp(this) { ... }` actually triggers the negotiations and instantiates the hardware[1]. No more node connections are allowed after `module` is instantatied. You write new hardware inside the body of the `LazyModuleImp`.

You will need to wrap your CORDIC implementation in a `DspBlock`, which will involve working with diplomatic nodes for your streaming interface. You will also need to add queues at the input and output of your CORDIC block and memory map the input and outputs of those queues. Read through the source code for places to add needed implementations.

You will also need to write a C program that exercises your CORDIC. `cordic.c` has some starter code for you. You should write a program that uses the CORDIC in both vectoring and rotation modes. Your program should print out something like:

```
XIN = XXXXX
YIN = XXXXX
ZIN = XXXXX
VEC = XXXXX

XOUT = XXXXX
YOUT = XXXXX
ZOUT = XXXXX
```

[1] `lazy val`s are a special kind of `val`. Normally, `val a = 1; val b = 2; ...` execute sequentially. You may have noticed this if you have a `val` refer to an object that is defined after- you'll get a null pointer error because the later `val` hasn't been set yet. `lazy val`s don't execute in any particular order. Instead, they are set the first time they are accessed. In the case of `LazyModule`s, `lazy val module` doesn't get set until the parent class "calls" `lazyMod.module`.