Chisel Looper
Agile Hardware Design

Sam Crow

My Generator: Guitar-Style Looper Pedal

What it does:

- record one or more audio tracks
- play back the combination of these audio tracks

Why?

So that a guitarist (or any musician) can play separate musical phrases independently and then hear their combined sound immediately without the need for other musicians or backing tracks

Interface

Sample Input & Output

signed integers, the most common uncompressed audio representation

Loop Address

for loading, allows overwriting a specific recording if desired

Load & Play

simple boolean

Parameters

Bytes per sample

I have only tested single byte recordings due to FileIO (see later slide)

Number of samples

limits the length of a recording

Max number of recordings

have tested with up to 4 tracks but no reason more wouldn’t work

How does it work

Single Mem() stores all the data, each loop one after the other

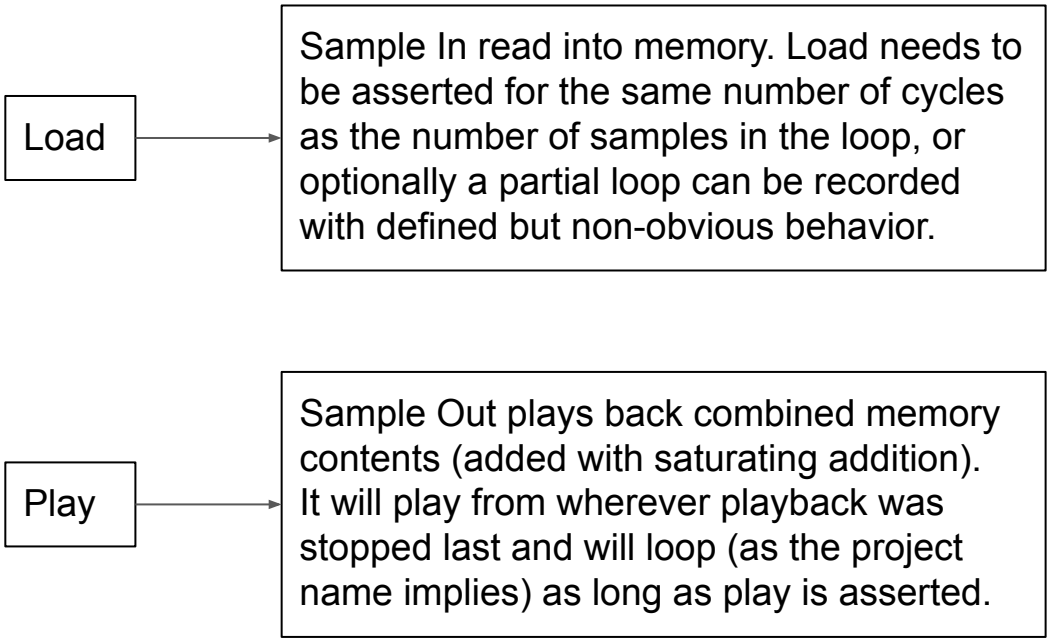
- Why mem? Because it is easier to use (no timing delay on reads simplifies the simulation of instantaneous playback). It is difficult but not impossible to convert this to a syncreadmem because it will require buffering and a different (but obvious) topology for the memory. Given more time I would have done it.

Loop Address determines where new recording will be stored (any from 1 to max number of loops)

Saturating addition (audio clipping) is done to get output from the combination of all loops

Important note: sample rate of this system depends on the clock!

Control Flow Diagram



Incremental Development Cycle

Each of these versions has an equivalent (working!) Scala model as well.

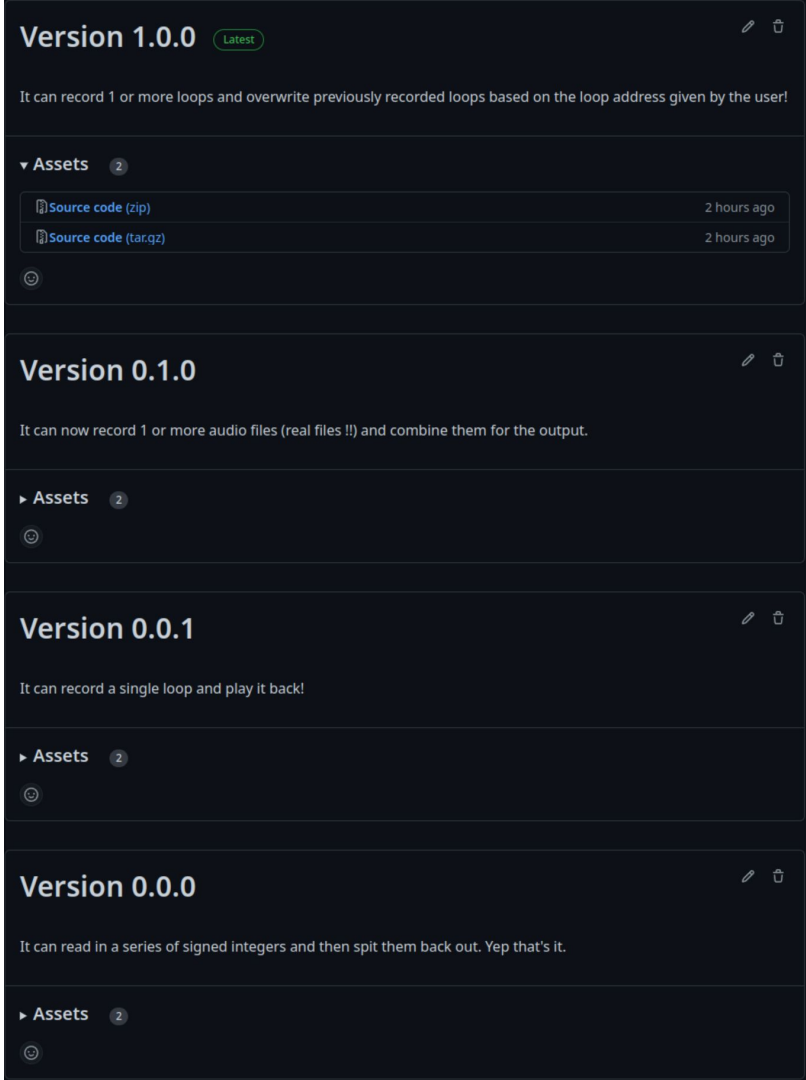
The Scala model had its own tests which were verified by a 3rd party tool (next slide).

Current Version with loop overwrite feature

Multi-loop recording

Single-loop recording

Initial version, just integer IO



Testing: Who watches the watcher? In this case, SoX

Sound eXchange (**SoX**) is a cross-platform audio editing software. It has a command-line interface, and is written in standard C. It is free software, licensed under GPL-2.0-or-later, with libsox licensed under LGPL-2.1-or-later, and distributed by Chris Bagwell through SourceForge.[1]

Thanks Chris, very cool 😎

I highly recommend this tool for any audio processing that seems like it *should* be possible on the command line. And more…!

Project Completeness

I completed all of the goals set out in my proposal and a bonus (loop overwriting)

Each function is tested and works (except multi-byte samples, which seems trivial)

Remaining ToDo items:

- Update the README with recent changes
- Add this presentation to the repo

Things I learned

About Chisel

- The Interval type. I didn’t use it but I learned about it (via ChatGPT)
- Yet again was reminded how easy it is to make width inference bugs…
- I learned that you can’t make a Vec of Mems. Especially odd since Mems are just Regs and you can make a Vec of those. Weird.

Other topics

- How to use SoX
- A bit of bash scripting to automate my SoX workflow

Demo: Hello and Welcome!

Questions?

Backup Slides

Can ChatGPT do it?

File IO

Correct :)

I need a way to read a file of bytes into a scala class

To read a file of bytes into a Scala class, you can use the Java NIO (New I/O) API. Here's an example code snippet:

scala Copy code

```scala
import java.nio.file.{Files, Paths}
import java.nio.ByteBuffer

case class MyClass(data: Array[Byte])

val filePath = "path/to/file"
val bytes = Files.readAllBytes(Paths.get(filePath))
val myObj = MyClass(bytes)
```

In this example, we define a case class `MyClass` that takes an array of bytes as its only argument. We then read the bytes of a file into a `ByteBuffer` using the `Files.readAllBytes` method, and pass the resulting byte array to an instance of `MyClass`.

Note that in this example, we assume that the file contains only data that can be represented as an array of bytes. If the file contains structured data such as serialized objects or strings, you'll need to use a different approach to deserialize the data into an instance of your class.

I need to do signed addition with saturation in chisel

In Chisel, signed addition with saturation can be implemented using the `Clip` operator. The `Clip` operator limits the output value to a given range, which is useful for implementing saturation.

Here's an example Chisel module that performs signed addition with saturation:

scala Copy code

```scala
import chisel3._

class SaturationAdder(width: Int) extends Module {
  val io = IO(new Bundle {
    val a = Input(SInt(width.W))
    val b = Input(SInt(width.W))
    val sum = Output(SInt(width.W))
  })

  // Perform signed addition
  val tempSum = io.a + io.b

  // Clip the result to the valid range for SInt
  val minVal = -(1 << (width-1))
  val maxVal = (1 << (width-1)) - 1
  io.sum := tempSum.clip(minVal.S, maxVal.S)
}
```

In this example, the `SaturationAdder` module takes two `SInt` inputs, `a` and `b`, and outputs their sum, `sum`. The `tempSum` variable computes the sum of `a` and `b` using Chisel's built-in addition operator for `SInt` types.

To implement saturation, we use the `clip` method of the `SInt` class to limit the range of `tempSum` to the valid range for signed integers. The minimum and maximum values for signed integers of `width` bits are calculated using bit-shifting operations.

Finally, we assign the clipped value to the output `sum`.

You can test this module by creating a test bench that generates random inputs and verifies that the output is correctly saturated.

clip only works on 'Interval' types, it does not work on 'SInt' types

You're right, I apologize for my mistake. The `clip` method is only available for `Interval` types in Chisel, and not for `SInt` types.

To perform signed addition with saturation on `SInt` types in Chisel, we can use a conditional statement to check if the result is within the valid range, and if it's not, we can set the output to the maximum or minimum value, depending on the sign of the result.

Here's an example Chisel module that performs signed addition with saturation using a conditional statement:

scala Copy code

```scala
import chisel3._

class SaturationAdder(width: Int) extends Module {
  val io = IO(new Bundle {
    val a = Input(SInt(width.W))
    val b = Input(SInt(width.W))
    val sum = Output(SInt(width.W))
  })

  // Perform signed addition
  val tempSum = io.a + io.b

  // Calculate the valid range for signed integers
  val minVal = -(1 << (width-1))
  val maxVal = (1 << (width-1)) - 1

  // Perform saturation using a conditional statement
  io.sum := Mux(tempSum >= minVal.S, Mux(tempSum <= maxVal.S, tempSum, maxVa
}
```

In this example, the `SaturationAdder` module takes two `SInt` inputs, `a` and `b`, and outputs their sum, `sum`. The `tempSum` variable computes the sum of `a` and `b` using Chisel's built-in addition operator for `SInt` types.