


```
In [ ]: val path = System.getProperty("user.dir") + "/source/load-ivy.sc"
interp.load.module(ammonite.ops.Path(java.nio.file.FileSystems.getDefault()).
```

```
In [ ]: import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

Chisel4ml: Generating Fast Implementations of Deeply Quantized Neural Networks using Chisel Generators

 No description has been provided for this image

Jure Vreča

jure.vreca@ijs.si

Chisel

- Constructing Hardware in a Scala Embedded Language
- Chisel is not HLS.
- Chis is a type-safe meta-programming language for synchronous digital logic design:
 - Parametrized types
 - Object-oriented programming
 - Functional programming
 - Static type checking

Note: Some of the slides and material were taken from:

<https://github.com/freechipsproject/chisel-bootcamp>

Example

 No description has been provided for this image

The diagram shows a simple FIR filter that outputs a moving average of the inputs. The z1 and z2 wires are outputs from the registers.

```
In [ ]: class MovingAverage3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })
  val z1 = RegNext(io.in) // Create a register whose input is connected to t
  val z2 = RegNext(z1)    // Create a register whose input is connected to t
  io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U) // `1.U` is an unsigned
}
```

- This code shows how to describe the previously shown circuit in Chisel HCL.
- Every hardware module in chisel must inherit from the Chisel class Module.
- Next, we define the input-output interface using the IO command.
- The RegNext function creates a register, where the input to the register is the argument, and the result represents the output of the register.
- Thus we define two registers with RegNext and connect them to z1 and z2.
- After that we simply compute the output in a straightforward fashion.

```
In [ ]: visualize(() => new MovingAverage3(8))
```

- We can visualize the resulting hardware with this visualize function.
- We can see that two registers are created, and a plethora of multiplication and addition nodes.
- The clock and reset are implicit, however, they can also be made explicit, if desired. For example if working with multiple clock domains.

```
In [ ]: //print(getFirrtl(new MovingAverage3(8)))
print(getVerilog(new MovingAverage3(8)))
```

- We can also print the Verilog and FIRRTL representation of the circuit.
- FIRRTL is an intermediate representation used by chisels backend
- As you can see the Verilog is not particularly readable, and also has a tree of if-defs.
- The FIRRTL representation looks a bit nicer in , and is very similar to Verilog.

FIR Generator

```
In [ ]: // Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt())
  })
  // Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
```

```

zs(0) := io.in
for (i <- 1 until coeffs.length) {
  zs(i) := zs(i-1)
}

// Do the multiplies
val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

// Sum up the products
io.out := products.reduce(_ +& _)
}

```

- The chisel code for the MovingAverage filter shown before is somewhat similar to equivalent code in Verilog or VHDL.
- A more appropriate usage of Chisel is to create generators that are reusable.
- This code snippet shows how to create a generalized implementation of a FIR filter in Chisel, where the coefficients are a parameters of the class.
- I will not discuss this code in detail, but in essence it uses Scas functional paradigms to create the hardware in a concise fashion.

```

In [ ]: // same 3-point moving average filter as before
visualize(() => new FirFilter(8, Seq(1.U, 1.U, 1.U)))


// 1-cycle delay as a FIR filter
//visualize(() => new FirFilter(8, Seq(0.U, 1.U)))

// 5-point FIR filter with a triangle impulse response
//visualize(() => new FirFilter(8, Seq(1.U, 2.U, 3.U, 2.U, 1.U)))

```

- The code shown on the previous slide generates the exact same hardware, when appropriate input coefficients are given.
- For example, if we input 1, 1, 1, then we get the exact same circuit we did before.
- We can also get cycle delay filter, or a more complicated 5-point triangle impulse response filter.

Artificial Neural Networks

 No description has been provided for this image

$$y = f\left(b + \sum_{i=0}^{N-1} x_i \cdot w_i\right)$$

- Lets now move on to artificial neural network models.
- In brief, ANNs are in the simple case composed by a set of layer, most whom are computed by neurons connected in specific patterns.
- The figure above shows an example of an artificial neuron model.
- x_0 through x_{N-1} represent the input vector, w represent the weight vectors, b is the bias value and y is the scalar output.
- So to compute a neuron we perform a dot product between the input and weight vectors and add the bias.
- The attained value is called the pre-activation and is input to a non-linear activation function f which computes the output of the neuron.

How does chisel4ml use Chisel?

```
In [ ]: def neuron[I <: Bits,
        W <: Bits,
        M <: Bits,
        A <: Bits,
        O <: Bits](in: Seq[I],
                  weights: Seq[W],
                  thresh: A,
                  mul: (I, W) => M,
                  add: Vec[M] => A,
                  actFn: (A, A) => O,
                  shift: Int): O = {
  val muls = VecInit((in zip weights).map{
    case (a,b) => mul(a,b)
  })
  val pAct = add(muls)
  val sAct = (pAct << shift.abs).asTypeOf(pAct)
  actFn(sAct, thresh)
}
```

- So how does chisel4ml create neurons?
- This is done by creating a parameterized implementation of a neuron.
- The neuron is parameterized by the input, weight and output type.
- It takes a sequence of inputs and weights, and a threshold value, which is the inverse of the bias value.
- It also takes as parameter three functions for multiplication, addition and the activation function.
- This creates a neuron implementation that is completely generic to the quantization scheme.

```
In [ ]: def mulUQ(i: SInt, w: SInt): SInt = i * w // Uniform quantization
def addUQ = (x: Vec[SInt]) => x.reduceTree(_ +& _)
```

```

def mulBW = (i: SInt, w: Bool) => Mux(w, i, -i) // Binary weight quantization
def mulBNN(i: Bool, w: Bool): Bool = ~(i ^ w) // Binarized quantization
def addBNN = (x: Vec[Bool]) => PopCount(x.asUInt)

def reluFn(act: SInt, thresh: SInt): UInt = Mux((act - thresh) > 0.S, (act - thresh), 0.S)
def signFn(act: UInt, thresh: UInt): Bool = act >= thresh

```

- This slide shows the multiplications and addition function that are input to the neuron function.
- The first functions mulUQ and addUQ, show a typical case where weights are signed integers of arbitrary bitwidths.
- The second multiplication function is for binary-weight quantization. In this scheme the neuron weights are binary, but the inputs are not.
- At the most extreme level of quantization there exists binarized neural networks, that have both inputs and weights binary. In this case the multiplication is transformed into the XNOR operation, and the addition is transformed into the population count operation.

```

In [ ]: class DummyUniformModule extends Module {
  val io = IO(new Bundle {
    val in = Input(Vec(3, SInt(4.W)))
    val out = Output(UInt())
  })
  io.out := neuron[SInt, SInt, SInt, SInt, UInt](in = io.in,
    weights = Seq(1.S, -2.S, 3.S),
    thresh = -1.S,
    mul = mulUQ,
    add = addUQ,
    actFn = reluFn,
    shift = 1
  )
}

```

- This slide shows an example usage of the neuron function where the module has 3 4-bit inputs and a single output.

```

In [ ]: visualize(() => new DummyUniformModule())
//print(getFirrtl(new DummyUniformModule()))
//print(getVerilog(new DummyUniformModule()))

```

```

In [ ]: class DummyBinarizedModule extends Module {
  val io = IO(new Bundle {
    val in = Input(Vec(3, Bool()))
    val out = Output(UInt())
  })
  io.out := neuron[Bool, Bool, Bool, UInt, Bool](in = io.in,
    weights = Seq(true.B, false.B, true.B),
    thresh = 2.U,
    mul = mulBNN,
    add = addBNN,
    actFn = reluFn,
    shift = 0
  )
}

```


```
}  
  
    mul = mulBNN,  
    add = addBNN,  
    actFn = signFn,  
    shift = 0  
)
```


- We can create a similar module with a binarized neuron.

```
In [ ]: visualize(() => new DummyBinarizedModule())  
print(getFirrtl(new DummyBinarizedModule()))  
//print(getVerilog(new DummyBinarizedModule()))
```

Other abstractions in Chisel4ml:

- ProcessingElement == layer
- ProcessingPipeline == model

 No description has been provided for this image

 No description has been provided for this image