Overview and Perspective
Chisel techniques
Uncore + RISCV-Rocket state of the art
Thoughts moving forward
Next assignment
Exploration Workflow

- input energy budget
- configuration creates generator instance
- simulation for performance
- ASIC workflow for energy usage
- configuration fed back for refinement

from “Rethinking Digital Design: Why Design Must Change” by Shacham et al in IEEE Micro magazine
What is a Generator?

- architectural template built out of modular components
  - parameterized with design options forming configuration
  - creates a whole family of designs
- configuration
  - drives synthesis and validation support
  - matches target and objectives
  - encompass domain-specific designer knowledge including trade-offs
- most design costs are amortized
Verilog with Generate – limited parameterization
Verilog created with Perl Scripts – awkward to use
Bluespec – unable to use types and values interchangeably
Genesis2 – interleaved languages like PHP
Software Generators

- Software is parameterized all the time
- Potentially closest is GUI driven parameters (e.g., settings)
- Create parameters in certain format
- Often include publish / subscribe
Java Beans

Standard for easily creating GUI around parameters

- permits web interface to object
- permits automatic GUI through introspection
- wraps many objects into one so that it can be passed around
- allows publish / subscribe protocol

requirements

- Empty Constructor
- Getter / Setter Methods
- Serializable
Types of Generator Parameters

- numeric
  - continuous
  - discrete
- symbolic

but what about
- collections – e.g., dimensions
- classes – e.g., packet format
- functions – e.g., constructor
- policies – e.g., coherence policy
- patterns – e.g., time multiplex
Categories of Generator Parameters

- architectural
  - visible to software
  - required inputs
- constrained
  - based on other parameters
  - could be same or derived in computed fashion
- micro-architectural
  - free to tune based on objectives
  - doesn’t affect ISA
<table>
<thead>
<tr>
<th>parameter name</th>
<th>Impacts</th>
<th>Parameter Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>word-size</td>
<td>memory block width, decoding of address vector, processor side interface width</td>
<td>constrained / inherited (requires information from the relevant CPU instance)</td>
</tr>
<tr>
<td>line-size</td>
<td>number of memory blocks, decoding of address vector, cache controller interface width</td>
<td>constrained / inherited (requires information from the relevant cache controller instance)</td>
</tr>
<tr>
<td>way-size</td>
<td>size of memory blocks</td>
<td>free (optimization -&gt; requires late binding)</td>
</tr>
<tr>
<td>associativity</td>
<td>number of memory blocks</td>
<td>free (optimization -&gt; requires late binding)</td>
</tr>
<tr>
<td>meta-data bits</td>
<td>line state, cache protocol (e.g., coherence)</td>
<td>architectural (set-by-user -&gt; requires late binding)</td>
</tr>
</tbody>
</table>

** from “Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogenous Compute Platforms” by Ofer Shacham in dissertation at Stanford University, 2011
Generator Stake Holders?

- application designers
  - what are all parameters?
  - how do I access them through GUI or web page?
- circuit designers
  - what is the generator API?
  - how to programmatically construct an instance?
- optimizers
  - what are free variables and ranges?
  - how do I uniformly sample the variable range?
- verify generator instances not the generator itself
- harder to create verification scheme that works across design space
- one example is relaxed contract that is independent of exact timing
- verification can be made modular and effort amortized
- coarse correctness can be fixed across design space and shared
- parameterization makes finding bugs easier because bugs occur in certain configurations orders of magnitude more frequently
- changing configuration is easier than writing adding additional tests

** from “Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogenous Compute Platforms” by Ofer Shacham in dissertation at Stanford University, 2011
Chisel Generator Techniques

- numeric sizes for word width or memory
- computed initial values
- algorithmic construction
- choosing between different implementations
- policy implementation
- type parameterization
- mix of all techniques
class Cache(...
    line_size: Int = 128,
    cache_depth: Int = 16
    ... ) extends Component {

    ...
    val addr_idx_width = log2(cache_depth).toInt
    val addr_off_width = log2(line_size/32).toInt
    val addr_tag_width = 32 - addr_idx_width - addr_off_width - 2
    ...
}

- sizes for wire widths or memory depths
- arbitrary computed values derived from parameters
class recodedFloatNCompare(SIG_WIDTH: Int, EXP_WIDTH: Int) extends Component {
  val io = new recodedFloat32Compare_io(SIG_WIDTH, EXP_WIDTH)

  val signA = io.a(SIG_WIDTH+EXP_WIDTH)
  val expA = io.a(SIG_WIDTH+EXP_WIDTH-1, SIG_WIDTH)
  val sigA = io.a(SIG_WIDTH-1,0)
  ...

  io.a_eq_b_invalid := isSignalingNaN A || isSignalingNaN B
  io.a_lt_b_invalid := isNaNA || isNaNB
  io.a_eq_b := !isNaNA && magEqual && (isZeroA || signEqual)
  io.a_lt_b := !io.a_lt_b_invalid &&
    Mux(signB, signA && !magLess && !magEqual,
    Mux(signA, !(isZeroA && isZeroB), magLess))
}

** can be done in Verilog but Chisel allows more powerful computed values
class Mul extends Component {
    val io = new Bundle {
        val x = UFix(INPUT, 4)
        val y = UFix(INPUT, 4)
        val z = UFix(OUTPUT, 8) }

    val muls = new Array[UFix](256)
    for (x <- 0 until 16; y <- 0 until 16)
        muls((x << 4) | y) = x * y

    val tbl = ROM(muls){ UFix(8) }

    io.z := tbl((io.x << 4) | io.y)
}

other examples include ucode
NxM Grid of Tiles
class Grid(numCols: Int, numRows: Int) extends Component {
    val io = new Bundle { val dat = new PortIO()( new Packet() ) }
    val grid = new ArrayBuffer[ArrayBuffer[Tile]]()
    def connectPorts (p1: PortIO[Packet], p2: PortIO[Packet]) = {
        val q12 = new Queue(2)({ new Packet() })
        p1.o <> q12.io.enq; p2.i <> q12.io.deq
        val q21 = new Queue(2)({ new Packet() })
        p1.i <> q21.io.deq; p2.o <> q21.io.enq
    }
    for (j <- 0 until numRows) {
        val row = new ArrayBuffer[Tile]()
        for (i <- 0 until numCols) {
            val t = new Tile()
            row += t
            t.io.tid := Cat(Bits(i+1, 4), Bits(j, 4)).toUFix;
            if (i > 0) connectPorts(row(i-1).io.datPorts(2), t.io.datPorts(0))
            if (j > 0) connectPorts(grid(j-1)(i).io.datPorts(3), t.io.datPorts(1))
        }
        grid += row
    }
}
```java
class Risc(isFastMul: Bool) extends Component {
  ...
  switch(op) {
    is(add_op) { rc := ra.toUFix + rb.toUFix }
    is(mul_op) { rc := if (isFastMul) fastMul(ra, rb) else Mul(ra, rb) }
    is(imm_op) { rc := (rai << UFix(8)) | rbi }
    ...
  }
  ...
}
```

- another example is FPU or no FPU
Instead of writing a SmallFilter and OddFilter, a better Filter solution would be to create a single reusable Filter class that allows the user to specify the filter function. We can do this by

- specifying a filter function as a Filter constructor argument:

```scala
class Filter (isOk: (UFix) => Bool) extends Component {
  val io = new FilterIO()
  io.out.data := io.in.data
  io.out.valid :=
    io.in.valid && isOk(io.in.data)
}
val odds = new Filter((x) => x & UFix(1))
val smalls = new Filter((x) => x < UFix(10))
```
abstract class CoherencePolicy {
    def isHit (cmd: Bits, state: UFix): Bool
    def isValid (state: UFix): Bool
    def needsWriteback (state: UFix): Bool
    def newStateOnHit(cmd: Bits, state: UFix): UFix
    ...
    def getAcquireTypeOnPrimaryMiss(cmd: Bits, state: UFix): UFix
    ...
    def newRelease (incoming: Probe, state: UFix): Release
    def messageHasData (reply: Release): Bool
    def messageHasData (acq: Acquire): Bool
    def messageHasData (reply: Grant): Bool
    ...
    def isCoherenceConflict(addr1: Bits, addr2: Bits): Bool
    def getGrantType(a_type: UFix, count: UFix): Bits
    def needsMemRead(a_type: UFix, global_state: UFix): Bool
    ...
}

** based on Henry Cook’s Uncore parameterization.
class MICoherence extends CoherencePolicyWithUncached {
  ...
  def isHit (cmd: Bits, state: UFix): Bool = state != tileInvalid
  def isValid (state: UFix): Bool = state != tileInvalid
    MuxLookup(cmd, (state === tileValid), Array(
      M_INV -> (state === tileValid),
      M_CLN -> (state === tileValid)
    ))
  }
  ...
}

class ReferenceChip (co: CoherencePolicy) extends Component {
  ...
  val hit = co.isHit(cmd, state)
  ...
}
Suppose we want to create a polymorphic decoupled interface (e.g., data with ready/value signals) to allow for arbitrary Chisel data types. We can do this by using:

- Scala parameterized types and
- a curried class constructor argument

We want to be able to write

```scala
val ufix32s = new FIFOIO(){ UFix(width = 32) }

class Packet extends Bundle {
    val header = UFix(width = 8)
    val body = Bits(width = 64)
}
val pkts = new FIFOIO(){ new Packet() }
```

but how do we define this parameterized `FIFOIO`?
First we need to learn about parameterized types in Scala. We can define a generic `Mux` function as taking a boolean condition and `con` and `alt` arguments (corresponding to then and else expressions) of type `T` as follows:

```scala
def Mux[T <: Bits](c: Bool, con: T, alt: T): T { ... }
```

where

- `T` is required to be a subclass of `Bits` and
- the type of `con` and `alt` are required to match.

You can think of the type parameter as a way of just constraining the types of the allowable arguments.
In Chisel we use special syntax for passing in a type constructor for parameterized types (such as Reg, Mem, and ROM). For example, for we can construct a reg using the following syntax:

```scala
val r = Reg(){ Bits(width = 32) }
```

You can write your own functions to allow this syntax and behavior as follows:

```scala
def myReg[T <: Data]()(type: => T) { ... }

myReg(){ Bits(width = 16) }
```

where the second parameter list has a single zero argument function parameter (aka thunk) that when called with no arguments produces a chisel type.
Now we can define `FIFOIO` and `FilterIO` using parameterized types and a curried argument as follows:

```scala
class FIFOIO[T <: Data](type: => T) extends Bundle {
  val data   = type.asOutput
  val valid  = Bool(OUTPUT)
  val ready  = Bool(INPUT)
}

class FilterIO[T <: Data](type: => T) extends Bundle {
  val in     = new FIFOIO(){ type }.flip
  val out    = new FIFOIO(){ type }
}
```

We can now define `FIFOIO` on arbitrary data types:

```scala
val ufix32s = new FIFOIO(){ UFix(width = 32) }
val pkts    = new FIFOIO(){ new Packet() }
```
class Router[T <: Bundle](val n: Int, sel: T => UFix)(data: => T) extends Component {
  val io = new Bundle { Vec(nPorts){ new PortIO()( data ) } } }  
  def routeN (arb: ioArbiter[T], rdyWires: Vec[Bool], readies: Seq[Bool], idx: Int) = {
    for (i <- 0 until n) {
      val (in, arb) = (io.ports(i).i, arbs.in(i))  
      val is_selected = (sel(in.bits) === Bits(idx));  
      arb.valid := in.valid && is_selected  
      arb.bits := in.bits  
      in.ready := rdyWires(i)  
      readies(i) = readies(i) || (arb.ready && is_selected);  
    }  
    arbs
  }
  val arbs = Vec(n){ (new Arbiter(n)( data )).io }
  val readies = (new Array[Int](n)).map(x => Bool(false))
  val rdyWires = Vec(n){ Bool() }
  for (j <- 0 until n)  
    io.ports(j).o <> routeN(arbs(j), rdyWires, readies, j).out
  for (j <- 0 until n)  
    rdyWires(j) := readies(j)  
}
current idea for how to do multiprocessor configuration in Chisel
by Henry Cook and Andrew Waterman
will motivate by building it up incrementally
parameters to component constructor

```scala
class Cache(sets: Int, assoc: Int, ...) extends Component {
    ...
}
val ca = new Cache(4, 2, ...)
```

- no organization of parameters
- hard to thread through construction
- no separation of configuration from elaboration
configuration object to component constructor

```scala
case class CacheConfig(sets: Int, assoc: Int = 16, ...) {
  val lines = sets * assoc
  require(isPow2(sets) && isPow2(assoc))

  ...
}
...
val co = CacheConfig(128, 4, ...)
val ca = new Cache(co)
```

- where case classes
  - don’t require `new`
  - support pattern matching
  - supply automatic copy method with overriding arguments

- what about implementation choices?
- what about functional or object-oriented parameters?
hardware generation protocol included in configuration object

case class CacheConfig(sets: Int, assoc: Int = 16, co = CoherencePolicy) {
  val lines = sets * assoc
  require(isPow2(sets) && isPow2(assoc))
  ...
}
...
val co = CacheConfig(128, 4, new MICOherencePolicy(), ...)
val ca = new Cache(co)

- does this break separation of concerns?
- still too many parameters
Scala implicit parameter

class Cache(implicit c: CacheConfig) { ... }
class CPU(implicit c: CacheConfig) {
    val ca = new Cache()
}
implicit val co = new CacheConfig(128, 4, new MICoherencePolicy(), ...)
val cpu = new CPU()

- avoids having to repeatedly pass config parameter
Organize parameters into hierarchy of configuration objects:

```scala
case class CacheConfig(sets: Int, assoc: Int = 16, ...) { ... }
case class CPUConfig(cac: CacheConfig, ...) { ... }
class CPU(implicit c: CPUConfig) {
  implicit val cac = c.cac
  val ca = new Cache()
}
val cac = new CacheConfig(128, 4, new MICoherencePolicy(), ...)
implicit val cpuc = new CPUConfig(cac, ...)
val cpu = new CPU()
```
Wrap parameters up to be introspectable:

```scala
case class CacheConfig(sets: ConfigPow2Int, assoc: ConfigPow2Int, ...) {
  val lines = sets.value * assoc.value
  ...
}
...
val co = new CacheConfig(ConfigPow2Int(128, min = 1, FREE), ConfigInt(4), ...)
val ca = new Cache(co)
```

- can introspect each element and know which are free
- can know range restrictions
- can know how to evenly sample in legal fashion
- can read and write user friendly form
- could query for values through GUI or webpage
Thoughts

- need standardized configuration mechanism
- can write rest of tool flow around it
- still don’t know how to deal with functional parameters
- hoping someone will run with these ideas as a project
■ read *Using a Configurable Processor Generator for Computer Architecture Prototyping*

■ come prepared with
  ■ the good
  ■ the bad
  ■ three questions