Advanced Parameterization Manual

Adam Izraelevitz
EECS Department, UC Berkeley
adamiz@eecs.berkeley.edu

October 1, 2014

1 Introduction

This document is a manual for using the advanced parameter library within *Chisel*. For more general information regarding *Chisel* as a hardware construction language, please see the Getting Started documentation.

As hardware designs grow in complexity, modularity becomes necessary for maintaince and verification. The primary use case for *Chisel* is describing diverse and highly-parameterized hardware generators, and we quickly realized that the traditional parameterization method forces brittleness into a design's source code and limits component reuse.

The outline of this document is as follows: in Section **??**, we describe the basic objects and methods for the advanced parameterization mechanism, as well as the required boilerplate to use it. In Section **??**, a series of increasingly complex examples of design patterns are described. For each example, we propose the simplest parameterization scheme which solves the problem. As the examples build in complexity, so do the parameterization requirements, until we we arrive at the described advanced parameterization mechanism. The next section, Section **??**, introduces the concept of `Knobs` and their relationship to design constraints and the `Parameters` object. Finally, in Section **??**, we explain multiple design heuristics which should be followed when using advanced parameterization.

2 Advanced Parameterization

Every *Chisel* Module has a member `params` of class `Parameters` that provides the mechanism for passing parameters between modules.

This section describes the following features: (1) the `Parameters` class and associated methods/members; (2) the basic usage model; (3) syntactic sugar; (4) boilerplate code for exposing parameters to external users/programs; (5) advanced functionality via Views (site, here, up).

2.1 Classes and Methods

The `Parameters` class has the following base methods:

```
class Parameters {
  // returns a value of type T
  def apply[T](key:Any):T

  // returns new Parameters class
  def alter(mask:(Any,View,View,View)=>Any):Parameters

  // returns a Module's Parameters instance
  def params:Parameters
}
```

`View` is a class containing a base method:

```
class View {
  // returns a value of type T
  def apply[T](key:Any):T
}
```

`Parameters` has a factory object containing one basic method:

```
object Parameters {
  // returns an empty Parameters instance
  def empty:Parameters
}
```

The `Module` factory object now has an additional apply method:

```
object Module {
  // returns a new Module of type T, initialized with a
       Parameters instance if  _p !=None.
  def apply[T<:Module](c: =>T)(implicit _p:
       Option[Parameters] = None):T
}
```

2.2 Basic Usage Model

This example shows the simplest usage of (1) querying params, (2) altering a Parameters object, and (3) passing a Parameters object to a Module:

```
class Tile extends Module {
  val width = params[Int]('width')
}
object Top {
  val parameters = Parameters.empty
  val tile_parameters = parameters.alter(
       (key,site,here,up) => {
    case 'width' => 64
  })
  def main(args:Array[String]) = {
    chiselMain(args,()=>Module(new
         Tile)(Some(tile_parameters)))
  }
}
```

Within the Module `Tile`, the `params` member is queried by calling Parameters.apply with the key and return value type.

In `Top`, an empty parameters is created by calling Parameters.empty; then it is altered with a function of type `(Any,View,View,View) => Any` to return a new Parameters instance, which is assigned to `tile_parameters`.

After wrapping `tile_parameters` within `Some:Option[Parameters]`, it is passed as a second argument to the Module object when passed to `chiselMain`.

2.3 Syntactic Sugar: Field[T]

The simple example requires the return type `Int` must be included as an argument to the apply method, otherwise the Scala compiler will throw an error:

```
class Tile extends Module {
  val width = params[Int]('width')
}
```

Alternatively, one can create a case object for each key which extends `Field[T]` and pass that directly into `params` apply method. Because `Field` contains the return type information, the type does not need to be passed:

```
case object Width extends Field[Int]
class Tile extends Module {
  val width = params(Width)
}
```

For the rest of the document, assume the key to every query is a case class that extends `Field[T]` with the correct return type.

2.4 Syntactic Sugar: Passing and Altering

As a module hierarchy is formed, `Parameters` objects are passed between a parent module and a child module. If specified by the programmer, these objects can be copied and altered prior to instantiating the child.

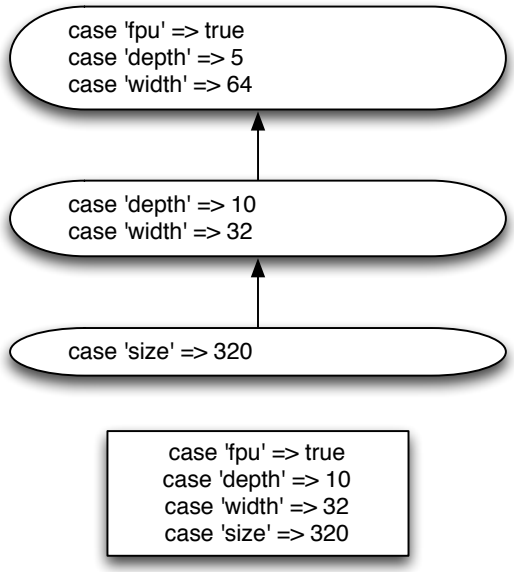


Figure 1: An example of Memory's key/value chain and flat map.

Anytime an alteration is performed, *Chisel* internally copies the existing chain of key/value mappings and attaches the provided key/value mappings to the bottom of this chain. When a query is evaluated, it first queries the chain's bottom key/value mapping. If there is no match, the query is then evaluated on the next key/value mapping in the chain, and so forth. If a query reaches the top of the chain with no matches, *Chisel* triggers a `ParameterUndefinedException`.

When instantiating a child, the parent can pass its `Parameters` object one of two ways:

1. Explicitly pass its `Parameters` object to its child via a second argument to the Module factory, wrapped in `Option[Parameters]`:

   ```
   class Tile extends Module {
     val width = params(Width)
     val core = Module(new Core)(Some(params))
     // Explicit passing of Tile's params to Core
   }
   ```

2. Implicitly pass its `Parameters` object to its child:

   ```
   class Tile extends Module {
     val width = params(Width)
     val core = Module(new Core)
     // Implicit passing of Tile's params to Core
   ```

```
}
```

If a parent wants to copy/alter the child's dictionary, the parent has two methods to do so:

1. Provide a PartialFunction mapping as an argument to the Module factory. Internally, *Chisel* will copy the parent's `Parameters` object and apply the alteration:

```
class Tile extends Module {
  val width = params(Width)
  val core = Module(new Core,{case Width => 32})
  // Provide PartialFunction to Module factory
       constructor to alter Core's
       \code{Parameters} object
}
```

2. Call the `Parameter.alter` function, which returns a new `Parameters` object. This approach gives the programmer access to the new `Parameters` object, as well as the ability to use `site`, `here`, and `up` (see Sections **??**, **??**, **??**) :

```
class Tile extends Module {
  val width = params(Width)
  val core_params = params.alter(
    (pname,site,here,up) => pname match {
      case Width => 32
    })
  val core = Module(new Core)(Some(core_params))
  // Use the Parameter.alter method to return an
       altered Parameter object. Only use when
       site, here, or up mechanisms are needed
}
```

A more complicated example of a alteration chain is shown in Figure **??** and describe below:

```
class Tile extends Module {
  ...
  val core = Module(new Core, {case FPU => true; case
       QDepth => 20; case Width => 64})
}
class Core extends Module {
  val fpu = params(FPU)
  val width = params(Width)
  val depth = params(Depth)
  val queue = Module(new Queue,{case Depth => depth*2;
       case Width => 32})
}
class Queue extends Module {
  val depth = params(Depth)
  val width = params(Width)
  val mem = Module(new Memory,{case Size => depth *
       width})
}
class Memory extends Module {
  val size = params(Size)
  val width = params(Width)
}
```

2.5 ChiselConfig and Boilerplate

Chisel's mechanism to seed the top-level parameters is through a `ChiselConfig` object. `ChiselConfig.topDefinitions` contains the highest parameter definitions and is of the following form:

```
case object Width extends Field[Int]
class DefaultConfig extends ChiselConfig {
  val topDefinitions:World.TopDefs = {
    (pname,site,here) => pname match {
      case Width => 32
    }
  }
}
```

Normally, a design calls `chiselMain.apply` to instantiate a design. To use *Chisel*'s parameterization mechanism and correctly seed a `ChiselConfig`, one should instead call `chiselMain.run` with the design NOT surrounded by the `Module` factory. The reason for this change is to preserve backwards compatibility with existing designs, although we intend to fix this in future releases.

An example of calling `chiselMain.run` is as follows:

```
object Run {
  def main(args: Array[String]): Unit = {
    chiselMain.run(args, () => new Tile())
  }
}
```

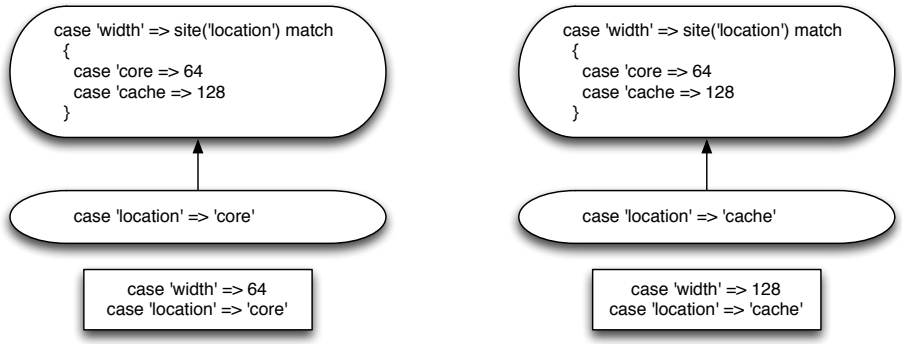
To instantiate a design with a specific `ChiselConfig`, simply call the *Chisel* compiler with the `--configInstance` *project_name.configClass_name* argument.

2.6 Using site

To help the designer express dependencies between parameters, we added the `site` mechanism. To understand its function, remember that conceptually, a queried Module's params member first looks at the bottom key/value mapping in its chain of key/-value mappings. If there is no match, the query moves up the chain.

Suppose we have some modules which have following form:

```
class Core extends Module {
  val data_width = params(Width)
  ...
}
class Cache extends Module {
  val line_width = params(Width)
  ...
}
```



(a) Core's key/value chain and flat map (b) Cache's key/value chain and flat map

Figure 2: For (a), site(Location) will return Core, while in (b) site(Location) will return Cache.

Unfortunately, both have identical queries for `Width` but, for this example's sake, have different semantic meaning. Inside a core, `Width` means the word size, while in the `Cache`, `Width` means the width of a cache line. We want to be able to easily tailor the parameter's response to either query.

The `site` mechanism allows a key/value mapping in the middle of the chain to make its own queries that start at the bottom of the chain.

Consider the following example:

```
class DefaultConfig extends ChiselConfig {
  val top:World.TopDefs = {
    (pname,site,here) => pname match {
       case Width => site(Location) match {
         case 'core' => 64 // data width
         case 'cache' => 128  // cache line width
       }
    }
  }
}
class Tile extends Module {
  val core = Module(new Core, {case Location => 'core'})
  val cache = Module(new Cache, {case Location =>
       'cache'})
}
```

The top-level key/value mapping is using `site` to query the bottom of the chain for `Location`. Depending on what value returns (either `'core'` or `'cache'`), the top-level key/value mapping produces a different value (Figure **??**).

2.7 Using here

If a parameter is a deterministic function of other parameters expressed at the same group in the key/-value mapping chain, one does not want to duplicate a value, as giving a new value would require multiple changes. Instead, one can use the `here` mechanism to query the same group of key/value mappings that `here` was called:

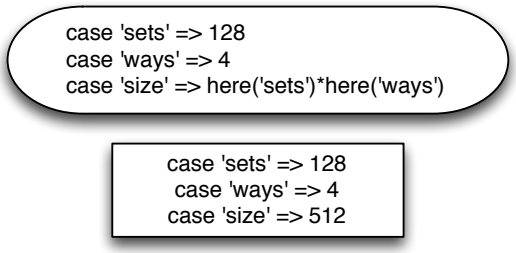


Figure 3: Instead of using 128 or 4 directly, we can access it via here(Sets) and here(Ways), respectively.

```
class Tile extends Module {
  val cache_params = params.alter(
    (pname, site, here, up) => pname match {
      case Sets => 128
      case Ways => 4
      case Size => here(Sets)*here(Ways)
    })
  val cache = Module(new Cache)(cache_params)
}
```

2.8 Using up

The `up` mechanism enables the user to query the parent group of key/value mappings. It is equivalent to calling `Parameters.apply` directly, but can be done within calling `Parameters.alter`. For an example use, see Section **??**.

3 Examples

The three goals of any parameterization scheme are: (1) all searchable parameters are exposed at the top level; (2) source code must never change when evaluating different points; (3) adding new parameters requires little source code change. After each example is described, we present the simplest parameterization scheme that supports the desired design space without violating any of the three goals. As examples grow in complexity, so too must the simplest parameterization scheme, until we arrive at the current advanced parameterization method.

3.1 Simple Parameters

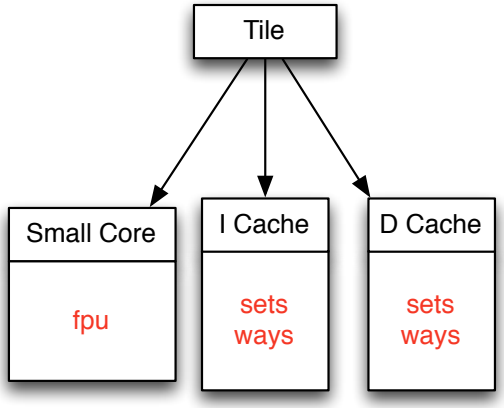


Figure 4: For a few number of parameters, the simplest scheme is to pass them directly via constructor arguments.

In this simple design, we only vary core and cache-specific parameters. The most straightforward parameterization scheme is passing all parameters via arguments to `Tile`'s constructor. These values are then passed to `Core` and `Cache` via their respective constructors:

```
class Tile (val fpu:Boolean, val ic_sets:Int, val
     ic_ways:Int, val dc_sets:Int, val dc_ways:Int)
     extends Module {
  val core = Module(new Core(fpu))
  val icache = Module(new Cache(ic_sets,ic_ways)
  val dcache = Module(new Cache(dc_sets,dc_ways))
  ...
}
class Core (val fpu:Boolean) {...}
class Cache(val sets:Int, val ways:Int) extends Module
     {...}
```

No source code changes are necessary to explore our parameter space, and all searchable parameters are exposed at the top. In addition, adding a new parameter, because this example is simple, requires very few changes to our source code.

3.2 Disjoint Parameter Sets

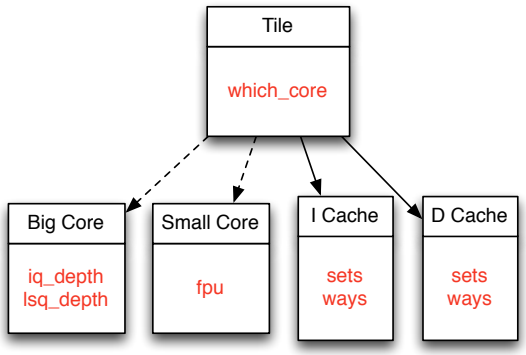


Figure 5: For disjoint parameter sets, we can group sets of parameters into configuration objects to pass as constructor arguments.

In this next design, we are designing a chip which could instantiate different cores, each with its own set of parameters. If we apply our simple solution, the number of arguments to `Tile`'s constructor would be huge as it must contain all parameters for all possible cores (which would likely be much greater than two cores!).

One could propose that a better solution would be to group parameters into configuration objects. For example, we could group all `BigCore` parameters into a `BigCoreConfig` case class, and all `SmallCore` parameters into a `SmallCoreConfig` class, both of which extend `CoreConfig`. In addition, we have our caches and `Tile` accept a `CacheConfig` and `TileConfig`, respectively, within their constructors.

```
abstract class CoreConfig {}
case class BigCoreConfig(iq_depth:Int, lsq_depth:Int)
     extends CoreConfig
case class SmallCoreConfig(fpu:Boolean) extends
     CoreConfig
case class CacheConfig(sets:Int, ways:Int)
case class TileConfig(cc:CoreConfig, icc:CacheConfig,
     dcc:CacheConfig)

class Tile (val tc:TileConfig) extends Module {
  val core = tc.cc match {
    case bcc:BigCoreConfig => Module(new
         BigCore(tc.bcc))
    case scc:SmallCoreConfig => Module(new
         SmallCore(tc.scc))
```

```
  }
  val icache = Module(new Cache(tc.icc)
  val dcache = Module(new Cache(tc.dcc))
  ...
}
...
```

3.3 Location-Independent Parameters

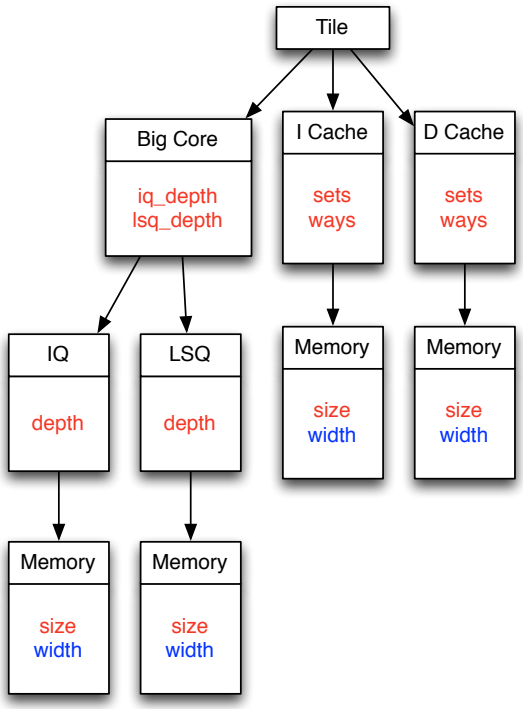


Figure 6: For location-independent parameters, every module has a parameter dictionary, which they can copy and alter before passing to each child module.

The subtle reason why nested configuration objects are extremely brittle is the structure of the nested configuration objects encodes the module hierarchy. Given a new design described in Figure **??**, we assume that `BigCore`'s IQ and LSQ, as well as the icache and dcache, instantiate a `Memory` module. This Memory module contains a `width` parameter, and in order for the design to function correctly, all of Memory widths must be set to the same value. To ensure this requirement, the follow code might be written:

```
case class MemConfig(size:Int, banks:Int, width:Int)
case class CacheConfig(sets:Int, ways:Int, mc:MemConfig)
case class QueueConfig(depth:Int, mc:MemConfig)

case class BigCoreConfig(iqc:QueueConfig,
     lsqc:QueueConfig, mc:MemConfig)

case class TileConfig(cc:CoreConfig, icc:CacheConfig,
     dcc:CacheConfig)

class Tile (val tc:TileConfig) extends Module {
  val core = tc.cc match {
    case bcc:BigCoreConfig => Module(new
         BigCore(tc.bcc))
    case scc:SmallCoreConfig => Module(new
         SmallCore(tc.scc))
  }
  val icache = Module(new Cache(tc.icc)
  val dcache = Module(new Cache(tc.dcc))

  require(tc.dcc.mc.width == tc.icc.mc.width)
  require(tc.bcc.iqc.mc.width == tc.bcc.lsqc.mc.width)
  require(tc.dcc.mc.width == tc.bcc.lsqc.mc.width)
  ...
}
...
```

The series of require statements is extremely brittle, as any change in our design's hierarchy requires massive rewrites of all of these statements. Omitting the require statements is not a viable option; these statements are necessary to enforce this fundamental design requirement.

This flaw in configuration objects leads us towards the first functionality of our custom parameterization solution, namely a copy/alter dictionary of type `Parameters`. We use this key-value structure (map or dictionary) to store a module's parameters.

To parameterize the design in Figure **??**, we implicitly pass the `Parameters` object and, if an alter is needed, provide a `PartialFunction` to the `Module` factory. Recall from Section **??** that the class `MyConfig` (extends `ChiselConfig`) must be passed to the *Chisel* compiler via the `--configInstance` flag to seed the top-level parameters:

```
class DefaultConfig() extends ChiselConfig {
  val top:World.TopDefs = {
    (pname,site,here) => pname match {
      case IQ_depth => 10
      case LSQ_depth =>10
      case Ic_sets => 128
      case Ic_ways => 2
      case Dc_sets => 512
      case Dc_ways => 4
      case Width => 64
      // since any module querying Width should return
           64, the name should NOT be unique to modules
    }
  }
}
class Tile extends Module {
  val core = Module(new Core)(params)
  val ic_sets = params(Ic_sets)
  val ic_ways = params(Ic_ways)
  val icache = Module(new Cache, {case Sets => ic_sets;
       case Ways => ic_ways})
```

```
  // we can rename Ic_sets to Sets, effectively
       isolating Cache's query keys from any design
       hierarchy dependence
  val dc_sets = params(Dc_sets)
  val dc_ways = params(Dc_ways)
  val dcache = Module(new Cache, {case Sets => dc_sets;
       case Ways => dc_ways})
  // similarly we rename Dc_sets to Sets and Dc_ways to
       Ways
}
class Core extends Module {
  val iqdepth = params(IQ_depth)
  val iq = Module(new Queue, {case Depth => iqdepth})
  val lsqdepth = params(LSQ_depth)
  val lsq = Module(new Queue, {case Depth => lsqdepth})
  ...
}
class Queue extends Module {
  val depth = params(Depth)
  val mem = Module(new Memory,{case Size => depth})
  ...
}
class Cache extends Module {
  val sets = params(Sets)
  val ways = params(Ways)
  val mem = Module(new Memory,{case Size => sets*ways})
}
class Memory extends Module {
  val size = params(Size)
  val width = params(Width)
}
```

Although this parameterization method is reasonably verbose, it scales well with adding parameters, requires no source changes, and allows a single parameter, such as `Width`, to change all leaf modules.

3.4 Location-Specific Parameters

As we saw in the previous section, copying and altering a `Parameters` object can be verbose. If we wanted to add an ECC parameter to our `Memory` module, which depends on where the `Memory` is instantiated, we would change source code in multiple parents to rename each parameter (e.g. ECC_icache => ECC).

In the example depicted in Figure **??**, we instead use the `site` functionality of our `Parameters` object to obtain location-specific information, and tailor the value we return to that location-specific value. After adding the location-specific information, we drastically reduce the amount of code changes necessary:

```
class DefaultConfig() extends ChiselConfig {
  val top:World.TopDefs = {
    (pname,site,here) => pname match {
      case Depth => site(Queue_type) match {
        case 'iq' => 20
        case 'lsq' => 10
      }
      case Sets => site(Cache_type) match {
        case 'i' => 128
```

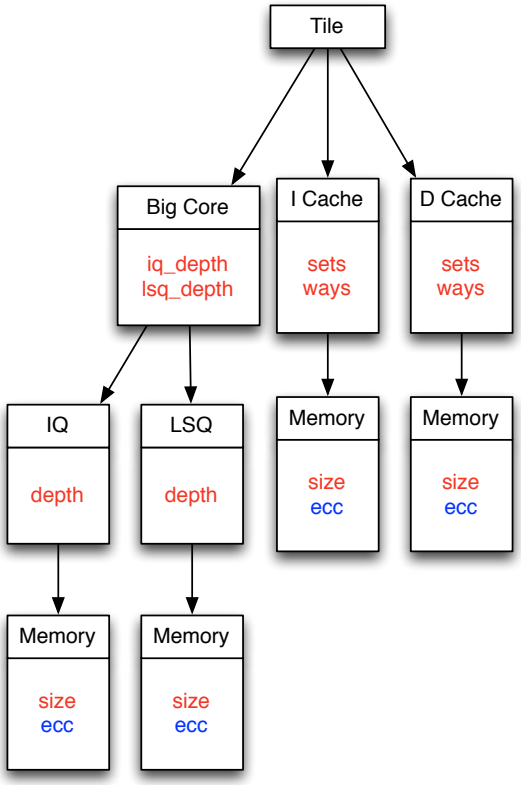


Figure 7: For location-dependent parameters, we can use the `site` mechanism to customize these parameters at the top level.

```
        case 'd' => 512
      }
      case Ways => site(Cache_type) match {
        case 'i' => 2
        case 'd' => 4
      }
      case Width => 64
      // since any module querying Width should return
           64, the name should NOT be unique to modules
      case ECC => site(Location) match {
        'incore' => false
        'incache' => true
      }
    }
  }
}
class Tile (val params:Parameters) extends Module {
  val core = Module(new Core,{Location => 'incore'})
  // we can give core and its child modules a location
       identifier

  val cacheparams = params.alter({Location =>
       'incache'})
  // we can give both caches and all their child
       modules a location identifier
  val icache = Module(new ICache)(cacheparams)
  val dcache = Module(new DCache)(cacheparams)
}
```

```
class Core extends Module {
  val iq = Module(new IQ)
  val lsq = Module(new LSQ)
  ...
}
class IQ extends Module {
  val depth = params(Depth)
  val mem = Module(new Memory, {Size = depth})
  // in some cases, using copy/alter is preferred
       instead of \code{site} (see Design Heuristics
       for more details)
  ...
}
class LSQ extends Module {
  val depth = params(Depth)
  val mem = Module(new Memory, {Size = depth})
  ...
}
class ICache extends Module {
  val sets = params(Sets)
  val ways = params(Ways)
  val mem = Module(new Memory,{Size => sets*ways})
}
class DCache extends Module {
  val sets = params(Sets)
  val ways = params(Ways)
  val mem = Module(new Memory, {Size => sets*ways})
}
class Memory extends Module {
  val size = params(Size)
  val ecc = params(ECC)
}
```

3.5 Derivative Parameters

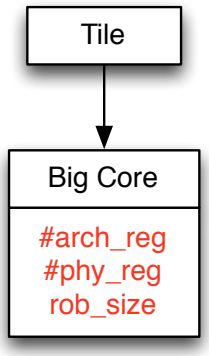


Figure 8: To derive a parameter from another top-level parameter, we can use the here functionality to avoid duplicating a parameter value.

In Figure **??**, we always want our ROB to be four-thirds the size of the difference between the number physical registers and the number of architectural registers. If we express this in `MyConfig.top`, it could look like the following:

```
case object NUM_arch_reg extends Field[Int]
case object NUM_phy_reg extends Field[Int]
case object ROB_size extends Field[Int]
class DefaultConfig() extends ChiselConfig {
  val top:World.TopDefs = {
    (pname,site,here) => pname match {
      case NUM_arch_reg => 32
      case NUM_phy_reg => 64
      case ROB_size => 4*(64-32)/3
  }
}
```

However, if we later increase the number of physical registers, we need to remember to update the value in the derivation of the ROB size. To avoid this potential error, one should use the 'here' functionality to query the same group of parameters:

```
class DefaultConfig() extends ChiselConfig {
  val top:World.TopDefs = {
    (pname,site,here) => pname match {
      case NUM_arch_reg => 32
      case NUM_phy_reg => 64
      case ROB_size => 4*(here(NUM_phy_reg) -
           here(NUM_arch_reg))/3
  }
}
```

3.6 Renaming Parameters

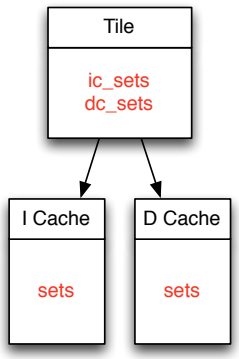


Figure 9: To rename or programmatically alter a parameter based on the previous value, one can use the up mechanism to query the parent's `Parameters` object.

In Figure **??**, both cache modules query for a `sets` parameter. However, `Tile` has `ic_sets` and `dc_sets` as parameters. To rename the parameters, we can read the parent value and alter the child's `Parameters` object:

```
class Tile extends Module {
  val ic_sets = params(Ic_sets)
  val ic = Module(new Cache,{case Sets => ic_sets})
```

```
  val dc_sets = params(Ic_sets)
  val dc = Module(new Cache,{case Sets => dc_sets})
    ...
}
```

Alternatively, we can use the 'up' mechanism within the Parameters.alter method to query the parent module's Parameter object:

```
class Tile extends Module {
  val ic_params = params.alter(
    (pname,site,here,up) => pname match {
      case Sets => up(Ic_sets)
    }
  )
  val ic = Module(new Cache)(ic_params)
  ...
}
```

In general one should never use the `up` mechanism as it is more verbose. However, it can be useful if the parent is making significant changes to a child's `Parameters` object, as all changes can be contained the `Parameter.alter` method because one has access to all three central mechanisms (`up`, `site`, and `here`).

4 External Interface

So far, this document has only describe mechanisms to manipulate parameters at a top-level class (`ChiselConfig`). However, to actually generate multiple C++ or Verilog designs, we need to manually change these parameters.

One would prefer to express design constraints (parameter ranges, dependencies, constraints) and leave the actual instantiation of a specific design separate from the expression of the valid design space.

With that motivation, *Chisel* has an additional feature based around the concept of "Knobs," or parameters that are created specifically to explore a design space. This section will describe Knobs and their uses, the Dump Object, adding constraints to parameters/Knobs, and the two modes to running the Chisel compiler: –configCollect and –configInstance.

4.1 Knobs

A generator has some parameters that are fixed, and others that dictate the specific design point being generated. These generator-level parameters, called Knobs, have an additional key-value mapping to allow external programs and users to easily overwrite their values.

Knobs can only be instantiated within a `ChiselConfig` subclass's `topDefinitions`:

```
package example
class MyConfig extends ChiselConfig {
  val topDefinitions:World.TopDefs = {
    (pname,site,here) => pname match {
      case NTiles => Knob('NTILES')
      case .... => .... // other non-generator
           parameters go here
    }
  }
  override val knobValues:Any=>Any = {
    case 'NTILES' => 1 // generator parameter assignment
  }
}
```

When the query `NTiles` matches within topDefinitions, the `Knob('NTILES')` is returned. Internally, Chisel will lookup `'NTILES'` within MyConfig.knobValues and return 1. As described in Section **??**, the flag required to execute a generator with this specific config is:

```
sbt run ...  --configInstance example.MyConfig
```

Suppose we wanted to instantiate a new design that had two tiles: simply use Scala's class inheritance and overwrite the knobValues method:

```
package example
class MyConfig2 extends MyConfig {
  override val knobValues:Any=>Any = {
    case 'NTILES' => 2 // will generate new design with
         2 tiles
  }
}
```

Notice that both classes can exist in the source code, so both designs can be instantiated from the commandline. For the new design with two tiles, simply call:

```
sbt run ...  --configInstance example.MyConfig2
```

4.2 Dump

Downstream from Chisel, other tools might need to know specific parameter/Knob assignments. If so, just pass the Knob/value to the Dump object, which will write the name and value to a file, then return the Knob/value:

```
package example
class MyConfig extends ChiselConfig {
  val topDefinitions:World.TopDefs = {
    (pname,site,here) => pname match {
      case Width => Dump('Width',64) // will return 64.
           Requires naming the parameter as the 1st
           argument
      case NTiles => Dump(Knob('NTILES')) // will
           return Knob('NTILES'), no name needed
    }
  }
  override val knobValues:Any=>Any = {
    case 'NTILES' => 1 // generator parameter assignment
  }
}
```

The name and value of each dumped parameter will be written to a `*.knb` file located in the directory set by `--targetDir` *path*.

4.3 Constraints

Now that external programs/users can easily overwrite a configuration's `knobValue` method, we have provided a mechanism for defining legal ranges for Knobs. Within a `ChiselConfig`, one can overwrite another method called `topConstraints`:

```
package example
class MyConfig extends ChiselConfig {
  val topDefinitions:World.TopDefs = {
    (pname,site,here) => pname match {
      case NTiles => Knob('NTILES')
    }
  }
  override val topConstraints:List[ViewSym=>Ex[Boolean]]
    = List( { ex => ex(NTiles) >  0 },
            { ex => ex(NTiles) <= 4 })
  override val knobValues:Any=>Any = {
    case 'NTILES' => 1 // generator parameter assignment
  }
}
```

Now, if someone tried to instantiate our design with the following configuration and command, it would fail:

```
package example
class BadConfig extends ChiselConfig {
  override val knobValues:Any=>Any = {
    case 'NTILES' => 5 // would violate our constraint,
         throws an error
  }
}

// throws 'Constriant failed' error
sbt run ... --configInstance example.BadConfig
```

Constraints can be declared anywhere in the design, not just at the top level, by calling a Parameter's `constrain` method:

```
package example
class MyConfig extends ChiselConfig {
  val topDefinitions:World.TopDefs = {
    (pname,site,here) => pname match {
      case NTiles => Knob('NTILES')
    }
  }
  override val knobValues:Any=>Any = {
    case 'NTILES' => 1 // generator parameter assignment
  }
}
class Tile extends Module {
  params.constrain( ex => ex(NTiles) >  0 )
  params.constrain( ex => ex(NTiles) <= 4 )
}
object Run {
  def main(args: Array[String]): Unit = {
    chiselMain.run(args, () => new Tile())
  }
}

sbt runMain example.Run ... --configInstance
     example.MyConfig
```

Finally, if a designer wants to know a design's constraints, they can execute Chisel with the `--configCollect` *project_name.config_name* flag, which will dump a list of the constraints to a `*.cst` file, located in the path specificed by `--targetDir` *path*:

```
sbt runMain example.Run ... --configCollect
     example.MyConfig --targetDir <path>
```

5 Design Heuristics

TODO

References

[1] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, Wawrzynek, J., Asanović *Chisel: Constructing Hardware in a Scala Embedded Language* in DAC '12.

[2] Odersky, M., Spoon, L., Venners, B. *Programming in Scala* by Artima.

[3] Payne, A., Wampler, D. *Programming Scala* by O'Reilly books.