Contents

List of Figures

1 Introduction

Chisel [2] is designed to be a *hardware construction language* so any design in Chisel will always have a well defined mapping to low level hardware blocks. Chisel's core programming features allow users to accurately describe hardware circuits and can be easily extended for higher level hardware designs. This paper categorizes Chisel hardware construction into three levels and discusses the merits and drawbacks of each level.

We first introduce the core features of the Chisel hardware construction language in Chapter 2. All hardware design in Chisel boils down to writing a Scala program that builds up a directed graph of `Nodes` which are Scala objects that map to basic circuit elements. Chisel provides basic operators for constructing and connecting nodes into subgraphs. The subgraphs can be organized into `Components` which can be connected to other `Components` to form larger and more complex graphs. Chisel code can be mapped to many different backends. We currently support a C++ backend for high speed simulation and a Verilog backend to leverage existing ASIC tools.

Chapter 3 presents the lowest level of hardware construction in Chisel. We consider this the lowest level since Chisel code is written at the granularity of Chisel's most basic element, `Nodes`. At this hardware construction level, Chisel user code looks identical to the targeted backend code. This can be advantageous if the backend has syntax that the tools can use to synthesize efficient hardware. The downside to this is that syntax in one backend might not have a parallel in another backend.

Moving up the abstraction ladder, Chapter 4 presents the next level of Chisel hardware construction in which the user writes code using Scala functions that generate hardware. Although the Chisel code might look similar to Chisel from Chapter 3, the resulting Chisel graph is much denser since this hardware construction level only uses the basic features of Chisel. Higher level hardware facilities are abstracted away into Scala functions that implement the same functionality using only basic Chisel `Nodes`. Although we no longer leverage backend tool support for special syntax, this approach has the advantage of decoupling Chisel code from the backends.

The last level of hardware construction, Chapter 5, analyzes and transforms Chisel graphs to produce different Chisel graphs. This hardware construction level is similar to the hardware construction level of Chapter 4 in that users still write Scala programs to build hardware. The difference is that these functions are invoked during elaboration time to modify a graph instead of during runtime to build the graph. This hardware construction level is useful for circuit designs that are difficult or undesirable to build during runtime.

1.1 Related Works

The most widely used hardware description languages (HDL), Verilog and VHDL, are actually ill-suited for designing efficient, high-performance hardware. These languages lack the powerful abstraction facilities that are readily available in modern programming languages. Without these facilities, hardware designers using these languages are relegated to writing undescriptive and non-reusable hardware modules which dramatically impedes the most important aspect of building efficient hardware: design space exploration.

One way to circumvent this issue is to design hardware from within a specific domain. Esterel [3] is synchronous programming language that uses event-based statement to describe reactive systems. Bluespec [4] uses guarded atomic actions to describe state change. Spiral [5] generates hardware for linear signal transformations from an input problem description. Although these languages allow designers to write more expressive hardware, they only work well when used for task that matches their model and will fall short when used for general purpose hardware construction.

1.2 Acknowledgments

The Chisel HDL is an ongoing project. Jonathan Bachrach is the original author. I joined the project in 2011 where I worked with Jonathan and Brian Richards to port an existing FPU library from Verilog to Chisel. I continued working with Jonathan to implement Chisel's type system. In 2012 I work with Yunsup Lee to port an existing vector machine to Chisel. During this time I developed Vec to better express the hardware in the vector machine. I eventually moved Vec into Chisel's library. Since then Jonathan has added support for bidirectional wires to Vec and Andrew Waterman has added new functionality to Vec such as `contains` and `forAll` methods. Jonathan is responsible for the original implementation of Chisel's `when` statement. Andrew implemented Chisel's elaboration time interface. Yunsup, Andrew, and I have written various SRAM backends for the chips we built. The automatic pipeline synthesis tool is CS294-88 class project that I developed with Wenyu Tang.

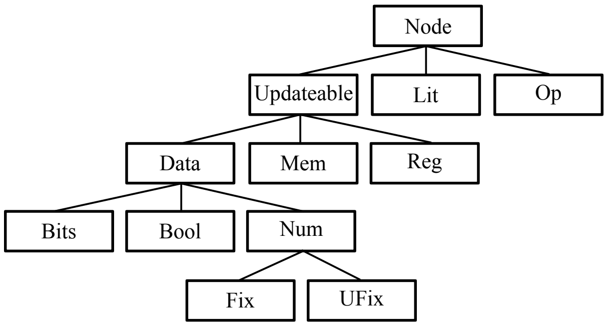
2 Chisel

Chisel is a domain-specific language embedded in the Scala programming language [7] so it is really just a library of Scala functions and data structures. Someone writing Chisel code is essentially writing a Scala program that uses the provided functions and data structures to construct hardware. This chapter introduces the core components of the Chisel hardware construction language; readers interested in a more a detailed specification of Chisel should consult the Chisel manual [1] which can be found on the Chisel website.

2.1 Node

Hardware circuits are represented in Chisel as directed graphs of `Nodes`, which are the base class from which all circuit elements are derived. Figure 1a shows the Chisel class hierarchy and Figure 1b shows the API for the Node class.

Types. Chisel has its own type system that is maintained independent of the Scala type system. Figure 1a shows the Chisel data types and their hierarchy (everything rooted at `Data`). `Bits` is used to represent raw collection of bits, `Bool` is used to represent Boolean literals, and `Num` is used to represent numbers. Note that `Num` has two subclasses, `Fix` and `UFix` which are used for signed and unsigned numbers respectively.



(a) Hierarchy

```
abstract class Node {
  // user assigned name of Node
  val name: String
  // incoming edges
  val input: ArrayBuffer[Node]
  // width inferance
  def inferWidth: Int
}
```

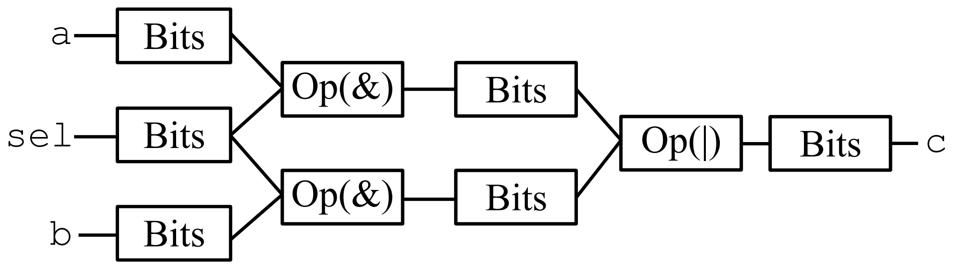
(b) API

Figure 1: Chisel Node

Arithmetic and Logic. Arithmetic and logic operations are represented with `Op` nodes. The Chisel manual has an exhaustive list of supported operations. Op nodes are constructed whenever a user invokes the corresponding type node member function for the operation. For example, suppose we have two `Bits` nodes, `a` and `b`, and want to connect them to the inputs of an AND gate. The Chisel code that will do this is `a & b` which is actually syntactic sugar for `a.&(b)`. The `&` function that Bits defines will instantiate an `Op` node to represent the `&` and put the two `Bits` nodes, `a` and `b`, into the `Op` node's input.

The example Chisel code in Figure 2 shows Bits nodes and Op nodes interacting to create a 2-input multiplexor built out of `&` and `|` gates. In this example, `a`, `b`, and `sel` are incoming `Bits`. Invoking `a & sel` will create a new `Op` and connect that node to a `Bits` since both inputs are `Bits`. The output of the circuit, `c`, will be the `Bits` connected to the `| Op`.

Literals The `Lit` node works in conjunction with type `Nodes` to represent literals. Figure 3 shows example Chisel code for instantiating literals. True and false literals are represented as Bools



```
val c = a & sel | b & ~sel
```

Figure 2: 2-Input Mux

connected to `Lit` nodes 1 and 0 respectively. Arbitrary bit strings can be represented with `Bits` Nodes and numbers can be represented with `Fix` and `UFix` Nodes, all of which are connected to `Lit Nodes` with the appropriate value.

Registers The `Reg` node, Chisel most basic state element, represents a positive-edge-triggered flip-flop. `Reg` nodes, unlike the previous nodes, are manually instantiated by the user. The manual details the different ways that a Reg can be constructed. Briefly, Regs can be constructed either with or without an update signal. `Reg`'s constructed without an update signal can be conditionally updated.

Memories `Mem Nodes` are used to represent random access memories. These memories can be accessed with either `Bits` or `UFix` nodes. Writes to these memories are positive-edge-triggered while reads can be either combination or positive-edge-triggered.

2.2 Component

In addition to `Nodes`, Chisel also provides `Components` to allow users to organize their directed graph of `Nodes` into subgraphs, adding a hierarchical structure to their hardware. Figure 4 shows an example of a Chisel implementation of a 4-input mux built from three 2-input muxes.

Interfaces When defining a new Component, users have to fill out the `io` field which is the input-output interface that Components expose to each other. An input or output node in Chisel is represented with a type `Node` where the direction field is set to either `INPUT` or `OUTPUT`. The example in Figure 2 shows Component interfaces built out of `Bits` nodes. Input and output nodes can be collected into Bundles, which are functionally similar to C structs, for better organization.

Wiring Once a Component's interface and circuit implementation are defined, users will need to connect Components together. Chisel provides two different ways for connecting Components

Lit(0)
Lit(1)
Bool
Bool
Lit(5)
Bits
Lit(6)
UFix
Lit(-3)
Fix

```
val t = Bool(true)
val f = Bool(false)
```

(a) Bool literals

```
val b = Bits("b101")
```

(b) Bits literals

```
val u = UFix(6)
```

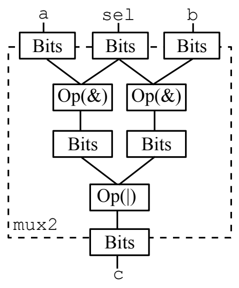
(c) UFix literals

```
val u = Fix(-3)
```

(d) Fix literals

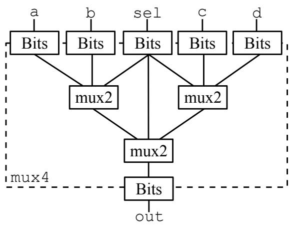
Figure 3: Chisel Literal Examples

```
class mux2 extends Component {
  val io = new Bundle {
    val a = Bits(INPUT, 1)
    val b = Bits(INPUT, 1)
    val sel = Bits(INPUT, 1)
    val out = Bits(OUTPUT, 1)
  }

  io.out := io.a & io.sel |
            io.b & io.sel
}
```



(a) 2-Input Mux

```
class mux4 extends Component {
  val io = new Bundle {
    val a = Bits(INPUT, 1)
    val b = Bits(INPUT, 1)
    val c = Bits(INPUT, 1)
    val d = Bits(INPUT, 2)
    val sel = Bits(INPUT, 2)
    val out = Bits(OUTPUT, 1)
  }

  val m1 = new mux2
  val m2 = new mux2
  val m3 = new mux2
  // connect the muxes...
}
```



(b) 4-Input Mux

Figure 4: Hierarchical 4-Input Mux

together. := is used for connections going from right to left while <> is used for connections that go in either direction. Both connection operators can be used to connect either individual ports or entire Bundles. If used for the latter, the connection operator will recursively walk through Bundle and wire each element individually.

2.3 Backend

The Backend takes as input a directed graph of `Nodes` and generates code for the specified backend. Figure 5 shows how Chisel's backend generates code from a directed graph of nodes. Chisel currently generates code for two backends: Verilog and C++.

Elaboration The elaboration phase performs backend specific transformation on the graph in order to generate valid code for that backend. For example, it is desirable to preserve Component hierarchy when targeting the Verilog backend, so the elaboration phase will do a graph walk to match nodes into components. For the C++ backend, the elaboration phase will transform the graph into a directed acyclic graph and perform a topological sort.

The elaboration phase also performs transformations and checks that are common to both backends such as width inference and combinational loop detection.

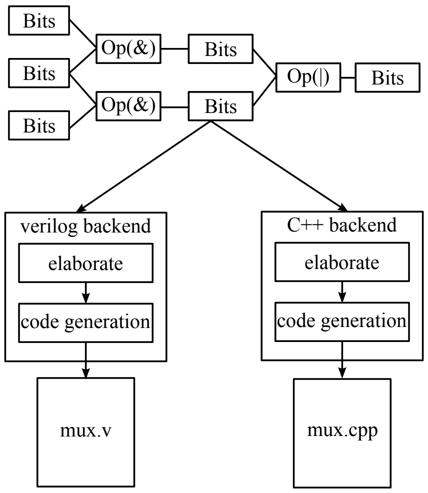
Code Generation Figure 5b shows the API for generating code from an elaborated graph of nodes. The `emitDec` function is called to generate the code for declaring the node, `emitRef` is called to generate the reference to a node, and `emitDef` is called to generate the code. As an example, suppose the Verilog backend is generating code for an `Op` node that performs an add. The `emitDec` function, when called, returns the string

```
wire [31:0] a;
```

assuming that the `Op` is 32-bits wide and its user assigned name is `a`. The `emitDef` function, when called, returns the string

```
assign a = b + c;
```

assuming that the two nodes in the `Op`'s input list have names `b` and `c`.



(a) Elaboration and Code Generation Process

```
abstract class Backend {
  // emit name
  def emitRef(n: Node): String
  // emit declaration
  def emitDec(n: Node): String
  // emit code
  def emitDef(n: Node): String
}
```

(b) Backend API

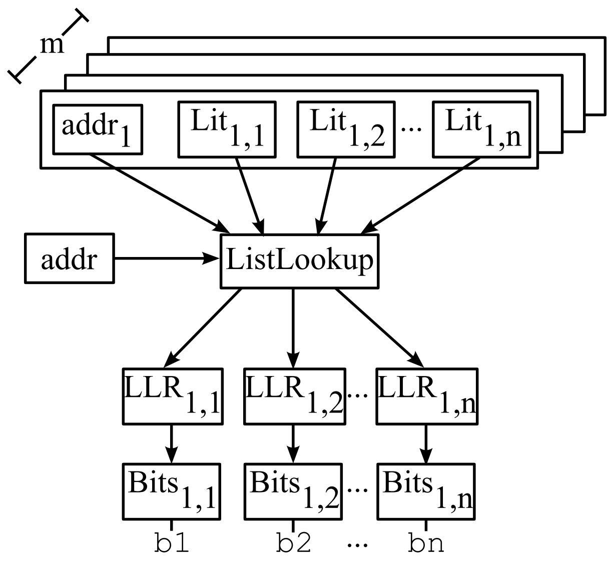
Figure 5: Chisel Backend

3 Basic Hardware Construction in Chisel

The Chisel constructs presented in the previous chapter are adequate for most hardware circuits. We have in fact built an entire floating point library using only those constructs. Occasionally, users might want to leverage syntax for hardware facilities available in a target backend from within Chisel. Suppose for example that a user is having Chisel target a backend that has support for highly optimized FFT circuits. One way to leverage this facility is to introduce a new type of `Node` into Chisel that maps directly to that syntax in the backend. This approach keeps the Chisel graph sparse since all the hardware construction complexity is pushed into the backends where the backend synthesis tools could possibly map the hardware facility to very dense circuits. The downside to this approach is that Chisel code using `Nodes` designed to target these backend hardware facilities now have a strong dependence on that backend. Furthermore, targeting a new backend would require effort to port the Chisel user code if the new backend does not have built-in support for the hardware facility.

The nuts and bolts to pay attention to when implementing a new `Node` are the node's code generation functions and the node's inputs list. Let's say we are implementing a new `Node` called `Foo`. If `Foo`'s code generation function need to reference nodes `B`, `A`, and `R`, then all three of those nodes must go into `Foo`'s input list. Omitting a node can lead to a scenario where Chisel is unable to reach the omitted node and thus never emits the backend code for that node. The result is `Foo`'s backend code will make reference to non-existing hardware. This chapter will show how `ListLookup` and `Vec` nodes can be implemented to target Verilog's case statements and arrays.

3.1 ListLookup



(a) ListLookup Node. Lit = `Literal` node, LLR = `ListLookupRef` nodes, and addr is shorthand for a `Bits` node used as an address. Boxes with (`addr`$_j$, `lits`$_{i,j}$) represent Scala tuples.

```
always @ (*) begin
  casez(addr)
    addr_1: begin
      b1 = lit_{1,1}
      b2 = lit_{1,2}
         ...
      bn = lit_{1,n}
    end
    addr_2: begin
      b1 = lit_{2,1}
      b2 = lit_{3,2}
         ...
      bn = lit_{3,n}
    end
       ...
end
```

(b) Generated ListLookup Code. There are m generated case statements each with n literal assignments.

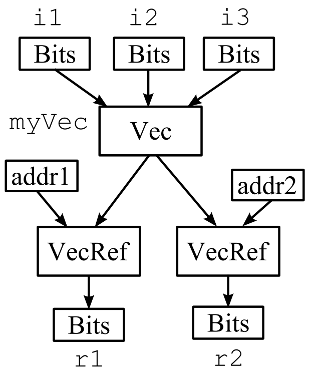
Figure 6: ListLookup as Chisel Node

We introduce `ListLookup` and `ListLookupRef` nodes to target Verilog's case statement from within Chisel which is useful for writing decode tables.

ListLookup Node.The inputs to this node are an address, a default list of literals, and a list of tuples $(\texttt{a}_i,\ \texttt{lits}_i)$ where $\texttt{lits}_i$ is a list of a literals. Figure 6a shows how the inputs to the `ListLookup` nodes are managed. This node is different from other nodes in that it does not have an entry for `emitDec` and `emitRef`. The only code generation function it fills out is the `emitDef` function which generates the case statements in Figure 6b. The ListLookup generates a case statement for every $(\texttt{a}_i,\ \texttt{lits}_i)$ tuple where each case statements will have an assignment for each literal in $\texttt{lits}_i$. The address input to the ListLookup is used as the lookup address into the case statement and the $\texttt{a}_i$'s are used to match against the address. If no address matches, the default assignment is used.

ListLookupRef Node. Creating a new `ListLookup` will not return the `ListLookup` to the user, rather, a list of `ListLookupRef` nodes are returned instead. The length of the `ListLookupRef` list matches the length of $\texttt{lits}_i$. `ListLookupRef` nodes do not have an entry for `emitDef` since the assignments to these nodes are actually done in the `emitDef` of a `ListLookup` node.

3.2 Vec



(a) Vec Node. Example three entry `Vec` with two read ports.

```
wire [31:0] myVec [0:2];
wire [31:0] r1;
wire [31:0] r2;

assign myVec[0] = i1;
assign myVec[1] = i2;
assign myVec[2] = i3;
assign r1 = myVec[addr1];
assign r2 = myVec[addr2];
```

(b) Generated Vec Code. The two `VecRef` nodes map to array accesses in Verilog.

Figure 7: Vec as Chisel Node

We use `Vec` and `VecRef` nodes to target 2-D Verilog arrays from within Chisel. Vecs are useful for organizing small collection of wires into an indexable structure.

Vec Node. The `Vec` node maps directly to the 2-D Verilog array. Users instantiate this node with the desired number of entries and then populate the node using := assignments. Figure 7a shows a 3-entry `Vec` where the user has connected `i1-3` to the `Vec`. The width of a `Vec` node is inferred to be the maximum width of its inputs (in this example, 32-bits wide). `Vec`'s `emitDec` function will return the string

```
wire [width-1:0] vec_name [0:entries-1];
```

where `vec_name` is the name of the node, `width` is the inferred width, and `entries` is the number of entries in the `Vec`. The `emitDef` function emits an assign statement for each entry in the Vec.

VecRef Node. After instantiating and populating the Vec, users will want to access the `Vec` with dynamic addresses. Each access with a new address will make a new `VecRef` node. The inputs to a `VecRef` node are an address and a `Vec` and its inferred with is the width of its input `Vec`. Figure 7a two `VecRef` nodes made from accessing the same `Vec` with two different addresses. `VecRef`'s `emitDec` simply declares the name and width of the node. Its `emitDef` entry will return the string

```
assign ref_name = vec_name[addr];
```

where `ref_name` is the name of the `VecRef` node, `vec_name` is the name of the `Vec` and `addr` is the address.

4 Advanced Hardware Construction in Chisel

Another way to support hardware facilities in Chisel, as opposed to building new nodes, is to write a Scala program that aggregates the features of Chisel from Chapter 2 into a directed graph of nodes that implement the same hardware functionality. This approach moves the complexity of the hardware facility into the Scala graph but makes the hardware facility much more accessible. Unlike the approach from Chapter 3, the hardware facilities are now built out of basic Chisel `Nodes` so there is no dependence on backend support and no designer effort is needed to target different backends. Users can fall back to this style of hardware construction if they are targeting a backend that does not have support for the hardware facility they are using. The downside is that the resulting Chisel graph is much denser which can lengthen the time spent in elaboration. We are also giving up on any optimization the backend synthesis tool could provide for supported hardware facilities.

The hardware facilities built in this section will use the same syntax as their counterparts from Chapter 3. The difference now is that we are reusing basic Chisel `Nodes` and not implementing new `Nodes`. This chapter will show how to implement ListLookup and Vec using only the basic Chisel nodes presented in Chapter 2.

4.1 ListLookup

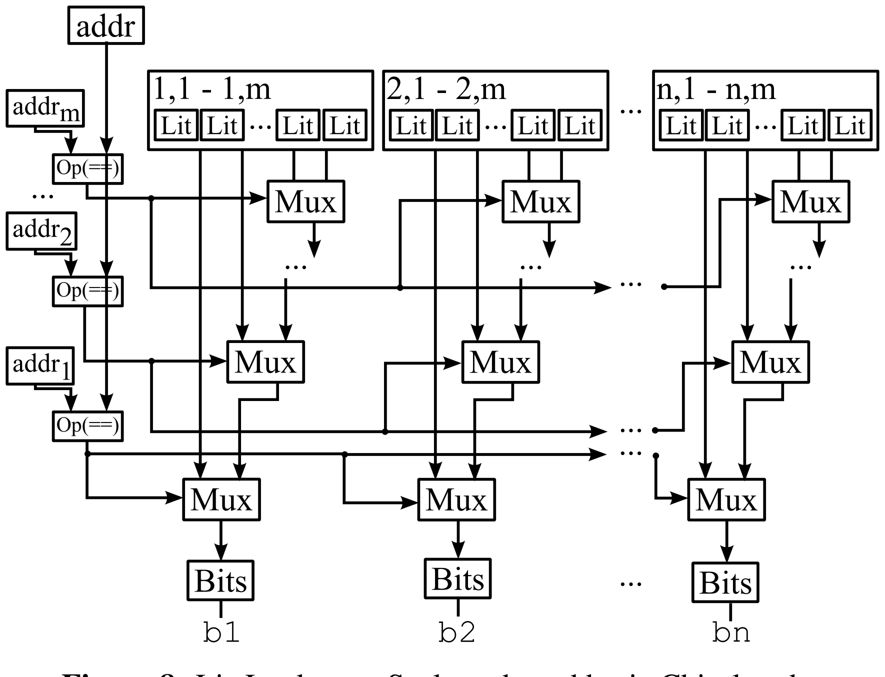


Figure 8: ListLookup as Scala code and basic Chisel nodes

The syntax for ListLookup remains the same. Users provide an address for lookup and a list of size m of tuples `(`a_i`,` $lits_i$`)` where a_i is the address to match the lookup address against and $lits_i$ is a list of size n of literals to return when there is a match. The ListLookup will then construct the graph in Figure 8. The function returns a list of `Bits` b_1`, ...` b_n who will take on the values of $lits_i$ if a_i matches the lookup address.

ListLookup graphs are constructed as follows. Given the input list of tuples $(a_i, \ lits_i)$, create n list of m tuples $((a_1, l_{1,k}), ...(a_j, l_{j,k}), ..., (a_m, l_{m,k}))$. Each of these list represents a mux chain and there are n such list (k ranges from 1 to n). The mux chains are constructed using a_i `===` `addr` as the select signal to the corresponding mux. If the address matches, then literal $l_{i,k}$ is outputted for Bits b_k.

4.2 Vec

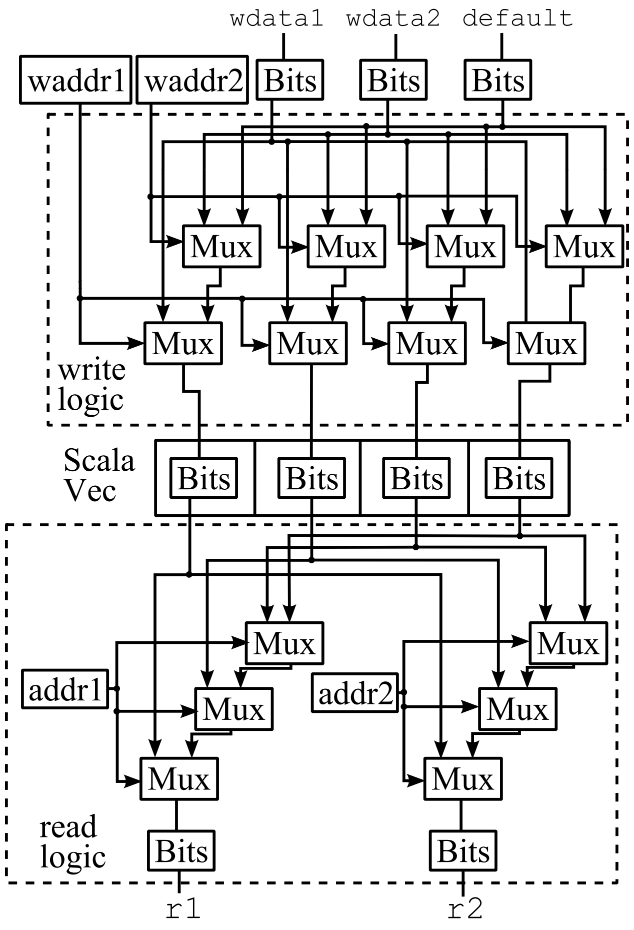


Figure 9: Vec as Scala code and basic Chisel nodes

Figure 9 shows the resulting graph from building a Vec using only basic Chisel nodes. Note that Vec is now a Scala array of `Bits` and is not a `Node`. The syntax for constructing a Vec is unchanged; users instantiate the Vec with the desired size. Since the Vec is now a Scala array, users populate the Vec using Scala array syntax, e.g.

```
v(1) = Bits(width=32)
```

where `v` is the name of the Vec. The syntax for reading and writing to the Vec is also unchanged. The difference is that reads and writes will now construct the graph in Figure 9 instead of instantiating special read and write nodes.

Reads A Vec can be indexed with either Scala integer or a Chisel `Node`. If a Scala integer is used, no `Nodes` are created and the element at that index is returned. If a Chisel `Node` is used, then a Mux chain is built to mux out the correct element of the Vec. The select signal to the mux is `addr === idx` where `addr` is the address and `idx` ranges from zero to length of the Vec minus 1. When `addr` matches an `idx`, the element at `v(idx)` is outputted.

Writes Like reads, a Vec can be written to using either a Scala integer or a Chisel `Node` for indexing. If a Scala integer is used, then the corresponding element is accessed and written to and no nodes are constructed. If a Chisel `Node` is used, then a mux is placed in front of each element in the Vec. The select signal to the mux is `waddr ==== idx` where `waddr` is the input write address and `idx` ranges from zero to the length of the Vec minus 1. If the write address matches the `idx`, then the element at `v(idx)` is written to with the write data, otherwise `v(idx)` is written to with the default data. If the `Vec` contains `Bits` then the user supplies the default data. If the `Vec` contains `Reg` nodes then no default signal is needed.

5 Elaboration Time Hardware Construction

The methods for constructing hardware in the previous chapters are all similar in that the user directly interfaces with them. The user directly constructs the directed subgraph of `Nodes` that implements the desired circuit and connects that subgraph to the main graph. We refer to that class of hardware construction as *runtime* hardware construction since the hardware is constructed as Scala is making a pass over user-written code. This chapter discusses another class of hardware construction that we refer to as *elaboration time* hardware construction since the hardware is constructed during Chisel elaboration. This class of hardware construction is useful since it is not always possible or desirable for the user to construct the circuit as they are building up the Chisel graph. The remainder of this chapter presents the interface for writing elaboration time hardware construction routines and three examples of how to use that interface.

5.1 Elaboration Time Hardware Construction Interface

The Chisel compiler maintains a list of functions that it invokes during elaboration time to refine the input user graph of `Nodes`. These functions must adhere to the interface described in Figure 10. The bodies of the functions in Figure 10a varies from transformation to transformation but will usually follow two phase format. The first phase performs some analysis on the Chisel graph to collect information, e.g. find all Reg nodes. The second phase will then use the information from the first phase to construct a circuit and perform some transformation on the graph, e.g. generate an enable circuit for every Reg node and connect the circuit into the Reg node.

(a) Elaboration Time Functions

```
def tranformA(c: Component): Unit = {
  // body
}

def transformB(c: Component): Unit = {
  // body
}
```

(b) Elaboration Time Hook

```
transforms += transformA
transforms += transformB
```

Figure 10: Elaboration Time Interface

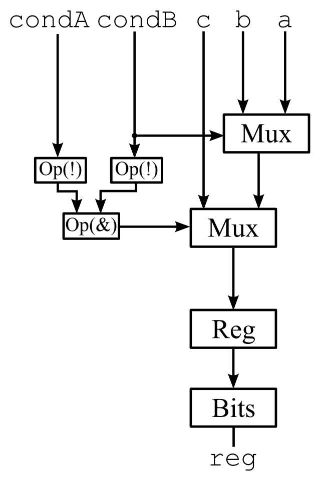
5.2 When

Chisel provides a `when - elsewhen - otherwise` statement for performing conditional updates on `Bits` and `Reg` nodes. This statement appears behavioral but actually constructs a hardware circuit to perform the update. The construction of this circuit is delayed until elaboration time since it is easier to construct the circuit once the user has specified all the updates than it is to construct the circuit as the user is specifying updates.

Runtime. Instead of constructing hardware whenever a `when` statement is executed, we simply record information to be used during elaboration. For this purpose, we maintain a mapping of node to list of update tuples with the format `(cond, bits)` where `cond` is the condition of the when statement and `bits` is the update data. These update tuples are constructed for every update statement within the body of a `when` statement and appended to the list mapped to the node on the

```
when (condA) {
  reg := updateA
} .elsewhen (condB) {
  reg := updateB
} .otherwise {
  reg := default
}
```

(a) When Code



(b) When Graph

Figure 11: When Statement

left hand side of that update statement. Note that the updates in this list do not have to be mutually exclusive. In the case when multiple updates are enabled, Chisel has the semantics that the last update in program order (i.e the last update in the list of updates) takes priority.

Figure 11a shows an example usage of when statements to conditionally update a register. When this statement is executed, it will add 3 update tuples to `reg`'s list of update tuples. The first statement adds the tuple `(condA, updateA)`, the second adds `(condB && !condA, updateB)`, and the third adds `(!condB && !condA, updateC)`.

Elaboration Time. The elaboration time transformation function does not need to perform any analysis since we store all the information for elaboration during runtime in the form of a mapping from node to list of update tuples. For each entry in this mapping, we construct a chain of `Mux` nodes from the list of update tuples and connect the output of that chain into the input of the node in the mapping. If the node in the mapping is a `Reg` node then nothing is done for the case when no updates are enabled. Otherwise, the node in the mapping is a `Bits` node and we have an additional `Mux` node to mux out a default signal provided by the user. If no default signal is provided, Chisel reports an error to the user.

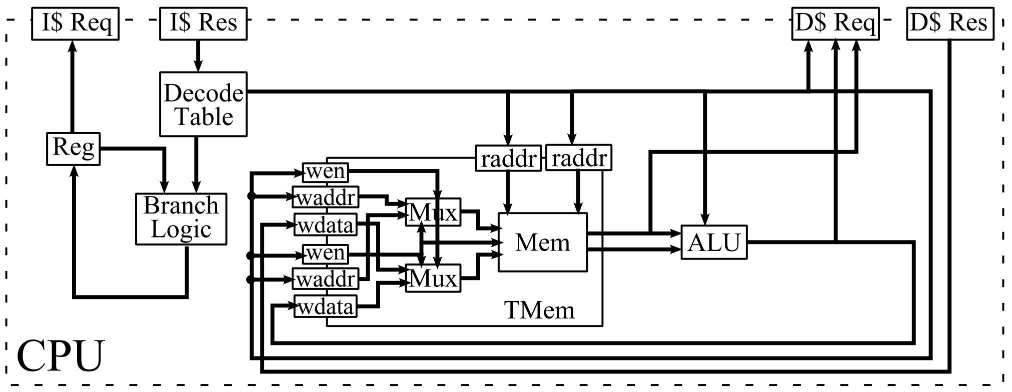
5.3 SRAM Backend

Users targeting the Verilog backend will want to map Mems to SRAMs with backend specific SRAM interfaces. One way to target these SRAM interfaces is to encode the interface into the Chisel user code. This has the disadvantageous effect of muddling the code with SRAM specific information and makes it an unnecessarily tedious to compile the Chisel for a different SRAM backend.

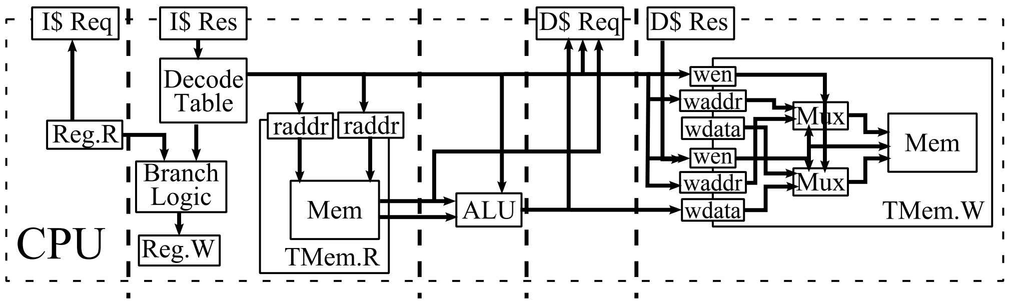
A better way to target backend specific SRAM interfaces is to write an elaboration time function that transforms the Chisel graph to target specific SRAM interfaces. Let's say we want to target an SRAM that has an `init` pin which needs to be wired to the top level module. We first perform either a breadth first search or depth first search to find all `Mem` nodes in the graph. For every `Mem` node, we create a new `Bits` node to stand in for the `init` pin. We add this node to the `Mem` node's input. Finally, we walk up the `Component` hierarchy starting at the `Component` that contains the `Mem` node and ending at the top level `Component`. For every `Component` along this path, we create a new port to connect up the `init` pin.

Targeting SRAM interfaces in this way allows us to decouple the SRAM backend from the user code. If we want to switch to a different SRAM target we only need to switch to a different elaboration time function and do not need to modify the user code.

5.4 Automatic Pipeline Synthesis



(a) Datapath Graph. CPU datapath represented as a directed dataflow graph in Chisel.



(b) Datapath DAG. 5-stage CPU datapath turned into a directed acyclic graph by breaking `Reg` nodes and `TransactionalMem` into read and write ports. The dashed lines represent pipeline boundaries.

Figure 12: 5-stage CPU Datapath. ALU = arithmetic logic unit. D$ = data cache. I$ = instruction cache. TMem = TransactionalMem. `x`.R = read port of node `x`. `x`.W = write port of node `x`

Pipelining is a commonly used microarchitectural technique for improving the performance of a datapath. Hardware designers usually determine up front the desired pipeline depth and then bake the corresponding pipelining logic into their datapath code. This approach clutters an otherwise easy to read datapath with extraneous logic unrelated to original functionality of the datapath, making it difficult to modify the datapath or even change the pipeline depth. A better approach that overcomes these issues is to separate the pipelining logic from the datapath specification [6]. The designer provides only the datapath and pipeline specification. Then during elaboration, Chisel generates the pipelining logic.

Datapath and Pipeline Specification. In our approach, the Chisel datapath code that implements a transaction is maintained separate of the datapath's pipeline specification. The datapath code is viewed as a set of architectural state elements and next-state logic that reads the architectural state and computes the updates. We use `Reg` nodes and `TransactionalMem`s to represent architectural state. For `Reg` nodes, we view the enable signal and the update signal as the `wen` and `wdata` of the Reg. The `Reg` node itself is treated as the `rdata`. A `TransactionalMem` is a Chisel `Component` that wraps a Chisel `Mem` node. `TransactionalMem`s are instantiated with the desired number of *virtual* write ports and *physical* write ports. Each virtual write port has an explicit `wen`, `wdata`, and `waddr` port. The `TransactionalMem` generates the logic to multiplex the virtual write ports over the physical write ports if there are more virtual write ports than physical write ports.

Our pipeline synthesis tools provides the functions `setPipelineComponent` for specifying the datapath to pipeline, and `setPipelineStage` for specifying in what pipeline stage a `Node` or `Component` belongs. These functions allow the user to succinctly describe a pipeline and unclutter the datapath code since the pipeline specification can be maintained independent of the datapath.

Pipeline Generation. During elaboration our pipeline synthesis tool reads the datapath and pipeline specification to determine register placement. We first mark nodes with the stage number that the user has indicated in their pipeline specification. For each node with a stage number we propagate the node's stage numbers to node's producers and consumers in a breadth first search like manner. We only propagate a stage number if (1) the node currently does not have a stage number conflict and (2) all of its children (producer and consumer nodes) are ready to receive a stage number. When receiving a stage number, a node will take a stage number that is either equal to the max stage number of its producers (if the node is receiving a propagation from its producer side) or equal to the min stage number of its consumers (if the node is receiving propagation from its consumer side).

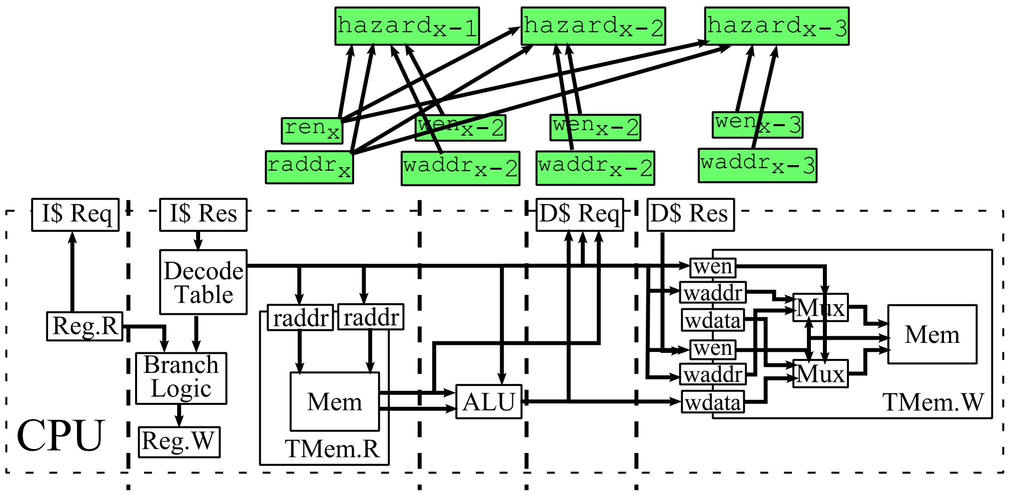
Nodes can be marked with multiple stage numbers since it is possible to reach nodes from different pipeline stages. Our tool adds pipeline registers to the node's inputs (between the node and its producer) or to the node's outputs (between the node and its consumer) to resolve stage number conflicts. Resolving a node's stage number conflict before propagation allows our tool to more deterministically assign stage numbers.

A node is ready for propagation from the producer side if all of its producers have a stage number. Likewise, a node is ready for propagation from the consumer side if all of its consumers have a stage number.

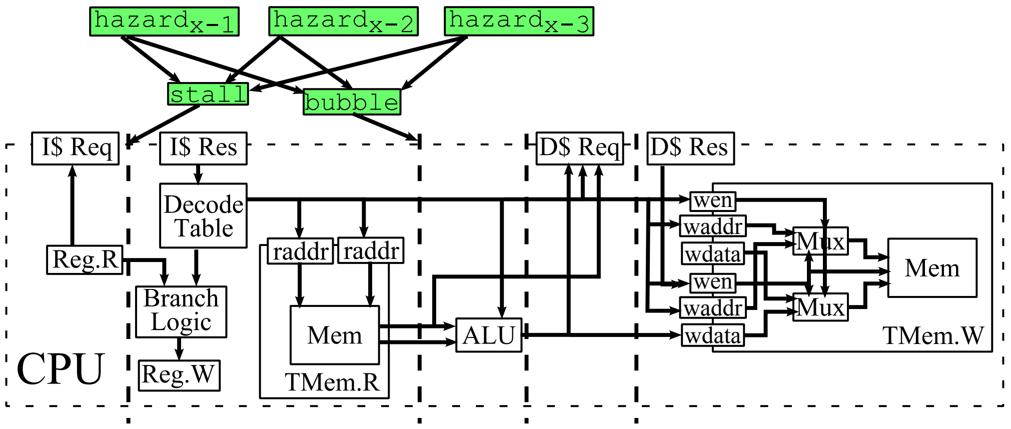
Once all nodes are marked with a stage number, we make one final pass through the Chisel graph to insert pipeline registers between adjacent nodes with different stage numbers.

Hazard Detection. Inserting pipeline registers into the datapath graph introduces pipelining hazards that are not present in the unpipelined datapath. Here we discuss how to handle data hazards which result from a transaction reading a state element (a `Reg` node or a `TransactionalMem`) before an earlier transaction writes to that state element.

We first search through the datapath graph to find all state elements. For every state element, we determine whether or not a hazard exists on it. If there is a hazard on a state element, we add the hazard condition into a list. For `Reg` or `TransactionalMem`, a hazard condition exists if the state element belongs in a stage that precedes the stage of its write enable or write data signal. The actual Boolean hazard condition is `wen && ren` for a `Reg` node and `wen && ren && waddr`



(a) Hazard Detection. All the hazards that the `TransactionalMem` generates in a 5-stage pipeline.



(b) Interlocks. The hazard conditions stall the first and second stage and push bubbles into the third stage.

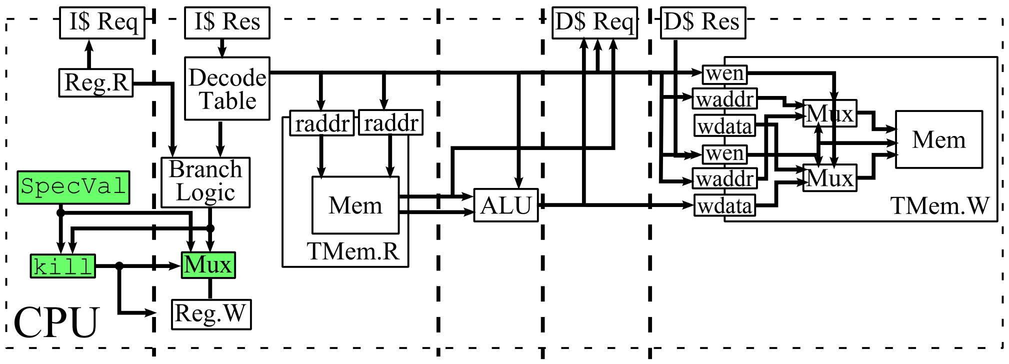
Figure 13: Resolving hazards through interlocks

`== raddr` for a `TransactionalMem`.

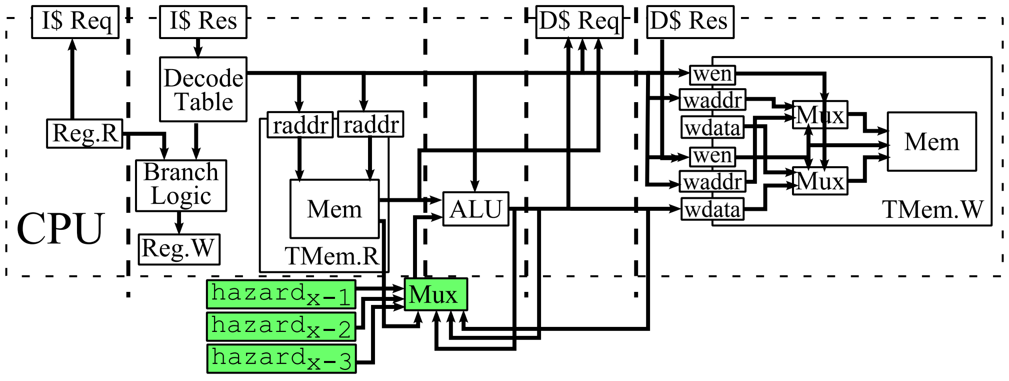
Figure 13a shows the hazard conditions that our tool identifies on the `TransactionalMem`. when analyzing the generated pipeline in Figure 12b. We first search the pipeline for all state elements. In this example, the `Reg` node and the `TransactionalMem` are the only state elements. We then examine the read and write ports of these state elements to determine whether or not there are any hazards. The `Reg` has its write port in the second stage but its read port is in the first stage. So we generate a hazard signal in the first stage. The `TransactionalMem` has its write port in the fifth stage but its read port is in the second stage. We trace backwards from the `wen` in the fifth stage to the second stage to find all the other `wen` signals. For each ($\texttt{wen}_i$, $\texttt{waddr}_i$) pair, we generate a hazard condition $\texttt{wen}_i$ `===` $\texttt{ren}_i$ `&&` $\texttt{waddr}_i$ `=== raddr`.

Hazard Resolution - Interlocks. The easiest way to resolve hazards is through interlocks. This hazard resolution method requires no additional input from the user. Every identified hazard is placed into the stage of the read port that generates the hazard. In Figure 13a, the hazards would be placed into the second stage because the corresponding read port is in the second stage. We then go through every stage and perform an `OR` reduction on all the hazards in that stage as well as all the hazard conditions in the following stages to produce the stall condition for that stage. The second part of interlocking a pipeline is to push bubbles. We go through every stage and generate a `pushBubble` signal if there is a hazard condition in that stage and no following stages have a hazard.

Figure 13b shows the interlock logic that our tool generates to handle the hazards from Figure 13a. Since the `TransactionalMem`'s read port is in the second stage, all the $\texttt{hazard}_i$s go into the second stage. When examining the second stage, the tool would `OR` reduce all the $\texttt{hazard}_i$s to generate a stall condition in the second stage. This stall condition is used to stall the second and first stage. The same hazard condition is then used to generate the `pushBubble` signal. There are no other sources of hazards further down the pipeline so the hazard condition in the second stage is sufficient for pushing a bubble into the second stage. Although not shown, we also generate interlock logic for `Reg` whose read port is in stage one and write port is in stage two. We generate a hazard condition in the first stage that is used to stall only the first stage. A bubble is pushed into stage two if this hazard condition is asserted the second stage's hazard condition is not asserted.



(a) Speculation. Using speculation to resolve hazards on the `Reg` node whose read port is in the first stage and write port is in the second stage.



(b) Bypassing. Bypassing network generated to forward ALU results from the third stage, fourth stage, and fifth stage to the `TransactionalMem`'s second read port.

Figure 14: Resolving hazards through speculation and bypassing

Hazard Resolution - Speculation. Our tool provides the function `speculate` that users can use to specify the stage element whose write value they want to speculate along with the value they want to use for speculation. This hazard resolution option is useful for transactions that produce data that takes on one value with a very high likelihood since we can have dependent transactions guess that value, allowing us to avoid stalls. In a processor pipeline, for example, transactions will usually write `PC+4` to the PC register. To take advantage of this regularity, we can guess that every transaction will write `PC+4` to the `PC` register. During elaboration, we modify the Chisel graph to use the speculation value that the user provides to update the node that the user marks for speculation. The speculation value is pipelined until the actual data value is produced. We compare the speculation value to the actual value. If the values match, we continue as normal. Otherwise, we flush every stage starting at the speculation stage up to but not including the stage where the speculation value is checked.

To see how our speculation tool works, let's examine the pipeline in Figure 14a. Notice that since every transaction writes to the `Reg` node in the second stage while another transaction reads the node in the first stage, the interlock hazard resolution option would cause our pipeline to stall every other cycle. This scenario is a good match for the speculation resolution option since transactions tend to write a very predictable value into that node. We use the `speculate` function to indicate that we are guessing that the `Reg` node will take on `SpecVal`. We mux in `SpecVal` into the `Reg` and then pipeline `SpecVal` once (since the `Reg` is read in the first stage and written in the second stage). We compare `SpecVal` to the actual value to generate a kill signal that is asserted whenever the two signals do not match. The kill signal is used to write the correct value into the `Reg` and we kill stage one.

Hazard Resolution - Bypassing. Transactions that depend on the results of previous transactions can make use of bypass networks to receive these results instead of interlocking until those results are committed. Our tool provides the functions `bypassTo` to annotate a read port for bypassing and `avoidBypassFrom` to annotate that a write path should not be bypassed. During elaboration, the pipeline synthesis tool generates the appropriate bypassing network. For each read port, the tool traces backwards from the corresponding write port to find all `wen`, `wdata`, and `waddr` signals and filters out the write ports from `avoidBypassFrom`. For each (`wen`, `wdata`, `waddr`) tuple that the tool finds, generate a mux at the read port to mux out `wdata` whenever `wen && waddr === raddr` giving priority to (`wen`, `wdata`, `waddr`) tuples from earlier pipeline stages. We then search for that condition in the hazard list and remove it to avoid generating the interlock logic for that condition.

Figure 14b shows an example bypass network. In this example, the user annotates the second read port of the `TransactionalMem` with `bypassTo` function and has told the tool to not bypass from the data cache with the `avoidBypassFrom` function. The tool generates a chain of three muxes since it found three write paths out of the ALU in the third, fourth, and fifth stage.

6 Conclusion and Future Work

We have classified Chisel hardware construction into three different levels. The first level of Chisel hardware construction has users working directly with `Nodes` to construct hardware. At the second level of hardware construction, Chisel `Nodes` are abstracted away into Scala functions. We present examples of two useful hardware facilities `ListLookup` and `Vec` implemented at both levels to highlight the advantages and disadvantages of each level. Finally, the third level of hardware construction has users writing elaboration time functions that the Chisel compiler invokes to build hardware. We present Chisel `when` statements, SRAM backends, and an automatic pipeline synthesis tool as examples of hardware construction at this level.

In future work, we would examine other ways of constructing hardware in Chisel. One possible way to further raise the hardware construction abstraction level would be to implement a process language DSL in Chisel for writing state machines. We would also explore other possible use cases of Chisel's elaboration interface such as generating multithreaded datapaths and FAME [8] transformations.

References

[1] J. Bachrach, K. Asanović, and H. Vo. Chisel manual. Oct. 2012.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 10(2), 1992.

[4] Bluespec Inc, Waltham, MA. *Bluespec(tm) SystemVerilog Reference Guide: Description of the Bluespec SystemVerilog Language and Libraries*, 2004.

[5] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Pschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), Apr. 2012.

[6] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. L. Lu. Automatic pipelining from transactional datapath specifications. In *Design Automation and Test in Europe*, DATE '10.

[7] M. e. a. Odersky. Scala programming language. *http://www.scala-lang.org/*.

[8] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for fame: Fpga architecture model execution. In *Proceedings of the 37th annual international symposium on Computer Architecture*, ISCA '10.